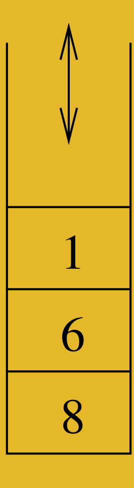
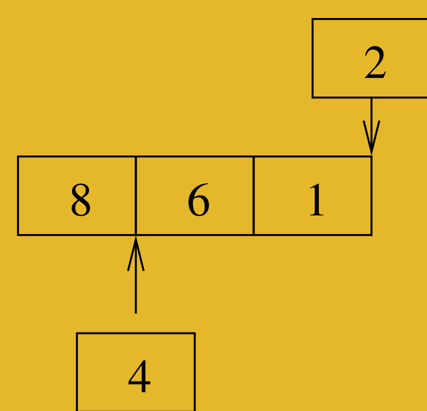
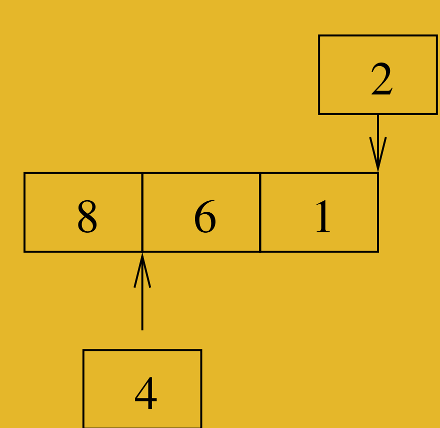
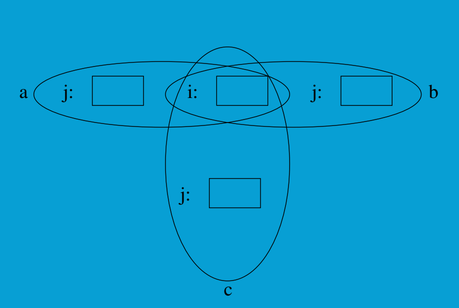


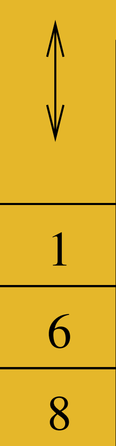


Apropament a les estructures de dades des del programari lliure

Josep Maria Ribó Balust







eines

Apropament a les estructures de dades des del programari lliure

Josep Maria Ribó Balust

Escola Politècnica Superior

Edicions de la Universitat de Lleida
Lleida, 2018

Dades CIP. Servei de Biblioteca i Documentació de la Universitat de Lleida

Ribó i Balust, Josep M. (Josep Maria)

Apropament a les estructures de dades des del programari lliure / Josep Maria Ribó Balust. – Lleida : Edicions de la Universitat de Lleida, 2018. – 346 pàgines ; 29,7cm.
ISBN 978-84-9144-101-4 (en línia)

1. Estructures de dades (Informàtica) 2. Programari lliure
004.422.63

ISBN
978-84-9144-101-4

Maquetació
Josep Maria Ribó Balust

Disseny de portada
cat & cas

Aquest document es distribueix amb una llicència *Creative Commons* Reconeixement-NoComercial-CompartirIgual.

Josep Maria Ribó Balust

Premi Torrens-Ibern 2014

"Apropament a les estructures de dades des del programari lliure"

El Josep Maria Ribó va recollir el premi a la Sala Prat de la Riba de l'Institut d'Estudis Catalans el dia de Sant Jordi de l'any 2014.

El 21 de maig de 2015 va morir després d'una intensa lluita per la vida en la que desbordava el seu amor pels qui teníem la sort de ser al seu costat, pels amics, companys, estudiants i tasca docent, per la humanitat, per la vida.

El Josep Maria desitjava publicar aquest treball amb llicència Creative Commons. Li vaig demanar ajut al seu estimat i admirat company de departament Juan Manuel Gimeno. A ell li agraeixo la seva generosa actitud amb la que va revisar el document abans de ser publicat. Esperem que el present document estigui lliure d'errors. En tot cas, us demanem disculpes pels que pugueu trobar.

Dedicat als estudiants d'Enginyeria Informàtica, a professors i altres interessats en la matèria, desitjo que gaudiu del Josep Maria a través del seu treball.

Silvia Miquel

Índex

Com llegir (i com saltar coses d') aquest llibre **vii**

1 Preliminars **1**

1.1 Tipus . . . 1
1.1.1 Variables . . . 2
1.1.2 Tipus? Quins tipus? . . . 4
1.1.3 Els tipus referència . . . 5
1.1.4 Classes . . . 8
1.1.5 Interfícies . . . 10
1.2 Generalitzacions i herència . . . 17
1.2.1 Relacions de generalització . . . 17
1.2.2 I l'herència . . . 19
1.2.3 Classes abstractes . . . 25
1.2.4 Interfícies i herència . . . 29
1.3 Polimorfisme . . . 30
1.3.1 Subtipus . . . 30
1.3.2 Concepte de polimorfisme . . . 33
1.3.3 Classes abstractes vs. interfícies(*) . . . 36
1.3.4 Conversions explícites cap avall i comprovació de tipus . . . 38
1.4 Genericitat . . . 43
1.4.1 Tipus genèrics . . . 43
1.4.2 Operacions genèriques . . . 49
1.4.3 Subtipus amb genèrics . . . 50
1.4.4 Comodins . . . 53
1.5 Racó lingüístic . . . 58

2 Estructures de dades d'accés seqüencial 61

2.1 Estructures de dades o contenidors . . . 61

2.1.1 Contenidors d'accés seqüencial . . . 63

2.2 Java Collections Framework: una jerarquia de contenidors . . . 64

2.2.1 Les interfícies del Java Collections Framework . . . 65

2.3 Iterables i iteradors . . . 67

2.3.1 La interfície `Iterable<T>` . . . 69

2.3.2 La interfície `Iterator<T>` . . . 70

2.4 Col·leccions . . . 75

2.4.1 Operacions de la interfície `Collection<T>` . . . 77

2.5 Llistes. Especificació . . . 80

2.5.1 Un exemple d'ús de llistes . . . 82

2.5.2 Iteradors sobre llistes: `Iterator<T>` i `ListIterator<T>` . . . 83

2.6 Implementant les llistes . . . 90

2.6.1 ArrayList: elements consecutius de la llista en posicions consecutives de memòria . . . 91

2.6.2 LinkedList: elements consecutius de la llista en posicions no consecutives de memòria . . . 92

2.6.3 Comparació entre LinkedList i ArrayList . . . 93

2.7 Implementant les llistes amb enllaços: LinkedList<T> . . . 95

2.7.1 AbstractCollection<T>(*) . . . 97

2.7.2 AbstractList<T>(*) . . . 102

2.7.3 AbstractSequentialList<T>(*) . . . 105

2.7.4 LinkedList<T> . . . 106

2.8 Racó lingüístic . . . 126

3 Arbres 129

3.1 Els arbres a vista d'ocell . . . 129

3.1.1 Arbres generals . . . 131

3.1.2 Arbres n-aris . . . 131

3.1.3 Arbres binaris . . . 132

3.1.4 Notació d'arbres . . . 132

3.1.5 Propietats dels arbres binaris . . . 134

3.1.6 Recorreguts d'arbres . . . 136

3.2 Especificant els arbres binaris . . . 137

3.2.1 Especificació dels arbres binaris . . . 137

3.2.2 Exemple: recorregut d'un arbre binari en preordre . . . 142

3.2.3 Exemple: recorregut d'un arbre binari per nivells . . . 144

3.3 Implementant els arbres binaris 146
3.3.1 Dues propostes per a la implementació dels arbres binaris 146
3.4 Implementant els arbres binaris amb un vector 146
3.5 Implementant els arbres binaris amb enllaços 149
3.5.1 Implementació bàsica 150
3.5.2 Arbres binaris enllaçats amb punters al pare 160
3.5.3 Amb enfilaments en inordre (*) 167
3.6 Racó lingüístic 171

4 Estructures de dades d'accés directe: les taules 173
4.1 Què és una taula? 173
4.1.1 Com podem modelitzar una taula? 175
4.1.2 Consideracions addicionals sobre les taules: 175
4.2 Especificant les taules 175
4.2.1 La interfície `Map<K, V>` 176
4.2.2 La interfície `SortedMap<K, V>` 181
4.3 Implementant les taules 183
4.3.1 Classe `AbstractMap<K, V>` 183
4.3.2 Formes bàsiques d'implementar taules 184
4.3.3 Vector ordenat i cerca dicotòmica 185
4.3.4 Funcions injectives 186
4.4 Racó lingüístic 189

5 Taules implementades amb funcions de dispersió 191
5.1 Les taules de dispersió a vista d'ocell 191
5.2 Funcions de dispersió (*) 194
5.2.1 Descomposició de h 195
5.2.2 Funcions de dispersió: h_1 196
5.2.3 Funcions de dispersió: h_2 197
5.2.4 Una funció de dispersió habitual 198
5.3 Estratègies de dispersió 199
5.4 Estratègies de dispersió. Dispersió tancada(*) 200
5.4.1 Algorismes d'inserció, consulta i eliminació 200
5.4.2 Algorisme d'inserció 202
5.4.3 Algorisme de consulta 203
5.4.4 Algorisme d'eliminació 204

5.4.5 Famílies de funcions de dispersió . . . 204
5.4.6 Família de dispersió lineal . . . 205
5.4.7 Família de dispersió quadràtica . . . 207
5.4.8 Família de dispersió doble . . . 209
5.4.9 Dispersió Robin Hood: una variant de la dispersió tancada . . . 210
5.5 Estratègies de dispersió. Dispersió oberta(*) . . . 211
5.5.1 Implementació de la dispersió oberta . . . 211
5.6 Cost de les taules de dispersió (*) . . . 213
5.6.1 Definicions prèvies . . . 213
5.6.2 Cost de la dispersió tancada . . . 213
5.6.3 Cost de la dispersió oberta . . . 216
5.6.4 Comparació de costos entre les dues estratègies de dispersió . . . 216
5.6.5 Quan és adient implementar una taula mitjançant alguna estratègia de dispersió? . . . 217
5.7 Implementació de les taules de dispersió al Java: `HashMap<K, V>` . . . 217
5.7.1 `HashMap<K, V>` . . . 218
5.7.2 Les constructores . . . 219
5.7.3 `HashMap.Entry<K, V>` . . . 220
5.7.4 La funció de dispersió: `hashCode()` . . . 221
5.7.5 La funció de restricció d'un valor de dispersió a la longitud d'un vector: `indexFor()` . . . 226
5.7.6 `get(key)` . . . 227
5.7.7 `put(key, value)` . . . 228
5.7.8 Més operacions . . . 228
5.8 Racó lingüístic . . . 229

6 Taules implementades amb arbres 231
6.1 Arbres per fer taules . . . 231
6.2 Arbres implementats com a ABC . . . 232
6.2.1 Idea dels ABC . . . 232
6.2.2 Consulta, inserció i eliminació de claus en un ABC . . . 234
6.2.3 Cost dels algorismes d'inserció, eliminació i consulta en un ABC . . . 237
6.2.4 Obtenció de parelles (clau,valor) ordenades per clau . . . 239
6.2.5 Una proposta d'implementació de taules amb ABC . . . 240
6.3 Taules implementades com a arbres B . . . 257
6.3.1 Arbres m-aris de cerca . . . 258
6.3.2 Arbres B. Definició . . . 259

6.3.3 Arbres B. Algorisme d'inserció 260
6.3.4 Arbres B. Algorisme d'eliminació 266
6.3.5 Ús dels arbres B 278
6.4 Racó lingüístic 278

7 Índexs 279
7.1 Estructures de dades en memòria persistent 279
7.1.1 Bases de dades (relacionals) 280
7.1.2 Emmagatzemament físic de les taules relacionals 282
7.2 Índexs en taules relacionals 287
7.3 Implementació d'índexs 290
7.3.1 Arbres B i B+ 290
7.3.2 Taules de dispersió persistent 294
7.3.3 Criteri d'ús de taules de dispersió persistents i arbres B+ com a índexs d'una taula relacional 296
7.4 Una proposta d'implementació d'índexs amb taules de dispersió persistents (*) 296
7.4.1 La interfície Packable 304
7.4.2 Representació de la taula de dispersió persistent 309
7.5 Racó lingüístic 311

A Excepcions 315
A.1 Mecanisme de gestió d'excepcions 315
A.2 Tipus d'excepcions 318
A.2.1 Excepcions que l'aplicació ha de tractar (*checked exceptions*) 318
A.2.2 Errors 319
A.2.3 Excepcions d'execució (*runtime exceptions*) 319
A.3 Exemple 320
A.4 `finally` 321

B Classes aniuades, locals i anònimes 323
B.1 Classes aniuades 323
B.1.1 Membres estàtics. Recordatori 324
B.1.2 Classes aniuades estàtiques 326
B.1.3 Classes aniuades no estàtiques 327
B.1.4 Quan usem classes aniuades de cada tipus 329

Com llegir (i com saltar coses d') aquest llibre

Presentació

La informàtica, o, més precisament, els programes, gestionen informació, una gran quantitat d'informació. Aquesta informació s'ha d'emmagatzemar a la memòria de l'ordinador o en mitjans d'emmagatzemament persistents (com ara un disc) de manera que s'hi pugui accedir tan eficientment com sigui possible. Les estructures de dades miren de resoldre aquest problema. O sigui, cerquen com estructurar la informació de manera que l'accés a ella sigui eficient.

D'aquesta manera, podem pensar en una estructura de dades com una col·lecció de dades emmagatzemades amb una certa gràcia en un mitjà (típicament, la memòria de l'ordinador o el disc) de manera que sigui senzill d'accedir-hi. Però com accedim a una estructura de dades? Doncs, depenent de què signifiquin i per a què volem fer servir les dades d'aquella estructura. Si l'estructura representa la cua d'una pastisseria, en tot moment necessitarem accedir al *primer* de la cua i necessitarem afegir els nous clients al *final* de la cua. Si representa un diccionari (una col·lecció de paraules amb la seva definició, sinònims i exemples d'ús), no ens interessa per a res saber quina paraula és la *primera*. En canvi, voldrem trobar ràpidament tota la informació associada a una paraula determinada. Si l'estructura representa una xarxa de carreteres, la informació que li demanarem serà: *quines són les ciutats veïnes de Lleida a la xarxa de carreteres*?

Veiem, doncs, que apareixen diferents estructures de dades segons l'ús que vulguem fer d'aquelles dades.

En aquest llibre estudiarem algunes d'aquestes estructures de dades: les *llistes* (i les seves parentes, les *cues* i les *piles*), els *arbres* i les *taules*. Totes aquestes estructures de dades viuran a la memòria principal. Pràcticament no direm res de les estructures de dades en memòria persistent (fitxers i bases de dades).

Aquest material serveix de base per a l'assignatura d'*Estructures de Dades* impartida al tercer quadrimestre del *grau en Enginyeria Informàtica* que s'ofereix a l'Escola Politècnica Superior de la Universitat de Lleida.

Les característiques més rellevants d'aquest llibre són:

- L'intent d'usar una terminologia tècnica estandarditzada i normativa. Ho expliquem a la secció següent.

- La utilització del llenguatge Java com a llenguatge de programació per implementar les estructures de dades.
- Més específicament, l'ús com a base de la biblioteca d'estructures de dades estàndard de Java: el *Java Collections Framework*, del qual s'expliquen, no només els seus principis d'ús, sinó també com s'ha implementat aquesta biblioteca.
- La presentació del codi que proposa el projecte de programari lliure *OpenJDK* per implementar el *Java Collection Framework* (només la part que es relaciona amb les estructures de dades que presenta el llibre).
- La distribució amb una llicència *copyleft* (que permet, en particular, el desenvolupament d'obres derivades).
- L'adaptació completa a l'assignatura esmentada més amunt. Per aquest motiu, pot ser una eina valuosa per als seus estudiants i també per a estudiants universitaris d'enginyeria que cursin assignatures similars.

La llengua i la terminologia tècnica

Escriure o llegir un text informàtic en una llengua que no sigui l'anglesa pot dur fàcilment a una sensació de contaminació, d'impuresa. Una miríada de termes anglesos comencen a surar inevitablement en el text. A vegades, i malgrat l'esforç d'estandardització que fa Termcat, no és senzill trobar un terme català normatiu o estàndard per referir-nos a un concepte informàtic que sí que té una expressió en anglès. Com a petit tast, us poso alguns exemples que han anat sortint en la redacció d'aquest llibre: *array, bucket, hash, map, table, cast, early binding, balanced tree*, ... Quina és la traducció normativa o, almenys, estàndard d'aquests i de molts altres termes? L'he buscada al diccionari de l'Institut d'Estudis Catalans [IEC], al Centre de terminologia de la llengua catalana (Termcat [TermC]) i al Diccionari de termes informàtics [CM94] elaborat pel Servei de llengües i terminologia de la UPC. En aquells casos en què no he pogut trobar una traducció adient, he proposat com a traducció un terme normatiu (i.e., que apareix al Diccionari de l'Institut d'Estudis Catalans) i que resulti natural per denotar aquell concepte (per exemple, perquè s'hagi usat a bastament per la comunitat informàtica per referir-s'hi). Aquests casos de proposta de termes no estandarditzats els he documentat al final de cada capítol en una secció titulada *Racó lingüístic*, acompanyada per aquesta icona.

Malgrat aquest esforç, el llibre conté encara una miríada de termes anglesos surant-hi. Això es deu a que és un llibre de programació amb molt de codi i el codi està escrit i, si em permeteu que sigui per un cop, dogmàtic, *s'ha d'escriure* en anglès. Per què? us preguntareu, per què hem de doblegar, un cop més, el genoll davant de l'imperi anglosaxó? Doncs per dos motius que, de fet, són el mateix:

- Tot el codi que conté aquest llibre està llicenciat com a programari lliure. Quasi per definició, el programari lliure té vocació d'universalitat, d'arribar a tothom, a qualsevol racó de món per a que pugui ser estudiat, redistribuït i millorat (de fet, el codi es pot consultar a `http://sedna.udl.cat/ed`. I la manera més efectiva d'aconseguir això és escriure'l en anglès. Per això, encara que no sigui estrictament obligatori, la immensa majoria de programari lliure s'escriu en anglès.

- Com ja hem dit, el llibre usa com a codi base per implementar moltes estructures de dades que s'hi expliquen, el de la biblioteca estàndard de contenidors del Java (*Java Collection Framework*). Però aquesta biblioteca és un projecte de programari lliure i el seu codi (que es copia i s'explica al llibre) està escrit en anglès.

Així doncs, permetrem que l'anglès del codi contamini, ni que sigui una mica, el text.

Continguts

Les característiques més significatives del contingut del llibre (i de l'assignatura en què es basa) són aquestes:

- Arrenca amb els conceptes de programació orientada a objectes (POO) que es necessitaran en la descripció de les estructures de dades que constitueixen el curs (vegeu el capítol 1).

  Essencialment, aquests conceptes són: tipus, classe, interfície, objecte, herència, polimorfisme i genericitat.

- Dedica la resta dels capítols a presentar diferents famílies d'estructures de dades. Aquesta presentació sempre té la mateixa estructura: *especificació* del tipus amb què definim l'estructura de dades i de les seves diverses *implementacions*.

- Amb més detall, les estructures de dades que presenta són:

  - *Llistes* i altres estructures d'accés seqüencial (vegeu el capítol 2).
  - *Arbres*, fent especial esment als binaris i a les iteracions sobre els seus elements (vegeu el capítol 3).
  - *Taules*. Aquesta és l'estructura de dades estrella del curs. El llibre dedica quatre capítols a descriure les taules:
    - Al capítol 4 s'especifiquen les taules i se'n plantegen algunes implementacions trivials.
    - El capítol 5 es dedica a presentar la implementació de taules basada en funcions de dispersió.
    - Al capítol 6 es presenten dues implementacions arborescents de taules: els arbres binaris de cerca i els arbres B.
    - Finalment, es destina el capítol 7 a presentar un dels usos més interessants de les taules: els índexs de les bases de dades. Aquest capítol dóna suport a l'assignatura Bases de Dades.

- No es presenten formalment els grafs com a part de la teoria del curs (ni tampoc en aquest llibre). Els estudiants coneixen formalment els grafs gràcies a l'assignatura Matemàtica Discreta (que es cursa també al tercer quadrimestre).

  Els aspectes derivats de la implementació dels grafs s'inclouen com a part del projecte de programació que constitueix la part pràctica de l'assignatura Estructures de Dades.

Llenguatge de programació

Tant el llibre com l'assignatura usen Java com a llenguatge de programació. Penso que és un llenguatge d'ampli ús a la comunitat d'enginyeria de programari i que és convenient que els estudiants en tinguin un bon coneixement, la qual cosa s'assolirà si s'usa en diverses assignatures, com, de fet, passa al grau en Enginyeria Informàtica de la UdL.

En general, el lleguatge Java té un disseny que m'agrada i ofereix bona part de les característiques necessàries per descriure les estructures de dades que són objecte d'estudi al curs. Al mateix temps, conté un API molt ampli, que inclou el *Java Collection Framework*, una biblioteca d'estructures de dades prou potent per als propòsits de l'assignatura.

Finalment, la implementació de referència de la *Java Standard Edition* és un projecte de programari lliure: *OpenJDK*. Sóc un fidel de l'església del programari lliure.

El Java Collections Framework

En aquest llibre (i a l'assignatura) es presenta la jerarquia i l'especificació de les estructures de dades que ofereix el *Java Collections Framework (JCF)* i que estan integrades a l'API estàndard de Java.

No només en presenta l'especificació i la jerarquia del JCF sinó també la implementació d'una part d'aquesta jerarquia. Aprofitant que *OpenJDK* és un projecte de programari lliure, es pot presentar als estudiants directament el seu codi. D'aquesta manera, els estudiants tenen ocasió de conèixer un codi eficient, professional i molt depurat, un codi que no és de joguina sinó que ha estat desenvolupat per a una biblioteca d'estructures de dades real i molt usada. A través d'aquest codi, els estudiants poden també introduir-se en tècniques d'enginyeria de programari que els seran presentades en assignatures posteriors.

En algunes ocasions s'ha simplificat una mica el codi per raons pedagògiques. En altres casos s'ha desenvolupat alguna estructura de dades que no forma part del JCF (e.g., `BSTMap`). En aquests casos, s'ha intentat inserir aquesta estructura a la jerarquia proporcionada per la JCF i seguir el disseny d'altres estructures similars.

Icones i convencions

El llibre conté una col·lecció d'icones que poden ajudar a localitzar més ràpidament els punts d'interès. Són aquestes:

- **Atenció!!!** Requadre que sintetitza i fixa conceptes, els quals se solen introduir en negreta. Els conceptes d'aquests requadres acompanyats d'aquesta icona són especialment importants i haurien de quedar ben clars.

- **Pensador.** El llibre planteja una qüestió que invita a pensar sobre alguna cosa que s'ha explicat. Sovint la resposta a aquella qüestió segueix la qüestió mateixa. L'objectiu és que el lector s'aturi un moment quan trobi aquesta icona i miri de contestar la pregunta abans de continuar llegint.

- **Ampliació.** Requadre que proposa algunes precisions o ampliacions al que s'ha comentat prèviament. Sol estar en lletra petita i la lectura es pot ometre sense perdre el fil del contingut. Únicament per al consum estudiants inquiets.

- **Exemple.** Descripció d'un exemple. Els exemples estan numerats i acompanyats d'aquesta icona. Cada exemple acaba amb tres requadres negres:

  ■ ■ ■

- **Codi incorrecte.** Codi que té algun problema. La icona ajuda que quedi clar que aquell codi té alguna dificultat sense necessitat de llegir el text.

- **Secció optativa.** Secció (i totes les seves subseccions) pensada per donar més material als estudiants inquiets. Aquest material no és necessari de cap manera per aprovar l'assignatura ni tan sols per treure una molt bona nota. Es poden saltar sense comprometre la comprensió del text. Les seccions optatives es reconeixen també perquè el títol va seguit de **(*)**.

- **Racó lingüístic.** Es troba al final de cada capítol i conté una proposta de terme tècnic per cada concepte introduït en aquell capítol per al qual no hem trobat un terme normatiu o estàndard en català.

Alguns parèntesis al text comencen amb **i.e.**, altres ho fan amb **e.g.**:

- **e.g.**: *Per exemple*. Del llatí, *exempli gratia*.
- **i.e.**: *Això és*. Del llatí, *id est*.

Text

He intentat que el text sigui clar. M'he ajudat de molts requadres, icones, comentaris, figures, codi i exemples. El resultat de tot plegat ha estat un text llarg. Potser prolix? Potser sí. Vosaltres ho heu de decidir, si és que el voleu llegir. En tot cas, jo espero que doni informació clara i que sigui de fàcil lectura.

Us agrairé, de debò, qualsevol comentari i/o correcció que em vulgueu fer: `josepma@eps.udl.cat`.

Llicència

Aquest document es distribueix amb una llicència *Creative Commons* Reconeixement-NoComercial-CompartirIgual.

Capítol 1

Preliminars

No podem començar un curs d'estructures de dades basat en el paradigma de la *programació orientada a objectes* (a partir d'ara, POO) sense assegurar-nos que entenem bé els fonaments d'aquest paradigma. I n'hi ha dos, d'aquests fonaments, que són del tot cabdals: *l'abstracció-encapsulament* i el *polimorfisme*. En aquest capítol assumirem que el primer d'aquests elements ja és conegut (reviseu, per exemple, [GG11]) i ens centrarem en el segon: el polimorfisme. És clar que, per acabar parlant de polimorfisme, ens caldrà revisar primer el concepte de *tipus* (vegeu la secció 1.1) i després haurem de presentar els de *generalització* i *herència* (vegeu la secció 1.2). Havent-nos menjat aquests dos entrants, tindrem (desitjablement) l'estómac preparat per al plat principal: *el polimorfisme* (vegeu la secció 1.3), que no és més que la conclusió natural de la generalització i l'herència.

En acabat tot això, encara ens quedarà una altre aspecte *preliminar* per tractar: *els tipus genèrics*. Aquest tema és especialment interessant perquè les estructures de dades que estudiarem a partir del capítol 2 usen sempre tipus genèrics per referir-se als tipus dels seus elements components: una `List<E>` voldrà dir una llista amb elements components de tipus genèric `E`. Però de tot això ja en parlarem a la secció 1.4.

Som-hi?

1.1 Tipus

Un concepte absolutament essencial a la POO és el de tipus i és el primer que ens cal discutir en el nostre camí cap al polimorfisme. El tipus d'una variable ens indica la *mena* d'aquesta variable. Què podem fer amb ella. Quines operacions li podem aplicar. Hem dit variable? Potser ens cal recordar primer, amb una mica de precisió, què és una variable.

1.1.1 Variables

En un llenguatge de POO com ara Java, una **variable** és un identificador que es refereix a la localització de memòria on es guarda una unitat d'informació que és gestionada per un determinat programa.

Exemple 1.1:

Llistat 1.1: Exemples de variables

```
1 int i = 10;
2
3 String st = "agamenon";
4
5 String[] ast = new String[10];
```

`i`, `st` i `ast` són variables.

■ ■ ■

Una variable té dos conceptes molt importants associats:

- El seu *valor*

  Això és, la informació que hi ha guardada dins de la localització de memòria a la qual es refereix la variable.

  A l'exemple anterior, el valor de la variable `i` és 10. De la mateixa manera, podríem pensar que el valor de la variable `st` és "agamenon", però això no és així. Per saber quin és el valor de la variable `st` cal avançar una mica més... fins arribar a la secció 1.1.3. De moment, deixem-ho aparcat.

- El seu *tipus*

  La mena d'informacions que pot emmagatzemar una variable.

  A l'exemple anterior, la variable `i` pot emmagatzemar valors enters (que notem com a `int`), la variable `st` pot emmagatzemar cadenes de caràcters (`String`s) i la variable `ast` pot emmagatzemar vectors[1] de `String`s.

  Tota variable que apareix en un programa ha de tenir un tipus associat. Per això la declarem.

[1]Vegeu el racó lingüístic del capítol.

Declarar una variable vol dir triar un identificador per referir-s'hi i associar-li un tipus.
(A aquest tipus se'l coneix com a *tipus-compilació*, però això és una altra història i no l'explicarem ara sinó a 1.3.2).

La instrucció `String st;` declara la variable identificada per `st` amb el tipus `String`.

Això només és un tastet dels *tipus*. El *tipus* és un concepte fonamental en programació i més encara en POO. Per això li dedicarem al llarg de tota aquesta secció 1.1 l'atenció que es mereix.

Famílies de variables

Hem vist alguns exemples de variables. Vegem-ne més.

Exemple 1.2:

Llistat 1.2: `PersonFromSomewhere`

```
1  public class PersonFromSomewhere {
2      private String nif;
3      private String firstName;
4      private String lastName;
5      private int age;
6      private static String birthPlace;
7
8      public static void setBirthPlace(String somewhere) {
9          PersonFromSomewhere.birthPlace = somewhere;
10     }
11
12     public String getCompleteName() {
13         String name;
14
15         name = this.firstName
16                    .concat(" ")
17                    .concat(this.lastName);
18         return name;
19     }
20 }
```

Aquesta classe modelitza les persones que són d'una localitat determinada. Si la localitat és Lleida, aleshores totes les persones instància[2] d'aquesta classe són de Lleida.

■ ■ ■

Evidentment, aquesta classe gestiona informació i, per fer-ho, usa variables de diferents tipus. Vegem-los:

[2]Recordem que una instància d'una classe és un objecte que té els atributs de la classe i es poden aplicar sobre ell les operacions de la classe

- *Atributs:* `nif`, `firstName`, `lastName`, `age` i `birthPlace`.

  Els atributs són variables que apareixen a la definició d'una classe i indiquen *com es representen internament a la memòria els objectes d'aquella classe*. Per exemple, els objectes que són instància de la classe `PersonFromSomewhere` es representen en termes de tres `Strings` (`firstName`, `lastName` i `nif`); un `int` per l'edat (`age`) i un altre `String` per la localitat (`birthPlace`).

  Per cert, hi ha dos tipus d'atributs:

  – *Atributs d'instància*
    Cada objecte instància de la classe rep una instància diferent d'aquell atribut. És el cas de `nif`, `firstName`, `lastName` i `age`. Aquests són els atributs més habituals.

  – *Atributs de classe*
    Tots els objectes instància de la classe reben una única instància compartida d'aquests atributs. És el cas de `birthPlace`. Es noten com a `static` a la definició de la classe i ens hi referim prefixant-los amb el nom de la classe (i no amb la variable amb què identifiquem l'objecte de la classe, com en el cas dels atributs instància). Recordeu la línia 9 del codi anterior:

```
1        PersonFromSomewhere.birthPlace = somewhere;
```

    Així passa també amb les operacions *estàtiques*:

```
1        PersonFromSomewhere.setBirthPlace("Lleida");
```

  La figura 1.1 (més avall) presenta com s'emmagatzemen els atributs de dos objectes de la classe `PersonFromSomewhere`.

- *Paràmetres:* `somewhere` (línia 8).

- *Variables locals:* `name` (línia 13).

1.1.2 Tipus? Quins tipus?

Hem dit que tota variable és d'un *tipus* que indica de *quina mena és aquella variable*. Intuïtivament això sembla clar: una variable de tipus `int` podrà contenir enters com ara -1 o 5. Una de tipus `String` que conté cadenes de caràcters com ara "lleida" o "joan". Però això només és el començament. Ara ens cal precisar-ho tot una mica. En particular, presentarem els diferents tipus que defineix el llenguatge Java i, aleshores, ens centrarem en un d'aquells tipus: els *tipus referència*, perquè aquests tipus són, precisament, els que ens permeten treballar polimòrficament.

Molts llenguatges de POO i, particularment, el Java, defineixen dues menes de tipus:

- *Els tipus primitius*

  Els tipus primitius són els més simples: `boolean, int, short, long, double, float, char, byte`.

  Es caracteritzen perquè estan predefinits pel llenguatge i per una altra cosa molt important:

El valor d'una variable de tipus primitiu és directament un valor d'aquell tipus.

O sigui, si tenim una declaració tal com:

```
1 int i = 10;
```

El valor guardat a la localització de memòria corresponent a `i` és 10, en la representació binària que calgui (segurament, en complement a 2).

- *Els tipus referència*

  Quins altres tipus coneixeu fora dels predefinits? Precisament aquells que apareixen en el context de la POO: els tipus definits per classes. O sigui: la classe `PersonFromSomewhere` defineix un tipus. I, per això, podem fer declaracions de l'estil:

```
1 PersonFromSomewhere p;
```

  Aquests són els tipus interessants per arribar al polimorfisme. Per tant, dedicarem un apartat al seu estudi.

1.1.3 Els tipus referència

Els tipus referència tenen una característica molt diferent de la dels tipus primitius:

El valor d'una variable de tipus referència és una referència a una localització de memòria on es troba un objecte d'aquell tipus.

En particular, *dues variables de tipus referència poden referir-se a un mateix objecte* (si totes dues tenen com a valor la mateixa localització de memòria).

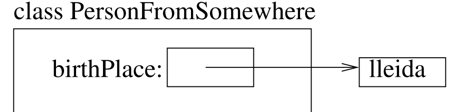
O sigui, la localització de memòria identificada per la variable no conté directament l'objecte sinó que conté l'adreça de memòria on es troba l'objecte.

La variable esdevé doncs, una *referència a l'objecte*.

Exemple 1.3:

Considerem el codi següent, il·lustrat a la figura 1.1:

```
1 int i = 10;
2 PersonFromSomewhere.setBirthPlace("lleida");
3 PersonFromSomewhere p1 =
4         new PersonFromSomewhere("paul", "bowles", 30);
5 PersonFromSomewhere p2 = p1;
```



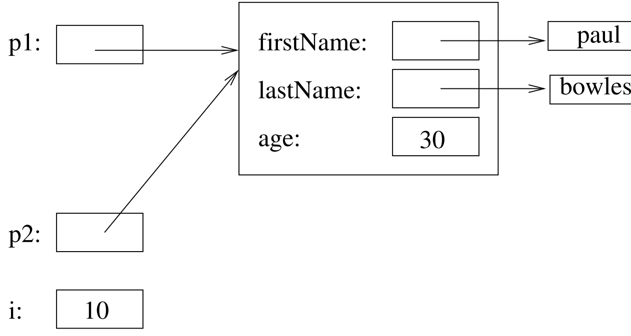


Figura 1.1: Representació de la classe `PersonFromSomewhere`

Comentaris:

- `i` és una variable d'un tipus primitiu i conté directament el valor d'aquell tipus (10).
- `p1` i `p2` són variables del tipus referència `PersonFromSomewhere`. El seu valor és una referència a la localització de memòria on s'emmagatzema l'únic objecte existent d'aquesta classe (i compartit per les dues variables).
- L'atribut `birthPlace` és un atribut de classe. Per tant només n'hi ha un de compartit per tots els objectes d'aquella classe.
- Els atributs `birthPlace`, `firstName` i `lastName` són variables d'un tipus referència (`String`). Per tant, el seu valor (com el de `p1` i `p2`) també són referències a la localització de memòria on es troben els objectes `String`.
- Per contra, l'atribut `age` és una variable d'un tipus primitiu (`int`). Per tant, el seu valor és directament un valor enter.

■ ■ ■

A Java hi ha tres tipus referència diferents:

- *Classes*
- *Interfícies*
- *Vectors*

D'aquests tres tipus referència només ens ocuparem de les *classes* i *interfícies* que són les que el polimorfisme usa com a base.

Les classes i els interfícies tenen una característica comuna:

Classes i **interfícies** defineixen un tipus referència en termes de *les operacions que es poden aplicar sobre les variables d'aquells tipus*.

(A vegades es diu que classes i interfícies defineixen un tipus referència *en termes del seu comportament*, atès que les operacions defineixen el comportament d'una instància d'aquell tipus).

Aquestes operacions han d'estar especificades mitjançant una **postcondició** i, si cal, una **precondició**.

A la llista d'operacions que defineix el tipus referència, conjuntament amb la seva especificació se les anomena **especificació del tipus** o **contracte del tipus**.

Al conjunt d'instàncies (objectes) que ofereixen les operacions definides per un tipus referència se les anomena **instàncies del tipus** o **objectes del tipus**.

Especificació d'un tipus

La biblioteca de classes i interfícies de Java proporciona bons exemples de com s'han d'especificar els tipus.

Exemple 1.4: Especificació de dues operacions de la classe `String`

```
int compareTo(String anotherString)
```

Compara dues cadenes de caràcters de manera lexicogràfica. La comparació es basa en el valor *Unicode* de cada caràcter de les cadenes. La seqüència de caràcters representada per aquest objecte `String` és comparada lexicogràficament amb la seqüència de caràcters representada per l'objecte `String` del paràmetre. El resultat és un enter negatiu si aquest objecte `String` precedeix lexicogràficament el del paràmetre. El resultat és un enter positiu si aquest objecte `String` segueix lexicogràficament el del paràmetre. El resultat és zero si els dos objectes `String` són iguals. `compareTo(obj)` retorna 0 exactament quan `equals(obj)` retorna `true`.

- *Paràmetres:* anotherString és l'objecte `String` amb què es fa la comparació.
- *Retorna:* el valor 0 si el paràmetre `String` és igual a aquest `String`; un valor menor que 0 si aquest `String` és lexicogràficament menor que el paràmetre i un valor més gran que 0 si aquest `String` és lexicogràficament més gran que el paràmetre.

```
public String concat(String str)
```

Concatena l'objecte `String` del paràmetre al final d'aquest `String`.
Si la longitud de l'objecte `String` del paràmetre és 0, aleshores es retorna l'objecte receptor (`this`). En cas contrari, es crea un objecte `String` nou que representa una seqüència de caràcters que és la concatenació de la seqüència de caràcters representada per aquest objecte `String` i la seqüència de caràcters representada pel paràmetre.

- *Paràmetres:*

  `str`: L'objecte `String` que es concatena al final d'aquest objecte `String`.

- *Retorna:*

  Un objecte `String` que representa la concatenació dels caràcters d'aquest objecte seguits pels caràcters del paràmetre.

Aquestes especificacions han estat obtingudes de **??**.

■ ■ ■

Els tipus que ens interessen per al treball amb polimorfisme queden definits quan establim *les operacions (especificades) que podem aplicar sobre les instàncies d'aquell tipus*. Classes i interfícies fan això i, per tant, **classes i interfícies defineixen tipus**. Però les classes fan una mica més. Vegem-ho.

1.1.4 Classes

Una classe és un tipus, juntament amb la seva implementació.

Per tant, una classe:

- Defineix un tipus en termes d'una llista d'operacions (especificades).
- Implementa aquell tipus.

Com a conseqüència, **definir una classe** voldrà dir:

1. **Definir el tipus** (que serà implementat per la classe) en termes d'una llista d'operacions que es podran aplicar a les seves instàncies.

   Considerem, com a exemple, la classe `Person`, la qual voldrem que exhibeixi la llista d'operacions següents:

   - `void setNif(String nif)`
   - `String getNif()`
   - `void setFirstName(String fname)`
   - `void setLastName(String lname)`

- `String getCompleteName()`
- `void setBirthDate(Date d)`
- `int getAge()`
- Altres operacions que hagi d'oferir el tipus `Person`.

A més a més d'enumerar les seves operacions, és important d'especificar-les tal com hem vist a la secció anterior.

Sabríeu especificar les operacions de la classe `Person` seguint l'exemple que hem presentat abans per a les operacions de la classe `String`?

2. **Implementar el tipus**

Implementar el tipus vol dir:

(a) *Representar la classe*, o sigui, *descriure com s'emmagatzema una instància d'aquella classe a la memòria.*

Això es fa mitjançant atributs.

```
1 public class Person {
2     private String nif;
3     private String firstName;
4     private String lastName;
5     private Date birthdate;
6
7     ...
8 }
```

Aquests atributs ens indiquen que una instància de la classe `Person` s'emmagatzemarà internament a la memòria (diem *es representarà*) mitjançant tres `Strings` (`firstName`, `lastName` i `nif`) i un objecte `Date` anomenat (`birthdate`).

> Al conjunt d'atributs d'una classe se l'anomena la **representació d'aquella classe**.
>
> També podem dir que **els atributs d'una classe constitueixen la seva estructura** (mentre que **les operacions constitueixen el seu comportament**).
>
> Els atributs d'una classe són habitualment privats.

(b) *Implementar les operacions definides al tipus.*

```
1 public class Person {
2     ...
3
4     public String getCompleteName() {
5         String name;
6
```

```
 7          name = this.firstName
 8                      .concat("␣")
 9                      .concat(this.lastName);
10          return name;
11      }
12
13      public void setFirstName(String name){
14          this.firstName = name;
15      }
16
17      ...
18  }
```

Aquesta implementació es fa d'acord amb l'especificació de les operacions i amb la representació triada per a la classe.

És important adonar-se que els atributs de la classe i la implementació de les seves operacions no constitueixen pròpiament el tipus sinó la seva implementació.

Hi ha vegades que aquesta implementació és parcial. O sigui, alguna o totes les operacions del tipus definit per la classe no estan implementades per aquesta. Aquest tipus de classes s'anomenen **classes abstractes**. Us pot semblar estrany que una classe es quedi a mitges en la implementació del tipus que defineix, però tot té una explicació. L'única cosa és que, per tal d'entendre aquesta explicació, necessitem introduir què és l'herència. I això ho farem a la secció 1.2, tot just després d'haver aprofundit una mica més en què són les interfícies.

1.1.5 Interfícies

Definició d'una interfície

Les **interfícies** són un tipus referència que conté la declaració de les operacions que aquell tipus ofereix.
A aquesta declaració de les operacions se l'anomena **capcelera** o bé **signatura**.

La interfície declara aquestes operacions. **No les implementa**.

Les interfícies són tipus. Per tant, es poden usar en tots els contextos en què cal un tipus. En particular, es poden declarar variables del tipus d'una interfície. Sobre aquelles variables podrem aplicar qualsevol de les operacions declarades per la interfície.

A més a més de la declaració d'operacions, una interfície pot contenir *declaració de constants*. Aquestes constants són, per defecte, `public`, `static` i `final`.

La definició d'una interfície no pot contenir atributs que no siguin constants.

Exemple 1.5:

Pensem en un reproductor MP3, o bé en un altre de DVD, o de CD o un sofisticat reproductor de tot tipus de medis audiovisuals (*media player*). Tots ells hauran d'oferir operacions per reproduir (`play`), fer una pausa (`pause`), avançar fins a la propera pista (`nextTrack`)... Dit d'una altra manera, *tots ells hauran d'implementar el tipus* `Player` que podem definir en termes d'una interfície, de la manera següent:

```
1 public interface Player {
2     public void play();
3     public void nextTrack();
4     public void previousTrack();
5     public void stop();
6     public void pause();
7     public int getnNbTracks();
8     public int getCurrentTrack();
9 }
```

Notem que aquesta interfície defineix un tipus que és completament independent del reproductor específic amb què treballem. En particular, és independent del tipus de mitjà audiovisual que es reprodueix (CD, DVD, MP3...).

■ ■ ■

Exemple 1.6:

Considerem el tipus `Vehicle`. Independentment de si es tracta d'una bicicleta, una moto, un cotxe, un camió o una llanxa, hem de poder aplicar sobre un objecte d'aquell tipus operacions per conèixer el seu propietari (`getOwner()`), la seva velocitat màxima (`getMaxSpeed()`) i la seva marca (`getBrand()`). Qualsevol vehicle haurà d'oferir aquestes operacions. Hem definit, doncs, un tipus del qual hauran de ser instància tots els objectes que siguin vehicles.

```
1 public interface Vehicle {
2     public String getOwner();
3     public int getMaxSpeed();
4     public String getBrand();
5 }
```

■ ■ ■

La definició d'un tipus requereix únicament donar un nom per al tipus i enumerar les declaracions de les operacions que ofereix (juntament amb l'especificació d'aquestes operacions). Des d'aquest punt de vista, és interessant notar que les interfícies són el que podríem anomenar *tipus purs* perquè no fan res més que això: defineixen un nom de tipus i la llista de les capceleres (o signatures, recordeu) de les operacions que ofereix. Una classe,

a més a més d'això, implementa aquelles operacions. Per això diem que una classe és *la implementació d'un tipus*, mentre que una interfície és únicament *la definició d'un tipus*.

Implementació d'una interfície

Atès que les interfícies són tipus, podem implementar-les mitjançant classes.

> Es diu que la classe `C` implementa la interfície `I` si `C` implementa totes les operacions declarades a `I`.
>
> Un objecte instància d'una classe que implementa una interfície té el tipus definit per aquella interfície.

Exemple 1.7: Implementació d'una interfície

La classe `CDPlayer` és una classe que ens permetrà reproduir CD. Les instàncies d'aquesta classe seran reproductors específics de CD. Resulta clar que aquesta classe ha d'implementar la interfície `Player` que hem presentat a la secció 1.1.5.

```
1 public class CDPlayer implements Player {
2     ...
3 }
```

Com que la classe `CDPlayer` implementa la interfície `Player`, aquesta haurà d'implementar (això és, proveir codi per a) totes les operacions declarades a aquesta interfície:

```
1 public class CDPlayer implements Player {
2     public void play() { ... }
3     public void nextTrack() { ... }
4     public void previousTrack() { ... }
5     public void stop() { ... }
6     public void pause() { ... }
7     public int getnNbTracks() { ... }
8     public int getCurrentTrack() { ... }
9 }
```

I, per fer això, cal donar una representació a aquesta classe mitjançant atributs. No és el nostre objectiu ara completar tota la implementació d'una classe complexa com `CDPlayer`, però en donem alguns detalls per mostrar-ne la flaire.

```
1 public class CDPlayer implements Player {
2     private int currentTrack;
3     private Track[] tracks;
4     private int nbTracks;
5     private Position header;
6     private String title;
```

```
7
8       public CDPlayer(Track[] tracks, String title) {
9           this.tracks = tracks;
10          this.title = title;
11          this.nbTracks = tracks.length;
12          this.currentTrack = 0;
13          this.header = new Position();
14      }
15
16      public void nextTrack() {
17          this.currentTrack =
18                  (this.currentTrack + 1) % this.nbTracks;
19      }
20
21      public int getNbTracks() {
22          return this.nbTracks;
23      }
24
25      public int getCurrentTrack() {
26          return this.currentTrack;
27      }
28
29      public String getCDTitle() {
30          return this.title;
31      }
32
33      public void previousTrack() { ... }
34      public void play() { ... }
35      public void stop() { ... }
36      public void pause() { ... }
37  }
```

Comentaris:

- A una classe que implementa una interfície li cal implementar totes les operacions de la interfície. En aquest cas, no hem entrat en el detall d'algunes d'elles (e.g.: `play()`, `stop()`...).
- Una classe que implementa una interfície pot oferir altres operacions diferents de les de la interfície implementada (e.g.: `getCDTitle()` o l'operació constructora).
- Notem també que la instrucció `this.tracks = tracks;` (vegeu la línia 9) assigna a l'atribut `this.tracks` el vector `tracks` que es passa per paràmetre. És important assenyalar que, com a conseqüència d'aquesta instrucció, `this.tracks` i `tracks` són dues referències al mateix vector. No se'n crea un de nou!!!

■ ■ ■

Exemple 1.8: La interfície `Identifiable`

Definim la interfície `Identifiable`. Les instàncies d'aquesta interfície es caracteritzaran perquè podran ser identificades mitjançant un `String`. Per exemple, un objecte *cotxe* pot ser una instància de la interfície `Identifiable` perquè es pot identificar amb la seva matrícula (que és un `String`). Igualment ho pot ser un objecte *persona* (identificable amb un `nif`) o un objecte *llibre*, que s'identifica amb un ISBN.

(Per cert, recordem que una interfície és un tipus; per tant, igualment podem dir que *un objecte és instància d'una interfície* com que *un objecte és instància d'un tipus*).

Vegem la definició de la interfície `Identifiable`:

```
1 public interface Identifiable {
2     public String getId();
3 }
```

Amb l'especificació següent per a l'operació `getId`:

> `public String getId();`
>
> Retorna l'identificador d'aquest objecte.

I ara podem recuperar la classe `Person`, els objectes de la qual es poden identificar amb el `nif`. Per tant, podem fer que aquesta classe implementi `Identifiable`:

```
1  public class Person implements Identifiable {
2      private String nif;
3      private String firstName;
4      private String lastName;
5
6      public String getId(){
7          return this.nif;
8      }
9
10     ...
11 }
```

■ ■ ■

> **Una classe pot implementar diverses interfícies.**
>
> Un objecte instància d'una classe que implementa diverses interfícies té el tipus definit per cadascuna de les interfícies implementades.

Exemple 1.9: Classe que implementa diverses interfícies

```
1  public class MiniCooper implements Vehicle, Identifiable {
2      private String owner;
3      private String plate;
4
5      public MiniCooper(String plate, String owner){
6          this.plate = plate;
7          this.owner = owner;
8      }
9
10     public String getOwner(){
11         return this.owner;
12     }
13
14     public int getMaxSpeed(){
15         return 160;
16     }
17
18     public String getBrand(){
19         return "BMW";
20     }
21
22     public String getId(){
23         return this.plate;
24     }
25
26     public void setOwner(String owner){
27         this.owner = owner;
28     }
29 }
```

Comentaris:

- La classe `MiniCooper` implementa `Vehicle` i `Identifiable`; per tant, ha d'implementar totes les operacions declarades en aquestes dues interfícies: `getOwner`, `getMaxSpeed` i `getBrand`, per un costat i `getId`, per l'altre.
- A més a més d'aquestes operacions, no hi ha cap dificultat en el fet que la classe `MiniCooper` defineixi altres operacions (com `setOwner` o l'operació constructora).
- A més a més, la classe `MiniCooper` defineix una representació per als objectes d'aquesta classe en termes dels atributs `owner` i `plate`. Aquests atributs són usats per tal d'implementar les operacions.
- Un objecte de la classe `MiniCooper` **és instància de tres tipus diferents**: `Vehicle`, `Identifiable` i `MiniCooper`.

Això, en particular, vol dir que una instància de `MiniCooper` es podrà posar allà on s'esperi una instància de qualsevol d'aquests tres tipus. A l'exemple següent ho aclarim.

■ ■ ■

Exemple 1.10: Els tres tipus d'una instància de `MiniCooper`

Continuant amb el darrer exemple, considerem les tres operacions `f1`, `f2` i `f3` següents:

```
1 void f1(Identifiable obj){...}
2
3 void f2(Vehicle v) {...}
4
5 void f3(MiniCooper mc) {...}
```

I considerem un objecte `car` instància de la classe `MiniCooper`. Aquest objecte `car` el podrem usar com a paràmetre de crides a totes tres operacions perquè `car` és alhora una instància d'`Identifiable`, de `Vehicle` i de `MiniCooper`:

```
1 MiniCooper car = new MiniCooper("1234ABC","Pepet popet");
2
3 f1(car);
4 f2(car);
5 f3(car);
```

Però compte!!!, `f1` només podrà aplicar a `car` l'operació `getId()`. `f2` només li podrà aplicar `getOwner()`, `getBrand()` i `getMaxSpeed()`. Finalment, `f3` li podrà aplicar totes les operacions definides a la classe `MiniCooper`:

```
1  void f1(Identifiable obj){
2      String id = obj.getId();          //OK!!
3      String owner = obj.getOwner();    //MALAMENT!!!
4  }
5
6  void f2(Vehicle v) {
7      String owner = v.getOwner();      //OK!!
8      String id = v.getId();            //MALAMENT!!!
9  }
10
11 void f3 (MiniCooper mc) {
12     String owner = mc.getOwner();     //OK!!
13     String id = mc.getId();           //OK!!
14 }
```

■ ■ ■

Herència múltiple i interfícies

L'herència múltiple permet que una classe sigui subclasse de dues o més superclasses. El Java no proporciona aquesta possibilitat. Per contra, a la secció anterior hem dit que una classe pot implementar diverses interfícies. Aquest fet es pot usar com a alternativa a l'herència múltiple.

Exemple 1.11: Herència múltiple amb interfícies

Si existeixen les interfícies `MarriedPerson` i `Employee`, podem definir `MarriedEmployee` de la manera següent:

```
1 public class MarriedEmployee implements MarriedPerson,Employee {
2       ...
3 }
```

■ ■ ■

1.2 Generalitzacions i herència

Caminem a bon pas vers el polimorfisme. Passada la primera etapa, en què hem reflexionat sobre els tipus i, en especial, sobre els tipus que són importants per treballar polimòrficament (*classes* i *interfícies*), ara ens cal introduir la relació de *generalització entre classes* (o entre interfícies) i *l'herència* que se'n deriva de manera natural. Aquests dos conceptes (generalització i herència) són utilíssims en el marc del disseny de programari orientat a objectes: no només permeten el polimorfisme sinó que fan també possible l'extensió de biblioteques de classes i el treball amb *frameworks*. Aquestes darreres aplicacions no les trobareu en aquests apunts sinó que apareixeran més endavant en els vostres estudis. Ara, però, parlem de generalització i d'herència.

1.2.1 Relacions de generalització

Les classes modelitzen conceptes (vehicles, cadenes de caràcters, dates, persones, treballadors...). I hi ha conceptes que són més generals que d'altres. Per exemple, en una empresa de disseny de programari, la classe `Employee` modelitza un concepte més general que el que modelitza la classe `Engineer` i també que el que modelitza la classe `ProjectManager`. Efectivament, a l'empresa hi ha diferents tipus d'empleats: uns d'ells són els enginyers, que s'encarreguen dels diferents aspectes del desenvolupament d'un projecte; uns altres són els caps de projecte (`ProjectManager`), que tenen una funció de coordinació i de gestió del projecte. També podríem trobar altres tipus específics d'empleats, com ara, els caps de departament (`DeptDirector`) o els administratius (`AdminEmployee`). Tots aquests tipus de treballadors tenen una cosa en comú: *són empleats de l'empresa*, per això modelitzen conceptes *més específics, més concrets* o *menys generals* que `Employee`.

Us sembla intuïtiu que `Employee` sigui més general que `Engineer`? Encara més: en una altra situació en què tinguem diverses classes, *com podrem saber si una d'elles és més general que una altra?* Alguna idea? Preneu-vos un moment per pensar-hi...

Hi ha una regla d'or que funciona gairebé sempre (noteu el *gairebé*) i és aquesta:

La regla és-un:

Siguin `C` i `S` dues classes. Si la frase "*un* `S` *és un* `C`" té sentit, *probablement* la classe `C` és més general que la classe `S`.

D'aquesta regla se'n diu *la regla és-un*.

Aplicant aquesta regla als empleats de l'empresa de programari veiem que té sentit dir: *un enginyer és un treballador* (i, per tant, `Engineer` és candidata a ser subclasse d'`Employee`); *un cap de projecte és un treballador* (i, per tant, `ProjectManager` és candidata a ser subclasse d'`Employee`)...

I també té sentit dir: *un cotxe és un vehicle* (i, aplicant la regla, `Vehicle` és més general que `Car`). O bé, *un home és una persona* i, per tant `Person` és més general que `Man` i també que `Woman`.

Notem que a l'inrevés no funciona: Un `Employee` no té per què ser un `Engineer`. També pot ser un `ProjectManager` o un `DepartmentDir`. Un `Vehicle` no té per què ser un `Car`, també pot ser un camió (e.g., `Lorry`) o una furgoneta (e.g., `Van`).

I ara, fixeu-vos en un altre detall: només hi ha un tipus d'enginyer en una empresa de programari? No. Hi ha enginyers especialitzats en l'adquisició dels requeriments de l'aplicació que cal desenvolupar (anomenem-los `ReqEngineers`); n'hi ha d'altres especialitzats en dissenyar-la (anomenem-los `DesignEngineers`) i, almenys, n'hi haurà d'altres que en faran les proves (`TestEngineers`).

Quina relació us sembla que hi haurà entre aquestes classes i `Engineer`?

És clar que sí: totes elles són més específiques (i.e., menys generals) que `Engineer`. Noteu que, aplicant la regla *és-un* té sentit dir: *un* `TestEngineer` *és un* `Engineer`. Per tant, tenim que `Engineer` és més específica que `Employee` però més general que `TestEngineer`.

S'anomena **superclasse** a una classe més general que una altra. S'anomena **subclasse** la classe específica.

En el Java una classe `S` que és subclasse de `C` es defineix de la manera següent, usant la paraula clau **extends**:

```
1 class S extends C {
2       ...
3 }
```

Una col·lecció de classes lligades per relacions de generalització (possiblement amb més de dos nivells) constitueix una **jerarquia de classes**.

Així tenim que `Engineer` és una subclasse de `Employee` i una superclasse de `TestEngineer` i també:

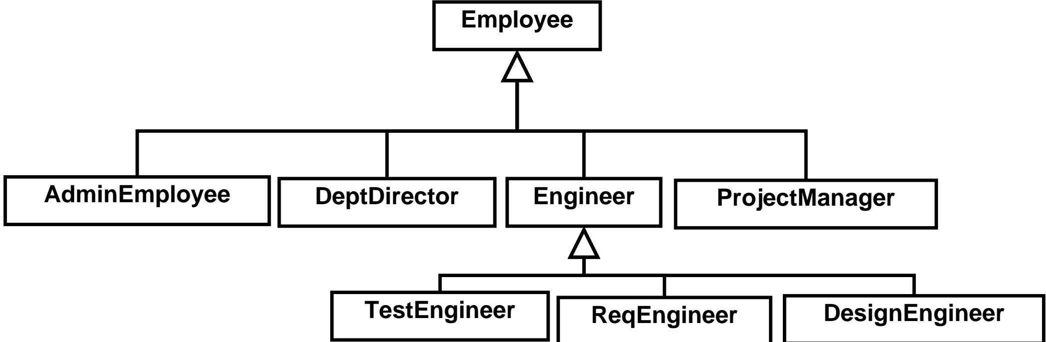


Figura 1.2: Representació gràfica de les relacions de generalització

```
1 public class Engineer extends Employee {
2       ...
3 }
4
5 public class TestEngineer extends Engineer {
6       ...
7 }
```

Les relacions de generalització les representem gràficament tal com s'il·lustra en la figura 1.2.

I ara ens podem preguntar: quina relació hi haurà entre l'estructura i el comportament d'una superclasse (per exemple, `Employee`) i una subclasse seva (per exemple, `Engineer`)? La resposta a aquesta pregunta ens porta a parlar de l'herència. Som-hi.

1.2.2 I l'herència

Hem dit que un `Engineer` és un `Employee`. Per tant, sembla raonable que si una instància de la classe `Employee` té els atributs `name`, `nif` i `email`, una instància d'`Engineer` disposi també d'aquests tres atributs. També sembla raonable (i pel mateix motiu) que les operacions que es poden aplicar sobre un `Employee` es puguin aplicar també sobre un `Engineer`. O sigui, si `eng` és una instància d'`Engineer` hauríem de poder fer `eng.getNif(), eng.setName("Paul")`... Diem que:

> Una subclasse hereta els atributs i les operacions definides a la seva superclasse.
>
> Per tant, no cal que els defineixi novament.

A més a més, una subclasse pot definir atributs nous. Per exemple, tot `Engineer` de l'empresa de programari està assignat a algun projecte que està desenvolupant l'empresa. Això no és

cert per a altres tipus d'empleats, com els administratius, que no estan assignats a cap projecte. Igualment, una subclasse pot definir operacions noves que s'afegiran a les que ja hereta de la superclasse. Per exemple, `Engineer` pot afegir les operacions `setProject(Project p)` (que assignarà l'enginyer a un determinat projecte) i `getProject()` (que retornarà el projecte al qual està assignat l'enginyer).

En ocasions serà necessari que la subclasse modifiqui la implementació d'una operació definida a la seva superclasse. Per exemple, la superclasse `Employee` disposarà d'una operació `toString` que retornarà un `String` amb les propietats de la instància d'`Employee` sobre la qual es cridi.

```
1 String stEmp = emp.toString();
```

`stEmp` pot referir-se a un `String` tal com:

"Empleat: Joan Cases. Nif: 34543322. Correu: jcases@softsol.com"

Però si aquest empleat fos un `Engineer`, no n'hi hauria prou amb això. La cadena retornada per `toString()` hauria de contenir, a més a més, el projecte al qual està assignat el Joan Cases i la informació que aquest empleat és un enginyer. O sigui:

"Empleat: Joan Cases. Nif: 34543322. Correu: jcases@softsol.com. Enginyer. Projecte: TerminalPuntVenda."

La figura 1.3 mostra com es representen gràficament les relacions d'herència que hem explicat i l'exemple següent, com s'implementen en el Java.

Exemple 1.12: Jerarquia de classes `Employee`

En aquest exemple, mostrem el codi d'unes quantes classes de la jerarquia arrelada a `Employee`. Aquesta jerarquia apareix gràficament a la figura 1.3. Comencem amb la mateixa classe `Employee`:

Llistat 1.3: Classe `Employee`

```
1  public class Employee {
2      protected String name;
3      protected String nif;
4      protected String email;
5
6      public Employee() {
7          this.name = "";
8          this.nif = "";
9          this.email = "";
10     }
11
```

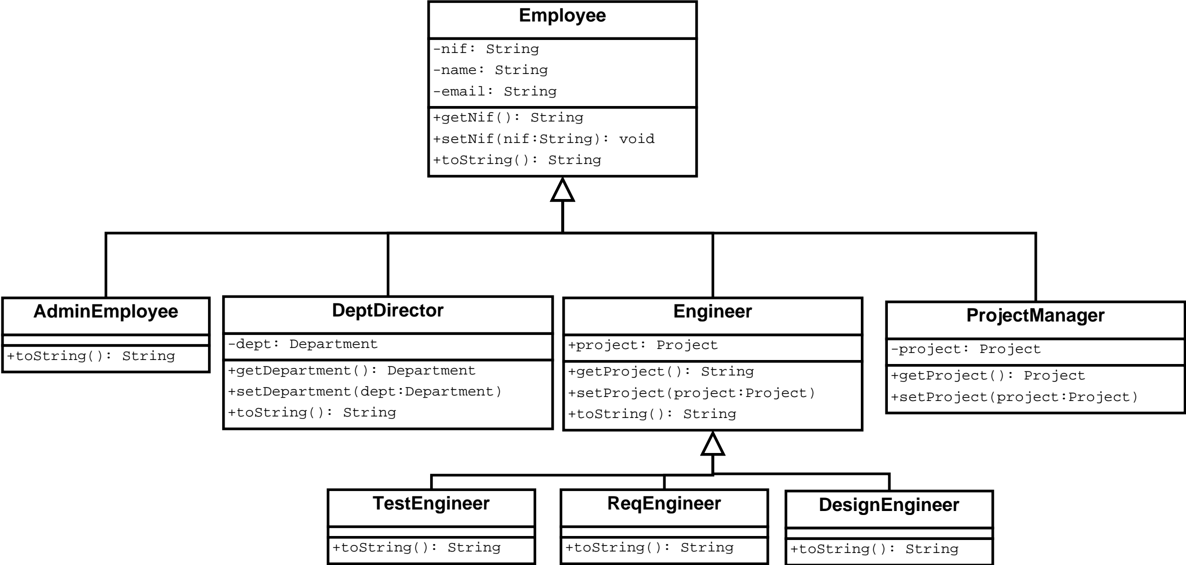


Figura 1.3: Jerarquia de classes `Employee`

```
12      public Employee(String name, String nif, String email) {
13          this.name = name;
14          this.nif = nif;
15          this.email = email;
16      }
17
18      public void setName(String name) {
19          this.name = name;
20      }
21
22      public String getName() {
23          return this.name;
24      }
25
26      public void setNif(String nif) {
27          this.nif = nif;
28      }
29
30      public String getNif() {
31          return this.nif;
32      }
33
34      public void setEmail(String email) {
35          this.email = email;
36      }
37
```

```
38     public String getEmail() {
39         return this.email;
40     }
41
42     public String toString() {
43         return "Empleat:␣" + this.name + "␣Nif:␣"
44                 + this.nif + "␣Email:␣" + this.email;
45     }
46 }
```

Presentem ara dues subclasses: `Engineer` i `AdminEmployee`:

Llistat 1.4: Classes `Engineer` i `AdminEmployee`

```
1  public class Engineer extends Employee {
2      protected Project project;
3
4      public Engineer(){
5          this.project = null;
6      }
7
8      public Engineer(String name, String nif,
9                      String email, Project p) {
10         super(name,nif,email);
11         this.project = p;
12     }
13
14     public void setProject(Project project) {
15         this.project = project;
16     }
17
18     public Project getProject() {
19         return this.project;
20     }
21
22     @Override
23     public String toString() {
24         return super.toString() + "␣Engineer␣"
25                 + "Project:␣" + this.project.getName();
26     }
27 }
28
29
30 public class AdminEmployee extends Employee {
31     public AdminEmployee() {}
32
33     public AdminEmployee(String name, String nif, String email) {
34         super(name,nif,email);
35     }
```

```
36
37     public String toString() {
38         return super.toString() + " Administrative employee";
39     }
40 }
```

Comentaris:

- Ambdues classes hereten els atributs d'`Employee` (`name, nif, email`). Aquests atributs no cal que es redefineixin.
- Ambdues classes hereten les operacions d'`Employee`: `get/setName(), get/setEmail(), get/setNif()`. Les constructores d'Employee no s'hereten. Per això totes dues subclasses d'`Employee` han de definir les seves.
- Per cert, les constructores de les subclasses `AdminEmployee` i `Engineer` criden a les constructores de `Employee`.
  - Si no es diu el contrari, criden implícitament a la constructora per defecte (i.e., la constructora sense paràmetres) de la superclasse. Això passa amb `public Engineer()` i `public AdminEmployee()` (vegeu les línies 4 i 31).
  - Si es vol cridar una constructora amb paràmetres, cal fer-ho explícitament mitjançant la instrucció `super`. Això és el que es fa a les línies 10 i 34 en què es crida explícitament a `Employee(name,nif,email)` mitjançant la instrucció `super(name,nif,email)`.

    Aquesta instrucció ha de ser la primera de l'operació constructora.

> O sigui: la creació d'un objecte d'una subclasse implica, en particular, la creació d'un objecte de la seva superclasse. Per això, quan es crea un objecte d'una subclasse cal cridar un constructor de la superclasse.
>
> Per defecte, es crida el constructor sense paràmetres de la superclasse (anomenat *constructor per defecte*).
>
> Es pot cridar un altre constructor de la superclasse mitjançant la instrucció `super(paràmetres-del-constructor-de-la-superclasse)`.
>
> Si no s'esmenta de manera explícita aquesta crida a un constructor de la superclasse, el compilador del Java insereix automàticament i implícita la crida `super()` a les operacions constructores de les subclasses.

- `Engineer` defineix un atribut (`project`) i dues operacions noves: `get/setProject()`.

  `AdminEmployee` no defineix cap atribut ni cap operació nova.
- Tant `Engineer` com `AdminEmployee` *redefineixen* l'operació `toString()`. Totes dues reaprofiten `Employee.toString()` per construir un `String` amb els atributs comuns (això s'aconsegueix amb la crida `super.toString()` de les línies 24 i 38.

  Notem que les operacions redefinides es marquen amb l'anotació `@Override`.

Etiqueta	Classe	Paquet	Subclasse	Resta
`public`	sí	sí	sí	sí
`protected`	sí	sí	sí	no
no etiqueta	sí	sí	no	no
`private`	sí	no	no	no

Figura 1.4: Etiquetes modificadores d'atributs i operacions

- Els atributs i les operacions privades s'hereten però no s'hi pot accedir directament des de la subclasse.

  Per això els atributs de la classe `Employee` han estat declarats com a `protected`.

> Els atributs i operacions declarats en una classe com a `protected` són visibles a les operacions de les seves subclasses.

■ ■ ■

A l'exemple anterior hem introduït l'etiqueta modificadora `protected` abans d'un atribut (o d'una operació). A més a més, ja coneixem les etiquetes `public`, `private` i l'absència d'etiqueta. El quadre de la figura 1.4 mostra la visibilitat de cada atribut o operació segons l'etiqueta modificadora amb què estigui definit. La informació d'aquesta figura l'hem de llegir de la manera següent:

Un atribut o operació declarat a una classe com a `protected` *serà visible a les operacions d'aquella classe, de qualsevol altra classe del mateix paquet i de qualsevol subclasse d'aquella classe; però no serà visible a les operacions d'altres classes que no siguin les anteriors*.

La classe `Object`

La classe `Object` és la classe que és al capdamunt de la jerarquia de classes del Java. Qualsevol altra classe del Java és subclasse directament o indirecta d'`Object`. Als exemples que hem mostrat fins ara `Employee` i `Person` són subclasses directes de `Object` i `Engineer` ho és indirecta.

La classe `Object` ofereix una col·lecció d'operacions que són heretades per qualsevol altra classe del Java. Entre aquestes operacions en destaquem dues:

- `obj.toString()`, retorna un `String` que conté el nom de la classe i l'adreça de memòria on està emmagatzemat l'objecte `obj`[3].

- `obj.equals(Object obj2)`, retorna cert si `obj` i `obj2` són referències al mateix objecte (i.e., si tots dos contenen la mateixa adreça) i fals en qualsevol altre cas.

[3]En realitat, no és exactament l'adreça de memòria sinó el resultat d'aplicar una funció de dispersió a aquella adreça. Al capítol 5 parlem més de les funcions de dispersió.

És una bona política redefinir aquestes dues operacions cada cop que es defineixi una nova classe. Als exemples d'aquest capítol ho hem fet amb `toString()` i més endavant ho farem amb `equals(obj)`.

Herència: resum final

El quadre següent resumeix el més important del que hem dit respecte de l'herència.

- Una subclasse **hereta els atributs i les operacions** de la seva superclasse (llevat de les operacions constructores que no s'hereten).

  A l'exemple: els atributs `name`, `nif` i `email`. Les operacions: `get/setName()` i `get/setNif()...`.

- Una subclasse pot **definir nous atributs o operacions** que s'afegiran a les heretades de la superclasse.

  A l'exemple, `Engineer` defineix el nou atribut `project` i les noves operacions `get/setProject()`.

- Una subclasse pot **redefinir algunes operacions** heretades de la superclasse.

  A l'exemple, tant `Engineer` com `AdminEmployee` redefineixen `toString()`.

- Una operació d'una subclasse pot cridar a l'operació que redefineix de la superclasse mitjançant la paraula clau `super`.

  Igualment, una operació constructora d'una subclasse pot cridar una operació constructora de la seva superclasse usant `super`.

- Els atributs i les operacions privades s'hereten però no s'hi pot accedir directament des de la subclasse. Per accedir directament a atributs i operacions des de les subclasses, cal declarar-los com a `protected`.

- Cada classe té **exactament** una superclasse (llevat de la classe `Object`, que no en té cap). Si en la definició d'una classe no s'indica quina serà la seva superclasse, aquesta és `Object`. Per tant, `Object` és, en darrera instància, ancestre de totes les classes definides en el Java.

- `Object` no té superclasse. És l'única excepció a la regla anterior que cada classe té una superclasse.

1.2.3 Classes abstractes

Tots els treballadors tenen una *categoria laboral* i guanyen un *salari* (si no fos per això, segurament, no treballarien) que depèn d'aquesta categoria. Un director de departament cobra un salari diferent que un enginyer o que un administratiu.

Així doncs, té sentit que la classe `Employee` incorpori les operacions `int getSalary()` i `String getCategory()`. I que totes les seves subclasses (e.g., `Engineer`, `ProjectDirector`) les heretin.

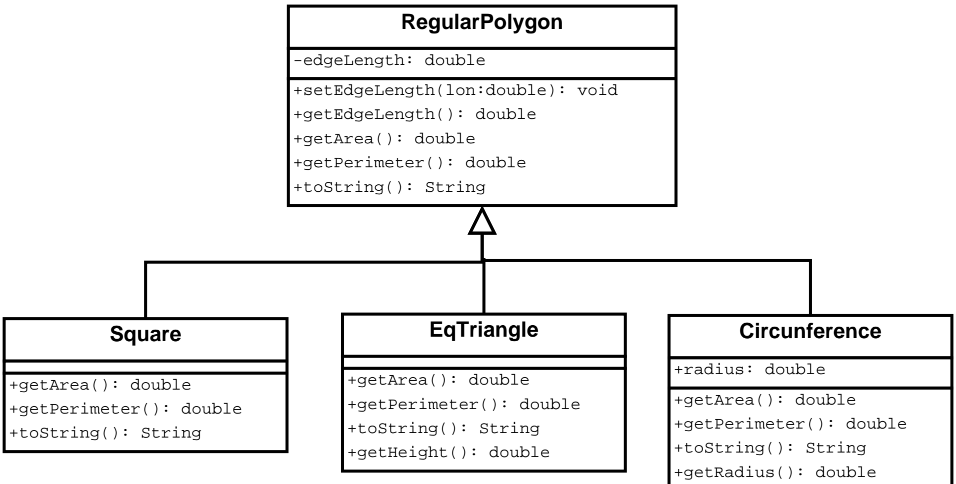


Figura 1.5: Jerarquia de classes `RegularPolygon`

Hi podria haver treballadors que no fossin instància de cap subclasse d'`Employee`? O sigui, treballadors que no fossin ni `Engineer` ni `AdministrationEmployee` ni `ProjectManager` ni...? Bé... Si ho acceptem, un d'aquells treballadors hauria de ser directament instància d'`Employee`. Per crear-lo faríem una cosa semblant a:

```
1 Employee misteryEmp = new Employee("34343422", "Joe Mistery", "Design");
```

Fins aquí tot bé, però quin serà el resultat de les crides `misteryEmp.getSalary()` i `misteryEmp.getCategory()`? Dit d'una altra manera, quina categoria té un treballador que no té cap categoria? Com calculem el sou d'un treballador, la categoria del qual no sabem?

Posem encara un altre exemple:

Exemple 1.13: Polígons regulars

Considerem la jerarquia de classes *polígons regulars* dibuixada a la figura 1.5.

És clar que tot polígon té una àrea i un perímetre, per tant podem afegir les operacions `getArea()` i `getPerimeter()` a la classe `RegularPolygon`. I, com que tenim la classe `RegularPolygon` amb aquestes operacions, podem fer:

```
1 RegularPolygon pol = new RegularPolygon(23.0);
2                            // 23.0 = mida d'un costat del poligon
3 double area = pol.getArea();
4 double perimeter = pol.getPerimeter();
```

Com podem calcular l'àrea o el perímetre d'un polígon el tipus del qual desconeixem?

■ ■ ■

Tots dos exemples ens permeten arribar a les mateixes conclusions:

1. *No podem implementar les operacions* `getSalary()` *ni* `getCategory()` *per a la classe* `Employee` ni tampoc les operacions `getArea()` ni `getPerimeter()` per a la classe `RegularPolygon`.

2. Si no podem implementar aquelles operacions tampoc no podrem cridar-les sobre un objecte de classe `Employee` o `RegularPolygon`. Per tant, *no té sentit que creem objectes de les classes* `Employee` *o* `RegularPolygon`.

Classe abstracta. Mètode abstracte

- Una classe abstracta és una classe que no es pot instanciar (i.e., no es poden crear objectes d'aquella classe).

- Notarem una classe com a abstracta, precedint-la per l'etiqueta `abstract`:

```
1 public abstract class Employee {
2      ...
3 }
```

- Una classe abstracta defineix sovint una *implementació parcial* (e.g., alguns atributs i/o algunes operacions implementades). Aquesta implementació parcial és compartida i completada per les seves subclasses.

- Una classe abstracta pot tenir mètodes no implementats. Aquests mètodes es noten precedint la seva declaració per l'etiqueta `abstract` i acabant-la amb `;`:

```
1 public abstract class Employee {
2      ...
3 
4      public abstract int getSalary();
5 }
```

- **Una classe abstracta defineix un tipus** que ofereix totes les operacions públiques declarades a la classe i també totes les operacions públiques de les seves superclasses.

Els mètodes abstractes d'una superclasse s'han de definir a les seves subclasses (no necessàriament a les subclasses directes). Mentre una subclasse no defineixi algun mètode declarat abstracte a una superclasse seva, aquella subclasse seguirà essent abstracta.

En el cas de `getSalary()` i `getCategory()` les podem definir així:

```
1 public class Engineer extends Employee {
2      ...
3 
4      public int getSalary() {
5          return 40000;
6      }
7 
8      public String getCategory() {
```

```
9           return "Engineer";
10      }
11  }
12
13
14  public class ProjectManager extends Employee {
15      ...
16
17      public int getSalary() {
18          return 60000;
19      }
20
21      public String getCategory() {
22          return "Project␣manager";
23      }
24  }
```

En conclusió, tenim dues classes abstractes: `Employee` i `RegularPolygon` de les quals no podem implementar alguns dels seus mètodes ni tampoc no podem instanciar-les (i.e., crear objectes d'aquestes classes). Aleshores.... *per què carai les hem de definir?* Doncs resulta que, malgrat tot, és convenient definir-les. Per què? (Preneu-vos un moment de reflexió abans de seguir llegint).

Almenys hi ha dues raons:

1. Proporcionen una representació i una implementació parcials a la resta de les classes de la jerarquia. Sense `Employee` i `RegularPolygon` ens caldria repetir els atributs comuns a totes les classes i caldria implementar a totes les classes de la jerarquia les operacions comunes amb exactament el mateix codi.

   Exemple 1.14:

   Si no definíssim `Employee` tindríem una repetició de codi del tot inútil:

```
1   public class Engineer {
2       private String nif;
3       private String name;
4       ...
5
6       public String getNif() {
7           return this.nif;
8       }
9
10      public String getName() {
11          return this.name;
12      }
13  }
14
15
16  public class DepartmentDirector {
17      private String nif;
```

```
18      private String name;
19      ...
20
21      public String getNif() {
22          return this.nif;
23      }
24
25      public String getName() {
26          return this.name;
27      }
28  }
```

■ ■ ■

2. Serveixen d'arrel a les jerarquies de treballadors i de polígons respectivament.

   Clar que aquesta no sembla una gran resposta: per què necessitem una arrel per a aquestes jerarquies? Per a que faci bonic? Per recordar-nos que tot el que tenim sota d'aquella arrel són, posem per cas, treballadors?

   Mmmm... no ben bé. En realitat, aquesta arrel (`Employee` o `RegularPolygon`) és la base necessària d'una de les eines més potents de la programació orientada a objectes. Però em temo que si voleu saber quina és aquesta eina haureu de llegir la secció 1.3.

   Abans, però, fem un petitíssim apunt sobre com es poden crear també jerarquies d'interfícies.

1.2.4 Interfícies i herència

De manera similar a com passa amb les classes, *una interfície (subinterfície) pot estendre una altra interfície (superinterfície)*.

Això pot tenir sentit en diferents casos:

1. Si té sentit definir una jerarquia d'interfícies en què la seva interfície arrel declari una llista d'operacions compartides per totes les interfícies de la jerarquia i cada *subinterfície* hi afegeixi algunes operacions que tinguin sentit només per aquella interfície concreta:

   Exemple 1.15:

```
1  public interface Van extends Vehicle {
2      public int getMaxLoad();
3  }
4
5
6  public interface Bicycle extends Vehicle {
7      public boolean isBTT();
8  }
```

   Aquestes dues interfícies hereten totes les operacions que la interfície `Vehicle` va declarar a la secció 1.1.5, com ara `getOwner()` o `getBrand()`. ■ ■ ■

2. Si, un cop definida una interfície (i implementada per diverses classes), ens adonem que hem d'afegir-hi més operacions, *no podem afegir-les-hi sense més ja que totes les classes que la implementaven esdevindran incorrectes*. Això es pot resoldre definint una subinterfície:

   Exemple 1.16:

   Molt temps més tard de la definició de la interfície `Vehicle` es vol controlar l'emissió de gasos En algun moment apareix el requeriment de recordar el seu any de fabricació. Com que no podem modificar la interfície `Vehicle` (que, probablement, haurà estat implementada per diverses classes) n'hi afegirem una *subinterfície*:

```
1 public interface VehicleWithProductionYear extends Vehicle {
2     public int getProductionYear();
3 }
```

   ■ ■ ■

1.3 Polimorfisme

1.3.1 Subtipus

Si un `Engineer` és un `Employee` sembla raonable que en aquells llocs d'un programa on s'espera una instància d'`Employee`, s'hi pugui posar una instància d'`Engineer`.

En particular, això passa a les *assignacions* i als *paràmetres de les operacions*.

Exemple 1.17:

```
1 void g(Employee emp) {
2     ...
3 }
4
5 Engineer eng = new Engineer(...);
6
7 g(eng);
```

■ ■ ■

Es diu que `Engineer` és un subtipus d'`Employee`. La propietat per la qual un `Employee` es pot substituir per un `Engineer` s'anomena *principi de substitució*. A l'inrevés no passa: o sigui, un subtipus (`Engineer`) no es pot substituir pel seu supertipus (`Employee`). Vegem-ho:

Exemple 1.18:

```
1 void f(Engineer e) {
2     ...
3 }
4
5 void g(Employee emp) {
```

```
 6     f(emp);              //MALAMENT!!!!
 7 }
 8
 9 ProjectManager pm = new ProjectManager(...);
10
11 g(pm);
```

Comentaris:

- Aquest exemple il·lustra el motiu pel qual no funciona: si substituïm una instància (`e`) d'un subtipus (`Engineer`) per una instància (`emp`) del seu supertipus (`Employee`), és possible que `emp` sigui, en realitat una instància d'algun altre subtipus d'`Employee` (en aquest cas, `ProjectManager`), que, clarament és incompatible amb `Engineer`).

■ ■ ■

Definim més formalment *subtipus*, *supertipus* i *principi de substitució*:

Un tipus `S` és un **subtipus** d'un altre tipus `T` (o, si voleu `T` és un **supertipus** de `S`) si:

- `T` és `Object`.

  (La classe `Object` és supertipus de qualsevol altre tipus).

- `S` és `T`.

  (Un tipus és subtipus de si mateix).

- `S` estén `T`.

  (Una classe és subtipus de la seva superclasse. Una interfície és subtipus de la seva *superinterfície*).

- `S` implementa `T`.

  (Una classe és subtipus de la interfície que implementa).

- `S` estén o implementa `S1` i `S1` és un subtipus de `T`.

  (La relació *és subtipus de* és transitiva: *Si* `S` *és subtipus de* `S1` *i* `S1` *ho és de* `T`, *aleshores* `S` *és subtipus de* `T`).

A vegades es diu que un subtipus **conforma** amb el seu supertipus.

Principi de substitució

Una variable d'un tipus `T` *pot ser substituïda en un programa per una altra d'un subtipus de* `T`.

Més específicament:

- A una variable d'un tipus `T` se li'n pot assignar una d'un subtipus `S` de `T`:

```
1 T t;
2 S s;
3 ...
4 t = s;
```

- Una operació amb un paràmetre de tipus `T` es pot cridar amb un paràmetre d'un subtipus `S` de `T`.

```
1 void f(T t) { ... }
2 S s;
3 f(s);
```

A l'inrevés no és correcte.

Exemple 1.19: Un exemple de conformació de tipus

Reprenem la jerarquia de treballadors que hem presentat a la secció 1.2 i suposem ara que volem escriure una operació que escrigui per la sortida estàndard el nom, la categoria (*engineer, project manager, department director...*) i el sou d'un treballador que pertanyi a alguna de les classes de la jerarquia i que es passarà com a paràmetre de l'operació. Quina seria la declaració d'aquella operació?

```
1 public void showNameCatAndSalary(??? emp) { ... }
```

Quin tipus hi posem al lloc de `???`?

Hi hem de posar un tipus que permeti fer crides com ara:

```
1 public static void main(String[] args) {
2     Engineer eng =
3             new Engineer("40999444", "Joel Pi", ...);
4     ProjectManager pmanager =
5             new ProjectManager("45551001", "Pau Prou", ...);
6
7     showNameCatAndSalary(eng);
8     showNameCatAndSalary(pmanager);
9 }
```

O sigui, hem de poder cridar l'operació amb un objecte de la classe `Engineer` o bé amb un de la classe `ProjectManager` o bé de qualsevol altra subclasse d'`Employee`. Com que els objectes de qualsevol d'aquestes subclasses *són subtipus d'*`Employee`, podem escriure:

```
1 public void showNameCatAndSalary(Employee emp) { ... }
```

Si escrivim aquesta declaració podrem cridar l'operació `showNameCatAndSalary` passant-li com a paràmetre un objecte de qualsevol subclasse d'`Employee`.

Qualsevol subclasse d'`Employee` *és subtipus* d'`Employee` (en altres paraules, *conforma amb* `Employee`) i això vol dir que es pot posar un objecte d'aquella classe allà on s'espera un objecte de tipus `Employee`.

■ ■ ■

> Per cert, algú podria pensar: com podem escriure la declaració `Employee emp` si havíem dit a la secció 1.2.3 que `Employee` era una classe abstracta i, per tant, no podem instanciar-la? Notem que ningú no l'està instanciant. Adonem-nos que `Employee emp` **no crea cap objecte de la classe Employee** (ni de cap altra classe) només declara que els objectes `obj` amb què es cridarà l'operació `showNameCatAndSalary(obj)` hauran de ser instàncies d'una classe que sigui subtipus d'`Employee`, això és, d'una subclasse d'`Employee`. Des d'aquest punt de vista, la classe abstracta `Employee` la veiem simplement com a tipus, no com a implementació.

1.3.2 Concepte de polimorfisme

Ara que ja sabem com declarar l'operació `showNameCatAndSalary(...)`, ens podem plantejar com implementar-la. Provem-ho:

Exemple 1.20: Operació polimòrfica

Llistat 1.5: showNameCatAndSalary

```
1  public void showNameCatAndSalary(Employee emp) {
2      System.out.println("Employee: "+emp.getName());
3      System.out.println("Category: "+emp.getCategory());
4      System.out.println("Salary: "+ emp.getSalary());
5      System.out.println("------------------------------");
6  }
7
8  public static void main(String[] args) {
9      Department dtest = new Department("test");
10     Project proj = new Project("InnovaCampus");
11     Engineer eng =
12             new Engineer("40999444", "Joel Pi", dtest, proj);
13     ProjectManager pmanager =
14             new ProjectManager("45551001", "Pau Prou", proj);
15
16     showNameCatAndSalary(eng);
17     showNameCatAndSalary(pmanager);
18 }
```

Quin és el resultat que esperem de l'execució d'aquest codi? Segurament ens decebria qualsevol cosa diferent de la següent:

Llistat 1.6: Resultat de showNameCatAndSalary

```
1 Employee: Joel Pi
2 Category: Engineer
3 Salary: 40000
4 ---------------------------
5 Employee: Pau Prou
6 Category: Project manager
7 Salary: 150000
8 ---------------------------
```

Per tant, esperem que les instruccions de les línies 3 i 4: `emp.getCategory()` i `emp.getSalary()` es comportin com si `emp` fos un `Engineer` a la crida `showNameCatAndSalary(eng)` (vegeu línia 16) i com si fos un `ProjectManager` a la crida `showNameCatAndSalary(pmanager)` (vegeu línia 17).

O, dit d'una manera més precisa, esperem que `emp.getCategory()` es vinculi amb `Engineer.getCategory()` en el primer cas i a `ProjectManager.getCategory()`, en el segon (i el mateix amb `emp.getSalary()`).

O sigui, necessitem que *una mateixa línia de codi* (e.g., la crida `emp.getCategory()` i també `emp.getSalary`) *es puguin vincular a diferents codis d'operació segons quin sigui el tipus que té* **en temps d'execució** *l'objecte sobre el qual es criden*:

emp és Engineer en temps d'execució: `emp.getCategory()` ⟶ `Engineer.getCategory()`
emp és ProjectManager en temps d'execució: `emp.getCategory()` ⟶ `ProjectManager.getCategory()`

■ ■ ■

Tipus compilació i tipus execució

En aquesta discussió que acabem de fer, hem pogut observar una idea nova i molt important, que és la base del polimorfisme:

Considerem una variable `x` declarada d'un *tipus referència* `T`. A `T` l'anomenem *tipus compilació* d'`x`.

Aquella variable `x` pot vincular-se, al llarg de l'execució de l'aplicació, a objectes diferents de subtipus diferents de `T`. Al subtipus `S` al qual `x` està vinculada en un instant determinat de l'execució de l'aplicació, l'anomenem *tipus execució* d'`x`.

Una variable té un únic tipus compilació però pot tenir diversos tipus execució al llarg de l'execució de l'aplicació.

Tots els tipus execució que tingui una variable han de ser subtipus del seu tipus compilació.

Exemple 1.21: Tipus compilació i tipus execució

A l'exemple anterior, la variable `emp` que és paràmetre de `showNameCatAndSalary(...)`:

```
1 public void showNameCatAndSalary(Employee emp) { ... }
```

té `Employee` com a *tipus compilació*. En la primera crida a l'operació té `Engineer` com a tipus execució i, a la segona té `ProjectManager` com a tipus execució.

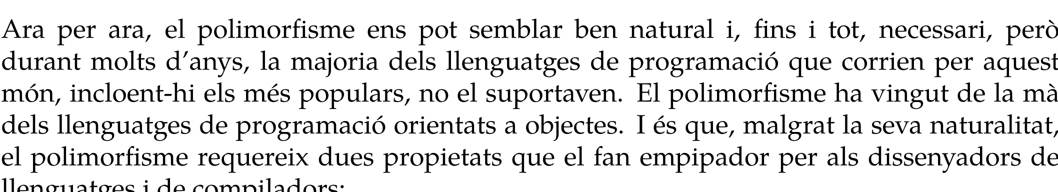
■ ■ ■

Definició de polimorfisme

Hem vist, doncs, que una mateixa crida a operació es pot vincular a operacions diferents en diferents execucions d'aquella crida i, per tant, la crida es comportarà de formes diferents, depenent del tipus execució de l'objecte sobre el qual es fa aquella crida. Heus ací el nom de *crida polimòrfica*: es tracta d'una crida que pot exhibir formes (més precisament, *comportaments*) diferents.

Tot plegat ens fa arribar a la definició de polimorfisme:

> **Definició de polimorfisme**
> Direm que la crida `ref.op()` té un comportament polimòrfic si en cada execució d'aquesta crida es vincula a l'operació `op` de la classe de l'objecte al qual es refereixi `ref` en el moment de fer la crida. Aquesta classe haurà de ser subtipus del tipus de la declaració de `ref` (vegeu la secció 1.3.1).
>
> *Exemple:* Si `emp` ha estat declarat: `Employee emp;` aleshores: `emp.getSalary()` es vincularà a `Engineer.getSalary()` o a `ProjectManager.getSalary()` depenent del tipus de l'objecte referit per `emp` quan es fa la crida. Notem que tant `Engineer` com `ProjectManager` són subtipus d'`Employee` perquè en són subclasses.



Ara per ara, el polimorfisme ens pot semblar ben natural i, fins i tot, necessari, però durant molts d'anys, la majoria dels llenguatges de programació que corrien per aquest món, incloent-hi els més populars, no el suportaven. El polimorfisme ha vingut de la mà dels llenguatges de programació orientats a objectes. I és que, malgrat la seva naturalitat, el polimorfisme requereix dues propietats que el fan empipador per als dissenyadors de llenguatges i de compiladors:

- *Conformació de tipus*

  Una referència a un tipus `T` ha de poder-se referir en diferents moments de l'execució d'un programa a qualsevol objecte d'un subtipus de `T` (recordeu la secció 1.3.1).

  És el cas del paràmetre `Employee emp` de `showNameCatAndSalary(Employee emp)`, que pot referir-se en diferents moments de l'execució d'un mateix programa a qualsevol objecte d'un subtipus d'`Employee` (qualsevol subclasse d'`Employee`).

- *Lligam dinàmic*

  La vinculació entre la crida a una operació i el codi que ha d'executar aquella crida es fa en temps d'execució i no en temps de compilació com era habitual en la gran majoria dels llenguatges de programació previs a la POO.

  En aquells llenguatges, quan es compilava una crida a funció, ja s'associava amb el seu codi. És clar que amb el polimorfisme això no es podrà fer perquè, en realitat, hi ha molts codis candidats a vincular-se a una crida específica. Només en temps d'execució podrem resoldre, en cada cas, quin és el codi triat.

1.3.3 Classes abstractes vs. interfícies(*)

Tant les classes abstractes com les interfícies defineixen tipus. En el cas de les classes abstractes, aquests tipus poden estar parcialment implementats. En el cas de les interfícies, no. Però totes dues defineixen tipus. A més a més, el tipus definit per una classe que implementa la interfície `I` *és subtipus* de `I` o, expressat d'una altra manera, *conforma amb* `I`. I el tipus definit per una classe que estén una classe abstracta `A` (i.e., una subclasse de `A`) també *és subtipus de* `A`.

Per tant, la nostra operació:

```
1 public void showNameCatAndSalary(Employee emp);
```

tindria sentit i podria actuar polimòrficament tant si haguéssim definit:

```
1 public abstract class Employee { ... }
2
3 public class Engineer extends Employee { ... }
4
5 public class ProjectManager extends Employee { ... }
```

com si haguéssim definit:

```
1 public interface Employee { ... }
2
3 public class Engineer implements Employee { ... }
4
5 public class ProjectManager implements Employee { ... }
```

Així doncs, com hem de dissenyar `Employee`? Com a interfície o com a classe abstracta? En aquest cas, la resposta és simple: *com a classe abstracta*. Per què? Penseu-hi un moment abans de continuar llegint...

La classe abstracta `Employee` defineix una estructura (atributs) que són compartits per tota la jerarquia de classes. Per exemple, `name, nif, email...`. Igualment, `Employee` defineix unes operacions, la implementació de les quals és novament compartida per tota la jerarquia: `getName(), getNif()`... Si haguéssim definit `Employee` com a *interfície*, hauríem hagut de repetir la definició d'aquests atributs i la implementació d'aquestes operacions a tota la jerarquia, mentre que ara en tenim prou de fer la definició un sol cop a la classe abstracta `Employee` (vegeu exemple en secció 1.2.3).

Ampliem ara l'exemple anterior: alguns tipus de treballadors de l'empresa tenen certes capacitats; per exemple, els `ProjectManagers` i els `DeptDirectors` tenen la capacitat de fer presentacions als clients, mentre que els altres tipus de treballadors no la tenen. Com podem dissenyar aquesta situació? De forma natural, amb interfícies.

```
1 public interface Communicator {
2     public Presentation generatePresentation();
3 }
4
5 public class DeptDirector extends Employee
6                           implements Communicator { ... }
7
8 public class ProjectManager extends Employee
9                             implements Communicator { ... }
```

Donem encara un tomb de cargol més. Considerem una altra capacitat que tenen alguns treballadors: la d'escriure informes tècnics. Això ho poden fer els `Engineers` i els `ProjectManagers`. No així els `DeptDirectors` que tenen la tasca de gestionar un departament i saben poca cosa d'aspectes tècnics. Tindríem:

```
1 public interface TechWriter{
2      public Report generateTechReport();
3 }
4
5 public class Engineer extends Employee
6                       implements TechWriter { ... }
7
8 public class ProjectManager extends Employee
9                             implements Communicator, TechWriter { ... }
```

Veiem que la classe `ProjectManager` pot comunicar als clients i també escriure informes tècnics. Per tant, implementa dues interfícies (i estén una classe!!!). Noteu que `ProjectManager` no podria estendre dues classes.

Conclusió

En general, usarem una classe abstracta si...

- Tenim diverses classes clarament lligades per una relació de generalització (`Engineer` *és un* `Employee`) i tals que la classe més general defineix uns atributs o implementació d'operacions compartida amb les altres classes.

  Exemple: `Employee` *és una classe abstracta que defineix uns atributs i una implementació d'operacions compartides per tota la jerarquia de treballadors.*

En general, usarem una interfície si...

- Tenim diverses classes que ofereixen totes unes determinades operacions però aquestes classes no estan lligades clarament amb la que declara aquelles operacions (`I`) per una relació de generalització (*és un*) o bé no tenen atributs o implementació d'operacions comunes amb `I`.

  Exemple: `TechWriter` *és una interfície implementada per* `Engineer` *i* `ProjectManager`.

- Una classe ha d'oferir diferents comportaments, cadascun dels quals vindrà descrit per una interfície. Recordem que una classe no pot estendre diverses superclasses però sí que pot implementar diverses interfícies.

  Exemple: `ProjectManager` *implementa les interfícies* `TechWriter` *i* `Communicator`.

En cas de dubte...

En cas de dubte (en particular, si no hi ha atributs o implementacions d'operacions que compartir per tota la col·lecció de classes), useu, preferiblement, interfícies.

1.3.4 Conversions explícites cap avall i comprovació de tipus

A vegades és necessari convertir una referència a un subtipus del tipus del qual està declarada. Vegem-ne un exemple.

Exemple 1.22:

La classe `Object` (que, com sabem, és l'arrel de la jerarquia de classes del Java) ofereix l'operació `equals`:

```
1 public boolean equals(Object obj) { ... }
```

Aquesta operació ens indica si el paràmetre `obj` "és igual"a l'objecte sobre el qual es crida l'operació:

Així, `obj1.equals(obj)` retorna `true` si `obj1` "és igual"a `obj`. Però *què vol dir "és igual"?* Si els objects són `Strings` voldrà dir que tenen els mateixos caràcters i en el mateix ordre; si són `Employees`, hauran de tenir els mateixos `nif`, `name` i `email`. Si són `DeptDirs`, a més a més, hauran de ser directors del mateix `dept`.

Sembla clar que haurem de redefinir l'operació `equals` per a cada classe, els objectes de la qual vulguem comparar. Fem-ho:

```
1 public class DeptDirector extends Employee {
2
```

```
3       @Override
4       public boolean equals(Object obj) {
5           ...
6       }
7       ...
8   }
```

La primera cosa és que, malgrat que el paràmetre esperem que sigui de classe `DeptDirector` (voldrem comparar `DeptDirectors` amb `DeptDirectors`) hem de declarar-lo de tipus `Object` atès que *la redefinició d'una operació en una subclasse ha de mantenir la signatura d'aquella operació a la superclasse* (recordeu 1.2).

Proposem ara una implementació per `DeptDirector.equals`:

Llistat 1.7: Implementació **ERRÒNIA** de `DeptDirector.equals`

```
1   public class DeptDirector extends Employee {
2
3       @Override
4       public boolean equals(Object obj) {
5         return this.getNif().equals(obj.getNif())     //MALAMENT!!!
6                && this.getName().equals(obj.getName())
7                && this.getEmail() == obj.getEmail()
8                && this.dept.equals(obj.dept) ;
9       }
10      ...
11  }
```

Simple, elegant i... **incorrecte!!!!** Per què? Penseu-hi un moment, sisplau...

Aquest codi generarà un error de compilació perquè `obj` està declarat de tipus `Object` i la classe `Object` (que implementa aquest tipus) *no defineix les operacions* `getName()`, `getNif()`, `getEmail()` *ni l'atribut* `dept`*!!!*. Per tant, crides com ara `obj.getNif()` no tenen cap sentit per al compilador.

■ ■ ■

Però això no hauria de ser cap problema atès que, en general, quan des d'un programa cridem l'operació `DeptDirector.equals(...)` ho farem amb un paràmetre de tipus `DeptDirector` i, per tant, aquell paràmetre *sí que tindrà les operacions* `getName()`, `getNif()` *i* `getEmail()`. Així doncs, podem resoldre el problema, simplement forçant una conversió explícita de tipus: introduirem al codi una instrucció que convertirà `obj` formalment en un `DeptDirector`:

Llistat 1.8: Implementació de `DeptDirector.equals` (segon intent encara no reeixit del tot)

```
1   public class DeptDirector extends Employee {
2
3       @Override
4       public boolean equals(Object obj) {
5           DeptDirector dir = (DeptDirector) obj;
6
```

```
7           return this.nif.equals(dir.nif)
8                   && this.name.equals(dir.name)
9                   && this.email.equals(dir.email)
10                  && this.dept.equals(dir.dept);
11      }
12      ...
13  }
```

La instrucció de la línia 5 fa que `dir` sigui una referència a `DeptDirector` que es refereixi al mateix objecte que `obj`. Per tant, ara sí que no hi ha cap problema de fer coses com `dir.getName()`. La instrucció continguda a la línia 5 se l'anomena *conversió explícita de tipus*. Com que en aquest cas, es converteix un *supertipus* en un *subtipus* (clarament, `DeptDirector` és subtipus d'`Object`) se l'anomena *conversió explícita de tipus cap avall*; o, si voleu, amb l'economia de termes que caracteritza als americans: *downcast*.

Conversió explícita de tipus cap avall
Instrucció que converteix explícitament una referència `t` que ha estat declarada de tipus `T` a una referència a un tipus `S` de manera que `S` és un subtipus de `T`.

```
1 T t;
2 S s = (S) t;
```

Perquè la conversió explícita de tipus cap avall sigui correcta, el tipus de l'objecte al qual es refereix `t` ha de ser `S` o un subtipus de `S`.

Com que `S` és sempre un subtipus de si mateix, podem simplificar la darrera frase dient simplement:

Perquè la conversió explícita de tipus cap avall sigui correcta, el tipus de l'objecte al qual es refereix `t` ha de ser un subtipus de `S`.

Aquest aspecte de la correctesa d'una conversió explícita de tipus convé reflexionar-lo una mica: La conversió explícita de tipus *cap amunt* funciona sempre: si `S` és un subtipus de `T`, aleshores un objecte d'un tipus `S` és també de tipus `T`. Per exemple:

```
1 Engineer eng = new Engineer();
2 Employee emp = (Employee) eng;
```

Funciona sempre perquè `eng` (definit de tipus `Engineer`) és també de tipus `Employee` perquè un `Engineer` és un `Employee`.

Però compte!!!, la conversió explícita de tipus *cap avall* només funcionarà, com hem dit, si l'objecte que volem convertir és una instància d'un subtipus del tipus al qual el volem convertir (a l'exemple, `obj` és una instància d'un subtipus de `DeptDirector`). Però què passarà si no és així? Hi ha dues situacions típicament problemàtiques:

1. *La referència que es vol convertir està declarada d'un tipus T_1 que no és supertipus del tipus T_2 al qual se la vol convertir.*

```
1 Integer i = new Integer(3);
2 DeptDirector dir = (DeptDirector) i;
```

Clarament, `Integer` no és supertipus de `DeptDirector`. De cap manera una referència a `Integer` pot referir-se a un objecte d'un subtipus de `DeptDirector`. Això genera el següent error de compilació.

`Inconvertible types. Required DeptDirector. Found Integer`

Aquest error no és gaire freqüent (i, si ho és, és senzill d'arreglar perquè ens n'adonem quan compilem el programa). El segon problema és més empipador:

2. *La referència que es vol convertir* **pot ser** *d'un subtipus del tipus al qual se la vol convertir però, en temps d'execució,* **no ho és**.

Per il·lustrar-ho, recordem novament el codi de l'operació `equals` (1.8):

```
1  public class DeptDirector extends Employee {
2
3      @Override
4      public boolean equals(Object obj) {
5          DeptDirector dir = (DeptDirector) obj;
6
7          return this.nif.equals(dir.nif)
8                  && this.name.equals(dir.name)
9                  && this.email.equals(dir.email)
10                 && this.dept.equals(dir.dept);
11     }
12     ...
13 }
```

La línia 5 converteix la referència `obj` a `DeptDirector`. `obj` està declarada de tipus `Object`, per tant, *pot ser d'un subtipus de* `DeptDirector` (per exemple, si cridem `equals` passant-li com a paràmetre un objecte instància de `DeptDirector` o de qualsevol subclasse de `DeptDirector`, si n'hi hagués):

```
1 DeptDirector dir1 = new DeptDirector(...);
2 DeptDirector dir2 = new DeptDirector(...);
3
4 if (dir1.equals(dir2)) { ... }
```

En aquest cas `obj` fa referència a un `DeptDirector` i tot va bé. Però podria passar que `obj` fes referència a un objecte d'un tipus que no fos subtipus de `DeptDirector`:

```
1 Engineer eng = new Engineer(...);
2
3 if (dir1.equals(eng)) { ... }
```

Clarament `obj` fa referència a un `Engineer` que no és subtipus de `DeptDirector`. Salten les alarmes.

Però quan salten? I com? Aquesta situació no genera un error de compilació. De fet, en general, aquest problema no és detectable en temps de compilació (se us acut per què?). Aleshores el Java el resol llençant una excepció de tipus `ClassCastException` a la instrucció:

```
1 DeptDirector dir = (DeptDirector) obj;
```

Si no tractem l'excepció que es produeix en la segona situació problemàtica que acabem de veure es provocarà una aturada de l'aplicació. Però, si hi pensem, aquesta aturada és del tot innecessària: un objecte de la classe `DeptDirector` no el considerarem mai igual a un de la classe `Engineer`; per tant, n'hi hauria prou que si es produeix un `ClassCastException` l'operació `DeptDirector.equals()` retorni `false`:

```
1  public class DeptDirector extends Employee {
2
3      @Override
4      public boolean equals(Object obj) {
5          try {
6              DeptDirector dir = (DeptDirector) obj;
7
8              return this.nif.equals(dir.nif)
9                      && this.name.equals(dir.name)
10                     && this.email.equals(dir.email)
11                     && this.dept.equals(dir.dept);
12         } catch (ClassCastException e) {
13             return false;
14         }
15     }
16     ...
17 }
```

Comprovació de tipus

Si volem evitar que es llenci l'excepció `ClassCastException` podem consultar el tipus d'un objecte en temps d'execució. El Java ens proporciona l'operador `instanceof` per fer això:

Operador instanceof
`ref instanceof T` retorna cert si `ref` és instància d'un subtipus de `T`.
O sigui, si `ref` és instància de `T`, d'una subclasse (directa o indirecta) de `T` o d'una classe que implementi la interfície `T` (en el cas que `T` sigui una interfície).

Exemple 1.23:

Us proposem ara una versió de `DeptDirector.equals` que usa `instanceof` en lloc de capturar l'excepció `ClassCastException` que ara ja no es podrà produir.

```
1  public class DeptDirector extends Employee {
2
3      @Override
4      public boolean equals(Object obj) {
5          if (obj instanceof DeptDirector) {
6              DeptDirector dir = (DeptDirector) obj;
7
8              return this.nif.equals(dir.nif)
9                      && this.name.equals(dir.name)
10                     && this.email.equals(dir.email)
11                     && this.dept.equals(dir.dept);
12         } else {
13             return false;
14         }
15     }
16     ...
17 }
```

Com es podria reescriure el codi d'aquesta operació sense posar cap `if`?

■ ■ ■

Aprofitant-nos de l'herència per implementar `equals()`

Encara no hem acabat (tot i que, afortunadament, ja falta poc): la implementació de `DeptDirector.equals()` encara no és del tot satisfactòria. Per veure per què, penseu en quina fóra la implementació d'`Engineer.equals()` o de `ProjectManager.equals()`. Hi detecteu alguna redundància de codi? Podeu pensar en alguna manera per estalviar-la?

1.4 Genericitat

1.4.1 Tipus genèrics

Aquest curs està dedicat a les estructures de dades: llistes, taules, arbres... Una llista, per exemple, és una seqüència d'elements d'un cert tipus. Però de quin tipus? Pot ser `Integer`, `Person`, `Employee`... Aquesta reflexió ens porta a definir diferents classes de llistes:

`ListOfInteger`, `ListOfEmployee`, `ListOfPerson`...

Us sembla raonable això?

Pensem-hi una mica: Per a cada tipus d'elements `T` que volguéssim posar en una llista hauríem de definir una `ListOfT`. N'hi hauria innombrables. No acabaríem mai. A més a més, cadascun d'aquells tipus llista es comportaria de la mateixa manera, tindria les mateixes operacions (afegir, treure o consultar elements de la llista). Si fos una classe, segurament tindria la mateixa representació... Quina pèrdua de temps!!!

El que plantegem en aquesta secció és definir un únic tipus (interfície o classe); per exemple, `List<T>` que depengui d'un paràmetre (`T`) que indica el tipus dels elements de la llista.

Vegem-ho.

Exemple 1.24: Declaració de tipus genèrics

El paquet `java.util` conté la definició d'una sèrie de classes i interfícies que globalment s'anomenen *les col·leccions del Java* (*Java collections*). En aquest paquet s'hi apleguen totes les estructures de dades que estudiarem en aquest curs. En particular conté la interfície `List<T>`.

La interfície `List<T>` modelitza una seqüència d'elements de tipus genèric `T`. Aquest `T` és un paràmetre que es podrà *invocar* amb un tipus concret cada cop que es necessiti treballar amb una llista amb elements que hagin de ser d'un tipus específic:

```
1  List<Integer> lint;
2
3  List<Engineer> leng;
```

- `lint` és una referència a una llista els elements de la qual seran `Integer`.
- `leng` és una referència a una llista els elements de la qual seran `Engineer`.

Només hi ha una interfície `List` (que, de fet, anomenarem `List<T>`), definida amb unes operacions determinades: `add` (afegeix un element a la llista), `get` (obté un element de la llista), `size` (retorna el nombre d'elements de la llista)... Però aquesta interfície admet diferents tipus d'elements components depenent del tipus amb el qual s'instanciï `T`.

Vegem com usem la interfície `List<T>`.

```
1  List<Engineer> leng = new LinkedList<Engineer>();
2
3  leng.add(new Engineer("12345678","Joan Petit"));
4
5  List<Integer> lint = new LinkedList<Integer>();
6
7  lint.add(new Integer(2));
8
9  int n = leng.size();
10 int n2 = lint.size();
11
12 int val = lint.get(0).intValue();
```

Comentaris:

- `LinkedList<T>` és una classe que implementa la interfície `List<T>`.
- L'operació `List.get(i)` retorna l'element que ocupa la posició `i` de la llista.

- El paràmetre dels genèrics (`T`) només es pot substituir per un tipus referència (interfícies, classes o vectors). Per tant, no fóra correcte: `List<int> li;`.

  Això sí, el compilador del Java és capaç d'afegir el codi que calgui per convertir tipus primitius (`int`, `double`...) en els seus tipus referència corresponents i viceversa. Per tant, el codi anterior l'hauríem pogut escriure també de la manera següent:

```
1 List<Integer> lint = new LinkedList<Integer>();
2
3 lint.add(2);
4
5 int val = lint.get(0);
```

■ ■ ■

Un **tipus genèric** és un tipus que depèn d'un o diversos paràmetres que, a la vegada, representen tipus referència (classes, interfícies o, fins i tot, vectors).

Exemple: `List<T>`
`T` és un paràmetre que podrà ser *invocat* amb un tipus determinat:

Exemples:
`List<Integer>`
`List<Engineer>`

Un tipus genèric es defineix de la manera següent:

```
1 public interface Box<T> {
2     public void put(T obj);
3     public void remove (T obj);
4     public T get(int i);
5     public boolean isEmpty();
6     public void reset();
7 }
```

Aquest codi defineix una interfície `Box`, que modelitza una caixa, que conté elements d'un cert tipus `T`. La podem declarar fent:

```
1 Box<String> bst;
```

Igualment, es pot definir una classe genèrica.

Implementació dels genèrics per instanciació (*)

Els llenguatges de programació poden usar dues grans maneres d'implementar els genèrics: per *instanciació* o per *esborrament* (en anglès, *erasure*). En aquesta secció introduïm la primera.

En els genèrics implementats per instanciació, una classe genèrica no és pròpiament una classe sinó més aviat una *plantilla de classe*, o sigui, unes *instruccions* per generar una classe. Quan el paràmetre tipus del genèric *s'instancia* amb un tipus concret (per exemple, una classe concreta) aleshores el compilador genera la classe pròpiament. Per a cada tipus diferent amb què el programador instancïi una classe genèrica, el compilador generarà una classe diferent.

El llenguatge C++ tria implementar-los per *instanciació*.

Exemple 1.25: Genèrics per instanciació en C++

```
1  template<class T>
2  class MyList {
3      T elems[100];
4      int nelems;
5  public:
6      MyList() {
7          ...
8      }
9
10     void add(const T& elem) {
11         ...
12     }
13
14     T get(int i) {
15         ...
16     }
17     ...
18 };
19
20 int main() {
21     MyList<Engineer> lis1;
22     MyList<int> lis2;
23
24     Engineer e1(...);
25     Engineer e2;
26     int i;
27
28     lis1.add(e1);
29     lis2.add(3);
30
31     e2 = lis1.get(0);
32     i = lis2.get(0);
33 }
```

Comentaris:

- `MyList<T>` és una classe genèrica escrita en C++ i, per tant, amb els genèrics implementats per instanciació.
- El compilador, a les línies 21 i 22 genera **dues** classes diferents: `MyList<Engineer>`

i `MyList<int>`. Per crear-les, utilitza les *instruccions* que apareixen a la plantilla `template<class T> class MyList....;` que, insistim, no és una classe sinó una plantilla de classe.

Essencialment, el compilador substitueix el paràmetre tipus (`T`) pel tipus que l'instancia (`Engineer` o `int`). O sigui, la línia 21 genera una cosa similar a:

```
1  class MyList_Engineer {
2      Engineer elems[100];
3      int nelems;
4  public:
5      MyList() {
6          ...
7      }
8
9      void add(const Engineer& elem) {
10         ...
11     }
12
13     Engineer get(int i) {
14         ...
15     }
16     ...
17 };
```

I la línia 22:

```
1  class MyList_int {
2      int elems[100];
3      int nelems;
4  public:
5      MyList() {
6          ...
7      }
8
9      void add(const int& elem) {
10         ...
11     }
12
13     int get(int i) {
14         ...
15     }
16     ...
17 };
```

- Les línies 28, 29, 30 i 31 usen les operacions de les classes corresponents assumint que cada classe ha instanciat `T` amb el tipus corresponent (`Engineer` o `int`).

■ ■ ■

Implementació dels genèrics per esborrament (*)

Un llenguatge, com ara el Java, que implementa els genèrics *per esborrament* (en anglès, *erasure*) defineix, per a cada classe genèrica *una única classe on el paràmetre tipus* `T` *ha estat substituït per* `Object` (i.e., per a la classe arrel de la jerarquia de classes).

A cada ocurrència a l'aplicació d'una classe genèrica, el seu paràmetre tipus és *esborrament* i substituït per conversions de tipus (en anglès, *casts*) als llocs escaients. Vegem-ne un exemple:

Exemple 1.26: Genèrics per esborrament en el Java

```
1  public class MyList<T> {
2      private T[] elems;
3      private int nelems;
4
5      public  MyList() {
6          ...
7      }
8
9      public void add(T elem) {
10         ...
11     }
12
13     public T get(int i) {
14         ...
15     }
16     ...
17 }
18
19 MyList<Engineer> lis1 = new MyList<Engineer>();
20 MyList<Integer> lis2 = new MyList<Integer>();
21
22 Engineer e1 = new Engineer(...);
23 Engineer e2;
24 int i;
25
26 lis1.add(e1);
27 lis2.add(3);
28
29 e2 = lis1.get(0);
30 i = lis2.get(0);
```

Aquest codi, el Java el transforma en alguna cosa que, conceptualment, seria similar a:

```
1  class MyList {
2      private Object[] elems;
3      private int nelems;
4
5      public  MyList() {
6          ...
7      }
8
```

```
 9      public void add(Object elem) {
10          ...
11      }
12
13      public Object get(int i) {
14          ...
15      }
16      ...
17  }
18
19  MyList lis1 = new MyList();
20  MyList lis2 = new MyList();
21
22  Engineer e1 = new Engineer(...);
23  Engineer e2;
24  int i;
25
26  lis1.add(e1);
27  lis2.add(new Integer(3));
28
29  e2 = ((Engineer)lis1.get(0));
30  i = ((Integer)lis2.get(0)).intValue();
```

Comentaris:

- Com hem dit abans, el compilador del Java considera una única classe (`MyList`), esborra els paràmetres tipus dels objectes declarats de classe genèrica (com `lis1` i `lis2`; vegeu les línies 19 i 20) i introdueix les conversions de tipus (*casts*) necessàries per assegurar que tots els paràmetres i retorns de les operacions amb paràmetre `T` corresponguin al tipus per als quals el programador ha instanciat `T`. Vegeu-ho, per exemple, a les línies 29 i 30.

■ ■ ■

Els genèrics per esborrament eviten la proliferació de classes idèntiques en tot excepte en la instanciació del paràmetre tipus però, per contra, plantegen majors dificultats per treballar polimòrficament i no permeten paràmetres que no siguin tipus referència (C++ permet, per exemple, constants o tipus primitius, com a paràmetres del genèric).

1.4.2 Operacions genèriques

Hi ha operacions que usen com a paràmetre o que retornen tipus genèrics (o el paràmetre d'un tipus genèric). Vegem com es declaren i s'invoquen aquestes operacions.

Exemple 1.27: Declaració i invocació d'operacions genèriques

La classe `Collections` del paquet `java.util` ofereix operacions que manipulen estructures de dades (que sovint, s'anomenen *col·leccions*). L'operació `Collections.replaceAll(...)` és una de les operacions oferides per `Collections`. Té aquesta especificació:

```
1 public static <T> boolean replaceAll(List<T> list,
2                                      T oldVal,
3                                      T newVal)
```

Reemplaça totes les ocurrències del valor `oldVal` a la llista `list` pel valor `newVal`.

Aquesta és una *operació genèrica*.

I la podem invocar normalment:

```
1 List<Engineer> leng = new LinkedList<Engineer>();
2 Engineer engOld = new Engineer("12345678","Joan Petit");
3 Engineer engNew = new Engineer("87654321","Petit Joan");
4
5 leng.add(engOld);
6
7 Collection.replaceAll(leng, engOld, engNew);
```

Notem que a la crida

```
1 Collection.replaceAll(leng, engOld, engNew);
```

no indiquem explícitament que `T` s'ha de substituir pel tipus `Engineer`. Així i tot funciona. I funciona perquè el compilador del Java és capaç d'inferir quin ha de ser aquell tipus.

■ ■ ■

En el cas de l'exemple anterior, inferir que `T` s'ha de substituir per `Engineer` és molt senzill. Però quan apareixen subtipus (e.g., `TestEngineer` és subtipus d'`Engineer`) i genèrics (e.g., `List<T>` és una interfície genèrica) la inferència del tipus amb el qual s'ha de substituir `T` en una operació genèrica no és sempre evident. Això és perquè els subtipus amb genèrics tenen un comportament poc intuïtiu. Expliquem, primer, com es comporten els subtipus amb genèrics i després com s'infereixen els tipus de substitució de `T` a les operacions genèriques.

1.4.3 Subtipus amb genèrics

El concepte de *subtipus* que vam presentar a la secció 1.3.1 l'hem d'enriquir amb l'aparició dels genèrics: *sota quines condicions podem dir que un tipus genèric és subtipus d'un altre?*

Un tipus `S1<T1>` és subtipus d'un altre `S2<T2>` si

- `S1` és subtipus de `S2` i
- `T1` és el mateix que `T2`.

Exemple 1.28: Subtipus amb genèrics

LinkedList<Integer> és subtipus de List<Integer>.

Perquè LinkedList és subtipus de List (atès que LinkedList és una classe que implementa la interfície List).

■ ■ ■

I aleshores, aplicant la definició, salta la sorpresa:

> List<Engineer> **no és subtipus** de List<Employee>.

Podeu imaginar-vos quins problemes ens portaria el fet que List<Engineer> fos subtipus de List<Employee>???

Vegem-ho:

Exemple 1.29: Problemes amb els subtipus

```
1 List<Engineer> leng = new LinkedList<Engineer>();
2 leng.add(new Engineer(...));
3
4 List<Employee> lemp = leng;                    //ERROR COMPILACIO
5
6 lemp.add(new ProjectManager(...));             //MALAMENT!!!
```

Comentaris:

- La instrucció de la línia 1 no planteja cap problema atès que, com hem vist, LinkedList<Engineer> és subtipus de List<Engineer>.
- La instrucció de la línia 4 seria acceptable si el Java admetés que List<Engineer> és un subtipus de List<Employee>. Com que no ho admet, genera un error de compilació. Però mantinguem aquesta instrucció com si fos correcta en el Java per veure a quins problemes ens pot portar.
- Com a conseqüència de la instrucció de la línia 4, lemp i leng es refereixen al mateix objecte que, en temps d'execució, és de tipus LinkedList<Engineer>. Per tant, la instrucció de la línia 6, que afegeix a lemp una instància de ProjectManager, és clarament errònia. Aquest és el perill.

■ ■ ■

El fet que List<Engineer> no pugui ser subtipus de List<Employee> és un problema perquè en ocasions voldrem que ho sigui. Per mitigar aquest problema apareixen les wildcards.

***Exemple 1.30: Necessitat de les** wildcards*

Suposem que volem enviar un correu electrònic amb un *text* i un *tema* determinats a tots els treballadors d'una llista de treballadors de l'empresa de programari. Si tenim sort, la llista estarà formada únicament per `Engineers` o bé únicament per `ProjectManagers` o per `DeptDirectors`...

Si, per exemple, la llista fos únicament d'enginyers, podríem fer una cosa de l'estil:

```
1 void sendEmailToEngineers(List<Engineer> leng,
2                          String text, String subject) {
3     for (Engineer eng : leng) {
4         sendEmail(eng.getEmail(), text, subject);
5     }
6 }
7
8 List<Engineer> leng = getListOfEngineers();
9 sendEmailToEngineers(leng);
```

Comentaris:

- Recordeu que el bucle `for(Engineer eng : leng)` és un bucle anomenat *per-cada* (ang: *for-each*) que recorre tots els elements d'una col·lecció i aplica a cadascun d'ells (`eng`) el tractament que indica el cos del bucle.
- En aquest cas, el tractament consisteix a enviar un missatge a aquell enginyer `eng`. L'operació `sendEmail(address,text,subject)`, que no hem desenvolupat, s'encarrega d'enviar un correu electrònic a l'adreça `address`, amb tema `subject` i contingut `text`.

■ ■ ■

És bona la solució que planteja aquest exemple? Penseu-hi un moment.

Si heu pensat bé, haureu vist que aquesta solució planteja, com a mínim, dos problemes fonamentals:

1. Caldria una operació `sendEmailToXXXX(...)` per a cada tipus de treballador diferent. Més encara, si en un futur es creen noves categories de treballadors, per a cadascun d'ells caldria crear una operació `sendEmailToXXXX(...)` apropiada per a aquella categoria nova.
2. Les llistes de treballadors haurien de ser per categories laborals. No hi podria haver llistes de casats, de menors de 40 anys, de majors de 60, d'interessats en la gastronomia...

En definitiva, la solució plantejada és pobra.

Una alternativa fóra tenir únicament una operació que rebés com a paràmetre una llista de treballadors de qualsevol mena. O sigui:

```
1 void sendEmailToEmployees(List<Employee> lemp,
2                          String text, String subject) {
3     for (Employee emp : lemp) {
```

```
4           sendEmail(emp.getEmail(), text, subject);
5       }
6   }
```

Però compte, perquè aleshores, si la llista que tenim és d'`Engineers` tenim problemes:

```
1 List<Engineer> leng = getListOfEngineers();
2 sendEmailToEmployees(leng);     //PROBLEMES!!!!
```

Per què tenim problemes? Efectivament, ho heu encertat (espero): el problema és que `List<Engineer>` *no és subtipus de* `List<Employee>`.

1.4.4 Comodins

Plantegem resoldre això usant com a paràmetre del genèric el caràcter `?` en lloc d'`Employee`. Aquest caràcter `?` actuarà com a *comodí* (en anglès, *wildcard*).

Comodins amb *extends*

Reconsiderem l'operació `sendEmailToEmployees`, ara amb comodins:

```
1 void sendEmailToEmployees(List<? extends Employee> lemp,
2                           String text, String subject) {
3     for (Employee emp : lemp) {
4         sendEmail(emp.getEmail(), text, subject);
5     }
6 }
```

La declaració

```
1 List<? extends Employee> lemp;
```

indica que `lemp` és una llista els elements de la qual són de qualsevol subtipus d'`Employee`. Per tant, `List<Engineer>` és subtipus de `List<? extends Employee>`. Per tant, és correcte:

```
1 List<Engineer> leng = getListOfEngineers();
2 sendEmailToEmployees(leng);
```

En aquest cas, a l'operació `sendEmailToEmployees` *llegia la llista* `lemp` i tot anava bé. Però.... què hagués passat si l'hagués escrit?

Exemple 1.31: Comodins amb* `extends` *i escriptura

```
1 void writeToAnExtendsWildcard(List<? extends Employee> lemp) {
2     lemp.add(new ProjectManager(...));    //PROBLEMES!!!!!
3 }
4
5 List<Engineer> leng = new LinkedList<Engineer>();
6 writeToAnExtendsWildCard(leng);
```

Malament!!! No podem inserir un `ProjectManager` a una llista d'`Engineer`!!!

■ ■ ■

La conclusió d'aquest exemple és la següent:

> Les estructures de dades (com ara llistes) que contenen elements de tipus `? extends T` es poden usar per llegir i/o per treure elements de l'estructura però no per afegir-n'hi.

I si n'hi volem afegir? Aleshores ens caldrà llegir la secció següent.

Comodins amb *super*

De la mateixa manera que `? extends T` indica *qualsevol tipus que sigui subtipus de* `T`, la notació `? super T` indica *qualsevol tipus que sigui supertipus de* `T`. I aquesta darrera és la que ens va bé per afegir elements a una estructura de dades. Considerem l'exemple següent:

Un exemple clàssic de l'ús de `? extends super T` és l'operació `Collections.copy`. Aquesta operació està especificada com segueix:

```
1 public static <T> void copy(List<? super T> dest,
2                             List<? extends T> src)
```

Copia tots els elements de la llista `src` a la llista `dest`.

Observeu, en primer lloc que, per ser una operació estàtica, caldrà cridar-la fent:
`Collections.copy(dest,src);`

Observeu també que la llista que llegim (`src`) conté elements de subtipus de `T` (incloent-hi, evidentment, `T`). I cadascun d'aquests elements d'`src` els escrivim sobre una altra llista (`dst`), els elements de la qual són d'un supertipus de `T` (novament, incloent-hi `T`). *Però qualsevol supertipus de* `T` *és també supertipus d'un subtipus de* `T`; per tant, no hi haurà cap problema en inserir a `dst` qualsevol element d'`src`.

Segurament, aquesta frase darrera convé reflexionar-la una mica i el següent exemple ens hi pot ajudar.

Exemple 1.32: Ús de** super **amb comodins

Suposem que `T` sigui `Engineer`. Aleshores, l'operació `copy` anterior tindrà la forma:

```
1 public static void copy(List<? super Engineer> dest,
2                         List<? extends Engineer> src) { ... }
```

I, aleshores, la podríem cridar de la manera següent:

```
1 List<Engineer> leng = new LinkedList<Engineer>();
2 List<TestEngineer> ltest = new LinkedList<TestEngineer>();
3
4 ltest.add(new TestEngineer(...));
5 Collections.copy(leng, ltest);
```

És cert que qualsevol supertipus d'`Engineer` (e.g., `Engineer`, `Employee`) és també supertipus d'un subtipus d'`Engineer` (e.g. `TestEngineer`)? És clar que sí. I, per tant, ho repetim: no hi haurà cap problema en inserir a `dst` qualsevol element d'`src`.

Finalment, com implementem `copy`?

```
1 public static <T> void copy (List<? super T> dest,
2                              List<? extends T> src) {
3     for (int i=0; i<src.size(); i++) {
4         dest.set(i, src.get(i));
5     }
6 }
```

■ ■ ■

La conclusió d'aquest exemple és la següent:

> Les estructures de dades (com ara llistes) declarades de tal manera que contenen elements de tipus `? super T` es poden usar per afegir elements a l'estructura.

Inferència del tipus que substitueix `T` a les operacions genèriques

Habitualment no és necessari, en invocar una operació genèrica, indicar explícitament el tipus amb el qual es fa la invocació (a l'exemple anterior, `Engineer`). Aquest tipus, quasi sempre, pot ser inferit pel compilador.

En general, el compilador *substitueix* `T` *pel tipus més específic que fa que la crida sigui correcta*.

Vegem un parell d'exemples d'inferència de tipus.

Exemple 1.33:

Recordem l'especificació de l'operació genèrica `Collections.replaceAll` (que reemplaça totes les ocurrències del valor `oldVal` a la llista `list` pel valor `newVal`).

```
1 public static <T> boolean replaceAll(List<T> list,
2                                      T oldVal,
3                                      T newVal)
```

I considerem el codi següent:

```
1 List<Number> lnum = new LinkedList<Number>();
2
3 lnum.add(1);
4 lnum.add(1.3);
5
6 Collections.replaceAll(lnum, 1, 2);
```

En el cas de la crida de la instrucció 6, el compilador infereix que `T` és `Number` perquè `Number` és el tipus més específic que fa que la crida `Collections.replaceAll(lnum, 1, 2)` es pugui fer.

■ ■ ■

Exemple 1.34:

```
1  List<Engineer> leng = new ArrayList<Engineer>();
2  List<TestEngineer> ltest = new ArrayList<TestEngineer>();
3
4  Engineer eng1 = new Engineer(...);
5  TestEngineer test1 = new TestEngineer(...);
6
7  leng.add(eng1);
8  leng.add(test1);
9
10 Collections.replaceAll(leng, test1, eng1);
11 Collections.replaceAll(ltest, test1, test1);
12 Collections.replaceAll(ltest, eng1, test1);    //ATENCIO: MALAMENT!!!
```

Recordem que `TestEngineer` és subtipus d'`Engineer`. Veiem quines inferències fa el compilador:

- Instrucció 10: s'infereix que `T` és `Engineer`.

  La crida `replaceAll(leng,test1,eng1)` conforma amb la declaració `replaceAll(List<Engineer>, Engineer, Engineer)` perquè el tipus de `leng` (`List<Engineer>`) és subtipus de `List<Engineer>`; el tipus de `test1` (`TestEngineer`) és subtipus d'`Engineer` i el tipus de `eng1` (`Engineer`) és subtipus d'`Engineer`.

- Instrucció 11: s'infereix que `T` és `TestEngineer`.

- Instrucció 12: provoca un **error de compilació** perquè no hi ha cap assignació de tipus a `T` que permeti la crida `replaceAll(ltest,eng1,test1)`. Vegem-ho:

- Si `T` és `TestEngineer`, el tipus d'`eng1` (`Engineer`) no és subtipus de `TestEngineer`. Per tant, `T` no pot ser `TestEngineer`.
- Si `T` és `Engineer`, el tipus de `ltest` (`List<TestEngineer>`) no és subtipus de `List<Engineer>`. Per tant, `T` no pot ser `Engineer`.
- El raonament anterior s'aplicaria també si `T` fos `Employee` o `Object`.

■ ■ ■

El mecanisme d'inferència de tipus funciona bé en general, però hi ha algun cas en què no és així. Aleshores, es pot incloure a la crida explícitament el tipus pel qual s'ha de fer la substitució del paràmetre genèric:

```
1 Collection.<Engineer>replaceAll(leng, engOld, engNew);
```

Un cas en què el mecanisme d'inferència de tipus genèrics no funciona correctament es produeix quan tenim crides aniuades a operacions genèriques. Suposem que tenim les declaracions de dues operacions genèriques `f` i `g`:

```
1 public static <T> List<T> f() {...}
2 public static <T> T g(List<T> lis) {...}
```

I les cridem aniuadament:

```
1 String s = g(f());
```

El compilador no és capaç d'inferir que el tipus de `T` ha de ser `String`. En aquest cas, hem d'indicar explícitament que cal fer la substitució de `T` per `String` de la manera següent:

```
1 String s = g(Generics.<String>f());
```

Resum de tot plegat

- `? extends T` indica qualsevol tipus `X` tal que `X` és subtipus de `T`.
- `? super S` indica qualsevol tipus `X` tal que `X` és supertipus d' `S`.
- `EstructuraSub<S>` és subtipus d'`Estructura<? extends T>` si `EstructuraSub<E>` és subtipus d'`Estructura<E>` i `S` és subtipus de `T` (per a qualsevol classe o interfície `E`)

  Ex: LinkedList<Engineer> és subtipus de List<? extends Employee> perquè LinkedList<E> és subtipus de List<E> i Engineer és subtipus de Employee.

- `EstructuraSup<S>` és supertipus d' `Estructura<? super T>` si `EstructuraSup<E>` és supertipus d'`Estructura<E>` i `S` és supertipus de `T` (per a qualsevol classe o interfície `E`)

  Ex: `List<Employee>` és supertipus de `LinkedList<? super Engineer>` perquè `List<E>` és supertipus de `LinkedList<E>` i `Employee` és supertipus d'`Engineer`.

- Usarem `Estructura<? extends T>` quan volem consultar o treure elements de l'estructura.
- Usarem `Estructura<? super T>` quan volem inserir elements a l'estructura.

1.5 Racó lingüístic

En aquesta secció comentem els termes tècnics usats en aquest capítol que no estan estandarditzats al diccionari de l'Institut d'Estudis Catalans, a Termcat o a [CM94].

- *Vector. Matriu.*

  Hem triat aquests termes com a traducció de l'anglès *unidimensional array* i *multidimensional array*, respectivament.

  Ens ha semblat més precís usar un terme diferent per referir-nos a un *unidimensional array* i a un *multidimensional array* perquè el terme *matriu* té clares connotacions multidimensionals en els àmbits de la matemàtica i també de la informàtica. Així i tot, la gran majoria dels *arrays* que s'usen en programació són unidimensionals. Heus ací la conveniència d'usar un terme específic per a ells.

  Tant Termcat com [CM94] tradueixen *array* genèricament per *matriu* cometent, en la nostra opinió, la imprecisió esmentada més amunt.

  El terme *array*, sovint, ha hagut de suportar traduccions infames. En molts textos llatinoamericans es tradueix encara avui per *arreglo*, la qual cosa produeix enrojolament.

- *Conversió de tipus. Coerció.*

  El terme anglès *cast* (o *type cast*) ha estat traduït en aquest text per *conversió de tipus* o també *coerció*. *Conversió de tipus* és una traducció literal, mentre que *coerció* proposa

una traducció figurada però prou intuïtiva del que es pretén. Efectivament, una conversió de tipus suposa un constrenyiment o coerció d'un objecte que formalment és d'un tipus determinat als requeriments d'un altre tipus. De fet, en anglès, en ocasions, s'usa el terme *coercion* per referir-se a la conversió de tipus.

Ni Termcat ni [CM94] proposen una traducció pels termes *type cast* o *coercion*.

- *Conversió de tipus cap avall* o *conversió cap avall* o *coerció cap avall*.

  Traducció proposada pel terme anglès *downcast*. Ni Termcat ni [CM94] proposen una traducció per aquest terme.

- *Comodí*.

  Traducció proposada pel terme anglès *wildcard*, seguint el criteri de Termcat per un concepte relativament similar (caràcter que substitueix un altre caràcter o un conjunt de caràcters). [CM94] no proposa cap traducció per *wildcard*.

- *Paquet*.

  Traducció proposada pel terme anglès *package*, referit a l'agrupació de classes i interfícies relacionades en un determinat espai de noms.

  Termcat no ofereix una traducció específica per *package* en l'àmbit informàtic i reserva el terme *paquet* per referir-se a un *paquet de programari*. Sembla, doncs, raonable i natural generalitzar el terme a una agrupació de components de programari (com classes i interfícies). [CM94] usa el terme *paquet* en un sentit diferent (agrupació de dades considerades com un tot) i no ofereix una traducció per *package*.

- *Tipus genèric*.

  Traducció natural proposada pel terme anglès *generic type*. Ni Termcat ni [CM94] proposen una traducció per aquest terme.

- *Biblioteca*.

  Traducció del terme anglès *library*. El terme *biblioteca* és proposat per Termcat però no per [CM94]. És interessant notar com, sovint, en llibres d'informàtica s'usa, al meu entendre, de manera incorrecta el terme *llibreria* (traducció immediata i dolenta de *library*) en lloc de *biblioteca*.

- *Lligam. Lligam estàtic, lligam dinàmic*

  Traducció del terme anglès *binding* en la seva accepció de relació entre la crida a una operació i el codi associat a aquella operació. Com que aquesta relació, quan s'estableix, té la característica de lligar la crida i el codi de l'operació, ens sembla intuïtiva la traducció per *lligam*.

  El lligam establert en temps de compilació (en anglès, *early binding*) el notem per *lligam estàtic* recollint la connotació que aquest lligam sempre és el mateix. No pot canviar en diferents execucions. El lligam establert en temps d'execució (en anglès, *dynamic binding*, l'anomenem *lligam dinàmic*.

  Ni Termcat ni [CM94] proposen una traducció per aquests termes.

Capítol 2

Estructures de dades d'accés seqüencial

Ara que ja coneixem què són el polimorfisme i els genèrics podem començar a parlar d'estructures de dades. Definirem estructura de dades com una col·lecció d'elements d'un cert tipus genèric `T`. Hi ha diferents tipus d'estructures de dades i, en aquest capítol, comencem parlant de les més simples: *les estructures de dades d'accés seqüencial*. Això és: les llistes, les piles, les cues i les cues dobles (en anglès, *deques*). Entre totes aquestes estructures de dades, la més general és la llista. Les altres es poden veure com a casos particulars d'ella. Per aquest motiu, ens dedicarem, majoritàriament, a estudiar les llistes: les especificarem i les implementarem. I ho farem seguint el *Java Collections Framework (JCF)*, el catàleg de classes i interfícies definides pel Java per treballar amb estructures de dades. El *JCF* ens ensenyarà, entre altres coses, què són els iteradors, i com podem implementar una llista enllaçada.

2.1 Estructures de dades o contenidors

Sovint, la informació que ha de manipular un programa apareix en forma d'una col·lecció d'objectes que tenen sentit unitari com a col·lecció però que cal tractar un a un. Per exemple, la llista dels estudiants del grau en Enginyeria Informàtica o la cua de ciutadans que esperen a les oficines del Registre Civil.

Aquesta col·lecció d'objectes s'anomena *estructura de dades* o també *contenidor* o, simplement, *col·lecció*. A les aplicacions, les estructures de dades les tractem en forma de classes genèriques que depenen d'un paràmetre tipus `T`, que representa el tipus dels objectes que conté l'estructura.

Per exemple, en el cas de la llista d'estudiants, el tipus dels objectes de la llista podria ser `Student` (aquest tipus substituiria `T`). Però, al mateix tipus llista podríem substituir `T` per `Professor` i aleshores tindríem una llista de professors.

Estructura de dades o contenidor:

Col·lecció d'elements d'un cert tipus `T`. Aquests elements són accessibles i manipulables a través d'una sèrie d'operacions:

- Les operacions d'accés i manipulació d'un contenidor permeten (en general): **inserir, eliminar** i **consultar** els seus elements.
- L'estructura dels elements que componen un contenidor i l'estratègia concreta amb què es pot accedir i manipular depenen del tipus específic de contenidor.

Els contenidors els modelitzem a les aplicacions en forma de *classes genèriques* que depenen d'un paràmetre tipus que se substitueix en cada cas pel tipus dels elements que ha de contenir el contenidor.

Aquestes classes genèriques ofereixen les operacions apropiades per accedir i manipular els objectes d'aquell contenidor.

En aquest curs estudiarem diferents famílies de contenidors. Cadascuna d'aquestes famílies oferirà unes característiques i forma de manipulació concretes dels seus elements. O sigui, *cada família oferirà unes operacions concretes.*

Famílies principals de contenidors:

- *Contenidors d'accés seqüencial*

  Piles, cues, llistes, cues dobles (seqüències d'objectes, cadascuna amb unes propietats específiques: vegeu capítol 2).
- *Contenidors arborescents*

  Arbres (vegeu capítol 3)
- *Contenidors d'accés directe per clau*

  Taules (col·leccions de parelles clau-valor: vegeu capítol 4)
- *Altres contenidors*

  Conjunts (en parlarem en diferents capítols)

Com podeu llegir al quadre anterior, aquest capítol està dedicat a les estructures de dades d'accés seqüencial.

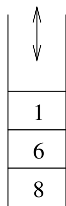


Figura 2.1: L'estructura de dades *pila*

2.1.1 Contenidors d'accés seqüencial

Contenidors d'accés seqüencial: (o estructures de dades d'accés seqüencial).
Contenidors compostos per elements que ocupen una *posició determinada al contenidor*.

Dues conseqüències d'aquesta definició:

- Podem parlar del primer, el segon, el tercer... elements.
- Cada element, llevat del darrer, tindrà un *següent*.

Habitualment es consideren quatre contenidors d'accés seqüencial: *llistes*, *piles*, *cues* i *cues dobles*. En realitat, les piles, les cues i les cues dobles es poden considerar casos particulars de llistes.

Pila:
Estructura de dades LIFO (i.e., el darrer en entrar és el primer en sortir).
Les insercions, eliminacions i consultes es fan *al capdamunt*.

La figura 2.1 mostra gràficament una pila.

Cua:
Estructura de dades FIFO (i.e., el primer en entrar és el primer en sortir).
Les insercions es fan al final de la cua i les eliminacions i consultes, al seu inici.

La figura 2.2 mostra gràficament una cua.

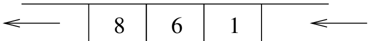


Figura 2.2: L'estructura de dades *cua*

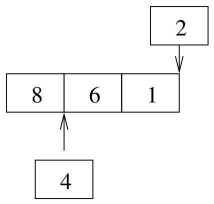
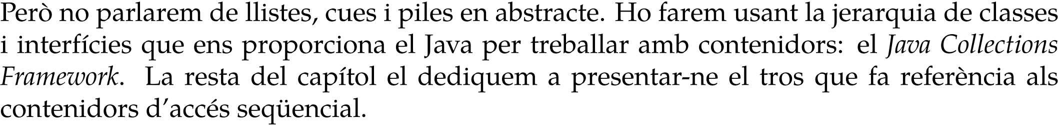


Figura 2.3: L'estructura de dades *llista*



Llista:
Generalització de cues i piles. Seqüència d'elements en què les insercions, eliminacions i consultes es fan en qualsevol posició.

La figura 2.3 mostra gràficament una llista.

Però no parlarem de llistes, cues i piles en abstracte. Ho farem usant la jerarquia de classes i interfícies que ens proporciona el Java per treballar amb contenidors: el *Java Collections Framework*. La resta del capítol el dediquem a presentar-ne el tros que fa referència als contenidors d'accés seqüencial.

2.2 Java Collections Framework: una jerarquia de contenidors

Hem parlat de llistes, cues, piles, taules, arbres, grafs... A més a més, cadascuna d'aquestes estructures es pot representar de maneres diferents a la memòria. Per exemple, en aquest capítol veurem que una llista es pot representar com un vector (en anglès *array*) que conté aquells objectes o com una col·lecció de nodes enllaçats (la figura 2.4 en dóna una idea intuïtiva que anirem refinant al llarg del capítol). En el primer cas, els elements contigus de la llista estan situats en posicions consecutives de memòria. En el segon cas, no: per accedir a l'element següent cal seguir un punter.

Lògicament, cadascuna d'aquestes dues maneres de representar la llista es definirà en una classe diferent (que podem anomenar, respectivament, `ArrayList<T>` i `LinkedList<T>`). I això passarà amb la resta de contenidors.

No és difícil adonar-se que si no organitzem adequadament tots aquests contenidors en una jerarquia de classes i interfícies ben pensada el resultat serà caòtic. Intentem-ho.

Per fer-ho, us proposo adoptar la jerarquia de contenidors que ens proposa el Java en el paquet `java.util` (i també en el `java.util.concurrent`). Aquests paquets contenen una sèrie de classes i interfícies que constitueixen el *Java Collection Framework (JCF)*.

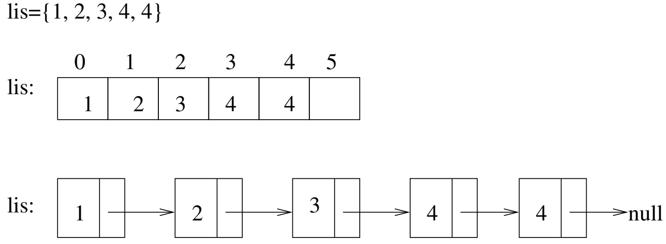


Figura 2.4: ArrayList i LinkedList: dues maneres de representar una llista

2.2.1 Les interfícies del Java Collections Framework

Hem dit més amunt que cada família de contenidors defineix una manera concreta d'accedir als objectes del contenidor, que es manifesta en termes d'una llista d'operacions. Dit d'una altra manera:

> Podem interpretar cada família diferent de contenidors com un tipus diferent.
>
> Com que, en general, la manera més adient i elegant de definir un tipus és mitjançant una interfície, *podem associar una interfície a cada família de contenidors*.
>
> Encara més, podem donar unitat i estructura a totes les interfícies que definirem generant una (o diverses) jerarquies d'interfícies que modelitzaran tipus de contenidors.

Una d'aquestes interfícies és List<T>, que defineix el tipus per a les llistes (seqüències d'objectes) . Algun estudiant poc informat podria preguntar: la interfície List<T> únicament declararà les operacions que ha d'oferir aquest contenidor però no proporcionarà la representació en memòria del contenidor (per exemple, en forma de vector) ni tampoc implementarà aquelles operacions. Si volem que un programa treballi amb un contenidor *llista*, finalment, l'haurem de representar i implementar les seves operacions. Així doncs, per què perdem el temps definint la interfície? Per què no definim directament una classe List<T>? *Què li contestaríeu a aquest estudiant despistat?*

Us proposo dues respostes:

Resposta 1

Hi pot haver, de fet, hi hauran diferents classes que implementaran el tipus (la interfície) List<T> d'acord amb diferents estratègies. Ja ho hem avançat a la figura 2.4: les classes ArrayList<T> i LinkedList<T> implementaran totes dues la interfície List<T>.

Cadascuna d'aquestes classes tindrà característiques diferents que la faran més apropiada per a cada situació concreta en què necessitem una llista. Però *totes dues implementaran la interfície* List<T> *i, per tant, les instàncies d'*ArrayList<T> *i* LinkedList<T> *seran instàncies del tipus* List<T>. Aquesta darrera frase la podem expressar encara d'una altra manera:

`ArrayList<T>` *i* `LinkedList<T>` *seran subtipus del tipus* `List<T>`.

Exemple 2.1:

Si una operació d'un programa (per exemple, anomenem-la `manageList`) ha de gestionar una llista que se li passa per paràmetre, no necessita saber de quina classe concreta és aquella llista:

```
1 public <T> void manageList(List<T> list) {
2     ...
3     list.add(obj);
4     ...
5 }
```

Aquesta operació `manageList(...)` podria ser cridada de la manera següent:

```
1 LinkedList<Student> lstud = new LinkedList<Student>();
2 ArrayList<Student> lstud2 = new ArrayList<Student>();
3
4 manageList(lstud);
5 manageList(lstud2);
```

I podrà aplicar sobre el paràmetre `list` qualsevol operació definida a la interfície `List<T>`. Per exemple, l'operació `add`.

L'operació `manageList`, dissenyada d'aquesta manera, és més general, no usa informació que no necessita estrictament, dóna més llibertat als seus usuaris, permet ser usada en contextos més amplis, no cal modificar-la si en un futur es crida amb un tipus diferent de llista. Calen més arguments?

■ ■ ■

Resposta 2

Definir `List<T>` com a interfície ens permet definir tipus que continguin les operacions de més d'un tipus ja definits.

Exemple 2.2: Contenidor llista conjunt

Imaginem que volem definir un contenidor `ListAndSet<T>` que es comporti com una llista (interfície `List<T>`) i com un conjunt (interfície `Set`). Podem definir-lo de la manera següent:

```
1 class ListAndSet<T> implements List<T>, Set<T> {
2     ...
3 }
```

Si `List<T>` i `Set<T>` fossin classes, això no seria possible:

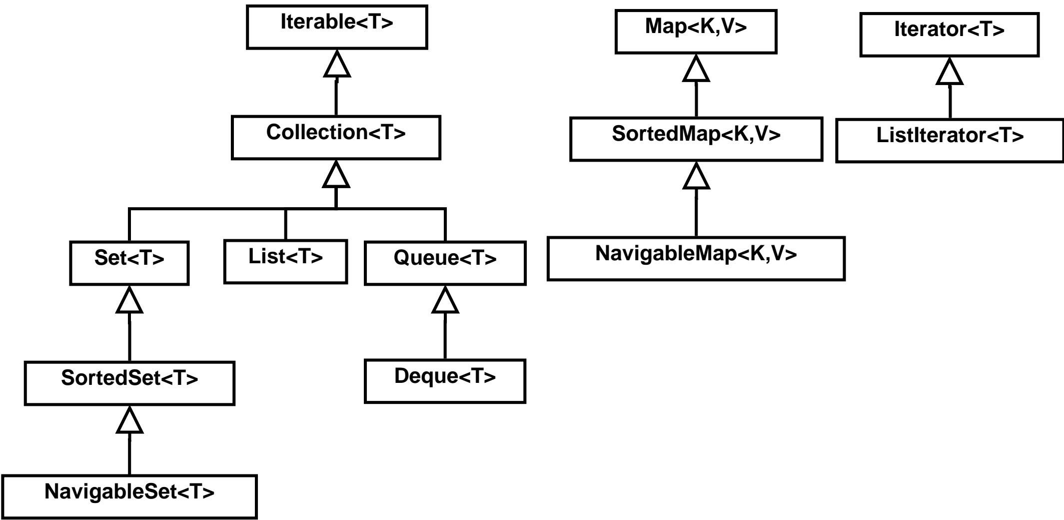


Figura 2.5: Algunes interfícies del JCF

Llistat 2.1: MALAMENT. UNA CLASSE NO POT TENIR DUES SUPERCLASSES!!!

```
1 class ListAndSet<T> extends List<T>, Set<T> {    //MALAMENT!!!!
2       ...
3 }
```

■ ■ ■

Ja és hora que donem un cop d'ull a les interfícies més importants definides al JCF. La figura 2.5 les presenta. Podem observar que aquestes interfícies estan estructurades en dues jerarquies: la que té com a arrel `Iterable<T>` i la que té com a arrel `Map<T>`.

En aquest capítol estem interessats en les estructures de dades d'accés seqüencial, que típicament estan constituïdes per les llistes i els seus casos particulars: les piles, les cues i altres estructures derivades com ara les cues dobles. Totes aquestes estructures es descriuen en la jerarquia de la JCF arrelada a `Iterable<T>`. Per tant, en aquest capítol ens centrarem en aquesta jerarquia. La que està arrelada a `Map<T>` la deixarem per al capítol 4.

2.3 Iterables i iteradors

Una de les característiques més rellevants d'un contenidor d'accés seqüencial és que tots els seus elements (llevat del darrer) tenen un següent. Dit d'una altra manera, podem recórrer (o iterar) els elements del contenidor accedint al *primer* i, a partir d'allí, obtenint el *següent* tots els cops que calgui fins que ja hàgim visitat tots els elements del contenidor.

Tots els contenidors d'accés seqüencial són *recorribles* (o *iterables*) en l'ordre natural donat per la posició dels seus elements.

A la JCF, totes les interfícies que modelitzen contenidors d'accés seqüencial són descendents de la interfície `Iterable<T>`.

A la JCF, les interfícies més importants que modelitzen contenidors d'accés seqüencial són `List<T>`, `Queue<T>` i `Deque<T>` (vegeu figura 2.5).
Aquestes interfícies són implementades per classes que defineixen contenidors d'accés seqüencial amb una representació i implementació concreta.
Exemple: `AbstractList<T>` i `LinkedList<T>` implementen `List<T>`.

La JCF conté contenidors d'accés seqüencial *que no estan definits com a interfície sinó directament com a classe*.
Exemple: les classes `Stack<T>`, `Vector<T>`

Com hem explicat al darrer punt del quadre anterior, la JCF proposa alguns contenidors d'accés seqüencial que no estan definits com a interfície sinó directament com a classe. Un exemple és el de la classe `Stack<T>`:

```
1 class Stack<T> implements List<T> ....
```

Per què la JCF fa aquesta definició?

Una raó pràctica anima a fer-ho d'aquesta manera: si es definís `Stack<T>` com a interfície, la JCF només proporcionaria una classe que la implementés (per exemple, `ArrayStack<T>`) ja que no és gaire probable que els usuaris del Java tinguin la necessitat d'usar una implementació diferent d'`Stack<T>` de la que ja ofereix la JCF. Possiblement per això es va decidir no definir la interfície `Stack<T>`.

Noteu que la majoria de les interfícies de la JCF són implementades per més d'una classe (per exemple, `List<T>` és implementada per `ArrayList<T>`, `LinkedList<T>` i la mateixa `Stack<T>`, entre d'altres). Alguna interfície, com ara `SortedMap<K,V>`, és només implementada per una classe (i.e., `TreeMap<K.V>`). Al capítol 4 estudiarem que un `SortedMap<K,V>` es pot implementar de moltes maneres diferents. Per tant, és possible que un usuari necessiti, en un moment determinat, treballar amb un `SortedMap<K,V>` diferent de `TreeMap<K,V>`. Si fos aquest el cas, podria definir una nova classe (e.g., `AVLMap<K,V>`) que implementés `SortedMap<K,V>`. Igualment, els dissenyadors del Java poden voler afegir, en un futur, una nova implementació de `SortedMap<K,V>` a la JCF. Per això sembla prudent definir `SortedMap<K,V>` com a interfície.

En qualsevol cas, en la nostra opinió, la definició d'`Stack<T>` també com a interfície hagués proporcionat un disseny més uniforme, elegant i extensible de la JCF.

El quadre anterior dóna una idea intuïtiva de que *les interfícies que modelitzen contenidors d'accés seqüencial són subinterfícies d'*Iterable<T>. En llegir-ho, algun estudiant podria pensar erròniament que el recíproc és cert. O sigui, tota subinterfície d'Iterable<T> és un contenidor d'accés seqüencial. Però aquesta idea no és certa.
Com hem dit, la interfície Iterable<T> és *superinterfície* dels tipus que defineixen contenidors d'accés seqüencials però també ho és d'un contenidor que no és d'accés seqüencial: Set<T>.
El tipus Set<T> modelitza els conjunts d'elements de tipus T. Com sabem, els conjunts són col·leccions d'elements no repetits i dotats d'operacions tals com *pertany, unió, intersecció ....* El concepte matemàtic de *conjunt* no imposa cap posició als elements que formen part del conjunt. No té sentit dir: l'element 3 ocupa la primera posició del conjunt. Per això, el tipus Set<T> *no ofereix cap operació de l'estil* Set<T>.get(position), que obtindria l'element d'un conjunt que ocupés la posició position. Insistim, aquesta operació **no és oferida per** Set<T>.
Noteu que això és ben diferent del cas d'una llista, que es defineix com a seqüència d'elements (amb possibles repeticions) i, per tant, existeix un primer, un segon elements... I existeix, en conseqüència, l'operació List<T>.get(position), que obté l'element de la llista que ocupa la posició position.
És molt natural que els contenidors d'accés seqüencial, els elements dels quals ocupen una posició dins del contenidor, siguin iterables, però potser ens pot estranyar una mica que un contenidor, com ara el Set<T>, els elements del qual no tenen posició definida, sigui també iterable (subinterfície d'Iterable<T>).
Se us acudeix per què la JCF ho defineix d'aquesta manera?

Hi ha una raó pràctica: tot i que el tipus Set<T> no fixi una posició per a cada element, sovint, manipulant conjunts ens fem preguntes com les següents: *Tots els elements d'aquest conjunt d'enters són parells?*, *Hi ha algun element d'aquest conjunt d'estudiants al qual li faltin menys de 30 ECTS per acabar el grau?*, *M'agradaria obtenir un llistat amb les matrícules i els nifs dels propietaris de tots els elements d'aquest conjunt de vehicles*.
Veiem doncs que, sovint, es necessita recórrer els elements d'un conjunt per aplicar un tractament a tots ells o esbrinar quins compleixen una determinada propietat. Per això, tot i no ser estrictament un contenidor *d'accés seqüencial*, és un contenidor *iterable*.

2.3.1 La interfície Iterable<T>

La interfície Iterable<T> proporciona un tipus que comparteixen tots aquells contenidors que poden ser *recorreguts* o *iterats*.

Per tant, podem pensar en un Iterable<T> com un **contenidor d'objectes de tipus** T **que pot ser recorregut (iterat).**

En particular, ja hem vist que els contenidors d'accés seqüencial són Iterable<T>.

Els objectes que són instància d'una classe que implementa la interfície Iterable<T> poden ser recorreguts amb un bucle *foreach*.

La interfície java.lang.Iterable no està definida al paquet java.util o java.util.concurrent com és el cas de la resta d'interfícies de la JCF.

Exemple 2.3: Bucle** foreach **per recórrer un objecte iterable

Llistat 2.2: Iteració d'un Iterable<T> amb bucle foreach

```
1 <T> void traversal(Iterable<T> col) {
2     for (T obj : col) {
```

```
3           System.out.println(obj.toString());
4       }
5   }
```

Comentaris:

- El bucle *foreach* itera sobre tots els elements del contenidor iterable col. A cadascun d'aquests elements (obj) li aplica el tractament de mostrar-lo per la sortida estàndard. Notem que obj és de tipus T perquè el contenidor col conté elements precisament d'aquest tipus.

■ ■ ■

La interfície Iterable<T> ofereix únicament una operació:

```
1   Iterator<T> iterator()
```

Retorna un iterador sobre una col·lecció d'elements de tipus T.

L'única operació d'Iterable<T> introdueix un concepte relacionat amb Iterable<T> però que és nou i del qual cal parlar amb detall: la interfície Iterator<T>.

2.3.2 La interfície Iterator<T>

L'operació Iterable<T>.iterator() retorna un objecte de tipus Iterator<T>. Però, què és un Iterator<T>?

Un objecte de tipus Iterator<T> permet recórrer un objecte Iterable<T>.

Com que els **contenidors d'accés seqüencial són** Iterable<T>, poden ser recorreguts mitjançant Iterator<T>.

Les operacions que ofereix la interfície Iterator<T> per tal de recórrer Iterable<T> són les presentades a la taula següent:

boolean hasNext() Retorna cert si encara queden elements a l'Iterable<T> per recórrer.
E next() Retorna el següent element de l'Iterable<T> per recórrer.
void remove() Elimina de l'Iterable<T> que està essent recorregut el darrer element retornat per aquest objecte Iterator<T>. Aquesta és una operació opcional. Hi pot haver classes que implementin Iterator<T> i que no l'ofereixin.

Proposem ara diversos exemples d'ús i d'implementació d'iteradors.

***Exemple 2.4: Un exemple d'ús d'*`Iterator<T>`

Vegem com podem implementar la iteració sobre un contenidor `Iterable<T>` del llistat 2.2 usant ara objectes `Iterator<T>`:

Llistat 2.3: Iteració d'un `Iterable<T>` amb iteradors

```
1 <T> void traversal(Iterable<T> col) {
2     Iterator<T> ite = col.iterator();
3 
4     while (ite.hasNext()) {
5         System.out.println(ite.next().toString());
6     }
7 }
```

Comentaris:

- Si comparem aquest codi amb el de 2.2 veurem que aquell és clarament superior. El mateix bucle *foreach* de 2.2 s'encarrega de declarar i gestionar els iteradors necessaris, així que al programador no li cal ni pensar-hi.

  De tota manera, l'ús dels iteradors no es limita a implementar bucles *foreach*. Veurem exemples durant el curs de treball amb iteradors.

- Noteu que aquest codi, igual que el presentat a 2.2, *es pot aplicar a qualsevol objecte que implementi la interfície* `Iterable<T>`. En particular a qualsevol `Collection<T>`: llistes, cues, conjunts, piles... La mateixa operació `traversal` funcionarà bé independentment de quin tipus de contenidor s'hagi de recórrer. L'únic requeriment és que aquell contenidor implementi `Iterable<T>`. Podeu veure com n'és de potent aquest codi?

- Un codi alternatiu al plantejat al llistat 2.3 és el següent:

```
1 <T> void traversal(Iterable<T> col) {
2     for (Iterator<T> ite = col.iterator(); ite.hasNext(); ) {
3         System.out.println(ite.next().toString());
4     }
5 }
```

  Aquest codi té dos avantatges respecte al de 2.3 i un inconvenient. L'inconvenient és que resulta més críptic. Els dos avantatges són que és més compacte i, en estar els iteradors limitats al codi del bucle, es minimitzen les possibilitats d'error.

■ ■ ■

Durant el curs pensarem (quasi) sempre en un `Iterable<T>` com un contenidor que emmagatzema una col·lecció d'objectes de tipus `T` dins d'alguna estructura en memòria. Però podem pensar-hi d'altres maneres. Per exemple: un `Iterable<T>` pot ser un *comptador enter*. Un iterador sobre aquest comptador anirà incrementant una unitat el comptador. Un altre exemple pot ser una *successió en termes matemàtics*, de manera que anem obtenint els

diferents termes de la successió seqüencialment: $a_1, a_2, ..., a_{n-1}, a_n$. Vegem com podem implementar tots dos `Iterables`.

Exemple 2.5: Exemple d'implementació d'*`Iterator<T>`*: el comptador

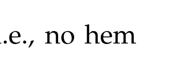
L'objectiu d'aquest exemple és implementar una classe `IntegerCounter` que implementi `Iterable<T>` i que permeti comptar enters fins a un màxim determinat en el moment de crear l'`IntegerCounter`.

La primera idea per resoldre aquest problema és la de dissenyar dues classes amb la forma següent:

```
1  public class IntegerCounter implements Iterable<Integer> {
2      private int max;
3
4      public IntegerCounter(int max) {
5          this.max = max;
6      }
7
8      public Iterator<Integer> iterator() {
9          return new IntCounterIterator(this);
10     }
11
12     public int getMaximum() {
13         return this.max;
14     }
15 }
16
17 public class IntCounterIterator implements Iterator<Integer> {
18     private int current;
19     private IntegerCounter counter;
20
21     public CounterIterator(IntegerCounter counter) {
22         this.current = 0;
23         this.counter = counter;
24     }
25
26     public Integer next() {
27         if (this.current >= counter.getMaximum())
28             throw new NoSuchElementException();
29         this.current++;
30         return new Integer(this.current);
31     }
32
33     public boolean hasNext() {
34         return this.current < counter.getMaximum();
35     }
36
37     public void remove() {
38         throw new UnsupportedOperationException();
39     }
40 }
```

Comentaris:

- Considerem la classe `IntegerCounter` com una classe d'objectes *iterables*. Per tant, haurà d'implementar la interfície `Iterable<Integer>` i, amb ella, l'operació `iterator()` que retornarà un objecte iterador sobre un objecte `IntegerCounter`.
- Definirem l'objecte iterador retornat per `iterator()` com un objecte de la classe `IntCounterIterator`, la qual, evidentment, haurà d'implementar la interfície `Iterator<Integer>`. L'objectiu d'aquesta classe és retornar-nos els diferents valors que pot prendre l'objecte comptador (`IntegerCounter`).
- La classe `IntCounterIterator` es representa mitjançant:
  - Un enter (`current`) que ens indica quin és el valor actual del comptador.
  - Una referència (`counter`) al comptador sobre el qual estem iterant. Per què creieu que és necessària aquesta referència? (mireu el codi).



- L'operació `hasNext()` comprova si encara podem continuar comptant (i.e., no hem arribat al màxim) i `next()` obté el següent valor del comptador.

Observem que les classes `IntegerCounter` i `IntCounterIterator` estan molt entrelligades.

- És estrictament necessari que ningú més que la pròpia `IntegerCounter` conegui l'existència de la classe concreta que usarà com a iterador (i.e., `IntCounterIterator`)?

  A l'usuari d'`IntegerCounter` només li interessa saber que aquesta classe li ha retornat un iterador sobre el qual pot fer `hasNext()`, `next()` i, potser, `remove()`. No li interessa *quina classe específica li han retornat per poder fer això*. Aquesta mateixa idea l'hem repetida en molts contextos diferents.

- Atès que estan tan entrellaçades, en algun cas pot resultar convenient que la classe iterador (i.e., `IntCounterIterator`) tingui accés a la part privada de la classe iterable (i.e., `IntegerCounter`).

  En l'exemple anterior, l'iterador havia d'accedir al valor màxim del comptador. Això ho hem resolt fàcilment amb una crida a l'operació `getMaximum()` (vegeu línies 27 i 34). En altres casos, però, el contenidor serà més complex i les seves operacions no seran les apropiades per tal que l'iterador pugui avançar per l'iterable. En aquests casos més complexos, a l'iterador li serà molt convenient poder accedir a la part privada del contenidor, la qual cosa és impossible amb la solució plantejada.

  Un exemple en què aquesta idea es veu més clara el trobarem a la secció 2.7.

> La manera de millorar el disseny, tot resolent les dificultats plantejades, consisteix a definir `IntCounterIterator` com a *classe interna (*inner class*) de* `IntegerCounter`:
>
> Com a classe interna a `IntegerCounter` podrà accedir a la seva part privada i, precísament per ser una classe interna, ningú més no sabrà que existeix.

La implementació amb aquesta millora queda així:

```
1  public class IntegerCounter implements Iterable<Integer> {
2      private final int MAX = 100;
3      private int max;
4
5      public IntegerCounter() {
6          this.max = MAX;
7      }
8
9      public IntegerCounter(int max) {
10         this.max = max;
11     }
12
13     public Iterator<Integer> iterator() {
14         return new CounterIterator();
15     }
16
17     private class CounterIterator implements Iterator<Integer> {
18         private int current;
19
20         public CounterIterator() {
21             this.current = 0;
22         }
23
24         public Integer next() {
25             if (this.current >= max)
26                 throw new NoSuchElementException();
27             current++;
28             return new Integer(this.current);
29         }
30
31         public boolean hasNext() {
32             return this.current < max;
33         }
34
35         public void remove() {
36             throw new UnsupportedOperationException();
37         }
38     }
39 }
```

■ ■ ■

2.4 Col·leccions

La interfície `java.util.Collection<T>`

La interfície `java.util.Collection<T>` és una subinterfície de `java.lang.Iterable<T>` que ofereix les operacions necessàries per gestionar una col·lecció general d'objectes.

- *Gestionar una col·lecció d'objectes* vol dir *afegir-hi objectes, treure'n, consultar-los...*
- Que es tracti d'una *col·lecció general d'objectes* vol dir que la interfície `Collection` no fa cap hipòtesi sobre quin tipus específic de col·lecció és i, per tant, només ofereix operacions generals que es puguin aplicar a *qualsevol tipus de col·lecció*.

Un exemple del darrer aspecte indicat al quadre anterior és el següent:

En un `Set<T>` (*conjunt*) el concepte *posició d'un element del conjunt* no té sentit: en els conjunts, els elements no tenen definida una posició. En canvi, en una `List<T>`, cada element ocupa una posició determinada. Per tant té sentit parlar de l'element de la posició n. Ara bé, `Collection<T>` és superinterfície de `Set<T>` i també de `List<T>`. Per tant, l'operació:

```
1 void  add (int position, T element)
```

que afegiria l'element `element` a la posició `position` d'aquesta col·lecció **no està definida**. Efectivament, aquesta operació no tindria cap sentit per a la interfície `Set<T>`, que l'heretaria.

En canvi, la interfície `Collection<T>` sí que defineix una operació `add` més general que és igualment aplicable a llistes i a conjunts.

```
1 boolean add(T element)
```

amb l'especificació (simplificada) següent:

L'operació `boolean add(T element)` *garanteix que* `element` *és un element d'aquesta col·lecció. Retorna* `true` *si aquesta col·lecció canvia com a conseqüència de l'operació i* `false` *si no ho fa.*

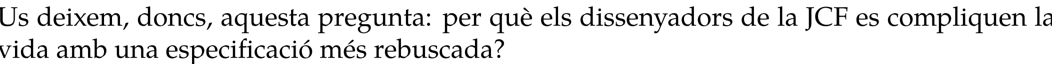
L'especificació completa de l'operació la presentarem més endavant.

Hauríeu especificat l'operació `add(element)` d'aquesta manera? No us sembla una mica rebuscada? No hauríeu preferit una cosa de l'estil:

L'operació `void add(T element)` *afegeix* `element` *a aquesta col·lecció.*

(Incorrecta: aquesta no és l'especificació de Collection<T>.add(element)!!!)

Aquesta especificació alternativa (incorrecta, insistim!!!) no retorna cap `boolean` i indica simplement que l'operació `add` ha afegit un nou element a la col·lecció.



Us deixem, doncs, aquesta pregunta: per què els dissenyadors de la JCF es compliquen la vida amb una especificació més rebuscada?

Selecció de col·leccions

Les reflexions que hem fet fins ara sobre la interfície `Collection<T>` ens porten a un criteri interessant sobre *quan hem d'usar la interfície* `Collection<T>` *en lloc d'altres*:

> Useu la interfície `Collection<T>` en aquells casos en què cal tractar amb un contenidor general d'elements sense que ens importi de quin tipus específic és aquell contenidor (e.g., no ens importa si el contenidor és una pila, una cua, una llista o un conjunt...).
>
> En general, és preferible declarar un objecte del tipus més general (més elevat a la jerarquia de tipus) que permeti aplicar sobre aquell objecte les operacions que necessitem aplicar-li.

Vegem un exemple de l'ensenyament que ofereix aquest quadre:

Exemple 2.6: Selecció del tipus més general possible

El llistat 2.2, a la secció 2.3.1, presentava una operació que mostrava per a la sortida estàndard el contingut d'un contenidor. Era aquesta:

```
1 <T> void traversal(Iterable<T> col) {
2     for (T obj : col) {
3         System.out.println(obj.toString());
4     }
5 }
```

Fem-hi ara un petit canvi: substituïm el tipus del paràmetre i fem que aquest tipus sigui `List<T>`:

```
1 <T> void traversal2(List<T> lis) {
2     for (T obj : lis) {
3         System.out.println(obj.toString());
4     }
5 }
```

Quina de les dues operacions us sembla més adient?

Noteu que l'únic que fa l'operació `traversal` és iterar pels elements d'una llista mitjançant un bucle *foreach*. Aquest bucle es pot aplicar a qualsevol `Iterable<T>` (o sigui, a qualsevol tipus que tingui iteradors definits). `List<T>` és un descendent d'`Iterable<T>` i, per tant, li podem aplicar el bucle *foreach*. Així doncs, `traversal2` és correcte... però no és òptim:

l'operació traversal2 només és aplicable a List<T> mentre que traversal és aplicable a qualsevol Iterable<T>. En particular, és aplicable a List<T> però també ho és a Set<T> o a qualsevol altre Iterable<T> que puguem definir en un futur.

> traversal(...) és més general que traversal2(...) i, habitualment, preferible.

En aquest cas, hem trobat un tipus (Iterable<T>) encara més general que Collection<T> que passar per paràmetre de l'operació.

■ ■ ■

2.4.1 Operacions de la interfície Collection<T>

Dediquem aquesta secció a presentar les operacions de la interfície Collection<T> que ens semblen més importants:

boolean add(T obj) Garanteix que aquesta col·lecció conté l'element obj (operació opcional).
boolean addAll(Collection<? extends T> c) Afegeix tots els elements de c a aquesta col·lecció (operació opcional).
void clear() Elimina tots els elements d'aquesta col·lecció (operació opcional).
boolean contains(Object obj) Retorna true si aquesta col·lecció conté l'element obj. Més formalment, retorna true si i només si aquesta col·lecció conté almenys un element e de manera que (si (obj==null), e==null; si no obj.equals(e)).
boolean containsAll(Collection<?> c) Retorna true si aquesta col·lecció conté tots els elements de c.
boolean equals(Object obj) Compara aquesta col·lecció amb obj per igualtat.
boolean isEmpty() Retorna true si aquesta col·lecció no conté cap element.
Iterator<T> iterator() Retorna un iterador sobre els elements d'aquesta col·lecció.
boolean remove(Object obj) Elimina una instància d'obj d'aquesta col·lecció si hi és present. Més formalment, elimina un element e de manera que (si (obj==null), e==null; si no obj.equals(e)). Retorna true si aquesta col·lecció s'ha modificat com a conseqüència de l'operació (operació opcional).
boolean removeAll(Collection<?> c) Elimina tots els elements d'aquesta col·lecció que també estan continguts a c (operació opcional).
int size() Retorna el nombre d'elements d'aquesta col·lecció.
Object[] toArray() Retorna un vector que conté tots els elements d'aquesta col·lecció.

Les especificacions de les operacions que apareixen en aquesta taula són molt abreujades. No són les especificacions completes. En particular, no indiquen la resposta de les operacions davant de situacions d'error.

És important que, abans d'usar una operació definida a l'API (*Application Programming Interface*) del Java, doneu un cop d'ull a l'especificació completa d'aquella operació. Per exemple, l'especificació completa de Collection<T> la podeu trobar a:

http://download.oracle.com/javase/7/docs/api/java/util/Collection.html

En general, n'hi ha prou de cercar a l'internet la cadena *java api nom-classe* per obtenir aquesta especificació.

L'especificació de la totalitat de l'API de la JSE v. 7 (*Java Standard Edition, version 7*) la podeu trobar a:
`http://download.oracle.com/javase/7/docs/api/`

Exemple 2.7: Especificació completa de l'operació `add(obj)`

Acabem aquesta secció, mostrant, com a exemple, l'especificació completa de l'operació `boolean add(T obj)` extreta de l'especificació de l'API del Java:

`boolean add(E o)`
Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)
Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.
If a collection refuses to add a particular element for any reason other than that it already contains the element, it must throw an exception (rather than returning false). This preserves the invariant that a collection always contains the specified element after this call returns.

- **Parameters:**
  - `o`: element whose presence in this collection is to be ensured.
- **Returns:**
  true if this collection changed as a result of the call
- **Throws:**
  - `UnsupportedOperationException`:
    `add` is not supported by this collection.
  - `ClassCastException`:
    Class of the specified element prevents it from being added to this collection.
  - `NullPointerException`:
    If the specified element is null and this collection does not support null elements.
  - `IllegalArgumentException`:
    Some aspect of this element prevents it from being added to this collection.

■ ■ ■

Algunes operacions de la interfície `Collection<T>` estan definides en termes de l'operació `Object.equals(Object obj)`. En particular, `remove(Object obj)` i `contains(Object obj)`. La primera elimina un element `e` de la col·lecció *igual* a `obj` i la segona comprova si la col·lecció conté un element `e` *igual* a `obj`. En tots dos casos, el criteri usat per a la igualtat és el que proporciona l'operació `e.equals(obj)`. Atès que el paràmetre que rep `equals` és de tipus `Object`, l'API del Java especifica les operacions `remove` i `contains` amb un paràmetre `Object` en lloc de: `remove(T x)` o `contains(T x)` (on `T` representa el paràmetre del genèric, això és: el tipus dels components de la llista). Serà responsabilitat de la pròpia operació `equals` determinar sota quines condicions el paràmetre `obj` és igual a un determinat element de la llista. Habitualment, si el tipus del paràmetre `obj` no és `T`, `e.equals(obj)` retornarà fals, però l'operació `equals` podria actuar de manera diferent.
En definitiva, usant el paràmetre `Object` en lloc de `T` no posem restriccions innecessàries a les operacions `contains(Object obj)` o `remove(Object obj)`.

2.5 Llistes. Especificació

La interfície `java.util.List<T>`

La interfície `java.util.List<T>` és una subinterfície de `java.lang.Collection<T>` que ofereix les operacions necessàries per gestionar una llista d'objectes.

Recordem la definició de llista que hem donat a la secció 2.1.1:

Una **llista** és una col·lecció en què cadascun dels objectes que la componen ocupa una *posició determinada en la col·lecció*.

A més a més, les insercions, eliminacions i consultes es poden fer a qualsevol posició de la llista.

Així doncs, la interfície `java.util.List<T>` afegeix operacions a les que hereta de `java.util.Collection<T>` per tal de:

- Introduir el concepte de posició en una col·lecció.
- Permetre fer insercions, eliminacions o consultes d'elements a qualsevol punt de la llista.

Recordem de la secció 2.1.1 que les llistes les podíem veure com a generalitzacions de les piles i les cues en què insercions, eliminacions i consultes només es feien als extrems del contenidor (Piles: insercions, eliminacions i consultes a l'inici. Cues: insercions al final i consultes i eliminacions, a l'inici).

Vegem quines són les més importants d'aquestes operacions:

<table>
<tr><td>boolean add(E o)
Appends the specified element to the end of this list (optional operation).</td></tr>
<tr><td>void add(int index, E element)
Inserts the specified element at the specified position in this list (optional operation).</td></tr>
<tr><td>boolean addAll(int index, Collection<? extends E> c)
Inserts all of the elements in the specified collection into this list at the specified position (optional operation).</td></tr>
<tr><td>E get(int index)
Returns the element at the specified position in this list.</td></tr>
<tr><td>int indexOf(Object o)
Returns the index in this list of the first occurrence of the specified element,
or -1 if this list does not contain this element.</td></tr>
<tr><td>int lastIndexOf(Object o)
Returns the index in this list of the last occurrence of the specified element,
or -1 if this list does not contain this element.</td></tr>
<tr><td>E remove(int index)
Removes the element at the specified position in this list (optional operation).</td></tr>
<tr><td>boolean remove(Object o)
Removes the first occurrence in this list of the specified element (optional operation).</td></tr>
<tr><td>E set(int index, E element)
Replaces the element at the specified position in this list with the specified element (optional operation).</td></tr>
<tr><td>List<T> subList(int fromIndex, int toIndex)
Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.</td></tr>
<tr><td>Iterator<T> iterator()
Returns an iterator over the elements in this list in proper sequence.</td></tr>
<tr><td>ListIterator<T> listIterator()
Returns a list iterator of the elements in this list (in proper sequence).</td></tr>
<tr><td>ListIterator<T> listIterator(int index)
Returns a list iterator of the elements in this list (in proper sequence),
starting at the specified position in this list.</td></tr>
</table>

Comentaris sobre la llista d'operacions presentada:

- Noteu que, a més a més, la interfície `List<T>` hereta les operacions definides a `Collection<T>`.
- Algunes operacions, com ara `add(...)`, `iterator()...` són heretades per `List<T>` de les seves superinterfícies (`Collection<T>` i `Iterable<T>`). Així i tot, les afegim aquí perquè `List<T>` les fa més específiques, les concreta més. Posem un exemple:

  ***Exemple 2.8: Especificacions diferents d'*`add(obj)` *a les interfícies* `Collection<T>` *i* `List<T>`.**

  L'operació `add(obj)`, com altres operacions, presenta especificacions diferents de `Collection<T>` i a la seva subinterfície `List<T>`. L'especificació de `List.add(obj)` és més específica. Per aquest motiu aquesta especificació s'ha de redefinir a `List<T>`.

  – `Collection.add(obj)`:

    `boolean add(T obj)`
    Assegura que aquesta col·lecció conté l'objecte `obj`.

  – `List.add(obj)`:

    `boolean add(T obj)`
    Afegeix `obj` al final d'aquesta llista.

  ■ ■ ■

- Les operacions relacionades amb la iteració de llistes (`iterator()`, `listIterator()` i `listIterator(i)`) mereixen una atenció especial:
  - Per un costat, `iterator()` redefineix el sentit de l'operació `Iterable<T>.iterator()` heretada.
  - Per l'altre, `listIterator()` i `listIterator(i)` introdueixen una interfície nova (`ListIterator<T>`).

  La secció 2.5.2 la dediquem a explicar amb més detall la iteració de les llistes.

2.5.1 Un exemple d'ús de llistes

Llista d'employees

***Exemple 2.9: Una llista es pot iterar amb un bucle** foreach*

L'operació `finishProject(...)` cancel·la el projecte `project` a cadascun dels enginyers que el tenien assignat. En donem dues versions:

- La primera aprofita el fet que una llista és *iterable* i la recorre amb un bucle *foreach*.

```
1 void finishProject(List<? extends Employee> lemp,
2                     Project project) {
3     for (Employee emp: lemp) {
4         if (emp instanceof Engineer
5                 && ((Engineer) emp).getProject()
6                                     .equals(project))
7             ((Engineer) emp).setProject(null);
8     }
9 }
```

- La segona usa les operacions `size()` i `get(pos)` de la llista per tal de recórrer-la.

```
1 void finishProject2(List<? extends Employee> lemp,
2                      Project project) {
3     for (int i = 0; i< lemp.size(); i++) {
4         Employee emp = lemp.get(i);
5
6         if (emp instanceof Engineer
7                 && ((Engineer) emp).getProject()
8                                     .equals(project))
9             ((Engineer) emp).setProject(null);
10  }
11 }
```

Quina de les dues operacions us sembla més eficient? Pot dependre aquesta eficiència de la representació de la llista `lemp` que les operacions reben per paràmetre?

■ ■ ■

Aprofundim tot seguit en les llistes com a objectes iterables.

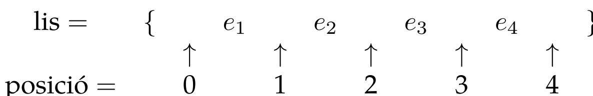
2.5.2 Iteradors sobre llistes: `Iterator<T>` i `ListIterator<T>`

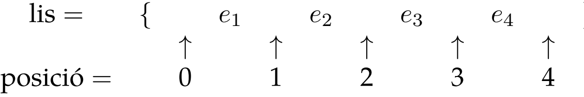
Els objectes instància de `List<T>` són `Iterable<T>` i, per tant, es poden recórrer amb l'ajut d'iteradors.

> Un iterador sobre una llista la recorrerà en l'ordre natural que ocupen aquells elements a la llista (o en ordre invers).
>
> Ens podem imaginar els iteradors com a objectes que **assenyalen una posició entre dos elements de la llista.**
>
> La posició que ocupa un iterador en una llista correspon a l'índex del següent element de la llista (o correspon a `size()` si l'iterador assenyala la posició després del darrer element o si la llista és buida).
>
> Una llista d'n elements té $n+1$ posicions possibles per a un iterador que la recorri: $0..n$.

Exemple 2.10: Posicions vàlides d'un iterador sobre una llista

Considerem una llista amb 4 elements. Un iterador pot prendre 5 posicions diferents en aquesta llista (0..4). Cadascuna d'aquestes posicions coincideix amb l'índex del proper element que serà retornat per l'iterador mitjançant l'operació `next()` excepte per la posició 4. `4` és una posició vàlida per a un iterador tot i que no hi ha cap element de la llista amb índex = 4.





lis =	{		e_1		e_2		e_3		e_4		}
		↑		↑		↑		↑		↑	
posició =		0		1		2		3		4	

■ ■ ■

Exemple 2.11: Iteradors sobre una llista buida

Una llista buida té una única posició vàlida per a iteradors.

lis =	{		}
		↑	
posició =		0	

■ ■ ■

Tenint en compte això, estudiem ara les tres operacions proporcionades per la interfície `List<T>` i que ens permeten obtenir un iterador sobre una llista:

- `Iterator<T> iterator()`
- `ListIterator<T> listIterator()`
- `ListIterator<T> listIterator(int index)`

Som-hi:

Iterator<T> iterator()

Aquesta operació *redefineix* `Iterable<T>.iterator()` en el sentit que ens assegura que l'iterador que ens retorna recorrerà la llista *en l'ordre en què hi estan disposats els elements*. Recordem que `Iterable<T>.iterator()` no feia cap referència a l'ordre en què es retornarien els elements. Novament, vegem un exemple de com una *subinterfície* particularitza l'especificació d'una operació declarada a la seva *superinterfície*.

Vegem gràficament el comportament d'aquest iterador:

- *Obtenció d'un iterador sobre una llista:*

$$\text{lis} = \{ \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad \}$$

```
Iterator<SomeType> it = lis.iterator();
```

$$\text{lis} = \{ \quad \underset{\text{it}}{\uparrow} \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad \}$$

L'iterador `it` s'ha situat a l'índex anterior al primer element de la llista.

Un cop tenim un iterador `it` a la llista, sobre `it` podem aplicar-hi les operacions de la interfície `Iterator<T>` que van ser presentades a 2.3.2: `hasNext()`, `next()` i `remove`. Fem-ho:

- `it.hasNext()`

$$\text{lis} = \{ \quad \underset{\text{it}}{\uparrow} \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad \}$$

```
boolean hasnext = it.hasNext();

hasnext == true
```

$$\text{lis} = \{ \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad \underset{\text{it}}{\uparrow} \quad \}$$

```
boolean hasnext = it.hasNext();

hasnext == false
```

$$\text{lis} = \{ \quad \underset{\text{it}}{\uparrow} \quad \}$$

```
boolean hasnext = it.hasNext();

hasnext == false
```

- `next()`

$$\text{lis} = \{ \quad \underset{\text{it}}{\uparrow} \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad \}$$

```
SomeType nextElem = it.next();
```

$$\text{lis} = \{ \quad e_1 \quad \underset{\text{it}}{\uparrow} \quad e_2 \quad e_3 \quad e_4 \quad \}$$

```
nextElem == e1
```

$$\text{lis} = \{ \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad \underset{\text{it}}{\uparrow} \quad \}$$

```
SomeType nextElem = it.next();
```

Es llença l'excepció `NoSuchElementException`

- `it.remove()`

  Elimina de la llista el darrer element que ha estat retornat amb una crida a `it.next()`. `it.remove()` es pot cridar només una vegada per a cada crida a `it.next()`. En particular, `it.remove()` no es pot cridar després d'una altra crida a `remove()`. El comportament dels iteradors no està definit si en un procés d'iteració de la llista, aquesta es modifica usant `remove()` i altres operacions de modificació (e.g., `add(...)`).

  Vegem gràficament el comportament típic d'`it.remove()`:

lis = { ↑it e_1 e_2 e_3 e_4 }

instr1: `SomeType nextElem = it.next();`

lis = { e_1 ↑it e_2 e_3 e_4 }

`nextElem == e1`

instr2: `it.remove();`

lis = { ↑it e_2 e_3 e_4 }

---

lis = { ↑it e_1 e_2 e_3 e_4 }

```
SomeType nextElem = it.next(); //instr1

it.remove(); //instr2

it.remove(); //instr3
```

La *instr3* llença una excepció `IllegalStateException`.

Sobre un iterador obtingut fent

```
1 Iterator<SomeType> it = lis.iterator();
```

només podrem realitzar les operacions `hasNext()`, `next()` i `remove()`. Aquestes operacions ens permeten utilitzar els iteradors per tal de *consultar* (`next()`) i *eliminar* (`remove()`) elements de qualsevol posició de la llista. No trobeu a faltar cap operació? Hi ha alguna altra cosa que ens agradaria fer amb els elements a una llista i que no la puguem fer amb les operacions d'`Iterator<T>`?

Efectivament, falta la possibilitat d'inserir elements a qualsevol posició de la llista *usant els iteradors*. Per fer-ho necessitarem usar un tipus d'iteradors més potents: són els `ListIterator<T>` que descriurem a la secció següent (per cert, aquests nous iteradors també ens permetran iterar els elements de la llista cap enrere: des del final cap al començament).

Potser alguns haureu pensat: per què necessitem inserir elements a la llista a la posició marcada per un iterador si tenim l'operació:

```
1 List.add(int position, E element)
```

que ens insereix `element` a la posició `position` de la llista? I, per descomptat, `position` pot ser qualsevol posició de la llista. És una bona observació. Així i tot, és convenient disposar d'una manera de fer les insercions a una posició arbitrària de la llista amb l'ajut d'iteradors. Per què?

ListIterator<T> listIterator()

Aquesta operació és similar a l'anterior (vegeu `void Iterator<T> iterator()`) però el que ens retorna és un tipus d'iterador més potent: el `ListIterator<T>`.

`ListIterator<T>` és una subinterfície d'`Iterator<T>`. Per tant, hereta totes les seves operacions i, a més a més, en defineix de noves. Introduïm aquestes operacions noves:

void add(E o) Inserts the specified element into the list (optional operation).
boolean hasPrevious() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
int nextIndex() Returns the index of the element that would be returned by a subsequent call to next.
E previous() Returns the previous element in the list.
int previousIndex() Returns the index of the element that would be returned by a subsequent call to previous.
void set(E o) Replaces the last element returned by next or previous with the specified element (optional operation).

I, de totes aquestes operacions, vegem amb detall el funcionament de `add(obj)` i d'`set(obj)`:

- `it.add(obj)`

  Aquesta operació insereix l'element `obj` immediatament abans de la posició ocupada per l'iterador `it`, tal com ho expressa l'especificació de l'operació a l'API del Java:

  > *Insereix l'element* `obj` *immediatament abans del següent element de la llista que retornaria l'operació* `next()` *(si existeix aquest element) i després de l'element que seria retornat per* `previous()` *si aquest elemetnt existeix.*

  `add(obj)` també llença excepcions en determinats casos. Doneu un cop d'ull a l'especificació per als detalls: http://download.oracle.com/javase/6/docs/api/java/util/ListIterator.html.

$$\text{lis} = \{\ e_1\ \ e_2\ \underset{\text{it}}{\uparrow}\ e_3\ \ e_4\ \}$$

`it.add(x);`

$$\text{lis} = \{\ e_1\ \ e_2\ \ x\ \underset{\text{it}}{\uparrow}\ e_3\ \ e_4\ \}$$

$$\text{lis} = \{\ \underset{\text{it}}{\uparrow}\ \}$$

`it.add(x);`

$$\text{lis} = \{\ x\ \underset{\text{it}}{\uparrow}\ \}$$

- `it.set(obj)`

  Aquesta operació substitueix el darrer element retornat per `next()` o per `previous()` pel paràmetre `obj`.

`it.next();`

$$\text{lis} = \{\ e_1\ \ e_2\ \underset{\text{it}}{\uparrow}\ e_3\ \ e_4\ \}$$

`it.set(x);`

$$\text{lis} = \{\ e_1\ \ x\ \underset{\text{it}}{\uparrow}\ e_3\ \ e_4\ \}$$

La crida a `set(obj)` es pot fer només si no s'ha fet cap crida a `ListIterator.remove` o `ListIterator.add` després de la darrera crida a `next()` o `previous()`.

```
               it.next();
lis = {   e1    e2     e3    e4    }
                    ↑
                    it

               it.remove();
lis = {   e1    e3    e4    }
               ↑
               it

               it.set(x);
```

Es llença una excepció `IllegalStateException` perquè s'ha cridat un `it.remove()` (o `it.add(obj)`) després d'`it.next()` i abans d'`it.set(x)`.

ListIterator<T> listIterator(int index)

Retorna un iterador sobre els elements d'aquesta llista (en la seqüència correcta) que comença a la posició especificada. Aquesta posició, com ja hem dit, pot variar entre 0 i lis.size().

Un exemple d'ús d'iteradors sobre una llista

La classe `java.util.Collections` ofereix un bon ventall d'algorismes sobre col·leccions. Entre ells, un (anomenat `reverse`) que capgira els elements d'una llista.

`public static void reverse(List<?> list)`
Capgira l'ordre dels elements de la llista `list`.

En particular, converteix la llista `{1, 2, 3, 4}` en la llista `{4, 3, 2, 1}`.

La implementació que presentem està basada en l'*OpenJDK*[1].

```
1  public static void reverse (List<?> list) {
2      int size = list.size();
3      ListIterator fwd = list.listIterator();
4      ListIterator rev = list.listIterator(size);
5      int mid = list.size() / 2;
6
7      for (int i = 0; i<mid; i++) {
8          Object tmp = fwd.next();
9          fwd.set(rev.previous());
10         rev.set(temp);
11     }
12 }
```

[1] http://openjdk.java.net/

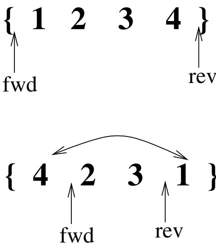


Figura 2.6: Esquema del funcionament de `reverse`

Comentaris:

- La figura 2.6 mostra l'esquema de funcionament de l'operació presentada. `fwd` i `rev` són dos iteradors definits sobre la llista que cal capgirar. `fwd` es desplaçarà cap endavant i `rev` cap enrere. A cada iteració substituirem els elements de les posicions `i` i `size()-1-i`. Ens caldrà capgirar la meitat dels elements (si `size()` és parell) o la meitat menys 1 (si `size()` és senar). Per això `i= 0..list.size()/2`.

- Com podríem substituir la instrucció de la línia 5 usant l'operador de desplaçament de manera que resultés més eficient?

  Podeu consultar-ho a la implementació de l'OpenJDK. Per exemple:

  `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Collections.java#Collections`

2.6 Implementant les llistes

Recordem, primer de tot, que quan parlem d'*implementacions de llistes* (o de qualsevol altre contenidor) ens referim a:

1. Com podem representar una llista (i.e., com podem emmagatzemar un objecte llista a la memòria).

2. Un cop triada una representació, com podem implementar les operacions del contenidor d'acord amb aquella representació i amb la seva especificació.

Hi ha dues maneres clàssiques de representar una llista a la memòria:

1. Posant els elements consecutius de la llista en posicions consecutives de memòria.

2. Posant els elements consecutius de la llista en posicions no necessàriament consecutives de memòria.

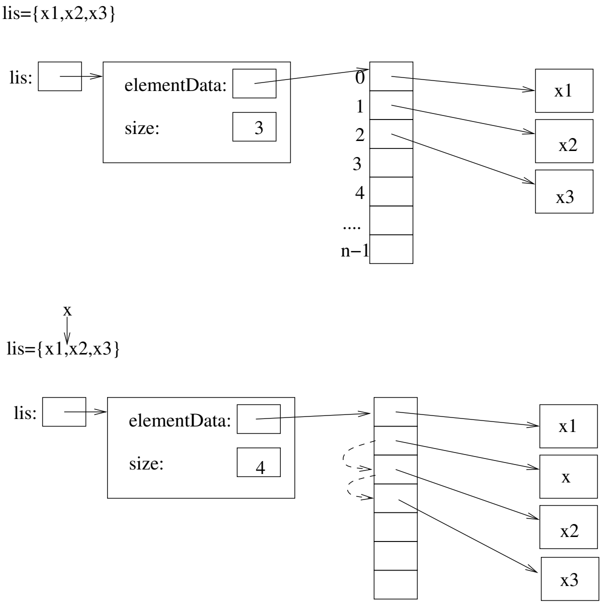


Figura 2.7: Representació de la classe `ArrayList<T>`

2.6.1 ArrayList: elements consecutius de la llista en posicions consecutives de memòria

Si volem representar una llista de manera que els seus elements consecutius es trobin situats en posicions consecutives de memòria, podem usar un vector de manera que els elements successius de la llista se situin en índexs correlatius del vector.

La classe `ArrayList<T>` representa una llista en forma d'vector redimensionable automàticament.

L'element i-èsim de la llista ocupa l'índex i del vector ($i \in 0..$`size()`-1).

```
1 public class ArrayList<T> implements List<T> ... {
2     private transient Object[] elementData;
3     private int size;
4     ...
5 }
```

La figura 2.7 mostra gràficament com es pot representar una llista en forma de vector i com es fan les insercions en aquesta llista.

Representada en forma de vector

El fet que la llista estigui representada en forma de vector:

- Fa que els elements de la llista estiguin situats en posicions consecutives de memòria: l'índex i d'un vector està situat en memòria immediatament abans que l'índex $i+1$. Això fa que es pugui accedir a l'índex i d'un vector molt eficientment.
- Permet un accés eficient per índex.

  Accedir a l'índex j d'una llista representada com a vector vol dir accedir a l'índex j del vector que la representa (`a[j]`) i aquesta és una operació que es pot fer amb temps constant $O(1)$.

  Penseu-hi un moment: quina és l'adreça de memòria on està situat l'índex 6 d'un vector de valors de tipus `int` que comença a l'adreça `base` sabent que cada `int` ocupa 4 bytes?

  Per tant, les operacions: `get(index)`, `set(index,element)` i `listIterator(index)` tenen un cost O(1).
- Permet insercions i eliminacions eficients al final de la llista.

  Les operacions `add(element` i `remove(size()-1)` tenen un cost O(1).
- Penalitza el cost de les insercions i eliminacions.

  Inserir un element a l'índex j implica moure una posició a la dreta tots els elements que ocupen els índexs $j...size()-1$. Una cosa similar passa amb les eliminacions.

  Per tant, les operacions: `add(index,element)`,`remove(index)`,`ListIterator<T>.add(element)` i `ListIterator<T>.remove()` tenen un cost O(n).

  Així i tot, aquestes operacions són relativament eficients atès que, en general, la còpia d'un vector s'implementa en maquinari.

Redimensionable automàticament

El fet que un `ArrayList<T>` sigui redimensionable de manera automàtica permet que, mentre hi hagi memòria disponible, es puguin afegir elements a la llista. Si un nou element no cap al vector que representa la llista, l'operació d'inserció (`add`) crea un vector més gran i copia el vell al nou abans de fer la inserció del nou element.

Notem, però, que aquest procés de redimensionat d'un vector (que implica una còpia de `size()` referències) afegeix un petit sobrecost a les operacions de la llista.

2.6.2 LinkedList: elements consecutius de la llista en posicions no consecutives de memòria

La classe `LinkedList<T>` representa una llista en forma de nodes enllaçats.
Cada node conté un enllaç al següent i també a l'anterior.
A aquesta representació se l'anomena *llista doblement enllaçada*.

```
1 public class LinkedList<T> implements List<T> {
2     private Entry<T> header;
3     private int size;
4     ...
5 }
```

Noteu que la classe `LinkedList<T>` es representa com una referència (`header`) al primer node. Noteu també que la classe que modelitza els nodes s'anomena `Entry<T>`.

La figura 2.11 (a la secció 2.7) mostra una idea de com es pot representar una llista doblement enllaçada. *Aquesta representació no és encara la definitiva*. En parlarem amb més detall a la secció 2.7.

La representació d'una llista amb nodes enllaçats porta associada:

- Un accés ineficient per índex.

  Efectivament, per accedir a l'índex i de la llista no podem fer altra cosa que recórrer els enllaços des del primer fins a l'i (o, si és més curt, des del darrer fins a l'i).

  Per tant, les operacions: `get(index)`, `set(index,element)` i `listIterator(index)` tenen un cost $O(n)$ (en què n és el nombre d'elements de la llista).

- Permet insercions i eliminacions a l'inici i al final de la llista eficients.

  Les operacions `add(element)`, `add(0,element)` i `remove(0)`, `remove(size()-1)` tenen un cost O(1).

  Penseu per què.

- Les insercions, consultes, eliminacions i modificacions en una posició arbitrària de la llista *sense usar iteradors* (o sigui, amb les operacions `add(index,element)`, `remove(index)`, `get(index)` i `set(index,element)`) tenen un cost $O(n)$ perquè, en primer lloc, cal situar-se sobre l'índex on cal fer la inserció, consulta, modificació o eliminació i això ja hem vist que té un cost $O(n)$.

- Les insercions, consultes, eliminacions i modificacions en una posició arbitrària de la llista *usant iteradors* (o sigui, amb les operacions `ListIterator<T>.add(element)`, `ListIterator<T>.next()`,`ListIterator<T>.set(element)` i `ListIterator<T>.remove()`) tenen un cost $O(1)$.

  Per explicar aquest cost, noteu que, a diferència del cas de l'`ArrayList<T>`, no cal fer cap desplaçament cap a la dreta o cap a l'esquerra en les insercions i eliminacions. Noteu també que, a diferència de les operacions sobre una posició arbitrària sense usar iteradors, ara *no cal desplaçar-se fins a una posició determinada ja que les operacions es fan sobre la posició assenyalada pel mateix iterador*.

2.6.3 Comparació entre LinkedList i ArrayList

Les taules següents miren de resumir els costos de les operacions principals d'inserció, consulta, modificació i eliminació en els dos tipus de llistes que estem presentant.

	get(index)	set(index,element)	listIterator(index)	add(index,element)
ArrayList<T>	O(1)	O(1)	O(1)	O(n)
LinkedList<T>	O(n)	O(n)	O(n)	O(n)

	remove(index)	ListIterator<T>.add(element)	ListIterator<T>.remove()
ArrayList<T>	O(n)	O(n)	O(n)
LinkedList<T>	O(n)	O(1)	O(1)

	add(element)	remove(size()-1)	add(0,element)	remove(0)
ArrayList<T>	O(1)	O(1)	O(n)	O(n)
LinkedList<T>	O(1)	O(1)	O(1)	O(1)

A més a més de la informació que mostren les taules anteriors, recordem:

- L'ArrayList<T> es redimensiona automàticament quan no hi ha més espai per afegir nous elements i això implica un petit sobrecost en temps respecte de la LinkedList<T> ja que la redimensió entranya una còpia del vector vell en un altre de nou, més gran.
- Cada node d'una LinkedList<T> ha d'emmagatzemar dues referències (next i previous) cap als nodes anterior i posterior a aquell node. Això comporta un petit sobrecost en espai respecte de l'ArrayList<T> ja que, en aquesta darrera, l'element anterior es troba físicament immediatament abans i el posterior, immediatament després. No calen, doncs, enllaços per localitzar-los.

Conclusions

Algunes idees per decidir-se entre ArrayList<T> i LinkedList<T> a l'hora d'usar una llista.

- Si a la llista cal accedir aleatòriament als elements per índex, millor ArrayList<T>.
- Si no s'han de realitzar sobre la llista gaires insercions/eliminacions d'elements en posicions aleatòries, millor ArrayList<T>.
- Si sobre la llista es fan fonamentalment recorreguts seqüencials i insercions/eliminacions d'elements en el transcurs d'aquells recorreguts, millor LinkedList<T>.
- Si sobre una llista es fan fonamentalment insercions/eliminacions a l'inici de la llista (índex 0), millor LinkedList<T>.
- Si sobre una llista es fan fonamentalment insercions/eliminacions al final de la llista (índex size()-1), millor ArrayList<T>.
- Si cap dels anteriors criteris és decisiu, millor ArrayList<T>.

Fins aquí hem mostrat generalitats de com es poden representar les llistes (ArrayList<T> i LinkedList<T>) i de quines són les característiques fonamentals (en particular, respecte al cost de les operacions) de cada representació. Ara ens centrarem en la representació en forma de LinkedList<T> i la discutirem amb tot detall.

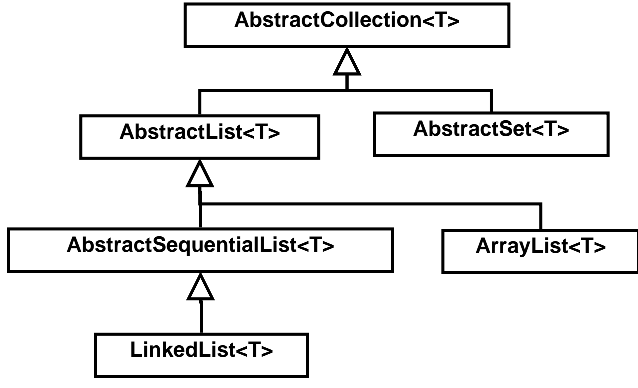


Figura 2.8: Jerarquia de classes arrelada a `AbstractCollection<T>`

2.7 Implementant les llistes amb enllaços: LinkedList<T>

La figura 2.8 mostra la jerarquia amb les classes ancestres (superclasses) de `LinkedList<T>`. En veure aquesta figura sorgeix una pregunta immediata: si volem proposar una implementació de llistes enllaçades, per què no implementem directament `LinkedList<T>`? Per què hem d'implementar també classes com ara `AbstractCollection<T>`, `AbstractList<T>` i `AbstractSequentialList<T>`?

La resposta a aquesta pregunta la podem entendre més clarament quan considerem la jerarquia completa a la mateixa figura 2.8. Si introduïm classes com ara `AbstractCollection<T>`, `AbstractList<T>` i `AbstractSequentialList<T>`, potser podrem implementar algunes operacions en aquestes classes i estalviar de repetir aquesta implementació a `LinkedList<T>`, `ArrayList<T>`, `Stack<T>`, `EnumSet<T>`, `HashSet<T>`, `TreeSet<T>` i moltes d'altres.

A més a més, és possible que un programador necessiti estendre la jerarquia de llistes amb un nova classe, `SimpleLinkedList<T>`, que podria implementar una llista amb enllaços només en una direcció (cada element estaria enllaçat només amb el següent, no com a `LinkedList<T>`, en què cada element està enllaçat amb l'anterior i el següent). En aquest cas, gràcies al fet que les classes `AbstractList<T>` i `AbstractSequentialList<T>` ja implementen bona part de les operacions que hauria d'oferir `SimpleLinkedList<T>`, la feina per a l'implementador d'aquesta classe fóra molt més senzilla.

Implementeu les classes i les seves operacions el més amunt que sigui possible a la jerarquia. Si ho feu així:

- Evitareu haver de repetir el mateix codi a diferents subclasses.
- Facilitareu la futura extensió d'una jerarquia amb noves classes.

En algunes ocasions, a una superclasse es dóna una implementació per a alguna operació que després a la subclasse es pot refinar (o sigui, es pot fer més eficient o més específica).

Aquest és el criteri que seguim en la implementació de LinkedList<T> (i també, d'ArrayList<T>, encara que a la implementació d'aquesta darrera classe no ens hi dedicarem).

Així doncs, les classes AbstractCollection<T>, AbstractList<T> i AbstractSequentialList<T> són classes abstractes que proporcionen un esquelet d'implementació de la interfície List<T> i es poden usar com a superclasse per implementar diferents classes concretes de llistes, com ara ArrayList<T> o LinkedList<T>. Gràcies a l'existència d'aquestes classes, un programador que vulgui implementar una llista concreta (e.g., LinkedList<T>) només ha d'implementar unes quantes operacions.

Hi ha, però, una diferència entre totes dues classes:

- AbstractList<T> vol ser un esquelet per a la implementació de classes llista basades en vectors (o sigui, classes llista, la representació de les quals permeti un accés als seus elements per posició en temps constant).

  Per això AbstractList<T> és la superclasse d'ArrayList<T>.
  De manera natural, també ho serà d'altres classes que implementin una llista amb una estructura (com un vector) que permeti un accés directe per posició.

- AbstractSequentialList<T>, per la seva banda, està dissenyada per a ser l'esquelet de les llistes enllaçades.

  Per tant, AbstractSequentialList<T> és la superclasse de LinkedList<T>.
  De manera natural, també ho serà d'altres classes futures que implementin una llista en forma enllaçada.

  AbstractSequentialList<T> es defineix com a subclasse d'AbstractList<T>.

En els propers apartats, doncs, donarem un cop d'ull a la idea de disseny de les classes `AbstractCollection<T>`, `AbstractList<T>`, `AbstractLinkedList<T>` i, finalment, `LinkedList<T>` i veurem com es duu a la pràctica aquest criteri.

La implementació que proposem d'aquestes classes està basada en l'*OpenJDK*. Aquest és el projecte de programari lliure (amb llicència GPL (v. 2)) que desenvolupa les versions 6 i 7 de *Java Standard Edition*.
Així i tot, hi ha algunes diferències degudes al fet que:

- Recorrem sempre les llistes des del primer element cap endavant (si es cerca un element que es troba proper al final de la llista seria més ràpid recórrer-les des del darrer cap al primer, com fa l'*OpenJDK*).
- No controlem que es facin sobre la llista modificacions amb operacions que no siguin ListIterator<T>.add i ListIterator<T>.remove mentre la llista s'està recorrent/modificant amb iteradors. Aquestes modificacions poden deixar els iteradors en una situació indefinida.
  Els iteradors que impedeixen aquestes modificacions s'anomenen *fail-fast*.

No considerem aquests dos aspectes per no complicar més la presentació de la implementació de `LinkedList<T>`.

Una manera senzilla de consultar el codi de la Java Standard Edition és a través de *grepcode*:
`http://grepcode.com/snapshot/repository.grepcode.com/java/root/jdk/openjdk/6-b14/`

Us recomano que ho feu.

2.7.1 AbstractCollection<T>(*)

Aquesta classe es defineix així:

```
1 public abstract class AbstractCollection <T> implements Collection <T> {
2       ...
3 }
```

Per tant, `AbstractCollection<T>` és una classe abstracta que implementa les operacions declarades a la interfície `Collection<T>`. Vegem com ho fa:

Operacions abstractes

Les dues operacions següents no les implementa `AbstractCollection<T>`.

```
1 public abstract Iterator <T>  iterator ();
2 public abstract int size ();
```

Aquestes operacions s'implementaran en classes de nivells inferiors de la jerarquia.

Operacions implementades en termes d'altres operacions de la mateixa classe

Sembla que, com que no sabem com s'acabaran representant les diferents col·leccions hereves de `AbstractCollection<T>`, no podem implementar cap de les operacions d'aquesta classe i hauríem de definir-les totes com a abstractes. Això no és cert. Hi ha operacions que es poden escriure en termes d'altres operacions ofertes a la mateixa classe i que s'implementaran

en classes de nivells inferiors de la jerarquia. Aquesta és una tècnica molt potent de disseny de biblioteques de programari orientades a objectes. Vegem-ne alguns exemples:

- boolean isEmpty()

  Quina ha de ser la implementació d'isEmpty() a AbstractCollection<T> per tal que pugui servir per a qualsevol subclasse seva?

  Considereu la següent implementació d'AbstractCollection<T>.isEmpty():

```
1 public boolean isEmpty() {
2     return size() == 0;
3 }
```

Aquesta implementació sembla raonable atès que un contenidor estarà buit si i només si el nombre d'elements que conté (i.e., size()) és zero. Però hem dit que l'operació size() és abstracta a AbstractCollection<T>. Així doncs, quina operació size() estem cridant?

> Usant el polimorfisme, estem cridant l'operació size() de la classe de l'objecte sobre el qual cridem a isEmpty().

Vegem un exemple:

Exemple 2.12:

```
1  <T> void foo(Collection<T> col) {
2      boolean b = col.isEmpty();
3      ...
4  }
5
6  lis = new LinkedList<Integer>();
7
8  lis2 = new ArrayList<Integer>();
9
10 foo(lis);
11 foo(lis2);
```

La seqüència de crides que generarà la instrucció de la línia 10 es mostra a la figura 2.9. Noteu que finalment es cridarà LinkedList.size(). I, és clar, a LinkedList sí que s'haurà definit l'operació size().

Una cosa similar ocorre per a la línia 11. Aquesta línia acabarà generant una crida a ArrayList.size() ja que, en aquest cas, lis2 és de classe ArrayList<Integer>.

■ ■ ■

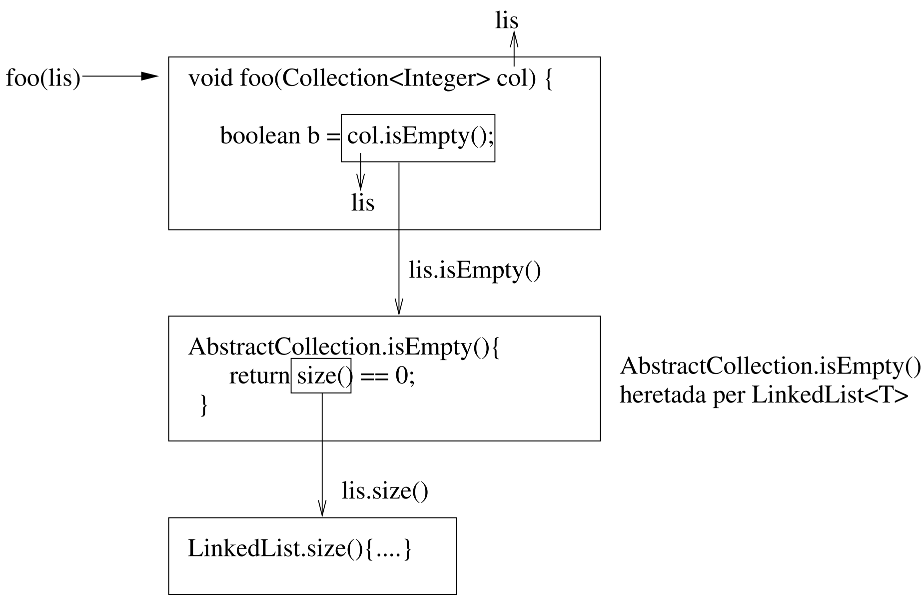


Figura 2.9: Seqüència de crides generada per `foo(lis)`

> El disseny d'una operació d'una superclasse (e.g., `AbstractCollection<T>` en termes de crides a operacions (e.g., `size()`) que s'implementen (possiblement de manera diferent) a les seves subclasses (e.g, `LinkedList<T>, ArraList<T>...`) constitueix un patró de disseny de programari anomenat *template method*.

- `void clear(); boolean remove(Object obj); bool contains(); bool containsAll(); Object[] toArray(); String toString()`

  Una consideració especial mereixen les operacions que es poden implementar en termes d'iteradors. Les implementacions d'aquestes operacions usen la mateixa tècnica que `isEmpty()`: *cridar operacions que s'implementaran en les seves subclasses.* La diferència és que aquestes últimes s'implementen en termes d'iteradors que recorreran una col·lecció. Posem l'exemple de `toString()`:

```
1  public String  toString() {
2      Iterator<T> i = iterator();
3      if (!i.hasNext())
4          return "[]";
5      StringBuilder sb = new StringBuilder();
6      sb.append('[');
7      for (;;) {
8          T e = i.next();
9          sb.append(e == this ? "(this Collection)" : e);
10         if (!i.hasNext())
11             return sb.append(']').toString();
```

```
12            sb.append(", ");
13        }
14  }
```

Comentaris:

– La línia 2 obté un iterador sobre aquest contenidor usant l'operació `Collection<T>.iterator()`. Altres línies (com ara 3, 8) operen sobre aquest iterador: `i.hasNext()`, `i.next()`.... Aquestes operacions de manipulació dels iteradors són les mateixes *per a tots els tipus de col·lecció*. Per tant, aquesta implementació de `toString()` ens servirà per a totes les col·leccions.

  Cada classe que proporcioni una implementació específica de col·lecció (e.g., `LinkedList<T>`) implementarà operacions com ara `iterator()`, `hasNext()`, `next()`... d'acord amb la representació específica d'aquesta classe i dels seus iteradors.

  Així doncs, tenim unes operacions cridades des d'`AbstractCollection<T>.toString()` i implementades a `LinkedList<T>` i a `ArrayList<T>`. És la mateixa idea (tot i que en una operació, `toString()`, un xic més complicada) que la que ens hem trobat a `isEmpty()`.

– Potser us haurà estranyat l'ús de la classe (potser desconeguda) `StringBuilder`. Penseu-la com una cadena de caràcters a la que es poden afegir elements al final mitjançant l'operació `append(...)`.
  En lloc de fer:

```
1    StringBuilder sb = new StringBuilder();
2    sb.append('[');
```

  podríem fer:

```
1    String st = new String();
2    st = st + "[";
```

  Però la implementació amb `StringBuilder` té un cost menor ja que permet afegir caràcters al final d'una cadena en lloc d'haver de crear una cadena nova cada cop que s'hi volen afegir caràcters (recordem que a un objecte de la classe `String` no s'hi poden afegir més caràcters perquè és immutable).

– I, per cert: què carai significa la instrucció de la línia 9 i per què s'ha afegit al codi?

```
1        sb.append(e == this ? "(this  Collection)" : e);
```

– Noteu, finalment, que a la mateixa línia 9, el compilador del Java transformarà `e` en `e.toString()`.

Operacions no suportades:

Algunes operacions de la interfície `Collection<T>` (com també passa amb la interfície `List<T>`) són opcionals: `add(...)`, `remove(...)`, `clear(...)`, `retainAll(...)`, `addAll(...)`, `removeAll(...)`. Ho són perquè les classes que la implementen poden tenir restriccions. Per

exemple, podríem definir una classe `InmutableLinkedList<T>` que no admetés modificacions després de la seva creació. Sobre un objecte `ill` instància d'aquesta classe no s'hauria de poder cridar cap de les operacions opcionals esmentades més amunt (e.g., `ill.add(obj)` no hauria de ser possible).

Aquestes operacions s'implementen llençant una excepció de tipus `UnsupportedOperationException`:

```
1 public void add(E element) {
2     throw new UnsupportedOperationException();
3 }
```

A vegades, aquesta excepció es llença, diguem, d'una manera indirecta. Considerem, per exemple, les operacions `retainAll`, `addAll` i `removeAll`. Aquestes operacions també són opcionals, però no llencen directament una excepció sinó que estan implementades en termes de `retain`, `add` i `remove`, respectivament.

Considerem, per exemple, l'operació `addAll(c)`, l'objectiu de la qual és afegir tots els elements de `c` a aquesta col·lecció:

```
1 public boolean addAll(Collection<? extends E> c) {
2     boolean modified = false;
3     Iterator<? extends E> e = c.iterator();
4
5     while (e.hasNext()) {
6         if (add(e.next()))
7             modified = true;
8     }
9     return modified;
10 }
```

Noteu que `addAll(c)` propagarà l'excepció `UnsupportedOperationException()` llençada per l'operació `add` de totes aquelles llistes que no suporten aquesta operació.

Per cert, per què penseu que `add(obj)` s'implementa directament retornant `UnsupportedOperationException` i, en canvi, `remove(obj)` (que també és una operació opcional i, per tant, pot estar no suportada en alguns tipus de llistes) està perfectament implementada?

```
1 public boolean remove(Object o) {
2     Iterator<T> e = iterator();
3     if (o == null) {
4         while (e.hasNext()) {
5             if (e.next()==null) {
6                 e.remove();
7                 return true;
8             }
9         }
10     } else {
11         while (e.hasNext()) {
12             if (o.equals(e.next())) {
13                 e.remove();
14                 return true;
```

```
15                }
16            }
17        }
18        return false;
19    }
```

2.7.2 AbstractList<T>(*)

Aquesta classe es defineix així:

```
1 public abstract class AbstractList<T> extends AbstractCollection<T>
2                                       implements List<T> {
3     ...
4 }
```

Per tant, és una classe abstracta que té `AbstractCollection<T>` com a superclasse i implementa la interfície `List<T>`. Per tant, a les operacions definides a `Collection<T>`, `AbstractCollection<T>` hi afegeix les que ofereix `List<T>` que, essencialment, tenen a veure amb la posició dels elements al contenidor.

`AbstractList<T>` no proporciona cap representació per a les llistes. Això ho faran les seves subclasses. En canvi, sí que implementa operacions de `List<T>`.

Vegem de quina manera ho fa, segons el tipus d'operacions:

Operacions heretades d'`AbstractCollection<T>`

`AbstractList<T>` essencialment hereta sense modificar la majoria de les operacions implementades a `AbstractCollection<T>`.

Alguna d'aquestes operacions, però és redefinida per `AbstractList<T>`. Considerem l'exemple d'`add(obj)`:

```
1 public boolean add(E e) {
2     add(size(), e);
3     return true;
4 }
```

Aquesta operació, a `AbstractCollection<T>`, llençava directament `UnsupportedOperationException`. Per què us sembla que s'ha fet aquesta redefinició?

Operacions declarades abstractes a `AbstractCollection<T>` i definides a `AbstractList<T>`

D'altra banda, l'operació `Iterator<T> iterator()` que era abstracta a `AbstractCollection<T>` ara és implementada de la manera següent:

```
1 public Iterator<T> iterator() {
2     return new Itr();
3 }
```

On `Itr` és una classe interna a `AbstractList<T>` i privada que representa iteradors sobre una `AbstractList<T>`. Ens podem preguntar com és possible implementar uns iteradors si no sabem com està representada internament la llista (aquesta representació no es farà fins a `ArrayList<T>` o `LinkedList<T>`). Però, en realitat, sí que podem fer-ho: n'hi ha prou de representar els iteradors com un índex (`cursor`) que indicarà l'índex de l'element que retornarà la següent crida a `next()`. A partir de l'índex, podem obtenir l'element cridant l'operació `get(index)`. Mostrem molt breument els elements més destacats:

```
1  private class Itr implements Iterator<T> {
2      int cursor = 0;
3      ...
4
5      public boolean hasNext() {
6          return cursor != size();
7      }
8
9      public T next() {
10         try {
11             int i = cursor;
12             T next = get(i);
13             cursor = i + 1;
14             return next;
15         } catch (IndexOutOfBoundsException e) {
16             throw new NoSuchElementException();
17         }
18     }
19     ...
20 }
```

Comentaris:

- L'operació `get(index)`, cridada a la línia 12, és abstracta a `AbstractList<T>`. Quan s'executi aquest codi, es cridarà, veritablement, una operació abstracta?

Operacions pròpies de la intefície `List<T>`

Ens referim a les operacions que defineix `List<T>` específicament i que, per tant, no són heretades de `Collection<T>`.

Aquestes operacions solen fer referència a les posicions dels elements de la llista (recordem que aquesta és precisament l'aportació de `List<T>` a `Collection<T>`).

- Algunes de les operacions específiques de `List<T>` poden ser implementades en termes d'iteradors. Per exemple:

```
1  public boolean addAll(int index, Collection<? extends E> c) {
2      rangeCheckForAdd(index);
3      // Mira si l'index esta dins del rang d'indexos
4      // d'aquesta llista
5
```

```
 6      boolean modified = false;
 7      Iterator<? extends E> e = c.iterator();
 8
 9      while (e.hasNext()) {
10          add(index++, e.next());
11          modified = true;
12      }
13      return modified;
14  }
```

Comentaris:

- Recordem que l'objectiu d'`addAll` afegirà tots els elements de `c` a partir de l'índex `index` d'aquesta col·lecció.
- Noteu que `addAll` propagarà l'excepció `UnsupportedOperationException()` llençada per l'operació `add` de totes aquelles llistes que no suporten aquesta operació.

Altres operacions d'aquesta mena són: `indexOf(T obj)`, `lastIndexOf(T obj)`, `sublist(fromIndex, toIndex)`.

Vegem, per exemple, la implementació d'`indexOf(obj)`. Aquesta operació retorna l'índex de la llista que conté la primera ocurrència de l'objecte `obj`. Si `obj` no pertany a la llista, retorna -1. `indexOf(obj)` es pot implementar com segueix:

```
 1  public int indexOf(Object o) {
 2      ListIterator<T> e = listIterator();
 3      if (o == null) {
 4          while (e.hasNext())
 5              if (e.next()==null)
 6                  return e.previousIndex();
 7      } else {
 8          while (e.hasNext())
 9              if (o.equals(e.next()))
10                  return e.previousIndex();
11      }
12      return -1;
13  }
```

- Les operacions opcionals que no poden ser implementades eficientment amb iteradors, llencen una excepció:

```
 1  public void add(int index, T element) {
 2      throw new UnsupportedOperationException();
 3  }
 4
 5  public T set(int index, T element) {
 6      throw new UnsupportedOperationException();
 7  }
 8
 9  public T remove(int index) {
10      throw new UnsupportedOperationException();
```

```
11  }
```

Com implementaríem amb iteradors qualsevol d'aquestes operacions? Seria una implementació eficient?

- Finalment trobem T get(int index), que no pot ser implementades eficientment amb iteradors i que tampoc no és opcional: aquesta està definida com a abstracta.

```
1   abstract public E get(int index);
```

2.7.3 AbstractSequentialList<T>(*)

Hem dit a l'inici de la secció 2.7 que AbstracList<T> és més apropiada com a superclasse per a les implementacions de llistes basades en vectors (i, per tant, dotades d'accés directe per posició), com ara ArrayList<T> i AbstractSequentialList<T>, per implementacions de llistes amb accés seqüencial als seus elements, com ara LinkedList<T>.

Les diferències entre AbstractSequentialList<T> i AbstractList<T> les podem sintetitzar en els aspectes següents:

- Algunes operacions a AbstractList<T> no estan implementades (o estan implementades trivialment llençant una excepció) perquè a nivell d'AbstractList<T> només podríem implementar-les en termes d'iteradors i sabem que ArrayList<T>, la subclasse natural d'AbstractList<T>, podrà implementar-les més eficientment perquè la representació dels elements de la llista en forma de vector proporciona un accés directe per posició que evita haver d'accedir a l'element que ocupa una posició determinada a través d'un iterador.

  En canvi, LinkedList<T> no tindrà aquesta opció d'accés directe als seus elements per posició i haurà d'usar iteradors per accedir a la posició index de la llista. Per tant, la mateixa superclasse de LinkedList<T> (AbstractSequentialList<T>) ja pot oferir una implementació d'aquestes operacions basada en iteradors.

  Aquestes operacions són:

  T get(int index),T set(int index, T element),void add(int index, T element), T remove(int index).

  Vegem un exemple d'això amb l'operació get:

  Exemple 2.13:

  AbstractList<T> defineix T get(int index) com a operació abstracta. En canvi, AbstractSequentialList<T> la defineix així:

```
1 public T get(int index) {
2     try {
3         return listIterator(index).next();
4     } catch (NoSuchElementException exc) {
5         throw new IndexOutOfBoundsException("Index: " + index);
6     }
7 }
```

Com veiem, la implementació d'`AbstractSequentialList<T>.get(i)` està basada a obtenir un iterador a l'índex `index` de la llista (`listIterator(index)`). Això està bé en el cas de `LinkedList<T>` ja que la representació d'aquesta classe no ens permet cap accés més directe a la posició `i`. Per tant, aquesta implementació de `get(i)` pot ser adient per a `LinkedList<T>`, la subclasse coneguda d'`AbstractSequentialList<T>`.

En canvi, en el cas d'`ArrayList<T>`, la representació de la llista en termes de vectors ens permetrà un accés directe a la posició `i` (simplement, consultant l'índex `i` del vector que representa la llista). Si aquest vector és `elementData`:

```
1 public T get(int index) {
2     rangeCheck(index);
3     // Per assegurar que l'index es dins del
4     // rang d'indexs definits pel vector
5
6     return (T) elementData[index];
7 }
```

Però des d'`AbstractList<T>` no tenim accés al vector `elementData`. De fet, aquest vector ni tan sols existeix quan s'implementa `AbstractList<T>`. És per això que definim `T get(int index)` com a abstracta a `AbstractList<T>`.

■ ■ ■

- `AbstractList<T>` proporciona a les seves subclasses (entre elles, `AbstractSequentialList<T>`) una implementació bàsica de les operacions `Iterator<T> iterator()` i `ListIterator<T> listIterator(int index)`.

  Aquestes operacions seran després redefinides per d'altres de més eficients tant a `LinkedList<T>` com a `ArrayList<T>`.

2.7.4 LinkedList<T>

Subclasse d'`AbstractSequentialList<T>` que implementa les llistes com a seqüència d'elements en memòria no seqüencial. O sigui, el tercer element de la llista no es troba emmagatzemat a la memòria immediatament després del segon. Aleshores, un cop arribem al segon element, com podem saber on es troba el tercer? Amb un enllaç que anirà des del segon al tercer. En general, cada element disposarà d'un enllaç al següent element de la llista.... i també d'un altre enllaç a l'anterior. A les següents seccions discutim amb detall com es fa la representació i la implementació de la classe `LinkedList<T>`.

Representació

Fem una primera aproximació a la representació de `LinkedList<T>`:

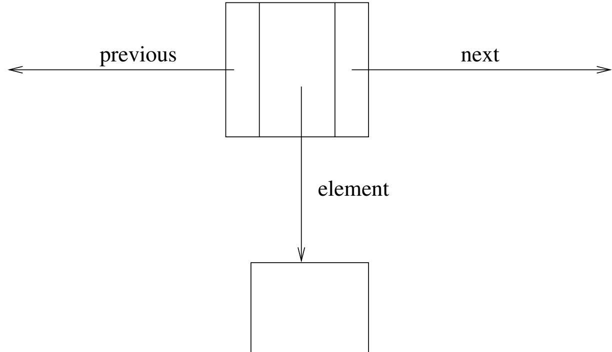


Figura 2.10: Proposta de representació d'un node (`Entry<T>`) de la `LinkedList<T>`

> La classe `LinkedList<T>` representa les llistes com una seqüència de nodes (que anomenem `Entrys`) de manera que, cada node conté:
> - Una referència a un element de la llista (`element`).
> - Una referència al següent node de la llista (`next`).
> - Una referència a l'anterior node de la llista (`previous`).
>
> En concret, una llista es representa com una referència (que anomenem `header`) al seu primer node.

Les figures 2.10 i 2.11 donen una idea gràfica d'aquesta proposta de representació que encara no serà la final.

Aquesta idea inicial de representació no és plenament satisfactòria:

- Els algorismes que manipulen la llista (`add`,`remove` ...) es poden trobar amb nodes que tenen dues referències inicialitzades (al node següent i a l'anterior). Aquest és el cas dels elements que ocupen les posicions diferents de la primera i la darrera de la llista). I també es poden trobar amb elements als quals manca una de les dues referències (és el cas del primer i del darrer elements).

  Encara més, els algorismes que manipulen la llista es poden trobar que el `header` es refereixi a `null` (si la llista és buida).

  Tractar tots aquests casos requeriria que els algorismes que manipulen la llista haguessin de considerar molts casos particulars, la qual cosa els fa més complicats i menys elegants.

  Per evitar aquest problema, intentarem proposar una representació de manera que:

  - Fins i tot la llista buida disposi d'un node (que eviti que `header==null` en aquest cas).
  - Cadascun dels nodes de la llista tinguin inicialitzades les seves referències al node següent i a l'anterior.

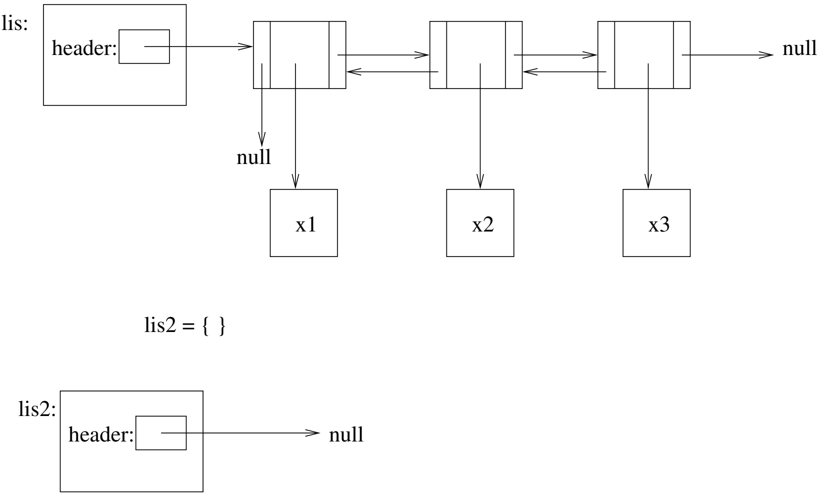


Figura 2.11: Proposta inicial de representació d'una `LinkedList<T>`

- Fóra bo, per tal d'implementar l'operació `size()`, de guardar en un atribut el nombre d'elements que hi ha a la llista.

Considerant aquestes dues millores, obtenim la representació que mostra la figura 2.12.

La representació d'una `LinkedList<T>` que mostra la figura 2.12 la podem caracteritzar de la manera següent:

- El primer node en la representació d'una llista qualsevol és un *node virtual* que no conté cap element de la llista.

  Aquest node és un artifici per tal que la implementació de les operacions sigui més elegant i no contingui una col·lecció inacabable d'`ifs` per tal de distingir els diferents casos particulars.

- El següent node del darrer dels que formen la llista és el *node virtual*. De la mateixa manera, l'anterior del *node virtual* és el darrer.
- Una llista buida només consta del *node virtual* amb les referències a anterior i següent apuntant a ell mateix.

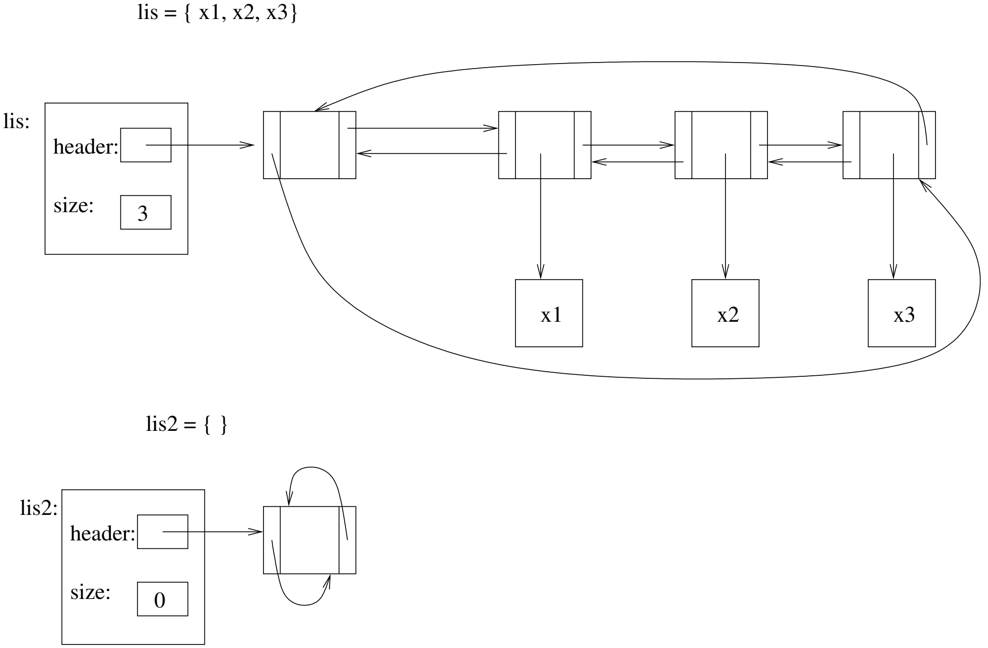


Figura 2.12: Proposta final de representació d'una `LinkedList<T>`

> **Invariant de la representació**
> A les propietats de la representació d'una classe `C` que són certes per qualsevol instància d'aquella classe se les anomena **invariant de la representació**.
>
> Les operacions constructores d'una classe s'han d'assegurar que, al final, l'objecte construït compleixi l'invariant de la representació.
>
> La resta d'operacions de la classe s'han d'assegurar que si s'apliquen sobre un objecte que compleix l'invariant de la representació, quan acabi cadascuna l'objecte seguirà complint el mateix invariant.

Notem, finalment, que, amb aquesta representació, aconseguim els nostres propòsits: `header` sempre apunta a un node (al virtual, concretament), mai a `null`, i les referències a anterior i següent de qualsevol node estan inicialitzades en tot moment.

Aquesta representació es pot codificar en el Java de la manera següent:

```
1  public class LinkedList<T> extends  AbstractSequentialList<T>
2                              implements List<T>, Cloneable {
3
4      private Entry<T> header = new Entry<T>(null, null, null);
5      private int size = 0;
6
7      private static class Entry<T> {
8          T element;
9          Entry<T> next;
```

```
10          Entry<T> previous;
11
12          Entry(T element, Entry<T> next, Entry<T> previous) {
13              this.element = element;
14              this.next = next;
15              this.previous = previous;
16          }
17      }
18
19      //.........
20  }
```

Comentaris:

- Ja hem dit que aprofitaríem la implementació que ofereix `AbstractSequentialList<T>`, per tal d'estalviar-nos feina a l'hora d'implementar `LinkedList<T>`. D'altra banda i com era d'esperar, `LinkedList<T>` implementa la interfície `List<T>`.

  Fem també que implementi la interfície `Cloneable`, la qual cosa vol dir que haurà d'oferir una operació per *clonar* una llista (o sigui, per fer-ne una còpia exacta).

- Les classes estàtiques aniuades (com ara `Entry<T>`) són classes que es defineixen com a membres estàtics d'una altra classe (en aquest cas, `LinkedList<T>`). Això es pot fer per diverses raons. Les més habituals són perquè:

  – La classe estàtica aniuada només és utilitzada per la classe allà on està definida com a membre.

  – La classe estàtica aniuada no necessita accedir als membres no estàtics de la classe allà on està definida.

  Això és el que passa en el cas d'`Entry<T>`: la definim com a estàtica aniuada perquè és una classe que només serà usada per `LinkedList<T>` i no accedirà als membres no estàtics de `LinkedList<T>` (si doneu un cop d'ull a la implementació d'aquesta classe veureu que és efectivament així).

  Així doncs, no oferirem `Entry<T>` a cap altra classe i, per tant, no la incloem en el catàleg de classes de la nostra biblioteca.

> Atenció: a l'apèndix B proporcionem una discussió més detallada sobre les classes aniuades.

Implementació

Un cop discutida la representació de la classe `LinkedList<T>` passem a implementar les seves operacions. Aquesta implementació haurà de:

- Basar-se en la representació que acabem de mostrar
- Complir l'especificació de l'API `LinkedList<T>` del Java.

Estrictament parlant, per implementar `LinkedList<T>`, definida com a subclasse d'`AbstractSequentialList<T>`, només caldrà implementar les operacions:

- Constructora,
- size(),
- listIterator(int index).

I, per als `listIterators` que haurà de proporcionar la llista:

- hasNext(),
- next(),
- hasPrevious(),
- previous(),
- remove(T e),
- add(T e),
- previousIndex(),
- nextIndex().

La resta d'operacions ja estan implementades a alguna de les seves superclasses (`AbstractSequentialList<T>`, `AbstractList<T>` i `AbstractCollection<T>`). Així i tot, la implementació en què ens basem (*OpenJDK*) n'implementa algunes altres per tal de donar-ne una implementació més eficient. Nosaltres ens centrarem en les més representatives de les obligatòries i, després, farem algun comentari sobre la redefinició d'algunes operacions per assolir major eficiència. Som-hi.

Implementació. Constructores

`LinkedList()`
Operació constructora per defecte. Construeix una llista buida.

```
1  public LinkedList() {
2      header.next = header.previous = header;
3  }
```

Comentaris:

- Construeix l'estructura que es mostra a la figura 2.12 corresponent a una llista buida.

`LinkedList(Collection<? extends T> c)`
Construeix una llista que conté els elements de la col·lecció `c` en l'ordre en què els retorna l'iterador de la col·lecció.

```
1 public LinkedList(Collection<? extends E> c) {
2     this();
3     addAll(c);
4 }
```

Comentaris:

- Primer construeix una llista buida cridant a `this()` (l'operació constructora per defecte que hem presentat més amunt).
- Després hi afegeix tots els elements de la col·lecció `c`. Això ho fem delegant a `AbstractList<T>.addAll(c)`.

> Si tenim una operació que ofereix la funcionalitat que necessitem en un moment determinat, usem-la abans que recodificar aquella funcionalitat.

Implementació. `size`

`int size()`
Retorna el nombre d'elements a la llista.

La implementació d'aquesta operació és trivial gràcies a l'atribut `size` que hem incorporat:

```
1 public int size() {
2     return size;
3 }
```

Implementació. `listIterator`

`ListIterator<T> listIterator(int index)`
Retorna un iterador als elements d'aquesta llista (en l'ordre en què apareixen a la llista). Aquest iterador (usant l'operació `next()`) retornarà els elements de la llista començant pel que es troba a la posició `index`.
Paràmetres: index, índex del proper element de la llista que ha de ser retornat mitjançant una crida al mètode `next()`. **Retorna:** un iterador sobre els elements d'aquesta llista (en la seqüència en què aquests elements apareixen a la llista) **Llença:** IndexOutOfBoundsException, si `index` és fora de rang (index < 0 || index > size()).

```
1 public ListIterator<T> listIterator(int index) {
2     return new ListItr(index);
3 }
```

Comentaris:

- L'operació `listIterator(index)` ha de retornar un iterador que sigui instància d'una classe que implementi la interfície `ListIterator<T>`. Però de quina classe pot ser

aquest iterador? Fins ara no hem definit cap classe que implementi ListIterator<T>. Per tant, sembla raonable fer-ho ara. ListItr serà una classe que implementarà ListIterator<T> tot proporcionant iteradors sobre LinkedList<T>. A la propera secció expliquem les intimitats d'aquesta classe.

La classe ListItr

Com acabem de veure, l'operació LinkedList<T>.listIterator(index) retorna un objecte iterador de la classe ListItr. Aquesta classe haurà d'implementar la interfície ListIterator<T> i haurà de permetre recórrer llistes de classe LinkedList<T>.

Si hi pensem una mica veurem que ListItr té dues característiques:

- Ha de conèixer la representació de LinkedList<T>. Efectivament, un iterador s'haurà de moure pels nodes de la LinkedList<T> i haurà de retornar els elements de la llista continguts a cada node (Entry<T>); per tant serà bo que tingui coneixement d'aquests nodes i dels seus atributs next, previous i element de Entry<T>. També serà bo que conegui l'atribut header d'LinkedList<T> per tal de poder iniciar el recorregut de la llista. Però tots aquests atributs *són privats d'*Entry<T> *i de* LinkedList<T>*!*. Així doncs, com hi podrà accedir la classe ListItr?

- Els usuaris de LinkedList<T> que necessiten un iterador sobre una llista, i fan la crida:

```
1   ListIterator<T> it = lis.listIterator(0);
```

  no estan interessats a saber quina classe concreta els retornarà l'operació listIterator(index). Només necessiten que se'ls retorni un objecte iterador d'alguna classe que implementi ListIterator<T>. No els importa quina sigui aquesta classe. Per tant, **no caldrà que coneguin l'existència d'una classe** ListItr.

Què us sembla que ens suggereixen aquestes dues reflexions? Penseu-hi un moment.

La manera que una classe I (e.g., ListItr) pugui veure els atributs privats d'una altra E (e.g., LinkedList<T>) i, al mateix temps, que la classe I no sigui coneguda pels usuaris d'E és definir I com a *classe aniuada no estàtica i privada a* E.

Les classes aniuades no estàtiques s'anomenen simplement *classes internes* (*inner classes*).

Així doncs, definirem ListItr com a *classe interna* privada de LinkedList<T>.

```
1 public class LinkedList<T> extends  AbstractSequentialList<T>
2                              implements List<T>, Cloneable {
3     ...
4     private class ListItr implements ListIterator<T> {
5         ...
6     }
7     ...
8 }
```

I ara ve la qüestió més important: com podem representar la classe `ListItr`? Ens n'ocupem a la propera secció.

La classe `ListItr`. Representació

Fem un recordatori dels iteradors sobre les llistes (vegeu la secció 2.5.2):

Els iteradors sobre llistes estan situats en posicions *entre elements* de la llista. Un iterador `it` ocupa la posició corresponent a:

- L'índex del proper element de la llista que retornarà l'operació `it.next()` (en aquest cas l'iterador es troba entre dos elements de la llista o abans del primer).
- El nombre d'elements de la llista (en aquest cas, l'iterador es troba després del darrer element de la llista o la llista és buida).

En total, un iterador sobre una llista d'n elements té $n + 1$ posicions possibles.

Aquestes posicions corresponen als índexs $0 \ldots n$

lis =	{		e_1		e_2		e_3		e_4		}
		↑		↑		↑		↑		↑	
posició =		0		1		2		3		4	

La conceptualització que acabem de fer de `ListIterator<T>` ens suggereix immediatament una manera natural de representar objectes de la classe `ListItr` que ha d'implementar `ListIterator<T>`. La figura 2.13 proposa aquesta representació clara.

La representació mostrada a la figura 2.13 es caracteritza per:

- L'atribut `next` *apunta* al següent element de la llista a ser retornat mitjançant l'operació `next()` o `header`, si l'iterador està situat a l'índex `size()` (i.e., l'iterador està situat en una posició més enllà del darrer element de la llista).
- L'atribut `nextIndex` conté l'índex ocupat per l'iterador (0...`size()`). Si `hasNext()` retorna true, aquest índex coincideix amb el que ocupa el següent element a ser retornat per l'operació `next()`.
- L'atribut `lastReturned` assenyala el darrer element retornat per una operació `next()` o `previous()`. Si no s'ha fet cap operació `next()` o `previous()` o bé s'ha cridat una operació de modificació (`add(e)`, `remove()`) després de `next()` o `previous()`, aleshores `lastReturned` assenyala `header`.

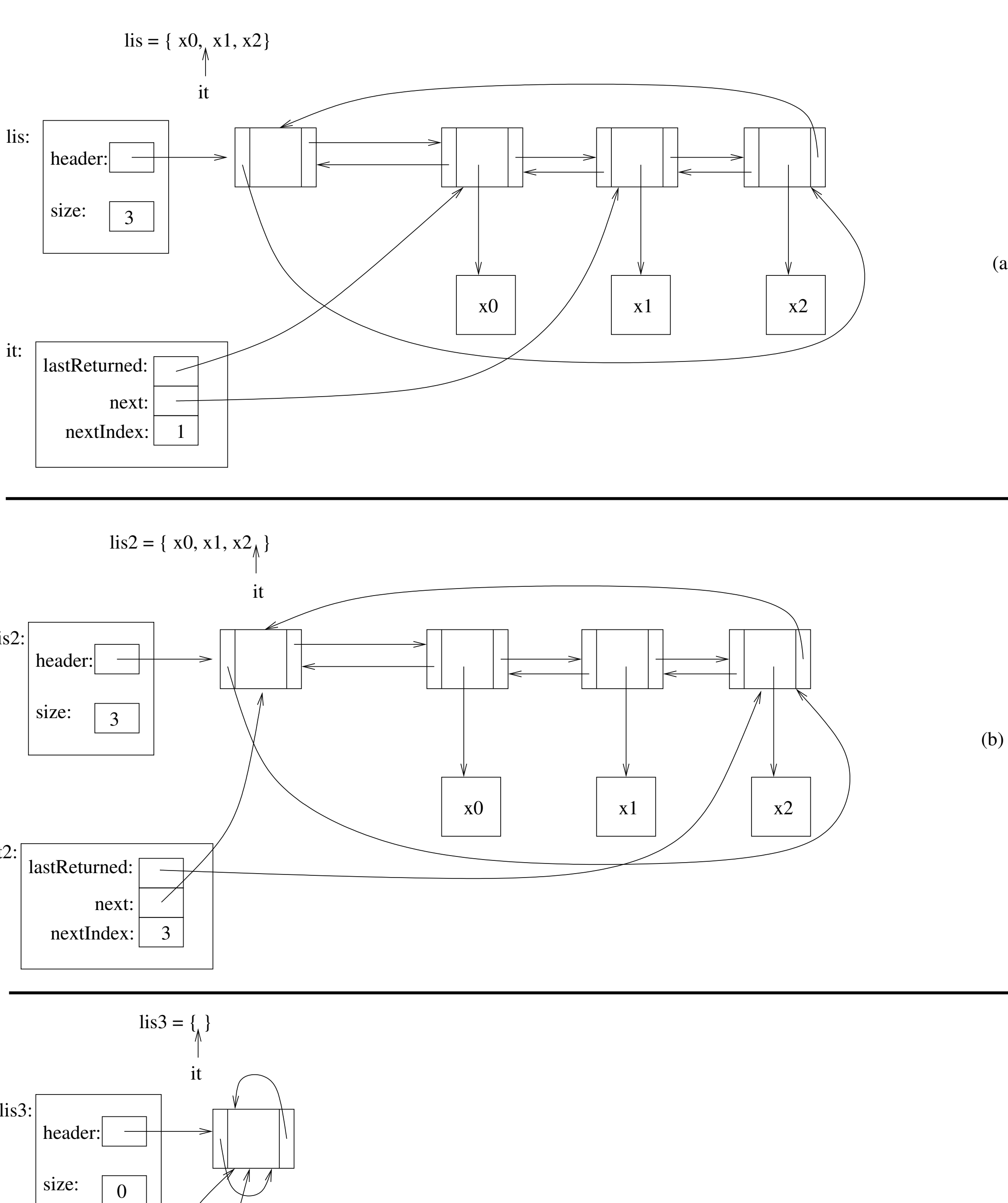


Figura 2.13: Proposta de representació de la classe `ListItr`

Notem que, **de cap manera** `lastReturned` ha d'assenyalar sempre l'anterior de `next`. Als exemples que apareixen a la figura 2.13 ocorre d'aquesta manera però no és necessari que sempre sigui així. Penseu un moment en situacions en què `lastReturned` no apuntarà l'anterior de `next`.

En general, `lastReturned` assenyalarà:

- A `header` si no s'ha fet encara cap operació sobre l'iterador o bé si la darrera operació feta ha estat `add(x)` o `remove()`.
- A `next.previous` si s'està fent un recorregut *cap endavant* de la llista.
- A `next` si s'està fent un recorregut *cap enrere* de la llista.

Seguidament, presentem la codificació en el Java de la representació de la figura 2.13 i també l'operació constructora:

```
1  private class ListItr implements ListIterator<T> {
2      private Entry<T> lastReturned = header;
3      private Entry<T> next;
4      private int nextIndex;
5
6      ListItr(int index) {
7          if (index < 0 || index > size)
8              throw new IndexOutOfBoundsException
9                      ("Index: " + index + ", Size: " + size);
10
11         next = header.next;
12         for (nextIndex = 0; nextIndex<index; nextIndex++)
13             next = next.next;
14     }
15     ...
16 }
```

Comentaris:

- Notem la inicialització de `lastReturned` a `header` quan encara no s'ha cridat a cap operació `next()` o `previous()`.

- Com cal interpretar la instrucció de la línia 13: `next = next.next;`?

`ListItr.hasNext`

```
1 public boolean hasNext() {
2     return nextIndex != size;
3 }
```

`ListItr.next`

```
1 public T next() {
2     if (nextIndex == size)
3         throw new NoSuchElementException();
4     lastReturned = next;
5     next = next.next;
6     nextIndex++;
7     return lastReturned.element;
8 }
```

`ListItr.add`

`void add(T o)` Insereix `o` a la llista sobre la qual es mou aquest iterador. `o` s'insereix immediatament abans de la posició que ocupa l'iterador.
Paràmetres: `o`, l'element que cal inserir. **Llença:** `UnsupportedOperationException`, si el mètode `add` no és compatible amb aquesta llista iterador. `ClassCastException`, si la classe de l'element especificat evita que s'afegeixi a aquesta llista. `IllegalArgumentException`, si algun aspecte de l'element evita que s'afegeixi a aquesta llista.

Llistat 2.4: Operació `add(e)`

```
1 public void add(T e) {
2     lastReturned = header;
3     addBefore(e, next);
4     nextIndex++;
5 }
```

Comentaris:

- La implementació d'aquesta operació mostra l'ús del disseny descendent o, si voleu, de la separació d'objectius (*separation of concerns*): `add(e)` ha de fer tres coses:

  - Inicialitzar `lastReturned` a `LinkedList<T>.header` (indicant així que s'ha cridat alguna altra operació (la mateixa `add`) després de la darrera crida a `next()` o `previous()`). Això és necessari per poder implementar adequadament `set(o)` i `remove()`. Per què?
  - Afegir l'element `e` immediatament abans de `next`.
  - Com que hem afegit un nou element abans de `next`, ara `next` ocupa l'índex immediatament següent; per tant, cal actualitzar `nextIndex`.

  D'aquestes tres coses, la segona:

  - És complexa (cal crear un nou node (`Entry<T>`), i actualitzar les referències `next` i `previous` d'alguns nodes).

- Cal fer-la en altres operacions (a més a més de `ListItr.add(e)`). Per tant, caldria repetir el mateix codi d'afegir un element abans d'un node en diferents llocs.

  En quines altres operacions us sembla que pot ser útil una crida a `addBefore(e,node)`?

Per tant, és raonable delegar el procés d'afegir un element immediatament abans d'un node determinat en una operació `addBefore(e,node)` que s'especialitzarà en aquest procés i que serà reutilitzada des d'altres operacions.

> Quan detectem que en la implementació d'una operació cal fer un procés d'una certa complexitat i/o que és necessari també en la implementació d'altres operacions *implementarem aquell procés en una nova operació privada*.

`LinkedList.addBefore`

El llistat 2.5 i la figura 2.14 mostren el funcionament de l'operació privada `LinkedList<T>.addBefore(e,entry)`.

A la figura, els números entre parèntesis indiquen en quina instrucció es produeix aquell canvi.

Llistat 2.5: Operació `addBefore(e,entry)`

```
1 private Entry<T> addBefore(E e, Entry<T> entry) {
2     Entry<T> newEntry = new Entry<T>(e, entry, entry.previous);
3     newEntry.previous.next = newEntry;
4     newEntry.next.previous = newEntry;
5     size++;
6     return newEntry;
7 }
```

Comentaris:

- Noteu que la representació de la llista amb el *node virtual* inicial i els dobles punters (`next` i `previous`) fa que el procés `addBefore` permeti inserir un nou node abans de qualsevol altre node de la llista, sense haver de considerar en cap moment un cas particular (llista buida, inserció al final de la llista...).

  En particular, quina crida faríeu a `addBefore(e,entry)` si vulguéssiu afegir un element al final de la llista?

  Encara més, assegureu-vos que `addBefore(e,header)`, en una llista buida, funciona correctament.

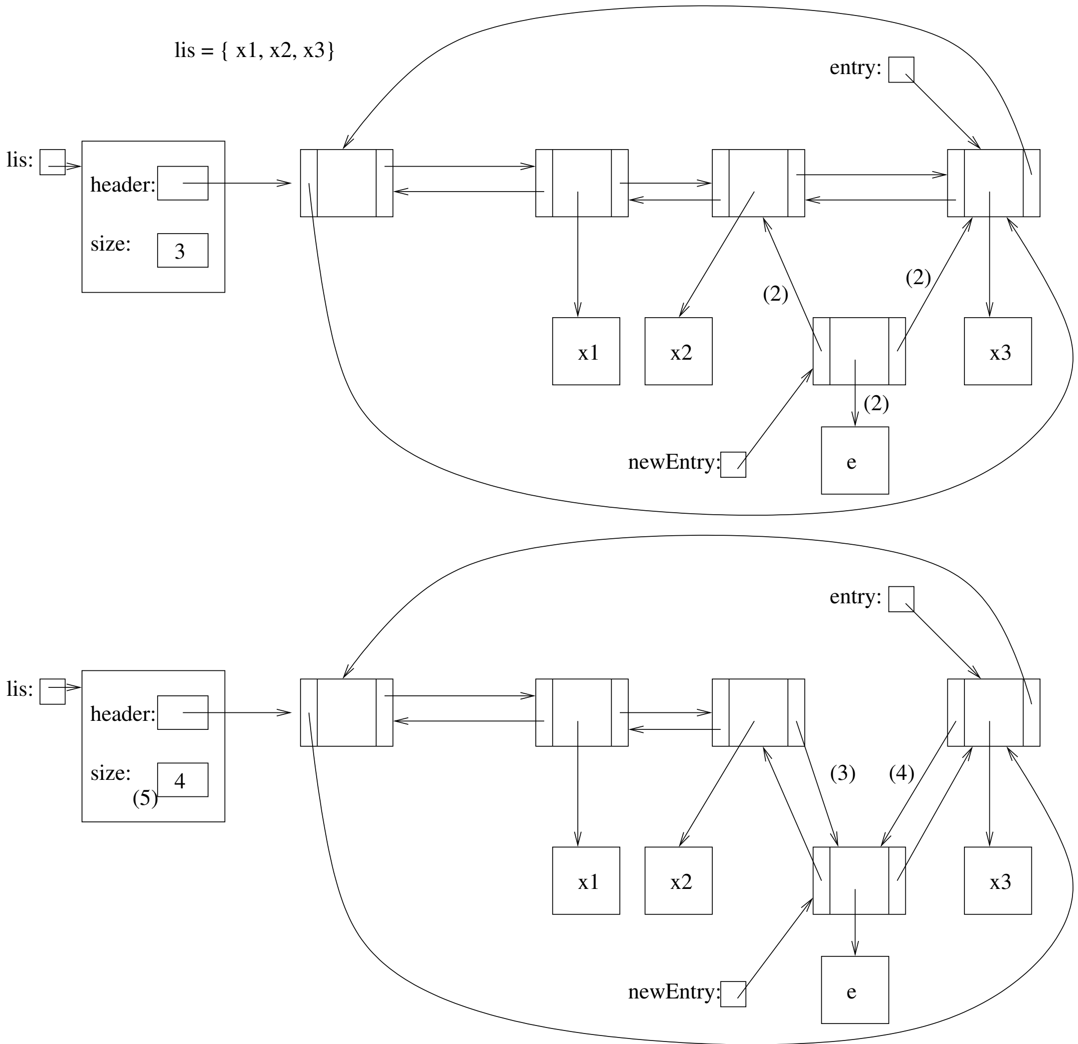


Figura 2.14: Funcionament d'`addBefore`

`ListItr.remove`

`void remove()` Elimina de la llista el darrer element que va ser retornat per una crida a `next()` o `previous()`. Aquesta crida només es pot fer un cop per a cada crida a `next()` o a `previous()`. En particular, es pot fer només si `ListIterator.add(e)` no s'ha cridat després de la darrera crida a `next()` o `previous()`.
Llença: `UnsupportedOperationException`, si l'operació `remove` no és suportada per aquest iterador. `IllegalStateException`, si no s'ha cridat `next()` ni `previous()`, o bé s'han cridat `remove()` o `add(e)` després de la darrera crida a `next()` o `previous()`.

```
1  public void remove() {
2      Entry<T> lastNext = lastReturned.next;
3      try {
4          LinkedList.this.remove(lastReturned);
5      } catch (NoSuchElementException e) {
6          throw new IllegalStateException();
7      }
8      if (next == lastReturned)
9          next = lastNext;
10     else
11         nextIndex--;
12     lastReturned = header;
13 }
```

Comentaris:

- Noteu que, com en el cas de l'operació `add(e)`, l'eliminació de l'element és delegada a l'operació privada `LinkedList<T>.remove(entry)` que es presenta a l'apartat següent.
- Considerem la línia 4:

  ```
  1        LinkedList.this.remove(lastReturned);
  ```

  `LinkedList.this` és una construcció que no hem vist mai i ens molesta una mica. Suposem que hi omitim `LinkedList`. Ens queda:

  ```
  1     this.remove(lastReturned);
  ```

  Aquesta instrucció està continguda al codi de `ListItr.remove()`. Per tant, `this` indica una instància de `ListItr`. Però la línia 4 vol cridar `LinkedList<T>.remove(Entry<T>);` per tant `this` hauria de ser instància de `LinkedList<T>`. En aquestes condicions, `LinkedList.this` es refereix a la instància de `LinkedList<T>` associada a la instància `this` de `ListItr`.

  O sigui, `LinkedList.this` es refereix a la instància de `LinkedList<T>` *que va ser responsable de la creació de la instància* `this` *de* `ListItr` .

La construcció `className.this` s'usa freqüentment per accedir a la instància de la classe externa des d'una operació d'una classe interna no estàtica (*inner class*).

Per què us sembla que no hem hagut de recórrer a aquesta construcció a la línia 3 del llistat 2.4?

- L'operació privada `LinkedList<T>.remove(entry)` elimina el node `entry`. Hi ha un node que no pot eliminar: el *node virtual*, el que està situat a l'inici de la llista (recordem que una llista, ni que sigui buida, *sempre té, com a mínim el node virtual a l'inici*). Si `entry` correspon a aquest node virtual llença l'excepció `NoSuchElementException`.

Com ja hem vist, l'operació `ListItr.remove()` elimina el darrer element retornat per una crida a `next()` o a `previous()` i si, posteriorment a la darrera crida a `next` o a `previous` s'ha modificat la llista mitjançant `ListItr.add()` o la mateixa `ListItr.remove()`, la crida `ListItr.remove()` llença `IllegalStateException`. Una forma natural d'aconseguir aquest comportament és fer:

```
1 lastReturned = header;
```

a cada crida a `ListItr.add(e)` o a `ListItr.remove()`. Fent-ho així, una crida posterior a `ListItr.remove()` executarà , segons hem vist:

```
1 LinkedList.this.remove(lastReturned);
```

I, com que `lastReturned == header`, `LinkedList<T>.remove(entry)` llençarà l'excepció `NoSuchElementException`, la qual serà capturada per

```
1 try {
2     LinkedList.this.remove(lastReturned);
3 } catch (NoSuchElementException e) {
4     throw new IllegalStateException();
5 }
```

que a la seva vegada, llençarà `IllegalStateException`, i s'obtindrà així el resultat desitjat.

Ara podeu veure més clar per què resulta natural fer `lastReturned = header` a les operacions `add` i `remove`.

El tractament associat a la captura d'una excepció que consisteix a llençar-ne una altra:

```
1 try {
2     LinkedList.this.remove(lastReturned);
3 } catch (NoSuchElementException e) {
4     throw new IllegalStateException();
5 }
```

s'usa moltes vegades en disseny orientat a objectes. Entre altres coses permet adequar les excepcions que llença una operació amb les que estan indicades a la seva especificació.

- La instrucció condicional (`if`) de la línia 8 discrimina els dos casos següents:
  - `it.previous(); it.remove()`: eliminem un element en un recorregut de la llista cap enrere. Aquesta situació es descriu a la figura 2.15.

    Si recorrem cap enrere la llista, necessàriament `lastReturned==next` (vegeu la figura 2.15(b)). En aquest cas, l'element que eliminem és precisament `next` (que és el mateix que `lastReturned`). Per tant, cal reassignar `next` adequadament: `next=lastReturned.next;` (vegeu la figura 2.15(c)).

- `it.next(); it.remove()`: eliminem un element en un recorregut de la llista cap endavant.

  És el segon cas. Ara només cal decrementar `nextIndex` ja que l'eliminació es fa en un índex anterior a `nextIndex`.

Notem a la figura 2.15 que, com a conseqüència de la crida a `it.remove()`, `lastReturned` torna a `header`. Si ara es vol fer una nova crida a `it.remove()`, caldrà fer-ne primer una a `it.next()` o `it.previous()`.

`LinkedList.remove`

La figura 2.16 i el llistat 2.6 mostren el funcionament de l'operació privada `LinkedList<T>.remove(entry)`.

A la figura, els números entre parèntesis indiquen en quina instrucció es produeix aquell canvi.

Llistat 2.6: Operació `remove(entry)`

```
1  private E remove(Entry<T> e) {
2      if (e == header)
3          throw new NoSuchElementException();
4
5      E result = e.element;
6      e.previous.next = e.next;
7      e.next.previous = e.previous;
8      e.next = e.previous = null;
9      e.element = null;
10     size--;
11     return result;
12 }
```

Comentaris:

- Les instruccions de les línies 8 i 9 són fonamentals. Per què?

Implementació. Redefinició d'operacions implementades a les superclasses (*)

Ja hem dit que gràcies al fet que `LinkedList<T>` es defineix com a subclasse d'`AbstractSequentialList<T>` i aquesta última d'`AbstractList<T>`, hereta moltes operacions de les dues darreres classes que, per tant, no cal implementar novament. Així i tot, pot convenir implementar-ne algunes per raons d'eficiència. Efectivament, un cop hem decidit una representació per a `LinkedList<T>` (la qual cosa fem, evidentment, a `LinkedList<T>`) podem implementar més eficientment algunes operacions. Vegem-ne un exemple.

Exemple 2.14:

L'operació

```
1  public boolean add(T e)
```

it = lis.listIterator(3);

lis:
header:
size: 3

it:
lastReturned:
next:
nextIndex: 3

x0 x1 x2

lis = { x0, x1, x2 }
it

(a)

it.previous();
it.previous(); //Executem dues operacions previous();

lastNext

lis:
header:
size: 3

it:
lastReturned:
next:
nextIndex: 1

x0 x1 x2

lis = { x0, x1, x2 }
it

(b)

it.remove(); //Volem eliminar l'element retornat pel darrer previous(): x1

lastNext

lis:
header:
size: 3

it:
lastReturned:
next:
nextIndex: 1

x0 x1 x2

lis = { x0, x2 }
it

(c)

Figura 2.15: `it.remove()` en recorregut cap enrere

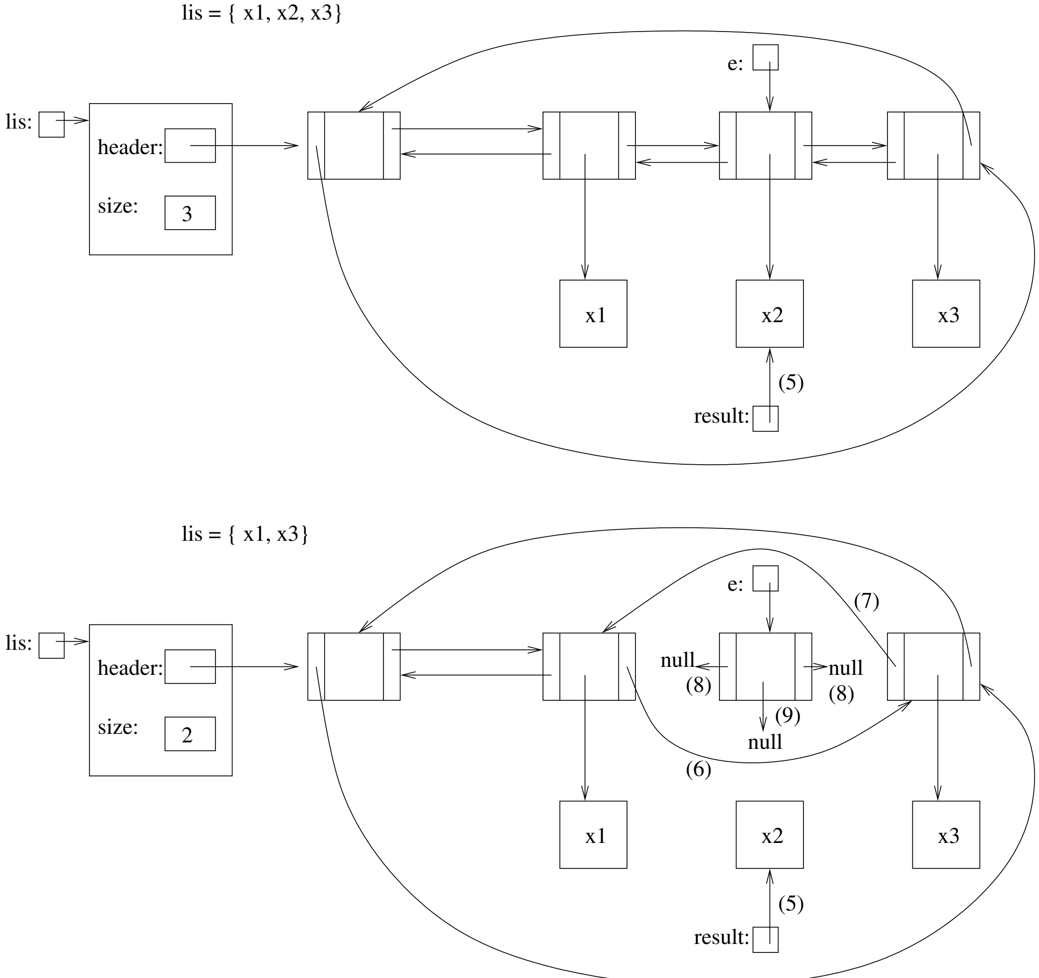


Figura 2.16: Funcionament de `remove`

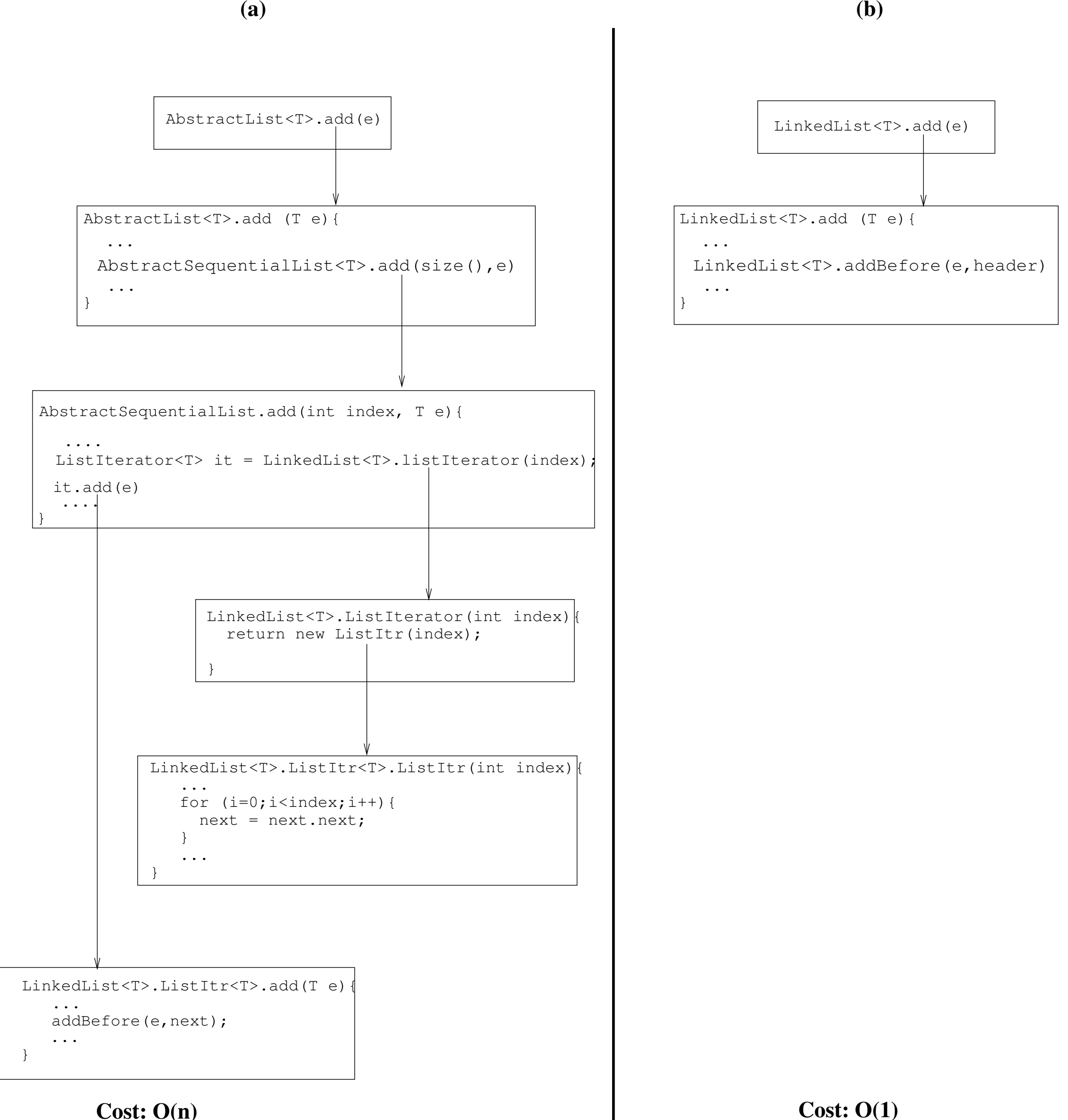


Figura 2.17: Implementacions d'`AbstractList<T>.add` i `LinkedList<T>.add`

està definida a la interfície `List<T>` i, per tant, s'ha d'implementar a `LinkedList<T>`. Però com que aquesta operació ja està implementada a `AbstractList<T>` que és superclasse (concretament, àvia) de `LinkedList<T>`, aquesta darrera classe l'hereta i, per tant, no caldria implementar-la. Així i tot, els dissenyadors del paquet *java.util* del Java decideixen implementar-la. Per què?

Perquè la classe `LinkedList<T>` és la que coneix els detalls de la representació de les llistes enllaçades i, en conseqüència, pot donar una implementació molt més directa i eficient d'`add(T e)`, sense necessitat d'usar iteradors. El següent llistat ens mostra la senzillesa d'aquesta implementació:

```
1 public boolean add(T e) {
2     addBefore(e, header);
3     return true;
4 }
```

En particular, noteu que el cost d'aquesta operació és $O(1)$.

En canvi, la implementació que la classe `AbstractList<T>` fa de la mateixa operació és molt més complexa i ineficient. La figura 2.17(a) mostra com `AbstractList<T>.add(e)` acaba generant també una crida a `addBefore(...)`, però després de fer molta més feina i crides a altres operacions. En particular, després de recórrer amb un iterador tota la llista fins a trobar el seu darrer node (anomenat `next` al codi). Aleshores ja podrà afegir l'element `e` immediatament abans del darrer node (`next`): `addBefore(e,next))`.

En conseqüència, el cost de l'operació `AbstractList<T>.add(e)` és $O(n)$.

> `LinkedList<T>` té una informació valuosíssima que `AbstractList<T>` no té: sap que el darrer node d'una llista serà sempre `header`. En canvi, `AbstractList<T>` ni tan sols sap que una llista de classe `LinkedList<T>` està formada per nodes (`Entrys`).
>
> De fet, ni tan sols sap que existeix una classe anomenada `LinkedList<T>`.

■ ■ ■

2.8 Racó lingüístic

En aquesta secció comentem els termes tècnics usats en aquest capítol que no estan estandarditzats al diccionari de l'Institut d'Estudis Catalans, a Termcat o a [CM94].

- *Contenidor (o contenidor de dades), Col·lecció.*

  Proposem aquests dos termes com a sinònims d'*estructura de dades* i els usem tots tres indistintament al llarg del volum.

  Ni *Contenidor* ni *Col·lecció* no estan estandarditzat ni a Termcat ni a [CM94]. Així i tot, pensem que aquests termes aporten una idea molt més intuïtiva i natural que l'estandarditzat *estructura de dades*. L'objectiu no és, evidentment, canviar un terme que ja ha estat acceptat i introduït al lèxic estàndard informàtic sinó oferir-ne unes alternatives més intuïtives.

- *Punter.*

  És el terme estandarditzat a Termcat com a traducció de l'anglès *pointer*. A [CM94] no apareix. L'esmentem en aquest racó per fer notar que el terme *apuntador*, que s'usa molt sovint en el mateix context no és el terme estàndard català.

- *Enllaç. Llista enllaçada.*

  Tradicionalment el terme anglès `linked list` es tradueix per *llista encadenada* o *llista enllaçada*. Així i tot, cap dels dos termes no es troba estandarditzat a Termcat ni tampoc a [CM94]. Ens sembla que la idea d'enllac (i.e., connexió) d'una dada amb la següent expressa amb major precisió el fonament de les llistes enllaçades. En anglès, el terme *chained list* que, durant temps es va fer servir, ha estat substituït progressivament per *linked list*.

  De la mateixa manera i pels mateixos motius triem *enllaç* com a traducció de *link*.

- *Cua doble*

  Traduïm l'estructura de dades que en anglès s'anomena *deque (double-ended queue)* per *cua doble*. Proposem aquesta denominació perquè la possibilitat de fer moviments als dos extrems de l'estructura ens permet considerar-la com si fossin dues cues: una té el primer element a l'esquerra i el darrer a la dreta i l'altra a l'inrevés.

  Ni Termcat ni [CM94] proposen una traducció estandarditzada per *deque*.

- *Iterable. Iterador*

  De manera natural, traduïm els termes *iterable* i *iterator* per *iterable* i *iterador*, respectivament. Cap dels dos termes no apareix a Termcat ni a [CM94].

Capítol 3

Arbres

Les llistes eren seqüències i, com a tals, tenien definida una noció natural de *següent*: cada element estava enllaçat a un altre, que li feia de *següent*. En aquest capítol, presentem els arbres, unes estructures de dades en què, cada element no té un únic següent natural sinó que està associat a dos o més elements, als que anomenarem *fills*. Els arbres són estructures de dades que definirem recursivament i que donen molt de joc per modelitzar problemes. En particular, amb ells es poden representar la col·lecció de crides recursives que fa un programa amb recursivitat múltiple, les cues amb prioritat i també les taules. A aquest darrer exemple, li dedicarem tot el capítol 6.

3.1 Els arbres a vista d'ocell

Un dels aspectes que (espero) quedaran més clars al llarg de la descripció del capítol és que els arbres són estructures de dades eminentment recursives. Així doncs, per què no en donem una definició, ella mateixa recursiva?

Un arbre amb elements components de tipus `T` és un contenidor d'elements de tipus `T` que pot prendre les formes següents:

1. Un **arbre buit** (sense cap element).

2. Un arbre no buit format per:

   - Un element de tipus `T` anomenat **arrel**.
   - Una llista d'arbres disjunts associats a l'arrel que s'anomenen **subarbres fills** (primer fill, segon fill...).

     En particular, aquesta llista d'arbres associats a l'arrel pot ser buida. En aquest cas, l'únic element de l'arbre seria l'arrel.

Hi ha moltes varietats d'arbre. El tipus de llista que tinguem determinarà la varietat específica d'arbre.

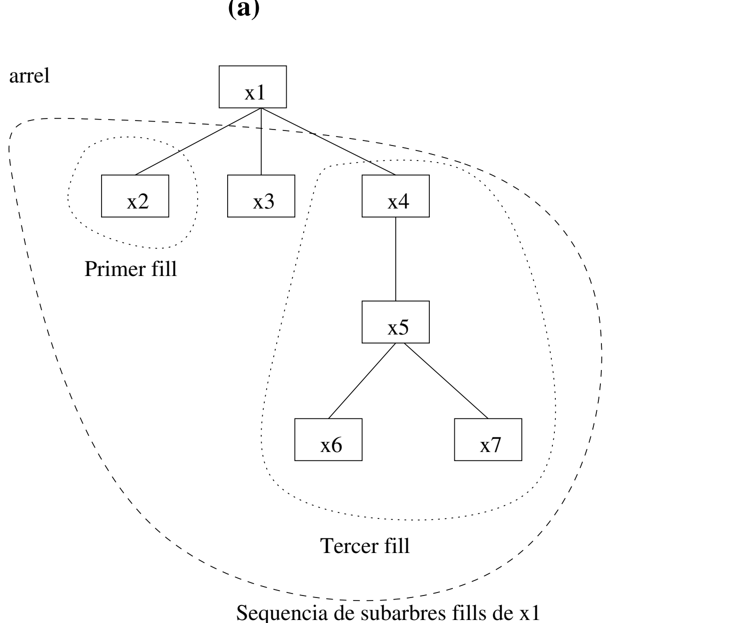


Figura 3.1: (a): Un arbre. (b): Un arbre buit

Exemple 3.1: Un primer exemple d'arbres

La figura 3.1 mostra diversos arbres i els seus elements:

(a) Arbre format per un element arrel (x1) i una llista de tres subarbres fills: els arrelats respectivament als elements x2, x3 i x4.

El primer subarbre fill està arrelat a x2 i té una llista de subarbres buida.

El tercer subarbre fill està arrelat a x4 i té una llista de subarbres formada per un únic element: el subarbre arrelat a x5.

(b) Arbre buit (efectivament, no hi ha res dibuixat).

■ ■ ■

Una propietat molt important dels arbres (que podeu comprovar a la figura) és la següent:

Existeix un únic camí entre l'arrel d'un arbre i qualsevol dels seus elements.

Hem dit que segons el tipus de llista podem distingir diferents varietats d'arbres. Vegem-ho:

3.1.1 Arbres generals

La llista de fills associada a l'arrel no té cap restricció. En particular:

- No està limitat el nombre de fills que pot tenir.
- No està fixada la posició d'un fill determinat en aquesta llista.

Exemple 3.2: Arbres generals

Si considerem l'arbre de la figura 3.1(a) com un arbre general, veiem que l'arbre arrelat a `x1` té associada una seqüència amb tres fills, mentre que l'arbre arrelat a `x2` té associada una seqüència amb zero fills (seqüència buida). Per altra banda, l'arbre arrelat a `x4` té associada una seqüència amb un fill.

Encara més, podem anomenar l'arbre arrelat a `x2` *primer fill*. A l'arbre arrelat a `x3`, *segon fill* i *tercer fill* a l'arrelat a `x4`. Si ara eliminem el primer fill d'un arbre general (l'arrelat a `x2`), el que fins ara era el segon fill (l'arrelat a `x3`) passa a ser el primer. I el que era el tercer (l'arrelat a `x4`) passa a ser el segon. Ara no hi ha tercer fill. O sigui, la posició que ocupa un fill dins de la seqüència de fills no està fixada.

■ ■ ■

3.1.2 Arbres n-aris

En els arbres n-aris, la llista de fills associada a l'arrel té algunes restriccions:

- Té exactament n elements.

  O sigui, l'arrel d'un arbre n-ari té exactament n subarbres fills. Noteu que alguns d'aquests fills poden ser l'arbre buit.

- Cada fill ocupa una posició determinada i invariable en aquesta llista.

  Parlem doncs de primer fill, segon fill,..., n-èsim fill. L'eliminació del fill i-èsim fa que aquest fill i-èsim sigui l'arbre n-ari buit. La posició dels altres fills no es veu alterada.

En aquest cas, ens podem imaginar com si la llista de subarbres associada a l'arrel fos, en realitat, *un vector*: ja sabem que un vector té un nombre fixat d'elements (que anomenem `.length`) i que cada element ocupa un índex determinat i invariable. Si l'element que ocupa l'índex 4 és esborrat, l'element que ocupava l'índex 5 del vector, ara seguirà ocupant el mateix índex 5. No passarà a ocupar el 4.

Exemple 3.3: Arbres 3-aris

Tornem a l'arbre de la figura 3.1(a). Considerem-lo ara com un arbre 3-ari. L'arbre arrelat a `x2` és el primer fill d'aquest arbre. L'arbre arrelat a `x3` és el segon fill i l'arrelat a `x4`, el tercer fill.

L'arbre arrelat a `x2` té tres fills: tots tres, arbres 3-aris buits. El mateix li passa a l'arbre arrelat a `x3`. Per la seva banda, l'arbre arrelat a `x4` té també tres fills: el primer fill, és l'arbre 3-ari buit. El segon fill és l'arbre 3-ari arrelat a `x5` i el tercer fill és, novament, l'arbre 3-ari buit.

Finalment, si eliminem l'arbre arrelat a `x2`, l'arbre arrelat a `x3` continua essent el segon fill d'`x1` i l'arrelat a `x4` segueix essent el tercer fill d'`x1`.

■ ■ ■

3.1.3 Arbres binaris

Un cas particular d'arbre n-ari el trobem per a $n = 2$. En aquest cas tenim els **arbres binaris**. Els arbres binaris tenen una importància enorme en el món de la programació i de les estructures de dades. Per això, tot i ser un cas particular d'arbre n-ari, mereixen que ens hi aturem una estona. De fet, a partir d'ara treballarem, principalment, amb arbres binaris. En particular, dedicarem les seccions 3.2 i 3.3 a especificar i a implementar aquest tipus d'arbres.

Comencem donant la definició d'arbre binari que s'obté de particularitzar el concepte d'arbre n-ari per a $n = 2$:

Un **arbre binari** amb elements components de tipus `T` és un contenidor d'elements de tipus `T` que pot prendre les formes següents:

1. Un **arbre binari buit** (sense cap element).
2. Un arbre binari no buit format per:
   - Un element de tipus `T` anomenat **arrel**.
   - Dos arbres binaris disjunts associats a l'arrel i anomenats **fill esquerre** i **fill dret**.

Exemple 3.4: Un arbre binari

La figura 3.2 mostra un arbre binari. Noteu com l'arbre arrelat a `x2` és el fill esquerre d'`x1`, mentre que l'arbre arrelat a `x3` és el fill dret.

Si considerem l'arbre arrelat a `x2`, veurem que té l'arbre binari buit com a fill esquerre i l'arbre arrelat a `x4` com a fill dret.

L'arbre arrelat a `x4` té l'arbre binari buit com a fill esquerre i dret.

■ ■ ■

A partir d'ara, com ja hem indicat, treballarem quasi sempre amb arbres binaris.

Abans d'acabar aquesta secció preliminar donem una mica de notació que usarem sovint amb els arbres i parlem dels recorreguts sobre arbres.

3.1.4 Notació d'arbres

En aquesta secció proposem una mica de notació que s'usa sovint en l'àmbit dels arbres. La proposarem amb exemples d'arbres binaris però aquesta notació és igualment vàlida per a arbres generals.

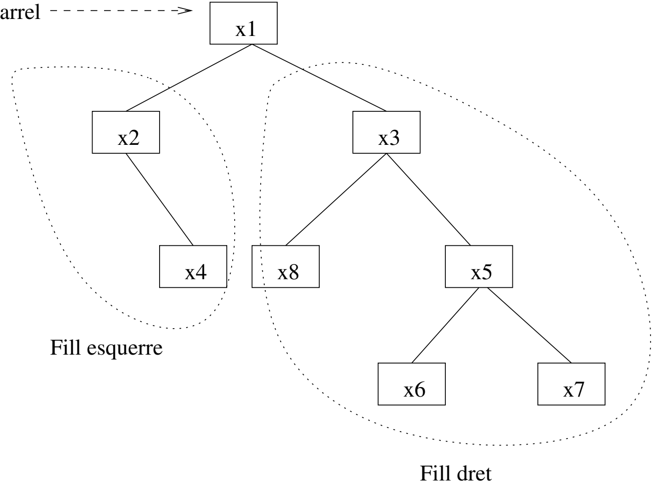


Figura 3.2: Arbre binari

Considerem novament la figura 3.2. En aquesta figura diem que:

- `x1`...`x8` són **elements** o **nodes** de l'arbre binari.
- `x1` és un node especial: **l'arrel** de l'arbre binari.
- `x1` és el **pare** d'`x2` i d'`x3`.

  Noteu que l'arrel és l'únic node d'un arbre que no té pare. Noteu també que cada node té un únic pare.
- `x2` i `x3` són fills d'`x1`.
- `x2` i `x3` són **germans**. També ho són `x5` i `x8`.
- A vegades es diu que `x1` és l'**avi** d'`x4`, `x5` i `x8`. També es pot dir que `x1` és **avantpassat** d'`x4` o que `x4` és **descendent** d'`x1`.
- En un arbre binari els nodes que tenen dos fills no buits s'anomenen **nodes interiors** o **nodes de grau 2**. Els que en tenen menys de dos s'anomenen **nodes terminals** o **nodes de grau 1 o 0**.

  `x1` és un node interior. `x2` i `x4` són nodes terminals.
- En particular, els nodes que no tenen cap fill no buit s'anomenen **fulles**. Per exemple, `x4`, `x6` o `x8` són fulles.
- Es diu que l'arrel ocupa el **nivell** 1 de l'arbre. Els nodes `x2` i `x3` ocupen el nivell 2. El nivell 3 és ocupat pels nodes `x4`, `x8` i `x5`. Finalment, `x6` i `x7` ocupen el nivell 4.

  Més formalment:

  – L'arrel d'un arbre binari ocupa el nivell 1.

  - Si un node d'un arbre binari ocupa el nivell i, els dos nodes fill esquerre i fill dret d'aquest arbre ocupen el nivell $i + 1$.

- Un **camí** és una llista de nodes que porten des d'un node (usualment, l'arrel) fins a un altre (moltes vegades, una fulla o un element terminal).

  Per exemple, a l'arbre de la figura 3.2, `{x1, x3, x5, x7}` és un camí que duu des de l'arrel fins a la fulla `x7`.

- L'**alçada** d'un arbre és la longitud del camí més llarg entre l'arrel i una fulla. O, si voleu, la longitud del camí entre l'arrel i la fulla més llunyana.

  L'alçada de l'arbre amb què estem treballant tota l'estona és 4.

3.1.5 Propietats dels arbres binaris

Les propietats següents es compleixen per als arbres binaris:

1. El nombre màxim de nodes al nivell k d'un arbre binari és 2^{k-1}.
2. El nombre màxim de nodes d'un arbre binari d'alçada k és $2^k - 1$.
3. El nombre de fulles en un arbre binari no buit és una més que el nombre de nodes de grau 2 (amb dos fills no buits).

   O sigui:

   Si n_0 és el nombre de fulles d'un arbre binari no buit i n_2 és el nombre de nodes amb dos fills no buits en aquell mateix arbre es té:

   $n_0 = n_2 + 1$

Les dues primeres propietats es demostren per inducció sobre l'alçada de l'arbre binari. La darrera, a partir de l'observació de l'estructura d'un arbre binari. Les sabríeu demostrar en detall?

Al capítol 4 veurem que els arbres binaris ens proporcionen una estructura de dades molt convenient per fer cerques. Però no tots els arbres binaris seran estructures adients per tal de fer cerques. Necessitarem que estiguin *molt plens*:

Anomenem **arbre binari complet** a un arbre d'alçada k que té $2^k - 1$ nodes (i.e., el nombre màxim de nodes).

Observem que, aplicant la propietat 2 anterior, l'alçada d'un arbre binari complet que conté n nodes és $\log(n + 1)$.

Aquesta propietat és la clau per tal de fer cerques eficients en un arbre binari complet: si aconseguim desenvolupar algoritmes de cerca que a cada pas de l'algoritme baixin un nivell

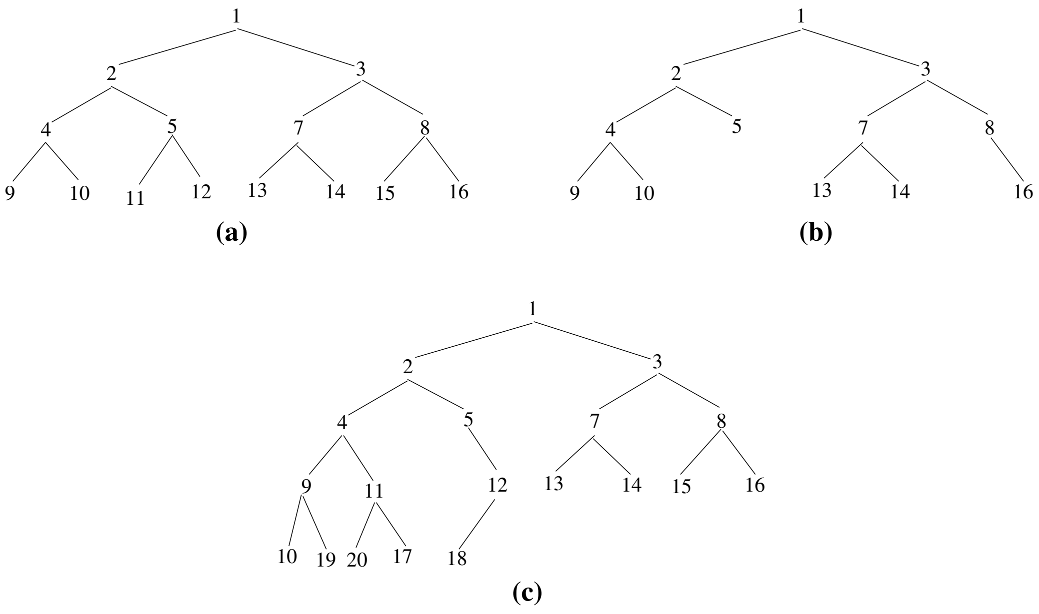


Figura 3.3: Arbre binari complet (a), equilibrat (b) i no equilibrat (c)

en l'arbre, en un nombre logarítmic de passos (respecte del nombre total de nodes de l'arbre) haurem aconseguit trobar la dada que buscàvem. I un nombre logarítmic de passos és un nombre molt baix. Els algoritmes que es comporten d'aquesta manera es diu que tenen un cost $O(log\ n)$, on n és el nombre de nodes de l'arbre. Noteu que un algoritme $O(log\ n)$ és capaç de tractar el doble de dades amb només un pas més. Per què?

Malauradament, no sempre disposarem d'arbres complets. Hi ha uns altres arbres binaris que no són complets però que també es comporten bé en les cerques: són els arbres equilibrats.

Anomenem **arbre binari equilibrat** un arbre binari que, per a qualsevol node de l'arbre, presenta un valor absolut de la diferència d'alçada entre el seu subarbre fill dret i el seu subarbre fill esquerre menor o igual que 1.

El comportament dels arbres binaris equilibrats respecte de les cerques també serà bo. En particular, també aconseguiran un cost $O(log\ n)$ en les cerques.

La figura 3.3 mostra un arbre binari complet (a), un d'equilibrat (b) i un de no equilibrat (c). Per què aquest darrer arbre no és equilibrat?

3.1.6 Recorreguts d'arbres

Un aspecte fonamental per als arbres és la possibilitat de recórrer-los. Això és, de *visitar tots els seus elements*. Però, en quin ordre els visitem?

Una llista tenia definida una ordenació dels seus elements. Aquesta ordenació donava el recorregut natural de la llista. Per això, els iteradors recorrien la llista en l'ordre en què estaven situats els seus elements (o en l'ordre invers). O sigui, començant pel primer element i avançant fins al darrer o començant pel darrer i retrocedint fins al primer. Però en un arbre no hi ha un primer i darrer elements "naturals". Podríem argumentar que el primer pot ser l'arrel, però quin és el segon, el tercer i el darrer?

En realitat, tenim dues maneres "naturals" de recórrer un arbre: *en amplada* i *en fondària*.

Recorregut d'un arbre en amplada

El **recorregut d'un arbre en amplada** té la propietat següent:
Només es visita un node de nivell i si s'han visitat prèviament tots els nodes de nivell $i - 1$.

És clar que recórrer en amplada un arbre correspon a recórrer-lo **per nivells**: o sigui, primer els nodes de nivell 1, després els de nivell 2... Per això aquest recorregut s'anomena *en amplada* o *per nivells*.

L'algoritme per recórrer un arbre binari en amplada o nivells usa una cua (vegeu la secció 3.2.3).

Exemple 3.5: Recorregut en amplada d'un arbre

Tornem a fixar-nos en la figura 3.2. El recorregut d'aquest arbre en amplada és:

`{x1,x2,x3,x4,x8,x5,x6,x7}`

■ ■ ■

Recorregut d'un arbre en fondària

El **recorregut d'un arbre en fondària** es basa en la idea següent:
Només es visita un altre node del mateix nivell que un node `nod` *si s'han visitat prèviament tots els descendents de* `nod`.

Els algoritmes per recórrer un arbre binari en fondària són naturalment recursius o usen una pila (vegeu la secció 3.2.2).

El recorregut en fondària d'un arbre té una ambigüïtat: si partim d'un node (posem `x1` a l'exemple de la figura 3.2), sabem que no en podem visitar cap altre del mateix nivell

(en aquest cas no n'hi ha cap més perquè `x1` és l'arrel) fins que no hàgim visitat tots els descendents d'`x1`. Això vol dir: visitar el fill esquerre d'`x1`, visitar el fill dret d'`x1` i visitar el mateix `x1`. Però en quin ordre fem aquestes tres visites? Si mantenim el criteri de visitar sempre abans el fill esquerre que el dret, tenim tres ordres possibles:

- **Preordre:** visita l'arrel, després recorre el fill esquerre i, finalment, el dret.
- **Inordre:** recorre el fill esquerre, visita l'arrel i, finalment, recorre el dret.
- **Postordre:** recorre el fill esquerre, després el dret i, finalment, visita l'arrel.

S'entén que, en tots tres casos, els fills (esquerre o dret) es recorren amb el mateix tipus de recorregut amb el que s'està recorrent l'arbre. Per exemple, en el recorregut en preordre, es visita l'arrel i després es recorre el fill esquerre *en preordre* i el dret *també en preordre*.

Exemple 3.6: Recorregut en fondària d'un arbre

Recorrem l'arbre de la figura 3.2 en fondària amb els tres recorreguts presentats:

- *Preordre:* `x1, x2, x4, x3, x8, x5, x6, x7`
- *Inordre:* `x2, x4, x1, x8, x3, x6, x5, x7`
- *Postordre:* `x4, x2, x8, x6, x7, x5, x3, x1`

■ ■ ■

Quan hàgim presentat la interfície `BinaryTree<T>` podrem donar els algoritmes per recórrer arbres. Si ja esteu impacients, n'hi ha prou que continueu llegint.

3.2 Especificant els arbres binaris

En aquesta secció proposarem una interfície `BinaryTree<T>` que proporcioni les operacions necessàries per treballar amb arbres binaris. Aquesta interfície i les seves operacions reflectiran la naturalesa recursiva dels arbres binaris. Insistirem en aquesta naturalesa recursiva a la secció 3.3 en què implementarem els arbres binaris.

Un cop hàgim proporcionat una interfície per treballar amb els arbres binaris, donarem una sèrie de propietats dels arbres binaris i proposarem un parell d'exemples en què veurem com s'utilitza la interfície proposada. Som-hi.

3.2.1 Especificació dels arbres binaris

La interfície `BinaryTree<T>` que permetrà treballar amb arbres binaris d'elements de tipus `T` *no forma part de la JCF* però la definirem aprofitant la jerarquia d'interfícies que constitueixen la JCF.

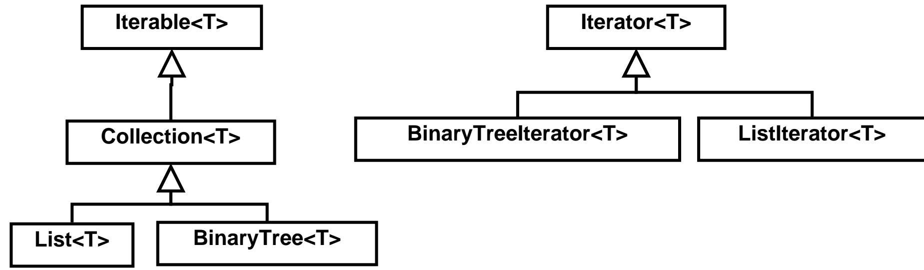


Figura 3.4: Situació de les interfícies `BinaryTree<T>` i `BinaryTreeIterator<T>` a la jerarquia d'interfícies de contenidors

En particular, un arbre binari el podem veure com una col·lecció d'elements de tipus `T`. Més encara, té sentit que iterem sobre els elements d'un arbre binari: de fet, a la secció 3.1.6, hem presentat diferents recorreguts –o iteracions– possibles sobre els arbres binaris. Per tant, té sentit que la interfície `BinaryTree<T>` sigui subinterfície de `Collection<T>` i, per tant, d'`Iterable<T>`. A més a més, de la mateixa manera que a les llistes vam definir una interfície d'iteradors específica per recórrer-les (`ListIterator<T>`), també ara podem definir-ne una per als arbres binaris. L'anomenarem `BinaryTreeIterator<T>`. La situació d'aquestes interfícies a la jerarquia de la JCF es mostra a la figura 3.4

`BinaryTree<T>`

La interfície `BinaryTree<T>` heretarà les operacions típiques de `Collection<T>` i `Iterable<T>` per tal de treballar amb col·leccions... però, quines operacions específiques haurà de definir per tal de treballar amb arbres binaris?

- Un arbre binari ha estat definit recursivament en termes d'un element arrel (de tipus `T`) i dos arbres fills: el dret i l'esquerre. Per tant, tindrà sentit definir operacions per obtenir aquests elements:

```
1     BinaryTree<T> getLeftCh();
2     BinaryTree<T> getRightCh();
3     T getRoot();
```

- Igualment, d'un arbre binari podrem eliminar el fill dret o l'esquerre. Això ens suggereix dues noves operacions:

```
1     void removeLeftCh();
2     void removeRightCh();
```

- També té sentit oferir una operació per obtenir l'alçada de l'arbre:

```
1     int height();
```

- Finalment, per recórrer una col·lecció, la interfície `Iterable<T>` ens proporciona l'operació `iterator()`. Però aquesta operació és insuficient per als arbres, atès que disposem d'almenys 4 maneres diferents de recórrer-los: per nivells, preordre, inordre i postordre. Per tant, proporcionarem les operacions addicionals següents:

```
1  Iterator<T> iterator();
2  BinaryTreeIterator<T> iteratorPre();
3  BinaryTreeIterator<T> iteratorIn();
4  BinaryTreeIterator<T> iteratorPost();
5  BinaryTreeIterator<T> iteratorLevels();
```

- Noteu que no hem inclòs operacions d'inserció de fills esquerre o dret. No ho hem fet perquè aquestes operacions no són naturals en un arbre binari que no s'ha definit per a un propòsit concret. Penseu un moment: què voldria dir afegir un fill a l'arbre que té arrel `x3` a la figura 3.2? Aquest fill, l'afegim com a fill dret o com a fill esquerre? Què fem amb el fill dret o esquerre que ara té l'arbre arrelat a `x3`?

  En canvi, si haguéssim definit un arbre binari per a un propòsit concret (per exemple, un *arbre binari de cerca* (vegeu capítol 4) o una cua amb prioritat (implementada amb l'ajut d'un arbre binari)) aleshores sí que tindria un sentit ben precís afegir un nou element a l'arbre i així ho farem quan presentem aquestes estructures.

  El requadre següent proposa una reflexió al voltant d'aquesta idea.

El Java no inclou la interfície `BinaryTree<T>` ni tampoc cap classe que tingui com a objectiu la implementació dels arbres binaris.
Això és degut a que les aplicacions, en general, no usen els arbres (i, en particular, els binaris) directament. És molt freqüent que una aplicació necessiti usar una *llista* per emmagatzemar-hi elements en un cert ordre, o una *taula* per poder tenir accés ràpidament a un element a través de la seva clau (estudiarem les taules al capítol 4), o una *cua* per tenir constància de l'ordre en què ha arribat una col·lecció d'elements que cal tractar... En canvi, les aplicacions (en general, ho repetim) no necessiten usar arbres directament. Són, doncs, els arbres, inútils?
Ben al contrari, els arbres resulten utilíssims *per implementar altres estructures*: proporcionen implementacions molt bones per a taules, conjunts, cues amb prioritat, gestors de tornejos, llistes del llenguatge de programació *Lisp*... Però cadascuna d'aquestes estructures usa els arbres d'una manera diferent, amb unes operacions diferents. Per exemple, cada estructura específica entendrà d'una manera diferent el sentit d'afegir (eliminar) un element a (de) un arbre o de construir un arbre o...
Per això el Java no defineix directament la interfície o la classe `BinaryTree<T>` però sí que defineix les classes `TreeMap<T>`, `TreeSet<T>` o `PriorityQueue<T>` que usen arbres per implementar una taula, un conjunt o una cua amb prioritat, respectivament.
Nosaltres, en aquest curs, tenim les estructures de dades com a objecte d'estudi i, per aquest motiu, definirem una interfície `BinaryTree<T>` que contingui les operacions *més pures dels arbres binaris* i alguna classe que implementi aquesta interfície. Segurament, a les nostres aplicacions no usarem directament aquesta interfície o les classes que la implementin, però ens serviran per comprendre les tècniques de treball amb arbres, les quals ens seran utilíssimes quan, més endavant, vulguem usar arbres per representar una taula, una cua amb prioritat o una llista de `Lisp`.

Amb les operacions que hem indicat i les heretades de `Collection<T>`, la interfície `BinaryTree<T>` queda de la manera següent:

```
1  public interface BinaryTree<T> extends Collection<T> {
2      void clear();
3      boolean contains(Object o);
4      boolean containsAll(Collection<?> c);
5      boolean equals(Object o);
6      BinaryTree<T> getLeftCh();
7      BinaryTree<T> getRightCh();
8      T getRoot();
9      int height();
10     boolean isEmpty();
11     Iterator<T> iterator();
12     BinaryTreeIterator<T> iteratorPre();
13     BinaryTreeIterator<T> iteratorIn();
14     BinaryTreeIterator<T> iteratorPost();
15     BinaryTreeIterator<T> iteratorLevels();
16     void removeLeftCh();
17     void removeRightCh();
18     int size();
19     Object[] toArray();
20 }
```

La taula següent mostra l'especificació de les operacions més rellevants:

<table>
<tr><td><code>boolean contains(Object obj)</code>
Retorna cert si aquest arbre conté un element <code>elem</code> de manera que <code>elem.equals(obj)</code> retorna cert.</td></tr>
<tr><td><code>boolean equals(Object obj)</code>
Retorna cert si aquest arbre:
és l'arbre binari buit i <code>obj</code> també, o bé
té una arrel <code>elemRoot</code> de manera que
<code>elemRoot.equals(obj.getRoot())</code> és cert i
<code>this.getLeftChild().equals(obj.getLeftChild()</code> és cert i
<code>this.getRightChild().equals(obj.getRightChild()</code> és cert.</td></tr>
<tr><td><code>BinaryTree<T> getLeftCh()</code>
Retorna l'arbre binari fill esquerre d'aquest arbre.
Llença:
<code>NoSuchElementException</code> si aquest arbre és l'arbre binari buit.</td></tr>
<tr><td><code>BinaryTree<T> getRightCh()</code>
Retorna l'arbre binari fill dret d'aquest arbre.
Llença:
<code>NoSuchElementException</code> si aquest arbre és l'arbre binari buit.</td></tr>
<tr><td><code>T getRoot();</code>
Retorna l'arrel d'aquest arbre.
Llença:
<code>NoSuchElementException</code> si aquest arbre és l'arbre binari buit.</td></tr>
</table>

Especificació BinaryTree<T> (continuació)
int height(); Retorna l'alçada d'aquest arbre.
Iterator<T> iterator() Retorna un iterador en preordre sobre aquest arbre. El primer element retornat per l'operació next() serà el primer element de l'arbre en preordre.
BinaryTreeIterator<T> iteratorPre() Retorna un iterador en preordre sobre aquest arbre. El primer element retornat per l'operació next() serà el primer element de l'arbre en preordre.
BinaryTreeIterator<T> iteratorIn() Retorna un iterador en inordre sobre aquest arbre. El primer element retornat per l'operació next() serà el primer element de l'arbre en inordre.
BinaryTreeIterator<T> iteratorPost() Retorna un iterador en postordre sobre aquest arbre. El primer element retornat per l'operació next() serà el primer element de l'arbre en postordre.
BinaryTreeIterator<T> iteratorLevels() Retorna un iterador per nivells sobre aquest arbre. El primer element retornat per l'operació next() serà el primer element de l'arbre per nivells.
void removeLeftCh() Elimina, en cas d'existir, el fill esquerre d'aquest arbre.
void removeRightCh() Elimina, en cas d'existir, el fill dret d'aquest arbre.

BinaryTreeIterator<T>

Proposem la següent interfície BinaryTreeIterator<T>:

```
1 public interface BinaryTreeIterator<T> extends Iterator<T> {
2     boolean hasNext();
3     T next();
4     void set(T o);
5 }
```

La interfície BinaryTreeIterator<T> que proposem:

- Inclou les operacions definides a Iterator<T>: hasNext() i next() amb el significat esperat.

  Queda clar que per a cada classe C que implementi BinaryTreeIterator<T>, l'operació next() retornarà el següent element en preordre, inordre, postordre o nivells depenent de l'estratègia d'iteració que tingui C.

- No inclou l'operació Iterator<T>.remove(), definida a Iterator<T> com a opcional. Ja hem comentat abans que l'estratègia d'eliminació d'un node a un arbre és una

operació que depèn de l'aplicació concreta que es vulgui donar a aquell arbre. Com que nosaltres estem definint uns arbres binaris sense cap propòsit concret, no inclourem operacions per afegir (eliminar) elements als (dels) arbres.

Per tant, les classes que implementin la interfície `BinaryTreeIterator<T>` contindran el codi següent:

```
1 public void remove() {
2     throw new UnsupportedOperationException();
3 }
```

- Afegeix l'operació:

`void set(T obj)`
Canvia el darrer element retornat per `next()` per `obj`.
Llença:
`IllegalStateException` si no s'ha cridat encara `next()`.

3.2.2 Exemple: recorregut d'un arbre binari en preordre

Un cop especificada la interfície `BinaryTree<T>`, podem usar-la. En aquesta secció us proposem implementar un recorregut d'un arbre binari en preordre.

Podem implementar un recorregut d'un arbre en preordre de dues maneres: usant els iteradors que ens proporciona la interfície `BinaryTree<T>` o bé, sense usar-los. Fem-ho de les dues maneres:

Recorregut en preordre usant iteradors

Considereu l'operació següent:

Llistat 3.1: Recorregut en preordre d'un arbre binari

```
1   public <T> List<T> preorderTraversal(BinaryTree<T> t) {
2       List<T> lis = new ArrayList<T>();
3
4       if (t == null) return lis;
5
6       BinaryTreeIterator<T> it = t.iteratorPre();
7       while (it.hasNext()) {
8           lis.add(it.next());
9       }
10      return lis;
11  }
```

Comentaris:

- La instrucció de la línia 6 retorna un iterador sobre l'arbre `t` en preordre. Per tant, les línies següents simplement van obtenint els diferents elements de l'arbre en l'ordre

en què els subministra l'iterador (i.e., en preordre) i els van col·locant a una llista que finalment és retornada.

- Noteu que abans de fer `t.iteratorPre()` ens hem d'assegurar que l'arbre `t` no sigui `null`. Si ho fos la instrucció `t.iteratorPre()` llençaria una excepció. Quina?

Podem fer una reflexió sobre el codi del llistat 3.1: oi que si volguéssim implementar un recorregut en inordre, postordre o nivells, l'única diferència en el codi seria la línia 6?

Per exemple, si volguéssim un recorregut per nivells, la línia 6 hauria de ser:

```
1     BinaryTreeIterator<T> it = t.iteratorLevels();
```

La resta del codi fóra exactament igual.

Aprofitant aquesta idea, us proposo el problema d'implementar una nova operació `traversal(...)` modificant el que cregueu oportú de `preorderTraversal` de manera que la nova operació pugui servir per recórrer un arbre en qualsevol estratègia de recorregut (nivells, preordre, inordre, postordre).

Recorregut en preordre sense usar iteradors

Usant les operacions que proporciona la interfície `BinaryTree<T>` i aprofitant la definició recursiva dels arbres binaris podem proposar un algoritme recursiu de recorregut en preordre d'un arbre sense usar iteradors. La idea de l'algoritme s'obté immediatament de la definició de recorregut en preordre:

> Primer cal visitar l'arrel, després recórrer en preordre el fill esquerre i, finalment, recórrer en preordre el fill dret.

Per tant, ens queda:

```
1     public <T> List<T> preorderTraversalRec(BinaryTree<T> t) {
2
3         if (t == null) return new ArrayList<T>();
4
5         List<T> lis = new ArrayList<T>();
6
7         lis.add(t.getRoot());
8         lis.addAll(preorderTraversalRec(t.getLeftCh()));
9         lis.addAll(preorderTraversalRec(t.getRightCh()));
10
11        return lis;
12    }
```

Comentaris:

- Recordeu que les operacions de `List<T>`: `add` i `addAll` afegeixen un element i tots els elements d'una col·lecció al final de la llista sobre la qual s'apliquen.

Aquest algoritme planteja de forma natural dues reflexions:

1. Com es faria un recorregut recursiu en postordre o inordre? La resposta és immediata i us la deixo com a exercici.
2. I si no volguéssiu fer servir la recursivitat?

   Aleshores hauríeu de dissenyar un algoritme molt semblant al que presento a la secció 3.2.3. Això sí: us caldria usar una pila i no una cua com s'usa allí. Per què no doneu un cop d'ull a la classe `Stack<T>` de la JCF i intenteu proposar un algorisme no recursiu que faci un recorregut en preordre sense usar iteradors?

   I, ja que hi sou, per què no feu el mateix en inordre i postordre?

3.2.3 Exemple: recorregut d'un arbre binari per nivells

Plantejarem directament un recorregut per nivells sense usar iteradors. Si voleu usar iteradors, n'hi ha prou que doneu un cop d'ull a 3.2.2.

Recordem la definició de recorregut d'un arbre per nivells o en amplada:

> En un recorregut d'un arbre en amplada o per nivells:
> *Només es visita un node de nivell i si s'han visitat prèviament tots els nodes de nivell $i-1$.*

Els nodes de nivell i són:

- L'arrel (si $i = 1$) o bé
- els fills de nodes de nivell $i-1$.

Per tant, descobrirem els nodes de nivell i quan visitem els nodes de nivell $i-1$ perquè són els seus fills. L'arrel la "descobrirem" a l'inici de l'algoritme de recorregut.

El procés de visita d'un node consistirà a:

1. Posar-lo al final de la llista que contindrà els elements de l'arbre per nivells i que retornarem.
2. Descobrir els seus fills i no visitar-los directament sinó que els afegirem a una cua.

   El primer node que posarem a la cua serà l'arrel, l'únic node de nivell 1 i, per tant, el primer que ha de ser visitat.

D'aquesta manera, en tot moment, el primer element de la cua (`nod`) és el proper node que ha de ser visitat. El procés acaba quan la cua és buida i, per tant, ja no queden nodes per visitar. Vegem-ho:

```
1  public <T> List<T> breadthFirstTraversal(BinaryTree<T> t) {
2      Queue<BinaryTree<T>> q = new LinkedList<BinaryTree<T>>();
3      BinaryTree<T> troot, leftCh, rightCh;
4      List<T> lis = new ArrayList<T>();
5
6      if (t != null && !t.isEmpty()) q.offer(t);
7      while (!q.isEmpty()) {
8          troot = q.poll();
9          lis.add(troot.getRoot());
10
11         leftCh = troot.getLeftCh();
12         rightCh = troot.getRightCh();
13         if (!leftCh.isEmpty()) q.offer(leftCh);
14         if (!rightCh.isEmpty()) q.offer(rightCh);
15     }
16     return lis;
17 }
```

Comentaris:

- Quan diem que *la cua* `q` *conté el node* `nod`, més exactament volem dir que *la cua* `q` *conté el subarbre arrelat a* `nod`.

- L'invariant del bucle de l'algorisme anterior és aquest:

Invariant del bucle:

La cua `q` conté (en aquest ordre):

1. Tots els nodes d'un determinat nivell (e.g. i) que encara no s'han visitat i

2. tots els nodes de nivell $i+1$ corresponents als fills dels nodes de nivell i que ja s'han visitat.

Tots els nodes de nodes de nivell menor al del primer de la cua $(1 \ldots i-1)$ ja han estat visitats.

Per tant, el primer de la cua és el proper element que ha de ser visitat.

- La línia 6 introdueix a la cua l'arrel de l'arbre abans de visitar-la. Però només si `t` no és un arbre buit. Com a conseqüència, noteu que es retornarà la llista buida en el cas que `t` sigui buit.

- La primera instrucció del bucle (vegeu la línia 8) obté el primer element de la cua per tal de visitar-lo (`troot`).

- Noteu que només col·loquem els fills del node que estem visitant (`troot`) si no són buits.

3.3 Implementant els arbres binaris

3.3.1 Dues propostes per a la implementació dels arbres binaris

1. Amb un vector

   Aquesta estratègia va bé per representar arbres binaris complets o molt plens. També s'utilitza per representar cues amb prioritat.

2. Amb enllaços

   Cada element de l'arbre es posa en un node semblant als que usàvem per a les llistes enllaçades (vegeu el capítol 2). En aquest cas, cada node tindrà dos punters: un al node arrel del seu fill esquerre i l'altre al node arrel del seu fill dret.

3.4 Implementant els arbres binaris amb un vector

Un arbre binari `t` que té n nodes es pot representar en forma de vector usant el criteri següent:

- L'arrel de `t` està situada a l'índex 1 del vector (l'índex 0 del vector no s'usarà en aquesta representació).
- Si `nod` és un node de `t` situat a l'índex i del vector, aleshores:
  - El *node fill esquerre* de `nod` està situat a l'índex $2i$ del vector.
    Si $2i > n$, vol dir que `nod` no té fill esquerre.
  - El *node fill dret* de `nod` està situat a l'índex $2i + 1$ del vector.
    Si $2i + 1 > n$, vol dir que `nod` no té fill dret.

La figura 3.5 mostra la representació d'un arbre binari usant un vector.

Aquesta representació compleix les propietats següents:

Propietats de la representació dels arbres binaris amb vectors

Si `t` és un arbre binari:

1. Aquesta representació assigna un índex diferent del vector que representa `t` a cada node de `t`.

2. Suposem que `nod` és un node de `t`, diferent de l'arrel, que ocupa l'índex i del vector que representa t. Aleshores, el pare de `nod` ocupa l'índex i div 2 del mateix vector.

3. Si `t` té n nodes, el vector que el representa haurà de tenir, en el cas pitjor, una capacitat (`length`) $c = 2^n$ nodes.

Aquestes propietats mereixen alguns comentaris:

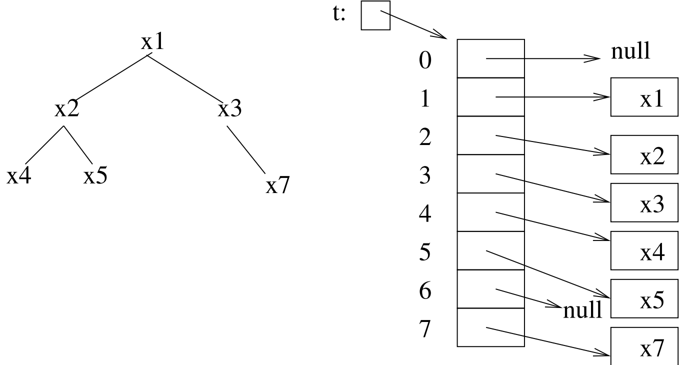


Figura 3.5: Representació d'un arbre binari `t` mitjançant un vector

- Aquesta representació permet d'una manera molt simple accedir tant als fills dret i esquerre com al pare del node que ocupa l'índex i.

  Efectivament, el fill esquerre ocuparà l'índex $2i$. El fill dret, l'índex $2i + 1$ i el pare, l'índex i div 2.

  Comproveu-ho a la figura 3.5.

- La darrera propietat és especialment crítica i limita enormement l'ús d'aquesta representació en forma de vector per als arbres. Podeu imaginar per què?

  Efectivament, aquesta representació malbarata moltíssim l'espai per a arbres no complets. La capacitat que hauria de tenir un vector per tal de representar un arbre d'n nodes és, com a màxim (i segons estableix la propietat 3) de 2^n índexs.

  Si el nostre arbre binari és complet, segons establien les propietats presentades a la secció 3.1.5, el seu nombre de nodes és precisament $2^n - 1$. Aquest nombre coincideix amb la capacitat màxima del vector que necessitem per representar-lo (2^n; un índex més que $2^n - 1$ perquè l'índex 0 del vector no s'usa en la representació presentada).

> O sigui, la representació d'un arbre complet en forma de vector és òptima en el sentit que cada índex del vector (llevat de l'índex 0) estarà ocupat per un node de l'arbre.
> No es malbaratarà gens d'espai del vector.

Aquesta situació es mostra a la figura 3.6(a).

Si, per contra, l'arbre binari que volem representar no és complet aquesta representació malbarata molt l'espai.

El cas més espectacular és el de la figura 3.6(b). L'arbre té 6 nodes, i, per aquest motiu, segons la propietat 1, necessita un vector amb un màxim de $2^6 = 64$ índexs per representar-lo. En aquest cas es necessiten tots 64 ja que, com que tots els nodes són fills drets, el darrer ocupa l'índex 63.

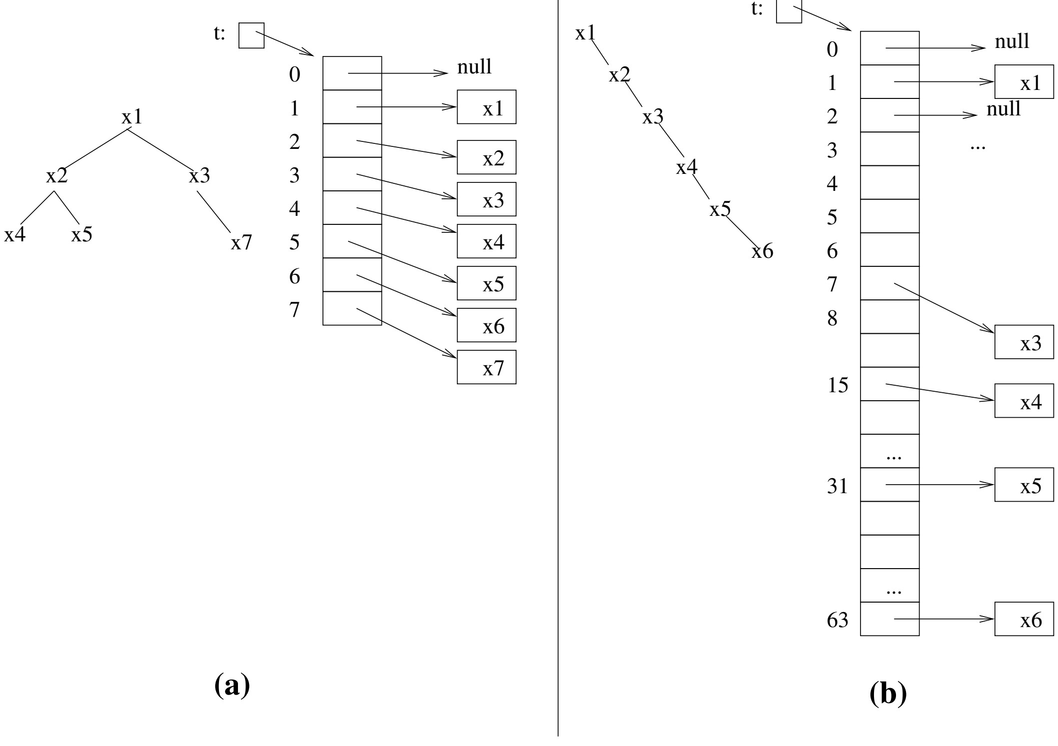


Figura 3.6: Representació d'un arbre binari complet i un de no complet mitjançant un vector

D’aquests 64 índexs del vector, només n’ocupem 6. Malbaratem molt espai.

En realitat, l’espai que malbaratem és prohibitiu. Imagineu-vos un arbre amb **només 100 nodes** (que poc, direu!). Quina capacitat de vector li hauríem de reservar en el cas pitjor? Segons la propietat 1:

$$2^{100} = 1267650600228229401496703205376$$

Si cada índex del vector ocupa un byte (únicament un byte!), l’espai que ocuparia aquest vector fóra de l’ordre de 1267.650.600.228.229.401.496 gigabytes!

O sigui, 1267 trilions de gigabytes. O, si voleu, 1267 milions de bilions de gigabytes. O, si voleu, encara: 1267 milions de milions de milions de gigabytes!

Algú disposa d’un disc amb aquesta capacitat a casa seva?

Encara pitjor, d’aquesta quantitat immensa de bytes, **l’arbre només n’ocuparia 100!**. La resta estarien buits! Usant un adjectiu que està de moda darrerament, podríem dir que la representació d’un arbre binari mitjançant un vector és *insostenible* per a qualsevol altre arbre que no sigui complet o molt dens.

Però la cosa pot ser molt pitjor encara. Imagineu-vos ara un arbre de 1.000 nodes. No es tracta pas d’un arbre descomunal. Així i tot, quants bytes ocuparia el vector necessari per emmagatzemar-lo?

$$\begin{aligned} 2^{1000} = \quad & 10715086071862673209484250490600018105614048117055336074437503883 \\ & 7035105112493612249319837881569585812759467291755314682518714528 \\ & 5692314043598457757469857480393456777482423098542107460506237114 \\ & 1877954182153046474983581941267398767559165543946077062914571196 \\ & 477686542167660429831652624386837205668069376 \end{aligned}$$

Podeu comparar-lo amb aquell nombre tan enorme que hem estat discutint abans:

$$2^{100} = 1267650600228229401496703205376$$

La representació d’un arbre binari mitjançant un vector s’ha de reservar per als casos d’arbres binaris complets o molt densos.

3.5 Implementant els arbres binaris amb enllaços

En aquesta secció presentarem tres maneres d’implementar els arbres binaris amb enllaços. La primera és la bàsica i correspon amb el que hem explicat a la secció 3.3.1. En determinades circumstàncies (per exemple, per tal de treballar amb iteradors de manera més eficient o bé en el cas que l’accés al pare d’un node sigui important) podem requerir variants d’aquesta implementació bàsica: amb punters al pare o amb enfilaments. Parlem-ne.

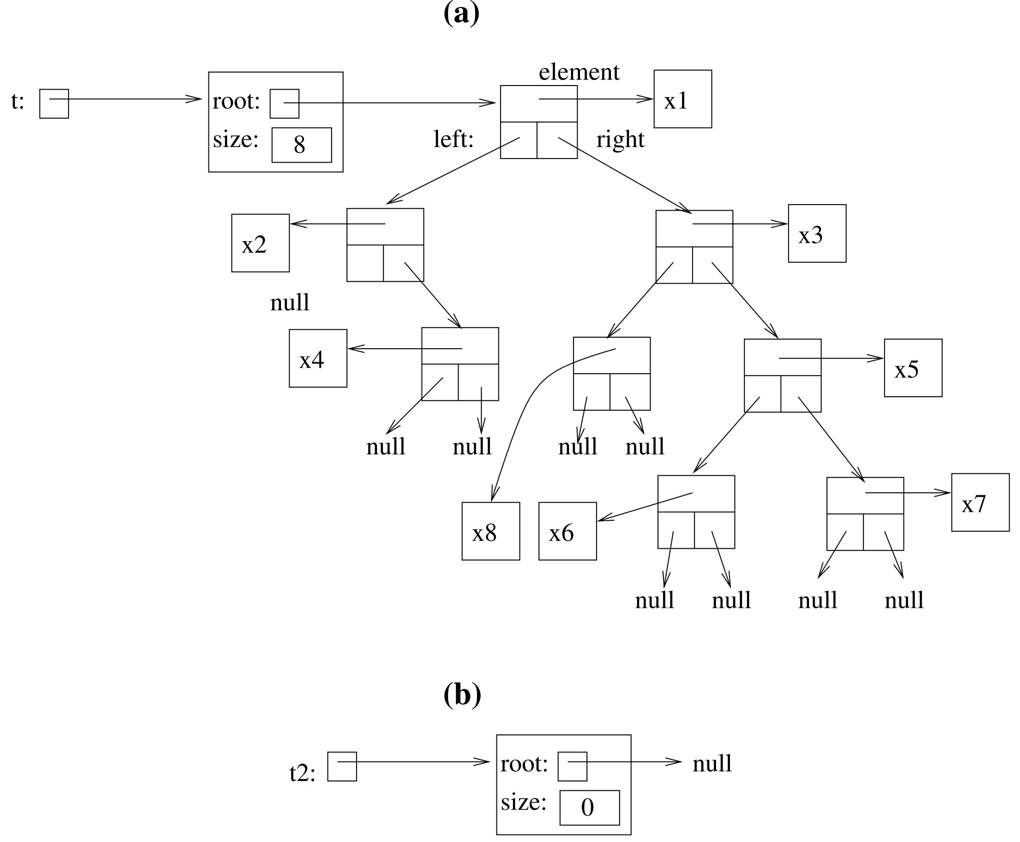


Figura 3.7: Representació amb enllaços bàsica dels arbres binaris

3.5.1 Implementació bàsica

Com ja hem esbossat a la secció 3.3.1, la representació més bàsica d'un arbre binari amb enllaços és la següent:

Un arbre binari es representa mitjançant un atribut `root` que conté una referència al **node arrel** i un altre atribut `size` que indica el nombre d'elements que conté l'arbre binari.

A la vegada, cada node conté una referència a l'**element** contingut en aquell node, al seu **fill esquerre** i al seu **fill dret**.

La figura 3.7 mostra aquesta representació. En particular, (a) mostra com es representaria internament l'arbre binari de la figura 3.2 i (b) mostra la representació d'un arbre binari buit.

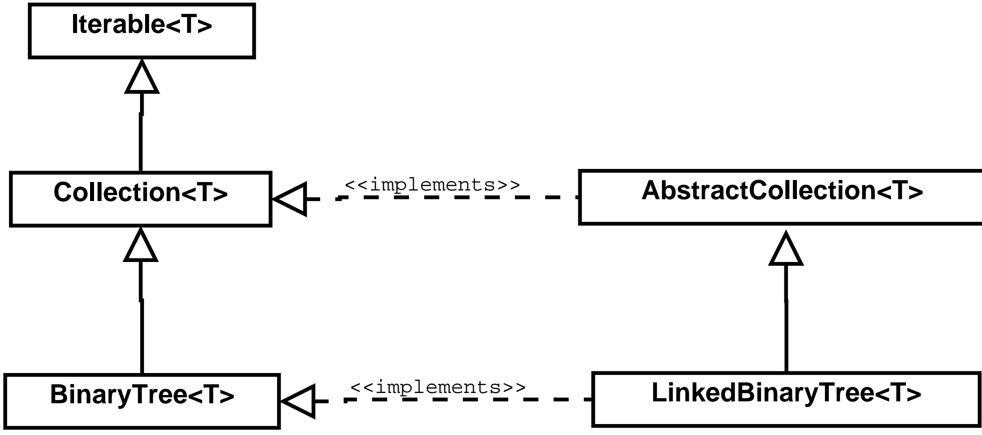


Figura 3.8: La classe `LinkedBinaryTree<T>` a la jerarquia de contenidors

La classe `LinkedBinaryTree<T>` conté la implementació bàsica dels arbres binaris amb enllaços. Com ja vam veure al capítol 2 en el context de les llistes, resulta una bona idea definir `LinkedBinaryTree<T>` com a subclasse de la classe de la JCF `AbstractCollection<T>` (vegeu la figura 3.8). Efectivament, si ho fem així, hi haurà tot un seguit d'operacions de `LinkedBinaryTree<T>` que *no ens caldrà implementar* perquè ja estan implementades a `AbstractCollection<T>`.

I com pot ser possible això? Com és possible que els dissenyadors de la JCF, molt de temps abans que existís la classe `LinkedBinaryTree<T>` (i sense saber-ne, per tant, la representació) ja poguessin implementar operacions d'aquesta classe?
Per contestar aquesta pregunta, doneu un cop d'ull a la secció 2.7.1, del capítol 2. La idea fonamental és que hi ha operacions d'una classe que es poden implementar en termes d'altres operacions de la classe. Un cas típic és l'ús d'iteradors per implementar operacions. Considereu, per exemple, la implementació que `AbstractCollection<T>` fa de `contains(obj)` i comproveu que, efectivament, constitueix una implementació vàlida per a `LinkedBinaryTree<T>.contains(Object o)`, independentment de quina sigui la representació triada per a `LinkedBinaryTree<T>`:

```
1  public boolean contains(Object o) {
2      Iterator<E> e = iterator();
3      if (o == null) {
4          while (e.hasNext())
5              if (e.next()==null)
6                  return true;
7      } else {
8          while (e.hasNext())
9              if (o.equals(e.next()))
10                 return true;
11     }
12     return false;
13 }
```

Així i tot, a vegades convé redefinir una operació heretada de la superclasse per tal de donar-li una implementació més eficient. Aquest serà precisament el cas de `contains(o)`. Més avall en parlem.

Finalment, recordem que `LinkedBinaryTree<T>` implementa `BinaryTree<T>`. Amb tot, ens queda:

Llistat 3.2: Representació de `LinkedBinaryTree<T>`

```
1  public class LinkedBinaryTree<T> extends AbstractCollection<T>
2                                   implements BinaryTree<T> {
3      private Entry<T> root;
4      int size;
5
6      private static class Entry<T> {
7          T element;
8          Entry<T> left;
9          Entry<T> right;
10         Entry(T element, Entry<T> left, Entry<T> right) {
11             this.element = element;
12             this.left = left;
13             this.right = right;
14         }
15     }
16 }
```

Ara implementem les operacions de `LinkedBinaryTree<T>`.

Constructores

```
1     public LinkedBinaryTree() {
2         root = null;
3         size = 0;
4     }
5
6     public LinkedBinaryTree(T elem,
7                             LinkedBinaryTree<T> left,
8                             LinkedBinaryTree<T> right) {
9         Entry<T> leftCh = (left == null ? null : left.root);
10        Entry<T> rightCh = (right == null ? null : right.root);
11
12        int leftS = (left == null ? 0 : left.size);
13        int rightS = (right == null ? 0 : right.size);
14
15        root = new Entry<T>(elem,leftCh,rightCh);
16        size = leftS + rightS + 1;
17    }
18
19    private LinkedBinaryTree(Entry<T> root) {
20        this.root = root;
21        this.size = size(root);
22    }
23
24    private int size(Entry<T> root) {
25       if (root == null) return 0;
26       return size(root.left) + size(root.right) + 1;
27    }
```

Comentaris:

- La constructora sense paràmetres genera un arbre binari buit.
- La segona constructora genera l'arbre binari que té com a arrel `elem`, com a fill esquerre, l'arbre `left`, i com a fill dret, l'arbre `right`.

  Cal anar amb compte amb aquesta constructora perquè si `left` o `right` fossin `null` no es podria accedir a `left.root` o `right.root`. Quina excepció es llençaria?
- La tercera constructora:

  ```
  1   private LinkedBinaryTree(Entry<T> root)
  ```

  genera un arbre que té com a arrel el node `root`. Aquesta constructora la usarem en operacions com ara `getLeftCh()` (vegeu la seva descripció a 3.5.1).

  Noteu que, per tal de saber el nombre d'elements que conté l'arbre amb arrel `root`, usem l'operació privada:

  ```
  1   private int size(Entry<T> root)
  ```

  que obté el nombre de descendents de `root` (o sigui, el nombre d'elements que tindrà l'arbre arrelat a `root`).

  Per cert, per què creieu que hem fet privades totes dues operacions?

`contains(o)`

```
1   @Override
2   public boolean contains(Object o) {
3       return containsRec(o, root);
4   }
5
6   private boolean containsRec(Object o, Entry<T> root) {
7       if (root == null) return false;
8       return root.element.equals(o)
9               || containsRec(o, root.left)
10              || containsRec(0, root.right));
11  }
```

Comentaris:

- L'operació `contains(o)` no és necessari implementar-la a `LinkedBinaryTree<T>` perquè s'hereta d'`AbstractCollection<T>`. Així i tot, la implementació d'`AbstractCollection<T>`, que no coneix com està veritablement representat l'arbre, recorre l'arbre amb un iterador per tal de trobar un element com `o`. `LinkedBinaryTree<T>` coneix la representació de l'arbre i, per això pot fer una implementació més directa i una mica més eficient de l'operació. Així doncs, la redefinim.

  L'anotació `@Override` indica que `LinkedBinaryTree<T>.contains(o)` redefineix `AbstractCollection<T>.contains(o)`.

- La implementació d'aquesta operació és molt alliçonadora de com implementar operacions de manera recursiva:

  - El codi de `contains(o)` crida una operació privada recursiva:

    ```
    1   private boolean containsRec(Object o, Entry<T> node);
    ```

  - Aquesta operació estableix de manera recursiva si algun descendent del node `node` (incloent-se ell mateix) conté l'element `o`.
  - El codi de `containsRec(o,node)` comprova si `o` és igual a l'element del propi node `node`. Si ho és retorna cert, si no, estudia recursivament si `o` es troba en un descendent del node fill esquerre de `node` o en un descendent del node fill dret de `node`.

    El cas trivial de la recursivitat s'assoleix quan `containsRec(o,node)` es crida amb `node = null` o bÃ© qual s'ha trobat l'element.

  Aquest esquema s'usa en altres operacions de la classe `LinkedBinaryTree<T>`. En particular, `equals(o)` i `height()`, el codi de les quals mostrem tot seguit.

`equals(o)`

```
1   @Override
2   public boolean equals(Object o) {
3       if (o == null || !(o instanceof LinkedBinaryTree<T>))
4           return false;
5       LinkedBinaryTree<T> bt = (LinkedBinaryTree<T>) o;
6       return equalsRec(bt.root,root);
7   }
8
9   private boolean equalsRec(Entry<T> root1, Entry<T> root2) {
10      if (root1 == null || root2 == null) return root1 == root2;
11
12      return root1.element.equals(root2.element)
13              && equalsRec(root1.left, root2.left)
14              && equalsRec(root1.right, root2.right);
15  }
```

`height()`

```
1 public int height() {
2     return heightRec(root);
3 }
4
5 private int heightRec(Entry<T> root) {
6     if (root == null) return 0;
7     return Math.max(heightRec(root.left),
8                     heightRec(root.right)) + 1;
9 }
```

`getLeftCh(),getRightCh(),getRoot()`

```
1   public BinaryTree<T> getLeftCh() {
2       if (root == null) throw new NoSuchElementException();
3
4       return new LinkedBinaryTree<T>(root.left);
5   }
```

Comentaris:

- La temptació a l'hora d'implementar `getLeftCh()` és que aquesta operació retorni directament `root.left`:

```
1 public BinaryTree<T> getLeftCh() {
2     if (root == null) throw new NoSuchElementException();
3
4     return root.left; //MALAMENT!
5 }
```

Però això no és correcte perquè l'operació ha de retornar un `BinaryTree<T>` i no un node (`Entry<T>`). Així doncs, hem de convertir el node `root.left` en `BinaryTree<T>`, la qual cosa fem usant l'operació constructora `private LinkedBinaryTree(Entry<T> root)` descrita a 3.5.1.

`removeLeftCh(),removeRightCh()`

```
1    public void removeLeftCh() {
2        if (root != null) {
3           size   = size(root.right)+1;
4           root.left = null;
5        }
6    }
```

Comentaris:

- L'operació `size(node)` retorna el nombre d'elements descendents del node `node`. L'hem descrita a 3.5.1.

`iterator()`

Ara comencen les operacions d'iteració, les que ens faran ballar una mica més el cap. Comencem, però, suaus.

```
1    public Iterator<T> iterator() {
2        return new PreBinTreeItr();
3    }
```

Comentaris:

- L'operació `iterator()` la incloem perquè forma part de les interfícies `Iterable<T>` i `Collection<T>` que hem triat com a superinterfícies de `BinaryTree<T>`. Aquesta operació ens ha de retornar un iterador sobre l'arbre binari. Però amb quina estratègia d'iteració? En principi, podríem triar-ne qualsevol, però hem de ser consistents amb l'especificació de l'operació que indicava que `iterator()` retorna un iterador sobre els arbres binaris *en preordre*.

- Així doncs, `iterator()` retorna un iterador de classe `PreBinTreeItr`. Aquesta classe modelitza un iterador per a arbres binaris que els recorre en preordre. Segueix la mateixa idea de la secció 2.7, quan vam introduir la classe `ListItr` que permetia iterar sobre llistes. La descriurem a les seccions següents.

`iteratorPre(),iteratorIn(),iteratorPost(),iteratorLevels()`

Aquestes operacions no les hereta `BinaryTree<T>` de les superinterfícies `Collection<T>` o `Iterable<T>` sinó que són específiques de `BinaryTree<T>`. Segueixen la mateixa idea que `iterator()` però retornen cadascuna un iterador amb l'estratègia corresponent. Vegem-ho:

```
1  public BinaryTreeIterator<T> iteratorPre() {
2      return new PreBinTreeItr();
3  }
4
5  public BinaryTreeIterator<T> iteratorIn() {
6      return new InBinTreeItr();
7  }
8  ...
```

Ara, és clar, ens cal definir les classes `PreBinTreeItr`, `InBinTreeItr`, `PostBinTreeItr` i `LevelsBinTreeItr`. Com que totes es defineixen de la mateixa manera, presentarem només la primera.

`PreBinTreeItr`

La classe `PreBinTreeItr` implementa la interfície `BinaryTreeIterator<T>`.

Els objectes que són instància d'aquesta classe permeten recórrer un arbre binari en preordre.

Aquesta classe es definirà de manera similar a `ListItr` a la secció 2.7, del capítol 2:

```
1  public class LinkedBinaryTree<T> extends AbstractCollection<T>
2                                   implements BinaryTree<T> {
3      ...
4      private class PreBinTreeItr implements BinaryTreeIterator<T> {
5          ...
6
7          public boolean hasNext() {
8              ...
9          }
10
11         public T next() {
12             ...
13         }
14
15         public void set(T o) {
16             ...
17         }
18
19         public void remove() {
20             throw new UnsupportedOperationException();
21         }
22     }
23     ...
24 }
```

Notem que, com ja hem explicat a la secció 3.2.1, l'operació `remove()` (que és una operació opcional de la interfície `Iterator<T>`) no la incloem a `BinaryTreeIterator<T>`.

Fins aquí tot ha estat fàcil, però ara comencen les dificultats: com ens ho podem fer per implementar les operacions `next()`, `hasNext()`, `set(o)`?

Tornem a l'arbre de la figura 3.2: si el darrer element servit per l'iterador ha estat `x4`, una crida a `next()` ha de retornar `x3`. Però per viatjar fins a `x3` hem de desplaçar l'iterador dues vegades cap al node pare (`x2, x1`) i una vegada cap al node fill dret (`x3`). Des d'un node podem baixar cap al seu fill dret però no tenim cap atribut que ens permeti anar al node pare. L'enriquiment de la classe node (`Entry<T>`) amb una referència al pare facilita la implementació de classes iteradors i és el tema de la secció 3.5.2.

Però ara un node no conté aquesta referència al pare. Què fem? Penseu-ho un moment.

Us plantejo una estratègia basada en el resultat de la secció 3.2.2, on hem obtingut una llista amb els elements que constitueixen el recorregut en preordre d'un arbre:

> A l'operació constructora de la classe `PreBinTreeItr` podem usar l'algoritme de la secció 3.2.2 per tal d'obtenir una llista amb els elements de l'arbre binari que volem recórrer ordenats en preordre. Seguidament, obtenim un iterador sobre aquesta llista i implementem les operacions `PreBinTreeItr.hasNext()` i `PreBinTreeItr.next()` en termes de les mateixes operacions però ara sobre l'iterador de la llista.
>
> L'efecte serà una iteració en preordre sobre l'arbre binari.

Vegem com es pot implementar aquesta idea:

Llistat 3.3: Implementació incompleta de `PreBinTreeItr`

```
1  private class PreBinTreeItr implements BinaryTreeIterator<T> {
2      List<T> listTree;
3      ListIterator<T> it;
4
5      public PreBinTreeItr() {
6          listTree = preorder(LinkedBinaryTree.this.root);
7          it = listTree.listIterator();
8      }
9
10     public boolean hasNext() {
11         return it.hasNext();
12     }
13
14     public T next() {
15         return it.next();
16     }
17
18     public void set(T o) {
19         ...
20     }
```

```
21
22     public void remove() {
23         throw new UnsupportedOperationException();
24     }
25
26     private List<T> preorder(Entry<T> root) {
27         List<T> lis = new ArrayList<T>();
28
29         if (root != null) {
30             lis.add(root.element);
31             lis.addAll(preorder(root.left));
32             lis.addAll(preorder(root.right));
33         }
34         return lis;
35     }
36 }
```

Comentaris:

- La clau d'aquesta classe és a l'operació constructora: allí es genera una llista amb els elements de l'arbre ordenats en preordre:

  ```
  1             listTree = preorder(LinkedBinaryTree.this.root);
  ```

  I, aleshores s'obté un iterador sobre aquesta llista:

  ```
  1             it = listTree.listIterator();
  ```

- A partir d'aquí, n'hi ha prou de resseguir aquest iterador `it` cada cop que es vulguin obtenir nous elements de l'arbre en preordre:

  ```
  1         public T next() {
  2             return it.next();
  3         }
  ```

- A la construcció de la línia 6:

  `listTree = preorder(root);`

  `root` es refereix a l'atribut `root` de l'arbre de classe `LinkedBinaryTree` al qual està associat aquest iterador. En cas d'ambigüitat es pot canviar la referència `root` de la manera següent:

  `listTree= preorder(LinkedBinaryTree.this.root);`

  Aquesta construcció `NomClasse.this.nomAtribut` ens va aparèixer també a la secció 2.7.4 del capítol 2. Doneu-hi un cop d'ull per a més detalls.

Ja hem acabat? Mmm... no del tot. Aquesta classe té dues dificultats, les veieu?

1. La implementació de l'operació `PreBinTreeItr.set(o)` planteja una dificultat important: si la implementem reduint-la a `ListIterator<T>.set(o)` canviarem el darrer element de la llista `listTree` retornat per `next()` pel paràmetre `o`, però *no modificarem per a res aquell element dins l'arbre binari!*, tal com és l'objectiu de `PreBinTreeItr.set(o)`.

   La resolució d'aquest problema pot comportar alguns canvis a la representació de la classe `PreBinTreeItr` i a la implementació de les seves operacions, amb relació a la proposta del llistat 3.3.

   Com ho faríeu?

2. La creació d'un iterador de classe `PreBinTreeItr` sobre un arbre implica, en el moment de la seva creació, el recorregut de tot l'arbre. Imaginem que es tracti d'un arbre molt gran i que volguéssim l'iterador simplement per iterar sobre uns quants elements de l'arbre. En aquestes condicions, la solució que hem donat recorrerà sempre i innecessàriament la totalitat de l'arbre incorrent en un sobrecost.

   Si aquest aspecte és rellevant, la solució que hem donat pot no ser acceptable i hem d'explorar noves idees. Una d'elles és la de proveir per a cada node un punter al seu node pare. Continueu llegint?

3.5.2 Arbres binaris enllaçats amb punters al pare

Afegir a la representació d'un node de l'arbre binari un atribut amb una referència al pare d'aquell node té dos avantatges:

- Permet definir iteradors sobre els arbres sense necessitat d'obtenir d'entrada una llista amb els elements ordenats amb el criteri d'iteració.
- Hi ha aplicacions en què els arbres es recorren de baix cap dalt: de les fulles a l'arrel. En aquells casos resulta vital que cada node disposi d'una referència al seu node pare.

Representació de `BinaryTree<T>` amb punters al pare

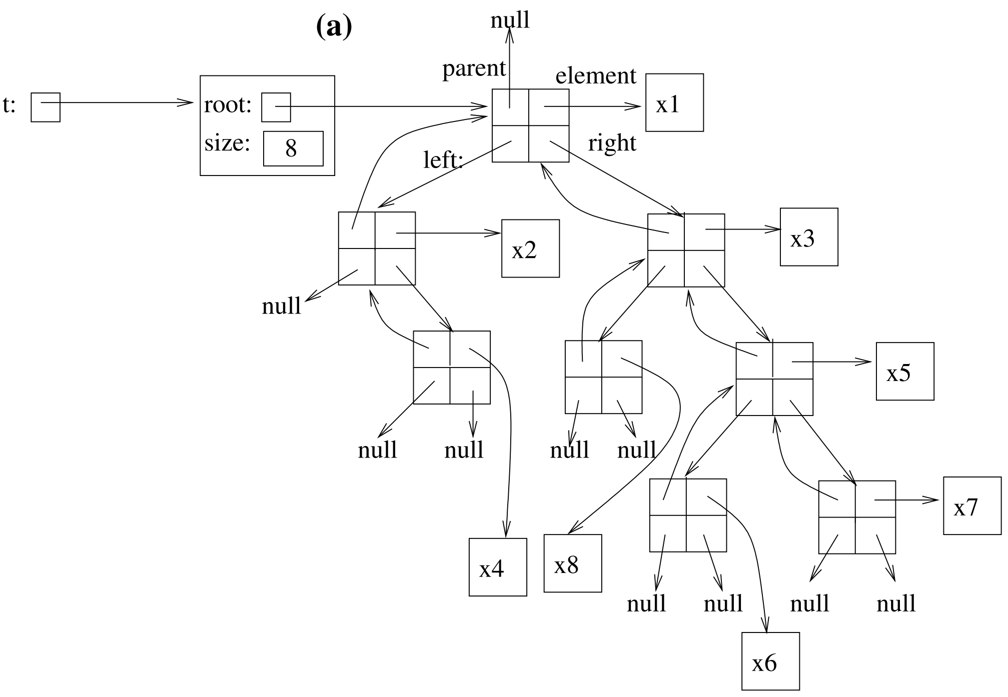
La manera més immediata de representar un arbre binari amb nodes enllaçats i punter al pare és a partir de la representació bàsica de la secció 3.5.1 i afegir a cada node un punter al pare. Això ens portaria a la representació que mostra la figura 3.9. L'apartat (a) mostra la representació d'un arbre binari no buit, l'apartat (b) un de buit i l'apartat (c) els quatre enllaços que té un node.

Aquesta representació funciona bé però és millorable. Pensem-hi un moment:

Clarament, si disposem de punters al pare, canviarem la representació de la classe `PreBinTreeItr`. És raonable pensar que podem representar aquesta classe amb una referència a `next`:

```
1     private class PreBinTreeItr implements BinaryTreeIterator<T> {
2         Entry<T> next;
3         ...
4     }
```

O sigui, l'atribut `next` de l'objecte iterador es referirà al següent element que retornarà l'iterador quan es cridi l'operació `PreBinTreeItr.next()`. Noteu que aquesta idea és



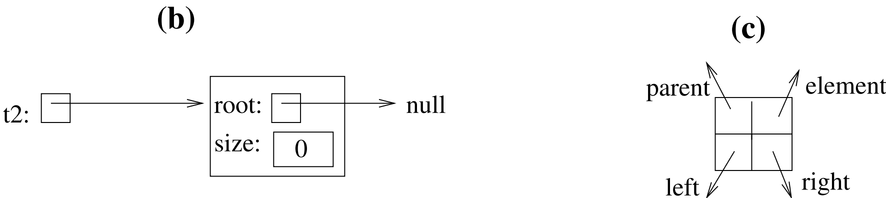


Figura 3.9: Representació **no definitiva** d'arbres binaris amb punter al pare

exactament la mateixa que usàvem a la implementació dels iteradors per `LinkedList<T>` (vegeu 2.7).

I aleshores sorgeixen algunes preguntes: on apuntarà l'atribut `next` després d'haver retornat el darrer element del recorregut en preordre de l'arbre? On apuntarà l'atribut `next` en un arbre buit? Evidentment, podríem fer que apuntés a `null` però això complicaria una mica les operacions de tractament dels iteradors, especialment, `next()` i `hasNext()`: hi afegiria més condicions, més casos especials a tractar separadament.

La solució que proposem és la mateixa que vam prendre en el cas de `LinkedList<T>`: afegirem una `arrel virtual` a l'arbre de manera que la mateixa mecànica de l'operació `next()` faci que l'atribut `next` de l'iterador es refereixi a aquesta arrel virtual després de retornar el darrer element de l'arbre.

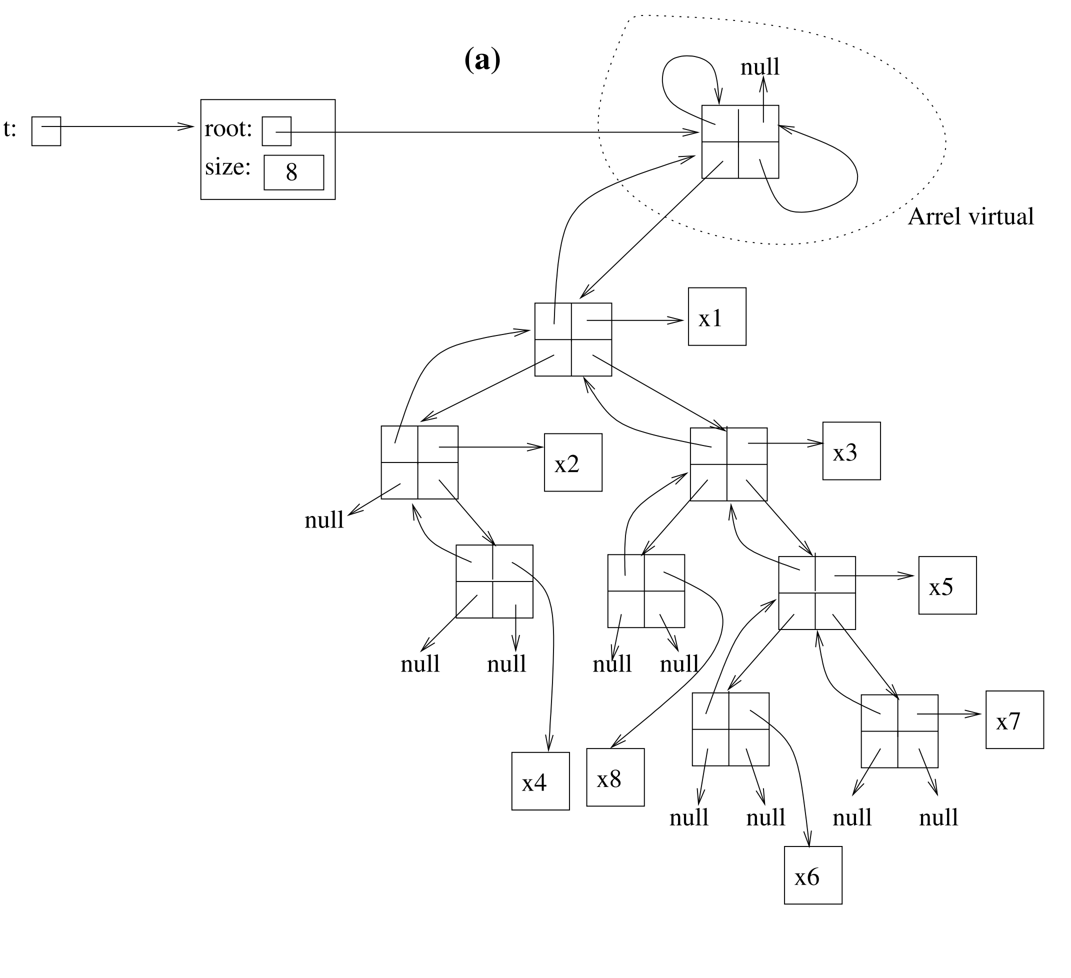
En definitiva, la nova representació dels arbres binaris amb enllaços i punter al pare és la que es presenta a la figura 3.10 i al llistat 3.4. La sintetitzem al quadre següent:

Representació d'un arbre binari amb punter al pare

- Cada node afegirà un atribut `parent` que apuntarà al pare d'aquell node en l'arbre binari.
- L'atribut `root` apuntarà a una *arrel virtual* de manera que el seu fill esquerre (`left`) apuntarà a l'arrel real de l'arbre i el seu fill dret (`right`) apuntarà a la mateixa arrel virtual. El pare (`parent`) apuntarà a la mateixa arrel virtual.
- L'arbre buit estarà representat per l'arrel virtual que tindrà un fill esquerre (`left`) que apuntarà a `null`, i un fill dret (`right`) i pare (`parent`) que apuntaran a la mateixa arrel virtual.

Llistat 3.4: Definició de la classe `LinkedBinaryTreeParent`

```
1  public class LinkedBinaryTreeParent<T>
2                          extends AbstractCollection<T>
3                          implements BinaryTree<T> {
4      private Entry<T> root;
5      int size;
6
7      private static class Entry<T> {
8          T element;
9          Entry<T> left;
10         Entry<T> right;
11         Entry<T> parent;
12         Entry(T element,
13               Entry<T> left, Entry<T> right, Entry<T> parent) {
14             this.element = element;
15             this.left = left;
16             this.right = right;
```



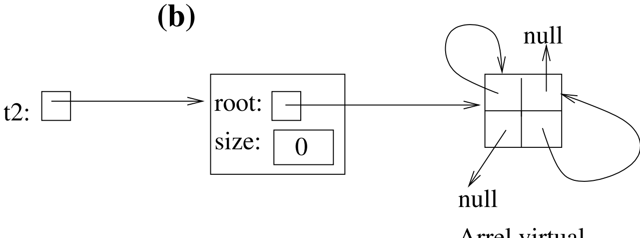


Figura 3.10: Representació **final** dels arbres binaris amb punter al pare

```
17              this.parent = parent;
18          }
19
20      }
21
22      public LinkedBinaryTreeParent() {
23          root = new Entry<T>(null, null, null, null);
24          root.parent = root;
25          root.left = null;
26          root.right = root;
27          size = 0;
28      }
29
30      public LinkedBinaryTreeParent(T elem,
31                                    LinkedBinaryTreeParent<T> left,
32                                    LinkedBinaryTreeParent<T> right) {
33          Entry<T> leftCh = (left == null ? null : left.root.left);
34          Entry<T> rightCh = (right == null ? null : right.root.left);
35
36          int leftS = (left == null ? 0 : left.size);
37          int rightS = (right == null ? 0 : right.size);
38
39          Entry<T> root = new Entry<T>(null, null, null, null);
40
41          root.left = new Entry<T>(elem, leftCh, rightCh, root);
42          root.right = root;
43          root.parent = root;
44
45          if (leftCh != null) leftCh.parent = root;
46          if (rightCh != null) rightCh.parent = root;
47
48          size = leftS + rightS + 1;
49      }
50      ...
51  }
```

Comentaris:

- Noteu que l'arrel real de l'arbre queda definida com el fill esquerre de l'arrel virtual.

- Noteu també que el fill dret de l'arrel virtual es refereix a la mateixa arrel virtual. El mateix que fa el pare.

- L'arbre buit consta d'una arrel virtual amb un fill esquerre que apunta a `null` i un fill dret i pare que s'apunten a ella mateixa.

Us deixo com a exercici estudiar com afecta aquesta nova representació (amb el punter al pare i l'arrel virtual) a la resta de les operacions de la classe.

La classe `PreBinTreeItr`

La classe que implementa els iteradors en preordre per a un arbre binari representat amb punters al pare té el codi següent:

Llistat 3.5: Implementació de la classe `PreBinTreeItr` en un arbre binari amb enllaços i punter al pare

```
1  private class PreBinTreeItr implements BinaryTreeIterator<T> {
2      Entry<T> lastReturned;
3      Entry<T> next;
4
5      public PreBinTreeItr() {
6          lastReturned = root;
7          if (size == 0) next = root;
8          else next = root.left;
9      }
10
11     public boolean hasNext() {
12         return next != root;
13     }
14
15     public T next() {
16         T previous;
17         if (next == root) throw new NoSuchElementException();
18         lastReturned = next;
19         if (next.left != null) next = next.left;
20         else if (next.right != null) next = next.right;
21         else {
22             do{
23                 previous = next;
24                 next = next.parent;
25             } while (next.right == null || next.right == previous);
26             next = next.right;
27         }
28         return lastReturned.element;
29     }
30
31     public void set(T o) {
32         if (lastReturned == root) throw new IllegalStateException();
33         lastReturned.element = (T) o;
34     }
35 }
```

Comentaris:

- L'operació `hasNext()`, de forma consistent al que hem explicat abans, retornarà cert si `next != root`. O sigui, haurem arribat al final del recorregut quan `next` apunti a l'arrel virtual.
- L'operació `next()` comença recordant a la variable `lastReturned` el node que ha de retornar l'operació `next()`. El node `next` és el que s'ha de retornar quan es crida l'operació `next()` i és aquest el que posem a `lastReturned`.

Noteu també que `lastReturned == root` és el criteri que s'usarà per determinar si l'operació `set(...)` es crida en un estat il·legal.

L'operació `next()` cercarà ara el següent en preordre de `lastReturned`.

- Per trobar el següent en preordre de `lastReturned`, l'operació `next()` ha de considerar diferents possibilitats:

  - Clarament, el fill esquerre d'un node, si existeix, és el següent en preordre d'aquell node.
  - Si no existeix el fill esquerre, aleshores ho és el dret (si existeix, és clar).
  - Si no existeixen ni fill esquerre ni dret, aleshores per trobar el següent en preordre del node `lastReturned` iniciarem un camí ascendent a l'arbre a partir de `lastReturned` (si és necessari, fins a arribar a l'arrel virtual). En concret:

> El següent en preordre del node `lastReturned` és *el node fill dret del primer ancestre de* `lastReturned` *que:*
> 1. *Tingui fill dret i que*
> 2. *compleixi que* `lastReturned` *és al seu subarbre fill esquerre.*

  Per establir la segona condició, ens convindrà recordar en tot moment el darrer node recorregut en el camí ascendent des de `lastReturned`. Efectivament, si anomenem `next` el node actual en aquest camí ascendent, `lastReturned` és al subarbre fill esquerre de `next` si i només si el node anterior en aquest camí ascendent era el fill esquerre de `next`. Per tal de recordar el node anterior en el camí ascendent des de `lastReturned` usarem `previous`.

  - A la sortida del bucle *do-while* (vegeu la línia 25) serà el primer cop des que hem començat el recorregut ascendent en què es complirà:

```
next.right != null && next.right != previous
```

  Com que `next.right != null`, `next` tindrà fill dret. Com que, a més a més, `next.right != previous`, es complirà que pugem des del fill esquerre de `next` i, per tant, `lastReturned` és al subarbre fill esquerre de `next`. I, justament, aquestes són les dues condicions que assenyala el quadre anterior.

  - Un cop `next` es refereix al primer node que compleix les condicions 1 i 2 del quadre anterior, per obtenir el següent en preordre a `lastReturned` només cal avançar fins al fill dret de `next`, que és el que fa la instrucció de la línia 26.

- Noteu que, sense haver de considerar cap cas particular, quan el darrer element retornat per l'operació `next()` és el darrer element en preordre i cridem novament `next()`, l'atribut `next` passa a apuntar a l'arrel virtual.

  Aquesta propietat ocorre també en el recorregut en inordre. Malauradament no ocorre en el recorregut en postordre, però no és greu ja que la identificació del darrer element del recorregut en postordre és trivial: sempre és l'arrel de l'arbre.

- Una altra propietat de l'arrel virtual en els recorreguts en preordre i inordre és que si l'operació `next()` no llencés l'excepció `NoSuchElementException` (ho fa perquè aquesta operació està especificada d'aquesta manera) l'atribut `next` es desplaçaria al primer element del recorregut en preordre o inordre i la iteració tornaria a començar. Es podrien fer, doncs, iteracions circulars.

Un problema interessant és implementar les classes `InBinTreeItr` i `PostBinTreeItr` en el cas d'arbres implementats amb enllaços i punter al pare.

3.5.3 Amb enfilaments en inordre (*)

El recorregut en inordre és d'un interès especial perquè, com veurem al capítol 4 (en la secció que s'ocupa dels arbres binaris de cerca), un recorregut en inordre d'un arbre binari de cerca permet obtenir els elements de l'arbre ordenats per clau. El significat exacte d'això ara és poc important. Quedem-nos simplement amb la idea de la importància del recorregut en inordre d'un arbre.

D'altra banda, a la representació amb punter al pare, necessitem mantenir un punter al pare a cada node i, al mateix temps, desaprofitem els punter a fill esquerre i fill dret de les fulles (que apunten a `null`).

Aquesta idea ens porta a una representació alternativa dels arbres binaris que permetran implementar iteradors fàcilment (particularment, en inordre) sense haver de mantenir una referència al pare i aprofitant els punter a fill esquerre i dret de les fulles. Aquesta representació és la següent:

Representació d'un arbre binari amb enfilaments en inordre

- El punter a fill esquerre (`left`) d'un node `nod` que no tingui fill esquerre apuntarà al node anterior a `nod` en inordre.

  Ens referirem a aquest punter com a **enfilament esquerre** per distingir-lo d'un fill esquerre autèntic.

- El punter a fill dret (`right`) d'un node `nod` que no tingui fill dret apuntarà al node següent a `nod` en inordre.

  Ens referirem a aquest punter com a **enfilament dret** per distingir-lo d'un fill dret autèntic.

- L'atribut `root` de l'arbre apuntarà a una *arrel virtual* de manera que el seu fill esquerre (`left`) apuntarà a l'arrel real de l'arbre i el seu fill dret (`right`) apuntarà a la mateixa arrel virtual.

- L'arbre buit estarà representat per l'arrel virtual que tindrà un *enfilament esquerre* (`left`) que apuntarà a la mateixa arrel virtual i un *fill dret* (`right`) que també apuntarà a la mateixa arrel virtual.

Noteu que incorporem, com en el cas dels arbres binaris amb punter al pare, una arrel virtual. La raó és la mateixa: simplificar les operacions d'iteració i tenir un node on apuntarà

l'atribut `next` de l'objecte iterador un cop hagi retornat el darrer element del recorregut o en el cas d'un arbre buit.

La figura 3.11 mostra gràficament la representació dels arbres binaris amb enfilaments en inordre. Noteu que hem situat els elements dins dels nodes per simplificar la figura. En realitat, els elements estan referenciats per l'atribut `element` del node, tal com hem fet a les anteriors figures. Noteu també que els punters que són enfilaments estan dibuixats amb traç discontinu mentre que els que són fills estan dibuixats amb traç continu.

La classe `LinkedBinaryTreeThread<T>`

Llistat 3.6: Definició de la classe `LinkedBinaryTreeThread<T>`

```
1  public class LinkedBinaryTreeThread<T>
2                          extends AbstractCollection<T>
3                          implements BinaryTree<T> {
4      private Entry<T> root;
5      int size;
6
7      private static class Entry<T> {
8          T element;
9          Entry<T> left;
10         Entry<T> right;
11         boolean isThreadLeft;
12         boolean isThreadRight;
13
14         Entry(T element, Entry<T> left, Entry<T> right,
15               boolean isThreadLeft, boolean isThreadRight) {
16             this.element = element;
17             this.left = left;
18             this.right = right;
19             this.isThreadLeft = isThreadLeft;
20             this.isThreadRight = isThreadRight;
21         }
22     }
23     ...
24 }
```

Comentaris:

- Hem afegit dos atributs booleans a la classe `Entry<T>` (`isThreadLeft`, `isThreadRight`) per distingir si un node té fills o enfilaments als punters `left` i `right`.

Us queda com a exercici pensar com afecten els enfilaments a la resta d'operacions de la classe , particularment a les operacions constructores.

La classe `InBinTreeItr`

La classe que implementa els iteradors en inordre per a un arbre binari representat amb enfilaments en inordre té el codi següent:

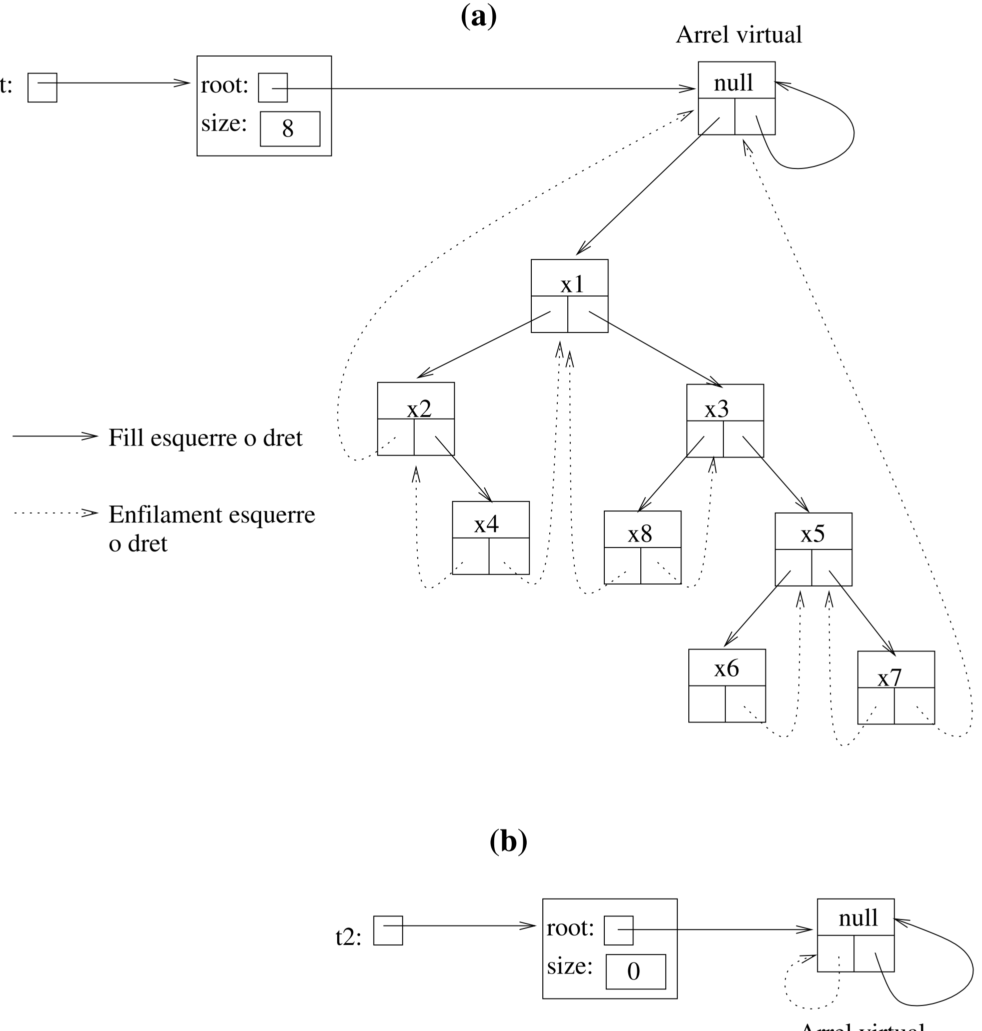


Figura 3.11: Representació dels arbres binaris amb enfilaments en inordre

Llistat 3.7: Implementació de la classe `InBinTreeItr` en un arbre binari amb enfilaments

```
1  private class InBinTreeItr implements BinaryTreeIterator<T> {
2      Entry<T> lastReturned;
3      Entry<T> next;
4  
5      public InBinTreeItr() {
6          lastReturned = root;
7          next = root;
8          next();
9      }
10 
11     public boolean hasNext() {
12         return next != root;
13     }
14 
15     public T next() {
16         if (next == root) throw new NoSuchElementException();
17 
18         lastReturned = next;
19 
20         boolean thread = next.isThreadRight;
21 
22         next = next.right;
23 
24         if (!thread)
25             while(!next.isThreadLeft)
26                 next = next.left;
27 
28         return lastReturned.element;
29     }
30 
31     public void set(T o) {
32         if (lastReturned == root)
33             throw new IllegalStateException();
34         lastReturned.element = (T) o;
35     }
36 }
```

Comentaris:

- Noteu a l'operació `next()` que si el node del qual cal trobar el següent té un enfilament dret, el node apuntat per aquell enfilament serà directament el següent en inordre. Si, per contra, té un fill dret, aleshores el següent en inordre serà el terminal esquerre d'aquell fill dret. I això és el que calcula `next()`.
- Noteu també que en aplicar `next()` al darrer element en inordre, l'atribut `next` es mou fins a l'arrel virtual.
- I noteu, finalment, que si `next` es refereix a l'arrel virtual, i aleshores cridem l'operació `next()`, l'atribut `next` es mou fins al primer en inordre de l'arbre (que correspon sempre amb el terminal esquerre).

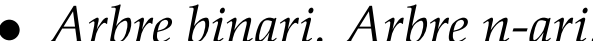
  Per aquest motiu implementem l'operació constructora:

```
1       public InBinTreeItr() {
2           lastReturned = root;
3           next = root;
4           next();
5       }
```

Per cert, els enfilaments en inordre també permeten implementar iteradors en preordre i en postordre. Penseu com serien les operacions `hasNext()` i `next()` en tots dos casos.

3.6 Racó lingüístic

En aquesta secció comentem els termes tècnics usats en aquest capítol que no estan estandarditzats al diccionari de l'Institut d'Estudis Catalans, a Termcat o a [CM94].



- *Arbre binari. Arbre n-ari.*

  La definició d'*arbre binari* i *arbre n-ari* pot variar en diferents fonts. En particular, [CM94] distingeix entre *arbre binari* i arbre *2-ari* (o, conseqüentment, entre arbre ternari i arbre 3-ari) depenent de si importa o no la posició que cada fill ocupa a l'arbre. En la majoria de literatura que hem consultat aquesta distinció no es fa. Aquest volum, tampoc no la fa.

  Termcat no defineix arbres n-aris.

- *Arbre equilibrat.*

  Traducció que proposem al terme anglès *balanced tree*. Preferim *equilibrat* a *balancejat* perquè el significat de *balancejat* és que *ha estat mogut oscil·lant a costat i costat de la situació d'equilibri*. Mentre que *equilibrat* significa *posat en equilibri*, que és precisament el cas d'un *balanced tree*. Entenem que en castellà, el terme *árbol balanceado*, que ha esdevingut l'estàndard, sí que és correcte perquè en aquesta llengua, *balancear* té l'accepció de posar en equilibri.

  Ni Termcat ni [CM94] defineixen cap terme estàndard per *balanced tree*.

- *Recorregut en amplada. Recorregut en fondària o profunditat.*

  Traduccions naturals per als termes anglesos *breadth first traversal* i *depth first traversal*, respectivament.

  Ni Termcat ni [CM94] defineixen cap terme estàndard per ambdós tipus de recorregut.

- *Recorregut en preordre, inordre i postordre.*

  Traduccions naturals per als termes anglesos *preorder, inorder, postorder traversal*, respectivament.

  Ni Termcat ni [CM94] defineixen cap terme estàndard per aquests tipus de recorregut.

- *Arbre enfilat*

  Traducció que proposem per al terme anglès *threaded tree*.

  Ni Termcat ni [CM94] defineixen cap terme estàndard per aquest tipus d'arbre. Així i tot, *arbre enfilat* s'ha usat freqüentment.

Capítol 4

Estructures de dades d'accés directe: les taules

Heu notat que la forma d'accés a la informació emmagatzemada en les estructures de dades que hem estudiat fins ara (llistes i arbres) és una mica limitada? Bàsicament, podem recórrer una llista o un arbre (i.e., accedir al següent d'un element donat) i, en el cas de les llistes, podem, a més a més, accedir-hi per posició (accedir al quart element de la llista, per exemple). Però en cap cas podem accedir-hi directament per contingut (per exemple, accedir ràpidament a l'element de la llista que té per nom Joan).

Ha arribat el moment d'ocupar-nos d'aquest tipus d'accés. Anomenarem *taules* (en anglès, *maps*) les estructures de dades que ens permeten accedir eficientment per contingut a un element emmagatzemat a l'estructura. L'ús de les taules ens obrirà horitzons molt interessants. En particular, les taules permeten accedir eficientment als registres d'una base de dades (a través dels anomenats índexs, que estudiarem al capítol 7). D'altra banda, per implementar taules necessitarem les tècniques de llistes i arbres que hem vist als capítols anteriors.

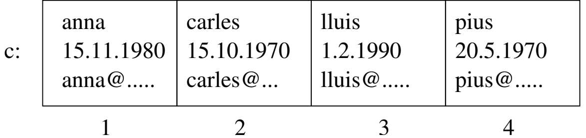
A l'estudi de les taules dedicarem els propers 4 capítols: en aquest en donem el concepte, les especifiquem i en presentem algunes implementacions senzilles. Als dos propers (5 i 6) presentem les dues famílies d'implementacions de taules més eficients (*dispersió* i *arborescent*). Finalment, al capítol 7, explorem l'ús de les taules com a índexs d'una taula d'una base de dades.

Com veieu, hem arribat al moment culminant del curs.

4.1 Què és una taula?

Suposem que tenim una col·lecció d'elements (per exemple, la de la figura 4.1). Com podem accedir a aquests elements?

1. **Accés seqüencial:** obtenir el *següent* element (e.g., el següent d'anna és carles).



	1	2	3	4
c:	anna 15.11.1980 anna@.....	carles 15.10.1970 carles@...	lluis 1.2.1990 lluis@.....	pius 20.5.1970 pius@.....

Figura 4.1: Tipus d'accés a una col·lecció

2. **Accés per posició:** obtenir l'element que ocupa la posició p al contenidor (e.g. l'element que ocupa la posició 3 és lluis).

3. **Accés per contingut:** obtenir un element a partir d'un fragment del seu contingut (e.g., obtenim les dades de l'element anomenat *pius*. Aquestes dades són: data de naixement: 20.5.1970 i e-mail: pius@...).

En els capítols anteriors (singularment, al capítol 2) hem tractat els accessos seqüencials i per posició. Aquest capítol s'ocupa del tercer accés: *l'accés per contingut*. Concretament, el fragment del contingut d'un element que usarem per accedir a aquest element s'anomena *clau*.

Clau:
Fragment del contingut d'un element que l'identifica (de manera única) entre la resta dels elements d'un contenidor (i.e., no hi ha dos elements al contenidor amb la mateixa clau).

Exemples de claus: nif, matrícula, número de compte d'estalvi, telèfon...

Taula:
Estructura de dades que ens permet un accés eficient als seus elements a partir de la clau.

Moltes vegades, la clau serà una cadena de caràcters (*nif*, *matrícula*...), però aquest no serà un requeriment.

4.1.1 Com podem modelitzar una taula?

Intuïtivament, una taula es pot modelitzar com *una col·lecció de parelles* [clau, valor]:

```
m={[ "12345678A",<12345678A, pepet, 15.11.1970, pepet@meumail.com> ],
   [ "18888886S",<18888886S, anna,15.10.1980,anna@meumail.com> ],
   [ "33333888X",<33333888X, mari,1.1.1970,mari@meumail.com> ]...
  }
```

De manera que es pot accedir al valor eficientment a través de la clau.
En particular, l'accés a un valor de la taula a través de la seva clau s'ha d'aconseguir amb un cost $O(1)$ o $O(\log n)$ on n és el nombre d'elements emmagatzemats a la taula.

A vegades, ens interessarà pensar en una taula des d'un punt de vista més formal:

Algebraicament, podem veure una taula com una funció injectiva entre un conjunt de claus i un altre de valors:
$m : K \longrightarrow V$
$m(k) = v$ (v és el valor associat a la clau k).

- K: conjunt de claus

  (també anomenades *índexs, identificadors*).

- V: conjunt de valors

  (també anomenats *informació, contingut*).

4.1.2 Consideracions addicionals sobre les taules:

- Existeixen variants de les taules en què la clau no és identificadora: **multitaula** (en anglès, *multimap*). Poden contenir diversos elements amb la mateixa clau.

  Les multitaules no són estudiades en aquest curs.

- Per tal d'identificar un element d'una taula es podria necessitar més d'un atribut d'un element (e.g., un habitatge en una ciutat s'identifica amb la unió dels atributs `carrer, número, pis, porta`).

4.2 Especificant les taules

La JCF defineix el tipus *taula* mitjançant una sèrie d'interfícies. En aquest curs presentarem dues d'aquestes interfícies: Map<K, V> i SortedMap<K, V>. [NW06] dóna més informació sobre altres interfícies definides per la JCF que ofereixen maneres diferents de treballar amb taules.

4.2.1 La interfície `Map<K, V>`

La interfície `Map<K, V>` modelitza una taula com una col·lecció de parelles *(clau, valor)* de manera que la clau és de tipus `K` i el valor, de tipus `V`.

Un objecte de tipus `Map<K, V>` no pot contenir dues parelles *(clau, valor)* diferents amb la mateixa clau.

La interfície `Map<K, V>` està definida de la manera següent per l'*OpenJDK* (algunes operacions s'ometen):

Llistat 4.1: La interfície `Map<K, V>`

```
1  public interface Map<K, V> {
2      int size();
3      boolean isEmpty();
4      boolean containsKey(Object key);
5      V get(Object key);
6      V put(K key, V value);
7      V remove(Object key);
8
9      Set<K> keySet();
10     Collection<V> values();
11     Set<Map.Entry<K, V>> entrySet();
12
13     interface Entry<K, V> {
14         K getKey();
15         V getValue();
16         V setValue(V value);
17         boolean equals(Object o);
18         int hashCode();
19     }
20     ...
21 }
```

En les properes seccions s'explicarà aquesta interfície.

Especificació de la interfície `Map<K, V>`

Vegem, seguidament, les operacions més significatives que ofereix la interfície `Map<K, V>` agrupades per tipus d'operació:

- *Operacions consultores*

`boolean containsKey(Object key)` Retorna cert si la taula conté una aplicació per a la clau especificada.
`V get(Object key)` Retorna el valor al qual s'aplica la clau especificada, o nul si aquesta taula no conté cap aplicació per a la clau.
`boolean isEmpty()` Retorna cert si aquesta taula no conté cap aplicació clau-valor. `int size()` Retorna el nombre d'aplicacions clau-valor en aquesta taula.

Comentaris:

- L'operació estrella de les taules és `get(key)`. Aquesta operació ens aporta l'accés directe per clau.
- Per tal que aquest accés faci honor al seu nom (directe), les operacions `get(key)` i `containsKey(key)` han de ser eficients. En particular, això vol dir que hauran de tenir un cost de O(1) o O(log n), on n és el nombre de parelles *(clau, valor)* emmagatzemades a la taula.

• *Operacions modificadores*

`V put(K key, V value)` Associa el valor especificat amb la clau especificada en aquesta taula (operació opcional).
`V remove(Object key)` Elimina l'aplicació per a una clau d'aquesta taula si hi és present (operació opcional).

Comentaris:

- Val la pena donar un cop d'ull a l'especificació completa de les operacions `put` i `remove` perquè a l'especificació resumida que es dóna a la taula anterior no queda clar el comportament detallat de l'operació si, per exemple, la clau que es vol inserir ja forma part de la taula. En aquest cas se substituiria el valor vell pel nou, però vegem tal com ho estableix la mateixa especificació de la JCF:

`V put(K key, V value)` Associa el valor especificat amb la clau especificada en aquesta taula (operació opcional). Si la taula prèviament contenia una aplicació per a la clau, el valor anterior és reemplaçat pel valor especificat.
Paràmetres: key - clau amb la qual el valor especificat s'ha d'associar. value - valor que s'ha d'associar amb la clau especificada.
Retorna: el valor previ associat a la clau, o nul si no hi hagués cap aplicació per a la clau.
Llança: UnsupportedOperationException - si l'operació `put` no és compatible amb aquesta taula ClassCastException - si la classe de la clau o valor especificats evita que sigui emmagatzemat en aquesta taula NullPointerException - si la clau o valor especificats és nul i aquesta taula no permet claus ni valors nuls IllegalArgumentException - si alguna propietat de la clau o valor especificats evita que sigui emmagatzemat en aquesta taula

Sisplau, doneu també un cop d'ull a l'especificació completa de `remove(key)`.

– Noteu que, tant l'operació `put(k,v)` com `remove(k)`, retornen el valor associat a la clau `k` abans de la crida a l'operació `put` o `remove`. Si la clau `k` no era a la taula abans de la crida a l'operació retorna `null`.

– Les operacions `remove(key)` i `put(key,value)` no han de ser necessàriament tan eficients com les consultores. Així i tot, intentarem que ho siguin. De fet, a les implementacions que oferirem de les taules, aquestes operacions tindran una eficiència comparable a la de les consultores.

• *Operacions per iterar sobre els elements d'una taula*

La interfície `Map<K, V>` no és subinterfície d'`Iterable`. Per tant, **no podem definir directament iteradors sobre** `Map`**s**. Això no vol dir que no es pugui iterar sobre les claus emmagatzemades en un `Map`, sobre els valors o, fins i tot, sobre les parelles *(clau, valor)*. `Map<K, V>` ofereix tres operacions que ens retornen col·leccions que contenen els elements d'un objecte `Map<K, V>`. I, és clar, les col·leccions sí que són iterables! Vegem aquestes operacions:

`Set<K> keySet()` Retorna una vista de conjunt de les claus contingudes en aquesta taula.
`Collection<V> values()` Retorna una vista de conjunt dels valors continguts en aquesta taula.
`Set<Map.Entry<K, V» entrySet()` Retorna una vista de conjunt de les aplicacions contingudes en aquesta taula.

Comentaris:

– Les operacions `keySet()` i `values()` ens permeten iterar sobre les claus i els valors d'un `Map` respectivament.

Ens retornen una col·lecció (en particular, un `Set`) de claus i de valors respectivament. I com que les col·leccions es poden iterar podem fer coses de l'estil:

```
1  <K, V> K getFirstKey(Map<K, V> m) {
2      Iterator<K> it = m.keySet().iterator();
3
4      if (it.hasNext()) return it.next();
5      else return null;
6  }
```

L'ordre en què es retornen les claus o els valors *no està definit*, en general. Així i tot, veurem que algunes implementacions de `Map<K, V>`, com ara `TreeMap<K, V>` sí que garanteixen una determinada ordenació dels seus elements. En particular, es garanteix que els elements es retornen per ordre ascendent de clau.

– `Map.Entry<K, V>` és una interfície continguda (*aniuada*) dins de `Map<K, V>`. Aquesta interfície aniuada modelitza una parella *(clau, valor)*. `Map.Entry<K, V>` ofereix, entre d'altres, operacions per obtenir la clau i el valor de la parella *(clau, valor)* que modelitza. També ofereix una operació per tal de modificar el valor (segon component) d'una parella. Vegem-ho:

K getKey() Retorna la clau corresponent a aquesta entrada.
V getValue() Retorna el valor corresponent a aquesta entrada.
int hashCode() Retorna el valor de codi hash d'aquesta entrada de la taula.
V setValue(V value) Substitueix el valor corresponent a aquesta entrada amb el valor especificat (operació opcional).

Les classes que implementen `Map<K, V>` defineixen com a classe interna una classe que, a la seva vegada, implementa `Map.Entry<K, V>`:

```
1  public class HashMap<K, V> ... implements Map<K, V> ... {
2      ...
3      static class Entry<K, V> implements Map.Entry<K, V> {
4          final K key;
5          V value;
6          ...
7      }
8      ...
9  }
```

`Map.Entry<K, V>` s'utilitza per tal d'iterar sobre les parelles *(clau, valor)* d'un `Map<K, V>`. Efectivament, si doneu un cop d'ull a la capçalera de l'operació `entrySet()` (de `Map<K, V>`):

```
1  Set<Map.Entry<K, V>> entrySet()
```

veureu que aquesta operació retorna una col·lecció (concretament, un `Set`) de `Map.Entry<K, V>`. Dit d'una altra manera:

> `entrySet()` retorna el conjunt de parelles *(clau, valor)* que constitueixen l'objecte `Map` sobre el qual es crida.

Com que és una col·lecció (i, per tant, subtipus d'`Iterable`) podem iterar sobre ella:

```
1  <K, V> void traversalMap(Map<K, V> m) {
2      Map.Entry<K, V> set = m.entrySet();
3
4      for (Map.Entry<K, V> entry: set){
5          System.out.println(entry.getKey(), entry.getValue());
6      }
7  }
```

- Les operacions `keySet()` i `entrySet()` retornen un `Set`, o sigui, un *conjunt*. Un conjunt es caracteritza perquè no té elements repetits. Això és consistent amb el fet que no hi poden haver repeticions al conjunt de claus d'un `Map` ni, per extensió, al conjunt de parelles *(clau, valor)* allotjades en un `Map`.

Per contra, l'operació `values()` retorna una `Collection<V>`. Un `Map` pot contenir valors repetits i per això no fóra adequat retornar un `Set<V>` en aquest cas.

L'operació `equals` per comparar claus

La classe que actuï com a clau d'un `Map` haurà de tenir una definició adequada de l'operació `equals(Object obj)`. Aquesta operació s'usa sovint en el context de les taules. Considerem aquests exemples: volem obtenir l'objecte del `Map` que té una determinada clau (`get(key)`); volem eliminar l'objecte del `Map` que té una determinada clau (`remove(key)`); volem saber si una determinada clau està emmagatzemada al `Map` (`containsKey(key)`)... En tots tres casos, ens cal comparar una determinada clau que passem pel paràmetre (`key`) amb les claus existents al `Map`. Per aquest motiu, sovint, l'operació `equals` de la classe `K` estarà redefinida.

Recordem que, per defecte, una classe hereta l'operació `Object.equals(...)` que considera dues referències iguals únicament si es refereixen al mateix objecte. Vegem-ne un exemple:

Exemple 4.1:

Suposem que la classe `Person` no redefineixi l'operació `equals(Object obj)`. Considerem el codi següent:

```
1  Person p1 = new Person("12345678", "Joe");
2  Person p2 = new Person("12345678", "Joe");
3  Person p3 = p1;
4
5  boolean b = p1.equals(p3);      //b==true
6                                  //p1 i p3 son referencies al
7                                  //mateix objecte.
8
9  boolean b2 = p1.equals(p2);     //b2==false.
10                                 //p1 i p2 tenen el mateix valor pero
11                                 //no son el mateix objecte.
```

■ ■ ■

Exemple 4.2: Una taula de pisos

Suposem que volem crear una taula de pisos. Per identificar el pis usarem la seva adreça. Tindrem, per tant, la classe `Address` que usarem com a clau per a la taula:

```
1  public class Address {
2      private String street;
3      private int number;
4      private int floor;
5      private String city;
6      ...
7  }
```

Considerarem també una classe `Flat` que contindrà tota la informació que volem emmagatzemar dels pisos:

```
1 public class Flat {
2     private Address adr;
3     private int m2;
4     private int nrooms;
5     private int ntoilets;
6     ...
7 }
```

I una taula:

```
1 Map<Address, Flat> m = new HashMap<Address, Flat>();
2             //HashMap es una implementacio especifica
3             //de taula.
```

Si no redefinim l'operació `Address.equals(...)` la taula no funcionarà com esperem:

Llistat 4.2: Problemes per a la no-redefinició d'`equals`

```
1 Map<Address, Flat> m = new HashMap<Address, Flat>();
2
3 Address a1 = new Address("Carrer␣Pi", 2, 3, "Lleida");
4 Address a2 = new Address("Carrer␣Pi", 2, 3, "Lleida");
5 Flat f;
6
7 m.put(a1, new Flat(....));
8 f = m.get(a2);
9 ...
```

Segurament, amb aquest codi esperaríem que, després de l'execució de la línia 8, `f` valgués l'objecte `Flat` que hem introduït a la taula a la línia 7. Però no serà així. Contràriament, `f == null`. Per què?

Per tal de que la taula funcioni correctament haurem de redefinir dos mÃ¨todes heretats de la classe `Object`: el ja conegut `equals` i un, que presentarem al capÃtol 5, anomenat `hashCode`.

■ ■ ■

Idealment no s'haurien d'usar objectes mutables (això és, objectes el valor dels quals pot ser modificat) com a claus de `Map`s. Si es modifica un objecte `k` que actua com a clau en una parella `(k, v)` inserida a un `Map m` i després es crida:
`obj = m.get(k)`
el comportament d'aquesta operació no està definit. En particular, és possible que el valor `v` associat a `k` no pugui ser recuperat per l'operació.
Si es permeten objectes mutables com a claus, no s'hauria de modificar la clau de cap objecte inserit al `Map` de tal manera que s'alteri el resultat de l'operació `equals` sobre aquella clau.

4.2.2 La interfície `SortedMap<K, V>`

La interfície `Map<K, V>` que acabem de descriure tenia la particularitat que no estava definit l'ordre en què una iteració sobre els elements del *Map* retornava aquests elements.

Recordem que Map<K, V> defineix operacions per iterar sobre les claus (keySet()), els valors (values()) o, directament, sobre les parelles *(clau, valor)* (entrySet()).

La interfície SortedMap<K, V> garanteix que les iteracions realitzades sobre les col·leccions keySet(), values() i entrySet() retornaran els elements en ordre ascendent de clau.

Les claus que formin part d'un SortedMap<K, V> hauran de ser *comparables*. Més encara, hauran de constituir un *ordre total* (o sigui, qualsevol parella de claus hauran de ser comparables). Hi ha dues maneres d'aconseguir-ho:

- El tipus de les claus (i.e., K) implementa la interfície Comparable. En aquest cas, dos claus qualssevol k1, k2 es podran comparar mitjançant l'operació compareTo:

  ```
  k1.compareTo(k2);
  ```

- El SortedMap<K, V> té definit un comparador (i.e., un objecte que implementa la interfície Comparator). Aquest comparador, típicament es passa com a paràmetre del constructor de la classe que implementa SortedMap<K, V>. Cada cop que es volen comparar dues claus k1 i k2 es crida:

  ```
  comparator.compare(k1,k2)
  ```

Tant si s'usa la interfície Comparable com un Comparator es requerirà que l'operació equals de la classe K sigui consistent amb compare(k1,k2) (en el cas d'usar la interfície Comparator) o amb k1.compareTo(k2) (en el cas d'usar Comparable).

> Direm que k1.equals(k2) és consistent amb comparator.compare(k1,k2) si es compleix:
>
> ```
> (compare(k1, k2)==0) == (k1.equals(k2))
> ```
>
> De manera similar, k1.equals(k2) és consistent amb k1.compareTo(k2) si es compleix:
>
> ```
> (k1.compareTo(k2)==0) == (k1.equals(k2))
> ```

La interfície SortedMap<K, V> es defineix com a subinterfície de Map<K, V>:

```
1 public interface SortedMap<K, V> extends Map<K, V>{...}
```

i, afegeix a les operacions declarades per Map<K, V>, entre d'altres, les operacions següents:

Comparator<? super K> comparator() Retorna el comparador utilitzat per ordenar les claus en aquesta taula, o nul si aquesta taula utilitza l'ordre natural de les seves claus.
K firstKey() Retorna la primera clau (la més baixa) que actualment hi ha en aquesta taula.
SortedMap<K, V> headMap(K toKey) Retorna una vista de la part de la taula amb claus estrictament més petites que toKey.
K lastKey() Retorna la darrera clau (la més alta) que actualment hi ha en aquesta taula.
SortedMap<K, V> subMap(K fromKey, K toKey) Retorna una vista de la part de la taula amb claus que van de fromKey, inclòs, fins a toKey, exclòs.
SortedMap<K, V> tailMap(K fromKey) Retorna una vista de la part d'aquesta taula amb claus majors o iguals que fromKey.

A la secció 6.2.5 (del capítol 6) proposem la classe BSTMap<K, V> que implementa la interfície SortedMap<K, V> i treballa amb classes K que són Comparables (i.e., que implementen la interfície Comparable).

4.3 Implementant les taules

A la secció anterior hem presentat el comportament de les taules (de la interfície Map<K, V>). En particular, hem vist que, sigui quina sigui la representació que triem per a aquesta estructura de dades, voldrem que l'operació m.get(k) tingui un cost O(1) o O(log n), on n és el nombre d'elements continguts a la taula. La pregunta ara és evident: com ens ho farem per aconseguir això? Hi ha força estratègies i en les properes seccions en presentarem algunes. Deixem les més interessants per als capítols 5 i 6.

La JCF proposa algunes implementacions de Map<K, V> i, a més, com també feia en el cas de la interfície List<T>, ens proporciona una classe de suport que ens facilita el disseny de classes que implementen Map<K, V>. Aquesta classe de suport és AbstractMap<K, V>. Parlem-ne una mica abans de presentar les diferents estratègies d'implementació de les taules.

4.3.1 Classe AbstractMap<K, V>

El Java ofereix aquesta classe de suport per implementar les taules.

public abstract class AbstractMap<K, V> implements Map<K, V> ...

Aquesta classe no proposa cap estratègia específica d'implementació de les taules, simplement es limita a implementar algunes operacions de Map<K, V> en termes d'iteradors. Mostrem l'operació get(k) a tall d'exemple:

```
1 public V get(Object key) {
2     Iterator<Entry<K, V>> i = entrySet().iterator();
3
4     if (key == null) {
5         ...
```

```
 6      } else {
 7          while (i.hasNext()) {
 8              Entry<K, V> e = i.next();
 9              if (key.equals(e.getKey()))
10                  return e.getValue();
11          }
12      }
13      return null;
14  }
```

Comentaris:

- Veiem que l'estratègia que segueix `get` per trobar una clau al `Map` és definir un iterador sobre les parelles *(clau, valor)* d'aquest `Map`:

  ```
  1      Iterator<Entry<K, V>> i = entrySet().iterator();
  ```

  i recórrer-les fins que en trobi una que tingui la clau cercada o fins que s'arribi al final.

  Això és tot el que podem fer si no tenim una representació concreta de la taula. Aquesta representació concreta s'ajorna per a les subclasses d'`AbstractMap<K, V>`. Igualment, seran aquestes subclasses les que definiran l'operació `entrySet()`, d'acord amb la representació específica de la taula.

- L'estratègia per `get(k)` basada en el recorregut de les parelles de la taula amb un iterador té un cost O(n), on n és el nombre d'elements de la taula. Per aquest motiu no és acceptable com a implementació de `Map<K, V>`.

La importància de la classe `AbstractMap<K, V>` és que per construir una classe que implementi `Map<K, V>` n'hi ha prou de crear una subclasse `S` d'`AbstractMap<K, V>` i definir per a `S` les operacions `entrySet`, `put` i les constructores corresponents. Així i tot, serà convenient que `S` redefineixi també aquelles operacions (e.g., `get`) que siguin optimitzables amb la representació de la taula proposada per `S`.

I com podem representar aquesta subclasse `S`? Quines maneres eficients tenim de representar o implementar una taula?

4.3.2 Formes bàsiques d'implementar taules

Podem considerar estratègies diferents d'implementacions de taules, que mantenen el requeriment de l'eficiència en l'operació `get` en O(1) o bé O(log n):

- Un vector ordenat i ús de la cerca dicotòmica.
- Un vector amb una funció de mapeig injectiva.
- Un vector amb funcions de dispersió.
- Un arbre binari de cerca o derivat (e.g., AVL).
- Un arbre vermell-negre.

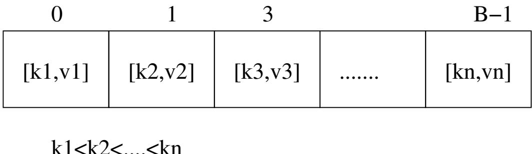


Figura 4.2: Taula implementada amb un vector ordenat per clau

- Un arbre B o derivat (e.g., B+).

Entre aquestes estratègies, les que més s'utilitzen són:

- Les basades en funcions de dispersió (vegeu 5).
- Les arborescents (ABC, AVL, vermell-negre, B...) (vegeu 6, 6.3).

Així i tot, per anar fent boca, mostrem en primer lloc a les seccions 4.3.3 i 4.3.4 les implementacions més bàsiques.

4.3.3 Vector ordenat i cerca dicotòmica

La idea d'aquesta implementació és la següent:

> Els elements de la taula s'emmagatzemen en un vector de parelles *(clau, valor)* que està ordenat per ordre ascendent de clau.
>
> L'accés a la parella que té una clau determinada es farà mitjançant l'algoritme de la cerca dicotòmica i, per tant, tindrà un cost d'O(log n).

Característiques:

- Exigeix una ordenació total de les claus. O sigui, el tipus de les claus (`K`) haurà d'implementar `Comparable` o s'haurà d'associar un `Comparator` a la taula (de manera similar a com hem indicat a 4.2.2).
- Cost de la consulta per clau (`get(k)`): O(log n) (amb cerca dicotòmica).
- Cost de la inserció/eliminació: O(n) (cal mantenir l'ordenació i, per tant, desplaçar convenientment les parelles *(clau, valor)*).
- Necessitat de tenir una estimació del nombre màxim d'elements (o redimensionar el vector quan el nombre màxim prefixat se sobrepassa).

- Permet recorreguts ordenats per clau de la taula ja que el vector que representa la taula està ordenat per clau.

Aquesta implementació és acceptable si el volum d'insercions/eliminacions a la taula és baix o si les insercions/eliminacions no són operacions crítiques.

4.3.4 Funcions injectives

La idea d'aquesta implementació és la següent:

- Dissenyarem una funció f que estableixi una aplicació injectiva entre el conjunt de claus (que anomenarem, a partir d'ara, *espai de claus*) i el conjunt d'índexs d'un vector w ($w[0...B-1]$). Aquest conjunt d'índexs l'anomenarem a partir d'ara *espai d'índexs*:

$$f : \mathtt{K} \longrightarrow [0...B-1]$$

- La parella (k, v) se situarà a l'índex $f(k)$ del vector w: $w[f(k)]$

Si $f(k) = i$, la parella (k, v) se situarà a l'índex $w[i]$.
Vector w:

0	1	2	3	..	i	..	$B-1$
					(k,v)		

Està garantit que a cada posició de w li correspon una única clau (f és injectiva).

Exemple 4.3:

- Considerem un hotel amb 7 plantes i 40 habitacions per planta.
- La numeració de les habitacions és la següent:
  - Planta 1: 100-139
  - Planta 2: 200-239
  - ...
  - Planta 7: 700-739
- Volem accedir immediatament a les dades dels clients d'una determinada habitació (nom, data arribada, despeses que han fet...).

  Per això dissenyem una taula amb:

  - Clau: número d'habitació.
  - Valor: informació dels clients d'aquella habitació.

- Implementem la taula mitjançant la funció injectiva següent:

$$f(n) = ((n \texttt{ div } 100) - 1) * 40 + (n \texttt{ mod } 100)$$

  (n és el número de l'habitació).

■ ■ ■

Característiques de la implementació de taules mitjançant funcions injectives:

- Cost inserció, eliminació, modificació, consulta: cost del còmput de f.

  Cal aconseguir una f amb cost de còmput baix (idealment O(1)).

- La mida de l'espai de claus ha de ser menor o igual que la mida de l'espai d'índexs (en cas contrari, f no pot ser injectiva).

Aquesta implementació és excel·lent... però gairebé mai no és aplicable!

Condicions d'aplicabilitat de la implementació d'una taula mitjançant funcions injectives

1. Podem trobar una funció injectiva de cost baix que mapeja l'espai de claus (K) a l'espai d'índexs (A).

2. $\mid K \mid \leq \mid A \mid$.

   La mida de l'espai de claus menor o igual que la mida de l'espai d'índexs, o sigui, com a màxim tindrem tantes claus com índexs al *vector* (vegeu la figura 4.3):

$$\neg \exists k_1, k_2 \in K \texttt{ tal que } f(k_1) = f(k_2)$$

Habitualment la condició 2 fa fracassar l'ús d'aquesta implementació.

Exemple 4.4: Cas de la biblioteca.

Volem gestionar els usuaris d'una biblioteca mitjançant una taula a què s'accedirà per NIF. Preveiem que hi haurà al voltant de 500 usuaris de la biblioteca:

- $\mid K \mid = 100.000.000$ (el nombre de NIF possibles)
- $\mid A \mid = 500$

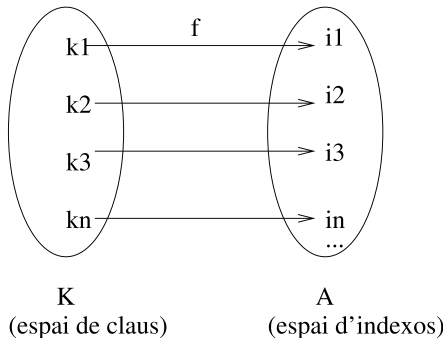


Figura 4.3: Implementació d'una taula amb funcions injectives

Dels 100.000.000 de claus possibles, només n'haurem de tractar unes 500.

Però no sabem quines!

Per tant, l'espai de claus continua tenint una mida de 100.000.000 ($\mid K \mid = 100.000.000$) (vegeu la figura 4.4).

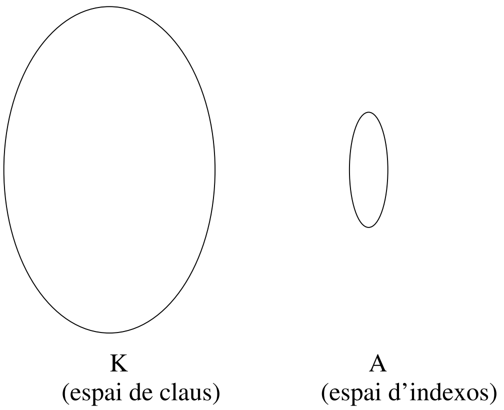


Figura 4.4: Implementació d'una taula amb funcions injectives (2)

Aquesta és la situació més habitual. I en aquesta situació no es pot plantejar una representació de la taula amb una funció injectiva.

■ ■ ■

Ara ja estem en condicions de presentar la implementació estrella de taules: mantindrem la idea d'usar una funció que mapegi l'espai de claus al d'índexs però relaxarem l'exigència de la injectivitat d'aquesta funció: l'espai de claus podrà ser més gran que l'espai d'índexs com passa a l'exemple de la biblioteca. Això ens portarà a la implementació de les taules anomenada *taules de dispersió*. M'acompanyeu?

4.4 Racó lingüístic

En aquesta secció comentem els termes tècnics usats en aquest capítol que no estan estandarditzats al diccionari de l'Institut d'Estudis Catalans, a Termcat o a [CM94].

- *Taula. Taula associativa. Taula relacional.*

  El diccionari de l'Institut d'Estudis Catalans i [CM94] inclouen el terme *taula* per referir-se a l'estructura de dades anomenada en anglès *map*. Per contra, Termcat no conté cap accepció del terme *taula* que faci referència als *maps*. L'única accepció de taula recollida per Termcat en un context informàtic es refereix a una taula d'una base de dades relacional.

  I aquesta és, precisament, la limitació del terme *taula* com a traducció de `map`: introdueix una ambigüitat entre dos conceptes ben diferents: els *maps*, per un costat i les taules d'una base de dades relacional, per un altre.

  Per aquest motiu, en aquest volum proposem resoldre l'ambigüitat usant dos termes diferents:

  – *Taula associativa*, per referir-se als *maps*.
  – *Taula relacional*, per referir-se a les taules de les bases de dades relacionals.

  Així i tot, com aquest llibre està orientat a les estructures de dades en memòria i només necessita taules relacionals de manera molt puntual al capítol 7, simplifiquem la notació anomenant simplement *taules* a les taules associatives (vegeu els capítols 3, 4, 5 i 6) i *taules relacionals* a les de les bases de dades.

  Una alternativa a *taula associativa* podria ser *diccionari*, traduint literalment el terme anglès *dictionary* que en algunes ocasions s'empra també per referir-se a les taules associatives. Però no és un terme natural en el context de les estructures de dades en llengua catalana; per això, no l'hem adoptat.

- *Índex d'un element d'un vector*.

  Usem el terme *índex d'un element d'un vector* a la posició que ocupa aquell element dins del vector. Recordem que aquesta posició és precisament la clau per accedir a aquell element.

  En realitat, aquest terme és un cas molt particular del terme *índex* que està definit pel diccionari de l'Institut d'Estudis Catalans.

- *Espai de claus. Espai d'índexs*.

  Traducció no estandarditzada dels termes anglesos *Key space* i *Address space*.

Capítol 5

Taules implementades amb funcions de dispersió

Al capítol anterior hem acabat presentant una estratègia d'implementació de taules basada en usar una funció injectiva per a mapejar un espai de claus en un espai d'índexs (d'un vector). Hem vist que aquesta estratègia és excel·lent però la immensa majoria dels cops, inaplicable perquè, en general, l'espai de claus és molt més gran que el d'índexs i això fa impossible l'ús d'una funció injectiva que apliqui l'espai de claus al d'índexs. En aquest capítol ens preguntarem què passaria si relaxéssim la petició que la funció sigui injectiva. O sigui, deixem que un mateix índex del vector pugui encabir més d'una parella (clau, valor). En aquest capítol veurem que aquesta relaxació ens porta a (segurament) l'estratègia més utilitzada per implementar una taula: l'ús de funcions de dispersió (en anglès *hash functions*). En aquest capítol, presentarem aquesta estratègia, en descriurem diferents variants, justificarem que, si està ben dissenyada, té una eficiència excel·lent, en veurem les limitacions i, finalment, presentarem la implementació que fa l'*OpenJDK* de la classe `HashMap<K, V>`.

Interessant, no us sembla?

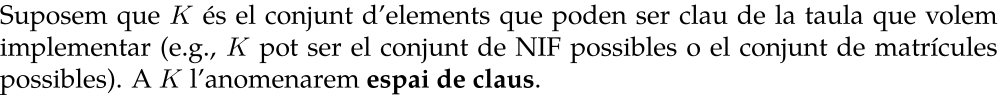
5.1 Les taules de dispersió a vista d'ocell

L'exemple de la biblioteca (vegeu la pàgina 187) us haurà convençut (espero...) que, en general, no resulta pràctic aplicar un conjunt de claus als índexs d'un vector amb una funció injectiva (i.e., de manera que cada clau s'apliqui en un índex diferent). La idea de les implementacions de taules usant funcions de dispersió (o funcions de dispersió) és mantenir aquesta aplicació de l'espai de claus a l'espai d'índexs d'un vector com fèiem a 4.3.4 però relaxant el requeriment que la funció d'aplicació hagi de ser injectiva. Com a conseqüència immediata, podrem tenir diferents claus aplicades per la funció (no injectiva) al mateix índex del vector. Veigem on ens porta tot això. I comencem formalitzant la idea d'implementació d'una taula amb una funció de dispersió o *hash*.

Suposem que K és el conjunt d'elements que poden ser clau de la taula que volem implementar (e.g., K pot ser el conjunt de NIF possibles o el conjunt de matrícules possibles). A K l'anomenarem **espai de claus**.
Suposem que A és el conjunt d'índexs del vector que usarem per emmagatzemar les parelles (*clau*, *valor*) que constitueixen la taula. A A l'anomenarem **espai d'índexs** i típicament constarà de B índexs d'un vector t, des de $t[0]$ a $t[B-1]$.
Suposem que h és una funció entre K i A:

$$h : K \longrightarrow [0...B-1]$$

La implementació de les taules mitjançant funcions de dispersió consisteix a situar la parella (k, v) a l'índex $h(k)$ del vector t. O sigui, a $t[h(k)]$:



A la funció h l'anomenarem **funció de dispersió**. Les taules implementades amb funcions de dispersió les anomenem **taules de dispersió**.

I atenció:

No estem forçant en cap moment que h sigui injectiva. Per tant, hi podrà haver més d'una clau aplicada a un índex determinat del vector.

Aquesta estratègia d'implementació resulta apropiada si l'espai de claus és més gran que l'espai d'índexs, que és la situació habitual (vegeu 5.1).

Els que tingueu una inclinació per la matemàtica potser haureu observat que h indueix una relació d'equivalència a K:

$$\forall k_1, k_2 \in K \bullet k_1 \; \texttt{R} \; k_2 \; \texttt{sii} \; h(k_1) = h(k_2)$$

Encara més, A és el conjunt quocient d'aquesta relació d'equivalència.

Exemple 5.1:

A l'exemple de la biblioteca, volem aplicar un espai de claus de 100.000.000 de NIF possibles (qualsevol ciutadà, per tant, qualsevol NIF, es podria fer soci de la biblioteca en un vector de 500 índexs (només esperem 500 socis en aquesta biblioteca)).

Per tant, és clar que una funció h que faci aquesta aplicació no pot ser mai de la vida injectiva (vegeu, altre cop, la figura 5.1).

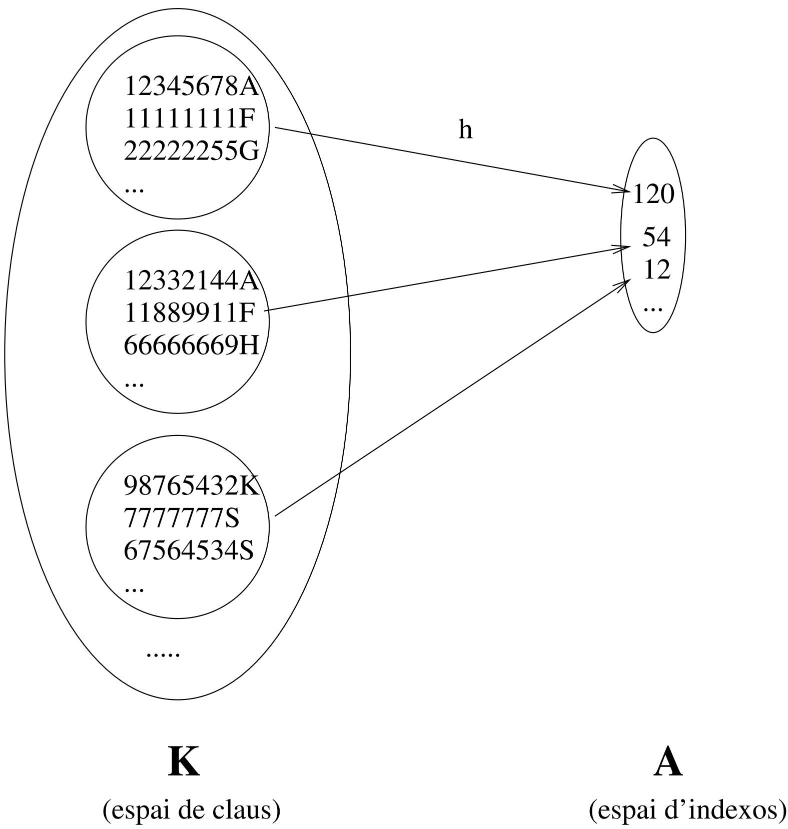


Figura 5.1: Aplicació de l'espai de claus al d'índexs mitjançant una funció de dispersió

▪ ▪ ▪

La presentació que hem fet de les taules de dispersió ens suscita dues reflexions que són les claus de les taules de dispersió:

1. La funció de dispersió h ha de tenir la propietat de **distribuir uniformement** les claus de K a l'espai d'índexs per tal d'evitar que hi hagi índexs als quals correspongui una acumulació de claus i altres índexs que rebin poques claus o cap.

   Quina forma haurà de tenir h per complir la propietat de la distribució uniforme?

2. Encara que h distribueixi molt bé, serà inevitable que h assigni el mateix índex a claus diferents (recordem que $\mid K \mid >>>> \mid A \mid$)

   Per tant tindrem claus k_1, k_2 (amb $k_1 \neq k_2$) de manera que
   $h(k_1) = h(k_2)$

   Aquestes claus s'anomenen *claus sinònimes*.

   Quina estratègia utilitzarem per situar dues claus sinònimes a l'espai d'índexs???

A les seccions 5.2 intentem contestar la primera i a les seccions 5.4 i 5.5, la segona. Però abans, acabem aquesta secció fixant una mica de terminologia que s'usa per descriure taules de dispersió:

Terminologia bàsica:

- **Espai de claus:**

  Conjunt de claus susceptibles de ser introduïdes a la taula en l'exemple que tractem.

  Les claus seran cadenes de caràcters.

- **Espai d'índexs:**

  Conjunt de tots els índexs d'un vector on podem situar les parelles (k, v)

  Considerarem B índexs numerats $0...B-1$.

  Depenent de l'estratègia de dispersió (vegeu 5.5 i 5.4) a un índex s'hi pot aplicar una única parella (k, v) o diverses.

  En aquest darrer cas, a un índex se l'anomena **contenidor** (**bucket**).

- **Funció de dispersió** (**funció de dispersió**)

  $h : K \longrightarrow [0...B-1]$

  Funció no injectiva que assigna a cada clau de l'espai de claus un índex de l'espai d'índexs ($0...B-1$).

  Com que h no és injectiva, podrà passar que envïi diverses claus al mateix índex.

  Vegeu 5.2.

- **Claus sinònimes:**

  Dues claus diferents k_1, k_2 de manera que: $h(k_1) = h(k_2)$.

  Es diu que dues claus sinònimes **col·lisionen**.

  L'objectiu de les estratègies de dispersió és gestionar adequadament les claus sinònimes (vegeu 5.5 i 5.4).

I ara ja és hora de plantejar-nos com han de ser les funcions de dispersió.

5.2 Funcions de dispersió (*)

Per facilitar la presentació d'aquesta secció, suposarem que l'espai de claus K és el tipus `String`.

Així doncs, una funció de dispersió h es defineix:
$h : String \longrightarrow [0...B-1]$
$k \longrightarrow h(k)$

I ha de complir les propietats següents:

1. h ha de ser fàcilment computable.
2. h ha d'aplicar tot l'espai d'índexs (i.e., ha de poder generar qualsevol valor entre $[0...B-1]$).
3. h ha de dispersar les claus uniformement en $[0...B-1]$.

La propietat 3 (dispersió uniforme) vol dir que h no ha de donar lloc a acumulacions de claus sobre uns quants índexs (als quals h envia moltes claus) mentre que uns altres en reben moltes menys (potser zero).

En particular, és especialment important que h dispersi uniformement:

1. Els subconjunts de claus aleatòries.

   h no ha d'introduir tendència en els subconjunts de claus que no la tenien.

   Ex. $h(k) = 0$ fóra pèssima!

2. Els subconjunts de claus que segueixen una determinada tendència previsible en el nostre domini.

   Ex. En el problema de la biblioteca, fóra pèssima la funció de dispersió següent:

   $h(k) =$ els primers dos dígits del NIF.

   ja que, presumiblement, els usuaris de la nostra biblioteca procediran de manera majoritària de la província on estigui situada aquella biblioteca i la numeració dels NIF d'una província tendeix a repetir els dos primers dígits.

   (En canvi, la h proposada dispersaria uniformement un conjunt de claus aleatòries en 100 índexs).

Hi ha dos consells pràctics que són necessaris per tal que h dispersi uniformement:

- El càlcul de $h(k)$ ha d'involucrar tots els caràcters de la clau k (s'incompleix a l'exemple anterior).
- El càlcul de $h(k)$ ha de ser sensible a l'ordre dels caràcters que componen k:

  $h(``c_1c_2c_3") \neq h(``c_2c_1c_3")$

  on c_i és el caràcter $i-sim$ de la clau k.

5.2.1 Descomposició de h

Per estudiar les funcions de dispersió resulta convenient descompondre h en dues funcions h_1, h_2:

$h : String \longrightarrow [0...B-1]$
$h_1 : String \longrightarrow \mathbb{N}$
$h_2 : \mathbb{N} \longrightarrow [0...B-1]$

de tal manera que $h(k) = h_2(h_1(k))$
($\mathbb{N}$ nota els nombres naturals).

Seguidament estudiarem totes dues funcions (h_1, h_2) per separat.

5.2.2 Funcions de dispersió: h_1

Proposem alguns exemples de funció de dispersió h_1:

1. *Suma dels codis de tots els caràcters de la clau:*

   Suposem que $k = c_1 \cdot c_2 \cdot ... \cdot c_n$
   (k és la concatenació dels caràcters c_1, c_2,..., c_n).

   $$h_1(k) = \sum_{i=1}^{|k|} \texttt{codi}(c_i)$$

   on $\texttt{codi}(c_i)$ és un codi numèric associat al caràcter c_i (e.g., el codi ASCII del caràcter).

   Característiques de h_1:

   - Facilitat de càlcul `:-)`
   - No sensible a l'ordre dels components de la clau `:-(`
     $h_1(c_1 \cdot c_2 \cdot c_3) = h_1(c_3 \cdot c_2 \cdot c_1)$
   - Fa una aplicació a un espai d'índexs possiblement petit: `:-(`
     $h_1(k) \leq \texttt{CODIMAX} * | k |$
     (`CODIMAX` és el valor màxim que pot prendre $\texttt{codi}(c)$)
   - Els índexs més petits possiblement no seran aplicats `:-(`

2. *Suma ponderada dels codis de tots els caràcters de la clau:*

   $$h_1(k) = \sum_{i=1}^{|k|} \texttt{codi}(c_i) b^i$$

b és una potència de 2 que s'apropa al nombre de caràcters diferents de l'alfabet amb què formem les claus ($b = 32, 64, 128, 256$...).

Característiques de h_1:

- Permet generar un valor diferent per a cada clau diferent `:-)`
- Permet aplicar espais d'índexs grans `:-)`
- Permet una millor distribució `:-)`
  $h_1(c_1 \cdot c_2 \cdot c_3) \neq h_2(c_2 \cdot c_3 \cdot c_1)$
- Càlcul més laboriós però eficient si $b = 2^n$ (només cal fer desplaçaments de bits).
- Possible sobreeiximent `:-(`
  Que es pot alleugerir modificant la funció:

  $$h_1(k) = \sum_{i=1}^{|k|} (\texttt{codi}(c_i) b^i) \texttt{ mod } B$$

  Però compte amb la relació entre b i B:

  – $B \neq b^r$ (si no, l'operació mòdul reté només els r darrers caràcters de la clau).
  – $B \neq b - 1$ (si no, $h_1(c_1 \cdot c_2 \cdot c_3) = h_1(c_3 \cdot c_1 \cdot c_2)$).

5.2.3 Funcions de dispersió: h_2

h_2 també s'anomena *funció de restricció d'un natural a un interval*. Vegem-ne alguns exemples:

1. *Mòdul*

$$h_2(x) = x \texttt{ mod } B$$

Aquesta és la funció de restricció més usada.

La bondat de la distribució de h_2 depèn de l'encert en escollir B:

- $B = 2r$
  $h_2(x)$ conservarà la paritat d'x (si x parell, $h_2(x)$ parell; si x senar, $h_2(x)$ senar).
- $B = 2^r$
  $h_2(x)$ depèn només dels r darrers bits d'x.
- $B = 10^r$
  $h_2(x)$ depèn només dels r darrers dígits decimals d'x.
- $B = b^r$
  $h_2(x)$ depèn només dels r darrers dígits en base b d'x.

Recepta:
B **ha de ser un nombre primer o, almenys, no ha de tenir divisors menors que 20.**

2. *Plegament*

Partir la clau en m parts de la mateixa longitud (llevat, potser, de la darrera) i combinar-les usant un determinat operador (suma, o exclusiu...).

Exemple 5.2:

$h_2(154893021938450) = 154 + 893 + 21 + 938 + 450$

Es pot completar usant un mòdul.

■ ■ ■

3. **Dígits centrals del quadrat**

$h_2(x) =$ els r dígits centrals d'x^2 $(1 \leq r \leq \log_{10} B)$

La raó d'aquesta funció és que els dígits centrals d'x^2 depenen de tots els productes parcials, no de cap part concreta d'x.

5.2.4 Una funció de dispersió habitual

Una funció de dispersió que sol funcionar molt bé la majoria de les vegades és:

$$h(k) = (\sum_{i=1}^{|k|} \texttt{codi}(c_i) b^i) \ \texttt{mod} \ B$$

amb:

- c_i $(1 \leq i \leq \mid K \mid)$: caràcters que componen la clau k.
- `codi(`c_i`)`: el codi associat al caràcter c_i en algun sistema de codificació (e.g., ASCII).
- B: primer.
- b: potència de 2 de l'ordre del nombre de caràcters diferents de l'alfabet de les claus.

Consideracions:

- B i b poden canviar per tal d'obtenir una funció més adient al conjunt de claus que componen el nostre problema particular.
- Aquesta funció s'obté de la composició de les dues funcions següents (que s'han presentat anteriorment):

$$h_1(k) = \sum_{i=1}^{|k|} \texttt{codi}(c_i) b^i$$

i

$$h_2(x) = x \texttt{ mod } B$$

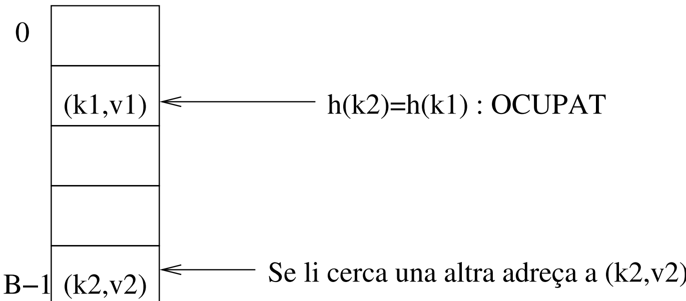
$$h(k) = h_2(h_1(k))$$

5.3 Estratègies de dispersió

Ara toca contestar la segona pregunta que ens fèiem al final de la secció 5.1:
què passa quan volem inserir dues claus k_1, k_2 sinònimes (i.e., $h(k_1) = h(k_2)$)?
Tenim dues estratègies generals:

1. *A cada índex del vector hi posem una única parella (k,v).*

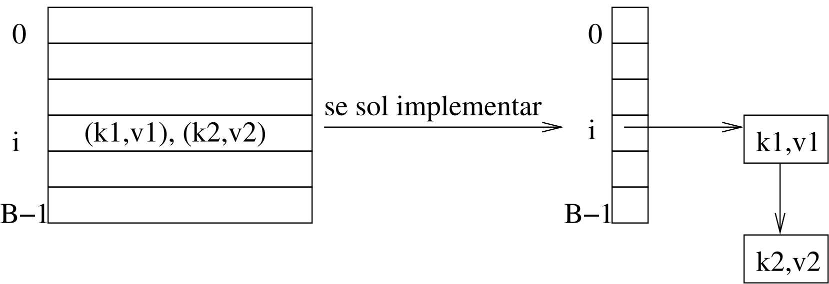
   Per tant, caldrà cercar un nou índex per col·locar k_2.

   Aquesta estratègia s'anomena **dispersió tancada** o **adreçament obert**.

2. *A cada índex del vector hi posem més d'una parella (k,v)*

   Cada índex es veu com un *contenidor* (*bucket*) capaç d'encabir diverses parelles (*k,v*). Se sol implementar associant una llista de parelles (*k,v*) a cada índex del vector usat per representar la taula:

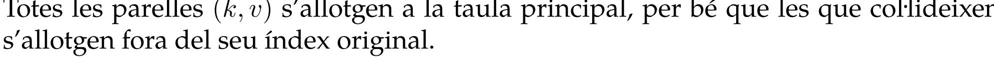


Aquesta estratègia s'anomena **dispersió oberta** o **dispersió enllaçada**.

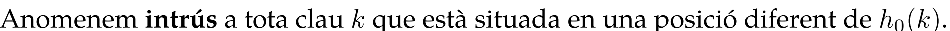
A les properes dues seccions hi entrem en detall.

5.4 Estratègies de dispersió. Dispersió tancada(*)

La dispersió tancada es basa en l'estratègia següent:



Totes les parelles (k, v) s'allotgen a la taula principal, per bé que les que col·lideixen s'allotgen fora del seu índex original.



Anomenem **intrús** a tota clau k que està situada en una posició diferent de $h_0(k)$.

Cada clau tindrà una llista d'índexs alternatius que seran provats en ordre fins a trobar-ne un que estigui lliure.

$$h(k) = h_0(k),\ h_1(k),\ h_2(k), ...,\ h_{B-1}(k)$$

$h_i(k)$ és l'índex on intentarem col·locar la clau k si els índexs $h_0(k), h_1(k), ..., h_{i-1}(k)$ estaven ocupats.
Aquesta llista d'índexs alternatius s'anomena **seqüència de proves de la clau** k.

El conjunt de funcions h_i $(0 \leq i < B)$ s'anomena **família de funcions de dispersió.**

Ara toca veure com són els algorismes d'inserció, consulta i eliminació en una dispersió tancada.

5.4.1 Algorismes d'inserció, consulta i eliminació

Donem un cop d'ull a aquests algorismes.

Cada índex del vector *t* que s'usa per representar la taula conté una terna: (*k,v,marca*). *marca* indica l'estat d'aquell índex: pot ser *ocupat*, si l'índex conté una parella (*k,v*) de la taula; pot ser *lliure*, si l'índex no conté cap parella (*k,v*) de la taula o pot ser *esborrat* si l'índex va contenir una parella (*k,v*) però aquesta va ser eliminada en algun moment anterior.

En aquestes condicions, i a grans trets, la idea d'aquests algorismes és la següent:

- *Consulta del valor associat a una clau k:*

  Calcular $i_0 = h_o(k), i_1 = h_1(k), ..., i_j = h_j(k)$ fins a trobar un índex j de manera que $t[i_j]$ està *ocupat* amb la clau k o bé $t[i_j]$ és un índex *lliure*. En aquest segon cas conclourem que la clau cercada no es troba a la taula.

- *Inserció d'una parella (k,v):*

  Calcular $i_0 = h_o(k), i_1 = h_1(k), ..., i_j = h_j(k)$ fins a trobar un índex j de manera que $t[i_j]$ està *ocupat* amb la clau k o bé $t[i_j]$ és *lliure*. En aquest segon cas conclourem que la clau cercada no es troba a la taula i s'inserirà la parella (k, v) a $t[i_j]$ (que canviarà el seu estat a *ocupat*). En el primer cas, es modificarà el valor associat a k per v.

- *Eliminació d'una clau k i del seu valor associat:*

  Calcular $i_0 = h_o(k), i_1 = h_1(k), ..., i_j = h_j(k)$ fins a trobar un índex j de manera que $t[i_j]$ està *ocupat* amb la clau k o bé $t[i_j]$ està lliure. En aquest segon cas conclourem que la clau cercada no es troba a la taula. En el primer cas, es marcarà $t[i_j]$ com a *esborrat*.

I ara una reflexió: per què ens compliquem la vida definint tres estats per a cada índex del vector? Per què no només dos (*lliure* i *ocupat*)? Què ens aporta l'estat *esborrat*? Aquesta és una qüestió important. Penseu-hi un moment, sisplau.

La resposta és la següent: l'algorisme d'eliminació haurà de marcar l'índex d'on fa l'eliminació com a *esborrat* (no simplement com a *lliure*) per tal que l'algorisme de consulta no interrompi el recorregut de la llista de sinònims quan trobi un índex esborrat.

Un índex marcat com a *esborrat* serà considerat com a:

- *lliure* per l'algorisme d'inserció.

  (però, compte!: abans de poder usar un índex esborrat, l'algorisme d'inserció s'haurà d'assegurar que la clau que es vol inserir no es troba ja a la taula).

- *ocupat* per l'algorisme de consulta.

Tot seguit presentem els tres algorismes que regulen les insercions, consultes i eliminacions en una taula de dispersió.

5.4.2 Algorisme d'inserció

```
acció inserir (k: TClau, v: TValor) és
  lliure:=fals;
  trobEsborrada:=fals;
  trobat:=fals;
  i := 0;
  mentre (i < B ∧ ¬lliure ∧¬ trobat) fer
    pos:= h_i(k);
    si t[pos].marca= LLIURE llavors lliure:= cert;
    si no si t[pos].marca= ESBORRADA llavors
        trobEsborrada:=cert;
        posEsborrada:=pos;
    si no si t[pos].clau= k llavors
        trobat:=cert;
    fsi
    i := i + 1;
  fmentre
  si trobat llavors t[pos].valor:= v;
  si no si trobEsborrada llavors
    t[posEsborrada].clau:= k;
    t[posEsborrada].valor:= v;
    t[posEsborrada].marca:=OCUPADA;
  si no si lliure llavors
    t[pos].clau:= k;
    t[pos].valor:= v;
    t[pos].marca:=OCUPADA;
  si no ERROR (no hi ha espai lliure a t)
  fsi
facció
```

Comentaris:

- *trobat*=cert: la clau k és present al vector t a l'índex *pos*.
- *trobEsborrada*=cert: s'ha trobat un índex amb *marca*=ESBORRADA a l'índex *posEsborrada*.
- *lliure*=cert: s'ha trobat un índex amb *marca*=LLIURE.
- L'algorisme d'inserció, en el moment que troba una posició esborrada continua la seva cerca per assegurar-se que la clau que es vol inserir no es trobi ja a la taula.
- La condició de sortida del bucle és:
  - *i*=*B*: hem fet B proves, ho interpretarem com que el vector és ple. No hi cap una altra parella.

    i=*B* no sempre ha de significar que el vector estigui ple. Podria ser que la família h_i ($0 \leq i < B$) no apliqués tots els índexs del vector (existeix un índex s de manera que $h_i(k) \neq s$ per a tot i: $0 \leq i < B$). En qualsevol cas, mai no hauríem d'arribar a fer B proves per inserir una clau. Si aquest és el cas vol dir que la nostra taula de dispersió ha degenerat (és massa plena) o bé que la funció h és inadequada.

– *lliure*=*cert*: la darrera prova ha trobat un índex lliure i no hem trobat, anteriorment, la clau k. Inserirem la parella en aquest índex o a *posEsborrada* si prèviament s'havia trobat una posició esborrada.

– *trobat*=*cert*: la darrera prova ha trobat un índex que conté la clau que volem inserir. Modificarem el valor associat a aquest índex.

5.4.3 Algorisme de consulta

```
funció consulta (k: TClau) retorna (v: TValor, trobat: booleà) és
  lliure:=fals;
  trobat:=fals;
  i := 0;
  mentre (i < B ∧ ¬lliure ∧¬ trobat) fer
    pos:= h_i(k);
    si t[pos].marca= LLIURE llavors lliure:= cert;
    si no si t[pos].clau= k ∧ t[pos].marca=OCUPADA llavors
        trobat:=cert;
    si no i := i + 1;        fsi
  fmentre
  si trobat llavors v := t[pos].valor; fsi
facció
```

Comentaris:

- *trobat*=cert: s'ha trobat una casella ocupada amb la clau k.
- *lliure*=cert: s'ha trobat una casella lliure $\Longrightarrow k \notin t$.
- Notem que l'algorisme de consulta tracta les caselles amb marca=ESBORRADA com si fossin caselles ocupades (i.e., cal continuar la cerca fins a fer B intents o fins que es trobi una posició amb marca=LLIURE.

5.4.4 Algorisme d'eliminació

```
acció eliminació (k: TClau) és
   lliure:=fals;
   trobat:=fals;
   i := 0;
   mentre (i < B ∧ ¬lliure ∧¬ trobat) fer
      pos:= h_i(k);
      si t[pos].marca= LLIURE llavors lliure:= cert;
      si no si t[pos].clau= k ∧ t[pos].marca=OCUPADA llavors
            trobat:=cert;
      si no i := i + 1;        fsi
   fmentre
   si trobat llavors t[pos].marca:=ESBORRADA;
   si no k no existia a t
   fsi
facció
```

5.4.5 Famílies de funcions de dispersió

Fins ara hem suposat que existia una família de funcions $h_i(k)$ per $i = 0...B-1$. Ara és el moment de veure quin aspecte poden tenir aquestes famílies de funcions:

Família de funcions de dispersió:
Família de B funcions $(h_0, ..., h_{B-1})$ que generen una seqüència de B candidats a allotjar una clau k:

$h_0(k) = h(k),\ h_1(k),\ h_2(k), ...,\ h_{B-1}(k)$

$h_i : String \longrightarrow [0...B-1]$

$h_i(k)$ s'anomena la prova i-èsima de la clau k.

Hi ha tres famílies habituals de funcions de dispersió:

- Família de dispersió lineal.
- Família de dispersió quadràtica.
- Família de dispersió doble.

És desitjable que una família de funcions de dispersió compleixi dues propietats:

1. Tots els índexs j del vector t són candidats a allotjar una clau k qualsevol:

$\forall k \in String, \forall j, 0 \leq j < B : (\exists i, 0 \leq i < B : h_i(k) = j)$

Aquesta propietat assegura que una clau no s'allotja **únicament si la taula és plena**.

2. Dues claus sinònimes tenen seqüències d'allotjament diferents.

   Si $k_1 \neq k_2$ i $h(k_1) = h(k_2)$

   les seqüències:

   $h_0(k_1),\ h_1(k_1), ...,\ h_{B-1}(k_1)$ i

   $h_0(k_2),\ h_1(k_2), ...,\ h_{B-1}(k_2)$

   són força diferents.

   Aquesta propietat evita fenòmens d'apinyament (vegeu l'explicació més endavant).

5.4.6 Família de dispersió lineal

Es defineix de la manera següent:

$$h_i(k) = (h(k) + ci) \ \texttt{mod} \ B$$

on c és una constant (usualment $c = 1$).

Propietats:

- Els sinònims es van col·locant equiseparats cadascun de l'anterior (si $c = 1$, cada sinònim es posa a la següent posició lliure del vector).
- Si c i B són primers entre si, es generen successivament totes les posicions del vector.
- Tots els sinònims tenen la mateixa seqüència de dispersió (i aquesta és la mateixa que la dels intrusos que es troben al seu camí).

  Aquesta propietat fa que la dispersió lineal generi **apinyament primari** (*primary clustering*) (vegeu més endavant).

Exemple 5.3:

Considerem la taula de dispersió definida de la manera següent:

$B = 11$
$h(k) = k \ \texttt{mod} \ 11$
$h_i(k) = (h(k) + i) \ \texttt{mod} \ 11$
(Per simplificar l'exemple, prenem una clau k numèrica.)

- *Seqüència de claus a inserir:*

  27, 49, 104, 16, 17, 18, 87

- *Insercions:*
  - $h_0(27) = 5$
  - $h_0(49) = 5; h_1(49) = 6$
  - $h_0(104) = 5; h_1(104) = 6; h_2(104) = 7$
  - $h_0(16) = 5; h_1(16) = 6; h_2(16) = 7; h_3(16) = 8$
  - $h_0(17) = 6; h_1(17) = 7; h_2(17) = 8; h_3(17) = 9$
  - $h_0(18) = 7; h_1(18) = 8; h_2(18) = 9; h_3(18) = 10$
  - $h_0(87) = 10; h_1(87) = 0$

0	1	2	3	4	5	6	7	8	9	10
87					27	49	104	16	17	18

■ ■ ■

L'exemple mostra que la dispersió lineal genera problemes d'apinyament primari. Vegem-ho.

Apinyament primari:

Es diu que una família de funcions de dispersió genera apinyament primari si per a qualsevol clau k es compleix:

1. La seqüència de proves de k és la mateixa que la seqüència de proves de qualsevol k_2 sinònim de k:

$$\forall i, 0 \leq i \leq B-1 : h_i(k) = h_i(k_2)$$

2. La seqüència de proves de k és la mateixa que la seqüència de proves d'un intrús qualsevol k_2 situat a $h_0(k)$

   (a partir de la prova j, $(0 < j < B)$ que ha servit perquè l'intrús k_2 es posés a l'índex que correspon a k: $h_0(k) = h_j(k_2)$):

$$\exists j, 0 < j < B : \forall i, 0 \leq i < B : h_i(k) = h_{j+i}(k_2).$$

Exemple 5.4:

I ara, si recordem l'exemple anterior, podrem veure que, efectivament, *les famílies de funcions de dispersió lineal generen apinyament primari*:

- Les claus sinònimes (27, 49, 104, 16) tenen la mateixa seqüència de dispersió (propietat 1).

- A més a més, quan una clau (e.g., 17) es troba amb un intrús a la posició que li pertoca (49), la seqüència de dispersió de 17 i la de 49 (a partir de $h_1(49)$) són idèntiques (propietat 2).

- L'apinyament primari provoca cadenes de sinònims i d'intrusos que tendeixen a allargar-se i a fer degenerar la taula (l'accés cada vegada es va tornant més seqüencial).

  La probabilitat que una nova inserció a la taula faci allargar encara més una cadena "seqüencial" de sinònims i intrusos creix cada cop (qualsevol clau que sigui adreçada a una posició d'una d'aquestes cadenes contribuirà a fer-la una unitat més llarga).

  A l'exemple, s'ha format una cadena de 7 sinònims i intrusos.

  Qualsevol nova clau k_r que sigui adreçada per h a alguna de les posicions ocupades (e.g., $h(k_r) = 9$) augmentarà en una unitat la longitud de la cadena (k_r se situarà a l'índex 1).

■ ■ ■

5.4.7 Família de dispersió quadràtica

Aquesta família es defineix de la manera següent:

$$h_i(k) = (h(k) + ci^2) \ \texttt{mod} \ B$$

on c és una constant (usualment $c = 1$).

Propietats:

- En general no hi ha garantida que la seqüència de proves d'una clau generi totes les posicions del vector $(0...B-1)$.

- Dues claus sinònimes tenen la mateixa seqüència de proves (apinyament secundari. Vegeu més endavant).

- Una clau ja no té la mateixa seqüència de proves que un intrús que es trobi al seu camí (elimina l'apinyament primari).

- Tot i l'apinyament secundari, funciona bastant millor que la família de dispersió lineal.

  En particular, acaba amb les seqüències de sinònims i intrusos.

Exemple 5.5:
Considerem la taula de dispersió definida de la manera següent:

$B = 11$
$h(k) = k \texttt{ mod } 11$
$h_i(k) = (h(k) + i^2) \texttt{ mod } 11$
(Per simplificar l'exemple, prenem una clau k numèrica.)

- *Seqüència de claus a inserir:*
  27, 49, 104, 16, 17, 18, 87
- *Insercions:*
  - $h_0(27) = 5$
  - $h_0(49) = 5$; $h_1(49) = 6$
  - $h_0(104) = 5$; $h_1(104) = 6$; $h_2(104) = 9$
  - $h_0(16) = 5$; $h_1(16) = 6$; $h_2(16) = 9$; $h_3(16) = 3$
  - $h_0(17) = 6$; $h_1(17) = 7$
  - $h_0(18) = 7$; $h_1(18) = 8$;
  - $h_0(87) = 10$;

0	1	2	3	4	5	6	7	8	9	10
			16		27	49	17	18	104	87

■ ■ ■

L'exemple mostra que la dispersió quadràtica genera problemes d'apinyament secundari. Vegem-ho.

Apinyament secundari:

Es diu que una família de funcions de dispersió genera apinyament secundari si, per a qualsevol clau k es compleix que:

La seqüència de proves de k és la mateixa que la seqüència de proves de qualsevol k_2 sinònim de k:

$$\forall i : 0 \leq i \leq B - 1 \bullet h_i(k) = h_i(k_2)$$

O sigui, la propietat 1 de l'apinyament primari.

I, efectivament, si recordem el darrer exemple, comprovarem com *la família de funcions de dispersió quadràtica genera apinyament secundari*:

- Les claus sinònimes (27, 49, 104, 16) tenen la mateixa seqüència de dispersió.

Com a aspecte positiu, hem d'observar que *la família de funcions de dispersió quadràtica resol l'apinyament primari de la família lineal:*

- Quan una clau (e.g., 17) es troba amb un intrús a la posició que li pertoca (49), la seqüència de dispersió de 17 i la de 49 (a partir de $h_1(49)$) **ja no són idèntiques** i, per tant, la inserció de 17 no contribueix a allargar la cadena de sinònims i intrusos (17 es col·loca a la segona prova, mentre que abans es col·locava a la quarta).

L'apinyament secundari no és ni de bon tros tan greu com l'apinyament primari ja que només provoca problemes amb els sinònims i si la taula està ben dimensionada i la funció h dispersa adequadament, el nombre de sinònims no ha de ser gaire elevat.

5.4.8 Família de dispersió doble

Aquesta família de dispersió es defineix de la manera següent:

$$h_i(k) = (h(k) + ih'(k)) \texttt{ mod } B$$

on h' és una altra funció de dispersió.

Propietats:

- És poc probable que dues claus que siguin sinònimes respecte de la funció h també ho siguin respecte de h'.
- La dispersió doble resol el problema de l'apinyament secundari.

Exemple 5.6:

Considerem la taula de dispersió definida de la manera següent:

$B = 11$

$h(k) = k \texttt{ mod } 11$

$h'(k) = \sum_{r=1}^{r=|k|} k_i$
(Suposem que $k = k_1 \cdot k_2 \cdot ... \cdot k_n$; k_i és un dígit de k.)

$h_i(k) = (h(k) + h'(k)) \texttt{ mod } 11$

(Per simplificar l'exemple, prenem una clau k numèrica.)

- *Seqüència de claus a inserir:*
  27, 49, 104, 16, 17, 18, 87
- *Insercions:*
  - $h_0(27) = 5$
  - $h_0(49) = 5; h_1(49) = (5 + 13) \texttt{ mod } 11 = 7$
  - $h_0(104) = 5; h_1(104) = (5 + 5) \texttt{ mod } 11 = 10$
  - $h_0(16) = 5; h_1(16) = (5 + 7) \texttt{ mod } 11 = 1$
  - $h_0(17) = 6$
  - $h_0(18) = 7; h_1(18) = (7 + 9) \texttt{ mod } 11 = 5;$
    $h_2(18) = (7 + 18) \texttt{ mod } 11 = 3$
  - $h_0(87) = 10; h_1(87) = (10 + 15) \texttt{ mod } 11 = 3;$
    $h_2(87) = (10 + 30) \texttt{ mod } 11 = 7; h_3(87) = (10 + 45) \texttt{ mod } 11 = 0$

0	1	2	3	4	5	6	7	8	9	10
87	16		18		27	17	49			104

En aquest exemple es pot apreciar que *la dispersió doble elimina els problemes d'apinyament primari i secundari*.

■ ■ ■

5.4.9 Dispersió Robin Hood: una variant de la dispersió tancada

L'estratègia de dispersió *Robin Hood* segueix la idea següent:

Estratègia Robin Hood:

- Planteja l'objectiu d'igualar el nombre d'intents necessaris per inserir les diferents claus.
- D'aquesta manera s'evitarà que hi hagi **claus privilegiades** que van ser inserides amb un nombre de proves molt petit i d'altres de **castigades** que han necessitat moltes proves per tal de ser inserides.

Esquema de l'algorisme d'inserció d'una clau

Seguint aquesta idea, vegem esquemàticament com es desenvolupa la prova i per situar la clau k:

- $pos = h_i(k)$
- Suposem que $v[pos]$ està ocupada per una clau k' que s'hi va situar a la prova j:
  - Si $i \leq j \longrightarrow$ Provar la inserció de k a $h_{i+1}(k)$

- Si $i > j \longrightarrow$
    - Inserir k a $v[pos]$ (i, per tant, desallotjar k' de $v[pos]$).
    - Provar la inserció de k' a $h_{j+1}(k')$.

5.5 Estratègies de dispersió. Dispersió oberta(*)

La dispersió oberta es basa en l'estratègia següent:
Cada índex (`0...B-1`) representa un *contenidor* (o *bucket*) on cap més d'una clau amb el seu valor associat.

0	
	<k1,v1>, <k2,v2> ... <kn,vn>
B−1	

h(k1)=h(k2)=...=h(kn)

La dispersió oberta evita l'apinyament primari i les tècniques de readreçament.

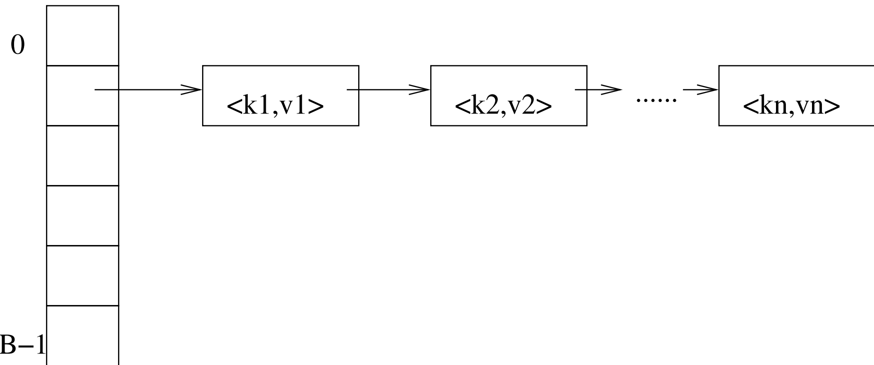
5.5.1 Implementació de la dispersió oberta

Usualment es consideren dues tècniques:

- La llista de sinònims es manté enllaçada en memòria dinàmica.
- La llista de sinònims es manté en un *vector d'excedents*.

Usualment, a la dispersió oberta també se l'anomena *dispersió enllaçada*.

- *Dispersió oberta amb enllaços en memòria dinàmica*.

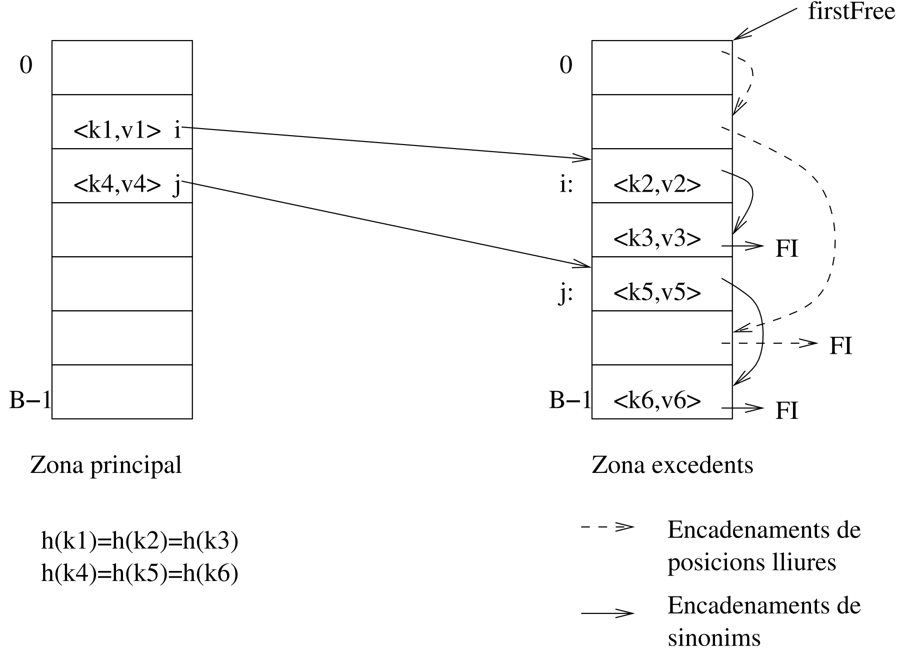


Característiques:

- Podem tenir més de `B` parelles a la taula.
- En realitat, podem inserir noves parelles mentre hi hagi memòria disponible. (**Però, compte!, la taula degenerarà si hi ha molts sinònims**).
- Els sinònims s'allotgen en memòria dinàmica que **és gestionada pel sistema, no pel programador**.
- Estem fent servir espai addicional per implementar els punters.

- *Dispersió oberta amb enllaços en zona d'excedents*

  La figura següent mostra de manera gràfica quina és la idea d'aquesta implementació de la dispersió oberta:



Característiques:

- No cal usar punters (els punters són, en realitat, índexs d'un vector).
- No es reutilitza la classe `LinkedList` per gestionar els sinònims.
- La zona d'excedents actua com a espai lliure per allotjar-hi sinònims. La gestió d'aquest espai lliure la fa el programador, no el sistema com en el cas anterior. Per tant, la implementació és més complexa.
- L'espai per a sinònims està limitat a la capacitat de la zona d'excedents. Si s'excedeix aquest espai, cal redimensionar la taula.

La JCF defineix una classe `HashMap<K, V>` i la implementa usant l'estratègia de *dispersió oberta*. Tot seguit ho presentem (vegeu la secció 5.7). Però abans, potser serà interessant donar un cop d'ull a quin rendiment permeten assolir les taules de dispersió.

5.6 Cost de les taules de dispersió (*)

En aquesta secció aprendrem que si les taules de dispersió estan ben dissenyades, la seva eficiència és excel·lent. Per això, estudiarem el cost de les dues estratègies de dispersió: la tancada i l'oberta. Però primer, ens cal definir un parell de conceptes.

5.6.1 Definicions prèvies

- $E(i, B)$: és el nombre mitjà de proves que cal fer per inserir l'element $i+1$-. èsim en una taula amb capacitat per a B elements
- $T_f(N, B)$: nombre mitjà de proves que cal fer en una dispersió tancada abans. d'adonar-nos que una clau no pertany a una taula amb capacitat per a B elements i N elements presents
- $T_s(N, B)$: nombre mitjà de proves que cal fer en una dispersió tancada. per trobar una clau present en una taula amb capacitat per a B elements i N elements presents

Per estudiar els costos calcularem de la dispersió tancada i de la dispersió oberta T_f i T_s. I per calcular T_f i T_s ens ajudarem de E. Calculem, ara sí, el cost de les dispersions tancada i oberta.

5.6.2 Cost de la dispersió tancada

Càlcul de $E(i, B)$

- $E(0, B) = 1$
  Per inserir el primer element en una taula buida només cal fer una prova.

- $E(i, B) = \frac{B-i}{B} \cdot 1 + \frac{i}{B}(1 + E(i-1, B-1))$

  Per què?:

  Es fa la hipòtesi que h **distribueix uniformement (1)**, i, amb aquesta hipòtesi:

  - La probabilitat que la inserció de l'element $i+1$ es faci amb una única prova és $\frac{B-i}{B}$ perquè hi ha $B-i$ cel·les buides i B cel·les totals.
  - Si la primera prova ha generat una posició ocupada (això ha passat amb probabilitat $\frac{i}{B}$), haurem de fer una segona prova per col·locar la clau.

    Si fem la hipòtesi que h **no repeteix posicions (2)**, el nombre de proves que necessitarem serà:

    1 (la primera) + el nombre de proves per col·locar la i-èsima clau en un vector amb capacitat $B-1$ (ja que les successives proves no ens tornaran a generar les posicions ja provades): $1 + E(i-1, B-1)$.

La solució de l'equació recurrent plantejada és:

$$E(i, B) = \frac{B+1}{B+1-i} \quad (0 \leq i < B)$$

Factor de càrrega

Definim factor de càrrega (α) com:

$\alpha = \frac{N}{B+1}$

- N: nombre d'elements presents a la taula
- B: capacitat de la taula

α indica aproximadament la ràtio d'ocupació de la taula.

Càlcul de T_f i T_e

- $T_f(N, B) \approx E(N, B) = \frac{a}{1-\alpha}$

  (Hem de fer les mateixes proves per inserir una nova clau que per adonar-nos que una clau no hi és.)

- $T_e(N, B) \approx \frac{E(0,B)+E(1,B)+\ldots+E(N-1,B)}{N} \approx \frac{1}{\alpha} Ln(\frac{1}{1-\alpha})$

  (Mitjana del nombre de proves necessàries per inserir cada clau, ja que la clau que cerquem serà una de les inserides i costarà el mateix trobar-la que el que va costar inserir-la.)

Conclusions:
El cost de la consulta d'una clau en una dispersió tancada depén:

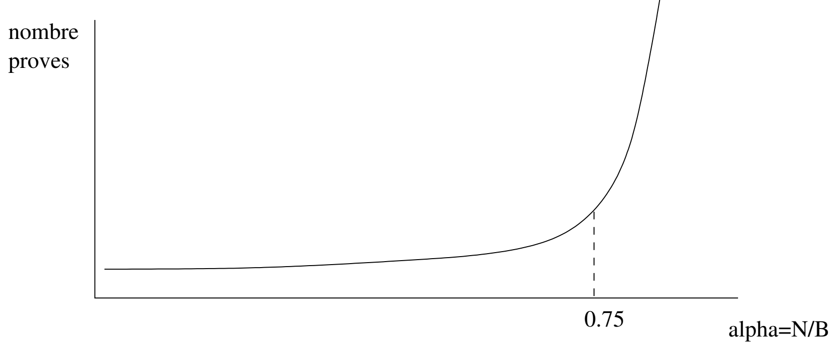
1. De la bondad de h:
    - h ha de distribuir uniformement les claus que es volen inserir.
    - h no ha de generar posicions repetides en proves diferents.
    - h s'ha de poder calcular eficientment.
2. De la ràtio d'ocupació de la taula (α).

El cost no depèn directament del nombre d'elements emmagatzemats a la taula sinó de la ràtio entre el nombre d'elements emmagatzemats i la capacitat del vector.

Exemple 5.7:

- $\alpha = 0.5: \quad T_f = 2 \;\; T_e = 1.39 \Longrightarrow O(1)$
- $\alpha = 0.75: \quad T_f = 4 \;\; T_e = 1.85 \Longrightarrow O(1)$

  Per a una taula plena al 75 %, el nombre de proves que hauríem de fer per adonar-nos que una clau no hi és és aproximadament 4 (independentment de si a la taula hi ha 100, 1000 o 10000 claus).
- $\alpha \approx 1: \quad T_f \to N \;\; T_e \to N \Longrightarrow O(N)$



Una taula implementada com a dispersió tancada que:

- Està dotada d'una funció de dispersió que dispersa uniformement les claus que volem introduir a la taula.
- Té una ràtio d'ocupació menor o igual al 75 %.

Té un cost dels algorismes de inserció, eliminació o consulta d'una clau de O(1).

■ ■ ■

5.6.3 Cost de la dispersió oberta

Consideracions:

- Fem novament la hipòtesi que h distribueix uniformement
- En una dispersió oberta, N pot ser més gran que B (ja que en una posició de la taula hi pot haver més d'una clau).
- El nombre mitjà de claus sinònimes que hi haurà en una posició és $\alpha = N/B$ (perquè les N claus presents a la taula es distribuiran uniformement entre les B posicions disponibles).

Càlcul de T_f i T_e

- $T_e(N, B) \approx 1 + \frac{1}{2}\alpha$
  1 fa referència a l'accés inicial a $h(k)$.
- $T_f(N, B) \approx \alpha$
  Caldrà recórrer la llista de sinònims sencera.

Conclusions

- Com en el cas de la dispersió tancada, el cost depèn només de α.
- Si h dispersa bé les claus de la nostra taula i α no és massa gran:
  Cost de les insercions, consultes o eliminacions en una taula implementada com a dispersió oberta: $O(1)$.

5.6.4 Comparació de costos entre les dues estratègies de dispersió

Cost temporal dispersió oberta vs. dispersió tancada

- Quantitativament, els resultats anteriors de T_f i T_s afavoreixen una mica la dispersió oberta.
- El més important és que en tots dos casos s'assoleix un cost $O(1)$ si:
  - h distribueix uniformement.
  - La taula està ben dimensionada (B adequada).

  I recordem que:

  h i B són paràmetres que el programadador pot controlar.

- Si una funció de dispersió distribueix malament només un percentatge de les claus de la taula, la dispersió tancada farà degenerar **tota la taula** mentre que en una dispersió oberta només es veuran afectades les claus mal distribuïdes.

5.6.5 Quan és adient implementar una taula mitjançant alguna estratègia de dispersió?

- La implementació d'una taula mitjançant funcions de dispersió és **excel·lent** si es compleixen les condicions següents:
  - Tenim un espai de claus molt gran que cal aplicar en un espai d'índexs significativament més petit (e.g., els 500 usuaris d'una biblioteca identificats per NIF).
  - Podem trobar una funció de dispersió que distribueixi uniformement les claus de la nostra situació concreta i que es pugui calcular eficientment.
  - Mantenim la taula amb una ràtio d'ocupació (α) raonable (si α puja molt, aleshores cal redimensionar la taula: generar una B més gran i inserir totes les claus des de la taula vella a la nova).
  - **No** s'han de fer recorreguts de la taula ordenats per clau.
- Contràriament, la implementació d'una taula mitjançant funcions de dispersió és una **mala idea** si
  - Cal recórrer la taula en ordre de claus.

5.7 Implementació de les taules de dispersió al Java: HashMap<K, V>

L'API del Java ofereix la classe HashMap<K, V>. HashMap<K, V> es defineix com una subclasse d'AbstractMap<K, V> que implementa la interfície Map<K, V> usant funcions de dispersió.

En particular, l'*OpenJDK* ofereix una implementació de HashMap<K, V> que implementa les taules de dispersió usant l'estratègia de *dispersió oberta*.

A les seccions següents donarem un cop d'ull als aspectes més rellevants d'aquesta implementació.

Per tal de no apartar-nos de l'objectiu fonamental d'aquests apunts, no hem considerat alguns aspectes de la implementació de l'*OpenJDK*. Si voleu revisar amb més detall el codi íntegre, el trobareu a:
`http://grepcode.com/snapshot/repository.grepcode.com/java/root/jdk/openjdk/6-b14/`

L'*OpenJDK* és un projecte de programari lliure. Com a conseqüència, podeu consultar el seu codi i aprendre de quina manera bons programadors van resoldre els problemes de disseny amb què es van trobar. El codi lliure és una manera magnífica de posar el coneixement a l'abast de tothom. Consultar-lo i, fins i tot, col·laborar-hi és una manera esplèndida d'aprendre a programar.

5.7.1 `HashMap<K, V>`

Comencem mostrant la definició que fa l'*OpenJDK* de la classe `HashMap<K, V>`.

Llistat 5.1: Definició de la classe `HashMap<K, V>`

```
1  public class HashMap<K, V>
2                 extends AbstractMap<K, V>
3                 implements Map<K, V>, Cloneable, Serializable {
4
5      Entry[] table;
6      int size;
7
8      static final int DEFAULT_INITIAL_CAPACITY = 16;
9      static final int MAXIMUM_CAPACITY = 1 << 30;
10     static final float DEFAULT_LOAD_FACTOR = 0.75f;
11
12     int threshold;
13     final float loadFactor;
14     ...
15 }
```

Comentaris:

- La taula de dispersió es representa essencialment com un vector `table` de nodes (`Entry<K, V>`).

  La classe `Entry<K, V>` implementa la interfície `Map.Entry<K, V>` que, com ja hem explicat a 4.2.1, modelitza una parella (*clau, valor*). Així doncs, *la taula de dispersió es representa com un vector de parelles (clau, valor)*.

  A més a més, cada parella *(clau, valor)* contindrà un enllaç al següent sinònim per poder implementar d'aquesta manera la dispersió oberta. De tot això en parlarem amb més detall a 5.7.3.

- `DEFAULT_INITIAL_CAPACITY = 16`

  Un objecte `HashMap<K, V>` és redimensionable. La capacitat inicial per defecte és 16 (el fet de ser un nombre petit disminueix el cost de les iteracions per a tota la taula).

  La capacitat inicial es pot inicialitzar amb un paràmetre d'un constructor.

> En qualsevol cas, la capacitat de la taula en la implementació de l'*OpenJDK* de `HashMap<K, V>` ha de ser una potència de 2.

- `MAXIMUM_CAPACITY = 1 « 30`

  La capacitat màxima del vector que contindrà la taula de dispersió és de 2^{30}.

  Podeu raonar que 1 « 30 és precisament a 2^{30}?

- `DEFAULT_LOAD_FACTOR = 0.75f`

  El factor de càrrega per defecte. Si se supera el factor de càrrega, cal redimensionar el vector.

- `loadfactor`

  El factor de càrrega màxim del vector que representa la taula.

- `threshold`

  El proper nombre d'elements al vector de manera que, quan s'assoleixi, cal redimensionar el vector.

  Per tant, *threshold* = *capacitat de la taula* × *factor de càrrega*.

  Exemple 5.8:

  Si la capacitat de la taula és 100 i el factor de càrrega és 0,75, *threshold* valdrà 75 i indicarà que quan la taula assoleixi 75 elements cal redimensionar-la.

  ■ ■ ■

- Quan es crea un `HashMap<K, V>`, es pot fixar *la capacitat inicial* del vector que contindrà els elements de la taula i també el seu *factor de càrrega*. La capacitat del vector (i, per tant, de la taula) s'incrementarà automàticament quan el nombre d'elements que contingui superi el factor de càrrega.

  Això ho veurem a la propera secció.

5.7.2 Les constructores

Proposem dues constructores:

- La constructora per defecte:

```
1    public HashMap() {
2        this.loadFactor = DEFAULT_LOAD_FACTOR;
3        threshold = (int)(DEFAULT_INITIAL_CAPACITY *
4                          DEFAULT_LOAD_FACTOR);
5        table = new Entry[DEFAULT_INITIAL_CAPACITY];
6    }
```

Crea un `HashMap<K, V>` amb el factor de càrrega per defecte (0,75) i la capacitat inicial per defecte (16).

- La constructora que permet a l'usuari fixar una capacitat inicial i un factor de càrrega.

```
1  public HashMap(int initialCapacity, float loadFactor) {
2      if (initialCapacity < 0)
3          throw new IllegalArgumentException(
4                  "Illegal␣initial␣capacity:␣"
5                  + initialCapacity);
6      if (initialCapacity > MAXIMUM_CAPACITY)
7          initialCapacity = MAXIMUM_CAPACITY;
8      if (loadFactor <= 0 || Float.isNaN(loadFactor))
9          throw new IllegalArgumentException(
10                 "Illegal␣load␣factor:␣"
11                 + loadFactor);
12     int capacity = 1;
13     while (capacity < initialCapacity)
14         capacity <<= 1;
15
16     this.loadFactor = loadFactor;
17     threshold = (int)(capacity * loadFactor);
18     table = new Entry[capacity];
19 }
```

Comentaris:

– Noteu que la capacitat amb què es crearà el vector serà la primera potència de 2 més gran o igual que `initialCapacity`. Recordeu que l'*OpenJDK* força a que la capacitat del vector que representa la taula de dispersió sigui una potència de 2.

– Noteu també que `capacity <<= 1` obté el mateix resultat que: `capacity = capacity * 2` (per què?) i com que `capacity` s'ha inicialitzat a 1, anirà prenent a cada volta del bucle valors potència de 2.

5.7.3 `HashMap.Entry<K, V>`

La classe `HashMap.Entry<K, V>` implementa la interfície `Map.Entry<K, V>` per a una taula de dispersió implementada en forma de dispersió oberta. Per tant modelitza un node que conté una parella *(clau, valor)*. Vegem el seu codi:

```
1  public class HashMap<K, V> ... {
2      ...
3
4      static class Entry<K, V> implements Map.Entry<K, V> {
5          final K key;
6          V value;
7          Entry<K, V> next;
8          final int hash;
9
10         Entry(int h, K k, V v, Entry<K, V> n) {
11             value = v;
```

```
12              next = n;
13              key = k;
14              hash = h;
15          }
16
17          public final K getKey() {
18              return key;
19          }
20
21          public final V getValue() {
22              return value;
23          }
24
25          public final V setValue(V newValue) {
26              V oldValue = value;
27              value = newValue;
28              return oldValue;
29          }
30          ...
31      }
32      ...
33  }
```

Comentaris:

- Com que `HashMap.Entry<K, V>` implementa la interfície `Map.Entry<K, V>` ha de contenir una parella *(clau,valor)*.

  Hem vist que `HashMap<K, V>` es representa com un vector d'`Entry<K, V>`. Això, doncs, vol dir que es representa com un vector de parelles *(clau, valor)*.

- A més a més, `HashMap<K, V>` implementa una taula de dispersió seguint l'estratègia del *dispersió oberta*. Per tant, `HashMap.Entry<K, V>` serà, en realitat, el primer d'una llista de nodes que contindran les claus sinònimes que la funció de dispersió envia a l'índex del vector `table` que conté aquest node.

  Aquesta llista de claus sinònimes la representem amb enllaços i l'atribut `next` de `HashMap.Entry<K, V>` apunta al següent sinònim de la llista (vegeu 5.5 per recordar l'estratègia de dispersió oberta).

- L'atribut `hash` conté el valor de la funció de dispersió per a la clau continguda al node.

5.7.4 La funció de dispersió: `hashCode()`

Com s'ho fa el Java per associar un valor de dispersió a una clau?

Com que qualsevol classe d'objectes pot actuar com a clau, el Java proposa associar a cada classe una operació:

```
1 public int hashCode()
```

que retorni el valor de dispersió associat a aquell objecte. Aquesta operació està definida a la classe `Object` i és heretada per qualsevol classe.

Fantàstic!, potser haureu pensat: el Java ja ens dóna una funció de dispersió que s'anomena `hashCode()` i que qualsevol classe hereta; per tant, no ens hem de preocupar per aquest tema!!! Malauradament, les coses no són tan simples. Expliquem-ho una mica.

L'operació `hashCode()`, segons l'especificació de l'API del Java, retorna un valor de tipus `int` que compleix el següent contracte:

Contracte de l'operació `hashCode()`:

- Si `hashCode()` s'invoca sobre el mateix objecte més d'un cop durant una execució d'una aplicació, `hashCode()` ha de retornar el mateix valor (sempre que no s'hagi modificat cap informació de l'objecte que afecti el resultat de l'operació `equals()`).

  Si es modifica una informació de l'objecte que afecti el resultat d'`equals()` o bé es crida `hashCode()` en una altra execució de l'aplicació, el seu resultat pot canviar.

- Si `o1.equals(o2)` retorna cert, aleshores `o1.hashCode() == o2.hashCode()` ha de ser també cert.

  O sigui, `hashCode()` retorna el mateix valor aplicat sobre dos objectes que són iguals segons `equals()`.

  Noteu que això implica que si es canvia el sentit d'`equals()`*, probablement, també caldrà canviar la implementació de* `hashCode()`.

- **No es requereix** que si `o1.equals(o2)` retorna fals, `o1.hashCode()` sigui diferent d'`o2.hashCode()`.

  Per tant, dos objectes diferents `o1` i `o2` (`o1.equals(o2) == false`) poden retornar el mateix valor per a `hashCode()`: `o1.hashCode() == o2.hashCode()`.

  De tota manera, si aquesta situació s'evita, es pot millorar el rendiment de les taules de dispersió.

Una implementació possible per a l'operació `hashCode()`, d'acord amb aquest contracte, és la que el Java proposa a la classe `Object`:

Implementació de `Object.hashCode()`**:**

`o.hashCode()` *retorna l'adreça de* `o` *en hexadecimal*.

Aquesta implementació compleix, efectivament, el contracte especificat atès que `Object.equals()` considera iguals dos objectes si i només si són el mateix:

```
1  public boolean equals(Object obj) {
2      return (this == obj);
3  }
```

Sisplau, comproveu que, efectivament, la implementació que el Java fa de `Object.hashCode()` compleix el contracte.

Però aquesta implementació no és satisfactòria per ser usada en una taula de dispersió. Podeu pensar per què abans de continuar llegint?

Considerem el següent exemple:

Exemple 5.9: Un* `hashCode()` *poc apropiat

Considerem la següent classe `Person`:

Llistat 5.2: `Person`

```
1  public class Person{
2      private String nif;
3      private String name;
4      private int age;
5      ...
6  }
7
8
9  Map<Person, Address> m =
10         new HashMap<Person, Address>();
11
12 Person p1 = new Person("12345678A", "Josep", 34);
13 Person p2 = new Person("12345678A", "Josep", 34);
14
15 m.put(p1, new Address("Paeria", 9));
16
17 Address a = m.get(p2);
```

Segurament, voldrem que `m.get(p2)` retorni *Paeria, 9* com a adreça del ciutadà amb nif "12345678A". Però possiblement no serà així perquè `p1.hashCode() != p2.hashCode()`. Per què?

Si `p1.hashCode() != p2.hashCode()`, a la secció següent (5.7.5) veurem que segurament `p1` i `p2` seran aplicats a dos índexs diferents del vector i, per tant, la cerca de `p2` no trobarà `p1`.

Encara més, a la secció 5.7.6 descobrirem que, fins i tot si `p1` i `p2` fossin aplicats al mateix índex, la cerca de `p2` tampoc no trobaria `p1`!

■ ■ ■

Per evitar aquest problema, en general, cal redefinir `equals()` i `hashCode()`.

- *La redefinició d'*`equals()` retornarà cert quan dos objectes siguin conceptualment

iguals.

Exemple 5.10: `Person.equals()`

Dos objectes `Person` són conceptualment iguals si tenen iguals tots els seus atributs.

```
1 @Override
2 public boolean equals(Object obj){
3     return (obj instanceof Person)
4             && this.nif.equals(((Person) obj).nif)
5             && this.name.equals(((Person) obj).name)
6             && this.age == ((Person) obj).age;
7 }
```

Noteu que també ens hem d'assegurar que el paràmetre `obj` sigui del tipus `Person`.
■ ■ ■

- *La redefinició de* `hashCode()` retornarà un `int` que complirà el contracte d'aquesta operació. En essència:

  (1) Quan es cridi a `hashCode()` repetidament sobre el mateix objecte, retornarà el mateix valor.

  (2) Quan es cridi sobre dos objectes `o1, o2` de manera que `o1.equals(o2)` retornarà el mateix valor.

  A més a més, *intentarem* que compleixi dues propietats més que, si bé no són estrictament necessàries, faran `hashCode()` més robust i els `HashMaps`, més eficients:

  (3) Quan es cridi `hashCode()` sobre dos objectes `o1,o2` de manera que `o1.equals(o2)` sigui **fals**, retornarà un valor diferent.
  Aquesta propietat és la més delicada. Si l'assolim (o ens hi aproximem) i la taula està ben dimensionada, probablement tindrà pocs sinònims.

  (4) Quan es cridi `hashCode()` sobre el mateix objecte *en diferents execucions de l'aplicació*, retornarà el mateix valor.
  Aquesta propietat ens serà necessària si volem implementar taules de hash en fitxers (vegeu 7.4).

Exemple 5.11: `Person.hashCode()`

Us proposo una implementació per `Person.hashCode()`:

```
1  @Override
2  public int hashCode(){
3      int result = 23;
4      result = 31*result + age;
5      result = 31*result + nif.hashCode();
6      result = 31*result + name.hashCode();
7
8      return result;
9  }
```

Comproveu que aquest codi efectivament complirà les condicions (1), (2) i (4) anteriors. Què podem dir respecte del compliment de la (3)?

■ ■ ■

Al codi de `hashCode()` que hem proposat a l'exemple anterior hi podem identificar alguns dels elements que explicàvem a la secció 5.2 per a les funcions de dispersió que anomenàvem h_1. Per implementar-la, hem seguit la recepta que [JB08] proposa per tal d'implementar l'operació `hashCode()`. La reproduïm seguidament:

Recepta d'implementació de `hashCode()` proposada per [JB08]:

1. Guardeu algun valor constant diferent de zero, diguem 17, en una variable `int` anomenada resultat.
2. Per a cada espai f important per al vostre objecte (cada espai pres en compte pel mètode `equals`, que és), feu el següent:
   (a) Calculeu un codi hash `int` c per a l'espai:
      i. Si l'espai és un valor lògic, calculeu `(f ? 0 : 1)`.
      ii. Si l'espai és un `byte`, `char`, `short`, o `int`, calculeu `(int)f`.
      iii. Si l'espai és un `long`, calculeu `(int) (f ^ (f >>> 32))`.
      iv. Si l'espai és un `float`, calculeu `Float.floatToInt Bits(f)`.
      v. Si l'espai és un `double`, calculeu `Double.doubleToLongBits(f)`, i llavors feu `hash` al `long` resultant com al pas 2.a.iii. If the field is a double, compute `Double.doubleToLongBits(f)`, and then hash the resuting long as in step 2.a.iii.
      vi. Si l'espai és una referència objecte i el mètode `equals` d'aquesta classe compara l'espai mitjançant la invocació recursiva d'`equals`, recursivament invoca `hashCode` sobre l'espai. Si es requereix una comparació més complexa, calculeu una representació canònica per a aquest espai i invoqueu `hashCode` sobre la representació canònica. Si el valor d'aquest espai és nul, retorna zero (o alguna altra constant, però acostuma a ser zero).
      vii. Si l'espai és un vector, tracteu-lo com si cada element fos un espai separat. És a dir, calculeu un `hashCode` per a cada element significatiu aplicant aquestes regles recursivament, i combineu aquests valors com es descriu al pas 2.b.
   (b) Combineu el `hashCode` c calculat al pas a com segueix:
   ```
   result = 37*result + c;
   ```
3. Retorneu el resultat.
4. En acabar d'escriure el mètode `hashCode`, pregunteu-vos si les instàncies iguals tenen tenen codis hash iguals. Si no és així, esbrineu per què i corregiu el problema.

(Extret de [JB08], pàgines 47-48).

Però encara no hem acabat. La redefinició d'`equals()` i `hashCode()` no ho és tot. El valor que ens retorni `hashCode()` (que hauria de ser l'índex del vector on hem d'inserir la parella *(clau, valor)* pot ser molt més gran que la capacitat d'aquell vector. Què fem, aleshores? Ens caldrà restringir el valor retornat per `hashCode()` al rang de definició del nostre vector. Seguiu llegint, sisplau.

5.7.5 La funció de restricció d'un valor de dispersió a la longitud d'un vector: `indexFor()`

key.hashCode() retorna un `int` que pot ser més gran que la capacitat del vector. A la secció 5.2 hem proposat la funció `mod` (residu de la divisió entera) per restringir un enter a la capacitat del vector. L'operació `indexFor(h,length)` fa aquesta operació: o sigui, calcula el residu de la divisió de l'enter `h` entre `length`. Clarament, `length` és la capacitat del vector que representa la taula de dispersió i `h` és el valor retornat per `hashCode()`.

Conceptualment, podem pensar que `indexFor` està implementada d'aquesta manera:

```
1 static int indexFor(int h, int length) {
2     return h % length;
3 }
```

L'*OpenJDK* proposa una manera més eficient per implementar el residu de la divisió entera:

```
1 static int indexFor(int h, int length) {
2     return h & (length -1);
3 }
```

Veiem que aquesta implementació calcula també `h % length`.

Hem dit a 5.7.1 que la capacitat del vector que representa una taula de dispersió (`length`) ha de ser una potència de 2.

Suposem que `length` $= 2^r$. Aleshores:

`length` $-1 = 2^r - 1 = \overbrace{0\cdots0}^{32-r}\overbrace{1\cdots1}^{r}$

i `h & (length -1)` estarà format pels darrers r bits d'`h`.

Els darrers r bits de `h` constitueixen precisament el residu de la divisió entre un natural qualsevol i la potència de dos `length`.

Exemple 5.12:

Partim que `length` $= 32 = 2^5$.

Suposem que tenim una certa clau `key` sobre la que apliquem `key.hashCode()` i el resultat és 151. Clarament, $151 > 32 - 1$, per tant cal restringir 151 al rang $0 \cdots 31$.

Cridem a `indexFor(151,32)` i obtenim:

$$
\begin{array}{lll}
151_2 = & 00000000000000000000000010010111 & \\
(32-1)_2 = & 00000000000000000000000000011111 & \\
151_2 \& 31_2 = & 00000000000000000000000000010111 = & 23
\end{array}
$$

Observeu que $151\%32 = 23$ i que $10111 = 23$ són els darrers 5 bits de 151_2.

I, per tant, l'operació `get(key)` cercaria la clau `key` a l'índex 23 del vector. Per cert, a la propera secció podem veure com s'ho fa exactament l'operació `get()` per trobar una clau.

■ ■ ■

Si heu llegit la secció 5.2 ara podreu entendre que `hashCode()` fa el paper que en aquella secció feia la funció h_1 i `indexFor(...)` fa el paper d'h_2:

- h_1 intentava, com `hashCode()` aleatoritzar al màxim possible les claus (desitjablement, fent que si $k_1 \neq k_2$, aleshores $h_1(k_1) \neq h_1(k_2)$). h_1 no es preocupava del rang que prenguessin els valors retornats.

  I podeu notar també les semblances entre el procés de càlcul que hem proposat ([JB08]) per a `hashCode()` i el que mostràvem a la secció 5.2 per a h_1.

- h_2 feia, com `indexFor(...)`, la restricció a $0 \cdots$ `length` $- 1$

  A la secció 5.2, usàvem l'operador `mod` per fer la restricció i aquí, aprofitant que la longitud és sempre una potència de 2, un `AND` amb una màscara $1 \cdots 1$.

En definitiva, si a 5.2 dèiem que $i = h(k) = h_2(h_1(k)$ (on i és l'índex del vector on cal inserir k), a la implementació de l'*OpenJDK* trobem, equivalentment:
`i = indexFor(k.hashCode(), table.length)`
on `i` és també l'índex de del vector `table` on cal inserir `k`.

Us mostrem seguidament una versió una mica simplificada de les operacions `get` i `put` que proposa l'*OpenJDK*:

5.7.6 `get(key)`

```
1  public V get(Object key) {
2      int hash =key.hashCode();
3      for (Entry<K, V> e = table[indexFor(hash, table.length)];
4           e != null;
5           e = e.next) {
6
7          Object k;
8          if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
9              return e.value;
10     }
11     return null;
12 }
```

Comentaris:

- El bucle `for` recorre la llista de sinònims que està situada a l'índex `indexFor(key.hashCode(), table.length)` del vector `table`.

  Recordem que cada node d'aquesta llista és de classe `Entry<K, V>` i, per tant, té una referència `next` al proper node de la llista (i.e., al proper sinònim).

- Si en aquest procés de recorregut de la llista de sinònims trobem un node amb clau igual a la clau cercada (`key`) retornem el valor associat a ella:

```
1          if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
2              return e.value;
```

5.7.7 `put(key, value)`

```
1  public V put(K key, V value) {
2      int hash = key.hashCode();
3      int i = indexFor(hash, table.length);
4      for (Entry<K, V> e = table[i]; e != null; e = e.next) {
5          Object k;
6          if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
7              V oldValue = e.value;
8              e.value = value;
9              return oldValue;
10         }
11     }
12     addEntry(hash, key, value, i);
13     return null;
14 }
15
16 void addEntry(int hash, K key, V value, int bucketIndex) {
17     Entry<K, V> e = table[bucketIndex];
18     table[bucketIndex] = new Entry<K, V>(hash, key, value, e);
19     if (size++ >= threshold)
20         resize(2 * table.length);
21 }
```

Comentaris:

- `put(k,v)` cerca, en primer lloc, la clau `key` al vector. Ho fa seguint la mateixa estratègia que a l'operació `get` que acabem de presentar. Si la troba substitueix el vell valor associat al node que contenia la parella de clau `key` pel valor `value` passat per paràmetre a l'operació. I retorna (d'acord amb l'especificació de `put`) el valor vell (el que ha estat substituït).
- Si no ha trobat la clau al vector, la incorpora mitjançant una crida a `addEntry(key,value,bucketIndex)`, la qual s'encarregarà d'afegir la parella (`key`, `value`) a l'índex `bucketIndex` del vector.
- Noteu a la línia 18 que el nou node (`new Entry<K, V>(hash,key,value,e)`) que contindrà la nova parella s'insereix a la primera posició de la llista de sinònims de l'índex `bucketIndex`. Efectivament, `table[bucketIndex]` es refereix directament a aquest nou node i l'atribut `next` d'aquest nou node es referirà al que fins ara era el primer node de la llista de sinònims de `table[bucketIndex]`. Com sabem això darrer?

- Finalment, si el nombre de parelles *(clau,valor)* supera un determinat llindar (`threshold`) es redimensiona la taula, multiplicant per 2 el nombre d'índexs (recordeu que la mida d'una taula sempre havia de ser una potència de 2? Si no, vegeu 5.7.1).

  El codi de `resize` no es mostra aquí. Vegeu [OJDK] per als detalls.

5.7.8 Més operacions

La classe `HashMap<K, V>` ofereix moltes altres operacions, com ara `remove(k)` o les que permeten fer iteracions sobre els elements d'un `HashMap`. Per als detalls sobre la primera,

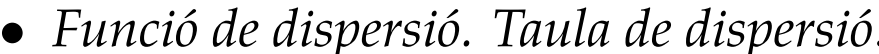
us proposo un cop d'ull a [OJDK].

Respecte de les operacions que permeten iteracions sobre una taula, les estudiarem quan tractem la segona gran família d'implementacions de taules: les implementacions arborescents. La iteració sobre una taula representada en forma arborescent té la gràcia que permet obtenir les claus ordenades, la qual cosa no podem fer en taules de dispersió. Per això té més sentit estudiar les iteracions sobre els elements d'una taula en aquell context. Concretament, ho farem a la secció 6.2.5.

Però no avancem esdeveniments. Abans d'iterar sobre els elements d'una taula representada arborescentment hem d'explicar com es poden usar arbres per representar taules. Comencem tan bon punt passeu pàgina.

5.8 Racó lingüístic

En aquesta secció comentem els termes tècnics usats en aquest capítol que no estan estandarditzats al diccionari de l'Institut d'Estudis Catalans, a Termcat o a [CM94].



- *Funció de dispersió. Taula de dispersió.*

  No hem pogut trobar a Termcat ni a [CM94] una manera estàndard de traduir els termes anglesos *hash function* i *hash map*. En criptografia i telecomunicacions s'usa el terme *funció resum* per denotar una funció que permet transformar un bloc de dades en una cadena de longitud fixa que representa aquell bloc. Aquest terme *funció resum* no és d'ús habitual en estructures de dades ni tampoc captura la propietat fonamental que ens interessa a l'hora d'implementar *hash maps* que consisteix a que la *hash function* dispersi les claus de la manera més aleatòria possible. Per aquestes dues raons no hem usat el terme *funció resum*. Triem, en canvi, *funció de dispersió* que, tot i no ser estàndard, sí que té un ús ampli per la comunitat d'enginyeria de programari i estructures de dades.

  Igualment, traduïm *hash map* per *taula de dispersió* o, més acuradament, *taula associativa de dispersió*.

- *Dispersió tancada. Adreçament obert*.

  Els escollim com a traduccions més naturals dels termes anglesos *closed hashing* i el seu sinònim *open addressing*.

  Ambdós termes no estan estandarditzats ni a Termcat ni tampoc a [CM94].

- *Dispersió oberta. Dispersió enllaçada*.

  Els escollim com a traduccions més naturals dels termes anglesos *open hashing* i el seu sinònim *chained hashing*. Seguint el criteri que ja hem esmentat al capítol 2, preferim *enllaçat* que *encadenat*.

  Ambdós termes no estan estandarditzats ni a Termcat ni tampoc a [CM94].

- *Apinyament primari. Apinyament secundari.*

  Els escollim com a traduccions més naturals dels termes anglesos *primary clustering* i *secondary clustering*.

  Ambdós termes no estan estandarditzats ni a Termcat ni tampoc a [CM94].

Capítol 6

Taules implementades amb arbres

Encara que les taules se solen implementar en forma de *taules de dispersió*, no sempre es fa així. La raó més important ja l'hem dita al capítol anterior: les *taules de dispersió* no permeten obtenir els seus elements ordenats per ordre de clau. Existeix una altra família de representacions de taules, les arborescents, que sí que ho permeten. I el millor de tot és que el preu que hem de pagar per obtenir aquesta propietat addicional no és gaire elevat. Els algorismes de consulta, inserció i eliminació passen d'O(1) a O(log n) que, en la gran majoria d'usos de les taules, resulta absolutament irrellevant. Bona part de les representacions arborescents de taules són evolucions dels anomenats *arbres binaris de cerca (ABC)*, així que començarem el capítol presentant-los i proposant-ne una implementació integrada en la JCF. També aprendrem que els ABC tenen algunes dificultats que fan que habitualment s'emprin millores d'ells com a implementació de les taules. Presentarem una d'aquestes millores: els arbres B, que s'usen àmpliament per accedir eficientment a informació en una base de dades, però d'això ens n'ocuparem al capítol 7.

6.1 Arbres per fer taules

Els arbres es poden usar com a manera alternativa d'implementar les taules. La idea d'aquestes implementacions arborescents és la següent (recordeu la secció 3.1.5):

> Els arbres equilibrats tenen un nombre de nivells proporcional al logaritme del seu nombre de nodes.

Basant-nos en aquest fet:

- Representarem una taula en forma d'arbre equilibrat de manera que cada node de l'arbre correspongui a una parella (*clau*, *valor*)[1] de la taula.

[1]Sovint, (*clau*, *valor*) ho abreujarem com a (*k*,*v*)

- Dissenyarem algorismes de consulta que permeten baixar un nivell de l'arbre a cada pas.

  El cost d'aquests algorismes de consulta serà O(log n), essent n el nombre de parelles (k,v) de la taula (i.e., el nombre de nodes de l'arbre). Per tant, obtindrem un algorisme eficient de cerca d'una clau de la taula.

El cost O(log n) de la cerca no sembla un argument suficient per representar una taula de manera arborescent. Al cap i a la fi, la representació amb taules de dispersió assolia una eficiència O(1).

> Veurem que el gran avantatge de les representacions arborescents de taules respecte de les basades en dispersió és que les representacions arborescents permeten obtenir les claus de la taula de manera ordenada.

Les implementacions arborescents més típiques de les taules inclouen: **arbres binaris de cerca, AVL, Arbres B, arbres B+, arbres 2-3, arbres roig-negre....** De fet, totes aquestes famílies d'implementacions són millores de la primera: *els arbres binaris de cerca (ABC)*. En les seccions següents els estudiarem. Més endavant, a partir de la seccó 6.3, presentarem els arbres B.

6.2 Arbres implementats com a ABC

6.2.1 Idea dels ABC

Per assolir el cost O(log n) en les consultes per clau, hem d'aconseguir que l'algorisme que cerqui una clau a l'arbre baixi un nivell a cada pas. Com assoleixen això els ABC? Doncs assegurant-se que *en tot moment es compleixi que les claus emmagatzemades al fill esquerre d'un ABC siguin menors que la clau de l'arrel de l'ABC i aquesta, a la seva vegada, sigui més petita que totes les claus emmagatzemades al fill dret de l'arbre.*

Més formalment:

> Un arbre binari de cerca (ABC) és:
> (1) Un arbre binari buit o bé
>
> (2) un arbre binari que conté parelles *(clau, valor)* a cada node i de manera que:
>
> (a) La clau associada a l'arrel és més gran que totes les claus del seu subarbre fill esquerre.
>
> (b) La clau associada a l'arrel és més petita que totes les claus del seu subarbre fill dret.
>
> (c) Els subarbres fill esquerre i fill dret són arbres binaris de cerca.
>
> (d) No hi ha dos nodes amb la mateixa clau a l'ABC.

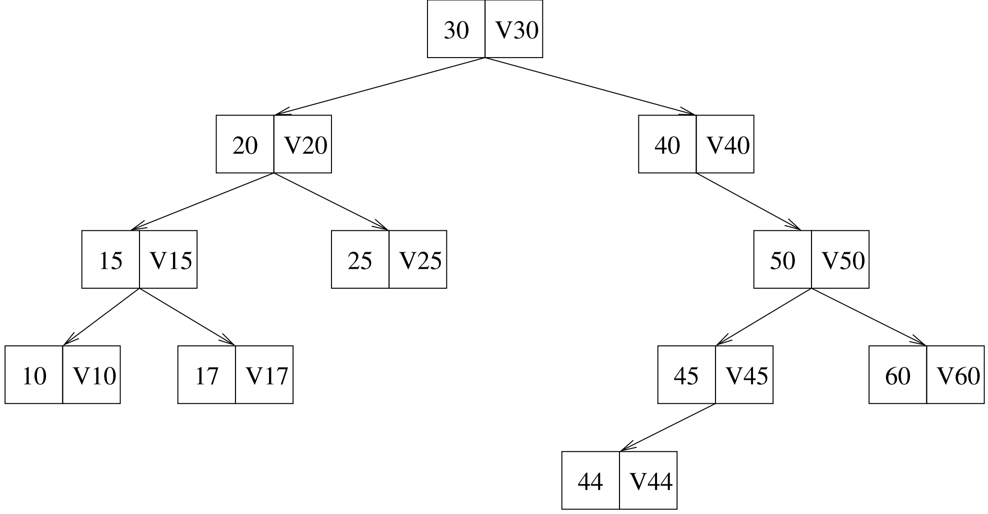


Figura 6.1: Un ABC

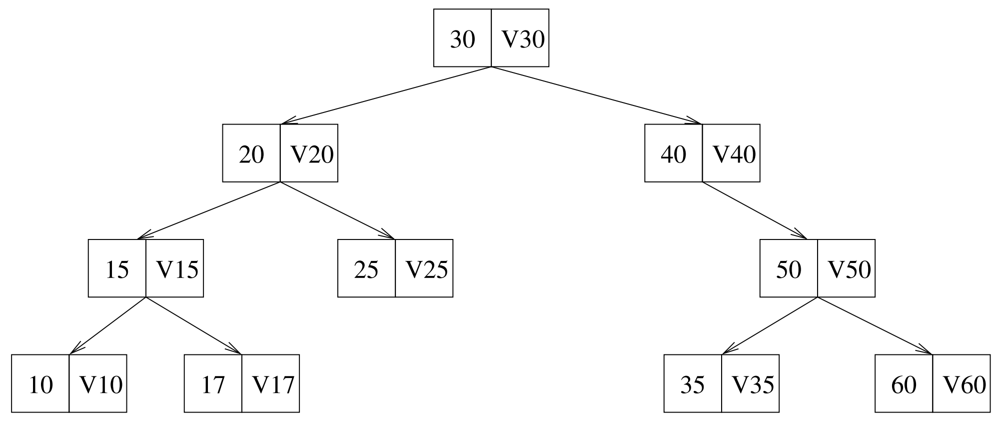


Figura 6.2: Atenció: no és un ABC

Exemple 6.1:

La figura 6.1 mostra un ABC amb claus enteres. Notem que cada node conté una parella (k,v). Per contra, l'arbre de la figura 6.2 no és un ABC. Per què?

■ ■ ■

6.2.2 Consulta, inserció i eliminació de claus en un ABC

Consulta

La distribució de claus en un ABC que es desprèn de la seva definició permet implementar cerques de claus que compleixin la condició de baixar un nivell a l'arbre en cada pas. Efectivament, això es pot fer en un ABC de la manera següent:

Esquema de l'algorisme de consulta en un ABC

- Comparem la clau de l'arrel de l'ABC amb la clau que cerquem.
  - Si la que cerquem és més petita, baixem a l'arrel del subarbre fill esquerre i continuem la cerca des d'allí.
  - Si és més gran, baixem a l'arrel del subarbre fill dret i continuem la cerca des d'allí.
  - Si és igual, ja hem trobat la clau.
  - Si ja no podem baixar més i no l'hem trobada, la clau no es troba a l'ABC.

Exemple 6.2:

A l'ABC de la figura 6.1 hi cerquem la clau `45`. Com que és més gran que l'arrel (`30`), baixem al fill dret, amb arrel `40`. Novament, `45` és més gran que `40` i tornem a baixar al seu fill dret (que té com a arrel, `50`). Ara, `45` és menor que `50` i, per tant, baixem al fill esquerre on trobem finalment la clau cercada. Retornarem el valor `V45` associat a la clau `45`.

A cada pas hem baixat un nivell a l'ABC. Com a màxim hem fet tants passos com nivells.

■ ■ ■

Les claus en taules implementades en forma d'ABC han de ser *comparables* dos a dos. No n'hi ha prou d'establir que una clau és diferent d'una altra (com passava amb les taules de dispersió). Hem de saber si una clau és més gran, més petita o igual que una altra. O sigui, l'espai de claus ha de formar un ordre total.

A la implementació que proposem de taules amb ABC, demanem que la classe que instancïi `K` (això és, la classe de la qual les claus són instància) implementi la interfície `Comparable`.

Tot seguit, proposem un algorisme recursiu que cerca el valor associat a la clau `k` a l'ABC `a`.

```
1 funcio cerca (a:ABC, k:Clau) retorna V {
2   si (a.arbreBuit()) retorna null;
3   si (a.arrel().clau() < k) retorna cerca(a.fillDret(),k);
4   si (a.arrel().clau() > k) retorna cerca(a.fillEsquerre(),k);
```

```
5     retorna a.arrel().valor();
6   }
```

I, a continuació, la versió iterativa d'aquest algorisme:

```
1   funcio cerca (a:ABC, k:Clau) retorna V {
2
3     mentre (! a.arbreBuit() && a.arrel() != k) {
4       si (a.arrel() < k) a = a.fillDret();
5       si no a = a.fillEsquerre();
6     }
7     si (a.arbreBuit() ) retorna null;
8     si no retorna a.arrel();
9   }
```

Comentaris:

- Noteu que l'algorisme suposa que les condicions del bucle (vegeu la línia 3) s'avaluaran d'esquerra a dreta i que si la primera retorna fals no s'avaluarà la segona (així s'avaluen les expressions en el Java, el C++ i la majoria dels llenguatges). Si no es fes aquesta hipòtesi (o si les condicions del bucle estiguessin posades a l'inrevés), l'algorisme seria incorrecte. Per què? Proposa un algorisme alternatiu que funcioni bé independentment de l'ordre de les condicions del bucle.

Inserció

Per inserir una parella (`k, v`) en una taula implementada com a ABC, cal aplicar, en primer lloc, l'algorisme de cerca per determinar on s'ha de situar la parella.

Esquema de l'algorisme d'inserció d'una parella (clau,valor) en un ABC

- Si la clau que es vol inserir ja és a la taula, simplement cal modificar el seu valor associat.
- Si la clau que es vol inserir no és a la taula, s'ha de:
  - Determinar el node `nod` de l'ABC allà on cal inserir la parella (`k,v`) com a fill esquerre o dret de `nod`.

    Necessàriament, `nod` serà una fulla o tindrà un únic fill. Per què?
  - Inserir la parella (`k,v`) com a fill de `nod`.

Exemple 6.3:

A l'ABC de la figura 6.1 hi inserim la parella (46, V46). Localitzem el node on cal inserir-la com a filla. Aquest node és el que té com a clau 45. Inserim (46, V46) com a fill dret del node que té com a clau 45.

■ ■ ■

Eliminació

En el procés d'eliminació d'una clau d'una taula implementada com a ABC es poden produir quatre situacions: (1) la clau que es vol eliminar no és a l'ABC, (2) la clau es troba en una fulla, (3) la clau es troba en un element terminal i (4) la clau es troba en un node intermedi no terminal. Els casos (1), (2) i (3) són senzills. El (4) requereix una mica més d'atenció:

Esquema de l'algorisme d'eliminació d'una parella (clau,valor) en un ABC

- Cercar el node `nod` de l'ABC on hi ha la clau que es vol eliminar. Si la clau no es troba a l'ABC, no cal fer res.
- Si `nod` és una fulla, eliminar-la.
- Si `nod` és un node terminal (només té un fill no nul), considerar l'únic fill de `nod` (`f`) i el node pare de `nod` (`p`). Substituir el fill `nod` de `p` per `f`.
- Si `nod` és un node intermig no terminal ni fulla (té dos fills no nuls), substituir la parella (`k,v`) de `nod` per la parella (`k1,v1`) immediatament anterior (o següent) en l'ordre de les claus. Aquesta parella ocuparà el node `nod1` que serà *el node terminal dret del fill esquerre de* `nod` (o *el node terminal esquerre del fill dret de* `nod`).

  `nod1` serà un node terminal que contindrà la mateixa clau que `nod`.
- Eliminar recursivament `nod1`.

Exemple 6.4:

De l'ABC de la figura 6.1 n'eliminem la clau 40. Com que ocupa un node terminal (només té un fill no nul: el fill dret 50), fem que aquest fill dret 50 sigui el fill dret del pare de 40: 30. Queda l'ABC de la figura 6.3.

De l'ABC de la figura 6.1 n'eliminem la clau 20. Com que ocupa un node intermedi no terminal ni fulla (té dos fills no nuls: 15 i 50), substituïm la clau 20 per 17 i, seguidament, eliminem 17. Queda l'ABC de la figura 6.4.

■ ■ ■

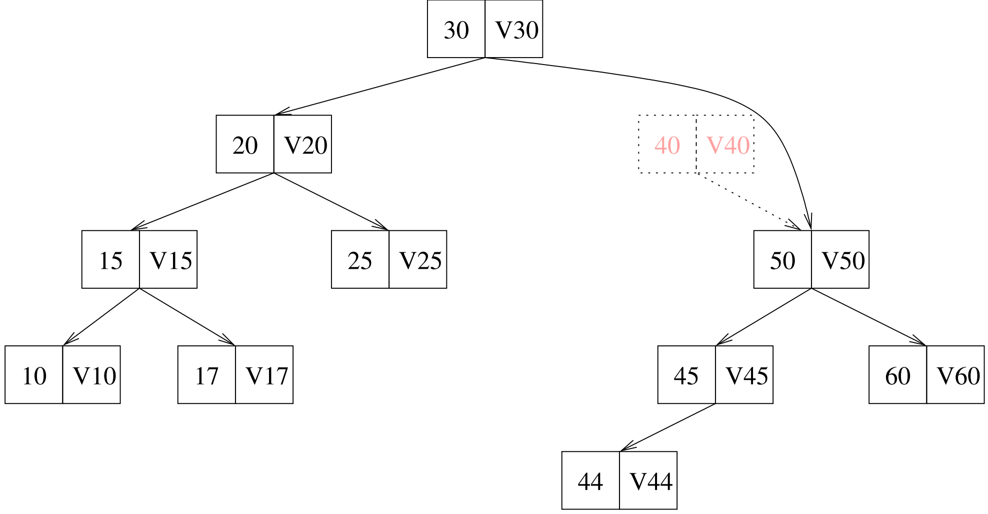


Figura 6.3: Eliminació de la clau 40

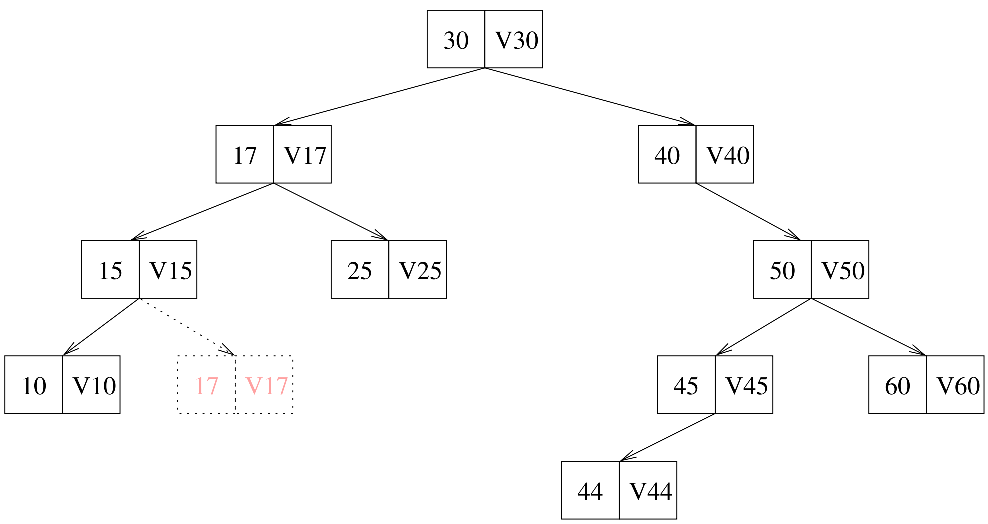


Figura 6.4: Eliminació de la clau 20

6.2.3 Cost dels algorismes d'inserció, eliminació i consulta en un ABC

Ara que coneixem la mecànica dels algorismes d'inserció, eliminació i consulta d'un ABC, ens podem preguntar pel seu cost. Aquest cost es dedueix dels fets següents:

1. Els algorismes baixen un nivell a l'ABC a cada pas.

2. Un pas dels algorismes correspon amb un nombre constant d'operacions.

3. El nombre de nivells d'un ABC equilibrat és proporcional al logaritme del nombre de nodes de l'ABC (vegeu 3.1.5).

Podem deduir d'aquests fets que el cost de les operacions d'inserció, consulta i eliminació de claus en un ABC és logarítmic? Penseu-hi un moment.

La resposta és no. Un no ben rotund. Pot passar, de fet, sovint passarà, que l'ABC que representi la taula *no estigui equilibrat*. En aquest cas, l'ABC podria degenerar tant que, de fet, esdevingués una llista.

Exemple 6.5:

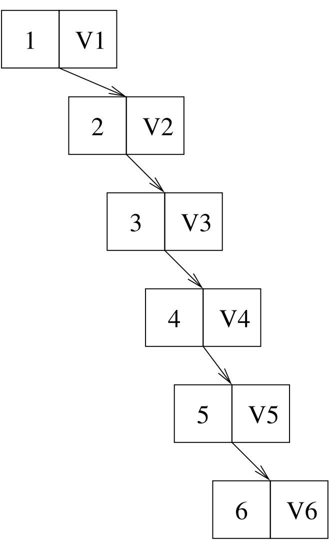


Figura 6.5: ABC degenerat

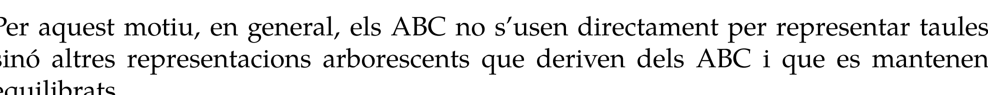
Inserim en una taula buida implementada com a ABC les claus següents i en aquest ordre: `1, 2, 3, 4, 5, 6`. L'ABC a què donaran lloc seguint l'algorisme d'inserció presentat més amunt és el de la figura 6.5.

Evidentment, la cerca d'una clau en un ABC com aquest té, en el cas pitjor, un cost O(n), on n és el nombre de claus presents a la taula.

■ ■ ■

- El cost en el cas pitjor dels algorismes de consulta, inserció i eliminació en una taula implementada com a ABC **equilibrat** és $O(\log_2 n)$, on n és el nombre de claus existents a la taula.

- Si no podem garantir que l'ABC sigui equilibrat, el cost en el cas pitjor dels algorismes d'inserció, consulta i eliminació en una taula implementada com a ABC és $O(n)$, on n és el nombre de claus existents a la taula.



Per aquest motiu, en general, els ABC no s'usen directament per representar taules sinó altres representacions arborescents que deriven dels ABC i que es mantenen equilibrats.
(La secció 6.3 explora una d'aquestes representacions.)

6.2.4 Obtenció de parelles (clau,valor) ordenades per clau

La secció anterior, la del cost, potser ha resultat una mica decebedora: Hem muntat tota aquesta enredada per obtenir què? Per obtenir una representació de les taules mitjançant ABC els quals:

- Si aconseguim que estiguin equilibrats, podrem arribar a un cost $O(\log n)$ per fer una cerca, que és bo però pitjor que el cost que obteníem amb la representació de les taules amb funcions de dispersió ($O(1)$). O sigui, el cas millor dels ABC és pitjor que la representació amb funcions de dispersió.
- A més a més, haurem d'aconseguir que els arbres estiguin equilibrats. Hi ha tècniques per aconseguir-ho (AVL, arbres roig-negre, arbres B...) però ja us podeu imaginar que aquestes tècniques introduiran més elements de complexitat a la programació dels arbres.

Ens estem, doncs, complicant la vida innecessàriament? Per quin motiu introduir representacions arborescents de taules si no milloren el cost de les que teníem i, a més, resulten més complexes de gestionar? Doncs perquè si es recorre un ABC usant un dels recorreguts que vam mostrar al capítol 3 s'obtenen les claus ordenades. Podeu pensar un moment en quin recorregut és aquest?

Propietat:
El recorregut en inordre d'un ABC obté les claus en ordre ascendent.

Justificació de la propietat. Justifiquem la propietat amb un raonament inductiu: si l'ABC tingués únicament un nivell, seria una fulla i, clarament, el seu recorregut en inordre fóra ordenat perquè només contindria una clau.

Suposem ara que la propietat és certa per a ABC de fins a $n-1$ nivells i veiem que també ho serà per ABC que tinguin n nivells. Prenem un ABC a d'n nivells i recorrem-lo en inordre:

- En primer lloc, recorrerem (també en inordre) el seu subarbre fill esquerre. Però aquest subarbre fill esquerre té menys d'n nivells i, per tant, usant la hipòtesi d'inducció, el seu recorregut en inordre mantindrà l'ordre ascendent de les claus.
- Continuarem el recorregut en inordre de a recorrent l'arrel. La clau de l'arrel, per definició d'ABC, és més gran que totes les claus del subarbre fill esquerre; per tant, l'ordenació no es trencarà quan afegim la clau de l'arrel al final de totes les del fill esquerre.
- Finalment, recorrerem ara en inordre el fill dret de a. Com que el fill dret de a té menys de n nivells, el seu recorregut en inordre obtindrà una seqüència ordenada ascendentment de claus. Com que totes les claus del subarbre fill dret són més grans que la de l'arrel (per definició d'ABC) resulta que obtindrem finalment la seqüència de totes les claus de l'ABC ordenades.

Exemple 6.6:

Com a prova, podeu fer el recorregut en inordre de l'ABC de la figura 6.1.

■ ■ ■

Recepta:

Quan calgui obtenir les claus d'una taula ordenades, és preferible (en general) usar una representació arborescent de les taules.

Quan no calgui obtenir les claus d'una taula ordenades, és preferible (en general) usar una representació de les taules amb funcions de dispersió.

6.2.5 Una proposta d'implementació de taules amb ABC

En aquesta secció presentem una implementació arborescent de taules, en forma d'ABC. La JCF també proposa una implementació arborescent de les taules (la classe `TreeMap<K, V>`). Així i tot, els arbres usats a `TreeMap<K, V>` no són ABC sinó *arbres roig-negre*, que estan basats en els ABC però presenten una major eficiència perquè es mantenen equilibrats. Com que la immensa majoria de representacions arborescents de taules es basen en els ABC, nosaltres presentem aquí la seva implementació. A la secció 6.3 proposarem una altra representació arborescent de taules, els *arbres B* que milloren l'eficiència dels ABC.

La classe `BSTMap<K, V>`. Idea general

La classe que implementa les taules usant ABC l'anomenarem `BSTMap<K, V>`. Aquesta classe s'insereix a la jerarquia de classes i interfícies proposada per la JCF. En particular, implementa la interfície `SortedMap<K, V>` i és filla d'`AbstractMap<K, V>`. Aquesta darrera classe ens proporciona una infraestructura que fa que puguem implementar la interfície

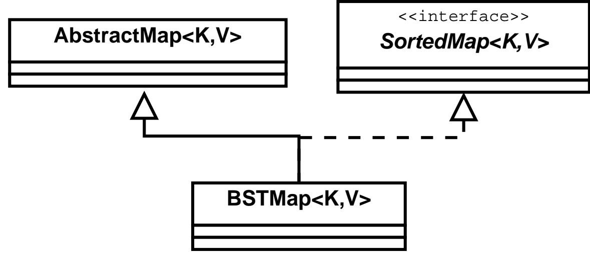


Figura 6.6: Jerarquia de classes i interfícies que afecta `BSTMap<K, V>`

`SortedMap<K, V>` sense haver d'implementar totes les seves operacions (per a algunes operacions aprofitarem, com ja vam fer amb la classe `HashMap<K, V>`, les implementacions que apareixen a `AbstractMap<K, V>`).

Així, imposem dos requeriments a la classe `BSTMap<K, V>`:

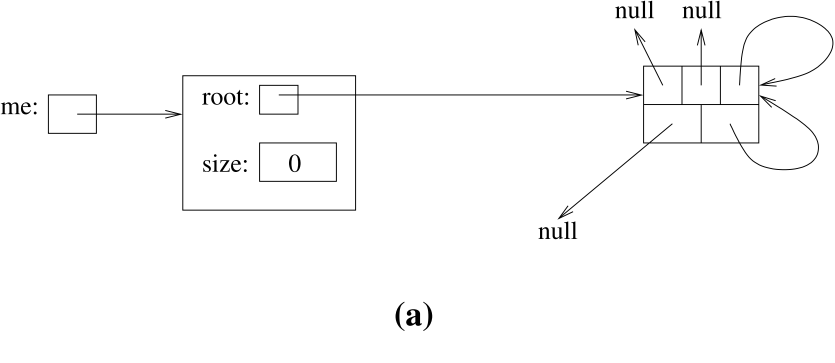
- Les classes que instanciïn `K` hauran d'implementar la interfície `Comparable`.
- La classe `BSTMap<K, V>` no admet claus nul·les.

La classe `BSTMap<K, V>`. Representació

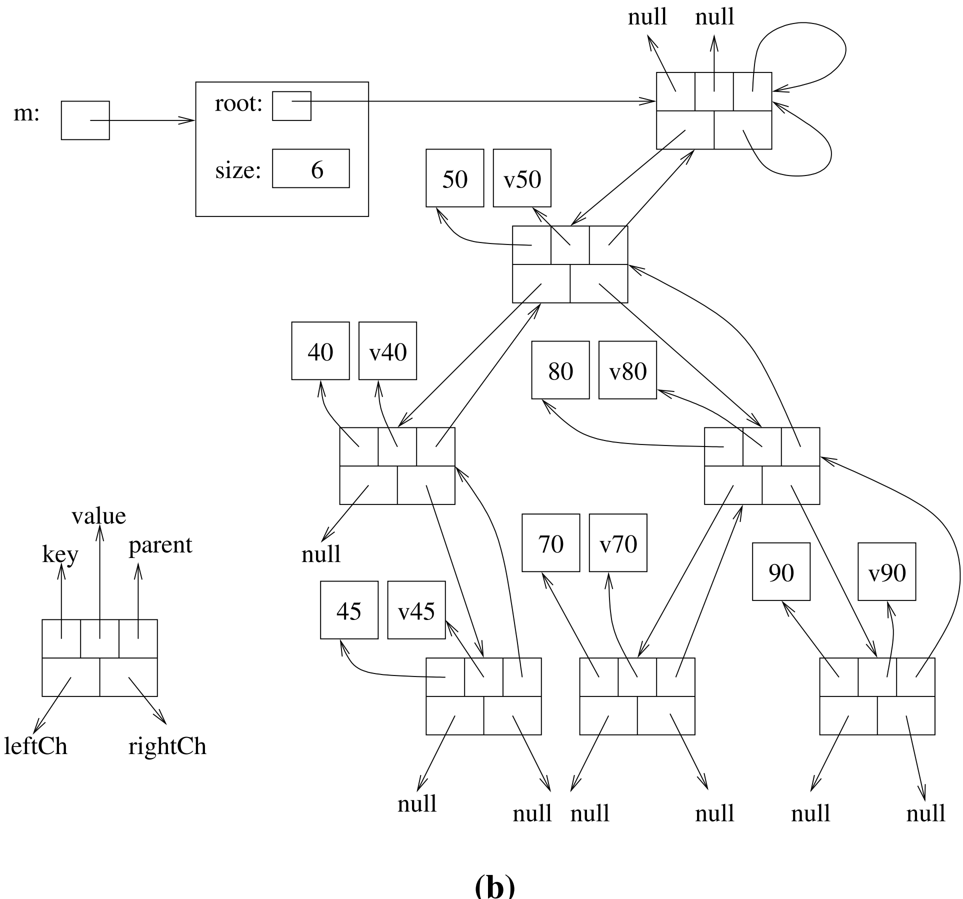
Representarem una taula com un arbre binari amb enllaços que contindrà un ABC. Cada node d'aquell arbre contindrà una parella (*clau, valor*). A més a més, per tal de facilitar les iteracions sobre els elements de la taula, afegirem a cada node un punter al seu pare. La figura 6.7 mostra la representació d'una taula buida (a) i d'una taula amb elements (b).

La implementació d'un arbre binari amb enllaços i punters al pare s'explica a les seccions 3.5 i 3.5.2.

El codi següent mostra la representació de la classe `BSTMap` i la creació d'una taula buida a l'operació constructora (d'acord amb la figura 6.7).



(a)



(b)

Figura 6.7: Representació d'una taula com a ABC amb enllaços i punter al pare

Llistat 6.1: La classe `BSTMap<K, V>`

```
1 public class BSTMap<K, V> extends AbstractMap<K, V>
```

```
2                             implements SortedMap<K, V> {
3     private int size;
4     private Entry<K, V> root;
5
6     public BSTMap() {
7         this.size = 0;
8         this.root = new Entry<K, V>();
9
10        this.root.leftCh = null;
11        this.root.rightCh = this.root;
12        this.root.parent = this.root;
13
14        this.root.key = null;
15        this.root.value = null;
16    }
17    ...
18 }
```

Comentaris:

- Notem que la construcció d'una taula buida, crea un node arrel, virtual, com es pot apreciar a l'operació constructora de `BSTMap<K, V>`.

Els nodes de l'ABC que representa la taula són de classe `Entry<K, V>`. La definició de la classe `Entry<K, V>` es fa com a classe aniuada estàtica, de manera similar a com s'ha fet en el cas de `HashMap<K, V>` i de `LinkedList<T>`:

Llistat 6.2: La classe `Entry<K, V>`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                             implements SortedMap<K, V> {
3     ...
4     private static class Entry<K, V> implements Map.Entry<K, V> {
5         K key;
6         V value;
7         Entry<K, V> leftCh;
8         Entry<K, V> rightCh;
9         Entry<K, V> parent;
10
11        Entry(K key, V value,
12              Entry<K, V> leftCh, Entry<K, V> rightCh,
13              Entry<K, V> parent) {
14            this.key = key;
15            this.value = value;
16            this.leftCh = leftCh;
17            this.rightCh = rightCh;
18            this.parent = parent;
19        }
20
21        public K getKey() { return key; }
22
23        public V getValue() { return value; }
```

```
24
25          public V setValue(V value) {
26              V oldValue = value;
27              this.value = value;
28              return oldValue;
29          }
30
31          public String toString() {
32              if (key == null || value == null) return "null";
33              return "Entry:␣key=" + key.toString()
34                      + "␣␣value=" + value.toString();
35          }
36      }
37      ...
38  }
```

La classe `BSTMap<K, V>`. `findKey(k)`

Les tres operacions més importants de `BSTMap<K, V>` que cal implementar són `get(k)`, `put(k,v)` i `remove(k)`. En totes tres, cal, en primer lloc, localitzar una clau: per retornar el seu valor associat, en el cas de `get`); per trobar el lloc on s'ha d'inserir (o modificar el seu valor associat), en el cas de `put`, i per eliminar-la, juntament amb el seu valor associat, en el cas de `remove`. Us proposo, en primer lloc, implementar una operació privada `findKey(k)` que retorni el node on es troba la clau `k`. Aquesta operació la cridarem des de les implementacions de `get/put/remove`.

Compte!: la clau `k` pot no trobar-se a l'arbre. En aquell cas, què retornarà `findKey(k)`? Podria retornar `null`, és clar, però això no ens ajudaria a implementar `put(k,v)`, la qual necessita obtenir el node de l'arbre del qual penjarà la nova parella (`k`,`v`). Què pot retornar, doncs, `findKey(k)` si `k` no pertany a la taula?

Ho heu endevinat: el node del qual penjarà (`k,v`) un cop s'hagi inserit. Aquest node serà una fulla o un element terminal, concretament, el darrer node visitat des de l'arrel. Vegem-ho:

`Entry<K, V> findKey(K key)` Si `key` és a l'ABC, retorna el node que conté la clau `key`. En cas contrari, retorna el pare del node que contindria `key` si aquesta clau fos present a l'ABC. El node retornat és una fulla o element terminal.
Llença: `IllegalArgumentException` si el paràmetre no és comparable.

Llistat 6.3: L'operació `findKey(k)`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                           implements SortedMap<K, V> {
3      ...
4      private Entry<K, V> findKey(Object key) {
5          try {
6              Comparable keyc = (Comparable) key;
7              Entry<K, V> current = this.root.leftCh;
8              Entry<K, V> parent = this.root;
9
10             while (current != null
```

```
11                      && !keyc.equals(current.key)) {
12                  parent = current;
13                  if (keyc.compareTo(current.key) < 0) {
14                      current = current.leftCh;
15                  } else {
16                      current = current.rightCh;
17                  }
18              }
19              if (current == null){
20                  return parent;
21              } else{
22                  return current;
23              }
24          } catch (ClassCastException e) {
25              throw new IllegalArgumentException();
26          }
27      }
28      ...
29  }
```

Comentaris:

- Notem que `findKey(k)` està definida com a operació privada. Efectivament, es tracta d'una operació de suport a la implementació d'altres operacions de `BSTMap<K, V>` que no oferirem als clients de la classe. A aquestes operacions, als llibres anglosaxons, sovint se les anomena *helpers*.
- Implementem la comparació entre la clau `key` i una clau qualsevol de l'ABC amb l'operació `compareTo(k)`. Aquesta operació ha de ser implementada per qualsevol classe que implementi `Comparable`. I, com hem dit abans, demanem que la classe de la qual les claus són instància, implementi aquesta interfície.

La classe `BSTMap<K, V>`. `get(k)`

La seva implementació és immediata a partir de les especificacions de `get(k)` i `findKey(k)` (vegeu l'especificació de `get(k)` a 4.2.1).

Llistat 6.4: L'operació `findKey(k)`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                             implements SortedMap<K, V> {
3      ...
4      @Override
5      public V get(Object key) {
6          Entry<K, V> entry =  findKey(key);
7          if (key.equals(entry.key)) {
8              return entry.value;
9          }
10         return null;
11     }
12     ...
13 }
```

Comentaris:

- Funcionaria aquest codi sobre una taula buida? Raoneu-ho.

La classe `BSTMap<K, V>.put(k,v)`

Proposem la implementació següent per a l'operació `put(k,v)`:

Llistat 6.5: L'operació `put(k,v)`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                              implements SortedMap<K, V> {
3      ...
4      @Override
5      public V put(K key, V value) {
6          V old = null;
7          try {
8              Entry<K, V> entry =  findKey(key);
9              if (key.equals(entry.key)) {
10                 old = entry.value;
11                 entry.value = value;
12             } else {
13                 Entry<K, V> newEntry =
14                     new Entry(key, value, null, null, entry);
15                 if (root == entry
16                     || ((Comparable) key).compareTo(entry.key) < 0) {
17                     entry.leftCh = newEntry;
18                 } else {
19                     entry.rightCh = newEntry;
20                 }
21                 size++;
22             }
23         } catch (Exception e) {
24             throw new IllegalArgumentException();
25         }
26         return old;
27     }
28     ...
29 }
```

Comentaris:

- Decidim, en primer lloc (vegeu la línia 9), si `key` era ja a la taula o no. Si hi era, sobreescrivim el seu valor associat (vegeu la línia 11).
- Si no hi era creem un nou node (`newEntry`) per encabir la nova parella (`key, value`) (vegeu la línia 13). En aquest cas, caldrà inserir aquest nou node com a fill dret o esquerre de `entry`.
- `newEntry` serà un fill esquerre d'`entry` si la taula és buida (o sigui, si `entry` és l'arrel virtual (`root == entry`) o bé si la clau d'`entry` és més gran que `key` (`((Comparable) key).compareTo(entry.key) < 0`)).

En cas contrari, `newEntry` serà un fill dret d'`entry`.

- Si `key` és `null`, la condició de la línia 9 llençarà `NullPointerException` com correspon a l'especificació detallada de l'operació `put(k,v)` en els casos en què la taula no admeti claus nul·les, com és el cas de `BSTMap<K, V>`.

- Què passaria si la taula a què s'insereix la nova clau és buida? Funcionaria l'operació `put`? Quina comparació es faria a la línia 9 i quin resultat donaria? Raoneu-ho.

La classe `BSTMap<K, V>.remove(k)`

Implementem `remove(k)` de manera recursiva seguint la idea mostrada a la pàgina 236:

Llistat 6.6: L'operació `remove(k)`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                               implements SortedMap<K, V> {
3      ...
4      @Override
5      public V remove(Object key) {
6          V old = null;
7          Entry<K, V> entry =  findKey(key);
8          if (key.equals(entry.key)){
9              old = entry.value;
10             removeRec(entry);
11         }
12         return old;
13     }
14
15     private void removeRec(Entry<K, V> entry) {
16         if (isLeaf(entry)) {
17             removeLeaf(entry);
18         } else if (isTerminal(entry)) {
19             removeTerminal(entry);
20         } else {
21             Entry<K, V> next = rightLeftMostEntry(entry);
22             entry.key = next.key;
23             entry.value = next.value;
24             removeRec(next);
25         }
26     }
27     ...
28 }
```

Comentaris:

- Si la clau a eliminar és en una fulla o element terminal (`entry`), l'eliminem fàcilment.

- Si és en un node intermedi, la substituïm per la següent clau de l'ABC en ordre (la més petita del seu subarbre fill dret: `next = rightLeftMostEntry(entry)`) i, aleshores, eliminem `next` recursivament.

- Us queda com a exercici la implementació de totes les operacions necessàries per completar `removeRec`. En particular, `removeLeaf`, `removeTerminal`, `rightLeftMostEntry`...

Iterant sobre un `BSTMap`

L'estratègia general d'implementació que proposem en aquesta secció dels iteradors sobre `BSTMap` està fortament basada en les estratègies d'iteració sobre `Map`s mostrades al projecte OpenJDK. No són exactament iguals perquè, com ja hem explicat, la JCF no conté la classe `BSTMap<K, V>` que en aquestes seccions implementem.

Si recordem el que vam comentar a la secció 4.2.1, sobre una instància `lis` de tipus `List<T>` s'hi pot iterar directament simplement obtenint un iterador sobre `lis`:

```
1  void f(List<T> lis) {
2
3      Iterator<T> it = lis.iterator();
4
5      while (it.hasNext()) { ... }
6      ...
7  }
```

Això és així perquè la interfície `List<T>` és subinterfície d'`Iterable<T>`. Però aquest no és el cas de `SortedMap<K, V>`. Per tant, és incorrecte fer:

```
1  void f(SortedMap<K, V> m) {
2      Iterator<T> it = m.iterator();   //INCORRECTE!!!!
3  }
```

Per iterar sobre elements de tipus `SortedMap<K, V>` (i, en particular, sobre `BSTMap<K, V>`) cal obtenir els elements de la taula sobre els qual volem iterar en forma de col·lecció iterable.

`SortedMap<K, V>` (de la mateixa manera que `Map<K, V>`) ens dóna tres operacions per fer això (vegeu 4.2.1):

- `Set<K> keySet()`. Ens retorna el conjunt de claus de la taula. Un conjunt és iterable i, per tant, a través d'ell, podem iterar sobre les claus de la taula.
- `Collection<V> values()`. Ens retorna una col·lecció amb tots els valors de la taula per iterar sobre ells. No retorna un conjunt perquè, a diferència de les claus, hi pot haver valors repetits a la taula.
- `Set<Map.Entry<K, V» entrySet()`. Retorna el conjunt de parelles (*clau*, *valor*) de la taula, per iterar sobre elles.

Vegem com podem dissenyar adequadament la infraestructura que necessitem per implementar aquestes tres operacions. Comencem per `keySet()`.

Suposem que `m` és un `BSTMap<K, V>`, per a unes classes específiques de claus (`K`) i de valors (`V`):

```
1 public class K ... { ... }
2 public class V { ... }
3
4 SortedMap<K, V> m = new BSTMap<K, V>();
```

En aquestes condicions, la crida

```
1 Set<K> c = m.keySet();
```

ha de retornar un conjunt `c` d'elements de tipus `K` de manera que, en fer

```
1 Iterator<K> it = c.iterator();
```

obtinguem un iterador `it` que permeti iterar sobre totes les claus de la taula `m`. Aquesta reflexió ens planteja dues qüestions:

1. De quina classe serà el conjunt `c` retornat per `m.keySet()`?
2. De quina classe serà l'iterador `it` retornat per `c.iterator()`?

Us proposo definir dues noves classes, una per al conjunt i una altra per a l'iterador:

- `KeySet`: classe que representa conjunts amb elements de tipus `K`. Clarament, aquesta classe haurà d'implementar la interfície `Set<T>`.

  `m.entrySet()` retornarà un objecte de classe `KeySet`.

- `KeyIterator`: Classe que representa iteradors que iteren sobre elements de tipus `K`. Aquesta classe haurà d'implementar la interfície `Iterator<T>`.

  `m.entrySet().iterator()` retornarà un objecte de classe `KeyIterator`.

Les classes `KeySet` i `KeyIterator` no les farem visibles als clients de `BSTMap<K, V>`. Al capdavall, a aquests clients els interessa que l'objecte retornat per `c = m.keySet()` es comporti com un conjunt d'elements de tipus `K` i que `c.iterator()` es comporti com un iterador sobre totes les claus de `m`.

O sigui, als clients de `BSTMap<K, V>` no els interessa el tipus específic de les classes retornades per `m.keySet()` i `c.iterator()`. **Només els interessa que implementin respectivament les interfícies `Set<T>` i `Iterator<T>`.**

Amb aquestes idees proposem el codi següent:

Llistat 6.7: L'operació `keySet()`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                            implements SortedMap<K, V> {
3      ...
4      @Override
5      public Set<K> keySet() {
6          return new KeySet();
7      }
8
9      private final class KeySet extends AbstractSet<K> {
10         public Iterator<K> iterator() {
11             return new KeyIterator();
12         }
13
14         public int size() {
15             return size;
16         }
17     }
18
19     private final class KeyIterator ... {
20         ...
21     }
22     ...
23 }
```

Comentaris:

- L'operació `keySet()` retorna un objecte de la classe `KeySet`. La definició d'aquesta classe es mostra a la línia 9: és una una subclasse d'`AbstractSet<K>`. O sigui, un conjunt de claus.
- Com ja hem vist amb altres classes (com `AbstractList` o `AbstractMap`), `AbstractSet` ens proporciona la infraestructura per implementar un conjunt molt fàcilment. Essencialment, només ens cal crear-ne una subclasse i implementar les operacions `size()` i `iterator()`. Aquesta darrera és la que retorna un iterador que té la capacitat de retornar totes les claus de la taula ordenades.
- A la línia 19 iniciem la definició de la classe `KeyIterator`. Per donar la definició completa encara ens calen algunes explicacions addicionals, així que deixem-la amb punts suspensius, de moment.
- Amb aquest codi, cada cop que cridem `keySet()` es genera un nou conjunt amb les claus de la taula. Com ho faríem per evitar la generació d'un nou conjunt cada cop que cridem `keySet()`?

Però la interfície `SortedMap<K, V>`, a més a més de `keySet()`, ens ofereix dues operacions més que ens retornen col·leccions que usarem per iterar sobre els elements de la taula: `values()` i `entrySet()`.

Els objectes retornats per `values()` o `entryIterator()` poden ser de tipus `KeySet`? Certament, no. `KeySet` representa un conjunt d'elements de tipus `K` però `values()` retorna un con-

junt amb elements de tipus `V` i `entrySet()`, un conjunt amb elements de tipus `Entry<K, V>`. Per tant, definirem dues noves classes:

- `Values`: col·lecció d'elements de tipus `V`. L'usarem per representar els objectes retornats per `values()`.
- `EntrySet`: conjunt d'elements de tipus `Entry<K, V>`. L'usarem per representar els objectes retornats per `entrySet()`.

Igualment, tampoc no podem usar la classe `KeyIterator` (que és un `Iterator<K>`) per representar els iteradors retornats per `values().iterator()` (de tipus `Iterator<V>`) i `entrySet().iterator()` (de tipus `Iterator<Entry<K, V>`). Definirem, doncs, dues classes més:

- `ValueIterator`: iterador sobre una col·lecció d'objectes de tipus `V`.
- `EntryIterator`: iterador sobre un conjunt d'elements de tipus `Entry<K, V>`.

El codi queda així:

Llistat 6.8: L'operació `keySet()`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                           implements SortedMap<K, V> {
3      ...
4      @Override
5      public Set<K> keySet() {
6          return new KeySet();
7      }
8
9      @Override
10     public Collection<V> values() {
11         return new Values();
12     }
13
14     @Override
15     public Set<Map.Entry<K, V>> entrySet() {
16         return new EntrySet();
17     }
18
19     private final class KeySet extends AbstractSet<K> {
20         public Iterator<K> iterator() {
21             return new KeyIterator();
22         }
23
24         public int size() {
25             return size;
26         }
27     }
28
29     private final class Values extends AbstractCollection<V> {
30         public Iterator<V> iterator() {
```

```
31            return new ValueIterator();
32        }
33
34        public int size() {
35            return size;
36        }
37    }
38
39    private final class EntrySet extends AbstractSet<Map.Entry<K, V>> {
40        public Iterator<Map.Entry<K, V>> iterator() {
41            return new EntryIterator();
42        }
43
44        public int size() {
45            return size;
46        }
47    }
48
49    private final class KeyIterator ... {
50        ...
51    }
52
53    private final class ValueIterator ... {
54        ...
55    }
56
57    private final class EntryIterator ... {
58        ...
59    }
60    ...
61 }
```

En aquest codi, els punts suspensius a les definicions de les classes `KeyIterator`, `ValueIterator` i `EntryIterator` ens indiquen que aquestes definicions no estan completes. Ens hi posem?

Per definir aquestes tres classes us proposo la reflexió següent:

Tots tres tipus d'iteradors: `KeyIterator`*,* `ValueIterator` *i* `EntryIterator`*, en realitat, iteraran sobre els elements d'un* `BSTMap<K, V>` *i obtindran els elements corresponents (claus, valors o parelles (clau, valor)) en el mateix ordre ascendent de clau.* Vegem-ne un exemple.

Exemple 6.7:

Si `m` és el `BSTMap<K, V>` mostrat a la figura 6.1.

- `it = m.keySet().iterator()` és un `KeyIterator` que en successives crides a `it.next()` obtindrà en aquest ordre: {10, 15, 17, 20, 25, 30, 40, 44, 45, 50, 60}.
- `it2 = m.values().iterator()` és un `ValueIterator` que en successives crides a `it.next()` obtindrà en aquest ordre: {v10, v15, v17, v20, v25, v30, v40, v44, v45, v50, v60}.
- Finalment, `it3 = m.entrySet().iterator()` és un `EntryIterator` que, en successives crides a `it.next()` obtindrà en aquest ordre: {(10,v10), (15,v15), (17,v17), (20,v20), (25,v25),

(30,v30), (40,v40), (44,v44), (45,v45), (50,v50), (60,v60)}.

Per tant, tenim que:

Totes tres classes d'iteradors comparteixen la mateixa estratègia d'iteració sobre un `BSTMap<K, V>`: *la de recórrer en inordre l'ABC que representa la taula*.
L'única diferència entre elles és sobre quin objecte específic es fa la iteració: `K`, en el cas de `KeyIterator`, `V`, en el cas de `ValueIterator` i `Entry<K, V>`, en el cas d'`EntryIterator`.

I, en qualsevol cas, `K` i `V` són parts d'`Entry<K, V>`.

I aquesta reflexió ens suggereix la idea següent:

Podem crear una classe `BSTIterator<T>` que implementi `Iterator<T>` i que sigui responsable de dissenyar l'estratègia d'iteració sobre un `BSTMap<K, V>` (i.e., de dissenyar el recorregut en inordre de l'ABC que representa la taula).

`KeyIterator`, `ValueIterator` i `EntryIterator` podrien ser subclasses seves.

Aquesta idea està implementada al codi següent:

Llistat 6.9: Esquelet de les classe `BSTIterator` `KeyIterator` `ValueIterator` i `EntryIterator`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                            implements SortedMap<K, V> {
3      ...
4      private abstract class BSTIterator<T> implements Iterator<T> {
5          //Representacio de la classe...
6
7          BSTIterator() { ... }
8
9          public final boolean hasNext() { ... }
10
11         public T next() { ... }
12
13         public void remove() { ... }
14     }
15
16     private final class KeyIterator extends BSTIterator<K> { ... }
17
18     private final class ValueIterator extends BSTIterator<V> { ... }
19
```

```
20      private final class EntryIterator
21                  extends BSTIterator<Map.Entry<K, V>> { ... }
22      ...
23  }
```

Ara, doncs, hem d'implementar `BSTIterator<T>` i les seves subclasses: `KeyIterator`, `ValueIterator` i `EntryIterator`. Tal com ho hem muntat tot, les subclasses basaran fortament el seu funcionament en el de la superclasse i només hi afegiran una petita matisació, la matisació de *sobre quins elements específics es fa la iteració*. Dit d'una altra manera: *què ha de retornar l'operació* `next()` *de cada subclasse?* (`KeyIterator.next()`, `ValueIterator.next()` i `EntryIterator.next()`.) Comencem implementant aquestes operacions.

Tal com hem vist, totes tres classes segueixen la mateixa estratègia d'iteració. Per tant, en tots tres casos `next()` obtindrà *el següent node en inordre de l'ABC que representa la taula i sobre el qual es fa la iteració*. `KeyIterator.next()` retornarà la clau d'aquell node; `ValueIterator.next()` en retornarà el valor, i, finalment, `EntryIterator.next()` retornarà la parella (*clau, valor*) d'aquell node.

Doncs us proposo que dotem `BSTIterator<T>` d'una operació (que anomenarem `nextEntry()`) que retornarà el node següent en inordre de l'ABC. `KeyIterator.next()` agafarà la clau del node retornat per `nextEntry()`. `ValueIterator.next()` n'agafarà el valor i `EntryIterator.next()`, la totalitat del node:

Llistat 6.10: Implementant `next()`

```
1   public class BSTMap<K, V> extends AbstractMap<K, V>
2                               implements SortedMap<K, V> {
3       ...
4       private final class EntryIterator
5                   extends BSTIterator<Map.Entry<K, V>> {
6           public Map.Entry<K, V> next() {
7               return nextEntry();
8           }
9       }
10
11      private final class ValueIterator extends BSTIterator<V> {
12          public V next() {
13              return nextEntry().value;
14          }
15      }
16
17      private final class KeyIterator extends BSTIterator<K> {
18          public K next() {
19              return nextEntry().key;
20          }
21      }
22      ...
23  }
```

I ja no ens queda més remei que llençar-nos a implementar `BSTIterator<T>`.

Iterant sobre un `BSTMap`. La classe `BSTIterator<T>`

Comencem recordant l'esquelet de la seva definició (que ja apareixia al llistat 6.9). A aquest esquelet hi afegim l'operació `nextEntry()` que acabem d'introduir i n'eliminem `next()`, que quedarà com a operació abstracta a `BSTIterator<T>`, perquè hem vist que era implementada a les seves subclasses:

Llistat 6.11: Esquelet de les classe `BSTIterator`

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                              implements SortedMap<K, V> {
3      ...
4      private abstract class BSTIterator<T> implements Iterator<T> {
5          //Representacio de la classe...
6
7          BSTIterator() { ... }
8
9          public final boolean hasNext() { ... }
10
11         public void remove() { ... }
12
13         final Entry<K, V> nextEntry() { ... }
14     }
15     ...
16 }
```

Per implementar `BSTIterator<T>` seguirem les idees d'iteració sobre arbres enllaçats amb punter al pare mostrades a la secció 3.5.2 del capítol 3. Proposarem una implementació bàsica que contindrà únicament l'operació constructora, `hasNext()` i `nextEntry()`. Fem algunes consideracions per implementar aquesta classe:

- Recordem de la secció anterior que `nextEntry()` retorna la següent parella (*clau, valor*) en l'ordre d'iteració.
- No caldrà implementar `BSTIterator<T>.next()` perquè aquesta operació ja està implementada convenientment en les tres subclasses de `BSTIterator<T>`: `EntryIterator`, `KeyIterator`, `ValueIterator`.
- Possiblement, l'avantatge més important de les implementacions arborescents de les taules és la possibilitat de fer iteracions per ordre de clau. Per aquest motiu, voldrem que la nostra implementació de `BSTIterator<T>` permeti iterar per a la col·lecció de parelles (*clau, valor*) en ordre ascendent de clau.

Vegem la implementació de la classe `BSTIterator<T>` que proposem:

Llistat 6.12: Tres classes d'iteradors

```
1  public class BSTMap<K, V> extends AbstractMap<K, V>
2                              implements SortedMap<K, V> {
3      ...
4      private abstract class BSTIterator<T> implements Iterator<T> {
5          Entry<K, V> next;         // propera entrada per a retornar
```

```
6
7           BSTIterator() {
8               next = root;
9               if (size > 0) { // car a la primera entrada
10                  while (next.leftCh != null) {
11                      next = next.leftCh;
12                  }
13              }
14          }
15
16          public final boolean hasNext() {
17              return next != root;
18          }
19
20          final Entry<K, V> nextEntry() {
21              if (next == root) throw new NoSuchElementException();
22
23              Entry<K, V> lastReturned = next;
24
25              if (next.rightCh != null) {
26                  next = next.rightCh;
27                  next = leftMostEntry(next);
28              } else {
29                  while (isRightChild(next)) {
30                      next = next.parent;
31                  }
32                  next = next.parent;
33              }
34              return lastReturned;
35          }
36
37          public void remove() {
38              throw new java.lang.UnsupportedOperationException();
39          }
40      }
41      ...
42  }
```

Comentaris:

- L'atribut `next` contindrà en tot moment la propera parella (*clau*, *valor*) que retornarà l'iterador. Aquest és l'invariant de la representació de la classe.

- Per aquest motiu, l'operació constructora situa `next` al primer en inordre (i.e., la parella amb clau menor de la taula com es discuteix a 6.2.4). Recordem que el primer en inordre és el terminal esquerre de l'ABC.

- L'operació `nextEntry()` retorna l'`entry` (node) referit per `next` a l'inici de l'operació i fa avançar `next` al següent `entry` en inordre. Tal com s'ha discutit a 6.2.4, el següent en inordre correspon a la següent parella en ordre ascendent de claus.

- El criteri per avançar fins al següent en inordre és aquest:

  - Si `next` té fill dret, el següent en inordre de `next` serà el terminal esquerre del seu fill dret (i.e., el primer en inordre del seu fill dret).
  - Si `next` no té fill dret, el següent en inordre de `next` serà el pare del primer node que sigui fill esquerre en el camí ascendent des de `next` fins a l'arrel virtual.

    Feu unes proves sobre un ABC per convèncer-vos d'això.

- Aplicant aquest criteri, el següent del darrer node en inordre de l'ABC serà l'arrel virtual (`root`) (per què?) i això ens dóna un criteri per implementar l'operació `hasNext()`:

  Haurem arribat al final del recorregut (i, per tant, `hasNext()==false`) si `next == root`.

- `remove()` és una operació optativa de la interfície `Iterator<T>` i la implementació que proposem no la suporta. (Exercici: proveu d'implementar-la.)

6.3 Taules implementades com a arbres B

Els arbres binaris de cerca (ABC) proporcionen una estratègia per implementar una taula amb la possibilitat de fer recorreguts ordenats per clau. Però ja hem vist que pateixen un problema molt important: els ABC no tenen per què ser complets ni equilibrats, la qual cosa compromet seriosament la seva eficiència. Per resoldre aquest problema sorgeixen diferents estratègies basades en la idea dels ABC però que en milloren el rendiment: arbres *AVL*, *roig-negre*, *B* (i les seves variants).

En aquesta secció presentarem els arbres B.

Els arbres B milloren els ABC essencialment en dos aspectes:

- Es mantenen automàticament equilibrats i això permet assolir sempre un cost logarítmic respecte del nombre de nodes de la taula.

- En general, un node té més de dos fills. Aquest *factor de ramificació* (el nombre màxim de fills que té cada node) superior al dels arbres binaris de cerca permeten una eficiència superior.

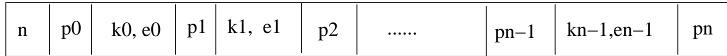
  Efectivament, amb un factor de ramificació més gran, el nombre de nivells que es necessiten per encabir n nodes és més petit. Per exemple, necessitem un arbre binari complet de 10 nivells per encabir unes 1000 claus, però en tenim prou amb un arbre 4-ari complet de 6 nivells per encabir-ne més de 1300.

En les properes seccions presentarem els arbres B. Comencem definint *arbres m-aris de cerca*.

6.3.1 Arbres m-aris de cerca

Un *arbre m-ari de cerca* és un arbre m-ari en què:

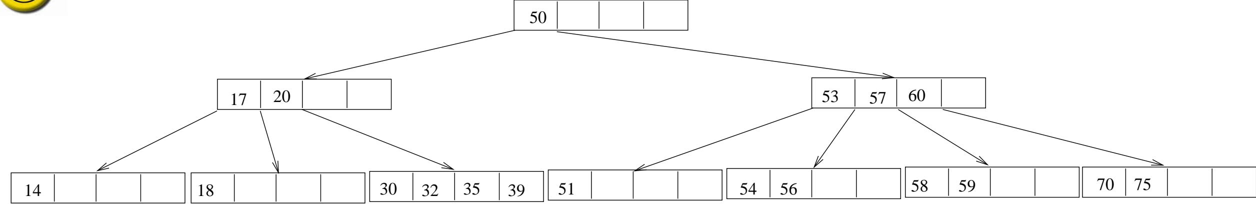
- Cada node té l'estructura següent:



  - m és l'ordre de l'arbre (i.e., el nombre màxim de fills que pot tenir un node).
  - n és el nombre de claus emmagatzemades a un node ($n \leq m - 1$).
  - $n + 1$ és el nombre de fills d'aquell node.
  - $k_0, \ldots, k_{n-1}$ són les n claus emmagatzemades al node.
  - e_i ($0 \leq i \leq n - 1$) és el valor associat a la clau k_i.
    (Sovint, els arbres m-aris de cerca s'usen com a índexs. En aquest cas, e_i indica la pàgina del fitxer de dades on s'emmagatzema l'enregistrament que té la clau k_i). El capítol 7 mostra més detalls en relació amb els índexs.
  - $p_0, \ldots, p_n$ són punters a altres nodes de l'arbre.
- $\forall i : 0 \leq i \leq n - 2 : k_i < k_{i+1}$.
- $\forall i : 0 \leq i \leq n - 1$: les claus del node apuntat per p_i són menors que k_i.
- $\forall i : 1 \leq i \leq n$; les claus apuntades per p_i són més grans que k_{i-1}.
- Els nodes apuntats per $p_0, \ldots, p_n$ són arrels d'arbres m-aris de cerca.

Exemple 6.8:

El següent és un arbre 5-ari de cerca (també es diu que el seu ordre és 5).



En aquest exemple i en els següents no mostrem els valors associats a cada clau. Només les claus.

■ ■ ■

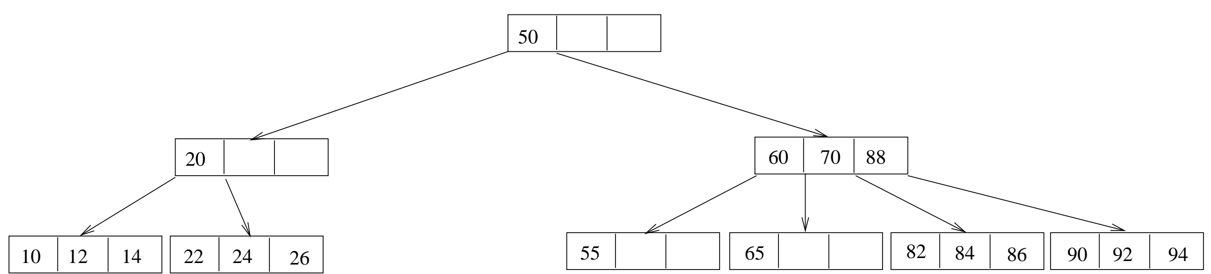
L'objectiu dels arbres m-aris de cerca és millorar el comportament d'un arbre binari de cerca augmentant el factor de ramificació (de 2 a m).

6.3.2 Arbres B. Definició

Un *arbre B* d'ordre m és un arbre m-ari de cerca en què:

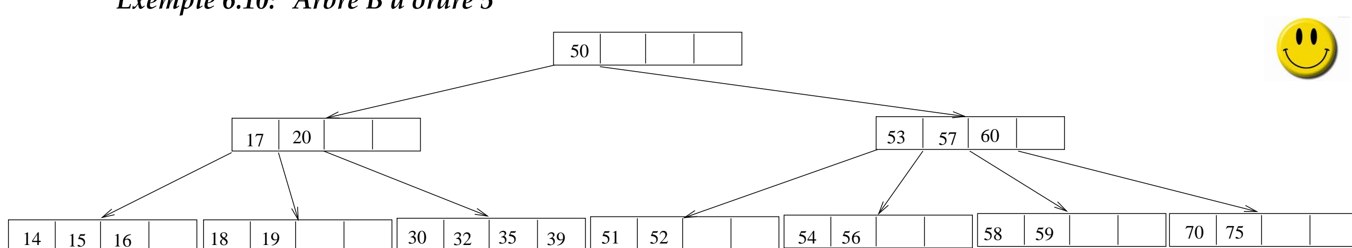
- Cada node intermedi (amb la possible excepció de l'arrel) ha de tenir un mínim de $\lceil \frac{m}{2} \rceil$ fills ($\lceil \frac{m}{2} \rceil - 1$ claus).
- Cada fulla ha de tenir un mínim de $\lceil \frac{m}{2} \rceil - 1$ claus.
- L'arrel ha de tenir almenys 2 fills (llevat del cas que sigui una fulla).
- Totes les fulles estan al mateix nivell.

Exemple 6.9: Arbre B d'ordre 4



■ ■ ■

Exemple 6.10: Arbre B d'ordre 5



■ ■ ■

Els arbres B milloren l'eficiència dels ABC en dos sentits:

- N'augmenten el factor de ramificació.
- El mantenen equilibrat.

El cost d'una cerca d'una clau en un arbre B d'ordre m i amb n claus inserides és:

- Cas pitjor: O($\log_{m/2}(n)$).
- Cas millor: O($\log_m(n)$).

Com es fan les insercions i eliminacions en un arbre B per mantenir sempre aquesta estructura? A les properes seccions ho mostrem.

6.3.3 Arbres B. Algorisme d'inserció

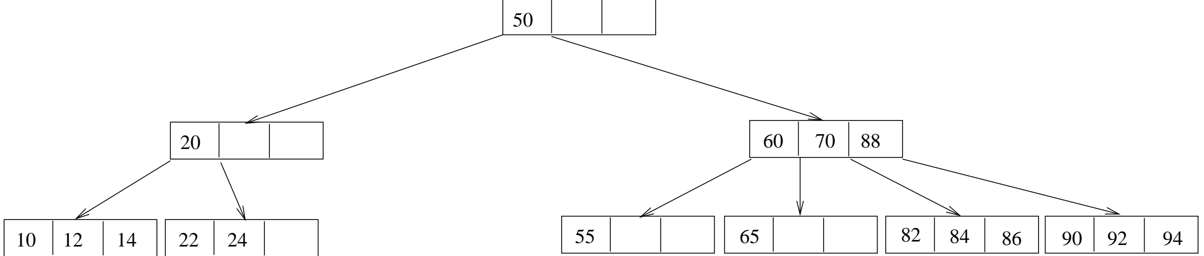
El procediment general d'inserció d'una parella (*clau, valor*) en un arbre B és el següent.

acció `inserir (a:ArbreB, k:Clau, v:Valor)`

1. Cercar la clau `k` a l'arbre.
2. Si es troba, substituir el valor associat a `k` per `v`.
3. Si no es troba, ens situem a la fulla `nod` on s'hauria de col·locar `k`.
4. Si la fulla `nod` no és plena, s'hi posa `(k,v)`.
5. Si és plena, es crea una nova fulla `germa` (*split*) i es distribueixen les claus de `nod` i `k` entre `nod` i `germa` (excepte la clau del mig `kmig`).
6. Situar `kmig` al node pare de `nod` seguint els passos 4-5. Crear convenientment un punter a `germa`.

Exemple 6.11: Inserció en un arbre B. Inserció de 26

Als exemples usarem l'arbre B següent:

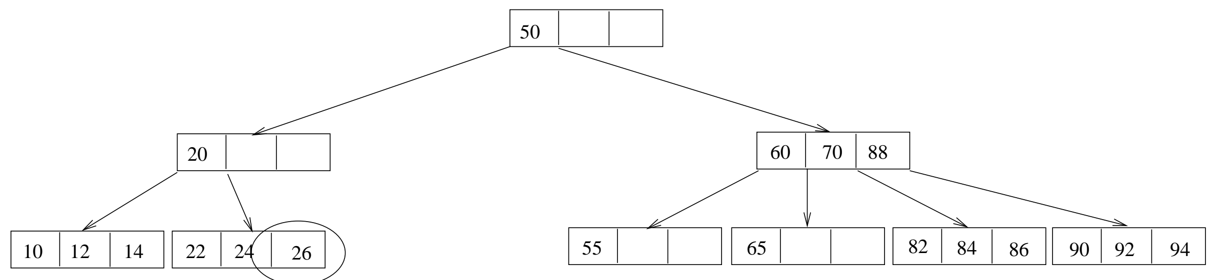


Procedim a inserir-hi la clau 26:

1. Cerquem la clau 26.

   No hi és.

2. La fulla on s'ha d'inserir la clau 26 no és plena: hi posem aquesta clau i acabem.



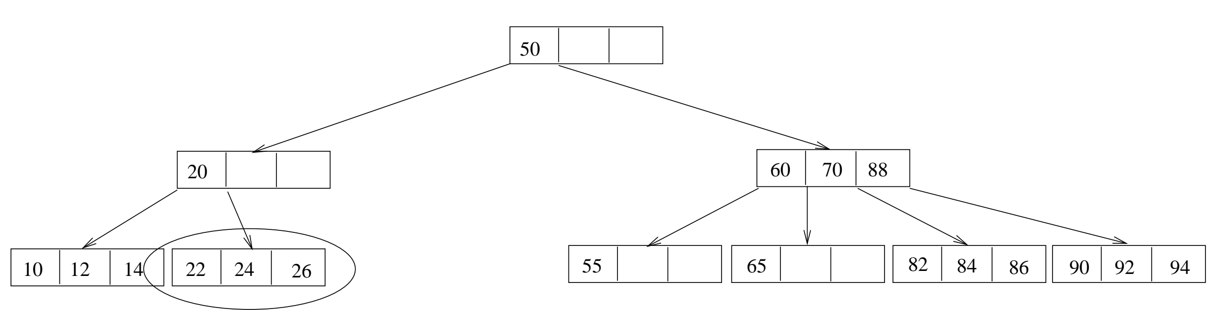
■ ■ ■

Exemple 6.12: Inserció de 30

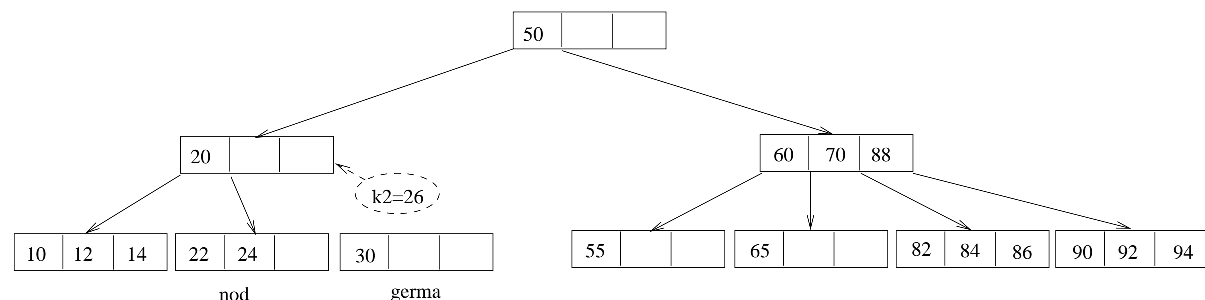
1. Cerquem la clau 30.

   No hi és.

2. La fulla `nod` on s'ha d'inserir la clau 30 és plena.

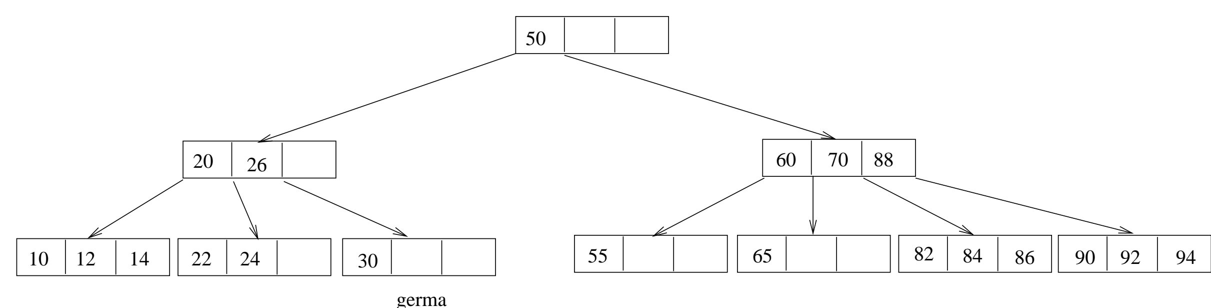


3. Apliquem *split*.



Generem un node `germa` i distribuïm les claus de `nod` i la clau 30 entre `nod` i `germa`. La clau del mig (26) s'haurà de col·locar al node pare. Des del node pare també s'haurà d'apuntar a `germa`.

4. Inserim 26 al node pare i actualitzem `germa` com a fill d'aquest pare.



Com que el node pare no estava ple, no ha calgut fer cap nou *split* i ja hem acabat.

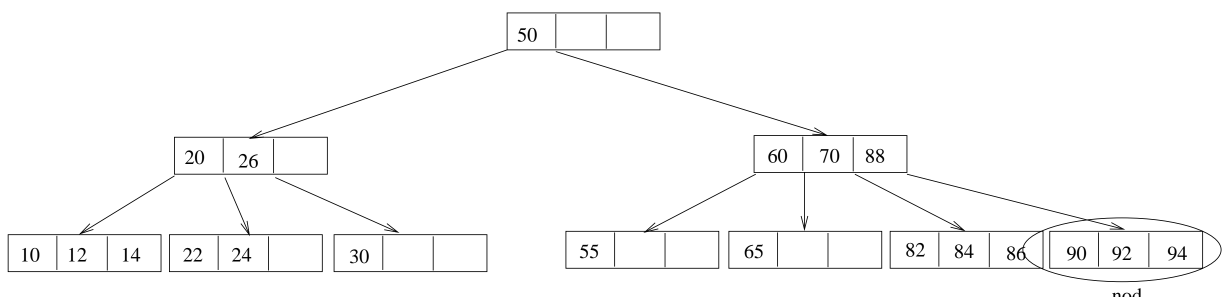
■ ■ ■

Exemple 6.13: Inserció de 91

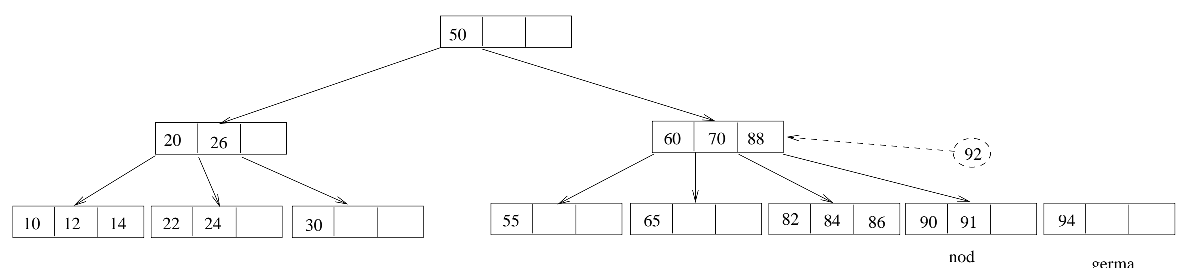
1. Cerquem la clau 91.

   No hi és.

2. La fulla `nod` on s'ha d'inserir la clau 91 és plena.

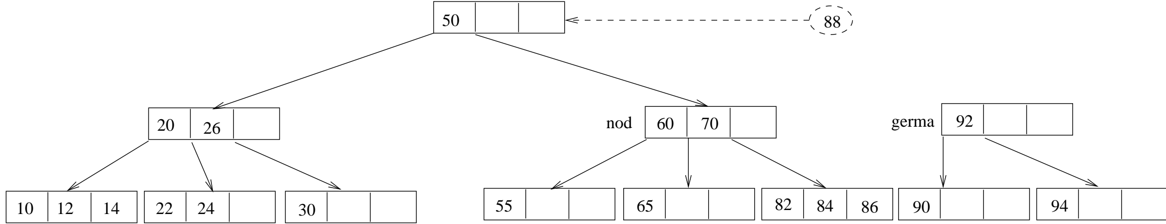


3. Apliquem *split*.

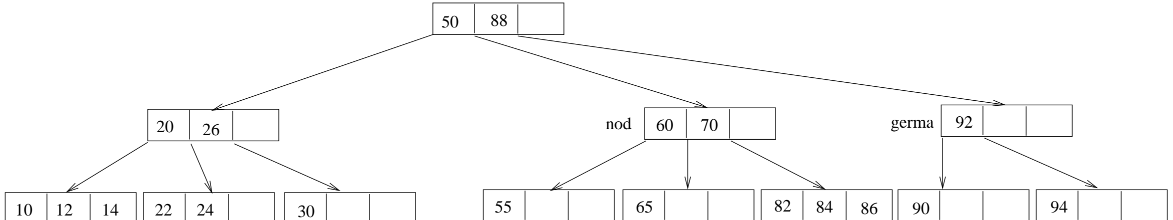


   Generem un node `germa` i distribuïm les claus de `nod` i la clau 91 entre `nod` i `germa`. La clau del mig (92) s'haurà de col·locar al node pare. Des del nivell del node pare també s'haurà de resoldre la filiació de `germa`.

4. Com que el pare està ple, apliquem un nou *split* sobre el node pare (que ara anomenem `nod`). Aquest *split* ens genera un `germa` i una clau (88) que inserirem al nivell superior.



5. Completem el procés d'inserció col·locant la clau 88 al node arrel (no cal fer un nou *split* perquè el node arrel no és ple) i apuntant `germa`. Ja hem acabat.



■ ■ ■

Algorisme d'inserció (*)

Aquesta secció mostra una descripció més detallada, usant pseudocodi de l'algorisme d'inserció.

acció `inserir (a: ArbreB, k: Clau, v:Valor)`

Post:
El parell <`k`,`v`> s'ha inserit a l'arbre B `a`. Si `k` ja hi era present, s'ha actualitzat el seu valor associat.
El procés d'inserció pot donar lloc a un increment dels nodes o de l'alçada de l'arbre B `a`.

```
acció inserir (a:ArbreB, k:Clau, v:Valor)
    inserit:=fals;
    <trobat,nod>:=cerca(a,k);
    si trobat llavors //Clau ja existent a l'arbre B
       substituirValor(nod,k,v);
       inserit:=cert;
    si no si nod=NULL llavors //Inserció en un arbre B buit
       nod:=crearNode();
       arrel:=nod;
       pfill:=NULL;
    fsi
    mentre ¬ inserit fer
       si esPle(nod) llavors
          <k2,v2,germa>:=split(nod,k,v,pfill);
          k:=k2; v:=v3; pfill:=germa;
          si (tePare(nod)) llavors nod:=pare(nod);
          si no
             arrel:=crearNode();
             primerFill(arrel):=nod;
             nod:=arrel;
          fsi
       si no // Node d'inserció nod no ple
          inserirParell(k,v,nod,pfill);
          inserit:=cert;
       fsi
    fmentre
facció
Inv: hem d'inserir el parell <k,v> al node nod i cal situar pfill com un fill de nod
```

L'algorisme d'inserció usa les accions `cerca` i `split` que especifiquem tot seguit:

funció cerca (`a: ArbreB, k: Clau`) retorna <`trobat: Booleà, nod: Node`>)

Post:

- `trobat=cert` si la clau `k` es troba a l'arbre `a` i fals en qualsevol altre cas.
- Si `trobat=cert`, `nod` és el node en què es troba la clau `k`. En qualsevol altre cas, `nod` és la fulla allà on s'hauria d'inserir `k`.

funció split (`k: Clau, v:Valor, nod:Node, pfill:Node`)
retorna <`k2:Clau, v2:Valor, germa: Node`>)

Post:
S'ha creat un nou node `germa` i s'han repartit la clau `k` i les que es trobaven a `nod` (en total `m`, on `m` és l'ordre de l'arbre B) de la manera següent:

- Les $\lceil (m)/2 \rceil$ més petites es posen a `nod`.
- Les $\lfloor (m)/2 \rfloor$ més grans es posen a `germa`.
- La clau del mig i el seu valor associat es retornen (`k2, v2`) per tal de poder ser situades al pare de `nod`.

S'ha situat `pfill` com a fill de `nod` o de `germa`, segons convingui.
Com a resultat de l'aplicació d'aquesta funció es retorna també el node `germa` que haurà de ser col·locat com a fill del pare de `nod` (o d'algun altre node del nivell superior).

6.3.4 Arbres B. Algorisme d'eliminació

La idea de l'algorisme d'eliminació d'una clau en un arbre B és la següent:

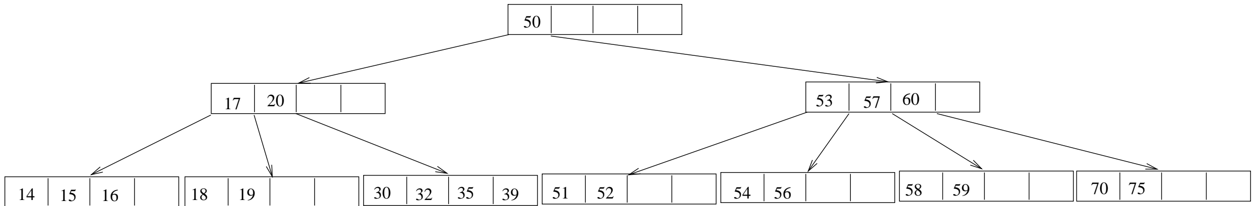
acció `eliminar (a:ArbreB, k: Clau)`

1. Cercar la clau `k`.
2. Si la clau `k` es troba en un node intermedi: substituir la parella `(k,v)` per la parella `(k',v')`, on `k'` és la clau següent a `k` en ordre ascendent. `k'` estarà situada en una fulla.
3. Eliminar `(k',v')` de la fulla on es troben seguint el procediment d'eliminació d'una clau d'una fulla (es descriu seguidament).
4. Si la clau `k` es troba en una fulla `nod`, eliminar la parella `(k,v)` d'aquella fulla i
   (a) Si a `nod` resten almenys $\lceil \frac{m}{2} \rceil - 1$ claus, hem acabat.
   (b) Si no, si `nod` té algun germà amb excedent de claus (més de $\lceil \frac{m}{2} \rceil - 1$), redistribuir-les.
   (c) Si no, fusionar `nod` amb un germà adjacent. Aquesta fusió provocarà que el pare perdi un fill i una clau. Anomenar `nod` al pare i tornar a (a).

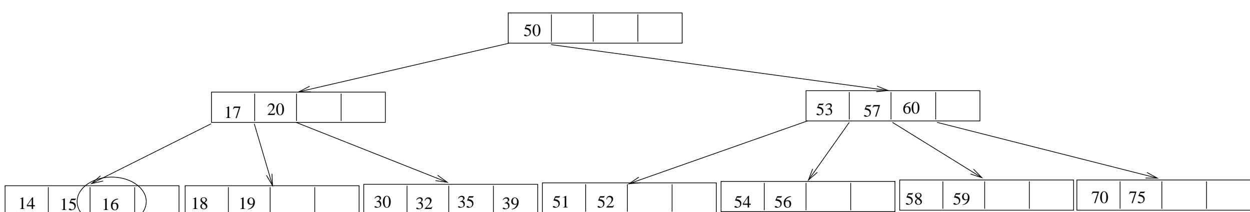
Apliquem aquest algorisme als exemples següents.

Exemple 6.14: Eliminació de 16

Als exemples usarem l'arbre B següent:

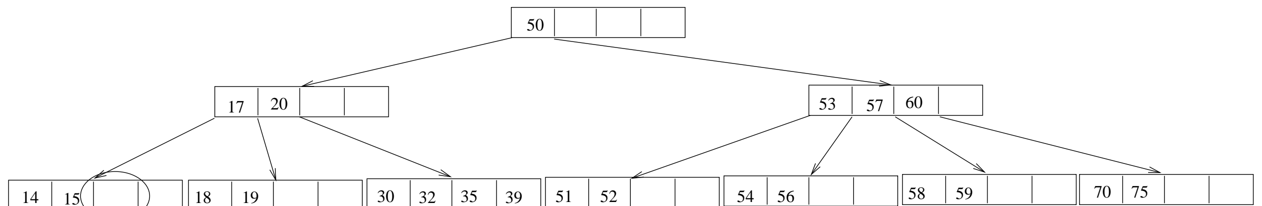


1. Cerquem la clau 16.



   La trobem en una fulla.

2. Eliminem la clau 16 de la fulla on es troba[2].



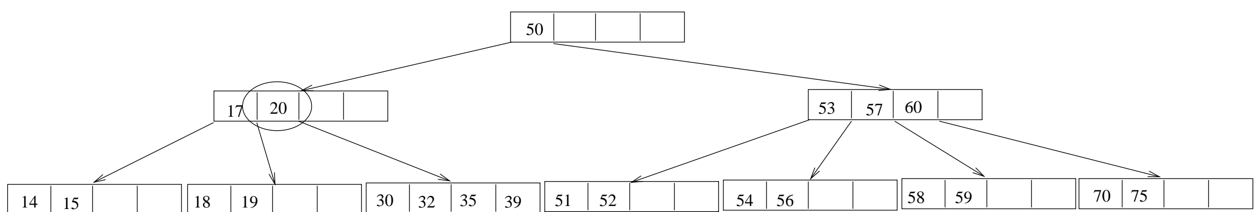
   Com que després d'eliminar 16 la fulla té encara suficients claus, hem acabat.

■ ■ ■

[2] Recordem que quan al text parlem d'eliminar una clau estem incloent l'eliminació del valor que porta associat aquella clau.

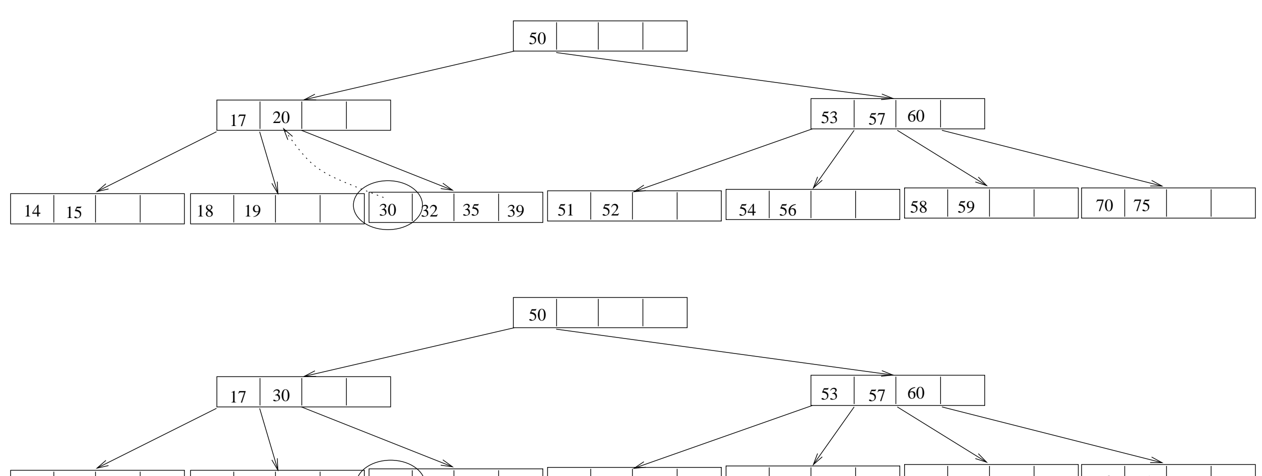
Exemple 6.15: Eliminació de 20

1. Cerquem la clau 20.



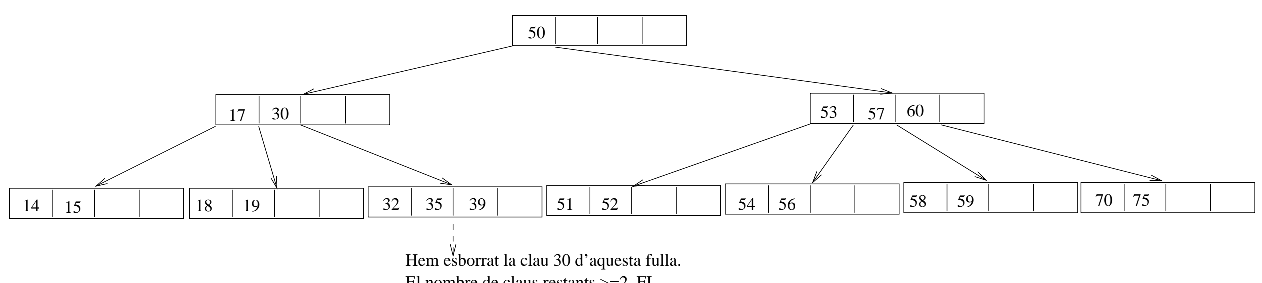
   La trobem en un node interior.

2. La substituïm per la següent clau en ordre alfabètic (que estarà en una fulla): 30.



   Ara hem d'eliminar la clau 30 de la fulla.

3. Cridem `eliminarFulla(...)` per tal d'eliminar la clau 30 de la fulla.

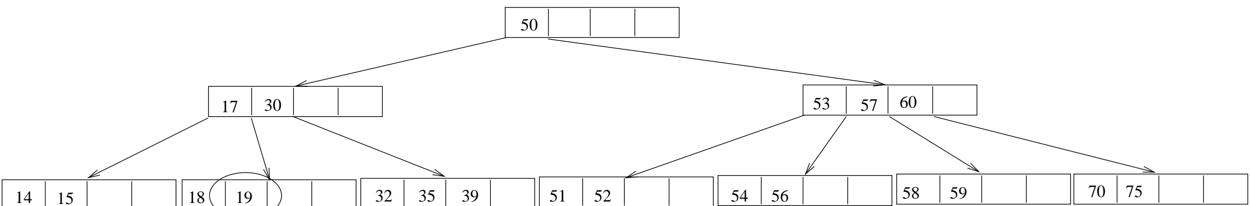


S'esborra la clau 30. La fulla queda amb un nombre de claus suficient. Hem acabat.

■ ■ ■

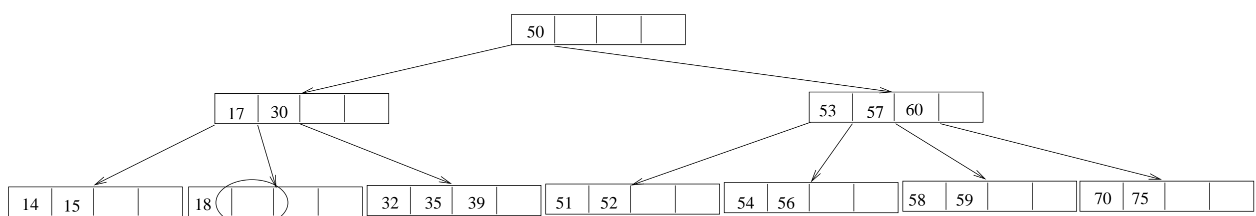
Exemple 6.16: Eliminació de 19

1. Cerquem la clau 19.



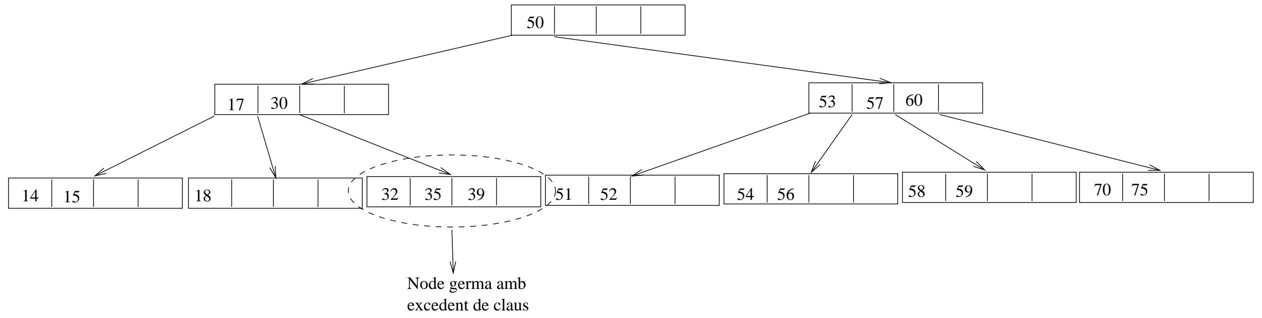
   La trobem en una fulla.

2. Apliquem el procediment d'eliminació d'una clau d'una fulla.



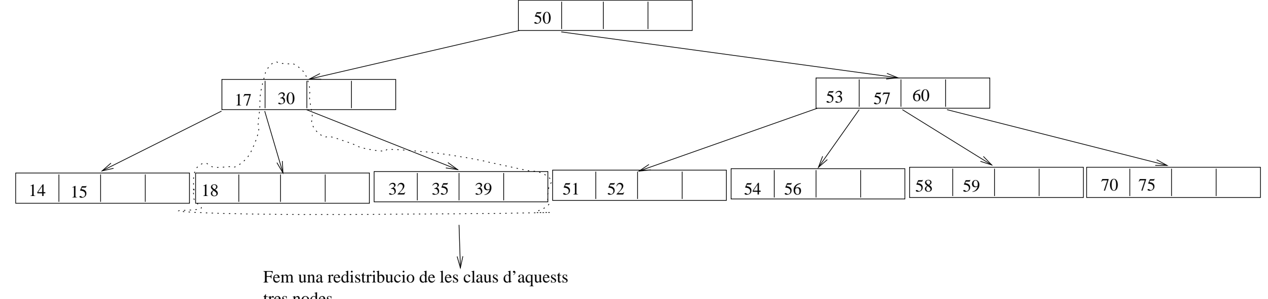
   El nombre de claus que queden a la fulla (1) és menor que el mínim (2).

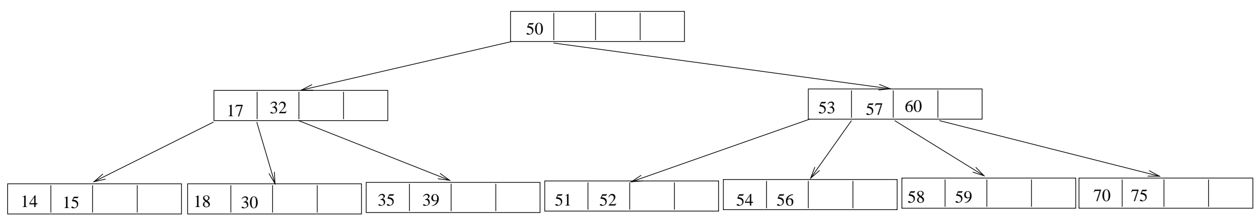
3. Té la fulla alguna germana amb excedent de claus?



   Sí, la fulla de la seva dreta té 3 claus i només en necessita un mínim de 2.

4. Redistribuïm les claus dels dos germans i el seu pare i acabem.

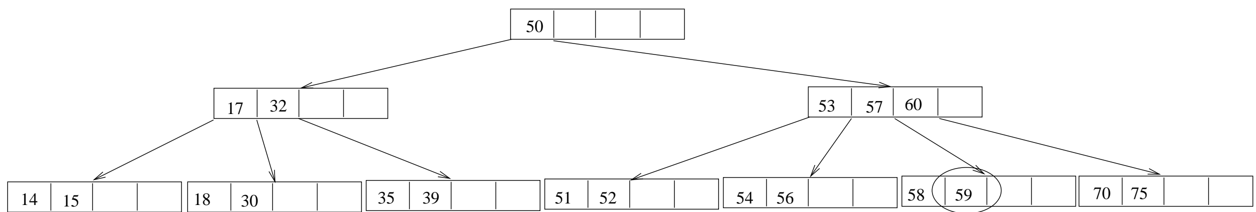




■ ■ ■

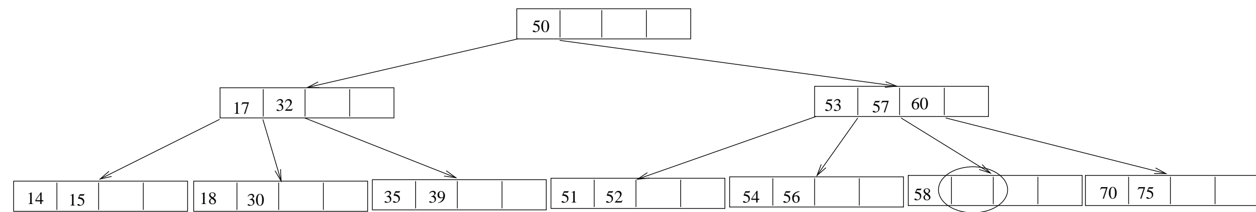
Exemple 6.17: Eliminació de 59

1. Cerquem la clau 59.



La trobem en una fulla.

2. Apliquem l'algorisme d'eliminació d'una clau d'una fulla.

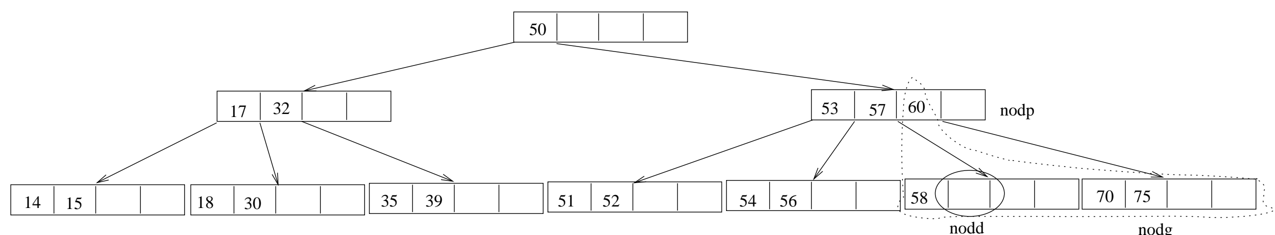


El nombre de claus que queden a la fulla (1) és menor que el mínim (2).

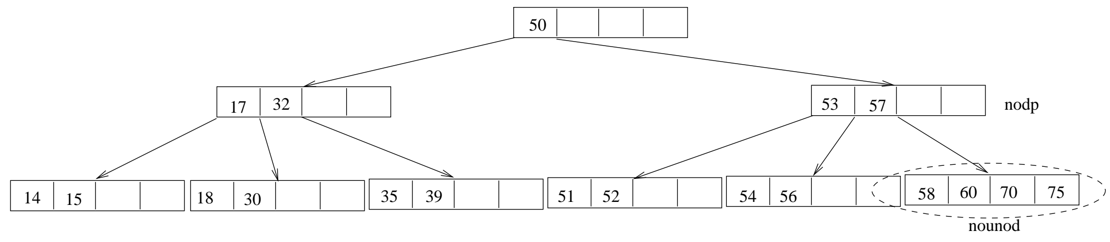
3. Té la fulla alguna germana amb excedent de claus?

   No.

4. Fusionem la fulla (`nodd`) que ha perdut una clau amb un dels seu germans (`nodg`). Cridem `fusioGermans(...))`.



Fusionem els nodes `nodd` i `nodg` per construir un únic `nounod`, fill de `nodp`. Distribuïm les claus adequadament entre `nodp` i `nounod` per tal de mantenir les propietats dels arbres B.



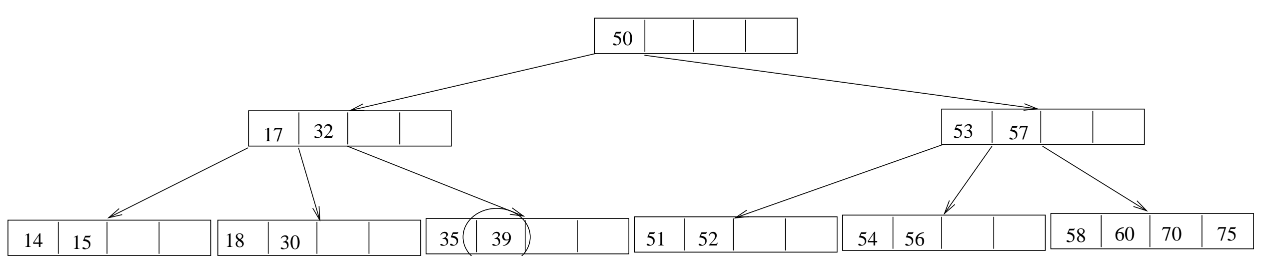
5. El pare (`nodp`), ha quedat amb un nombre suficient de claus?

   Sí. Acabem.

■ ■ ■

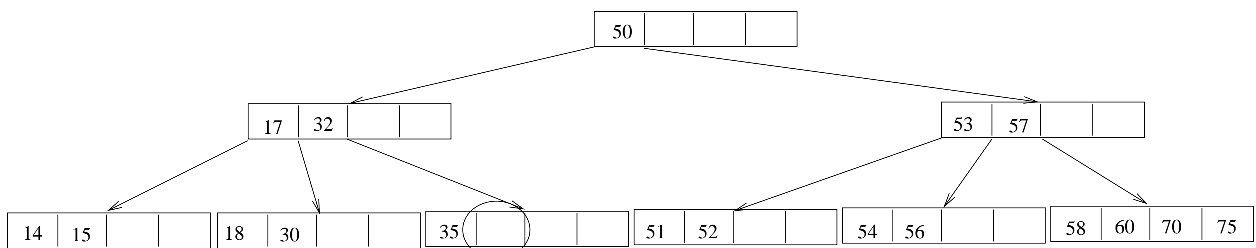
Exemple 6.18: Eliminació de 39

1. Cerquem la clau 39.



La trobem en una fulla.

2. Apliquem l'algorisme d'eliminació d'una clau d'una fulla.

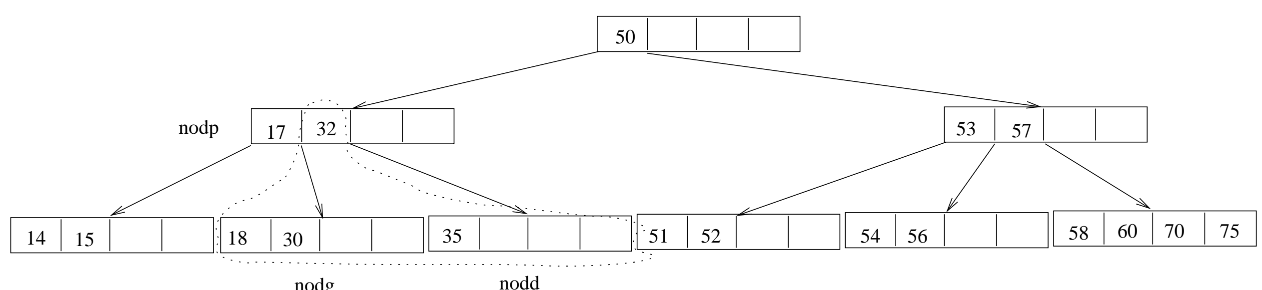


El nombre de claus que queden a la fulla (1) és menor que el mínim (2).

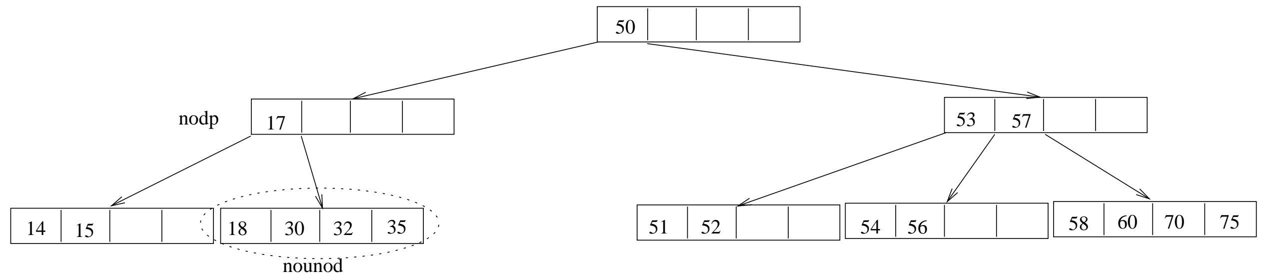
3. Té la fulla alguna germana amb excedent de claus?

   No.

4. Fusionem la fulla que ha perdut una clau amb un germà seu.



Fusionem els nodes `nodd` i `nodg` per construir un únic `nounod`, fill de `nodp`. Distribuïm les claus adequadament entre `nodp` i `nounod` per tal de mantenir les propietats dels arbres B.



5. El pare (`nodp`), ha quedat amb un nombre suficient de claus?

   No.

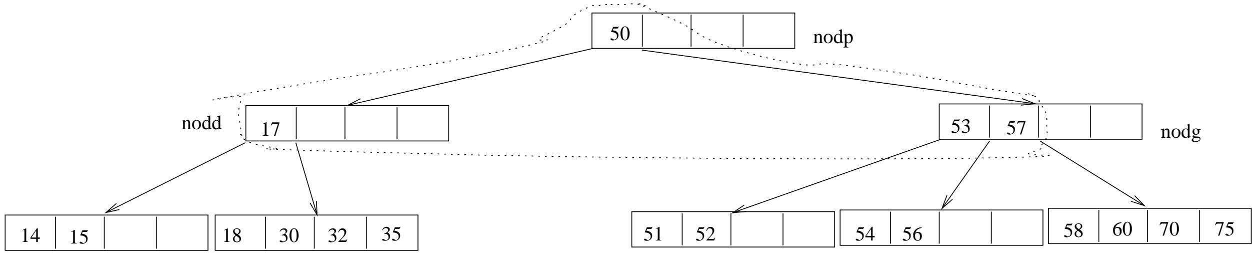
Ara hem d'aplicar la **redistribució** o la **fusió** amb els germans al nivell del node pare (`nodp`), perquè ara és aquest el node amb dèficit de claus.

Anomenem `nodd` a `nodp`.

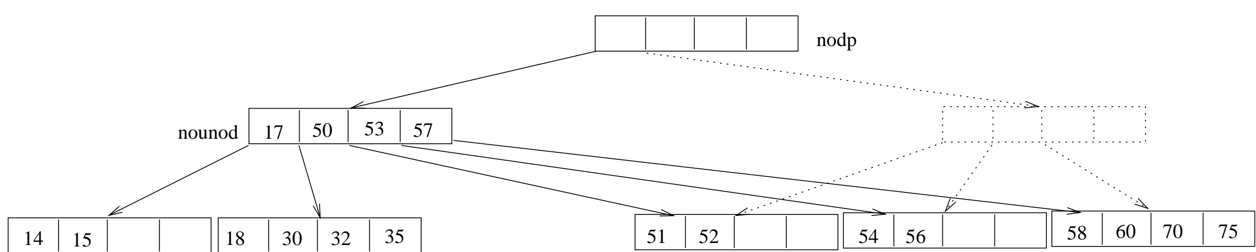
6. Algun node germà de `nod` té excedent de claus?

   No.

7. Fusionem `nom` amb el seu germà (`nodg`). Cridem a `fusioGermans(...))`.



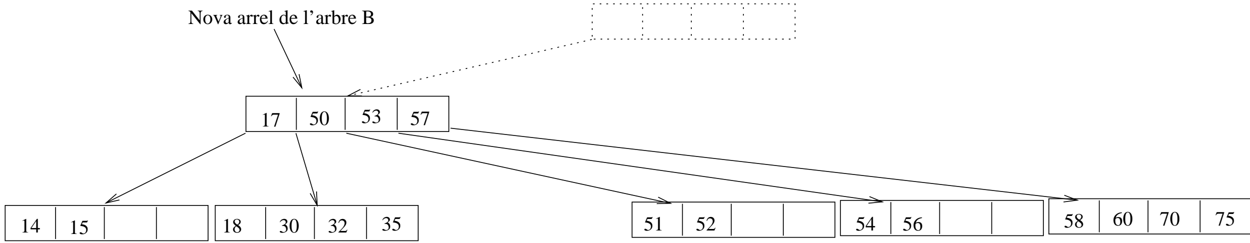
Fusionem els nodes `nodd` i `nodg` per construir un únic `nounod`, fill de `nodp`. Distribuïm les claus adequadament entre `nodp` i `nounod` per tal de mantenir les propietats dels arbres b.



8. Hem arribat ja a l'arrel?

   Sí. `nodp` és ja l'arrel.

9. L'arrel és buida i no és una fulla. Per tant, la nova arrel és l'únic fill de l'arrel antiga.



Com a conseqüència, reduïm un nivell a l'arbre B resultant.

■ ■ ■

Algorisme d'eliminació en un arbre B (*)

Aquesta secció mostra una descripció més detallada, usant pseudocodi de l'algorisme d'eliminacion d'un clau.

```
acció eliminar (a:ArbreB, k: Clau)
   <trobat,nodElim>:=cerca(a,k);
   si trobat llavors
     si ésFulla(nodElim,a) llavors
       eliminarFulla(a,k,nodElim);
     si no
       <kseg,fulla>:=substituir(a,k);
       eliminarFulla(a,kseg,fulla);
     fsi
   fsi
facció
```

Proposem, seguidament, l'especificació de les accions o funcions que apareixen a aquest algorisme.

```
funció cerca (a: ArbreB, k: Clau)
                         retorna <trobat: Boolea, nod: Node>
```

Post:

- `trobat=cert` si la clau `k` es troba a l'arbre `a` i en qualsevol altre cas.
- Si `trobat=cert`, `nod` és el node en al que es troba la clau `k`. En qualsevol altre cas, `nod` és la fulla allà on s'hauria d'inserir `k`.

```
funció substituir (a: ArbreB, k: Clau)
                         retorna <kseg: Clau, fulla: Node>
```

Post:
S'ha substituït la clau `k` del node intermedi on es trobava per la que la segueix en ordre ascendent dins l'arbre `a`.

- `kseg` és la clau que segueix a `k` en ordre alfabètic dins de l'arbre `a`.
- `fulla` és la fulla de l'arbre `a` on es troba `kseg` (vegeu la figura 6.8).

Comentaris:

- `kseg` sempre existeix.

<kseg,fulla>:=substituir(a,20)

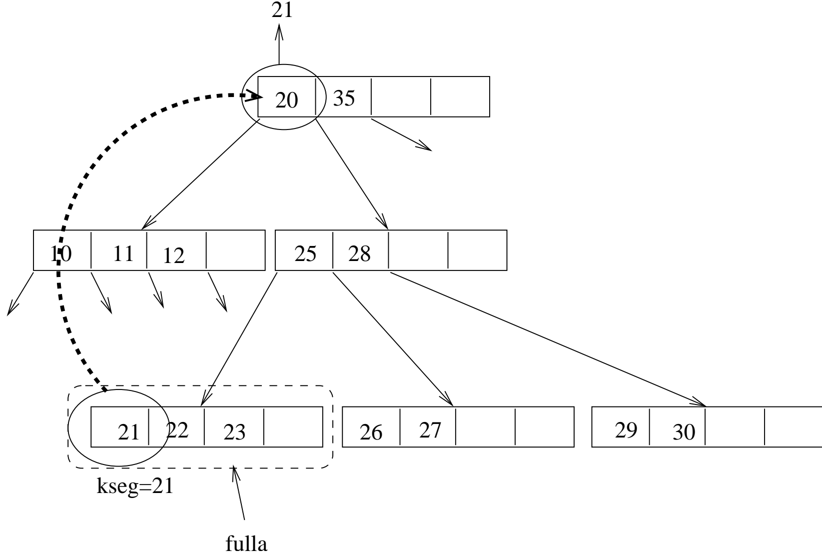


Figura 6.8: Substitució d'una clau d'un node intermedi per una altra d'una fulla

- `kseg` es troba sempre en una fulla.
- Com a conseqüència de la substitució de `k` per `kseg`, la clau `kseg` està repetida a l'arbre i cal eliminar-la de la fulla `fulla`.
- A aquest procés de substitució d'una clau d'un node intermedi per la seva següent en ordre alfabètic a vegades se l'anomena *aspiració*.

```
acció eliminarFulla (a:ArbreB, k:  Clau, nod:  Node)
   esborrar(k,nod);
   mentre ¬ teClausSuficients(nod) ∧ ¬ esArrel(nod) fer
     <trobat,nodgerma>:=cercarGermaAmbExcedents(nod);
     si trobat llavors
       redistribucioGermans(nod,nodGerma,pare(nod));
     si no
       nounod:=fusioGermans(nod,nodGerma,pare(nod));
       nod:=pare(nounod);
     fsi
   fmentre
   si esBuit(nod) ∧ esFulla(nod) llavors a:=arbreBuit();
   si esBuit(nod) ∧ ¬ esFulla(nod) llavors arrel(a):=primerFill(nod);

facció
```

funció `teClausSuficients (nod: Node)` **retorna** `b: Boolea`
Post:
Retorna cert si el nombre de claus de `nod` és més gran o igual que $\lceil m/2 \rceil - 1$ i fals en qualsevol altre cas.

funció `cercarGermaAmbExcedents (nod: Node)`
retorna `<trobat:Boolea, nodgerma: Node>`
Post:
Si el node `nod` té algun germà amb un nombre de claus més gran que $\lceil m/2 \rceil - 1$, `trobat=cert` i `nodgerma` es refereix a aquell node. En qualsevol altre cas, `trobat=fals` i `nodgerma` és indefinit.

funció `fusioGermans (nodd: Node, nodg: Node, nodp: Node)`
retorna `nounod: Node`
Pre:

- `nodd` és un node al qual hem eliminat una clau i ha quedat amb dèficit de claus (per tant, `nodd` conté $\lceil (m/2) \rceil - 2$ claus).
- `nodg` és un node germà adjacent a `nodd`. El node `nodg` té exactament $\lceil (m/2) \rceil$ claus (si en tingués més, hauríem pogut fer una redistribució).
- `nodp` és el node pare de `nodg` i de `nodd`.
- `kp` és la clau de `nodp` que es troba alfabèticament ordenada entre les claus de `nodg` i les de `nodd`.

Post:
Els nodes `nodd` i `nodg` s'han fusionat en un de sol: `nounod`. `nounod` és fill de `nodp` i conté les claus següents:

- Les $\lceil (m/2) \rceil - 2$ claus que contenia `nodd`.
- Les $\lceil (m/2) - 1 \rceil$ claus que contenia `nodg`.
- `kp`.

(Vegeu la figura 6.9).

Comentaris:

En total `nounod` té $\lceil (m/2) \rceil - 2 + \lceil (m/2) \rceil - 1 + 1$ claus. Aquest nombre és menor o igual que $m - 1$ (el nombre màxim de claus que pot contenir un node).

Els nodes que eren fills de `nodd` i `nodg` ara ho són de `nounod` i s'han distribuït als llocs corresponents per tal de respectar la definició dels arbres B.

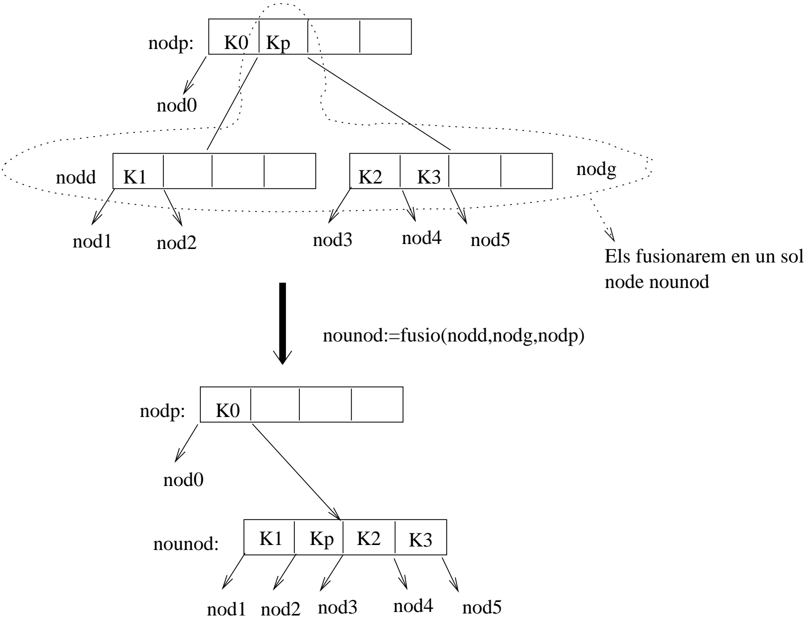


Figura 6.9: Fusió de dos nodes en un arbre B.

acció `redistribucioGermans (nodd: Node, nodg: Node, nodp: Node)`

Pre:

- `nodd` és un node al qual hem eliminat una clau i ha quedat amb dèficit de claus (per tant, `nodd` conté $\lceil(m/2)\rceil - 2$ claus).
- `nodg` és un node germà adjacent a `nodd`. El node `nodg` té més de $\lceil(m/2)\rceil - 1$ claus (i, per tant, n'hi podem treure una).
- `nodp` és el node pare de `nodg` i de `nodd`.
- `kp` és la clau de `nodp` que es troba alfabèticament ordenada entre les claus de `nodg` i les de `nodd`.

Post:
S'han redistribuït les claus de `nodd`, de `nodg` i de `nodp` de tal manera que nodd ha guanyat una clau i tots tres nodes continuen conservant les propietats dels arbres B.
En particular:

- `kp` passa a `nodd`.
- La clau menor de `nodg` es desplaça a `nodp`, en substitució de `kp`.
- Els nodes fills de nodg es rearrangen convenientment (per tal de conservar les propietats dels arbres B):

(Vegeu la figura 6.10.)

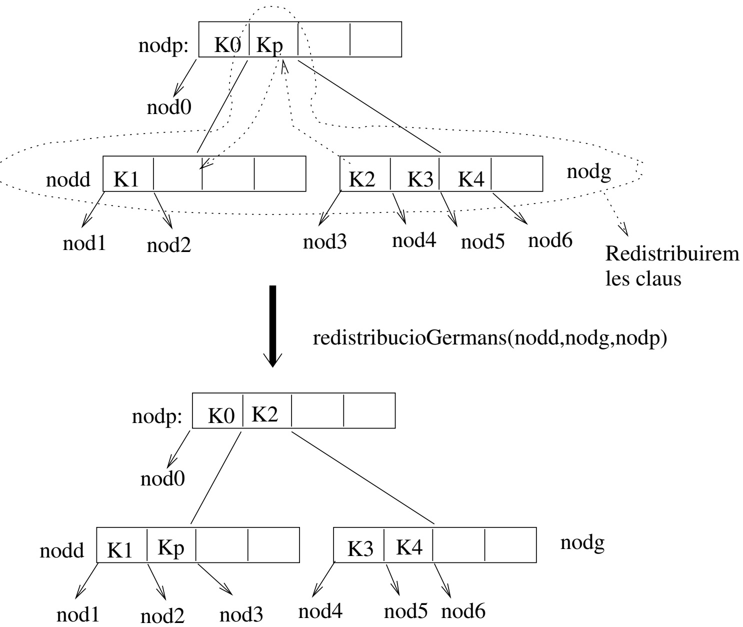


Figura 6.10: Redistribució de claus en un arbre B.

6.3.5 Ús dels arbres B

Els arbres B proporcionen una implementació arborescent de les taules que milloren els ABC perquè poden tenir un factor de ramificació més gran que 2 i, a més a més, es mantenen equilibrats. L'àmbit en què s'usen més els arbres B (altres estructures derivades) és el dels índexs de bases de dades. Però aquesta és una altra història i l'explicarem al capítol següent. Només us cal passar pàgina.

6.4 Racó lingüístic

En aquesta secció comentem els termes tècnics usats en aquest capítol que no estan estandarditzats al diccionari de l'Institut d'Estudis Catalans, a Termcat o a [CM94].

- *Arbre binari de cerca. Arbre m-ari de cerca.*

  Traducció natural dels termes anglesos *Binary Search Tree* i *m-ary Search Tree*.

- *Arbre B. Arbre B+.*

  Traducció natural dels termes anglesos *B Tree* i *B+ Tree*.

- *Arbre roig-negre.*

  Traducció natural del terme anglès *Red-Black Tree*.

Capítol 7

Índexs

Un dels serveis més importants que ofereix la informàtica a la societat és l'emmagatzemament de volums ingents d'informació i la recuperació eficient d'aquesta informació.

Les taules que hem presentat als capítols anteriors són una primera resposta a l'emmagatzemament i recuperació eficient d'informació ja que ens permeten fer cerques ràpides per clau en col·leccions de dades. Però tenen una limitació molt important: les taules que hem vist s'emmagatzemen en memòria i, per tant, el volum d'informació que poden encabir és molt limitat i, a més, la informació és volàtil: quan s'acaba l'aplicació, es perd.

Les bases de dades resolen aquestes dificultats perquè proporcionen una eina d'emmagatzemament i recuperació de grans volums d'informació que és alhora persistent i eficient. Ara bé, per aconseguir aquesta eficiència, les bases de dades empren unes estructures de dades auxiliars que acceleren la cerca d'informació. Aquestes estructures de dades són taules com les que hem estudiat als capítols anteriors però que es guarden en un fitxer i, per tant, són persistents.

Aquestes taules persistents que ens serveixen per accelerar la cerca d'informació en una base de dades s'anomenen *índexs* i en aquest capítol els introduirem.

7.1 Estructures de dades en memòria persistent

En els capítols anteriors hem fet sempre la hipòtesi que les dades emmagatzemades a les llistes, els arbres o les taules que hem descrit estaven situades a la memòria. Aquest model ens serveix en moltíssimes ocasions. Vosaltres mateixos us heu trobat amb multitud d'aplicacions que necessiten treballar amb una llista o una taula d'elements que caben perfectament a la memòria i que, possiblement, no es necessitaran més quan el programa acabi la seva execució. Com hem vist als capítols precedents, en aquestes situacions usem les classes i interfícies com ara `List<T>`, `Map<K, V>`, `LinkedList<T>`... que ens proporciona la JCF.

Ara bé, les estructures de dades en memòria pateixen dos problemes importants que fan que, en algunes ocasions, siguin poc pràctiques:

- Imagineu que tenim un volum molt important de dades, tan important que, possiblement, no càpiga a la memòria i, per tant, no càpiga a cap estructura de dades que puguem emmagatzemar en memòria. I, tanmateix, imagineu que necessitem accedir eficientment a aquelles dades.

- Imagineu que les dades hagin de ser persistents. O sigui, necessitem que les dades (possiblement en un volum molt important) es conservin quan acabi l'execució del programa per tal de poder-les usar més endavant.

  (Us imagineu que cada matí haguéssim de tornar a entrar les dades de tots els cotxes matriculats a Lleida per tal que els treballadors del servei de trànsit, durant el dia, poguessin usar un aplicatiu per fer consultes sobre aquelles dades? Ridícul, oi?)

Quan tenim un volum molt important de dades que requereixen persistència no és adient emmagatzemar-les en cap de les estructures que hem vist. En aquells casos, aquelles dades s'insereixen en una *base de dades*.

7.1.1 Bases de dades (relacionals)

En un context informàtic, una **base de dades** és un magatzem de dades que:

- És capaç de contenir un volum molt important de dades de manera organitzada.
- És capaç d'emmagatzemar-les de forma persistent.
- Permet gestionar-les (inserir, eliminar, modificar, consultar) eficientment.

Existeixen diferents models de bases de dades. Avui per avui (i des de fa una pila d'anys), el més utilitzat[1] és l'anomenat *model relacional*.

[1] No l'únic, que quedi clar. De fet, darrerament, els models no relacionals, que usualment es coneixen amb el nom genèric de *No-SQL* estan prenent molta importància.

Una **base de dades relacional** és una base de dades en què les dades s'organitzen conceptualment en forma de **taules relacionals**.

Usualment, una taula relacional modelitza un tipus d'objectes (e.g., `Estudiant`, `Titulació`, `Assignatura`, `Factura`, `Departament`...).

Cada taula relacional està descrita en termes d'una llista d'**atributs** que descriuen el tipus d'objectes modelitzats per la taula (`Estudiant: nom`, `nif`, `data naixement`; `Titulació`: `identificador`, `nom`, `centre`...).

Una taula relacional la podem imaginar composta per *fileres* i *columnes*:

- Una **columna** d'una taula relacional representa un atribut d'aquella taula (e.g., les columnes `nom`, `nif` i `data de naixement` a la taula `Estudiant`).
- Una **filera** d'una taula relacional representa una **tupla** o **registre**, això és, una instància concreta del tipus d'objectes que modelitza aquella taula.

  Un registre està compost per un valor concret per a cada atribut de la taula relacional.

La **clau primària** d'una taula relacional està formada per un subconjunt d'atributs de la taula que són identificadors. Això vol dir que *no hi pot haver dos registres amb els mateixos valors per al conjunt d'aquells atributs*. En moltes ocasions, la clau primària està formada per un únic atribut. En aquest cas no hi pot haver dos registres amb el mateix valor per a l'atribut que actua com a clau primària.

Dues taules relacionals poden estar relacionades. Per exemple, la taula `Estudiant` està relacionada amb la taula `Titulació` per la relació `estudia-titulació`. Això vol dir que un registre `Estudiant` pot estar relacionat amb un registre `Titulació` (i.e., la titulació que estudia).

Exemple 7.1:

La figura 7.1 mostra les taules relacionals `Estudiant i Titulació`, amb els seus *atributs* (columnes) i *registres* (fileres). La figura mostra tres estudiants diferents i tres titulacions.

L'atribut `nif` de la taula `Estudiant` n'és la clau primària. L'atribut `identificador` de la taula `Titulació` n'és l'identificador. Notem que no hi ha dos estudiants amb el mateix `nif` ni tampoc dues titulacions amb el mateix `identificador`. No hi ha dificultat que dos estudiants tinguin la mateixa data de naixement (fins i tot, podrien tenir el mateix nom!).

Un registre `Estudiant` està relacionat amb un registre `Titulació`. Per exemple, el *pep* estudia el *grau en Enginyeria Informàtica*, com *l'anna*. Mentre que la *gemma* estudia el *màster en Enginyeria Informàtica*. Noteu que un registre `Titulació`, com ara, *grau en E. Informàtica* pot estar relacionat (ser estudiat) per molts estudiants; en canvi, un estudiant només pot estudiar una titulació (imposem aquesta restricció al nostre model). Quan ens trobem amb una relació com aquesta, diem que es tracta d'una **relació** 1-n.

Aquesta relació 1-n entre les dues taules relacionals es representa amb un nou atribut a la taula `Estudiant` (`títol`) que relaciona cada registre de la taula `Estudiant` amb el registre de

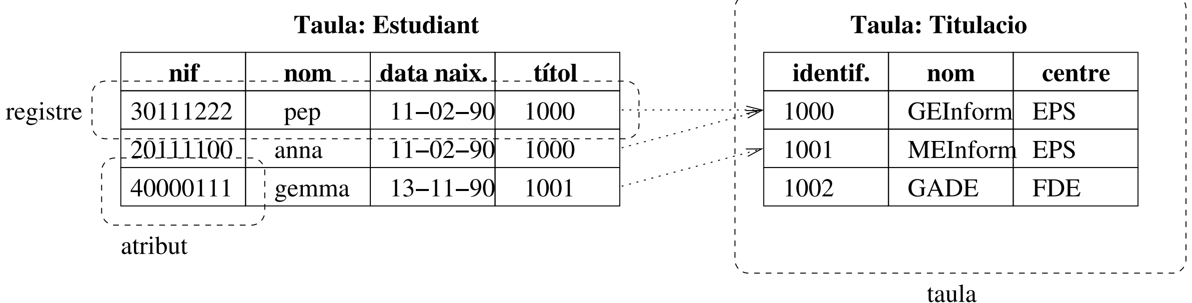


Figura 7.1: Exemple de dues taules en una base de dades relacional

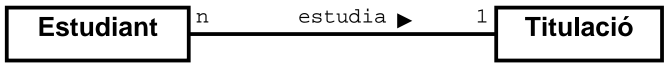


Figura 7.2: Relació 1-n entre dues taules d'una base de dades

la taula `Titulacio` corresponent a la titulació que estudia. En concret, per a cada estudiant, `títol` conté *la clau primària de la titulació que estudia*. Per cert, a l'atribut `títol` se l'anomena **clau forana**. La figura 7.2 mostra la representació gràfica de la relació entre dues taules en el llenguatge de modelització UML.

■ ■ ■

Atenció: tenim un problema de notació!

En aquest capítol necessitarem parlar sovint de *taula d'una base de dades* en el sentit que acabem de definir. Però als capítols 4, 5 i 6 hem definit una *taula* com a una col·lecció de parelles (*clau*,*valor*) que permeten un accés eficient per clau. Són dos conceptes diferents i, malauradament, se'ls nota de la mateixa manera.

Per aquest motiu, **en aquest capítol, sempre que ens vulguem referir a les taules en la seva accepció de col·lecció de parelles (*clau*, *valor*) (amb el sentit dels capítols 4, 5 i 6) usarem el terme *taula associativa*. Quan ens vulguem referir a les taules d'una base de dades, usarem el terme *taula relacional*.**
Si, en una situació concreta, no hi ha cap dubte sobre a quin tipus de taula ens estem referint, podrem usar simplement el terme *taula*.

7.1.2 Emmagatzemament físic de les taules relacionals

Una cosa és com veiem una base de dades conceptualment (en forma de taules relacionals amb registres i atributs i relacions entre elles) i una altra cosa és com s'emmagatzemen

aquestes taules al disc per tal d'assegurar la persistència de la base de dades.

Dels detalls d'aquest emmagatzemament (i, en general, de tota la gestió d'una base de dades) se n'encarrega una aplicació anomenada **sistema gestor de bases de dades (SGBD)** (*Database Management System*, *DBMS*, en anglès). Aquestes aplicacions són molt complexes i cadascuna d'elles té la seva manera específica de gestionar l'emmagatzemament de les taules relacionals al disc. El nostre propòsit aquí no és el d'explicar els detalls d'aquest emmagatzemament sinó, més aviat, de donar-ne algunes idees que motivin la necessitat de disposar d'*índexs* (la paraula màgica que dóna títol al capítol) per accedir eficientment als registres de les taules relacionals.

Comencem explicant on s'emmagatzemen les taules relacionals.

Fitxers

Si volem que les taules relacionals d'una base de dades siguin persistents, finalment s'hauran d'emmagatzemar en fitxers en un disc o un altre mitjà d'emmagatzemament persistent.

El SGBD emmagatzema una taula relacional en un o diversos fitxers. En aquest capítol, per simplificar, suposarem que cada taula està emmagatzemada en un fitxer diferent.

Pàgines i blocs

Un fitxer es descompon en pàgines.

Una **pàgina** és un paquet de dades del fitxer contigües. Les pàgines tenen una mida fixa. Aquesta mida típicament pot ser de 4 o 8 KB.

Una pàgina és la unitat de lectura/escriptura del SGBD. O sigui, quan el SGBD llegeix o escriu informació al disc, llegeix o escriu una pàgina. Cada pàgina està identificada per un número de pàgina.

Una pàgina conté un nombre determinat de registres de la taula relacional. Un registre no es trenca en dos pàgines diferents. Per aquest motiu, la mida d'una pàgina sol delimitar la mida màxima que pot tenir un registre.

Si volguéssim baixar una mica més de nivell aprendríem que, físicament, una pàgina s'emmagatzema en un **bloc** del disc i que un bloc és un conjunt de sectors del disc que el sistema operatiu pot adreçar. Un bloc té la mateixa mida que una pàgina. Però tot això ja és baixar massa de nivell per als propòsits d'aquest capítol.

Exemple 7.2:

Ara, doncs, ens podem imaginar una de les nostres taules relacionals, per exemple, la taula `Estudiant` amb els seus atributs `nif`, `nom` i `data de naixement` emmagatzemada a les

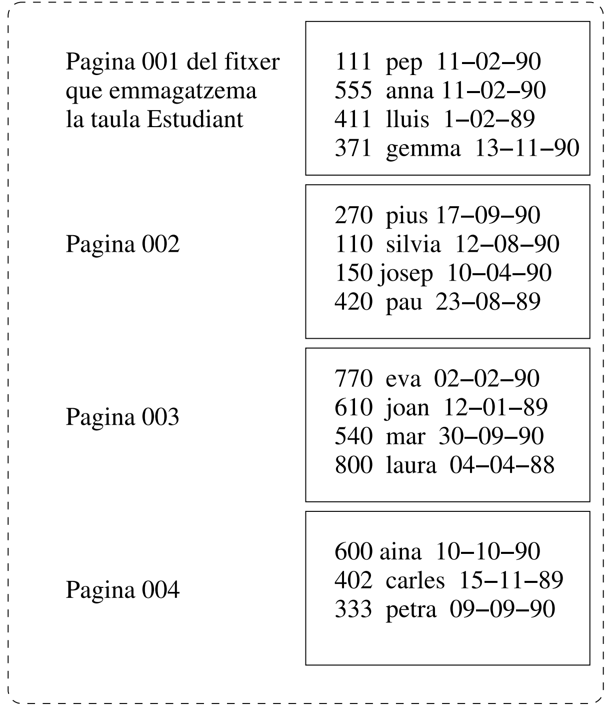


Figura 7.3: Emmagatzemament d'una taula

pàgines d'un fitxer. La figura 7.3 ens en mostra un exemple d'aquesta taula. En particular, el primer estudiant representat té com a `nif`, *111*[2], com a `nom`, *pep*, i com a `data de naixement`, *11-02-1990*.

En aquest exemple, cada pàgina conté 4 registres.

Fixeu-vos bé en aquesta taula i en aquest exemple: la farem servir a la majoria dels exemples d'aquest capítol. En aquests exemples suposarem sempre que:

- La taula `Estudiant` té tres atributs: `nif`, `nom`, `data naixement`.
- El `nif` serà la clau primària de la taula i tindrà tres dígits.
- No la relacionarem amb altres taules (per exemple, amb la taula `Titulació` i, per tant, no inclourem l'atribut clau forana `titol`).
- El SGBD representa la taula `Estudiant` en un únic fitxer amb pàgines de 4 registres (estudiants).

■ ■ ■

[2] D'acord!, no s'ha vist mai un NIF de tres dígits, però ens anirà molt bé en els propers exemples per estalviar espai a les figures.

En resum:
Els registres d'una taula relacional s'emmagatzemen en pàgines d'un fitxer. Quan el SGBD necessita llegir/escriure un d'aquells registres, llegeix/escriu la totalitat de la pàgina allà on es troba.

En aquest context, què passa quan el SGBD vol accedir a la informació d'un registre d'una taula relacional?

Cercant un registre a una taula relacional

Fóra bo que un usuari de la base de dades d'estudiants pogués accedir eficientment a la informació d'un estudiant determinat entre tots els estudiants emmagatzemats a la taula `Estudiant`. Per això, les bases de dades relacionals usen un llenguatge de consultes i manipulació de la informació de la base de dades anomenat SQL. Pensem, per exemple, en tres consultes molt simples que podríem fer sobre la taula `Estudiant`:

1. *Cercar tota la informació de l'estudiant que té NIF 333.*

   Aquesta consulta, usant el llenguatge SQL, es faria d'aquesta manera:

   ```
   1 select * from Estudiant where nif = '333'
   ```

   que es pot interpretar com: *selecciona tots els atributs del registre (o registres) de la taula* `Estudiant` *que té el valor 333 a l'atribut* `nif` *i retorna una llista amb tots ells.*

   En aquest cas, com que l'atribut `nif` és la clau primària, només hi pot haver un estudiant que compleixi la condició.

2. *Cercar tota la informació dels estudiants que s'anomenin* `pep`.

   ```
   1 select * from Estudiant where nom = 'pep'
   ```

3. *Cercar tota la informació dels estudiants que van néixer el 11-02-1990.*

   ```
   1 select * from Estudiant where dataNaix = '02/11/90'
   ```

   Notem que en aquest cas, caldria retornar dos estudiants: el `pep` i l'`anna` van néixer en aquesta data.

Com s'ho farà el SGBD per resoldre la primera consulta?

Si la taula relacional contingués només un centenar d'estudiants no hi hauria gaires dificultats: es podrien llegir seqüencialment totes les pàgines del fitxer que emmagatzemaria la taula d'estudiants fins a trobar el registre que tingués el NIF demanat. El problema és que ja hem dit que les bases de dades les usem típicament quan tenim moltes dades a emmagatzemar: desenes de milers, potser milions! En aquestes condicions, un accés seqüencial al fitxer que conté la taula relacional seria inadmissible.

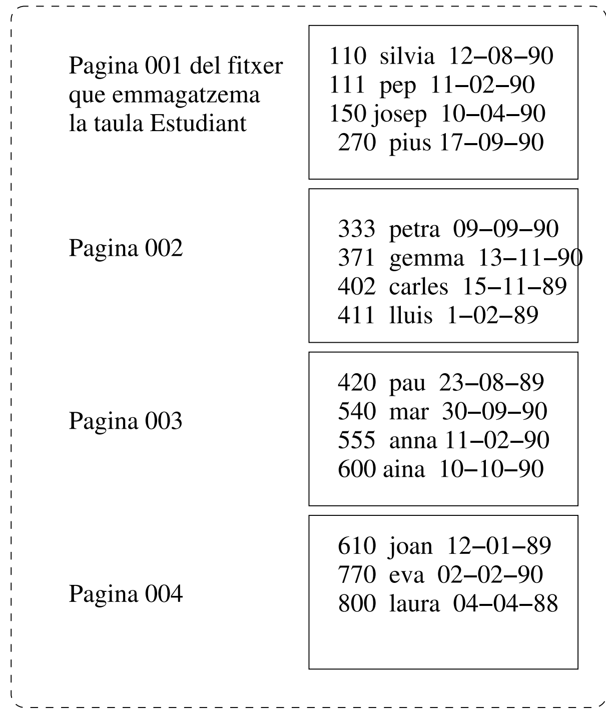


Figura 7.4: Emmagatzemament d'una taula relacional mitjançant un fitxer ordenat per NIF

Què fem, doncs? Una alternativa senzilla seria ordenar físicament el fitxer que conté la taula d'estudiants[3] per ordre de NIF, tal com es mostra a la figura 7.4. Aleshores podríem usar una cerca dicotòmica en disc per tal d'accedir ràpidament a l'estudiant desitjat.

Hi veieu alguna dificultat amb aquesta proposta? Preneu-vos un moment per pensar-hi.

Espero que n'hàgiu vist almenys dues, de dificultats:

- Què passaria si, a més de la consulta per `nif`, volguéssim fer les consultes 2 `i/o` 3 de la llista anterior? Un fitxer ordenat físicament per `nif` no ens ajuda a trobar els estudiants que s'anomenin *pep* ni tampoc els que van néixer en una data determinada. És acceptable poder fer només consultes d'estudiants per `nif`?
- Cada cop que volguéssim fer la inserció d'un nou estudiant hauríem de mantenir l'ordenació física del fitxer i això ens portaria a moure tots els registres amb nif superior al del nou estudiant inserit una posició cap avall al fitxer. Aquest moviment massiu de dades resultaria del tot inacceptable. Una cosa similar passaria amb les eliminacions.

Què fem, doncs?

[3] Recordem que estem suposant, per simplicitat, que aquesta taula relacional s'emmagatzema en un únic fitxer.

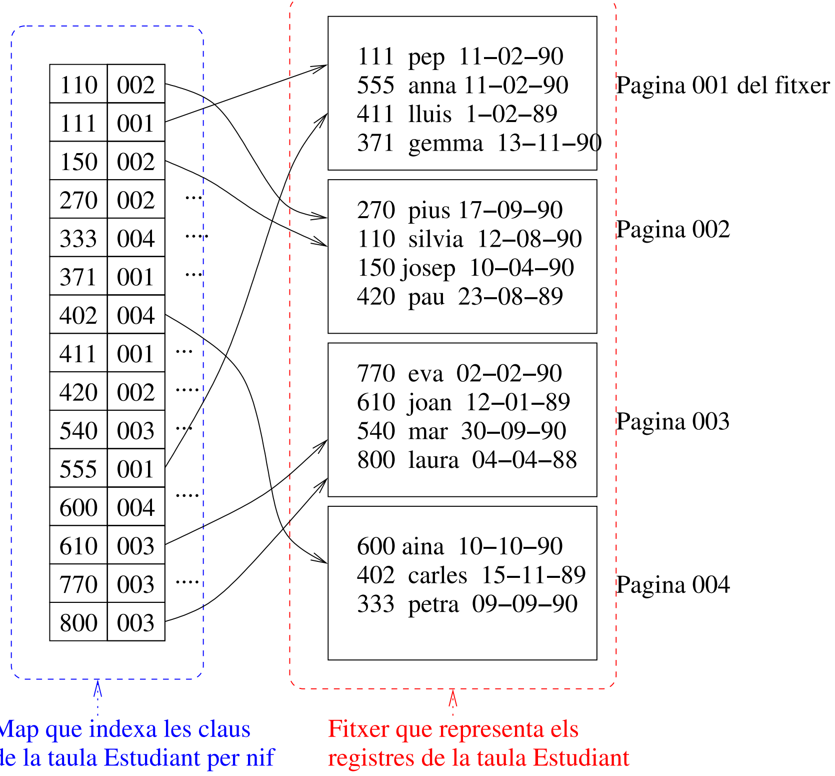


Figura 7.5: Primer exemple d'índex

I si associéssim al fitxer que conté la taula relacional d'estudiants una taula associativa (i.e., de l'estil dels que vam presentar als capítols 4,5 i 6)?

Vegem de quina manera una taula associativa ens pot ajudar a accelerar la cerca d'un registre d'una taula relacional.

7.2 Índexs en taules relacionals

Plantegem un exemple:

Exemple 7.3:

La figura 7.5 mostra la taula `Estudiant` representada físicament mitjançant un fitxer que conté diverses pàgines. Aquesta taula relacional té una taula associativa associat per trobar més fàcilment un determinat estudiant a partir del seu NIF. Hem après al capítol 4 que una taula associativa és una col·lecció de parelles (*clau*, *valor*). Quines són, doncs, les claus i els valors en aquesta taula associativa?

La clau és el NIF d'un estudiant i el valor associat a aquell NIF és **el número de pàgina on es troba l'estudiant amb aquell NIF al fitxer que emmagatzema la taula** `Estudiant`.

Així doncs, l'estudiant amb NIF 110 es troba a la pàgina 001 del fitxer que emmagatzema la taula `Estudiant`, el que té NIF 150 es troba a la pàgina 002 i així successivament.

Sabem que l'accés a una taula associativa per clau és molt eficient; així doncs, l'accés als estudiants de la taula `Estudiant` per NIF serà molt eficient amb l'ajut d'aquesta taula associativa vinculada.

Aquesta taula associativa vinculada al fitxer que conté la taula `Estudiant` s'anomena **índex**.

■ ■ ■

Un **índex** és una taula associativa[a] que està associada a una taula relacional i que serveix per aconseguir un accés més eficient als registres d'aquesta taula relacional a través d'un dels seus atributs.

Un índex té, com a **clau** l'atribut a través del qual es vol fer l'accés eficient a la taula relacional (e.g., NIF) i com a **valor**, un número de pàgina (o una llista de números de pàgines, si aquell atribut no és identificador).
El número de pàgina (`n`) associat a un determinat valor de la clau (`c`) indica que el registre amb `clau = c` (e.g., `nif = 123`) es troba a la pàgina `n` del fitxer que emmagatzema la taula relacional.

[a] i.e., una taula de les que vam presentar als capítols 4,5 i 6.

Cal fer alguns comentaris amb relació als índexs:

- Com passa sovint, el preu que es paga per obtenir l'accés més eficient a la taula relacional és una pèrdua d'espai ja que, evidentment, l'índex ocupa un espai.

- Així i tot, l'índex ocupa un espai típicament molt inferior al del fitxer(s) que emmagatzema la taula relacional a la qual està associada l'índex. Efectivament, una taula relacional conté molts atributs. L'índex només conté l'atribut pel qual s'ha de fer la cerca i el número de pàgina on es troba l'element. En conseqüència, ocuparà molt menys espai.

- Amb els índexs resolem les dues dificultats que plantjàvem a la pàgina 286:

  - Per un costat, podem fer cerques eficients per diferents atributs d'una taula relacional. Com ho farem a l'exemple anterior per fer cerques per *nom d'estudiant*, a més a més de per *NIF*? És clar: definint un *segon índex*, en aquest cas, per nom d'estudiant (vegeu figura 7.6).
  - Per l'altre costat, no hem de patir pels costos que puguin tenir les operacions d'inserció o eliminació d'entrades de l'índex. Al capdavall, un índex és una taula associativa i ens hem passat els capítols 4, 5 i 6 explicant tècniques per

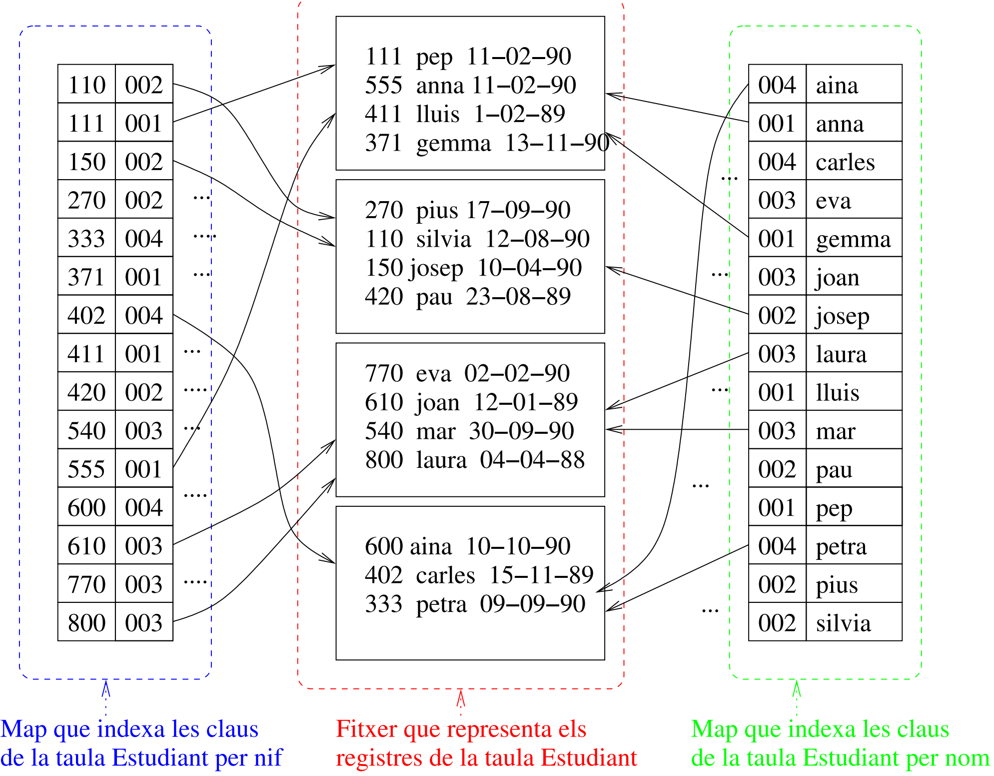


Figura 7.6: Definint un segon índex per nom associat a la taula relacional `Estudiant`

implementar taules associatives eficients (també per a les operacions d'inserció i eliminació de claus).

D'acord... reconec que aquest argument té una petita esquerda. Tot seguit en parlem.

- Haureu notat que hi ha una diferència fonamental entre les taules associatives que vam presentar als capítols 4, 5 i 6 i els índexs que plantegem ara. Quina és? Penseu-hi un moment.

  Efectivament, les taules associatives dels capítols 4 i successius s'emmagatzemaven en memòria. En canvi, els índexs són taules associatives que hauran de ser persistents (com les taules relacionals a què apuntaran) i, per tant, *s'emmagatzemaran al disc*. Potser això ens obligarà a fer algunes precisions respecte de les implementacions que vam veure en aquells capítols.

Doncs, en resposta a la darrera consideració, dedicarem la secció 7.3 a parlar d'implementacions d'índexs. Veurem que la idea és agafar les implementacions de taules associatives que ja coneixem i fer-hi les modificacions escaients per tal que es puguin emmagatzemar en disc de

manera eficient, tot i conservant les propietats de les taules associatives (insercions, eliminacions i, sobretot, consultes eficients i obtenció de claus ordenades en les taules associatives arborescents).

I moltes coses més...

Aquest capítol només pretén contextualitzar mínimament els índexs de les bases de dades per justificar que les taules associatives que hem presentat als capítols 4, 5 i 6, modificades convenientment per fer-les persistents, poden ser útils per fer més eficient l'accés als elements d'una taula relacional.

Ens deixem moltes coses per explicar. Un tast:

- Hem vist índexs **densos** (o sigui, índexs que tenen una entrada per a cada registre de la taula relacional). Però si la taula relacional estigués ordenada (per exemple, per `nif`), podríem definir índexs **no densos** (*sparse*, en anglès) que només haurien de contenir una entrada per al primer registre de cada pàgina.
- Els atributs que hem indexat han estat sempre atributs que no tenien repeticions (el `nif` perquè era clau primària i el `nom` perquè, astutament, no hem posat repeticions). Però també podem indexar atributs que admeten repeticions. Però aleshores, cada entrada de l'índex podrà tenir associada una llista de pàgines on hi ha registres amb aquell valor de l'atribut.
- I moltes coses més...

Però ara, aprofitem les coses que sí que hem après per encetar allò que hem vingut a fer a aquest capítol: com podem usar les taules associatives que vam presentar als capítols anteriors per tal d'implementar índexs i així accedir més ràpidament als registres d'una taula relacional.

7.3 Implementació d'índexs

Hem vist dues famílies d'implementacions astutes de taules associatives: dispersió (vegeu el capítol 5) i arborescents (vegeu el capítol 6).

Doncs en aquesta secció plantejarem dues maneres d'implementar índexs que correspondran a cadascuna de les dues famílies d'implementacions de taules associatives:

- Els arbres B+ (que són uns parents dels arbres B que hem presentat al capítol 6).
- Les funcions de dispersió (que és basen en la mateixa idea que la implementació de taules associatives amb funcions de dispersió que hem presentat al capítol 5).

7.3.1 Arbres B i B+

Una utilització clàssica dels arbres B és per a servir d'índex en una base de dades. Per fer-los servir per a aquest propòsit és interessant fer algunes consideracions. Comencem per l'obtenció ordenada de les claus.

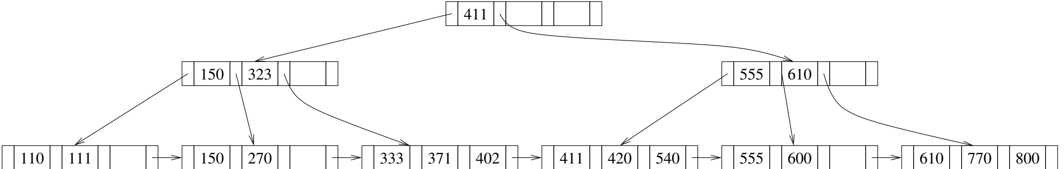


Figura 7.7: Arbre B+

Obtenció ordenada de les claus: arbres B+

Ja hem indicat a 6.2.4 que les estructures arborescents són interessants per implementar taules associatives perquè permeten obtenir els elements de la taula associativa ordenats per clau. La manera d'obtenir les claus en ordre és recórrer l'arbre que representa la taula en inordre (vegeu 6.2.4). Si l'arbre és a la memòria, el seu recorregut en inordre és acceptable. Per contra, si l'arbre està emmagatzemat totalment o parcial al disc, el seu recorregut en inordre pot ser més complicat i, sobretot, més ineficient.

Per resoldre aquesta dificultat van aparèixer els arbres B+.

Els **arbres B+** són una variant dels arbres B que es caracteritzen perquè tenen totes les claus replicades a les fulles i cada fulla està enllaçada a la següent en l'ordre ascendent de les claus.

Igualment, cada fulla pot encadenar-se a l'anterior (en ordre ascendent de claus) i, d'aquesta manera es poden fer recorreguts en ordre ascendent i descendent de clau.

Exemple 7.4:

La figura 7.7 mostra un exemple d'arbre B+ que conté les claus de la taula `Estudiant` de la base de dades que apareix representada a la figura 7.5.

Com ho faríem per tal d'usar aquest arbre B+ com a índex per a aquella taula?

■ ■ ■

Arbres B+ com a índexs

Per usar els arbres B+ com a índexs d'una taula relacional farem el següent:

- La gestió dels arbres B+ (com la dels arbres B) es basa en la lectura i escriptura de nodes d'aquests arbres. Per això, **situarem cada node en una pàgina diferent** (recordeu que les pàgines constitueixen la unitat de lectura/escriptura del SGBD).
- En general, com més gran sigui l'ordre de l'arbre B+ (i, per tant, el seu factor de ramificació), millor serà l'eficiència de les cerques. El fet que un node sigui tan gran

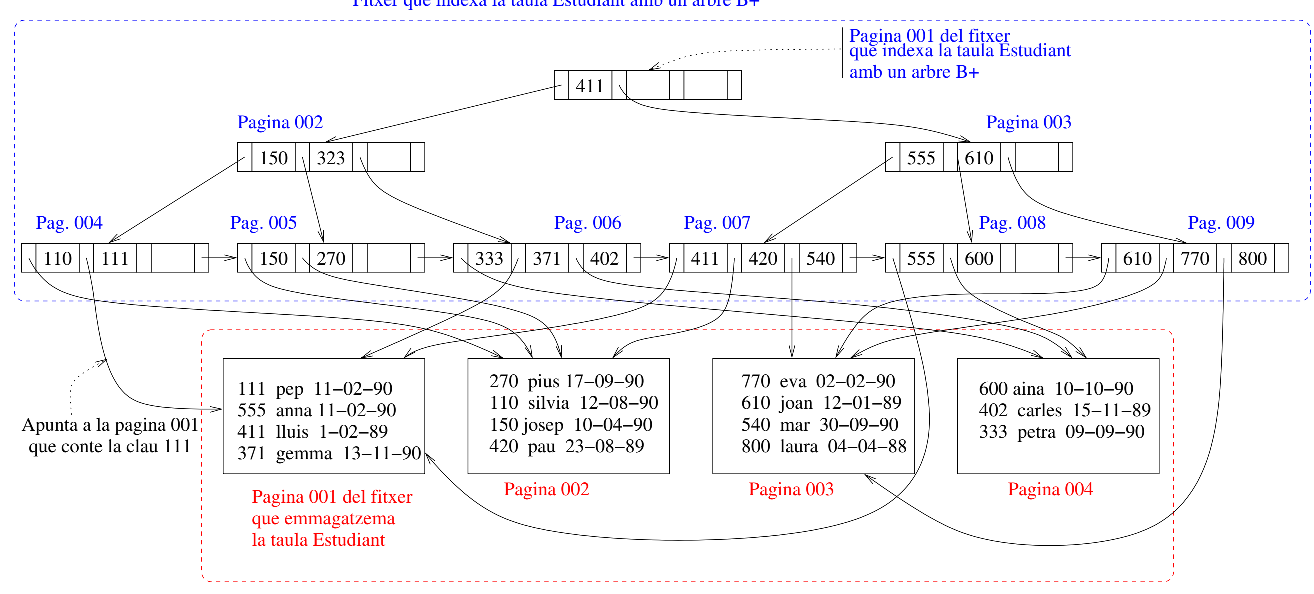


Figura 7.8: Arbre B+ que actua com a índex de la taula `Estudiant`

com una pàgina ens permetrà encabir moltes claus en un node i, per tant, **assolir un factor de ramificació important**.

- Un node intermedi (i també l'arrel) de l'arbre B+ apuntarà a les pàgines que contenen els seus nodes fills. Les fulles de l'arbre B+ apuntaran a les pàgines del fitxer que estan indexant. O sigui, a les pàgines del fitxer que emmagatzema la taula relacional per a la qual s'ha definit l'índex.

Aquestes consideracions es mostren gràficament a la figura 7.8, en l'exemple particular d'un índex sobre la taula `Estudiant` amb la quà estem treballant tota l'estona.

Números típics d'un arbre B+

L'ordre d'un arbre B+ (això és, el nombre màxim de fills que pot tenir un dels seus nodes) pot ser, en situacions reals, més gran que 100. Això dóna lloc a un factor de ramificació considerable que pot reduir el nombre de nivells de l'arbre B+ a tres o quatre per a taules relacionals de milions de registres. Poder trobar una clau llegint, com a màxim, 3 o 4 pàgines de disc (més una cinquena pàgina per accedir finalment a la informació del registre corresponent a aquella clau) és certament un accés quasi tan eficient com l'obtingut amb tècniques de dispersió. Fem uns càlculs senzills per raonar d'on surten aquests números.

Exemple 7.5:

Suposem que volem dissenyar un índex en forma d'arbre B+ per a l'atribut `nom` de la taula `Estudiant`. Suposem que el SGBD gestioni pàgines de 8 KB (la qual cosa és habitual).

Suposem també que l'atribut `nom` té una mida de 30 bytes i suposem, finalment, que necessitem 10 bytes per adreçar les pàgines del fitxer que emmagatzema la taula `Estudiant`.

- Una fulla d'un arbre B+ està formada per n blocs (*clau, adreça de pàgina*) + 1 adreça de la pàgina corresponent a la fulla següent (opcionalment, també pot tenir l'adreça de la pàgina corresponent a la fulla anterior). Si menystenim, per simplificar els càlculs, l'espai ocupat pel punter a la pàgina següent, tenim que una fulla està formada per n blocs de $10+30 = 40$ bytes cadascun. I, per tant, en una fulla hi caben aproximadament $n = \lfloor \frac{8000}{40} \rfloor = 200$ claus.

  Considerem, com a exemple, una fulla de l'arbre B+ de la figura 7.8. Aquesta fulla està formada per 4 parelles (*adreça pàgina, clau*) més una adreça de la pàgina corresponent a la fulla següent.

- Aquests mateixos càlculs valen a un node intermedi. Un node intermedi té n blocs (*adreça de pàgina node fill amb claus menors, clau*) i una darrera adreça de la pàgina que conté les claus més grans que la clau més gran del node. Si menystenim aquesta darrera adreça per simplificar el càlcul, tornem a obtenir, aproximadament, 199 claus i 200 fills. Obtenim, doncs, un factor de ramificació de 200.

Per calcular el nombre de nivells que pot tenir aquest arbre B+, fem encara dues hipòtesis més:

- Suposem que els nodes estan plens, en mitjana, al 50% i que
- la taula `Estudiant` té 2.000.000 de registres (un nombre més que considerable, no trobeu?).

Si cada node està ple al 50%, vol dir que cada node contindrà, aproximadament, 100 claus. Per tant, necessitarem $\frac{2.000.000}{100} = 20.000$ nodes.

I, si el factor de ramificació és aproximadament 100 (100 claus per node i 101 fills), necessitaríem que l'arbre tingués entre 3 i 4 nivells (amb 3 nivells encabiríem $100^3 = 1.000.000$ nodes i amb 4, $100^4 = 100.000.000$ nodes).

Més que suficient.

■ ■ ■

Aquest rendiment dels arbres B+ justifica que, habitualment, els primers dos nivells d'un arbre B+ (que poden ocupar entre 100 i 200 pàgines) s'emmagatzemin a la memòria per fer més ràpid l'accés.
Noteu que si fem això, el SGBD pot accedir a un registre d'una taula relacional que conté milions de registres llegint 2 o 3 pàgines del disc. L'eficiència és molt gran.

Fins aquí l'ús dels arbres B+ com a índexs. Ara, recordem que hi havia una altra manera molt eficient d'implementar taules associatives: les *taules de dispersió* (vegeu capítol 5). Vegem què cal fer per convertir les taules (associatives) de dispersió que vam presentar al capítol 5 en índexs.

7.3.2 Taules de dispersió persistent

Si volem que la implementació de les taules de dispersió que vam presentar al capítol 5 es pugui usar per implementar índexs de taules relacionals, necessitarem dues coses:

1. Emmagatzemar les parelles (*clau, adreça de pàgina*) en un **fitxer** (i no en un vector com fèiem al capítol 5).
2. Tenir un accés al fitxer **directe per posició**, per emular així l'accés directe també per posició que proporcionen els vectors i que és tan necessari per implementar una taula de dispersió: `t[h(k)]`. Efectivament, recordem que la implementació d'una taula de dispersió es basa a col·locar les parelles (*k,v*) a la posició `h(k)` del vector `t` que emmagatzema la taula de dispersió.

Per complir els dos requeriments anteriors, implementarem les taules de dispersió persistents usant *fitxers d'accés aleatori*, com els que es van estudiar a l'assignatura de Programació 2 (vegeu [GG11]). Recordeu que aquells fitxers oferien operacions per accedir-hi per posició:

```
1  void seek(long pos)
```

Aquesta operació posiciona el capçal de lectura/escriptura al byte del fitxer que ocupa la posició `pos`.

Una **taula de dispersió persistent** és una taula de dispersió (més precisament, una *taula associativa de dispersió*) en què les parelles (*clau, valor*) que formen la taula s'emmagatzemen en un *fitxer d'accés aleatori* en lloc d'en un vector.

Les taules de dispersió persistents es poden usar si es necessita un accés directe per clau a unes dades que són persistents i, per tant, es poden usar com a *índexs de bases de dades relacionals*.

Exemple 7.6:

La figura 7.9 mostra una taula de dispersió persistent fent d'índex per `nif` de la taula `Estudiant` que ja ens és familiar. Aquesta taula de dispersió persistent està emmagatzemada en un fitxer d'accés aleatori. Aquest fitxer està compost per pàgines tal com està indicat a la figura. Cada pàgina té capacitat per a n parelles (*clau, adreça pàgina fitxer Estudiant*).

Noteu que n podrà ser molt més gran que el nombre de registres continguts en una pàgina del fitxer que emmagatzema la taula `Estudiant` perquè, típicament, la informació continguda en un registre és molt més voluminosa que una parella (*clau, adreça*). Noteu que, per aquest motiu, el fitxer que conté l'índex ocuparà molt menys espai que el fitxer que conté la taula `Estudiant`.

La manera d'accedir a la informació del registre que té `nif=110` és la següent:

1. Càlcul d'$h(110) = 003$.

   003 és la pàgina del fitxer índex on es troba la clau 110.

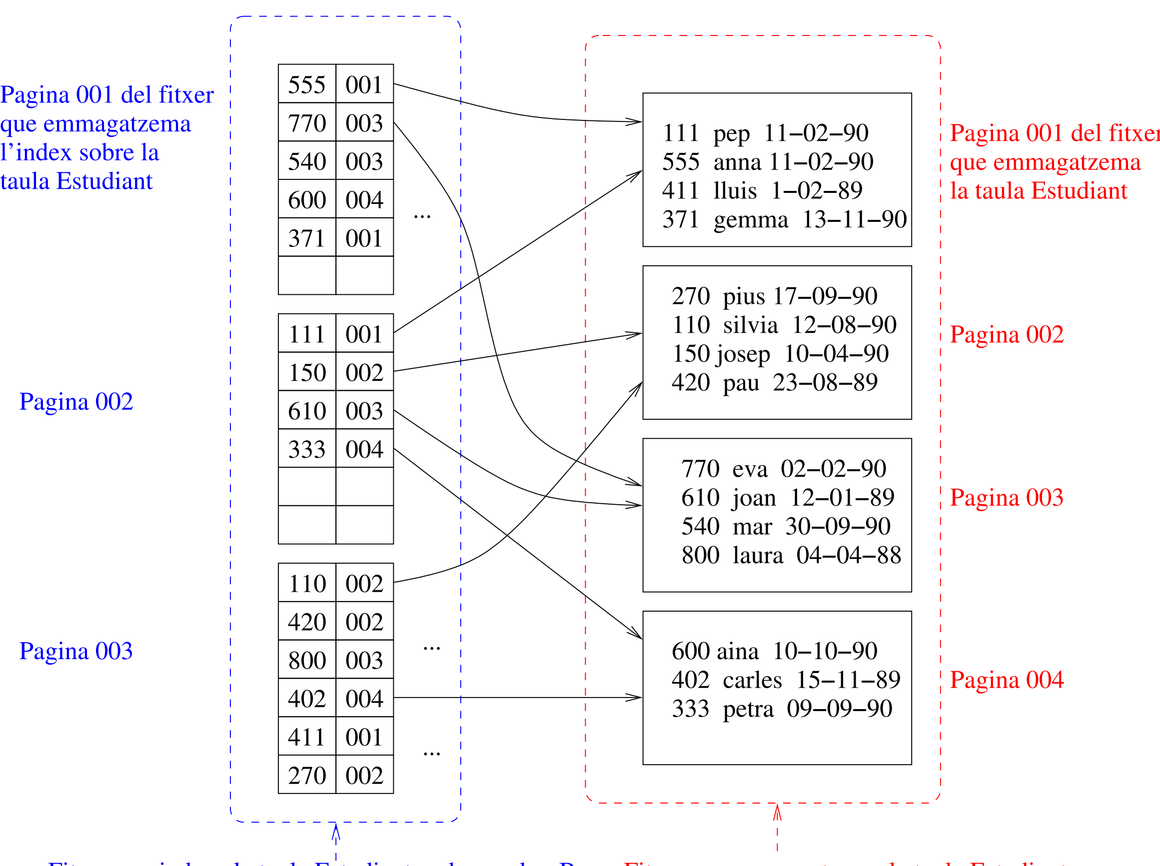


Figura 7.9: Taula de dispersió persistent fent d'índex

2. Càrrega en memòria de la pàgina 003 de l'índex a memòria i localització de la clau 110 i el seu valor associat, 002 que és precisament, la pàgina del fitxer que conté la taula `Estudiant` on es troba el registre amb `nif`= 110.

3. Càrrega en memòria de la pàgina 002 del fitxer que conté la taula `Estudiant` i localització en aquesta pàgina del registre amb `nif`= 110.

■ ■ ■

Noteu que si, a l'exemple anterior, hi afegim un segon índex, pel qual la clau sigui `nom`, aconseguim accedir eficientment a la taula relacional de persones per `nif` i per `nom`.

7.3.3 Criteri d'ús de taules de dispersió persistents i arbres B+ com a índexs d'una taula relacional

Arbres B+

Els índexs implementats com a arbre B+ poden resoldre consultes que requereixin un ordre en els registres retornats. Per exemple,

```
1 select * from Estudiant where nif <= '444'
```

En general, els operadors relacionals <, >, >= són també gestionats eficientment pels arbres B+.

L'operador d'igualtat (=) també és gestionat correctament per un índex en forma d'arbre B+.

Taules de dispersió persistents

Per contra, els índexs implementats com a taules de dispersió persistents no gestionen bé els operadors relacionals. Són adients únicament per a consultes per igualtat de la forma:

```
1 select * from Estudiant where nif = '444'
```

Creació d'índexs per a una taula relacional

Habitualment, el SGBD crea automàticament un índex per a la clau primària de cada taula relacional. A més a més, el SGBD permet que el dissenyador de la base de dades pugui crear més índexs. La comanda SQL que s'usa per crear un índex és, en general, similar a aquesta:

```
1 CREATE INDEX nomEstudiant ON Estudiant USING hash (nom);
```

Aquesta comanda crea un índex *hash* per a la taula `Estudiant` segons la columna `nom`.

7.4 Una proposta d'implementació d'índexs amb taules de dispersió persistents (*)

Ens podem imaginar una taula (associativa) de dispersió persistent (vegeu l'estructura de la figura 7.9(a)) com una col·lecció de parelles *(clau, valor)* que es troben emmagatzemades, no en un vector a la memòria, sinó en un fitxer al disc. El procediment que tindrem per accedir a una d'aquestes parelles, a partir de la seva clau, serà similar al que teníem per accedir a les taules de dispersió en la memòria:

Aplicarem una funció de dispersió a la clau `key` per tal d'obtenir l'índex allà on estarà situada la parella `(key,value)` al fitxer i accedirem a aquell índex.

```
1 int index = indexFor(key.hashCode(),tableCapacity));
```

La diferència és que, en el cas de les taules de dispersió en memòria, l'índex al qual s'ha convertit la clau és *l'índex d'un vector*, mentre que en el cas de les taules de dispersió persistents serà *l'índex d'un fitxer*. I això tindrà unes conseqüències gens banals.

Vegem quines són aquestes conseqüències:

Necessitat de fitxers d'accés directe

Accedir a l'índex `i` del vector `table` és molt fàcil:

```
1 node = table[i];
```

A més a més, sabem que el cost de l'operació `table[i]` és $O(1)$.

Com podem fer per accedir, similarment, a l'índex `i` d'un fitxer amb el mateix cost asimptòtic $O(1)$?

Necessitarem fitxers que disposin d'una operació per accedir eficientment a la posició que desitgem del fitxer. Aquests fitxers existeixen (si no, els apunts no contindrien aquesta secció ;-)). S'anomenen *fitxers d'accés aleatori* (o d'accés directe) i podeu trobar-ne una referència a [GG11], capítol 4: "Manejo básico de archivos en Java". Vegem un exemple d'ús d'aquest tipus de fitxers.

Exemple 7.7: Lectura en un fitxer d'accés aleatori

Per llegir els `k` bytes consecutius que es troben a partir de la posició `n` del fitxer d'accés aleatori `f`, farem:

```
1  int k = 20;
2  long n = 300;
3  try {
4      RandomAccessFile f = new RandomAccessFile("myFile.dat","r");
5      byte[] a = new byte[k];
6
7      f.seek(n);
8      f.read(a);
9      f.close();
10 } catch (IOException e) {
11     e. printStackTrace();
12 }
```

■ ■ ■

Vegem ara un altre exemple, en aquest cas, d'escriptura:

Exemple 7.8: Escriptura en un fitxer d'accés aleatori

Per escriure els `k` bytes que es troben al vector `a` al fitxer d'accés aleatori `f` a partir de la posició `n`, farem:

```
1  int k = 20;
2  long n = 300;
3  try {
4      RandomAccessFile f = new RandomAccessFile("myFile.dat","rw");
5      byte[] a = new byte[k];
6      fillWithData(a); // Omple el vector a amb k bytes d'informaci\'o.
7
8      f.seek(n);
9      f.write(a);
10     f.close();
11 } catch (IOException e) {
12     e. printStackTrace();
13 }
```

■ ■ ■

Us haureu adonat que, als exemples anteriors, quan parlem de *posicions* estem parlant de *bytes*. O sigui, el que un fitxer d'accés aleatori es deixa fer fàcilment és accedir al seu byte `n`.

Però la funció de dispersió aplicada a una clau `k` no ens retorna el numero de byte al qual hem de posar la parella `(k, v)` sinó, més aviat, el número de registre dins del fitxer allà on li correspon anar a aquella parella. Dit d'una altra manera, voldrem que el nostre fitxer estigui *indexat per registres, no per bytes*.

Per exemple, si considerem que cada registre del fitxer conté una parella (clau, valor), $h(k) = 3$ no ens vol dir que k hagi d'anar al byte número 3 del fitxer sinó més aviat al registre 3, o sigui, al quart registre del fitxer. A cadascun dels tres registres anteriors (0, 1 i 2) hi podrà haver també una parella (clau, valor).

Ara, per posicionar-nos a un determinat registre, haurem de fer:

```
1      ...
2      f.seek(n * SIZE_OF_RECORD);
3      f.read(a);
4      ...
```

> I aquest criteri, és clar, ens obliga que tots els registres tinguin la mateixa mida `SIZE_OF_RECORD`.

A l'exemple que acabem de donar un registre era una parella (clau, valor). Mmm... aquesta no és la millor idea en el cas d'una taula de dispersió persistent... Això ens proposa una

nova pregunta: què és, en el cas d'una taula de dispersió persistent, un *registre*? Continueu llegint, sisplau...

Fitxers de registres-contenidors

Considereu la proposta següent per veure què pot ser un registre en el context d'una taula de dispersió persistent:

> **Registre-contenidor** (en anglès, *bucket*):
> Cada *índex* (o *posició*) del fitxer que emmagatzema la taula de dispersió persistent contindrà un registre que anomenarem **registre-contenidor**. El fitxer disposarà de B registres-contenidors, numerats des del 0 fins al $B - 1$.
>
> Un *registre-contenidor* té la capacitat per contenir una col·lecció d'`M` parelles *(clau, valor)*. (Habitualment, aquesta col·lecció de parelles *(clau, valor)* es representa com un vector (`pairs`) d'`M` elements.)
>
> El conjunt de parelles *(clau, valor)* contingudes en un *registre-contenidor* en un moment determinat són sinònimes.
>
> O sigui, l'aplicació de la funció de dispersió a cadascuna de les claus emmagatzemades al *registre-contenidor* dóna com a resultat l'índex d'aquell *registre-contenidor*.

Si la taula de dispersió (que representem com a fitxer) està ben dimensionada i la funció de dispersió (la composició de `hashCode()` i `indexFor(...)`) dispersa bé les claus, el nombre de claus sinònimes a la taula hauria de ser petit; en general, menor que `M` (el nombre de parelles (*clau,valor*) sinònimes que caben en un registre-contenidor). Però pot passar que la taula no estigui ben dimensionada, que la funció de dispersió no dispersi prou bé les claus o, fins i tot, que en una taula que funciona bé globalment, algun dels índexs del fitxer rebi més d'`M` claus sinònimes.

En definitiva, és possible que la funció de dispersió envïi una clau a un *retistre-contenidor* que ja estigui ple. Què fem aleshores?

Gestió de claus sinònimes:

Una possibilitat molt natural és ampliar el fitxer més enllà de l'índex `B` amb nous registres-contenidors que contindran les claus que no caben als registres-contenidors corresponents als `B` primers índexs del fitxer.

Aquesta part del fitxer (des de l'índex `B` a l'índex `C`), l'anomenarem **zona d'excedents** per diferenciar-la de la primera part (índexs 0...`B-1`) o **zona principal**.

La gestió de les claus sinònimes usant aquesta zona d'excedents funcionaria de la manera següent:

Gestió de les claus sinònimes:

- Si un registre-contenidor `buc` de la zona principal és ple, la resta de claus sinònimes que no càpiguen a `buc` se situaran al registre-contenidor de la zona d'excedents que té l'índex indicat a l'atribut `next` de `buc` (`buc.next`).
- Si aquest registre-contenidor `buc.next` també s'omplís, la resta de claus sinònimes es trobarien a `buc.next.next`. I així successivament fins que s'acabin les claus sinònimes o ja no quedin més registres-contenidors lliures a la zona d'excedents.

  D'aquesta manera es formarà una llista enllaçada de registres-contenidors amb claus sinònimes. Cada índex (i.e., registre-contenidor) de la zona principal és l'inici d'una d'aquestes llistes de sinònims.

- El registre-contenidor que ocupa l'índex `B` s'usarà únicament per saber quin és el primer registre-contenidor lliure de la zona d'excedents.

  Cada registre-contenidor lliure `freeBuc` de la zona d'excedents contindrà l'índex del proper registre-contenidor lliure d'aquella zona a `freeBuc.next`.

  D'aquesta manera es formarà una llista enllaçada de registres-contenidors lliures a la zona d'excedents.
- Si, com a conseqüència d'una eliminació d'una clau de la taula, un registre-contenidor de la zona d'excedents queda lliure, s'ha de retornar a la llista de registres-contenidors lliures de la zona d'excedents.

Les figures 7.10 i 7.11 mostren gràficament aquestes idees.

A la figura 7.11, podem veure que l'eliminació de la clau `K9` provoca que el registre-contenidor 8 es quedi sense cap element i, per tant, retorni a la llista de registres-contenidors lliures dins la zona d'excedents.

El procediment per aconseguir això consisteix a convertir el registre-contenidor 8 en el primer de la llista de lliures. El que fins ara era el primer (registre-contenidor 5) passa a ser el següent del 8 (el nou primer).

Dues consideracions per acabar aquest apartat:

- Noteu que aquesta organització és una organització típica de dispersió oberta: cada clau té associat un índex del vector/fitxer i a partir d'aquell índex obtenim, encadenats, tots els sinònims d'aquella clau. Més particularment, és una organització similar a la de dispersió oberta amb zona d'excedents presentat a 5.5. Una diferència important amb aquella és que, en l'organització presentada a 5.5, cada índex del vector contenia un únic sinònim. En el cas de la dispersió persistent, cada índex del fitxer és un registres-contenidors que conté molts sinònims. Per què us sembla que ho fem així?

  El motiu principal és perquè els accessos al disc són molt més lents que els accessos a la memòria. Per això procurem, amb un sol accés a un índex del fitxer, obtenir un bon grapat de sinònims. Una alternativa molt pitjor fóra que en cada índex hi hagués un

Situacio inicial

Zona	Index					next
ZONA PRINCIPAL	0	free	free	free	free	−1
	1	K10 V10 busy	K11 V11 busy	free	free	−1
	2	K1 V1 busy	K2 V2 busy	K3 V3 busy	K4 V4 busy	6
	3	free	free	free	free	−1
ZONA D'EXCEDENTS	4					5
	5	free	free	free	free	7
	6	K5 V5 busy	K6 V6 busy	K7 V7 busy	K8 V8 busy	8
	7	free	free	free	free	−1
	8	K9 V9 busy	free	free	free	−1

	pairs[0]	pairs[1]	pairs[2]	pairs[3]	next
key					
value					
free					

3=M−1

B=4
C=9
M=4

Llista de sinonims:{2,6,8}
Llista de buckets lliures:{5,7}
L'atribut next de l'index B marca el primer index de la llista de lliures.

K1,...,K9 son claus sinonimes:
indexFor(K1.hashCode(),4)=...=
indexFor(K9.hashCode(),4)

Figura 7.10: Representació de les taules de dispersió persistents

sol sinònim i, per arribar a la clau que cerquem, haguéssim de generar operacions de lectura a força índexs diferents.

I implementar una taula de dispersió persistent amb una dispersió tancada (adaptant a fitxers el que s'explica a 5.4)? Tindria sentit? Fóra adient?

- La discussió que hem fet en aquest apartat que ara acabem ha servit per raonar que veuríem els fitxers compostos de registre-contenidor, però ja hem dit que als fitxers d'accés aleatori s'hi accedeix per bytes:

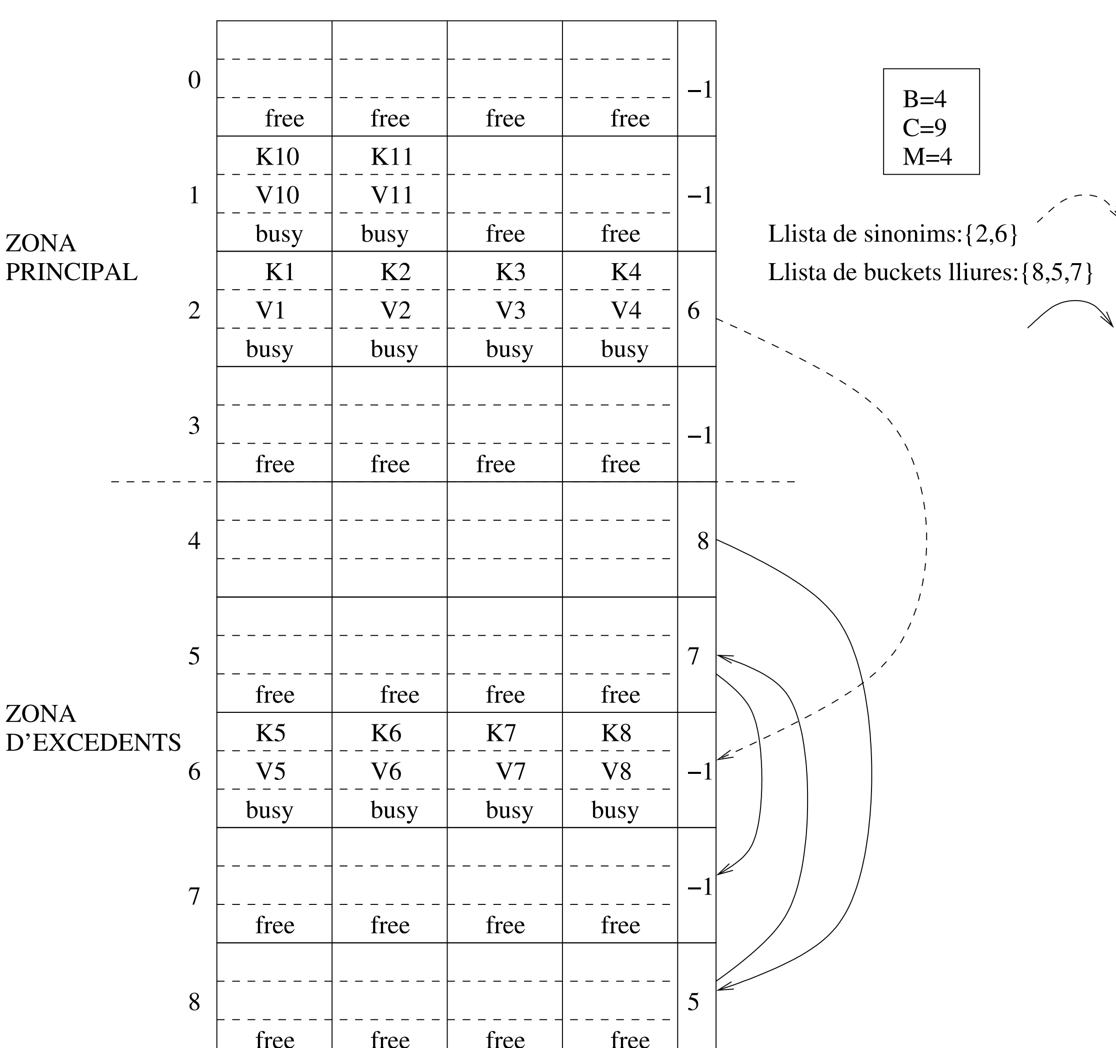


Figura 7.11: Taula de dispersió persistent de la figura 7.10 després de l'eliminació de `K9`

```
1       f.seek(nbyte);
```

on `nbyte` fa referència a un índex, *expressat en bytes* del fitxer. La manera més fàcil per accedir eficientment a un *índex expressat en registres-contenidors* és fer que tots els registres-contenidors tinguin la mateixa mida en bytes (tal com passa amb els vectors). D'aquesta manera l'accés al registre-contenidor `i` es farà:

```
1       f.seek(i * bucketSize);
```

on bucketSize és la mida, en bytes, d'un registre-contenidor.

Així doncs,

> Tots els registres-contenidors tindran la mateixa mida.

L'ús de fitxers per tal de representar taules de dispersió persistents ens ha portat a necessitar **fitxers d'accés aleatori** i que els registres fossin **registres-contenidors**. I encara ens portarà a una tercera cosa: haurem d'empaquetar aquests registres...

Necessitat d'empaquetar registres

Voldrem que les taules de dispersió persistents tinguin, si fa no fa, la mateixa flexibilitat que les HashMap<K, V>. O sigui, virtualment, qualsevol classe podrà actuar com a clau i també com a valor. Per tant, haurem d'estar en condicions d'emmagatzemar al fitxer que representarà la taula persistent objectes de virtualment qualsevol classe.

Malauradament, un fitxer d'accés aleatori no entén d'objectes: és un fitxer binari que podem imaginar com una seqüència de bytes. Per tant, a un fitxer d'accés aleatori no hi podem llegir/escriure objectes directament: hi llegirem i escriurem *vectors de bytes*.

Encara pitjor: les classes, les instàncies de les quals voldrem posar al fitxer, poden estar formades per molts atributs de diferents tipus. Alguns d'aquests atributs poden ser referències (això és: adreces de memòria que assenyalen la localització d'un objecte d'un cert tipus).

Exemple 7.9:

En aquesta classe Person tots els seus atributs llevat d'age són referències.

```
1    class Person {
2        String nif;          //referencia a un objecte de class String
3        String name;         //referencia a un objecte de class String
4        Company worksFor;    //referencia a un objecte de class Company
5        int age;             //valor enter
6    }
```

■ ■ ■

Ja ens podem imaginar que fóra bastant inútil posar a un fitxer la referència que ocupa un altre objecte a la memòria. Les referències a la memòria no són persistents: els objectes que contenen es desallotgen quan s'acaba l'aplicació i això fa impossible que, a partir d'aquell moment, es pugui recuperar l'objecte que ocupava aquella referència.

Males notícies, doncs: voldrem crear una taula persistent de (per exemple) persones i, per tant, voldrem posar Persons a un fitxer d'accés aleatori, però aquest fitxer només entendrà que hi posem seqüències (vectors) de bytes. I, per descomptat, de cap manera hi podrem posar les referències a altres objectes que un determinat objecte pugui contenir. Com ho resoldrem això?

Recordeu el que vau estudiar a Programació 2? Si ho feu, ja haureu endevinat que el que caldrà fer és:

1. Convertir a *vectors de bytes* cada objecte que vulguem inserir al fitxer que representarà la taula de dispersió persistent. D'aquest procés se'n diu *empaquetar* un objecte.

   Quan l'objecte estigui empaquetat (i, per tant, convertit en un vector de bytes), ja es podrà inserir al fitxer.

2. Per recuperar l'objecte, llegirem el vector de bytes de l'índex del fitxer on el vam col·locar i, aleshores, el reconvertirem en un objecte com l'original. D'aquest procés se'n diu *desempaquetar*.

   Un cop l'objecte estigui desempaquetat ja el podrem usar normalment al programa.

A [GG11] es defineix la classe `PackUtils` que dóna operacions per empaquetar i desempaquetar tipus primitius (`int`, `boolean`, `long`, `double`...) i `Strings`. Aquí usarem extensivament aquesta classe i li afegirem alguna eina més per tal d'empaquetar i desempaquetar objectes de tipus arbitraris. En particular, recordem que el procés d'empaquetament haurà d'incloure convertir en vector de bytes els atributs que siguin referències a altres objectes (i novament reconvertir-los a referències en el procés contrari de desempaquetament).

L'eina bàsica que proposem per fer això és la interfície `Packable`. Presentem-la.

7.4.1 La interfície Packable

Als fitxers d'accés aleatori hi posarem vectors de bytes; per tant, els objectes que vulguem posar al fitxer que representa la taula de dispersió hauran de ser *convertibles en vectors de bytes*.

> Anomenarem `Packable` al tipus que tindran els objectes que són convertibles en *vectors de bytes*.
>
> El tipus `Packable` es definirà com una interfície.

Una taula de dispersió persistent és, com qualsevol altra taula associativa, una col·lecció de parelles *(clau, valor)*. Així doncs, el fitxer que representarà la taula haurà de contenir claus i valors. Per tant:

> En una taula de dispersió persistent amb claus de tipus `K` i valors de tipus `V`, `K` i `V` hauran de ser de tipus `Packable`.

La interfície `Packable` la definim de la manera següent:

Llistat 7.1: La interfície `Packable`

```
1 public interface Packable extends Cloneable {
2     byte[] pack() throws PackingException;
3     void unpack(byte[] record, int offset) throws PackingException;
4     int getPackedSize() throws PackingException;
5     Object clone() throws CloneNotSupportedException;
6 }
```

El quadre següent especifica aquestes operacions:

`byte[] pack()` Retorna un vector de bytes que conté la versió empaquetada d'aquest objecte. **Llença:** una `PackingException` si hi ha algun problema en fer l'empaquetament.
`void unpack(byte[] record, int offset)` Desempaqueta sobre aquest objecte la informació continguda al vector `record` a partir de l'índex `offset`. S'ha de complir: `unpack(obj.pack(),0) == obj`. **Llença:** `PackingException` si hi ha algun problema en fer el desempaquetament.
`int getPackedSize()` Retorna el nombre de bytes que ocupa aquest objecte empaquetat. **Llença:** `PackingException` si hi ha algun problema en fer el càlcul de la mida de l'objecte empaquetat.
`Object clone()` Retorna una referència a una còpia d'aquest objecte. **Llença:** `CloneNotSupportedException` si aquest objecte no es pot clonar.

Suposem que volem crear una taula de dispersió persistent en què les claus siguin cadenes de caràcters i els valors, persones. Com ja hem dit que tant les claus com els valors haurien de ser `Packable`, haurem de crear les classes `PackableString` i `Person`. Els dos exemples següents n'ofereixen una proposta de disseny.

Exemple 7.10: La classe `PackableString`

```
1  public class PackableString implements Packable {
2      final int DEFAULT_MAX_SIZE = 20;
3      int maxSize;
4      String st;
5
6      public PackableString(int max, String s) {
7          if (max<=0)
8              maxSize = DEFAULT_MAX_SIZE;
9          else
10             maxSize = max;
11         st = s;
12     }
13
14     @Override
15     public int hashCode() {
16         return st.hashCode();
17     }
18
19     @Override
20     public boolean equals(Object obj) {
21         PackableString ps = (PackableString) obj;
22         return ps.st.equals(this.st) && (ps.maxSize == this.maxSize);
23     }
```

```
24
25     @Override
26     public Packable clone() throws CloneNotSupportedException {
27         PackableString aux = null;
28         aux = (PackableString) super.clone();
29         aux.st = new String(st);
30         aux.maxSize = maxSize;
31         return aux;
32     }
33
34     public byte[] pack() throws PackingException {
35         try {
36             byte buffer[] = new byte[getPackedSize()];
37             PackUtils.packLimitedString(st, maxSize, buffer, 0);
38             PackUtils.packInt(maxSize, buffer, maxSize * 2);
39             return buffer;
40         } catch (Exception e) {
41             throw new PackingException();
42         }
43     }
44
45     public void unpack(byte[] record, int offset)
46             throws PackingException {
47         try {
48             st = PackUtils.unpackLimitedString(maxSize, record, offset);
49             maxSize = PackUtils.unpackInt(record, offset + maxSize * 2);
50         } catch (Exception e) {
51             throw new PackingException();
52         }
53     }
54
55     public int getPackedSize() {
56         return maxSize * 2 + 4;
57     }
58 }
```

Comentaris:

- Notem la redefinició de `hashCode()` i `equals(...)` atès que volem usar aquesta classe `PackableString` com a clau d'una taula de dispersió persistent (vegeu la secció 5.7.4).

  Noteu també com la redefinició de `hashCode()` delega en el hashCode de la classe `String`. Segons l'especificació de l'API del Java, el `hashCode` per a l'objecte `String` `s` es calcula fent:

  `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]` (on `n` és la longitud de `s`).

  Aquest càlcul és consistent amb les propietats de `hashCode()` que vam presentar a 5.7.4.

- Podem apreciar com `pack()` i `unpack(...)` deleguen en les respectives operacions d'empaquetament i desempaquetament ofertes per `PackUtils`. Recordeu que aquesta classe es defineix a [GG11]. Repasseu-la, sisplau.

- Pel que fa a `getPackedSize()` (vegeu les línies 55-57), noteu que la mida en bytes d'un `int` és 4 i la d'un `char`, 2.

■ ■ ■

Exemple 7.11: La classe `Person`

En la hipòtesi que vulguem crear una taula de dispersió persistent on la clau sigui un `PackableString` (per exemple, un NIF) i el valor, una `Person`, necessitarem que `Person` sigui també `Packable`. Mostrem aquí simplement el codi de les operacions `pack()`, `unpack(...)` i `packedSize()` de `Person`.

```
1  public class Person implements Packable {
2      private final int MAX_CAR_NIF = 9;
3      private final int MAX_CAR_NAME = 30;
4
5      public String nif;   //maxLength= 9 caracters: --->18 bytes
6      public String name;  //maxLength = 30 caracters:--->60 bytes
7      public int age;      //4 bytes
8
9      public byte[] pack() throws PackingException {
10         try {
11             byte[] buffer = new byte[getPackedSize()];
12
13             PackUtils.packLimitedString(nif, MAX_CAR_NIF, buffer, 0);
14             PackUtils.packLimitedString(name, MAX_CAR_NAME, buffer,
15                                         MAX_CAR_NIF * 2);
16             PackUtils.packInt(age, buffer,
17                               MAX_CAR_NIF * 2 + MAX_CAR_NAME * 2);
18             return buffer;
19         } catch(Exception e) {
20             throw new PackingException();
21         }
22     }
23
24     public void unpack(byte[] record, int offset)
25             throws PackingException {
26         try {
27             nif = PackUtils.unpackLimitedString(
28                     MAX_CAR_NIF, record, offset);
29             name = PackUtils.unpackLimitedString(
30                     MAX_CAR_NAME, record, offset + MAX_CAR_NIF * 2);
31             age = PackUtils.unpackInt(record,
32                     offset + MAX_CAR_NAME * 2 + MAX_CAR_NIF * 2);
33         } catch (Exception e) {
34             throw new PackingException();
35         }
36     }
37
38     public int getPackedSize() {
39         return MAX_CAR_NAME * 2 + MAX_CAR_NIF * 2 + 4;
40     }
41 }
```

■ ■ ■

7.4.2 Representació de la taula de dispersió persistent

La classe `PHashMap<K, V>`

Modelitzarem la taula de dispersió persistent mitjançant la classe `PHashMap<K, V>`. El llistat 7.2 mostra els aspectes més importants de la representació d'aquesta classe. Després del llistat el comentem.

Llistat 7.2: La classe `PHashMap<K, V>`

```
1  public class PHashMap<K, V> extends AbstractMap<K, V>
2                              implements Map<K, V> {
3
4      private static final int MAX_DEFAULT_BUCKETS = 16;
5      private static final  int BUCKET_CAPACITY = 16;
6      private static final int MAX_DEFAULT_FILE_CAPACITY = 40;
7
8      private RandomAccessFile table;
9      private int tableCapacity;
10         //Capacitat de la zona principal de la taula (en els registres)
11     private int fileCapacity;
12         //Capacitat de la zona principal i d'excedent
13         //de la taula (en els registres)
14
15
16     private int bucketSize;
17         //mida del registre-contenidor empaquetat (en  bytes).
18
19     private static Packable kPrototype;
20     private static Packable vPrototype;
21     ...
22 }
```

Comentaris:

- La classe `PHashMap<K, V>` modelitza una taula de dispersió persistent amb claus de tipus `K` i valors de tipus `V`.

  Com era d'esperar, aquesta classe implementa `Map<K, V>`. A més a més, per tal de no haver-nos de preocupar per la implementació de tot un seguit d'operacions, `PHashMap<K, V>` hereta d'`AbstractMap<K, V>` (vegeu la secció 4.3.1).

- Representem la taula, tal com hem indicat a les seccions anteriors, com un `RandomAccessFile`, que anomenarem `table`.

  Considerarem, conceptualment, el fitxer `table` com *un fitxer de registres-contenidors* que està dividit en una zona principal i una zona d'excedents.

- Hi ha uns paràmetres que completen la definició d'aquest fitxer:

- `tableCapacity`: nombre de registres-contenidors de la zona principal del fitxer que representa la taula (0... `tableCapacity` −1).

  El valor per defecte d'aquest paràmetre és `MAX_DEFAULT_BUCKETS_MAIN_AREA = 16`.
- `fileCapacity`: nombre de registres-contenidors totals del fitxer que representa la taula (0... `fileCapacity` −1).

  Per tant, la zona d'excedents s'estén entre els registres: (`tableCapacity`... `fileCapacity` −1).

  El valor per defecte d'aquest paràmetre és `MAX_DEFAULT_BUCKETS_FILE = 40`.
- `bucketSize`: nombre de bytes que ocupa un registre-contenidor.
- `BUCKET_CAPACITY`: el nombre de parelles (*clau,valor*) que pot contenir un registre-contenidor. El seu valor és 16.

La classe `Bucket<K, V>`

El fitxer d'accés aleatori que representa una taula de dispersió persistent té registres-contenidors com a registres. Un registre-contenidor és una col·lecció de parelles *(clau,valor)*, amb la clau de tipus genèric `K` i el valor, de tipus també genèric `V`.

Ja és hora que definim la classe `Bucket<K, V>`:

```
1  public class PHashMap<K, V> extends AbstractMap<K, V>
2                              implements Map<K, V> {
3      ...
4      private static class Bucket<K, V> implements Packable {
5          Entry<K, V>[] pairs;
6          int npairs;
7          int next;
8
9          Bucket() throws Exception {
10             pairs = new Entry[BUCKET_CAPACITY];
11             for (int i=0;i<BUCKET_CAPACITY;i++) {
12                 pairs[i] = new Entry<K, V>();
13             }
14
15             npairs=0;
16             next=-1;
17         }
18     }
19     ...
20 }
```

Comentaris:

- Definim `Bucket<K, V>` com una classe privada, estàtica, interna a `PHashMap`. És la mateixa estratègia que vam usar al capítol 2 per definir la classe `LinkedList.Entry<K, V>`.
- Els enregistraments del fitxer que representa la taula són de classe `Bucket<K, V>`. Recordem que, per tal de poder inserir enregistraments en un fitxer d'accés aleatori

els hem d'empaquetar (vegeu 7.4 i 7.4.1). Així doncs, la classe `Bucket` implementa la interfície `Packable`.

- Un registre-contenidor el representem com:
  - `pairs`: un vector de parelles *(clau, valor)* sinònimes. Aquestes parelles estan modelitzades per la classe `Entry<K, V>`.

    La longitud del vector `pairs`, això és, el nombre de parelles que es poden encabir en un registre-contenidor, coincideix amb la constant `BUCKET_CAPACITY`.
  - `npairs`: el nombre de parelles que hi ha al vector en un moment determinat. Inicialment, és clar, hi ha zero parelles al vector.
  - `next`: el següent en la llista de registres-contenidors amb sinònims o en la de registres-contenidors lliures (segons sigui el cas: vegeu la figura 7.10).

    El final de la llista de sinònims o de lliures ve donat per `next = -1`. Així inicialitzem aquest atribut a cada registre-contenidor a l'inici.

Hem dit que la classe `Bucket` havia d'implementar la interfície `Packable`. Fem-ho.

La classe `Entry<K, V>`

> Una `Entry<K, V>` és essencialment una parella (*clau, valor*), on la clau és de tipus genèric `K` i el valor, de tipus genèric `V`. A vegades anomenem a les `Entry`s, *nodes*.
>
> A més a més de la parella (*clau, valor*), afegirem a la representació d'`Entry<K, V>` un atribut booleà que indicarà si el node està lliure o no (o sigui, si la clau i el valor es refereixen a elements autèntics de la taula).

```
1 private static class Entry<K, V> implements Map.Entry<K, V>, Packable {
2     K key;
3     V value;
4     boolean free;
5     ...
6 }
```

Comentaris:

- `Entry<K, V>` implementa `Map.Entry<K, V>`; per tant, haurà d'implementar les operacions: `getKey()`, `getValue()`, `setValue(...)`. Podeu implementar-les vosaltres mateixos.

7.5 Racó lingüístic

En aquesta secció comentem els termes tècnics usats en aquest capítol que no estan estandarditzats al diccionari de l'Institut d'Estudis Catalans, a Termcat o a [CM94].

- *Taula relacional. Taula associativa*.

  Aquests dos termes han estat introduïts al racó lingüístic del capítol 4 per resoldre l'ambigüitat inherent al terme *taula*. Els recordem en aquest capítol perquè se'n fa un ús extensiu.

- *Registre-contenidor*.

  Usem aquest terme com a traducció de l'anglès *bucket*. Si no hi ha ambigüitat es pot simplificar per *contenidor*, però com hem definit al capítol 2 *contenidor* com a sinònim d'*estructura de dades*, preferim proposar *registre-contenidor*.

  No hem trobat ni a Termcat ni a [CM94] cap traducció estàndard per *bucket*.

- *Taula de dispersió persistent*.

  L'usem com a traducció de l'anglès *persistent hash map*. Més precisament, el terme complet és *taula associativa de dispersió persistent*. El simplifiquem perquè, en general, quan usem el terme *taula de dispersió* ens referim a taules associatives i no a taules relacionals.

- *Byte*.

  Aquest terme sí que és estàndard i apareix al diccionari de l'Institut d'Estudis Catalans. Notem aquí, simplement, que preferim usar aquest al també normatiu *octet* que és d'ús estrany a la comunitat d'enginyeria de programari.

Bibliografia

[CM94] Cervera A.,Merenciano, J.M.:*Diccionari de termes informàtics*. Servei de llengües i terminologia de la UPC. 1994.

[JB08] Bloch, J:*Effective Java*. Addison Wesley, 2008.

[GG11] Gimeno, JM; González, JL:*Apunts de programació 2*.

[IEC] Diccionari de l'Institut d'Estudis Catalans. `http://dlc.iec.cat/`. Últim accés 20-11-13.

[JAPI] API de Java d'Oracle. `http://download.oracle.com/javase/6/docs/api/`. Últim accés 19-09-13.

[JTuto] Tutorial de Java d'Oracle. `http://download.oracle.com/javase/tutorial/`. Últim accés 19-09-13.

[NW06] Naftalin, M; Wadler, P:*Java Generics and Collections*. O'Reilly, 2006.

[OJDK] Projecte *OpenJDK* `openjdk.java.net`. Últim accés 19-10-13.

[TermC] Web del Centre de terminologia de la llengua catalana. `http://www.termcat.cat/ca/`. Últim accés 29-11-13.

Apèndix A

Excepcions

Una excepció és un esdeveniment (sovint algun tipus d'error) que trenca la seqüència d'instruccions normals de l'aplicació.

Exemples:

- Una aplicació fa una divisió per zero.
- Una aplicació intenta accedir a una posició d'un vector fora del seu rang de definició.
- Una aplicació intenta llegir un fitxer però hi ha un problema en el maquinari que ho impedeix.
- Una aplicació intenta obtenir un element d'una col·lecció buida.
- Una aplicació intenta inserir un element a una col·lecció que ja no admet més elements (per exemple, perquè està plena).
- Una aplicació intenta inserir un element que ja es trobava en una col·lecció que no admet elements repetits.

A.1 Mecanisme de gestió d'excepcions

Les excepcions s'activen (*es llencen*) quan una operació detecta una situació anòmala.

Quan una excepció és llençada s'atura l'execució de la seqüència normal d'instruccions i es busca alguna altra operació de la seqüència de crides que *capturi* l'excepció llençada (mitjançant un bloc `try-catch`).

Si no hi ha cap operació que capturi l'excepció llençada, s'atura l'aplicació de manera anormal.

Vegem com actua aquest mecanisme amb més detall (vegeu la figura A.1):

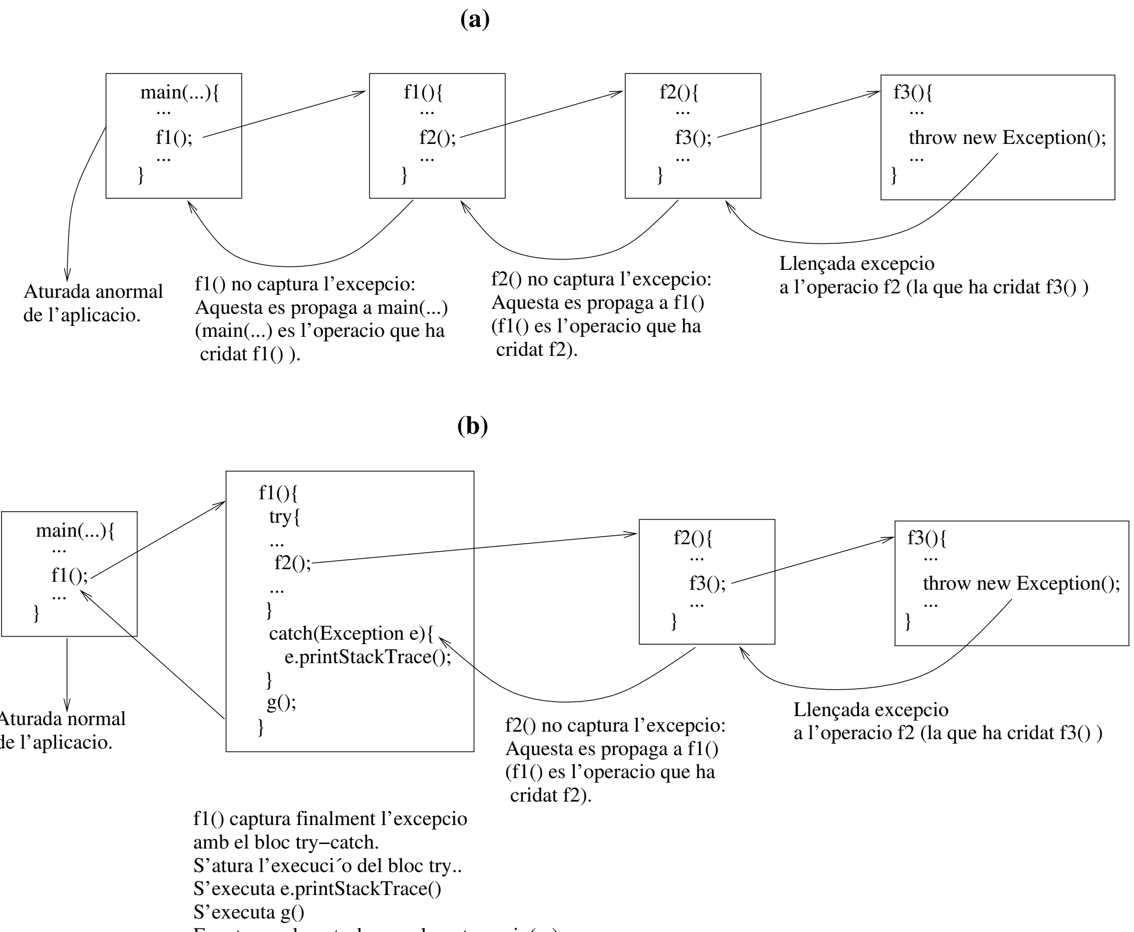


Figura A.1: Mecanisme de gestió de les excepcions

1. `main(...)` crida `f1()`, aquesta `f2()` i finalment, `f2` crida `f3()`.

   D'aquesta seqüència se'n diu *la seqüència de crides d'una aplicació*.

2. L'acció `f3()` que detecta la situació anòmala:

   (a) Atura immediatament la seva execució.

   (b) *Llença* un objecte d'una classe `Exception` a l'acció `f2()` que ha cridat `f3()`: `throw new Exception();`

   O sigui: l'excepció s'envia a l'operació anterior dins de la seqüència de crides de l'aplicació. En aquest cas, aquesta operació és `f2()`.

```
1 void f3() {
2     ...
3     if (situacio-anomala) throw Exception();
4     else ...execucio normal
5 }
```

3. L'operació `f2()` no conté cap bloc `try-catch` que capturi l'excepció llençada per `f3()`. Per tant, *no s'executa cap altra instrucció d'`f2` i l'excepció es propaga a `f1()`* (`f1()` és l'operació que va cridar `f2()`: l'anterior dins de la seqüència de crides).

4. Si `f1()` tampoc no conté cap bloc `try-catch` que capturi l'excepció, aquesta es propagarà a `main(...)`.

   Si `main(...)` tampoc no la captura, s'aturarà l'aplicació de forma anormal.

   Aquesta és la situació mostrada a la figura A.1(a).

5. Si `f1()` crida `f2()` en un bloc `try` i *captura* l'excepció `Exception` en un bloc `catch` associat al bloc `try` anterior, s'executa el codi d'aquell bloc `catch`, el qual tracta l'excepció.

> És molt important notar que no s'executa cap més instrucció del bloc `try` dins del qual hi ha la crida a `f2()` que ha llençat l'excepció.

```
1 void f1() {
2     try {
3
4         f2();  //Si f2() llenca una excepcio, no s'executa cap
5                //mes instruccio d'aquest bloc try!!!!
6
7     } catch (Exception e) {
8         ...tractament de l'excepcio
9 ␣␣␣␣}
10
11 ␣␣␣␣g();
12 }
```

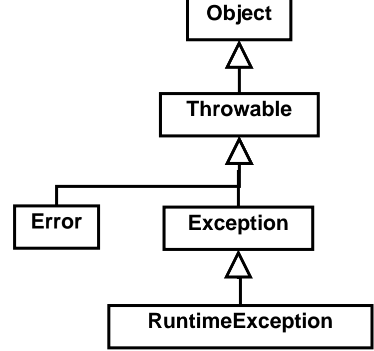


Figura A.2: Jerarquia d'excepcions

En aquest cas, l'execució de l'operació `f1()` continua normalment amb les instruccions que segueixen el bloc `catch`. En aquest cas, es cridaria normalment a `g()` i, en acabar `g()` i `f1()`, es retornaria el control a `main()`.

Aquesta és la situació mostrada a la figura A.1(b).

A.2 Tipus d'excepcions

El Java considera tres tipus d'excepcions:

1. *Excepcions* identificables i resolubles des de l'aplicació.
2. *Errors* aliens a l'aplicació.
3. Excepcions *Runtime* que solen denotar errors de programació.

La figura A.2 mostra la jerarquia d'excepcions definida al Java. Les classes més importants de la figura són explicades als apartats següents.

A.2.1 Excepcions que l'aplicació ha de tractar (*checked exceptions*)

Són situacions anòmales *dins de la lògica de l'aplicació*. Com que formen part de la lògica de l'aplicació, aquesta s'hi pot anticipar i, per tant, les pot tractar adequadament. Exemples:

- Una aplicació intenta obtenir un element d'una col·lecció buida.
- Una aplicació intenta inserir un element a una col·lecció que ja no admet més elements (per exemple, perquè està plena).
- Una aplicació intenta inserir un element que ja es trobava en una col·lecció que no admet elements repetits.
- Una aplicació intenta obrir un fitxer que no existeix a la carpeta on se cerca.

Aquestes excepcions són subclasses de `java.lang.Exception` i una aplicació les ha de tractar. Per tant:

- O bé cada operació captura (amb un bloc `try-catch`) les que es generen durant la seva execució:

```
1   void f1() {
2       try {
3           f2();
4       } catch (SomeException e) {
5           ...
6       }
7   }
```

- O bé l'operació declara que pot llençar aquella excepció:

```
1       void f1() throws SomeException {
2           f2();
3       }
```

En aquest cas, no cal que f1 inclogui el bloc `try-catch`.

A.2.2 Errors

Els errors són excepcions fora de la lògica de l'aplicació i que una aplicació no pot anticipar o resoldre. Exemple:

- Una aplicació intenta obrir un fitxer però, per un error de maquinari no pot fer-ho. Això genera un error de tipus `java.io.IOError`.

Una aplicació no té per què capturar-los amb un bloc `try-catch` ni tampoc declarar que els llença.

Els errors són subclasses de `java.lang.Error`.

A.2.3 Excepcions d'execució (*runtime exceptions*)

Solen correspondre a errors (*bugs*) al programa. Exemples:

- Una aplicació fa una divisió per zero.
- Una aplicació intenta accedir a una posició d'un vector fora del seu rang de definició.
- Una aplicació intenta posar en un vector un element d'un tipus incompatible amb el tipus dels elements del vector.
- Una aplicació intenta fer la conversió explícita (*cast*) d'una variable a un tipus no compatible amb el de la variable.
- Una aplicació intenta crear un vector amb rang negatiu.

- Una aplicació intenta accedir a una operació d'una classe que no està suportada per la classe.

Una aplicació no té per què capturar aquestes excepcions amb un bloc `try-catch` ni tampoc declarar que les llença. Habitualment, el que cal fer amb una aplicació que genera aquestes excepcions és depurar-la, o sigui: arreglar els errors de programació que conté.

Aquest tipus d'excepcions són subclasses de `java.lang.RuntimeException`.

A.3 Exemple

Una pila (stack) és una estructura de dades (o sigui, una col·lecció d'elements) en què el proper element que es treurà de la pila és el darrer que hi va entrar.

D'això se'n diu gestió LIFO (*Last Input, First Output*).

Implementem una pila d'enters (`StackOfInteger`).

```
1  public class StackOfInteger {
2      private int[] elems;
3      int nelems;
4
5      final int MAXSIZE=100;
6
7      public StackOfInteger() {
8          elems = new int[MAXSIZE];
9          nelems = 0;
10     }
11
12     public StackOfInteger(int max) {
13         if (max <=0) max = MAXSIZE;
14         elems = new int[max];
15         nelems = 0;
16     }
17
18     public void push(int e) throws StackOverflowException {
19         if (nelems == elems.length)
20             throw new StackOverflowException();
21         else{
22             elems[nelems] = e;
23             nelems++;
24         }
25     }
26
27     public void pop() throws EmptyStackException {
28         if (nelems == 0)
29             throw new EmptyStackException();
30         else
31             nelems--;
32     }
33
34     public int top() throws EmptyStackException{
```

```
35          if (nelems == 0)
36              throw new EmptyStackException();
37          else
38              return elems[nelems-1];
39      }
40  }
```

```
1   public class Main {
2       public static void main(String[] args) {
3           try {
4               StackOfInteger s = new StackOfInteger(20);
5
6               s.push(3);
7               s.pop();
8               s.pop();
9           } catch(EmptyStackException e) {
10              System.out.println("Empty stack");
11              e.printStackTrace();
12          } catch(StackOverflowException e) {
13              System.out.println("Stack overflow");
14              e.printStackTrace();
15          }
16      }
17  }
```

Comentaris:

- Notem que un bloc `try` té associat un bloc `catch` per a cada excepció que vol capturar.

  Aquest programa acabarà llençant una excepció. De quin tipus?

Finalment, cal definir les classes `StackOverflowException` i `EmptyStackException`:

```
1   public class EmptyStackException extends Exception { ... }
2   public class StackOverflowException extends Exception { ... }
```

A.4 `finally`

A continuació dels blocs `catch` associats a un determinat `try`, hi pot haver un bloc `finally`. La idea d'aquest bloc és que les seves instruccions s'executaran sempre, a la fi del bloc `try` al qual està associat, independentment de si aquest bloc acaba normalment o anormalment (i.e., es llencen excepcions i s'executa algun `catch`). Exemple:

```
1       public void op(StackOfInteger s) {
2           try {
3               s.push(3);
4               s.pop();
5               s.pop();
6           } catch (EmptyStackException e) {
```

```
7             System.out.println("Empty stack");
8             e.printStackTrace();
9         } catch (StackOverflowException e) {
10            System.out.println("Stack overflow");
11            e.printStackTrace();
12        } finally {
13            s.empty();   //Operacio que buida una pila
14                         //(no l'hem implementada)
15        }
16    }
17 }
```

La clàusula `finally` assegura que la pila `s` que es passa com a paràmetre es buidarà abans d'acabar `op`, passi el que passi amb les operacions de l'interior del `try`.

Apèndix B

Classes aniuades, locals i anònimes

B.1 Classes aniuades

Una classe aniuada és una classe que està definida com a membre d'una altra classe.

```
1 public class C {
2     public class A { ... }
3     ...
4 }
```

A la classe C, de què és membre una classe aniuada A, l'anomenarem en aquest text *classe contenidora* (direm que C *és una classe contenidora de* A). A una classe que no és aniuada de cap altra classe, l'anomenarem en aquest text *classe externa*.

- El nom complet d'aquestes classes està format per la classe contenidora, seguida de punt, seguida de la classe aniuada:

  `C.A`

  (on `C` és el nom de la classe contenidora i `A` el de la classe aniuada).

- Una classe aniuada pot ser, a la seva vegada, contenidora d'una altra classe.

  ```
  1 public class C {
  2     public class A {
  3         public class A2 { ... }
  4         ...
  5     }
  6     ...
  7 }
  ```

  `C.A` és aniuada de `C` i contenidora de `C.A.A2`.

El Java diferencia dos tipus de classes aniuades: les *estàtiques* i les *no estàtiques*. Abans de descriure-les, fem un petit recordatori del funcionament dels membres estàtics.

B.1.1 Membres estàtics. Recordatori

Exemple B.1: Membres estàtics i no estàtics d'una classe

Considerem la classe següent que té definits alguns membres estàtics.

```
1  public class A {
2      static int i;
3
4      int j;
5
6      static void f() {
7          i = 10;   //OK
8          j = 4;    //MALAMENT!
9          h();      //MALAMENT!
10     }
11
12     void h() { ... }
13
14     void g() {
15         i = 10;   //OK
16         j = 4;    //OK
17         h();      //OK
18     }
19 }
20
21 ...
22
23 A a = new A();
24 A b = new A();
25 A c = new A();
```

- Que `i` sigui un atribut estàtic vol dir que hi ha un únic enter `i` compartit per a totes les instàncies de la classe (en canvi, hi ha un enter `j` diferent per a cada instància de `A`). La figura B.1 ho mostra.
- Que `f` sigui una operació estàtica vol dir que `f` no es crida sobre una instància de la classe `A` sinó directament sobre la mateixa classe `A`: `A.f()` i, per tant, no té sentit que `f` accedeixi als membres de la instància sobre la qual es crida (com hem dit, no es crida sobre cap instància). Per aquest motiu, el codi de les línies 8 i 9 és incorrecte. En canvi, `g`, que no és estàtica, es crida sobre una instància de `A` i pot accedir als seus membres. A les línia 15, 16 i 17, es crida `g` sobre la instància `this` de `A`. Aquestes línies són equivalents a fer `this.i=10; this.j = 4;` i `this.g();`, respectivament.

■ ■ ■

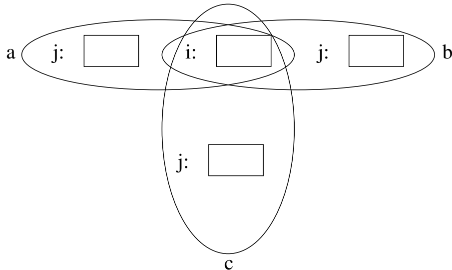


Figura B.1: Atributs estàtics i no estàtics

> En definitiva, els membres estàtics d'una classe no estan associats a una instància d'aquella classe. Els membres no estàtics, sí.

Donem ara un cop d'ull a les classes aniuades estàtiques i no estàtiques. Comencem mostrant un exemple amb codi al qual ens anirem referint al llarg de l'explicació:

Exemple B.2: Un exemple de classes aniuades

```
1  public class OuterClass {
2      private int i = 10;
3      private static int j = 20;
4      int k = 30;
5
6      public void f() { ... }
7
8      public class NonStaticNestedClass {
9          int i = 100;
10
11         public void fInner() {
12             //System.out.println(this.j); MALAMENT
13             System.out.println("NonStaticInnerClass.i=" + i);
14             System.out.println("NonStaticInnerClass.i=" + this.i);
15             System.out.println("OuterClass.i=" + OuterClass.this.i);
16             System.out.println("OuterClass.k=" + OuterClass.this.k);
17             System.out.println("OuterClass.k=" + k);
18         }
19     }
20
21     public static class StaticNestedClass {
22         public void fStatic() {
23             System.out.println("OuterClass.j" + j);
24             StaticNestedClass2 x = new StaticNestedClass2();
25             //NonStaticNestedClass nsic =
26             //         new NonStaticNestedClass(); MALAMENT
27         }
28     }
```

```
29
30      ...
31
32      public static void main(String[] args) {
33          OuterClass o1 = new OuterClass();
34          OuterClass.NonStaticNestedClass o2 =
35                  o1.new NonStaticNestedClass();
36          //OuterClass.StaticNestedClass =
37          //        o1.new StaticNestedClass(); MALAMENT
38
39          OuterClass.StaticNestedClass oinner =
40                  new OuterClass.StaticNestedClass();
41          oinner.fStatic();
42          o1.k = 300;
43          o2.fInner();
44      }
45  }
```

■ ■ ■

B.1.2 Classes aniuades estàtiques

Les classes aniuades estàtiques són classes definides com a membre estàtic d'una altra classe.

Aquestes classes es coneixen als textos de referència del llenguatge Java com a:

- **static nested classes** o
- **top-level (static) nested classes**.

A l'exemple anterior, la classe `StaticNestedClass` és una classe aniuada estàtica.

Igual que la resta dels membres estàtics d'una classe, una classe aniuada estàtica està associada a la classe contenidora, *no a una instància de la classe contenidora*.

Això té les implicacions següents:

- Una operació d'una classe aniuada estàtica no pot fer referència a un membre no estàtic (atribut, operació o classe) de la seva classe contenidora. Per aquest motiu, les instruccions 24 i 25 en l'exemple anterior són incorrectes.

  Sí que pot fer referència a membres estàtics de la classe contenidora (vegeu l'accés a l'atribut `j` de l'operació `fStatic`). En particular, pot fer referència als membre **estàtics i privats** de la classe contenidora.

- Les classes aniuades estàtiques *tenen el mateix comportament que les classes externes*. La diferència més important és que, com que han estat definides com a membre d'una classe contenidora, tenen el seu nom prefixat pel de la classe contenidora (`C.A` en lloc de simplement `A`).

Així, per exemple, quan volem crear una instància d'una d'aquestes classes només ens cal fer (vegeu la línia 40):

```
1       OuterClass.StaticNestedClass  oinner =
2               new OuterClass.StaticNestedClass();
```

Exactament com si creéssim una instància d'una classe anomenada `OuterClass.StaticNestedClass`.

És incorrecte, en canvi, el codi de la línia 43:

```
1       //OuterClass.StaticNestedClass =
2       //          o1.new  StaticNestedClass();  MALAMENT
```

No es pot crear una instància d'una classe aniuada estàtica associada a un membre (`o1`) de la classe contenidora.

- Quan es compila una classe contenidora C que conté les classes aniuades estàtiques $A1$ i $A2$ es generen els fitxers:

  $C\$A1$ i $C\$A2$, cadascun dels quals conté la classe que el seu nom indica. Aquest fet reforça la idea de la independència entre una classe contenidora i una classe aniuada estàtica.

  A l'exemple, es generarà la classe `OuterClass$StaticNestedClass`.

B.1.3 Classes aniuades no estàtiques

Les classes aniuades no estàtiques són classes definides com a membre no estàtic d'una altra classe, anomenada *classe contenidora*.

Aquestes classes es coneixen als textos de referència del llenguatge Java com a:
- **Inner classes**

A l'exemple anterior, la classe `NonStaticNestedClass` és una classe aniuada no estàtica.

Com que aquestes classes són membres no estàtics de la classe contenidora, es caracteritzen perquè *una instància d'una classe aniuada no estàtica forma part d'una instància de la seva classe contenidora*.

Com a conseqüència:

- Des d'una operació d'una classe aniuada no estàtica es pot accedir als atributs i operacions d'instància (i.e., no estàtics) de la classe contenidora.

  Això és el que passa a l'exemple quan des de l'operació `finner` de la classe aniuada no estàtica `NonStaticNestedClass` accedim a l'atribut `k` definit a la classe contenidora `OuterClass`:

```
1   public void fInner() {
2       System.out.println(k);
3   }
```

L'atribut `k` és un atribut d'instància de la classe contenidora `OuterClass`. Concretament, `k` *es refereix a l'atribut* `k` *de la instància d'*`OuterClass` *a la qual està associada la instància de* `NonStaticNestedClass` *amb què es crida* `fInner`. Una mica embolicat? Retornem a l'exemple i recordem les instruccions rellevants del `main`:

```
1  public static void main(String[] args) {
2      OuterClass o1 = new OuterClass();
3      OuterClass.NonStaticNestedClass o2 =
4              o1.new NonStaticNestedClass();
5      o1.k = 300;
6      o2.fInner();
7  }
```

L'operació `fInner` es crida sobre l'objecte `o2` de la classe aniuada `NonStaticNestedClass`. Per ser l'objecte `o2` d'una classe aniuada no estàtica, està associat a un objecte de la seva classe contenidora. Aquest objecte és `o1`. Ara, la crida a `o2.fInner()` escriurà el valor de l'atribut `k` de l'objecte de la classe contenidora al qual està associat `o2`. O sigui, `o1.k`, això és, `300`.

- Les operacions de les classes aniuades no estàtiques *poden accedir als membres privats de les seves classes contenidores*. Per exemple, l'accés de `fInner()` a l'atribut `i` d'`OuterClass`.

Accés a la instància de la classe contenidora associada a una instància de la classe aniuada no estàtica

Un aspecte important quan treballem amb classes aniuades no estàtiques és *la manera d'accedir des d'una operació de la classe aniuada no estàtica als membres definits a la seva classe contenidora*, com per exemple, la manera d'accedir des de l'operació `fInner()` a l'atribut `k` que hem vist a l'apartat anterior.

Em direu que això ja ho hem vist i que no té res d'especial. Mmm... És cert que ho hem vist, però també és cert que hem fet una mica de trampa. Imagineu que volguéssim usar `this` per accedir a l'atribut `k`. El codi:

```
1  public void fInner() {
2      System.out.println(this.k);   //COMPTE! MALAMENT!
3  }
```

estaria malament. Per què?

Ho heu encertat (espero): perquè `this` fa referència a `o2` (o sigui, a un objecte de la classe aniuada `NonStaticNestedClass`), i aquesta classe **no té definit cap atribut** `k`! Si no posem `this`, el compilador cerca l'atribut `k` a la classe aniuada i, si no el troba, entén que l'ha d'anar a buscar a la classe contenidora; per això l'exemple a l'apartat anterior ens funcionava. Però si posem `this`, el compilador entén que volem l'atribut `k` definit per l'objecte `o2` i aquest atribut no existeix.

El problema es resoldria si tinguéssim una manera de referir-nos a l'objecte de la classe contenidora associat a `o2`. Aquesta manera és: `OuterClass.this` i el codi de `fInner` quedaria:

```
1  public void fInner() {
```

```
2       System.out.println(OuterClass.this.k);
3   }
```

`OuterClass.this` vol dir: *l'objecte de la classe contenidora al qual està associat l'objecte* `o2` *amb què cridem* `fInner()` (aquest objecte de la classe `OuterClass` és `o1`).

Pot semblar que ens compliquem la vida definint `OuterClass.this` quan podríem accedir directament a `k`, com fèiem en l'apartat anterior. Pot semblar-ho però no és cert: a vegades resulta necessari usar la notació `OuterClass.this`. És el cas de l'accés a l'atribut `i`. Aquest atribut està definit a `OuterClass` i sobrecarregat (redefinit) a `NonStaticInnerClass`. Quan a `fInner()` ens referim a `i` o a `this.i` ens estem referint a l'atribut `i` definit a `NonStaticInnerClass`. Aquest atribut amaga (*shadows*) la definició que es fa a `OuterClass`. Com ens podrem referir des de `fInner()` a `OuterClass.i`? Doncs usant la notació `OuterClass.this.i`:

```
1   public void fInner() {
2       System.out.println(i);  //NonStaticInnerClass.i
3       System.out.println(this.i);  //NonStaticInnerClass.i
4       System.out.println(OuterClass.this.i);  //OuterClass.i
5   }
```

B.1.4 Quan usem classes aniuades de cada tipus

Usarem classes aniuades quan:

- Una classe `A` (la que s'aniuarà) és usada únicament per una altra `C` (la contenidora). Per què cal exposar `A` al món si només la fa servir `C`?

  Aquesta situació ens la trobem al capítol 2 amb les classes `ListItr` i `Entry`.

  - `Entry` representa un node d'una llista enllaçada (`LinkedList`). Els clients de la classe `LinkedList` no usen nodes directament (de fet, fan abstracció de com està implementada la llista i, per tant, dels nodes).
  - `ListItr` representa un iterador sobre una llista. És cert que els clients de les llistes (`ArrayList`, `LinkedList`) operen amb iteradors, però només coneixen la interfície `ListIterator`, fan abstracció de la classe específica d'iterador que implementarà aquella interfície. Per tant, *no usen directament la classe* `ListItr` *i, per tant, aquesta es pot fer classe aniuada d'*`ArrayList` *o de* `LinkedList`.

  Això ho trobareu explicat amb més detall a la secció 2.7.4

- Una classe `A` (que serà l'aniuada) està tan interlligada amb una altra `C` que és bo que accedeixi a atributs o operacions privades de `C`. Aquest accés d'una classe a la part privada d'una altra pot ser convenient per raons d'eficiència o per evitar la definició d'una sèrie d'operacions per a la classe `C` pensades únicament per què hi pugui accedir `A` però que no resultin apropiades per a l'especificació de `C`.

  Si les dues classes són externes això no serà possible. En canvi, si `A` s'aniua dins de `C`, podrà accedir als membres privats de `C` sense problemes.

  Aquest és el cas de `ListItr`, al capítol 2. Aquesta classe accedeix als atributs privats de la classe `LinkedList<T>` (vegeu la pàgina 113).

- Per agrupar classes lògicament subordinades.

Sovint, ens trobem amb situacions en què dues o tres d'aquestes consideracions són aplicables. Aquests casos suggereixen fortament el disseny de classes aniuades.

I com decidirem si usem una classe aniuada estàtica o no estàtica?

- Si la classe aniuada és essencialment independent de la contenidora o no és imprescindible que accedeixi a membres estàtics d'ella, optarem per un aniuament estàtic.

  Aquest és el cas de la classe `LinkedList.Entry<T>` (vegeu 106). Aquesta classe representa un node d'una llista enllaçada. Només és usada per `LinkedList<T>` i no és imprescindible que accedeixi a la part privada d'aquesta classe. En conseqüència, la implementem com a classe aniuada estàtica.

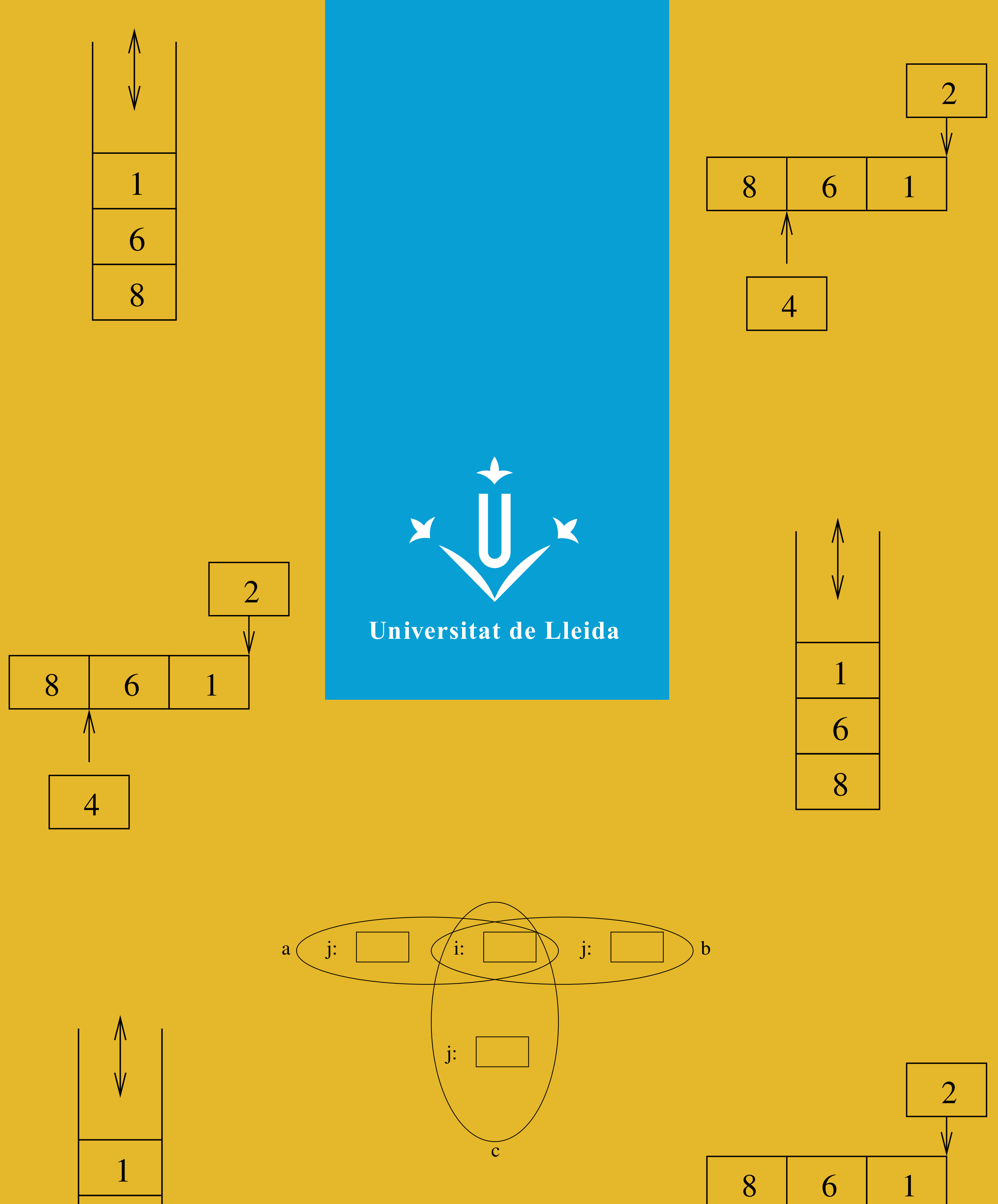
  També al capítol 4 trobem un altre exemple de classe aniuada estàtica: la classe `HashMap.Entry<K,V>` (vegeu 176 i 220) que implementa la interfície `Map.Entry<K,V>`, que és una interfície aniuada dins de la interfície `Map<K,V>`. El motiu pel qual `HashMap.Entry<K,V>` és aniuada estàtica és similar al de `LinkedList.Entry<T>`.

- Si una instància de la classe aniuada `A` ha d'accedir a atributs o operacions d'una instància específica de la contenidora `C`, aleshores usarem l'aniuament no estàtic.

  Com ja hem dit, aquest és el cas de la classe `ListItr` (vegeu novament la pàgina 113).

Useu només classes aniuades no estàtiques si és absolutament imprescindible (o, almenys, molt convenient) que la classe aniuada accedeixi als membres no estàtics de la classe contenidora.

Si ho podeu evitar, useu classes aniuades estàtiques.


2
8
6
1
4
Universitat de Lleida
a
j:
i:
j:
b
j:
c