Estructures d'accés directe

Josep M. Ribó

25 de gener de 2011

Capítol 3. Contingut

3.1 Introducció

Contingut

3.1.1 Taules. Idea intuïtiva

c:	anna 15.11.1980 anna@.....	carles 15.10.1970 carles@...	lluis 1.2.1990 lluis@.....	pius 20.5.1970 pius@.....
	1	2	3	4

- **Accés seqüencial:** Obtenir el $següent$ element (e.g., el següent d'anna és carles)

- **Accés per posició:** Obtenir l'element que ocupa la posició p al contenidor (e.g. l'element que ocupa la posició 3 és el lluis)

- **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@....)

3.1.1 Taules. Idea intuïtiva

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)
- Considerarem la clau com una **cadena de caràcters**
- Exemples de claus: nif, matrícula, número de compte d'estalvi, telèfon...

Taula (map, dictionary):

Estructura de dades que ens permet un accés eficient als seus elements a partir de la clau

Com podem modelitzar una taula?

- **Com una col.lecció de parelles** `[clau, valor]`:

```
m={[ "12345678A",<pepet,15.11.1970,pepet@meumail.com> ],
   [ "18888886S",<anna,15.10.1980,anna@meumail.com> ],
   [ "33333888X",<mari,1.1.1970,mari@meumail.com> ]...
  }
```

  De manera que es pot accedir al valor eficientment a través de la clau

- **Algebraicament, 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 *índexos, identificadors*)
  - V: Conjunt de valors
    (també anomenats *informació, contingut*)

3.1.1 Taules. Idea intuïtiva

Consideracions addicionals sobre les taules:

- L'accés per clau ha de ser **eficient**: s'ha d'aconseguir amb un cost **O(1)** o **O(log n)**

  (`n` és el nombre d'elements emmagatzemats a la taula)

- Existeixen variants de les taules en què la clau no és identificadora: **multitaula (multimap)**. Poden contenir varis elements amb la mateixa clau

  Les multitaules no són estudiades en aquests apunts

- En una taula, la clau podria no ser una cadena de caràcters.

  Aquests apunts es limiten a taules, les claus de les quals són cadenes de caràcters

  En general, es podrà fer una transformació de la clau a una cadena de caràcters

- Per tal d'identificar un element d'una taula es podria necessitar més d'un atribut d'un element (e.g., Un pis en una ciutat s'identifica amb la unió dels atributs `carrer`, `número`, `pis`, `porta`)

  Aquests apunts es limiten a taules amb una clau composta per un sol atribut

3.1.2 Taules. Operacions

Sobre una taula es defineixen les operacions principals següents:

- **Creació d'una taula buida**

  `Map<T> m;`

  Crea una taula `m` buida que podrà contenir elements de tipus `T`

- **Inserció d'una parella [clau, valor] en una taula**

  `m.put(k,v);`

  La parella amb clau `k` i valor `v`: `[k,v]` s'ha afegit a la taula `m`

- **Eliminació d'una clau (i del seu valor associat)**

  `m.remove(k);`

  Elimina de la taula `m` la clau `k` i el seu valor associat

- **Consulta del valor associat a una clau**

  `m.get(k,v);`

  Recupera el valor `v` associat a la clau `k` dins la taula `m`

  Observació: Voldrem que aquesta operació tingui un cost O(1) o O(log n)

3.1.2 Taules. Operacions

Les operacions anteriors són les clàssiques, però sovint ens plantegem dues noves necessitats:

1. **Recorreguts de taules**
   - Per ordre de clau
   - En un ordre no especificat
2. **Referències *directes* a elements que s'han inserit dins d'una taula**

Per resoldre 1 i 2 : **ITERADORS SOBRE TAULES**

A les properes transparències presentarem el problema al qual es refereix 2

3.1.3 Referències directes als elements d'una taula

- Considerem la classe que modelitza els usuaris d'una biblioteca

```
class User{
  MyString name;
  MyString nif;
  MyString address;
  MyString phone;
public:
 ...
};
```

- Volem accedir ràpidament als usuaris de la biblioteca pels atributs `name` i `nif`. Per això considerem dues taules:
  - `mname`: clau= nom
  - `mnif`: clau=nif

  Quins seran els valors que emmagatzemarem a aquestes taules?

SOLUCIÓ 1: Dues taules d'usuaris

```
Map<User> mnif;
Map<User> mname;
User u1(MyString("pepet"), MyString("12345678A"),
        MyString("C. Badall, 23"),
        MyString("973.303030"));

mnif.put(MyString("12345678A"),u1);
mname.put(MyString("pepet"),u1);
```

mnif:	12345678A user1	22222222S user2	12333333H user3	12121212L user4

mname:	pepet user1	anneta user2	lluiset user3	pius user4

Característiques:

- L'usuari es replica a totes dues taules
- Accés molt eficient tant per nom com per nif

Problemes:

- Malbaratament d'espai
- Possibles inconsistències degudes als elements replicats

SOLUCIÓ 2: Una taula d'usuaris i una de nifs

```
Map<User> mnif;
Map<MyString> mname;

User u1(MyString("pepet"), MyString("12345678A"),
        MyString("C. Badall, 23"),
        MyString("973.303030"));

mnif.put(MyString("12345678A"),u1);
mname.put(MyString("pepet"),MyString("12345678A"));
```

mnif:

12345678A user1	22222222S user2	12333333H user3	12121212L user4

mname:

pepet 12345678A	anneta 22222222S	lluiset 12333333H	pius 12121212L

La taula mname associa a cada nom el seu nif.

- **Accés per nif:**

  ```
  mnif.get(MyString("12345678A"),u);
  ```

- **Accés per nom:**

  1. Accés al nif de l'usuari a partir del seu nom:

     ```
     mname.get(MyString("pepet"),nif);
     ```

     `nif` contindrà el nif de "pepet": "12345678A"
  2. Accés al contingut de l'usuari a partir del seu nif:

     ```
     mnif.get(nif,u);
     ```

     `u` contindrà les dades de l'usuari que té nif `nif` (i nom "pepet")

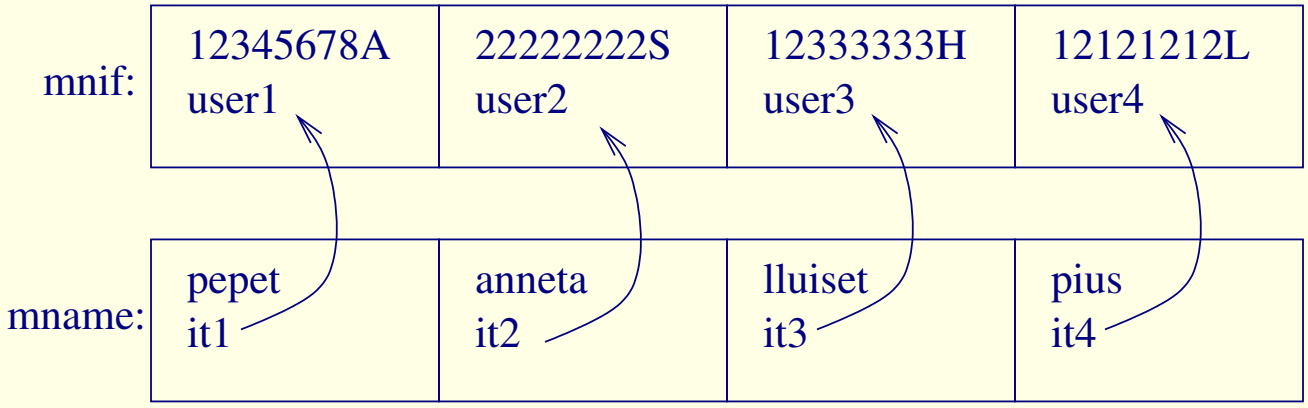
Característiques:

- Gestió més eficient de l'espai (encara que no òptima)
- Temps d'accés acceptable (encara que no òptim):

  S'accedeix a dues taules per clau (cost baix)
- Aquesta és una solució raonable

SOLUCIÓ 3: Una taula d'usuaris i una d'iteradors a la primera taula

Idea de la solució:
La taula de noms no conté la clau de la taula de nifs (com a la solució 2) sinó directament un iterador a un dels usuaris emmagatzemats a la taula de nifs



Problemes:

- Com podem obtenir l'iterador a un element situat dins de la taula `mnif`?
- Com podem accedir a un element de la taula a partir d'un iterador?

Solució:

Definim tres operacions noves sobre les taules:

- `m.put(k,v,it);`

  Insereix la parella `(k,v)` a la taula `m` i retorna un iterador `it` que es refereix a la còpia del valor `v` que acaba de ser inserida dins la taula `m`

- `m.get(it,v);`

  Obté l'element `v` dins la taula `m` al qual es refereix l'iterador `it`.

  Aquesta operació s'ha de fer amb cost O(1)

  Per tant és encara potencialment més eficient que l'accés per clau

  Aquesta operació no és estrictament necessària: es pot subsitituir per `*it;`

- `m.get(k,it);`

  Retorna un iterador `it` que es refereix al valor associat a la clau `k` dins de la taula `m`

  Aquesta operació s'ha de fer amb cost O(1) o O(log n)

A l'exemple que estem mostrant tindríem el següent:

```
Map<User> mnif;
Map<MapIterator<User> > mname;
MapIterator<User> it;

User u1(MyString("pepet"), MyString("12345678A"),
        MyString("C. Badall, 23"),
        MyString("973.303030"));

mnif.put(MyString("12345678A"),u1,it);  //(1)
mname.put(MyString("pepet"),it,it2);    //(2)
```

(1): Inserim a la taula `mnif` la parella `["12345678A",u1]`. `it` és l'iterador a la còpia de `u1` inserida a `mnif`

(2): Inserim a la taula `mname` la parella `["pepet",it]`. `it` es refereix al valor de l'usuari associat a la clau "pepet"dins la taula `mnif`. L'operació retorna un iterador `it2` a la còpia de `it` inserida a `mname`.

`it2` no l'usarem en aquest exemple.

La declaració de `it2` hauria de ser:

```
MapIterator<MapIterator<User> > it2;
```

- **Accés per nif:**

```
mnif.get(MyString("12345678A"),u);
```

- **Accés per nom:**

1. Accés a l'iterador a la taula `mnif` a partir del nom de l'usuari:

   ```
   mname.get(MyString("pepet"),it);
   ```

   `it` contindrà un iterador a les dades de l'usuari "pepet" a la taula `mnif`
2. Accés al contingut de l'usuari a partir de `it`:

   ```
   mnif.get(it,u);
   ```

   o bé:

   ```
   u=*it;
   ```

   `u` contindrà les dades de l'usuari que té nom "pepet"

Característiques:

- Gestió òptima de l'espai
- Temps d'accés òptim (exigim que l'operació `*it` i `mnif.get(it,u)` tinguin un cost O(1)
- **Solució força bona**

3.2 La classe Map. Especificació

Contingut

3.2.1 Especificació de la classe Map<T>

3.2.2 Especificació de la classe MapIterator<T>

3.2.1 Especificació de la classe `Map`

Qüestions prèvies a l'especificació de la classe `Map<T>`:

- Modelitzarem una taula com una col.lecció de parelles (clau,valor) tal que:
  - *Clau:* Cadena de caràcters (`MyString`)
  - *Valor:* de classe `T`

  Notem que aquesta modelització no suposa cap ordenació específica dels elements de la taula

- Modelitzarem una taula `m` respecte un iterador `it`: `m(it)`, com una de les possibles seqüències que es poden construir amb els elements de `m` tal que:

  `m(it)=( s1,[k,v]*s2 )` significa: `*it=v`

Els iteradors definits sobre una instància de `Map<T>` indueixen una seqüènciació dels elements d'aquesta instància

Sovint, no especificarem en quin ordre recorrerà l'iterador els elements de la taula

Es poden definir diferents classes d'iteradors sobre una classe de taules amb diferents estratègies de recorregut

3.2.1 Especificació de la classe Map

Descripció de la classe Map<T>

- La classe Map<T> és una subclasse de Container que modelitza col.lecions de parelles [clau, valor] on:
  - la clau és identificaora i de tipus MyString i
  - el valor, de tipus genèric T.
- A les instàncies de la classe Map<T> les anomenem *taules*
- Map<T> és una classe abstracta. Les implementacions concretes de taules es faran en les subclasses de Map<T>
- Les implementacions concretes de Map<T> hauran de proveir un accés eficient (O(log n) o O(1)) als elements de la taula per clau
- Map<T> està provista d'iteradors

Classe Map<T>. Operacions:

- `Map<T>::Map();`
  - **Crida:** `Map<T> m;`
  - **Prec:**
  - **Post:** `m` és una taula buida que podrà contenir parelles (clau,valor) tal que el tipus de la clau serà `MyString` i el tipus del valor, `T`

- `void Map<T>::put(const MyString k, const T& v, MapIterator<T>& it);`

  - **Crida:** `m.put(k,v,it)`
  - **Prec:**
  - **Post:**

    si existeix `v1:T` tal que `[k,v1]` pertany a `m'`, aleshores:
    - `m=(m'-[k,v1])` $\cup$ $\{$`[k,v]`$\}$ i
    - `m(it)=(s1,[k,v]*s2)`
      on `(s1,[k,v]*s2)` constitueix una certa seqüenciació dels elements de `m`.

    si no
    - `m=m'` $\cup$ $\{$`[k,v]`$\}$ i
    - `m(it)=(s1,[k,v]*s2)`
      on (s1,[k,v]*s2) constitueix una certa seqüènciació dels elements de `m`.

    (Si `m'` conté un parell `[k,v1]`, el valor `v1` es substituït per `v`
    si no `m=m'` amb el parell `[k,v]` afegit
    `it` es refereix al valor `v` que acabem d'inserir a `m`)

    si la parella [k,v] no es poden inserir per manca d'espai es llença l'excepció `FullMapException`
  - **Observacions:** Operació virtual pura

- `void Map<T>::remove(const MyString k)`
  - **Crida:** `m.remove(k);`
  - **Prec:**
  - **Post:**
    si existeix `v:T` tal que `[k,v]` pertany a `m'` aleshores `m=m'-[k,v]`
    si no `m=m'`
    (Elimina la clau `k` (amb el seu valor associat) de la taula `m'` si aquesta clau pertany a `m'`)
  - **Observacions:** Operació virtual pura

- `void Map<T>::remove(MapIterator<T>& it)`
  - **Crida:** `m.remove(it);`
  - **Prec:**
  - **Post:**
    si `m(it)'=(s1,[k,v]*s2)` aleshores `m(it)=(s1,s2)`
    si `m(it)'=(s1,∅)` `m=m'` i s'ha llençat l'excepció `IteratorException`
    (Elimina la parella [k, v] a la qual s'està referint l'iterador `it`. Si aquest iterador no s'està referint a cap element de `m'`, llença una excepció `IteratorException`
  - **Observacions:** Operació virtual pura

- `void Map<T>::get (const MyString k, T& v, bool& found)`
  - **Crida:** `t.get(k,v,found)`
  - **Prec:**
  - **Post:**
    si existeix `v1:T` tal que `[k,v1]` pertany a `m'` aleshores `v=v1` i `m=m'` i `found=cert`
    si no `m=m'`, `v` és indefinit i `found=fals`
  - **Observacions:** Operació virtual pura i const.
    El cost de les redefinicions d'aquesta operació ha de ser O(log n) o O(1)

- `void Map<T>::get (const MapIterator<T>& it, T& v)`

  - **Crida:** `t.get(it,v)`
  - **Prec:**
  - **Post:**
    si `m(it)'=(s1,[k,v1]*s2)` aleshores `v=v1` i `m=m'`
    si `m(it)'=(s1,`$\emptyset$`)` `m=m'` i s'ha llençat l'excepció `IteratorException`
    (`v` és una còpia del valor al que es refereix `it` dins `m`. Si `it` no es refereix a cap element de `m`, llença una excepció `IteratorException`)
  - **Observacions:** Operació virtual pura i const.
    El cost de les redefinicions d'aquesta operació ha de ser O(1).
    `m.get(it,v);` és equivalent a `v.copy(*it);`

- `void Map<T>::get (const MyString k, MapIterator<T>& it, bool& found)`
  - **Crida:** `t.get(k,it,found)`
  - **Prec:**
  - **Post:**
    si existeix `v1:T` tal que `[k,v1]` pertany a `m'` aleshores `*it=v1` i `m=m'` i `found=cert`
    si no `m=m'`, `v` és indefinit i `found=fals`
  - **Observacions:** Operació virtual pura i const.
    El cost de les redefinicions d'aquesta operació ha de ser O(log n) o O(1)

- `bool Map<T>::operator==(const Object&);`
  - **Crida:** `b=(m==m2);`
  - **Prec:**
  - **Post:**
    `b` és cert si `m` i `m2` són dos maps que contenen les mateixes claus i cadascuna té el mateix valor associat. `m=m'` i `m2=m2'`
  - **Observacions:** Operació virtual pura i const

- `void Map<T>::copy(const Object&);`
  - **Crida:** `m.copy(m2);`
  - **Prec:**
  - **Post:** `m=m2'` i `m2=m2'`
    (i.e., `m` és una taula que conté una una còpia de la taula (map) `m2`, la qual, no s'ha modificat)
  - **Observacions:** Operació virtual pura.

- `Object* Map<T>::clone();`
  - **Crida:** `pm=m.clone();`
  - **Prec:**
  - **Post:**
    `pm` és un apuntador a un map que és una còpia exacta de `m`.
  - **Observacions:** Operació virtual pura i const

- `MyString Map<T>::toString);`
  - **Crida:** `st=m.toString();`
  - **Prec:**
  - **Post:**
    `st` conté una representació textual del contingut de `m`. `m` no s'ha modificat.
  - **Observacions:** Operació virtual i const.

3.2.2 Especificació de la classe `MapIterator`

Descripció de la classe `MapIterator<T>`

- La classe `MapIterator<T>` és una subclasse de `Iterator<T>` que modelitza els iteradors sobre taules.
- Un iterador sobre una taula es refereix a un valor emmagatzemat a la mateixa
- `MapIterator<T>` és una classe abstracta. Les implementacions concretes d'iteradors sobre taules es faran en les subclasses de `MapIterator<T>`
- Les implementacions concretes de `MapIterator<T>` hauran de proveir un accés eficient (O(1)) al valor de la taula al que es refereixen

Operacions:

Les operacions de la classe `MapIterator<T>` són les que hereta de la classe `Iterator<T>`. S'hi afegeixen les següents amb significat obvi:

- `MapIterator()`
- `MapIterator(const Map<T>& m);`
- `MapIterator<T>& operator=(const MapIterator<T>&);`
- `bool operator==(const MapIterator<T>&);`

3.3. Algunes implementacions de la classe `Map`

Una taula es pot implementar seguint estratègies diferents:

1. Implementació en forma de llista enllaçada
2. Implementació en forma de vector ordenat
3. Implementació amb una funció injectiva
4. Implementació amb funcions de dispersió
5. Implementacions arborescents

En aquest capítol presentarem les 4 primeres formes d'implementació (i dedicarem una atenció molt especial a la quarta)
.

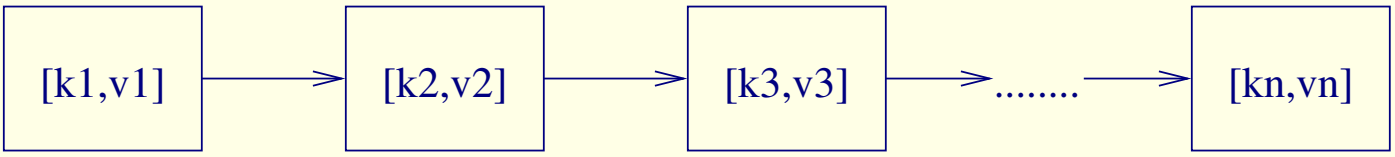
Les implementacions arborescents seran presentades al capítol 4

3.3. Implementacions de la classe `Map`. Llista

Considerem una taula `m`:

`m={[k1,v1], [k2,v2], [k3, v3],....,[kn,vn]}`

Implementació en forma de llista encadenada:



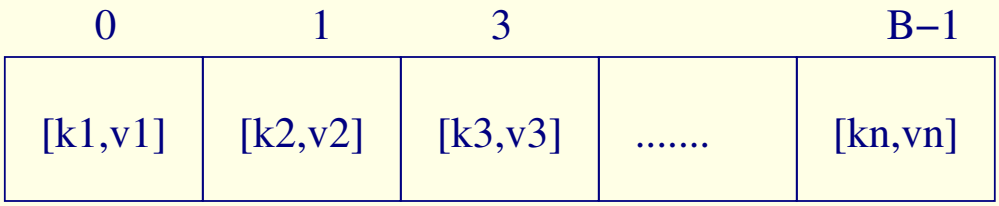
Cost consulta per clau: $O(n)$

Només acceptable si sabem segur que n és petit

Absolutament inacceptable en els casos habituals

3.3. Implementacions de Map. Vector ordenat

Idea:
Els elements de la taula s'emmagatzemen en un vector de parelles [clau, valor] tal que està ordenat alfabèticament per clau



k1<k2<....<kn

Característiques:

- Cost consulta per clau: O(log n) (amb cerca dicotòmica)
- Cost inserció/eliminació: O(n) (cal mantenir l'ordenació)
- 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

Acceptable si el volum d'insercions/eliminacions a la taula és baix o si les insercions/eliminacions no són operacions crítiques

3.3. Implementacions de la classe Map. Funcions injectives

Idea:

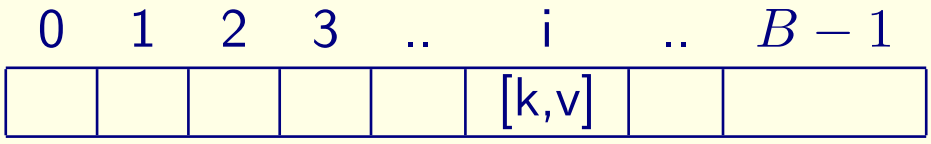
- Dissenyar una funció f que estableixi una alicació injectiva entre l'espai de claus i l'espai d'índexos d'un vector w ($w[0..B-1]$):

$$f : \texttt{String} \longrightarrow [0..B-1]$$

- La parella $[k, v]$ se situarà a l'índex $f(k)$ del vector w: $w[f(k)]$

Si $f(k) = i$:

Vector w:



Està garantit que a cada posició de w hi correspon una única clau (f és injectiva)

Exemple:

- 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'adreces (en cas contrari, f no pot ser injectiva)

Aquesta implementació és excel.lent.
Però no sempre és aplicable

Condicions d'aplicabilitat de la implementació d'una taula mitjançant funcions injectives

1. Podem trobar una f injectiva de cost de còmput baix (idealment O(1))

2. $| K | \leq | A |$

   (La mida de l'espai de claus menor o igual que la mida de l'espai d'adreces)

Habitualment la condició 2 fa fracassar l'ús d'aquesta implementació

Veiem-ne un parell d'exemples

Exemples:

- **Cas de l'hotel:**
  - $| K | = 280$ (el nombre d'habitacions de l'hotel)
  - $| A | = 280$

  Coneixem exactament **quantes claus** tenim i **quines són**: les únques claus vàlides són les 280: 100-139....700-739

- **Cas de la biblioteca:**

  Volem gestionar els usuaris d'una biblioteca mitjançant una taula a la que s'accedirà per NIF. Preveiem que hi haurà al voltant de 500 usuaris de la biblioteca:

  - $| K | = 100.000.000$ (el nombre de NIFs possibles)
  - $| A | = 500$

  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

  Aquesta és la situació més habitual

3.3. Implementacions de la classe Map. Funcions de dispersió

Idea:

Mantenim la mateixa idea que per la implementació amb funcions injectives, o sigui:

- Dissenyar una funció h entre l'espai de claus i l'espai d'índexos d'un vector ($[0..B-1]$):

$$h : \texttt{String} \longrightarrow [0..B-1]$$

- La parella $[k, v]$ se situarà a l'índex $h(k)$ del vector

Però ara no forçarem que h sigui injectiva
Per tant, hi podrà haver més d'una clau mapejada a una adreça determinada del vector

Aquesta és la implementació més interessant que estudiarem al capítol 3.

Hi dedicarem els apartats: 3.4.1-3.4.5

3.3. Implementacions de la classe Map. Implementacions arborescents

Idea:

Els arbres equilibrats tenen un nombre de nivells proporcional al logaritme del seu nombre de nodes

- Representarem una taula en forma d'arbre de manera que cada node de l'arbre correspongui a una parella [k,v] de la taula

- Dissenyarem algorismes de consulta que permeten baixar un nivell 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)

Les implementacions arborescents més típiques de taules són: **arbres binaris de cerca, AVLs, Arbres B, arbres B+, arbres 2-3, arbres Red-black....**

Al capítol 4 estudiarem els arbres binaris de cerca

3.4 Implementació de taules amb funcions de dispersió

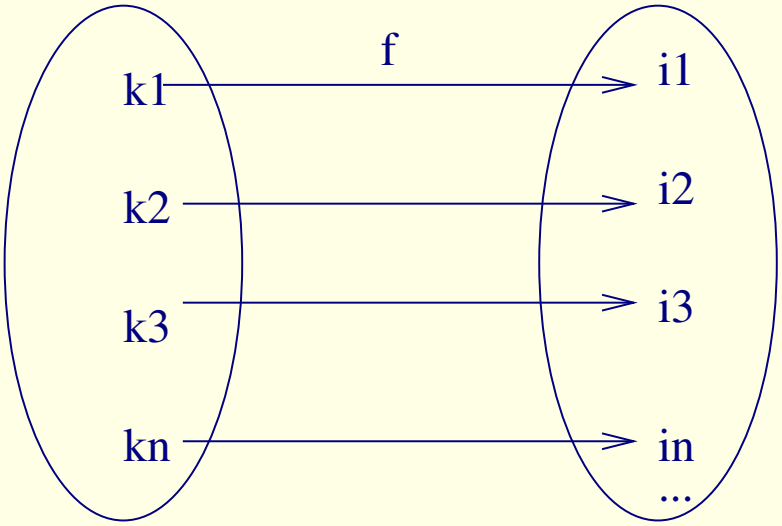
Contingut:

3.4.1 Introducció a les funcions de dispersió

Condicions d'aplicabilitat de la implementació de taules mitjançant funcions injectives:

1. Podem trobar una funció injectiva de cost baix que mapeja l'espai de claus (K) a l'espai d'adreces (A)
2. $| K | \leq | A |$

   (com a màxim tindrem tantes claus com adreces)



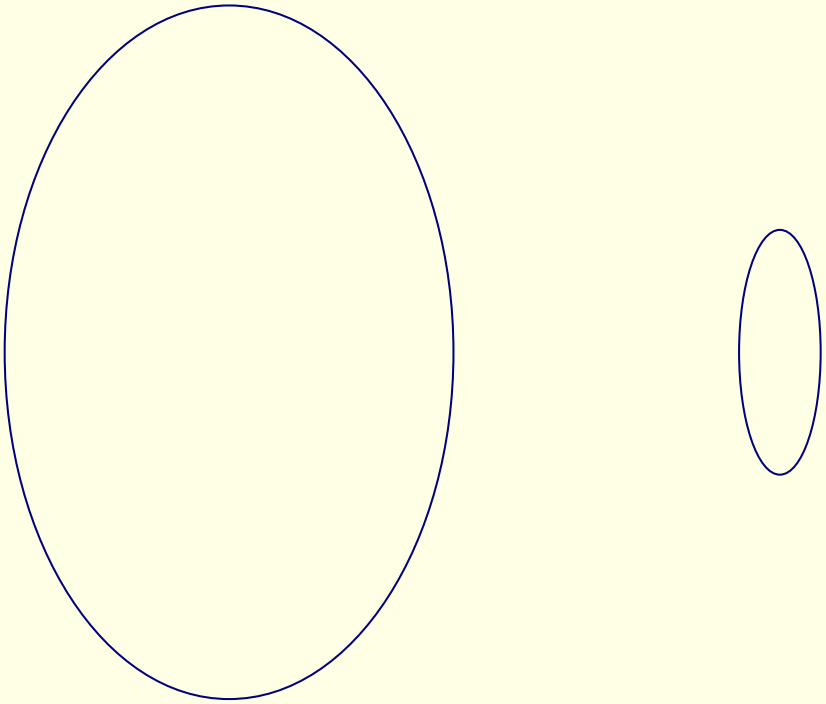
$$\neg\exists k_1, k_2 \in K \texttt{ tal que } f(k_1) = f(k_2)$$

Usualment 2 no es compleix:

Exemple:

Considerem una biblioteca amb 500 socis identificats per nif:

$$\begin{array}{|l} \hline |\ K\ | = 100.000.000 \\ |\ A\ | = 500 \\ \hline \end{array}$$

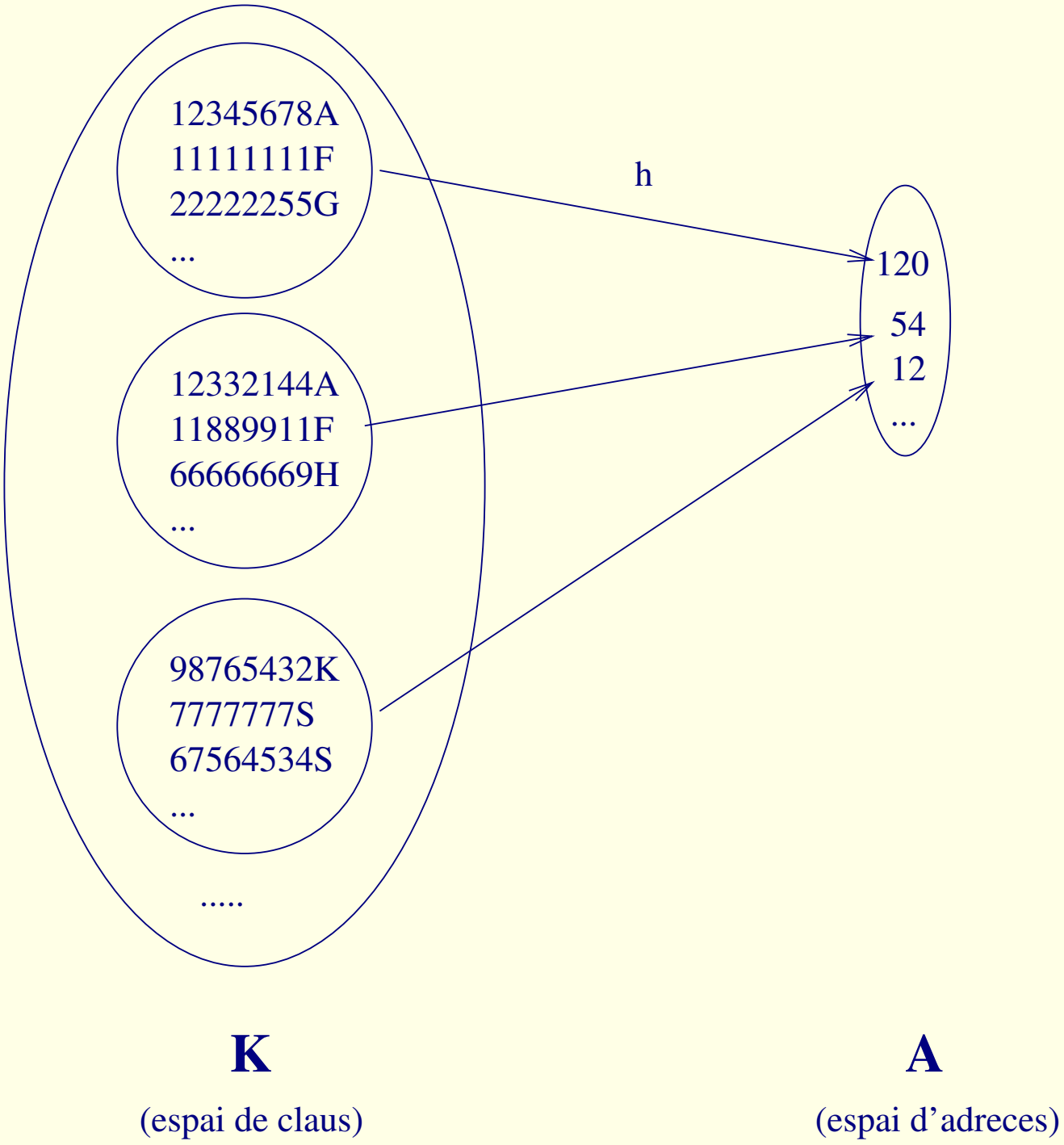


K
(espai de claus)

A
(espai d'adreces)

Usualment la implementació amb funcions injectives no és aplicable

Solució:
Permetre que la funció de mapeig (que anomenarem h) **no sigui injectiva**



Consideracions:

- La funció h mapeja tot un conjunt de claus de K a una mateixa adreça de A!!!!!

- Això no és greu si considerem que de les 100.000.000 de claus disponibles, només n'usarem unes 500

  Per tant, si dimensionem bé l'espai d'adreces A i triem una bona funció h , no hi hauria d'haver moltes claus que siguin enviades per h a la mateixa adreça

- h indueix una relació d'equivalència a K:

$$\forall k_1, k_2 \in K \bullet k_1 \ \texttt{R} \ k_2 \ \texttt{sii} \ h(k_1) = h(k_2)$$

- A és el conjunt quocient d'aquesta relació d'equivalència

Conclusió:

- La implementació de les taules mitjançant funcions de dispersió consisteix a assignar la parella $[k, v]$ a l'adreça de l'espai d'adreces indicada per $h(k)$,

  on h és una funció no injectiva anomenada *funció de hash o de dispersió*:

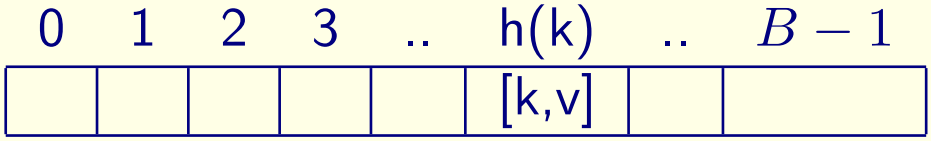
  $h : String \longrightarrow [0..B-1]$

- Aquesta implementació es pot usar si:

  - L'espai de claus potencials és molt més gran que l'espai d'adreces i
  - No cal una obtenció de les claus ordenades

K: String
A=[0..$B-1$]
$h : String \longrightarrow [0..B-1]$

La parella $[k, v]$ se situarà (en principi) a l'índex $h(k)$ del vector:



Dues reflexions:

1. La funció de dispersió h ha de tenir la propietat de **distribuir uniformement** les claus de K a l'espai d'adreces per tal d'evitar que hi hagi adreces a les quals correspongui una acumulació de claus i altres adreces que rebin poques claus o cap

   Quina forma haurà de tenir h per complir la propietat de la distribució uniforme?
   Vegeu 3.4.2

2. Encara que h distribueixi molt bé, serà inevitable que h assigni la mateixa adreça a claus diferents (recordem que $\mid K \mid >>>> \mid A \mid$)

   Per tant tindrem claus k_1, k_2 (amb $k_1 \neq k_2$) t.q. $h(k_1) = h(k_2)$

   Aquestes claus s'anomenen $claus\ sinònimes$

   Quina estratègia utilitzarem per situar dues claus sinònimes a l'espai d'adreces???
   Vegeu 3.4.3 i 3.4.4

3.4.1 Introducció a les funcions de dispersió. Glossari

- **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'adreces:**

  Conjunt de totes les adreces on podem situar les parelles (k, v)

  Considerarem B adreces numerades $0..B-1$

  Depenent de l'estratègia de dispersió (3.4.3-3.4.4) a una adreça s'hi pot mapejar una única parella (k, v) o vàries

  En aquest darrer cas, a una adreça se l'anomena **galleda** (**bucket**)

- **Funció de dispersió (funció de hash)**

  $h : String \longrightarrow [0..B-1]$

  Funció no injectiva que assigna a cada clau de l'espai de claus una adreça de l'espai d'adreces $(0..B-1)$

  Com h no és injectiva, podrà passar que envïi vàries claus a la mateixa adreça

  (Vegeu 3.4.2)

- **Claus sinònimes:**

  Dues claus diferents k_1, k_2 tals 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 3.4.3-3.4.4)

3.4.2 Funcions de dispersió

$h : String \longrightarrow [0..B-1]$
$k \longrightarrow h(k)$

Propietats que ha de complir h:

1. h ha de ser fàcilment computable
2. h ha de mapejar tot l'espai d'adreces (ha de poder generar qualsevol valor entre $[0..B-1]$)
3. h ha de dispersar les claus uniformement en $[0..B-1]$

Què significa la propietat 3 (dispersió uniforme)?

h no ha de donar lloc a acumulacions de claus sobre unes quantes adreces (a les quals h envia moltes claus) mentre que unes altres en reben moltes menys (potser zero)

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 NIFs d'una província tendeix a repetir els dos primers dígits

   (En canvi, la h proposada dispersaria uniformement un conjunt de claus aleatòries)

Dos consells pràctics:

- 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:

  $h(\text{"}k_1k_2k_3\text{"}) \neq h(\text{"}k_2k_1k_3\text{"})$

Descomposició de h:

Podem veure descomposada h en dues funcions h_1, h_2:

$h : String \longrightarrow [0..B-1]$
$h_1 : String \longrightarrow Naturals$
$h_2 : Naturals \longrightarrow [0..B-1]$

de tal manera que $h(k) = h_2(h_1(k))$

Seguidament estudiarem totes dues funcions (h_1, h_2) per separat

3.4.2 Funcions de dispersió:

$$h_1 : String \longrightarrow Natural$$

1. **Suma dels codis de tots els caràcters de la clau:**

   Sigui $k = c_1 \cdot c_2 \cdot ... \cdot c_n$

   (k és la concatenació dels caràcters c_1, c_2, ..., c_n)

$$h_1(k) = \sum_{i=1}^{|k|} \texttt{codi}(c_i)$$

   Característiques:

   - Facilitat de càlcul :-)
   - No sensible a l'ordre dels components de la clau :-(
     $h_1(c_1 \cdot c_2 \cdot c_3) = h_1(c_3 \cdot c_2 \cdot c_1)$
   - Mapeja un espai d'adreces possiblement petit: :-(
     $h_1(k) \leq \texttt{CODIMAX} * \mid k \mid$
     (CODIMAX és el valor màxim que pot prendre codi(c))
   - Les adreces més petites possiblement no seran mapejades :-(

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 el formem les claus ($b = 32, 64, 128, 256...$)

Característiques:

- Permet generar un valor diferent per cada clau diferent :-)
- Permet mapejar espais d'adreces 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 :-(
  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, $h_1(c_1 \cdot c_2 \cdot c_3) = h_1(c_3 \cdot c_1 \cdot c_2)$)

3.4.2 Funcions de dispersió:

$$h_2 : Natural \longrightarrow [0..B-1]$$

(h_2 s'anomena funció de restricció d'un natural a un interval)

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:

- $B = 2 * r$
  $h_2(x)$ conservarà la paritat de 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 de x
- $B = 10^r$
  $h_2(x)$ depèn només dels r darrers dígits decimals de x
- $B = b^r$
  $h_2(x)$ depèn només dels r darrers dígits en base b de x

B ha de ser un nombre primer o, almenys, no ha de tenir divisors menors que 20

2. Plegament

> Particionar la clau en m parts de la mateixa longitud (llevat, potser, de la darrera) i combinar-les usant un determinat operador (suma, or-exclusiu...)

Exemple:

$h_2(154893021938450) = 154 + 893 + 21 + 938 + 450$

Es pot completar usant un mòdul

3. Dígits centrals del quadrat

> $h_2(x) =$ els r dígits centrals de x^2 $(1 \leq r \leq \log_{10} B)$

Els dígits centrals de x^2 depenen de tots els productes parcials, no de cap part concreta de x

3.4.2 Funcions de dispersió: Una funció pràctica

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
- $\texttt{codi}(c_i)$ 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

$$h_2(x) = x \ \texttt{mod} \ B$$

$$h(k) = h_2(h_1(k))$$

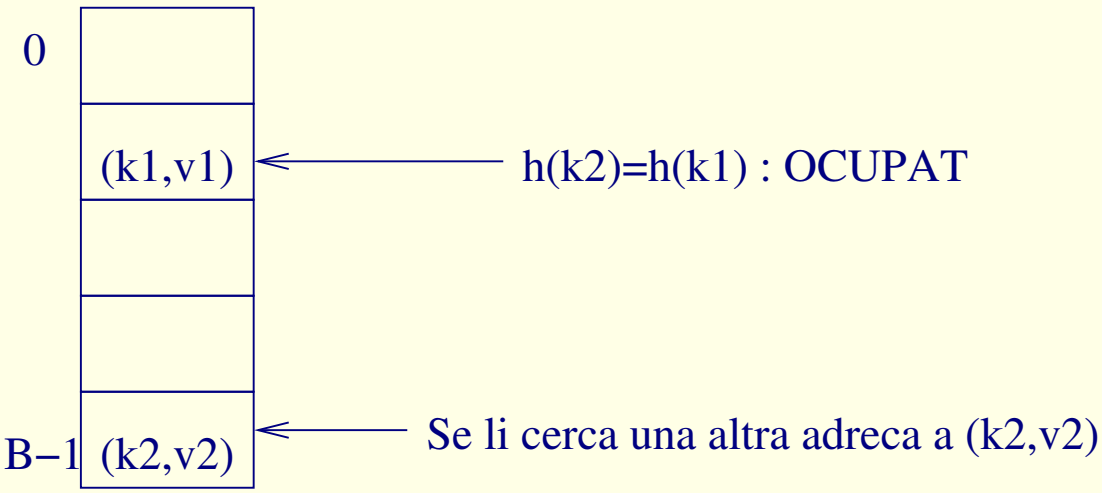
3.4.3 Estratègies de dispersió

Què passa quan volem inserir dues claus k_1, k_2 sinònimes???
$h(k_1) = h(k_2)$

Dues estratègies generals:

1. **A cada adreça del vector: Una única parella (k,v)**

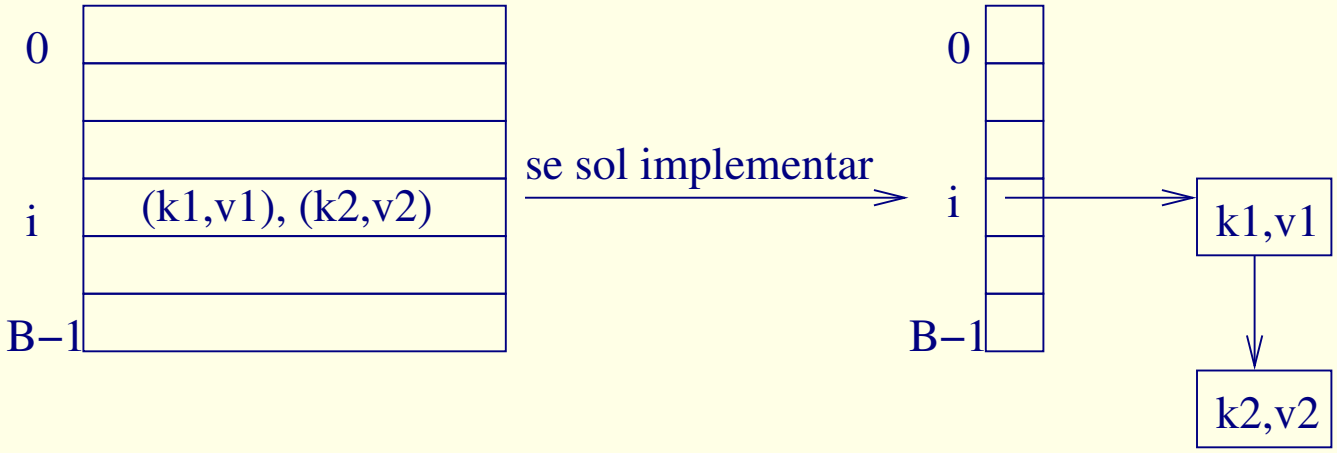
   Per tant, caldrà cercar una nova adreça per col.locar k_2



   Aquesta estratègia s'anomena **hash tancat** o **adreçament obert**

2. A cada adreça del vector: Més d'una parella (k,v)

Cada adreça es veu com una $galleda$ ($bucket$) capaç d'encabir vàries parelles (k,v)



Aquesta estratègia s'anomena **hash obert** o **hash encadenat**

3.4.3 Estratègies de dispersió: Hash tancat

Estratègia:
Totes les parelles (k, v) s'allotgen a la taula principal, per bé que les que col.lisionen s'allotgen fora de la seva adreça original
Intrús:
Clau k que està situada en una posició diferent de $h_0(k)$

- Cada clau haurà de tenir una llista d'adreces alternatives que seran provades en ordre fins trobar-ne una que estigui lliure

$$h(k) = h_0(k),\ h_1(k),\ h_2(k), ...,\ h_{B-1}(k)$$

Aquesta llista d'adreces alternatives s'anomena **seqüència de proves de la clau** k

$h_i(k)$ és l'adreça on intentarem col.locar la clau k si les adreces $h_0(k), h_1(k), ..., h_{i-1}(k)$ estaven ocupades

Les h_i $(0 \leq i < B)$ s'anomenen famílies de funcions de dispersió

- L'algorisme d'eliminació haurà de marcar l'adreça d'on fa l'eliminació com a $esborrada$ per tal que l'algorisme de consulta no interrompi el recorregut de la llista de sinònims a l'adreça esborrada

- Una adreça marcada com a $esborrada$ serà considerada com a
  - $lliure$ per l'algorisme d'inserció
    (però compte!!: Abans de poder usar una adreça esborrada, l'algorimse d'inserció s'haurà d'assegurar que la clau que es vol inserir no es troba ja a la taula)
  - $ocupada$ per l'algorisme de consulta

3.4.3 Hash tancat. Algorisme d'inserció

acció inserir (k: TClau, v: TValor) **és**
  $lliure$:=fals;
  $trobEsborrada$:=fals;
  $trobat$:=fals;
  $i := 0$;
  mentre ($i < B \wedge \neg lliure \wedge \neg\ trobat$) **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

- $trobat$=cert: La clau k es present al vector t a la posició pos

- $trobEsborrada$=cert: S'ha trobat una posició amb $marca$=ESBORRADA

- $lliure$=cert: S'ha trobat una posició amb $marca$=LLIURE

Notem que 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

3.4.3 Hash tancat. Algorisme de consulta

acció consulta (k: TClau, v:**sort** TValor, $trobat$: **sort** booleà) **és**
  $lliure$:=fals;
  $trobat$:=fals;
  $i := 0$;
  mentre $(i < B \wedge \neg lliure \wedge \neg\ trobat)$ **fer**
   pos:= $h_i(k)$;
   si $t[pos].marca$= LLIURE **llavors** $lliure$:= cert;
   si no si $t[pos].clau = k \wedge t[pos].marca$=OCUPADA **llavors**
     $trobat$:=cert;
   si no $i := i + 1$; **fsi**
  fmentre
  si $trobat$ **llavors** $v := t[pos].valor$; **fsi**
facció

- $trobat$=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 fer B intents o fins que es trobi una posició amb marca=LLIURE

3.4.3 Hash tancat. Algorisme d'eliminació

acció eliminació (k: TClau) **és**
  $lliure$:=fals;
  $trobat$:=fals;
  $i := 0$;
  mentre $(i < B \wedge \neg lliure \wedge \neg\ trobat)$ **fer**
   pos:= $h_i(k)$;
   si $t[pos].marca$= LLIURE **llavors** $lliure$:= cert;
   si no si $t[pos].clau = k \wedge t[pos].marca$=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

3.4.3 Hash tancat. Famílies de funcions de dispersió

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

Tres famílies de funcions de dispersió:

- Dispersió lineal
- Dispersió quadràtica
- Dispersió doble

Dues propietats interessants per una família de funcions de dispersió:

1. Totes les adreces j de la taula t són candidates a allotjar una clau k qualsevol:

   $\forall k \in String, \forall j : 0 \leq j < B \bullet (\exists i : 0 \leq i < B \bullet h_i(k) = j)$

   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 explicació més endavant)

3.4.3 Hash tancat. Dispersió lineal

$$h_i(k) = (h(k) + c * i) \ \texttt{mod} \ B$$

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 son primers entre sí, 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 de dispersió lineal

$B = 11$
$h(k) = k \ \texttt{mod}\ 11$
$h_i(k) = (h(k) + i) \ \texttt{mod}\ 11$
(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

La dispersió lineal genera problemes d'apinyament primari

3.4.3 Hash tancat. Apinyament primari

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:

$$\forall i : 0 \leq i \leq B-1 \bullet h_i(k) = h_i(k_2)$$

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 per a que l'intrús k_2 es posés a l'adreça que correspon a k: $h_0(k) = h_j(k_2)$):

$$\exists j : 0 < j < B \bullet \forall i : 0 \leq i < B \bullet h_i(k) = h_{j+i}(k_2)$$

La família de funcions de dispersió lineal genera 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)

3.4.3 Hash tancat. Dispersió quadràtica

$$h_i(k) = (h(k) + c * i^2) \ \texttt{mod} \ B$$

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. Veure 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 de dispersió quadràtica

$B = 11$
$h(k) = k \ \texttt{mod} \ 11$
$h_i(k) = (h(k) + i^2) \ \texttt{mod} \ 11$
(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

La dispersió quadràtica elimina els problemes d'apinyament primari i genera problemes d'apinyament secundari

3.4.3 Hash tancat. Apinyament secundari

Apinyament secundari:

Es diu que una família de funcions de dispersió genera apinyament primari 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

La família de funcions de dispersió quadràtica genera apinyament secundari:

- Les claus sinònimes (27, 49, 104, 16) tenen la mateixa seqüència de dispersió

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 molt elevat

3.4.3 Hash tancat. Dispersió doble

$$h_i(k) = (h(k) + i * h'(k)) \ \texttt{mod} \ B$$

on h' és una altra funció de dispersió

Propietats:

- És poc probable que dues claus que siguin sinònimes respecte la funció h també ho siguin respecte h'
- La dispersió doble elimina el problema de l'apinyament secundari

Exemple de dispersió doble

$B = 11$

$h(k) = k \ \texttt{mod} \ 11$

$h'(k) = \sum_{r=1}^{r=|k|} k_i$
(Suposem $k = k_1 \cdot k_2 \cdot ... \cdot k_n$; 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

La dispersió quadràtica elimina els problemes d'apinyament primari i secundari

3.4.3 Hash tancat. Implementació

La implementació d'un hash la parametritzarem respecte:

- El tipus dels valors: `TVALOR`
- La mida del vector: `B`
- La funció de hash: `FH`

- **Fitxer HashMap.h**

```
template <class TVALOR>
class Node{
public:
  TVALOR val;
  MyString key;
  int marca; //FREE,OCCUPIED,DELETED
};

template <class TVALOR, class FH, unsigned int B>
class HashMap :public Map<T>{
  Node<TVALOR> v[B];
  FH h;
public:
....
void put(const MyString pkey, TVALOR& val,
         HashMapIterator<TVALOR>& it)
{
 ....
 pos=h(k,i);
 ....
}
```

- **Fitxer FHash.h**

```
template <unsigned int B>
class FHash{
public:
 unsigned int operator()(const MyString k, unsigned int i)
 {

  ....implementacio de h(k,i)
 }
};
```

- **Fitxer Client.cc**

```
#include "FHash.h"
#include "HashMap.h"
#include "HashMapIterator.h"

class Student{
 .....
};

int main()
{
  HashMap<Student, FHash<1013>, 1013> m;

  m.put(k,v,it);

}
```

3.4.3 Hash tancat. Variants

Variant Robin Hood

Idea:

- 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 altres **castigades** que han necessitat moltes proves per tal de ser inserides

Esquema de l'algorisme d'inserció d'una clau

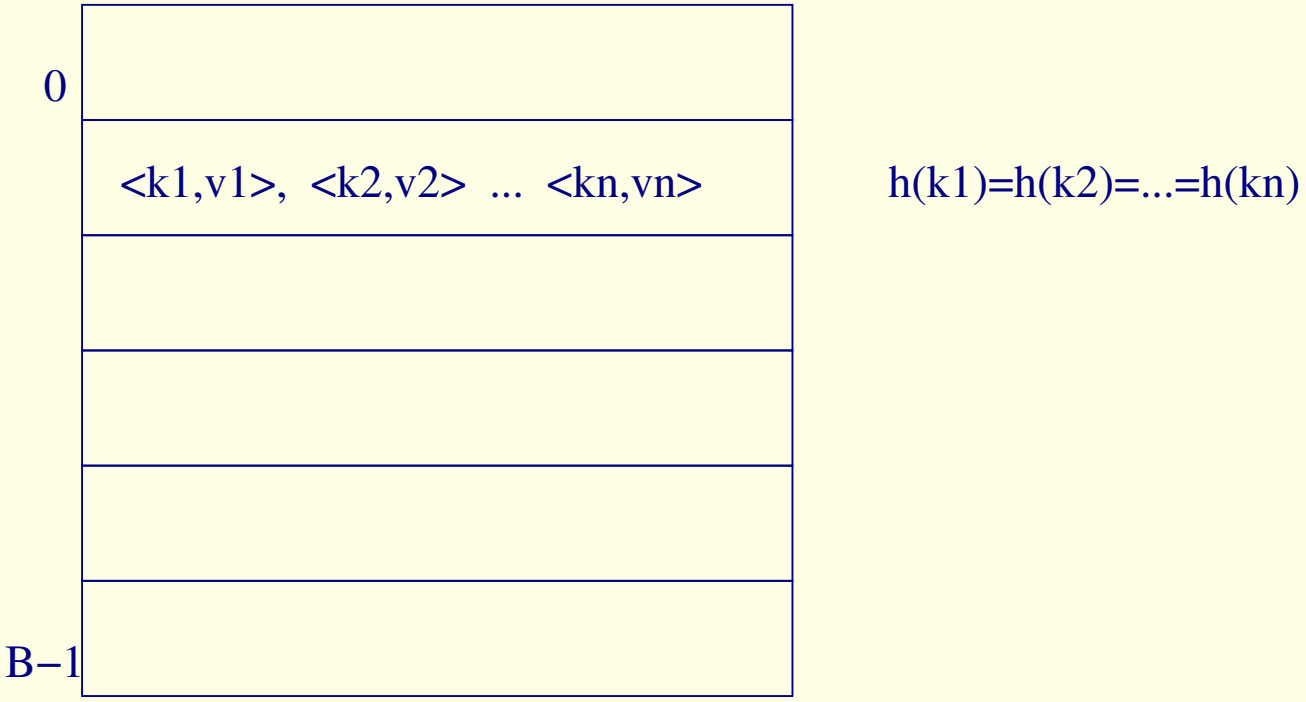
Desenvolupament de la prova i per situar la clau k:

- $pos = h_i(k)$
- 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 $h_{j+1}(k')$

3.4.4 Estratègies de dispersió. Hash obert

Idea:

Cada adreça (`0..B-1`) representa una *galleda* (o *bucket*) on hi cap més d'una clau amb el seu valor associat



El hash obert evita l'apinyament primari i les tècniques de readreçament

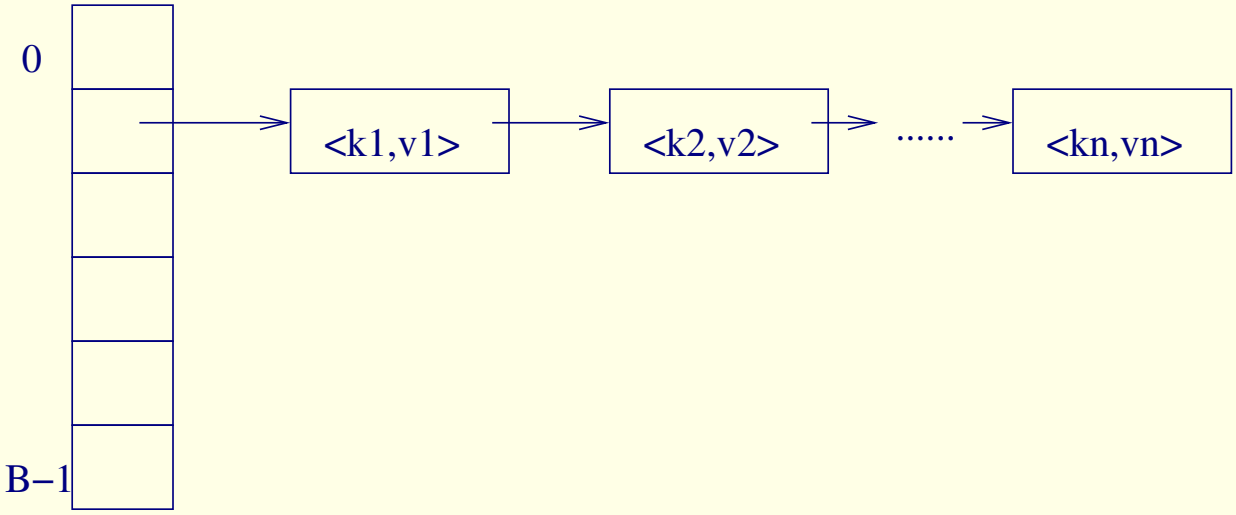
Implementació del hash obert:

Usualment es consideren dues tècniques:

- La llista de sinònims es manté encadenada en memòria dinàmica
- La llista de sinònims es manté en un *vector d'excedents*

Usualment, al hash obert també se l'anomena *hash encadenat*

3.4.4 Hash obert. Encadenaments 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!!! el hash 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 apuntadors

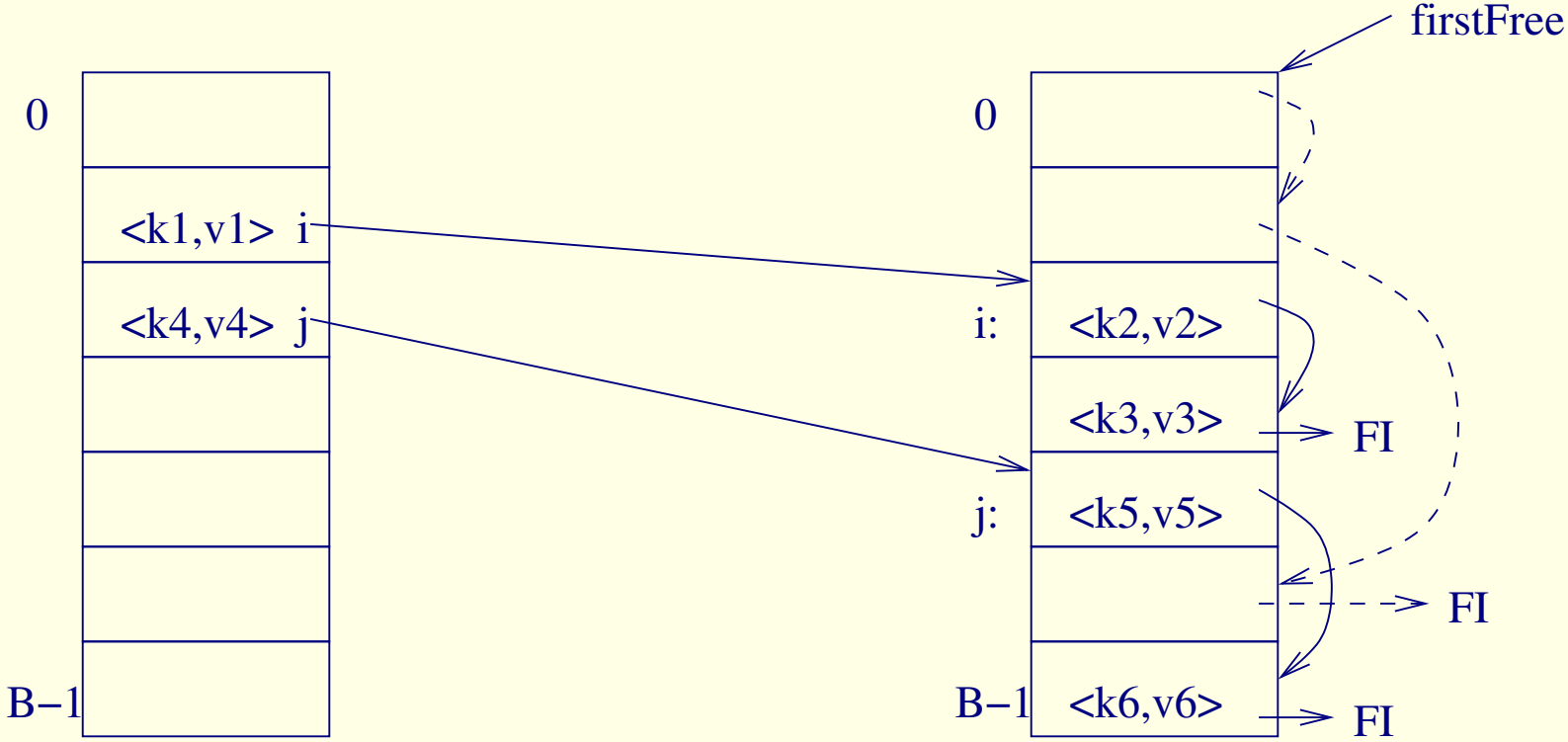
Esquema d'implementació

```
template <class T>
class Node{
 MyString key;
 T val;
};

template <class T,class FH,unsigned int B>
class ChainedHashMap :public Map<T>{

  LinkedList<Node<T> > v[B];
  FH h;
public:

  //Operations....
};
```

Observacions:

- Reutilitzem la classe `LinkedList`
- La classe `ChainedHashMap` segueix essent genèrica respecte:
  - El tipus dels valors emmagatzemats (`T`)
  - La funció de dispersió (`FH`)
  - La mida del vector (`B`)

3.4.4 Hash obert. Encadenaments en zona d'excedents



Zona principal

Zona excedents

h(k1)=h(k2)=h(k3)
h(k4)=h(k5)=h(k6)

Encadenaments de posicions lliures

Encadenaments de sinonims

3.4.4 Hash obert. Encadenaments en zona d'excedents

Característiques:

- No cal usar apuntadors (els apuntadors són, en realitat, índexos d'un array)

- 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

```
template <class T>
class Node{
 MyString key;
 T val;
 int next;
};

template <class T,class FH,unsigned int B>
class ChainedHashMap2 :public Map<T>{

  Node<T>  mainZone[B];
  Node<T>  secZone[B];
  int firstFree;
  FH h;
public:

  //Operations....
};
```

Observacions:

- `mainZone` és l'array que allotja la zona principal
- `secZone` és l'array que allotja la zona d'excedents.

  La capacitat de la zona d'excedents pot ser diferent
- `firstFree` fa referència al primer element lliure de la zona d'excedents (`secZone`)
- `next` permet l'encadenament de les posicions lliures i dels sinònims

3.4.5 Estratègies de dispersió. Cost

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 un hash tancat 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 un hash tancat per trobar una clau present en una taula amb capacitat per a B elements i N elements presents

Per estudiar els costos calcularem del hash tancat i del hash obert calcularem T_f i T_s

Per fer aquests càlculs ens ajudarem de E

3.4.5 Cost hash tancat

Càlcul de $E(i, B)$

- $E(0, B) = 1$

  Per inserir el primer element en una taula buida només cal fer una prova

- $E(i, B) = \frac{B-i}{B} * 1 + \frac{i}{B} * (1 + E(i-1, B-1))$

  Per què?:

  - Es fa la hipòtesi que h **distribueix uniformement [1]**,
  - 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 array 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 ratio 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)

- $T_e(N,B) \approx \frac{E(0,B)+E(1,B)+...+E(N-1,B)}{N} \approx \frac{1}{\alpha} * Ln(\frac{1}{1-\alpha})$

  (Promig 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 un hash tancat depèn:

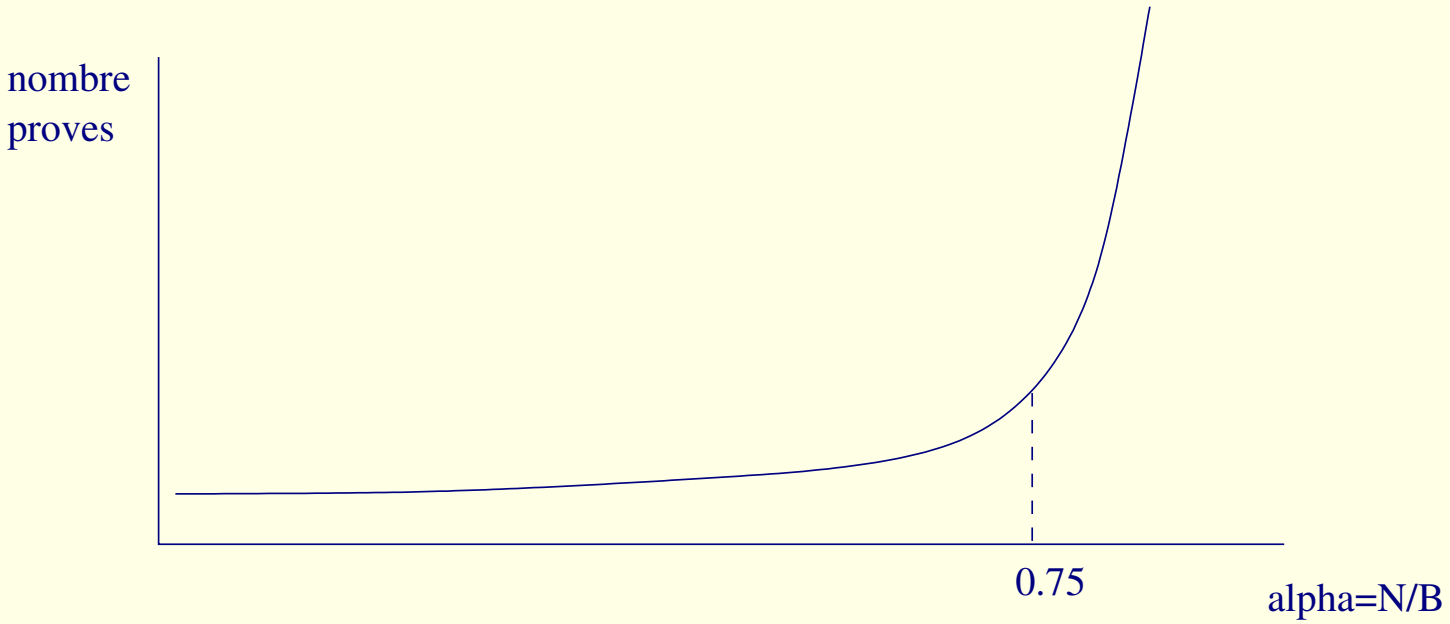
1. De la bondad de h:
   - h ha de distribuir uniformement les claus que es volen inserir
   - h no ha de generar posicions repetides en proves diferents
   - h s'ha de poder calcular eficientment
2. De la ratio d'ocupació de la taula (α)

   El cost no depèn directament del nombre d'elements emmagatzemats a la taula sinó de la ratio entre el nombre d'elements emmagatzemats i la capacitat

- $\alpha = 0.5: \quad T_f = 2 \;\; T_e = 1.39 \Longrightarrow O(1)$
- $\alpha = 0.75: \quad T_f = 4 \;\; T_e = 1.85 \Longrightarrow O(1)$

  Per 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 hash tancat tal que:

- Està dotada de una funció de dispersió que dispersi uniformement les claus que volem introduir a la taula

- Té una ratio d'ocupació menor o igual al 75%

té un cost dels algorismes de inserció/eliminació/consulta d'una clau de O(1)

3.4.5 Cost hash obert

Consideracions

- Fem novament la hipòtesi que h distribueix uniformement
- En un hash obert, N pot ser més gran que B (ja que en una posició de la taula hi pot haver més d'una clau)
- El nombre promig 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 del hash tancat, 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/eliminacions en una taula implementada com a hash obert: $O(1)$

3.4.5 Comparació de costos

Cost temporal hash obert vs. hash tancat

- Quantitativament, els resultats anteriors de T_f i T_s afavoreixen una mica al hash obert

- 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 de la classe `HashMap<T,FH,B>` que el programadador pot controlar

- Si una funció de hash distribueix malament només un percentatge de les claus de la taula, el hash tancat farà degenerar **tota la taula** mentre que en un hash obert només es veuran afectades les claus mal distribuïdes

3.4.6 Estratègies de dispersió. Conclusió

- 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 mapejar en un espai d'adreces significativament més petit (e.g., Els 500 usuaris d'una biblioteca identificats per NIF)
  - Podem trobar una funció de hash que distribueixi uniformement les claus de la nostra situació concreta i que es pugui calcular eficientment
  - Mantenim la taula amb una ratio 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

3.5 Els fitxers relatius

Contingut

3.5.1 Idea intuïtiva

Amb els fitxers seqüencials no podíem accedir de manera eficient a un element (enregistrament) del fitxer (i.e., per accedir a un element particular havíem de passar per tots els anteriors)

Els fitxers relatius ens permetran accedir molt eficientment a l'element que ocupa la posició p del fitxer.

Podem imaginar un fitxer relatiu com un vector amb dues propietats addicionals:

1. Persistència (pel seu emmagatzemament en memòria externa)

2. La capacitat del fitxer relatiu és indefinida

Un fitxer relatiu és un fitxer que permet accés eficient (per lectura, escriptura o modificació) a l'element que ocupa la posició p
La capacitat d'un fitxer relatiu és indefinida

3.5.1 Idea intuïtiva

A més a més, les tècniques de dipersió ($hash$) seran aplicables als fitxers relatius de tal manera que podrem accedir eficientment, no només a un element del fitxer per posició, sinó també per clau

Per tal de treballar amb fitxers relatius definirem una classe `DirectFile<T>`, que modelitzarà els fitxers relatius de tipus component genèric `T`

3.5.1 Operacions

Les operacions més importants de la classe `DirectFile<T>` són les següents:

- `f.open(nomFitxer)`

  Lliga un fitxer físic del disc anomenat `nomFitxer` amb un objecte del programa `f`.

- `f.close()`

  Deslliga l'objecte fitxer relatiu `f` del fitxer físic al qual s'havia lligat per l'operació `open`.

- `f.getState()`

  Indica l'estat en què ha quedat l'objecte `f` després de la darrera operació feta sobre ell. 0 indica que tot ha anat bé. Considerarem 6 possibles causes d'error (vegeu especificació).

- `f.readDir(x, pos)`

  Llegeix l'element del fitxer `f` que es troba a la posició `pos` i el posa a `x`. (`x` es un objecte de classe T). Després de cridar aquesta operació caldrà consultar l'estat del fitxer (e.g., podríem haver llegit una posició incorrecta).

- `f.locate(pos)`

  Posiciona el punter de lectura del fitxer a l'element que ocupa la posició `pos`.

- `f.readSeq(x)`

  Llegeix el següent element de l'objecte fitxer seqüencial `f` i el posa a `x`. (`x` es un objecte de classe `T`). Per poder cridar aquesta operació cal haver fet primer una lectura directa o bé un posicionament.

- `f.endOfFile()`

  Indica si el procés de lectura ha arribat a la fi del fitxer (i.e., si ha llegit tots els elements de classe `T` del fitxer `f` i, a més a més, la marca de final).

- `f.write(x,pos)`

  Escriu `x` a la posició `pos` del fitxer `f`. Si la posició `pos` era verge ara ha estat escrita. Si contenia algun element, aquest element ha estat actualitzat per `x`. (`x` és una objecte de classe `T`).

3.5.2 Classe `DirectFile`. Especificació

(f' fa referencia a l'objecte abans d'una operacio determinada)

- `DirectFile(const MyString);`

  `DirectFile<T> f(nomfit);`

  Pre:

  Post:

  S'ha lligat l'objecte `f` amb el fitxer extern de nom `nomfit` per fer-hi operacions de lectura/escriptura i `f.getState()=0`. Si no existia cap fitxer extern de nom `nomfit`, se'n crea un de nou, buit

  Si `f'` ja estava obert, `f.getState()=1`

  Si s'ha produït algun error d'E/S, `f.getState()=3` i `f` no ha quedat vinculat a `nomfit`

- `void open(const MyString);`

  `f.open(nomf);`

  Mateix comportament que l'operacio constructora

- `void close();`

  `f.close();`

  Pre:

  Post:

  Si `f` està obert, això és, si `f` està lligat a algun fitxer extern `F`, es deslliga de `F`. Ja no es poden fer operacions de lectura/escriptura sobre `f` fins que no es torni a obrir.

  Si `f` no està obert, `f.getState()=2`

- `int getState();`

  `i=f.getState();`

  Pre:

  Post:

  `i` conté l'estat després de la darrera operació realitzada sobre `f`.

  - `i=0` $\longrightarrow$ La darrera operació ha tingut èxit.
  - `i=1` $\longrightarrow$ La darrera operació ha produït un error de FITXER OBERT.
  - `i=2` $\longrightarrow$ La darrera operació ha produït un error de FITXER NO OBERT.
  - `i=3` $\longrightarrow$ La darrera operació ha produït algun altre tipus d'error d'E/S
  - `i=4` $\longrightarrow$ La darrera operació ha produït un error de FALTA ESPAI EN DISC.
  - `i=5` $\longrightarrow$ La darrera operació ha produït un error de LECTURA MES ENLLA DE FDF.
  - `i=6` $\longrightarrow$ La darrera operació ha produït un error de POSICIO INCORRECTA

  Observacions: Operació `const`

- `void readDir(T&, int);`

  `f.readDir(x,pos);`

  Pre:

  Post:

  `x` conté una còpia de l'element que ocupa la posicio `pos` del fitxer relatiu `f` i el capçal de lectura de `f` ha quedat posicionat a `pos+1` i `f.getState()=0`. (El primer element ocupa la posicio 0. L'element i-èsim de tipus `T` emmagatzemat al fitxer ocupa la posicio `i-1`).

  Si `pos`=f.getSize(), `f.endOfFile()=cert, x=indefinit, f.getState()=0`

  Si `pos` és una posició incorrecta (`pos` <0 o `pos`> f.getSize()) `f.getState()=6`

  Si `f` no ha sigut obert prèviament, `f.getState()=2`

- `void readSeq(T&);`

  `f.readSeq(x);`

  Pre:

  Abans de cridar a aquesta operacio s'ha de cridar a `f.readDir(..)` o a `f.locate(..)` .

  Post:

  `x` conté l'element de tipus `T` de `f` sobre el qual estava posicionat el capçal de lectura abans de l'execució de l'operació i el capçal de lectura s'ha desplaçat una posició endavant i `f.getState()=0`.

  Si el capçal de lectura abans de l'execució de l'operació estava posicionat sobre `<EOF>`, `x` és indefinit i `f.endOfFile()=cert` i `f.getState()=0`

  Si `f` no ha sigut obert prèviament, `f.getState()=2`

  Si `f'.endOfFile()=cert`, `f.getState()=5` i `f.endOfFile()=cert`.

- `bool endOfFile();`

  `b=f.endOfFile();`

  Pre:

  Post:

  `b=cert` si el capçal de lectura del fitxer relatiu `f` està situat en una posició indefinida més enllà de `<EOF>`; en cas contrari `b=fals`. `f.getState()=0`

  Si `f` no ha sigut obert, `f.getState()=2` i `b=indefinit`.

- `void write(const T&, int);`

  `f.write(x,pos);`

  Pre:

  Post:

  S'ha escrit `x` a la posició `pos` de `f` i `f.getState()=0`.

  Si la posició `pos` contenia algun element, aquest element ha estat substituït per `x`. (`x` és una objecte de classe `T`).

  Si `pos` és una posició incorrecta (`pos` <0 o `pos` $>$ `f.getSize()`), `f.getState()=6`

  Si no hi ha prou espai al disc o es produeix algun problema en l'escriptura, `f.getState()=4` i `f` queda en estat indefinit.

  Si `f` no està obert, `f.getState()=2`

  Observació: Quan s'escriu en un fitxer de la classe `DirectFile` no es poden deixar posicions intermèdies buides. O sigui, si un `DirectFile` té 6 elements (ocupats entre les posicions 0..5) no es pot fer, per exemple: `f.write(x,10);` . Només es pot escriure a una posició entre 0 i 6. Si s'escriu a la posició 6, s'augmenta el nombre d'enregistraments del fitxer en un.

- `void locate(long);`

  `f.locate(pos);`

  Pre:

  Post:

  El capçal de lectura s'ha posicionat sobre la posició `pos` de `f`. La propera operació `f.readSeq(x)` llegirà l'element de la posició `pos`.

  Si `pos` és una posició incorrecta (`pos` <0 o `pos`> f.getSize()) `f.getState()`=6.

  Si `f` no està obert, `f.getState()`=2.

- `long getSize();`

  `i=f.getSize();`

  Pre:

  Post:

  `i` és el nombre d'elements de tipus `T` continguts al fitxer relatiu `f`. `f.getState()`=0.

  Si `f` no està obert, `f.getState()`=2.

- `bool getOpen();`

  `b=f.getOpen();`

  Pre:

  Post:

  b és cert si `f` està obert i fals altrament. `f.getState()=0`.

- `MyString getName();`

  `namef=f.getName();`

  Pre:

  Post:

  `namef` és el nom del fitxer `f` i `f.getState()=0`.

  Si `f` no està obert, `f.getState()=2` i `namef` és indefinit.

- `bool operator==(const Object&);`
  - **Crida:** `b=(f==f2);`
  - **Prec:**
  - **Post:**
    b és cert si `f` i `f2` són dos fitxers directes que contenen els mateixos elements.
    `f=f'` i `f2=f2'`

- `void copy(const Object&);`
  - **Crida:** `f.copy(f2);`
  - **Prec:** `f` és un fitxer directe obert i vinculat a un fitxer físic.
  - **Post:**
    `f` és un fitxer directe que conserva el mateix nom que abans de la crida a l'operació però que conté una còpia del fitxer directe `f2`, el qual, no s'ha modificat.

- `MyString toString();`
  - **Crida:** `st=f.toString();`
  - **Prec:**
  - **Post:**
    `st` conté una representació textual del contingut del fitxer directe `f`. `f` no s'ha modificat.
  - **Observacions:** Operació virtual i const.

3.5.3 Classe `DirectFile`. Implementació amb *streams*

Strams amb accés directe

La biblioteca estàndar de C++ ofereix una sèrie d'operacions que ens ajuden a fer lectures/escriptures sobre una posició determinada d'un fitxer:

Classe `istream`

- `istream& seekg(streampos pos);`

  Canvia la posició actual del punter de lectura del fitxer al byte número `pos` (començant des de 0) d'aquest fitxer.

- `istream& seekg(streamoff desplac, ios::seed_dir origen);`

  Desplaça la posició actual del punter de lectura del fitxer `desplac` bytes a partir de la posició indicada per `origen`. Hi ha tres posicions `origen` definides:

  - `ios::beg`: Des de l'inici del fitxer
  - `ios::car`: Des de la posició actual del fitxer
  - `ios::end`: Des del final del fitxer

- `streampos tellg();`

  Retorna la posició actual del punter de lectura del fitxer en termes de posició absoluta (mesurada com a nombre de bytes des de l'inici del fitxer)

  El tipus `streampos` està definit a `iostream.h` i sol implementar-se com un `long`

Classe `ostream`

Les mateixes operacions que abans però, en aquest cas, relatives al capçal d'escriptura

- istream& seekp(streampos pos);
- istream& seekp(streamoff desplac, ios::seed_dir origen);
- streampos tellp();

3.5.3 Classe DirectFile. Implementació amb *streams*

Exemple d'ús d'aquestes operacions

```
#include <fstream>
void main()
{
  int i =30;
  fstream f;

  f.open("prova.dat", ios::out| ios::binary);
  f.seekp(17,ios::beg);
  f.write((char *)&i, sizeof (int));
  f.close();

  f.open("prova.dat", ios::in| ios::binary);
  f.seekg(17,ios::beg);
  f.read((char *)&i, sizeof (int));
  f.close();

  cout<<"i="<<i<<endl;
}
```

3.5.3 Classe `DirectFile`. Implementació

Pràctica 3.2

3.6 Taules persistents. Implementació amb fitxers d'accés directe

Contingut:

3.6.1 Especificació de la taula persistent

Objectiu:

Dotar la classe `HashMap` (que conté la implementació d'una taula en forma de taula de dispersió) de persistència.

Operacions:

A més a més de les operacions pròpies de les taules, les taules persistents defineixen les següents:

- `t.open(nomf)`

  Vincula una taula persistent `t` al fitxer físic anomenat `nomf`, que servirà per emmagatzemar-la.

- `t.close()`

  Acaba la vinculació de la taula persistent `t` al fitxer físic que l'emmagatzemava.

3.6.1 Especificació de la taula persistent

- `HashMapP<T,FH,B> t`

  Pre:

  Post:

  `t` és una taula implementada com a taula de dispersió i persistent. Els valors emmagatzemats a `t` són de tipus genèric `T`, la capacitat inicial de la taula és `B` (unsigned int) i la funció de dispersió s'encapsula a la classe genèrica `FH`.

  La classe que instancïi `FH` haurà de sobrecarregar l'`operator()` de la manera següent:

  ```
  unsigned int operator()(char* k, unsigned int i)
  ```

- `HashMapP<T,FH,B> t(char* nomf)`

  Pre:

  Post:

  Mateix comportament que `t.open(nomf)`

  (vegeu tot seguit)

- `void open(char* nomf)`

  `t.open(nomf);`

  Pre:

  Post:

  Ha vinculat la taula persistent `t` al fitxer relatiu anomenat `nomf`, que servirà per emmagatzemar-la.

  Si ja existeix un fitxer físic amb nom `nomf`, la taula `t` s'obre amb els elements continguts en aquest fitxer

  Si no existeix cap fitxer anomenat `nomf` se'n crea un de nou i la taula `t` s'obre buida.

  Si el fitxer físic `nomf` al que es vol vincular la taula `t` existeix però no té la mida declarada a la seva creació (`B` elements) es llença l'excepció `IncorrectFileException`

  Si `t` ja estava vinculada a un fitxer, o hi ha algun altre problema amb l'apertura es llença l'excepció `IOException`

- `void close()`

  `t.close();`

  Pre:

  Post:

  Ha acabat la vinculació de la taula persistent `t` al fitxer físic que l'emmagatzemava

  Si `t` no estava vinculada a cap físic, llença l'excepció `IOException`

- `void put(char* k, const T& v, MapIterator<T>& it);`

  `t.put(k,v,it);`

  Pre:

  Post:

  – si existeix `v1:T` t.q. `[k,v1]` pertany a `t'`:
  `t=(t'-[k,v1])` $\cup$ $\{$`[k,v]`$\}$ i `t(it)=<s1,[k,v]*s2>`
  de tal manera que `<s1,[k,v]*s2>` conté una certa seqüenciació dels elements de `t`.
  – si no:
  `t=t'` $\cup$ $\{$`[k,v]`$\}$ i `t(it)=<s1,[k,v]*s2>` de tal manera que <s1,[k,v]*s2> conte una certa sequenciació dels elements de `t`.
  (Si `t'` conte un parell `[k,v1]`, el valor `v1` és substituït per `v` si no `t= t'` amb el parell `[k,v]` afegit. `it` es refereix al valor `v` que acabem d'inserir a `t`.)

  Si `t'` no està vinculada a cap fitxer, es llença l'excepció `IOException`

  Si el parell `(k,v)` no es pot inserir a `t'` perquè la taula es plena, es llença l'excepció `FullMapException`

- `void remove(char* k)`

  `t.remove(k);`

  Prec:

  Post:

  – si existeix `v:T` t.q. `[k,v]` pertany a `t'`: `t=t'-[k,v]`
  – si no `t=t'`

  Si `t'` no està vinculada a cap fitxer, es llença l'excepció `IOException`

- `void remove(MapIterator<T>& it)`

  `t.remove(it);`

  Prec:

  Post:

  si `t(it)'=(s1,[k,v]*s2)` aleshores `t(it)=(s1,s2)`

  si `t(it)'=(s1,`$\emptyset$`)` aleshores `t=t'` i s'ha llençat l'excepció `IteratorException`

  (Elimina la parella [k, v] a la qual s'està referint l'iterador `it`. Si aquest iterador no s'està referint a cap element de `t'`, llença una excepció `IteratorException`)

  Si `t'` no estava vinculada a cap fitxer, llença l'excepció `IOException`

- `void get (char* k, T& v, bool& trobat)`

  `t.get(k,v,trobat);`

  Prec:

  Post:

  – si existeix `v1:T` t.q. `[k,v1]` pertany a `t'`: `v=v1` i `t=t'` i `trobat=cert`
  – si no: `t=t'`, `v` és indefinit i `trobat=fals`

  Si `t'` no estava vinculada a cap fitxer físic, es llença l'excepció `IOException`

  Observacions: Operació const.

- `void get (const IteradorTaula<T>& it, T& v)`

  `t.get(it,v);`

  Prec:

  Post:

  si `t(it)'=(s1,[k,v1]*s2)` aleshores `v=v1` i `t=t'`

  si `t(it)'=(s1,`$\emptyset$`)` `t=t'` i s'ha llençat l'excepció `IteratorException`

  (`v` és una còpia del valor al que es refereix `it` dins `t`. Si `it` no es refereix a cap element de `t`, llença una excepció `IteratorException`)

  Si `t'` no estava vinculada a cap fitxer, llença l'excepció `IOException`

  Observacions: `t.get(it,v)` és equivalent a `v=*it;`

  Operació const

- `void get (char* k, IteradorTaula<T>& it, bool& trobat)`

  `t.get(k,it,trobat);`

  Prec:

  Post:

  – si existeix `v1:T` tal que `[k,v1]` pertany a `t'` aleshores `*it=v1` i `t=t'` i `trobat=cert`
  – si no, `t=t'`, `it` és indefinit i `trobat=fals`

  (Retorna un iterador al valor associat a la clau `k` dins de la taula `t`)

  Si `t` no estava vinculada a cap fitxer, llença l'excepció `IOException`

  Observacions: Operació const.

- `bool operator==(const Container& c)`

  `b=(t1==t2);`

  Pre:

  Post:

  `b` és cert si les taules `t1` i `t2` contenen el mateix nombre d'elements i amb exactament les mateixes claus

  Si `t1` o `t2` no estaven vinculades a cap fitxer, llença l'excepció `IOException`

  Observacions: Operació const.

- `Container& operator=(const Container& c)`

  Operació no permesa

3.6.2 Implementació amb fitxers relatius

Idees principals de la implementació:

- Partirem de la implementació de les taules mitjançant funcions de dispersió (classe `HashMap<T,FH,B>`)
- Els fitxers relatius ens permeten un accés directe per posició (com els arrays)
- Per tant, per aconseguir una taula persistent, substituïm l'array `v` de la implementació de `HashMap` per un fitxer relatiu `f`
- Afegirem també les operacions de vinculació d'una taula amb un fitxer físic (`open`) i de desvinculació (`close`)
- Serà convenient que les claus i els valors emmagatzemats a la taula persistent no continguin apuntadors a memòria dinàmica (en cas contrari, si no es tracta adequadament, al fitxer s'hi emmagatzemarien aquests apuntadors en lloc dels valors als quals es refereixen i es perdria la persistència)
- La resta de la implementació la mantindrem essencialment igual (hash tancat, representació dels nodes, algorismes d'inserció/eliminació/consulta...)

3.6.2 Implementació amb fitxers relatius

Idea d'implementació de la classe `HashMapP`

(Taula persistent implementada amb funcions de dispersió)

```
template <class T>
class Node{
public:
 char key[MAXC];
 T val;
 int mark;  //(FREE, OCCUPIED, DELETED)
};

template <class T, class FH, unsigned int B>
class HashMapP :public Taula<T>
{
 private:
  DirectFile<Node<T> > f;
  FH h;

  public:
 //..... The specified operations......
};
```

Canvis més significatius respecte `HashMap`:

- **En lloc de fer:**

```
template <class T>
class Node{
public:
 char*  key;
 T val;
 int mark;  //(FREE, OCCUPIED, DELETED)
};
```

farem:

```
template <class T>
class Node{
public:
 char  key[MAXC];
 T val;
 int mark;  //(FREE, OCCUPIED, DELETED)
};
```

O sigui: *Els caràcters de la clau s'emmagatzemen al node (no usem memòria dinàmica)*

- **Demanarem que la classe que instancïi `T` tampoc no tingui atributs que siguin apuntadors a objectes**

  La implementació sense tenir en compte aquestes dues consideracions és possible però força més complexa. No la considerarem en aquest curs

- **En lloc de fer:**

```
pos=h(k,i);
node=v[pos];
```

farem:

```
pos=h(k,i);
f.readDir(node,pos);
```

- **En lloc de fer:**

```
v[pos]=node;
```

farem:

```
f.write(node,pos);
```

3.6.2 Implementació amb fitxers relatius i buckets

Objectiu:

Millorem la implementació anterior introduint la idea de $buckets$ (galledes)

Idea:

- Cada posició del fitxer relatiu que representa la taula **no conté una única parella (clau, valor) sinó un conjunt de parelles (clau, valor)**
- Cada cop que accedim a una posició del fitxer obtindrem, $en\ un\ sol\ accés$, un conjunt de claus k que la funció de hash (en alguna prova $h_i(k)$) ha allotjat a aquella posició
- Aquesta tècnica pot reduir el nombre d'operacions d'E/S fetes sobre el fitxer

Mètode

- **Representació**
  - La taula ja no es representa com un fitxer relatiu de nodes (i.e., de ternes `<clau, valor, marca>`) sinó com un fitxer de $buckets$
  - A la seva vegada, cada bucket es representa com:
    - Un vector amb capacitat per a M parelles (clau, valor) que contindran les diferents parelles emmagatzemades al $bucket$
    - Un natural que indicarà el nombre de parelles presents al bucket (inicialment n'hi haurà 0 i, a mesura que n'hi anem inserint, el nombre augmentarà). Un $bucket$ no pot contenir més de M parelles.
    - Un booleà que indiqui si alguna vegada el bucket ha estat ple

- **Inserció**
  - La parella (k, valor) es col.loca al bucket que li assigna la funció de hash (o sigui, a $h(k)$) i s'augmenta el nombre de parelles presents al $bucket$
  - En el cas que el $bucket$ ja estés ple, caldrà cercar una nova ubicació per la parella (k,valor) que es vol inserir mitjançant una segona prova: $h_1(k)$
    Novament, si aquest segon $bucket$ també estés ple, caldria encara una altra prova ($h_2(k)$)...
  - Si com a conseqüència d'una inserció s'ha omplert un bucket, cal col.locar a cert el booleà que enregistra aquesta situació.

- **Consulta**

  Per trobar k, consulteu els buckets de la seqüència de proves de k $(h_0(k), h_1(k), ...)$ fins que:

  1. Es troba k o bé
  2. Es troba un bucket que no està ple i que mai no ha estat ple o bé
  3. S'han fet B proves $(h_{B-1}(k))$

  En els dos darrers casos, k no es troba a la taula.

  Notem que és necessari recordar si un bucket ha estat ple en alguna ocasió. Si ha estat així, és possible que haguéssim intentat col.locar en aquell bucket la clau k que ara cerquem i no ho haguéssim pogut fer perquè estava ple i, per tant, haguéssim hagut de continuar la seqüència de proves de k.

- **Eliminació**

  Per eliminar la clau k, cal consultar si hi és (aplicant l'algorisme de l'apartat anterior). Si es troba, pot ser substituïda per la darrera del bucket i decremetar el nombre de parelles presents al bucket.

Nota: Els buckets també poden usar-se per emmagatzemar enregistraments de mida variable

3.6.2 Implementació amb fitxers relatius i buckets

```
template <class T, class FH, unsigned int B>
class HashMapPIterator;

class IncorrectFileException{};
class IOException{};

template <class T>
class Node{
public:
 char key[MAXC]; //MAXC: longitud maxima de la clau
 T val;
};

template <class T>
class Bucket{
public:
 Node<T> pairs[M]; //M: capacitat del bucket
 int npairs;
 bool everfull;

 Bucket(){
   npairs=0;
   everfull=false;
 }
};
```

```
template <class T, class FH, unsigned int B>
class HashMapP :public Map<T>{

  DirectFile<Bucket<T> > f;
  FH h;
 public:

 //....Specified operations....
};
```