ÍNDEX

1) Introducció:...2

2) Els Grafs:...4

2.1) Definició de Graf. 4
2.2) Definició de Graf Dirigit. 5
2.3) Algunes propietats dels grafs. 6
2.4) Implementacions típiques de grafs i dígrafs. 7
2.5) Grafs Ponderats. 9

3) Especificació de la jerarquia de Grafs:............11

3.1) Una jerarquia d'Estructura de dades 11
3.2) Jerarquia de l'estructura de Grafs i Dígrafs............................13
3.3) Explicació de les classes creades........14
3.4) Especificació de les Classes.............15

4) Implementació de la jerarquia de Grafs..........18

5) Conclussions i Treballs Futurs:.......................25

5.1) STL: Comentaris..........................25
5.3) Com a Treballs Futurs....................26

Annex 1 27

Annex 2 41

6) Bibliografia: 48

1.- Introducció:

Durant l'elaboració d'aquest projecte, m'he pogut adonar de la importància de programar amb un llenguatge orientat a objectes. En aquest cas, he utilitzat el llenguatge de programació prou conegut com el C++. Tanmateix, aquest llenguatge compta amb una biblioteca estàndard d'estructures de dades molt potent, anomenada STL ("Standard Template Library").

En la ***Enginyeria Tècnica d'Informàtica de Gestió (ETIG)***, es troba una assignatura, anomenada ***Estructures de dades i algorismes (Edalg)***, la qual es direcciona cap aquesta vessant.

En particular, es presenta una jerarquia de Contenidors, els quals faciliten l'emmagatzemament de dades i la seva posterior manipulació. Aquestes classes anomenades ***contenidors*** fan referència a diferent tipologia d'estructuració de les dades. Aquestes poden arribar a emmagatzemar gran quantitat d'informació i d'enllaços entre elles d'una forma clara, precisa i senzilla. Aquestes classes són conegudes en el món de la programació,com; *Llistes, Piles, Cues, Maps, Dobles Llistes enllaçades, Multimaps, entre altres.*

Per aquesta raó, la nostra proposta es direccionava en la formació d'una jerarquia estructurada de Grafs i Grafs Dirigits(Digrafs). Per a poder fer aquest projecte hem partit de la jerarquia de classes presentada a l'assignatura d'Edalg: *Object ,Container , MyString* i a partir d'aquí hem començat a desenvolupar tota l'estructura jeràrquica de les diferents classes que componen la formació del Graf en sí. Aquestes classes, son; ***Vèrtex, Graph,GAL (List Adjacent Graph),GAM (Matrix Adjacent Graph), Digraph, DAL (List Adjacent Digraph) i DAM (Matrix Adjacent Digraph).***

La jerarquia de Grafs que hem implementat l'oferim amb la llicència GPL de codi obert(que tothom la pugui utilitzar sense cap tipus de problema).

Per a fer la codificació s'ha utilitzat l'editor de Linux "***Kate***" i el compilador "***g++***". S'han utilitzat les presentades a l'assignatura d'Edalg: ***Object, Container, MyString*** i les ja conegudes del ***STL: List, Iterator,Map,etc...***

La memòria consta de cinc capítols:

El ***primer capítol***, és una breu introducció a la documentació,a on es destaca -per damunt- les característiques més senyalades d'aquest projecte.

En ***segon capítol***, tindríem la part teòrica del tema que avarca el treball de final de carrera: Els Grafs dirigits, els no dirigits, quines propietats tenen, etc.

El ***tercer capítol***, es mostra la jerarquia original (la de l'assignatura d'Edalg),i la jerarquia creada per nosaltres.

En el ***quart capítol***, ens trobaríem amb tot el codi en C++ que s'ha utilitzat per a implementat amb codi l'estructura mostrada en l'anterior apartat.

Per últim, tindríem les ***conclusions*** obtingudes a partir de la biblioteca STL, la codificació de la jerarquia en sí i de tot en conjunt ; l'estructuració en paper, obtenir informació de la STL, utilitzar el compilador del Linux (g++),etc.

2.- Els Grafs:

Aquest capítol conté un resum dels aspectes més introductoris de la Teoria de Grafs. Podem trobar molta més informació sobre aquest tema a [2].

2.1.- Definició de Graf.

Un graf G = (V,A) està format per un conjunt finit i no buit V i per un conjunt A de parells no ordenats d'elements diferents de v.
Als elements de V els anomenarem vèrtexs i als elements de A arestes. Si a = {u,v} és una aresta,direm que u i v són vèrtexs adjacents, escrivint u ~ v, i també direm que l'aresta a és incident amb els vèrtexs u i v. Normalment, per abreujar la notació escriurem a = uv en lloc de a = {u,v}.

L'***ordre d'un graf*** G = (V,A) és el nombre de vèrtexs de G, és a dir, el cardinal de V que [2]denotarem per |V|; la mida de G és el nombre d'arestes de G, és a dir, |A|.

Exemple: Donat el graf G = (V,A), on

V = {a,b,c,d,e,f} i A = {{a,b},{a,c},{b,c},{b,d},{b,e},{c,d},{d,e}},

L'***ordre*** de G és 6 i la seva ***mida*** és 7. Pel que fa a les adjacències observeu que, per exemple, a i b són adjacents, però en canvi a i d no ho són. Aquestes relacions es veuen immediatament si es representa el graf mitjançant un dibuix on els vèrtexs són punts del pla i les arestes, línies que uneixen vèrtexs adjacents. Així, el graf G donat anteriorment pot representar-se mitjançant el dibuix de la ***figura 2.1***.

Quan a la representació gràfica d'un graf cal assenyalar, d'una banda, la seva arbitrarietat i, de l'altra, el caràcter intuïtiu d'aquesta que ens serà de gran ajuda a l'hora de comprendre conceptes i arguments abstractes relatius als grafs.
També hem de dir que, dissortadament, no hi ha una terminologia unificada en la literatura sobre teoria de grafs.

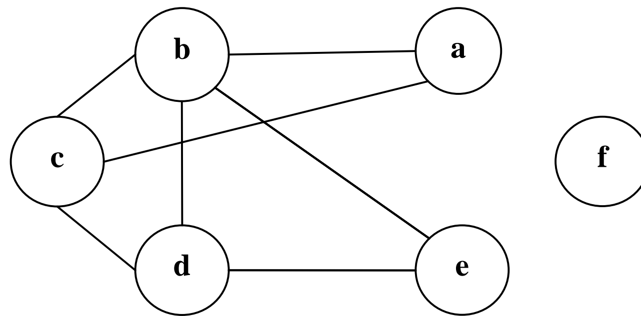


Figura 2.1: Representació gràfica d'un graf.

2.2.- Definició de Graf Dirigit.

El concepte de dígraf o graf dirigit deriva directament del concepte de graf exigint que les arestes, que ara en direm arcs, siguin parells ordenats de vèrtexs diferents.

Un dígraf G = (V,A) està format per un conjunt finit i no buit V i per un conjunt A de parells ordenats d'elements diferents de V.

Els elements de V s'anomenen ***vèrtexs*** i els de A s'anomenen ***arcs***. Si a = (u,v) és un arc direm que u és adjacent cap a v i que v és adjacent des de u. També direm que l'arc a = (u,v) és incident desde u cap a v. Normalment, per abreujar la notació escriurem a= uv en lloc de a = (u,v).

Els conceptes d'***ordre*** i ***mida*** d'un dígraf són anàlegs als d'un graf,és a dir, l'ordre del dígraf G = (V,A) és el nombre de vèrtexs de V i la mida de G és el nombre d'arcs de G.

Exemple: El dígraf G = (V,A), on V = {a,b,c} i A = {(a,b),(a,c),(b,a),(c,b)}, és un dígraf d'ordre 3 i mida 4.

Una altra manera de representar un dígraf és mitjançant un dibuix on els vèrtexs es representen per punts del pla i on cada arc a = uv es representa per una línia que va des del punt associat a u al punt corresponent a v i indicarem la direcció per mitja d'una punta de fletxa. Així, el dígraf de l'exemple anterior pot representar-se mitjançant el dibuix de la ***figura 2.2***.

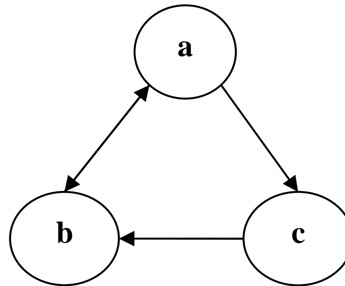


Figura 2.2: Representació gràfica d'un digraf.

2.3.- Algunes propietats dels grafs.

El ***Grau d'un vèrte***x v en un graf G = (V,A), que denotarem per g(v),és el nombre d'arestes de G incidents amb v.

Als vèrtexs de Grau 0 els anomenarem ***vèrtexs aïllats*** i als de grau 1 ***vèrtexs terminals***.

Direm que dos ***grafs*** G = (V,a) i G' = (V',A') són ***isomorfs***, i ho denotarem per $G \cong G'$,

Si existeix una aplicació bijectiva f de V en V' tal que preserva les adjacències; és a dir,

$$\forall u,v \in V,\quad u \sim v \longleftrightarrow \int(u) \sim \int(v)$$

El ***graf nul*** d'ordre n,que denotarem per Nn,és un graf que té n vèrtexs i no té cap aresta, és a dir, Nn = (V,A) on |V| = n i on A =0.

El ***graf complet*** d'ordre n,que denotarem per Kn, és un graf amb n vèrtexs, on cada vèrtex és adjacent a tots els restants.

Un ***graf*** G = (V,A) és ***d-regular*** si tots els vèrtexs de G tenen Grau d.

Un ***graf*** G = (V,A) és ***bipartit*** si el conjunt V de vèrtexs admet una partició en dos subconjunts V1 i V2 tals que qualsevol aresta de G uneix un vèrtex de V1 amb un vèrtex de V2,és a dir,

Si uv $\in$ A, aleshores (u $\in$ V1 i v $\in$ V2) o (v $\in$ V1 i u $\in$ V2).

Un graf H = (V',A') és un ***subgraf*** de G = (V,A) si $V' \subseteq V$ i $A' \subseteq A$. Quan V' = V direm que H és un subgraf generador de G.

Si G = (V,A) és un graf i S és un subconjunt de V, el ***subgraf induït*** per S, que denotarem per <S> , és el graf < S > = (S,A'), on els elements de A' són les arestes de G que uneixen vèrtexs de S,és a dir,

$$A' = \{ uv \in A \mid u \in S \text{ i } v \in S \}.$$

El ***graf complementari*** d'un graf G = (V,A) és el graf $\overline{G} = (V,\overline{A})$, on $\overline{A}$ = P2(V) – A és el conjunt d'arestes que no pertanyen a A; és a dir, dos vèrtexs diferents $u,v \in V$ són adjacents en $\overline{G}$ si, i només si, no ho són en G.

2.4.- Implementacions típiques de grafs i dígrafs.

Tant la representació d'un graf com un parell (V,A) de conjunts finits on els elements de A són de la forma {u,v}, sent u i v vèrtexs diferents de V, com la representació mitjançant un dibuix, no resulten adequades si volem que siguin processades per un ordinador. A continuació presentem dues representacions matricials d'un graf, que ens proporcionen la informació rellevant d'aquest i que per la seva estructura resulten molt fàcilment manipulables.

La ***Matriu d'Adjacència*** d'un graf G = (V,A), amb el conjunt de vèrtexs V = {v1,v2,…,vn},és la matriu quadrada **Ma**(G)=(a_{ij}) de tamany n x n definida de la manera següent:

$$a_{ij} = \begin{cases} 1 & \text{si } v_i \text{ i } v_j \text{ són adjacents.} \\ 0 & \text{altrament.} \end{cases}$$

La ***Matriu d'Incidència*** d'un graf G = (V,A), amb el conjunt de vèrtexs V = {v1,v2,…,vn} i amb el conjunt d'arestes A = {a1,a2,…,an},és la matriu **Mi**(G) = (bij) de tamany n x m definida de la manera següent:

$$b_{ij} = \begin{cases} 1 & \text{si } v_i \text{ és incident amb } a_j. \\ 0 & \text{altrament.} \end{cases}$$

Exemple: Donat el Graf G = (V,A), on V = {a,b,c,d} i A = {ab,ac,ad,bd} - representat en la ***figura 2.4*** -, la matriu d'adjacència i la matriu

d’incidència de G són:

$$\mathbf{Ma}(G) = \begin{pmatrix} 0111 \\ 1001 \\ 1000 \\ 1100 \end{pmatrix} \quad \text{i} \quad \mathbf{Mi}\,(G) = \begin{pmatrix} 1110 \\ 1001 \\ 0100 \\ 0011 \end{pmatrix}, \text{ respectivament.}$$

Les següents propietats, relatives a les matrius d’adjacència i d’incidència d’un graf, es dedueixen directament de les seves corresponents definicions:

1. La matriu d’adjacència d’un graf és una matriu simètrica. A més, el nombre d’uns de cadascuna de les seves files coincideix amb el grau del vèrtex corresponent.

2. El nombre d’uns de cadascuna de les files de la matriu d’incidència d’un graf és igual al grau del vèrtex corresponent, i el nombre d’uns de cadascuna de les seves columnes és 2.

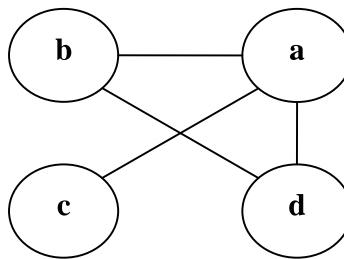


Figura 2.4

Observeu que el lema de les encaixades de mans també es pot demostrar emprant la matriu d’incidència del graf. Així, la suma de les files d’aquesta matriu, que correspon a la suma dels graus dels vèrtexs del graf, és igual a la suma de les columnes ja que totes dues sumes coincideixen amb la suma de tots els elements de la matriu, i aquesta representa el doble del nombre d’arestes del graf.

En cas que hi hagi molts parells de vèrtexs de G no adjacents entre sí, tant la matriu d’adjacència com la d’incidència resulten representacions ineficients pel gran nombre d’entrades nul·les que contenen. Aleshores cal pensar a utilitzar una estructura més eficient, des del punt de vista de memòria necessitada , com ara la llista d’adjacències.

La ***Llista d’adjacències*** d’un graf G amb conjunt de vèrtexs.

V={v1,v2,...,vn}

És una llista formada per n subllistes, una per a cada vèrtex vi, on

figuren els vèrtexs adjacents al corresponent vèrtex vi.

Exemple: Sigui el graf G = (V,A),on

V ={a,b,c,d,e} i A = {ab,ac,ad,ae,bc,de}.

La llista d'adjacències d'aquest graf es pot representar mitjançant la taula següent:

a	b	c	d	e
b	a	a	a	a
c	c	b	c	d
d		d	e	
e				

2.5.- Grafs Ponderats.

Un graf(dígraf) ponderat G = (V,a,w) està format per un graf (dígraf) G = (V,A) i per una aplicació

$$w: A \longrightarrow R_+$$
$$a \longrightarrow W(a) \text{ anomenat pes o cost de a.}$$

En molts casos, es precís atribuir a cada aresta un nombre específic, anomenat valuació, ponderació o cost segons el context i s'obté així un graf ponderat.
Normalment, és un graf amb una funció v: A → R+.

Exemple: Un representant comercial té que visitar n ciutats connectades entre sí per carreteres, el seu interés previsible serà minimitzar la distància recorreguda(o el temps, per si es poden prevenir retencions).El graf corresponents tindria com a vèrtexs les ciutats, com arestes les carreteres i la valuació (o cost) seria la distancia que hi ha entre elles. De moment, no es coneixen mètodes generals per a trobar un cicle de cost mínim, però sí per als camins desde a fins a b.

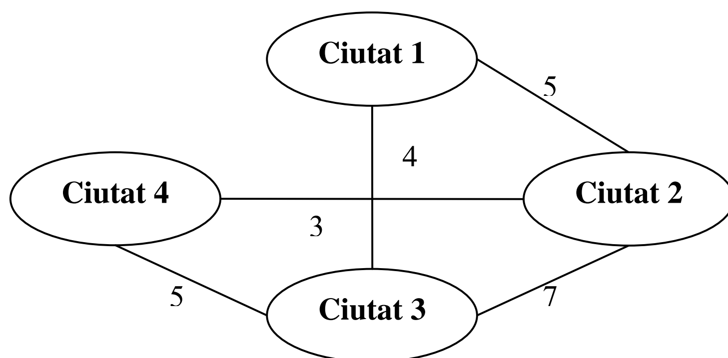


Figura 2.5: Graf Ponderat.

3.- Especificació de la jerarquia de Grafs:

3.1.- Una jerarquia d'Estructura de dades.

1ª part de la estructura.

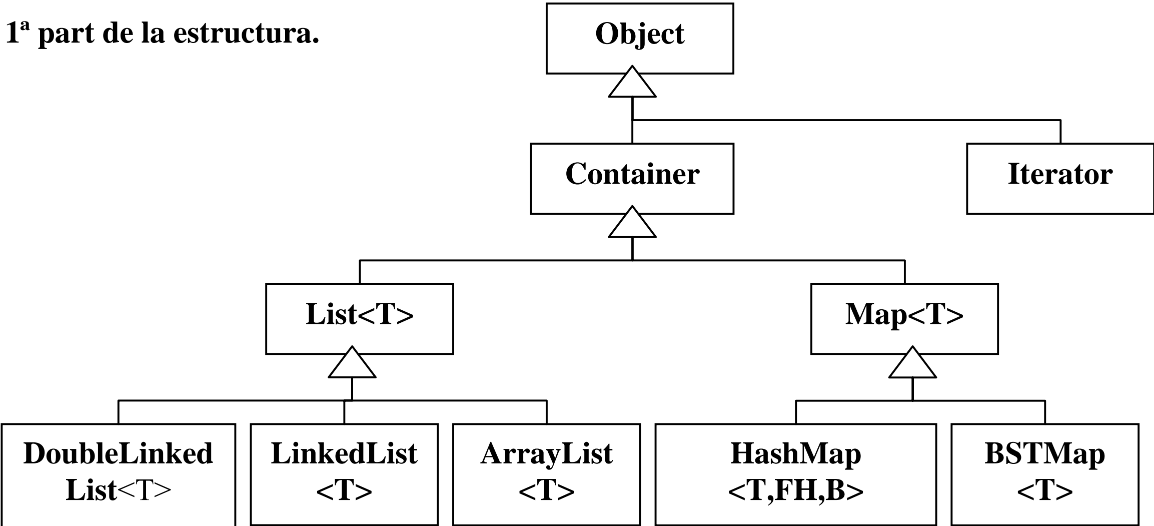


Figura 3.1

2ª part de la estructura.

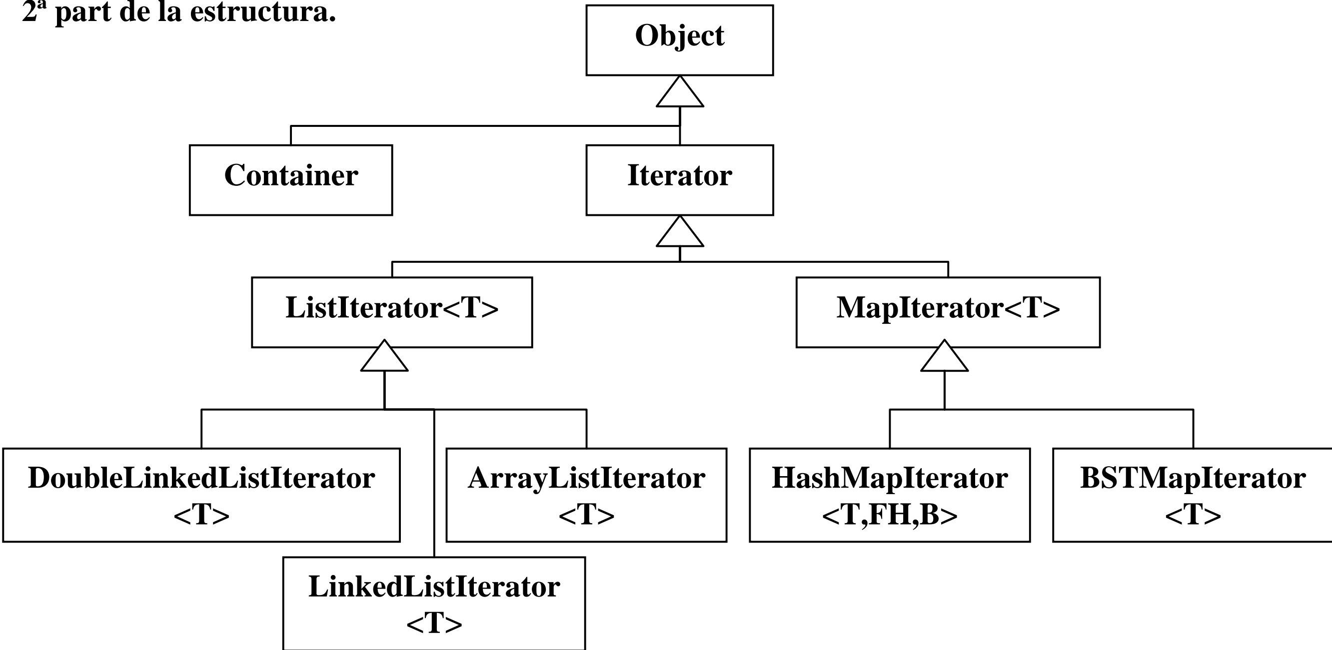


Figura 3.2

Les figures 3.1 i 3.2 mostren la jerarquia d'estructures de dades presentada a l'assignatura d'Edalg.

Com es pot apreciar, l'estructura que es segueix està basada en l'emmagatzemament de la informació. La major part de les estructures de dades depenen d'un tipus component, que instancia el tipus genèric T que apareix a les classes de les figures. Així la classe List<T>, vol dir una llista amb elements de tipus T. T es pot instanciar amb qualsevol classe de la jerarquia(ex: MyString). Les classes que depenen d'un tipus corresponent s'anomenen plantilles(Templates).

La segona part de la estructura(figura 3.2) presenta la jerarquia d'iteradors. Els iteradors permeten recórrer de manera senzilla i elegant els elements d'un contenidor. I permeten, un accés més ràpid a un element específic d'aquell contenidor.

La jerarquia consta d'una classe, anomenada Object (Objecte), la qual conté el patró genèric dels mètodes necessaris per qualsevol cosa vinculada a la creació i manipulació de qualsevol objecte. Aquesta classe farà que totes les que es posin a posteriori, heretin d'ella (sinó es directament….serà indirectament). Aquests mètodes són:

virtual void copy(const Object&)=0: Es crea un node auxil·liar I còpia còpia l'Objecte entrat com a paràmetre.

virtual Object* clone() const=0: Fà una còpia exacta de l'objecte entrat per paràmetre i el retorna.

virtual bool operator==(const Object&) const =0: Retorna si són o no són iguals l'objecte entrat per paràmetre amb l'actual.

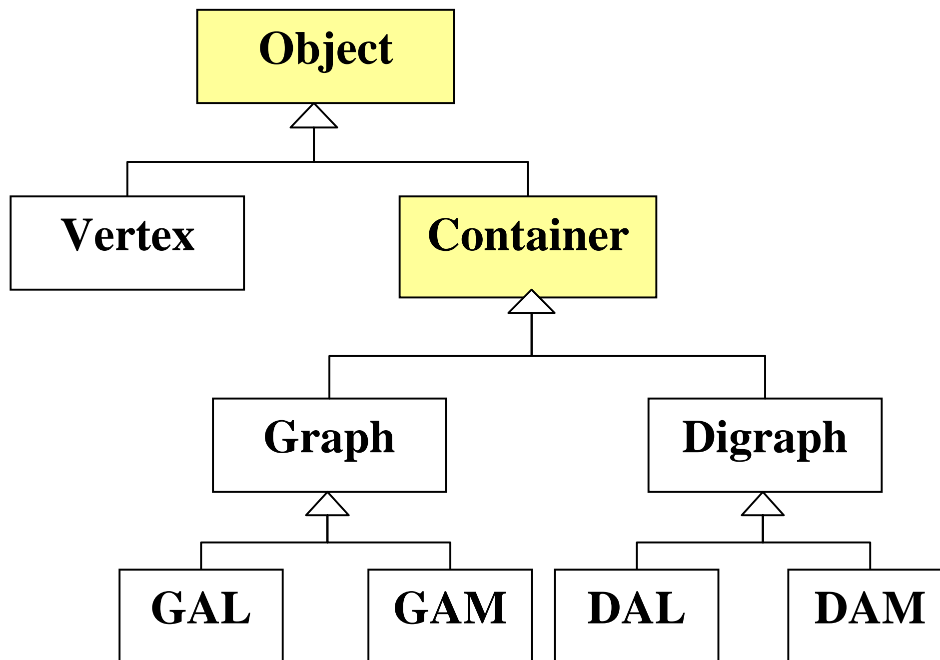
virtual MyString toString() const: Retorna tots els elements que conté l'objecte en qüestió.

Després de la classe principal, ens trobem amb una bifurcació; Container i Iterator.
L' una conté una jerarquia interna de diferents formes d' emmagatzemar l'objecte creat (la estructura de dades),aquestes serien: llistes, Mapes, Arrays, etc.
L'altra constaria d'una altra jerarquia diferent a la d'abans ja que aquesta, ens permet recórrer els elements d'una classe. Les operacions que s'associen als iteradors permeten accedir al primer o darrer element, avançar, retrocedir.

L'assignatura d'Edalg ensenya les diferents formes de classificar la informació i les respectives manipulacions que es poden fer a cada tipo de Contenidor.
Depenen de la informació a introduir-hi i a tractar, l'usuari haurà de ser conscient de quina capacitat de memòria requereix i, a partir d'aquí,haurà de triar el contenidor que s'adeqüi a les seves necessitats; rapidesa, fiabilitat, requeriment de memòria, accés a les dades,etc…

L'objectiu d'aquest treball és enriquir la jerarquia presentada amb noves classes per crear i manipular grafs. Aquestes noves classes s'integraràn a la jerarquia existent amb relacions d'herència.

3.2.- Jerarquia de l'estructura de Grafs i Dígrafs.



Com es pot apreciar a l'esquema de dalt, s'ha agafat les classes: Object, Container i MyString i, a partir d'elles, s'ha procedit a derivar-ne una nova jerarquia que aprofita l'herència amb les classes anteriors. S'ha pogut utilitzar classes i mètodes heretats de la jerarquia anterior, minimitzant la implementació de nou codi. D'aquesta manera disminuïm el temps de creació i implementació del mateix projecte.

3.3.- Explicació de les classes creades.

Graph: Aquesta és una classe Abstracta que modelitza els objectes Graf. La seva funció és donar una petita estructura inicial a les classes que heretaran d'ella. Aquesta estructura contindrà les especificacions dels mètodes que hauran d'implementar les classes que heretin ja que aquests mètodes són virtuals purs.

GAL: És una classe que hereta de Graph. El seu objectiu és oferir una implementació dels grafs en forma de llista d'adjacències.

GAM: És una classe que hereta de Graph. El seu objectiu és oferir una implementació dels grafs en forma d'una matriu d'adjacències. Aquesta classe conté la forma de crear-la per a la pròpia estructura de dades del Graph.

Digraph: Aquesta és una classe Abstracta que modelitza els objectes grafs dirigits. La seva funció és donar una petita estructura inicial a les classes que heretaran d'ella. Aquesta estructura contindrà les especificacions dels mètodes que hauran d'implementar les classes que heretin ja que aquests mètodes són virtuals purs.

DAL: És una classe que hereta de Digraph. El seu objectiu és oferir una implementació dels grafs dirigits en forma de llista d'adjacències.

DAM: És una classe que hereta de Digraph. Aquesta conté la forma de crear una Matriu d'Adjacències per a la pròpia estructura de dades del Digraph.

3.4.- Especificació de les Classes.

Graph: Aquesta classe encapçalaria els mètodes a emprar de les dues subclasses de més avall: GAL i GAM. També, és la que dona l'Herència a les altres dues de tots els seus mètodes(la majoria virtuals pures). Aquesta és la única manera de poder tindre estructurada la jerarquia(o sub-jerarquia...,si parlem de tot el conjunt...),les classes les quals hereten i la posterior implementació dels mètodes heretats.

- **Graph():**Constructor per defecte (sense pas de paràmetres inicials). Crea un graf buit(sense vèrtexs ni arestes).
- **virtual void copy(const Object&)=0:** És un mètode de la classe Virtual Pura. Això vol dir que, totes les classes que heretin d'aquesta podran utilitzar-la i implementar-la com millor els i vagi.
- **virtual void addVertex(T&)=0:** Afegeix un node al graf.
- **virtual void addEdge(MyString v1,MyString v2)=0:** Afegeix una aresta entre els vèrtexs identificats de ***v1*** i ***v2***.
  Si algun dels vèrtexs v1,v2 no pertany al graf es produeix un error. Si l'aresta v1v2 ja pertany al graf, no s'afegeix novament.
- **virtual list<MyString> GetAdjacents(MyString v1)=0:** A partir d'un vèrtex entrat, retorna la seva llista d'adjacències.
- **virtual T GetVertex(MyString id)=0:** Retorna totes les dades referents al identificador del node entrat.
- **bool isRelated(MyString&, MyString&):** Entrat dos vèrtex, aquest mètode retorna si estàn connectats o no.

T es pot instanciar per qualsevol classe que ofereixi l'operació MyString getkey().
Per tal d'aconseguir aquest comportament derivaran qualsevol classe que hagi d'actuar com a vèrtex d'un graf/dígraf de la classe abstracta Vèrtex definida de la manera segënt:

virtual MyString getkey()=0;

GAL: Aquesta classe proposa una implementació dels grafs en forma de llista d'adjacències(***GAL: "Graph Adjacents List"***).Per a poder donar-nos aquesta informació, en originar-se es crea un map del tipus ***NodeGAL<T>***,el qual farà que cada casella del map contingui l'estructura especificada per NodeGAL(un referenciador de T i una llista d'Iteradors a map,aquesta del mateix tipus de dades, NodeGAL).
També, contindrà dos comptadors; l'un per a comptar el nombre de vèrtexs, i l'altre per a comptar el nombre d'adjacències que conté el Graf. Vegeu el capítol 4.

Operacions:

- **GAL():** Constructor. Crea un GAL buit sense v ni m.

- **void visualitzacio():** Mostra tot el graf: els vèrtexs i cadascuna de les adjacències d'aquests.

- **MyString Tot_GAL():** Mostra tot el graf.

La resta d'operacions són heretades i ja han estat especificades anteriorment.

Digraph: Aquesta classe modelitza els grafs dirigits(vegeu capítol 2) i defineix els mètodes a emprar de les dues subclasses de més avall: DAL i DAM.

S'ha de tindre en compte que,els ***Dígrafs són igualment Grafs però*** aquests, ***tenen la propietat en que les seves adjacències van dirigides***. És a dir, pot ser que es vagi de ***a*** cap a ***b***,i que no es vagi de ***b*** cap a ***a*** (cosa que en els Grafs unidireccionals,això no pot passar mai perquè la connexió es mútua).
A part d'això, les estructures i els mètodes són els mateixos. L'únic que s'ha tingut en compte es la direccionalitat de les adjacències, rés més:

- **Digraph():**Constructor per defecte (sense pas de paràmetres inicials). Crea un graf buit(sense vèrtexs ni arestes).

- **virtual void copy(const Object&)=0:** És un mètode de la classe Virtual Pura. Això vol dir que, totes les classes que heretin d'aquesta podran utilitzar-la i implementar-la com millor els i vagi.

- **virtual void addVertex(T&)=0:** Afegeix un node al Dígraf.

- **virtual void addEdge(MyString v1,MyString v2)=0:** Afegeix una aresta entre els vèrtexs identificats de ***v1*** i ***v2***.
  Si algun dels vèrtexs v1,v2 no pertany al dígraf es produeix un error. Si l'aresta v1v2 ja pertany al graf, no s'afegeix novament.

- **virtual list<MyString> GetAdjacents(MyString v1)=0:** A partir d'un vèrtex entrat, retorna la seva llista d'adjacències.

- **virtual T GetVertex(MyString id)=0:** Retorna totes les dades referents al identificador del node entrat.

- **bool isRelated(MyString&, MyString&):** Entrat dos vèrtex, aquest mètode retorna si estàn connectats o no.

Les classes que instancïin T hauran de ser derivades de Vèrtex (vegeu l'especificació de Graf).

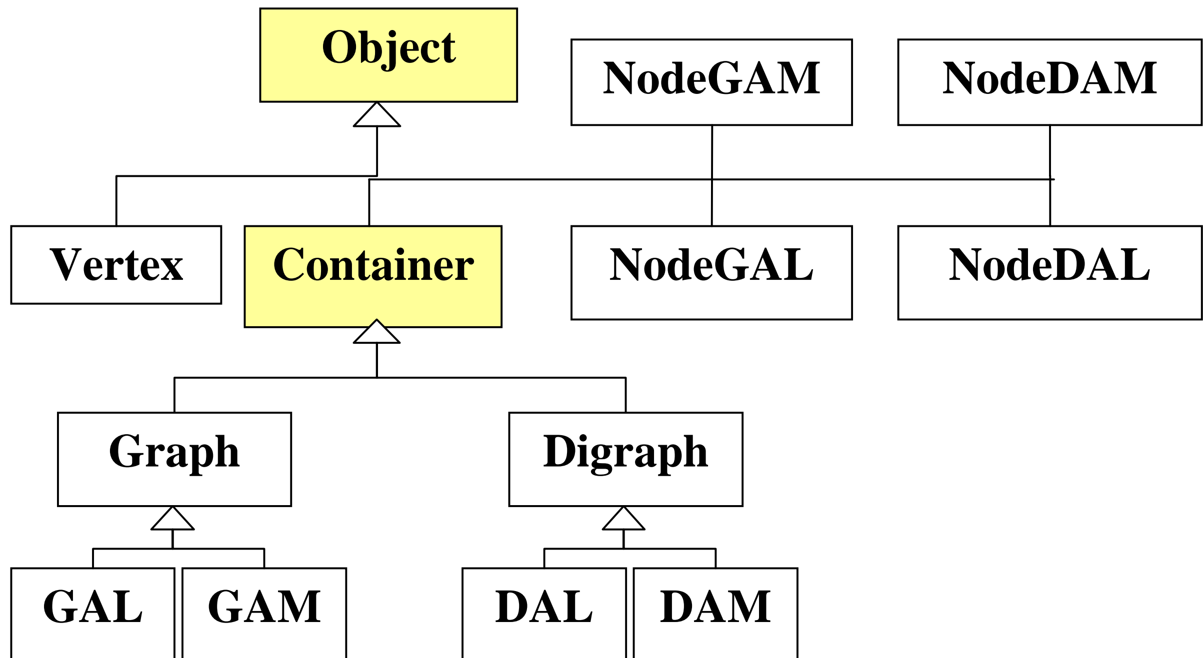
DAL: Aquesta classe ens dona la Llista d'Adjacències del propi Graph(***DAL: "Digraph Adjacents List"***).Per a poder donar-nos aquesta informació, al originar-se es crea un map del tipus ***NodeDAL<T>***,el qual farà que cada casella del map contingui l'estructura especificada per NodeDAL(un referenciador de T i una llista d'Iteradors a map,aquesta del mateix tipus de dades, NodeDAL).
També, contindrà dos comptadors; l'un per a comptar el nombre de vèrtexs, i l'altre per a comptar el nombre d'adjacències que conté el Dígraf.

- **DAL():** Constructor. Crea un GAL buit sense v ni m.

- **void visualitzacio():** Mostra tot el dígraf: els vèrtexs i cadascuna de les adjacències d'aquests.

- **MyString Tot_DAL():** Mostra tot el graf.

4.- Implementació de la jerarquia de Grafs.



GAL:

Part Privada:

```
map<MyString,NodeGAL<T>,MyStringCompare > g;
int nvertex;
int nedges;
```

Nodegal:

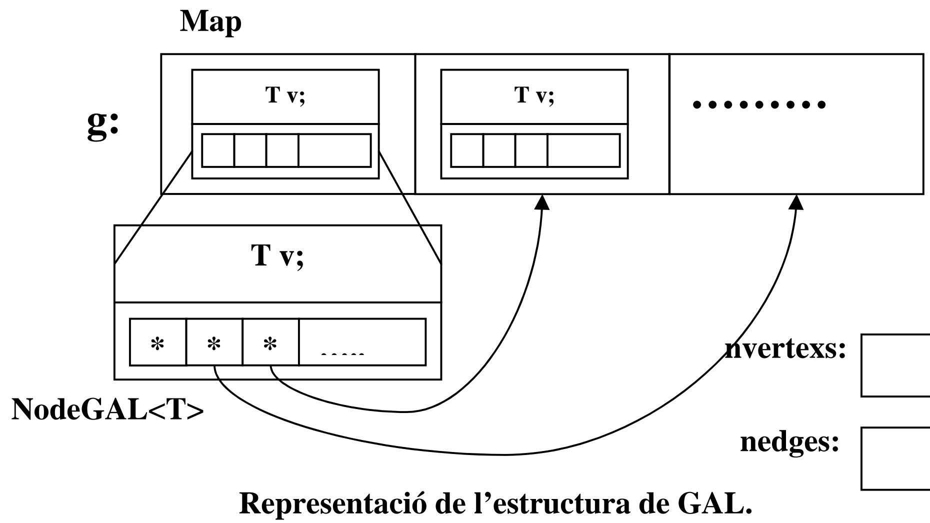
```
T v;
list<class map<MyString,NodeGAL<T>,MyStringCompare>::iterator> ladj;
```

Explicació: La classe conté,en la seva creació, un contenidor del tipus map i dos variables del tipus enter.
Alhora de crear-se el contenidor,s'ha de tindre en compte que es crearà del tipus de la Classe NodeGAL<T>.És a dir,cada casella que tindrà el map,estarà dissenyada per emmagatzemar dades del tipus de NodeGAL<T>.

La classe NodeGAL<T> contindrà un referenciador a el node creat amb una plantilla específica. D'aquesta manera,dins de la seva pròpia estructura, podrà accedir a qualsevol informació del mateix node.
A part del referenciador,la classe compta amb una llista d'iteradors,aquests del mateix tipus que les caselles del propi map.Aquests, seran els mateixos que s'enllacin amb els identificadors del mateix map creat, en aquest cas, per GAL. La llista només contindrà les adreces de memòria en que es troben els identificadors del mateix

map,solsament contindrà aquells que siguin adjacents al propi node.

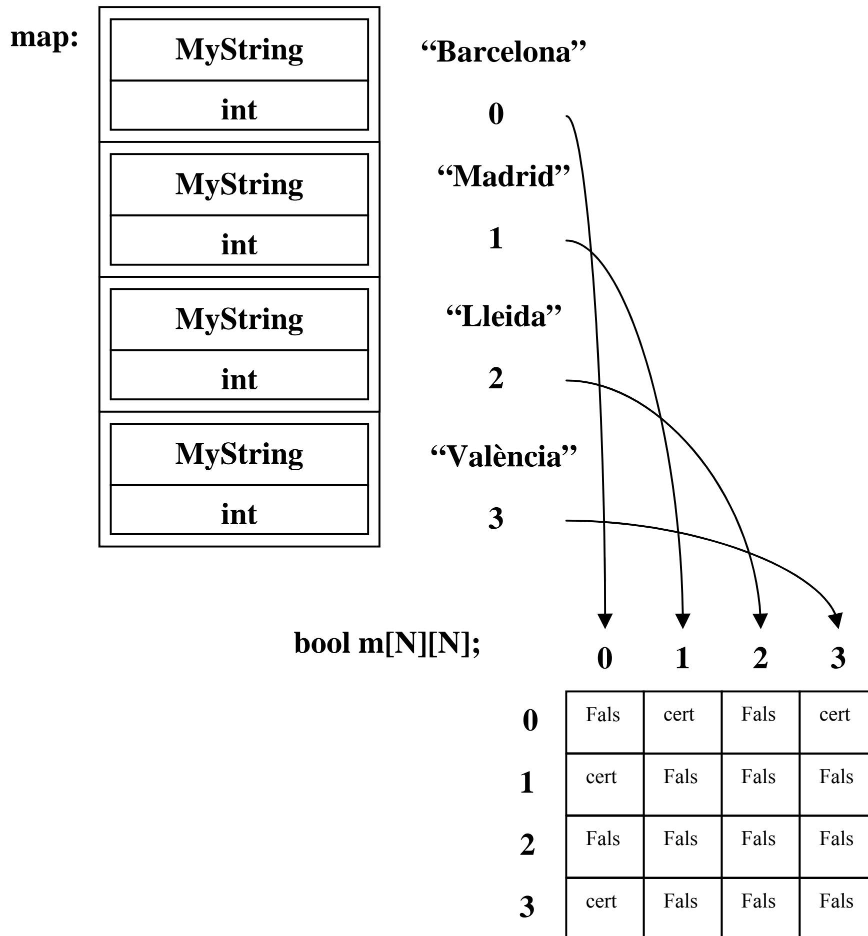
El contenidor tindrà la funció d'emmagatzemar totes les dades referents al Graf, com; els vèrtexs i les adjacències.Les dos variables del tipus enters,només serviran per anar comptant –en temps d'execussió- quants vèrtexs i adjacències es troben dintre del Graf.



Representació de l'estructura de GAL.

GAM:

1a. Opció:



Creació de la Matriu amb l’ajuda d’un map.

En primer lloc, s’hauria de crear una matriu Bidimensional de tipus booleà(cert/fals). Aquesta serà la que guardarà l’informació de les adjacències dels respectius nodes.

Alhora, s’hauria de crear un map per a relacionar l’identificador de cada vèrtex amb el natural amb el qual s’accedirà a la matriu.

La matriu consta de files i columnes. Les dos fan referència –en aquest exemple- a les 4 ciutats: Barcelona, Madrid, Lleida i València.

Cada ciutat està representada dintre de la matriu per un número enter. Quan es dongui una adjacència entre dues ciutats, s'haurà d'anar al map, obtindre, a partir de l'identificador, el número de fila i columna de la matriu. Un cop obtinguda aquesta informació, s'haurà de recorrer la matriu per arrivar a la casella corresponent(la que relaciona les dues ciutats),i posar-li com a valor "CERT"(true). D'aquesta manera l'Adjacència entre les dues ciutats estarà creada.

La informació que guarda la matriu és booleana, pertant, cada casella que tingui com a valor "cert" representarà que hi hà una adjacència entre les posicions de la respectiva casella(m[x][y]), 'x' faria referència a una ciutat i la 'y' a l'altra ciutat relacionada.

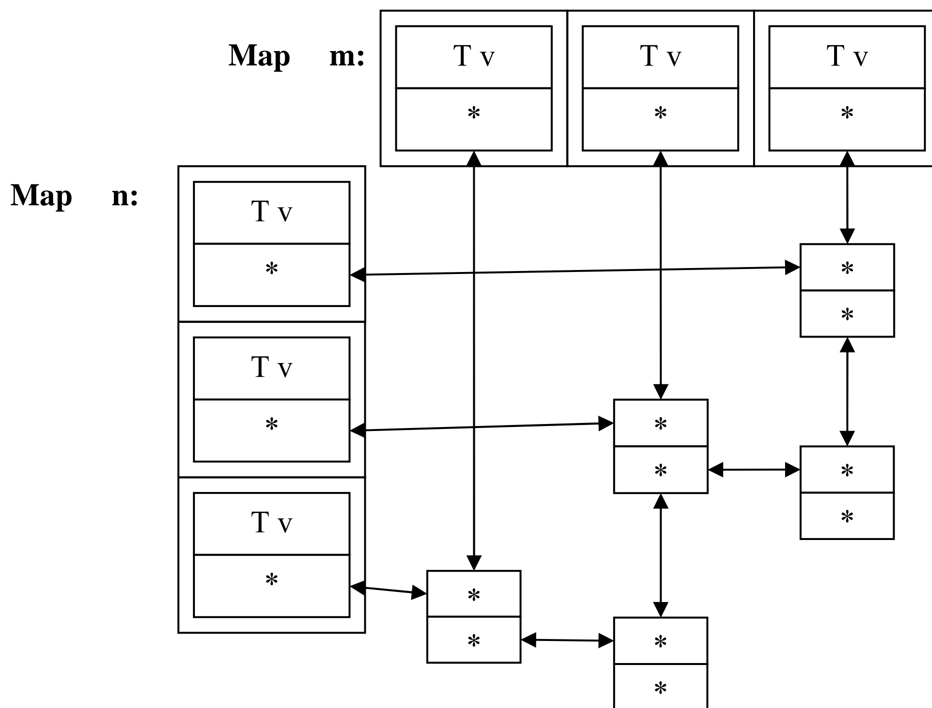
Ventatjes:

- Simplicitat d'implemetació.

Desventatjes:

- Molta despesa d'esoai si és un graf dispers.

2a. Opció:



Creació de la Matriu amb Mulillista.

Aquesta opció requereix la creació de dos maps, aquests d'idèntica estructura. Aquesta estructura, haurà de tindre: un referenciador al tipus T i un punter a una llista doblement enllaçada.

S’hauràn de crear tantes multillistes com caselles tingui el map(només s’haurà de tindre en compte un sol map, mai els dos), ja que cada casella conté una llista d’adjacències pròpia.

També s’haurà de crear una estructura independent(node), la qual estarà formada per dos apuntadors. Aquests apuntadors seràn del mateix tipus que la llista doble i construiràn els nodes de la llista. El primer la recorrerà horitzontalment(per fila),i el segón verticalment(per columna).

Aleshores, quan es produeixi una adjacència, s’haurà de crear un node, format per només dos apuntadors, els quals ún serà per fer l’enllaç cap a la llista horitzontal i l’altre farà el mateix però en la llista vertical(com es pot veure a la figura de dalt).

Ventatjes:

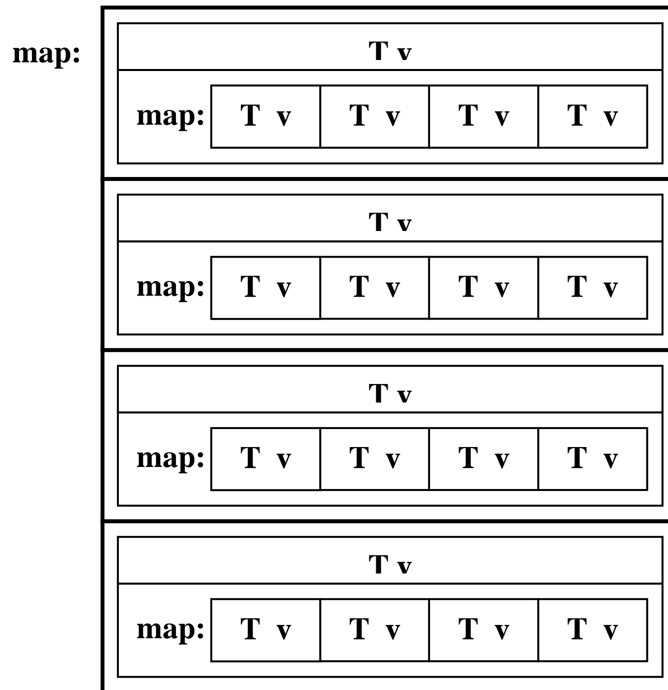
- Resol el problema de l’espai en grafs dispersos.

Desventatjes:

- S’introdueix una ineficiència temporal que correspon a tota la columna amb la que es refereix cada node de cada llista corresponent a cada filera(o a la inversa).

Això es pot resoldre fent que cada node de la multillista referencïi la filera i la columna d’on procedeix.
Però aquesta solució degrada l’eficiència espaial.

3a. Opció:



Creació d'una Matriu de maps.

En aquesta opció, crearíem un map que, en cadascuna de les seves caselles, en contingués un altre.
El primer faria referència a la filera i el segón emmagatzemarà l'informació de totes les columnes que tinguin alguna adjacència amb éll.

Cal dir que, l'utilització d'aquest map seria de la mateixa forma que si estiguéssim utilitzant una matriu bidimensional.

Per un costat tindríem:

```
Map< MyString,map<MyString>,MyStringCompare> map1;
```

...per l'altre costat, tindríem:

```
Map<MyString,T,MyStringCompare> map2;
```

D'aquesta manera, m[f][c] de l'opció 1 es convertiria en:

```
g.Map1[id].map2[id]
```

Ventatjes:

- Utilització simple i eficient temporalment (comparable a l'opció 1).

Desventatjes:

- Depenent de quina sigui la implementació del map no es resol el problema de la ineficiència espaial(sobretot en grafs dispersos).

DAL:

Part Privada:

```
map<MyString,NodeDAL<T>,MyStringCompare > g;
int nvertex;
int nedges;
```

NodeDAL:

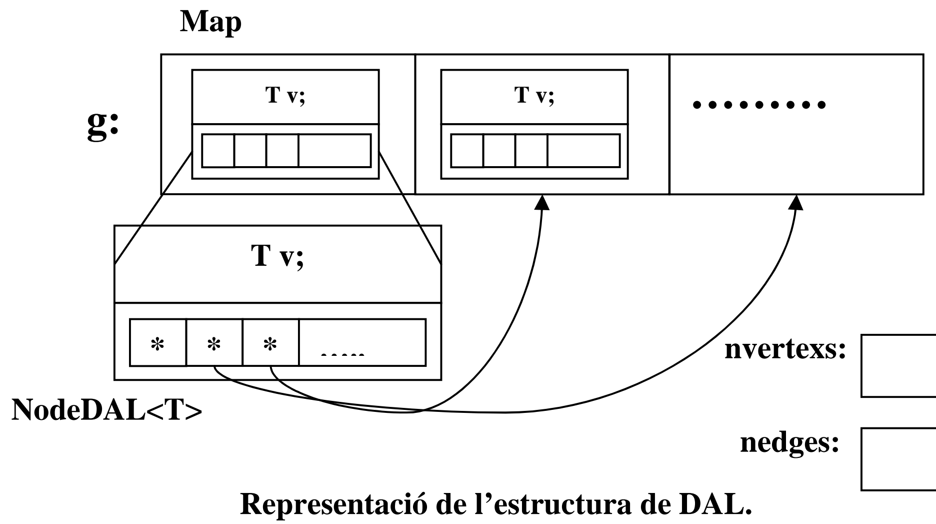
```
T v;
list<class map<MyString,NodeDAL<T>,MyStringCompare>::iterator> ladj;
```

Explicació:

La classe conté,en la seva creació, un contenidor del tipus map i dos variables del tipus enter.
Alhora de crear-se el contenidor,s’ha de tindre en compte que es crearà del tipus de la Classe NodeDAL<T>.És a dir,cada casella que tindrà el map,estarà dissenyada per emmagatzemar dades del tipus de NodeDAL<T>.

La classe NodeDAL contindrà un referenciador a el node creat amb una plantilla específica. D’aquesta manera,dins de la seva pròpia estructura, podrà accedir a qualsevol informació del mateix node.
A part del referenciador,la classe compta amb una llista d’iteradors,aquests del mateix tipus que les caselles del propi map.Aquests, seran els mateixos que s’enllacin amb els identificadors del mateix map creat, en aquest cas, per DAL. La llista només contindrà les adreces de memòria en que es troben els identificadors del mateix map,solsament contindrà aquells que siguin adjacents al propi node.
També em de tindre en compte que,aquesta classe serveix per a facilitar d’informació del tipus de dades que emmagatzemarà cada casella del map(de la classe DAL).És a dir,cada cassella que contingui el map, estarà format per l’estructura de NodeDAL<T>.

El contenidor tindrà la funció d’emmagatzemar totes les dades referents al Dígraf, com; els vèrtexs i les seves respectives adjacències(s’ha de tindre en compte que aquest tipus de Graf és dirigit).Les dos variables del tipus enters,només serviran per anar comptant –en temps d’execució- quants vèrtexs i adjacències es troben dintre del Dígraf.



Representació de l'estructura de DAL.

Aquestes classes són la unitat mínima d'informació en que es compon l'estructura de dades. Són les que faciliten el tipus de format de dades en que s'haurà de regir el/s contenidor/s assignat/s a l'estructura de dades.D'aquestes classes en tenim una per cadascuna de les maneres que tenim de crear el graf. Aquestes,son:

5.- Conclussions i Treballs Futurs.

Durant el temps que ha durat la realització d'aquest projecte, s'ha anat seguint una sèrie d'objectius:
La primera cosa que es va tindre que fer va ser l'estructura en paper de totes les classes que tenien que formar part de l'estructura a crear. Un cop varem tindre l'estructura, s'havia de saber amb quins tipus de contenidors podríem comptar per l'el·laboració de la mateixa.
Tanmateix,la biblioteca estandarditzada(STL) no era massa coneguda per mi,aleshores vaig tindre que aprofundir en d'aquest aspecte.
Poder-me informar i enterar-me'n de tot lo que podia utilitzar em va portar bastant temps però, un cop assolit els coneixements, va ser molt ràpid fer l'implementació del codi.

STL: Comentaris:

La Biblioteca Estàndard de Plantilles (Standard Template Library) coneguda també per les sigles STL, és un conjunt de tipus de dades abstractes, funcions y algoritmes dissenyats per a utilitzar diferents tipus de dades específiques per a una aplicació determinada. Cadascun dels tipus de dades abstractes conté funcions útils, com també, la sobrecàrrega d'operadors.L'esperit de la Biblioteca Estàndard de Plantilles és la idea de programació genèrica, l'implementació d'algoritmes o estructures de dades sense ser depenent del tipus de dades que serà manipulat.

La STL preveu algunes característiques com la utilització automàtica de memòria, també val a dir que és més segur ja que s'evita el desbordament del buffer quan s'utilitzen vectors o estructures de dades similars.

La STL ofereix al programador diverses avantatges a més de la gestió de memòria i de la seva seguretat.

Primer, posseeix diversos tipus predefinits per a poder simplificar el disseny del programa; el programador ja no ha d'escriure la seva pròpia classe per a utilitzar els diferents contenidors(arrays, vectors,llistes,map,etc.).

Segon, s'ofereix potència al tenir algoritmes independents del tipus, englobant òbviament els algoritmes d'ordenació i la seva cerca, els que s'utilitzen en la majoria de les aplicacions.

Una altra característica important que té la STL, és que el rendiment de les aplicacions creades per un mateix, no es veuen massa afectades al utilitzar aquesta biblioteca, ja que gasta pocs recursos. A més a més és convenient utilitzar un contenidor d'aquesta classe enlloc d'utilitzar una creada per nosaltres mateixos,perquè el nostre propi codi mai estarà tant optimitzat com les classes d'aquesta biblioteca (s'ha de tenir en compte l'evolució, el tractament i la depuració que ha tingut aquesta biblioteca al llarg del temps.).

Com a Treballs Futurs:

- Implementar una jerarquia d'Iteradors.
- Implementar la jerarquia dels Grafs,amb Grafs Ponderats.
- Implementar els Algoritmes sobre Grafs; DFS ,BFS ,Dijkstra ,Kruskal ,Huffman ,etc.
- Implementar un editor, de dibuix, per a l'estructura de Grafs.

Annex 1: CODI DE LA BIBLIOTECA

```
#ifndef Vertex_H
#define Vertex_H

#include "Object.h"
#include "MyString.h"

class Vertex:public Object
	{
	public:
		virtual MyString getkey()=0;
	};

#endif
//-------------------------------------
#ifndef NodeGAL_H
#define NodeGAL_H

#include <iostream>
#include <iterator>
#include <list>
#include <map>

#include "MyString.h"
#include "Object.h"


template <class T>
class NodeGAL : public Object
{
public:
	T v;
	list<class map<MyString,NodeGAL<T>,MyStringCompare>::iterator >
ladj;

	NodeGAL(){};
	NodeGAL(T& vr):v(vr){};
	void copy(const Object& o);
	bool operator==(const Object& o) const;
	MyString toString(MyString v1,Object& o);
	Object* clone() const;
};

template<class T>
void NodeGAL<T>::copy(const Object& o)
	{
	const NodeGAL& c=dynamic_cast<const NodeGAL<T>&>(o);
	v.copy(c.v);
	};

template<class T>
bool NodeGAL<T>::operator==(const Object& o) const
	{
		const NodeGAL<T>& c=dynamic_cast<const NodeGAL<T>&>(o);
```

```
            return(ladj==c.ladj && v==c.v);
      };

template<class T>
MyString NodeGAL<T>::toString(MyString v1,Object& o)
      {
      MyString aux("NODEGAL: ");
      aux = aux + v1;
      aux = aux + " :{";

class  map< MyString,NodeGAL<T>,MyStringCompare>::iterator i =this-
>g.find(v1);
class  list< class map< MyString,NodeGAL<T>,MyStringCompare>::iterator>
*pladj;

pladj = this->g[v1].ladj;

for(i=pladj->begin(); i!=pladj.end(); i++)
      {
      aux = aux + "(";
      aux = aux + MyString((*i).first);
      aux = aux + "),";
      };

      aux = aux + "}\n";
      return aux;
      };

template<class T>
Object* NodeGAL<T>::clone() const
      {
            NodeGAL<T>* aux;
            aux=new NodeGAL<T>;
            aux->copy(*this);
            return aux;
      };
#endif
//------------------------------------
#ifndef NodeDAL_H
#define NodeDAL_H

#include <iostream>
#include <iterator>
#include <list>
#include <map>

#include "MyString.h"
#include "Object.h"


template <class T>
class NodeDAL : public Object
{
public:
      T v;
      list<class map<MyString,NodeDAL<T>,MyStringCompare>::iterator >
ladj;
```

```
	NodeDAL(){};
	NodeDAL(T& vr):v(vr){};
	void copy(const Object& o);
	bool operator==(const Object& o) const;
	MyString toString(MyString v1,Object& o);
	Object* clone() const;
};

template<class T>
void NodeDAL<T>::copy(const Object& o)
	{
	const NodeDAL& c=dynamic_cast<const NodeDAL<T>&>(o);
	v.copy(c.v);
	};

template<class T>
bool NodeDAL<T>::operator==(const Object& o) const
	{
		const NodeDAL<T>& c=dynamic_cast<const NodeDAL<T>&>(o);
		return(ladj==c.ladj && v==c.v);
	};


template<class T>
MyString NodeDAL<T>::toString(MyString v1,Object& o)
	{
	MyString aux("NODEGAL: ");
	aux = aux + v1;
	aux = aux + " :{";

class  map< MyString,NodeDAL<T>,MyStringCompare>::iterator i =this-
>g.find(v1);
class  list< class map< MyString,NodeDAL<T>,MyStringCompare>::iterator>
*pladj;

	pladj = this->g[v1].ladj;

	for(i=pladj->begin(); i!=pladj.end(); i++)
		{
		aux = aux + "(";
		aux = aux + MyString((*i).first);
		aux = aux + "),";
		};

	aux = aux + "}\n";
	return aux;
	};

template<class T>
Object* NodeDAL<T>::clone() const
	{
		NodeDAL<T>* aux;
		aux=new NodeDAL<T>;
```

```
            aux->copy(*this);
            return aux;
      };
#endif
//-----------------------------------------
#ifndef Graph_H
#define Graph_H

#include <cstdlib>
#include <iostream>
#include "Object.h"
#include "Container.h"
#include "MyString.h"

using namespace std;

template<class T>
class Graph:public Container
      {
      public:
            Graph();
            virtual void copy(const Object&)=0;
            virtual Object* clone() const=0;
            virtual void addNode(T&)=0;
            virtual bool operator==(const Object&) const=0;
            virtual void addEdge(MyString,MyString)=0;
            MyString toString() const;
            virtual MyString GetAdjacents(MyString v1)=0;
            virtual MyString GetVertex()=0;
            virtual bool isRelated( MyString& str1, MyString& str2);
      };

template<class T>
Graph<T>::Graph(){};

template<class T>
MyString Graph<T>::toString() const
      {
            return MyString("GRAPH");
      };
#endif
//-----------------------------------------
#ifndef GAL_H
#define GAL_H

#include <map>
#include <iostream>
#include <string>
#include <list>
#include "stdlib.h"
#include "MyString.h"
#include "Graph.h"
#include "NodeGAL.h"
```

```
template <class T>
class GAL:public Graph<T>
	{
		map<MyString,NodeGAL<T>,MyStringCompare > g;
		int nvertex;
		int nedges;

	public:
		GAL(){nvertex=0; nedges=0;};
		void addNode (T& v);
		void visualitzacio();
		bool operator==(const Object&) const;
		Object* clone() const;
		MyString toString() const;
		void copy(const Object&);
		void addEdge(MyString v1,MyString v2);
		MyString GetAdjacents(MyString v1);
		MyString GetVertex();
		MyString Tot_GAL();
		bool isRelated( MyString& str1, MyString& str2);

	};

template <class T>
void GAL<T>::visualitzacio()
	{
	MyString aux("\n\t\t\tGAL:\n\t\t\t====\n\n");
	aux = aux + GetVertex();
	cout<<aux;
	}


template <class T>
void GAL<T>::addEdge(MyString v1,MyString v2)
	{
class list< class map<
MyString,NodeGAL<T>,MyStringCompare>::iterator>::iterator  itv1,itv2;
NodeGAL<T> node1,node2;
class list< class map< MyString,NodeGAL<T>,MyStringCompare>::iterator>
*pladj;
class map< MyString,NodeGAL<T>,MyStringCompare>::iterator ite1=this-
>g.find(v1);
class map< MyString,NodeGAL<T>,MyStringCompare>::iterator ite2=this-
>g.find(v2);

	if (ite1==this->g.end() || ite2==this->g.end() )
	{
		cout<<"No existeixen les ciutats"<<endl;
	}
	else
	{
```

```
            pladj=&this->g[v1].ladj;
            itv1=pladj->begin();

            while ( itv1!=pladj->end() && (*itv1)!=ite2) itv1++;

            node1=this->g[v1];
            node2=this->g[v2];

            if (itv1 == pladj->end()) this->nedges++;

            node1.ladj.push_front(ite2);
            node2.ladj.push_front(ite1);

            this->g[v1]=node1;
            this->g[v2]=node2;
      }
}

template <class T>
void GAL<T>::addNode(T& v)
      {
            NodeGAL<T> aux(v);
            this->g[v.getkey()]=aux;
            this->nvertex++;
      }

template <class T>
void GAL<T>::copy(const Object& o)
      {
            const GAL<T>& c = dynamic_cast<const GAL<T>&>(o);
            this->g = c.g;
            this->nvertex = c.nvertex;
            this->nedges = c.nedges;
      }

template <class T>
bool GAL<T>::operator==(const Object& o) const
      {
      bool iguals;
      MyString k1,k2;

      const GAL<T>& graf = dynamic_cast<const GAL<T>& >(o);

      class map<MyString,NodeGAL<T>,MyStringCompare>::const_iterator itg;
      class map<MyString,NodeGAL<T>,MyStringCompare>::const_iterator itt;
      class list< class map<
MyString,NodeGAL<T>,MyStringCompare>::iterator>::const_iterator
itv1,itv2;
      if (graf.nvertex != this->nvertex ) return false;
      if (graf.nedges != this->nedges ) return false;

      itg=graf.g.begin();

      iguals=true;

      while ( itg != graf.g.end() && iguals)
            {
```

```
            itt=this->g.begin();
            while ( itt != this->g.end() && (*itt)!=(*itg))
                  {
                  itt++;
                  }
            iguals= (itt != this->g.end());
            itg++;
            }

      itg=graf.g.begin();

      while ( itg != graf.g.end() && iguals )
      {
      itv1=(*itg).second.ladj.begin();
      while (itv1 != (*itg).second.ladj.end() && iguals )
            {
            k1=(*itg).first;
            k2=(*(*itv1)).first;
            iguals=this->isRelated(k1,k2);
            itv1++;
            }
      itg++;
      }

      return iguals;
      }
template <class T>
bool GAL<T>::isRelated( MyString& str1, MyString& str2)
      {
      class map<MyString,NodeGAL<T>,MyStringCompare>::const_iterator itt;
      class list< class map<
MyString,NodeGAL<T>,MyStringCompare>::iterator>::const_iterator  itv1;
      NodeGAL<T> nod1;

      try
      {

      nod1=this->g[str1];
      this->g[str2];

      itv1=nod1.ladj.begin();

      while (itv1 !=nod1.ladj.end() && !((*(*itv1)).first == str2))
itv1++;

      return itv1 !=nod1.ladj.end();
      }
      catch(...){return false;}

      }

template <class T>
MyString GAL<T>::toString() const
      {
typename map< MyString,NodeGAL<T>,MyStringCompare>::const_iterator
it=this->g.begin();
```

```
	MyString aux("\n\n\t\t\ttoString():"),vertex;
	list<MyString> adjacencies;
	aux = aux + "\n\n";
	aux = aux + "\t\t\tNum. Vertexs: ";
	aux = aux + MyString(this->nvertex);
	aux = aux + "\n\t\t\tNum. d'Adjacencies: ";
	aux = aux + MyString(this->nedges);
	aux = aux + "\n\n";
	aux = aux + "[";

	while(it!=this->g.end())
		{
		vertex = (*it).second.v.toString();
		aux = aux + vertex;
		aux = aux + ",(";
		adjacencies = const_cast<GAL<T>*>(this)-
>GetAdjacents(vertex);

		for(list<MyString>::iterator itl=adjacencies.begin(); itl !=
adjacencies.end();itl++)
		{
		cout<<(*itl)<<" ";
		aux = aux + (*itl);
		aux = aux + ",";
		};
		aux = aux + ")]";
		it++;
		}
	return aux;
	}
template <class T>
Object* GAL<T>::clone() const
	{
		GAL<T>* aux;
		aux=new GAL<T>;
		aux->copy(*this);
		return aux;
	}


template <class T>
list<MyString> GAL<T>::GetAdjacents(MyString& v1)
	{
class list< class map< MyString,NodeGAL<T>,MyStringCompare>::iterator>
*pladj;
class list< class map<
MyString,NodeGAL<T>,MyStringCompare>::iterator>::iterator itl;

	class list<MyString> lret;

	MyString aux(""),adjacencies("");

	pladj = &this->g[v1].ladj;
	itl= pladj->begin();
	aux = aux + v1;
	aux = aux + ",(";
	while(itl!=pladj->end())
```

```
            {
            adjacencies = (*(*itl)).second.v.toString();
            lret.push_front(adjacencies);
            itl++;

            }
      return lret;
      }

template <class T>
T GAL<T>::GetVertex(MyString& vert)
      {
      NodeGAL<T> node1;
      node1=this->g[vert];
      return node1.v;
      }

template <class T>
MyString GAL<T>::Tot_GAL()
      {
      typename map< MyString,NodeGAL<T>,MyStringCompare>::iterator it;
      MyString aux(""),vertex(""),adjacencies("");
      aux = aux + "\n\n{";
      for(it=this->g.begin();it!=this->g.end();it++)
      {
            aux = aux + "[";
            vertex = (*it).second.v.toString();
            aux = aux + GetAdjacents(vertex);
      }
      aux = aux + "}";
      return aux;
      }

#endif

//-----------------------------------
#ifndef Digraph_H
#define Digraph_H

#include <cstdlib>
#include <iostream>
#include "Object.h"
#include "Container.h"
#include "MyString.h"

using namespace std;

template<class T>
class Digraph:public Container
      {
      public:
            Digraph();
            virtual void copy(const Object&)=0;
            virtual Object* clone() const=0;
            virtual void addNode(T&)=0;
            virtual bool operator==(const Object&) const=0;
            virtual void addEdge(MyString,MyString)=0;
```

```
		MyString toString() const;
		virtual MyString GetAdjacents(MyString v1)=0;
		virtual MyString GetVertex()=0;
		virtual bool isRelated( MyString& str1, MyString& str2)=0;

	};

template<class T>
Digraph<T>::Digraph(){};





template<class T>
MyString Digraph<T>::toString() const
	{
		return MyString("DIGRAPH");
	};
#endif
//------------------------------------
#ifndef DAL_H
#define DAL_H

#include <map>
#include <iostream>
#include <string>
#include <list>
#include "stdlib.h"
#include "MyString.h"
#include "Digraph.h"
#include "NodeDAL.h"

template <class T>
class DAL:public Digraph<T>
	{
		map<MyString,NodeDAL<T>,MyStringCompare > g;
		int nvertex;
		int nedges;

	public:
		DAL(){nvertex=0; nedges=0;};
		void addNode (T& v);
		void visualitzacio();
		bool operator==(const Object&) const;
		Object* clone() const;
		MyString toString() const;
		void copy(const Object&);
		void addEdge(MyString v1,MyString v2);
		MyString GetAdjacents(MyString v1);
		MyString GetVertex();
		MyString Tot_DAL();
		bool isRelated( MyString& str1, MyString& str2);

	};
```

```
template <class T>
void DAL<T>::visualitzacio()
      {
      MyString aux("\n\t\t\tDAL:\n\t\t\t====\n\n");
      aux = aux + GetVertex();
      cout<<aux;
      }

template <class T>
void DAL<T>::addEdge(MyString v1,MyString v2)
      {
class list< class map<
MyString,NodeDAL<T>,MyStringCompare>::iterator>::iterator  itv1,itv2;
NodeDAL<T> node1,node2;
class list< class map< MyString,NodeDAL<T>,MyStringCompare>::iterator>
*pladj;
class map< MyString,NodeDAL<T>,MyStringCompare>::iterator ite1=this-
>g.find(v1);
class map< MyString,NodeDAL<T>,MyStringCompare>::iterator ite2=this-
>g.find(v2);

      if (ite1==this->g.end() || ite2==this->g.end() )
            {
            cout<<"No existeixen les ciutats"<<endl;
            }
      else
            {
            pladj=&this->g[v1].ladj;
            itv1=pladj->begin();

            while ( itv1!=pladj->end() && (*itv1)!=ite2) itv1++;

            node1=this->g[v1];

            if (itv1 == pladj->end()) this->nedges++;

            node1.ladj.push_front(ite2);

            this->g[v1]=node1;
            }
}

template <class T>
void DAL<T>::addNode(T& v)
      {
      NodeDAL<T> aux(v);
      this->g[v.getkey()]=aux;
      this->nvertex++;
      }

template <class T>
void DAL<T>::copy(const Object& o)
      {
      const DAL<T>& c = dynamic_cast<const DAL<T>&>(o);
      this->g = c.g;
      this->nvertex = c.nvertex;
      this->nedges = c.nedges;
```

```
      }

template <class T>
bool DAL<T>::operator==(const Object& o) const
      {
      bool iguals;
      MyString k1,k2;

      const DAL<T>& graf = dynamic_cast<const GAL<T>& >(o);

      class map<MyString,NodeDAL<T>,MyStringCompare>::const_iterator itg;
      class map<MyString,NodeDAL<T>,MyStringCompare>::const_iterator itt;
      class list< class map<
MyString,NodeDAL<T>,MyStringCompare>::iterator>::const_iterator
itv1,itv2;
      if (graf.nvertex != this->nvertex ) return false;
      if (graf.nedges != this->nedges ) return false;

      itg=graf.g.begin();

      iguals=true;

      while ( itg != graf.g.end() && iguals)
            {
            itt=this->g.begin();
            while ( itt != this->g.end() && (*itt)!=(*itg))
                  {
                  itt++;
                  }
            iguals= (itt != this->g.end());
            itg++;
            }

      itg=graf.g.begin();

      while ( itg != graf.g.end() && iguals )
      {
      itv1=(*itg).second.ladj.begin();
      while (itv1 != (*itg).second.ladj.end() && iguals )
            {
            k1=(*itg).first;
            k2=(*(*itv1)).first;
            iguals=this->isRelated(k1,k2);
            itv1++;
            }
      itg++;
      }

      return iguals;
      }
template <class T>
bool DAL<T>::isRelated( MyString& str1, MyString& str2)
      {
      class map<MyString,NodeDAL<T>,MyStringCompare>::const_iterator itt;
      class list< class map<
MyString,NodeDAL<T>,MyStringCompare>::iterator>::const_iterator  itv1;
      NodeDAL<T> nod1;
```

```
	try
	{

	nod1=this->g[str1];
	this->g[str2];

	itv1=nod1.ladj.begin();

	while (itv1 !=nod1.ladj.end() && !((*(*itv1)).first == str2))
itv1++;

	return itv1 !=nod1.ladj.end();
	}
	catch(...){return false;}

	}


template <class T>
MyString DAL<T>::toString() const
	{
typename map< MyString,NodeDAL<T>,MyStringCompare>::const_iterator
it=this->g.begin();

	MyString aux("\n\n\t\t\ttoString():"),vertex;
	MyString adjacencies;
	aux = aux + "\n\n";
	aux = aux + "\t\t\tNum. Vertexs: ";
	aux = aux + MyString(this->nvertex);
	aux = aux + "\n\t\t\tNum. d'Adjacencies: ";
	aux = aux + MyString(this->nedges);
	aux = aux + "\n\n";
	aux = aux + "[";

	while(it!=this->g.end())
		{
		vertex = (*it).second.v.toString();
		adjacencies = const_cast<DAL<T>*>(this)-
>GetAdjacents(vertex);
		aux = aux + vertex;
		aux = aux + adjacencies;
		it++;
		}
	return aux;
	}

template <class T>
Object* DAL<T>::clone() const
	{
	DAL<T>* aux;
	aux=new DAL<T>;
	aux->copy(*this);
	return aux;
	}
```

```
template <class T>
list<MyString> DAL<T>::GetAdjacents(MyString& v1)
	{
class list< class map< MyString,NodeDAL<T>,MyStringCompare>::iterator>
*pladj;
class list< class map<
MyString,NodeDAL<T>,MyStringCompare>::iterator>::iterator itl;

	class list<MyString> lret;

	MyString aux(""),adjacencies("");

	pladj = &this->g[v1].ladj;
	itl= pladj->begin();
	aux = aux + v1;
	aux = aux + ",(";
	while(itl!=pladj->end())
		{
		adjacencies = (*(*itl)).second.v.toString();
		lret.push_front(adjacencies);
		itl++;

		}
	return lret;
	}

template <class T>
T DAL<T>::GetVertex(MyString& vert)
	{
	NodeDAL<T> node1;
	node1=this->g[vert];
	return node1.v;
	}


template <class T>
MyString DAL<T>::GetVertex()
	{
	typename map< MyString,NodeDAL<T>,MyStringCompare>::iterator it;
	MyString aux(""),vertex;

	for(it=this->g.begin();it!=this->g.end();it++)
		{
		vertex = (*it).second.v.toString();
		aux = aux + vertex;
		aux = aux + (" - ");
		}
	aux = aux + "\n";
	return aux;
	}


template <class T>
MyString DAL<T>::Tot_DAL()
	{
	typename map< MyString,NodeDAL<T>,MyStringCompare>::iterator it;
	MyString aux(""),vertex(""),adjacencies("");
```

```
	aux = aux + "\n\n{";
	for(it=this->g.begin();it!=this->g.end();it++)
		{
		aux = aux + "[";
		vertex = (*it).second.v.toString();
		aux = aux + GetAdjacents(vertex);
		}
	aux = aux + "}\n";
	return aux;
	}
#endif
//------------------------------------
```

Annex 2: EXEMPLE

```
#ifndef City_H
#define City_H

#include "Vertex.h"
#include "MyString.h"

class City:public Vertex
	{
		MyString name;
		int hab;

	public:
		MyString getkey();
		City();
		City(char* n,int h):name(n),hab(h){};
		City(MyString& n,int h):name(n),hab(h){};
		void copy(const Object&);
		bool operator==(const Object&) const;
		MyString toString() const;
		Object* clone() const;
	};
#endif
//------------------------------------
#include <iostream>

#include "City.h"
#include "Object.h"
#include "MyString.h"

MyString City::getkey()
	{
		return(name);
	};

City::City()
	{
		name=MyString("Lleida");
		hab=10000;
	};
```

```
void City::copy(const Object& o)
	{
		const City& c=dynamic_cast<const City&>(o);
		name=c.name;
		hab=c.hab;
	};

bool City::operator==(const Object& o) const
	{
		const City& c=dynamic_cast<const City&>(o);
		return(name==c.name && hab==c.hab);
	};

MyString City::toString() const
	{
		MyString aux("");
		aux.copy(aux+name);
		return aux;
	};

Object* City::clone() const
	{
		City* aux;
		aux=new City;
		aux->copy(*this);
		return aux;
	};
//------------------------------------
#include <iostream>
#include <stdio.h>
#include <iterator>
#include <string>
#include <stdlib.h>

using namespace std;

#include "GAL.h"
#include "DAL.h"
#include "MyString.h"
#include "City.h"

void Introd_Nova_Ciutat(MyString& nom_Ciutat,int& num_Hab);
void Relacionar_Ciutats(MyString& nc1,MyString& nc2);
void Adjacencies_de_la_Ciutat(MyString& nc1);
void MENU_Ppal(int& opcio);
void MENU_GRAF(int& opcio);
void MENU_GAL(int& opcio);
void ACCIO_GAL(int& opcio);
void MENU_DIGRAF(int& opcio);
void MENU_DAL(int& opcio);
void ACCIO_DAL(int& opcio);
```

```
int main(void)
      {
            int opcio;

            MENU_Ppal(opcio);

            return(0);
      }
void MENU_Ppal(int& opcio)
      {
            do
            {
                  system("clear");
                  cout<<"\n\t\t\tMENU PRINCIPAL:\n\n";
                  cout<<"\t\t1.-GRAF.\n";
                  cout<<"\t\t2.-DIGRAF. \n";
                  cout<<"\t\t3.-Sortir. \n";
                  cout<<"\t\t-->Triar opcio (1-3): ";
                        cin>>opcio;
                  switch(opcio)
                        {
                        case 1:
                              system("clear");
                              MENU_GRAF(opcio);
                              break;
                        case 2:
                              system("clear");
                              MENU_DIGRAF(opcio);
                              break;
                        };
            }while(opcio!=3);
      }
void MENU_GRAF(int& opcio)
      {
            cout<<"\n\t\t\tMENU GRAF:\n\n";
            cout<<"\t\t1.-GAL.\n";
            cout<<"\t\t2.-Sortir. \n";
            cout<<"\t\t-->Triar opcio (1-2): ";
                  do{
                        cin.clear();
                        cin.ignore( 2000, '\n' );
                        cin>>opcio;
                  }while ( cin.fail() );
            switch(opcio)
                  {
                  case 1:
                        system("clear");
                        ACCIO_GAL(opcio);
                        break;
                  }
      }
```

```
void MENU_GAL(int& opcio)
      {
            cout<<"\n\t\t\tMENU GAL:\n\n";
            cout<<"\t\t1.-Insertar:('Lleida',10000) \n";
            cout<<"\t\t2.-Insertar una nova Ciutat. \n";
            cout<<"\t\t3.-Insertar les RELACIONS entre Ciutats. \n";
            cout<<"\t\t4.-Donar les ADJACENCIES d'una Ciutat.\n";
            cout<<"\t\t5.-Vissualitzar tot el Graf.\n";
            cout<<"\t\t6.-Sortir. \n";
            cout<<"\t\t-->Triar opcio (1-6): ";
                  do{
                        cin.clear();
                        cin.ignore( 2000, '\n' );
                        cin>>opcio;
                  }while ( cin.fail() );
      }
void ACCIO_GAL(int& opcio)
{
      GAL<City> gr;
      City aux;
      int num_Hab;
      list<MyString> adj;
      MyString nom_Ciutat(""),nc1(""),nc2(""),crels("");
      do
      {
            MENU_GAL(opcio);
            switch(opcio)
            {
                  case 1:
                        aux=City("Lleida",10000);
                        gr.addNode(aux);
                        break;
                  case 2:
                        Introd_Nova_Ciutat(nom_Ciutat,num_Hab);
                        aux = City(nom_Ciutat,num_Hab);
                        gr.addNode(aux);
                        break;
                  case 3:
                        Relacionar_Ciutats(nc1,nc2);
                        gr.addEdge(nc1,nc2);
                        break;
                  case 4:
                        Adjacencies_de_la_Ciutat(nc1);
                        adj=gr.GetAdjacents(nc1);
                        for(list<MyString>::iterator itl=adj.begin(); itl
!= adj.end();itl++)
                              {
                              cout<<"\n"<<(*itl)<<" ";
                              };
                        break;
                  case 5:
                        cout << gr.Tot_GAL();
                        break;
            };
      }while(opcio!=6);
}
```

```
void MENU_DIGRAF(int& opcio)
	{
		cout<<"\n\t\t\tMENU DIGRAF:\n\n";
		cout<<"\t\t1.-DAL.\n";
		cout<<"\t\t2.-Sortir. \n";
		cout<<"\t\t-->Triar opcio (1-3): ";
			do{
				cin.clear();
				cin.ignore( 2000, '\n' );
				cin>>opcio;
			}while ( cin.fail() );
		switch(opcio)
			{
			case 1:
				system("clear");
				ACCIO_DAL(opcio);
				break;
			}
	}
void MENU_DAL(int& opcio)
	{
		cout<<"\n\t\t\tMENU DAL:\n\n";
		cout<<"\t\t1.-Insertar:('Lleida',10000) \n";
		cout<<"\t\t2.-Insertar una nova Ciutat. \n";
		cout<<"\t\t3.-Insertar les RELACIONS entre Ciutats. \n";
		cout<<"\t\t4.-Donar les ADJACENCIES d'una Ciutat.\n";
		cout<<"\t\t5.-Vissualitzar tot el Digraf.\n";
		cout<<"\t\t6.-Sortir. \n";
		cout<<"\t\t-->Triar opcio (1-6): ";
			do{
				cin.clear();
				cin.ignore( 2000, '\n' );
				cin>>opcio;
			}while ( cin.fail() );
	}
void ACCIO_DAL(int& opcio)
{
	DAL<City> gr;
	City aux;
	int num_Hab;
	list<MyString> adj;
	MyString nom_Ciutat(""),nc1(""),nc2(""),crels("");
	do
	{
		MENU_DAL(opcio);
		switch(opcio)
		{
			case 1:
				aux=City("Lleida",10000);
				gr.addNode(aux);
				break;
			case 2:
				Introd_Nova_Ciutat(nom_Ciutat,num_Hab);
				aux = City(nom_Ciutat,num_Hab);
				gr.addNode(aux);
				break;
			case 3:
```

```
                        Relacionar_Ciutats(nc1,nc2);
                        gr.addEdge(nc1,nc2);
                        break;
                  case 4:
                        Adjacencies_de_la_Ciutat(nc1);
                        adj=gr.GetAdjacents(nc1);
                        for(list<MyString>::iterator itl=adj.begin(); itl
!= adj.end();itl++)
                              {
                              cout<<"\n"<<(*itl)<<" ";
                              };
                        break;
                  case 5:
                        cout << gr.Tot_DAL();
                        break;
            };
      }while(opcio!=6);
}
void Introd_Nova_Ciutat(MyString& nom_Ciutat,int& num_Hab)
      {
            char nom_c[30];

            cout<<"\n\n\t\tInserta el nom de la nova Ciutat: ";
            scanf("%s",&nom_c);
            nom_Ciutat = MyString(nom_c);

            cout<<"\n\n\t\tInserta el numero d'Habitants: ";
            cin >> num_Hab;

      }
void Relacionar_Ciutats(MyString& nc1,MyString& nc2)
      {
            char nom_c1[30],nom_c2[30];

            cout<<"\n\n\t\tIntrodueix el nom de la Ciutat: ";
                  scanf("%s",&nom_c1);
            cout<<"\n\t\tIntrodueix la Ciutat a relacionar: ";
                  scanf("%s",&nom_c2);
            nc1 = MyString(nom_c1);
            nc2 = MyString(nom_c2);
      }
void Adjacencies_de_la_Ciutat(MyString& nc1)
      {
            cout<<"\n\t\t Introdueix el nom de la Ciutat: ";
            nc1.get(cin,'\0');
      }
```

6.-Bibliografia:

[1] Ribó,Josep M.; Maksumic,Ismet; Cehajic, Sinisa: "Introduction to Object Oriented Programing in C++". Ed.Universitat de Lleida, 2005
ISBN 84-8409-199-6.

[2] Gimbert,Joan; Moreno,Ramiro; Ribó,Josep M.; Valls,Magda: "Apropament a la teoria de grafs i als seus algorismes".
Ed.Universitat de Lleida, 1998
ISBN 84-89727-65-1.

[3] Stroustrup, Bjarne: "C++ programming Language".
Ed. Addison Wesley ISBN 0-201-53992-6.

[4] Tutorial C++ STL: www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html
C/C++ Reference: http://www.cppreference.com/index.html