UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Politècnica Superior d'Enginyeria
de Vilanova i la Geltrú

PUBLICACIÓ DOCENT

APUNTS DE TEORIA

ESIN

AUTOR: Bernardino Casas i Jordi Esteve
ASSIGNATURA: Estructura de la Informació (ESIN)
QUADRIMESTRE: Q3
TITULACIÓ: Grau en Informàtica
DEPARTAMENT: Ciències de la Computació

Vilanova i la Geltrú, 20 d’agost de 2018

Índex de continguts

TEMA 1. PROGRAMACIÓ ORIENTADA A OBJECTES

1.1. ABSTRACCIÓ

L'abstracció és un mecanisme que permet ocultar els detalls no rellevants. Així es redueix el volum d'informació que es manipula a la vegada.

En la programació s'usen dos mecanismes bàsics d'abstracció:

- **Abstracció FUNCIONAL**: Es sap QUÈ fa una funció però no es sap COM ho fa. Els primers llenguatges de programació (C, FORTRAN, COBOL) ja incorporen aquest mecanisme.
- **Abstracció DE DADES**: La idea és la mateixa que l'anterior però aplicada a les dades. Es sap QUÈ es pot fer amb les dades però no COM s'aconsegueix. Els llenguatges de programació incorporen aquest mecanisme molt més tard (C++, java, PHP, Python, ...).

Definició: Podem definir un tipus abstracte de dades (TAD) com a un conjunt de valors sobre els quals podem aplicar un conjunt donat d'operacions que compleixen determinades propietats.

QUÈ VOL DIR ABSTRACTE?
Utilitzem el terme **abstracte** per denotar el fet de que els objectes d'un tipus o classe poden ser manipulats mitjançant les seves operacions sense que sigui necessari conèixer cap detall sobre la seva representació.

Exemple: En els llenguatges de programació existeixen tipus predefinits (enter, real, lògic, caràcter, ...). Aquests tipus en realitat són TADs que els tenim a la nostra disposició. Un programa qualsevol pot efectuar l'operació suma d'enters ($x+y$) amb la seguretat que aquesta operació sempre calcularà la suma de x i y independentment de la representació interna dels enters a la màquina (complement a 2, signe i magnitud, ...).

Idea important: La manipulació dels objectes d'un tipus o classe només depèn de l'especificació dels mètodes i les propietats de les seves operacions i valors, i és independent de la implementació d'aquesta classe.

Dues conseqüències:

1) Per a una especificació (única) d'una classe poden haver-hi moltes implementacions associades, totes elles igualment vàlides. Les característiques de cada implementació són les que determinen sota quines condicions és recomanable usar-la.

2) Qualsevol canvi en la implementació d'una classe dins d'una aplicació no afecta al seu ús, la qual cosa confereix a les aplicacions un grau molt elevat de robustesa als canvis.

1.2. PROGRAMACIÓ ORIENTADA A OBJECTES

La **programació orientada a objectes** expressa un programa com un conjunt d'objectes que col·laboren entre ells per realitzar tasques.

Un **objecte** és un grapat de funcions i procediments, tots relacionats amb un concepte particular del Món Real com ara: una taula, un compte bancari o un jugador de futbol. Altres parts del programa poden accedir a l'objecte només cridant les seves funcions i procediments (aquelles que puguin ser cridades des de l'exterior).

OBJECTES

Els objectes són entitats que combinen estat, comportament i identitat.

Estat : Seran un o varis atributs als que s'hauran assignat uns valors concrets (dades).

Comportament : Està definit pels procediments o mètodes amb que es pot operar aquest objecte, és a dir, quines operacions es poden realitzar amb ell.

Identitat : Propietat d'un objecte que el diferencia de la resta, dit d'una altra manera, és el seu identificador.

La identitat dels objectes serveix en programació per comparar si dos objectes són iguals. No és estrany trobar que a molts llenguatges de programació la identitat d'un objecte està

determinada per la direcció de memòria de l'ordinador on es troba l'objecte.

Habitualment només es permet canviar l'estat d'un objecte mitjançant els seus mètodes.

Un **objecte** es pot veure com una caixa que permet guardar la representació d'un valor abstracte. La classe a la que pertany un objecte defineix els valors permesos i els mètodes (operacions) que es poden aplicar sobre l'objecte. Un **objecte** és una instància concreta d'una classe.

A primera vista pot semblar molta feina escriure procediments i funcions per controlar l'accés a l'estructura que implementa una classe. Però aquesta metodologia de disseny aporta propietats avantatjoses:

- **Correctesa:** Facilitat de prova.
- **Eficiència:** Ja que la implementació es pot canviar depenent de l'ús de la classe.
- **Llegibilitat:** Ja que augmenta el nivell dels conceptes que intervenen als programes.
- **Modificabilitat** i **manteniment:** Ja que és més modular.
- **Organització:** Repartició de feines en l'equip de desenvolupament.
- **Reusabilitat:** Reutilització de les classes en altres aplicacions.

- **Transparència:** Les classes són transparents a la implementació, per tant els programes estan escrits en un nivell més alt d'abstracció.

1.3. ESPECIFICACIÓ D'UNA CLASSE

L'especificació d'una **classe** ens permet formular el comportament d'aquesta classe. Una especificació ha de ser sempre precisa i completa. Aquesta es composa de:

- La **SIGNATURA**, que està formada per tots els mètodes que té associada la classe.
- La descripció del **COMPORTAMENT** dels mètodes o operacions. Això es farà amb una notació informal. El comportament en aquests apunts estarà ubicat *abans* del mètode i en un *requadre*.

S'emmagatzemen les especificacions en fitxers amb extensió **.HPP**.

En la definició d'una classe hi ha dos tipus de components:

- Els **atributs** que guarden la informació continguda en un objecte. Tots els objectes d'una classe tenen els mateixos atributs, i poden diferir en el valor dels atributs. Els atributs representen al valor abstracte d'aquell objecte, també anomenat **estat**.
- Els **mètodes** són les operacions que es permeten aplicar sobre els objectes de la classe.

Les diferents instàncies d'una classe (objectes) s'identifiquen generalment per un nom o identificador propi. Si X és el nom d'una classe, la declaració

```
X meu_obj;
```

diu que *meu_obj* és un objecte de la classe X. *meu_obj* és l'identificador de l'objecte.

1.3.1. Classificació de les operacions

Les operacions d'una classe les podem classificar en:

- Operacions **CREADORES**: Són funcions que serveixen per crear objectes nous de la classe, ja siguin objectes inicialitzats amb un mínim d'informació o el resultat de càlculs més complexos. La crida a aquestes funcions és un dels casos particulars on es permet l'assignació entre objectes.

  En C++ es pot definir un tipus particular d'operacions creadores, anomenades **constructores**. Es tracta de funcions especials ja que:

  - ➔ NO tenen tipus de retorn. Sempre retornen un nou objecte de la classe.
  - ➔ Tenen com a nom el mateix de la classe.

> **OPERACIONS GENERADORES**
> Anomenem operacions **GENERADORES** al **subconjunt mínim** de les operacions creadores que permeten generar, per aplicacions successives, tots els objectes possibles d'una classe.

- Operacions **MODIFICADORES**: Transformen l'objecte que les invoca, si és necessari amb informació aportada per altres paràmetres. Conceptualment no haurien de poder modificar altres objectes encara que C++ permet fer-ho mitjançant el pas de paràmetres per referència. Normalment seran accions.

- Operacions **CONSULTORES**: Proporcionen informació sobre l'objecte que les invoca, potser amb l'ajuda d'altres paràmetres. Generalment són funcions, menys si han de retornar varis resultats, que seran accions amb diferents paràmetres de sortida. No retornen objectes nous de la classe.

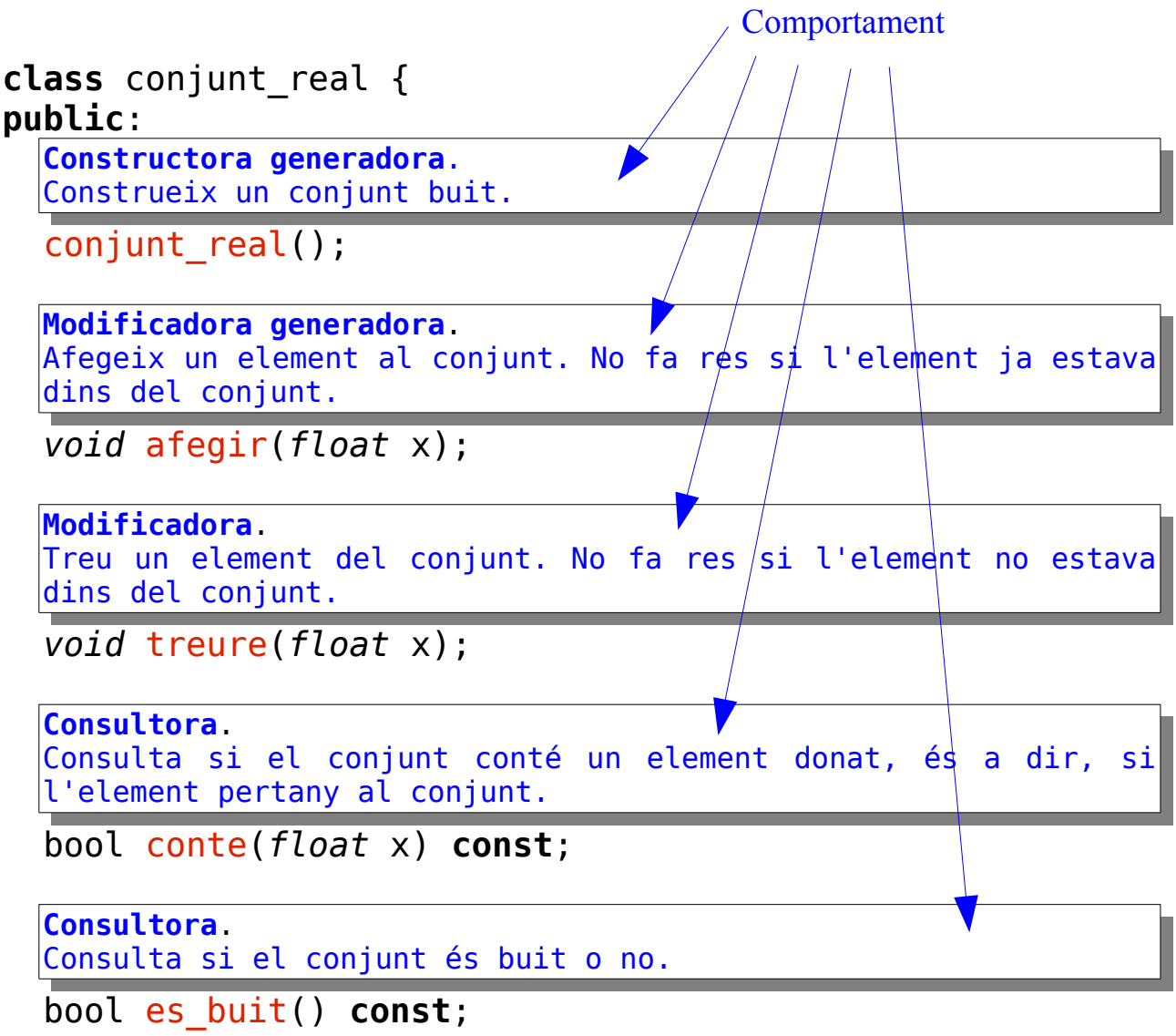
  En C++ les operacions consultores SEMPRE inclouen el modificador **const** al final de la capçalera del mètode.

- Operacions **DESTRUCTORES:** Cal implementar operacions per destruir l'objecte, doncs sovint aquest ocupa espai de memòria. Normalment només cal definir-lo quan l'objecte utilitza memòria dinàmica i cal alliberar-la.

En resum:

* Un mètode **constructor** descriu como es construeix un objecte (instància).
* Un mètode que examina però que no canvia l'estat de l'objecte associat és un **consultor**.
* Un mètode que canvia l'estat de l'objecte associat és un **modificador**.
* Un mètode que destrueix l'objecte s'anomena **destructor**.

Exemple: Ara veurem la signatura de la classe *conjunt de reals* amb alguns dels seus mètodes. Per cada mètode s'ha indicat quin tipus d'operació és.




```
class conjunt_real {
public:
   Constructora generadora.
   Construeix un conjunt buit.
   conjunt_real();

   Modificadora generadora.
   Afegeix un element al conjunt. No fa res si l'element ja estava
   dins del conjunt.
   void afegir(float x);

   Modificadora.
   Treu un element del conjunt. No fa res si l'element no estava
   dins del conjunt.
   void treure(float x);

   Consultora.
   Consulta si el conjunt conté un element donat, és a dir, si
   l'element pertany al conjunt.
   bool conte(float x) const;

   Consultora.
   Consulta si el conjunt és buit o no.
   bool es_buit() const;

};
```

Per norma general les especificacions s'emmagatzemaran en fitxers amb el mateix nom que la classe que conté. Aquesta especificació es desaria a **conjunt_real.hpp**.

1.4. MECANISMES PER CREAR CLASSES COMPLEXES

Quan especifiquem classes més complexes, ens trobem amb la necessitat de poder usar alguna de les classes especificades prèviament. Estudiarem tres mecanismes:

- Ús d'altres classes.
- Visibilitat.
- Parametrització i instanciació.

1.4.1. Ús d'altres classes

Mitjançant la clàusula especial del llenguatge C++ **#include** podem incorporar altres especificacions de classes, operacions i constants emmagatzemades en fitxers.

Es pot usar de dues maneres diferents:

> **#include "nomf"**: es busca el fitxer *nomf* en el directori actual.
> **#include <nomf>**: es busca el fitxer *nomf* en els directoris de la biblioteca estàndard.

Aquest mecanisme permet:

- Definir una classe nova que necessita classes ja existents. Per exemple, per definir la classe *conjunt_racionals* necessitem utilitzar la classe *racional*.
- Enriquir una o més classes amb noves operacions (usant biblioteques externes).

1.4.2. Visibilitat

Si no es diu el contrari tots els mètodes i atributs d'una classe estaran amagats per l'exterior, és a dir, no seran visibles. Per tal d'indicar una altra visibilitat cal utilitzar el **modificadors de visibilitat**:

```
public:
    //capçaleres dels mètodes visibles
    ...

private:
    //capçaleres dels mètodes ocults
    ...
```

Depenent del modificador la visibilitat serà:

- **public:** Un component que és **públic** pot ser accedit per qualsevol mètode de qualsevol classe.
- **private**: Un component **privat** només pot ser accedit des de la mateixa classe.

Normalment els atributs són declarats com privats i els mètodes d'ús general es declaren com públics.

Mitjançant la declaració privada dels atributs es garanteix que es fa un bon ús dels objectes, mantenint la coherència de la informació.

Si bé es poden fer visibles totes les operacions d'una classe, pot ser interessant amagar algunes d'elles. Això és necessari per tal d'evitar un ús indiscriminat dels mètodes o per poder-los redefinir.

1.4.3. Parametrització i instanciació

És un mecanisme que permet crear un motlle per especificar diferents tipus. Es pot descriure una classe que depengui de diversos paràmetres que poden ser noms de tipus o noms d'operacions. És el que s'anomena **especificació parametritzada o genèrica**.

Exemple: Fixem-nos en la semblança de les especificacions dels conjunts de reals i dels conjunts d'enters. En l'exemple només s'inclou la part pública de la classe de cada classe.

```
class conjunt_real {
public:
  Construeix un conjunt buit.
  conjunt_real();

  Afegeix un element al conjunt. No fa res si l'element ja estava
  dins del conjunt.
  void afegir(float x);

  Treu un element del conjunt. No fa res si l'element no estava
  dins del conjunt.
  void treure(float x);

  Consulta si el conjunt conté un element donat, és a dir, si
  l'element pertany al conjunt.
  bool conte(float x) const;

  Consulta si el conjunt és buit o no.
  bool es_buit() const;

};
```

```
class conjunt_enter {
public:
  Construeix un conjunt buit.
  conjunt_enter();

  Afegeix un element al conjunt. No fa res si l'element ja estava
  dins del conjunt.
  void afegir(int x);

  Treu un element del conjunt. No fa res si l'element no estava
  dins del conjunt.
  void treure(int x);

  Consulta si el conjunt conté un element donat, és a dir, si
  l'element pertany al conjunt.
  bool conte(int x) const;

  Consulta si el conjunt és buit o no.
  bool es_buit() const;

};
```

Una solució a la repetició de classes semblants és la de fer una **especificació genèrica** o parametritzada (dit també classe plantilla).

Les classes plantilla en C++ han de començar amb la paraula reservada **template**. A continuació cal incloure com a mínim un paràmetre entre < >. Aquest paràmetre ha d'estar precedit per la paraula reservada **class** o **typename**. Un cop fet això ja es pot escriure el cos de la declaració de la classe (amb els mètodes i els atributs).

```
template <typename T>
class X {

  ...

};
```

En el següent exemple definirem una classe genèrica de conjunt d'elements; el paràmetre formal és el tipus d'element (ELEM):

```
template <typename ELEM>
class conjunt {
public:
  Construeix un conjunt buit.
  conjunt();

  Afegeix un element al conjunt. No fa res si l'element ja estava
  dins del conjunt.
  void afegir(const ELEM &x);

  Treu un element del conjunt. No fa res si l'element no estava
  dins del conjunt.
  void treure(const ELEM &x);

  Consulta si el conjunt conté un element donat, és a dir, si
  l'element pertany al conjunt.
  bool conte(const ELEM &x) const;

  Consulta si el conjunt és buit o no.
  bool es_buit() const;

};
```

El conjunt de naturals o de reals els definirem mitjançant una instància de la classe genèrica, la qual cosa consisteix a associar uns paràmetres reals als formals:

```
// creació d'un conjunt d'enters
conjunt<int> cj1;

// creació d'un conjunt de reals
conjunt<float> cj2;
```

1.5. CRITERIS EN LA IMPLEMENTACIÓ DE CLASSES

Una vegada hem especificat una classe, és el moment d'implementar-la en termes del llenguatge de programació.

QUÈ VOL DIR IMPLEMENTAR?

Implementar un classe consisteix en:

1. Escollir una **representació "adient" per la classe**.
2. **Codificar les operacions visibles** de la classe en funció de la representació escollida, de manera que segueixi l'especificació.

Representació "adient" vol dir que totes les operacions són codificades amb el màxim d'eficiència temporal i espacial (és molt difícil que totes les operacions ho compleixin). En general es determinen una sèrie d'operacions crítiques que necessiten ser el més ràpid possible, i sobre aquestes demanarem el màxim d'eficiència temporal possible sense que se'ns dispari excessivament el cost espacial.

- Per construir la representació disposem de **mecanismes d'estructuració** de tipus propis dels llenguatges de programació: vectors, tuples i punters.
- Per implementar les operacions disposem de les típiques **estructures de control**: seqüencials, alternatives, repetitives, ... I també de mecanismes d'encapsulament de codi en funcions i accions.

La implementació de la classe s'emmagatzema en un fitxer .CPP. Així podem tenir més d'una implementació per a una mateixa especificació.

A més a més, com ja hem vist una implementació pot usar altres classes. Això es declara amb la clàusula **#include**.

INCLOURE ESPECIFICACIONS

Quan es vulgui usar una classe amb el **include** SEMPRE inclourem les especificacions i MAI les implementacions, ja què amb les primeres coneixem el comportament de la classe, que és el que realment ens interessa.

D'aquesta forma programarem independentment de les implementacions.

Exemple: Anem a implementar la classe conjunt d'elements. En l'especificació que hem vist fins ara no hi ha cap limitació en el nombre d'elements que hi cap en el conjunt: és una especificació de conjunts infinits.

Una possible representació feta amb taules estàtiques no podrà treballar amb conjunts infinits ja que hi haurà una dimensió màxima.

Per aquest motiu s'ha afegit:

- una constant pública MAX que és el nombre màxim d'elements del conjunt
- una nova operació consultora es_ple per saber si el conjunt ja està ple i així evitar possibles errors.

La signatura de la classe conjunt d'elements amb la representació (part privada) seria:

```
template <typename ELEM>
class conjunt {
public:
  static const int MAX = 50;

  Construeix un conjunt buit.
  conjunt();

  Afegeix un element al conjunt. No fa res si l'element ja estava
  dins del conjunt o si el conjunt està ple.
  void afegir(const ELEM &x);

  Treu un element del conjunt. No fa res si l'element no estava
  dins del conjunt.
  void treure(const ELEM &x);

  Consulta si el conjunt conté un element donat, és a dir, si
  l'element pertany al conjunt.
  bool conte(const ELEM &x) const;

  Consulta si el conjunt és buit o no.
  bool es_buit() const;

  Consulta si el conjunt és ple o no.
  bool es_ple() const;

private:
  ELEM _A[MAX];
  int _pl;
};
```

TEMA 2. EFICIÈNCIA TEMPORAL I ESPACIAL

2.1. INTRODUCCIÓ

Un dels objectius d'un programa és mantenir baix el consum de recursos. El concepte d'**eficiència** és un concepte relatiu, en el sentit de que es compara l'eficiència d'un programa amb un altre que fa el mateix. Així doncs direm que un programa és ineficient si n'hi ha un altre de més eficient. I direm que un programa és eficient quan no se'ns acut un altre més eficient.

Mesurar l'eficiència d'un algorisme equival a mesurar la quantitat de recursos necessaris per a la seva execució.

Per obtenir les prediccions d'eficiència d'un algorisme farem un anàlisis d'aquest i estudiarem les instruccions una per una. Hem de dir que l'eficiència depèn d'una sèrie de factors:

- La màquina on s'executa.
- La qualitat del codi generat (compilador, linker, ...).
- La grandària de les dades.

A priori els dos primers factors no els podem controlar: donat un algorisme no sabem quina màquina utilitzarem ni tampoc com serà el codi generat.

El que farem serà mesurar l'eficiència depenen de la grandària de les dades d'entrada:

1er) Determinar que entenem per grandària de dades.

2on) Descomposar l'algorisme en unitats fonamentals i calcular l'eficiència per a cada una d'elles.

Exemples:

- Avaluar una expressió: Suma dels temps d'execució de les operacions o funcions que hi hagin.
- Avaluar una assignació a una variable: Temps d'avaluar l'expressió + temps d'assignació.
- Lectura d'una variable: Temps constant de lectura.
- Avaluar una alternativa: Temps d'avaluar l'expressió booleana + temps màxim de les dues branques (cas pitjor).
- Avaluar una repetitiva (bucle): Suma del temps de cada volta (multiplicar el nombre de voltes per la suma del temps de la condició del bucle més el temps de les instruccions del cos).

2.2. NOTACIONS ASIMPTÒTIQUES

Utilitzarem aquestes notacions per a classificar funcions en base a la seva velocitat de creixement respecte a la grandària de les dades (ordre de magnitud).

Fou introduïda a:

> D. E. Knuth, “Big Omicron and Big Omega and Big Theta”, ACM SIGACT News, 8 (1976), pàg. 18-23.

La idea és la d'establir un ordre relatiu entre funcions. Donades dues funcions $f = f(x)$ i $g=g(x)$, hi pot haver valors de x on $f(x)<g(x)$ i valors de x on $f(x)>g(x)$, per la qual cosa no té sentit dir que, per exemple, $f(x)<g(x)$.

Per exemple, $1000n>n^2$ per a valors petits de n, però n^2 creix més ràpidament que $1000n$ i, per tant, $n^2>1000n$ per a valors grans de n.

2.2.1. Definicions

- La notació **O gran**:

> O(f): Denota el conjunt de les funcions g que com a molt creixen tan ràpidament com f. La seva definició formal és:
>
> $$O(f) = \{ g \ / \ \exists c_o \in R^+ \quad \exists n_o \in N \quad \forall n \geq n_o \quad g(n) \leq c_o f(n) \}$$
>
> n_o: Indica a partir de quin punt f és una cota superior per a g.
>
> c_o: Formalitza l'expressió "mòdul constant multiplicatiu".

Exemples:

> $f(n)=n^2 \quad g(n)=n \qquad \Rightarrow \quad g \in O(f)$ però $f \notin O(g)$
>
> $f(n)=3n^2 \quad g(n)=100n^2 \qquad \Rightarrow \quad g \in O(f)$ i $f \in O(g)$

- La notació **Ω gran** (Omega gran):

  Ω (f): Denota el conjunt de les funcions *g* que creixen tan o més ràpidament que *f*. La seva definició formal és:

  $$\Omega(f) = \{ g \ / \ \exists c_o \in R^+ \ \ \exists n_o \in N \ \ \forall n \geq n_o \ \ g(n) \geq c_o f(n) \}$$

  n_o: Indica a partir de quin punt *f* és una cota inferior per a *g*.

Exemples:

$f(n)=n^2 \quad g(n)=n \qquad \Rightarrow \quad f \in \Omega(g)$ però $g \notin \Omega(f)$

$f(n)=3n^2 \quad g(n)=100n^2 \qquad \Rightarrow \quad g \in \Omega(f)$ i $f \in \Omega(g)$

- La notació **Θ gran** (Theta gran):

  $\Theta(f)$:Denota el conjunt de les funcions *g* que creixen exactament al mateix ritme que *f*. La seva definició formal és:

  $$\lim_{n\to\infty} f(n)/g(n) \in R-\{0\} \Rightarrow g \in \Theta(f) \ \text{ i } \ f \in \Theta(g)$$

  o també es pot definir com:

  $$\Theta(f) = \{ g \ / \ g \in O(f), \ g \in \Omega(f) \}$$

Exemples:

$f(n)=n^2 \quad g(n)=n \qquad \Rightarrow \quad f \notin \Theta(g)$ i $g \notin \Theta(f)$

$f(n)=3n^2 \quad g(n)=100n^2 \qquad \Rightarrow \quad g \in \Theta(f)$ i $f \in \Theta(g)$

La Θ **gran** és la notació que <u>més utilitzarem</u>.

2.2.2. Propietats

Aquestes propietats són vàlides per les 3 notacions (excepte la 7).

1) **Reflexivitat**:

$$f \in \Theta(f)$$

2) **Transitivitat**:

$$\text{Si } h \in \Theta(g) \text{ i } g \in \Theta(f) \Rightarrow h \in \Theta(f)$$

3) **Regla de la suma**:

$$\text{Si } g_1 \in \Theta(f_1) \text{ i } g_2 \in \Theta(f_2) \Rightarrow g_1+g_2 \in \Theta(\max(f_1, f_2))$$

o el que és el mateix:

$$\Theta(f_1) + \Theta(f_2) = \Theta(\max(f_1, f_2))$$

4) **Regla del producte**:

$$\text{Si } g_1 \in \Theta(f_1) \text{ i } g_2 \in \Theta(f_2) \Rightarrow g_1 \cdot g_2 \in \Theta(f_1 \cdot f_2)$$

o el que és el mateix:

$$\Theta(f_1) \cdot \Theta(f_2) = \Theta(f_1 \cdot f_2)$$

5) **Invariança aditiva**:

$$\forall c \in R^+ \quad g \in \Theta(f) \Leftrightarrow c+g \in \Theta(f)$$

6) **Invariança multiplicativa**:

$$\forall c \in R^+ \quad g \in \Theta(f) \Leftrightarrow c \cdot g \in \Theta(f)$$

7) **Simetria**:

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$

2.2.3. Formes de creixement freqüents

$\Theta(1)$: Cost **constant** (no depèn de la mida de les dades).

$\Theta(\log n)$: Cost **logarítmic** (creix a poc a poc).

> **Exemple**: Cerca dicotòmica en un vector ordenat de n elements.

$\Theta(n)$: Cost **lineal**.

> **Exemple**: Cerca seqüencial d'un element en un vector desordenat de n elements.

$\Theta(n \cdot \log n)$: Cost **quasi-lineal**.

> **Exemple**: Ordenació d'un vector de n elements.

$\Theta(n^k)$: k és constant. Cost **polinòmic**.

> $\Theta(n^2)$: Cost **quadràtic**. **Exemple**: El recorregut d'una matriu de n files i n columnes.
>
> $\Theta(n^3)$: Cost **cúbic**. **Exemple**: El producte de dues matrius de n files i n columnes.

$\Theta(k^n)$: k és constant. Cost **exponencial**.

> **Exemple**: La cerca (heurística) en un espai d'estats d'amplada k i profunditat n.

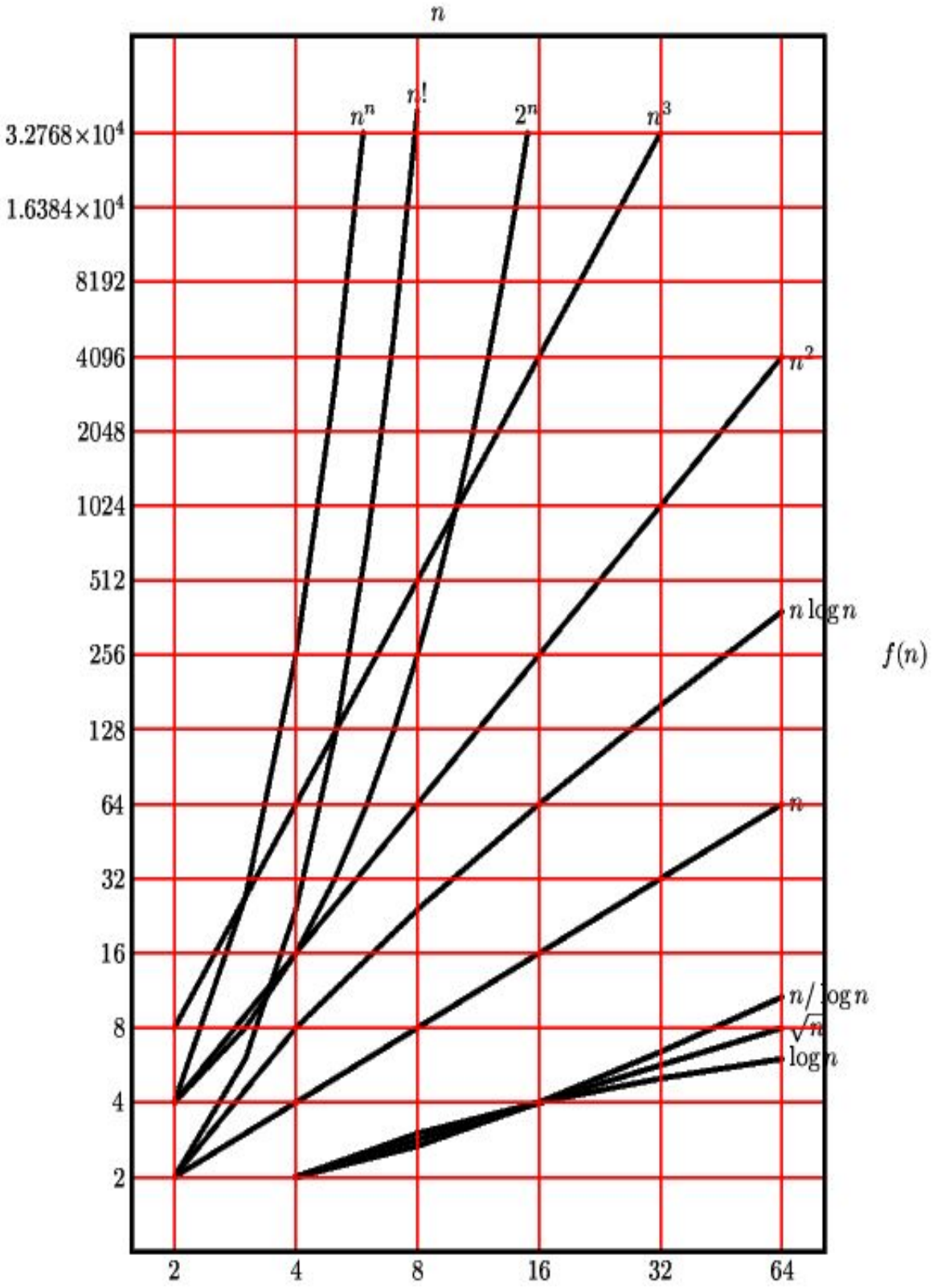
$\Theta(n!)$: Cost **factorial**.

Taula comparativa de les formes de creixement freqüents

n	$\Theta(\log n)$	$\Theta(\sqrt{n})$	$\Theta(n)$	$\Theta(n \log n)$
1	0	1	1	0
10	2,3	3	10	23
100	4,6	10	100	461
10^3	6,9	32	10^3	6908
10^4	9,2	100	10^4	92103
10^5	11,5	316	10^5	1151290

n	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$	$\Theta(n!)$
1	1	1	2	1
10	100	10^3	1024	10^6
100	10^4	10^6	10^{30}	10^{158}
10^3	10^6	10^9	10^{301}	10^{2567}
10^4	10^8	10^{12}	10^{3010}	10^{35659}
10^5	10^{10}	10^{15}	10^{30103}	10^{456573}

Gràfica comparativa de les formes de creixement freqüents:



2.3. ANÀLISI ASIMPTÒTICA DE L'EFICIÈNCIA TEMPORAL D'UN ALGORISME

Per a calcular el cost d'un algorisme utilitzarem el següent:

1) Operacions bàsiques:

Assignació, operacions de lectura o escriptura (E/S) i les comparacions (==, != , <, ...) de tipus de dades elementals.

Cost : $\Theta(1)$ (constant)

2) Estructura **seqüencial**:

instrucció$_1$;
instrucció$_2$;
...
instrucció$_n$;

S'aplica la regla de la suma:

T(seqüència) = màxim(cost(instrucció$_1$), cost(instrucció$_2$), ..., cost(instrucció$_n$))

3) Estructura **alternativa**:

```
if (cond) {
     i1;
}
else {
     i2;
}
```

T(alternativa) = màxim(cost(cond), cost(i_1), cost(i_2))

4) Estructura **repetitiva**:

```
while (cond) {
     cos;
}
```

T(repetitiva) = $\Sigma_{nº\_iteracions}$(cost(cond) + cost(cos)) =

nº_iteracions * màxim(cost(cond), cost(cos))

Resum de costos

Operacions bàsiques: $\Theta(1)$

Estructura seqüencial:
màxim(Θ(instruccio_1), ..., Θ(instruccio_n))

Estructura alternativa:
màxim(Θ(cond), Θ(cos_if), Θ(cost_else))

Estructura repetitiva:
nº_iteracions * màxim(Θ(cond), Θ(cos))

Exemple. Ordenació d'un vector pel mètode de la bombolla.

```
void OrdenacioBombolla(int A[MAX], nat n) {
  nat i, j;
  for (i=1; i<=n-1; i++) {
    for (j=n; j>=i+1; j--) {
      if (A[j-1] > A[j]) {
        nat aux = A[j-1];
        A[j-1] = A[j];
        A[j] = aux;
      }
    }
  }
}
```

- El cos de la condició té cost constant o $\Theta(1)$
- La condició A[j-1] > A[j] té cost constant i, per tant, el **if** (...) {...} té cost $\Theta(1)$.
- Es fan n-i iteracions interiors: **for** (j=n; j>=i+1; j--) {...}
- Es fan n-1 iteracions exteriors: **for** (i=1; i<=n-1; i++) {...}

Per tant el cost total serà:

$$\sum_{i=1}^{n-1}(n-i)=1+2+3+\cdots+n-1=\frac{n(n+1)}{2}-n=\frac{n(n-1)}{2}=\Theta(n^2)$$

Exemple: Càlcul del factorial de n amb una funció recursiva.

```
nat factorial(nat n) {
    if (n==0) {
        return(1);
    }
    else {
        return(n * factorial(n-1));
    }
}
```

L'equació de la recurrència d'aquesta funció serà:

$$T(n)=\begin{cases} k_1 & si\, n=0 \\ T(n-1)+k_2 & si\, n\geqslant 1 \end{cases}$$

Sent k_1 i k_2 constants

Per tant el cost total serà:

$$\begin{aligned} T(n) &= T(n-1)+k_2= \\ &= T(n-2)+k_2+k_2= \\ &= T(n-3)+k_2+k_2+k_2= \\ &= T(n-4)+k_2+k_2+k_2+k_2= \\ &= \cdots = \\ &= T(n-n)+\underbrace{k_2+\cdots+k_2}_{n} \\ &= T(0)+n\,k_2= \\ &= k_1+n\,k_2= \ \Theta(n) \end{aligned}$$

2.3.1. Teoremes mestres

Les equacions de recurrència apareixen en els algorismes recursius. Fent una simplificació tenim dos casos:

- La talla del problema decreix **aritmèticament**

  Una crida recursiva sobre un problema de talla *n* genera *a* crides recursives sobre subproblemes de talla *n-c,* amb *c* constant.

$$T(n)=a\cdot T(n-c)+g(n)$$

- La talla del problema decreix **geomètricament**

  Una crida recursiva sobre un problema de talla *n* genera *a* crides recursives sobre subproblemes de talla *n/b,* amb *b* constant i *b*>1.

$$T(n)=a\cdot T(n/b)+g(n)$$

ÚS DELS TEOREMES MESTRES

Per tal de poder aplicar algun dels teoremes mestres, l'equació de la recurrència ha de encaixar amb la del teorema.

Per exemple, la funció g(n), com bé diu el teorema, ha de ser d'ordre polinòmic.

Això implica que hi haurà vegades en que no es pugui aplicar el teorema mestre i calgui calcular el cost de manera "manual".

Decreixement aritmètic

Teorema: *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n-c)+g(n) & si\ n_0\leqslant n \end{cases}$$

on n_0 és una constant, $c\geqslant 1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb k constant. Aleshores,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<1 \\ \Theta\left(n^{k+1}\right) & si\ a=1 \\ \Theta\left(a^{n/c}\right) & si\ a>1 \end{cases}$$

a = nombre de crides recursives que s'efectuen

c = de quina forma es redueixen les dades

k = el cost de la resta d'operacions que es realitzen abans i després de la crida recursiva.

Exemple: La solució de l'equació de recurrència

$$T(1)=1$$

$$T(n)=T(n-1)+n \quad per\ a\ \ n\geqslant 2$$

és $T(n)=\Theta(n^2)$, doncs $a=1,\ c=1,\ k=1,\ n^{k+1}=n^2$

Decreixement geomètric

Teorema. *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n/b)+g(n) & si\ n_0\leqslant n \end{cases}$$

on n_0 és una constant, $b>1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb $k\geqslant 0$ constant. Aleshores,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<b^k \\ \Theta\left(n^k\log n\right) & si\ a=b^k \\ \Theta\left(n^{\log_b a}\right) & si\ a>b^k \end{cases}$$

a = nombre de crides recursives que s'efectuen

b = de quina forma es redueixen les dades

k = el cost de la resta d'operacions que es realitzen abans i després de la crida recursiva.

Exemple: La solució de l'equació de recurrència

$$T(1)=\Theta(1)$$

$$T(n)=2\,T(n/2)+n \qquad per\ a\ \ n\geqslant 2$$

és $T(n)=\Theta(n\log n)$, doncs $2=a=b^k=2^1$

2.4. ANÀLISI ASIMPTÒTICA DE L'EFICIÈNCIA ESPACIAL D'UN ALGORISME

Per a calcular l'espai de memòria que ocupa un algorisme utilitzarem el següent:

1. Espai que ocupa un objecte (de tipus predefinit o bé escalar):

   Constant.

   $$E(\text{tipus}) = \Theta(1)$$

2. Espai que ocupa una **taula** (vector):

   Producte de l'espai que ocupa cada component per la dimensió de la taula.

   $$E(\text{tipus}[N]) = N \cdot E(\text{tipus})$$

3. Espai que ocupa una **tupla** (**struct** o **class** en C++):

   Suma de l'espai que ocupen els camps de la tupla.

   $$E(\textbf{tupla}\ \{\ \text{tipus}_1\ c_1;\ ...;\ \text{tipus}_n\ c_n\ \}) = \text{màxim}\ (E(\text{tipus}_1),\ ...,\ E(\text{tipus}_n))$$

TEMA 3. ESTRUCTURES LINIALS ESTÀTIQUES

3.1. CONCEPTE DE SEQÜÈNCIA

- Les estructures lineals són aquelles que implementen les seqüències d'elements.

- **Definició**: Sobre un conjunt de base V (enters, caràcter, ...) podem definir les seqüències d'elements de V (V^*) de manera recursiva:

  - $\lambda \in V^*$ (seqüència buida)

  - $\forall v \in V,\ s \in V^* \ :\ vs \in V^*$

- Les **operacions** que ens interessa tenir sobre seqüències són:

  - Crear la seqüència buida.

  - Inserir un element dins d'una seqüència.

  - Esborrar un element de la seqüència.

  - Obtenir un element de la seqüència.

  - Decidir si una seqüència és buida o no.

- **Comportament** d'una seqüència: per definir el comportament d'una seqüència cal determinar:

  - A quina posició s'insereix un element nou.

  - Quin element de la seqüència s'esborra o s'obté.

- Tres tipus bàsics d'estructures lineals:

  - piles.
  - cues.
  - llistes.

- Per a cadascuna d'aquestes estructures veurem:

  - Descripció intuïtiva del comportament.

  - Algun exemple d'ús.

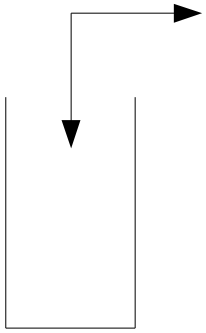
  - Especificació de la classe.

  - Implementació de la classe (una o més).

3.2. ESPECIFICACIÓ DE LES PILES

3.2.1. Concepte de Pila

- Les **piles** (anglès: *stack*) són estructures de dades que implementen les seqüències sota filosofia **LIFO** (Last-In, First-Out), l'últim que entra és el primer que surt.



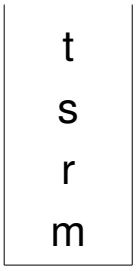
  - ✔ Els elements s'insereixen d'un en un.
  - ✔ Es treuen en l'ordre invers al qual s'han inserit.
  - ✔ L'únic element que es pot obtenir és l'últim inserit.

3.2.2. Exemple d'ús

- Gestió de les adreces de retorn en les crides.

```
programa P        acció A1        acció A2        acció A3
  ...               ...             ...             ...
 A1;               A2;             A3;             facció
r:                s:              t:
  ...               ...             ...
fprograma         facció          facció
```

En un punt d'execució d'*A3* es disposen els següents punts de retorn dels procediments dins d'una pila. *m* és l'adreça a la qual el programa ha de retornar el control.



- Altres exemples d'ús:
  - ↙ Transformació de programes recursius en iteratius
  - ↙ Implementació a baix nivell de funcions i accions: Pas de paràmetres, variables locals, ...

3.2.3. Operacions

- Les **operacions bàsiques** que es realitzen habitualment sobre una pila són:
  - Crear la pila buida (constructor).
  - Afegir un element (apilar).
  - Treure un element (desapilar).
  - Consultar un element (cim).
  - Decidir si la pila és buida o no (es_buida).

-

REGLA DELS TRES GRANS

La regla del tres grans (anglès: *the Law of the Big Three*) indica que si es necessita una implementació no trivial del: constructor per còpia, destructor o operador d'assignació caldrà implementar els altres dos mètodes.

Aquests tres mètodes són automàticament creats pel compilador (implementació d'ofici) si no són explícitament declarats pel programador.

SEMPRE que una classe empri memòria dinàmica caldrà implementar aquests tres mètodes, ja que les implementacions d'ofici amb punters no funcionen com nosaltres voldríem.

Així doncs, per ser més flexibles, en les nostres especificacions sempre apareixeran aquests tres mètodes.

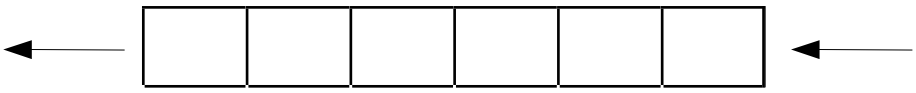
3.2.4. Especificació

```
template <typename T>
class pila {
public:
  Construeix una pila buida.
  pila() throw(error);

  Tres grans: constructora per còpia, operador d'assignació i
  destructora.
  pila(const pila<T> &p) throw(error);
  pila<T>& operator=(const pila<T> &p) throw(error);
  ~pila() throw();

  Afegeix un element a dalt de tot de la pila.
  void apilar(const T &x) throw(error);

  Treu el primer element de la pila. Llança un error si la pila
  és buida.
  void desapilar() throw(error);

  Obté l'element cim de la pila. Llança una excepció si la pila
  és buida.
  const T& cim() const throw(error);

  Consulta si la pila és buida o no.
  bool es_buida() const throw();

  // Altres operacions útils
  Crea una nova pila amb el resultat d'apilar x sobre la pila
  actual.
  pila<T> operator&(const T &x) const throw(error);

  Crea una nova pila amb la resta d'elements (els que estan per
  sota del cim).
  pila<T> resta() const throw(error);

  Gestió d'errors.
  static const int PilaBuida = 300;
};
```

3.3. ESPECIFICACIÓ DE LES CUES

3.3.1. Concepte de Cua

- La diferència entre les piles i les **cues** (anglès: *queue*) és que en aquestes últimes els elements de base s'insereixen per un extrem de la seqüència i s'extreuen per l'altre (política FIFO: First-In, First-Out: El primer element que entra a la cua és el primer en sortir). L'element que es pot consultar en tot moment és el primer inserit.



  - ↙ Els elements s'insereixen d'un en un.
  - ↙ Es treuen amb el mateix ordre en què s'han inserit.
  - ↙ L'únic element que es pot obtenir és el primer inserit.

3.3.2. Exemple d'ús

- Assignació de la CPU a processos d'usuaris pel sistema operatiu. Si suposem que el sistema és just, els processos que demanen el processador s'encuen, de manera que, quan el que s'està executant actualment acaba, passa a executar-se el que porta més temps esperant.
- Altres **usos** de les cues:
  - ↙ Recorregut per nivells d'arbres i en amplada de grafs.
  - ↙ Simulació (per exemple flux de vehicles en un peatge).

- Cues d'impressió, de treballs en batch, de missatges, d'esdeveniments, ...
- Buffers de disc, de teclat, ...

3.3.3. Operacions

- Les **operacions** bàsiques que es realitzen habitualment sobre una cua són:

  - Crear la cua buida (constructora).
  - Inserir un element (encuar).
  - Extreure un element (desencuar).
  - Consultar un element (primer).
  - Decidir si la cua està buida o no (es_buida).

PROGRAMACIÓ GENÈRICA

La **programació genèrica** és un idea molt útil perquè permet aplicar el mateix algorisme a tipus de dades diferents. Aquest mecanisme ens ajuda a:

* separar els algorismes dels tipus de dades.
* augmentar la modularitat dels programes.
* minimitzar la duplicació de codi.

En C++ la *genericitat* s'aconsegueix mitjançant l'ús de **Plantilles** (anglès: *templates*) tal com hem vist al tema 1.

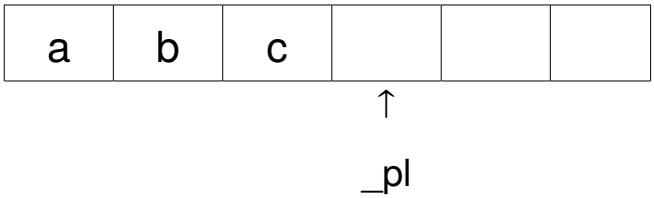
3.3.4. Especificació

```
template <typename T>
class cua {
public:
  Construeix una cua buida.
  cua() throw(error);

  Tres grans: constructora per còpia, operador d'assignació i
  destructora.
  cua(const cua<T> &c) throw(error);
  cua<T>& operator=(const cua<T> &c) throw(error);
  ~cua() throw();

  Afegeix un element al final de la cua.
  void encuar(const T &x) throw(error);

  Treu el primer element de la cua. Llança un error si la cua és
  buida.
  void desencuar() throw(error);

  Obté el primer element de la cua. Llança un error si la cua és
  buida.
  const T& primer() const throw(error);

  Consulta si la cua és buida o no.
  bool es_buida() const throw();

  // Altres operacions útils
  Crea una nova cua amb el resultat d'encuar x sobre la cua
  actual.
  cua<T> operator&(const T &x) const throw(error);

  Crea una nova cua amb la resta d'elements (els que estan
  després del primer).
  cua<T> resta() const throw(error);

  Gestió d'errors.
  static const int CuaBuida = 310;
};
```

3.4. IMPLEMENTACIÓ DE PILES I CUES

3.4.1. Decisions sobre la representació de les piles

- Els elements de les piles s'hauran d'emmagatzemar en alguna estructura de dades, concretament dins d'un vector. Introduirem en la representació de la pila un enter que faci el paper d'apuntador a la primera posició buida del vector.



- La implementació de la pila amb un vector fa que la pila ja no sigui de dimensió infinita. Hem de modificar l'especificació afegint un control d'error en intentar afegir un element a la pila plena. Es disposa d'una constant MAX que indica el màxim d'elements de la pila.
- La implementació tindrà un cost temporal òptim $\Theta(1)$ en totes les operacions.
- El cost espacial és pobre, perquè una pila té un espai reservat de *MAX*, independentment del nombre d'elements que la formen en un moment donat. A més, la política d'implementar una pila amb un vector ens obliga a determinar quin és el nombre d'elements que caben a la pila.

  Això es pot solucionar si s'implementa la pila com una llista enllaçada usant memòria dinàmica (veure la secció 4.3).

3.4.2. Representació de les piles

```
template <typename T>
class pila {
public:
  pila() throw(error);

  pila(const pila<T> &p) throw(error);
  pila<T>& operator=(const pila<T> &p) throw(error);
  ~pila() throw();

  void apilar(const T &x) throw(error);
  void desapilar() throw(error);

  const T& cim() const throw(error);
  bool es_buida() const throw();

  // Altres operacions útils
  pila<T> operator&(const T &x) const throw(error);
  pila<T> resta() const throw(error);

  static const int PilaBuida = 300;

  // A l'especificació que ja hem vist cal afegir
  // el següent:
  static const nat MAX = 100;

  Consulta si la pila és plena o no.
  bool es_plena() const throw();

  static const int PilaPlena = 301;


private:
  T _taula[MAX]; // [0..MAX-1]
  nat _pl;

  // Mètodes privats
  void copiar(const pila<T> &p) throw(error);
};
```

NOU

3.4.3. Implementació de les piles

FITXER D'IMPLEMENTACIÓ fitxer.t

Cal recordar que les classes genèriques en C++ (template) tenen la particularitat que la implementació no es posa en el fitxer .CPP sinó en un altre fitxer que anomenarem .T.

```
// Cost: Θ(1)
template <typename T>
pila<T>::pila() : _pl(0) throw(error) { }

// Cost: Θ(n)
template <typename T>
void pila<T>::copiar(const pila<T> &p) throw(error) {
  for (nat i=0; i < p._pl; ++i) {
    _taula[i] = p._taula[i];
  }
  _pl = p._pl;
}

// Cost: Θ(n)
template <typename T>
pila<T>::pila(const pila<T> &p) throw(error) {
  copiar(p);
}

// Cost: Θ(n)
template <typename T>
pila<T>& pila<T>::operator=(const pila<T> &p) throw(error)
{
  if (this != &p) {
    copiar(p);
  }
  return *this;
}
// Cost: Θ(1)
template <typename T>
pila<T>::~pila() throw() { }
  // No es fa res ja que no s'usa memòria dinàmica.
```

```
// Cost: Θ(1)
template <typename T>
void pila<T>::apilar(const T &x) throw(error) {
  // Donat que la pila és finita cal comprovar abans que
  // hi hagi espai per posar el nou element.
  if (es_plena()) {
    throw error(PilaPlena);
  }
  _taula[_pl] = x;
  ++_pl;
}

// Cost: Θ(1)
template <typename T>
void pila<T>::desapilar() throw(error) {
  if (es_buida()) {
    throw error(PilaBuida);
  }
  --_pl;
}

// Cost: Θ(1)
template <typename T>
const T& pila<T>::cim() const throw(error) {
  if (es_buida()) {
    throw error(PilaBuida);
  }
  return _taula[_pl-1];
}

// Cost: Θ(1)
template <typename T>
bool pila<T>::es_buida() const throw() {
  return _pl == 0;
}



// Cost: Θ(1)
template <typename T>
bool pila<T>::es_plena() const throw() {
  return _pl == MAX;
}
```

```
// Cost: Θ(n)
template <typename T>
pila<T> pila<T>::operator&(const T &x) const throw() {
  pila<T> p(*this); // constructor per còpia
  p.apilar(x);
  return p;
}

// Cost: Θ(n)
template <typename T>
pila<T> pila<T>::resta() const throw() {
  pila<T> p(*this); // constructor per còpia
  p.desapilar();
  return p;
}
```

3.4.4. Enriquiments i modificacions

- Suposem que estem escrivint un programa que usa el tipus pila i que necessita molt freqüentment una funció, denominada fondària, que compta el nombre d'elements que hi ha en una pila.

SOLUCIÓ 1: Definim un enriquiment del tipus pila amb aquesta nova operació. Treballant des de fora de la classe no podem accedir a la seva representació, sinó que cal manipular-lo usant les operacions existents:

```
template <typename T>
class pila_fondaria : public pila<T> {
public:
  nat fondaria() const throw();
};
```

```
// Implementació

// Cost: Θ(n)
template <typename T>
nat pila_fondaria<T>::fondaria() const throw() {
  pila<T> p(*this);  // constructor per còpia
  nat cnt = 0;
  while (not p.es_buida()) {
    ++cnt;
    p.desapila();
  }
  return cnt;
}
```

- **Inconvenient**: El problema d'aquesta solució és obvi: el cost temporal. Per a una pila de *n* elements, el cost temporal de l'operació fondària és Θ(n).

SOLUCIÓ 2: Introduir la funció fondària dins de la classe pila. Llavors podem manipular la representació del tipus i en conseqüència el cost temporal de la funció baixa a Θ(1):

```
template <typename T>
class pila {
public:
  ...
  nat fondaria() const throw();
  ...
};


// Implementació

// Cost: Θ(1)
template <typename T>
nat pila<T>::fondaria() const throw() {
  return _pl;
}
```

- **Inconvenients**:

  - Introduïm una operació molt particular en la definició d'un tipus d'interès general, la qual cosa pot ser bona o no.

  - La modificació de la classe pot ser problemàtica. Si intentem substituir la versió de la classe per la nova, podem introduir inadvertidament algun error.

En resum, tenim dos **enfocaments bàsics** quan definim una classe:

> * Pensar en les classes com un proveïment de les operacions **indispensables** per construir-ne de més complicades en classes d'enriquiment.
>
> * Posar a la classe **totes** aquelles operacions que se'ns acudeixin, buscant més eficiència a la implementació.

Decisions sobre la representació de les cues

- Els elements de la cua es distribueixen en un vector amb uns quants apuntadors:

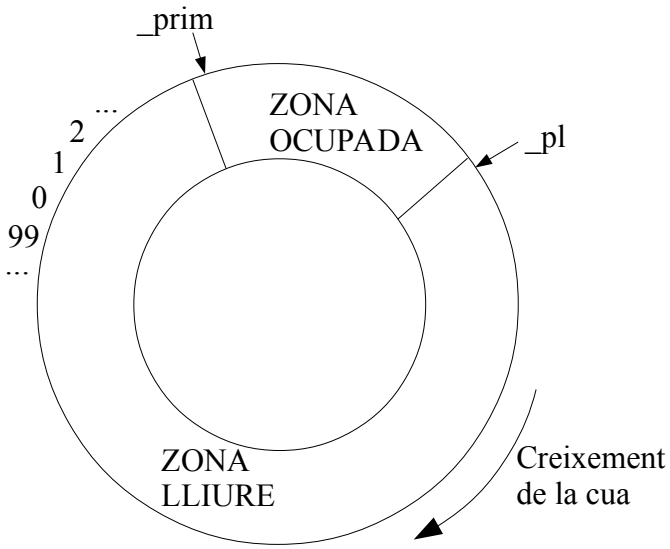
  1) Apuntador de lloc lliure, *_pl*, a la posició on inserir l'element següent.

2) Apuntador al primer element de la cua, *_prim*.

El problema apareix en desencuar elements i moure l'apuntador *_prim*: es perd aquest espai alliberat del vector ja que no es possible tornar-lo a aprofitar.

Si decidim que el primer element de la cua ocupi sempre la primera posició del vector, cada vegada que es desencua hem de moure tots els elements una posició (ineficient).

SOLUCIÓ: Un apuntador al primer element que es mogui cada vegada que desencuem un element. I per reaprofitar les posicions inicials lliures del vector, considerarem el vector com una estructura circular, on després del darrer element del vector ve el primer: *_pl = (_pl+1) % MAX*



3) Necessitem un indicador de si la cua està plena o buida, ja que fins ara la condició d'una cua buida és la mateixa que la condició de cua plena:

$$\_prim == \_pl$$

Per això hem d'introduir un booleà addicional o bé tenir un comptador d'elements a la cua (*_cnt*). Aquesta última opció és especialment útil si volem implementar una operació que ens digui el nombre d'elements d'una cua dins del classe cua.

- La implementació tindrà un cost temporal $\Theta(1)$ en totes les operacions.
- Es disposa d'una constant MAX que indica el màxim d'elements de la pila.
- El cost espacial és pobre, perquè una cua té un espai reservat de MAX, independentment del nombre d'elements que la formen en un moment donat. A més, la política d'implementar una cua amb un vector ens obliga a determinar quin és el nombre d'elements que caben a la cua.

  Això es pot solucionar si s'implementa la cua com una llista enllaçada usant memòria dinàmica (veure la secció 4.3).

3.4.5. Representació de la classe cua

```
template <typename T>
class cua {
public:
  cua() throw(error);

  cua(const cua<T> &c) throw(error);
  cua<T>& operator=(const cua<T> &c) throw(error);
  ~cua() throw(error);

  void encuar(const T &x) throw(error);
  void desencuar() throw(error);

  const T& primer() const throw(error);
  bool es_buida() const throw();

  // Altres operacions útils
  cua<T> operator&(const T &x) const throw(error);
  cua<T> resta() const throw(error);

  static const int CuaBuida = 310

  // A l'especificació que ja hem vist cal afegir
  // el següent:
  static const nat MAX = 100;

  Consulta si la cua és plena o no.
  bool es_plena() const throw();

  static const int CuaPlena = 311;

private:
  T _taula[MAX]; // [0..MAX-1]
  nat _prim, _pl, _cnt;

  // Mètodes privats
  void copiar(const cua<T> &c) throw(error);
};
```

NOU

3.4.6. Implementació de la cua

Per no escriure massa codi només implementarem els mètodes bàsics de la classe.

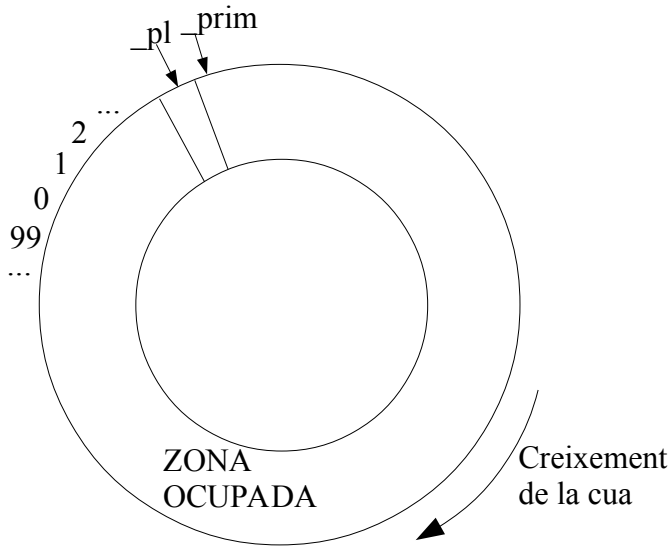
```
// Cost: Θ(1)
template <typename T>
cua<T>::cua() : _prim(0), _pl(0), _cnt(0) throw(error) {
}

// Cost: Θ(n)
template <typename T>
void cua<T>::copiar(const cua<T> &c) throw(error) {
  for (nat i=0; i < MAX; ++i) {
    _taula[i] = c._taula[i];
  }
  _prim = c._prim;
  _pl = c._pl;
  _cnt = c._cnt;
}

// Cost: Θ(n)
template <typename T>
cua<T>::cua(const cua<T> &c) throw(error) {
  copiar(c);
}

// Cost: Θ(n)
template <typename T>
cua<T>& cua<T>::operator=(const cua<T> &c) throw(error) {
  if (this != &c) {
    copiar(c);
  }
  return *this;
}

// Cost: Θ(1)
template <typename T>
cua<T>::~cua() throw() {
  // No es fa res ja que no s'usa memòria dinàmica.
}
```

```
// Cost: Θ(1)
template <typename T>
void cua<T>::encuar(const T &x) throw(error) {
  if (es_plena()) {
    throw error(CuaPlena);
  }
  _taula[_pl] = x;
  _pl = (_pl+1) % MAX;
  ++_cnt;
}

// Cost: Θ(1)
template <typename T>
void cua<T>::desencuar() throw(error) {
  if (es_buida()) {
    throw error(CuaBuida);
  }
  _prim = (_prim+1) % MAX;
  --_cnt;
}

// Cost: Θ(1)
template <typename T>
const T& cua<T>::primer() const throw(error) {
  if (es_buida()) {
    throw error(CuaBuida);
  }
  return _taula[_prim];
}

// Cost: Θ(1)
template <typename T>
bool cua<T>::es_buida() const throw() {
  return _cnt == 0;
}

// Cost: Θ(1)
template <typename T>
bool cua<T>::es_plena() const throw() {
  return _cnt == MAX;
}
```

Dues variants habituals d'aquesta representació són:

1.Podem estalviar-nos el comptador desaprofitant una posició del vector, de manera que la condició de cua plena passa a ser que *_pl* apunti a la posició anterior a la qual apunta *_prim*, mentre que la condició de cua buida continua essent la mateixa.

Exemple de cua plena:



2.Podem estalviar-nos l'apuntador *_pl* substituint-ne qualsevol referència per l'expressió:

(_prim+_cnt) % MAX.

Si sabem on comença la cua i quants elements hi ha, sabem on és la posició lliure de la cua.

3.4.7. Implementació de diverses piles

Què passa si implementem les dues piles per separat?

- aprofitem totes les operacions de la classe pila, però
- podem arribar a desaprofitar molt d'espai, si s'omple una pila totalment i l'altra queda buida.

SOLUCIÓ 1: Fer una representació conjunta de les dues piles. Això implica definir una nova classe **dues_piles** que repeteix l'especificació de les piles però distingint la pila de treball (per exemple amb un natural).

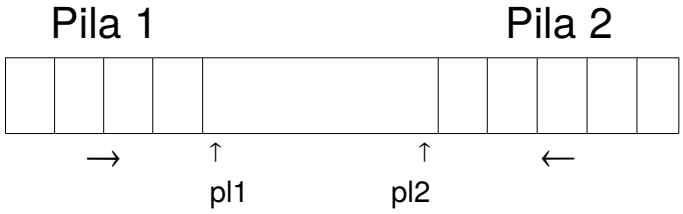
En les següents especificacions no s'inclou el comportament dels mètodes per major brevetat del codi.

```
template <typename T>
class dues_piles {
public:
  dues_piles() throw(error);

  dues_piles(const dues_piles<T> &p) throw(error);
  dues_piles<T>& operator=(const dues_piles<T> &p)
    throw(error);
  ~dues_piles() throw();

  void apilar(nat p, const T &x) throw(error);
  void desapilar(nat p) throw(error);

  const T& cim(nat p) const throw(error);
  bool es_buida(nat p) const throw();

};
```

SOLUCIÓ 2: Una altra opció és duplicar les operacions. La primera solució és més compacta.

```
template <typename T>
class dues_piles {
public:
  dues_piles() throw(error);

  dues_piles(const dues_piles<T> &p) throw(error);
  dues_piles<T>& operator=(const dues_piles<T> &p)
    throw(error);
  ~dues_piles() throw();

  void apilar1(const T &x) throw(error);
  void apilar2(const T &x) throw(error);
  void desapilar1() throw(error);
  void desapilar2() throw(error);

  const T& cim1() const throw(error);
  const T& cim2() const throw(error);
  bool es_buida1() const throw();
  bool es_buida2() const throw();

};
```

Degut a que només hi ha canvis en un extrem de la pila (el cim), podem implementar amb facilitat dues piles en un únic vector si les col·loquem enfrontades, de manera que creixin en sentit contrari. Així una pila no tindrà espai si realment no hi ha espai lliure dins el vector.



3.5. LLISTES AMB PUNT D'INTERÈS

3.5.1. Concepte de llista

Definició: Les **llistes** (anglès: *list*) són la generalització dels dos tipus anteriors: mentre que en una pila i en una cua només s'hi pot consultar (inserir, esborrar i consultar) un element d'una posició determinada, en una llista s'hi pot:

- Inserir en qualsevol posició.

- Esborrar-ne qualsevol element.

- Consultar-ne qualsevol element.

Anomenem **longitud** al número d'elements d'una llista. Considerarem llistes **finites** (de longitud finita) i **homogènies** (constituïdes per elements del mateix tipus).

3.5.2. Especificació de les llistes amb punt d'interès

- Es defineix l'existència d'un element distingit dins la seqüència. Aquest és el que serveix de referència per a les operacions.
- Sempre distingirem el tros de seqüència a l'esquerra del punt d'interès i el tros a la dreta.

```
template <typename T>
class llista_pi {
public:
```

```
Crea una llista buida amb el punt d'interès indefinit.
```

```
llista_pi() throw(error);
```

```
Tres grans.
```

```
llista_pi(const llista_pi<T> &l) throw(error);
```

```
  llista_pi<T>& operator=(const llista_pi<T> &l)
    throw(error);
  ~llista_pi() throw();

  Insereix l'element x darrera de l'element apuntat pel PI; si el
  PI és indefinit, insereix x com primer element de la llista.
  Genera una excepció si la llista està plena.
  void inserir(const T &x) throw(error);

  Elimina l'element apuntat pel PI; no fa res si el PI és
  indefinit; el PI passa a apuntar al successor de l'element
  eliminat o queda indefinit si l'element eliminat no tenia
  successor.
  void esborrar() throw();

  Situa el PI en el primer element de la llista o queda indefinit
  si la llista és buida.
  void principi() throw();

  Mou el PI al successor de l'element apuntat pel PI, quedant el
  PI indefinit si apuntava a l'últim de la llista; no fa res si
  el PI estava indefinit.
  void avancar() throw();

  Retorna l'element apuntat pel PI; llança una excepció si el PI
  estava indefinit.
  const T& actual() const throw(error);

  Retorna cert si i només si el PI està indefinit (apunta a
  l'element "final" fictici).
  bool final() const throw();

  Retorna la longitud de la llista.
  nat longitud() const throw();

  Retorna cert si només si la llista és buida.
  bool es_buida() const throw();

  Gestió d'errors.
  static const int PIIndef = 320
};
```

Existeixen dues estratègies diferents per a concebre la classe llista, pensant en si volem dissenyar la marca (el que ens permet recórrer la llista) de manera interna o externa.

- **Interna**: Tenim una única classe "***llista amb punt d'interès***": La classe guarda els elements de la llista i disposa d'una única marca que apunta a l'element d'interès. És la que s'ha utilitzat en l'especificació anterior.

- **Externa**: Aquesta implementació ofereix dues classes: llista i el seu ***iterador***. Els iteradors són una marca que apunta a un element de la llista però són independents de la llista.
  És una solució més flexible doncs podem disposar de diferents iteradors sobre una mateixa llista però també és més perillosa. Òbviament aquesta solució és la millor en la majoria dels casos i la biblioteca STL de C++ és la que utilitza.

3.5.3. Alguns algorismes sobre llistes amb punt d'interès

A) Recorregut de la llista tractant tots els seus elements:

```
L.principi();
while (not L.final()) {
     v = L.actual();
     tractar(v);
     L.avancar();
}
```

B) Localització d'un element v dins la llista:

```
L.principi();
bool trobat = false;
while (not L.final() and not trobat) {
  if (L.actual() == v) {
    trobat = true;
  }
  else {
    L.avancar();
  }
}
if (trobat) {
  // tractament per trobat
}
else {
  // tractament per no trobat
}
```

Podem escollir entre dues estratègies per implementar les llistes:

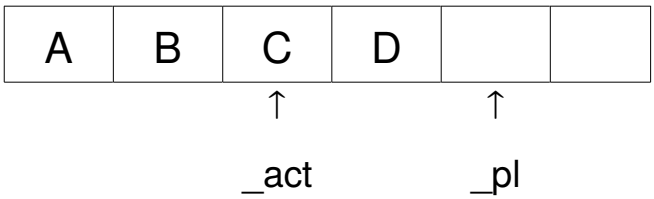
- **Representació seqüencial**: Els elements s'emmagatzemen dins d'un vector complint-se que elements consecutius a la llista ocupen posicions consecutives dins del vector. Veure 3.5.4
- **Representació encadenada**: S'introdueix el concepte d'encadenament. Tot element del vector identifica explícitament la posició que ocupa el seu successor a la llista. Veure 3.5.5

3.5.4. Representació seqüencial

- La filosofia és idèntica a la representació de les piles i les cues.
- Necessitem un apuntador addicional per denotar l'element distingit.

- La implementació de la llista amb un vector fa que la llista ja no sigui de dimensió infinita. Hem de modificar l'especificació afegint un control d'error en intentar afegir un element a la llista plena.
- Es disposa d'una constant MAX que indica el màxim d'elements de la pila.

Exemple: Representació de la llista <A B | C D>. El punt d'interès està situat sobre l'element C.



3.5.4.1. Representació de la classe llista seqüencial

```
template <typename T>
class llista_pi {
public:
  llista_pi() throw(error);

  llista_pi(const llista_pi<T> &l) throw(error);
  llista_pi<T>& operator=(const llista_pi<T> &l)
    throw(error);
  ~llista_pi() throw();

  void insereix(const T &x) throw(error);
  void esborra() throw();

  void principi() throw();
  void avanca() throw();

  const T& actual() const throw(error);
  bool final() const throw();
  nat longitud() const throw();
  bool es_buida() const throw();

  static const int PIIndef = 320;
```

```
  // A l'especificació que ja hem vist cal afegir
  // el següent:
  Nombre màxim d'elements.
  static const nat MAX = 100;

  Retorna cert si només si la llista és plena.
  bool es_plena() const throw();

  static const int LlistaPlena = 321

private:
  int _taula[MAX];
  nat _act, _pl;

  // Mètodes privats
  void copia(const llista_pi<T> &l) throw(error) {
};
```

NOU

3.5.4.2. Implementació de la classe llista seqüencial

```
// Cost: Θ(1)
template <typename T>
llista_pi<T>::llista_pi() : _act(0), _pl(0) throw(error)
{ }

// Mètode privat
// Cost: Θ(n)
template <typename T>
void llista_pi<T>::copia(const llista_pi<T> &l)
throw(error) {
  for (nat i=0; i < l._pl; ++i) {
    _taula[i] = l._taula[i];
  }
  _pl = l._pl;
  _act = l._act;
}
```

```
// Cost: Θ(n)
template <typename T>
llista_pi<T>::llista_pi(const llista_pi<T> &l)
throw(error) {
  copiar(l);
}

// Cost: Θ(n)
template <typename T>
llista_pi<T>& llista_pi<T>::operator=(const llista_pi<T>
&l) throw(error) {
  if (this != &l) {
    copiar(l);
  }
  return *this;
}

// Cost: Θ(1)
template <typename T>
llista_pi<T>::~llista_pi() throw() {
  // No es fa res ja que no s'usa memòria dinàmica.
}

// Cost: Θ(1)
template <typename T>
int llista_pi<T>::actual() const throw() {
  if (final()) {
    throw error(PIIndef);
  }
  return _taula[_act];
}

// Cost: Θ(n)
template <typename T>
void llista_pi<T>::esborra() throw() {
  if (not final()) {
    // desplacem els elements a la dreta del punt
    // d'interès un lloc cap a l'esquerra
    for (nat i=_act; i < _pl-1; ++i) {
      _taula[i] = _taula[i+1];
    }
    --_pl;
  }
}
```

```
// Cost: Θ(n)
template <typename T>
void llista_pi<T>::insereix(const T &x) throw(error) {
  if (es_plena()) {
    throw error(LlistaPlena);
  }
  if (final()) {
    // inserim l'element al principi de la llista
    for (nat i=_pl; i > 0; --i) {
      _taula[i] = _taula[i-1];
    }
    ++_pl;
    _taula[0] = x;
  }
  else {
    // desplacem els elements a la dreta del punt
    // d'interès un lloc cap a la dreta
    for (nat i=_pl; i > _act; --i) {
      _taula[i] = _taula[i-1];
    }
    ++_pl;
    _taula[_act] = x;
    ++_act;
  }
}

// Cost: Θ(1)
template <typename T>
void llista_pi<T>::principi() throw() {
  _act = 0;
}

// Cost: Θ(1)
template <typename T>
void llista_pi<T>::avanca() throw() {
  if (not final()) {
    ++_act;
  }
}
```

```
// Cost: Θ(1)
template <typename T>
nat llista_pi<T>::longitud() const throw() {
  return _pl;
}

// Cost: Θ(1)
template <typename T>
bool llista_pi<T>::final() const throw() {
  return _act == _pl;
}

// Cost: Θ(1)
bool llista_pi<T>::es_buida() const throw() {
  return _pl == 0;
}

// Cost: Θ(1)
bool llista_pi<T>::es_plena() const throw() {
  return _pl == MAX;
}
```

- **Inconvenient** de la representació seqüencial:

  Per inserir/esborrar a qualsevol posició del vector, hem de moure alguns elements a la dreta/esquerra i això resulta un cost lineal. Quan aquestes operacions són freqüents o la dimensió dels elements és molt gran, el cost lineal és inadmissible. La solució és usar la representació encadenada.

3.5.5. Representació encadenada (linked)

- Els elements consecutius de la llista ja no ocuparan posicions consecutives dins del vector.
- **Encadenament** (anglès: *link*): Una posició del vector que conté l'element v_k de la llista inclou un camp addicional que conté la posició que ocupa l'element v_{k+1}.
- El preu a pagar per estalviar moviments d'elements és simplement afegir un camp enter *enc* a cada posició del vector. Aquest enter indicarà la posició del vector on es troba el següent element. Hi posarem el valor -1 en el camp *enc* del darrer element per indicar que no hi ha cap més element.

```
struct node {
  int v;
  int enc;
};

node _taula[MAX-1];
int _act, _prim;
```

- **Gestió de l'espai lliure**: Cada vegada que inserim un element hem d'obtenir una posició del vector on emmagatzemar-lo i, en esborrar-lo, hem de recuperar aquesta posició com a reutilitzable.

  SOLUCIÓ:

  1. Marcar les posicions del vector com a ocupades o lliures.
  2. Considerar que els llocs lliures també formen una estructura lineal, sobre la qual disposem d'operacions per obtenir un element, esborrar-ne i inserir-ne (millor solució).

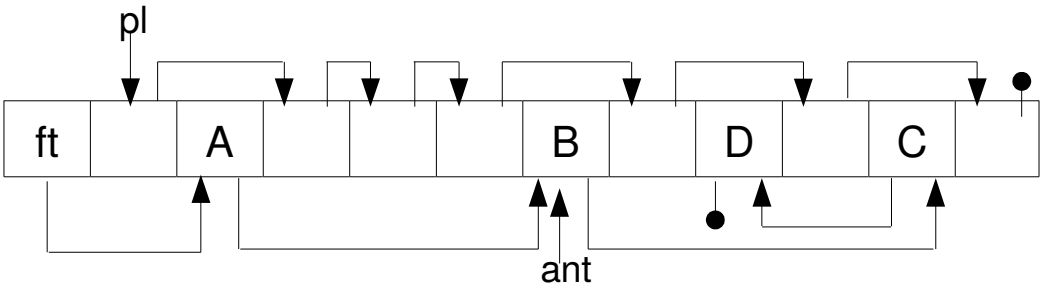
- **Modificacions dels encadenaments**: En inserir i esborrar l'element d'interès cal modificar l'encadenament de l'element anterior a ell perquè apunti a un altre.

  SOLUCIÓ: Consisteix a no tenir un apuntador a l'element actual, sinó a l'**anterior** a l'actual i així tant la inserció com la supressió queden amb cost constant.

- **Quin és l'element anterior a l'actual en una llista buida?**

  SOLUCIÓ: Utilitzar l'**element fantasma**. És un element fictici que sempre és a la llista, des de que es crea, de manera que la llista mai no està buida i l'element actual sempre té un predecessor. Aquest element ocupa sempre la mateixa posició (per exemple la posició 0). Desaprofitem un element del vector però els algorismes són més simples.

Exemple: Representació de la llista <A B | C D>. El punt d'interès està situat sobre l'element C.



TEMA 4. ESTRUCTURES LINEALS DINÀMIQUES

4.1. PROBLEMÀTICA A RESOLDRE

La implementació amb vectors té tres problemes:

- Es desaprofiten les posicions del vector que no estan emmagatzemant cap element.
- Necessitem predeterminar un màxim en la dimensió de l'estructura de dades.
- Els algorismes han d'ocupar-se de la gestió de l'espai lliure del vector.

4.2. IMPLEMENTACIÓ AMB PUNTERS (MEMÒRIA DINÀMICA)

La majoria de llenguatges comercials disposen d'un tipus de dades anomenat **punter** o apuntador (anglès: *pointer*). La notació de C++ és:

- Donat un tipus **T**, el tipus punter a **T** es defineix com **T***
- Una variable o objecte de tipus **T*** pot guardar l'adreça de memòria d'una variable o objecte de tipus **T**
- Un punter amb el valor especial de **NULL** no apunta enlloc.
- Donat un punter **p**, ***p** denota l'objecte apuntat per **p**.

- Molt sovint l'objecte que apunta un punter **p** és una tupla i volem consultar/modificar un dels camps de la tupla apuntada per **p**. És més còmode fer servir l'operador -> enlloc dels dos operadors * (objecte apuntat per un punter) i . (selector de camp).

  Exemple:

```
T *p, *q; // p i q són punters del tipus T
          // T és una tupla amb els camps info i seg
// En aquest punt caldria demanar memòria per
// guardar objectes de tipus T apuntats per p q
p->info = v;     // equival a fer   (*p).info = v
p->seg = q->seg; // assignació de punters a T
```

El tipus punter permet:

- Obtenir espai per guardar objectes d'un tipus determinat.
- Retornar aquest espai quan ja no necessitem més l'objecte.

Aquestes dues operacions es tradueixen en l'existència de dues primitives sobre el tipus punter:

- **new**(tipus): Donat un tipus, obté la memòria necessària per emmagatzemar un objecte d'aquest tipus i retorna un punter amb l'adreça de memòria on està l'objecte (*malloc* en C, *new* en Pascal).
- **delete**(punter)**:** Donat un punter, allibera la memòria usada per l'objecte que estava situat en l'adreça indicada pel punter (*free* en C, *dispose* en Pascal).

A més amb memòria dinàmica també podem crear taules dinàmiques. Per fer això utilitzarem les primitives **new** i **delete** però amb corxets ([ ]):

- **new** tipus [mida]: Donat un tipus, obté la memòria necessària per crear una taula d'una dimensió que tingui el nombre d'elements indicats. Retorna un punter amb l'adreça de memòria de la taula on està l'objecte.
- **delete[ ]** punter: Donat un punter, allibera la memòria usada per l'objecte que estava situat en l'adreça indicada pel punter.

Exemple:

```
struct T {
  int info;
  T* seg; // Es poden definir camps que siguin punters
          // al mateix tipus definit
};

T *p, *q;     // p i q són punters del tipus T
p = new(T);   // p apunta a un objecte de tipus T o NULL
              // si no queda memòria
q = new(T);
p->info = 5; // equival a fer   (*p).info = 5
q->info = 8;
p->seg = q;   // Hem enllaçat les dos tuples (llista que
              // conté dos elements 5, 8)
delete(q);    //  Eliminem  l'objecte  apuntat  per  q.  Ara
q=NULL;
q->info = 3; // Això provocaria un error, doncs q no
              // apunta a cap objecte
q = p;        // q i p apunten al mateix objecte
q->info = 3; // p->info també conté 3, doncs p i q
              // apunten al mateix objecte
delete(p);    // Eliminem l'objecte apuntat per p.
              // q apunta a un objecte inexistent
```

Les taules dinàmiques es comporten de la mateixa manera que les taules estàtiques però la mida es pot indicar en temps d'execució.

```
int n, *a = NULL;
cin >> n;
a = new int[n]; // Reserva n enters.
for (int i=0; i<n; ++i) {
   a[i] = 0;   // Inicialitza tots els elements a 0.
}
...            // Usar com un array normal.
delete[] a;    // S'allibera la taula.
```

La memòria reservada mitjançant la primitiva **new** s'allibera mitjançant la primitiva **delete**, mentre que la memòria reservada amb la primitiva **new[]** s'allibera mitjançant **delete[]**.

REGLA D'OR DE LA MEMÒRIA DINÀMICA

Tota la memòria que es reservi durant el programa s'ha d'alliberar abans de sortir del programa. No seguir aquesta regla és una actitud molt irresponsable, i en la majoria dels casos té conseqüències desastroses. No us fieu de que aquestes variables s'alliberen soles al acabar el programa, no sempre és veritat.

4.2.1. **Avantatges** de la implementació amb punters

- No cal predeterminar un nombre màxim d'elements a l'estructura.
- No cal gestionar l'espai lliure.
- A cada moment, l'espai gastat per la nostra estructura és estrictament el necessari llevat de l'espai requerit pels encadenaments.

4.2.2. **Desavantatges** de la implementació amb punters

- No és veritat que la memòria sigui infinita (En qualsevol moment durant l'execució es pot exhaurir l'espai, amb l'agreujant de no conèixer a priori la capacitat màxima de l'estructura).
- Els punters són referències directes a memòria (És possible que es modifiquin insospitadament dades i codi en altres punts del programa a causa d'errors algorísmics).
- Que el sistema operatiu faci tota la gestió de llocs lliures és còmode, però de vegades pot no interessar-nos (Pot ser ineficient quan per exemple volem eliminar l'estructura sencera).
- Diversos punters poden designar un mateix objecte (La modificació de l'objecte usant un punter determinat té com a efecte col·lateral la modificació de l'objecte apuntat per la resta de punters).
- És molt dificultós fer demostracions de correctesa.
- La depuració és molt més dificultosa (Perquè cal estudiar l'ocupació de la memòria).

- Alguns esquemes típics de llenguatges (assignació, entrada/sortida, ...) no funcionen de la manera esperada:

```
T *p, *q;
fstream f; // aquest fitxer emmagatzema elements
           // de tipus T.
...
/* Escriu al fitxer el valor del punter que és una
   adreça de memòria. No té cap sentit. */
f << p;
/* Escriu tot el que hi hagi dins de la posició de
  memòria apuntada per p, però també s'escriurà el
  camp seg de la tupla T que és una adreça de
  memòria. No tindrà sentit llegir posteriorment
  les dades del fitxer.*/
f << *p;
```

- **Referències penjades** (anglès: *dangling reference*): És una particularització de problema anterior. Si executem la seqüència d'instruccions següent:

```
nodes *p, *q;
p = new(node);
q = p;
delete(p); // q queda "penjat": el seu valor és
           // diferent de NULL però no apunta a cap
           // objecte vàlid
```

- **Retalls** (anglès: *garbage*): Problema simètric a l'anterior. Es produeix "brossa". Si executem la seqüència d'instruccions següent:

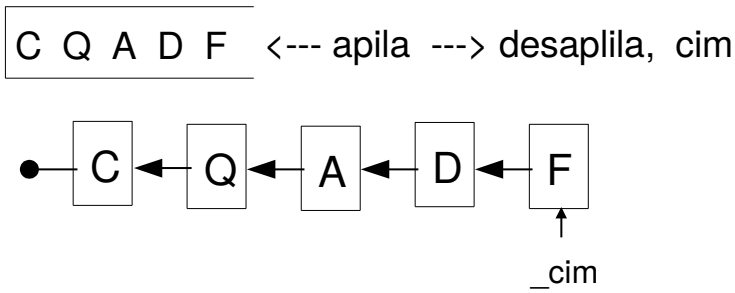
```
nodes *p, *q;
p = new(node);
q = new(node);
q = p; // l'objecte inicialment associat a q queda
       // inaccessible després de l'assignació
```

4.3. PILES I CUES IMPLEMENTADES AMB MEMÒRIA DINÀMICA

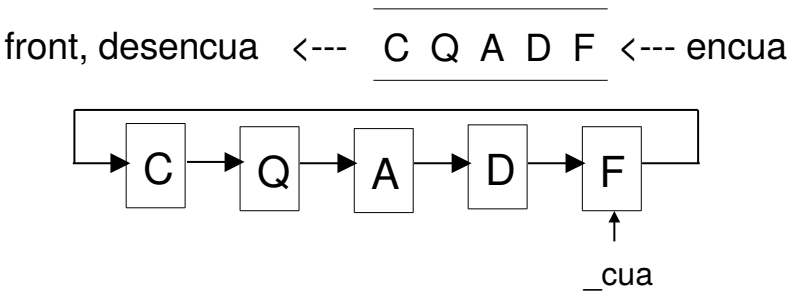
El problema de la mida prefixada de les piles i cues implementades en un vector es pot evitar amb una implementació enllaçada en memòria dinàmica.

Piles: Cada node contindrà un element i un punter al node apilat immediatament abans. L'element del fons de la pila apuntarà a NULL. La pila consistirà en un punter al node del cim de la pila (NULL si la pila està buida).

```
struct node {
    T info;
    node* seg;
};

node *_cim;
```



Cues: Cada node contindrà un element i un punter al node encuat immediatament després. Una possible solució és tancar l'estructura circularment de manera que l'últim element de la cua apunti al primer. Llavors la cua consistirà en un punter al node del darrer element encuat (NULL si la cua està buida).



4.3.1. Representació de la classe

```
template <typename T>
class cua {
public:
  ...

private:
  struct node {
    T info;
    node* seg;
  };
  node* _cua;

  node* copiar(node* n, node* fi, node* ini) throw(error);
};
```

4.3.2. Implementació de la classe

```
// Cost: Θ(1)
template <typename T>
cua<T>::cua() : _cua(NULL) throw(error) {
}

// Cost: Θ(n)
template <typename T>
cua<T>::cua(const cua<T> &c) : _cua(NULL) throw(error) {
  if (c._cua != NULL) {
    _cua = new node;
    try {
      _cua->info = c._cua->info;
      _cua->seg = copiar(c._cua->seg, c._cua, _cua);
    }
    catch (error) {
      delete(_cua);
      throw;
    }
  }
}
```

```
// Mètode privat
// Cost: Θ(n)
template <typename T>
typename cua<T>::node* cua<T>::copiar(node* n, node* fi,
node* ini) throw(error) {
  node* aux;
  if (n != fi) {
    aux = new node;
    try {
      aux->info = n->info;
      aux->seg = copiar(n->seg, fi, ini);
    }
    catch (error) {
      delete aux;
      throw;
    }
  }
  else {
    aux = ini;
  }
  return aux;
}

// Cost: Θ(n)
template <typename T>
cua<T>& cua<T>::operator=(const cua<T> &c)
throw(error) {
  if (this != &c) {
    cua<T> caux(c);
    node* naux = _cua;
    _cua = caux._cua;
    caux._cua = naux;
  }
  return *this;
}
```

```
// Cost: Θ(n)
template <typename T>
cua<T>::~cua() throw() {
  if (_cua != NULL) {
    node* fi = _cua;
    _cua = _cua->seg;
    while (_cua != fi) {
      node* aux = _cua;
      _cua = _cua->seg;
      delete aux;
    }
    delete(_cua);
  }
}


// Cost: Θ(1)
template <typename T>
void cua<T>::encuar(const T &x) throw(error) {
  node* p = new(node);
  try {
    p->info = x; // aquesta línia està entre try i catch
                 // donat que no sabem si assignar el
                 // tipus T pot generar un error (p.e si
  }              // T usés memòria dinàmica).
  catch (error) {
    delete p;
    throw;
  }
  if (_cua == NULL) {
    p->seg = p;  // cua amb un únic element que s'apunta
  }              // a sí mateix
  else {
    p->seg = _cua->seg;
    _cua->seg = p;
  }
  _cua = p;
}
```

```
// Cost: Θ(1)
template <typename T>
void cua<T>::desencuar() throw(error) {
  if (_cua==NULL) {
    throw error(CuaBuida);
  }
  node* p = _cua->seg;
  if (p == _cua) {
    _cua = NULL; // desencuem una cua que tenia un únic
  }              // element
  else {
    _cua->seg = p->seg;
  }
  delete(p);
}

// Cost: Θ(1)
template <typename T>
const T& cua<T>::primer() const throw(error) {
  if (_cua==NULL) {
    throw error(CuaBuida);
  }
  return (_cua->seg->info);
}

// Cost: Θ(1)
template <typename T>
bool cua<T>::es_buida() const throw() {
  return (_cua==NULL);
}
```

Totes les operacions de les piles i cues (exceptuant les tres grans) es poden implementar amb cost Θ(1). El cost espacial és proporcional al número d'elements de la pila/cua: Θ(n). L'inconvenient respecte la implementació en vector és que per cada element a guardar gastem l'espai d'un punter.

Donat que ara s'utilitza memòria dinàmica no cal tenir un mètode anomenat **es_plena** ja que la capacitat només ve determinada per la memòria del sistema.

4.4. LLISTES IMPLEMENTADES AMB MEMÒRIA DINÀMICA

Per implementar les llistes en memòria dinàmica utilitzarem les mateixes solucions de les llistes encadenades implementades en vector.

- Ara els encadenaments seran punters a memòria.

- La llista consistirà en una estructura encadenada de nodes. Cada node contindrà:

  - Un element.
  - Un apuntador al següent node (el que conté el següent element de la llista).

- La llista tindrà un node fictici a la capçalera (fantasma) i mantindrà dos punters addicionals: un apuntarà al fantasma i l'altre al node anterior al punt d'interès.

- Apuntem al node anterior al punt d'interès enlloc d'apuntar directament al punt d'interès per poder implementar l'operació *esborra()* (eliminació del punt d'interès) amb cost constant.

- Igual que en l'anterior especificació el punt d'interès és intern a la classe.

4.4.1. Especificació de la classe

```
template <typename T>
class llista_pi {
public:
```

```
Constructora per defecte. Crea una llista buida amb punt
d'interès indefinit.
llista_pi() throw(error);
```

```
Tres grans.
```

```
llista_pi(const llista_pi<T>& l) throw(error);
llista_pi<T>& operator=(const llista_pi<T>& l)
              throw(error);
~llista_pi() throw();
```

```
Insereix l'element x davant de l'element apuntat pel PI; si el PI és indefinit, insereix x com primer element de la llista.
```

```
void insereix(const T& x) throw(error);
```

```
Elimina l'element apuntat pel PI; no fa res si el PI és indefinit; el PI passa a apuntar al successor de l'element eliminat o queda indefinit si l'element eliminat no tenia successor.
```

```
void esborra() throw();
```

```
Longitud de la llista.
```

```
nat longitud() const throw();
```

```
Retorna cert si i només si la llista és buida.
```

```
bool es_buida() const throw();
```

```
Situa el PI en el primer element de la llista o queda indefinit si la llista està buida.
```

```
void principi() throw();
```

```
Mou el PI al successor de l'element apuntat pel PI, quedant el PI indefinit si apuntava a l'últim de la llista; no fa res si el PI estava indefinit.
```

```
void avanca() throw(error);
```

```
Retorna l'element apuntat pel PI; llança una excepció si el PI estava indefinit.
```

```
const T& actual() const throw(error);
```

```
Retorna cert si i només si el PI està indefinit (apunta a l'element "final" fictici).
```

```
bool final() const throw();

static const int PIIndef = 320;
```

```
private:
  struct node {
    T info;
    node* next;
  };
  node* _head;  // punter al fantasma
  node* _antpi; // punter al node anterior al punt
                // d'interès
  nat _sz;      // mida de la llista

  node* copia_llista(node* orig) throw(error);
  void destrueix_llista(node* p) throw();
  void swap(llista_pi<T>& l) throw();
};
```

4.4.2. Implementació de la classe

```
// Cost: Θ(1)
template <typename T>
llista_pi<T>::llista_pi() throw(error) {
  _head = new node;
  _head -> next = NULL;
  _antpi = _head;
  _sz = 0;
}

// Cost: Θ(n)
template <typename T>
typename llista_pi<T>::node* llista_pi<T>::copia_llista
               (node* orig) throw(error) {
  node* dst = NULL;
  if (orig != NULL) {
    dst = new node;
    try {
      dst -> info = orig -> info;
      dst -> next = copia_llista(orig -> next);
    } catch (const error& e) {
      delete dst;
      throw;
    }
  }
  return dst;
}
```

```
// Cost: Θ(n)
template <typename T>
llista_pi<T>::llista_pi(const llista_pi<T>& l)
               throw(error) {
  _head = copia_llista(l._head);
  _sz = l._sz;
  _antpi = _head;
  node* p = l._head;
  while (p != l._antpi) {
    _antpi = _antpi->next;
    p = p->next;
  }
}

// Cost: Θ(1)
template <typename T>
void llista_pi<T>::swap(llista_pi<T>& L) throw() {
  node* auxn = _head;
  _head = L._head;
  L._head = auxn;
  auxn = _antpi;
  _antpi = L._antpi;
  L._antpi = auxn;
  int auxs = _sz;
  _sz = L._sz;
  L._sz = auxs;
}
```

CONSTRUCTOR PER CÒPIA

Existeixen 2 formes de cridar el constructor per còpia:

Tipus b(a);

Tipus b = a;

La segona només es pot usar quan la constructora només té un sol paràmetre.

```
// Cost: Θ(n)
template <typename T>
llista_pi<T>& llista_pi<T>::operator=
               (const llista_pi<T>& l) throw(error) {
  if (this != &l) {
    llista_pi<T> aux = l;
    swap(aux);
  }
  return *this;
}
```

```
// Cost: Θ(n)
template <typename T>
void llista_pi<T>::destrueix_llista(node* p) throw() {
  if (p != NULL) {
    destrueix_llista(p->next);
    delete p;
  }
}


// Cost: Θ(n)
template <typename T>
llista_pi<T>::~llista_pi() throw() {
  destrueix_llista(_head);
}


// Cost: Θ(1)
template <typename T>
void llista_pi<T>::insereix(const T& x) throw(error) {
  node* nn = new node;
  try {
    nn->info = x;
  } catch(const error& e) {  // com que el tipus T és
    delete nn;               // desconegut no sabem si
    throw;                   // utilitza memòria dinàmica
  }
  if (_antpi->next != NULL) {
    nn->next = _antpi->next;
    _antpi->next = nn;
    _antpi = nn;
  }
  else {
    nn->next = _head->next;
    _head->next = nn;
  }
  ++_sz;
}
```

```
// Cost: Θ(1)
template <typename T>
void llista_pi<T>::esborra() throw() {
  if (_antpi->next != NULL) {
    node* todel = _antpi->next;
    _antpi->next = todel->next;
    delete todel;
    --_sz;
  }
}

// Cost: Θ(1)
template <typename T>
nat llista_pi<T>::longitud() const throw() {
  return _sz;
}

// Cost: Θ(1)
template <typename T>
bool llista_pi<T>::es_buida() const throw() {
  return _head->next == NULL;
  // equival a: return sz == 0
}

// Cost: Θ(1)
template <typename T>
void llista_pi<T>::principi() throw() {
  _antpi = _head;
}

// Cost: Θ(1)
template <typename T>
void llista_pi<T>::avanca() throw(error) {
  if (_antpi->next != NULL) {
    _antpi = _antpi->next;
  }
}
```

```
// Cost: Θ(1)
template <typename T>
const T& llista_pi<T>::actual() const throw(error) {
  if (_antpi->next == NULL) {
    throw error(PIIndef);
  }
  return _antpi->next->info;
}

// Cost: Θ(1)
template <typename T>
bool llista_pi<T>::final() const throw() {
  return _antpi->next == NULL;
}
```

Si volem disposar d'operacions per inserir elements al final de la llista o consultar el darrer element de la llista seria convenient disposar d'un punter a l'últim element.

Per implementar de forma eficient una operació que esborri el darrer element de la llista no serveix la solució anterior. Seria necessari disposar de llistes doblement encadenades (veure la secció 4.5).

El cost en espai d'una llista de n elements és $(s_{elem}+s_p)*n + 2s_p$, on s_{elem} és la mida d'un objecte de tipus *elem* i s_p és l'espai d'un punter (típicament 4 bytes).

4.5. ALGUNES VARIANTS DE LA IMPLEMENTACIÓ DE LLISTES

4.5.1. Llistes circulars

En les llistes circulars (anglès: *circular list*) l'últim element de la llista s'encadena amb el primer. És útil en les següents situacions:

- Volem tenir els elements encadenats sense distingir quin és el primer i l'últim, o que el paper de primer element vagi canviant dinàmicament.
- Alguns algorismes poden quedar simplificats si no es distingeix l'últim element.
- Des de tot element pot accedir-se a qualsevol altre.
- Permet que la representació de cues amb encadenaments només necessiti un apuntador a l'últim element, perquè el primer sempre serà l'apuntat per l'últim.

En aquest mateix apartat veurem un exemple de classe llista implementada amb memòria dinàmica mitjançant una llista circular.

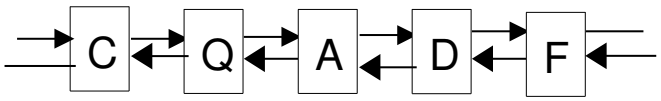
4.5.2. Llistes doblement encadenades (double-linked list)

Suposem que enriquim classe llista amb noves operacions:

- **inserir últim** un nou element al final de la llista.
- **esborrar últim** element de la llista.
- retrocedir el punt d'interès (situar-lo en el punt **previ**).
- situar el punt d'interès al **final** de la llista.

- Les llistes simplement encadenades no són adequades per implementar les operacions *esborra_ultim* i *previ*: donat un element, per saber quin és l'anterior cal recórrer la llista des del començament i resulta un cost lineal.

- Cal modificar la representació per obtenir llistes doblement encadenades.



- Per simplificar els algorismes d'inserció i de supressió s'afegeix a l'inici un element fantasma, per tal que la llista no estigui mai buida. A més és aconsellable implementar la llista circularment.

- Ara ja no cal tenir un punter a l'element anterior del punt d'interès.

En aquest cas la **classe llista** s'implementa:

* **doblement enllaçada**,
* **circular**
* **amb element fantasma**
* punt d'interès extern: **iteradors**

4.5.3. Especificació i representació de la classe

```
template <typename T>
class llista_itr {
private:
  struct node {
    T info;
    node* next;
    node* prev;
  };
  node* _head; // punter al fantasma
  nat _sz;     // mida de la llista

  node* copiar_llista(node* orig, node* orighead,
        node* h) throw(error);
  void destruir_llista(node* p, node* h) throw();
  void swap(llista_itr<T>& l) throw();

public:
  Constructora. Crea una llista buida.
  llista_itr() throw(error);

  Tres grans.
  llista_itr(const llista_itr<T>& l) throw(error);
  llista_itr<T>& operator=(const llista_itr<T>& l)
         throw(error);
  ~llista_itr() throw();


  Iteradors sobre llistes.
  Un objecte iterador sempre està associat a una llista
  particular; només poden ser creats mitjançant els mètodes
  Llista<T>::primer, Llista<T>::ultim i Llista<T>::indef. Cada
  llista té el seu iterador indefinit propi: L1.indef() !=
  L2.indef()
  friend class iterador;

  class iterador {
  public:
    friend class llista_itr;

    Per la classe iterador NO cal redefinir els tres grans ja que
    ens serveixen les implementacions d'ofici.
```

```
  Accedeix a l'element apuntat per l'iterador o llança una
  excepció si l'iterador és indefinit.
  const T& operator*() const throw(error);

  Operadors per avançar (pre i postincrement) i per retrocedir
  (pre i postdecrement); no fan res si l'iterador és indefinit.
  iterador& operator++() throw();
  iterador operator++(int) throw();
  iterador& operator--() throw();
  iterador operator--(int) throw();

  Operadors d'igualtat i desigualtat entre iteradors.
  bool operator==(iterador it) const throw();
  bool operator!=(iterador it) const throw();

  static const int IteradorIndef = 330;

private:
  Constructora. Crea un iterador indefinit.
  No pot usar-se fora de la classe iterador o de la classe
  llista_itr.
  iterador() throw();

  node* _p;  // punter al node actual
  node* _h;  // punter al fantasma de la llista per
             // poder saber quan ho hem recorregut tot.
};

Insereix l'element x darrera/davant de l'element apuntat per
l'iterador; si l'iterador it és indefinit, insereix_darrera
afegeix x com primer de la llista, i insereix_davant afegeix x
com últim de la llista; en abstracte un iterador indefinit
"apunta" a un element fictici que és al mateix temps
predecessor del primer i successor de l'últim.
void inserir_darrera(const T& x, iterador it)
                     throw(error);
void inserir_davant(const T& x, iterador it)
                     throw(error);
```

```
  Tant esborra_avnc com esborra_darr eliminen l'element apuntat
  per l'iterador, menys en el cas que it és indefinit (llavors no
  fan res); amb esborra_avnc l'iterador passa a apuntar al
  successor de l'element eliminat o queda indefinit si l'element
  eliminat és l'últim; amb esborra_darr l'iterador passa a
  apuntar al predecessor de l'element eliminat o queda indefinit
  si l'eliminat és el primer element.
  void esborrar_avnc(iterador& it) throw();
  void esborrar_darr(iterador& it) throw();

  Longitud de la llista.
  nat longitud() const throw();

  Retorna cert si i només si la llista és buida.
  bool es_buida() const throw();

  Retorna un iterador al primer/últim element o un iterador
  indefinit si la llista és buida.
  iterador primer() const throw();
  iterador ultim() const throw();

  Retorna un iterador indefinit.
  iterador indef() const throw();
};
```

4.5.4. Implementació de la classe

```
// Cost: Θ(1)
template <typename T>
llista_itr<T>::llista_itr() throw(error) {
  _head = new node;
  _head->next = _head->prev = _head;
  _sz = 0;
}
```

```
// Cost: Θ(n)
template <typename T>
llista_itr<T>::llista_itr(const llista_itr<T> &l)
throw(error) {
  _head = new node;
  _head->next = copiar_llista(l._head->next,
                              l._head, _head);
  _head->next->prev = _head;
  _sz = l._sz;
}

// Cost: Θ(n)
template <typename T>
typename llista_itr<T>::node*
llista_itr<T>::copiar_llista(node* orig,
             node* orig_head, node* h) throw(error) {
  node* dst = h;
  if (orig != orig_head) {
    dst = new node;
    try {
      dst->info = orig->info;
      dst->next = copiar_llista(orig->next,
                                orig_head, h);
      dst -> next -> prev = dst;
    }
    catch (const error& e) {
      delete dst;
      throw;
    }
  }
  return dst;
}

// Cost: Θ(n)
template <typename T>
llista_itr<T>& llista_itr<T>::operator=
        (const llista_itr<T> & l) throw(error) {
  if (this != &l) {
    llista_itr<T> aux = l;
    swap(aux);
  }
  return *this;
}
```

```
// Cost: Θ(1)
template <typename T>
void llista_itr<T>::swap(llista_itr<T> &l) throw() {
  node* auxn = _head;
  _head = l._head;
  l._head = auxn;
  int auxs = _sz;
  _sz = l._sz;
  l._sz = auxs;
}

// Cost: Θ(n)
template <typename T>
void llista_itr<T>::destruir_llista(node* p, node* h)
throw() {
  if (p != h) {
    destruir_llista(p->next, h);
    delete p;
  }
}

// Cost: Θ(n)
template <typename T>
llista_itr<T>::~llista_itr() throw() {
  destruir_llista(_head->next, _head);
  delete _head;
}


// Cost: Θ(1)
```

El constructor per defecte de la classe iterador és privat per tal d'obligar a crear els iteradors mitjançant els mètodes **primer**, **final** i **indef** de la classe.

```
template <typename T>
llista_itr<T>::iterador::iterador() throw() {
}
```

POLIMORFISME ALS MÈTODES ++a I a++

En C++ és possible declarar dues funcions diferents que tinguin el mateix nom. Les funcions han de diferenciar-se en la llista paràmetres, ja sigui en el nombre de variables o bé en el tipus dels arguments, MAI amb el tipus de retorn. Per aquest motiu, per poder diferenciar els mètodes preincrement i postincrement, el mètode postincrement té un paràmetre que no s'utilitza.

```
// Cost: Θ(1)
Operador preincrement ++a
template <typename T>
typename llista_itr<T>::iterador&
llista_itr<T>::iterador::operator++() throw() {
  if (_p != _h) {
    _p = _p->next;
  }
  return *this;
}

// Cost: Θ(1)
Operador postincrement a++
template <typename T>
typename llista_itr<T>::iterador
llista_itr<T>::iterador::operator++(int) throw() {
  iterador tmp(*this);
  ++(*this); // es crida al mètode de preincrement
  return tmp;
}

// Cost: Θ(1)
template <typename T>
bool llista_itr<T>::iterador::operator==(iterador it)
            const throw() {
  return (_p == it._p) and (_h == it._h);
}
```

```
// Cost: Θ(1)
template <typename T>
bool llista_itr<T>::iterador::operator!=(iterador it)
            const throw() {
  return not (*this == it);
}

// Cost: Θ(1)
```

Operador predecrement --a

```
template <typename T>
typename llista_itr<T>::iterador&
llista_itr<T>::iterador::operator--() throw() {
  if (_p != _h) {
    _p = _p->prev;
  }
  return *this;
}

// Cost: Θ(1)
```

Operador postdecrement a--

```
template <typename T>
typename llista_itr<T>::iterador
llista_itr<T>::iterador::operator--(int) throw() {
  iterador tmp(*this);
  --(*this); // es crida al mètode de predecrement
  return tmp;
}

// Cost: Θ(1)
template <typename T>
const T& llista_itr<T>::iterador::operator*() const
throw(error) {
  if (_p == _h) {
    throw error(IteradorIndef);
  }
  return _p->info;
}
```

```
// Cost: Θ(1)
template <typename T>
void llista_itr<T>::inserir_davant
            (const T& x, iterador it) throw(error) {
  node* nn = new node;
  // no sabem com és el tipus T. Cap la possibilitat
  // que usi memòria dinàmica i per tant cal comprovar
  // que tot vagi bé.
  try {
    nn->info = x;
  }
  catch(const error& e) {
    delete nn;
    throw;
  }
  nn->next = it._p->next;
  nn->prev = it._p;
  it._p->next = nn;
  nn->next->prev = nn;
  ++_sz;
}

// Cost: Θ(1)
template <typename T>
void llista_itr<T>::inserir_darrera
            (const T& x, iterador it) throw(error) {
  node* nn = new node;
  try {
    nn -> info = x;
  }
  catch(const error& e) {
    delete nn;
    throw;
  }
  nn->prev = it._p->prev;
  nn->next = it._p;
  it._p -> prev = nn;
  nn -> prev -> next = nn;
  ++_sz;
}
```

```
// Cost: Θ(1)
template <typename T>
void llista_itr<T>::esborrar_avnc(iterador& it) throw() {
  if (it._p != _head) {
    node* todel = it._p;
    todel->prev->next = todel->next;
    todel->next->prev = todel->prev;
    delete todel;
    --_sz;
    it._p = it._p -> next;
  }
}

// Cost: Θ(1)
template <typename T>
void llista_itr<T>::esborrar_darr(iterador& it) throw() {
  if (it._p != _head) {
    node* todel = it._p;
    todel->prev->next = todel->next;
    todel->next->prev = todel->prev;
    delete todel;
    --_sz;
    it._p = it._p->prev;
  }
}

// Cost: Θ(1)
template <typename T>
nat llista_itr<T>::longitud() const throw() {
  return _sz;
}

// Cost: Θ(1)
template <typename T>
bool llista_itr<T>::es_buida() const throw() {
  return _sz == 0;
}
```

```
// Cost: Θ(1)
template <typename T>
typename llista_itr<T>::iterador llista_itr<T>::primer()
            const throw() {
  iterador it;
  it._p = _head->next;
  it._h = _head;
  return it;
}

// Cost: Θ(1)
template <typename T>
typename llista_itr<T>::iterador llista_itr<T>::ultim()
            const throw() {
  iterador it;
  it._p = _head -> prev;
  it._h = _head;
  return it;
}

// Cost: Θ(1)
template <typename T>
typename llista_itr<T>::iterador llista_itr<T>::indef()
            const throw() {
  iterador it;
  it._p = _head;
  it._h = _head;
  return it;
}
```

Es pot ampliar la classe llista amb les operacions:

- concatena(): Concatena 2 llistes per formar una única llista.
- parteix(): Parteix la llista en dues (es parteix pel punt d'interès).

4.6. CRITERIS PER IMPLEMENTAR CLASSES

4.6.1. Introducció

Hi ha una sèrie de problemes greus derivats de l'ús indiscriminat del mecanisme de punters que ocasionen una pèrdua de transparència de la representació del tipus. Són:

- Problema de l'assignació.
- Problema de la comparació.
- Problema dels paràmetres d'entrada.
- Problema de les variables auxiliars.
- Problema de la reinicialització.

4.6.2. Problema de l'assignació

Donats dos objectes de la classe llista, l'assignació ***L1 = L2***

- És correcta per a llistes implementades amb vectors.
- És incorrecta per a llistes implementades amb punters.

La correctesa depèn de la implementació concreta d'aquesta classe (inacceptable). Per solucionar aquest problema tota classe *A* ha sobreescriure l'operació *assignació* que fa una rèplica exacta d'un objecte de la classe *A*.

SOLUCIÓ: En el llenguatge de programació C++ es pot definir dins d'una classe un mètode anomenat **operator=** que es crida automàticament quan s'assignen objectes.

4.6.3. Problema de la comparació

Donats dos objectes de la classe llista, la comparació ***L1 == L2***

- És correcta per a llistes en vector seqüencial.

- És incorrecta per a llistes encadenades (en vector o punters)

Com en el cas anterior tota classe *A* ha d'oferir una operació d'*igualtat* que, donats dos objectes de la classe *A*, els compara.

SOLUCIÓ: En el llenguatge de programació C++ es pot sobrecarregar l'operador de comparació **operator==**

4.6.4. Problema dels paràmetres d'entrada

Donat la classe *A* i donada una funció que declara un objecte de la classe *A* com a paràmetre d'entrada, si *A* està implementada amb punters qualsevol canvi en l'objecte es reflecteix a la sortida de la funció. Això passa perquè el paràmetre d'entrada és el punter mateix però no l'objecte que no es pot protegir de cap manera.

SOLUCIÓ: En el llenguatge de programació C++ es crida automàticament el **constructor per còpia** sempre que es passen paràmetres. Per aquest motiu és necessari implementar sempre el constructor per còpia (Llei dels tres grans) en cas que aquesta classe usi memòria dinàmica.

Per motius d'eficiència, algunes de les operacions implementades amb funcions es poden passar a codificar amb accions i passar els paràmetres per referència.

4.6.5. Problema de les variables auxiliars

Sigui una variable de classe *A* declarada dins d'una funció. Si al final de l'execució de la funció la variable conté informació i *A* està implementada amb punters aquesta informació queda com a retalls.

Per evitar aquest malfuncionament, la classe *A* ha d'oferir una funció *destrueix* que alliberi l'espai ocupat per una variable de tipus *T* i és necessari cridar aquesta operació abans de sortir de la funció.

SOLUCIÓ: En el llenguatge de programació C++ es pot definir dins d'una classe un mètode anomenat **destructor** que es crida automàticament quan s'ha de destruir un objecte.

4.6.6. Problema de la reinicialització

Si es vol reaprofitar un objecte de classe *A* (implementat amb punters) creant-ne un de nou o assignant-li un altre objecte, l'espai ocupat per l'objecte prèviament a la nova creació esdevé inaccessible. És necessari, doncs, invocar abans a l'operació *destrueix* sobre l'objecte.

SOLUCIÓ: En el llenguatge de programació C++ es soluciona fent que el mètode **assignació** de la classe cridi prèviament al destructor.

4.7. ORDENACIÓ PER FUSIÓ

4.7.1. Introducció

L'ordenació per fusió (anglès: *merge-sort*) és un algorisme molt adequat per ordenar llistes encadenades perquè és fàcil partir-les i fusionar-les canviant els encadenaments de cada node.

És un dels primers algorismes eficients d'ordenació que es van proposar. Variants d'aquest algorisme són particularment útils per a l'ordenació de dades residents en memòria externa.

Suposem que els elements a ordenar estan dins d'una llista simplement encadenada, no circular, sense element fantasma i on no cal distingir cap punt d'interès. Així el tipus llista serà un únic punter que apunti al primer node de la llista.

4.7.2. Especificació

Només s'han indicat els mètodes imprescindibles de la classe llista per tal que funcioni aquest algorisme d'ordenació.

```
template <typename T>
class llista {
public:
  ...
  nat longitud() const throw();
  void sort() throw();
  ...

private:
  struct node {
    T info;       // element a ordenar
    node* seg;    // llista simplement encadenada
  };
  node* _head;    // apunta al primer node de la llista
  nat _sz;
```

```
    Mètode privat. Parteix la llista actual en dues de longituds
    similars: l'actual i L2.
    void partir(llista<T> &L2, int m) throw();

    Mètode privat de classe. Fusiona les llistes apuntades pels
    nodes n1 i n2.
    static void fusionar(node* &n1, node* &n2) throw();

    Mètode privat recursiu d'ordenació.
    void mergeSort(int n) throw();

};
```

4.7.3. Passos de l'algorisme

L'algorisme d'ordenació d'una llista per fusió realitza els següents passos:

1) Parteix per la meitat la llista inicial (que anomenarem L) obtenint dues subllistes: L i L2 de longituds similars.

2) Ordena cadascuna de les dues subllistes: fa dues crides recursives a *mergeSort()*, una amb la subllista L i l'altra amb la subllista L2.

3) Finalment fusiona les subllistes ja ordenades L i L2 per obtenir la llista final L ordenada.

Si la llista a ordenar és suficientment petita es pot usar un mètode més simple i eficaç (no cal esperar a que les llistes tinguin un sol element per aturar la recursivitat).

4.7.4. Implementació

```
template <typename T>
void llista<T>::mergeSort(int n) throw() {
  llista<T> L2;
  if (n>1) {
    nat m = n / 2;
    // parteix la llista en curs en dues: l'actual i L2
    partir(L2, m);
    mergeSort(m);
    L2.mergeSort(n-m);
    // fusiona la llista en curs i L2 en la primera
    fusionar(_head, L2._head);
  }
}
```

La implementació del mètode sort és la següent:

```
template <typename T>
void llista<T>::sort() throw() {
  mergeSort(longitud());
}
```

El mètode *partir()* parteix la llista en curs de longitud *n* en dues subllistes L (que conté els *m* primers elements) i L2 (que conté els restants *n-m* elements).

```
template <typename T>
void llista<T>::partir(llista<T> &L2, int m) throw() {
  node* p = _head;
  while (m>1) {
    p = p->seg;
    --m;
  }
  L2._head = p->seg;
  p->seg = NULL;
}
```

Per implementar l'acció *fusionar()* raonarem de la següent manera: El resultat de fusionar dues llistes L i L2 és:

- Si L és buida és la llista L2.
- Si L2 és buida és la llista L.
- Si L i L2 no són buides compararem els seus primers elements. El menor dels dos serà el primer element de la llista fusionada i a continuació vindrà la llista resultat de fusionar la subllista que succeeix a aquest primer element amb l'altra llista.

Per tal de fer el mètode fusionar més eficient en comptes de passar-li llistes li passarem nodes.

```
template <typename T>
void llista<T>::fusionar(node* &n1, node* &n2) throw() {
  if (n1 == NULL) {
    n1 = n2;
    n2 = NULL;
  }
  if (n1 != NULL and n2 != NULL) {
    if (n1->info > n2->info) {
      // intercanviem n1 i n2
      node *aux = n1;
      n1 = n2;
      n2 = aux;
    }
    node *aux = n1->seg;
    fusionar(aux, n2);
    n1->seg = aux;
  }
}
```

Podríem implementar una versió iterativa de l'acció *fusionar()* que seria lleugerament més eficient.

4.7.5. Costos

El cost de l'acció *partir()* i *fusionar()* una llista de *n* elements és **Θ(n)** (l'acció *fusionar()* visita cada element de les llistes L i L2 una vegada). Per això el cost d'executar una sola vegada l'acció MergeSort és **Θ(n)** (cost lineal).

Com que MergeSort fa dues crides recursives amb llistes que cada vegada són la meitat de grans que les anteriors, MergeSort té un cost logarítmic multiplicat pel cost d'executar una sola vegada l'acció MergeSort.

Per tant, el cost temporal d'aplicar MergeSort a una llista de *n* elements és

$$\mathbf{T_{MergeSort}(n) = \Theta(\, n \cdot log(n)\,).}$$

Obtindrem el mateix resultat si resolem l'equació de recurrència:

$$\mathbf{T}\text{MergeSort}\mathbf{(n) = 2 \cdot T}\text{MergeSort}\mathbf{(n/2) + \Theta(n)}$$

TEMA 5. ARBRES

5.1. ARBRES GENERALS

5.1.1. Definició d'Arbre general

La definició més genèrica d'arbre és la que els relaciona amb un tipus de graf:

Definició 1: Un arbre (lliure) T és un graf no dirigit connex i acíclic.

Però normalment s'utilitzen arbres arrelats i orientats:

- **Arrelats**: Es distingeix un dels nodes de l'arbre com arrel, induint a la resta de nodes a adoptar una relació jeràrquica.
- **Orientats**: S'imposa un ordre entre els subarbres de cada node.

Encara que també es pot definir com:

Definició 2: Un arbre general T de grandària n, $n>0$, és una col·lecció de n nodes, un dels quals s'anomena arrel, i els $n-1$ nodes restants formen una seqüència de $k \geq 0$ arbres T_1, ... T_k de grandàries $n_1, ..., n_k$, connectats a l'arrel i que compleixen:

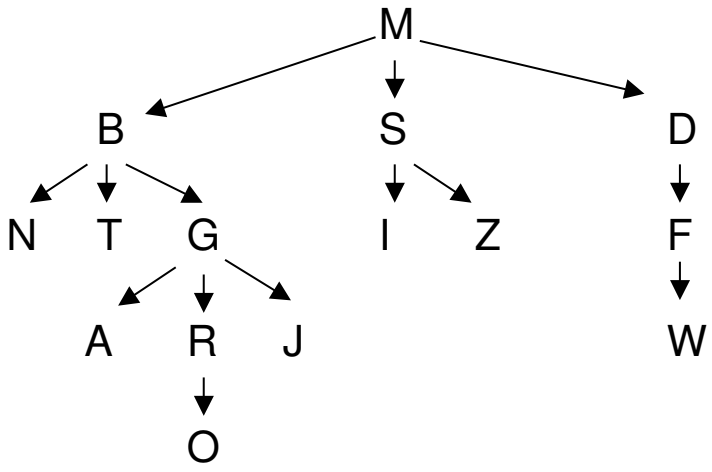
- $n_i > 0$, per $1 \leq i \leq k$
- $n-1 = n_1 + n_2 + ... + n_k$

5.1.2. Altres definicions

Sigui un arbre T d'arrel r que té k subarbres amb arrels $r_1, ..., r_k$.

- $r_1, ..., r_k$ són els **fills** de r.
- r és el **pare** de $r_1, ..., r_k$.
- Tot node de T, excepte l'arrel, té un pare.
- $r_1, ..., r_k$ són **germans**, doncs tenen el mateix pare.
- El **grau** d'un node és el nombre de fills que té aquest node.
- Donat un arbre T i qualsevol node x de T existeix un **camí únic** des de l'arrel fins a x.
- Un node x és un **descendent** de un node y si y està en el camí únic de l'arrel a x. A la inversa, y és un **antecessor** de x.
- Si un node no té descendents (el seu grau és 0) és diu que és una **fulla**.
- La **profunditat** d'un node x és la longitud del camí entre l'arrel i el node x.
- El **nivell** k d'un arbre T el formen tots aquells nodes que tenen profunditat k.
- L'**altura** d'un node x és la longitud màxima entre x i les fulles descendents de x.
- L'**altura d'un arbre** és l'altura de la seva arrel.

Exemple:



- B, S, D són germans i fills de M.
- M és pare de B, S i D.
- El grau de M és 3.
- N, T, A, O, J, I, Z i W són fulles.
- R té una profunditat de 3 i una altura de 1.
- El nivell 3 està format pels nodes A, R, J i W.
- L'altura de l'arbre és 4 (és l'altura de l'arrel M).

5.2. ESPECIFICACIÓ D'ARBRES GENERALS

Per la definició d'arbre general tot arbre es correspon a un terme de la forma $x \times T_1 \times \cdots \times T_k$. Per tant, sols cal una operació generadora pura que ens crearà un arbre a partir d'un element i una seqüència d'arbres.

Fixem-nos que no existeix l'arbre general buit.

Necessitarem consultores per poder accedir a l'arrel i als subarbres de l'arrel.

5.2.1. Especificació d'un arbre general amb accés per primer fill–següent germà

Podem accedir als subarbres d'un node accedint al primer subarbre (primer fill) i a partir d'ell als germans. Incorporem operacions similars a les utilitzades per recórrer les llistes amb punt d'interès.

Per recórrer els subarbres d'un node utilitzarem:

- **arrel**: hauria de retornar un iterador sobre el primer arbre del bosc o un iterador nul si el bosc és buit.
- **primogènit**: hauria de retornar un iterador sobre l'arrel del primer arbre del bosc, format pels fills de l'arrel sobre el que s'aplica.
- **seg_germà**: hauria de retornar un iterador a l'arrel de següent arbre del bosc (tot iterador apunta a l'arrel d'un arbre que és part d'un cert bosc).
- **final**: ens hauria de retornar un iterador no vàlid. Ens permet saber si hem arribat al final de la llista de germans.

Sovint a una seqüència d'arbres s'anomena bosc. A diferència dels arbres, un bosc pot ser buit, ja que és una seqüència i aquestes poden ser buides.

```
template <typename T>
class Arbre {
public:
```

```
Construeix un Arbre format per un únic node que conté a x.
```

```
Arbre(const T &x) throw(error);
```

```
Tres grans.
```

```
Arbre(const Arbre<T> &a) throw(error);
Arbre& operator=(const Arbre<T> &a) throw(error);
~Arbre() throw();
```

```
Col·loca l'Arbre donat com a primer fill de l'arrel de l'arbre
sobre el que s'aplica el mètode i l'arbre a queda invalidat;
després de fer b.afegir_fill(a), a no és un arbre vàlid.
```

```
void afegir_fill(Arbre<T> &a) throw(error);
```

```
Iterador sobre arbre general.
```

```
friend class iterador;
class iterador {
public:
  friend class Arbre;
```

```
  Construeix un iterador no vàlid.
```

```
  iterador() throw();
```

```
  Retorna el subarbre al que apunta l'iterador; llança un error
  si l'iterador no és vàlid.
```

```
  Arbre<T> arbre() const throw(error);
```

```
  Retorna l'element del node al que apunta l'iterador o llança
  un error si l'iterador no és vàlid.
```

```
  T operator*() const throw(error);
```

```
    Retorna un iterador al primogenit del node al que apunta;
    llança un error si l'iterador no és vàlid.
    iterador primogenit() const throw(error);

    Retorna un iterador al següent germà del node al que apunta;
    llança un error si l'iterador no és vàlid.
    iterador seg_germa() const throw(error);

    Operadors de comparació.
    bool operator==(const iterador &it) const
                    throw(error) ;
    bool operator!=(const iterador &it) const
                    throw(error) ;

    static const int IteradorInvalid = 401;

  private:
    // Aquí aniria la representació de la classe iterador
    ...
  };

  Retorna un iterador al node arrel de l'Arbre (un iterador no
  vàlid si l'arbre no és vàlid).
  iterador arrel() const throw();

  Retorna un iterador no vàlid.
  iterador final() const throw();

  static const int ArbreInvalid = 400;

private:
  // Aquí aniria la representació de la classe Arbre
  ...
};
```

5.3. ESPECIFICACIÓ D'ARBRES BINARIS (M-ARIS)

Un **arbre binari** és un cas especial d'arbre general en que tots els nodes són fulles o bé tenen grau 2.

En un arbre binari es diferencia els dos possibles fills que pot tenir cada node. Per exemple, els següents arbres són dos arbres binaris diferents (en canvi, des de la perspectiva d'arbres generals són el mateix, doncs són arbres amb un node que només té un únic fill).

Normalment a les fulles d'un arbre binari no es guarda informació. Per això en la signatura s'ha incorporat una operació per tal de crear l'arbre binari buit.

Fixem-nos que en la representació gràfica d'un arbre binari no és sol dibuixar les fulles (arbres binaris buits). Si en els següents exemples dibuixeu les fulles buides, observareu que els nodes P i H tenen grau 2.

5.3.1. Especificació d'arbres binaris

L'especificació d'arbres binaris que heu vist fins ara a altres assignatures és la següent:

```
template <typename T>
class Abin {
public:
  Construeix l'arbre binari buit.
  Abin() throw(error);

  Tres grans.
  Abin(const Abin<T> &a) throw(error);
  Abin& operator=(const Abin<T> &a) throw(error);
  ~Abin() throw();

  Constructora: crea un arbre binari l'arrel del qual conté
  l'element x, i amb els subarbre dret fdret i el subarbre
  esquerre fesq. Els arbres fesq i fdret es destrueixen, és a
  dir, després d'aplicar la constructora fesq i fdret són buits;
  pot propagar un error de memòria dinàmica si manca memòria pel
  node arrel.
  Abin(Abin<T> &fesq, const T &x, Abin<T> &fdret)
       throw(error);

  Obtenir element de dalt de tot de l'arbre.
  const T& arrel() const throw();

  Obtenir el subarbre que queda per l'esquerra.
  const Abin<T>& fe() const throw();

  Obtenir el subarbre que queda per la dreta.
  const Abin<T>& fd() const throw();

  Saber si l'arbre es buit o no.
  bool es_buit() const throw();

};
```

Una especificació més eficient i versàtil amb iteradors es pot veure a continuació. Aquesta segona especificació és la que emprem durant tot el curs.

```
template <typename T>
class Abin {
public:
```

```
Construeix l'arbre binari buit.
```

```
  Abin() throw(error);
```

```
Tres grans.
```

```
  Abin(const Abin<T> &a) throw(error);
  Abin& operator=(const Abin<T> &a) throw(error);
  ~Abin() throw();
```

```
Constructora: crea un arbre binari l'arrel del qual conté
l'element x, i amb els subarbre dret fdret i el subarbre
esquerre fesq. Els arbres fesq i fdret es destrueixen, és a
dir, després d'aplicar la constructora fesq i fdret són buits;
pot propagar un error de memòria dinàmica si manca memòria pel
node arrel.
```

```
  Abin(Abin<T> &fesq, const T &x, Abin<T> &fdret)
       throw(error);
```

```
Retorna cert ssi l'arbre és buit.
```

```
  bool es_buit() const throw();
```

```
Iterador sobre arbres binaris.
```

```
  friend class iterador;

  class iterador {
  public:
    friend class Abin;
```

```
    Construeix un iterador no vàlid.
```

```
    iterador() throw();
```

```
    Retorna el subarbre al que apunta l'iterador; llança un error
    si l'iterador no és vàlid.
    Abin<T> arbre() const throw(error);

    Retorna l'element en el node al que apunta l'iterador, o
    llança un error si l'iterador no és vàlid.
    T operator*() const throw(error);

    Retorna un iterador al fill esquerre o al fill dret; llança
    un error si l'iterador no és vàlid.
    iterador fesq() const throw(error);
    iterador fdret() const throw(error);

    Operadors de comparació.
    bool operator==(const iterador &it) const throw();
    bool operator!=(const iterador &it) const throw();

    static const int IteradorInvalid = 410;

  private:
    // Aquí aniria la representació de la classe iterador
    ...
  };

  Retorna un iterador al node arrel.
  iterador arrel() const throw();

  Retorna un iterador no vàlid.
  iterador final() const throw()

private:
  // Aquí aniria la representació de la classe Abin
  ...
};
```

5.3.2. Especificació d'arbres m-aris

Els **arbres m-aris** són una generalització dels arbre binaris (2-aris):

Definició 3: Un arbre és m-ari si tots els seus nodes o bé són fulles o bé tenen grau m.

És habitual que l'accés als subarbres d'un arbre m-ari es faci per posició. La posició serà un natural dins l'interval [1..m].

```
template <typename T>
class Arb_m_ari {
public:
```

```
Construeix l'arbre m-ari buit.
```

```
Arb_m_ari(int m) throw(error);
```

```
Tres grans.
```

```
Arb_m_ari(const Arb_m_ari<T> &a) throw(error);
Arb_m_ari& operator=(const Arb_m_ari<T> &a)
                     throw(error);
~Arb_m_ari() throw();
```

```
Constructora: crea un arbre m-ari l'arrel del qual conté
l'element x, i amb els subarbres a[0], a[1], ..., a[m-1]; els
subarbres a[i] es destrueixen, és a dir, després d'aplicar la
constructora el vector d'entrada és buit (cada a[i] és buit);
es considera que cada a[] té almenys m subarbres, en cas
contrari el comportament és indefinit; pot propagar un error de
memòria dinàmica si falta memòria pel node arrel.
```

```
Arb_m_ari(const T &x, Arb_m_ari a[]) throw(error);
```

```
Retorna cert ssi l'arbre és buit.
```

```
bool es_buit() const throw();
```

```
Iterador sobre arbres m-aris.
```

```
friend class iterador;
```

```
  class iterador {
  public:
    friend class Arb_m_ari;

    Construeix un iterador no vàlid.
    iterador() throw();

    Retorna el subarbre al que apunta l'iterador; llança un error
    si l'iterador no és vàlid.
    Arb_m_ari<T> arbre() const throw(error);

    Retorna l'element en el node al que apunta l'iterador, o
    llança un error si l'iterador no és vàlid.
    T operator*() const throw(error);

    Retorna un iterador al fill i-èssim del node apuntat; llança
    un error si l'iterador no és vàlid o si i està fora del rang
    0..m-1.
    iterador fill(int i) const throw(error);

    Operadors de comparació.
    bool operator==(const iterador &it) const throw();
    bool operator!=(const iterador &it) const throw();

  private:
    // Aquí aniria la representació de la classe iterador
    ...
  };

  Retorna un iterador al node arrel.
  iterador arrel() const throw();

  Retorna un iterador no vàlid.
  iterador final() const throw()

private:
  // Aquí aniria la representació de la classe
  // Arbre_m_ari
  ...

};
```

5.4. USOS DELS ARBRES

Els arbres tenen molts usos. Destaquem els següents:

- Representació d'expressions i arbres sintàctics (anglès: *parse trees*)
- Sistemes de fitxers (jerarquia de subdirectoris/carpetes i fitxers)
- Implementació de cues de prioritat (anglès: *priority queues*)
- Implementació de particions (anglès: *MFSets*)
- Implementació de diccionaris i índexs per a bases de dades:
  - Arbres binaris de cerca: BST (*binary search tree)*, AVL
  - Tries
  - B-trees
  - ...

RECURSIVITAT ALS ARBRES

La definició d'**arbre** com ja hem vist és una *definició recursiva* (un arbre està format per altres arbres). Per aquesta raó la majoria dels algorismes sobre arbres són de manera natural recursius.

5.5. RECORREGUTS D'UN ARBRE

5.5.1. Introducció

Un **recorregut** (anglès: *traversal*) d'un arbre visita tots els nodes d'un arbre de forma sistemàtica, sense repeticions i tenint en compte un cert ordre prefixat.

Una de les utilitats és crear una llista amb tots els elements de l'arbre en l'ordre desitjat.

El recorregut un arbre és independent de la seva implementació.

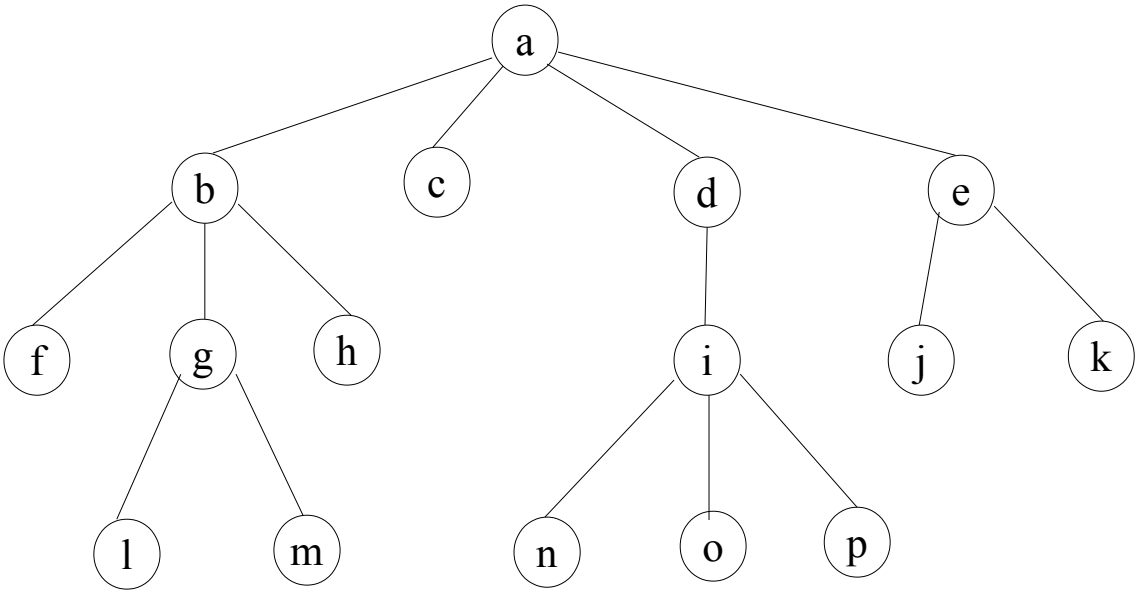
Hi ha 3 tipus de recorreguts d'arbres:

- **Preordre**: Es visita l'arrel i després els nodes dels subarbres respectant l'ordre d'aquests.
- **Postordre**: Es visita els nodes dels subarbres respectant l'ordre d'aquests i finalment es visita l'arrel.
- **Per nivells**: Es visiten els nodes per nivells, de menor a major nivell, i respectant l'ordre dels subarbres dins de cada nivell.

Els arbres binaris es poden recórrer de les 3 maneres anteriors i també en **inordre** (es visita el fill esquerre, l'arrel i el fill dret).

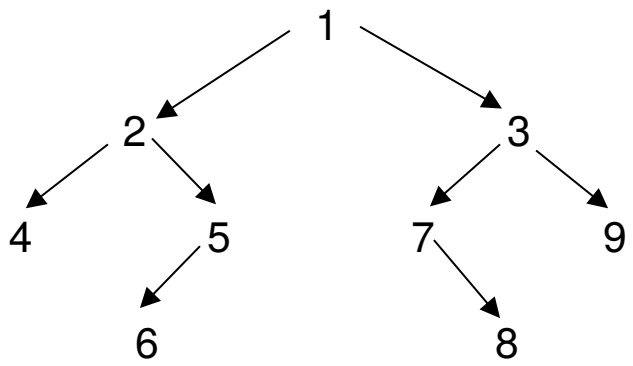
NOTA: La classe llista, pila i cua (list, stack i queue respectivament) que farem servir en aquestes implementacions pertanyen a la biblioteca STL.

Exemple arbre general:



Preordre: a, b, f, g, l, m, h, c, d, i, n, o, p, e, j, k
Postordre: f, l, m, g, h, b, c, n, o, p, i, d, j, k, e, a
Nivells: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p

Exemple arbre binari:



Preordre: 1, 2, 4, 5, 6, 3, 7, 8, 9.
Postordre: 4, 6, 5, 2, 8, 7, 9, 3, 1.
Nivells: 1, 2, 3, 4, 5, 7, 9, 6, 8.
Inordre: 4, 2, 6, 5, 1, 7, 8, 3, 9.

5.5.2. Recorregut en Preordre

5.5.2.1. Implementació recursiva del recorregut en preordre (arbre general)

Al final del mètode la llista *lpre* contindrà els elements d'*a* en preordre.

```
template <typename T>
void rec_preordre (const Arbre<T> &a, list<T> &lpre) {
  rec_preordre(a.arrel(), a.final(), lpre);
}

template <typename T>
void rec_preordre (Arbre<T>::iterador it,
                   Arbre<T>::iterador end,
                   list<T> &lpre) {
  if (it != end) {
    lpre.push_back(*it);
    rec_preordre(it.primogenit(), end, lpre);
    rec_preordre(it.seg_germa(), end, lpre);
  }
}
```

Crida inicial:

```
Arbre<int> a;
...
list<int> l;
rec_preordre(a, l);
```

El cas d'arbres binaris és similar: Es visita primer l'arrel, després recorregut en preordre del subarbre esquerre i després recorregut en preordre del subarbre dret.

5.5.2.2. Implementació iterativa del recorregut en preordre d'un arbre binari

En aquesta implementació s'utilitza una pila d'iteradors d'arbres com a variable auxiliar.

```
Al final del mètode la llista lpre contindrà els elements d'a en preordre.
```

```
template <typename T>
void rec_preordre (const Abin<T> &a, list<T> &lpre) {
  stack<Abin<T>::iterador> s;

  if (not a.es_buit()) {
    s.push(a.arrel());
  }
  while (not s.empty())  {
    Abin<T>::iterador it = s.top();
    s.pop();
    lpre.push_back(*it);
    if (it.fdret() != a.final()) {
      s.push(it.fdret());
    }
    if (it.fesq() != a.final()) {
      s.push(it.fesq());
    }
  }
}
```

5.5.3. Recorregut en Postordre

5.5.3.1. Implementació recursiva del recorregut en postordre (arbre general)

Al final del mètode la llista *lpost* contindrà els elements d'*a* en postordre.

```
template <typename T>
void rec_postordre (const Arbre<T> &a, list<T> &lpost) {
  rec_postordre(a.arrel(), a.final(), lpost);
}


template <typename T>
void rec_postordre (Arbre<T>::iterador it,
                    Arbre<T>::iterador end,
                    list<T> &lpost) {
  if (it != end) {
    rec_postordre(it.primogenit(), end, lpost);
    lpost.push_back(*it);
    rec_postordre(it.seg_germa(), end, lpost);
  }
}
```

Crida inicial:

```
Arbre<int> a;
...
list<int> l;
rec_postordre(a, l);
```

El cas d'arbres binaris és similar: Es fa primer el recorregut en postordre del subarbre esquerre, després es fa el recorregut en postordre del subarbre dret i finalment es visita l'arrel.

5.5.3.2. Implementació iterativa del recorregut en postordre d'un arbre binari

El recorregut en postordre equival a fer un recorregut en preordre especular (intercanvi de fill esquerre i fill dret) i a capgirar la llista resultant. L'algorisme a continuació fa ús d'aquesta propietat. Per capgirar la llista s'insereixen els elements sempre pel principi de la llista.

```
template <typename T>
void rec_postordre (const Abin<T> &a, list<T> &lpost) {
  stack<Abin<T>::iterador> s;

  if (not a.es_buit()) {
    s.push(a.arrel());
  }
  while (not s.empty())  {
    Abin<T>::iterador it = s.top();
    s.pop();
    lpost.push_front(*it);
    if (it.fesq() != a.final()) {
      s.push(it.fesq());
    }
    if (it.fdret() != a.final()) {
      s.push(it.fdret());
    }
  }
}
```

5.5.4. Recorregut per nivells

Per recórrer un arbre per nivells utilitzarem una cua d'arbres (cua d'iteradors d'arbre), on guardarem aquells arbres que la seva arrel encara no s'ha visitat. Al principi encuarem l'arbre inicial.

El procés a seguir és simple: s'extreu un arbre de la cua, es visita l'arrel, es col·loquen tots els seus fills a la cua i repetim el procés.

```
template <typename T>
void rec_nivells (const Arbre<T> &a, list<T> &lniv) {
  queue<Arbre<T>::iterador> q;

  q.push_back(a.arrel());
  while (not q.empty())  {
    Arbre<T>::iterador it = q.pop_front();
    lniv.push_back(*it);
    it  = it.primogenit();
    while (it != a.final()) {
      q.push_back(it);
      it = it.seg_germa();
    }
  }
}
```

5.5.5. Recorregut en Inordre

El recorregut en inordre només té sentit fer-ho en els arbres binaris. Es fa primer el recorregut en inordre del subarbre esquerre, després es visita l'arrel i després es fa el recorregut en inordre del subarbre dret.

5.5.5.1. Implementació recursiva del recorregut en inordre

```
Al final del mètode la llista lin contindrà els elements d'a en inordre.
```

```
template <typename T>
void rec_inordre (const Abin<T> &a, list<T> &lin) {
  rec_inordre(a.arrel(), a.final(), lin);
}


template <typename T>
void rec_inordre (Abin<T>::iterador it,
                  Abin<T>::iterador end,
                  list<T> &lin) {
  if (it != end) {
    rec_inordre(it.fesq(), end, lin);
    lin.push_back(*it);
    rec_inordre(it.fdret(), end, lin);
  }
}
```

Crida inicial:

```
Abin<int> a;
...
list<int> l;
rec_inordre(a, l);
```

5.5.5.2. Implementació iterativa del recorregut en inordre

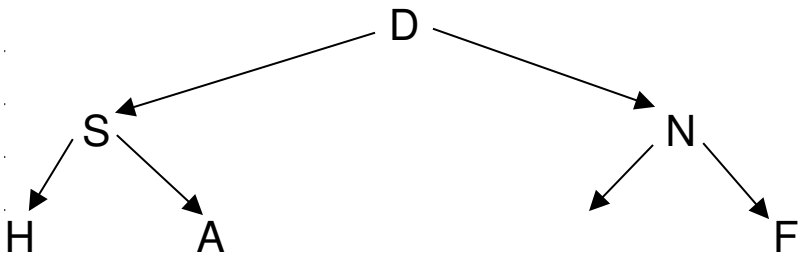
En aquesta implementació s'utilitza una pila d'iteradors d'arbres com a variable auxiliar. Fixeu-vos que a la pila d'iteradors s'apila primer l'iterador del fill dret, després l'arrel (creant un arbre que només conté l'arrel doncs el fills esquerra i dret són buits) i finalment l'iterador del fill esquerra. Així en les següents iteracions trobem els elements de la pila en inordre.

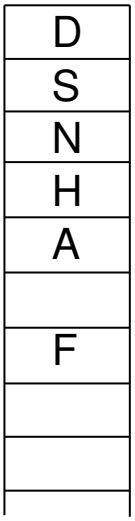
```
template <typename T>
void rec_inordre (const Abin<T> &a, list<T> &lin) {
  stack<Abin<T>::iterador> s;

  if (not a.es_buit()) {
    s.push(a.arrel());
  }
  while (not s.empty()) {
    Abin<T>::iterador it = p.top();
    s.pop();
    if (it.fesq() == a.final() and
        it.fdret() == a.final()) {
      lin.push_back(*it);
    }
    else {
      if (it.fdret() != a.final())
        s.push(it.fdret());
      }
      s.push((Abin<T>(Abin<T>(), *it, Abin<T>()))
             .arrel());
      if (it.fesq() != a.final()) {
        s.push(it.fesq());
      }
    }
  }
}
```

5.6. IMPLEMENTACIÓ D'ARBRES BINARIS

5.6.1. Implementació amb vector

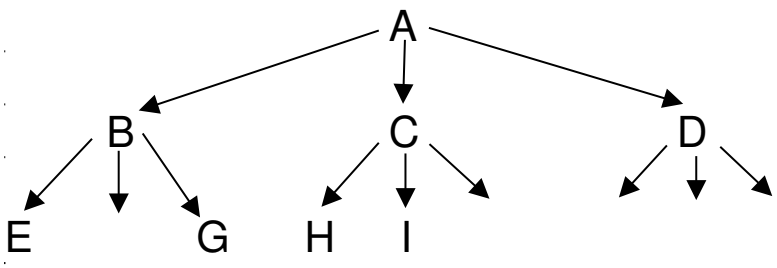
- Si l'arbre és **quasi complet** (tots els nivells de l'arbre estan gairebé plens) és convenient la implementació en vector.
- Els nodes de l'arbre es guarden com si féssim un recorregut per nivells: en la primera posició del vector es guarda l'arrel, en la 2ª i 3ª posició els seus fills esquerre i dret respectivament, ...
- Avantatges: Ens estalviem punters i l'accés a pares i fills és immediat.

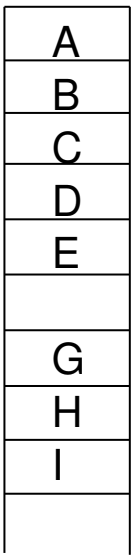




Node i ----->	fill esquerre:	2i
	fill dret:	2i + 1
	pare:	$\lfloor i/2 \rfloor$

- Generalització a un arbre m-ari (per exemple arbre 3-ari)





Node i ----->	fill central:	3i
	fill esquerre:	3i –1
	fill dret:	3i + 1
	pare:	$\lfloor (i+1)/3 \rfloor$

5.6.2. Implementació amb punters

- Per cada node guardarem la informació del node i dos punters que apunten els fills esquerre i dret.
- Si fes falta accedir al pare, afegirem un punter que apunti al pare.

5.6.2.1. Representació de la classe Abin amb punters

```
template <typename T>
class Abin {
public:
  ...
  class iterador {
    ...
    private:
      Abin<T>::node* _p;
  };

private:
  struct node {
    node* f_esq;
    node* f_dret;
    node* pare; // opcional
    T info;
  };
  node* _arrel;

  ...
};
```

5.6.2.2. Implementació de la classe Abin amb punters

La implementació d'aquesta classe no és completa. Només implementarem els mètodes més rellevants de la classe.

```
// Cost: Θ(1)
template <typename T>
Abin<T>::Abin() : _arrel(NULL) throw(error) {
}
```

Fixeu-vos que la segona constructora d'Abin és intrusiva: no fa còpia dels subarbres *ae* i *ad* i, per tant, l'arbre resultat els incorpora directament.

```
// Cost: Θ(1)
template <typename T>
Abin<T>::Abin(Abin<T> &ae, const T &x, Abin<T> &ad)
throw(error) {
  _arrel = new node;
  try {
    _arrel -> info = x;
  }
  catch (error) {
    delete _arrel;
    throw;
  }
  _arrel -> f_esq = ae._arrel;
  ae._arrel = NULL;
  _arrel -> f_dret = ad._arrel;
  ad._arrel = NULL;
}

// Cost: Θ(n)
template <typename T>
Abin<T>::Abin(const Abin<T> &a) throw(error) {
  ...
}

// Cost: Θ(n)
template <typename T>
Abin<T>& Abin<T>::operator=(const Abin<T> &a)
throw(error) {
  ...
}

// Cost: Θ(n)
template <typename T>
Abin<T>::~Abin() throw() {
  ...
}
```

```
// Cost: Θ(1)
template <typename T>
Abin<T>::iterador Abin<T>::arrel() const throw() {
  iterador it;
  it._p = _arrel;
  return it;
}

// Cost: Θ(1)
template <typename T>
Abin<T>::iterador Abin<T>::final() const throw() {
  return iterador();
}

// Cost: Θ(1)
template <typename T>
bool Abin<T>::es_buit() const throw() {
  return (_arrel == NULL);
}


// Cost: Θ(1)
template <typename T>
Abin<T>::iterador::iterador() : _p(NULL) throw() {
}

// Cost: Θ(1)
template <typename T>
Abin<T>::iterador Abin<T>::iterador::fesq() const
throw(error) {
  if (_p == NULL)
    throw error(IteradorInvalid);
  iterador it;
  it._p = _p -> f_esq;
  return it;
}
```

```
// Cost: Θ(1)
template <typename T>
Abin<T>::iterador Abin<T>::iterador::fdret() const
throw(error) {
  if (_p == NULL)
    throw error(IteradorInvalid);
  iterador it;
  it._p = _p -> f_dret;
  return it;
}

// Cost: Θ(1)
template <typename T>
T Abin<T>::iterador::operator*() const throw(error) {
  if (_p == NULL)
    throw error(IteradorInvalid);
  return _p -> info;
}

// Cost: Θ(n)
template <typename T>
Abin<T> Abin<T>::iterador::arbre() const throw(error) {
  if (_p == NULL)
    throw error(IteradorInvalid);
  Abin<T> a;
  a._arrel = _p;
  Abin<T> aux(a); // fem la copia
  a._arrel = NULL;
  return aux;
}

// Cost: Θ(1)
template <typename T>
bool  Abin<T>::iterador::operator==(const  iterador  &it)
const throw() {
  return _p == it._p;
};

// Cost: Θ(1)
template <typename T>
bool  Abin<T>::iterador::operator!=(const  iterador  &it)
const throw() {
  return _p != it._p;
};
```

5.6.3. Arbres binaris enfilats i els seus recorreguts

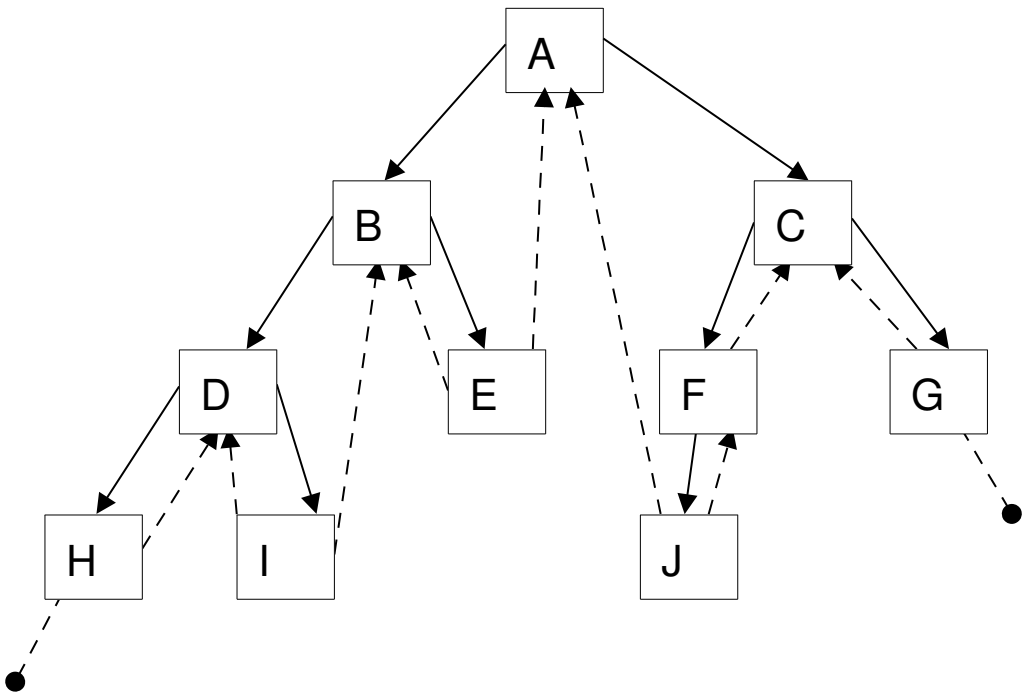
- Fins ara hem vist el recorregut en inordre utilitzant una pila. L'objectiu ara és fer el mateix sense la pila.

SOLUCIÓ 1: Tenir a cada node punters als fills i al pare. D'aquesta manera ens podem moure amb més facilitat.

SOLUCIÓ 2: Utilitzar arbres enfilats en inordre. La idea bàsica és substituir els punters nuls per punters anomenats **fils** (threads) a altres nodes de l'arbre.

- Si el punter al fill esquerre és NULL, ho canviem per un punter al node predecessor en inordre.
- Si el punter al fill dret és NULL, ho canviem per un punter al node successor en inordre.

Exemple:



5.6.3.1. Representació de la classe d'arbres binaris enfilats

- La representació de la classe d'arbres enfilats seria la següent:

```
template <typename T>
class abin_enf {
public:
  ...
private:
  struct node {
    bool thread_esq;  // ens diu si el fill esquerre és
    node* fesq;       // un fill o és un thread.
    T info;
    bool thread_dret; // ens diu si el fill dret és un
    node* fdret;      // fill o és un thread.
  }
  node* _arrel;
};
```

EXCEPCIONS EN ELS CONSTRUCTORS

Les **excepcions** són una manera de reaccionar a circumstàncies excepcionals del nostre programa (errors d'execució).

Per donar major flexibilitat en les especificacions habitualment el *constructor per defecte*, *constructor per còpia* i l'*operador d'assignació* podran retornar un error. Si en la implementació s'utilitza **memòria dinàmica** el sistema pot generar una excepció per manca de memòria.

Però això no vol dir que sempre es produiran errors.

5.6.3.2. Recorreguts amb arbres enfilats

- Recorregut en **Preordre**: el primer node és l'arrel de l'arbre. Donat un node n el seu successor en preordre és:
  - el fill esquerre si en té sinó
  - el fill dret si en té sinó
  - si és una fulla cal seguir els fils (threads) drets fins arribar a una node que enlloc de fil (thread) dret tingui fill dret. Aquest últim és el successor.

- Recorregut en **Inordre**: el primer node es localitza baixant recursivament per la branca més a l'esquerra de l'arbre. Donat un node n el seu successor és:
  - el primer en inordre del fill dret si en té (cal baixar recursivament per la branca més a l'esquerra del fill dret)
  - sinó es segueix el fil (thread) dret, doncs apunta al successor en inordre.

Un arbre binari es pot enfilar fàcilment si:

- Afegim 2 punters (que apuntin al primer i al darrer en inordre).

- L'operació **constructora**, a més a més d'arrelar els subarbres esquerre i dret amb la nova arrel, afegeix aquests dos nous fils:
  - El punter dret del darrer en inordre del subarbre esquerre apuntarà a la nova arrel (abans apuntava a NULL).
  - El punter esquerra del primer en inordre del subarbre dret apuntarà a la nova arrel (abans apuntava a NULL).

5.7. IMPLEMENTACIÓ D'ARBRES GENERALS

La implementació general d'arbres consisteix en emmagatzemar els elements en nodes cadascun dels quals conté una llista d'apuntadors (o cursors) a les arrels dels seus subarbres.

5.7.1. Implementació amb vector de punters

La implementació en un sol vector és poc adequada doncs difícilment els arbres generals són quasi complets i es desaprofita molt l'espai. A més a més hem d'escollir com a grau el màxim nombre de fills que té un node.

- Si el grau màxim dels nodes està acotat i no és gaire gran podem guardar a cada node un vector de punters que apuntin als fills.
- L'arbre es representa simplement com un punter que apunta al node arrel.

```
template <typename T>
class Arbre {
  ...
private:
  Arbre() throw(error);
  const nat MAXFILLS = ... ;
  struct node {
     T info;
     node* fills[MAXFILLS];
  };
  node* _arrel;
};
```

- Permet una implementació eficient de l'accés als fills per posició.
- Aquest implementació és especialment atractiva per arbres *m*-aris amb un *m* petit.

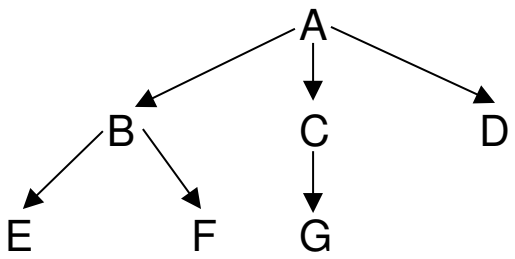
- En cas que el nombre de fills màxim és desconegués a priori es podria caldria fer servir taules dinàmiques.

5.7.2. Implementació amb punters

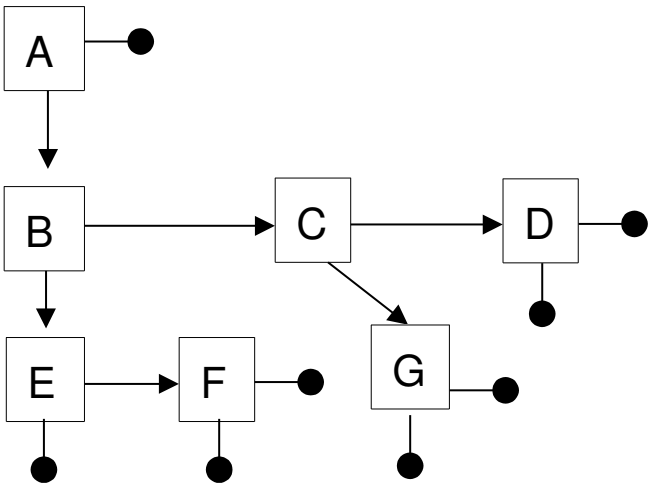
Si el grau màxim no està acotat, és gran o hi ha molta variació possible, llavors convindrà representar la llista de fills amb una llista enllaçada.

Cada node de l'arbre estarà representat per una llista que començarà pel primer fill i enllaçarà els germans en ordre.

- Aquesta representació s'anomena **primer fill - següent germà**.
- Cada node de la llista contindrà un element i dos apuntadors: al seu primer fill i al seu següent germà (el germà de la dreta).
- Amb aquesta representació transformarem un arbre general en un arbre binari. Per exemple, aquest arbre



quedaria representat de la següent manera:



- Un arbre o un bosc (seqüència d'arbres) consistirà en un punter al node arrel. El punter següent germà del node arrel serà NULL en un arbre (l'arrel no té germans) però pot ser aprofitat per crear una llista d'arbres (bosc).

5.7.2.1. Representació d'un arbre general amb accés a primer fill – següent germà.

Cal indicar la representació tant per la classe Arbre com pel seu iterador.

```
template <typename T>
class Arbre {
public:
  ...
  friend class iterador {
    ...
  private:
    Arbre<T>::node* _p;
  };
  ...

private:
  Arbre();
  struct node {
    T info;
    node* primf;
    node* seggerm;
  };
  node* _arrel;
  node* copia_arbre(node* p) throw(error);
  void destrueix_arbre(node* p) throw();
};
```

5.7.2.2. Implementació d'un arbre general amb accés a primer fill – següent germà.

El constructor per defecte està definit com un mètode privat.

```
// Cost: Θ(1)
template <typename T>
Arbre<T>::Arbre() : _arrel(NULL) {
}
```

```
// Cost: Θ(1)
```
Construcció d'un arbre que conté un sol element x a l'arrel.
```
template <typename T>
Arbre<T>::Arbre(const T &x) throw(error) {
  _arrel = new node;
  try {
    _arrel -> info = x;
    _arrel -> seggerm = NULL;
    _arrel -> primf = NULL;
  }
  catch (error) {
    delete _arrel;
    throw;
  }
}
```

La còpia es fa seguint un recorregut en preordre.
```
// Cost: Θ(n)
template <typename T>
typename Arbre<T>::node* Arbre<T>::copia_arbre(node* p)
throw(error) {
  node* aux = NULL;
  if (p != NULL) {
    aux = new node;
    try {
      aux -> info = p -> info;
      aux -> primf = aux -> seggerm = NULL;
      aux -> primf = copia_arbre(p -> primf);
      aux -> seggerm = copia_arbre(p -> seggerm);
    }
    catch (error) {
      destrueix_arbre(aux);
    }
  }
  return aux;
}

// Cost: Θ(n)
template <typename T>
Arbre<T>::Arbre(const Arbre<T> &a) throw(error) {
  _arrel = copia_arbre(a._arrel);
}
```

```
// Cost: Θ(n)
template <typename T>
Arbre<T>&  Arbre<T>::operator=(const Arbre<T> &a)
throw(error) {
  Arbre<T> tmp(a);
  node* aux = _arrel;
  _arrel = tmp._arrel;
  tmp._arrel = aux;
  return *this;
}
```

La destrucció es fa seguint un recorregut en postordre.

```
// Cost: Θ(n)
template <typename T>
void Arbre<T>::destrueix_arbre(node* p) throw() {
  if (p != NULL) {
    destrueix_arbre(p -> primf);
    destrueix_arbre(p -> seggerm);
    delete p;
  }
}

// Cost: Θ(n)
template <typename T>
Arbre<T>::~Arbre() throw() {
  destrueix_arbre(_arrel);
}

// Cost: Θ(1)
template <typename T>
void Arbre<T>::afegir_fill(Arbre<T> &a) throw(error) {
  if (_arrel == NULL or a._arrel == NULL or
    a._arrel -> seggerm != NULL) {
    throw error(ArbreInvalid);
  }
  a._arrel -> seggerm = _arrel -> primf;
  _arrel -> primf = a._arrel;
  a._arrel = NULL;
}
```

```
// Cost: Θ(1)
template <typename T>
typename Arbre<T>::iterador Arbre<T>::arrel() const
throw() {
  iterador it;
  it._p = _arrel;
  return it;
}

// Cost: Θ(1)
template <typename T>
typename Arbre<T>::iterador Arbre<T>::final() const
throw() {
  return iterador();
}


// Cost: Θ(1)
template <typename T>
Arbre<T>::iterador::iterador() : _p(NULL) throw() {
}

// Cost: Θ(1)
template <typename T>
T Arbre<T>::iterador::operator*() const throw(error) {
  if (_p == NULL) {
    throw error(IteradorInvalid);
  }
  return _p -> info;
}

// Cost: Θ(1)
template <typename T>
typename Arbre<T>::iterador Arbre<T>::iterador::primogenit()
const throw(error) {
  if (_p == NULL) {
    throw error(IteradorInvalid);
  }
  iterador it;
  it._p = _p -> primf;
  return it;
}
```

```
// Cost: Θ(1)
template <typename T>
typename Arbre<T>::iterador Arbre<T>::iterador::seg_germa()
const throw(error) {
  if (_p == NULL) {
    throw error(IteradorInvalid);
  }
  iterador it;
  it._p = _p -> seggerm;
  return it;
}

// Cost: Θ(n)
template <typename T>
Arbre<T> Arbre<T>::iterador::arbre() const throw(error) {
  if (_p == NULL) {
    throw error(IteradorInvalid);
  }
  Arbre<T> a;
  a._arrel = _p;
  Arbre<T> aux(a);
  a._arrel = NULL;
  return aux;
}
```

TEMA 6. DICCIONARIS

6.1. CONCEPTES

Volem modelitzar la classe de les funcions:

$$f: K \rightarrow V$$

Dels valors del domini (K) en direm **claus**.

Dels valors de l'abast (V), **informació** o simplement **valors**.

- **Funció total**: Totes les claus tenen informació associada.
- **Funció parcial**: No totes les claus tenen informació associada.

  Una funció parcial pot transformar-se en total associant el valor **indefinit** a totes les claus sense informació associada.

Aquesta classe és coneguda amb el nom de **taula** (anglès: *lookup table*, *symbol table*) o **diccionari**.

També es pot considerar els diccionaris com conjunts de parells clau i valor, amb la restricció que no hi hagi dos parells amb la mateixa clau, de manera que la clau identifica unívocament el parell.

Un diccionari proporciona operacions per tal de localitzar un element donada la seva clau (cerca per clau) i obtenir el seu valor associat.

Per exemple, un diccionari podria guardar la informació dels estudiants d'una escola i accedir a la informació d'un estudiant en concret donat el seu DNI (si s'ha escollit el DNI com la clau per identificar els estudiants).

Els **conjunts** són un cas particular de diccionari. En un conjunt podem considerar que:

- Els elements del conjunt són les claus.
- El seu valor associat és un booleà: cert pels elements del conjunt i fals pels que no hi són.

En el cas dels conjunts, l'operació que consulta el valor associat a una clau és la típica operació que indica si l'element pertany al conjunt.

6.1.1. Classificació dels diccionaris

Podem classificar les classes diccionaris segons les operacions d'actualització permeses. En tots els casos disposem d'una operació per consultar el valor associat a una clau.

- **Estàtic**: Els *n* elements que formaran el diccionari són coneguts. No es permeten insercions o eliminacions posteriors. La classe ofereix una operació per crear un diccionari a partir d'un vector o llista d'elements.

- **Semidinàmic**: Es permet la inserció de nous elements i la modificació d'un valor associat a una clau donada, però no les eliminacions.
- **Dinàmic**: Es permet la inserció de nous elements i la modificació i l'eliminació d'elements existents donada la seva clau.

Especificarem el cas més genèric: els **diccionaris dinàmics**.

6.2. ESPECIFICACIÓ

6.2.1. Especificació bàsica

```
template <typename Clau, typename Valor>
class dicc {
public:
  Constructora. Crea un diccionari buit.
  dicc() throw(error);

  Les tres grans.
  dicc(const dicc &d) throw(error);
  dicc& operator=(const dicc &d) throw(error);
  ~dicc() throw();

  Afegeix el parell <k, v> al diccionari si no hi havia cap
  parell amb la clau k; en cas contrari substitueix el valor
  antic per v.
  void insereix(const Clau &k, const Valor &v)
              throw(error);

  Elimina el parell <k, v> si existeix un parell que té com a
  clau k; no fa res en cas contrari.
  void elimina(const Clau &k) throw();

  Retorna cert si i només si el diccionari conté un parell amb la
  clau donada.
  bool existeix(const Clau &k) const throw();

  Retorna el valor associat a la clau donada en cas que existeixi
  un parell amb la clau k; llança una excepció si la clau no
  existeix.
  Valor consulta(const Clau &k) const throw(error);

private:
  ...
};
```

6.2.2. Operacions addicionals

És freqüent que una classe diccionari tingui altres operacions addicionals. Per exemple:

- Operacions entre dos o més diccionaris: unió, intersecció i diferència de diccionaris.

- Operacions específiques quan les claus són strings: p.e. trobar tots els strings que comencen amb un prefix donat.

- Operacions específiques quan les claus admeten una relació d'ordre total (tenim una operació de comparació entre claus):
    - Operacions per examinar els elements per ordre creixent/decreixent similars a les utilitzades per recórrer les llistes (diccionaris recorribles).
    - Operacions per rang: Consultar l'element i-èssim, eliminar l'element i-èssim, ...
    - Eliminar tots els elements que la seva clau estigui entremig de dos claus K1 i K2 donades, determinar quants elements hi ha amb una clau menor a una clau donada, ...

6.3. DICCIONARIS RECORRIBLES

Una família important de diccionari la constitueixen els anomenats diccionaris **recorribles** o **ordenats**.

Si les claus admeten una relació d'ordre total, és a dir, tenim una operació de comparació < entre claus, pot ser útil que la classe ofereixi operacions que permetin examinar els elements (o parells clau-valor). Examinarem els elements per ordre decreixent de les seves claus.

En una primera implementació dels diccionaris recorribles podem usar un punt d'interès per tal de desplaçar-nos pel diccionari.

```
template <typename Clau, typename Valor>
class diccRecorrible {
public:
  typedef pair<Clau, Valor> pair_cv;
  ...

  Retorna una llista amb tots els parells (clau, valor) del
  diccionari en ordre ascendent.
  void llista_dicc(list<pair_cv> &l) const throw(error);

  Funcions per recórrer les claus en ordre ascendent mitjançant
  un punt d'interès.
  void principi() throw(error);
  void avanca() throw(error);
  pair_cv actual() const throw(error);
  bool final() const throw();

  ...
};
```

Els recorreguts en un diccionari també es poden realitzar mitjançant iteradors.

```
template <typename Clau, typename Valor>
class diccRecorrible {
public:
  typedef pair<Clau, Valor> pair_cv;

  ...

  Iterador del diccionari amb els mètodes habituals (veure
  l'iterador de la classe llista per més informació).
  friend class iterador {
  public:
    friend class diccRecorrible;
    iterador();
    ...

    Accedeix al parell clau-valor apuntat per l'iterador.
    pair_cv operator*() const throw(error);

    Pre- i postincrement; avancen l'iterador.
    iterador& operator++() throw();
    ...

    Operadors de comparació.
    bool operator==(const iterador &it) const throw();
    ...
  };

  Iteradors al principi i al final (sentinella) del diccionari.
  iterador principi() const throw();
  iterador final() const throw();
};
```

6.4. USOS DELS DICCIONARIS

L'aplicació dels diccionaris o taules en el camp de la programació és diversa. Per exemple:

- Taula de símbols (noms de paraules reservades, variables, constants, accions, funcions) d'un compilador.
- Taula per emmagatzemar els nodes d'un graf.
- Taules i índexs utilitzats per un sistema gestor de bases de dades.
- ...

OBJECTIU DELS DICCIONARIS

Així com les estructures lineals estan orientades a l'accés consecutiu a tots els seus elements, els diccionaris estan orientats a l'accés individual als seus elements. L'objectiu és que les operacions *insereix*, *elimina, consulta* i *existeix* tinguin un cost menor de $\Theta(n)$.

6.5. IMPLEMENTACIÓ

- **Vector indexat per les claus**: Si les claus són enteres, el seu nombre no és gaire gran i són consecutives dins d'un rang [ClauMIN .. ClauMAX] pot ser adequat utilitzar un vector indexat per les claus. Cada element del vector emmagatzemarà el valor associat a la clau o el valor indefinit si la clau no s'ha inserit en el diccionari. El cost de totes les operacions, excepte *crea*, seria constant.

- **Llista enllaçada desordenada**: Cada node de la llista conté un parell clau-valor. El cost de les insercions, eliminacions i consultes és $\Theta(n)$, sent n el número d'elements del diccionari. El cost en espai també és $\Theta(n)$ i tots els algorismes són molt simples. És una solució acceptable si els diccionaris són petits. Si cal suportar operacions de recorregut ordenat no seria una opció recomanable.

- **Llista enllaçada desordenada** amb una estratègia d'**autoorganització:** El cost en el cas pitjor és el mateix que l'anterior però el cost mig pot millorar molt si hi ha un grau elevat de localitat de referència en les cerques de les claus (si algunes claus es cerquen molt més sovint que d'altres).

- **Llista ordenada seqüencial** dins d'un **vector**: Només és convenient quan el diccionari és estàtic o les insercions i eliminacions es produeixen molt de tant en tant. Les consultes per clau es poden implementar amb l'algorisme de cerca dicotòmica o binària amb cost $\Theta(\log n)$. També suporta de manera eficient recorreguts ordenats.

- **Llista enllaçada ordenada**: El cost de les insercions, eliminacions i consultes segueix sent $\Theta(n)$ tant en el cas pitjor com en el cas mig. El seu avantatge és que permet implementar fàcilment operacions de recorregut ordenat del diccionari i d'unió, intersecció i diferencia entre diccionaris.
- Altres implementacions que examinarem:
  - **Arbres de cerca** (BSTs Binary Search Trees, AVLs).
  - **Taules de dispersió** (Hashing tables).
  - **Arbres digitals** (tries i variants).

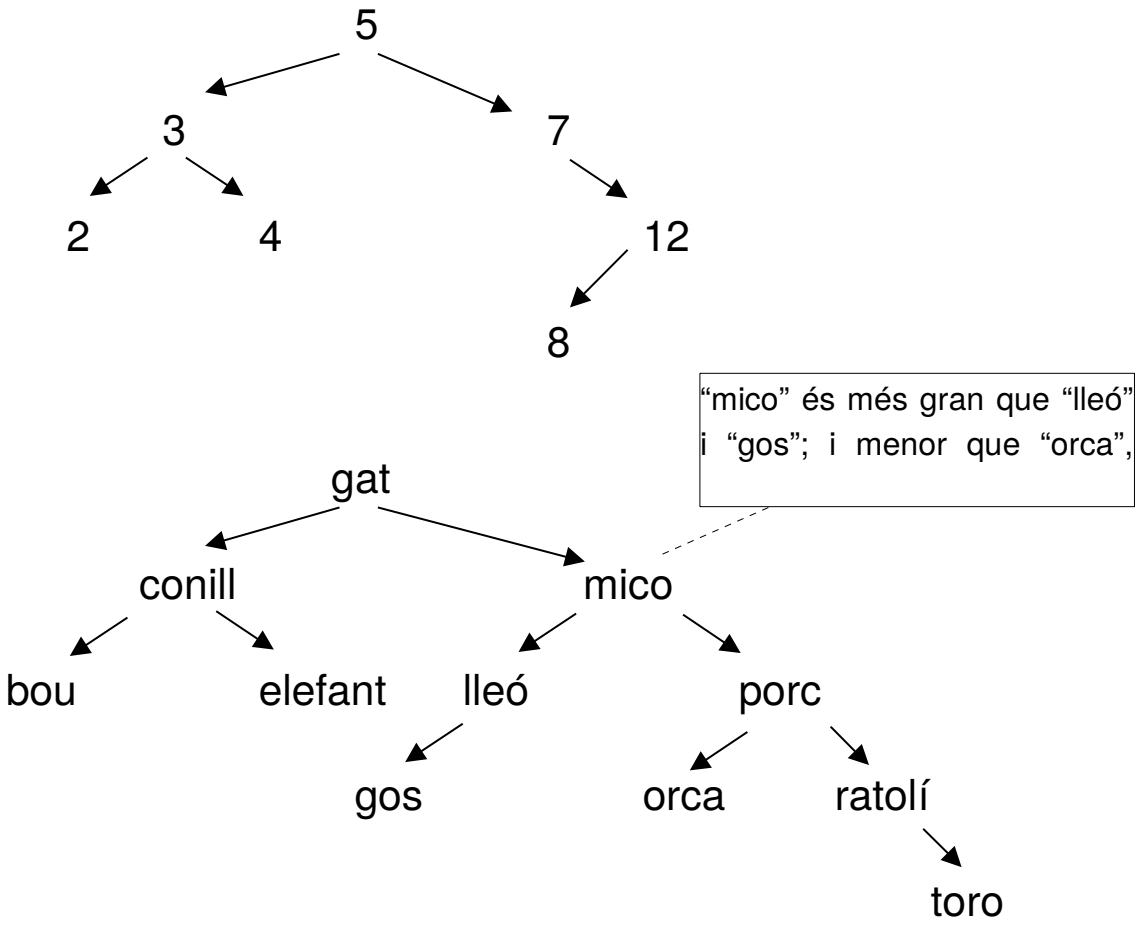
6.6. ARBRES BINARIS DE CERCA (Binary Search Trees)

6.6.1. Definició i exemples

Definició: Un arbre binari de cerca és un arbre binari buit o un arbre binari tal que per a tot node, la clau del node és més gran que qualsevol de les claus del subarbre esquerre i és més petita que qualsevol de les claus del subarbre dret.

- No és necessari que l'arbre binari sigui complet ni ple.
- Quan s'utilitza un BST per implementar un diccionari que guarda parells <clau, valor>, dins de cada node de l'arbre s'emmagatzemarà el valor associat a la clau del node.

Exemples:



6.6.2. Especificació

- Un arbre binari de cerca es pot implementar usant qualsevol de les formes que hem vist per a arbre binaris. Encadenada amb punters és la més usual.

```
template <typename Clau, typename Valor>
class dicc {
public:
  ...
  void insereix(const Clau &k, const Valor &v)
         throw(error);
  void elimina(const Clau &k) throw();
  void consulta(const Clau &k, bool &hi_es, Valor &v)
         const throw(error);
  ...

private:
  struct node {
    Clau _k;
    Valor _v;
    node* _esq;   // fill esquerre
    node* _dret;  // fill dret
    node(const Clau &k, const Valor &v, node* esq = NULL,
          node* dret = NULL) throw(error);
  };
  node *_arrel;

  Mètodes privats.
  static node* consulta_bst(node *n, const Clau &k)
         throw();
  static node* insereix_bst(node *n, const Clau &k,
         Valor &v) throw(error);
  static node* elimina_bst(node *n, const Clau &k)
         throw();
  static node* ajunta(node *t1, node* t2) throw();
  static node* elimina_maxim(node *n) throw();
  ...
};
```

MÈTODES PRIVATS DE CLASSE

Les operacions privades de classe es declaren com a mètodes **static** dins de l'HPP i són operacions que no poden accedir als atributs d'aquesta classe, però si que poden "veure" els tipus privats de la classe. Només hi ha una única instància d'un mètode privat de classe per tots els objectes d'una classe.

6.6.3. Operacions i cost associat

6.6.3.1. **a) Consultar** una clau *k*:

Mirem si *k* coincideix amb la clau que hi ha a l'arrel:

- Si coincideix ja l'hem trobat i la cerca s'acaba.
- Si *k* < que la clau de l'arrel llavors baixem pel fill esquerre. Segons la definició de BST si hi ha algun element la clau del qual sigui *k*, llavors aquest element es troba en el subarbre esquerre.
- Si *k* > que la clau de l'arrel llavors baixem pel fill dret. Anàleg al cas anterior.

El cost és lineal $\Theta(h)$ sent *h* l'alçada de l'arbre. En el cas pitjor *h=n* (això succeeix quan el BST s'ha degenerat perquè s'ha convertit en una llista, per ex. quan tots els nodes són fill dret de l'anterior).

El mètode **consulta** utilitza el mètode **consulta_bst** que a continuació veurem una implementació iterativa i una recursiva.

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::consulta(const Clau &k, bool
&hi_es, Valor &v) const throw(error) {
  node *n = consulta_bst(_arrel, k);
  if (n == NULL) {
    hi_es = false;
  }
  else {
    hi_es = true;
    v = n->_v
  }
}
```

Versió recursiva

Implementació recursiva.
Mètode privat que rep un apuntador *p* a l'arbre a partir del qual s'ha de fer cerca i una clau *k*. Retorna un apuntador NULL si *k* no està present en el BST, o bé, un apuntador que apunta al node del BST que conté la clau *k*.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::consulta_bst(node *n, const Clau &k)
throw() {
  if (n == NULL or n->_k == k) {
    return n;
  }
  else {
    if (k < n->_k) {
      return consulta_bst(n->_esq, k);
    }
    else { // k > p->_k
      return consulta_bst(n->_dret, k);
    }
  }
}
```

Versió iterativa

Donat que el mètode de cerca recursiu és **recursiu final** obtenir una versió iterativa és immediat.

```
Implementació iterativa.
Mètode privat que rep un apuntador p al subarbre a partir del
qual s'ha de fer cerca i una clau k. Retorna un apuntador NULL si
k no està present en el BST, o bé, un apuntador que apunta al
node del BST que conté la clau k.
```

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::consulta_bst(node *n, const Clau &k)
throw() {
  while (n != NULL and k != n->_k ) {
    if (k < n->_k) {
      n = n->_esq;
    }
    else { // k > n->_k
      n = n->_dret;
    }
  }
  return n;
}
```

6.6.3.2. **b)** Obtenir la **llista ordenada** de tots els elements del BST:

Com que fill_esq < node < fill_dret ⇒ Cal recórrer l'arbre en inordre:

- Primer els elements del subarbre esquerre en inordre
- Després l'arrel
- Després els elements del subarbre dret en inordre

El cost en aquest cas és lineal $\Theta(n)$, amb n el número de nodes de l'arbre.

6.6.3.3. **c) Mínim** i **Màxim**

Mínim és el primer element en inordre ⇒ Fill que està més a l'esquerre.

Màxim és l'últim element en inordre ⇒ Fill que està més a la dreta.

El cost és $\Theta(h)$.

6.6.3.4. **d) Inserir** un element

Quan hem d'inserir un nou element, sempre queda incorporat com a fulla de l'arbre. Per saber on és la posició dins de l'arbre hem de baixar des de l'arrel cap a l'esquerra o cap a la dreta depenent del valors que trobem. Se segueix un raonament molt similar al utilitzat per desenvolupar l'algorisme de cerca.

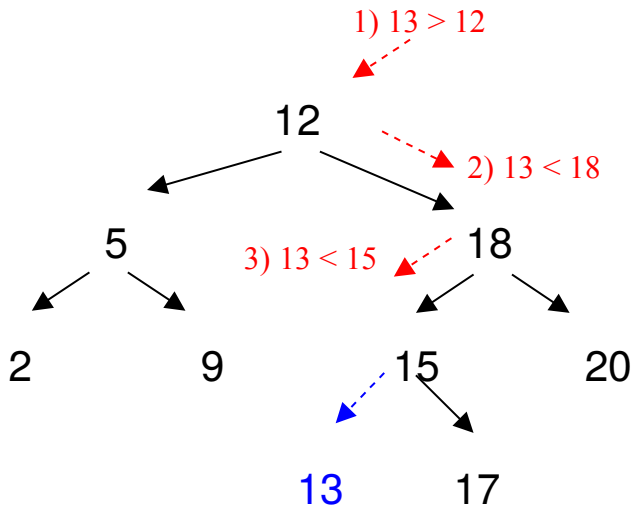
Al final, o bé trobem l'element (llavors no hem d'inserir-lo sinó canviar el valor), o bé trobem el lloc on hem de fer la inserció.

Per exemple, per inserir l'element 13 en el següent arbre on només es mostren les claus es seguirien els següents passos:

1. Es comença per l'arrel, donat que la nova clau és més gran que l'arrel continuarem per l'arbre dret.

2. L'arrel del subarbre dret és més gran que la nova clau (2). Per tant es continua per l'arbre esquerre.

3. Atès que aquest subarbre ja no té fill esquerre per on continuar això vol dir que ja hem arribat a una fulla i és on afegirem el nou element.



Abans de passar a la implementació de les insercions, veurem la implementació (trivial) de la constructora de la classe node.

```
Constructor de la classe node.
Implementem la constructora del node per tal de fer més senzilla
la implementació dels altres mètodes.
```

```
template <typename Clau, typename Valor>
dicc<Clau, Valor>::node::node (const Clau &k,
     const Valor &v, node* esq, node* dret) throw(error) :
     _k(k), _v(v), _esq(esq), _dret(dret) {
}
```

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::insereix(const Clau &k,
const Valor &v) throw(error) {
  _arrel = insereix_bst(_arrel, k, v);
}
```

Versió recursiva

Mètode privat de classe.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::insereix_bst (node *n, const Clau &k,
const Valor &v) throw(error) {
  if (n == NULL) {
    return new node(k, v);
  }
  else {
    if (k < n->_k) {
      n->_esq = insereix_bst(n->_esq, k, v);
    }
    else if (k > n->_k) {
      n->_dret = insereix_bst(n->_dret, k, v);
    }
    else {
      n->_v = v;
    }
    return n;
  }
}
```

Versió iterativa

La versió iterativa és més complexa, ja que a més de localitzar la fulla en la qual s'ha de realitzar la inserció, caldrà mantenir un apuntador *pare* el qual serà el pare del nou node.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::insereix_bst(node *p, const Clau &k,
const Valor &v) throw(error) {
  node *pare = NULL;
  node *p_orig = p;

  // El BST estava buit
  if (p == NULL) {
    p_orig = new node(k, v);
  }
  else {
    // busquem el lloc on inserir el nou element
    while (p != NULL and p->_k != k) {
      pare = p;
      if (k < p->_k) {
        p = p->_esq;
      }
      else {
        p = p->_dret;
      }
    }
    // inserim el nou node com a fulla o modifiquem el
    // valor associat si ja hi havia un node amb la clau
    // donada.
    if (p == NULL) {
      if (k < pare->_k)
        pare->_esq = new node(k, v);
      }
      else {
        pare->_dret = new node(k, v);
      }
    }
    else {
      p->_v = v;      // La clau ja existia
    }
  }
  return p_orig;
}
```

El cost d'inserir un element és $\Theta(h)$.

6.6.3.5. **e) Eliminar** un element

És l'operació més complexa sobre arbre binaris de cerca. S'han de distingir diferents situacions:

1) Eliminar una fulla (un node en el que els dos subarbres són buits) ⇒ Simplement s'elimina de l'arbre

2) Eliminar un node amb un fill ⇒ Enllaçar el pare del node eliminat amb l'únic fill.

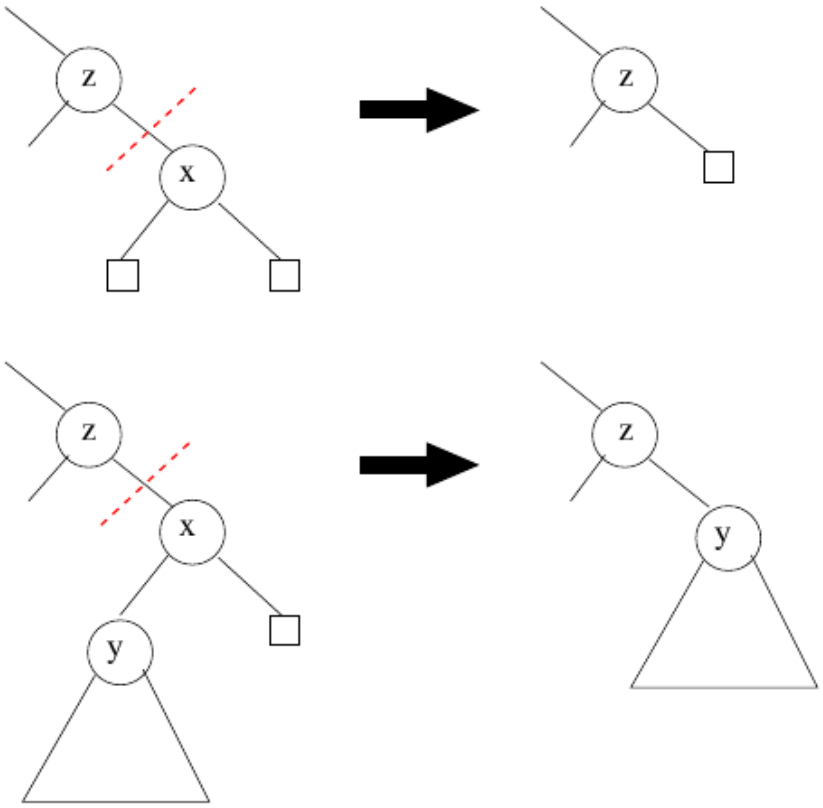


Figura 5.1: Eliminació en un BST,
casos 1 i 2

3) Eliminar un node amb dos fills ⇒ Aquí es poden utilitzar dos mètodes equivalents:

3.1) Amb el successor ⇒ Canviar el node a eliminar pel seu successor i eliminar el successor. El successor és el mínim del subarbre dret i, per tant, serà fàcil eliminar-lo utilitzant el mètode 1 o 2.

3.2) Amb el predecessor ⇒ Canviar el node a eliminar pel seu predecessor i eliminar el predecessor. El predecessor és el màxim del subarbre esquerre i, per tant, serà fàcil eliminar-lo utilitzant el mètode 1 o 2.

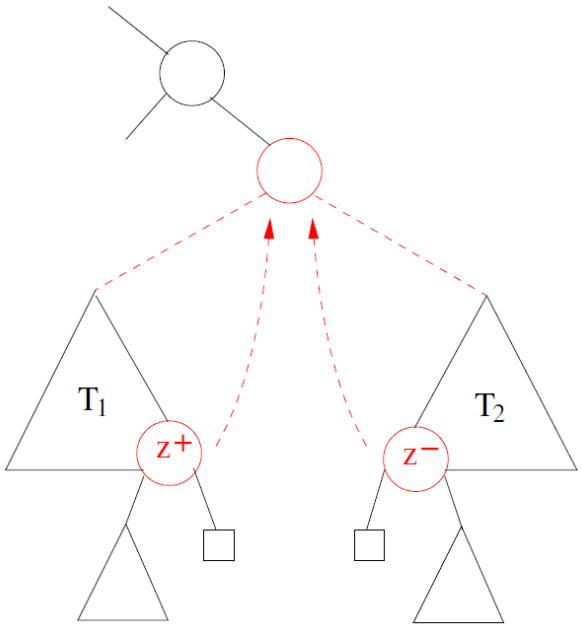


Figura 5.2: Eliminació en un BST,
cas 3

El cost d'eliminar un element és $\Theta(h)$.

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::elimina (const Clau &k) throw() {
  _arrel = elimina_bst(_arrel, k);
}
```

Mètode privat recursiu.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::elimina_bst (node *n, const Clau &k) {
  node *p = n;
  if (n != NULL) {
    if (k < n->_k) {
      n->_esq = elimina_bst(n->_esq, k);
    }
    else if (k > n->_k) {
      n->_dret = elimina_bst(n->_dret, k);
    }
    else {
      n = ajunta(n->_esq, n->_dret);
      delete(p);
    }
  }
  return n;
}
```

Per ajuntar dos BSTs, si un dels dos és buit el resultat és l'altre. Si els dos arbres `t1` i `t2` no són buits utilitzarem la tècnica comentada en el punt 3.2: el màxim del subarbre `t1` (l'esquerre) el col·locarem com arrel, `t1` sense el màxim serà el fill esquerre i `t2` serà el fill dret.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::ajunta (node *t1, node *t2) throw() {
  if (t1 == NULL) {
    return t2;
  }
  if (t2 == NULL) {
    return t1;
  }
  node* p = elimina_maxim(t1);
  p->_dret = t2;
  return p;
}
```

L'acció elimina_màxim elimina el node de clau màxima de l'arbre donat i retorna un punter a aquest node que ha quedat deslligat de l'arbre. Per trobar el màxim de l'arbre cal recórrer els fills drets començant des de l'arrel.

```
Aquest mètode rep un apuntador a l'arrel d'un BST i ens retorna
l'apuntador al nou BST. L'arrel del nou BST és l'element màxim
del BST antic. En el nou BST el subarbre dret és NULL i
l'esquerre és el BST que s'obté d'eliminar l'element màxim.
```

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::elimina_maxim (node* p) throw() {
  node *p_orig = p, *pare = NULL;
  while (p->_dret != NULL) {
    pare = p;
    p = p->_dret;
  }
  if (pare != NULL) {
    pare->_dret = p->_esq;    // p és fill dret de pare
    p->_esq = p_orig;
  }
  return p;
}
```

Experimentalment s'ha comprovat que convé alternar entre el predecessor i el successor del node a eliminar cada vegada que cal eliminar un node del BST. Ho podem fer mitjançant una decisió aleatòria o amb un booleà que canviï alternativament de cert a fals. S'ha observat que, si ho fem així, el BST té els nodes més ben repartits i, per tant, menor alçada.

6.6.4. Altres algorismes sobre BSTs

Els BSTs permeten vàries operacions, sent els algorismes corresponents simples i amb costos mitjos raonables (típicament lineal respecte l'alçada de l'arbre $\Theta(h)$).

Exemple 1: Donades dues claus k1 i k2, k1 $<$ k2, volem implementar una operació que retorni una llista ordenada de tots els elements que tinguin la seva clau k entremig de les dues donades, o sigui k1 $\leq$ k $\leq$ k2. Farem servir la classe list de la biblioteca STL per retornar els elements.

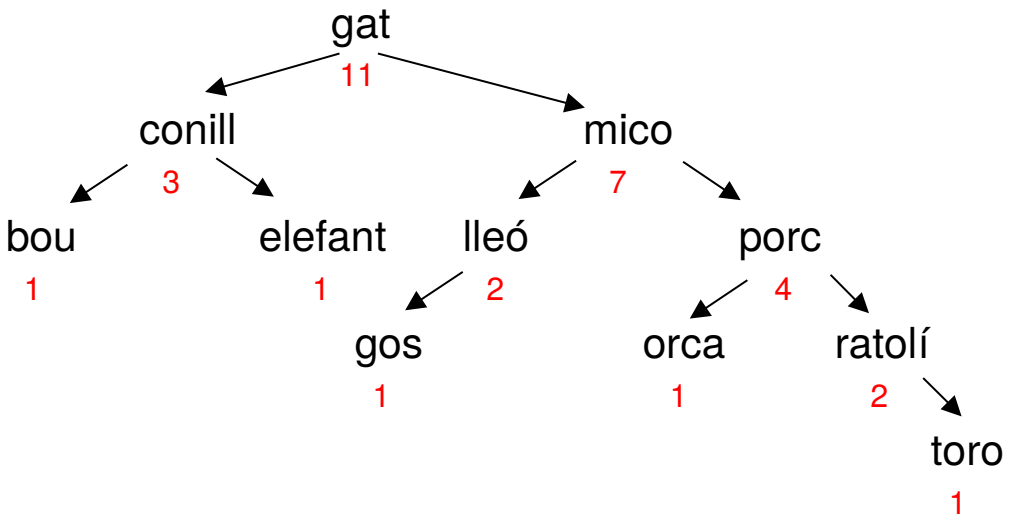
```
Retorna una llista ordenada de tots els elements que tinguin la
seva clau k entremig de les dues donades, o sigui k1 ≤ k ≤ k2.
Suposem que la llista L és buida.
```

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::llista_interval(const Clau &k1,
const Clau &k2, list<Valor> &L) const throw(error) {
  rllista_interval(_arrel, k1, k2, L);
}
```

```
template <typename Clau, typename Valor>
static void dicc<Clau, Valor>::rllista_interval(node *n,
const Clau &k1, const Clau &k2, list<Valor> &L)
throw(error) {
  if (n != NULL) {
    if (k1 <= n->_k) {
      rllista_interval(n->_esq, k1, k2, L);
    }
    if (k1 <= n->_k and n->_k <= k2) {
      L.push_back(n->_v);
    }
    if (n->_k <= k2) {
      rllista_interval(n->_dret, k1, k2, L);
    }
  }
}
```

Exemple 2: Consulta o eliminació d'un element del BST per rang (per exemple buscar el 5è element del diccionari).

Cal fer una lleugera modificació de l'estructura de dades del BST: afegir un enter a cada node que guardi la grandària del subarbre que penja d'ell. Per localitzar un element per rang, aquest enter que hem afegit permet decidir recursivament per quina branca s'ha de continuar la cerca.



Si cerquem el 4at element en l'exemple anterior, com que a l'esquerra en hi ha tres (conill conté un 3), el 4at element serà el gat. Si cerquem l'animal amb la posició<4 caldrà baixar pel fill esquerra i si cerquem l'animal amb posició>4 caldrà baixar pel fill dret. Observeu que si baixem pel fill dret caldrà cercar l'element de la posició = posició_anterior - #elements_subarbre_esquerre - 1. Per exemple, si cerquem el 6è element, baixarem pel fill dret i cercarem el 2on element del subarbre que penja del mico (posició = 6 - 3 - 1 = 2).

Quan inserim/eliminem un element caldrà incrementar/decrementar el nombre d'elements per a tots els nodes que formen el camí que s'ha seguit durant la cerca.

6.7. ARBRES BINARIS DE CERCA EQUILIBRATS (AVL's)

6.7.1. Definició i exemples

- El cost de la majoria d'operacions dels Arbres Binaris de Cerca són $\Theta(h)$. En el pitjor cas, un Arbre Binari de Cerca de n elements pot tenir una alçada $h = n$. Llavors totes les operacions tenen cost lineal $\Theta(n)$ sent n el número de nodes.
- Per a millorar aquest cost tenim dues possibilitats:

  a) No fer rés perquè es pot demostrar que si en un Arbre Binari de Cerca introduïm els elements de manera aleatòria l'alçada de l'arbre és $\Theta(\log n)$.

  b) Podem forçar que les insercions i supressions dins de l'arbre mantinguin l'alçada en log n. Es tracta d'aconseguir que els elements de l'arbre estiguin repartits d'una forma més o menys equilibrada. Si ho aconseguim, l'alçada serà log n i, per tant, el cost de les operacions serà logarítmic. Aquest és l'objectiu dels Arbres Binaris de Cerca Equilibrats.

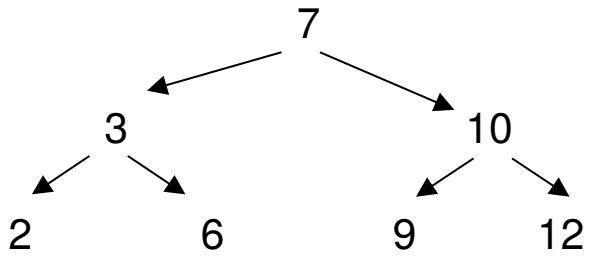
Definició: Un **Arbre Binari de Cerca Equilibrat** és un arbre binari buit o un arbre binari tal que per a tot node es compleix que el factor d'equilibri és ≤ 1.

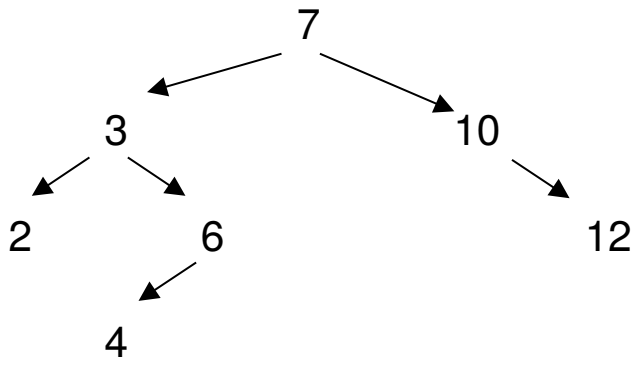
- Factor d'equilibri d'un node =

  | alçada (fill_esquerre) – alçada (fill_dret) |

- La diferència màxima entre les alçades dels subarbres d'un node és $\leq$ 1.

Exemples:





6.7.2. Inserció en un arbre AVL

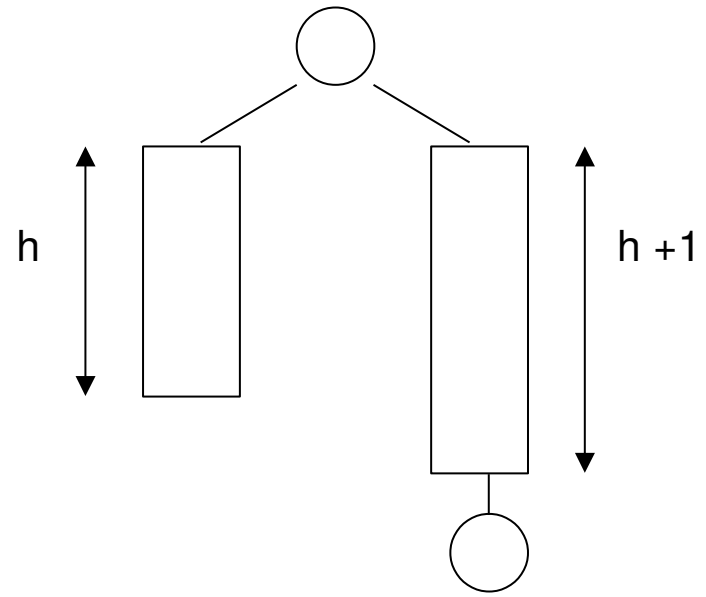
- L'operació d'inserció té dues etapes:

    - D'una banda s'ha d'inserir tenint en compte que és un arbre de cerca ⇒ si el node és més petit que l'arrel baixar per l'esquerra sinó per la dreta.

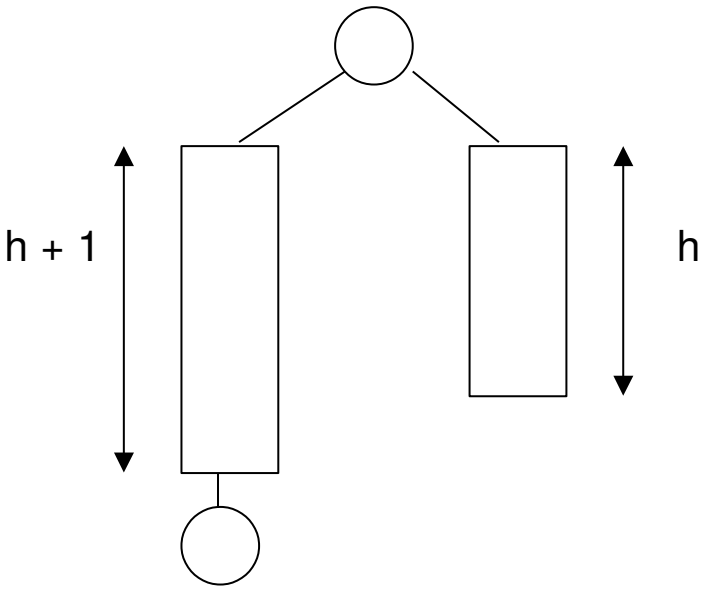
    - D'altra banda, cal assegurar que l'arbre queda equilibrat, reestructurant-lo si cal.

- El desequilibri es produeix quan existeix un subarbre A de l'arbre que es troba en qualsevol del dos casos següents:

A) el subarbre dret de A té una alçada superior en una unitat al subarbre esquerre de A i el node corresponent s'insereix al subarbre dret de A, llavors provoca un increment en 1 de la seva alçada.

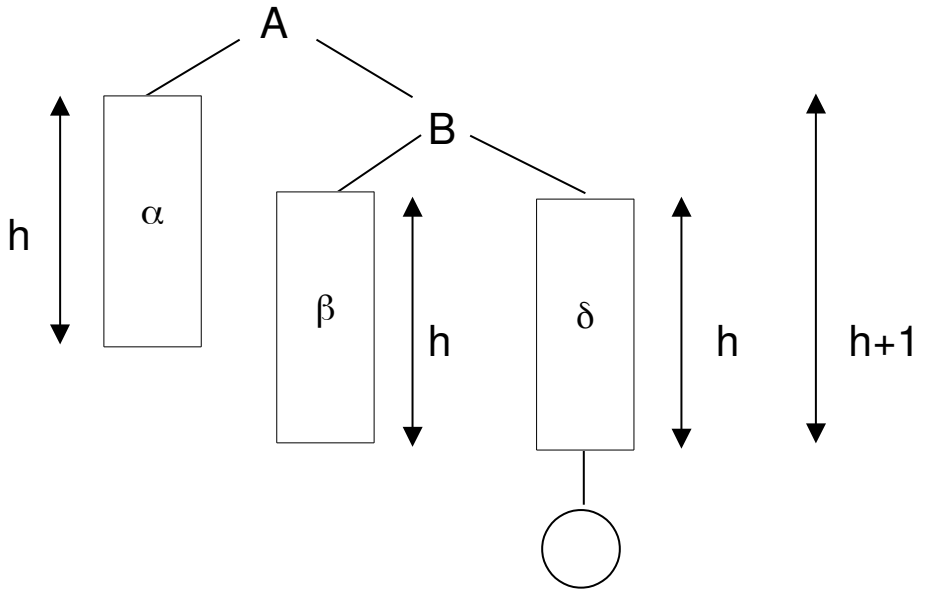


B) El subarbre esquerre de A té una alçada superior en una unitat al subarbre dret de A i el node corresponent s'insereix al subarbre esquerre de A, llavors provoca un increment en 1 de la seva alçada.



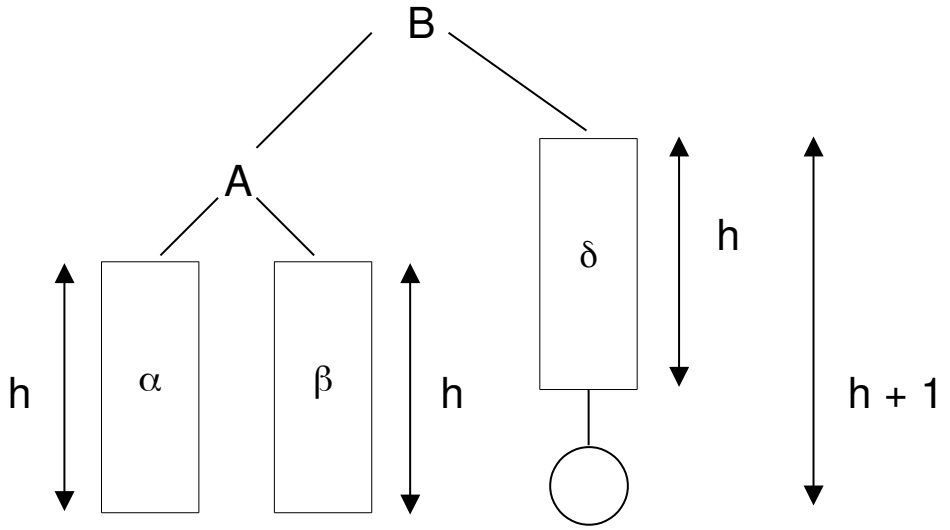
- Notem que ambdues situacions són simètriques i, per això ens centrarem només en la primera. Hi ha dos subcasos:
    - Cas DD (Dreta-Dreta)
    - Cas DE (Dreta- Esquerre)

- Cas Dreta-Dreta: el node s'insereix en el subarbre dret del subarbre dret.



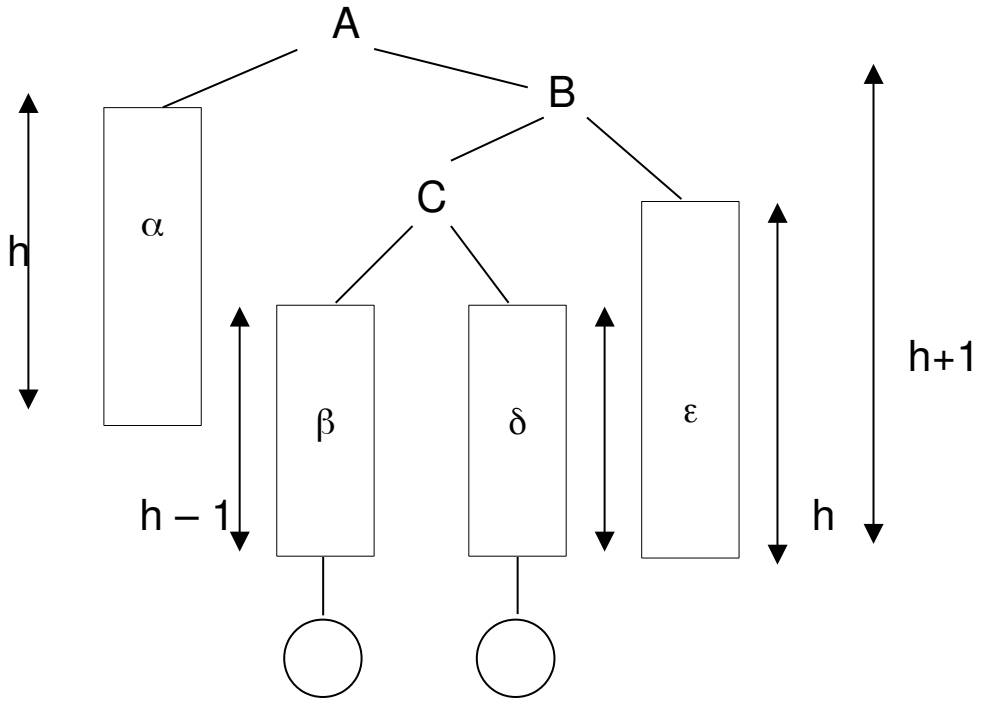
Recorregut en inordre : $\alpha A \beta B \delta$

<u>Reestructuració:</u>



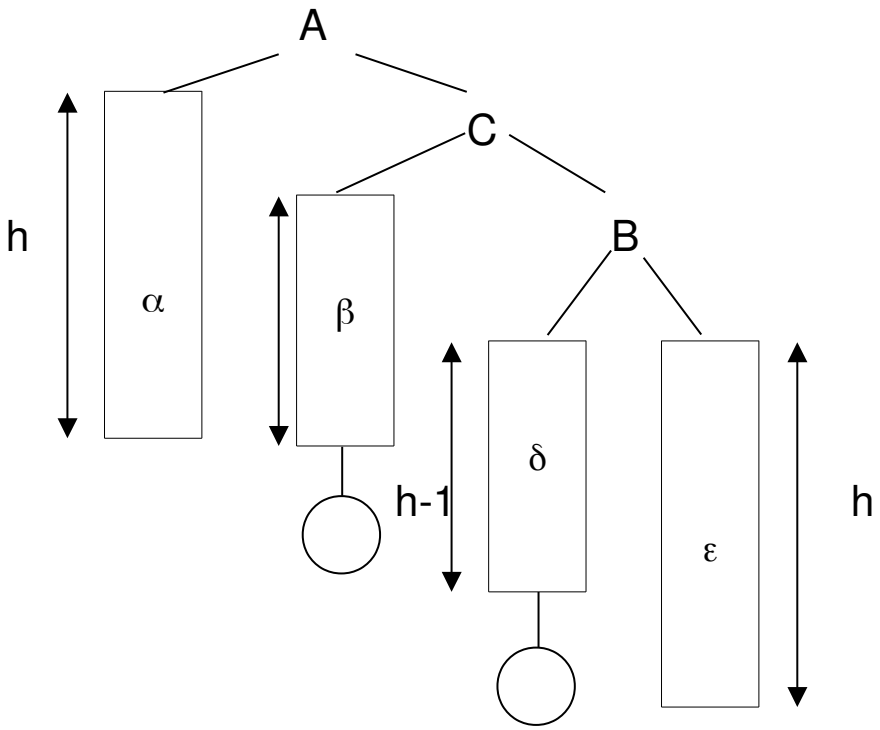
Recorregut en inordre: $\alpha A \beta B \delta$

- Cas Dreta-Esquerre: el node s'insereix en el subarbre esquerre del subarbre dret.



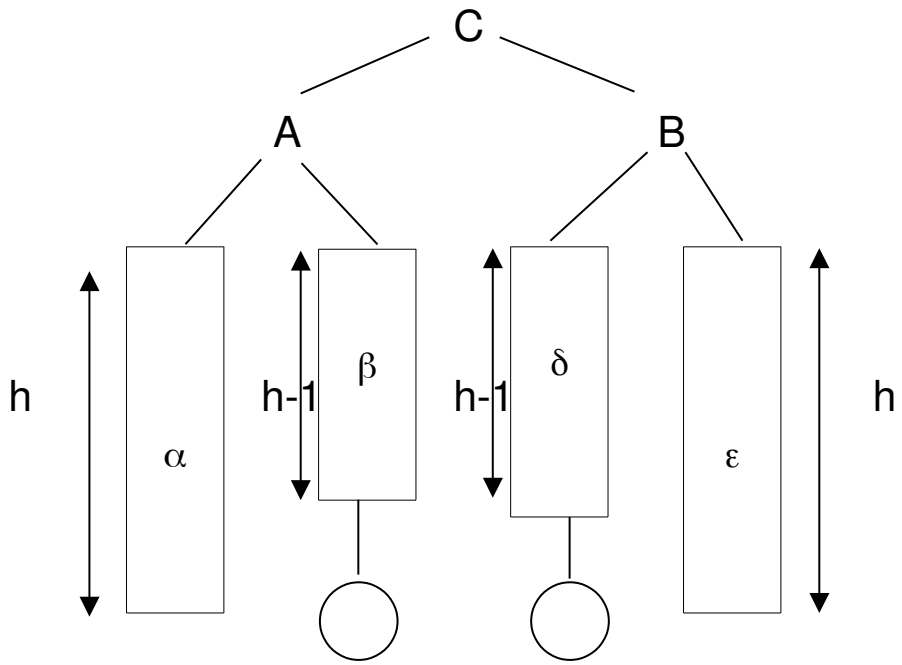
Recorregut en inordre : αAβCδBε

<u>Reestructuració: (Pas 1)</u>



Recorregut en inordre : αAβCδBε

Reestructuració: (Pas 2)



Recorregut en inordre : $\alpha A \beta C \delta B \varepsilon$

6.7.3. Supressió en un arbre AVL

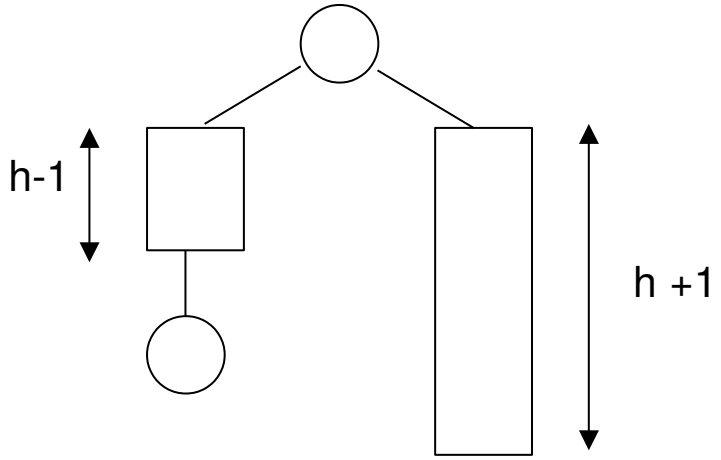
- Els dos cassos possibles de desequilibri a la supressió són equivalents al procés d'inserció però ara el desequilibri es produeix perquè l'alçada d'un subarbre disminueix per sota del màxim tolerat. Els casos són:

A) El subarbre dret d'A té una alçada superior en una unitat al subarbre esquerre d'A i el node corresponent s'elimina del subarbre esquerre.

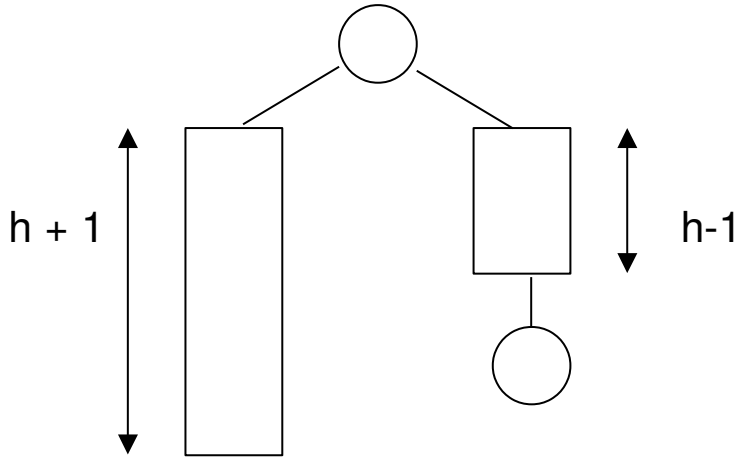
B) El subarbre esquerre d'A té una alçada superior en una unitat al subarbre dret d'A i el node corresponent s'elimina del subarbre dret, llavors provoca un decrement en 1 de la seva alçada.

Ara els veurem amb detall.

A) el subarbre dret d'A té una alçada superior en una unitat al subarbre esquerre d'A i el node corresponent s'elimina del subarbre esquerre, llavors provoca un decrement en 1 de la seva alçada.

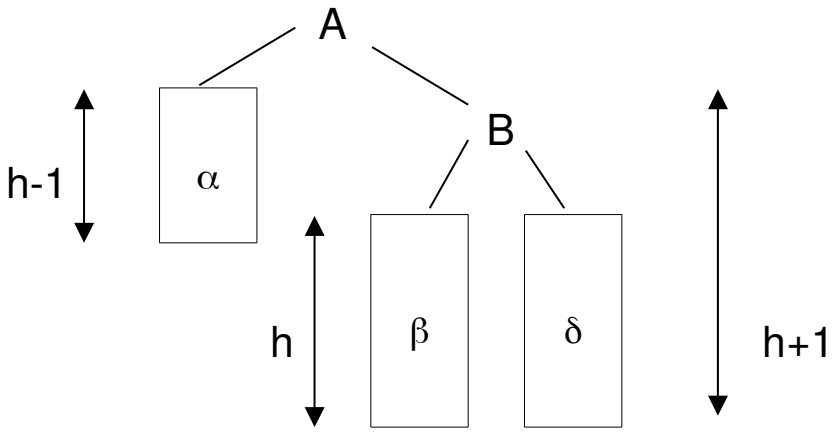


B) El subarbre esquerre de A té una alçada superior en una unitat al subarbre dret de A i el node corresponent s'elimina del subarbre dret, llavors provoca un decrement en 1 de la seva alçada.

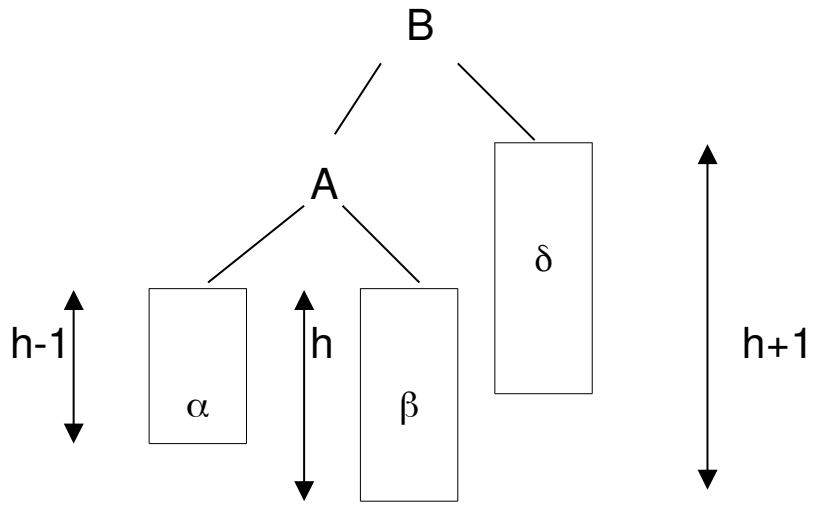


- Notem que ambdues situacions són simètriques i, per això ens centrarem només en la primera. Hi ha tres subcasos:

1) Sigui B el subarbre dret de A. Si els dos subarbres de B són de la mateixa alçada ens trobem amb el desequilibri Dreta-Dreta del cas de la inserció, que es resol de la mateixa manera.

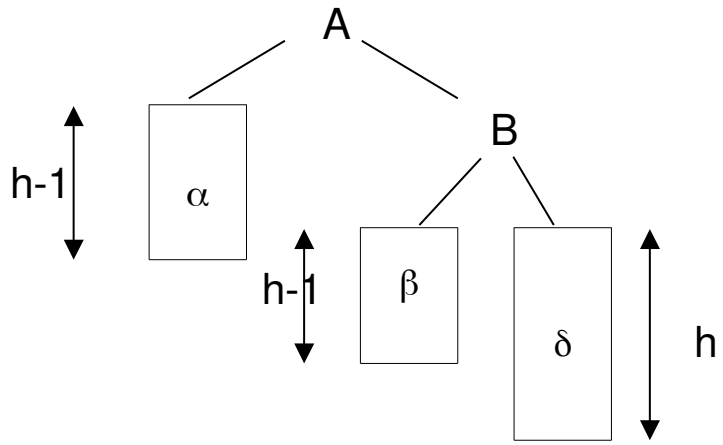


Reestructuració:

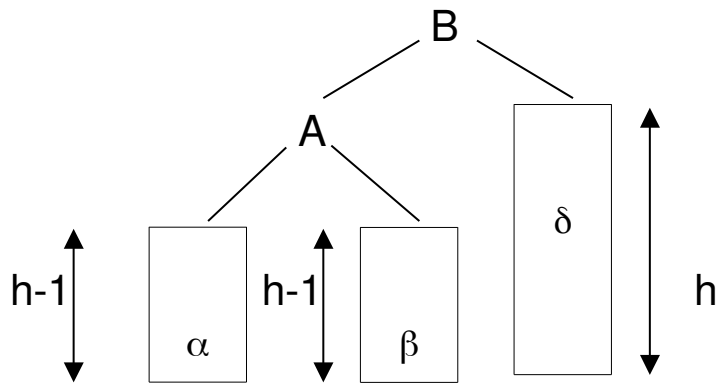


L'arbre resultant té la mateixa alçada abans i després de la supressió, per la qual cosa n'hi ha prou amb aquesta rotació per restablir l'equilibri.

2) Sigui B el subarbre dret de A. Si la alçada del subarbre esquerre de B és menor que la alçada del subarbre dret, la rotació és exactament la mateixa que abans.

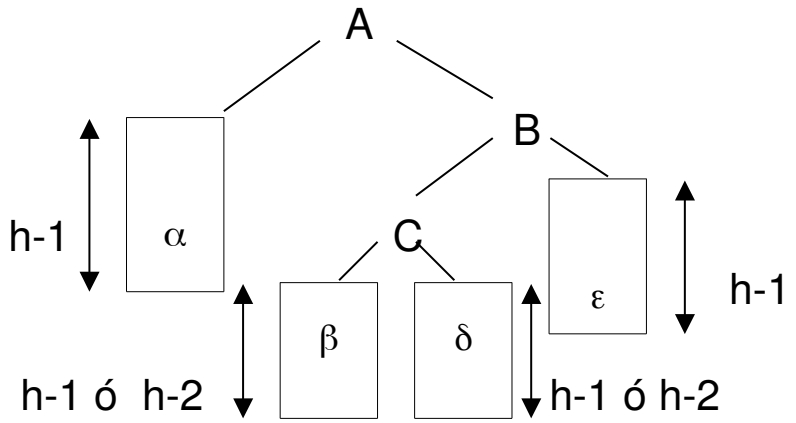


Reestructuració:

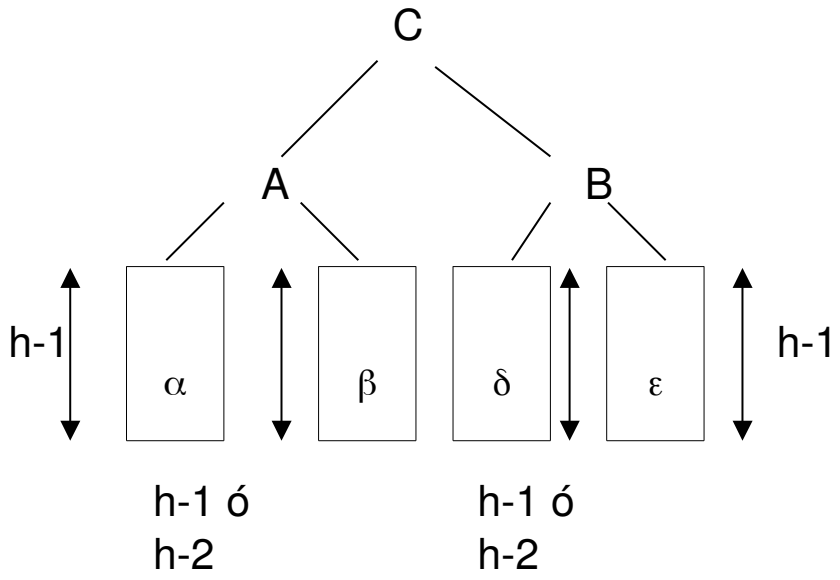


L'alçada de l'arbre resultat és una unitat més petita que abans de la supressió. Aquest fet és rellevant, perquè obliga a examinar si algun subarbre que l'engloba també es desequilibra.

3) Sigui B el subarbre dret de A. Si la alçada del subarbre esquerre de B és major que la alçada del subarbre dret, la rotació és similar al cas Dreta-Esquerre.



Reestructuració:



També aquí l'alçada de l'arbre resultat és una unitat més petita que abans de la supressió. Aquest fet és rellevant, perquè obliga a examinar si algun subarbre que l'engloba també es desequilibra.

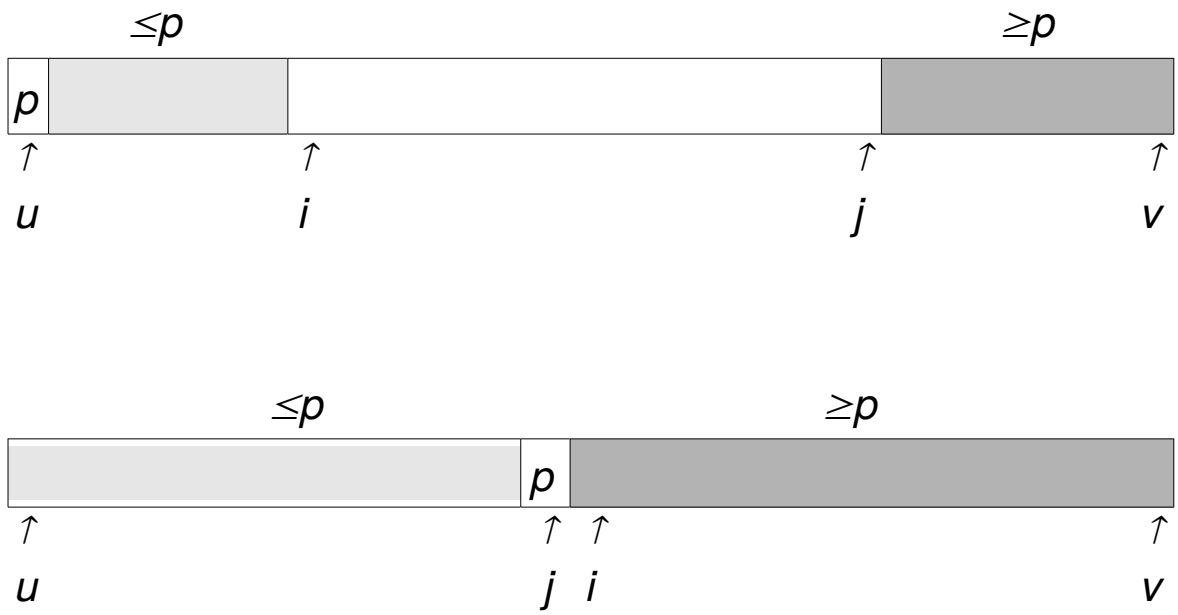
6.8. ALGORISME D'ORDENACIÓ QUICKSORT

6.8.1. Introducció

Quicksort és un algorisme que utilitza el principi de divideix i venceràs, però a diferència d'altres algorismes similars (per ex. mergesort), no assegura que la divisió es faci en parts de mida similar.

La tasca important del *quicksort* és la partició. Donat un element *p* (per exemple el primer) anomenat **pivot**, es reorganitzen els elements col·locant a l'esquerra del pivot els més petits i a la dreta els més grans.

Evolució de la partició:



Després només caldrà ordenar els trossos a l'esquerra i a la dreta del pivot fent dues crides recursives a *quicksort*.

La fusió no és necessària, doncs els trossos a ordenar ja queden ben repartits a cada banda del pivot.

Mentre que en el *mergesort* la partició és simple i la feina es realitza durant l'etapa de fusió, en el *quicksort* és tot el contrari.

6.8.2. Implementació

Acció per ordenar el tros de vector A[u..v]

```
template <typename T>
void quicksort(T A[], nat u, nat v) {
  if (v-u+1 <= M) {
    //utilitzar un algorisme d'ordenació simple
  }
  else {
    nat k = particio(A, u, v);
    quicksort(A, u, k-1);
    quicksort(A, k+1, v);
  }
}
```

Com a algorisme d'ordenació simple es pot utilitzar per ex. el d'ordenació per inserció. Una bona alternativa és, enlloc d'ordenar per inserció cada tros de *M* o menys elements dins del mètode *quicksort*, ordenar per inserció tot el vector sencer al final de tot:

```
quicksort(A, 0, n);
ordena_insercio(A, 0, n);
```

Com que el vector A està quasi ordenat després d'aplicar *quicksort*, l'ordenació per inserció té un cost $\Theta(n)$, sent *n* el nombre d'elements a ordenar. S'ha estimat que l'elecció òptima pel llindar *M* és entre 20 i 25.

Hi ha moltes maneres de fer la partició. En Bentley & McIlroy (1993) es presenta un procediment de partició molt eficient, fins i tot si hi ha elements repetits. Nosaltres veurem un algorisme bàsic però raonablement eficaç.

- S'agafa com a pivot *p* el primer element A[*u*].
- Es mantenen dos índexs *i* i *j* de forma que A[*u*+1..*i*-1] conté els elements menors o iguals que el pivot i A[*j*+1..*v*] conté els elements majors o iguals.
- Els índexs *i* i *j* recorren el vector d'esquerra a dreta i de dreta a esquerra respectivament fins que A[*i*]>*p* i A[*j*]<*p* o es creuen (*i*=*j*+1). En el primer cas s'intercanvien els elements A[*i*] i A[*j*] i es continua.

```
template <typename T>
nat particio(T A[], nat u, nat v) {
  nat i, j;
  T p = A[u];
  i = u+1;
  j = v;
  while (i < j+1) {
    while (i<j+1 and A[i]<=p) {
      ++i;
    }
    while (i<j+1 and A[j]>=p) {
      --j;
    }
    if (i < j+1){
      T aux = A[j];   // intercanvi
      A[j] = A[i];
      A[i] = aux;
    }
  }
  T aux = A[u];  // intercanvi pivot i últim dels menors
  A[u] = A[j];
  A[j] = aux;
  return j;
}
```

6.8.3. Cost

El cost de *quicksort* en cas pitjor és $\Theta(n^2)$. Això passa si en la majoria de casos un dels trossos té molts pocs elements i l'altre els té gairebé tots. És el cas de si el vector ja estava ordenat creixentment o decreixentment.

Tanmateix, si ordenem un vector desordenat, normalment el pivot quedarà més o menys centrat cap a la meitat del vector. Si això succeeix en la majoria d'iteracions, tindrem un cas similar al de *mergesort*: el cost d'una iteració de *quicksort* (l'acció *partició*) és lineal i com que es fan dues crides amb trossos més o menys la meitat de l'original resulta un cost total quasi-lineal $\Theta(n \log n)$.

6.9. TAULES DE DISPERSIÓ

6.9.1. Definició

Una **taula de dispersió** (anglès: *hash table* o *taula d'adreçament calculat*) emmagatzema un conjunt d'elements identificats amb una clau:

- dins d'una taula D[0..M-1]
- mitjançant una **funció de dispersió** *h* que va del conjunt de claus K fins al conjunt de posicions de la taula D:

claus	→	valors de dispersió (posicions taula)
h: K	→	0..M-1

Idealment la funció de dispersió *h* hauria de ser injectiva (a cada clau li correspondria una posició de la taula diferent). Normalment això no serà així i a claus diferents els hi pot correspondre la mateixa posició de la taula.

Donades dues claus *x* i *y* diferents, es diu que *x* i *y* són **sinònims** o que han produït una **col·lisió** si $h(x)=h(y)$.

Si la funció de dispersió dispersa eficaçment hi hauran poques col·lisions (la probabilitat de que moltes claus tinguin el mateix valor de dispersió serà baixa) i la idea de les taules de dispersió seguirà sent bona.

Per implementar eficaçment les taules de dispersió cal resoldre dues qüestions:

1. Dissenyar funcions de dispersió *h* que dispersin eficaçment i siguin relativament ràpides de calcular.
2. Definir estratègies per resoldre les col·lisions que apareguin.

A continuació veurem com resoldre aquests dos punts.

El **factor de càrrega (α)** (anglès: *load factor*) és una mesura de l'ocupació de la taula de dispersió. Quan se supera aquest llindar caldria aplicar una **redispersió** en la taula per tal d'incrementar la seva mida (per més informació veure l'apartat 6.9.5 Redispersió).

El càlcul d'aquest indicador és el següent:

$$\alpha = \frac{nombre\ actual\ d'\ elements}{mida\ de\ la\ taula}$$

El nombre esperat de claus de la taula de dispersió i el factor de càrrega haurien de tenir-se en compte per inicialitzar la capacitat inicial de la taula per minimitzar el nombre de redispersions.

Si la capacitat inicial és més gran que el nombre de claus dividit pel factor de càrrega, no seran necessàries operacions de redispersió.

6.9.2. Especificació

Atès que es desconeix el tipus de la clau cal encapsular la funció de dispersió dins d'una classe Hash (que s'ha d'implementar per cada tipus de Clau)

```
template <typename T>
class Hash {
public:
  int operator()(const T &x) const throw();
};
```

COMPTE!!

La classe **Hash<T>** defineix l' **operator( )** de manera que si h és un objecte de classe Hash<T> i x és un objecte de classe T, podem aplicar **h(x)**. Aquesta crida ens retornarà un nombre enter.

```
template <typename Clau, typename Valor,
          typename HashFunct = Hash>
class dicc {
public:
  ...
  void consulta(const Clau &k, bool &hi_es, Valor &v)
       const throw(error);
  void insereix(const Clau &k, const Valor &v)
       throw(error);
  void elimina(const Clau &k) throw();
  ...

private:
  struct node_hash {
    Clau _k;
    Valor _v;
    ...
  };

  node_hash *_taula; // taula amb les parelles
                     // clau-valor
  //ó node_hash **_taula; // taula de punters a node

  nat _M;         // mida de la taula
  nat _quants;    // nº d'elements guardats al diccionari

  static int hash(const Clau &k) throw() {
    HashFunct<Clau> h;
    return h(k) % _M;
  }
};
```

UN BON FACTOR DE CÀRREGA

Com a regla general, un valor per defecte de factor de càrrega és 0.75 ja que ofereix un bon equilibri entre els costos de temps i espai.

Valors més alts fan decréixer la sobrecàrrega de l'espai però incrementa el cost de la cerca.

6.9.3. Funcions de dispersió

La funció de dispersió ha de complir les següents propietats:

- **Distribució uniforme**: Volem que tots els valors de dispersió tinguin el mateix nombre de claus associades.
- **Independència de l'aparença de la clau**: Ens interessa que claus similars no tinguin valors de dispersió iguals.
- **Exhaustivitat**: Volem que tot valor de dispersió tingui com a mínim una clau associada. Així cap posició de la taula de dispersió queda desaprofitada a priori.
- **Rapidesa de càlcul**.

La construcció de bones funcions de dispersió no és fàcil i requereix coneixements sòlids de vàries especialitats de les matemàtiques. Està molt relacionada amb la construcció de generadors de números pseudoaleatoris.

Una manera senzilla de construir funcions de dispersió i que dóna bons resultats és:

1. Calcular un número enter positiu a partir de la representació binaria de la clau (per ex. sumant el bytes que la composen).
2. Al resultat anterior fer mòdul M, sent M la grandària de la taula. Així obtenim una natural dins del rang [0..M-1]. Es recomana que M sigui un número primer.

L'operació privada **hash** de la classe dicc calcularà el mòdul mitjançant:

```
h(x) % _M
```

de manera que obtindrem un índex vàlid de la taula, entre 0 i _M-1.

Per tal que la funció de dispersió *h* dispersi millor es pot sofisticar una mica més utilitzant alguna de les següents estratègies:

- **Suma ponderada** de tots els bytes/caràcters/dígits:

  ```
  s = s + codi_ascii(s[i]) * b^i
  // b és la base utilitzada
  ```

  Això evita que el valor d'un byte sigui el mateix independent de la posició on aparegui dins la clau.

  Però és més ineficient doncs es necessita calcular una potència i un producte. Una bona opció és agafar b=2.

- **Desplegament**: Es parteix la clau K en *m* parts de la mateixa longitud i es combinen aquestes parts de determinada manera (sumes, OR lògic, XOR lògic). Finalment es fa mòdul M.

  Exemple:

  K=123456 *m*=3 *hash*(K)=(12+34+56) mod M

- **Quadrat**: S'eleva al quadrat el número representat pels *m* bits/bytes centrals de la clau K. Finalment es fa mòdul M.

  Si M és una potència de 2, es pot agafar directament els $\log_2$ M bits centrals de K^2.

  Exemple: K=12 K^2 = 144 = 10010000
  si M=32 agafarem els 5 bits centrals: 01000

  Aquest és un exemple d'una transformació no lineal.

Caldria tenir definit una especialització de la classe Hash per la nostra classe Clau. Per exemple si la classe Clau fos igual a un string o un enter unes possibles especialitzacions serien:

```
Especialització del template Hash per T = string:
Suma no ponderada dels caràcters.
```

```
template <>
class Hash<string> {
public:
  int operator()(const string &x) const throw() {
    nat n = 0;
    for (nat i=0; i < x.length(); ++i) {
      n = n + x[i]*i; // a n sumen el codi ascii
    }
    return n;
  }
};
```

```
Especialització del template Hash per T = int:
Bits centrals del quadrat del número multiplicat per PI.
```

```
template <>
class Hash<int> {
static long const MULT = 31415926;
public:
  int operator()(const int &x) const throw() {
    long y = ((x * x * MULT) << 20) >> 4;
    return y;
  }
};
```

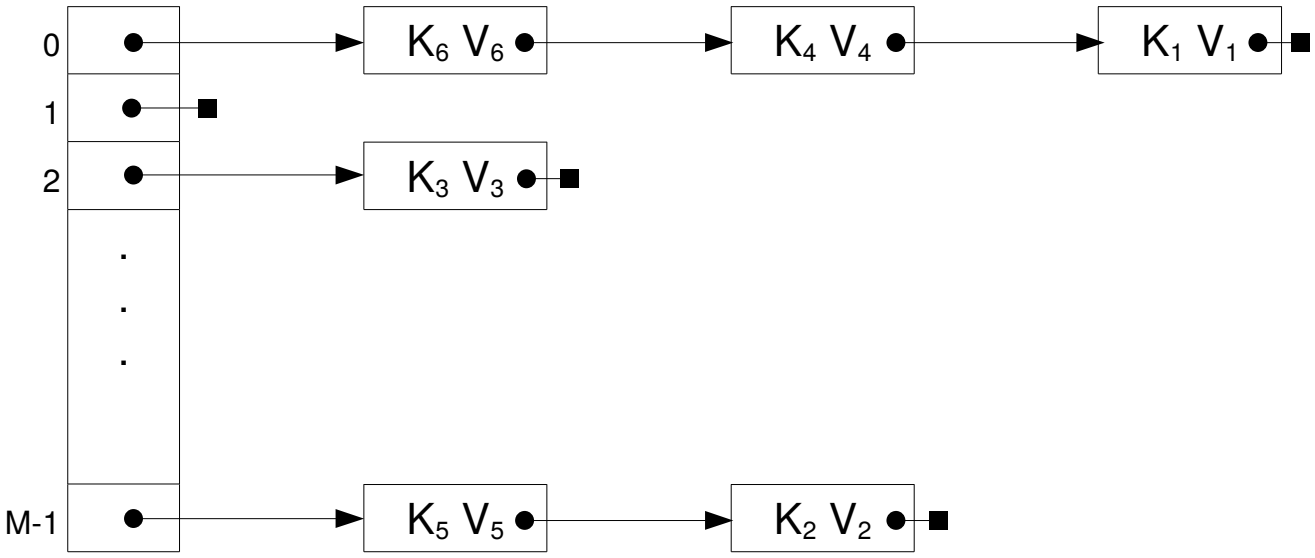
6.9.4. Estratègies de resolució de col·lisions

Existeixen dues grans famílies per organitzar una taula de dispersió que resolgui les col·lisions. Per cadascuna d'elles estudiarem una estratègia en concret:

- **Dispersió oberta** (anglès: *open hashing*)
  - **Sinònims encadenats** (anglès: *separate chaining*) indirectes i directes: Els sinònims es guarden formant llistes encadenades.
- **Direccionament obert** (anglès: *open addressing*)
  - **Sondeig lineal** (anglès: *linear probing*): Si una component de la taula ja està ocupada es busca una altra que estigui lliure. El sondeig lineal és la tècnica més senzilla: busca els llocs lliures de la taula de forma correlativa.

6.9.4.1. Sinònims encadenats indirectes

Cada entrada a la taula apunta a una llista encadenada de sinònims. S'anomena indirecte perquè la taula no guarda els elements en si, sinó un enllaç (punter) al primer element de la llista de sinònims.



Nota: Els índexs *i* de K_i V_i indiquen un possible ordre d'inserció.

La representació d'aquesta classe és la següent:

```
template <typename Clau, typename Valor>
class dicc {
  ...
private:
  struct node_hash {
    Clau _k;
    Valor _v;
    node_hash* _seg;
    node_hash(const Clau &k, const Valor &v,
         node_hash* seg = NULL) throw(error);
  };
  node_hash **_taula;  // taula amb punters a les llistes
  nat _M;  // mida de la taula
  nat _quants;  // nº d'elements guardats al diccionari
  ...
};
```

Una possible implementació d'aquesta classe es pot veure a continuació.

```
template <typename Clau, typename Valor, typename
HashFunct>
dicc<Clau, Valor, HashFunct>::dicc() throw(error) :
_quants(0) {
  _M = 51;
  _taula = new node_hash*[_M];
  for (int i=0; i < _M; ++i) {
    _taula[i] = NULL;
  }
}
```

COMPTE!!

La mida inicial de la taula és 51 ja que no hi ha una estimació del nombre de claus de la taula. En aquest cas cal implementar la redispersió per tal que la classe sigui completament funcional.

Constructora del node.

```
template <typename Clau, typename Valor, typename HasFunct>
dicc<Clau, Valor, HashFunct>::node_hash::node_hash (const Clau &k, const Valor &v, node_hash* seg) throw(error)
    : _k(k), _v(v), _seg(seg) {
}
```

COMPTE!!

Cal implementar-lo ja que es desconeix si les classes Clau i Valor tenen constructor per defecte.

La consulta d'un element és ben simple: s'accedeix a la llista apropiada mitjançant la funció de hash, i es realitza un recorregut seqüencial de la llista fins que es troba un node amb la clau donada o s'ha examinat tota la llista.

```
template <typename Clau, typename Valor, typename HasFunct>
void dicc<Clau, Valor, HasFunct>::consulta (const Clau &k, bool &hi_es, Valor &v) const throw(error) {
  int i = hash(k);
  node_hash* p = _taula[i];
  hi_es = false;
  while (p != NULL and not hi_es) {
    if (p->_k == k) {
      hi_es = true;
      v = p->_v;
    }
    else {
      p = p->_seg;
    }
  }
}
```

Per fer la inserció s'accedeix a la llista corresponent mitjançant la funció de hash, i es recorre per determinar si ja existia o no un element amb la clau donada. En el primer cas, es modifica el valor associat; i en el segon s'afegeix un nou node a la llista.
Donat que les llistes de sinònims contenen generalment pocs elements el més efectiu és efectuar les insercions a l'inici.

```
template <typename Clau, typename Valor, typename HasFunct>
void dicc<Clau, Valor>::insereix (const Clau &k,
const Valor &v) throw(error) {
  int i = hash(k);
  node_hash *p = _taula[i];
  bool trobat = false;
  while (p != NULL and not trobat) {
    if (p->_k == k) {
      trobat = true;
    }
    else {
      p = p->_seg;
    }
  }
  if (trobat) {
    // Només canviem el valor associat
    p->_v = v;
  }
  else {
    // Cal crear un nou node i l'afegim al principi
    _taula[i] = new node_hash(k, v, _taula[i]);
    ++_quants;
  }
}
```

COMPTE!!

No oblidar-se d'afegir el tipus **HashFunct** com a tercer paràmetre del template a la declaració de la classe i a tots els mètodes de la classe si s'està implementant una taula de dispersió on la clau és genèrica.

En l'eliminació cal controlar si l'element esborrat era el primer de la llista per actualitzar correctament la taula.

```
template <typename Clau, typename Valor, typename HasFunct>
void dicc<Clau, Valor, HashFunct>::elimina (const Clau &k) throw() {
  nat i = hash(k);
  node_hash *p = _taula[i], *ant=NULL;
  bool trobat = false;
  while (p != NULL and not trobat) {
    if (p->_k == k) {
      trobat = true;
    }
    else {
      ant = p;
      p = p->_seg;
    }
  }
  if (trobat) {
    if (ant == NULL) {
      _taula[i] = p->seg; // Era el primer
    }
    else {
      ant->seg = p->seg;
    }
    delete(p);
    --_quants;
  }
}
```

Algunes consideracions generals:

- Les llistes estan simplement encadenades i no fa falta utilitzar fantasmes ni tancar-les circularment. De fet aquestes llistes en general tindran pocs elements.
- Es pot considerar la possibilitat d'ordenar els sinònims de cada llista per clau. Això permet reduir el cost de les cerques sense èxit.
- Si el *factor de càrrega* α = _quants /_M és un valor raonablement petit podem considerar que el cost mitjà de totes les operacions del diccionari (excepte la constructora) és $\Theta(1)$.
- L'inconvenient de les taules de dispersió encadenades indirectes és que l'accés al primer element de la llista de sinònims no és immediat i que ocupa força espai degut als encadenaments.

6.9.4.2. Sinònims encadenats directes

La taula està dividida en dues zones:

- La *zona principal* de M elements: La posició *i* o bé està buida o bé conté un parell <k,v> tal que hash(k)=*i*

- La *zona d'excedents* que ocupa la resta de la taula i guarda les claus sinònimes. Quan es produeix una col·lisió, el nou sinònim passa a aquesta zona. Tots els sinònims d'un mateix valor de dispersió estan encadenats (ara els encadenaments són enters).

S'anomena directe perquè la taula guarda realment els elements de manera que ens estalviem un accés respecte a les indirectes. Així l'accés al primer element de la llista de sinònims és immediat.

La codificació de les operacions és una mica més complicat que en el cas de taules encadenades indirectes doncs cal distingir si una posició és lliure o no i també cal fer la gestió de les posicions lliures de la zona d'excedents (per ex. amb una llista encadenada dels llocs lliures).

Cal anar amb molt de compte amb l'eliminació amb els sinònims encadenats directament:

- Quan eliminem un element de la zona d'excedents l'eliminem de la llista de sinònims incorporant-lo a la llista de llocs lliures.

- Quan eliminem un element de la zona principal cal evitar de perdre l'encadenament amb els sinònims.

  Es podem utilitzar dues estratègies:

  1. **Traslladar el primer element sinònim** des de la zona d'excedents a la zona principal i eliminar aquest. No convé si el tipus clau+valor ocupa molt d'espai.
  2. **Marcar l'element com a *esborrat*** (s'interpreta com *ocupat* per les consultes i eliminacions i *lliure* per les insercions). L'inconvenient és que es perd l'avantatge de l'accés directe.

MIDA DE LA ZONA D'EXCEDENTS

Per obtenir bons resultats la zona d'excedents no hauria de passar del **14%** de la dimensió total de la taula.

6.9.4.3. Direccionament obert: Sondeig lineal

En les estratègies de direccionament obert els sinònims es guarden dins de la mateixa taula de dispersió.

Per a cada clau K es defineix una seqüència de posicions i_0=h(K), i_1, i_2, ... que determinen on pot estar (consulta) o on anirà a parar (inserció) la clau K.

Hi ha diferents estratègies segons la seqüència de posicions sondejades. La més senzilla de totes és el sondeig lineal:

$$i_0 = h(\mathrm{K}), \quad i_1 = (i_0 + 1) \bmod \mathrm{M}, \quad i_2 = (i_1 + 1) \bmod \mathrm{M}, \ldots$$

```
template <typename Clau, typename Valor>
class dicc {
public:
  ...
private:
  enum Estat {lliure, esborrat, ocupat};
  struct node_hash {
    Clau _k;
    Valor _v;
    Estat _est;
  };

  node_hash *_taula; // taula amb parells <k,v>
  nat _quants;   // nº d'elements guardats al diccionari
  nat _M;        // mida de la taula

  int busca_node(const Clau &k) const throw();
};
```

En el cas de la inserció cal assegurar-se que almenys hi ha un lloc no ocupat dins de la taula abans d'inserir, _quants < _M. En cas contrari es pot generar un error o fer més gran la taula de dispersió (per més informació veure l'apartat 6.9.5 Redispersió)

```
Retorna la posició on es troba l'element amb la clau indicada o,
en cas que no es trobi la clau, la primera posició no ocupada.
```

```
template <typename Clau, typename Valor>
int dicc<Clau, Valor>::busca_node(const Clau &k) const
throw() {
  nat i = hash(k);

  // prilliure és la primera posició esborrada que
  // trobem, val -1 si no trobem cap posició esborrada.

  nat prilliure = -1;
  // comptem el nombre d'elements que visitem per només
  // fer una passada.
  nat cont = 0;
  while (_taula[i]._k != k and _taula[i]._est != lliure
         and cont < _M) {
    ++cont;
    if (_taula[i]._est == esborrat and prilliure == -1) {
      prilliure = i;
    }
    i = (i+1) % _M;
  }
  if (_taula[i]._est == lliure and _taula[i]._k != k)
    if (prilliure != -1)
      i = prilliure;

  return i;
}

template <typename Clau, typename Valor>
void dicc<Clau, Valor>::insereix(const Clau &k, const
Valor &v) throw(error) {
  nat i = busca_node(k);
  if (_taula[i]._est == ocupat and _taula[i]._k != k) {
    // redispersió
  }
  if (_taula[i]._est != ocupat) {
    ++_quants;
  }
  _taula[i]._k = k;
  _taula[i]._v = v;
  _taula[i]._est = ocupat;
}
```

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::consulta(const Clau &k, bool
&hi_es, Valor &v) const throw(error) {
  nat i = busca_node(k);
  if (_taula[i]._est == ocupat and _taula[i]._k == k) {
    v = _taula[i]._v;
    hi_es = true;
  }
  else {
    hi_es = false;
  }
}
```

L'eliminació d'elements en taules de direccionament obert és complicada. Si una clau K col·lisiona i després de fer diferents sondeigs es diposita en una posició determinada, és perquè les posicions anteriors estaven ocupades. Si eliminem algun element de les posicions anteriors no podem marcar la posició com a LLIURE, doncs una cerca posterior de la clau K fracassaria.

Hem d'afegir un tercer estat i marcar la posició eliminada com ESBORRADA.

Una posició marcada com ESBORRADA:

- En les cerques es comporta com si fos ocupada.
- En les insercions es pot aprofitar per col·locar nous elements.

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::elimina(const Clau &k) const
throw() {
  nat i = busca_node(k);
  if (_taula[i]._est == ocupat and _taula[i]._k == k) {
    _taula[i]._est = esborrat;
  }
}
```

Les eliminacions degraden notablement el rendiment de les taules de direccionament obert ja que les posicions ESBORRADES compten igual que les realment ocupades quan es busquen elements.

L'avantatge de les taules de direccionament obert és que la seva representació és molt compacte (no ocupa espai de memòria addicional amb encadenaments). L'inconvenient és que es degeneren si es realitzen insercions i eliminacions alternativament o si el *factor de càrrega* és gran (α proper a 1).
Si $\alpha < 1$ el cost de les cerques amb èxit (i les modificacions) serà proporcional a la fórmula

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

i el cost de les cerques sense èxit (i les insercions) serà proporcional a la fórmula

$$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

APINYAMENT

Un fenomen indesitjable que s'accentua quan α és proper a 1 és l'**apinyament** (anglès: *clustering*). Es produeix quan molts elements no poden ocupar la seva posició "preferida" a l'estar ocupada per un altre element que no té perquè ser un sinònim (són els "invasors"). Grups de sinònims més o menys dispersos acaben fonent-se en grans apinyaments. Llavors, quan busquem un clau, tenim que examinar no només els seus sinònims sinó altres claus que no tenen cap relació amb ella.

6.9.5. Redispersió

Si el nostre llenguatge de programació permet definir la grandària d'una taula en temps d'execució podem utilitzar la tècnica de **redispersió** (anglès: *rehash*).

Si el *factor de càrrega* α = _quants / _M és molt alt superant un cert llindar:

1. Es reclama a la memòria dinàmica una taula de grandària el doble de l'actual.
2. Es reinserta tota la informació que contenia la taula actual a la nova taula. Caldrà recórrer seqüencialment la taula actual i cadascun dels elements presents s'insereix a la nova taula utilitzant una nova funció de dispersió. Això té un cost proporcional a la grandària de la taula ($\Theta(n)$) però es fa molt de tant en tant.

La redispersió permet que el diccionari creixi sense límits prefixats, garantint un bon rendiment de totes les operacions i sense malgastar massa memòria, doncs la mateixa tècnica es pot aplicar a la inversa, per evitar que el *factor de càrrega* sigui excessivament baix.

Pot demostrar-se que, encara que una operació individual d'inserció o eliminació en un diccionari pugui tenir cost $\Theta(n)$ degut a la redispersió, una seqüència de n operacions d'inserció o eliminació tindrà cost $\Theta(n)$ i, per tant, en terme mig cada operació té cost $\Theta(1)$.

6.10. TRIES

6.10.1. Definició i exemples

Les claus que identifiquen als elements d'un diccionari estan formades per una seqüència de **símbols** (per ex. caràcters, dígits, bits). La descomposició de les claus en símbols es pot aprofitar per implementar les operacions típiques d'un diccionari de manera notablement eficient.

A més a més, sovint necessitem operacions en el diccionari basades en la descomposició de les claus en símbols. Per exemple, donada una col·lecció de paraules *C* i un prefix *p* retornar totes les paraules de *C* que comencen amb el prefix *p*.

Considerem que els símbols pertanyen a un **alfabet** finit de $m \geq 2$ elements $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$.

Exemple:

Si les claus les descomposem en bits, utilitzarem un alfabet que té m=2 elements $\Sigma = \{0, 1\}$.

- Σ^* denota el conjunt de les seqüències (cadenes) formades per símbols de Σ.
- Donades 2 seqüències *u* i *v*, $u \cdot v$ denota la seqüència resultant de concatenar *u* i *v*.

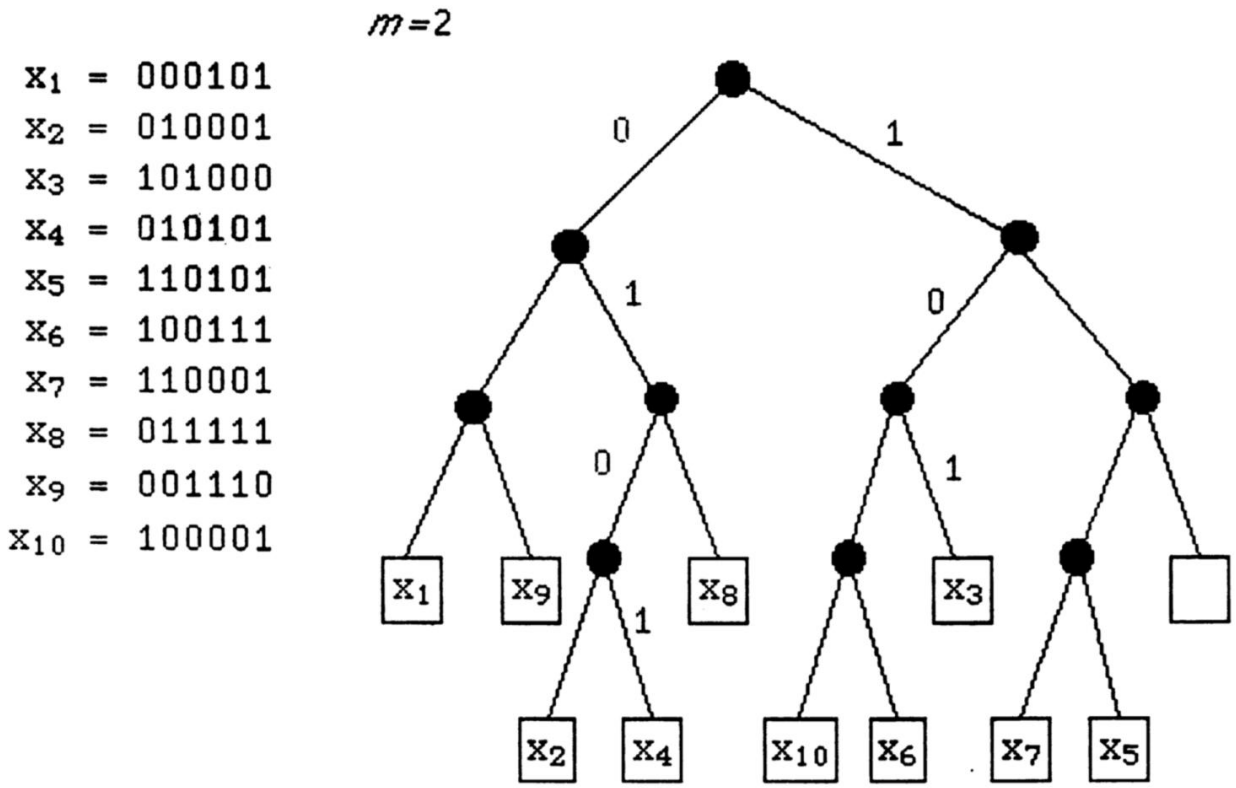
A partir d'un conjunt finit de seqüències $X \subset \Sigma^*$ de idèntica longitud podem construir un **trie** *T* tal i com indica aquesta definició:

Definició: El **trie** *T* és un arbre m-ari definit recursivament com:

1. Si *X* és buit llavors *T* és un arbre buit.
2. Si *X* conté un sol element llavors *T* és un arbre amb un únic node que conté a l'únic element de *X*.
3. Si $|X| \geq 2$, sigui T_i el trie corresponent a fórmula $X_i = \{y \ / \ \sigma_i \cdot y \in X \ \wedge \ \sigma_i \in \Sigma\}$. Llavors *T* és un arbre mari constituït per una arrel que té com a fills els *m* subarbres T_1, T_2, ... , T_m.

Exemple:

Trie construït a partir de 10 seqüències, totes elles formades per 6 símbols binaris. Com que *m*=2 el trie resultant és un arbre binari.



Lema 1: Si les arestes del trie T corresponent a un conjunt X s'etiqueten mitjançant els símbols de Σ de forma que l'aresta que uneix l'arrel amb el primer subarbre s'etiqueta amb σ_1, la que uneix l'arrel amb el segon subarbre s'etiqueta σ_2, ... llavors les etiquetes del camí que ens porta des de l'arrel fins a una fulla no buida que conté a x constitueix el prefix més curt que distingeix unívocament a x (cap altre element de X comença amb el mateix prefix).

Lema 2: Sigui p l'etiqueta corresponent a un camí que va des de l'arrel d'un trie T fins a un cert node (intern o fulla) de T. Llavors el subarbre que penja d'aquest node conté tots els elements de X que tenen com a prefix p (i no més elements).

Lema 3: Donat un conjunt $X \subset \Sigma^*$ de seqüències d'igual longitud, el seu trie corresponent és únic. En particular T no depèn de l'ordre en que estiguin els elements de X.

Lema 4: L'alçada d'un trie T és igual a la longitud mínima del prefix necessari per distingir qualsevol dos elements del conjunt que representa el trie. En particular, si l és la longitud de les seqüències en X, l'alçada de T serà $\leq l$.

La definició d'un trie imposa que totes les seqüències siguin d'igual longitud, el qual és molt restrictiu.

Si no exigim aquesta condició, com podem distingir dos elements x i y si x és prefix de y?

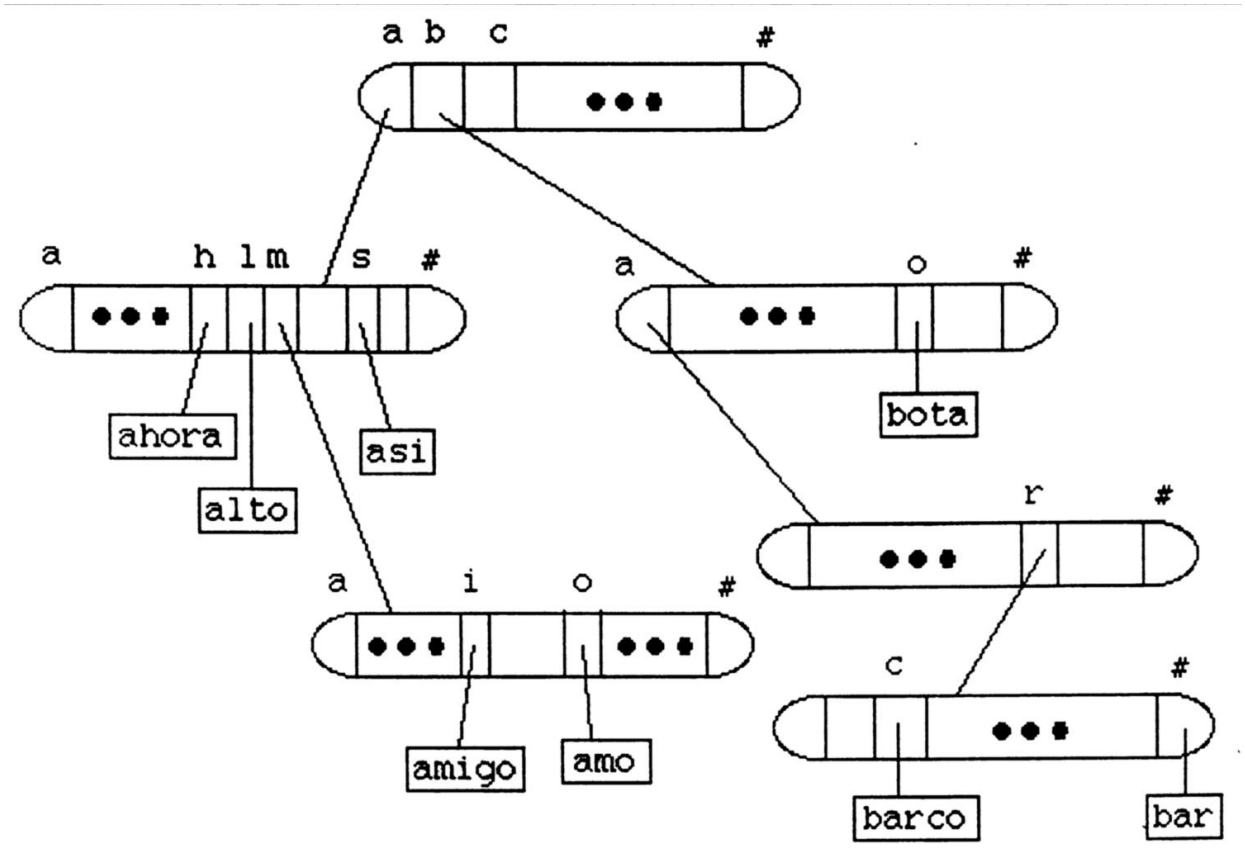
Una solució habitual consisteix en ampliar l'alfabet Σ amb un **símbol especial de fi de seqüència** (per ex. #) i marcar cadascuna de les seqüències en X amb aquest símbol. Això garanteix que cap de les seqüències marcades és prefix de les altres. L'inconvenient és que s'ha de treballar amb un alfabet de m+1 símbols i, per tant, amb arbres (m+1)-aris.

Exemple:

Trie construït a partir de les seqüències

X= {ahora, alto, amigo, amo, asi, bar, barco, bota},

totes elles formades per símbols caràcters. Com que m=25 el trie resultant és un arbre 26-ari.

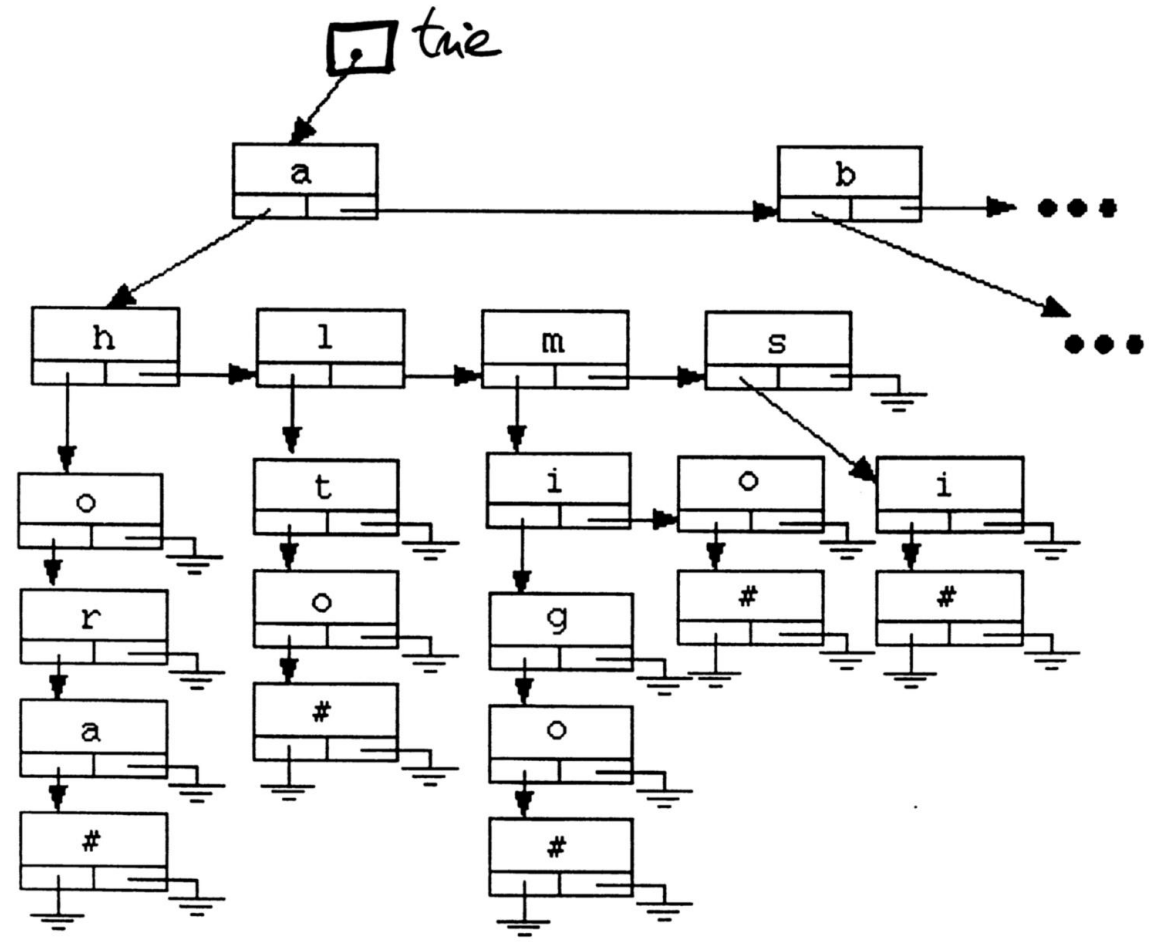


6.10.2. Tècniques d'implementació

Les tècniques d'implementació dels tries són les convencionals pels arbres:

- Si s'utilitza un **vector de punters per node**, els símbols de Σ solen utilitzar-se com índexs (utilitzant una funció *ind*: $\Sigma \rightarrow \{1,..., m\}$).
  Les fulles que contenen els elements de X poden emmagatzemar exclusivament els sufixes restants, ja que el prefix està ja codificat en el camí de l'arrel a la fulla (veure figura anterior).

- Si s'utilitza la representació **primer fill-següent germà**, cada node guarda un símbol i dos punters, un al primer fill i l'altre al següent germà.
  Com que acostuma a haver-hi un ordre sobre l'alfabet Σ, la llista de fills de cada node acostuma a ordenar-se seguint aquest ordre. La figura següent mostra un exemple en que, per simplicitat, s'utilitza el mateix tipus de node per guardar els sufixes restants de cada clau. Així evitem usar nodes i apuntadors a nodes de diferent tipus.


trie
a
b
h
l
m
s
o
t
i
o
i
r
o
g
#
#
a
#
o
#
#

6.10.3. Implementació primer fill – següent germà

IMPORTANT. La classe amb que es vulgui instanciar Clau ha de suportar les següents operacions:

```
Retorna la longitud >= 0 de la clau.
int size() const throw();
```

```
Retorna l'i-èssim símbol de la clau. El primer símbol és i==0.
Simbol operator[](int i) throw(error);
```

Caldrà especialitzar la funció especial segons el tipus dels símbols de les claus del diccionari. Per exemple pel tipus string seria:

```
Retorna el símbol especial fi de clau.
template <>
char especial<string>() {
     return '#';
}

template <class Simbol, class Clau, class Valor>
class diccDigital {
private:
  struct node_trie {
    Simbol _c;
    node_trie* _primfill; // primer fill
    node_trie* _seggerma; // següent germà
    Valor _v;
  };
  node_trie *_arrel;

  static node_trie* consulta_node (node_trie *p,
    const Clau &k, nat i) throw();

public:
  void consulta (const Clau &k, bool &hi_es, Valor &v)
     const throw(error);
  ...
};
```

```
// Cost: Θ(k.length())
template <class S, class C, class V>
void diccDigital<S, C, V>::consulta (const Clau &k, bool
&hi_es, Valor &v) const throw() {
  node_trie *n = consulta_node(_arrel, k, 0);
  if (n == NULL) {
    hi_es = false;
  }
  else {
    v = n->_v;
    hi_es = true;
  }
}
```

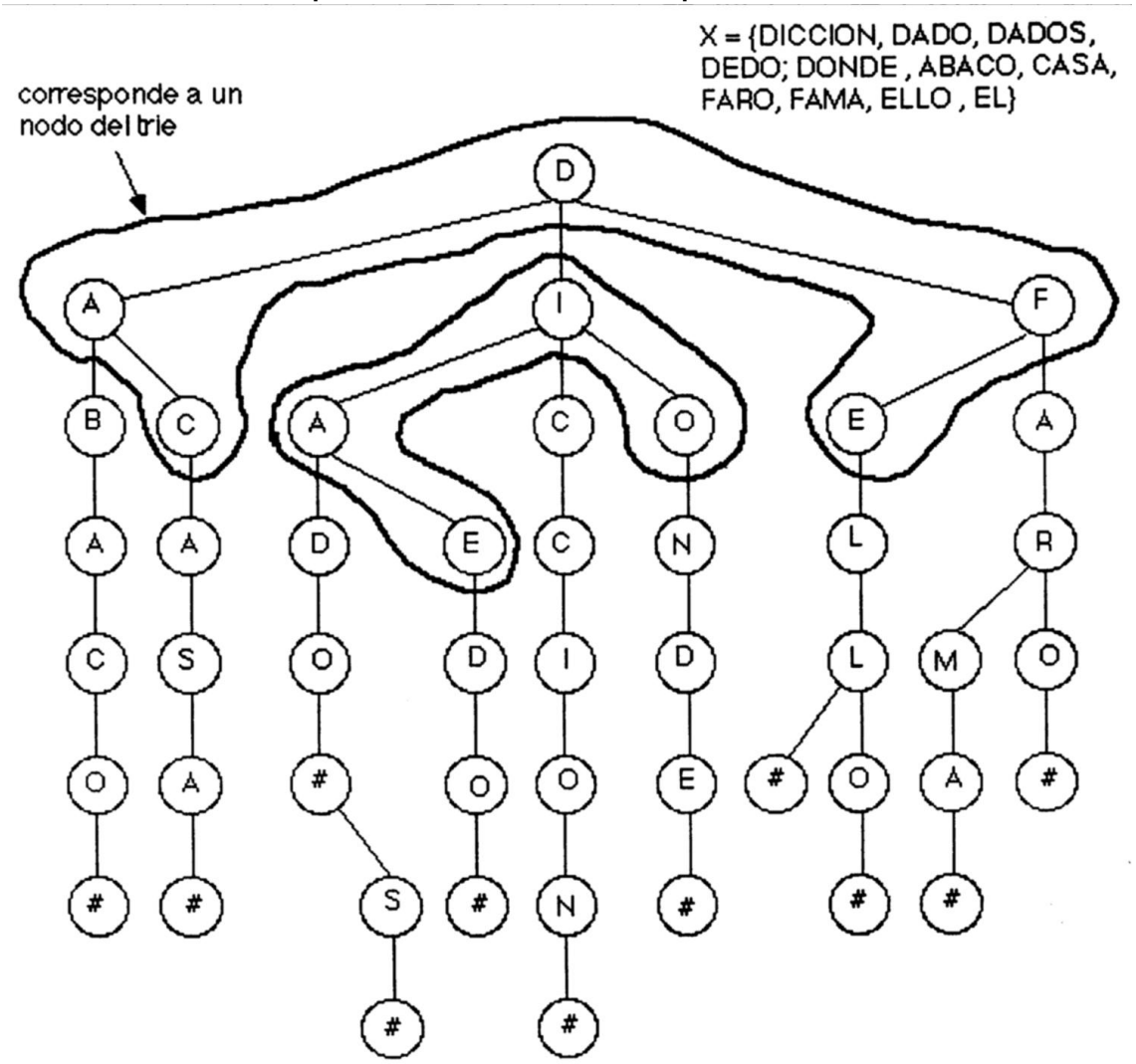
Mètode privat de classe.

```
// Cost: Θ(k.length())
template <class S, class C, class V>
typename diccDigital<S, C, V>::node_trie*
diccDigital<S, C, V>::consulta_node (node_trie *n, const
Clau &k, nat i) throw() {
  node_trie *res = NULL;
  if (n != NULL) {
    if (i == k.length() and n->_c == especial<Clau>()) {
      res = n;
    }
    else if (n->_c > k[i]) {
      res = NULL;
    }
    else if (n->_c < k[i]) {
      res = consulta_node(n->_seggerma, k, i);
    }
    else if (n->_c == k[i]) {
      res = consulta_node(n->_primfill, k, i+1);
    }
  }
  return res;
}
```

6.10.4. Arbre ternari de cerca

Una alternativa que combina eficiència en l'accés als subarbres i estalvi de memòria consisteix en implementar cada node del trie com un BST. L'estructura resultant s'anomena **arbre ternari de cerca** (anglès: *ternary search tree*) ja que cada node conté tres apuntadors:

- Dos punters al fill esquerra i fill dret del BST que conté els diferents símbols *i*-èssims de totes les claus que tenen el mateix prefix format per *i*-1 elements.
- Un punter (l'anomenem central) a l'arrel del subarbre que conté, formant un BST, els símbols de la següent posició de totes les claus que tenen el mateix prefix.



```
template <class Simbol, class Clau, class Valor>
class diccDigital {
private:
  struct node_tst {  // tst = ternary search tree
    Simbol _c;
    node_tst* _esq;
    node_tst* _dret;
    node_tst* _cen;
    Valor _v;
  };
  node_tst *_arrel;

  // operacions privades
  static node_tst* rconsulta (node_tst *n, nat i,
         const Clau &k) throw();
  static node_tst* rinsereix (node_tst *n, nat i,
         const Clau &k, const Valor &v) throw(error);

public:
  ...
  void consulta (const Clau &k, bool &hi_es, Valor &v)
    const throw();
  void insereix (const Clau &k, const Valor &v)
    throw(error);
  ...
};


// Cost: Θ(k.length() * log(#símbols))
template <class Simbol, class Clau, class Valor>
void diccDigital::consulta (const Clau &k, bool &hi_es,
Valor &v) const throw() {
  node_tst *n = rconsulta(_arrel, 0, k);
  if (n == NULL) {
    hi_es = false;
  }
  else {
    v = n->_v;
    hi_es = true;
  }
}
```

Mètode privat de classe.

```
// Cost: Θ(k.length() * log(#símbols))
template <class Simbol, class Clau, class Valor>
typename diccDigital::node_tst*
diccDigital::rconsulta (node_tst *n, nat i, const Clau
&k) throw() {
  node_tst *res = NULL;
  if (n != NULL) {
    if (i == k.length() and n->_c == especial<Clau>()) {
      res = n;
    }
    else if (n->_c > k[i]) {
      res = rconsulta(n->_esq, i, k);
    }
    else if (n->_c < k[i]) {
      res = rconsulta(n->_dret, i, k);
    }
    else if (n->_c == k[i]) {
      res = rconsulta(n->_cen, i+1, k);
    }
  }
  return res;
}
```

Insereix un parell <Clau, Valor> en el diccionari. Actualitza el valor si la Clau ja era present al diccionari.

Cal tenir present que cal afegir un sentinella al final de la clau. La funció especial() retorna el símbol usat per indicar el final de la clau, per ex. si la Clau és string llavors especial() retorna '#'.

```
// Cost: Θ(k.longitud() * log(#símbols))
template <class Simbol, class Clau, class Valor>
void diccDigital::insereix (const Clau &k, const Valor
&v) throw(error) {
  // Afegir el sentinella al final de la clau
  Clau k2 = k + especial<Clau>();
  _arrel = rinsereix(_arrel, 0, k2, v);
}
```

Operació privada de classe
Recorrerem tots els símbols de la clau. Cal tenir en compte que s'ha afegit el símbol nul al final de la clau.

```
// Cost: Θ(k.length() * log(#símbols))
template <class Simbol, class Clau, class Valor>
typename diccDigital::node*
diccDigital::rinsereix (node_tst *n, nat i, const Clau
&k, const Valor &v) throw(error) {
  if (n == NULL) {
    n = new node_tst;
    n->_esq = n->_dret = n->_cen = NULL;
    n->_c = k[i];
    try {
      if (i < k.length()-1) {
        n->_cen = rinsereix(n->_cen, i+1, k, v);
      }
      else { // i == k.length()-1; k[i] == Simbol()
        n->_v = v;
      }
    }
    catch (error) {
      delete n;
      throw;
    }
  }
  else {
    if (n->_c > k[i]) {
      n->_esq = rinsereix(n->_esq, i, k, v);
    }
    else if (n->_c < k[i]) {
      n->_dret = rinsereix(n->_dret, i, k, v);
    }
    else {   // (n->_c == k[i])
      n->_cen = rinsereix(n->_cen, i+1, k, v);
    }
  }
  return n;
}
```

6.11. RADIX SORT

6.11.1. Introducció

La descomposició digital (en símbols o dígits) de les claus també pot utilitzar-se per l'ordenació. Els algorismes basats en aquest principi s'anomenen d'**ordenació digital** (anglès: *radix sort*).

Estudiarem el cas general en que la clau la descomposem en una seqüència de bits.

Considerem que hem d'ordenar un vector de *n* elements cadascun dels quals és una seqüència de *l* bits.

6.11.2. Funcionament

Si ordenem el vector segons el bit de major pes, després cada bloc resultant l'ordenem segons el bit de següent pes, i així successivament, haurem ordenat tot el vector.

Acció per ordenar el tros de vector A[u..v] tenint en compte el bit *r*-èssim.

```
template <typename T>
void radixsort(T A[], nat u, nat v, nat r) {
  if (u<v and r>=0) {
    nat k = particio_radix(A, u, v, r);
    radixsort(A, u, k, r-1);
    radixsort(A, k+1, v, r-1);
  }
}
```

La crida inicial és:

```
radixsort(A, 0, n-1, l).
```

La següent funció mostra una manera de realitzar la partició del tros de vector A[u..v] tenint en compte el bit *r*-èssim. La manera de procedir és molt similar a la partició del quick-sort.

Donat un element *x*, bit(*x*, *r*) retorna el bit *r*-èssim de *x*.

```
template <typename T>
nat particio_radix(T A[], nat u, nat v, nat r) {
  nat i = u, j = v;
  while (i < j+1) {
    while (i<j+1 and bit(A[i],r)==0) {
      ++i;
    }
    while (i<j+1 and bit(A[j],r)==1) {
      --j;
    }
    // intercanvi entre elements apuntats per i <--> j
    if (i < j+1) {
      T aux = A[i];
      A[i] = A[j];
      A[j] = aux;
    }
  }
  return j;
}
```

6.11.3. Cost

Cadascuna de les etapes de radixsort té un cost lineal. Com que el número d'etapes és *l* el cost de l'algorisme és $\Theta(n \cdot l)$.

Una altra forma de deduir el cost és considerar el cost associat a cada element del vector: Un element qualsevol és examinat (i a vegades intercanviat amb un altre) com a molt *l* vegades, per tant el cost total és $\Theta(n \cdot l)$.

TEMA 7. CUES DE PRIORITAT

7.1. CONCEPTES

Una **cua de prioritat** (anglès: *priority queue*) és una col·lecció d'elements on cada element té associat un valor susceptible d'ordenació denominat prioritat. Una cua de prioritat es caracteritza per admetre:

- insercions de nous elements.
- consulta de l'element de prioritat mínima.
- esborrar l'element de prioritat mínima.

De forma anàloga es poden definir cues de prioritat que admeten la consulta i l'eliminació de l'element de màxima prioritat a la col·lecció.

7.2. ESPECIFICACIÓ

A l'especificació següent assumirem que el tipus *Prio* ofereix una relació d'ordre total $<$.

D'altra banda, es pot donar el cas que existeixin diversos elements amb la mateixa prioritat i en aquest cas és irrellevant quin dels elements retorna l'operació *min* o elimina *elim_min*.

A vegades s'utilitza una operació *prio_min* que retorna la prioritat mínima.

```
template <typename Elem, typename Prio>
class CuaPrio {
public:
  Constructora, crea una cua buida.
  CuaPrio() throw(error);

  Tres grans.
  CuaPrio(const CuaPrio &p) throw(error);
  CuaPrio& operator=(const CuaPrio &p) throw(error);
  ~CuaPrio() throw();

  Afegeix l'element x amb prioritat p a la cua de prioritat.
  void insereix(const Elem &x, const Prio &p)
    throw(error);

  Retorna un element de mínima prioritat en la cua de prioritat.
  Llança un error si la cua és buida.
  Elem min() const throw(error);

  Retorna la mínima prioritat present en la cua de prioritat.
  Llança un error si la cua és buida.
  Prio prio_min() const throw(error);

  Elimina un element de mínima prioritat de la cua de prioritat.
  Llança un error si la cua és buida.
  void elim_min() throw(error);

  Retorna cert si i només si la cua és buida.
  bool es_buida() const throw();

private:
  ...
};
```

7.3. USOS DE LES CUES DE PRIORITAT

Les cues de prioritat tenen múltiples usos:

A) **Algorismes voraços**.

Amb freqüència s'utilitzen per a implementar algorismes voraços. Aquest tipus d'algorismes normalment tenen una iteració principal, i una de les tasques a realitzar a cadascuna d'aquestes iteracions és seleccionar un element que minimitzi o (maximitzi) un cert criteri. El conjunt d'elements entre els quals ha d'efectuar la selecció és freqüentment dinàmic i admet insercions eficients. Alguns d'aquests algorismes són:

- Algorismes de Kruskal i Prim pel càlcul de l'arbre d'expansió mínim d'un graf etiquetat.
- Algorisme de Dijkstra pel càlcul de camins mínims en un graf etiquetat.
- Construcció de codis de Huffman (codis binaris de longitud mitja mínima).

B) **Ordenació**.

Una altra tasca en la que poden utilitzar cues de prioritat és l'ordenació. Utilitzem una cua de prioritat per ordenar la informació de menor a major segons la prioritat.

Exemple:

En aquest exemple tenim dues taules: info i clau. Es vol que els elements d'aquestes dues taules estiguin ordenats per la clau.

```
template <typename Elem, typename Prio>
void ordenar(Elem info[], Prio clau[], nat n)
throw(error) {
  CuaPrio<Elem, Prio> c;
  for (i=0; i<n; ++i) {
    c.insereix(info[i], clau[i]);
  }
  for (nat i=0; i<n; ++i) {
    info[i] = c.min();
    clau[i] = c.prio_min();
    c.elim_min();
  }
}
```

C) **Element k-èssim**

També s'utilitzen cues de prioritat per a trobar l'**element k-èssim** d'un vector no ordenat.

Es col·loquen els k primers elements del vector en una *max-cua* i a continuació es fa un recorregut de la resta del vector, actualitzant la cua de prioritat cada vegada que l'element és menor que el màxim dels elements de la cua, eliminant al màxim i inserint l'element en curs.

```
template <typename T>
T k_essim(T v[], nat k) {
  CuaPrio<T, T> c;
  nat i;
  for (i=0; i<k; ++i) {
    c.inserir(v[i], v[i]);
  }
  for (i=k; i<n; ++i) {
    if (v[i] < c.max()) {
      c.elim_max();
      c.insereix(v[i], v[i]);
    }
  }
  return c.max();
}
```

7.4. IMPLEMENTACIÓ

La majoria de les tècniques utilitzades en la implementació de diccionaris poden ser utilitzades per la implementació de cues de prioritat, a excepció de les taules de dispersió i els tries.

Es pot utilitzar:

- **Llista ordenada per prioritat**. Tant la consulta com l'eliminació del mínim són trivials i els seu cost és $\Theta(1)$. Però les *insercions* tenen **cost lineal**, tant en cas pitjor com en el promig.

- **Arbre de cerca** (equilibrat o no). S'utilitza com a criteri d'ordre dels seus elements les corresponents prioritats. S'ha de modificar lleugerament l'invariant de la representació per a admetre i tractar adequadament les prioritats repetides. En aquest cas es pot garantir que totes les operacions (insercions, consultes, eliminacions) tenen un cost Θ**(log n)** en el pitjor cas si el BST és equilibrat (AVL), i en el cas mig si no ho és.

- **Skip lists**. Tenen un rendiment idèntic al que ofereixen els BST's (malgrat els costos són **logarítmics** només en el cas mig, aquest cas mig no depèn ni de l'ordre d'inserció ni de l'existència de poques prioritats repetides).

- **Taula de llistes o cues**. Si el conjunt de possibles prioritats és reduït llavors serà convenient utilitzar aquesta estructura. Cada llista o cua correspon a una prioritat o interval reduït de prioritats.

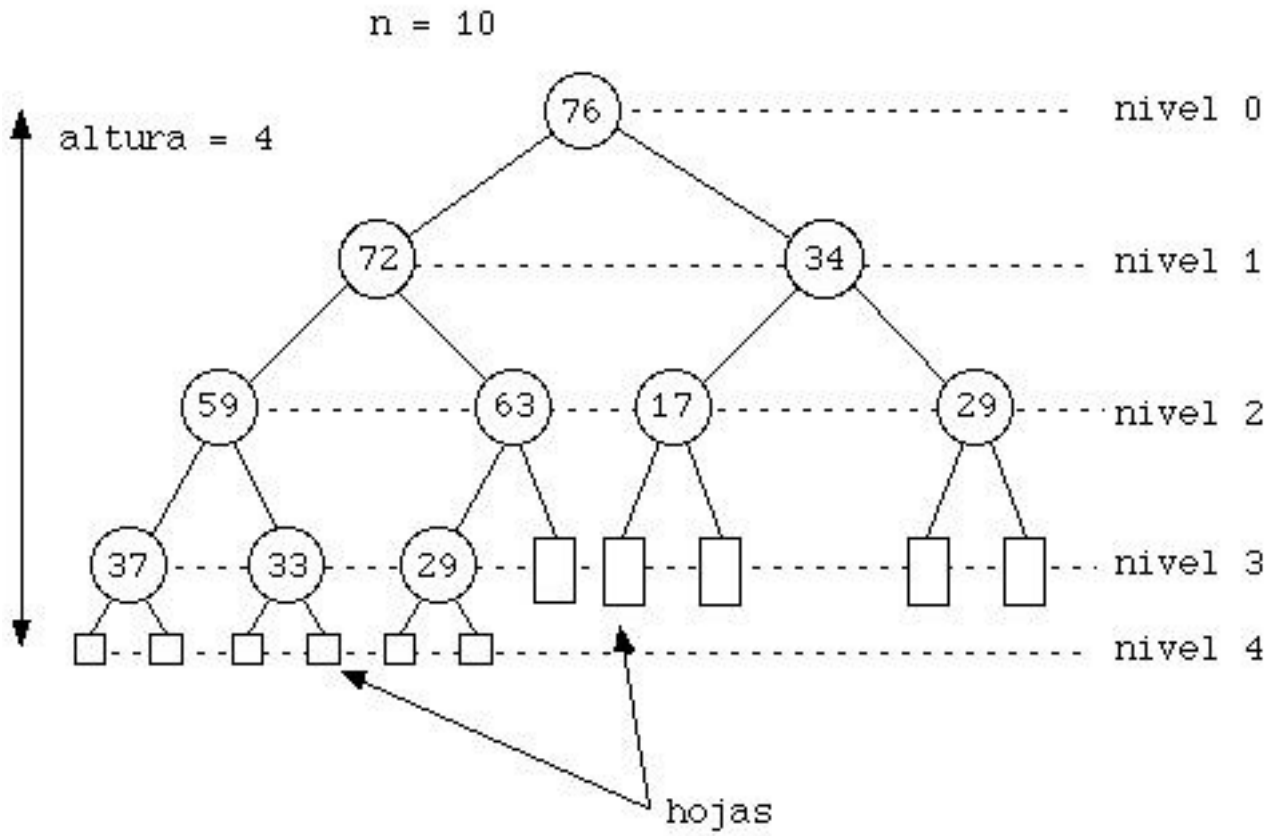
- **Monticles**. En la resta d'aquest capítol estudiarem una tècnica específica per a la implementació de cues de prioritat basada en els denominats *monticles*.

7.5. MONTICLES

Un **monticle** (anglès: *heap*) és un arbre binari tal que:

1. Totes les fulles (sub-arbres buits) es situen en els dos últims nivells de l'arbre.
2. Al nivell antepenúltim el nombre de nodes amb un únic fill és com a màxim un. Aquest únic fill serà el seu fill esquerre, i tots els nodes a la seva dreta son nodes interns sense fills.
3. L'element (la seva prioritat) emmagatzemat en un node qualsevol és més gran (o menor) o igual que els elements emmagatzemats als seus fills esquerre i dret.

Direm que un monticle és un arbre binari quasi-complet degut a les propietats 1-2. La propietat 3 es denomina **ordre del monticle**, i parlarem de *max-heaps* si els elements són >= que els seus fills o *min-heaps* si són <=. A partir d'ara només considerarem *max-heaps*.



De les propietats 1-3 es desprenen dues conseqüències importants:

1.L'element màxim es troba a l'arrel.

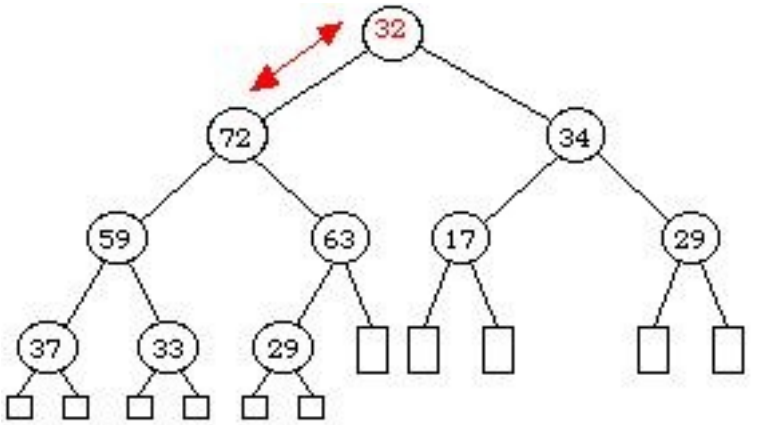
2.Un heap de n elements té alçada h = $\lceil \log_2(n+1) \rceil$

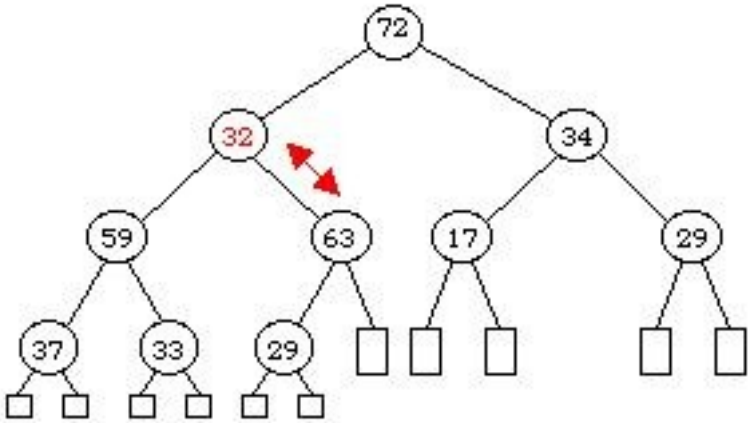
La consulta del màxim és senzilla i eficient doncs només cal examinar l'arrel.

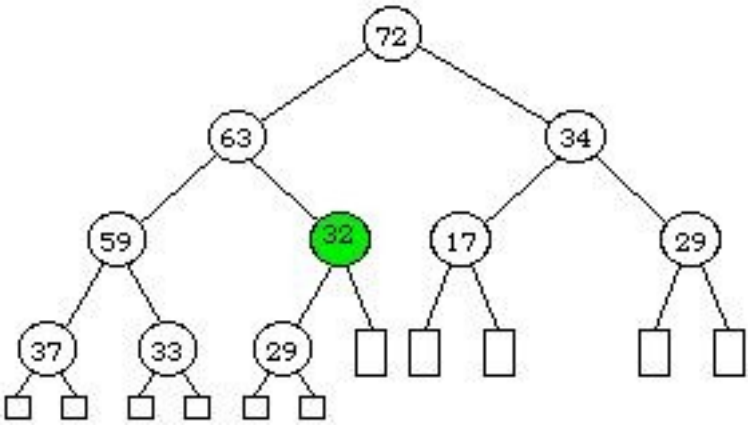
7.5.1. Eliminació del màxim

Un procediment que s'utilitza sovint consisteix en ubicar l'últim element del monticle (situat a l'últim nivell més a la dreta) a l'arrel, substituint al màxim; això garanteix que es compleixin les propietats 1 i 2. Però com que la propietat 3 deixa de complir-se, s'ha de restablir l'invariant amb un procediment privat denominat enfonsar.

Aquest procediment consisteix en intercanviar un node amb el més gran dels seus dos fills si el node és menor que algun d'ells, i repetir aquest pas fins que l'invariant s'hagi restablert.


32
72
34
59
63
17
29
37
33
29


72
32
34
59
63
17
29
37
33
29


72
63
34
59
32
17
29
37
33
29

7.5.2. Afegir un nou element

Una possibilitat consisteix en col·locar el nou element com a *últim element del monticle*, justament a la dreta de l'últim o com a primer d'un nou nivell. Per això s'ha de localitzar al pare de la primera fulla i substituir-la per un nou node amb l'element a inserir.

A continuació s'ha de restablir l'ordre del monticle utilitzant per això un procediment denominat ***surar***, que treballa de manera similar però a la inversa d'***enfonsar***: el node en curs es compara amb el seu pare i es realitza l'intercanvi si aquest node és mes gran que el pare, iterant aquest pas mentre sigui necessari.

7.5.3. Implementació amb vector

Donat que l'alçada del *heap* és log n, el cost de les insercions i eliminacions és logarítmic. Es pot implementar un *heap* mitjançant memòria dinàmica. La representació escollida ha d'incloure apuntadors al fill esquerre i al dret i també al **pare**, i resoldre de manera eficaç la localització de l'últim element i del pare de la primera fulla.

Una alternativa atractiva és la implementació de *heaps* mitjançant un vector. No es malgasta massa espai donat que el heap és quasi-complet. Les regles per a representar els elements del *heap* en un vector són simples:

1. A[1] conté l'arrel.
2. Si $2i \leq n$ llavors A[2*i*] conté el fill esquerre de l'element en A[*i*] i si $2i+1 \leq n$ llavors A[2*i*+1] conté al fill dret de A[*i*].
3.. Si *i* **div** 2 ≥ 1 llavors A[*i* **div** 2] conté al pare de A[*i*].

Cal destacar el fet que les regles anteriors impliquen que els elements del monticle s'ubiquen en posicions consecutives del vector, col·locant l'arrel a la primera posició i anant recorrent l'arbre per nivells d'esquerra a dreta.

```
template <typename Elem, typename Prio>
class CuaPrio {
  ...
  static const int CuaPrioPlena = 310;
  static const int CuaPrioBuida = 320;

private:
  static const nat MAX_ELEM = 100;

  Nombre d'elements en el heap.
  nat _nelems;

  Taula de MAX_ELEMS parells <Elem, Prio>.
  La component 0 no s'usa.
  pair<Elem, Prio> _h[MAX_ELEM+1];

  void enfonsar (nat p) throw();
  void surar (nat p) throw();
};
```

Només implementarem aquells mètodes més destacats de la classe. La implementació de les tres grans i la resta de mètodes és trivial.

```
template <typename Elem, typename Prio>
CuaPrio<Elem, Prio>::CuaPrio() throw(error) : _nelems(0)
{ }

template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::insereix(const Elem &x, const
Prio &p) throw(error) {
  if (_nelems == MAX_ELEM) throw error(CuaPrioPlena);
  ++_nelems;
  _h[_nelems] = make_pair(x, p);
  surar(_nelems);
}
```

```
template <typename Elem, typename Prio>
Elem CuaPrio<Elem, Prio>::min() const throw(error) {
  if (_nelems == 0) throw error(CuaPrioBuida);
  return _h[1].first;
}

template <typename Elem, typename Prio>
Prio CuaPrio<Elem, Prio>::prio_min() const throw(error) {
  if (_nelems == 0) throw error(CuaPrioBuida);
  return _h[1].second;
}

template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::elim_min() throw(error) {
  if (_nelems == 0) throw error(CuaPrioBuida);
  swap(_h[1], _h[_nelems]);
  --_nelems;
  enfonsar(1);
}
```

Operació privada (versió recursiva)
Enfonsa al node j-èssim fins a restablir l'ordre del monticle a _h; els subarbres del node j són heaps.
Cost : $\Theta(\log_2(n/j))$; .

```
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::enfonsar(nat j) throw() {
  nat hj;
  // si j no té fill esquerre ja hem acabat
  if (2*j <= _nelems) {
    hj = 2*j;
    if (hj < _nelems and
        _h[hj].second > _h[hj+1].second) {
      ++hj;
    }
    // hj apunta al fill de mínima prioritat de j.
    // Si la prioritat de j és major que la prioritat del
    // seu fill menor cal intercanviar i seguir enfonsant.
    if (_h[j].second > _h[hj].second) {
      swap(_h[j], _h[hj]);
      enfonsar(hj);
    }
  }
}
```

```
Operació privada (versió iterativa).
Enfonsa al node p-èssim fins a restablir l'ordre del monticle a
_h; els subarbres del node p són heaps.
Cost : Θ(log2(n/p))
```

```
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::enfonsar(nat p) throw() {
  bool fi = false;
  while ((2*p <= _nelems) and not fi) {
     nat hj = 2*p;
    if (hj < _nelems and
          _h[hj].second > _h[hj+1].second) {
      ++hj;
    }
    if (_h[p].second > _h[hj].second) {
      swap(_h[p], _h[hj]);
      p = hj;
    }
    else {
      fi = true;
    }
  }
}
```

```
Operació privada (versió iterativa)
Fa surar el node p-èssim fins a restablir l'ordre del monticle;
tots els nodes excepte el p-èssim satisfan la propietat 3.
Cost : Θ(log2(p))
```

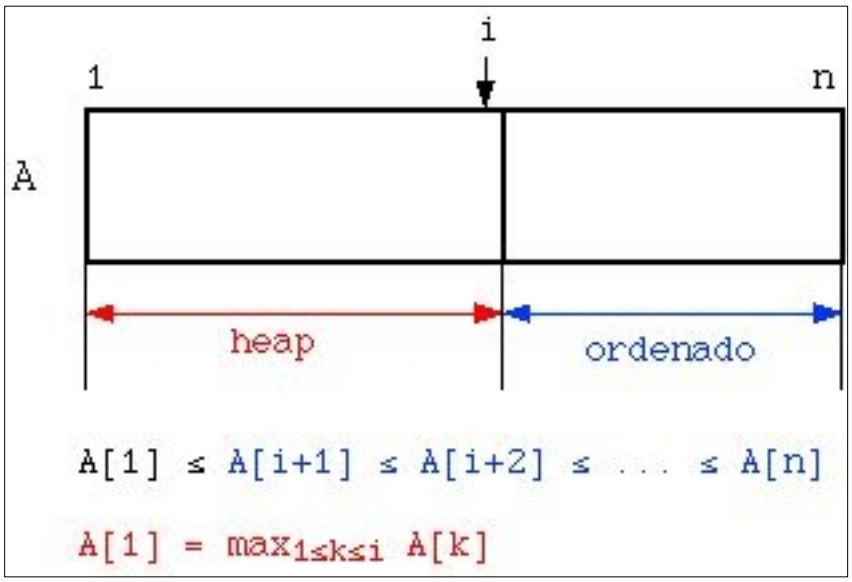
```
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::surar(nat p) throw() {
  nat q;
  bool fi = false;
  while (p > 1 and not fi) {
    q = p / 2;
    if (_h[q].second > _h[p].second) {
      swap(_h[p], _h[q]);
      p = q;
    }
    else {
      fi = true;
    }
  }
}
```

7.6. HEAPSORT

7.6.1. Introducció

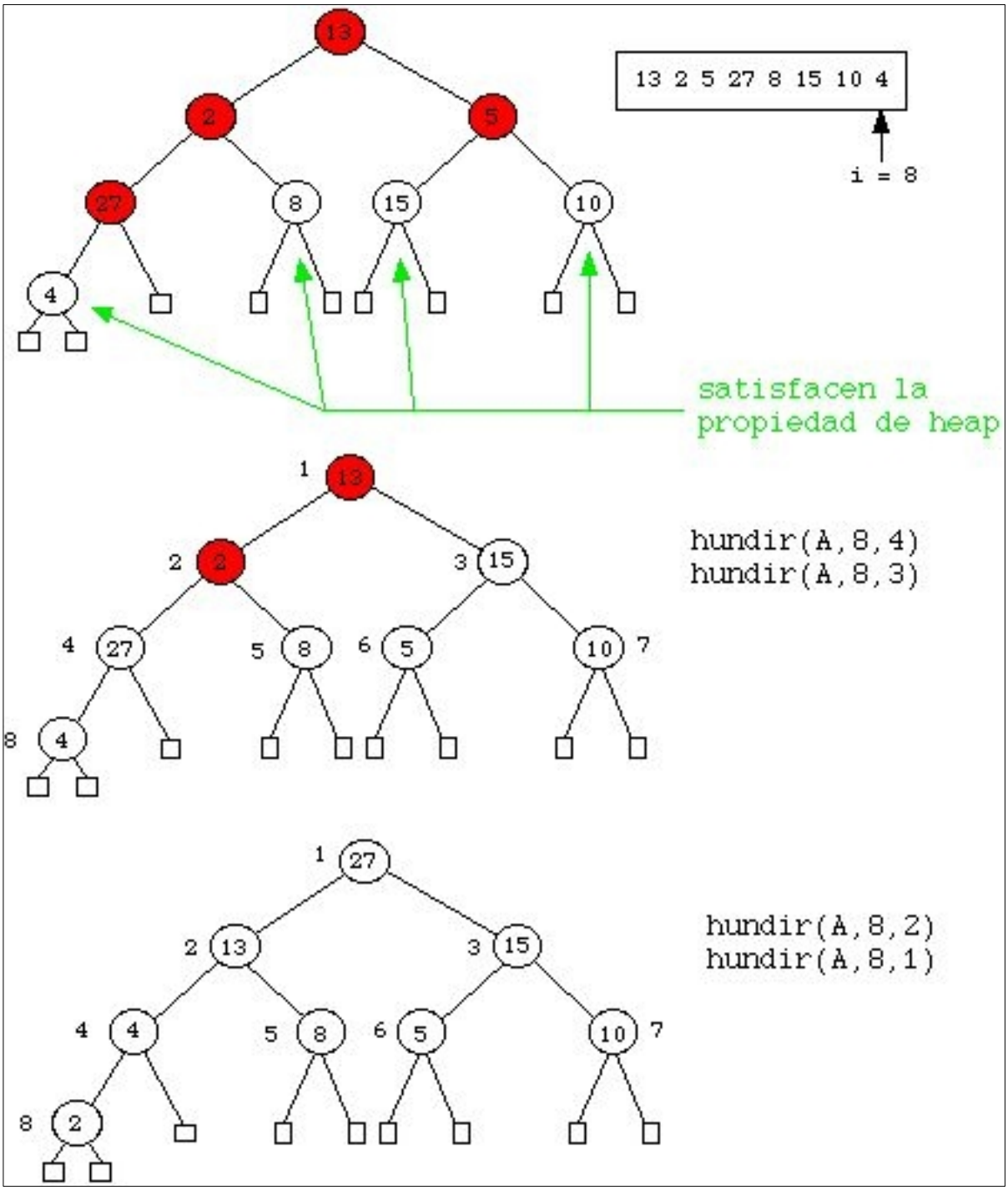
Heapsort (Williams, 1964) ordena una vector de *n* elements construint un *heap* amb els *n* elements i extraient-los un a un del *heap* a continuació.

El propi vector que emmagatzema als *n* elements s'utilitza per a construir el *heap*, de manera que *heapsort* actua in-situ i només requereix un espai auxiliar de memòria constant.

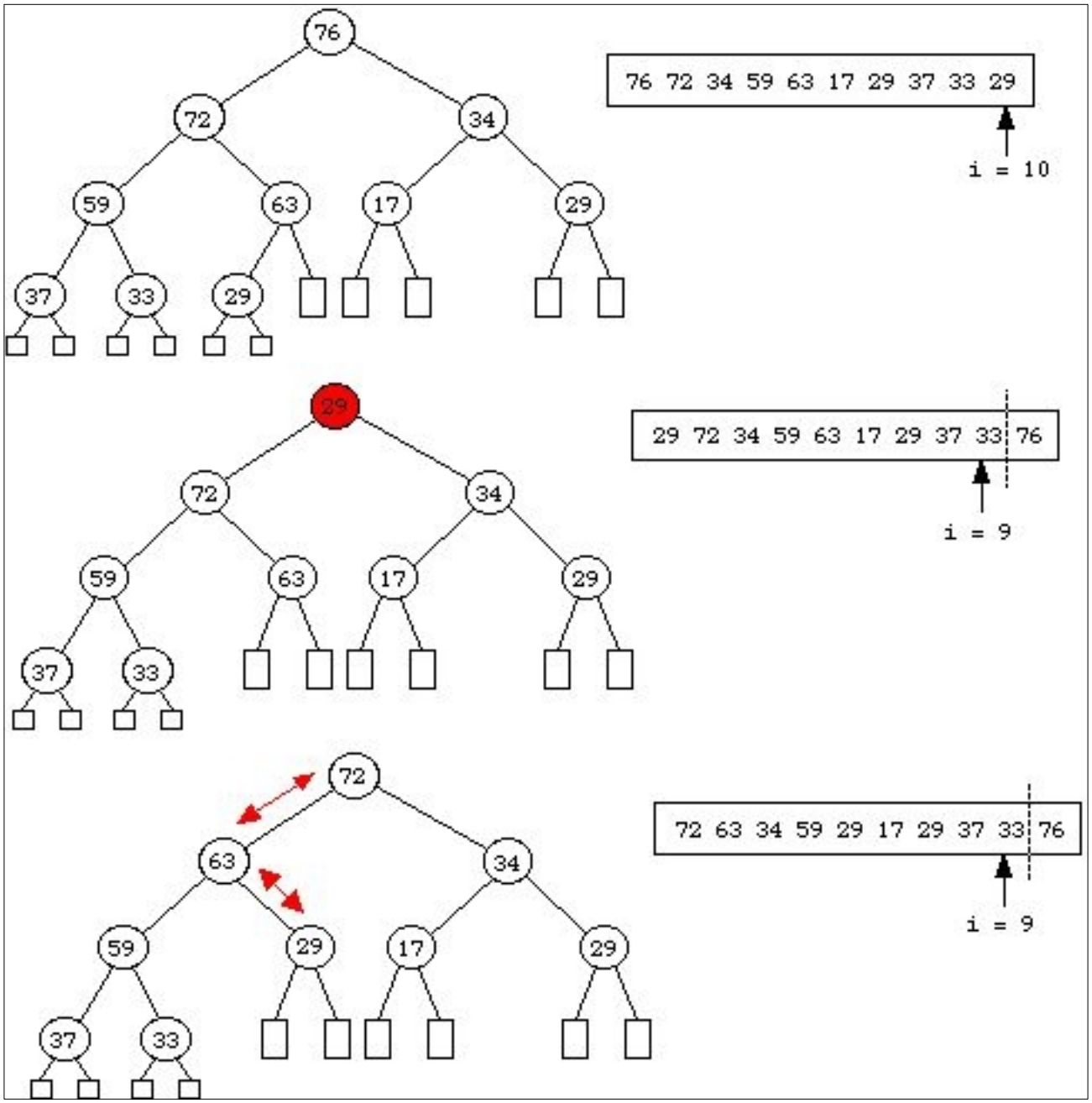


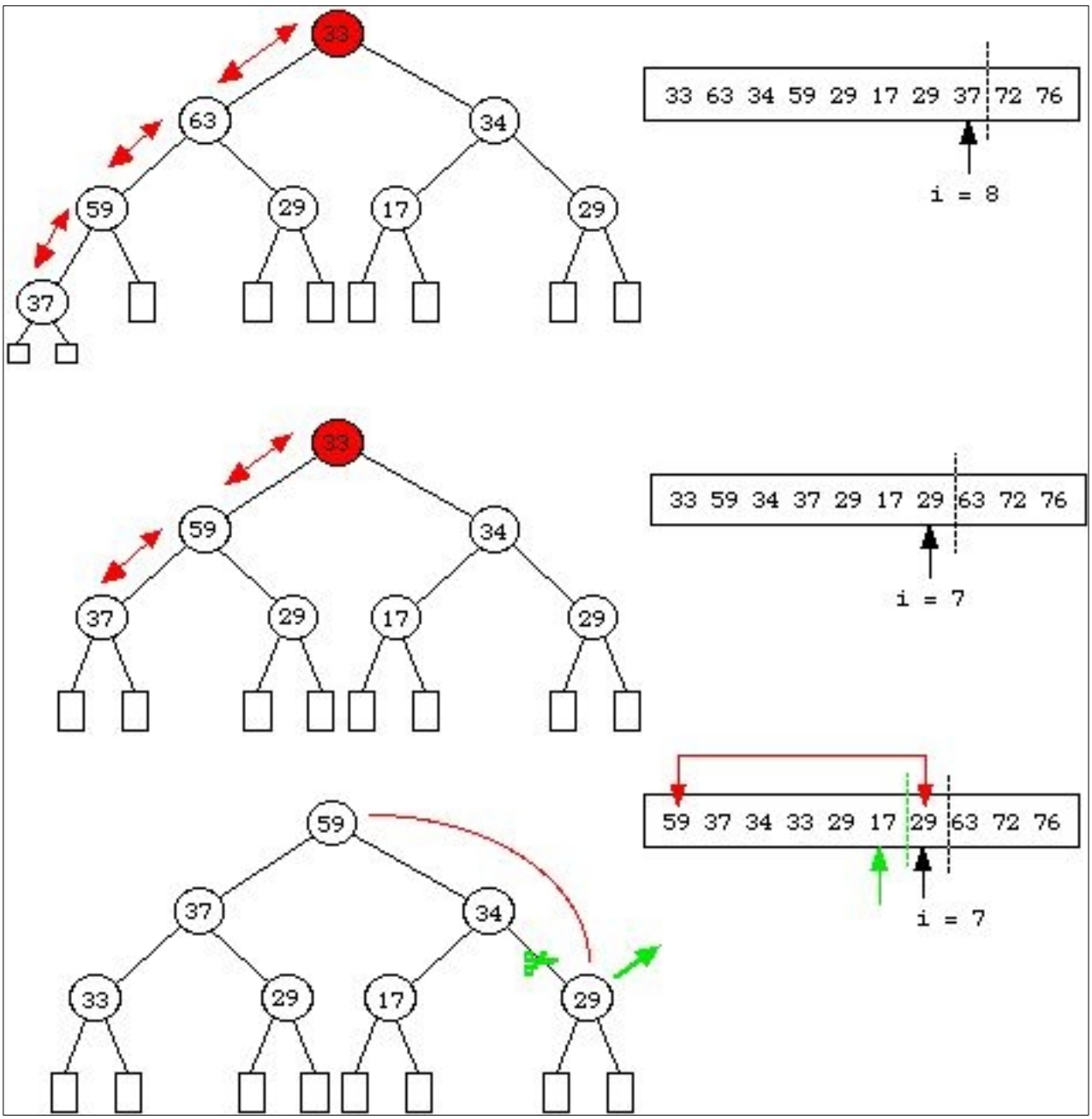
7.6.2. Funcionament de l'algorisme de heapsort

7.6.2.1. Creació del MAXHEAP



7.6.2.2. Ordenació del MAXHEAP




33
63
34
59
29
17
29
37
33 63 34 59 29 17 29 37 72 76
i = 8
33
59
34
37
29
17
29
33 59 34 37 29 17 29 63 72 76
i = 7
59
37
34
33
29
17
29
59 37 34 33 29 17 29 63 72 76
i = 7

7.6.3. Implementació

Per implementar aquest algorisme podem usar la classe CuaPrio vista anteriorment o fer-ho directament sobre el vector.

7.6.3.1. Usant la classe **CuaPrio**

Ordenació usant la classe CuaPrio.

```
template <typename T>
void heap_sort(T A[], nat n) throw(error){
  CuaPrio<T, T> c;
  for (nat i=0; i < n; ++i) {   // crea el heap
    c.inserir(A[i], A[i]);
  }
  for (nat i=n-1; i >= 0; --i) {
    A[i] = c.max();
    c.elim_max();
  }
}
```

7.6.3.2. Sense usar la classe **CuaPrio**

Ordenació sense usar la classe CuaPrio.

```
template <typename T>
void heapsort(T A[], nat n) {
  // Dóna estructura de max-heap al vector A de T's;
  // aquí cada element s'identifica com la seva
  // prioritat.
  for (nat i=n/2; i>0; --i) { // crea el maxheap
    enfonsar (A, n, i);
  }
  nat i=n;
  while (i > 0) {
    swap(A[1], A[i]);
    --i;
    // es crida un mètode per enfonsar el valor
    enfonsar(A, i, 1);
  }
}
```

Es pot comprovar que la primera opció és menys eficient ja que es fa una còpia del vector.

7.6.4. Cost

El cost d'aquest algorisme és $\Theta(n \log n)$ (fins i tot en el cas pitjor) si tots els elements són diferents.

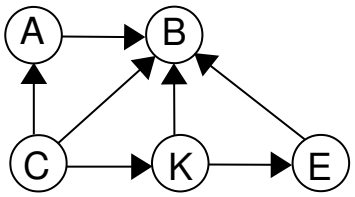
A la pràctica el seu cost és superior al de quicksort, donat que el factor constant multiplicatiu del terme *n log n* és més gran.

TEMA 8. GRAFS

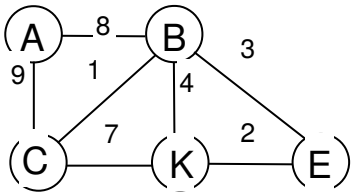
8.1. INTRODUCCIÓ

- Un graf s'utilitza per representar relacions arbitràries entre objectes del mateix tipus.
- Els objectes reben el nom de nodes o **vèrtexs** i les relacions entre ells s'anomenen **arestes**.
- Un graf G format pel conjunt de vèrtexs V i pel conjunt d'arestes A, es denota pel parell G=(V,A).
- Existeixen grafs **dirigits** i **no dirigits** depenent de si les arestes estan orientades o no ho estan.
- Existeixen grafs **etiquetats** i **no etiquetats** en funció de si les arestes tenen o no informació associada.

Graf dirigit i no etiquetat



Graf no dirigit i etiquetat



Definició: Donat un domini V de vèrtexs i un domini E d'etiquetes, definim un graf dirigit i etiquetat G com una funció que associa etiquetes a parells de vèrtexs.

$$G \in \{f: V \times V \rightarrow E\}$$

Per a un graf no etiquetat la definició és similar: G és una funció que associa un booleà a parells de vèrtexs.

8.2. DEFINICIONS

8.2.1. Adjacències

Sigui G=(V,A) un graf NO DIRIGIT. Sigui v un vèrtex de G. Es defineix:

- **Adjacents** de v, adj(v) = { v'∈V | (v,v') ∈A }
- **Grau** de v, grau(v) = | adj(v) |

Sigui G=(V,A) un graf DIRIGIT. Sigui v un vèrtex de G. Es defineix:

- **Successors** de v, succ(v) = { v'∈V | (v,v')∈A }
- **Predecessors** de v, pred(v) = { v'∈V | (v',v)∈A }
- **Adjacents** de v, adj(v) = succ(v) ∪ pred(v)
- **Grau** de v, grau(v) = | |succ(v)| - |pred(v)| |
- **Grau d'entrada** de v , grau_e(v) = |pred(v)|
- **Grau de sortida** de v, grau_s(v) = |succ(v)|

8.2.2. Camins

Un **camí** de longitud $n \geq 0$ en un graf G=(V,A) és una successió $\{v_0, v_1, ..., v_n\}$ tal que:

- Tots els elements de la successió són vèrtexs, es a dir, $\forall i: 0 \leq i \leq n: v_i \in V$
- Existeix aresta entre tot parell de vèrtexs consecutius de la successió, o sigui, $\forall i: 0 \leq i < n: (v_i, v_{i+1}) \in A$.

Donat un camí $\{v_0, v_1, ..., v_n\}$ es diu que:

- Els seus **extrems** són v_0 i v_n i la resta de vèrtexs s'anomenen **intermedis**.
- És **propi** si $n>0$. Hi ha, com a mínim, dos vèrtexs en la seqüència i, per tant, la seva longitud és ≥ 1.
- És **obert** si $v_0 \neq v_n$
- És **tancat** si $v_0 = v_n$
- És **simple** si no es repeteixen arestes
- És **elemental** si no es repeteixen vèrtexs, excepte potser els extrems.

Tot camí elemental és simple (si no es repeteixen els vèrtexs segur que no es repeteixen les arestes).

Un **cicle elemental** és un camí tancat, propi i elemental (una seqüència de vèrtexs, de longitud major que 0, en la que coincideixen els extrems i no es repeteixen ni arestes ni vèrtexs).

8.2.3. Connectivitat

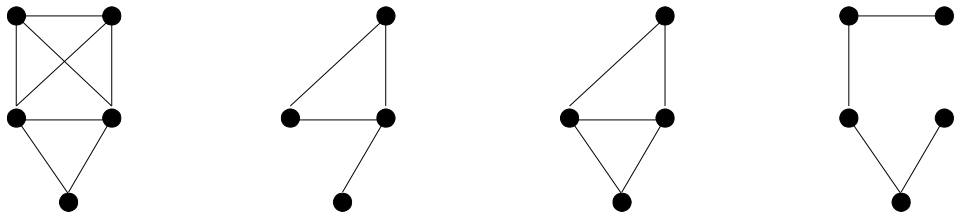
Sigui G=(V,A) un graf NO DIRIGIT. Es diu que:

- És **connex** si existeix camí entre tot parell de vèrtexs.
- És un **bosc** si no conté cicles.
- És un **arbre no dirigit** si és un bosc connex.

Sigui G=(V,A) un graf DIRIGIT. Es diu que:

- És **fortament connex** si existeix camí entre tot parell de vèrtexs en ambdós sentits.
- Un **subgraf** del graf G=(V,A), és el graf H=(W,B) tal que W ⊆ V i B ⊆ A i B ⊆ WxW.
- Un **subgraf induït** del graf G=(V,A) és el graf H=(W,B) tal que W ⊆ V i B conté aquelles arestes de A tal que els seus vèrtexs pertanyen a W.
- Un **arbre lliure** del graf G=(V,A) és un subgraf d'ell, H=(W,B), tal que és un arbre no dirigit i conté tots els vèrtexs de G, es a dir, W=V.

Exemple:



G = (V,A) | Subgraf de G | Subgraf induït de G | Arbre lliure de G

8.2.4. Alguns grafs particulars

8.2.4.1. Graf complet

Un graf G=(V,A) és **COMPLET** si existeix aresta entre tot parell de vèrtexs de V.

- El número d'arestes, |A|, d'un graf complet dirigit és exactament |A|=n·(n-1).
- Si es tracta d'un graf no dirigit llavors |A|=n·(n-1)/2.

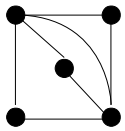
8.2.4.2. Graf eularià

Es diu que un graf G=(V,A) és **EULERIÀ** si existeix un camí tancat, de longitud major que zero, simple (no es repeteixen arestes) però no necessàriament elemental, que inclogui totes les arestes de G.

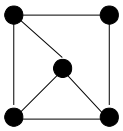
Lema: Un graf no dirigit i connex és eulerià si i només si el grau de tot vèrtex es parell.

Lema: Un graf dirigit i fortament connex és eulerià si i només si el grau de tot vèrtex és zero.

Exemple:



Eulerià

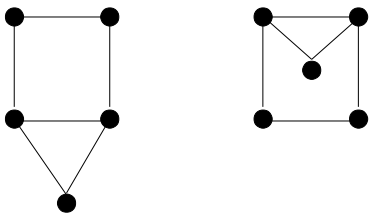


NO Eulerià

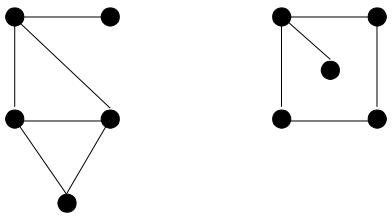
8.2.4.3. Graf hamiltonià

Un graf G=(V,A) és **HAMILTONIÀ** si existeix un camí tancat i elemental (no es repeteixen vèrtexs) que conté tots els vèrtexs de G. Si existeix, el camí es diu circuit hamiltonià.

Exemple:



Hamiltonians



NO Hamiltonians

8.3. ESPECIFICACIÓ

8.3.1. Definició de la classe

Especificació d'un graf dirigit i etiquetat amb operacions que permeten afegir, consultar i eliminar vèrtexs i arestes. Alguns mètodes no caldrien en cas que es volgués representar algun altre tipus de graf (p. ex: si fos no dirigit no caldrien els mètodes successors ni predecessors).

> **IMPORTANT!!**
> Les classes V i E han de tenir definides la constructora per còpia, assignació, la destructora i els operadors de comparació ==, !=.

```
template <typename V, typename E>
class graf {
public:
```

```
Generadores: constructora, afegeix vèrtex i afegeix aresta. En
el cas del mètode af_aresta genera un error en el cas de si
algun dels vèrtexs indicat no existeix o els dos vèrtexs són
iguals (no es permet que una aresta tingui el mateix origen i
destí).
```

```
graf() throw(error);
void afegeix_vertex(const V &v) throw(error);
void afegeix_aresta(const V &u, const V &v, const E &e)
  throw(error);
```

```
Tres grans.
```

```
graf(const graf &g) throw(error);
graf& operator=(const graf &g) throw(error);
~graf() throw();
```

```
  Consultores: valor de l'etiqueta d'una aresta, existeix vèrtex,
  existeix aresta i vèrtexs successors d'un donat. En el cas dels
  mètodes valor, existeix_aresta i adjacents generen un error si
  algun dels vèrtexs indicat no existeix.
  E valor(const V &u, const V &v) const throw(error);
  bool existeix_vertex(const V &v) const throw();
  bool existeix_aresta(const V &u, const V &v) const
    throw(error);
  void adjacents(const V &v, list<V> &l) const
    throw(error);

  Consultores: obté una llista amb els vèrtexs successors o
  predecessors del vèrtex indicat. En el cas que el vèrtex no
  tingui successors o predecessors la llista serà buida. Aquests
  mètodes generen un error si el vèrtex indicat no existeix.
  void successors(const V &v, list<V> &l) const
    throw(error);
  void predecessors(const V &v, list<V> &l) const
    throw(error);

  Constructores modificadores: elimina vèrtex i elimina aresta.
  En cas que el vèrtex o l'aresta no existeixi no fa res.
  void elimina_vertex(const V &v) throw();
  void elimina_aresta(const V &u, const V &v) throw();

  Gestió d'errors.
  const static int VertexNoExisteix = 800;
  const static int MateixOrigeniDestí = 801;

  Definició d'aresta i vertexs.
  typedef aresta pair<V, V>;
  typedef vertexs conjunt<V>;

private:
  vertexs _verts;
  ...

};
```

Recordem com seria l'especificació de la classe conjunt.

```
template <typename T>
class conjunt {
public:
  ...

  class iterator {
    friend class conjunt;
    iterator() throw();
    T& operator*() const throw(error);
    iterator& operator++() throw();
    ...
  };

  iterator begin() const throw();
  iterator end() const throw();
  nat size() const throw();
};
```

8.3.2. Enriquiment de l'especificació

Per exemple podríem afegir les següents operacions:

- ex-camí: Aquesta operació retorna un booleà que indica si existeix camí entre dos vèrtexs donats en un graf dirigit.
- descend: Descendents d'un vèrtex v. Aquesta operació retorna el conjunt de tots aquells vèrtexs d'un graf dirigit tals que existeix camí des de v fins a qualsevol d'ells.

8.4. IMPLEMENTACIONS DE GRAFS

8.4.1. Consideracions prèvies

Donat un graf G=(V,A) denotem per:

- n=|V| el nombre de vèrtexs o nodes del graf
- a=|A| el nombre d'arestes del graf

IMPORTANT!!
Gairebé totes les operacions de grafs tenen com arguments un o dos vèrtexs. Suposarem que mitjançant alguna de les tècniques de diccionaris existents s'ha implementat de forma eficient la inserció, consulta i eliminació d'un vèrtex a partir del seu identificador.

8.4.2. Matrius d'adjacència

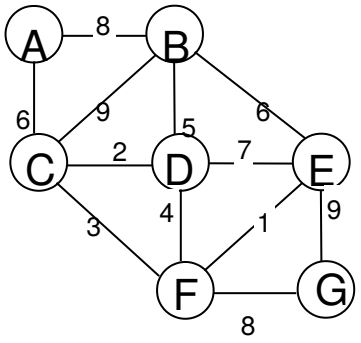
Si el graf G=(V,A) és no etiquetat s'implementa en una matriu de booleans M[1..n,1..n] de forma que ($\forall v,w \in V$: M[v,w]=CERT $\Leftrightarrow$ $(v,w) \in A$).

Si el graf està etiquetat serà necessari, en lloc d'una matriu de booleans, una matriu del tipus de les etiquetes del graf.

L'espai ocupat per la matriu és de l'ordre de $\Theta(n^2)$:

- Un graf no dirigit només necessita la meitat de l'espai $(n^2-n)/2$, doncs la matriu d'adjacència és simètrica i tan sols cal emmagatzemar la part triangular superior.
- Un graf dirigit el necessita tot excepte la diagonal, és a dir, n^2-n.

Exemple: graf no dirigit i etiquetat implementat amb una matriu:



	A	B	C	D	E	F	G
A		8	6				
B			9	5	6		
C				2		3	
D					7	4	
E						1	9
F							8
G							

8.4.2.1. Cost de les matrius d'adjacència

El cost temporal de les operacions bàsiques del graf són:

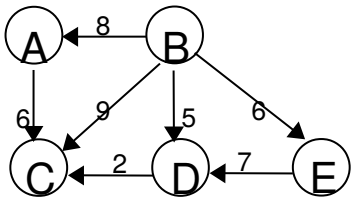
- Crear el graf (*constructora*) és $\Theta(n^2)$.
- Afegir aresta (*afegeix_aresta*), existeix aresta (*existeix_aresta*), eliminar aresta (*elimina_aresta*), valor d'una etiqueta (*valor*) són $\Theta(1)$.
- Adjacents a un vèrtex (*adjacents*) és $\Theta(n)$.
- Per a un graf dirigit el cost de calcular els successors (*successors*) o els predecessores (*predecessors*) d'un vèrtex donat és $\Theta(n)$.

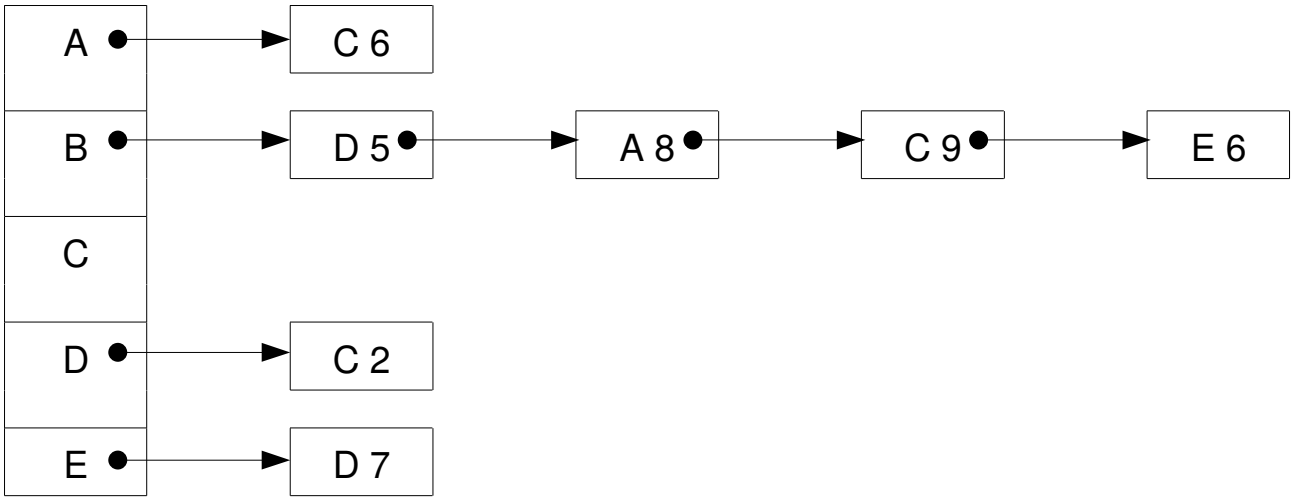
En general, si el nombre d'arestes del graf és elevat, les matrius d'adjacència tenen bons costos espacial i temporal per a les operacions habituals.

8.4.3. Llistes d'adjacència

En el cas de les llistes d'adjacència, l'estructura que s'utilitza per a implementar un graf G=(V,A) és un vector L[1..n] tal que L[i], amb $1 \leq i \leq n$, és una llista formada pels identificadors dels vèrtexs que són adjacents (successors, si el graf és dirigit) al vèrtex amb identificador i.

Exemple: graf dirigit i etiquetat implementat amb una llista:





L'espai ocupat per aquesta implementació és de l'ordre $\Theta(n+a)$.

8.4.3.1. Cost de les llistes d'adjacència

Examinant el cost temporal d'algunes operacions bàsiques, s'obté:

- Crear l'estructura (*constructora*) és $\Theta(n)$.
- Afegir aresta (*afegeix_aresta*), existència d'aresta (*existeix_aresta*) i eliminar aresta (*elimina_aresta*) necessiten comprovar si existeix l'aresta. En el pitjor cas requerirà recórrer la llista completa, que pot tenir una grandària màxima de n-1 elements. Per tant, tenen un cost $\Theta(n)$.

Una implementació de la llista d'arestes en un AVL (arbre binari equilibrat) permet reduir el cost d'aquesta operació a $\Theta(\log n)$.

El cost de les operacions que calcula els successors (*successors*) i predecessors (*predecessors*) d'un vèrtex donat en un graf DIRIGIT és el següent:

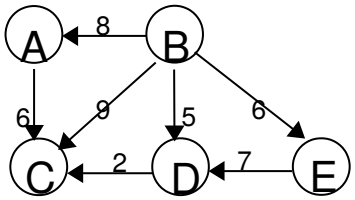
- Per obtenir els successores d'un vèrtex només cal recórrer tota la seva llista associada, per tant, $\Theta(n)$.
- Per obtenir els predecessores s'ha de recórrer, en el pitjor cas, tota l'estructura per veure en quines llistes apareix el vèrtex del que s'estan buscant els seus predecessores, per tant, $\Theta(n+a)$.

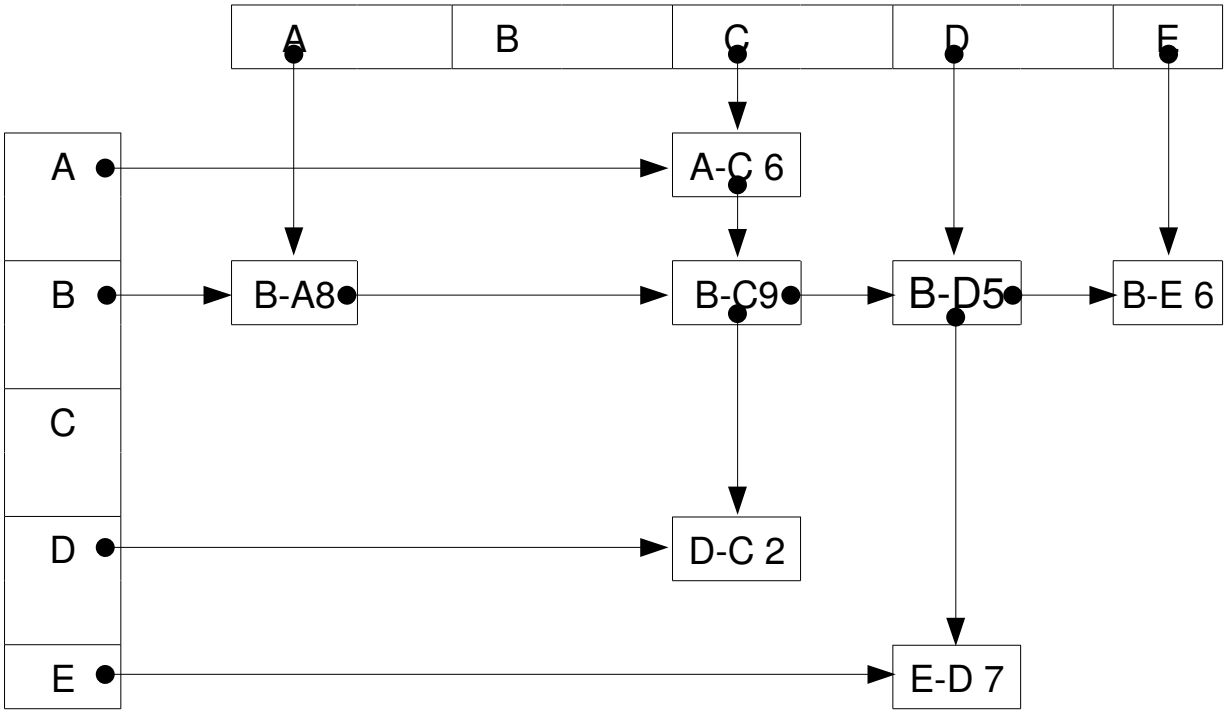
Si les llistes de successors d'un vèrtex estan implementades amb un AVL llavors el cost de l'operació predecessors és $\Theta(n \cdot \log n)$.

8.4.4. Multillistes d'adjacència

La implementació d'un graf usant multillistes només te sentit per grafs dirigits. El seu objectiu és millorar el cost de l'operació que obté els predecessores per que passi a tenir cost $\Theta(n)$ en lloc del cost $\Theta(n+a)$. S'aconsegueix, a més a més, que el cost de la resta d'operacions sigui el mateix que el que es tenia utilitzant llistes.

Exemple: graf dirigit i etiquetat implementat amb una multillista:





En general, per implementar un graf és convenient utilitzar llistes d'adjacència quan:

- El graf és poc dens.
- El problema a resoldre ha de recórrer totes les arestes.

8.5. RECORREGUTS SOBRE GRAFS

8.5.1. Recorregut en profunditat prioritària

L'algorisme de **recorregut en profunditat prioritària** (anglès: *depth-first search* o *DFS*), es caracteritza perquè permet recórrer completament el graf, tots els vèrtexs i totes les arestes.

- Si el graf és no dirigit, es passa 2 cops per cada aresta (un en cada sentit).
- Si és dirigit es passa un sol cop per cada aresta.

Els vèrtexs es visiten aplicant el criteri 'primer en profunditat'.

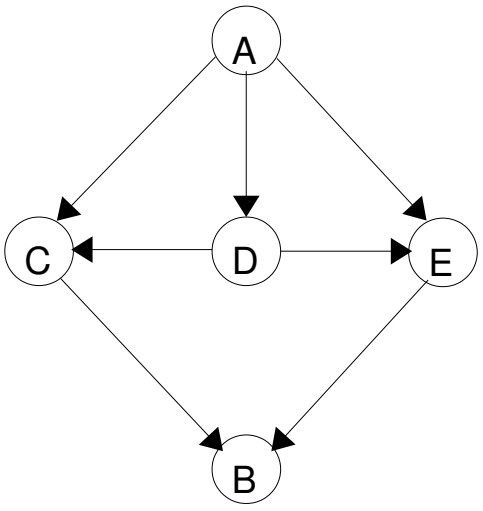
Primer en profunditat significa que donat un vèrtex v que no hagi estat visitat, DFS primer visitarà v i després, recursivament aplicarà DFS sobre cadascun dels adjacents/successors d'aquest vèrtex que encara no hagin estat visitats.

S'inicia el recorregut començant per un vèrtex qualsevol i acaba quan tots els vèrtexs han sigut visitats.

L'ordre dels vèrtexs que produeix el DFS no és únic.

Exemple:

Recorreguts en profunditat (els recorreguts s'inicien en el vèrtex A):



Recorregut en profunditat prioritària (DFS): A, C, B, D, E

Recorregut en profunditat cap enrere: B, C, E, D, A

COM AFEGIR MÉS INFORMACIÓ AL VÈRTEX

Com es pot afegir un nou camp d'informació (vist) a una classe de la qual desconeixem el tipus? La resposta és ben simple: NO es pot. Però aquest problema té solució. Es pot encapsular tota la informació en una nova classe node i fer que un atribut d'aquest node sigui el vèrtex. D'aquesta manera podríem afegir al node tota la informació extra que necessitéssim.

- A cada vèrtex cal que tinguem associat un valor que indiqui si el recorregut ha passat per ell. Inicialment tots els vèrtexs han de tenir aquest valor a '**no visitat**'.
- En el moment en que el recorregut arriba a un vèrtex amb aquest valor, el modifica posant-lo a '**visitat**'. Significa que s'ha arribat a aquest vèrtex per un, el primer explorat, dels camins que menen a ell i que es ve d'un vèrtex que ja havia estat visitat (excepte pel vèrtex inicial).
- Mai més s'arribarà a aquest vèrtex pel mateix camí, encara que es pugui arribar per altres camins diferents.

8.5.1.2. Implementació

```
template <typename V, typename E>
void graf<V, E>::rec_prof() throw() {
  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    (*v).vist = false;
  }

  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    if (not (*v).vist) {
      rec_prof(*v);
    }
  }
}


template <typename V, typename E>
void graf<V, E>::rec_prof(V &u) throw() {
  u.vist = true;

  TRACTAR( u );            /* PREWORK */

  list<V> l;
  adjacents(u, l);
  // w avaluarà a false quan arribem a l'últim vèrtex
  for (list<V>::iterator w=l.begin(); w!=l.end(); ++w) {
    if (not (*w).vist) {
      rec_prof (w);
    }

    TRACTAR( u, w );       /* POSTWORK */

  }

  /* Hem marcat a vist tots els vèrtexs del graf
     accessibles des d'u per camins formats exclusivament
     per vèrtexs que no estaven vistos el graf */
}
```

8.5.1.3. Cost de l'algorisme

Suposem que prework i postwork tenen cost constant.

Depenent de quina sigui la representació del graf tindrem els següents costos:

- **matriu d'adjacència**: $\Theta(n^2)$
- **llistes d'adjacència**: $\Theta(n+a)$ degut a que es recorren totes les arestes dues vegades, una en cada sentit, i a que el bucle exterior recorre tots els vèrtexs.

Existeix una clara relació entre el recorregut en preordre d'un arbre i el DFS sobre un graf.

Es pot fer un recorregut simètric al DFS que consisteixi en no visitar un node fins que no hem visitat tots els seus descendents. Seria la generalització del recorregut en postordre d'un arbre. S'anomena "**Recorregut en profunditat cap enrere**".

8.5.2. Recorregut en amplada

L'algorisme de **recorregut en amplada** (anglès: *breadth-first search* o *BFS*) generalitza el concepte de recorregut per nivells d'un arbre:

- Després de visitar un vèrtex es visiten els successors, després els successors dels successors, i així reiteradament.
- Si després de visitar tots els descendents del primer node encara en queden per visitar, es repeteix el procés començant per un dels nodes no visitats.

8.5.2.1. Descripció de l'algorisme

L'algorisme utilitza una cua per a poder aplicar l'estratègia exposada. Cal marcar el vèrtex com a vist quan l'encuem per no encuar-lo més d'una vegada.

L'acció principal és idèntica a la del DFS.

8.5.2.2. Implementació

```
template <typename V, typename E>
void graf<V, E>::rec_amplada() throw() {
  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    (*v).vist = false;
  }
  for (conjunt<T>::iterador v = _verts.begin();
        v != verts.end(); ++v) {
    if (not (*v).vist) {
      rec_amplada (*v);
    }
  }
}
```

```
template <typename V, typename E>
void graf<V, E>::rec_amplada(V &u) throw() {
  cua<V> c;
  c.encua(u);
  u.vist = true;
  while (not c.es_buida()) {
    V v = c.cap();
    c.desencua();
    TRACTAR(v);
    list<V> l;
    adjacents(v, l);
    for (list<V>::iterator w= l.begin();
         w != l.end(); ++w) {
      if (not (*w).vist) {
        c.encua(w);
        (*w).vist = true;
      }
    }
  }
}
```

8.5.2.3. Cost de l'algorisme

Aquesta versió té el mateix cost que l'algorisme DFS:

- **matriu d'adjacència**: $\Theta(n^2)$
- **llistes d'adjacència**: $\Theta(n+a)$

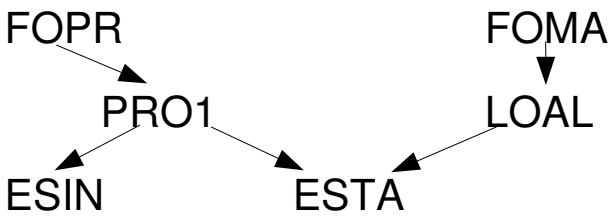
8.5.3. Recorregut en ordenació topològica

L'**ordenació topològica** és un recorregut només aplicable a grafs dirigits acíclics.

Un vèrtex només es visita si han estat visitats tots els seus predecessors dins del graf. Per tant, en aquest recorregut les arestes defineixen una restricció més forta sobre l'ordre de visita dels vèrtexs.

Exemple 1:

Pla d'estudis amb prerequisits a les assignatures.

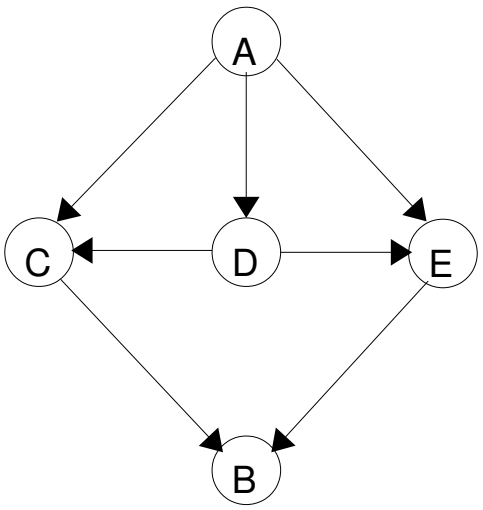


Quan un alumne es matricula ha de tenir en compte que:

- En començar només pot cursar les assignatures que no tinguin cap prerequisit (cap predecessor).
- Només pot cursar una assignatura si han estat cursats prèviament tots els seus prerequisits (tots els seus predecessors).

Exemple 2:

Recorreguts en profunditat (els recorreguts s'inicien en el vèrtex A):



Recorregut en ordenació topològica: A, C, D, E, B

Vector inicial de predecessors:

A	B	C	D	E
0	2	2	1	2

8.5.3.1. Descripció de l'algorisme

- El funcionament de l'algorisme consisteix en anar escollint els vèrtexs que no tenen predecessors (que són els que es poden visitar) i, a mesura que s'escullen, s'esborren les arestes que en surten. La implementació directa d'això és costosa ja que cal cercar els vèrtexs sense predecessors.
- Una alternativa consisteix en calcular prèviament quants predecessors té cada vèrtex, emmagatzemar el resultat en un vector i actualitzar-lo sempre que s'incorpori un nou vèrtex a la solució. El vector guarda el nombre de predecessors de cada vèrtex que no són a la solució. Només escollirem aquells vèrtexs que tinguin un 0 en el vector (els que pertanyen al conjunt *zeros*).

8.5.3.2. Implementació

```
template <typename V, typename E>
void graf::rec_ordenacio_topologica()
  conjunt<V> zeros; // crea el conjunt buit
  nat num_pred = new nat[_verts.size()];

  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    num_pred[*v] = 0;
    zeros.afegeix(*v);    // afegeix tots els vèrtexs
  }
  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    list<V> l;
    successors(v, l);
    for (list<V>::iterator w=l.begin(); w != l.end();
                                ++w) {
      ++num_pred[*w];
      zeros.elimina(*w); // elimina els vèrtexs amb pred.
    }
  }
  while (not zeros.es_buit()) {
    V v = *(zeros.begin());
    zeros.elimina(v);
    TRACTAR( v );
    list<V> l;
    successors(v, l);
    for (list<V>::iterator w= l.begin(); w != l.end();
                                ++w) {
      --num_pred[*w];
      if (num_pred[*w]==0)
        zeros.afegeix(*w);
      }
    }
  }
}
```

8.5.3.3. Cost de l'algorisme

El cost temporal és el mateix que el de l'algorisme DFS:

- **matriu d'adjacència**: $\Theta(n^2)$
- **llistes d'adjacència**: $\Theta(n+a)$

8.6. CONNECTIVITAT I CICLICITAT

8.6.1. Connectivitat

8.6.1.1. Problema

Determinar si un graf no dirigit és connex o no.

> **Propietat**
>
> Si un graf és connex és possible arribar a tots els vèrtexs del graf començant un recorregut per qualsevol d'ells.
> Es necessita un algorisme que recorri tot el graf i que permeti esbrinar a quins vèrtexs es pot arribar a partir d'un donat. Per això utilitzarem un DFS.

8.6.1.2. Descripció de l'algorisme

La solució proposada per aquest problema determina el nombre de components connexes del graf. Si el nombre de components és major que 1 llavors G no és connex.

A més a més, tots els vèrtexs d'una component connexa s'etiqueten amb un natural que és diferent per cadascuna de les components connexes que té el graf inicial.

8.6.1.3. Implementació

```
template <typename V, typename E>
nat graf<V, E>::compo_connexes() throw() {
  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    (*v).vist = false;
    (*v).nc = 0; // Nº de component a la que pertany el
  }              // vèrtex.
```

```
  nc = 0;
  for (vertexs::iterador v = _verts.begin();
         v != _verts.end(); ++v) {
    if (not (*v).vist) {
      ++nc;
      connex(*v, nc);
    }
  }
  return nc;
}

template <typename V, typename E>
void graf<V, E>::connex(V &u, nat numc) throw() {
  /* Anotar el número de component connexa a la que
     pertany el vèrtex és el PREWORK. */
  u.vist = true;
  u.nc = numc;

  list<V> l;
  adjacents(u, l);
  // w avaluarà a false quan arribem a l'últim vèrtex
  for (list<V>::iterator w = l.begin();
         w != l.end(); ++w) {
    if (not (*w).vist) {
      connex (*w, numc);
    }
  }
  /* Hem marcat a vist tots els vèrtexs del graf
     accessibles des d'u per camins formats exclusivament
     per vèrtexs que no estaven vistos, segons l'ordre
     de visita del recorregut en profunditat. També se'ls
     ha afegit una informació que indica a quin nº de
     component connexa pertanyen. */
}
```

8.6.1.4. Cost de l'algorisme

El cost d'aquest algorisme és idèntic al de REC-PROF.

8.6.2. Test de ciclicitat

8.6.2.1. Problema

Donat un graf dirigit, determinar si té cicles o no.

> **Propietat**
>
> En un graf amb cicles, en algun moment durant el recorregut s'arriba a un vèrtex vist anteriorment per una aresta que no estava visitada.

8.6.2.2. Descripció de l'algorisme

El mètode que segueix l'algorisme proposat és mantenir, pel camí que va des de l'arrel al vèrtex actual, quins vèrtexs en formen part. Si un vèrtex apareix més d'una vegada en aquest camí és que hi ha cicle.

8.6.2.3. Implementació

```
template <typename V, typename E>
bool graf<V, E>::cicles() throw() {
  for (vertexs::iterador v = _verts.begin();
         v != _verts.end(); ++v) {
    (*v).vist = false;
    (*v).en_cami = false;   // no hi ha cap vèrtex en el
  }                         // camí de l'arrel al vèrtex actual

  bool trobat = false;    // inicialment no hi ha cicles
  vertexs::iterador v = _verts.begin();
  while (v != _verts.end() and not trobat) {
    if (not (*v).vist) {
      trobat = cicles(*v);
    }
    ++v;
  }
  return trobat;
}
```

```
template <typename V, typename E>
bool graf<V, E>::cicles(V &u) throw() {
  /* PREWORK: Anotem que s'ha visitat u i que aquest es
     troba en el camí de l'arrel a ell mateix */
  u.vist = true;
  u.en_cami = true;

  bool trobat = false;
  list<V> l;
  successors(u, l);
  list<vertex>::iterator w = l.begin();
  while (w != l.end() and not trobat) {
    if ((*w).vist) {
      if (w.en_cami) {
        // Hem trobat un cicle! Es recorre l'aresta (u,w)
        // però ja existia camí de w a u
        trobat = true;
      }
    }
    else {
      trobat = cicles(*w);
    }
    ++w;
  }

  /* POSTWORK: Ja s'ha recorregut tota la descendència
     d'u i, per tant, s'abandona el camí actual des de
     l'arrel (es va desmuntant el camí). */
  u.en_cami = false;

  return trobat;

  /* Hem marcat a 'vist' tots els vèrtexs accessibles des
     d'u per camins formats exclusivament per vèrtexs que
     no estaven vistos. b indicarà si en el conjunt de
     camins que tenen en comú des de l'arrel fins u i
     completat amb la descendència d'u hi ha cicles. */
}
```

8.6.2.4. Cost de l'algorisme

Aquesta versió té el mateix cost que l'algorisme REC-PROF.

8.7. ARBRES D'EXPANSIÓ MÍNIMA

8.7.1. Introducció

Sigui G=(V,A) un graf no dirigit, etiquetat amb valors naturals i connex. Dissenyar un algorisme que calculi un subgraf de G, T=(V,B) amb B $\subseteq$ A, connex i sense cicles, tal que la suma dels pesos de les arestes de T sigui mínima.

El subgraf T que s'ha de calcular s'anomena l'arbre lliure associat a G o **arbre d'expansió mínima** de G (anglès: *minimun spaning tree (MST)*).

8.7.2. Algorisme de Kruskal

L'algorisme de Kruskal construeix el MST partint d'un subgraf de G, T=(V,conjunt_buit), i a base d'afegir una aresta de G cada vegada a T, el fa créixer fins que el subgraf T=(V,B) és el MST desitjat.

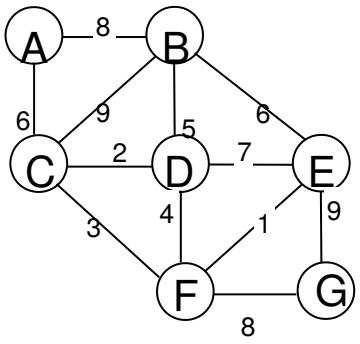
A cada pas s'escull aquella aresta de valor mínim, d'entre totes les que no s'han processat, tal que connecti dos vèrtexs que formen part de dos components connexes diferents (de dos grups diferents de vèrtexs de T).

Això dóna garanties de que no es formen cicles i que T sempre és un bosc format per arbres lliures.

Per reduir el cost del procés de selecció de l'aresta de valor mínim, s'ordena prèviament les arestes de G per ordre creixent del valor de la seva etiqueta.

Exemple:

Sigui G el graf de la següent figura:



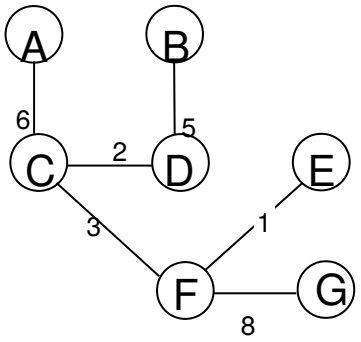
La llista de les arestes de G ordenades per ordre creixent de valor és:

F-E	C-D	C-F	D-F	D-B	A-C	B-E	D-E	A-B	F-G	B-C	G-E

La taula següent mostra el treball de les 10 iteracions que realitza el bucle.

Inicialment cada vèrtex pertany a un arbre diferent i al final tots formen part del mateix arbre, el MST. Les arestes examinades i no refusades són les arestes que formen el MST.

Etapa	**Aresta examinada**	**Grups de vèrtexs**
Inicialització	---	{A}, {B}, {C}, {D}, {E}, {F}, {G}
1	F-E valor 1	{A}, {B}, {C}, {D}, {E, F}, {G}
2	C-D valor 2	{A}, {B}, {C, D}, {E, F}, {G}
3	C-F valor 3	{A}, {B}, {C, D, E, F}, {G}
4	D-F valor 4	Aresta refusada (forma cicle)
5	D-B valor 5	{A}, {B, C, D, E, F}, {G}
6	A-C valor 6	{A, B, C, D, E, F}, {G}
7	B-E valor 6	Aresta refusada (forma cicle)
8	D-E valor 7	Aresta refusada (forma cicle)
9	A-B valor 8	Aresta refusada (forma cicle)
10	F-G valor 8	{A, B, C, D, E, F, G}



pes (T) = 25

8.7.2.1. Implementació

- En la implementació de l'algorisme de Kruskal, s'utilitza la variable MF que és del tipus partició (MFSet) i que s'usa per mantenir els diferents grups de vèrtexs (blocs de la partició).
- Per tal de poder emprar la classe genèrica partició la classe V (vèrtex) cal que admeti la conversió a int, és a dir, donat un valor de v de la classe, **int(v)** és una expressió vàlida.
- A més a més, necessitem a la classe graf un mètode anomenat arestes que ens retorni totes les arestes del graf.
- Suposem que G=(V,A) és un graf no dirigit, connex i etiquetat amb valors naturals.

```
template <typename V, typename E>
void graf<V, E>::kruskal(conjunt<aresta> &T) throw() {
  MFSet<V> MF;
  list<aresta> A0 = arestes();
  ordenar_creixentment(A0);

  // A l'inici cada vèrtex forma part d'un grup diferent
  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    mf.afegeix(*v);
  }

  list<aresta>::iterator it = begin();
  while (T.size() < _verts.size()-1) {
    aresta p = *it; // p=(u,v) és l'aresta de valor mínim
    ++it;
    V u = p.first();
    V v = p.second();
    V x = MF.find(u);
    V y = MF.find(v);
    if (x != y) {
      MF.union(x, y);
      T.afegeix(p);
    }
  }
}
```

8.7.2.2. Cost de l'algorisme

El cost d'ordenar el conjunt d'arestes és $\Theta(a \cdot \log n)$ i el d'inicialitzar el MFSet és $\Theta(n)$.

Cada iteració realitza dos operacions *find* i, alguns cops (en total n-1 vegades) una operació *union*. El bucle, en el pitjor dels casos, realitza 2·a operacions *find* i n-1 operacions *union*. Sigui m=2a + n-1, llavors el bucle costa $O(m \cdot \log^* n)$ si el MFSet s'implementa amb alguna tècnica d'unió per rang o per pes i de compressió de camins. Aproximadament és $O(a \cdot \log^* n)$.

Com que el cost del bucle és menor que el cost de les inicialitzacions, el cost de l'algorisme és $\Theta(a \cdot \log n)$.

8.8. ALGORISMES DE CAMINS MÍNIMS

8.8.1. Introducció

Existeix una col·lecció de problemes, denominats **problemes de camins mínims** (anglès: *shortest-paths problems*), basats en el concepte de camí mínim.

Sigui G=(V,A) un graf dirigit i etiquetat amb valors naturals. Es defineix el **pes** del camí p, p=<v_0,v_1,v_2, ...,v_k>, com la suma dels valors de les arestes que el composen:

$$pes(p)=\sum_{i=1}^{k} valor(G, v_{i-1}, v_i)$$

Es defineix el **camí mínim** del vèrtex u al v en G, amb u,v∈V, com el camí de menor pes d'entre tots els existents entre u i v. Si existeixen diferents camins amb idèntic menor pes qualsevol d'ells és un camí mínim.

Si no hi ha camí entre u i v el pes del camí mínim és ∞.

La col·lecció de problemes de camins mínims està composada per quatre variants:

1. **Single_source shortest_paths problem**: S'ha de trobar el camí més curt entre un determinat vèrtex, source, i tots els demés vèrtexs del graf. Aquest problema es resol eficientment utilitzant l'algorisme de Dijkstra que és un algorisme voraç.

2. **Single_destination shortest_paths problem**: S'ha de trobar el camí més curt des de tots els vèrtexs a un vèrtex determinat, destination. Es resol aplicant Dijkstra al graf de partida però canviant el sentit de totes les arestes.

3. **Single_pair shortest_paths problem**: Donats dos vèrtexs determinats del graf, source i destination, trobar el camí més curt entre ells. En el cas pitjor no hi ha un algorisme millor que el propi Dijkstra.

4. **All_pairs shortest_paths problem**: S'ha de trobar el camí més curt entre els vèrtexs u i v, per tot parell de vèrtexs del graf. Es resol aplicant Dijkstra a tots els vèrtexs del graf. També es pot utilitzar l'algorisme de Floyd que és més elegant però no més eficient i que segueix l'esquema de Programació Dinàmica.

8.8.2. Algorisme de Dijkstra

L'algorisme de Dijkstra (1959) resol el problema de trobar el camí més curt entre un vèrtex donat, anomenat inicial, i tots els demés vèrtexs del graf.

8.8.2.1. Descripció de l'algorisme

L'algorisme de dijkstra funciona de la següent manera:

- El conjunt de vèrtexs del graf es troba distribuït en dos conjunts disjunts: el conjunt de VISTOS i el de NO_VISTOS.
- A VISTOS estan els vèrtexs pels que ja es coneix quina és la longitud del camí més curt entre el vèrtex inicial i ell.
- En el conjunt NO_VISTOS estan els demés vèrtexs i per cadascun d'ells es guarda la longitud del camí més curt des del vèrtex inicial fins a ell que no surt de VISTOS. Suposarem que aquestes longituds es guarden en el vector D.
- A cada iteració s'escull el vèrtex v de NO_VISTOS amb D[v] mínima de manera que v deixa NO_VISTOS, passa a VISTOS i la longitud del camí més curt entre el vèrtex inicial i v és, precisament, D[v].
- Tots aquells vèrtexs u de NO_VISTOS que siguin successors de v actualitzen convenientment el seu D[u].
- L'algorisme acaba quan tots els vèrtexs estan a VISTOS.

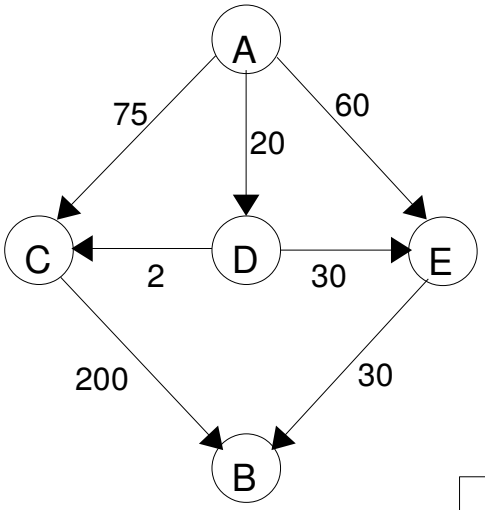
G=(V,A) és un graf etiquetat i pot ser dirigit o no dirigit. En aquest cas és dirigit. Per facilitar la lectura de l'algorisme es suposa que el graf està implementat en una matriu i que M[u][v] conté el valor de l'aresta que va del vèrtex u al v i, si no hi ha aresta, conté el valor infinit.

Exemple:

Agafant com a punt de partida el graf de la següent figura, observem com funciona l'algorisme.

S'escull com a vèrtex inicial el vèrtex amb etiqueta A.

La següent taula mostra en quin ordre van entrant els vèrtexs en el conjunt VISTOS i cóm es va modificant el vector D que conté les distàncies mínimes.



D					VISTOS
[A]	[B]	[C]	[D]	[E]	
0	∞	75	20	60	{A}
0	∞	22	**20**	50	{A, D}
0	222	**22**	**20**	50	{A, D, C}
0	80	**22**	**20**	**50**	{A, D, C, E}
0	**80**	**22**	**20**	**50**	{A, D, C, E, B}

Els valors en negreta de la taula corresponen a valors definitius i, per tant, contenen la distància mínima entre el vèrtex A i el que indica la columna.

Suposem que la distància d'un vèrtex a si mateix és zero.

8.8.2.2. Implementació

```
Calcula la distància mínima entre el vèrtex indicat i tota la
resta. Retorna un vector d'n etiquetes.
```

```
template <typename V, typename E>
E* graf::dijkstra(const V &v_ini) {
  E *D = new E[_verts.size()];
  conjunt<V> VISTOS;

  for (vertexs::iterator v = _verts.begin();
        v != _verts.end(); ++v) {
    D[*v] = M[v_ini][*v];
  }
  D[v_ini] = 0;
  VISTOS.insereix(v_ini);

  while (VISTOS.size() < n) {
    // S'obté el vèrtex u tal que no pertany a VISTOS i
    // té D mínima
    V u = minim_no_vistos(D, VISTOS);
    VISTOS.afegeix(u);
    list<V> l;
    successors(u, l);
    for (list<V>::iterator v = l.begin();
           v != l.end(); ++v) {
      if ((not VISTOS.pertany(*v)) and
          (D[*v] > D[u] + M[u][*v])) {
        D[*v] = D[u] + M[u][*v];
      }
    }
  }

  /* ∀u / u∈V:  D[u] conté la longitud del camí més curt
     des de v_ini a u que no surt de VISTOS, i com VISTOS
     ja conté tots els vèrtexs, a D hi ha les distàncies
     mínimes definitives */
  return D;
}
```

8.8.2.3. Cost de l'algorisme

El bucle que inicialitza el vector D costa $\Theta(n)$ i el bucle principal, que calcula els camins mínims, fa n-1 iteracions. En el cost de cada iteració intervenen dos factors: l'obtenció del mínim, cost lineal perquè s'ha de recórrer D, i l'ajust de D.

Respecte a aquesta última operació:

- Si el graf està implementat en una matriu d'adjacència, el seu cost també és $\Theta(n)$ i, per tant, el cost total de l'algorisme és $\Theta(n^2)$.
- Si el graf està implementat en llistes d'adjacència i s'utilitza una cua de prioritat (implementada en un min-Heap) per accelerar l'elecció del mínim, llavors seleccionar el mínim costa $\Theta(1)$, la construcció del Heap $\Theta(n)$ i cada operació d'inserció o eliminació del mínim requereix $\Theta(\log n)$.

  Com que en total s'efectuen, en el pitjor dels casos, $\Theta(a)$ operacions d'inserció i n operacions d'eliminació del mínim tenim que l'algorisme triga $\Theta((n+a)\cdot\log n)$ que resulta millor que el de les matrius quan el graf és poc dens.

8.8.2.4. Implementació utilitzant una cua de prioritat

```
Calcula la distància mínima entre el vèrtex indicat i tota la
resta. Retorna un vector d'n etiquetes.
```

```
template <typename V, typename E>
E* graf<V,E>::dijkstra(const V &v_ini) {
  E *D = new E[_verts.size()];
  cuaprioritat C;

  for (vertexs::iterador v = _verts.begin();
        v != verts.end(); ++v) {
    D[*v] = HUGE_VAL;  // ∞
  }
  D[v_ini] = 0;
  C.insereix(v_ini, D[v_ini]);

  while (not C.buida()) {
    V u = C.minim();
    C.elimina_minim();

    // Ajustar el vector D
    list<V> l;
    successors(u, l);
    for (list<V>::iterador v = l.begin();
          v != l.end(); v++) {
      if (D[*v] > D[u]+M[u][*v]) {
        D[*v] = D[u]+M[u][*v];
        if (C.hi_és(*v)) {
          C.substitueix(*v, D[*v]);
        }
        else {
          C.insereix(*v, D[*v]);
        }
      }
    }
  }
  return D;
}
```

8.8.3. Algorisme de Floyd

L'algorisme de Floyd resol el problema de trobar el camí més curt entre tot parell de vèrtexs del graf.

Aquest problema també es podria resoldre aplicant repetidament l'algorisme de Dijkstra variant el node inicial.

L'algorisme de Floyd proporciona una solució més compacta i elegant pensada especialment per a aquesta situació.

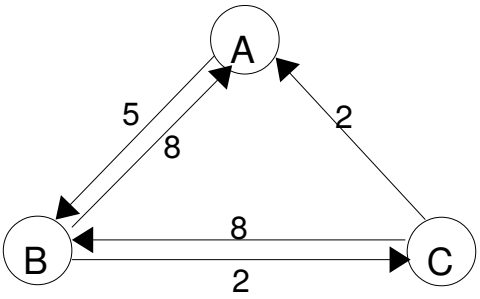
8.8.3.1. Descripció de l'algorisme

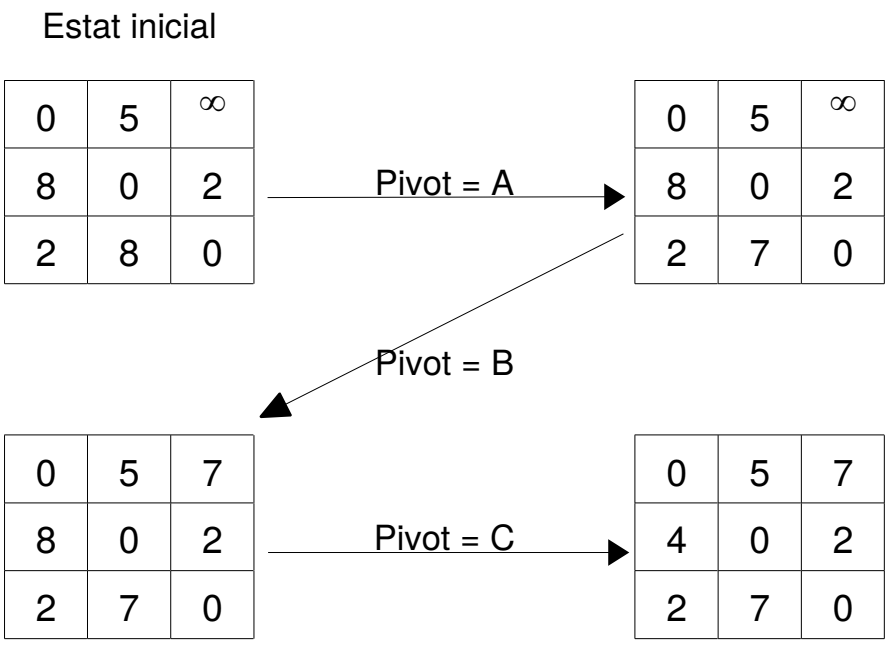
L'algorisme de Floyd funciona de la següent manera:

- Utilitza una matriu quadrada D indexada per parells de vèrtexs per guardar la distància mínima entre cada parell de vèrtexs.
- En el bucle principal tracta un vèrtex, anomenat pivot, cada vegada.
- Quan u és el pivot, es compleix que D(v,w) és la longitud del camí més curt entre v i w format íntegrament per pivots de passos anteriors.
- L'actualització de D consisteix a comprovar, per a tot parell de nodes v i w, si la distància entre ells es pot reduir tot passant pel pivot u mitjançant el càlcul de la fórmula:

$$D[v][w] = \min(D[v][w], D[v][u]+D[u][w])$$

Exemple:





8.8.3.2. Implementació

Mètode privat.
Crea una matriu de n x n.

```
// Cost: Θ(k.length() * log(#símbols))
template <typename V, typename E>
E** graf<V, E>::crear_matriu(int n) {
  E** m = new (E*)[n];
  for (nat i = 0; i < n; ++i) {
    m[i] = new E[n];
  }
  return m;
}
```

```
template <typename V, typename E>
E** graf<V, E>::floyd() {
  E** D = crear_matriu(_verts.size());
  for (vertexs::iterador v = _verts.begin();
         v != _verts.end(); ++v) {
    for (vertexs::iterador w = _verts.begin();
           w != _verts.end(); ++w) {
      D[*v][*w] = _M[*v][*w];
    }
    D[*v][*v] = 0;
  }

  for (vertexs::iterador u = _verts.begin();
         u != _verts.end(); ++u) {
    for (vertexs::iterador v = _verts.begin();
           v != _verts.end(); ++v) {
      for (vertexs::iterador w = _verts.begin();
             w != _verts.end(); ++w) {
        if (D[*v][*w] > D[*v][*u] + D[*u][*w]) {
          D[*v][*w] = D[*v][*u] + D[*u][*w];
        }
      }
    }
  }
  return D;
}
```

8.8.3.3. Diferències entre aquest algorisme i el de Dijkstra

- Els vèrtexs es tracten en un ordre que no té res a veure amb les distàncies.
- No es pot assegurar que cap distància mínima sigui definitiva fins que acaba l'algorisme.

8.8.3.4. Cost de l'algorisme

L'eficiència temporal és $\Theta(n^3)$. L'aplicació reiterada de Dijkstra té el mateix cost. La constant multiplicativa de l'algorisme de Floyd és menor que la de Dijkstra degut a la simplicitat de cada iteració.