UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Politècnica Superior d'Enginyeria
de Vilanova i la Geltrú

PUBLICACIÓ DOCENT

Apunts de teoria d'ESIN

AUTOR: Bernardino Casas, Jordi Esteve
ASSIGNATURA: Estructura de la Informació (ESIN)
CURS: Q3
TITULACIONS: Grau en Informàtica
DEPARTAMENT: Ciències de la Computació
Curs: 2023/2024

Vilanova i la Geltrú, 29 de setembre de 2023

Índex

1

Programació orientada a objectes

Quan fas servir C++ és més difícil que et disparis a tu mateix en el peu, però quan ho fas, et voles la cama sencera.

Bjarne Stroustrup (1950-)

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 se 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. Se 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 1: Tipus Abstracte de Dades

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, …).

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.

Definició 2: Objecte

Un **objecte** és un grapat de *funcions* i *procediments*, tots relacionats habitualment 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**: 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 mes 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.

Les especificacions s'emmagatzemen 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.

Tipus operacions

- 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"// cerca el fitxer nomf en el directori actual.
#include <nomf> // cerca 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 {
  ...
};
```

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'estructur** 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;
};
```

2

Eficiència temporal i espacial

La intel·ligència es basa en l'eficiència d'una espècie a l'hora de fer les coses que necessita per sobreviure.

Charles Darwin (1809-1882)

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:

1. **Determinar** que entenem per **grandària de dades**.
2. **Descomposar** l'algorisme en unitats fonamentals i **calcular** l'eficiènc 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 notacions asimptòtiques per a classificar funcions en base a la seva velocitat de creixement respecte a la grandària de les dades (l'ordre de magnitud del creixement).

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

Definició 3: Notació O gran

$O(f)$: Denota el conjunt de les funcions g que creixen menys ràpidament o igual a f. La seva definició formal és:

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

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:

$$f(n) = n^2 \quad g(n) = n \quad \Rightarrow \quad g \in O(f), \quad f \notin O(g)$$

$$f(n) = 3n^2 \quad g(n) = 100n^2 \quad \Rightarrow \quad g \in O(f), \quad f \in O(g)$$

Definició 4: Notació Ω gran (Omega gran)

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

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

n_o: Indica a partir de quin punt f és una cota inferior per a g.
c_o: Formalitza l'expressió "mòdul constant multiplicatiu".

Exemples:

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

Definició 5: 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:

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

o també es pot definir com:

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

Exemples:

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

La notació Θ gran és la notació que més utilitzarem.

2.2.2 Propietats

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

1. **Reflexivitat**:

   $f \in \Theta(f)$

2. **Transitivitat**:

   $Si \quad h \in \Theta(g) \quad i \quad g \in \Theta(f) \quad \Rightarrow \quad h \in \Theta(f)$

3. **Regla de la suma**:

   $Si \quad g1 \in \Theta(f1) \quad i \quad g2 \in \Theta(f2) \quad \Rightarrow \quad g1+g2 \in \Theta(\max(f1, f2))$

   o el que és el mateix:

   $\Theta(f1) + \Theta(f2) = \Theta(\max(f1, f2))$

4. **Regla del producte**:

   $Si \quad g1 \in \Theta(f1) \quad i \quad g2 \in \Theta(f2) \quad \Rightarrow \quad g1 \cdot g2 \in \Theta(f1 \cdot f2)$

   o el que és el mateix:

   $\Theta(f1) \cdot \Theta(f2) = \Theta(f1 \cdot f2)$

5. **Invariança aditiva**:

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

6. **Invariança multiplicativa**:

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

7. **Simetria**:

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

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 eficient 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**.

Taules 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:

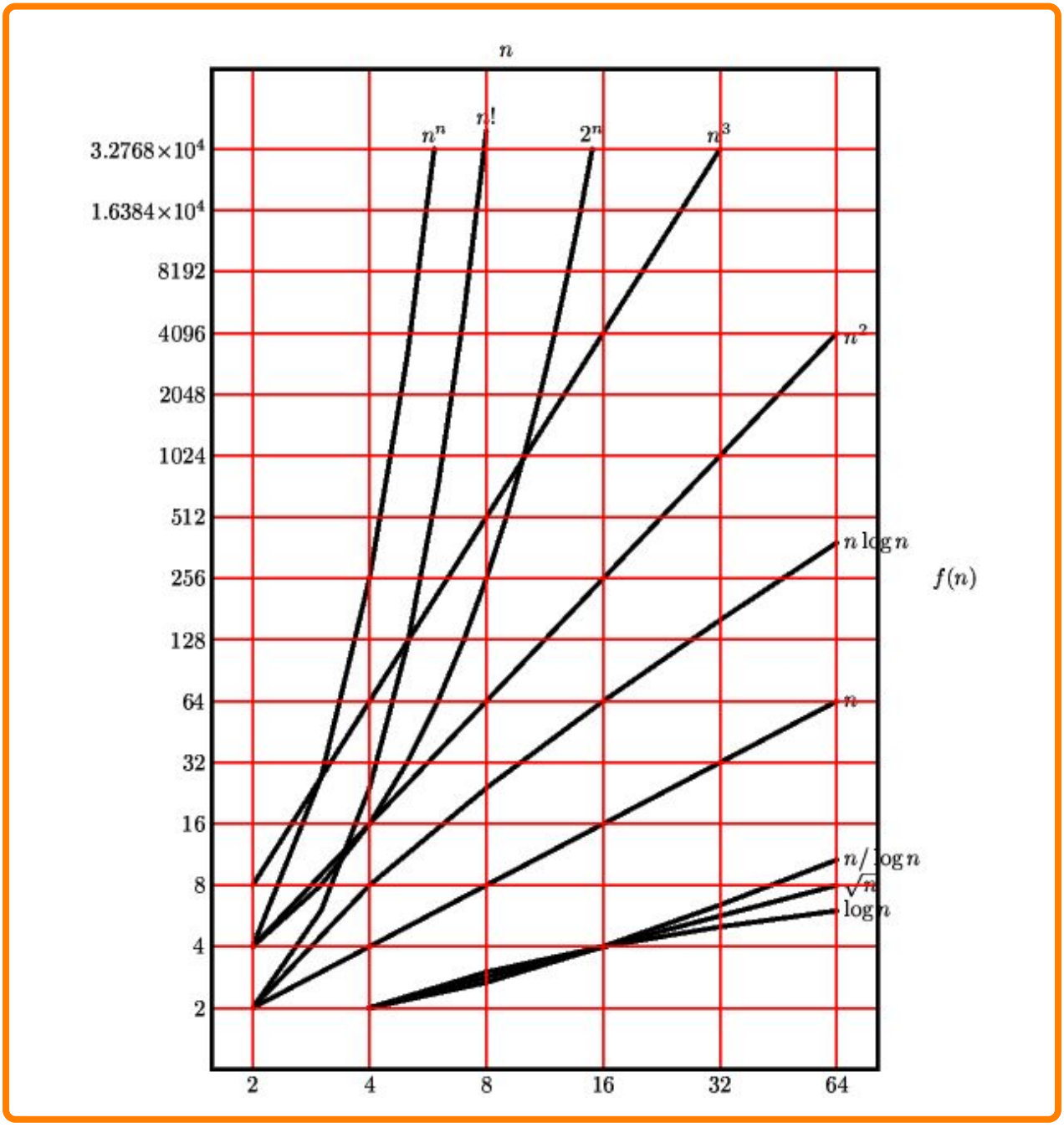


Figura 2.1: Formes de creixement freqüents

2.3 Anàlisi asimptòtica de l'eficiència temporal

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:

   $T(seqüència) =$
   $\max(cost(instrucció_1), cost(instrucció_2), \ldots, cost(instrucció_n))$

3. **Estructura alternativa**:

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

   $T(alternativa) = \max(cost(cond), cost(i_1), cost(i_2))$

4. **Estructura repetitiva**:

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

   $T(repetitiva) = \Sigma_{n^\circ\ iteracions}(cost(cond) + cost(cos)) =$
   $n^\circ\ iteracions * \max(cost(cond), cost(cos))$

Resum de costos

- Operacions bàsiques: $\Theta(1)$
- Estructura seqüencial:

  $\max(\Theta(instrucció_1), ..., \Theta(instrucció_n))$
- Estructura alternativa:

  $\max(\Theta(cond), \Theta(cost\ if), \Theta(cost\ else))$
- Estructura repetitiva:

  $n^\circ\ iteracions * \max(\Theta(cost\ cond), \Theta(cost\ cos))$

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]) {
        int aux = A[j-1];
        A[j-1] = A[j];
        A[j] = aux;
      }
    }
  }
}
```

- El cos del `if` 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:

$$\begin{aligned}\Sigma_{i=1}^{n-1}(n-i) &= 1+2+3+\cdots+n-1 \\ &= \tfrac{n(n+1)}{2} - n = \tfrac{n(n-1)}{2} = \Theta(n^2)\end{aligned}$$

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:

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

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 i $c \geq 1$.

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

- 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, tal com hem fet anteriorment amb el càlcul del factorial.

2.3.1.1 Decreixement aritmètic

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

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

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

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

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

$$\begin{aligned} &T(1) = 1 \\ &T(n) = T(n-1) + n \quad per\ a\ n \geq 2 \end{aligned}$$

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

2.3.1.2 Decreixement geomètric

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

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

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

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

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

$$\begin{aligned} &T(1) = \Theta(1) \\ &T(n) = 2\,T(n/2) + n \quad per\ a\ n \geq 2 \end{aligned}$$

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

2.4 Anàlisi asimptòtica de l'eficiència espacial

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

- Espai que ocupa un **objecte** (de tipus predefinit o bé escalar):

  Constant.

  $E(tipus) = \Theta(1)$

- Espai que ocupa una **taula** (vector):

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

  $E(tipus[N]) = N \cdot E(tipus)$

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

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

  $E(tupla\{tipus_1 c_1; \dots; tipus_n c_n\}) = \max(E(tipus_1), \dots, E(tipus_n))$

3

Estructures lineals estàtiques

Qui salva una vida, salva el món sencer. La llista de Schindler

Thomas Keneally, (1935-)

3.1 Concepte de seqüència

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

Definició 6: Seqüència

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)
- $\forall v \in V,\ s \in V* \ \Rightarrow\ vs \in V*$

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.

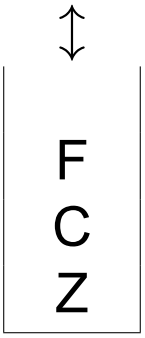
Per a cadascuna d'aquestes estructures veurem:

- 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** (*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 a accions o funcions.

```
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 de les accions 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, …

3.3 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 (*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.

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.

3.3.1 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;
};
```

3.4 Especificació de les cues

3.4.1 Concepte de Cua

La diferència entre les piles i les **cues** (*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.

3.4.2 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.
- 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'esdeveniment …
- Buffers de disc, de teclat, …

3.4.3 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** (*templates*) tal com hem vist al tema 1.

3.4.4 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;
};
```

3.5 Implementació de piles i cues

3.5.1 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.

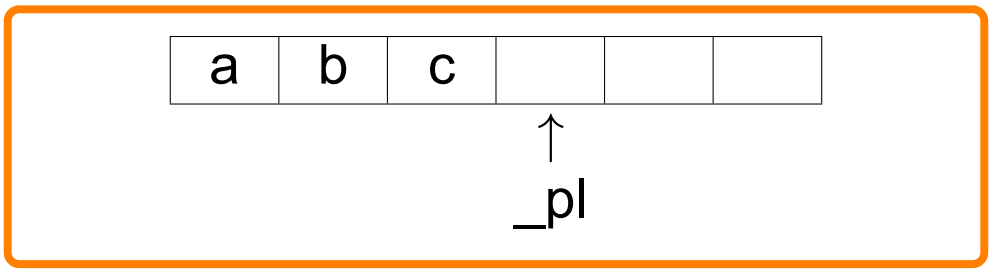


Figura 3.1: Pila estàtica

- 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.

3.5.2 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);
};
```

3.5.3 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 amb extensió `.cpp` sinó que per conveni es posa en un altre fitxer amb extensió `.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). No es fa res ja que no s'usa memòria dinàmica.
template <typename T>
pila<T>::~pila() throw() { }
```

```
// 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;
}
```

3.5.4 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 $\Theta(n)$.

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 $\Theta(1)$:

```
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 in-

advertidament algun error.

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.

3.5.5 Representació de la classe cua

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`

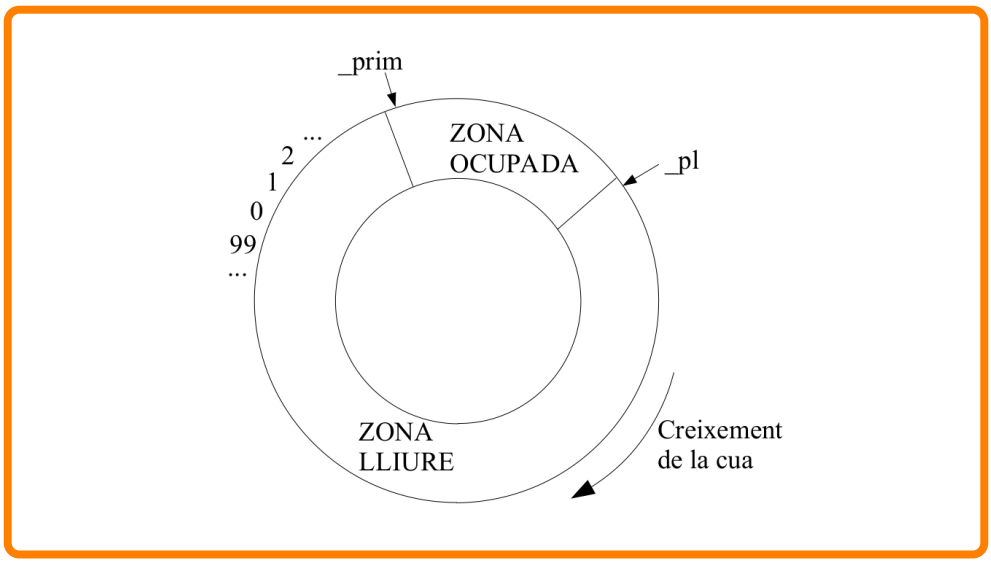


Figura 3.2: Cua estàtica circular

- 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.

```
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);
};
```

3.5.6 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:

- 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:

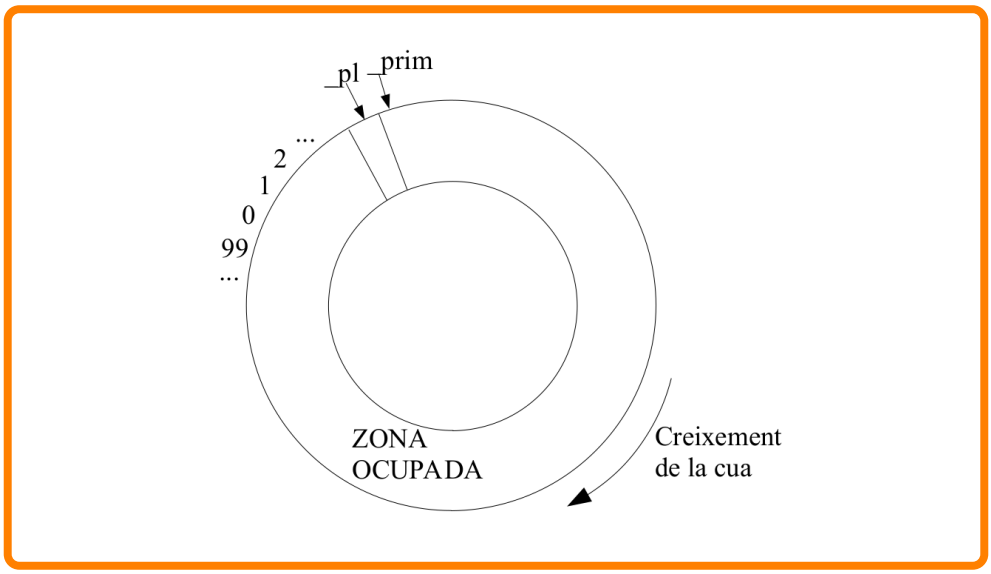


Figura 3.3: Cua estàtica circular plena

- 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.

3.6 Llistes amb punt d'interès

3.6.1 Concepte de llista

Definició 7: Llista

Les **llistes** (*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).

3.6.2 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 (**punt d'interès**) 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.

3.6.3 Alguns algorismes sobre llistes amb punt d'interès

- Recorregut de la llista tractant tots els seus elements:

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

- 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.
- **Representació encadenada**: S'introdueix el concepte d'encadenam Tot element del vector identifica explícitament la posició que ocupa el seu successor a la llista.

3.6.4 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.

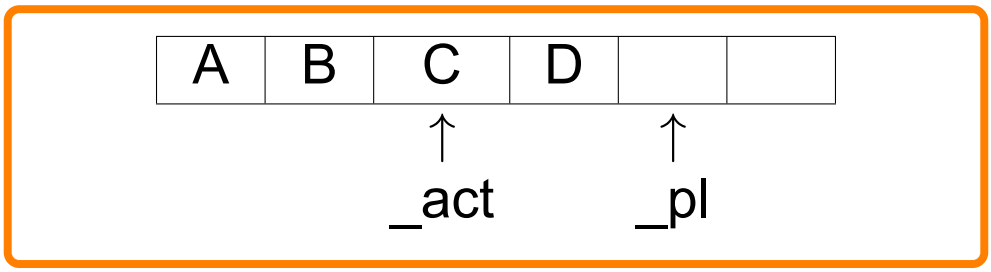


Figura 3.4: Llista estàtica seqüencial

3.6.4.1 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);
};
```

3.6.4.2 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.

3.6.5 Representació encadenada (linked)

- Els elements consecutius de la llista ja no ocuparan posicions consecutives dins del vector.
- **Encadenament** (*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**:
  - Marcar les posicions del vector com a ocupades o lliures.
  - 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.

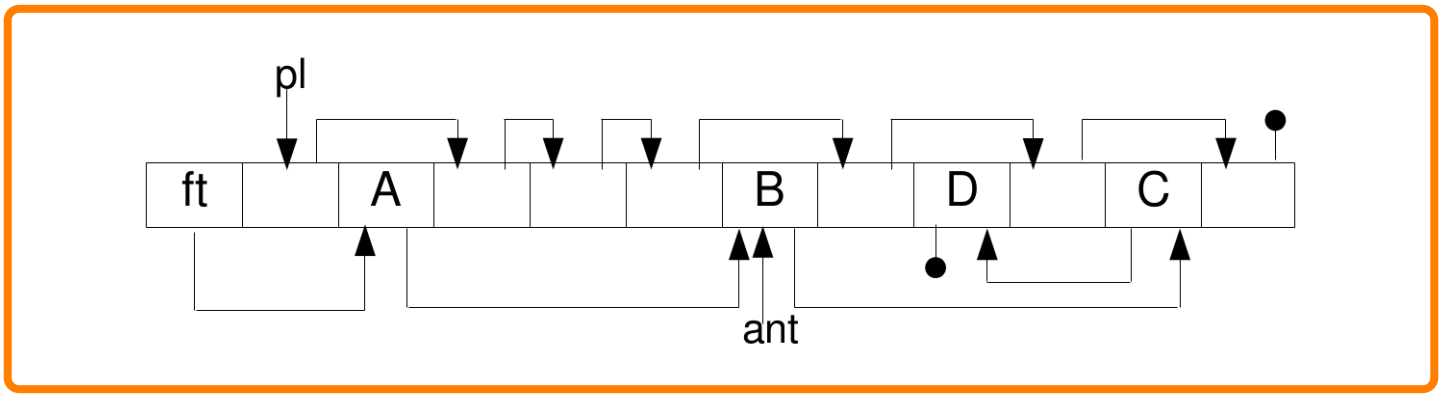


Figura 3.5: Llista estàtica encadenada

4

Estructures lineals dinàmiques

La humilitat, el perdó, la claredat i l'amor són les dinàmiques de la llibertat. Són els fonaments del poder autèntic.

Gary Zukav (1942-)

4.1 Problemàtica a resoldre

La implementació d'estructures amb **memòria estàtica** (arrays o vectors estàtics) a on es reserva la memòria necessària durant la **programació/compilació** dels programes 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.

Per evitar aquests problemes es fa necessari treballar amb **memòria dinàmica** per tal d'obtenir i alliberar memòria durant l'**execució** dels programes.

4.2 Implementació amb punters

La majoria de llenguatges comercials disposen d'un tipus de dades anomenat **punter** o apuntador (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 de **NULL** o **nullptr** 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:

```
struct node {
  int info;  // Guarda la informació útil, per ex. un enter
  node* seg; // Podem enllaçar varis nodes amb el camp seg
};
node *p, *q;       // p i q són punters del tipus node.
... // Cal demanar memòria per objectes de tipus node apuntats per p i q
p->info = v;       // equival a fer (*p).info = v
p->seg = q->seg;   // assignació de punters que apunten a node
```

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).

Exemple:

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

node *p, *q;    // p i q són punters del tipus node
p = new(node); // p apunta a un objecte de tipus node
                // o nullptr si no queda memòria
q = new(node);
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 objecte apuntat per q. Ara q=nullptr
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
```

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

- `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.

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 = nullptr;
cin >> n;
a = new int[n]; // Reserva n enters a la taula a.
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 a.
```

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:

```
node *p, *q;
fstream f; // aquest fitxer emmagatzema elements
           // de tipus node.
...
/* 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 node que és una adreça de
  memòria. No tindrà sentit llegir posteriorment
  les dades del fitxer.*/
f << *p;
```

- **Referències penjades** (*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 nullptr però no apunta a cap objecte vàlid
```

- **Retalls** (*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 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 **nullptr**. La pila consistirà en un punter al node del cim de la pila (**nullptr** si la pila està buida).

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

node *_cim;
```

C Q A D F ⟵ apila ⟶ desaplila, cim

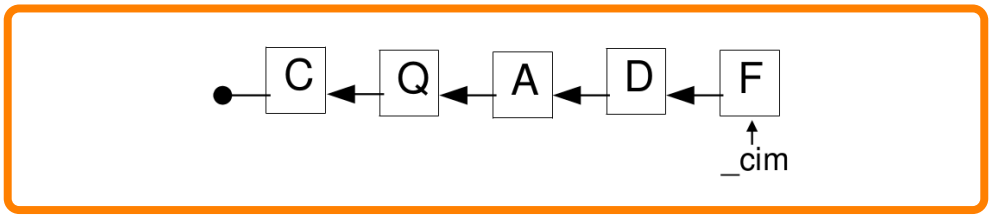


Figura 4.1: Pila implementada amb punters

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 (**nullptr** si la cua està buida).

front, desencua ⟵ C Q A D F ⟵ encua

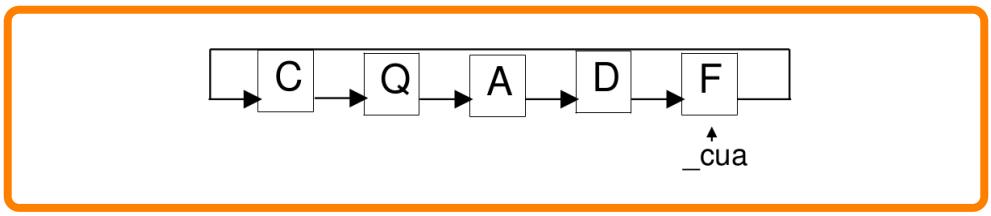


Figura 4.2: Cua implementada amb punters

4.3.1 Representació de la classe cua

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

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

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

4.3.2 Implementació de la classe cua

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

// Cost: Θ(n)
template <typename T>
cua<T>::cua(const cua<T> &c) : _cua(nullptr) throw(error) {
  if (c._cua != nullptr) {
    _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 != nullptr) {
    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 == nullptr) {
    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==nullptr) {
    throw error(CuaBuida);
  }
  node* p = _cua->seg;
  if (p == _cua) {
    _cua = nullptr; // 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==nullptr) {
    throw error(CuaBuida);
  }
  return (_cua->seg->info);
}

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

Totes les operacions de les piles i cues (exceptuant les tres grans) es poden implementar amb cost $\Theta(1)$. El cost espacial és proporcional al número d'elements de la pila/cua: $\Theta(n)$. 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 llista_pi

```
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

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

4.4.2 Implementació de la classe llista_pi

```
// Cost: Θ(1)
template <typename T>
llista_pi<T>::llista_pi() throw(error) {
  _head = new node;
  _head -> next = nullptr;
  _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 = nullptr;
  if (orig != nullptr) {
    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;
}

// 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 != nullptr) {
    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 != nullptr) {
    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 != nullptr) {
    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 == nullptr;
  // 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 != nullptr) {
    _antpi = _antpi->next;
  }
}

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

```
  return _antpi->next->info;
}

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

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.

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 8 bytes).

4.5 Variants de la implementació de llistes

4.5.1 Llistes circulars

En les llistes circulars (*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

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 ele-

ment, per saber quin és l'anterior cal recórrer la llista des del començament i tindria un cost lineal.

- Cal modificar la representació per obtenir llistes doblement encadenades (double-linked list).

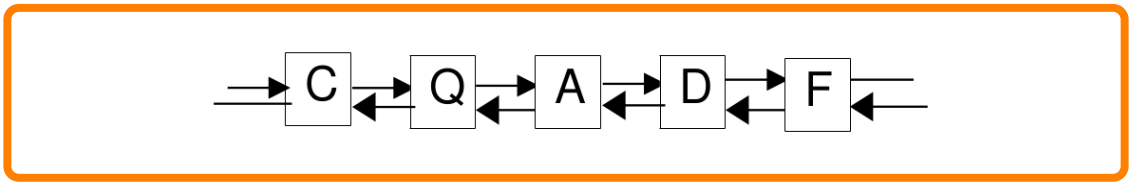


Figura 4.3: Llista doblement encadenada

- 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 llista_itr

```
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

  static node* copiar_llista(node* orig, node* orighead,
        node* h) throw(error);
  static 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 llista_itr

```
// 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'obli
a crear els iteradors mitjançant els mètodes primer, final i indef de la
classe.
template <typename T>
llista_itr<T>::iterador::iterador() throw() {
}

// 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;
}
```

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.

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 (*mergesort*) 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 sublistes: L i L2 de longituds similars.
2. Ordena cadascuna de les dues sublistes: fa dues crides recursives a `mergeSort()`, una amb la sublista L i l'altra amb la sublista L2.
3. Finalment fusiona les sublistes 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 sublistes 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 = nullptr;
}
```

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 punters a nodes.

```
template <typename T>
void llista<T>::fusionar(node* &n1, node* &n2) throw() {
  if (n1 == nullptr) {
    n1 = n2;
    n2 = nullptr;
  }
  if (n1 != nullptr and n2 != nullptr) {
    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 $\Theta(n)$ (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 $\Theta(n)$ (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

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

Obtindrem el mateix resultat si resolem l'equació de recurrència, doncs MergeSort fa dues crides a si mateixa amb llistes amb la meitat d'elements i les accions `partir()` i `fusionar()` tenen cost lineal:

$$T_{MergeSort}(n) = 2 \cdot T_{MergeSort}(n/2) + \Theta(n)$$

5

Arbres

Si no coneixes la història, no saps res. Ets una fulla que no sap que forma part d'un arbre.

Michael Crichton (1942-2008)

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 8: Arbre general

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.

Definició 9: Arbre general

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 $T_1, \ldots, T_k$ de grandàries $n_1, \ldots, n_k$, connectats a l'arrel i que compleixen:

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

5.1.2 Altres definicions

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

- $r_1, \ldots, r_k$ són els **fills** de r.
- r és el **pare** de $r_1, \ldots, r_k$.
- Tot node de T, excepte l'arrel, té un pare.
- $r_1, \ldots, r_k$ 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:

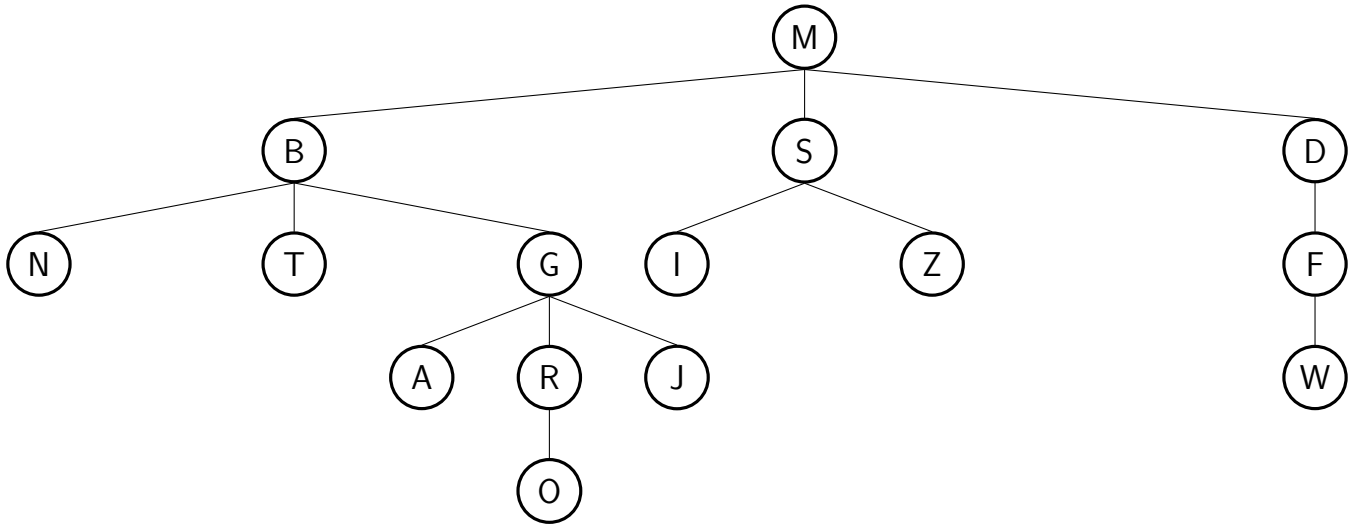


Figura 5.1: Exemple d'arbre.

- 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 del primer fill. 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();
```

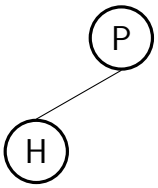
```
Retorna un iterador no vàlid.
```

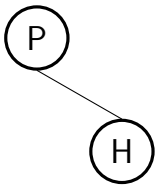
```
  iterador final() const throw();

  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.

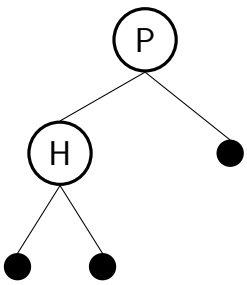
En un arbre binari es diferencia els dos possibles fills que pot tenir cada node. Per exemple, els següents arbres són dos arbres binaris diferents (en canvi, des de la perspectiva d'arbres generals són el mateix, doncs són arbres amb un node que només té un únic fill).

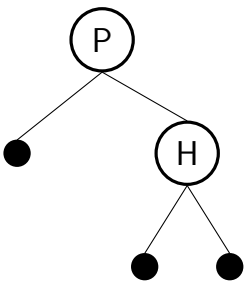




Normalment a les fulles d'un arbre binari no es guarda informació. Per això en la signatura s'ha incorporat una operació per tal de crear l'arbre binari buit.

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 10: Arbre m-ari

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
  // Arb_m_ari
  ...

};
```

5.4 Usos dels arbres

Els arbres tenen molts usos. Destaquem els següents:

- Representació d'expressions i arbres sintàctics (*parse trees*)
- Sistemes de fitxers (jerarquia de subdirectoris/carpetes i fitxers)
- Implementació de cues de prioritat (*priority queues*)
- Implementació de particions (*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** (*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:

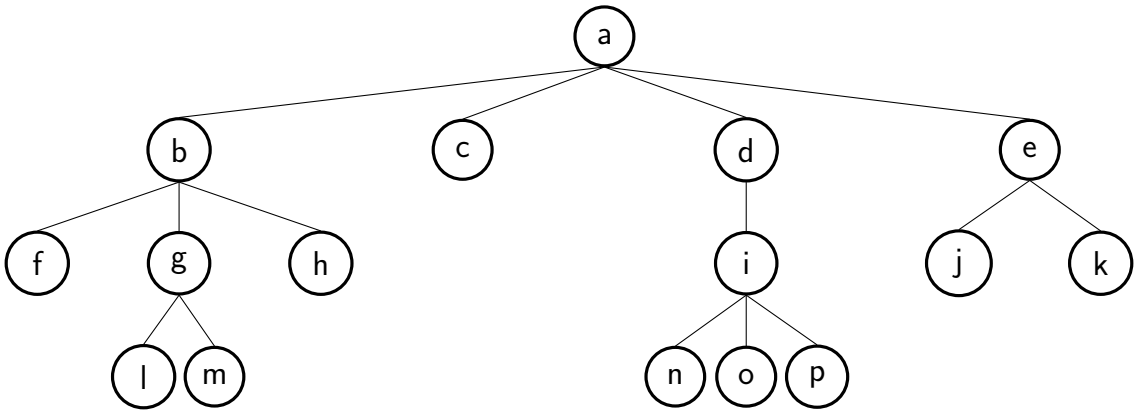


Figura 5.2: Exemple d'arbre general.

- 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:

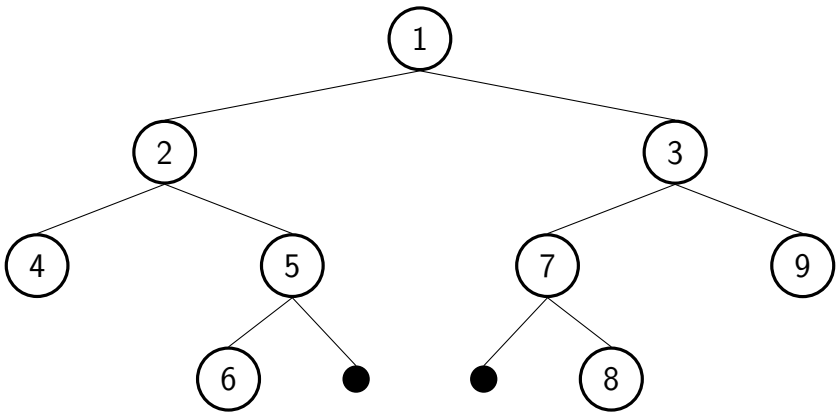


Figura 5.3: Exemple d'arbre binari.

- 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

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.

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

template <typename T>
void rec_preordre (Abin<T>::iterador it,
                   Abin<T>::iterador end,
                   list<T> &lpre) {
```

```
    if (it != end) {
      lpre.push_back(*it);
      rec_preordre(it.fesq(), end, lpre);
      rec_preordre(it.fdret(), end, lpre);
    }
  }
```

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

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.

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

template <typename T>
void rec_postordre (Abin<T>::iterador it,
```

```
                    Abin<T>::iterador end,
                    list<T> &lpost) {
  if (it != end) {
    rec_postordre(it.fesq(), end, lpost);
    rec_postordre(it.fdret(), end, lpost);
    lpost.push_back(*it);
  }
}
```

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 al 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 un iterador a l'arrel de l'arbre inicial.

El procés a seguir és simple: s'extreu un iterador de la cua, es visita l'arrel apuntada per l'iterador, es col·loquen tots els iteradors dels 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);
  }
}
```

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.

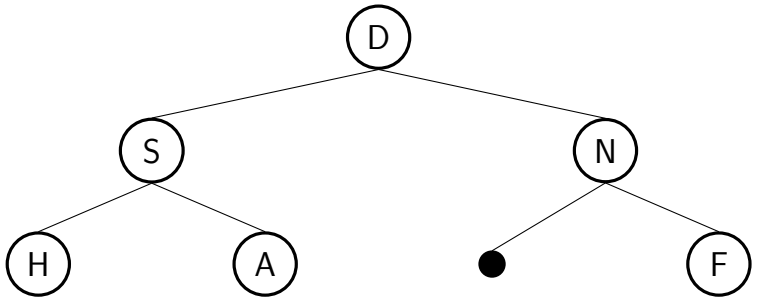
```
template <typename T>
void rec_inordre (const Abin<T> &a, list<T> &lin) {
  stack<Abin<T>::iterador> s;

  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.

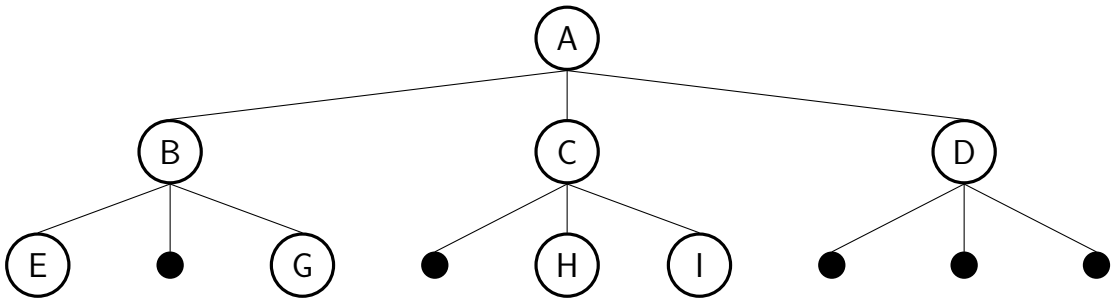


D	S	N	H	A		F			
1	2	3	4	5	6	7	8	9	10

A partir d'un 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)



A	B	C	D	E		G		H	I	
1	2	3	4	5	6	7	8	9	10	11

A partir d'un 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(nullptr) throw(error) {
}
```

```
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 = nullptr;
  _arrel -> f_dret = ad._arrel;
  ad._arrel = nullptr;
}

// 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 == nullptr);
}


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

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

```
// Cost: Θ(n)
template <typename T>
Abin<T> Abin<T>::iterador::arbre() const throw(error) {
  if (_p == nullptr)
  throw error(IteradorInvalid);
  Abin<T> a;
  a._arrel = _p;
  Abin<T> aux(a); // fem la copia
  a._arrel = nullptr;
  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 nullptr, ho canviem per un punter al node predecessor en inordre.

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

Exemple:

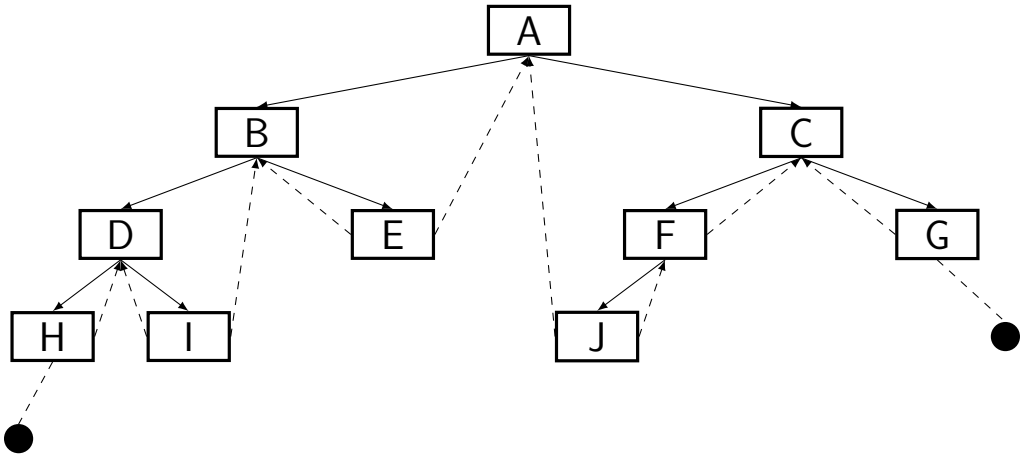


Figura 5.4: Representació arbre enfilat.

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;
  node* _primer_inordre; // Guardar els punters al primer i darrer
  node* _ultim_inordre;  // node en inordre permet enfilar-ho fàcilment
};
```

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 nullptr).
  - El punter esquerra del primer en inordre del subarbre dret apuntarà a la nova arrel (abans apuntava a nullptr).

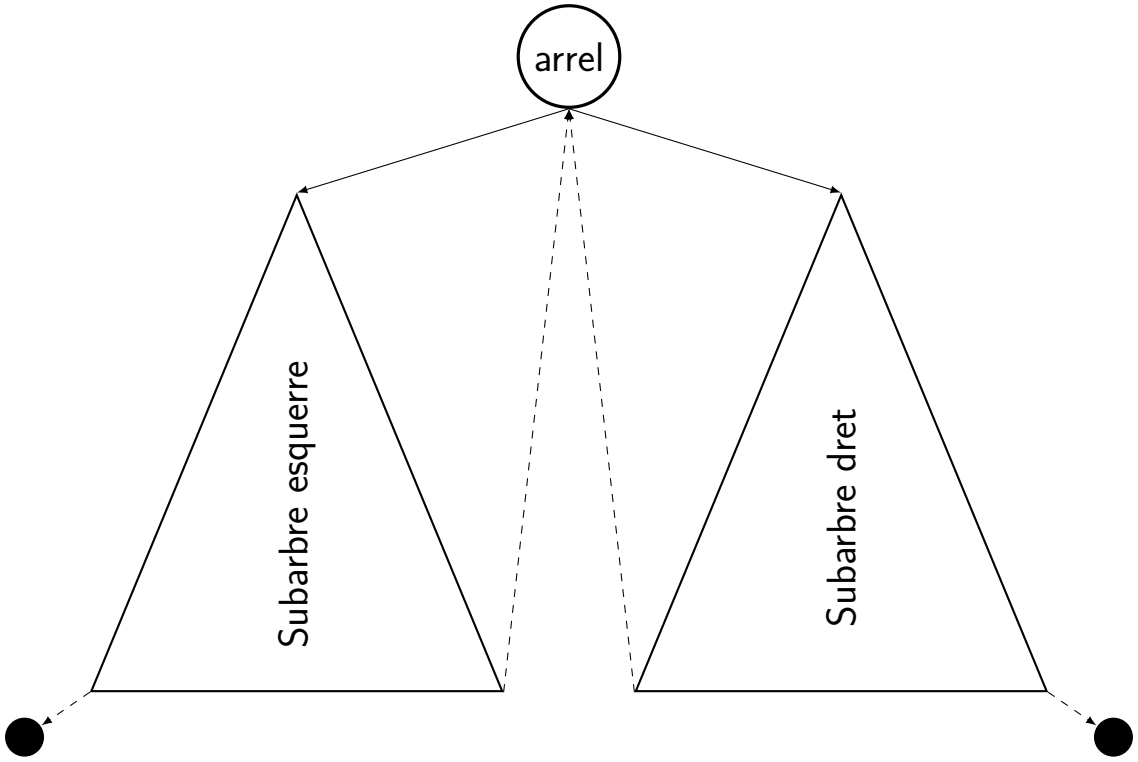


Figura 5.5: Enfilant un arbre binari.

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.

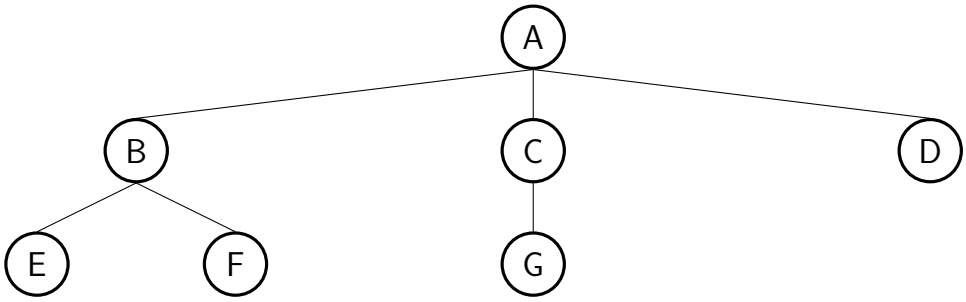
- En cas que el nombre de fills màxim és desconegués a priori es podria caldria fer servir taules dinàmiques, per exemple usant la classe `vector` de la llibreria STL.

5.7.2 Implementació amb punters

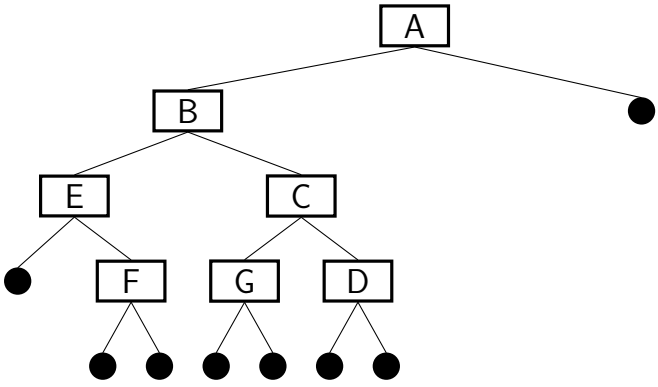
Si el grau màxim no està acotat, és gran o hi ha molta variació possible, llavors convindrà representar la llista de fills amb una llista enllaçada.

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 nullptr 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;
  static node* copia_arbre(node* p) throw(error);
  static 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(nullptr) {
}
```

Construcció d'un arbre que conté un sol element x a l'arrel.

```
// Cost: Θ(1)
template <typename T>
Arbre<T>::Arbre(const T &x) throw(error) {
  _arrel = new node;
  try {
    _arrel -> info = x;
    _arrel -> seggerm = nullptr;
    _arrel -> primf = nullptr;
  }
  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 = nullptr;
  if (p != nullptr) {
    aux = new node;
    try {
      aux -> info = p -> info;
      aux -> primf = aux -> seggerm = nullptr;
      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 != nullptr) {
    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 == nullptr or a._arrel == nullptr or
  a._arrel -> seggerm != nullptr) {
    throw error(ArbreInvalid);
  }
  a._arrel -> seggerm = _arrel -> primf;
  _arrel -> primf = a._arrel;
  a._arrel = nullptr;
}

// 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(nullptr) throw() {
}

// Cost: Θ(1)
template <typename T>
T Arbre<T>::iterador::operator*() const throw(error) {
  if (_p == nullptr) {
    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 == nullptr) {
    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 == nullptr) {
    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 == nullptr) {
    throw error(IteradorInvalid);
  }
  Arbre<T> a;
  a._arrel = _p;
  Arbre<T> aux(a);
  a._arrel = nullptr;
  return aux;
}
```

Índex alfabètic