## Programació 2 Facultat d'Informàtica d'Informàtica, UPC

#### Professorat de PRO2

#### Primavera 2020

- Colaboracions (en ordre alfabètic): Juan Luis Esteban, Ricard Gavaldà, Conrado Martínez, Fernando Orejas
- Aquestes transparències no substitueixen els apunts de l'assignatura, els complementen
- #QuédateEnCasa



## Temari

Part 1: Disseny Modular

Part 2: Estructures Lineals

Part 3: Arbres

Part 4: Disseny Iteratiu: Verificació i Derivació

Part 5: Disseny Recursiu

Part 6: Millores de Eficiència

Part 7: Tipus Recursius de Dades

## Part I

## Disseny Modular

- Abstracció i disseny modular
- 2 Scomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- 5 Jerarquies de Classes i Mòduls
- 6 Implementació de classes



8) Biblioteques i Genericita

#### Motivació

- Com abordar projectes grans
- Quins ajuts ens pot donar el llenguatge de programació
- I quina disciplina hem de seguir els programadors

## Com abordar programes grans

Descomposició en *mòduls*. Clàssica en enginyeria

#### Facilita

- raonar sobre correctesa, eficiència, etc. per parts
- fer programes llegibles, reusables, mantenibles, etc.
- treballar en equip

## Què és una bona descomposició modular?

- Independència: canvis en un mòdul no han d'obligar a modificar altres mòduls.
- Coherència interna: els mòduls tenen significat per si mateixos. Interactuen amb altres mòduls de manera simple i ben definida

#### **Abstracció**

Eina de raonament en programes grans:

Oblidar, temporalment, alguns detalls del problema per tal de transformar-lo en un o bé més simple o bé més general

## Especificació Pre/Post

```
/* Pre: a>0 i b\geq 0 */
int pot(int a, int b);
/* Post: el resultat és a multiplicat per ell mateix
b vegades */
```

## Especificació vs. implementació

- Regla: Un canvi en la implementació d'una funció que respecti la Pre/Post no pot mai fer que un programa que la usa deixi de funcionar
- Especificació = Contracte entre usuari i implementador
- Especificació = Abstracció de l'implementació

## Part I

## Disseny Modular

- 1) Abs acció i disseny modular
- 2 Descomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- 5 Jerarquies de Classes i Mòduls
- 6 Implementació de classes



8) Biblioteques i Genericita

## Tipus de mòduls

- Mòdul funcional: conté un conjunt d'operacions noves necessàries per resoldre algun problema o subproblema
- Mòdul de dades: conté la definició d'un nou tipus i les seves operacions; és habitual a Programació 2

#### Com els fem "abstractes"?

- Mòdul funcional: només deixem veure les especificacions de les operacions
- Mòdul de dades: només deixem veure les capçaleres de les operacions del tipus i una explicació de com es comporten

## Abstracció per dades: tipus predefinits

#### int:

- Valors enters MININT .. MAXINT
- Operacions +, \*, %, /, <, >, ==, ...
- a+b == b+a; a\*b == b\*a, a\*(b+c)==a\*b+a\*c, etc. (si no hi ha overflow)
- a+0 == a, a\*1 == a, a == a, a < a+1, etc.</li>
- ...

Que s'implementin en base 2 com a vectors de bits és irrellevant per a la majoria de problemes de Programació 1 i Programació 2

## Abstracció per dades: nous tipus

#### Solució insatisfactòria - pro1

No hi ha res amagat. No hi ha contracte Si decidim representar els complexos com a mòdul + angle (forma polar), cal canviar totes les accions/funcions que usen el tipus

## Tipus Abstracte de Dades (TADs)

Definim un tipus no per com està implementat, sinó per quines operacions podem fer amb les variables del tipus

#### Un tipus es defineix donant:

- El nom del tipus
- Operacions per construir, modificar i consultar elements del tipus
- Descripció de qué fan les operacions (no com)
- Un tipus de dades pot tenir diverses implementacions. El tipus és la seva especificació, no les seves implementacions

## TADs i independència entre mòduls

- Fase d'especificació: Decidir operacions del TAD i contractes d'ús
- Fase d'implementació: Decidir una representació i codificar les operacions

#### Conseqüència:

Un canvi en la implementació d'un TAD que no afecti l'especificació de les seves operacions no pot mai fer que un programa que usa el TAD deixi de funcionar

## Part I

## Disseny Modular

- 1) Abs vacció i disseny modular
- 2 Scomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- 5 Jerarquies de Classes i Mòduls
- 6 Implementació de classes



8 Biblioteque Genericitat

## Orientació a objectes

Una manera de separar especificació d'implementació, d'implementar Tipus Abstractes de Dades A Programació 2 només veurem *una part* de la utilitat d'aquesta manera de pensar

Més en altres assignatures: herència i polimorfisme

## Classes i objectes

- Les variables i constants d'un tipus són objectes
- Una classe és el patró comú al objectes d'un tipus
- A l'inrevés: Donada una classe, podem definir-ne objectes o instàncies
- Cada classe defineix els atributs (= camps) i els mètodes (= operacions) del tipus.
- Cada objecte és propietari dels seus atributs i mètodes
- Els mètodes tenen un paràmetre implícit: el seu propietari o objecte sobre el qual s'aplica el mètode
- Podem fer més accions/funcions que operen amb el tipus, però si no són dins de la classe no són mètodes

## Exemple: La classe Estudiant

#### Un Estudiant es caracteritza per:

- Un DNI, que és un enter no negatiu, obligatori
- Una nota, optativa. Si en té, és un real (double) entre 0 i un cert valor màxim (p.ex., 10). Si no la té, es considera NP

## Exemple d'ús d'Estudiant: canviar un NP per 0

#### Ús de la classe Estudiant

```
/* Pre: tots els elements de v són Estudiants
    amb DNIs diferents */
bool canvia_np_per_zero(vector<Estudiant>& v, int dni);
/* Post: si v conté un Estudiant amb DNI = dni, i aquest
    no té nota, llavors aquest estudiant passa
    a tenir nota 0; la resta de v no canvia;
    el resultat diu si l'estudiant s'ha trobat */
```

## Exemple d'ús d'Estudiant: canviar un NP per 0

```
Ús de la classe Estudiant
bool canvia_np_per_zero(vector<Estudiant>& v, int dni) {
  int i = 0;
  while (i < v.size()) {
    if (v[i].consultar_DNI() == dni) {
      if (not v[i].te_nota())
        v[i].afegir_nota(0);
      return true;
    ++i;
  return false:
```

## Exemple d'ús d'Estudiant: calcular nota mitjana

#### Un altre exemple d'ús

## Exemple d'ús d'Estudiant: calcular nota mitjana

## Un altre exemple d'ús

```
double nota_mitjana(const vector<Estudiant>& v) {
     int n = 0;
     double suma = 0;
     for (int i = 0; i < v.size(); ++i) {</pre>
         if (v[i].te_nota()) {
             ++n;
              suma += v[i].consultar_nota();
     if (n > 0)
        return suma/n;
     else
        return -1;
```

## Paràmetre implícit

#### En C++ sense OO:

# Declaració d'una funció/acció /\* Pre: -- \*/ bool te\_nota(const Estudiant &e); /\* Post: El resultat és cert si i només si e té nota \*/

#### Amb OO:

#### Declaració d'un mètode

Noteu el const referit a l'objecte implícit

## Exemple OO: crida a un mètode

#### Forma general:

```
<nom_de_l'objecte>.<nom_del_mètode>(<altres paràmetres>)
```

#### Crida a una funció

```
bool b = te_nota(est);
```

### Aplicació/invocació d'un mètode

```
b = est.te_nota();
```

## Exemple OO: crida a un mètode modificador

Fixeu-vos: sense const, el paràmetre implícit pot ser modificat pel mètode ⇒ el mètode rep el seu paràmetre implícit per referència

```
Crida
est.modificar_nota(x);
```

## Part I

## Disseny Modular

- 1) Abs vacció i disseny modular
- 2 Scomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- 5 Jerarquies de Classes i Mòduls
- Implementació de classes



8) Biblioteques i Genericita

## Tipus d'operacions: Creadores o Constructores

Creadores = funcions que serveixen per crear objectes nous En C++, hi ha constructores:

- Tenen el mateix nom de la classe i crean un objecte nou d'aquest tipus
- No ténen paràmetre implícit! Crean un object abans no existent!
- La llista de paràmetres permet distingir entre diverses constructores
- Constructora per defecte: sense paràmetres, crea un objecte nou sense informació

## Tipus d'operacions: Constructores

## Exemple 1: Constructora por defecte

```
Estudiant est;
```

#### Exemple 2: Constructora amb un paràmetre de tipus int

```
Estudiant est(46567987);
```

#### Qué passa quan fem les següents declaracions?

```
vector<char> v;
vector<char> w(10);
vector<char> linea(10,'*');
vector< vector<char> > M(10, vector<char>(5,'-'));
```

## Tipus d'operacions: Destructora

# Destructora ~nom\_classe() { ... }

- En C++, una operació destructora d'objectes de la classe
- Fa operacions que puguin fer falta abans que l'objecte desaparegui
- Rarament la cridarem. No en parlarem més fins al Tema 7
- L'operació per defecte no fa res; s'aplica si no n'escrivim cap
- Podem redefinir-la
- Es crida automàticament al final de cada bloc amb les variables declarades en el bloc

## Tipus d'operacions: Destructora

Pregunta: qué passa si declarem vector<Estudiant>
v(10)?

## Tipus d'operacions: *Modificadores*

- Transformen l'objecte propietari (paràmetre implícit), potser amb informació aportada per altres paràmetres
- Normalment en C++ retornen void; són accions
- Seguretat: Tots els canvis es fan via mètodes ben definits

## Tipus d'operacions: Consultores

- Proporcionen informació sobre l'objecte propietari, potser amb ajut d'informació aportada per altres paràmetres
- Normalment porten const perquè no modifiquen el paràmetre implícit
- Normalment funcions, tret que hagin de retornar més d'un resultat; en aquest cas poden ser accions amb més d'un paràmetre de sortida (passat per referència)

## Tipus d'operacions: Consultores

```
Exemple 1: Ús d'un mètode consultor
double x = est.consultar_nota();
```

```
Exemple 2: Ús d'un mètode consultor

if (est.te_nota()) ... else ...
```

Aquest mètode consultor és necessari perquè hi ha operacions que tenen com a precondició que l'estudiant tingui o no tingui nota

### Mètodes de classe

- Són propis de la classe, no de cada objecte
- No tenen paràmetre implícit

### Mètode de classe

#### Crida d'un mètode de classe

```
cin >> nota;
if (nota >= 0 and nota <= Estudiant::nota_maxima())
  e.modificar_nota(nota);
else
  cout << "La nota introduïda no és vàlida" << endl;</pre>
```

## Especificació de classes en C++

```
class Estudiant {
public:
// Constructores
Estudiant();
/* Pre: cert. */
/* Post: el resultat és un estudiant amb DNI = 0
         i sense nota */
Estudiant (int dni):
/* Pre: dni >= 0 */
/* Post: el resultat és un estudiant amb DNT = dni
         i sense nota */
// Destructora: esborra automàticament els objectes
               locals en sortir d'un àmbit de
                visibilitat
~Estudiant():
```

## Especificació de classes en C++

```
// Modificadores
void afegir nota(double nota);
/* Pre: l'estudiant implícit no té nota i
        'nota' és una nota vàlida */
/* Post: la nota de l'estudiant implícit
         passa a ser 'nota' */
void modificar_nota(double nota);
/* Pre: l'estudiant implícit té nota i
        'nota' és una nota vàlida */
/* Post: la nota de l'estudiant implícit
        passa a ser 'nota' */
```

## Especificació de classes en C++

```
// Consultores
int consultar DNI() const;
/* Pre: cert. */
/* Post: retorna el DNI de l'estudiant */
bool te nota() const;
/* Pre: cert */
/* Post: retorna cert si i només si
         l'estudiant té nota */
double consultar nota() const;
/* Pre: l'estudiant té nota */
/* Post: retorna la nota de l'estudiant */
// Mètodes de classe
static double nota maxima();
/* Pre: cert. */
/* Post: retorna la nota màxima que pot
         tenir qualsevol estudiant */
};
```

## Part I

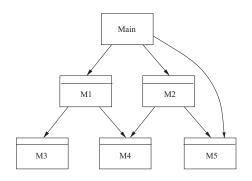
# Disseny Modular

- 1) Abs vacció i disseny modular
- 2 scomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- 5 Jerarquies de Classes i Mòduls
- 6 Implementació de classes



8) Biblioteques i Genericita

## Diagrames modulars



"Jerarquia de classes" en programació OO Diferents tipus d'arcs per a diferents tipus de relacions En aquest curs, només relacions d'ús

## Diagrames modulars

- Mòdul del programa principal: sense arcs entrants
- Resta mòduls: mòduls de dades o funcionals

- Graf acíclic, no necessàriament un arbre
- "Jerarquia de classes" en programació OO

### Relacions entre mòduls

Programa = Conjunt de mòduls relacionats / dependents

Un mòdul pot:

- definir un nou tipus de dades, a partir d'altres
- ampliar/enriquir un tipus amb noves operacions

### Relacions entre mòduls

Programa = Conjunt de mòduls relacionats / dependents

### Un mòdul pot:

- definir un nou tipus de dades, a partir d'altres
- ampliar/enriquir un tipus amb noves operacions

#### Les relacions d'ús poden ser:

- visibles, en especificació
- ocultes, per una implementació concreta

## Exercici: Conjunt d'Estudiants

Volem definir una nova classe Cjt\_estudiants, per gestionar conjunts d'estudiants

Relació d'ús: Dins de Cjt\_estudiants.hh

```
#include "Estudiant.hh"
```

Important: per especificar Cjt\_estudiants no cal saber la implementació de la classe Estudiant

```
#include "Estudiant.hh"
// Un Cjt_estudiant Representa un conjunt d'estudiants,
// ordenat per DNI creixent amb un màxim nombre d'Estudiants
// Es poden consultar i modificar els seus elements
// (Estudiants) per DNI o per posició en l'ordre
// creixent de DNT
class Cjt_estudiants {
public:
// Constructores
/* Pre: cert. */
/* Post: crea un conjunt d'estudiants buit */
Cit estudiants();
// Destructora
~Cit estudiants();
```

```
// Modificadores
/* Pre: el conjunt no conté cap estudiant amb el DNI
        de l'Estudiant 'est'; la mida actual del conjunt
        és menor que la mida màxima permesa */
/* Post: s'ha afegit l'estudiant 'est' al conjunt */
void afegir estudiant (const Estudiant &est);
/* Pre: el conjunt conté un estudiant amb el mateix DNI
        que l'Estudiant 'est' */
/* Post: l'Estudiant 'est' substitueix a l'estudiant del
         conjunt original que tenia el mateix DNI que 'est' */
void modificar estudiant(const Estudiant &est);
```

```
// Consultores
/* Pre: cert. */
/* Post: el resultat és el nombre d'estudiants del conjunt */
int mida() const;
/* Pre: dni > 0 */
/* Post: torna cert si i només si el conjunt conté un Estudiant
         amb DNI iqual al donat (dni) */
bool existeix estudiant (int dni) const;
/* Pre: el conjunt conté un Estudiant amb DNI = dni */
/* Post: el resultat és l'Estudiant amb DNT = dni
         present en el conjunt */
Estudiant consultar estudiant (int dni) const;
```

```
// Lectura i escriptura
/* Pre: cert. */
/* Post: el paràmetre implícit conté el conjunt d'estudiants
         llegits del canal estàndard d'entrada */
void llegir();
/* Pre: cert. */
/* Post: s'han escrit en canal estàndard de sortida els
         estudiants del conjunt en ordre ascendent per DNI */
void escriure() const;
private:
// elements privats de la classe: atributs,
// mètodes privats, ...
};
```

## Part I

# Disseny Modular

- 1) Abs vacció i disseny modular
- 2 Descomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- Jerarquies de Classes i Mòduls
- 6 Implementació de classes



8 Biblioteque Genericitat

## Implementació d'una classe

#### Fases:

- Implementar el tipus: Triar una representació. Definir els atributs amb tipus ja existents
- Implementar les operacions: Codificar les seves operacions en termes d'instruccions
- Pot ser convenient definir mètodes i funcions auxiliars (privades, no visibles des de fora)

### **Fitxers**

Ideal: Separació completa públic i privat

- hh: Capçaleres de les operacions públiques
- .cc: Atributs i codi de totes les operacions

### **Fitxers**

- Ideal: Separació completa públic i privat
  - hh: Capçaleres de les operacions públiques
  - .cc: Atributs i codi de totes les operacions

#### Com es fa:

- .hh ⇒ part private: Atributs i capçaleres de operacions privades
- .hh ⇒ part public: Capçaleres de les operacions públiques
- .cc ⇒ Implementació de les operacions públiques i privades

```
Fitxer Estudiant.hh
class Estudiant {
public:
// Lectura i escriptura
void llegir();
void escriure() const;
private:
int dni;
double nota;
bool amb nota;
static const int MAX NOTA = 10:
/* Invariant de la representacio:
   dni >= 0
   si amb nota llavors 0 <= nota <= MAX NOTA
*/
};
```

- En aquesta implementació de la classe Estudiant no es defineixen mètodes privats
- La constant estàtica MAX\_NOTA és un atribut de classe

## Fitxer Estudiant.cc #include <iostream> #include "Estudiant.hh" using namespace std; Estudiant::Estudiant() /\* Pre: cert \*/ /\* Post: crea un estudiant amb DNI=0 i sense nota \*/ dni = 0; amb nota = false; Estudiant::Estudiant(int d) /\* Pre: dni>=0 \*/ /\* Post: crea un estudiant amb DNI=d i sense nota \*/ dni = d; amb nota = false; Estudiant::~Estudiant() {}

```
void Estudiant::afegir nota(double n)
/* Pre: l'Estudiant no té nota i 0 <= n <= nota_maxima() */</pre>
/* Post: la nota de l'Estudiant passa a ser "nota" */
    nota = n;
    amb nota = true;
void Estudiant::modificar_nota(double n)
/* Pre: l'Estudiant té nota i 0 <= n <= nota maxima() */</pre>
/* Post: la nota de l'Estudiant passa a ser n */
    nota = n;
int Estudiant::consultar_DNI() const
/* Pre: cert */
/* Post: torna el DNT de l'Estudiant */
    return dni;
```

```
double Estudiant::consultar nota() const
/* Pre: l'Estudiant té nota */
/* Post: torna la nota de l'Estudiant */
    return nota:
double Estudiant::nota_maxima() // aquí no es posa "static"
/* Pre: cert. */
/* Post: torna la nota màxima que pot tenir qualsevol Estudiant
    return MAX NOTA;
bool Estudiant::te_nota() const
/* Pre: cert */
/* Post: torna cert si i només si l'Estudiant té nota */
    return amb nota;
```

```
void Estudiant::llegir()
/* Pre: el cin conté un enter no negatiu d i un double n */
/* Post: l'Estudiant passar a tenir el DNI d; si n està
         en el rang [0..nota_maxima()] llavors la nota de
         l'Estudiant passa a ser n; altrament l'Estudiant
         es queda sense nota */
    cin >> dni;
    double x:
    cin >> x;
    if (x >= 0 \text{ and } x <= MAX_NOTA) {
       nota = x:
       amb nota = true;
    } else {
      amb nota = false;
```

## Exercici: implementació alternativa del tipus Estudiant

### Objectiu: demostrar la independència de la implementació

- Eliminem l'atribut booleà
- Si l'atribut nota és -1, l'estudiant no té nota

No canviem l'especificació  $\rightarrow$  no cal revisar classes que l'usen

### **Atributs**

- Variables o constants de tipus previs
- Sempre els declarem a la part private

### **Atributs**

- Variables o constants de tipus previs
- Sempre els declarem a la part private
- const = és una constant (no modificable)

### **Atributs**

- Variables o constants de tipus previs
- Sempre els declarem a la part private
- const = és una constant (no modificable)
- static = és un atribut de classe (compartit per tots els objectes de la classe)
  - S'accedeix amb classe::atribut i no objecte.atribut

## Operacions auxiliars privades

- Útils per a implementar les públiques
- Capçalera a la part private del fitxer .hh: Només poden ser cridades per un mètode públic o privat de la classe
- Tenen accés als atributs dels objectes de la classe
- Poden ser mètodes (s'apliquen sobre l'objecte propietari) o mètodes de classe (static)

# Implementació de mètodes públics i privats

```
Notació: nom_classe::nom_operacio(...)
```

- :: vol dir "No estem implementant una op. nova, sinó la que ja haviem declarat abans"
- Iliguem cada operació amb les seva capçalera al corresponent arxiu .hh
- dóna el dret d'accedir a la part private de la classe
- tant per a ops. públiques com privades
- en la implementació dels mètodes de classe no es posa static

# Accés a un camp/atribut

Forma general: nom\_objecte.nom\_atribut

Exemple: x.c

Si x és un objecte de tipus  $\mathbb{T},$  llavors  $_{\mathbb{C}}$  ha de ser un atribut de  $\mathbb{T}$ 

# Accés a un camp/atribut

#### Casos particulars:

- Quan accedim als atributs de l'objecte implícit en un mètode només s'escriu el nom de l'atribut
  - Ex: dni sols es refereix al camp dni del paràmetre implícit

# Accés a un camp/atribut

#### Casos particulars:

- Quan accedim als atributs de l'objecte implícit en un mètode només s'escriu el nom de l'atribut
  - Ex: dni sols es refereix al camp dni del paràmetre implícit
- En alguns casos molt específics necessitem referir-nos explícitament a l'objecte implícit d'un mètode: this = el paràmetre implícit = l'objecte propietari. Usos:
  - Desambiguar quan en aquell àmbit de visibilitat hi ha una variable amb el mateix nom que un atribut.
  - Pasar el paràmetre implícit com a paràmetre explícit d'una operació
  - En realitat, this és un apuntador a l'objecte implícit;
     l'objecte implícit és \*this

# Exemple: Implementació de Cjt\_estudiants

#### Representació i invariant:

 Un atribut de classe constant MAX\_NEST, que estableix el màxim nombre d'estudiants que pot haver en un conjunt

# Exemple: Implementació de Cjt\_estudiants

#### Representació i invariant:

- Un atribut de classe constant MAX\_NEST, que estableix el màxim nombre d'estudiants que pot haver en un conjunt
- Un atribut enter nest, el nombre d'estudiants en el conjunt, 0 ≤ nest ≤ MAX\_NEST
- Un atribut vest que és un vector de Estudiants, de mida MAX\_NEST i que estarà ordenat per dni en tot moment
  - els mètodes afegir\_estudiant i llegir\_cjt\_estudiants seràn més complexos (i costosos en temps) per a garantir que els continguts del vector vest estàn ordenats
  - afavoreix la cerca (perquè es pot fer dicotòmica)

# Exemple: Implementació de Cjt\_estudiants

#### Operacions privades

- Un mètode privat ordenar\_cjt\_estudiants
- Un mètode de classe (static) privat cerca\_dicot, rep explícitament el vector d'Estudiant sobre el qual es fa la cerca

# Invariant de la representació

- Propietats dels atributs que ens comprometem a mantenir en la implementació de les operacions
- Queda garantit si només es manipula la representació amb ops. constructores i modificadores
- Implícit com a Pre i Post a totes les operacions
- És bona praxis escriure'l junt amb la representació: molt bona documentació!

# Invariant de la representació

#### Classe Estudiant

- dni >= 0
- si (amb\_nota) llavors (0 <= nota <= MAX\_NOTA)

o be (implementacio sense booleà)

- dni >= 0
- (nota == -1) o bé (0 <= nota <= MAX\_NOTA)

# Invariant de la representació

#### Classe Cjt\_estudiants

- 0 <= nest <= vest.size() = MAX\_NEST,</li>
- tots els estudiants de vest[0..nest-1] tenen dnis diferents,
- vest[0..nest-1] està ordenat creixentment pels DNI dels estudiants

#### Cjt\_estudiants.hh

```
class Cjt_estudiants {
private:
    vector<Estudiant> vest;
    int nest;
    static const int MAX_NEST = 20;
    /*
     Invariant de la representacio:
     * 0 <= nest <= vest.size() = MAX NEST,
     * tots els estudiants en vest[0..nest-1] tenen DNI
       diferents. i
     * vest[0..nest-1] esta ordenat creixentment pels DNI
       dels estudiants
    */
```

```
Cit estudiants.hh
class Cit estudiants {
private:
void ordenar cjt estudiants();
/* Pre: cert */
/* Post: els Estudiants del conjunt estan ordenats
         creixentment pels seus DNI */
static int cerca dicot(const vector<Estudiant> &vest,
                       int left, int right, int x);
/* Pre: vest[left..right] està ordenat creixentment
        per DNI, 0 <= left, right < vest.size() */
/* Post: si a vest[left..right] hi ha un element
         amb DNI = x, el resultat és una posicio que
         el conté; si no, el resultat es -1 */
};
```

```
Cjt_estudiants.cc

#include "Cjt_estudiants.hh"
#include <iostream>
using namespace std;

Cjt_estudiants::Cjt_estudiants() {
    nest = 0;
    vest = vector<Estudiant>(MAX_NEST);
}

Cjt_estudiants::~Cjt_estudiants() {}
```

```
void Cjt_estudiants::afegir_estudiant(const Estudiant &est) {
   int dni = est.consultar_DNI();
   int i = nest - 1;
   while (i >= 0 and dni < vest[i].consultar_DNI()) {
      vest[i + 1] = vest[i];
      --i;
   }
   vest[i + 1] = est;
   ++nest;
}</pre>
```

```
Cjt_estudiants.cc
void Cit estudiants::modificar estudiant(const Estudiant &est) {
    // per la Pre, segur que trobem el DNI d'est com a DNI
    // d'algun element de vest[0..nest-1]; apliquem-hi la cerca
   // dicotomica
    int i = cerca dicot(vest, 0, nest-1, est.consultar DNI());
   // i es la posicio amb el DNI d'est
   vest[i] = est;
void Cjt_estudiants::modificar_iessim(int i, const Estudiant &est)
   vest[i-1] = est;
```

```
Cjt_estudiants.cc
int Cjt_estudiants::mida() const {
    return nest;
int Cjt_estudiants::mida_maxima() {
    return MAX_NEST;
bool Cjt_estudiants::existeix_estudiant(int dni) const {
    int i = cerca_dicot(vest, 0, nest-1, dni);
    return (i != -1);
```

```
Cjt_estudiants.cc

Estudiant Cjt_estudiants::consultar_estudiant(int dni) const {
    int i = cerca_dicot(vest, 0, nest-1, dni);
    return vest[i];
}

Estudiant Cjt_estudiants::consultar_iessim(int i) const {
    return vest[i-1];
```

```
void Cjt_estudiants.cc

void Cjt_estudiants::llegir() {
    cin >> nest;
    for (int i = 0; i < nest; ++i) vest[i].llegir();
    ordenar_cjt_estudiants();
    // noteu que l'apliquem sobre el objecte implÃcit
}

void Cjt_estudiants::escriure() const {
    for (int i = 0; i < nest; ++i) vest[i].escriure();
}</pre>
```

```
Cit estudiants.cc
// observem que no hi ha referencia a nest
int Cjt estudiants::cerca dicot(const vector<Estudiant> &vest,
                                 int left, int right, int x) {
    int i: bool found = false:
    while (left <= right and not found) {</pre>
        i = (left + right)/2;
        if (x < vest[i].consultar DNI()) right = i - 1;</pre>
        else if (x > vest[i].consultar DNI()) left = i + 1;
        else found = true;
    // si l'element buscat existeix, i es la posicio que volem
    if (found) return i;
    else return -1;
```

```
// ordena el vector d'estudiants per dni creiexentment,
// usant el metode de seleccio
// Es un exemple. Milloraria usant algorismes d'ordenacio
// mes rapids.
void Cjt_estudiants::ordenar_cjt_estudiants() {
   for (int i = 0; i < nest - 1; ++i) {</pre>
          int min dni = vest[i].consultar DNI();
          int pos min = i;
          for (int j = i+1; j < nest - 1; ++j)
                 if (min dni > vest[j].consultar DNI()) {
                        pos min = i;
                        min dni = vest[j].consultar_DNI();
          Estudiant etemp = vest[i];
          vest[i] = vest[pos_min];
          vest[pos min] = etemp;
```

#### **Observacions**

Algunes especificacions poden incloure *anotacions sobre eficiència*, que restringeixen el ventall d'implementacions vàlides

- Especificació #1 de Cjt\_estudiants,
  - afegir\_estudiant: "tarda temps proporcional a la mida del conjunt"
  - existeix\_estudiant: "tarda temps logarítmic en la mida del conjunt"
- Especificació #2:
  - afegir\_estudiant: "tarda temps constant"
  - existeix\_estudiant: "tarda temps lineal en la mida del conjunt"

# Part I

# Disseny Modular

- 1) Abs vacció i disseny modular
- 2 Procomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- 5 Jerarquies de Classes i Mòduls
- 6 Implementació de classes

- 7 Ampliacions de tipus de dades: mòduls funcionals i llibreries
- 8 Biblioteque Genericitat

# Ampliacions de tipus de dades

#### Ampliar un TAD: afegir noves funcionalitats Mecanismes d'ampliació en OO

- Modificar la classe existent per afegir nous mètodes
- Enriquiment = definir les noves operacions fora de la classe
- Merència amb mecanismes del llenguatge (no a PRO2)

#### Solució 1: Modificar la classe

- Afegir les capçaleres dels nous mètodes en el fitxer . hh
- Implementar els nous mètodes en el fitxer .cc

Pro: Sovint més eficient

Con : Cal tenir accés *i entendre* la implementació original

#### Solució 1: Modificar la classe

- Afegir les capçaleres dels nous mètodes en el fitxer . hh
- Implementar els nous mètodes en el fitxer .cc

Pro: Sovint més eficient

Con : Cal tenir accés i entendre la implementació original

Addicionalment, pot modificar-se la representació del tipus (per poder soportar eficientment els nous mètodes) i això pot significar modificar el codi de les operacions ja existents

# Solució 2: Definir operacions fora de la classe

- No es modifica ni l'especificació ni la implementació de la classe original
- En un mòdul funcional nou (.hh i .cc), o en la classe que les usa
- Accions i funcions convencionals, no són mètodes

# Solució 2: Definir operacions fora de la classe

#### Avantatges:

- No cal tenir permís per modificar classe original
- No engreixa la classe original amb mètodes d'ús puntual
- No cal canviar el codi de les ops si canviés la implementació (només s'usen els mètodes públics)

#### Inconvenients:

- Possible ineficiència
- Incongruència amb el disseny OO

#### Com triar entre Solució 1 i Solució 2?

#### Solució 1:

- Si són ops. essencials al significat del tipus, generals i potencialment útils en moltes situacions (reusables)
- Quan solucioni problemes d'eficiència de la Solució 2

#### Solució 2:

- Quan només s'apliquen a un problema particular i no sembla que es pugui reutilitzar en altres contextes
- Quan cal evitar que la classe original creixi desmesuradament; potser s'ha de plantejar un redisseny de les classes i introduir-ne noves classes

## Exemple: Ampliació de Cjt\_estudiants

Volem afegir a Cjt\_estudiants operacions per a:

- donat el DNI d'un estudiant que sabem que és al conjunt, esborrar-lo del conjunt
- sabent que el conjunt no és buit, obtenir l'estudiant de nota màxima

#### Solució 1: Modificar la classe

```
class Cjt_estudiants {
public:
void esborrar estudiant(int dni);
/* Pre: el conjunt conté un estudiant amb DNI = dni */
/* Post: el conjunt conté els mateixos estudiants que
   l'original menys l'estudiant amb DNI donat */
. . .
Estudiant estudiant_nota_max() const;
/* Pre: el conjunt conté almenys un estudiant amb nota */
/* Post: el resultat és l'estudiant del conjunt amb
   nota màxima; si en té més d'un, és el de dni més petit */
};
```

#### Solució 1: Modificar la classe. Nou .hh

```
private:
    vector<Estudiant> vest;
    int nest;
    static const int MAX_NEST = 60;
    int imax; /* Aquest atribut és nou */

    /* Invariant de la representacio:
    ...
    imax val -1 si cap estudiant té nota, i altrament conté l'index més petit en vest d'un estudiant amb nota màxima */
```

#### Solució 1: Modificar la classe

- Creadores: imax s'inicialitza a -1
- Segueix sent -1 mentre cap estudiant del conjunt té nota
- afegir\_estudiant: s'actualitza imax, si cal
- Idem amb modificar\_estudiant i modificar\_iessim
- estudiant\_nota\_max(): retorna vest[imax]
- afegir\_estudiant i estudiant\_nota\_max ténen temps constant=independent del nombre d'estudiants en el conjunt
- modificar\_estudiant i modificar\_iessim podem necessitar un recorregut del vector sencer per a actualitzar imax

#### Solució 1: Modificar la classe

#### Solució 2: Definir mètodes fora de la classe

```
Nou fitxer E_Cjt_estudiants.hh

#include "Estudiant.hh"
#include "Cjt_estudiants.hh"

void esborrar_estudiant(Cjt_estudiants &Cest, int dni);
/* Pre: Cest conté un estudiant amb DNI = dni */
/* Post: Cest conté els mateixos estudiants que el seu valor original menys l'estudiant amb DNI dni */

Estudiant estudiant_nota_max(const Cjt_estudiants& Cest);
/* Pre: Cest conté almenys un estudiant amb nota */
/* Post: el resultat és l'estudiant de Cest amb nota màxima;
si en té més d'un, és el de dni més petit */
```

#### Solució 2: Definir mètodes fora de la classe

```
#include "E_Cjt_estudiants.hh"
/* Pre: Cest conté un estudiant a Cest amb DNI = dni */
/* Post: Cest conté els mateixos estudiants que el seu
         valor original menys l'estudiant amb DNI = dni */
void esborrar_estudiant(Cjt_estudiants &Cest, int dni) {
    Cit estudiants Cestaux;
    int i = 1;
    while (dni != Cest.consultar_iessim(i).consultar_DNI()) {
        Cestaux.afegir estudiant(Cest.consultar iessim(i));
        ++i;
    // per la pre, segur que trobarem a Cest un estudiant
    // amb DNI = dni; en aquest punt del programa, aquest
    // estudiant és Cest.consultar iessim(i);
    // ara hem d'afegir els elements següents a Cestaux
    for (int i = i+1; i <= Cest.mida(); ++i)</pre>
        Cestaux.afegir estudiant(Cest.consultar iessim(j));
    Cest = Cestaux:
```

#### Solució 2: Definir mètodes fora de la classe

```
/* Pre: Cest conté almenys un estudiant amb nota */
/* Post: el resultat és l'estudiant de Cest amb nota màxima;
   si en té més d'un, és el de dni més petit */
Estudiant estudiant_nota_max(const Cjt_estudiants &Cest) {
    int i = 1;
    while (not Cest.consultar_iessim(i).te_nota()) ++i;
    int imax = i; ++i;
    // per la pre, segur que trobarem a Cest un estudiant
    // amb nota; imax n'és el primer; comprovem la resta
    while (i <= Cest.mida()) {</pre>
        if (Cest.consultar_iessim(i).te_nota())
            if (Cest.consultar iessim(imax).consultar nota() <</pre>
                   Cest.consultar iessim(i).consultar nota())
                imax = i:
        ++i;
    return Cest.consultar_iessim(imax);
```

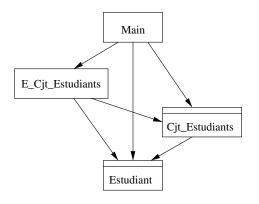
#### Solució 2: Definir mètodes fora de la classe

#### Observació:

- estudiant\_nota\_max() temps lineal (no constant)
- esborrar\_estudiant lineal com abans, però més lenta

### Diagrama modular

Un hipotètic programa principal que usa les dues operacions de  $E\_Cjt\_estudiants$ 



# Part I

# Disseny Modular

- 1) Abs vacció i disseny modular
- 2 Procomposició funcional i per dades
- Orientació a objectes
- Especificació i ús de classes
- 5 Jerarquies de Classes i Mòduls
- 6 Implementació de classes



8 Biblioteques i Genericitat

### **Biblioteques**

Col.leccions de mòduls que amplien el llenguatge

La *Standard C++ Library* ofereix una gran varietat de mòduls funcionals i de dades com ara iostream, string, cmath,...

## Standard Template Library (STL)

- La STL és un subconjunt de la biblioteca estàndar de C++.
   Inclou mòduls funcionals i de dades genèrics
- Template = plantilla
- Classes i funcions genèriques : classes i funcions amb tipus com a paràmetres Exemples:
  - Programació 1: vector<T>
  - Programació 2: queue<T>, stack<T>, list<T>

### **Templates**

#### Una funció genèrica

### **Templates**

### Ús d'una funció genèrica

## Part II

## **Estructures Lineals**

- 9 Estructures lineals: Generalitats
- 10 Jusus pila (stack)
  - 11) El tipus cua (queue)
- 12 Llistes

#### Estructures lineals: Generalitats

Una estructura lineal C és un conjunt d'elements d'un cert tipus T

$$C = [a_1, a_2, \ldots, a_n]$$

en el que es defineix una relació de successió

- Per tot i,  $1 \le i < n$ ,  $a_{i+1}$  és el successor de  $a_i$ . L'últim element  $a_n$  no té successor.
- Per tot i,  $1 < i \le n$ ,  $a_{i-1}$  és el predecessor de  $a_i$ . El primer element  $a_1$  no té predecessor.

Si n = 0 la estructura està buida

### Estructures lineals: Generalitats

- Piles (stack)
- Cues (queue)
- Llistes (list)

## Part II

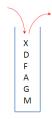
## **Estructures Lineals**

- Estructures lineals: Generalitats
- 10 El tipus pila (stack)
- 11) El tipus cua (queue)
- 12 Llistes

#### La classe stack

Ofereix tres operacions bàsiques:

- Afegir un nou element al final (empilar)
- Treure l'últim element (desempilar)
- Examinar l'últim element (cim)



LIFO - Last In, First Out: el darrer que ha entrat serà el primer en sortir, i és l'únic accessible

### Especificació de stack

```
La classe stack

template <class T> class stack {
public:
    // Constructores

/* Pre: cert */
    /* Post: crea una pila buida */
stack();

// Destructora
    ~stack();
```

### Especificació de stack

#### La classe stack

```
// Modificadores

/* Pre: la pila és [a_1, \ldots, a_n], n \geq 0 */

/* Post: s'afegit l'element x com a últim de la pila, es a dir, la pila és ara [a_1, \ldots, a_n, x] */

void push (const T& x);

/* Pre: la pila és [a_1, \ldots, a_n] i no està buida (n > 0) */

/* Post: s'ha eliminat el darrer element de la pila original, es a dir, la pila ara és [a_1, \ldots, a_{n-1}] */

void pop();
```

## Especificació de stack

```
La classe stack
// Consultores
/* Pre: la pila és [a_1,\ldots,a_n] i no està buida (n>0) */
/* Post: Retorna a_n */
T top() const;
/* Pre: cert */
/* Post: Retorna cert si i només si la pila està buida */
bool empty() const;
private:
};
```

## Part II

## **Estructures Lineals**

- (9) Estructures lineals: Generalitats
- 10 jous pila (stack)
- Il tipus cua (queue)
- 12 Llistes

### La classe queue

#### Ofereix tres operacions bàsiques:

- Afegir un nou element al final (encuar)
- Treure el primer element (desencuar)
- Examinar el primer element (front)



FIFO - First In, First Out: el primer que ha entrat serà el primer en sortir i és l'únic accessible

## Exemple d'evolució d'una cua

```
queue<int> c;
c.push(1); c.push(2); c.push(3);
c.pop();
c.push(4); c.push(5);
c.pop();
c.push(6); c.push(7);
```

1

1 2

1 2 3

2 3

2 3 4

2 3 4 5

3 4 5

3 4 5 6

3 4 5 6 7

## Especificació de la classe queue

```
La classe queue

template <class T> class queue {
public:
// Constructores

/* Pre: cert */
/* Post: crea una cua buida */
queue();

// Destructora
~queue();
```

### Especificació de queue

#### La classe queue

```
// Modificadores

/* Pre: la cua és [a_1, \ldots, a_n], n \ge 0 */

/* Post: s'afegit l'element x com a últim de la cua, es a dir, la cua és ara [a_1, \ldots, a_n, x] */

void push(const T& x);

/* Pre: la cua és [a_1, \ldots, a_n] i no està buida (n > 0) */

/* Post: s'ha eliminat el primer element de la cua original, es a dir, la cua ara és [a_2, \ldots, a_n] */

void pop();
```

## Especificació de queue

```
La classe queue
// Consultores
/* Pre: la cua és [a_1,\ldots,a_n] i no està buida (n>0) */
/* Post: Retorna a_1 */
T front() const;
/* Pre: cert. */
/* Post: Retorna cert si i només si la cua està buida */
bool empty() const;
private:
};
```

# Part II

## **Estructures Lineals**

- 9 Est ctures lineals: Generalitats
- 10 pus pila (stack)
- El tipus cua (queue)
- 12 Llistes
  - Llistes i Iteradors
  - Especificació de la classe Llista
  - Exemples d'operacions amb llistes
  - Splice
  - Accés directe: Ilistes vs. vectors



## Part II

## **Estructures Lineals**

- 9 Estructures lineals: Generalitats
- 10 El tipus pila (stack)
- El tipus cua (queue)
- 12 Llistes
  - Llistes i Iteradors
  - Especificació de la classe Llista
  - Exemples d'operacions amb llistes
  - Splice
  - Accés directe: Ilistes vs. vectors
  - Fusió ordenada

#### Llistes

Les llistes ens ofereixen operacions per a fer:

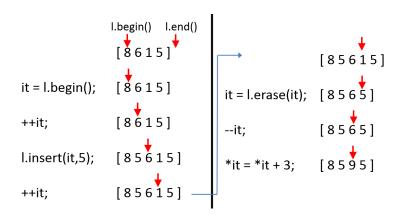
- Recorreguts seqüencials de tots els elements
- Inserció d'un element nou a qualsevol punt de la seqüència
- Eliminació d'un element qualsevol
- Concatenació

- El mecanisme que permet fer això amb les list de la STL són els iteradors
- Un iterador és un objecte que designa (marca, apunta, referencia) un element d'una llista o un altre contenidor
- Operacions sobre iteradors:
  - Avançar al següent element: ++it
  - Retrocedir a l'anterior: --it
  - Comparar iteradors: it1==it2, it1!=it2
  - Accedir a l'objecte designat: \*it

- Llistes i iteradors "treballen" coordinadament
- Operacions de llistes amb iteradors:
  - L.insert(it, x): insereix a la llista L un nou element x com a predecessor de l'element apuntat per it
  - itsuc = L.erase(it): elimina de la llista L l'element apuntat per it; retorna un iterador al successor de l'element esborrat

- Llistes i iteradors "treballen" coordinadament
- Operacions que ens tornen iteradors:
  - L.begin(): torna un iterador apuntant al primer element de la llista L
  - L.end(): torna un iterador apuntant "fora"—a un element fictici successor de l'últim—de la llista L
  - Si L és buida, aleshores L.begin() == L.end()

## Exemple d'evolució d'una llista



```
list<Estudiant> 1;
list<string> lp;

list<Estudiant>::iterator it = l.begin();
list<Estudiant>::iterator it2 = l.end();
list<string>::iterator it3 = lp.begin();

it = it3; // error!! són de tipus diferents
```

Cada tipus d'iterador es defineix com a subclasse de la classe "contenidora"

## Iteradors: Recorreguts

#### Esquema frequent:

```
list<T> L;
list<T>::iterator it = L.begin();
while (it != L.end() and not condició sobre *it)) {
    accedir a *it
    ++it;
}
```

#### Iteradors constants

- Iteradors constants (const\_iterator): prohibeixen modificar l'objecte referenciat per l'iterador
- S'han d'utilitzar per a recòrrer una llista rebuda per referència constant

#### Iteradors constants

```
void imprimir_llista(const list<Estudiant>& L) {
  for(list<Estudiant>::const_iterator it = L.begin();
     it != L.end(); ++it)
     (*it).escriure();
}
// en comptes de (*it).escriure() podem posar
// it -> escriure();
```

## Part II

## **Estructures Lineals**

- 9 Estructures lineals: Generalitats
- 10 El tipus pila (stack)
- El tipus cua (queue)
- 12 Llistes
  - Llistes i Iteradors
  - Especificació de la classe Llista
  - Exemples d'operacions amb llistes
  - Splice
  - Accés directe: llistes vs. vectors
  - Fusió ordenada

## Especificació de la classe list

```
template <class T> class list {
public:
// Subclasses de la classe llista
     class iterator { ... };
     class const_iterator { ... };
// Constructores
/* Pre: cert */
/* Post: El resultat es una llista sense cap element */
list();
// Destructora
~list();
```

```
// Modificadores
/* Pre: cert */
/* Post: La llista implícita queda buida */
void clear();

/* Pre: it referencia algun element existent a; a la llista o
        és igual a end(), la llista és [a1,...,an] */
/* Post: L'element s'ha inserit davant de l'element referenciat
per it, la llista és ara [a1,...,x,ai,...], torna
un iterador a l'element que s'acaba d'afegir */
iterator insert(iterator it, const T& x);
```

```
/* Pre: it referencia algun element a_i existent a la
         llista [a_1,\ldots,a_n], n>0 */
/* Post: S'ha eliminat l'element referenciat per it, la llista
          és ara [a_1,\ldots,a_{-1},a_{i+1},\ldots,a_n] i torna
          un iterador al sucessor de l'element eliminat */
iterator erase(iterator it);
/* Pre: l = [y_1, \dots, y_m], l i la llista implícita són
         objectes diferents, i it referencia algún element x_i de
         la llista implícita [x_1, \ldots, x_n] * /
/* Post: La llista implícita és ara
         [x_1,\ldots,x_{i-1},y_1,\ldots,y_m,x_i,\ldots,x_n] i l és buida \star/
void splice(iterator it, list& l);
```

```
// Consultores

/* Pre: cert */

/* Post: torna cert si i només si la llista és buida */
bool empty() const;

/* Pre: cert */

/* Post: torna el nombre d'elements de la llista*/
int size() const;
```

```
// tornen iteradors al primer element de la llista
const_iterator begin() const;
iterator begin();
// tornen iteradors a l'element fictici succesor de l'últim
// de la llista
const_iterator end() const;
iterator end();
private:
...
```

La inserció d'elements nous als extrems de la llista i l'esborrat dels extrems de la llista es pot fer sense iteradors:

```
l.push_back(x); // = l.insert(l.end(), x);
l.push_front(x); // = l.insert(l.begin(), x);

l.pop_front(); // = l.erase(l.begin());
l.pop_back(); // = it = l.end(); l.erase(--it);
```

# Part II

## **Estructures Lineals**

- 9 Estructures lineals: Generalitats
- 10 El tipus pila (stack)
- El tipus cua (queue)
- 12 Llistes
  - Llistes i Iteradors
  - Especificació de la classe Llista
  - Exemples d'operacions amb llistes
  - Splice
  - Accés directe: Ilistes vs. vectors
  - Fusió ordenada

## Sumar tots els elements d'una llista d'enters

## Cerca senzilla en una llista d'enters

## Exercici: cerca en una llista d'estudiants

```
/* Pre: cert */
/* Post: El resultat ens indica si hi ha algun estudiant
  amb dni x a l o no */
bool pertany(const list<Estudiant>& l, int x);
```

# Modificar una llista sumant un valor $\Bbbk$ a tots els elements

```
/* Pre: l = [x<sub>1</sub>,...,x<sub>n</sub>] */
/* Post: l = [x<sub>1</sub> + k,x<sub>2</sub> + k,...,x<sub>n</sub> + k] */
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
```

### Una alternativa

#### En comptes de fer

```
*it += k;
++it;
```

podriem eliminar l'element i tornar a afegir-ho

```
int aux = (*it) + k;
it = l.erase(it); // it apunta al successor
l.insert(it,aux);
```

però és molt menys eficient (implica creació+destrucció d'objectes!)

# Dir si una llista és capicua

[4,8,5,8,4], [7], [4,8,8,4] són capicues

```
/* Pre: cert */
/* Post: El resultat diu si l es capicua */
bool capicua(const list<int>& l);
```

# Dir si una llista és capicua

## Dir si una llista és capicua

- Exercici: Penseu com fer-ho sense usar l.size().
- Cada element s'ha de consultar un cop com a molt.
- Recordeu que no es pot comparar it1 < it2</li>

#### Inserint elements ordenadament

#### Exemple:

Donada una llista l d'strings en ordre alfabètic no decreixent i un nou string s, inserir s a la llista l, mantentint l'ordre.

```
// Pre: l = L // Post: l conté els elements d'L i s, i està en ordre // no decreixent void inserir_ordenadament(list<string>& l, string s);
```

#### Inserint elements ordenadament

```
void inserir_ordenadament(list<string>& 1, string s) {
   list<string>::iterator it = ...;
    ...
   // it == l.end() ó *it és un string ≥ s
   // per tant s s'ha d'inserir com a predecessor
   // de l'element apuntat per it
   l.insert(it, s);
}
```

#### Inserint elements ordenadament

```
void inserir_ordenadament(list<string>& 1, string s) {
   list<string>::iterator it = 1.begin();
   while (it != 1.end() and *it < s) ++it;
   // it == 1.end() ó *it és un string ≥ s
   // per tant s s'ha d'inserir com a predecessor
   // de l'element apuntat per it
   l.insert(it, x);
}</pre>
```

```
Cit Estudiant.hh
class Cit Estudiants {
public:
private:
// lest està ordenada per DNI creixent, emmgatzema
// els estudiants del conjunt
   list<Estudiant> lest;
// Pre: cert
// Post: torna un iterador a un estudiant amb DNI = dni
// si hi ha algún estudiant amb aguest DNI, altrament
// l'iterador apunta al primer estudiant amb DNI > dni o
// l'iterador és l.end() si no hi ha cap estudiant amb DNI
// més gran que dni
   static list<Estudiant>::const iterator
     cerca estudiant (const list < Estudiant > & l. int dni);
```

```
Cit Estudiant.cc
void Cjt_estudiants::afegir_estudiant(const Estudiant& est) {
  int dni = est.consultar DNI();
  list<Estudiant>::iterator it = cerca estudiant(lest, dni);
  // it == lest.end() o it -> consultar DNI() > dni
  // l'estudiant 'est' segur que no està en el conjunt
  lest.insert(it, est);
void Cjt_estudiants::modificar_estudiant(const Estudiant& est) {
  int dni = est.consultar DNI();
  list<Estudiant>::iterator it = cerca estudiant(lest, dni);
  // it -> consultar DNI() == dni) {
  // l'estudiant est segur que està en el conjunt
  *it = est:
```

```
cjt_Estudiant.cc
int Cjt_Estudiant::mida() const {
   return lest.size();
}
bool Cjt_Estudiant::existeix_estudiant(int dni) const {
   list<Estudiant>::const_iterator it =
        cerca_estudiant(lest,dni);
   return it != lest.end() and it -> consultar_DNI() == dni;
}
```

```
Cjt_Estudiant.cc

Estudiant Cjt_Estudiant::consultar_estudiant(int dni) const {
   list<Estudiant>::const_iterator it =
        cerca_estudiant(lest,dni);
   return *it;
}
```

```
void Cjt_Estudiant::escriure() const {
  list<Estudiant>::const_iterator it = lest.begin();
  bool first = true;
  while (it != lest.end()) {
    if (first) first = false; else cout << " ";
    cout << it -> escriure();
        // invoca el mètode Estudiant::escriure()
        // sobre l'estudiant al qual apunta it
  }
}
```

La implementació dels mètodes consultar\_iessim i modificar\_iessim no és complicada, però fa palès que les llistes (list) no són el més apropiat, ja que no tenim accés directe. Més sobre aquest punt i les diferències entre llistes i vectors més endavant—resteu a l'espera.

# Part II

## **Estructures Lineals**

- 9 Estructures lineals: Generalitats
- 10 El tipus pila (stack)
- El tipus cua (queue)
- 12 Llistes
  - Llistes i Iteradors
  - Especificació de la classe Llista
  - Exemples d'operacions amb llistes
  - Splice
  - Accés directe: Ilistes vs. vectors
  - Fusió ordenada

# Splice: Insert a l'engrós!

```
• Si 11 = [1,2,3,4,5,6], it apunta al 4, i 12 = [10,20,30] llavors

11.splice(it,12),
queda
```

# Splice: Insert a l'engrós!

## Splice: Insert a l'engrós!

- Per concatenar dues llistes farem:11.splice(11.end(),12)
- La STL de C++ té variants més complexes de splice que fan altres tipus de "transferència" de continguts entre llistes
- El cost de splice és constant, no depèn de les longituds de les llistes

# Part II

## **Estructures Lineals**

- 9 Estructures lineals: Generalitats
- 10 El tipus pila (stack)
- El tipus cua (queue)
- 12 Llistes
  - Llistes i Iteradors
  - Especificació de la classe Llista
  - Exemples d'operacions amb llistes
  - Splice
  - Accés directe: Ilistes vs. vectors
  - Fusió ordenada

 Recorregut seqüencial: Temps lineal en els dos casos, constant per element

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element
  - Constant en llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element
  - Constant en llistes
  - En vectors, afegir al final és constant en mitjana (push\_back())

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element
  - Constant en llistes
  - En vectors, afegir al final és constant en mitjana (push\_back())
  - En vectors, inserir pel mig és costós

 Esborrar un element: constant en llistes, costós en vectors (excepte l'últim pop\_back())

#### Vectors vs. Ilistes

- Esborrar un element: constant en llistes, costós en vectors (excepte l'últim pop\_back())
- Splice: constant en llistes, costós en vectors

## Accés directe?

Accés directe per posició en llistes. Si cal ...

```
// pre: 0 <= i < l.size()
// post: retorna l'i-essim element de l
template <typename T>
T get(const list<T>& l, int i) {
    list<T>::const_iterator it = l.begin();
    for (int j = 0; j < i; ++j) ++it;
    return *it;
}</pre>
```

## Accés directe

## Cost lineal

```
list<double>::iterator it = 1.begin();
double sum = 0;
while (it != 1.end()) {
   sum += *it; ++it;
}
```

## Cost quadràtic

```
double sum = 0;
for (int i = 0; i < 1.size(); ++i)
    sum += get(1,i);</pre>
```

## Part II

## **Estructures Lineals**

- 9 Estructures lineals: Generalitats
- 10 El tipus pila (stack)
- El tipus cua (queue)
- 12 Llistes
  - Llistes i Iteradors
  - Especificació de la classe Llista
  - Exemples d'operacions amb llistes
  - Splice
  - Accés directe: Ilistes vs. vectors
  - Fusió ordenada

Suposem que tenim una llista *l* ordenada (per DNI) d'Estudiants, i una altra llista també ordenada per DNI d'elements del tipus

En aquesta segona llista es detalla: estudiants que s'han de donar d'alta (assumim que efectivament NO estàn a la llista l); estudiants que s'han de donar de baixa (i estàn a la llista l o s'han afegit) i estudiants als quals se'ls ha de modificar o agregar nota (ja hi eren a la llista l o s'han afegit).

Una solució eficient d'aquest problema ha d'aprofitar que les dues llistes estàn ordenades alfabèticament!

Per exemple (substituint DNIs per noms per fer més entenedor l'exemple) si

```
l = [\langle \mathsf{ALICE}, 3.5 \rangle, \langle \mathsf{BOB}, 4.1 \rangle, \langle \mathsf{CHARLIE}, 7.4 \rangle, \langle \mathsf{DAISY}, 6.8 \rangle, \\ \langle \mathsf{HELEN}, 9.1 \rangle, \langle \mathsf{JOHN}, 3.7 \rangle, \langle \mathsf{MARY}, 5.3 \rangle]
```

i la llista d'actualitzacions és

```
\begin{split} \textit{lact} = [\langle \text{'a'}, \langle \text{ALBERT}, \textit{NP} \rangle \rangle, \langle \text{'b'}, \langle \text{BOB}, \ldots \rangle \rangle, \langle \text{'m'}, \langle \text{HELEN}, 10 \rangle \rangle, \\ \langle \text{'b'}, \langle \text{JOHN}, \ldots \rangle \rangle, \langle \text{'a'}, \langle \text{JOHN}, 4 \rangle \rangle, \\ \langle \text{'m'}, \langle \text{JOHN}, 4.2 \rangle \rangle, \langle \text{'m'}, \langle \text{JOHN}, 4.1 \rangle \rangle, \langle \text{'a'}, \langle \text{PETER}, 5.5 \rangle \rangle] \end{split}
```

el resultat d'actualitzar l seria

```
\begin{split} l = [\langle \mathsf{ALBERT}, NP \rangle, \langle \mathsf{ALICE}, 3.5 \rangle, \langle \mathsf{CHARLIE}, 7.4 \rangle, \langle \mathsf{DAISY}, 6.8 \rangle, \\ \langle \mathsf{HELEN}, 10 \rangle, \langle \mathsf{JOHN}, 4.1 \rangle, \langle \mathsf{MARY}, 5.3 \rangle, \langle \mathsf{PETER}, 5.5 \rangle] \end{split}
```

```
void actualitza_llista_Estudiants(list<Estudiant>& 1,
                       const list<Actualitzacio>& lact) {
  list<Estudiant>::iterator it = l.begin();
  list < Actualitzacio > :: iterator itact = lact.begin();
  while (it != l.end() and itact != lact.end()) {
    if (it -> consultar_DNI() <</pre>
        itact -> est.consultar_DNI())
     // l'estudiant al que apunta it no està afectat
     // per cap actualització
        ++it;
    . . .
  // processar la resta d'actualitzacions
  // que puqui haver
```

```
else if (it -> consultar_DNI() >
            itact -> est.consultar_DNI()) {
 // això ha de ser un 'alta' necessàriament
 it = l.insert(it, itact -> est);
 // it apunta a l'element que acabem d'afegir
++itact;
} else {
 // it -> consultar_DNI() == itact -> est.consultar_DNI()
// això és necessàriament una 'baixa'
// o una 'modificació'
 if (itact -> op == 'b')
  it = l.erase(it);
 else
   *it = itact -> est;
 ++itact;
```

```
// si itact == lact.end() no queda cap
// actualització més per processar i no cal fer res més;
// altrament totes les actualitzacions
// pendents afecten estudiants amb DNI més gran
// que qualsevol que haqués a l, i it == l.end()
 while (itact != lact.end()) {
    if (itact -> op == 'a')
    it = l.insert(l.end(), itact -> est);
    // it apunta a l'element que acabem d'afegir
    ++itact:
    } else {
    // si és una baixa o modificació ha d'afectar
    // a un element acabat d'afegir (i apuntat per it)
    if (itact -> op == 'b')
     l.erase(it);
    else
      *it = itact -> est;
    ++itact;
```

Suposem que l té n=10000 elements i que lact conté m=1000 actualitzacions. Si l'actualització de l la fessim amb

```
itact = lact.begin();
while (itact != lact.end()) {
   Estudiant e = itact -> est;
   char op = itact -> op;
   if (op == 'a')
        afegeix(l, est);
   else if (op == 'b')
        elimina(l, est);
else
   modifica(l, est);
}
```

o quelcom equivalent hauriem de fer unes  $n \cdot m = 10^7$  (deu millions) d'operacions en el pitjor dels casos i de l'ordre de 5 millions d'operacions en promig, ja que operacions com afegir, elimina o modifica han de recòrrer la llista l (en promig la meitat).

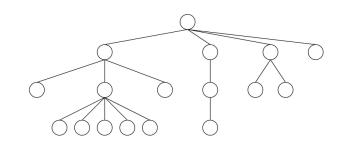
Però el nostre algorisme de "fusió ordenada" avança en cada pas/iteració en una de les dues llistes (o totes dues) i per tant el nombre d'operacions que farem serà de l'ordre d'n + m = 11000 operacions. La diferència és enorme! És pràcticament 1000 vegades més eficient!

Les llistes ens han ajudat molt en aquest cas, doncs són molt flexibles i podem afegir i eliminar elements apuntats per iteradors molt eficientment. Un algorisme semblant treballant amb vectors hauria de produir un vector auxiliar l' amb el resultat de l'actualització i en acabar copiar l' sobre l. No podem fer altes o baixes eficientment directament sobre l. Si treballem amb dades ordenades SEMPRE hem de pensar com podem treure profit i fer que els nostres algorismes siguin més eficients.

# Part III Arbres

- 13 Arbres generals i arbres N-aris
- 14 A bes binaris: Classe BinTree
- 15) Operacions amb arbres binaris
- 16 Recorreguts canònics d'arbres

- node o nus
- fill, pare
- descendent, ascendent
- germà
- arrel, fulla
- camí
- nivell; alçària



Definicions com a graf:

Def. 1: un arbre és un graf dirigit tal que o bé és buit, o bé té un node anomenat arrel tal que hi ha exactament un camí de l'arrel a qualsevol altre node

Definicions com a graf:

Def. 1: un arbre és un graf dirigit tal que o bé és buit, o bé té un node anomenat arrel tal que hi ha exactament un camí de l'arrel a qualsevol altre node

Def. 2: un arbre és un graf no dirigit, connex, amb un arc menys que nodes i un node distingit anomenat arrel

Definicions com a graf:

Def. 1: un arbre és un graf dirigit tal que o bé és buit, o bé té un node anomenat arrel tal que hi ha exactament un camí de l'arrel a qualsevol altre node

Def. 2: un arbre és un graf no dirigit, connex, amb un arc menys que nodes i un node distingit anomenat arrel

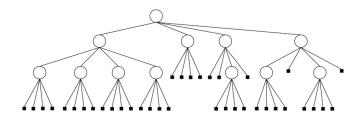
Les dues són poc útils algorísmicament

Un arbre o bé és l'arbre buit o bé és un node anomenat arrel amb zero o més arbres successors anomenats fills o subarbres

Un arbre o bé és l'arbre buit o bé és un node anomenat arrel amb zero o més arbres successors anomenats fills o subarbres

- Es presta a tractaments algorísmics recursius
- Tècnicament la definició correspón a arbres arrelats ordenats

#### Arbres N-aris



- Def.: Tots els subarbres no buits tenen exactament el mateix nombre de fills, N, que poden ser buits o no
- Exemple: Arbre 4-ari; quadrats negres = arbres buits
- Per claredat convé representar explícitament els arbres buits en els arbres N-aris; típicament els subarbres buits es representen mitjançant un quadrat (negre o blanc)

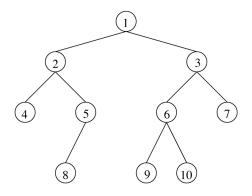
# Part III Arbres

- 13) Arbres generals i arbres N-aris
- 14 Arbres binaris: Classe BinTree
  - 15 Operacions amb arbres binaris
- 16 Recorreguts canònics d'arbres

### Arbres binaris

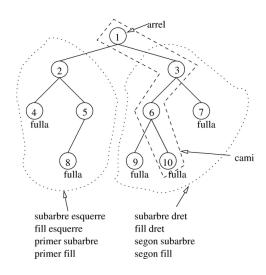
- Cas particular dels arbre N-aris amb N=2
- Quan diem arbres sense detallar més, ens referim per defecte a arbres binaris
- Els dos fills d'un node són anomenats esquerre i dret

## Exemple d'arbre binari



No hem dibuixat els subarbres buits amb quadrats negres com en l'exemple d'arbre 4-ari. La inclinació de cada aresta indica si el fill és dret o esquerre

## Exemple d'arbre binari



## Especificació dels arbres binaris

```
template <typename T> class BinTree {
public:
BinTree():
/* Pre: cert */
/* Post: crea un arbre buit */
BinTree (const T& x):
/* Pre: cert */
/* Post: crea un arbre binari amb un sol node, l'arrel,
          que conté x, i els seus fills esquerre i dret
          són buits */
BinTree (const T& x, const BinTree& left, const BinTree& right);
/* Pre: cert */
/* Post: crea un arbre binari amb x a l'arrel,
          i left i right com com a fills esquerre
          i dret, respectivament */
```

## Especificació dels arbres binaris

```
// Consultores:
bool empty() const;
/* Pre: cert. */
/* Post: retorna cert si i només si
        l'arbre és buit */
BinTree left() const;
/* Pre: L'arbre implícit no és buit */
/* Post: retorna el fill esquerre de l'arbre implícit */
BinTree right() const;
/* Pre: L'arbre implicit no és buit */
/* Post: retorna el fill dret de l'arbre implícit */
const T& value() const;
/* Pre: L'arbre implicit no és buit */
/* Post: retorna el valor de l'arrel de l'arbre */
```

## Especificació dels arbres binaris

- Cap modificadora! La única manera de modificar un arbre és construir l'arbre modificat i assignar-lo a l'original.
- Totes les operacions requereixen temps constant (excepte la destructora)
- Important per al temps constant: tot és const, no es fan còpies dels fills
- En l'assignació

a1 = a2:

requereix temps constant excepte en el cas de que al no "comparteixi" cap subarbre amb cap altre objecte; llavors el temps necessari és proporcional a la mida d'al doncs cal destruir-lo

# Part III Arbres

- Arbres generals i arbres N-aris
- 14 Asses binaris: Classe BinTree
- 15 Operacions amb arbres binaris
- 16 Recorreguts canònics d'arbres

### Mida d'un arbre

```
/* Pre: cert */
/* Post: El resultat és el nombre de nodes d' a */
template <typename T>
int size(const BinTree<T>& a);
```

#### Mida d'un arbre

```
/* Pre: cert */
/* Post: El resultat és el nombre de nodes d'a */
template <typename T>
int size(const BinTree<T>& a) {
   if (a.empty()) return 0;
   else return 1 + size(a.left()) + size(a.right());
}
```

## Alçària d'un arbre

Def.: L'alçària d'un arbre és la longitud del camí (nombre de nodes) més llarg de l'arrel a una fulla

#### Especificació:

```
/* Pre: cert */
/* Post: El resultat és l'alçària de l'arbre a*/
template <typename T>
int alcaria(const BinTree<T>& a);
```

## Alçària d'un arbre

- $alcaria(\square) = 0$
- si a no buit, ...

```
alcaria(a) = 1 + max(alcaria(a.left()), alcaria(a.right()))
```

Demostració: per inducció!

## Alçària d'un arbre

```
/* Pre: cert */
/* Post: El resultat és l'alçària de l'arbre a */
template <typename T>
int alcaria(const BinTree<T>& a) {
  if (a.empty())
    return 0;
  else
    return 1 + max(alcaria(a.left(),alcaria(a.right());
}
```

#### Cerca d'un valor en un arbre

```
/* Pre: cert */
/* Post: El resultat indica si x és a l'arbre a o no */
template <typename T>
bool cerca(const BinTree<int>& a, const T& x);
```

#### Cerca d'un valor en un arbre

```
cerca(buit,x) = fals \\ cerca(a,x) = cert, \quad si \; arrel(a) = x \\ cerca(a,x) = cerca(a.left(),x) \; or \; cerca(a.right(),x), \quad si \; arrel(a) \neq x
```

#### Cerca d'un valor en un arbre

És imprescindible que l'operador d'igualtat estigui definit per elements del tipus T: si x y z són de tipus T llavors x == y ha d'estar definit! Nota: Eficient perquè or és condicional

#### Sumar un valor k a tots els nodes

```
/* Pre: cert */
/* Post: retorna un arbre amb la mateixa forma que a,
i en el qual cada node val k més el valor del node
corresponent en a */
BinTree suma(const BinTree<int>& a, int k);
```

#### Sumar un valor k a tots els nodes

#### Sumar un valor k a tots els nodes

Fem-ho sobre el mateix arbre, com una acció:

```
/* Pre: a = A */
/* Post: deixa en a el resultat de sumar k a l'arbre A */
void suma(BinTree<int>& a, int k) {
   if (not a.empty()) { // si és buit no cal fer res
      BinTree<int> l = a.left();
      BinTree<int> r = a.right();
      suma(l, k);
      suma(r, k);
      a = BinTree<int>(a.value() + k, l, r);
}
```

# Part III Arbres

- Arbres generals i arbres N-aris
- 14 A ses binaris: Classe BinTree
- 15) Operacions amb arbres binaris
- 16 Recorreguts canònics d'arbres

Mètodes més habituals per visitar els nodes d'un arbre (per fer recorreguts o cerques):

Recorreguts en profunditat

Mètodes més habituals per visitar els nodes d'un arbre (per fer recorreguts o cerques):

- Recorreguts en profunditat
  - En preordre

Mètodes més habituals per visitar els nodes d'un arbre (per fer recorreguts o cerques):

- Recorreguts en profunditat
  - En preordre
  - En inordre

Mètodes més habituals per visitar els nodes d'un arbre (per fer recorreguts o cerques):

- Recorreguts en profunditat
  - En preordre
  - En inordre
  - En postordre

Mètodes més habituals per visitar els nodes d'un arbre (per fer recorreguts o cerques):

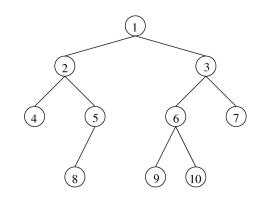
- Recorreguts en profunditat
  - En preordre
  - En inordre
  - En postordre
- Recorregut en amplada o per nivells

# Recorreguts en profunditat: preordre

- visitar l'arrel
- recórrer fill esquerre (en preordre)
- recórrer fill dret (en preordre)

## Exemple:

1, 2, 4, 5, 8, 3, 6, 9, 10 i 7

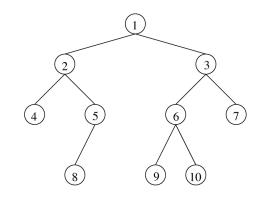


# Recorreguts en profunditat: inordre

- recórrer fill esquerre (en inordre)
- visitar l'arrel
- recórrer fill dret (en inordre)

# Exemple:

4, 2, 8, 5, 1, 9, 6, 10, 3, i 7

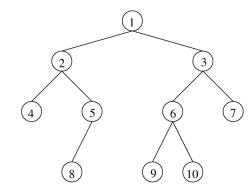


# Recorreguts en profunditat: postordre

- recórrer fill esquerre (en postordre)
- recórrer fill dret (en postordre)
- visitar l'arrel

## Exemple:

4, 8, 5, 2, 9, 10, 6, 7, 3, i 1



# Recorregut en preordre

Exercici: Com canviem les instruccions del mig per obtenir els recorreguts en inordre i en postordre?

# Recorregut en inordre

Manera alternativa: afegir a una llista donada Obtenim el recorregut fent una crida inicial amb la llista buida

```
/* Pre: 1 = I_1 */
/* Post: l conté L seguida dels nodes d'a en inordre */
template <typename T>
void inorder(const BinTree<T>& a, list<T>& l) {
    if (not a.empty()) {
        inorder(a.left(),1);
        1.push back(a.value());
        inorder(a.right(),1);
// Ús:
BinTree<int> a:
list<int> rec;
inorder (a, rec);
```

# Recorregut en amplada o per nivells

Visita d'els nodes d'un arbre donat de manera que:

- ullet tots els nodes del nivell i s'han visitat abans que els del nivell i+1
- dins de cada nivell, els nodes es visiten d'esquerra a dreta

#### Recorregut en amplada o per nivells Es fa amb una cua Repetir:

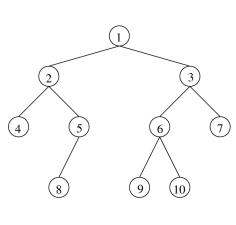
- agafar primer arbre de la cua;
- visitar la seva arrel;
- ficar els seus dos fills a la cua;

# Recorregut en amplada o per nivells Es fa amb una cua

#### Repetir:

- agafar primer arbre de la cua;
  - visitar la seva arrel;
- ficar els seus dos fills a la cua;

Al llarg de tot l'algorisme, en cada iyeració la cua conté alguns nodes del nivell k seguits de nodes del nivel k+1que són fills dels nodes de nivell k que ja han sigut visitats i no són a la cua. En cap moment la cua conté nodes de més de dos nivells consecutius i mai un node de nivell k+1precedeix un altre de nivel k a



# Recorregut en amplada

```
/* Pre: cert */
/* Post: El resultat conté el recorregut d'a en amplada */
template <typename T>
list<T> nivells(const BinTree<T>& a) {
    list<T> l; // inicialment, buida
    if (not a.empty())
        queue < BinTree < T > c;
        c.push(a);
        while (not c.empty()) {
           BinTree<T> aux = c.front();
           : () gog. p
           1.push back(aux.value());
           if (not aux.left().emptv()) c.push(aux.left());
           if (not aux.right().empty()) c.push(aux.right());
    return 1:
```

# Part IV

# Disseny Iteratiu: Verificació i Derivació

- Correctesa de programes
- 18) Estats i assercions
- Correctesa de programes iteratius
- 20 Disseny induction

# Correctesa d'un programa

#### Definició:

L'estat d'un programa en un punt determinat de la execució vé donat pel valor de totes les variables actives en aquell punt.

```
// Estat = ( x = 3, y = 7, ... )
++x;
// Estat = ( x = 4, y = 7, ... )
```

# Correctesa d'un programa

Definició: Correcció d'un programa

Si l'estat inicial del programa o funció satisfà la Precondició, llavors el programa acaba en un nombre finit de pasos i l'estat final satisfà la Postcondició

# Correctesa d'un programa

Definició: Correcció d'un programa

Si l'estat inicial del programa o funció satisfà la Precondició, llavors el programa acaba en un nombre finit de pasos i l'estat final satisfà la Postcondició

- Com sabem que un programa és correcte?
- Només podem fer un nombre finit (i petit) de proves
- Raonament genèric sobre els estats del programa

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X^Y
```

#### Ho he provat i ...

```
// Pre: x = X and y = Y ≥ 0
int p = 1;
while (y > 0) {
  p = p * x;
  y = y - 1;
}
// Post: p = X<sup>Y</sup>
```

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X^Y
```

Ho he provat i . . . amb 5 i 3 dona 125

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X^Y
```

Ho he provat i ...
...amb 5 i 3 dona 125
...amb 0 i 100 dona 0

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X^Y
```

Ho he provat i ... ... amb 5 i 3 dona 125 ... amb 0 i 100 dona 0 ... amb -4 i 2 dona 16

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X^Y
```

Ho he provat i ...
...amb 5 i 3 dona 125
...amb 0 i 100 dona 0
...amb -4 i 2 dona 16
...amb 4 i -2 no cal provar (no es compleix la Pre)

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X^Y
```

```
Ho he provat i ...
...amb 5 i 3 dona 125
...amb 0 i 100 dona 0
...amb -4 i 2 dona 16
...amb 4 i -2 no cal provar (no es compleix la Pre)
...per tant, és correcte!
```

```
// Pre: x = X and y = Y \ge 0

int p = 1;

while (y > 0) {

 p = p * x;

 y = y - 1;

}

// Post: p = X^Y
```

```
Ho he provat i ...
...amb 5 i 3 dona 125
...amb 0 i 100 dona 0
...amb -4 i 2 dona 16
...amb 4 i -2 no cal provar (no es compleix la Pre)
...per tant, és correcte!
```

Nombre finit (petit) de casos ≠ Tots els casos

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X^Y
```

"Inicialitzem p a 1 (el producte de 0 factors).

```
// Pre: x = X and y = Y \ge 0

int p = 1;

while (y > 0) {

p = p * x;

y = y - 1;

}

// Post: p = X^Y
```

```
// Pre: x = X and y = Y ≥ 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = XY
```

"Inicialitzem p a 1 (el producte de 0 factors). Llavors, anem multiplicant p per x i decrementant y en cada pas.

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
  p = p * x;
  y = y - 1;
}
// Post: p = X^Y
```

"Inicialitzem p a 1 (el producte de 0 factors).

Llavors, anem multiplicant p per x i decrementant y en cada pas.

Repetim fins que y = 0, i llavors ja hem acabat.

#### Demostració de correctesa?

```
// Pre: x = X and y = Y \ge 0
int p = 1;
while (y > 0) {
  p = p * x;
  y = y - 1;
}
// Post: p = X^Y
```

"Inicialitzem p a 1 (el producte de 0 factors).

Llavors, anem multiplicant p per x i decrementant y en cada pas.

Repetim fins que y=0, i llavors ja hem acabat.

Ja es veu que a p tindrem  $X^Y$ ."

#### Demostració de correctesa?

```
// Pre: x = X and y = Y \ge 0

int p = 1;

while (y > 0) {

p = p * x;

y = y - 1;

}

// Post: p = X^Y
```

"Inicialitzem p a 1 (el producte de 0 factors).

Llavors, anem multiplicant p per x i decrementant y en cada pas.

Repetim fins que y = 0, i llavors ja hem acabat.

Ja es veu que a p tindrem  $X^Y$ ."

Llegir el programa



Dir per què satisfà la seva espec

### Com ho fem, doncs?

- Raonament genèric sobre tots els estats possibles
- L'eina principal és la inducció
- En programes recursius, aplicada directament
- En programes iteratius, amagada en els invariants

# Part IV

# Disseny Iteratiu: Verificació i Derivació

- 17 Correctesa de programes
- 18 Estats i assercions
- Correctesa de programes iteratius
- 20 Disseny induction

 Recordem: estat d'un programa = valors de totes les variables

```
(x = 10, y = -5, b = true)
(x = 10, y = -15, b = false)
```

 Recordem: estat d'un programa = valors de totes les variables

```
(x = 10, y = -5, b = true)
(x = 10, y = -15, b = false)
```

Asserció: Descripció d'un conjunt d'estats

```
P(x,y,b) = b = (x + y > 0)
```

 Recordem: estat d'un programa = valors de totes les variables

```
(x = 10, y = -5, b = true)
(x = 10, y = -15, b = false)
```

Asserció: Descripció d'un conjunt d'estats

$$P(x,y,b) = b = (x + y > 0)$$

 El comentari "// P" o "/\* P \*/" en un programa vol dir "en aquest punt l'estat delprograma compleix P"

 Recordem: estat d'un programa = valors de totes les variables

```
(x = 10, y = -5, b = true)
(x = 10, y = -15, b = false)
```

Asserció: Descripció d'un conjunt d'estats

```
P(x,y,b) = b = (x + y > 0)
```

- El comentari "// P" o "/\* P \*/" en un programa vol dir "en aquest punt l'estat delprograma compleix P"
- La Precondició (Pre) és l'asserció que l'estat inicial ha de satisfer
- La Postcondició (Post) és l'asserció que ha de ser certa per l'estat final; altrament el programa no satisfà l'especificació

 Mètode: Anotarem el programa amb assercions que descriuen els estats en diferents punts, i argumentarem que cada anotació està ben feta

- Mètode: Anotarem el programa amb assercions que descriuen els estats en diferents punts, i argumentarem que cada anotació està ben feta
- Un programa és correcte si és cert que

```
/* Pre */ programa /* Post */
```

Donada una asserció P,  $P(x \leftarrow E)$  és l'asserció resultant de reemplaçar simultàniament les aparicions d'x en l'asserció P per l'expressió E, e.g.,  $P = "x \ge 5"$ ,

$$P(x \leftarrow y + 3) = "y + 3 \ge 5"$$

Assignació:

$$/* P(x \leftarrow E) */ x = E /* P */$$

Composició seqüèncial:

Si /\* 
$$P_1$$
 \*/ S1 /\*  $Q_1$  \*/ és correcte, /\*  $P_2$  \*/ S2 /\*  $Q_2$  \*/ és correcte i  $Q_1$   $\Longrightarrow$   $P_2$  llavors /\*  $P_1$  \*/ S1; S2 /\*  $Q_2$  \*/

és correcte.

Composició alternativa/condicional:

```
Si /* P \land B */ S1 /* Q */ és correcte,
i /* P \land \neg B */ S2 /* Q */ és correcte llavors
    /* P */
    if (B) S1
    else S2
    /* Q */
```

és correcte.

# Part IV

# Disseny Iteratiu: Verificació i Derivació

- 17 Confectesa de programes
- (18) Estats i assercions
- 19 Correctesa de programes iteratius
- 20 Disseny induction

#### Correctesa d'un bucle

### Esquema bàsic

```
// Pre: P
inicialitzacions;
// Pre (del bucle): P'
while (B) {
  cos
}
// Post (del bucle): Q'
tractament final;
// Post: Q
```

### L'invariant: Concepte i ús

- Invariant: Una asserció I que és certa després de qualsevol nombre d'iteracions (inclós 0); per tant cal que P' ⇒ I
- A més, quan el bucle acaba, implica la Post:
   I ∧ ¬B ⇒ Q'

# L'invariant: Concepte i ús

- Invariant: Una asserció I que és certa després de qualsevol nombre d'iteracions (inclós 0); per tant cal que P' ⇒ I
- A més, quan el bucle acaba, implica la Post:
   I ∧ ¬B ⇒ Q'
- Que l'asserció I és un invariant es demostra per inducció sobre el nombre d'iteracions i: s'ha de cumplir

### Esquema bàsic

```
// I \wedge B cos del bucle // I
```

# L'invariant: Concepte i ús

- Invariant: Una asserció I que és certa després de qualsevol nombre d'iteracions (inclós 0); per tant cal que P' ⇒ I
- A més, quan el bucle acaba, implica la Post:  $I \wedge \neg B \implies Q'$
- Que l'asserció I és un invariant es demostra per inducció sobre el nombre d'iteracions i: s'ha de cumplir

# Esquema bàsic

```
\begin{array}{c} //\ I \wedge B \\ \\ \text{cos del bucle} \\ //\ I \end{array}
```

- Finalment, cal demostrar (potser usant l'invariant I) que el bucle segur que acaba
- Trobar i explicitar l'invariant d'un bucle és molt bona documentació d'un bucle: explica per què funciona!

#### Demostració d'acabament

- Funció de fita: Una funció f sobre les variables que diuen quantes iteracions queden com a molt
- Ha de tenir valor enter no negatiu: per a qualsevol estat del programa  $f \ge 0$
- Cal que decreixi (al menys en 1) a cada iteració
- Si fem una iteració més, segur que f > 0

#### Passos

0 Inventar un invariant I i una funció de fita f Demostrar que:

- 1 Les inicialitzacions del bucle estableixen l'invariant:  $P' \implies I$
- 2 Si es compleix l'invariant i s'entra en el bucle, al final d'una iteració torna a complir-se l'invariant:  $/*I \land B */ \cos /*I */$
- 3 L'invariant i la negació de la condició d'entrada al bucle impliquen la Postcondició: I ∧ ¬B ⇒ Q'
- 4 La funció de fita decreix a cada iteració:
  - $/\star~I \wedge B \wedge f = F~\star/~\cos~/\star~I \wedge f < F~\star/$
- 5 Si entrem un cop més al bucle, la funció de fita és estrictament positiva:  $I \wedge B \implies f > 0$

# Exemple: Exponenciació

```
// Pre: x = X \land y = Y \geq 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = XY
```

Invariant:

$$x = X \wedge y \geq 0 \wedge p \cdot X^y = X^Y$$

Fita: y

# Exemple: Exponenciació

```
// Pre: x = X \land y = Y \geq 0
int p = 1;
while (y > 0) {
   if (y % 2 == 0) { x = x*x; y = y/2; }
   else { p = p * x; y = y - 1; }
}
// Post: p = X^Y
```

Invariant:

$$y \geq 0 \wedge p \cdot x^y = X^Y$$

Fita: y

#### Una mica de notació

- Donat un vector v de talla n i dos entergs i, jamb  $0 \le i, j < n$ , v[i...j] denota el subvector entre les components i i j; si i > j llavors v[i...j] és un subvector buit
- Donada una llista L i dos iteradors it1 i it2 tals que it2 apunta a un element posterior a l'apuntat por it1 (o it2 == it1) llavors L[it1:it2) denota la subllista de L el primer element de la qual és l'apuntat per it1 i l'últim element és el predecessor de l'element apuntat per it2
- ullet  $L[:it) \equiv L[L. ext{begin}(),it)$
- ullet  $L[it:) \equiv L[it,L.end())$
- $\bullet$   $L[:) \equiv L$

### Exemple: Suma d'un vector

```
// Pre: cert
double suma(const vector<double>& v) {
   int i = 0;
   double s = 0;
   while (i < v.size()) {
        s += v[i];
        ++i;
   }
   return s;
}
// Post: el resultat es la suma de tots els elements de v</pre>
```

Invariant:

$$0 \leq i \leq v. exttt{size}() \land s = exttt{suma de } v[0..i-1]$$

• Fita: v.size() - i

### Exemple: Cerca un element en una llista

```
// Pre: cert
bool pertany(const list<double>& l, double x) {
    list<double>::const_iterator it = l.begin();
    bool trobat = false;
    while (it != l.end() and not trobat) {
        if (*it == x) trobat = true;
        ++it;
    }
    return trobat;
}
// Post: el resultat indica si x apareix en l
```

### Exemple: Cerca un element en una llista

```
// Pre: cert
bool pertany(const list<double>& l, double x) {
    list<double>::const_iterator it = l.begin();
    bool trobat = false;
    while (it != l.end() and not trobat) {
        if (*it == x) trobat = true;
        ++it;
    }
    return trobat;
}
// Post: el resultat indica si x apareix en l
```

Invariant:

```
it apunta a un element de l ó it = l.end(), i trobat = "x pertany a l[:it)"
```

• Funció de fita: nombre d'elements de la subllista l[it:)

### Exemple: variació de cerca lineal

```
// Pre: cert
// Post: retorna la posició en v d'un estudiant amb dni x,
// o bé -1 si cap estudiant de v té dni x
int posicio(int x, const vector<Estudiant>& v) {
   int i = 0;
  bool trobat = false:
   while (i < v.size() and not trobat) {</pre>
      if (v[i].consultar dni() == x) trobat = true;
      else ++i:
   if (trobat) return i;
   else return -1;
```

Invariant:

```
0 \leq i \leq v.\mathtt{size}()) \land x \not\in v[0..i-1] \land trobat = \text{``}i < v.\mathtt{size}() \land v[i] = x\text{''}
```

Funció de fita:

```
v.\mathtt{size}()-i-[\![trobat]\!], \qquad [\![P]\!]=1 \ \mathrm{si}\ P \ \mathrm{\acute{e}s} \ \mathrm{cert}, \ \mathrm{i}\ [\![P]\!]=0 \ \mathrm{si}\ P \ \mathrm{\acute{e}s} \ \mathrm{fals}
```

### Exemple: Suma d'una pila

Donada una pila d'enters, calcular-ne la suma dels elements:

```
// Pre: p = [a<sub>1</sub>,...,a<sub>n</sub>]
int suma(stack<int>& p) {
   int s = 0;
   while (not p.empty()) {
        s += p.top();
        p.pop();
   }
   return s;
}
// Post: suma(p) = a<sub>1</sub> + ··· + a<sub>n</sub> ∧ p = []
```

- Invariant:  $\exists i: 0 \leq i \leq n: p = [a_1, \dots, a_{n-i}] \land s = \sum_{k=n+1-i}^{n} a_k = a_n + a_{n-1} + \dots + a_{n-i+1}$  (\*)
- Funció de fita: alçada de p

```
(*) Quan i=n entendrem que p=[a_1,\ldots,a_{n-i}]=[]; de manera semblant quan i=0 llavors s=0 (el rang de sumació és buit)
```

### Exemple: Sumar k a una llista

Problema: donada una llista i un enter k, transformar-la en una altra resultant de sumar k a cada element de la llista original.

```
// Pre: l = [a<sub>1</sub>,...,a<sub>n</sub>]
void suma_k(list<int>& l, int k) {
    list<int>::iterator it;
    it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
// Post: l = [a<sub>1</sub> + k,...,a<sub>n</sub> + k]
```

- Invariant:  $\exists i: 1 \leq i \leq n+1: l[it:) = [a_i, \ldots, a_n] \land l[:it) = [a_1+k, \ldots, a_{i-1}+k]$  (\*)
- Funció de fita: nombre d'elements de *l[it:*)

```
(*) Quan i=n+1 entendrem que l=[a_i,\ldots,a_n]=[]; de manera semblant quan i=1 llavors laux=[a_1+k,\ldots,a_{i-1}+k]=[]
```

# Exemple: Revessar una llista

```
// Pre: l = [a<sub>1</sub>,...,a<sub>n</sub>]
void revessa(list<int>& l) {
    list<int> laux;
    while (not l.empty()) {
        laux.push_front(*(l.begin());
        l.pop_front();
    }
    // laux = [a<sub>n</sub>,...,a<sub>1</sub>]
    l = laux;
}
// Post: l = [a<sub>n</sub>,...,a<sub>1</sub>]
```

Invariant:

$$\exists i: 1 \leq i \leq n+1: l=[a_i,\ldots,a_n] \wedge laux=[a_{i-1},\ldots,a_1]$$

• Funció de fita: l.size()

Exercici: Directament sobre 1, evitant la llista auxiliar

### Exemple: cerca dicotòmica

```
// Pre: 0 < esq = E \land D = dre < v.size() \land esq < dre + 1
// \wedge v està ordenat creixentment.
// Post: oldsymbol{x} és a oldsymbol{v}[E..D] si i nomes si
// \qquad 0 < esq < v.size() \land v[esq] = x
int posicio (double x, const vector <double > & v,
              int esq, int dre) {
   while (esq < dre)
        int pos = (esq + dre)/2;
        if (v[pos] < x) esq = pos + 1;
        else dre = pos;
   return esq;
```

- Invariant:  $x \in v[E..D] \Leftrightarrow x \in v[esq..dre] \wedge ...$
- Fita: dre esq. Millor encara:  $f = \log_2(dre esq + 1)$

### Exemple: comptar nombre d'elements diferents

```
// Pre: cert
// Post: retorna el nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
   int n = 0;
   int i = 0;
   while (i < v.size()) {
       int j = i-1;
       while (j \ge 0 \text{ and } v[j] != v[i]) --j;
       if (i < 0) ++n;
       ++i;
   return n;
```

# Exemple: comptar nombre d'elements diferents

```
// n = N \land i < v.size()
int j = i-1;
while (j >= 0 and v[j] != v[i]) --j;
if (j < 0) ++n;
// n = N si v[i] \notin v[0..i-1] \land
// n = N+1 si v[i] \in v[0..i-1]
```

Invariant (del bucle intern sobre j):

$$v[i] 
ot\in v[j+1..i-1] \land j \ge -1$$

Fita: j + 1

### Exemple: comptar nombre d'elements diferents

```
// Pre: cert
// Post: diferents(v) = nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
  int n = 0;
  int i = 0;
  while (i < v.size()) {
     int j = i-1; ...; if (j < 0) ++n;
    // n = nombre d'elements diferents a v[0..i]
    ++i;
  return n;
```

Invariant (del bucle extern sobre i):

```
0 \leq i \leq v.	exttt{size}() \land n = 	exttt{nombre d'elements diferents a } v[0..i-1]
```

Fita: n − i

# Exemple: comptar nombre d'elements diferents (2)

#### Amb una funció separada:

```
// Pre: [a..b] \subset [0..v.size()-1]
// Post: retorna cert sii x \in v[a..b]
template <typename T>
bool apareix(const T& x, const vector<T>& v, int a, int b);
// Pre: cert
// Post: retorna el nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
   int n = 0;
   int i = 0:
   while (i < v.size()) {
       if (not apareix(v[i], v, 0, i-1)) ++n;
       ++i;
   return n;
```

# Exemple: comptar nombre d'elements diferents (2)

#### Invariant:

```
0 \leq i \leq v.	exttt{size}() \land n = 	exttt{nombre d'elements diferents en } v[0..i-1]
```

- Es fa servir l'especificació d'apareix per verificar
- Podem verificar independentment la correcció de diferents i d'apareix > Modularitat!

### Invariants "gràfics"

Sovint podem donar una representació gràfica esquemàtica d'un invariant (i en general d'una asserció), molt més intuitiva i senzilla d'entendre.

Exemple: Donat un vector v d'enters, escriu un procediment que reorganitzi els seus continguts de manera que els elements parells apareguin abans que els elements senars.

```
// Pre: cert
// Post: ???
void reorganitza_parells_senars(vector<int>& v);
```

```
// Pre: v = V
// Post: ???
void reorganitza_parells_senars(vector<int>& v);
```

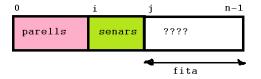
Postcondició formal: v és una permutació de V,
 n = v.size() > 0 i

$$\exists i : 0 \leq i < n : \Big( orall j : 0 \leq j < i : v[j] \mod 2 = 0$$
 
$$\land \forall j : i \leq j < n : v[j] \mod 2 = 1 \Big)$$

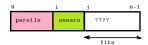
Postcondició "gràfica":



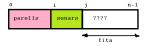
```
// Pre: v = V
// Post: ???
void reorganitza_parells_senars(vector<int>& v);
```



```
// Pre: v = V
// Post: ...
void reorganitza_parells_senars(vector<int>& v) {
   int i = 0; int j = 0;
   while (j < v.size()) {
     if (v[j] % 2 == 0) {
        swap(v[i], v[j]); ++i;
     }
     ++j;
}</pre>
```



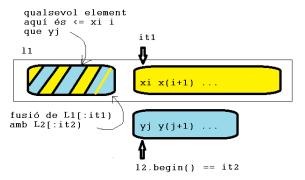
```
// Pre: v = V
// Post: ...
void reorganitza_parells_senars(vector<int>& v) {
  int i = 0;
  for (int j = 0; j < v.size(); ++j)
    if (v[j] % 2 == 0) {
      swap(v[i], v[j]); ++i;
    }
}</pre>
```



```
// Pre: l1 = L1 = [x_1, \dots, x_m] \land l2 = L2 = [y_1, \dots, y_n] \land // x_1 \le x_2 \le \dots \le x_m \land y_1 \le y_2 \le \dots \le y_n // Post: l1 = [z_1, \dots, z_{m+n}] \land l2 = [] // \land \{z_1, \dots, z_{m+n}\} = \{x_1, \dots, x_m\} \cup \{y_1, \dots, y_n\} // \land z_1 \le z_2 \le \dots \le z_{m+n} template <typename T> void fusionar(list<T>& 11, list<T>& 12);
```

N.B. Suposem que hi ha un ordre  $\leq$  definit entre elements de tipus  ${\scriptscriptstyle \mathbb{T}}$ 

- Fita: min(mida de l1[it1:), l2.size())
- Invariant:



```
template <typename T>
void fusionar(list<T>& 11, list<T>& 12) {
 list<T>::iterator it1 = l1.begin();
 list<T>::iterator it2 = 12.begin();
 while (it1 != 11.end() and it2 != 12.end()) {
    if (*it1 <= *it2) ++it1;
   else {
      11.insert(it1,*it2);
      it2 = 12.erase(it2);
```

```
template <typename T>
void fusionar(list<T>& l1, list<T>& l2) {
   list<T>::iterator it1 = l1.begin();
   list<T>::iterator it2 = l2.begin();
   while (it1 != l1.end() and it2 != l2.end()) { ... }
   // it1 = l1.end() i l1[:it1) és la fusió de L1 amb L2[:it2)
   // o it2 = l2.end() i l1[:it1) és la fusió de L1[:it1] amb L2

// it1 = l1.end() i l1 és la fusió de L1 amb L2[:it2)
   // o it2 = l2.end() i l1 és la fusió de L1 amb L2[:it2)
   // o it2 = l2.end() i l1 és la fusió de L1 amb L2
   l1.splice(l1.end(), l2);
}
```

## Part IV

# Disseny Iteratiu: Verificació i Derivació

- 17 Conjectesa de programes
- 18) Estats i assercions
- 19 Correctesa de programes iteratius
- 20 Disseny inductiu

### Disseny inductiu o derivació

Invertim el procés: de la "justificació" a l'algorisme

Donades Pre i Post, proposar:

- un invariant que les generalitzi les dues
- deduim les inicialitzacions que, amb la Pre, estableixin l'invariant
- un cos del bucle que mantingui l'invariant
- una condició del bucle que, negada, i junt amb l'invariant impliqui la Post

```
// Pre: x=X\geq 0 ?????
// Post: a=\lfloor \sqrt{X}\rfloor (part entera per defecte de l'arrel quadrada de X)
```

```
// Pre: x=X\geq 0 ????  
// Post: a=\lfloor \sqrt{X}\rfloor (part entera per defecte de l'arrel quadrada de X)
```

Post: 
$$a^2 \le X < (a+1)^2$$
 Invariant:  $I = (x = X \ge 0) \land (a^2 \le x < b^2)$ 

```
// Pre: x=X\geq 0 ????  
// Post: a=\lfloor \sqrt{X}\rfloor (part entera per defecte de l'arrel quadrada de X)
```

```
Post: a^2 \le X < (a+1)^2 Invariant: I = (x = X \ge 0) \land (a^2 \le x < b^2)
```

```
int a = ?;
int b = ?;
while (B) {
   int c = (a+b)/2;
   if (?) a = c;
   else b = c;
}
```

```
int a = ?;
int b = ?;
while (B) {
   int c = (a+b)/2;
   if (?) a = c;
   else b = c;
}
```

- $I \wedge b \leq a+1 \implies a^2 \leq x < b^2 = (a+1)^2$ , és a dir,  $a = \lfloor \sqrt{x} \rfloor$  si  $b \leq a+1$  (ja que  $a \leq b$  sempre tindrem b = a+1); per tant la condició del bucle ha de ser la negació: b > a+1
- Com  $0 \le x = X < (X+1)^2$ , fent a = 0; ib = x+1; establim l'invariant
- Si entrem al bucle tindrem a < c < b, i si  $x < c^2$  llavors fent b = c; es torna a satisfer l'invariant; de manera similar, si  $c^2 \le x$  llavors fer a = c; reestablirà l'invariant.

```
int a = 0;
int b = x+1;
while (b > a+1) {
   int c = (a+b)/2;
   if (c * c <= x) a = c;
   else b = c;
}</pre>
```

El nostre algorisme és el conegut mètode de la bisecció per a trobar arrels de funcions continues

Donat un vector v, comptar quantes parelles (v[i], v[i+1]) conté tals que v[i] < v[i+1].

```
int parelles_ordenades(const vector<int>& v);
```

#### Invariant:

$$I=v. exttt{size}()=0 \lor \left(1 \le i \le v. exttt{size}() \land 
ight.$$
  $p= exttt{nombre}$  de parelles ordenades en  $v[0..i-1]
ight)$ 

- La variable i recorre el vector
- La variable p compta les parelles ordenades vistes fins al moment durant el recorregut

1. Com establim l'invariant al principi?

```
int p = 0;
int i = 1; // v[0..0] no conté cap parella
```

2. Quan acabem? I acabem satisfent la Post? La darrera parella que cal comprovar és (v[n-2],v[n-1]), amb n=v.size(). Si la nostra condició de sortida és i=n, hem provat totes les parelles (v[j-1],v[j]) amb j< n, inclós el cas j=n-1, que és la (v[n-2],v[n-1]) i ja hem acabat Podem posar de condició del bucle i< v.size(), que cobreix bé els casos n=0 i n=1, i que implica sortir quan i=v.size()

3. Com avancem mantenim l'invariant? Volem avançar fent i=i+1. Posem que ho fem com a darrera instrucció del cos del bucle. Per tant just abans d'incrementar s'hauria de complir que hem provat totes les parelles (v[j-1],v[j]) amb  $j \leq i$ . La que falta doncs és la parella amb j=i, que és (v[i-1],v[i])

```
// I \( i < v.size() \)

if (v[i-1] < v[i]) p = p + 1;

i = i + 1;

// I
```

```
int parelles_ordenades(const vector<int>& v) {
   int p = 0;
   for (int i = 1; i < v.size(); ++i)
        // Inv: v.size() = 0 ó
        // (p = nombre de parelles ordenades en v[0..i-1]
        // i 1 \leq i \leq v.size())
        // Fita: 0 si v.size() = 0, v.size() - i altrament
        if (v[i-1] < v[i]) ++p;
   return p;
}</pre>
```

#### Ordenació

- Ordenar un vector v: Deixar-lo de manera que "per a tot i,  $0 \le i < v.size() 1$ ,  $v[i] \le v[i+1]$
- Invariant:

"
$$v[0..j]$$
 està ordenat" . . .

Fixem-nos que quan j = v.size() - 1 ja tenim tot el vector ordenat.

#### Ordenació

- ① Ordenar un vector v: Deixar-lo de manera que "per a tot i,  $0 \le i < v.size() 1$ ,  $v[i] \le v[i+1]$
- Invariant:

"
$$v[0..j]$$
 està ordenat" . . .

Fixem-nos que quan j = v.size() - 1 ja tenim tot el vector ordenat.

"i tots els elements de v[0..j] són més petits o iguals que els de v[j+1..v.size()-1]"

#### Ordenació

"v[0..j] està ordenat i tots els elements de v[0..j] són més petits o iguals que els de v[j+1..v.size()-1]"

#### Si incrementem *j*:

- v[0..j] segueix ordenat! Per què?
- Però no és cert que "v[0..j] és més petit que v[j+1..v.size()-1]"
- Només és cert si v[j+1] era un element mínim de v[j+1...v.size()-1]
- Que hem de fer?
  - Buscar un valor mínim de v[j + 1...v.size() 1]
  - Intercanviar-lo amb v[j+1]
  - Incrementant j es reestableix l'invaraint

Aquest és l'algorisme d'ordenació per selecció.

Exercici: Si no posem la segona part de l'invariant, deriveu la ordenació per inserció

```
// Pre: v.\text{size}() > 0

// Post: el resultat és i tal que v[0] + ... + v[i] és màxima

// i - 1 \le i < v.\text{size}()

int psm(const vector<double>& v);
```

```
// Post: el resultat és i tal que v[0]+...+v[i] és màxima // i -1 \leq i < v.	ext{size}()
```

Què vol dir "és màxima"? Sigui  $n=v.\mathtt{size}()$  i definim  $S_i=\sum_{k=0}^i v[k]$ . Per conveni,  $S_{-1}=0$ . Llavors estem dient que el resultat és el valor  $i, -1 \le i < n$ , tal que

$$S_i = \max\{S_k \mid -1 \le k < n\}$$

Això suggereix que la nostra solució faci un bucle sobre j amb l'invariant:

```
// Inv: S_i = \max\{S_k \mid -1 \leq k < j\} \land -1 \leq i < j < n
```

```
// Inv: -1 \le i < j < v.{
m size}(),
// v[0] + \cdots + v[i] \ge v[0] + \cdots + v[k] per a tot k \in [-1..j-1],
// sum = v[0] + \cdots + v[j-1],
// sumi = v[0] + \cdots + v[i]
```

```
// Pre: v.size() > 0
int psm(const vector<double>& v) {
   int i = -1; int j = 0;
   double sum = 0; double sumi = 0;
   while (j < v.size()) {</pre>
       sum += v[j];
       if (sum > sumi) {
          sumi = sum;
          i = i:
       ++ 1;
   return i;
// Post: el resultat és i tal que v[0] + ... + v[i] és màxima
// i -1 \le i < v.size()
```

#### Percentatge d'estudiants presentats (amb nota)

- Donat un conjunt d'estudiants (Cjt\_estudiants) retornem el percentatge d'estudiants presentats (amb nota) del vector
- Especificació:

```
// Pre: C conté almenys un estudiant // Post: el resultat es el percentatge de presentats de C double presentats(const Cjt_estudiants& C);
```

#### Percentatge d'estudiants presentats (amb nota)

- Per saber el % de presentats calculem primer el nombre d'estudiants amb nota
- Tindrem una potscondició P' després del bucle: "npres és el nombre d'estudiants presentants de C".
- Un cop tenim P<sup>I</sup>, és immediat obtenir la postcondició de la funció Post.
- Per obtenir P' recorrerem el conjunt C comptant els estudiants amb nota
- Invariant:

```
// Inv: 1 \le i \le C.mida() +1
// i npres = nombre d'estudiants amb nota entre els primers i-1 estudiants en ordre creixent // de DNI
```

#### Percentatge d'estudiants presentats (amb nota)

```
// Pre: C conté almenvs un estudiant
double presentats(const Cjt_estudiants& C) {
  int npres = 0;
  // Inv: 1 < i < |C|,
  // npres = nombre d'estudiants amb nota entre
  // els i-1 primers
  for (int i = 1; i <= C.mida(); ++i) {</pre>
     if (C.consultar iessim(i).te nota())
        ++npres;
     // npres = nombre d'estudiants amb nota entre els
     // i primers
  //\ P'\colon npres és el nombre d'estudiants presentants de C
  return double (npres) / C.mida () *100;
// Post: el resultat és el percentatge d'estudiants
// presentats de C
```

Donat un vector d'estudiants, modificar-lo arrodonint-ne les notes a la dècima més propera (es pot fer com a acció o com a funció).

```
// Pre: cert
// Post: vest té les notes dels estudiants arrodonides
// a la dècima més propera del seu valor inicial
void arrodonir_notes(vector<Estudiant>& vest);
```

Farem un recorregut pels elements del vector i suposem que disposem de la funció:

```
// Pre: cert
// Post: retorna el valor més proper a x amb un sol decimal
double arrodoniment(double x) {
  return 0.1*round(x*10);
}
```

Invariant: igual que la postcondició però aplicada només a la part tractada del vector

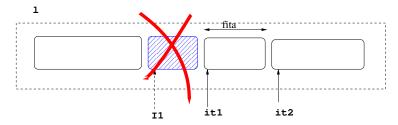
```
// Inv: vest[0..i-1] té les notes dels estudiants arrodonides // a la dècima més propera del seu valor inicial, // 0 \le i \le vest.size()
```

- Quan i = vest.size(), Inv  $\implies$  Post
- Si cada iteració incrementa i, per mantenir l'invariant abans hem d'arrodonir vest[i]

```
// Pre: cert
void arrodonir notes(vector<Estudiant> &vest) {
  int n = vest.size();
  int i = 0;
 // Inv: ...
 while (i < n) {
    if (vest[i].te_nota()) {
      double aux = arrodoniment(vest[i].consultar nota());
      vest[i].modificar_nota(aux);
    ++i;
// Post: vest té les notes dels estudiants arrodonides
// a la dècima més propera del seu valor inicial
```

```
// Pre: it1 = I_1 i it2 = I_2 apuntem elements de la llista L i // it2 apunta a un element igual o posterior a l'element // apuntat per it1; l = L // Post: La llista l conté tots els elements d'L, excepte // els que hi havia entre l'element originalment apuntat per it1 i // el predecessor de l'element originalment apuntat per it2, i.e., // l = L[:I_1) \cdot L[I_2:); '.' denota la concatenació // de llistes template <class T> void elimina_subllista(list<T>& l, list<T>::iterator it1, list<T>::iterator it2);
```

Com a invariant proposem que la subllista entre el valor original  $I_1$  d'it1 i el predecessor de l'element al qual apunta it1 ha sigut eliminada; quan it1 = it2 tindrem la postcondició:



#### Invariant "formal":

```
// Inv: it1 i it2=I_2 apuntem elements de la llista L i it2 apunta a un element igual o posterior a l'element apuntat per it1, l=L[:I_1)\cdot L[it1:)
```

- L'invariant és compleix des del primer moment
- Si it1 = it2, llavors l'invariant implica la postcondició
- Si l'invariant és cert i  $it1 \neq it2$ , llavors eliminant l'element apuntat per it1 i avançant it1 reestablim l'invariant
- La funció de fita és la talla de L[it1: it2); la precondició (i l'invariant) garanteixen que és ≥ 0, i amb cada iteració disminuirà en una unitat

```
// Pre: it1 = I_1 i it2 = I_2 apuntem elements de la llista L i
// it2 apunta a un element iqual posterior a l'element apuntat
// per it1; l=L
template <class T>
void elimina subllista(list<T>& l,
         list<T>::iterator it1, list<T>::iterator it2) {
    while (it1 != it2)
      // Inv: it1 i it2 = I_2 apuntem elements de la
           llista L i it2 apunta a un element
      // iqual o posterior a l'element apuntat
      // per it1, l = L[:I_1) \cdot L[it1:]
      it1 = l.erase(it1):
// Post: La llista l conté tots els elements d'L, excepte
// els que hi havia entre I_1 i el predecessor de I_2, i.e.,
// l = L[: I_1) \cdot L[I_2:)
```

## Part V

# Disseny Recursiu

- 21 Recursió, definicions recursives i inducció
- 22 Procipis de disseny recursiu
- 23 Immersió de funcions: Afebliment de la postcondició
- 24 Immersió de funcions: Enfortiment de la precondició (\*)
- 25 Recursivitat lineal final i algorismes iteratius (\*\*)

## Alguns conceptes bàsics en disseny recursiu

- Recursió és inducció. Comenceu amb una definició recursiva
- Si no podeu fer recursió, proveu d'afegir més paràmetres (funció d'immersió)
- Si es repeteixen càlculs, afegiu paràmetres per recordar-los (immersió d'eficiència)

### Alçària d'una pila

Vam veure versions iterativa i recursiva:

```
int alcaria_iter(stack<int>& p) {
    int n = 0;
    while (not p.empty()) {
      ++n;
       p.pop();
    return n;
int alcaria_rec(stack<int>& p) {
    if (p.empty()) return 0;
    else {
       p.pop();
        return 1 + alcaria_rec(p);
```

#### D'on hem tret la versió recursiva??

L'alçaria de la pila  $P = [e_1, e_2, \dots, e_n]$  és n

D'on hem tret la versió recursiva??

L'alçaria de la pila 
$$P = [e_1, e_2, \dots, e_n]$$
 és  $n$ 

Lema: això és equivalent a

- Si P és buida, alçària(P) = 0
- Altrament, alçària(P) = 1 + alçària(desapilar(P))

#### D'on hem tret la versió recursiva??

En la definició recursiva, desapilar(P) és una funció abstracta que ens retorna la pila resultant de desapilar el cim de la pila P:

```
// Pre: p=P no és una pila buida p.pop(); // Post: p=\operatorname{desapilar}(P)
```

El mètode pop modifica la pila sobre la qual s'aplica i retorna void.

```
n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1
```

```
/* Pre: n > 0 */
/* Post: retorna n! */
int fact(int n) {
 int f = 1;
  // n = N > 0
   while (n > 0) {
    // Inv: f = N!/n! \wedge n > 0
    // Fita: n
    f = f * n;
    --n;
   return f;
```

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdot \cdot 2 \cdot 1$$

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

#### Lema:

- 0! = 1
- per a tot n > 0,  $n! = n \cdot (n-1)!$

```
n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1
```

#### Lema:

- 0! = 1
- per a tot n > 0,  $n! = n \cdot (n-1)!$

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Definició amb "..." suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements "més petits"

Definició amb "..." suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements "més petits"

Exemple: descomposem una pila no buida  $P=[a_1,\ldots,a_n]$  en  $\operatorname{cim}(P)=a_n$  (que podem obtenir amb  $\operatorname{P.top}()$ ) i desapilar $(P)=[a_1,\ldots,a_{n-1}]$  que obtindrem mitjançant  $\operatorname{P.pop}()$ .

Definició amb "..." suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements "més petits"

Exemple: descomposem una pila no buida  $P = [a_1, \ldots, a_n]$  en  $\operatorname{cim}(P) = a_n$  (que podem obtenir amb P.top()) i desapilar $(P) = [a_1, \ldots, a_{n-1}]$  que obtindrem mitjançant P.pop().

Exemple: per el càlcul de  $x^y$  es pot fer la descomposició  $x^y=x\cdot x^{y-1}$  o  $x^y=(x^2)^{y/2}$  si y és par.

Definició amb "..." suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements "més petits"

Exemple: descomposem una pila no buida  $P = [a_1, \ldots, a_n]$  en  $\operatorname{cim}(P) = a_n$  (que podem obtenir amb P.top()) i desapilar $(P) = [a_1, \ldots, a_{n-1}]$  que obtindrem mitjançant P.pop().

Exemple: per el càlcul de  $x^y$  es pot fer la descomposició  $x^y=x\cdot x^{y-1}$  o  $x^y=(x^2)^{y/2}$  si y és par.

Sovint fem la transformació inconscientment. Si cal formalitzar-la, necessitem inducció

## Suma dels elements d'una pila

Si 
$$P = [a_1, \dots, a_n]$$
, llavors suma $(P) = a_1 + \dots + a_n$   
Inductivament:

$$suma(P) = egin{cases} 0 & ext{si } P ext{ \'es buida}, \\ cim(P) + suma(desapilar(P)) & ext{altrament}. \end{cases}$$

### Cerca en una pila

Donada una pila p i un element x, dir si x apareix en pVersió recursiva:

- si p és buida, x no apareix a p
- altrament, si P no és buida . . .

```
x \in p \Leftrightarrow x = \operatorname{cim}(P) \lor x \in \operatorname{desapilar}(p)
```

### Cerca en una pila

```
template <class T>
bool cerca(stack<T>& p, const T& x) {
  if (p.empty())
    return false;
  else if (x == p.top() return true;
  else {
    p.pop();
    return cerca(p, x);
  }
}
```

Donades dues piles  $p=[p_1,\ldots,p_n]$  i  $q=[q_1,\ldots,q_m],$  dir si són iguals

 $\implies m=n$  i per a cada posició  $i, 1 \leq i \leq n, p_i=q_i$ .

Donades dues piles  $p=[p_1,\ldots,p_n]$  i  $q=[q_1,\ldots,q_m]$ , dir si són iguals

```
\implies m=n i per a cada posició i,\,1\leq i\leq n,\,p_i=q_i.
```

#### Versió recursiva:

- si p i q són buides, són iguals
- si p és buida i q no, o a l'inrevés, llavors són diferents
- si p i q no són buides, ...
  - $sicim(p) \neq cim(q)$  (p.top() != q.top(), les piles són diferents;
  - si cim(p) = cim(q) llavors p = q si i només si desapilar(p) = desapilar(q)

```
/* Pre: p = P, q = Q */
/* Post: Retorna cert si i només si P = Q */
bool piles_iguals(stack<int>& p, stack<int>& q) {
    if (p.empty() and q.empty()) return true;
    else if (p.empty() or q.empty()) return false;
    else if (p.top() != q.top()) return false;
    else {
        p.pop(); q.pop();
        return piles_iguals(p,q);
    }
}
```

### Igualtat de piles (versió alternativa)

```
/* Pre: p = P, q = Q */
/* Post: Retorna cert si i només si P = Q */
bool piles_iguals(stack<int>& p, stack<int>& q) {
    if (p.empty() or q.empty())
        return p.empty() and q.empty();
    if (p.top() != q.top())
        return false;
    p.pop(); q.pop();
    return piles_iguals(p,q);
}
```

- La versió iterativa queda com a exercici.
- Cal aplicar l'esquema de cerca seqüèncial, **no** el de recorregut!. Podem concloure que  $P \neq Q$  si trobem  $p_i \neq q_i$  o no buidem simultàniament de les dues piles (una és buida i l'altra no)

- La versió iterativa queda com a exercici.
- Cal aplicar l'esquema de cerca seqüèncial, **no** el de recorregut!. Podem concloure que  $P \neq Q$  si trobem  $p_i \neq q_i$  o no buidem simultàniament de les dues piles (una és buida i l'altra no)
- Observació: és temptador comprovar abans que res si alcaria (p) != alcaria (q)
- Seria eficient (i convenient), però només si tenim una operació p.size() que no recorre la pila! Pero seria ineficient si hem de calcular el nombre d'elements de la pila recorrent (i destruint) la pila

## Part V

# Disseny Recursiu

- (21) Recusió, definicions recursives i inducció
- 22 Principis de disseny recursiu
- 23 Immersió de funcions: Afebliment de la postcondició
- 24 Immersió de funcions: Enfortiment de la precondició (\*)
- 25 Recursivitat lineal final i algorismes iteratius (\*\*)

### Principis de disseny recursiu

#### Volem implementar recursivament una funció

```
// Pre: propietat satisfeta per x
// Post: la funció retorna un valor F(x)
tipus_sortida F(tipus_entrada x);
```

#### o un procediment

```
// Pre: propietat satisfeta per x=X // Post: res compleix una certa propietat en termes d'X void F(T1 x, T2\& res);
```

### Principis de disseny recursiu

N.B. x i res poden ser més d'un paràmetre; en el cas dels procediments podem tenir paràmetres de entrada/sortida:

```
// Pre: propietat satisfeta per x=X
// Post: x=X' compleix una certa propietat en termes
// del seu valor original X
void F(T1\& x)
```

## Principis de disseny recursiu

#### Cal identificar:

- Un o més casos base: Valors de paràmetres en què podem satisfer la Post amb càlculs directes
- Un o més casos recursius: Valors de paràmetres en què podem satisfer la Post si tinguessim el resultat per a alguns paràmetres x' "més petits" que x

### Estratègia

- 1 Triar una funció de "mida" |x| dels paràmetres x tal que
  - $|x| \le 0 \implies$  som en un cas base
  - les crides recursives es fan amb paràmetres  $x^\prime$  amb  $|x^\prime| < |x|$
  - ha de ser sempre un enter: per tot  $x, |x| \in \mathbb{Z}$

Fonament: tota seqüència decreixent d'enters no negatius és finita

Transformar la definició rebuda del que volem calcular en una definició recursiva (si no ho és d'entrada)

## Correctesa d'un algorisme recursiu

A demostrar: Amb tot valor x dels paràmetres que satisfaci Pre,

- l'algorisme acaba nombre finit de crides recursives
- i acaba satisfent Post(x)

#### Acabament: nombre finit de crides recursives

Fonament:

Tota seqüència decreixent de nombres enters no negatius és finita

#### Acabament: nombre finit de crides recursives

#### Fonament:

Tota seqüència decreixent de nombres enters no negatius és finita

#### Formalització:

- Triem una funció de mida | · | dels paràmetres que sempre té valor enter
- Demostrem: Si  $|x| \le 0$  l'algorisme tracta x amb un cas base  $\implies$  cap crida recursiva
- Demostrem: Cada crida recursiva fa decrèixer la mida dels paràmetres, i.e., si la funció  $\mathbb{F}$  amb paràmetre x fa la crida recursiva  $\mathbb{F}(x')$  llavors |x'| < |x|

N.B. Noteu la similitud entre les propietats de la funció de mida i les de la funció de cota d'una iteració

## Correctesa d'un algorisme recursiu

- A demostrar: Si el paràmetre x satisfà la precondició llavors el resultat satisfà la postcondició (una funció d'x)
- Quan x és un cas base ( $|x| \le 0$ , no hi ha recursió): es demostra directament aplicant les tècniques de les lliçons anteriors

### Correctesa d'un algorisme recursiu

 Si x no és un cas base (|x| > 0), apliquem l'hipòtesi d'inducció:

```
H.I. = "Si x' compleix la precondició (Pre(x') és cert) i |x'| < |x| llavors l'algorisme acaba en temps finit i es compleix la postcondició (Post(x'))"
```

### Correctesa d'un algorisme recursiu

 Si x no és un cas base (|x| > 0), apliquem l'hipòtesi d'inducció:

```
H.I. = "Si x' compleix la precondició (Pre(x') és cert) i |x'| < |x| llavors l'algorisme acaba en temps finit i es compleix la postcondició (Post(x'))"
```

• Hem de demostrar que qualsevol crida recursiva  $\mathbb{F}(x')$  quan x no és un cas base (|x| > 0) compleix: 1)  $\mathbb{Pre}(x')$ ; 2) |x'| < |x|. Podem aplicar llavors l'H.I.

### Correctesa d'un algorisme recursiu

 Si x no és un cas base (|x| > 0), apliquem l'hipòtesi d'inducció:

```
H.I. = "Si x' compleix la precondició (Pre(x') és cert) i |x'| < |x| llavors l'algorisme acaba en temps finit i es compleix la postcondició (Post(x'))"
```

- Hem de demostrar que qualsevol crida recursiva  $\mathbb{F}(x')$  quan x no és un cas base (|x| > 0) compleix: 1)  $\mathbb{Pre}(x')$ ; 2) |x'| < |x|. Podem aplicar llavors l'H.I.
- Aplicant l'H.I. deduïm que després d'una crida recursiva Post(x'); cal demostrar que l'estat al qual s'arriva just després o fent alguns càlculs addicionals satisfà Post(x)

### Exemples

- Exponenciació ràpida
- Factorial
- Nombres binomials
- Ordenació per fusió (Mergesort) en un vector

- Revessar una Ilista
- Cercar un element en una cua
- Sumar k als elements d'un arbre

#### **Factorial**

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
   if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Recordem: n! = 1 si n = 0;  $n! = n \cdot (n - 1)!$  si n > 0

• Acabament: mida = |n| = n. Sempre enter, |n| = 0 és cas base, amb |n| = n > 0 es fa la crida fact (n-1), |n-1| = n - 1 < |n| = n

#### **Factorial**

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Recordem: n! = 1 si n = 0;  $n! = n \cdot (n - 1)!$  si n > 0

- Acabament: mida = |n| = n. Sempre enter, |n| = 0 és cas base, amb |n| = n > 0 es fa la crida fact (n-1), |n-1| = n 1 < |n| = n
- Correcció: si n=0 retornem 1 (0! = 1); si n>0 es fa crida amb fact (n-1), llavors  $n-1\geq 0$ , |n-1|<|n| i podem aplicar H.I.

### **Factorial**

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Recordem: n! = 1 si n = 0;  $n! = n \cdot (n - 1)!$  si n > 0

- Acabament: mida = |n| = n. Sempre enter, |n| = 0 és cas base, amb |n| = n > 0 es fa la crida fact (n-1), |n-1| = n 1 < |n| = n
- Correcció: si n=0 retornem 1 (0! = 1); si n>0 es fa crida amb fact (n-1), llavors  $n-1\geq 0$ , |n-1|<|n| i podem aplicar H.I.
- Correcció: H.I.  $\implies$  fact (n-1) = (n-1)!, per tant fact (n) retorna  $n \cdot (n-1)! = n!$

```
// Pre: x > 0 \land y \ge 0

// Post: retorna x^y

int potencia(int x, int y);
```

#### Observem que

$$x^y = egin{cases} 1 & ext{si } y = 0 \ x \cdot (x^2)^\lambda & ext{si } y = 2\lambda + 1 > 0 ext{ és senar} \ (x^2)^\lambda & ext{si } y = 2\lambda \geq 0 ext{ és parell} \end{cases}$$

```
int potencia(int x, int y) {
   if (y == 0) return 1;
   else if (y%2 == 1) return x*potencia(x*x,y/2);
   else return potencia(x*x,y/2);
}
```

• Acabament: podem agafar |y|=y, però també  $|y|=\lceil 1+\log_2(y)\rceil$ , ja que  $\lceil 1+\log_2(y/2)\rceil=\lceil \log_2(y)\rceil<\lceil 1+\log_2(y)\rceil$ . Sempre enter (per això fem servir  $\lceil \cdot \rceil$ ). Si  $|y|\leq 0$  estem en un cas base ( $\log_2 y \leq -1 \implies y \leq 1/2 \implies y \leq 0$ ). Si |y|>0 llavors no estem en un cas base,  $y\geq 1$  i |y/2|<|y|.

```
int potencia(int x, int y) {
   if (y == 0) return 1;
   else if (y%2 == 1) return x*potencia(x*x,y/2);
   else return potencia(x*x,y/2);
}
```

- Correcció: si y=0 llavors retornem  $x^0=1$ . Si y>0, es fa la crida recursiva potencia (x\*x, y/2). Com x>0, tenim  $x^2>0$ . I com y>0, llavors  $y/2\geq 0$ . A més |y/2|<|y|. Es pot aplicar H.I.
- Correcció: si  $y=2\lambda$  és parell, per H.I. potencia (x\*x, y/2) retorna  $(x^2)^{\lambda}=x^{2\lambda}=x^y$  i la funció retorna el resultat correcte. Si  $y=2\lambda+1$  és senar, per H.I. potencia (x\*x, y/2) retorna  $(x^2)^{\lambda}=x^{2\lambda}=x^{y-1}$ ; llavors la funció retorna  $x\cdot x^{y-1}=x^y$ , el resultat correcte.

Exercici: Demostreu la correcció de la següent implementació alternativa:

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else {
        int p = potencia(x, y/2);
        if (y%2 == 0) return p * p;
        else return x * p * p;
    }
}
```

```
// Pre: n \ge m \ge 0

// Post: retorna \binom{n}{m}

int binomial(int n, int m);
```

#### Recordem:

$$egin{pmatrix} n \ m \end{pmatrix} = rac{n!}{m! \cdot (n-m)!}$$

és el nombre de subconjunts de  $\{1, \ldots, n\}$  de mida m

## Nombres binomials: Disseny

Hi ha diverses definicions recursives equivalents, que porten a solucions d'eficiència i elegància diferents

Triant 
$$|(n, m)| = n$$
:

$$egin{aligned} \binom{n}{m} &= egin{cases} 1 & ext{si } n = m \ rac{n \cdot (n-1)!}{m! \cdot (n-m) \cdot (n-1-m)!} &= rac{n}{n-m} \cdot \binom{n-1}{m} & ext{si } n > m \end{cases}$$

Triant |(n, m)| = m:

$$egin{pmatrix} n \ m \end{pmatrix} = egin{cases} 1 & ext{si } m=0 \ rac{n!\cdot(n-m+1)}{m\cdot(m-1)!\cdot(n-m+1)\cdot(n-m)!} = rac{n-m+1}{m}inom{n}{m-1} & ext{si } m>0 \end{cases}$$

Triant |(n, m)| = m:

$$egin{pmatrix} n \ m \end{pmatrix} = egin{cases} 1 & ext{si } m = 0 \ rac{n! \cdot (n-m+1)}{m \cdot (m-1)! \cdot (n-m+1) \cdot (n-m)!} = rac{n-m+1}{m} inom{n}{m-1} & ext{si } m > 0 \end{cases}$$

O bé descomposem així:

$$egin{pmatrix} n \ m \end{pmatrix} = egin{cases} 1 & ext{si } m = 0 \ rac{n \cdot (n-1)!}{m \cdot (m-1)! \cdot ((n-1)-(m-1))!} = rac{n}{m} inom{n-1}{m-1} & ext{si } m > 0 \end{cases}$$

Triant |(n, m)| = m:

$$egin{pmatrix} n \ m \end{pmatrix} = egin{cases} 1 & ext{si } m = 0 \ rac{n! \cdot (n - m + 1)}{m \cdot (m - 1)! \cdot (n - m + 1) \cdot (n - m)!} = rac{n - m + 1}{m} inom{n}{m - 1} & ext{si } m > 0 \end{cases}$$

O bé descomposem així:

$$egin{pmatrix} n \ m \end{pmatrix} = egin{cases} 1 & ext{si } m = 0 \ rac{n \cdot (n-1)!}{m \cdot (m-1)! \cdot ((n-1) - (m-1))!} = rac{n}{m} inom{n-1}{m-1} & ext{si } m > 0 \end{cases}$$

(O bé fem servir el triangle de Tartaglia:

$$egin{pmatrix} n \ m \end{pmatrix} = egin{pmatrix} n-1 \ m \end{pmatrix} + egin{pmatrix} n-1 \ m-1 \end{pmatrix}$$

però la solució és més ineficient)

```
// Pre: n > m > 0
// Post: retorna (<sup>n</sup><sub>m</sub>)
int binomial (int n, int m) {
  if (m == 0) return 1;
  else return (binomial (n-1, m-1)/m) * n;
}
```

En aquesta implementació prenem cura de fer primer la divisió  $\binom{n-1}{m-1}$  és divisible entre m) i després fer el producte. Això pot evitar problemes de *overflow*.

Exercici: escriviu funcions recursives i raoneu la seva correctesa basant-se en les altres definicions recursives examinades.

## Mergesort

```
// Pre: 0 \le e \le d \le v.\text{size}() \land v = V
// Post: v[0..e-1] = V[0..e-1] \wedge v[d+1..n-1] = V[d+1..n-1] \wedge
         v[e..d] ordenat creixentment i és una permutació de V[e..d]
template <class T>
void mergesort(vector<T>& v, int e, int d) {
    if (e < d) {
        int m = (e + d)/2;
       mergesort (v, e, m);
       mergesort(v, m + 1, d);
       fusiona(v, e, m, d);
// Pre: 0 \le e \le m < d < v.size() \land v = V \land
/// v[e..m] i v[m+1..d] estàn ordenats creixentment
// Post: v[0..e-1] = V[0..e-1] \wedge v[d+1..n-1] = V[d+1..n-1] \wedge
        v[e..d] ordenat creixentment i és una permutació de V[e..d]
template <class T>
void fusiona(vector<T>& v, int e, int m, int d);
```

Mida = nombre d'elements a ordenar en el subvector - 1 = |d - e|.

#### Revessar una Ilista

```
// Pre: l = [a_1, \ldots, a_n] \land n \ge 0

// Post:l = [a_n, a_{n-1}, \ldots, a_1]

template <class T>

void revessar(list<T>& 1);
```

- Per a qualsevol estructura seqüencial  $l = [a_1, \ldots, a_n]$  no buida definim  $\text{head}(l) = [a_1], \text{tail}(l) = [a_2, \ldots, a_n];$  donades dues seqüències  $l_1$  i  $l_2$  denotem  $l_1 \cdot l_2$  la seqüència resultant de concatenar-les (head, tail i · són operacions abstractes).
- Per a tota Ilista  $l \neq []$ ,  $l = head(l) \cdot tail(l)$ .

Llavors podrem donar una definició de la funció abstracta revessar:

- revessar([]) = []
- Si  $l \neq []$  Ilavors revessar $(l) = \text{revessar}(\text{tail}) \cdot \text{head}(l)$

#### Revessar una Ilista

```
// Pre: l = [a_1, ..., a_n] \land n \ge 0

// Post: l = [a_n, a_{n-1}, ..., a_1]

template <class T>

void revessar(list<T>& 1) {

   if (not l.empty()) {

        T x = *(l.begin());

        l.erase(l.begin()); // = l.pop_front();

        revessar(l);

        l.insert(l.end(),x); // = l.push_back(x);

   }

}
```

Mida: l.size()

#### Cerca d'un element en una cua

```
// Pre: c = C
// Post: retorna cert si i només si x ∈ C
template <class T>
bool cerca(queue<T>& c, const T& x);
```

Definició no recursiva:  $cerca(e_1 \dots e_n, x) = (\exists i : e_i = x)$ 

Definició recursiva, amb mida c.size():

- cerca([], x) = false
- Si  $c = [a_1, \ldots, a_n] \neq []$  llavors

$$\operatorname{cerca}(c,x) = (x = \operatorname{front}(c)) \vee \operatorname{cerca}(\operatorname{tail}(c),x)$$

#### Cerca d'un element en una cua

```
// Pre: c = C
// Post: retorna cert si i només si x ∈ C
template <class T>
bool cerca(queue<T>& c, const T& x) {
    if (c.empty()) return false;
    else if (c.front() == x) return true;
    else {
        c.pop();
        return cerca(c, x);
    }
}
```

Mida: c.size()

#### Sumar k als elements d'un arbre

Definició no recursiva: la donada ("tots els nodes de l'arbre") Definició recursiva: amb mida

|a| = nombre de nodes de l'arbre:

- $suma(\Box, k) = \Box$
- ullet suma(plantar( $x, a_1, a_2$ ), k) = plantar(x + k, suma( $a_1, k$ ), suma( $a_2, k$ ))

#### Sumar k als elements d'un arbre

• Acabament: si |a| = 0 l'arbre és buit i estem en un cas base; amb |a| > 0 tenim un cas recursiu, i les crides a suma són amb a.left() i a.right(); els dos subarbres tenen mida inferior a la d'a

#### Sumar k als elements d'un arbre

- Correcció: si |a| = 0 la solució retorna un arbre buit. Si |a| > 0 la precondició de les dos crides recursives es compleix i els paràmetres són de mida inferior:  $|a| \cdot |a| \cdot |a|$ ,  $|a| \cdot |a| \cdot |a|$ . L'H.I. es pot aplicar.
- Correcció: directament de la definició (i aplicació de l'H.I.), la funció retorna el resultat correcte.

## Part V

# Disseny Recursiu

- (21) Regissió, definicions recursives i inducció
- (22) Procipis de disseny recursiu
- 23 Immersió de funcions: Afebliment de la postcondició
- 24 Immersió de funcions: Enfortiment de la precondició (\*)
- 25 Recursivitat lineal final i algorismes iteratius (\*\*)

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
// estudiant amb DNI = x
bool cerca(const vector<Estudiant> &v, int x);
```

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
// estudiant amb DNI = x
bool cerca(const vector<Estudiant> &v, int x);
```

Problema: Què fem decrèixer? No podem fer més petit el vector!

Creem una còpia del vector de mida v.size() - 1? Molt ineficient!

#### Plantegem:

```
// Pre: x és un DNI vàlid i 0 \le j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
// estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j);
```

#### Plantegem:

```
// Pre: \boldsymbol{x} és un DNI vàlid i 0 \leq j < v.\text{size}()
// Post: retorna cert si i només si v[0..j] conté al menys un
// estudiant amb DNI = \boldsymbol{x}
bool i_cerca(const vector<Estudiant>& v, int x, int j);
```

#### Buscar en tot el vector és

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
// estudiant amb DNI = x i v no és buit
bool cerca(const vector<Estudiant> &v, int x) {
  return i_cerca(v, x, v.size()-1);
}
```

#### Plantegem:

```
// Pre: x és un DNI vàlid i 0 \le j < v.\text{size}()

// Post: retorna cert si i només si v[0..j] conté al menys un

// estudiant amb DNI = x

bool i_cerca(const vector<Estudiant>& v, int x, int j);
```

#### Buscar en tot el vector és

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
// estudiant amb DNI = x i v no és buit
bool cerca(const vector<Estudiant> &v, int x) {
  return i_cerca(v, x, v.size()-1);
}
```

i ara podem fer créixer o decrèixer j!

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
// estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
   if (j == 0)
      return v[0].consultar_DNI() == x;
   else if (v[j].consultar_DNI() == x)
      return true;
   else
      return i_cerca(v,x,j-1);
}
```

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
// estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
   if (j == 0)
      return v[0].consultar_DNI() == x;
   else if (v[j].consultar_DNI() == x)
      return true;
   else
      return i_cerca(v,x,j-1);
}
```

Correctesa: Inducció sobre j: Si v[j] conté un estudiant amb DNI = x llavors v[0...j] conté un estudiant amb DNI = x. En cas contrari, si  $\operatorname{cerca}(v, x, j - 1)$  retorna cert llavors v[0...j - 1], i pert tant v[0...j] conté un estudiant amb DNI = x.

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
// estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
   if (j == 0)
      return v[0].consultar_DNI() == x;
   else if (v[j].consultar_DNI() == x)
      return true;
   else
      return i_cerca(v,x,j-1);
}
```

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
// estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
   if (j == 0)
      return v[0].consultar_DNI() == x;
   else if (v[j].consultar_DNI() == x)
      return true;
   else
      return i_cerca(v,x,j-1);
}
```

Correctesa: Inducció sobre *j*:

Però si cerca(v,x,j-1) retorna fals, llavors v[0..j-1] no conté un estudiant amb DNI=x i v[0..j] tampoc.

Alternativa: posar " $-1 \le j$ " en la Pre (v[0..-1] denota un subvector buit)

```
// Pre: x és un DNI vàlid i -1 \le j < v.\text{size}()
// Post: retorna cert si i només si v[0..j] conté al menys un
// estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
    if (j < 0) return false;
    if (v[j].consultar_DNI() == x) return true;
    return i_cerca(v,x,j-1);
}
```

Motiu: codi més compacte, funciona inclús si el vector v.size() = 0.

## Funció d'immersió: funció auxiliar

La funció original crida la funció d'immersió

- Fixant els paràmetres addicionals
- Ignorant alguns dels resultats retornats

# Canvis en l'especificació: Immersions

Canvis en els paràmetres impliquen canvis en l'especificació:

- Afebliment de la post: la crida recursiva només fa una part de la feina
- Enfortiment de la pre: la crida recursiva rep feta una part de la feina, ella la completa

La primera sol ser més natural. La segona té l'avantatge que dóna solucions més fàcils de transformar a iteratives (si calgués)

## Suma dels elements d'un vector: afebliment de la Post

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v);
```

# Suma dels elements d'un vector: afebliment de la Post

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v);
```

## Funció d'immersió

```
// Pre: -1 \le i < v.\text{size}()

// Post: retorna la suma de v[0..i]

int i_suma(const vector<int>& v, int i);
```

## Suma dels elements d'un vector: afebliment de la Post

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v);
```

# Funció d'immersió // Pre: -1 ≤ i < v.size() // Post: retorna la suma de v[0..i] int i\_suma(const vector<int>& v, int i);

```
Crida inicial

// Pre: cert

// Post: retorna la suma de v */
int suma(const vector<int>& v) {
   return i_suma(v, v.size()-1);
}
```

# Implementació de la funció d'immersió

```
// Pre: -1 \leq i < v.size()
// Post: retorna la suma de v[0..i]
int i_suma(const vector<int>& v, int i) {
   if (i < 0)
      return 0;
   else
      return i_suma(v, i-1) + v[i];
}</pre>
```

## Immersió alternativa

## Funció d'inmersió

```
// Pre: 0 \le i \le v.size()

// Post: retorna la suma de v[i..v.size()-1]

int i_suma(const vector<int>& v, int i);
```

## Immersió alternativa

# Funció d'inmersió // Pre: 0 \le i \le v.size() // Post: retorna la suma de v[i..v.size() - 1] int i\_suma(const vector \int \delta v, int i);

## Crida inicial

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v) {
   return i_suma(v, 0);
}
```

# Implementació de la funció d'immersió

```
// Pre: 0 \leq i \leq v.size()
// Post: retorna la suma de v[i..v.size() - 1]
int i_suma(const vector<int>& v, int i) {
   if (i == v.size())
      return 0;
   else
      return v[i]+i_suma(v, i+1);
}
```

## Cerca en un vector ordenat

```
// Pre: v.\operatorname{size}() > 0 i v està ordenat creixentment

// Post: Retorna una posicio on és troba l'element x dins

// el vector v, si x \in v. Si x \notin v, retorna -1.

template <class T>

int cerca(const vector<T>& v, const T& x);
```

## Afebliment de la post

## Afebliments de la Post possibles:

- canviar v per v[0..i]
- ullet o canviar v per v[j..v.size()-1]
- ... o les dues coses: canviar v per v[i..j]!

```
// Pre: -1 \le i, j \le v.\text{size}(), i \le j+1 \text{ i } v està ordenat creixentment // Post: Retorna una posicio on és troba l'element x dins // el subvector v[i..j], si x \in v[i..j]. Si x \notin v, retorna -1. template <class T> int i_cerca(const vector<T>& v, const T& x, int i, int j);
```

# Afebliment de la post

## Afebliments de la Post possibles:

- canviar v per v[0..i]
- ullet o canviar v per v[j..v.size()-1]
- ... o les dues coses: canviar v per v[i..j]!

```
// Pre: -1 \le i, j \le v.\text{size}(), i \le j+1 \text{ i } v

// està ordenat creixentment

// Post: Retorna una posicio on és troba l'element x dins

// el subvector v[i..j], si x \in v[i..j]. Si x \notin v,

// retorna -1.

template <class T>

int i_cerca(const vector<T>& v, const T& x, int i, int j);
```

- En les dues primeres alternatives d'afebliment de la Post
   cerca següencial
- Amb la tercera podem fer cerca dicotòmica

## **Immersions**

#### No oblideu de:

- Dir quina immersió fareu, quin paràmetre afegir
- Donar la capçalera de la nova funció d'immersió
- Especificar-la! (paper dels nous paràmetres / resultats)
- Donar la crida inicial des de la funció original

# Part V

# Disseny Recursiu

- (21) Recusió, definicions recursives i inducció
- 22 Cipis de disseny recursiu
- 23 Immersió de funcions: Afebliment de la postcondició
- 24 Immersió de funcions: Enfortiment de la precondició (\*)
- 25 Recursivitat lineal final i algorismes iteratius (\*\*)

## Suma dels elements d'un vector

```
// Pre: cert
// Post: retorna la suma dels elements de v
int suma(const vector<int>& v);
```

## Suma dels elements d'un vector

```
// Pre: cert
// Post: retorna la suma dels elements de v
int suma(const vector<int>& v);
```

#### Enfortiment de la Pre:

```
// Pre: 0 \le i \le v.\text{size}(), i // sum és la suma dels elements de v[0..i-1] // Post: retorna la suma dels elements de v int i_suma(const vector<int>& v, int i, int sum);
```

## Per enfortiment de la Pre

```
// Pre: 0 \le i \le v.\text{size}(), i // sum és la suma dels elements de v[0..i-1] // Post: retorna la suma dels elements de v int i_suma(const vector<int>& v, int i, int sum);
```

## Per enfortiment de la Pre

```
// Pre: 0 \le i \le v.\text{size}(), i
// sum és la suma dels elements de v[0..i-1]
// Post: retorna la suma dels elements de v
int i_suma(const vector<int>& v, int i, int sum);
```

## Crida inicial per calcular tota la suma del vector:

```
// Pre: cert
// Post: retorna la suma dels elements de v
int suma(const vector<int>& v) {
  return i_suma(v, 0, 0);
}
```

# Implementació de la funció d'immersió

```
// Pre: 0 \leq i \leq v.size(), i
// sum \(\delta\) s la suma dels elements de v[0..i-1]
// Post: retorna la suma dels elements de v
int i_suma(const vector<int>\& v, int i, int sum) {
   if (i == v.size())
      return sum;
   else
      return i_suma(v, i+1, sum+v[i]);
}
```

# Part V

# Disseny Recursiu

- 21) Recusió, definicions recursives i inducció
- 22 Decipis de disseny recursiu
- 23 Immersió de funcions: Afebliment de la postcondició
- 24 Immersió de funcions: Enfortiment de la precondició (\*)
- 25 Recursivitat lineal final i algorismes iteratius (\*\*)

 Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (tail recursion):

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (tail recursion):
  - La darrera instrucció que s'executa (cas recursiu) és la crida recursiva

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (tail recursion):
  - La darrera instrucció que s'executa (cas recursiu) és la crida recursiva
  - El resultat de la funció (cas recursiu) és el resultat que s'ha obtingut de la crida recursiva, sense cap modificació

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (tail recursion):
  - La darrera instrucció que s'executa (cas recursiu) és la crida recursiva
  - El resultat de la funció (cas recursiu) és el resultat que s'ha obtingut de la crida recursiva, sense cap modificació
- Motivació: mètode simple per transformar un algorisme recursiu lineal final en un algorisme iteratiu

# Exemple: factorial

```
int fact(int n) {
   if (n <= 1) return n;
   else return n * fact(n - 1);
}</pre>
```

Recursivitat lineal però no final: és fa el producte després de la crida recursiva

# Exemple: factorial

### Enfortiment de la Pre:

```
// Pre: n ≥ i ≥ 0 ∧ p = i!
// Post: retorna n!
int i_fact(int n, int i, int p) {
   if (i == n)
      return p;
   else
      return i_fact(n, i + 1, p * (i + 1));
}
```

- Recursivitat lineal final
- Crida inicial: fact(n) ≡ i\_fact(n,0,1)

## Exemple: factorial

- Transformem paràmetres en variables locals
- 2 La Pre és l'invariant del bucle
- La crida inicial dona com inicialitzar les variables

## Transformació recursivitat lineal final a iteració, II

## Funció recursiva lineal final:

## Iteració equivalent:

```
// Pre: P(x,y)
T2 func_iter(T1 x, T3 y) {
    T2 res;

while (not cas_base(x)) {
      y = new_y(x,y);
      x = new_x(x);
    }
    res = sol_directa(x,y);
    return s;
}
// Post: Q(x,y,s)
```

## Suma d'un vector d'enters

## Implementació recursiva final:

Llamada inicial:  $suma(v) \equiv i\_suma(v, 0, 0)$ 

## Transformació a iteratiu

```
/* Pre: cert */
int suma iter(const vector<int>& v) {
 int i = 0; int sum = 0;
 // Inv: 0 < i < v.size() i sum és
 // la suma dels elements de v[0..i-1]
  while (i != v.size()) {
   sum += v[i]; // sum + v[i]
   ++i; // i+1
  return suma;
/* Post: el valor retornat es la suma de tots els elements
  del vector v */
```

# Part VI

# Millores de Eficiència

- 26 Eliminació de càlculs repetits
- 27 homersions d'eficiència
- <sup>28</sup> Més exemples

#### Iteració:

 Afegim variables locals que recorden càlculs ja efectuats per a la propera iteració

#### Iteració:

- Afegim variables locals que recorden càlculs ja efectuats per a la propera iteració
- No apareixen en la Pre ni la Post. L'especificació no canvia

#### Iteració:

- Afegim variables locals que recorden càlculs ja efectuats per a la propera iteració
- No apareixen en la Pre ni la Post. L'especificació no canvia
- Però apareixen a l'invariant. Cal dir què valen a cada iteració

### Recursió:

 Les variables locals no serveixen. Es creen noves a cada crida

# Eficiència per eliminació de càlculs repetits

#### Recursió:

- Les variables locals no serveixen. Es creen noves a cada crida
- Funció d'immersió d'eficiència, recursiva: Nous paràmetres d'entrada o de sortida

# Eficiència per eliminació de càlculs repetits

#### Recursió:

- Les variables locals no serveixen. Es creen noves a cada crida
- Funció d'immersió d'eficiència, recursiva: Nous paràmetres d'entrada o de sortida
- S'han d'afegir a la Pre/Post!

# Eficiència per eliminació de càlculs repetits

#### Recursió:

- Les variables locals no serveixen. Es creen noves a cada crida
- Funció d'immersió d'eficiència, recursiva: Nous paràmetres d'entrada o de sortida
- S'han d'afegir a la Pre/Post!
- La funció desitjada no és recursiva, crida a la d'immersió

# Part VI

# Millores de Eficiència

- (26) Eliminació de càlculs repetits
- 27 Immersions d'eficiència
- 28) Més exemples

## Concepte d'immersió d'eficiència

- Font frequent d'ineficiència: Repetir càlculs ja fets
- En programes iteratius: Guardar variables temporals que guarden resultats d'una iteració a la següent
- En programes recursius, una variable temporal és local a cada crida recursiva. No guarda resultats d'una crida a l'altra
- Immersió d'eficiència: Introducció de paràmetres o resultats addicionals per transmetre valors ja calculats en/a altres crides
- Pot haver de fer-se a més una immersió/generalització per tal de possibilitar la solució recursiva.

## Exemple: suma dels k anteriors

```
// Pre: v.\text{size}() > k \ge 0

// Post: retorna cert sii hi ha algun i entre k i

// v.\text{size}() - 1 tal que v[i] = v[i-k] + ... + v[i-1]

bool kanteriors(const vector<double>& v, int k);
```

# Exemple: suma dels k anteriors

```
bool kanteriors(const vector<double>& v, int k) {
   int i = k;
   while (i < v.size()) {
        // Inv: no hi ha cap j < i tal que
        // v[j] = v[j - k] + ... + v[j - 1]
        if (v[i] == suma(v, i-k, i-1)) return true;
        ++i;
   }
   return false;
}</pre>
```

suma (v, i-k, i-1) té cost proporcional a  $k \to \cos t$  total proporcional  $(n-k) \cdot k$ 

## Exemple: suma dels k anteriors

### Millora: propagar la suma dels k anteriors

```
bool kanteriors (const vector < double > & v, int k) {
    double sum = 0:
    for (int j = 0; j < k; ++j) sum += v[j];
    int i = k;
    while (i < v.size()) {
       // Inv: no hi ha cap j < i tal que
       |v(j)| = v(j-k) + ... + v(j-1)
       // i sum = v[i-k] + ... + v[i-1]
       if (v[i] == sum) return true;
       sum = sum - v[i-k] + v[i];
       ++i;
    return false;
```

cost total proporcional a n, independent de k!

```
// Pre: v.\text{size}() > k \ge 0

// Post: retorna cert sii hi ha algun i entre k i

// v.\text{size}() - 1 tal que v[i] = v[i-k] + \ldots + v[i-1]

bool kanteriors(const vector<double>& v, int k);
```

### Primer, cal immersió d'especificació:

```
// Pre: v.\text{size}() \ge m \ge k \ge 0

// Post: retorna cert sii hi ha algun i entre m i

// v.\text{size}() - 1 tal que v[i] = v[i-k] + \ldots + v[i-1]

bool i_kanteriors(const vector<double>& v, int v, int v);
```

```
bool i_kanteriors(const vector<double>& v, int k, int m) {
   if (m == v.size()) return false;
   else if (v[m] == suma(v,m-k,m-1)) return true;
   else return i_kanteriors(v,k,m+1);
}
```

```
bool i_kanteriors(const vector<double>& v, int k, int m) {
   if (m == v.size()) return false;
   else if (v[m] == suma(v,m-k,m-1)) return true;
   else return i_kanteriors(v,k,m+1);
}
```

Problema: fem k sumes a cada crida  $\rightarrow$  cost total  $(n - k) \cdot k$ 

### Immersió d'eficiència:

#### Crida inicial:

```
bool kanteriors(const vector<double>& v, int k) {
    return ie_kanteriors(v,k,k,suma(v,0,k-1));
}
```

Immersió d'eficiència alternativa: afegim la suma com a resultat, en comptes de com a paràmetre d'entrada

```
// Pre: v.\text{size}() \ge m \ge k \ge 0

// Post: retorna cert sii hi ha algun i entre m i v.\text{size}()-1

// tal que v[i] = v[i-k] + \ldots + v[i-1], i a més

// sum = v[m-k] + \cdots + v[m-1]

bool ie_kanteriors(const vector<double>& v,

int k, int m, double& sum);
```

Exercici: la implementació i la crida inicial.

Pista: cas base: m = v.size()

Diem que en un vector un element és frontissa si és igual que la diferència entre els que el segueixen i els que el precedeixen

```
Exemples: [1,3,11,6,5,4] [2,1,1] [1,2,1,0,4]
```

```
// Pre: cert
// Post: retorna el nombre d'elements frontissa de v
int frontisses(const vector<double>& v);
```

```
int frontisses(const vector<double>& v) {
   int i = 0;
   int nf = 0;
   while (i < v.size()) {
        // Inv: 0 \le i \le v.size()
        // i nf = nombre d'elements frontissa a v[0..i-1]
        if (v[i] == suma(v,i+1,v.size()-1) - suma(v,0,i-1)) ++nf;
        ++i;
   }
   return nf;
}</pre>
```

```
int frontisses(const vector<double>& v) {
    int i = 0;
    int nf = 0;
    while (i < v.size()) {
        // Inv: 0 \le i \le v.size()
        // i nf = nombre d'elements frontissa a v[0..i-1]
        if (v[i] == suma(v,i+1,v.size()-1) - suma(v,0,i-1)) ++nf;
        ++i;
    }
    return nf;
}</pre>
```

Com que suma (v, a, b) té cost proporcional a b - a, el cost d'aquesta funció és proporcional a  $(v.size())^2$  - quadràtic!

### Millora: reaprofitar sumes fetes

```
int frontisses(const vector<double>& v) {
  double sumapost = suma(v,1,v.size()-1);
  double sumaant = 0:
  int i = 0: int nf = 0:
  while (i < v.size()) {
   // Inv: 0 < i < v.size() i
   // nf = nombre de frontisses a v[0..i-1]
   // i sumaant és la suma de v[0..i-1],
   // i sumapost és la suma de v[i+1..v.size()-1]
    if (v[i] == sumapost-sumaant) ++nf;
    sumaant += v[i];
    if (i < v.size()-1) sumapost -= v[i+1];
   ++i;
  return nf;
```

Millora: reaprofitar sumes fetes

```
int frontisses(const vector<double>& v) {
 double sumapost = suma(v,1,v.size()-1);
 double sumaant = 0:
 int i = 0: int nf = 0:
 while (i < v.size()) {
   // Inv: 0 < i < v.size() i
   // nf = nombre de frontisses a v[0..i-1]
   // i sumaant és la suma de v[0..i-1],
   // i sumapost és la suma de v[i+1..v.size()-1]
   if (v[i] == sumapost-sumaant) ++nf;
   sumaant += v[i];
   if (i < v.size()-1) sumapost -= v[i+1];
   ++i;
 return nf;
```

Cost lineal - de l'ordre de v.size()

Millora: reaprofitar sumes fetes

```
int frontisses(const vector<double>& v) {
 double sumapost = suma(v,1,v.size()-1);
 double sumaant = 0:
 int i = 0; int nf = 0;
 while (i < v.size()) {
   // Inv: 0 < i < v.size() i
   // nf = nombre de frontisses a v[0..i-1]
   // i sumaant és la suma de v[0..i-1],
   // i sumapost és la suma de v[i+1..v.size()-1]
   if (v[i] == sumapost-sumaant) ++nf;
   sumaant += v[i];
   if (i < v.size()-1) sumapost -= v[i+1];
   ++i;
 return nf;
```

Cost lineal - de l'ordre de v.size()

Lleugera millora: mantenir directament sumapost - sumaant

```
// Pre: cert
// Post: retorna el nombre d'elements frontissa en v
int frontisses(const vector<double>& v);
```

```
// Pre: cert
// Post: retorna el nombre d'elements frontissa en v
int frontisses(const vector<double>& v);
```

### Primer, cal immersió d'especificació:

```
// Pre: 0 \le i \le v.\text{size}()

// Post: retorna el nombre d'elements frontissa en v[0..i-1]

int i_frontisses(const vector<double>& v, int i);
```

```
// Pre: 0 \leq i \leq v.size()
// Post: retorna el nombre d'elements frontissa en v[0..i-1]
int i_frontisses(const vector<double>& v, int i) {
    if (i == 0) return 0;
    else {
        int nf = i_frontisses(v,i-1);
        if (v[i-1] == suma(v,i,v.size()-1)-suma(v,0,i-2)) ++nf;
        return nf;
    }
}
```

```
// Pre: 0 \leq i \leq v.size()
// Post: retorna el nombre d'elements frontissa en v[0..i-1]
int i_frontisses(const vector<double>& v, int i) {
    if (i == 0) return 0;
    else {
        int nf = i_frontisses(v,i-1);
        if (v[i-1] == suma(v,i,v.size()-1)-suma(v,0,i-2)) ++nf;
        return nf;
    }
}
```

Problema: recàlcul de sumes - quadràtic

Immersió d'eficiència afegint paràmetres d'entrada: Passem suma(posteriors), suma(anteriors) com a paràmetres

Immersió d'eficiència afegint paràmetres d'entrada: Passem suma(posteriors), suma(anteriors) com a paràmetres

```
Crida inicial: frontisses (v) és
```

```
ie\_frontisses(v, v.size()-1, suma(v, 0, v.size()-2), 0)
```

Implementació queda com a exercici De l'ordre de *N* operacions - lineal

### Alternativa, immersió d'eficiència afegint resultats

### Alternativa, immersió d'eficiència afegint resultats

#### Crida inicial:

```
int frontisses(const vector<double>& v) {
    double sa,sp;
    return ie_frontisses(v,v.size(),sa,sp);
}
```

```
// Pre: 0 < i < v.size() = N
// Post: retorna el nombre d'elements frontissa en v[0..i-1] i
// sumaant = suma(v, 0, i-1) i
// sumapost = suma(v,i+1,v.size()-1)
int ie frontisses (const vector < double > & v, int i,
                  double& sumaant, double& sumapost)) {
    if (i == 0) {
       sumaant = 0; sumapost = suma(v, 0, v.size()-1);
       return 0;
    } else {
       double sa, sp;
       int nf = ie_frontisses(v, i-1, sa, sp);
       if (v[i-1] == sp-sa) ++nf;
       sumaant = sa+v[i-1]; sumapost = sp-v[i];
       return nf;
```

Donat un arbre de doubles, construir-ne un altre de la mateixa forma que a cada node conté la mitjana dels valors del subarbre arrelat al node corresponent de l'original

```
// Pre: cert
// Post: retorna l'arbre de mitjanes d'a
BinTree<double> arbre_mitjanes(const BinTree<double>& a);
```

Donat un arbre de doubles, construir-ne un altre de la mateixa forma que a cada node conté la mitjana dels valors del subarbre arrelat al node corresponent de l'original

```
// Pre: cert
// Post: retorna l'arbre de mitjanes d'a
BinTree<double> arbre_mitjanes(const BinTree<double>& a);
```

Dificultat: la mitjana d'un arbre NO es pot calcular a partir de l'arrel i les mitjanes dels dos subarbres

## Arbre de mitjanes: Solució ineficient

```
BinTree<double> arbre_mitjanes(const BinTree<double>& a) {
   if (not a.empty()) {
      double x = a.value();
      BinTree<double> b1 = arbre_mitjanes(a.left());
      BinTree<double> b2 = arbre_mitjanes(a.right());
      double s1 = suma(a.left()); double s2 = suma(a.right());
      int n1 = talla(a.left()); int n2 = talla(a.right());
      return BinTree<double>((x+s1+s2)/(1+n1+n2),b1,b2);
   }
}
```

Ineficient: Tres recorreguts d'a (recursió, suma, mida)

Immersió d'eficiència: un sol recorregut que retorni a més suma i mida

Immersió d'eficiència: un sol recorregut que retorni a més suma i mida

#### Crida inicial:

```
BinTree<double> arbre_mitjanes(const BinTree<double>& a) {
   double s; int n;
   BinTree<double> b;
   ie_arbre_mitjanes(a,b,s,n);
   return b;
}
```

```
void ie_arbre_mitjanes(const BinTree<double>& a, BinTree<double>& b,
                       double& s, int& n) {
   if (a.empty()) {
       s = 0: n = 0:
   } else {
       double s1, s2; int n1, n2;
       double x = a.value():
       BinTree < double > b1, b2;
       ie arbre mitjanes(a.left(),b1,s1,n1);
       ie_arbre_mitjanes(a.right(),b2,s2,n2);
       s = x + s1 + s2;
       n = 1 + n1 + n2;
       b = BinTree < double > (s/n,b1,b2);
```

Cost lineal, un sol recorregut

## Determinar si un arbre és equilibrat

Concepte important en estructures de dades avançades: Un arbre és equilibrat si i només si

- els seus dos fills són equilibrats, i a més
- la diferència d'alçades dels subarbres fills no supera la unitat.

## Determinar si un arbre és equilibrat

```
// Pre: cert
// Post: retorna cert si i només si a és un arbre equilibrat
bool equilibrat(const BinTree<int> &a);
```

## Determinar si un arbre és equilibrat

```
// Pre: cert
// Post: retorna cert si i només si a és un arbre equilibrat
bool equilibrat(const BinTree<int> &a);
```

#### Suposem que ja tenim implementada la funció

```
// Pre: cert
// Post: retorna la longitud del camí més llarg de l'arrel
// a una fulla de l'arbre a
int alcaria(const BinTree<int> &a);
```

i que tarda temps proporcional a la mida de l'arbre

## Implementació, II

- Quin és el cost de l'algorisme?
  - Analitzem l'arbre de crides...

- Quin és el cost de l'algorisme?
  - Analitzem l'arbre de crides...
  - Pensem en un arbre lineal (cap fill dret, fill només esquerre)

. . .

- Quin és el cost de l'algorisme?
  - Analitzem l'arbre de crides...
  - Pensem en un arbre lineal (cap fill dret, fill només esquerre)
  - $|a|^2/2$

- Quin és el cost de l'algorisme?
  - Analitzem l'arbre de crides...
  - Pensem en un arbre lineal (cap fill dret, fill només esquerre)
  - $|a|^2/2$
  - (don't panic: A EDA practicarem això)

- Quin és el cost de l'algorisme?
  - Analitzem l'arbre de crides...
  - Pensem en un arbre lineal (cap fill dret, fill només esquerre)
  - $|a|^2/2$
  - (don't panic: A EDA practicarem això)
- Com evitem repetir càlculs, recorrer cada arbre molts cops?

#### Solució: immersió d'eficiència

Retornar més informació per evitar repetir càlculs:

```
// Pre: cert
// Post: en el parell retornat
// - "first" indica si a es un arbre equilibrat
// - "second" conté l'alçaria de l'arbre si a és equilibrat
pair<bool, int> i_equilibrat(const BinTree<int>& a);
```

#### Solució: immersió d'eficiència

Retornar més informació per evitar repetir càlculs:

```
// Pre: cert
// Post: en el parell retornat
// - "first" indica si a es un arbre equilibrat
// - "second" conté l'alçaria de l'arbre si a és equilibrat
pair<bool, int> i_equilibrat(const BinTree<int>& a);
```

#### Crida inicial:

```
// Pre: cert
// Post: retorna cert ssi a és un arbre equilibrat
bool equilibrat2(const BinTree<int>& a) {
  pair<bool, int> e = i_equilibrat(a);
  return e.first;
}
```

## Implementació de la funció d'immersió

```
// Pre: cert
   // Post: en el parell retornat
   // - "first" indica si a es un arbre equilibrat
   // - "second" és l'alcaria de l'arbre, si a és equilibrat
pair<bool, int> i equilibrat(const BinTree<int>& a) {
   if (a.empty()) return make_pair<true, 0>;
   else {
     pair<bool, int> e1 = i_equilibrat(a.left());
     if (el.first) {
       pair<bool, int> e2 = i equilibrat(a.right());
       bool eq = e2.first and (abs(e1.second - e2.second) <= 1);</pre>
                 // ja sabem que el.first == true
       if (eq)
         return make pair(true,1 + max(e1.second, e2.second));
       else
         return make_pair(false, -1); //l'alçaria és irrellevant
     } else { // e1.first == false
         return make_pair(false, -1);
```

## Part VI

## Millores de Eficiència

- (26) Eliminació de càlculs repetits
- 27 hamersions d'eficiència
- 28 Més exemples

## Funció exponencial

#### Sèrie de Taylor de l'exponencial

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^i}{i!} + \dots$$
 (1)

```
// Pre: x>0 i n\geq 0

// Post: el valor retornat es la suma dels n primers termes

// de l'expansio en sèrie de Taylor de e^x, és a dir,

// retorna \sum_{k=0}^{n-1} x^k/k!

double exponencial(double x, int n);
```

## Implementació

```
double exponencial(double x, int n) {
  double e = 0;
  int i = 0;
  while (i < n) {
     e += potencia(x,i)/factorial(i);
     ++i;
  }
  return e;
}</pre>
```

#### L'invariant és:

$$0 \leq i \leq n \wedge e = \sum_{k=0}^{i-1} \frac{x^k}{k!}$$

## Implementació

#### Especificació de les dues funcions auxiliars:

```
// Pre: x > 0 i i ≥ 0
// Post: retorna x<sup>i</sup>
double potencia(double x, int i);

// Pre: n ≥ 0
// Post: retorna n!
int factorial(int n);
```

Càlculs repetits tant al factorial com a la potència Mantenir variable p = potencia(x, i)

Càlculs repetits tant al factorial com a la potència

Mantenir variable  $p = \mathtt{potencia}(x,i) = x * \mathtt{potencia}(x,i-1)$ 

Mantenir variable f = factorial(i)

Càlculs repetits tant al factorial com a la potència

```
Mantenir variable p = potencia(x, i) = x * potencia(x, i - 1)
```

Mantenir variable f = factorial(i) = i \* factorial(i-1)

Càlculs repetits tant al factorial com a la potència

```
Mantenir variable p = potencia(x, i) = x * potencia(x, i - 1)
```

Mantenir variable f = factorial(i) = i \* factorial(i-1)

Problema:  $x^i$  i i! creixen molt ràpid

però  $x^i/i!$  decreix

$$t_i = rac{x^i}{i!}$$

$$t_i = rac{x^i}{i!} = rac{x^{i-1} * x}{(i-1)! * i}$$

$$t_i = rac{x^i}{i!} = rac{x^{i-1} * x}{(i-1)! * i} = t_{i-1} rac{x}{i}, \qquad i > 0$$

$$t_i = rac{x^i}{i!} = rac{x^{i-1} * x}{(i-1)! * i} = t_{i-1}rac{x}{i}, \qquad i > 0$$

$$t_0=x^0/0!=1$$

## Implementació

```
double exponencial(double x, int n) {
   double e = 0;
   double t = 1; // t = t_0 = x^0/0!
   int i = 0;
   // Inv: 0 \le i \le n, t = x^i/i!
   // i e = \sum_{k=0}^{i-1} x^k / k!
   while (i < n) {
      e += t;
       ++i;
       t = t * x/i;
   return e;
```

#### Exercici: Funció cosinus

#### Sèrie de Taylor del cosinus

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$
 (2)

```
// Pre: n \ge 0

// Post: e conté la suma dels n primers termes de

// l'expansio en sèrie de Taylor de \cos(x)

double cosinus(double x, int n);
```

## Exemple: La successió de Fibonacci

$$F_n = egin{cases} n & ext{si } n \leq 1, \ F_{n-1} + F_{n-2} & ext{si } n \geq 2. \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \dots$$

## Implementació recursiva

```
// Pre: n≥0
// Post: retorna Fn
int fibonacci(int n) {
   if (n <= 1) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}</pre>
```

## Implementació recursiva

```
// Pre: n ≥ 0
// Post: retorna F<sub>n</sub>
int fibonacci(int n) {
   if (n <= 1) return n;
   else return fibonacci(n-1) + fibonacci(n-2);
}</pre>
```

Quants cops cridem fibonacci(n-i) en executar fibonacci(n)?

## Implementació recursiva

```
// Pre: n≥0
// Post: retorna F<sub>n</sub>
int fibonacci(int n) {
   if (n <= 1) return n;
   else return fibonacci(n-1) + fibonacci(n-2);
}</pre>
```

Quants cops cridem fibonacci(n-i) en executar fibonacci(n)? Resposta:  $F_i$  cops (intenteu demostrar-ho per inducció)

## Cost temporal?

```
F_n creix com \phi^n (\phi = (1+\sqrt{5})/2 = \text{ra\'o àuria} = \text{soluci\'o de } (\phi = 1+1/\phi) =  \simeq 1.618033...)
```

Càlcul molt lent

#### Truc

Suposem que tenim el parell  $\langle F_{n-1}, F_{n-2} \rangle$ . Llavors, el parell  $\langle F_n, F_{n-1} \rangle$  és

$$\langle F_n, F_{n-1} \rangle = \langle F_{n-1} + F_{n-2}, F_{n-1} \rangle$$

#### Versió iterativa

```
// Pre: n > 0
// Post: retorna F_n
int fibonacci(int n) {
   if (n <= 1) return n;
   else {
       int f1 = 1;
       int f2 = 0;
       int i = 2;
       // Inv: f1 = F_{i-1} \wedge f2 = F_{i-2} \wedge 2 < i < n
       while (i <= n) {
           int temp = f1;
           f1 = f1 + f2;
            f2 = temp;
            ++i;
        return f1;
```

# Detecció de la repetició de càlculs en programes recursius

```
#include <utility>  
// Pre: n > 0  
// Post: retorna \langle F_n, F_{n-1} \rangle  
pair<int, int> i_fibonacci(int n);
```

## Implementació funció d'immersió

```
// Pre: n > 0
// Post: retorna \langle F_n, F_{n-1} \rangle
pair < int, int > i_fibonacci (int n) {
    if (n == 1) return make_pair(1,0);
    } else {
        pair < int, int > p = i_fibonacci (n - 1);
        // HI: p.first i p.second contenen F_{n-1} i F_{n-2} resp.
        return make_pair(p.first + p.second, p.first);
    }
}
```

#### Crida a la funció d'immersió

```
// Pre: n≥0
// Post: retorna F<sub>n</sub>
int fibonacci(int n) {
   if (n == 0) return 0;
   else return i_fibonacci(int n).first;
}
```

#### Alternativa

### Funcions que retornen més d'un valor → paràmetres per referència

```
// Pre: n > 0

// Post: retorna f1 = F_n i f2 = F_{n-1}

void i_fibonacci(int n, int& f1, int& f2);
```

# Part VII

# Tipus Recursius de Dades

- 29 Apuntadors i memòria dinàmica
- 30 Plus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35 Albre Wais
  - (36) Arbres generals
  - 37 Implementació de mètodes: accedint la representació
  - 38) Estructures de dades noves

### **Apuntadors**

En C++, per a cada tipus  ${\mathbb T}$  hi ha un altre tipus "apuntador a  ${\mathbb T}$ "

### **Apuntadors**

En C++, per a cada tipus  $\mathbb{T}$  hi ha un altre tipus "apuntador a  $\mathbb{T}$ "

Una variable de tipus "apuntador a T" pot contenir

- una referència a una variable o objecte de tipus T,
- o un valor especial nullptr
- o res sensat, si no ha estat inicialitzada

### **Apuntadors**

En C++, per a cada tipus  $\mathbb{T}$  hi ha un altre tipus "apuntador a  $\mathbb{T}$ "

Una variable de tipus "apuntador a T" pot contenir

- una referència a una variable o objecte de tipus T,
- o un valor especial nullptr
- o res sensat, si no ha estat inicialitzada

La referència pot estar implementada amb una adreça de memòria o d'altres maneres; és irrellevant a Programació 2

### **Operadors**

- T\* p: Declaració de variable p com "apuntador a T"
- \*p: Objecte referenciat pel apuntador p
- Omposició de \* i el selector . de struct/class
- &v: Referència a v, de tipus "apuntador al tipus de v"
- onew, delete: Creació i destrucció de memòria dinàmica

## Exemple

```
int x;
int* p;
p = &x;
x = 5;
cout << *p << endl; // escriu 5
*p = 3;
cout << x << endl; // escriu 3</pre>
```

#### **Observacions**

Error accedir a \*p si p no referencia cap objecte

és a dir, si p == nullptr o si p no inicialitzat

### Observacions

```
Estudiant x;
Estudiant* p;
```

- x sempre referenciarà el mateix objecte mentre viu
- $\star p$  pot anar referenciant diferents objectes quan canviem el valor de p

### **Observacions**

```
Estudiant x;
Estudiant* p;
```

- x sempre referenciarà el mateix objecte mentre viu
- $\star p$  pot anar referenciant diferents objectes quan canviem el valor de p

- Quan fem p = &x, tenim un objecte amb dos noms, \*p i x
- Això se'n diu aliasing. Molt útil però pot ser perillós

### Exemple

```
int x = 1;
int y = 2;
int* p = &x;
int* q = &y;
cout << x << " " << y << endl; // escriu "1 2"
*q = *p;
cout << x << " " << y << endl; // escriu "1 1"
*q = 3;
cout << x << " " << y << endl; // escriu "1 3"
q = p;
*q = 4;
cout << x << " " << y << endl; // escriu "4 3"
// en aquest punt, pdem referir-nos a x de 3 maneres: x, *p i *q</pre>
```

# **Preguntes**

#### Declarem

```
int x; int* p; int* q;
```

### És sempre cert que...

- \* (&x) == x?
- & (\*p) == p?
- $\bullet p == q implica (*p) == (*q)?$
- (\*p) == (\*q) implica p == q?

## Apuntadors i structs

És molt frequent tenir un apuntador a un struct o un objecte d'una classe, i voler accedir a un camp de l'struct apuntat, invocar un mètode de l'objecte, etc.

```
Notació còmoda: p->camp equival a (*p).camp, p->mètode(...) equival a (*p).mètode(...)
```

### Apuntadors i structs

És molt frequent tenir un apuntador a un struct o un objecte d'una classe, i voler accedir a un camp de l'struct apuntat, invocar un mètode de l'objecte, etc.

```
Notació còmoda: p->camp equival a (*p).camp, p->mètode(...) equival a (*p).mètode(...)
```

### Exemple

```
struct par {
    string nom;
    int edat;
};

par* ppar = ...;
++ppar -> edat;

Estudiant* pe = ...
...
if (pe->te_nota()) { cout << pe->consultar_DNI() << endl; }</pre>
```

### Apuntadors i structs

És molt frequent tenir un apuntador a un struct o un objecte d'una classe, i voler accedir a un camp de l'struct apuntat, invocar un mètode de l'objecte, etc.

```
Notació còmoda: p->camp equival a (*p).camp, p->mètode(...) equival a (*p).mètode(...)
```

```
Exemple

struct par {
    string nom;
    int edat;
};

par* ppar = ...;
++ppar -> edat;

Estudiant* pe = ...
...
if (pe->te_nota()) { cout << pe->consultar_DNI() << endl; }</pre>
```

#### Ho hem vist abans:

apuntador this al paràmetre implícit. (\*this, this->)

Quan declarem un apuntador T\* p, està indefinit. El definim:

• Fent-lo apuntar a un objecte del tipus T ja existent:

```
p = q 0 p = &x;
```

Quan declarem un apuntador T\* p, està indefinit. El definim:

 $\bullet$  Fent-lo apuntar a un objecte del tipus  ${\tt T}$  ja existent:

$$p = q 0 p = &x$$

 O donant-li el valor nullptr, per explicitar "no referencia res"

Quan declarem un apuntador T\* p, està indefinit. El definim:

 $\bullet$  Fent-lo apuntar a un objecte del tipus  ${\tt T}$  ja existent:

```
p = q 0 p = &x;
```

- O donant-li el valor nullptr, per explicitar "no referencia res"
- Reservant memòria perquè apunti a un nou objecte:

```
p = new T;
```

Quan declarem un apuntador T\* p, està indefinit. El definim:

ullet Fent-lo apuntar a un objecte del tipus  ${\mathbb T}$  ja existent:

$$p = q 0 p = &x$$

- O donant-li el valor nullptr, per explicitar "no referencia res"
- Reservant memòria perquè apunti a un nou objecte:

```
p = new T;
```

- Aquest objecte no tindrà nom propi: només ⋆p
- Queda inaccessible! si modifiquem p i no hi ha cap altre apuntador que l'hi apunta

#### new and delete

### Operacions de gestió de memòria dinàmica:

- new T: reserva memòria dinàmica per a un nou objecte, li aplica la creadora de T i retorna un apuntador a ell
- delete p: aplica la destructora del tipus a l'objecte apuntat per p i allibera la memòria que ocupa ("esborra" l'objecte)

#### new and delete

#### Operacions de gestió de memòria dinàmica:

- new T: reserva memòria dinàmica per a un nou objecte, li aplica la creadora de T i retorna un apuntador a ell
- delete p: aplica la destructora del tipus a l'objecte apuntat per p i allibera la memòria que ocupa ("esborra" l'objecte)
- Atenció: "delete p" NO esborra el punter p; esborra l'objecte apuntat per p
- el valor de p després de delete p és indefinit

### Exemples

```
struct T {
  int camp1;
  bool camp2;
void f(...) {
           // es crida la creadora de T
 T x:
 x.camp1 = 20; x.camp2 = true;
  T* p = new T; // p apunta a un objecte nou;
                  // crida la constructora de T
 p->camp1 = 30; p->camp2 = false;
  . . .
  delete p; // es crida destructora de T
             // i s'allibera *p; p indefinit
// i aquí es crida automàticament a la destructora de T
// de la variable local x
```

 Deixar memòria sense alliberar (objectes dinàmics sense esborrar) → memory leaks

- Deixar memòria sense alliberar (objectes dinàmics sense esborrar) → memory leaks
- Accedir a memòria ja alliberada (objectes esborrats).
   Vigileu amb l'aliasing → dangling references

- Deixar memòria sense alliberar (objectes dinàmics sense esborrar) → memory leaks
- Accedir a memòria ja alliberada (objectes esborrats).
   Vigileu amb l'aliasing → dangling references
- delete de memòria no creada amb new

- Deixar memòria sense alliberar (objectes dinàmics sense esborrar) → memory leaks
- Accedir a memòria ja alliberada (objectes esborrats).
   Vigileu amb l'aliasing → dangling references
- delete de memòria no creada amb new
- confondre "p = nullptr" amb "delete p"
  - els dos s'hauran d'usar, però en circumstàncies diferents

### Exemples d'errors

```
void f(...) {
 T x; ...
 T*p = &x;
 T \star q = new T;
 T* r = q; // r i q apunten al mateix valor
  T* s = new T;
  delete p; // ERROR: *p no creat amb new
  delete q: // OK
  if ((q->camp1 == 0) {...} // ERROR: q indefinit
  if (q == nullptr) {...} // PERILL: q indefinit
  r->camp1 = 3; // ERROR: r indefinit, *r
                  // alliberat amb delete q
  // ERROR: no fem delete s i *s inaccessible: leak!
```

### Vectors d'apuntadors

Els apuntadors permeten moure objectes més eficientment.

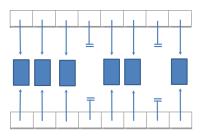
```
void moure(vector<Estudiant*>& v, vector<Estudiant*>& w) {
   for (int i = 0; i < v.size(); ++i) {
      w[i] = v[i];
      v[i] = nullptr;
   }
}</pre>
```

### Vectors d'apuntadors

Els apuntadors permeten moure objectes més eficientment.

```
void moure(vector<Estudiant*>& v, vector<Estudiant*>& w) {
   for (int i = 0; i < v.size(); ++i) {
      w[i] = v[i];
      v[i] = nullptr;
   }
}</pre>
```

#### Si no posem nullptrs en v tenim:



## Assignació, còpia, destrucció

#### Còpia:

- Assignació entre apuntadors a objectes no implica una còpia d'objectes
- Fonamental definir constructora per còpia per al corresponent tipus. Usarà new. La constructora per còpia per defecte crea un nou object a partir d'un altre, copiant atribut a atribut.
- També sovint es redefineix també l'operació =, per defecte fa assignació atribut a atribut de l'objecte origen a l'objecte destí.
  - Copiar/assignar atributs que siguin punters → aliasing!

# Assignació, còpia, destrucció

#### Esborrament:

- La destructora per defecte destruirà atributs que siguin punters però no els objectes als quals apuntin! → memory leaks
- Cal definir la destructora de la classe de manera que s'alliberi tots els objects creats a meòria dinàmica per a representar un object de la classe

# Pas d'apuntadors com a paràmetres

Pas d'un objecte *X* que conté apuntadors com a paràmetre d'entrada:

- Pas per valor:
  - Fa servir la constructora per còpia, hem definirla si la constructora per còpia per defecte no serveix
  - Si l'objecte X té atributs que són punters, la constructora per còpia per defecte ens porta a una situació d'aliasing
- Pas per referència: passem X no es canviarà, però no es garanteix que no es modifiquin objectes apuntats per components de X

# Part VII

# Tipus Recursius de Dades

- (29) Apu tadors i memòria dinàmica
- 30 Tipus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35) Albrea Nais
  - 36 Arbres generals
  - (37) Implementació de mètodes: accedint la representació
  - 38) Estructures de dades noves

### Tipus recursius de dades?

Els programes són dades més operacions (p.ex., accions o funcions)

Hem vist accions i funcions recursives: casos directes + casos recursius

Té sentit parlar de tipus de dades recursius?

De fet, hem pensat en alguns tipus de manera recursiva:

- Piles: una pila o bé és buida o bé és push(una altra pila,valor)
- Cues, Ilistes: idem
- Arbres: un arbre, o bé és buit o bé és plantar(valor,arbre1,arbre2)

De fet, hem pensat en alguns tipus de manera recursiva:

- Piles: una pila o bé és buida o bé és push(una altra pila,valor)
- Cues, Ilistes: idem
- Arbres: un arbre, o bé és buit o bé és plantar(valor,arbre1,arbre2)

Només n'hem vist algunes implementacions amb vectors, no recursives

De fet, hem pensat en alguns tipus de manera recursiva:

- Piles: una pila o bé és buida o bé és push(una altra pila,valor)
- Cues, Ilistes: idem
- Arbres: un arbre, o bé és buit o bé és plantar(valor,arbre1,arbre2)

Només n'hem vist algunes implementacions amb vectors, no recursives

Una definició recursiva d'aquests tipus de dades podria donar:

- Correspondència natural amb definició recursiva
- No posar límits a priori en la mida

Implementació en C++??

```
class stack<T> {
  private:
    bool es_buida;
    T valor;
    stack<T> resta_pila;
  public:
    ...
};
```

Problema: En C++, quan es crea un objecte es crida recursivament a les creadores de totes les seves components. Procés infinit

#### Com es fa: la Pila

```
template <class T> class stack {
 private:
   // tipus privat nou
   struct node_pila {
      T info;
       node_pila* seq; // <-- recursivitat</pre>
   };
   node_pila* cim; // primer d'una cadena de nodes
   ... // especificació d'operacions privades
 public:
   ... // especificació d'operacions públiques
};
```

#### Com es fa: la Pila

```
template <class T> class stack {
 private:
   // tipus privat nou
   struct node pila {
       T info:
       node pila* seg: // <-- recursivitat
   };
   int altura;
                           // guardada un sol cop
   node_pila* cim; // primer d'una cadena de nodes
    ... // especificació d'operacions privades
 public:
    ... // especificació d'operacions públiques
};
```

Els apuntadors seg no s'inicialitzen automàticament: no es creen objectes recursivament quan es crea un stack

#### Definició d'una estructura de dades recursiva I

#### Dos nivells:

- Superior: classe amb atributs
  - Informació global de l'estructura (que no volem que es repeteixi per a cada element)
  - Apuntadors a alguns elements distingits (el primer, l'últim, etc., segons el que calgui).
- Inferior: struct privada que defineix nodes enllaçats per apuntadors
  - informació d'un i només un element de l'estructura
  - apuntador a un o més nodes "següents"

#### Avantatges de les estructures de dades recursives

 Correspondència natural amb una definició recursiva abstracta

#### Avantatges de les estructures de dades recursives

- Correspondència natural amb una definició recursiva abstracta
- No cal fixar a priori un nombre màxim d'elements
- Es pot anar demanant memòria per als nous nodes a mesura que s'hi volen afegir elements

#### Avantatges de les estructures de dades recursives

- Correspondència natural amb una definició recursiva abstracta
- No cal fixar a priori un nombre màxim d'elements
- Es pot anar demanant memòria per als nous nodes a mesura que s'hi volen afegir elements
- Eficiència: modificant enllaços entre nodes podem:
  - inserir o esborrar elements sense moure els altres
  - moure parts senceres de l'estructura sense fer còpies

# Part VII

# Tipus Recursius de Dades

- (29) Apu ytadors i memòria dinàmica
- 30 Prous recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de Ilistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35 Albre Wais
  - (36) Arbres generals
  - 37 Implementació de mètodes: accedint la representació
  - 38) Estructures de dades noves

#### Implementació de piles

```
template <class T> class stack {
 private:
   // tipus privat nou
   struct node_pila {
       T info;
       node_pila* seq; // nullptr indica final de cadena
   };
   int altura;
                           // guardada un sol cop
   node_pila* cim; // element en el cim de la pila
    ... // especificació d'operacions privades
 public:
    ... // especificació d'operacions públiques
};
```

#### Mètodes públics: construcció/destrucció

```
stack() {
    altura = 0;
    cim = nullptr;
}

// Constructora per copia
stack(const stack& original) {
    altura = original.altura;
    cim = copia_node_pila(original.cim);
}
```

# Mètodes públics: construcció/destrucció, modificació

```
~stack() {
    esborra_node_pila(cim);
}

void clear() {
    esborra_node_pila(cim);
    altura = 0;
    cim = nullptr;
}
```

#### Mètodes públics: consultors

```
T top() const {
// Pre: el p.i. és una pila no buida
// = en termes d'implementacio, cim != nullptr
    return cim -> info;
bool empty() const {
    return cim == nullptr;
int size() const {
    return altura;
```

#### Mètodes públics: modificadors

```
void push(const T& x) {
   node_pila* aux = new node_pila; // espai per al nou element
   aux -> info = x;
   aux -> seg = cim
   cim = aux;
   ++altura;
}
```

#### Mètodes públics: modificadors

```
void pop() {
  // Pre: el p.i. és una pila no buida
  // => cim != nullptr
   node_pila* aux = cim; // conserva l'accés a primer
   cim = cim -> seg; // avança
   delete aux; // allibera l'espai de l'antic cim
   --altura;
}
```

#### Mètodes privats I

```
static node_pila* copia_node_pila(node_pila* m) {
/* Pre: cert. */
/* Post: si m és nullptr, el resultat és nullptr; en cas contrari
         el resultat apunta al primer node d'una cadena
         de nodes que són còpia de la cadena que té
         el node apuntat per m com a primer */
    if (m == nullptr) return nullptr;
    else {
        node_pila* n = new node_pila;
        n \rightarrow info = m \rightarrow info;
        n -> seg = copia node pila(m -> seg);
        return n;
```

Exercici: Versió iterativa

# Mètodes privats II

Exercici: Versió iterativa

# Mètodes públics: redefinició operador assignació

```
stack<int> p1, p2, p3;
...
p1 = p2 = p3;
```

# Mètodes públics: redefinició operador assignació

```
stack<int> p1, p2, p3;
...
p1 = p2 = p3;
```

L'assignació en C++ és un operador: una funció que retorna un valor, amb paràmetre implícit que queda modificat, i un paràmetre explícit no modificable

# Mètodes públics: redefinició operador assignació

```
stack<int> p1, p2, p3;
...
p1 = p2 = p3;
```

L'assignació en C++ és un operador: una funció que retorna un valor, amb paràmetre implícit que queda modificat, i un paràmetre explícit no modificable

Return (\*this): necessari per a encadenaments d'assignacions

```
stack& operator=(const stack& original) {
   if (this != &original) {
      node_pila* aux = copia_node_pila(original.cim);
      esborra_node_pila(cim); // si no, leak!
      altura = original.altura;
      cim = aux;
   }
   return *this;
}
```

#### Implementació de cues

- Cal poder accedir tant tant al primer element (per consultar-lo o eliminar-lo) com a l'últim (per afegir un de nou)
- Atribut per la llargada (o mida) de la cua

#### Definició de la classe

```
template <class T> class queue {
 private:
    struct node cua {
        T info;
        node_cua* seg;
    };
    int longitud;
    node_cua* primer;
    node_cua* ultim;
    ... // especificació i implementació d'operacions privades
 public:
    ... // especificació i implementació d'operacions públiques
};
```

#### Mètodes privats: copiar i esborrar cadenes I

```
static node_cua* copia_node_cua(node_cua* m, node_cua*& u) {
/* Pre: cert. */
/* Post: si m és nullptr, el resultat i u són nullptr; en cas contrari
   el resultat apunta al primer node d'una cadena de nodes
   que són còpia de de la cadena que té el node apuntat per m
   com a primer, i u apunta a l'últim node */
    if (m == nullptr) { u = nullptr; return nullptr; }
    else {
        node cua* n = new node cua;
        n \rightarrow info = m \rightarrow info;
        n -> seg = copia_node_cua(m- > seg, u);
        if (n -> seq == nullptr) u = n;
        return n;
```

# Mètodes privats: copiar i esborrar cadenes II

#### Mètodes privats: construccció/destrucció

```
queue() {
    longitud = 0;
    primer = ultim = nullptr;
queue (const queue& original)
    longitud = original.longitud;
    primer = copia_node_cua(original.primer, ultim);
~queue() {
    esborra node cua(primer);
```

# Mètodes públics: redefinició de l'operador d'assignació

```
queue& operator=(const queue& original) {
   if (this != &original) {
      node_cua* auxp, *auxu;
      auxp = copia_node_cua(original.primer, auxu);
      esborra_node_cua(primer); // si no, leak!
      longitud = original.longitud;
      primer = auxp;
      ultim = auxu;
   }
   return *this;
}
```

#### Mètodes públics: modificadors I

```
void clear() {
    esborra_node_cua(primer);
    longitud = 0;
    primer_node = nullptr;
    ultim_node = nullptr;
void push (const T& x) {
    node_cua* aux = new node_cua;
    aux -> info = x;
    aux -> seg = nullptr;
    if (primer == nullptr) primer = aux;
    else ultim -> seq = aux;
    ultim = aux;
    ++longitud;
```

#### Mètodes públics: modificadors I

```
void pop() {
// Pre: el p.i. és una cua no buida
// = en termes d'implementacio, primer != nullptr
    node_cua* aux = primer;
    if (primer == ultim) {
        primer = ultim = nullptr;
    } else primer = primer -> seg;
    delete aux;
    --longitud;
}
```

#### Mètodes públics: consultors

```
T front() const {
// Pre: el p.i. és una cua no buida
// = en termes d'implementacio, primer != nullptr
    return primer -> info;
bool empty() const {
    return longitud == 0;
int size() const {
    return longitud;
```

# Exemple d'increment d'eficiència

```
// Pre: cert
// Post: retorna cert si la cua conté x
bool cerca(const T& x) const {
   node_cua* aux = primer;
   while (aux != nullptr) {
      if (aux -> info == x) return true;
      aux = aux -> seg;
   }
   return false;
}
```

la cua és const &, no és destruida, no hi ha còpies

... però s'ha de tenir accés a la representació!

# Part VII

# Tipus Recursius de Dades

- (29) Apu tadors i memòria dinàmica
- 30 Jous recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35 Albre Wais
  - (36) Arbres generals
  - 37 Implementació de mètodes: accedint la representació
  - 38) Estructures de dades noves

#### Implementació de Llista

En aquest curs no implementarem els iteradors de manera general

Implementarem Ilistes amb punt d'interès

Funcionalitats similars, algunes restriccions

# Novetat tipus llista: punt d'interès

#### Podem:

- Desplaçar endavant i enrere el punt d'interès
- Afegir i eliminar just al punt d'interès
- Consultar i modificar l'element al punt d'interès

# Novetat tipus llista: punt d'interès

#### Podem:

- Desplaçar endavant i enrere el punt d'interès
- Afegir i eliminar just al punt d'interès
- Consultar i modificar l'element al punt d'interès
- Implementació: atribut (privat) de tipus apuntador a node
- Modularitat: punt d'interès part del tipus, no tipus apart
- Efecte lateral: queda modificat si es modifica en una funció que rep la llista per referència no const

# Definició classe Llista, I

```
template <class T> class Llista {
 private:
    struct node llista {
        T info:
        node llista* seq;
        node llista* ant:
    };
    int longitud;
    node_llista* primer;
    node llista* ultim;
    node_llista* act;  // apuntador a punt d'interes
    ... // especificació i implementació d'operacions privades
 public:
    ... // especificació i implementació d'operacions públiques
};
```

# Definició classe Llista, II

- Apuntadors per accés ràpid a següent, anterior, primer i darrer, i punt d'interès
- act == nullptr vol dir "punt d'interès sobre l'element fictici posterior a l'últim"
- Conveni Ilista buida: longitud zero i els tres apuntadors (primer, ultim i act) nuls
- Llista amb un element: longitud 1 i únic altre cas en què primer == ultim
- "cap a la dreta" == cap a l'últim; "cap a l'esquerra" == cap al primer; "a la dreta de tot" == sobre l'element fictici del final

## Constructures i destructora

```
Llista() {
 longitud = 0;
  primer = nullptr;
 ultim = nullptr;
  act = nullptr;
Llista(const Llista& original)
    longitud = original.longitud;
    primer = copia_node_llista(original.primer, original.act,
                                     ultim, act);
~Llista() {
    esborra_node_llista(primer);
```

# Copiar cadena de nodes

# Copiar cadena de nodes

```
static node_llista* copia_node_llista(
               node llista* m, node llista* oact,
               node llista*& u. node llista*& a) {
    if (m == nullptr) { u = nullptr; a = nullptr; return nullptr; }
    else {
        node_llista* n = new node_llista;
        n \rightarrow info = m \rightarrow info;
        n -> ant = nullptr;
        n -> seg = copia_node_llista(m -> seg, oact, u, a);
        if (n -> seg != nullptr) n -> seg -> ant = n;
        if (n -> seg == nullptr) u = n;
        // else, u es el que hagi retornat la crida recursiva
        // es podria fer com a "else"
        if (m == oact) a = n;
        // else, a es el que hagi retornat la crida recursiva
        return n:
```

#### Esborrar cadena de nodes

Exercici: La versió iterativa

# Redefinició de l'assignació

# Redefinició de l'assignació: una tècnica alternativa

```
// intercanvi de la llista implícita amb la llista aux
void Swap(LLista& aux) {
  swap(longitud, aux.longitud);
  swap(primer, aux.primer);
  swap(ultim, aux.ultim);
  swap(act, aux.act);
Llista& operator=(const Llista& original)
 Llista aux = original: // amb la construcció per còpia
  Swap (aux);
  return *this;
```

## Modificadores I

```
void l_buida() {
    esborra_node_llista(primer);
    longitud = 0;
    primer = nullptr;
    ultim = nullptr;
    act = nullptr;
}
```

## Modificadores II

```
void afegir(const T& x) {
/* Pre: cert. */
/* Post: la llista queda com originalment, però amb x
   afegit a l'esquerra del punt d'interès */
    node_llista* aux = new node_llista;
    aux \rightarrow info = x;
    aux -> seg = act;
    if (longitud == 0) { // la llista es buida
        aux -> ant = nullptr;
        primer = aux;
        ultim = aux;
    } else if (act == nullptr) {
        aux -> ant = ultim;
        ultim -> seg = aux;
        ultim = aux;
```

## Modificadores III

## (continuació)

```
else if (act == primer) {
    aux -> ant = nullptr;
    act -> ant = aux;
    primer = aux;
} else {
    aux -> ant = act -> ant;
    act -> ant -> seg = aux;
    act -> ant = aux;
}
++longitud;
}
```

#### Modificadores IV

```
void eliminar() {
/* Pre: la llista no és buida i el seu punt d'interès
        no és a la dreta de tot */
/* Post: la llista queda com originalment però sense l'element
         on estava el punt d'interès i amb el nou punt d'interès
         apuntant al successor de l'element esborrat */
    node llista* aux = act; // conserva l'accés al node actual
    if (longitud == 1) {
        primer = nullptr;
        ultim = nullptr;
    } else if (act == primer) {
        primer = act -> seq;
        primer -> ant = nullptr;
```

## Modificadores V

#### (continuació)

```
else if (act == ultim) {
    ultim = act -> ant;
    ultim -> seg = nullptr;
else {
    act -> ant -> seg = act -> seg;
    act -> seq -> ant = act -> ant;
act = act -> seg; // avança el punt d'interès
delete aux; // allibera l'espai de l'element esborrat
--longitud;
```

#### Modificadores VI

Interès: concatenació més eficient que la basada en afegir

```
void concat(Llista% 1) {
/* Pre: 1 = L */
/* Post: la llista conté els seus elements originals seguits pels
         de L, l queda buida, i el punt d'interés passa a ser el
         primer element */
    if (l.longitud > 0) { // l buida \rightarrow no cal fer res
        if (longitud == 0) {
            primer = l.primer;
        } else {
            ultim -> seg = l.primer;
            1.primer -> ant = ultim:
        ultim = 1.ultim;
        longitud += l.longitud;
        l.primer = l.ultim = l.act = nullptr; l.longitud = 0;
    act = primer;
```

## Consultores

```
bool es_buida() const {
    return primer == nullptr;
}
int mida() const {
    return longitud;
}
```

# Noves operacions per a consultar i modificar l'element actual

```
T actual() const { // equival a consultar *it
/* Pre: la llista no és buida i el seu punt d'interès
        no està sobre l'element fictici del final */
/* Post: el resultat és l'element apuntat pel punt d'interès */
    return act -> info;
void modifica_actual(const T &x) { // equival a fer *it = x
/* Pre: la llista no és buida i el seu punt d'interès no està
        a la dreta de tot*/
/* Post: la llista queda com originalment, però amb x reemplaçant
       l'element actual */
    act \rightarrow info = x;
```

# Noves operacions per a moure el punt d'interès I

```
void inici() { // equival a fer it = l.begin()
/* Pre: cert */
/* Post: el punt d'interès de la llista apunta al primer
   element de la llista, o a la dreta de tot si la llista és buida */
   act = primer;
void fi() {      // equival a fer it = l.end()
/* Pre: cert */
/* Post: el punt d'interès gueda situat
    sobre l'element fictici del final */
   act = nullptr;
```

# Noves operacions per a moure el punt d'interès II

```
void avanca() { // equival a fer ++it
/* Pre: el punt d'interès no està a la dreta de tot */
/* Post: el punt d'interès apunta al successor de l'element al qual
         apuntava originalment, és a dir es mou cap a la dreta
         del seu al valor original */
    act = act -> seq:
void retrocedeix() { // equival a fer --it
/* Pre: el punt d'interès no és el primer element de la llista */
/* Post: el punt d'interès apunta al predecessor de l'element al qual
         apuntava originalment, o apunta a l'últim element de la llist
         si estava apuntant a la dreta de tot; és a dir es mou cap a
         l'esquerra del seu al valor original */
    if (act == nullptr) act = ultim;
   else act = act -> ant;
```

# Noves operacions per a moure el punt d'interès III

```
bool dreta_de_tot() const { // equival a comparar it == 1.end()
/* Pre: cert. */
/* Post: retorna cert si i només si el punt d'interès
         és a la dreta de tot */
   return act == nullptr;
bool sobre_el_primer() const { // equival a comparar it == 1.begin()
/* Pre: cert */
/* Post: si la llista no és buida, retorna cert si i només si
         el punt d'interès és damunt el primer element; si la llista
         és buida retorna cert si i només si
         punt d'interès si està a la dreta de tot */
   return act == primer;
```

# Part VII

# Tipus Recursius de Dades

- (29) Apu ytadors i memòria dinàmica
- 30 Hous recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- Arbres binaris

- 35) Albrea Nais
  - 36 Arbres generals
  - (37) Implementació de mètodes: accedint la representació
  - 38) Estructures de dades noves

### Llistes doblement encadenades amb sentinella

#### Implementació de llistes amb sentinella:

- Node extra; no conté cap element real
- Objectiu: simplificar el codi d'algunes operacions com ara afegir i eliminar
- L'estructura mai té apuntadors amb valor nullptr. El sentinella fa el paper que tenien aquests

## Llistes amb sentinella

#### Llista buida:

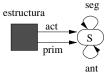
• següent i anterior del sentinella = sentinella

#### Llista no buida:

- següent del sentinella = primer de la llista
- anterior del sentinella = darrer de la llista
- sentinella = anterior del primer
- sentinella = següent del darrer

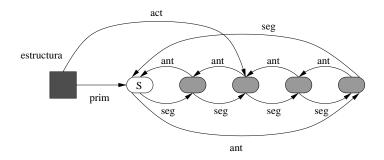
# Llistes amb sentinella

#### Llista buida:



# Esquema estructura interna llistes doblement encadenades amb sentinella

#### Llista no buida:



# Un nou atribut privat

sent apunta sempre al node sentinella, que existeix fins i tot quan la llista és buida

```
template <class T> class Llista {
    private:
      struct node llista {
          T info;
          node llista* seq;
          node llista* ant;
      };
      int longitud;
      node_llista* sent;
      node_llista* act;
      ... // especificació i implementació d'operacions privades
    public:
      ... // especificació i implementació d'operacions públiques
};
```

Implementació de privades i públiques → apunts

# Part VII

# Tipus Recursius de Dades

- (29) Apu ytadors i memòria dinàmica
- 30 Plous recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35 Albred Walis
  - 36 Arbres generals
  - (37) Implementació de mètodes: accedint la representació
  - 38) Estructures de dades noves

#### Definició de la classe Arbre

No coincideix amb la classe BinTree Principals operacions:

- a.plantar(x,a1,a2): Demana que a sigui buit, i sigui objecte diferent d'a1 i a2. Deixa a1 i a2 buits.
- a.fills (a1, a2): Demana que a1 i a2 siguin buits, i tots tres objectes a, a1 i a2 han de ser objectes diferents.

#### Definició de la classe Arbre

No coincideix amb la classe BinTree Principals operacions:

- a.plantar(x, a1, a2): Demana que a sigui buit, i sigui objecte diferent d'a1 i a2. Deixa a1 i a2 buits.
- a.fills (a1, a2): Demana que a1 i a2 siguin buits, i tots tres objectes a, a1 i a2 han de ser objectes diferents.

Això fa que per recorrer un arbre s'hagi de "desmuntar". Sovint ineficient.

Inconvenient solucionat a BinTree amb *smart pointers* de C++, que no són part de l'assignatura.

## Definició de la classe Arbre

- struct del node conté dos apuntadors a node
- Arbre buit = atribut arrel és nul

```
template <class T> class Arbre {
    private:
        struct node_arbre {
            T info;
            node_arbre* esq;
            node_arbre* dre;
        };
        node_arbre* arrel;
        ... // especificació i implementació d'operacions privades
    public:
        ... // especificació i implementació d'operacions públiques
};
```

#### Constructores i destructora

```
Arbre() {
/* Pre: cert */
/* Post: crea un arbre buit */
    arrel = nullptr;
Arbre (const Arbre & original) {
/* Pre: cert. */
/* Post: crea un arbre que és una còpia d'original */
    arrel = copia_node_arbre(original.arrel);
~Arbre() {
    esborra node arbre(arrel);
```

# Copiar jerarquies de nodes

```
static node_arbre* copia_node_arbre(node_arbre* m) {
/* Pre: cert */
/* Post: el resultat és nullptr si m és nullptr; si no, el resultat ap
        al node arrel d'una jerarquia de nodes que és una còpia de
        la jerarquia de nodes que té el node apuntat per m com a arrel
    if (m == nullptr) return nullptr;
    else {
        node arbre* n = new node arbre;
        n \rightarrow info = m \rightarrow info;
        n -> esq = copia_node_arbre(m -> esq);
        n -> dre = copia_node_arbre(m -> dre);
        return n;
```

Notem l'operador = del tipus T usat com a una operació de còpia

# Esborrar jerarquies de nodes

## Operador d'assignació i modificadores I

```
Arbre& operator=(const Arbre& original) {
    if (this != &original) {
      node_arbre* aux = copia_node_arbre(original.arrel);
      esborra_node_arbre(arrel);
      arrel = aux;
    return *this;
void a_buit() {
    esborra node arbre(arrel);
    arrel = nullptr;
```

### Modificadores II

```
void plantar(const T &x, Arbre &a1, Arbre &a2) {
/* Pre: l'arbre implícit és buit, al = A1, a2 = A2,
   al i a2 són objectes diferents de l'arbre implícit */
/* Post: l'arbre implícit té x com a arrel, Al com a fill esquerre
         i A2 com a fill dret; a1 i a2 són buits */
    node_arbre* aux = new node_arbre;
    aux \rightarrow info = x:
    aux -> esq = a1.arrel;
    if (a2.arrel != a1.arrel or a2.arrel == nullptr)
       aux -> dre = a2.arrel;
    else
      aux -> dre = copia node arbre(a2.arrel);
    arrel = aux;
    al.arrel = nullptr;
    a2.arrel = nullptr:
```

### Modificadores III

#### Consultores

```
T arrel() const {
/* Pre: l'arbre no és buit */
/* Post: retorna el valor de l'arrel de l'arbre */
    return arrel -> info;
}
bool es_buit() const {
/* Pre: cert */
/* Post: retorna cert si i només si l'arbre és buit */
    return arrel == nullptr;
}
```

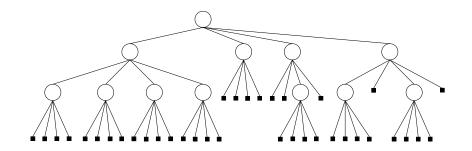
# Part VII

# Tipus Recursius de Dades

- (29) Apu tadors i memòria dinàmica
- 30 Plus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

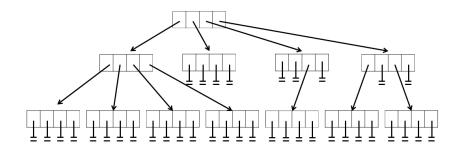
- 35 Arbres N-aris
  - 36 Arbres generals
  - 37 Implementació de mètodes: accedint la representació
  - 38 Estructures de dades noves

## Arbres N-aris



- Generalització dels arbres binaris
- N: nombre de fills (binaris: *N*=2)

# Arbres N-aris: Implementació



un node conté vector anb N apuntadors a node, un per a cada fill operació "consultar i-èssim" eficient: accés directe

#### Definició classe ArbreNari

## Copiar jerarquies de nodes

```
static node arbreNari* copia node arbreNari(node arbreNari* m) {
/* Pre: cert */
/* Post: el resultat és nullptr si m és nullptr; en cas contrari,
         el resultat apunta al node arrel d'una jerarquia de nodes
         que és una còpia de la jerarquia de nodes que té el node
         apuntat per m com a arrel */
    if (m == nullptr) return nullptr;
    else {
        node_arbreNari* n = new node_arbreNari;
        n \rightarrow info = m \rightarrow info;
        int N = m -> child.size();
        n -> child = vector<node arbreNari *> (N);
        for (int i = 0; i < N; ++i)
            n -> child[i] = copia node arbreNari(m -> child[i]);
        return n:
```

Conté recursió i iteració

## Esborrar jerarquies de nodes

### Constructores/destructora I

```
ArbreNari(int n) {
/* Pre: cert. */
/* Post: l'arbre implícit és un arbre buit d'aritat n */
   N = n;
   root = nullptr;
ArbreNari(const T& x, int n) {
/* Pre: cert */
/* Post: l'arbre implícit és un arbre amb arrel x i
         n fills buits */
    N = n;
    root = new node arbreNari;
    root \rightarrow info = x;
    root -> child = vector<node_arbreNari*>(N, nullptr);
```

## Constructores/destructora II

```
ArbreNari(const ArbreNari& original) {
/* Pre: cert */
/* Post: el resultat és una arbre còpia d'original */
    N = original.N;
    root = copia_node_arbreNari(original.root);
}

~ArbreNari() {
    esborra_node_arbreNari(root);
}
```

### Modificadores I

```
/* Pre: l'arbre implícit té la mateixa aritat que original */
/* Post: l'arbre implícit és una còpia d'original */
ArbreNari& operator=(const ArbreNari& original) {
   if (this != &original) {
      node_ArbreNari* aux = copia_node_arbreNari(original.root);
      esborra_node_arbreNari(root);
      root = aux;
   }
   return *this;
}
```

### Modificadores II

```
void plantar(const T& x, vector<ArbreNari>& v) {
/* Pre: l'arbre implícit és buit, v = V, v.size() és l'aritat
         de l'arbre implícit, tots els components de v tenen la
         mateixa aritat que l'arbre implícit, i tots són objectes
         diferents entre sí i diferents de l'arbre implícit */
/* Post: l'arbre implícit té x com a arrel i els seus fills són iquals
         que els components de V: v conté arbres buits */
    root = new node arbreNari;
    root \rightarrow info = x:
    root -> child = vector<node arbreNari*>(N);
    for (int i = 0; i < N; ++i) {</pre>
        root -> child[i] = v[i].root;
        v[i].root = nullptr;
```

### Modificadores III

```
void fill(const ArbreNari& a, int i) {
/* Pre: l'arbre implícit és buit i de la mateixa aritat que a,
   a no és buit, i està entre 1 i el nombre de fills d'a */
/* Post: l'arbre implícit és una còpia del fill i-èssim d'a */
   root = copia node arbreNari(a.root -> child[i-1]);
void fills(vector<ArbreNari>& v) {
/* Pre: l'arbre implícit és A. un arbre no buit.
        v és un vector buit */
/* Post: v conté els fills d'A i l'arbre implícit és buit */
   v = vector<ArbreNari>(N, ArbreNari(N));
   for (int i = 0; i < N; ++i)
        v[i].root = root -> child[i];
   delete root;
   root = nullptr;
```

#### Consultores

```
T arrel() const {
/* Pre: l'arbre implícit no és buit */
/* Post: el resultat és el valor a l'arrel de l'arbre implícit */
    return root -> info;
bool es buit() const {
/* Pre: cert. */
/* Post: el resultat indica si l'arbre implícit és un arbre buit
    return root == nullptr;
int aritat() const {
/* Pre: cert */
/* Post: el resultat és l'aritat de l'arbre implícit */
    return N;
```

## Eficiència de recorreguts arbres N-aris

Observació: fills ens permet recorreguts eficients

- fills té cost N, siguin els subarbres molt grans o molt petits
- No hi ha còpia d'arbres

Podria fer-se copiant cada fill amb fill, però és ineficient

### Suma de tots elements d'un arbre

```
/* Pre: a = A */
/* Post: el resultat és la suma dels elements d'A */
int suma(ArbreNari<int>& a) {
    if (a.es_buit()) return 0;
    else {
        int s = a.arrel();
        int N = a.aritat();
        vector< ArbreNari<int> > v:
        a.fills(v);
        for (int i = 0; i < N; ++i) s += suma(v[i]);</pre>
        return s;
```

### Sumar un valor k a cada node d'un arbre

```
/* Pre: a = A */
/* Post: a és com A però havent sumat k a tots els seus elements */
void suma_k(ArbreNari<int>& a, int k) {
    if (not a.es_buit()) {
        int s = a.arrel() + k;
        int N = a.aritat();
        vector<ArbreNari<int> > v;
        a.fills(v);
        for (int i = 0; i < N; ++i) suma_k(v[i], k);
        a.plantar(s, v);
    }
}</pre>
```

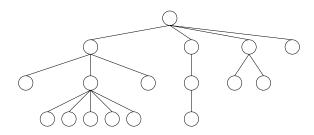
# Part VII

# Tipus Recursius de Dades

- (29) Apu tadors i memòria dinàmica
- 30 Plus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35 Arbre
  - 36 Arbres generals
  - (37) Implication ció de mètodes: accedint la representació
  - 38) Estructures de dades noves

# Arbres generals I



- Nombre indeterminat de fills, no necessàriament el mateix a cada subarbre
- Propietat important: Un arbre general
  - o és l'arbre buit
  - o té qualsevol nombre (fins i tot zero) de fills, cap dels quals és buit

## Arbres generals II. Implementacions

- vector d'apuntadors de mida = nombre de fills
  - "consultar i-èssim" eficient
  - "eliminar fill i-èssim" potser és ineficient
  - (que no existeix en arbres *N*-aris!)
  - és la implementació que descriurem

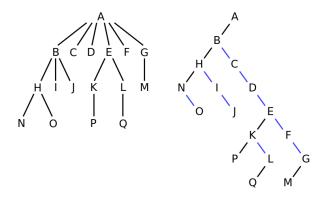
## Arbres generals II. Implementacions

- vector d'apuntadors de mida = nombre de fills
  - "consultar *i*-èssim" eficient
  - "eliminar fill i-èssim" potser és ineficient
  - (que no existeix en arbres *N*-aris!)
  - és la implementació que descriurem
- Ilista d'apuntadors a fills
  - "consultar i-èssim" ineficient (accés seqüencial)
  - però ok per recorreguts seqüencials
  - "eliminar fill actual" eficient

## Arbres generals II. Implementacions

- vector d'apuntadors de mida = nombre de fills
  - "consultar *i*-èssim" eficient
  - "eliminar fill i-èssim" potser és ineficient
  - (que no existeix en arbres *N*-aris!)
  - és la implementació que descriurem
- Ilista d'apuntadors a fills
  - "consultar i-èssim" ineficient (accés seqüencial)
  - però ok per recorreguts seqüencials
  - "eliminar fill actual" eficient
- arbre binari "primer fill, germà dret"
  - reimplementació sobre arbres binaris

## "primer fill, germà dret": exemple



(font: http://en.wikipedia.org/wiki/Left-child\_
right-sibling\_binary\_tree)

### Definició de la classe ArbreGen

```
template <class T> class ArbreGen
    private:
        struct node_arbreGen {
            T info;
            vector<node_arbreGen*> child;
        };
        node_arbreGen* root;
            ... // operacions privades
    public:
        ... // operacions públiques
};
```

Important: Ja no tenim un atribut amb el nombre de fills per a tot l'arbre; ni per a cada node. Es pot obtenir amb child.size()

# Copiar i esborrar jerarquies de nodes

Idèntiques a les dels arbres N-aris (només canviar tipus dels nodes)

### Constructores/destructores I

```
ArbreGen() {
/* Pre: cert. */
/* Post: l'arbre implícit és un arbre general buit */
    root = nullptr;
ArbreGen (const T &x) {
/* Pre: cert. */
/* Post: l'arbre implícit és un arbre general amb arrel x
         i 0 fills */
    root = new node arbreGen;
    root \rightarrow info = x;
    // no cal fer arrel -> child = vector<node arbreGen*>(0);
```

## Constructores/destructores II

```
ArbreGen(const ArbreGen& original) {
/* Pre: cert */
/* Post: el resultat és una arbre còpia d'original */
   root = copia_node_arbreGen(original.root);
}

~ArbreGen() {
   esborra_node_arbreGen(root);
}
```

#### Modificadores I

```
ArbreGen& operator=(const ArbreGen& original) {
  if (this != &original) {
    node_ArbreGen* aux = copia_node_arbreGen(original.root);
    esborra_node_arbreGen(root);
    root = aux;
  return *this;
void a buit() {
/* Pre: cert */
/* Post: l'arbre implícit és un arbre general buit */
    esborra_node_arbreGen(root);
   root = nullptr;
```

#### Modificadores II

```
void plantar(const T &x) {
/* Pre: l'arbre implícit és buit */
/* Post: l'arbre implícit té x com a arrel i sense fills */
    root = new node arbreGen;
       // inclou un root -> child = vector<node_arbreGen*>(0);
   root \rightarrow info = x:
void plantar(const T &x, vector<ArbreGen> &v) {
/* Pre: l'arbre implícit és buit, v = V, cap component
         de v és un arbre buit */
/* Post: l'arbre implícit té x com a arrel i els elements de V
         com a fills: v conté només arbres buits */
    root = new node arbreGen;
    root \rightarrow info = x;
    int n = v.size();
    root -> child = vector<node arbreGen*>(n);
    for (int i = 0; i < n; ++i) {
        root -> child[i] = v[i].root;
        v[i].root = nullptr;
```

### Modificadores III

Nota: aquí necessitem fer push\_back (...)

#### Modificadores IV

```
void fill(const ArbreGen& a, int i) {
/* Pre: l'arbre implícit és buit, a no és buit, i està entre 1 i
   el nombre de fills d'a */
/* Post: l'arbre implícit és una còpia del fill i-èssim d'a */
   root = copia node arbreGen(a.root -> child[i-1]);
void fills(vector<ArbreGen> &v) {
/* Pre: l'arbre implícit no és buit, li diem A, i no és cap
        dels components de v*/
/* Post: l'arbre implícit és buit, v passa a contenir els fills
        de l'arbre A */
    int n = root -> child.size();
   v = vector<ArbreGen>(n);
    for (int i = 0; i < n; ++i) v[i].root = root -> child[i];
   delete root; root = nullptr;
```

#### Consultores

```
T arrel() const {
/* Pre: l'arbre implícit no és buit */
/* Post: el resultat és el valor de l'arrel de l'arbre implícit
    return root -> info;
bool es buit() const {
/* Pre: cert. */
/* Post: el resultat indica si l'arbre implícit és un arbre buit
    return root == nullptr;
int nombre fills() const {
/* Pre: l'arbre implicit no és buit */
/* Post: el resultat és el nombre de fills de l'arbre implícit */
    return root -> child.size();
```

### Exemple: suma de tots els elements

```
int suma(ArbreGen<int>& a) {
/* Pre: a = A */
/* Post: el resultat és la suma dels elements d'A */
    int s:
    if (a.es buit()) s = 0;
    else {
      s = a.arrel();
      vector<ArbreGen<int> > v;
       a.fills(v);
       int n = v.size();
       for (int i = 0; i < n; ++i) s += suma(v[i]);</pre>
    return s;
```

### Exemple: sumar k a cada element

```
void suma k(ArbreGen<int>& a, int k) {
/* Pre: a = A */
/* Post: a és com A però havent sumat k a tots els seus elements
    if (not a.es buit()) {
        int s = a.arrel() + k;
        vector<ArbreGen<int> > v:
        a.fills(v);
        int n = v.size();
        // si n == 0, el bucle no fa res i es planta v que és
        // un vector buit d'ArbreGen. és a dir, la nova arrel conté
        // a.arrel() + k, i no tindrà cap fill, com originalment
        for (int i = 0; i < n; ++i) suma_k(v[i], k);</pre>
        a.plantar(s, v);
```

## Part VII

# Tipus Recursius de Dades

- (29) Apu tadors i memòria dinàmica
- 30 Plus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35 Albres A
  - 36) Arbres ge
  - 37 Implementació de mètodes: accedint la representació
  - (38) Estructures de dades noves

### Implementacions amb accés a la representació

- Avantatge: Eficiència. Assignació d'apuntadors vs. còpia d'estructures
- Inconvenient: Lligades a una representació. No modulars
- Exemple: sort com a mètode de la classe list a STL

### Cerca d'un element en una pila

```
class Pila {
    ...
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const;
    ...
};
```

### Cerca en una pila: versió iterativa

```
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const {
   node_pila* act = cim;
   /* Inv: cap node entre [cim, act) té info = x */
   while (act != nullptr) {
      if (act -> info == x) return true;
       act = act -> seguent;
   }
   return false;
}
```

### Cerca en una pila: versió recursiva I

```
class Pila {
    ...
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const;
    ...
};
```

Problema: La recursió és (node  $\rightarrow$  node), no (pila  $\rightarrow$  pila)!

## Cerca en una pila: versió recursiva I

```
class Pila {
    ...
/* Pre: cert */
/* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const;
    ...
};
```

Problema: La recursió és (node  $\rightarrow$  node), no (pila  $\rightarrow$  pila)!

 $\begin{array}{l} \textbf{Immersió:} \rightarrow \textbf{operació auxiliar, recursiva, amb paràmetre} \\ \texttt{node\_pila*} \end{array}$ 

la crida inicial fa el pas (pila → node\_pila\*)

### Cerca en una pila: versió recursiva II

```
/* Pre: cert */
  /* Post: retorna cert ssi x apareix a la pila implícita */
bool cerca(const T &x) const {
    return cerca_pila_node(cim, x);
}

/* Pre: cert */
/* Post: retorna cert ssi x apareix a la llista
    de nodes que comença a n */
static bool cerca_pila_node(node_pila* n, const T &x);
```

Atenció a l'static!

### Cerca en una pila: versió recursiva III

Compte: precondició de l'operador ->

#### Sumar un valor a tots els elements d'un arbre binari

El plantegem com a nou mètode de la classe arbre binari

```
/* Pre: A és el valor inicial del 'arbre implícit */
/* Post: l'arbre implícit és l'arbre A però havent sumat k
a tots els seus elements */
void inc_arbre(const T& k);
...
```

#### Sumar un valor a tots els elements d'un arbre binari

```
/* Pre: A és el valor inicial del arbre implícit */
/* Post: l'arbre implícit és l'arbre A però havent sumat k
         a tots els seus elements */
void inc arbre(const T& k) {
    inc node (a.arrel, k);
/* Pre: cert */
/* Post: el node apuntat per n i tots els seus descendents tenen
         al camp info la suma de k i el seu valor original */
static void inc node(node arbre* n, int k) {
    if (n != nullptr) {
        n \rightarrow info += k:
        inc node (n \rightarrow esq, k);
        inc node(n -> dre, k);
```

### Substitució de fulles per un arbre I

Substituir totes les fulles de l'arbre implícit que continguin el valor x per un altre arbre donat as

```
/* Pre: A es el valor inicial del p.i. */
/* Post: l'arbre és com A però havent substituït
    les fulles que contenien x per l'arbre as */
void subst(const T& x, const ArbreBin<T>& as);
```

### Substitució de fulles per un arbre II

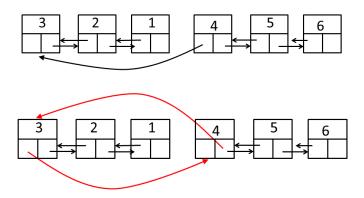
## Substitució de fulles per un arbre III

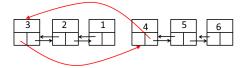
```
static node_arbre* subst_node(node_arbre* n, const T& x,
                                  const ArbreBin<T>& as) {
    if (n == nullptr) return nullptr;
    // n != nullptr
    if (n -> info == x and
         n -> esg == nullptr and n -> dre == nullptr) {
      ^{\prime\prime} n apunta a una fulla que conté el valor x
      delete n; // no cal fer esborra_node_arbre(n);
      n = copia_node_arbre(as.arrel);
    } else {
       n \rightarrow esq = subst_node(n \rightarrow esq, x, as);
       n \rightarrow dre = subst node(n \rightarrow dre, x, as);
    return n:
```

Atenció al retorn de l'apuntador a l'arrel de l'arbre resultant. L'alternativa és passar n per referència

- 1. Solució amb ops. de la classe: insert, còpies de node ...
- 2. Solució tocant representació: assignacions d'apuntadors

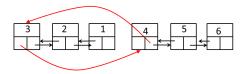
Simular "esborrar el primer de I1, afegir-lo primer a I2"





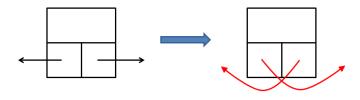
```
void revessar() {
  node_llista* n = primer;
  while (n != ultim) {
    /* Inv: per tots els nodes anteriors al que apunta n,
        els apuntadors a anterior i següent han estat
        intercanviats respecte a l'original */
        node_llista* suc = n -> seg;
        n -> seg = n -> ant;
        n -> ant -> ant = n;
        n = suc;
    }
    ... // continua
}
```

#### Revessar una llista, v1



```
if (n != nullptr and n != primer) {
   // n == ultim != primer (i la llista
   // no és buida!)
   n -> seg = n->ant;
   n -> ant = nullptr;
   ultim = primer;
   primer = n;
}
```





```
void revessar() {
  node_llista* n = primer;
  while (n != nullptr) {
    /* Inv: per als nodes anteriors al que apunta n,
        els apuntadors a anterior i seguent han estat
        intercanviats respecte a l'original */
        swap(n -> seg, n -> ant);
        n = n -> ant;
    }
    swap(primer, ultim);
}
```



```
void revessar() {
   node_llista* n = primer;
   while (n != nullptr) {
     /* Inv: per als nodes anteriors al que apunta n,
        els apuntadors a anterior i seguent han estat
        intercanviats respecte a l'original */
        swap(n -> seg, n -> ant);
        n = n -> ant;
   }
   swap(primer, ultim);
}
```

Exercici: versió recursiva

## Part VII

# Tipus Recursius de Dades

- (29) Apu ytadors i memòria dinàmica
- 30 Plus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris

- 35 A bre W
  - (36) Arbres geserals
  - 37 Implemento ció de mètodes: accedint la representació
  - 38 Estructures de dades noves
    - Cues ordenades
    - Multillistes

### Part VII

## Tipus Recursius de Dades

- 29 Apuntadors i memòria dinàmica
- 30 Tipus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris
- 35 Arbres N-aris
- 36 Arbres generals
- 37 Implementació de mètodes: accedint la representació
- Estructures de dades noves
  - Cues ordenades
  - Multillistes

#### Cues ordenades

- Modificació de la classe Cua: propietat addicional de poder ser recorregudes en ordre creixent respecte al valor dels seus elements
- Dos tipus d'ordre: cronològic (com fins ara) + per valor (nou)
- Cal que hi hagi un operador < definit en el tipus o classe dels elements
- Cal redefinir la implementació amb més apuntadors

#### Apuntadors:

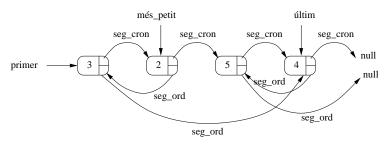
- primer, ultim i seg per gestionar l'ordre d'arribada a la cua (ordre estàndar de cua, cronològic).
- mes\_petit i seg\_ord per gestionar l'ordre creixent segons el valor dels elements.

#### Nova definició de la classe

```
template <class T> class CuaOrd {
 private:
    struct node cuaOrd {
        T info:
        node_cuaOrd* seg_ord;
        node_cuaOrd* seq;
    };
    int longitud;
    node_cuaOrd* primer;
    node cuaOrd* ultim;
    node cuaOrd* mes petit;
    ... // especificació i implementació d'operacions privades
 public:
    ... // especificació i implementació d'operacions públiques
};
```

## Esquema de la implementació

#### Exemple:



### Implementació cues ordenades

- Veurem només dues operacions públiques: demanar\_torn (push) i concatenar
- Exercici: especificació i implementació d'altres operacions que caldria incloure

#### Demanar torn (push) I

### Demanar torn (push) II

```
void demanar_torn(const T& x) {
  node_cuaOrd* n = new node_cuaOrd;
  n \rightarrow info = x;
  n -> seg = nullptr;
  if (primer == nullptr) {
   primer = ultim = n;
   mes_petit = n;
    n -> seg_ord = nullptr;
  } else {
  ++longitud;
```

### Demanar torn (push) III

```
} else {
  // la cua conté altres elements
  // (primer != nullptr => mes_petit != nullptr)
  // 1. el nou node és l'últim en ordre cronològic
  ultim -> seg = n;
  ultim = n;
  // 2. ara inserim el nou node ón pertoca en
  // ordre creixent
  mes_petit = inserta_ord(mes_petit, n);
}
```

### Demanar torn (push) III

```
// Pre: la cadena que começa a p sequint els apuntadors seq_ord
// està en ordre creixent de valor, n != nullptr
// Post: retorna un apuntador al primer de la cadena resultant
// d'inserir el node apuntat per n en ordre creixent a la cadena
// que comença a p
static node_cuaOrd* inserta_ord(node_cuaOrd* p, node cuaOrd* n)
  if (p == nullptr) return n;
  if (n -> info  info) {
    n \rightarrow seq ord = p;
    return n;
  } else {
     p -> seq_ord = inserta_ord(p -> seq_ord, n)
     return p;
```

#### Concatenar I

```
void concatenar(CuaOrd& c2) { 
 /* Pre: la cuaOrd implícita és C_1, c2 = C_2 */ 
 /* Post: la cuaOrd implícita representa la concatenació de C_1 i C_2 en el ordre cronològic (és a dir, tot element de C_2 vé després de qualsevol element de C_1 en ordre cronològic); la cuaOrd implícita també representa la fusió de C_1 i C_2 en el ordre creixent; finalment c2 queda buida */
```

#### Concatenar II

```
void concatenar(CuaOrd &c2) {
  if (c2.primer == nullptr) return;
  // només caldrà fer alguna cosa si c2 no és buida
  if (primer == nullptr) {
    // si el la cuaOrd implícita és buida, llavors
    // li transferim els continguts de c2
     primer = c2.primer;
      ultim = c2.ultim:
      mes_petit = c2.mes_petit;
  } else { ... }
  // la cuaOrd implícita augmenta la seva
  // longitud en tants elements com tenia c2
  longitud += c2.longitud;
  // i buidem la cuaOrd c2
  c2.primer = c2.ultim = c2.mes petit = nullptr;
  c2.longitud = 0;
```

#### Concatenar III

```
{ // ni la cuaOrd ni c2 són buides
  // connectem la cuaOrd i c2
  // pero orde cronològic
  ultim -> seg = c2.primer; // amb el primer de c2
  ultim = c2.ultim_node; // i actualitzem l'últim

  // ara fem la fusió dels nodes de les dues cues segon
  // l'ordre creixent;
  mes_petit = fusiona(mes_petit, c2.mes_petit);
}
```

#### Concatenar IV

```
static node_cuaOrd* fusiona(node_cuaOrd* n1, node_cuaOrd* n2) {
   if (n1 == nullptr) return n2;
   if (n2 == nullptr) return n1;
   // n1 != nullptr and n2 != nullptr
   if (n1 -> info <= n2 -> info) {
      n1 -> seg_ord = fusiona(n1 -> seg_ord, n2);
      return n1;
   } else {
      n2 -> seg_ord = fusiona(n1, n2 -> seg_ord);
      return n2;
   }
}
```

### Part VII

## Tipus Recursius de Dades

- Apuntadors i memòria dinàmica
- 30 Tipus recursius de dades: Generalitats
- 31 Piles i cues
- 32 Implementació de llistes
- 33 Llistes doblement encadenades amb sentinella
- 34 Arbres binaris
- 35 Arbres N-aris
- 36 Arbres generals
- Implementació de mètodes: accedint la representació
- 38 Estructures de dades noves
  - Cues ordenades
    - Multillistes

#### Multillistes: Motivació

Volem guardar una taula molt gran però molt *esparsa*: molts elements nuls

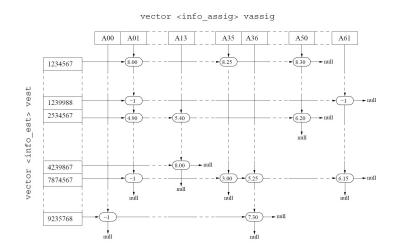
#### Necessitem:

- Donat un índex de fila, recuperar tots els elements no nuls de la fila
- Donat un índex de columna, recuperar tots els elements no nuls de la columna

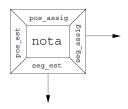
Exemple: Taula per guardar cursos de la FIB:

"l'estudiant X estava matriculat a Y i ha tret nota Z"

### Multillistes: Esquema



#### Multillistes: Node



Implementació i detalls: → apunts