Tipus Abstractes de Dades Lineals

Lluís Alsedà

Departament de Matemàtiques
Universitat Autònoma de Barcelona

http://www.mat.uab.cat/∼alseda

abril 2015

(vers. 4.0.0)

UAB Universitat Autònoma de Barcelona

DEPARTAMENT DE **MATEMÀTIQUES**

Continguts

Tipus abstractes de dades — Introducció 1
Exemple: Nombres complexos i el Conjunt de Mandelbrot 14

Llistes doblement enllaçades 22
Funcions de gestió 28
Funcions de moviment 32
Funcions d'informació 40
Cerca seqüencial en llistes enllaçades dobles 46
Funcions d'actualització 50
Exemple senzill d'aplicació: Successions de Farey 69
Cerca binària en llistes enllaçades dobles 75
Ordenació de llistes enllaçades dobles 82

Cues 135
Implementació d'una cua senzilla de mida fixada 138
Implementació d'una cua de mida arbitrària amb llistes enllaçades 141

Stacks 147
Implementació d'un stack senzill de mida fixada 149
Implementació d'un stack de mida arbitrària amb llistes enllaçades 151
Exemple: l'Span del preu d'una acció de borsa 155

Índex

1. Introducció
2. Exemple: el tipus de dades `int` del **C**
3. Un exemple general
4. Especificació dels TAD
5. Implementació dels TAD
6. Ús dels TAD
7. Un exemple (incomplet): Nombres complexos
8. Tipus de TAD

Introducció

Amb l'aparició dels llenguatges de programació estructurats a la dècada dels 60 sorgeix el concepte de *tipus de dades* definit com un conjunt de valors que serveix de domini de certes operacions.

En aquests llenguatges (**C**, Pascal i similars), els tipus de dades serveixen sobretot per classificar els objectes dels programes (variables, paràmetres i constants) i determinar quins valors poden prendre i quines operacions hi són aplicables.

Aquesta noció, però, es va revelar insuficient atès que l'ús de les dades dins dels programes no coneixia altra restricció que les imposades pel compilador, la qual cosa era molt inconvenient en els nous tipus de dades definits per l'usuari, sobretot perquè no es restringia de cap manera el seu àmbit de manipulació.

Per solucionar aquesta mancança es va introduir el concepte de *Tipus Abstracte de Dades (TAD)*, que considera un tipus de dades no només com el conjunt de valors que el caracteritza sinó també com les operacions que s'hi poden aplicar, juntament amb les propietats que determinen inequívocament el seu comportament.

En realitat, el concepte de TAD ja existeix en els llenguatges de programació estructurats sota la forma dels tipus predefinits, que es poden considerar com tipus abstractes amb un lleuger canvi d'enfocament.

Exemple: el tipus de dades `int` del C

La definició del TAD corresponent consisteix a determinar:

- **Quins són els seus valors:** tots aquells nombres enters dins l'interval `[minint, maxint]`.
- **Quines són les seves operacions:** la suma, la resta, el producte, el quocient i el residu de la divisió.
- **Quines són les propietats que compleixen aquestes operacions:** n'hi ha moltes; per enumerar-ne algunes: $a + b = b + a$, $a * 0 = 0$, etc.

Exemple: el tipus de dades `int` del C: Resumint

Es pot definir un tipus abstracte de dades com un conjunt de valors sobre els quals s'aplica un conjunt donat d'operacions que compleixen determinades propietats.

abstracte prové d'_abstracció_

Els valors d'un tipus poden ser manipulats mitjançant les seves operacions si se saben les propietats que aquestes compleixen, sense que sigui necessari cap més coneixement sobre el tipus; en concret, la seva implementació a la màquina és absolutament irrellevant.

Dit en altres paraules, la manipulació dels objectes d'un tipus només depèn del comportament descrit a la seva especificació i és independent de la seva implementació

Exemple: el tipus de dades `int` del C: Resumint

Qualsevol programa escrit en aquest llenguatge pot efectuar l'operació $x + y$ (essent x i y dues variables enteres) amb la certesa que sempre calcularà la suma dels objectes x i y, independentment de la representació interna dels enters a la màquina que està executant el programa (complement a 2, signe i magnitud, etc.) perquè, sigui quina sigui, la seva definició, el **C** assegura que la suma es comporta d'una manera determinada.

Aquesta característica s'anomena *independència de la representació* i dota els programes de robustesa.

Un exemple general

La importància dels TAD apareix en considerar la seva aplicació per a crear nous tipus de dades. Imaginem que en una aplicació de l'àmbit de l'enginyeria, cal treballar amb el tipus dels números complexos. Si es defineix un TAD per als números complexos, cal determinar el següent:

- El domini de valors del tipus: tots els números complexos de parts real i imaginària de tipus real i compreses en l'interval `[minreal, maxreal]`.
- Les operacions que es poden aplicar sobre el valor del tipus. El nostre coneixement dels números complexos identifica la suma, la resta, etc., si bé el context de l'aplicació pot fer que se n'hi introdueixi alguna de nova.
- Les propietats que compleixen aquestes operacions. Les pròpies d'aquests números.

Un exemple general

L'aplicació en qüestió podrà utilitzar aquest nou TAD sense conèixer cap detall de la seva representació. És a dir, l'expressió `suma(a, b)`, essent `a` i `b` dues dades d'aquest TAD i essent `suma` l'operació de sumar dos complexos, es comportarà de la manera esperada independentment de si els números complexos es representen en notació binomial, polar o com sigui. És a dir, la manipulació dels complexos depèn de la seva especificació i és independent de la seva implementació.

Especificació dels TAD: estableix les propietats que el defineixen

L'especificació d'un TAD li dóna nom, introdueix les seves operacions i en fixa el comportament. La llista d'operacions del TAD s'anomena *signatura o interfície del TAD*. Per a cada operació, se'n diu el nom, el tipus dels seus paràmetres i el tipus del valor que retorna; de tot això també se'n diu *signatura o interfície*, però ara d'una operació. El comportament es pot determinar de diferents maneres: mitjançant comentaris informals, barrejant explicacions informals i formals, emprant notacions gràfiques o mitjançant especificacions totalment formals.

Especificació dels TAD: estableix les propietats que el defineixen

Per tal que sigui útil, una especificació ha de ser:

precisa: només ha de dir allò realment imprescindible,

general: adaptable a diferents contextos,

llegible: que serveixi com a instrument de comunicació entre l'especificador i els usuaris del tipus, d'una banda, i entre l'especificador i l'implementador, de l'altra i

no ambigua: que eviti posteriors problemes d'interpretació.

L'especificació del tipus, que és única, defineix totalment el seu comportament a qualsevol usuari que el necessiti. Segons el seu grau de formalisme, serà més o menys fàcil d'escriure i de llegir i més o menys propensa a ambigüitats i incompleteses.

Implementació dels TAD: determina una representació per als valors del tipus i codifica les seves operacions a partir d'aquesta representació

La implementació d'un TAD determina una representació dels valors d'aquest i la codificació de les operacions segons aquesta representació. La representació consistirà a escollir el següent:

1. una *estructura de dades* adequada per a l'especificació donada mitjançant vectors, tuples, estructures i similars;
2. un predicat de correctesa anomenat *invariant de la representació* (aquest predicat estableix quines configuracions de l'estructura de dades són vàlides i quines són prohibides).

És molt important triar una estructura de dades adequada atesos els requisits d'utilització del TAD, principalment pel que fa a l'eficiència en temps i espai.

Implementació dels TAD: determina una representació per als valors del tipus i codifica les seves operacions a partir d'aquesta representació

Es fa usant un llenguatge de programació convencional.

Per tal que sigui útil, una especificació ha de ser:

estructurada: per facilitar-ne el desenvolupament,

eficient: per optimitzar l'ús de recursos del computador,

llegible: per facilitar-ne la modificació i el manteniment.

Ús dels TAD

Una vegada definit el TAD, es pot fer servir de la mateixa manera que s'utilitza un tipus de dades predefinit, més enllà de certes diferències notacionals que hi pugui haver. És a dir, se'n poden declarar variables, paràmetres, utilitzar-los en altres tipus, etc.

La programació amb TAD comporta certes propietats beneficioses:

Abstracció: Els programes poden fer servir el TAD coneixent només la seva semàntica i desconeixent els detalls de la seva implementació

Correcció: Els TAD es poden utilitzar com a unitats en el procés de prova de programes, cosa que facilita la detecció i correcció d'errors

Eficiència: Els TAD es poden implementar de manera diferent en cada context d'ús, cosa que n'afavoreix un comportament òptim.

Un exemple (incomplet): Nombres complexos

Declaració

```
typedef struct { double re, im; } complex;
typedef struct { double r, theta; } polarcomplex;
```

Funcions de manipulació i conversió

```
complex init(double x, double y) { complex t;
   t.re = x; t.im = y;
   return t;
}
void show(complex a) {
  printf("%f + %f i\n", a.re, a.im);
}
double real(complex a) { return a.re; }
double imag(complex a) { return a.im; }
double modulus(complex a) {
  return sqrt(a.re * a.re + a.im * a.im);
}
complex chngsign(complex a) {
   a.re = -a.re; a.im = -a.im;
   return a;
}
complex conj(complex a) {a.im = -a.im; return a;}
```

Nota

`chngsign`, `conj` i `reciprocal` funcionen sense modificar el valor d'entrada degut a que en **C** els paràmetres es passen a les funcions *per valor*.

Funcions aritmètiques bàsiques

```
complex add(complex a, complex b) { complex t;
   t.re = a.re + b.re;
   t.im = a.im + b.im;
   return t;
}
complex subtract(complex a, complex b) {
   return add(a, chngsign(b));
}
complex mult(complex a, complex b) { complex t;
   t.re = a.re * b.re - a.im * b.im;
   t.im = a.re * b.im + a.im * b.re;
   return t;
}
complex reciprocal(complex a) {
   double modsq = a.re * a.re + a.im * a.im;
   a.re /= modsq; a.im = -a.im/modsq;
   return a;
}

complex quo(complex a, complex b) {
  return mult(a, reciprocal(b));
}
```

Un exemple (incomplet): Nombres complexos

Conversió a polars i la funció pow

```
complex polarcomplextocomplex(polarcomplex u){ complex t;
   t.re = u.r * cos(u.theta);
   t.im = u.r * sin(u.theta);
   return t;
}
polarcomplex complextopolar(complex a){ polarcomplex u;
   u.r = modulus(a);
   u.theta = atan2(a.im,a.re);
   return u;
}
complex complexpow(complex a, int n){ polarcomplex u;
  switch ( n ) {
    case 0:  return init(1.0, 0.0); break;
    case 1:  return a; break;
    case -1: return reciprocal(a); break;
    case 2:  return mult(a,a); break;
    case -2: return reciprocal(mult(a,a)); break;
    case 3:  return mult(a,mult(a,a)); break;
    case -3: return reciprocal(mult(a,mult(a,a))); break;
    default:
      u = complextopolar(a);
      u.r = pow(u.r, n); u.theta = n*u.theta;
      return polarcomplextocomplex(u);
      break;
  }
}
```

Exemple: El conjunt de Mandelbrot de $z^2 + c$

Denotem $p_c(z) = z^2 + c$ amb $z, c \in \mathbb{C}$.

Estem interessats en iterar $p_c(z)$. Així denotem

$$p_c^{n+1}(z) := p_c(p_c^n(z)) \quad \text{per} \quad n \geq 0.$$

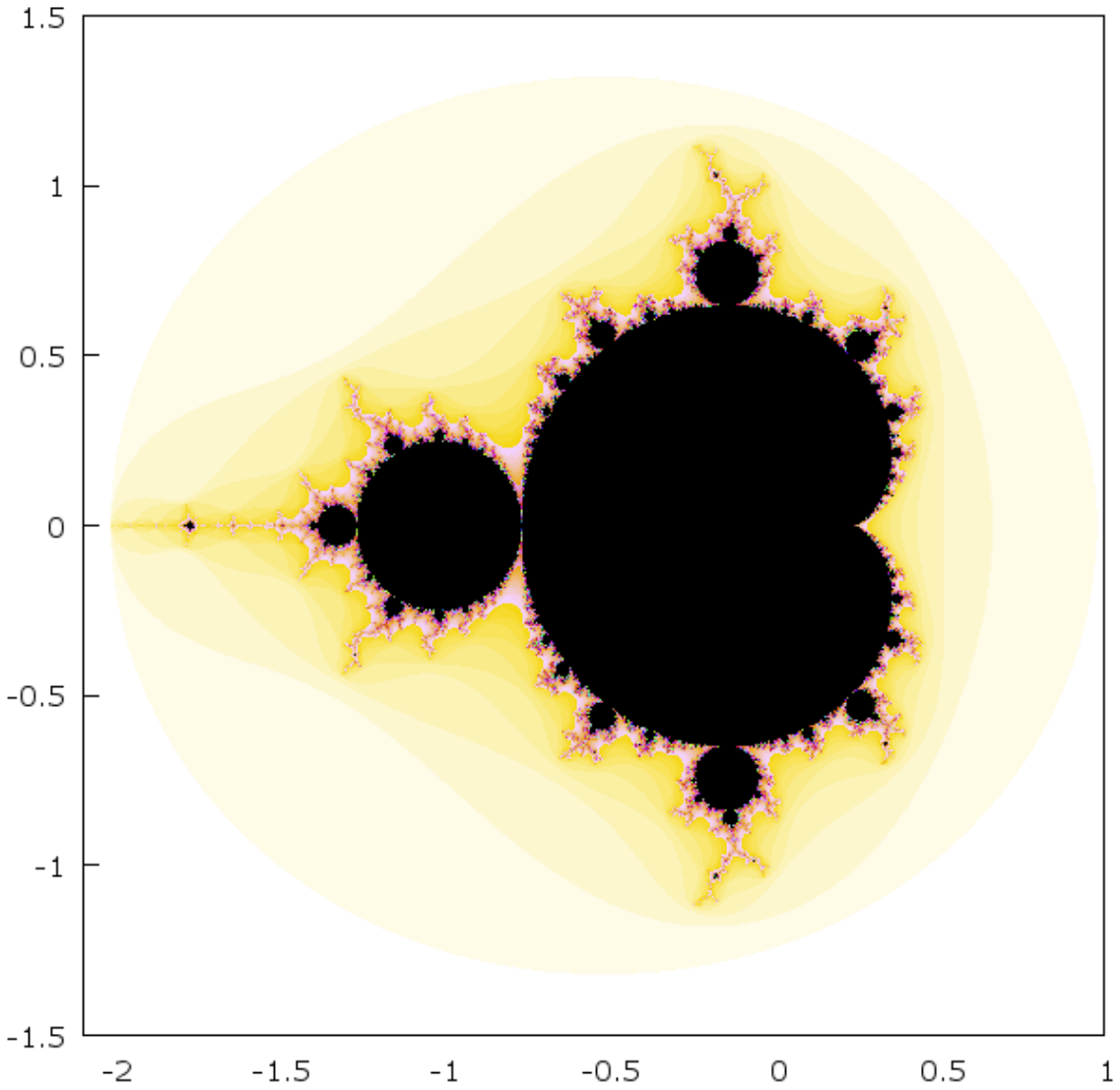
Per completesa, $P_c^0(z) = z$ i, així, $P_c^1(z) = P_c(z)$.

El conjunt de Mandelbrot es defineix com el conjunt dels $c \in \mathbb{C}$ tals que la successió de mòduls dels iterats de 0 és afitada (zona negra a la figura de la pàgina següent). És a dir, existeix M tal que $|P_c^n(0)| \leq M$ per a tot $n \in \mathbb{N}$.

Es pot veure que la successió de mòduls dels iterats de 0 és no afitada si i només si el mòdul de $p_c^n(0)$ és més gran que 2 per algun n. És a dir, el conjunt de Mandelbrot es pot definir com:

$$\{c \in \mathbb{C} \colon |P_c^n(0)| \leq 2 \text{ per a tot } n \geq 0\}.$$

El conjunt de Mandelbrot a partir del fitxer de resultats



Càlcul del conjunt de Mandelbrot

Observem que $P_c(0) = c$ i que $|0| = 0 < 2$. Per tant,

$$\{c \in \mathbb{C}: |P_c^n(c)| \leq 2 \text{ per a tot } n \geq 0\} = \{c \in \mathbb{C}: |c| \leq 2 \text{ i } |P_c^n(c)| \leq 2 \text{ per a tot } n \geq 1\}.$$

Donat que no podem fer una quantitat infinita d'iterats definim el *conjunt de Mandelbrot aproximat* com el conjunt

$$\{c \in \mathbb{C}: |c| \leq 2 \text{ i } |P_c^n(c)| \leq 2 \text{ per } n = 1, 2, \ldots, \texttt{MAXIT} - 1\}$$

per un valor `MAXIT` prou gran.

El programa de la plana següent fa aquest calcul: per a cada punt $c = x + iy$ tal que $|x + iy| \leq 2$ escriu x, y i el nombre $n <$`MAXIT` tal que $|P_c^n(c)| > 2$ o $n =$`MAXIT` si això no ha passat (aquests darrers punts són els que formen part del conjunt de Mandelbrot aproximat).

Càlcul del conjunt de Mandelbrot (continuació)

Notem a més que $|z| = |\overline{z}|$ i $\overline{z} \cdot \overline{z} = \overline{z \cdot z}$. Per tant, $P_c^n(c) = \overline{P_{\overline{c}}^n(\overline{c})}$ i $|P_c^n(c)| = |P_{\overline{c}}^n(\overline{c})|$. És a dir, el conjunt de Mandelbrot és simètric respecte de l'eix real i solament cal calcular-lo pels punts de la forma $x + iy$ amb $y \geq 0$ i $|x + iy| \leq 2$. El programa implementa aquesta simplificació i en escriure x, y i n també escriu x, $-y$ i n. Aquesta darrera n és correcta per la simetria que acabem d'explicar.

La figura s'ha obtingut a partir del fitxer de resultats dibuixant en negre els punts del conjunt de Mandelbrot ($n =$`MAXIT`), en blanc els punts amb $n = 2$ i en gradient de color de negre a blanc els altres valors de n: més fosc més iterats es necessiten per a escapar i, per tant, mes "aprop" del conjunt de Mandelbrot. El dibuix s'ha obtingut a partir del fitxer de resultats amb el programa `gnuplot`.

Exemple: El conjunt de Mandelbrot de $z^2 + c$

```
#include <stdio.h>
#include <math.h>
#define pas 0.001

typedef struct { double re, im; } complex;

complex init(double x, double y) { complex t; t.re = x; t.im = y; return t; }
double modulus(complex a) { return sqrt(a.re * a.re + a.im * a.im); }
complex add(complex a, complex b) { complex t; t.re=a.re + b.re; t.im=a.im + b.im; return t; }
complex mult(complex a, complex b) { complex t;
   t.re = a.re * b.re - a.im * b.im;
   t.im = a.re * b.im + a.im * b.re;
   return t;
}

#define MAXIT 255
unsigned char mand(complex c) { register unsigned char i=1; complex z = c;
    do { z = add(mult(z,z), c); i++; } while ( modulus(z) <= 2.0 && i < MAXIT );
    return i;
}

void main() {
  double x, y;

  for (x=-2.0; x <= 2.0; x += pas){
    for (y=0.0; y <= 2.0; y += pas){
        unsigned char niter = mand(init(x,y));
        printf("%lf %lf %u\n", x, y, niter);
        printf("%lf %lf %u\n", x, -y, niter);
    }
  }
}
```

El fitxer de resultats

```
   ..............
-1.170000 0.197000 94
-1.170000 -0.197000 94
-1.170000 0.198000 106
-1.170000 -0.198000 106
-1.170000 0.199000 35
-1.170000 -0.199000 35
-1.170000 0.200000 29
   ..............
```

Exemples de TAD de manipulació de dades

- *Stacks (piles)*,
- *Cues*,
- *Llistes enllaçades*,
- *Arbres*,
- *conjunts*,
- *funcions* (mantenen relacions entre elements de dos dominis: la *clau* i l'*abast*),
- *Arrays associatius*,
- *Vectors esparsos* i *matrius esparses*
- etc

Índex: Llistes doblement enllaçades

1. Introducció
2. Estructura i elements de les llistes
3. Operacions base de les llistes
4. Funcions de gestió: Declaració; Inicialització (`InicialitzaLlista`); `BorraLlista`; `OrdenaIndex`; `MergeSort`
5. Funcions de moviment: `InicideLlista`; `FinaldeLlista`; `SaltaN`; `DeSaltaA`; `Nth`; `seguent`; `anterior`; recorregut seqüencial endavant estàndard
6. Funcions d'informació: `EsBuida`; `Llargada`; `ImprimeixNode`; `ImprimeixLlista`; `pertany`;
7. Cerca en llistes enllaçades dobles: `CercaSequencial` (diverses); `CercaBinaria`

continua...

Índex: Llistes doblement enllaçades (cont.)

8. Funcions d'actualització: `InsereixNodeInicial` (`push`); `InsereixNodeFinal`; `InsereixEntreNodes`; `BorraNode`
9. Funcions d'actualització derivades: `InsereixNodeabans`; `InsereixNodedespres`; `BorraNodeInicial` (`pop`); `BorraNodeFinal`
10. Exemple: Inserció d'un node a un lloc determinat per mantenir una ordenació (cerca numèrica)
11. Exemple senzill d'aplicació (de llistes simples): Successions de Farey
12. Ordenació de llistes enllaçades dobles:
    - Creant un índex d'ordenació (vector d'apuntadors a estructures): `CreaIndex` i reordenant la llista usant un vector d'apuntadors a estructures intermedi: `OrdenaUsantIndex`
    - *Merge Sort recursiu*: la funció `MergeSortRecursiu` i *Merge Sort no recursiu*: la funció `MergeSort`

Introducció

Una llista enllaçada és una col·lecció d'elements disposats seqüencialment que permet la inserció i eliminació d'elements en qualsevol lloc de la seqüència.

En una llista enllaçada podem inserir i eliminar nodes sense haver de conèixer la mida de la llista i de les dades. L'eina que permet el funcionament d'aquest mecanisme és la de reservar i alliberar els espais de memòria dels nodes mitjançant l'assignació dinàmica.

En l'ús de les llistes dinàmiques cal sempre tenir present fer la reserva de memòria com a pas previ a la creació d'un node i d'alliberar-la en el moment en què un node desapareix.

Les llistes enllaçades poden ser simples i dobles. En una llista simple, de cada node solament podem saltar al següent. Per tant, solament podem recórrer la llista endavant. En una llista doblement enllaçada, de cada node podem saltar al posterior i a l'anterior. Per tant, podem recórrer la llista tan endavant com endarrere.

En aquestes notes estudiarem el cas més complet de les llistes doblement enllaçades, ja que el cas simple n'és una particularització.

No contemplarem el cas de llistes circulars.

Estructura i elements de les llistes

- Una llista no buida està composada per una successió ordenada de nodes amb un *primer node* i un *darrer node*.

  Usualment els nodes s'implementen com estructures (`struct`).

  Cada una d'aquestes estructures, apart de contenir les dades pròpies del node, conté un o mes apuntadors a estructures del mateix tipus que serveixen per a relacionar els elements de la llista.

- Tota llista comença a un apuntador al primer element de la llista. Aquest apuntador és l'element bàsic d'accés a la llista (*clau de la llista*). La seva preservació és fonamental per accedir a la llista.

  Per facilitat d'ús també es pot declarar i mantenir (actualitzar) un apuntador al darrer element de la llista per accedir-hi en ordre seqüencial invers.

- Tot node de la llista té un apuntador `seg` que apunta al node següent a la llista. En el cas del darrer node, que no té següent, `seg = NULL`. Per tant, el darrer node és l'únic amb aquesta propietat.

- Tot node de la llista té un apuntador `prev` que apunta al node anterior a la llista. En el cas del primer node, que no té anterior, `prev = NULL`. Per tant, el primer node és l'únic amb aquesta propietat.

Operacions base de les llistes

Les operacions base necessàries per a gestionar usar i mantenir una llista enllaçada, que formen part de l'especificació completa del TAD *llista enllaçada doble*, són (entre altres):

Funcions de gestió

- Declaració
- Inicialització (*InicialitzaLlista*)
- *BorraLlista*
- *OrdenaIndex*
- *MergeSort*

Funcions de moviment

- *InicideLlista*
- *SaltaN*
- *FinaldeLlista*
- *DeSaltaA*
- *Nth*

Funcions d'informació

- *EsBuida*
- *Llargada*
- *ImprimeixNode*
- *ImprimeixLlista*
- *CercaSequencial*
- *CercaBinaria*

Funcions base d'actualització

- *InsereixNodeInicial* (*push*)
- *InsereixNodeFinal*
- *InsereixEntreNodes*
- *BorraNode*

Funcions d'actualització derivades

- *InsereixNodeabans*
- *InsereixNodedespres*
- *BorraNodeInicial* (*pop*)
- *BorraNodeFinal*

Operacions base de les llistes 1/2

Operacions base de les llistes (cont.)

Per a desenvolupar i il·lustrar els procediments i algorismes citats usarem un exemple concret: La gestió d'una llista d'alumnes amb les seves notes.

Per a fixar idees

usem una base de dades (fitxer) d'alumnes com, per exemple:

dadesalumnes.dat

```
Pere;Barniol Serra;3.5;6.6
Joan;Lopez Garcia;8.4;5.5
Ramon;Marti Perez;6.3;7.4
Maria;Barniol Roca;9.5;7.4
```

Operacions base de les llistes 2/2

Funcions de gestió (excepte ordenació)

Declaració dels elements de la llista

```
typedef struct Node_Alumne {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
  struct Node_Alumne *prev, *seg;
} alumne;
```

Declaració del contenidor de la llista i la funció d'inicialització `InicialitzaLlista`

```
typedef struct {
  alumne *start;
  alumne *end;
  unsigned nalumnes;
} LlistaAlumnes;
```

Es podrien afegir altres paràmetres de la llista, que també ens interessi mantenir.

```
void InicialitzaLlista (LlistaAlumnes * llista) {
  llista->start = llista->end = NULL;
  llista->nalumnes = 0U;
}
```

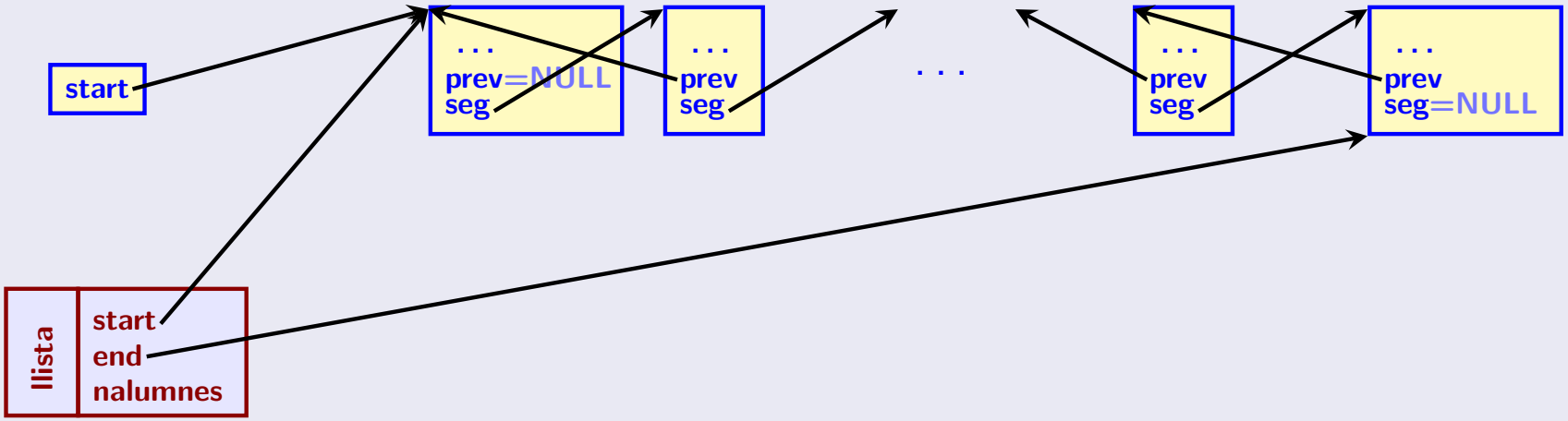
El contenidor `LlistaAlumnes` i la funció d'inicialització associada no són imprescindibles. Es suficient usar un apuntador `start` a l'inici de la llista.

El manteniment d'`end` i `nalumnes` agilitza algunes operacions. En canvi complica una mica la programació.

Funcions de gestió (excepte ordenació) 1/4

Funcions de gestió (excepte ordenació) (cont.)

Exemple gràfic de llista enllaçada doble i la seva organització



Declaració i inicialització completa de la llista usant una funció d'inicialització

```
typedef struct Node_Alumne {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
  struct Node_Alumne *prev, *seg;
} alumne;
int main () {
  LlistaAlumnes segon_grupA;
  InicialitzaLlista (&segon_grupA);
```

```
typedef struct {
    alumne *start;
    alumne *end;
    unsigned nalumnes;
} LlistaAlumnes;
```

Funcions de gestió (excepte ordenació) (cont.)

Declaració i inicialització de la llista usant una funció d'inicialització

```
LlistaAlumnes segon_grupA;
InicialitzaLlista(&segon_grupA);
```

Observació

Notem que a les funcions que necessiten modificar els paràmetres de la llista (`start`, `end` i `nalumnes`), com són `InicialitzaLlista`, `BorraLLista`, `BorraNode` i altres, cal que els hi passem com a paràmetre un apuntador a la llista `LlistaAlumnes *`. El fet de copiar el contingut de l'estructura `LlistaAlumnes` per valor com a paràmetre no funciona.

- Veure la funció `ImprimeixLlista` on es fa ús del fet que l'estructura solament es modifica internament si no es passa com a apuntador, per a usar `llista.start` com a índex del bucle.

Una altra possibilitat: declaració i inicialització de la llista (a la declaració)

```
LlistaAlumnes segon_grupA = {NULL, NULL, 0U};
```

Alternativament si no hem definit l'estructura `LlistaAlumnes`:

Declaració i inicialització de la variable d'accés a la llista

```
alumne *segon_grupA_start = NULL;
```

En aquest cas també podem mantenir l'apuntador `end` i la variable `nalumnes` però sense poder mantenir la unitat dels paràmetres de la llista.

Funcions de gestió (excepte ordenació) (cont.)

BorraLlista

```
void BorraLlista (LlistaAlumnes * llista) {
  register alumne *node_curr = llista->start, *aux;
  while (node_curr) {
      aux = node_curr; node_curr = node_curr->seg; free (aux);
  }
  InicialitzaLlista (llista);
}
```

Ús de BorraLlista

```
BorraLlista (&segon_grupA);
```

Recorregut seqüencial endavant estàndard:
`BorraLlista` l'implementa amb in bucle `while`. Funcionament:

- `node_curr = llista->start` inicialitza `node_curr` al començament de la llista en una variable ràpida (`registre`) que s'usa per a agilitzar el recorregut al bucle.
- Saltem successivament al node següent amb
  `node_curr = node_curr->seg`
- Finalment, es para el bucle quan `node_curr` és fals (és a dir quan és `NULL`). Això passa perquè `node_curr` és l'apuntador `seg=NULL` del darrer node i hem saltat després del final de la llista.

Funcions de moviment

InicideLlista i FinaldeLlista

```
alumne * InicideLlista(LlistaAlumnes llista){ return llista.start; }
alumne * FinaldeLlista(LlistaAlumnes llista){ return llista.end; }
```

seguent i anterior

```
alumne * seguent (alumne * alu_ori) {
    return ((alu_ori) ? alu_ori->seg : NULL);
}
alumne * anterior (alumne * alu_ori) {
    return ((alu_ori) ? alu_ori->prev : NULL);
}
```

seguent i anterior

Per eficiència usem un *if aritmètic*.

Ús de InicideLlista, FinaldeLlista, seguent i anterior

```
alumne * alu = InicideLlista (segon_grupA));
ImprimeixNode (FinaldeLlista (segon_grupA));
ImprimeixNode (seguent (InicideLlista (segon_grupA)));   Segon alumne
alu = anterior( FinaldeLlista (segon_grupA)));           Penúltim alumne
```

Funcions de moviment (cont.)

`SaltaN` implementa un salt dins de la llista de mida `N` (on `N` pot ser positiu, negatiu o zero) *relatiu a una posició actual donada* per l'apuntador `alu_ori` i retorna l'apuntador al node destí del salt.

`SaltaN` — Salt relatiu a partir d'una posició actual

```
alumne * SaltaN (const long N, alumne * alu_ori) {
  register long n; register alumne * curr;
  if (N >= 0L) for (n = 0L, curr = alu_ori; n < N && curr;
                    n++, curr = curr->seg);
  else if (N < 0L) for (n = 0L, curr = alu_ori; n > N && curr;
                        n--, curr = curr->prev);
  return curr;
}
```

Observació

Totes les funcions d'indexat (numeració) dels elements de la llista els compten de `0` a `llista.nalumnes-1` com si fos l'index d'un vector. És a dir, el primer node té index `0`, el segon té index `1`, ..., i el darrer té index `llista.nalumnes-1`.

Funcions de moviment (cont.)

Ús de SaltaN

```
alu = SaltaN (-2, FinaldeLlista (segon_grupA));                    Antepenúltim
ImprimeixNode (SaltaN (1, InicideLlista (segon_grupA)));           Segon alumne
alu = SaltaN (0, NULL);                                  ERROR (i a més no salta)
ImprimeixNode (SaltaN (19, InicideLlista (segon_grupA)));          El No. 20
```

Funcionament de SaltaN

Com es pot veure per l'`if (N >= 0L) ....  else if (N < 0L) ...`, separem els cassos `N >= 0` i `N < 0` i, al final, retornem la posició nova calculada: `return curr`.

Explicarem en detall el funcionament del cas `N >= 0`. L'altre és anàleg canviant el signes de l'actualització de la `n`, de les desigualtats i `->seg` per `->prev`.

Hem de recórrer la llista seqüencialment `N` passos endavant i retornar l'adreça de l'element aconseguit.

Funcionament de SaltaN (cont.)

El salt es pot implementar amb un bucle sense cos (és a dir: tota la feina es fa a la part d'inicialització/control/actualització)

```
for (n = 0L, curr = alu_ori; n < N && curr; n++, curr=curr->seg);
```

Aquest bucle té un control d'iteració d'`N` repeticions: `n=0L; n < N; n++`.
Per altra banda, a cada repetició, apart d'actualitzar `n` es salta una posició endavant a partir de la posició donada `curr`:

`n++, curr=curr->seg`.

El fet d'assignar l'adreça del node inicial a una nova variable `register`

`curr = alu_ori`

és per accelerar les iteracions del bucle.

Observem que si `N = 0` el bucle no fa cap iteració i retornem l'adreça `curr` inicial sense modificar.

Funcionament de `SaltaN` (cont.)

Com que el control del bucle és `n < N && curr` aquest s'acabarà retornant `return curr` quan:

- `n = N && curr != NULL`: en aquest cas ja hem fet `N` salts i `curr` (com que no és `NULL`) apunta a l'element de la llista que correspon a saltar `N` endavant a partir de la posició `curr` inicial. Retornem correctament aquest valor.
- `n <= N && curr == NULL`: Com que `curr == NULL` hem sortit de la llista i, per tant, no existeix el node que correspon a saltar `N` posicions endavant a partir de la posició `curr` inicial (la llista s'acaba abans). En aquest cas retornem `curr`, que és `NULL` i s'interpreta com un codi d'error.

Funcions de moviment (cont.)

La funció `DeSaltaA` implementa un salt absolut dins de la llista, des de la posició definida per `ori` (posició numèrica) i `alu_ori` (apuntador a l'element de la posició `ori` de la llista) a la posició `fi` (posició numèrica).

La funció torna un apuntador a l'element de la llista de la posició `fi`.

No controla que `ori` i `fi` estiguin dintre de rang (`0...llista.nalumnes-1`) ni controla la consistència entre `alu_ori` i `ori` (és a dir, que `alu_ori` apunti a l'element `ori` de la llista). És responsabilitat del mòdul que la crida assegurar aquestes restriccions.

`DeSaltaA` — Salt absolut d'una posició especificada a una altra

```
alumne * DeSaltaA (unsigned ori, alumne * alu_ori, unsigned fi) {
  register unsigned n; register alumne *curr;
  if (fi < ori)
    for (n=ori, curr=alu_ori; n > fi; n--, curr=curr->prev);
  else
    for (n=ori, curr=alu_ori; n < fi; n++, curr=curr->seg);
  return curr;
}
```

Funcions de moviment (cont.)

Funcionament de `DeSaltaA`

El codi de la funció `DeSaltaA` és similar al de la funció `SaltaN` amb dues diferències òbvies:

- L'indexat absolut del bucle (d'`ori` a `fi`). Això afecta al control de la variable del bucle.
- Com que suposem que `ori` i `fi` són dintre de rang, no cal comprovar que no ens sortim de la llista (és a dir que `curr != NULL`). Per tant, el control del bucle esdevé
  `n > fi` o `n < fi` en comptes de `n > fi && curr` o `n < fi && curr`.

Com a exemple d'ús veure les funcions `Nth` i `CercaBinaria`.

Funcions de moviment (cont.)

Nth — Anar a l'element N-èssim de la llista — desplaçament absolut

```
alumne * Nth (LlistaAlumnes llista, register unsigned N) {
  if (N > llista.nalumnes / 2) {
          if (N >= llista.nalumnes) return NULL;
          return DeSaltaA (llista.nalumnes-1, llista.end, N);
  }
  return DeSaltaA (OU, llista.start, N);
}
```

Ús d'Nth

```
alu = Nth (segon_grupA, Llargada(segon_grupA)-3);
ImprimeixNode (Nth (segon_grupA, 1U));
ImprimeixNode (Nth (segon_grupA, 19U));
```

Antepenúltim

Segon alumne

El No. 20

Funcionament de Nth

La funció `Nth` és clara a partir de la funció `DeSaltA`.
El fet que separem el cas `N > llista.nalumnes / 2` és per eficiència: si `N` és gran és millor arribar-hi des d'el final.

Funcions d'informació

EsBuida

```
#include <stdbool.h>
bool EsBuida (LlistaAlumnes llista) {return (llista.start == NULL);}
```

Llargada

```
unsigned Llargada (LlistaAlumnes llista) {return (llista.nalumnes);}
```

Ús de EsBuida i Llargada (veure també la funció BorraNode)

```
if (EsBuida (segon_grupA)) fprintf (stderr, "\nERRORLLista buida!);
else printf ("Nombre de nodes: %d\n", Llargada (segon_grupA));

unsigned nalu = Llargada (segon_grupA);
if(nalu > 0) printf ("La llista té %d nodes\n", nalu);

BorraLlista (&segon_grupA);
if (EsBuida (segon_grupA)) printf ("Llista buidada amb éxit\n");
else fprintf (stderr, "\nERROR: No s'ha pogut buidar la llista");
```

Funcions d'informació (cont.)

`Llargada` calculada (per si no es manté la variable `nalumnes` a `LlistaAlumnes`)

```
unsigned Llargada (LlistaAlumnes llista) {
  register alumne *alu;  register unsigned n = 0U;
  for (alu = llista.start; alu; alu = alu->seg) n++;
  return n;
}
```

Recorregut seqüencial endavant estàndard:
La funció `Llargada` calculada l'implementa amb in bucle `for`. Funcionament:

- `alu = llista.start` inicialitza `alu` al començament de la llista en una variable ràpida (`registre`) que s'usa per a agilitzar el recorregut al bucle.
- Saltem successivament al node següent amb `alu = alu->seg`.
- Finalment, es para el bucle quan `alu` és fals (és a dir quan és `NULL`). Això passa perquè `alu` és l'apuntador `prev=NULL` del darrer node i hem saltat després del final de la llista.
- El bucle s'usa per a incrementar el comptador `n++` i, al final, es retorna el valor del comptador. Observem que el comptador s'ha incrementat en una unitat al visitar cada element de la llista. Per tant, el seu valor final és la llargada de la llista.

Funcions d'informació (cont.)

ImprimeixNode

```
void ImprimeixNode (alumne *n) {
  if (n == NULL) return;
  printf("%s, %s: Teor=%f; Pract=%f; Final=%f\n",
         n->cognoms, n->nom,
         n->nteoria, n->npract, n->nfinal);
}
```

Nota

Fins aquí no ens ha calgut saber la informació especifica que conté un node de la llista. Solament hem necessitat saber que cada node conté els apuntadors `prev` i `seg` i els seus convenis explicats abans.

ImprimeixLlista

```
void ImprimeixLlista (LlistaAlumnes llista) {
   while (llista.start){ ImprimeixNode(llista.start);
         llista.start = llista.start->seg;
   }
}
```

Implementació d'ImprimeixLlista

Realitza un recorregut seqüencial endavant estàndard amb un bucle `while` com la funció `BorraLlista` usant `llista.start` com a variable del bucle.
L'acció que es fa a cada iteració del bucle és imprimir el node actual apuntat per `llista.start`.
Donat que el procés d'impressió és lent d'entrada no cal (és inútil?) accelerar el bucle amb una variable `registre`, com hem fet a altres recorreguts seqüencials.
Notem que això no modifica ni `llista.start` ni el node llista original; solament modifiquem la còpia local de `llista.start` (*pas per valor*).

Funcions d'informació (cont.)

ImprimeixLlistaEndarrere

```
void ImprimeixLlistaEndarrere (LlistaAlumnes llista) {
   while (llista.end){ ImprimeixNode(llista.end);
         llista.end = llista.end->prev;
   }
}
```

Implementació d'ImprimeixLlistaEndarrere

És totalment anàloga a la d'`ImprimeixLlista` canviant `llista.start` per `llista.end` i `llista.start->seg` per `llista.end->prev`.

Ús d'ImprimeixNode, ImprimeixLlista i ImprimeixLlistaEndarrere

```
ImprimeixLlista (segon_grupA);
ImprimeixLlistaEndarrere (segon_grupA);
ImprimeixNode (SaltaN (7, InicideLlista (segon_grupA)));
```

Altres exemples d'ús de la funció `ImprimeixNode` s'han donat abans i se'n poden trobar al codi de les funcions `ImprimeixLlista` i `ImprimeixLlistaEndarrere`.

Exercici

Torres Templado, Maria	Angelats Filella, Adria
Morros Barrero, Xavier	Subirana Lopez, Roberto
Casino Coll, Alex	Galindo Casino, Omar
Molina Fernandez, Ethel	Pecharroman Abadal, Daniel
Orriols Fernandez, Gerard	Soler Soler, Joan
Lopez Garcia, Gerard	Filella Vivancos, Sonia
Olmo Carrasco, Alejandro	Chia Gonzalez, Ethel
Vivar Gonzalez, Cristina	Ferrando Ravella, Alex
Ferrando Gomez, Adrian	Coll Alcalde, Joaquim
Loza Garcia, Joan	Mendez Prosopio, Albert
Barrero Filella, Juanita	Fernandez Argon, Juanita
Sabater Fernandez, Ethel	Garcia Ferrando, Emili
Beltran Guimera, David	Navarro Ferrando, Marcel
Artero Molina, Victoria	Mendez Vinyas, Anna
Boix Fernandez, Belinda	Serra Carrasco, Francisco
Gil Quintana, Emili	Ruzafa Barba, Roberto
Vasquez Abadal, David	Porcel Lorenzo, Noelia
Martinez Almendral, Joaquim	Ventura Abadal, Sergi
Fernandez Buendia, David	Coll Frigola, Edgar
Sabater Angelats, Juanita	Perello Duran, Patricia
Chassignet Buxo, Victor	Ventura Batlle, Daniel
Templado Prada, Ethel	Vidal Subirana, Sonia
Vidal Dauner, Marcel	Loza Serra, Sonia
Perello Navarro, Montserrat	Subirana Boix, Emili
Buendia Prosopio, Montserrat	Almendral Porcel, Ana
Serra Martinez, Marc	Solanas Galindo, Joan
Soler Batlle, Maria	Chia Batlle, Victor
Rovira Fernandez, Emili	Compte Sanchez, Alex
Alcalde Ravella, Edgar	Cueto Batlle, Ethel
Ruzafa Vinyas, Victor	Esteve Chia, Roberto
Filella Barba, Alexander	Frigola Beltran, Noelia
Pecharroman Argon, Francisco	Sabater Ravella, Victor
Ravella Arcusa, Adria	Gallego Cueto, Laura
Navarro Mendez, Gerard	Fernandez Gonzalez, Victor
Pignatelli Ravella, Belinda	Alsina Gonzalez, Roger
Garcia Gomez, Nuria	Solanas Lopez, Joaquim
Vasquez Arcusa, Alejandro	Guimera Aliau, Noelia
Gomez Sanchez, Cristina	Martinez Amer, Ethel
Busquet Serra, Victoria	Beltran Rovira, Edgar
Frigola Fernandez, Victoria	Perez Dauner, Marcel
Fernandez Cueto, Eric	Porcel Artero, Anna
Figueras Filella, Omar	Plantada Julia, Omar
Navarro Amer, Alejandro	Lorenzo Alcalde, Mireia
Argon Fernandez, Daniel	Beltran Ravella, Adria
Garcia Templado, David	Vidal Gonzalez, Anna
Buendia Almendral, David	Gonzalez Galindo, Ethel
Rodriguez Dauner, Omar	Garcia Carrasco, Adria
Vidal Esteve, Joaquim	Ruzafa Compte, Joaquim
Perello Carrasco, Daniel	Gonzalez Fernandez, Sergi
Mayoral Lopez, David	

Fer un procediment que imprimeixi la llista en dues columnes.
La primera columna ocupa de la posició 1 a la 38 de cada línia i conté (per ordre i de dalt a baix) els elements $1, 2, \ldots, \lfloor (n+1)/2 \rfloor$ de la llista.
La segona columna ocupa de la posició 40 a la 79 de cada línia i conté (per ordre i de dalt a baix) els elements $\lfloor (n+1)/2 \rfloor + 1, \lfloor (n+1)/2 \rfloor + 2, \ldots, n$ de la llista.

Nota: : $\lfloor (n+1)/2 \rfloor = n/2$ si n és parell i $\lfloor (n+1)/2 \rfloor = (n+1)/2$ si n és senar.

Funcions d'informació (cont.)

pertany: Comprova, per consistència, si un node pertany a la llista especificada

```
bool pertany (alumne * node, LlistaAlumnes llista) {
   register alumne *alu;
   if (node == NULL || EsBuida(llista)) return false;
   for (alu=llista.start; alu; alu=alu->seg)
     if (alu == node) return true;
   return false;
}
```

Ús de pertany

Veure les funcions `InsereixNodeabans`; `InsereixNodedespres` i `BorraNode`.

Implementació de pertany

S'usa per a comprovar la consistència dels paràmetres d'una funció: volem decidir si, en especificar, `(alumne * node, LlistaAlumnes llista)` el node apuntat per `node` pertany a la llista `llista`.

Desgraciadament això implica una laboriosa cerca seqüencial com les que s'han discutit abans (veure les funcions `Llargada calculada` i `SaltaN` per la implementació de la cerca amb un bucle `for`).

El codi implementat comença una cerca seqüencial estàndard endavant a partir de `llista.start` intentant trobar `node`. En aquest cas la informació és consistent i es torna `true`. En cas contrari es torna `false`.

Cerca seqüencial en llistes enllaçades dobles

CercaSequencial_cognom

```
alumne * CercaSequencial_cognom (const char *cognom,
                                 LlistaAlumnes llista) {
   register alumne *alu; unsigned cognomlen = strlen(cognom);
   for (alu=llista.start; alu; alu=alu->seg)
     if (strncmp(cognom, alu->cognoms, cognomlen) == 0)
       return alu;
   return NULL;
}
```

Implementació de CercaSequencial_cognom

Implementa una cerca seqüencial endavant a partir de `llista.start` per a localitzar el primer registre amb el cognom especificat. L'ús de la funció `strncmp` s'ha discutit abastament a les transparències d'**Estructures en C**.

Si el troba torna un apuntador al registre trobat. En cas contrari torna `NULL` com a codi d'error.

`unsigned cognomlen = strlen(cognom);` s'usa per a estalviar aquest càlcul moltes vegades dins del bucle.

Cerca seqüencial en llistes enllaçades dobles (cont.)

CercaSequencial_nota_endarrere_print_tots

```
void CercaSequencial_nota_endarrere_print_tots (
        const float nota,  LlistaAlumnes llista) {
     register alumne *alu;
     for (alu=llista.end; alu; alu=alu->prev)
       if (fabs(alu->nfinal-nota) <= 5.0e-3) ImprimeixNode(alu);
}
```

Implementació de CercaSequencial_nota_endarrere_print_tots

Implementa una cerca seqüencial endarrere a partir de `llista.end` per a localitzar *tots* els registres amb nota igual a la nota especificada (amb dos dígits de precisió). L'acció que es realitza per aquests registres és la d'imprimir-los.

La necessitat d'usar `fabs` (`fabs(...) <= ...`) per a una comparació en `float` s'ha discutit a les transparències d'**Estructures en C**.

Cerca seqüencial en llistes enllaçades dobles (cont.)

CercaSequencial_nota_ge

```
alumne * CercaSequencial_nota_ge (const float nota,
                                  LlistaAlumnes llista) {
   register alumne *alu;
   for (alu=llista.start; alu; alu=alu->seg)
     if (alu->nfinal >= nota) return alu;
   return NULL;
}
```

Implementació de CercaSequencial_nota_ge

Implementa una cerca seqüencial endavant a partir de `llista.start` per a localitzar el primer registre amb nota més gran o igual que la nota especificada.

Si el troba torna un apuntador al registre trobat. En cas contrari (és a dir, quan `nota` és més gran que totes les notes de la llista) torna `NULL` com a codi d'error.

Cerca seqüencial en llistes enllaçades dobles (cont.)

Ús de les funcions `CercaSequencial` (primer alumne amb nfinal `>= 7.2`)

```
alu = CercaSequencial_nota_ge (7.2, segon_grupA);
if (alu) { printf ("Alumne trobat: "); ImprimeixNode (alu); }
else fprintf (stderr, "\nERROR: Alumne no trobat\n");
```

Ús de les funcions `CercaSequencial` (busquem l'alumne `Barniol` – el primer)

```
alu = CercaSequencial_cognom ("Barniol", segon_grupA);
if (alu) { printf ("Alumne trobat: "); ImprimeixNode (alu); }
else printf ("Alumne Barniol no trobat\n");
```

Ús de les funcions `CercaSequencial` (llista inversa dels alumnes que tenen nota final `5.63`)

```
CercaSequencial_nota_endarrere_print_tots (5.63, segon_grupA);
```

Funcions d'actualització: la funció BorraNode

BorraNode

```
void BorraNode (alumne * node_a_borrar, LlistaAlumnes * llista) {
  if (!pertany (node_a_borrar, *llista)) return;
  if (Llargada (*llista) == 1U) {
      free (llista->start); InicialitzaLlista (llista); return;
  }
```

Alternativament es pot usar `BorraLLista`

```
  if (node_a_borrar == llista->start) {
      llista->start = node_a_borrar->seg;
      (llista->start)->prev = NULL;
  } else if (node_a_borrar == llista->end) {
      llista->end = node_a_borrar->prev;
      (llista->end)->seg = NULL;
  } else {
      (node_a_borrar->seg)->prev = node_a_borrar->prev;
      (node_a_borrar->prev)->seg = node_a_borrar->seg;
  }
  free (node_a_borrar);
  llista->nalumnes--;
}
```

Funcions d'actualització derivades (usant les funcions d'actualització)

BorraNodeInicial, pop i BorraNodeFinal

```
void BorraNodeInicial (LlistaAlumnes * llista) {
     BorraNode (llista->start, llista);
}
void pop (LlistaAlumnes * llista) {
     BorraNode (llista->start, llista);
}
void BorraNodeFinal (LlistaAlumnes * llista) {
  BorraNode (llista->end, llista);
}
```

Observació

Totes elles usen la funció genèrica `BorraNode`, de la que en són especialitzacions. De fet aquestes funcions no caldrien i solament es creen per consistència amb les de la família "insereix".

Ús de les funcions d'esborrat

Ús de la funció `BorraNode` (busquem i esborrem el primer `Barniol` de la llista)

```
alu = CercaSequencial_cognom ("Barniol", segon_grupA);
if (alu) {
    printf("Borrant l'alumne Barniol\n");
    BorraNode (alu, &segon_grupA);
    ImprimeixLlista (segon_grupA);
} else printf ("Alumne Barniol no trobat\n");
```

Ús de les funcions `BorraNodeInicial`, `pop` i `BorraNodeFinal` (esborrem els dos primers elements de la llista i el darrer)

```
pop (&segon_grupA);
BorraNodeInicial (&segon_grupA);
BorraNodeFinal (&segon_grupA);
```

El primer

El segon ara primer

El darrer

Funcions d'actualització: la funció BorraNode en detall I

```
if (!pertany (node_a_borrar, *llista)) return;
```

Aquesta instrucció acaba correctament la funció si la llista és buida, si `node_a_borrar == NULL` o `node_a_borrar` no pertany a `llista` (error de consistència de paràmetres). Veure la funció `pertany` a la Pàgina 45/165.

En qualsevol d'aquests cassos no s'ha d'esborrar cap node.

```
if (Llargada (*llista) == 1U) {
    free (llista->start); InicialitzaLlista (llista); return;
} // Alternativament es pot usar BorraLLista
```

És auto-explicatiu. Si la llista té un únic element i el volem esborrar estem esborrant tota la llista.

Podem usar `BorraLLista` o esborrar el node sense actualitzar cap apuntador `free (llista->start);` i reinicialitzar la llista `InicialitzaLlista (llista);`.

La funció BorraNode en detall 1/4

Funcions d'actualització: la funció `BorraNode` en detall II

El procediment general

consisteix a redefinir els apuntadors entre el node anterior i el posterior del node que volem esborrar (si existeixen) i seguidament alliberar efectivament el node i actualitzar el nombre d'alumnes de la llista: Aquest és un codi comú a totes les opcions, que s'ha d'executar al final del procés.

```
free (node_a_borrar);
llista->nalumnes--;
```

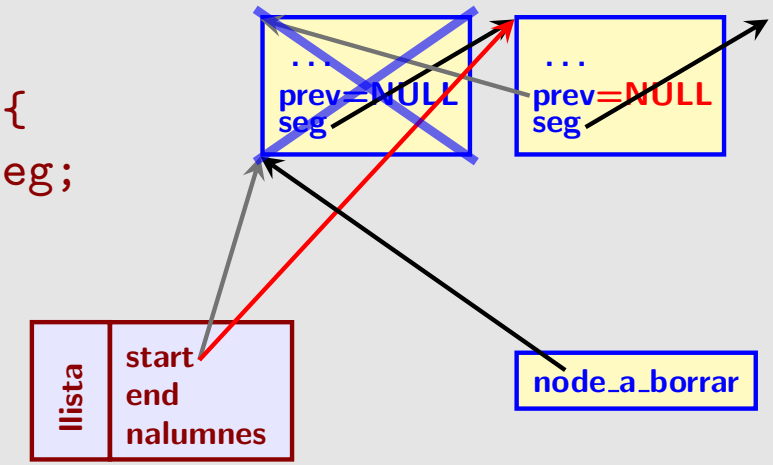
Seguidament passem a discutir en detall cada un dels tres cassos de redefinició dels apuntadors entre el node anterior i el posterior del node que volem esborrar.

La funció `BorraNode` en detall 2/4

Funcions d'actualització: la funció BorraNode en detall III

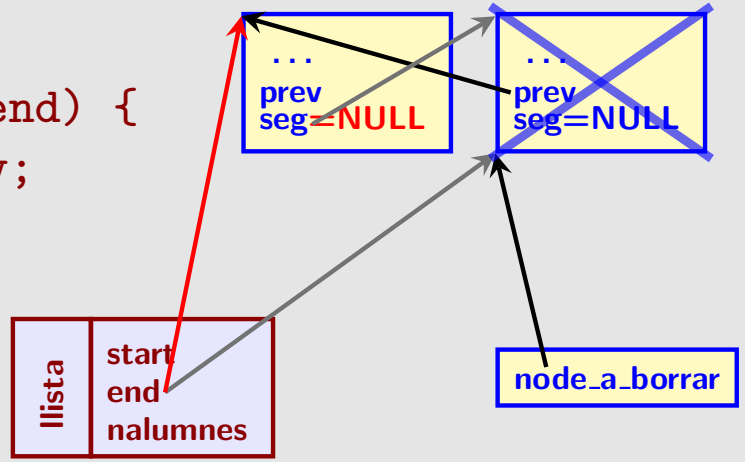
BorraNode — Cas: primer node

```
if (node_a_borrar == llista->start) {
    llista->start = node_a_borrar->seg;
    (llista->start)->prev = NULL;
```



BorraNode — Cas: node final

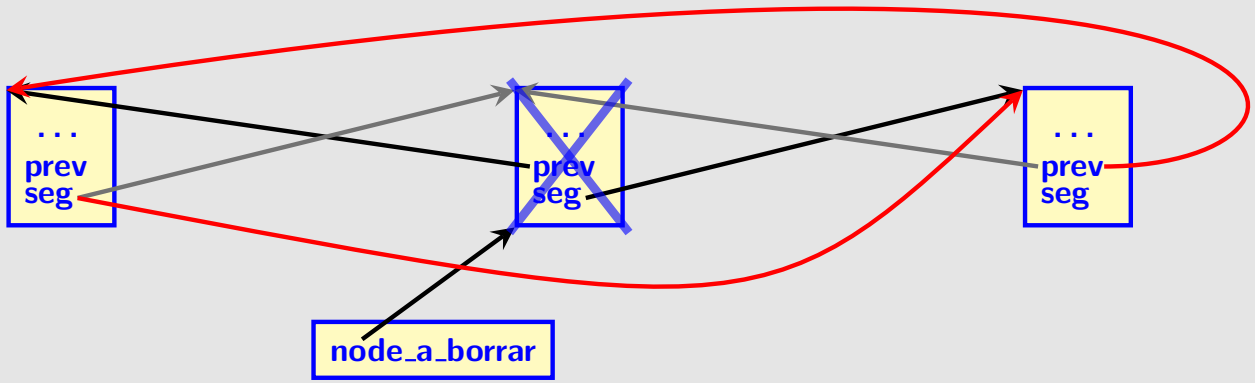
```
} else if (node_a_borrar == llista->end) {
    llista->end = node_a_borrar->prev;
    (llista->end)->seg = NULL;
```



La funció BorraNode en detall 3/4

BorraNode — Cas: node intermedi

```
} else {
    (node_a_borrar->seg)->prev = node_a_borrar->prev;
    (node_a_borrar->prev)->seg = node_a_borrar->seg;
}
```



La funció BorraNode en detall 4/4

Funcions d'actualització: les funcions `push` i `InsereixNodeInicial` en detall

`push` i `InsereixNodeInicial`

```
bool InsereixNodeInicial (LlistaAlumnes * llista) { return push (llista); }
bool push (LlistaAlumnes * llista) { alumne *aux;
  if ((aux = (alumne *) malloc (sizeof (alumne))) == NULL) return false;
  aux->prev = NULL;
  aux->seg = llista->start;
  if (llista->start == NULL) llista->end = aux;
  else (llista->start)->prev = aux;
  llista->start = aux;
  llista->nalumnes++;
  return true;
}
```

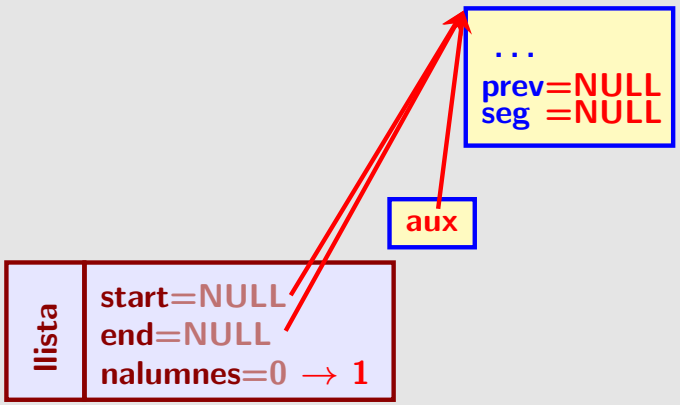
Creació estàndard del nou node

`*aux` serà el nou primer node de la llista.
Per tant el seu apuntador `prev` ha de ser `NULL` i l'hem d'inserir abans de `llista->start` (és dir, el seu apuntador `seg` ha d'apuntar al node `llista->start`).
Veure les figures.

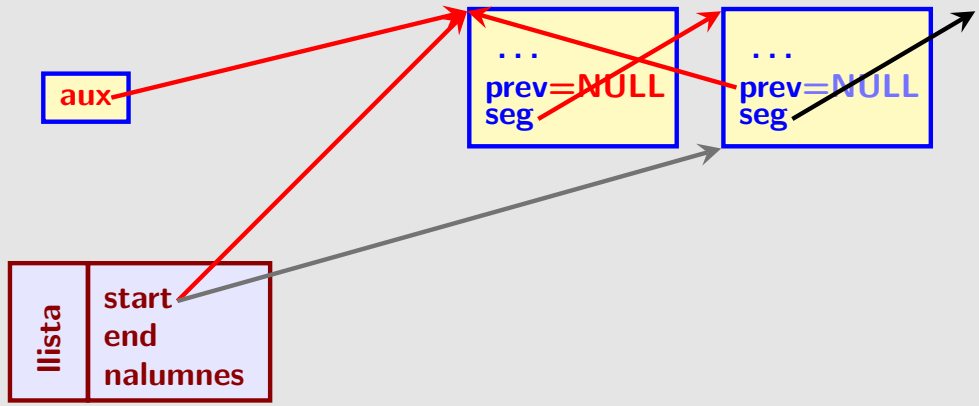
Si la llista no és buida (`llista->start` $\neq$ `NULL`) cal fer que l'anterior de `llista->start` sigui el nou primer node de la llista: `aux`.
Veure la segona figura.

Si la llista és buida (`llista->start=NULL`) aquest és l'únic (i darrer) node de la llista.

Cas de llista buida



Cas general: `llista.start` ja estava inicialitzat



Funcions d'actualització: la funció InsereixNodeFinal

Si la llista és buida (`llista->end=NULL`) aquest és l'únic (i primer) node de la llista.

InsereixNodeFinal

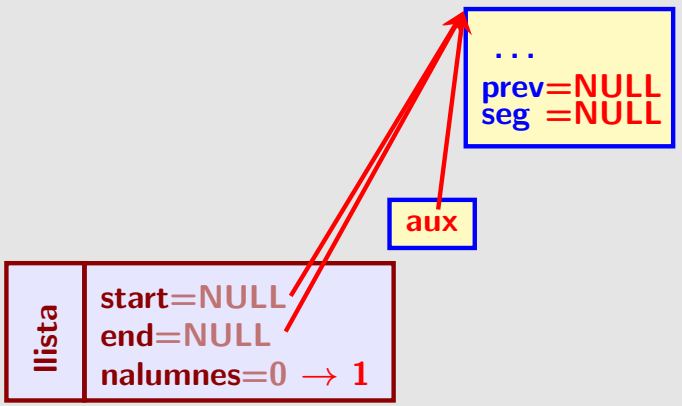
```
bool InsereixNodeFinal (LlistaAlumnes * llista) { alumne *aux;
  if ((aux = (alumne *) malloc (sizeof (alumne))) == NULL) return false;
  aux->seg = NULL;
  aux->prev = llista->end;
  if (llista->end == NULL) llista->start = aux;
  else (llista->end)->seg = aux;
  llista->end = aux;
  llista->nalumnes++;
  return true;
}
```

Creació estàndard del nou node

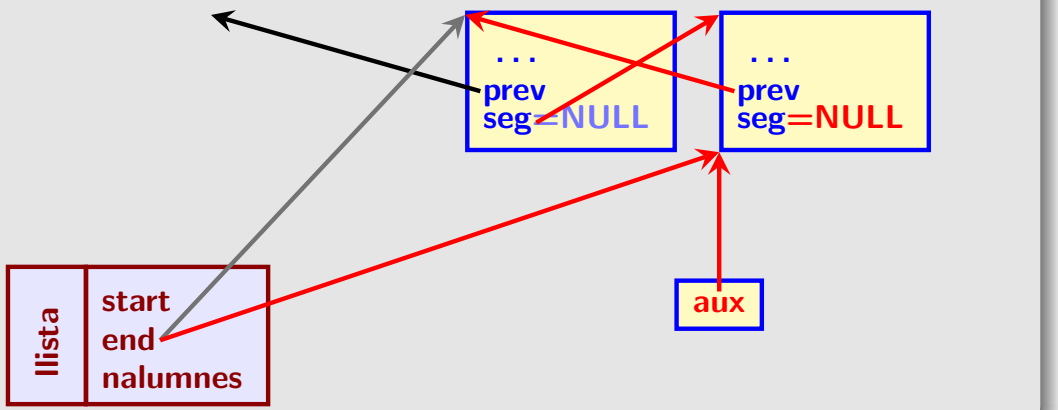
`*aux` serà el nou darrer node de la llista.
Per tant el seu apuntador `seg` ha de ser `NULL` i l'hem d'inserir després de `llista->end` (és dir, el seu apuntador `prev` ha d'apuntar al node `llista->end`).
Veure les figures.

Si la llista no és buida (`llista->end` $\neq$ `NULL`) cal fer que el següent de `llista->end` sigui el nou darrer node de la llista: `aux`.
Veure la segona figura.

Cas de llista buida



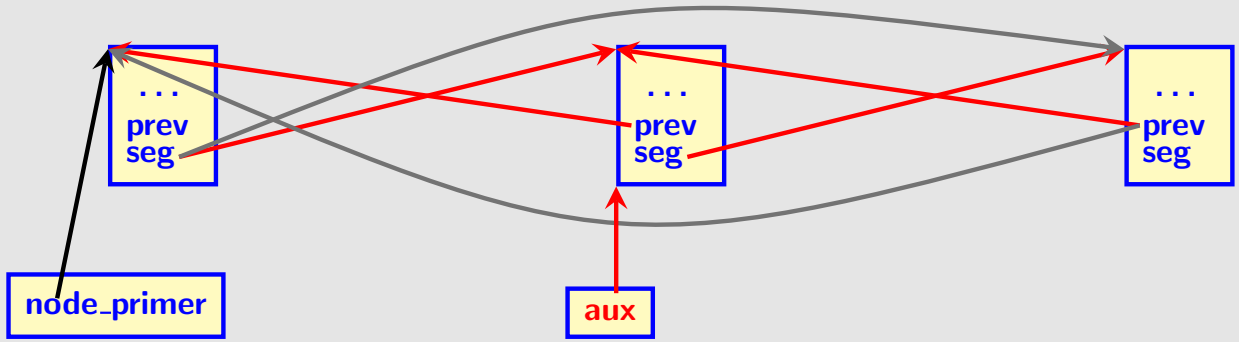
Cas general: `llista.end` ja estava inicialitzat



Funcions d'actualització: la funció InsereixEntreNodes

InsereixEntreNodes

```
alumne * InsereixEntreNodes (alumne * node_primer, LlistaAlumnes * llista) { alumne *aux;
  if (node_primer == NULL ||
      node_primer->seg == NULL) return NULL;   ERROR: node_primer no està definit o no és el primer de dos nodes
  aux = (alumne *) malloc (sizeof (alumne));   Creació estàndard del nou node
  if (aux == NULL) return NULL;                ERROR: de memòria. No es pot crear aux
  aux->prev = node_primer;                     L'anterior del nou node aux és node_primer
  aux->seg = node_primer->seg;                 El següent del nou node aux és el següent de node_primer
  (node_primer->seg)->prev = aux;              L'anterior del següent de node_primer ara serà el nou node aux
  node_primer->seg = aux;                      El següent de node_primer és el nou node aux
  llista->nalumnes++;                          No comprovem que llista és la que correspon a node_primer
  return aux;                                  Retornem l'adreça del node inserit
}
```



Funcions d'actualització derivades (usant les funcions d'actualització)

InsereixNodeabans

```
alumne * InsereixNodeabans (alumne * node_insercio,
                            LlistaAlumnes * llista) {
  if (node_insercio == NULL) {
      if (push (llista)) return llista->start;
      else return NULL;
  }
  if (!pertany (node_insercio, *llista)) return NULL;
  if (node_insercio->prev == NULL) {
      if (push (llista)) return llista->start;
      else return NULL;
  }
  return InsereixEntreNodes (node_insercio->prev, llista);
}
```

Funcions d'actualització derivades (usant les funcions d'actualització) (cont.)

Notes d'implementació de la funció `InsereixNodeabans`

- Usualment trobarem `node_insercio == NULL` només quan la llista sigui buida. En aquest cas cal afegir el primer element de la llista, inicialitzant adequadament tots els camps del contenidor `llista`. Deleguem aquesta feina a la funció `push` amb control d'error: `if (push (llista)) return llista->start; else return NULL;`// Notem que també podríem usar la funció d'afegir a final de llista.
- `if (!pertany (node_insercio, *llista)) return NULL;`
  torna correctament un codi d'error (`return NULL`) si la llista és buida o `node_insercio` no pertany a `llista` (veure la funció `pertany` a 45/165).
- Si `node_insercio->prev == NULL` volem inserir abans del primer element de la llista. Com abans, deleguem aquesta feina a la funció `push` amb control d'error.
- Finalment, `node_insercio->prev != NULL` i deleguem la feina a la funció `InsereixEntreNodes`. El primer paràmetre d'aquesta funció és un apuntador `node_primer` a un element de `llista`, que no pot ser el darrer, i insereix el nou node entre `node_primer` i el seu següent. La inserció *abans* de `node_insercio` s'aconsegueix cridant la funció `InsereixEntreNodes` amb el paràmetre `node_insercio->prev` (que és l'adreça del node anterior al donat).

Funcions d'actualització derivades (usant les funcions d'actualització) (cont.)

InsereixNodedespres

```
alumne * InsereixNodedespres (alumne * node_insercio,
                              LlistaAlumnes * llista) {
  if (node_insercio == NULL) {
      if (InsereixNodeFinal (llista)) return llista->end;
      else return NULL;
  }
  if (!pertany (node_insercio, *llista)) return NULL;
  if (node_insercio->seg == NULL) {
      if (InsereixNodeFinal (llista)) return llista->end;
      else return NULL;
  }
  return InsereixEntreNodes (node_insercio, llista);
}
```

Funcions d'actualització derivades 3/4

Funcions d'actualització derivades (usant les funcions d'actualització) (cont.)

Notes d'implementació de la funció `InsereixNodedespres`

- Quan `node_insercio == NULL` interpretem que volem inserir al final de la llista. Deleguem aquesta feina a la funció `InsereixNodeFinal` amb control d'error:
  ```
  if (InsereixNodeFinal (llista)) return llista->end;
  else return NULL;
  ```
- Com a la funció anterior,
  `if (!pertany (node_insercio, *llista)) return NULL;`
  torna correctament un codi d'error (`return NULL`) si la llista és buida o `node_insercio` no pertany a `llista` (veure la funció `pertany` a la Pàgina 45/165).
- Si `node_insercio->seg == NULL` volem inserir després del darrer element de la llista. Com abans deleguem aquesta feina a la funció `InsereixNodeFinal` amb control d'error.
- Finalment, `node_insercio->seg != NULL` i deleguem la feina a la funció `InsereixEntreNodes` però aquesta vegada l'hi podem passar l'apuntador `node_insercio` ja que volem inserir a continuació.

Ús de les funcions d'inserció: Lectura d'un fitxer i emplenat (per davant) de la llista

Ús de la funció push

```
while (!feof (fin)) { alumne * inici;
    if (!push (&segon_grupA)) return 1;    ERROR. Cal el missatge d'error
    inici = InicideLlista (segon_grupA);   Adreça del nou node
    fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", inici->nom,
            inici->cognoms, &(inici->nteoria), &(inici->npract));
    inici->nfinal = NotaFinal (inici->nteoria, inici->npract);
}
fclose (fin);

if (EsBuida (segon_grupA)) printf ("\nERROR: LLista buida!");
else {
  printf ("S'ha llegit: %d nodes\n", Llargada (segon_grupA));
  ImprimeixLlista (segon_grupA);
}
```

Ús de les funcions d'inserció: Lectura d'un fitxer i emplenat (per davant) de la llista (cont.) — més senzill i sense les comprovacions

Ús de la funció `InsereixNodeabans` (més senzill)

```
while (!feof (fin)) { alumne * inici = NULL;
    inici = InsereixNodeabans (inici, &segon_grupA);
    if(inici == NULL) return 1;    ERROR. Cal el missatge d'error
    fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", inici->nom,
            inici->cognoms, &(inici->nteoria), &(inici->npract));
    inici->nfinal = NotaFinal (inici->nteoria, inici->npract);
}
fclose (fin);
```

Observació

Notem que `inici = InsereixNodeabans (inici, &segon_grupA);` afegeix a principi de llista perquè al començament `inici = NULL`. Les altres vegades `inici` $\neq$ `NULL` i afegeix abans de `inici` (novament a principi de llista).
Notem que `InsereixNodeabans` retorna l'apuntador `inici` al node afegit que sempre és el primer de la llista, que és on estem afegint els elements nous.

Ús de les funcions d'inserció: Lectura d'un fitxer i emplenat (pel final) de la llista

Ús de la funció `InsereixNodeFinal`
Lectura d'un fitxer i emplenat (per darrere) de la llista

```
while (!feof (fin)) { alumne * final;
    if (!InsereixNodeFinal (&segon_grupA)) return 1;
    final = FinaldeLlista (segon_grupA);
    fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", final->nom,
            final->cognoms, &(final->nteoria), &(final->npract));
    final->nfinal = NotaFinal (final->nteoria, final->npract);
} ; fclose (fin);
```

Ús de la funció `InsereixNodedespres` — més simple

```
while (!feof (fin)) { alumne * final = NULL;
    final = InsereixNodedespres (final, &segon_grupA);
    if(final == NULL) return 1;
    fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", ....
} ; fclose (fin);
```

Ús de les funcions d'inserció

Un exemple general d'ús de les tres funcions base: `push`, `InsereixEntreNodes` i `InsereixNodeFinal` es pot trobar a les funcions derivades `InsereixNodeabans` i `InsereixNodedespres`.

A continuació es dona un exemple d'inserció d'un node al lloc corresponent segons l'ordre de nota final (cerca numèrica) per mantenir una ordenació. Usem les funcions `InsereixNodeabans` i `InsereixNodeFinal`.

Les dades del node s'entren per pantalla.

Ús de les funcions d'inserció: segons l'ordre de la llista

Ús de les funcions InsereixNodeabans i InsereixNodeFinal

```
char nom[20], cognoms[40]; float npract, nteoria, nfinal;

printf ("Entra nom;cognoms;nteoria;npract: ");
scanf ("%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f",
        nom, cognoms, &nteoria, &npract);
nfinal = 0.6*nteoria + 0.4 * npract;

alu = CercaSequencial_nota_ge (nfinal, segon_grupA);
if(alu == NULL) { // nfinal és la mes gran de la llista. Cal inserir al final
   if( ! InsereixNodeFinal(&segon_grupA) ) return 1;  ERROR. Cal el missatge d'error
   alu = FinaldeLlista(segon_grupA);
} else { En aquest cas alu->nfinal >= nfinal i, per tant, cal inserir abans d'alu
   alu = InsereixNodeabans (alu, &segon_grupA);
}
if (alu == NULL) return 1; ERROR. Controla l'èxit de FinaldeLlista i InsereixNodeabans
strcpy (alu->nom, nom); strcpy (alu->cognoms, cognoms);
alu->nteoria = nteoria; alu->npract = npract; alu->nfinal = nfinal;
printf ("Comprovació inserció");
ImprimeixLlista (segon_grupA);
```

Exemple senzill d'aplicació: Successions de Farey

Definició

La *successió de Farey d'ordre n* ($n \geq 1$), $\mathcal{F}_n$, és la successió de racionals entre 0 i 1 ordenats en ordre creixent de magnitud i tals que, en fracció irreductible, tenen denominadors menors o iguals que n.
Cada successió de Farey comença amb el valor 0, denotat per la fracció $\frac{0}{1}$, i acaba amb el valor 1, denotat per la fracció $\frac{1}{1}$.
Dues fraccions consecutives a $\mathcal{F}_n$ s'anomenen *veïns de Farey*.

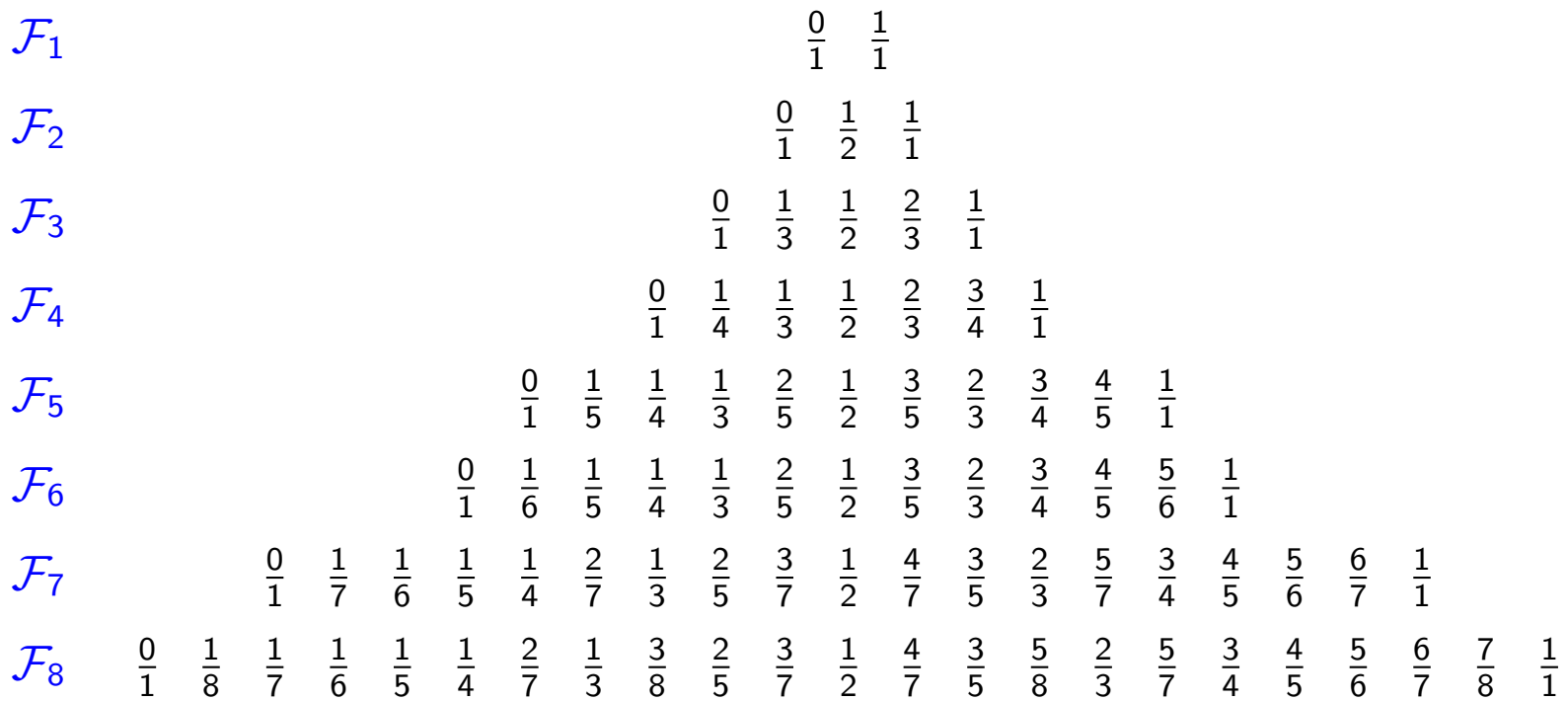
Veïns de Farey

1. $\frac{p}{q} < \frac{r}{s}$ són veïns de Farey a $\mathcal{F}_{\max\{q,s\}}$ si i només si $qr - ps = 1$.
2. Si $\frac{p}{q} < \frac{r}{s}$, llavors $\frac{p}{q} < \frac{p+r}{q+s} < \frac{r}{s}$. A més, si $\frac{p}{q}$ i $\frac{r}{s}$ són veïns de Farey, $\frac{p}{q} < \frac{p+r}{q+s}$ i $\frac{p+r}{q+s} < \frac{r}{s}$ són veïns de Farey a $\mathcal{F}_{q+s}$.

Suma de Farey

La fracció $\frac{p+r}{q+s}$ s'anomena *suma de Farey de* $\frac{p}{q}$ *i* $\frac{r}{s}$.

Algunes successions de Farey — L'arbre de Farey



Les successions de Farey $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}_5$, $\mathcal{F}_6$, $\mathcal{F}_7$ i $\mathcal{F}_8$ organitzades en el que s'anomena l'*arbre de Farey*

Càlcul de les successions de Farey

L'algorisme estàndard de càlcul de les successions de Farey és usar la propietat (1) de la transparència 69/165. Donats p i q, 'algorisme consisteix a iterar sobre la variable s mitjançant un bucle per a trobar-ne un valor tal que l'equació $qr - ps = 1$ tingui solució per un r enter. Notem que aquest és un algorisme de complexitat quadràtica.

Un algorisme ràpid de càlcul d'$\mathcal{F}_n$ consisteix a construir-la iterativament a partir d'$\mathcal{F}_1$ usant la propietat (2) de la transparència 69/165 (comproveu-ho a la transparència 70/165). Una bona estratègia per a guardar $\mathcal{F}_n$ en memòria és usar una llista enllaçada de mida arbitrària.

A més:

Observació

Per definició, $\mathcal{F}_n$ és simètrica respecte de $\frac{1}{2}$.
Això vol dir que si $\frac{0}{1} \leq \frac{p}{q} \leq \frac{1}{2}$ pertany a $\mathcal{F}_n$, llavors $1 - \frac{p}{q} = \frac{q-p}{q}$ també hi pertany.

Per tant, per a calcular $\mathcal{F}_n$ amb $n \geq 2$, de fet, solament cal calcular-ne els elements $\frac{0}{1} \leq \frac{p}{q} \leq \frac{1}{2}$ i afegir-hi $1 - \frac{p}{q} = \frac{q-p}{q}$ en ordre invers ($\mathcal{F}_1$ s'ha de tractar com a cas especial).

Càlcul de les successions de Farey 1/2

Càlcul de les successions de Farey (cont.)

Degut al que hem dit abans hem de recórrer la llista en ordre directe i invers. Això justifica la necessitat d'usar una llista doble.

Aquest algorisme es mostra a les dues transparències següents.

El procediment és lluny de ser el millor pel que fa a gestió de memòria però, quant a velocitat, probablement és el més ràpid.

Càlcul de les successions de Farey 2/2

Codi del càlcul de les Successions de Farey per $n \geq 2$ (I)

Definició i funcions de la llista enllaçada

```
typedef struct racional {
  unsigned num;
  unsigned den;
  struct racional *seg, *prev;
} racional;
```

Imprimeix $\mathcal{F}_n$ usant la simetria anterior. No val per cas $n = 1$.
El primer bucle imprimeix la part calculada d'$\mathcal{F}_n$. S'acaba quan `start` apunta al darrer node ($\frac{1}{2}$), que no s'imprimeix.
El segon bucle complementa la meitat calculada d'$\mathcal{F}_n$ usant la simetria. Imprimeix $1 - \frac{p}{q} = \frac{q-p}{q}$ en ordre invers començant pel darrer node ($\frac{1}{2}$ — sense excloure'l).

```
void ImprimeixFareySucc(racional *start){
   for ( ; start->seg ; start=start->seg) printf("%u/%u ", start->num, start->den);
   for ( ; start ; start=start->prev) printf("%u/%u ", start->den - start->num, start->den);
   printf("\n\n");
}

racional *InsereixNodeAContinuacio(racional *pq, int num, unsigned den){ racional *aux;
     if ((aux = (racional *) malloc (sizeof(racional))) == NULL) return NULL;
     aux->num=num; aux->den = den;

     if (pq == NULL) { aux->seg=NULL; aux->prev = NULL; }
     else { aux->seg=pq->seg; aux->prev=pq;
            if (pq->seg) pq->seg->prev=aux; pq->seg=aux;
     }
     return aux;
}

racional *InsereixFarey(racional *pq){
     if (pq == NULL || pq->seg == NULL) return NULL;
     return InsereixNodeAContinuacio(pq, pq->num + pq->seg->num, pq->den + pq->seg->den);
}
```

`InsereixNodeAContinuacio` és una funció estàndard d'inserció de nodes

`InsereixFarey` és una especialització (per simplicitat) de `InsereixNodeAContinuacio` pel cas de successions de Farey. Ja implementa que el racional sigui la suma de Farey dels dos nodes entre els que s'insereix.

Codi del càlcul de les Successions de Farey per $n \geq 2$ (II)

El programa

```
#include <stdio.h>
#include <stdlib.h>
void Error(const char miss[], int ret) { fprintf (stderr, "\nERROR: %s\n\n", miss); exit(ret); }

int main (int argc, char *argv[]) {
  unsigned  ordre, i;
  racional *start = NULL, *curr;

  if (argc !=2) { fprintf (stderr, "\nÚs: %s ordre\n\n", argv[0]); return 1; }
  if (atoi(argv[1]) < 1) Error("valor ilegal d'ordre", 2); ordre = atoi(argv[1]);

  if (ordre == 1) { printf("0/1 1/1\n\n"); return 0; }

   if ((start = InsereixNodeAContinuacio(NULL, 0, 1U)) == NULL) Error("al inserir un node", 5);
   if (!InsereixNodeAContinuacio(start, 1, 2U)) Error("al inserir un node", 5);

   for (i=3; i<=ordre; i++){
       for (curr=start; curr->seg; curr=curr->seg)
           if (curr->den + curr->seg->den <= ordre && !InsereixFarey(curr))
                                                  Error("al inserir un node de Farey", 6);
   }
   ImprimeixFareySucc(start);
   return 0;
}
```

Cas especial

Inicialització de la successió. Si `ordre = 2`; ja hem acabat. Podem imprimir

Calcula la primera meitat d'$\mathcal{F}_n$ iteративament per $n = 2, 3, \ldots,$`ordre` a partir d'$\mathcal{F}_{n-1}$. A cada iteració afegim les sumes de Farey de veïns de la successió anterior que pertanyin a $\mathcal{F}_{\texttt{ordre}}$ (`curr->den + curr->seg->den <= ordre`)

Resultats

```
./Farey.exe 10
0/1 1/10 1/9 1/8 1/7 1/6 1/5 2/9 1/4 2/7 3/10 1/3 3/8 2/5 3/7 4/9
                    1/2 5/9 4/7 3/5 5/8 2/3 7/10 5/7 3/4 7/9 4/5 5/6 6/7 7/8 8/9 9/10 1/1
```

Cerca binària en llistes enllaçades dobles

Introducció i motivació (adaptat d'un post d'el *guru Bobby* a stack**overflow**)

Una aplicació directa de la cerca binària en una llista doblement enllaçada funcionaria mitjançant el càlcul dels índexs dels punts migs de cada iteració (igual que en el cas d'un vector) i, a continuació, accedir a cada un a partir de l'inici de la llista saltant un nombre adequat de passos.

Aquest procediment és, de fet, molt lent (i clarament pitjor que la cerca seqüencial). Per exemple, si l'element que es busca és al final de la llista, cada localització del punt mig de l'interval és més llarga i similar a una cerca seqüencial de tota la llista (ometent les comparacions de claus).

Això dóna una complexitat de $\mathcal{O}(n \log(n))$ (més gran que $\mathcal{O}(n) \approx \frac{n}{2}$ per la cerca seqüencial) a aquesta implementació de l'algorisme. No obstant això, podem accelerar aquest procés a una una complexitat $\mathcal{O}(n)$ (com la cerca seqüencial), en triar un enfocament mes hàbil del problema.

L'algorisme anterior és lent perquè cada vegada que busquem un punt mig d'un interval iniciem la cerca des del principi de la llista.

Cerca binària en llistes enllaçades dobles (cont.)

Introducció i motivació (adaptat d'un post d'el *guru Bobby* a stackoverflow)

Observem que, de fet, no necessitem fer això. Després d'arribar al punt mig de l'interval la primera vegada (posició $\frac{n}{2}$ de la llista) sabem que la propera consulta que farem serà en la posició $\frac{n}{4}$ o bé en la $\frac{3n}{4}$, que és només a distància $\frac{n}{4}$ del punt on som en aquell moment (enfront de les distàncies $\frac{n}{4}$ o $\frac{3n}{4}$ si partim del principi de la llista).

Òbviament, el que cal fer és saltar directament a $\frac{n}{4}$ o $\frac{3n}{4}$ des de $\frac{n}{2}$ i repetir aquest proces en totes les iteracions de l'algorisme en lloc d'iniciar tots els moviments des de l'inici de la llista.

Observem que el desplaçament és ara de l'ordre de la meitat de l'interval de cerca i això es repetirà durant tot l'algorisme. Per tant, la llargada dels desplaçaments serà $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$, que tendeix a zero al iterar. Per tant, el treball total a realitzar, en el pitjor dels casos, és de només $\mathcal{O}(n)$, que és molt millor que el $\mathcal{O}(n \log(n))$ discutit abans.

La pregunta natural ara és per què cal usar cerca binària si la seqüencial tarda el mateix?

Cerca binària en llistes enllaçades dobles (cont.)

Introducció i motivació (adaptat d'un post d'el *guru Bobby* a stackoverflow)

Un gran avantatge d'aquest enfocament és que tot i que el temps d'execució és $\mathcal{O}(n)$ (com la cerca seqüencial), només hem de fer, en total, $\mathcal{O}(\log(n))$ comparacions de claus (un per pas de la recerca binària — les altres operacions corresponen a desplaçaments sense comparacions). En el cas que les comparacions siguin "cares" (per exemple comparant strings llargs) podríem acabar fent menys feina utilitzant una cerca binària que amb la cerca lineal normal.

Per exemple en una llista de mida n fem $\frac{n}{2}$ comparacions i desplaçaments en mitjana per la cerca seqüencial, mentre que amb cerca binària fem aproximadament n desplaçaments i $\lceil \log_2(n) \rceil$ comparacions (on $\lceil x \rceil$ denota l'enter mes petit que és mes gran o igual que x).

Cerca binària en llistes enllaçades dobles (cont.)

Introducció i motivació (adaptat d'un post d'el *guru Bobby* a stack**overflow**)

Prenem com a unitat de temps el temps que ens costa fer un desplaçament i anomenem κ el nombre d'unitats de temps que ens costa fer una comparació. Llavors, el temps mitjà d'una cerca seqüencial és $\frac{n}{2}(1+\kappa)$ i el d'una cerca binària és aproximadament $n + \lceil \log_2(n) \rceil \kappa$. L'inequació

$$n + \lceil \log_2(n) \rceil \kappa < \tfrac{n}{2}(1+\kappa)$$

és equivalent a

$$\kappa > \frac{n}{n - 2\lceil \log_2(n) \rceil}.$$

Per exemple per una llista de mida $n = 2^{22} = 4194304$ elements tenim que és millor usar cerca binària a partir de $\kappa \approx 1.0000105$ comparacions (encara que això una vegada més depèn de la optimització del codi i de l'ús que el programa faci del "cache").

Cerca binària en llistes enllaçades dobles (cont.)

CercaBinaria en versió punt mig pm i amplada de l'interval ai

```
alumne * CercaBinaria_cognom (const char *cognom,
                              LlistaAlumnes llista) {
  register unsigned start = 0UL, afterend = llista.nalumnes,
                                 middle_ant = 0U, middle;
  register alumne *middle_alu = llista.start;
  register unsigned short cognomlen = strlen (cognom);
  register int rescom;

  while (afterend > start) {
      middle = start + ((afterend - start - 1) >> 1);
      middle_alu = DeSaltaA (middle_ant, middle_alu, middle);
      middle_ant = middle;
      rescom = strncmp (cognom, middle_alu->cognoms, cognomlen);
      if (rescom == 0) return middle_alu;
      else if (rescom > 0) start = middle + 1;
      else afterend = middle;
    }
  return NULL;
}
```

Cerca binària en llistes enllaçades dobles (cont.)

Ús de la funció `CercaBinaria` (busquem `Lopez` – el primer)

```
alu = CercaBinaria_cognom ("Lopez", segon_grupA);
if (alu) { printf ("Alumne trobat: "); ImprimeixNode (alu); }
else printf ("Alumne Lopez no trobat\n");
```

Observació (una motivació per usar llistes dobles)

Com hem vist abans, la cerca binària solament té sentit en llistes dobles on podem anar sense passar per l'origen als centres dels intervals. En llistes simples a cada iteració cal passar per l'origen de la llista, que fa la cerca molt més ineficient que la cerca seqüencial (a menys que el temps de les comparacions sigui *enorme*).

Cerca binària en llistes enllaçades dobles (cont.)

Implementació de `CercaBinaria`

Es totalment anàloga a la funció corresponent que es mostra a les transparències d'**Estructures en C** amb les següents diferències:

- El fet d'usar una llista enllaçada controlada per una estructura `LlistaAlumnes` canvia els paràmetres d'entrada i `afterend = llista.nalumnes`.
- Apart de calcular el punt mig de la llista hem de recordar i mantenir el punt mig anterior `middle_ant` (inicialitzat a `0U`) i actualitzat amb `middle_ant = middle;`.
- Per altra banda l'accés al punt mig de la llista ara s'ha de fer amb un apuntador `register alumne *middle_alu` (inicialitzat a `llista.start`).
- El desplaçament de `middle_alu` (actualització) des de `middle_ant` a la posició `middle` ara ens correspon a nosaltres i no al sistema (i és "car"). El fem usant la funció de salt absolut de `middle_ant` a `middle`: `middle_alu = DeSaltaA (middle_ant, middle_alu, middle);`

Ordenació de llistes enllaçades dobles

Tractarem dues estratègies:

- Ús d'un vector d'apuntadors intermedi (com ja s'ha estudiat a les a les transparències d'**Estructures en C**). Una vegada ordenat el vector d'apuntadors podem reordenar la llista o podem mantenir-lo com una indexació addicional de la llista.
- Reordenació directa de la llista. L'algorisme que sembla més adequat per a reordenar llistes enllaçades és el *Merge Sort*. El discutirem en la seva versió recursiva i no recursiva.

Observació

A diferencia de la cerca binària, els algorismes d'ordenació no necessiten llistes doblement enllaçades. Solament usen una direcció. De fet l'aplicació d'aquests algorismes a llistes dobles és una mica més complicada ja que cal mantenir la duplicitat d'enllaços.

La discussió la farem per llistes dobles per completesa.

Ordenació de llistes enllaçades dobles (cont.)

Per a desenvolupar els algorismes d'ordenació (i per variar 😉) usarem un exemple nou: Els nodes d'un mapa que s'usa en problemes de *routing*.

Els elements de la llista `node_map`

```
typedef struct NODE
{ char id[11];          Clau d'identificació del node
  char *name;
  double lat, lon;      Posició del node al mapa: latitud i longitud
  struct NODE *prev, *seg;
} node_map;
```

El contenidor `map` de la llista i la funció d'inicialització `IniMap`

```
typedef struct
{ node_map *start;
  node_map *end;
  unsigned long nnodes;
} map;
```

```
void IniMap (map * llista){
  llista->start = llista->end = NULL;
  llista->nnodes = 0UL;
}
```

Ordenació de llistes enllaçades dobles 2/3

Ordenació de llistes enllaçades dobles (cont.)

Les dades — el fitxer de nodes del mapa (per fixar idees)

```
### Maximum number of fields: 5306
### Maximum width of @name: 184 chars
### Number of nodes: 23895681
### Format: node|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|node_lat|node_lon
node|171773|||||||38.6048094|-0.0489952
node|171933|||traffic_signals|||||40.4200658|-3.7016652
node|171948|||||||40.4202342|-3.6877944
node|448261|Durango||||||43.1796927|-2.6427547
node|448262|||||||43.1792494|-2.6396184
node|449015|Iurreta/Durango/BI-623 Vitoria-Gasteiz||motorway_junction||88|||43.1811634|-2.6493499
node|449017|||||||43.1839004|-2.6569465
node|449092|BI-2713 Larrabetzua/N-634||motorway_junction||27|||43.2483724|-2.803004
node|449104|||||||43.2608098|-2.8109874
        ......................
```

Aquest procediment *no* reordena efectivament la llista. Crea un nou índex d'ordenació (vector d'apuntadors a estructures) i l'ordena amb el criteri especificat mitjançant la funció de comparació.

És un mètode ràpid, senzill i eficient d'ordenació.

Un avantatge és que, en acabar, tenim la llista ordenada de dues maneres: l'ordenació donada pels enllaços (que es pot recórrer endavant i endarrere) i l'ordenació donada pel vector d'apuntadors, que funciona com a índex de les dades. En aquest cas, l'índex també es pot recórrer endavant i endarrere donat que és un vector estàndard.

Si necessitem diverses ordenacions de la llista aquesta és la millor manera de procedir; creant diversos índexs.

Els desavantatges d'aquesta opció són:

- El vector d'índex usa més memòria (l'equivalent a un apuntador per a cada estructura — que no és gaire; això és el que costa passar una llista simple a doble),
- L'index no és fàcil d'actualitzar en afegir o esborrar dades.

Ordenació de llistes enllaçades dobles: vector d'índex 1/5

En aquest cas és convenient modificar el contenidor `map` de la llista i la funció d'inicialització `IniMap` com segueix:

El contenidor `map` de la llista i la funció d'inicialització `IniMap`

```
typedef struct
{ node_map *start;
  node_map *end;
  unsigned long nnodes;
  node_map **index;
} map;
```

```
void IniMap (map * llista){
  llista->start = llista->end = NULL;
  llista->nnodes = 0UL;
  llista->index=NULL;
}
```

Ordenació de llistes enllaçades dobles: vector d'índex 2/5

Ordenació de llistes enllaçades dobles (cont.)

Creació d'un vector d'índex d'apuntadors a estructures

A més, per les dues funcions que usen un vector d'apuntadors a estructures com a estratègia d'ordenació, necessitarem una funció de comparació apropiada:

La funció de comparació ComparaID

```
int ComparaID (const void *a, const void *b) {
    return -strcmp ( (*((node_map **) a))->id,
                     (*((node_map **) b))->id );
}
```

Observació

Si tenim diverses funcions d'ordenació podem implementar diferents ordenacions "en calent", a partir de comandes d'usuari entrades pel teclat o de menús.

Funcionament de ComparaID

Recordem la discussió que ja hem fet a les transparències d'**Estructures en C**: `a` i `b` es declaren com apuntadors de tipus *void* però, en realitat, són apuntadors a algun element del vector `index` d'una estructura del tipus `map` (posem que `a` és un apuntador `void` a `list->index[1]`).
Per accedir al `id` del node apuntat per `list->index[1]`, en primer lloc cal forçar el tipus d'`a` a un apuntador del mateix tipus que `list->index` (que ha estat declarat com `node_map **`). Aquest apuntador és `(node_map **) a` (que apunta a `list->index[1]` amb el tipus correcte).
Llavors, `*((node_map **) a) = list->index[1]` i, per tant, s'accedeix al camp `id` del node apuntat per `list->index[1]` amb `(*((node_map **) a))->id`.

Ordenació de llistes enllaçades dobles: vector d'índex 3/5

Ordenació de llistes enllaçades dobles (cont.)

Creació d'un vector d'índex d'apuntadors a estructures

La funció CreaIndex

```
void CreaIndex (map * list,
           int (*compare) (const void *, const void *)) {
  register unsigned long i;

  if (list->nnodes < 2 ) return;

  list->index =
        (node_map **) malloc (list->nnodes * sizeof (node_map *);
  if (list->index == NULL) return;

  for ( list->index[0]=list->start, i=1;
        i < list->nnodes;
        list->index[i]=list->index[i-1]->seg, i++ ) ;

  qsort(list->index, list->nnodes, sizeof(node_map *), compare);
}
```

Per a usar una funció de comparació. Dins de `CreaIndex` la funció es diu `compare`.

En aquest cas no hi ha res per ordenar. A la resta de la funció `list->nnodes >= 2`.

Creació del vector d'índex.

Procediment estàndard d'inicialització del vector d'índex `list->index` (ja explicat a les transparències d'**Estructures en C**).

Ordenació usual de vectors amb `qsort`

Ordenació de llistes enllaçades dobles: vector d'índex 4/5

Ordenació de llistes enllaçades dobles (cont.)

Creació d'un vector d'índex d'apuntadors a estructures

Ús de la funció `CreaIndex` (ometent la funció `LlegeixDadesDelMapa`)

```
map mapa_c;

IniMap (&mapa_c);
LlegeixDadesDelMapa(&mapa_c);

CreaIndex (&mapa_c, ComparaID);
if(!mapa_c.index) fprintf (stderr, "Error a l'index ordenat\n");
ImprimeixLlista_Index (mapa_c);
```

`&mapa_c` perquè modifiquem l'estructura (al carregar `mapa_c.index`).

La funció auxiliar `ImprimeixLlista_Index`

Exemple de recorregut seqüencial usant un vector d'índex

```
void ImprimeixLlista_Index (map llista) {
  register unsigned long n;
  if(llista.index == NULL) return;
  for (n=0UL; n < llista.nnodes; n++)
              ImprimeixNode (llista.index[n]);
}
```

Aquest procediment reordena efectivament la llista amb l'ajut d'un vector d'apuntadors intermedi (com ja s'ha estudiat a les transparències d'**Estructures en C**). Una vegada ordenat el vector d'apuntadors es reordena la llista.

És un mètode bastant ràpid i senzill d'ordenació.

Els desavantatges d'aquesta opció són:

- Tarda més que el procediment anterior ja que, a més de l'ordenació del vector d'apuntadors, cal reordenar la llista separadament.
- El vector d'índex usa més memòria (l'equivalent a un apuntador per a cada estructura — equivalent al que costa passar una llista simple a doble). Encara que l'increment d'ús de memòria sigui relativament petit, quan tenim grans quantitats de dades (i estem al límit de la memòria) això pot ser un problema.

Per a tornar un codi de control/error del resultat.

La funció OrdenaUsantIndex

```
int OrdenaUsantIndex (map * list, int (*compare) (const void *, const void *))
{ node_map **ptr_list; register unsigned long i;

  if (!list || !list->start ) return 1;
  if (list->nnodes == 1 ) return 0;

  ptr_list = (node_map **) malloc (list->nnodes * sizeof (node_map *));
  if (ptr_list == NULL) return 3;

  for(ptr_list[0]=list->start, i=1; i < list->nnodes;
                                    ptr_list[i]=ptr_list[i-1]->seg, i++ ) ;

  qsort(ptr_list, list->nnodes, sizeof(node_map *), compare);

  list->start = ptr_list[0]; list->end=ptr_list[list->nnodes-1];
  (list->start)->prev = (list->end)->seg = NULL;
  for(i=1 ; i < list->nnodes ; i++) {
    ptr_list[i-1]->seg = ptr_list[i]; ptr_list[i]->prev = ptr_list[i-1];
  }

  free(ptr_list);
  return 0;
}
```

Adaptat de la funció CreaIndex per a tornar un codi de control/error del resultat.

Fins aquí, similar a la funció CreaIndex.

Alliberament del vector auxiliar d'índex.

Ordenació de llistes enllaçades dobles (cont.)

Ordenació mitjançant un vector intermedi d'apuntadors a estructures

La reestructuració dels apuntadors de la llista a partir de l'índex ordenat

```
list->start = ptr_list[0]; list->end=ptr_list[list->nnodes-1];
(list->start)->prev = (list->end)->seg = NULL;
for(i=1 ; i < list->nnodes ; i++) {
  ptr_list[i-1]->seg = ptr_list[i]; ptr_list[i]->prev = ptr_list[i-1];
}
```

Funcionament:

- `list->start = ptr_list[0]; list->end=ptr_list[list->nnodes-1];`: estableix l'inici i el final de la llista.
- `(list->start)->prev = (list->end)->seg = NULL;`: estableix els apuntadors de "finalització" de la llista.

Pels altres nodes (`for(i=1 ; i < list->nnodes ; i++)`):

- `ptr_list[i-1]->seg = ptr_list[i];`: estableix que `ptr_list[i];` és el següent de `ptr_list[i-1]`, i
- `ptr_list[i]->prev = ptr_list[i-1];`: estableix que `ptr_list[i-1];` és l'anterior de `ptr_list[i]`.

Ús de la funció `OrdenaUsantIndex` (ometent la funció `LlegeixDadesDelMapa`)

```
map mapa_c;

IniMap (&mapa_c);
LlegeixDadesDelMapa(&mapa_c);

int err = OrdenaUsantIndex (&mapa_c, ComparaID);
if(err) fprintf (stderr, "Error a l'index ordenat\n");
ImprimeixLlista (mapa_c);
```

Ara no tenim índex. Hem reestructurat la llista. Impressió estàndard.

L'algorisme *Merge Sort* reordena directament la llista sense l'ajut d'un vector d'apuntadors intermedi. És l'algorisme que sembla més adequat per a reordenar llistes enllaçades.

Té complexitat $\mathcal{O}(n \log n)$ i és del tipus *divideix i venceràs*. A més és fàcilment paral·lelitzable.

Algorisme Merge Sort (John von Neumann, 1945)

1. Dividir la llista en n llistes d'un únic element cada una. Òbviament, cada una d'aquestes llistes està ordenada.
2. Per $k = 0, 1, 2, \ldots, \widetilde{k}$ fusionar dues llistes consecutives (ja ordenades) de mida 2^k en una única llista ordenada de mida 2^{k+1}. El darrer pas es produeix per $\widetilde{k}$ tal que $2^{\widetilde{k}} < n \leq 2 \cdot 2^{\widetilde{k}}$ i consisteix a barrejar una llista de mida $2^{\widetilde{k}}$ amb una de mida $n - 2^{\widetilde{k}} \leq 2^{\widetilde{k}}$ per a obtenir la llista de mida n ordenada.

Començarem amb la versió recursiva de l'algorisme Merge Sort, que és més simple (més avall discutirem la no recursiva).

És un mètode bastant ràpid i senzill d'ordenació.

El desavantatge principal de la recursivitat és que queden tasques pendents a l'stack del programa durant la seva execució. Amb grans quantitats de dades l'stack pot arribar a ser gran i, si estem al límit de la memòria, això pot ser un problema.

La implementació de l'algorisme es fa separant dues tasques:

- la funció `MergeSort_PasRecursiu` s'encarrega de la tasca de divisió de cada llista en dues semi-llistes i, una vegada s'han ordenat, crida a
- la funció encarregada de la tasca de barrejar les dues semi-llistes ordenades en una sola llista ordenada de mida ($\approx$) doble (`BarrejaDuesLlistesOrdenades`).

En tot aquest procés tractarem la llista com si fos simple. Solament ens preocuparem de mantenir els apuntadors endavant en cada una de les iteracions.

Donada la representació de dades que hem triat, la funció que usarem (`MergeSortRecursiu`) solament fa el paper de funció "capçalera": inicia el procés i, en acabat, normalitza la llista creant els enllaços endarrere. Ordenació de llistes enllaçades dobles: Merge Sort recursiu 3/13

La reconstrucció dels apuntadors endarrere es delega a la següent funció:

La funció `ReconstrueixApuntadorsprev`

```
void ReconstrueixApuntadorsprev (map *list) {
 list->start->prev = NULL;
 for ( list->end = list->start;
       (list->end)->seg;
       ((list->end)->seg)->prev = list->end, list->end = (list->end)->seg );
}
```

La funció de comparació en aquest cas és:

La funció de comparació `ComparaID`

```
int ComparaID (const void *a, const void *b) {
    return -strcmp (((node_map *) a)->id, ((node_map *) b)->id);
}
```

Observació

Com abans, si tenim diverses funcions d'ordenació podem implementar diferents ordenacions "en calent", a partir de comandes d'usuari entrades pel teclat o de menús.

Funcionament de `ReconstrueixApuntadorsprev`

- `list->start->prev = NULL;`: Garanteix el conveni que fixa que l'apuntador inicial endarrere és `NULL`.
- `for ( list->end = list->start`: Usarem `list->end` de variable del bucle. Comencem al principi de la llista.
- `(list->end)->seg`: Iterem mentre `(list->end)->seg`$\neq$`NULL`. Equivalentment, el bucle acaba quan `list->end` apunta al final de la llista. Això deixa `list->end` correctament definit al final del procés.
- `((list->end)->seg)->prev = list->end`: Fa que l'apuntador `prev` del node següent a l'actual `(list->end)->seg` apunti al node actual `list->end`. És la instrucció més important, que estableix els apuntadors endarrere. El control del bucle assegura que això es fa mentre hi ha un següent i, per tant, no es fa pel darrer node del bucle.
- `list->end = (list->end)->seg`: Actualitza el node actual. A la següent iteració `list->end` serà el següent de l'actual; és a dir, el que ara és `(list->end)->seg`.

La funció MergeSortRecursiu

```
void MergeSortRecursiu (map *list, int (*compare) (const void *, const void *)) {
 list->start = MergeSort_PasRecursiu(list->start, list->nnodes, compare);
 ReconstrueixApuntadorsprev(list);
}
```

El funcionament de `MergeSortRecursiu` és clar.

De fet és la funció `MergeSort_PasRecursiu` qui fa la feina. Per aquest motiu li passem la funció `compare`.

Aquesta funció torna un apuntador al nou inici de la llista que permet reinicialitzar correctament la variable `start`.

Al final, crida la funció `ReconstrueixApuntadorsprev` que reconstrueix els apuntadors endarrere.

La funció MergeSort_PasRecursiu

```
node_map * MergeSort_PasRecursiu ( node_map *start, unsigned long nelements,
                int (*compare) (const void *, const void *)) {
   register unsigned long nmid=(nelements >> 1), i;
   register node_map *middle; node_map *start_next;

   if (nelements < 2) return start;
   for (i=1, middle=start; i < nmid; i++, middle = middle->seg);
   start_next=middle->seg; middle->seg=NULL;
   return BarrejaDuesLlistesOrdenades(
              MergeSort_PasRecursiu(start, nmid, compare),
              MergeSort_PasRecursiu(start_next, nelements-nmid, compare),
              compare );
}
```

nmid = floor(nelements/2).

Per a usar una funció de comparació.

En aquest cas no hi ha res per ordenar. A la resta de la funció nelements >= 2.

Separació en dues llistes

Aquest és el pas recursiu. Es divideix la llista en dues meitats (que comencen a start i start_next) i es fa que les ordeni la mateixa funció MergeSort_PasRecursiu. Aquesta funció torna apuntadors als nous inicis de les llistes ordenades, que s'ajunten ordenadament mitjançant la funció BarrejaDuesLlistesOrdenades (observem que també l'hi hem de passar la funció compare perquè pugui ordenar la unió de les dues llistes). BarrejaDuesLlistesOrdenades retorna un apuntador a la llista ordenada que s'obté com unió unió de les dues mitges llistes, que és el valor de retorn de MergeSort_PasRecursiu.

La separació en dues llistes

```
for (i=1, middle=start; i < nmid; i++, middle = middle->seg);
start_next=middle->seg; middle->seg=NULL;
```

Funcionament:

- `for (i=1, middle=start; i < nmid; i++, middle=middle->seg);`: Recorregut seqüencial estàndard endavant `nmid` passos. En acabar, `middle` apunta al node número `nmid`, que és el final de la primera meitat de la llista.
- `start_next=middle->seg;`: Defineix l'inici del segon bloc de la llista que, òbviament té `nelements-nmid` elements. Com que `middle` apunta al node final de de la primera meitat de la llista, l'inici de la segona meitat és `middle->seg`. Observem que la segona llista acaba correctament amb un `NULL` perquè així ho feia la llista inicial.
- `middle->seg=NULL;`: Posa `NULL` a l'apuntador `seg` del node final del primer bloc per a acabar correctament aquesta llista. Observem que les dues darreres instruccions no es poden permutar.

La funció BarrejaDuesLlistesOrdenades

```
node_map * BarrejaDuesLlistesOrdenades (
                node_map *a,
                node_map *b,
                int (*compare) (const void *, const void *)) {
  node_map *start; register node_map *darrer;

  if (compare (a, b) <= 0) { start = darrer = a; a=a->seg; }
  else { start = darrer = b; b=b->seg; }

  while (a && b) {
      if (compare (a, b) <= 0) { darrer->seg = a; a=a->seg; }
      else { darrer->seg = b; b=b->seg; }
      darrer = darrer->seg;
  }

  darrer->seg = (a) ? a : b;

  return start;
}
```

Funcionament de `BarrejaDuesLlistesOrdenades`

- **Ordenació dels dos primers elements: Inicialització**

```
if (compare (a, b) <= 0) { start = darrer = a; a=a->seg; }
else { start = darrer = b; b=b->seg; }
```

Fem separadament la primera comparació per a inicialitzar correctament l'apuntador `start` a l'inici de la llista ordenada.

Si `compare (a, b) <= 0` tenim que `*a` va abans que `*b`.
Llavors, `start = darrer = a;` marca `*a` com inici de la llista `start` i al mateix temps com a node darrer de la nova llista ordenada.

Seguidament actualitzem `a=a->seg;` que fa que `a` apunti al següent node de la primera llista, que és el que haurem de comparar a continuació.

Si `compare (a, b) > 0` (`else`) fem anàlogament amb `*b`, el primer node la la llista `b`.

Funcionament de `BarrejaDuesLlistesOrdenades` (cont.)

- **Ordenació de la resta mentre hi ha elements a les dues llistes**

```
while (a && b) {
    if (compare (a, b) <= 0) { darrer->seg = a; a=a->seg; }
    else { darrer->seg = b; b=b->seg; }
    darrer = darrer->seg;
}
```

`while (a && b)` (és a dir, mentre hi ha elements a les dues llistes) fem les tres accions següents, com al pas inicial:

- Si `compare (a, b) <= 0` tenim que `*a` va abans que `*b` a la llista ordenada. Llavors, `darrer->seg = a;` marca `*a` com node següent a `darrer` (de la llista ordenada).

  Seguidament actualitzem `a=a->seg;` que fa que `a` apunti al següent node de la primera llista, que és el que haurem de comparar a continuació.
- Si `compare (a, b) > 0` (`else`) fem anàlogament amb `b` en comptes d'`a`.
- Finalment, `darrer = darrer->seg;` defineix el node que acabem d'afegir a la llista ordenada com a `darrer`.

Funcionament de `BarrejaDuesLlistesOrdenades` (cont.)?

- **S'afegeix la resta al final**
  `darrer->seg = (a) ? a : b;`

Aquesta és la instrucció que s'executa al sortir del bucle quan `a && b` és fals.

Tenim `a=NULL` o bé `b=NULL` (no pot passar que simultàniament `a=b=NULL` ja que els elements de la llista els col·loquem d'un en un). Dit d'una altra manera: o bé hem col·locat a la llista ordenada tots els elements de la llista `a` (`a=NULL`) i ens queden elements de la llista `b` o bé hem col·locat a la llista ordenada tots els elements de la llista `b` i ens queden elements de la llista `a`.

La instrucció `darrer->seg = (a) ? a : b;` defineix

$$\texttt{darrer->seg} = \begin{cases} \texttt{a} & \text{si } \texttt{a != NULL i} \\ \texttt{b} & \text{si } \texttt{a == NULL.} \end{cases}$$

És a a dir, afegeix al final de la llista ordenada el tros de la llista que queda.

Funcionament de `BarrejaDuesLlistesOrdenades` (cont.)?

- **S'afegeix la resta al final**

```
darrer->seg = (a) ? a : b;
```

Notem que:

- Com que cada una de les dues llistes inicials era ordenada, el tros que afegim al final de la nova llista és ordenat.
- A més, el fet que els elements que afegim en aquest pas no s'hagin afegit abans diu que tots ells són més grans que els que ja hem posat a la llista. Per tant la nova llista queda correctament ordenada.
- Aquest afegit dóna una llista correctament acabada amb un `NULL` ja que cada una de les dues llistes complia aquesta condició.

- `return start;`: Doncs això. Retorna l'apuntador a l'inici de la nova llista ordenada.

Aquest algorisme és clarament més lent que els anteriors (veure la comparativa de temps a continuació) i és més difícil i delicat de programació que el recursiu.

El gran avantatge d'aquesta opció és que pràcticament no fa servir memòria addicional (a la usada per la llista). Amb grans quantitats de dades això pot representar la diferencia entre poder o no poder reordenar les dades.

Com abans, la implementació de l'algorisme (`MergeSort`) separa la tasca de barrejar dues semi-llistes ordenades en una sola llista ordenada de mida la suma de les dues inicials: (`BarrejaDuesLlistesOrdenadesConsecutives`).

Desgraciadament ara tot és una mica més complicat que abans.

La funció MergeSort

```
void MergeSort (map *list, int (*compare) (const void *, const void *)) {
  register unsigned long nelements = 2UL, blockSize, NB, i;
  register node_map *listseg, *darrer, **Plistseg;

  if (!list || !list->start || !list->start->seg) return;

  darrer = list->start->seg; listseg = darrer->seg;
  if (compare (list->start, darrer) > 0) {
      darrer->seg = list->start; list->start = darrer; darrer = darrer->seg;
  }
  while (listseg && listseg->seg) { nelements += 2;
    if (compare (listseg, listseg->seg) <= 0) {
            darrer->seg = listseg; darrer = listseg->seg; listseg = darrer->seg;
    } else {
            darrer->seg = listseg->seg; darrer = listseg;
            listseg = listseg->seg->seg; darrer->seg->seg = darrer;
    }
  }
  if (listseg) { nelements++; darrer->seg = listseg; darrer = listseg; }
  darrer->seg = NULL;
```

Per a usar una funció de comparació.

En aquest cas no hi ha res per ordenar. La llista té com a màxim un element.

Pas de blockSize=1 a blockSize=2

La funció MergeSort no recursiu (cont.)

Pas de blockSize=2 endavant

```
  NB = nelements >> 1;
  for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1) { NB >>= 1;
    for (i = 0, Plistseg = &(list->start); i < NB; i++, Plistseg = &(darrer->seg))
      darrer = BarrejaDuesLlistesOrdenades (Plistseg, blockSize, blockSize, compare);

    unsigned long r = ((NB) ? (nelements & ((blockSize << 1) - 1UL)) : nelements);
    if( r > blockSize)
       BarrejaDuesLlistesOrdenades (Plistseg, blockSize, r - blockSize, compare);
  }

   ReconstrueixApuntadorsprev(list);
}
```

Procés final: reconstrueix els apuntadors `prev` endarrere per a regularitzar la llista. Explicat abans.

Funcionament de `MergeSort`: Pas de `blockSize=1` a `blockSize=2`

En aquest pas "sortim" d'`nelements` llistes d'un element (`blockSize=1`) que comencen a `list->start` i volem construir una llista formada per $\approx$ `nelements/2` llistes *ordenades* de *dos* elements (`blockSize=2`).

Al final del procés volem que `list->start` apunti al principi d'aquesta nova llista i l'apuntador `darrer` n'apunti al final.

Observem que no coneixem `nelements` (de fet per aquest pas no ens fa falta). Ho aprofitem per a a calcular `nelements` mentre fem aquesta iteració.

El procediment serà el seguent: `list->start` apuntarà a la part construïda de la llista de parells ordenats i l'apuntador `listseg` apuntarà a la part de la llista inicial que encara queda per ordenar per parelles.

A cada pas incorporarem els elements `listseg` i `listseg->seg` de manera ordenada al final de la llista ordenada per parells (és a dir, seran `darrer` i `darrer->seg`) i avançarem `listseg` dues posicions.

Anem a discutir en detall la implementació d'aquest pas.

Funcionament de `MergeSort`: Pas de `blockSize=1` a `blockSize=2`

- **Inicialització de la llista i ordenació dels dos primers elements**

```
darrer = list->start->seg; listseg = darrer->seg;
if (compare (list->start, darrer) > 0) {
    darrer->seg = list->start; list->start = darrer; darrer = darrer->seg;
}
```

Notem que la inicialització `nelements = 2UL` és consistent amb aquest inici.

Comencem suposant que els dos primers elements de la llista (`list->start` i `list->start->seg`) ja estan ben ordenats.

Llavors `list->start` està ben definit, el principi de la llista ordenada està correctament format pels elements `list->start` i `list->start->seg` en l'ordre en que ja hi són, i `darrer = list->start->seg`.

Per altra banda, el principi de la part de la llista pendent d'ordenar per parelles és el tercer element de la llista inicial:

`listseg = darrer->seg = list->start->seg->seg`.

Tot això és correcte si `list->start` és més petit o igual que `list->start->seg` (es a dir, si `compare (list->start, darrer) <= 0`).

Funcionament de MergeSort: Pas de blockSize=1 a blockSize=2

- `darrer = list->start->seg; listseg = darrer->seg;`
  `if (compare (list->start, darrer) > 0) {`
  `darrer->seg = list->start; list->start = darrer; darrer = darrer->seg; }`

Si `compare (list->start, darrer) > 0` el que hem fet no val i hem d'inicialitzar la llista correctament. Observem que, malgrat això, `listseg` està ben definit i apunta correctament al tercer element de la llista.

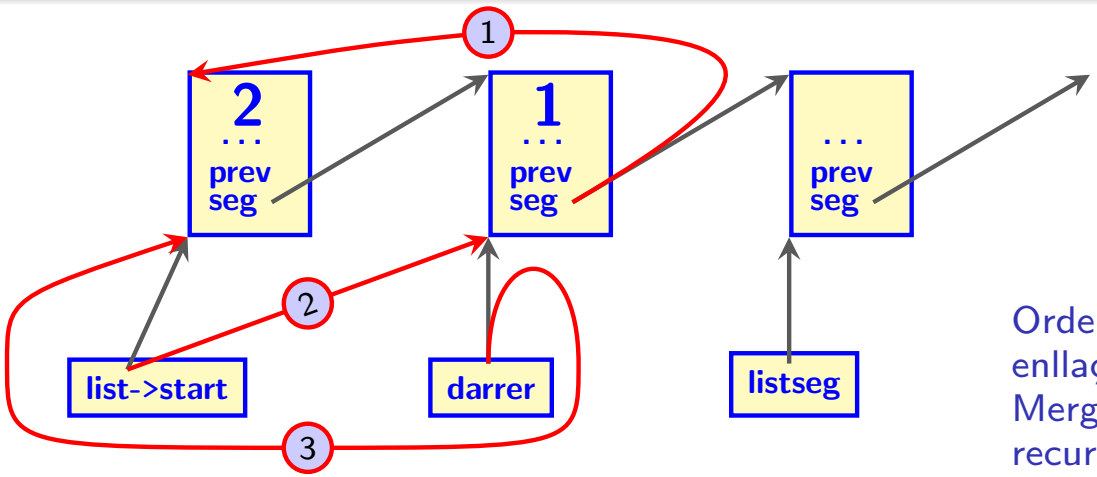
Hem de canviar l'inici de la llista ordenada. El node apuntat per `list->start` és el primer i ha de passar al final, i el node apuntat per `darrer` ha de passar a ser el primer:

1: `darrer->seg = list->start;` defineix que el seguent de `darrer` (és a dir el nou segon element donat que `darrer` ara apunta al - nou - primer element de la llista) és el que *abans* era el primer element de la llista. Òbviament aquesta darrera assignació cal fer-la abans de modificar `list->start`.

2: `list->start = darrer;` defineix que el segon node serà el primer de la nova llista i

3: Finalment cal fer que `darrer`, que ara a punta a l'inici de la llista, apunti al final de la llista. Això ho aconsegueix `darrer = darrer->seg;`

Això acaba la inicialització d'aquest pas de l'algorisme (de `blockSize=1` a `blockSize=2`).



Ordenació de llistes enllaçades dobles: Merge Sort no recursiu 6/27

Funcionament de `MergeSort`: Pas de `blockSize=1` a `blockSize=2`

- **Creació de la resta de la llista ordenada per parelles**

```
while (listseg && listseg->seg) { nelements += 2;
  if (compare (listseg, listseg->seg) <= 0) {
         darrer->seg = listseg; darrer = listseg->seg; listseg = darrer->seg;
  } else { darrer->seg = listseg->seg; darrer = listseg;
           listseg = listseg->seg->seg; darrer->seg->seg = darrer; }
}
```

Aquest és el bucle principal d'aquest pas.
Itera mentre `listseg != NULL` i `listseg->seg != NULL`.

Això vol dir que la part de la llista pendent d'ordenar té com a mínim dos elements, que afegirem ordenadament a la llista ordenada per parelles. En aquest cas, cal incrementar la comptabilitat del nombre d'elements en 2: `nelements += 2;`

Al processament iteratiu del contingut del bucle cal separar dos cassos:

Funcionament de MergeSort: Pas de blockSize=1 a blockSize=2

- **Creació de la resta de la llista ordenada per parelles: primer cas**

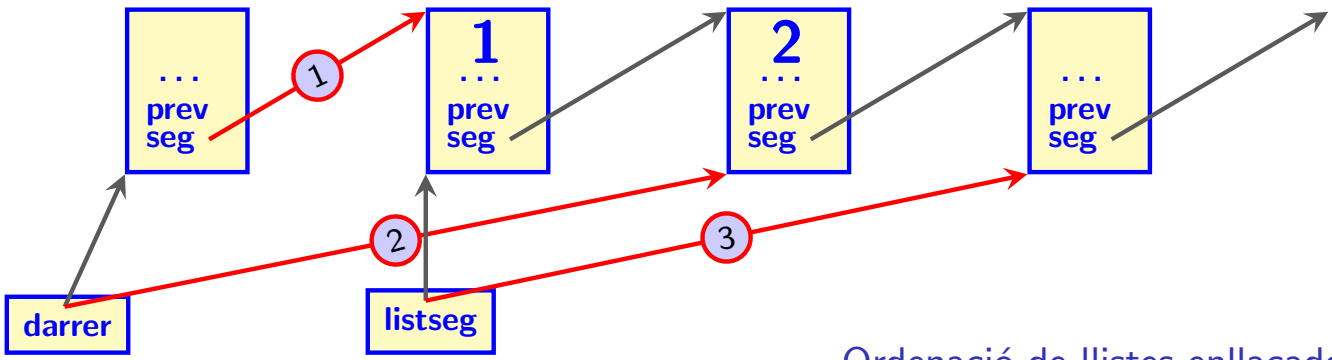
```
while (listseg && listseg->seg) { nelements += 2;
  if (compare (listseg, listseg->seg) <= 0) {
        darrer->seg = listseg; darrer = listseg->seg; listseg = darrer->seg;
  }
```

(és a dir, si `listseg` és més petit o igual que `listseg->seg`).
Notem que `listseg` i `listseg->seg` ja són en l'ordre correcte i no cal modificar `listseg->seg`.

1: `darrer->seg = listseg;` posa `listseg` a continuació del darrer element de la llista.

2: `darrer = listseg->seg;` defineix `darrer` com `listseg->seg`, que és el nou final de la llista.

3: `listseg = darrer->seg;` defineix el nou inici de la llista pendent com `darrer->seg = (listseg->seg)->seg`, que és l'element següent de `listseg->seg`.



Ordenació de llistes enllaçades dobles:
Merge Sort no recursiu 8/27

Funcionament de `MergeSort`: Pas de `blockSize=1` a `blockSize=2`

- **Creació de la resta de la llista ordenada per parelles: segon cas**

```
while (listseg && listseg->seg) { nelements += 2;
    ............
    else { darrer->seg = listseg->seg; darrer = listseg;
           listseg = listseg->seg->seg; darrer->seg->seg = darrer; }
```

(és a dir, si `compare (listseg, listseg->seg)> 0` o, equivalentment, `listseg->seg` és més petit o igual que `listseg`):

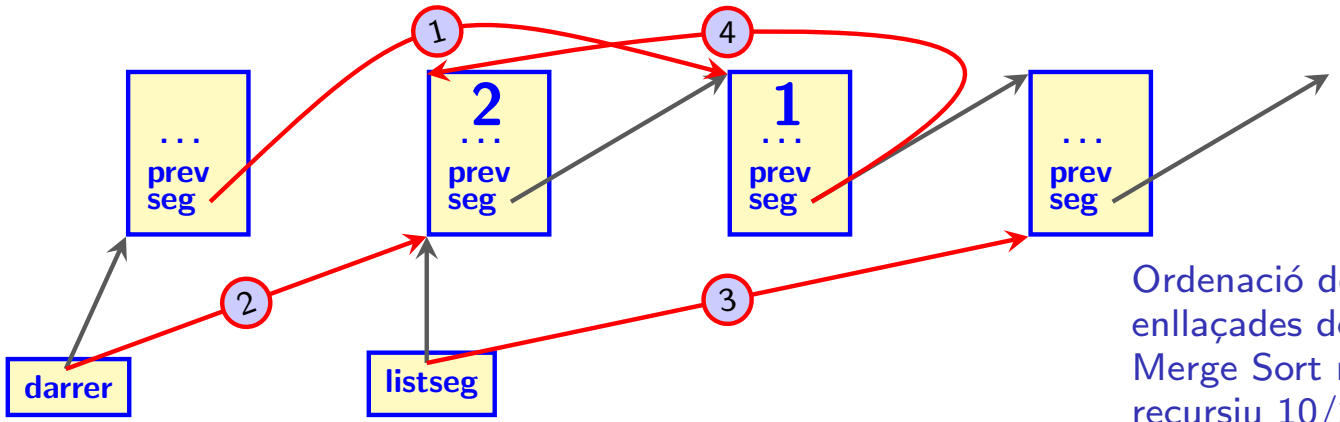
1: `darrer->seg = listseg->seg;`	posa `listseg->seg` a continuació del darrer element de la llista ja que `listseg->seg` va abans de `listseg`.
2: `darrer = listseg;`	defineix `darrer` com `listseg`, que ha de ser el nou final de la llista.
3: `listseg = listseg->seg->seg;`	defineix el nou inici de la llista pendent com `listseg->seg->seg`, que és l'element següent de `listseg->seg`.

Funcionament de MergeSort: Pas de blockSize=1 a blockSize=2

- **Creació de la resta de la llista ordenada per parelles: segon cas**

```
while (listseg && listseg->seg) { nelements += 2;
    ............
    else { darrer->seg = listseg->seg; darrer = listseg;
           listseg = listseg->seg->seg; darrer->seg->seg = darrer; }
```

4: `darrer->seg->seg = darrer;`
hauríem de posar el node apuntat per `listseg` com a següent del node apuntat per `listseg->seg`. És a dir, hauríem de fer `listseg->seg->seg = listseg;`. El problema és que `listseg` ja no apunta al primer dels dos nodes que estem ordenant (això ho hem modificat a la instrucció anterior). L'apuntador que apunta al primer dels dos nodes que estem ordenant ara és `darrer` (ho hem definit així dues assignacions més enrere). Així `listseg->seg->seg = listseg;` no és correcte però si que ho és `darrer->seg->seg = darrer;`, que produeix el resultat buscat.



Ordenació de llistes enllaçades dobles: Merge Sort no recursiu 10/27

Funcionament de `MergeSort`: Pas de `blockSize=1` a `blockSize=2`

- **Finalització de la llista ordenada per parelles**

```
if (listseg) { nelements++; darrer->seg = listseg; darrer = listseg; }
darrer->seg = NULL;
```

`if (listseg)` és veritat vol dir que `listseg != NULL` i, com que hem sortit del bucle, `listseg->seg == NULL`. Això vol dir que `listseg` apunta al darrer element de la llista i que solament queda aquest element per afegir a la llista ordenada per parelles (en aquest cas el darrer bloc serà una "parella" d'un únic element).

Per tant, `nelements` s'ha d'augmentar en una unitat: `nelements++;` i el nombre d'elements de la llista és senar (consistent amb el fet que la darrera "parella" de la llista serà d'un únic element).

Per afegir `listseg` al final de la llista: `darrer->seg = listseg;`.
Per re-definir `darrer` com el (nou) final de la llista: `darrer = listseg;`.

Finalment, `darrer->seg = NULL;` acaba la llista correctament amb un `NULL` com es requereix.

Funcionament de MergeSort: Bucle general

```
NB = nelements >> 1;
for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1) { NB >>= 1;
  for (i = 0, Plistseg = &(list->start); i < NB; i++, Plistseg = &(darrer->seg))
       darrer = BarrejaDuesLlistesOrdenades (Plistseg, blockSize, blockSize, compare);

  unsigned long r = ((NB) ? (nelements & ((blockSize << 1) - 1UL)) : nelements);
  if(r > blockSize) BarrejaDuesLlistesOrdenades(Plistseg, blockSize, r - blockSize, compare);
}
```

`for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1)`
Aquest és el bucle principal de la ordenació, que implementa la ordenació iterativa. A cada iteració tenim blocs de mida `blockSize` (excepte el darrer que pot tenir mida més curta), que han estat ordenats abans i els aparellem per a obtenir blocs ordenats de mida `2*blockSize` (excepte el darrer que pot tenir mida més curta).

Observem que, degut a que ja hem fet el pas de `blockSize=1` a `blockSize=2`, podem començar aquest bucle amb `blockSize = 2UL` per passar a `blockSize=4`.

Per altra banda, obtindrem tota la llista ordenada quan `blockSize` $<$ `nelements` $\leq$ `2*blockSize`. Llavors passem d'un bloc ordenat de mida `blockSize` i un bloc ordenat de mida més petit o igual que `blockSize`, a un únic bloc ordenat de mida `nelements`, que és tota la llista. Òbviament aquest és el final de l'algorisme. Això explica la condició `nelements > blockSize` del bucle: iterarem mentre aquesta condició es compleixi i acabarem la primera vegada que `nelements` $\geq$ `2*blockSize` donat que, llavors, ja tindrem tota la llista ordenada.

Ordenació de llistes enllaçades dobles: Merge Sort no recursiu 12/27

Funcionament de `MergeSort`: Bucle general

- `NB = nelements >> 1;`
  `for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1) { NB >>= 1;`
  - `blockSize <<= 1`: és l'actualització del bucle: `blockSize` $\leftarrow$ `2*blockSize`.
    La manera més fàcil de multiplicar un enter sense signe per 2 és desplaçant els seus bits un lloc a l'esquerra: `blockSize <<= 1` $\Longleftrightarrow$ `blockSize = blockSize << 1`.

Per a poder explicar les instruccions

- `NB = nelements >> 1;`
- `NB >>= 1;`

ens cal definir `NB` i estudiar una mica la seva aritmètica.

Definició

Definim `NB` com el nombre de blocs sencers de mida `2*blockSize` en que es por dividir la llista.

Òbviament aquest nombre es pot calcular dividint `nelements` per `2*blockSize`:

$$\texttt{NB} = \texttt{floor(nelements/(2*blockSize))} = \texttt{nelements/(2*blockSize)},$$

(on la darrera igualtat solament és correcte si fem aritmètica entera).

Notem que, llavors,

$$\texttt{nelements} = \texttt{NB*2*blockSize} + \texttt{r} \quad \text{amb} \quad 0 \leq \texttt{r} < \texttt{2*blockSize},$$

i la llista es divideix en `NB` blocs de mida `2*blockSize` i un bloc addicional de mida $0 \leq \texttt{r} \leq \texttt{2*blockSize}$.

Funcionament de `MergeSort`: Bucle general

- `NB = nelements >> 1;`
  `for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1) { NB >>= 1;`

Observació

Siguin n i k enters no negatius. Llavors, $\lfloor \frac{n}{2k} \rfloor = \lfloor \lfloor \frac{n}{k} \rfloor / 2 \rfloor$ (on $\lfloor \cdot \rfloor$ denota la part entera — `floor(·)`).

La conseqüència d'aquest fet és que quan `blockSize` $\rightarrow$ `2*blockSize`, tenim que

`NB` $\rightarrow$ `floor(NB/2)` $=$ `NB/2` $=$ `NB >> 1`

(on les igualtats solament són correctes amb aritmètica entera).

Demostració: Denotem $p := \lfloor \frac{n}{k} \rfloor$ i $q := \lfloor \lfloor \frac{n}{k} \rfloor / 2 \rfloor = \lfloor p/2 \rfloor$. Hem de veure que $q = \lfloor \frac{n}{2k} \rfloor$.
Notem que $p := \lfloor \frac{n}{k} \rfloor$ és equivalent a $n = kp + u$ amb $0 \leq u \leq k-1$ i $q := \lfloor p/2 \rfloor$ és equivalent a $p = 2q + v$ amb $0 \leq v \leq 1$.
Per tant, $n = kp + u = k(2q+v) + u = (2k)q + kv + u$ amb $0 \leq kv + u \leq k + k - 1 = 2k - 1$.
Llavors, $\frac{n}{2k} = q + \frac{kv+u}{2k} < q + 1$, que és equivalent a $q = \lfloor \frac{n}{2k} \rfloor$. □

- `NB >>= 1;`: A cada iteració del bucle fem `blockSize` $\rightarrow$ `2*blockSize`. Per tant, per la observació anterior, `NB` $\rightarrow$ `NB >> 1` $\Longleftrightarrow$ `NB >>= 1;`.
- `NB = nelements >> 1;`: Quan comencem el bucle passem de `blockSize=1` a `blockSize=2` (inicialització). Donat que la instrucció `NB >>= 1;` dins del bucle ens passarà del valor de `NB` corresponent a `blockSize=1` al valor corresponent a `blockSize=2` hem d'inicialitzar `NB` d'acord amb el valor `blockSize=1`:

  `NB` $=$ `floor(nelements/2)` $=$ `nelements/2` $=$ `nelements >> 1`

  (on les igualtats solament són correctes amb aritmètica entera).

Funcionament de MergeSort: Bucle general

- **Processat i ordenació dels NB blocs de mida 2*blockSize elements**

```
for (i = 0, Plistseg = &(list->start); i < NB; i++, Plistseg = &(darrer->seg))
    darrer = BarrejaDuesLlistesOrdenades (Plistseg, blockSize, blockSize, compare);
```

La funció BarrejaDuesLlistesOrdenadesConsecutives

```
node_map * BarrejaDuesLlistesOrdenadesConsecutives ( node_map ** start,
           register unsigned long startSize, register unsigned long endSize,
           int (*compare) (const void *, const void *))
```

accepta un apuntador doble `node_map ** start` (per permetre'n la modificació) amb `*start` apuntant a l'inici d'una parella de blocs de la llista de mida `startSize` el primer i `endSize` el segon; i realitza l'acció de barrejar aquests dos blocs de manera ordenada.

Per a fer això usa la funció `compare` per a comparar els elements i decidir la seva posició al bloc ordenat, en funció de les comparacions.

En acabar, la funció `BarrejaDuesLlistesOrdenadesConsecutives` retorna (com a valor de `return`) un apuntador al darrer node del bloc (de mida `startSize + endSize`) ordenat.
A més, l'apuntador `seg` d'aquest darrer node apunta al primer node del bloc següent a la llista (o `seg = NULL` si no hi ha cap més bloc a la llista).
`*start` es redefineix de manera que apunti al primer node del bloc que acabem d'ordenar. Això permet mantenir automàticament `list->start` quan processem el primer bloc.

Amb l'ajut d'aquesta funció el bucle

```
for (i = 0, Plistseg = &(list->start); i < NB; i++, Plistseg = &(darrer->seg))
```

(en particular, `for (i = 0; i < NB; i++)`)

processa els `NB` blocs inicials complets de mida `2*blockSize` ja que `BarrejaDuesLlistesOrdenadesConsecutives` es crida amb `startSize = blockSize` i `endSize = blockSize`.

Funcionament de `MergeSort`: Bucle general

- **Processat i ordenació dels `NB` blocs de mida `2*blockSize` elements**

```
for (i = 0, Plistseg = &(list->start); i < NB; i++, Plistseg = &(darrer->seg))
     darrer = BarrejaDuesLlistesOrdenades (Plistseg, blockSize, blockSize, compare);
```

El mecanisme d'iteració efectiva d'un bloc de mida `2*blockSize` al següent s'implementa mitjançant 3 elements (recordem que `Plistseg` s'ha declarat `register node_map **Plistseg`):

- `darrer = BarrejaDuesLlistesOrdenades (Plistseg, blockSize, blockSize,...`: Aquesta és la instrucció base del bucle. Barreja de manera ordenada els dos blocs de mida `blockSize` que comencen al node apuntat per `*Plistseg`, i retorna un apuntador al darrer node del bloc processat. L'adreça d'aquest node es guarda a `darrer`.
- `Plistseg = &(list->start)`: Això fixa `*Plistseg = list->start` (és a dir, `*Plistseg` apunta al principi de tota la llista) a la primera iteració del bucle. És a dir, el primer bloc que processa el bucle comença al principi de la llista. A més, al final del procés del primer bloc, `list->start = *Plistseg` apunta al nou inici de la llista (mantenint automàticament l'apuntador `list->start`).
- `Plistseg = &(darrer->seg)`: Aquesta és una de les instruccions d'actualització per totes les iteracions (exclosa la primera). Recordem que `darrer->seg` apunta al primer node de la llista després del bloc que hem processat a la iteració anterior.
  Per tant, `*Plistseg = darrer->seg` fixa correctament l'inici del nou bloc que ha de processar `BarrejaDuesLlistesOrdenadesConsecutives(Plistseg,...`.

Funcionament de MergeSort: Pas de blockSize >= 2 a 2*blockSize

- **Processat del darrer bloc (de menys de 2*blockSize elements) si existeix**

```
for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1) { NB >>= 1;
            ..............
  unsigned long r = ((NB) ? (nelements & ((blockSize << 1) - 1UL)) : nelements);
  if(r > blockSize) BarrejaDuesLlistesOrdenades(Plistseg, blockSize, r - blockSize, compare);
}
```

La fórmula per a calcular r = (NB)?(nelements & ((blockSize<<1)-1UL)):nelements

Com s'ha dit a l'estudi aritmètic de la variable NB, el darrer bloc de cada iteració del bucle

`for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1)`

té mida

`r = nelements - NB*2*blockSize = nelements % (2*blockSize)`.

Aquesta fórmula per a calcular r té dos problemes:

1. Si nelements és gran i som a la darrera iteració del bucle (per exemple quan 2*blockSize $>$ nelements $>$ blockSize $\geq$ ULONG_MAX/2 — recordem que blockSize s'ha declarat unsigned long), tindrem un overflow al calcular el número 2*blockSize $>$ ULONG_MAX. Per a evitar això notem que 2*blockSize $>$ nelements
   és equivalent a NB=0 i r=nelements. Llavors la fórmula anterior es pot substituir per l'*if aritmètic*
   `r = ((NB) ? (nelements % (2*blockSize)) :  nelements);`
   que, quan NB = 0, retorna directament nelements sense fer aritmètica.

   Observem que NB > 0 és equivalent a 2*blockSize $\leq$ nelements i, el fet que nelements sigui un unsigned long vàlid ens assegura que podem calcular 2*blockSize.

Funcionament de `MergeSort`: Pas de `blockSize >= 2` a `2*blockSize`

- **Processat del darrer bloc (de menys de `2*blockSize` elements) si existeix**

```
for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1) { NB >>= 1;
     unsigned long r = ((NB) ? (nelements & ((blockSize << 1) - 1UL)) : nelements);
```

La fórmula per a calcular `r = (NB)?(nelements & ((blockSize<<1)-1UL)):nelements`

2. La instrucció `nelements % (2*blockSize)` té una aritmètica molt complicada ja que es calcula com: `nelements - floor(nelements / (2*blockSize)) * (2*blockSize)`.
   Aprofitant que `blockSize` és potència de dos podem calcular `nelements % (2*blockSize)` amb un nivell d'aritmètica minimal, indicat per valors grans de `nelements` i `blockSize`.
   Si n és un enter `unsigned` llavors $\lfloor \frac{n}{2^k} \rfloor 2^k$ = `((n >> k) << k)` té la mateixa expressió en binari que n excepte els bits més a la dreta del k, que són tots zero (comptant el primer bit — el de més a la dreta — com 0).
   Llavors $n \pmod{2^k} = n - \lfloor \frac{n}{2^k} \rfloor 2^k$ és l'enter que té la mateixa expressió en binari que n a la dreta del bit k i el bit k i els de la seva esquerra són zero.
   Per altra banda, si m és una màscara que en binari té 1's a la dreta del bit k i el bit k i els de la seva esquerra són zero, `n % 2^k` $= n \pmod{2^k} = n - \lfloor \frac{n}{2^k} \rfloor 2^k =$ `n & m`.
   El nombre enter que en binari té 1's a la dreta del bit k i el bit k i els de la seva esquerra són zero és $2^k - 1$ (és a dir, $m = 2^k - 1$).
   Donat que `blockSize` (i per tant, `2*blockSize`) és una potència de 2, resulta que
   `nelements % (2*blockSize) = nelements & ((blockSize << 1) - 1UL)`
   (recordem que `blockSize << 1 = 2*blockSize`).

Ajuntant les conclusions d'(1) i (2) tenim la fórmula per
`r = ((NB) ? (nelements & ((blockSize << 1) - 1UL)) : nelements)`

Funcionament de `MergeSort`: Pas de `blockSize >= 2` a `2*blockSize`

- **Processat del darrer bloc (de menys de `2*blockSize` elements) si existeix**

```
for (blockSize = 2UL; nelements > blockSize; blockSize <<= 1) { NB >>= 1;
                .............
  unsigned long r = ((NB) ? (nelements & ((blockSize << 1) - 1UL)) : nelements);
  if(r > blockSize) BarrejaDuesLlistesOrdenades(Plistseg, blockSize, r - blockSize, compare);
}
```

  - `if(r > blockSize)`: En aquest cas queda per processar un bloc de mida `blocSize` i un de mida `r - blocSize`, que cal ajuntar ordenant-los. Aquesta feina la fa la instrucció `BarrejaDuesLlistesOrdenades(Plistseg, blockSize, r - blockSize, compare);` que dóna una llista correctament acabada amb un `NULL` ja que no hi ha cap node pendent de processar a continuació. Per aquest motiu, a diferència del que hem fet abans, no cal guardar l'apuntador `darrer` al darrer node de la llista.
  - `r <= blockSize`: En aquest cas ens queda un únic bloc ja ordenat, de mida $0 \leq$ `r`. Tal com hem dit abans, la funció `darrer = BarrejaDuesLlistesOrdenadesConsecutives` fa que `darrer` apunti al darrer node del darrer bloc processat i `darrer->seg` apunta al primer node del bloc següent a la llista (o `darrer->seg = NULL` si `r=0`). Per tant, el romanent d'`r` nodes ordenats ja està afegit a la llista i la llista acaba correctament amb un `NULL` heretat de la iteració anterior.
    Resumint, en aquest cas no cal fer res més.

La funció BarrejaDuesLlistesOrdenadesConsecutives

```
node_map * BarrejaDuesLlistesOrdenadesConsecutives ( node_map ** start,
             register unsigned long startSize, register unsigned long endSize,
             int (*compare) (const void *, const void *)) {
  register node_map *startRest = *start, *endRest, *darrer;
  register unsigned long i;

  for (i = 0UL, endRest = *start; i < startSize; i++, endRest = endRest->seg);

  if (compare (startRest, endRest) <= 0) {
          darrer = startRest; startRest = startRest->seg; --startSize;
  } else { darrer = *start = endRest; endRest = endRest->seg; --endSize; }
  while (startSize && endSize) {
     if (compare (startRest, endRest) <= 0) {
              darrer->seg = startRest; startRest = startRest->seg; --startSize;
     } else { darrer->seg = endRest; endRest = endRest->seg; --endSize; }
     darrer = darrer->seg;
  }
  if (startSize) {
       for (darrer->seg = startRest; startSize; startSize--, darrer=darrer->seg);
       darrer->seg = endRest;
  } else for (darrer->seg = endRest; endSize; endSize--, darrer=darrer->seg);
  return darrer;
}
```

Funcionament de BarrejaDuesLlistesOrdenadesConsecutives

Aquesta funció és similar a `BarrejaDuesLlistesOrdenades` a nivell de procés, amb alguns afegits degut al context on s'usarà i a que en lloc de detectar els finals dels blocs amb `NULL`'s ho fem comptant elements.

Recordem que la funció accepta un apuntador doble `node_map ** start` (per permetre'n la modificació) amb `*start` apuntant a l'inici d'una parella de blocs de la llista de mida `startSize` el primer i `endSize` el segon; i realitza l'acció de barrejar aquests dos blocs de manera ordenada.

Per a fer això usa la funció `compare` per a comparar els elements i decidir la seva posició al bloc ordenat, en funció de les comparacions.

La funció acaba amb `return darrer;` que retorna l'apuntador al darrer node del bloc (de mida `startSize + endSize`) ordenat. A més, la funció implementa que l'apuntador `darrer->seg` (d'aquest darrer node) apunti al primer node del bloc següent a la llista (o `darrer->seg = NULL` si no hi ha cap més bloc a la llista). Això s'usa a la funció que crida `BarrejaDuesLlistesOrdenadesConsecutives` per a l'aplicació reiterativa d'aquesta funció.

`*start` es redefineix de manera que apunti al primer node del bloc que acabem d'ordenar.

A continuació discutim la funció per parts.

`startRest` és un apuntador ràpid (`register`) que usarem per a apuntar a la part del bloc inicial que encara falta per col·locar a la unió ordenada dels dos blocs. Per això s'inicialitza `startRest = *start` a principi del primer bloc.

L'apuntador `endRest` juga el mateix paper respecte del segon bloc. Notem que `endRest` no es pot inicialitzar. L'hem de posicionar amb una cerca seqüencial estàndard a partir del principi del primer bloc `*start`.

Funcionament de BarrejaDuesLlistesOrdenadesConsecutives

- **Localitzem l'inici del segon bloc**

```
for (i = 0UL, endRest = *start; i < startSize; i++, endRest = endRest->seg);
```

Aquesta part del codi implementa una cerca seqüencial estàndard endavant per a posicionar l'apuntador `endRest` a principi del segon bloc.

La part del bucle:

```
for (i = 0UL; i < startSize; i++);
```

realitza `startSize` iteracions que consisteixen en `endRest = endRest->seg` (saltar al node següent) a partir de l'inici del bloc actual (de mida `startSize + endSize`): `endRest = *start`.
El fet d'utilitzar l'apuntador `endRest` per avançar per la llista fa que, en acabar el bucle, aquest apuntador estigui correctament inicialitzat a l'inici del segon bloc.

Funcionament de BarrejaDuesLlistesOrdenadesConsecutives

- **Ordenació dels dos primers elements: Inicialització**

```
if (compare (startRest, endRest) <= 0) {
        darrer = startRest; startRest = startRest->seg; --startSize;
} else { darrer = *start = endRest; endRest = endRest->seg; --endSize; }
```

Com a la funció `BarrejaDuesLlistesOrdenades`, fem separadament la primera comparació per a inicialitzar correctament l'apuntador `*start` a l'inici del bloc ordenat.

Si `compare (startRest, endRest) <= 0` tenim que `*startRest` va abans que `*endRest`. Llavors, `*start` (que coincideix amb `startRest`) és correcte. `darrer = startRest;` marca `*startRest` com a actual fi de bloc ordenat.

Seguidament actualitzem `startRest=startRest->seg;` que fa que `startRest` apunti al següent node del primer bloc, que és el que haurem de comparar a continuació. A més, decrementem `--startSize;` que és la variable que ens diu quants elements queden al bloc inicial pendents de passar al bloc ordenat.

Si `compare (startRest, endRest) > 0` (`else`) fem anàlogament amb `*endRest`, el primer node la la llista `*endRest`. La única diferència és que, ara, l'inici del bloc ordenat és `endRest;` i hem de fer que `*start` hi apunti: `darrer = *start = endRest;`

Funcionament de BarrejaDuesLlistesOrdenadesConsecutives

- **Ordenació de la resta mentre hi ha elements als dos blocs**

```
while (startSize && endSize) {
   if (compare (startRest, endRest) <= 0) {
            darrer->seg = startRest; startRest = startRest->seg; --startSize;
   } else { darrer->seg = endRest; endRest = endRest->seg; --endSize; }
   darrer = darrer->seg;
}
```

`while (startSize && endSize)` (és a dir, mentre hi ha elements als dos blocs) fem les tres accions següents, com al pas inicial:

- Si `compare (startRest, endRest) <= 0` tenim que `*startRest` va abans que `*endRest` al bloc ordenat. Llavors, `darrer->seg = startRest;` marca `*startRest` com node següent a `darrer` (del bloc ordenat).

  Seguidament actualitzem `startRest=startRest->seg;` que fa que `startRest` apunti al següent node del primer bloc, que és el que haurem de comparar a continuació i decrementem `--startSize;` que és la variable que ens diu quants elements queden al bloc inicial pendents de passar al bloc ordenat.
- Si `compare (startRest, endRest) > 0` (`else`) fem anàlogament amb `endRest` en comptes d'`startRest`.
- Finalment, `darrer = darrer->seg;` defineix el node que acabem d'afegir al bloc ordenat com a `darrer`.

Funcionament de BarrejaDuesLlistesOrdenadesConsecutives

- **S'afegeix la resta al final**

```
if (startSize) {
    for (darrer->seg = startRest; startSize; startSize--, darrer=darrer->seg);
    darrer->seg = endRest;
} else for (darrer->seg = endRest; endSize; endSize--, darrer=darrer->seg);
```

Aquesta és la instrucció que s'executa al sortir del bucle quan `startSize && endSize` és fals.

Tenim `startSize=0` o bé `endSize=0` (no pot passar que simultàniament `startSize=endSize=0` ja que els elements del bloc els col·loquem d'un en un). Dit d'una altra manera: o bé hem col·locat al bloc ordenat tots els elements de la llista `startRest` (`startSize=0`) i ens queda col·locar els elements del bloc `endRest` o bé hem col·locat al bloc ordenat tots els elements del bloc `endRest` i ens queda col·locar els elements del bloc `startRest`.

Notem que:

- Com que cada un dels dos blocs inicials era ordenat, el tros que afegim al final del nou bloc és ordenat.
- A més, el fet que els elements que afegim en aquest pas no s'hagin afegit abans diu que tots ells són més grans que els que ja hem posat al bloc. Per tant el nou bloc unió dels dos inicials queda correctament ordenat.

El problema que tenim ara (a diferència de la funció `BarrejaDuesLlistesOrdenades`) és que hem de retornar un apuntador `darrer` al darrer node del bloc ordenat i, a més, `darrer->seg` ha d'apuntar al primer node del bloc següent a la llista (o `darrer->seg = NULL` si no hi ha cap més bloc a la llista). Això ens obliga a recórrer el bloc romanent fins al final.

Funcionament de BarrejaDuesLlistesOrdenadesConsecutives

- **S'afegeix la resta al final**

```
if (startSize) {
    for (darrer->seg = startRest; startSize; startSize--, darrer=darrer->seg);
    darrer->seg = endRest;
} else for (darrer->seg = endRest; endSize; endSize--, darrer=darrer->seg);
```

  - `if (startSize)`: En aquest cas `startSize > 0` i ens queda col·locar al final del bloc ordenat els elements del bloc `startRest`. El bucle
    `for (darrer->seg = startRest; startSize; startSize--, darrer=darrer->seg);`
    fa dues accions:
    - La inicialització `darrer->seg = startRest;` connecta la part ja ordenada del bloc amb el que queda del bloc inicial `startRest` posant aquesta part a continuació de la part ordenada.
    - Recórrer el que queda del bloc inicial `startRest` (`darrer=darrer->seg`) mentre hi quedin elements (això és controla amb la instrucció d'actualització `startSize--` que porta el control del nombre d'elements que queden i la instrucció de control `startSize` que determina que el bucle pari quan `startSize=0`). Observem que, en acabar el bucle, `darrer` apunta al darrer node del bloc inicial que, de fet, és el darrer node del bloc ordenat.

    Finalment, `darrer->seg = endRest;` fa que `darrer->seg` apunti al primer node del bloc següent a la llista (o `darrer->seg = NULL` si no hi ha cap més bloc a la llista.
    Notem que el primer node del bloc següent a la llista serà l'apuntador `endRest` una vegada aquest segon bloc s'hagi esgotat (`endSize=0`) i, si no hi ha cap més bloc a la llista, `endRest=NULL`.

Funcionament de BarrejaDuesLlistesOrdenadesConsecutives

- **S'afegeix la resta al final**

```
if (startSize) {
    for (darrer->seg = startRest; startSize; startSize--, darrer=darrer->seg);
    darrer->seg = endRest;
} else for (darrer->seg = endRest; endSize; endSize--, darrer=darrer->seg);
```

  - `if (startSize==0)` (`else`): En aquest cas ens queda col·locar al final del bloc ordenat els elements del bloc `endRest`. Això ho fem igual que abans amb el bucle
    `for (darrer->seg = endRest; endSize; endSize--, darrer=darrer->seg);`
    Observem que en aquest cas l'assignació `darrer->seg = endRest;` ja la realitza el bucle final.

- `return darrer;`: Doncs això. Retorna l'apuntador al final del nou bloc ordenat

Comparativa dels quatre procediments d'ordenació

La comparativa s'ha fet ordenant una llista enllaçada doble de 23.895.681 nodes respecte del camp `id` en l'ordre lexicogràfic (`strcmp`) invers en un ordinador amb un processador Intel i7-4770K @ 3.50GHz amb 32Gb de memòria RAM DDR3 @ 1867 MHz.

Funció	CPU (segs.)
`CreaIndex`	2.21
`OrdenaUsantIndex`	2.40
`MergeSortRecursiu`	2.99
`MergeSort`	5.85

Índex

1. Introducció
2. Implementació d'una cua senzilla de mida fixada
3. Implementació d'una cua de mida arbitrària amb llistes enllaçades
4. Exercicis
5. Aplicacions de les cues

Introducció

Una cua és una estructura *First In – First Out (FIFO)* on la primera dada en entrar és la primera en sortir:

- Un element es pot inserir en qualsevol moment a la cua, però només l'element que ha estat més temps a la cua pot ser retirat.
- Els elements s'insereixen a la part posterior (en cua) i es retiren per la part davantera.

Un exemple d'aquesta estructura són les cues de la vida real.

Les funcions fonamentals d'una cua són:

encua Insereix un objecte a la part posterior de la cua

desencua Treu l'objecte de la part frontal de la cua i el torna si la cua no està buida. En cas contrari torna un codi d'error.

Així mateix una cua pot tenir associades funcions auxiliars. Entre elles:

mida retorna el nombre d'objectes a la cua

CuaEsBuida retorna un valor booleà que indica si la cua està buida o no

Implementació d'una cua senzilla de mida fixada

Definicions bàsiques

```
typedef struct { int n, ll; } CuaElem;

#define QUEUESIZE 150
typedef struct {
   unsigned primer, darreradarrer;
   CuaElem Elements[QUEUESIZE];
} ContenidorCua;
```

`primer` és l'index del primer de la cua: `Elements[primer]` és el primer element de la cua.
`darreradarrer` és l'index següent al darrer de la cua: `Elements[darreradarrer-1]` és el darrer element de la cua.
Si `darreradarrer` $\leq$ `primer` la cua és buida.

Constructor

```
ContenidorCua *InicialitzaCua () { ContenidorCua * aux;
  if ((aux = (ContenidorCua *) malloc(sizeof (ContenidorCua))) == NULL) return NULL;
  aux->primer = aux->darreradarrer = 0; // Cua buida
  return (aux);
}
```

Destructor

```
void AlliberaCua( ContenidorCua *Cua ){ free(Cua); }
```

Implementació de encua

encua

```
int encua( int n, int ll, ContenidorCua *Cua ){
  if ( CuaEsPlena(Cua) ) return 0;

  if ( Cua->darreradarrer >= QUEUESIZE) { unsigned int lim = Cua->darreradarrer - Cua->primer;
      for (Cua->darreradarrer=0; Cua->darreradarrer < lim ; Cua->darreradarrer++)
          Cua->Elements[Cua->darreradarrer] = Cua->Elements[Cua->primer + Cua->darreradarrer] ;
      Cua->primer = 0;
  }

  Cua->Elements[Cua->darreradarrer].n = n;
  Cua->Elements[Cua->darreradarrer].ll = ll;
  Cua->darreradarrer++;
  return 1;
}
```

Aquí `!CuaEsPlena(Cua)` i `Cua->darreradarrer >= QUEUESIZE`. Això vol dir que l'espai lliure de la cua és tot al començament. En particular `Cua->primer > 0`. Cal reorganitzar la cua.
`lim` és el nombre d'elements que hi ha a la cua.
El bucle

```
for (Cua->darreradarrer=0 ;
     Cua->darreradarrer < lim
;
     Cua->darreradarrer++ )
```

trasllada els elements de la cua a l'inici i deixa el valor de `Cua->darreradarrer` igual a `lim` que és l'index del següent al darrer de la (nova) cua.
Ara la cua és tota a l'inici i el seu primer element té index 0. Per tant, cal fer `Cua->primer = 0;`

Exemple d'ús

```
if (!encua(5,7,Cua)) printf("Cua plena\n");
```

Funció auxiliar CuaEsPlena

```
int CuaEsPlena( ContenidorCua *Cua ){
    return (Cua->darreradarrer - Cua->primer >= QUEUESIZE) ;
}
```

Implementació de desencua

desencua

```
int n, ll;

int desencua( int *n, int *ll, ContenidorCua *Cua ){
  if ( CuaEsBuida(Cua) ) return 0;
  *n = Cua->Elements[Cua->primer].n;
  *ll = Cua->Elements[Cua->primer].ll;
  Cua->primer++;
  return 1;
}
```

Exemple d'ús

```
if (!desencua(&n,&ll,Cua)) printf("Cua buida\n");
```

Funció auxiliar CuaEsBuida

```
int CuaEsBuida( ContenidorCua *Cua ){
    return !(Cua->primer < Cua->darreradarrer) ;
}
```

Implementació d'una cua de mida arbitrària amb llistes enllaçades
Declaracions i funcions auxiliars

Declaració dels elements de la cua

```
typedef struct CuaElem {
   unsigned int h;
   struct CuaElem *seg;
} CuaElem;
```

El contenidor de la cua

```
typedef struct {
  CuaElem *start, *end;
  unsigned nel;
  unsigned long altes_dades;
} Cua;
```

Funcions senzilles de gestió de la cua

```
void IniCua (Cua *Q) {
     Q->start = Q->end = NULL;
     Q->nel = Q->altes_dades = 0U;
}
int CuaEsBuida( Cua Q ){ return ( Q.start == NULL ); }
int CuaEsBuida( Cua Q ){ return ( Q.nel == 0U ); }
CuaElem * top( Cua Q ) { return Q.start; }
```

Versió alternativa de la funció `CuaEsBuida`

Implementació d'una cua de mida arbitrària amb llistes enllaçades
Més sobre funcions auxiliars

Inicialització

```
Cua LaCua;
IniCua(&LaCua);
```

Perquè serveix la funció `top`: Per exemple per recuperar o modificar dades del primer de la llista

```
htotal += top(LaCua)->h;
top(LaCua)->h = 2.0*hnou + 4;
```

`AlliberaCua` (com la funció `BorraLlista` que ja hem vist)

```
void AlliberaCua( Cua *Q ){ CuaElem *aux;
   while(Q->start){
     aux = Q->start; Q->start = Q->start->seg; free (aux);
   }
   IniCua(Q);
}
```

Exercici

Programa una funció `ImprimeixCua` que imprimeixi seqüencialment tots els elements de la cua.

Implementació d'una cua de mida arbitrària amb llistes enllaçades: la funció encua

Per poder modificar l'inici de la cua

Inici:
* Crear l'estructura aux
* Omplir-la amb les dades: aux->h=h;
* Inicialitzar apuntador seg: aux->seg=NULL;

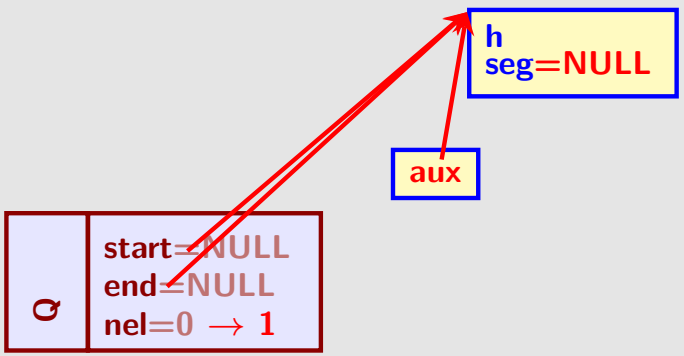
encua

```
CuaElem * encua( Cua *Q, unsigned int h){
  CuaElem *aux=(CuaElem *) malloc(sizeof(CuaElem));
  if( aux == NULL) return NULL;
  aux->h=h; aux->seg=NULL;

  if( Q->start ) Q->end->seg=aux;  else Q->start=aux;
  Q->end=aux; Q->nel++;
  return aux;
}
```

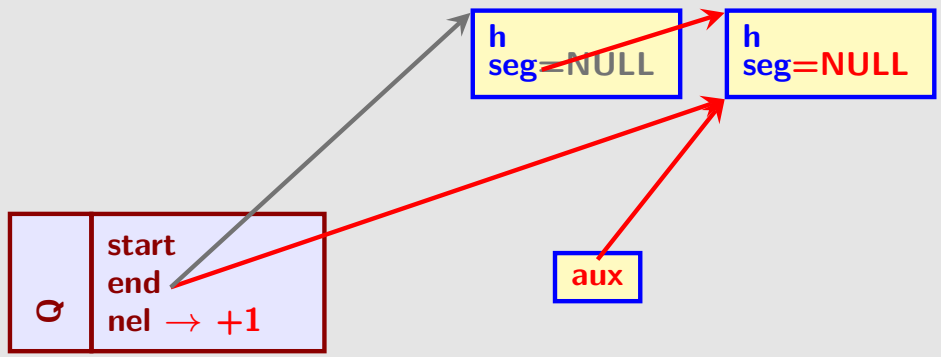
Exemple d'ús

```
CuaElem *last = encua(&LaCua, 10);
if (!last)){
   fprintf (stderr, "\nERROR: Cua overfull.\n");
   return 11;
}
last->altres_si_hi_es = 177;
```

Cas de cua buida



Cas general: Q->start ja estava inicialitzat



Implementació d'una cua de mida arbitrària amb llistes enllaçades: la funció desencua

Per poder modificar l'inici de la cua

desencua

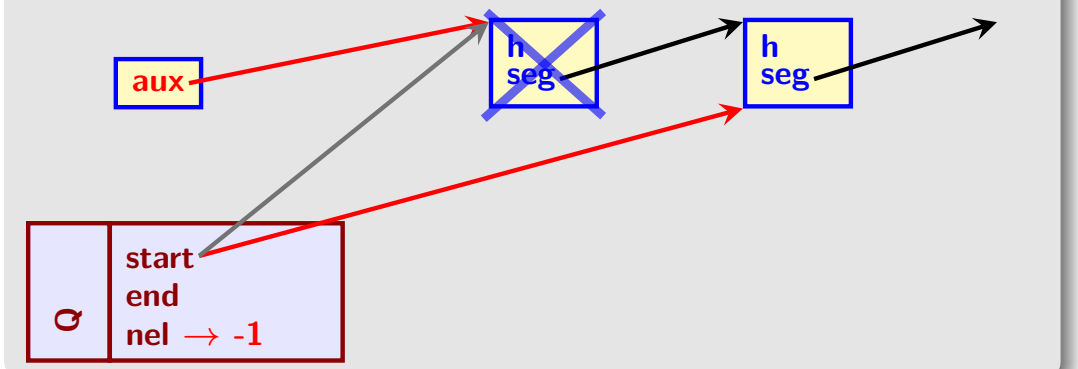
```
void desencua( Cua *Q ){
  if( CuaEsBuida(*Q) ) return;
  CuaElem *aux = Q->start;
  Q->start=aux->seg;
  Q->nel--;
  free(aux);
}
```

Quan buidem la cua (en treure'n el darrer element) reconstruïm l'assignació inicial `Q->start = NULL` (`Q->start = aux->seg = NULL`).

Exemple d'ús

```
desencua(&LaCua);
```

desencua



La funció encua ens permet fer una versió mes clara d'`AlliberaCua`

Encara que, segurament, aquesta versió no és més eficient.

La funció `AlliberaCua` més clara

```
void AlliberaCua( Cua *Q ){
   while( !CuaEsBuida(*Q) ) desencua(Q);
   IniCua(Q);
}
```

Aplicacions

- *Sistemes operatius*: gestió seqüencial de processos i distribució de tasques. Per exemple: cues d'impressió, distribució dels temps de CPU,...
- *Scheduling*: programació de tasques (per exemple l'ordre en que un viatjant de comerç fa les visites), distribució de tasques en diferents processadors (màquines – això es fa amb diverses cues; per exemple una per màquina). La distribució de temps de CPU és un cas particular d'això i els horaris d'una escola per aules (cada aula es considera una màquina) també.
- Procés seqüencial de dades en temps real.
- Simulació de processos que involucrin cues a fi de millorar-ne l'eficiència.

Índex

1. Introducció
2. Implementació d'un stack senzill de mida fixada
3. Implementació d'un stack de mida arbitrària amb llistes enllaçades
4. Exemple: l'Span del preu d'una acció de borsa
   - Algorisme simple — complexitat quadràtica
   - Algorisme amb complexitat lineal usant un stack
   - Algorisme amb complexitat lineal: implementació amb un stack de mida fixada
   - Algorisme amb complexitat lineal: implementació amb un stack de mida arbitrària amb una llista enllaçada
5. Exercici: Imprimir un fitxer amb l'ordre de línies invertit.

Introducció

Un stack és una col·lecció d'elements on només es pot accedir a un extrem anomenat la part superior de la pila.

L'operació de l'addició d'un element a la part superior de la pila s'anomena *push* i l'operació de treure l'element de la part superior de l'stack es diu *pop*.

El nom fa referència a una pila de safates de cafeteria. Els nous objectes s'afegeixen a la part superior mitjançant l'operació *push* i solament es poden eliminar treien l'element superior amb l'operació *pop*. Entre les funcions auxiliars associades a un stack podem tenir: *mida* que retorna el nombre d'objectes a la pila i *StackEsBuit* que retorna un valor booleà que indica si l'stack és buit o no.

Per raons òbvies, una pila també s'anomena *últim d'entrar — primer a sortir (LIFO)*.

Implementació d'un stack senzill de mida fixada

Definicions bàsiques

```
typedef struct { int n, ll; } StElem;

#define STACKSIZE 15
typedef struct {
   int nelements;
   StElem Elements[STACKSIZE];
} ContenidorStack;
```

Constructor

```
ContenidorStack *InicialitzaStack () {
  ContenidorStack * aux;
  aux = (ContenidorStack *)
        malloc(sizeof(ContenidorStack));
  if ( aux == NULL ) return NULL;
  aux->nelements = 0;
  return (aux);
}
```

Exemple d'ús

```
ContenidorStack *stackP = InicialitzaStack();
```

o alternativament i més senzill: declaració directa

```
// Inicialització incompleta
ContenidorStack stack = { 0 };
```

Destructor

```
void StackFree( ContenidorStack *Stack ){
  free(Stack);
}
```

Exemple d'ús

```
StackFree(stackP);
```

En el cas de declaració directa no es pot alliberar memòria.

Implementació de push i pop

push

```
int push( int n, int ll, ContenidorStack *Stack ){
  if ( StackEsPle(Stack) ) return 0;
  Stack->Elements[Stack->nelements].n = n;
  Stack->Elements[Stack->nelements].ll = ll;
  Stack->nelements++;
  return 1;
}
```

Exemple d'ús

```
if ( !push(5, 7, StackP) ) printf("Stack plè\n");
```

Funció auxiliar StackEsPle

```
int StackEsPle( ContenidorStack *Stack ){
    return (Stack->nelements == STACKSIZE) ;
}
```

pop

```
int pop( int *n, int *ll, ContenidorStack *Stack ){
  if ( StackEsBuit(Stack) ) return 0;
  Stack->nelements--;
  *n = Stack->Elements[Stack->nelements].n;
  *ll = Stack->Elements[Stack->nelements].ll;
  return 1;
}
```

Exemple d'ús

```
if ( !pop(&n,&ll,StackP) ) printf("Stack buit\n");
```

Funció auxiliar StackEsBuit

```
int StackEsBuit( ContenidorStack *Stack ){
    return (Stack->nelements == 0) ;
}
```

Implementació d'un stack de mida arbitrària amb llistes enllaçades
Declaracions i funcions auxiliars

Declaració dels elements del stack

```
typedef struct StElem {
   unsigned int h;
   struct StElem *lower;
} StElem;
```

El contenidor del stack

```
typedef struct {
  StElem *start;
  unsigned nel;
} Stack;
```

Funcions senzilles de gestió del stack

```
void IniStack (Stack *S) { S->start = NULL; S->nel = 0U; }
int StackEsBuit( Stack S ){ return ( S.start == NULL ); }
int StackEsBuit( Stack S ){ return ( S.nel == 0U ); }
StElem * top( Stack S ) { return S.start; }
```

Versió alternativa de la funció `StackEsBuit`

Implementació d'una stack de mida arbitrària amb llistes enllaçades
Més sobre funcions auxiliars

Inicialització

```
Stack Pila;
IniStack(&Pila);
```

Perquè serveix la funció `top`: Per exemple per recuperar o modificar dades del primer de la llista

```
htotal += top(Pila)->h;
top(Pila)->h = 2.0*hnou + 4;
```

`StackFree` (com la funció `AlliberaCua` que ja hem vist)

```
void StackFree( Stack *S ){ StElem *aux;
   while(S->start){
     aux = S->start; S->start = S->start->lower; free (aux);
   }
   IniStack(S);
}
```

Implementació d'una stack de mida arbitrària amb llistes enllaçades: la funció push

Per poder modificar l'inici del stack

Inici:
* Crear l'estructura aux
* Omplir-la amb les dades: aux->h=h;
* Inicialitzar apuntador lower: aux->lower=NULL;

push

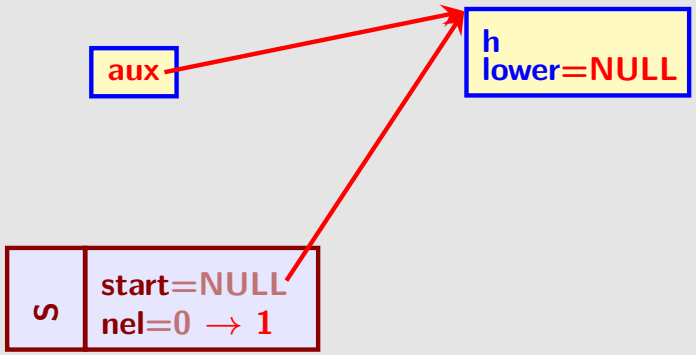
```
StElem * push( Stack *S, unsigned int h){
  StElem *aux=(StElem *) malloc(sizeof(StElem));
  if( aux == NULL) return NULL;
  aux->h=h;

  aux->lower=S->start;
  S->start=aux;
  S->nel++;
  return aux;
}
```

L'assignació inicial S->start = NULL, assegura que lower = NULL per l'element base de la pila (és a dir el primer element que ha entrat a la pila).

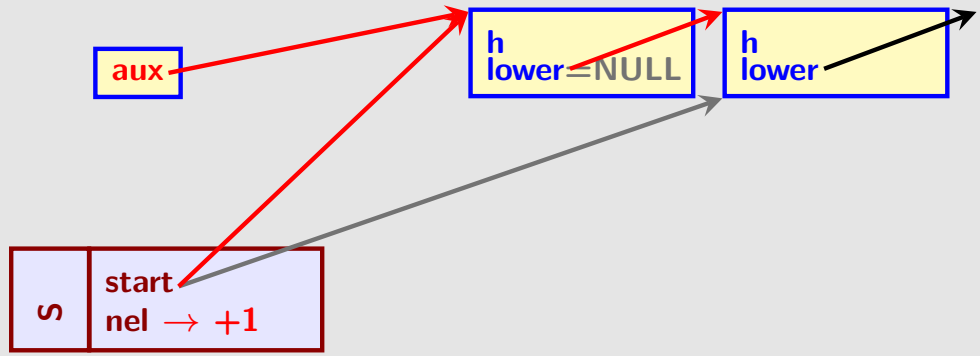
Exemple d'ús

```
StElem *last = push(&Pila, 10);
if (!last)){
   fprintf (stderr, "\nERROR: Stack overfull\n");
   return 11;
}
last->altres_si_hi_es = 177;
```

Cas de stack buida



Cas general: S->start ja estava inicialitzat



Implementació d'una stack de mida arbitrària amb llistes enllaçades: la funció pop

Per poder modificar l'inici del stack

pop

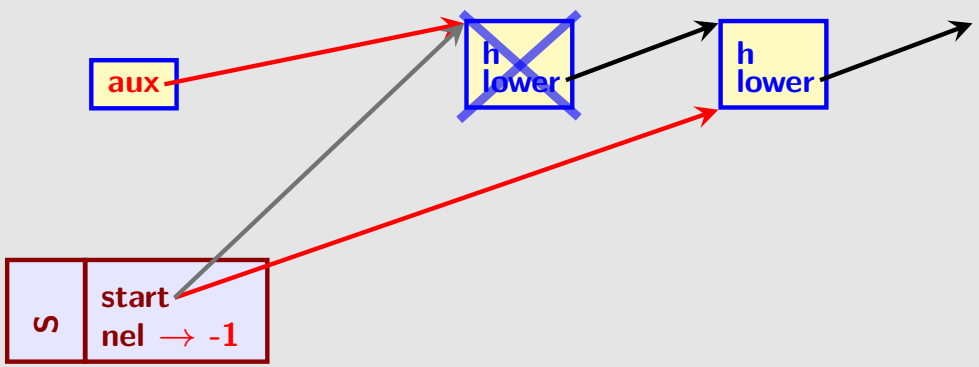
```
void pop( Stack *S ){
  if( StackEsBuit(*S) ) return;
  StElem *aux = S->start;
  S->start=aux->lower;
  S->nel--;
  free(aux);
}
```

Quan buidem la pila (en treure'n el darrer element) reconstruïm l'assignació inicial `S->start = NULL` (`S->start = aux->lower = NULL`).

Exemple d'ús

```
pop(&Pila);
```

pop



La funció StackFree més clara

```
void StackFree( Stack *S ){
   while( !StackEsBuit(*S) ) pop(S);
   IniStack(S);
}
```

Exemple: l'Span del preu d'una acció de borsa

L'*Span* del preu d'una acció de borsa en un dia donat es defineix com el nombre màxim de dies consecutius anteriors al dia donat, tals que el preu de l'acció en el dia corresponent és menor o igual que el preu en el dia donat.

Més precisament, si la sèrie de preus de l'acció en consideració és $P[0], P[1], \ldots, P[n-1]$,

$$\text{Span}(i) := \max\{k \in \mathbb{Z}^+ : k \leq i \quad \text{i} \quad P[j] \leq P[i] \quad \text{per } j = i-k, i-k+1, \ldots, i\}.$$

En particular,

- $\text{Span}(i) = 0 \Leftrightarrow P[i-1] > P[i]$ i
- $\text{Span}(i) = i \Leftrightarrow P[i] \geq P[j]$ per $j = 0, 1, \ldots, i$.

Exemple: l'Span del preu d'una acció de borsa
Algorisme simple — complexitat quadràtica

El càlcul efectiu és pot fer amb un algorisme simple que és el resultat d'aplicar directament la definició d'Span.
Aquest algorisme, però, té complexitat quadràtica.

Algorisme simple — complexitat quadràtica (ometem la lectura de les dades)

```
#define Ndades 4448

double P[Ndades]; /* La sèrie de borsa */
unsigned int S[Ndades] = { 0U }; /* Vector d'Spans. S[0] = 0 */
register unsigned int i, k;

for (i=1; i< Ndades; i++){ S[i] = i;
    for (k=1; k<= i; k++) if (P[i-k] > P[i]) { S[i] = k-1; break; }
}
```

Algorisme amb complexitat lineal. Usant un stack. Introducció

El càlcul de l'Span es pot simplificar molt si sabem el dia més proper abans de l'actual, denotat per $h(\cdot)$, tal que el preu de l'acció en aquest dia és més gran que el preu actual. Si no existeix tal dia es dona el valor -1 a h.

Més precisament:

Definició

$$h(i) = \begin{cases} -1 & \text{si } P[j] \leq P[i] \text{ per } j = 0, 1, \ldots, i \\ \max\{k \in \mathbb{Z}^+ : k \leq i \quad \text{i} \quad P[k] > P[i]\} & \text{en cas contrari} \end{cases}$$

Amb aquesta definició tenim,

$$\text{Span}(i) = i - 1 - h(i).$$

Per aplicar aquesta estratègia al càlcul d'Span(i) s'utilitza un stack.

Algorisme amb complexitat lineal usant un stack. Introducció

Definim l'*stack corresponent al dia i* com

$$i := h^0(i) > h(i) > h(h(i)) > \ldots > h^\ell(i) \geq 0 \quad \text{amb} \quad h(h^\ell(i)) = -1.$$

Nota: L'stack corresponent al dia i conté al menys l'element i (a dalt).
Per les definicions anteriors, $P[h^{j+1}(i)] > P[h^j(i)]$ per $j = 0, 1, \ldots, \ell - 1$ i

$$P[h^j(i)] \geq P[k] \text{ per } \begin{cases} j < \ell \text{ i } k = h^{j+1}(i) + 1, h^{j+1}(i) + 2, \ldots, h^j(i), \\ j = \ell \text{ i } k = 0, 1, \ldots, h^\ell(i). \end{cases}$$

En conseqüència, si $k \in \{0, 1, 2, \ldots, \ell + 1\}$ és tal que

$$\begin{cases} P[h^n(i-1)] \leq P[i] & \text{per } n = 0, 1, \ldots, k-1; \text{ i} \\ P[h^n(i-1)] > P[i] & \text{per } n = k, k+1, \ldots, \ell, \end{cases}$$

$$h^0(i) = i; \; h^j(i) = h^{k+j-1}(i-1) \text{ amb } j = 1, 2, \ldots, \ell - k + 1.$$

Nota: Si $k = \ell + 1$ l'stack corresponent al dia i solament conté $h^0(i) = i$.

Resumint:

L'stack pel dia i s'obté a partir de l'stack del dia $i - 1$ eliminant (`pop`) els elements superiors $h^n(i-1)$ de l'stack tals que $P[h^n(i-1)] \leq P[i]$ i afegint (`push`) i al damunt de la resta.

Motivació a usar el mètode de l'stack

L'algoritme directe té complexitat quadràtica Kn^2 respecte del nombre de dades n (veure el codi a la transparència 156/165) mentre que l'algorisme usant stacks té complexitat lineal $\widetilde{K}n$. Això implica una gran diferència, per n gran, com mostra la taula següent:

Nombre de dades	Alg. directe	Alg. amb stack
10^5	0"0.471'	0"0.057'
10^6	3"46.175'	0"0.589'
$2 \cdot 10^6$	18"40.350'	0"1.252'
$3 \cdot 10^6$	44"49.004'	0"1.884'
$4 \cdot 10^6$	83"44.855'	0"2.688'

Aquests resultats justifiquen amb escreix, per n gran, l'ús de l'algorisme amb stack malgrat que té una complexitat lògica superior a l'algorisme directe.

Nota

Les proves de temps de la taula anterior s'han fer amb un ordinador amb un processador Intel i7-4770K @ 3.50GHz amb 32Gb de memòria RAM DDR3 @ 1867 MHz.

Algorisme amb complexitat lineal
Implementació amb un stack de mida fixada

Declaracions i funcions base

```
#define Ndades 4448
#define STACKSIZE 1700
typedef struct {
   unsigned int nelements;
   unsigned int Elements[STACKSIZE]; /* No cal fer servir estructures pels elements de l'stack */
} ContenidorStack;

int StackEsBuit( ContenidorStack *Stack ){ return (Stack->nelements == 0); }
int StackEsPle( ContenidorStack *Stack ){ return (Stack->nelements == STACKSIZE); }

ContenidorStack *InicialitzaStack () { ContenidorStack * aux;
  if ((aux = (ContenidorStack *) malloc(sizeof (ContenidorStack))) == NULL) return NULL;
  aux->nelements = 0;
  return (aux);
}
void StackFree( ContenidorStack *Stack ){ free(Stack); }

int push( unsigned int h, ContenidorStack *Stack ){
  if ( StackEsPle(Stack) ) return 0;
  Stack->Elements[Stack->nelements] = h;
  Stack->nelements++;
  return 1;
}

/* pop dividit en dues operacions; top no comprova si stack es buit */
unsigned int top( ContenidorStack *Stack ){ return Stack->Elements[Stack->nelements-1]; }
void deletetop( ContenidorStack *Stack ){ if (Stack->nelements) Stack->nelements--; }
```

L'opció d'usar un stack de mida fixada, de fet, no és aconsellable.
Problema: La mida de l'stack depèn de les dades i no és previsible ni es pot triar d'antuvi de manera raonable.

Algorisme amb complexitat lineal
Implementació amb un stack de mida fixada

Implementació de l'algorisme

```
double P[Ndades]; /* La sèrie de borsa */
unsigned int S[Ndades] = { 0U }; /* Vector d'Spans. S[0] = 0 */
register unsigned int i;
ContenidorStack *Stack ;

Stack = InicialitzaStack ();
push(0U, Stack); /* Inicialització. Per i=0 l'stack solament conté 0 */
for (i=1; i < Ndades ; i++){
    while( !StackEsBuit(Stack) && P[i] >= P[top(Stack)] ) deletetop(Stack);
    if (StackEsBuit(Stack)) S[i] = i;
    else S[i] = i - 1 - top(Stack);
    if (!push(i, Stack)){
       fprintf (stderr, "\nERROR: Stack overfull. Impossible continuar...\n\n");
       return 11;
    }
}
StackFree( Stack );
```

Algorisme amb complexitat lineal
Implementació amb un stack de mida arbitrària amb una llista enllaçada

Declaracions i funcions base

```
#define Ndades 4448

/* L'stack s'inicialitza com StElem *Stack = NULL; */
typedef struct StElem { unsigned int h; struct StElem *lower; } StElem;

int StackEsBuit( StElem *Stack ){ return (Stack == NULL ); }

int push( unsigned int h, StElem **Stack ){ StElem *aux;
  if ((aux=(StElem *) malloc(sizeof(StElem))) == NULL) return 0;
  aux->h=h;
  aux->lower=*Stack;
  *Stack= aux;
  return 1;
}

/* pop dividit en dues operacions; no comproven si stack es buit */
unsigned int top( StElem *Stack ){ return Stack->h; }
void deletetop( StElem **Stack ){
  StElem *aux = *Stack;
  *Stack = aux->lower;
  free(aux);
}

void StackFree( StElem **Stack ){ while(!StackEsBuit(*Stack)) deletetop(Stack); }
```

Algorisme amb complexitat lineal
Implementació amb un stack de mida arbitrària amb una llista enllaçada

Implementació de l'algorisme

```
double P[Ndades]; /* La sèrie de borsa */
unsigned int S[Ndades] = { 0U }; /* Vector d'Spans. S[0] = 0 */
register unsigned int i;
StElem *Stack = NULL;

push(0U, &Stack); /* Initializing Stack */
for (i=1; i < Ndades ; i++){
    while( !StackEsBuit(Stack) && P[i] >= P[top(Stack)] ) deletetop(&Stack);
    if (StackEsBuit(Stack)) S[i] = i;
    else S[i] = i - 1 - top(Stack);
    if (!push(i, &Stack)){
       fprintf (stderr, "\nERROR: Stack overfull. Impossible continuar...\n\n");
       return 11;
    }
}
StackFree(&Stack);
```

Exercici

Escriure un programa que usi un stack que contingui les línies d'un fitxer i que imprimeixi el fitxer amb l'ordre de línies invertit.

Fi: estructures lineals i moltes dades