Tipus recursius de dades

Assignatura PRO2

Tardor 2023

Índex

Capítol 10

Tipus recursius de dades

10.1 Introducció a l'ús de tipus recursius de dades

Ja sabem com usar la recursivitat per escriure operacions repetitives: permetem que una operació es cridi a si mateixa per tal d'aplicar el mateix tractament a tots o una part dels elements dels seus paràmetres. S'han de complir, però, certes condicions perquè no hi hagi un nombre infinit de crides:

- els paràmetres de l'operació han de decréixer a cada crida recursiva
- ha d'haver-hi com a mínim un cas directe, és a dir, una opció on no hi hagi cap crida recursiva

Podem aplicar la mateixa idea per construir una estructura complexa de dades. Els elements de l'estructura es poden distribuir en una sèrie d'ítems enllaçats entre ells que anomenarem nodes, de manera que cada node contindrà un element de l'estructura com a camp i almenys un altre camp que es referirà a un altre node del mateix tipus (l'enllaç, que és on apareix la recursivitat). Obviament, no es permet construir estructures d'infinits elements, però a més necessitarem una mena de cas directe. El que fem és definir un valor nul per qualsevol tipus de node, que no té cap camp. Aquest serà la marca de que l'estructura ja s'ha acabat.

Les estructures de dades així construïdes tenen un parell de propietats molt útils. Primera, no cal saber a priori un nombre màxim d'elements per reservar la memòria necessària per emmagatzemar l'estructura, sinó que es pot anar demanant memòria pels nous nodes a mida que es volen afegir elements a l'estructura. Segona, podrem inserir o esborrar elements a l'estructura sense necessitat de moure els altres elements (com passaria en un vector), sinó que només caldrà modificar alguns enllaços entre nodes.

El constructor bàsic de tipus que farem servir en C++ per definir nodes és l'`struct`, el qual ens permet representar tuples de dades heterogènies. A més, introduirem un nou constructor de tipus, anomenat *punter*, que usarem per implementar els enllaços entre nodes.

El concepte de punter procedeix de la gestió de memòria a baix nivell. Considerem que tot objecte d'un programa estarà emmagatzemat en la memòria de l'ordinador on s'executi el

programa ocupant una sèrie de cel·les de memòria, contigües o no. Així, un punter a un cert tipus podrà contenir l'adreça de la primera cel·la de memòria d'un objecte d'aquest tipus. En aquest cas, es diu que una variable de tipus punter "apunta" a l'objecte corresponent. Si un punter no apunta a cap objecte es diu que té el valor `NULL`.

En aquest curs només farem servir punters a tuples (és a dir, a objectes definits amb el constructor *struct*), però en general es poden definir punters a qualsevol tipus. Hem de notar que no tots els llenguatges de programació disposen d'aquest constructor *punter* i que, els que el tenen, no necessàriament els fan servir de la mateixa manera.

10.2 El constructor de tipus punter i la gestió de memòria dinàmica

En C++, per definir un tipus punter a un altre tipus es fa servir una notació molt senzilla: si tenim el tipus `T1`, automàticament podem construir el tipus `T1*`, que significa `punter a T1`. En declarar una variable del tipus `T1*`, per exemple `x`, es diu que aquesta està indefinida. Per definir-la es pot fer alguna d'aquestes operacions:

- Fer-li apuntar a un objecte del tipus `T1` ja existent; això es pot fer assignant-li un altre punter a l'objecte o directament l'adreça de memòria de l'objecte
- Reservar memòria perquè apunti a un objecte nou
- Donar-li el valor `NULL`

Si un punter `x` apunta a un objecte del tipus `T1`, accedim a aquest objecte amb `*x`.

Exemples:

```
struct T1 {
  int camp1;
  bool camp2;
};

T1 tup; // declara i reserva memòria per un objecte tup del tipus T1
tup.camp1= 5;     // assigna valors als camps de l'objecte tup
tup.camp2= true;  // sense usar cap punter

T1* x;  // x és un punter (no inicialitzat) a objectes del tipus T1,
        // qualsevol consulta dóna error fins que no fem l'inicialització

x= &tup;            // fa que x apunti a l'objecte tup (li assigna la seva adreça)
(*x).camp2= false;  // i permet modificar els camps de tup usant el punter x

x= new T1; // reserva memòria per un nou objecte del tipus T1 i fa que x apunti a aquest,
```

```
          // els valors dels camps de *x poden ser qualssevol

(*x).camp1= 0;  // donem valors als camps de *x; també es pot fer amb x->camp1= 0;
(*x).camp2= false;   // i  x->camp2= false;

T1* y;  // y és un punter (no inicialitzat) a objectes del tipus T1

y= x;   // y apunta al mateix objecte que x (aliasing)

(*y).camp1= 20;  // modifiquem el camp1 d'y (i també el de x)

x= NULL;  // x ja no apunta enlloc; qualsevol consulta als camps de *x dóna error

if (x==NULL) (*y).camp2= true;  // un punter inicialitzat es pot comparar
                                // amb NULL i amb d'altres punters

delete y;   // allibera memòria de l'objecte apuntat per y
            // (també es diu que s'esborra l'objecte)
            // només es poden esborrar objectes creats prèviament amb un new
```

Com mostra l'exemple anterior, l'objecte apuntat per un punter es pot esborrar amb l'operació `delete` (més acuradament, es diu que s'allibera la memòria a la qual apunta el punter). Quan es programa amb punters s'ha de procurar no deixar objectes inútils sense esborrar. També hem de tenir cura quan més d'un punter apunta al mateix objecte (*aliasing*), perquè tot i que "s'esborri" l'objecte apuntat per un d'ells, podríem encara mantenir l'accés mitjançant els altres punters a l'objecte esborrat (amb conseqüències impredictibles).

D'altra banda, hem vist també com l'operació `new` serveix per reservar memòria dinàmica per un nou objecte del tipus especificat i retornar un punter a aquest.

10.3 El problema de l'assignació de punters

Ens referirem ara a l'assignació de punters. Si `a` és un punter al tipus `T1`, que no ha estat inicialitzat o té valor `NULL`, la instrucció `a=b` comporta que `a` passa a tenir el mateix valor que `b`. Si `b` apunta a un objecte de `T1`, `a` passa a apuntar al mateix objecte. Si `a` apuntava anteriorment a un altre objecte, aquest no sofreix cap canvi intern, però podria convertir-se en inaccessible per al programa, si no és apuntat per d'altres punters.

Quan un mateix objecte és apuntat per més d'un punter es diu que hi ha un "aliasing", i la conseqüència és que s'han de mesurar molt bé les manipulacions que s'apliquen a través d'un punter, perquè també afecten a tots els altres punters que apunten al mateix objecte. D'això ens en podrem aprofitar en ocassions, però també pot donar lloc a resultats inesperats (i, no cal dir-ho, incorrectes).

Un exemple d'aquesta situació és l'assignació entre nodes. Com que al seu contingut ens trobem amb punters, no podem suposar que funcionarà com una assignació amb variables de

tipus simples, on les dues variables passen a tenir el mateix valor però són independents. Per tant, si es vol crear un node nou a partir d'un altre ja existent, s'ha de definir una operació de còpia per al corresponent tipus. S'haurà de fer el mateix amb tots els tipus construïts a partir de nodes. Per últim, un cas similar es té amb l'esborrat: quan fem `delete n`, no alliberem els possibles nodes següents del node apuntat per `n`. Si volem això, hem de definir una operació d'esborrat per a l'estructura.

Per extensió, tenim una situació semblant quan passem un punter (o alguna cosa que contingui punters) com a paràmetre d'entrada d'una operació, ja sigui passat per valor o per referència constant. En el cas de pas per valor, no podrem comptar amb que es faci una còpia de tota l'estructura enllaçada a partir del punter. Per tant, si dins de l'operació es modifica qualsevol element d'aquesta estructura, el canvi és permanent. Passa el mateix si el punter es passa per referència constant, ja que això no garanteix que no es pugui modificar l'objecte apuntat.

Com a conclusió, si una classe d'objectes s'ha definit mitjançant punters, cal distinguir entre quan dos objectes són el mateix (tenen noms diferents però els seus punters són els mateixos i, per tant, apunten als mateixos nodes) i quan són iguals (els nodes contenen la mateixa informació, siguin els mateixos o no).

10.4 Definició d'estructures recursives de dades

Tornem a la definició d'una estructura recursiva. Per tenir la capacitat d'emmagatzemar informació no afectada per la recursivitat, definirem cada estructura en dos nivells: el superior serà una classe amb tota la informació global de l'estructura (que no volem que es repeteixi per a cada element), i punters a alguns elements distingits (el primer, l'últim, etc., segons el que es necessiti). El nivell inferior serà una tupla privada (`struct`) dins de la classe on es definirà el tipus dels components de l'estructura, pròpiament dits, que normalment direm *nodes*. Cada node constarà de la informació corresponent més un punter al següent element de l'estructura (o més d'un, si l'estructura és arborescent), o l'anterior si volem poder recórrer l'estructura en sentit contrari, per exemple.

```
class estructura_rec {

  private:
     struct node {
         tipus_info info;
         node* següent;  // <--- aqui hi ha la recursivitat
     };

     tipus_que_sigui info_general;
     node*  element_distingit_1;
          ...
  public:
          ...
};
```

Com a propietat invariant de la representació, farem que cada element d'un objecte tingui associat el seu propi node, és a dir, no farem servir un mateix node per representar més d'un element al mateix temps. Tampoc permetrem que dos objectes que no siguin el mateix compartexin nodes, és a dir, un mateix node pot aparèixer a la seqüència de "següents" d'un objecte i només un. Així evitarem que si es modifiquen parts d'un objecte s'afectin involuntàriament altres parts del mateix objecte o d'altres.

En els següents apartats mostrarem implementacions basades en tipus recursius (nodes) i punters per les estructures de dades lineals i arborescents que ja hem usat durant el curs i també veurem la implementació de noves estructures de dades usant les mateixes eines. Cal dir que en el cas de les piles, cues i llistes, introduirem les nostres pròpies implementacions i no mostrarem com estan implementades les classes `stack`, `queue` i `list` de la Standard Template Library (STL) de C++. Tanmateix, les classes i operacions que us donarem implementades, tot i que tindran un nom diferent a les corresponents de la STL, tindran una especificació "equivalent" a la d'aquestes i permetran cobrir les mateixes funcionalitats. En el cas de les llistes, però, hi haurà com veurem alguna petita diferència pel fet de que, en lloc de fer servir iteradors externs a la llista per recórrer-la, usarem un únic cursor intern de la llista (anomenat punt d'interès) que només serà manejable amb operacions de la pròpia classe Llista.

Les classes que us presentarem seran classes genèriques gràcies a l'ús del mecanisme `template` de C++. Encara que no és l'única manera possible d'obtenir genericitat, sí és la més natural i elegant. La implementació de classes `template` (que també anomenarem *templatitzades*), però, té associat un petit problema: no podrem fer l'habitual divisió entre un arxiu `.hpp` amb la definició de la classe (per incloure des d'altres mòduls) i un arxiu `.cpp` amb el codi de les operacions (per compilar separadament), que hem vist en classes no templatitzades, com ara `Estudiant`. Això és degut a que el compilador de C++ no permet compilar codi que usi classes templatitzades sense veure les instanciacions de les mateixes; per tant, cal incloure tot el codi de la classe templatitzada per poder compilar el mòdul usuari d'aquesta, sense esperar a una etapa posterior de muntatge amb un hipotètic arxiu `.o` (com faríem amb `Estudiant`).

Tot i així, el C++ ens proporciona mecanismes que ens permetrien dividir el codi de la classe templatitzada en dos arxius `.hpp` i `.cpp` de manera semblant a la que hem vist fins ara per classes no templatitzades (en el benentès que caldria incloure els dos arxius a l'hora de compilar i no només el `.hpp`). Ara bé, aquests mecanismes suposen una càrrega de notació sintàctica addicional que dificulten innecessàriament la llegibilitat del codi. Per aquesta raó, inclourem la implementació de les operacions dins de la definició de la classe templatitzada en un únic arxiu `.hpp` (no hi haurà arxiu `.cpp` a part). Un exemple d'això és l'arxiu `Arbre.hpp` que heu vist i usat a la sessió 7 del laboratori de PRO2.

10.5 Piles

Les piles són estructures lineals, és a dir, cada element que no sigui nul tindrà un següent i només un. Per aprofitar les possibilitats de la definició inclourem un camp amb l'altura de la pila, que haurem d'actualitzar cada vegada que entri o surti algun element. Suposem que es poden crear piles de qualsevol tipus, que representem amb el nom `T` (i que és el paràmetre del `template`).

```
template <class T> class Pila {
  private:
    struct node_pila {
        T info;
        node_pila* seguent;
    };
    int altura;
    node_pila* primer_node;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

El nostre conveni serà que la pila buida està formada per un valor d'`altura` zero i un `primer_node` nul. Per una altra part, el cim d'una pila no buida serà el seu primer node. Amb això, totes les operacions són immediates, tret del constructor de còpia i del destructor, els quals necessiten recórrer tots els nodes de la pila.

Les operacions privades que tindrem són per copiar i esborrar cadenes de `node_pila`; les farem servir com a auxiliars de la creadora copiadora, de la destructora i de la redefinició de l'assignació. Les declararem `static` i les inclourem dins de l'apartat `private:` de la definició de la classe `Pila`.

Aquí tenim una versió recursiva i una iterativa per copiar cadenes de `node_pila`:

```
static node_pila* copia_node_pila(node_pila* m)
/* Pre: cert */
/* Post: si m és NULL, el resultat és NULL; en cas contrari,
   el resultat apunta al primer node d'una cadena de nodes que són còpia
   de la cadena que té el node apuntat per m com a primer */
{
  node_pila* n;
  if (m==NULL) n=NULL;
  else {
    n = new node_pila;
    n->info = m->info;
    n->seguent = copia_node_pila(m->seguent);
  }
  return n;
}

static node_pila* copia_node_pila(node_pila* m){
  node_pila* n;
  if (m == nullptr) n = nullptr;
  else{
    n = new node_pila;
    node_pila* aux = n;
```

```
    n->info = m->info;
    m = m->seguent;
    while (m != nullptr){
      aux->seguent = new node_pila;
      aux = aux->seguent;
      aux->info = m->info;
      m = m->seguent;
    }
    aux->seguent = nullptr;
  }
  return n;
}
```

Aquí tenim una versió recursiva i una iterativa per esborrar cadenes de `node_pila`:

```
static void esborra_node_pila(node_pila* m)
/* Pre: cert */
/* Post: no fa res si m és NULL, en cas contrari, allibera espai dels nodes de
         la cadena que té el node apuntat per m com a primer */
{
  if (m != NULL) {
    esborra_node_pila(m->seguent);
    delete m;
  }
}

static void esborra_node_pila(node_pila* m)
{
  while (m != NULL) {
    node_pila* aux = m;
    m = m->seguent;
    delete aux;
   }
}
```

Les operacions públiques de `Pila`, que inclourem dins de l'apartat `public:` de la definició de la classe `Pila`, s'especifiquen i implementen a continuació:

```
Pila()
/* Pre: cert */
/* Post: El resultat és una pila sense cap element */
{
  altura= 0;
  primer_node= NULL;
```

```
}

Pila(const Pila& original)
/* Pre: cert */
/* Post: El resultat és una còpia d'original */
{
  altura= original.altura;
  primer_node = copia_node_pila(original.primer_node);
}

~Pila()
// Destructora: Esborra automàticament els objectes locals en
// sortir d'un àmbit de visibilitat
{
  esborra_node_pila(primer_node);
}

Pila& operator=(const Pila& original)
/* Pre: cert */
/* Post: El p.i. passa a ser una còpia d'original i el contingut anterior del p.i.
   ha estat esborrat (excepte si el p.i. i original ja eren el mateix objecte) */
{
  if (this != &original) {
    altura= original.altura;
    esborra_node_pila(primer_node);
    primer_node = copia_node_pila(original.primer_node);
  }
  return *this;
}

void p_buida()
/* Pre: cert */
/* Post: El p.i. passa a ser una pila sense elements i qualsevol
   contingut anterior del p.i. ha estat esborrat */
{
  esborra_node_pila(primer_node);
  altura= 0;
  primer_node= NULL;
}

void empilar (const T& x)
/* Pre: cert */
/* Post: El p.i. és com el p.i. original amb x afegit com a darrer element */
{
  node_pila* aux;
```

```
  aux= new node_pila; // reserva espai pel nou element
  aux->info= x;
  aux->seguent= primer_node;
  primer_node= aux;
  ++altura;
}

void desempilar ()
/* Pre: el p.i. és una pila no buida (<=> primer_node != NULL) */
/* Post: El p.i. és com el p.i. original però sense el darrer
         element afegit al p.i. original */
{
  node_pila* aux;
  aux= primer_node; // conserva l'accés al primer node abans d'avançar
  primer_node= primer_node->seguent; // avança
  delete aux; // allibera l'espai de l'antic cim
  --altura;
}

T cim() const
/* Pre: el p.i. és una pila no buida (<=> primer_node != NULL) */
/* Post: el resultat és el darrer element afegit al p.i. */
{
  return primer_node->info;
}

bool es_buida() const
/* Pre: cert */
/* Post: El resultat indica si el p.i. és una pila buida o no */
{
  return primer_node==NULL;
}

int mida() const
/* Pre: cert */
/* Post: El resultat és el nombre d'elements del p.i. */
{
  return altura;
}
```

Algunes d'aquestes operacions mereixen un comentari que fem extensiu a la resta de classes d'aquest capítol. La creadora copiadora i l'assignació s'han de programar expressament ja que si deixem que actuin amb el seu funcionament per defecte, no obtindríem còpies reals dels objectes, sinó simplement assignacions de punters. De la mateixa manera, la destructora s'ha de programar ja que la seva versió per defecte no aniria visitant els nodes un per un esborrant-los.

Noteu que la creadora copiadora i l'assignació fan essencialment el mateix, tret que la primera produeix un objecte nou i la segona parteix d'un paràmetre implícit que ja existeix de manera que ha d'alliberar l'espai d'aquest abans de fer la còpia (només si les dues piles no són la mateixa). Noteu també que per comprovar si la pila original és la mateixa que el paràmetre implícit es fa servir l'operador de referenciació. Aquest és l'únic ús d'aquest operador que permetem en aquest curs.

L'assignació té una propietat especial: no només copia la pila original al paràmetre implícit sinó que a més retorna una referència a una altra còpia del mateix objecte. Això és el que permet fer assignacions del tipus `p = q = r;` L'opció de retornar referències a objectes obre un ventall de possibilitats que no explorarem aquí (de fet, la considerem prohibida tret d'aquesta operació) però que veureu a d'altres assignatures més avançades.

L'operació `empilar` crea un nou node i assigna al seu camp `info` l'objecte `x` que afegeix a l'estructura. Això garantirà la no compartició de nodes als objectes de la classe.

L'operació `desempilar` allibera l'espai de l'element que treu de l'estructura. Això permetrà que no quedi espai desaprofitat ("memory leak").

10.6 Cues

Les cues també són estructures lineals. Apliquem les mateixes idees, amb la diferència de que ara necessitem accés tant al primer element (per consultar-lo o esborrar-lo) com a l'últim (per afegir un de nou).

```
template <class T> class Cua {
  private:
    struct node_cua {
        T info;
        node_cua* seguent;
    };
    int longitud;
    node_cua* primer_node;
    node_cua* ultim_node;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

El nostre conveni serà que la cua buida està formada per un valor de longitud zero 0 i dos punters a node nuls.

Les operacions privades que necessitarem són per copiar i esborrar cadenes de `node_cua`:

Aquí tenim una versió recursiva i una iterativa per copiar cadenes de `node_cua`:

```
static node_cua* copia_node_cua(node_cua* m, node_cua* &u)
/* Pre: cert */
/* Post: si m és NULL, el resultat i u són NULL; en cas contrari,
```

```
   el resultat apunta al primer node d'una cadena de nodes que són còpia
   de la cadena que té el node apuntat per m com a primer, i u apunta a
   l'últim node */
{
  node_cua* n;
  if (m==NULL) {n=NULL; u=NULL;}
  else {
    n = new node_cua;
    n->info = m->info;
    n->seguent = copia_node_cua(m->seguent, u);
    if (n->seguent == NULL) u= n;
  }
  return n;
}

static node_cua* copia_node_cua(node_cua* m, node_cua* &u){
  node_cua* n;
  if (m == nullptr) n = u = nullptr;
  else{
    n = new node_cua;
    node_cua* aux = n;
    n->info = m->info;
    m = m->seguent;
    while (m != nullptr){
      aux->seguent = new node_cua;
      aux = aux->seguent;
      aux->info = m->info;
      m = m->seguent;
    }
    u = aux;
    u->seguent = nullptr;
  }
  return n;
}
```

Aquí tenim una versió recursiva i una iterativa per esborrar cadenes de `node_cua`:

```
static void esborra_node_cua(node_cua* m)
/* Pre: cert */
/* Post: no fa res si m és NULL, en cas contrari, allibera espai dels nodes de
         la cadena que té el node apuntat per m com a primer */
{
  if (m != NULL) {
```

```
    esborra_node_cua(m->seguent);
    delete m;
  }
}

static void esborra_node_cua(node_pila* m)
{
  while (m != NULL) {
    node_pila* aux = m;
    m = m->seguent;
    delete aux;
   }
}
```

Les operacions públiques s'especifiquen i implementen a continuació:

```
Cua()
/* Pre: cert */
/* Post: El resultat és una cua sense cap element */
{
  longitud= 0;
  primer_node= NULL;
  ultim_node= NULL;
}

Cua(const Cua& original)
/* Pre: cert */
/* Post: El resultat és una còpia d'original */
{
  longitud= original.longitud;
  primer_node= copia_node_cua(original.primer_node, ultim_node);
}

~Cua()
// Destructora: Esborra automàticament els objectes locals en
// sortir d'un àmbit de visibilitat
{
  esborra_node_cua(primer_node);
}

Cua& operator=(const Cua& original)
/* Pre: cert */
/* Post: El p.i. passa a ser una còpia d'original i qualsevol
     contingut anterior del p.i. ha estat esborrat
```

```
     (excepte si el p.i. i original ja eren el mateix objecte) */
{
  if (this != &original) {
    longitud= original.longitud;
    esborra_node_cua(primer_node);
    primer_node = copia_node_cua(original.primer_node, ultim_node);
  }
  return *this;
}

void c_buida()
/* Pre: cert */
/* Post: El p.i. passa a ser una cua sense elements i qualsevol
     contingut anterior del p.i. ha estat esborrat */
{
  esborra_node_cua(primer_node);
  longitud= 0;
  primer_node= NULL;
  ultim_node= NULL;
}

void demanar_torn(const T& x)
/* Pre: cert */
/* Post: El p.i. és com el p.i. original amb x afegit
         com a darrer element */
{
  node_cua* aux;
  aux= new node_cua; // reserva espai pel nou element
  aux->info= x;
  aux->seguent= NULL;
  if (primer_node == NULL) primer_node= aux;
  else ultim_node->seguent= aux;
  ultim_node= aux;
  ++longitud;
}

void avançar()
/* Pre: el p.i. és una cua no buida (<=> primer_node != NULL) */
/* Post: El p.i. és com el p.i. original però sense el primer
         element afegit al p.i. original */
{
  node_cua* aux;
  aux= primer_node; // conserva l'accés al primer node abans d'avançar
  if (primer_node->seguent == NULL) {
    primer_node= NULL;  ultim_node= NULL;
```

```
  }
  else primer_node= primer_node->seguent; // avança
  delete aux; // allibera l'espai de l'antic cim
  --longitud;
}

T primer() const
/* Pre: el p.i. és una cua no buida (<=> primer_node != NULL) */
/* Post: el resultat és el primer element afegit al p.i. */
{
  return primer_node->info;
}

bool es_buida() const
/* Pre: cert */
/* Post: El resultat indica si el p.i. és una cua buida o no */
{
  return longitud==0;
}

int mida() const
/* Pre: cert */
/* Post: El resultat és el nombre d'elements del p.i. */
{
  return longitud;
}
```

10.7 Llistes amb iteradors (i dos sentinelles)

La classe `List` que presentem ara disposa d'iteradors i és comporta de manera pràcticament idèntica a la classe `list` de la STL de C++. Una diferència és que no es poden combinar iteradors mutables i constants en una mateixa expressió. De la mateixa manera que passa a la classe `list` de la STL, tota llista, buida o no, té un element fictici que està darrera del darrer element no fictici.

En un altre capítol hi ha llistes sense iteradors.

```
template <typename T> class List {
private:

  // Items:

  class Item {
  public:
```

```
    T value;
    Item *next;
    Item *prev;
  };

  // Data:

  int _size;
  Item iteminf,itemsup;
```

Una llista buida tindrà dos elements (sentinelles), el valor dels quals és irrellevant, enllaçats entre ells. Aquí tenim l'operació constructora. L'atribut `_size` és zero doncs els dos items no compten com a elements.

```
  List()
  /* Pre: cert*/
  /* Post: el resultat és una llista sense elements */
  {
    _size = 0;
    iteminf.next = &itemsup;
    itemsup.prev = &iteminf;
  }
```

No cal donar valors als altres camps dels elements sentinella perquè mai els usarem.

Ara presentem una sèrie d'operacions auxiliars per inserir i extreure elements individuals i copiar i esborrar cadenes d'elements. Noteu que como que seran privades, podem esmentar elements privats a les especificacions, cosa que no podríem fer amb operacions públiques.

```
void insertItem(Item *pitemprev, Item *pitem)
  /* Pre: pitemprev apunta a un element del p.i.
     pitemprev != pitem,
     pitemprev != &itemsup;
     pitemprev.next = P,
     pitem != nullptr,
     pitem apunta a un element singular */
  /* Post: inserta al p.i. el node apuntat per pitem entre el node apuntat
     per pitemprev i P, augmenta _size en 1 */
  {
    pitem->next = pitemprev->next;
    pitem->next->prev = pitem;
    pitem->prev = pitemprev;
    pitemprev->next = pitem;
    _size++;
  }
```

```
void insertItem(Item *pitemprev, const T &value)
/* Pre: pitemprev apunta un element del p.i.
   pitemprev != &itemsup; pitemprev.next = P */
/* Post: inserta al p.i. un node amb valor value entre el node apuntat
   per pitemprev i P, augmenta _size en 1 */
{
  Item *pitem = new Item;
  pitem->value = value;
  insertItem(pitemprev, pitem);
}

void extractItem(Item *pitem)
/* Pre: pitem != & iteminf, pitem != &itemsup
   pitem apunta a un element del p.i. */
/* Post: enllaça doblement el node anterior a pitem amb el posterior
   a pitem, disminueix _size en 1 */
{
  pitem->next->prev = pitem->prev;
  pitem->prev->next = pitem->next;
  _size--;
}

void removeItem(Item *pitem)
/* Pre: pitem != & iteminf, pitem != &itemsup */
/* Post: enllaça doblement el node anterior a pitem amb el posterior
   a pitem, s'allibera la memòria del node apuntat per pitem,
   disminueix _size en 1 */
{
  extractItem(pitem);
  delete pitem;
}

void removeItems() {
/* Pre: _size = S */
/* Post: s'ha alliberat la memòria dels S nodes entre iteminf i itemsup,
   iteminf.next = &itemsup, itemsup.prev = &iteminf, _size = 0*/
  while (_size > 0)
    removeItem(iteminf.next);
}

void copyItems(const List & l) {
/* Pre: cert*/
/* Post: copia els elements de la llista l al p.i. */
  for (Item *pitem = l.itemsup.prev; pitem != &l.iteminf; pitem = pitem->prev)
    insertItem(&iteminf, pitem->value);
```

```
}
```

Un cop vistes les operacions privades la resta d'operacions públiques són més fàcils d'entendre.

La constructora copiadora comença amb una llista buida i va copiant d'un en un els elements de la llista `l`.

```
List(const List &l)
/* Pre: cert */
/* Post: El resultat és una còpia d'l */
{
  _size = 0;
  iteminf.next = &itemsup;
  itemsup.prev = &iteminf;
  copyItems(l);
}
```

L'assignació té l'estructura usual. Comprova que l'origen sigui diferent al destí, esborrar els possibles elements de l'origen per tal que no es produexi un *memory leak* i copia de un en un els elements de la llista origen `l` .

```
List &operator=(const List &l)
/* Pre: cert */
/* Post: El p.i. passa a ser una còpia d'l i qualsevol
   contingut anterior del p.i. ha estat esborrat
  (excepte si el p.i. i l ja eren el mateix objecte) */
{
  if (this != &l) {
    removeItems();
    copyItems(l);
  }
  return *this;
}
```

La destructora és molt senzilla, doncs només crida a una operació auxiliar.

```
~List()
// Destructora: Esborra automàticament els objectes locals en
// sortir d'un àmbit de visibilitat
{
  removeItems();
}
```

Tenim una sèrie d'operacions públiques estàndard per consultar el nombre d'elements i afegir i treure elements pels extrems. Són molt senzilles i no requereixen gaire explicació.

```
int size() const
/* Pre: cert */
/* Post: el resultat és el nombre de nodes del p.i
   (els nodes iteminf i itemsup no es conten) */
{
  return _size;
}

bool empty() const
/* Pre: cert */
/* Post: el resultat és si el p.i té nodes o no
   (els nodes iteminf i itemsup no conten) */
{
  return _size == 0;
}

void push_back(const T &value)
/* Pre: cert */
/* Post: s'inserta un node amb valor value al final del p.i. */
{
  insertItem(itemsup.prev, value);
}

void push_front(const T &value)
/* Pre: cert */
/* Post: s'inserta un node amb valor value al principi del p.i. */
{
  insertItem(&iteminf, value);
}

void pop_back()
/* Pre: el p.i. no és buit */
/* Post: s'esborra el primer node del p.i. */
{
  if (_size == 0) {
    cerr << "Error: pop_back on empty list" << endl;
    exit(1);
  }
  removeItem(itemsup.prev);
}

void pop_front()
/* Pre: el p.i. no és buit */
/* Post: s'esborra el darrer node del p.i. */
{
```

```
    if (_size == 0) {
      cerr << "Error: pop_front on empty list" << endl;
      exit(1);
    }
    removeItem(iteminf.next);
  }
```

Els iteradors estan implementats en una altra classe i fan servir mecanismes més complexos d'orientació a objectes dels que s'expliquen en aquesta assignatura, per exemple, l'ús de `friends`. En l'operació de desreferenciar tornem una referència, cosa que no fem en cap altre lloc.

En un altre capítol (Llistes amb punt d'interès) hi ha una implementació de llistes autocontinguda en una classe, però és menys versàtil.

Un objecte iterador conté una punter a una llista i a un element. Podem veure les operacions per avançar i retrocedir, desreferenciar l'iterador i comparar iteradors, necessàries per moure's per la llista.

Hi ha operacions per avançar i retrocedir amb un paràmetre `int`. Aquest paràmetre només serveix per diferenciar la versió de preincremente de la de postincrement.

```
class iterator {
  friend class List;
private:
  List *plist;
  Item *pitem;
public:

  // Preincrement
  iterator operator++()
  /* Pre: el p.i apunta a un element E de la llista,
     que no és el end() */
  /* Post: el p.i apunta a l'element següent a E
     el resultat és el p.i. */
  {
    if (pitem == &(plist->itemsup)) {
      cerr << "Error: ++iterator at the end of list" << endl;
      exit(1);
    }
    pitem = pitem->next;
    return *this;
  }

  // Postincrement
  iterator operator++(int)
  /* Pre: p.i. = I,
     el p.i apunta a un element E de la llista,
```

```
   que no és el end() */
/* Post: el p.i apunta a l'element següent a E,
   el resultat és I */
{
  if (pitem == &(plist->itemsup)) {
    cerr << "Error: iterator++ at the end of list" << endl;
    exit(1);
  }
  iterator aux = *this;
  pitem = pitem->next;
  return aux;
}

// Predecrement
iterator operator--()
/* Pre: el p.i apunta a un element E de la llista que
   no és el begin() */
/* Post: el p.i apunta a l'element anterior a E,
   el resultat és el p.i. */
{
  if (pitem == plist->iteminf.next) {
   cerr << "Error: --iterator at the beginning of list" << endl;
   exit(1);
  }
  pitem = pitem->prev;
  return *this;
}

// Postdecrement
iterator operator--(int)
/* Pre: p.i. = I,
   el p.i apunta a un element E de la llista que no és el begin() */
/* Post: el p.i apunta a l'element anterior a E,
   el resultat és I */
{
  if (pitem == plist->iteminf.next) {
    cerr << "Error: iterator-- at the beginning of list" << endl;
    exit(1);
  }
  iterator aux;
  aux = *this;
  pitem = pitem->prev;
  return aux;
}
```

```
  T& operator*()
  /* Pre: el p.i apunta a un element E de la llista,
     que no és el end() */
  /* Post: el resultat és el valor de l'element E */
  {
    if (pitem == &(plist->itemsup)) {
      cerr << "Error: iterator at the end of list" << endl;
      exit(1);
    }
    return pitem->value;
  }

  bool operator==(const iterator &it) const
  /* Pre: cert */
  /* Post: el resultat ens diu si els dos iteradors són iguals */
  {
    return plist == it.plist and pitem == it.pitem;
  }

  bool operator!=(const iterator &it) const
  /* Pre: cert */
  /* Post: el resultat ens diu si els dos iteradors són diferents */
  {
    return not (plist == it.plist and pitem == it.pitem);
  }

};
```

Ja només ens manca associar llistes amb iteradors i poder afegir i esborrar elements. Com es pot veure l'operació `begin` ens proporciona un iterador, associat a la llista p.i., que apunta al primer element de veritat del p.i o a l'element fictici si no hi ha cap altre. L'operació `end` ens proporciona un iterador, associat a la llista p.i, que apunta a l'element fictici del p.i.

```
iterator begin ()
/* Pre: cert */
/* Post: el resultat apunta al primer element del p.i o
   a l'element fictici si el p.i. és buit */
{
  iterator it;
  it.plist = this;
  it.pitem = iteminf.next;
  return it;
}

iterator end()
/* Pre: cert */
```

```
/* Post: el resultat apunta a l'element fictici */
{
  iterator it;
  it.plist = this;
  it.pitem = &itemsup;
  return it;
}
```

Les operacions per inserir i esborrar fent servir iteradors usen operacions auxiliars de llista i ens tornen un iterador imitant una de les capçeleres de les operacions de la classe `list` de C++.

```
iterator insert(iterator it, const T & value)
/* Pre: it ha d'apuntar a un element del p.i o a l'end() del p.i */
/* Post: s'inserta un element amb valor value abans de l'element
   al que apunta it i el resultat apunta al nou element inserit */
{
  if (it.plist != this) {
    cout << "Error: insert with an iterator not on this list" << endl;
    exit(1);
  }
  iterator res = it;
  insertItem(res.pitem->prev, value);
  res.pitem = res.pitem->prev;
  return res;
}

iterator erase(iterator it)
/* Pre: it ha d'apuntar a un element del p.i
   que no sigui l'end() */
/* Post: s'elimina l'element apuntat per it, el
   resultat apunta a l'element posterior a l'eliminat */
{
  if (it.plist != this) {
    cout << "Error: erase with an iterator not on this list" << endl;
    exit(1);
  }
  if (it.pitem == &itemsup) {
    cout << "Error: erase with an iterator pointing to the end of the list" << endl;
    exit(1);
  }
  iterator res = it;
  res.pitem = res.pitem->next;
  removeItem(res.pitem->prev);
  return res;
}
```

També hi ha iteradors constants. No afegim aquí la seva implementació doncs és molt semblant a la dels iteradors mutables. Només direm que per declarar-los hem d'esciure `const_iterator` en comptes d'`iterator` i que les operacions per a posicionar un iterador al principi i al final de la llista es diuen `cbegin()` i `cend()` en comptes de `begin()` i `end()`.

10.8 Llistes amb punt d'interès

La classe `Llista` que implementarem amb nodes enllaçats és una mica diferent que la classe `list` de la STL de C++ que havíem vist fins ara, ja que no veurem en aquest curs com es fa la implementació d'iteradors. Tanmateix, les operacions públiques que implementarem a `Llista` ens permetran mantenir totes les funcionalitats que teníem fins ara, incloent el desplaçament endavant i endarrera d'un punt d'interès per recórrer els elements de la llista. Les operacions d'afegir i eliminar també vindran determinades per la posició del punt d'interès.

La diferència serà que en lloc d'usar objectes iteradors com a punt d'interès, usarem un camp intern de tipus punter a node. De fet, la solució que presentarem ara és més modular que la que usa iteradors per consultar i modificar els elements de la llista, ja que el punt d'interès formarà part de la representació interna de la llista i no serà un objecte que es pugui manipular independentment d'aquesta, com passa en el cas dels iteradors. Així ara tindrem operacions dins de la classe Llista per consultar i modificar l'element apuntat pel punt d'interès, cosa que no teníem en la classe `list`, que usava iteradors.

D'altra banda, ara només podrem tenir un únic punt d'interès per llista, mentre que amb iteradors existeix la possibilitat d'usar més d'un punt d'interès (iterador) simultàniament sobre la mateixa llista. Un altre efecte lateral del fet de tenir dins de la classe `Llista` les operacions per a moure el punt d'interès és que, per a qualsevol operació de cerca o recorregut d'una llista, no podrem passar la llista com a paràmetre per referència constant, encara que no s'hagin de modificar els elements, ja que el punt d'interès sí es modifica i forma part de la llista. Depenent de si volem que la llista quedi modificada o no, passarem aquesta per referència o per valor respectivament.

```
template <class T> class Llista {
  private:
    struct node_llista {
        T info;
        node_llista* seg;
        node_llista* ant;
    };
    int longitud;
    node_llista* primer_node;
    node_llista* ultim_node;
    node_llista* act;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
```

```
};
```

Si no és nul, el punter a node `act` apuntarà a l'element consultable de la llista, al que anomenarem actual. Un valor nul d'`act` s'interpreta com si el punt d'interès estigués situat a la dreta de tot, a sobre d'un element fictici posterior a l'últim element real. Voldrem tenir un accés ràpid tant al primer com a l'últim node de la llista, i per això l'estructura inclou dos punters a aquests nodes. El nostre conveni serà que la llista buida està formada per un valor de `longitud` zero i tres punters (`primer_node`, `ultim_node` i `act`) nuls. Per una altra part, els nodes han de contenir punters al següent i a l'anterior per poder desplaçar `act` endavant i endarrera.

Les operacions privades que necessitarem són per copiar i esborrar cadenes de `node_llista`:

```
static node_llista* copia_node_llista(node_llista* m, node_llista* oact,
                                      node_llista* &u, node_llista* &a)
/* Pre: cert */
/* Post: si m és NULL, el resultat, u i a són NULL; en cas contrari,
   el resultat apunta al primer node d'una cadena de nodes que són còpia de la
   cadena que té el node apuntat per m com a primer, u apunta a l'últim node,
   a és o bé NULL si oact no apunta a cap node de la cadena que comença amb m
   o bé apunta al node còpia del node apuntat per oact */
{
  node_llista* n;
  if (m==NULL) {n=NULL; u=NULL; a=NULL;}
  else {
    n = new node_llista;
    n->info = m->info;
    n->ant = NULL;
    n->seg = copia_node_llista(m->seg, oact, u, a);
    if (n->seg == NULL) u = n;
    else (n->seg)->ant = n;
    if (m == oact) a = n;
  }
  return n;
}

static void esborra_node_llista(node_llista* m)
/* Pre: cert */
/* Post: no fa res si m és NULL, en cas contrari, allibera espai dels nodes de
         la cadena que té el node apuntat per m com a primer */
{
  if (m != NULL) {
    esborra_node_llista(m->seg);
    delete m;
  }
}
```

Les operacions públiques s'especifiquen i s'implementen a continuació:

```
// Constructores

Llista()
/* Pre: cert */
/* Post: el resultat és una llista buida */
{
  longitud= 0;
  primer_node= NULL;
  ultim_node= NULL;
  act= NULL;
}

Llista(const Llista& original)
/* Pre: cert */
/* Post: El resultat és una còpia d'original */
{
    longitud= original.longitud;
    primer_node = copia_node_llista(original.primer_node, original.act,
                                    ultim_node, act);
}

// Destructora

~Llista()
// Destructora: Esborra automàticament els objectes locals en
// sortir d'un àmbit de visibilitat
{
  esborra_node_llista(primer_node);
}

// Redefinició de l'assignació

Llista& operator=(const Llista& original)
/* Pre: cert */
/* Post: El p.i. passa a ser una còpia d'original i qualsevol
     contingut anterior del p.i. ha estat esborrat
     (excepte si el p.i. i original ja eren el mateix objecte) */
{
  if (this != &original) {
    longitud= original.longitud;
    esborra_node_llista(primer_node);
    primer_node = copia_node_llista(original.primer_node, original.act,
                                    ultim_node, act);
  }
  return *this;
```

```
}

// Modificadores

void l_buida()
/* Pre: cert */
/* Post: El p.i. passa a ser una llista sense elements i qualsevol
     contingut anterior del p.i. ha estat esborrat */
{
  esborra_node_llista(primer_node);
  longitud= 0;
  primer_node= NULL;
  ultim_node= NULL;
  act= NULL;
}

void afegir (const T& x)
/* Pre: cert */
/* Post: el p.i. és com el seu valor original, però amb x
   afegit a l'esquerra del punt d'interès */
{
  node_llista* aux;
  aux= new node_llista; // reserva espai pel nou element
  aux->info= x;
  aux->seg= act;
  if (longitud==0) {
    aux->ant= NULL;
    primer_node= aux;
    ultim_node= aux;
  }
  else if (act==NULL) {
    aux->ant= ultim_node;
    ultim_node->seg= aux;
    ultim_node= aux;
  }
  else if (act==primer_node) {
    aux->ant= NULL;
    act->ant= aux;
    primer_node= aux;
  }
  else {
    aux->ant= act->ant;
    (act->ant)->seg= aux;
    act->ant= aux;
  }
```

```
  ++longitud;
}

void eliminar()
/* Pre: el p.i. és una llista no buida i el seu punt d'interès no està a la dreta de tot */
/* Post: El p.i. és com el p.i. original sense l'element on estava el punt d'interès i
   amb el nou punt d'interès avançat una posició a la dreta */
{
  node_llista* aux;
  aux= act; // conserva l'accés al node actual
  if (longitud==1) {
    primer_node= NULL;
    ultim_node= NULL;
  }
  else if (act==primer_node) {
    primer_node= act->seg;
    primer_node->ant= NULL;
  }
  else if (act==ultim_node) {
    ultim_node= act->ant;
    ultim_node->seg= NULL;
  }
  else {
    (act->ant)->seg= act->seg;
    (act->seg)->ant= act->ant;
  }
  act= act->seg; // avança el punt d'interès
  delete aux; // allibera l'espai de l'element esborrat
  --longitud;
}

void concat(Llista& l)
/* Pre: l=L */
/* Post: el p.i. té els seus elements originals seguits pels de
   L, l és buida, i el punt d'interès del p.i. queda situat a
   l'inici */
{
  if (l.longitud>0) {
    if (longitud==0) {
      primer_node= l.primer_node;
      ultim_node= l.ultim_node;
      longitud= l.longitud;
    }
    else {
      ultim_node->seg= l.primer_node;
```

```
      (l.primer_node)->ant= ultim_node;
      ultim_node= l.ultim_node;
      longitud+= l.longitud;
    }
    l.primer_node= NULL;
    l.ultim_node= NULL;
    l.act= NULL;
    l.longitud= 0;
  }
  act= primer_node;
}

// Consultores

bool es_buida() const
/* Pre: cert */
/* Post: El resultat indica si el p.i. és una llista buida o no */
{
  return primer_node==NULL;
}

int mida() const
/* Pre: cert */
/* Post: El resultat és el nombre d'elements de la llista p.i. */
{
  return longitud;
}

// Noves operacions per a consultar i modificar l'element actual
//
// Als comentaris de les següents operacions equiparem el punter act de la llista
// amb un iterador it que referencia l'element actual


T actual() const
/* Pre: el p.i. és una llista no buida i el seu punt d'interès no està a la dreta de tot */
/* Post: el resultat és l'element actual del p.i. */
{
  return act->info; // equival a consultar *it si it és un iterador a l'actual
}

void modifica_actual(const T &x)
/* Pre: el p.i. és una llista no buida i el seu punt d'interès no està a la dreta de tot */
/* Post: el p.i. és com el seu valor original, però amb x reemplaçant l'element actual */
{
```

```
  act->info= x; // equival a fer *it=x; si it és un iterador a l'actual
}

// Noves operacions per a moure el punt d'interès

void inici()
/* Pre: cert */
/* Post: el punt d'interès del p.i. està situat a sobre del primer element de la llista
   o a la dreta de tot si la llista és buida */
{
  act= primer_node; // equival a fer it=l.begin();
}

void fi()
/* Pre: cert */
/* Post: el punt d'interès del p.i. està situat a la dreta de tot */
{
  act= NULL; // equival a fer it=l.end();
}

void avança()
/* Pre: el punt d'interès del p.i. no està a la dreta de tot */
/* Post: el punt d'interès del p.i. està situat una posició més a la dreta que al valor
   original del p.i. */
{
  act= act->seg; // equival a fer ++it;
}

void retrocedeix()
/* Pre: el punt d'interès del p.i. no està a sobre del primer element de la llista*/
/* Post: el punt d'interès del p.i. està situat una posició més a l'esquerra que al
   valor original del p.i. */
{
  if (act==NULL) act=ultim_node; // equival a fer --it;
  else act= act->ant;
}

bool dreta_de_tot() const
/* Pre: cert */
/* Post: el resultat indica si el punt d'interès del p.i. està a la dreta de tot */
{
  return act==NULL;  // equival a comparar it==l.end()
}

bool sobre_el_primer() const
```

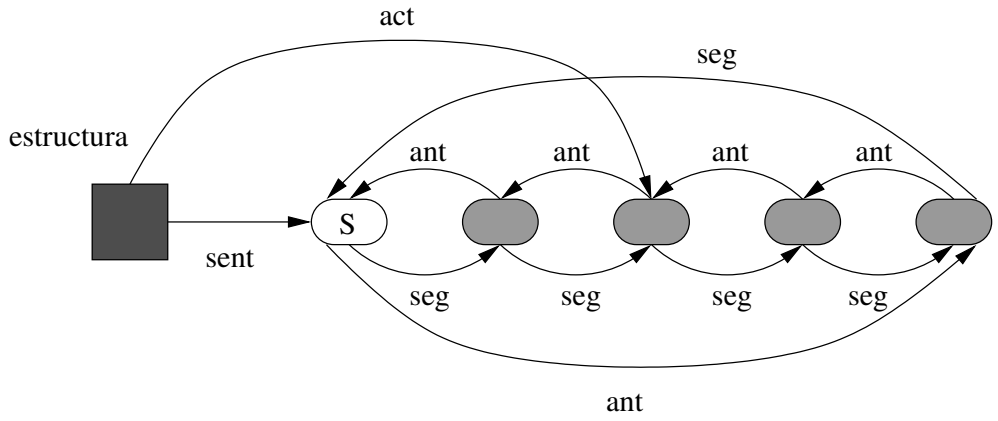

Figura 10.1: Llista amb sentinella

```
/* Pre: cert */
/* Post: el resultat indica si el punt d'interès del p.i. està a sobre del
   primer element del p.i. o està a la dreta de tot si la llista és buida */
{
  return act==primer_node;  // equival a comparar it==l.begin()
}
```

10.9 Llistes circulars doblement encadenades amb sentinella

A la secció anterior hem vist com implementar les llistes amb punt d'interès usant una estructura doblement encadenada de nodes, on el nombre de nodes era el mateix que el nombre d'elements de la llista. En aquesta veurem com, si afegim un node extra anomenat sentinella que no conté cap element real, podem simplificar el codi d'algunes operacions de la classe `Llista`, com ara `afegir` i `eliminar`. L'estructura mai tindrà punters amb valor null i el sentinella farà el paper que aquest tenia a l'exemple anterior. Una possible representació gràfica es mostra a la figura 10.1

L'element sentinella (marcat amb S) és sempre el primer de l'estructura però quan vulguem anar a l'inici, en realitat anirem al següent del sentinella. No està prohibit que l'element actual sigui el sentinella però, si l'és, no es pot esborrar, modificar ni consultar. Diem que el darrer element de l'estructura és l'anterior del sentinella. A més, definim el propi sentinella com al següent del darrer.

Si l'estructura és buida, el sentinella s'apuntarà a si mateix. Tindrà l'aspecte mostrat a la figura 10.2

Mostrem una possible implementació

```
template <class T> class Llista {
  private:
    struct node_llista {
```

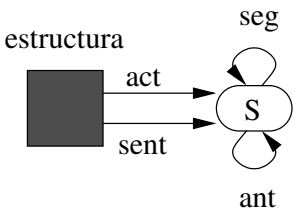

Figura 10.2: Llista amb sentinella buida

```
        T info;
        node_llista* seg;
        node_llista* ant;
    };
    int longitud;
    node_llista* sent;
    node_llista* act;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

El punter a node `sent` apuntarà sempre al sentinella, inclús quan la llista és buida. En aquest últim cas, el següent i l'anterior del sentinella seran el propi sentinella, mentre que si la llista no és buida, el següent del sentinella serà el primer de la llista i l'anterior del sentinella serà l'últim de la llista. Simètricament, si la llista no és buida, el sentinella serà al mateix temps l'anterior del primer i el següent de l'últim.

El punter a node `act` mai valdrà nul sinó que apuntarà a l'element consultable de la llista (l'actual) o bé apuntarà al sentinella en el cas de que el punt d'interès estigui situat a la dreta de tot.

Els punters a node `seg` i `ant` tampoc valdran nul mai, i això obliga a reimplementar les operacions privades que necessitem per copiar i esborrar cadenes de `node_llista`:

```
static node_llista* copia_node_llista(node_llista* m, node_llista* s, node_llista* oact,
                                      node_llista* &ns, node_llista* &a)
/* Pre: s apunta a un sentinella, m i s pertanyen a la mateixa cadena de nodes */
/* Post: si m apunta a s, el resultat, ns i a apunten a una còpia del sentinella;
   en cas contrari, el resultat apunta al primer node d'una cadena de nodes que
   són còpia de la cadena que té el node apuntat per m com a primer i acaba en s,
   ns apunta a la còpia del sentinella s, i a apunta al node còpia del node apuntat
   per oact */
{
  node_llista* n = new node_llista;
  if (m==s) {n->ant=n; n->seg=n; ns=n; a=n;}
```

```
  else {
    n->info = m->info;
    n->seg = copia_node_llista(m->seg, s, oact, ns, a);
    (n->seg)->ant= n;
    ns->seg = n;
    n->ant = ns;
    if (m == oact) a= n;
  }
  return n;
}


static void esborra_node_llista(node_llista* m, node_llista* s)
/* Pre: s apunta a un sentinella, m i s pertanyen a la mateixa cadena de nodes  */
/* Post: si m apunta a s, esborra el sentinella; en cas contrari, allibera espai
         dels nodes de la cadena que té el node apuntat per m com a primer i acaba
         en s */
{
  if (m != s) {
    esborra_node_llista(m->seg, s);
  }
  delete m;
}
```

Les operacions públiques de `Llista` queden implementades de la següent manera (no tornem a escriure les precondicions i les postcondicions que coincideixen amb les de la versió anterior):

```
// Constructores

Llista() {
  longitud= 0;
  sent = new node_llista;
  sent->seg= sent;
  sent->ant= sent;
  act= sent;
}

Llista(const Llista& original) {
    longitud= original.longitud;
    node_llista* aux;
    aux = copia_node_llista((original.sent)->seg, original.sent, original.act,
                            sent, act);
}

// Destructora
```

```
~Llista() {
  esborra_node_llista(sent->seg, sent);
}

// Redefinició de l'assignació

Llista& operator=(const Llista& original) {
  if (this != &original) {
    longitud= original.longitud;
    esborra_node_llista(sent->seg, sent);
    node_llista* aux;
    aux = copia_node_llista((original.sent)->seg, original.sent, original.act,
                            sent, act);
  }
  return *this;
}

// Modificadores

void l_buida() {
  esborra_node_llista(sent->seg, sent);
  longitud= 0;
  sent = new node_llista;
  sent->seg= sent;
  sent->ant= sent;
  act= sent;
}

void afegir (const T& x) {
  node_llista* aux;
  aux= new node_llista; // reserva espai pel nou element
  aux->info= x;
  aux->seg= act;
  aux->ant= act->ant;
  (act->ant)->seg= aux;
  act->ant= aux;
  ++longitud;
}

void eliminar()
/* Pre: l'original + act != sent */
{
  node_llista* aux;
  aux= act; // conserva l'accés al node actual
  (act->ant)->seg= act->seg;
```

```
  (act->seg)->ant= act->ant;
  act= act->seg; // avança el punt d'interès
  delete aux; // allibera l'espai de l'element esborrat
  --longitud;
}

void concat(Llista& l) {
  if (l.longitud>0) {
    if (longitud==0)  swap(sent,l.sent);
    else {
      (sent->ant)->seg= (l.sent)->seg;   // connectem les dues llistes
      ((l.sent)->seg)->ant= sent->ant;
      sent->ant= (l.sent)->ant;    // adaptem sent al fet que l'ultim element
      ((l.sent)->ant)->seg= sent;  // del p.i. passa a ser l'ultim element de l
      (l.sent)->seg= l.sent;    // el sentinella de l passa a
      (l.sent)->ant= l.sent;    // apuntar-se a si mateix
    }
    l.act= l.sent;
    longitud+= l.longitud;
    l.longitud= 0;
  }
  act= sent->seg;
}

// Consultores

bool es_buida() const {
  return longitud==0;
}

int mida() const {
  return longitud;
}

// Noves operacions per a consultar i modificar l'element actual

T actual() const
/* Pre: l'original + act != sent */
{
  return act->info;
}

void modifica_actual(const T &x)
/* Pre: l'original + act != sent */
{
```

```
  act->info= x;
}

// Noves operacions per a moure el punt d'interès

void inici() {
  act= sent->seg;
}

void fi() {
  act= sent;
}

void avança() {
  act= act->seg;
}

void retrocedeix() {
  act= act->ant;
}

bool dreta_de_tot() const {
  return act==sent;
}

bool sobre_el_primer() const {
  return act==(sent->seg);
}
```

10.10 Arbres binaris

Un element d'un arbre binari pot tenir fins a dos següents. Això es reflecteix a la definició de l'estructura fent que el `struct` del node contingui a la seva vegada dos punters a node.

```
template <class T> class Arbre {
  private:
    struct node_arbre {
        T info;
        node_arbre* segE;
        node_arbre* segD;
    };
    node_arbre* primer_node;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
```

```
};
```

Com que ara no hi ha cap informació més que la dels nodes, un arbre buit només consta d'un punter a node nul.

Les operacions privades que necessitarem són per copiar i esborrar jerarquies de nodes:

```
static node_arbre* copia_node_arbre(node_arbre* m)
/* Pre: cert */
/* Post: el resultat és NULL si m és NULL; en cas contrari, el resultat apunta
         al node arrel d'una jerarquia de nodes que és una còpia de la
         jerarquia de nodes que té el node apuntat per m com a arrel */
{
  node_arbre* n;
  if (m==NULL) n=NULL;
  else {
    n = new node_arbre;
    n->info = m->info;
    n->segE = copia_node_arbre(m->segE);
    n->segD = copia_node_arbre(m->segD);
  }
  return n;
}

static void esborra_node_arbre(node_arbre* m)
/* Pre: cert */
/* Post no fa res si m és NULL; en cas contrari, allibera espai de tots els nodes
        de la jerarquia que té el node apuntat per m com a arrel */
{
  if (m != NULL) {
    esborra_node_arbre(m->segE);
    esborra_node_arbre(m->segD);
    delete m;
  }
}
```

L'operació privada `copia_node_arbre` genera recursivament una còpia de la jerarquia de nodes que penja d'un punter donat a `node_arbre`. Es farà servir en la constructora de còpia de la clase i en l'assignació d'arbres. Hem suposat que l'operador d'assignació del tipus `T` de la info funciona com a una operació de còpia.

L'operació privada `esborra_node_arbre` allibera recursivament tots els nodes apuntats per un punter donat a `node_arbre`. Es farà servir en la destructora d'arbre, en l'assignació i en l'acció modificadora de buidar un arbre.

Les operacions públiques s'especifiquen i s'implementen a continuació:

```
Arbre()
/* Pre: cert */
/* Post: el resultat és un arbre buit */
{
  primer_node= NULL;
}

Arbre(const Arbre& original)
/* Pre: cert */
/* Post: el resultat és una còpia d'original */
{
  primer_node = copia_node_arbre(original.primer_node);
}

~Arbre()
// Destructora: Esborra automàticament els objectes locals en
// sortir d'un àmbit de visibilitat
{
  esborra_node_arbre(primer_node);
}

Arbre& operator=(const Arbre& original)
/* Pre: cert */
/* Post: El p.i. passa a ser una còpia d'original i qualsevol contingut
        anterior del p.i. ha estat esborrat (excepte si el p.i. i original
        ja eren el mateix objecte) */
{
  if (this != &original) {
    esborra_node_arbre(primer_node);
    primer_node = copia_node_arbre(original.primer_node);
  }
  return *this;
}

void a_buit()
/* Pre: cert */
/* Post: el p.i. és un arbre buit i qualsevol contingut anterior del p.i.
         ha estat esborrat  */
{
  esborra_node_arbre(primer_node);
  primer_node= NULL;
}

void plantar(const T &x, Arbre &a1, Arbre &a2)
/* Pre: el p.i. és buit, a1=A1, a2=A2, el p.i. no és el mateix objecte que a1 ni que a2 */
```

```
/* Post: el p.i. és un arbre amb arrel igual a x, amb fill esquerre igual a A1
         i amb fill dret igual a A2;  a1 i a2 són buits */
{
  node_arbre* aux;
  aux= new node_arbre;
  aux->info= x;
  aux->segE= a1.primer_node;
  if (a2.primer_node != a1.primer_node  or  a2.primer_node == NULL)
    aux->segD= a2.primer_node;
  else
    aux->segD= copia_node_arbre(a2.primer_node);
  primer_node= aux;
  a1.primer_node= NULL;
  a2.primer_node= NULL;
}

void fills (Arbre &fe, Arbre &fd)
/* Pre: el p.i. no és buit i li diem A, fe i fd són buits
        i no són el mateix objecte */
/* Post: fe és el fill esquerre d'A, fd és el fill dret d'A, el p.i. és buit */
{
   node_arbre* aux;
   aux= primer_node;
   fe.primer_node= aux->segE;
   fd.primer_node= aux->segD;
   primer_node= NULL;
   delete aux;
}

T arrel() const
/* Pre: el p.i. no és buit */
/* Post: el resultat és l'arrel del p.i. */
{
    return primer_node->info;
}

bool es_buit() const
/* Pre: cert */
/* Post: el resultat indica si el p.i. és un arbre buit */
{
    return (primer_node==NULL);
}
```

Noteu que en el cas de fer una crida `a.plantar(x,a,b)` la Post de l'operació ens diu al mateix temps que, a l'acabar l'operació, `a` és buit i `a` és l'arbre resultat de plantar (això últim perquè `a` també és el paràmetre implícit). Evidentment no es poden satisfer les dues coses al

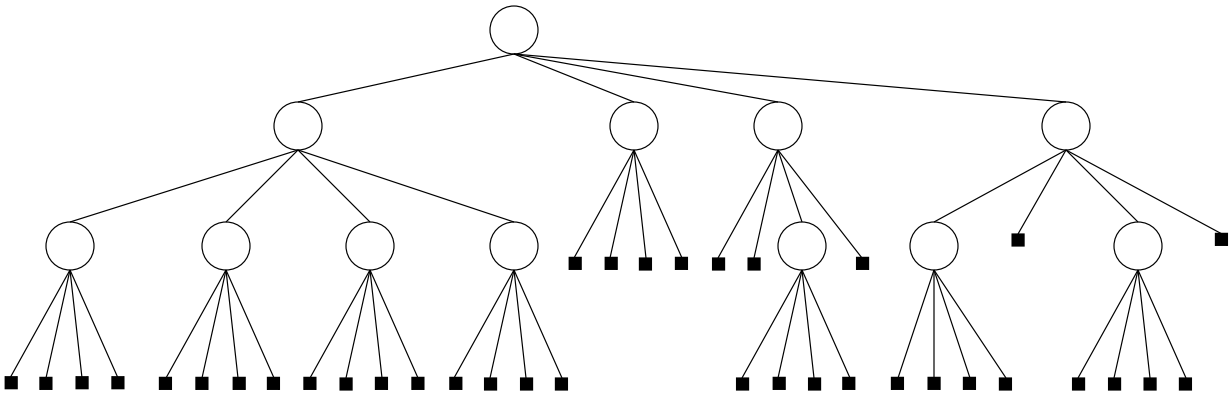

Figura 10.3: Arbre 4-ari

mateix temps i per tant hem dit a la precondició que aquesta crida no és vàlida.

En canvi, sí podem cridar `a.plantar(x,b,b)`, és a dir, permetem que els objectes `a1` i `a2` siguin el mateix arbre. Per tal de garantir la no compartició de nodes i no perdre eficiència, si `a1` i `a2` no són el mateix, simplement aprofitem els seus nodes, conectant-los amb el node de la nova arrel; en cas contrari, en fem una còpia per tal que els nodes connectats per `segE` no siguin els mateixos que els connectats per `segD`.

10.11 Arbres N-aris

Els arbres N-aris són una generalització directa dels arbres binaris on tot arbre no buit té exactament N fills, siguin aquests fills arbres buits o no. Això es reflecteix a la definició de l'estructura fent que el `struct` del node contingui N punters a node, un per cada fill. Per a permetre l'accés directe al fill i-èssim (un cop s'ha arribat al node) cal agrupar els N punters en un vector.

```
template <class T> class ArbreNari {
  private:
    struct node_arbreNari {
        T info;
        vector<node_arbreNari*> seg;
    };
    int N;
    node_arbreNari* primer_node;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

A més del punter al primer node, l'estructura contindrà també un camp enter amb el valor de N donat pel constructor. Així un arbre buit consta d'un punter a node nul i del camp amb el valor de N.

Novament, les operacions privades es dediquen a copiar i esborrar jerarquies de nodes:

```
static node_arbreNari* copia_node_arbreNari(node_arbreNari* m)
/* Pre: cert */
/* Post: el resultat és NULL si m és NULL; en cas contrari, el resultat apunta
         al node arrel d'una jerarquia de nodes que és una còpia de la
         jerarquia de nodes que té el node apuntat per m com a arrel */
{
  node_arbreNari* n;
  if (m==NULL) n=NULL;
  else {
    n = new node_arbreNari;
    n->info = m->info;
    int N = m->seg.size();
    n->seg = vector<node_arbreNari*>(N);
    for (int i=0; i<N; ++i)
      n->seg[i] = copia_node_arbreNari(m->seg[i]);
  }
  return n;
}

static void esborra_node_arbreNari(node_arbreNari* m)
/* Pre: cert */
/* Post no fa res si m és NULL; en cas contrari, allibera espai de tots els nodes
        de la jerarquia que té el node apuntat per m com a arrel */
{
  if (m != NULL) {
    int N = m->seg.size();
    for (int i=0; i<N; ++i)
      esborra_node_arbreNari(m->seg[i]);
    delete m;
  }
}
```

Les operacions públiques s'especifiquen i s'implementen a continuació:

```
ArbreNari(int n)
/* Pre: n>0 */
/* Post: el resultat és un arbre buit d'aritat n */
{
  N = n;
  primer_node= NULL;
}

ArbreNari(const T &x, int n)
```

```
/* Pre: n>0 */
/* Post: el resultat és un arbre d'aritat n amb arrel x i n fills buits */
{
  N = n;
  primer_node= new node_arbreNari;
  primer_node->info = x;
  primer_node->seg = vector<node_arbreNari*>(N);
  for (int i=0; i<N; ++i)
    primer_node->seg[i] = NULL;
}

ArbreNari(const ArbreNari& original)
/* Pre: cert */
/* Post: el resultat és una còpia d'original */
{
  N = original.N;
  primer_node = copia_node_arbreNari(original.primer_node);
}

~ArbreNari()
// Destructora: Esborra automàticament els objectes locals en
// sortir d'un àmbit de visibilitat
{
  esborra_node_arbreNari(primer_node);
}

ArbreNari& operator=(const ArbreNari& original)
/* Pre: cert */
/* Post: El p.i. passa a ser una còpia d'original i qualsevol
     contingut anterior del p.i. ha estat esborrat
     (excepte si el p.i. i original ja eren el mateix objecte) */
{
  if (this != &original) {
    esborra_node_arbreNari(primer_node);
    N = original.N;
    primer_node = copia_node_arbreNari(original.primer_node);
  }
  return *this;
}

void a_buit()
/* Pre: cert */
/* Post: el p.i. és un arbre N-ari buit i qualsevol
         contingut anterior del p.i. ha estat esborrat  */
{
```

```
  esborra_node_arbreNari(primer_node);
  primer_node= NULL;
}

void plantar(const T &x, vector<ArbreNari> &v)
/* Pre:  el p.i. és buit; v.size() és igual a l'aritat del p.i.,
         tots els components de v tenen la mateixa aritat que el p.i.
         i cap d'ells és el mateix objecte que el p.i. */
/* Post: el p.i. té x com a arrel i els N elements originals
         de v com a fills, v passa a contenir arbres buits */
{
  node_arbreNari* aux= new node_arbreNari;
  aux->info= x;
  aux->seg = vector<node_arbreNari*>(N);
  for (int i=0; i<N; ++i) {
    aux->seg[i] = v[i].primer_node;
    v[i].primer_node = NULL;
  }
  primer_node= aux;
}

void fills(vector<ArbreNari> &v)
/* Pre:  el p.i. no és buit i li diem A, v és un vector buit */
/* Post: el p.i. és buit, v passa a contenir els N fills de l'arbre A  */
{
  node_arbreNari* aux= primer_node;
  v = vector<ArbreNari> (N, ArbreNari(N));
  for (int i=0; i<N; ++i)
    v[i].primer_node = aux->seg[i];
  primer_node= NULL;
  delete aux;
}

T arrel() const
/* Pre: el p.i. no és buit */
/* Post: el resultat és l'arrel del p.i. */
{
    return primer_node->info;
}

bool es_buit() const
/* Pre: cert */
/* Post: el resultat indica si el p.i. és un arbre buit o no */
{
    return (primer_node==NULL);
```

```
}

int aritat() const
/* Pre: cert */
/* Post: el resultat és l'aritat del p.i. */
{
    return N;
}
```

Noteu que a l'operació `plantar` no pot haver-hi situacions d'aliasing amb els arbres del vector, ja que quan s'ompli aquest la creadora de còpia i l'assignació de la classe farien les còpies corresponents.

A la figura 10.3, els quadrats negres representen arbres buits. Perquè l'estructura quedi clara convé representar explícitament els arbres buits en els arbres N-aris. Per instanciar arbres d'un tipus concret escriurem `ArbreNari<tipus> nom_arbre;`, per exemple `ArbreNari<int> a;` o `ArbreNari<Estudiant> b;`, etc. L'operació `fills`, que obté de cop (sense necessitat de copiar nodes) un vector d'arbres amb els N fills, ens permet fer un recorregut eficient dels arbres N-aris (de manera similar a l'operació `fills` que hem vist als arbres binaris). A continuació donem un exemple de com usar aquesta operació per sumar els elements d'un arbre N-ari d'enters:

```
int suma_elements(ArbreNari<int> &a)
/* Pre: a=A */
/* Post: el resultat és la suma dels elements d'A */
{
  int s;
  if (a.es_buit()) s=0;
  else {
    s= a.arrel();
    int N = a.aritat();
    vector<ArbreNari<int> > v;
    a.fills(v);
    for (int i=0; i<N; ++i)
      s+= suma_elements(v[i]);
  }
  return s;
}
```

I finalment un exemple de com modificar un arbre N-ari usant les operacions primitives de la classe:

```
void inc_arbreNari(ArbreNari<int> &a, int k)
/* Pre: a=A */
/* Post: a és com A però havent sumat k a tots els seus elements */
{
  if (not a.es_buit()) {
```

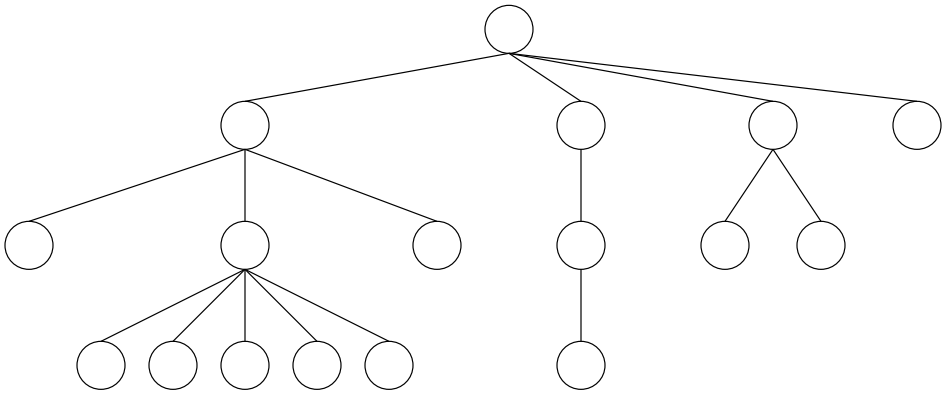

Figura 10.4: Exemple d'un arbre general

```
    int s = a.arrel() + k;
    int N = a.aritat();
    vector<ArbreNari<int> > v;
    a.fills(v);
    for (int i=0; i<N; ++i)
      inc_arbreNari(v[i],k);
    a.plantar(s,v);
  }
}
```

10.12 Arbres generals

Un arbre general pot ser un arbre buit o un arbre no buit que no conté arbres buits com a subarbres. Es diu arbre general perquè cada subarbre pot tenir un nombre indeterminat, fins i tot zero, de fills no buits, però cap fill buit. Si un subarbre no té cap fill, aquest subarbre consta d'un únic node que és una fulla de l'arbre que conté el subarbre. A la figura 10.4 podem veure representat gràficament un exemple d'arbre general.

Els arbres generals es caracteritzen doncs per tenir un nombre indeterminat de fills no buits que pot variar a cada subarbre. Existeixen diferents alternatives per implementar arbres generals usant nodes i punters. Una possibilitat és usar la mateixa estructura que als arbres binaris seguint l'estratègia de representació anomenada "primer fill, germà dret". Una altra possibilitat és tenir a cada node una llista de punters a node per accedir als fills. En els dos casos, si es decideix incloure una operació per consultar el fill i-èssim, la seva implementació seria poc eficient ja que caldria recórrer primer els punters als fills anteriors. Això seria especialment dolent si volguéssim consultar tots els fills i cada vegada haguéssim de començar des del primer.

Tenint a cada node un vector de punters a node podrem garantir un accés ràpid als fills, com en el cas dels arbres N-aris, però ara haurem de manegar d'alguna manera el problema de tenir un nombre variable i indeterminat de fills a cada subarbre. Si imposéssim una restricció sobre

el nombre màxim de fills podriem dimensionar els vectors de punters de cada node amb aquest màxim i afegir un camp enter al node que ens indiqués els fills reals; amb això però, podríem estar malbaratant molt espai de memòria. Si el llenguatge de programació permet emprar vectors dinàmics, com és el cas del C++ mitjançant l'operació `push_back`, ens podrem estalviar el dimensionament previ del vector, però cal tenir en compte que la implementació pròpia de l'operació `push_back` pot ser ineficient en alguns casos (tant en temps com en espai de memòria). Tanmateix, seguirem aquesta idea en la següent implementació dels arbres generals.

```
template <class T> class ArbreGen {
  private:
    struct node_arbreGen {
        T info;
        vector<node_arbreGen*> seg;
    };
    node_arbreGen* primer_node;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Noteu que ara no tenim un camp enter a nivell global de l'estructura pel nombre de fills. És més, tampoc cal afegir-lo dins dels nodes, perquè podrem saber el nombre de fills següents consultant la `size` del vector `seg`.

Les operacions privades de copiar i esborrar jerarquies de nodes d'arbres generals queden idèntiques a les dels arbres N-aris que hem vist abans. Només cal substituir els noms dels nodes corresponents.

Les operacions públiques s'especifiquen i s'implementen a continuació:

```
ArbreGen()
/* Pre: cert */
/* Post: el resultat és un arbre general buit */
{
  primer_node= NULL;
}

ArbreGen(const T &x)
/* Pre: cert */
/* Post: el resultat és un arbre general amb arrel x i sense fills */
{
  primer_node= new node_arbreGen;
  primer_node->info = x;
  // No cal fer primer_node->seg = vector<node_arbreGen*>(0);
}

ArbreGen(const ArbreGen& original)
```

```
/* Pre: cert */
/* Post: el resultat és una còpia d'original */
{
  primer_node = copia_node_arbreGen(original.primer_node);
}

~ArbreGen()
// Destructora: Esborra automàticament els objectes locals en
// sortir d'un àmbit de visibilitat
{
  esborra_node_arbreGen(primer_node);
}

ArbreGen& operator=(const ArbreGen& original)
/* Pre: cert */
/* Post: El p.i. passa a ser una còpia d'original i qualsevol
     contingut anterior del p.i. ha estat esborrat
     (excepte si el p.i. i original ja eren el mateix objecte) */
{
  if (this != &original) {
    esborra_node_arbreGen(primer_node);
    primer_node = copia_node_arbreGen(original.primer_node);
  }
  return *this;
}

void a_buit()
/* Pre: cert */
/* Post: el p.i. és un arbre general buit i qualsevol
         contingut anterior del p.i. ha estat esborrat  */
{
  esborra_node_arbreGen(primer_node);
  primer_node= NULL;
}

void plantar(const T &x)
/* Pre: el p.i. és buit */
/* Post: el p.i. té x com a arrel i zero fills */
{
  primer_node= new node_arbreGen;
  primer_node->info= x;
  // No cal fer primer_node->seg = vector<node_arbreGen*>(0);
}

void plantar(const T &x, vector<ArbreGen> &v)
```

```
/* Pre:  el p.i. és buit i v.size()>0 i cap arbre de v és buit */
/* Post: el p.i. té x com a arrel i els elements originals
         de v com a fills, v passa a contenir arbres buits */
{
  node_arbreGen* aux= new node_arbreGen;
  aux->info= x;
  int n = v.size();
  aux->seg = vector<node_arbreGen*>(n);
  for (int i=0; i<n; ++i) {
    aux->seg[i] = v[i].primer_node;
    v[i].primer_node = NULL;
  }
  primer_node= aux;
}

void afegir_fill(const ArbreGen &a)
/* Pre: el p.i. i a no són buits */
/* Post: el p.i. té un fill més que a l'inici, i aquest nou darrer fill
         és una còpia de l'arbre a */
{
   (primer_node->seg).push_back(copia_node_arbreGen(a.primer_node));
}

void fill(const ArbreGen &a, int i)
/* Pre: el p.i. és buit, a no és buit, i està entre 1 i el nombre de fills d'a */
/* Post: el p.i és una còpia del fill i-èssim d'a */
{
   primer_node= copia_node_arbreGen((a.primer_node)->seg[i-1]);
}

void fills(vector<ArbreGen> &v)
/* Pre:  el p.i. no és buit i li diem A, v és un vector buit */
/* Post: el p.i. és buit, v passa a contenir els fills de l'arbre A  */
{
  node_arbreGen* aux= primer_node;
  int n = aux->seg.size();
  v = vector<ArbreGen> (n);
  for (int i=0; i<n; ++i)
    v[i].primer_node = aux->seg[i];
  primer_node= NULL;
  delete aux;
}

T arrel() const
/* Pre: el p.i. no és buit */
```

```
/* Post: el resultat és l'arrel del p.i. */
{
    return primer_node->info;
}

bool es_buit() const
/* Pre: cert */
/* Post: el resultat indica si el p.i. és un arbre buit */
{
    return (primer_node==NULL);
}

int nombre_fills() const
/* Pre: el p.i. no és buit */
/* Post: el resultat és el nombre de fills del p.i. */
{
    return (primer_node->seg).size();
}
```

Per instanciar arbres generals d'un tipus concret escriurem `ArbreGen<tipus> nom_arbre;`, per exemple `ArbreGen<int> a;` o `ArbreGen<Estudiant> b;`, etc. Noteu que perquè es compleixi que un arbre general no buit no pot contenir arbres generals buits, com a precondició d'`afegir_fill(const ArbreGen& a)`, tenim que `a` no pot ser buit. Pel mateix motiu, a la precondició de `plantar(const T &x, vector<ArbreGen> &v)` hi apareix que cap arbre de `v` és buit.

A continuació donem un exemple d'ús de les operacions per sumar els elements d'un arbre general d'enters. Observeu que a la precondició anomenem `A` al valor inicial del paràmetre d'entrada `a` perquè `a` queda modificat donat que al fer servir l'operació `fills` l'arbre `a` es buida.

```
int suma_elements(ArbreGen<int> &a)
/* Pre: a=A */
/* Post: el resultat és la suma dels elements d'A */
{
  int s;
  if (a.es_buit()) s=0;
  else {
    s= a.arrel();
    vector<ArbreGen<int> > v;
    a.fills(v);
    int n=v.size();
    for (int i=0; i<n; ++i)
      s+= suma_elements(v[i]);
    }
  }
```

```
  return s;
}
```

I finalment un exemple de com modificar un arbre general usant les operacions primitives de la classe:

```
void inc_arbreGen(ArbreGen<int> &a, int k)
/* Pre: a=A */
/* Post: a és com A però havent sumat k a tots els seus elements */
{
  if (not a.es_buit()) {
    int s = a.arrel() + k;
    vector<ArbreGen<int> > v;
    a.fills(v);
    int n=v.size();
    if (n==0)
      a.plantar(s);
    else {
      for (int i=0; i<n; ++i)
        inc_arbreGen(v[i],k);
      a.plantar(s,v);
    }
  }
}
```

10.13 Operacions no primitives

Qualsevol operació pot ser implementada directament sobre una estructura recursiva, sense fer servir les operacions primitives, *sempre que es faci dins de la classe de l'estructura*. En conseqüència, obtindríem una operació primitiva nova. Per exemple, podem escriure el següent codi (dins de la classe Pila) per buscar un element en una pila:

```
bool cerca_pila (const T &x) const
/* Pre: cert */
/* Post: el resultat indica si existeix un element x a la pila p.i. */
{
    return cerca_pila_node(primer_node, x);
}
```

L'operació privada `cerca_pila_node` fa la cerca recursivament:

```
static bool cerca_pila_node (node_pila* n, const T &x)
/* Pre: cert */
/* Post: el resultat diu si x és l'info de *n o d'un dels seus següents */
{
```

```
  bool b;
  if (n==NULL) b=false;
  else if (n->info==x) b=true;
  else b=cerca_pila_node(n->seguent, x);
  return b;
}
```

També es pot resoldre iterativament. Noteu que s'ha de fer servir un punter auxiliar per recórrer la pila sense destruir-la. Recordeu que si modifiquem qualsevol enllaç de la pila, els canvis són permanents.

```
bool cerca_pila (const T &x)
/* Pre: cert */
/* Post: el resultat indica si existeix un element x a la pila p.i. */
{
   node_pila* act;
   act= primer_node;
   bool b= false;
   /*Inv: b= x hi és a la part visitada del p.i. <=>
          b= x hi és a algun node anterior a act */
   while (act!=NULL and not b) {
     b= (act->info == x);
     act= act->seguent;
   }
   return b;
}
```

Pel cas d'accions recursives tenim el mateix. Escrivim una acció dins la classe Arbre que sumi un valor `k` a tots els elements d'un arbre binari d'elements d'un tipus `T` on hi hagi definit un operador de suma.

```
void inc_arbre (const T &k)
/* Pre: A és el valor inicial del p.i. */
/* Post: el p.i. és com A però havent sumat k a tots els seus elements */
{
  inc_node(primer_node, k);
}
```

L'operació privada `inc_node` fa els increments recursivament:

```
static void inc_node (node_arbre* n, const T &k)
/* Pre: cert */
/* Post: el node apuntat per n i tots els seus següents tenen al seu camp
   info la suma de k i el seu valor original */
{
  if (n!=NULL) {
```

```
    n->info += k;
    inc_node(n->segE, k);
    inc_node(n->segD, k);
  }
}
```

Si ara volem fer el mateix procés amb un arbre N-ari, usant la seva representació amb punters, tindrem una implementació interna alternativa a la implementació externa de l'operació `inc_arbreNari` que hem presentat a la secció 10.6 pels arbres N-aris d'enters. La nova implementació, interna a la classe, constaria d'una operació pública

```
void inc_arbreNari (const T &k)
/* Pre: A és el valor inicial del p.i. */
/* Post: el p.i. és com A però havent sumat k a tots els seus elements */
{
  inc_nodeNari(primer_node, k);
}
```

i d'una operació privada recursiva (amb un bucle dins del cas recursiu)

```
static void inc_nodeNari (node_arbreNari* n, const T &k)
/* Pre: cert */
/* Post: el node apuntat per n i tots els seus següents tenen al seu camp
   info la suma de k i el seu valor original */
{
  if (n!=NULL) {
    n->info += k;
    int s = (n->seg).size();
    for (int i=0; i<s; ++i)
      inc_nodeNari(n->seg[i], k);
  }
}
```

A continuació, mostrarem un exemple amb arbres binaris on cal fer servir l'operació de còpia de nodes. Volem una acció que substituexi totes les fulles del paràmetre implícit que continguin el valor `x` per l'arbre `as`.

```
void subst (const T &x, const Arbre &as)
/* Pre: A és el valor inicial del p.i. */
/* Post: el p.i. és com A però havent substituit les fulles que contenien x
         per l'arbre as */
{
  subst_node(primer_node, x, as.primer_node);
}
```

Definim l'operació auxiliar:

```
static void subst_node (node_arbre* &n, const T &x, node_arbre* ns)
/* Pre: cert */
/* Post: els nodes de la jerarquia de nodes que comença al node apuntat per n
   tals que el seu camp info valia x i no tenien següents han estat substituits
   per una còpia de la jerarquia de nodes que comença al node apuntat per ns */
{
  if (n!=NULL) {
    if (n->info == x and n->segE == NULL and n->segD == NULL) {
      delete n;                // si estem a una fulla, l'esborrem
      n= copia_node_arbre(ns); // i copiem el nou node
    }
    else {
      subst_node(n->segE, x, ns);
      subst_node(n->segD, x, ns);
    }
  }
}
```

Ara veurem com generalitzar l'acció anterior al cas dels arbres N-aris. Donem primer l'acció principal i després l'auxiliar recursiva:

```
void substNari (const T &x, const ArbreNari &as)
/* Pre: A és el valor inicial del p.i. (un arbre N-ari) */
/* Post: el p.i. és com A però havent substituit les fulles que contenien x
         per l'arbre N-ari as */
{
  subst_nodeNari(primer_node, x, as.primer_node);
}


static void subst_nodeNari (node_arbreNari* &n, const T &x, node_arbreNari* ns)
/* Pre: cert */
/* Post: els nodes N-aris de la jerarquia de nodes que comença al node
   apuntat per n tals que el seu camp info valia x i no tenien següents
   han estat substituits per una còpia de la jerarquia de nodes N-aris
   que comença al node apuntat per ns */
{
  if (n!=NULL) {
    bool b = (n->info == x);
    int i = 0;
    int s = (n->seg).size();
    while (i<s and b) {        // si la info és la que ens interessa,
      b = (n->seg[i] == NULL); // comprovem si n és una fulla
      ++i;                     // (tots els seus següents són NULL)
    }
    if (b) {
```

```
      delete n;                      // si estem a una fulla, l'esborrem
      n = copia_node_arbreNari(ns); // i copiem el nou node
    }
    else
      for (i=0; i<s; ++i)
        subst_nodeNari(n->seg[i], x, ns);
  }
}
```

Presentem ara l'especificació i implementació d'una acció similar pels arbres generals. Noteu els canvis fets respecte a la solució acabada de donar pels arbres N-aris (altres que els canvis d'uns quants noms de tipus i operacions). En particular, fixeu-vos que afegim a la precondició de l'acció principal que l'arbre `as` no pot ser buit, ja que un subarbre d'un arbre general no pot ser substituit per un arbre buit.

```
void substGen (const T &x, const ArbreGen &as)
/* Pre: A és el valor inicial del p.i. (un arbre general),
        as és un arbre general no buit */
/* Post: el p.i. és com A però havent substituit les fulles que contenien x
         per l'arbre general as */
{
  if (primer_node != NULL)
    subst_nodeGen(primer_node, x, as.primer_node);
}

static void subst_nodeGen (node_arbreGen* &n, const T &x, node_arbreGen* ns)
/* Pre: n!=NULL, ns!=NULL */
/* Post: els nodes generals de la jerarquia de nodes que comença al node
   apuntat per n tals que el seu camp info valia x i no tenien següents
   han estat substituits per una còpia de la jerarquia de nodes generals
   que comença al node apuntat per ns */
{
    int s = (n->seg).size(); // nombre de següents (fills)
    if (n->info == x and s == 0) { // ara n és fulla si s=0;
      delete n;                    // si estem a una fulla, l'esborrem
      n = copia_node_arbreGen(ns); // i copiem el nou node
    }
    else
      for (int i=0; i<s; ++i)
        subst_nodeGen(n->seg[i], x, ns);
}
```

A continuació, dissenyarem una operació que inverteixi l'ordre dels elements d'una llista amb punt d'interès. La resoldrem iterativament basant-nos en la idea d'intercanviar els punters als nodes anterior i següent de cada node. Suposem que és una nova operació de la classe

`Llista` i per tant té un paràmetre implícit d'aquesta classe. El codi que us donem a continuació correspondria a la implementació de les llistes sense sentinella. Us proposem com a exercici modificar el codi de l'operació per adaptar-lo al cas de la implementació amb sentinella.

```
void invertir ()
/* Pre: cert */
/* Post: el p.i. té els mateixos elements que a l'inici però amb l'ordre invertit
         i el seu punt d'interés no s'ha mogut */
{
  node_llista* n = primer_node;
  while (n != ultim_node) {
    /* Inv: els nodes anteriors al que apunta n en la cadena que comença a primer_node
            han intercanviat els seus punters a l'anterior i al següent node */
    node_llista* aux = n->seg;
    n->seg = n->ant;
    n->ant = aux;
    n = aux;
  }
  if (n!=NULL and n!=primer_node) {
    n->seg = n->ant;
    n->ant = NULL;
    ultim_node = primer_node;
    primer_node = n;
  }
}
```

No fem la iteració quan `n == ultim_node`, perquè això suposaria aplicar instruccions redundants quan la llista és buida o té només un element. Les instruccions posteriors al bucle s'encarreguen de tractar el darrer node quan no estem en cap d'aquests casos i d'intercanviar `primer_node` i `ultim_node`.

Ara bé, es pot evitar l'esmentada redundància amb un `if` abans del bucle. Això permet tractar el darrer node dins del bucle i simplificar les instruccions posteriors deixant-les com a un simple `swap`. A més, el que fa el cos del bucle per a cada node és només intercanviar els punters `ant` i `seg`, per tant es pot resoldre de manera equivalent amb un altre `swap`.

```
void invertir ()
/* Pre: cert */
/* Post: el p.i. té els mateixos elements que a l'inici però amb l'ordre invertit
         i el seu punt d'interés no s'ha mogut */
{
  if (longitud > 1) {
    node_llista* n = primer_node;
    while (n != NULL) {
    /* Inv: els nodes anteriors al que apunta n en la cadena que comença a primer_node
            han intercanviat els seus punters a l'anterior i al següent node */
```

```
      node_llista* aux = n->seg;
      swap(n->seg, n->ant);
      n = aux;
    }
    swap(primer_node, ultim_node);
  }
}
```

Deixem com a exercici fer una última millora: evitar l'ús del punter `aux` dins del bucle.

D'aquest exemple es pot extreure una important lliçó. Moltes vegades podem partir d'un disseny bàsic i correcte i fer-lo evolucionar mitjant petits refinaments i millores. Normalment, aquest procés és més eficaç i segur que intentar trobar la solució òptima a la primera.

També donem la versió d'invertir un llista amb dos sentinelles i iteradors. És molt semblant als exemples anteriors, excepte pel fet que cal tractar els sentinelles que són dos `Item` i no dos punters a `Item`.

```
  void invertir()
  /* Pre: cert */
  /* Post: el p.i. té els mateixos elements que a l'inici però
     amb l'ordre invertit i els iteradors no han canviat */
  {
    if (_size > 0){
      Item* p = itemsup.prev;
      while(p != &iteminf){
        swap(p->prev,p->next);
        p=p->next;
      }
      swap(iteminf.next, itemsup.prev);
      iteminf.next->prev = &iteminf;
      itemsup.prev->next = &itemsup;
    }
  }
```

També es pot fer el recorregut dels nodes de principi a fi. Observeu el subtil canvi per avançar a la següent iteració.

```
  void invertir()
  /* Pre: cert */
  /* Post: el p.i. té els mateixos elements que a l'inici però
     amb l'ordre invertit i els iteradors no han canviat */
  {
    if (_size > 0){
      Item* p = iteminf.next;
      while(p != &itemsup){
        swap(p->prev,p->next);
```

```
        p=p->prev;
      }
      swap(iteminf.next, itemsup.prev);
      iteminf.next->prev = &iteminf;
      itemsup.prev->next = &itemsup;
    }
  }
```

Ara presentem un exemple d'una operació d'una llista amb doble sentinella amb paràmetres `iterator`. S'enllacen els elements apuntats pels iteradors i s'allibera la memòria dels elements que hi havia entre ells per a evitar *memory leaks*.

```
  void esborrar_segment(iterator it1, iterator it2)
  /* Pre: it1 i it2 apunten a elements del p.i, l'element apuntat per
     it1 no està després de l'element apuntat per it2 */
  /* Post: s'han esborrat els elements entre it1 i it2 */
  {
    if (it1 != it2){
      Item* p;
      p = it1.pitem->next;
      it1.pitem->next = it2.pitem;
      it2.pitem->prev = it1.pitem;
      while (p != it2.pitem){
        _size--;
        Item* q = p;
        p = p->next;
        delete q;
      }
    }
  }
```

Per últim, aprofitarem aquesta secció per veure com si implementem el recorregut per nivells d'un arbre binari dins de la clase Arbre, podem aconseguir una versió amb cost lineal, ja que així es pot evitar la ineficiència per còpies d'objectes que vam detectar als capítols anteriors.

La idea és que quan fem la immersió amb punters a nodes en lloc d'arbres podem adaptar l'esquema original fent servir una cua de punters a node, de manera que els `push` i els `front` tindran cost constant.

```
void nivells(list<T>& l) const
/* Pre: l és buida */
/* Post: l conté els nodes del p.i. en ordre creixent respecte al nivell
         al qual es troben, i els de cada nivell en ordre d'esquerra a dreta */
{
  nivells_aux(primer_node,l);
}
```

```
static void nivells_aux(node_arbre* n, list<T>& l)
/* Pre: l és buida */
/* Post: l conté els nodes de la jerarquia de nodes que comença al node apuntat
         per n, en ordre creixent respecte al nivell al qual es troben,
         i els de cada nivell en ordre d'esquerra a dreta */
{
  if(n!=NULL){
    queue <node_arbre*> c;
    c.push(n);
    while(not c.empty()){
      n=c.front();
      l.insert(l.end(),n->info);
      node_arbre* n2 = n->segD;
      n=n->segE;
      if (n!=NULL) c.push(n);
      if (n2!=NULL) c.push(n2);
      c.pop();
    }
  }
}
```

El cost de la nova versió és lineal, perque el `while` fa tantes voltes com la mida de l'arbre (a cada volta es copia un element a la llista) i el cost de cada volta és ara constant. Com a curiositat final, intenteu averiguar quina és la longitud màxima que pot arribar a tenir la cua auxiliar, cosa que determinarà l'espai addicional que requereix aquest algorisme.

La mateixa adaptació es pot fer per a la versió recursiva, que us deixem com a exercici. També es poden obtenir versions eficients d'altres operacions que facin tractaments per nivells, com ara cerques. Noteu que la llista s'ha hagut de declarar instanciada amb el mateix tipus genèric del contingut de l'arbre (el "paràmetre" `T` de la declaració *template*) ja que d'altra manera no es pot garantir que la llista admeti els elements de l'arbre en tots els casos.

10.14 Una estructura nova: cues ordenades

Suposem que volem modificar la classe `Cua` per produir cues d'elements que tinguin la propietat addicional de poder ser recorregudes en ordre creixent respecte al valor dels seus elements (cal que hi hagi un operador < definit en el tipus o classe dels elements). Per a això redefinim la implementació amb més punters.

```
template <class T> class CuaOrd {
  private:
    struct node_cuaOrd {
        T info;
        node_cuaOrd* seg_ord;
```

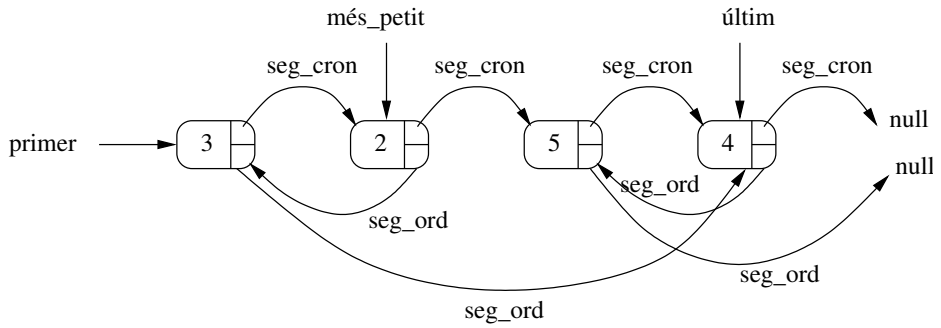

Figura 10.5: Exemple de cua ordenada

```
        node_cuaOrd* seg_cron;
    };
    int longitud;
    node_cuaOrd* primer_node;
    node_cuaOrd* ultim_node;
    node_cuaOrd* mes_petit;
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Els punters `primer_node`, `ultim_node` i `seg_cron` es fan servir per gestionar l'ordre d'arribada a la cua, que anomenem ordre cronològic, mentre que els punters `mes_petit` i `seg_ord` es fan servir per gestionar l'ordre creixent segons el valor dels elements.

Pel que fa a les operacions, aquí només implementarem dues operacions públiques, concretament l'operació `demanar_torn` i l'operació `concatenar`. Deixem com a exercici les modificacions que s'haurien d'aplicar a la resta d'operacions de la classe per adaptar-les a la nova definició dels camps: `avançar` hauria de sofrir una evolució similar a la de `demanar_torn`; la copiadora de nodes es pot fer de manera semblant, però resultaria de cost cuadràtic (per fer-la lineal s'han d'aplicar idees molt astutes), la resta no canviarien gaire respecte a les cues originals.

Per provar aquesta estructura, no només hauríeu de programar operacions que recorrin les cues per l'ordre cronològic sinó també d'altres que les recorrin seguint l'ordre del valor dels seus elements.

```
void demanar_torn(const T& x)
/* Pre: cert */
/* Post: el p.i. ha quedat modificat afegint x com a darrer element
         a l'ordre cronològic i on li toqui a l'ordre creixent */
{
  node_cuaOrd*  n;
  n= new node_cuaOrd;
  n->info= x;
```

```
  n->seg_cron= NULL;
  if (primer_node == NULL) {
    primer_node= n;
    ultim_node= n;
    mes_petit= n;
    n->seg_ord= NULL;
  }
  else {
    ultim_node->seg_cron= n;
    ultim_node= n;
    // Ara s'actualitza l'ordre creixent
    if (x < mes_petit->info) {
      n->seg_ord= mes_petit;
      mes_petit= n;
    }
    else {
      node_cuaOrd* ant = mes_petit;
      bool trobat = false;
      while (ant->seg_ord!=NULL and not trobat) {
        if (x < (ant->seg_ord)->info) trobat=true;
        else ant= ant->seg_ord;
      }
      n->seg_ord= ant->seg_ord;
      ant->seg_ord= n;
    }
  }
  ++longitud;
}

void concatenar(CuaOrd &c2)
/* Pre: cert */
/* Post: el p.i. ha estat modificat posant després del seu últim element
         (cronològic) tots els elements de c2 en el mateix ordre cronològic
          en el qual hi eren a c2, i amb tots els elements reorganitzats
         per satisfer l'ordre creixent; c2 queda buida */
{
  if (c2.primer_node != NULL) { // només cal fer alguna cosa si c2 no és buida
    if (primer_node == NULL) {  // si el p.i. és buit passa a tenir els camps de c2
      primer_node = c2.primer_node;
      ultim_node = c2.ultim_node;
      mes_petit = c2.mes_petit;
    }
    else {                                  // sinó, connectem l'últim del p.i.
      ultim_node->seg_cron = c2.primer_node;  // amb el primer de c2
      ultim_node = c2.ultim_node;             // i actualitzem aquell
      // Ara fem el merge dels nodes de les dues cues segon l'ordre creixent;
```

```
      // els nodes tractats i connectats a la nova cua arriben fins ant
      node_cuaOrd *ant, *act1, *act2;
      act1 = mes_petit;  act2 = c2.mes_petit;
      if (act2->info < act1->info) {
        mes_petit= act2;
        ant= act2;
        act2= act2->seg_ord;
      }
      else {
        ant= act1;
        act1= act1->seg_ord;
      }
      while (act1!=NULL and act2!=NULL) {
        if (act2->info < act1->info) {
          ant->seg_ord= act2;
          ant= act2;
          act2= act2->seg_ord;
        }
        else {
          ant->seg_ord= act1;
          ant= act1;
          act1= act1->seg_ord;
        }
      }
      if (act1 != NULL) ant->seg_ord= act1;
      else ant->seg_ord= act2;
    }
    // Actualitzem longitud del p.i. i els camps de c2
    longitud += c2.longitud;
    c2.primer_node = NULL;
    c2.ultim_node = NULL;
    c2.mes_petit = NULL;
    c2.longitud = 0;
  }
}
```

10.15 Una estructura nova: multillistes

Considerem dos conjunts on cada parell format per un element de cada conjunt pot tenir una relació o no tenir-la. Volem una classe que representi els dos conjunts *i les relacions entre els seus elements*, amb operacions que permetin establir i trencar relacions, consultar si dos elements estan relacionats, consultar tots els elements relacionats amb un de donat, etc.

Si els conjunts són indexables i d'una mida raonable, cadascun d'aquests es pot representar amb un vector i l'estructura que proposaríem per representar les relacions seria probablement

una matriu (vector de vectors), on la posició (i, j) contindria la informació associada a la relació de l'element *i*-èsim del primer conjunt i l'element *j*-èsim del segon (si existeix o no i altres dades rellevants). No obstant, si sabem que hi haurà relativament pocs parells d'elements relacionats, tindrem moltes posicions desaprofitades.

Per exemple, suposem que la FIB vol una classe per guardar les notes de les assignatures matriculades per tots els seus alumnes del Grau durant un quadrimestre determinat. Suposem que la FIB ofereix unes 60 assignatures del Grau i que hi té uns 2000 alumnes. L'esmentada matriu ocuparia 120000 posicions. Però, com sabem, la mitjana d'assignatures per alumne és aproximadament 3, per tant només necessitaríem representar unes 6000 relacions, és a dir, la matriu només aprofitaria el 5% de les seves posicions.

Per evitar aquesta ineficiència (i no empitjorar massa el cost en temps de les diferents operacions) proposem la següent estructura, que anomenem *multillista*. L'adaptem a l'exemple de les assignatures i els estudiants però es pot implementar per tal que sigui genèrica. Més endevant comentem possibles variacions.

```
class multi_llist_est{
private:

  struct nodeML{
    double nota;
    nodeML* seg_est;
    nodeML* seg_assig;
    int pos_est;
    int pos_assig;
  };

  struct info_assig{
    string nom_assig;
    nodeML* primer_est;
  };

  struct info_est{
    int DNI;
    nodeML* primera_assig;
  };

  vector <info_assig>  vassig; // assignatures, ordenat alfabeticament
  vector <info_est> vest; // estudiants, ordenat per DNI

   ...  // especificació i implementació d'operacions privades

  public:

    ...  // especificació i implementació d'operacions públiques
};
```

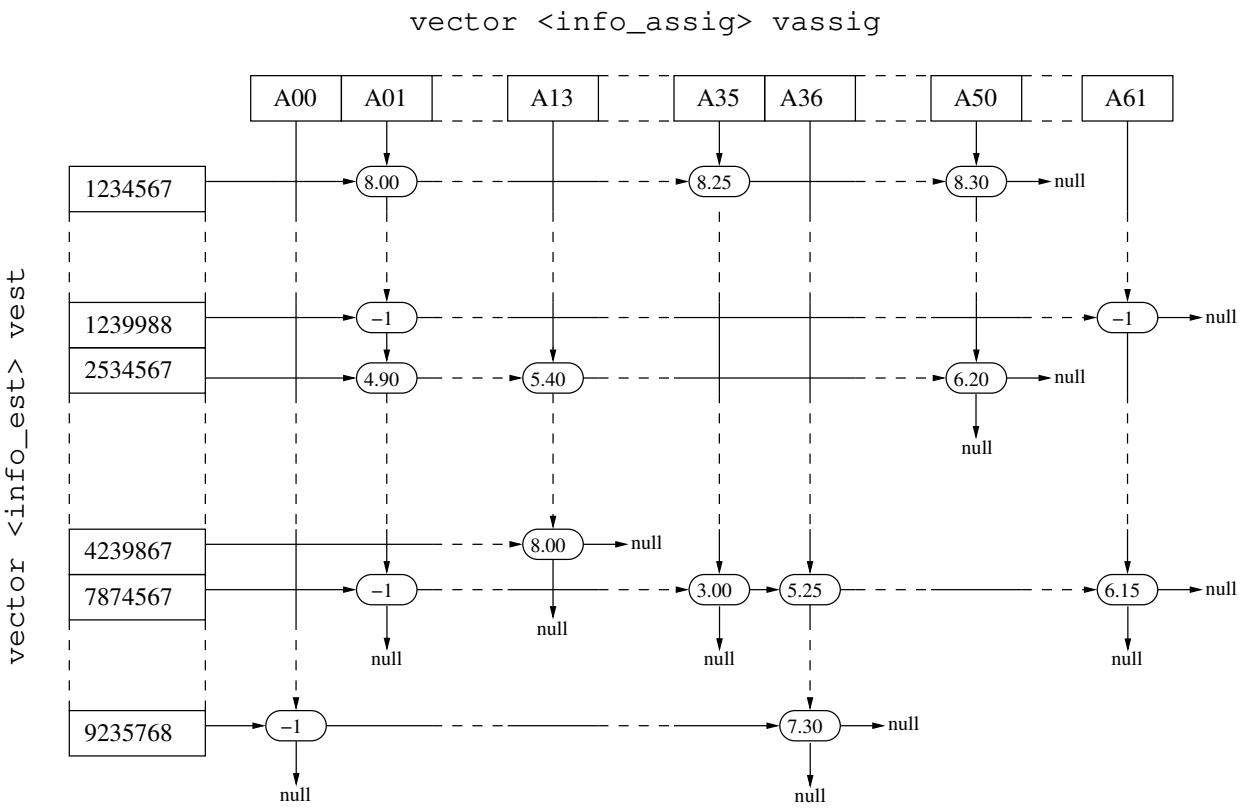

Figura 10.6: Exemple de multillista

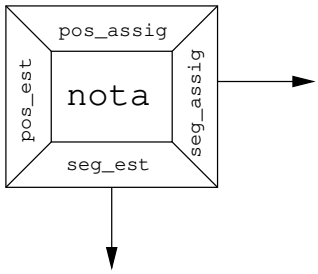

Figura 10.7: Detall d'un `nodeML`

A la figura 10.6 mostrem com queda gràficament l'estructura. Els vectors `vassig` i `vest` representen els conjunts d'assignatures i estudiants, respectivament. Suposem que aquests conjunts no canviaran de mida i, per tant, els vectors estaran plens una vegada inicialitzats: en cas contrari hauríem d'afegir més camps per conèixer el nombre d'elements de cada moment i operacions d'alta i baixa d'elements. No podrem fer servir accès directe, ja que no hi ha una indexació automàtica de les assignatures i els estudiants amb un interval que comenci per 0 (es pot fer, però, amb funcions de conversió, donant lloc a tècniques que veureu a cursos més avançats). Per tant, cada vegada que volguem fer alguna cosa amb un estudiant o assignatura determinats, els haurem de buscar als vectors. Això sí, podem mantenir-los ordenats, per fer cerques més ràpides.

Si els conjunts no tenen una mida afitada, podem representar-los amb llistes, arbres binaris de cerca o estructures més sofisticades, que queden fora de l'àmbit d'aquesta assignatura.

Noteu que els vectors no només contenen la info dels seus elements: cada element porta associat un punter a un `nodeML`, que representa la primera relació de l'estudiant o assignatura corresponent. Cada node d'aquests (veure figura 10.7) contindrà la info rellevant per a la relació (en aquest cas, la nota, si existeix), un punter a cadascuna de les següents relacions (la de l'estudiant i la de l'assignatura) i les posicions de l'estudiant i l'assignatura als vectors dels conjunts respectius (per saber quins són aquests quan recorreguem les seqüències de nodes definides pels punters). El que hem fet, en el fons, és associar una llista a cada estudiant i a cada assignatura, tot i compartint la informació de cada relació.

Pel que fa a les operacions, farem servir la següent especificació:

```
public:

  // Constructores

  multi_llist_est (int m, int n);
  /* Pre: m>0, n>0 */
  /* Post: el resultat és una ML de m assignatures i n estudiants
     on cap estudiant està relacionat amb cap assignatura; els DNI
     i els noms d'assignatura es llegeixen pel cin */

  // Modificadores
```

```
void alta_assig_est (int dni, const string& s, bool& b);
/* Pre: dni és un DNI vàlid del p.i.; s és un nom d'assignatura vàlid del p.i. */
/* Post: si dni no té relació amb s al p.i., passa a tenir-ne (sense nota)
   i b és true; en cas contrari, el p.i. no canvia i b és false */

void posar_nota (int dni, const string& s, double x);
/* Pre: dni és un DNI vàlid del p.i.; s és un nom d'assignatura vàlid
   del p.i.; dni té relació amb s al p.i., x és una nota vàlida */
/* Post: dni passa a tenir nota x en l'assignatura s del p.i.  */

void baixa_assig_est (int dni, const string& s, bool& b);
/* Pre: dni és un DNI vàlid del p.i.; s és un nom d'assignatura vàlid del p.i. */
/* Post: si dni té relació amb s al p.i., deixa de tenir-ne i b és true;
   en cas contrari, el p.i. no canvia i b és false */

// Consultores

double buscar_nota(int dni, const string& s) const;
/* Pre: dni és un DNI vàlid del p.i.; s és un nom d'assignatura vàlid del p.i. */
/* Post: si dni té relació amb s al p.i. el resultat és la nota, si en té
   i -1 si no; si no té relació amb s, el resultat és -2 */

int num_assig () const;
/* Pre: cert */
/* Post: El resultat és el nombre d'assignatures del p.i. */

int num_estudiants () const;
/* Pre: cert */
/* Post: El resultat és el nombre d'estudiants del p.i. */


// Lectura i escriptura

void llistar_est () const;
/* Pre: cert */
/* Post: S'han escrit els estudiants del p.i. pel canal standard de
   sortida, ordenats per DNI */

void llistar_assig () const;
/* Pre: cert */
/* Post: S'han escrit las assignatures del p.i. pel canal standard de
   salida, ordenades alfabeticament */

void llistar_est_assig (const string& s) const;
```

```
  /* Pre: s és una assignatura del p.i. */
  /* Post: S'han escrit els estudiants de s al p.i. pel canal standard de
     sortida, ordenats por DNI, amb la seva nota */

  void llistar_assig_est (int dni) const;
  /* Pre: dni és un estudiant del p.i.  */
  /* Post: S'han escrit les assignatures de dni al p.i. pel canal standard
     de sortida, ordenades alfabeticament, amb la seva nota */

 private:

  static int cerca_dicot_assig(const vector<info_assig> &v, int left, int right,
                               const string& s);
   /* Pre: v[left..right] està ordenat creixentment pel nom de les assignatures,
      0<=left, right<v.size() */
   /* Post: si a v[left..right] hi ha un element amb nom = s, el resultat és
      una posició que el conté; si no, el resultat és -1 */

  static int cerca_dicot_est(const vector<info_est> &v, int left, int right, int n);
   /* Pre: v[left..right] està ordenat creixentment pel DNI dels estudiants,
      0<=left, right<v.size() */
   /* Post: si a v[left..right] hi ha un element amb DNI = n, el resultat és
      una posició que el conté; si no, el resultat és -1 */
```

Només implementarem dues operacions públiques, concretament l'operació `alta_assig_est` i l'operació `posar_nota`. Deixem com a exercici la implementació de la resta d'operacions de la classe.

```
void multi_llist_est::alta_assig_est (int dni, const string& s, bool& b){
  // busquem les coordenades de l'est i l'assig
  int pos_assig = cerca_dicot_assig(vassig,0,vassig.size()-1,s);
  int pos_est = cerca_dicot_est(vest,0,vest.size()-1,dni);

  // busquem les posicions on hauría d'anar el nou node
  // (sempre recordant l'anterior, per permetre la inserció);
  // primer busquem a la llista de l'estudiant, presumiblement molt més
  // curta, encara que cada comparació és una mica més costosa

  bool be = false;
  nodeML* aux_e= vest[pos_est].primera_assig;
  nodeML* ant_e=NULL;
  while(aux_e != NULL and not be){
    // parem si trobem el nom de l'assignatura o un de més gran
    be = s<=vassig[aux_e->pos_assig].nom_assig;
    if (not be) {ant_e=aux_e; aux_e= aux_e->seg_assig;}
  }
```

```
// be indica que l'estudiant té l'assignatura que busquem o una de més gran
// (per tal de no recórrer més nodes dels necessaris); aux_e apunta a
// l'assignatura que busquem si existeix i a la primera més gran si no;
// ant_e apunta a l'anterior, si n'hi ha, i val NULL si no;
// si be és fals, totes són menors que s i aux_e és NULL

if (be and s==vassig[aux_e->pos_assig].nom_assig)  b= false;
// si l'assignatura i l'estudiant ja estan relacionats, no cal fer res més

else {
  b=true;

  // en cas contrari, busquem on s'ha de posar el node a la llista
  // de l'assignatura; sabem que l'estudiant no apareixerà
  nodeML* aux_a= vassig[pos_assig].primer_est;
  bool ba = false;
  nodeML* ant_a=NULL;
  while(aux_a != NULL and not ba){
    ba=vest[aux_a->pos_est].DNI>dni;
    if (not ba) {ant_a=aux_a; aux_a= aux_a->seg_est;}
  }
  // aux_a apunta al primer estudiant amb dni més gran que n, si n'hi ha, i val
  // NULL si no; ant_a apunta a l'estudiant anterior, si n'hi ha, i val NULL si no

  // creem el node
  nodeML* nou_node = new nodeML;
  nou_node->nota=-1.;
  nou_node->pos_est=pos_est;
  nou_node->pos_assig=pos_assig;

  // insertem el node a la llista d'assignatures de l'estudiant
  if (ant_e == NULL) { // el nou node va al principi
    vest[pos_est].primera_assig=nou_node;
  }
  else { // el nou node va després d'ant_e
    ant_e->seg_assig=nou_node;
  }
  nou_node->seg_assig=aux_e; // és fa igualment tant si aux_e és NULL com si no

  // insertem el node a la llista d'estudiants de l'assignatura
  if (ant_a == NULL) { // el nou node va al principi
    vassig[pos_assig].primer_est=nou_node;
  }
  else { // el nou node va després d'ant_a
```

```
      ant_a->seg_est=nou_node;
    }
    nou_node->seg_est=aux_a; // és fa igualment tant si aux_e és NULL com si no
  }
}

void multi_llist_est::posar_nota (int dni, const string& s, double x){
  // nuevament, busquem a la llista de l'est, que serà més curta
  int pos_est = cerca_dicot_est(vest,0,vest.size()-1,dni);
  bool b = false;
  nodeML* aux= vest[pos_est].primera_assig;
  while(not b){
    // parem quan trobem el nom d'assignatura (sabem que hi és)
    b = s==vassig[aux->pos_assig].nom_assig;
    if (not b) {aux=aux->seg_assig;}
  }
  aux->nota=x; // modifiquem el camp nota del node
}
```

Les posicions dels vectors emmagatzemades als nodes es poden estalviar en alguns casos, per exemple fent que les seqüències de nodes siguin circulars (el següent de l'últim és el primer), però llavors hem d'introduir elements sentinella. Un cas on aquesta tècnica representa una millora notable és quan se sap que totes les seqüències en una de les direccions són molt curtes, ja que llavors podem fer tota la volta i retornar al principi per conèixer quin és l'element que estem tractant sense gaire cost. L'exemple dels estudiants i les assignatures es pot beneficiar d'aquesta tècnica, ja que cada estudiant té molt poques assignatures respecte al nombre total d'aquestes.