Tipus recursius de dades (2a sessió)

R. Ferrer i Cancho

Universitat Politècnica de Catalunya

PRO2 (curs 2016-2017)
Versió 0.4

Avís: aquesta presentació no pretén ser un substitut dels apunts oficials de l'assignatura.

On som?

- Tema 7: Tipus recursius de dades
- 11a sessió

Avui

- Com s'implementen llistes (dues maneres diferents), arbres binaris.

Implementació amb nodes enllaçats de la classe `Llista`

Nova variant de *Llista*

- Permet mateixes funcionalitats.
- Novetat: punt d'interès enlloc d'iterador.

No veurem en aquest curs com es fa la implementació d'iteradors

Novetat tipus llista: punt d'interès

- Afegir operacions de desplaçament endavant i endarrera del punt d'interès.
- Operacions d'afegir i eliminar determinades per la posició del punt d'interès.
- Operacions per consultar i modificar l'element apuntat pel punt d'interès.
- Implementació: atribut (privat) de tipus punter a node.
- Avantatge: més modularitat (punt d'interès no manipulable independentment d'aquesta).
- Efecte lateral: passar llista per ref (no constant) per a qualsevol operació de cerca o recorregut.

Definició classe `LLista` I

- Apuntador a node `act` apunta a l'element consultable de la llista (si no es nul)
- Element actual: element apuntat per `act`
- `act` nul s'interpreta com si el punt d'interès estigués al final de tot (a sobre d'un element fictici posterior a l'últim element real).
- Accés ràpid tant al primer com a l'últim node de la llista (= dos punters a aquests nodes).
- Conveni llista buida: `longitud` zero i els tres apuntadors (`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.

Definició classe LLista I

```
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
};
```

Copiar cadenes

```
node_llista* copia_node_llista(node_llista* m, node_llista* oact,
                               node_llista* &u, node_llista* &a) {
/* Pre: cert */
/* Post: si m és nullptr, el resultat, u i a són nullptr; en cas contrari,
   el resultat apunta al primer node d'una cadena de nodes que són còpia de
   de la cadena que té el node apuntat per m com a primer, u apunta a l'últim
   node, i a és o bé nullptr 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 == nullptr) {n = nullptr; u = nullptr; a = nullptr;}
  else {
    n = new node_llista;
    n->info = m->info;
    n->ant = nullptr;
    n->seg = copia_node_llista(m->seg, oact, u, a);
    if (n->seg == nullptr) u = n;
    else (n->seg)->ant = n;
    if (m == oact) a = n;
  }
  return n;
}
```

Esborrar cadenes

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

Constructures i destructora

```
Llista() {
  longitud = 0;
  primer_node = nullptr;
  ultim_node = nullptr;
  act = nullptr;
}

Llista(const Llista& original) {
    longitud= original.longitud;
    primer_node = copia_node_llista(original.primer_node, original.act,
                                    ultim_node, act);
}

~Llista() {
  esborra_node_llista(primer_node);
}
```

Redefinició de l'assignació

```
Llista& operator=(const Llista& original) {
  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 I

```
void l_buida() {
  esborra_node_llista(primer_node);
  longitud = 0;
  primer_node = nullptr;
  ultim_node = nullptr;
  act = nullptr;
}
```

Modificadores II

```
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) { // la llista es buida
    aux->ant = nullptr;
    primer_node = aux;
    ultim_node = aux;
  }
  else if (act == nullptr) { // el punt d'interès es troba
                             // passat el darrer element
    aux->ant = ultim_node;
    ultim_node->seg = aux;
    ultim_node = aux;
  }
  ...
```

Modificadores III

(continuació)

```
  else if (act == primer_node) {
    aux->ant = nullptr;
    act->ant = aux;
    primer_node = aux;
  }
  else {
    aux->ant = act->ant;
    (act->ant)->seg = aux;
    act->ant = aux;
  }
  ++longitud;
}
```

Modificadores IV

```
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 = nullptr;
    ultim_node = nullptr;
  }
  else if (act == primer_node) {
    primer_node = act->seg;
    primer_node->ant = nullptr;
  }
  ...
```

Modificadores V

(continuació)

```
  ...
  else if (act == ultim_node) {
    ultim_node = act->ant;
    ultim_node->seg = nullptr;
  }
  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;
}
```

Modificadores VI

Interès: concatenació eficient respecte concatenació cridant a `afegir`.

```
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'interés del p.i. queda situat a
   l'inici */
  if (l.longitud > 0) {  // si la llista l és buida no cal fer res
    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 = l.ultim_node = l.act = nullptr;
    l.longitud = 0;
  }
  act = primer_node;
}
```

Consultores

```
bool es_buida() const {
  return primer_node == nullptr;
}

int mida() const {
  return longitud;
}
```

Noves operacions per a consultar i modificar l'element actual

```
T actual() const { // equival a consultar *it si it és un iterador
/* 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;
}

void modifica_actual(const T &x) { // equival a fer *it=x (it és un iterador)
/* 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;
}
```

Noves operacions per a moure el punt d'interès I

```
void inici() {   // equival a fer it=l.begin(); si it és un iterador
/* 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;
}

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

void avanca() {  // equival a fer ++it; si it és un iterador
/* 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;
}
```

Noves operacions per a moure el punt d'interès I

```
void retrocedeix() {  // equival a fer --it; si it és un iterador
/* 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 == nullptr) act = ultim_node;
  else act = act->ant;
}

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

bool sobre_el_primer() const { // equival a comparar it==l.begin() si it és un iterador
/* 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;
}
```

Llistes doblement encadenades amb sentinella

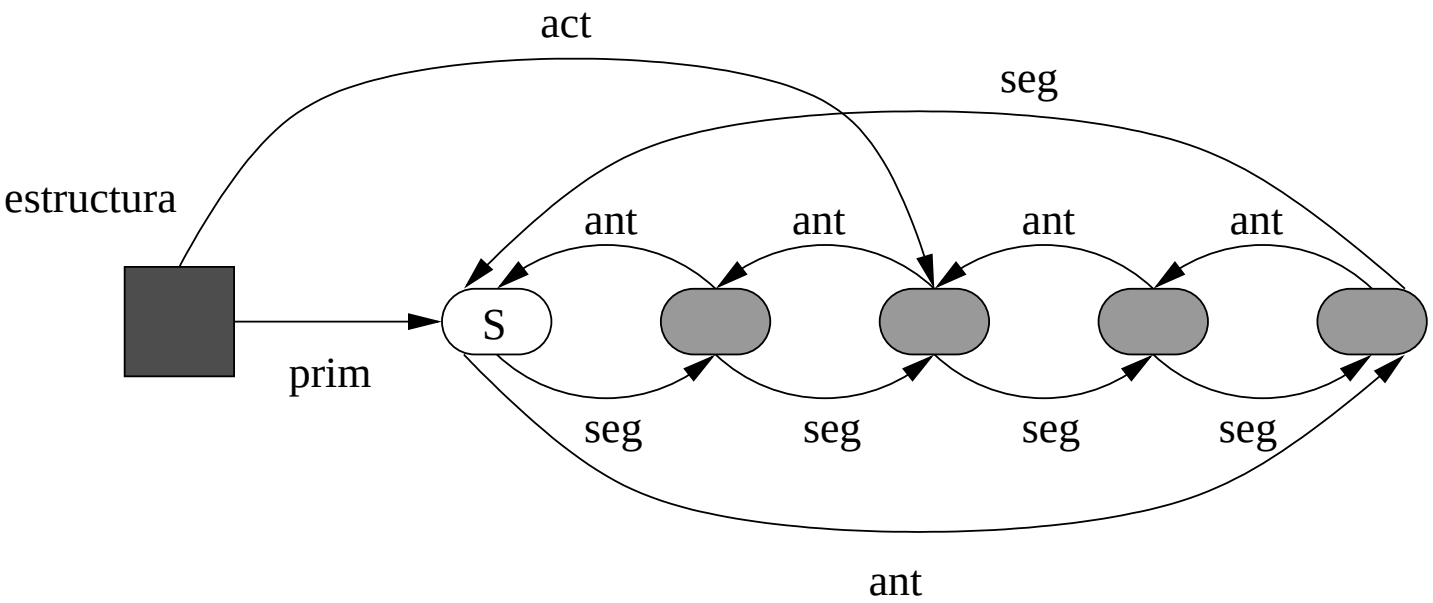
Implementacions llistes:

- Sense sentinella: nombre de nodes = nombre d'elements de la llista.
- Amb sentinella: nombre de nodes = nombre d'elements de la llista + 1

Sentinella:

- Node extra; no conté cap element real.
- Objectiu: simplificar el codi d'algunes operacions com ara `afegir` i `eliminar`.
- L'estructura mai tindrà punters amb valor `nullptr`. El sentinella farà el paper que aquests tenien.

Esquema estructura interna llistes doblement encadenades amb sentinella



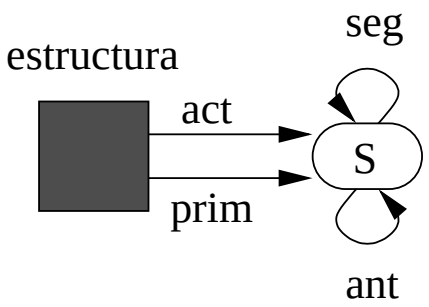
El sentinella

S, l'element sentinella:

- 1er element de l'estructura.
- Anar a l'inici és anar 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.
- El darrer element de l'estructura és l'anterior del sentinella.
- El sentinella és el següent del darrer.

Esquema per llistes buides (amb sentinella i doble encadenament)

Si l'estructura és buida, el sentinella s'apunta a ell mateix.



Un nou atribut privat

sent apunta sempre al sentinella (fins i tot quan la llista és buida).

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

Encadenaments I

Llista buida:

- següent i anterior del sentinella = sentinella

Llista no buida:

- següent del sentinella = primer de la llista
- anterior del sentinella = darrer de la llista
- sentinella = anterior del primer
- sentinella = següent del darrer

Encadenaments II

`act` mai valdrà nul

- `act` apuntarà a l'element consultable de la llista (l'actual) o bé
- `act` 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,

- cal reimplementar les operacions privades per copiar i esborrar cadenes de `node_llista`.

Nova còpia de cadenes de node_llista

```
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;
}
```

Nou esborrament de cadenes de node_llista

```
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;
}
```

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);
}
```

Atenció: valor inicial del primer paràmetre de `copia_node_llista`.

Destructora i assignació

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 I

```
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;
  aux->info = x;
  aux->seg = act;
  aux->ant = act->ant;
  (act->ant)->seg = aux;
  act->ant = aux;
  ++longitud;
}
```

Modificadores II

```
void eliminar()
/* Pre: l'original + act != sent */
{
  node_llista* aux;
  aux= act;
  (act->ant)->seg = act->seg;
  (act->seg)->ant = act->ant;
  act = act->seg;
  delete aux;
  --longitud;
}
```

Modificadores III (ús de swap())

```
void concat(Llista& l) {
  if (l.longitud > 0) {
    if (longitud == 0) swap(set, 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 ell mateix
    }
    l.act= l.sent;
    longitud+= l.longitud;
    l.longitud= 0;
  }
  act = sent->seg;
}
```

Operacions que no canvien amb sentinella o sense

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);
}
```

Definició classe Arbre

- Un element d'un arbre binari: entre 0 i 2 següents.
- struct del node conté dos puntera 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
};
```

Arbre buit: atribut primer_node a node nul

Copiar jerarquies de nodes

Genera recursivament una còpia de la jerarquia de nodes que penja d'un punter a `node_arbre` donat.

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

- Usada en la constructora de còpia d'arbre.
- Hem suposat que l'operador d'assignació del tipus `T` de la info funciona com a una operació de còpia.

Esborrar jerarquies de nodes

Alliberar recursivament tots els nodes apuntats per un punter a `node_arbre`.

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

Usada per la destructora d'arbre i en l'acció modificadora de buidar un arbre.

Constructores i destructora

```
Arbre() {
/* Pre: cert */
/* Post: el p.i. és un arbre buit */
  primer_node= nullptr;
}

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

~Arbre() {
  esborra_node_arbre(primer_node);
}
```

Operador d'assignació i modificadores I

```
Arbre& operator=(const Arbre& original) {
  if (this != &original) {
    esborra_node_arbre(primer_node);
    primer_node = copia_node_arbre(original.primer_node);
  }
  return *this;
}

void a_buit() {
  esborra_node_arbre(primer_node);
  primer_node= nullptr;
}
```

Modificadores II

```
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 == nullptr)
    aux->segD = a2.primer_node;
  else
    aux->segD = copia_node_arbre(a2.primer_node);
  primer_node = aux;
  a1.primer_node= nullptr;
  a2.primer_node= nullptr;
}
```

Què podria passar amb `a.plantar(x,a,b)` (prohibit per la precondició)?

Modificadores III

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

Compte amb

- Què podria passar amb `a.fills(b,b)` (prohibit per la precondició)?
- `a.fills(a,b)`: precondició impossible de satisfer

Consultores

```
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 == nullptr;
}
```