Tipus recursius de dades, II

Ricard Gavaldà

Programació 2
Facultat d'Informàtica de Barcelona, UPC
Primavera 2018

Aquesta presentació no substitueix els apunts

Contingut

Implementació de llistes

Implementació de `Llista`

En aquest curs no implementarem els iteradors de manera general

Implementarem *llistes amb punt d'interès*

Funcionalitats similars, algunes restriccions

Novetat tipus llista: punt d'interès

Podem:

- Desplaçar endavant i enrere el punt d'interès
- Afegir i eliminar just al punt d'interès
- Consultar i modificar l'element al punt d'interès

- Implementació: atribut (privat) de tipus apuntador a node
- Modularitat: punt d'interès part del tipus, no tipus apart
- Efecte lateral: queda modificat si es modifica en una funció que rep la llista per referència no const

Definició classe `Llista`, I

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

Definició classe `Llista`, II

- Apuntadors per accés ràpid a següent, anterior, primer i darrer, i punt d'interès
- `act == NULL` vol dir "punt d'interès sobre l'element fictici posterior a l'últim"
- Conveni llista buida: `longitud` zero i els tres apuntadors (`primer_node`, `ultim_node` i `act`) nuls
- Llista amb un element: `longitud` 1 i únic altre cas en què `primer_node == ultim_node`
- "cap a la dreta" == cap a l'últim; "cap a l'esquerra" == cap al primer; "a la dreta de tot" == sobre l'element fictici del final

Constructures i destructora

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

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);
}
```

Copiar cadena de nodes

```
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 de la cadena que té el node apuntat per m com a primer,
   u apunta a l'últim node,
   i 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 */
```

Copiar cadena de nodes

```
static node_llista* copia_node_llista(
                node_llista* m, node_llista* oact,
                node_llista* &u, node_llista* &a) {
    if (m == NULL) { u = NULL; a = NULL; return NULL; }
    else {
        node_llista* 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) (n->seg)->ant = n;
        if (n->seg == NULL) u = n;
        // else, u es el que hagi retornat la crida recursiva
        // es podria fer com a "else"
        if (m == oact) a = n;
        // else, a es el que hagi retornat la crida recursiva
        return n;
    }
}
```

Esborrar cadena de nodes

```
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;
    }
}
```

Exercici: La versió iterativa

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 = NULL;
    ultim_node = NULL;
    act = NULL;
}
```

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 = new node_llista;
    aux->info = x;
    aux->seg = act;
    if (longitud == 0) { // la llista es buida
        aux->ant = NULL;
        primer_node = aux;
        ultim_node = aux;
    } else if (act == NULL) {
        aux->ant = ultim_node;
        ultim_node->seg = aux;
        ultim_node = aux;
    }
    ...
```

Modificadores III

(continuació)

```
    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;
}
```

Modificadores IV

```
void eliminar() {
/* Pre: el p.i. és una llista no buida i el seu punt d'interès
        no és 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 un element
    més cap 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;
    }
    ...
```

Modificadores V

(continuació)

```
    ...
    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;
}
```

Modificadores VI

Interès: concatenació més eficient que la basada en `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. és al primer element */
    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 = NULL;
        l.longitud = 0;
    }
    act = primer_node;
}
```

Consultores

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

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

Noves operacions per a consultar i modificar l'element actual

```
T actual() const { // equival a consultar *it
/* Pre: el p.i. és una llista no buida i el seu punt d'interès
        no està sobre l'element fictici del final */
/* Post: el resultat és l'element al punt d'interès del p.i. */
    return act->info;
}

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

Noves operacions per a moure el punt d'interès I

```
void inici() {   // equival a fer it = l.begin()
/* Pre: cert */
/* Post: el punt d'interès 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()
/* Pre: cert */
/* Post: el punt d'interès del p.i. està situat
    sobre l'element fictici del final */
    act = NULL;
}
```

Noves operacions per a moure el punt d'interès II

```
void avanca() {  // equival a fer ++it
/* 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
   cap a la dreta que el seu al valor original */
    act = act->seg;
}

void retrocedeix() {  // equival a fer --it
/* Pre: el punt d'interès del p.i. no és el primer element */
/* 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;
    else act = act->ant;
}
```

Noves operacions per a moure el punt d'interès III

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

bool sobre_el_primer() const { // equival a comparar it == l.begin()
/* Pre: cert */
/* Post: si el p.i. no és buit, el resultat indica si el punt d'interès és
   damunt el primer element del p.i.; si és buit indica si el
   punt d'interès si està a la dreta de tot */
    return act == primer_node;
}
```

Llistes doblement encadenades amb sentinella

Llistes doblement encadenades amb sentinella

Implementació de llistes amb sentinella:

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

Llistes amb sentinella

Llista buida:

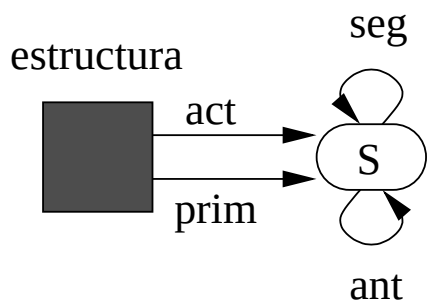
- següent i anterior del sentinella = sentinella

Llista no buida:

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

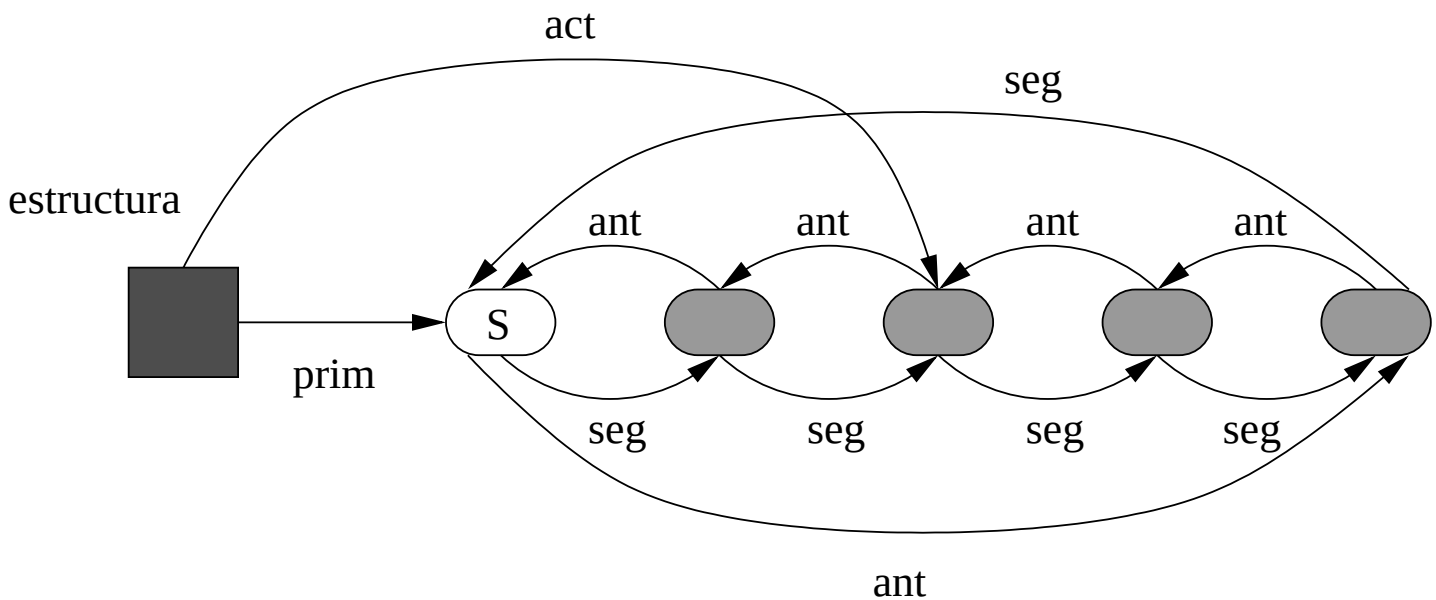
Llistes amb sentinella

Llista buida:



Esquema estructura interna llistes doblement encadenades amb sentinella

Llista no buida:



Implementació dels arbres binaris

Definició de la classe `Arbre`

No coincideix amb la classe `BinTree`
Principals operacions:

- `a.plantar(x,a1,a2)`: Demana que a sigui buit, i sigui objecte diferent d'`a1` i `a2`. Deixa `a1` i `a2` buits.
- `a.fills(a1,a2)`: Demana que Deixa `a1` i `a2` siguin buits, i tots tres objectes `a`, `a1` i `a2` han de ser objectes diferents.

Això fa que per recorrer un arbre s'hagi de "desmuntar". Sovint ineficient.
Inconvenient solucionat a `BinTree` amb *smart pointers* de C++, que no són part de l'assignatura.

Definició de la classe `Arbre`

- `struct` del node conté dos apuntadors a node
- Arbre buit = atribut `primer_node` és nul

```
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
};
```

Constructores i destructora

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

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);
}
```

Copiar jerarquies de nodes

```
static node_arbre* copia_node_arbre(node_arbre* m) {
/* Pre: cert */
/* Post: el resultat és NULL si m és NULL; si no, el resultat apunta
         al node arrel d'una jerarquia de nodes que és una còpia de
         la jerarquia de nodes que té el node apuntat per m com a arrel */
    if (m == NULL) return NULL;
    else {
        node_arbre* n = new node_arbre;
        n->info = m->info;
        n->segE = copia_node_arbre(m->segE);
        n->segD = copia_node_arbre(m->segD);
        return n;
    }
}
```

Notem l'operador = del tipus `T` usat com a una operació de còpia

Esborrar jerarquies de nodes

```
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;
    }
}
```

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 = NULL;
}
```

Modificadores II

```
void plantar(const T &x, Arbre &a1, Arbre &a2) {
/* Pre: el p.i. és buit, a1 = A1, a2 = A2,
   a1 i a2 són objectes diferents del p.i. */
/* Post: el p.i. té x com a arrel, A1 com a fill esquerre i
   A2 com a fill dret; a1 i a2 són buits */
    node_arbre* 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;
}
```

Modificadores III

```
void fills(Arbre &fe, Arbre &fd) {
/* Pre: el p.i. no està buit i li diem A;
        fe, fd són buits i són objectes diferents */
/* Post: fe és igual que el fill esquerre d'A, fd és igual que
         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 = NULL;
}
```

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 == NULL;
}
```