Tipus Recursius de Dades II
Programació 2
Facultat d'Informàtica d'Informàtica, UPC

Conrado Martínez

Primavera 2019

- Apunts basats en els d'en Ricard Gavaldà
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Part I

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

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_llista* ultim;
    node_llista* act;           // apuntador a punt d'interes
    ...  // especificació i implementació d'operacions privades
  public:
    ...  // especificació i implementació d'operacions públiques
};
```

Definició classe `Llista`, II

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

Constructures i destructora

```
Llista() {
  longitud = 0;
  primer = nullptr;
  ultim = nullptr;
  act = nullptr;
}

Llista(const Llista& original) {
    longitud = original.longitud;
    primer = copia_node_llista(original.primer, original.act,
                                        ultim, act);
}

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

Copiar cadena de nodes

```
static 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 no
   que són còpia de de la cadena que té el node apuntat per m com a prir
   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 */
```

Copiar cadena de nodes

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

Esborrar cadena de nodes

```
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;
    }
}
```

Exercici: La versió iterativa

Redefinició de l'assignació

```
Llista& operator=(const Llista& original) {
    if (this != &original) {
      longitud = original.longitud;
      node_llista *ultim_aux, *act_aux;
      node_llista* primer_aux = copia_node_llista(original.primer,
                                    original.act, ultim_aux, act_aux);
      esborra_node_llista(primer);
      primer = primer_aux; ultim = ultim_aux; act = act_aux;
  }
  return *this;
}
```

Redefinició de l'assignació: una tècnica alternativa

```
// intercanvi de la llista implícita amb la llista aux
void Swap(LLista& aux) {
  swap(longitud, aux.longitud);
  swap(primer, aux.primer);
  swap(ultim, aux.ultim);
  swap(act, aux.act);
}

Llista& operator=(const Llista& original) {
  Llista aux = original; // amb la construcció per còpia
  Swap(aux);
  return *this;
}
```

Modificadores I

```
void l_buida() {
    esborra_node_llista(primer);
    longitud = 0;
    primer = nullptr;
    ultim = nullptr;
    act = nullptr;
}
```

Modificadores II

```
void afegir(const T& x) {
/* Pre: cert */
/* Post: la llista queda com originalment, però amb x
   afegit a l'esquerra del punt d'interès */
    node_llista* aux = new node_llista;
    aux -> info = x;
    aux -> seg = act;
    if (longitud == 0) { // la llista es buida
        aux -> ant = nullptr;
        primer = aux;
        ultim = aux;
    } else if (act == nullptr) {
        aux -> ant = ultim;
        ultim -> seg = aux;
        ultim = aux;
    }
    ...
```

Modificadores III

(continuació)

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

Modificadores IV

```
void eliminar() {
/* Pre: la llista no és buida i el seu punt d'interès
        no és a la dreta de tot */
/* Post: la llista queda com originalment però sense l'element
         on estava el punt d'interès i amb el nou punt d'interès
         apuntant al successor de l'element esborrat */

    node_llista* aux = act; // conserva l'accés al node actual
    if (longitud == 1) {
        primer = nullptr;
        ultim = nullptr;
    } else if (act == primer) {
        primer = act -> seg;
        primer -> ant = nullptr;
    }
    ...
```

Modificadores V

(continuació)

```
    ...
    else if (act == ultim) {
        ultim = act -> ant;
        ultim -> 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 més eficient que la basada en `afegir`

```
void concat(Llista& l) {
/* Pre: l = L */
/* Post: la llista conté els seus elements originals seguits pels
         de L, l queda buida, i el punt d'interés passa a ser el
         primer element */
    if (l.longitud > 0) {  // l buida → no cal fer res
        if (longitud == 0) {
            primer = l.primer;
        } else {
            ultim -> seg = l.primer;
            l.primer -> ant = ultim;
        }
        ultim = l.ultim;
        longitud += l.longitud;
        l.primer = l.ultim = l.act = nullptr; l.longitud = 0;
    }
    act = primer;
}
```

Consultores

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

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

Noves operacions per a consultar i modificar l'element actual

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

void modifica_actual(const T &x) { // equival a fer *it = x
/* Pre: la llista no és buida i el seu punt d'interès no està
        a la dreta de tot*/
/* Post: la llista queda com originalment, però amb x reemplaçant
        l'element actual */
    act -> info = x;
}
```

Noves operacions per a moure el punt d'interès I

```
void inici() {    // equival a fer it = l.begin()
/* Pre: cert */
/* Post: el punt d'interès de la llista apunta al primer
   element de la llista, o a la dreta de tot si la llista és buida */
    act = primer;
}

void fi() {       // equival a fer it = l.end()
/* Pre: cert */
/* Post: el punt d'interès queda situat
    sobre l'element fictici del final */
    act = nullptr;
}
```

Noves operacions per a moure el punt d'interès II

```
void avanca() {  // equival a fer ++it
/* Pre: el punt d'interès no està a la dreta de tot */
/* Post: el punt d'interès apunta al successor de l'element al qual
         apuntava originalment, és a dir es mou cap a la dreta
         del seu al valor original */
    act = act -> seg;
}

void retrocedeix() {  // equival a fer --it
/* Pre: el punt d'interès no és el primer element de la llista */
/* Post: el punt d'interès apunta al predecessor de l'element al qual
         apuntava originalment, o apunta a l'últim eleemnt de la llista
         si estava apuntant a la dreta de tot; és a dir es mou cap a
         l'esquerra del seu al valor original */
    if (act == nullptr) act = ultim;
    else act = act -> ant;
}
```

Noves operacions per a moure el punt d'interès III

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

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

Part I

Llistes doblement encadenades amb sentinella

Implementació de llistes amb sentinella:

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

Llistes amb sentinella

Llista buida:

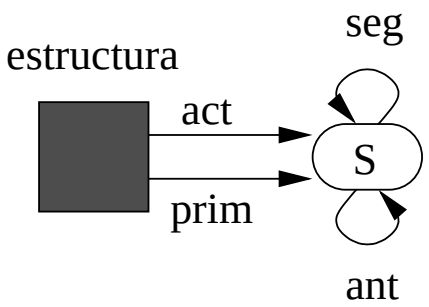
- següent i anterior del sentinella = sentinella

Llista no buida:

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

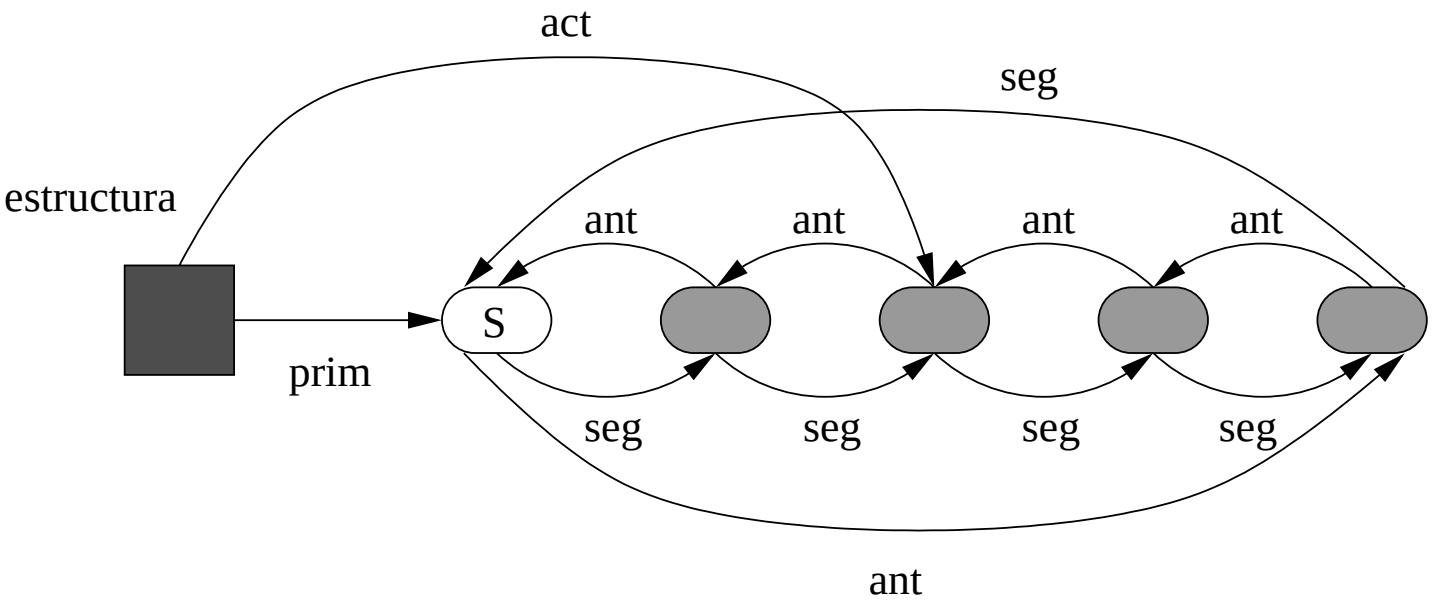
Llistes amb sentinella

Llista buida:



Esquema estructura interna llistes doblement encadenades amb sentinella

Llista no buida:



Un nou atribut privat

`sent` apunta sempre al node sentinella, que existeix fins i tot quan la llista és buida

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

Implementació de privades i públiques $\longrightarrow$ apunts

Part I

Definició de la classe `Arbre`

No coincideix amb la classe `BinTree`
Principals operacions:

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

Definició de la classe `Arbre`

No coincideix amb la classe `BinTree`
Principals operacions:

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

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

Definició de la classe `Arbre`

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

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

Constructores i destructora

```
Arbre() {
/* Pre: cert */
/* Post: crea un arbre buit */
    arrel = nullptr;
}

Arbre(const Arbre& original) {
/* Pre: cert */
/* Post: crea un arbre que és una còpia d'original */
    arrel = copia_node_arbre(original.arrel);
}

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

Copiar jerarquies de nodes

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

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

Esborrar jerarquies de nodes

```
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 -> esq);
        esborra_node_arbre(m -> dre);
        delete m;
    }
}
```

Operador d'assignació i modificadores I

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

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

Modificadores II

```
void plantar(const T &x, Arbre &a1, Arbre &a2) {
/* Pre: l'arbre implícit és buit, a1 = A1, a2 = A2,
   a1 i a2 són objectes diferents de l'arbre implícit */
/* Post: l'arbre implícit 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 -> esq = a1.arrel;
    if (a2.arrel != a1.arrel or a2.arrel == nullptr)
        aux -> dre = a2.arrel;
    else
      aux -> dre = copia_node_arbre(a2.arrel);
    arrel = aux;
    a1.arrel = nullptr;
    a2.arrel = nullptr;
}
```

Modificadores III

```
void fills(Arbre& fe, Arbre& fd) {
/* Pre: l'arbre no està buit,
        fe, fd són dos arbres buits i són objectes diferents */
/* Post: fe és el fill esquerre de l'arbre implícit original,
         fd és el fill dret de l'arbre implícit original,
         l'arbre implícit queda buit */
    fe.arrel = arrel -> esq;
    fd.arrel = arrel -> dre;
    delete arrel;
    arrel = nullptr;
}
```

Consultores

```
T arrel() const {
/* Pre: l'arbre no és buit */
/* Post: retorna el valor de l'arrel de l'arbre */
    return arrel -> info;
}

bool es_buit() const {
/* Pre: cert */
/* Post: retorna cert si i només si l'arbre és buit */
    return arrel == nullptr;
}
```