# Estructures arborescents

Assignatura PRO2

Setembre 2017

# Índex

# Capítol 4

# Estructures de dades arborescents

Un arbre és una estructura de dades no lineal, donat que els seus elements poden tenir més d'un element successor. Cada element d'un arbre es diu *node*. Un arbre no buit, és a dir, que tingui almenys un element, té un node distingit anomenat *arrel*, que és l'unic consultable individualment. Aquest node arrel està connectat amb zero o més arbres, que es denominen *fills*. Cada un dels fills és consultable individualment. Una *fulla* és un node sense successors. Un *camí* és una successió de nodes que van de l'arrel a una fulla. L'*alçària* d'un arbre és la longitud del camí més llarg. Podem veure representats gràficament els conceptes anteriors a la figura 4.1. Consulteu-la abans de continuar la lectura.

Hi ha diferents tipus d'arbres. En primer lloc tenim l'*arbre general*. Es diu així perquè cada subarbre no buit pot tenir un nombre indeterminat, fins i tot zero, de fills no buits, i cap fill buit. Si un subarbre no buit no té cap fill, aquest subarbre consta d'un únic node que és una fulla de l'arbre que conté el subarbre. Un arbre general pot ser un arbre buit o un arbre no buit que no conté arbres buits com a subarbres.

Un altre tipus d'arbre és l'*arbre n-ari* on tots els subarbres no buits tenen exactament el mateix nombre de fills, siguin aquests fills arbres buits o no. El valor $n$ seria un paràmetre de l'operació constructora de la classe i podríem declarar arbres $n$-aris de diferents aritats.

En aquest capítol tractarem sobre un tipus concret d'arbre, l'*arbre binari*. Sobre els arbres generals i $n$-aris es tractarà en capítols posteriors.

## 4.1 Especificació de la classe genèrica dels arbres binaris

Els arbres binaris són un cas particular dels arbre $n$-aris, on $n = 2$. A partir d'ara en aquest capítol quan parlem d'arbres, ens referirem a arbres binaris.

```
template <typename T> class BinTree {

// Tipus de modul: dades
// Descripció del tipus: Arbre genèric que o bé és buit o bé tot
// subarbre seu té exactament dos fills.
```

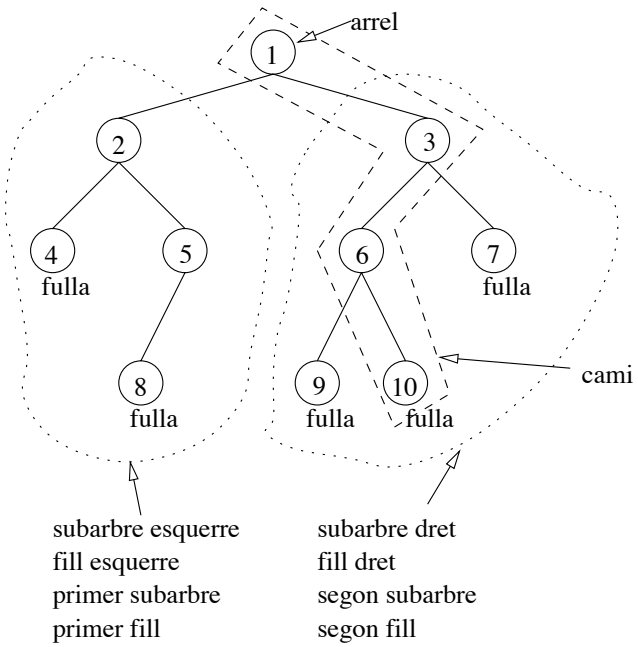


Figura 4.1: Termes comuns relatius a arbres

```
public:

    BinTree ();
    /* Pre: cert */
    /* Post: El resultat és un arbre sense cap element */

    BinTree (const T& x);
    /* Pre:  cert */
    /* Post: El resultat té x com a arrel, i els seus fills esquerre
             i dret són buits */

    BinTree (const T& x, const BinTree& left, const BinTree& right);
    /* Pre:  cert */
    /* Post: El paràmetre implícit té x com a arrel, left com a fill esquerre
             i right com a fill dret */

    bool empty () const;
    /* Pre: cert */
    /* Post: El resultat indica si el paràmetre implícit és buit o no */
```

```
    BinTree left () const;
    /* Pre: El paràmetre implícit no és buit */
    /* Post: El resultat és el fill esquerre del p.i. */

    BinTree right () const;
    /* Pre: El paràmetre implícit no és buit */
    /* Post: El resultat és el fill dret del p.i. */

    const T& value () const;
    /* Pre: El paràmetre implícit no és buit */
    /* Post: El resultat és l'arrel del paràmetre implícit */

};
```

El cost de totes aquestes operacions és constant. A diferència d'altres classes que hem vist, l'assignació i la constructora copiadora també tenen cost constant. S'ha de tenir en compte que, en aquesta implementació, aquestes dues operacions produeixen arbres amb nodes compartits (fan el que es diu *shallow copy*). Això no representa cap problema en aquest context, donat que els nodes no es poden modificar, només es poden crear de nous mitjançant constructores. Per això, quan vulguem modificar un arbre en realitat el reconstruirem amb nous nodes, per tant no s'afectaran els altres arbres amb els quals podria compartir nodes originalment.

L'operació destructora és l'única que no té cost constant perquè ha d'alliberar la memòria de tots els nodes d'un arbre (en realitat, només dels no compartits) i això evidentment té cost lineal al cas pitjor. Per últim, es podria haver afegit una operàció per fer una *deep copy* que també tindria cost lineal.

A la figura 4.1 veiem que l'arrel de l'arbre és 1, i té dos fills. El subarbre esquerre té com a arrel 2 i el subarbre dret té com a arrel 3. Tots els nodes tenen dos fills. Si els dos fills són arbres buits, els nodes es diuen fulles. Els nodes que contenen 4, 8, 9, 10 i 7 són fulles. Els arbres buits no estan representats.

## 4.2 Ús d'arbres binaris: operacions de cerca i recorregut

Un detall important és que quan només volem explorar un arbre, per exemple per fer una cerca o un recorregut; i no volem modificar-lo, ni canviant el valor d'un node ni canviant la seva estructura, llavors no hem de fer servir les constructores `BinTree`. Si l'arbre s'ha de modificar, llavors serà imprescindible usar aquestes constructores.

### 4.2.1 Alçària d'un arbre

Un senzill exemple d'exploració d'un arbre. Només es calcula una propietat de la estructura de l'arbre, no es consulta en cap moment el contingut dels nodes, és a dir, no es fa servir l'operació `value`.

```
int length(const BinTree<int>& a)
/* Pre: cert */
/* Post: El resultat és la longitud del camí més llarg de l'arrel a una fulla
   de l'arbre a */
{
  int x;
  if (a.empty()) x=0;
  else {
    int y=length(a.left());
    int z=length(a.right());
    if (y>=z) x=y+1;
    else x=z+1;
  }
  return x;
}
```

### 4.2.2 Mida d'un arbre

Un altre exemple molt similar.

```
int size(const BinTree<int>& a)
/* Pre: cert */
/* Post: El resultat és el nombre de nodes de l'arbre a */
{
  int x;
  if (a.empty()) x=0;
  else x=size(a.left())+size(a.right())+1;
  return x;
}
```

### 4.2.3 Cerca d'un valor `int` en un arbre de `int`

Un exemple una mica diferent, perquè tot i que també és tracta d'explorar l'arbre, no necessàriament visitarem tots els nodes de l'arbre. A més, també cal consultar el contingut dels nodes que visitem.

```
bool find(const BinTree<int>& a, int n)
/* Pre: cert */
/* Post: El resultat indica si n és a l'arbre a */
{
  bool b;
  if (a.empty()) b=false;
  else if (a.value()==n) b=true;
  else {
    b=find(a.left(),n);
```

```
    if (not b) b=find(a.right(),n);
  }
  return b;
}
```

### 4.2.4 Modificació i generació d'arbres

Veurem com en aquest tipus de situacions sí que hem de fer ser servir les constructores de la classe, com hem dit abans. Sumarem un valor `k` a tots els nodes d'un arbre d'`int`, en dues versions.

Primer oferim la versió funció. L'arbre resultat s'ha d'anar generant. Noteu com construïm i retornem arbres sense passar per variables auxilars.

```
BinTree<int> add(int k, const BinTree<int>& a) {
/* Pre: cert */
/* Post: El valor de cada node del resultat és la suma del valor del node
   corresponent d'a i el valor k */
    if (a.empty()) {
        return BinTree<int>();
    } else {
        return BinTree<int>(a.value() + k, add(k, a.left()), add(k, a.right()));
    }
}
```

A la versió acció, en el cas base, quan l'arbre és buit, l'arbre modificat és ell mateix i no cal fer res, per aquest motiu només hi ha el cas recursiu. Noteu com modifiquem l'arbre original amb l'única eina que tenim, asignant-li una crida a una constructora.

```
void addVoid(int k, BinTree<int>& a) {
/* Pre: a=A */
/* Post: El valor de cada node d'a és la suma del valor del node corresponent d'A
   i el valor k */
    if (not a.empty()) {
        BinTree<int> a1 = a.left();
        BinTree<int> a2 = a.right();
        addVoid(k, a1);
        addVoid(k, a2);
        a = BinTree<int>(a.value() + k, a1, a2);
    }
}
```

### 4.2.5 Recorreguts típics d'arbres

En aquesta secció explicarem els mètodes més habituals per visitar els nodes d'un arbre, amb intenció de fer recorreguts o cerques. Quan es visita un node es fa alguna cosa amb ell, com

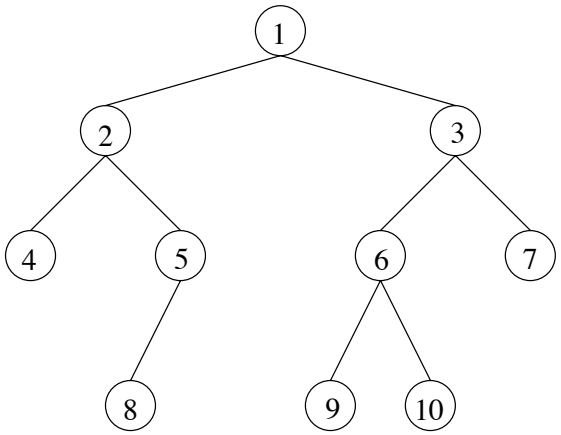


Figura 4.2: Exemple d'un arbre binari

modificar-lo, calcular la seva aportació a una funció de l'arbre, etc. El que diferencia els diversos mètodes és l'*ordre* de visita als nodes.

Hi ha dos tipus bàsics de recorreguts, els *recorreguts en profunditat* i els *recorreguts en amplada*.

**Recorreguts en profunditat**

Com que un arbre buit no té nodes, qualsevol recorregut no hi farà res. Per una altra part, els recorreguts en profunditat d'un arbre no buit es defineixen com:

- *preordre*:

  1. visitar l'arrel
  2. recórrer l'arbre esquerre (en preordre)
  3. recórrer l'arbre dret (en preordre)

  El recorregut en preordre de l'arbre de la figura 4.2 és 1, 2, 4, 5, 8, 3, 6, 9, 10 i 7.

- *inordre*:

  1. recórrer l'arbre esquerre (en inordre)
  2. visitar l'arrel
  3. recórrer l'arbre dret (en inordre)

  El recorregut en inordre de l'arbre de la figura 4.2 és 4, 2, 8, 5, 1, 9, 6, 10, 3, i 7.

- *postordre*:

1. recórrer l'arbre esquerre (en postordre)
2. recórrer l'arbre dret (en postordre)
3. visitar l'arrel

El recorregut en postordre de l'arbre de la figura 4.2 és 4, 8, 5, 2, 9, 10, 6, 7, 3, i 1.

Noteu que els algoritmes presentats fins ara en aquest capítol es poden classificar en algun d'aquests tipus, encara que en alguns d'ells l'ordre de les visites no és rellevant.

Com a exemple més general, mostrem funcions que transformen arbres d'enters a llistes d'enters corresponents als recorreguts en preordre, inordre i postordre. És evident que recórrer un arbre buit produeix el mateix resultat en tots els casos, una llista buida.

En el cas del preordre, cal afegir l'arrel davant de la llista resultat de recórrer l'arbre esquerre. Per últim fem servir el mètode `splice` de la classe `list` per concatenar-hi la llista resultat de recórrer l'arbre dret.

```
list<int> preorder(const BinTree<int>& a)
/* Pre: cert */
/* Post: El resultat conté els nodes d'a en preordre */
{
  list<int> l;
  if (not a.empty()) {
    l=preorder(a.left());
    l.insert(l.begin(),a.value());
    l.splice(l.end(),preorder(a.right()));
  }
return l;
}
```

Observem que la instrucció `l=preordre(a1);` fa que s'hagin de copiar llistes, cosa que fa que aquesta funció sigui menys eficient que una acció equivalent. No ens hem de preocupar de l'eficiència de `splice` perquè funciona en temps constant quan hi ha dues llistes com en aquest cas, i un iterador, `l.end()`.

En comptes de donar també la versió acció del recorregut en preordre, donarem directament la versió acció del recorregut en inordre.

En el cas de l'inordre afegim l'arrel al final de la llista resultat de recórrer l'arbre esquerre i després afegim la llista resultat de recórrer l'arbre dret.

```
void inorder(const BinTree<int>& a, list<int>& l)
/* Pre:  l=L */
/* Post: l conté L seguit dels nodes d'a en inordre */
{
  if (not a.empty()){
    inorder(a.left(),l);
    l.insert(l.end(),a.value());
```

```
    inorder(a.right(),l);
  }
}
```

Com podeu observar al codi anterior, no cal ni fer còpies de llistes a cada crida recursiva, que a més són costoses, ni tampoc cal fer servir l'`splice`. A més, noteu que la primera crida a l'operació ha de ser amb una llista buida.

Al postordre primer concatenem les llistes resultat de recórrer respectivament el subarbre esquerre i el subarbre dret i posteriorment afegim l'arrel de l'arbre al final de la llista resultant. Donem només la versió acció. Els comentaris finals de l'anterior exemple també s'apliquen aquí.

```
void postorder(const BinTree<int>& a, list<int>& l)
/* Pre:  l=L */
/* Post: l conté L seguit dels nodes d'a en postordre */
{
  if (not a.empty()){
    postorder(a.left(),l);
    postorder(a.right(),l);
    l.insert(l.end(),a.value());
  }
}
```

### Recorreguts en amplada

Un altre tipus de recorregut típic és el *recorregut per nivells* o *recorregut en amplada*. Donat un arbre, diem que el *nivell* d'un node és la distància a l'arrel. Mostrem una versió d'aquest recorregut on escrivim els nodes en una llista: primer s'escriu l'arrel de l'arbre (és a dir, el node de nivell 0), després tots els nodes que estan al nivell 1, d'esquerra a dreta, després tots els nodes que estan a nivell 2, també d'esquerra a dreta, etc. Aquest recorregut per nivells aplicat a l'arbre de la figura 4.2 és 1, 2, 3, 4, 5, 6, 7, 8, 9, i 10.

```
void breadth_first_traversal(const BinTree<int>& a, list<int>& l)
/* Pre:  l és buida */
/* Post: l conté els nodes d'a en ordre creixent respecte al nivell on es troben,
i els de cada nivell en ordre d'esquerra a dreta */
{
  if(not a.empty()){
    queue <BinTree<int> > c;
    c.push(a);
    while(not c.empty()){
      BinTree<int> aux(c.front());
      l.insert(l.end(),aux.value());
      if (not aux.left().empty()) c.push(aux.left());
      if (not aux.right().empty()) c.push(aux.right());
```

```
      c.pop();
    }
  }
}
```

Aquest algorisme és molt diferent dels anteriors. Per començar és iteratiu en comptes de recursiu, i requereix una estructura de dades auxiliar de tipus `queue` per guardar, en ordre correcte, els subarbres no buits amb elements que encara no s'han afegit a la llista resultat.

Per declarar la cua s'han de separar amb un espai els dos caràcters > darrera d'`int`, com al cas de les matrius. Fixeu-vos que després d'inserir l'element `node` amb l'iterador `it` que val `l.end()` no cal tocar l'iterador, que segueix senyalant `l.end()`.

Aquest algorisme te cost lineal respecte al nombre de nodes de l'arbre. S'ha de consultar cada node però totes les operacions que s'apliquen als arbres són de cost constant.