

# Tipus recursius de dades, III

Ricard Gavalrà

Programació 2

Facultat d'Informàtica de Barcelona, UPC

Primavera 2018

Aquesta presentació no substitueix els apunts

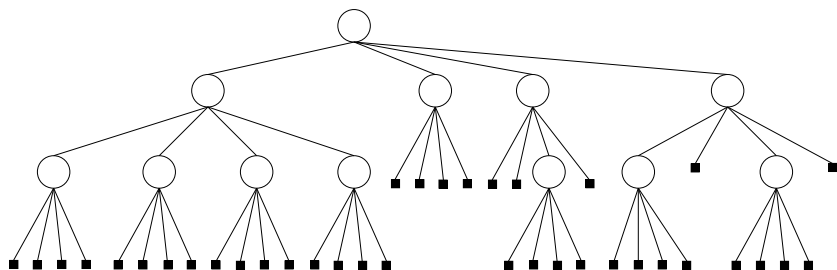
# Contingut

Arbres  $N$ -aris

Arbres generals

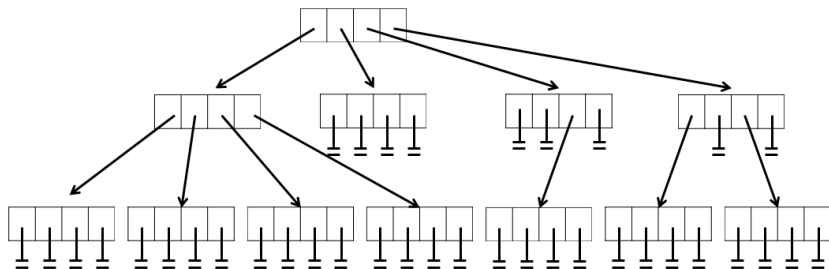
Arbres  $N$ -aris

## Arbres $N$ -aris



- Generalització dels arbres binaris
- $N$ : nombre de fills (binaris:  $N=2$ )

## Arbres $N$ -aris: Implementació



un node conté vector amb  $N$  apuntadors a node, un per a cada fill  
operació “consultar  $i$ -èssim” eficient: accés directe

## Definició classe ArbreNari

```
template <class T> class ArbreNari {
private:
    struct node_arbreNari {
        T info;
        vector<node_arbreNari*> seg;
    };
    int N;    // nombre de fills de cada subarbre
    node_arbreNari* primer_node;
    ... // operacions privades
public:
    ... // operacions públiques
};
```

## Copiar 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 */  
  
    if (m == NULL) return NULL;  
    else {  
        node_arbreNari* 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;  
    }  
}
```

Conté **recursió i iteració**

## Esborrar jerarquies de nodes

```
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;  
    }  
}
```



## Constructores/destructores I

```
ArbreNari(int n) {  
    /* Pre: cert */  
    /* Post: el p.i. és un arbre buit d'aritat n */  
    N = n;  
    primer_node = NULL;  
}
```

```
ArbreNari(const T &x, int n) {  
    /* Pre: cert */  
    /* Post: el p.i. és un arbre 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;  
}
```

## Constructores/destructores II

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

## Modificadores I

```
/* Pre: el p.i. té la mateixa aritatat que original */
/* Post: el p.i. és una còpia d'original */
ArbreNari& operator=(const ArbreNari& original) {
    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 buit de la mateixa aritatat que tenia */
    esborra_node_arbreNari(primer_node);
    primer_node = NULL;
}
```

## Modificadores II

```
void plantar(const T &x, vector<ArbreNari> &v) {  
    /* Pre:  el p.i. és buit, v = V, v.size() és l'aritat del p.i.  
            tots els components de v tenen la mateixa aritat que el p.i.,  
            i cap és el mateix objecte que el p.i. */  
    /* Post: el p.i. té x com a arrel i els seus fills són iguals  
            que els components de V; v conté arbres buits */  
    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] = v[i].primer_node;  
        v[i].primer_node = NULL;  
    }  
}
```

## Modificadores III

```
void fill(const ArbreNari &a, int i) {
/* Pre: el p.i. és buit i de la mateixa aritat que a,
   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_arbreNari((a.primer_node)->seg[i-1]);
}

void fills(vector<ArbreNari> &v) {
/* Pre:  el p.i. és A, un arbre no buit, v és un vector buit */
/* Post: v conté els fills d'A i el p.i. és buit */
    v = vector<ArbreNari>(N,ArbreNari(N));
    for (int i = 0; i < N; ++i)
        v[i].primer_node = primer_node->seg[i];
    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;  
}  
  
int aritat() const {  
    /* Pre: cert */  
    /* Post: el resultat és l'aritat del p.i. */  
    return N;  
}
```

## Eficiència de recorreguts arbres $N$ -aris

Observació: `fills` ens permet recorreguts eficients

- ▶ `fills` té cost  $N$ , siguin els subarbres molt grans o molt petits
- ▶ No hi ha còpia d'arbres

Podria fer-se copiant cada fill amb `fill`, però és ineficient

## Suma de tots elements d'un arbre

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

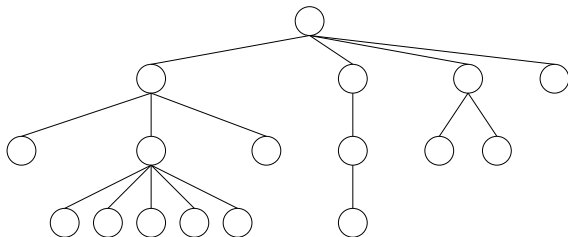


## Sumar un valor $k$ a cada node d'un arbre

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

## Arbres generals

# Arbres generals I

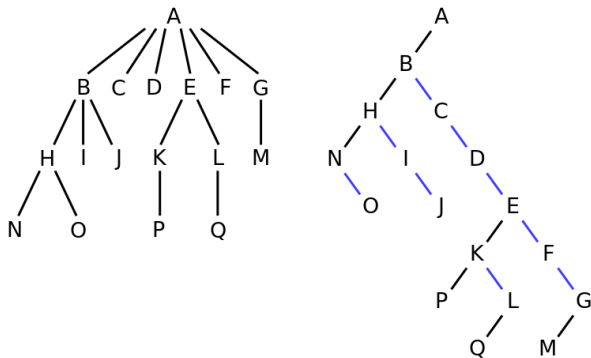


- ▶ Nombre indeterminat de fills, no necessàriament el mateix a cada subarbre
- ▶ Propietat important: Un arbre general
  - ▶ o és l'arbre buit
  - ▶ o té qualsevol nombre (fins i tot zero) de fills, cap dels quals és buit

# Arbres generals II. Implementacions

1. vector d'apuntadors de mida = nombre de fills
  - ▶ “consultar  $i$ -èssim” eficient
  - ▶ “eliminar fill  $i$ -èssim” potser és ineficient
  - ▶ (que no existeix en arbres  $N$ -aris!)
  - ▶ és la implementació que descriurem
2. *llista* d'apuntadors a fills
  - ▶ “consultar  $i$ -èssim” ineficient (accés seqüencial)
  - ▶ però ok per recorreguts seqüencials
  - ▶ ‘eliminar fill actual’ eficient
3. arbre binari “primer fill, germà dret”
  - ▶ reimplementació sobre arbres binaris

## “primer fill, germà dret”: exemple



(font: [http://en.wikipedia.org/wiki/Left-child\\_right-sibling\\_binary\\_tree](http://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree))

## Definició de la classe ArbreGen

```
template <class T> class ArbreGen
{
private:
    struct node_arbreGen {
        T info;
        vector<node_arbreGen*> seg;
    };
    node_arbreGen* primer_node;
    ... // operacions privades
public:
    ... // operacions públiques
};
```

Important: Ja no tenim un atribut amb el nombre de fills per a tot l'arbre; ni per a cada node. Es pot obtenir amb `seg.size()`

## Copiar i esborrar jerarquies de nodes

Idèntiques a les dels arbres  $N$ -aris (només canviar tipus dels nodes)

## Constructores/destructores I

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

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



## Constructores/destructores II

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

# Modificadores I

```
ArbreGen& operator=(const ArbreGen& original) {  
    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 */  
    esborra_node_arbreGen(primer_node);  
    primer_node = NULL;  
}
```

## Modificadores II

```
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;
    // inclou un primer_node->seg = vector<node_arbreGen*>(0);
    primer_node->info = x;

}

void plantar(const T &x, vector<ArbreGen> &v) {
/* Pre: el p.i. és buit, v = V, cap component de v és un arbre buit */
/* Post: el p.i. té x com a arrel i els elements de V
        com a fills; v conté només arbres buits */
    primer_node = new node_arbreGen;
    primer_node->info = x;
    int n = v.size();
    primer_node->seg = vector<node_arbreGen*>(n);
    for (int i = 0; i < n; ++i) {
        primer_node->seg[i] = v[i].primer_node;
        v[i].primer_node = NULL;
    }
}
```

## Modificadores III

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

**Nota:** primer ús de `vector.push_back(...)`

## Modificadores IV

```
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, li diem A, i no és cap dels components de
/* Post: el p.i. és buit, v passa a contenir els fills de l'arbre A */
    int n = primer_node->seg.size();
    v = vector<ArbreGen>(n);
    for (int i = 0; i < n; ++i) v[i].primer_node = primer_node->seg[i];
    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;  
}  
  
int nombre_fillls() const {  
    /* Pre: el p.i. no és buit */  
    /* Post: el resultat és el nombre de fillls del p.i. */  
    return (primer_node->seg).size();  
}
```

## Exemple: suma de tots els elements

```
int suma(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(v[i]);  
    }  
    return s;  
}
```

## Exemple: sumar $k$ a cada element

```
void suma_k(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,vector<int>(0));  
        else {  
            for (int i = 0; i < n; ++i) suma_k(v[i], k);  
            a.plantar(s, v);  
        }  
    }  
}
```