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

Professorat de PRO2

Tardor 2022

- Col·laboracions (en ordre alfabètic): Juan Luis Esteban, Ricard Gavaldà, Conrado Martínez, Fernando Orejas

- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Part I

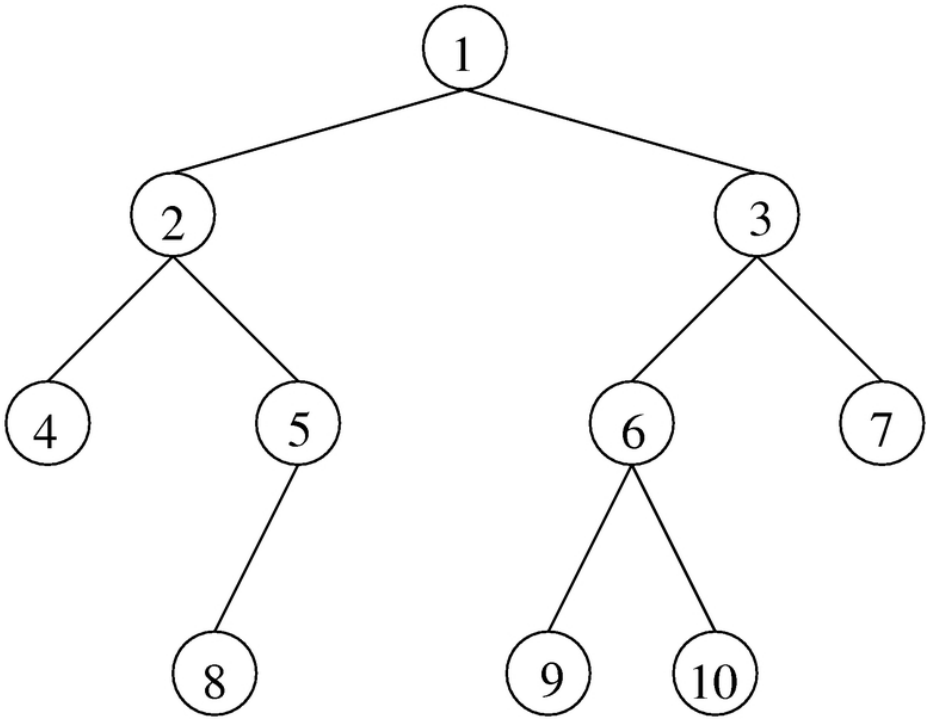
Arbres binaris (o simplement arbres)

Un arbre o bé és l'arbre buit
o bé és un node anomenat arrel amb zero, u o dos
arbres successors anomenats fills o subarbres

Es presta a tractaments algorísmics recursius

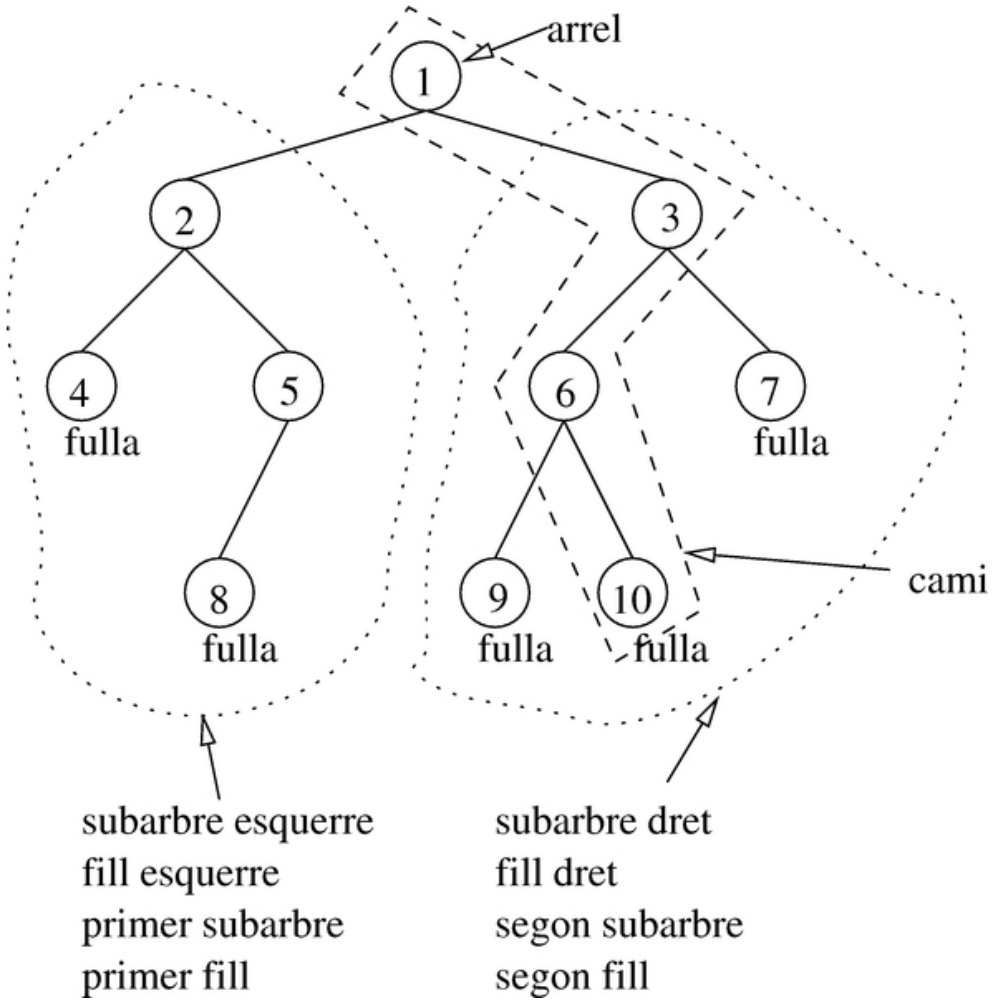
Els dos fills d'un node són anomenats esquerre i dret

Exemple d'arbre binari



No hem dibuixat els subarbres buits. La inclinació de cada aresta indica si el fill és dret o esquerre

Exemple d'arbre binari



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 `root` és nul

Els noms de camps, atributs i operacions poden variar en altres versions.

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

Constructores i destructora

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

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

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

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 apu
         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.root);
      esborra_node_arbre(root);
      root = aux;
    }
    return *this;
}

void a_buit() {
    esborra_node_arbre(root);
    root = 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.root;
    if (a2.root != a1.root or a2.root == nullptr)
        aux -> dre = a2.root;
    else
      aux -> dre = copia_node_arbre(a2.root);
    root = aux;
    a1.root = nullptr;
    a2.root = 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.root = root -> esq;
    fd.root = root -> dre;
    delete root;
    root = nullptr;
}
```

Consultores

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

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

Sumar un valor a tots els elements d'un arbre binari

El plantegem com a nou mètode de la classe arbre binari

```
    ...
/* Pre: A és el valor inicial del arbre implícit */
/* Post: l'arbre implícit és l'arbre A però havent sumat k
         a tots els seus elements */
void inc_arbre(const T& k);
    ...
```

Sumar un valor a tots els elements d'un arbre binari

```
/* Pre: A és el valor inicial del arbre implícit */
/* Post: l'arbre implícit és l'arbre A però havent sumat k
         a tots els seus elements */
void inc_arbre(const T& k) {
    inc_node(root, k);
}


/* Pre: cert */
/* Post: el node apuntat per n i tots els seus descendents tenen
         al camp info la suma de k i el seu valor original */
static void inc_node(node_arbre* n, int k) {
    if (n != nullptr) {
        n -> info += k;
        inc_node(n -> esq, k);
        inc_node(n -> dre, k);
    }
}
```

Substitució de fulles per un arbre I

Substituir totes les fulles de l'arbre implícit que continguin el valor `x` per un altre arbre donat `as`

```
/* Pre: A es el valor inicial del p.i. */
/* Post: l'arbre és com A però havent substituït
       les fulles que contenien x per l'arbre as */
void subst(const T& x, const ArbreBin<T>& as);
```

Substitució de fulles per un arbre II

```
void subst(const T& x, const ArbreBin<T>& as) {
    root = subst_node(root, x, as);
}


/* Pre: n apunta a l'arrel d'un (sub)arbre A */
/* Post: retorna un apuntador a l'arrel del subarbre resultant
de substituir cada fulla del subarbre amb arrel apuntada per n
que contingui el valor x per una còpia de l'ArbreBin as */
static node_arbre* subst_node(node_arbre* n, const T& x,
                              const ArbreBin<T>& as);
```

Substitució de fulles per un arbre III

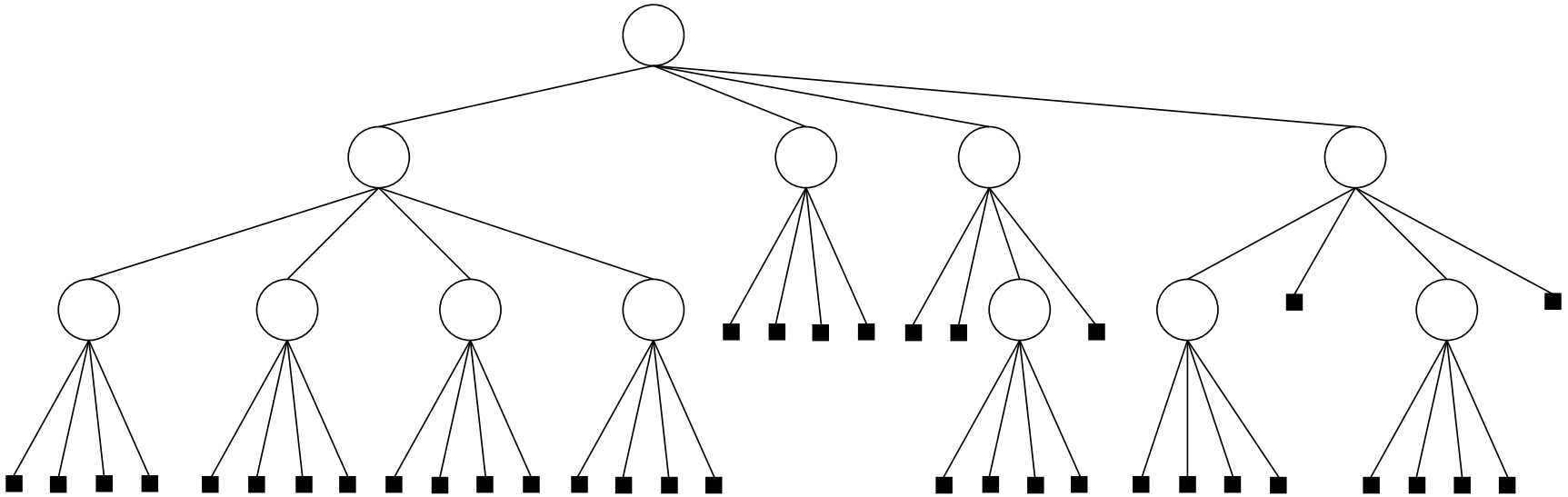
```
static node_arbre* subst_node(node_arbre* n, const T& x,
                              const ArbreBin<T>& as) {
    if (n == nullptr) return nullptr;

    // n != nullptr
    if (n -> info == x and
        n -> esq == nullptr and n -> dre == nullptr) {
      // n apunta a una fulla que conté el valor x
      delete n; // no cal fer esborra_node_arbre(n);
      n = copia_node_arbre(as.root);
    } else {
      n -> esq = subst_node(n -> esq, x, as);
      n -> dre = subst_node(n -> dre, x, as);
    }
    return n;
}
```

Atenció al retorn de l'apuntador a l'arrel de l'arbre resultant.
L'alternativa és passar `n` per referència

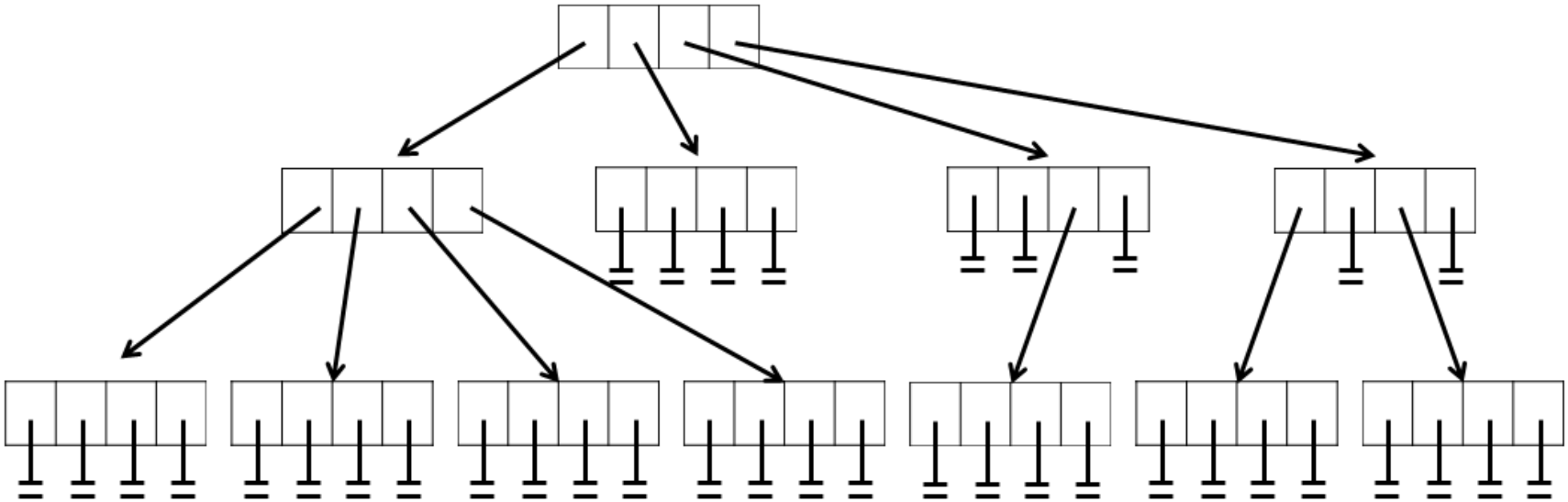
Part I

Arbres N-aris



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

Arbres N-aris: Implementació



un node conté vector anb N apuntadors a node, un per a cada fill

operació “consultar i-èssim” eficient: accés directe

Definició classe `ArbreNari`

Els noms de camps, atributs i operacions poden variar en altres versions.

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

Copiar jerarquies de nodes

```
static node_arbreNari* copia_node_arbreNari(node_arbreNari* 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 */

    if (m == nullptr) return nullptr;
    else {
        node_arbreNari* n = new node_arbreNari;
        n -> info = m -> info;
        int N = m -> child.size();
        n -> child = vector<node_arbreNari*>(N);
        for (int i = 0; i < N; ++i)
            n -> child[i] = copia_node_arbreNari(m -> child[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 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) {
        int N = m -> child.size();
        for (int i = 0; i < N; ++i)
            esborra_node_arbreNari(m -> child[i]);
        delete m;
    }
}
```

Constructores/destructora I

```
ArbreNari(int n) {
/* Pre: cert */
/* Post: l'arbre implícit és un arbre buit d'aritat n */
    N = n;
    root = nullptr;
}

ArbreNari(const T& x, int n) {
/* Pre: cert */
/* Post: l'arbre implícit és un arbre amb arrel x i
         n fills buits */
    N = n;
    root = new node_arbreNari;
    root -> info = x;
    root -> child = vector<node_arbreNari*>(N, nullptr);
}
```

Constructores/destructora II

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

~ArbreNari() {
    esborra_node_arbreNari(root);
}
```

Modificadores I

```
/* Pre: l'arbre implícit té la mateixa aritat que original */
/* Post: l'arbre implícit és una còpia d'original */
ArbreNari& operator=(const ArbreNari& original) {
  if (this != &original) {
    node_ArbreNari* aux = copia_node_arbreNari(original.root);
    esborra_node_arbreNari(root);
    root = aux;
    // No cal copiar l'aritat
  }
  return *this;
}
```

```
void a_buit() {
/* Pre: cert */
/* Post: l'arbre implícit és un arbre buit de la mateixa aritat
         que tenia */
    esborra_node_arbreNari(root);
    root = nullptr;
}
```

Modificadores II

```
void plantar(const T& x, vector<ArbreNari>& v) {
/* Pre:  l'arbre implícit és buit, v = V, v.size() és l'aritat
         de l'arbre implícit, tots els components de v tenen la
         mateixa aritat que l'arbre implícit, i tots són objectes
         diferents entre sí i diferents de l'arbre implícit */
/* Post: l'arbre implícit té x com a arrel i els seus fills són iguals
         que els components de V; v conté arbres buits */
    root = new node_arbreNari;
    root -> info = x;
    root -> child = vector<node_arbreNari*>(N);
    for (int i = 0; i < N; ++i) {
        root -> child[i] = v[i].root;
        v[i].root = nullptr;
    }
}
```

Modificadores III

```
void fill(const ArbreNari& a, int i) {
/* Pre: l'arbre implícit 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: l'arbre implícit és una còpia del fill i-èssim d'a */
   root = copia_node_arbreNari(a.root -> child[i-1]);
}

void fills(vector<ArbreNari>& v) {
/* Pre:  l'arbre implícit és A, un arbre no buit,
         v és un vector buit */
/* Post: v conté els fills d'A i l'arbre implícit és buit */
    v = vector<ArbreNari>(N, ArbreNari(N));
    for (int i = 0; i < N; ++i)
        v[i].root = root -> child[i];
    delete root;
    root = nullptr;
}
```

Consultores

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

bool es_buit() const {
/* Pre: cert */
/* Post: el resultat indica si l'arbre implícit és un arbre buit */
    return root == nullptr;
}

int aritat() const {
/* Pre: cert */
/* Post: el resultat és l'aritat de l'arbre implícit */
    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

Exemple d'us d'operacions primitives: 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;
    }
}
```

Exemple d'us d'operacions primitives: 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);
    }
}
```

Fer un nou arbre N-ari sumant k als nodes d'un altre

El plantegem com a nou mètode de la classe arbre N-ari

```
    ...
/* Pre: cert */
/* Post: a és com el p.i però havent sumat k
         a tots els seus elements */
void sumak(ArbreNari& a, const T& k) const;
    ...
```

Fer un nou arbre N-ari sumant k als nodes d'un altre

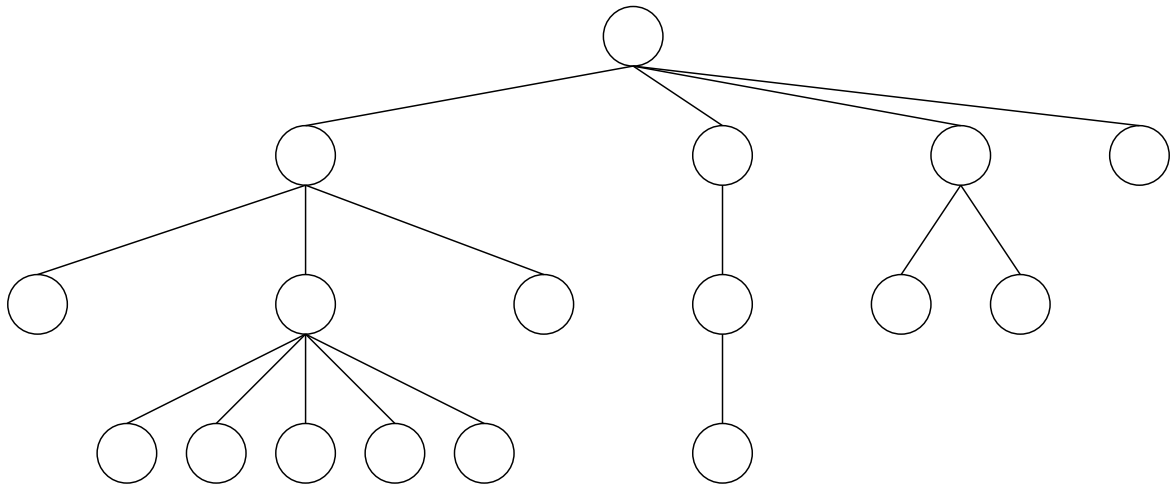
```
/* Pre: cert */
/* Post: el node apuntat per n i tots els seus descendents tenen
         al camp info la suma de k i el seu valor del seu node anàleg
         a partir de la sèrie de nodes que comencen a m */
static void sumak_rec(node_arbreNari* m, node_arbreNari* & n,
                      const T& k){
  if (m == nullptr) n = nullptr;
    else{
      n = new node_arbreNari;
      n->child = vector<node_arbreNari*> (m->child.size());
      for (int i = 0; i < m->seg.size(); i++){
        sumak_rec(m->child[i], n->child[i], k);
      }
      n->info = m->info + k;
  }
}
```

Fer un nou arbre N-ari sumant k als nodes d'un altre

```
/* Pre: cert */
/* Post: a és com el p.i però havent sumat k
         a tots els seus elements */
void sumak(ArbreNari& a, const T& k) const{
        esborra_node_arbreNari(a.root);
        sumak_rec(root, a.root, k);
        a.N = N; // potser a no tenia la mateixa aritat que el p.i.
}
```

Part I

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

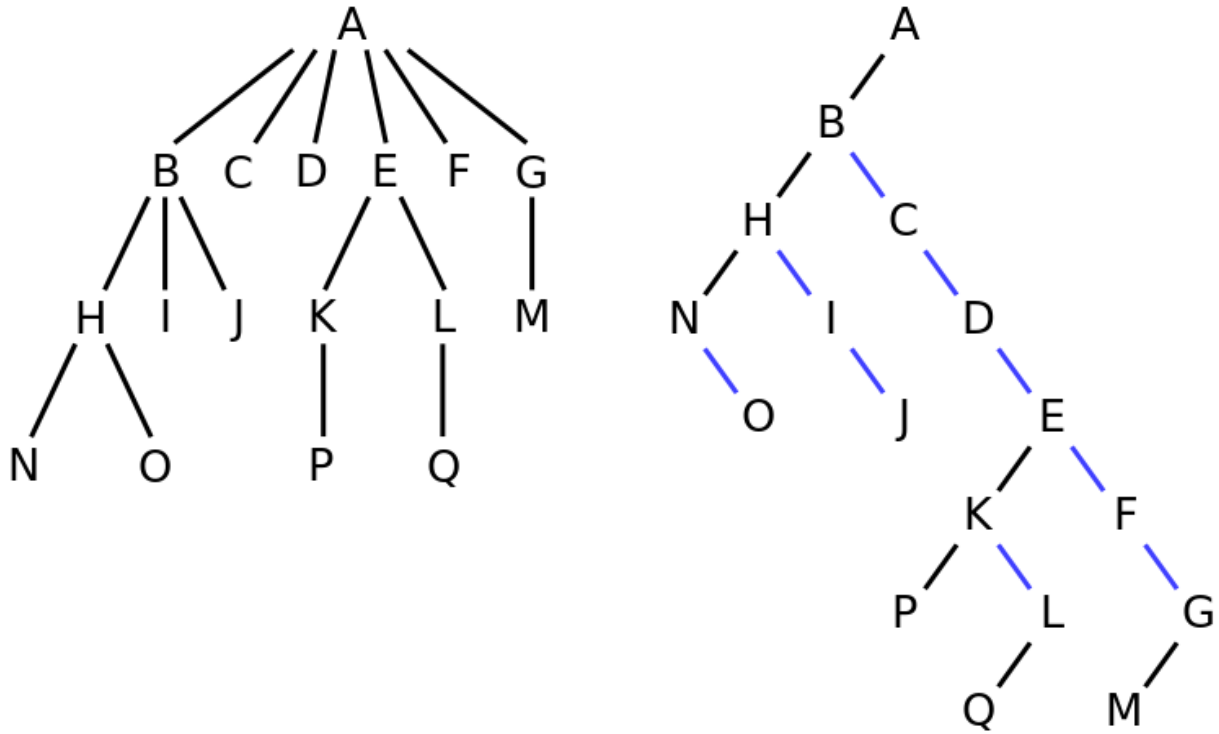
Arbres generals II. Implementacions

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

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, següent germà"
   - reimplementació sobre arbres binaris

“primer fill, següent germà”: exemple



(font: http://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree)

Definició de la classe `ArbreGen`

Els noms de camps, atributs i operacions poden variar en altres versions.

```
template <class T> class ArbreGen
    private:
        struct node_arbreGen {
            T info;
            vector<node_arbreGen*> child;
        };
        node_arbreGen* root;
        ...  // 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 `child.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: l'arbre implícit és un arbre general buit */
    root = nullptr;
}

ArbreGen(const T &x) {
/* Pre: cert */
/* Post: l'arbre implícit és un arbre general amb arrel x
         i 0 fills */
    root = new node_arbreGen;
    root -> info = x;
    // no cal fer arrel -> child = vector<node_arbreGen*>(0);
}
```

Constructores/destructores II

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

~ArbreGen() {
    esborra_node_arbreGen(root);
}
```

Modificadores I

```
ArbreGen& operator=(const ArbreGen& original) {
  if (this != &original) {
    node_ArbreGen* aux = copia_node_arbreGen(original.root);
    esborra_node_arbreGen(root);
    root = aux;
  }
  return *this;
}

void a_buit() {
/* Pre: cert */
/* Post: l'arbre implícit és un arbre general buit */
    esborra_node_arbreGen(root);
    root = nullptr;
}
```

Modificadores II

```
void plantar(const T &x) {
/* Pre: l'arbre implícit és buit */
/* Post: l'arbre implícit té x com a arrel i sense fills */
    root = new node_arbreGen;
       // inclou un root -> child = vector<node_arbreGen*>(0);
    root -> info = x;

}

void plantar(const T &x, vector<ArbreGen> &v) {
/* Pre:  l'arbre implícit és buit, v = V, cap component
         de v és un arbre buit */
/* Post: l'arbre implícit té x com a arrel i els elements de V
         com a fills; v conté només arbres buits */
    root = new node_arbreGen;
    root -> info = x;
    int n = v.size();
    root -> child = vector<node_arbreGen*>(n);
    for (int i = 0; i < n; ++i) {
        root -> child[i] = v[i].root;
        v[i].root = nullptr;
    }
}
```

Modificadores III

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

Nota: aquí necessitem fer `push_back(...)`

Modificadores IV

```
void fill(const ArbreGen& a, int i) {
/* Pre: l'arbre implícit és buit, a no és buit, i està entre 1 i
   el nombre de fills d'a */
/* Post: l'arbre implícit és una còpia del fill i-èssim d'a */
    root = copia_node_arbreGen(a.root -> child[i-1]);
}

void fills(vector<ArbreGen> &v) {
/* Pre:  l'arbre implícit no és buit, li diem A, i no és cap
         dels components de v*/
/* Post: l'arbre implícit és buit, v passa a contenir els fills
         de l'arbre A */
    int n = root -> child.size();
    v = vector<ArbreGen>(n);
    for (int i = 0; i < n; ++i) v[i].root = root -> child[i];
    delete root; root = nullptr;
}
```

Consultores

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

bool es_buit() const {
/* Pre: cert */
/* Post: el resultat indica si l'arbre implícit és un arbre buit */
    return root == nullptr;
}

int nombre_fills() const {
/* Pre: l'arbre implícit no és buit */
/* Post: el resultat és el nombre de fills de l'arbre implícit */
    return root -> child.size();
}
```

Exemple d'us d'operacions primitives: 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 d'us d'operacions primitives: 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();
        // si n == 0, el bucle no fa res i es planta v que és
        // un vector buit d'ArbreGen. és a dir, la nova arrel conté
        // a.arrel() + k, i no tindrà cap fill, com originalment
        for (int i = 0; i < n; ++i) suma_k(v[i], k);
        a.plantar(s, v);
    }
}
```

Màxim i mínim d'una arbre general

El plantegem com a nou mètode de la classe arbre general

```
    ...
/* Pre: el p.i no està buit */
/* Post: max conté el màxim del p.i.
         min conté el m'inim del p.i */
void max_min(T& max, T& min) const
    ...
```

Màxim i mínim d'una arbre general

```
/* Pre: el p.i no està buit */
/* Post: max conté el màxim del p.i.
         min conté el m'inim del p.i */
void max_min(T& max, T& min) const {
  max_min_rec(root, max, min);
}

/* Pre: m != nullptr */
/* Post: max conté el màxim dels nodes a partir d'm
         mix conté el mínim dels nodes a partir d'm   */
static void max_min_rec(node_arbreGen* m, T& max, T& min){
    max = min = m->info;
    int ari = m->child.size();
    for (int i=0; i<ari; ++i){
      T max_aux, min_aux;
      max_min_rec(m->child[i],max_aux,min_aux);
      if (max_aux > max) max = max_aux;
      if (min_aux < min) min = min_aux;
    }
}
```