Chapitre 7
Les arbres

Module Structures de données et programmation C

2éme ANNEE LEESM

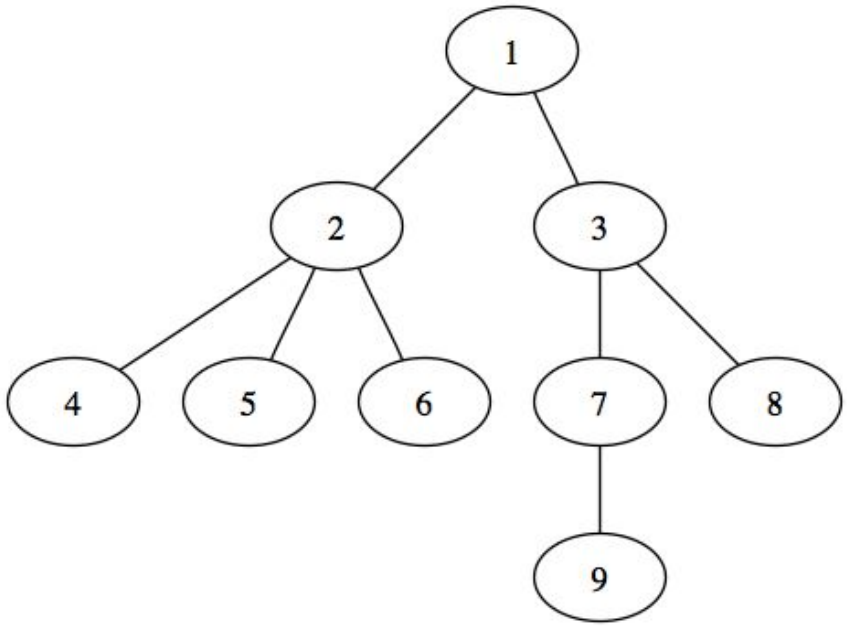
mlahby@gmail.com

26 mai 2021

Plan

Définition

- Un arbre (tree en anglais) est une structure de données dynamique et non linéaire.
- Un arbre = ensemble de sommets ou noeuds tel que :
  - $\exists$ un sommet unique appelé racine r qui n'a pas de supérieur
  - Tous les autres sommets sont atteints à partir de r par une branche unique
- Représentattion graphique d'un arbre :

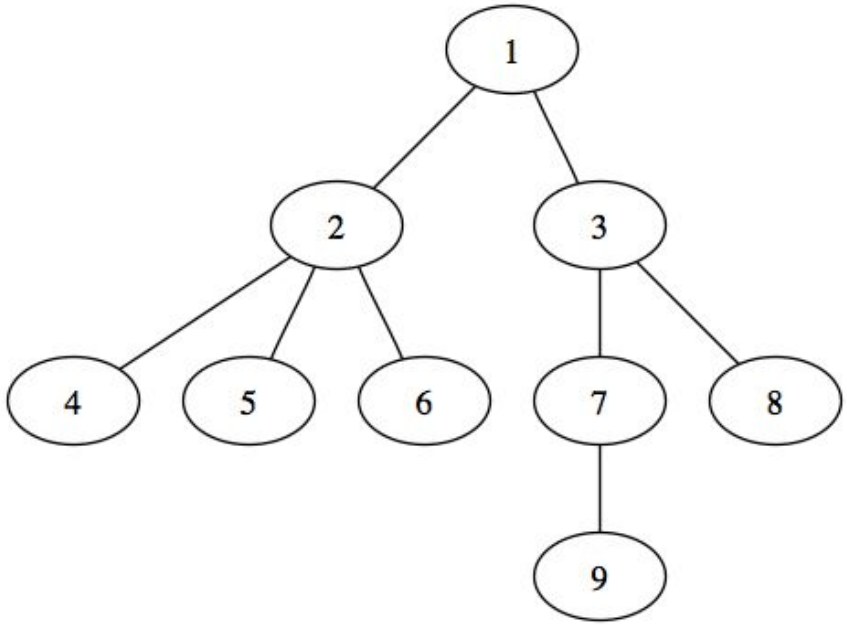


Terminologie et propriètés

- **noeud** = sommet
- **racine**= noeud sans père
- Une **feuille** est un sommet qui n'a pas de fils.
- **branche** : chemin entre 2 noeuds
- **père d'un sommet** : le prédécesseur d'un sommet
- **fils d'un sommet** : les successeurs d'un sommet
- **frères** : des sommets qui ont le même père
- Tout sommet x qui n'est pas la racine a :
  - un unique parent, noté parent(x) (appelé père parfois)
  - 0 ou plusieurs fils. fils(x) désigne l'ensemble des fils de x
- Si x et y sont des sommets tels que x soit sur le chemin de r à y alors
  - x est un ancêtre de y
  - y est un descendant de x

Terminologie et propriètés

⇒**Exemple :**



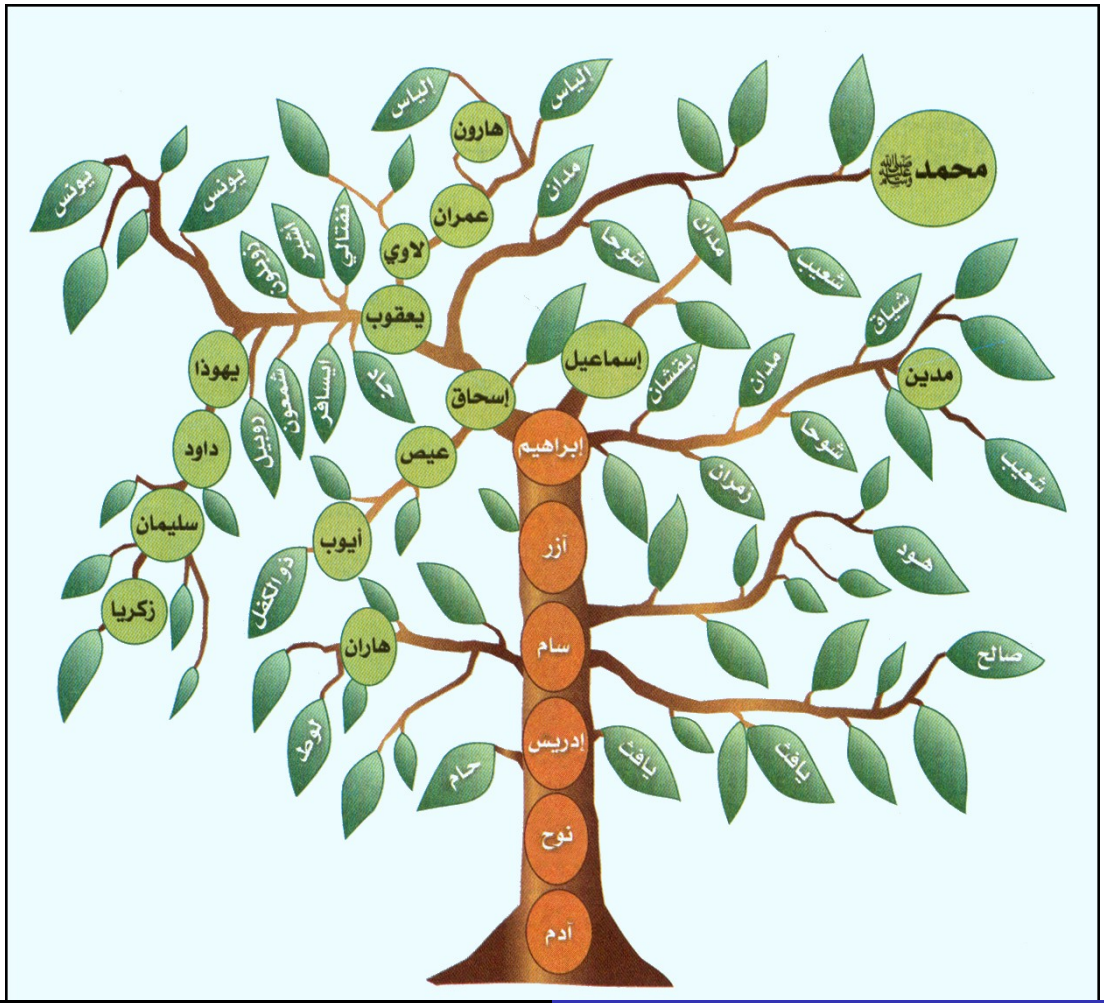
- 1 est la racine
- 4,5,6,9,8 sont les feuilles
- 9 est un descendant de 3, mais pas de 2
- 3 est un ancêtre de 9.

Applications

- Modèle pour les structures hiérachisées :
  - Arbre familial
  - Arbre généalogique
  - Structure de fichiers sous linux ou windows
  - Organigramme d'une entreprise
- Les expressions arithmétiques
- En linguistique et en compilation (arbres syntaxiques).
- Gérer des bases de données
- Utilisation pour des algorithmes de recherches rapides et efficaces (tri par tas)
- Algorithmique du texte (Huffamn, prefix-trees) : compression, recherche de motifs, détection de répétitions, ...
- Intelligence artificielle : arbres de décision
- Cryptoghraphie : algorithmes de recherche

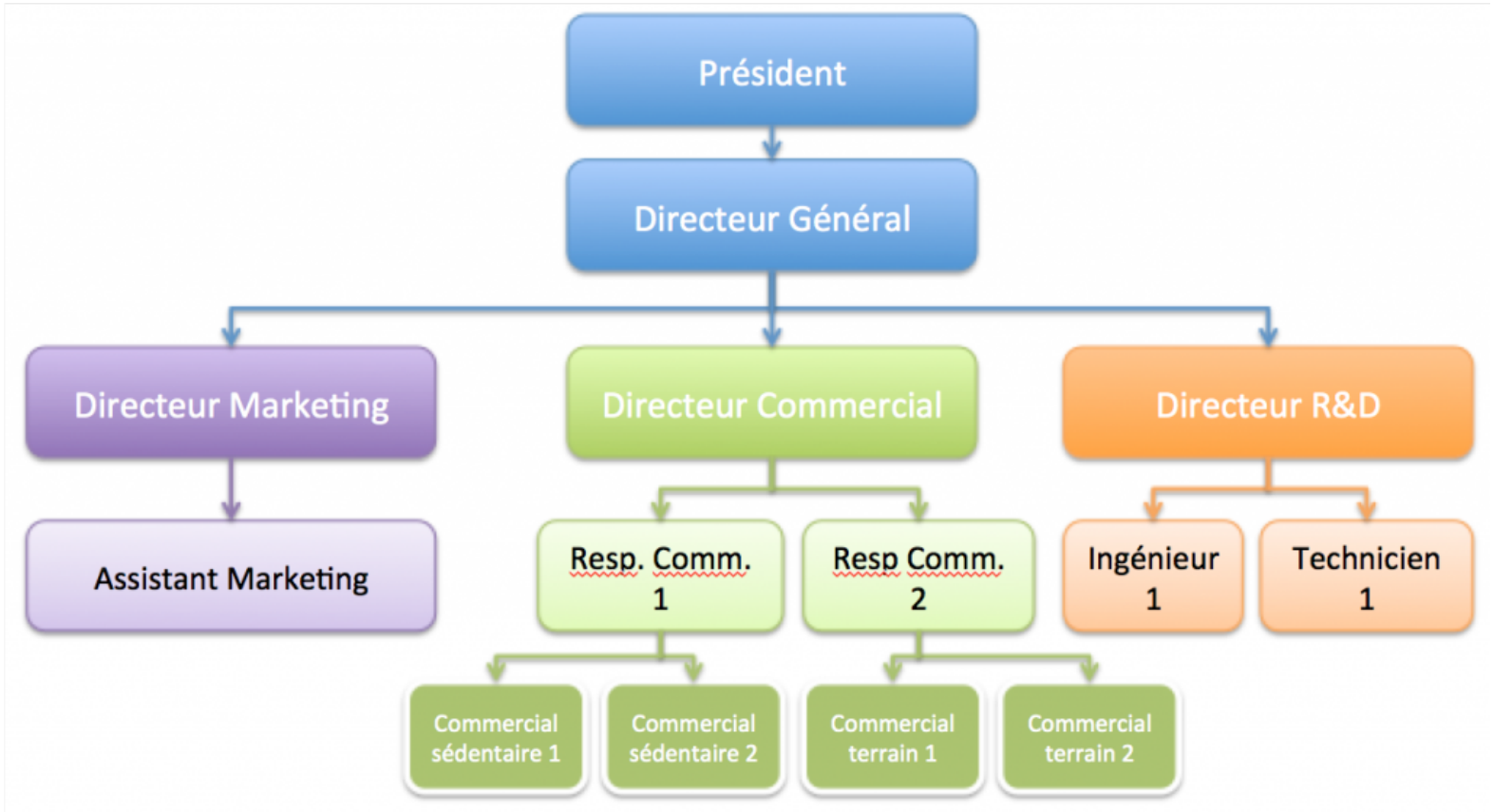
Applications

⇒ Exemple 1 : Arbre généalogique



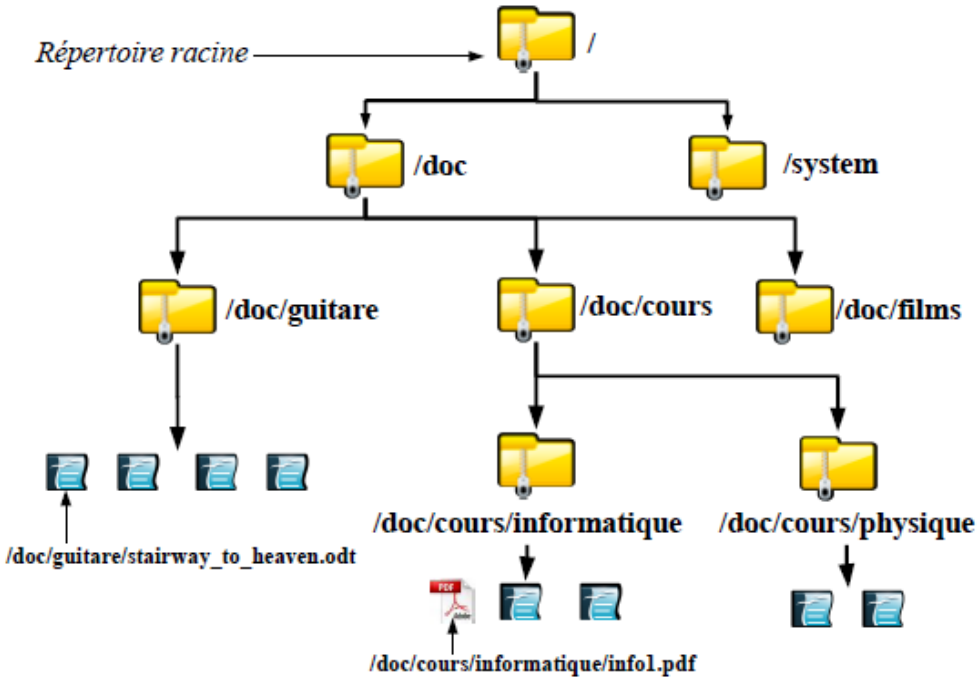
Applications

⇒ Exemple 2 : Organigramme d'entreprise



Applications

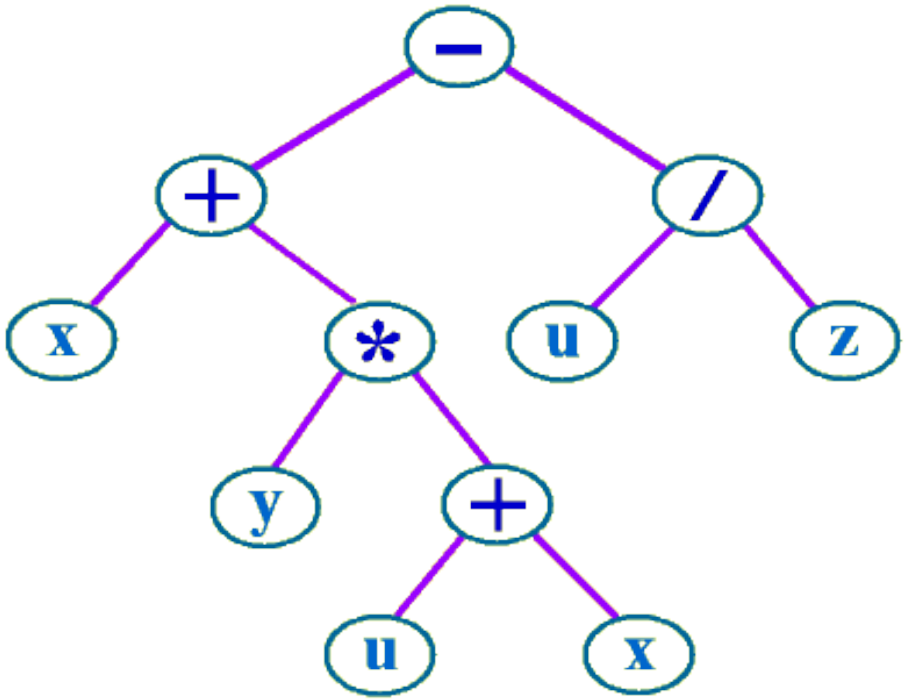
⇒ Exemple 3 : Arborescence des fichiers sous linux



Applications

⇒ Exemple 4 : Evaluation d'une expression algébrique

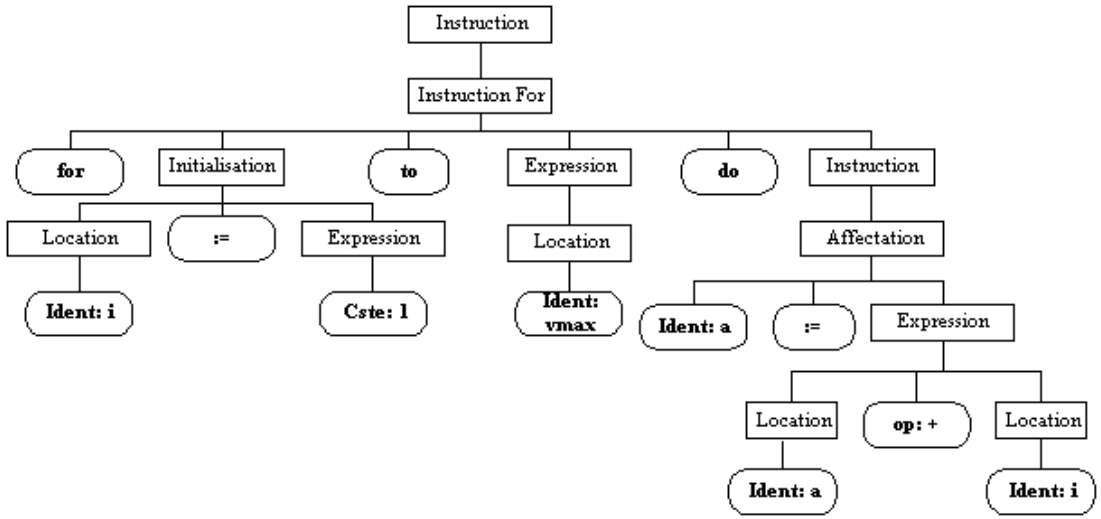
L'évaluation de l'expression arithmétique $x + y.(u + x) - u/z$ est représentée par l'arbre ci-dessous :



Applications

⇒ Exemple 5 : Arbre syntaxique

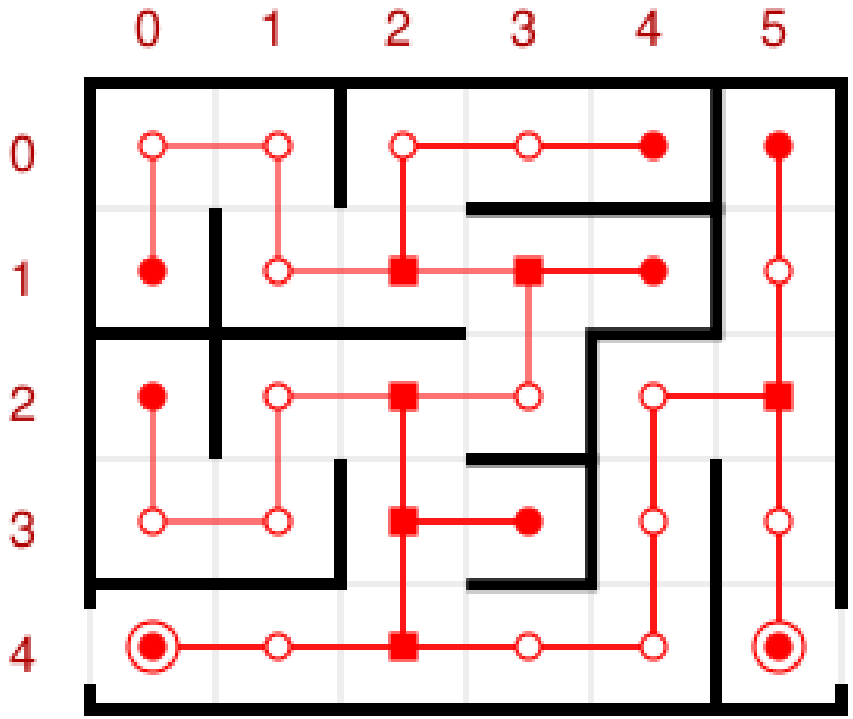
l'arbre suivant représente la structure de la phrase : for i :=1 to vmax do a :=a+i



Applications

⇒ Exemple 6 : Intelligence artificielle : arbres de décision

Modélisation et Création d'un Labyrinthe Rectangulaire en Deux Dimensions

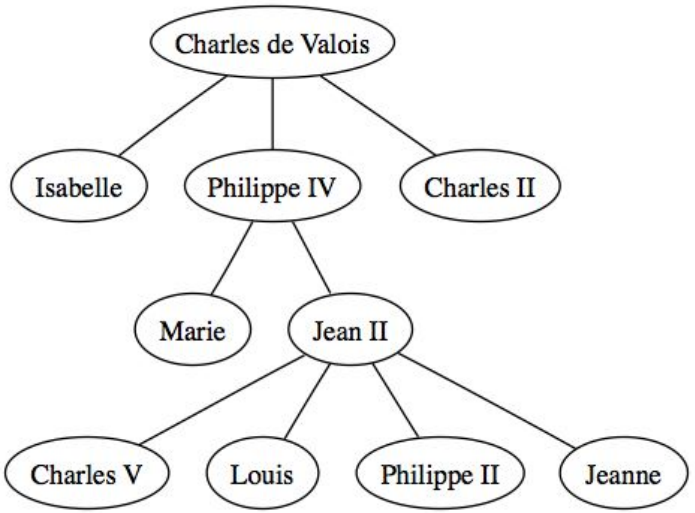


Mesures sur les arbres

Arité d'un arbre

- L'arité d'un arbre représente le nombre maximum de fils qu'il possède pour un sommet donné.
- Un arbre dont les noeuds ne comporteront qu'au maximum n fils sera d'arité n. On parlera alors d'arbre n-aire.
- Il existe un cas particulièrement utilisé : c'est **l'arbre binaire**.

⇒ **Exemple :** Voici un arbre généalogique dont l'arité est 4



Mesures sur les arbres

La profondeur d'un sommet

- La profondeur (niveau) d'un noeud est la longueur de la branche depuis la racine
- La profondeur d'un sommet est définie récursivement par :
  - prof(v) = 0 si v est la racine
  - prof(v) = prof(parent(v)) + 1

La hauteur d'un sommet

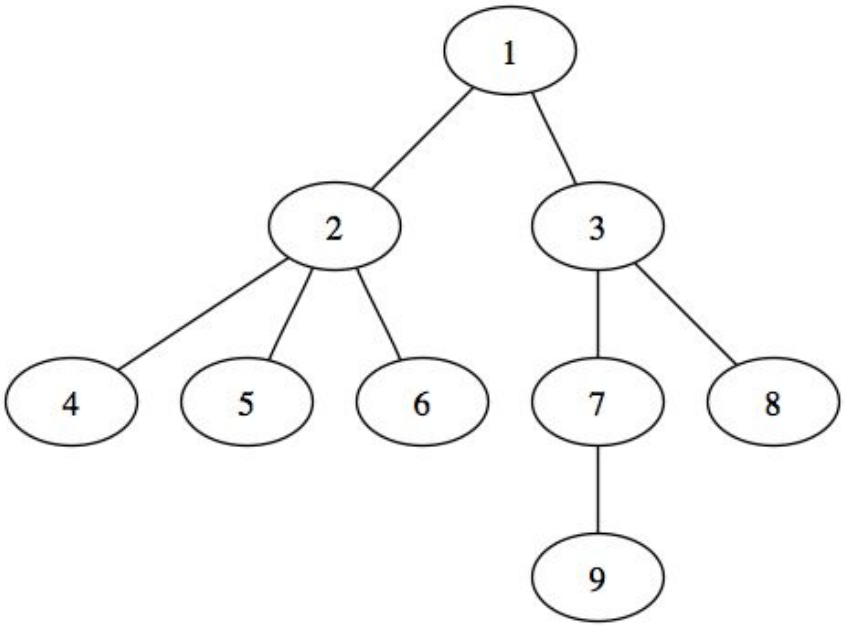
- La hauteur d'un sommet est la plus grande profondeur d'une feuille du sous-arbre dont il est la racine

Autres mesures

- La hauteur d'un arbre est la hauteur de sa racine
- La taille d'un arbre est le nombre de ses sommets
- Le degré d'un noeud est le nombre de fils que possède ce noeud.

Mesures sur les arbres

⇒**Exemple :**



- 2,3 sont à la profondeur 1
- 4,5,6,7,8 sont à la profondeur 2
- La hauteur de 3 est 2, celle de 6 est 0, celle de 3 est 0, celle de 1 est 4
- La hauteur de l'arbre est 5
- La taille de l'arbre est 9

Définition des arbres binaires

Définition informelle

- Dans un arbre binaire tout noeud a au plus deux fils
- Un arbre binaire possède exactement deux sous-arbres (éventuellement vides)

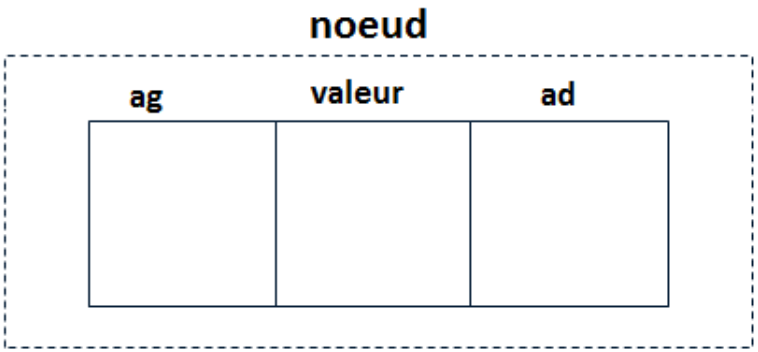
Définition récursive

- Soit vide
- Soit composé
  - d'une racine r
  - de 2 sous arbres binaires ABG et ABD disjoints
    * ABG : sous Arbre Binaire Gauche
    * ABD : sous Arbre Binaire Droit

La notion de noeud

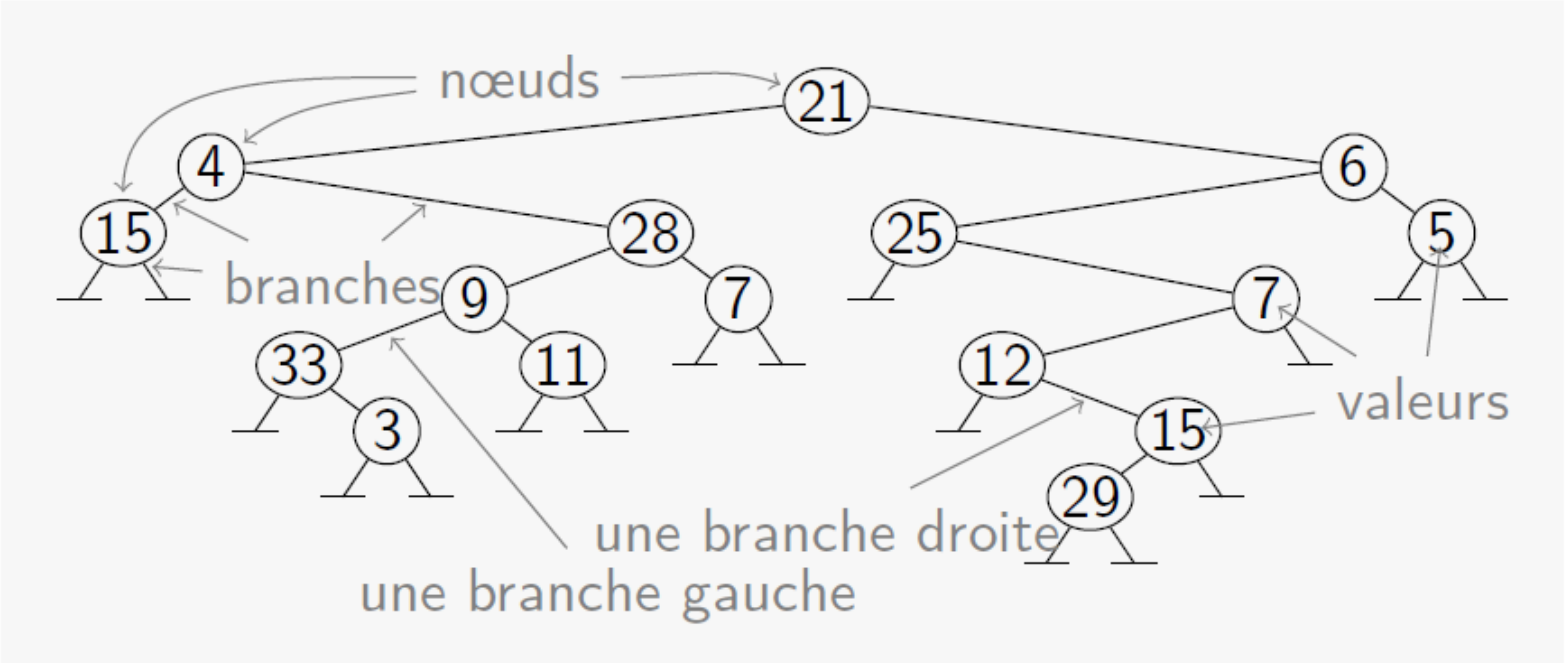
Définition

- Le noeud est le composant principal d'un arbre binaire
- Un noeud est une structure qui comporte trois champs :
  - **Le champ valeur** : il contient des informations sur l'élément représenté par la cellule ;
  - **Le champ ag** : il représente un pointeur qui contient l'adresse du sous-arbre gauche
  - **Le champ ad** : il représente un pointeur qui contient l'adresse du sous-arbre droit



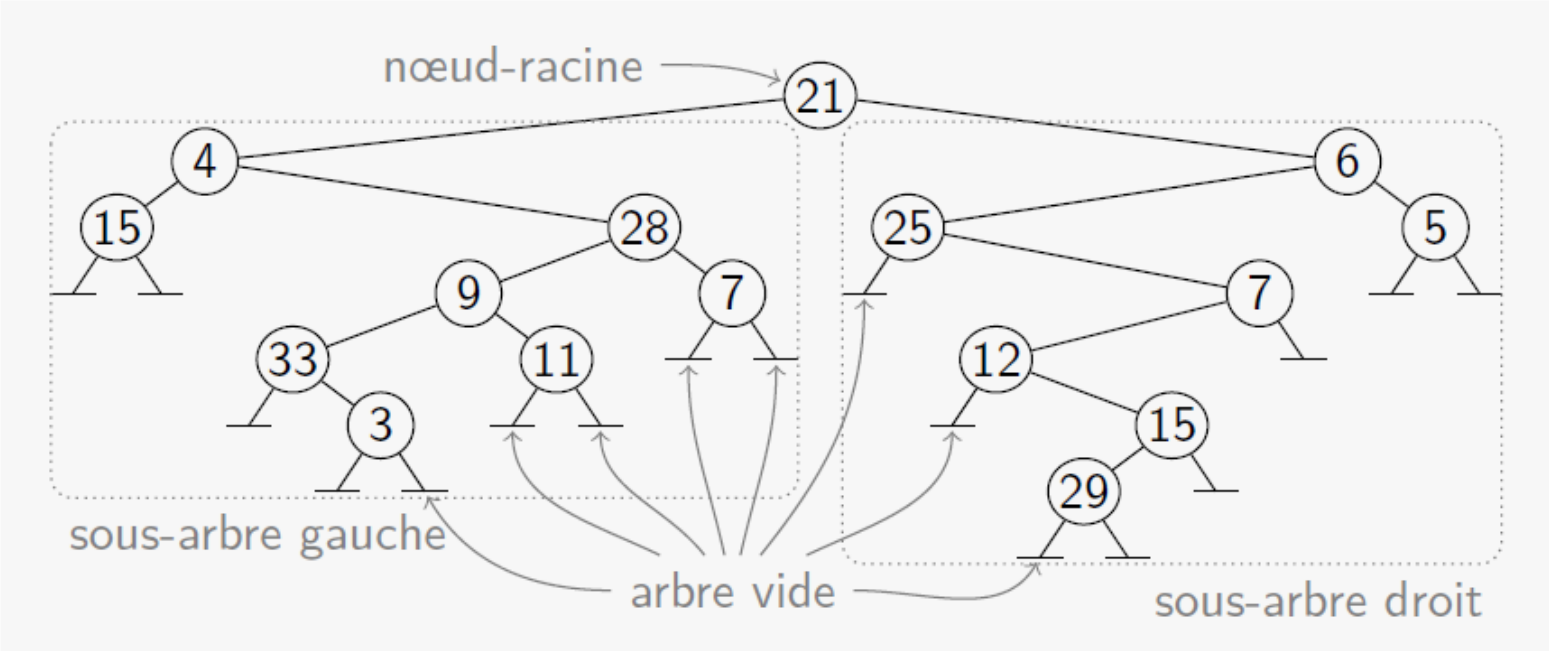
Représentation graphique d'un arbre binaire

⇒Définition informelle



Représentation graphique d'un arbre binaire

⇒**Définition récursive**

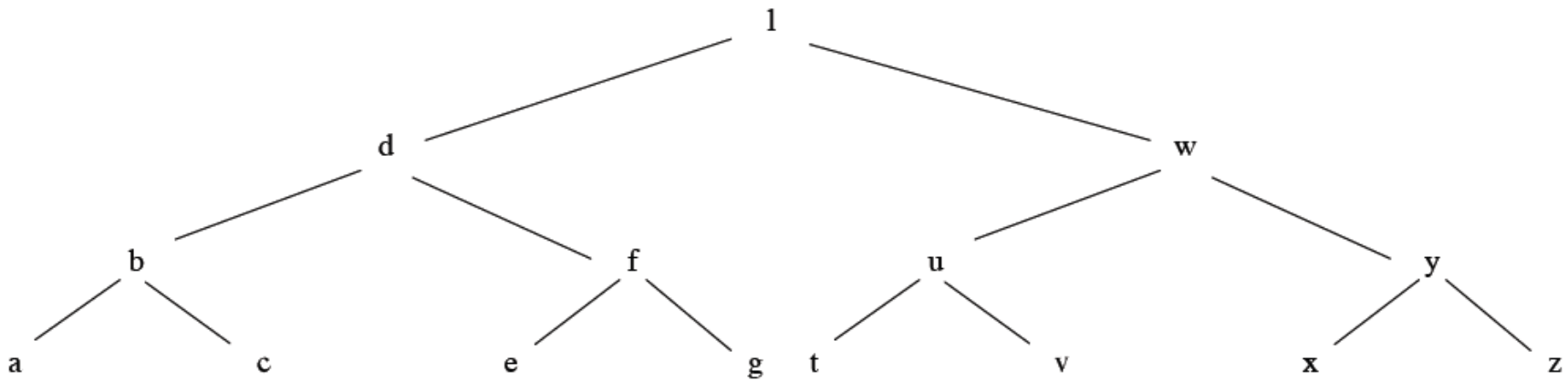


Arbres binaires particuliers

- **Arbre binaire dégénéré, filiforme** : Chaque noeud possède exactement un fils
  $\longrightarrow$ à éviter
- **Arbre binaire complet (uniforme)** : Chaque niveau est complètement rempli
  Cela signifie, tout sommet est soit une feuille au dernier niveau, soit possède exactement 2 fils
  $\longrightarrow$ situation idéale
- **Arbre binaire parfait (presque complet)** : Tous les niveaux sont complètement remplis sauf éventuellement le dernier et dans ce cas les feuilles sont le plus à gauche possible
- **Arbre binaire équilibré** : La différence de hauteur entre 2 frères ne peut dépasser 1

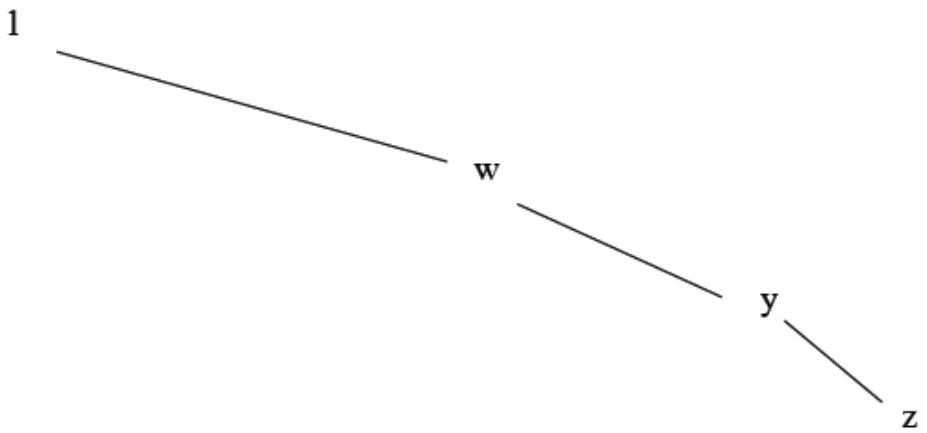
Arbres binaires particuliers

Arbre Parfait $\longrightarrow$ situation idéale



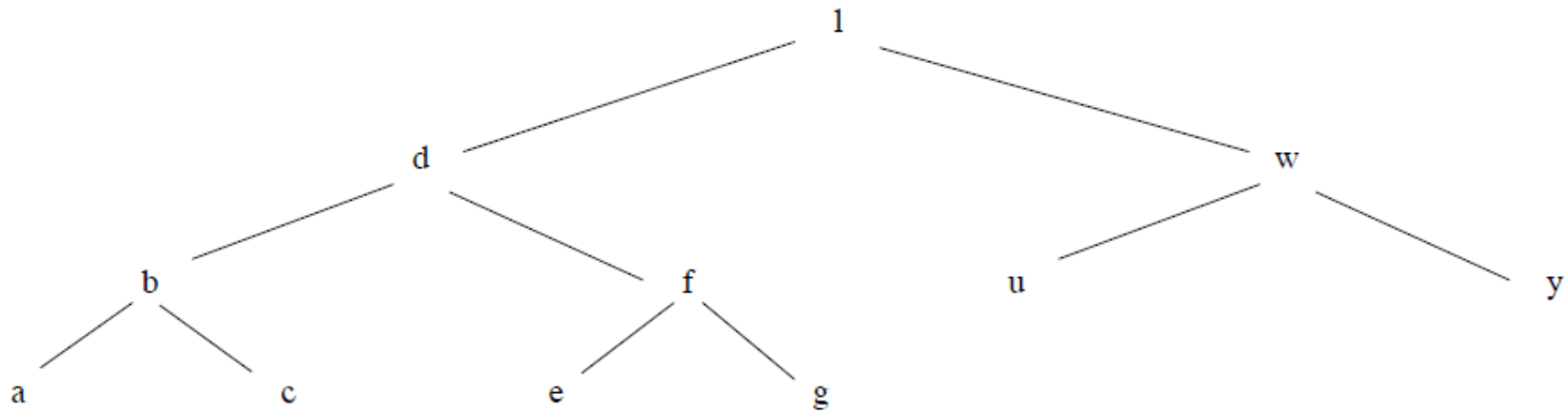
Arbres binaires particuliers

Arbre dégénéré $\longrightarrow$ Pire des situations



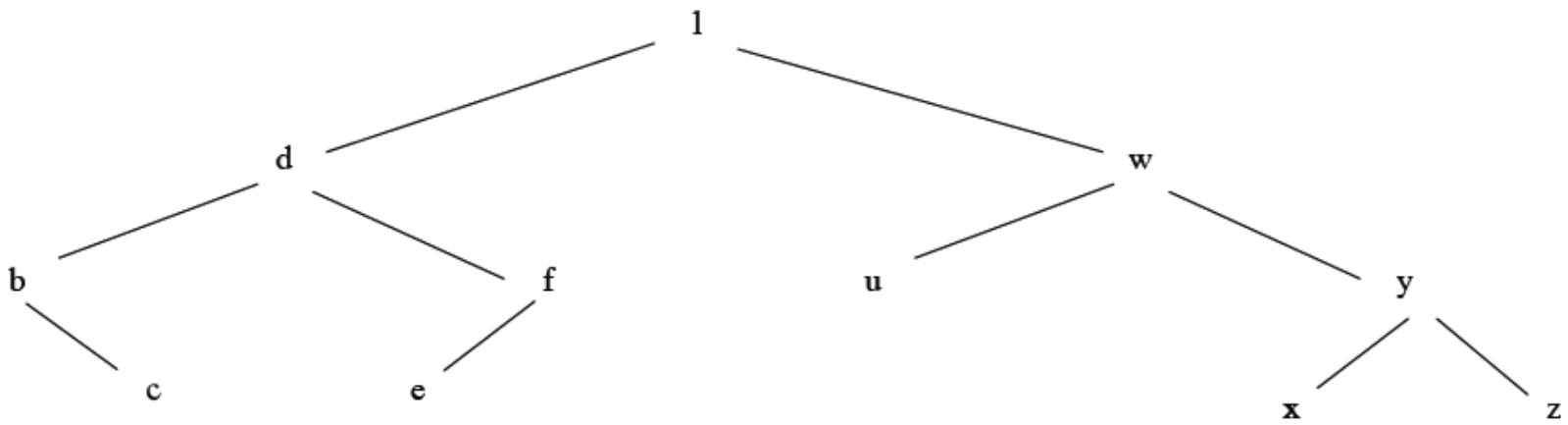
Arbres binaires particuliers

Arbre presque parfait



Arbres binaires particuliers

Arbre équilibré



Arbre binaire avec deux pointeurs sur ABG et ABD

- Un **arbre** binaire est soit :
  - un pointeur sur le noeud racine (arbre non vide)
  - le pointeur NULL (arbre vide)
- Un **noeud** est une structure à trois champs :
  - Une étiquette (valeur)
  - le sous-arbre gauche (ABG)
  - le sous-arbre droit (ABD)
- **Avantages** :
  - Définition récursive, simple à programmer,
  - la plus utilisée
- **Inconvénients** :
  - consomme de la mémoire dynamique
  - Temps d'exécution lent.

Arbre binaire avec deux pointeurs sur ABG et ABD

Définition d'un arbre binaire

```
typedef int Element ;
/* Définition du type noeud d'un arbre binaire */
typedef struct noeud{
    Element etiq ; /*le champ etiq peut avoir n'importe quel type*/
    struct noeud *ag ; /*pointeur contenant l'adresse du ABG*/
    struct noeud *ad ; /*pointeur contenant l'adresse du ABD*/
}Tnoeud ;
```

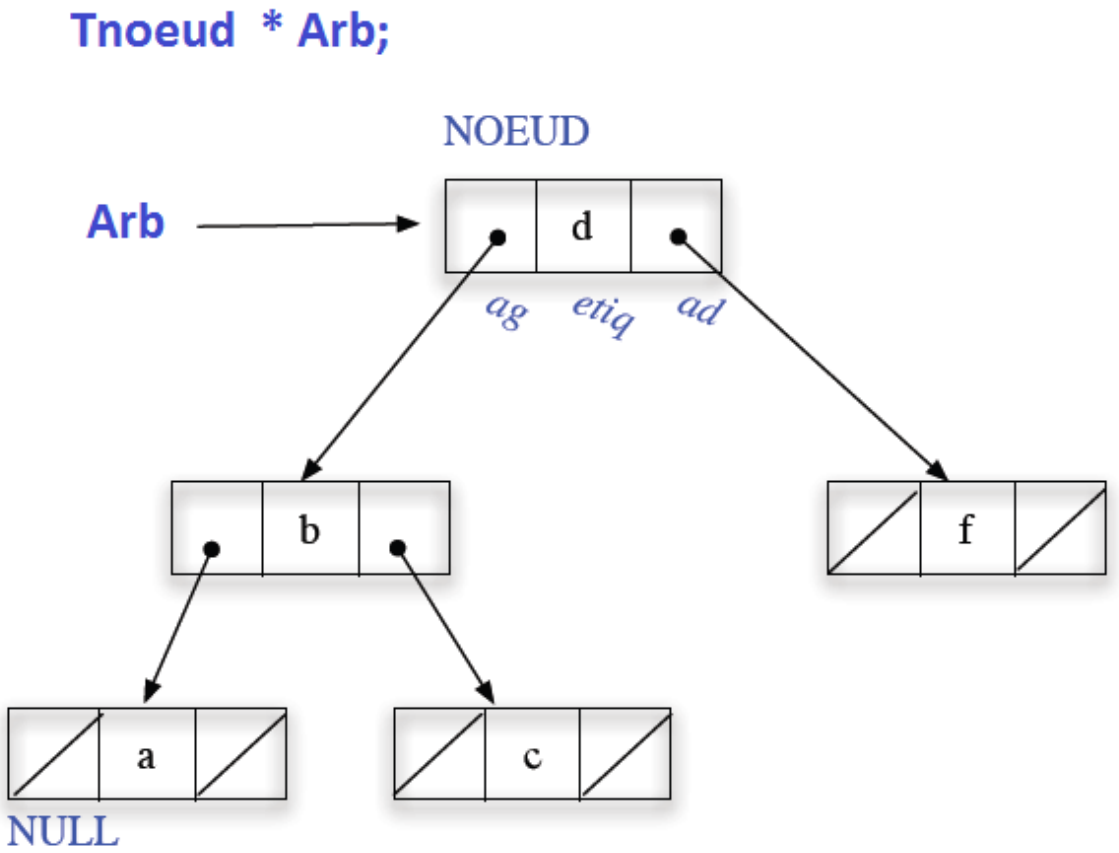
Déclaration d'un arbre binaire

- Pour déclarer une variable de type arbre, il suffit de déclarer un **pointeur** sur son **noeud racine**
- **Exemple :**

$$Tnoeud * Arb$$

Arbre binaire avec deux pointeurs sur ABG et ABD

⇒ **Exemple :**



Arbre binaire avec un tableau

- Un **arbre** binaire est une structure à deux champs :
  - Un tableau où sont mémorisés les noeuds
  - Un entier qui donne l'indice de la racine dans le tableau
- Un **noeud** est une structure à trois champs :
  - L'étiquette du noeud
  - L'indice de son fils gauche (ou 0 si pas de fils gauche)
  - L'indice de son fils droit (ou 0 si pas de fils droit)
- **Inconvénients** :
  - Définition non récursive (arbre $\neq$ sous-arbre)
  - Utilisée uniquement si on traite un arbre unique

Arbre binaire avec un tableau

Définition d'un arbre binaire

```
typedef int Element ;
/* Définition du type noeud d'un arbre binaire */
typedef struct noeud{
    Element etiq ; /*le champ etiq peut avoir n'importe quel type*/
    int fg ; /*indice du ABG*/
    int fd ; /*indice du ABD*/
}NOEUD ;
/* Définition d'un arbre binaire */
typedef struct {
    NOEUD tab[TAILLE_MAX] ; /*Tableau contenant les noeuds de l'arbre*/
    int racine ; /*indice de la racine*/
}Tnoeud ;
```

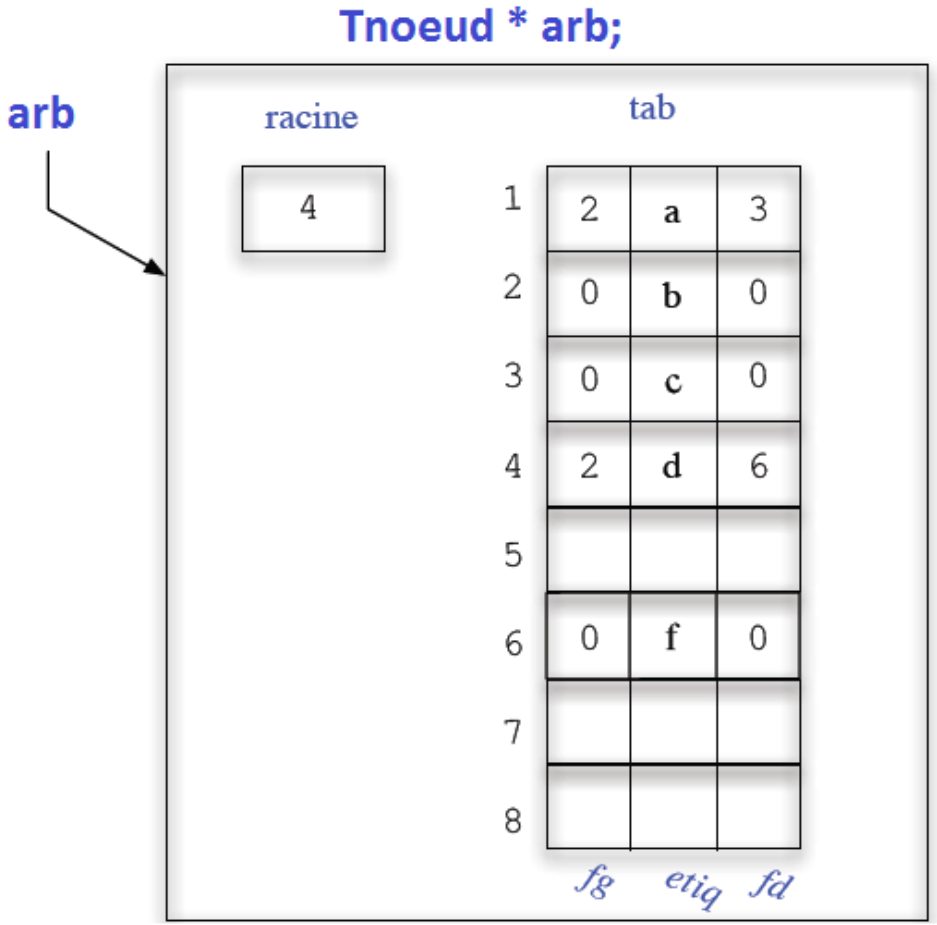
Déclaration d'un arbre binaire

- Pour déclarer un arbre, on utilise la syntaxe suivante :

$$Tnoeud * arb;$$

Arbre binaire avec un tableau

⇒ **Exemple :**



Créer un arbre vide :Tnoeud *ArbreBinVide(void)

Définition de la fonction

```
Tnoeud *ArbreBinVide(void)
{
    Tnoeud * A ;
    A=NULL ;
    return(A) ;
}
```

Remarque

⇒ On peut faire **typedef Tnoeud * arbreBin**, dans ce l'entête devient :

```
arbreBin ArbreBinVide(void)
{
    arbreBin A ;
    A=NULL ;
    return(A) ;
}
```

Tester si l'arbre binaire est vide : int EstVide(Tnoeud * A)

- La fonction **EstVide()** teste si l'arbre est vide ou non,
- elle renvoie 1 si elle est vide, et 0 sinon.

Définition de la fonction

```
int EstVide(Tnoeud * A)
{
    if (A==NULL)
        return 1;
    else
        return 0;
}
```

Autre méthode

```
arbreBin EstVide(arbreBin A)
{    if (A==NULL)
        return 1;
    else
        return 0;
}
```

Créer un noeud : Tnoeud * CreerNoeud(Element e, Tnoeud *l, Tnoeud *r) ;

- La fonction **CreerNoeud** renvoie un arbre binaire dont la racine est e, le fils gauche est l et le fils droit r.
- elle renvoie un message d'erreur s'il n' y a pas assez d'espace.

Définition de la fonction

```
Tnoeud * CreerNoeud(Element e, Tnoeud *l, Tnoeud *r)
    {Tnoeud *new ;
    new=(Tnoeud*)malloc(sizof(Tnoeud)) ;
    if (new==NULL)
        {printf("Allocation ratée !") ;
            exit(-1)//pour quitter le programme ;
        }
    else
        { new− >etiq=e ;
            new− >fg=l ;
            new− >fd=r ;
            return new ;
        }
    }
```

Créer une feuille : Tnoeud * CreerFeuille(Element e) ;

- La fonction **CreerFeuille** renvoie un arbre binaire dont la racine est e et les fils gauche et droit sont vides.

Définition de la fonction

```
Tnoeud * CreerFeuille(Element e)
    {
        return CreerNoeud(e,ArbreBinVide(),ArbreBinVide()) ;
    }
```

Une autre solution

```
arbreBin CreerFeuille(Element e)
    {arbreBin new ;
    new=(arbreBin)malloc(sizof(Tnoeud)) ;
    new− >etiq=e ;
    new− >fg=NULL ;
    new− >fd=NULL ;
    return new ;
    }
```

Valeur de la racine : Element ValRacine(Tnoeud * A) ;

- La fonction **ValRacine** renvoie l'élément à la racine de l'arbre A s'il n'est pas vide.
- Elle provoque une erreur si bt est vide.

Définition de la fonction

```
Element ValRacine(Tnoeud * A)
    {
    if (EstVide(A))
        {
            printf("Pas de noeud à la racine d'un arbre vide!!!");
            exit(-1)//pour quitter le programme;
        }
    else
        {
            return (A->etiq);
        }
    }
```

Adresse du fils gauche :Tnoeud * FilsGauche(Tnoeud *A) ;

- La fonction **FilsGauche** renvoie l'adresse du fils gauche de A s'il n'est pas vide.
- Elle provoque une erreur si A est vide.

Définition de la fonction

```
Tnoeud * FilsGauche(Tnoeud *bt)
    {
    if (EstVide(A))
        {
            printf("Pas de fils gauche dans un arbre vide!!!");
            exit(-1)//pour quitter le programme;
        }
    else
        {
            return (A->fg);
        }
    }
```

Adresse du fils droit : Tnoeud * FilsDroit(Tnoeud *A) ;

- La fonction **FilsDroit** renvoie l'adresse du fils droit de A s'il n'est pas vide.
- Elle provoque une erreur si A est vide.

Définition de la fonction

```
Tnoeud * FilsGauche(Tnoeud *bt)
    {
    if (EstVide(A))
        {
            printf("Pas de fils droit dans un arbre vide!!!");
            exit(-1)//pour quitter le programme;
        }
    else
        {
            return (A->fd);
        }
    }
```

Tester si un noeud est une feuille : int EstFeuille(Tnoeud * F)

- La fonction **EstFeuille()** teste si un noeud est une feuille
- elle renvoie 1 si elle est une feuille, et 0 sinon.

Définition de la fonction

```
int EstFeuille(Tnoeud * F)
{
    if (A==NULL)
        return 0;
    if (A->fg==NULL && A->fd==NULL)
        return 1;
    else
        return 0;
}
```

Autre solution

```
int EstFeuille(arbreBin F)
{
  return !EstVide(F) && EstVide(FilsGauche(F)) && EstVide(FilsDroit(F));
}
```

Insérer une feuille comme comme fils droit d'un noeud void insertDroit(Tnoeud *n, Element e)

- La fonction **insertDroit()** insère une feuille contenant e comme fils droit de n.
- elle provoque une erreur si n est vide ou si son fils droit n'est pas vide.

Définition de la fonction

```
void insertDroit(Tnoeud *n, Element e)
{
    if( !EstVide(n) && EstVide(FilsDroit(n)))
        n->fd=CreerFeuille(e);
    else
        {
            printf("Impossible d'insérer un noeud comme un fils droit !!");
            exit(-1)//pour quitter le programme;
        }
}
```

Insérer une feuille comme comme fils gauche d noeud void insertGauche(Tnoeud *n, Element e)

- La fonction **insertGauche()** insère une feuille contenant e comme un fils gauche de n.
- elle provoque une erreur si n est vide ou si son fils gauche n'est pas vide.

Définition de la fonction

```
void insertGauche(Tnoeud *n, Element e)
{
    if( !EstVide(n) && EstVide(FilsGauche(n)))
        n->fg=CreerFeuille(e);
    else
        {
            printf("Impossible d'insérer un noeud comme un fils gauche!!");
            exit(-1)//pour quitter le programme;
        }
}
```

Supprimer un fils droit d'un noeud Element SupprimerDroit(Tnoeud *n)

- La fonction **SupprimerDroit()** supprime et renvoie la racine du fils droit de n si c'est une feuille.
- Provoque une erreur si n est vide ou si son fils droit n'est pas une feuille.

Définition de la fonction

```
Element SupprimerDroit(Tnoeud *n)
{
    Element res;
    if(EstVide(n) || !EstFeuille(FilsDroit(n)))
        {
            printf("Impossible de supprimer le fils droit !!");
            exit(-1)//pour quitter le programme;
        }
    else
        {
            res=ValRacine(n->fd);
            n->fd=ArbreBinVide();
            return res;
        }
}
```

Supprimer un fils gauche d'un noeud Element SupprimerGauche(Tnoeud *n)

- La fonction **SupprimerGauche()** supprime et renvoie la racine du fils gauche de n si c'est une feuille.
- Provoque une erreur si n est vide ou si son fils gauche n'est pas une feuille.

Définition de la fonction

```
Element SupprimerGauche(Tnoeud *n)
{
    Element res;
    if(EstVide(n) || !EstFeuille(FilsGauche(n)))
        {
            printf("Impossible de supprimer le fils gauche!!");
            exit(-1)//pour quitter le programme;
        }
    else
        {
            res=ValRacine(n->fg);
            n->fg=ArbreBinVide();
            return res;
        }
}
```

Insèrer une feuille le plus à droite possible void insertDroitplusProfond(TNoeud **bt, Element e)

- La fonction **insertDroitplusProfond()** insère une feuille contenant e le plus à droite possible dans l'arbre *bt.
- Si l'abre est vide, l'insertion se fait à la racine.

Définition de la fonction

```
void insertDroitplusProfond(TNoeud **bt, Element e)
{
    Tnoud * tmp ;
    if(EstVide(*bt)
        *bt=CreerFeuille(e) ;
    else
        {
            tmp=*bt ;
            while( !Estvide(FilsDroit(tmp)))
                tmp=FilsDroit(tmp) ;
            insertDroit(tmp,e) ;
        }
}
```

Supprimer le noeud le plus à gauche possible void SupprimerGaucheplusProfond(TNoeud **bt)

- La fonction **SupprimerGaucheplusProfond()** supprime (et renvoie sa racine) le noeud le plus à gauche possible dans l'arbre *bt.
- Provoque une erreur si *bt est vide. Peut supprimer la racine si elle est le noeud le plus à gauche.

Définition de la fonction

```
void SupprimerGaucheplusProfond(TNoeud **bt){
    Tnoud * tmp ;
    Element res ;
    if(EstVide(*bt)
        printf("impossible ! ! !") ;
    if(EstVide(FilsGauche(*bt))){
            res=ValRacine(*bt) ;
            *bt=FilsDroit(*bt) ; }
    else{   tmp=*bt ;
            while( !Estvide(FilsGauche(FilsGauche(tmp))))
                tmp=FilsGauche(tmp) ;
            res=ValRacine(FilsGauche(tmp)) ;
            tmp- >fg=(tmp- >fg)- >fd ;}
    return res ; }
```

Rechercher un élément : int Rechercher(Tnoeud *a, Element v)

Définition de la fonction

```
int Rechercher(Tnoeud *a, Element v)
    {
    if (EstVide(a))
        return 0;
    if (a->etiq==v)
        return 1;
    if (Rechercher(FilsGauche(a),v))
        return 1;
    return Rechercher(FilsDroit(a),v))
    }
```

Notion de parcours

Défintion

- Un **parcours** est un algorithme qui appelle une **fonction**, sur tous les noeuds (ou les sous arbres) d'un arbre.
- L'ordre sur les nœuds dans lequel la procédure est appelée doit être fixé.
- Il y a de nombreux choix possibles qui sont classés en deux familles :
  1. Les algroithmes de parcours en **largeur**.
  2. Les algroithmes de parcours en **profondeur**.

Quelques exemples de fonctions

- affichage,
- liste des valeurs,
- modification
- etc.

Parcours en profondeur

Défintion

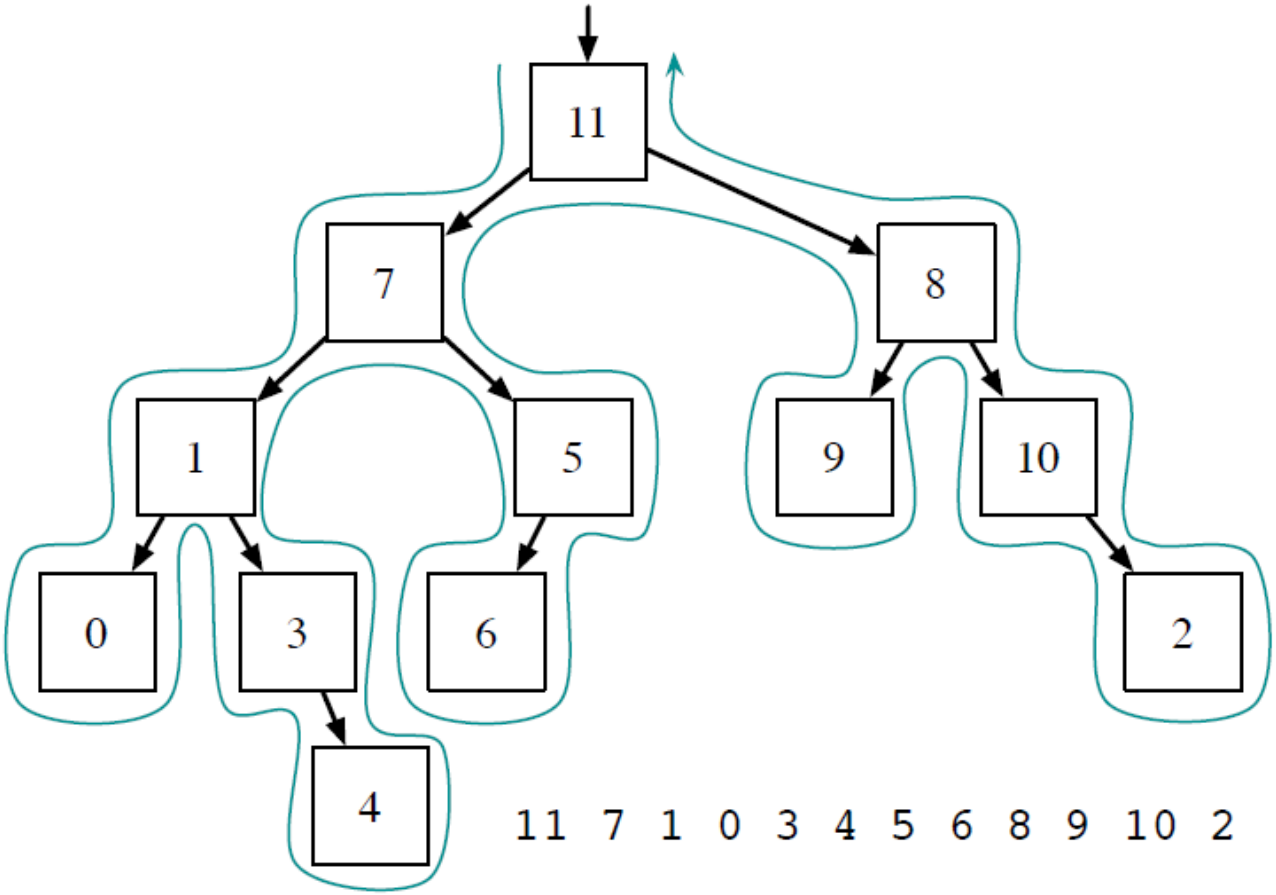
Un **parcours** est dit en **profondeur** lorsque, systématiquement, si l'arbre n'est pas vide, le parcours de l'un des deux sous-arbres est terminé avant que ne commence celui de l'autre.

Les différents types

- On distingue trois types :
  1. **Parcours préfixe** (Racine Gauche Droit)
     $\longrightarrow$ le noeud racine est traité au premier passage avant le parcours des sous-arbres
  2. **Parcours infixe** ou symétrique (Gauche Racine Droit)
     $\longrightarrow$ le noeud racine est traité au second passage après le parcours du sous-arbre gauche et avant le parcours du sous-arbre droit
  3. **Parcours postfixe** (Gauche Droit Racine)
     $\longrightarrow$ le noeud racine est traité au dernier passage après le parcours des sous-arbres

Parcours préfixe RGD

⇒Exemple :



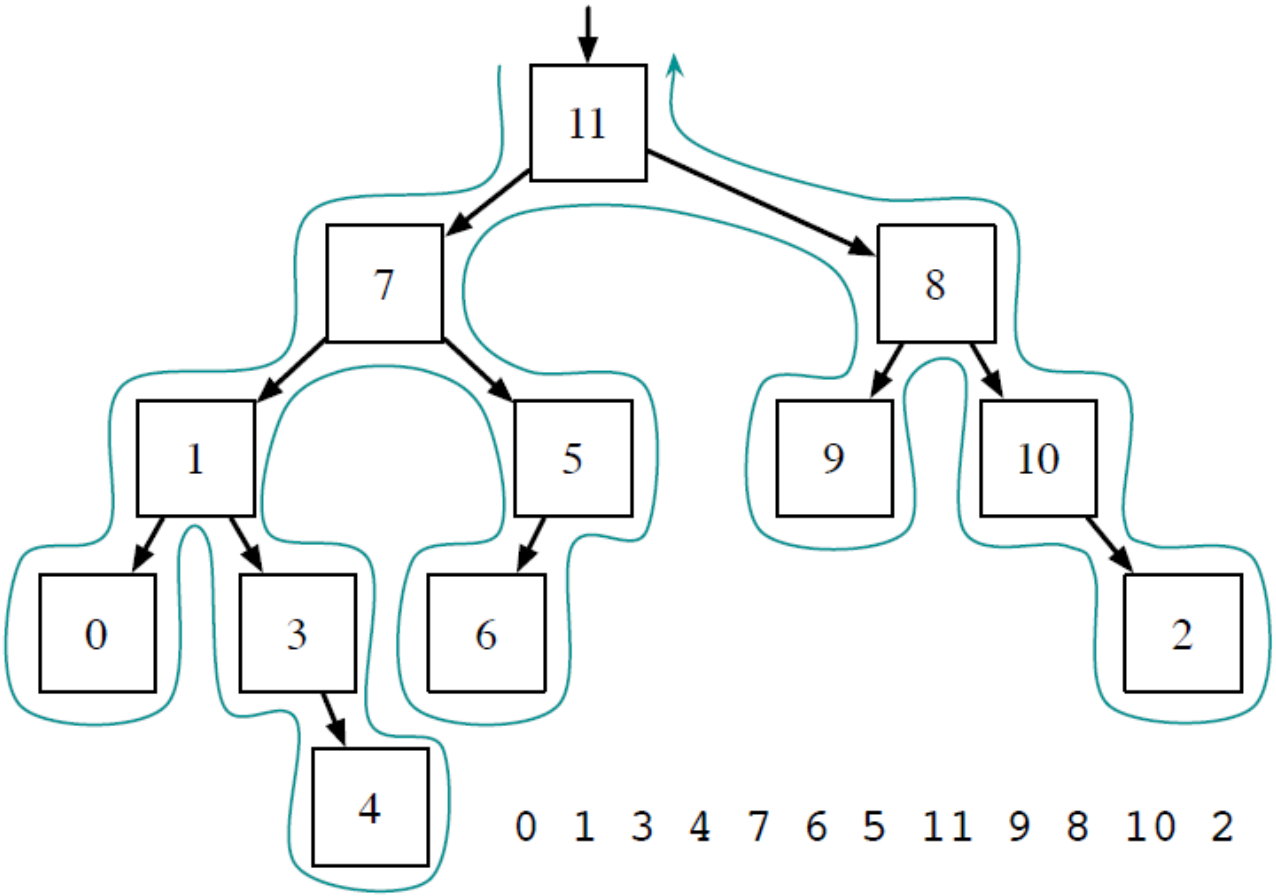
Algorithme de parcours préfixe RGD

- **Principe du parcours préfixe :**
  1. application d'une fonction F à la racine,
  2. parcours préfixe du sous-arbre gauche,
  3. parcours préfixe du sous-arbre droit
- **Code C :**

```
void ParcoursPrefixe(Tnoeud * A)
{
    if (A !=NULL)
        {
            printf("%d",ValRacine(A));
            ParcoursPrefixe(A->fg);
            ParcoursPrefixe(A->fd);
        }
}
```

Parcours infixe GRD

⇒Exemple :



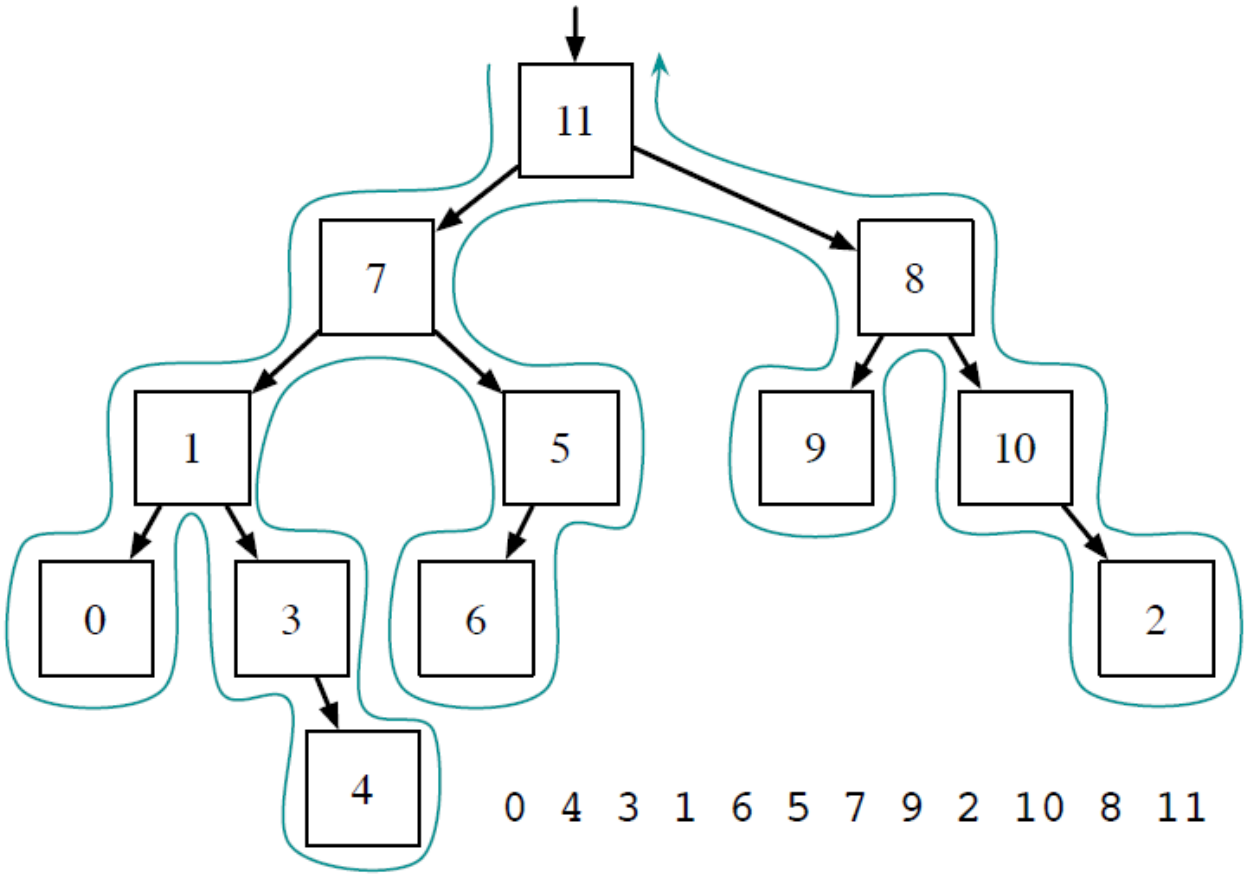
Algorithme de parcours infixe GRD

- **Principe du parcours infixe :**
  1. parcours infixe du sous-arbre gauche,
  2. application d'une fonction F à la racine,
  3. parcours infixe du sous-arbre droit
- **Code C :**

```
void ParcoursInfixe(Tnoeud * A)
{
    if (A !=NULL)
        {
            ParcoursInfixe(A->fg);
            printf("%d",ValRacine(A));
            ParcoursInfixe(A->fd);
        }
}
```

Parcours postfixe GDR

⇒Exemple :



Algorithme de parcours postfixe GDR

- **Principe du parcours postfixe :**
  1. parcours postfixe du sous-arbre gauche ;
  2. parcours postfixe du sous-arbre droit ;
  3. application d'une fonction F à la racine.
- **Code C :**

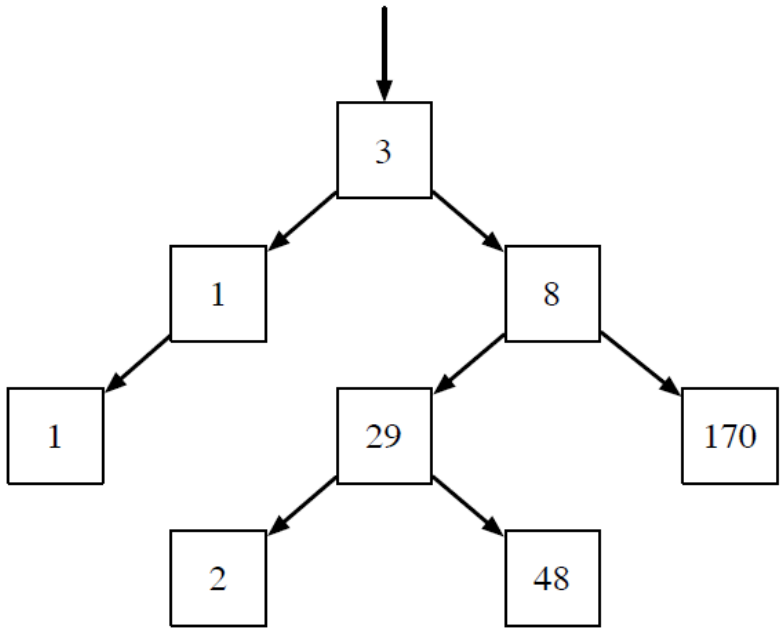
```
void ParcoursPostfixe(Tnoeud * A)
{
    if (A !=NULL)
        {
            ParcoursPostfixe(A- >fg) ;
            ParcoursPostfixe(A- >fd) ;
            printf("%d",ValRacine(A)) ;
        }
}
```

Parcours en largeur

Définition

Un **parcours** est dit **en largeur** lorsqu'il procède en croissant selon les niveaux.

Voici un parcours en largeur de gauche à droite



3 1 8 1 29 170 2 48

Algorithme de parcours en largeur

Idée : on remplace la pile d'appels par une file d'attente dans l'algorithme de parcours préfixe

```
ParcoursEnLargeur (Tnoeud *A)
Début
    créer une file vide F
    SI A est non vide ALORS
        Enfiler A dans F
        Tant que (F non vide) faire
            A <-- Défiler(F)
            traitement(A)
            SI ABG de A non vide ALORS
                Enfiler ABG de A dans F
            FSI
            SI ABD de A non vide ALORS
                Enfiler ABD de A dans F
            FSI
        FinTQ
    FSI
    détruire F
Fin
```