

Les Arbres

Module INFO 0401

2^{ème} année informatique

Reims – Sciences Exactes

Cours 7

A.HEBBACHE

sommaire

1 Structures de données arborescentes

- Définition et terminologie

- Algorithmes de parcours

2 Les arbres binaires de recherche

- Définition

- Opérations sur les ABRs

- Les AVLs

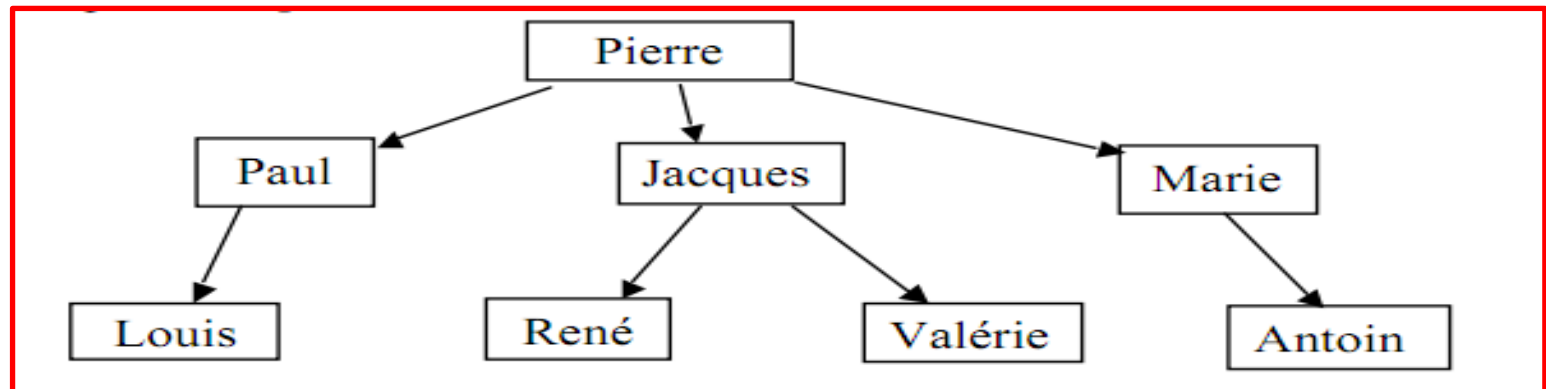
- Les arbres noir et blanc (bicolors)

Définition d'un arbre

- Un **arbre** est un ensemble de **noeuds**, organisés de façon hiérarchique, à partir d'un noeud distingué, appelé **racine**.
- Une propriété intrinsèque de la structure d'arbre est la récursivité, et les définitions des caractéristiques des arbres, aussi bien que les algorithmes qui manipulent des arbres s'écrivent très naturellement de manière récursive.

Définition

- ❑ Une **arborescence** est une structure en forme d'arbre qui permet d'organiser les données en mémoire ou sur disque, de manière logique et hiérarchisée.
- ❑ **Exemple** : Arbre généalogique



Définition

- Organisation hiérarchique des informations.
- S'applique à beaucoup de domaines :
 - arbre généalogique
 - structures syntaxiques
 - décomposition de structures :

type structuré,
expression,
arithmétique,
langue naturelle,
...

Définition

- La structure d'arbre est l'une des plus importantes et des plus spécifiques de l'informatique.

Par exemple, c'est sous forme d'arbre que sont organisés les fichiers dans des systèmes d'exploitation tels qu'UNIX/Linux ; c'est aussi sous forme d'arbres que sont représentés les programmes traités par un compilateur...

- Dans les systèmes d'exploitation, arbre associé à un document XML (voir exemple), arbre syntaxique, dictionnaire, relation d'inclusion d'ensembles ...

Exemple XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Commentaire -->
<ex:collection
  xml:lang="fr"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ex="http://exemple.org"
>
  <élément>Texte</élément>
  <dc:title>Astérix le Gaulois</dc:title>
  <ex:livre attribut="valeur" type="BD">
    <dc:title>Astérix chez les Belges</dc:title>
    <!-- élément répété -->
    <dc:creator>René Goscinny</dc:creator>
    <dc:creator>Albert Uderzo</dc:creator>
    <dc:description>
      <b>Astérix chez les Belges</b> est un album de
      <a href="http://fr.wikipedia.org/wiki/Bande_dessinée">bande dessinée</a>
      de la série Astérix le Gaulois créée par René Goscinny et Albert Uderzo.
      <br /><!-- élément vide -->
      Cet album publié en 1979 est le dernier de la série écrit par René Goscinny.
    </dc:description>
  </ex:livre>
</ex:collection>
```

XML

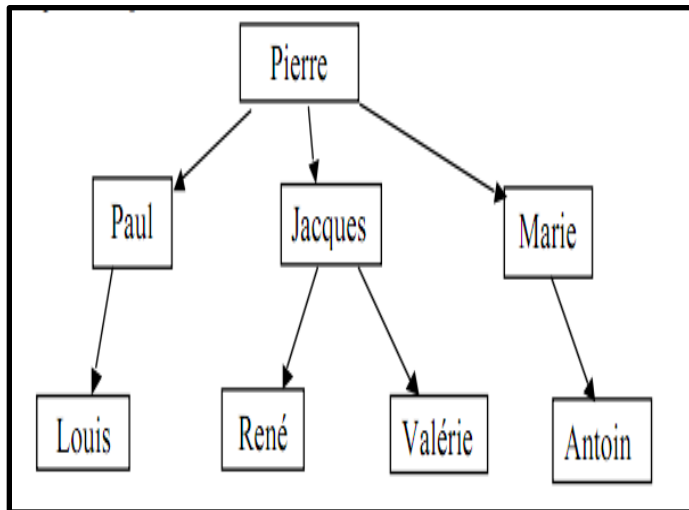
- Un document XML a toujours une et une seule racine, le nœud document.
- Dans le langage d'accès à un document XML, XPath, le nœud document est abrégé avec la barre oblique /, comme la racine de l'arborescence d'un système de fichiers Unix/Linux.
- La racine peut éventuellement comporter des enfants de type commentaire ou instruction de traitement, elle doit obligatoirement comporter un et un seul élément.

Définition

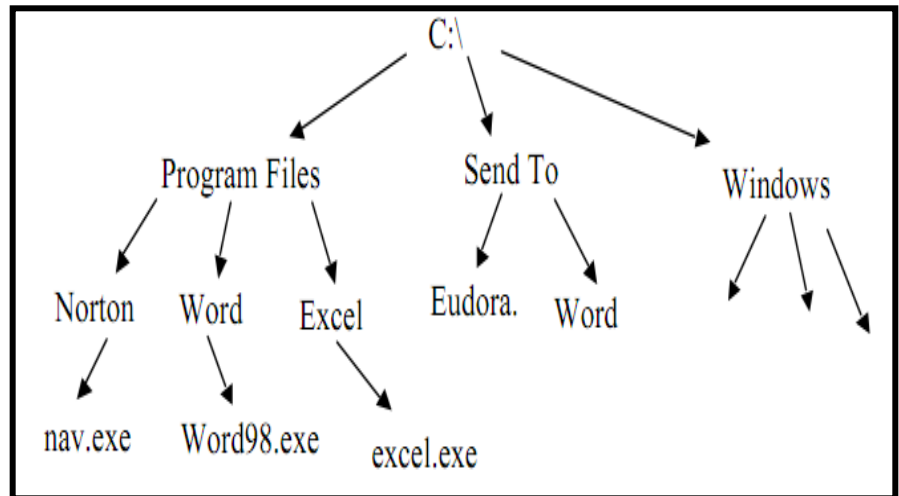
- **Les structures séquentielles ont des inconvénients**
 - En format contigu, les mises à jour sont fastidieuses
 - En format chaîné, les parcours sont de complexité linéaire.
- **Les structures arborescentes** permettent une amélioration globale des accès aux informations.

Définition

- Exemples



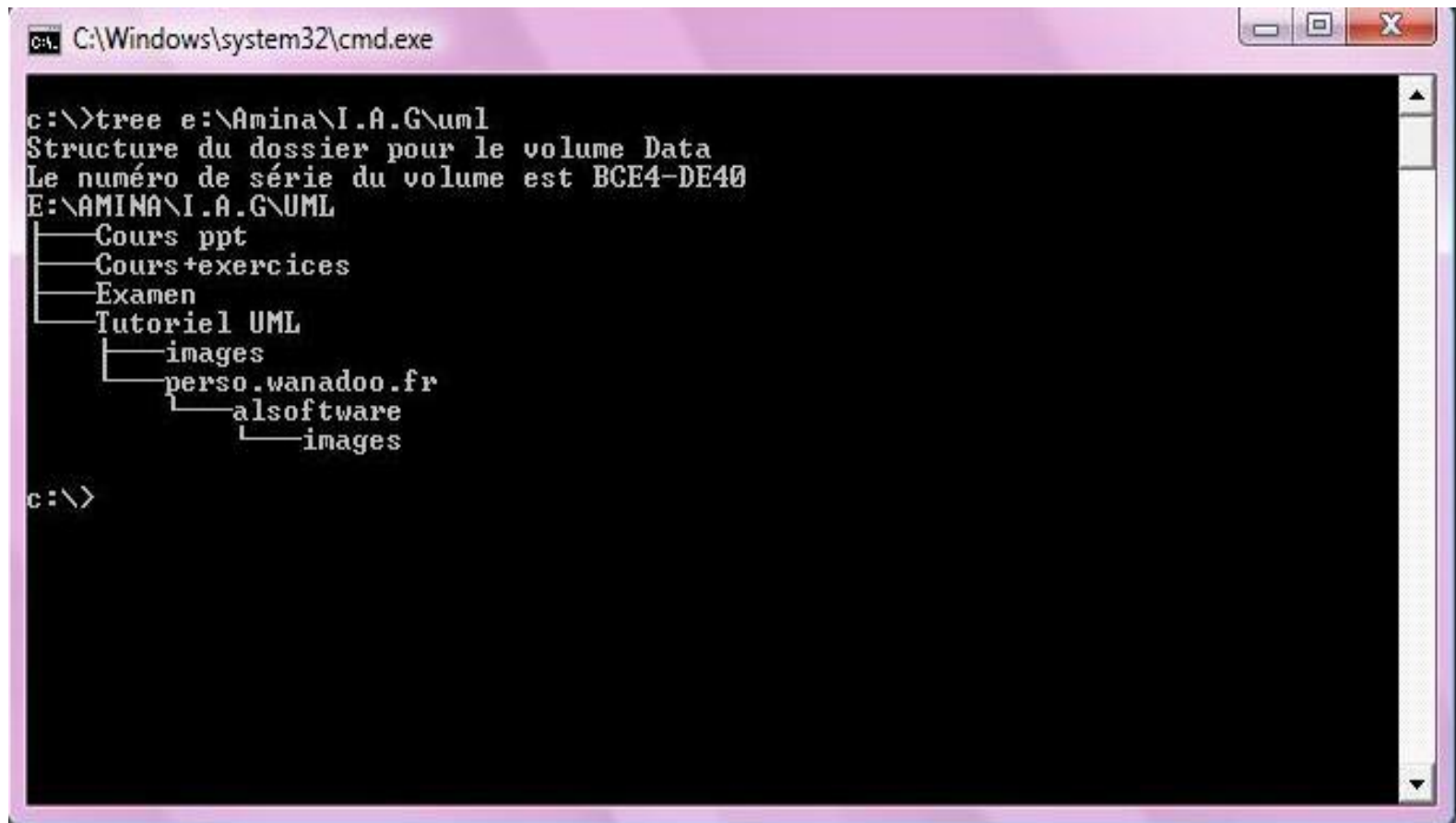
arbre généalogique



arborescence des répertoires et
des fichiers

Organisation d'un répertoire

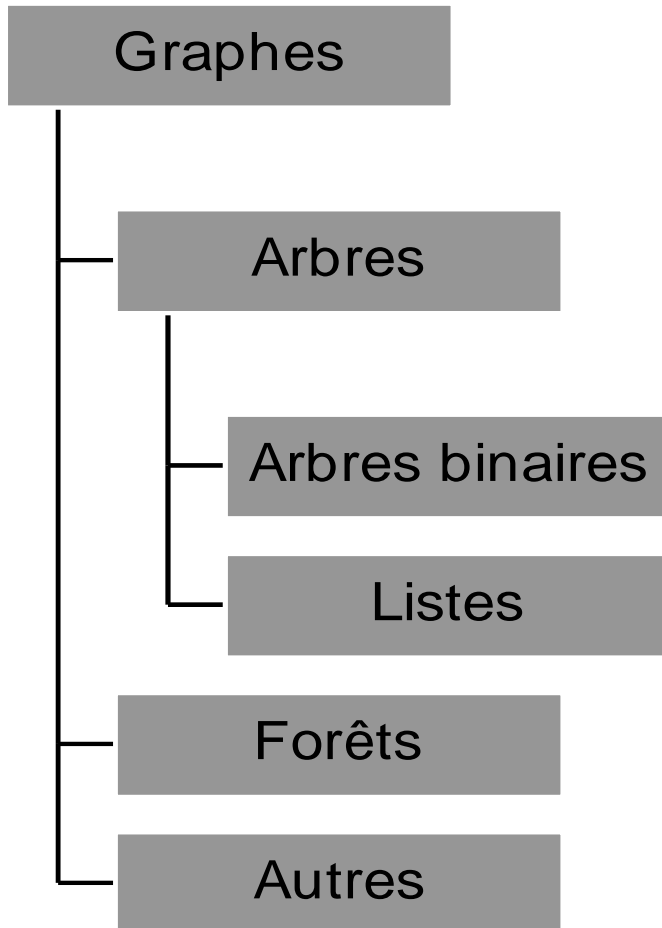
Exemple



```
C:\Windows\system32\cmd.exe

c:\>tree e:\Amina\I.A.G\uml
Structure du dossier pour le volume Data
Le numéro de série du volume est BCE4-DE40
E:\AMINA\I.A.G\UML
├── Cours ppt
├── Cours+exercices
├── Examen
├── Tutoriel UML
│   ├── images
│   ├── perso.wanadoo.fr
│   │   └── alsoftware
│   │       └── images
└──
```

Classification des structures



Arbre

Graphe purement hiérarchique

pas de cycle

un seul chemin d'un nœud à un autre

Arbre binaire

Tout nœud a au plus deux fils

Liste

Arbre dégénéré

Forêt

Ensemble d'arbres

Terminologie

- **Sur les arbres on définit les termes suivants :**

Racine

C'est le nœud qui n'a pas de prédécesseur.

La racine constitue la caractéristique d'un arbre.

Feuille

C'est le nœud qui n'a pas de successeur.

Une feuille est aussi appelée un **nœud externe**.

Nœud interne

C'est tout nœud qui admet au moins un successeur.

Fils d'un nœud : Ce sont ses successeurs.

Frères : Ce sont les successeurs issus d'un même nœud.

Père : C'est un nœud qui admet au moins un successeur.

Terminologie

| | |
|-------------------------|---|
| Sous arbre : | C'est une portion de l'arbre. |
| Descendants d'un nœud : | Ce sont tous les nœuds du sous arbre de racine nœud. |
| Ascendants d'un nœud : | Ce sont tous les nœuds se trouvant sur la branche de la racine vers ce nœud. |
| Branche : | Les ascendants d'une feuille constituent une branche de l'arbre. |
| Degré d'un nœud : | C'est le nombre de ses fils. |
| Niveau d'un nœud : | On dit que la racine a le niveau 0, ses fils ont le niveau 1, les fils des fils ont le niveau 2. Etc. |
| Profondeur de l'arbre : | C'est le niveau maximal des feuilles de l'arbre. |
| Forêt : | C'est un ensemble d'arbres. |

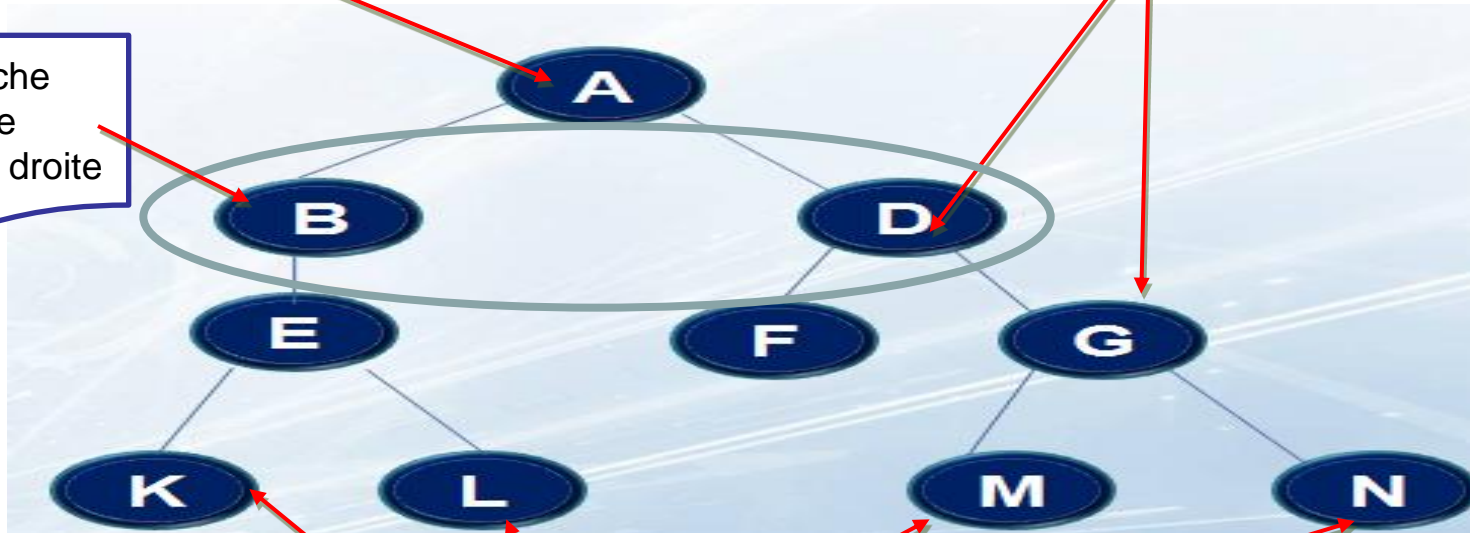
Terminologie

- Illustration

Racine
(nœud principal)

B,D,E,G
sont des nœuds internes

B : branche
gauche
D : branche droite

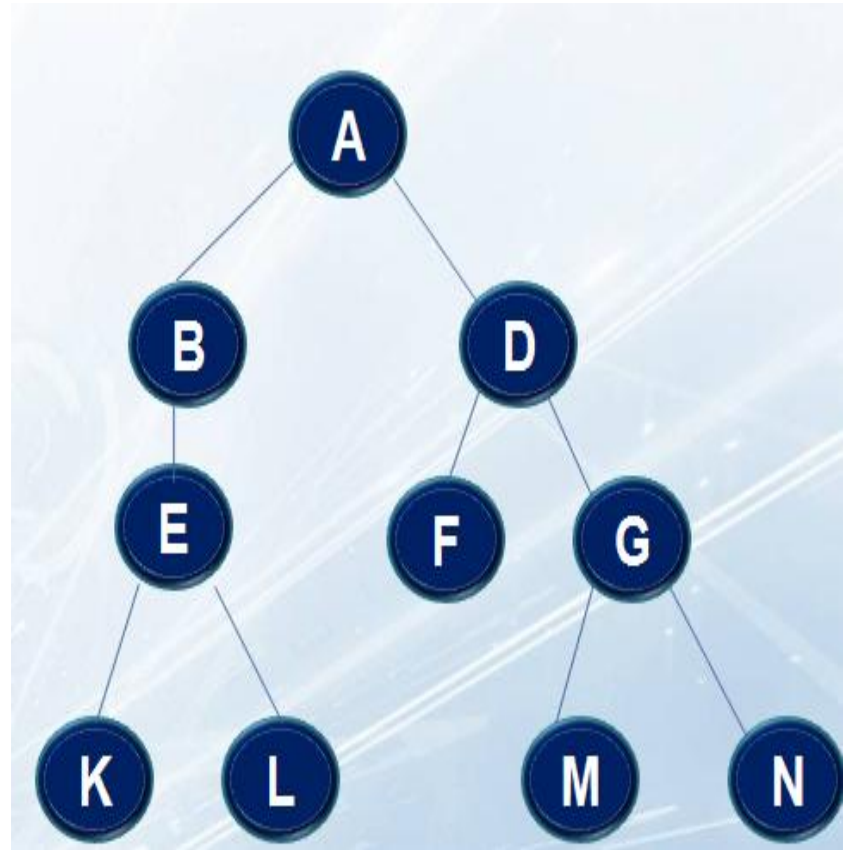


K, L, M,N,F sont des
nœuds externes (feuilles)

Terminologie

Illustration

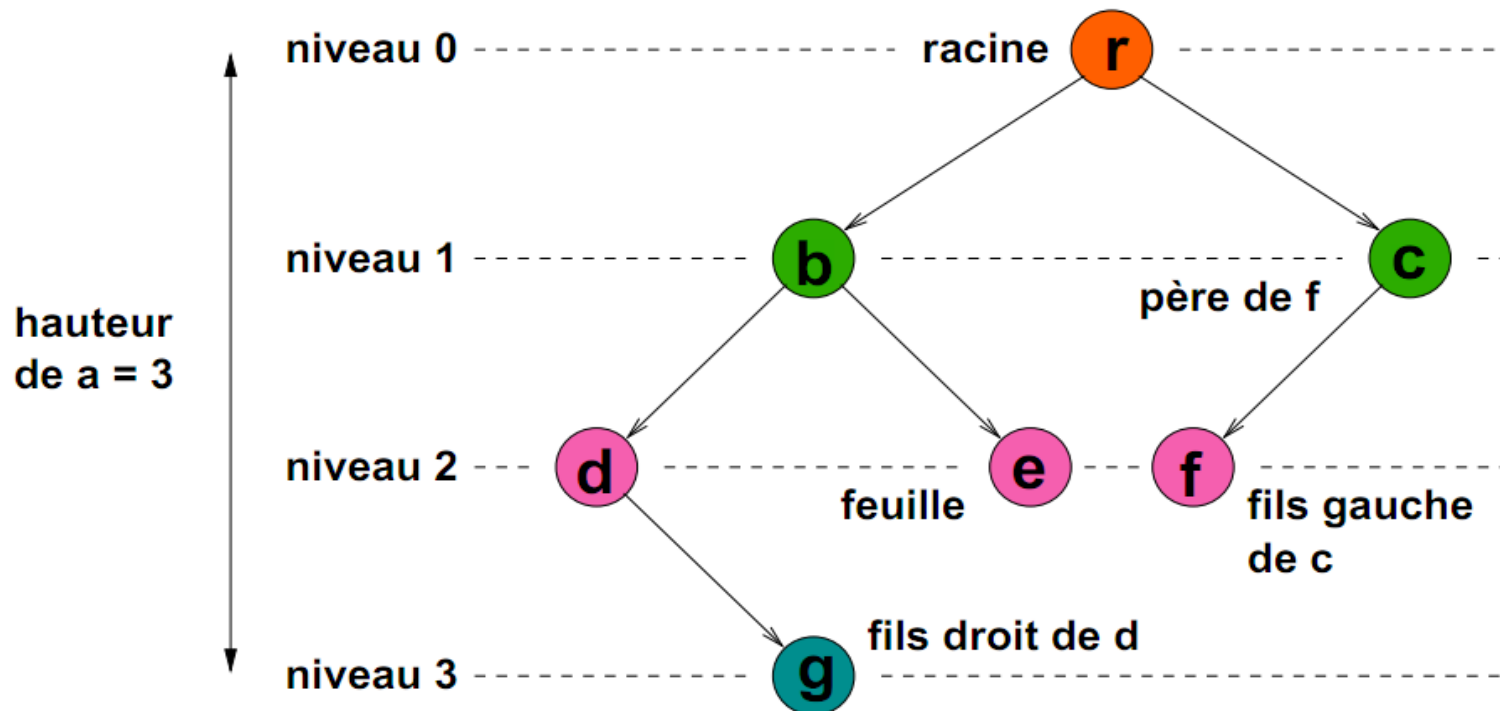
- Taille : ?
- Hauteur : ?
- Ordre : ?
 - Taille : 10
 - Hauteur : 3
 - Ordre : 2



Terminologie

- Illustration

-Exemple de la hauteur d'un arbre



Terminologie

- **Calcul de la hauteur d'un arbre:**

Pour calculer la hauteur d'un arbre, nous allons nous baser sur la définition récursive :

- un arbre vide est de hauteur **0**
- un arbre non vide a pour hauteur **1 + la hauteur maximale entre ses fils.**

Terminologie

- **Calcul du nombre de nœuds d'un arbre**

On utilise la définition récursive :

- Si l'arbre est vide : renvoyer **0**
- Sinon renvoyer **1 + la somme du nombre de nœuds des sous arbres.**

Les opérations sur les arbres

- Afin d'assurer une cohérence avec les autres structures de données (liste, pile, file)

⇒ décrire une abstraction d'un arbre avec une structure de données abstraite.

⇒ définir l'ensemble des opérations sur cette structure de données.

Les opérations sur les arbres

- Chaque nœud de l'arbre contient une valeur de type **TypeValeur**

| | |
|-------------------|--------------|
| creerArbre | → Arbre |
| estArbreVide | → Booléen |
| valeurRacineArbre | → TypeValeur |
| construireArbre | → Arbre |
| sousArbreGauche | → Arbre |
| sousArbreDroit | → Arbre |
| estElémentArbre | → Booléen |

Les opération sur les arbres

Algorithme de construction d'un arbre

```
Structure arbre{
    structure *fg;
    structure *fd;
    valeur  val;}

Construire_arbre(*racine:arbre ) {
    p,q:arbre;
    p = q = null;
    Allouer(p) ;
    p->valeur = val;
    p->fg = null;
    p->fd = null;
    Si (racine == NULL)
        racine = p;
        q = p;
    Sinon
        // insérer à gauche
        q->fg = p;
        q = p;
        //insérer à droite
        q->fd = p;
        q = p;
    fin
```

Les opération sur les arbres

Exemple

L'arbre suivant est créé par :

On crée le nœud contenant une valeur;

On initialise le nœud comme suit

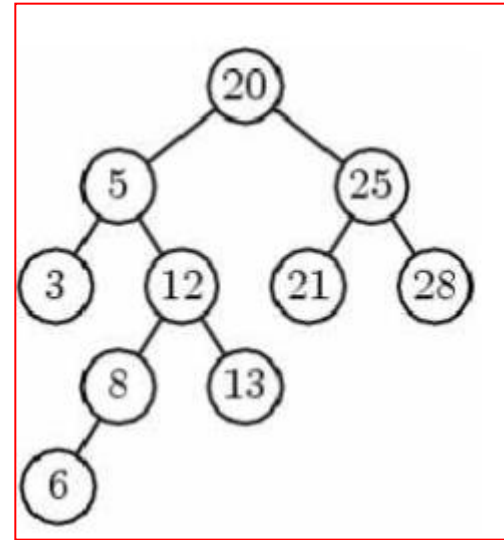
nœud->fg=null ; nœud->fd=null ;

Si la racine est null alors

on initialise la racine à ce nœud créé.

Sinon

On insère le nœud dans le sous arbre gauche ou bien dans le sous arbre droit.



Les opération sur les arbres

```
struct  pointeur *q;
struct  pointeur *racine;
main()
{
    int i;
    int j;
    int hauteur=5;
    racine=p=q=NULL;
    for(i=0;i<=5;i++)
    {
        p =malloc(sizeof(racine));

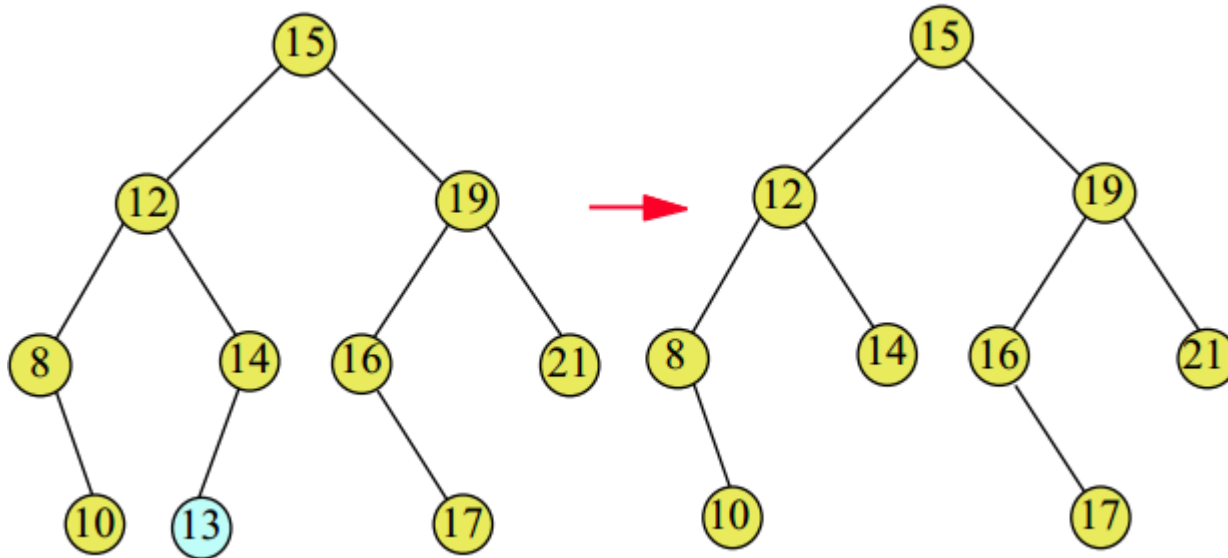
        p->e=5;//affecer une valeur pour l'element

        if(racine==NULL)
        {
            //initialisation de la racine de l'arbre
            racine=p;
            racine->fg=NULL;
            racine->fd=NULL;
            q=p;
            printf("la valeur de la racine est %d",racine->e);
        }
        else
        {
            q->fg=p;
            p =malloc(sizeof(racine));
            p->e=5+i;
            q->fd=q;
            q->e=i+i*3;
            printf("la val du fils gauche est%d\n",p->e);
            printf("la val du fils droit est%d\n",q->e);
            q=p;
        }
    }
}
```


Les opération sur les arbres

Suppression

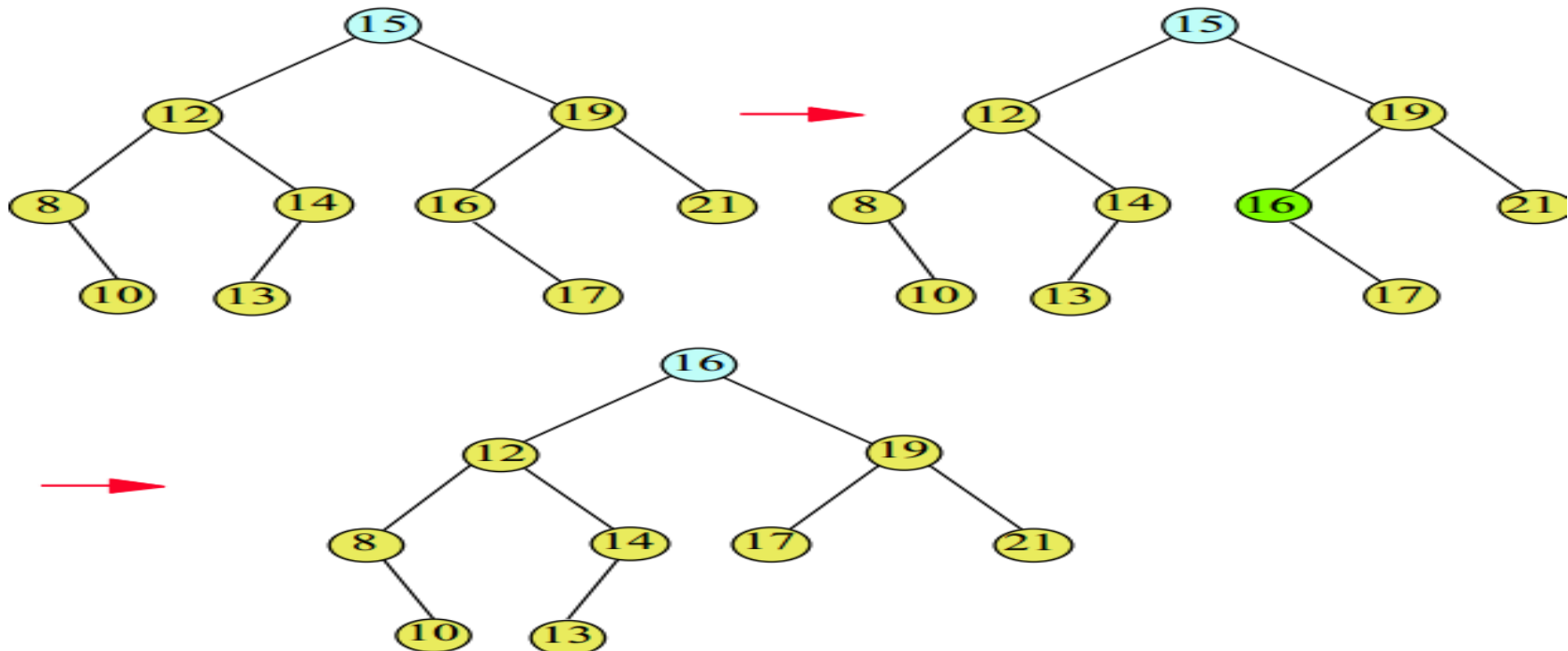
Si le nœud est une feuille : on le supprime



Les opérations sur les arbres

Si le nœud a deux fils : on supprime son successeur qui n'a pas de fils gauche.

On remplace la valeur de 15 par celle de 16.



Les opération sur les arbres

Algorithme de suppression dans arbre binaire

```
Supprimer(Arbre *arbre, Element val)
{ Arbre a = *arbre;
  Si a->fg = null et a->fd=null alors
    arbre = null;
  sinon
    si (val < a->element) alors Supprimer(&a->fg, val);
    sinon si(val > a->element) alors
      Supprimer( &a->fd,val);
    sinon si (a->fg = null) *arbre = a->fd;
      sinon si (a->fd = null) alors *arbre = a->fg;
      sinon (*a)->val = SupprimerSucc(&a);
    fin
  fin
}
```

Les opération sur les arbres

Algorithme de suppression

```
Element SupprimerSucc (Arbre *arbre)
{
    Arbre a = *arbre;
    Element val;

    if (a->fg == NULL )
    {
        val = a->valeur;
        a = NULL ;
        return val;
    }
    return SupprimerSucc (&(a->fg)) ;
}
```

Les algorithmes de parcours



Il existe 2 types de parcours

Parcours en largeur

- **Principe**

Parcours de l'arbre par niveau en partant du niveau zéro. Chaque niveau est parcouru de gauche à droite.

- Visiter un nœud;
- Mettre les fils, Gauche et Droit, non vides de ce nœud dans une file d'attente;
- Traiter le prochain nœud de la file d'attente;
- L'algorithme s'arrête lorsque la file d'attente est vide;

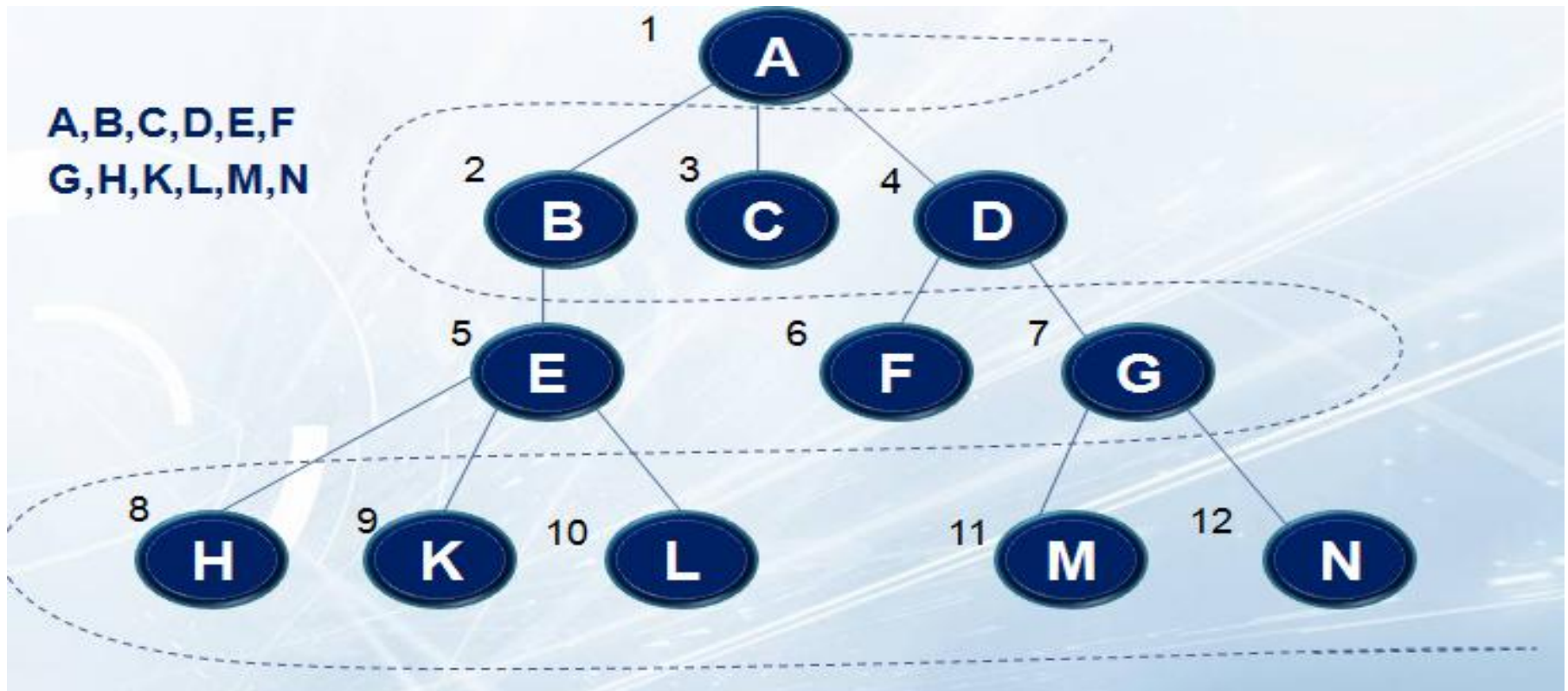
Parcours en largeur

- **Remarque**

- Au début, la file d'attente ne contient rien, nous y plaçons donc la racine de l'arbre que nous voulons traiter.
- Lorsque la file d'attente est vide, cela veut dire qu'aucun des nœuds parcourus précédemment n'avait de sous-arbre Gauche ni de sous-arbre Droit. Par conséquent, on a donc bien parcouru tous les nœuds de l'arbre.

Parcours en largeur

Exemple illustratif



Parcours en largeur

- Algorithme itératif de parcours en largeur

```
Parcours(Arbre a, file s)
{
    s.Ajouter(a); // encore à traiter
    while ( ! s.EstVide() )
    {
        Arbre t = s.Enlever(); // On traite un sommet
        if (t != null)
        {
            printf("%d", t->val);
            s.Ajouter(t->gauche); // Et on stocke le travail futur
            s.Ajouter(t->droite);
        }
    }
}
```

Parcours en profondeur

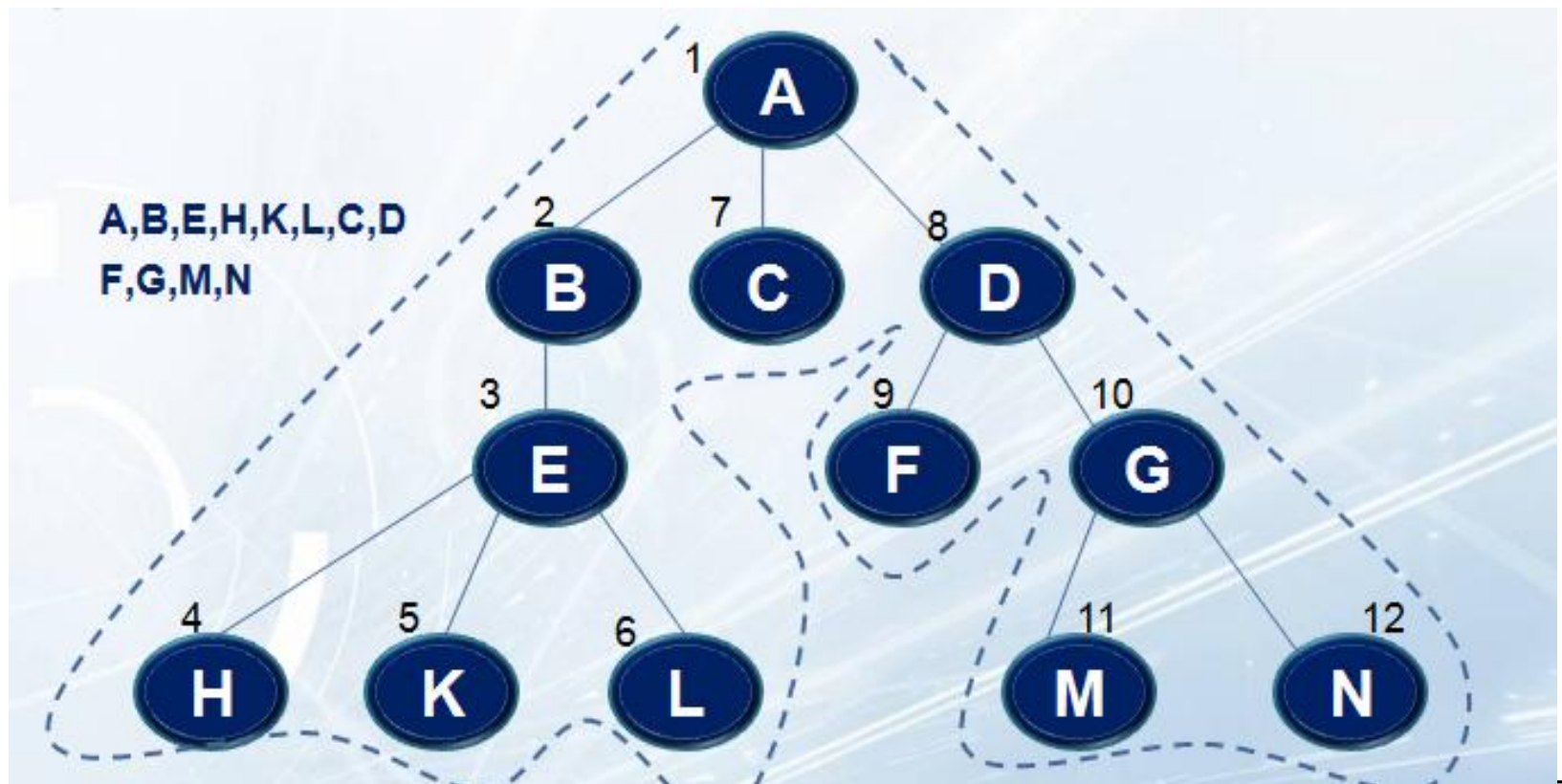
- **Principe**

On commence par la racine, puis son 1^{er} Fils, puis le 1^{er} Fils du 1^{er} Fils, ...

Quand on atteint une feuille, on revient en arrière jusqu'à trouver un Fils non encore parcouru.

Parcours en profondeur

- Exemple illustratif



Parcours en profondeur

- Les différentes façon de parcours en profondeur

❑ Parcours RGD

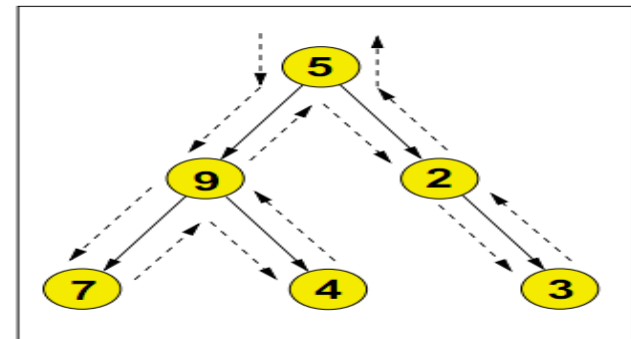
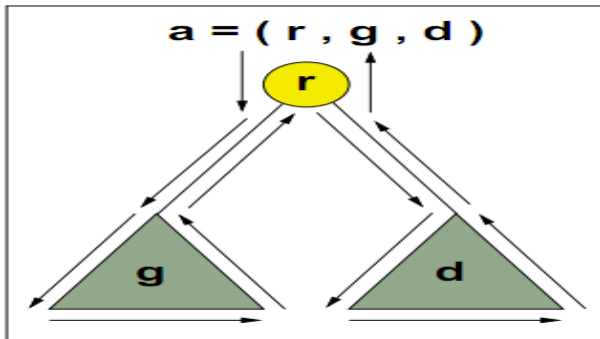
- traiter la racine
- parcours RGD de g
- parcours RGD de d
- $\rightarrow (5, 9, 7, 4, 2, 3)$

❑ Parcours GRD

- parcours GRD de g
- traiter la racine r
- parcours GRD de d
- $\rightarrow (7, 9, 4, 5, 2, 3)$

❑ Parcours GDR

- parcours GDR de g
- parcours GDR de d
- traiter la racine r
- $\rightarrow (7, 4, 9, 3, 2, 5)$



Parcours en profondeur

- **Un parcours en profondeur à gauche**

- Visiter le sous-arbre de gauche puis le sous-arbre de droite donne deux types de parcours :
 - un parcours à gauche,
 - un parcours à droite.

- ❖ **Remarque**

Dans la plupart des cas, nous utiliserons un parcours à gauche.

Parcours en profondeur

- **Un parcours en profondeur à gauche**

- **Parcours préfixe**

- Traiter la racine de l'arbre,
- Traiter le sous-arbre de gauche,
- Traiter du sous-arbre de droite.

Ce type de parcours peut être résumé en trois lettres :

R G D (pour Racine Gauche Droit).

Le parcours en profondeur

- **Un parcours en profondeur à gauche**

- **Parcours infixe**

- Traiter le sous-arbre Gauche,
- Traiter la racine,
- Traiter le sous-arbre Droit.

C'est donc un parcours G **R** D

Le parcours en profondeur

- **Un parcours en profondeur à gauche**

- **Parcours postfixe (suffixe)**

- Traiter le sous-arbre Gauche,
- Traite le sous-arbre Droit,
- Puis la racine.

C'est donc un parcours G D **R**

Algorithme de parcours en préfixe

```
typedef struct Noeud
{
    Element contenu;
    struct Noeud *filsG;
    struct Noeud *filsD;
} *Arbre;
```

```
ParcoursPrefixe (Arbre a)
{
    if (a != NULL)
    {
        Traiter _noeud(a->valeur)
        ParcoursPrefixe (a->filsG) ;
        ParcoursPrefixe (a->filsD) ;
    }
}
```

Algorithme de parcours en préfixe

```
typedef struct Noeud
{
    Element contenu;
    struct Noeud *filsG;
    struct Noeud *filsD;
} *Arbre;

ParcoursPrefixe (Arbre a)
{
    if (a != NULL)
    {
        Traiter _noeud(a->valeur)
        ParcoursPrefixe(a->filsG);
        ParcoursPrefixe(a->filsD);
    }
}
```

Algorithme de parcours en infixe

```
ParcoursInfixe(Arbre a)
{
    if (a != NULL)
    {
        ParcoursInfixe(a->filsG) ;
        Traiter_nœud(a->valeur) ;
        ParcoursInfixe(a->filsD) ;
    }
}
```

Algorithme de parcours en suffixe

```
ParcoursSuffixe(Arbre a)
{
    if (a != NULL)
    {
        ParcoursSuffixe(a->filsG) ;
        ParcoursSuffixe(a->filsD) ;
    }
}
```

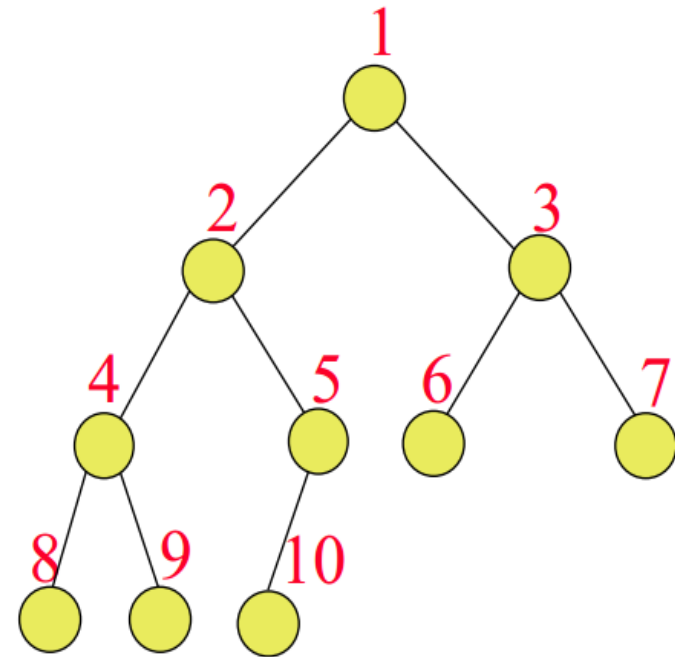
Parcours en profondeur

Exemple 1

Préfixe : 1, 2, 4, 8, 9, 5, 10, 3, 6, 7

Infixe : 8, 4, 9, 2, 10, 5, 1, 6, 3, 7

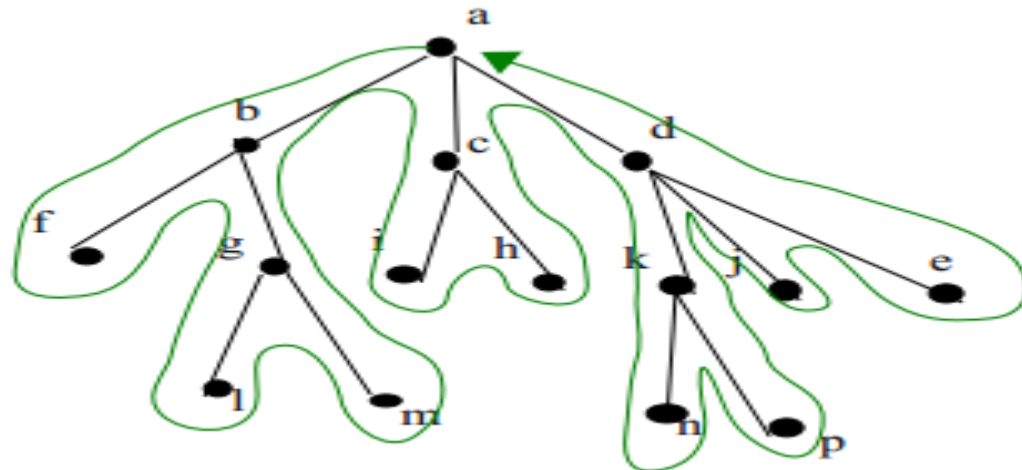
Suffixe : 8, 9, 4, 10, 5, 2, 6, 7, 3, 1



Parcours en profondeur

Exemple 2

- Préfixe (a,b,f,g,m,c,i,h,d,k,n,p,j,e)
- Postfixe (f,l,m,g,b,i,h,c,n,p,k,j,e,d,a)
- Infixe (f,b,l,g,m,a,i,c,h,n,k,p,d,j,e)



Parcours en profondeur infixe

Exemple

La pile des appels profondeur()

(le sommet – l'exécution courante est le dernier élément de la liste)

Debut

APPEL 1 .p(A) au début la pile contient la racine

APPEL 2 .p(A), p(B)

APPEL 3 .p(A), p(B), p(F)

On traite la feuille F

-dépiler F

FIN APPEL 3

On traite la feuille B

-dépiler B

FIN APPEL 2

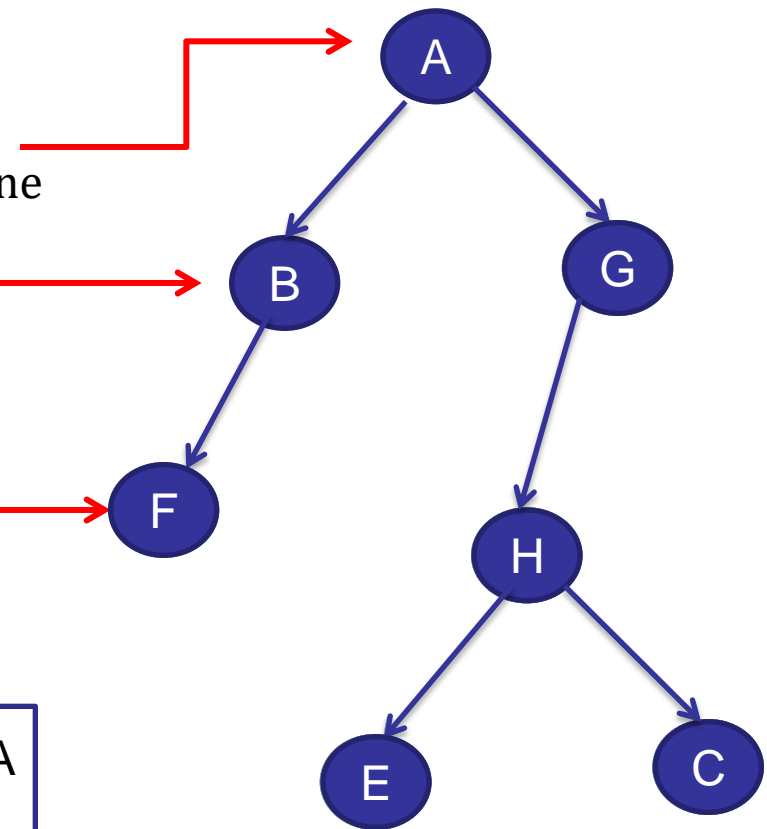
On traite la feuille A

-dépiler A

FIN APPEL 1

APPEL 2 : Fils Droit

P(A): empiler A
dans la pile P



Parcours en profondeur

Exemple (suite)

Début de l'appel Fils droit

APPEL 2 .p(G)

APPEL 3. p(G), p(H)

APPEL 4. p(G), p(H), p(E)

On traite la feuille E

-dépiler E

FIN APPEL 4

On traite la feuille H

dépiler H .

APPEL 5. p(G), p(C)

On traite la feuille C

-dépiler C

FIN APPEL 5

FIN APPEL 3

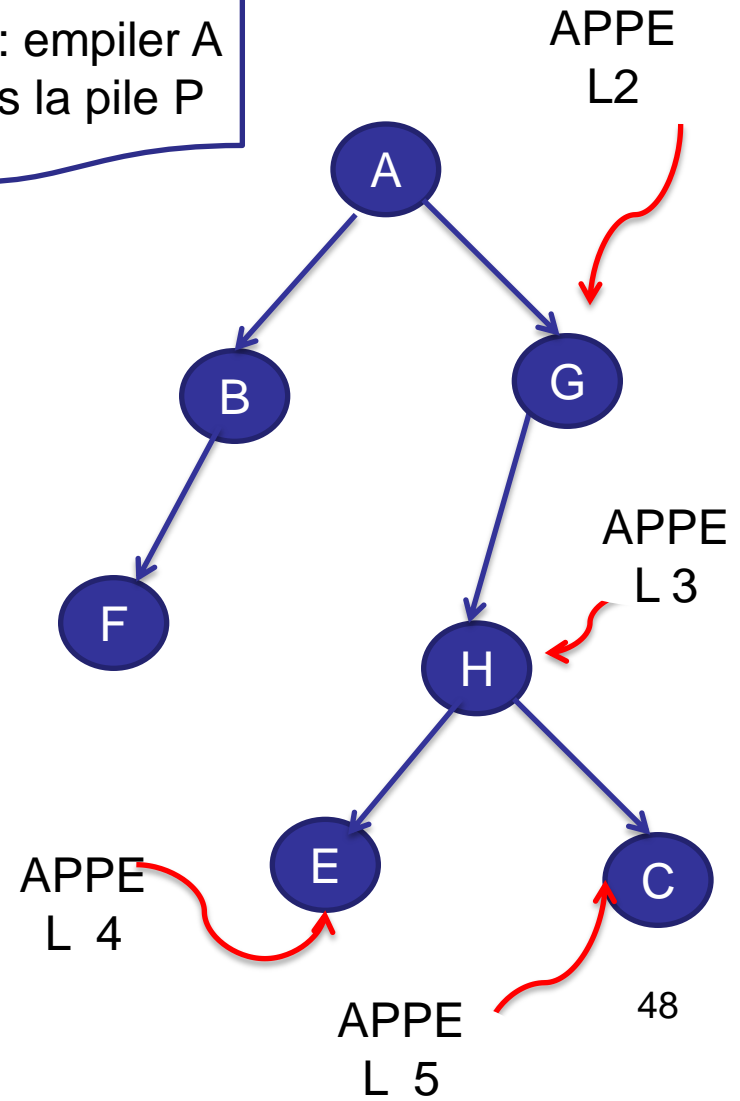
FIN APPEL 2

On traite la feuille G

-dépiler G

FIN APPEL 1

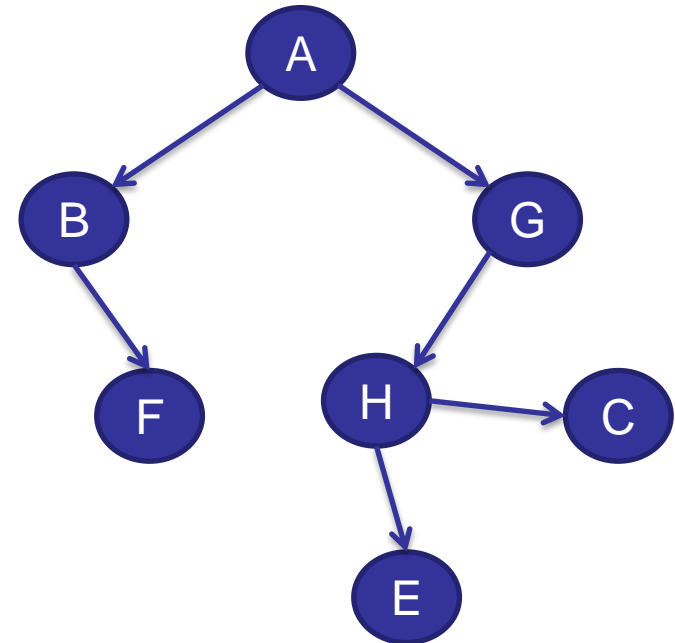
P(A): empiler A
dans la pile P



Parcours en profondeur

- Après l'exécution nous allons voir que les nœuds sont traités avec cet ordre en utilisant le parcours infixe:

F,B,A,E,H,C,G,



Algorithme de parcours préfixé itératif

- Il faut utiliser une pile pour préserver les valeurs successives de la racine et passer au sous arbre droit à chaque retour.
- On utilise une boucle TantQue

Algorithme de parcours préfixé itératif

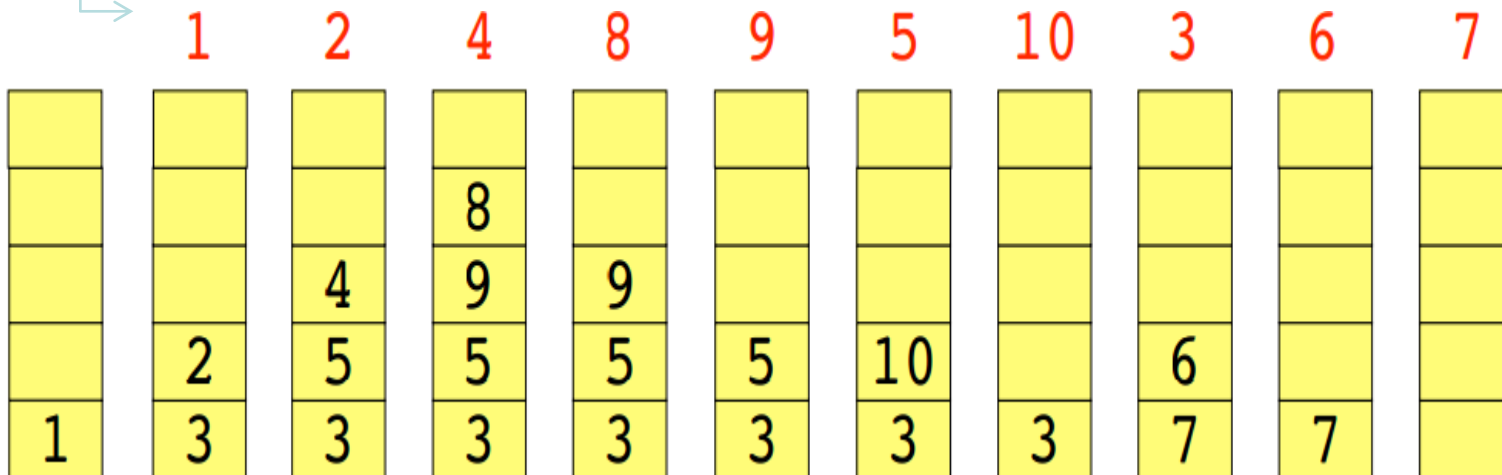
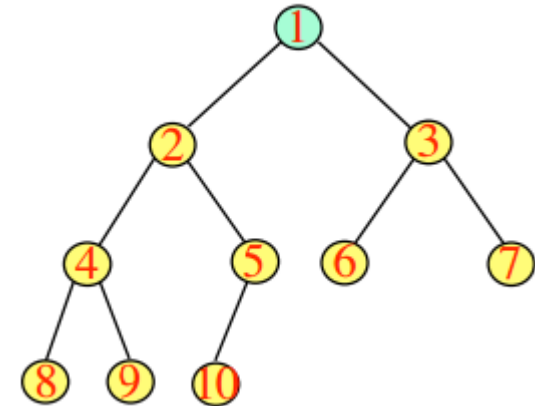
L'algorithme devient ainsi :

```
Empiler la racine;  
TantQue la pile n'est pas vide  
{  
    afficher le sommet de pile et le supprimer;  
    empiler le fils droit;  
    empiler le fils gauche  
}
```

Algorithme de parcours préfixé itératif

Exemple de déroulement

Etat de la pile durant le parcours



Algorithme de parcours préfixé itératif

Voici le programme itératif

```
parcoursPrefixeI(Arbre a)
{
    if (a == null) return(-1);

    Pile p = new Pile();
    p.ajouter(a);
    while ( ! p.estVide() )
    {
        a = p->valeur();
        p.supprimer();
        printf("%d ", a->contenu);
        if (a->fD != null) p->ajouter(a->fD);
        if (a->fG != null) p->ajouter(a->fG);
    }
}
```

Parcours en largeur à l'aide d'une file

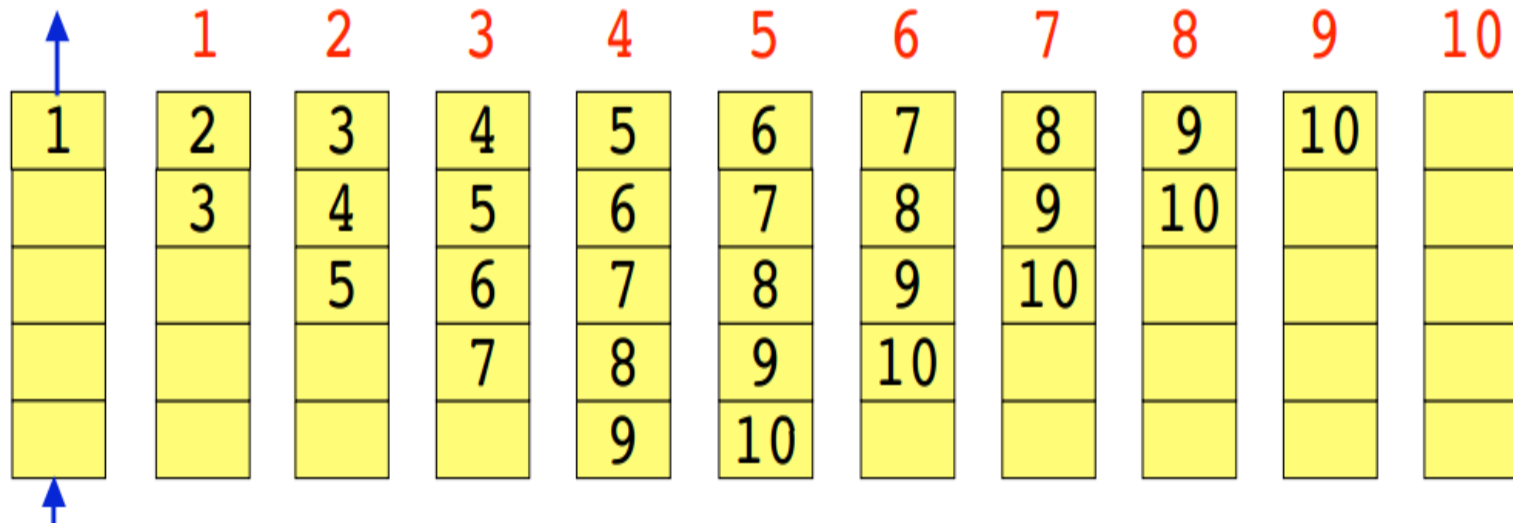
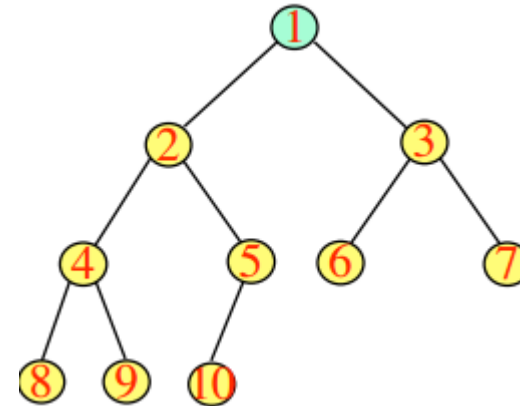
L'algorithme itératif est comme suit

```
Ajouter la racine;  
TantQue la file n'est pas vide  
{  
    afficher le début de la file et le supprimer;  
    ajouter le fils gauche;  
    ajouter le fils droit;  
}
```

Parcours en largeur à l'aide d'une file

Exemple

État de la file



Parcours en largeur à l'aide d'une file

Voici le programme itératif

```
parcoursLargeurI (Arbre a)
{
    if (a == null) return;
    File f = new File();
    f.ajouter(a);
    while ( ! f.estVide() )
    {
        a = f.valeur();
        f .supprimer();
        printf("%d ", a->contenu);
        if (a->fG != null) f->ajouter(a.fG);
        if (a->fD != null) f.ajouter(a->fD);
    }
}
```


Obtention de la forme postfixé

On utilise une pile pour obtenir la forme postfixée.
Voici cette expression arithmétique

$$((6 / (3 - 2 + 1)) * (9 - 6) * 4)$$

Alors la forme postfixé de cette expression est la suivante:

$$6\ 3\ 2\ -\ 1\ +\ /\ 9\ 6\ -\ *\ 4\ *$$

Obtention de la forme post fixé

Pour construire une expression postfixée on suit ces règles: On lit l'expression pas à pas.

(1) Si on a un entier, on l'affiche.

(2) Si on a (, on l'empile avec priorité 0

(3) Si on a), on dépile en affichant jusqu'à (.

(4) Si on a un opérateur +, -, *, / :

TantQue (priorité du sommet de pile \geq priorité du caractère) afficher puis effacer le sommet de pile.

Puis empiler le caractère entrant.

Obtention de la forme postfixé

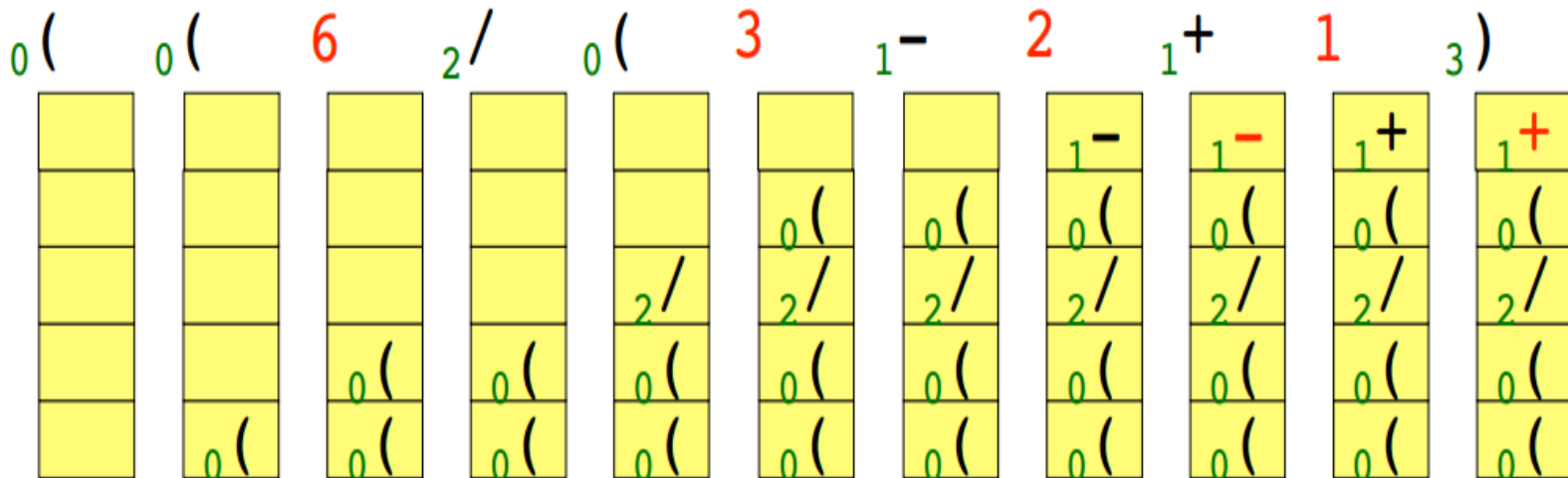
Priorité des opérateurs :

| | | | | | |
|---|---|---|---|---|---|
|) | / | * | + | - | (|
| 3 | 2 | 2 | 1 | 1 | 0 |

Obtenir : 6 3 2 - 1 + / 9 6 - * 2 *

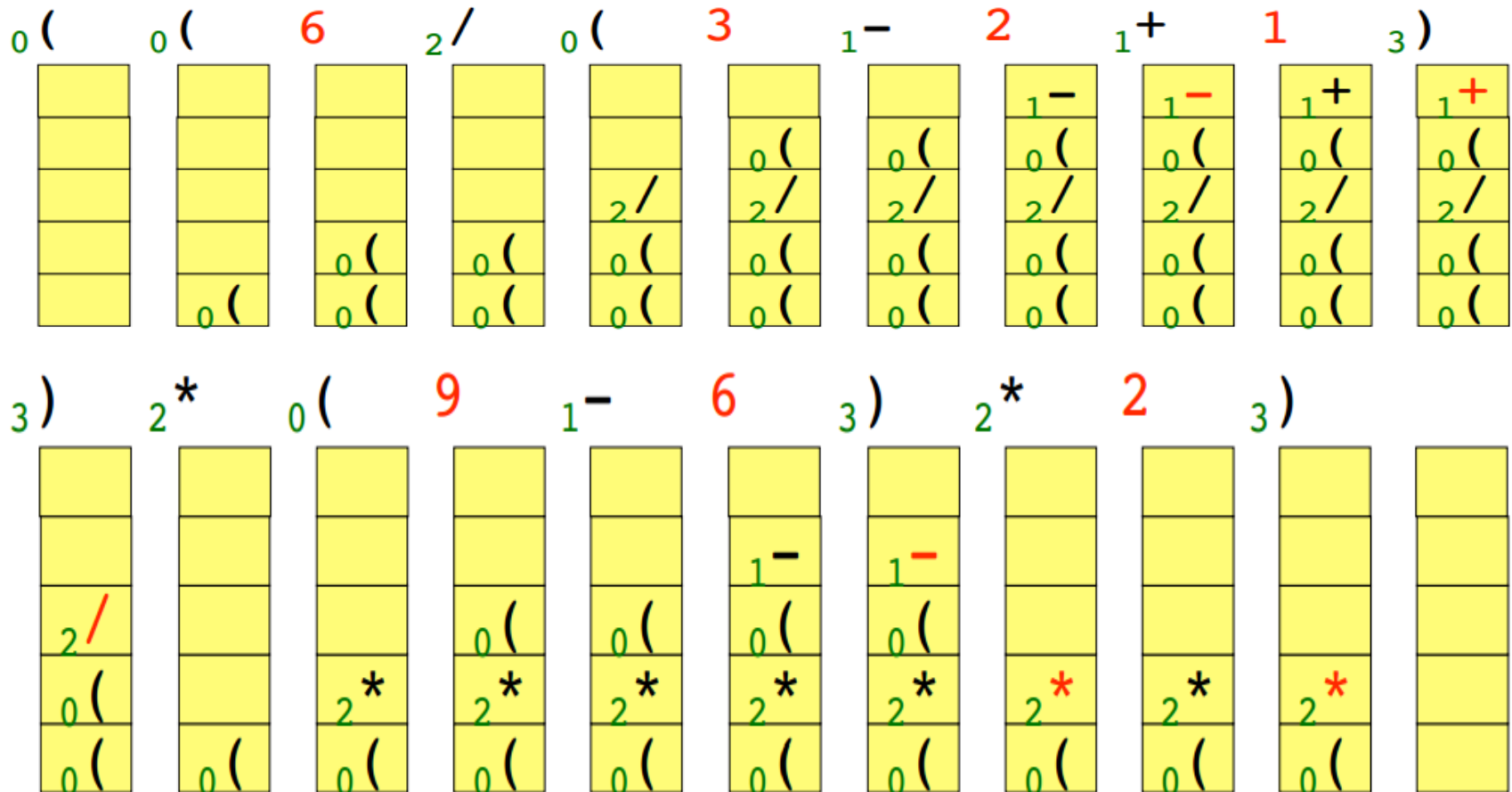
à partir de cette expression

$((6 / (3 - 2 + 1)) * (9 - 6) * 2)$



Obtention de la forme postfixé

Voici l'état de la pile durant la construction de la forme



Obtention de la forme postfixé

Application : (Expressions arithmétiques)

Voici cette expression $((2+(3*5))-7)$

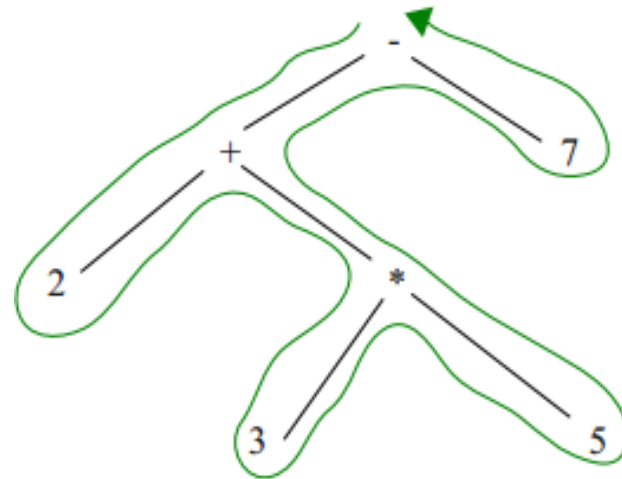
Avec l'Expression infixe on obtient les nœuds visités de cet ordre

$((2+(3*5))-7)$

(car les opérateurs se trouvent entre les opérandes, parenthèses () nécessaires).

parcours postfixe : (2,3,5,*,+,7,-)

(pas besoin de parenthèses () , une seule interprétation)



Annexe 1 – Tri par tas

Fonctions importantes pour le tri

Deux fonctions importantes du tri par tas :

- 1- Construction du tas (Remonter)
- 2- Refaire le tas (Redescendre)

```
Fonction Remonter(k)
Debut
  v ← T[k];
  TantQue v > T[k div 2] Faire
    T[k] ← T[k div 2];
    K ← k div 2;
  FinTQ
  T[k] ← v;
Fin

Fonction Créer_Tas
Debut
  Pour k allant de 2 à n Faire
    Remonter(k);
  FinPour
Fin
```

Annexe 1 – Tri par tas

Fonctions importantes pour le tri

Fonction Redescendre()

Debut

$V \leftarrow T[1];$

$k \leftarrow 1;$

TantQue $k \leq n \text{ div } 2$ Faire

$j \leftarrow 2k;$

si $((j < m) \text{ et } (T[j] < T[j+1]))$ Alors $j \leftarrow j+1;$

$T[k] \leftarrow T[j];$

Si $V < T[j]$ Alors $k \leftarrow j$ Sinon Sortir;

FinTQ

Fin

Fonction Trier(T : Tableau de taille N)

Debut

Créer tas(N);

$m \leftarrow N;$

Pour k allant de 2 à N Faire

$V \leftarrow T[k];$

$T[k] \leftarrow T[1];$

$T[1] \leftarrow V;$

FinPour

Décrementer(m);

Redescendre();

Fin

Annexe 2 – Listes Chainées

Quelques fonctions

```
// definition des types pour LISTE et CELLULE //
typedef struct cellule
{
    int    val;
    struct cellule *suiv;
} CELLULE;

typedef struct
{
    CELLULE    *tete;
    CELLULE    *queue;
} LISTE;
```


Annexe 2 – Listes Chainées

Quelques fonctions

```
LISTE      *CreerListe( int nbCellule )
{
    LISTE      *l;
    CELLULE     *c, *p;
    int         i;

    l = (LISTE *) malloc( sizeof( LISTE ) );

    for ( i = 0; i < nbCellule; i++ )
    {
        c = (CELLULE *) malloc(sizeof( CELLULE ));

        // chainage cellule
        if ( i == 0 )
        {
            l->tete = c;
            p = l->tete;
        }
        else
        {
            p->souv = c;
            p = p->souv;
        }
        // valeur cellule
        c->val = i+1;
    }
    p->souv = 0;
    l->queue = p;

    return( l );
}
```

Annexe 2 – Listes Chainées

Quelques fonctions

```
void    AfficheListe( LISTE *l )
{
    CELLULE *p;

    printf( "\nContenu LISTE:\n" );
    if ( l->tete == 0 )
        printf( "La liste est vide !\n" );
    else
    {
        for( p = l->tete; p != 0; p = p->suiv
)
        {
            printf( "%d\n", p->val );
        }
    }
}
```

Annexe 2 – Listes Chainées

Quelques fonctions

```
LISTE      *FusionListesAvecTri( LISTE *l1, LISTE *l2 )
{
    LISTE      *l;
    CELLULE     *c, *p1, *p2;
    int         val1, val2;

    l = (LISTE *) malloc(sizeof(LISTE ));
    l->tete = 0;
    l->queue = 0;

    p1 = l1->tete;
    p2 = l2->tete;
    while( p1 != 0 )
    {
        c = (CELLULE *) malloc(sizeof( CELLULE ));
        if ( p1 == l1->tete ) l->tete = c;
        if ( l->queue ) l->queue->suiv = c; // chaine nouvelle cellule avec dernière cellule
        l->queue = c;                        // l->queue pointe toujours la dernière cellule
        c->suiv = 0;
        if ( p2 == 0 )
        { // fin l2 //
            c->val = p1->val;
            p1 = p1->suiv;
        }
        else
        { // l2 pas encore finie //
            if ( p1->val <= p2->val )
            {
                c->val = p1->val;
                p1 = p1->suiv;
            }
            else
            {
                c->val = p2->val;
                p2 = p2->suiv;
            }
        }
    }
}
```

```

LISTE      *FusionListes( LISTE *l1, LISTE *l2 )
{
    LISTE      *l;
    CELLULE     *p, *c;

    // init //
    l = (LISTE *) malloc(sizeof( LISTE ));
    l->tete = 0;
    l->queue = 0;

    // fusion l1 et l2
    p = l1->tete;
    while ( p != 0 )
    {
        c = (CELLULE *) malloc(sizeof( CELLULE ));
        if ( p == l1->tete ) l->tete = c;           // première cellule de l1
        else l->queue->suiv = c;
        l->queue = c;                               // l->queue pointe toujours la dernière cellule ajoutée
        c->val = p->val;
        p = p->suiv;
    }

    p = l2->tete;
    while( p != 0 )
    {
        c = (CELLULE *) malloc(sizeof( CELLULE ));
        l->queue->suiv = c; // chainage fin l1 avec début l2
        l->queue = c;       // l->queue pointe toujours la dernière cellule ajoutée
        c->val = p->val;
        p = p->suiv;
    }
    l->queue->suiv = 0;
    return( l );
}

```

```

while( p2 != 0 )
{
    c = (CELLULE *) malloc(sizeof( CELLULE ));
    if ( ( l->tete == 0 ) && ( p2 == l2->tete ) ) l->tete = c;
    if ( l->queue ) l->queue->suiv = c; // chaine nouvelle cellule avec dernière cellule
    l->queue = c;                      // l->queue pointe toujours la dernière cellule
    c->suiv = 0;

    // l1 déjà finie //
    c->val = p2->val;
    p2 = p2->suiv;
}

l->queue->suiv = 0;
return( l );
}

```

```

int main()
{
    LISTE      *l1, *l2, *lst;
    CELLULE     *c, *p, *p1, *p2;
    int         i, taille_liste1, taille_liste2;
    int         val1, val2;

    printf( "\n\nPROGRAMME LISTE CHAINEE - Fusion de 2 listes trie'es\n" );

    // la taille de la liste est saisie au clavier //
    printf("Entrez la taille de la liste 1 a creer: ");
    scanf("%d", &taille_liste1);

    printf("Entrez la taille de la liste 2 a creer: ");
    scanf("%d", &taille_liste2);

    // creation des 2 listes chainees //
    printf("\nCreation des 2 listes...");

    // creation liste 1 //
    l1 = (LISTE *) malloc( sizeof( LISTE ) );

    for ( i = 0; i < taille_liste1; i++ )
    {
        c = (CELLULE *) malloc(sizeof( CELLULE ));

        // chainage cellule
        if ( i == 0 )
        {
            l1->tete = c;
            p = l1->tete;
        }
        else
        {
            p->suiv = c;
            p = p->suiv;
        }
    }
    p->suiv = 0;
    l1->queue = p;

```

```

// creation liste 2 //
l2 = (LISTE *) malloc( sizeof( LISTE ) );

for ( i = 0; i < taille_liste2; i++ )
{
    c = (CELLULE *) malloc(sizeof( CELLULE ));

    // chainage cellule
    if ( i == 0 )
    {
        l2->tete = c;
        p = l2->tete;
    }
    else
    {
        p->suiv = c;
        p = p->suiv;
    }
}
p->suiv = 0;
l2->queue = p;


// saisie des valeurs numeriques des cellules de liste 1 //
printf("\nVeuillez saisir des valeurs entieres croissantes des cellules de liste 1\n");
if ( taille_liste1 == 0 )
    printf( "Saisie ignoree car liste vide !\n" );
else
{
    for ( i = 0, p = l1->tete; i < taille_liste1; i++, p = p->suiv )
    {
        printf("Entrez valeur pour Liste 1, Cellule %d: ", i+1);
        scanf("%d", &p->val);
    }
}

```

```

// saisie des valeurs numeriques des cellules de liste 2 //
printf("\nVeuillez saisir des valeurs entieres croissantes des cellules de liste 2\n");
if ( taille_liste2 == 0 )
    printf( "Saisie ignoree car liste vide !\n" );
else
{
    for ( i = 0, p = l2->tete; i < taille_liste2; i++, p = p->suiv )
    {
        printf("Entrez valeur pour Liste 2, Cellule %d: ", i+1);
        scanf("%d", &p->val);
    }
}

// affichage contenu liste 1 //
printf( "\nAffichage du contenu de la liste 1 :\n" );
if ( taille_liste1 == 0 )
    printf( "La liste est vide !\n" );
else
{
    for ( i = 0, p = l1->tete; p != 0; i++, p = p->suiv )
    {
        printf( "Liste 1, Cellule %d = %d\n", i+1, p->val );
    }
}

// affichage contenu liste 2 //
printf( "\nAffichage du contenu de la liste 2 :\n" );
if ( taille_liste2 == 0 )
    printf( "La liste est vide !\n" );
else
{
    for ( i = 0, p = l2->tete; p != 0; i++, p = p->suiv )
    {
        printf( "Liste 2, Cellule %d = %d\n", i+1, p->val );
    }
}

```



```

// faire la fusiouon des listes 1 et 2 //
lst      = (LISTE *) malloc(sizeof(LISTE ));
lst->tete = 0;
lst->queue = 0;

p1      =      l1->tete;
p2      =      l2->tete;
while ( p1 != 0 )
{
    c = (CELLULE *) malloc(sizeof( CELLULE ));
    if ( p1 == l1->tete ) lst->tete = c;
    if ( lst->queue ) lst->queue->suiv = c;    // chaine nouvelle cellule avec dernière cellule
    lst->queue = c;                          // lst->queue pointe toujours la dernière cellule
        c->suiv = 0;

    if ( p2 == 0 )
    {
        // fin l2 //
        c->val = p1->val;
        p1 = p1->suiv;
    }
    else
    {
        // l2 pas encore finie //
        if ( p1->val <= p2->val )
        {
            c->val = p1->val;
            p1 = p1->suiv;
        }
        else
        {
            c->val = p2->val;
            p2 = p2->suiv;
        }
    }
}

```

```

while( p2 != 0 )
{
    c = (CELLULE *) malloc(sizeof( CELLULE ));
    if ( ( lst->tete == 0 ) && ( p2 == l2->tete ) ) lst->tete = c;
    if ( lst->queue ) lst->queue->suiv = c; // chaine nouvelle cellule avec dernière cellule
    lst->queue = c;                          // lst->queue pointe toujours la dernière cellule
    c->suiv = 0;

    // l1 déjà finie //
    c->val = p2->val;
    p2 = p2->suiv;
}
lst->queue->suiv = 0;

// Affichage contenu liste resultant de la fusion des listes 1 et 2 //
printf( "\nAffichage du contenu de la liste resultat fusion des listes 1 et 2 :\n" );
if ( lst->tete == 0 )
    printf( "La liste est vide !\n" );
else
{
    for ( i = 0, p = lst->tete; p != 0; i++, p = p->suiv )
    {
        printf( "Liste Fusion, Cellule %d = %d\n", i+1, p->val );
    }
}

// fin de programme //
printf( "\n\nFIN.\n\n" );
return 0;
}

```