CHAP 3 - I :

ARBRE BINAIRE

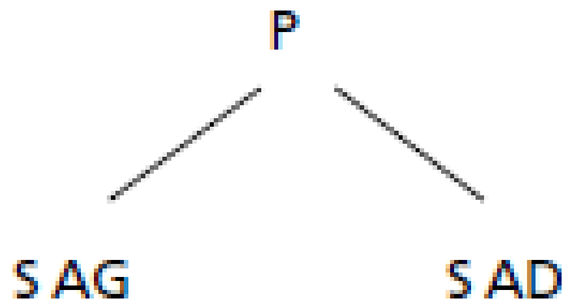
Université Sétif I
Faculté des sciences
Département d'informatique

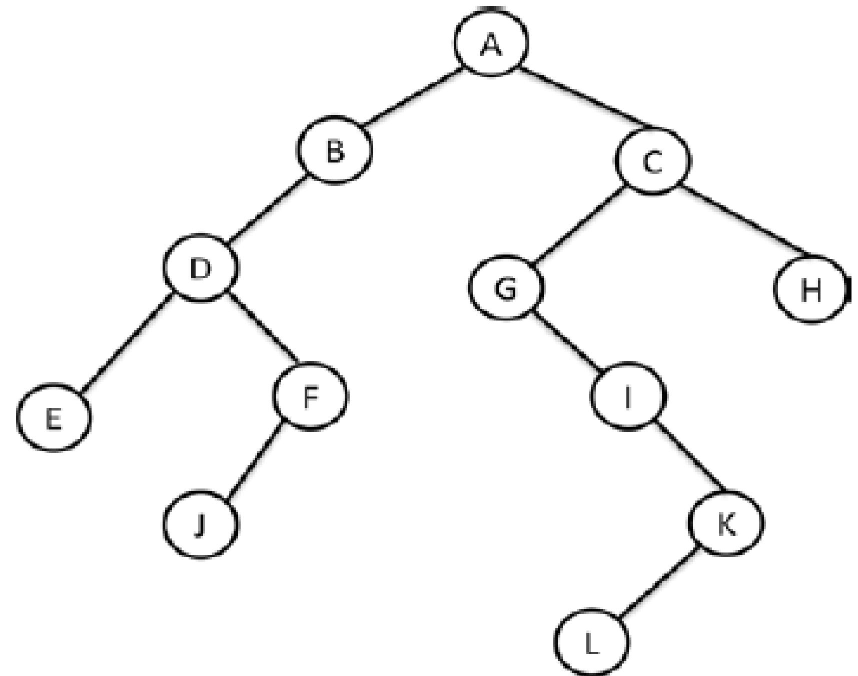
Algorithmique et Structures de Données

2017-2018 / Dr. L.Douidi

Définition

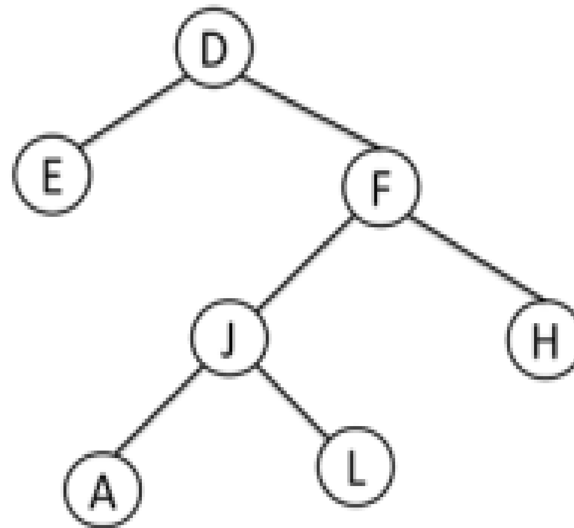
- Un arbre binaire est un arbre où chaque nœud est connecté au maximum à deux sous-arbres (un **sous-arbre gauche (SAG)** et un **sous-arbre droit (SAD)).** Ainsi le premier fils d'un nœud A est appelé **fils-gauche (SAG)** et le deuxième fils est appelé **fils-droit (SAD).**





Arbre strictement binaire

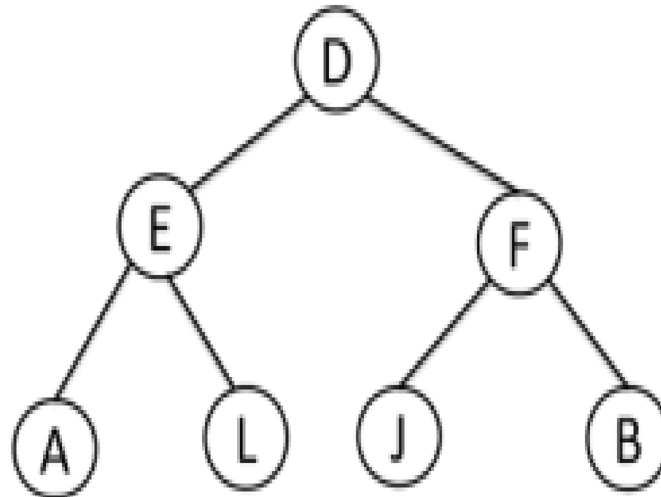
- Un arbre binaire est dit « **strictement binaire** » si chaque nœud interne a exactement **2 fils différents de NULL.**



Dans un arbre strictement binaire, le nombre de feuilles est toujours égal au nombre de nœuds internes + 1. Dans l'exemple, il y a 4 feuilles (E, A, L et H) et 3 nœuds internes (D, F et J).

Arbre complet

- Un arbre binaire est dit « **complet** » (ou **complètement équilibré**), s'il est strictement binaire et si toutes les feuilles se trouvent au même niveau :

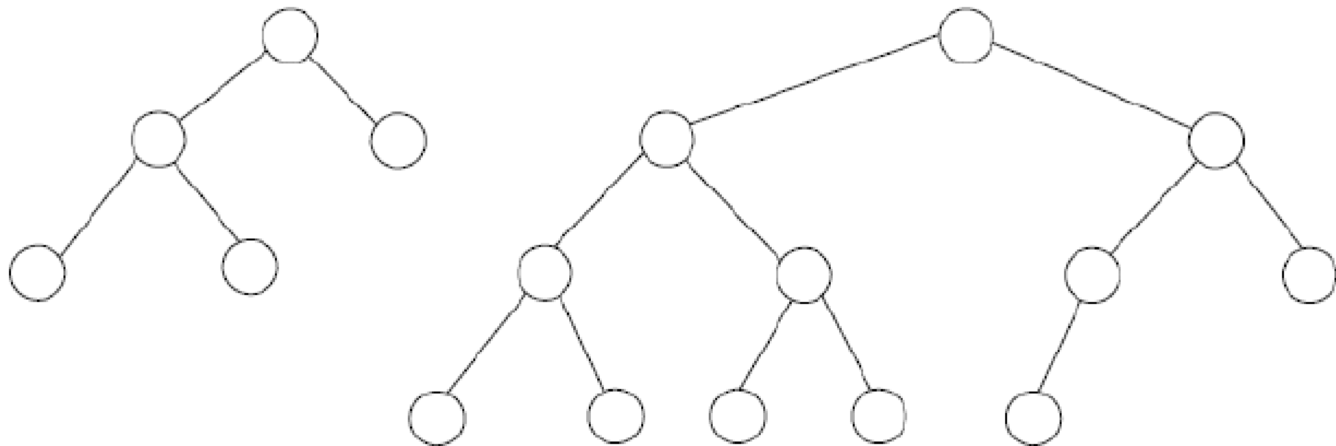


d=3
Nb nœuds= 7
Nb feuilles= 4

Dans un arbre binaire complet de profondeur d :
Le nombre total de nœuds = $2^0 + 2^1 + 2^2 + \dots 2^{d-1} = 2^d - 1$
Le nombre de feuilles = 2^{d-1}
le nombre total de nœuds (n) : $d = \log2(n+1)$

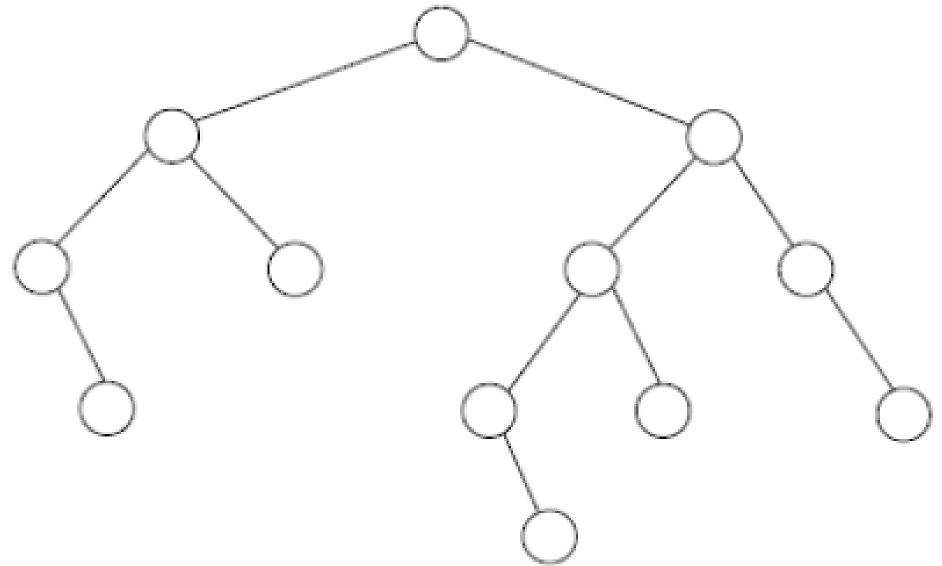
Arbre parfait

- Un arbre binaire est dit **parfait** si, en appelant h la hauteur de l'arbre,
  - les niveaux de profondeur 0, 1, …, h – 1 sont complètement remplis
  - alors que le niveau de profondeur h est rempli en partant de la gauche;
- nous donnons ci-après les arbres binaires parfaits à 5 nœuds et à 12 nœuds



Arbre équilibré

- Un **arbre binaire est dit équilibré** si, pour tout nœud de l'arbre, les sous-arbres gauche et droit ont des hauteurs qui diffèrent au plus de 1. L'arbre ci-dessous est équilibré.



L'arbre vide est équilibré.
Un arbre A, non-vide, est équilibré si et seulement si :
1. chacun de ses sous-arbre est équilibré et
2. pour toute paire A1, A2 de sous-arbres de A,
   | hauteur(A1)−hauteur(A2) | ≤ 1   NB: | valeur absolue |

Opérations sur AB

- Pour pouvoir manipuler des arbres de recherche, on doit donc disposer des fonctionnalités suivantes :

– booléen : est_vide(AB a) : une fonction qui teste si un arbre est vide ou non,

– AB : fils_gauche(AB a) et AB : fils_droit(AB a), des fonctions qui retournent les fils gauche et droit d'un arbre non vide,

– CLE : val(AB a), une fonction qui retourne l'´etiquette de la racine d'un arbre non vide,

– AB : cree_arbre vide(), une fonction qui crée un arbre vide,

– AB cree arbre(CLE v, AB fg, fd), une fonction qui crée un arbre dont les fils gauche et droit sont fg et fd et dont la racine est ´etiquetée par v

– change val(CLE v, AB a), une fonction qui remplace l'´etiquette de la racine de a par v, si a est non vide, et ne fait rien sinon

– change fg(AB fg, AB a) et change fd(AB fd, AB a), des fonctions qui remplacent le fils gauche de a (resp. le fils droit de a) par fg (resp. par fd), si a est non vide, et ne font rien sinon.

- **En C** on utilise en général des pointeurs pour représenter les liens entre un nœud et ses fils dans un arbre binaire.

Quelques fonctions de base pour manipuler un arbre binaire stockant des entiers

```
• Nœud * CréerRacine( Entier i)
début
    Nœud * A := new Nœud ;
    A.info := i ;
    A.fg := NIL ;     A.fd := NIL ;
    retourner A ;
Fin
```

```
• Booléen EstVide(Nœud * A)
début
    si  A = NIL alors retourner vrai ;
    sinon   retourner  faux ;
fin
```

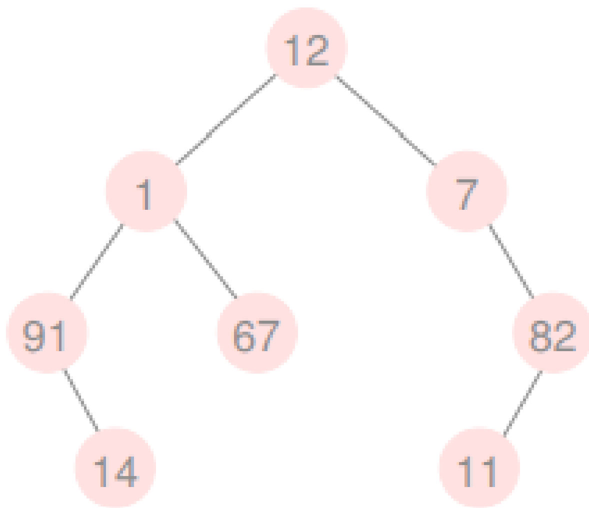
```
• Entier InfoRacine(Nœud * A)
début
    retourner A.info ;
Fin
```

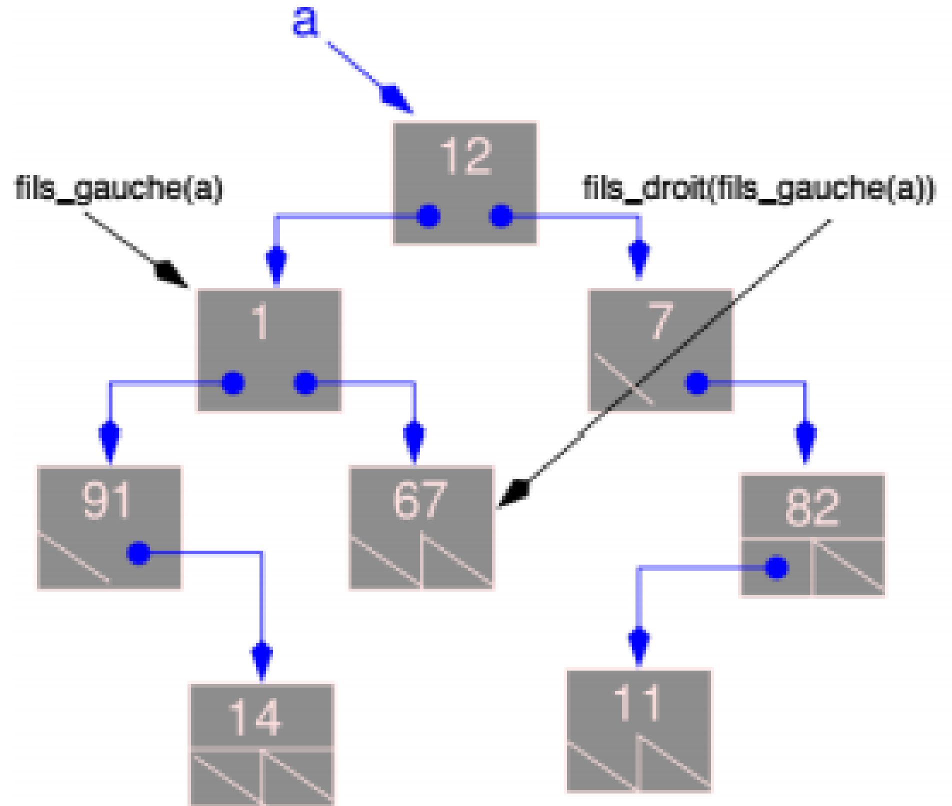
```
• Booléen EstFeuille(Nœud * A)
début
    si EstVide(FilsGauche(A)) et
    EstVide(FilsDroit(A)) alors
        retourner vrai ;
    sinon   retourner faux ;
fin
```

```
• Nœud * FilsGauche(Nœud * A)
début
    retourner A.fg ;
fin
```

```
• Nœud * FilsDroit(Nœud * A)
début
    retourner A.fd ;
fin
```

Arbres binaires : implémentation



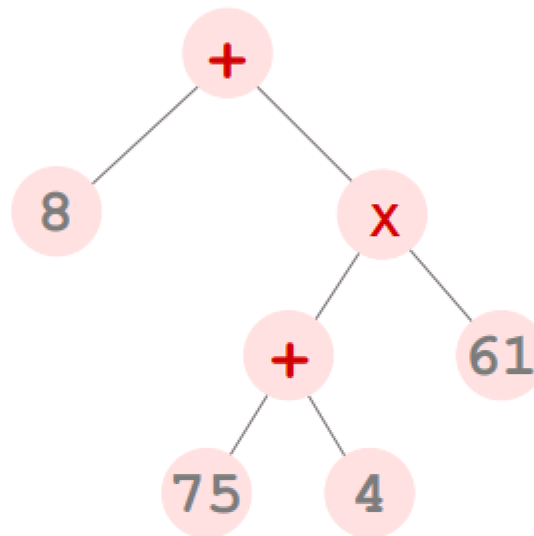


Arbres binaires : représentation

Arbre vide : ∅

Arbre non vide : triplet p = h(*x*,G,D)

- *x* est l'information ou étiquette ou valeur,
- **G** est le sous-arbre gauche de p, noté filsG(p),
- **D** est le sous-arbre droit de p, noté filsD (p).



(+, (8, ∅, ∅), (×, (+, (75, ∅, ∅), (4, ∅, ∅)), (61, ∅, ∅)))

Arbres binaires : implémentation

Définition du type ARBRE, pointeur sur un nœud

```
typedef struct noeud *ARBRE ;

/* Ici éventuellement l'inclusion d'autres définitions
   utilisant le type ARBRE déjà défini */

struct noeud
{
TYPE_VALEUR val;
ARBRE fg, fd;
};
```

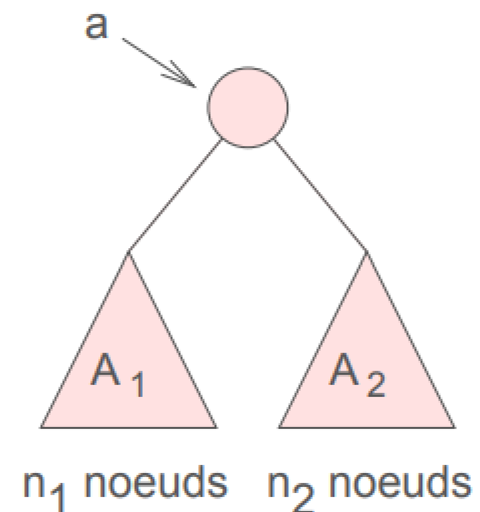
Fonction de création d'un nouveau nœud ARBRE

```
CreerNoeud(VALEUR v, ARBRE fg, ARBRE fd)
{
    ARBRE p;
    p = malloc(sizeof(struct noeud));
    /* vérifier si (p != NULL); */
    p->val = v;
    p->fg = fg;
    p->fd = fd;
    return p;
}
 ARBRE a;
/* dans main */
a = CreerNoeud(x, p, NULL);
```

Arbres binaires : hauteur

la hauteur de l'arbre dépendra des hauteurs des sous-arbre, auxquelles il faudra ajouter 1 pour la racine. Comme la hauteur d'un arbre est définie par rapport à son plus long chemin, il convient de garder la hauteur du plus haut des deux sous-arbre :

- La hauteur(A) d'un arbre A est :
  - 0 si l'arbre est vide
  - 1+max{hauteur((FilsGauche(A)),hauteur((FilsDroit(A))}, sinon



Arbres binaires : hauteur

On voit qu'il faut donc calculer la hauteur des deux fils avant de pouvoir calculer la hauteur de l'arbre. Il s'agit d'un parcours postfixé.

Entier **Hauteur**(Nœud * A)
début
si EstVide(A) alors
  retourner 0 ;
sinon
  Entier hg,hd ;
  hg := **Hauteur**(FilsGauche(A)) ;
  hd := **Hauteur**(FilsDroit(A)) ;
  retourner 1+max{hg,hd} ;
fin

Parcours d 'un arbre binaire

- Trois parcours possibles :

➢ préfixé (préordre): on traite la racine, puis le sous-arbre gauche, puis le sous-arbre droit (RGD)

➢ infixé (projectif ou symétrique) : on traite le sous-arbre gauche, puis la racine, puis le sous-arbre droit (GRD)

➢ postfixé (ordre terminal) : on traite le sous-arbre gauche, le sous-arbre droit, puis la racine (GDR)

```
Parcours(AB  a)
si NON est_vide(a)
    Traitement_prefixe(val(a))     // pour parcours prefixé
    Parcours(fils_gauche(a))
    Traitement_infixe(val(a))      // pour parcours infixé
    Parcours(fils_droit(a))
    Traitement_postfixe(val(a))    // pour parcours postfixé
```

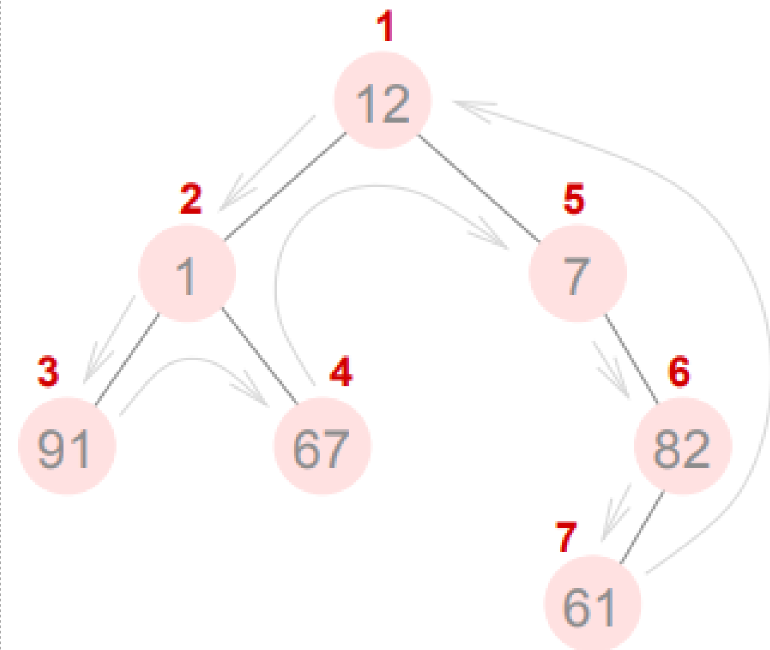
parcours préfixé (affichage)

- Affichage de la valeur d'un nœud avant les valeurs figurant dans ses sous-arbres

```
Affichage_préfixe(AB a)
  si NON est_vide(a)
    Afficher val(a)
    Affichage_préfixe(fils_gauche(a))
    Affichage_préfixe(fils_droit(a))
```

En C

```
void ParcoursPrefixe(ARBRE  a)
{
    if (a != NULL) {
        AfficherLaValeur(a->val);
        ParcoursPrefixe (a->fg);
        ParcoursPrefixe (a->fd);
    }
}
```

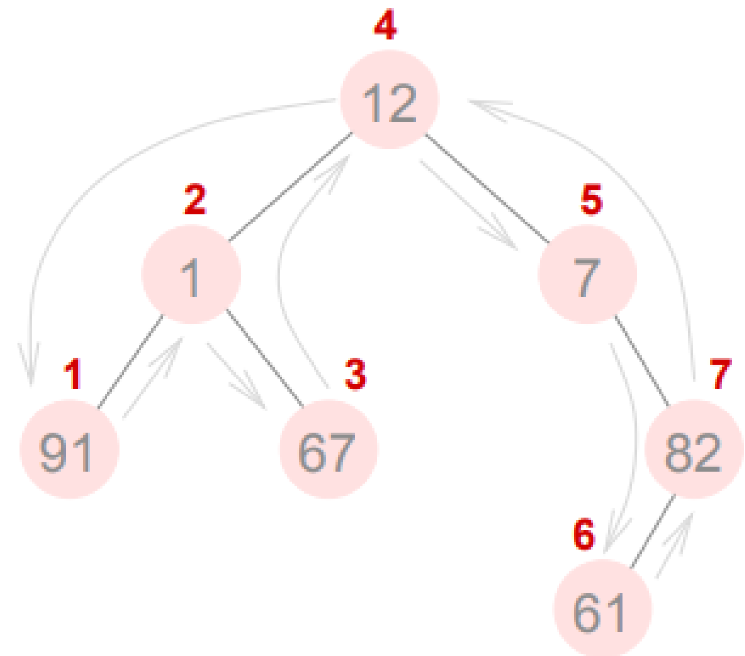


Résultat: 12 1 91 67 7 82 61

parcours infixé (affichage)

- affichage de la valeur d’un nœud après les valeurs figurant dans son sous-arbre gauche et avant les valeurs figurant dans son sous-arbre droit

```
Affichage_infixe(AB a)
si NON est_vide(a) alors
   Affichage_infixe(fils_gauche(a))
   Afficher val(a)
   Affichage_infixe(fils_droit(a))
Fsi
En C
void ParcoursInfixe(ARBRE a)
{
    if (a != NULL)
    {
        ParcoursInfixe (a->fg);
        AfficherLaValeur(a->val);
        ParcoursInfixe (a->fd);
    }
}
```



résultat: 91 1 67 12 7 61 82

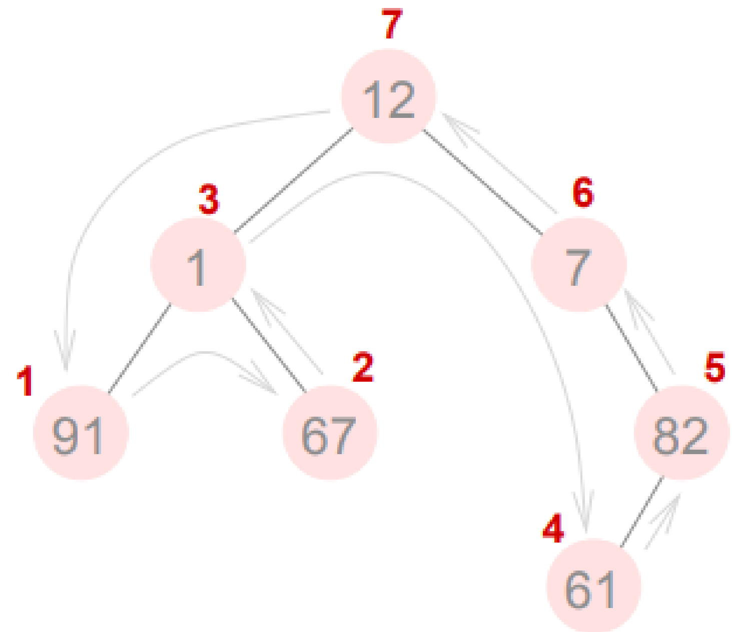
parcours postfixé (affichage)

- affichage de la valeur d'un nœud après les valeurs figurant dans ses sous-arbres

```
Affichage_postfixe(AB a)
si NON est_vide(a)  alors

Affichage_postfixe(fils_gauche(a
))

Affichage_postfixe(fils_droit(a))
   Afficher val(a)
```



Résultat: 91 67 1 61 82 7 12

Le parcours par niveaux ou parcours en largeur

il s’agit de parcourir d’abord tous les nœuds de profondeur 1, puis ceux de profondeur 2, puis ceux de profondeur 3, etc, et ce de gauche à droite.

le parcours par niveaux se déroule comme suit. On traite d’abord le nœud n1. Ce faisant, on rencontre ses deux fils n2 et n3, que l’on met en attente. Ensuite, on traite n2. On rencontre alors ses deux fils n4 et n5 qui doivent, eux aussi être mis en attente (dans une file).

Parcours en largeur

```
Procédure PL( noeud : Pointeur(TNoeud));
Var N : Pointeur(TNoeud) ;
Début
  Si (noeud = Nil) Alors
  InitFile ;
  Enfiler(noeud) ;
  Tant que (Non(FileVide)) faire
    Défiler(N) ;
    Afficher(Valeur(N)) ;
    Si (FG(N) = Nil) Alors
      Enfiler(FG(N)) ;
    Fin Si;
    Si (FD(N) = Nil) Alors
      Enfiler(FD(N)) ;
    Fin Si;
  Fin TQ;
  Fin Si;
Fin;
```

Rechercher K dans A(A.B.)

```
RechercheArbre(Noeud * A, Entier k)
début
   si EstVide(A) alors
         retourner faux ;
   sinon
         si A.info = k alors
                  retourner vrai;
   sinon si RechercheArbre(FilsGauche(A)) alors
                  retourner vrai ;
   sinon si RechercheArbre(FilsDroit(A)) alors
                  retourner vrai ;
   sinon
         retourner faux ;
Fin
```

Taille d'un arbre binaire

- La taille d'un arbre est le nombre de nœuds de cet arbre. La taille à partir du nœud pointé par racine est de 0 si l'arbre est NULL, et vaut 1 (le nœud pointé par racine) plus le nombre de nœuds du sous-arbre gauche et plus le nombre de nœuds du sous-arbre droit sinon.

```
int taille (Noeud* racine) {
    if (racine == NULL) {
            return 0;
    } else {
            return 1 + taille (racine->gauche) + taille (racine->droite);
            }
    }

    // nombre de noeuds de l'arbre
    int taille (Arbre* arbre) {
    return taille (arbre->racine);
    }
```

Feuilles de l'arbre binaire

- La fonction *estFeuille() est une fonction booléenne qui indique si le nœud pointé par* racine est une feuille (n'a pas de successeur).
- // Le nœud racine est-il une feuille ?

```
booleen estFeuille (Noeud* racine)
{
   return (racine->gauche==NULL) && (racine->droite==NULL);
}
```

Nombre de feuilles

- La fonction *nbFeuilles() compte le nombre de feuilles de l'arbre* ***binaire à partir*** du nœud racine. Si l'arbre est vide, le nombre de feuilles est 0 ; sinon si racine

repère une feuille, le nombre de feuilles est de 1, sinon, le nombre de feuilles en partant du nœud racine est le nombre de feuilles du SAG, plus le nombre de feuilles du SAD.

```
static int nbFeuilles (Noeud* racine) {
if (racine == NULL) {
    return 0;
    } else if ( estFeuille (racine) ) {
            return 1;
    } else {
            return nbFeuilles (racine->gauche) + nbFeuilles
    (racine->droite);
    }
}
```

énumère les feuilles de l'arbre binaire

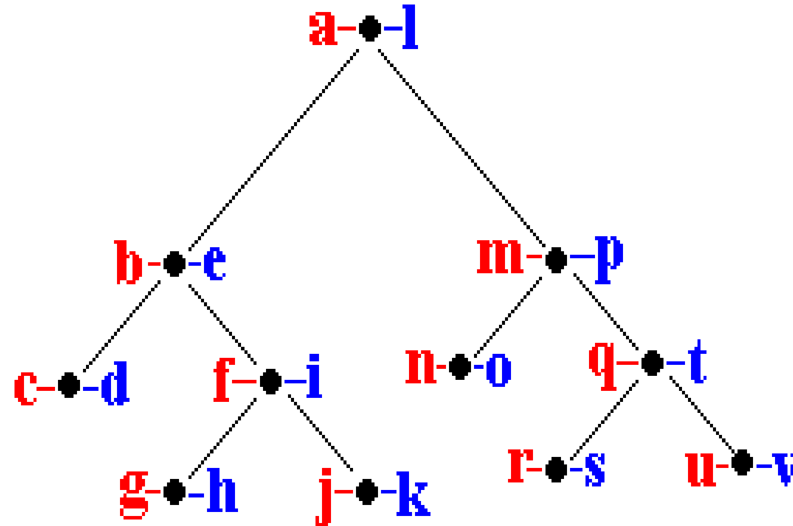
- La fonction *listerFeuilles() énumère les feuilles de l'arbre binaire. C'est un* parcours préfixé d'arbre avec écriture lorsque racine pointe sur une feuille.

```
static void listerFeuilles (Noeud* racine, char* (*toString) (Objet*))
{
if (racine != NULL) {
    if (estFeuille (racine)) {
        printf ("%s ", toString (racine->reference));
    } else {
        listerFeuilles (racine->gauche, toString);
        listerFeuilles (racine->droite, toString);
        }
    }
}

// lister les feuilles de l'arbre binaire
void listerFeuilles (Arbre* arbre)
{   listerFeuilles (arbre->racine, arbre->toString);  }
```

Exercice

- Soit l'arbre suivant possédant 2 attributs par nœuds (un symbole de type caractère)



- On propose le traitement en profondeur de l'arbre comme suit : L'attribut de gauche est écrit en descendant, l'attribut de droite est écrit en remontant, il n'y a pas d'attribut ni de traitement lors de l'examen du fils droit en venant du fils gauche.

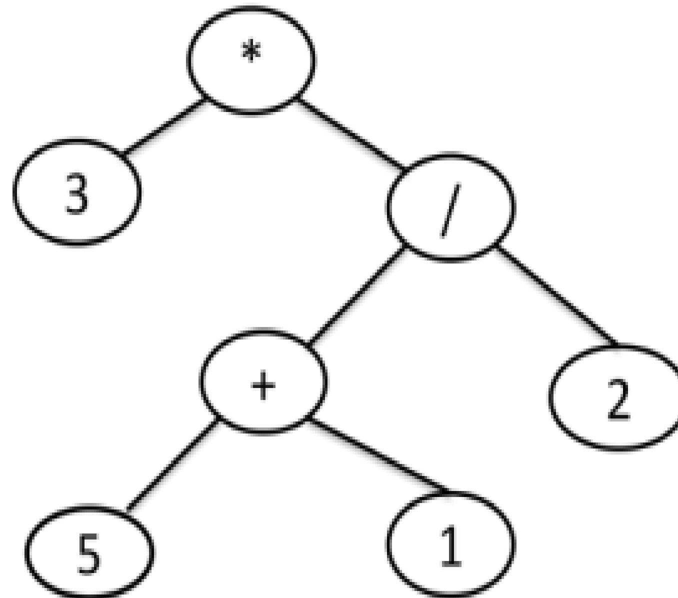
- Écrire la chaîne de caractère obtenue par le parcours ainsi défini.

Réponse : abcdfghjkiemnoqrsuvtpl

Application :

- **Représentation des expressions arithmétiques**
  Les expressions arithmétiques peuvent êtres représentées sous forme d'arbre binaire. Les nœuds internes contiennent des opérateurs, alors que les feuilles contiennent des valeurs (opérandes).

- Par exemple, l'expression 3 * ((5 + 1)/2) sera représentée par l'arbre suivant :



Exercice :

a. Donner une fonction récursive pour évaluer une expression arithmétique sous la forme d'un arbre binaire.

b. Une autre fonction pour imprimer l'expression infixée parenthèsée.

Comment évaluer un arbre binaire

- Pour évaluer le résultat d'un arbre, on applique tout simplement la formule suivante :

  - si le noeud à évaluer contient un entier, le résultat est égal à cet entier.

  - sinon, le résultat est égal à :

    "evalue(FilsGauche) **opération** evalue(FilsDroit) ",

    où "opération" désigne l'un des quatre opérateurs usuels.

- La procédure permettant d'évaluer le résultat de l'expression est la procédure récursive suivante :

```
function arbre.evalue(n: noeud): integer;      //récursive, évidemment
begin  if n.op = 'c' then // c'est une feuille "chiffre"
    evalue := n.valeur
    else  begin    // noeud opérateur
    case n.op of
            '+': evalue := evalue(n.FilsGauche) + evalue(n.FilsDroit);
            '-': evalue := evalue(n.FilsGauche) - evalue(n.FilsDroit);
            '*': evalue := evalue(n.FilsGauche) * evalue(n.FilsDroit);
            '/': evalue := evalue(n.FilsGauche) div evalue(n.FilsDroit);
    end;
    end;
end;
```

Fonction pour évaluer un arbre d'expression

```
Entier ÉvaluationExpression(Nœud * A)
début
   si EstVide(A) alors
         retourner 0 ;
   sinon si EstFeuille(A) alors
         retourner InfoRacine(A) ;
   sinon Entier vg,vd ;
         vg := ÉvaluationExpression(FilsGauche(A)) ;
         vd := ÉvaluationExpression(FilsDroit(A)) ;
   si InfoRacine(A) = + alors retourner vg+vd ;
   sinon si InfoRacine(A) = − alors retourner vg−vd ;
   sinon si InfoRacine(A) = ∗ alors retourner vg ∗ vd ;
   sinon retourner vg/vd ;
fin.
```

afficher une expression stockée dans un arbre

```
Entier AffichageExpression(Nœud * A)
début
si NON EstVide(A) alors
   si EstFeuille(A) alors
      Afficher Info(A) ;
   sinon
      Afficher '(' ;
      AffichageExpression(FilsGauche(A)) ;
      Afficher Info(A) ;
      AffichageExpression(FilsGauche(D)) ;
      Afficher ')' ;
   finsi
finsi
fin
```

Evaluation des expressions arithmétiques

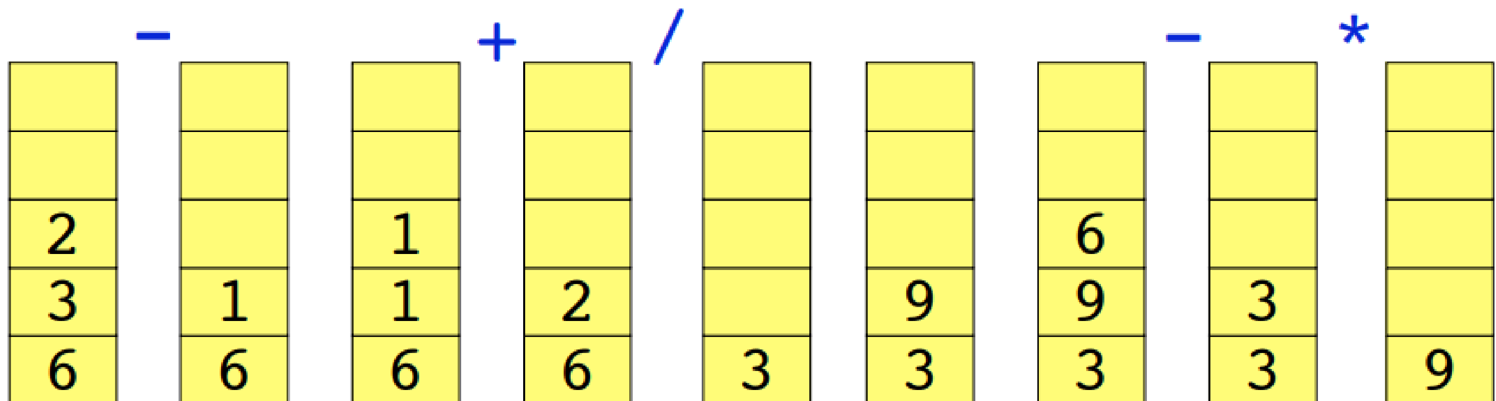
- Beaucoup plus simple sous forme postfixée.

Utilisation d'une pile.

On lit l'expression de la gauche vers la droite :

  - Si un nombre est rencontré, on l'empile
  - Si un opérateur est rencontré, on dépile deux fois, on applique l'opérateur aux deux nombres obtenus et on empile le résultat

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



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

Evaluation des expressions Postfixé

```
entier evalPostfix(caractère expression[1..N])
début
pile p ← vide;
pour(i de 1 àN)
   si(estNombre(expression[i])) alors
         empiler(p, nombre(expression[i]));
   sinon si(estOpérateur(expression[i]) alors
                  op1 ← dépiler(p);
                  op2 ← dépiler(p);
                  résultat ← opère(op1, op2, expression[i]);
                  empiler(p, résultat);
            finsi
   finsi
finpour
retourner (dépiler(p));
fin
```

NB: **estNombre, estOpérateur et** opère : fonctions à développées