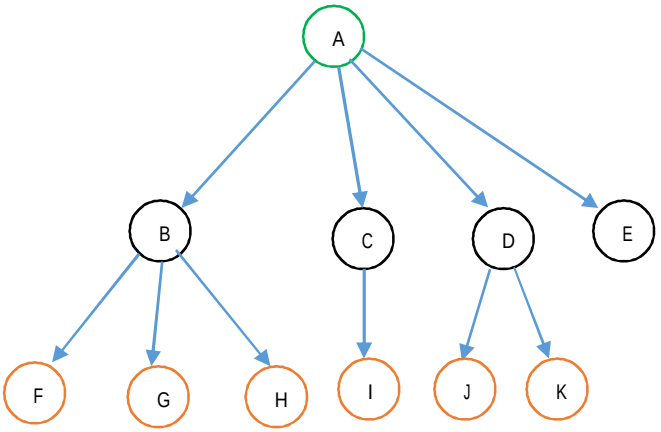
Chapitre III : Les Arbres

1. Introduction

Un arbre est une SDD hiérarchique (non linéaire) composée d'un ensemble de nœuds (maillons) connectés entre eux par des arêtes (père/fils). Chaque nœud contient une information et des pointeurs vers d'autres nœuds (d'autres sous arbres). Ils servent à représenter un ensemble de données structurées hiérarchiquement.
Lors d'un parcours d'un arbre, on a, à chaque élément, plusieurs choix possibles pour « continuer le parcours » : ramification.

L'importance des arbres en informatique est considérable. D'une part, les structures de type arborescent se rencontrent naturellement dans plusieurs cas pratiques de la vie quotidienne que l'on désire modéliser et manipuler à l'aide d'algorithmes. D'autre part, les arbres jouent un rôle très important dans de nombreux systèmes (base de données, système de fichiers, compression MP3, etc.).
Soit l'arbre A suivant :



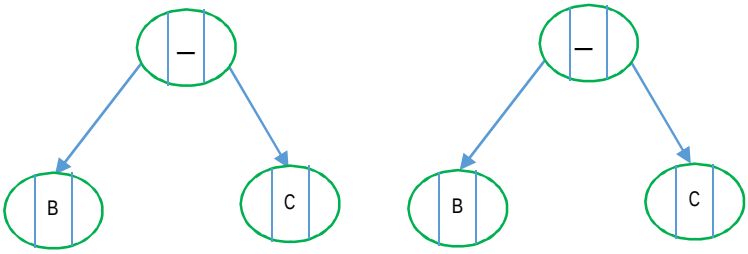
2. Définitions et terminologies

– Le prédécesseur d'un nœud x, s'il existe, s'appelle Père de x (père de C = A, père de G = B) ;
– Les successeurs d'un noud y, s'ils existes, s'appellent Fils de y (fils de A = {B, C, D}, fils de D= {I, J}) ;
– Le nœud qui n'a pas de père s'appelle racine (A) ;
– Les nœuds qui n'ont pas de fils s'appellent feuilles (E,F,G,H,I,J) ;
– Un nœud qui a un prédécesseur et un ou plusieurs successeurs s'appelle nœud intermédiaire (E,F,G,H,I,J) = nœuds internes-racine;

– Un nœud qui a au moins un fils s'appelle nœud interne (A, B, C, D) ;
– Les nœuds qu'on peut atteindre à partir d'un nœud x s'appellent les descendants de x (descendants de D = {I, J}) ;
– Les nœuds à partir desquels on peut atteindre un nœud y s'appellent les ascendants de y (ascendants de F = {A, B};
– Un sous arbre est une portion de l'arbre. Le nœud D est ses deux fils I et J forme un sous graphe de A ;
– Une branche : suite de nœuds passant de père à fils allant de la racine à une feuille. (A, K et A, B, F sont des branches).

3. Propriétés des arbres : nous présentons dans ce qui suit les propriétés les plus importantes concernant les arbres.
– Taille d'un arbre **:** c'est le nombre de nœuds qu'il possède (taille(A)=10) ;
– Niveau d'un nœud **:** c'est sa distance par rapport à la racine (niveau(A)=0, niveau(F)=2). Niveau (fils)=niveau (père) +1 ;

– Profondeur (Hauteur) d'un arbre **:** c'est le nombre d'arcs de la plus grande branche dans l'arbre = niveau maximum dans cet arbre) (hauteur(A)=3) ;
– Degré d'un nœud : le degré d'un nœud x est égal au nombre de ses fils (degré (B)=3) ;
– Degré d'un arbre **: c**'est le degré maximum de ses nœuds (degré(A)=4) ;
– Arbre ordonné **:** un arbre est dit ordonné si l'ordre de ces nœuds est significatif. Par exemple, les deux arbres ci-dessous sont différents car B-A ≠ A-B ;

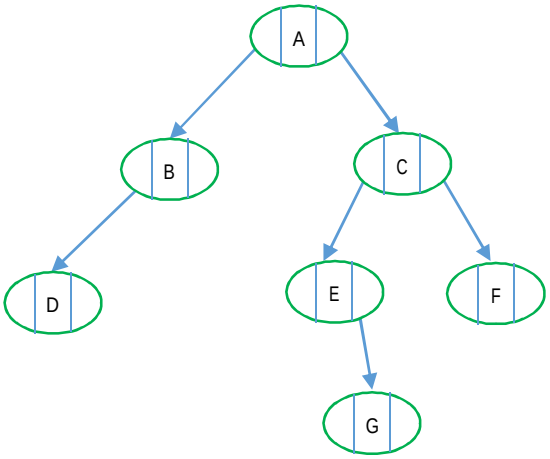


– Arbre balancé : un arbre est dit balancé si toutes ses branches ont la même longueur. L'arbre précédent n'est pas balancé car longueur de la branche (A, K) = 1 est différente de la longueur des autres branches qui est égale à 2 ;
– Arbre n-aire : un arbre n-aire est un arbre ou le degré maximum d'un nœud est égal à n.

4. Les arbres binaires

4.1. Définitions

– Arbre binaire **:** est un arbre dont le degré est égale à 2 : chaque nœud a au plus deux fils (gauche et droite). Soit l'arbre binaire suivant.

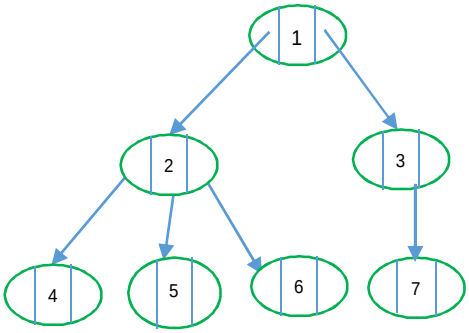


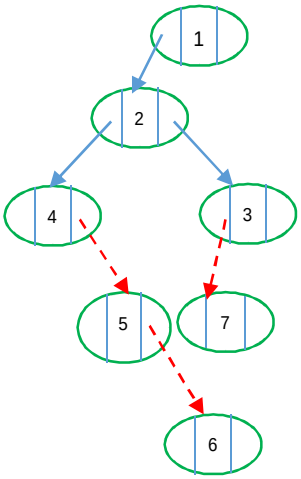
– Arbre binaire complet : un arbre binaire est dit complet si tous ses nœuds internes ont 2 fils et ses feuilles sont au même niveau (balancé). Dans un arbre binaire complet de profondeur d, le nombre total de nœuds=2^{d+1}-1, le nombre de feuilles=2^{d} et le nombre de nœuds internes=2^{d}-1.

4.2. Transformation d'un arbre quelconque en un arbre binaire équivalent.

Lorsque le nombre de successeurs des nœuds est variable, il est souvent préférable de convertir l'arbre en un arbre binaire équivalent. On mémorise alors pour chaque nœud, un pointeur vers son premier fils à gauche et un pointeur vers son frère immédiatement plus proche. Chaque nœud n dans l'arbre initiale correspondant au nœud n' dans l'arbre binaire : le fils gauche de ***n'*** est le nœud correspondant au premier fils de ***n***, et le fils droit de ***n'*** est le nœud correspondant au prochain frère de **n**.
Méthode pour chaque nœud x de l'arbre :
1- Détacher tous ses fils sauf celui le plus à gauche ;
2- Chainer (connecter) tous les fils de x ;
3- Incliner chaque ligne horizontale de 45°

Exemple Soit l'arbre a suivant :





4.3. Implémentation des arbres : Ils être représentés par des tableaux (statique) ou par des listes non linéaires (dynamique).

– Représentation dynamique : Chaque nœud comporte deux champs. Un champ de type simple contenant une information et un champ de type tableau contenant des pointeurs vers ses fils.

```
type Arbre=^Nœud ;
     Nœud = Enregistrement
                 info : Typeqq ;
                 fils : Tableau [1..NbFils] ^Nœud;
            fin ;
Var  a : Arbre ;
```

4.4. Implémentation des arbres binaires :

La structure d'un nœud de l'arbre binaire est la suivante :

```
type Arbre=^Nœud ;
     Nœud = Enregistrement
                 info : entier ;
                 FG : Arbre ;
                 FD : Arbre ;
            fin ;
Var  a : Arbre ;
```

4.5. Traitements sur les arbres binaires : Le parcours d'un arbre est l'un des traitements les plus importants. Il consiste à parcourir tous ses nœuds afin d'effectuer un ensemble de traitements sur cet arbre. On distingue deux types de parcours :

a- Parcours en profondeur : consiste à explorer l'arbre branche par branche. On distingue 3 types de parcours :

– Parcours Préordre (préfixe) **:** consiste à visiter d'abord la racine R ensuite parcourir récursivement les sous arbres gauche G et droit D, ce qui donne [RGD].

La procédure récursive qui affiche les valeurs en parcours préfixe d'un arbre de racine a est :

```
Procédure préfixe( a : Arbre ) ;
début
    si a <> nil alors
       écrire( a^.info) ;
       préfixe(a^.FG) ;
       préfixe(a^.FD) ;
    finsi ;
fin ;
```

– Parcours Inordre (infixe) : consiste d'abord à parcourir récursivement sous arbre gauche G, puis visiter la racine R, ensuite parcourir récursivement le sous arbre droit D, ce qui donne [GRD].

```
Procédure infixe( a : Arbre ) ;
début
    si a <> nil alors
        infixe (a^.FG) ;
        écrire( a^.info) ;
        infixe (a^.FD) ;
    finsi ;
fin ;
```

– Parcours Postordre (postfixe) : consiste d'abord à parcourir récursivement les sous arbres gauche G et droit D ensuite visiter le nœud racine R, ce qui donne [GDR].

```
Procédure postfixe( a : Arbre ) ;
début
    si a <> nil alors
        postfixe (a^.FG) ;
        postfixe (a^.FD) ;
        écrire( a^.info) ;
    finsi ;
fin ;
```

b- Parcours en largeur : consiste à explorer l'arbre niveau par niveau. Le parcours de l'arbre précédent en largeur donne l'ordre suivant : A, B, C, D, E, F, G

4.6. Opérations sur les AB

a- fonction ''vide'' **:** permet de vérifier si l'arbre est vide ou non.

```
fonction vide(a :arbre) :booléen ;
début
    si a=nil alors
        vide←vrai
    sinon
        vide←faux ;
    finsi ;
fin ;
```

b- fonction ''feuille'' **:** permet de vérifier si un nœud q est une feuille ou non.

```
fonction feuille(q :arbre) :booléen ;
début
    si q^.FG=nil et q^.FD=nil alors
        feuille←vrai
    sinon
        feuille←faux ;
    finsi ;
fin ;
```

c- fonction ''taille'' **:** permet de calculer le nombre de nœuds de l'arbre.

```
fonction taille(a :arbre) :entier ;
début
    si a =nil alors
        taille←0
    sinon
        taille←1+taille(a^.FG)+taille(a^.FD);
    finsi ;
fin ;
```

d- procédure ‘‘rechercher’’ **:** permet de vérifier si une valeur **val** existe dans un arbre **a**.

```
procédure rechercher(a :arbre, val : entier ; trv :booléen);
début
    si a ≠nil alors
        si a^.info=val alors
            trv←vrai
        sinon
            rechercher(a^.FG,val, trv)
        sinon si trv=faux alors
            rechercher(a^.FD,val, trv)
        finsi ;
    finsi ;
fin ;
```

```
fonction rechercher(a :arbre, val : entier) :booléen ;
début
    si a =nil alors
        rechercher←faux
    sinon si a^.info=val alors
        rechercher ←vrai
    sinon
        rechercher←rechercher(a^.FG,val)ou rechercher(a^.FD,val)
    finsi ;
fin ;
```

f- fonction ‘‘hauteur’’ **:** permet de calculer la taille de l’arbre (nombre de niveau.

```
fonction hauteur (a :arbre) :entier ;  //si hauteur=niveau maximum

début

  si a = nil alorst vide then

    hauteur← -1

  sinon

    hauteur←1 + max(hauteur(a^.FG), hauteur (a^.FD)) ;

  finSi ;

fin ;
```

g- fonction ‘‘ajouter **:** permet d’ajouter un nœud à l’arbre. Dans cet exemple, on l’ajoute après la feuille la plus à gauche.

```
fonction ajouter(a :arbre ; val : entier) :entier;
var q, p : arbre ;
début
    allouer(q) ; q^.info←val ; q^.FG←nil ; q^.FD←nil ;
    si a =nil alors
        a←q
   sinon
        p←a ;
        Tq p^.FG≠nil faire
            p←p^.FG
        FTq
        p^.FG←q ;
    finsi ;
fin ;
```

5. Les Arbres de Recherches Binaires (ABR)

Un ARB est une SD de base utilisée pour représenter des ensembles dont les éléments sont ordonnés par une relation d'ordre notée <. Ils sont un bon compromis pour un temps équilibré entre l'ajout, la suppression et la recherche.

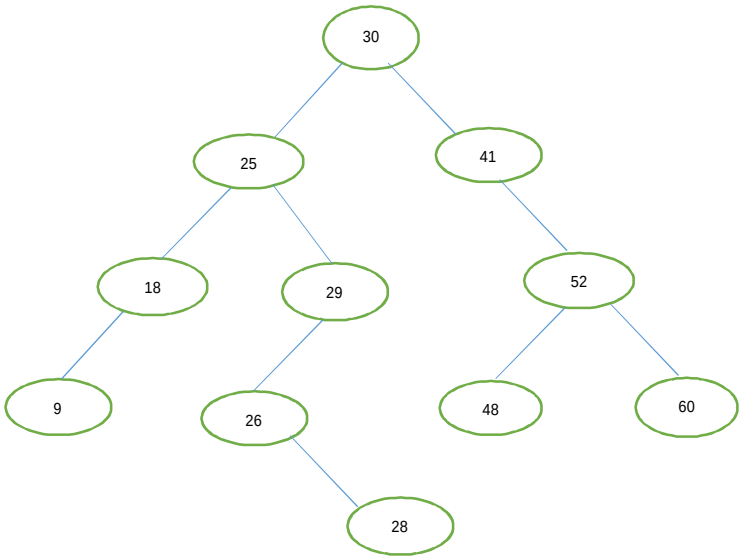
Définition **:** un ABR est un AB vérifiant les 2 propriétés suivantes (pour chaque nœud i) :

- Tous les éléments de son sous arbre de gauche ont des valeurs inférieure à celle du nœud i.
- Tous les éléments de son sous arbre de droit ont des valeurs supérieure à celle du nœud i.

Remarques :

• Toutes les valeurs dans un ARB sont distinctes (il n'y a pas de valeur en double).
• Le parcours en infixe d'un ARB donne la liste ordonnée de tous ses éléments.
• La valeur d'un élément de l'arbre est appelée clé.

Exemple



Le parcours infixe de cet arbre donne la liste ordonnée suivante : 9, 18, 25, 26, 28, 29, 30, 41, 48, 52, 60

Construction d'un ABR à partir d'une liste éléments : Il suffit d'insérer chaque élément dans l'arbre binaire à sa bonne place de telle sorte que l'arbre binaire soit toujours un ABR.

Exemple **:** Soit les valeurs de clés triées selon l'ordre d'arrivée **:**
14, 23, 4, 9, 17, 11, 28, 16, 3, 7

5.1. Opérations sur les arbres :

a- fonction "rechercher" **:** La recherche est dichotomique. A chaque étape, un sous arbre est éliminé. Par exemple, si on veut rechercher la valeur 26 alors :
rechercher(FG(30))→rechercher(FD(25))→rechercher(FG(29))→valeur trouvée

```
fonction rechercher(a: arbre; val : entier) : booléen ;
début
   Si a=nil alors
        rechercher←faux
   sinon si val=a^.info alors
        rechercher←vrai
   sinon si val<a^.info alors
        rechercher← rechercher(a^.FG)
   sinon
        rechercher← rechercher(a^.FD) ;
fin ;
```

b- procédure ‘‘insérer’’ : l'insertion d'un nœud n dans l’arbre ***a*** se fait toujours au niveau d'une feuille de telles sorte que l'arbre soit toujours un ABR. L’algorithme procède en 3 étapes :

1- S'assurer que le nœud **n** n'existe pas dans **a** ;
1- Rechercher la position (feuille) d’insertion ;
2- Chaîner le nouveau nœud à son père.

Exemple **:** Soit insérer la valeur 44 :
1- Rechercher la position d’insertion de 44, se positionner sur la feuille 48 ;
2- Insérer 44 à gauche de la feuille 48.

```
fonction  inserrer(var a: arbre; q : arbre) ;
début
   si a=nil alors
        a←q
   sinon
        si val> a^.info alors
                si a^.FD=nil alors
                        a^.FD←q ;
                sinon
                        inserrer(a^.FD ; q) ;
                finSi
        sinon
                si a^.Fg=nil alors
                        a^.FG←q ;
                sinon
                        inserrer(a^.FG ; q) ;
                finSi ;
        finSi ;
   finSi
fin ;
```

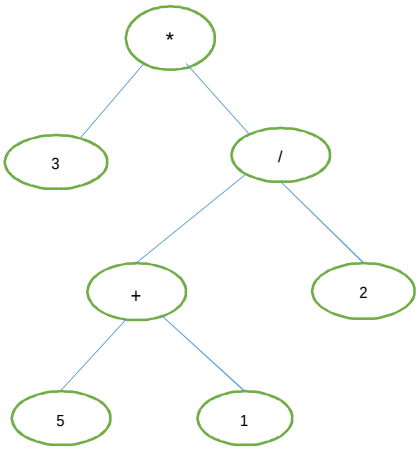
c- procédure ‘‘supprimer **:** pour supprimer un nœud **n** d’un ARB, il faudra d’abord le rechercher. Une fois trouvé, plusieurs cas de figure peuvent se présenter :

Cas		Exemple	Actions
FG(n)	FD(n)		
nil	nil	Feuilles(9, 25, 48, 60)	Remplacer n par nil (supprimer n).
nil	≠nil	Feuilles(18)	Remplacer n par n^.FD (chaîner le père de n avec son fils droit)
≠nil	nil	Feuille(26)	Remplacer n par n^.FG(chaîner le père de n avec son fils gauche)
≠nil	≠nil	25	**1.** Rechercher le plus petit descendant du sous arbre droit (descendre une fois à droite de n, puis descendre le plus à gauche possible : jusqu’à trouver un nœud dont le fg est à NIL) ou le plus grand descendant du sous arbre gauche(descendre une fois à gauche de n, puis descendre le plus à droite possible : jusqu’à trouver un nœud dont le fd est à NIL), soit p ; **2.** Supprimer n et remplacer le par p.

6. Applications des arbres binaires

Les arbres binaires sont des SDD très utilisées en programmation dans plusieurs applications. Nous donnons dans ce qui suit quelques exemples d'utilisation.

- Représentation des expressions arithmétiques **:** Une expression arithmétique peut être représentée sous forme d'arbre binaire. Les nœuds internes contiennent des opérateurs et les feuilles contiennent des valeurs (opérandes). Par exemple, l'expression 3*((5+1)/2) sera représentée par l'arbre suivant :

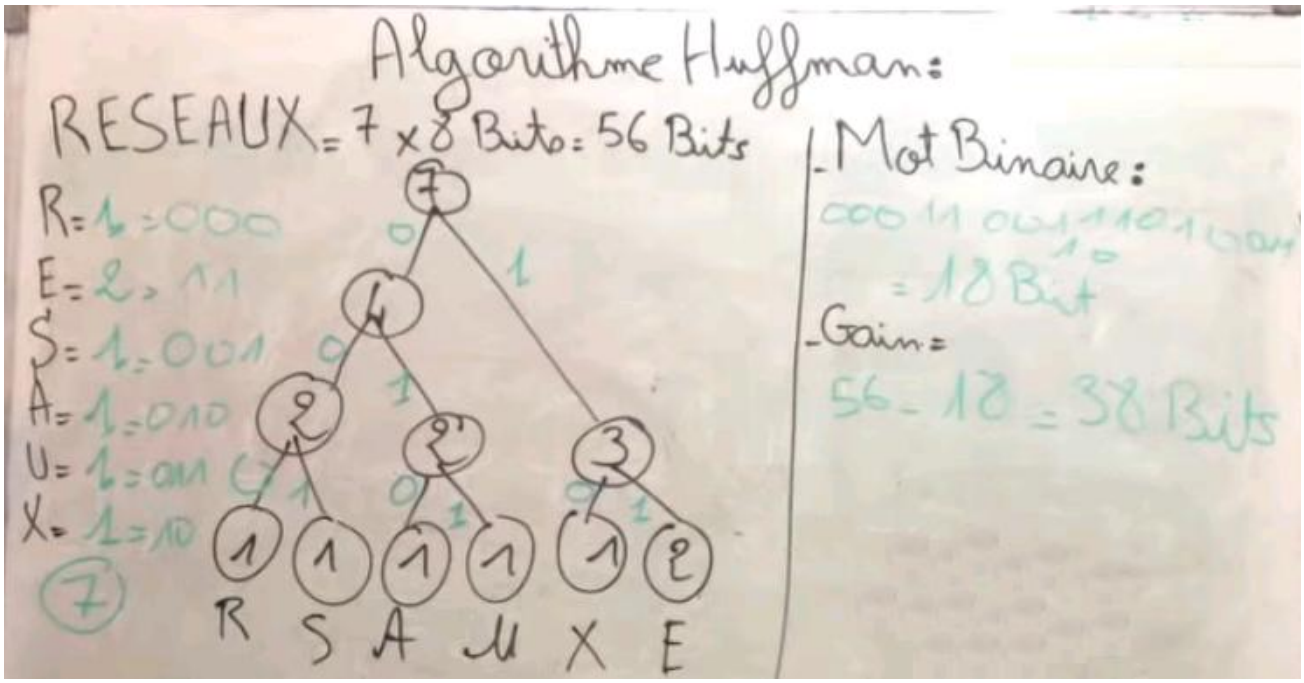


Codage de symboles (code de Huffman) **:** représenter par des codes de longueurs variables les symboles utilisés dans un message. Les symboles ayant une haute fréquence d'apparition se voient attribuer des codes de longueurs réduites, par contre les symboles avec une faible fréquence d'apparition sont codés avec un nombre de bits plus important. L'objectif de cet algorithme est de compresser des fichiers (messages).

1. On calcule les probabilités d'apparition de chaque symbole (caractère).
2. A chaque caractère du message, on construit un nœud (isolé) contenant ce caractère.
3. Choisir les 2 nœuds ayant la plus faible probabilité et non encore choisit
4. Créer un nouveau nœud (un caractère fictif) en connectant les 2 nœuds de l'étape précédente comme fils gauche et fils droit du nouveau nœud. Associé à ce nouveau nœud la somme des probabilités de ces 2 fils.
5. Répéter les deux dernières étapes jusqu'à ce qu'il n'y ait qu'un seul nœud non encore traité. C'est alors la racine de l'arbre de Huffman.

Par convention, on met des 0 sur les arrêtes père-fils gauche et des 1 sur les arrêtes père-fils droite.

Exemple : soit à coder le mot réseau avec le code de Huffman.



Conclusion : Les arbres binaires sont utiles quand une décision sur deux doit être prise à chaque étape d'un traitement. Ils permettant de résoudre certains problèmes de manière concise et élégante grâce aux techniques de la récursivité qui se justifie pleinement sur ces structures**.**

Exercice : Ecrire un sous-programme permettant de créer une liste l à partir d'un ABR a de telle sorte que les nœuds de l soient triés par ordre décroissant.