Chapitre5 : les arbres

1. Définitions :

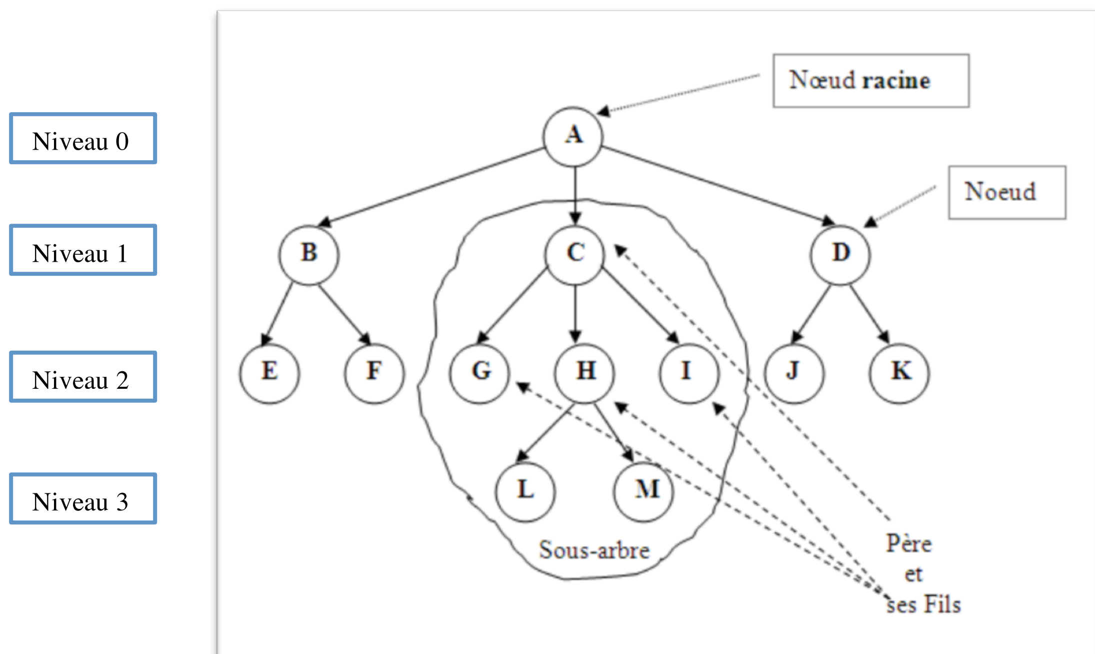
1.1. Définition d'un arbre :

Un arbre est une structure de données ***non linéaire*** sert à mémoriser des données d'une manière hiérarchiques. C'est un graphe sans cycle où chaque nœud a au plus un prédécesseur.

De manière plus simple, Il est constitué d'éléments que l'on appelle souvent des nœuds (*node*) ou sommets tel que :

- Le 1er nœud s'appel : **Racine** « La racine est un nœud qui n'a pas de père».
- Un nœud peut avoir : 0, 1 ou plusieurs sous nœuds (**Fils**).
- Un fils n'a qu'un seul **père**.
- **Les feuilles** sont des nœuds qui sont au bout des branches et qui n'ont pas de fils.
- **Les Frères :** des nœuds qui ont le même père

Exemples :



- Le **prédécesseur** s'il existe s'appelle ***père*** (père de C = A, père de L = H)
- Le **successeur** s'il existe s'appelle ***fils*** (fils de A = {B, C, D}, fils de H= {L, M})
- Le nœud qui n'a pas de prédécesseur s'appelle **racine** (A)
- Le nœud qui n'a pas de successeur s'appelle **feuille** (E, F, G, L, M, J, K).
- Descendants de C= {G, H, I, L, M}. Descendants de B={E, F} ,...
- Ascendants de L= {H, C, A}. Ascendants de E= {B, A} ,...

1.2. Taille d'un arbre :

- ➢ C'est le nombre de nœuds qu'il possède.
    - o Taille de l'arbre précédent = 13
- ➢ Un arbre vide est de taille égale à 0.

1.3. Niveau d'un nœud:

- Le niveau de la racine = 0
- Le niveau de chaque nœud est égale au niveau de son père plus 1
  - Niveau de E, F, G, H, I, J, K = 2

1.4. Hauteur d'un arbre (profondeur) :

- C'est le niveau maximum dans cet arbre.
  - Profondeur de l'arbre précédent = 3

1.5. Degré d'un nœud :

- Le degré d'un nœud est égal au nombre de ses fils.
  - Degré (A) = 3, Degré (B) =2, Degré (C)= 3, Degré (E)= 0, Degré (H) =2,...)

Remarque : le degré des feuilles est égal à « 0 ».

1.6. Degré d'un arbre :

- C'est le degré maximum de ses nœuds.
  - Degré de l'arbre précédent = 3.

2. Utilisation des arbres :

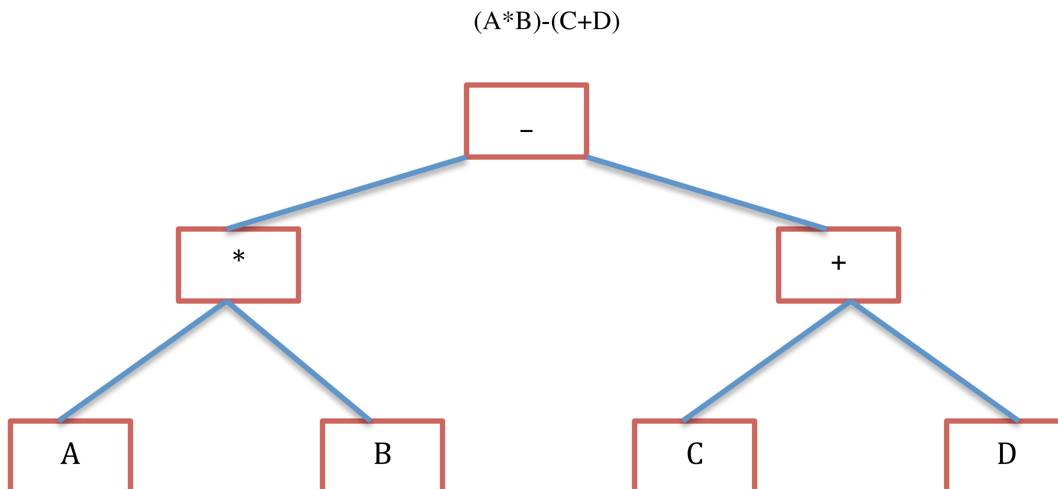
Les arbres constituent un outil important qui est couramment utilisé dans de nombreuses applications :

- Implémentations des SGBD.
- expressions arithmétiques.
- intelligence artificielle.
- représentation des arbres généalogiques.
- Génomique.
- découpage d'un livre en parties, chapitres, sections, paragraphes . . .
- hiérarchies de fichiers,
- etc.

<u>Ex : Représentation des expressions arithmétiques :</u>

- les nœuds intérieurs (degré>0) sont des opérations
- les feuilles (degré=0) sont des nombres ou des variables

(A*B)-(C+D)

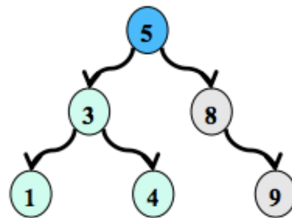


3. Types d'arbres :

Arbre complet: Un arbre est complet si toutes ses feuilles sont sur le même niveau.
Arbre ordonné : Un arbre ordonné est un arbre dans lequel les nœuds respectent un ordre bien déterminé
Arbre binaire : c'est un arbre où **le degré maximum** d'un nœud est égal a 2 ➔ **degré (nœud) ≤ 2**
Arbre binaire ordonné : c'est arbre où :

- Tout élément à gauche de la racine est inférieur à la valeur de la racine.
- Tout élément à droite de la racine est supérieur à la valeur de la racine.
- Cette propriété doit être vérifiée récursivement à tous les niveaux pour que l'arbre binaire soit ordonné.

Ex : arbre binaire complet ordonné :



4. Opérations usuelles sur un arbre :

- construire un arbre
- insérer un élément
- supprimer un élément (arbre de recherche) / suppression d'un sous arbre
- retrouver un élément
- le fils le plus à gauche d'un nœud
- le frère immédiat d'un nœud
- le père d'un nœud
- la racine de l'arbre

5. Implémentation des arbres :

Les arbres peuvent être représentés par des tableaux, des listes non linéaires ou tous les deux :

5.1. Représentation statique « Tableau » :

L'arbre du premier exemple peut être représenté par un tableau comme suit :

indice	info	Fils1	Fils2	Fils3
0	A	1	2	3
1	B	4	5	-1
2	C	6	7	8
3	D	9	10	-1
4	E	-1	-1	-1
5	F	-1	-1	-1
6	G	-1	-1	-1
7	H	11	12	-1
8	I	-1	-1	-1
9	J	-1	-1	-1
10	K	-1	-1	-1
11	L	-1	-1	-1
12	M	-1	-1	-1

Un arbre est représenté par un tableau d'enregistrement. Chaque enregistrement contient la valeur d'un nœud donnée et la position de ses fils dans le tableau.

Le nombre de fils dans un enregistrement est égal au degré de l'arbre. Ex ; degré = 3.

Ex :

```
struct  nœudStatic {
        char info [1] ;
        int fils1 ;
        int fils2 ;
        int fils3 ;

} ;
nœudStatic  Arbre [taille d'un arbre] ;
```

Si la valeur de l'un des fils = -1 ➔ ce nœud n'as pas de fils.

5.2. Représentation dynamique « pointeur» :

5.2.1. Arbre binaire :

<u>Nœud :</u> Un nœud est une structure (ou un enregistrement) qui contient au minimum trois champs (pour les arbres binaire) :

- un champ contenant l'élément du nœud, c'est **<u>l'information</u>** qui est importante. Cette information peut être un entier, une chaîne de caractère ou tout autre chose que l'on désire stocker.

- Les deux autres champs sont : le **<u>fils gauche</u>** et le **<u>fils droit</u>** du nœud.

  Ces fils sont en fait des arbres, on les appelle généralement les ***sous arbres gauches*** et les ***sous arbres droit*** du nœud.

De part cette définition, un arbre ne pourra donc être qu'un pointeur sur un nœud.

```
struct nœud {
        TypeElement     info;
        nœud*           FilsGauche;
        nœud*           FilsDroit;
};
typedef struct nœud* Arbre;
```

```
Déclaration :
Soit :
        Arbre    Racine1 = NULL;
Ou:
        nœud*    Racine1 = NULL;
Ou:
        struct nœud*     Racine1 = NULL;
```

5.2.2.Arbre N-aires :

Un arbre N-aire est un arbre dans lequel le nombre de fils d'un nœud n'est pas limité.

A. Représentation avec des pointeurs simple:

Pour les arbres quelconques, un nœud contient :

- un champ contenant l'information du nœud.
- Un tableau des pointeurs qui contient l'adresse de ses fils

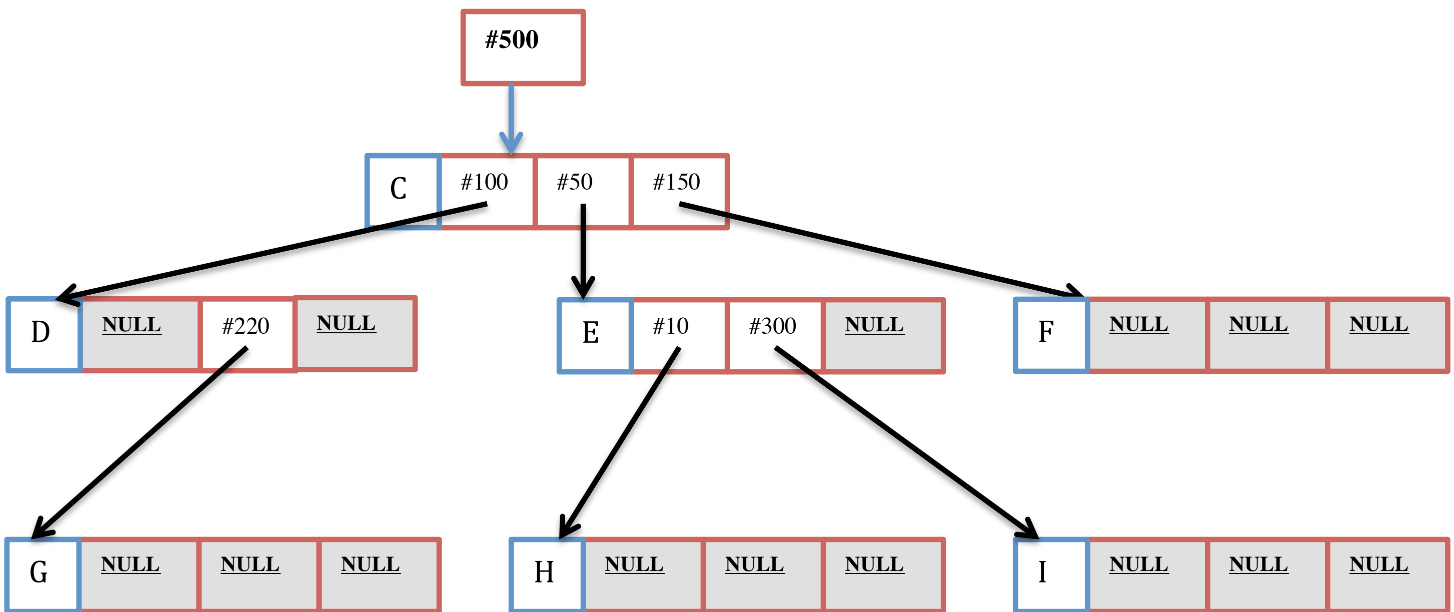
```
struct nœud {
        TypeElement        info;
        nœud*              Fils [degré_d_un_arbre ] ;
};
typedef struct nœud* Arbre;
```

Ex : Racine1 est un arbre « degré =3 » :

.... déclaration : ....

```
struct nœud {
    TypeElement        info;
    nœud*              Fils [3] ;
};
nœud* Racine ;
```

....



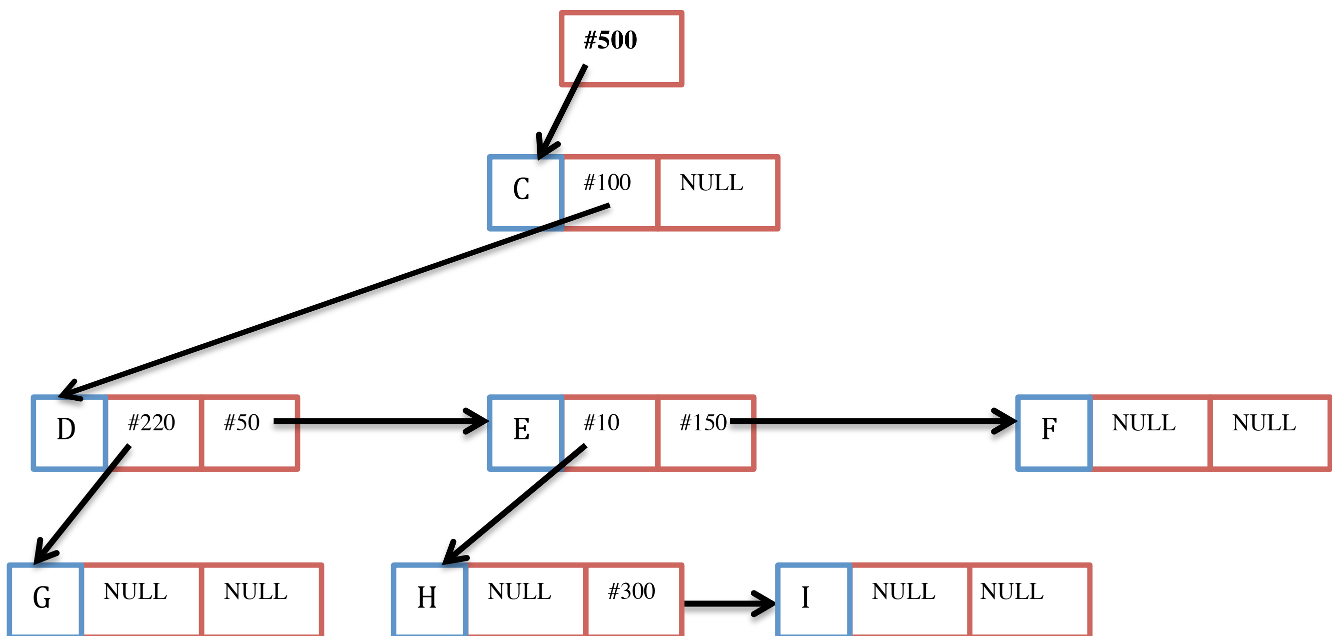
B. Représentation par des listes (la plus optimale) :

On les représente avec des pointeurs, en liant les fils d'un même nœud entre eux, comme dans une liste chainée.
A partir d'un nœud quelconque, on accède à son fils ainé (la liste de ses fils) et à son frère.

Dans ce cas on éliminera autant que possible les pointeurs NULL.

```
struct nœud {
        TypeElement     info;
        nœud*           FilsAiné
        nœud*           Frère ;
};
typedef struct nœud* Arbre;
```

Ex: l'arbre précédent aura la présentation suivante :



5.3. Représentation hybride (tableau + pointeurs) :

Il est tout a fait possible de représenter un arbre comme étant un tableau des listes où chaque case contient un enregistrement (info + la tête d'une liste). La liste contient les indices des fils de chaque nœud.

6. Operations sur les arbres binaire ordinaire:

Différents traitements peuvent être effectués sur un arbre, mais la problématique se situe au niveau du parcours de l'arbre. Afin de faciliter notre manipulation des arbres, nous allons créer quelques fonctions tels que :

- Tester si un arbre est vide.
- Récupérer le fils gauche ainsi que le fils droit d'un **arbre binaire**.
- Savoir si nous sommes sur une feuille
- Etc...

6.1. Tester si l'arbre est vide :

```
int EstVide( nœud* A) {
        return A == NULL;
}
```

Ou

```
int EstVide( Arbre A) {
        return A == NULL;
}
```

Tout simplement parce qu'on a donner le nom « Arbre » à « nœud* » : **typedef struct** nœud* Arbre;

6.2. Récupérer le fils gauche et droit d'un arbre binaire :

Il faut faire attention à un problème : le cas où l'arbre est vide. En effet, dans ce cas, il n'existe pas de sous arbre gauche ni de sous arbre droit. Pour régler ce problème nous décidons arbitrairement de renvoyer l'arbre vide comme fils d'un arbre vide.

```
Nœud* Gauche( nœud*  A)  {

        if ( EstVide (A) ){
                return NULL;
        }else{
                return (*A).FilsGauche;
        }
}
```

```
Nœud* Droit( nœud*  A)  {

        if ( EstVide (A) ){
                return NULL;
        }else{
                return (*A).FilsDroit;
        }
}
```

6.3. Tester si le nœud est une feuille:

Une feuille est un nœud non NULL dont tout ses fils son NULL.

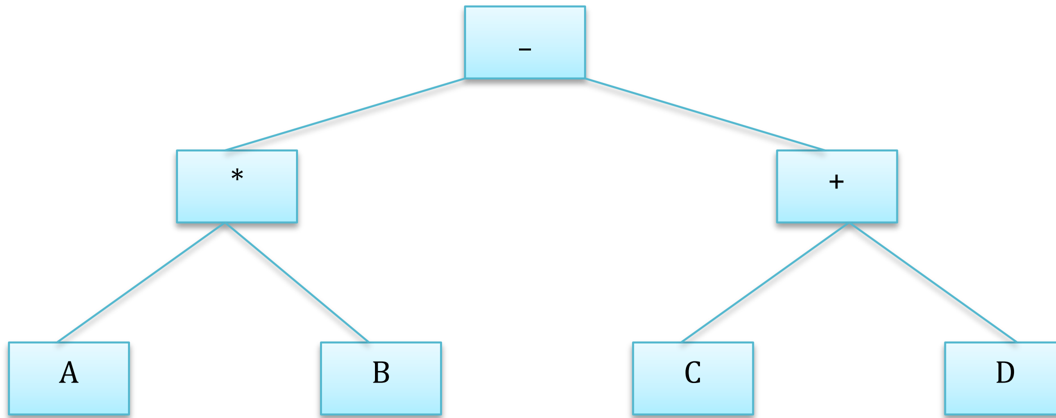
```
bool EstUneFeuille(nœud*  N) {
if (EstVide(N)) {
        return false;
}else if EstVide (Gauche (A)) && EstVide (Droit (A)) {
        return true;
}else{
        return false;
}
}
```

6.4. Parcours d'un arbre binaire:

Le parcours d'un arbre consiste à passer par tous ses nœuds. Les parcours permettent d'effectuer tout un ensemble de traitement sur les arbres. On distingue deux types de parcours :

6.4.1.Parcours en profondeur :

Dans un parcours en profondeur, on descend le plus profondément possible dans l'arbre puis, une fois qu'une feuille a été atteinte, on remonte pour explorer les autres branches en commençant par la branche "la plus basse" parmi celles non encore parcourues.



Dans le parcours en profondeur, il existe principalement 3 types de parcours :

A. Parcours préfixé :

Il consiste à effectuer ces étapes pour chaque nœud :

- visiter la partie information (affichage ou traitement).
- Parcourir le sous arbre gauche.
- Parcourir le sou arbre droit.

Explication :

Etape1 (nœud « - ») : on commence par visiter la racine et afficher sa valeur ➔ printf (‘' - '')
Etape2 (nœud « - ») : passer au sous arbre gauche (ne pas effectuer l'etape3, et passer directement au nœud « * ».
Etape1 (nœud « * ») : afficher le contenue de ce nœud ➔ printf (‘' * '')
Etape2 (nœud « * ») : passer au sous arbre gauche (ne pas effectuer l'etape3, et passer directement au nœud « A ».
Etape1 (nœud « A ») : afficher le contenue de ce nœud ➔ printf (‘' A '')
Etape2 (nœud « A ») : le nœud « A » n'a pas de fils gauche.
Etape3 (nœud « A ») : le nœud « A » n'a pas de fils droit ➔ on remonte au père de « A» (parce que le nœud « * » qui a appelé le nœud « A »).

Etape3 (nœud « * ») : est ce qu'il a un sous arbre droit ? Oui ➔ passer au sous arbre droit de « * ».

Etape1 (nœud « B») : afficher le contenue de ce nœud ➔ printf (‘' B '')
Etape2 (nœud « B ») : le nœud « B » n'a pas de fils gauche.
Etape3 (nœud « B ») : le nœud « B » n'a pas de fils droit ➔ on remonte au nœud « * » (parce que le nœud « * » qui a appelé le nœud « B »).

Attention !!! : On a fini toutes les étapes avec le nœud « * » ➔ on remonte au père de « * » qui est le nœud « - »

Etape3 (nœud « - ») : est ce qu'il a un sous arbre droit ? Oui ➔ passer au sous arbre droit de « + ».
Etc…

A la fin de ce parcours, l'ordre d'affichage (traitement) est le suivant : **- * A B + C D**

B. Parcours Infixé :

Il consiste à effectuer ces étapes pour chaque nœud :

- Parcourir le sous arbre gauche.
- visiter la partie information (affichage ou traitement).
- Parcourir le sou arbre droit.

Explication :

Etape1 (nœud « - ») : on commence par la racine. Est ce que le nœud **« - »** a un fils gauche ? oui ➔ passer au sous arbre gauche (ne pas effectuer l'etape2 et 3, et passer directement au nœud « * ».

Etape1 (nœud « * ») : Est ce que le nœud **« * »** possède un fils gauche ? oui ➔ passer au sous arbre gauche (ne pas effectuer l'etape2 et 3, et passer directement au nœud « A ».

Etape1 (nœud « A ») : Est ce que le nœud « **A »** possède un fils gauche ? non ➔ on passe à l'étape suivante pour ce nœud.
Etape2 (nœud « A ») : afficher le contenu de ce nœud ➔ printf (‘' A '')
Etape3 (nœud « A ») : le nœud « A » n'a pas de fils droit ➔ on remonte au père de « A» (parce que le nœud « * » qui a appelé le nœud « A »).

Etape2 (nœud « * ») : on a fini la 1ere étape, la 2eme étape c'est un affichage ➔ printf (‘' * '')
Etape3 (nœud «* ») : est ce qu'il possède un fils droit ? Oui ➔ passer aux sous arbre droit de « B».

Etape1 (nœud « B ») : Est ce que le nœud **« B »** possède un fils gauche ? non
Etape2 (nœud « B ») : on a fini la 1ere étape, la 2eme étape c'est un affichage ➔ printf (‘' B '')
Etape3 (nœud «B ») : le nœud « B » n'a pas de fils droit ➔ on remonte vers le père de « B»

On a fini toutes les étapes avec le nœud « * » ➔ on remonte au père de « * » qui est le nœud « - »

Etape2 (nœud « - ») : afficher le contenu de ce nœud ➔ printf (‘' - '')
Etape3 (nœud « - ») : est ce qu'il possède un fils droit ? Oui ➔ passer aux sous arbre droit de « - ».

Etc....

A la fin de ce parcours, l'ordre d'affichage (traitement) est le suivant : A * B – C + D

C. Parcours Postfixé :

Il consiste à effectuer ces étapes pour chaque nœud :

- Parcourir le sous arbre gauche.
- Parcourir le sou arbre droit.
- visiter la partie information (affichage ou traitement).

A la fin de ce parcours, l'ordre d'affichage (traitement) est le suivant : AB* CD+ -

6.4.2. Parcours en largeur:

Le parcours en largeur permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau 1 puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

A la fin de ce parcours, l'ordre d'affichage (traitement) est le suivant : - *+ ABCD

6.5. Création d'un arbre binaire (méthode Ascendante) :

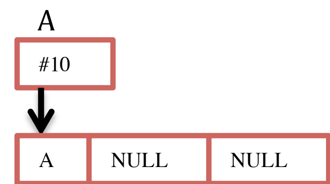
Pour créer un arbre, il faut tout d'abord créer un nœud, ensuite, on place dans les fils gauche et droit les sous arbres que l'on a passés en paramètre ainsi que la valeur associée au nœud.

```
Nœud*  Créer (TElement val,  Nœud*  SousArbreG, Nœud*  SousArbreD) {
            Nœud*  nouveau;
            nouveau =  (Nœud*) malloc(sizeof(Nœud));

            nouveau ->info = val;
            nouveau ->FilsGauche = SousArbreG;
            nouveau ->FilsDroit = SousArbreD;

return nouveau; }
```

Ex : création de l'arbre : (A*B)-(C+D)

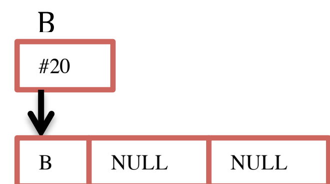
- **Etape1 : créer le nœud A (sous arbre) :**

  nœud* A ;
  A = créer ('A', NULL,NULL) ;



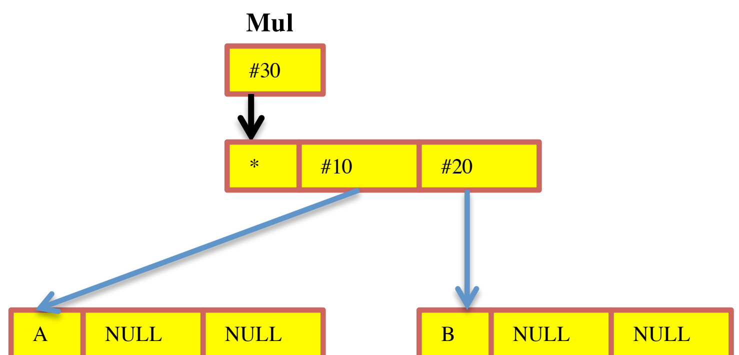
- **Etape2 : créer le nœud B (sous arbre) :**

  nœud* B ;
  A = créer ('B', NULL,NULL) ;



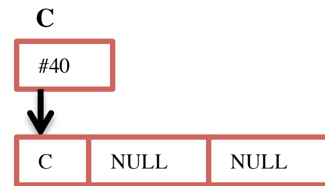
- **Etape3 : créer le nœud Mul (sous arbre) :**

  nœud* Mul;
  Mul = créer ('*', A, B) ;



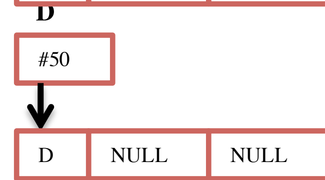
- **Etape4 : créer le nœud C (sous arbre) :**

nœud* C ;
C = créer ('C', NULL,NULL) ;



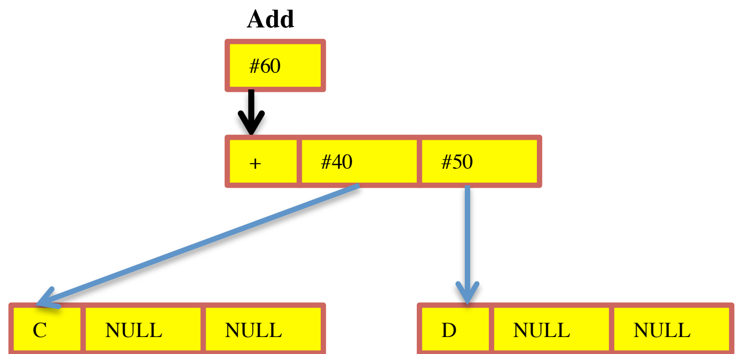
- **Etape5 : créer le nœud D (sous arbre) :**

nœud* D ;
D = créer ('D', NULL,NULL) ;



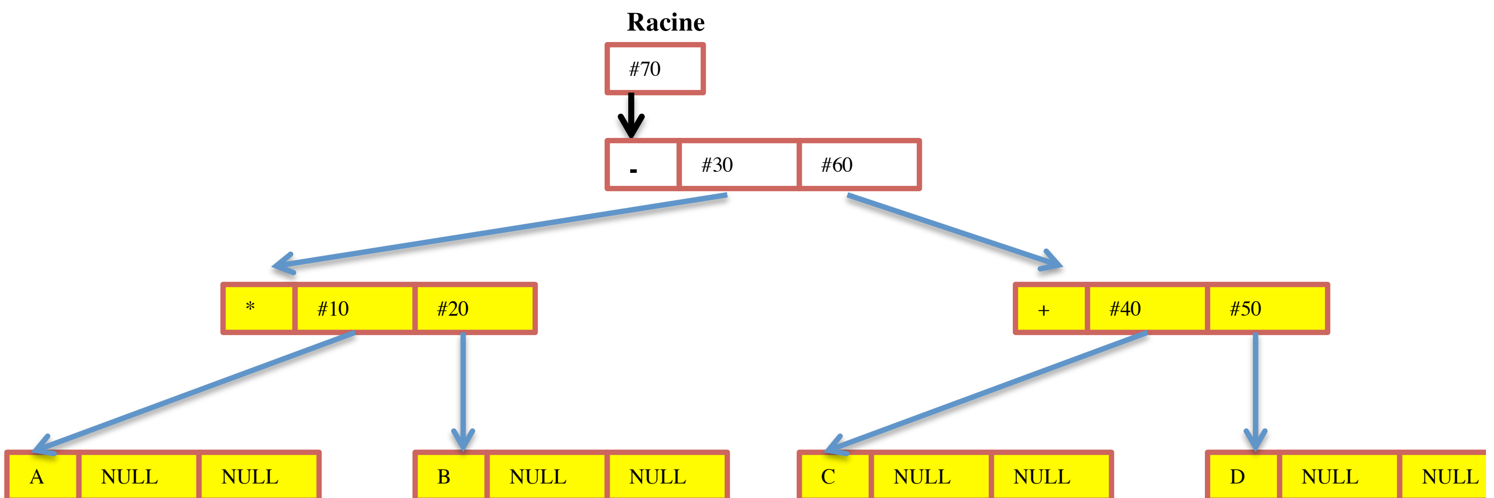
- **Etape6 : créer le nœud Add (sous arbre) :**

nœud* Add;
Add = créer ('+', C, D) ;



- **Etape7: créer le nœud Racine (l'arbre final) :**

nœud* Racine;
Racine = créer ('-', Mul, Add) ;



7. Les arbres binaires de recherches « ABR » (lexicographique) :

7.1. Définition :

Un arbre binaire de recherche est un arbre binaire dans lequel :

- Chaque nœud possède une clé, telle que chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une clé supérieure ou égale à celle-ci.

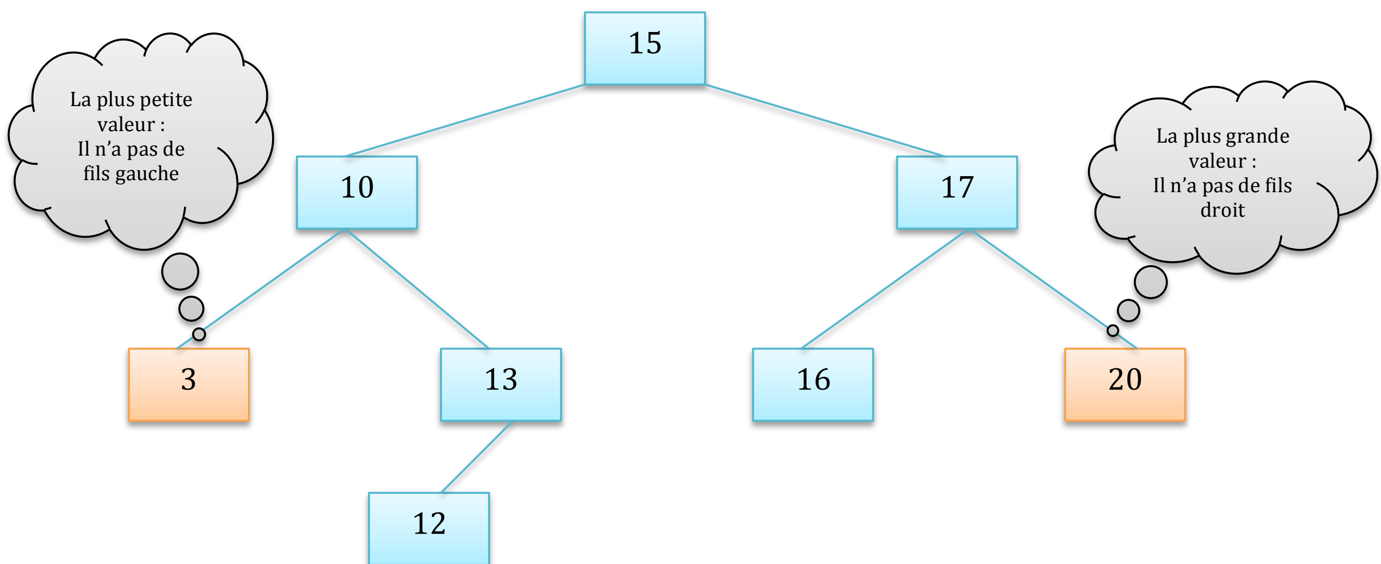
- Dans certains cas, le champ d'info jouera le rôle d'une clé (s'il le champ info est muni d'une relation d'ordre).

- Selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale.

- Les nœuds que l'on ajoute deviennent des feuilles de l'arbre.

NB : Le parcours infixé de l'arbre produit la suite *ordonnée* des clés

EX : le parcours **infixé** de l'arbre ci-dessous donne : 3, 10, 12, 13, 15, 16, 17, 20 (une suite ordonnée des clés)



7.2. Opérations:

En plus des opérations que nous avons déjà vu dans les arbres binaires ordinaires, l'ABR permet d'effectuer d'autres opérations d'une manière souple et flexible.

7.2.1. Accéder à la plus petite clé (la plus petite information) :

Pour accéder à la clé la plus petite dans un arbre binaire de recherche il suffit de descendre sur le fils gauche autant que possible jusqu'à trouver la valeur NULL.
Le dernier nœud visité, qui **n'a pas de fils gauche**, porte la valeur la plus petite de l'arbre.

7.2.2. Accéder à la plus grande clé (la plus grande information) :

Pour accéder à la clé la plus grande dans un arbre binaire de recherche il suffit de descendre sur le fils droit autant que possible jusqu'à trouver la valeur NULL.
Le dernier nœud visité, qui **n'a pas de fils droit**, porte la valeur la plus grande de l'arbre.

7.2.3.La recherche d'une valeur :

La recherche d'une valeur dans un ABR consiste à parcourir une branche en partant de la racine**, en descendant chaque fois** sur ***le fils gauche*** **ou** sur ***le fils droit*** selon la valeur (ou la clé) portée par le nœud est plus grande ou plus petite que la valeur cherchée.

- If x <= (*R).info ➔ R= (*R).FG ;
- If x > (*R).info ➔ R= (*R).FD ;

La recherche s'arrête dès que la valeur est rencontrée ➔ x= (*R).info

Ou

On a atteint l'extrémité d'une branche (le fils sur lequel il aurait fallu descendre n'existe pas) ➔ x n'existe pas dans l'arbre.

```
nœud *  recherche ( int x, nœud* R) {

        while (  (R != NULL)  &&  (x !=  (*R).info )  ) {

                // si la valeur de X est inferieur à l'information de ce nœud on visite le fils gauche
                if ( x  <  (*R).info  ){
                        R = (*R).FG ;

                // si la valeur de X est supérieur à l'information de ce nœud on visite le fils droit
                }else{
                        R= (*R).FD ;
                }
        }// Descendre jusqu'à trouver l'élément  ou  arrivé a une feuille (l'élément n'existe pas)

        // a la fin on retourne R , R soit = NULL , soit l'adresse du nœud qui contient X.
        return R ;
}
```

7.2.4. L'insertion d'une valeur :

Si on veut ajouter un élément dans un arbre binaire de recherche, ceci **doit être inséré comme une feuille**.

Le principe est le même que pour la recherche. Un nouveau nœud est créé avec la nouvelle valeur et **inséré à l'endroit où la recherche s'est arrêtée.**

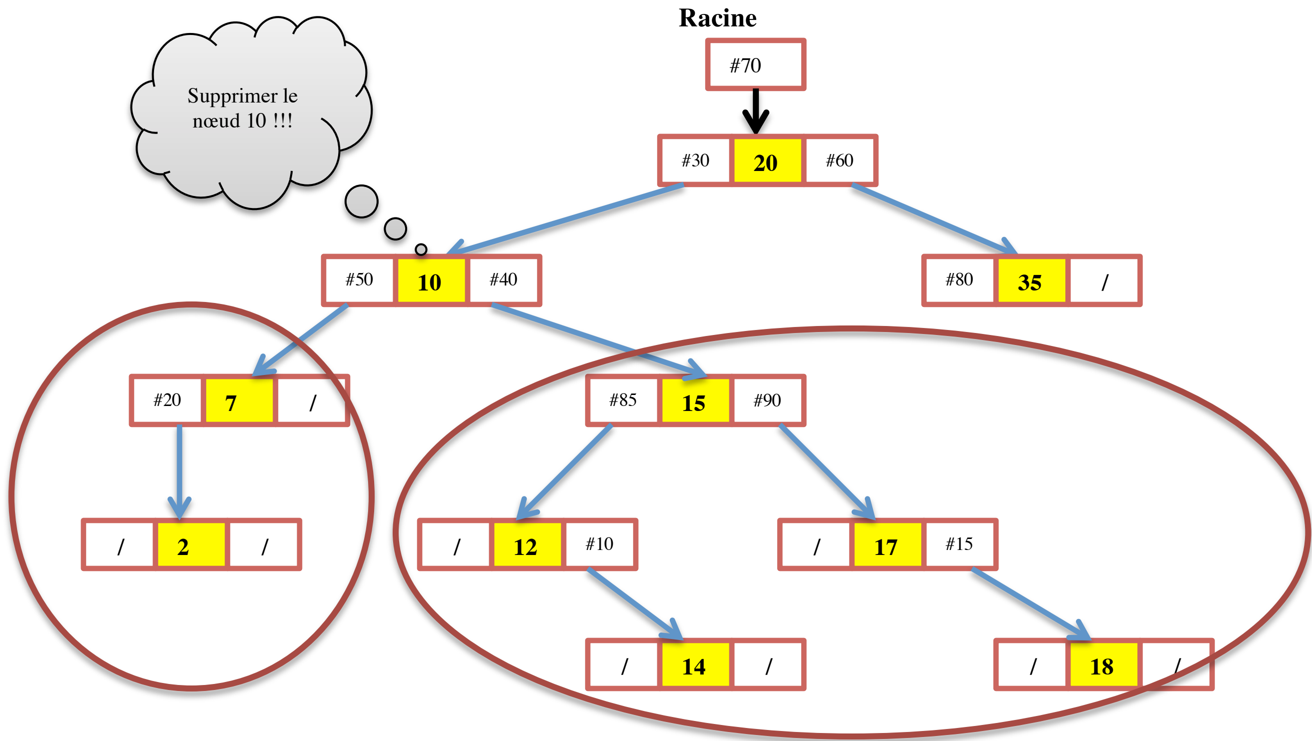
Dans l'exemple précèdent, si on veut insérer la valeur 14 il faut descendre jusqu'à trouver le nœud 13.
On arrive à ce nœud, il faut comparer 14 avec 13. Apres la comparaison il faut vérifier :

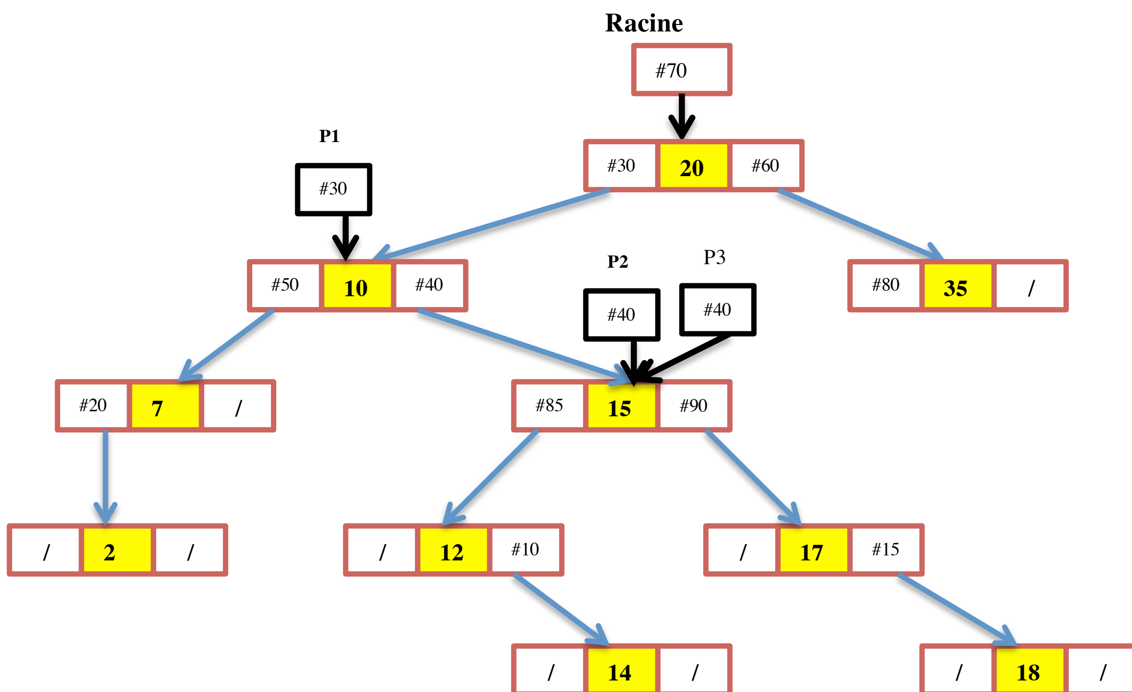
- si le fils droit = NULL ➔ on alloue un espace mémoire pour notre élément (14) ➔ puis on l'insère comme fils droit de 13. (comme dans notre exemple)
- Sinon on continuer à descendre.

7.2.5. La suppression d'un nœud :

Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé à partir de sa clé :

- **Suppression d'une feuille** : vu qu'elle n'a pas de fils, on le supprime tout simplement
- **Suppression d'un nœud avec un seul fils** : Il faut l'enlever de l'arbre en le remplaçant par son fils.
- **Suppression d'un nœud avec 2 fils :** dans ce cas on a le choix entre 2 cas :
  - Le remplacer par le nœud **le plus grand** (qui possède la plus grande valeur de clé) du **sous arbre gauche**.
  - Le remplacer par le nœud **le plus petit** (qui possède la plus petite valeur de clé) du **sous arbre droit.**





A. Cas1 : supprimer le nœud 10 et le remplacer par le nœud le plus petit du sous arbre droit → 12

- **1ere étape : créer un pointeur P2 et P3 qui se pointent vers le sous arbre droit :**

```
P2 = (*P1).FD ;
P3 = (*P1).FD ;
```

- **2eme étape : pointer P2 vers le père de 12 :**

```
while ( (* (*P2).FG ).FG != NULL ){

    P2= (*P2).FG ;
}
```

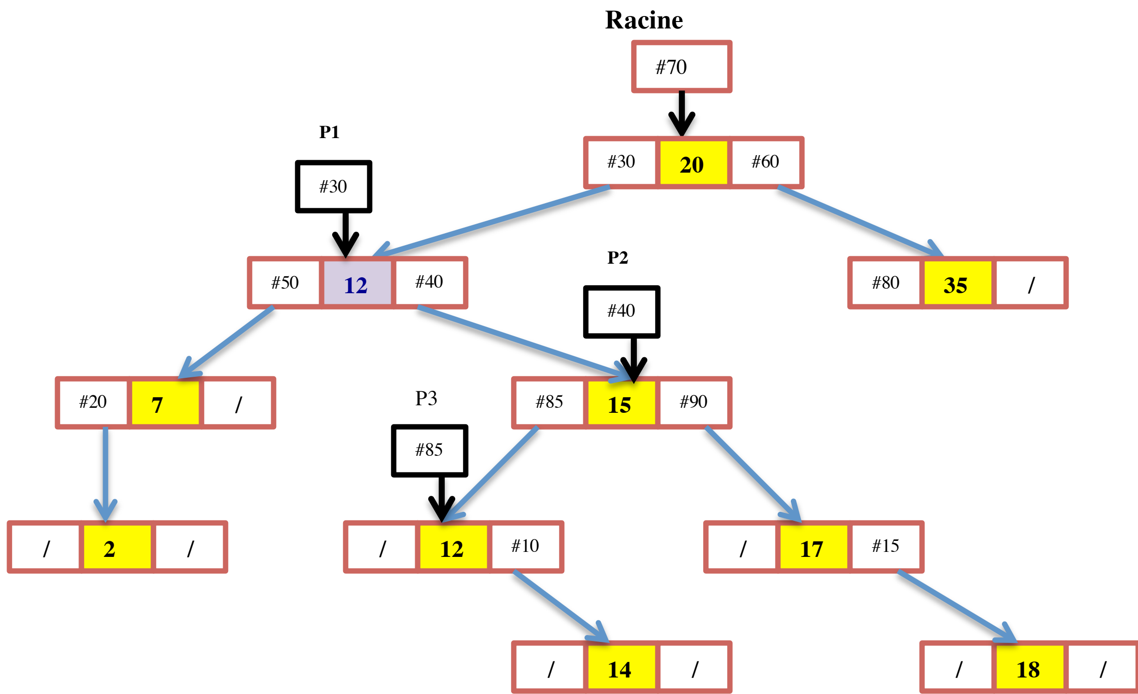
A la fin de cette boucle, P2 pointe vers le père du nouveau nœud (12)

- **3eme étape : pointer P3 vers le nœud le plus petit du sous arbre droit→ parcourir le sous arbre droit jusqu'à trouver un nœud qui n'a pas de fils gauche.**

```
while ( (*P3).FG != NULL ){
    P3= (*P3).FG ;
}
```

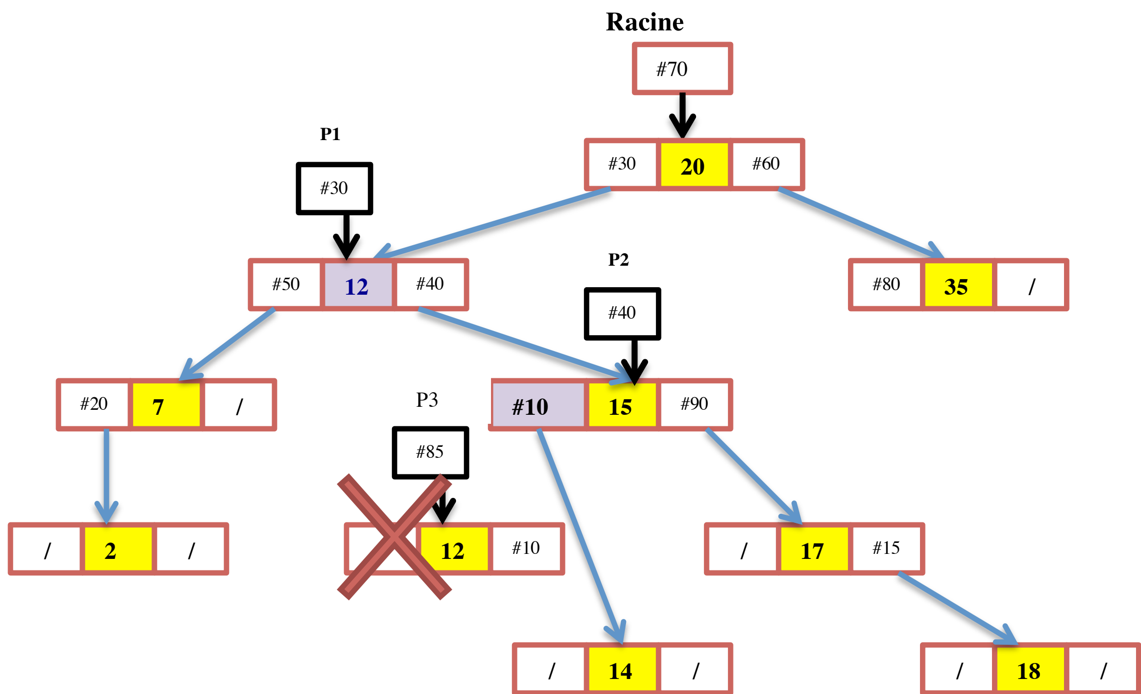
- **4eme étape : copier la valeur du nœud cible dans le nœud qu'on va supprimer**

```
(*P1).info = (*P3).info ;
```



- **5eme étape : supprimer le nœud le plus petit du sous arbre gauche ➔ ceci implique la modification du FG de son père pour le faire pointer vers le fils droit.**

```
(*P2).FG = (*P3).FD ;
free (P3) ;
```



NB : on faite on n'a pas supprimer l'espace mémoire du nœud 10. Mais, ceci du nœud qui va le remplacer.