Université Sultan Moulay Slimane
Ecole Supérieure de Technologie
- Beni Mellal -

STRUCTURES DE DONNEES

Filières : DUT **GI** / DUT **IDIA**
Semestre : S3
Année Universitaire : 2025/2026

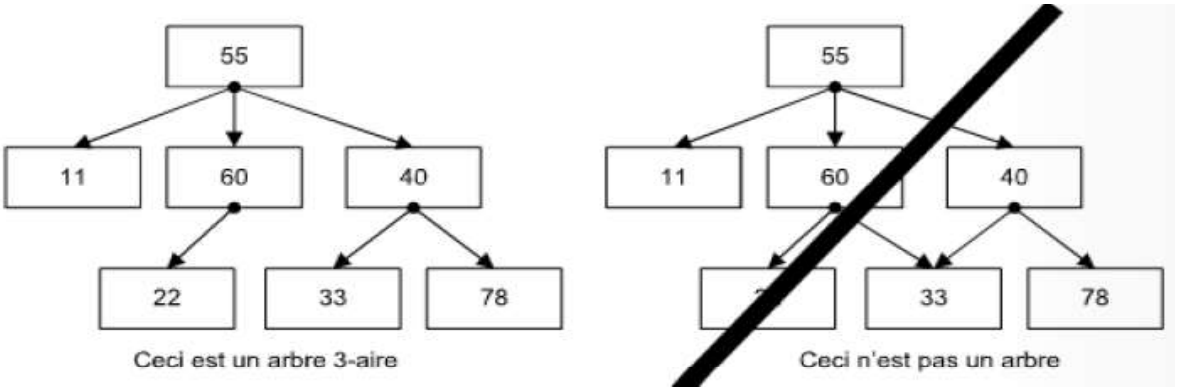
Pr. M. OUTANOUTE
m.outanoute@usms.ma

- Chapitre 7 -

LES ARBRES

1) DÉFINITION

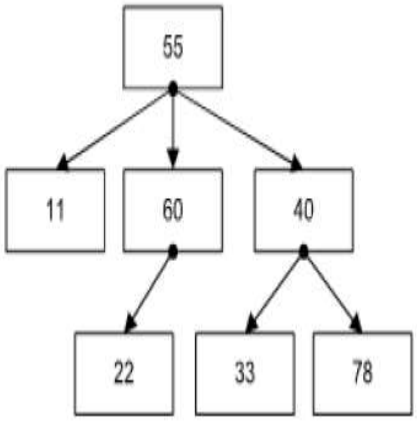
- Un arbre est une structure composée d'éléments appelés nœuds.
- Un arbre, appelé aussi arbre N-aire, chaque nœud possède au maximum N nœuds.
- La représentation d'un arbre en informatique se fait à l'envers : la racine se trouve en haut et les branches se développent vers le bas.



1) DÉFINITION

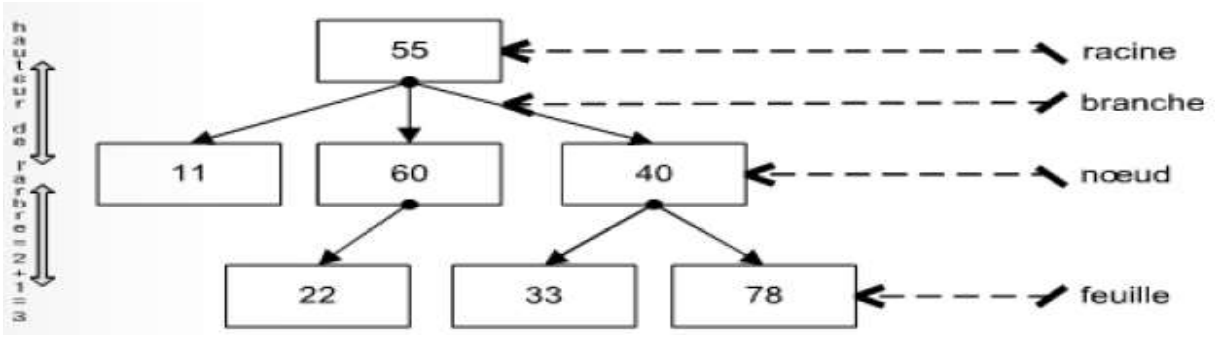
- Un arbre est constitué d'un noeud particulier appelé racine et d'une suite ordonnée éventuellement vide A1, A2, …, Ap d'arbres disjoints appelés sous-arbres de la racine.
- Un arbre contient donc au moins un nœud : sa racine. Tous les autres nœuds suivent directement ou indirectement la racine.
- La figure suivante représente l'arbre d'une descendance :

  - **La racine de cet arbre contient 55**
  - **Il est constitué de trois sous-arbres :**
    - **A1 : de racine 11, est réduit à 11**
    - **A2 : de racine 60, contient 60 et 22**
    - **A3 : de racine 40, contient 40, 33 et 78**



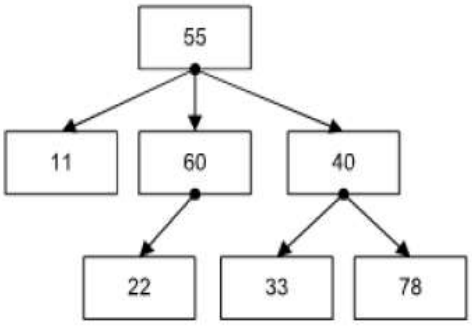
1) DÉFINITION

- Un nœud est aussi appelé sommet, contient un élément et indique les nœuds suivants.
- Les fils d'un nœud sont les racines de ses sous-arbres.
- Une feuille d'un arbre est un nœud sans fils (qui n'a pas de suivant) :
  11, 22,33 et 78 sont des feuilles.
- Une branche est un chemin qui rejoint deux nœuds
- La hauteur d'un nœud est égale au nombre de branches le séparant de la feuille la plus éloignée plus un.
- La profondeur d'un nœud est égale au nombres de branches le séparant de la racine.
- La hauteur d'un arbre vaut la hauteur de la racine.



2) CONSTRUCTION D'UN ARBRE PAR UN TABLEAU

- Chaque nœud de notre exemple possède au plus trois nœuds fils.
- On construit un tableau à deux dimensions où chaque indice représente un nœud.
- Pour connaître les suivants de chaque nœud, il suffit de voir leur indice dans les cases ad-hoc du tableau.
- L'absence d'un nœud suivant est représenté par la valeur -1.



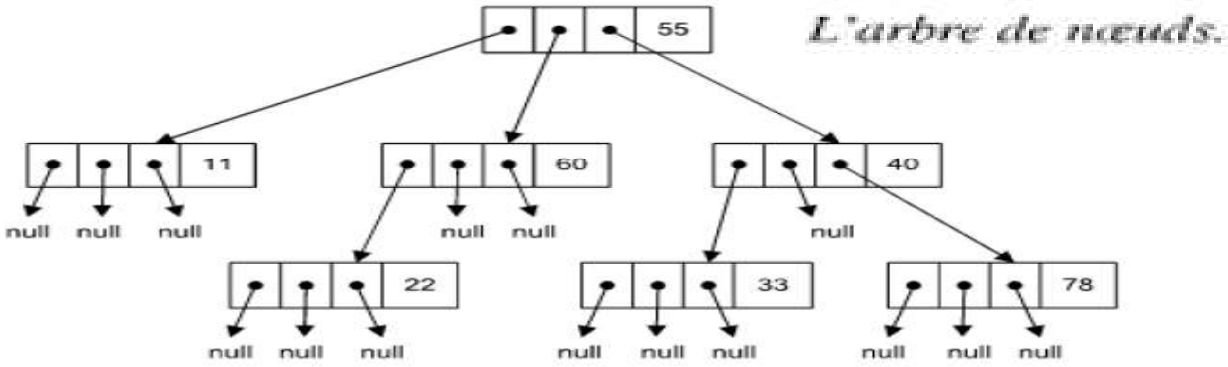
Numéro	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Valeur	55	11	60	40	22	33	78
Suivant 1	1	-1	4	5	-1	-1	-1
Suivant 2	2	-1	-1	-1	-1	-1	-1
Suivant 3	3	-1	-1	6	-1	-1	-1

- Cette construction est compliquée à gérer, et de ce fait, elle n'est pas très utilisée.

3) CONSTRUCTION D'UN ARBRE PAR UNE LISTE CHAINEE

- Un nœud pouvant référencer jusqu'à 3 nœuds suivants : 3 attributs gauche, milieu et droite ou dans un tableau de 3 éléments.
- Nous allons introduire la structure suivante :

```
typedef struct noeud {
        int valeur ;
        struct noeud *gauche ;
        struct noeud *milieu ;
        struct noeud *droite ;
} NoeudEntier ;
```



L'arbre de nœuds.

3) CONSTRUCTION D'UN ARBRE PAR UNE LISTE CHAINEE

- Pour un arbre N-aire de taille supérieure, il aurait plus pratique de stocker les sous arbres dans un tableau.
- Il est possible de décrire l'arbre avec une écriture standard utilisant les parenthèses : chaque sous arbre est représenté par la racine, suivie de ses suivants entre parenthèses.
- Chaque sous arbre est lui-même un arbre qui utilise la même notation :

(55(sous arbre gauche, sous arbre milieu, sous arbre droit))

sous arbre gauche s'ecrit : **(11)**

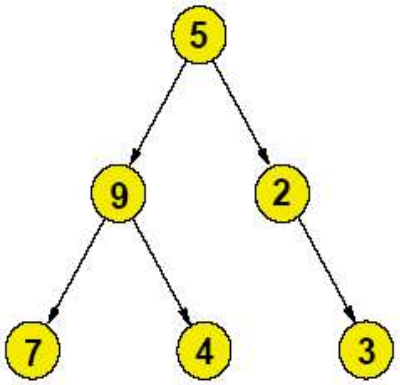
sous arbre milieu s'ecrit : **(60(22))**

sous arbre droit s'écrit : **(40(33,78))**

Ce qui donne : **(55(11,60(22),40(33,78)))**

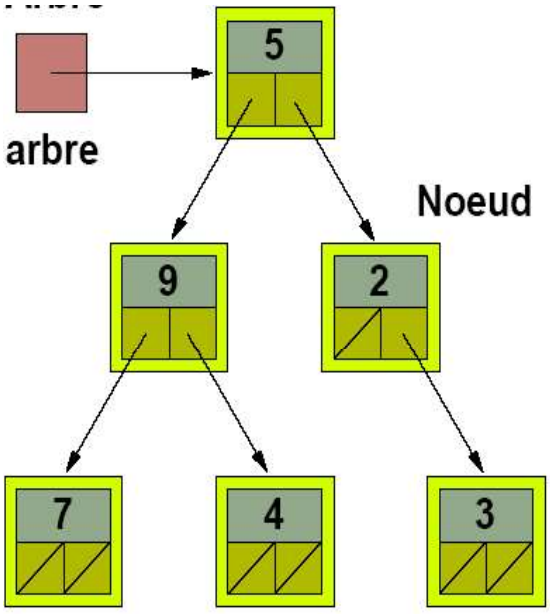
4) ARBRE BINAIRE

- Un arbre binaire est un arbre 2-aire : chaque nœud possède 0, 1 ou 2 suivants.
- Chaque arbre binaire peut posséder un arbre binaire droit et un arbre binaire gauche dont les racines sont respectivement son fils droit et son fils gauche.
- Pour coder un arbre binaire, on fait correspondre à chaque nœud :
  - Une structure contenant la donnée et deux adresses, une adresse pour chacun des deux noeuds fils.
  - Avec la convention qu'une adresse nulle indique un arbre binaire vide.
  - Mémoriser l'adresse de la racine pour pouvoir reconstituer tout l'arbre.



4) ARBRE BINAIRE

```
typedef struct noeud {
    int valeur ;
    struct noeud *sag ;
    struct noeud *sad ;
} Noeud ;
Nœud *arbre ;
```

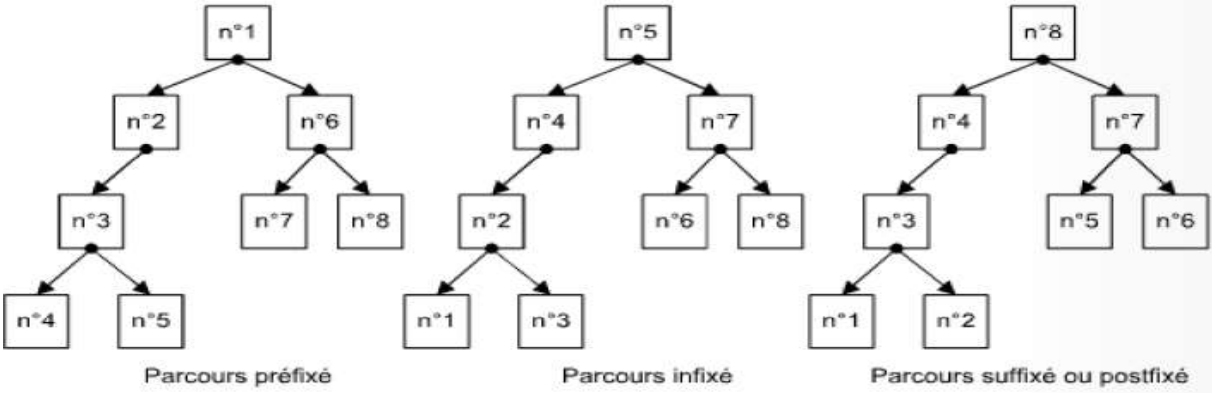


5) LES PARCOURS D'ARBRE

- Il existe 4 techniques pour parcourir l'ensemble des valeurs d'un arbre.
- 4 algorithmes implémentant ces différents parcours (récursifs).

❖ **Parcours en profondeur :**

- Trois algorithmes récursifs simples permettent le parcours en profondeur d'un arbre binaire : préfixé, infixé et postfixé.
- Tous les nœuds de l'arbre sont atteints branche par branche dans toute leur profondeur.



5) LES PARCOURS D'ARBRE

❑ **Parcours préfixé :**

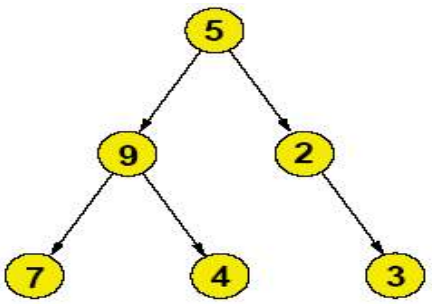
Le traitement se fait avant la visite des sous arbres.

```
void parcoursPrefixe(Noeud *ar) {
    printf("%d  ", ar->valeur) ;  //traitement
    if (ar->sag != NULL) parcoursPrefixe(ar->sag) ; // appel récursif
    if (ar->sad != NULL) parcoursPrefixe(ar->sad) ; // appel récursif
}
```

Exemple :

Le parcours préfixé donne :

5 9 7 4 2 3



5) LES PARCOURS D'ARBRE

❑ Parcours infixé :

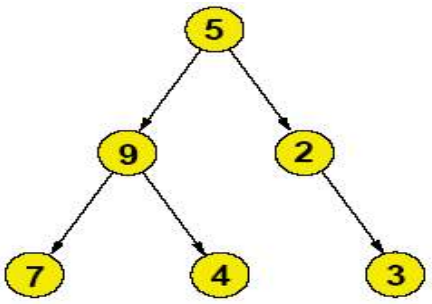
Le traitement se fait entre les deux visites des sous arbres.

```
void parcoursInfixe(Noeud *ar) {
    if (ar->sag != NULL) parcoursInfixe(ar->sag) ; // appel récursif
    printf("%d  ", ar->valeur) ;  //traitement
    if (ar->sad != NULL) parcoursInfixe(ar->sad) ; // appel récursif
}
```

Exemple :

Le parcours infixé donne :

7 9 4 5 2 3



5) LES PARCOURS D'ARBRE

❑ Parcours postfixé (suffixé) :

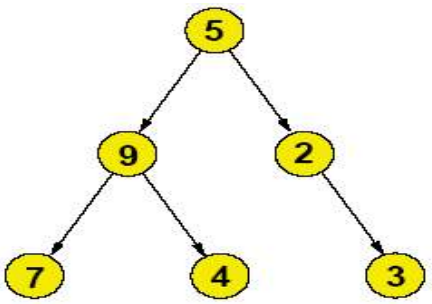
Le traitement se fait après la visite des sous arbres.

```
void parcoursPostfixe(Noeud *ar) {
    if (ar->sag != NULL) parcoursPostfixe(ar->sag) ; // appel récursif
    if (ar->sad != NULL) parcoursPostfixe(ar->sad) ; // appel récursif
    printf("%d  ", ar->valeur) ;  //traitement
}
```

Exemple :

Le parcours postfixé donne :

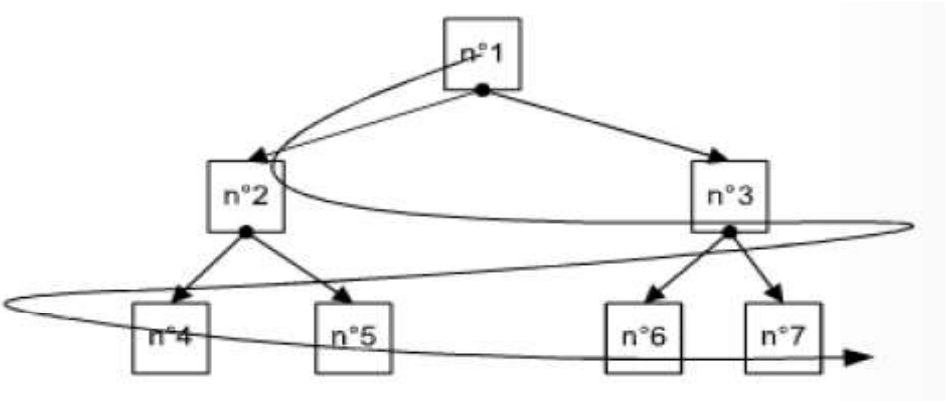
7 4 9 3 2 5



5) LES PARCOURS D'ARBRE

❖ **Parcours en largeur :**

- Tous les nœuds de l'arbre sont atteints depuis la racine, puis couche par couche de gauche à droit.
- Pour écrire cette méthode, il faut introduire une liste d'arbres où seront stockés les nœuds au fur et à mesure de leur passage.
- Il faut ajouter en tête et retirer en queue : une file (FIFO) aurait même suffi.

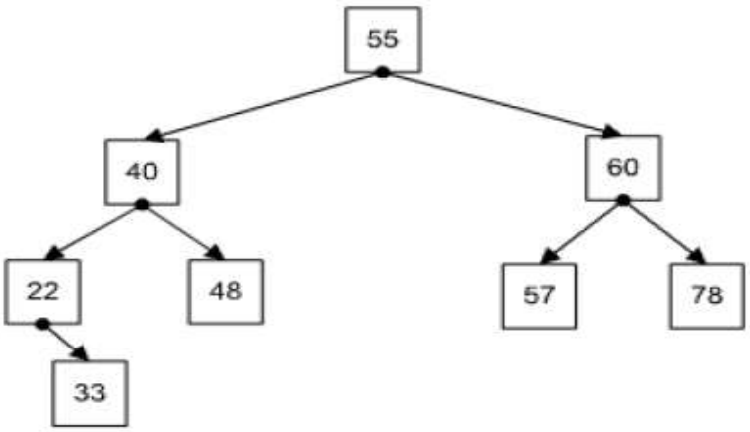


6) ARBRE BINAIRE DE RECHERCHE

➢ Un arbre binaire de recherche (ABR), appelé aussi arbre binaire ordonné, est un arbre tel que la valeur de chaque nœud est supérieure à celle du sous arbre gauche et inférieure à celle du sous arbre droit.

➢ Chaque valeur n'est stockée qu'une seule fois dans l'arbre.

```
typedef struct noeud {
    int valeur ;
    struct noeud *fg ;
    struct noeud *fd ;
} Noeud ;
```



6) ARBRE BINAIRE DE RECHERCHE

➢ **Initialiser l'arbre binaire de recherche :**

Cette fonction initialise les valeurs de la structure représentant l'arbre pointé par a, afin que celui-ci soit vide :

mettre le pointeur sur la racine égal à NULL.

```
void initialiser(Noeud *a) {
    a = NULL ;
}
```


6) ARBRE BINAIRE DE RECHERCHE

➢ **Préparer un nœud :**

- Cette fonction *alloue un nouveau noeud* et *place l'élément (valeur)* e *à l'intérieur.*
- Ses deux fils sont initialisés à la valeur NULL.
- La fonction retourne l'adresse de ce nouveau noeud.
- Au cas où l'allocation de mémoire échoue, NULL est renvoyée.

```
Noeud* preparerNoeud(int e) {
    Noeud *n;
    if ((n = (Noeud*) malloc(sizeof(Noeud))) == NULL) return NULL;
    n->valeur = e;
    n->fg = NULL;
    n->fd = NULL;
    return n;
}
```

6) ARBRE BINAIRE DE RECHERCHE

➢ **Ajouter un nœud :**

Cette fonction ajoute le noeud pointé par **v** dans l'arbre pointé par **b**.

❑ **Algorithme :**

- Parcourir l'arbre à partir de la racine pour descendre jusqu'à l'endroit où sera inséré le nouveau nœud.
- On prendra à gauche si la valeur du noeud visité est supérieure à la valeur du noeud à insérer.
- On prendra à droite si la valeur du noeud visité est inférieure.
- En cas d'égalité, l'insertion ne peut pas se faire et la fonction retourne -1.
- On arrête la descente quand le fils gauche ou droit choisi pour descendre vaut NULL. Le noeud pointé par v est alors inséré à ce niveau.

6) ARBRE BINAIRE DE RECHERCHE

➢ **Ajouter un nœud (version itérative):**

```
int ajouterNoeud_iterative(Noeud *b, Noeud *v) {
    Noeud *a = b, *s = NULL;  // s : parent du nœud courant
    while (a != NULL) {
        s = a ;
        if (v->valeur < a->valeur) a = a->fg;
        else if (v->valeur > a->valeur) a = a->fd;
        else {// valeur déjà existante
            printf("\n Impossible ! le noeud existe deja.\n");
            return -1;
    }
    // insertion du nouveau noeud
    if (v->valeur < s->valeur)  s->fg = v;
    else s->fd = v;
    return 0;
}
```

6) ARBRE BINAIRE DE RECHERCHE

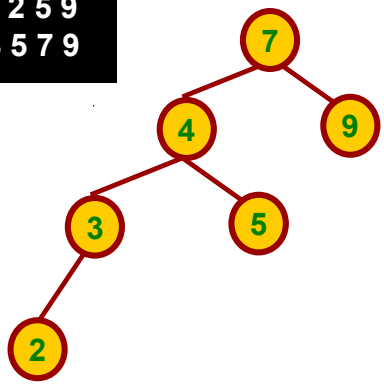
- **Ajouter un nœud (version récursive):**

```
int ajouterNoeud_recursive(Noeud *b, Noeud *v) {
    if (v->valeur < b->valeur) {
        if (b->fg == NULL) { b->fg = v; return 0; }
        else return ajouterNoeud_recursive(b->fg, v);
    }
    else if (v->valeur > b->valeur) {
        if (b->fd == NULL) { b->fd = v; return 0; }
        else return ajouterNoeud_recursive(b->fd, v);
    }
    else return -1; // valeur déjà existante
}
```


6) ARBRE BINAIRE DE RECHERCHE

- **Ajouter un nœud :**

```
Parcours Postfixe :  2 3 5 4 9 7
Parcours Prefixe  :  7 4 3 2 5 9
parcours Infixe   :  2 3 4 5 7 9
```




```
main(){
    int v;
    Noeud *racine;
    initialiser(racine);
    racine = preparerNoeud(7);
    do {
        printf(" donner v "); scanf("%d", &v);
        if (v != 0)  ajouterNoeud_iterative(racine, preparerNoeud(v));
    } while (v!=0);
    printf(" Parcours Postfixe :  "); parcoursPostfixe(racine); printf("\n");
    printf(" Parcours Prefixe  :  "); parcoursPrefixe(racine); printf("\n");
    printf(" Parcours Infixe   :  "); parcoursInfixe(racine); printf("\n");
}
```

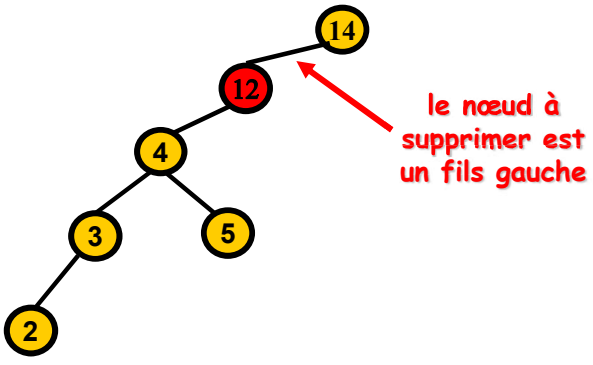
6) ARBRE BINAIRE DE RECHERCHE

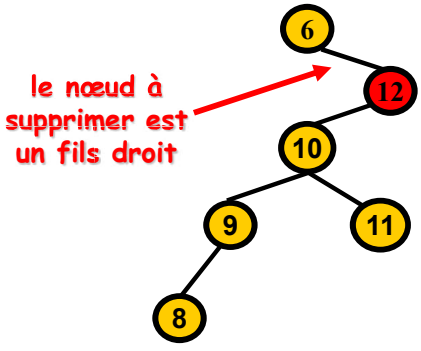
- **Suppression d'un nœud :**
  - Il y a trois cas possibles pour le nœud à supprimer :
    - **Le nœud est terminal (feuille).**
    - **Le nœud a un seul descendant (un fils gauche ou un fils droit).**
    - **Le nœud a deux descendants (un fils gauche et un fils droit).**
  - Pour le dernier cas on remplace le nœud à supprimer par :
    - **Le nœud le plus à droit de son arbre gauche.**
    - **Le nœud le plus à gauche de son arbre droit.**

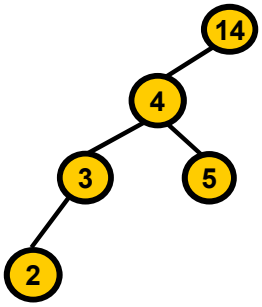
6) ARBRE BINAIRE DE RECHERCHE

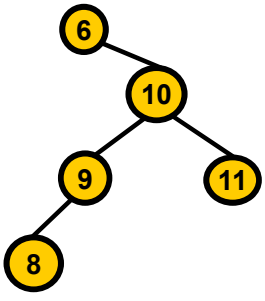
- **Suppression d'un nœud :** Le nœud à supprimer a un seul descendant

Supprimer le nœud de valeur 12 : **le nœud à supprimer possède un fils gauche.**





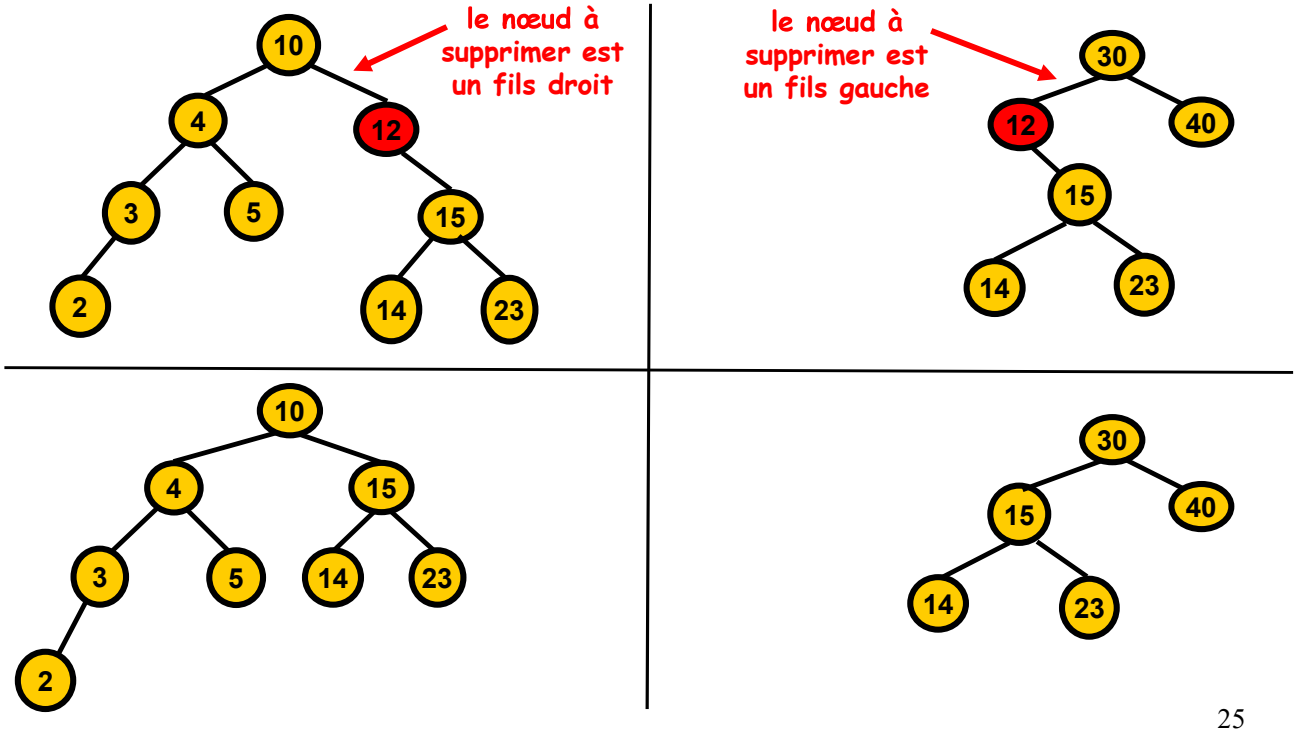




6) ARBRE BINAIRE DE RECHERCHE

➢ **Suppression d'un nœud : Le nœud à supprimer a un seul descendant**

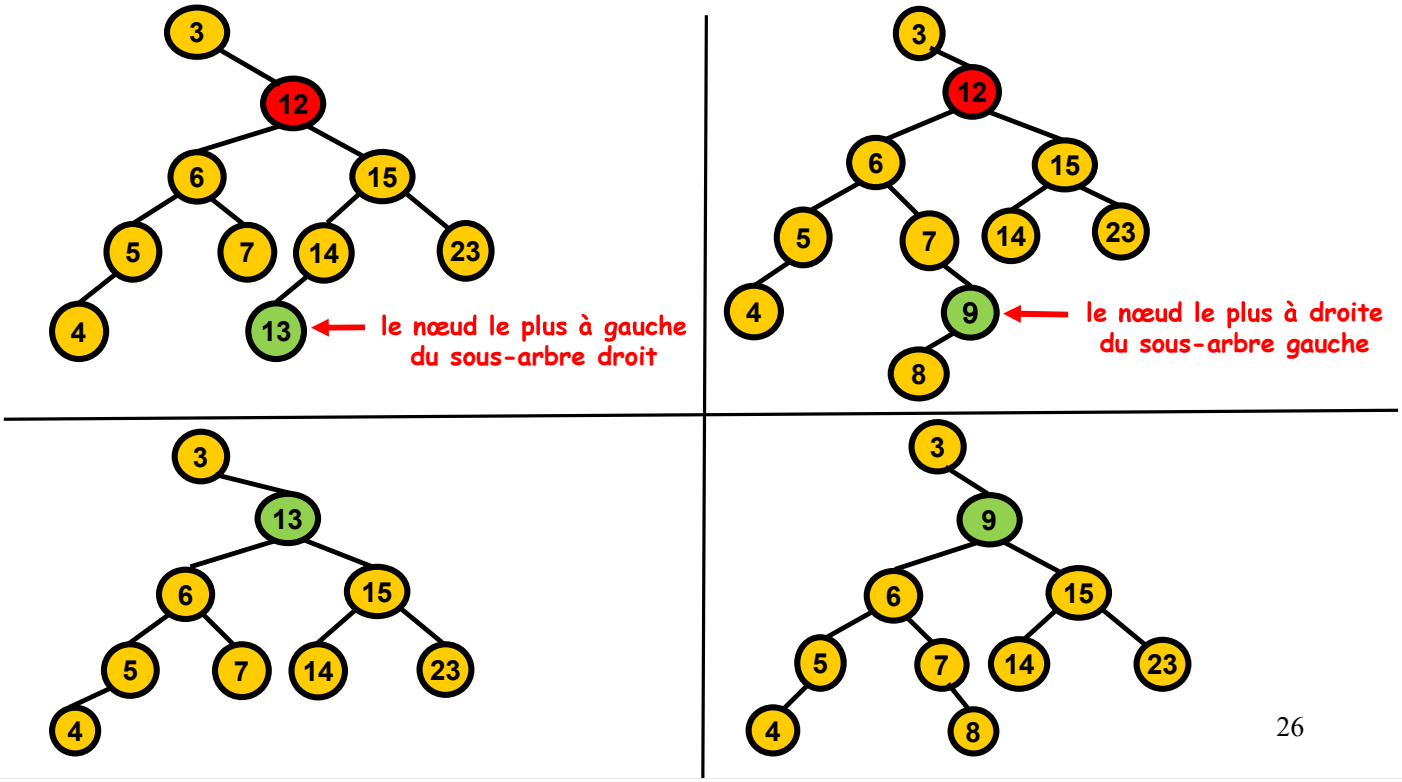
Supprimer le nœud de valeur 12 : **le nœud à supprimer possède un fils droit.**



6) ARBRE BINAIRE DE RECHERCHE

➢ **Suppression d'un nœud : Le nœud à supprimer a deux descendants**

Supprimer le nœud de valeur 12 : on a deux choix



6) ARBRE BINAIRE DE RECHERCHE

- **Suppression d'un nœud :**

Supprimer le nœud de valeur **x** dans un arbre dont la racine est pointé par **p**.

```
void Supprimer(Noeud *p, int x) {
    if (p == NULL) return;
    //recherche du nœud à supprimer
    if (x < p->valeur) Supprimer(p->fg, x);
    else if (x > p->valeur) Supprimer(p->fd, x);
    else { // // p pointe vers le nœud à supprimer
        Noeud *q = p;
        if (q->fg == NULL) { p = q->fd; free(q); }
        else if (q->fd == NULL) { p = q->fg; free(q); }
        else Remplacer_Droite(q); // ou Remplacer_Gauche(q)
    }
}
```


6) ARBRE BINAIRE DE RECHERCHE

- **Suppression d'un nœud : (1er choix)**

Remplacer le nœud pointé par **n** par le nœud le plus **à gauche** de son sous arbre droit. Ce dernier étant lui-même remplacé par son sous arbre droit.

```
Remplacer_Droite (Noeud *n) {
    Noeud *r = n->fd;  // commencer par le fils droite
    Noeud *s = n;  // s est le parent de r
    // descendre jusqu'au plus à gauche de ce sous-arbre droit
    while (r->fg != NULL) { s = r; r = r->fg; }
    n->valeur = r->valeur; // remplacer la valeur de n par celle de r
    // reconnecter le fils droit de r à son parent
    if (s == n) n->fd = r->fd; // r est le fils droit de n
    else s->fg = r->fd;
    free(r);
}
```

6) ARBRE BINAIRE DE RECHERCHE

➢ Suppression d'un nœud : (2ème choix)

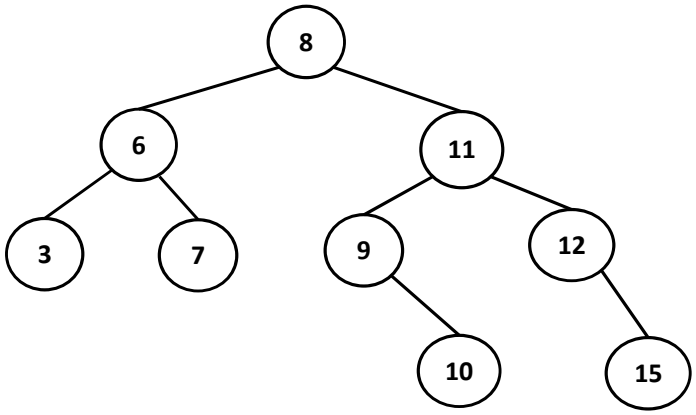
Remplacer le nœud pointé par **n** par le nœud le plus **à droite** de son sous arbre **gauche**. Ce dernier étant lui-même remplacé par son sous arbre gauche.

```
Remplacer_Gouche (Noeud *n) {
    Noeud *r = n->fg;  // commencer par le fils gauche
    Noeud *s = n;      // s est le parent de r
    // descendre jusqu'au plus à droite de ce sous-arbre gauche
    while (r->fd != NULL) { s = r; r = r->fd; }
    n->valeur = r->valeur; // remplacer la valeur de n par celle de r
    // reconnecter le fils gauche de r à son parent
    if (s == n)  n->fg = r->fg; // r est le fils gauche de n
    else  s->fd = r->fg;
    free(r);
}
```


6) ARBRE BINAIRE DE RECHERCHE

➢ Exercice d'application :

Soit l'arbre binaire de recherche (ABR) suivant:



1) Insérer les éléments suivants: **2**, **11**, **5**, **13** et **16** dans l'ABR.
2) Afficher l'ABR sous les trois types de parcours vus en cours.
3) Comment peut-on supprimer le nœud **11** de l'ABR.