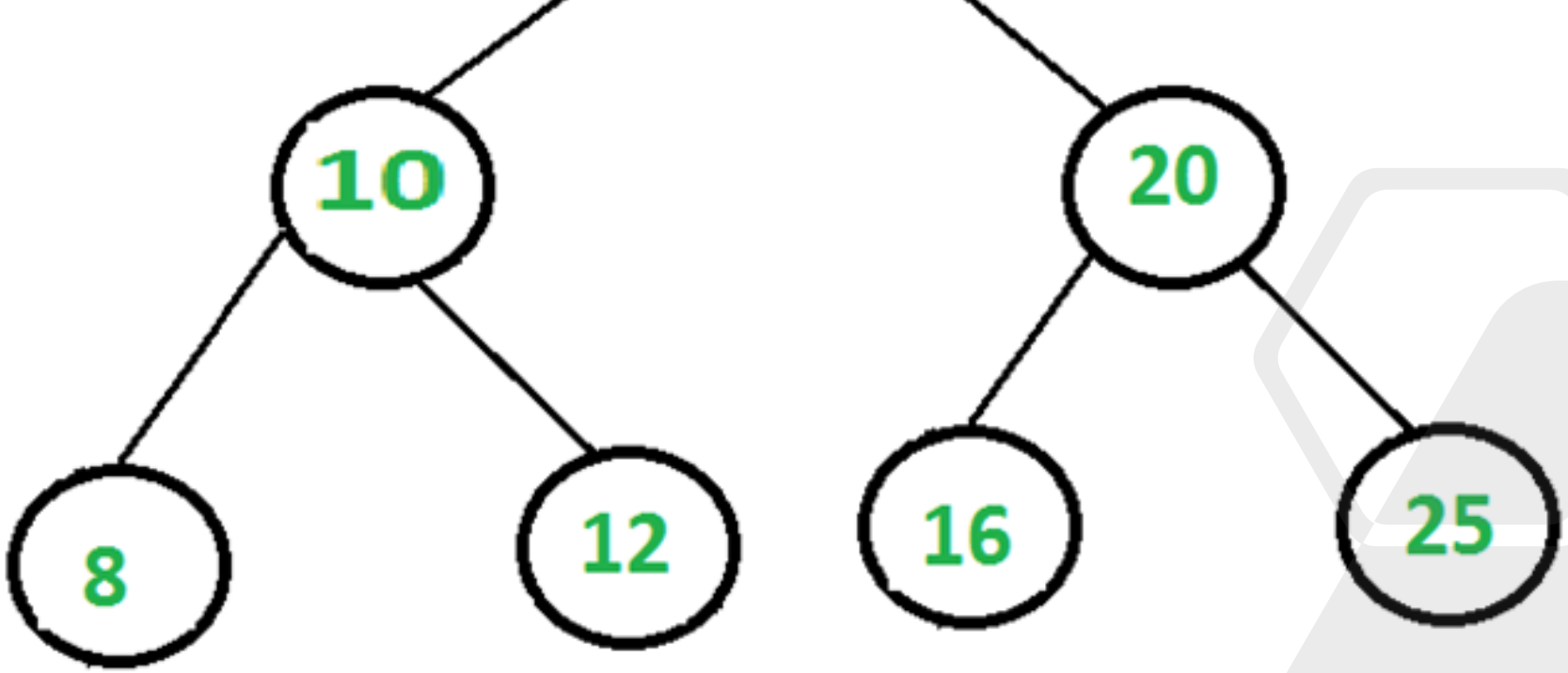
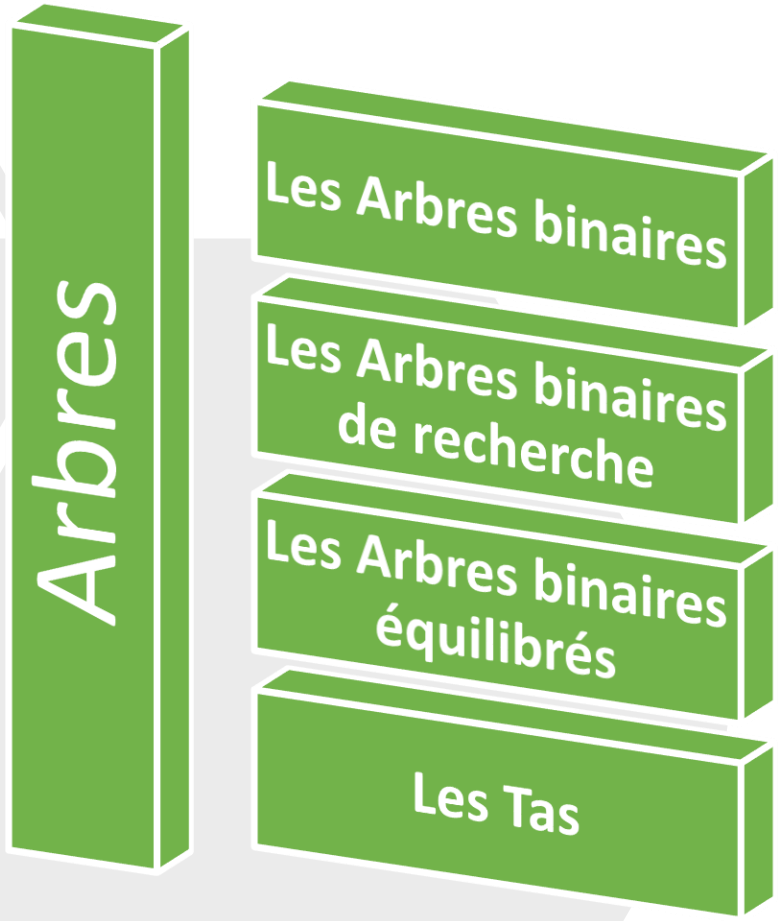
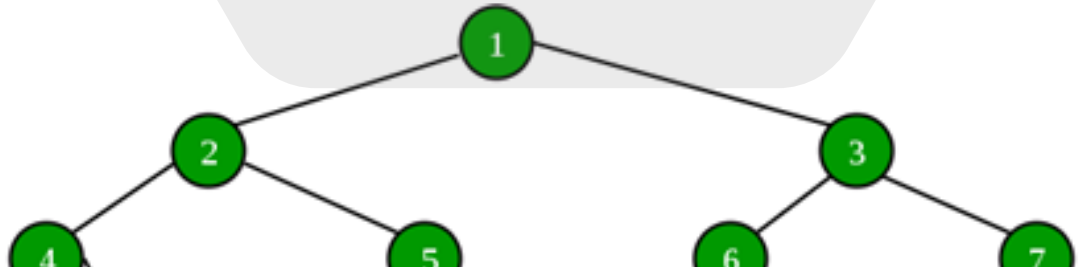




Chapitre 3:

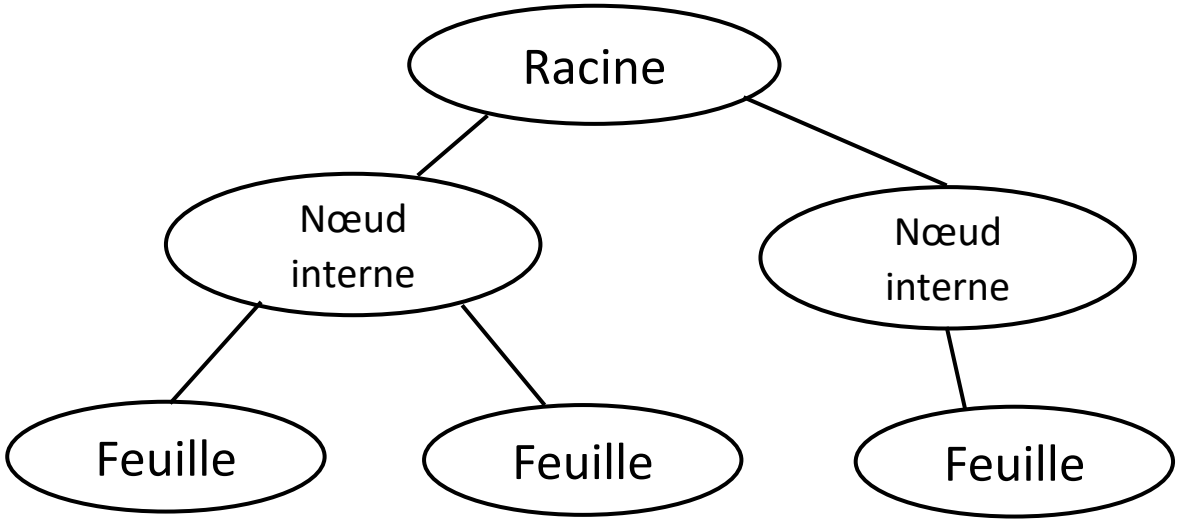
Structures hiérarchiques



Algorithmique & Complexité
BENAZZOUZ 2024-2025

1. Les Arbres

L'arbre est une structure de donnée qui généralise la liste : alors qu'une cellule de liste a un seul successeur (leur suivant), dans un arbre il peut y en avoir plusieurs. On parle alors de **noeud** (au lieu de cellule).

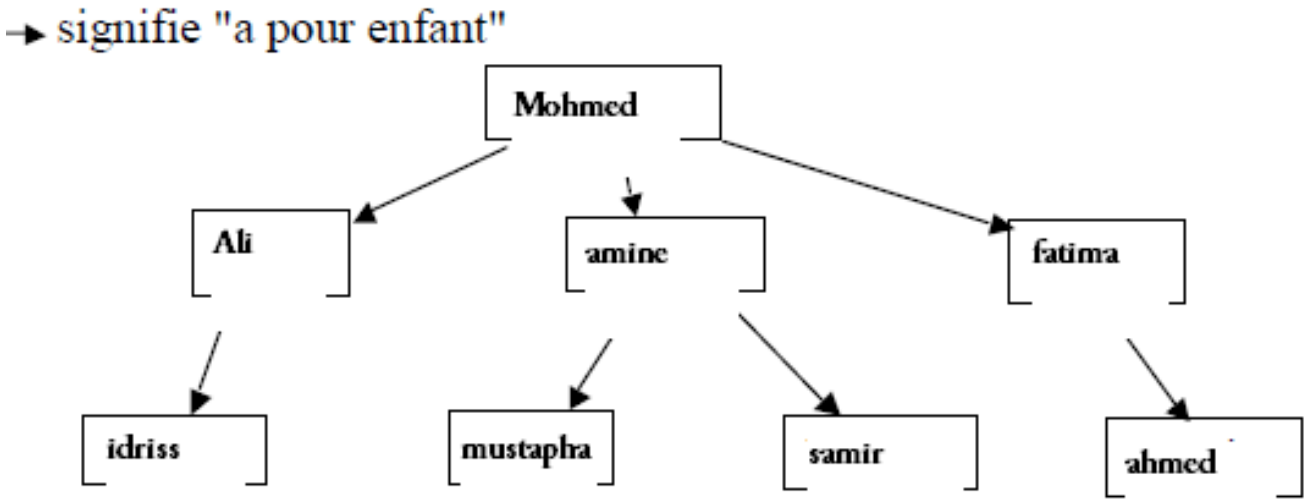


Un **noeud père** peut avoir plusieurs **noeud fils**. Un fils n'a qu'un seul père, et tous les noeuds ont un ancêtre commun appelé la **racine** de l'arbre (le seul noeud qui n'a pas de père).

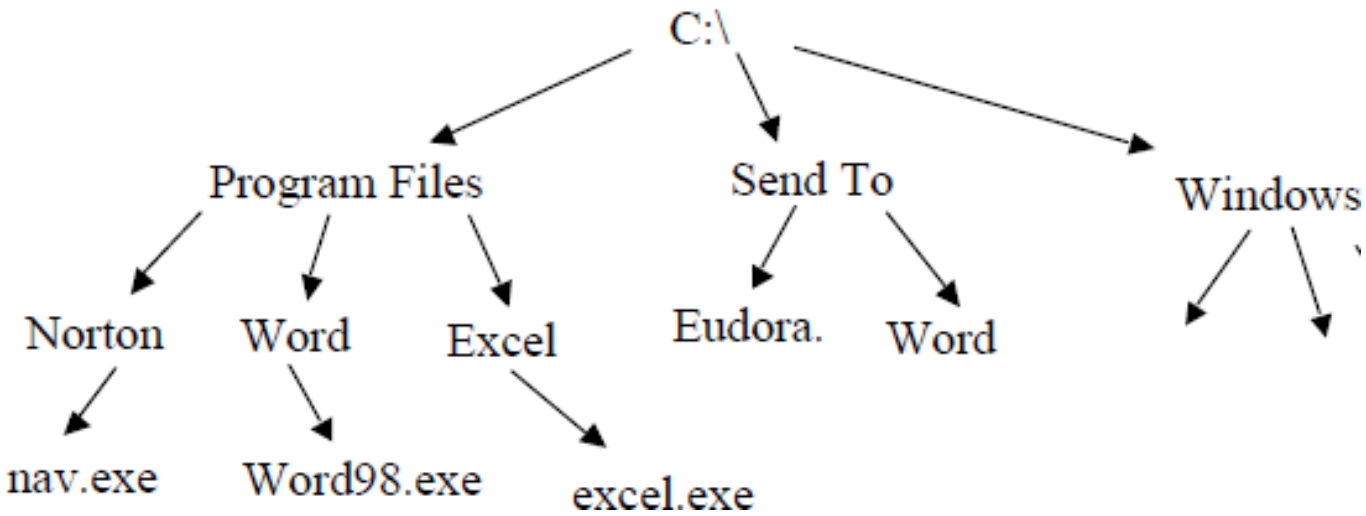
Dans un arbre, on distingue :

- les feuilles, éléments ne possédant pas de fils dans l'arbre ;
- les nœuds internes, éléments possédant des fils (sous-branches).
- La racine de l'arbre est le nœud ne possédant pas de parent.

- **Exemple 1 :** Arbre généalogique



- **Exemple 2 :** arborescence des répertoires



2. Représentation :

On indique ici une représentation par listechaînnéee. Il en existe d'autres,

Type **cellule** = {

valeur : Elem ;

fils : adresse d’une cellule; // le 1er fils

frere : adresse d’une cellule ; // le fils suivant du même père …

}

Type **arbre** = {

Racine : adresse d’une cellule

}

Quand A a deux fils, son fils gauche est A.fils et son fils droit est A.fils.frère

3. Primitives :

sommet racine(arbre A) : retourne la racine

sommet premier_fils(sommet x,arbre A) : retourne le fils le plus à gauche, ou VIDE

sommet frère(sommet x,arbre A) : le premier frère à sa droite

element val(sommet x,arbre A)

booleen arbre_est_vide(arbre A)

booleen existe_fils(sommet x,arbre A)

booleen existe_frère(sommet x,arbre A)

sommet père(sommet x,arbre A)

sommet ieme_fils(sommet x, entier i,arbre A)

arbre creer_vide()

creer_sommet(element e, arbre A)

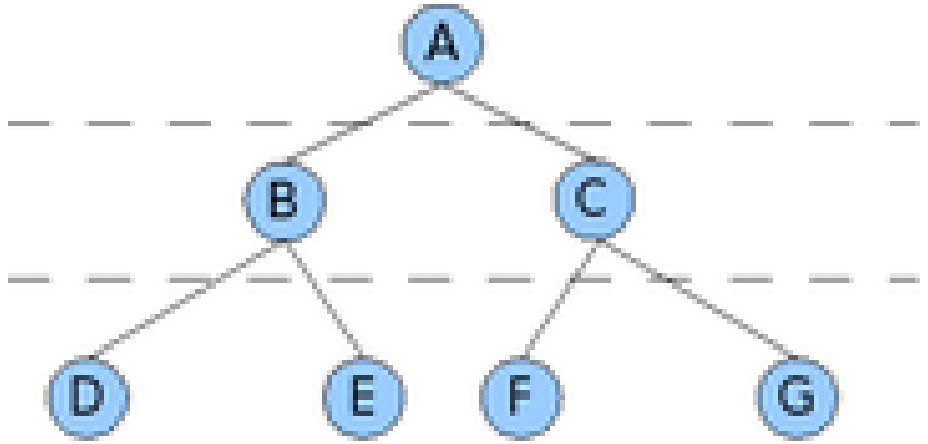
fait_racine(sommet,arbre)

inserer_fils(sommet pere,sommet nouv, entier place, arbre A)

supprimer_feuille(sommet,arbre)

4. Parcours d'un arbre :

4.1. Parcours en largeur :



On commence par la racine, puis tous ses fils, puis

les fils des fils etc… **A, B, C, D, E, F, G**

4.2 Parcours en profondeur :

Le parcours en profondeur est un parcours récursif sur un arbre. Il existe trois ordres pour cette méthode de parcours.

Parcours en profondeur préfixé :

Dans ce mode de parcours, le nœud courant est traité **avant** le traitement des nœuds gauches et droits. Le parcours sera **A, B, D, E, C, F puis G**.

Parcours en profondeur infixé :

Le nœud courant est traité **entre** le traitement des nœuds gauches et droits. Le parcours sera **D, B, E, A, F, C puis G**.

Parcours en profondeur suffixé ;

Le nœud courant est traité **après** le traitement des nœuds gauches et droits. Le parcours sera **D, E, B, F, G, C puis A**.

Algorithmes de Parcours

A. Parcours en largeur:

```
Algorithme Parcours (arbre A)
    locales : file F, sommet x
    debut
      CreerFile(F)
      Enfiler(racine(A),F)
      TantQue non file_vide(F) faire
            x<-SommetFile(F)
            Defiler(F)
            Si existe_fils(x) Alors
              Enfiler(x,F)
              TantQue existe_frere(x) faire
                 x<-frere(x)
                 Enfiler(x,F)
              Ftq
            FinSi
        Ftq
    fin
```

B. Parcours en profondeur

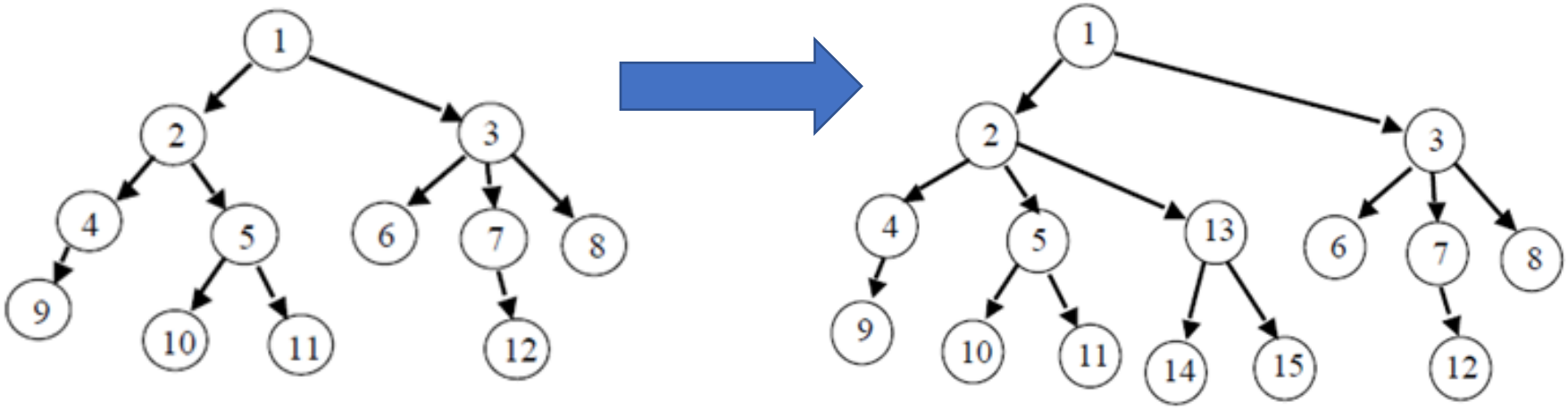
Algo récursif

```
Algorithme Parcours(arbre A)

    Parcours_rec(racine(A),A)
```

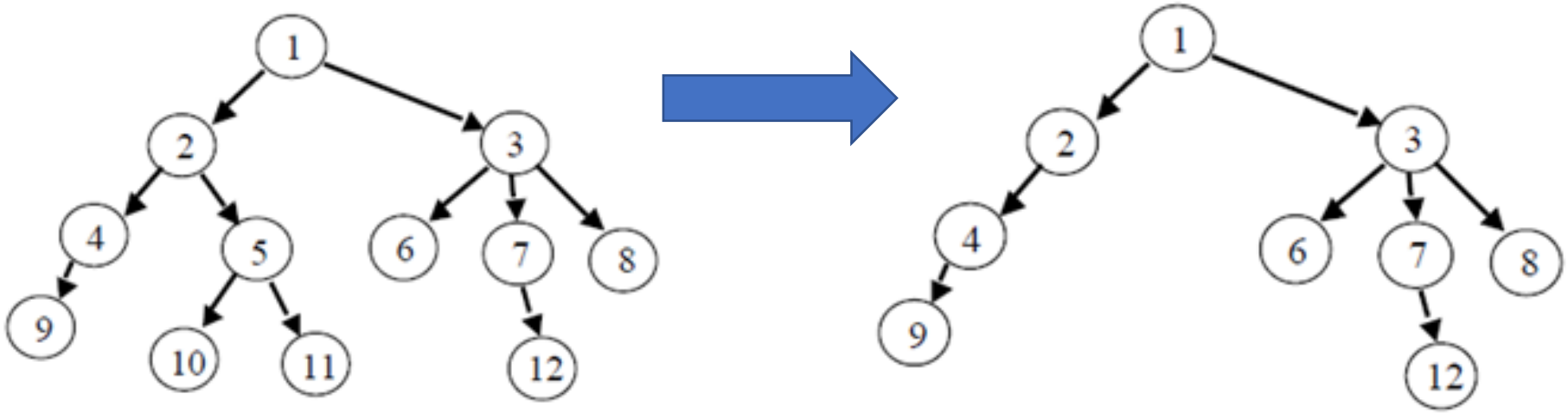
```
Algorithme Parcours_rec(sommet r , arbre A)
    // parcours du sous arbre de racine r
    locales : sommet f
    debut
        si r<>VIDE alors
            f <- premier_fils(r,A)
            tant que f<>VIDE faire
                Parcours_rec(f,A)
                f <- frere(f,A)
            fin tant que
        fin si
    fin
```

5-Opérations :

Ajout :



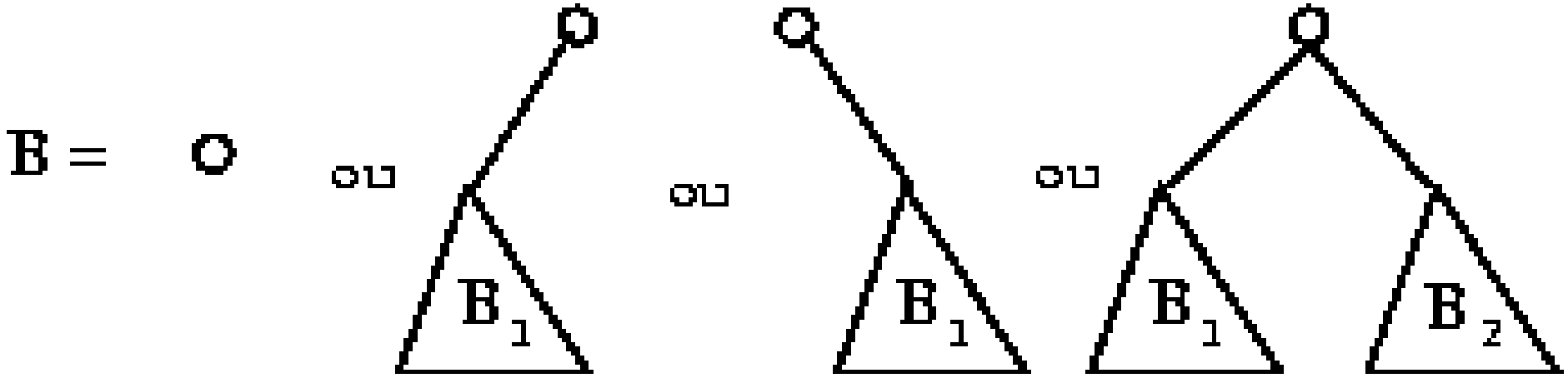
Suppression :



6. Arbre binaire (Binary Tree) :

6.1 Définitions : Soit un ensemble de ***sommets*** (ou encore ***noeuds***) auxquels sont associés des "valeurs" (les éléments à stocker : entier, chaîne, structure, ...). Un arbre binaire est défini récursivement de la manière suivante, un arbre binaire est composé de :

1- soit d'un seul sommet appelé ***racine***,
2- soit d'un sommet racine à la gauche duquel est accroché un ***sous-arbre*** binaire ***gauche***
3- soit d'un sommet racine à la droite duquel est accroché un sous-arbre binaire ***droit***
4- soit d'un sommet racine auquel sont accrochés un sous-arbre binaire droit et un sous-arbre binaire gauche.



6.2 Primitives:

sommet racine(arbre B) : retourne la racine de l'arbre

sommet fils_gauche(sommet x,arbre B) : retourne le sommet fils gauche de x ou VIDE

sommet fils_droit(sommet x,arbre B) : retourne le sommet fils droit de x ou VIDE

sommet pere(sommet x,arbre B) : retourne le sommet père de x ou VIDE

element val(sommet x,arbre B) : retourne l'élément stocké au sommet x

booleen est_vide(arbre B) : retourne vrai ou faux

On peur définir d'autres primitives :

Sommet interne, feuille, hauteur , profondeur etc…

6.3 Arbres binaires particuliers :

Définitions 1: On appelle arbre binaire ***complet*** un arbre binaire tel que chaque sommet possède 0 ou 2 fils.

Théorème 1 Un arbre binaire complet possède $2P+1$ sommets (nombre impair) dont P sommets internes et $P+1$ feuilles.

Définitions 2: On appelle arbre binaire ***parfait*** un arbre binaire (complet) tel que chaque sommet soit père de **deux** sous arbres de **même hauteur**.

Théorème 2 Un arbre binaire parfait possède $2^{h+1}-1$ sommets, où h est la hauteur de l'arbre.

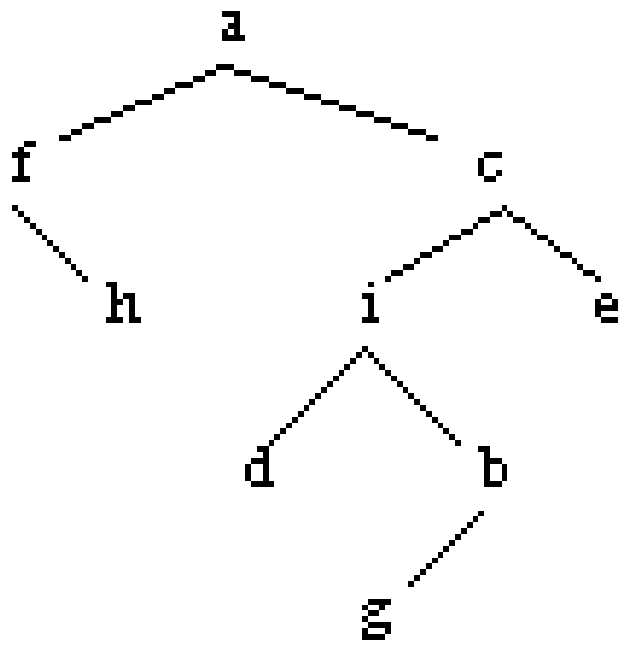
6.4 Implémentations :

- Tableaux FG et FD
- Listes chaînées

6.4.1 Arbre binaire sous forme de tableaux FG et FD :

type **sommet** = entier

type **arbre** = adresse de {
nbSommets : entier
FG : tableau [1..MAX] de sommets
FD : tableau [1..MAX] de sommets
VAL : tableau [1..MAX] d'elements
}



Par exemple, l'arbre précédent peut avoir comme implémentation :

A=

	1	2	3	4	5	6	7	8	9	10 ...	MAX
FG	9	5	0	6	0	0	0	0	3		
FD	8	1	0	0	7	0	0	0	4		
VAL	c	a	d	b	f	g	h	e	i		

NbSommets : 9

6.4.2 Arbre binaire sous forme de listes chaînées :

type **sommet** = adresse de {

FG : adresse d'un sommet

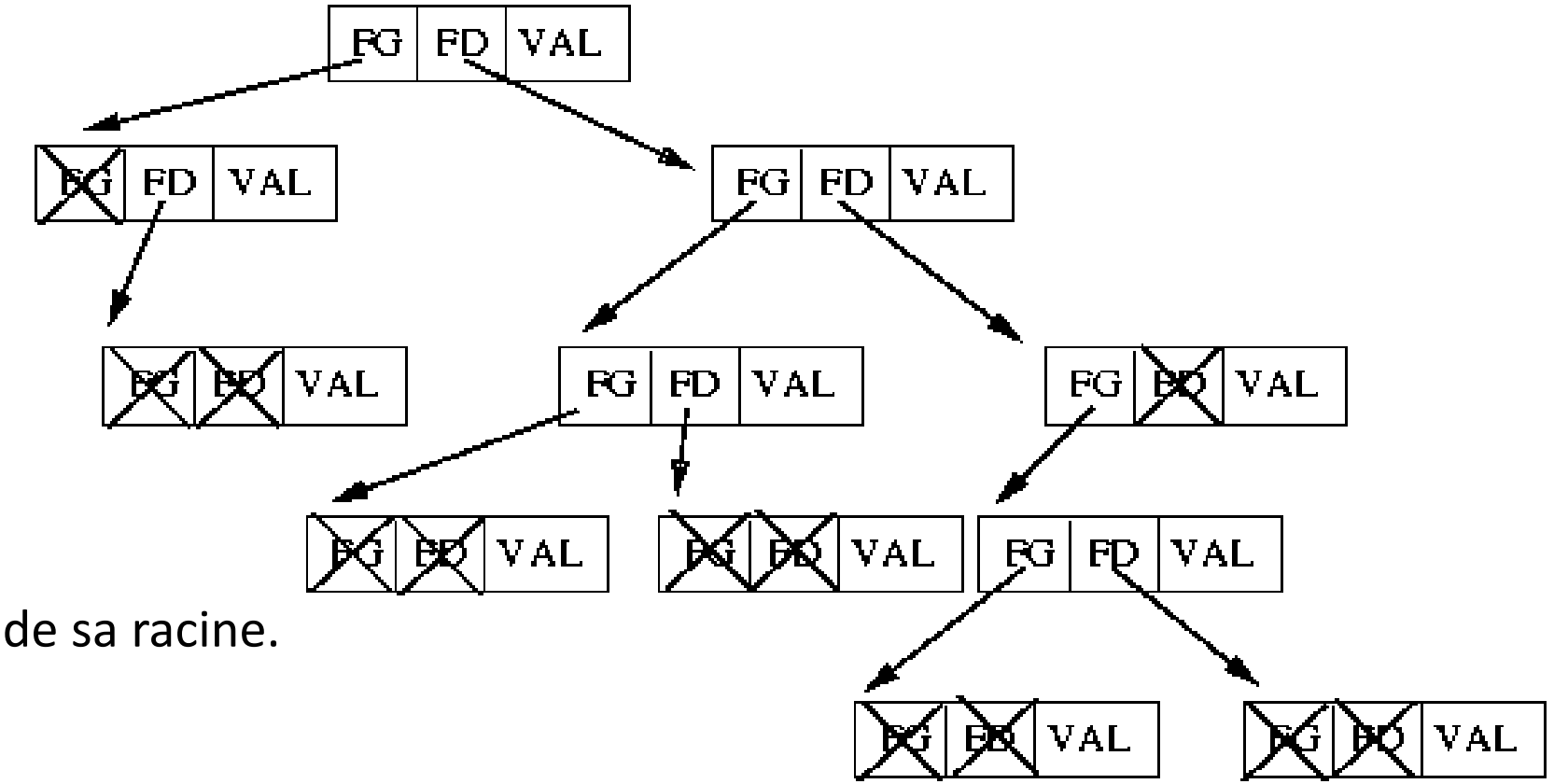
FD : adresse d'un sommet

val : element

}

type **arbre** = sommet

Un arbre est donné par l'adresse de sa racine.



Algorithme pere(sommet x,arbre B) : retourne un sommet

```
   // recherche RECURSIVE du pere de x dans le sous arbre (de racine) B
locales :       sommets :  tmp,p
   debut
     si (est_vide(B)) retourner VIDE
     sinon
        p = B   // racine de B
        si (p=x) alors retourner VIDE // cas particulier pere de la racine
        sinon
           si ( p->FG=x) ou (  p->FD=x) alors  retourner p
           sinon
```

```
            tmp = pere(x,  p->FG ) // recherche récursive dans le sous arbre gauche
            si (tmp=VIDE)    // s'il n'est pas a gauche, c'est qu'il est a droite
                  tmp = pere(x,  p->FD)
            finsi
            retourner tmp
        finsi
    finsi
  finsi
fin
```

Donc si c'est souvent utilisé, il sera plus commode de rajouter un pointeur vers le père dans la structure :

```
type sommet = adresse de {
                    FG : adresse d'un sommet
                    FD : adresse d'un sommet
                    PERE : adresse d'un sommet
                    val : element
                  }
type arbre = sommet
```

Ps : soit laisser la structure précédente et appeler l'Algo autant de fois soit rajouter un pointeur vers le père

Quelques primitives de constructions/modifications :

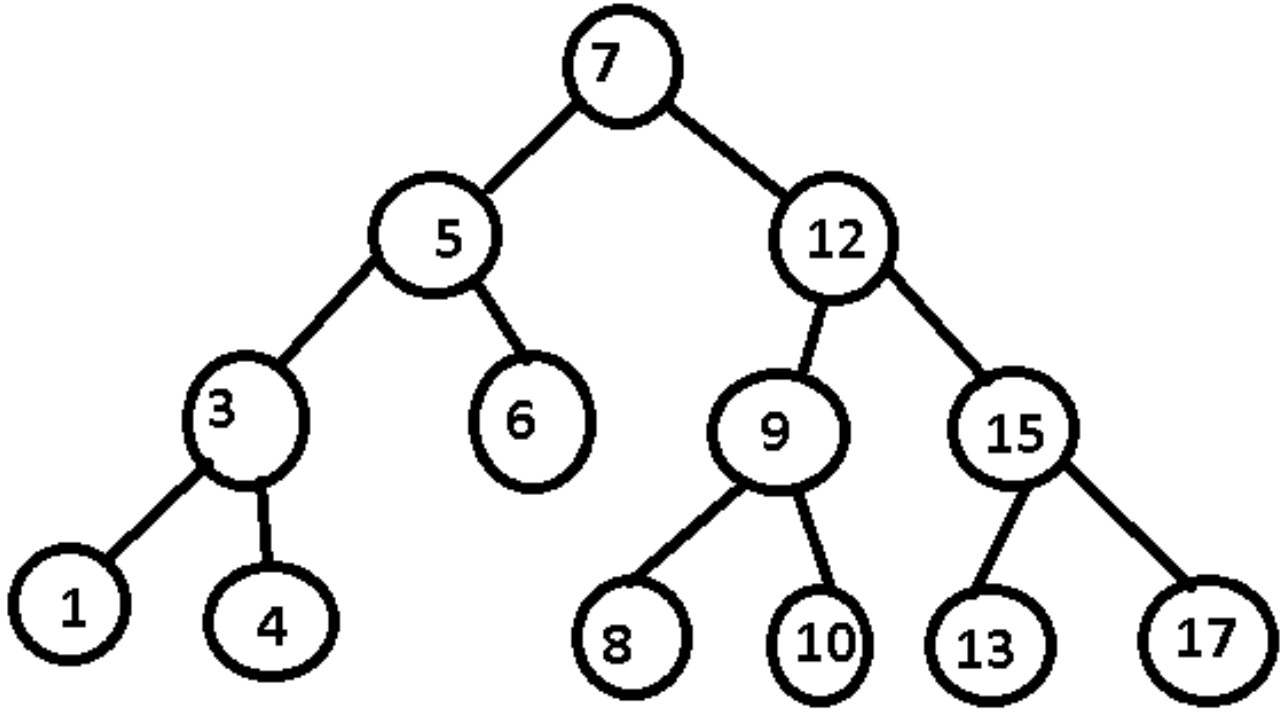
```
Algorithme fait_FG(sommet p,sommet fg,arbre B)

// avec cette implementation B en fait n'est pas modifie

      debut

            p->FG = fg

           // fg->pere = p

      fin
```

```
Algorithme supprime_feuille(sommet f, E/S arbre B)

   // l'arbre peut devenir vide donc etre modifie !!!!

   locales : sommet p

   debut

      si (f=B) alors   // la feuille = la racine

            liberer ( f )

            B = VIDE

      sinon
```

```
        p = pere(f)    // p = f->pere ?
        si  p->FG=f  alors     // feuille a gauche
            p->FG = VIDE
        sinon // feuille a droite
            p->FD = VIDE
        finsi
        liberer( f )
    finsi
fin
```

Arbre Binaire de Recherche

Binary Search Tree(**BST**)

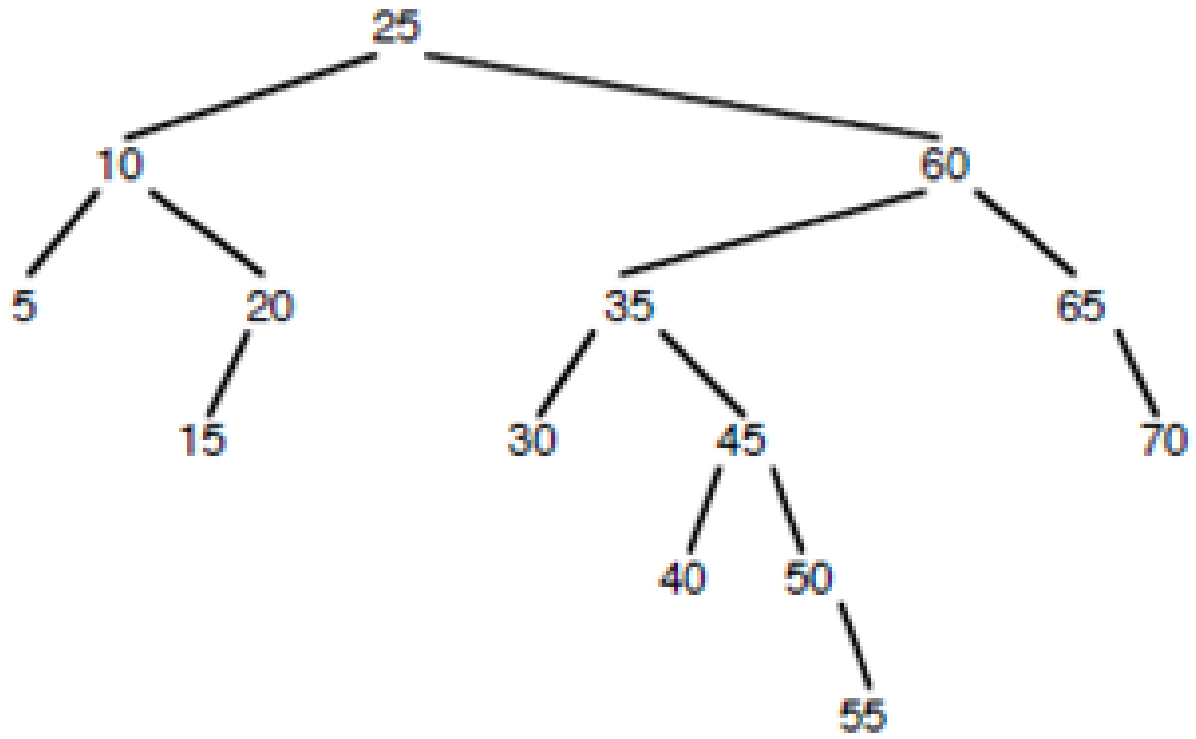


Arbre Binaire de Recherche

1. Définitions : Un *arbre binaire de recherche (ABR)* est un arbre binaire tel que la valeur stockée dans chaque sommet est supérieure à celles stockées dans son sous arbre gauche et inférieure à celles de son sous arbre droit.

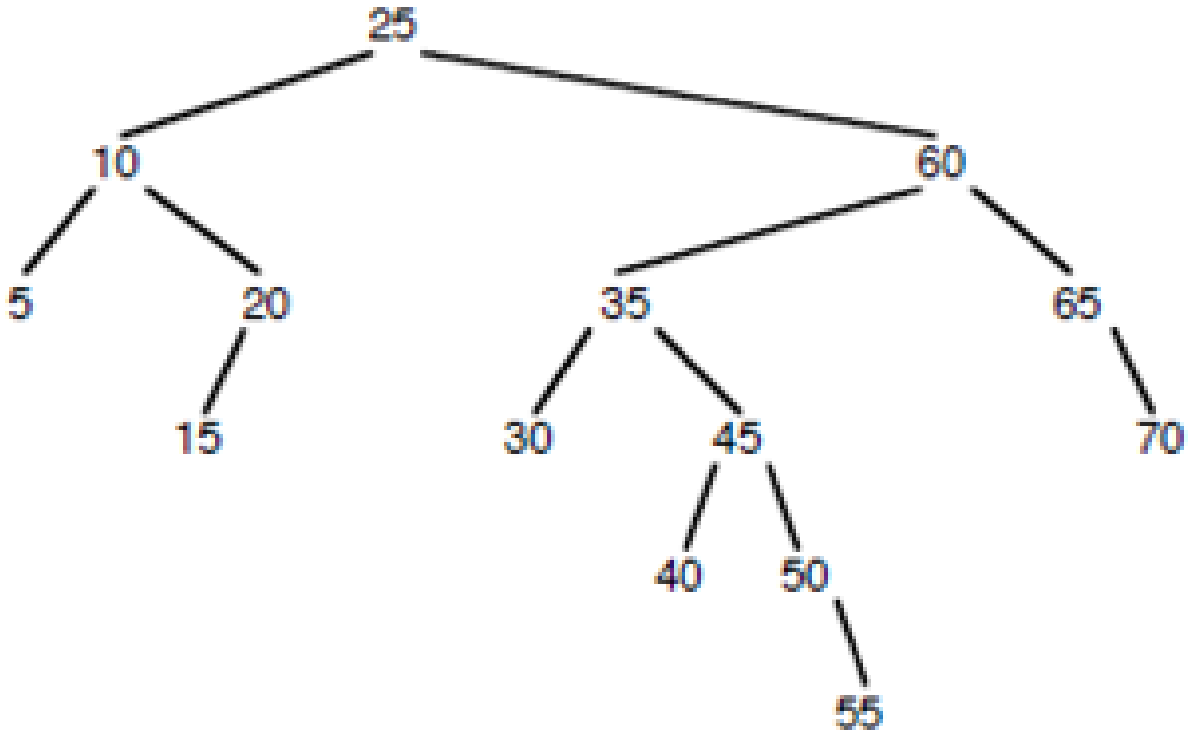
Exemple :

25	60	35	10	5	20	65	45	70	40	50	55	30	15



Exemple :

25	60	35	10	5	20	65	45	70	40	50	55	30	15



Une fois l'ABR construit, il n'y a plus qu'à exploiter l'une de ses propriétés pour trier l'ensemble des éléments. Il suffit d'effectuer le parcours infixe de l'arbre.

2. Recherche :

Recherche d'un élément e dans un arbre binaire de recherche

```
Algorithme Recherche(element e, arbre B): retourne un booleen

    debut
        retourner
         Rech_Aux(e,racine(B),B)
   fin
```

```
Algorithme Rech_Aux(element e, sommet r, arbre B) : retourne un booleen
// verifie si e appartient au sous-arbre de racine r (qui peut etre vide)
      debut
         Si r=VIDE Alors
                  retourner FAUX // l'élément n'est pas dans l'arbre
         Sinon
               Si ( r->val = e) Alors

                        retourner VRAI  // on l'a trouve
               Sinon
                     Si ( e < r->val ) Alors
                    // s'il existe, c'est dans le sous-arbre gauche
                           retourner
                           Rech_Aux(e, r->FG ,B)
```

```
Algorithme Rech_Aux(element e, sommet r, arbre B) : retourne un booleen
// suite...

                    Sinon
                    // s'il existe, c'est dans le sous-arbre droit
                       retourner
                        Rech_Aux(e, r->FD ,B)
                    Finsi
               Finsi
          Finsi
     FIN
```

3. Ajout :

Ici, on ajoute le sommet en tant que feuille.

Ajout d'un élément ***e*** dans un arbre binaire de Recherche.

- On suppose que l'élément n'est pas déjà dans l'arbre.
- l'arbre peut être vide.
- l'arbre sera modifié.

On pourrait aussi ajouter en tant que racine :

- couper l'arbre en deux sous arbres (binaires de recherche) G et D, G contenant tous les éléments de valeur plus petite que celle à ajouter, et D contenant tous les éléments de valeur plus grande que celle à ajouter,
- créer un nouveau sommet de valeur voulue en lui mettant G comme sous-arbre gauche et D comme sous arbre droit.

```
Algorithme Ajout(element e, E/S arbre B)
       locales : sommets r,y
    debut
      si est_vide(B) Alors
          y <- creer_sommet(e,B)
          fait_racine(y,B)
      sinon
          r=racine(B)
          si e < r->val à alors
                Ajout_aux(e,r->FG ,r,gauche,B)
          sinon Ajout_aux(e, r->FD ,r,droit,B)
          finsi
      finsi
    fin
```

```
Algorithme Ajout_aux(element e,sommet r, sommet pere,  lien,E/S arbre B)
// ajout dans le sous arbre de racine <r> qui est fils de <pere>
// <lien> egal droit ou gauche suivant que <r> est fils gauche ou droit de son pere
    locale : sommet nouv
    debut
     si r =VIDE alors
        nouv <- creer_sommet(e,B) // c'est ici qu'il faut l'ajouter
        si lien=droit alors    faire_FD(pere,nouv,B)
        sinon                  faire_FG(pere,nouv,B)
        finsi
```

```
Algorithme Ajout_aux(element e,sommet r, sommet pere,  lien,E/S arbre B)
// suite . . .


    sinon
        si   e <  r-> val  alors
            Ajout_aux(e,  r->FG  ,r ,gauche ,B)
        sinon
            Ajout_aux(e,  r->FD  ,r ,droit ,B)
        finsi
    finsi
fin
```

4. Suppression :

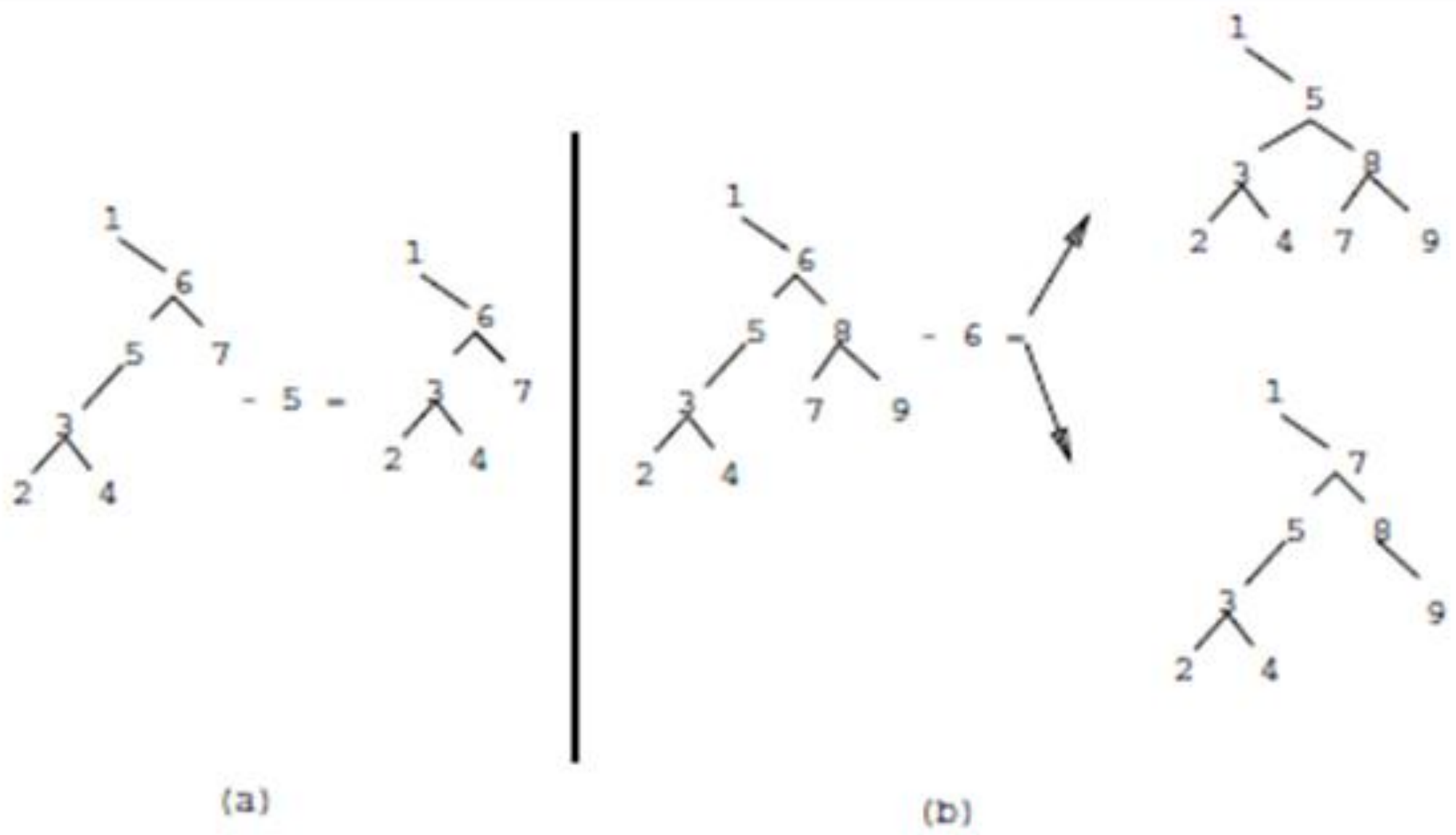
La suppression commence par la recherche de l'élément.

Puis :

1. si c'est une feuille, on la supprime directement
2. si c'est un sommet qui n'a qu'un fils, on le remplace par ce fils
3. si c'est un sommet qui a deux fils, on a deux solutions :
   - le remplacer par le sommet de plus grande valeur dans le sous-arbre gauche
   - le remplacer par le sommet de plus petite valeur dans le sous-arbre droit

   puis supprimer (récursivement) ce sommet !

Exemple :



5. ABR Particuliers :

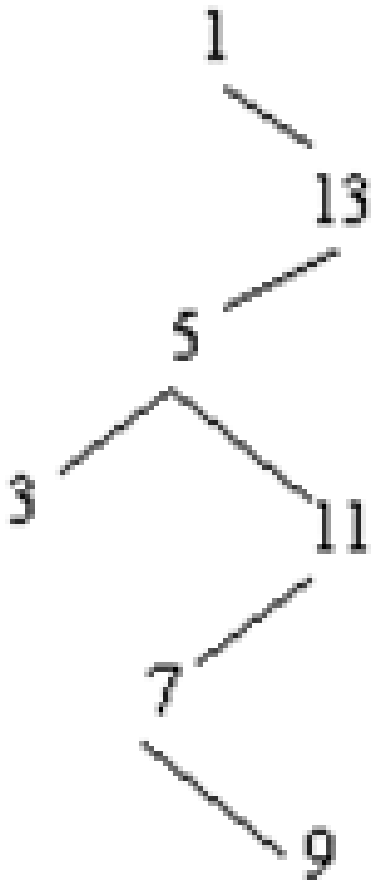
- **Arbres binaires de recherche équilibrés : AVL (Adelson-Velsky & Landis)**

Un arbre binaire est dit **H-équilibré (quasi-parfait)** si en **tout sommet** de l'arbre les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.

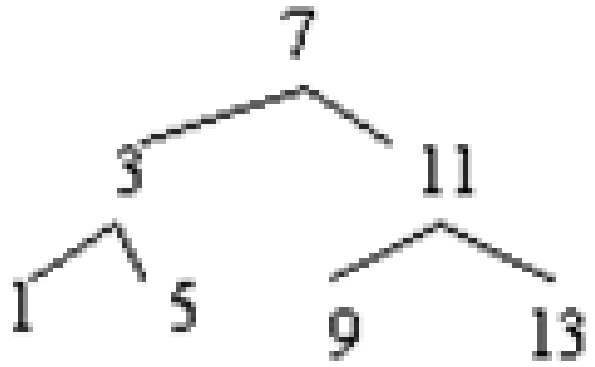
- *Rappel : complet (0 ou 2 fils); parfait (sommet est père 2 sous arbres même hauteur), quasi (diff pas plus1)*

Illustration :

- Recherche le 8 ?



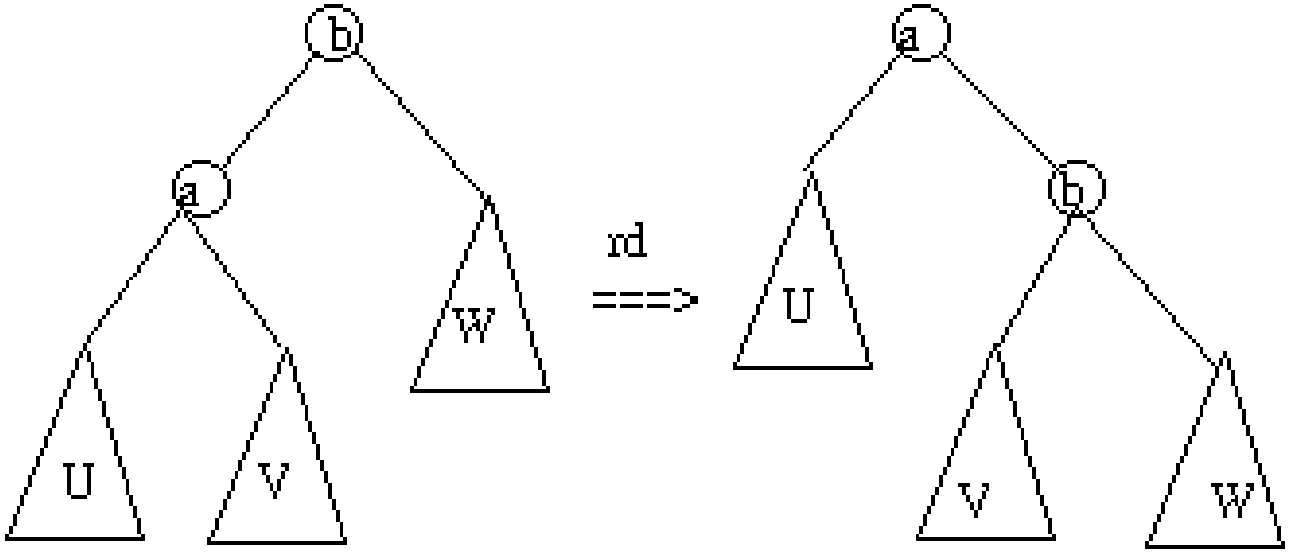
==> 1 3 5 79 11 13 ==>



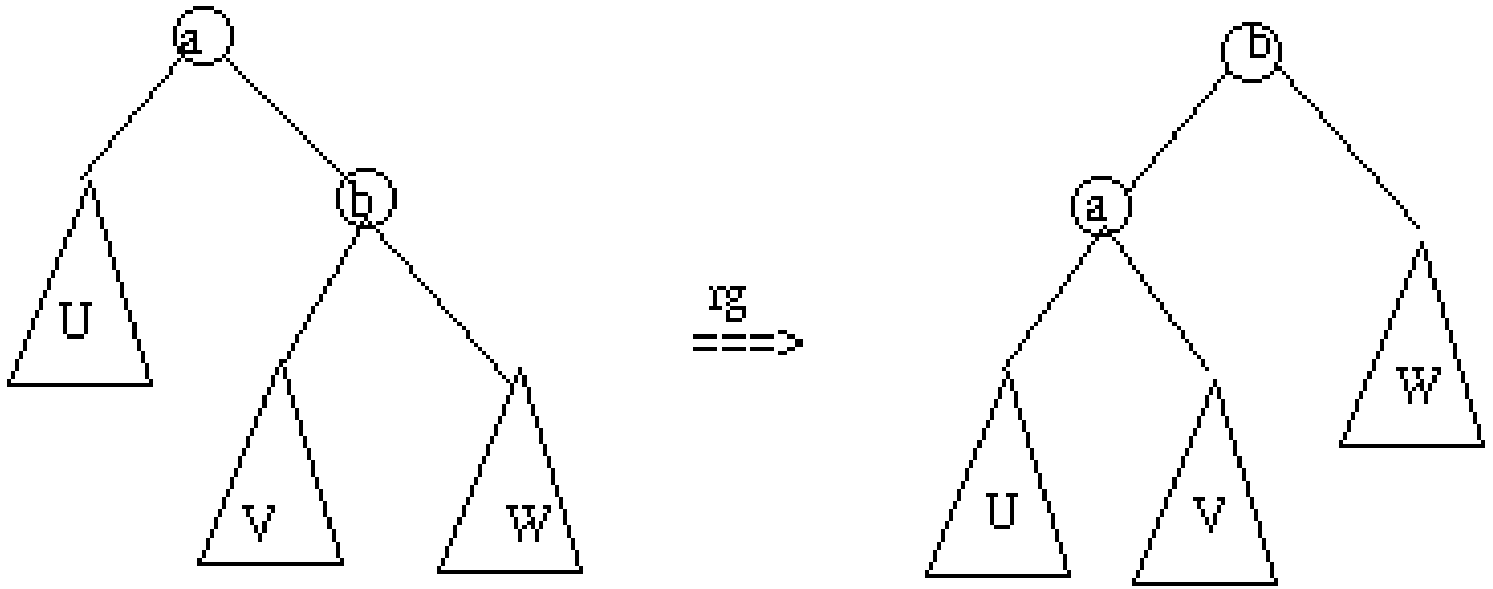
Opérations : *les mêmes algorithmes que pour un arbre binaire de recherche, à ceci près qu'il faut ajouter des rotations de rééquilibrage.*

Après chaque ajout ou suppression, vérifier si cela a entraîné un déséquilibre et donc rééquilibre par des **rotations**.

- **Rotation droite**

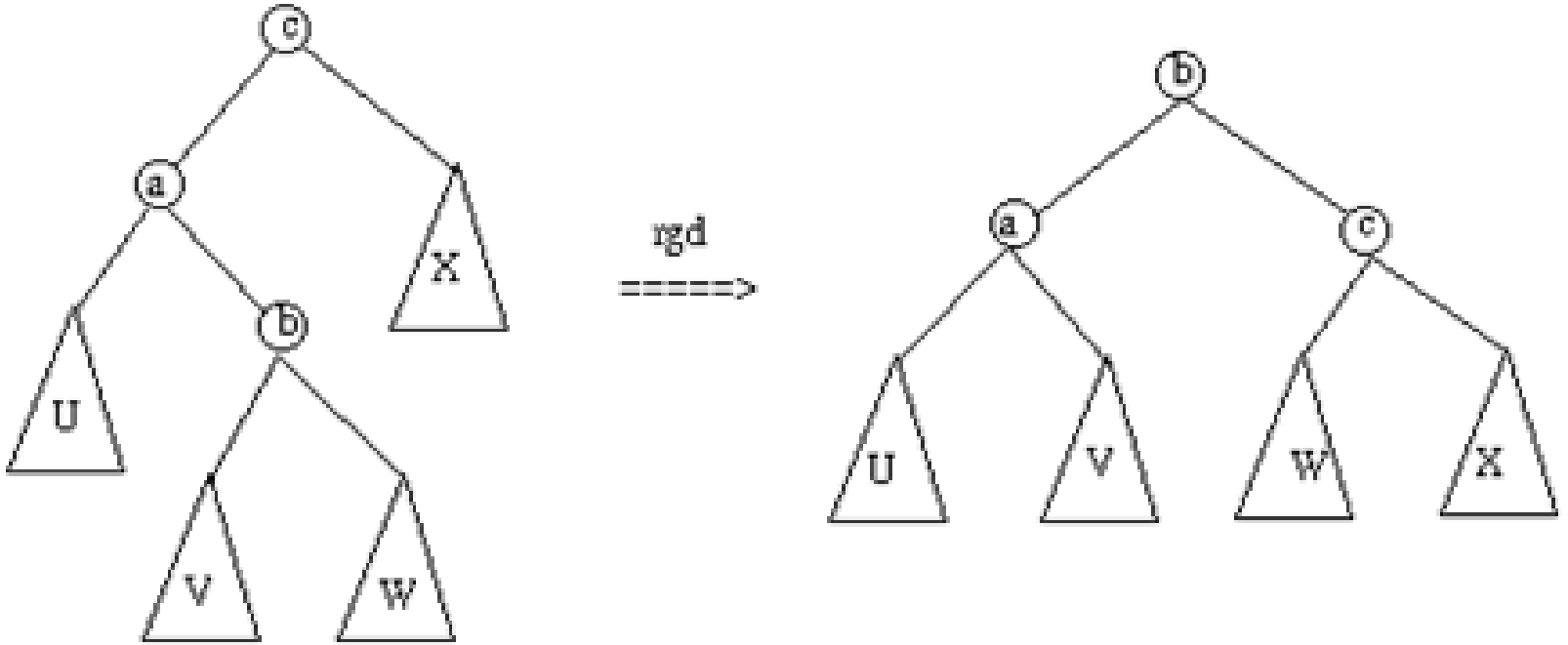


- **Rotation gauche**



Rotation gauche droite :

Cette rotation peut se voir aussi comme une rotation gauche du sous-arbre gauche suivie d'une rotation droite.



Rotation droite gauche :

rotation droite du sous-arbre droit suivie d'une rotation gauche.

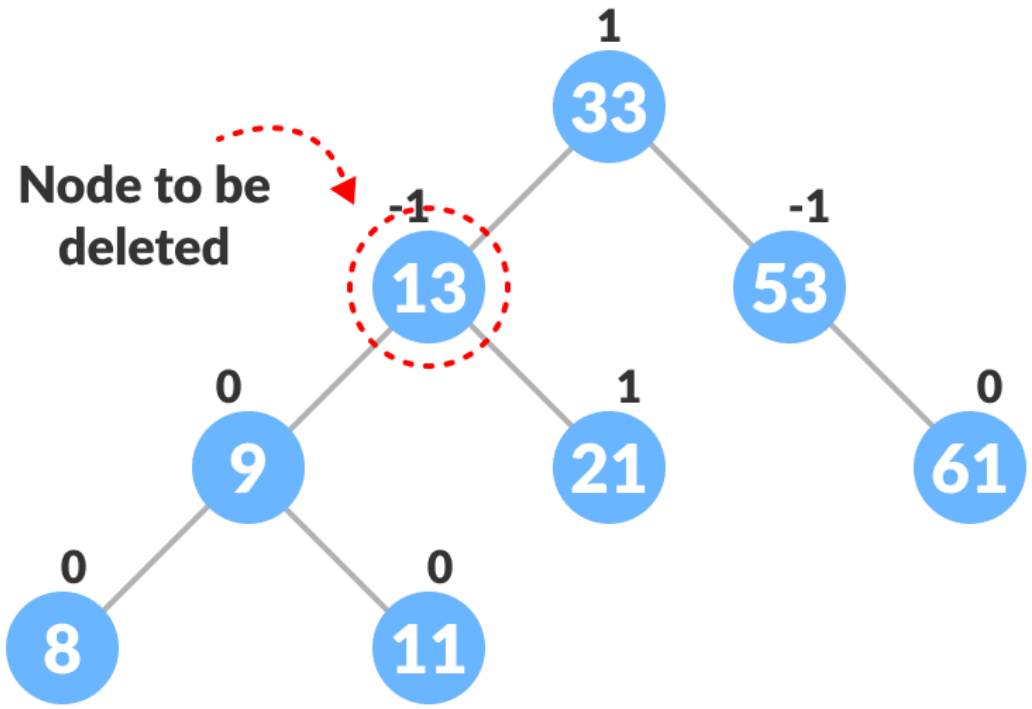
```
Algorithme rd(sommet b, E/S arbre B)
    // rotation droite du sous arbre de racine b
    locales : sommets a,v,pb
              coté = droit/gauche
    debut
       pb <- pere(b,B) //garder infos pour raccrocher le sous arbre dans B
       si pb<>VIDE alors
           si b=fils_gauche(pb,B) alors coté <- gauche
           sinon coté <- droit
           finsi
       finsi
       // === la rotation du sous arbre
       a <- fils_gauche(b,B)
       v <- fils_droit(a,B)
```

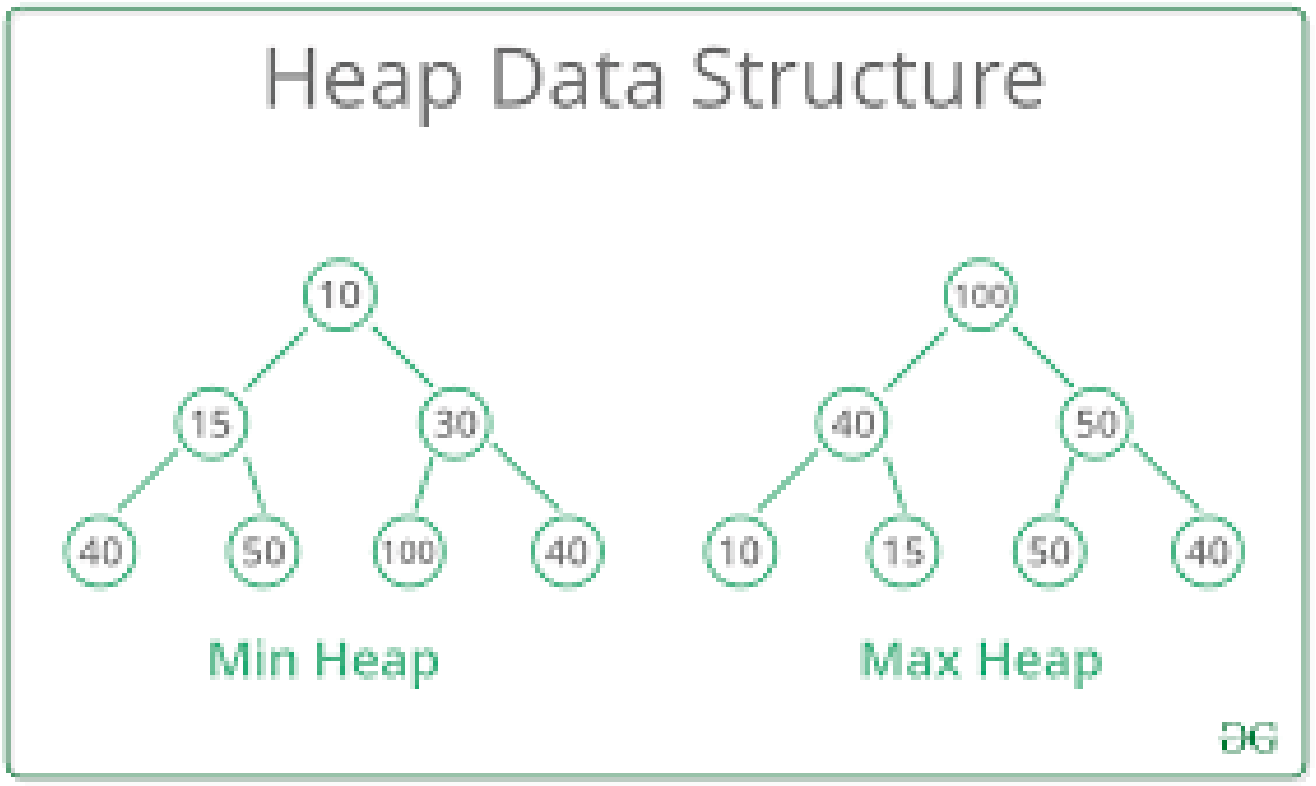
```
Algorithme rd(sommet b, E/S arbre B)
    // suite . . .

        fait_FG(b,v,B)
        fait_FD(a,b,B)
        // maintenant on raccroche dans l'arbre
        si pb<>VIDE alors
            si coté=gauche alors fait_FG(pb,a,B)
            sinon fait_FD(pb,a,B)
            finsi
        sinon
            fait_racine(a,B)
        finsi
    fin
```

La suppression :

La suppression d'un nœud d'un arbre AVL est similaire à celle d'un arbre binaire de recherche. La suppression peut provoquer un déséquilibre, donc rééquilibrer par des rotations.

La suppression dans un arbre AVL peut se faire aussi par rotations successives du nœud à supprimer jusqu'à une feuille.




Heap Data Structure
10
15
30
40
50
100
40
Min Heap
100
40
50
10
15
50
40
Max Heap

Les TAS

Heap

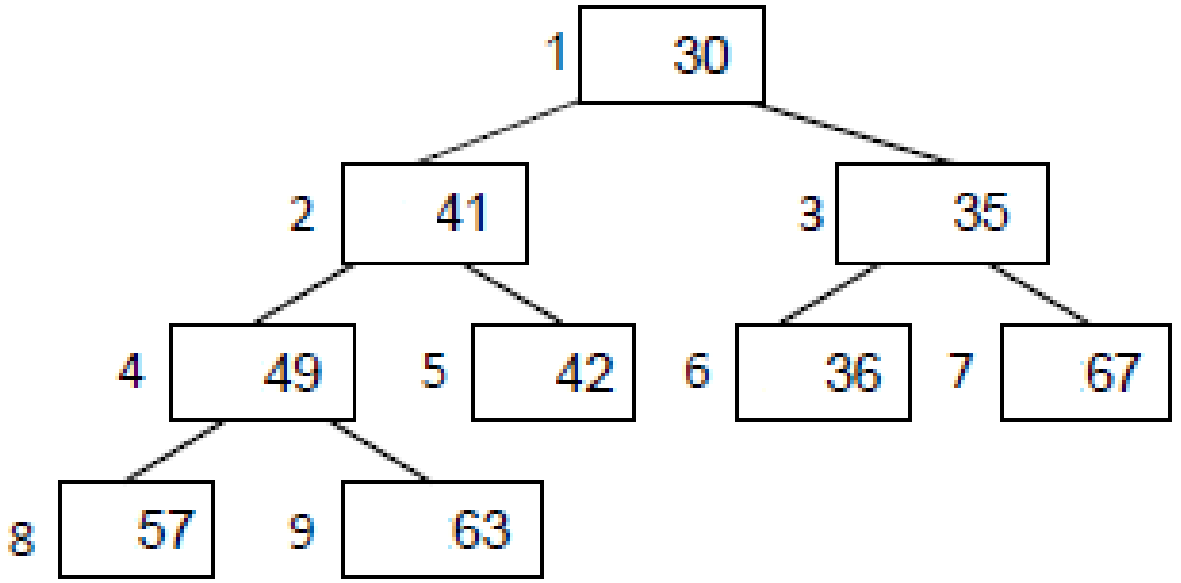
1. Définitions :

Un tas est un arbre binaire quasi-parfait tel que la valeur de chaque sommet est inférieure à la valeur de (son ou) ses fil(s).

Rappel : complet (0 ou 2 fils), parfait (sommet est père 2 sous arbres de même hauteurs), quasi (diff pas plus1).

Exemple :

tas min (il existe tas max)



2. Implémentation d'un arbre binaire (quasi-)parfait

Pour un arbre binaire-quasi parfait, on n'a pas besoin des trois tableaux FG , FD et VAL, un seul suffit !

type TAS = {

tab : tableau[1..MAX] d'éléments

Nb : entier //nb réel d'éléments (de sommets)

}

Le sommet tas.tab[i] :

a pour fils gauche : tas.tab[2*i]

a pour fils droit : tas.tab[2*i+1]

a pour père : tas.tab[i div 2]

est une feuille ssi 2*i > tas.Nb

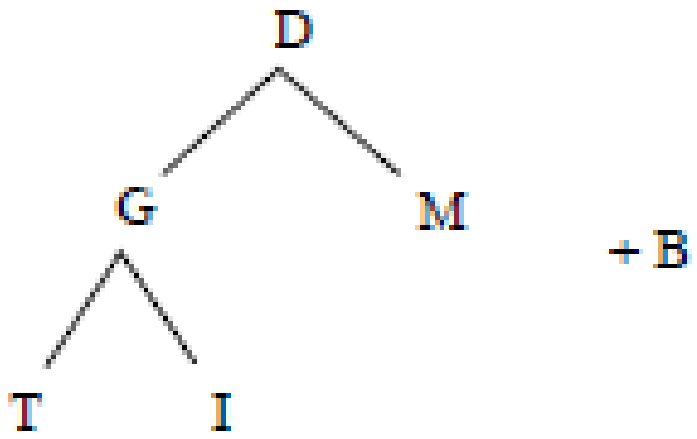
A PARTIR DE 1

Racine : nœud 0
Pour un nœud i,
Parent : $\lfloor (i-1)/2 \rfloor$
Fils gauche : 2i+1
Fils droit : 2i+2

3. Ajout :

On ajoute le sommet en tant que première feuille possible (à la fin du tableau) puis on l'échange avec son père jusqu'à ce que ça forme de nouveau un tas.

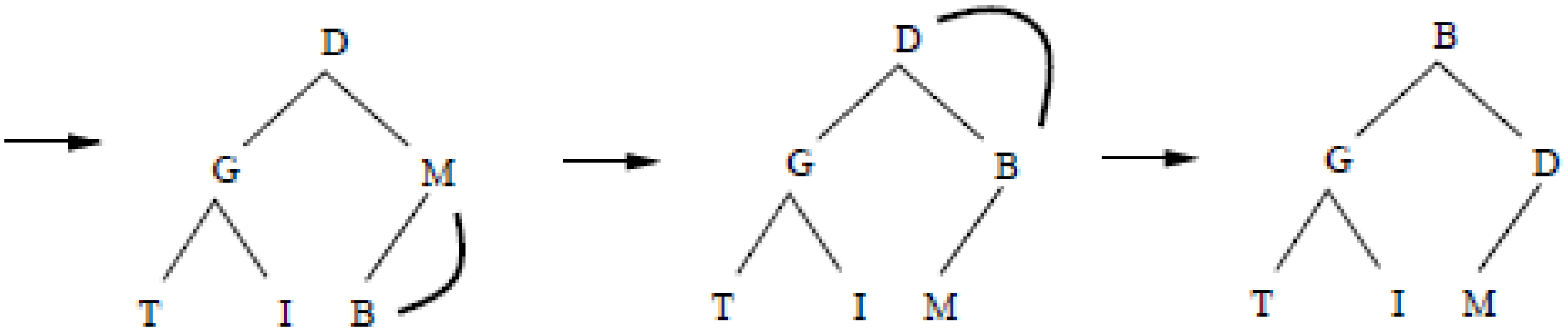
- **Exemple :**



3. Ajout :

On ajoute le sommet en tant que première feuille possible (à la fin du tableau) puis on l'échange avec son père jusqu'à ce que ça forme de nouveau un tas.

- **Exemple :**



3. Ajout : (suite)

```
Algo ajout(E/S TAS tas, element e)

        debut

         tas.Nb <- tas.Nb+1

         tas.tab[tas.Nb] <- e

         Monter(tas,tas.Nb)

       fin
```

3. Ajout : (suite)

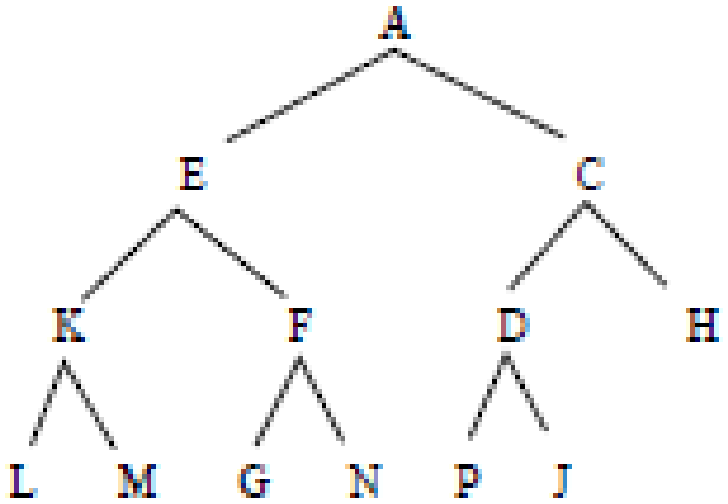
```
Algorithme Monter(E/S TAS tas, entier pos)
// monter dans le tas l'élément en position pos
     locale : entier pere
     debut
        si pos > 1 alors
           pere <- pos div 2
           si tas.tab[pere]>tas.tab[pos]
              permuter(tas.tab,pos,pere)
              Monter(tas,pere)
           finsi
        finsi
     fin
```

4. Epluchage :

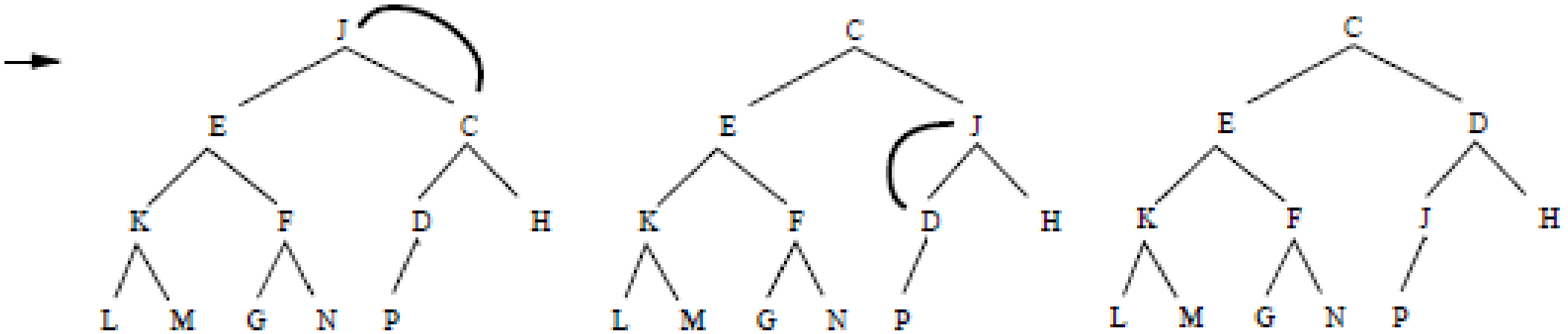
"**Eplucher**" c'est retourner le plus petit élément du tas, le supprimer et faire en sorte que ce qui reste soit toujours un tas.

- On garde la racine (le plus petit) dans une var temporaire.
- on met la dernière feuille à la place de la racine (on la supprime).
- itérer : on fait descendre cette "racine" à sa bonne place en l'échangeant avec le plus petit(plus grand pour tas max) de ses deux fils.
- on retourne la var gardée.

Exemple:



Oter le min



4. Epluchage : Algorithme-1

```
Algo Oter-min(E/S TAS tas):retourne element
     locales : element garde
     debut
       si tas.Nb > 0 alors
          garde <- tas.T[1]
          tas.tab[1] <- tas.tab[tas.Nb]
          tas.Nb <- tas.Nb-1
          Descendre(tas,1)
          retourner garde
       sinon
          erreur "tas vide"
       finsi
     fin
```

4. Epluchage : Algorithme-2

```
Algorithme Descendre(E/S TAS tas, entier pos)
//descendre dans le tas l'element en position pos
     locale : entier fils
     debut
       si pos <=(tas.Nb div 2) alors
           // ce n'est pas une feuille
           fils <- plus-petit-fils(tas,pos)
           si tas.tab[fils]<tas.tab[pos] alors
               permuter(tas.tab,pos,fils)
               Descendre(tas,fils)
           finsi
       finsi
     fin
```

4. Epluchage : Algorithme-3

```
Algo plus-petit-fils(TAS tas,entier pere) : retourne un entier
// retourne la position dans le tas du plus petit fils du sommet père , pere a au moins un fils gauche (n'est pas une feuille) mais peut etre pas de fils droit
     locale : entier fils
     debut
       fils <- 2*père // le fils gauche
       si fils+1<=tas.Nb alors
           // il y a un fils droit
           si tas.tab[fils+1]<tas.tab[fils] alors
                fils <- fils+1
           finsi
       finsi
       retourner fils
     fin
```

4. Tri pas tas (heapsort)

construire un tas contenant les éléments à trier : en ajoutant les éléments un par un.

éplucher le tas :

- écrire la racine (qui est le plus petit élément)
- la supprimer et réorganiser pour que ça reste un tas
- recommencer jusqu'à ce que le tas soit vide.

Exemple : tas max

Trier (9, 2, 11, 7, 4, 14, 3, 16, 8, 10, 15)

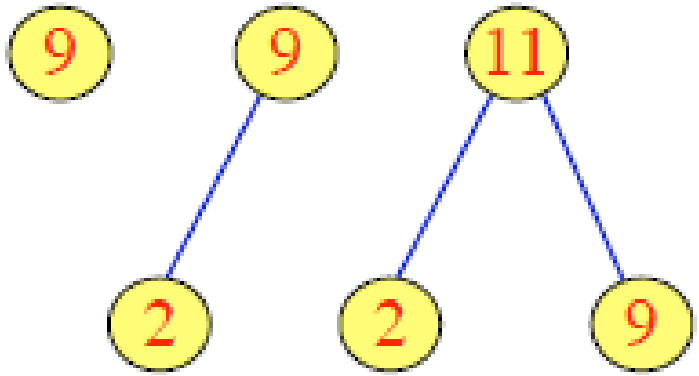
Principe :

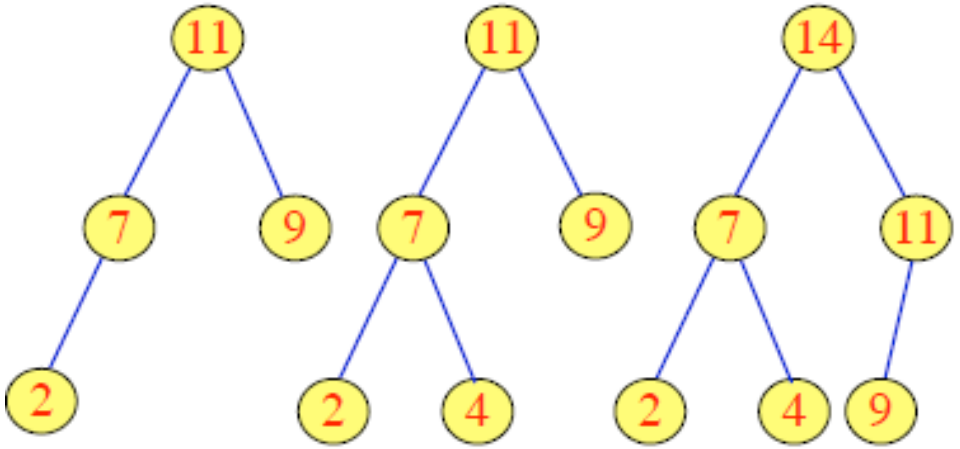
On part d'un tableau vide a. On commence par construire un tas en ajoutant successivement au tas vide les éléments a[0], a[1], ...

On répète ensuite les opérations suivantes :

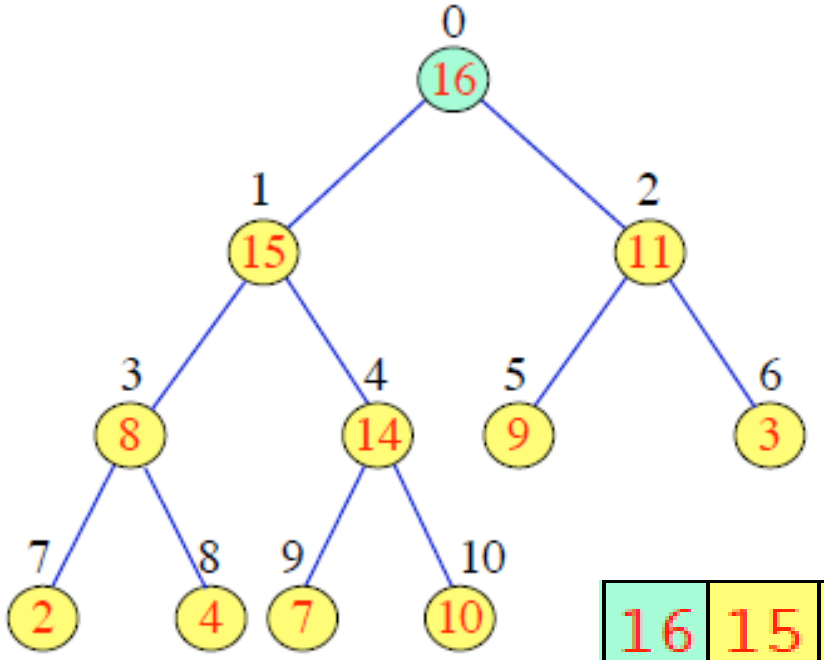
- prendre le maximum,
- le retirer du tas,
- le mettre à droite du tas

1. On ajoute un à un les éléments :



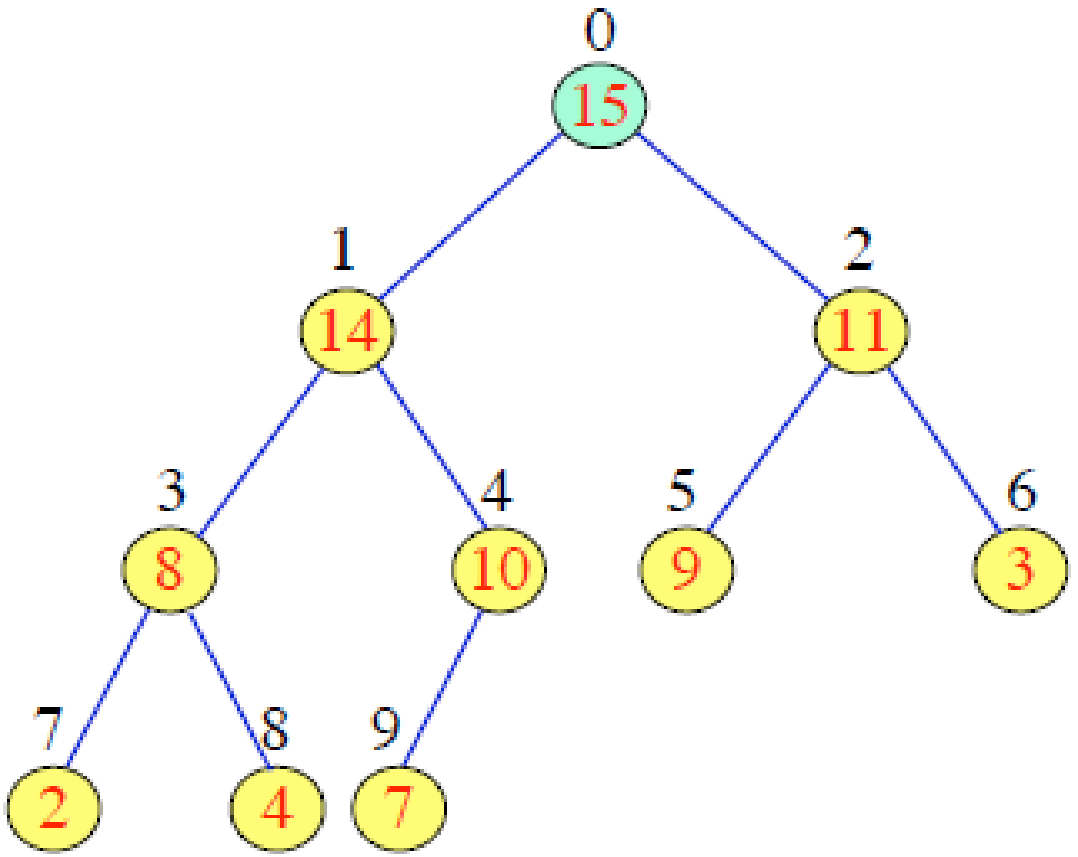


- Une fois le tas obtenu...



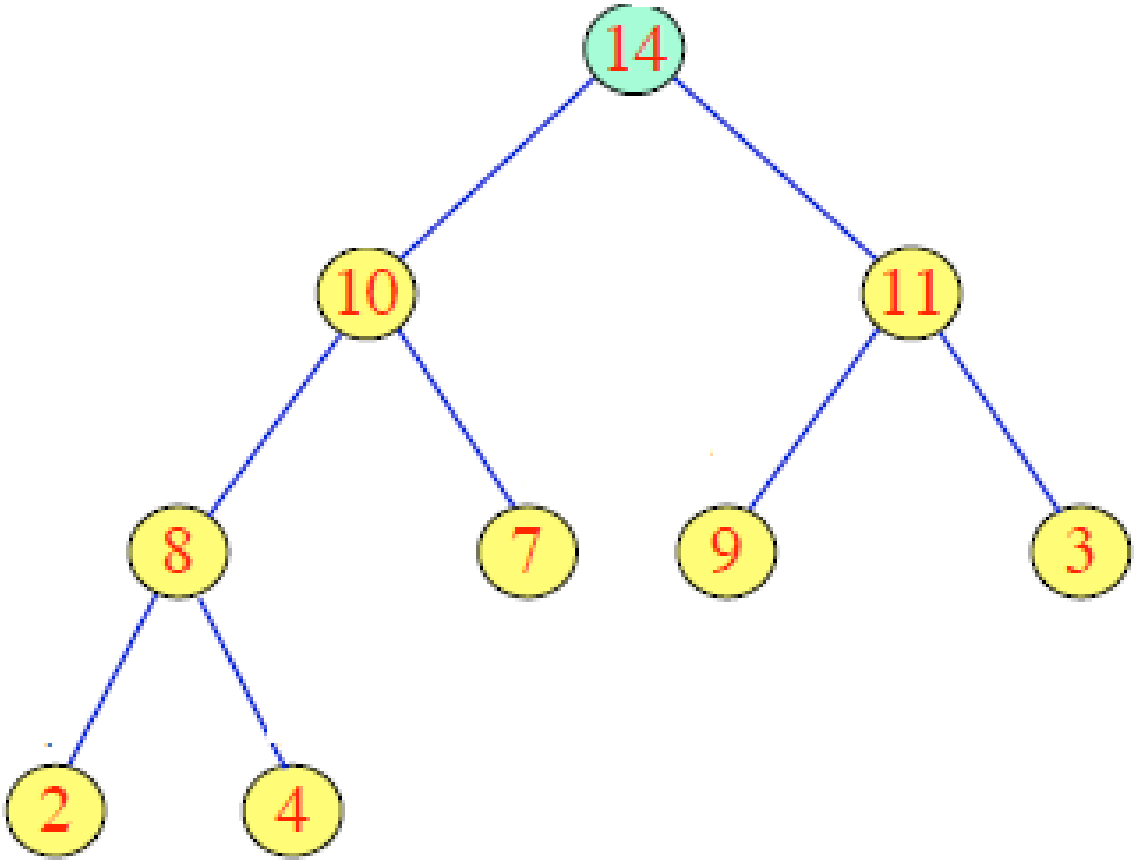
16	15	11	8	14	9	3	2	4	7	10

2. On retire les éléments un à un.



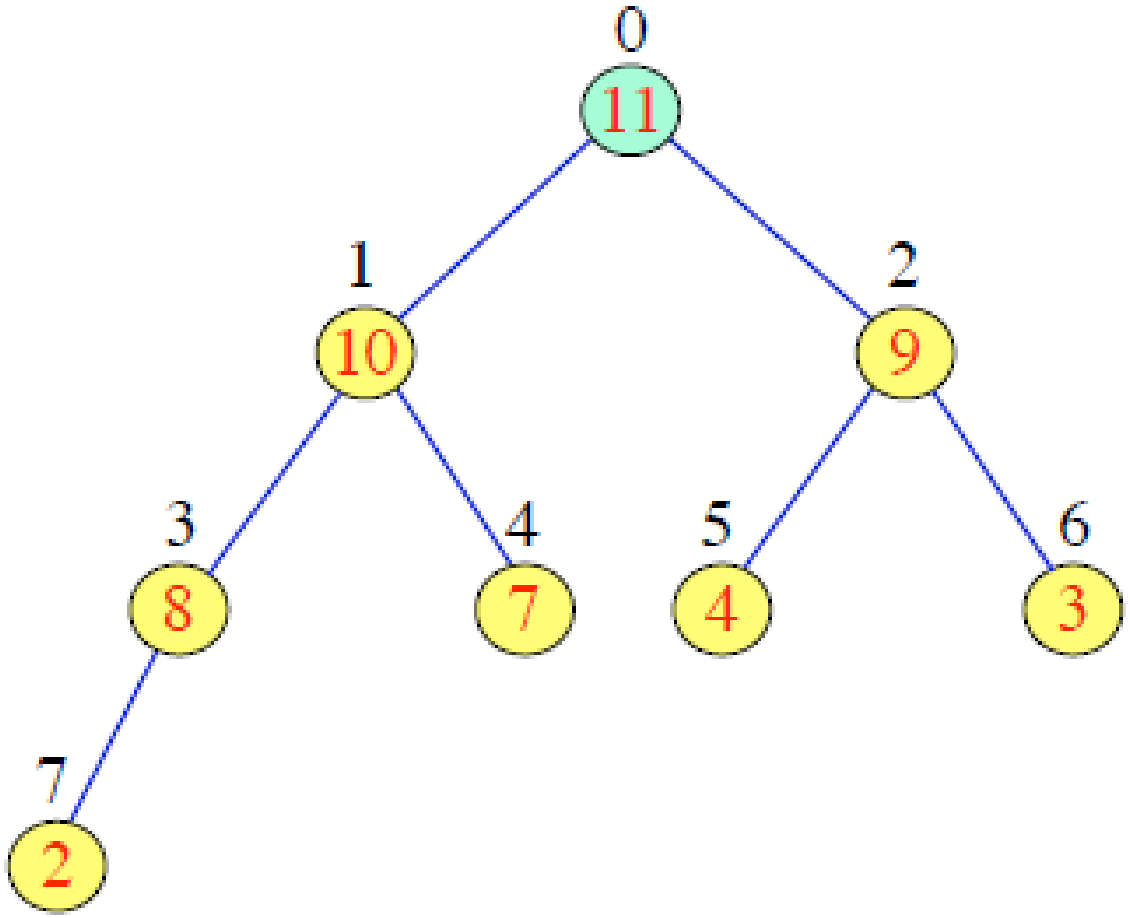
15	14	11	8	10	9	3	2	4	7	16

Ensuite



14	10	11	8	7	9	3	2	4	15	16

Ensuite



• • •

0	1	2	3	4	5	6	7	8	9	10
11	10	9	8	7	4	3	2	14	15	16