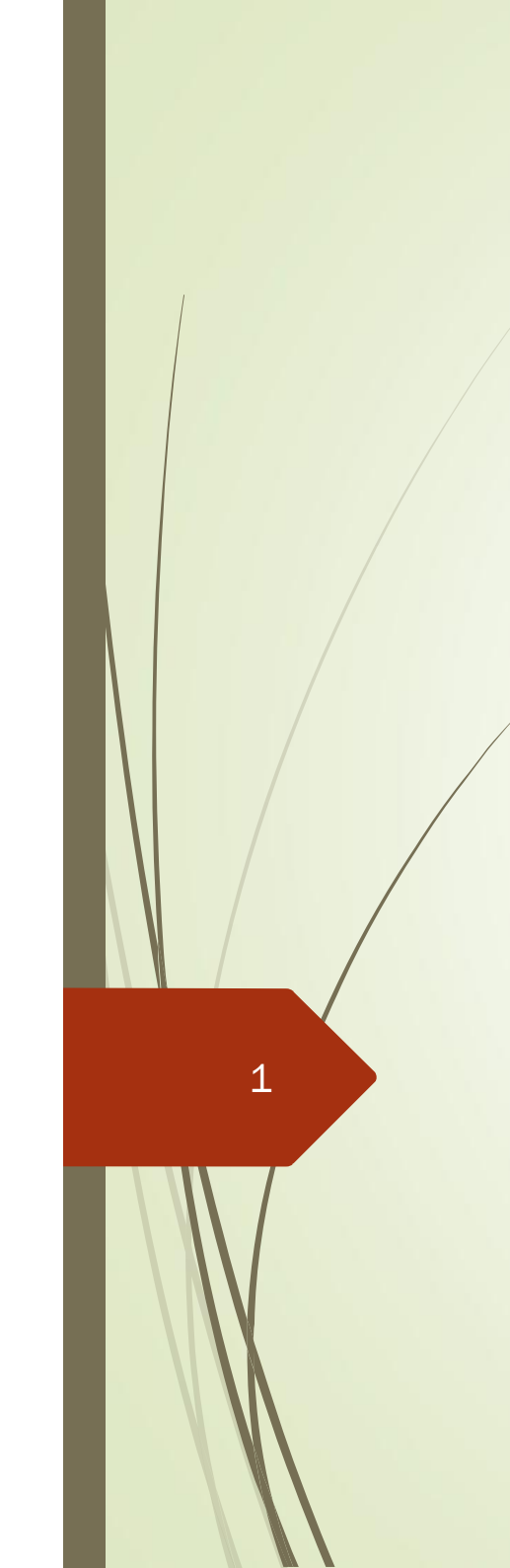




Arbres Binaires de Recherche (Binary Search Trees)

Pr F.Omary

2019-2020

Notion d'Arbre binaire de recherche

- ▶ C'est un arbre binaire particulier :
 - ▶ Permet d'obtenir un algorithme de recherche proche dans l'esprit de la recherche dichotomique ;
 - ▶ Pour lequel les opérations d'ajout et de suppression d'un élément sont aussi efficaces.
- ▶ Cet arbre utilise l'existence d'une relation d'ordre sur les éléments, représentée par une fonction clé, à valeur entière.

Arbre binaire de recherche

Définition

- Un arbre binaire de recherche (*binary search tree* en anglais), en abrégé ABR, est un arbre binaire tel que pour tout nœud :
 - les clés de tous les nœuds du sous-arbre gauche sont inférieures ou égales à la clé du nœud,
 - les clés de tous les nœuds du sous-arbre droit sont supérieures à la clé du nœud.
- Chaque nœud d'un arbre binaire de recherche désigne un élément qui est caractérisé par une clé (prise dans un ensemble totalement ordonné) et des informations associées à cette clé.
- Dans toute illustration d'un arbre binaire de recherche, seules les clés sont représentées. On supposera aussi que toute clé identifie de manière unique un élément.

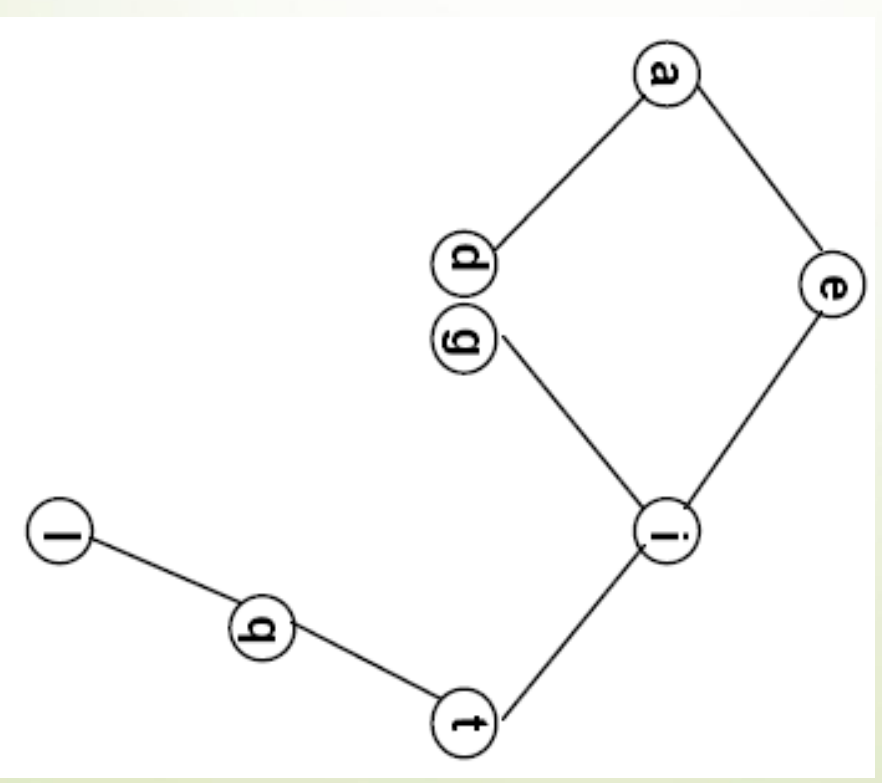
Arbre binaire de recherche

Exemple

► L'arbre de la figure suivante est un arbre binaire de recherche

► Cet arbre représente l'ensemble :

$E = \{a, d, e, g, i, l, q, t\}$
muni de l'ordre alphabétique



Arbre binaire de recherche

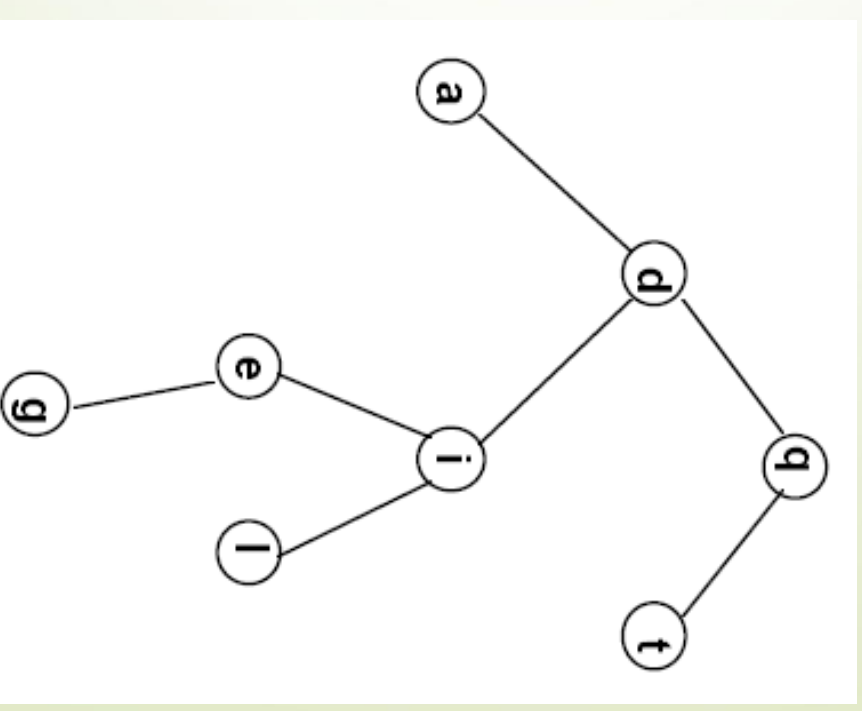
Remarque

- Plusieurs représentations possibles d'un même ensemble par un arbre binaire de recherche
- En effet, la structure précise de l'arbre binaire de recherche est déterminée :
 - par l'algorithme d'insertion utilisé,
 - et par l'ordre d'arrivée des éléments.

Exemple :

- L'arbre binaire de recherche de la figure qui suit représente aussi

$$E = \{a, d, e, g, i, l, q, t\}$$



Opérations sur les arbres binaires de recherche

- ▶ Le type abstrait arbre binaire de recherche, noté `Arbre_Rech`, est décrit de la même manière que le type `Arbre_Binaire`
- ▶ On reprend les opérations de base des arbres binaires, excepté le fait que dans des arbres binaires de recherche, on suppose l'existence de l'opération clé sur le type abstrait `ELelement`
- ▶ On définit, en tenant compte du critère d'ordre, les opérations spécifiques de ce type d'arbre concernant :
 - ▶ la recherche d'un élément dans l'arbre ;
 - ▶ l'insertion d'un élément dans l'arbre ;
 - ▶ la suppression d'un élément de l'arbre.

Recherche d'un élément

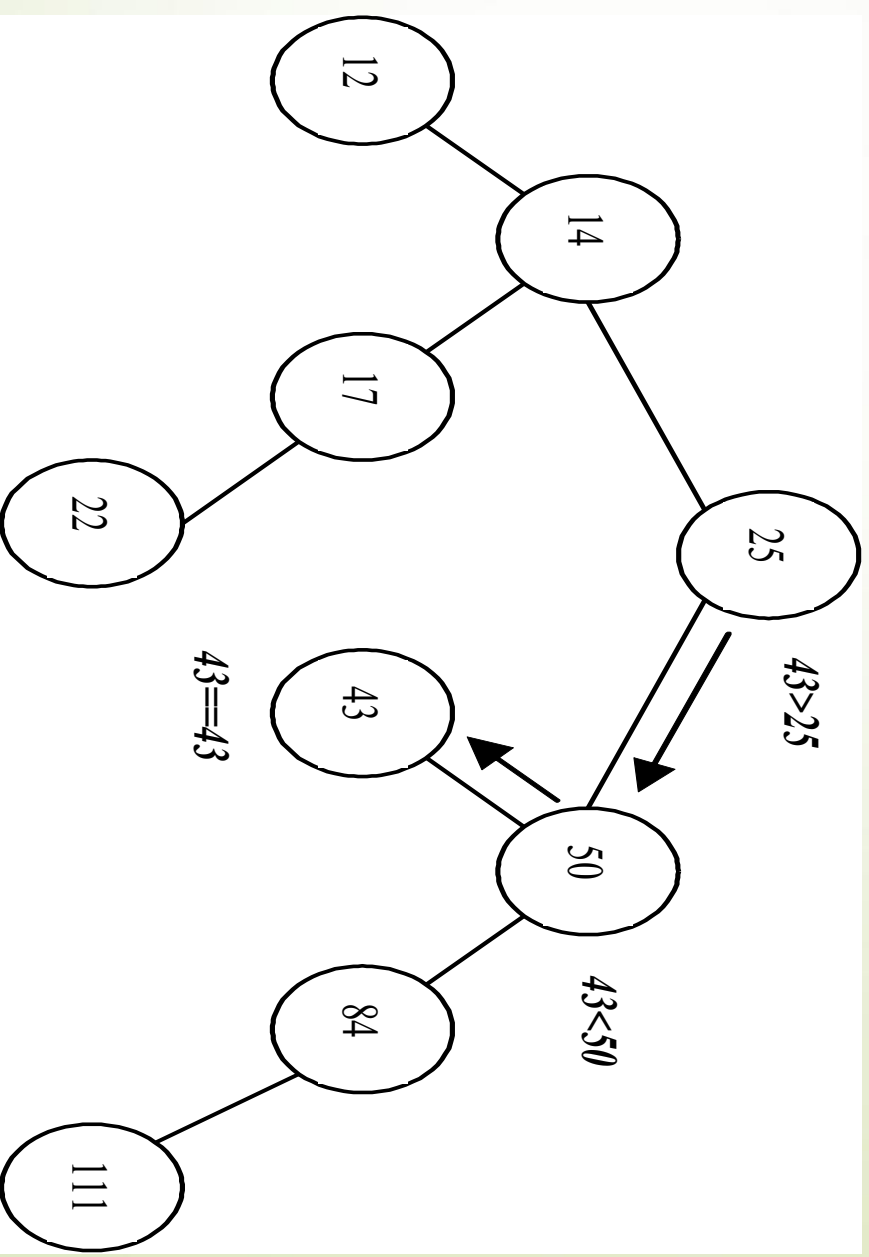
▀ Principe de l'algorithme :

- ▀ On compare la clé de l'élément cherché à la clé de la racine de l'arbre ;
- ▀ Si la clé est supérieure à la clé de la racine, on effectue une recherche dans le fils droit ;
- ▀ Si la clé est inférieure à la clé de la racine, on effectue une recherche dans le fils gauche ;
- ▀ La recherche s'arrête quand on ne peut plus continuer (échec) ou quand la clé de l'élément cherché est égale à la clé de la racine d'un sous arbre (succès).

Recherche d'un élément Exemple

► la figure suivante illustre la recherche de l'élément de clé 43 dans un arbre binaire de recherche.

► Les flèches indiquent le chemin de la recherche



Recherche d'un élément

Spécification

Extension Type Arbre_Rech

Utilise Élément, Booléen

Opérations

Rechercher : Élément x Arbre_Rech → Booléen

Axiomes

Soit, x : Élément, r : Nœud, G, D : Arbre_Rech

Rechercher(x, arbre_vide) = faux

si clé(x) = clé(contenu(r))

alors Rechercher(x, <r, G, D>) = vrai

si clé(x) < clé(contenu(r))

alors Rechercher(x, <r, G, D>) = Rechercher(x, G)

si clé(x) > clé(contenu(r))

alors Rechercher(x, <r, G, D>) = Rechercher(x, D)

Recherche d'un élément

Réalisation en C

```
Boolean Rechercher (Arbre_Rech A, Element e) {  
    if ( est_vide(A) == vrai )  
        return faux; // e n'est pas dans l'arbre  
    else {  
        if ( e == A->val )  
            return vrai; // e est dans l'arbre  
        else if ( e < A->val )  
            // on poursuit la recherche dans le SAG  
            du  
                // noeud courant  
                return Rechercher(A->fg , e);  
        else  
            // on poursuit la recherche dans le SAD  
            du  
                // noeud courant  
                return Rechercher(A->fd , e);  
    }  
}
```

Recherche d'un élément

Autre Spécification

Extension Type Arbre_Rech

Utilise Élément

Opérations

Rechercher : Élément x Arbre_Rech $\rightarrow$ Arbre_Rech

Axiomes

Soit, x : Élément, r : Nœud, G, D : Arbre_Rech

Rechercher(x , arbre_vide) = arbre_vide

si clé(x) = clé(contenu(r))

alors Rechercher(x , $\langle r, G, D \rangle$) = $\langle r, G, D \rangle$

si clé(x) < clé(contenu(r))

alors Rechercher(x , $\langle r, G, D \rangle$) = Rechercher(x , G)

si clé(x) > clé(contenu(r))

alors Rechercher(x , $\langle r, G, D \rangle$) = Rechercher(x , D)

Ajout d'un élément

- ▶ La technique d'ajout spécifiée ici est dite "*ajout en feuille*", car tout nouvel élément se voit placé sur une feuille de l'arbre

- ▶ Le principe est simple :

- ▶ si l'arbre initial est vide, le résultat est formé d'un arbre binaire de recherche réduit à sa racine, celle-ci contenant le nouvel élément ;
- ▶ sinon, l'ajout se fait (*récursivement*) dans le fils gauche ou le fils droit, suivant que l'élément à ajouter est de clé inférieure ou supérieure à celle de la racine.

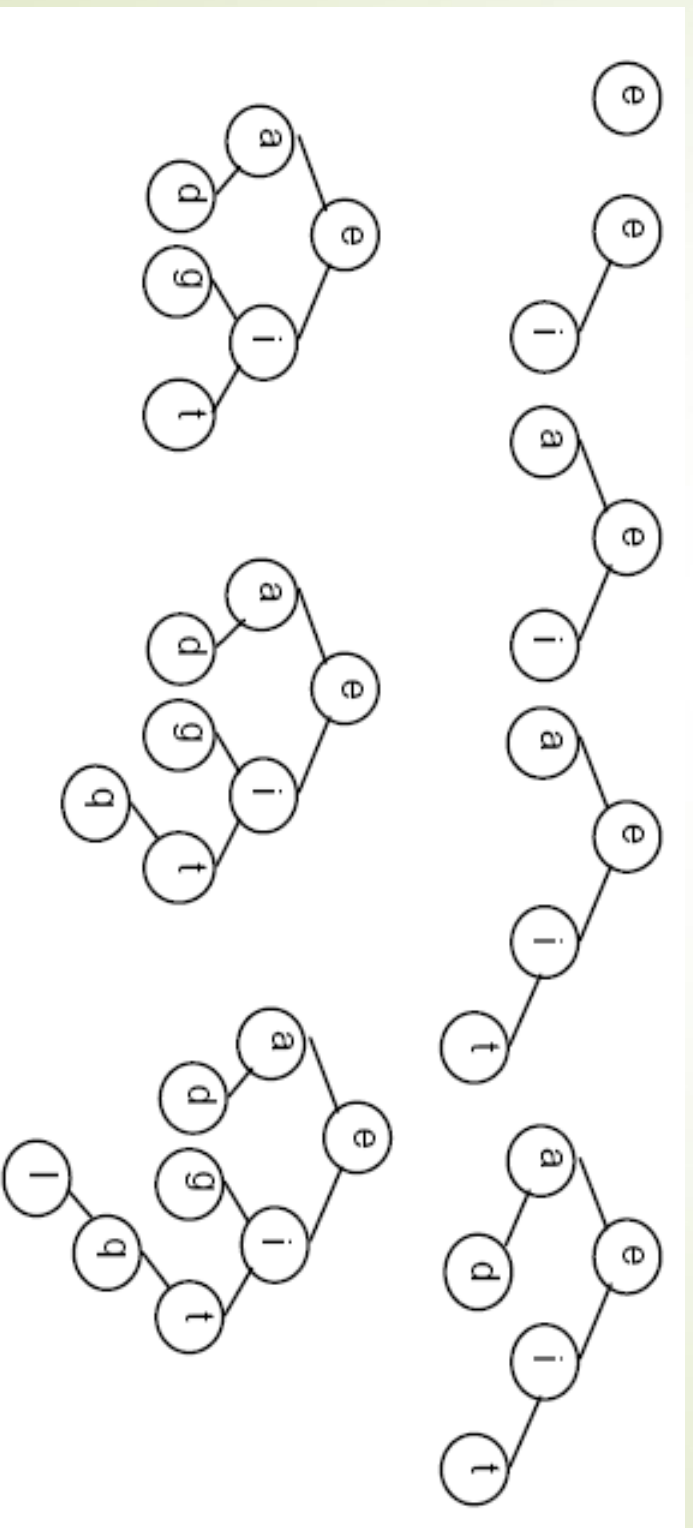
- ▶ Remarque :

- ▶ si l'élément à ajouter est déjà dans l'arbre, l'hypothèse d'unicité des éléments pour certaines applications fait qu'on ne réalise pas l'ajout

Ajout d'un élément

Exemple

- Les figures suivantes illustrent l'ajout successif de e, i, a, t, d, g, q et l dans un arbre binaire de recherche, initialement vide



Ajout "en feuille" d'un élément

Spécification

Extension Type Arbre_Rech

Utilise Élément

Opérations

Ajouter_feuille : Élément $\times$ Arbre_Rech $\rightarrow$ Arbre_Rech

Axiomes

Soit, x : Élément, r : Nœud, G, D : Arbre_Rech

Ajouter_feuille(x , arbre_vide) = $\langle x$, arbre_vide, arbre_vide $\rangle$

si clé(x) $\leq$ clé(contenu(r))

alors

Ajouter_feuille(x , $\langle r, G, D \rangle$) = $\langle r$,
Ajouter_feuille(x , G) , $D \rangle$

sinon

Ajouter_feuille(x , $\langle r, G, D \rangle$) = $\langle r, G$,
Ajouter_feuille(x , D) $\rangle$

Ajout "en feuille" d'un élément

Réalisation

```
fonction Ajouter_feuille(x : Elément, A : Arbre_Rech ) :  
  Arbre_Rech  
  si est_vide(A) alors  
    Pnoeud r = nouveau_noeud(x)  
    si est_vide(r) alors <erreur>  
    retourner cons(r, arbre_vide(), arbre_vide())  
  sinon  
    si x > contenu(racine(A)) alors  
      retourner cons(A, gauche(A), Ajouter_feuille(x, droite(A)))  
    sinon  
      Si x < contenu(racine(A)) alors  
        retourner cons(A, Ajouter_feuille(x, gauche(A)), droite(A))  
      fsi  
    fsi  
  ffonction
```

Ajout "en feuille" d'un élément

Réalisation en C

```
Arbre_Rech Ajouter_feuille(Element x, Arbre_Rech A) {
    if (est_vide(A)) {
        Pnoeud r = nouveau_noeud(x);
        if (r == NULL) {
            printf("Erreur : Pas assez de mémoire !\n");
            exit(-1);
        }
        return cons(r, arbre_vide());
    }
    else
        if (x > contenu(racine(A)))
            return cons(A, gauche(A), Ajouter_feuille(x, droite(A)));
        else
            if (x < contenu(racine(A)) // pas d'ajout lorsque x=contenu(A)
                return cons(A, Ajouter_feuille(x, gauche(A)), droite(A));
            }
}
```


Suppression d'un élément

La suppression est délicate :

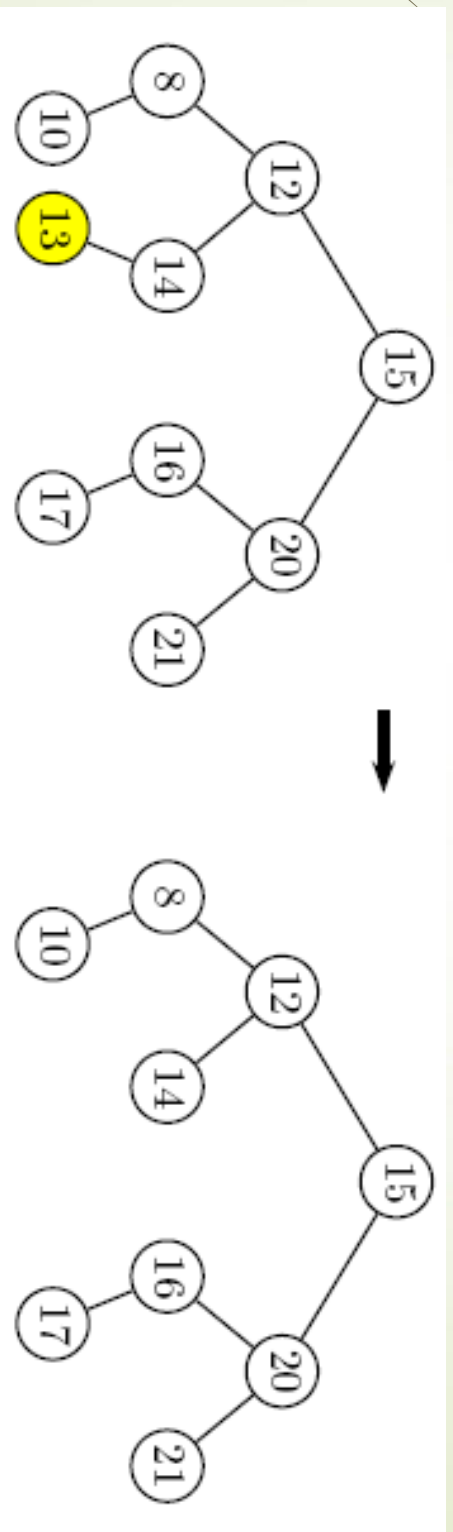
- Il faut réorganiser l'arbre pour qu'il vérifie la propriété d'un arbre binaire de recherche

La suppression commence par la recherche du nœud qui porte l'élément à supprimer. Ensuite, il y a trois cas à considérer, selon le nombre de fils du nœud à supprimer :

- si le nœud est sans fils (*une feuille*), la suppression est immédiate ;
- si le nœud a un seul fils, on le remplace par ce fils ;
- si le nœud a deux fils (*cas général*), on choisit de remplacer ce nœud, soit par le plus grand élément de son sous arbre gauche (*son prédécesseur*), soit par le plus petit élément de son sous arbre droit (*son successeur*).

Suppression d'un élément Exemple 1

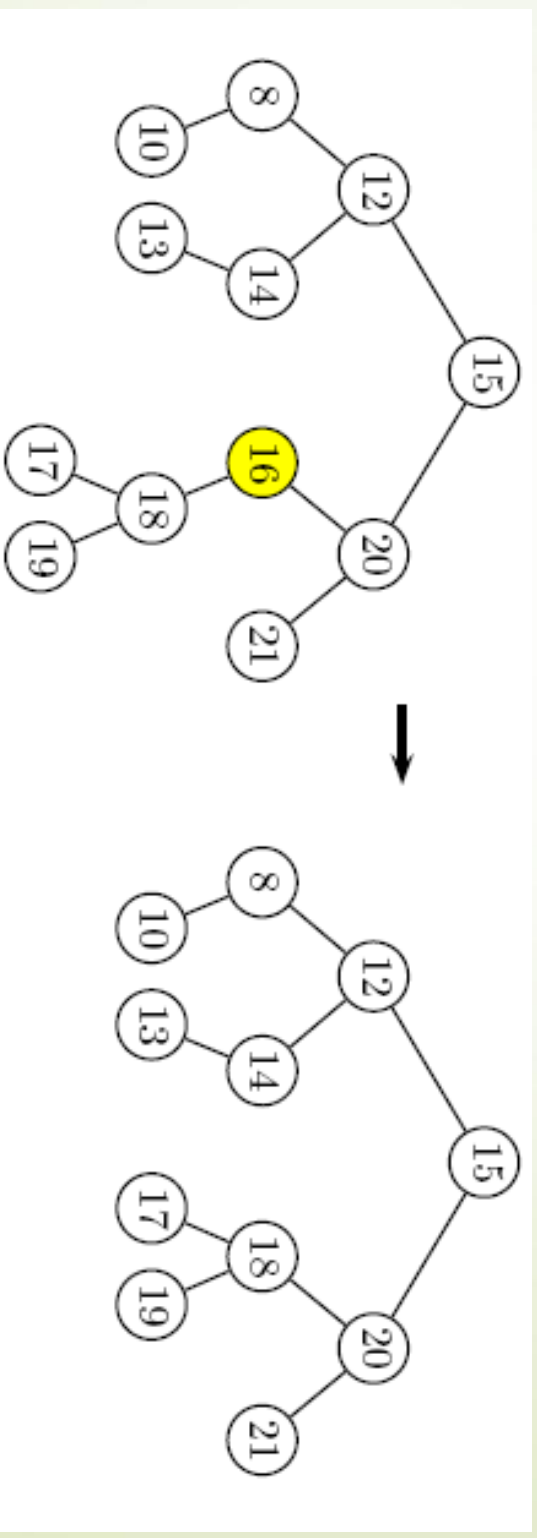
- La figure qui suit illustre la suppression de la feuille qui porte la clé 13



Suppression d'un élément

Exemple 2

- La figure qui suit illustre la suppression du nœud qui porte la clé 16

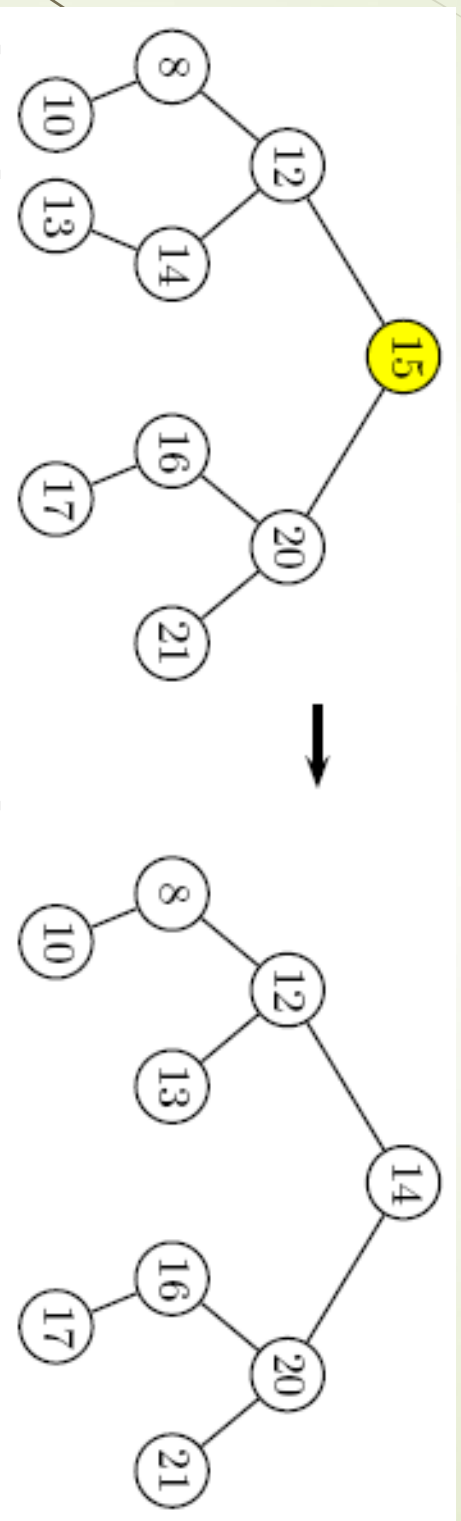


- Ce nœud n'a qu'un seul fils ; le sous arbre de racine portant la clé 18
- Ce sous arbre devient fils gauche du nœud qui porte la clé 20

Suppression d'un élément

Exemple 3

- La figure qui suit illustre le cas d'un nœud à deux fils.



- La clé 15 à supprimer se trouve à la racine de l'arbre. La racine a deux fils ; on choisit de remplacer sa clé par la clé de son prédécesseur.
- Ainsi, la clé 14 est mise à la racine de l'arbre. On est alors ramené à la suppression du nœud du prédécesseur.
- Comme le prédécesseur est le nœud le plus à droite du sous arbre gauche, il n'a pas de fils droit, donc il a zéro ou un fils, et sa suppression est couverte par les deux premiers cas.

Suppression d'un élément

Cas général

- ▶ On choisit ici de remplacer le noeud à supprimer par son prédécesseur (le noeud le plus à droite de son sous arbre gauche)
- ▶ On a besoin de deux opérations supplémentaires :
 - ▶ une opération *Max* qui retourne l'élément de clé maximale dans un arbre binaire de recherche ;
 - ▶ une opération *SupprimerMax* qui retourne l'arbre privé de son plus grand élément.

Suppression d'un élément: Spécification

Extension Type Arbre_Rech

Utilise Élément

Opérations

Max : Arbre_Rech → Élément
 SupprimerMax : Arbre_Rech → Arbre_Rech
 Supprimer : Élément x Arbre_Rech → Arbre_Rech

Pré-conditions

Max(A) est_défini_ssi est_vide(A) = faux
 SupprimerMax(A) est_défini_ssi est_vide(A) = faux

Axiomes

Soit, x : Élément, r : Nœud, G, D : Arbre_Rech
 si est_vide(D) = vrai alors Max(<r, G, D>) = r
 sinon Max(<r, G, D>) = Max(D)
 si est_vide(D) = vrai alors SupprimerMax(<r, G, D>) = G
 sinon SupprimerMax(<r, G, D>) = <r, G, SupprimerMax(D)>
 Supprimer(x, arbre_vide) = arbre_vide
 si clé(x) = clé(contenu(r)) et est_vide(D) = vrai
 alors Supprimer(x, <r, G, D>) = G
 sinon si clé(x) = clé(contenu(r)) et est_vide(G) = vrai
 alors Supprimer(x, <r, arbre_vide, D>) = D
 sinon si clé(x) = clé(contenu(r))
 alors Supprimer(x, <r, G, D>) = <Max(G), SupprimerMax(G), D>
 si clé(x) < clé(contenu(r))
 alors Supprimer(x, <r, G, D>) = <r, Supprimer(x, G), D>
 si clé(x) > clé(contenu(r))
 alors Supprimer(x, <r, G, D>) = <r, G, Supprimer(x, D)>

Suppression d'un élément

Réalisation

```
fonction Max(A : Arbre_Rech) : Pnoeud
(* A doit être non vide ! *)
si est_vide(droite(A))
alors retourner A
sinon retourner Max(droite(A))
fsi
ffonction

fonction SupprimerMax(A : Arbre_Rech) : Arbre_Rech
(* A doit être non vide ! *)
si est_vide(droite(A))
alors
retourner gauche(A)
sinon
retourner cons(A, gauche(A), SupprimerMax(droite(A)))
fsi
ffonction
```

Cette fonction retourne un pointeur sur le nœud contenant la plus grand élément d'un arbre binaire de recherche

Cette fonction supprime le plus grand élément d'un arbre binaire de recherche

Suppression d'un élément

Réalisation (suite)

```
fonction Supprimer(x : Elément, A : Arbre_Rech) : Arbre_Rech
si est_vide(A) alors retourner A    (* ou <erreur> *)
sinon
  si x > contenu(racine(A)) alors
    retourner cons(A, gauche(A), Supprimer(x, droite(A)))
  sinon
    si x < contenu(racine(A)) alors
      retourner cons(A, Supprimer(x, gauche(A)), droite(A))
    sinon // x= contenu (racine(A))
      si est_vide(droite(A)) alors retourner gauche(A)
      sinon
        si est_vide(gauche(A)) alors retourner droite(A)
        sinon // ni droite (A) est vide ni gauche(A)
          retourner cons(Max(gauche(A)), SupprimerMax(gauche(A)), droite(A))
        fsi
      fsi
    fsi
  fsi
fsi
ffonction
```


Arbre Binaire de Recherche

Complexité des Opérations

- On montre que, les opérations de recherche, insertion et suppression dans un arbre binaire de recherche contenant n éléments sont :
 - en moyenne en $O(\log_2(n))$;
 - dans le pire des cas en $O(h)$;où h désigne la hauteur de l'arbre
- Si l'arbre est dégénéré, sa hauteur étant $n-1$, ces trois opérations sont en $O(n)$
- Si l'arbre est équilibré, les opérations sont en $O(\log_2(n))$ (*d'où leur intérêt...*)

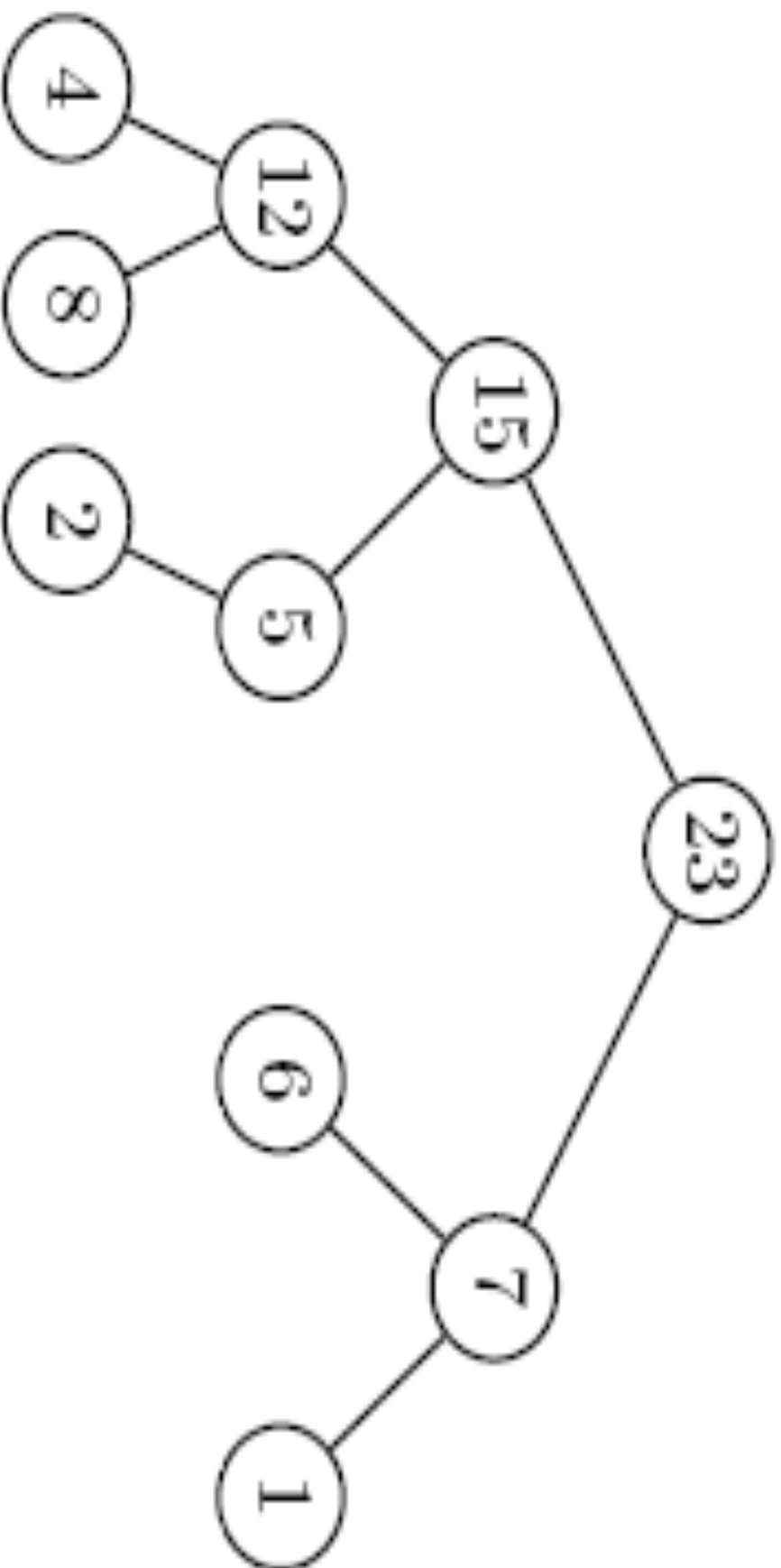
Arbores Maximiers ou Tas (Heaps)

Notion d'Arbre Maximier (ou Tas)

- Appelé aussi monceau (*Heap en anglais*)
- C'est un arbre binaire parfait tel que la clé de chaque noeud est supérieure ou égale aux clés de tous ses fils
- L'élément maximum de l'arbre se trouve donc à la racine
- Rappel :
 - Pour un arbre binaire parfait, tous les niveaux sont entièrement remplis sauf éventuellement le dernier et dans ce cas les feuilles du dernier niveau sont regroupées le plus à gauche possible
- Un tas est un arbre binaire partiellement ordonné :
 - Les noeuds sur chaque branche sont ordonnés sur celle-ci ;
 - Ceux d'un même niveau ne le sont pas nécessairement.
- Un tas dans lequel chaque noeud enfant a une clé inférieure (resp. supérieure) ou égale à la clé de son père est appelé arbre maximier (max heap) (resp. arbre minimier (min heap))

Arbre Maximier (ou Tas)

Exemple



Type Abstrait Tas

Type Tas

Utilise Booléen, Élément

Opérations

```
tas_vide      : → Tas
est_vide      : Tas → Booléen
max           : Tas → Élément
ajouter : Tas x Élément → Tas
supprimerMax : Tas → Tas
appartient    : Tas x Élément → Booléen
```

Préconditions

```
max(T) est_défini_ssi est_vide(T) = faux
supprimerMax(T) est_défini_ssi est_vide(T) = faux
ajouter(T,e) est_défini_ssi appartient(T,e) = faux
```

Axiomes

Soit, $T, T1 : \text{Tas}, e : \text{Elément}$

si $\text{est_vide}(T) = \text{vrai}$ alors $\text{appartient}(T,e) = \text{faux}$

$\text{appartient}(T, \text{max}(T)) = \text{vrai}$

si $\text{appartient}(T,e) = \text{vrai}$ alors $\text{max}(T) \geq e$

Opérations sur un Tas

- ▀ **tas_vide : → Tas**
 - ▀ Opération d'initialisation; crée un tas vide
- ▀ **est_vide : Tas → Booléen**
 - ▀ Vérifie si un tas est vide ou non
- ▀ **max : Tas → Élément**
 - ▀ Retourne le plus grand élément d'un tas
- ▀ **ajouter : Tas x Élément → Tas**
 - ▀ Ajoute un élément dans un tas
- ▀ **supprimerMax : Tas → Tas**
 - ▀ Supprime le plus grand élément d'un tas
- ▀ **appartient : Tas x Élément → Booléen**
 - ▀ Vérifie si un élément appartient ou non à un tas

Représentation d'un Tas

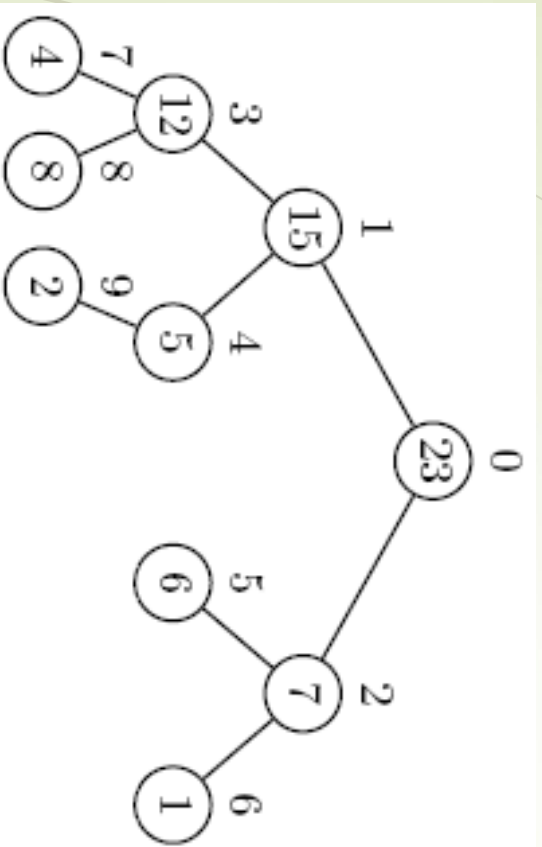
- ▶ Il existe une représentation compacte pour les arbres binaires parfaits, et donc pour les tas :
- ▶ La représentation par tableau, basée sur la numérotation des nœuds niveau par niveau et de gauche à droite
- ▶ Les numéros d'un nœud sont donc les indices dans un tableau. En outre, ce tableau s'organise de la façon suivante :
- ▶ le nœud racine a pour indice 0 ;
- ▶ soit le nœud d'indice i dans le tableau, son fils gauche a pour indice $2i + 1$, et son fils droit a pour indice $2(i+1)$;
- ▶ si un nœud a un indice $i \neq 0$, alors son père a pour indice $\lfloor (i-1)/2 \rfloor$

- ▶ On déduit de cette organisation, où n désigne le nombre d'éléments du tas, que :

- ▶ un nœud d'indice i est une feuille si $2i+1 \geq n$
- ▶ un nœud d'indice i a un fils droit si $2(i+1) < n$

Représentation d'un Tas

Exemple



0	1	2	3	4	5	6	7	8	9
23	15	7	12	5	6	1	4	8	2

Un tas avec sa numérotation hiérarchique

Représentation du tas par un tableau

Représentation en C d'un Tas

```
#define MAX_ELEMENTS 200 // taille
                             maximum du tas
typedef int Element // un élément est
                    un int
typedef struct {
    int taille; // nombre d'éléments dans le
                tas
    Element tableau[MAX]; // les éléments
                        du tas
} Tas;
```

Opérations sur un Tas

- ▶ **Trois opérations fondamentales :**
 - ▶ *Ajout d'un élément ;*
 - ▶ *Suppression du maximum ;*
 - ▶ *Recherche du maximum.*

Opération d'Ajout

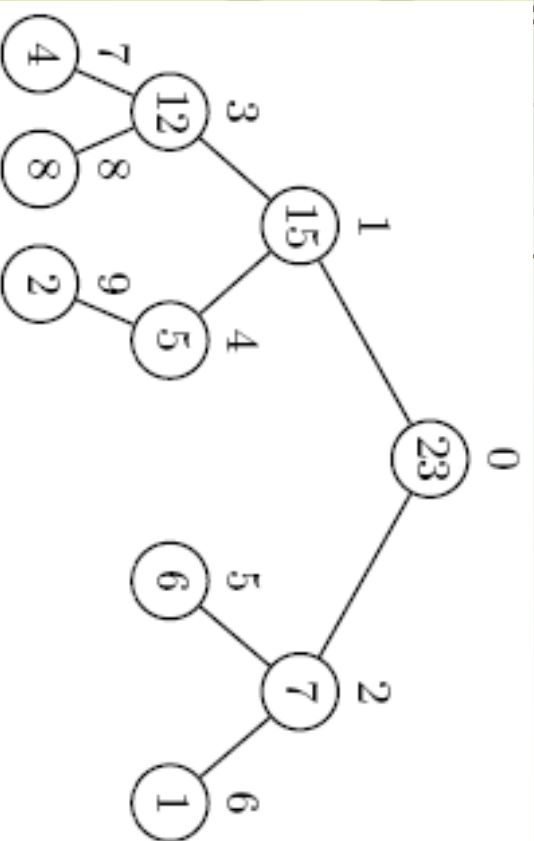
► *Principe :*

- Créer un nouveau nœud contenant la clé du nouvel élément ;
- Insérer cette clé le plus à gauche possible sur le dernier niveau du tas (ou si le dernier niveau est plein, à l'extrême gauche d'un nouveau niveau). La nouvelle clé est insérée dans la première case non utilisée du tableau ;
- Faire "remonter cette nouvelle clé" à sa place en la permutant avec la clé de son père, tant qu'elle est plus grande que celle de son père.

Opération d'Ajout

Exemple (1)

- Supposons qu'on veuille insérer la valeur 21 dans le tas représenté ci-dessous :
- On place la valeur 21 juste à droite de la dernière feuille,
- c'est-à-dire dans la case d'indice 10 dans le tableau.

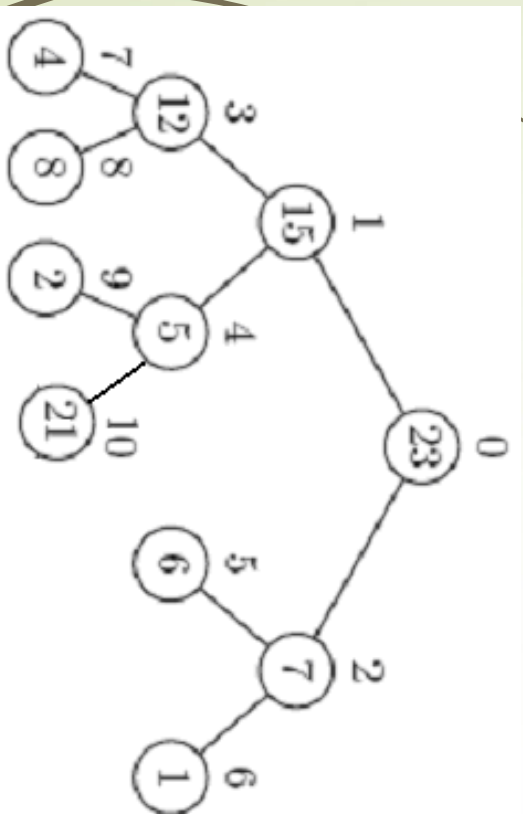


0	1	2	3	4	5	6	7	8	9
23	15	7	12	5	6	1	4	8	2

Opération d'Ajout

Exemple (2)

- On compare 21, la nouvelle donnée insérée, avec la donnée contenue dans le nœud père, autrement dit on compare la donnée de la case d'indice 10 du tableau avec la donnée de la case d'indice = 4.
- Puisque 21 est plus grand que 5, on les échange.



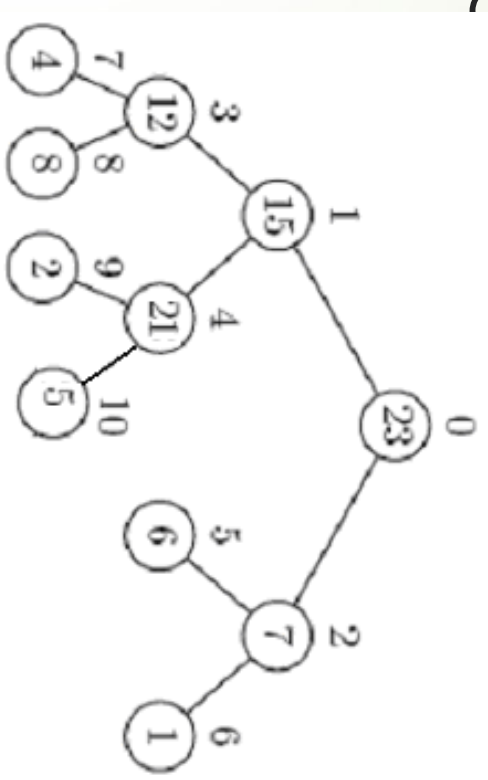
0	1	2	3	4	5	6	7	8	9	10
23	15	7	12	5	6	1	4	8	2	21

Opération d'Ajout

Exemple (3)

► Le nouvel arbre binaire obtenu n'est pas un tas :

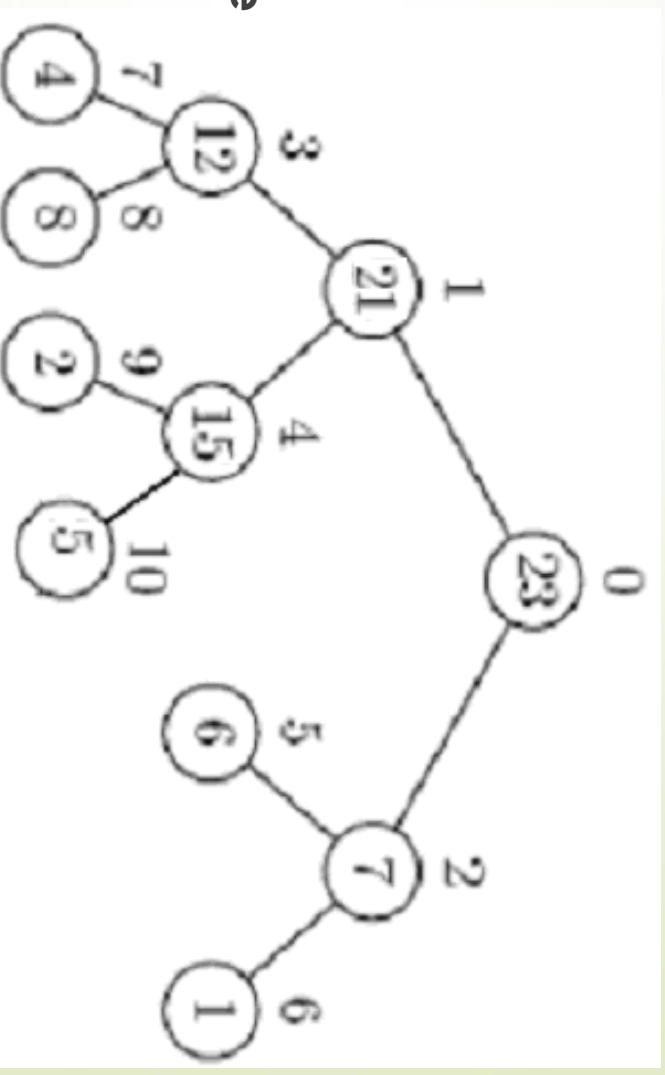
- La valeur 21 du nœud d'indice 4 est plus grande que la valeur 15 de son nœud père (d'indice = 1)
- Echanger les contenus des nœuds d'indices respectifs 1 et 4



0	1	2	3	4	5	6	7	8	9	10
23	15	7	12	21	6	1	4	8	2	5

Opération d'Ajout Exemple (4)

- Puisque 21 est plus petit que 23 :
- L'opération d'ajout est terminée
- On a bien obtenu un tas.



0	1	2	3	4	5	6	7	8	9	10
23	21	7	12	15	6	1	4	8	2	5

Opération d'Ajout

Pseudo-code

```
fonction ajouter(Tas t, Elément e) : Tas
début
     $i \leftarrow t.taille$ 
     $t.taille \leftarrow i+1$ 
     $t.tableau[i] \leftarrow e$ 
    tant que  $((i > 0)$  et
        faire {
             $(t.tableau[i] > t.tableau[(i-1) \text{ div } 2])$ 
            échanger( $t.tableau[i]$ ,  $t.tableau[(i-1) \text{ div } 2]$ )
             $i \leftarrow (i-1) \text{ div } 2$ 
        }
    retourner (t)
fin
```


Opération d'Ajout

La complexité de l'opération d'ajout est en $O(h)$, où h est la hauteur du tas :

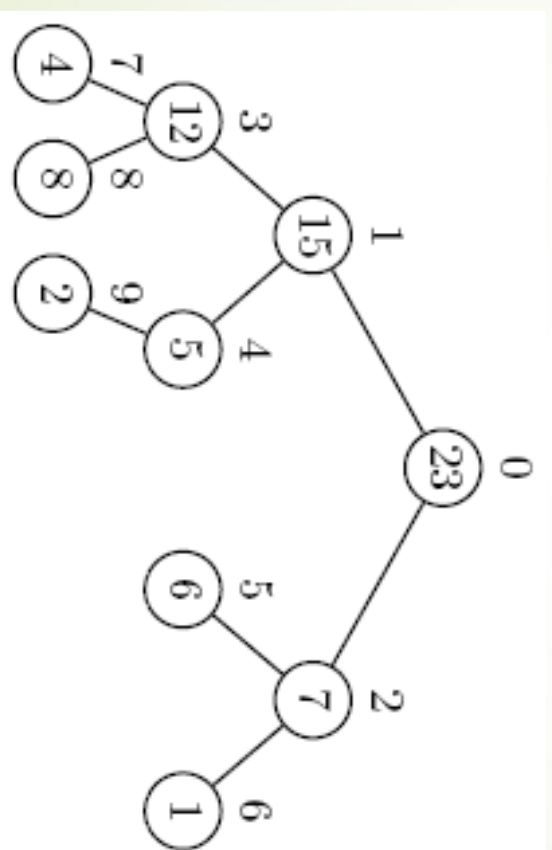
- On ne fait que remonter un chemin ascendant d'une feuille vers la racine (en s'arrêtant éventuellement avant).
- La hauteur d'un tas de taille n est précisément égale à $\lfloor \log_2(n) \rfloor$ et donc l'ajout demande un temps $O(\log(n))$.

Opération de Suppression du Maximum

- ▶ Principe :
- ▶ Remplacer la clé du nœud racine par la clé du nœud situé le plus à droite du dernier niveau du tas. Ce dernier nœud est alors supprimé ;
- ▶ Réorganiser l'arbre, pour qu'il respecte la définition du tas, en faisant descendre la clé de l'élément de la racine à sa bonne place en permutant avec le plus grand des fils.

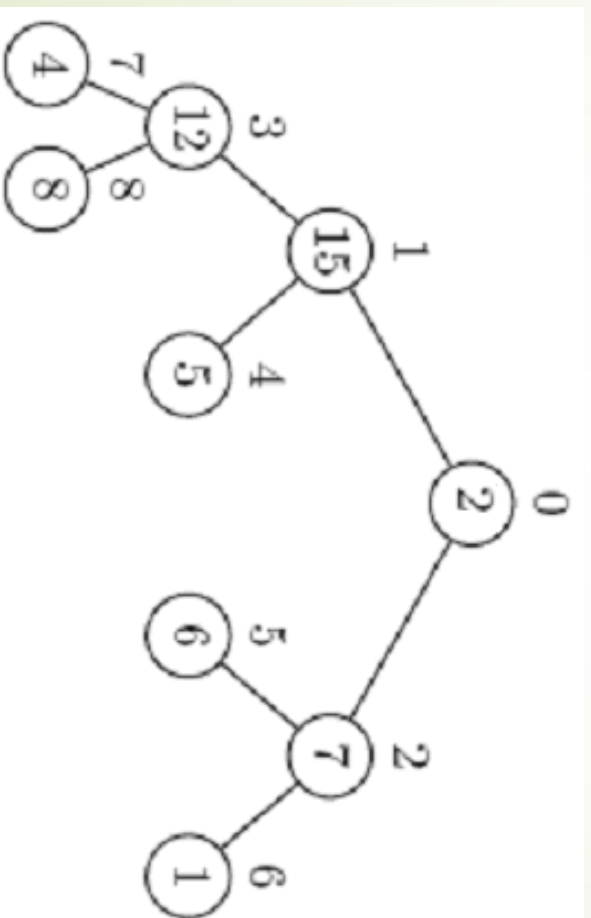
Opération de Suppression du Maximum (Exemple) (1)

- Supposons qu'on désire supprimer la valeur 23 contenue dans la racine du tas illustré par la figure suivante :



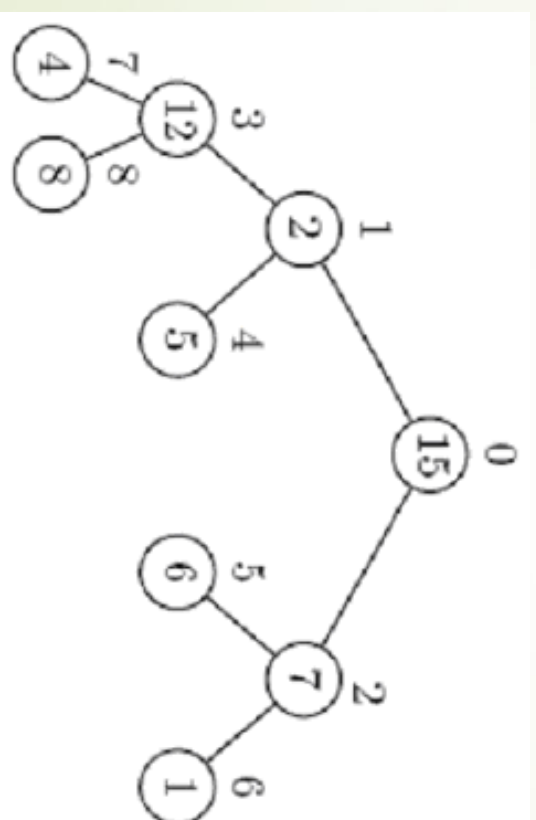
Opération de Suppression du Maximum (Exemple) (2)

- **On commence alors par remplacer le contenu du nœud racine par celui du dernier nœud du tas :**
- **Ce dernier nœud est alors supprimé ;**
- **Ceci est illustré par la figure suivante :**



Opération de Suppression du Maximum (Exemple) (3)

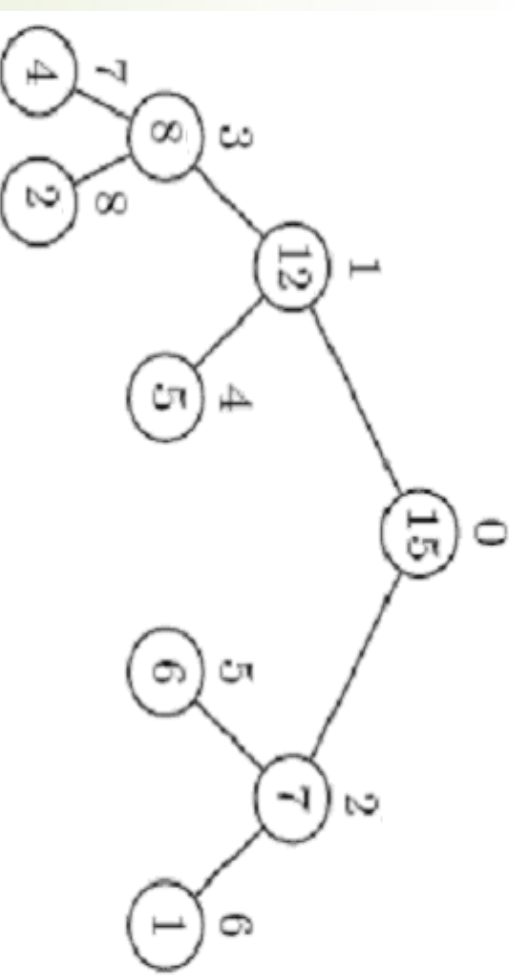
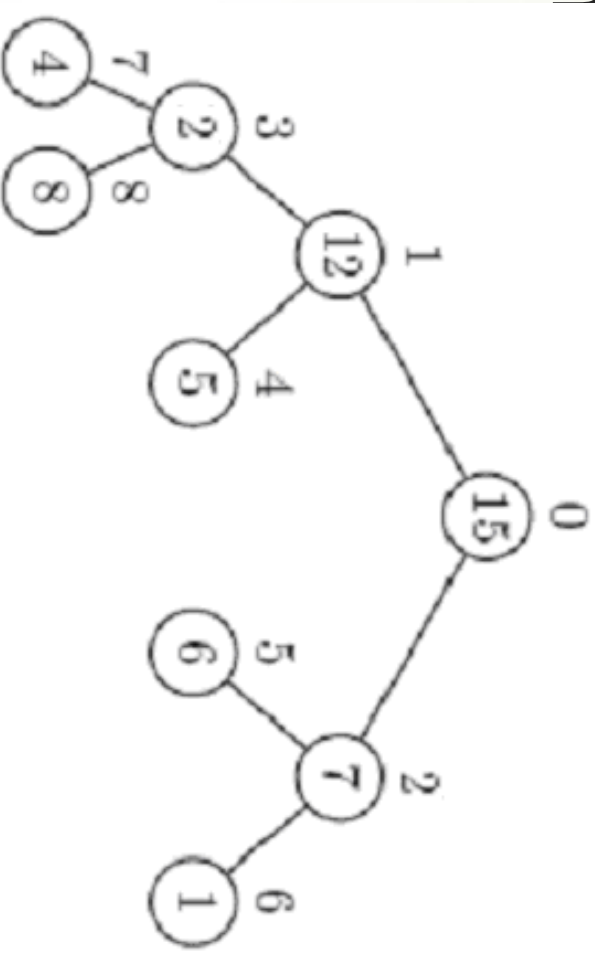
- ▶ **L'arbre obtenu est parfait mais n'est pas un tas :**
- ▶ **la clé contenue dans la racine a une valeur plus petite que les valeurs des clés de ses fils ;**
- ▶ **Cette clé de valeur 2 est alors échangée avec la plus grande clé de ses fils, à savoir 15 ;**
- ▶ **L'arbre obtenu est représenté par la figure suivante :**



Opération de Suppression du Maximum

Encore une fois, cet arbre n'est pas un tas. On le réorganise pour qu'il respecte la définition du tas

Le dernier arbre obtenu est bien un tas ; il est illustré par la figure suivante :



Opération de Suppression du Maximum (Pseudo-code) (1)

- ▶ Une version qui utilise la procédure *Entasser*
- ▶ La procédure *Entasser* :
 - ▶ permet de faire descendre la valeur en $t[i]$ de manière que l'arbre de racine en i devienne un tas ;
 - ▶ suppose que les sous arbres de racines en $2i+1$ (fils gauche du nœud en i) et en $2i+2$ (fils droit du nœud en i) sont des tas.

Opération de Suppression du Maximum (Pseudo-code) (2)

```
procédure Entasser(Tableau  $t[0 \dots n-1]$ , Entier  $i$ )  
début  
  si  $((2i+2 == n)$  ou  $(t[2i+1] \geq t[2i+2]))$  alors  
     $k \leftarrow 2i+1$   
  sinon  
     $k \leftarrow 2i+2$   
  fsi  
  si  $t[i] < t[k]$  alors  
    échanger ( $t[i]$ ,  $t[k]$ )  
    si  $k \leq ((n \text{ div } 2) - 1)$  alors  
      Entasser( $t$ ,  $k$ )  
    fsi  
  fsi  
fin
```


Opération de Suppression du Maximum (Pseudo-code) (3)

```
fonction supprimerMax (t : tas) : tas
(* le tas t est supposé non vide !! *)

début
  t.taille ← t.taille - 1
  t.tableau[0] ← t.tableau[t.taille]
  Entasser(t, 0)
retourner (t)
fin
```

Opération de Suppression du Maximum (Complexité)

- ▶ La complexité de la suppression est la même que celle de l'insertion, c-à-d $O(\log(n))$:
- ▶ *En effet, on ne fait que suivre un chemin descendant depuis la racine.*

Opération de Recherche du Maximum

(Pseudo-code & Complexité)

- ▀ L'opération de recherche du maximum est immédiate dans les tas
- ▀ Elle prend un temps constant $O(1)$

```
fonction Max (t : tas) : Elément
(* Le tas t est supposé non vide !! *)
début
  retourner (t.tabl[0])
fin
```

Exemples d'Applications des Tas

- ▶ **Files de priorités (Priority queues) :**
 - ▶ Les tas sont fréquemment utilisés pour implémenter des files de priorités.
 - ▶ A l'opposé des files standard, une file de priorités détruit l'élément de plus haute (ou plus basse) priorité.
 - ▶ La signification de la "priorité" d'un élément dépend de l'application
 - ▶ A tout instant, on peut insérer un élément de priorité arbitraire dans une file de priorités. Si l'application souhaite la destruction de l'élément de plus haute priorité, on utilise un arbre maximier.
- ▶ **Tri par tas (Heapsort) :**
 - ▶ Les opérations sur les tas permettent de résoudre un problème de tri à l'aide d'un algorithme appelé tri par tas (heapsort).
 - ▶ Cet algorithme a la même complexité temporelle, $O(n \log(n))$, que le tri rapide (quicksort). Mais, en pratique, une bonne implémentation de ce dernier le bat d'un petit facteur constant.

Algorithme du Tri par Tas (Principe)

- Supposons qu'on veut trier, en ordre croissant, un tableau T de n éléments.
- Principe :
 - L'algorithme du tri par tas commence, en utilisant la fonction *ConstruireTas*, par construire un tas dans le tableau à trier T ;
 - Ensuite, il prend l'élément maximal du tas, qui se trouve en $T[0]$, l'échange avec $T[n-1]$, et rétablit la propriété de tas, en utilisant l'appel de fonction *Entasser*($T, 0$) pour le nouveau tableau à $n-1$ éléments (*la case $T[n-1]$ n'est pas considérée*) ;
 - L'algorithme de tri par tas répète ce processus pour le tas de taille $n-1$ jusqu'à la taille 2.

Algorithme du Tri par Tas (Pseudo-code) (1)

```
fonction Tri_par_Tas(Tableau T[0 .. n-1]) :  
  Tableau  
  début  
    T ← ConstruireTas(T)  
    pour i ← (n-1) à 1 par pas -1 faire  
      Echanger(T[0], T[i])  
      n ← n-1  
    Entasser (T, i)  
  retourner (T)  
fin
```

*ConstruireTas produit un tas
à partir d'un tableau T*

*Entasser sert à garantir le maintien de la
propriété de tas pour l'arbre de racine en i*

Algorithme du Tri par Tas (Pseudo-code) (2)

```
fonction ConstruireTas(Tableau  $T[0 \dots n-1]$ ) : Tas  
début  
  pour  $i \leftarrow ((n \text{ div } 2) - 1)$  à 0 par pas -1 faire  
    Entasser ( $T, i$ )  
  retourner ( $T$ )  
fin
```

*Les feuilles sont
des tas à un
élément !*

ConstruireTas

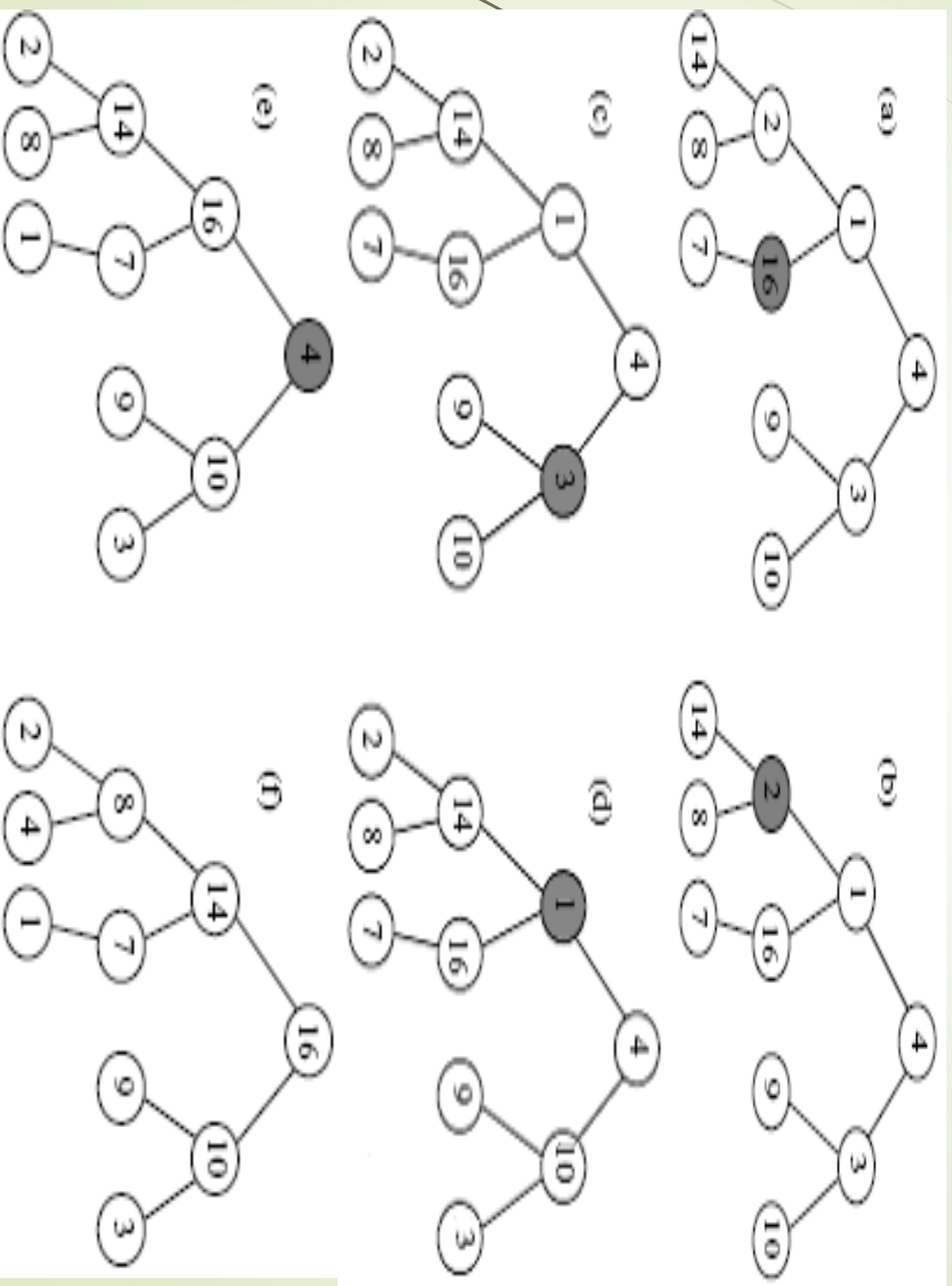
Exemple (1)

- Illustration de l'action ConstruireTas sur un tableau d'entiers contenant 10 éléments

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

- Remarquer que les nœuds qui portent les valeurs 9, 10, 14, 8 et 7 sont biens des feuilles, et donc des tas à un élément.

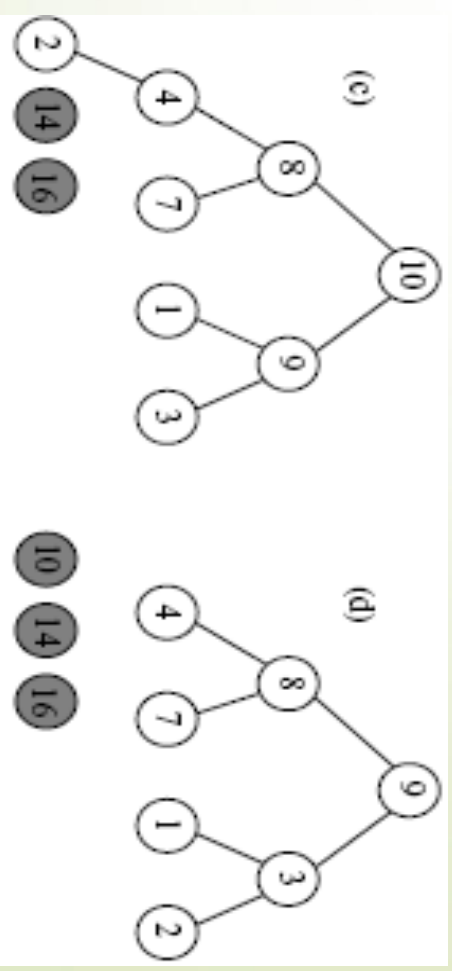
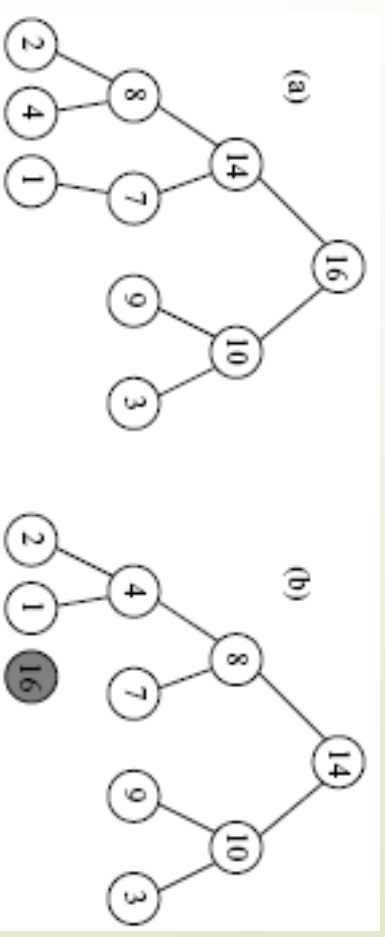
Construire Tas: Exemple (2)



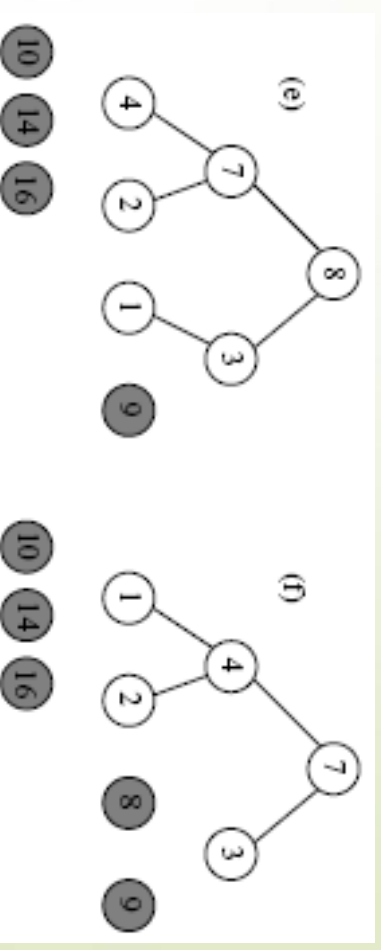
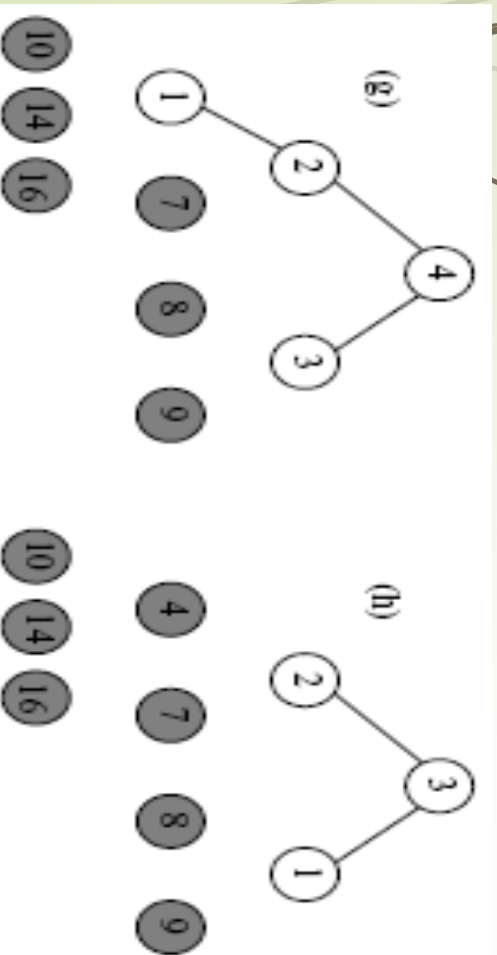
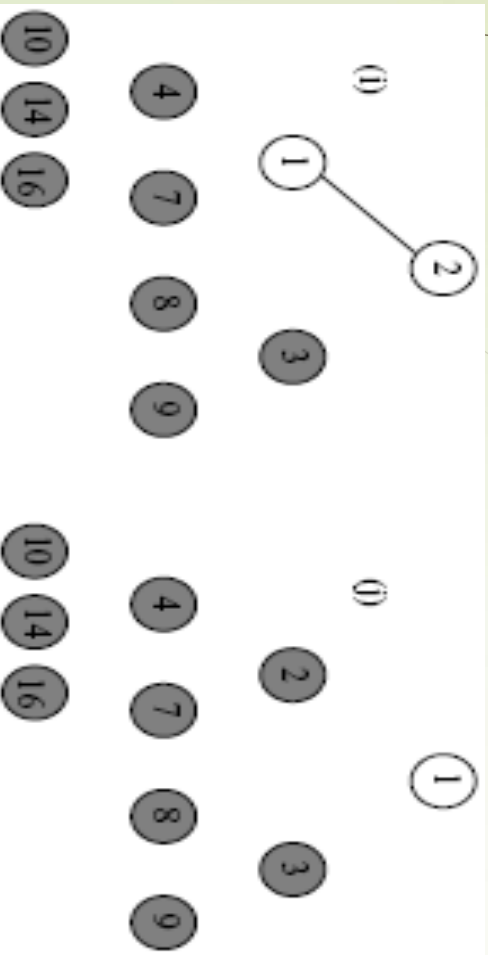
Tri par Tas : Exemple (1)

► Les figures qui suivent illustrent l'action du tri par tas après construction du tas

► Chaque tas est montré au début d'une itération de la boucle



Tri par Tas : Exemple (2)



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

Tableau final : trié

Algorithme du Tri par Tas

Complexité

- ▶ On montre que l'appel à `ConstruireTas` prend un temps $O(n)$
- ▶ Chacun des $(n-1)$ appels à *Entasser* prend un temps $O(\log(n))$
- ▶ Par conséquent, l'algorithme du tri par tas s'exécute en $O(n \log(n))$

Introduction aux Arbres de Recherche Équilibrés

▀ *(Balanced Search Trees)*

Notion d'Arbres de Recherche Équilibrés

- La définition des arbres équilibrés impose que la différence entre les hauteurs des fils gauche et des fils droit de tout noeud ne peut excéder 1
- Il faut donc maintenir l'équilibre de tous les noeuds au fur et à mesure des opérations d'insertion ou de suppression d'un noeud
- Quand il peut y avoir un déséquilibre trop important entre les deux fils d'un noeud, il faut recréer un équilibre par :
 - des rotations d'arbres ou par éclatement de noeuds (cas des arbres B)
- Les algorithmes de rééquilibrage sont très compliqués :
 - On cite entre autres, quelques exemples d'arbres équilibrés pour les quels les opérations de recherche, d'insertion et de suppression sont en $O(\log(n))$

Arbres de Recherche Équilibrés

Exemples (1)

► Les arbres AVL :

- Introduits par Adelson-Velskii Landis Landis (d'où le nom d'AVL) dans les années 60 ;
- Un arbre AVL est un arbre binaire de recherche stockant une information supplémentaire pour chaque nœud : son facteur d'équilibre ;
- Le facteur d'équilibre représente la différence des hauteurs entre son sous arbre gauche et son sous arbre droit ;
- Au fur et à mesure que des nœuds sont insérés ou supprimés, un arbre AVL s'ajuste de lui-même pour que tous ses facteurs d'équilibres restent à 0, -1 ou 1.

Arbres de Recherche Équilibrés

Exemples (2)

- *Les arbres rouges et noirs :*
 - *Des arbres binaires de recherche qui se maintiennent eux-mêmes approximativement équilibrés en colorant chaque nœud en rouge ou noir ;*
 - *En contrôlant cette information de couleur dans chaque nœud, on garantit qu'aucun chemin ne peut être deux fois plus long qu'un autre, de sorte que l'arbre reste équilibré.*

Arbres de Recherche Équilibrés

Exemples (3)

▀ **Les B arbres :**

- ▀ Arbres de recherche équilibrés qui sont conçus pour être efficaces sur d'énormes masses de données stockées sur mémoires secondaires ;
- ▀ Chaque nœud permet de stocker plusieurs clés ;
- ▀ Généralement, la taille d'un nœud est optimisée pour coïncider avec la taille d'un bloc (ou page) du périphérique, en vue d'économiser les coûteux accès d'entrées sorties.

