Cours 5 : Arbres

1. Définitions et algorithmes de base

1.1. **Arbres.** La structure d'arbre est présente de façon récurrente en algorithmique et plus généralement en informatique : **non linéaire** mais **sans boucle**, elle permet de modéliser de nombreux problèmes ou données.

Exemples : Arbre généalogique, structure de fichier.

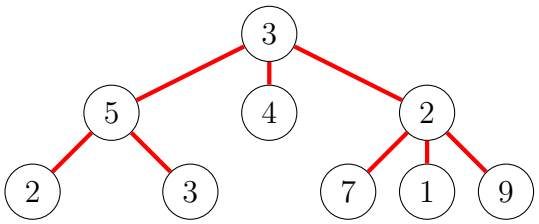
Nous allons donner ici une définition récursive des **arbres enracinés**.

Définition 1.1. *Un arbre est un* nœud *qui possède une liste (éventuellement vide) de fils qui sont eux même des arbres.*

Ici, un *nœud* est simplement une unité de mémoire qui contient de l'information. On considère qu'on ne peut pas *réutiliser* les nœuds précédents.

Exemple 1.2.

Un arbre dont les nœuds contiennent des entiers.



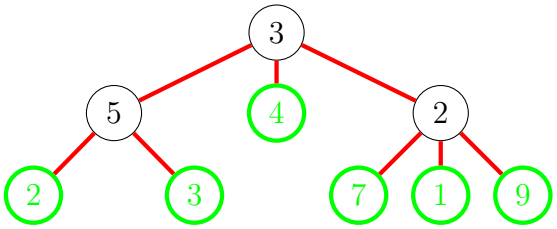
Remarque 1.3. *En théorie des graphes, on définit un arbre comme un* graphe acyclique connexe. *Notre définition revient à choisir une* racine *dans le graphe.*

Définition 1.4.

- *La **racine** de l'arbre est le nœud initial.*
- *Une **feuille** est un nœud dont la liste de fils est vide.*
- *Un **nœud interne** est un nœud dont la liste de fils n'est pas vide.*
- *La **taille** d'un arbre est son nombre de nœuds.*
- *La **hauteur** d'un arbre est le nombre maximal de nœuds qui sépare la racine d'une feuille.*

Exemple 1.5.

Dans l'exemple précédent, la valeur de la racine est 3, elle a trois fils dont les racine ont pour valeurs : 5, 4 et 2. Voici un dessin de l'arbre où toutes les feuilles ont été notées en gras et vert.

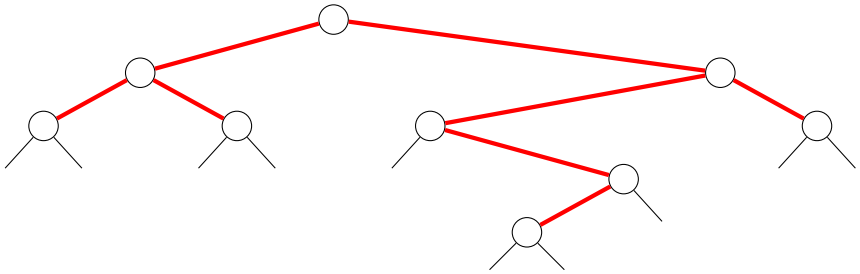


La taille est de 9 et la hauteur est de 3.

1.2. **Arbres binaires.** Les arbres binaires sont une classe d'arbres très courante : on se restreint au cas des **arbres à deux fils**. Plus précisément, la définition est la suivante.

Définition 1.6. *Un arbre binaire est :*
- *soit un arbre vide,*
- *soit la donnée de deux arbres binaires* fils gauche *et* fils droit*.*

Exemple 1.7.
Un arbre binaire. Les arbres vides sont représentés par des segments noirs.



Définition 1.8. *Un arbre binaire est dit* complet *si tous les noeuds ont soit deux fils non vide, soit deux fils vides.*

1.3. **Algorithmes courants.** Les arbres sont des structures **récursives**, la plupart des algorithmes sur les arbres se basent sur cette structure et sont donc eux même récursifs.

Exemple 1.9 (Calcul de la hauteur)**.**

```
Hauteur
Input : Un arbre A
Procédé :
    H <- 0
    Pour chaque fils f de A:
        HF = Hauteur(f)
        Si HF > H:
            H <- HF
    Retourner H + 1
Output : La hauteur de l'arbre
```

Complexité La donnée d'entrée est le **le nombre de nœuds de l'arbre**, c'est-à-dire sa taille. Pour calculer la complexité, on se demande combien de fois chaque nœud est-il visité. Dans le cas de l'algorithme précédent, la réponse est simple : une fois. La complexité est donc en $O(n)$.

Exemple 1.10 (Parcours d'arbre)**.**
Il y a deux façons principales de parcourir un arbre : en **profondeur** et en **largeur**.

Le **parcours en profondeur** suit le principe suivant : je vais parcourir les fils du nœud courant en parcourant *complètement* le sous-arbre d'un fils donné avant de passer au suivant. Sur l'arbre de l'exemple 1.2, on obtient : $3, 5, 2, 3, 4, 2, 7, 1, 9$.

```
ParcoursProf
Input : Un arbre A
Procédé :
    Afficher valeur de A
    Pour chaque fils f de A:
        ParcoursProf(f)
```

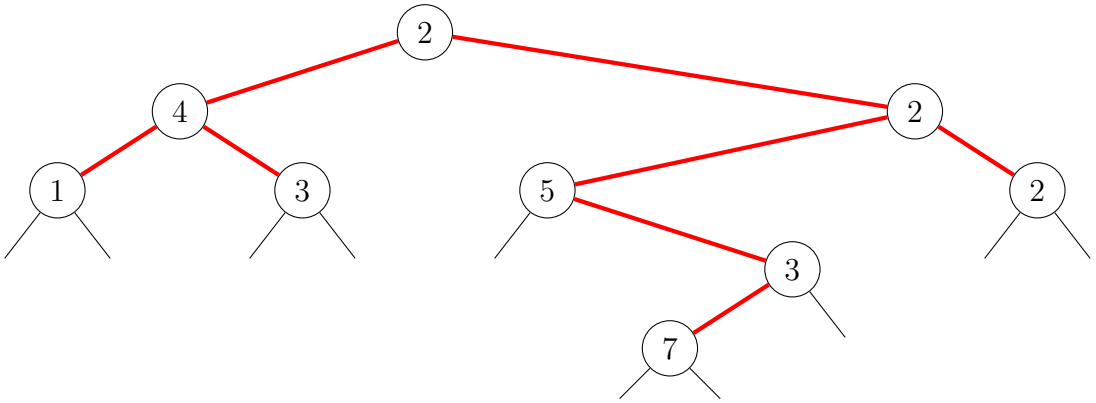
Le **parcours en largeur** est une lecture des nœuds par niveau : d'abord la racine, puis les nœuds à distance 1, puis à distance 2, etc. Sur l'exemple, précédent, on obtient : 3, 5, 4, 2, 2, 3, 7, 1, 9.

L'algorithme est particulier car il s'écrit de façon *itérative*.

```
ParcoursLarg
Input : Un arbre A
Procédé :
    L <- Liste vide
    AjouterFinDeListe(L,A)
    Tant que L n'est pas vide :
        P = SupprimerTeteListe(L) # supprime la tete de liste et la retourne
        Afficher valeur de P
        Pour chaque fils f de P:
            AjouterFinDeListe(L,f)
```

Exemple 1.11 (Parcours d'arbre binaire)**.**
Les arbres binaires peuvent aussi être parcours en largeur où en profondeur. Cependant, on distingue trois cas de parcours en profondeur : **préfixe infixe**, **suffixe**.

Voici un arbre binaire dont les nœuds contiennent des entiers.



Pour le **parcours préfixe**, on lit **la valeur de la racine** puis récursivement, **le fils gauche** et **le fils droit**. Sur l'arbre donné en exemple, on obtient : 2, 4, 1, 3, 2, 5, 3, 7, 2.

Pour le **parcours suffixe**, on lit d'abord récursivement, **le fils gauche** et **le fils droit** puis **la racine**. Sur l'arbre donné en exemple, on obtient : 1, 3, 4, 7, 3, 5, 2, 2, 2.

Enfin pour le **parcours infixe**, on lit d'abord récursivement **le fils gauche**, puis **la racine**, puis **le fils droit** . Sur l'arbre donné en exemple, on obtient : 1, 4, 3, 2, 5, 7, 3, 2, 2.

Les trois algorithmes sont récursifs et sont écrits sur le modèle du parcours en profondeur dans les arbres généraux. Voici par exemple le parcours infixe.

```
ParcoursInf
Input : Un arbre binaire A
Procédé :
    ParcoursInf(A.filsGauche)
    Afficher la valeur de A
    ParcoursInf(A.fisDroit)
```

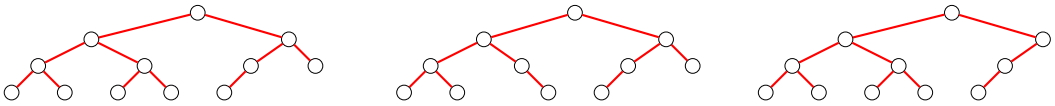
2. Application : tas

Les tas sont des arbres binaires particuliers qui s'adaptent bien à une **structure de tableau**. Ils sont utilisés dans des algorithmes de tris.

2.1. **Arbre binaire parfait et tableau.**

Définition 2.1. *Un arbre binaire est dit* ***parfait*** *si tous ses niveaux sont remplis sauf le dernier niveau (plus éloigné de la racine) et si les nœuds du dernier niveau sont alignés à gauche.*

Exemple 2.2.



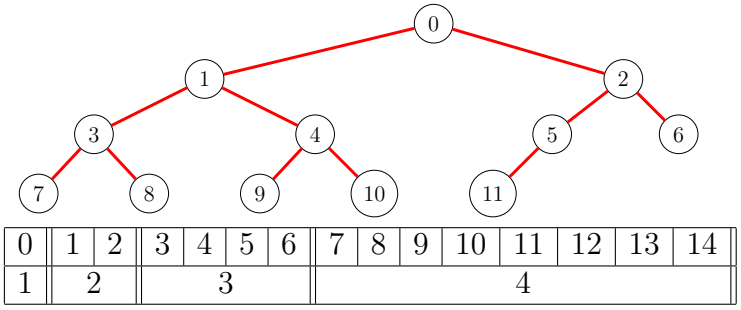
Le premier arbre est un arbre binaire parfait mais pas les deux autres : dans le second, les nœuds du dernier niveau ne sont pas alignés à gauche et dans le troisième, le niveau 3 n'est pas complet.

Étant donné que les niveaux sont remplis au maximum, leur taille est fixée. Ainsi on a :
Niveau 1 : 1 nœud (racine)
Niveau 2 : 2 nœuds
Niveau 3 : 4 nœuds
Niveau 4 : 8 nœuds
...
Niveau n **:** 2^{n-1} nœuds.

Dans l'exemple précédent, les trois premiers niveaux étaient remplis ainsi que 5 des 8 nœuds du niveau 4.

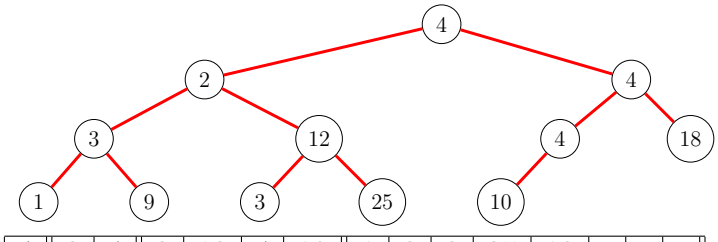
Cette structure particulière nous permet de représenter les arbres binaires parfaits en mémoire de façon beaucoup plus simple que pour les arbres binaires généraux. En effet, plutôt que d'utiliser la structure récursive classique, on peut utiliser un **tableau**. **L'indice d'une valeur dans le tableau détermine sa position dans l'arbre**.

Exemple 2.3.
On a donné à chaque nœud la valeur de son indice dans le tableau.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2		3				4							

A présent, si on utilise l'arbre pour stocker des entiers, voilà un exemple d'arbre et sa représentation en mémoire avec un tableau.



4	2	4	3	12	4	18	1	9	3	25	10			
1	2		3				4							

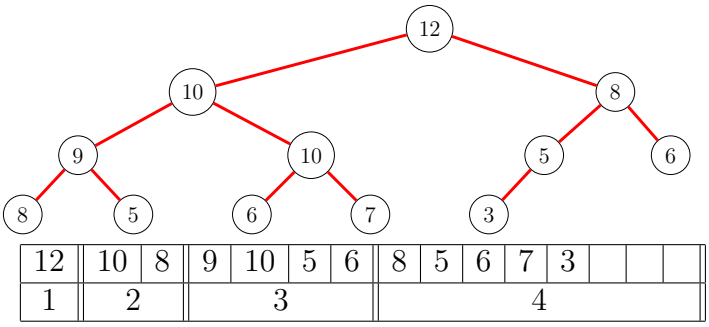
Il faut bien comprendre que la seule donnée en mémoire sera celle du tableau et qu'il *symbolise* l'arbre binaire.

Questions : Si un nœud est à l'indice i,
— Quel est l'indice de son nœud père ? $\lfloor \frac{i-1}{2} \rfloor$
— Quel est l'indice de son fils gauche ? $2i+1$
— Quel est l'indice de son fils droit ? $2i+2$
— A quel parcours de l'arbre correspond la lecture des valeurs du tableau ? Parcours en largeur
— Combien d'éléments peut-on mettre dans un arbres binaire parfait de hauteur h ? $2^h - 1$
— Quelle est la hauteur d'un arbre binaire parfait de taille n ? $\log(n) + 1$

2.2. Tas.

Définition 2.4. *Un Tas (heap en anglais) est un arbre binaire parfait tel que la valeur de chaque nœud soit supérieure ou égale à la valeur de ses fils.*

Exemple 2.5.



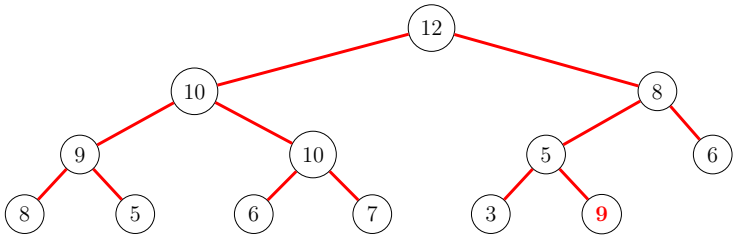
En particulier, l'élément maximal du tableau se trouve toujours à la racine du tas, donc en position 0 dans le tableau. On s'intéresse à deux algorithmes en particulier : **l'insertion dans le tas** et la **suppression du maximum**.

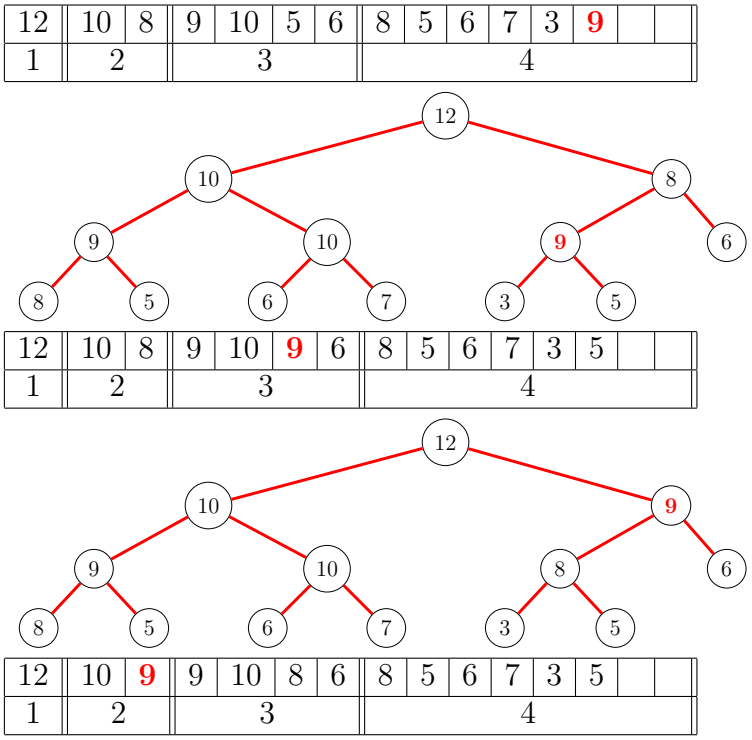
2.2.1. *Insertion.*

Pour **l'insertion**, on place le nouvel élément dans la première position libre puis on le fait remonter dans le tas jusqu'à ce que la condition soit atteinte.

```
InsertTas
Input :
    - t, un tableau qui représente un tas de taille n
    - v, l'élément à insérer
Procédé :
    t[n] <- v
    Tant que n > 0 :
        pere <- (n-1)/2
        Si T[pere] < T[n] :
            T[pere], T[n] <- T[n], T[pere]
            n <- pere
        Sinon :
            Sortir de la boucle
```

Exemple 2.6.
Insertion de 9 dans le tas précédent.



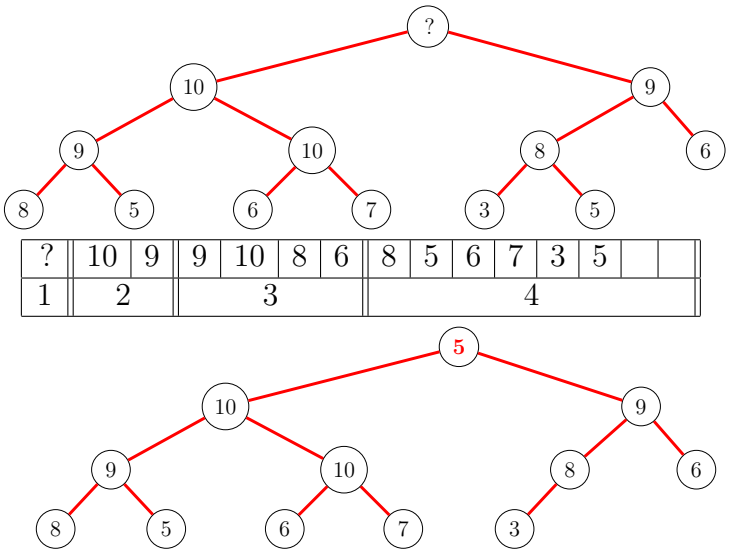


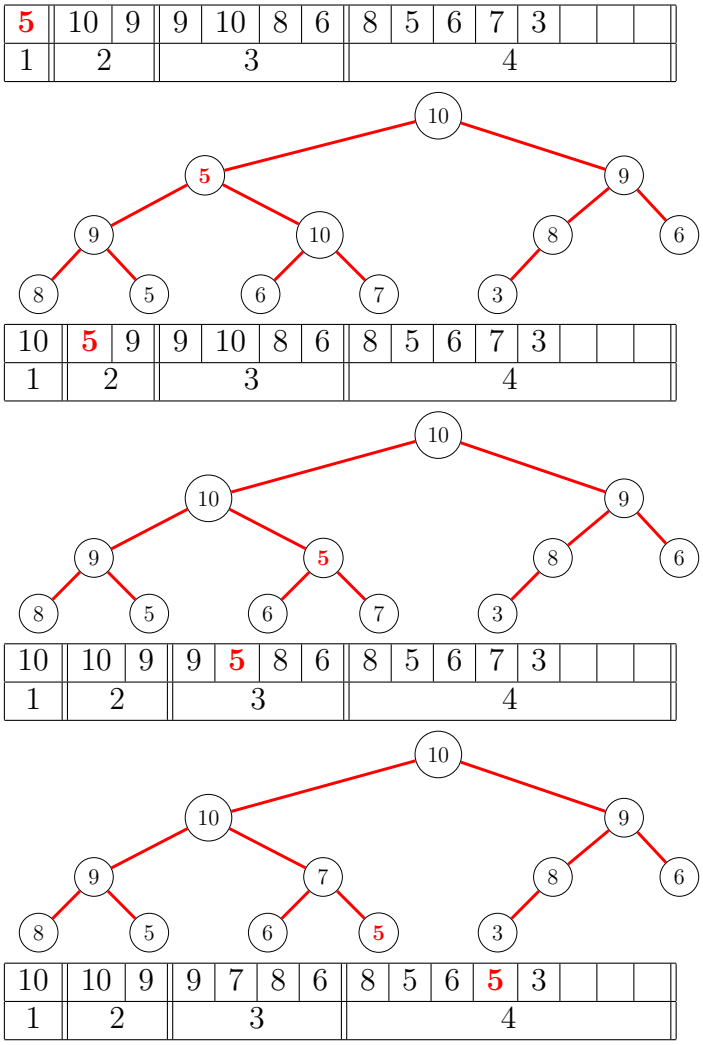
Exercice : la complexité de l'algorithme d'insertion est en $O(\log(n))$.

2.2.2. *Suppression du maximum.* Nous avons vu l'insertion, maintenant, regardons la suppression de l'élément maximum.

- En supprimant un élément du tableau, la condition de tassement sur les arbres binaires parfaits nous dit que la case laissée vide doit être **la dernière case du tableau**. On place donc le dernier élément du tableau en position racine.
- Il faut maintenant faire en sorte que la condition de décroissance du tas soit respectée. Si l'élément n'est pas plus grand que ses deux fils, **on l'échange avec le maximum des fils** ainsi la condition sera respectée localement au niveau de la racine.
- Si un des fils a été modifié, il faut vérifier la condition de décroissance sur ce fils et ainsi "descendre dans le tas" jusqu'à ce qu'on effectue plus d'échange ou qu'on ait atteint une feuille.

Exemple 2.7.





L'implantation précise de l'algorithme est laissée en exercice. Tout comme pour l'insertion, la complexité est en $O(\log(n))$.

2.3. **Tri par tas.**

2.3.1. *Algorithme.* On peut utiliser les deux algorithmes précédents pour trier les données d'un tableau, c'est ce qu'on appelle le **tri par tas** ou **heapsort** en anglais. Le principe est le suivant.

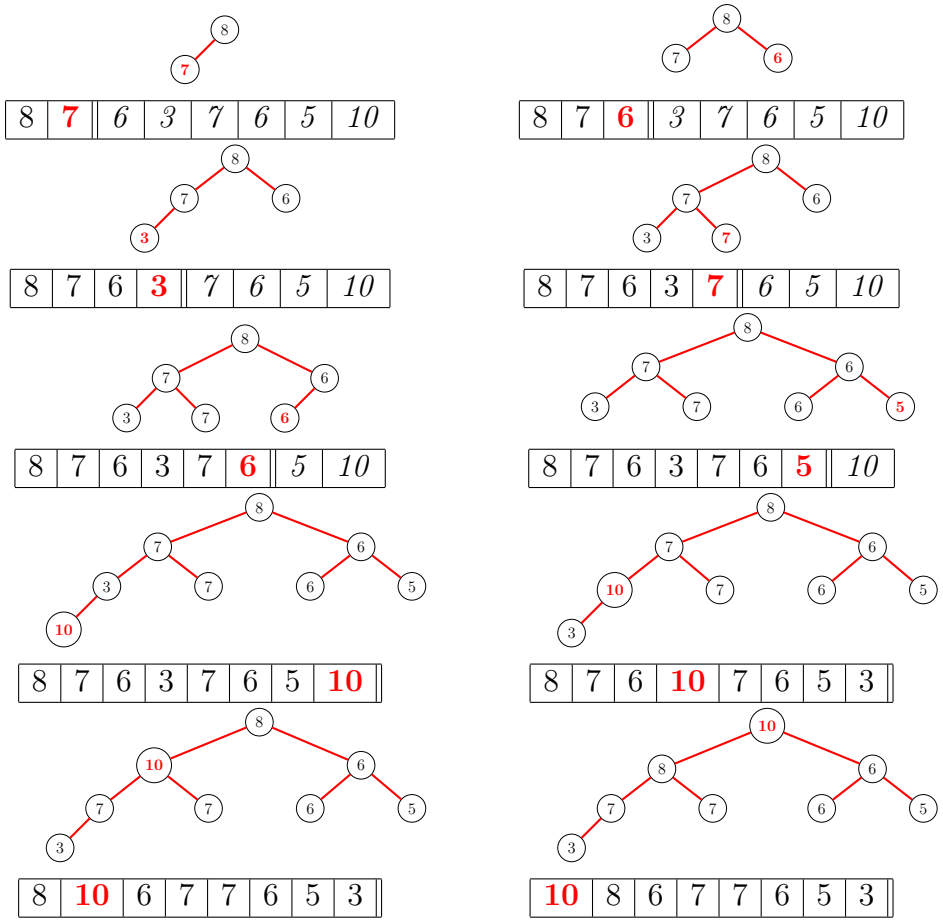
- La première étape consiste à fabriquer un tas en insérant une à une les valeurs du tableau selon l'algorithme d'insertion vu précédemment.
- La seconde étape consiste à supprimer une à une les données du tas. Chaque donnée supprimée est le maximum du tas, soit le maximum des données restantes : on reconstruit donc la liste des données de façon décroissante.

L'ensemble du tri peut se faire *en place* à l'intérieur d'un unique tableau. Au cours de chaque opération, le tableau sera toujours divisé en deux parties : le tas formera le début du tableau et le reste du tableau contiendra les données qui ne sont pas dans le tas. Pour l'étape 1, cela correspond aux données non triées que l'on va insérer dans le tas. Pour l'étape 2, ce sont les données déjà triées qui ont été successivement supprimées du tas.

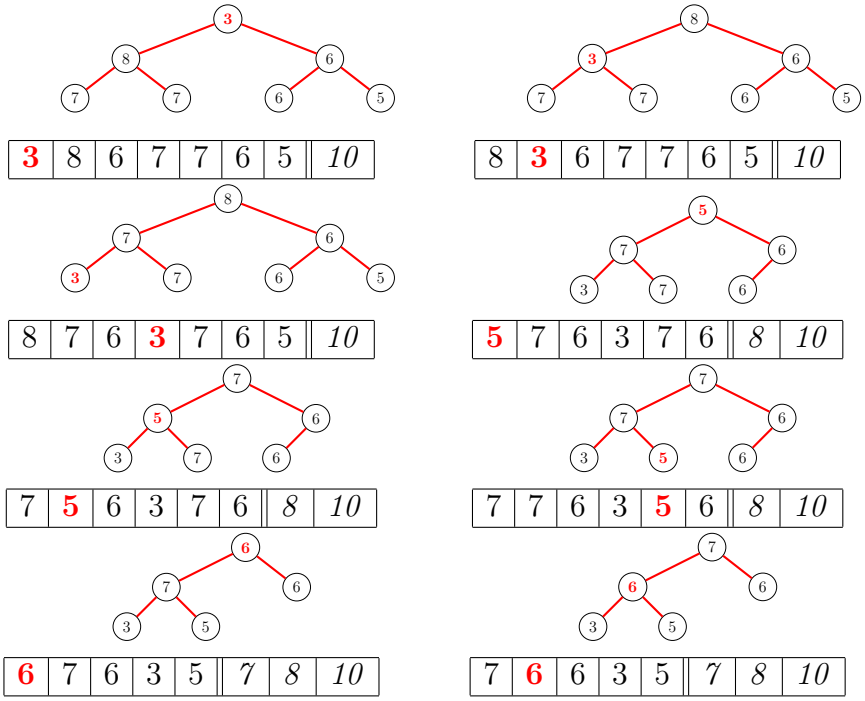
Exemple 2.8.
Tri du tableau

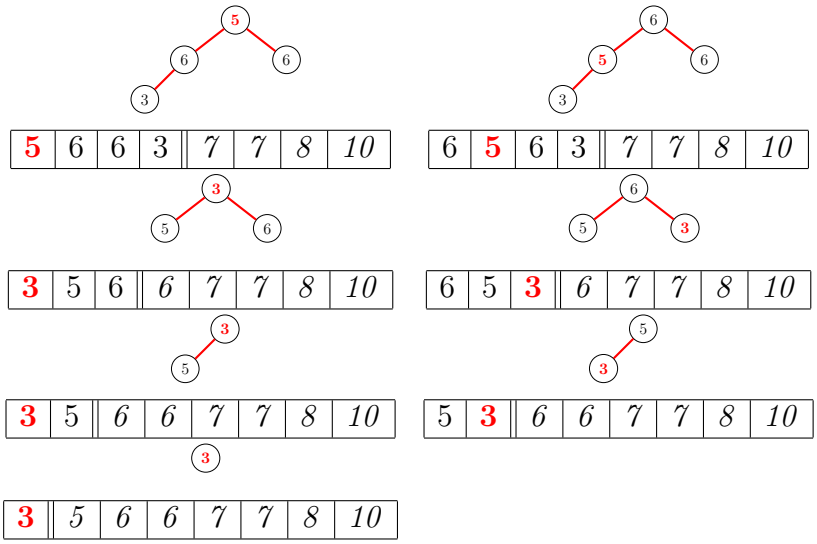
8	7	6	3	7	6	5	10

Etape 1 : Au départ on considère que le tas n'est formé que du premier élément du tableau,on insère les valeurs une à une.



Étape 2 : Toutes les valeurs du tableau on été insérées dans le tas, le maximum se trouve à la racine. On place alors ce maximum à la fin du tableau et on applique l'algorithme de suppression de la racine.





2.3.2. *Complexité.* Tel que nous l'avons décrit l'algorithme a une **complexité** en $O(n\log(n))$ aussi bien dans le pire des cas que en moyenne. Plus précisément, **chacune des deux étapes** à une complexité en $O(n\log(n))$, car chaque insertion et chaque délétion a une complexité en $O(\log(n))$. Il est possible d'effectuer l'étape 1 en $O(n)$ en partant d'un arbre binaire parfait contenant toutes les valeurs du tableau que l'on transforme en tas directement sans passer par une insertion : nous verrons cet algorithme en TP.

2.3.3. *Avantages / Inconvénients.*

Avantages
- Complexité $O(n\log(n))$ dans le pire des cas et en moyenne
- tri en place

Inconvénients
- Complexité $O(n\log(n))$ dans le meilleur des cas (donc pas linéaire)
- Tri non stable
- Implantation contre intuitive (on place d'abord les "grandes" valeurs au début)

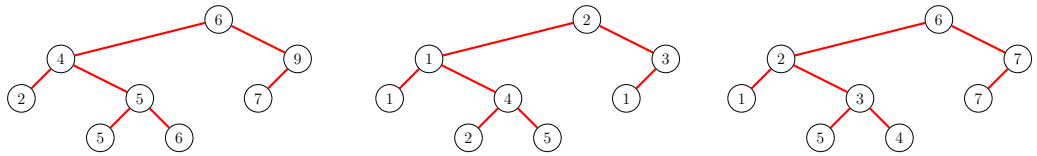
En pratique, le tri par tas est souvent plus lent que le quicksort mais sa meilleure complexité du pire des cas lui donne tout de même un avantage sur ce dernier.

3. Application : Arbre binaire de recherche

3.1. **Définition.** Les **Arbres binaires de recherche** ou (Binary Searh Tree en anglais) sont un autre exemple d'utilisation de la structure d'arbres binaires pour le tri de donnée.

Définition 3.1. *Un arbre binaire de recherche est un arbre binaire contenant des valeurs numériques tel que pour chaque nœud de valeur x dont les sous arbres gauche et droit sont T_G et T_L respectivement, on ait :*
- *toutes les valeurs de T_G sont **inférieures ou égales** x,*
- *toutes les valeurs de T_L sont **supérieures strictes** à x.*

Exemple 3.2.



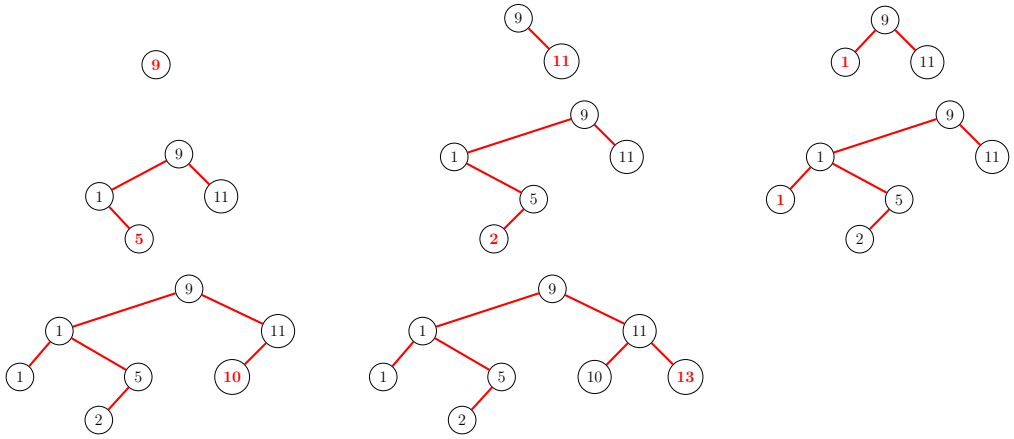
Le premier arbre est bien un arbre binaire de recherche mais pas les deux autres. Dans le second, on trouve par exemple les valeurs 4 et 5 dans le sous arbre gauche de la racine qui a pour valeur 2. Dans le troisième, on trouve la valeur 5 dans le sous-arbre gauche d'un nœud de valeur 3.

Question : quel parcours de l'arbre binaire de recherche donne la liste des valeurs dans l'ordre croissant ? Le parcours infixe.

3.2. **Insertion.** On cherche à rajouter une valeur dans l'arbre binaire de recherche. Pour cela, on va laisser intacte la structure existante et trouver **l'unique arbre vide** (fils droit ou fils gauche d'un nœud existant) tel qu'on puisse le remplacer par la nouvelle valeur en conservant les conditions d'un arbre binaire de recherche. On peut décrire l'algorithme récursivement de cette façon :

— Si l'arbre est vide, alors la racine de l'arbre devient un nœud contenant la nouvelle valeur.
— Sinon, si la valeur à insérer est plus petite ou égale que la valeur de la racine, je l'insère dans le sous-arbre gauche et sinon, dans le sous-arbre droit.

Exemple 3.3.
Insertion successive des valeurs : 9, 11, 1, 5, 2, 1, 10, 13.



Remarque : La complexité de l'insertion dépend de la hauteur de l'arbre. Dans un arbre binaire classique la hauteur n'est pas fixée (contrairement à un arbre binaire parfait). Dans le pire des cas, la hauteur peut être de taille n et l'insertion en $O(n)$, dans le meilleur des cas on obtient du $O(\log(n))$. Si on insère des données successives qui correspondent à un tirage aléatoire uniforme, on se retrouve dans un cas très similaire à celui du quicksort et on obtient une complexité de $O(\log(n))$ en moyenne.

3.3. **Tri par arbre binaire.** L'algorithme d'insertion donne directement un algorithme de tri : insertion des valeurs une à une dans l'arbre binaire, puis parcours infixe pour obtenir les données dans l'ordre croissant. Comme on l'a dit plus haut, la complexité dépendra des valeurs d'entrées. Dans le **meilleur des cas**, on obtiendra du $O(n\log(n))$ (car on effectue n insertions) et ce sera aussi la **complexité en moyenne**. Mais dans le **pire des cas**, on aura une complexité en $O(n^2)$.

Avantages
— Complexité $O(n\log(n))$ en moyenne

Inconvénients
— Complexité $O(n^2)$ dans le pire des cas
— Complexité $O(n\log(n))$ dans le meilleur des cas (donc pas linéaire)
— Utilise une structure externe (en particulier, complexité mémoire $O(n)$)

En pratique, ce tri a le même gros défaut que le tri rapide avec une complexité du pire des cas quadratique sans pour autant avoir ses avantages (tri en place et rapide en pratique). Il ne sera donc jamais utilisé tel quel. Cependant, une amélioration simple consiste à **rééquilibrer** l'arbre au moment de chaque insertion ce qui permet d'obtenir une complexité globale en $O(n\log(n))$. La nécessité d'une structure externe d'arbre binaire le rend peu pratique pour trier les données d'un tableau (par rapport à un tri rapide ou un tri par

tas) mais la structure pourra être utilisée pour stocker directement des données de façon ordonnée. Cette structure est à la base de nombreuses généralisations : arbres AVL, arbres rouges-noirs par exemple.