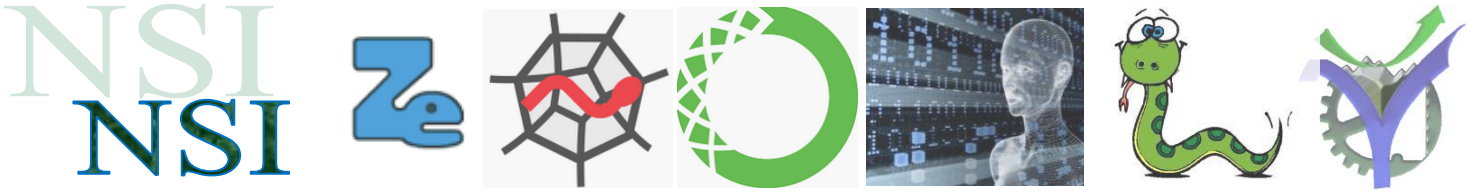
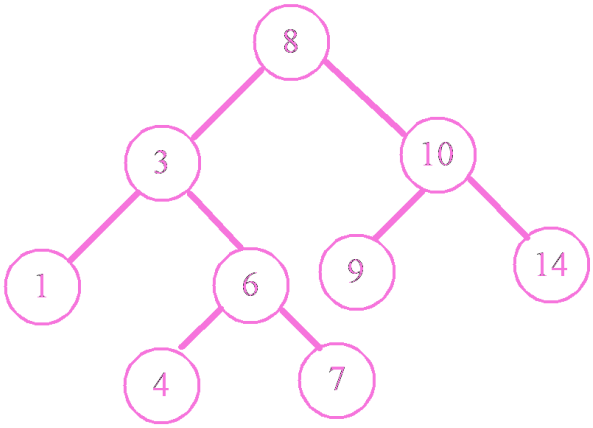


# Les arbres binaires de recherche ou ABR

## Résumé

Les arbres binaires de recherche sont des arbres binaires possédant une organisation particulière des nœuds.

Cette organisation permet d'effectuer des recherches de manière très efficace puisqu'elles sont effectuées dans le meilleur des cas avec une complexité dépendant de la hauteur de l'arbre.




## Sommaire

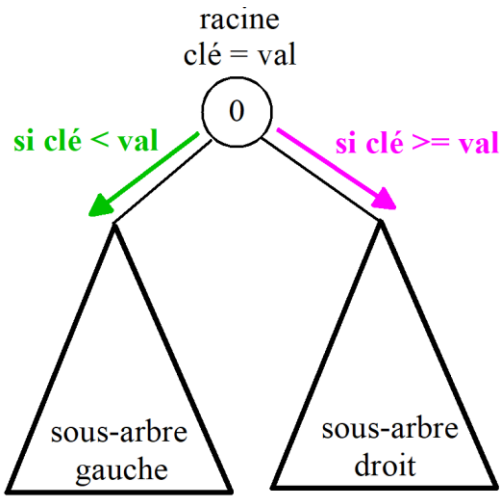
**1 Les arbres binaires de recherche ..... 2**
*1.1 Présentation ..... 2*
*1.2 Exemples ..... 2*
*1.3 Utilisation des arbres binaires de recherche ..... 3*
*1.4 Vers la forme optimale : arbres binaires de recherche complet, parfait ..... 4*
*1.5 Création d'un arbre de recherche parfait un défi à relever ! ..... 5*
*1.6 Le tri de valeurs avec des arbres binaires de recherche ..... 6*
*1.7 Opérations sur un arbre binaire de recherche ..... 6*
*1.8 Suppression de nœuds, successeur, prédécesseur ..... 7*
*1.9 Bilan en termes de complexité ..... 9*

**2 Implémentation des Abres Binaires de Recherche ..... 9**
*2.1 Implémentation en Python ..... 9*
*2.2 Ajout d'un nouveau nœud ..... 10*
*2.3 Recherche d'élément, d'un maximum ou d'un minimum ..... 10*
*2.4 Tri d'une liste avec un parcours infixe sur un ABR ..... 11*
*2.5 Le retour du défi ..... 11*

# 1 Les arbres binaires de recherche

## 1.1 Présentation

Un arbre binaire de recherche est un arbre binaire dans lequel les nœuds identifiés par des clés sont organisés par une relation d'ordre.




Lors de la création de l'arbre, ou bien lors de modifications, le nouveau nœud est inséré dans le sous arbre de gauche si la valeur de sa clé est inférieure à la clé racine, il est inséré dans le sous arbre de droite si cette valeur est supérieure ou égale à la clé racine.

Ce processus est itératif et s'applique à tous les nœuds existants lors d'un ajout ou d'une recherche :

- A gauche si la valeur est inférieure.
- A droite si la valeur est supérieure ou égale.

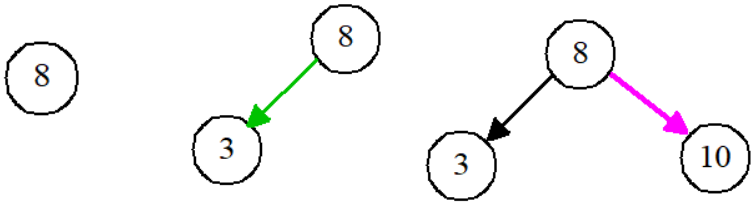
Cette organisation rend les recherches extrêmement efficaces. En effet nous verrons qu'elles sont de complexité $O(h)$ avec h la hauteur de l'arbre, ou $O(\log_2(N))$ avec N le nombre total de nœuds.

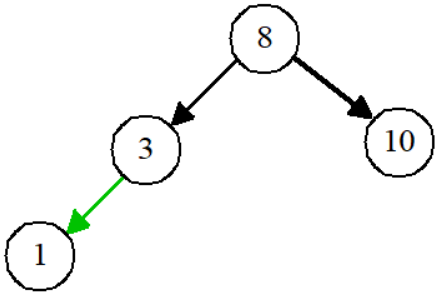
## 1.2 Exemples

Représentons l'ABR correspondant à l'insertion de nœuds dont les clés sont dans l'ordre de la liste ci-dessous :

8, 3, 10, 1, 6, 14, 4, 7, 13

1. On commence par le nœud 8
2. Ajout du nœud 3
3. Ajout du nœud 10
4. Ajout du nœud 1




**Q1.** Terminez le travail.

**Q2.** Même travail avec la suite nœuds 8, 13, 1, 10, 5

Nous pouvons observer que pour tous nœuds de l'arbre la valeur de sa clé est supérieure à toutes les clés des nœuds du sous arbre de gauche et est inférieure ou égale à toutes les valeurs des clés du sous arbre de droite.

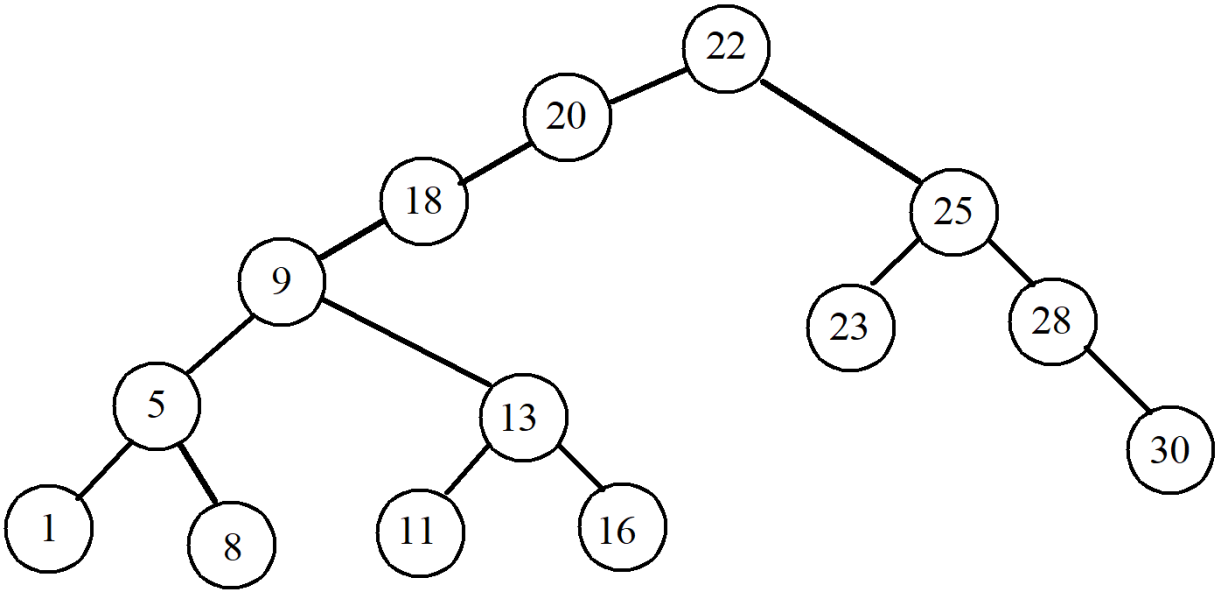
**Q3.** Représentez l'arbre dont les nœuds sont insérés avec les clés dans l'ordre ci-dessous :

```
liste_valeurs = [20,7,53,4,15,2,11,18,46,80,29,37,69,86,74,75]
```

## 1.3 Utilisation des arbres binaires de recherche

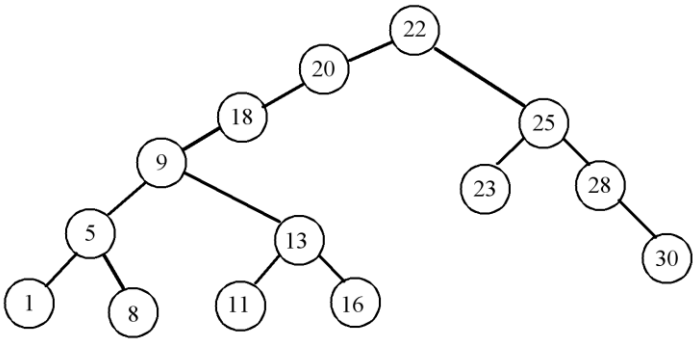
La relation d'ordre utilisée dans la structuration d'un arbre binaire de recherche permet de rendre très efficace les opérations de recherche …

Examinons les performances sur un arbre construit avec les valeurs des clés dans cet ordre 22, 25, 28, 30, 23, 20, 18, 9, 13, 5, 8, 11, 1, 16, 6, 3 :

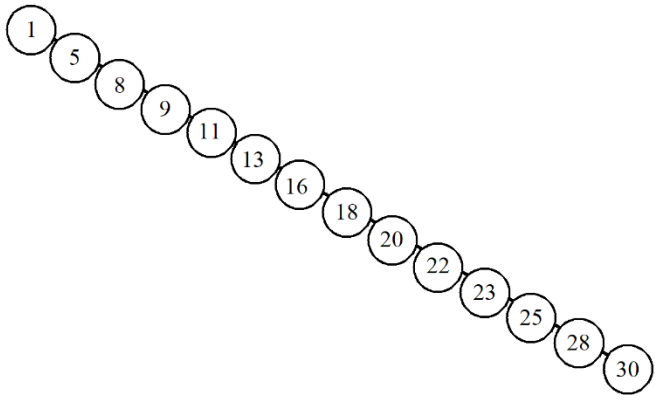



**Q4.** Quelle est la hauteur de cet arbre ?

**Q5.** En combien d'étapes pouvons-nous déterminer si la clé 17 est dans l'arbre ? Et la clé 30 ?



Les mêmes valeurs de clés sont rentrées maintenant dans un ordre croissant. L'arbre binaire de recherche obtenu est alors le suivant :




**Q6.** Quelle est la hauteur de cet arbre ?

**Q7.** Combien d'étapes pour la recherche de la clé 30 ?

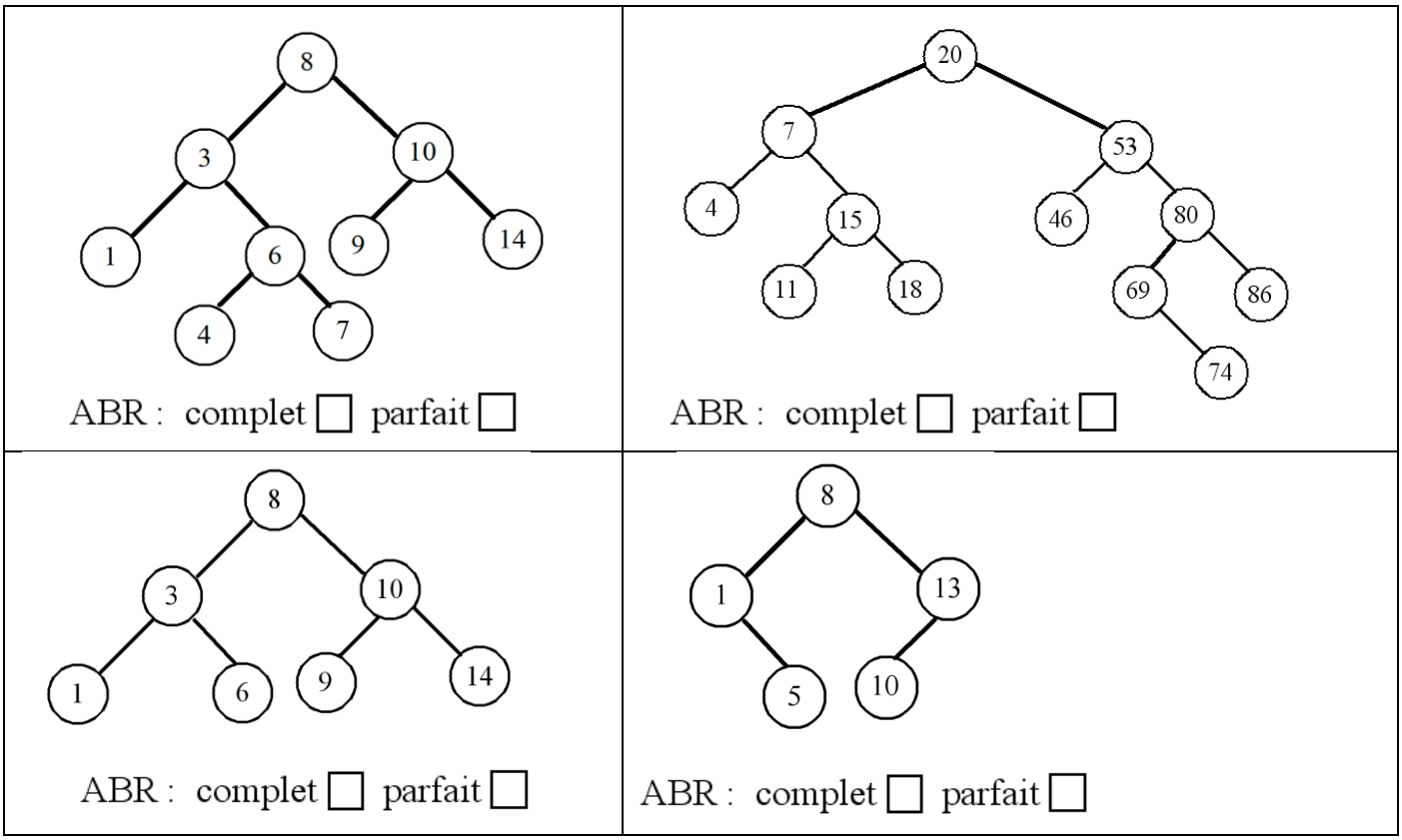
**Ce résultat est très important : le nombre d'étapes pour effectuer une recherche dépend de la hauteur de l'arbre. On est en *O*(h).**

Un arbre binaire de recherche doit donc être construit avec une hauteur minimale pour avoir de bonnes performances. Or un arbre binaire de recherche voit sa structure évoluer en fonction des ajouts et suppressions de clés au fur et à mesure de son utilisation.

## 1.4 Vers la forme optimale : arbres binaires de recherche complet, parfait

On appelle **arbre binaire complet** un arbre binaire tel que chaque sommet interne a exactement 2 fils. La propriété ne s'applique donc pas aux feuilles qui peuvent par ailleurs ne pas être sur le même niveau.

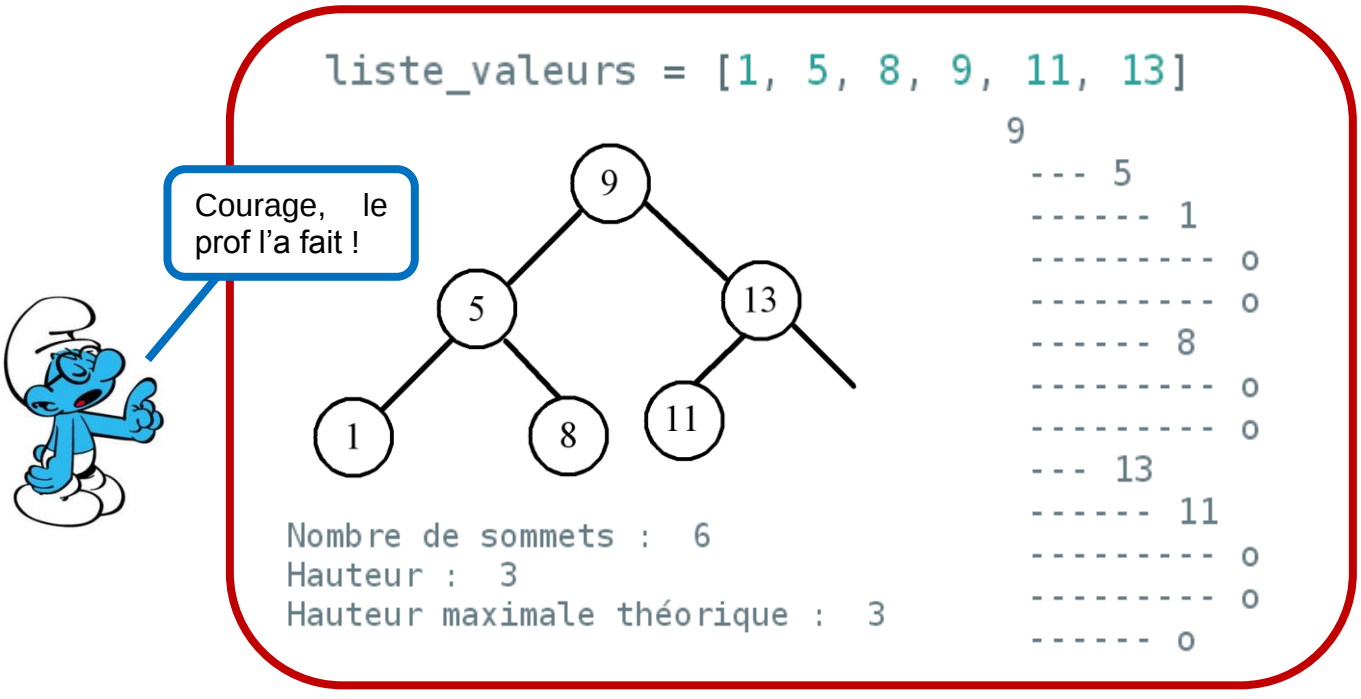
On appelle **arbre binaire parfait** un arbre binaire complet dans lequel toutes les feuilles sont à la même hauteur dans l'arbre.



**Q8.** Dessiner des arbres binaires de recherche de hauteur 3, 4, 5 avec le même ensemble de clés : 1, 4, 5, 10, 16, 17, 21.

## 1.5 Création d'un arbre de recherche parfait un défi à relever !

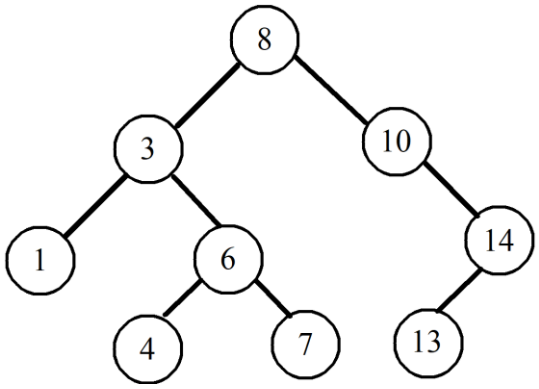
Il faut donc réussir à construire un arbre binaire de recherche parfait. Voilà un défi à relever !

## 1.6 Le tri de valeurs avec des arbres binaires de recherche

Les arbres binaires de recherche permettent les opérations de tris.

**Q9.** Réalisez le parcours infixe de l'arbre ci-dessous, que constatez-vous ?




## 1.7 Opérations sur un arbre binaire de recherche

Comme tous les types abstraits de données les arbres binaires de recherche sont pourvus d'opérations permettant de les manipuler.

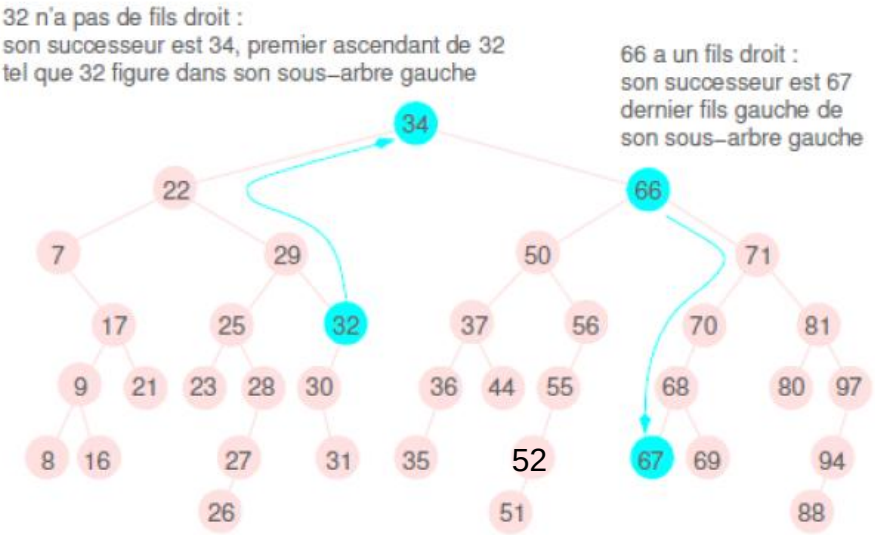
| |
|---|
| estVide(Arbre) :<br>Renvoi un Booléen à *Vrai* si l'arbre ne contient aucun élément sinon renvoi la valeur *Faux*. |
| Ajoute(x,Arbre)<br>Ajoute un nœud avec la valeur de clé x dans l'arbre. |
| hauteur(Arbre) :<br>Renvoi la hauteur de l'arbre. |
| taille(Arbre) :<br>Renvoi le nombre de nœuds de l'arbre. |
| imprimeArbre(Arbre)<br>Imprime un arbre. |
| rechercheValeur(x,Arbre)<br>Retourne un booléen à *Vrai* si la valeur x est présente dans l'arbre et retourne *Faux* dans le cas contraire. |
| rechercheMaximum(Arbre)<br>Renvoi la valeur la plus grande de l'arbre. |
| rechercheMinimum(Arbre)<br>Renvoi la valeur la plus petite de l'arbre. |

Les opérations de parcours en profondeur et en largeur vus dans le précédent document sont bien sûr utilisables.

## 1.8 Suppression de nœuds, successeur, prédécesseur

D'autres opérations sont possibles avec les arbres binaires de recherche. Elles ne sont pas au programme de la terminale NSI. Nous les citons à titre d'illustration[1] :

### Le successeur d'un nœud



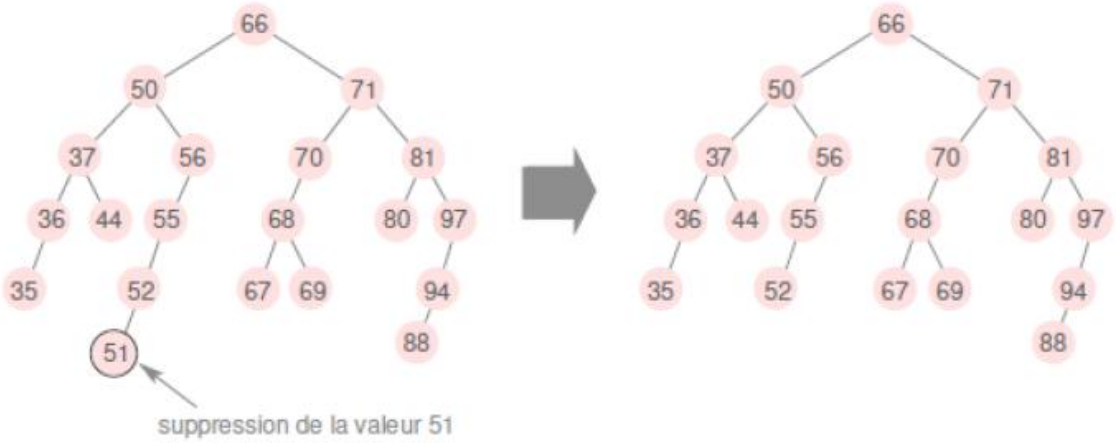

### Le prédécesseur d'un nœud

Le prédécesseur d'un nœud x est le nœud possédant la plus grande clé inférieure à clé(x)

Exemple sur l'arbre ci-dessus : 32 ⇨ 31 ; 17 ⇨ 16

### La suppression de nœuds

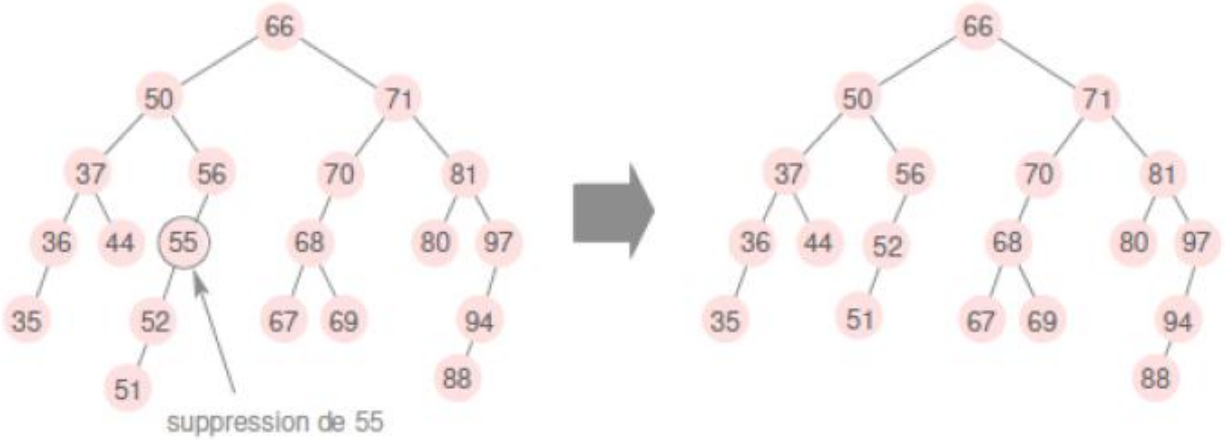
Cas 1 : pour une feuille




⇦

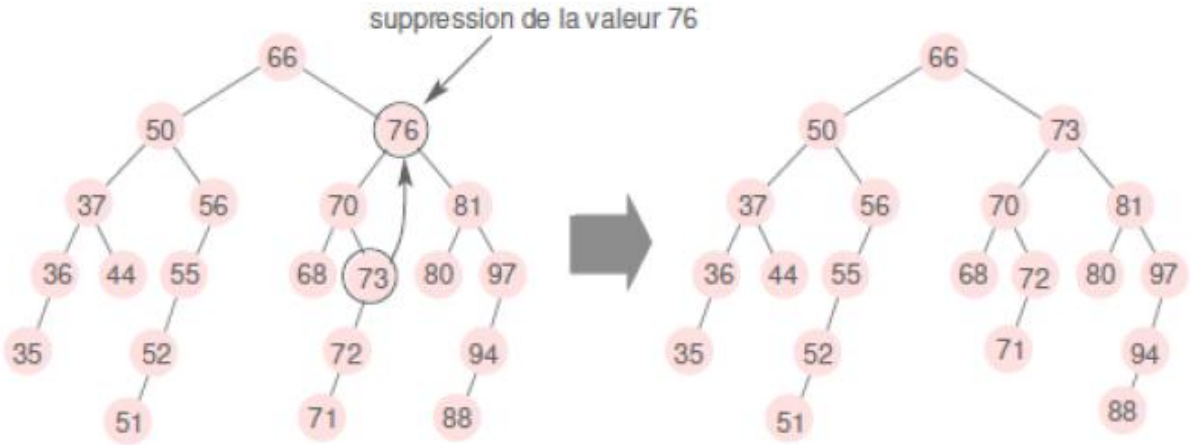
[1] Illustration issues du document
http://ressources.unisciel.fr/algoprog/s46bst/emodules/br00macours1/res/br00cours-texte-xxx.pdf

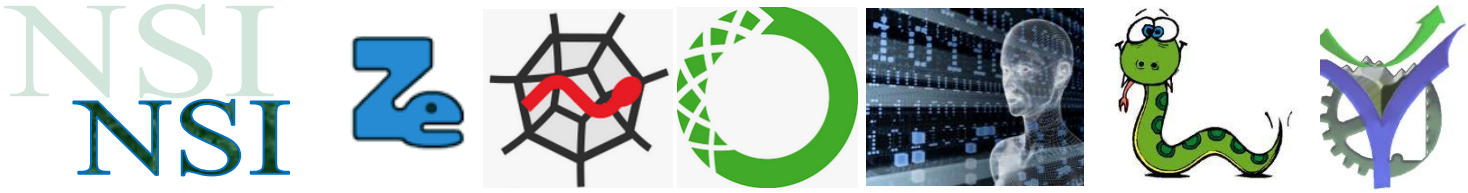
Cas 2 : pour un nœud avec un unique fils




Cas 3 : pour un nœud avec deux fils

Le noeud à supprimer a deux fils : on le remplace par son prédécesseur $q$ qui, dans ce cas, est toujours le maximum de son sous-arbre gauche (on peut prendre indifféremment son successeur qui est alors le minimum de son sous-arbre droit). Puisque le noeud $q$ a la valeur la plus grande dans le fils gauche, il n'a donc pas de fils droit, et peut être décroché comme on l'a fait dans les cas 1 et 2.





## 1.9 Bilan en termes de complexité

La complexité des algorithmes œuvrant sur les arbres de recherche dépend de la forme de l'arbre. Dans le pire cas la complexité est en O(n) où n est le nombre de nœud. Dans le cas où l'arbre est équilibré la complexité est idéale et est en O(h) et donc ne dépend que de la hauteur de celui-ci.

Si on insère dans un arbre binaire de recherche les clés dans un ordre aléatoire alors la théorie nous démontre que le comportement du cas moyen est plus proche du cas optimal que du cas défavorable.

**Q10.** Pour illustre l'assertion précédente créez tous les arbres binaires de recherche contenant les nœuds de valeurs de clés 1, 2, 3. Quelle est la proportion d'arbres parfait, donc ayant la meilleure efficacité, dans cette façon de procéder.

# 2 Implémentation des Abres Binaires de Recherche

## 2.1 Implémentation en Python

Nous retrouvons la même implémentation que celle utilisée pour les arbres car les arbres binaires de recherche en sont un cas particulier :

On retrouve la définition de la classe Nœud :

```
class Noeud:
    def __init__(self, x, fils_gauche=None, fils_droit=None):
        self.val = x
        self.fg = fils_gauche
        self.fd = fils_droit
```

Et la possibilité d'associer deux sous-arbres :

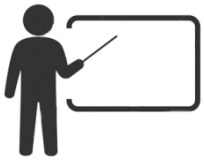
```
def nouveauNoeud(x, arbre_gauche, arbre_droit):
    '''
    Créé un noeud de valeur x en associant un sous arbre
    gauche et un sous arbre droit.
    '''
    return Noeud(x, arbre_gauche, arbre_droit)
```

## 2.2 Ajout d'un nouveau nœud

Voilà l'algorithme d'ajout d'un nouveau nœud dans un ABR a :

```
fonction ajoute(x,a)
si a est vide alors
        renvoyer Nœud(x,None,None)
        fin si
si x < a.val alors
        renvoyer Nœud(a.val, ajoute(x,a.fg), a.fd)
sinon si x > a.val alors
        renvoyer Nœud(a.val, a.fg, ajoute(x,a.fd))
sinon
        afficher 'Ajout refusé valeur déjà existante'
        renvoyer a
fin si
```

Script_ABR_1. Compléter dans le script ci-dessous la fonction ajoute(x,a). Utiliser cette fonction pour créer un ABR. Script_ABR_1.py



## 2.3 Recherche d'élément, d'un maximum ou d'un minimum

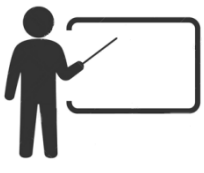
### Recherche de l'existence d'un élément

Utilisez une boucle non bornée while pour effectuer une recherche de la présence d'une valeur particulière.

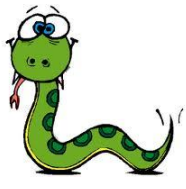
```
def rechercheValeur(x,arbre):
    '''
    Recherche d'une valeur dans un ABR
    Si l'ABR est vide renvoi False
    Si l'arbre est balayé sans trouver la valeur
    renvoi False.
    '''
    # A compléter
    while arbre is not None:


    return False
```

Script_ABR_2. Compléter le script ci-dessous avec la fonction rechercheValeur(x,arbre)



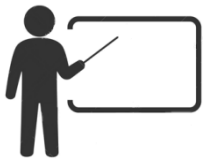


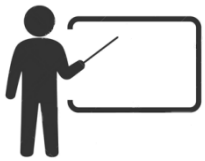

[Recherche des valeurs maximum et minimum]

Pour chaque cas on proposera une implémentation en programmation impérative, avec des boucles non bornée while, puis une implémentation récursive.

Script_ABR_3. Compléter le script ci-dessous avec la fonction rechercheMaximum(arbre)
puis rechercheMaximumRecursif(arbre)



Script_ABR_4. Compléter le script ci-dessous avec la fonction rechercheMinimum(arbre)
puis rechercheMinimumRecursif(arbre)



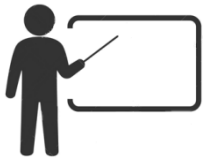
## 2.4 Tri d'une liste avec un parcours infixe sur un ABR

Nous avons vu qu'il est possible de triée une liste de valeurs en utilisant un arbre binaire de recherche. On effectue deux étapes :

a) création de l'ABR de recherche à partir de la liste des clés,

b) parcours infixe de cet arbre pour obtenir la liste triée.

Script_ABR_5. Compléter le script ci-dessous pour réaliser le tri de la liste de valeurs. Vous pouvez utiliser les fonctions définies sur les arbres binaires du chapitre précédent. Script_ABR_5.py



## 2.5 Le retour du défi

Pour rappel on cherche à créer automatiquement un arbre binaire de recherche parfait à partir d'une liste triée.