

# Algorithmes sur les arbres binaires et sur les arbres binaires de recherche

## Cours

Un arbre est une structure de données **hiérarchique** (les nœuds sont liés par des relations père-fils) et récursive.

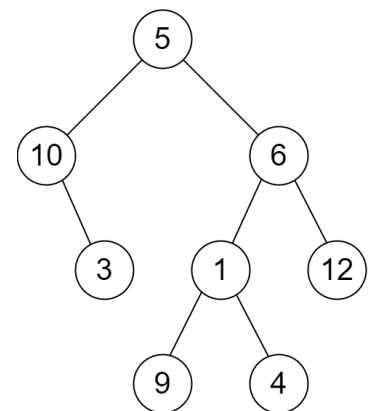
Un **arbre binaire (AB)** est un cas particulier d'arbre où chaque nœud possède au **maximum deux fils ordonnés** : un fils gauche et/ou un fils droit. Ils ne sont pas intervertibles !

La plupart des exercices de baccalauréat n'ont qu'une seule classe récursive et se contentent d'implémenter des arbres non vides, dit "enracinés".

On a **déjà implémenté un arbre binaire et ses nœuds** en Python avec deux classes, une classe `AB` permettant d'implémenter un arbre vide, et une classe `Noeud` récursive :

```
class AB:
    def __init__(self, racine = None):
        self.racine = racine # type : Noeud

class Noeud:
    def __init__(self, v, g = None, d = None):
        self.gauche = g # type Noeud
        self.droite = d # type Noeud
        self.valeur = v
```



Implémentons l'arbre ci-contre.

```
n9, n4 = Noeud(9), Noeud(4)
n1, n12 = Noeud(1, n9, n4), Noeud(12)
n6 = Noeud(6, n1, n12)
n3 = Noeud(3)
n10 = Noeud(10, d = n3)
arbre = AB(Noeud(5, n10, n6))
```

## Calcul de la taille d'un arbre binaire

## Cours

La taille d'un arbre est son nombre de nœuds. Un arbre vide a une taille de 0.

De façon générale, pour calculer la taille d'un arbre, il suffit de compter le nombre de nœuds à partir de la racine en parcourant toutes les branches.

Dans le cas d'un arbre binaire implémenté comme décrit ci-dessus, on peut créer une méthode récursive, `taille()`, de la classe `Noeud`. Cette méthode renvoie la **1 plus somme des tailles des deux fils** d'un noeud. Ainsi on parcourt tout l'arbre en partant de sa racine.

Pour la classe `AB`, on ajoute une autre méthode appelée aussi `taille()` qui renvoie `0` pour l'arbre vide ou sinon la taille de la racine.

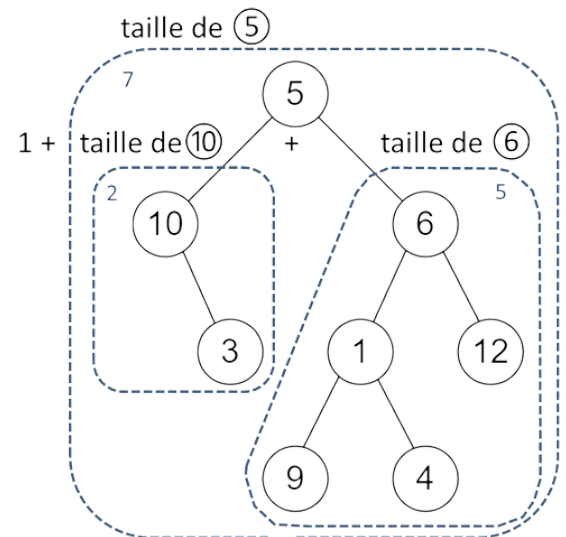
```
class AB:
    def taille(self):
        if self.racine == None: return 0
        # renvoie la taille du noeud racine
        return self.racine.taille()

class Noeud:

    def taille(self):
        # taille du sous arbre gauche
        if self.gauche is None:
            tg = 0
        else:
            tg = self.gauche.taille()

        # taille du sous arbre droit
        if self.droite is None:
            td = 0
        else:
            td = self.droite.taille()

        # la taille est 1 + la somme des deux
        return 1 + td + tg
```



Calculons la taille de l'arbre précédent :

```
>>> arbre.taille()
8
```

Le calcul de la taille a un coût proportionnel à au nombre de nœuds  $n$  de l'arbre (pour chaque nœud supplémentaire, on rajoute 3 opérations). La **complexité du calcul de la taille est en  $O(n)$** .

## Calcul de la hauteur d'un arbre binaire

⚠ Il n'existe pas de définition universelle pour la hauteur d'un arbre et la profondeur d'un nœud dans un arbre. Dans certains cas la profondeur des nœuds est comptée à partir de 0, la hauteur de l'arbre réduit à la racine est 0 et la hauteur de l'arbre vide est -1 (par convention).

## Cours

- La **profondeur** d'un nœud est le nombre de nœuds du chemin qui va de la racine au nœud. La profondeur de la racine est donc 1.
- La **hauteur** d'un arbre est le nombre de nœuds (ou niveaux) du plus long chemin d'une feuille à la racine. Un arbre réduit à la racine a une hauteur de 1, un arbre vide a une hauteur de 0.

Pour calculer la hauteur d'un arbre, il faut parcourir toutes ses branches à partir de la racine et prendre la profondeur de la feuille la plus éloignée.

Dans le cas arbre binaire implémenté comme décrit ci-dessus, on peut à nouveau créer une méthode récursive, `hauteur()`, de la classe `Nœud`. Cette méthode renvoie la **1 plus la plus grande hauteurs** de ses deux fils. ⚠ Les feuilles hauteur ont une hauteur de 1 (avec la convention choisie ici).

Pour la classe `AB`, on ajoute une autre méthode appelée aussi `hauteur()` qui renvoie 0 pour l'arbre vide ou sinon la hauteur du Nœud racine.

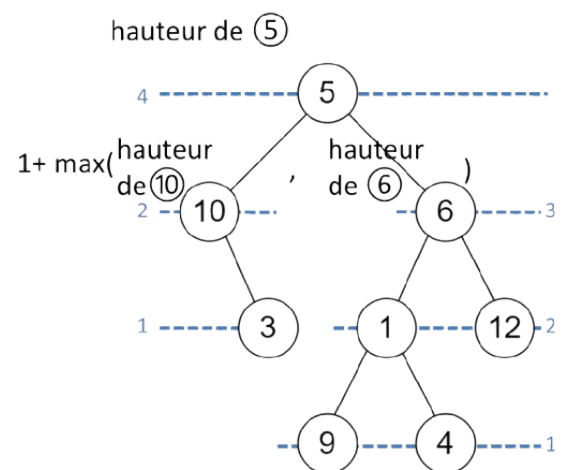
```
class AB:

    def hauteur(self):
        if self.racine == None: return 0
        # renvoie la hauteur du nœud racine
        return self.racine.hauteur()

class Nœud:
    def hauteur(self):
        # hauteur du sous arbre gauche
        if self.gauche is None:
            hg = 0 # par convention l'arbre vide a
une hauteur de 0
        else:
            hg = self.gauche.hauteur()

        # hauteur du sous arbre droit
        if self.droite is None:
            hd = 0 # par convention l'arbre vide a
une hauteur de 0
        else:
            hd = self.droite.hauteur()

        # la hauteur est 1 + le plus grand des deux
        return 1 + max(hg, hd)
```



Le calcul de la hauteur a un coût proportionnel à la taille du nombre de nœuds  $n$  de l'arbre (pour chaque nœud supplémentaire, on rajoute 3 opérations). La **complexité du calcul de la hauteur est en  $O(n)$** .

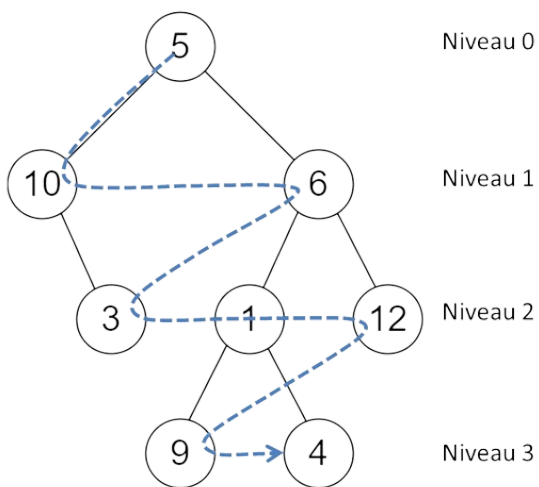
## Parcourir un arbre binaire

Parcourir un arbre binaire consiste à visiter tous les nœuds que contient cet arbre. Il existe de nombreux algorithmes de parcours qui visitent chaque nœuds dans un **ordre différent**, les plus courants<sup>1</sup> sont le parcours en largeur et le parcours en profondeur.

## Parcours en largeur (BFS)

### Cours

Le **parcours en largeur** ou **BFS** (*Breadth First Search*), consiste à parcourir l'arbre **niveau par niveau**, en partant de la racine. Les nœuds de niveau 0 sont d'abord parcourus puis les nœuds de niveau 1 et ainsi de suite. Dans chaque niveau, les nœuds sont parcourus de la **gauche vers la droite**.



L'implémentation se fait naturellement en utilisant une structure en file :

- La racine est d'abord mise dans la file, puis
- Tant que la file n'est pas vide:
  - On défile le premier nœud de la file
  - On note sa valeur dans le parcours
  - On enfile ses fils gauche et droite, dans cet ordre, s'ils existent.

La méthode est itérative, on n'utilise pas la récursivité de la classe `Noeud`, on peut donc l'ajouter au choix dans la classe `Noeud` ou dans la classe `AB` directement:

```

class Noeud:

    def parcours_larg(self):
        parcours = []
        file = []          # file de nœuds en attente

        file.append(self)
        while len(file) != 0:  # tant que la file n'est pas vide
            n = file.pop(0)    # on défile le premier nœud de la file
            parcours.append(n.valeur)  # on note sa valeur
            # on enfile le fils gauche s'il existe
            if n.gauche is not None:
                file.append(n.gauche)
  
```

```
# on enfile le fils droit s'il existe
if n.droite is not None:
    file.append(n.droite)

return parcours
```

et l'appel de la méthode pour le nœud racine :

```
class AB :
    def parcours_larg(self):
        if self.racine is None:
            print("l'arbre est vide")
        else:
            return self.racine.parcours_larg()

>>> arbre.parcours_larg()
[5, 10, 6, 3, 1, 12, 9, 4]
```

Le parcours en largeur contient une boucle tant que la file n'est pas vide c'est-à-dire tant que tous les nœuds n'ont pas été visités, le coût est donc proportionnel à la taille du nombre de nœuds  $n$  de l'arbre. La **complexité du parcours en largeur est en  $O(n)$** .

## Parcours en profondeur (DFS)

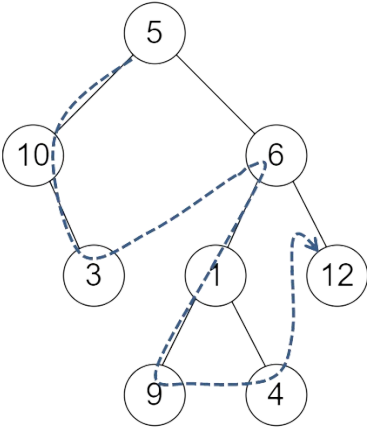
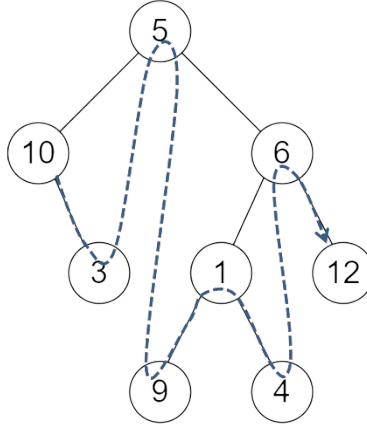
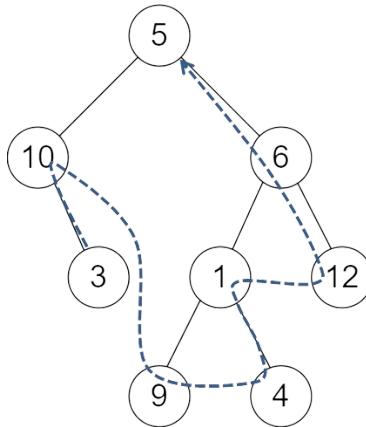
### Cours

Le **parcours en profondeur** ou **DFS** (*Depth First Search*) consiste à parcourir pour chaque nœud d'abord le sous-arbre gauche entièrement et ensuite le sous arbre droit.

Chaque nœud est parcouru trois fois, on peut donc décider de traiter le nœud :

- Avant le parcours de son sous-arbre gauche : parcours **préfixe** (ou **préordre**) ;
- Entre le parcours de sous-arbre gauche et celui de son sous-arbre droit : parcours **infixe** (ou **en ordre**) ;
- Après le parcours de son sous-arbre droit : parcours **postfixe** (ou **suffixe**, ou encore **postordre**).

Parcours préfixe	Parcours infixes	Parcours postfixes
1. Visite du nœud 2. Parcours branche gauche 3. Parcours branche droite	1. Parcours branche gauche 2. Visite du nœud 3. Parcours branche droite	1. Parcours branche gauche 2. Parcours branche droite 3. Visite du nœud

Parcours préfixe	Parcours infixe	Parcours postfixe
		
5 - 10 - 3 - 6 - 1 - 9 - 4 - 12	10 - 3 - 5 - 9 - 1 - 4 - 6 - 12	3 - 10 - 9 - 4 - 1 - 12 - 6 - 5

Les implémentations récursives de ces trois types de parcours en profondeur sont très semblables, seul l'ordre des instructions change : la ligne `parcours.append(self.valeur)` est placée avant, au milieu ou après les appels récursifs des parcours des fils gauche et droit :

### Parcours préfixe

```

1 class AB :
2     def parcours_prefixe(self):
3         if self.racine is None:
4             return []
5         else:
6             return self.racine.parcours_prefixe()
7
8 class Noeud:
9     def parcours_prefixe(self, parcours = None):
10        # Premier appel de la méthode sans renseigner parcours
11        if parcours is None:
12            parcours = []
13
14        # Ajout de la valeur avant les fils gauche et droit
15        parcours.append(self.valeur)
16
17        # Parcours du fils gauche
18        if self.gauche is not None:
19            self.gauche.parcours_prefixe(parcours)
20
21        # Parcours du fils droit
22        if self.droite is not None:
23            self.droite.parcours_prefixe(parcours)
24
25 >>> arbre.parcours_prefixe()
26 [5, 10, 3, 6, 1, 9, 4, 12]
```

### Parcours infixe

```

1 class AB :
2     def parcours_infixe(self):
3         if self.racine is None:
```

```

4         return []
5     else:
6         return self.racine.parcours_infixe()
7
8 class Noeud:
9     def parcours_infixe(self, parcours = None):
10         # Premier appel de la méthode sans renseigner parcours
11         if parcours is None:
12             parcours = []
13
14         # Parcours du fils gauche
15         if self.gauche is not None:
16             self.gauche.parcours_infixe(parcours)
17
18         # Ajout de la valeur après le fils gauche et avant le droit
19         parcours.append(self.valeur)
20
21         # Parcours du fils droit
22         if self.droite is not None:
23             self.droite.parcours_infixe(parcours)
24
25         return parcours
26
27 >>> arbre.parcours_infixe()
28 [10, 3, 5, 9, 1, 4, 6, 12]

```

### Parcours postfixe

```

1 class AB :
2     def parcours_postfixe(self):
3         if self.racine is None:
4             return []
5         else:
6             return self.racine.parcours_postfixe()
7
8 class Noeud:
9     def parcours_postfixe(self, parcours = None):
10         # Premier appel de la méthode sans renseigner parcours
11         if parcours is None:
12             parcours = []
13
14         # Parcours du fils gauche
15         if self.gauche is not None:
16             self.gauche.parcours_postfixe(parcours)
17
18         # Parcours du fils droit
19         if self.droite is not None:
20             self.droite.parcours_postfixe(parcours)
21
22         # Ajout de la valeur après ses fils gauche et droit
23         parcours.append(self.valeur)
24
25         return parcours
26
27
28 >>> arbre.parcours_postfixe()
29 [3, 10, 9, 4, 1, 12, 6, 5]

```

Noter qu'il est aussi possible d'implémenter le parcours **préfixe** de façon itérative, semblable au parcours en largeur, à la différence qu'on utilise une pile plutôt qu'une file<sup>2</sup>.

Le parcours en largeur a un coût proportionnel à la taille du nombre de nœuds  $n$  de l'arbre (pour chaque nœud supplémentaire, on rajoute 4 opérations). La **complexité du calcul de la hauteur est en  $O(n)$** .

## Algorithmes sur les arbres binaires de recherche

Les algorithmes de calculs de taille, de hauteur et de parcours d'arbres s'appliquent à tous les arbres binaires. Dans la suite de ce chapitre, les algorithmes s'appliquent à des arbres binaires particuliers : les arbres binaires de recherche.

« supérieur » et « inférieur » peuvent être au sens strict ou large en fonction de la définition donnée.

### Cours

Un **arbre binaire de recherche (ABR)** est un cas particulier d'arbre binaire où :

- Chaque nœud a une **clé<sup>4</sup> supérieure** à celles de tous les nœuds de son **sous-arbre gauche**.
- Chaque nœud a une **clé inférieure** à celles de tous les nœuds de son **sous-arbre droit**.

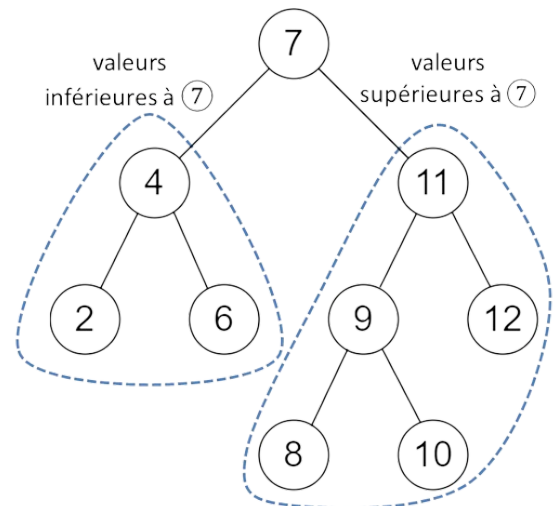
Tous les **sous-arbres sont aussi des ABR**.

On considère l'ABR suivant qui servira d'exemple pour la suite :

La classe `ABR` est une sous classe d' `AB` qui hérite<sup>3</sup> des méthodes d'un arbre binaire et permet d'ajouter ses propres méthodes.

```
class ABR(AB):
    pass

arbre_bin = ABR(Noeud(7,
    Noeud(4,
        Noeud(2),
        Noeud(6)),
    Noeud(11,
        Noeud(9,
            Noeud(8),
            Noeud(10)),
        Noeud(12))))
```



Les méthodes de la classe `AB` fonctionnent par héritage, en particulier le parcours infixe donne un résultat notable :

```
>>> arbre_bin.taille()
9
>>> arbre_bin.hauteur()
3

>>> arbre_bin.parcours_infixe()
[2, 4, 6, 7, 8, 9, 10, 11, 12]
```



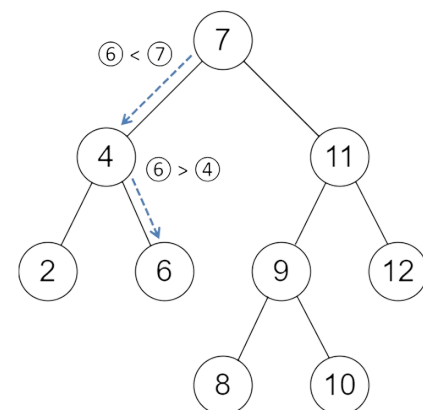
## Cours

Le **parcours infixe** d'un ABR renvoie les valeurs des nœuds dans l'ordre croissant.

### Rechercher une clé

Comme son nom l'indique, un ABR est spécifiquement adapté pour rechercher rapidement une clé. On parcourt l'ABR en partant de sa racine, et on observe trois cas :

- si la valeur recherchée est plus grande que la valeur du nœud parcouru, on descend à droite ;
- si la valeur recherchée est plus petite que la valeur du nœud parcouru, on descend à gauche ;
- si c'est la même, la clé a été trouvée.



Une fois arrivé à une feuille, la clé n'a pas été trouvée. L'exemple ci-contre montre la recherche de la valeur 6 dans l'arbre précédent.

Modifions la classe `ABR`, pour gérer immédiatement le cas de l'arbre vide (la valeur n'est pas trouvée) et appeler une méthode de la classe `Noeud`.

```
class ABR(AB):
    def recherche(self, v):
        """ Renvoie True si v est une valeur de l'arbre """
        if self.racine is None:
            return False
        return self.racine.recherche(v)
```

Voici ce que cela donne en récursif pour la classe `Noeud` :

```
class Noeud:
    def chercher(self, v):
        if v < self.valeur:      # On continue à chercher à gauche s'il y a un fils gauche
            if self.gauche is None:
                return False
            else:
                return self.gauche.chercher(v)
        elif v > self.valeur:    # On continue à chercher à gauche s'il y a un fils droit
            if self.droite is None:
                return False
            else:
                return self.droite.chercher(v)
        else:                    # On a trouvé v
            return True
```

ou en itératif :

```
def chercher(self, v):
    """ Renvoie le Noeud de valeur v
        None si aucun noeud n'a cette valeur
    """
    n = self.racine
```

```

while n is not None:
    if v < n.valeur:      # On continue à chercher v à gauche
        n = n.gauche
    if v > n.valeur:      # On continue à chercher v à gauche
        n = n.droite
    else:                # On a trouvé v
        return False
# Si on arrive ici, c'est qu'on n'a pas trouvé v
return True

```

A chaque appel récursif, ou itération de la boucle `while`, le nombre de nœuds à parcourir est divisé par un facteur 2 (dans un arbre équilibré), on en déduit que la **complexité de la recherche d'une clé est en  $O(\log_2(n))$** .

## Insérer une clé

### Cours

Lorsqu'on ajoute un nœud à un ABR, on **ajoute toujours une feuille de l'arbre, jamais un nœud interne**.

Il suffit donc de rechercher la feuille dont la clé est la plus proche de celle du nœud à insérer (comme pour l'algorithme de recherche), puis d'ajouter le nœud à gauche si sa clé est inférieure à celle de la feuille, et à droite dans le cas contraire. L'exemple ci-contre montre l'insertion de la valeur 5.

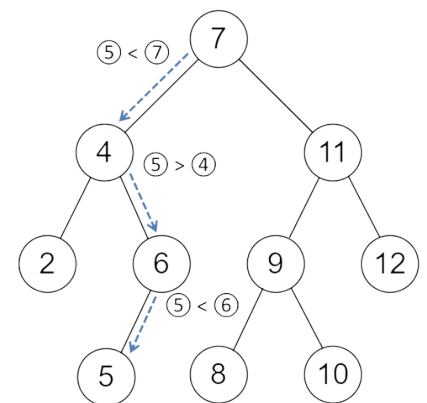
Dans le cas d'un ABR qui n'autorise pas les clés multiples, il faut empêcher l'insertion d'un nœud dont la clé est déjà dans l'arbre.

Modifions la classe `ABR`, pour gérer immédiatement le cas de l'arbre vide (on ajoute une racine) et appeler une méthode de la classe `Noeud`.

```

class ABR(AB):
    def inserer(self, v):
        """ insere la valeur v dans l'arbre """
        # si l'arbre est vide, on ajoute une racine
        if self.racine is None:
            self.racine = Noeud(v)
        else:
            self.racine.inserer(v)

```



Voici ce que cela donne en récursif pour la classe `Noeud` :

```

class Noeud:
    def inserer(self, v):
        """insere la valeur v dans l'ABR"""
        if v < self.valeur:
            if self.gauche is None:
                self.gauche = Noeud(v)      # on ajoute v à gauche
            else:
                self.gauche.inserer(v)      # on descend à gauche
        if v > self.valeur:
            if self.droite is None:
                self.droite = Noeud(v)      # on ajoute v à droite
            else:
                self.droite.inserer(v)      # on descend à droite

```

```
if v == self.valeur:
    print(v, "est déjà dans l'ABR")    # la valeur est déjà dans l'ABR
```

ou en itératif :

```
def inserer(self, v):
    """ Insère un nouveau Noeud de valeur v dans l'arbre """
    if self.racine is None:
        self.racine = Noeud(v)
    else:
        n = self.racine
        while n is not None:
            if v < n.valeur:
                if n.gauche is None:
                    n.gauche = Noeud(v)
                    return
                n = n.gauche
            elif v > n.valeur:
                if n.droite is None:
                    n.droite = Noeud(v)
                    return
                n = n.droite
            else:
                return # clé déjà dans l'arbre
```

La complexité de l'insertion d'une clé est en  $O(\log_2(n))$ .

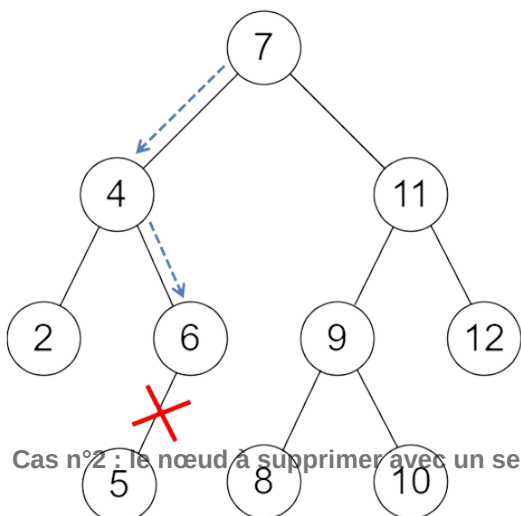
## Supprimer une clé (hors programme)

L'opération dépend du nombre de fils du nœud à supprimer.

### Cas n°1 : le nœud à supprimer est une feuille

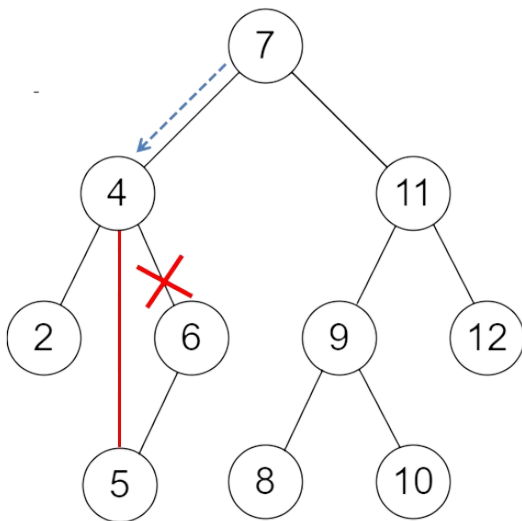
- On cherche le père du nœud à supprimer ;
- On supprime le lien père-fils.

Exemple : suppression du nœud 5



- On recherche le père du nœud à supprimer ;
- On redirige le lien père-fils vers le fils (unique) du nœud à supprimer.

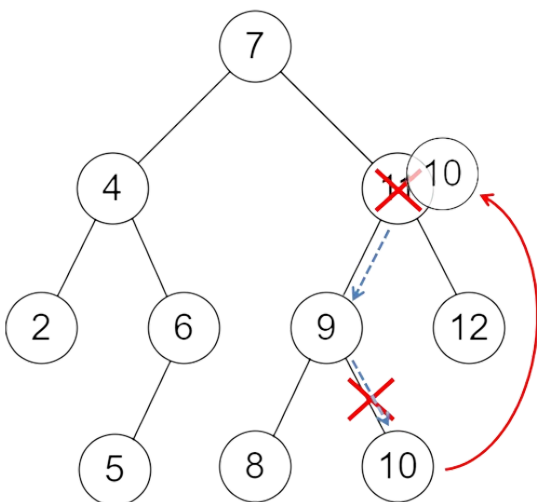
Exemple : suppression du nœud 6



### Cas n°3 : le nœud à supprimer avec deux fils

- On recherche le nœud de sa branche de gauche ayant la plus grande valeur, comme c'est le plus grand de cette branche, il n'a pas de fils droit (voir valeur max) ;
- On rompt le lien père-fils, (comme dans le cas n°2 ;) )
- Et on remplace la valeur du nœud à supprimer par celle du nœud décroché.

Exemple : suppression du nœud 11



Remarque : alternativement, on peut décrocher le nœud ayant la plus petite valeur de sa branche de droite.

```
def supprimer(self, v):
    """ Supprime le Nœud de valeur v dans l'arbre
        (s'il existe)
    """
    # on recherche le nœud à supprimer n et son pere p
```

```

p= None
n = self.racine
while n is not None:
    if v > n.valeur:
        p = n
        n = n.droite
    elif v < n.valeur:
        p = n
        n = n.gauche
    else:
        break
if n is None: return -1 # la valeur n'existe pas
fils = 0
if n.gauche is not None: fils += 1
if n.droite is not None: fils += 1

print(fils)
print(p.valeur)

if n.gauche is None and n.droite is None: # n est une feuille
    if p.droite == n: p.droite = None
    else: p.gauche = None
elif n.gauche is not None and n.droite is None: # 1 seul fils à gauche
    if p.droite == n: p.droite = n.gauche
    else: p.gauche = n.gauche
elif n.gauche is None and n.droite is not None: # 1 seul fils à droite
    if p.droite == n: p.droite = n.droite
    else: p.gauche = n.droite
else: # 2 fils
    # on cherche le max et son père
    max = n.gauche
    p = n
    while max.droite is not None:
        p = max
        max = max.droite
    p.droite = max.gauche # on rattache la gauche du max à p
    n.valeur = max.valeur # on remplace la valeur de n par le max

```

1. D'autres algorithmes existent, par exemple la recherche arborescente Monte-Carlo. ←

2.

```

class Noeud:
def parcours_prefixe(self):
    parcours = []
    pile = []
    pile.append(self)
    while len(pile) != 0: # tant que la pile n'est pas vide
        n = pile.pop() # on prend le noeud au sommet de la pile
        parcours.append(n.valeur) # on note sa valeur
        if n.droite is not None: pile.append(n.droite) #on empile le fils DROIT d'abord
        if n.gauche is not None: pile.append(n.gauche) #on empile le fils gauche

```

←|

3. L'héritage est un des grands principes de la programmation orientée objet (POO) permettant de créer une nouvelle classe à partir d'une classe existante. La sous-classe hérite des attributs et des méthodes de la classe mère et en ajoute de nouveaux. ←

4. Dans un ABR, l'étiquette est appelée « clé » ou « valeur ». ←