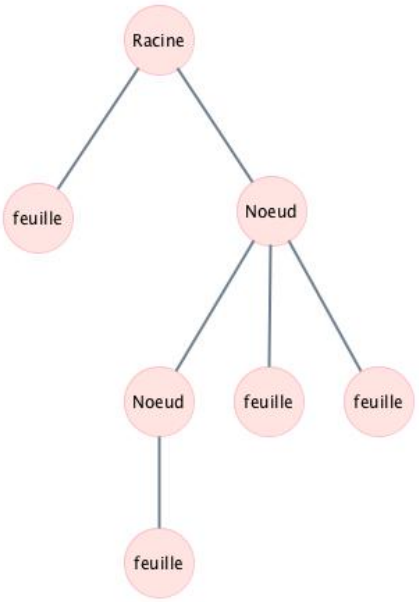
Structures de données hiérarchiques : les arbres

1. Définitions

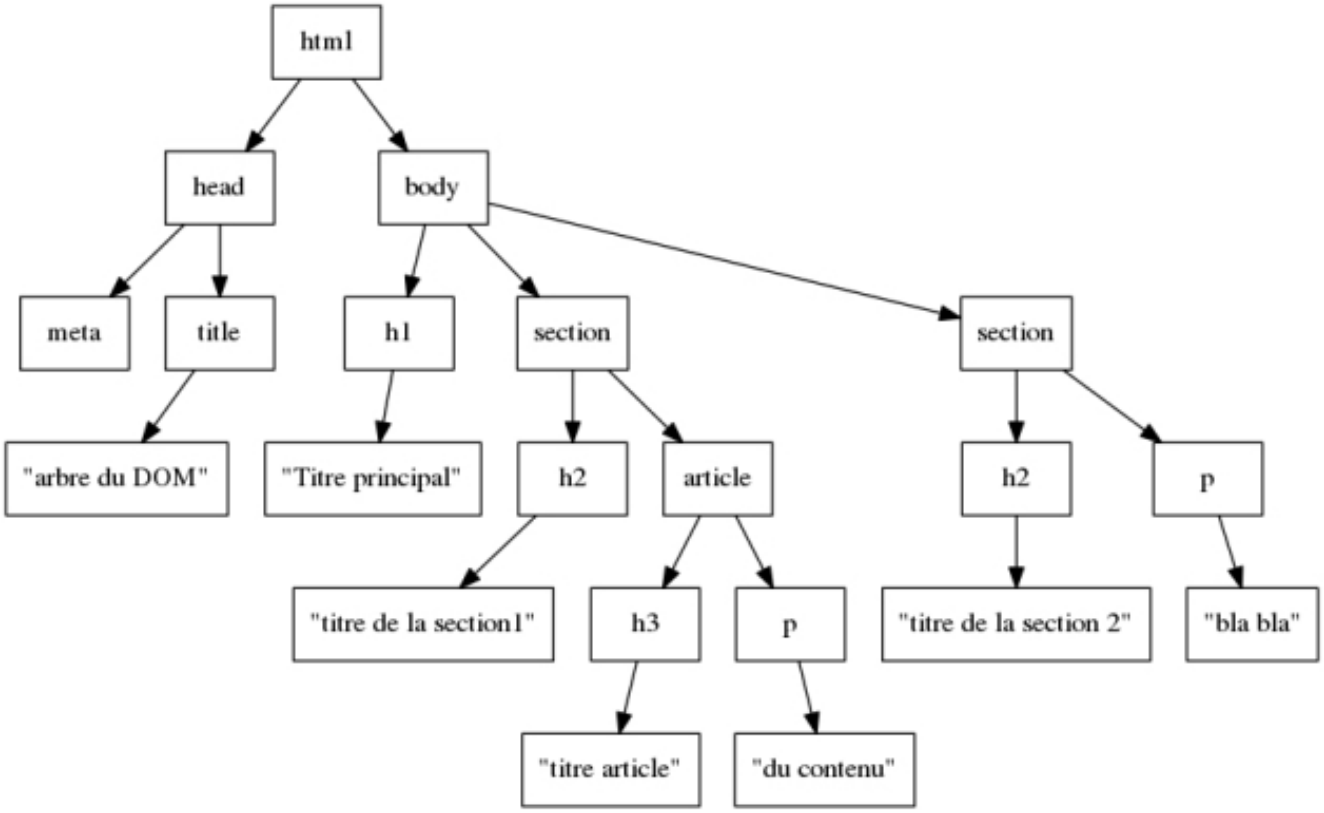
Un arbre en informatique est un type particulier de graphe. Comme nous n'avons pas encore vu à ce stade de l'année les graphes, cette définition n'est pas d'une grande utilité...
On dira donc qu'un arbre est constitué :

- d'une racine, sommet de "départ" de l'arbre ;
- de nœuds , sommets intermédiaires de l'arbre ;
- de feuilles, sommets "finaux" de l'arbre ;
- et de branches, qui relient les éléments précédents entre eux.

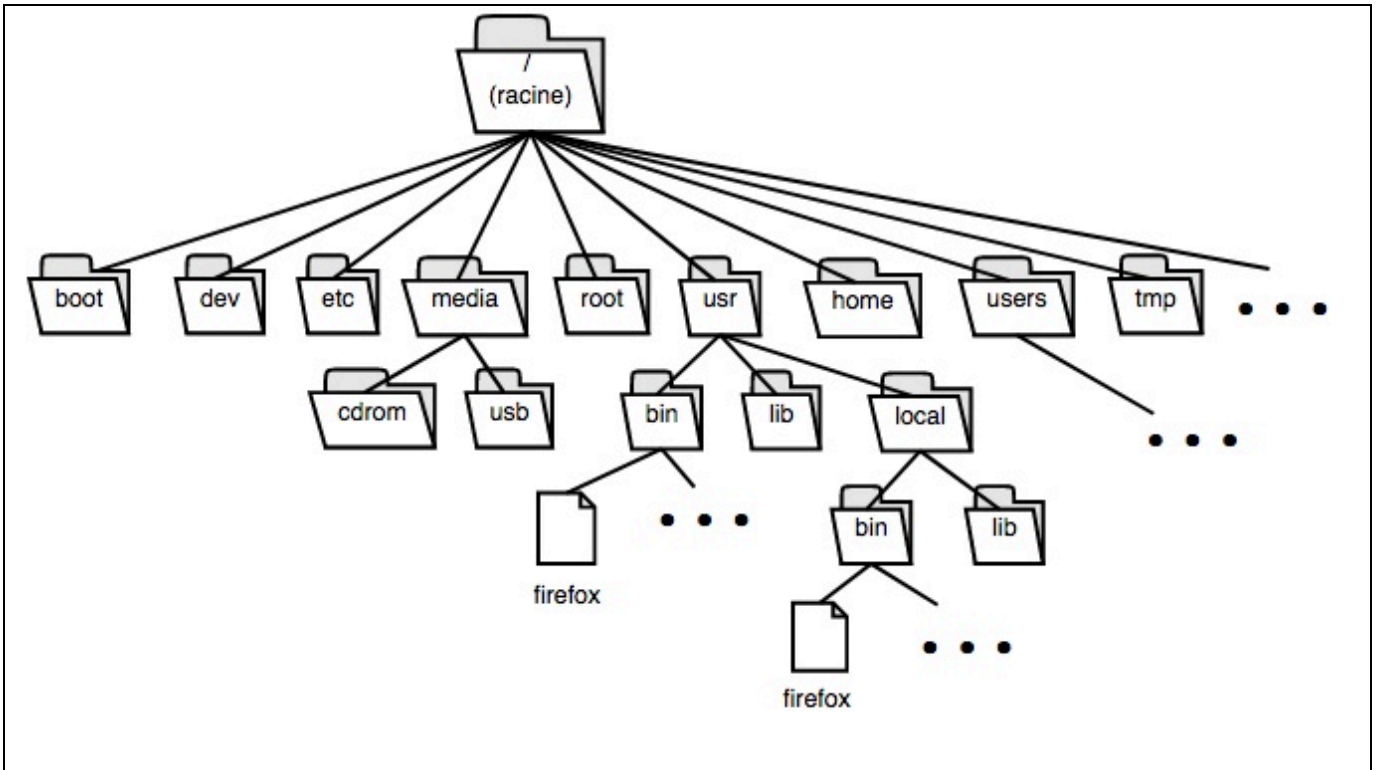


Les arbres informatiques ont ceci de particulier qu'ils poussent tête en bas (ils viennent de l'hémisphère sud). Les arbres sont des structures de données hiérarchiques, très utilisées en informatique. Ils sont orientés : la représentation standardisée, racine en haut, indique la relation "père-fils" entre les sommets. Lorsque deux sommets sont reliés par une branche, celui du haut est le père, et celui du bas est le fils. Un père peut avoir plusieurs fils, mais un fils ne peut pas avoir plusieurs pères (faites un dessin, on obtiendrait alors ce qu'on appelle un cycle dans le vocabulaire des graphes).

L'arborescence des fichiers ou d'un document html donne des exemples d'arbres en informatique.



Une arborescence html

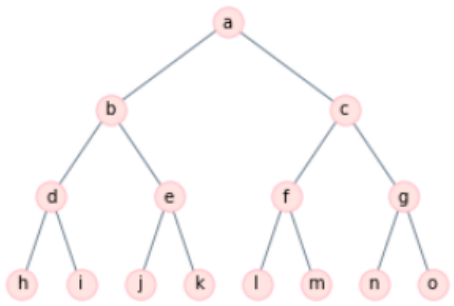
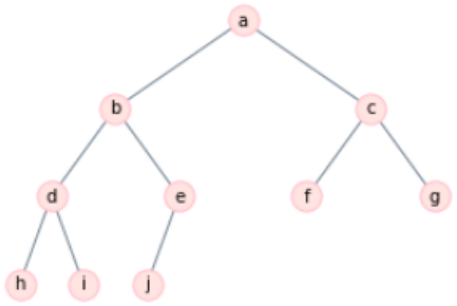


Une arborescence de fichiers

Définitions complémentaires
Les nœuds sont en général étiquetés, ci-contre les **étiquettes** sont les lettres a, b, c, etc…
La **taille** d'un arbre est le nombre de ses nœuds. Dans l'exemple ci-contre, la taille de l'arbre est 7.
La **hauteur** (ou **profondeur** ou **niveau**) d'un nœud X est définie comme le nombre de nœud à parcourir pour aller de la racine au nœud X. La hauteur de la racine est arbitrairement fixée à 0, (ou 1 suivant les définitions/sujets du bac). L'arbre vide a donc comme hauteur -1.
La **hauteur** de l'arbre est la plus grande des hauteurs de ses nœuds.
Exemple : le nœud ⓒ a pour hauteur 2, le nœud ⓕ a pour hauteur 3, le nœud ⓖ a pour hauteur 4 (avec 1 comme hauteur pour a). la hauteur de l'arbre est égale à la hauteur du nœud ⓖ, soit 4.
L'**arité** d'un nœud est le nombre de ses fils
Exemple : ⓐ a pour arité 2, ⓑ a pour arité 0, ⓒ a pour arité 3

Un arbre peut être défini de manière récursive : un arbre est soit un arbre vide, soit une racine et une liste de sous-arbres (éventuellement vide, auquel cas c'est une feuille). *C'est la définition qui est probablement la plus pertinente pour la vision « informatique » des arbres.*

En terminale, nous étudierons principalement les **arbres binaires** : un arbre binaire est un arbre dont tous les nœuds sont d'arité au maximum 2. Autrement fit, chaque père a au plus deux fils, appelés sous-arbre gauche et sous-arbre droit.
Les arbres binaires se rencontrent par exemple dans les compétitions sportives comme un tournoi de tennis.
Trois cas particuliers :

Arbre dégénéré ou filiforme	 Arbre parfait	 Arbre complet ou presque complet : c'est un arbre dont tous les niveaux sont complètement remplis, sauf éventuellement le dernier. Les feuilles du dernier niveau sont le plus à gauche possible

A noter que ces définitions varient suivant les auteurs et les livres… notamment « parfait » et « complet » peuvent être intervertis !

Lien entre taille et hauteur. Si un arbre binaire est de taille n et de hauteur h, alors :

- le nombre de sommets est au moins égal à la hauteur + 1 , et strictement inférieur à $2^{\text{hauteur}+1}$:

$$h+1 \leq n < 2^{h+1}$$

- De manière équivalente, la hauteur est strictement plus grand que le logarithme en base deux de la taille, et inférieur ou égal à la taille :

$$\log_2 n - 1 < h \leq n - 1$$

2. Une classe arbre binaire en Python

Le but est de créer une structure de données pour représenter un arbre binaire en Python. Les opérations sur les arbres binaires sont au minimum :

- Construction d'arbre vide
- Construction d'un arbre à partir d'un entier et de deux sous-arbres gauche et droit
- Test de vacuité
- Accès à la racine d'un arbre
- Accès au sous-arbre gauche
- Accès au sous-arbre droit

a. Première implémentation

On crée une classe nœud, puis on utilise ce nœud pour construire un arbre

```
class Noeud:
    def __init__(self,valeur,gauche,droit):
        self.n = valeur
        self.g = gauche
        self.d = droit

class ArbreBinaire:
    def __init__(self,c):
        self.r = c

    def creeVide():
        return ArbreBinaire(None)

    def creeNGD(valeur, gauche = None, droit = None):
        return ArbreBinaire(Noeud(valeur, gauche, droit))

    def estVide(self):
        return self.r is None
```

```
    def racine(self):
        assert not(self.r is None),'Arbre vide'
        # Deuxième version
        # if self.estVide():
        #     Raise index error('Arbre vide')
        # else :
        return self.r.n

    def filsGauche(self):
        assert not(self.r is None),'Arbre vide'
        return self.r.g

    def filsDroit(self):
        assert not(self.r is None),'Arbre vide'
        return self.r.d
```

b. Deuxième implémentation

Comme vu dans le notebook, la classe nœud suffit à créer un arbre, qui n'est rien d'autre qu'une suite récursive de nœuds. La classe ArbreBinaire du paragraphe précédent n'est donc pas obligatoire.

3. Algorithmes sur les arbres

Ces algorithmes sont à comprendre plus qu'à retenir

a. Taille et hauteur

Calcul de la taille : retourne le nombre de sommets de l'arbre (racine + nœuds + feuilles)

Fonction récursive taille(*arbre*) :
  Si *arbre* est vide
    Retourner 0
  Sinon
    Retourner 1 + taille (*fils gauche*) + taille (*fils droit*)

Calcul de la hauteur : retourne la hauteur de l'arbre

Fonction récursive hauteur(*arbre*) :
  Si *arbre* est vide
    Retourner 0
  Sinon
    Retourner 1 + max(hauteur (*fils gauche*), hauteur(*fils droit*))

b. Parcours en profondeur

Dans le parcours en profondeur, on parcourt d'abord la racine de l'arbre, puis récursivement les fils gauche et droit. L'ordre dans lequel est fait ce traitement donne les trois parcours possibles :

i. Parcours préfixe : racine – gauche – droit (la racine avant les enfants, la racine *pré*cèdent)
ii. Parcours infixe : gauche – racine – droit (la racine est au milieu, « *in* »)
iii. Parcours suffixe : gauche – droit – racine (la racine est en dernier, la racine *su*ccède)

Préfixe	Infixe	Suffixe
Fonction visitePréfixe(arbre) : **Si** arbre n'est pas vide : visiter racine visitePréfixe (fils gauche) visitePréfixe (fils droit)	**Fonction** visiteInfixe(arbre) : **Si** arbre n'est pas vide : visiteInfixe (fils gauche) visiter racine visiteInfixe (fils droit)	**Fonction** visiteSuffixe(arbre) : **Si** arbre n'est pas vide : visiteSuffixe (fils gauche) visiteSuffixe (fils droit) visiter racine

c. Parcours en largeur

Principe : on parcourt tous les nœuds de hauteur 1 (la racine), puis tous les nœuds de hauteur 2, ceux de hauteur 3 etc. Le parcours se fait en général de gauche à droite.
On utilise pour cela une file

Étapes de l'algorithme :

1 . Mettre le nœud source dans la file.
2. Retirer le nœud du début de la file pour le traiter.
3. Mettre le fils gauche et le fils droits lorsqu'ils sont non vides, non explorés, à la fin de la file.
4. Si la file n'est pas vide reprendre à l'étape 2.

En langage naturel :

```
Fonction parcours_largeur(arbre)
    f = créer_file_vide()
    enfiler(f, racine(arbre))
    tant que f est non vide :
        noeud = defiler(f)
        afficher noeud
        si fils_gauche(noeud) est non vide :
            enfiler fils_gauche(noeud)
        si fils_droit(noeud) est non vide :
            enfiler fils_droit(noeud)
```

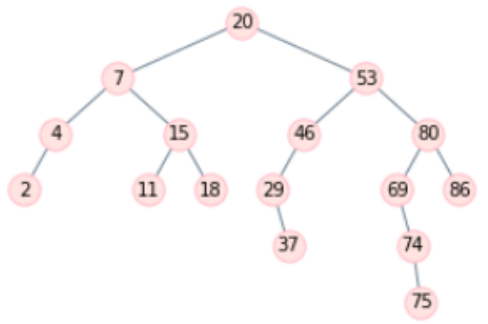
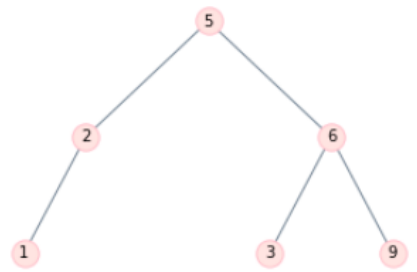
4. Arbres binaires de recherche (ABR)

a. Définition

Un arbre binaire de recherche, ou ABR, est un arbre binaire étiqueté possédant la propriété suivante : Pour tout nœud *x*, tous les nœuds situés dans le sous-arbre gauche de *x* ont une valeur inférieur ou égale à celle de *x*, et tous les nœuds situés dans le sous-arbre droit ont une valeur supérieure ou égale à celle de *x*.

Les arbres binaires de recherche servent, comme leur nom l'indique, à rechercher rapidement des éléments ordonnés.

Exemples :

	
Est un arbre binaire de recherche	N'est pas un arbre binaire de recherche

b. Algorithmes sur les ABR

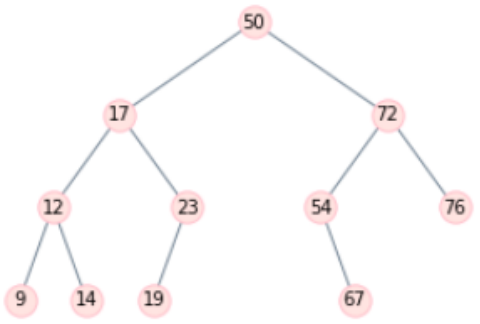
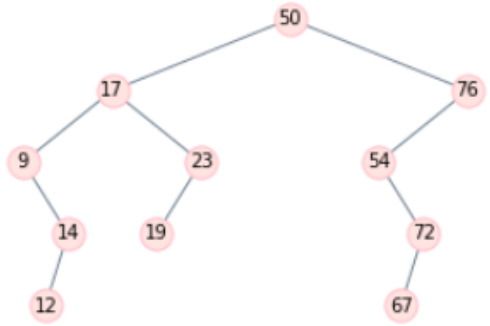
i. Recherche dans un ABR

Principe : suivant la valeur à rechercher, et la valeur du nœud sur lequel on est, on cherche soit dans le sous-arbre gauche, soit dans le sous-arbre droit

Fonction récursive recherche(*ABR, valeur*) :
    Si *ABR* est vide
        Retourner None # variante : retourner Faux
    Sinon
        valeur(x) = étiquette(*ABR*) # valeur de la racine
        Si *valeur* < *valeur(x)* :
            Retourner recherche(*fils_gauche , valeur*)
        Sinon si *valeur* > *valeur(x)* :
            Retourner recherche(*fils_droit , valeur*)

Sinon
Retourner ABR # variante : retourner Vrai

Pseudo-définition (peu rigoureuse et incomplète) : un arbre est équilibré si les hauteurs entre les sous-arbres gauche et droit sont différentes de 1 au maximum. Les arbres complets ou parfaits sont équilibrés.

	
Arbre binaire de recherche équilibré (tout en étant ni parfait, ni complet)	Arbre binaire de recherche déséquilibré. Cet arbre contient les mêmes données que celui de gauche.

Complexité :

Si l'arbre binaire de recherche est équilibré, alors le coût en temps de la recherche est en $O(\log n)$, où n est le nombre de nœuds de l'arbre.

Dans le cas ou l'arbre binaire de recherche n'est pas équilibré, la recherche est en $O(n)$. Le cas le pire étant le cas ou l'ABR est filiforme.

Remarque : on peut écrire une version itérative de cet algorithme (exercice intéressant)

ii. Insertion dans un ABR

Le problème de l'insertion dans un ABR correspond à celui de la construction de l'ABR. L'insertion dans un ABR commence par la recherche de l'endroit où l'on doit insérer la valeur. Si la valeur à insérer est plus petite –ou égale– que la valeur du nœud, on va à gauche, si elle est plus grande on va à droite. On arrive à un moment à un arbre vide : c'est là où on ajoute la nouvelle valeur. La gestion des arbres vides n'est cependant pas si simple dans cette approche.

Fonction récursive ajoute(*ABR, valeur*) :

Si *ABR* est vide
renvoyer Arbre(valeur, None, None)
Sinon si *valeur* <= étiquette_noeud(*ABR*)
renvoyer Arbre(étiquette_noeud(*ABR*), ajoute(*fils_gauche , valeur*), *fils_droit*)
Sinon
renvoyer Arbre(étiquette_noeud(*ABR*), *fils_gauche ,* ajoute(*fils_droit , valeur*))

Complexité : la complexité est identique à celle de la fonction de recherche. Dans le cas d'un arbre équilibré, elle est en $O(\log n)$.

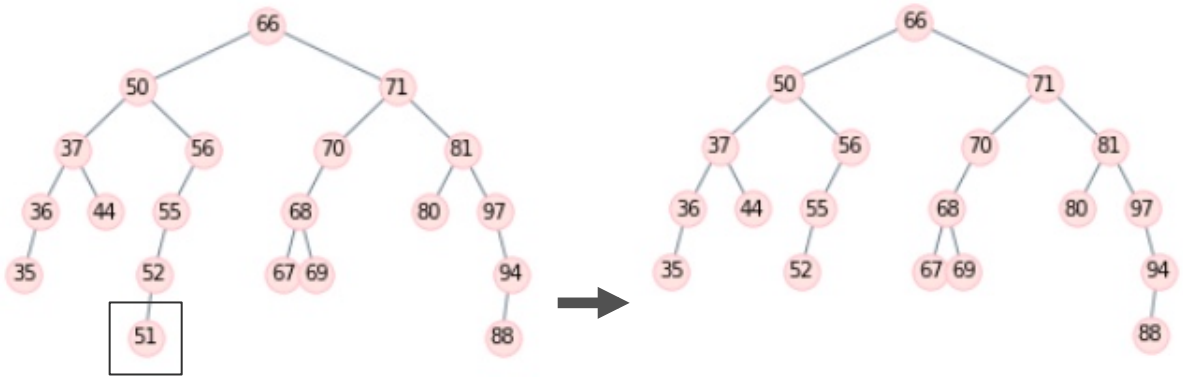
Remarque : cette méthode ne donne pas forcément des arbres équilibrés. Pour avoir un arbre équilibré, une méthode efficace est d'insérer les éléments dans un ordre aléatoire.

iii. Suppression dans un ABR

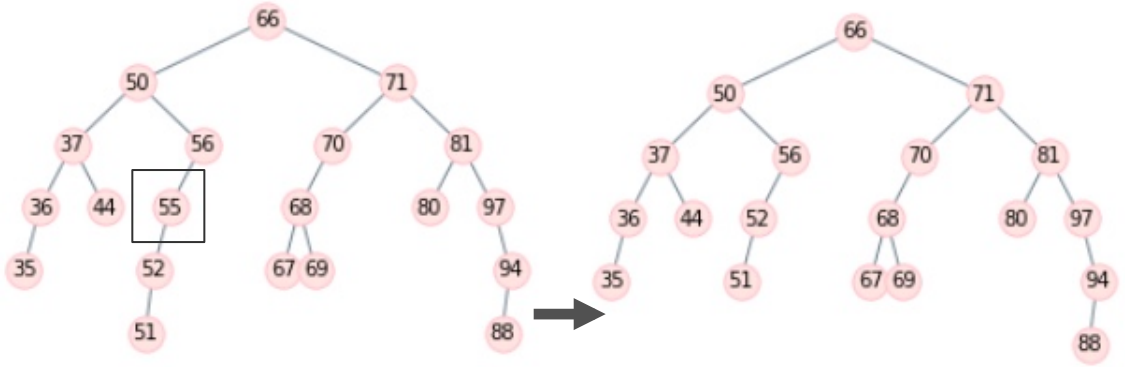
Cette partie n'est pas au programme du bac. Elle constitue un complément enrichissant en vitamines pour les neurones.

Le problème se décompose en trois cas, suivant le nombre de fils du nœud à supprimer.

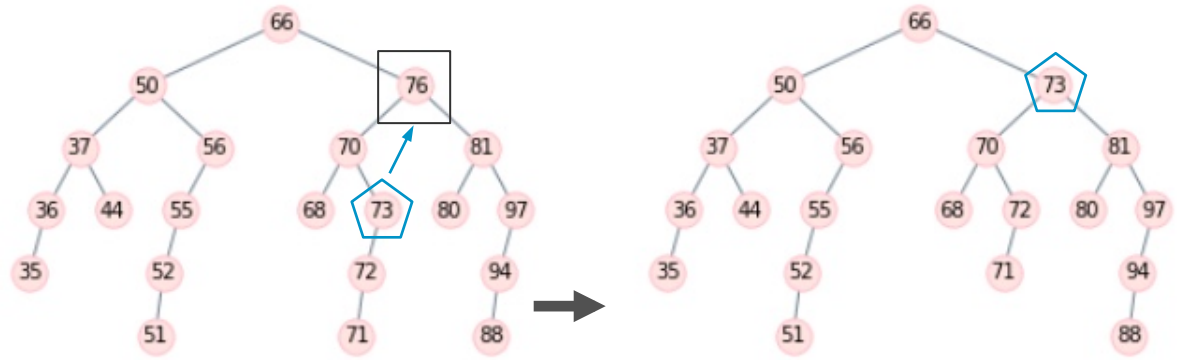
Si le nœud est une feuille (pas de fils), alors on supprime simplement le lien du père vers ce nœud (on *décroche* le nœud). Si ce lien n'existe pas, l'arbre devient l'arbre vide.



Si le nœud a un seul fils, alors on décroche le nœud comme précédemment, et on remplace son fils dans le nœud père. Si le père n'existe pas, le nœud fils devient la racine de l'arbre.



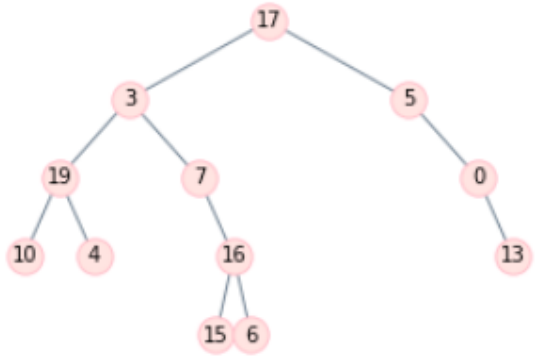
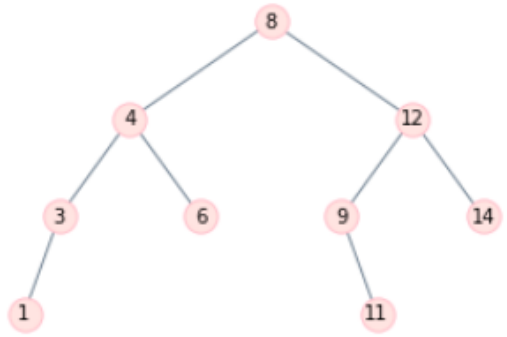
Si le nœud a deux fils, on cherche dans le sous-arbre gauche le nœud *MaxLocal* de valeur la plus grande (ou le nœud de valeur la plus petite dans le sous-arbre droit). La valeur de ce nœud va remplacer la valeur supprimée. Comme ce nœud *MaxLocal* a la valeur maximale dans le sous-arbre gauche, il n'a pas de fils droit. On peut donc le décrocher, comme on l'a fait dans le cas précédent.



5. Exercices arbres binaires

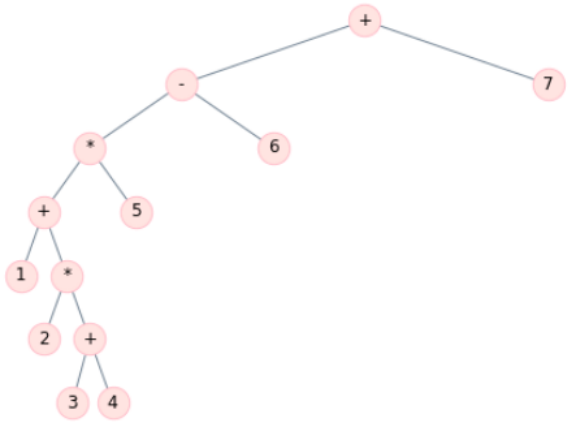
a. p 155 du livre
   Exercices 70, 71, 75, 78

b. Donner les parcours préfixe, infixe et suffixe de l'arbre 1 ci-dessous. Vous pouvez aussi vous entraîner avec l'arbre 2.

	
Arbre 1	Arbre 2

c. Donner un arbre binaire de hauteur aussi petite que possible, dont le parcours infixe affiche
6 3 2 7 4 8 5 0 1 9
Même question avec les parcours préfixe, suffixe et en largeur.
Y-a-t-il une seule solution à chaque fois ?

d. Retour sur la notation polonaise inversée (RPN)
L'arbre binaire ci-contre représente un calcul.
En rajoutant des parenthèses autour de chaque groupe « opérateur et opérandes », donner l'écriture du calcul correspondant.
Un des parcours (préfixe, infixe, suffixe) correspond à notre notation usuelle, un autre à la notation polonaise inversée. Identifier ces parcours.
Construire l'arbre correspondant au calcul en RPN :
2 4 3 * + 7 * 8 + 2 3 5 + * -



e. Étant donné deux parcours d'un arbre, on ne peut pas toujours reconstruire l'arbre de manière unique. Trouver des contre-exemples simples dans les cas où cela n'est pas possible. Dans les cas où c'est possible, écrire les algorithmes.

6. Exercices ABR

a. Qu'affiche le parcours infixe d'un ABR ?
b. Donner tous les ABR formés de trois nœuds et contenant les nombres 1, 2 et 3.

7. Pour la culture générale : utilité des arbres

- Jeux vidéos 3D : utilisés pour savoir quels objets doivent être représentés à l'écran (arbre de partition spatiale)
- Tables de routage. Permet au routeur de savoir où envoyer l'information sur internet
- Compression (codage de Huffman) pour le jpeg et le mp3
- Arbres syntaxiques : pour la compilation de programmes
- Certaines bases de données
- Arbres binaires de recherche :
  - Les arbres binaires de recherche équilibrés permettent une recherche en temps $O(\log n)$. C'est plus rapide que dans une liste, en $O(n)$, et moins rapide que dans un dictionnaire, en $O(1)$.
  - L'avantage par rapport au dictionnaire est la facilité de faire des calculs. Dans un dictionnaire, trouver la clé minimale est en $O(n)$, avoir les clés triées dans l'ordre en $O(n \log n)$, alors que ces opérations restent en $O(\log n)$ pour un arbre de recherche équilibré.

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.