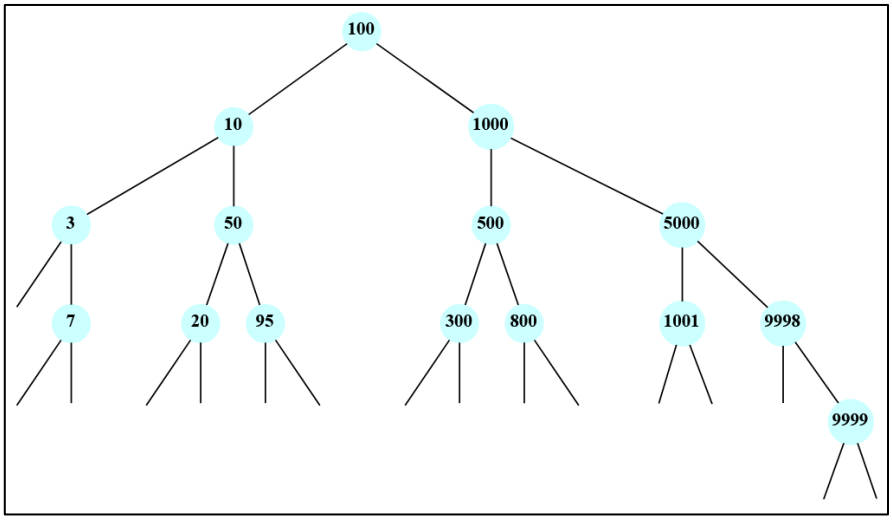
Arbres binaires de recherche : exercices

Exercice 1 :

On considère l'arbre binaire de recherche ci-contre.



1) On décide d'y insérer, avec l'algorithme vu en cours, la valeur 29 : où sera-t-elle insérée ?
   À gauche de 100.
   À droite de 10.
   À gauche de 50.
   À droite de 20.
   Dans un nœud qui sera fils droit de 20.

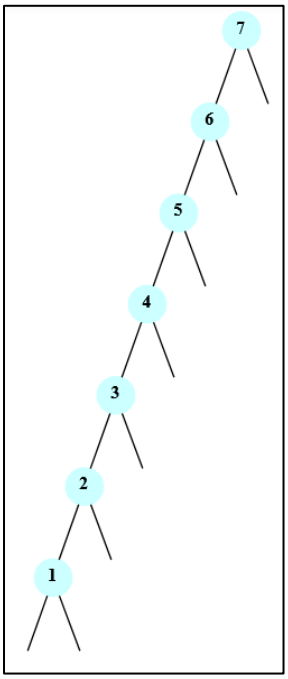
2) Même question avec 900.
   À droite de 100.
   À gauche de 1000.
   À droite de 500.
   À droite de 800.
   Dans un nœud qui sera fils droit de 800.

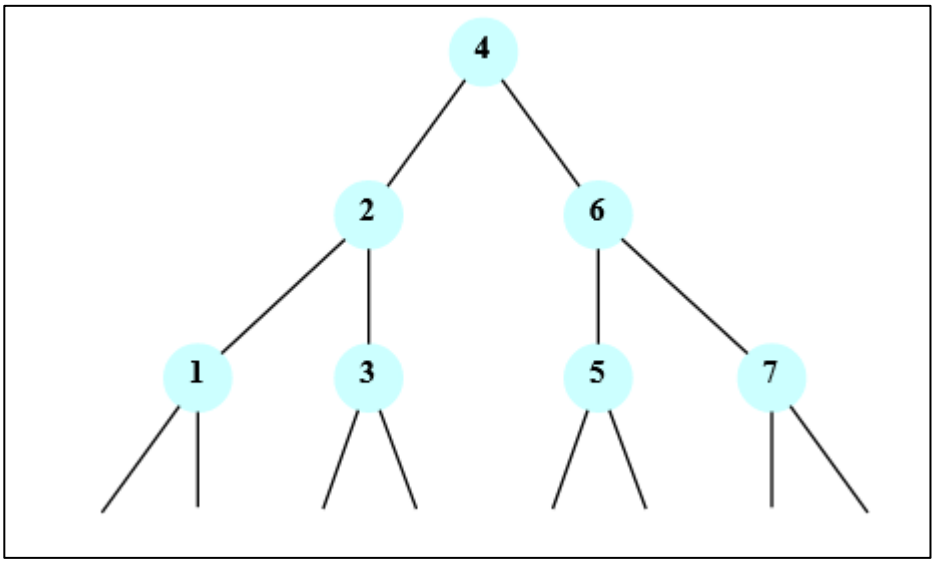
3) Même question avec 2 puis -1 puis -7 puis -12.
   2 sera dans un nœud fils gauche de 3.
   - 1 sera dans un nœud fils gauche de 2.
   -7 sera dans un nœud fils gauche de -1.
   - 12 sera dans un nœud fils gauche de -7.

4) Est-e que l'ordre dans lequel on insère les valeurs a une influence sur l'arbre binaire de recherche obtenu ?
   Oui

Exercice 2 :

Construire deux arbres binaires de recherche comportant les valeurs $1, 2, 3, 4, 5, 6$ et 7 : le premier de hauteur 7 et le second de hauteur 3.





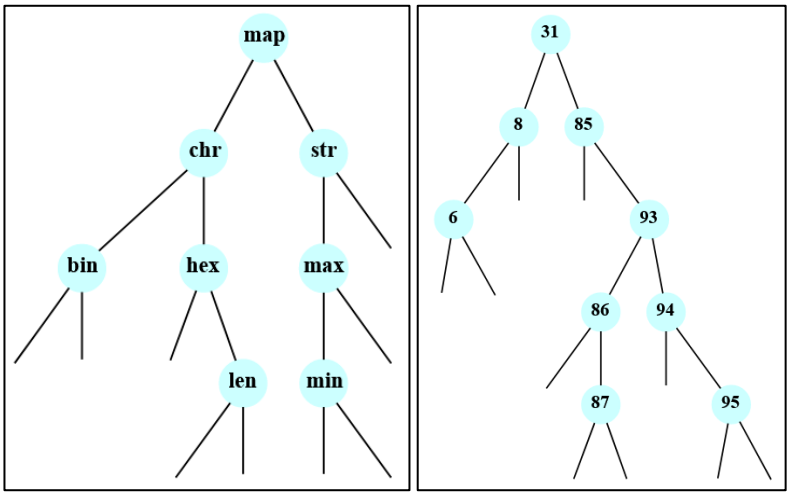
Exercice 3 :

1) Avec une complexité en $O(\log N)$, par combien est approximativement multipliée le temps d'exécution d'un algorithme lorsque la taille de l'entrée est multipliée par 1 000 000 ?
Par 20 si c'est log base 2 car $\log_2(1000\ 000) \approx 20$ ou par 6 si c'est log base 10 car $\log_{10}(1000\ 000) = 6$.

2) Et avec une complexité en $O(N)$?
Par 1000 000

3) Donner un exemple d'ABR A de taille 1000000 et d'un ABR B de taille 1000 tel que la recherche de l'élément 777 dans A est plus rapide que dans B.
Il suffit que dans l'ABR A l'élément 777 soit à la racine et pas dans l'ABR B.

Exercice 4 :



1) Les deux arbres binaires ci-contre sont-ils des ABR ?
Pour le premier non à cause de min (qu idevrait être fils droit de max)

Pour le second oui.

1) On parcourt l'arbre de gauche avec un parcours en largeur. Dans quel ordre les noeuds sont-ils parcourus ?
map, chr, str, bin, hex, max, len, min
Et avec un parcours infixe ?
bin, chr, hex, len, map, min, max, str

2) Même question pour l'arbre de droite.
31, 8, 85, 6, 93, 86, 94, 87, 95 en largeur
6, 8, 31, 85, 86, 87, 93, 94, 95 en parcours infixe (on remarque que c'est en ordre croissant)

3) Donner un algorithme de parcours d'arbre binaire de recherche tel que les nœuds sont parcourus dans l'ordre décroissant.
On prend le parcours infixe en visitant d'abord le sous-arbre droit, puis le nœud en cours, puis le sous-arbre gauche.

Exercice 5 :

Donner un algorithme permettant de renvoyer la valeur maximale d'un arbre binaire de recherche.
Même question avec la valeur minimale.
Pour la valeur maximale, on parcourt les fils droits jusqu'à arriver à un sous-rabre vide (en bas à droite de l'arbre")
Pour la valeur minimale, on fait la même chose avec les fils gauche (en bas à gauche de l'arbre)

Exercice 6 :

On suppose disposer d'une classe `ABR` disposant uniquement d'un attribut `_racine` ainsi que des méthodes `insere` et `recherche` (comme en TP). On suppose en outre que la méthode `insere` permet de construire un arbre *équilibré*.

1) L'utilisateur de la classe `ABR` a-t-il le droit d'utiliser l'attribut racine dans son code ? Peut-il le faire ?
Normalement non puisque le underscore _ est là pour dire que _racine est réservé pour l'implémentation de la classe et pas pour son utilisation (est réservé à l'nitérieur de la classe mais pas à l'extérieur).
Néanmoins en Python, cela ne protège rien et l'utilisateur peut tout de même utiliser _racine sans que Python ne provoque d'erreur.

2) Ecrire un algorithme `tab_2_abr` qui prend en argument un tableau d'entiers ou de chaînes de caractères et renvoie un arbre binaire de recherche comportant tous les éléments du tableau.

```
def tab_2_abr(tab):
    a = None
    for elt in tab:
        a = inserer(elt, a)
     return a
```

3) Ecrire un algorithme `abr_2_tab` qui prend en argument un arbre binaire de recherche et un tableau vide et mute le tableau pour qu'il contienne toutes les valeurs présentes dans l'ABR triées par ordre croissant (voir exercice 4 question 2 si besoin).
Il suffit de reprendre l'algorithme du parcours infixe :

```
def abr_2_tab(arbre, tab):
    if arbre != None:
        abr_2_tab(arbre.gauche, tab)
        tab.append(arbre.valeur)
        abr_2_tab(arbre.droit, tab)
```

Pour appeler cette fonction :

```
>>> tab_trie = []
>>> abr_2_tab(arbre_tab_trie)
```

4) En utilisant ce qui précède, montrer que l'on peut ainsi disposer d'une méthode de tri efficace. Quelle est sa complexité ?
On prend le tableau, on le transforme en arbre binaire de recherche puis on récupère le contenu de l'arbre binaire de recherche trié grâce au parcours infixe (on utilise tab_2_abr puis abr_2_tab).

Pour un tableau de taille N, on effectue N insertions ayant une complexité en O(h) où h est la hauteur de l'ABR.
Or si l'arbre est équilibré, h est équivalent à log(N) : on effectue N insertions de complexité au plus log(N). Soit du N log(N).

Pour le parcours infixe de l'ABR équilibré ainsi construit, la complexité est en O(N).

Au final, c'est la complexité N log(N) qui domine : le tri est en N log(N) ce qui est meilleur que du N^2 comme le tri par insertion ou le tri par sélection.

5) Vous implémentez cette méthode. Le tableau non trié est de taille $N = 3407$, le tableau trié est de taille $N' = 3341$. D'où peut venir le problème ?

Les doublons qui ont été oubliés par exemple ?