Nom : .....

Tri par arbre binaire de recherche

On rappelle qu'un arbre binaire de recherche est un arbre binaire (ici, étiqueté par des entiers) tel que chaque nœud est à la fois

- supérieur à tous les nœuds qui sont à sa gauche, et
- inférieur à tous les nœuds qui sont à sa droite.

Le but de cet exercice est d'étudier l'utilisation d'un arbre binaire de recherche (en abrégé ABR ou BST pour *Binary Search Tree*) pour trier une liste d'entiers supposés tous différents. Par exemple on considère la liste

6	3	1	7	5	9	8	2	4

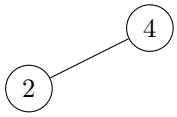
à trier dans l'ordre croissant. Pour cela on traite la liste comme une pile (c'est-à-dire qu'on n'utilise que la méthode `pop(self)` sur cette liste), et on utilise l'algorithme suivant pour construire un ABR étiqueté par les éléments de cette liste :

- Tant que la liste n'est pas vide, répéter
- ◦ dépiler un entier de la liste
  ◦ insérer cet élément dans l'ABR
- parcourir l'ABR pour reconstituer la liste triée.

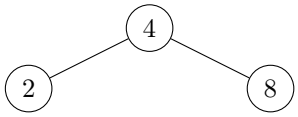
Avec la liste [6,3,1,7,5,9,8,2,4] on a les étapes suivantes de l'algorithme :

liste [6,3,1,7,5,9,8,2] (4 dépilé)

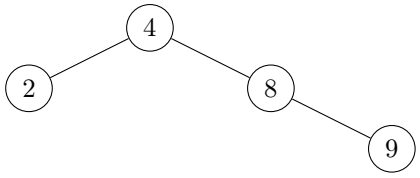
liste [6,3,1,7,5,9,8] (2 dépilé)



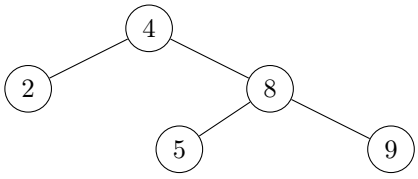
liste [6,3,1,7,5,9] (8 dépilé)



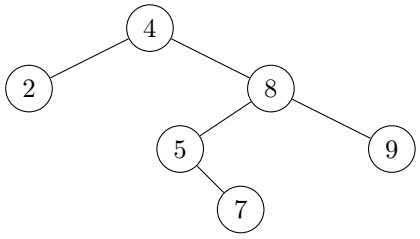
liste [6,3,1,7,5] (9 dépilé)



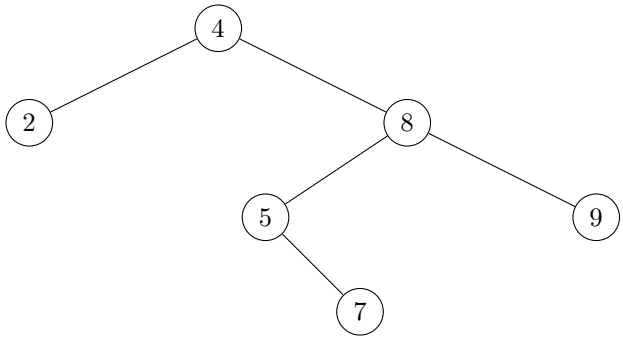
liste [6,3,1,7] (5 dépilé)



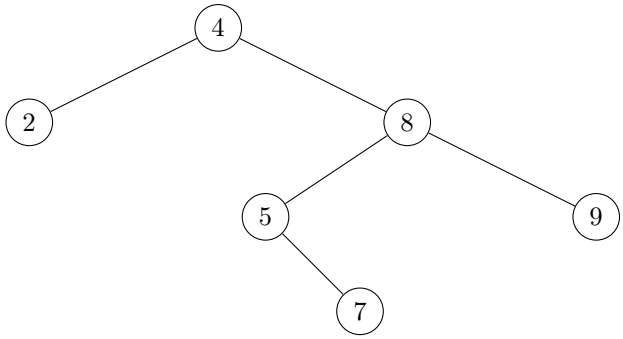
liste [6,3,1] (7 dépilé)



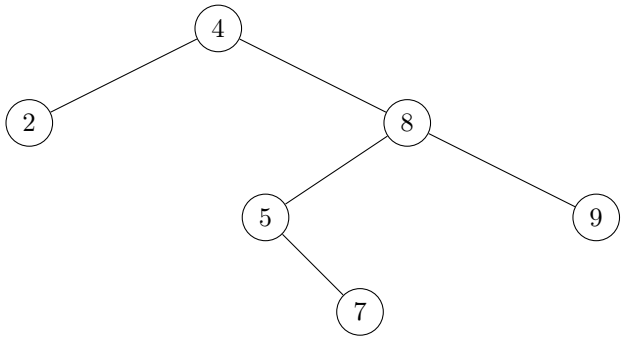
1. Compléter l'ABR en cours de construction, lors des étapes suivantes :
   (a) Liste [6,3] (1 dépilé)



   (b) Liste [6] (3 dépilé)



   (c) Liste vide (6 dépilé)



2. Une fois l'ABR construit, il suffit de le parcourir pour avoir la liste 1, 2, 3, 4, 5, 6, 7, 8, 9 dans l'ordre croissant. Ce parcours de l'arbre est un parcours
   - ☐ en largeur
   - ☐ préfixe
   - ☐ infixe
   - ☐ postfixe
3. On rappelle que la hauteur d'un arbre binaire est définie comme 1 de plus que la hauteur maximale des deux enfants. La hauteur d'une feuille étant supposée égale à 0. Par exemple, les hauteurs des arbres de la page 1 sont 0, 1, 1, 2, 2 et 2.
   (a) Quelle est la hauteur de l'ABR construit à la question 1(c) ?
   (b) Donner, si c'est possible, un autre ABR dont les étiquettes sont aussi 1, 2, 3, 4, 5, 6, 7, 8, 9 mais de hauteur plus petite que celle de l'ABR du 1(c). Si ce n'est pas possible, expliquer pourquoi, sinon le dessiner :

   (c) On définit la fonction `log2` d'un entier de la façon récursive suivante :

```
def log2(n: int) -> int:
    if n<2:
        return 1
    else:
        return 1+log2(n//2)
```

   Isoler la partie du code de la fonction `log2` qui fait que celle-ci est récursive (on pourra inscrire le numéro de la ligne où celle-ci apparaît) : .....
   Voici les premières valeurs de la fonction `log2(n)` :

`n`	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`log2(n)`	1	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5

   (d) On admet que `log2(n)-1` donne la hauteur minimale d'un arbre binaire de taille `n` (c'est-à-dire comprenant `n` nœuds).

Donner, si c'est possible, un ABR de hauteur 2 comprenant les nœuds étiquetés 1, 2, 3, 4, 5, 6, 7, 8, 9. Si ce n'est pas possible, expliquer pourquoi, sinon le dessiner :

4. On veut montrer la terminaison de l'algorithme de construction de l'ABR. Pour cela on prend pour variant
   - ☐ le plus petit entier de la liste
   - ☐ le plus grand entier de la liste
   - ☐ l'entier dépilé
   - ☐ la longueur de la liste restante

   (*on rappelle qu'un variant est un nombre entier qui décroît au cours de l'exécution de l'algorithme*)

5. La *profondeur* d'un nœud dans un arbre, est la longueur du chemin allant de la racine au nœud (ainsi, la hauteur de l'arbre est la plus grande profondeur possible parmi les nœuds de l'arbre). Dans l'ABR de la question 1, voici les profondeurs des nœuds étiquetés :

nœud	1	2	3	4	5	6	7	8	9
profondeur	2	1	2	0	2	4	3	1	2

   On admet que le nombre de comparaisons nécessaires pour insérer un nœud dans un ABR est égal à un de plus que sa profondeur.

   Pour construire l'ABR de la question 1, combien de comparaisons a-t-il fallu faire au total ? .....

6. On admet que le nombre moyen de comparaisons pour construire un ABR de taille `n` est proportionnel à `log2(n)`. On rappelle que le nombre moyen de comparaisons nécessaires pour effectuer un tri fusion sur une liste de taille `n` est proportionnel à `n*log2(n)`. Le tri par ABR est-il plus efficace que le tri fusion ? ..... Expliquer pourquoi :