Algorithmique
Correction Partiel n° 2 (P2)

INFO-SUP S2 – EPITA

7 juin 2021 - 8h30-10h30

Solution **(Arbres de recherches –** ***4 points*****)**

1. **Combien ?** Nombre d'arbres différents avec les valeurs 1, 2, 3 et 4 :
   (a) Arbres binaires de recherche : 14 (figure 1)
   (b) A-V.L. : 4 (figure 1)
   (c) Arbres 2-3-4 : 2 (figure 2)

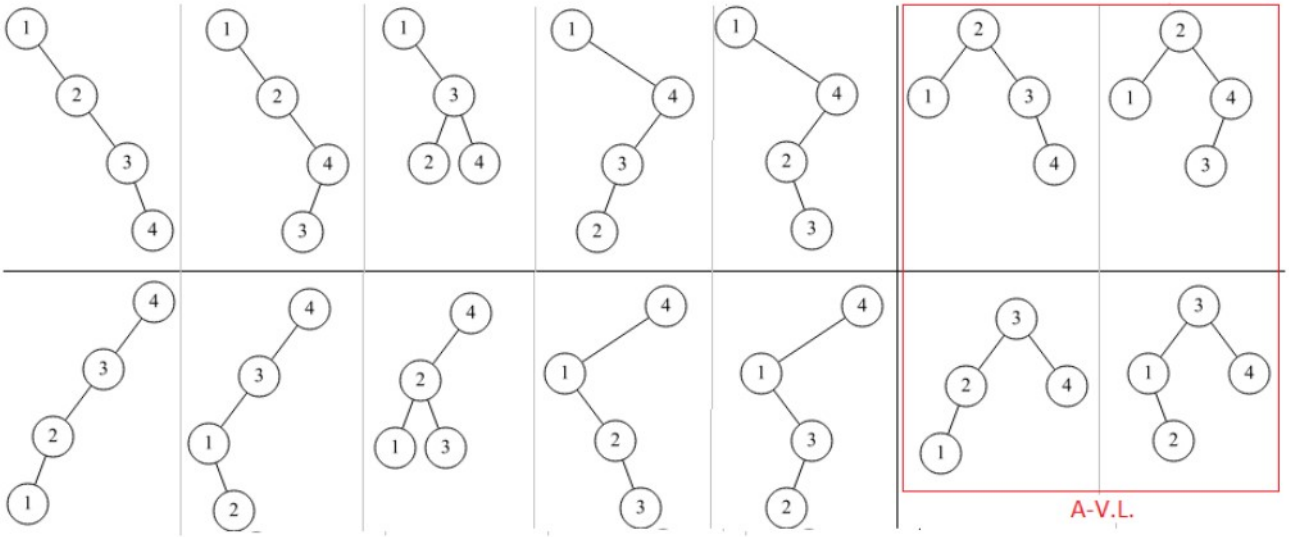


FIGURE 1 – BST with 4 values

FIGURE 2 – 2-3-4 trees with 4 values

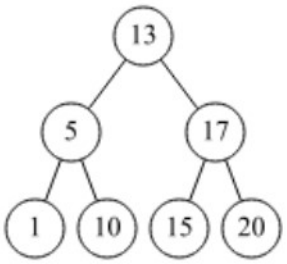
2. **Quoi ?** Quels arbres sont des arbres 2-3-4 ?

	oui	non
B_1	✓	
B_2	✓	
B_3		✓
B_4		✓

Solution **(Dessins – *4 points*)**

1. *Insertions*

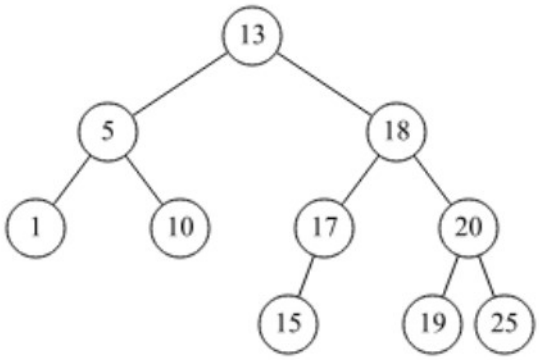
Arbre créé par insertions de 13, 10, 17, 15, 20, 1, 5 :



Rotations :

lrr(10) / rgd(10)

Arbre après ajouts de 25, 18, 19 :

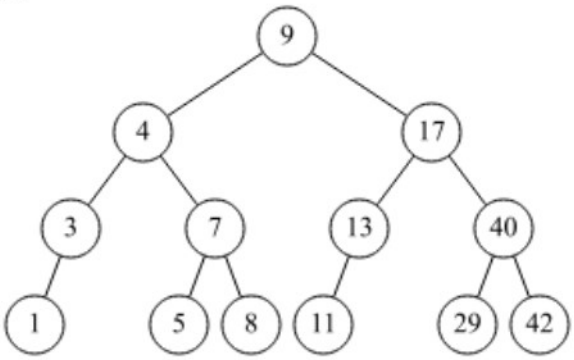


Rotations :

rlr(17) / rdg(17)

2. *Suppression*

Arbre après suppression de 23 :



Rotations :

rlr(29) / rdg(29)
rr(17) / rd(17)

***Solution 3* (Test – *4 points*)**

Spécifications :

La fonction `__testBST(`B`, `inf`, `sup`)` vérifie si l'arbre B est un arbre binaire de recherche avec ses valeurs dans l'intervalle $]inf, sup]$.

La fonction `testBST(`B`)` vérifie si l'arbre B est un arbre binaire de recherche.

```
1   infty = float('inf')
2
3   def __testBST(B, inf, sup):
4       if B == None:
5           return True
6       else:
7           if B.key > inf and B.key <= sup:
8               return __testBST(B.left, inf, B.key) \
9                   and __testBST(B.right, B.key, sup)
10          else:
11              return False
12
13  def testBST(B):
14      return __testBST(B, -infty, infty)
```

***Solution 4* (Génération – *5 points*)**

Spécifications :
La fonction `generation(`B`, `x`, `y`)` vérifie si 2 valeurs x et y différentes sont présentes et de même génération dans l'arbre binaire de recherche B dont les valeurs sont toutes distinctes.

Spécifications : *(Fonction suplémentaire)*
La fonction `search_level(`B`, `x`)` retourne la profondeur de l'élément x dans l'arbre binaire de recherche B si celui-ci est présent et -1 sinon.

– *Version 1 :*

```
1    def search_level(B, x):
2        if B == None:
3            return -1
4        else:
5            if x == B.key:
6                return 0
7            else:
8                if x < B.key:
9                    res = search_level(B.left, x)
10               else:
11                   res = search_level(B.right, x)
12               if res == -1:
13                   return -1
14               else:
15                   return 1 + res
16
17   def generation(B, x, y):
18       res_x = search_level(B, x)
19       if res_x == -1:
20           return False
21       else:
22           return res_x == search_level(B, y)
```

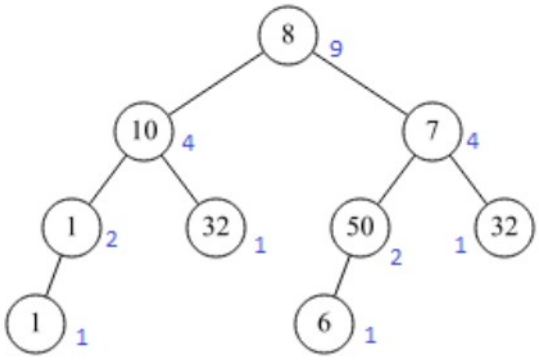
– *Version 2 :*

```
1    def search_level(B, x, d=0):
2        if B == None:
3            return -1
4        else:
5            if x == B.key:
6                return d
7            else:
8                if x < B.key:
9                    return search_level(B.left, x, d+1)
10               else:
11                   return search_level(B.right, x, d+1)
12
13   def generation(B, x, y):
14       dx = search_level(B,x)
15       return dx != -1 and dx == search_level(B,y)
```

Il y a des versions plus optimisées...

***Solution 5* (What is this ? – *3 points*)**

1. Arbre résultat de `mystery`$([1, 1, 10, 32, 8, 6, 50, 7, 32])$:



2. La liste L doit être **strictement croissante** pour que le résultat soit un arbre binaire de recherche.

3. L'arbre résultat est h-équilibré. En chaque nœud de l'arbre il y a au plus un différentiel de 1 entre les tailles des deux sous-arbres (liste "coupée" en 2) qui ont donc soit la même hauteur (déséquilibre 0), soit une différence de hauteur de 1 (déséquilibre 1).