Algorithmique
Correction Partiel n° 2 (P2)

INFO-SUP S2 – EPITA

22 mai 2019

Solution 1 **(La taille en plus – *4 points*)**

Spécifications :

La fonction `copyWithSize(`B`)`, avec B un arbre binaire "classique" (`BinTree`), retourne une copie de B avec la taille renseignée en chaque nœud (`BinTreeSize`).

```
# __copySize(B) returns the pair (copy of B: BinTreeSize, its size: int)
        def __copySize(B):
            if B == None:
                return(None, 0)
            else:
                (left, size1) = __copySize(B.left)
                (right, size2) = __copySize(B.right)
                size = 1 + size1 + size2
                return (BinTreeSize(B.key, left, right, size), size)

    # another version
        def __copySize2(B):
           if B == None:
               return(None, 0)
           else:
               C = BinTreeSize(B.key, None, None, 1)
               (C.left, size1) = __copySize2(B.left)
               (C.right, size2) = __copySize2(B.right)
               C.size += size1 + size2
               return (C, C.size)
```

```
        def copyWithSize(B):
            (C, size) = addSize(B)
            return C
```

Solution 2 **(Ajout avec mise à jour de la taille)**

Spécifications :

La fonction `addwithsize(`B`, `x`)`, ajoute x en feuille dans l'arbre binaire de recherche B (`BinTreeSize`) sauf si celui-ci est déjà présent. Elle retourne un couple : (l'arbre résultat, un booléen indiquant si l'insertion a eu lieu).

```
        def addBSTSize(x, A):
            if A == None:
                A = BinTreeSize(x, None, None, 1)
                return (A, True)
            else:
                if x < A.key:
                    (A.left, insert) = addBSTSize(x, A.left)
                elif x > A.key:
                    (A.right, insert) = addBSTSize(x, A.right)
                else:
                    insert= False
                if insert:
                    A.size += 1
                return (A, insert)
```

***Solution 3* (Médian – *7 points*)**

1. B ABR de n éléments dont le $k^{ème}$ élément ($1 \leq k \leq n$) se trouve en racine :
   — taille(g(B)) $= k - 1$
   — taille(d(B)) $= n - k$

2. *Définition abstraite de l'opération kieme (médian était donné) :*

   AXIOMES

   k = *taille*(G)+1 ⇒ *kieme*(<r, G, D>, k) = r
   k ≤ *taille* (G) ⇒ *kieme*(<r, G, D>, k) = *kieme*(G, k)
   k > *taille* (G) +1 ⇒ *kieme*(<r, G, D>, k) = *kieme*(D, k - *taille*(G) - 1)

3. **Spécifications :**
   La fonction `nthBST(B, k)` avec B un ABR non vide et $1 \leq k \leq taille(B)$, retourne l'arbre dont la racine contient le $k^{ème}$ élément de B .

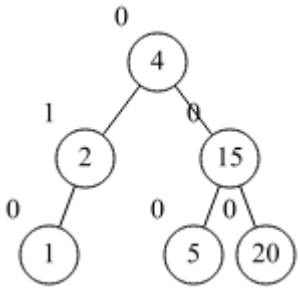
```
        def nthBST(B, k):

            if B.left == None:
                leftSize = 0
            else:
                leftSize = B.left.size

            if leftSize == k - 1:
                return B
            elif k <= leftSize:
                return nthBST(B.left, k)
            else:
                return nthBST(B.right, k - leftSize - 1)


        def nthBST2(B, k):

            if B.left == None:
                if k == 1:
                    return B
                else:
                    return nthBST2(B.right, k - 1)

            else:
                if k == B.left.size + 1:
                    return B
                elif k <= B.left.size:
                    return nthBST2(B.left, k)
                else:
                    return nthBST2(B.right, k - B.left.size - 1)
```

Spécifications :
La fonction `median(B)` retourne la valeur médiane de l'ABR B s'il est non vide, la valeur `None` sinon.

```
            def median(B):
                if B != None:
                    return nthBST(B, (B.size+1) // 2).key
                else:
                    return None
```

***Solution 4* (AVL – *3 points*)**

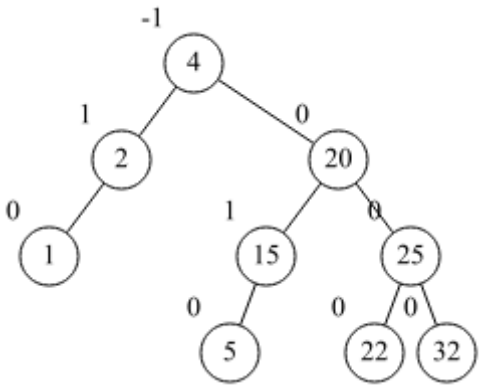
Arbre créé par insertions de 5, 15, 20, 2, 4, 1



Rotations

```
lr(rg) 5
lrr(rgd) 5
rr(rd) 15
```

Arbre après ajout de 32, 25, 22



Rotations

```
rlr(rdg) 20
rlr(rdg) 15
```

***Solution 5* (AVL - Ré-équilibrage – *3 points*)**

Spécifications :

La fonction `rebalancing(`A`)` prend en paramètre un AVL non vide A dont la racine a un déséquilibre dans $[-2, 2]$. Elle effectue si nécessaire une rotation pour ré-équilibrer A. Elle retourne un couple : l'arbre éventuellement modifié et un booléen indiquant si l'arbre a changé de hauteur.

```
1         def rebalancing(A):
2             if abs(A.bal) < 2:
3                 return (A, False)
4             if A.bal == 2:
5                 if A.left.bal == 1:
6                     return (rr(A), True)
7                 elif A.left.bal == 0:
8                     return (rr(A), False)
9                 else:
10                    return (lrr(A), True)
11            else:   # A.bal == -2
12                if A.right.bal == -1:
13                    return (lr(A), True)
14                elif A.right.bal == 0:
15                    return (lr(A), False)
16                else:
17                    return (rlr(A), True)
```