Algorithmique
Partiel n° 2 (P2)

Info-sup S2
Epita

22 mai 2019

Consignes (à lire) :

- Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - Aucune réponse au crayon de papier ne sera corrigée.
- La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- **Le code :**
  - Tout code doit être écrit dans le langage `Python` (pas de C, Caml, Algo ou autre).
  - **Tout code `Python` non indenté ne sera pas corrigé.**
  - Tout ce dont vous avez besoin (fonctions, méthodes) est indiqué en **annexe** !
- Durée : 2h00

Des ABR avec taille

Pour les exercices qui suivent, on ajoute une nouvelle implémentation des arbres binaires dans laquelle chaque nœud contient la taille (`size` !) de l'arbre dont il est racine : `BinTreeSize`.

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

Écrire la fonction `copyWithSize(`B`)` qui prend en paramètre un arbre binaire B "classique" (`BinTree` sans la taille) et qui retourne un autre arbre binaire, équivalent au premier (contenant les mêmes valeurs aux mêmes places) mais avec la taille renseignée en chaque nœud (`BinTreeSize`).

Exercice 2 (Ajout avec mise à jour de la taille– *4 points*)

Écrire une fonction **récursive** qui ajoute un élément en feuille dans un arbre binaire de recherche sauf si celui-ci est déjà présent.
L'arbre est représenté par le type `BinTreeSize`, il faut donc mettre à jour, lorsque nécessaire, le champ *size* en certains nœuds de l'arbre.

Exercice 3 (Médian – 7 *points*)

On s'intéresse à la recherche de la valeur médiane d'un arbre binaire de recherche B, c'est à dire celle qui, dans la liste des éléments en ordre croissant, se trouve à la place $(taille(B) + 1)\ DIV\ 2$.

Pour cela, on veut écrire une fonction `nthBST(`B`, k)` qui retourne le nœud contenant le $k^{ème}$ élément de l'ABR B (dans l'ordre des éléments croissants). Par exemple, l'appel à `nthBST(`B_1`, 3)` avec B_1 l'arbre de la figure 1 retournera le nœud contenant la valeur 5 et l'appel `nthBST(`B_1`, 9)` retournera le nœud contenant 18.

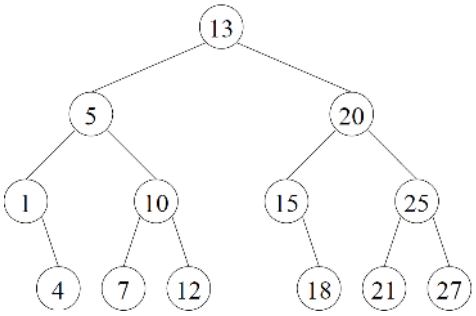

FIGURE 1 – ABR B_1

1. **Un peu d'aide :** Soit B un arbre binaire de recherche contenant n éléments. Si le $k^{ème}$ élément (avec $1 \leq k \leq n$) se trouve en racine, combien d'éléments contiennent les sous-arbres de B ?

2. **Étude abstraite :**
   On ajoute à la définition abstraite des arbres binaires l'opération *taille*, définie comme suit :

   OPÉRATIONS
   taille : ArbreBinaire → Entier

   AXIOMES
   taille (arbrevide) = 0
   taille (<o, G, D>) = 1 + *taille* (G) + *taille* (D)

   Donner une définition abstraite de l'opération *kieme* (utilisant obligatoirement l'opération *taille*) : compléter les définitions abstraites données.

3. **Implémentation :** Les fonctions à écrire utilisent des arbres binaires avec la taille renseignée en chaque nœud (`BinTreeSize`).
   – Écrire la fonction `nthBST(`B`, k)` qui retourne l'arbre contenant le $k^{ème}$ élément en racine. On supposera que cet élément existe toujours : $1 \leq k \leq taille(B)$.
   – Écrire la fonction `median(`B`)` qui retourne la valeur médiane de l'arbre binaire de recherche B s'il est non vide, la valeur `None` sinon.

A-V.L.

Exercice 4 (Construction – *3 points*)

À partir d'un arbre vide, construire l'AVL en insérant successivement les valeurs 5, 15, 20, 2, 4, 1, 32, 25, 22. Vous dessinerez l'arbre à deux étapes :
- après l'insertion de 1 ;
- l'arbre final.

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

Nous nous intéressons ici au ré-équilibrage d'un AVL après une insertion ou une suppression.

Rappel

Le ré-équilibrage d'un AVL après une modification (insertion ou suppression) se fait à la remontée :

1. A la remontée, si la hauteur du sous-arbre (où a eu lieu la modification) a changé alors le déséquilibre du nœud courant est mis à jour.
2. **Partie à écrire :** Si le déséquilibre est incorrect, alors une rotation est effectuée. Il est nécessaire de savoir si cette rotation a changé la hauteur de l'arbre.

Écrire la fonction `rebalancing(`A`)` qui rééquilibre si nécessaire l'AVL A après une modification du déséquilibre de sa racine. La fonction retourne l'arbre après éventuelle rotation, ainsi que le changement éventuel de hauteur induit (un booléen).
Vous pouvez utiliser les fonctions qui effectuent les rotations avec mises à jour des déséquilibres (`lr, rr, lrr, rlr`, voir annexes).

Annexes

Les arbres binaires

Les arbres binaires "classiques" : :

```
class BinTree:
    def __init__(self, key, left, right):
        self.key = key
        self.left = left
        self.right = right
```

Les arbres binaires avec taille :

```
    class BinTreeSize:
        def __init__(self, key, left, right, size):
            self.key = key
            self.left = left
            self.right = right
            self.size = size  # size of the tree!
```

Les AVL, avec les déséquilibres : :

Rappel : dans un A-V.L. les clés sont toutes distinctes.

```
class AVL:
    def __init__(self, key, left, right, bal):
        self.key = key
        self.left = left
        self.right = right
        self.bal = bal
```

Rotations (A:`AVL`) : chaque fonction ci-dessous retourne l'arbre A après rotation et mise à jour des déséquilibres.

- `lr`(A) : rotation gauche
- `rr`(A) : rotation droite
- `lrr`(A) : rotation gauche-droite
- `rlr`(A) : rotation droite-gauche

Vos fonctions

Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font).

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.