Algorithmics
Final Exam #2 (P2)

Undergraduate 1^{st} year S2
EPITA

May, 22th 2019

Instructions (read it) :

☐ You must answer on **the answer sheets provided.**

– No other sheet will be picked up. Keep your rough drafts.

– Answer within the provided space. **Answers outside will not be marked**: Use your drafts!

– Do not separate the sheets unless they can be re-stapled before handing in.

– Penciled answers will not be marked.

☐ The presentation is negatively marked, which means that you are marked out of 20 points and the presentation points (maximum of 2) are taken off this grade.

☐ **Code:**

– All code must be written in the language `Python` (no C, CAML, ALGO or anything else).

– **Any `Python` code not indented will not be marked.**

– All that you need (types, routines) is indicated in the **appendix** (last page)!

– Your functions must follow the given examples of application.

☐ Duration : 2h

BST with size

In the following exercises, we use a new implementation of binary trees where each node contains the size of which it is the root of: `BinTreeSize`.

Exercise 1 (Add the size – *4 points*)

Write the function `copyWithSize(`B`)` that takes a "classic" binary tree B (`BinTree` without the size) as parameter and returns an equivalent tree (containing the same values at the same places) but with the size specified in each node (`BinTreeSize`).

Exercise 2 (Insertion with size update– *4 points*)

Write a **recursive** function that adds in leaf a new element to a binary search tree, unless it is already present.
The tree is represented by the type `BinTreeSize`. Thus you have to update, when needed, the *size* field in some tree nodes.

Exercise 3 (Median – 7 *points*)

We will study the research of the median value in a binary search tree, that is, the value at the rank $size(B) + 1\ div\ 2$ in the list of elements in increasing order.

For this, we want to write the function `nthBST(`B`, `k`)` that returns the node that contains the k^{th} element of the tree B. For example, the call `nthBST(`B_1`, 3)` with B_1 the tree in figure 1 will return the node that contains the value 5 and `nthBST(`B_1`, 9)` will return the node that contains 18.

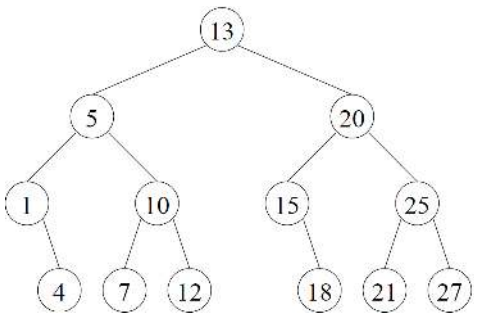

Figure 1: ABR B_1

1. **A little help:** Let B be a binary search tree with n elements. If the k^{th} element (with $1 \leq k \leq n$) is in the root, how many elements do the two subtrees of B contain?

2. **Abstract study:**

   The *size* operation, defined as follows, is added to the abstract definition of binary trees:

   OPERATIONS
   size : BinaryTree → Integer

   AXIOMS
   $size$ (emptytree) $= 0$
   $size$ (<o, L, R>) $= 1 + size$ (L) $+ size$ (R)

   Give an abstract definition of the operation *nth* (that has to use the operation *size*): complete the given definitions.

3. **Implementation:** The functions you have to write use binary trees with the size in each node (`BinTreeSize`).

   - Write the function `nthBST(`B`, `k`)` that returns the tree with the k^{th} element as root. We suppose that this element always exists: $1 \leq k \leq size(B)$.

   - Write the function `median(`B`)` that returns the median value of the binary search tree B if B non empty, the value `None` otherwise.

IF YOU ASK ME, HOBBES, THE WHOLE NOTION OF "INSTANT GRATIFICATION" IS A MYTH!

A-V.L.

Exercise 4 (Construction – *3 points*)

Starting with an empty tree build the AVL corresponding to the successive insertions of values 5, 15, 20, 2, 4, 1, 32, 25, 22. You have to draw the tree at two steps:

- after insertion of 1 ;
- the final tree.

Exercise 5 (AVL - Re-balancing – *3 points*)

We focus here on re-balancing an AVL after an insertion or a deletion.

Reminder

The rebalancing of an AVL after a modification (insertion or deletion) is made "in going up":

1. Going up if the height of the subtree (where the modification occurred) has changed then the current node balance factor is updated.
2. **Part to write:** If the balance factor is incorrect, then a rotation is performed. Knowing whether the rotation changes the tree height is required.

Write the function `rebalancing(`A`)` that re-balances the AVL A if required after a modification of the balance factor of its roots. The function returns the tree after potential rotation, as well as the possible induced height variations (a boolean).
You can use the functions that perform the rotations with balance factor updates (`lr, rr, lrr, rlr`, see appendix.)

WOW! WHAT A PROJECT!
COLOR THESE PICTURES! CUT AND PASTE! DRAW THOSE TREES! MORE CUTTING! MORE PASTING!
WHAT A LEARNING EXPERIENCE! YES, MA'AM, YOU'VE DONE IT AGAIN!
WHEN SHE'S HAPPY, WE'RE HAPPY..
© 1997 United Feature Syndicate, Inc.

Appendix

Binary Trees

Usual binary trees:

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

Binary trees with size:

```
    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!
```

AVL, with balance factors:

Reminder: in an A.-V.L keys are unique.

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

Rotations (A:`AVL`): each of the functions bellow returns the tree A after rotation and balance-factor updates.

- `lr(`A`)`: left rotation
- `rr(`A`)`: right rotation
- `lrr(`A`)`: left-right rotation
- `rlr(`A`)`: right-left rotation

Your functions

You can write your own functions as long as they are documented (we have to known what they do).
In any case, the last written function should be the one which answers the question.