Algorithms & Data Structures
SI3 - Polytech Nice-Sophia - Edition 2018

Lab #4: Binary Search Trees

This assignment will give you practice about binary search trees.

Part 1: introduction to binary search trees

In this part you have to write the basic methods inside the `BST` class. Check carefully this class and review the slides. You are to complete the following methods:

- `contains`: check if a given value is inside the binary search tree. (Already completed !)
- `insert`: add a new element in the binary search tree. If the element is already there, the method returns False, otherwise it returns True.
- `find_extremum` : return the minimum (or maximum) element of the binary search tree. If the tree is empty, these methods should throw the exception `ValueError`
- `remove_extremum` : remove the minimum (or maximum) element of the binary search tree and return the value removed.
- `remove`: remove an element in the binary search tree. If the element is not in the tree, it returns False, otherwise it returns True.

For all these methods, you must give the worst case run time complexity.

Supporting files:

- Lab4.py

Part 2: more methods on binary search trees

In this part you have to implement more algorithms on binary search trees (use the same class as in previous part). You are to complete the following method:

- `remove_compare` : remove from the tree all the element which are less than (or greater than) a given element. These functions should not use the remove method from previous part to obtain optimal complexity.

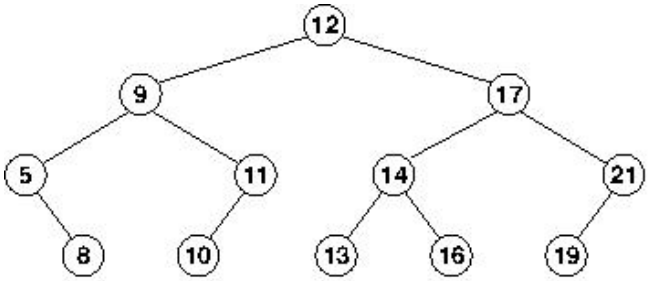
Part 3: iterator on binary search trees

Just like many other data structure in Python, we want a binary search tree to be iterable, i.e. the class `BST` must implement the method `__iter__`. This method must return an `iterator` over this tree. An iterator is a function that uses `yield` to generate successive values. We want to obtain an iterator that generate all values in the tree from minimum to maximum. You can also use `yield from` to generate successive values from another iterator (or from the same one with recursive calls).

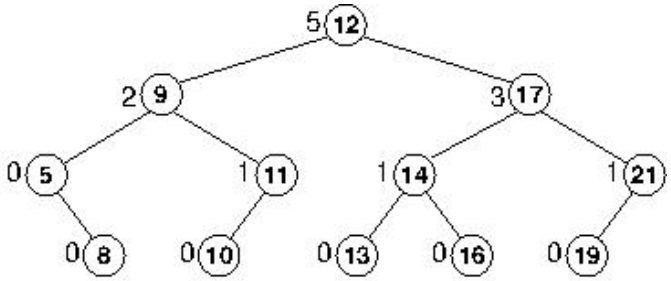
Once a tree is iterable, it can be used in `for` loops or in constructor like `list` automatically.

Part 4: binary search trees with rank

Since the elements of a binary search tree are stored somehow in order, it would be useful to access them by their *rank* (i.e. their index in an increasing sorted list). For example, in the binary search tree below, the element of rank 1 is 5, the element of rank 12 is 21, the element of rank 8 is 14 and the rank of element 9 is 3:



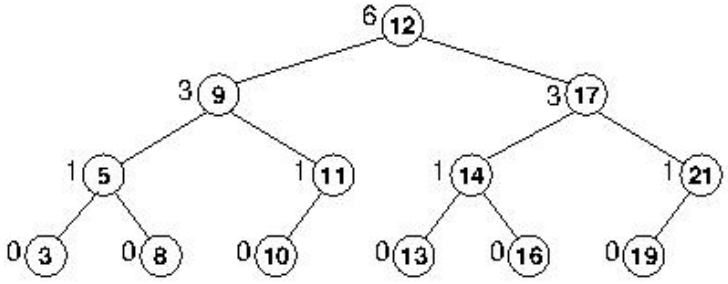
Binary Search Tree



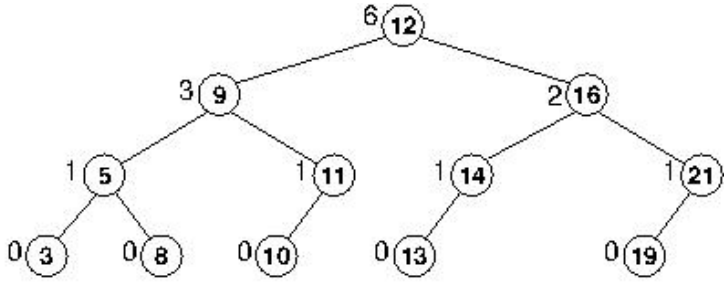
Ranked Binary Search Tree

- To manage the rank information, one could store the rank of each element in the corresponding tree node. Explain why this is not a good solution

Instead, a smart idea is to store in each node the *size of the left sub-tree*. On the previous picture, you can see the corresponding ranked binary search tree (on the right) of the initial binary search tree (on the left). The operations on a ranked binary search tree are identical to the one on a simple binary search tree except that some operations must update the new attribute. On the next picture you can see the consecutive results of inserting 3 and removing 17 from the previous ranked binary search tree:



after inserting (3)



after removing (17)

Ranks in python are defined from 0 (the minimum element) to n-1 (the maximum element) if there are n elements in the tree.
Using the provided class template `RankedBST`, you are to write:

- the methods `insert`, `remove_extremum` and `remove`, from part 1. You must adapt the methods which need to update the new attribute `sizeOfLeft`
- the method `__getitem__`: given a rank, this method returns the element in the tree of that rank. If the rank is out of bounds, the method returns `null`. Give as a comment the complexity of this method. Once implemented, the tree can be used as an array (`tree[i]`).
- the method `index`: given an element, this method returns its rank in the tree. If the element is not in the tree, the method raise an exception. Give as a comment the complexity of this method.

Part 5: AVL trees

This par consists only in the following write up questions:

- draw two AVL trees of height 4, one which holds as less element as possible, one which holds as much element as possible
- draw the AVL tree resulting of inserting the following integer value inside an initially empty AVL tree: 9, 4, 1, 3, 2, 8, 10, 6, 5, 11, 7
- give the ordering we could remove all the elements from the previous tree such that no re-balancing is needed
- give the minimum number of nodes (elements) of an AVL of height 10