TD3 – Binary Search trees

Version originale : Marc Gaetano et l'équipe pédagogique ASD

This assignment will give you practice with binary search trees.

Part 1: Introduction to binary search trees

In this part, you must write the basic methods inside the *BinarySearchTree* class. Check this class carefully and review the slides. You are to complete the following methods:

- **contains**: check if a given value is inside the binary search tree.
- **insert**: add a new element in the binary search tree. If the element is already there, the method does nothing.
- **findMin** and **findMax**: return the minimum (resp. maximum) element of the binary search tree. If the tree is empty, these methods should throw the exception *UnderflowException*
- **remove**: remove an element in the binary search tree. If the element is not in the tree, the method does nothing.

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

Supporting files:
Two approaches are possible.

- Either you use the structure given to you, which uses Innerclass; it has the advantage of simplifying operations that directly modify a node;
- or you are free and start only with the skeleton of code given to you.

The same tests should work in both cases; hence some of the comments left in the tests.

With Inner Class

- BinarySearchTreeWithInnerClass
- BinarySearchTreeWithInnerClassTest

Squeletton

- BinarySearchTree.java
- BinarySearchTreeTest.java

Part 2: more methods on binary search trees

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

- **removeLessThan** and **removeGreaterThan**: remove from the tree all the elements which are less than (resp. greater than) a given element. These functions should not use the remove method from the previous part.
- **toSortedList**: return a list of all the elements of the binary search tree sorted in increasing order (from the least to the greatest)
- the BinarySearchTree(List<AnyType> list) constructor: this constructor performs the reverse operation from the previous method. Given a sorted (increasing) list of elements, this constructor builds a binary search tree containing all these elements. Explain (as a comment in your code) why this is not symmetric to the previous operation.

Part 3: iterator on binary search trees

Just like in the Java collection framework, we want a binary search tree to be iterable, i.e., the class BinarySearchTree must implement the generic interface Iterable. This interface defines the method iterator, which, given a binary search tree, returns an iterator over this tree. An iterator allows reading consecutively all the elements from the tree in increasing order, using the methods hasNext and next defined in the Iterator interface.

- give a straightforward implementation for the inner class BSTiterator and give its memory usage complexity
- think of another implementation that can traverse the tree and minimize memory usage. Write the class BSTiterator accordingly. Give the complexity of this solution for memory usage when the tree is balanced

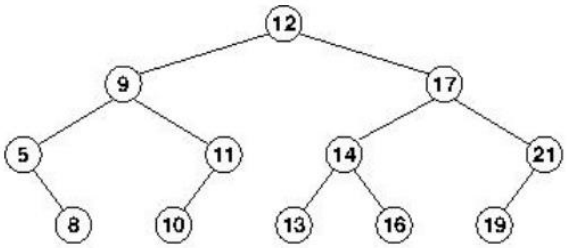
The template for the inner class BSTiterator is already in the class template BinarySearchTree.

Part 4: binary search trees with rank

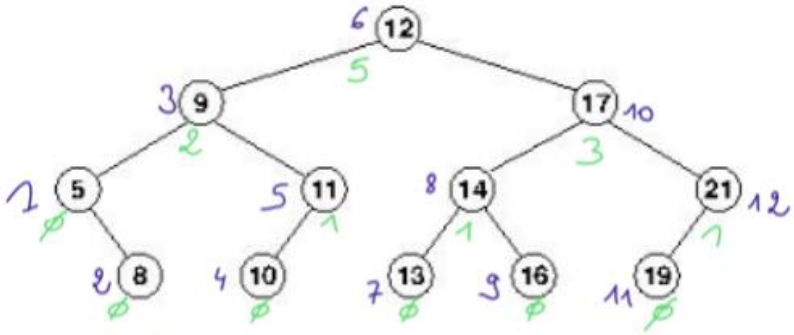
Since the (comparable) elements of a binary search tree are stored somehow in order, it would be useful to access them by their *rank* (i.e., their index in an increasingly sorted list).

The rank of a node value x in a tree is the number of the nodes whose values are ≤.

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:

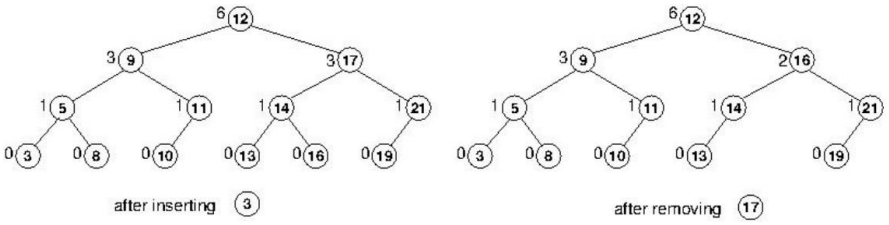


Binary Search Tree



- 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*. In 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 ones on a simple binary search tree, except that some operations must update the new attribute. In the next picture, you can see the consecutive results of inserting 3 and removing 17 from the previous ranked binary search tree:



Using the provided class template RankedBST, you are to write the following:

- The methods *contain*, *insert* and *remove*, from part 1. You must adapt the methods which need to update the new attribute sizeOfLeft
- The method *rank*: given an element, this method returns its rank in the tree. If the element is not in the tree, the method returns 0. Give as a comment the complexity of this method
- and the method element 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 a comment on the complexity of this method.

Tips :

* if the node is the current node, return the size of the left subtree + 1 (all nodes on the left are smaller than the current node)

* if the node is in the left subtree, return the rank of the node in the left subtree

* if the node is in the right sub-tree, return the rank of the node in the right sub-tree + the size of the left sub-tree + 1 (all nodes in the left sub-tree are smaller than the current node, and the searched node is larger than the current node (its ranks +1)).

Supporting file:
Two approaches are possible.

Either you use the structure given to you, which uses Innerclass; it has the advantage of simplifying operations that directly modify a node;

Or you are free and start only with the interface given to you.

Part 5: AVL trees

This part consists only of the following write-up questions:

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