140012065 Mark-Jan Nederhof 19/10/15

Over View:

For this practical we were asked to build a binary search tree of a set of integers from scratch. We had to implement the operations to add, find, remove and perform set union. We also had to optimize our implementation of the whole-set union as efficient as possible.

Design:

I have implemented three classes, Node, BinarySearchTree and Set and Testing.

I have used the node class to define the data structure of my binary tree. So it contains an Integer value, Node right, Node left and finally Node parent. I put node parent into the constructor as well because I thought it would make things easier later on when creating or deleting the tree. A node knowing as much information bout its surroundings can help a lot.

BinarySearchTree implements methods that concern it such as add, remove, contains and find Node. Those are the main methods for the class.
The add method adds the values as they are being inputted. It does not balance the tree in any way. And has a complexity of log(n).
The contains method returns a boolean if the value is found. And also has a complexity of log(n).
The remove method removes a value that needs to be deleted by the user. It does this first by finding the value by using find Node. Once this is done it does three checks. It checks firstly if the node has no children, if not children exist then it deletes it however before deleting the node it checks with side of the parent it is on. This is because deleting the value and node are different things.
The second check is to see if the node has a child. If it does have a child and depending on which side of the node it is on the node gets replaced by its one child and parent of this child the gets re-updated to that of the original nodes parent.
The final check is to see if the node has both children. If it does have both children, then we replace the node with the biggest value on the left side of the node. The remove method is also log(n).

The Set class implements the methods for set functions such as set union and intersection. It contains the methods flatten, merge, intersectionMerge, union, newUnionTree, intersection and newIntersectionTree.

To do union I thought it was necessary to first flatten the tree into the list so that manipulating that would be easier to do and more efficient. I then went on to merge the two lists that were made from two trees. Before finally creating a new tree out of the merged list.

Flatten complexity O(n):

I have optimized the method flatten so that the complexity is of n. It goes through the tree once while it adds them to an arraylist in order.

Merge complexity O(n):

I have optimized this so that again the complexity is n where merge is concerned. This is because I have one while loop, and in that while loop I go through two arraylists once. Check each element and add it so that they are sorted, I delete the checked elements. Once it reaches the end of one arraylist then adds the rest. I delete every checked element so that appending is easier to do.

NewUnionTree complexity O(n):

This is where the new tree is made. It is made of a tree that has Node and its children and is not related to the BinarySearchTree yet. It produces a balanced tree as well. This is done by finding the mid point and making that the main root node. It then finds the mid point of the left side of the array list and makes that a new node which is put as the child of the original node. This keeps getting done until the end of the list. The same thing is happening to the right side of the tree as well. By the end of it a tree is created. The complexity is O(n) as it creates the tree with out iterating through it more then it once. It also does not add any elements to the tree so it is not n(log(n)).

Union complexity O(n):

This is where all the methods are called to create the proper tree. The main Node of the newUnionTree is assigned to the root of the tree being created. The complexity is O(n) as it is simplified to O(n) from 3n. This is because Flatten, Merge and NewUnionTree are needed to create the set Union operation. As all three of these have the same complexities they become n+n+n, which is 3n, which then simplifies to n.

IntersectionMerge complexity O(n):

I have done something similar to merge. The main thing being changed is that I do not delete the elements any more as I increment instead. This is because all the elements still need to be checked.

NewIntersectionTree complexity O(n):

This is almost the exact same as newUnionTree. The only things changing is the names.

Intersection O(n):

This is almost the exact same as union. As it uses the same concepts and the only things changing is the names.

Testing:

I have tested the cases needed for each method through the j-unit class test.

Evaluation:

I tried to implement the one of the extension which was the set intersection method. I also did balancing However this was only done if one were to use union or intersection as I built it into them when the tree was being recreated. In terms of code clarity I think that the classes Node and BinarySearchTree are quite clear however the set class uses a lot of duplicated code. This is because I implemented intersection too late and therefore ended up using a lot of the same code as with union. This might've been able to be avoided if I had planned in advance.

Conclusion:

This was a very Interesting practical to do and I feel I have learnt a lot doing it. I am beginning to understand complexity better now I feel. I am also more comfortable with binary search trees. I found this practical difficult to do. Especially the optimizing of the algorithms however I think I managed to complete the basic task and the one of the extension tasks properly.