CS2001 130011035 Tutor: Mark-Jan Nederhof W10 Trees

Introduction:

This weeks Practical asked that we build an efficient representation of sets and their operations using trees.

Design/Implementation:

In this weeks practical we were able to make what we needed to in order to implement the tree structure. This involved being able to add new nodes to the tree, remove nodes and make a union of two trees. On top of this we had to implement the set structure, and since the set was to be stored in the tree I felt it best that when the set was actually used to manipulate the tree collection class. This meant that all methods in the set class were only a few lines long and very self explanatory. Also due to how I implemented the add methods, it made sure that the tree was a balanced tree.

In the BinaryTreeCollection class, the methods were implemented as such:

add (one parameter) – This was used to add an initial value to the root node of the tree if there wasn't one already. If there was one then it would call the other add method.

add(two parameters) – This would initially find out if the value which was to be added was greater than or smaller than the value of the node the method was currently on. Then it would recursively check through the values depending on this to find the first null value it would encounter. It would then set that to be the newest node in the tree.

Remove – This would check if there actually was anything in the tree, then if there was it would search through the tree to find the value which was to be removed. If the value had no sub-nodes, then the value was set at null. If there were sub-nodes, then the method would work to replace the node which was to be deleted with the smallest value which was larger than the one being replaced, or the only other one available.

Find node – This method would find a specific node within the tree. It would first check if the initial node was the correct node, if not it would find out if it should check the left side of the tree or the right side. Once this was decided, the method would recursively search through the tree to find the correct value.

Size – This method would check to see if the tree actually had any node in it, and if it doesn't, then it would return a size of 0. Other than that, it would recursively check the size of the left tree and right tree after adding one to store the initial root node.

Print stuff – This method was used as a link to print out a nice tree showing how the tree was balanced.

Union – This method would take two trees and merge them together. To do so it made use of an array list and the flatten method. Initially the method added the root nodes of both trees to the array list, before using the flatten method on both trees. The array list was then turned into an array, and looped through, using the add method to create a new tree.

Flatten – This method took in a tree, then recursively moved through it until the entire tree had been converted into the array list.

BinaryTreeNode class:

Remove – This method allowed the remove method in the BinaryTreeCollection class to remove the node correctly. To do so the method checked if the value to be removed was greater than or less than the node the check was on. Then would find the value to be removed, then use the method min Value to find which node should be used to replace the node which was removed if a replacement node was necessary.

Min Value – This method would find the node which should be used to replace the node being removed.

All the print methods – These would be used to print out nicely the tree, by finding out if the tree is going to the left or the right, then printing out different string values to show links between the different nodes of the tree.

Time complexities:

add (one and two parameters) – O(n log n) logarithmic as it will only need to search through a specific path to find the correct place to add the new node, and not need to search through the entire tree.

Remove (BinaryTreeCollection) – O(n log n) logarithmic as it will only need to search through a specific path to find the correct node to be removed, and not need to search through the entire tree.

Find node – O(n log n) logarithmic as it will only need to search through a specific path to find the correct node, and not need to search through the entire tree.

Size – O(n * n) quadratic as it will need to search through every node in the tree to find out how many nodes are in the tree.

Print stuff – O(n * n) quadratic as it will need to search through every node in the tree in order to print out the method.

Union – O(n log n) logarithmic as it will move through the list adding the new nodes in a balanced way, also it would look through the two trees, and if there were duplicates, then the program would not allow these to be added to the new tree created in the union method.

Flatten – O(n * n) quadratic as it will need to search through every node in the tree to find out all the values of all the nodes in the tree.

Remove (BinaryTreeNode class) – O(n log n) logarithmic as it will only need to search through a specific path to find the correct node to be removed, and not need to search through the entire tree.

Min value – O(n log n) logarithmic as it will only need to search through a specific path to find the correct node to be used as a replacement for the node being removed.

Print methods – O(n * n) as it needs to move through all the values of the tree in order to print out all of the values of all of the nodes of the tree.

Testing:

Test1: size – I added 10 nodes to the tree, then used the test to make sure that size returned 10.

Test2: add – Now that I knew that size was working, I added 6 nodes to the tree and tested that add worked.

Test3: add error – Since add worked when using correct input values, I had to test how it dealt with duplicates being added, and it reacted as it should by throwing a Duplicated Node Exception.

Test4: remove – I added 5 nodes to the tree and removed one, so the size should only be 4 and it was.

Test5: union – I added 5 nodes to two trees, before calling a union on the two trees. I then used the size function to make sure that ten values were present in the new tree, which there were.

Test6: union error – I made sure that the union method would not allow duplicates to be added.

Test7: find node – I used this test, to make sure that the find node function worked, by adding a number of nodes and asking the method to find the correct one.

Test8: find node error – If the find node method finds no node then it should throw a null pointer exception and it does.

Test9: set add – makes sure that the add method in the set class works, also tests if size in the set class works

Test10: set remove – again makes sure that the remove method in the set class works

Conclusion:

This weeks practical was okay. There were a few things which I was confused by but managed to muddle through in the end, like the remove function. Initially I had managed to get it to set the value of that node to null, but this would not work properly as every value in the branches stemming off that node would be removed as well. Another problem I had was understanding what the practical specification was asking for in terms of the set class. So the implemented methods are not very thorough, and are just calling the working methods in the other classes more or less. I also attempted an extension, in trying to increase the efficiency of the program, I made the tree so that it would be balanced. I am not sure if this qualifies as an extension but I thought that I would bring your attention towards this fact.