Jasque Saydyk

Professor Patrick Kelley

CS 249 Section 3140

19 March 2017

Project 4 - Binary Search Trees

Binary Search Tree

The implementation of the Binary Search Tree was relatively straightforward, especially when compared to the Red Black tree. The one method I spent a significant amount of time on was the getTreeString() method, as getting the right number of brackets at the end of the string was difficult. Aside from that, I used an iterative approach to the majority of the methods, like the put() and get() methods. For the toString(), getHeight(), and getTreeString() methods, I used recursion to complete the classes.

As for the profiling of the results from the class, a sample of the results for getting numbers that were randomly put into the tree. Of the 10,000 runs, the vast majority of them were between 0 to 1 microseconds, showing the O(n log(n)) nature of the data structure if it is roughly balanced. Occasionally you get random spikes of around 100 microseconds to 200 microseconds, but this is very small and relatively insignificant. The height of this tree ends up being 30, which is pretty balanced for this many items

```
number: 4809  Milliseconds: 0  Microseconds: 0
number: 4810  Milliseconds: 0  Microseconds: 0
number: 4811  Milliseconds: 0  Microseconds: 0
number: 4812  Milliseconds: 0  Microseconds: 0
number: 4813  Milliseconds: 0  Microseconds: 0
number: 4814  Milliseconds: 0  Microseconds: 0
number: 4815  Milliseconds: 0  Microseconds: 0
number: 4816  Milliseconds: 0  Microseconds: 109
number: 4817  Milliseconds: 0  Microseconds: 1
number: 4818  Milliseconds: 0  Microseconds: 1
number: 4819  Milliseconds: 0  Microseconds: 0
number: 4820  Milliseconds: 0  Microseconds: 0
```

```
number: 9998  Milliseconds: 0  Microseconds: 0
number: 9999  Milliseconds: 0  Microseconds: 0
The height of this Red Black Tree is: 30
```

As for the second profile where an ascending order of numbers is put into the binary tree, making it more akin to a Linked List than a Binary Tree. Below are two snapshots of the data,

one close to the beginning, the second towards the end. We see a gradual increase in the time it takes to retrieve data, from 0 to 3 microseconds to about 30 microseconds. There are anomalies throughout the results, where the time will randomly spike to anywhere around 200 to 800 microseconds, which I can not fathom what the cause of it might be. These results make sense, as the Binary Search Tree takes a form more akin to a Linked List. Observing the results and the algorithm for retrieve the items, which is a while loop, the complexity is O(n) in this case. This tree ended up having the height of 9,999 node levels, thus showing that it is practically a Linked List.

```
number: 823  Milliseconds: 0  Microseconds: 2
number: 824  Milliseconds: 0  Microseconds: 2
number: 825  Milliseconds: 0  Microseconds: 2
number: 826  Milliseconds: 0  Microseconds: 2
number: 827  Milliseconds: 0  Microseconds: 2
number: 828  Milliseconds: 0  Microseconds: 2
number: 829  Milliseconds: 0  Microseconds: 2
number: 830  Milliseconds: 0  Microseconds: 2
number: 831  Milliseconds: 0  Microseconds: 2
number: 832  Milliseconds: 0  Microseconds: 2
number: 833  Milliseconds: 0  Microseconds: 2
number: 834  Milliseconds: 0  Microseconds: 28
```

```
number: 9815  Milliseconds: 0  Microseconds: 26
number: 9816  Milliseconds: 0  Microseconds: 26
number: 9817  Milliseconds: 0  Microseconds: 26
number: 9818  Milliseconds: 0  Microseconds: 26
number: 9819  Milliseconds: 0  Microseconds: 26
number: 9820  Milliseconds: 0  Microseconds: 32
number: 9821  Milliseconds: 0  Microseconds: 27
number: 9822  Milliseconds: 0  Microseconds: 27
number: 9823  Milliseconds: 0  Microseconds: 27
number: 9824  Milliseconds: 0  Microseconds: 26
number: 9825  Milliseconds: 0  Microseconds: 26
number: 9826  Milliseconds: 0  Microseconds: 433
```

```
number: 9998  Milliseconds: 0  Microseconds: 30
number: 9999  Milliseconds: 0  Microseconds: 30
The height of this Red Black Tree is: 9999
```

Red Black Tree

The section of the project was bloody difficult, primarily because one has to figure out and be aware of the edge cases when it comes to correcting an insertion. All of the other methods were relatively simple adjustments to what was implemented in the Binary Search Tree class. For the Put() method, I settled on using a recursive approach for the correction of any errors that might occur, which was simpler to work with than a big while loop. While working on it, I determined 5 cases, the first was if the node was root and red, make it black and be done. The second case was checking if the parent is black, in which case be done. The third case checks if the uncle is also red, and if so, flips the colors of the child and the parent and uncle, then begins the recursion again on the parent. The fourth case, knowing that both the child and the parent are

red, checks to see if the parent and child are in a bent node structure, and if so, performs a specific bent rotation on it. The last case performs a straight rotation on the nodes.

I made extensive use of private methods to find the uncle and grandparent of the node and to perform the four different types of rotations. I made changes to the node class to give it a link to it's parent to make manipulating the tree easier, and I made it's color a boolean value, where I have Red as False, and True as Black. I have a private method to convert the boolean color value to a useful String, along with a getMinHeight method that finds the height of the shortest branch in the tree, which is used in a modified version of the getTreeString() method.

As for the profile of the Red Black Tree, I was unable to get the randomly generated Tree structure to output anything, however, a list of ascending numbers inserted into the tree works correctly. The results from this, as shown in the sample below, are consistently between 0 to 3 microseconds across the board, with some minor spikes caused by some events I can not determine. This, in contrast to the Binary Search Tree from before, shows how well the Red Black Tree maintains the balance of the Search Tree, and thus achieves a guarantee of O(n log(n)) speeds. The height of this tree is 22, showing the balanced nature of the tree.

```
number: 3476  Milliseconds: 0  Microseconds: 1
number: 3477  Milliseconds: 0  Microseconds: 0
number: 3478  Milliseconds: 0  Microseconds: 0
number: 3479  Milliseconds: 0  Microseconds: 1
number: 3480  Milliseconds: 0  Microseconds: 1
number: 3481  Milliseconds: 0  Microseconds: 0
number: 3482  Milliseconds: 0  Microseconds: 0
number: 3483  Milliseconds: 0  Microseconds: 0
number: 3484  Milliseconds: 0  Microseconds: 0
number: 3485  Milliseconds: 0  Microseconds: 0
number: 3486  Milliseconds: 0  Microseconds: 0
number: 3487  Milliseconds: 0  Microseconds: 0
number: 3488  Milliseconds: 0  Microseconds: 0
number: 3489  Milliseconds: 0  Microseconds: 0
number: 3490  Milliseconds: 0  Microseconds: 0
```

```
number: 9997  Milliseconds: 0  Microseconds: 0
number: 9998  Milliseconds: 0  Microseconds: 0
number: 9999  Milliseconds: 0  Microseconds: 0
The height of this Red Black Tree is: 22
```