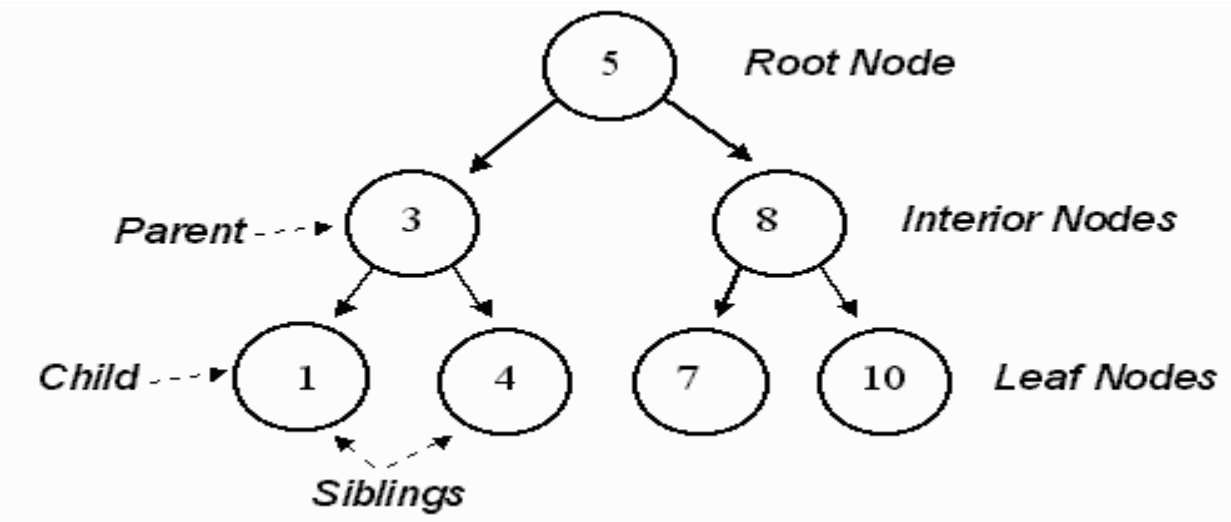
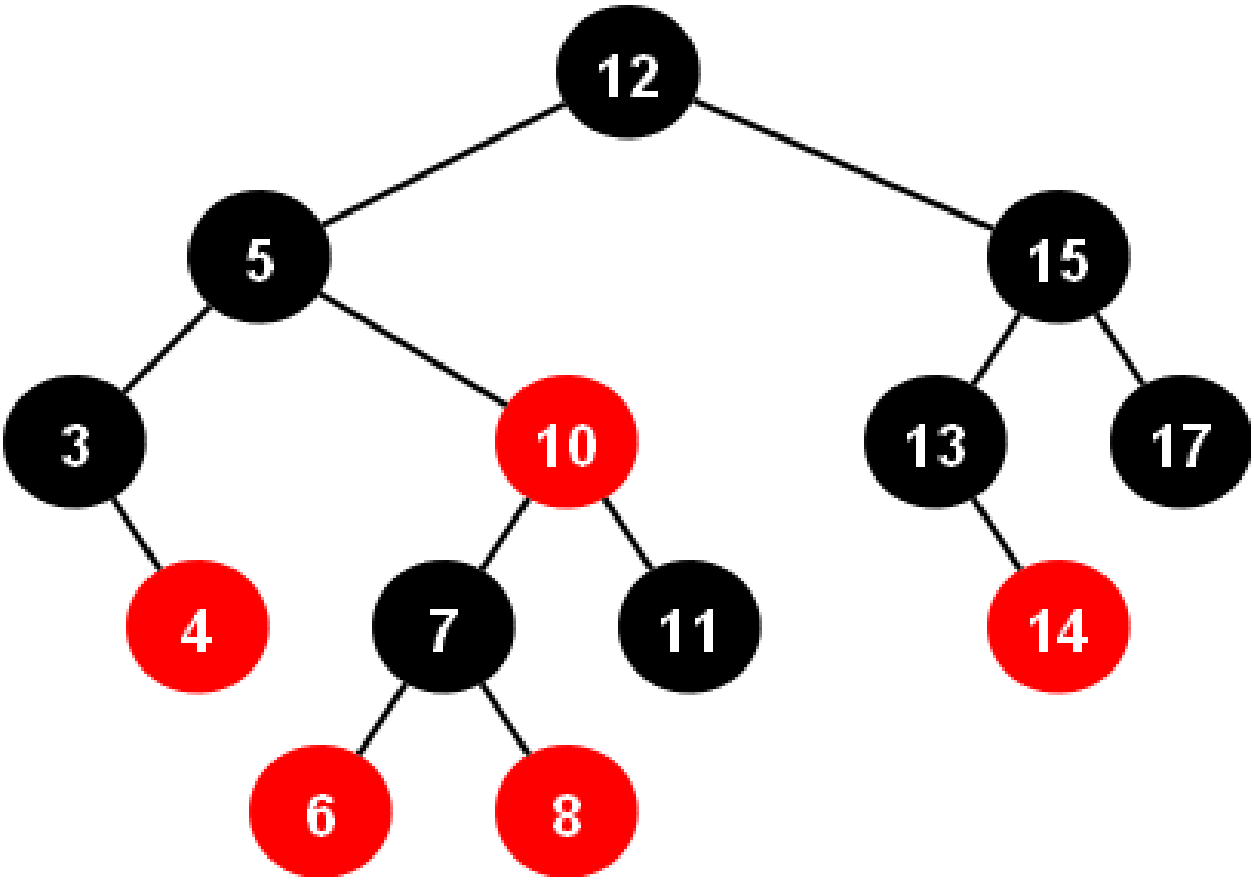
Have you ever wondered how computers can operate so efficiently? This is because computers can organize data in particular ways that are suited to many different applications. One of the ways that computers can organize data is through the use of trees. A tree is a widely used abstract data structure that simulates a hierarchical tree structure with a root value and sub-trees of children that are represented as a set of linked nodes. Trees are widely used because they can be defined recursively as a collection of nodes that start from the root. Each node is a data structure that consists of a value and a list of references to other nodes. The only constraints are that no reference is duplicated and no node ever points to the root.



Because trees are so widely used, there are many different types of trees that work best for different types of applications. The illustration above is an example of a Binary Search Tree. A Binary Search Tree is a special type of tree that follows search order property. Search order property is defined as the following: For every node x in the tree, keys of nodes in the left sub-tree are less than x's key and keys of nodes in the right sub-tree are greater than x's key. The key is the numerical value inside of each node, usually represented as an integer.

Notice from the illustration that 5 is the key of the root node and 3 < 5 so it goes on the left of the root node and 8 > 5 so it goes on the right. We also see that likewise 1 > 3 and 4 > 3 and so the nodes are positioned accordingly. This is also true on the right sub-tree as 7 > 8 is the left child and 10 > 8 is the right child.

This search order property is a critical component to what makes a Binary Search Tree. There are other trees out there that take advantage of the search order property offered by Binary Search Trees and expand upon it. One example of this is a Red-Black Tree.



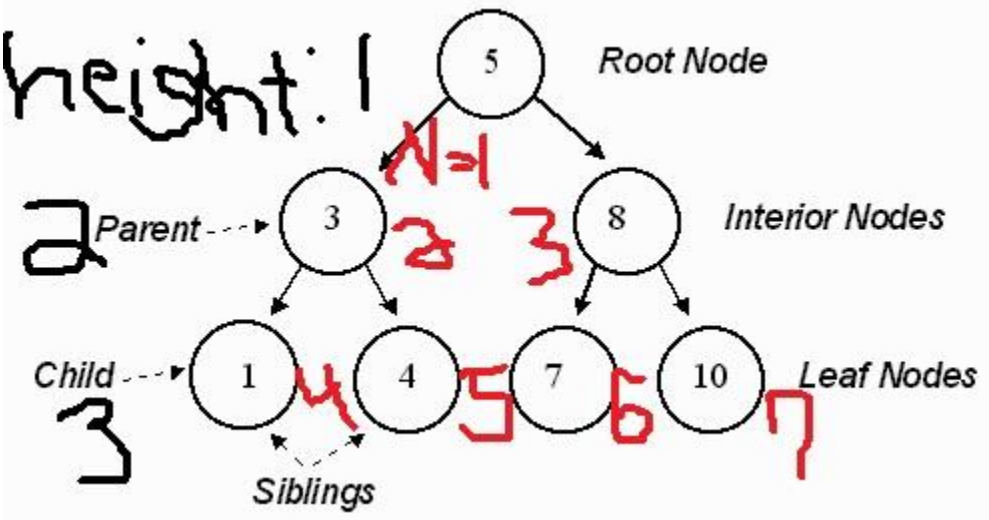
A Red-Black tree is a Binary Search Tree that has four additional properties:

1. Every node is colored either RED or BLACK.
2. The ROOT is BLACK.
3. RED nodes must have BLACK children.
4. Every path from the ROOT to a LEAF must have the same number of BLACK nodes.

These properties are very important because they give special characteristics to each node and to the tree as a whole. This is very useful, however it does make things like adding and removing nodes a bit more difficult. This is because we must now keep up with the color of each node and adjust the tree properly until all of the properties are satisfied.

The whole idea behind this project is that it is possible to write a program that will organize data to satisfy these conditions. This includes being able to define such things as the elements, Red-Black node, Red-Black Tree and being able to perform operations such as searching the tree, inserting/deleting nodes in the tree and removing the value of a node in the tree. It is also useful to be able to validate a given tree to see if it meets the required properties that define a Red-Black Tree.

Starting with search, Binary Search Trees and Red-Black Trees are highly efficient because the time-complexity it takes to search a tree is relatively small. Time-complexity is a computer science concept that quantifies the amount of time taken by an algorithm to run as a function of the length of the length of the string representing the input. The lower the time-complexity (as measured in term big O) the more time efficient that algorithm is to run. The time complexity for these trees are measured as O(logN). This is very efficient for an algorithm. Here is a list of common algorithm efficiencies: O(1) < O(logN) < O(N) < O(NlogN) < O(N^2). To prove this complexity of O(logN) let's refer back to the first illustration.



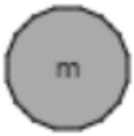
N	height	logN	Floor[X]	Floor[X]+1
1	1	0	0	1
2	2	1	1	2
3	2	1 < X < 2	1	2
4	3	2	2	3
5	3	2 < X < 3	2	3
6	3	2 < X < 3	2	3
7	3	2 < X < 3	2	3

As you can see from the chart, searching the tree by traversing the various nodes gives us a complexity that's on the order of logN. Adding another child would result in height of 4.

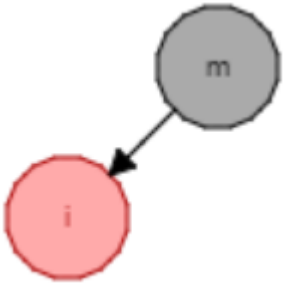
All abstract data structures are meaningless if there is no data to manage. That is why it is also important to discuss the insertion and deletion of nodes. To demonstrate insertion and deletion of nodes in a Red-Black Tree, I will be using the first 10 letters of my name as the input. I will insert a letter into the tree on 1st occurrence and delete the letter from the tree on a 2nd occurrence, alternating insertion and deletion for every subsequent occurrence. I will be evaluating these characters in alphabetical order, ignoring case. a < b < c < … < x < y < z. The characters are:

michaelgar

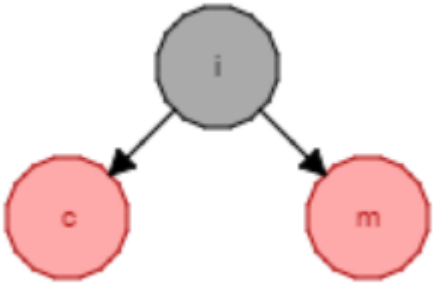
Starting with m:



Next is i:

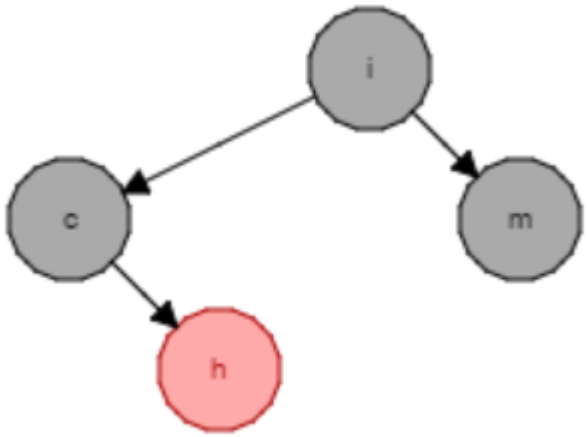


Next is c:



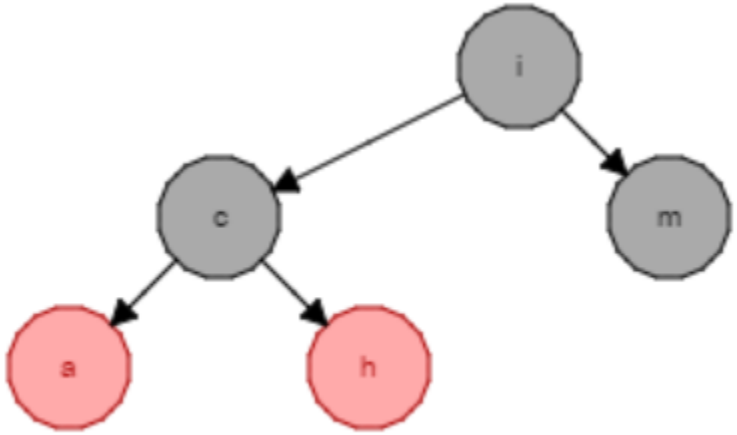
Notice that I became the new root and m because a red child on the right. This is an example of a single right rotation and recolor. Rotations and re-colorations are required to keep the Red-Black Tree properties in check (which in term keeps the tree balanced i.e. the size of each sub-tree differed by no more than 1)

Next is h:

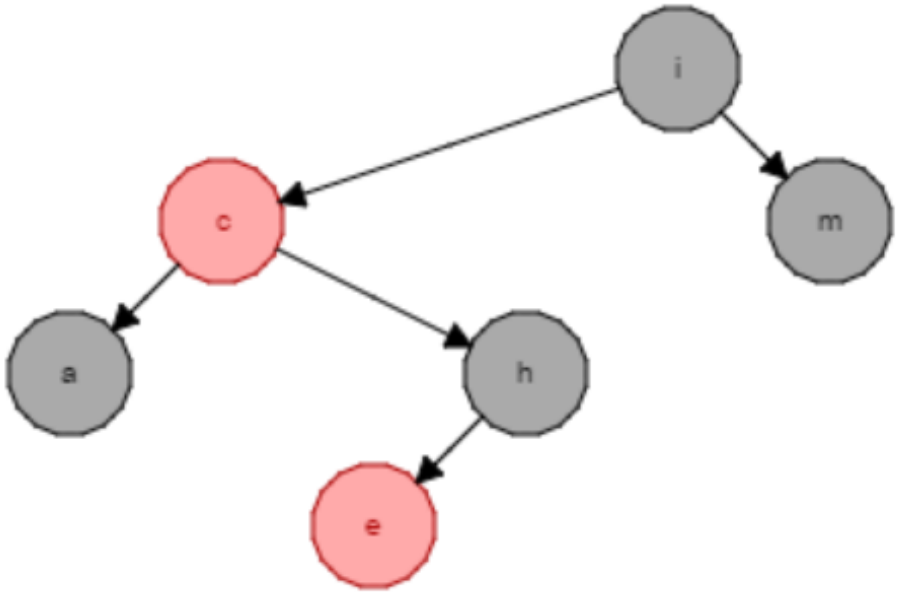


Again we see a re-coloration of the c and m nodes in order to maintain properties #3 and #4.

Next is a:

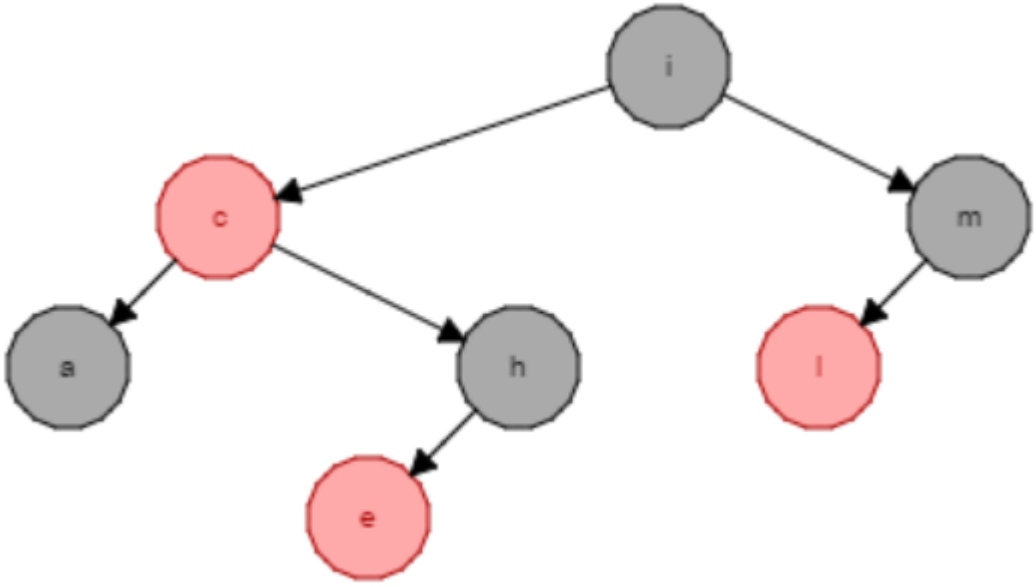


Next is e:

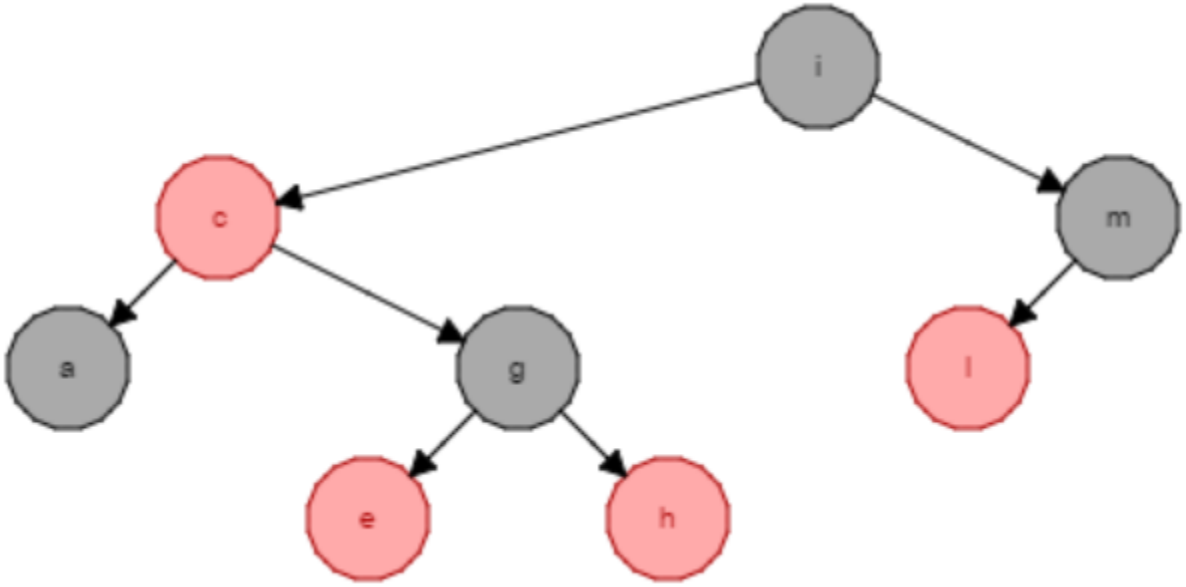


Again we see the re-coloration of nodes a, c, and h in order to maintain properties #3 and #4.

Next is l:

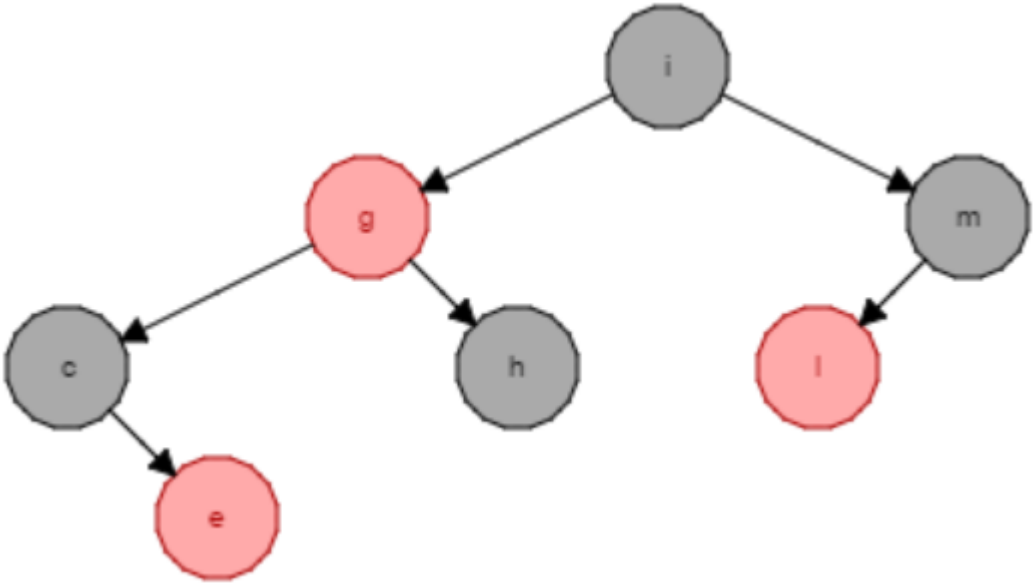


Next is g:



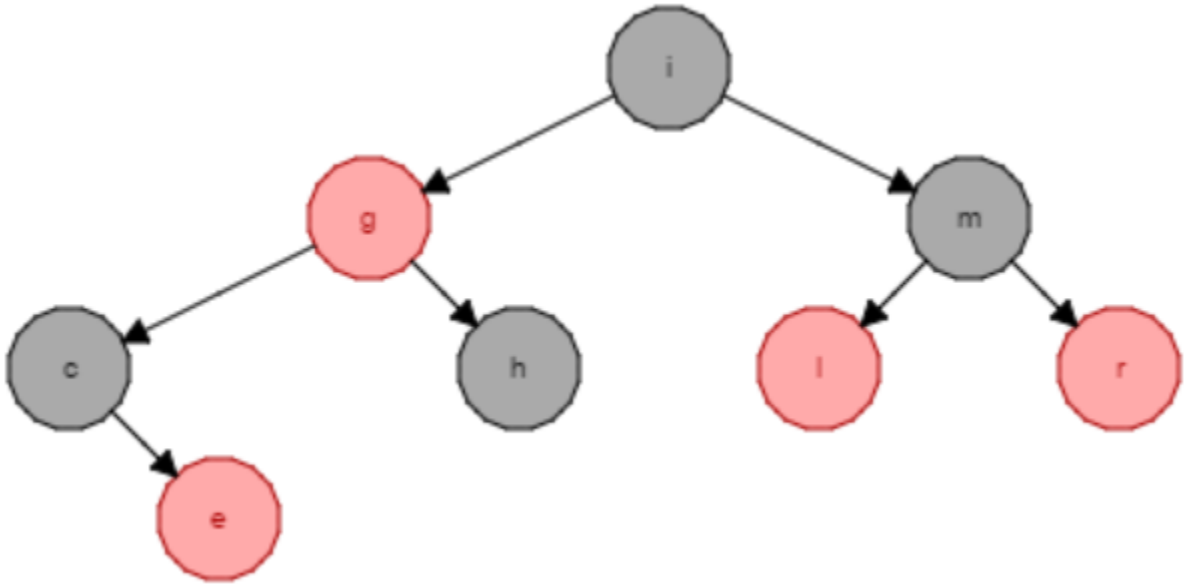
Here we see a single rotate left, followed by a single rotate right and a recolor of nodes h and g (since all insertions start as a red node).

Next is a:



Here we have a single rotate left and a recolor of the nodes c, g and h to accommodate for Property #4.

Last is r:



As you can see from this example, the more nodes that are present, the more there is to manage when additional insertions/deletions are made. The rotation and re-color operations are also vital in maintaining the properties of Red-Black Trees after performing insertions/deletions. I feel like this example really helps clarify the importance of rotation more than just trying to explain the process in words would.

Because these properties of Red-Black Trees are so important, it is very useful to have a way to validate these properties. If these validations are being made in the form of String inputs, it is very important to be able to analyze the String format to take it through the test of the Red-Black Tree properties. Not only do we have to check for the four properties, but we must also check for and maintain Search Order Property, since the Red-Black Tree is an extension to the Binary Search Tree.

While I knew all of this going into the creation of the RBTValidator class that checks these properties, I was not sure how to process the String in a way that would maintain all of this information. However, I did check for the first property which will be present in the first line of the given string. This means that

any string given that starts with “root:RED” is immediately invalid because the root node must be black. However, just because the first line starts with “root:BLACK” doesn’t mean that it’s a valid Red-Black Tree because the other properties may fail, including search-order property. Idealistically, my RBTValidator would have been able to validate trees of any size rather than just trees of size 1.

I have learned a whole lot after going through this assignment. I think what I have learned the most is that you definitely should not procrastinate when it comes to programming. You may know the general concepts, but turning those concepts into executable code is something different entirely that takes very deep understanding of not only the subject material but also the language and syntax of which the code must be written in.

I found that while the concepts behind Red-Black Trees are fairly easy to grasp, the coding was not. There are so many factors that go into the coding that it takes a lot of time to sit and think logically through the problem in a way that can be represented by code.

My biggest regret about completing this project was that I took the posted Check-Ins for granted. I thought that I could blast my way through the project in the final week and that was not the case at all. I missed out on the key feedback that I could have used to be 100% sure that my toString method was functional and that I could pass some of the test evaluations. Unfortunately the result feels like a rushed piece of work that doesn’t truly represent what I know and what I can offer. However, I am really glad that I have had this experience now instead of later in my career. Ultimately, I feel like I learned a really valuable life lesson from his project!

Works Cited

Pictures from:

http://www.powayusd.com/pusdtbes/images/tree_example.gif

http://cs.lmu.edu/~ray/images/redblacktree.png

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Definitions and Code samples:

Weiss, M. (2010). *Data structures & problem solving using Java* (4th ed.). Boston: Pearson/Addison Wesley.

Data Structures and Abstractions with Java (3rd rev. ed.). (2011). Upper Saddle River: Pearson Education (US).