30/34
A
MAY 2018

Computer Science Extended Essay:

Investigating the time complexities of the AVL Tree and Red-Black Tree insertion algorithms

Research question:

How does the **re-balancing algorithm efficiency** of an ***Adelson-Velskii and Landis Tree*** compare to that of a ***Red-Black Tree*** in terms of **time complexity** upon **insertion of values**?

Essay word count: 3997

Table of Contents

1. Introduction

This essay will focus on the structure of binary search trees, a relatively complex data structure which can be very useful in many applications. This essay will specifically look into the *Adelson-Velskii and Landis* (*AVL*) *Tree* and the *Red-Black Tree*, which are two types of binary search tree. Given a set of values inserted into both trees, the *time complexity* for the insertion operations for both trees will be investigated. *Time complexity* is the term used to refer to the amount of time taken for an algorithm to run given a set of input values of a certain size[1]. Hence, the question: *How does the* ***re-balancing efficiency*** *of an* ***Adelson-Velskii and Landis Tree*** *compare to that of a* ***Red-Black Tree*** *in terms of* ***time complexity*** *upon* ***insertion of values****?* This area links to Topic 5 of the IB Higher Level Computer Science course.

2. Theory

2.1 Binary Search Trees

A binary search tree is a data structure with a defined behavior and is the basis of the two trees being looked into. The word *binary* refers to "being composed of two things"[2]. For trees, it means each item in a tree must point to a maximum of two other items (referred to as *children*). This means zero children or one child are also allowed. An item of a binary search tree is commonly referred to as a *node*. This term will be used for the remainder of the essay to describe values in trees.

---

[1] Adamchik, V. S., 2009. *Algorithmic Complexity.* [Online]
Available at: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html
[Accessed June 2017].

[2] Dictionary.com Unabridged, n.d. *Binary.* [Online]
Available at: http://www.dictionary.com/browse/binary
[Accessed May 2017].

A binary search tree must choose where to place a value when it is inserted. Each node in the tree can be compared in some way (for example, numbers by size and text lexicographically). Each node will have a left child pointer and a right child pointer, which points to another node in the tree. The left child of a node must have a value 'less than' the node, which means the right child must have a value 'greater than' the node.

When inserting a node:

1. If the tree is empty, the root of the tree (the top value or more commonly known as the **root node**) is set to this node.
2. If a root node exists and its value is 'greater than' the value being inserted, the same process will occur with the root's left child.
3. If a root node exists and its value is 'less than' the value being inserted, the same process will occur with the root's right child.
4. This will occur until there is a position in the tree where a left child or right child doesn't exist for a node. This will be where the new node is placed.

An example of a typical Binary Data Tree is shown in ***Figure 2.1.1*** below.

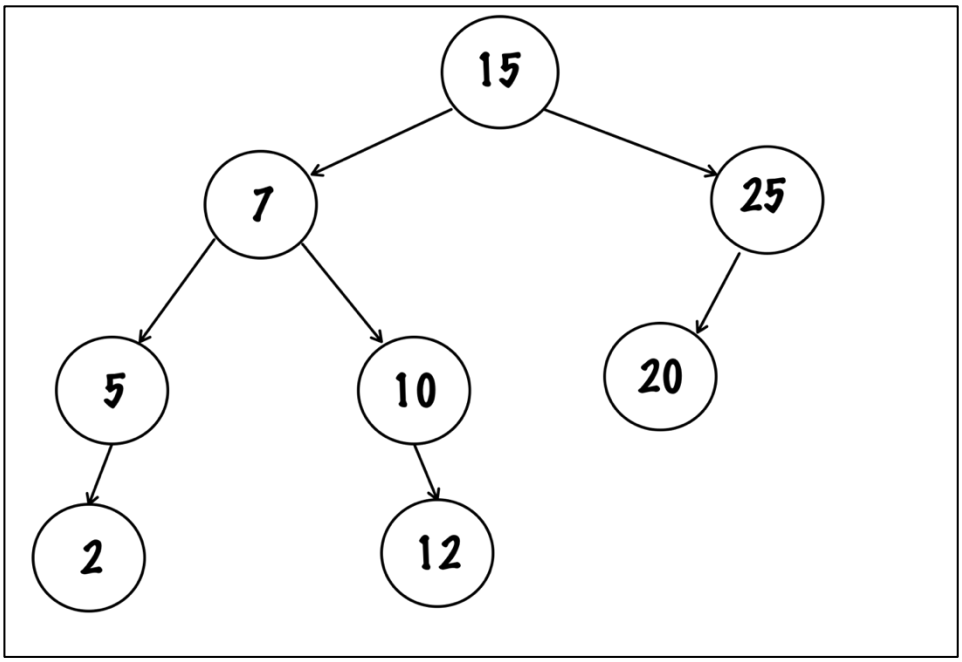

Figure 2.1.1: Example of a Binary Data Tree

The 'Search' in 'Binary Search Tree' comes from the main purpose of using the structure in the first place: searching it. Searching follows a similar process to insertion.

By organizing nodes in this structure, searching for values can happen very efficiently compared to, say, a linear search. A notation used to measure the worst-case efficiency of an algorithm is the *Big-O notation*[3]. For an array, it is $O(N)$, where N is the size of the array. For binary search trees, the searching efficiency is $O(\log_2 N)$, which is a massive difference compared to $O(N)$ as shown in **Figure 2.1.2** below.

[3] Massachusetts Institute of Technology, 2003. *Big O Notation.* [Online] Available at: http://web.mit.edu/16.070/www/lecture/big_o.pdf [Accessed June 2017].

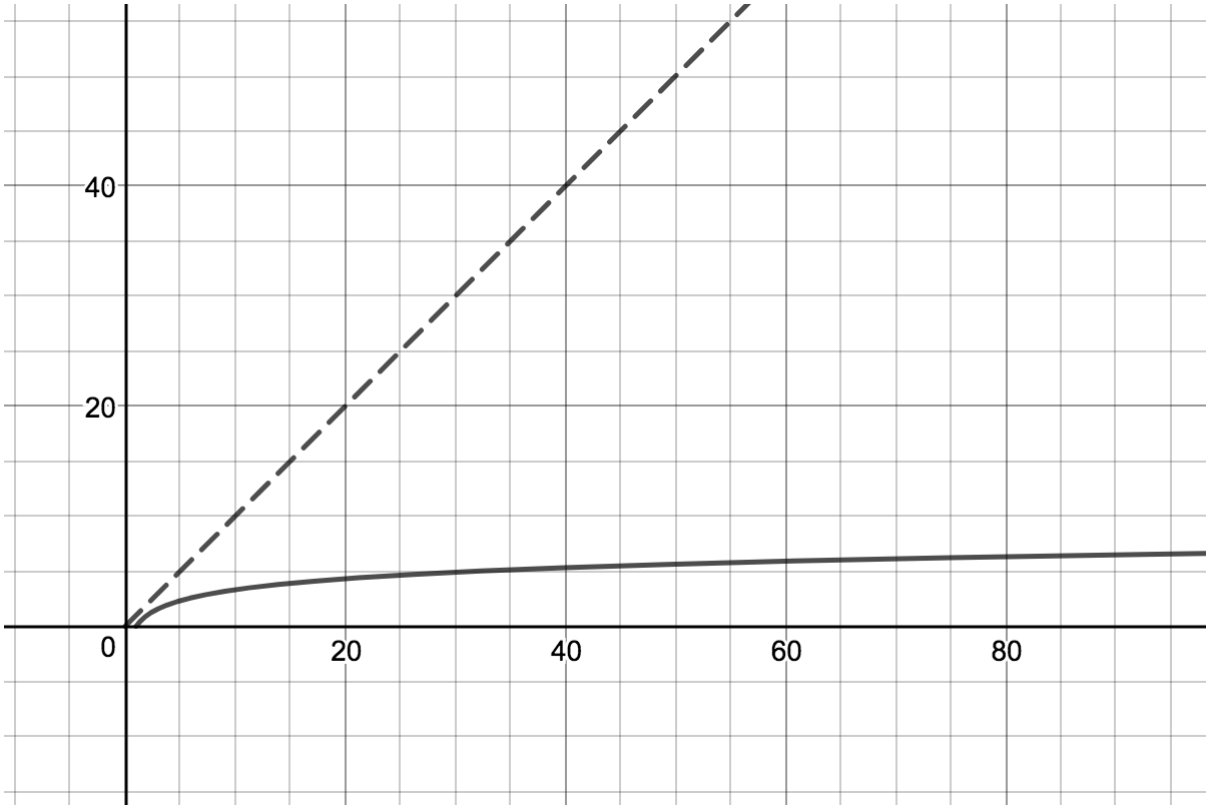

Figure 2.1.2: A graph to show the worst-case number of searches for the two data structures (y-axis) over the number of values stored (x-axis)

Dashed line – Array
Solid line – Binary Search Tree

While this seems like an amazing feature of binary search trees, consider the insertion of the values: 1, 2, 3, 4, 5 in that order. The fact is a tree like this would form:

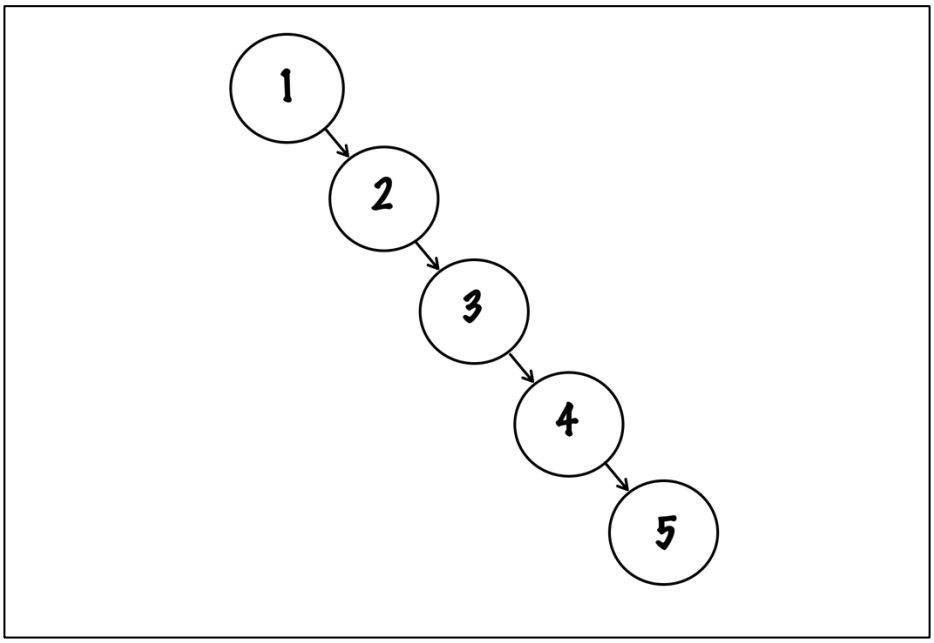

Figure 2.1.3: An unbalanced binary data tree

As seen, this structure looks similar to a regular list. In fact, for worst-case scenario, the efficiency would be $O(5)$, as like a linear search. This would not be considered a binary search tree.

In order to solve the problem, trees will have **balancing algorithms** implemented in order to maintain the structure they need (reducing the number of searches required for a value). Different implementations of the same structure can be given, where both have the same behavior but a different method in ensuring this behavior. This will mean that some implementations are bound to be better than others in certain ways. Taking the balancing algorithm into account, both AVL Trees and Red-Black Trees have different definitions of how they go about balancing themselves. These will be explored in detail below.

The AVL Tree and Red-Black Tree algorithms used in the sections below were retrieved from online. I have requested and received permission from the creator of the algorithms: Dr. Mark Allen Weiss. The permission letter and reply can be found in ***Appendix D*** (page 44).

2.2 Adelson-Velskii and Landis Tree

An Adelson-Velskii and Landis (AVL) Tree makes use of a *height-balance property*, which states that, for each node, the *height difference* of the children of that node differ by 1 at most[4]. This means if any height difference is more than 1, the tree is considered to be unbalanced. The term *height difference* refers to the difference in height between the child nodes on the left side of a node and the right side, where a *height* refers to the number of nodes in the longest path

[4] Goodrich, M. T., Tamassia, R. & Goldwasser, M. H., 2014. "11.3 AVL Trees" pg. 490, *Data Structures and Algorithms in Java.* Sixth Edition ed. s.l.:Wiley.

from a node down to a *leaf* (inclusive). Note that the term *leaf* is used to denote a node with no children. Each side of a node will have a height, which means the height difference will be the absolute value of the left height minus the right height (or vice versa). Some implementations will give one side negative unit values for height and sum the values of the left and right heights to obtain the difference. For an AVL Tree to be balanced, **all** nodes must have a height difference of 0 or 1.

Now the AVL Tree algorithm will be looked into more closely. Please refer to the Java code in ***Appendix A1*** (page 33) and ***Appendix A2*** (page 37) for the AVL Tree algorithm being examined.

The *insert()* function from *AvlTree.java* is shown below:

```
01.  private AvlNode insert(Comparable x, AvlNode t) {
02.      if (t == null)
03.          t = new AvlNode(x, null, null);
04.      else if (x.compareTo(t.element) < 0) {
05.          t.left = insert(x, t.left);
06.          if (height(t.left) - height(t.right) == 2)
07.              if (x.compareTo(t.left.element) < 0)
08.                  t = rotateWithLeftChild(t);
09.              else
10.                  t = doubleWithLeftChild(t);
11.      } else if (x.compareTo(t.element) > 0) {
12.          t.right = insert(x, t.right);
13.          if (height(t.right) - height(t.left) == 2)
14.              if (x.compareTo(t.right.element) > 0)
15.                  t = rotateWithRightChild(t);
16.              else
17.                  t = doubleWithRightChild(t);
18.      } else
19.          ;  // Duplicate; do nothing
20.      t.height = max(height(t.left), height(t.right)) + 1;
21.      return t;
22.  }
```

Figure 2.2.1: AvlTree insert() function[5]

[5] Weiss, M. A., n.d. *AvlTree.java.* [Online]
Available at: https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java
[Accessed January 2017].

Referring to **Figure 2.2.1**, the *insert()* function takes a recursive approach to inserting values into the tree. The parameter *x* refers to the value to be inserted and the parameter *t* refers to the current node, starting with the root node.

How the *insert()* function restructures the tree after insertion depends on the *height* property of the nodes. Balancing is required when the condition on **line 6** or **line 13** is true. That is, when the height difference of the node *t* is equal to 2. When this condition is satisfied, two possible methods of restructuring are possible depending on the condition on **line 7** or **line 14**. Note that restructuring, if required, will occur after the value is actually inserted on **line 5** or **line 12**.

For the situation where a node is inserted to the left or right child of the node *t* and the height different of *t* is 2 (**line 6** or **line 13** from ***Figure 2.2.1*** is true):

Note: for all AVL tree diagrams below, the number at the top-right of a node is the height difference of that node.

1. If the value is being inserted to the left of *t* (**line 4** from ***Figure 2.2.1*** is true) and *x* is less than the value of the left child of *t* (**line 7** from ***Figure 2.2.1*** is true) the function *rotateWithLeftChild()* will be executed and *t* will be set to the function's return value. The Java code for this function is shown below.

```
01.  private static AvlNode rotateWithLeftChild(AvlNode k2) {
02.      AvlNode k1 = k2.left;
03.      k2.left = k1.right;
04.      k1.right = k2;
05.      k2.height = max(height(k2.left), height(k2.right)) + 1;
06.      k1.height = max(height(k1.left), k2.height) + 1;
07.      return k1;
08.  }
```

Figure 2.2.2: AvlTree rotateWithLeftChild() function[6]

[6] Weiss, M. A., n.d. *AvlTree.java.* [Online]
Available at: https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java
[Accessed January 2017].

This will rotate the subtree of root *t* such that the new root is *t*'s left child and the original node *t* is this root's right child. A diagram illustrating this is shown below (where **A** is inserted into a tree which has values **C** and **B**):

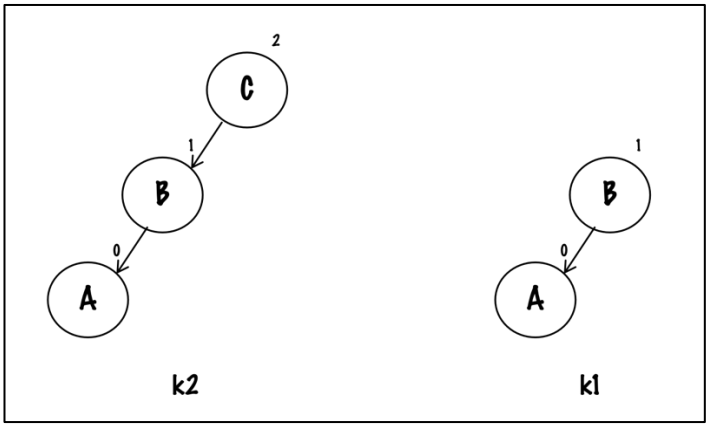

Figure 2.2.3: k2 and k1 after line 2 from Figure 2.2.2 is executed

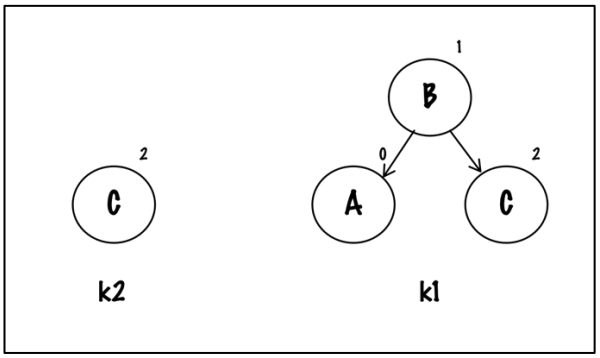

Figure 2.2.4: k2 and k1 after line 3 and line 4 from Figure 2.2.2 are executed

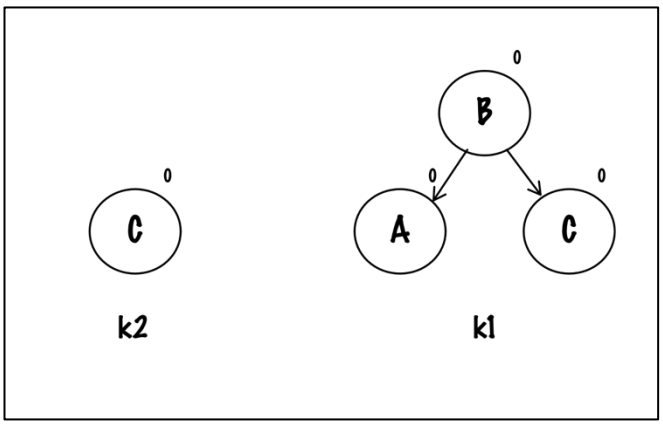

Figure 2.2.5: k2 and k1 after line 5 and line 6 from Figure 2.2.2 are executed

So the tree *k2* from **Figure 2.2.5** is returned.

2. If the value is being inserted to the right of *t* (**line 11** from ***Figure 2.2.1*** is true) and *x* is greater than the value of the right child of *t* (**line 14** from ***Figure 2.2.1*** is true), the function *rotateWithRightChild()* will be executed and *t* will be set to the function's return value. The Java code for this function is shown below.

```
01.  private static AvlNode rotateWithRightChild(AvlNode k1) {
02.      AvlNode k2 = k1.right;
03.      k1.right = k2.left;
04.      k2.left = k1;
05.      k1.height = max(height(k1.left), height(k1.right)) + 1;
06.      k2.height = max(height(k2.right), k1.height) + 1;
07.      return k2;
08.  }
```

Figure 2.2.6: AvlTree rotateWithRightChild() function[7]

This will rotate the subtree of root *t* such that the new root is *t*'s right child and the original node *t* is this root's left child. A diagram illustrating this is shown below (where **C** is inserted into a tree which has values **A** and **B**):

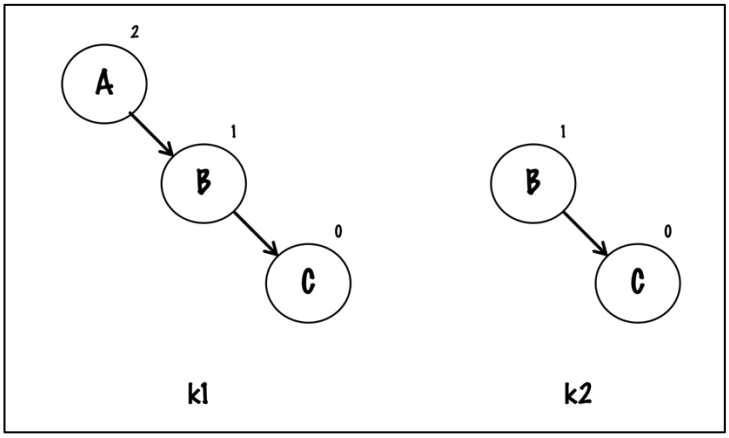

Figure 2.2.7: k1 and k2 after line 2 from Figure 2.2.6 is executed

[7] Weiss, M. A., n.d. *AvlTree.java.* [Online]
Available at: https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java
[Accessed January 2017].

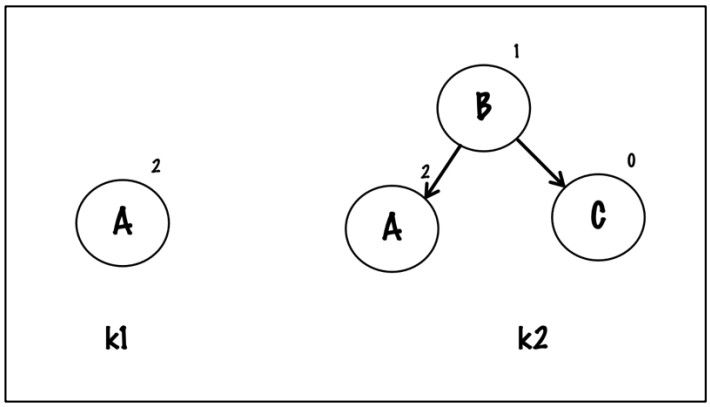

Figure 2.2.8: k1 and k2 after line 3 and line 4 from Figure 2.2.6 are executed

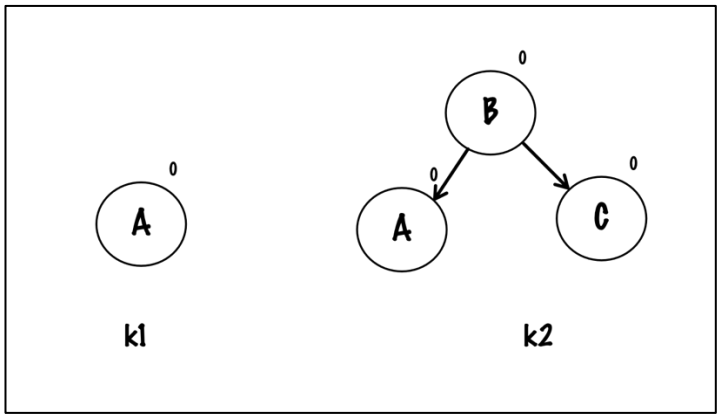

Figure 2.2.9: k1 and k2 after line 5 and line 6 from Figure 2.2.6 are executed

So the tree *k2* from **Figure 2.2.9** is returned.

3. If the value is being inserted to the left of *t* (**line 4** from ***Figure 2.2.1*** is true) and *x* is greater than the value of the left child of *t* (**line 7** from ***Figure 2.2.1*** is false) the function *doubleWithLeftChild()* will be executed and *t* will be set to the function's return value. The Java code for this function is shown below.

```
01.  private static AvlNode doubleWithLeftChild(AvlNode k3) {
02.      k3.left = rotateWithRightChild(k3.left);
03.      return rotateWithLeftChild(k3);
04.  }
```

Figure 2.2.10: AvlTree doubleWithLeftChild() function[8]

[8] Weiss, M. A., n.d. *AvlTree.java.* [Online]
Available at: https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java
[Accessed January 2017].

This will rotate the subtree of root *t*'s left child by the left child's right child and then rotate *t* by its left child. A diagram illustrating this is shown below (where **B** is inserted into a tree which has values **C** and **A**):

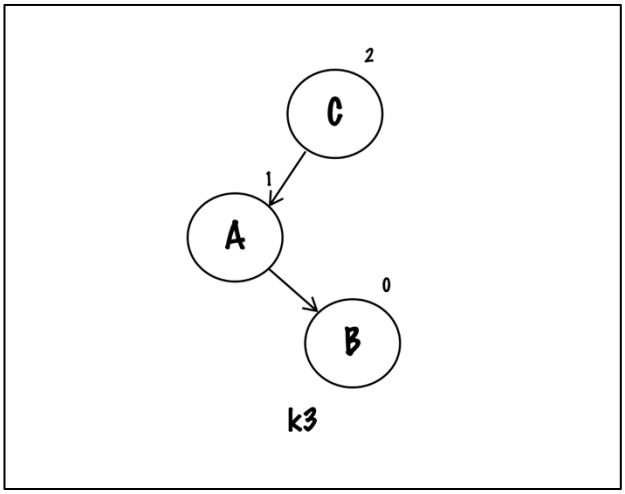

Figure 2.2.11: initial value of k3 from Figure 2.2.10

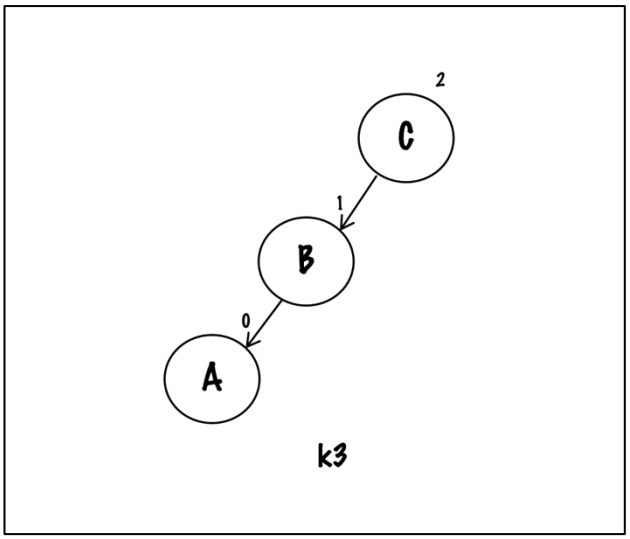

Figure 2.2.12: k3 after line 2 from Figure 2.2.10 is executed

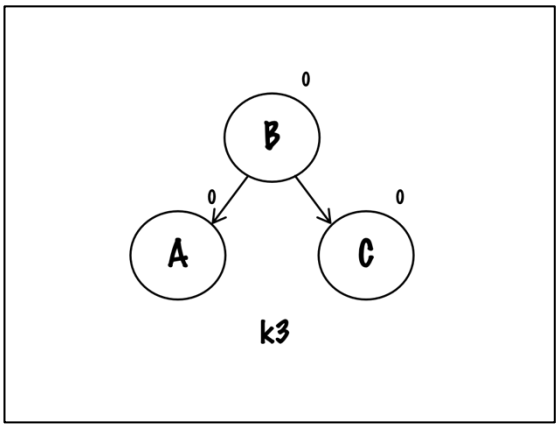

Figure 2.2.13: k3 after line 3 from Figure 2.2.10 is executed

So the value of *k3* from **Figure 2.2.13** is returned.

4. If the value is being inserted to the right of *t* (**line 11** from ***Figure 2.2.1*** is true) and *x* is less than the value of the right child of *t* (**line 14** from ***Figure 2.2.1*** is false), the function *doubleWithRightChild()* will be executed and *t* will be set to the function's return value. The Java code for this function is shown below.

```
01.  private static AvlNode doubleWithRightChild(AvlNode k1) {
02.      k1.right = rotateWithLeftChild(k1.right);
03.      return rotateWithRightChild(k1);
04.  }
```

Figure 2.2.14: AvlTree doubleWithRightChild() function[9]

This will rotate the subtree of root *t*'s right child by the right child's left child and then rotate *t* by its right child. A diagram illustrating this is shown below (where **B** is inserted into a tree which has values **A** and **C**):

[9] Weiss, M. A., n.d. *AvlTree.java.* [Online]
Available at: https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java
[Accessed January 2017].

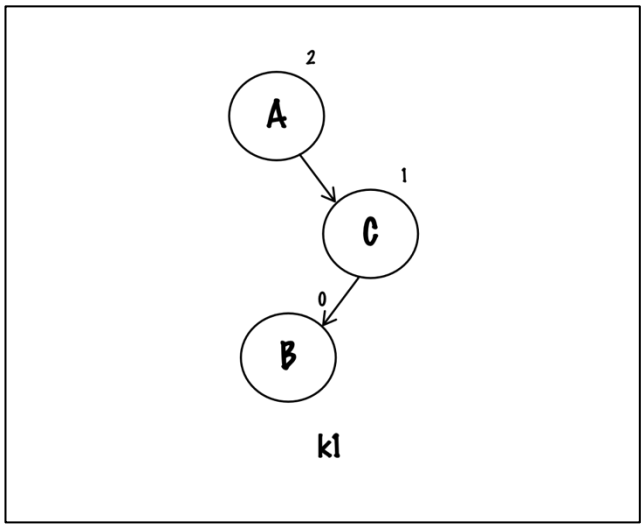

Figure 2.2.15: initial value of k1 from Figure 2.2.14

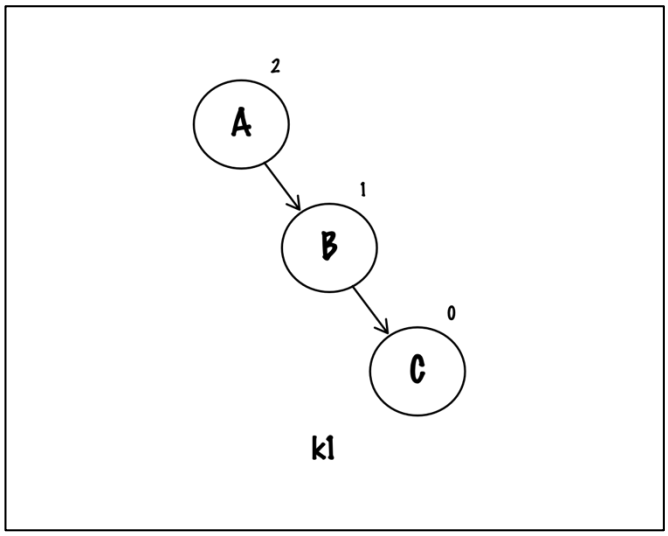

Figure 2.2.16: k1 after line 2 from Figure 2.2.14 is executed

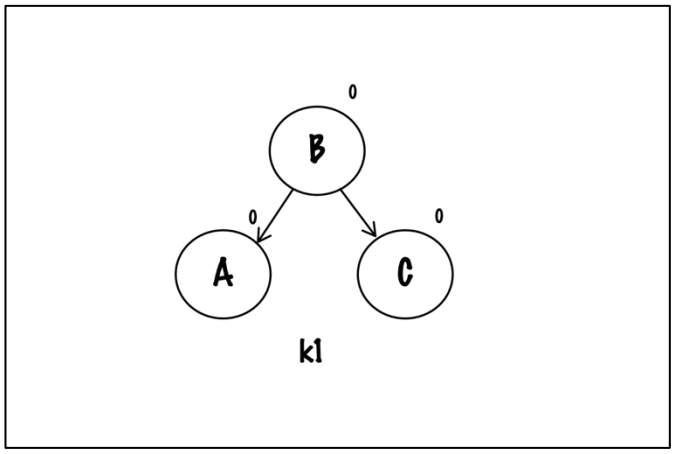

Figure 2.2.17: k1 after line 3 from Figure 2.2.14 is executed

So the value of *k1* from **Figure 2.2.7** is returned.

2.3 Red-Black Tree

A Red-Black Tree makes use of a number of rules which must be followed to maintain balance of its nodes. Each node can be colored **red** or **black** (as the name suggests) and the rules which must be followed relate to the coloring of nodes. Note that each inserted node is red by default. The red-black rules are listed below:

1. **The root node is always black.** If any restructuring occurs such that the root node is changed, it is important to ensure that the new root node is colored black.
2. **The children of a red node must be black.** For a value inserted as a child of a red node, it must be colored black. In all other cases, an inserted node is colored red.
3. **All paths from the root to a leaf of the tree have the same *black depth***. This means that there must be the same number of black nodes for each and every path.[10]

Re-balancing of nodes will occur if one of the rules is broken. For example, if a node is inserted as a child of a red node (which itself is inserted as a red node), restructuring will need to occur since the **2nd rule** is violated.

When a node, ***X***, is inserted and restructuring is required (a rule is violated), there are two possible situations which can occur and each situation will have a different approach to re-balancing the area of the tree which requires it. Let ***P*** be the parent of ***X***, let ***S*** be the sibling of the parent and let ***G*** be the grandparent of ***X***.

1. **If S is black or null**, then restructuring followed by re-coloring occurs:

[10] Goodrich, M. T., Tamassia, R. & Goldwasser, M. H., 2014. *Data Structures and Algorithms in Java.* Sixth Edition ed. s.l.:Wiley.

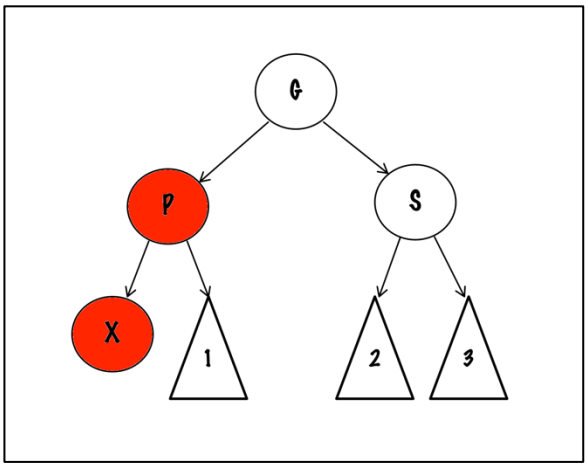

Figure 2.3.1 (a): Red-Black Tree with X inserted

X is inserted as a red node and the red child rule is violated.

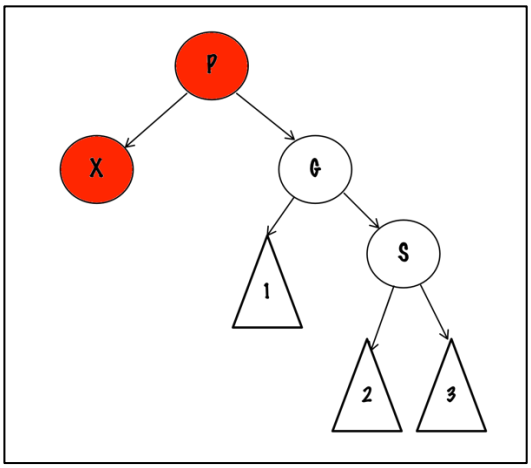

Figure 2.3.1 (b): Red-Black Tree restructured

From ***X***, ***P*** and ***G*** put in order from lowest to highest, the middle value, ***P***, is selected to be the new root node of the subtree and becomes a parent of the other two values: ***X*** and ***G***. ***G*** will inherit ***P***'s left subtree: ***subtree 1*** as its left child.

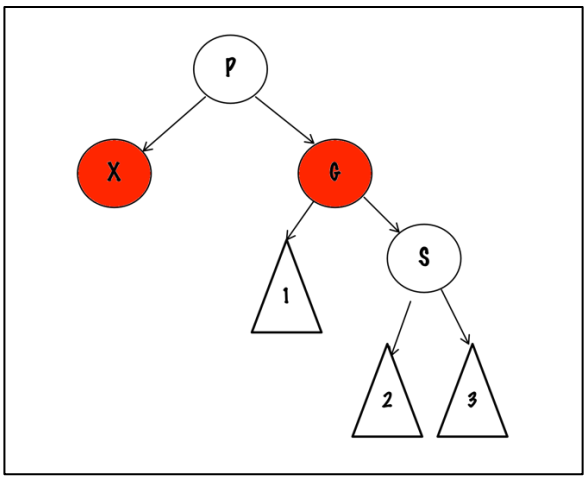

Figure 2.3.1 (c): Red-Black Tree re-colored

Finally, re-coloring occurs so that the tree can follow the rules set.

Another example is shown below:

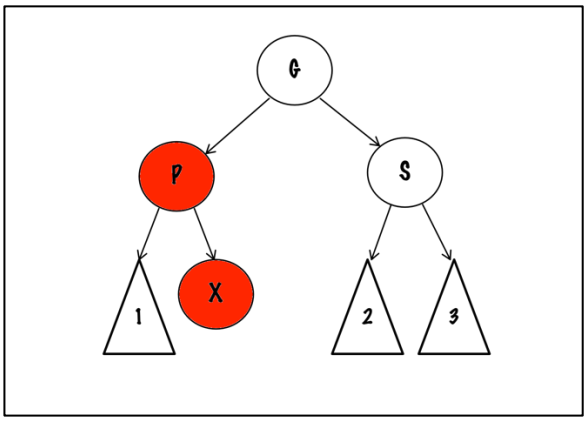

Figure 2.3.2 (a): Red-Black Tree with X inserted

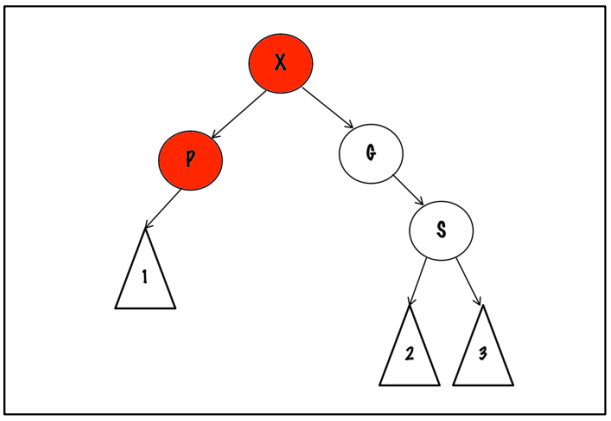

Figure 2.3.2 (b): Red-Black restructured

X is inserted as a red node and the red child rule is violated.

From ***X***, ***P*** and ***G*** put in order from lowest to highest, the middle value, ***X***, is selected to be the new root node of the subtree and becomes a parent of the other two values: ***P*** and ***G***.

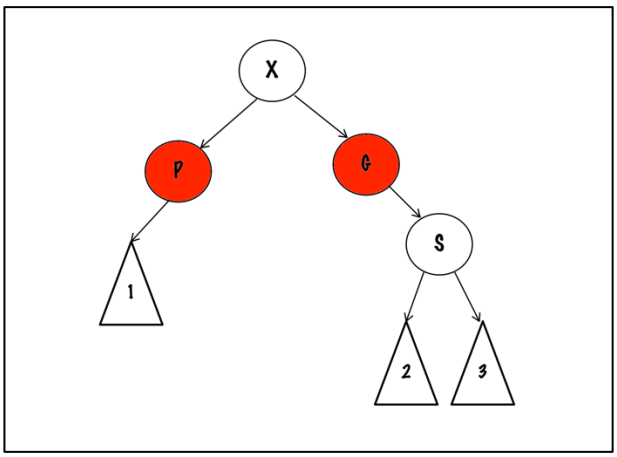

Figure 2.3.2 (c): Red-Black Tree re-colored

Finally, re-coloring occurs so that the tree can follow the rules set.

2. **If *S* is red**, then there is only need for re-coloring to ensure the tree follows the rules.

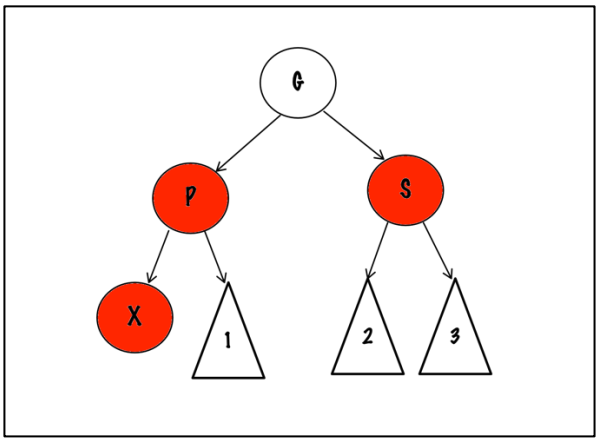

Figure 2.3.3 (a): Red-Black Tree with X inserted

X is inserted as a red node and the red child rule is violated.

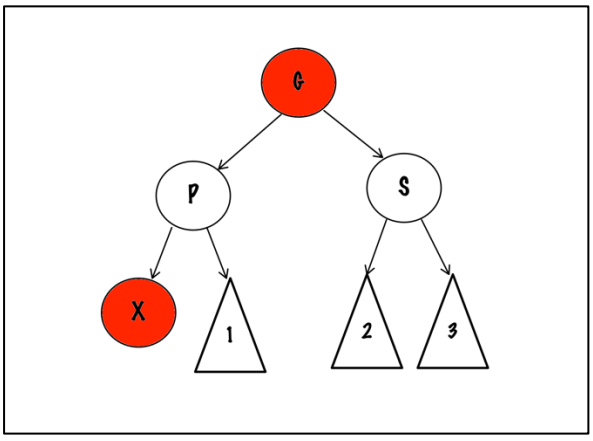

Figure 2.3.3 (b): Red-Black Tree re-colored

Since ***S*** is red, re-coloring is done on ***P***, ***G*** and ***S*** to ensure the tree follows the rules.

The only exception for this situation is if ***G*** is the root node of the tree. If this is the case, it must be colored black in order to follow the black root rule.

After successive insertions, the tree is gradually re-balanced. Seeing the Red-Black Tree algorithm, it may be apparent that the AVL Tree algorithm takes more care in re-balancing itself. This may be the case. This point will be discussed further in **section 3** of the essay.

Now the Red-Black Tree algorithm will be looked into more closely. Please refer to the Java code in ***Appendix A3*** (page 38) and ***Appendix A4*** (page 42) for the Red-Black Tree algorithm being examined.

The next part will not be looked into in as much detail as the AVL Tree algorithm was. This is because the Red-Black Tree algorithm has been described in theory in a good amount of detail above. Additionally, the implementation below does not handle the restructuring in exactly the same way as described above. However, the same result will be obtained and with the same processes as above (such as rotations and re-colorings).

Firstly, it is important to know that there are variables defined for the RedBlackTree Java class which are used in the insertion operation. These will not be explained in much detail as they are only either constants or temporary variables used to aid the insertion process as well as some other processes which will not be looked into.

```
01.  private RedBlackNode header;
02.  private static RedBlackNode nullNode;
03.
04.  static {// Static initializer for nullNode
05.      nullNode = new RedBlackNode(null);
06.      nullNode.left = nullNode.right = nullNode;
07.  }
08.
09.  static final int BLACK = 1;    // Black must be 1
10.  static final int RED = 0;
11.
12.  // Used in insert routine and its helpers
13.  private static RedBlackNode current;
14.  private static RedBlackNode parent;
15.  private static RedBlackNode grand;
16.  private static RedBlackNode great;
```

Figure 2.3.4: RedBlackTree constants and temporary variables function[11]

The only important thing to note here is the *header* variable, which points to the root of the tree, must be set to the lowest possible comparable value when the RedBlackTree class is instantiated (for example, if 32-bit integers are to be inserted into the tree, the predefined constant: *Integer.MIN_VALUE* (which is approximately -2.15 billion) should be used).

The *insert()* function is shown below:

[11] Weiss, M. A., n.d. *RedBlackTree.java.* [Online]
Available at:
https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/RedBlackTree.java
[Accessed January 2017].

```
01.  public void insert(Comparable item) {
02.      current = parent = grand = header;
03.      nullNode.element = item;
04.
05.      while (current.element.compareTo(item) != 0) {
06.          great = grand;
07.          grand = parent;
08.          parent = current;
09.          current = item.compareTo(current.element) < 0 ?
10.                  current.left : current.right;
11.
12.          // Check if two red children; fix if so
13.          if (current.left.color == RED && current.right.color == RED)
14.              handleReorient(item);
15.      }
16.
17.      // Insertion fails if already present
18.      if (current != nullNode)
19.          return;
20.      current = new RedBlackNode(item, nullNode, nullNode);
21.
22.      // Attach to parent
23.      if (item.compareTo(parent.element) < 0)
24.          parent.left = current;
25.      else
26.          parent.right = current;
27.      handleReorient(item);
28.  }
```

Figure 2.3.5: RedBlackTree insert() function[12]

For this implementation, the function uses a loop (***lines 5-15*** from **Figure 2.3.5**) to determine where in the tree to insert the node. The important restructuring part, however, is in the *handleReorient()* function, which is called both in the loop when the *current* node's children are red and at the end of the *insert()* function. The function is shown below:

[12] Weiss, M. A., n.d. *RedBlackTree.java.* [Online]
Available at:
https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/RedBlackTree.java
[Accessed January 2017].

```
01.  private void handleReorient(Comparable item) {
02.      // Do the color flip
03.      current.color = RED;
04.      current.left.color = BLACK;
05.      current.right.color = BLACK;
06.
07.      if (parent.color == RED){   // Have to rotate
08.          grand.color = RED;
09.          if ((item.compareTo(grand.element) < 0) !=
10.                  (item.compareTo(parent.element) < 0))
11.              parent = rotate(item, grand);  // Start dbl rotate
12.          current = rotate(item, great);
13.          current.color = BLACK;
14.      }
15.      header.right.color = BLACK; // Make root black
16.  }
```

Figure 2.3.6: RedBlackTree handleReorient() function[13]

The *handleReorient()* function will handle both re-coloring and rotations for nodes to maintain balance. To explain the program in **Figure 2.3.6**, ***lines 3-5*** will handle the re-coloring for the current node and its children. As seen with the explained theory above, there will be a color change for the *grandparent* if the *parent* is red (red-child rule violation). Then, depending on the comparable properties of the *item* being inserted and the *current* node's *parent* and *grandparent* (***lines 9-10***), there will be a rotation by the *parent* and the *grandparent*, the value of which is returned and set to the *parent* node. Then, the current node will be set to the return value of a rotation by the *great grandparent*. Although *great grandparent* wasn't mentioned when the theory was described in detail, it is used in the implementation in order for a rotated subtree with a *grandfather* root to be set as a child of the *great grandfather* (please refer to ***Appendix A3*** for further information about how this works). Finally, the header color is set to black to satisfy the black root rule.

The *rotate()* function consists of code for a standard Red-Black rotation. This will not be explained in detail. More information about this can be found in ***Appendix A3***.

[13] Weiss, M. A., n.d. *RedBlackTree.java.* [Online]
Available at:
https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/RedBlackTree.java
[Accessed January 2017].

3. Hypothesis and Applied Theory

The theory of both tree algorithms have been described and explained in a good amount of detail. Now it is important to consider which of the two algorithms is most efficient. Time complexity was brought up at the beginning of this essay but not applied when the two algorithms were explored. An experiment will be carried out to measure the time taken for each tree to add sets of values. It should be said that both trees have an efficiency of $O(\log_2 N)$ for insertion[14]. However, this efficiency is for the physical insertion of the values and doesn't take into account the re-balancing required after it. This experiment *will* take into account the re-balancing required after insertions as time is physically being measured.

It was mentioned earlier that the AVL tree may take more care in ensuring that it is balanced. This is due to the AVL Tree algorithm checking the height difference of all traversed nodes after each insertion and re-balancing itself if any height difference is 2. The RedBlack Tree algorithm, however, re-balances and/or recolors nodes based on violated rules. Hence, the RedBlack Tree algorithm will do less physical restructuring than the AVL Tree algorithm.

The experiment will measure the relationship between **time**, ***y***, and **size of sets being inserted**, ***x***. By varying the size of the sets of values inserted into the tree, a clear relationship between these two variables should be determined and how this relationship differs between the AVL tree and the Red-Black tree should be seen.

I hypothesize that there will be a **logarithmic relationship** between ***x*** and ***y*** as described above. I also believe that the Red-Black tree will insert values and re-balance itself in a **lower time**

[14] Rowell, E., n.d. *Know Thy Complexities.* [Online]
Available at: http://bigocheatsheet.com/
[Accessed April 2017].

than the AVL tree for all sets of data. Since the efficiencies of the whole of both insertion processes are being measured, there will only be need to measure the time it takes for a number of values to be inserted into a tree using the *insert()* functions of both tree classes.

4. Methodology

The experimental procedure was briefly described and explained above. The specific procedure, with reference to the Java code being run, will be explained in this section.

4.1 Independent variables

The independent variables in this procedure refer to what will be changed in the experiment. I will be changing the **size of the sets of data**. Each set of data will be successive integers from 1 to *N*, where *N* is increased in increments of 100, starting from 100 and ending at 1000 (so there will be a total of 10 sets of data). My decision of incrementing *N* by 100 is to ensure that there aren't too many points when the graphs are plotted, but so that enough data is inserted to illustrate the trees' natures and to plot a suitable graph in which a clear enough relationship can be seen. My decision for the data to be in ascending order is to maximize the amount of time it takes for both trees to balance the data.

4.2 Dependent variable

The only dependent variable being measured in this experiment is the **time it takes for each set of data to be inserted into both trees**. This will be measured using difference of the *System Nanotimes* before and after insertion and will give a time in nanoseconds. This is the most precise measure of time possible by the system being used.

4.3 Controlled variables

Variable	Description	Specifications (if applicable)
Computer and operating system used	I will be running the program on my laptop: a MacBook Pro	**Version**: 10.10.5 **Processor**: 2.6 GHz Intel Core i5 **Memory**: 8GB 1600 MHz DDR3
Integrated Development Environment (IDE) used	I will be running the program using a single IDE	**IDE**: IntelliJ IDEA Ultimate 2017.2.1 **Build**: #IU-172.3544.35 **Java Runtime Environment**: 1.8.0_152-release-915-b6 x86_64 **Java Virtual Machine**: OpenJDK 64-Bit Server VM
Same algorithm used	The algorithm from ***Appendix A*** will be used in this experiment.	
Same functions called	The same functions will be called in the programs for every set being tested.	
Same data type used	The experiment will be only using the **int** (32-bit integer) data type for all sets being tested.	

4.4 Procedure

The procedure for the experiment is as follows:

1. Set up the program to insert all sets of values into both AVL and Red-Black Tree algorithms and time each insertion. Output the time in nanoseconds for each of the trees into a text file (please refer to ***Appendix B*** (page 43) for the program used to test the sets).

2. Run the program to have it output the times of all insertions of the sets.

3. Take averages of the times for each set on each tree.

5. Data processing and graph

5.1 Data collection and processing

Below shows the average times for all sets which have been tested. For raw, un-averaged results, please refer to ***Appendix C*** (page 43).

	Average Time (nanoseconds)	
Set Size	**AVL Tree**	**Red-Black Tree**
100	535878	409542
200	832316	524580
300	1097632	841614
400	1094302	1186661
500	1278388	1761996
600	1368342	1956118
700	1710333	2252886
800	1755713	2517534
900	1830463	2590616
1000	2587892	2629516

Figure 5.1.1: Average insertion times of sets tested for both trees

5.2 Graph of time against set size

Below shows a graph of time against set size for the sets of values inserted into both trees.

Key: Blue = AVL Tree, Red = Red-Black Tree

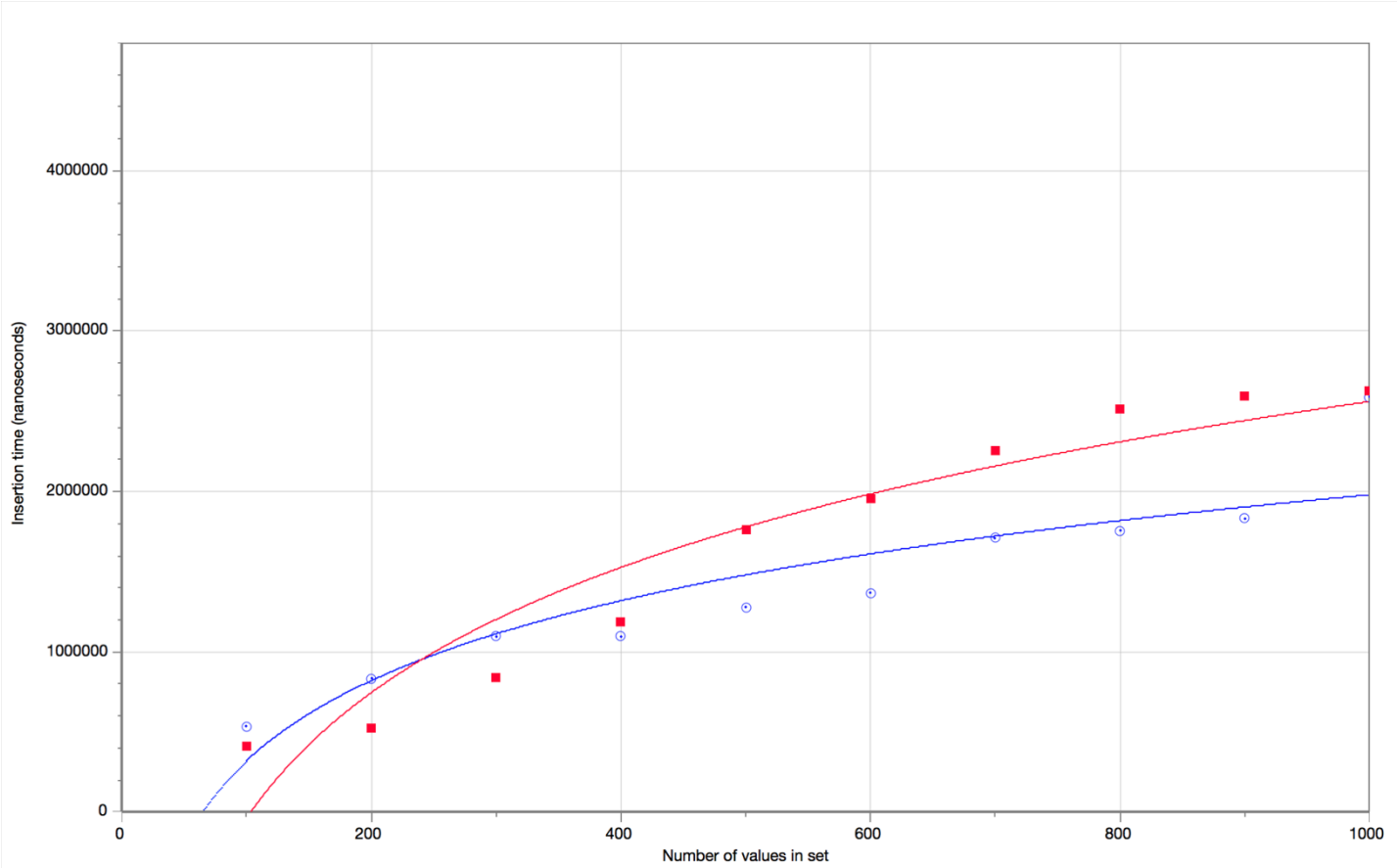

Figure 5.2.1: Graph of insertion time (y) against number of values in set (x) of the AVL and Red-Black trees

6. Results discussion

My hypothesis of the logarithmic relationship has been shown to be correct as seen in the graph. However, my other hypothesis of the red-black tree having a better efficiency in terms of time complexity was shown to not be true for all set values. Referring to the graph in **Figure 5.2.1**, there is a point of intersection between the two graphs at the coordinates: (954000, 240), approximately. This shows that set of size 240 or below will be inserted into the Red-Black tree at a lower time than the AVL tree, but when the set size is over 240, it will be inserted into the AVL tree at a lower time than the Red-Black tree. Upon seeing this, I was astounded and wondered why this was the case.

I initially thought that perhaps that it was due to the problem with the Red-Black tree addressed previously, in which less checks made as values are inserted would mean that the Red-Black tree would gradually tend to become unbalanced. This would then increase insertion time as more nodes were added.

Another possibility which I considered was an issue with the algorithm implementation I used. As more data is inserted into the Red-Black tree, there must be more variables to set multiple times (as a loop is used). Since the Red-Black implementation relied on these variables to handle insertions, it may have just been the time taken for these variables to be set which affected the overall insertion time after a certain number of insertions. Additionally, the AVL implementation made use of recursion (unlike the Red-Black implementation). This could have potentially affected the time taken for the AVL Tree to be lower than the Red-Black tree after a certain number of values are inserted.

I have looked into the maximum heights of an AVL Tree and a Red-Black Tree with N values and have found out that the maximum height from the root to the deepest leaf is approximately $\mathbf{1.44\ log_2(N+2)}$ **for an AVL Tree**[15] and approximately $\mathbf{2\ log_2(N+1)}$ **for a Red-Black Tree**[16]. Plotting these values on a graph with x-axis being N gives:

[15] Alexander, E., n.d. *AVL Trees.* [Online]
Available at: http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html
[Accessed January 2017].
[16] Narahari, Y., n.d. *Height of a Red-Black Tree.* [Online]
Available at: http://lcm.csa.iisc.ernet.in/dsa/node115.html
[Accessed August 2017].

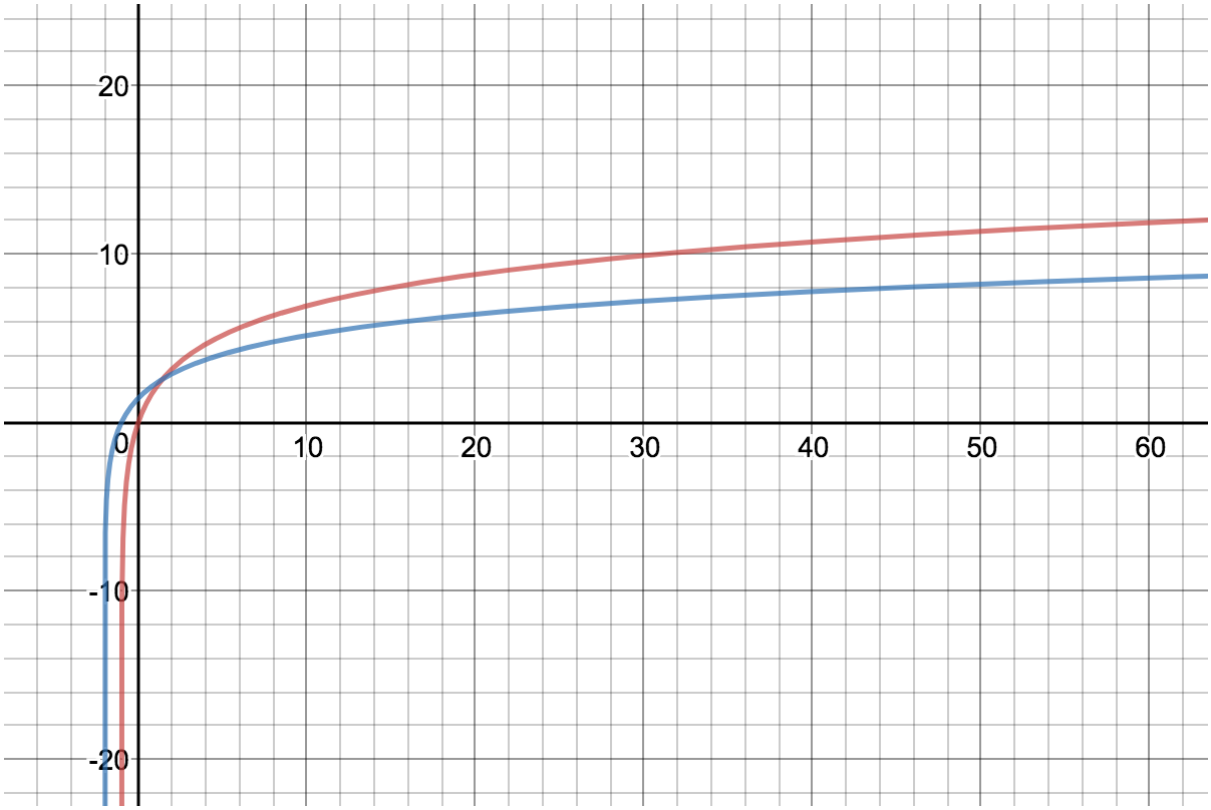

Figure 6.1.1: Graphs of 2 $\log_2(x+1)$ and 1.44 $\log_2(x+2)$

$1.44\log_2(N+2)$ [AVL Tree] = **Blue Graph**
$2\log_2(N+1)$ [Red-Black Tree] = **Red Graph**

As can be seen, the shape of the graphs in **Figure 6.1.1** matches those of the graphs in **Figure 5.2.1**, which were obtained from the experiment.

7. Conclusion

This experiment aimed to use the theory behind AVL and Red-Black trees explained in **section 2** of the essay and practically apply it to see the relationship between insertion time and number of values inserted into the AVL and Red-Black trees. As expected, there is a logarithmic relationship between time and number of values inserted which is apparent in the graph in **Figure 5.2.1**. To take it further, the investigation also aimed to use the theory behind the two trees to see how the time-set size relationship differed for each algorithm.

Ordered sets were used to ensure that every insertion would cause required restructuring. Due to this, the Red-Black Tree would increase in height more than the AVL Tree would since the Red-Black Tree would have a larger height on the right side of the root node. Since the AVL Tree takes better care when balancing itself (rotation for every ordered insertion), however, the AVL Tree does not run into this problem. Hence, I am concluding: **for ordered sets, the Red-Black Tree is more insertion-efficient than the AVL Tree for values < 240. However, the AVL Tree is more insertion-efficient than the Red-Black Tree for values > 240**.

To answer the research question of this essay, my answer would be that the re-balancing algorithm efficiency of both the AVL Tree and the Red-Black tree in terms of time complexity would depend on the number of values inserted as well as how the values are inserted. As seen with the graph of results, the Red-Black Tree is more efficient than the AVL Tree is for a few ordered values. However, with larger ordered values, the AVL Tree proves to be more efficient than the Red-Black tree and as values are increased even further beyond 1000, the AVL Tree, in the long run, proves to be more insertion-efficient than that of the Red-Black Tree.

Bibliography

Adamchik, V. S., 2009. *Algorithmic Complexity.* [Online]
Available at: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html
[Accessed June 2017].

Alexander, E., n.d. *AVL Trees.* [Online]
Available at: http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html
[Accessed 23 August 2017].

Dictionary.com Unabridged, n.d. *Binary.* [Online]
Available at: http://www.dictionary.com/browse/binary
[Accessed May 2017].

Goodrich, M. T., Tamassia, R. & Goldwasser, M. H., 2014. *Data Structures and Algorithms in Java.* Sixth Edition ed. s.l.:Wiley.

Massachusetts Institute of Technology, 2003. *Big O Notation.* [Online]
Available at: http://web.mit.edu/16.070/www/lecture/big_o.pdf
[Accessed June 2017].

Narahari, Y., n.d. *Height of a Red-Black Tree.* [Online]
Available at: http://lcm.csa.iisc.ernet.in/dsa/node115.html
[Accessed August 2017].

Paton, J., n.d. *Red-Black Trees.* [Online]
Available at: http://pages.cs.wisc.edu/~paton/readings/Red-Black-Trees
[Accessed January 2017].

Rowell, E., n.d. *Know Thy Complexities.* [Online]
Available at: http://bigocheatsheet.com/
[Accessed April 2017].

Weiss, M. A., n.d. *AvlNode.java.* [Online]
Available at: https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlNode.java
[Accessed January 2017].

Weiss, M. A., n.d. *AvlTree.java.* [Online]
Available at: https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java
[Accessed January 2017].

Weiss, M. A., n.d. *RedBlackNode.java.* [Online]
Available at:
https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/RedBlackNode.java
[Accessed January 2017].

Weiss, M. A., n.d. *RedBlackTree.java.* [Online]
Available at:
https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/RedBlackTree.java
[Accessed January 2017].

Appendices

Appendix A: Tree/TreeNode Libraries

A1: AvlTree.java (Weiss, n.d.)

```
// BinarySearchTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements an AVL tree.
 * Note that all "matching" is based on the compareTo method.
 *
 * @author Mark Allen Weiss
 */
public class AvlTree {
        /**
         * Construct the tree.
         */
        public AvlTree() {
                root = null;
        }

        /**
         * Insert into the tree; duplicates are ignored.
         *
         * @param x the item to insert.
         */
        public void insert(Comparable x) {
                root = insert(x, root);
        }

        /**
         * Remove from the tree. Nothing is done if x is not found.
         *
         * @param x the item to remove.
         */
        public void remove(Comparable x) {
```

```
        System.out.println("Sorry, remove unimplemented");
}

/**
 * Find the smallest item in the tree.
 *
 * @return smallest item or null if empty.
 */
public Comparable findMin() {
        return elementAt(findMin(root));
}

/**
 * Find the largest item in the tree.
 *
 * @return the largest item of null if empty.
 */
public Comparable findMax() {
        return elementAt(findMax(root));
}

/**
 * Find an item in the tree.
 *
 * @param x the item to search for.
 * @return the matching item or null if not found.
 */
public Comparable find(Comparable x) {
        return elementAt(find(x, root));
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty() {
        root = null;
}

/**
 * Test if the tree is logically empty.
 *
 * @return true if empty, false otherwise.
 */
public boolean isEmpty() {
        return root == null;
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree() {
```

```
        if (isEmpty())
                System.out.println("Empty tree");
        else
                printTree(root);
}

/**
 * Internal method to get element field.
 *
 * @param t the node.
 * @return the element field or null if t is null.
 */
private Comparable elementAt(AvlNode t) {
        return t == null ? null : t.element;
}

/**
 * Internal method to insert into a subtree.
 *
 * @param x the item to insert.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private AvlNode insert(Comparable x, AvlNode t) {
        if (t == null)
                t = new AvlNode(x, null, null);
        else if (x.compareTo(t.element) < 0) {
                t.left = insert(x, t.left);
                if (height(t.left) - height(t.right) == 2)
                        if (x.compareTo(t.left.element) < 0)
                                t = rotateWithLeftChild(t);
                        else
                                t = doubleWithLeftChild(t);
        } else if (x.compareTo(t.element) > 0) {
                t.right = insert(x, t.right);
                if (height(t.right) - height(t.left) == 2)
                        if (x.compareTo(t.right.element) > 0)
                                t = rotateWithRightChild(t);
                        else
                                t = doubleWithRightChild(t);
        } else
                ;  // Duplicate; do nothing
        t.height = max(height(t.left), height(t.right)) + 1;
        return t;
}

/**
 * Internal method to find the smallest item in a subtree.
 *
 * @param t the node that roots the tree.
 * @return node containing the smallest item.
```

```
     */
    private AvlNode findMin(AvlNode t) {
        if (t == null)
            return t;

        while (t.left != null)
            t = t.left;
        return t;
    }

    /**
     * Internal method to find the largest item in a subtree.
     *
     * @param t the node that roots the tree.
     * @return node containing the largest item.
     */
    private AvlNode findMax(AvlNode t) {
        if (t == null)
            return t;

        while (t.right != null)
            t = t.right;
        return t;
    }

    /**
     * Internal method to find an item in a subtree.
     *
     * @param x is item to search for.
     * @param t the node that roots the tree.
     * @return node containing the matched item.
     */
    private AvlNode find(Comparable x, AvlNode t) {
        while (t != null)
            if (x.compareTo(t.element) < 0)
                t = t.left;
            else if (x.compareTo(t.element) > 0)
                t = t.right;
            else
                return t;    // Match

        return null;   // No match
    }

    /**
     * Internal method to print a subtree in sorted order.
     *
     * @param t the node that roots the tree.
     */
    private void printTree(AvlNode t) {
        if (t != null) {
```

```
                printTree(t.left);
                System.out.println(t.element);
                printTree(t.right);
        }
}

/**
 * Return the height of node t, or -1, if null.
 */
private static int height(AvlNode t) {
        return t == null ? -1 : t.height;
}

/**
 * Return maximum of lhs and rhs.
 */
private static int max(int lhs, int rhs) {
        return lhs > rhs ? lhs : rhs;
}

/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then return new root.
 */
private static AvlNode rotateWithLeftChild(AvlNode k2) {
        AvlNode k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        k2.height = max(height(k2.left), height(k2.right)) + 1;
        k1.height = max(height(k1.left), k2.height) + 1;
        return k1;
}

/**
 * Rotate binary tree node with right child.
 * For AVL trees, this is a single rotation for case 4.
 * Update heights, then return new root.
 */
private static AvlNode rotateWithRightChild(AvlNode k1) {
        AvlNode k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        k1.height = max(height(k1.left), height(k1.right)) + 1;
        k2.height = max(height(k2.right), k1.height) + 1;
        return k2;
}

/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
```

```
     * For AVL trees, this is a double rotation for case 2.
     * Update heights, then return new root.
     */
    private static AvlNode doubleWithLeftChild(AvlNode k3) {
        k3.left = rotateWithRightChild(k3.left);
        return rotateWithLeftChild(k3);
    }

    /**
     * Double rotate binary tree node: first right child
     * with its left child; then node k1 with new right child.
     * For AVL trees, this is a double rotation for case 3.
     * Update heights, then return new root.
     */
    private static AvlNode doubleWithRightChild(AvlNode k1) {
        k1.right = rotateWithLeftChild(k1.right);
        return rotateWithRightChild(k1);
    }

    /**
     * The tree root.
     */
    private AvlNode root;
}
```

A2: AvlNode.java (Weiss, n.d.)

```
// Basic node stored in AVL trees
// Note that this class is not accessible outside
// of package DataStructures

class AvlNode {
    // Constructors
    AvlNode(Comparable theElement) {
        this(theElement, null, null);
    }

    AvlNode(Comparable theElement, AvlNode lt, AvlNode rt) {
        element = theElement;
        left = lt;
        right = rt;
        height = 0;
    }

    // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    AvlNode left;         // Left child
    AvlNode right;        // Right child
    int height;       // Height
}
```

A3: RedBlackTree.java (Weiss, n.d.)

```
// RedBlackTree class
//
// CONSTRUCTION: with a negative infinity sentinel
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements a red-black tree.
 * Note that all "matching" is based on the compareTo method.
 *
 * @author Mark Allen Weiss
 */
public class RedBlackTree {

        /**
         * Construct the tree.
         *
         * @param negInf a value less than or equal to all others.
         */
        public RedBlackTree(Comparable negInf) {
                header = new RedBlackNode(negInf);
                header.left = header.right = nullNode;
        }

        /**
         * Insert into the tree. Does nothing if item already present.
         *
         * @param item the item to insert.
         */
        public void insert(Comparable item) {
                current = parent = grand = header;
                nullNode.element  = item;

                while (current.element.compareTo(item) != 0) {
                        great = grand;
                        grand = parent;
                        parent = current;
                        current = item.compareTo(current.element) < 0 ?
                                        current.left : current.right;

                        // Check if two red children; fix if so
```

```
                if (current.left.color == RED && current.right.color == RED)
                        handleReorient(item);
        }

        // Insertion fails if already present
        if (current != nullNode)
                return;
        current = new RedBlackNode(item, nullNode, nullNode);

        // Attach to parent
        if (item.compareTo(parent.element) < 0)
                parent.left = current;
        else
                parent.right = current;
        handleReorient(item);
}

/**
 * Remove from the tree.
 * Not implemented in this version.
 *
 * @param x the item to remove.
 */
public void remove(Comparable x) {
        System.out.println("Remove is not implemented");
}

/**
 * Find the smallest item  the tree.
 *
 * @return the smallest item or null if empty.
 */
public Comparable findMin() {
        if (isEmpty())
                return null;

        RedBlackNode itr = header.right;

        while (itr.left != nullNode)
                itr = itr.left;

        return itr.element;
}

/**
 * Find the largest item in the tree.
 *
 * @return the largest item or null if empty.
 */
public Comparable findMax() {
        if (isEmpty())
```

```
                return null;

        RedBlackNode itr = header.right;

        while (itr.right != nullNode)
                itr = itr.right;

        return itr.element;
}

/**
 * Find an item in the tree.
 *
 * @param x the item to search for.
 * @return the matching item or null if not found.
 */
public Comparable find(Comparable x) {
        nullNode.element = x;
        current = header.right;

        for (; ; ) {
                if (x.compareTo(current.element) < 0)
                        current = current.left;
                else if (x.compareTo(current.element) > 0)
                        current = current.right;
                else if (current != nullNode)
                        return current.element;
                else
                        return null;
        }
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty() {
        header.right = nullNode;
}

/**
 * Test if the tree is logically empty.
 *
 * @return true if empty, false otherwise.
 */
public boolean isEmpty() {
        return header.right == nullNode;
}

/**
 * Print the tree contents in sorted order.
 */
```

```
public void printTree() {
        if (isEmpty())
                System.out.println("Empty tree");
        else
                printTree(header.right);
}

/**
 * Internal method to print a subtree in sorted order.
 *
 * @param t the node that roots the tree.
 */
private void printTree(RedBlackNode t) {
        if (t != nullNode) {
                printTree(t.left);
                System.out.println(t.element);
                printTree(t.right);
        }
}

/**
 * Internal routine that is called during an insertion
 * if a node has two red children. Performs flip and rotations.
 *
 * @param item the item being inserted.
 */
private void handleReorient(Comparable item) {
        // Do the color flip
        current.color = RED;
        current.left.color = BLACK;
        current.right.color = BLACK;

        if (parent.color == RED)   // Have to rotate
        {
                grand.color = RED;
                if ((item.compareTo(grand.element) < 0) !=
                                (item.compareTo(parent.element) < 0))
                        parent = rotate(item, grand);  // Start dbl rotate
                current = rotate(item, great);
                current.color = BLACK;
        }
        header.right.color = BLACK; // Make root black
}

/**
 * Internal routine that performs a single or double rotation.
 * Because the result is attached to the parent, there are four cases.
 * Called by handleReorient.
 *
 * @param item   the item in handleReorient.
 * @param parent the parent of the root of the rotated subtree.
```

```
     * @return the root of the rotated subtree.
     */
    private RedBlackNode rotate(Comparable item, RedBlackNode parent) {
            if (item.compareTo(parent.element) < 0)
                    return parent.left = item.compareTo(parent.left.element) < 0 ?
                                    rotateWithLeftChild(parent.left) :  // LL
                                    rotateWithRightChild(parent.left);  // LR
            else
                    return parent.right = item.compareTo(parent.right.element) < 0 ?
                                    rotateWithLeftChild(parent.right) :  // RL
                                    rotateWithRightChild(parent.right);  // RR
    }

    /**
     * Rotate binary tree node with left child.
     */
    static RedBlackNode rotateWithLeftChild(RedBlackNode k2) {
            RedBlackNode k1 = k2.left;
            k2.left = k1.right;
            k1.right = k2;
            return k1;
    }

    /**
     * Rotate binary tree node with right child.
     */
    static RedBlackNode rotateWithRightChild(RedBlackNode k1) {
            RedBlackNode k2 = k1.right;
            k1.right = k2.left;
            k2.left = k1;
            return k2;
    }

    private RedBlackNode header;
    private static RedBlackNode nullNode;

    static        // Static initializer for nullNode
    {
            nullNode = new RedBlackNode(null);
            nullNode.left = nullNode.right = nullNode;
    }

    static final int BeLACK = 1;    // Black must be 1
    static final int RED = 0;

    // Used in insert routine and its helpers
    private static RedBlackNode current;
    private static RedBlackNode parent;
    private static RedBlackNode grand;
    private static RedBlackNode great;
```

```
}
```

A4: RedBlackNode.java (Weiss, n.d.)

```
// Basic node stored in red-black trees
// Note that this class is not accessible outside
// of package DataStructures

class RedBlackNode {
        // Constructors
        RedBlackNode(Comparable theElement) {
                this(theElement, null, null);
        }

        RedBlackNode(Comparable theElement, RedBlackNode lt, RedBlackNode rt) {
                element = theElement;
                left = lt;
                right = rt;
                color = RedBlackTree.BLACK;
        }

        // Friendly data; accessible by other package routines
        Comparable element;     // The data in the node
        RedBlackNode left;      // Left child
        RedBlackNode right;     // Right child
        int color;      // Color
}
```

Appendix B: Program used in the experiment

```
int set = 100; // Change and re-run program

for (int trial = 1; trial <= 10; trial++) {
        AvlTree avl = new AvlTree();
        RedBlackTree rb = new RedBlackTree(Double.MIN_VALUE);

        long startAVL = System.nanoTime();
        for (double i = 1; i <= set; i++)
                avl.insert(i);
        long endAVL = System.nanoTime();

        long startRB = System.nanoTime();
        for (double i = 1; i <= set; i++)
                rb.insert(i);
        long endRB = System.nanoTime();

        System.out.println("AVL Trial " + trial + ": " + (endAVL - startAVL));
        System.out.println("RB Trial " + trial + ": " + (endRB - startRB));
}
```

Appendix C: Raw data of times obtained

C1: Raw and average times for AVL Tree

Set Size	100	200	300	400	500	600	700	800	900	1000
Trial 1	2581280	5983053	6669072	4445067	7613113	5983508	6363343	11501305	6973136	11601223
Trial 2	488279	397182	1935790	2275095	504328	851024	1854499	1057391	910888	1867284
Trial 3	410746	232996	631174	772049	522816	724961	646244	1945648	1170284	1054020
Trial 4	272462	189196	275478	696903	1046400	434082	2021447	842967	1338265	1087820
Trial 5	985350	133719	229312	464093	441713	438329	1745406	259145	988840	2134978
Trial 6	135056	161752	199271	1048937	1008765	550976	1500212	152406	921561	6765564
Trial 7	203909	352340	238741	274836	410040	1159039	736276	1085024	3434569	390173
Trial 8	68037	167254	190581	308889	410341	499898	507880	362423	481209	554644
Trial 9	109732	472998	374234	329820	410973	521406	1560302	150863	1771136	195349
Trial 10	103929	232673	232664	327328	415393	2520195	167725	199961	314738	227860
Average	**535878**	**832316**	**1097632**	**1094302**	**1278388**	**1368342**	**1710333**	**1755713**	**1830463**	**2587892**

C2: Raw and average times for Red-Black Tree

Set Size	100	200	300	400	500	600	700	800	900	1000
Trial 1	897100	1572146	4719353	3807996	4511711	4618547	3827439	5471420	7612243	4993088
Trial 2	398344	2047101	445611	4682227	415191	444595	565356	4396306	957851	1071088
Trial 3	1603153	419289	471936	779954	1291588	502841	458196	9157614	8801192	2338015
Trial 4	771489	161819	236261	365249	383348	513776	395184	631247	5122684	6833644
Trial 5	50030	136920	1074495	325017	669508	4761305	9042658	749707	851016	3701804
Trial 6	67463	196400	272305	352181	447824	6479630	2272656	1919854	1329396	949009
Trial 7	100926	175193	273745	405009	7965308	831211	1359304	967175	331589	950182
Trial 8	69530	163721	236786	361201	1165498	462080	730359	609491	300666	3204756
Trial 9	68191	170886	436382	351122	422865	429500	3276667	557460	290566	1265100
Trial 10	69194	202322	249268	436653	347118	517693	601042	715065	308959	988471
Average	**409542**	**524580**	**841614**	**1186661**	**1761996**	**1956118**	**2252886**	**2517534**	**2590616**	**2629516**

Appendix D: Permission letter from Dr. Mark Allen Weiss

D1: Permission email sent to Dr. Weiss

Dear Dr. Weiss,

My name is [redacted]. I am a senior student of an international school in Southeast Asia, [redacted], and I am doing the IB Diploma Program. As part of the diploma program, I must write a 4000-word research paper on a topic of my choice and I have decided to do my topic on comparing two binary search trees: AVL and Red-Black. After looking into various resources online, I have found your source code for the AVL Tree and RedBlack Tree algorithms in Java:

AvlTree.java: http://users.cis.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java
AvlNode.java: http://users.cis.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlNode.java
RedBlackTree.java: http://users.cis.fiu.edu/~weiss/dsaajava/code/DataStructures/RedBlackTree.java
RedBlackNode.java: http://users.cis.fiu.edu/~weiss/dsaajava/code/DataStructures/RedBlackNode.java

I was wondering if I would be allowed to use these resources for the experimental procedure (which involves timing the insertions of sets of data into both trees) as well as discussing the algorithms in my paper, given that I cite them. Note that this essay will not be sold or published online. The only people who will have access to it are myself, my supervisor and the examiner who will mark it in the summer of 2018. My supervisor has also looked into ordering your book "Data Structures and Algorithm Analysis in Java", as it would provide as a good resource for future students at the school who would also like to write computer science research papers.

Please let me know if this is okay with you and thank you very much for taking time to read this.

Yours sincerely,
[redacted]

D2: Reply email from Dr. Weiss

Dear [redacted]

That is fine... best of luck with your paper.

Regards,

Mark Weiss