Measuring the efficiency of Binary Search Trees

Research Question:
How does the re-balancing algorithm efficiency of a Splay Tree compare to that of an Adelson- Velskii and Landis Tree in terms of time complexity upon insertion of values?

Computer Science Extended Essay
Word Count: 3990

Table of Content

1. Introduction

1.1 Motivation

As an enthusiast of Computer Science, I've always found the nature of "Data" fascinating! Today, in most things we use in our routines, data is all around us in very different forms. In this research, I'll be exploring the complexity behind data, by looking at the structure of binary search trees. I hope to reach a clear conclusion on the comparison between two different types of binary search trees while raising my understanding of them throughout the research.

1.2 Background

This research will specifically look into two different binary search trees; the Splay Tree and the Adelson-Velskii and Landis Tree (AVL), and will compare their time complexity for the insertion of a given set of values. The term "time complexity" describes how long an algorithm takes to execute given a collection of input values of a specific size. Consequently, this research will explore "***How the re-balancing algorithm efficiency of a Splay Tree compares to that of an Adelson-Velskii and Landis Tree in terms of time complexity upon insertion of values***".

2. Theory

2.1 Binary Search Trees

The basis of the two trees under investigation is a binary search tree, a data structure with well-defined behavior. Binary means "being consisting of two things", as in each item (node) in a tree points to a maximum of two other nodes, which are referred to, as children. However, a node can also be pointing at one, or no other child at all.

When a value is inserted, a binary search tree must determine where to place it. This placement is done, regarding the conditions each node has. Each node in the tree has a left child pointer and a right

child pointer that both points to different nodes. A node's right child must have a value greater than the node, and therefore the left child must have a value smaller than the node. The first value, placed at the top of the tree, is called the root node, also referred to as a parent or grandparent of a child node.

Inserting a new node can be done by having a simple “if - else” approach:

A. ***If*** the tree has no nodes, the node will be placed as the root node.

B. ***Else, If*** the tree has a root node, and its value is greater than the inserted node’s value, this process will occur for the root node’s left child position as well.

C. ***Else, If*** the tree has a root node, and its value is smaller than the inserted node’s value, this process will occur for the root node’s right child position as well.

D. **Else, If** the root node doesn't have a left or right child, the node will be placed there.

E. ***Else***, the property of the inserted node does not match the other existing ones.

I wrote the full code for insertion (in C), using the approach of the above algorithm for further understanding **(Appendix E)**. **Figures 2.1.A** and **2.1.B** show a snippet of that code.

```
10  struct node *insert(struct node *root, int val)
11  {
12      if(root == NULL)
13          return getNewNode(val);
14
15      if(root->key < val)
16          root->right = insert(root->right,val);
17
18      else if(root->key > val)
19          root->left = insert(root->left,val);
20
21      return root;
22  }
```

Figure 2.1.A: Insertion “if-else” statement in a Binary Tree.

```
10  struct node *getNewNode(int val)
11  {
12      struct node *newNode = malloc(sizeof(struct node));
13      newNode->key   = val;
14      newNode->left  = NULL;
15      newNode->right = NULL;
16
17      return newNode;
18  }
```

Figure 2.1.B: function for insertion in Binary Tree

The "key" variable in **Figure 2.1.A** indicates the variable type's value stored in the root. In binary search trees, a key (value) is placed in a node (container). In a real-life insertion, the insert method will contain both the node's (or root's) id and the key inserted. As shown on **lines 16 & 19**'s insert function.

Note: the "NULL" keyword used in the programs above, refers to an empty node (without any value inside), or non-existent.

In **Figure 2.1.B**, the getNewNode( ) function is shown. The function is only executed (as seen in **Figure 2.1.A**) when the node being inserted is the first one in the tree (root == NULL). Notice how after the value being inserted was set as the "key" in **line 13**, the algorithm automatically produces 2 empty children for the root in **lines 14-15**. This prevents from this process happening in the next insertion of values, and therefore improves time efficiency.

Another important matter about BSTs is the variety of data able to be inserted into them. A BST can contain Strings or Doubles, etc. with the same algorithm that it holds integers with. However, these data cannot be inserted into a tree together, as the algorithm will not be able to find a relationship between them. Therefore option "E" stated above exists, for making sure the property of the data being inserted match one another.

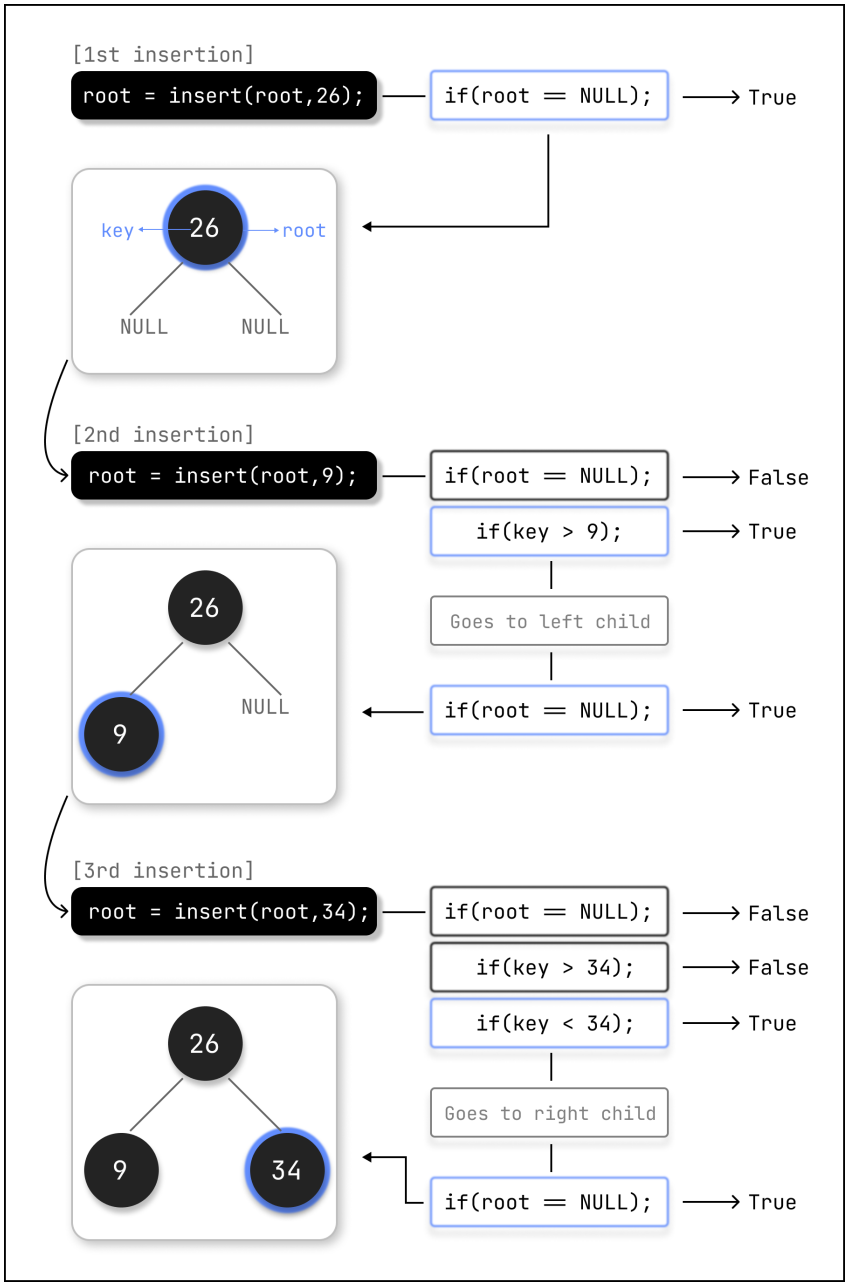

Figure 2.1.D: Simple Visualization of the "Insertion of nodes in a Binary Tree"

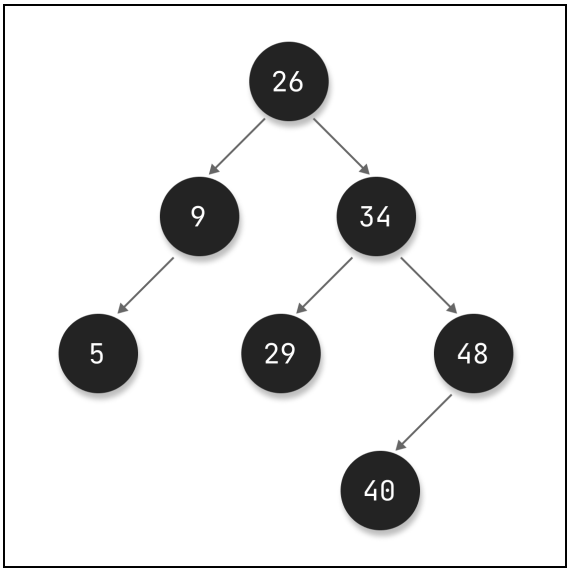

Figure 2.1.C: Visual example of Binary Data Tree

The search function is the main purpose of a Binary Search Tree, as indicated by the "search" in it. The searching process has the same approach as insertion.

Because of the way nodes are organized in a BST structure, searching for values is much more efficient than other methods, like a linear search. There are various notations available to measure this

efficiency, one of which is the "**Big-O Notation**", used to measure the worst-case efficiency of an algorithm. The searching efficiency for an array is 0(N), and 0(log2 N) for a BST, where N indicates the size of the structure. The difference in efficiency, therefore, is recognizable and massive, as shown in **Figure 2.1.E**, where the y-axis shows the two data structures, and the x-axis shows the number of values stored.

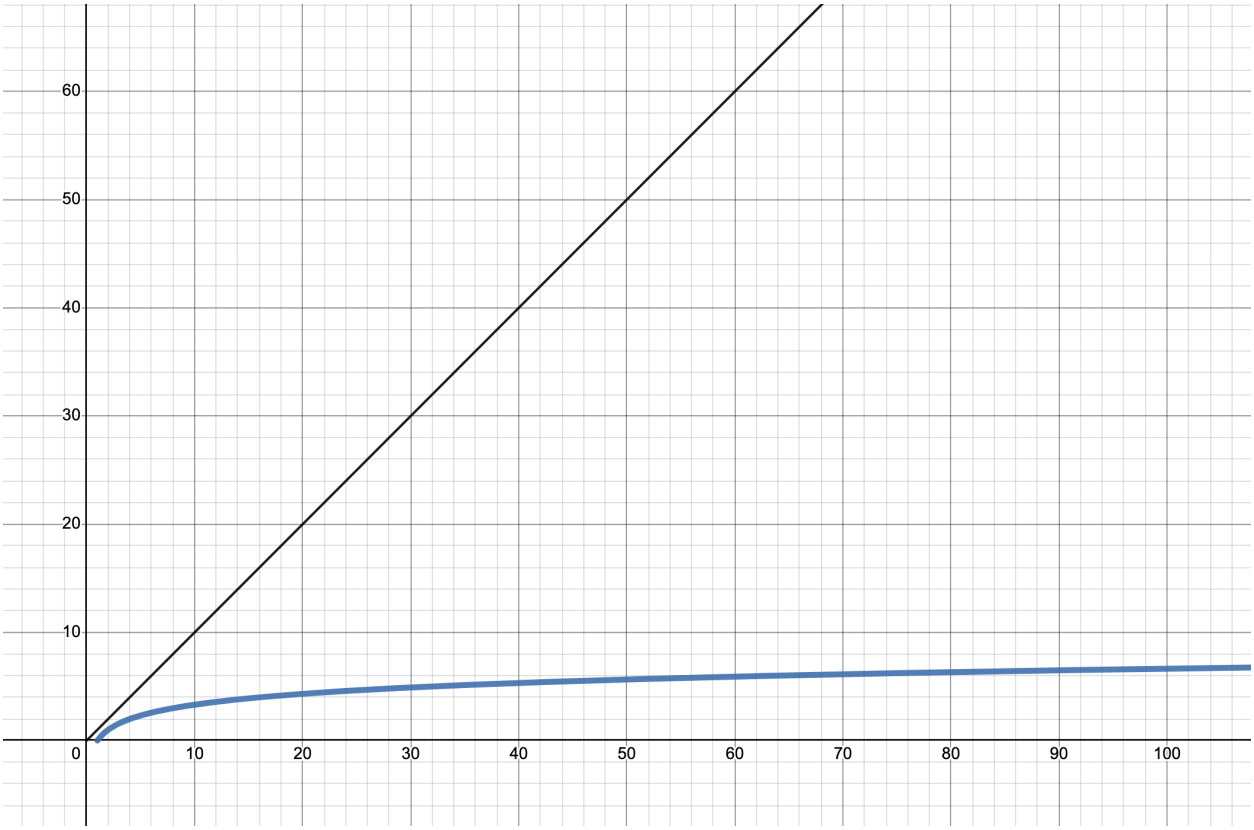

Figure 2.1.E: Graph showing the worst-case number of searches for BST and an Array

BLACK — ARRAY
BLUE — BINARY SEARCH TREE

Although this is a great feature of the BST, this structure has an exception. When a sequence of nodes, like 1,2,3,4, gets inserted into the tree, its search efficiency is calculated like an Array, as the tree's structure is unbalanced, as shown in **Figure 2.1.F**.

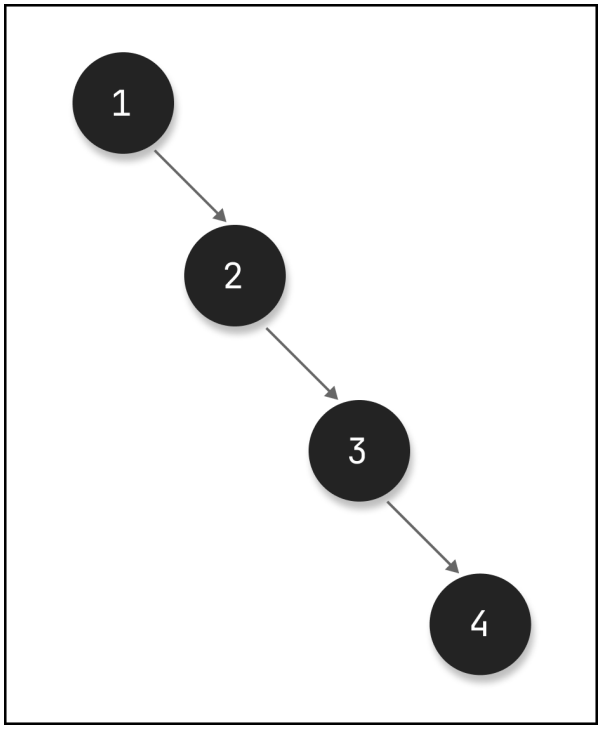

Figure 2.1.F: Example of an unbalanced BST

The structure will function similarly to a linear search in such exceptional insertions. This case scenario's search efficiency will therefore be, 0(4). Therefore this structure is not considered a BST.

Trees will incorporate balancing algorithms to address the issue and maintain the structure they require (reducing the number of searches needed to find a value). Different methods for implementing the same structure are possible, each of which has the same behavior but uses a different technique to ensure it, which means some implementations are better than others in various ways. Both Splay Trees and AVL Trees have distinct definitions of how they balance themselves or increase their efficiency when the balancing algorithm is taken into account, which we will be exploring in detail.

2.2 Splay Trees

Unlike most BSTs, the Splay Tree does not have any specific conditions to balance itself, as no operation can be found efficient. A Splay Tree doesn't always need to be balanced, however, a method known as Splaying is implemented to make the tree more balanced every time upon insertion, deletion, etc. It is therefore important to acknowledge that in this case, maintaining balance is only done to have better time efficiency.

In question to the tree's root, splaying moves the node. In our case, upon insertion, the splay tree takes the BST's approach, and moves the newly inserted node, to its appropriate root. The splay tree uses various rotations to move elements upward in the tree, while they also shorten the path to any nodes along the path to the splayed node. This latter effect means that splaying operations don't necessarily make the tree more balanced, however, it tends to move the newly inserted nodes upward, and therefore make them more accessible, and the tree more efficient.

2.2.1 Zig & Zag

When the node being splayed, is the child of the root, either node is a left child of the root, in which case every node moves one place to the right, or the node is a right child of its parent in which case every node moves one place to the left.

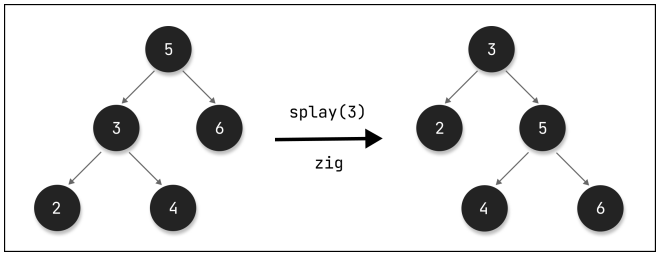

Figure 2.2.A: Example of a zig rotation

2.2.2 Zig-Zig & Zag-Zag

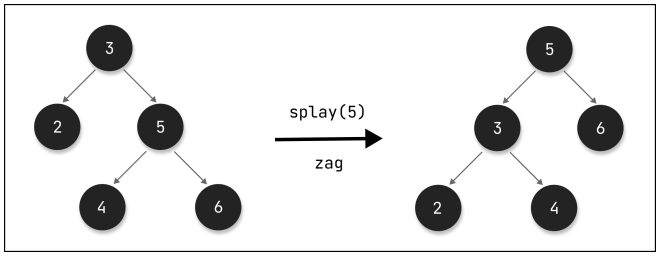

Figure 2.2.B: Example of a zag rotation

In this rotation, lower down in the tree rotations are performed in pairs so that nodes on the path from the splayed node to the root move closer to the root on average. Same as the previous rotation, while the node is the left child, every node moves two steps right, and vice versa.

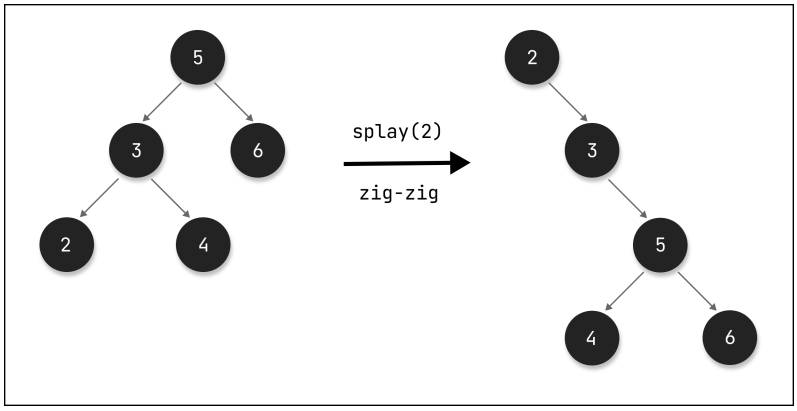

Figure 2.2.C: Example of a zig-zig rotation (Double Zig)

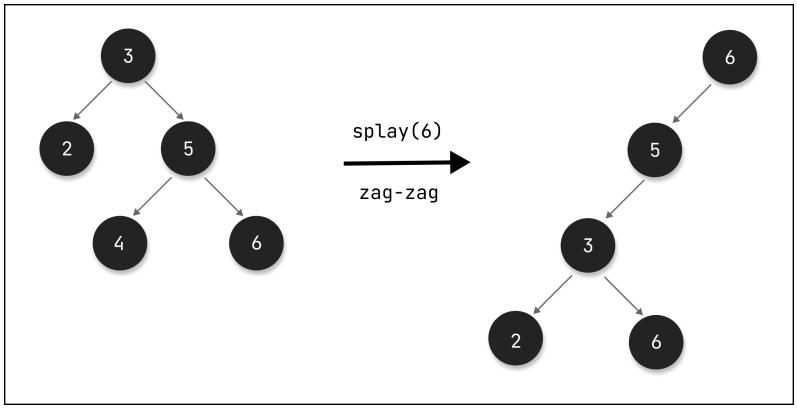

Figure 2.2.D: Example of a zag-zag rotation (Double Zag)

2.2.3 Zig-Zag & Zag-Zig

In this case, the splayed node is the left child of a right child or vice versa, therefore both right and left movements happen together. The rotations produce a subtree whose height is less than that of the original tree. **Thus, this rotation improves the balance of the tree**.

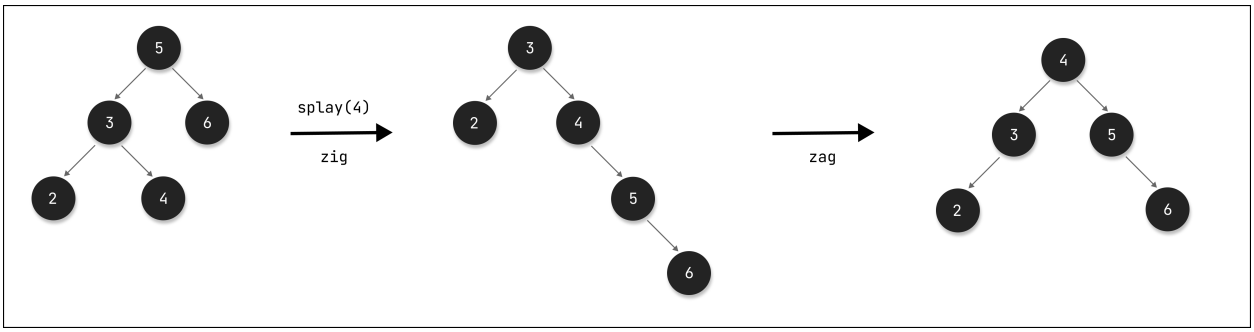

Figure 2.2.E: Example of a zig-zag rotation

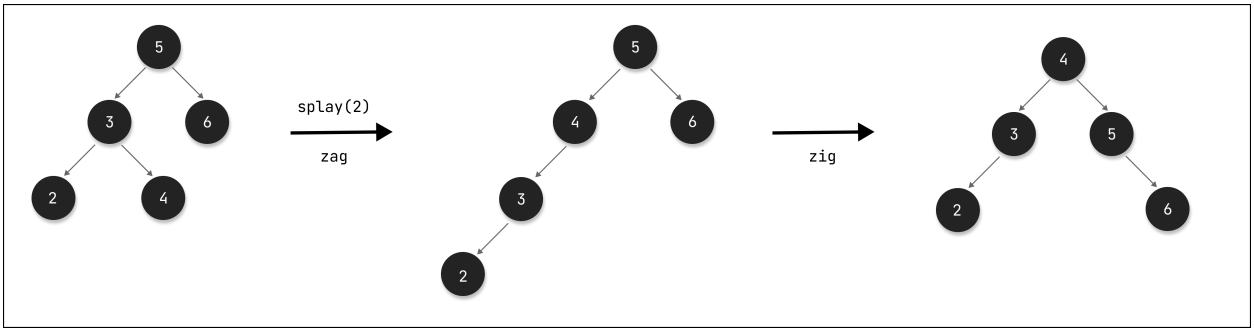

Figure 2.2.F: Example of a zag-zig rotation

Note: In double rotations such as Zig-Zig or Zag-Zig, etc. the rotations will execute separately (the second rotation will only occur after all nodes have completed their first rotation).

Although only the zig-zag & zag-zig rotations balance the tree, as it results in its height to reduce, the splaying method holds the most recently accessed nodes at the top of the tree, which is more influential in time efficiency.

I'll be investigating the insertion time complexity of the Splay Tree, using Dr. Weiss's open-source Java code **(Appendix C)**. The two rotation algorithms can be found in **Figures 2.2.G** and **2.2.H**.

```
241     private static <AnyType> BinaryNode<AnyType> rotateWithLeftChild( BinaryNode<AnyType> k2 )
242     {
243         BinaryNode<AnyType> k1 = k2.left;
244         k2.left = k1.right;
245         k1.right = k2;
246         return k1;
247     }
```

Figure 2.2.H: Algorithm used for Left child rotation

```
241     private static <AnyType> BinaryNode<AnyType> rotateWithRightChild( BinaryNode<AnyType> k1 )
242     {
243         BinaryNode<AnyType> k2 = k1.right;
244         k1.right = k2.left;
245         k2.left = k1;
246         return k2;
247     }
```

Figure 2.2.G: Algorithm used for right child rotation

It can be seen that while rotating with each child, the opposite direction will be rotated. This is done by assigning a direction to each variable in **line 243**, applying the rotation in **line 244**, and finally adjusting variables by swapping them in **line 245**.

As we explore the time complexities upon insertion, let's look at the insertion function provided in the full code **(Appendix C)**. I will be using the same function, for testing the insertion's time and comparison of the two trees.

```
42      public void insert( AnyType x )
43      {
44          if( newNode == null )
45              newNode = new BinaryNode<AnyType>( null );
46          newNode.element = x;
47
48          if( root == nullNode )
49          {
50              newNode.left = newNode.right = nullNode;
51              root = newNode;
52          }
53          else
54          {
55              root = splay( x, root );
56
57              int compareResult = x.compareTo( root.element );
58
59              if( compareResult < 0 )
60              {
61                  newNode.left = root.left;
62                  newNode.right = root;
63                  root.left = nullNode;
64                  root = newNode;
65              }
66              else
67              if( compareResult > 0 )
68              {
69                  newNode.right = root.right;
70                  newNode.left = root;
71                  root.right = nullNode;
72                  root = newNode;
73              }
74              else
75                  return;   // No duplicates
76          }
77          newNode = null;   // So next insert will call new
78      }
```

Figure 2.2.I: Splay Tree's *insert*( ) function

In **Figure 2.2.I**, the x variable indicates the new node being inserted. The program first checks if any nodes are being inserted, and once it makes sure there are none, it will insert a new one on **lines 44-46**. The program then checks if the tree has a root, the same as the BST's approach in **lines 48-52**. If there is a root, the algorithm will start the splaying method. It compares *x* with the existing parent and splays accordingly as shown above in the left and right child rotation functions in **lines 57-73**. Notice how

in **line 77** there is a newNode structured after the previous one was inserted, so it can be used in the next insertion and improve time efficiency.

Figure 2.2.J visualizes the insertion algorithm provided in the function in **Figure 2.2.I**:

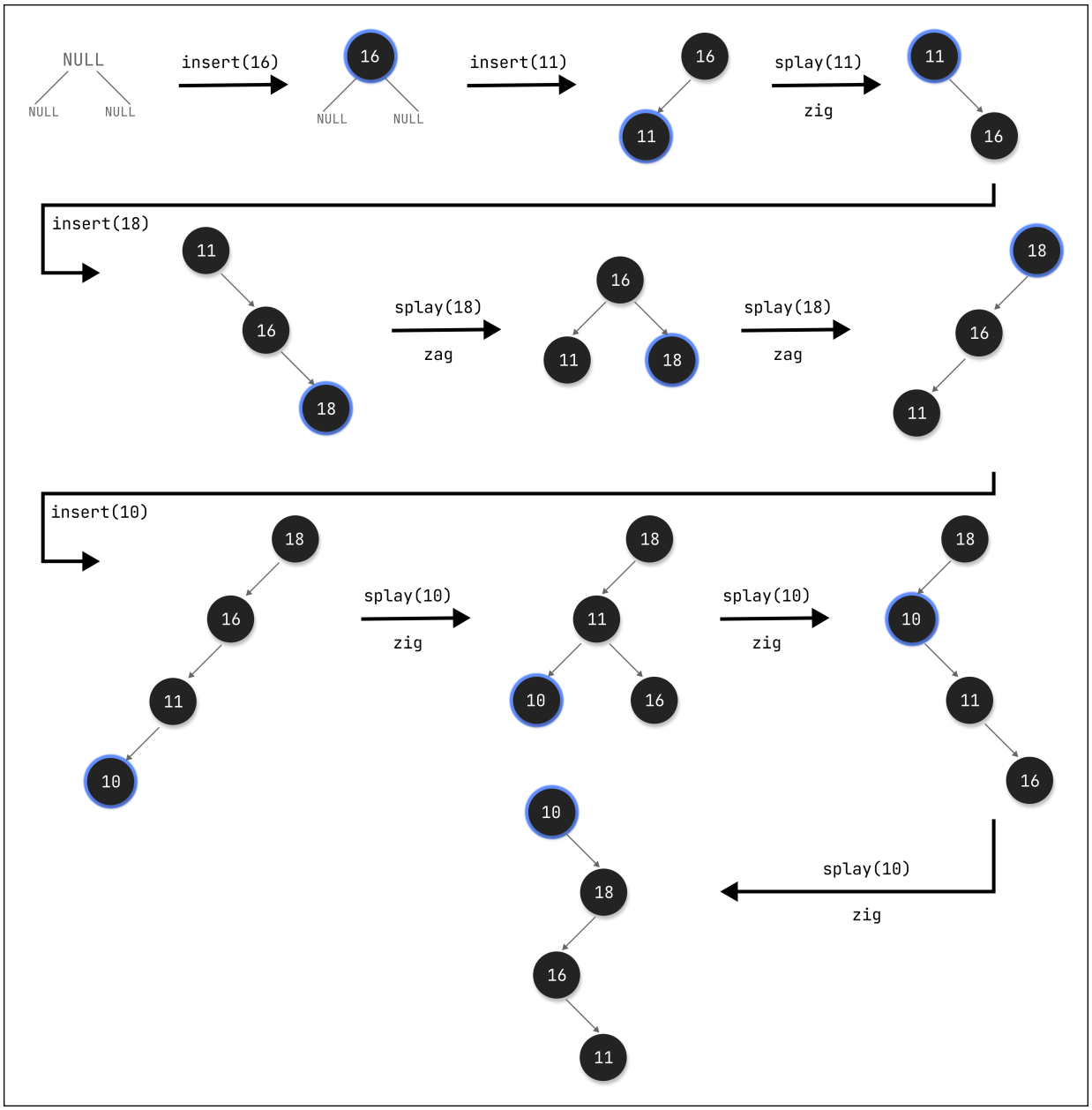

Figure 2.2.J: Splay Tree's Insert( ) function visual

2.3 Adelson-Velskii and Landis (AVL) Tree

A **height-balance** feature is used by the AVL trees which specifies that each node's children can only have a maximum height difference of one between them. Therefore if there is a height discrepancy of more than 1, the tree is said to be unbalanced. In the height-balance feature, “height difference” indicates the space between the left and right nodes, whereas “height” refers to the number of nodes in between a node, a node without children (leaf).

As each side of a node has a height:

$$HeightDifference = height_{left} - height_{right}$$

&

$$HeightDifference = height_{right} - height_{left}$$

Some BSTs return a negative value HeightDifference, which is not acceptable for the AVL tree. An AVL Tree needs all nodes to have a height difference of **either 0 or 1** to be balanced.

Unlike the splay tree, the AVL tree undertakes a specific algorithm to rebalance after every access. The process begins with a standard BST insertion (or any access), and if a node is found unbalanced, a set of rotations will take place, same as splaying, this time to ensure balance in the tree.

When a tree is unbalanced on the right, it will be balanced with a left rotation (and vice-versa). And while there is both, a double rotation will take place:

2.3.1 Single rotation

In single rotations, each node moves one space from its current position, dependent on whether it is a Left rotation (LL) or right (RR).

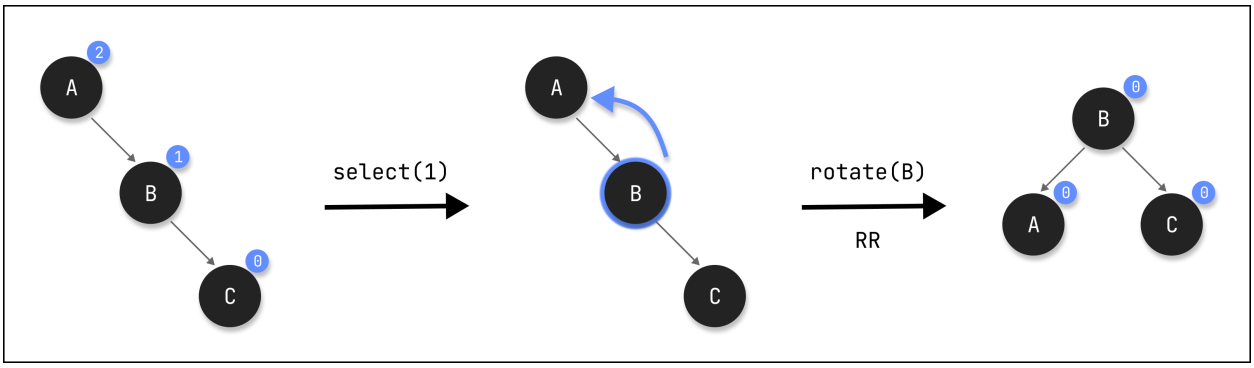

Figure 2.3.A: Visual of a RR rotation

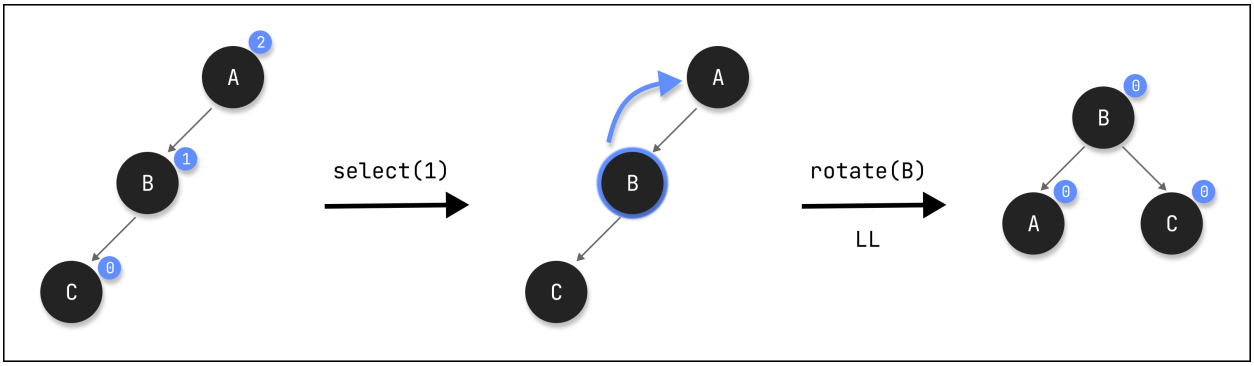

Figure 2.3.B: Visual of a LL rotation

2.3.2 Double rotation

In double rotations, each node moves two spaces from its current position in different directions, dependent on whether it is a left-right rotation (LR) or right-left (RL). First RR/LL rotation is performed on the subtree and then LL/RR rotation is performed on the full tree, by the full tree we mean the first node from the path of the inserted node whose balance factor is other than -1, 0, or 1.

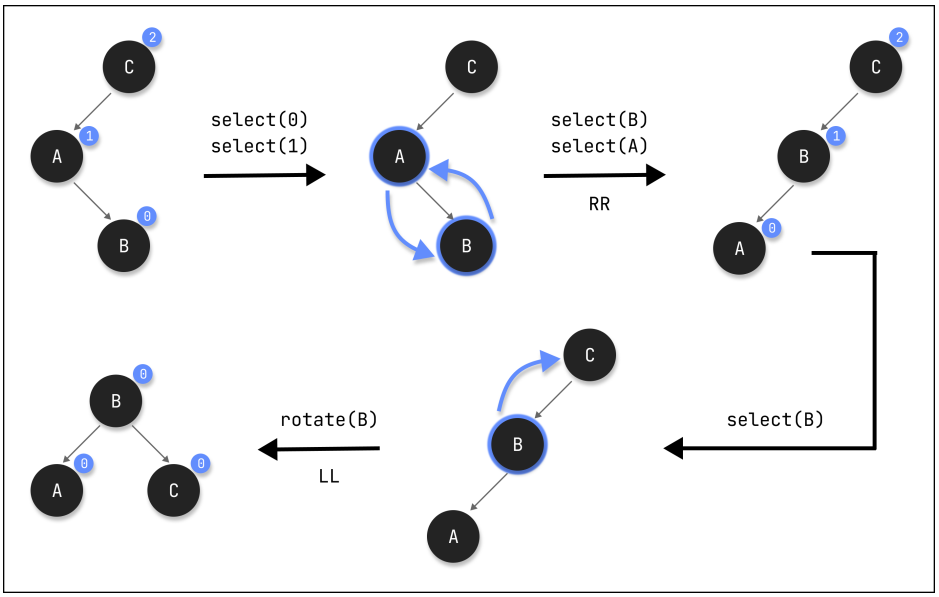

Figure 2.3.C: Visual of a LR rotation

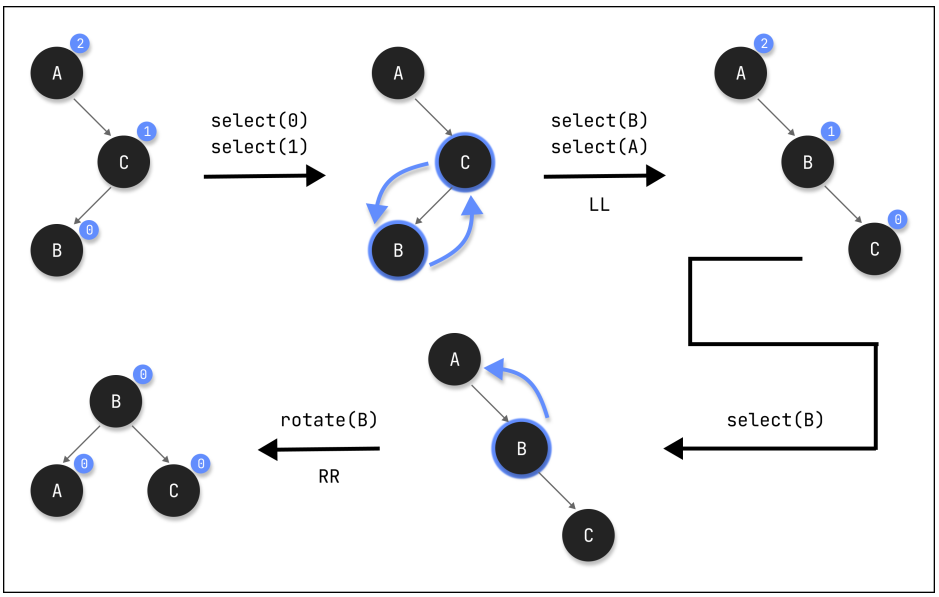

Figure 2.3.D: Visual of a RL rotation

Furthermore, let's look at the AVL Tree's insert function (by Dr.Weiss) in detail **(Appendix D)**.

```
121         private AvlNode insert( Comparable x, AvlNode t )
122         {
123             if( t == null )
124                 t = new AvlNode( x, null, null );
125             else if( x.compareTo( t.element ) < 0 )
126             {
127                 t.left = insert( x, t.left );
128                 if( height( t.left ) - height( t.right ) == 2 )
129                     if( x.compareTo( t.left.element ) < 0 )
130                         t = rotateWithLeftChild( t );
131                     else
132                         t = doubleWithLeftChild( t );
133             }
134             else if( x.compareTo( t.element ) > 0 )
135             {
136                 t.right = insert( x, t.right );
137                 if( height( t.right ) - height( t.left ) == 2 )
138                     if( x.compareTo( t.right.element ) > 0 )
139                         t = rotateWithRightChild( t );
140                     else
141                         t = doubleWithRightChild( t );
142             }
143             else
144                 ;  // Duplicate; do nothing
145             t.height = max( height( t.left ), height( t.right ) ) + 1;
146             return t;
147         }
```

Figure 2.3.E: AVL Tree's *insert*( ) function

The function above, the same as the Splay Tree, take a recursive approach for value insertion; where *x indicates* the value being inserted, and *t*, the current node, beginning with the root.

The function balances the tree after an insertion takes place, according to the nodes' height property. Therefore as explained above, **when *t*'s height difference equals 2, the tree needs balancing**.

Therefore when conditions on **lines 128 and 137** are true, the restructuring process will start after the value is inserted on **lines 127 and 136**, depending on the conditions on **lines 129 and 138**.

When a node is being inserted in one of t's children, and $HeightDifference_t = 2$, we will use the AVL rotations as discussed above. I will explain the algorithm with an (if-else) approach:

A. ***If*** the node is being inserted into t_{left} and $x < value_{left}$, a LL single rotation will occur:

```
233         private static AvlNode rotateWithLeftChild( AvlNode k2 )
234         {
235             AvlNode k1 = k2.left;
236             k2.left = k1.right;
237             k1.right = k2;
238             k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
239             k1.height = max( height( k1.left ), k2.height ) + 1;
240             return k1;
241         }
```

Figure 2.3.F: AVL Tree's *rotatewithleftchild*( ) function

B. ***Else, if*** the node is being inserted into t_{left} and $x > value_{left}$, an RL double rotation will occur:

```
264         private static AvlNode doubleWithLeftChild( AvlNode k3 )
265         {
266             k3.left = rotateWithRightChild( k3.left );
267             return rotateWithLeftChild( k3 );
268         }
```

Figure 2.3.G: AVL Tree's *doublewithleftchild*( ) function

C. ***Else, if*** the node is being inserted into t_{right} and $x < value_{right}$, a RR single rotation will occur

```
249         private static AvlNode rotateWithRightChild( AvlNode k1 )
250         {
251             AvlNode k2 = k1.right;
252             k1.right = k2.left;
253             k2.left = k1;
254             k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
255             k2.height = max( height( k2.right ), k1.height ) + 1;
256             return k2;
257         }
```

Figure 2.3.H: AVL Tree's *rotatewithrightchild*( ) function

D. ***Else, if*** the node is being inserted into t_{right} and $x > value_{right}$, an LR double rotation will occur:

```
277         private static AvlNode doubleWithRightChild( AvlNode k1 )
278         {
279             k1.right = rotateWithLeftChild( k1.right );
280             return rotateWithRightChild( k1 );
281         }
```

Figure 2.3.I: AVL Tree's *doublewithrightchild*( ) function

3. Hypothesis

3.1 Approach

After exploring the two trees in depth, it's time to evaluate their efficiency, by investigating their time complexity upon node insertion. To do this, an experiment will be conducted to measure the time it takes for each tree to add a specific number of value sets. Although this experiment seems to be theoretically incorrect as both Splay and AVL trees have the same efficiency of $0(log_2 N)$ upon insertion; however, that is only for the physical insertion, and not the re-balancing. Our approach will in fact take the re-balancing done after insertion into account, and the time complexity will be measured.

3.2 Expectations

Firstly, between the two trees, the AVL Tree is the one that prioritizes balance in the tree. As discussed the AVL tree **always** balances itself after each insertion if $HeightDifference = 2$. On the other hand, the Splay Tree doesn't ensure balance in the tree and considers efficiency to be the accessibility of newly accessed (inserted) nodes. As explained, the Splay Tree takes an approach to do so with "Splaying", in which only the "Zig-Zag" and "Zag-Zig" rotations will result in a balanced tree. Consequently, the Splay Tree might be faster as it does less restructuring than the AVL Tree.

To determine the relationship between the trees upon node insertion, we have to measure the size of sets being **inserted** (x) and the **time** (y) and evaluate their relation. My hypothesis is that the Splay Tree will perform faster than the AVL Tree. I have this expectation, as the AVL Tree has to re-balance

itself after each insertion, while the Splay Tree is re-balanced upon access, as only the node inserted will be rotated towards the top of the tree. I also have the hypothesis that the gap between the trees' efficiency will grow, as the set sizes get larger, as the AVL Tree will have to do more restructuring.

4. Experimentation

For the experimentation, only the time it takes to insert a number of values into a tree using the $insert()$ functions of both tree classes will need to be assessed since the efficiency of both insertion procedures as a whole is being evaluated.

4.1 Fixed Variables

The fixed variable or the variable being measured for time complexity is the ***time*** **taken for a specific number of data sets to be inserted into a tree, and for the re-balancing algorithm being executed**. To achieve maximum precision the time measurement will be in nanoseconds; and to prevent errors, the IDE's time will not be considered, instead, the time between the value input and output will be measured with the system's time (in nanoseconds).

4.2 Independent Variables

The independent variable or the variable I will be changing throughout the experiment is the **data sets'** ***sizes*** **being inserted in the tree**. To have maximum precision, we are gonna use an ascending sequence of data set sizes, so the time it takes for both trees to balance the data is optimized. The number of sets being taken in this experiment is important, as they should be enough to make a suitable tree, can plot as a graph, and be accurate enough to analyze for our comparison. However, too many sets of data will overcrowd the graph and reduce readability. Therefore in this experiment, I will be taking 10 different sets of data as my inputs.

However, there are two challenges we face:

1. The data inserted in the AVL Tree may just happen to automatically be in an order which makes the AVL Tree balanced, in which situation, the AVL Tree will not execute its re-balancing algorithm, and the experiment will lack precision. To fix this problem, sets will be inserted in the tree in **linear order** to ensure re-balancing in the tree.

2. If sets get inserted in order, they will cause the Splay Tree to take a simple linear form, which will not cause the Splay Tree to Splay, and no rotations will be executed. To ensure that splaying takes place, sets will be inserted in the Splay Tree **randomly**. However, we are gonna ensure that repetition does not take place.

The sets will be in the form of 1 to N. N being an arithmetic sequence, with the first term (a) being 100 and a common difference (d) of 100.

$$\sum_{i=1}^{10} 100n:$$

{1 - 100}, {1 - 200}, {1 - 300},{1 - 400}, ... , {1 - 1000}

A code was written **(Appendix A)**, with attention to all the above factors, to test the program, and give us the results in nanoseconds.

4.3 Constant Variables

It is important to note that for this experiment various other variables are considered to affect the time complexity. Although I cannot change those variables, they can still provide us with reasonable results, as they are a base standard for most people around the world, which don't have special abilities or disabilities.

Constant Variables	Description	Specifications
Machine (Computer)	MacBook Pro — Apple	Processor: 2.6 GHz 6-core Intel Core i7 Memory: 16 GB 2667 MHz DDR4 Operating System: MacOS Monterey — Version 12.2.1
Integrated Development Environment (IDE)	IntelliJ IDEA — JetBrains	Version: 2022.2.1 (Community Edition) Build: #IC-222.3739.54 Runtime version: 17.0.3+7-b469.37 x86_64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.
Data Types	32-bit Integer (int)	Minimum Value: -2^{31} Maximum Value: $2^{31} - 1$
Programming Language	Java 8	Version: Java 8 — Update 321 (build 1.8.0_321-b07) OpenJDK version: 16.0.2 (2021-07-20) OpenJDK Runtime Environment: build 16.0.2+7-67 OpenJDK 64-Bit Server VM: build 16.0.2+7-67, mixed mode, sharing

Figure 4.3.A: Constant Variables

4.4 Testing Procedure

My procedure for this experiment is:

The 20 chosen sets will be inserted into both AVL and Splay Trees. However, insertion will happen in a random order to avoid the Splay Tree from taking a linear form. Note that repetition of values will not take place. **(Appendix A)**

I. Calculating the output time in nanoseconds, by calculating the difference between input time and output time. **(Appendix A)**

II. Taking the average of the raw data, and using them to plot a graph. **(Appendix B)**

5. Data Representation

5.1 Numeric Presentation

Figure 5.1.A shows the average time taken for each set to be executed 10 times. This data is based on the raw data collected **(Appendix B)**.

Tree Type	Splay Tree		AVL Tree	
Unit	NanoSec	Sec	NanoSec	Sec
N = 100	154510	0.0001545105	485692	0.0004856927
N = 200	352022	0.0003520218	615789	0.000615789
N = 300	709732	0.0007097328	836625	0.0008366249
N = 400	731322	0.0007313216	832379	0.0008323788
N = 500	1012296	0.0010122963	1024942	0.001024942
N = 600	1168866	0.0011688656	1183494	0.0011834935
N = 700	1363188	0.0013631881	1507808	0.0015078088
N = 800	1401023	0.0014010231	1598709	0.0015987094
N = 900	1722341	0.0017223405	1751382	0.0017513822
N = 1000	2001427	0.0020014273	2253014	0.0022530143

Figure 5.1.A: Average insertion times for AVL & Splay Tree

5.2 Graph Presentation

Using the data presented in **Figure 5.1.A** the graph of time (y) against set sizes (x) was plotted to better compare the efficiency of the Splay and AVL Trees.

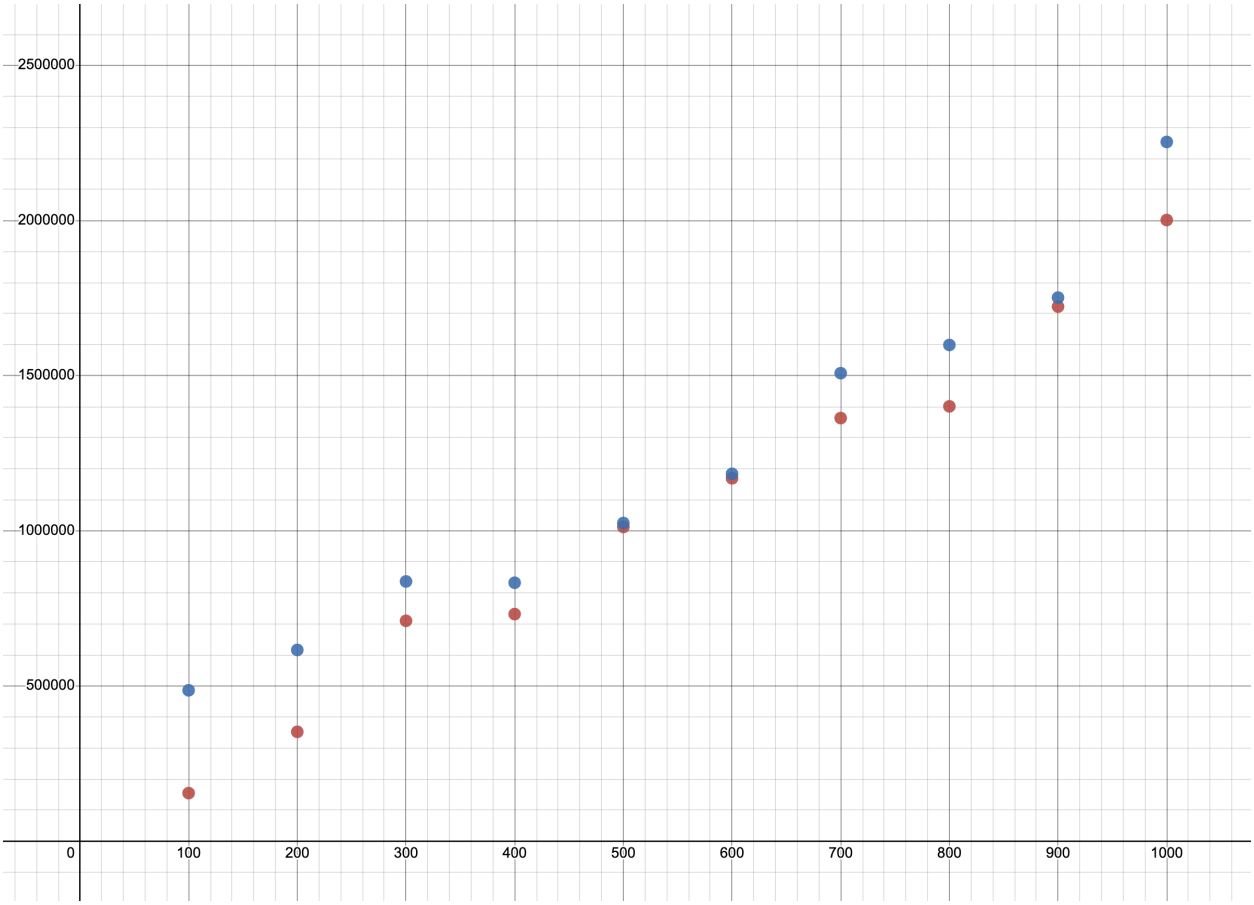

Figure 5.2.A: Averages' points

RED — SPLAY TREE
BLUE — AVL TREE

To achieve a better comparison the average of the points is plotted in **Figure 5.2.A** will be graphed; using the following formula (Exponential Regression):

$$y \sim a b^{x} \{0 < x\}$$

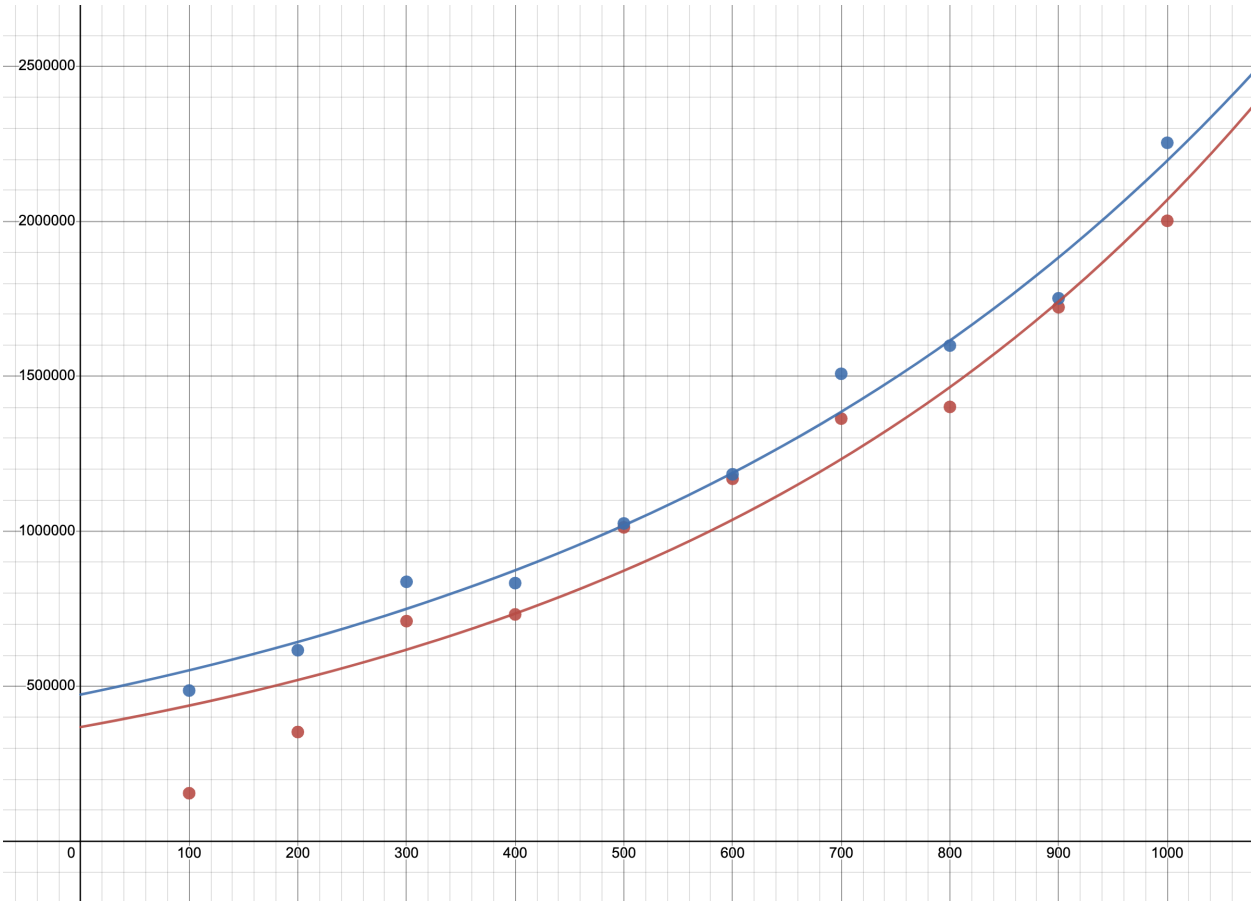

Figure 5.2.B: Exponential Regression of the average points

RED — SPLAY TREE
BLUE — AVL TREE

6. Test Analysis

6.1 Hypothesis Evaluation

As presented in **Figure 5.2.B**, the Splay Tree is proven to be more time efficient than the AVL Tree, which matches my hypothesis. However, my other hypothesis about the growing gap between the efficiency of the two trees was proven to be somewhat false. Referring to **Figure 5.2.B**, from the (0,0) to (500,1018619) coordinates, the gap between the points can be seen gradually increasing (approximately), as expected. However, from the (500,1018619) to (1000,2127220.5) coordinates, the gap between the points can be seen gradually decreasing (approximately). To understand why this is happening, I decided

to look at the gap between the average values of each set size, for each tree individually; by taking their difference $(Avg_2 - Avg_1)$ & $(Avg_{splay} - Avg_{avl})$.

Interestingly, after calculating the average gaps, I saw a relation between them. From set size 100 to 500, the time gap between the tests is actually decreasing and after approximately collapsing with each other in the set size of 500 to 600, they start increasing from set size 700 to 1000. This relation can be clearly seen in **Figure 5.2.A**. However, by calculating the gap between the average times in the AVL Tree, I noticed that the AVL Tree has fewer gaps between each set size than the Splay (lower slope). That suggests, that at some point the AVL Tree will become more efficient than the Splay.

I was stunned and wondered why this is happening, while the bigger the set sizes get, AVL Tree's re-balancing algorithm will become more complex and require more restructuring, as the Splaying system will remain. After further research, I found out that the AVL Tree is able to perform lookups in parallel, unlike the Splay Tree, and therefore saves time.

Another reason which came to my mind was the order in the data being inserted into the trees. As we wanted to prevent the Splay tree from becoming linear, the data was inserted randomly, however for the AVL Tree, the data was linear **(Appendix A)**. This may have effected the process of re-balancing or Splaying in each tree and made the AVL Tree perform faster, with larger data. However as this will not be the case in real-life applications, the AVL's advantage is not considered.

Therefore we can argue, that the data represented in this experiment, however correct, but are not matched with the real-life applications of the trees from an "overall" scope. However, for this research, they can clearly prove **the better time efficiency of the Splay Tree, compared to AVL upon insertion of data.**

According to the experiment's result averages which vary from approximately $1.5 \cdot 10^5 - 2 \cdot 10^6$ for the Splay Tree, and approximately $4.8 \cdot 10^5 - 2.2 \cdot 10^6$ for the Splay Tree, it can be suggested that the Splay Tree is far faster than the AVL in most cases, especially not multithreaded.

6.2 Relation Analysis

By observing the data above, we can see a logarithmic relationship being in place between the trees, as each tree have a variance higher than 0.9 which indicates a reasonable fit for a logarithmic model between time and the size of sets in this comparison.

7. Conclusion

The experiment investigated the relationship between time and the size of sets being inserted into the Spay Tree and the AVL Tree, using the background theory explained. The theory explored helped me build a hypothesis about the Splay Tree being more time efficient, which was later proven to be true. To take it further we also discussed the relation between the time gaps in the data and the logarithmic relationship of time and size.

As the experiment was modeling an operation taking place every second and over trillions of data, I came up with a creative method to test these trees upon insertion, by inserting randomly in the Spay Tree and using ordered sets to ensure the requirement of restructuring for the AVL Tree. This proved to me that although the Splay Tree is faster, at times is more efficient to put the AVL into use. Hence, my conclusion is: for applications having multithreaded environments with a large number of lookups required, the AVL Tree can perform better, as it can run parallel tasks. But in applications in which a large number of data are being inserted into a tree, the Splay Tree will be much faster and will minimize the total run-time of tree lookups, therefore they are more efficient.

Splay trees are also more memory-efficient than AVL trees because they do not need to store balance information in the nodes. However, again because of the parallel working functionality, the AVL Tree may perform better in multithreaded environments.

My answer to the research question of this research is that the efficiency of the two trees is dependent on the task being completed, although in the terms of time complexity upon insertion of values the Splay Tree will perform more efficiently in ordinary applications. Therefore we can conclude that an AVL Tree is better used in an environment or database with frequent lookups and fewer insertion and

deletion actions, like a dictionary. A Splay Tree on the hand is better used on data structures that are frequently modified, as it can handle new node actions faster and more efficiently, like the ‘rope’ data structure, when various modifications are done in the data structure.

Bibliography

[in alphabetical order]

[1] Allen, Peter K. “Splay Trees.” CS CU, Columbia University, 2014, https://www.cs.columbia.edu/~allen/S14/NOTES/splaytrees.pdf.

[2] Geeks for Geeks. “Ropes Data Structure (Fast String Concatenation).” GeeksforGeeks, July 2022, https://www.geeksforgeeks.org/ropes-data-structure-fast-string-concatenation/.

[3] Hadzilacos, Vassos. “Time Complexity of Algorithms .” Cs.toronto.edu, University of Toronto, https://www.cs.toronto.edu/~vassos/teaching/c73/handouts/brief-complexity.pdf.

[4] Java Point. “Binary Search Tree .” www.javatpoint.com, https://www.javatpoint.com/binary- search-tree.

[5] Mareš, Martin. “Lecture Notes on Data Structures.” Martin Mareš: Lecture Notes on Data Structures, University Canada West, Feb. 2020, https://mj.ucw.cz/vyuka/dsnotes/02-splay.pdf.

[6] Massachusetts Institute of Technology . “Big O Notation .” Big O Notation , MIT, http:// web.mit.edu/16.070/www/lecture/big_o.pdf.

[7] Mount, Dave. “CMSC 420: Lecture 5 AVL Trees - UMD.” AVL Trees , University of Maryland , 2020, https://www.cs.umd.edu/class/fall2020/cmsc420-0201/Lects/lect05-avl.pdf.

[8] Tutorials Point Team. “Data Structure and Algorithms - AVL Trees.” Tutorials Point, https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm.

[9] Weiss , Mark A. “AvlNode.java.” Users.cs.fiu.edu, Knight Foundation School of Computing and Information Sciences , https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlNode.java.

[10] Weiss, Mark A. “AvlTree.java.” Users.cs.fiu.edu, Knight Foundation School of Computing and Information Sciences , https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java.

[11] Weiss, Mark A. “SplayTree.java.” Users.cs.fiu.edu, Knight Foundation School of Computing and Information Sciences , https://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/SplayTree.java.

[12] Yse, Diego Lopez. “Essential Programming: Time Complexity.” Medium, Towards Data Science, 22 Nov. 2020, https://towardsdatascience.com/essential-programming-time-complexity-a95bb2608cac.

Appendices

Appendix A: Experiment's Program

```
set_Min = 1
set_Max = 100 //changes

for(int trial = 1; trial <= 10; trial++) {
    AvlTree avl = new AvlTree();
    SplayTree spl = new SplayTree();


long startAVL = System.nanoTime();
for (double a = set_Min; a <= set; a++) {
    avl.insert(a);
}
long endAVL = System.nanoTime();

long startSPL = System.nanoTime();
ArrayList<Integer> list = new ArrayList<Integer>();
for (int b = set_Min; b <= set_Max; b++) {
    list.add(new Integer(b));
}
    Collections.shuffle(list);
    for (int b = 0; b < set_Max; b++) {
        spl.insert(b);
}
long endSPL = System.nanoTime();

System.out.println("AVL Trial " + trial + ": " + (endAVL - startAVL));
System.out.println("Splay Trial " + trial + ": " + (endSPL - startSPL));

}
```

Appendix B: Raw Data

B1: Splay Tree

Set Sizes	100		200		300		400		500	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	201744	0.000201744	436492	0.000436492	2078642	0.002078642	2197050	0.00219705	3173905	0.003173905
Test 2	156784	0.000156784	282684	0.000282684	252479	0.000252479	856579	0.000856579	1079528	0.001079528
Test 3	93055	0.000093055	323414	0.000323414	709368	0.000709368	433070	0.00043307	1397736	0.001397736
Test 4	149931	0.000149931	99583	0.000099583	632834	0.000632834	144919	0.000144919	357274	0.000357274
Test 5	71035	0.000071035	244933	0.000244933	870898	0.000870898	433070	0.00043307	1475498	0.001475498
Test 6	328255	0.000328255	1099802	0.001099802	178642	0.000178642	635810	0.00063581	995519	0.000995519
Test 7	248163	0.000248163	336590	0.00033659	564300	0.0005643	543831	0.000543831	514676	0.000514676
Test 8	163265	0.000163265	208628	0.000208628	1171240	0.00117124	1252773	0.001252773	354114	0.000354114
Test 9	52125	0.000052125	292247	0.000292247	474865	0.000474865	433070	0.00043307	659687	0.000659687
Test 10	80748	0.000080748	195845	0.000195845	164060	0.00016406	383044	0.000383044	115026	0.000115026
Avg.	**154510**	**0.0001545105**	**352022**	**0.0003520218**	**709732**	**0.0007097328**	**731322**	**0.0007313216**	**1012296**	**0.0010122963**

Set Sizes	600		700		800		900		1000	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	2416212	0.002416212	1160855	0.001160855	3017459	0.003017459	4477437	0.004477437	1841080	0.00184108
Test 2	999352	0.000999352	3492116	0.003492116	1693436	0.001693436	3818446	0.003818446	5390321	0.005390321
Test 3	1213070	0.00121307	405161	0.000405161	559638	0.000559638	329510	0.00032951	3167011	0.003167011
Test 4	1001918	0.001001918	516346	0.000516346	1468798	0.001468798	527725	0.000527725	1183997	0.001183997
Test 5	1973998	0.001973998	2460033	0.002460033	2336282	0.002336282	872963	0.000872963	715350	0.00071535
Test 6	670895	0.000670895	1870045	0.001870045	981249	0.000981249	2001991	0.002001991	1459759	0.001459759
Test 7	313603	0.000313603	966336	0.000966336	1440729	0.001440729	1602935	0.001602935	1431305	0.001431305
Test 8	2026700	0.0020267	318571	0.000318571	599940	0.00059994	1924004	0.001924004	3036682	0.003036682
Test 9	298094	0.000298094	1429579	0.001429579	720056	0.000720056	703180	0.00070318	782415	0.000782415
Test 10	774814	0.000774814	1012839	0.001012839	1192644	0.001192644	965214	0.000965214	1006353	0.001006353
Avg.	**1168866**	**0.0011688656**	**1363188**	**0.0013631881**	**1401023**	**0.0014010231**	**1722341**	**0.0017223405**	**2001427**	**0.0020014273**

B2: AVL Tree

Set Sizes	100		200		300		400		500	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	110863	0.000110863	1102057	0.001102057	889309	0.000889309	382803	0.000382803	921303	0.000921303
Test 2	246155	0.000246155	390278	0.000390278	563106	0.000563106	791573	0.000791573	792251	0.000792251
Test 3	480647	0.000480647	582118	0.000582118	661011	0.000661011	1703945	0.001703945	589451	0.000589451
Test 4	372377	0.000372377	348519	0.000348519	2170925	0.002170925	1729676	0.001729676	636593	0.000636593
Test 5	1452811	0.001452811	101846	0.000101846	175012	0.000175012	621867	0.000621867	924509	0.000924509
Test 6	1051363	0.001051363	175065	0.000175065	961058	0.000961058	591686	0.000591686	1730891	0.001730891
Test 7	282112	0.000282112	193408	0.000193408	492268	0.000492268	871308	0.000871308	497046	0.000497046
Test 8	333940	0.00033394	2069873	0.002069873	1036203	0.001036203	451748	0.000451748	1927493	0.001927493
Test 9	411330	0.00041133	494060	0.00049406	656422	0.000656422	628317	0.000628317	501453	0.000501453
Test 10	115329	0.000115329	700666	0.000700666	760935	0.000760935	550865	0.000550865	1728430	0.00172843
Avg.	485692	0.0004856927	615789	0.000615789	836625	0.0008366249	832379	0.0008323788	1024942	0.001024942

Set Sizes	600		700		800		900		1000	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	662959	0.000662959	2634935	0.002634935	4157917	0.004157917	3540938	0.003540938	6721712	0.006721712
Test 2	2402168	0.002402168	1357045	0.001357045	2046193	0.002046193	2232438	0.002232438	3846201	0.003846201
Test 3	493687	0.000493687	1127999	0.001127999	166998	0.000166998	652454	0.000652454	2037499	0.002037499
Test 4	811742	0.000811742	712934	0.000712934	1834990	0.00183499	4720899	0.004720899	1455077	0.001455077
Test 5	1398992	0.001398992	881823	0.000881823	1707961	0.001707961	598151	0.000598151	408907	0.000408907
Test 6	914825	0.000914825	1972825	0.001972825	2654468	0.002654468	1616466	0.001616466	674037	0.000674037
Test 7	3630656	0.003630656	665437	0.000665437	1497781	0.001497781	1008058	0.001008058	1787696	0.001787696
Test 8	706410	0.00070641	3013506	0.003013506	1095276	0.001095276	1041388	0.001041388	370219	0.000370219
Test 9	495274	0.000495274	1072825	0.001072825	218558	0.000218558	1251639	0.001251639	4183004	0.004183004
Test 10	318222	0.000318222	1638759	0.001638759	606952	0.000606952	851391	0.000851391	1045791	0.001045791
Avg.	1183494	0.0011834935	1507808	0.0015078088	1598709	0.0015987094	1751382	0.0017513822	2253014	0.0022530143

Appendix C: Splay Tree's Java Code (Mark Weiss)

```
// SplayTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// boolean contains( x )  --> Return true if x is found
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Throws UnderflowException as appropriate

/**
 * Implements a top-down splay tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class SplayTree<AnyType extends Comparable<? super AnyType>>
{
    /**
     * Construct the tree.
     */
    public SplayTree( )
    {
        nullNode = new BinaryNode<AnyType>( null );
        nullNode.left = nullNode.right = nullNode;
        root = nullNode;
    }

        private BinaryNode<AnyType> newNode = null;   // Used between different
inserts

    /**
     * Insert into the tree.
     * @param x the item to insert.
     */
```

```
public void insert( AnyType x )
{
   if( newNode == null )
      newNode = new BinaryNode<AnyType>( null );
   newNode.element = x;

   if( root == nullNode )
   {
      newNode.left = newNode.right = nullNode;
      root = newNode;
   }
   else
   {
      root = splay( x, root );

      int compareResult = x.compareTo( root.element );

      if( compareResult < 0 )
      {
         newNode.left = root.left;
         newNode.right = root;
         root.left = nullNode;
         root = newNode;
      }
      else
      if( compareResult > 0 )
      {
         newNode.right = root.right;
         newNode.left = root;
         root.right = nullNode;
         root = newNode;
      }
      else
         return;   // No duplicates
   }
   newNode = null;   // So next insert will call new
}
```

```
/**
 * Remove from the tree.
 * @param x the item to remove.
 */
public void remove( AnyType x )
{
    if( !contains( x ) )
        return;

    BinaryNode<AnyType> newTree;

        // If x is found, it will be splayed to the root by contains
    if( root.left == nullNode )
        newTree = root.right;
    else
    {
        // Find the maximum in the left subtree
        // Splay it to the root; and then attach right child
        newTree = root.left;
        newTree = splay( x, newTree );
        newTree.right = root.right;
    }
    root = newTree;
}

/**
 * Find the smallest item in the tree.
 * Not the most efficient implementation (uses two passes), but has correct
 *     amortized behavior.
 * A good alternative is to first call find with parameter
 *     smaller than any item in the tree, then call findMin.
 * @return the smallest item or throw UnderflowException if empty.
 */
public AnyType findMin( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    BinaryNode<AnyType> ptr = root;

    while( ptr.left != nullNode )
```

```
        ptr = ptr.left;

    root = splay( ptr.element, root );
    return ptr.element;
}

/**
 * Find the largest item in the tree.
 * Not the most efficient implementation (uses two passes), but has correct
 *     amortized behavior.
 * A good alternative is to first call find with parameter
 *     larger than any item in the tree, then call findMax.
 * @return the largest item or throw UnderflowException if empty.
 */
public AnyType findMax( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    BinaryNode<AnyType> ptr = root;

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

    root = splay( ptr.element, root );
    return ptr.element;
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return true if x is found; otherwise false.
 */
public boolean contains( AnyType x )
{
    if( isEmpty( ) )
        return false;

    root = splay( x, root );

    return root.element.compareTo( x ) == 0;
}
```

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

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

      private BinaryNode<AnyType> header = new BinaryNode<AnyType>( null ); // For splay

    /**
     * Internal method to perform a top-down splay.
     * The last accessed node becomes the new root.
     * @param x the target item to splay around.
     * @param t the root of the subtree to splay.
     * @return the subtree after the splay.
     */
    private BinaryNode<AnyType> splay( AnyType x, BinaryNode<AnyType> t )
    {
        BinaryNode<AnyType> leftTreeMax, rightTreeMin;
        header.left = header.right = nullNode;
        leftTreeMax = rightTreeMin = header;
        nullNode.element = x;   // Guarantee a match
        for( ; ; )
        {
            int compareResult = x.compareTo( t.element );

            if( compareResult < 0 )
            {
                if( x.compareTo( t.left.element ) < 0 )
                    t = rotateWithLeftChild( t );
```

```
            if( t.left == nullNode )
                break;
            // Link Right
            rightTreeMin.left = t;
            rightTreeMin = t;
            t = t.left;
        }

        else if( compareResult > 0 )
        {
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            if( t.right == nullNode )
                break;
            // Link Left
            leftTreeMax.right = t;
            leftTreeMax = t;
            t = t.right;
        }
        else
            break;
    }

    leftTreeMax.right = t.left;
    rightTreeMin.left = t.right;
    t.left = header.right;
    t.right = header.left;
    return t;
  }


  /**
   * Rotate binary tree node with left child.
   * For AVL trees, this is a single rotation for case 1.
   */
        private static <AnyType> BinaryNode<AnyType>
rotateWithLeftChild( BinaryNode<AnyType> k2 )
  {
    BinaryNode<AnyType> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
```

```
    }

    /**
     * Rotate binary tree node with right child.
     * For AVL trees, this is a single rotation for case 4.
     */
            private static <AnyType> BinaryNode<AnyType>
rotateWithRightChild( BinaryNode<AnyType> k1 )
    {
        BinaryNode<AnyType> k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        return k2;
    }

    // Basic node stored in unbalanced binary search trees
    private static class BinaryNode<AnyType>
    {
            // Constructors
        BinaryNode( AnyType theElement )
        {
            this( theElement, null, null );
        }

                        BinaryNode( AnyType theElement, BinaryNode<AnyType> lt,
BinaryNode<AnyType> rt )
        {
            element  = theElement;
            left     = lt;
            right    = rt;
        }

        AnyType element;            // The data in the node
        BinaryNode<AnyType> left;   // Left child
        BinaryNode<AnyType> right;  // Right child
    }

    private BinaryNode<AnyType> root;
    private BinaryNode<AnyType> nullNode;


        // Test program; should print min and max and nothing else
```

```
    public static void main( String [ ] args )
    {
        SplayTree<Integer> t = new SplayTree<Integer>( );
        final int NUMS = 40000;
        final int GAP  =   307;

        System.out.println( "Checking... (no bad output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( i );
        System.out.println( "Inserts complete" );

        for( int i = 1; i < NUMS; i += 2 )
            t.remove( i );
        System.out.println( "Removes complete" );

        if( t.findMin( ) != 2 || t.findMax( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 2; i < NUMS; i += 2 )
            if( !t.contains( i ) )
                System.out.println( "Error: find fails for " + i );

        for( int i = 1; i < NUMS; i += 2 )
            if( t.contains( i ) )
                System.out.println( "Error: Found deleted item " + i );
    }
}
```

Appendix D: AVL Tree's Java Code (Mark Weiss)

D1: AvlTree.java

```
// 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;


    // Test program
public static void main( String [ ] args )
{
    AvlTree t = new AvlTree( );
    final int NUMS = 4000;
```

```
        final int GAP  =  37;

        System.out.println( "Checking... (no more output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new MyInteger( i ) );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 1 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 1 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 1; i < NUMS; i++ )
            if( ((MyInteger)(t.find( new MyInteger( i ) ))).intValue( ) != i )
                System.out.println( "Find error1!" );
    }
}
```

D2: AvlNode.java

```
// 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
}
```

Appendix E: Binary Search Tree C Code

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
   int key;
   struct node *left;
   struct node *right;
};

struct node *getNewNode(int val)
{
   struct node *newNode = malloc(sizeof(struct node));
   newNode->key   = val;
   newNode->left  = NULL;
   newNode->right = NULL;

   return newNode;
}

struct node *insert(struct node *root, int val)
{
   if(root == NULL)
      return getNewNode(val);

   if(root->key < val)
      root->right = insert(root->right,val);

   else if(root->key > val)
      root->left = insert(root->left,val);

   return root;
}

void inorder(struct node *root)
{
   if(root == NULL)
      return;
   inorder(root->left);
```

```
    printf("%d ",root->key);
    inorder(root->right);
}

int main()
{
    struct node *root = NULL;
    root = insert(root,100);.
    root = insert(root,50);
    root = insert(root,150);
    root = insert(root,50);

    inorder(root);

    return 0;
}
```