Extended Essay

In terms of time complexity, to what extent is a Red-Black Tree more algorithmically efficient than an Adelson-Velsky and Landis Tree upon the deletion of values?

By Catherine Franco
Subject: Computer Science
Word Count: 3,506
(excluding diagrams, sidenotes, citations, appendices, bibliography, and table of contents)

Table of Contents

1. Introduction

The following research question will be explored in this writing:

In terms of time complexity, to what extent is a Red-Black Tree more algorithmically efficient than an Adelson-Velsky and Landis Tree upon the deletion of values?

This essay intends to generally investigate binary search trees, which are data structures that contain nodes that follow the following properties:

1. Nodes assigned to the left subtree are less than the root node
2. Nodes assigned to the right subtree are greater than the root node
3. Both subtrees must be binary search trees

This essay will also be focusing on and discussing two specific types of binary search trees: Red-Black Trees and AVL Trees (Adelson-Velsky and Landis). Both, and all types of binary search trees, are extremely useful in many real world applications such as organizing and storing data. However, it's essential to examine the difference between the efficiency of the two types of binary search trees in terms of time complexity and the deletion function of values. Time complexity is the amount of time it takes for an algorithm to complete, represented as a function of input length[1]. Time complexity allows scientists to analyze faster executions and predict requirements for memory usage.

The following section will focus on a more in depth analysis of binary search trees, Red-Black Trees, and Adelson-Velsky and Landis Trees; along with a hypothesis.

[1] Great Learning Team, et al. "Why Is Time Complexity Essential and What Is Time Complexity?" *GreatLearning Blog: Free Resources What Matters to Shape Your Career!*, 8 Jan. 2022, https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/

2. Key Concepts

2.1 Binary Search Trees

Binary search trees are node-based trees (nodes are equivalent to the items) used to prioritize the organization of inputted values. Binary search trees are often mistaken with binary trees, however binary search trees are instead variants of binary trees and prioritize the ordering of nodes with searching, insertion, traversal, and deletion. The definition of binary is "made up of two parts or things[2]," which corresponds to binary search trees because each node contains a maximum of up to two children (referred to as items as well.) Binary search trees also differ from normal binary trees because they hold the characteristic that all values in the left subtree are less than the root node, and all the values in the right subtree are greater than the root node.

Figure 2.1.1 represents a diagram of a typical binary search tree.

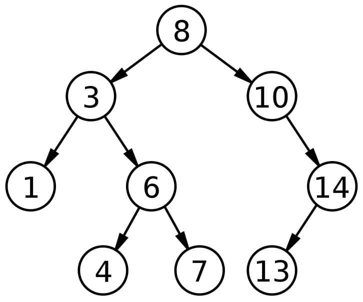


Figure 2.1.1

[2] "Binary Meaning." *Binary Meaning | Best 27 Definitions of Binary*, https://www.yourdictionary.com/binary

The delete function is "used to delete the specified node from a binary search tree.[3]" The delete operation comprises of three possibilities:

1. Removing the node with no children from the tree. The parent would be replaced with *nil* as its child. Diagram shown in Figure 2.1.2. Pseudocode shown in ***Appendix A***.
2. Deleting a node with one child. The node would be copied to the node and then deleted. The child will be elevated either left or right to the node's position. Shown in Figure 2.1.3. Pseudocode shown in ***Appendix A***.
3. Deleting a node with two children. Once the inorder successor of the node is found, the contents of the inorder successor are copied to the node and then the inorder successor proceeds to be deleted[4]. The inorder successor is defined as "the node with the smallest key greater than the key of the input node[5]." Shown in Figure 2.1.3. Pseudocode shown in ***Appendix A***.

The reason Binary Search Trees are frequently used for the organization of nodes is because it's normally considered more efficient than other methods such as a linear search. The worst case of time complexity for binary search trees is considered by this notation: O(log N), while the method of searching linear is notated as O(N). The comparison is graphically shown below in Figure 2.1.4 (via Desmos Graphing).

(All figures below refer to node values in the "delete" portion of the diagram.)

[3]"Deletion in Binary Search Tree - Javatpoint." *Www.javatpoint.com*, https://www.javatpoint.com/deletion-in-binary-search-tree

[4]"Inorder Successor in Binary Search Tree." *GeeksforGeeks*, 10 Jan. 2022, https://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/

[5] "Inorder Successor in Binary Search Tree - Tutorialspoint.dev." *Tutorialspoint.Dev*, https://tutorialspoint.dev/data-structure/binary-search-tree/inorder-successor-in-binary-search-tree

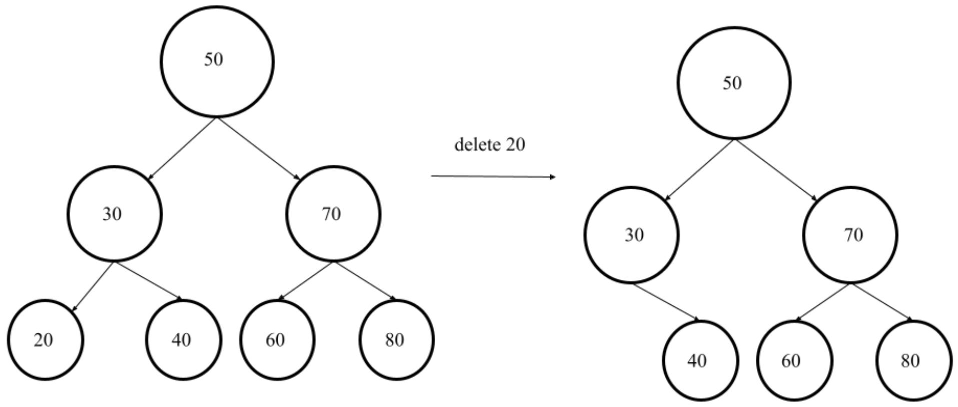


Figure 2.1.1

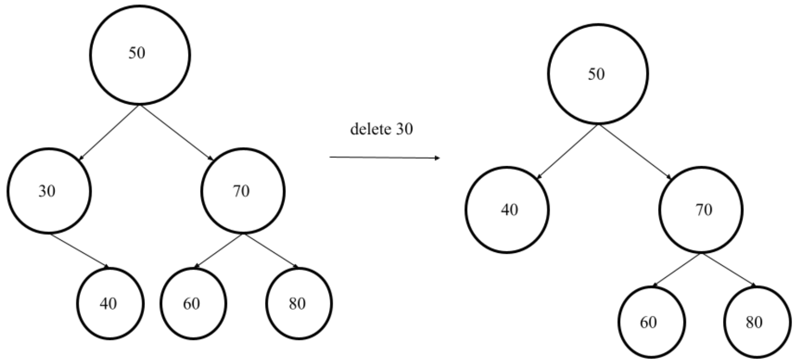


Figure 2.1.2

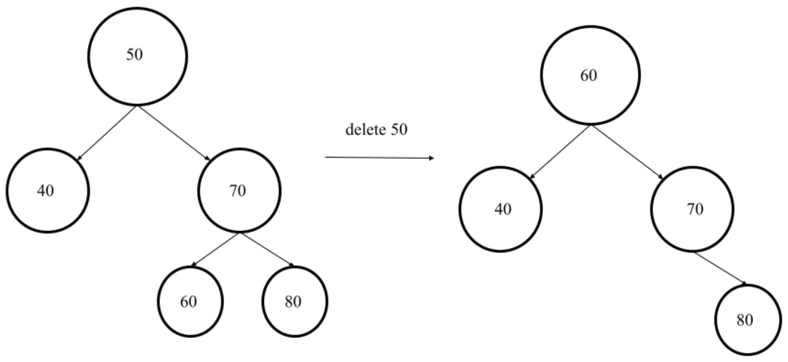


Figure 2.1.3

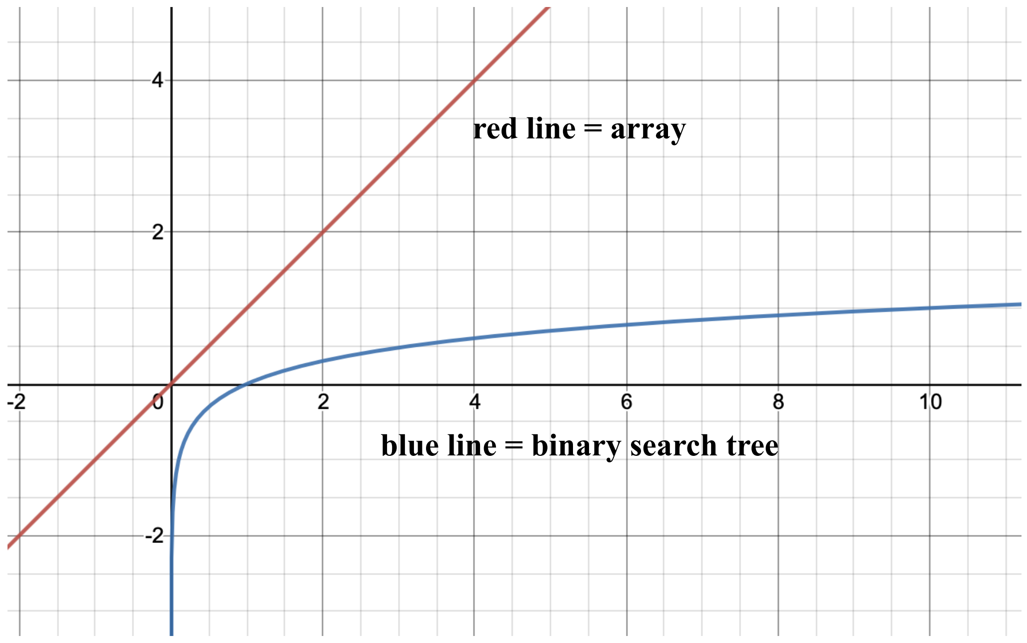


Figure 2.1.4

Although it may seem clear binary search trees seem to have an advantage over linear search methods, the scenario changes when an unbalanced data tree is used. In fact, if the values 1, 2, 3, 4, and 5 are used for a binary search tree, the worst case efficiency is equivalent to a linear search's efficiency (O(5)). Figure 2.1.5 represents an unbalanced data tree.

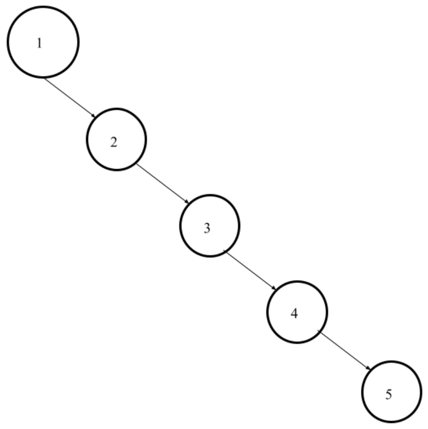


Figure 2.1.5

To avoid this issue, algorithms must be balanced. Balanced algorithms will be implemented in order to ensure that structure is maintained within the Red-Black trees and AVL trees. However, Red-Black trees and AVL trees have different methods and perceptions of balance within their algorithms, and those perceptions will be explored more in depth in further sections.

2.2 Red Black Trees

Red black trees are special types of self balancing binary search trees. What makes a red black tree is the nodes; each node contains extra bits of "red" or "black" in order to maintain balance within the algorithm. During processes like deletion and insertion, this fundamental is important in balance. Below are the requirements that make and apply to red black trees[6]:

1. Each node in the tree is red or black.
2. The root node is ALWAYS black.
3. Two red nodes CAN'T appear consecutively.
4. There are the same number of black nodes for each path in the tree.

Figure 2.2.1 represents a case in which one of the rules is broken (3rd). When a rule is broken, rebalancing occurs.

Below are explanations of Red Black Trees' rebalancing algorithm for deletion. First, the algorithm will be explained through tree diagrams, and then through actual programmed implementation.

There are several ways to complete the deletion method, and each way is for separate scenarios. For the following scenarios[7], let X be the node to be deleted and Y be the child that replaces X (*the delete "value" in the diagrams refers to a node value)*.

[6] "Red-Black Tree: Set 1 (Introduction)." *GeeksforGeeks*, 18 Dec. 2021, https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

[7]"Binary Search Tree: Set 2 (Delete)." *GeeksforGeeks*, 1 Feb. 2022, https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

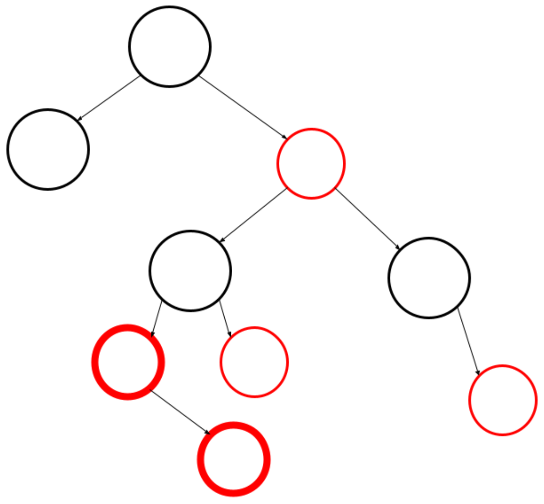
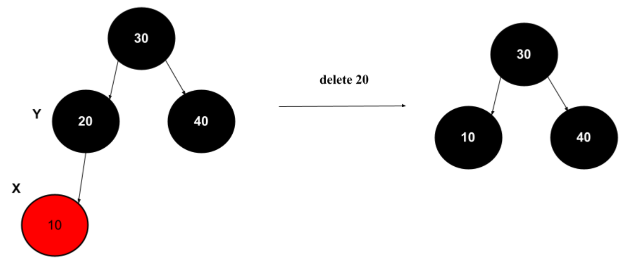


Figure 2.2.1 (highlighted red portion is the 3rd rule being broken, two red consecutive nodes)

1. A normal, standard binary search tree deletion can be completed (showcased previously). This method is only used in cases in which a node is leaf or has one child.



2. If either X or Y is red, the replaced child is marked black. X is a parent of Y, so both cannot be the color red. According to the third rule as well, there can't be two consecutive red nodes. An example of this case is shown in Figure 2.2.2.

Figure 2.2.2 (if either X or Y is black)

3. If both X and Y are black:

- If Y is double black, convert to a single black.
- If X is leaf, Y is NULL and the color black. Deleting a black leaf causes a double black.

The figures below (2.2.3-2.2.7) show explained representations and diagrams of scenario 3 and its possible cases (*the delete "value" refers to a node value)*.

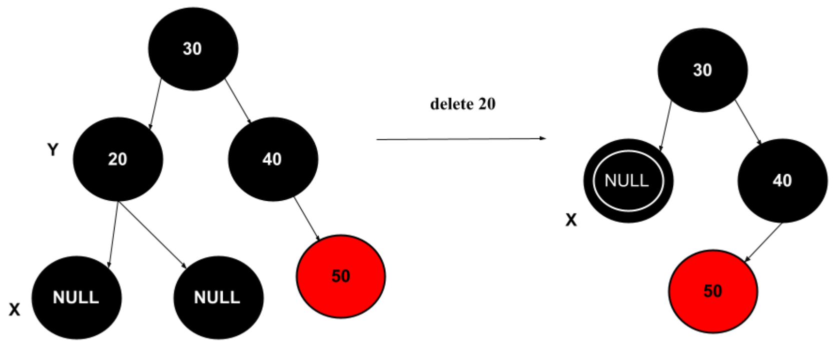


Figure 2.2.3 (if both X and Y are black; Y is leaf, X is null, deletion of black leaf = double black)

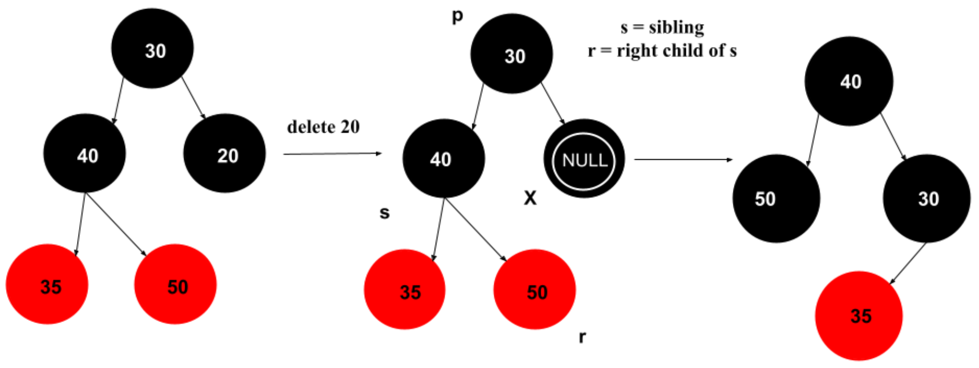


Figure 2.2.4 (If atleast one black sibling's children is red, perform rotations)

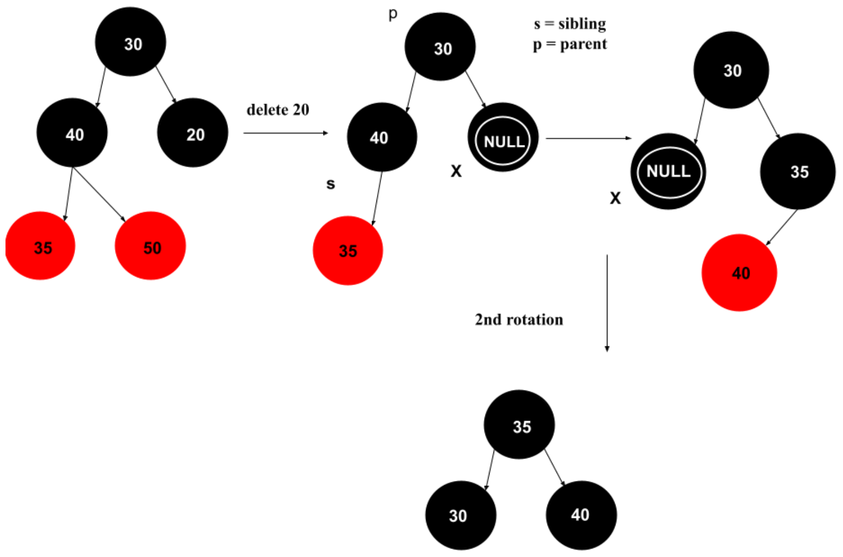


Figure 2.2.5 (If the sibling is the right child of the parent)

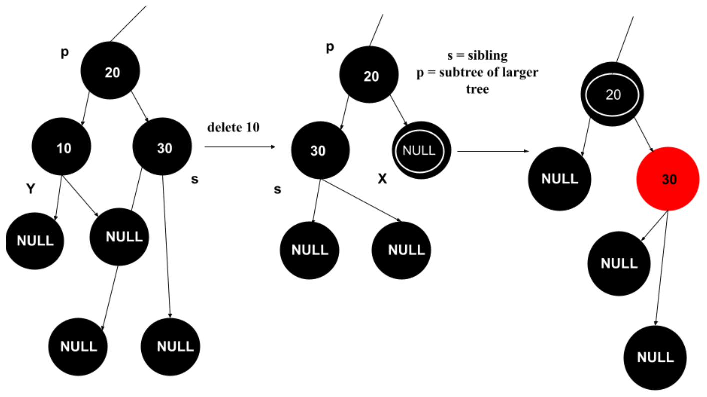


Figure 2.2.6 (If both children under a black sibling are black)

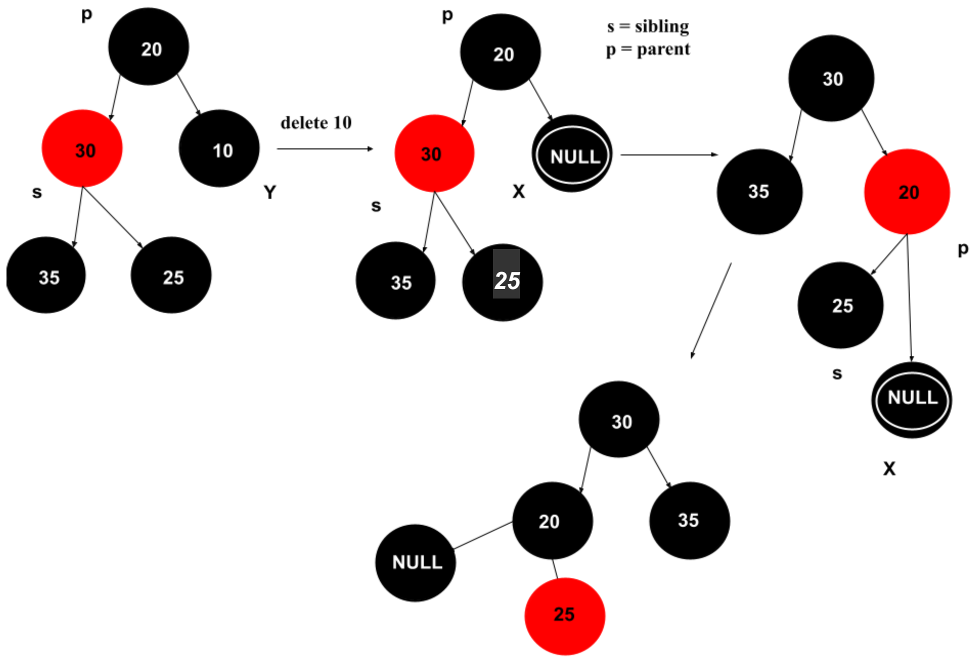
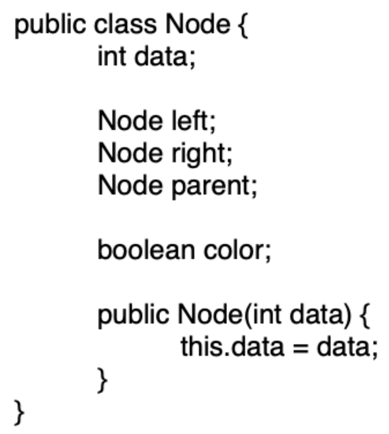


Figure 2.2.7 (If sibling is red; perform rotation)

Essentially, after deletion occurs, the Red Black Tree must be rebalanced after a black node is deleted because it violates black depth property. The Red Black Tree algorithm can be explored further through the coded implementation referred to in ***Appendix B***, as well as pseudocode referred to in ***Appendix A***. Below is an analysis of a coded Java implementation of the deletion method (borrowed from the source in subscription[8]). The coded implementation is not an exact representation of the restructuring process described above, however, it has the same results.

To begin, there must be a reference to the original Java implementation for a binary search tree:



```
public class Node {
        int data;

        Node left;
        Node right;
        Node parent;

        boolean color;

        public Node(int data) {
                this.data = data;
        }
}
```

Figure 2.2.8 (refer to Appendix B)

---

[8]Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/RedBlackTree.java

The statement “boolean color” refers to the determination for each node being either red or black. Red can be classified as false and black as true.

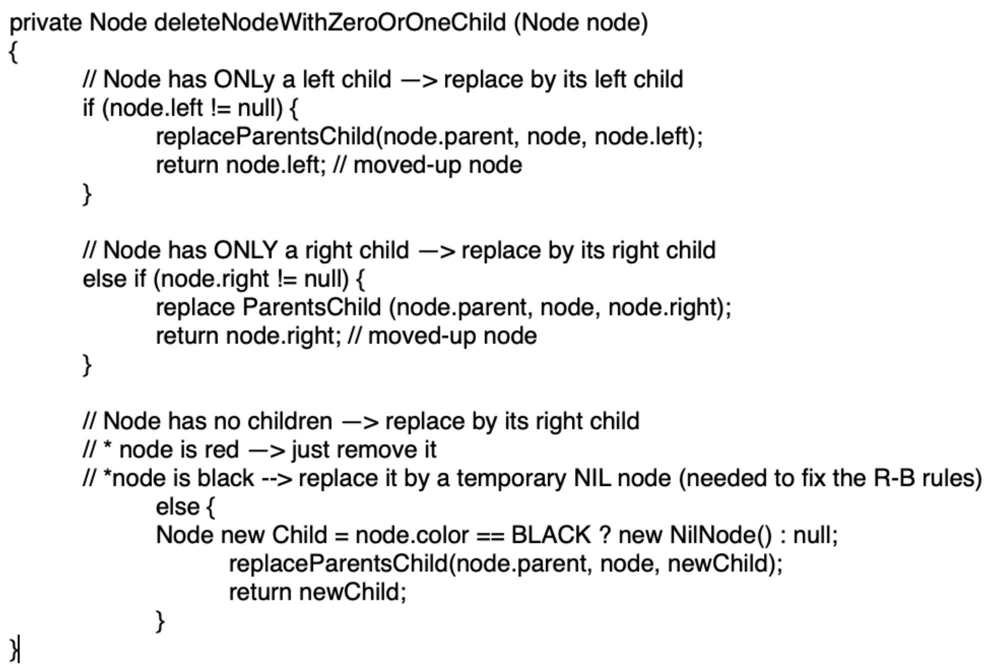
Below is the Java implementation[9] of the actual deletion method (deleteNode()) of a red black tree:

```
public void deleteNode(int key) {
    Node node = root;

    while (node != null && node.data != key) {

        if (key < node.data) {
            node = node.left;
        } else {
            node = node.right;
        }
    }

    if (node == null) {
        return;
    }

Node movedUpNode;
boolean deletedNodeColor;

    // Node has zero or one child
    if (node.left == null || node.right == null) {
        movedUpNode = deletedNodeWithZeroOrOneChild(node);
        deletedNodeColor = node.color;
    }

    // Node has two children
    else {
        Node inOrderSuccessor = findMinimum(node.right);

        node.data = inOrderSuccessor.data;

        movedUpNode = deleteNodeWithZeroOneChild(inOrderSuccessor);
        deletedNodeColor = inOrderSuccessor.color;
    }

    if (deletedNodeColor == BLACK) {
        fixRedBlackPropertiesAfterDelete(movedUpNode);

        if (movedUpNode.getClass() == NilNode.class) {
            replaceParentsChild(movedUpNode.parent, movedUpNode, null);
        }
    }
}
```

Figure 2.2.9 (deletion method, refer to Appendix B)

[9]Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/RedBlackTree.java

This first section[10] including the while statement initally finds the node to be deleted then proceeds to traverse the tree left to right depending on the key. The if (node.left == null || node.right == null) statement determines whether the node has zero or one child, while the else statement below it finds the inorder successor if the node has two children. If the node has at most one child, we call the method deleteNodeWithZeroOrOneChild()[11]:



```
private Node deleteNodeWithZeroOrOneChild (Node node)
{
        // Node has ONLy a left child —> replace by its left child
        if (node.left != null) {
                replaceParentsChild(node.parent, node, node.left);
                return node.left; // moved-up node
        }

        // Node has ONLY a right child —> replace by its right child
        else if (node.right != null) {
                replace ParentsChild (node.parent, node, node.right);
                return node.right; // moved-up node
        }

        // Node has no children —> replace by its right child
        // * node is red —> just remove it
        // *node is black --> replace it by a temporary NIL node (needed to fix the R-B rules)
                else {
                Node new Child = node.color == BLACK ? new NilNode() : null;
                        replaceParentsChild(node.parent, node, newChild);
                        return newChild;
                }
}
```

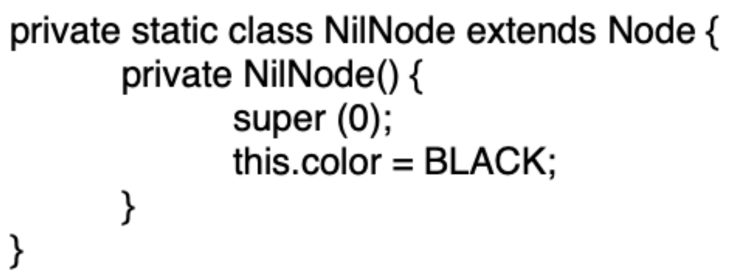
Figure 2.2.10 (deleteNodewithZeroOrOneChild() method)

[10], [11] Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/RedBlackTree.java

As called in Figure 2.2.10, if the node has ONLY a left child, it will be replaced by its corresponding left child. If the node has ONLY a right child, it will be replaced by its corresponding right child.

A special case is where a black node with no children is deleted, which is solved through the final else statement in Figure 2.2.10. However, deleting a black node also results in the tree needing to be rebalanced. The rebalancing algorithm will be discussed shortly.

If the node that is deleted does not hold any children, one of its NIL leaves transfers to its current position. A special placeholder is used to navigate from the NIL leaf to the parent node[12]:



```
private static class NilNode extends Node {
        private NilNode() {
                super (0);
                this.color = BLACK;
        }
}
```

Figure 2.2.11

However, it's different if the node to be deleted holds two children. The findMinimum() function is used to find the inorder successor[13]:

[12],[13] Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/RedBlackTree.java

```
public Node findMinimum (Node node) {
        while (node.left != null) {
                node = node.left;
        }
        return node
}
```

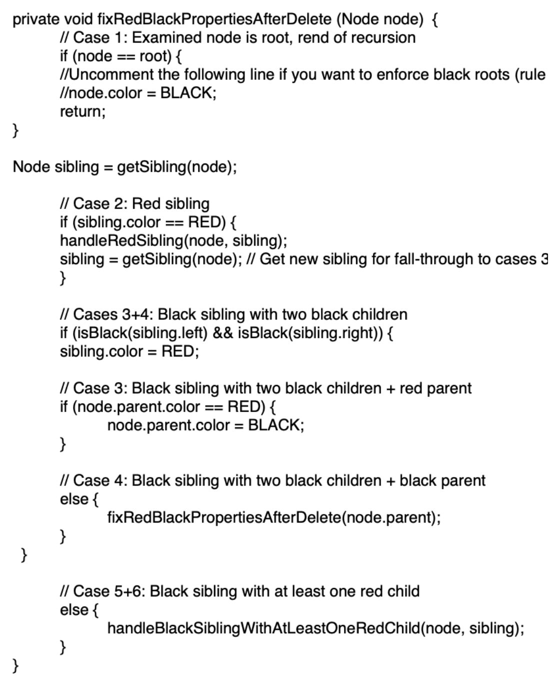
Figure 2.2.12

The data from the inorder successor is then copied to the node bound to be deleted. The previous method, deleteNodeWithZeroOrOneChild(), is then called again to delete the inorder successor from the Red Black tree.

If the node to be deleted is red, there are no violations and no necessity to rebalance/repair the tree. However, if the node to be deleted is black, there is a need for repair. There are six cases that are implemented and commented in the following Java program[14]:

1. Deleted node is root
2. Sibling is red
3. Sibling is black and holds two black children, parent is red
4. Sibling is black and holds two black children, parent is black
5. Sibling is black and has at least one red child, “outer nephew” is black
6. Sibling is black and has at least one red child, “outer nephew” is red

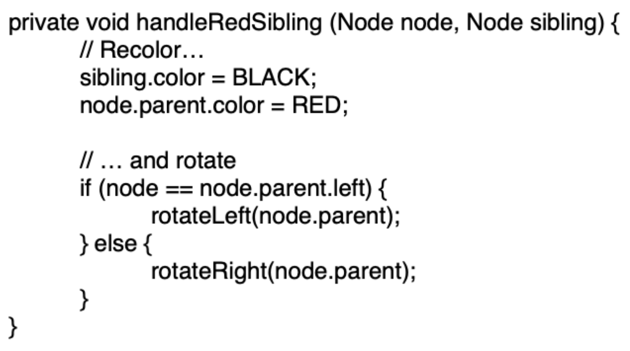
[14] Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/RedBlackTree.java



```
private void fixRedBlackPropertiesAfterDelete (Node node)  {
          // Case 1: Examined node is root, rend of recursion
          if (node == root) {
          //Uncomment the following line if you want to enforce black roots (rule
          //node.color = BLACK;
          return;
}

Node sibling = getSibling(node);

          // Case 2: Red sibling
          if (sibling.color == RED) {
          handleRedSibling(node, sibling);
          sibling = getSibling(node); // Get new sibling for fall-through to cases 3
          }

          // Cases 3+4: Black sibling with two black children
          if (isBlack(sibling.left) && isBlack(sibling.right)) {
          sibling.color = RED;

          // Case 3: Black sibling with two black children + red parent
          if (node.parent.color == RED) {
                    node.parent.color = BLACK;
          }

          // Case 4: Black sibling with two black children + black parent
          else {
                    fixRedBlackPropertiesAfterDelete(node.parent);
          }
 }

          // Case 5+6: Black sibling with at least one red child
          else {
                    handleBlackSiblingWithAtLeastOneRedChild(node, sibling);
          }
}
```

Figure 2.2.13

Case 2 handleRedSibling()[15]:

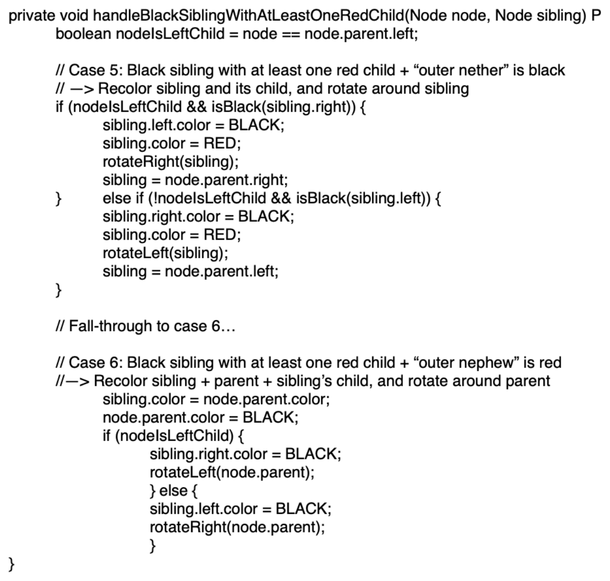


```
private void handleRedSibling (Node node, Node sibling) {
          // Recolor…
          sibling.color = BLACK;
          node.parent.color = RED;

          // … and rotate
          if (node == node.parent.left) {
                    rotateLeft(node.parent);
          } else {
                    rotateRight(node.parent);
          }
}
```

Figure 2.2.14

[15] Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/RedBlackTree.java

Cases 5 and 6 handleBlackSiblingWithAtLeastOneRedChild():



```
private void handleBlackSiblingWithAtLeastOneRedChild(Node node, Node sibling) P
        boolean nodeIsLeftChild = node == node.parent.left;

        // Case 5: Black sibling with at least one red child + “outer nether” is black
        // —> Recolor sibling and its child, and rotate around sibling
        if (nodeIsLeftChild && isBlack(sibling.right)) {
                sibling.left.color = BLACK;
                sibling.color = RED;
                rotateRight(sibling);
                sibling = node.parent.right;
        }       else if (!nodeIsLeftChild && isBlack(sibling.left)) {
                sibling.right.color = BLACK;
                sibling.color = RED;
                rotateLeft(sibling);
                sibling = node.parent.left;
        }

        // Fall-through to case 6…

        // Case 6: Black sibling with at least one red child + “outer nephew” is red
        //—> Recolor sibling + parent + sibling’s child, and rotate around parent
                sibling.color = node.parent.color;
                node.parent.color = BLACK;
                if (nodeIsLeftChild) {
                        sibling.right.color = BLACK;
                        rotateLeft(node.parent);
                        } else {
                        sibling.left.color = BLACK;
                        rotateRight(node.parent);
                        }
}
```

Figure 2.2.15

2.3 Adelson-Velsky and Landis Trees

An AVL tree is a type of binary search tree that is significant because it uses a height balancing property. To understand this, the height of the tree refers to the number of nodes from the root node to the farthest leaf (or longest path). This height balancing property is considered to be “the difference between the height of the left subtree and that of the right subtree of that node[16].” An AVL tree is considered balanced if it has a balance factor of either -1, 0, 1. In other words, the tree is only balanced if the height difference between all the nodes is 0 or 1. If the balance factor is ever affected after deletion, a process called rotation occurs.

[16]https://compsciedu.com/Data-Structures-and-Algorithms/Trees/discussion/14251

There are two forms of rotation, along with two subcategories of those forms. The first form is a single rotation. A single rotation reverses the roles and order of the child and the parent (node). In other words, the child becomes the parent and the parent becomes the child. One subcategory of this form is a Left Left Rotation, in which every node transfers to the left of its current position. Below is an example of Left Left Rotation:

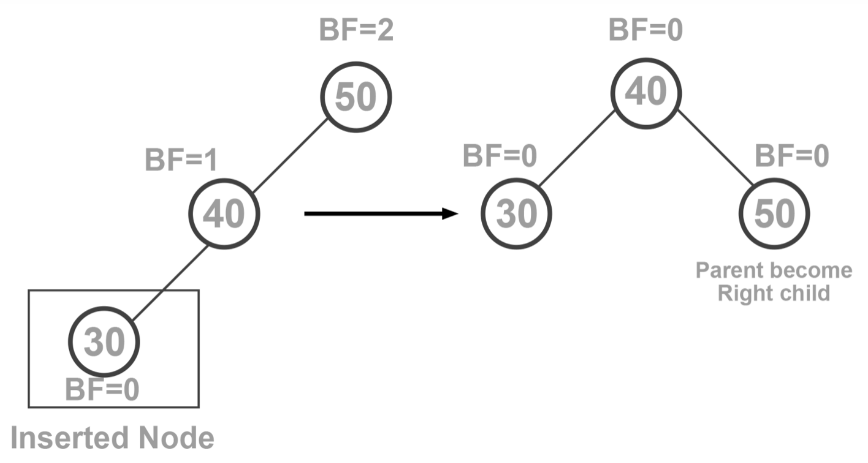


Figure 2.3.1

Now, the other subcategory of the single rotation is Right Right. This rotation follows a similar pattern to the previous subcategory, however every node transfers to the right of its current position instead:

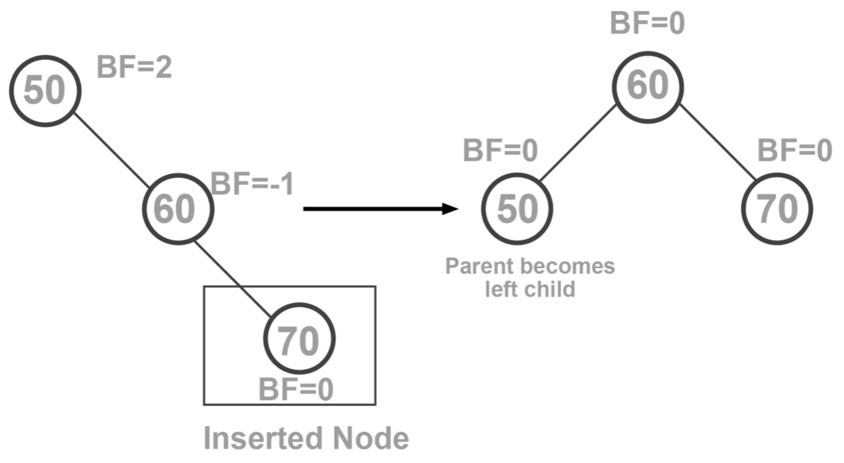


Figure 2.3.2

The other form of rotation during the deletion process for AVL trees is double rotation. Although the previous subcategories and single rotation fix Right Right and Left Left situations, it does not resolve Right Left or Left Right situations. Double rotation resolves this because it is the combination of two single rotations. The Left Right resolution begins with a single left rotation followed by a single right rotation. In this case, every node transfers left and then the new node attached to the new tree transfers again to the right. The same situation applies to the Right Left resolution, just in opposite order. Figure 2.3.3 represents a LR situation and Figure 2.3.4 represents a RL situation.

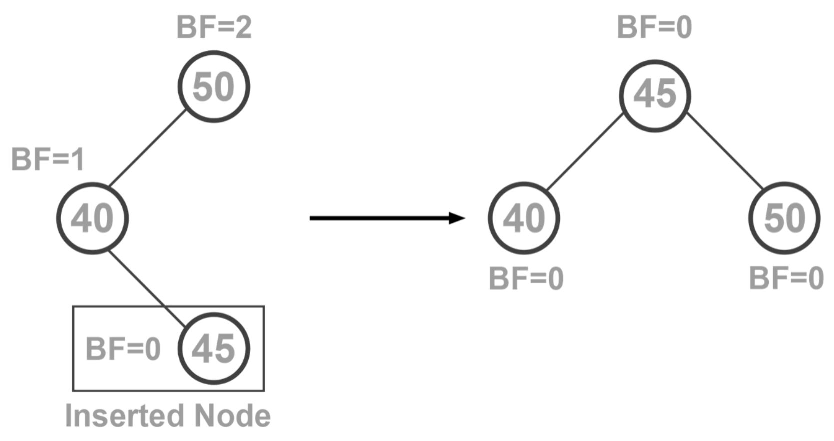


Figure 2.3.3 (LR)

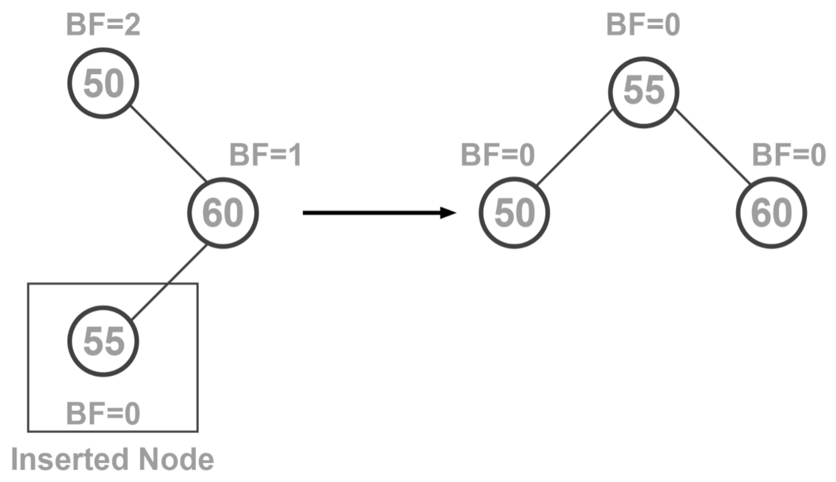


Figure 2.3.4 (RL)

Specifically for the deletion operation, there are algorithmic steps to ensure correct deletion:

1. Locate node to be deleted, delete if there is no child
2. If node has one child node, replace content of deletion node with the child node and continue to remove the node

3. With two child nodes, locate the inorder successor node with no child node and replace contents of the deletion node
4. Update balance factor of the tree

For example, Figure 2.3.5-2.3.7 will represent a typical deletion operation for AVL trees. The value "25" is to be deleted, and the tree is balanced in Figure 2.3.7.

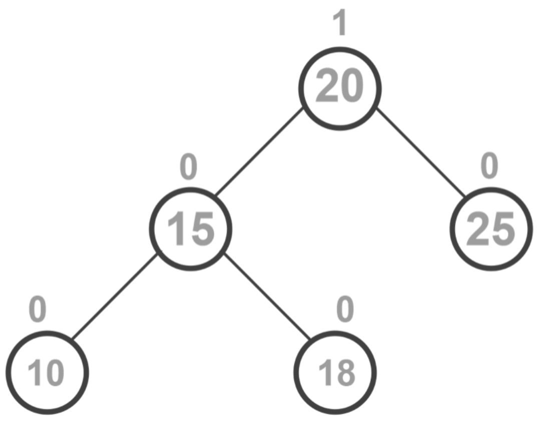


Figure 2.3.5 (Original Tree)

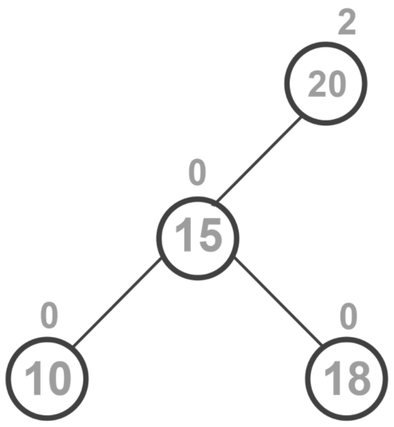


Figure 2.3.6 (Tree after deletion)

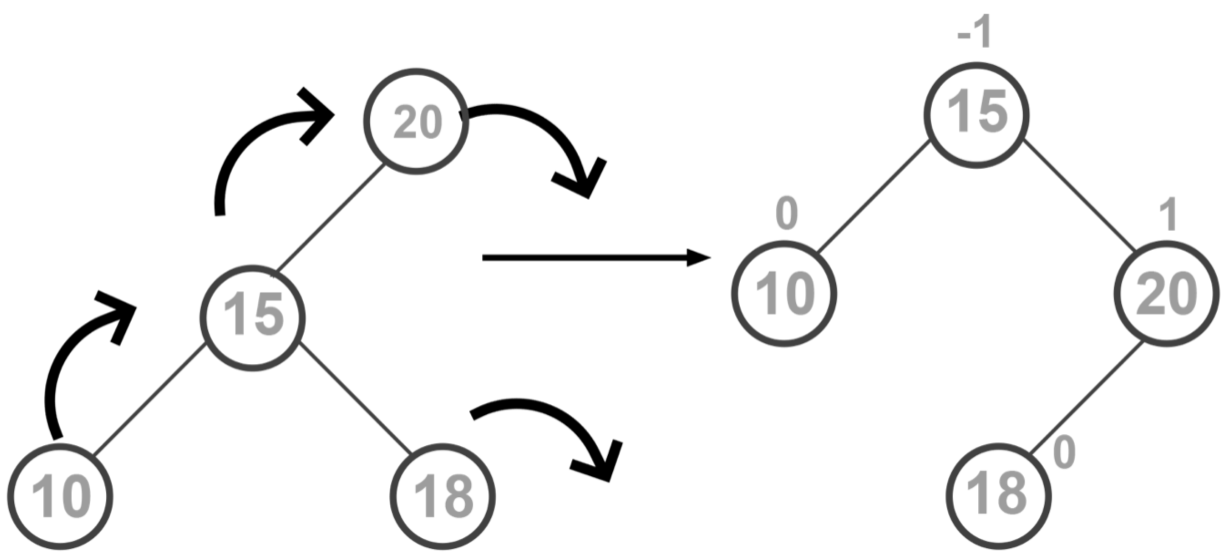


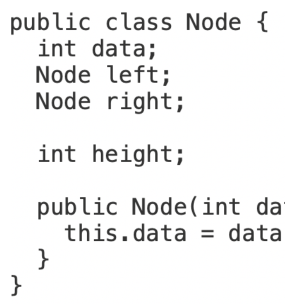
Figure 2.3.7 (Tree after balancing)

The AVL Tree algorithm can be explored further through the coded implementation referred to in ***Appendix B***, as well as pseudocode referred to in ***Appendix A***. Below is an analysis of a coded Java implementation of the deletion method (borrowed from the source in subscription[17]).

Similarly to before, to begin implementation for an AVL tree, there must be a reference to the original binary search tree. However, instead of carrying the statement “boolean color” to represent the colors red and black, AVL trees will instead carry an “int” statement of height, as AVL trees store the height of the subtree who holds the root node[18]:

---

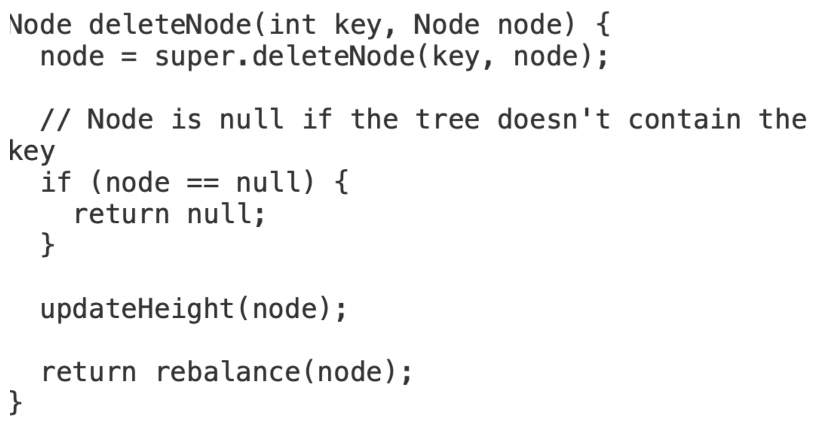
[17],[18] Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/BinarySearchTreeRecursive.java



```
public class Node {
  int data;
  Node left;
  Node right;

  int height;

  public Node(int data) {
    this.data = data;
  }
}
```

Figure 2.3.8

Below is the implementation for the deletion of a node[19].



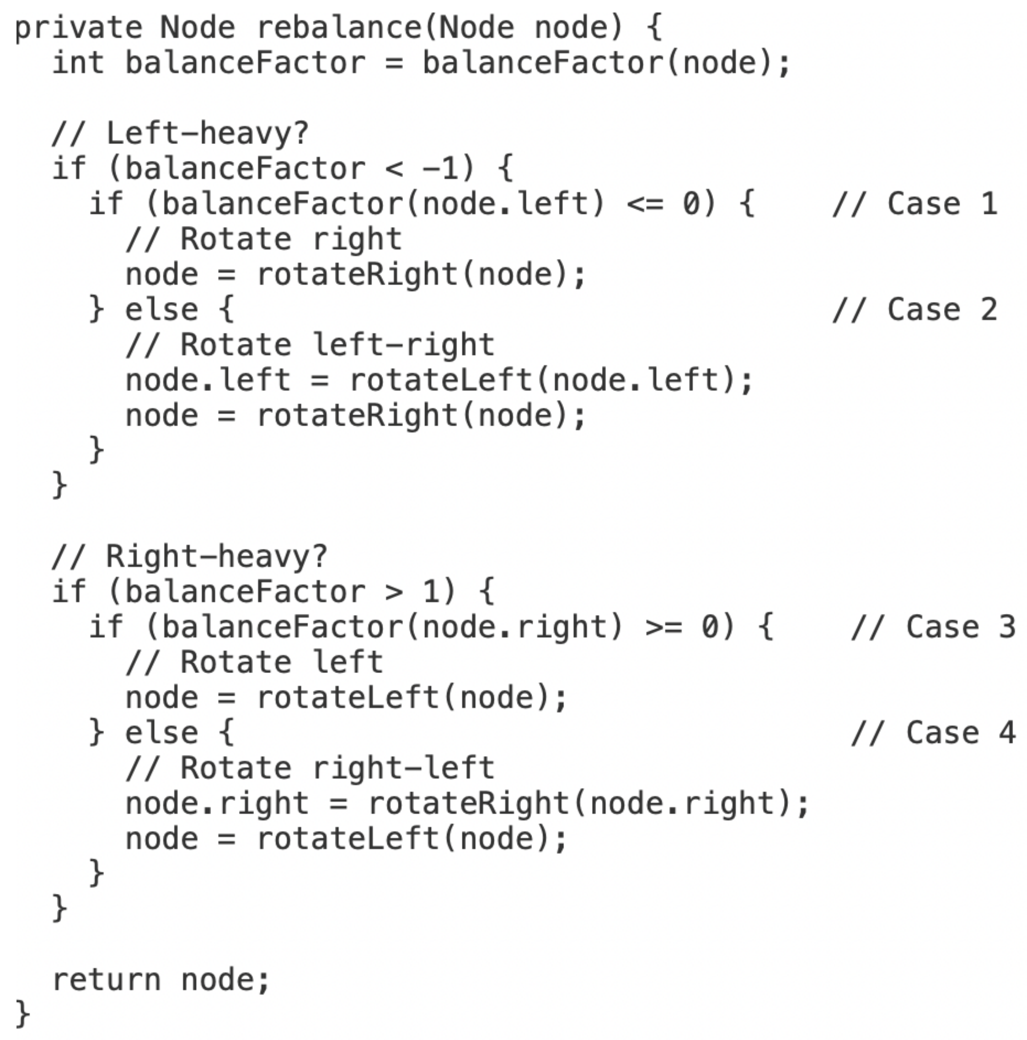
```
Node deleteNode(int key, Node node) {
  node = super.deleteNode(key, node);

  // Node is null if the tree doesn't contain the
key
  if (node == null) {
    return null;
  }

  updateHeight(node);

  return rebalance(node);
}
```

Figure 2.3.9

Refer to the full implementation of the super.deleteNode() method in ***Appendix B.***

[19] Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/BinarySearchTreeRecursive.java

As stated previously, there are four types of rotations. The Java implementation below and referenced in ***Appendix B*** showcase the rebalancing algorithm for AVL Trees under the four cases[20].



```java
private Node rebalance(Node node) {
  int balanceFactor = balanceFactor(node);

  // Left-heavy?
  if (balanceFactor < -1) {
    if (balanceFactor(node.left) <= 0) {    // Case 1
      // Rotate right
      node = rotateRight(node);
    } else {                                // Case 2
      // Rotate left-right
      node.left = rotateLeft(node.left);
      node = rotateRight(node);
    }
  }

  // Right-heavy?
  if (balanceFactor > 1) {
    if (balanceFactor(node.right) >= 0) {    // Case 3
      // Rotate left
      node = rotateLeft(node);
    } else {                                 // Case 4
      // Rotate right-left
      node.right = rotateRight(node.right);
      node = rotateLeft(node);
    }
  }

  return node;
}
```

Figure 2.3.10

[20] Woltmann, Sven. *GitHub*,

https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/BinarySearchTreeRecursive.java

3. Hypothesis and Applied Theory

Now that both types of binary search trees have been explored in depth in terms of the way they operate and how their deletion method is implemented through diagrams and code, it is essential to investigate which tree is more efficient in terms of time complexity. To continue with this investigation, an experiment will be conducted to measure the time it takes for each tree to delete certain values. It is concluded that both Red Black trees and AVL trees share the same time complexity of O(log n)[21] for their deletion operation. This value may be correct for the time complexity of the physical deletion operation itself, but it does not take into account the rebalancing mechanism afterwards. The experiment conducted for this investigation *will* take into account the rebalancing after the deletion operation.

The deletion method is considered to be easier and less complex in Red Black Trees compared to AVL Trees because of the fewer rotations it takes to complete the method. The Red Black Tree also does not contain a rebalancing factor, however it does store one bit of information that determines the color red or black for each node. It is also stated that Red Black Trees are not as strictly balanced compared to AVL Trees[22], and Red Black Trees tend to rebalance less than AVL Trees as well.. With that being said, **I hypothesize that the Red Black Trees' deletion operation will be more algorithmically efficient in terms of time complexity compared to AVL trees' deletion operation.**

[21]"Red Black Tree vs AVL Tree - Javatpoint." *Www.javatpoint.com*, https://www.javatpoint.com/red-black-tree-vs-avl-tree

[22]"Red Black Tree vs AVL Tree - Javatpoint." *Www.javatpoint.com*, https://www.javatpoint.com/red-black-tree-vs-avl-tree

The experiment will also measure **the size of the sets being deleted (x) and the total time it takes for completion (y)**. As the size of the sets are altered and vary throughout the experiment, a clear explanation and comparison can be made between the two trees. **I also hypothesize that there will be a logarithmic relationship between x and y.**

4. Methodology, Variables, and Procedure

This section will go in depth with the procedure of the experiment along with the Java code being referenced.

4.1 Independent Variables

As stated prior, the size of the sets will be changed throughout the experiment to test out the total operation time. The size of the sets is the independent variable in the experiment. All the sets of data will consist of successive integers from 1 to *N*. *N* will begin at the value of 100 and end at 1000, increasing in separate increments of 100; hence, there will be ten different sets of values. These values were chosen mainly to ensure that there can be a precise and clear graphical representation of the data's relationship. The increasing order of values for *N* was mainly chosen to maximize the total amount of time for the deletion operation of both trees.

4.2 Dependent Variables

Also stated previously, the dependent variable of this experiment that relies on the independent variable of the size of the sets is the total time, or the y variable. The time will be measured with the *Java.lang.System.nanoTime()* method which will deliver a total time in nanoseconds. With the method and system being used, nanoseconds are the most precise measurement possible for this experiment.

4.3 Controlled Variables

Below is a table of the controlled variables of this experiment.

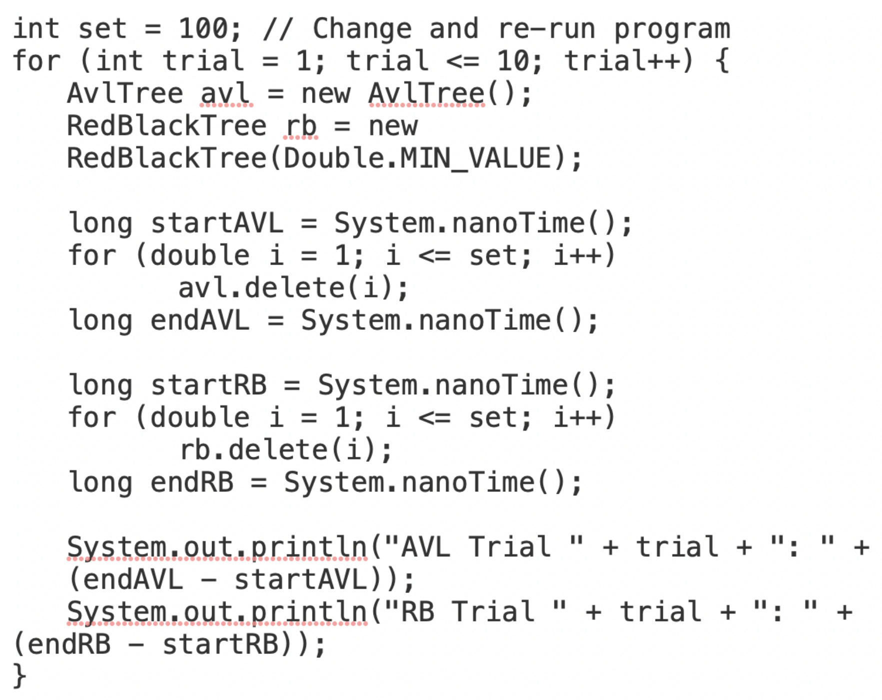
Variable	Description	Specifications
Laptop being used	This experiment will be operated on my Macbook Pro	**Version:** 11.6 **Processor:** 2.3 GhZ Dual-Core Intel Core I5 **Memory:** 8 GB 2133 MHz LPDDR3
IDE	This experiment and code will be ran on an IDE on my computer	**IDE:** Eclipse
Algorithm	The algorithm will be the same Java implementation from ***Appendix B***	
Function	The functions called for each set will be the same	
Data type	The ***int*** data type will be used for all sets	

4.4 Procedure

The procedure is as follows:

1. Begin and run the program, insert all necessary values, and time each deletion operation. The time's output will be measured in nanoseconds; record this in a separate text file.
2. Continue to run the program so that it can present the outputted time values
3. Record the average of the times for each set of each binary search tree

Below is an implementation of the Java program being used to test the sets of values. This is also referred to in ***Appendix B***.



```
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.delete(i);
    long endAVL = System.nanoTime();

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

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

Figure 4.4.1

5. Data Processing and Graphical Representation

5.1 Data collection

Below is a table showing the average times collected for the deletion operation from the program.

	Average time (nanoseconds)	Average time (nanoseconds)
Set Size	**RB Trees**	**AVL Trees**
100	538,490	409,284
200	831,394	523,859
300	1,082,012	841,683
400	1,092,389	1,183,671
500	1,273,840	1,764,894
600	1,362,893	1,957,368
700	1,730,230	2,253,647
800	1,745,837	2,517,389
900	1,834,283	2,583,918
1000	2,578,239	2,617,390

Figure 5.1.1 (Table of average deletion times)

5.2 Graphical Representation (Time vs size)

Below is the graphical representation of this experiment's data of time vs the size of the sets deleted.

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

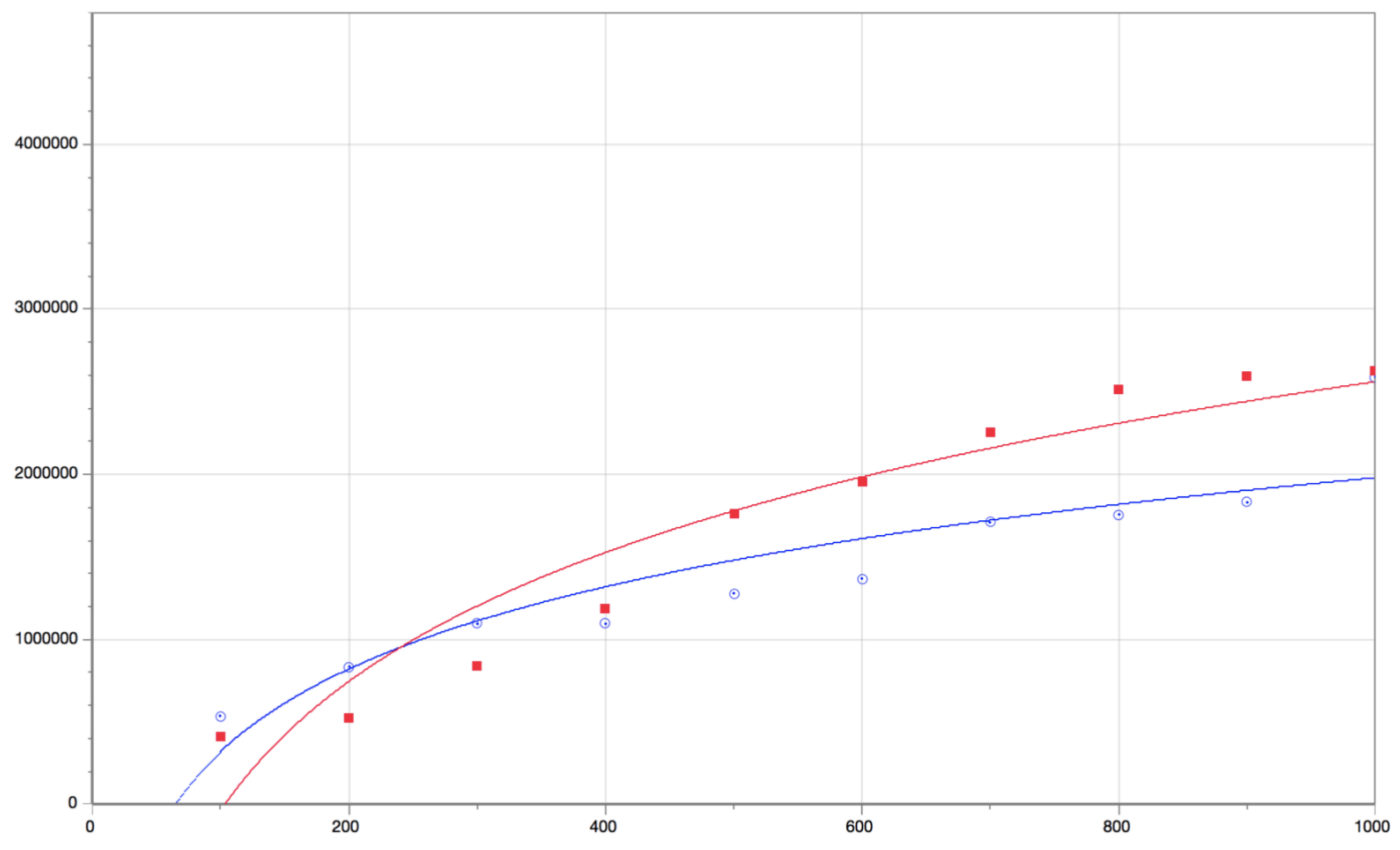


Figure 5.2 (y axis is deletion time in nanoseconds, x axis is the number of values in the set)

6. Results

According to the graphical representation of the data collected from the experiment, there seems to be a logarithmic relationship between the amount of time it takes for the deletion process in nanoseconds and the number of values in the set. This proves my hypothesis on the relationship being logarithmic correct. Part of my hypothesis can be incorrect though, as red black trees were not ALWAYS more efficient than AVL trees for all value sets in terms of time complexity. According to the graphical representation or Figure 5.2, there seems to be an intersection between the two lines at approximately 250 values. Because of this, it can be concluded that the Red Black Tree is faster and more efficient in terms of time complexity *only* if the amount of values in the set is under 250. Encountering this number and discovery in general sparked my interest vastly, and I wanted to further my exploration and understanding of what the statistic meant.

One possibility can be the fact that although Red Black trees are initially faster than AVL trees, they tend to become unbalanced over time. This would eventually lead to slower rebalancing times in comparison to AVL trees.

I discovered that the maximum heights of both Red Black Trees and AVL Trees are as followed:

$1.44 \log_2(N+2)$ for AVL Trees[23]

$2\log_2(N+1)$ for Red Black Trees[24]

These plotted on a graph:

[23]"Practice Questions on Height Balanced/Avl Tree." *GeeksforGeeks*, 7 Feb. 2018, https://www.geeksforgeeks.org/practice-questions-height-balancedavl-tree/

[24]Steyn, Barry. "Maximum Height of a Red-Black Tree." *Doctrina*, https://doctrina.org/maximum-height-of-red-black-tree.html

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

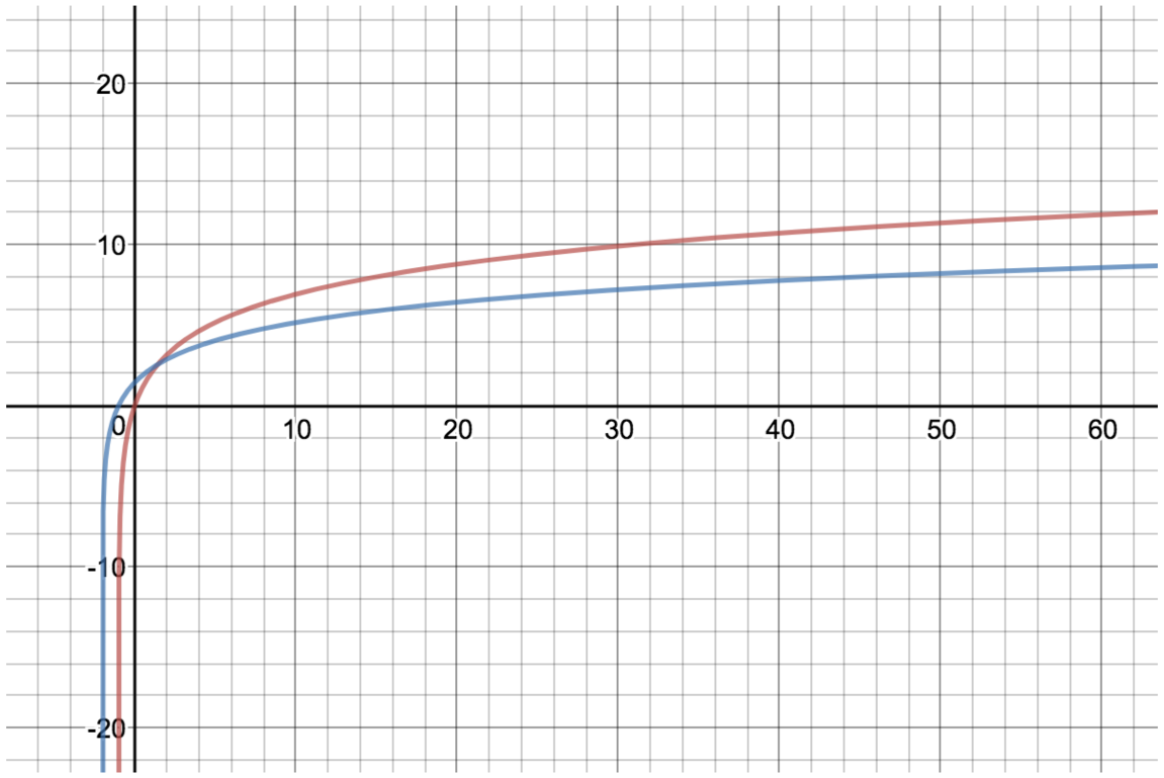


Figure 6.1

The Red graph represents a Red Black Tree and the Blue graph represents an AVL tree. The graph in Figure 6.1 is nearly identical to the original graph plotted by the points of collected data.

7. Conclusion

This essay and experiment as a whole intended to find the extent to which Red Black Trees are more efficient during the deletion process in terms of time complexity in comparison to AVL Trees. With the functions and application of the theories of both binary search trees, this experiment was able to discover a logarithmic relationship between the total time it takes for the deletion process to complete (including the rebalancing algorithm) and the size of sets of values.

For required restructuring and repair, the experiment used ordered sets of values. Since Red Black Trees had greater heights than AVL Trees on the root node’s right side, it was more and more frequent that Red Black Trees would increase in height as well. However, Red Black Trees typically were not as sensitive and caring as AVL Trees were in terms of rebalancing. In this case, AVL Trees are more efficient in terms of time complexity during the deletion process with values over 250, while Red Black Trees are more efficient in terms of time complexity during the deletion process with values under 250.

Overall, the efficiency and comparison between the two binary search trees largely are dependent on the amount of values in the ordered sets. Time complexity can also vary depending on whether the rebalancing algorithm is considered. In a shorter run with lower values, it may be more efficient to use Red Black Trees. However, with higher values, it’s more efficient to use AVL Trees as a sorting algorithm.

8. Appendices

Appendix A: Pseudocode

Deletion Method for Binary Search Trees:

```
Node deleteNode(Node root, int valueToDelete) {
  if root = null
    return node
  if root.value < valueToDelete
    deleteNode(root.right, valueToDelete)
  if root.value > valueToDelete
    deleteNode(root.left, valueToDelete)
  else
    if (isLeafNode(root))
      return null

    if (root.right == null)
      return root.left
    if (root.left == null)
      return root.right
    else
      minValue = findMinInRightSubtree(root)
      root.value = minValue
      removeDuplicateNode(root)
      return root
```

Deletion Method for Red Black Trees:

```
RB-DELETE(T, z)
    if z->left = null or z->right = null
         then y ← z
     else y ← TREE-SUCCESSOR(z)
     if y->left ≠ null
         then x ← y->left
     else x ← y->right
     x->p ← y->p
     if y->p = null
         then T->root ← x
     else if y = y->p->left
         then y->p->left ← x
```

```
   else y->p->right ← x
   if y 3≠ z
       then z->key ← y->key
   copy y's satellite data into z
   if y->color = BLACK
       then RB-DELETE-FIXUP(T, x)
   return y
RB-DELETE-FIXUP(T, x)
  while x ≠ T->root and x->color = BLACK
       do if x = x->p->left
           then w ← x->p->right
       if w->color = RED
           then w->color ← BLACK Case 1
       x->p->color ← RED Case 1
       LEFT-ROTATE(T, x->p) Case 1
       w ← x->p->right Case 1
       if w->left->color = BLACK and w->right->color = BLACK
           then w->color ← RED Case 2
       x ← x->p Case 2
       else if w->right->color = BLACK
```

```
        then w->left->color ← BLACK Case 3
    w->color ← RED Case 3
    RIGHT-ROTATE(T, w) Case 3
    w ← x->p->right Case 3
    w->color ← x->p->color Case 4
    x->p->color ← BLACK Case 4
    w->right->color ← BLACK Case 4
    LEFT-ROTATE(T, x->p) Case 4
    x ← T->root Case 4
    else (same as then clause with "right" and "left"
exchanged)
        x->color ← BLACK
```

Deletion Method for AVL Trees:

```
Algorithm removeAVL (k,T):
     Input: A key, k, and an AVL tree, T
     Output: An update of T to now have an item (k, e) removed
     v ←  IterativeTreeSearch(k,T)
```

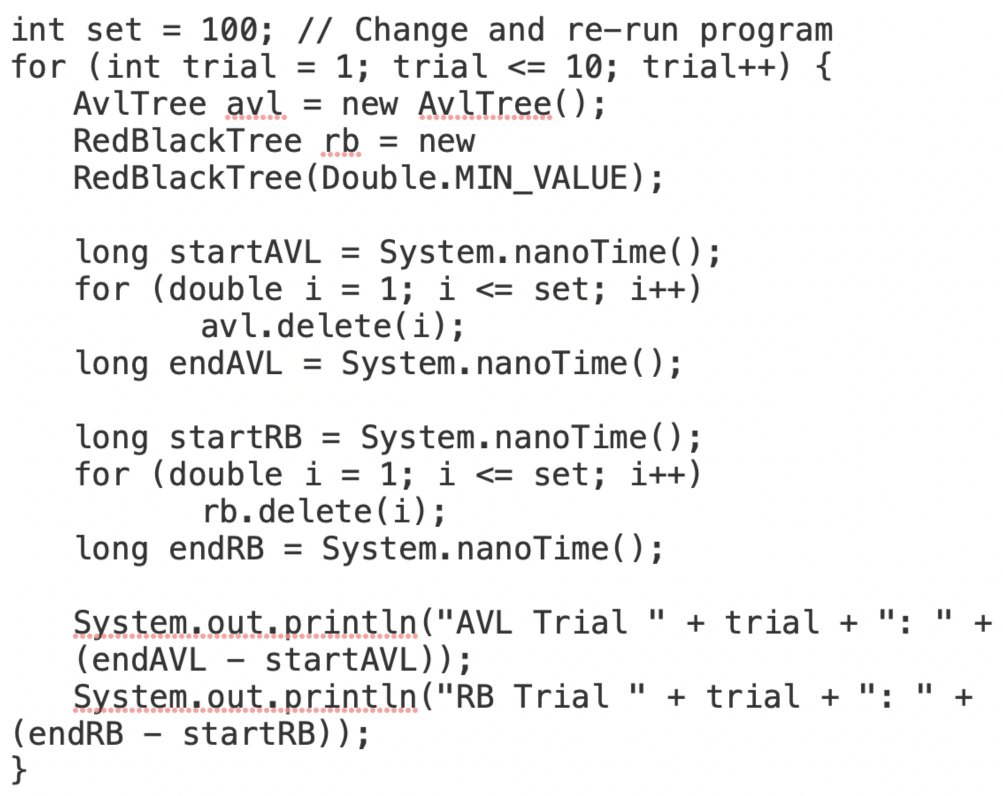
```
    if v is an external node then
         Return "There is no item with key k in T"
    if v has no external-node child then
         Let u be the node in T with key nearest to k
         Move u's key-value pair to v
         v ← u
    Let w be v's smallest height child
    Remove w and v from T, replacing v with w's sibling, z
    rebalanceAVL(z, T)
```

Appendix B: Java Program Used

Java implementation for binary search tree:

```
public class Node {
  int data;
  Node left;

  Node right;

  Node parent;



  boolean color;



  public Node(int data) {

    this.data = data;

  }

}
```

Java implementation of program used for 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.delete(i);
   long endAVL = System.nanoTime();

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

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

Java implementation of Red Black Tree deletion and repair/rebalancing:

```
package eu.happycoders.binarytree;

/**
 * A red-black tree implementation with <code>int</code> keys.
 *
 * @author <a href="sven@happycoders.eu">Sven Woltmann</a>
 */
public class RedBlackTree extends BaseBinaryTree implements BinarySearchTree {

  static final boolean RED = false;
  static final boolean BLACK = true;

  @Override
  public Node searchNode(int key) {
    Node node = root;
    while (node != null) {
      if (key == node.data) {
        return node;
      } else if (key < node.data) {
        node = node.left;
      } else {
        node = node.right;
      }
    }

    return null;
  }

  // -- Insertion
----------------------------------------------------------------------------------

  @Override
  public void insertNode(int key) {
    Node node = root;
    Node parent = null;

    // Traverse the tree to the left or right depending on the key
    while (node != null) {
      parent = node;
      if (key < node.data) {
        node = node.left;
      } else if (key > node.data) {
        node = node.right;
      } else {
        throw new IllegalArgumentException("BST already contains a node with key " +
key);
      }
    }

    // Insert new node
    Node newNode = new Node(key);
    newNode.color = RED;
    if (parent == null) {
      root = newNode;
    } else if (key < parent.data) {
      parent.left = newNode;
    } else {
      parent.right = newNode;
    }
    newNode.parent = parent;

    fixRedBlackPropertiesAfterInsert(newNode);
  }

  @SuppressWarnings("squid:S125") // Ignore SonarCloud complains about commented code
line 70.
  private void fixRedBlackPropertiesAfterInsert(Node node) {
    Node parent = node.parent;

    // Case 1: Parent is null, we've reached the root, the end of the recursion
    if (parent == null) {
      // Uncomment the following line if you want to enforce black roots (rule 2):
      // node.color = BLACK;
      return;
    }

    // Parent is black --> nothing to do
    if (parent.color == BLACK) {
      return;
    }

    // From here on, parent is red
    Node grandparent = parent.parent;

    // Case 2:
    // Not having a grandparent means that parent is the root. If we enforce black
roots
    // (rule 2), grandparent will never be null, and the following if-then block can
be
    // removed.
    if (grandparent == null) {
      // As this method is only called on red nodes (either on newly inserted ones -
or -
      // recursively on red grandparents), all we have to do is to recolor the root
black.
      parent.color = BLACK;
      return;
    }

    // Get the uncle (may be null/nil, in which case its color is BLACK)
    Node uncle = getUncle(parent);

    // Case 3: Uncle is red -> recolor parent, grandparent and uncle
    if (uncle != null && uncle.color == RED) {
      parent.color = BLACK;
      grandparent.color = RED;
      uncle.color = BLACK;

      // Call recursively for grandparent, which is now red.
      // It might be root or have a red parent, in which case we need to fix more...
      fixRedBlackPropertiesAfterInsert(grandparent);
    }

    // Note on performance:
    // It would be faster to do the uncle color check within the following code. This
way
    // we would avoid checking the grandparent-parent direction twice (once in
getUncle()
    // and once in the following else-if). But for better understanding of the code,
    // I left the uncle color check as a separate step.

    // Parent is left child of grandparent
    else if (parent == grandparent.left) {
      // Case 4a: Uncle is black and node is left->right "inner child" of its
grandparent
      if (node == parent.right) {
        rotateLeft(parent);
        // Let "parent" point to the new root node of the rotated sub-tree.
        // It will be recolored in the next step, which we're going to fall-through
to.
        parent = node;
      }

      // Case 5a: Uncle is black and node is left->left "outer child" of its
grandparent
      rotateRight(grandparent);

      // Recolor original parent and grandparent
      parent.color = BLACK;
      grandparent.color = RED;
    }

    // Parent is right child of grandparent
    else {
      // Case 4b: Uncle is black and node is right->left "inner child" of its
grandparent
      if (node == parent.left) {
        rotateRight(parent);

        // Let "parent" point to the new root node of the rotated sub-tree.
        // It will be recolored in the next step, which we're going to fall-through
to.
        parent = node;
      }

      // Case 5b: Uncle is black and node is right->right "outer child" of its
grandparent
      rotateLeft(grandparent);

      // Recolor original parent and grandparent
      parent.color = BLACK;
      grandparent.color = RED;
    }
  }

  private Node getUncle(Node parent) {
    Node grandparent = parent.parent;
    if (grandparent.left == parent) {
      return grandparent.right;
    } else if (grandparent.right == parent) {
      return grandparent.left;
    } else {
      throw new IllegalStateException("Parent is not a child of its grandparent");
    }
  }

  // -- Deletion
-----------------------------------------------------------------------------------

  @SuppressWarnings("squid:S2259") // SonarCloud issues an incorrect potential NPE
warning
  @Override
  public void deleteNode(int key) {
    Node node = root;

    // Find the node to be deleted
    while (node != null && node.data != key) {
      // Traverse the tree to the left or right depending on the key
      if (key < node.data) {
        node = node.left;
      } else {
        node = node.right;
      }
    }

    // Node not found?
    if (node == null) {
      return;
    }

    // At this point, "node" is the node to be deleted

    // In this variable, we'll store the node at which we're going to start to fix the
R-B
    // properties after deleting a node.
    Node movedUpNode;
    boolean deletedNodeColor;

    // Node has zero or one child
    if (node.left == null || node.right == null) {
      movedUpNode = deleteNodeWithZeroOrOneChild(node);
      deletedNodeColor = node.color;
    }

    // Node has two children
    else {
      // Find minimum node of right subtree ("inorder successor" of current node)
      Node inOrderSuccessor = findMinimum(node.right);

      // Copy inorder successor's data to current node (keep its color!)
      node.data = inOrderSuccessor.data;

      // Delete inorder successor just as we would delete a node with 0 or 1 child
      movedUpNode = deleteNodeWithZeroOrOneChild(inOrderSuccessor);
      deletedNodeColor = inOrderSuccessor.color;
    }

    if (deletedNodeColor == BLACK) {
      fixRedBlackPropertiesAfterDelete(movedUpNode);

      // Remove the temporary NIL node
      if (movedUpNode.getClass() == NilNode.class) {
        replaceParentsChild(movedUpNode.parent, movedUpNode, null);
      }
    }
  }
  private Node deleteNodeWithZeroOrOneChild(Node node) {
    // Node has ONLY a left child --> replace by its left child
    if (node.left != null) {
      replaceParentsChild(node.parent, node, node.left);
      return node.left; // moved-up node
    }

    // Node has ONLY a right child --> replace by its right child
    else if (node.right != null) {
      replaceParentsChild(node.parent, node, node.right);
      return node.right; // moved-up node
    }

    // Node has no children -->
    // * node is red --> just remove it
    // * node is black --> replace it by a temporary NIL node (needed to fix the R-B
rules)
    else {
      Node newChild = node.color == BLACK ? new NilNode() : null;
      replaceParentsChild(node.parent, node, newChild);
      return newChild;
    }
  }

  private Node findMinimum(Node node) {
    while (node.left != null) {
      node = node.left;
    }
    return node;
  }

  @SuppressWarnings("squid:S125") // Ignore SonarCloud complains about commented code
line 256.
  private void fixRedBlackPropertiesAfterDelete(Node node) {
    // Case 1: Examined node is root, end of recursion
    if (node == root) {
      // Uncomment the following line if you want to enforce black roots (rule 2):
      // node.color = BLACK;
      return;
    }

    Node sibling = getSibling(node);

    // Case 2: Red sibling
    if (sibling.color == RED) {
      handleRedSibling(node, sibling);
      sibling = getSibling(node); // Get new sibling for fall-through to cases 3-6
    }

    // Cases 3+4: Black sibling with two black children
    if (isBlack(sibling.left) && isBlack(sibling.right)) {
      sibling.color = RED;

      // Case 3: Black sibling with two black children + red parent
      if (node.parent.color == RED) {
```

```
      node.parent.color = BLACK;
    }

    // Case 4: Black sibling with two black children + black parent
    else {
      fixRedBlackPropertiesAfterDelete(node.parent);
    }
  }

  // Case 5+6: Black sibling with at least one red child
  else {
    handleBlackSiblingWithAtLeastOneRedChild(node, sibling);
  }
}

private void handleRedSibling(Node node, Node sibling) {
  // Recolor...
  sibling.color = BLACK;
  node.parent.color = RED;

  // ... and rotate
  if (node == node.parent.left) {
    rotateLeft(node.parent);
  } else {
    rotateRight(node.parent);
  }
}

private void handleBlackSiblingWithAtLeastOneRedChild(Node node, Node sibling) {
  boolean nodeIsLeftChild = node == node.parent.left;

  // Case 5: Black sibling with at least one red child + "outer nephew" is black
  // --> Recolor sibling and its child, and rotate around sibling
  if (nodeIsLeftChild && isBlack(sibling.right)) {
    sibling.left.color = BLACK;
    sibling.color = RED;
    rotateRight(sibling);
    sibling = node.parent.right;
  } else if (!nodeIsLeftChild && isBlack(sibling.left)) {
    sibling.right.color = BLACK;
    sibling.color = RED;
    rotateLeft(sibling);
    sibling = node.parent.left;
  }

  // Fall-through to case 6...

  // Case 6: Black sibling with at least one red child + "outer nephew" is red
  // --> Recolor sibling + parent + sibling's child, and rotate around parent
  sibling.color = node.parent.color;
  node.parent.color = BLACK;
  if (nodeIsLeftChild) {
    sibling.right.color = BLACK;
    rotateLeft(node.parent);
  } else {
      sibling.left.color = BLACK;
      rotateRight(node.parent);
    }
  }

  private Node getSibling(Node node) {
    Node parent = node.parent;
    if (node == parent.left) {
      return parent.right;
    } else if (node == parent.right) {
      return parent.left;
    } else {
      throw new IllegalStateException("Parent is not a child of its grandparent");
    }
  }

  private boolean isBlack(Node node) {
    return node == null || node.color == BLACK;
  }

  private static class NilNode extends Node {
    private NilNode() {
      super(0);
      this.color = BLACK;
    }
  }

  // -- Helpers for insertion and deletion
--------------------------------------------------------

  private void rotateRight(Node node) {
    Node parent = node.parent;
    Node leftChild = node.left;

    node.left = leftChild.right;
    if (leftChild.right != null) {
      leftChild.right.parent = node;
    }

    leftChild.right = node;
    node.parent = leftChild;

    replaceParentsChild(parent, node, leftChild);
  }

  private void rotateLeft(Node node) {
    Node parent = node.parent;
    Node rightChild = node.right;

    node.right = rightChild.left;
    if (rightChild.left != null) {
      rightChild.left.parent = node;
    }

    rightChild.left = node;
    node.parent = rightChild;

    replaceParentsChild(parent, node, rightChild);
  }

  private void replaceParentsChild(Node parent, Node oldChild, Node newChild) {
    if (parent == null) {
      root = newChild;
    } else if (parent.left == oldChild) {
      parent.left = newChild;
    } else if (parent.right == oldChild) {
      parent.right = newChild;
    } else {
      throw new IllegalStateException("Node is not a child of its parent");
    }

    if (newChild != null) {
      newChild.parent = parent;
    }
  }

  // -- For toString()
-----------------------------------------------------------------------------

  @Override
  protected void appendNodeToString(Node node, StringBuilder builder) {
    builder.append(node.data).append(node.color == RED ? "[R]" : "[B]");
  }
}
```

Java implementation for AVL Tree deletion

```
package eu.happycoders.binarytree;

/**
 * A recursive binary search tree implementation with <code>int</code> keys.
 *
 * @author <a href="sven@happycoders.eu">Sven Woltmann</a>
 */
public class BinarySearchTreeRecursive extends BaseBinaryTree implements
BinarySearchTree {

  @Override
  public Node searchNode(int key) {
    return searchNode(key, root);
  }

  private Node searchNode(int key, Node node) {
    if (node == null) {
      return null;
    }
    if (key == node.data) {
      return node;
    } else if (key < node.data) {
      return searchNode(key, node.left);
    } else {
      return searchNode(key, node.right);
    }
  }

  @Override
  public void insertNode(int key) {
    root = insertNode(key, root);
  }

  Node insertNode(int key, Node node) {
    // No node at current position --> store new node at current position
    if (node == null) {
      node = new Node(key);
    }

    // Otherwise, traverse the tree to the left or right depending on the key
    else if (key < node.data) {
      node.left = insertNode(key, node.left);
    } else if (key > node.data) {
      node.right = insertNode(key, node.right);
    } else {
      throw new IllegalArgumentException("BST already contains a node with key " +
key);
    }

    return node;
  }

  @Override
  public void deleteNode(int key) {
    root = deleteNode(key, root);
  }

  Node deleteNode(int key, Node node) {
    // No node at current position --> go up the recursion
    if (node == null) {
      return null;
    }

    // Traverse the tree to the left or right depending on the key
    if (key < node.data) {
      node.left = deleteNode(key, node.left);
    } else if (key > node.data) {
      node.right = deleteNode(key, node.right);
    }

    // At this point, "node" is the node to be deleted

    // Node has no children --> just delete it
    else if (node.left == null && node.right == null) {
      node = null;
    }

    // Node has only one child --> replace node by its single child
    else if (node.left == null) {
      node = node.right;
    } else if (node.right == null) {
      node = node.left;
    }

    // Node has two children
    else {
      deleteNodeWithTwoChildren(node);
    }

    return node;
  }

  private void deleteNodeWithTwoChildren(Node node) {
    // Find minimum node of right subtree ("inorder successor" of current node)
    Node inOrderSuccessor = findMinimum(node.right);

    // Copy inorder successor's data to current node
    node.data = inOrderSuccessor.data;

    // Delete inorder successor recursively
    node.right = deleteNode(inOrderSuccessor.data, node.right);
  }

  private Node findMinimum(Node node) {
    while (node.left != null) {
      node = node.left;
    }
    return node;
  }
}
```

Java implementation for AVL Tree rebalancing algorithm:

```
package eu.happycoders.binarytree;

import static java.lang.Math.max;

/**
 * An AVL tree implementation with <code>int</code> keys.
 *
 * @author <a href="sven@happycoders.eu">Sven Woltmann</a>
 */
public class AvlTree extends BinarySearchTreeRecursive {

  @Override
  Node insertNode(int key, Node node) {
    node = super.insertNode(key, node);
    updateHeight(node);

    return rebalance(node);
  }

  @Override
  Node deleteNode(int key, Node node) {
    node = super.deleteNode(key, node);

    // Node is null if the tree doesn't contain the key
    if (node == null) {
      return null;
    }

    updateHeight(node);

    return rebalance(node);
  }

  private void updateHeight(Node node) {
    int leftChildHeight = height(node.left);
    int rightChildHeight = height(node.right);
    node.height = max(leftChildHeight, rightChildHeight) + 1;
  }

  private Node rebalance(Node node) {
    int balanceFactor = balanceFactor(node);

    // Left-heavy?
    if (balanceFactor < -1) {
      if (balanceFactor(node.left) <= 0) {
        // Rotate right
        node = rotateRight(node);
      } else {
        // Rotate left-right
        node.left = rotateLeft(node.left);
        node = rotateRight(node);
      }
    }

    // Right-heavy?
    if (balanceFactor > 1) {
      if (balanceFactor(node.right) >= 0) {
        // Rotate left
        node = rotateLeft(node);
      } else {
        // Rotate right-left
        node.right = rotateRight(node.right);
        node = rotateLeft(node);
      }
    }

    return node;
  }
```

```
  private Node rotateRight(Node node) {
    Node leftChild = node.left;

    node.left = leftChild.right;
    leftChild.right = node;

    updateHeight(node);
    updateHeight(leftChild);

    return leftChild;
  }

  private Node rotateLeft(Node node) {
    Node rightChild = node.right;

    node.right = rightChild.left;
    rightChild.left = node;

    updateHeight(node);
    updateHeight(rightChild);

    return rightChild;
  }

  private int balanceFactor(Node node) {
    return height(node.right) - height(node.left);
  }

  private int height(Node node) {
    return node != null ? node.height : -1;
  }

  @Override
  protected void appendNodeToString(Node node, StringBuilder builder) {
    builder
        .append(node.data)
        .append("[H=")
        .append(height(node))
        .append(", BF=")
        .append(balanceFactor(node))
        .append(']');
  }
}
```

Bibliography (MLA 9)

AVL Trees - University of California, Irvine. https://www.ics.uci.edu/~goodrich/teach/cs260P/notes/AVLTrees.pdf

“Binary Meaning.” *Binary Meaning | Best 27 Definitions of Binary*, https://www.yourdictionary.com/binary

“Binary Search Tree: Set 2 (Delete).” *GeeksforGeeks*, 1 Feb. 2022, https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

“Delete Node from Binary Search TreeEdit Pagepage History.” *Delete Node From Binary Search Tree | CodePath Android Cliffnotes*, https://guides.codepath.com/compsci/Delete-Node-From-Binary-Search-Tree

“Deletion in Binary Search Tree - Javatpoint.” *Www.javatpoint.com*, https://www.javatpoint.com/deletion-in-binary-search-tree

Gupta, Vipul. “Red Black Tree: Deletion.” *OpenGenus IQ: Computing Expertise & Legacy*, OpenGenus IQ: Computing Expertise & Legacy, 17 Apr. 2019, https://iq.opengenus.org/red-black-tree-deletion/

https://compsciedu.com/Data-Structures-and-Algorithms/Trees/discussion/14251

“Inorder Successor in Binary Search Tree.” *GeeksforGeeks*, 10 Jan. 2022, https://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/

“Inorder Successor in Binary Search Tree - Tutorialspoint.dev.” *Tutorialspoint.Dev*, https://tutorialspoint.dev/data-structure/binary-search-tree/inorder-successor-in-binary-search-tree

Great Learning Team, et al. “Why Is Time Complexity Essential and What Is Time Complexity?” *GreatLearning Blog: Free Resources What Matters to Shape Your Career!*, 8 Jan. 2022, https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/

“Practice Questions on Height Balanced/Avl Tree.” *GeeksforGeeks*, 7 Feb. 2018, https://www.geeksforgeeks.org/practice-questions-height-balancedavl-tree/

Pseudocode for AVL Balanced Binary ... - Swarthmore College. https://www.cs.swarthmore.edu/~brody/cs35/f14/Labs/extras/08/avl_pseudo.pdf

“Red Black Tree vs AVL Tree - Javatpoint.” *Www.javatpoint.com*, https://www.javatpoint.com/red-black-tree-vs-avl-tree

"Red-Black Tree: Set 1 (Introduction)." *GeeksforGeeks*, 18 Dec. 2021, https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

Steyn, Barry. "Maximum Height of a Red-Black Tree." *Doctrina*, https://doctrina.org/maximum-height-of-red-black-tree.html

Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/AvlTree.java

Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/BinarySearchTreeRecursive.java

Woltmann, Sven. *GitHub*, https://raw.githubusercontent.com/SvenWoltmann/binary-tree/main/src/main/java/eu/happycoders/binarytree/RedBlackTree.java