Comp 251 Proofs Assignment Report

260971542 No Collaborators

Comp251, Winter 2023

Claim 1

Deleting a node x from an AVL tree takes $O(log(n))$ time.

Proof

This proof is adapted from Demaine, Indyk, Kellis, *Introduction to Algorithms: AVL Trees* [2].

Proof. There are 3 components of deleting a node x from an AVL Tree:

1. Searching for the node in the AVL Tree:
   For this step of the procedure, we can assume that the traversing of the AVL tree to get to node x will be of the time complexity $O(log(n))$, as it is one of the other complexity proofs.

2. Deleting the node in the AVL Tree:
   For this step of the procedure, there are three options when deleting a node x. If x does not have children, it can removed without other steps. If x has one child node y, y replaces the position of x when it gets deleted. If x has two child nodes, y on its left and z on its right, the left-most node, in this case, y, will replace the x node, now having z attached to y. Since this is a simple replacement of a node, once the position has been obtained, is $O(1)$.

3. Re-balancing the AVL Tree:
   After deletion, we need to re-balance the height of each node. Let height h represent the current height of node x. h would be updated of node x, which would update the height of node x's parent, until there are no parents left, reaching the top of the tree. The time complexity is $O(h)$, which becomes $O(1)$ as h is an integer value. The next step would be to replace the tree using a Left Rotation (LL), Right Rotation (RR), Left-Right Rotation (LR), or Right-Left Rotation (RL). For a left rotation, if we have a node a who has a node b on its right, and b has a node c on its right, a left rotation is needed, where each node is shifted to the left, causing b to be the root with a on its left and c on its right. For a right rotation, let node a be the root of the tree with node b connected on its left, and let node c be connected to node b on the left. A right rotation would make b the root node, with c on its left and a on its right. Each Right-Left rotation and Left-Right rotation is a combination of the Left and Right rotations, in the named order. Each one of these rotations is of $O(1)$ time complexity. Even though multiple rotations can occur when needing to re-balance the height of the graph, as each rotation is of $O(1)$ complexity, the time consumed is $O(1)$ in total.

So, the total running time of deleting a node from an AVL Tree is $O(1+1+nlog(n))$ which can be summed to $O(nlog(n))$. $\square$

Proof Summary

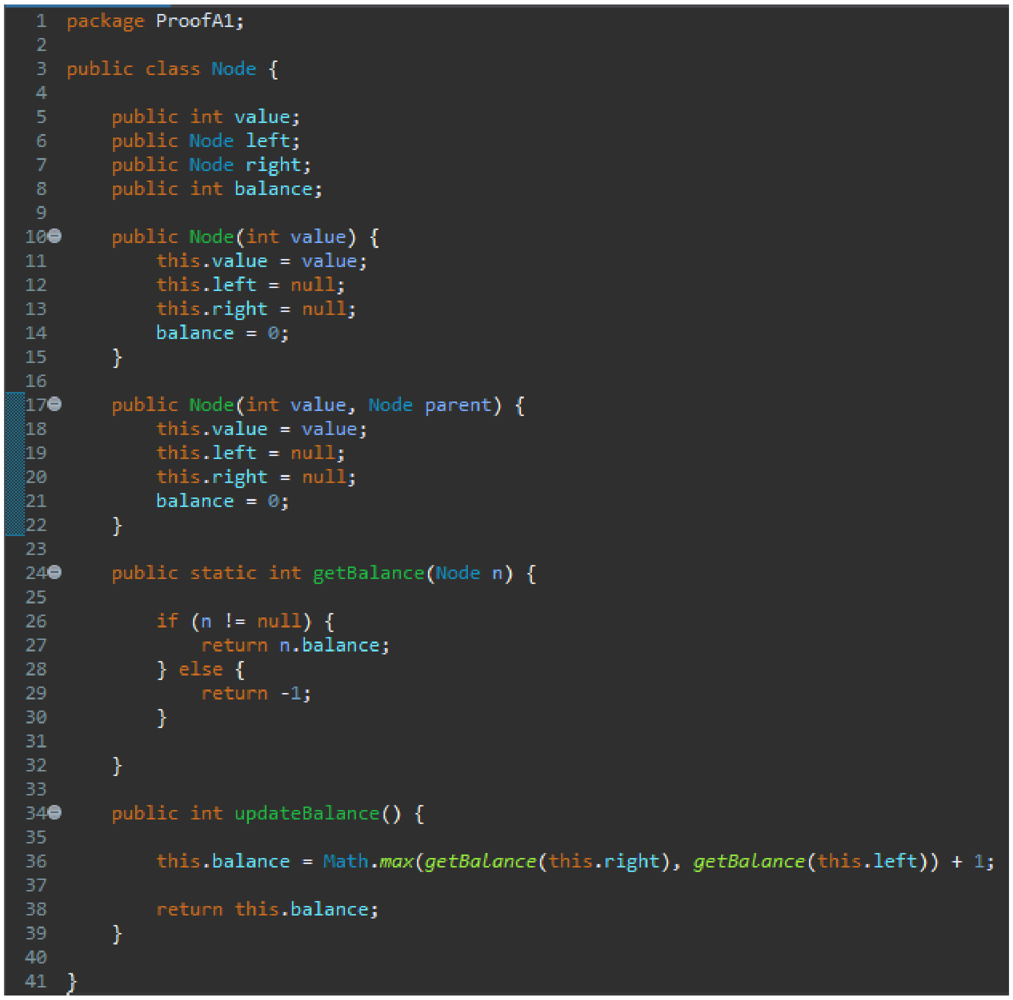
We argue the running time of each of the three main components in the procedure. The first component is traversing and finding the node to be deleted. As it is part of another proof, we can assume that this portion is $O(log(n))$. The next component is the deletion of the node which is ran in $O(1) time$, since we already located the node and are shifting the children from the node to replace the deleted node. The last component is the re-balancing of the tree, which can result in multiple re-balancing procedures to occur, but since each action of re-balancing runs in $O(1)$ time complexity, it can be summed into $O(1)$. Combining these time complexities together, we have $O(nlog(n))$.

Algorithm

Below is the Java code for deleting a node in an AVL Tree. I partially developed some of this code, while the some parts were obtained and modified from an online source [4]. I developed the classes Node (Figure 1) and the graphing function (Figure 4), while the AVLTree functions which included re-balancing (Figure 2) and node deletion (Figure 3).

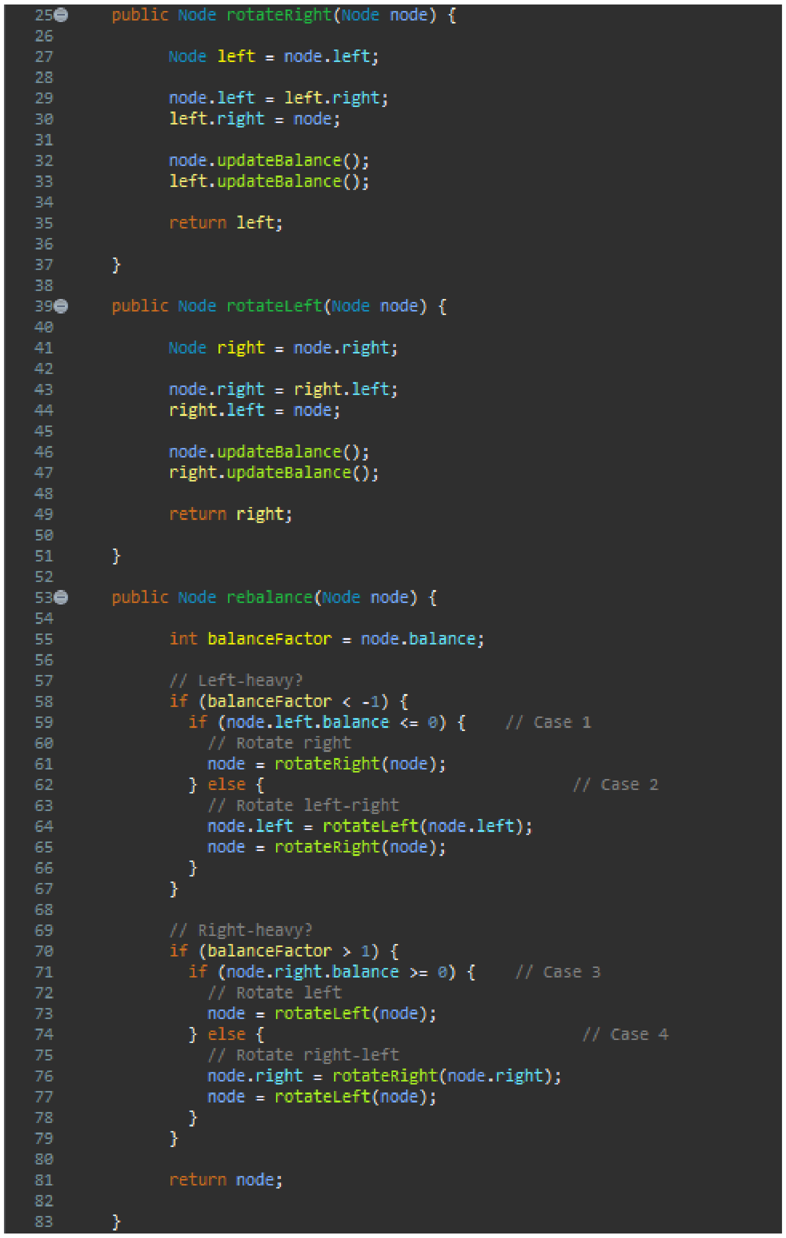
I timed the execution of the deletion of a node in an AVL Tree where the number of nodes increase with each run, and a random node is selected and deleted from the tree. I started with the number of nodes n at $10,000,000$ which increases up to $1,000,000,000$ nodes in the end. The graph randomly picks a node by taking the size of the tree, and using the $Math.random$ function to create a number between 0 and n.

The graph in Figure 4 is measured in microseconds, which shows the execution time as a function of the number of nodes within the tree. We expected the execution time to be of the function $log(n)$ as the worse case possible, but the graph does not necessarily prove it, as some of the run times exceeded greater than the worse time complexity, but that could be due to a higher constant base value to be added to the function.



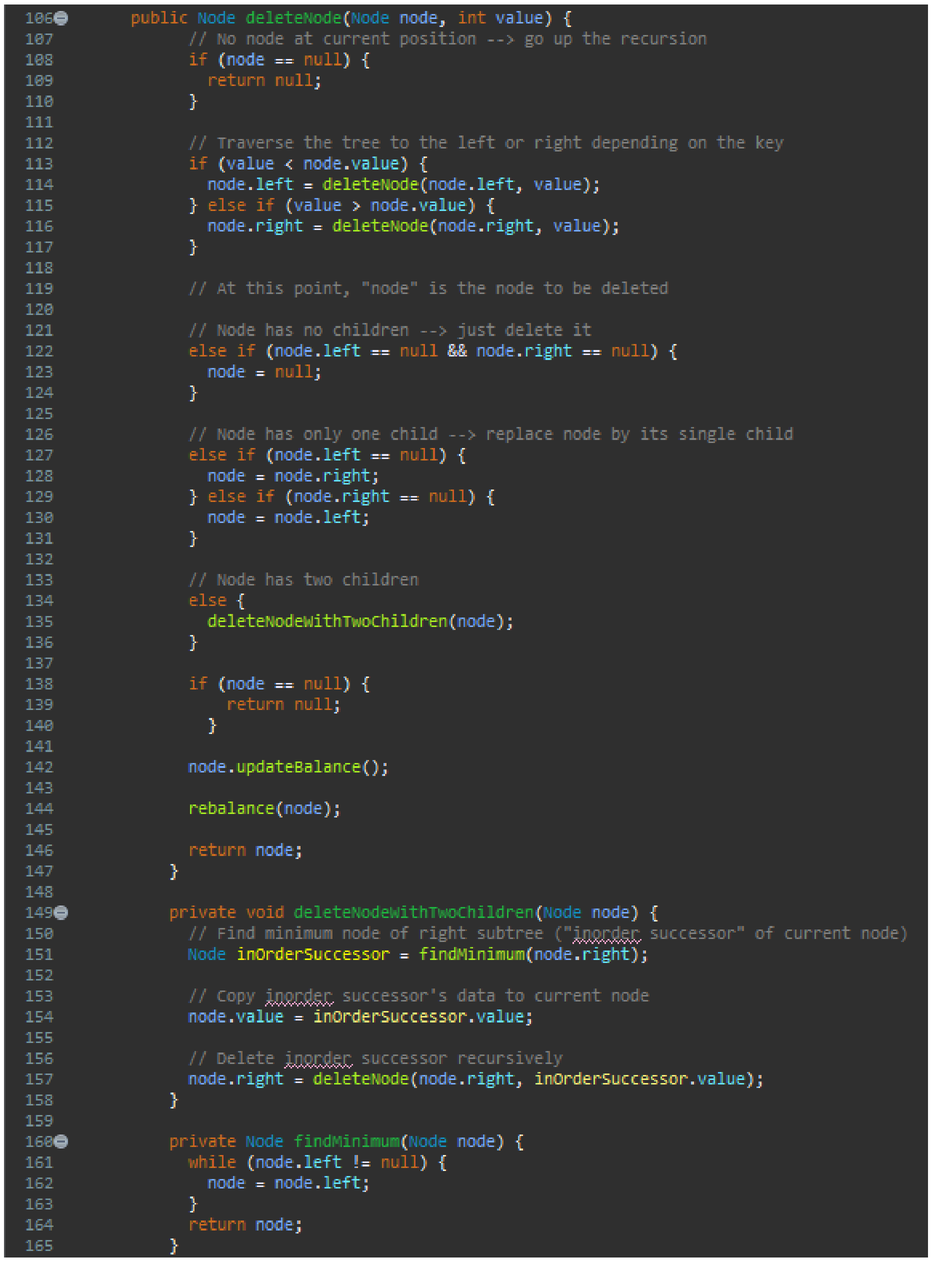
```java
package ProofA1;

public class Node {

    public int value;
    public Node left;
    public Node right;
    public int balance;

    public Node(int value) {
        this.value = value;
        this.left = null;
        this.right = null;
        balance = 0;
    }

    public Node(int value, Node parent) {
        this.value = value;
        this.left = null;
        this.right = null;
        balance = 0;
    }

    public static int getBalance(Node n) {

        if (n != null) {
            return n.balance;
        } else {
            return -1;
        }

    }

    public int updateBalance() {

        this.balance = Math.max(getBalance(this.right), getBalance(this.left)) + 1;

        return this.balance;
    }

}
```

Figure 1: Node class from `Node.java`.



```java
public Node rotateRight(Node node) {

      Node left = node.left;

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

      node.updateBalance();
      left.updateBalance();

      return left;

}

public Node rotateLeft(Node node) {

      Node right = node.right;

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

      node.updateBalance();
      right.updateBalance();

      return right;

}

public Node rebalance(Node node) {

      int balanceFactor = node.balance;

      // Left-heavy?
      if (balanceFactor < -1) {
        if (node.left.balance <= 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 (node.right.balance >= 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: Re-balancing functions from `AVLTree.java`.



```java
    public Node deleteNode(Node node, int value) {
        // 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 (value < node.value) {
          node.left = deleteNode(node.left, value);
        } else if (value > node.value) {
          node.right = deleteNode(node.right, value);
        }

        // 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);
        }

        if (node == null) {
            return null;
          }

        node.updateBalance();

        rebalance(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.value = inOrderSuccessor.value;

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

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

Figure 3: Delete Node functions from `AVLTree.java`.

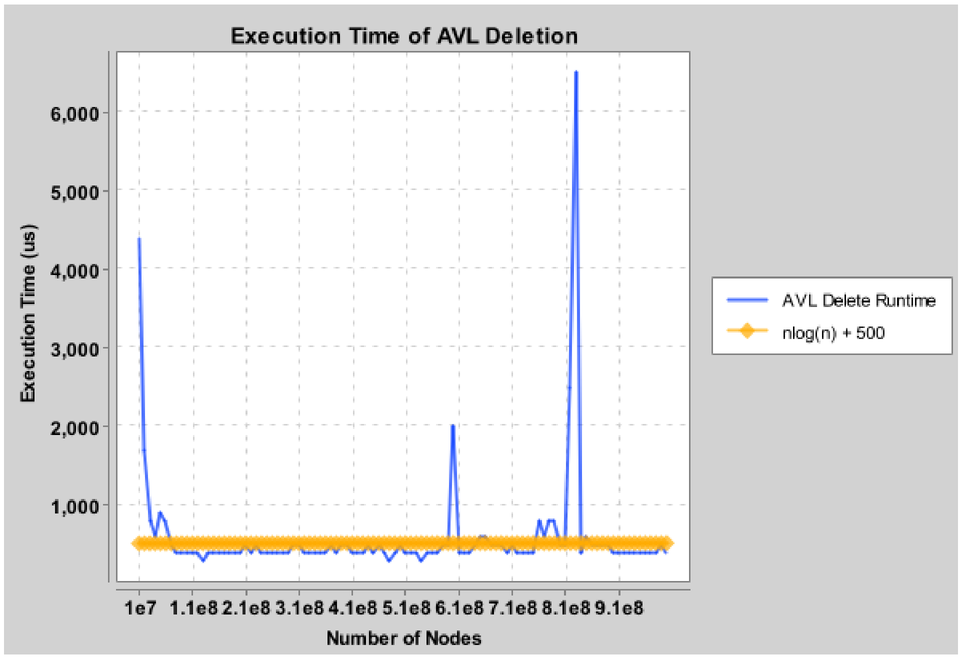


Figure 4: Execution time in microseconds of the deletion of a node from an AVL Tree as a function of n. The code to generate this graph is in `AVLProof.java`.

Real World Application

An AVL tree re-balances itself after insertion and deletion so that the parent node's children do not have a size difference of greater than one. This re-balancing makes it slower when inserting and deleting, but faster during the search, only having a time complexity of $O(nlog(n))$. A useful application would be routing tables, which are stored in routers and used to determine where data packets are supposed to be directed. As there is the possibility of needing to have more searching capabilities than adding and removing, an AVL tree can be used for the routing table, minimizing the time to search for each end location [3].

Claim 2

The DIJKSTRA'S ALGORITHM, ran on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof

This proof is a paraphrased version of the proof written in [1], page 680-682.

Proof. Let S be the initial empty set of vertices and Q be the initial set of vertices will all of G's vertices We can prove this through contradictions by assuming that a vertex u, does not follow the rule of $u.d! = \delta(s, u)$ when adding u to S. From this we can assume three things:

1. $u \neq s$ as $u.d = s.d = 0$ would break the assumption, as $\delta(s, s) = 0$.
2. A path p exists between u and s, as without so $\delta(s, u) = \infty$ and $u.d = \infty$ at the time of insertion, breaking our assumption.
3. u will be selected out of order, where at least one of the connected nodes is relaxed, as not doing so would make $u.d = \delta(s, u)$.

From this, let $x, y \in V$, where $x \in S$ and $y \in Q$. Let $p1, p2 \in E$, where $p1$ connects s to x and $p2$ connects y to u.
Let y be the predecessor vertex to u and be the shortest path between s and u. Let x be the predecessor vertex to y.
As $x \in S$, we can assume that $x.d = \delta(s, x)$ and that the $Relax(x, y)$ has occurred, filling a value into $y.d$. As u is the first vertex to make the assumption of $u \neq \delta(s, y)$, we can assume that $y.d = \delta(s, y)$. As y is the predecessor vertex to u and the weights are non-negative, we can assume that $\delta(s, y) \leq \delta(s, y)$. Therefore, this creates $y.d = \delta(s, y) \leq \delta(s, u) \leq u.d$.
If we let u be the next node taken from $EXTRACT\_MIN(Q)$, that would mean that $u.d <= y.d$.
From those two, we can use the upper-bound limits and find that $y.d = \delta(s, y) = \delta(s, u) = u.d$. This breaks our contradiction of $u.d \neq \delta(s, u)$, which should then be proven true for all subsequent cases $v.d = \delta(s, v)$ upon v being inserted into S, $v \in V$. Therefore, upon the termination of Dijkstra's algorithm, all $v.d = \delta(s, v)$, $v \in V$. □

Proof summary

The core concept of the proof is that a vertex's distance is the smallest it can possibly be when entering the "visited" set of vertices. This is proven by assuming that a vertex's distance from the source node is not the smallest it can be, creating a contradiction. From this, we can assume that the vertex is not the source vertex, there has to be a path, and that this can only happen if that specific vertex is picked out of order, as these all break the contradiction immediately.
If there is a vertex before the contradiction vertex, then the contradiction vertex's distance from the source is at least greater or equal to the distance of the previous vertex, assuming that all weights are positive. However, if the contradiction vertex is picked from the priority queue before the vertex before it, it would mean that the previous vertex has an equal or greater distance

from the source from the contradiction vertex. Using the upper bounds of both, we can prove an equality that the contradiction vertex's distance from the source at the time of insertion into the "visited" set, is the smallest it can be.

Algorithm

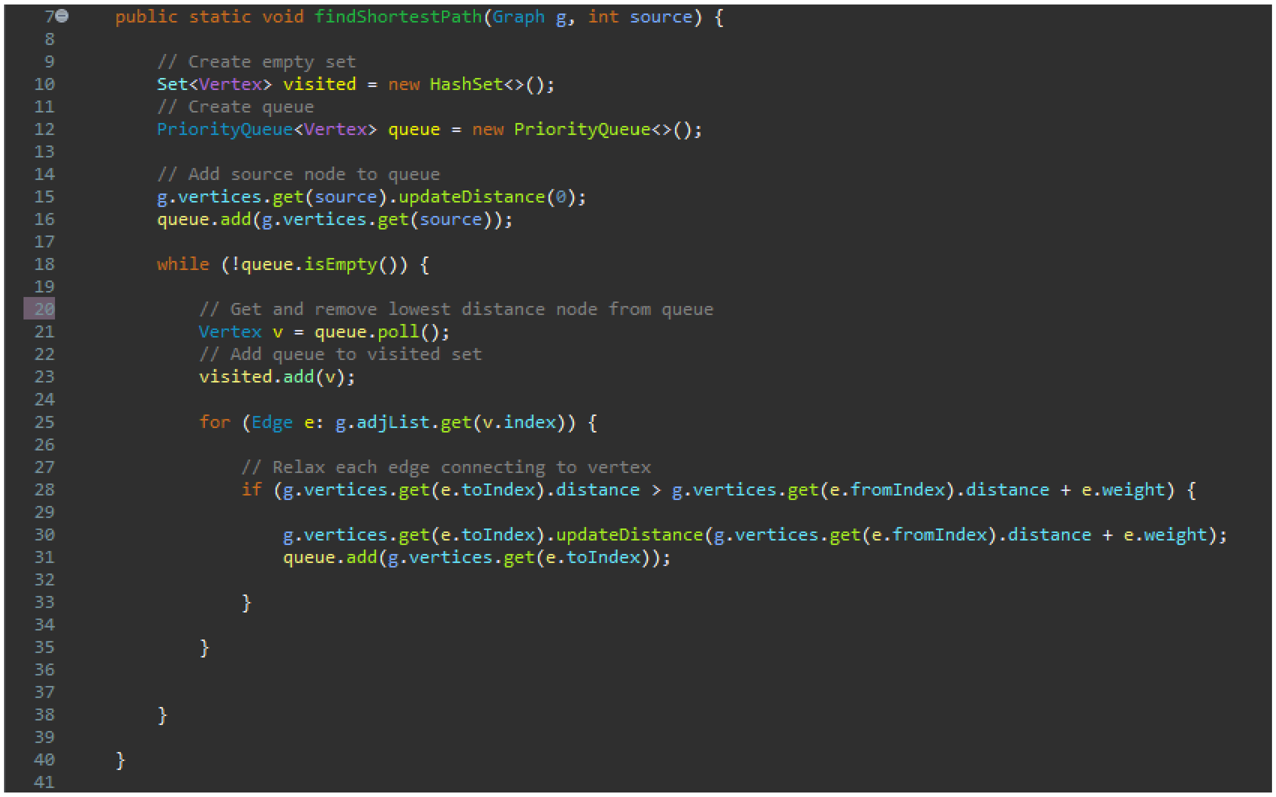
Below is the Java code for finding the shortest path using Dijkstra's Algorithm (Figure 5). The implementation required many basic classes, including Vertex (Figure 6), Edge (Figure 7), and Graph (Figure 8).

The main function ran the algorithm on three test cases, one being general and the other two being edge cases. The construction of the test cases is shown in Figure 9 and the output of the test cases is in Figure 10.

The general case contained four vertices $\{1, 2, 3, 4\}$ as well as five edges connecting them $\{(1, 2, 3), (1, 3, 5), (2, 3, 1), (2, 4, 6), (3, 4, 1)\}$ in the format: (starting vertex, ending vertex, weight). This case is clear as the shortest path is to go from index $\{1- > 2- > 3- > 4\}$, gathering a total of $3 + 1 + 1 = 6$ weight.

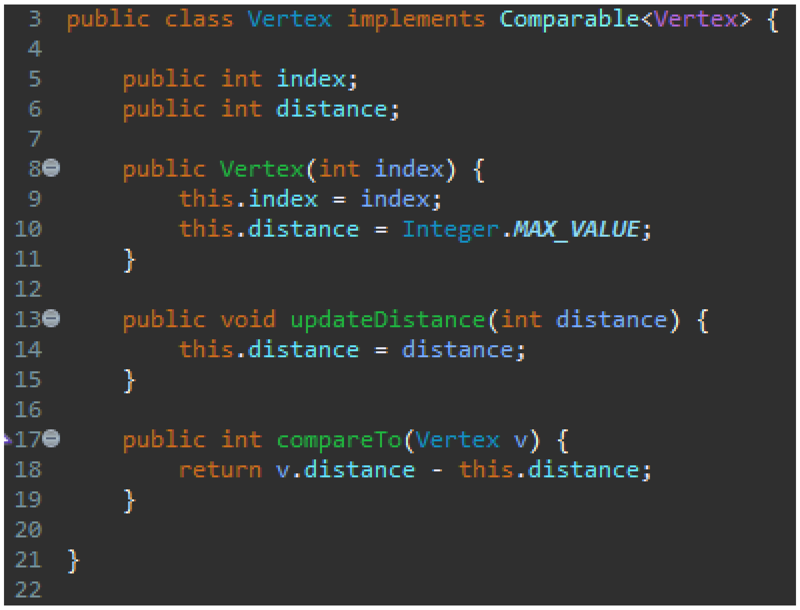
The next edge case is the empty edge case, $E = \{\}$, and there is only one vertex, the source vertex, in this case $\{1\}$. The expected and observed output is for the source vertex to be at a weight of zero.

The last edge case is when there is a Vertex that is not connected by any edges. The vertices are $\{1, 2, 3, 4\}$ and the edges are $\{(1, 2, 3), (1, 3, 5), (2, 3, 1)\}$, where vertex index four is not connected. In this case, the output is the shortest path to each vertex, except for index four, where the distance from the source node is the maximum integer allowed, which represents ∞.



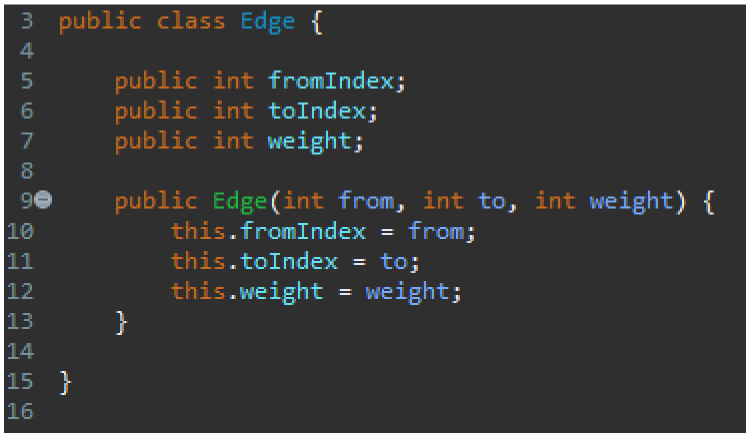
```java
    public static void findShortestPath(Graph g, int source) {

        // Create empty set
        Set<Vertex> visited = new HashSet<>();
        // Create queue
        PriorityQueue<Vertex> queue = new PriorityQueue<>();

        // Add source node to queue
        g.vertices.get(source).updateDistance(0);
        queue.add(g.vertices.get(source));

        while (!queue.isEmpty()) {

            // Get and remove lowest distance node from queue
            Vertex v = queue.poll();
            // Add queue to visited set
            visited.add(v);

            for (Edge e: g.adjList.get(v.index)) {

                // Relax each edge connecting to vertex
                if (g.vertices.get(e.toIndex).distance > g.vertices.get(e.fromIndex).distance + e.weight) {

                    g.vertices.get(e.toIndex).updateDistance(g.vertices.get(e.fromIndex).distance + e.weight);
                    queue.add(g.vertices.get(e.toIndex));

                }

            }


        }

    }

```

Figure 5: Algorithm for finding the shortest-path in a graph from `Dijkstra.java`.



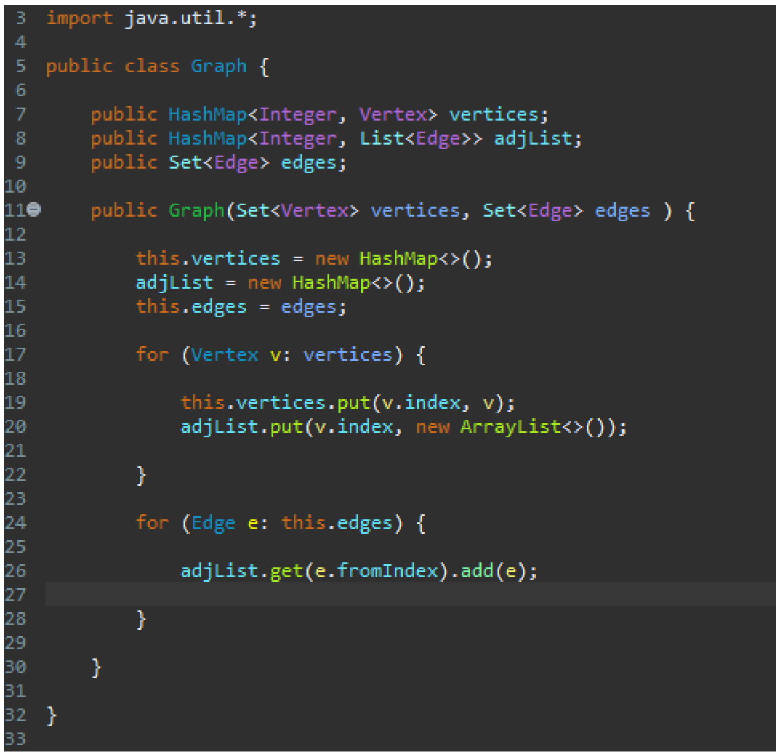
```java
public class Vertex implements Comparable<Vertex> {

    public int index;
    public int distance;

    public Vertex(int index) {
        this.index = index;
        this.distance = Integer.MAX_VALUE;
    }

    public void updateDistance(int distance) {
        this.distance = distance;
    }

    public int compareTo(Vertex v) {
        return v.distance - this.distance;
    }

}
```

Figure 6: Vertex class from `Vertex.java`.



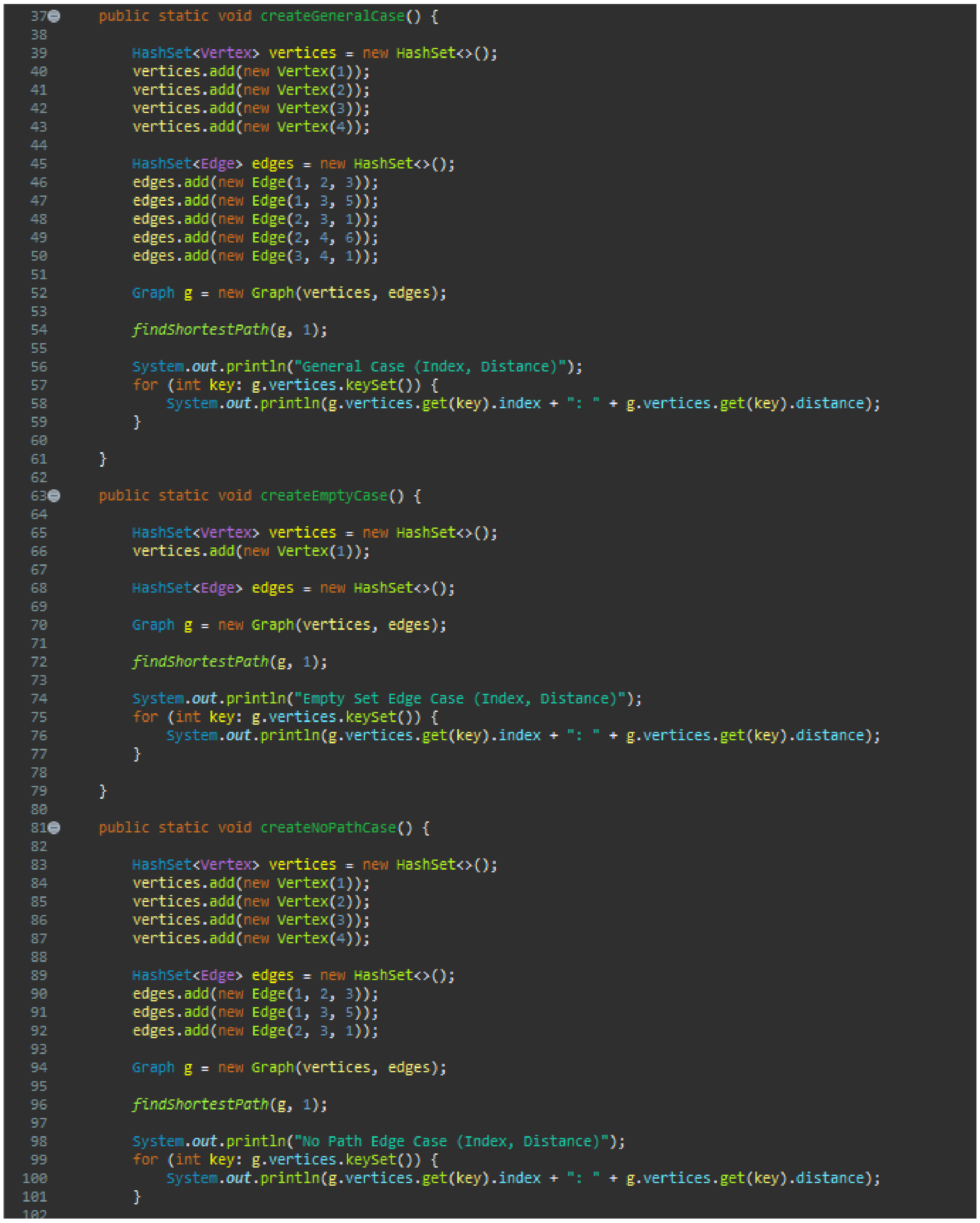
```java
public class Edge {

    public int fromIndex;
    public int toIndex;
    public int weight;

    public Edge(int from, int to, int weight) {
        this.fromIndex = from;
        this.toIndex = to;
        this.weight = weight;
    }

}
```

Figure 7: Edge class from `Edge.java`.



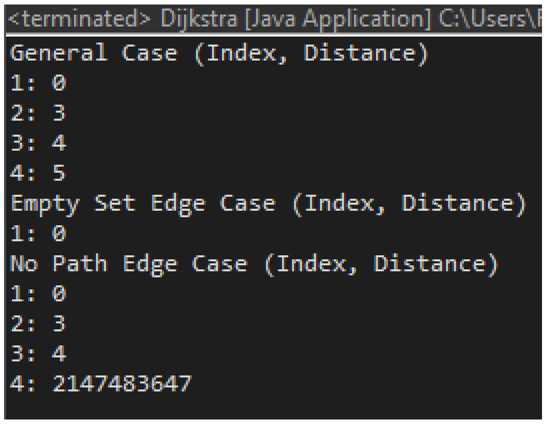
```java
import java.util.*;

public class Graph {

    public HashMap<Integer, Vertex> vertices;
    public HashMap<Integer, List<Edge>> adjList;
    public Set<Edge> edges;

    public Graph(Set<Vertex> vertices, Set<Edge> edges ) {

        this.vertices = new HashMap<>();
        adjList = new HashMap<>();
        this.edges = edges;

        for (Vertex v: vertices) {

            this.vertices.put(v.index, v);
            adjList.put(v.index, new ArrayList<>());

        }

        for (Edge e: this.edges) {

            adjList.get(e.fromIndex).add(e);

        }

    }

}
```

Figure 8: Graph class from `Graph.java`.



```java
    public static void createGeneralCase() {

        HashSet<Vertex> vertices = new HashSet<>();
        vertices.add(new Vertex(1));
        vertices.add(new Vertex(2));
        vertices.add(new Vertex(3));
        vertices.add(new Vertex(4));

        HashSet<Edge> edges = new HashSet<>();
        edges.add(new Edge(1, 2, 3));
        edges.add(new Edge(1, 3, 5));
        edges.add(new Edge(2, 3, 1));
        edges.add(new Edge(2, 4, 6));
        edges.add(new Edge(3, 4, 1));

        Graph g = new Graph(vertices, edges);

        findShortestPath(g, 1);

        System.out.println("General Case (Index, Distance)");
        for (int key: g.vertices.keySet()) {
            System.out.println(g.vertices.get(key).index + ": " + g.vertices.get(key).distance);
        }

    }

    public static void createEmptyCase() {

        HashSet<Vertex> vertices = new HashSet<>();
        vertices.add(new Vertex(1));

        HashSet<Edge> edges = new HashSet<>();

        Graph g = new Graph(vertices, edges);

        findShortestPath(g, 1);

        System.out.println("Empty Set Edge Case (Index, Distance)");
        for (int key: g.vertices.keySet()) {
            System.out.println(g.vertices.get(key).index + ": " + g.vertices.get(key).distance);
        }

    }

    public static void createNoPathCase() {

        HashSet<Vertex> vertices = new HashSet<>();
        vertices.add(new Vertex(1));
        vertices.add(new Vertex(2));
        vertices.add(new Vertex(3));
        vertices.add(new Vertex(4));

        HashSet<Edge> edges = new HashSet<>();
        edges.add(new Edge(1, 2, 3));
        edges.add(new Edge(1, 3, 5));
        edges.add(new Edge(2, 3, 1));

        Graph g = new Graph(vertices, edges);

        findShortestPath(g, 1);

        System.out.println("No Path Edge Case (Index, Distance)");
        for (int key: g.vertices.keySet()) {
            System.out.println(g.vertices.get(key).index + ": " + g.vertices.get(key).distance);
        }
```

Figure 9: Functions to create general and edge test cases for Dijkstra's algorithm from `Dijkstra.java`.



```
<terminated> Dijkstra [Java Application] C:\Users\
General Case (Index, Distance)
1: 0
2: 3
3: 4
4: 5
Empty Set Edge Case (Index, Distance)
1: 0
No Path Edge Case (Index, Distance)
1: 0
2: 3
3: 4
4: 2147483647
```

Figure 10: Console output after running the findShortestPath method on the three test cases.

Real World Application

This algorithm can be used to find the single-source shortest path on weighted, directed graphs, where the edges only contain non-negative numbers by finding the lowest non-visited vertex and adding its connected vertices to the queue ([1] pages 679). A real-world example of Dijkstra's Algorithm would be digital mapping services, like Google Maps, to find directions and a path to a location within the shortest amount of time possible. As there are many paths and methods to get to a single location, Dijkstra's Algorithm can be used to find the shortest path whether it be by walking, by public transport, or by car [3].

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Erik Demaine, Piotr Indyk, and Manolis Kellis. Lecture notes in avl trees. `https://courses.csail.mit.edu/6.006/spring11/rec/rec04.pdf`, February 2011.

[3] Dipesh Kumar. Applications of dijkstra's shortest path algorithm. `https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm`. Geek for Geeks.

[4] Sven Woltmann. Avl tree. `https://www.happycoders.eu/algorithms/avl-tree-java/`, August 2021.