Proof Activity Report

261047851

Comp251, Winter 2023

Claim 1

It takes $\mathcal{O}(\log n)$ time to search for a node x in an AVL tree.

Proof

The following proof is summarized from Goodrich and Tamassia *Algorithm Design and Applications* [6].

Proof. Searching for a node in an AVL tree takes time $\mathcal{O}(\log n)$ with respect to the height of the tree.

1. Recurrence for minimum number of internal nodes

   We can write a recurrence relation to determine the minimum number of internal nodes n_h. We use h to denote the height of the tree. When $h = 1$, the number of internal nodes is 1 and $n_1 = 1$. When $h = 2$, the number of internal nodes is 2 and $n_2 = 2$.

   For a general height h, the tree will have a root and two subtrees, also AVL trees. The height of the subtrees are $h-1$ and $h-2$ respectively. We can obtain the following formula for $h \geq 3$:

   $$n_h = 1 + n_{h-1} + n_{h-2}$$

   In the above, h is equal to the root and its two subtrees denoted by n_{h-1} and n_{h-2}. We note this is a Fibonacci sequence with n_{h-1} always greater than n_{h-2}, allowing us to simplify the equation: $n_h > 2n_{h-2}$

   Simplifying further as done in COMP 251 Lecture 6, Slide 17 [4]:

   $$n_h = 2 \times 2 \times n_{h-4} > 2 \times 2 \times 2 \times n_{h-6} \cdots > 2^{\frac{h}{2}}$$

   $$n_h > 2^{\frac{h}{2}}$$

   Taking log of both sides:

   $$2 \times log(n_h) > h$$

   $$h = \mathcal{O}(\log n)$$

   The time it takes to search for a node depends on the height of the tree and therefore takes time $\mathcal{O}(\log n)$

□

Proof Summary

An AVL tree is a self-balancing binary search tree. For every internal node, the height of the left vs. right child differs by at most one. An AVL tree with a minimum number of internal nodes represents the worst case example of an AVL tree where the left and the right child differ by a height of one for every subtree. As with binary search trees, searching for a node in the tree is dependent on the tree height. However, as AVL trees are self-balancing, ensuring there is never a larger difference than one between the subtree heights, the height of the tree can be bounded by $\mathcal{O}(\log n)$.

Searching for a node involves beginning with a lead/head node and querying whether the node in question's value is greater than or less than the value of the head node. The head node is then updated and the function recursively calls itself, advancing down either the left or the right subtree. There is a constant amount of time taken for the if-else statements to determine which subtree to traverse and $\mathcal{O}(\log n)$ recursive calls are made to search the height of the tree in the worst case.

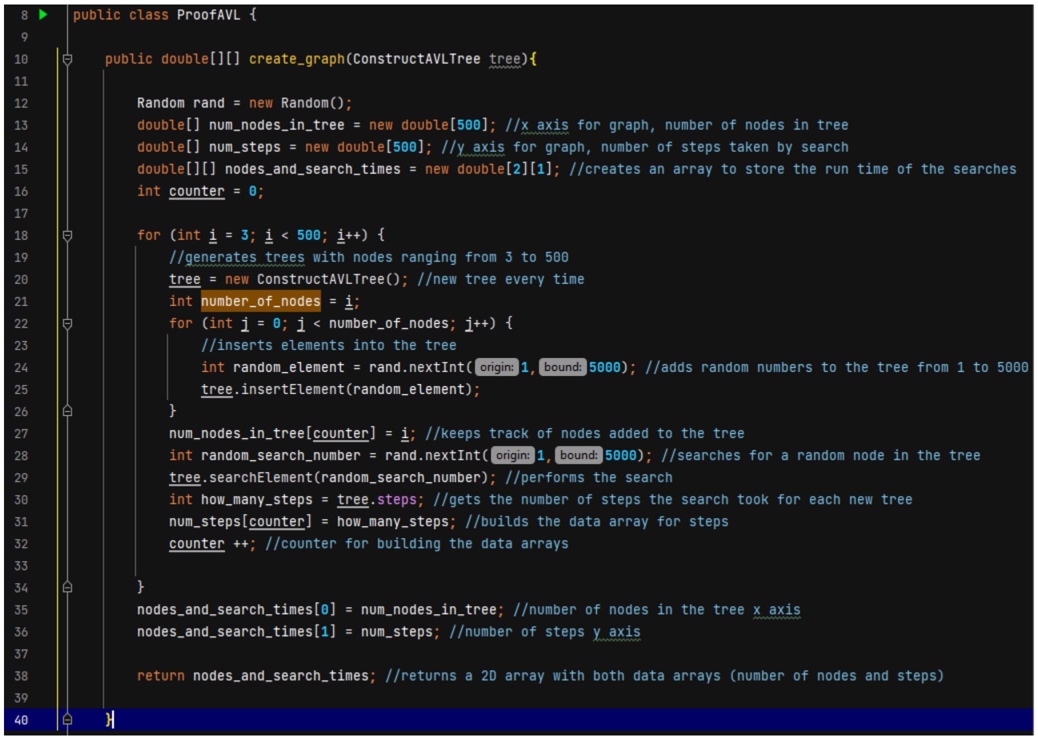
Algorithm

The following are sections of Java code used to execute a search on a random node in an AVL tree. The code was taken from: JavaTpoint [2]. The first figure (Figure 1, Lines 9-37), shows how the data arrays for the graph are generated. One array represents the number of nodes in the AVL tree, and another represents the number of steps taken by the search algorithm in searching for a random node. The first for loop creates AVL trees with 3-500 nodes. The second for loop inserts the desired number of nodes. Each node's element is a randomly generated integer between 1 and 5000. The node array is updated and a search is performed on a randomly generated integer, again in the range 1 to 5000. The number of recursive calls to the search function is recorded in the number of steps array. The function returns both data arrays. The values can then be printed and copied into an Excel sheet to generate a graph (Figure 2).

The code for the ConstructAVLTree and Node class was taken from https://www.javatpoint.com/avl-tree-program-in-java and is written by JavaTpoint 2021. [2] The comments are my own. (Figure 3) is the node class for the tree. Each node's element is an integer. It has a left/right child field, as well as a height property.(Figure 4) and (Figure 5) shows the code needed to initialize an AVL tree as well as the function to recursively insert nodes.

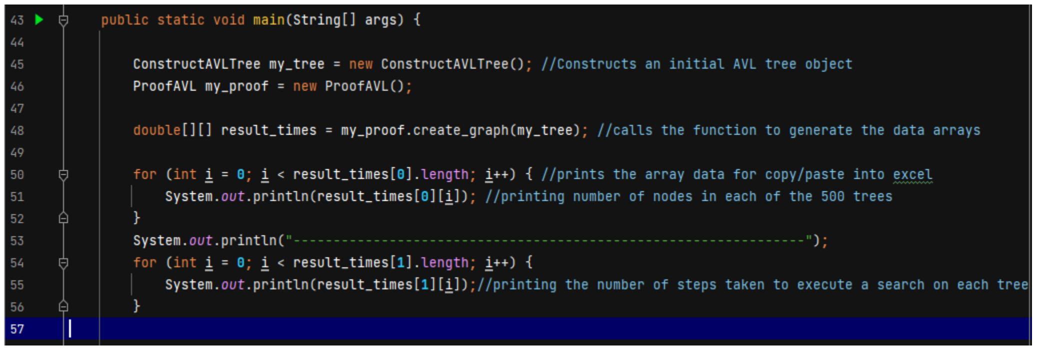
(Figure 6) and (Figure7) show the rotation methods used to maintain the AVL properties of the tree, ensuring that the left and right subtrees do not differ by more than one. Multiple rotation methods are needed to cover the cases when nodes to rotate also have children.

Finally we see the search function itself in (Figure 8). The search begins by querying whether the search element in question is greater or smaller than the root. The search then begins to advance down one of the subtrees, calling itself recursively and each time asking whether the search element is greater or smaller than the parent node. The search ends when the element is found or when the height of the tree is traversed without finding the element. As the AVL trees are balanced, we expect this search time to be $\mathcal{O}(\log n)$. By using the method mentioned above to generate 500 trees and executing a search on each of them, we plot their node quantity on the x axis and search time in steps on the y axis. We attain the graph in (Figure 9) and can see that this conforms to $\mathcal{O}(\log n)$.



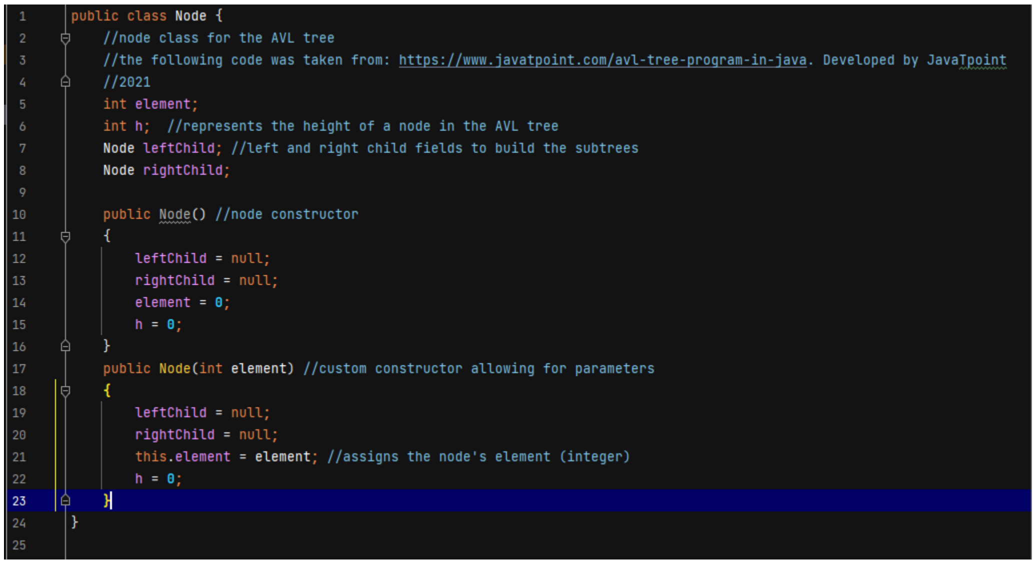
```java
public class ProofAVL {

    public double[][] create_graph(ConstructAVLTree tree){

        Random rand = new Random();
        double[] num_nodes_in_tree = new double[500]; //x axis for graph, number of nodes in tree
        double[] num_steps = new double[500]; //y axis for graph, number of steps taken by search
        double[][] nodes_and_search_times = new double[2][1]; //creates an array to store the run time of the searches
        int counter = 0;

        for (int i = 3; i < 500; i++) {
            //generates trees with nodes ranging from 3 to 500
            tree = new ConstructAVLTree(); //new tree every time
            int number_of_nodes = i;
            for (int j = 0; j < number_of_nodes; j++) {
                //inserts elements into the tree
                int random_element = rand.nextInt( origin: 1, bound: 5000); //adds random numbers to the tree from 1 to 5000
                tree.insertElement(random_element);
            }
            num_nodes_in_tree[counter] = i; //keeps track of nodes added to the tree
            int random_search_number = rand.nextInt( origin: 1, bound: 5000); //searches for a random node in the tree
            tree.searchElement(random_search_number); //performs the search
            int how_many_steps = tree.steps; //gets the number of steps the search took for each new tree
            num_steps[counter] = how_many_steps; //builds the data array for steps
            counter ++; //counter for building the data arrays

        }
        nodes_and_search_times[0] = num_nodes_in_tree; //number of nodes in the tree x axis
        nodes_and_search_times[1] = num_steps; //number of steps y axis

        return nodes_and_search_times; //returns a 2D array with both data arrays (number of nodes and steps)

    }
```

Figure 1: Create Graph from `ProofAVL.java`.



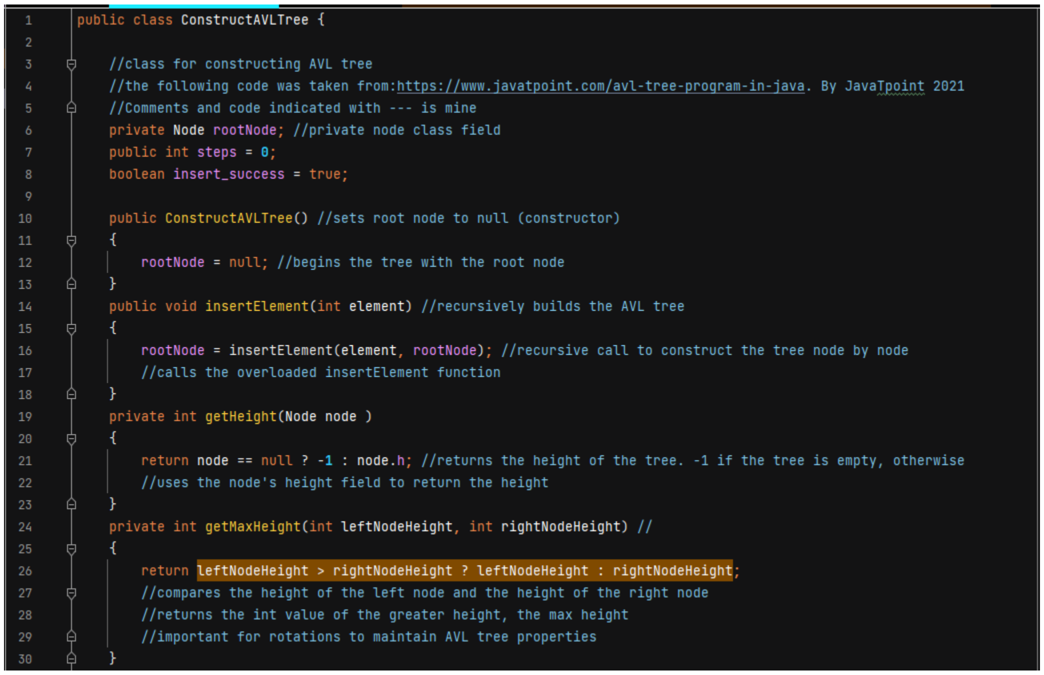
```java
    public static void main(String[] args) {

        ConstructAVLTree my_tree = new ConstructAVLTree(); //Constructs an initial AVL tree object
        ProofAVL my_proof = new ProofAVL();

        double[][] result_times = my_proof.create_graph(my_tree); //calls the function to generate the data arrays

        for (int i = 0; i < result_times[0].length; i++) { //prints the array data for copy/paste into excel
            System.out.println(result_times[0][i]); //printing number of nodes in each of the 500 trees
        }
        System.out.println("------------------------------------------------------------");
        for (int i = 0; i < result_times[1].length; i++) {
            System.out.println(result_times[1][i]);//printing the number of steps taken to execute a search on each tree
        }

```

Figure 2: Main method from `ProofAVL.java`.



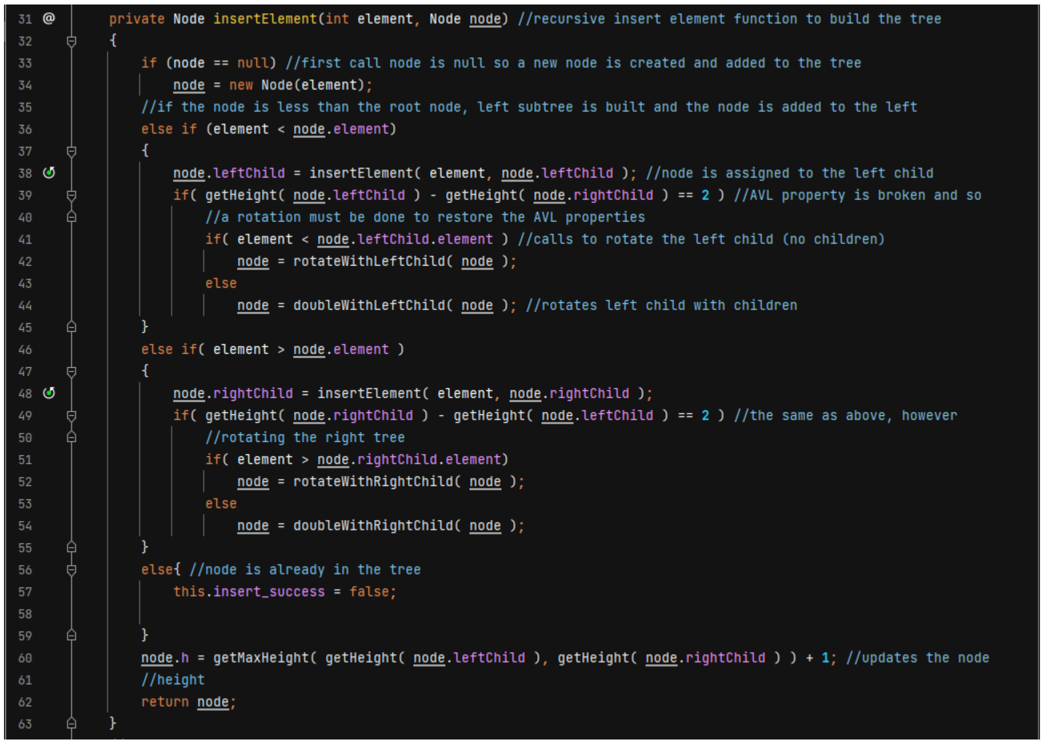
```java
public class Node {
    //node class for the AVL tree
    //the following code was taken from: https://www.javatpoint.com/avl-tree-program-in-java. Developed by JavaTpoint
    //2021
    int element;
    int h;  //represents the height of a node in the AVL tree
    Node leftChild; //left and right child fields to build the subtrees
    Node rightChild;

    public Node() //node constructor
    {
        leftChild = null;
        rightChild = null;
        element = 0;
        h = 0;
    }
    public Node(int element) //custom constructor allowing for parameters
    {
        leftChild = null;
        rightChild = null;
        this.element = element; //assigns the node's element (integer)
        h = 0;
    }
}
```

Figure 3: Node class from `Node.java`.



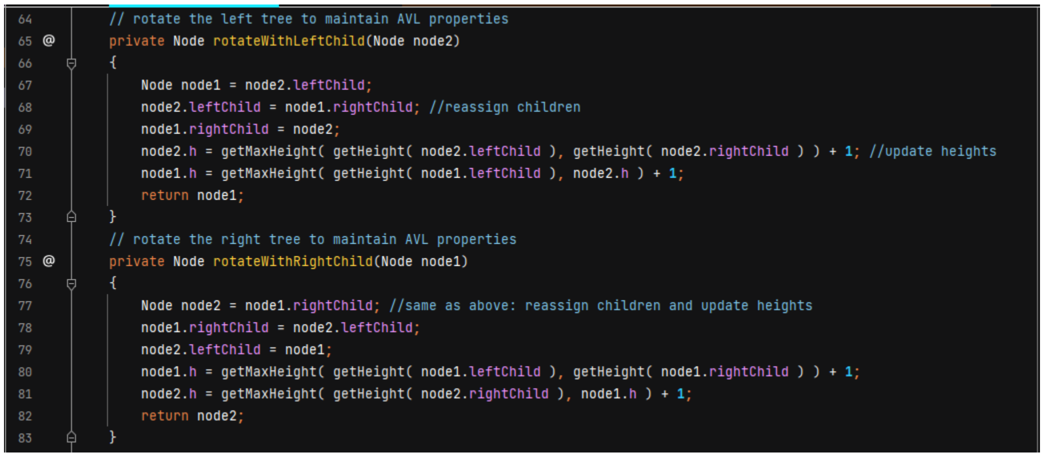
```java
public class ConstructAVLTree {

    //class for constructing AVL tree
    //the following code was taken from:https://www.javatpoint.com/avl-tree-program-in-java. By JavaTpoint 2021
    //Comments and code indicated with --- is mine
    private Node rootNode; //private node class field
    public int steps = 0;
    boolean insert_success = true;

    public ConstructAVLTree() //sets root node to null (constructor)
    {
        rootNode = null; //begins the tree with the root node
    }
    public void insertElement(int element) //recursively builds the AVL tree
    {
        rootNode = insertElement(element, rootNode); //recursive call to construct the tree node by node
        //calls the overloaded insertElement function
    }
    private int getHeight(Node node )
    {
        return node == null ? -1 : node.h; //returns the height of the tree. -1 if the tree is empty, otherwise
        //uses the node's height field to return the height
    }
    private int getMaxHeight(int leftNodeHeight, int rightNodeHeight) //
    {
        return leftNodeHeight > rightNodeHeight ? leftNodeHeight : rightNodeHeight;
        //compares the height of the left node and the height of the right node
        //returns the int value of the greater height, the max height
        //important for rotations to maintain AVL tree properties
    }
```

Figure 4: ConstructAVLTree class from `ConstructAVLTree.java`.



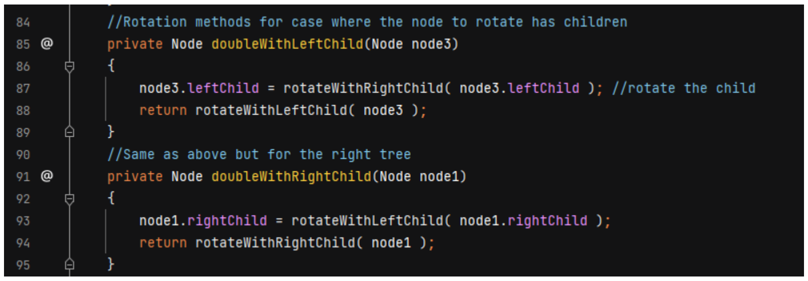
```java
private Node insertElement(int element, Node node) //recursive insert element function to build the tree
{
    if (node == null) //first call node is null so a new node is created and added to the tree
        node = new Node(element);
    //if the node is less than the root node, left subtree is built and the node is added to the left
    else if (element < node.element)
    {
        node.leftChild = insertElement( element, node.leftChild ); //node is assigned to the left child
        if( getHeight( node.leftChild ) - getHeight( node.rightChild ) == 2 ) //AVL property is broken and so
            //a rotation must be done to restore the AVL properties
            if( element < node.leftChild.element ) //calls to rotate the left child (no children)
                node = rotateWithLeftChild( node );
            else
                node = doubleWithLeftChild( node ); //rotates left child with children
    }
    else if( element > node.element )
    {
        node.rightChild = insertElement( element, node.rightChild );
        if( getHeight( node.rightChild ) - getHeight( node.leftChild ) == 2 ) //the same as above, however
            //rotating the right tree
            if( element > node.rightChild.element)
                node = rotateWithRightChild( node );
            else
                node = doubleWithRightChild( node );
    }
    else{ //node is already in the tree
        this.insert_success = false;

    }
    node.h = getMaxHeight( getHeight( node.leftChild ), getHeight( node.rightChild ) ) + 1; //updates the node
    //height
    return node;
}
```

Figure 5: ConstructAVLTree class from `ConstructAVLTree.java`.



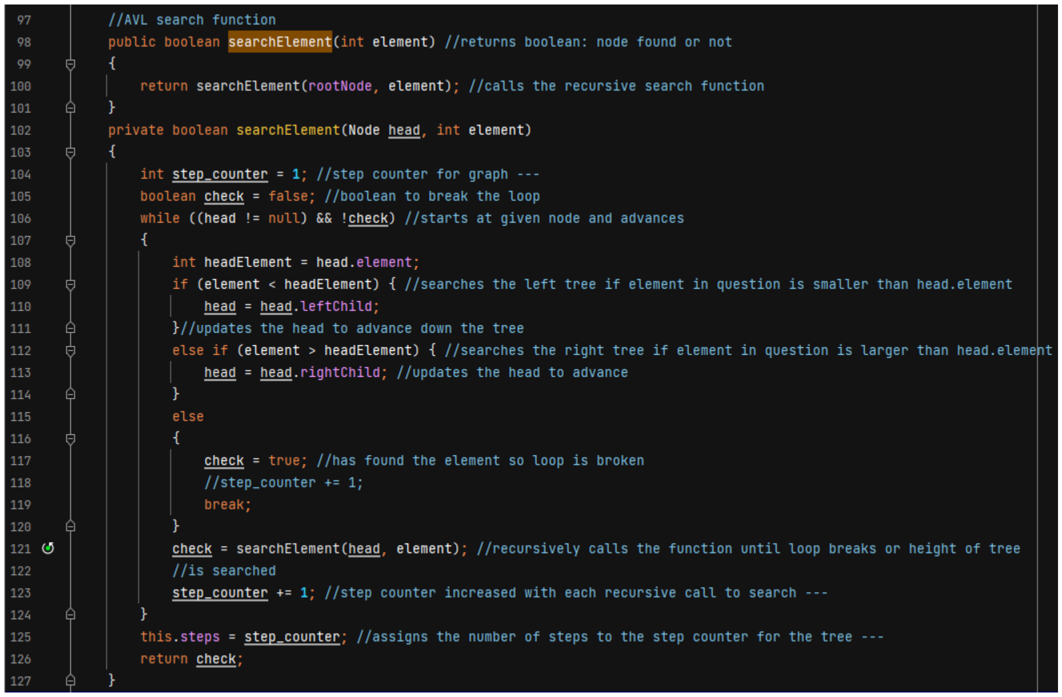
```java
// rotate the left tree to maintain AVL properties
private Node rotateWithLeftChild(Node node2)
{
    Node node1 = node2.leftChild;
    node2.leftChild = node1.rightChild; //reassign children
    node1.rightChild = node2;
    node2.h = getMaxHeight( getHeight( node2.leftChild ), getHeight( node2.rightChild ) ) + 1; //update heights
    node1.h = getMaxHeight( getHeight( node1.leftChild ), node2.h ) + 1;
    return node1;
}
// rotate the right tree to maintain AVL properties
private Node rotateWithRightChild(Node node1)
{
    Node node2 = node1.rightChild; //same as above: reassign children and update heights
    node1.rightChild = node2.leftChild;
    node2.leftChild = node1;
    node1.h = getMaxHeight( getHeight( node1.leftChild ), getHeight( node1.rightChild ) ) + 1;
    node2.h = getMaxHeight( getHeight( node2.rightChild ), node1.h ) + 1;
    return node2;
}
```

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



```java
//Rotation methods for case where the node to rotate has children
private Node doubleWithLeftChild(Node node3)
{
    node3.leftChild = rotateWithRightChild( node3.leftChild ); //rotate the child
    return rotateWithLeftChild( node3 );
}
//Same as above but for the right tree
private Node doubleWithRightChild(Node node1)
{
    node1.rightChild = rotateWithLeftChild( node1.rightChild );
    return rotateWithRightChild( node1 );
}
```

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



```java
//AVL search function
public boolean searchElement(int element) //returns boolean: node found or not
{
    return searchElement(rootNode, element); //calls the recursive search function
}
private boolean searchElement(Node head, int element)
{
    int step_counter = 1; //step counter for graph ---
    boolean check = false; //boolean to break the loop
    while ((head != null) && !check) //starts at given node and advances
    {
        int headElement = head.element;
        if (element < headElement) { //searches the left tree if element in question is smaller than head.element
            head = head.leftChild;
        }//updates the head to advance down the tree
        else if (element > headElement) { //searches the right tree if element in question is larger than head.element
            head = head.rightChild; //updates the head to advance
        }
        else
        {
            check = true; //has found the element so loop is broken
            //step_counter += 1;
            break;
        }
        check = searchElement(head, element); //recursively calls the function until loop breaks or height of tree
        //is searched
        step_counter += 1; //step counter increased with each recursive call to search ---
    }
    this.steps = step_counter; //assigns the number of steps to the step counter for the tree ---
    return check;
}
```

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

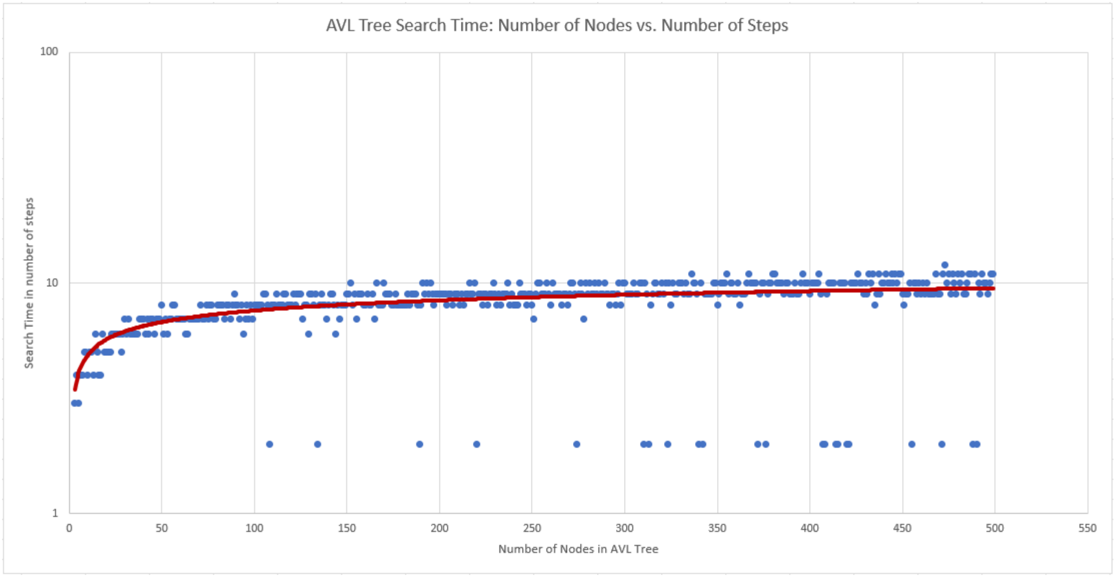


Figure 9: Search Time (number of steps) vs. Nodes in an AVL Tree. The data from this graph is generated using the function in `ProofAVL.java`.

Real World Application

Searching for a node in an AVL tree has many applications, for example in Cryptography. In one example, Zhongyuan et. al made use of AVL trees to store the IDs and public keys needed for managing and securing wireless sensor networks. These are wireless networks consisting of a number of sensor nodes and can be used for industrial, military, or medical uses. An attacker could theoretically gain access to a node and take over the network, so ensuring that the links between neighboring nodes remains secure is paramount. The neighboring nodes' IDs and public keys are stored in the AVL tree and search time is greatly reduced. The above example is from "An Efficient Key Management Scheme Based on ECC and AVL Tree for Large Scale Wireless Sensor Networks" by Zhongyuan et. al. [7]

Claim 2

Dijkstra's Algorithm : Dijkstra's algorithm, run 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 taken from: Introduction to Algorithms, Third edition [5]

It uses a loop invariant and proof by contradiction to prove the claim that Dijkstra's algorithm, on a directed graph with positive weights, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$. We prove that the algorithm is indeed finding the shortest path ($u.d = \delta(s, u)$) from source to destination. We designate the loop invariant as $v.d = \delta(s, v)$, meaning that at each iteration of the while loop, $v.d$ represents the shortest path from the source s to v. For contradiction we assume that the next vertex u added to the set of vertices S is $u.d \neq \delta(s, u)$. We show that $u \neq s$ because s is part of the set S at initialization and $s.d = \delta(s, s) = 0$. This also means the set is not empty when we add u, implying there is a path from s to u. Before we add u, there exists a path p that connects s to a vertex in the set $V - S$, where V is the set of all vertices. Consider the first vertex, call it y on this path. Call x y's predecessor. We then have a path $s \hookrightarrow x \hookrightarrow y \hookrightarrow u$. Assume that $y.d = \delta(s, y)$. Since $u.d \neq \delta(s, u)$ we have $x.d = \delta(s, x)$. On our path $\delta(s, y) \leq \delta(s, u)$, so $y.d \leq u.d$. Here we derive a contradiction as y and u are already vertices in $V - S$ when we choose u. This gives us that $u.d \leq y.d$ as y was outside of the initial explored shortest path set. These equalities then only hold when $y.d = u.d$ and we see that $u.d = \delta(s, u)$, contradicting $u.d \neq \delta(s, u)$. Each iteration of the loop will give $u.d = \delta(s, u)$. $\square$

Proof summary

This summary is a summary of the above proof taken from [5]. This proof uses a form of induction on a loop invariant to prove that Dijkstra's algorithm finds the shortest path from a source node to a destination node. We want to prove the claim that on each iteration of the while loop, the vertex being added to the set of shortest path vertices, represents the shortest weighted, directed, distance from source to the vertex in question. As this property is maintained for the duration of the algorithm, terminating after the priority queue used to explore nodes and add them to the shortest path set is empty, we attain a shortest path from source to destination. We prove the claim by contradiction. First we assume that Dijkstra has been run and the shortest path computed for some initial number of vertices in a graph. We consider a destination vertex and for contradiction assume that it does not represent the shortest weighted path from the source to itself. This implies that there could be a shorter path from the source to the node in question. We then consider a node on the path y and its predecessor x. Because the destination node is not the node that gives the shortest weighted distance from the source, we know that x represents the shortest path from source to x. We can then say that the weighted distance of y is less than or equal to the weighted distance of the destination node. We derive our contradiction when we consider that y, and our destination node were already part of the set of vertices not yet included in the shortest path, so we must have that the destination node is less than or equal to the weighted distance of y. This equality only holds when y and the destination node are the same vertex. This implies that there is no shorter path and the next vertex we add has to be the vertex that gives the shortest weighted path from the source to itself. This property is maintained for every iteration of the loop and terminates when the priority queue for exploring vertices is empty or all the nodes have been assessed. This results in finding the shortest path from a source to destination vertex.

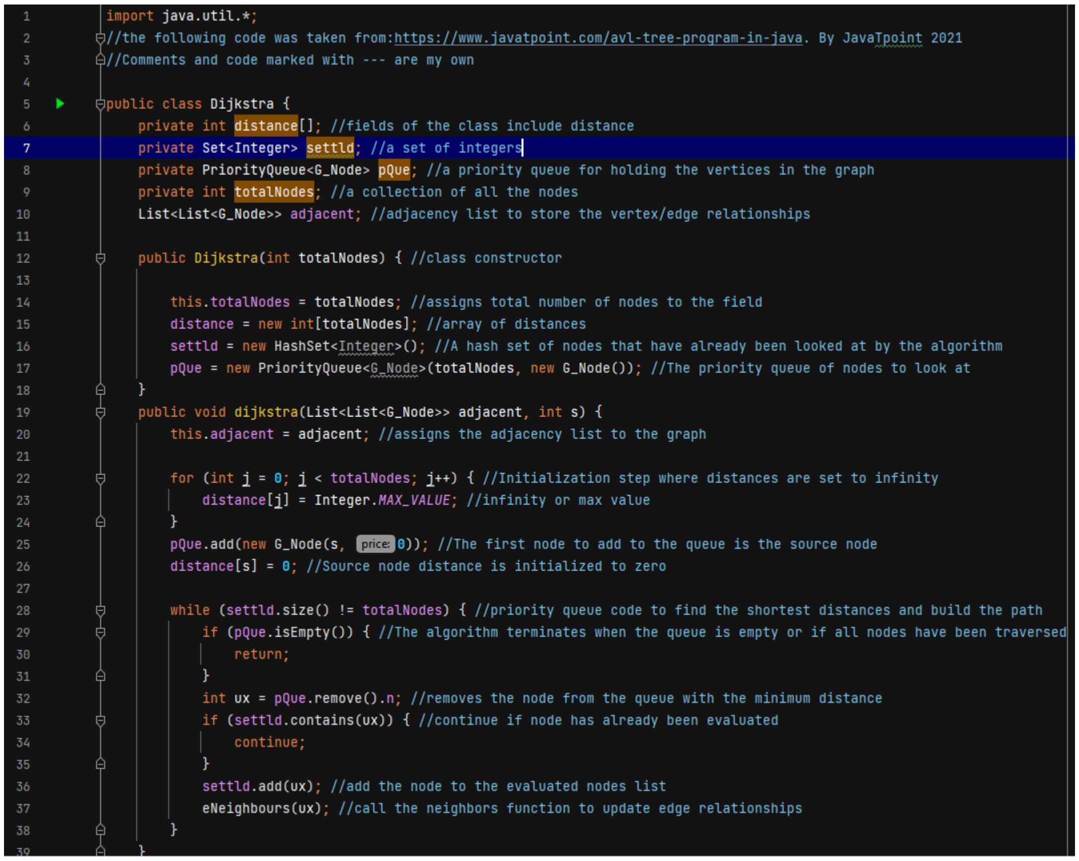
Algorithm

The following are sections of code implementing Dijkstra shortest path algorithm. The code to implement the algorithm was taken from: JavaTpoint [3]. This implementation represents the graph using adjacency lists and employs a priority queue to find the shortest path. The comparator interface is used to compare node weights and a helper function which updates neighbor relationships is included. Figure 10 shows the Dijkstra class, with a priority queue and adjacency list as field variables of the class, as well as a list of ”settled” nodes. These nodes represent nodes that have already been evaluated. Figure 11 shows the helper function to the priority queue which evaluates neighboring nodes in the adjacency list and updates distances and weights when building the shortest path. Figure 12 shows the node class that implements comparator used to build the graph.

Figure 13 shows the main method where we configure the first test case: The General Case (represented in Figures 14 and Figure 15). A graph object is created and built by adding edges between nodes and recording the weight. 9 nodes are added to the graph and relationships between these nodes and edges are created and assigned weights. For example, a node 0 is added and connected to node 1, costing a weight of 7. After the algorithm terminates, the shortest path from the source node to each destination node is printed. Looking at the shortest path from 0 to 2 for example, the path has a weight of 8. We can follow the edge relationships in the adjacency list to see that this is correct. 0 is adjacent to 1 at a cost of 3. 1 is adjacent to 8 at a cost of 4. 8 is adjacent to 2 at a cost of 1. The total cost for the shortest path from 0 to 2 is therefore 8. 1 is also adjacent to 2 and 7, but we didn't choose those because their weights don't give us the shortest path from 0 to 2.

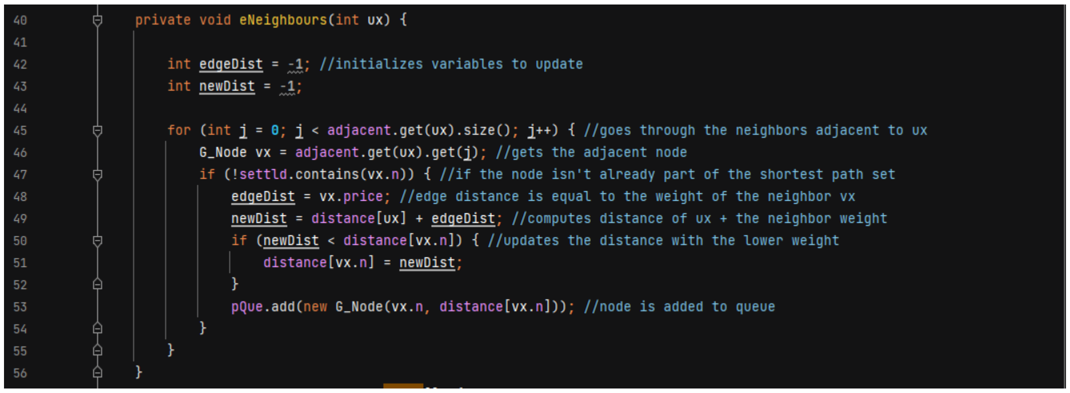
Edge case 1, represented in Figure 16 occurs when the graph only consists of one node, the source node. We expect the shortest distance from the source node to itself to be 0. This is what we can see in the print output from the general case example that includes this edge case. The distance from 0 to 0 is 0, as expected. This occurs because we have accounted for this case by initializing the source node's weight and distance to itself to be zero. This is an extra step that differs from initializing all the other node distances to infinity and can cause many problems if this step is omitted.

Edge case 2, represented in Figures 17 and Figure 18 occurs if there is no path from the source node to the destination node. For this edge case we create a new adjacency list with the 9 nodes from previously. This time however, we connect most of the nodes to node 4. This leaves no path from 0 to nodes 2,3,5,6,7 and 8. This is what we see represented in the output in Figure 18. Because we initialized distances to be infinity at the beginning of the algorithm (or in this case Int MAX), we expect to see this value returned as the distance from the source to the destination. This confirms that there is no path from the source to destination and the behaviour is as expected.



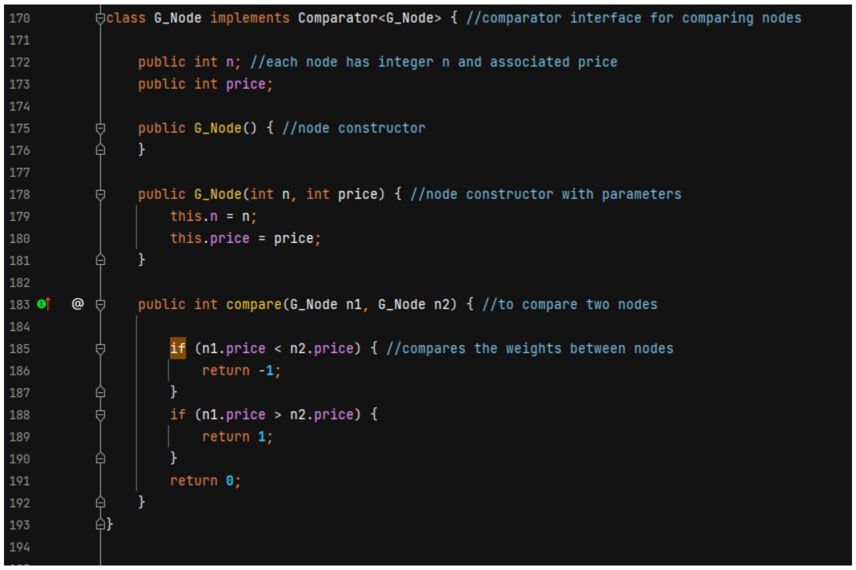
```java
import java.util.*;
//the following code was taken from:https://www.javatpoint.com/avl-tree-program-in-java. By JavaTpoint 2021
//Comments and code marked with --- are my own

public class Dijkstra {
    private int distance[]; //fields of the class include distance
    private Set<Integer> settld; //a set of integers
    private PriorityQueue<G_Node> pQue; //a priority queue for holding the vertices in the graph
    private int totalNodes; //a collection of all the nodes
    List<List<G_Node>> adjacent; //adjacency list to store the vertex/edge relationships

    public Dijkstra(int totalNodes) { //class constructor

        this.totalNodes = totalNodes; //assigns total number of nodes to the field
        distance = new int[totalNodes]; //array of distances
        settld = new HashSet<Integer>(); //A hash set of nodes that have already been looked at by the algorithm
        pQue = new PriorityQueue<G_Node>(totalNodes, new G_Node()); //The priority queue of nodes to look at
    }
    public void dijkstra(List<List<G_Node>> adjacent, int s) {
        this.adjacent = adjacent; //assigns the adjacency list to the graph

        for (int j = 0; j < totalNodes; j++) { //Initialization step where distances are set to infinity
            distance[j] = Integer.MAX_VALUE; //infinity or max value
        }
        pQue.add(new G_Node(s, price: 0)); //The first node to add to the queue is the source node
        distance[s] = 0; //Source node distance is initialized to zero

        while (settld.size() != totalNodes) { //priority queue code to find the shortest distances and build the path
            if (pQue.isEmpty()) { //The algorithm terminates when the queue is empty or if all nodes have been traversed
                return;
            }
            int ux = pQue.remove().n; //removes the node from the queue with the minimum distance
            if (settld.contains(ux)) { //continue if node has already been evaluated
                continue;
            }
            settld.add(ux); //add the node to the evaluated nodes list
            eNeighbours(ux); //call the neighbors function to update edge relationships
        }
    }
```

Figure 10: Dijkstra class from `Dijkstra.java`.



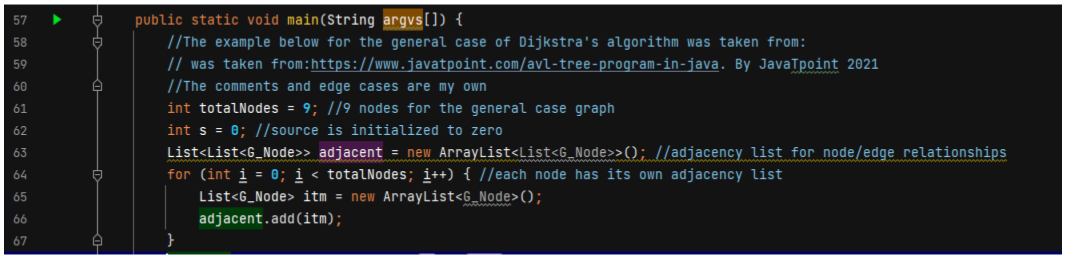
```java
private void eNeighbours(int ux) {

    int edgeDist = -1; //initializes variables to update
    int newDist = -1;

    for (int j = 0; j < adjacent.get(ux).size(); j++) { //goes through the neighbors adjacent to ux
        G_Node vx = adjacent.get(ux).get(j); //gets the adjacent node
        if (!settld.contains(vx.n)) { //if the node isn't already part of the shortest path set
            edgeDist = vx.price; //edge distance is equal to the weight of the neighbor vx
            newDist = distance[ux] + edgeDist; //computes distance of ux + the neighbor weight
            if (newDist < distance[vx.n]) { //updates the distance with the lower weight
                distance[vx.n] = newDist;
            }
            pQue.add(new G_Node(vx.n, distance[vx.n])); //node is added to queue
        }
    }
}
```

Figure 11: Neighbor function to update neighboring edge weights `Dijkstra.java`.



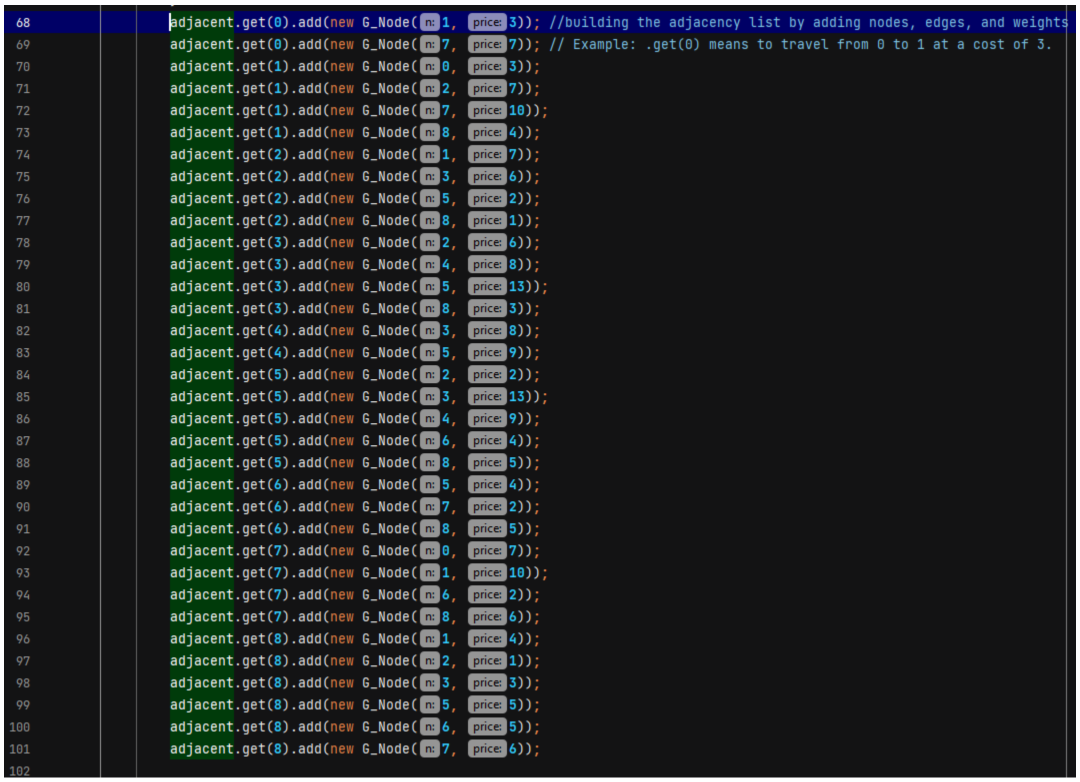
```java
class G_Node implements Comparator<G_Node> { //comparator interface for comparing nodes

    public int n; //each node has integer n and associated price
    public int price;

    public G_Node() { //node constructor
    }

    public G_Node(int n, int price) { //node constructor with parameters
        this.n = n;
        this.price = price;
    }

    @
    public int compare(G_Node n1, G_Node n2) { //to compare two nodes

        if (n1.price < n2.price) { //compares the weights between nodes
            return -1;
        }
        if (n1.price > n2.price) {
            return 1;
        }
        return 0;
    }
}

```

Figure 12: G Node class that implements Comparator from `Dijkstra.java`.



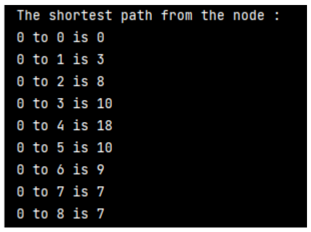
```java
public static void main(String argvs[]) {
    //The example below for the general case of Dijkstra's algorithm was taken from:
    // was taken from:https://www.javatpoint.com/avl-tree-program-in-java. By JavaTpoint 2021
    //The comments and edge cases are my own
    int totalNodes = 9; //9 nodes for the general case graph
    int s = 0; //source is initialized to zero
    List<List<G_Node>> adjacent = new ArrayList<List<G_Node>>(); //adjacency list for node/edge relationships
    for (int i = 0; i < totalNodes; i++) { //each node has its own adjacency list
        List<G_Node> itm = new ArrayList<G_Node>();
        adjacent.add(itm);
    }
```

Figure 13: Main method from `Dijkstra.java`.



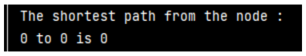
```
adjacent.get(0).add(new G_Node(n: 1, price: 3)); //building the adjacency list by adding nodes, edges, and weights
adjacent.get(0).add(new G_Node(n: 7, price: 7)); // Example: .get(0) means to travel from 0 to 1 at a cost of 3.
adjacent.get(1).add(new G_Node(n: 0, price: 3));
adjacent.get(1).add(new G_Node(n: 2, price: 7));
adjacent.get(1).add(new G_Node(n: 7, price: 10));
adjacent.get(1).add(new G_Node(n: 8, price: 4));
adjacent.get(2).add(new G_Node(n: 1, price: 7));
adjacent.get(2).add(new G_Node(n: 3, price: 6));
adjacent.get(2).add(new G_Node(n: 5, price: 2));
adjacent.get(2).add(new G_Node(n: 8, price: 1));
adjacent.get(3).add(new G_Node(n: 2, price: 6));
adjacent.get(3).add(new G_Node(n: 4, price: 8));
adjacent.get(3).add(new G_Node(n: 5, price: 13));
adjacent.get(3).add(new G_Node(n: 8, price: 3));
adjacent.get(4).add(new G_Node(n: 3, price: 8));
adjacent.get(4).add(new G_Node(n: 5, price: 9));
adjacent.get(5).add(new G_Node(n: 2, price: 2));
adjacent.get(5).add(new G_Node(n: 3, price: 13));
adjacent.get(5).add(new G_Node(n: 4, price: 9));
adjacent.get(5).add(new G_Node(n: 6, price: 4));
adjacent.get(5).add(new G_Node(n: 8, price: 5));
adjacent.get(6).add(new G_Node(n: 5, price: 4));
adjacent.get(6).add(new G_Node(n: 7, price: 2));
adjacent.get(6).add(new G_Node(n: 8, price: 5));
adjacent.get(7).add(new G_Node(n: 0, price: 7));
adjacent.get(7).add(new G_Node(n: 1, price: 10));
adjacent.get(7).add(new G_Node(n: 6, price: 2));
adjacent.get(7).add(new G_Node(n: 8, price: 6));
adjacent.get(8).add(new G_Node(n: 1, price: 4));
adjacent.get(8).add(new G_Node(n: 2, price: 1));
adjacent.get(8).add(new G_Node(n: 3, price: 3));
adjacent.get(8).add(new G_Node(n: 5, price: 5));
adjacent.get(8).add(new G_Node(n: 6, price: 5));
adjacent.get(8).add(new G_Node(n: 7, price: 6));
```

Figure 14: Building the adjacency list for the General Case from `Dijkstra.java`.



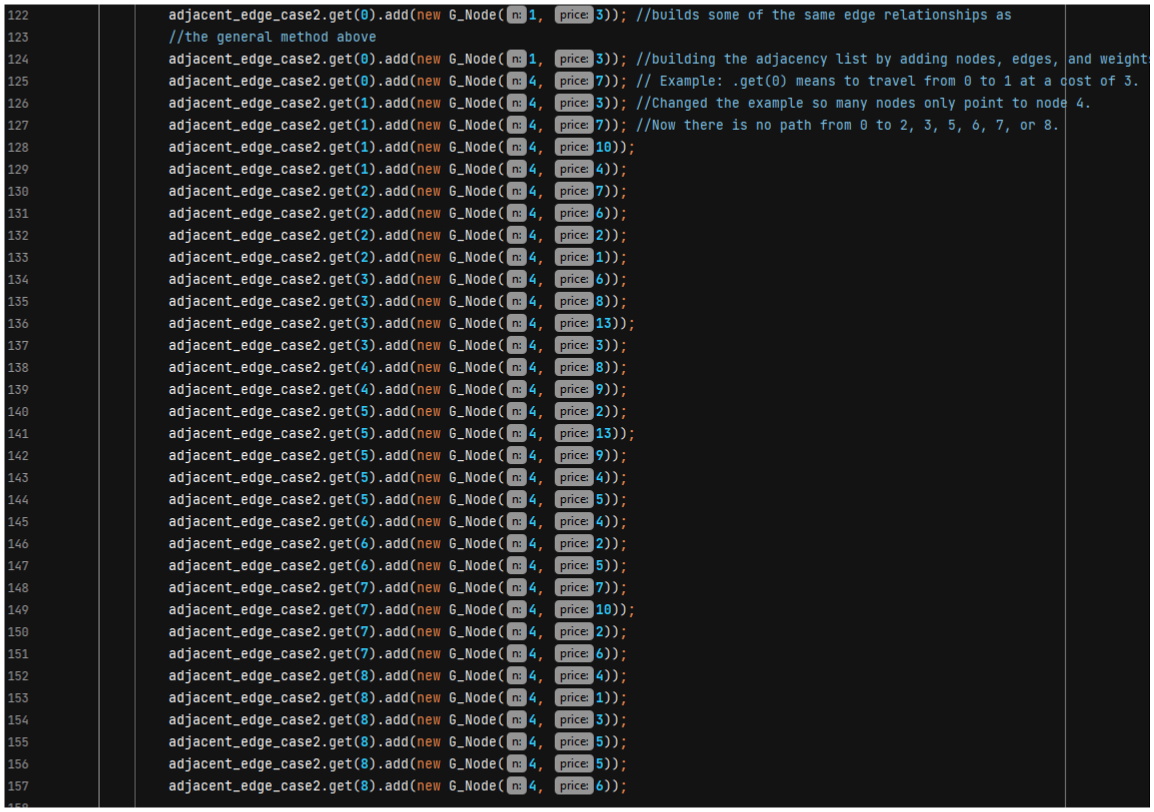
```
The shortest path from the node :
0 to 0 is 0
0 to 1 is 3
0 to 2 is 8
0 to 3 is 10
0 to 4 is 18
0 to 5 is 10
0 to 6 is 9
0 to 7 is 7
0 to 8 is 7
```

Figure 15: General Case output from `Dijkstra.java`.



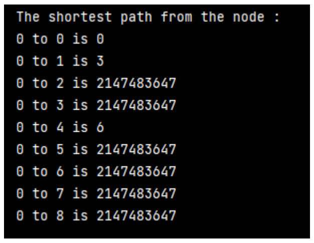
```
The shortest path from the node :
0 to 0 is 0
```

Figure 16: General Edge Case 1. Only one node `Dijkstra.java`.



```java
        adjacent_edge_case2.get(0).add(new G_Node( n: 1,  price: 3)); //builds some of the same edge relationships as
        //the general method above
        adjacent_edge_case2.get(0).add(new G_Node( n: 1,  price: 3)); //building the adjacency list by adding nodes, edges, and weight
        adjacent_edge_case2.get(0).add(new G_Node( n: 4,  price: 7)); // Example: .get(0) means to travel from 0 to 1 at a cost of 3.
        adjacent_edge_case2.get(1).add(new G_Node( n: 4,  price: 3)); //Changed the example so many nodes only point to node 4.
        adjacent_edge_case2.get(1).add(new G_Node( n: 4,  price: 7)); //Now there is no path from 0 to 2, 3, 5, 6, 7, or 8.
        adjacent_edge_case2.get(1).add(new G_Node( n: 4,  price: 10));
        adjacent_edge_case2.get(1).add(new G_Node( n: 4,  price: 4));
        adjacent_edge_case2.get(2).add(new G_Node( n: 4,  price: 7));
        adjacent_edge_case2.get(2).add(new G_Node( n: 4,  price: 6));
        adjacent_edge_case2.get(2).add(new G_Node( n: 4,  price: 2));
        adjacent_edge_case2.get(2).add(new G_Node( n: 4,  price: 1));
        adjacent_edge_case2.get(3).add(new G_Node( n: 4,  price: 6));
        adjacent_edge_case2.get(3).add(new G_Node( n: 4,  price: 8));
        adjacent_edge_case2.get(3).add(new G_Node( n: 4,  price: 13));
        adjacent_edge_case2.get(3).add(new G_Node( n: 4,  price: 3));
        adjacent_edge_case2.get(4).add(new G_Node( n: 4,  price: 8));
        adjacent_edge_case2.get(4).add(new G_Node( n: 4,  price: 9));
        adjacent_edge_case2.get(5).add(new G_Node( n: 4,  price: 2));
        adjacent_edge_case2.get(5).add(new G_Node( n: 4,  price: 13));
        adjacent_edge_case2.get(5).add(new G_Node( n: 4,  price: 9));
        adjacent_edge_case2.get(5).add(new G_Node( n: 4,  price: 4));
        adjacent_edge_case2.get(5).add(new G_Node( n: 4,  price: 5));
        adjacent_edge_case2.get(6).add(new G_Node( n: 4,  price: 4));
        adjacent_edge_case2.get(6).add(new G_Node( n: 4,  price: 2));
        adjacent_edge_case2.get(6).add(new G_Node( n: 4,  price: 5));
        adjacent_edge_case2.get(7).add(new G_Node( n: 4,  price: 7));
        adjacent_edge_case2.get(7).add(new G_Node( n: 4,  price: 10));
        adjacent_edge_case2.get(7).add(new G_Node( n: 4,  price: 2));
        adjacent_edge_case2.get(7).add(new G_Node( n: 4,  price: 6));
        adjacent_edge_case2.get(8).add(new G_Node( n: 4,  price: 4));
        adjacent_edge_case2.get(8).add(new G_Node( n: 4,  price: 1));
        adjacent_edge_case2.get(8).add(new G_Node( n: 4,  price: 3));
        adjacent_edge_case2.get(8).add(new G_Node( n: 4,  price: 5));
        adjacent_edge_case2.get(8).add(new G_Node( n: 4,  price: 5));
        adjacent_edge_case2.get(8).add(new G_Node( n: 4,  price: 6));
```

Figure 17: Adjacency list for edge case 2 `Dijkstra.java`.



```
The shortest path from the node :
0 to 0 is 0
0 to 1 is 3
0 to 2 is 2147483647
0 to 3 is 2147483647
0 to 4 is 6
0 to 5 is 2147483647
0 to 6 is 2147483647
0 to 7 is 2147483647
0 to 8 is 2147483647
```

Figure 18: Edge case 2 output `Dijkstra.java`.

Real World Application

One of the real world applications of Dijkstra's algorithm is in mapping technology. Google Maps for example, would use an algorithm like Dijkstra to help compute the shortest path between two locations. To know the shortest distance from campus to a grocery store, we could consider campus the starting vertex and the grocery store the destination vertex. There are many locations in between campus and the grocery store, and Dijkstra could be applied considering each of these locations as vertices along a potential shortest path graph. The result would be the shortest path from campus to a grocery store. This example is taken from [1].

References

[1] Applications of dijkstra's shortest path alogrithm. https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/. geeks.

[2] Avl tree code. https://www.javatpoint.com/avl-tree-program-in-java. JavaTPoint.

[3] Djikstra algorithm java. https://www.javatpoint.com/dijkstra-algorithm-java. DJavaTpoint.

[4] Comp 251 avl trees lecture 6 slide 17, 2023.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. MIT press, 2009.

[6] Michael Goodrich and Roberto Tamassia. *Algorithm Design and Applications*. Wiley press, 2015.

[7] Zhongyuan Qin and Xinshuai Zhang. An efficient key management scheme based on ecc and avl tree for large scale wireless sensor networks. page https://doi.org/10.1155/2015/691498. International Journal of Distributed Sensor Networks, 2015.