GTU Department of Computer Engineering

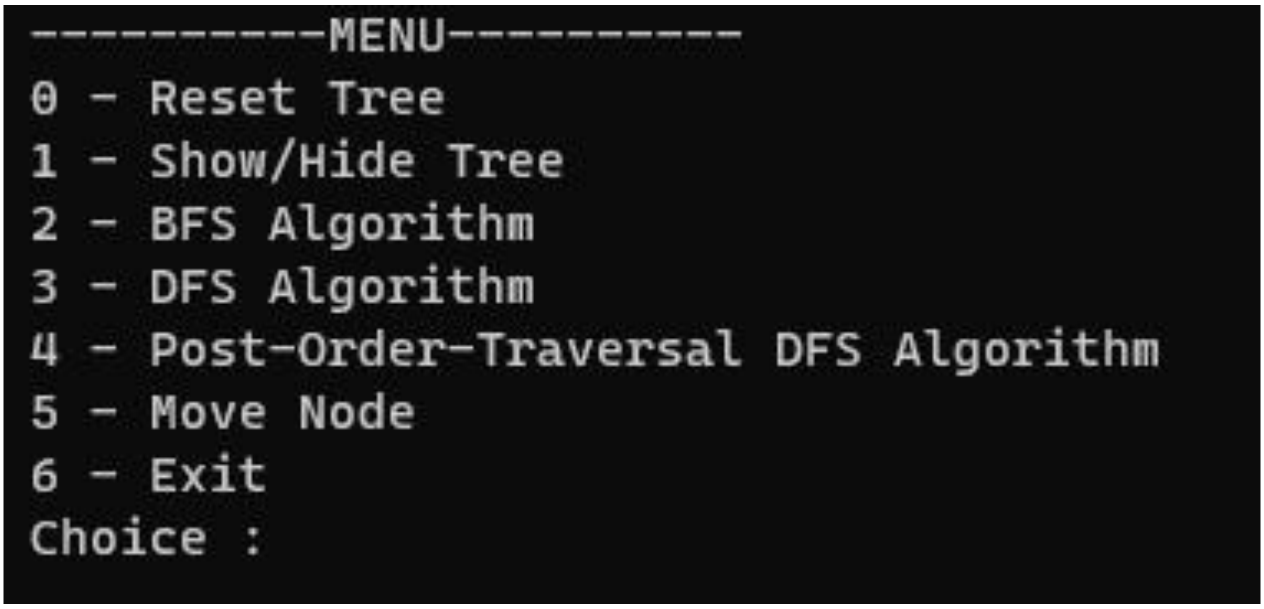
CSE 222/505 – Spring 2023

Homework 5 Report

Barış Batuhan Bolat

210104004029

1. Program Usage



```
----------MENU----------
0 - Reset Tree
1 - Show/Hide Tree
2 - BFS Algorithm
3 - DFS Algorithm
4 - Post-Order-Traversal DFS Algorithm
5 - Move Node
6 - Exit
Choice :
```

0 – Reset Tree --> Returns tree the original form(You can change txt file content without close program. Make sure press 0 after change txt file).

1 – Show/Hide Tree --> Changes the visibility of tree window

2 – BFS Algorithm --> Executes BFS Search algorithm

3 – DFS Algorithm --> Executes DFS Search algorithm

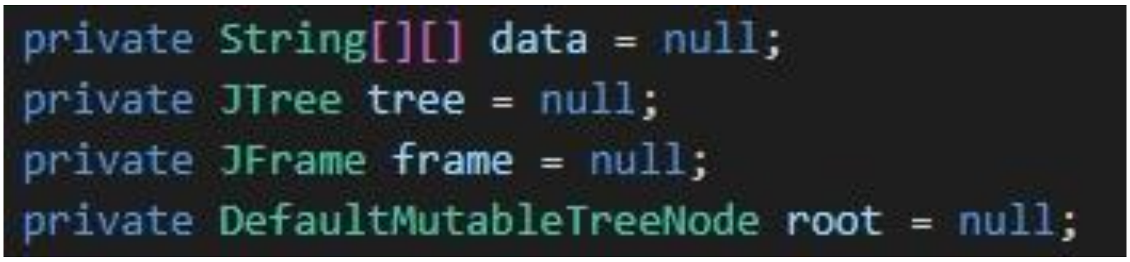
4 – Post-Order DFS Algorithm --> Executes Post-Order DFS Search algorithm

5 – Move Node --> Changes move location

6 – Exit --> Closes program

2. Method Implementations

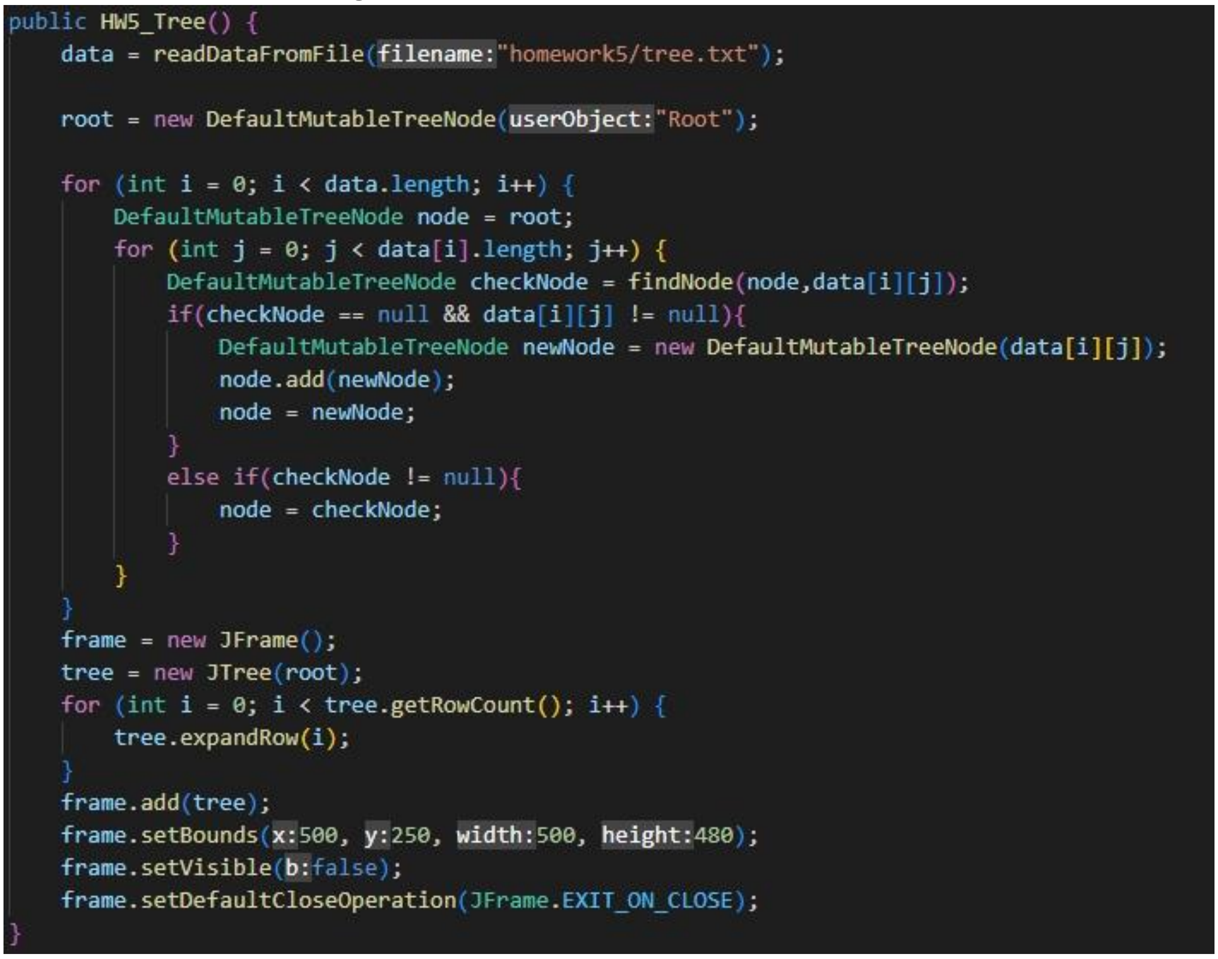
- **Fields**



```java
private String[][] data = null;
private JTree tree = null;
private JFrame frame = null;
private DefaultMutableTreeNode root = null;
```

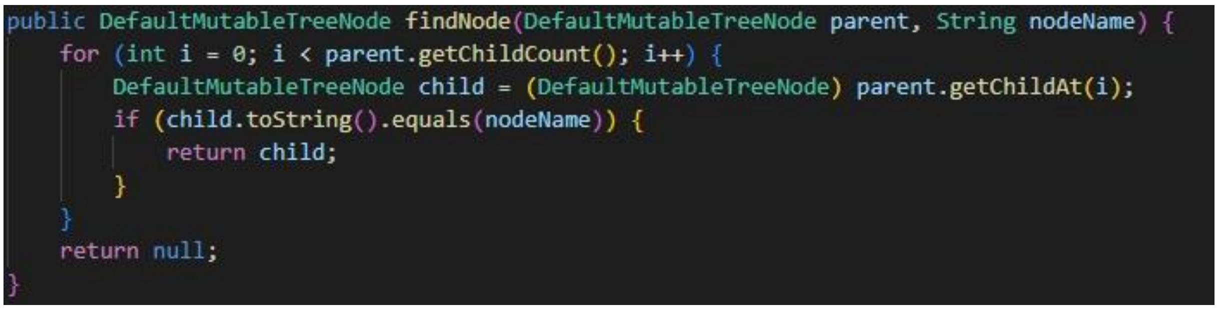
1. Constructor

- First, the root node is created after reading the tree as a 2-dimensional string. It is then neatly added to the tree, starting with the root node, checking if each string in the 2D string has been created before. I used a simple search method called "findNode" for checking.



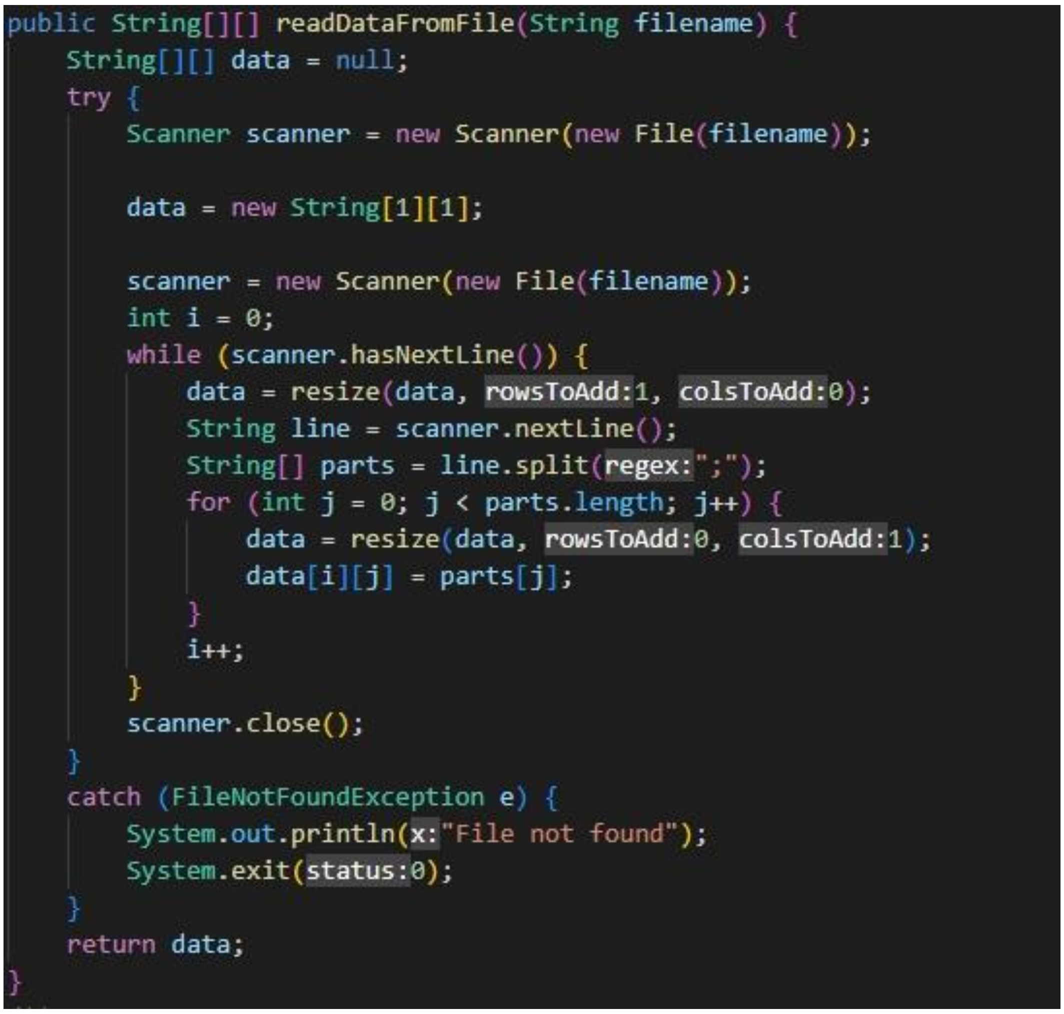
```java
public HW5_Tree() {
    data = readDataFromFile(filename:"homework5/tree.txt");

    root = new DefaultMutableTreeNode(userObject:"Root");

    for (int i = 0; i < data.length; i++) {
        DefaultMutableTreeNode node = root;
        for (int j = 0; j < data[i].length; j++) {
            DefaultMutableTreeNode checkNode = findNode(node,data[i][j]);
            if(checkNode == null && data[i][j] != null){
                DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(data[i][j]);
                node.add(newNode);
                node = newNode;
            }
            else if(checkNode != null){
                node = checkNode;
            }
        }
    }
    frame = new JFrame();
    tree = new JTree(root);
    for (int i = 0; i < tree.getRowCount(); i++) {
        tree.expandRow(i);
    }
    frame.add(tree);
    frame.setBounds(x:500, y:250, width:500, height:480);
    frame.setVisible(b:false);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

- This function searches all child nodes of the given parent node and tries to find the node corresponding to the given string.



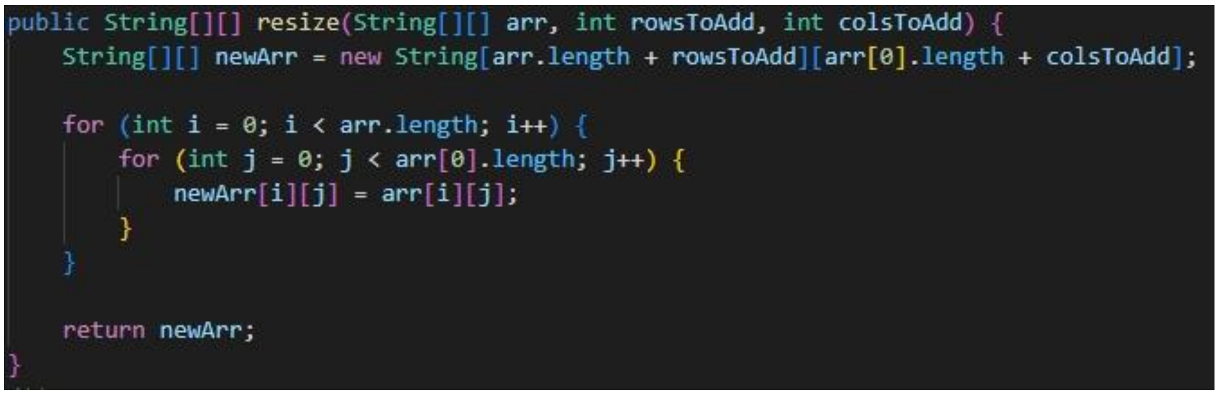
```java
public DefaultMutableTreeNode findNode(DefaultMutableTreeNode parent, String nodeName) {
    for (int i = 0; i < parent.getChildCount(); i++) {
        DefaultMutableTreeNode child = (DefaultMutableTreeNode) parent.getChildAt(i);
        if (child.toString().equals(nodeName)) {
            return child;
        }
    }
    return null;
}
```

- This method reads the tree expression in the file with the Scanner class and converts it to a dynamic 2D string. First, the file is opened with the path given as a parameter. Then the 2D string is unwrapped to hold a single string. While the file is being read line by line, one more row is added to the 2D string at the beginning of each line and the read lines are divided into desired parts by the "split()" method of the String class. The divided parts are added by increasing the columns of the 2D string by one.



```java
public String[][] readDataFromFile(String filename) {
    String[][] data = null;
    try {
        Scanner scanner = new Scanner(new File(filename));

        data = new String[1][1];

        scanner = new Scanner(new File(filename));
        int i = 0;
        while (scanner.hasNextLine()) {
            data = resize(data, rowsToAdd:1, colsToAdd:0);
            String line = scanner.nextLine();
            String[] parts = line.split(regex:";");
            for (int j = 0; j < parts.length; j++) {
                data = resize(data, rowsToAdd:0, colsToAdd:1);
                data[i][j] = parts[j];
            }
            i++;
        }
        scanner.close();
    }
    catch (FileNotFoundException e) {
        System.out.println(x:"File not found");
        System.exit(status:0);
    }
    return data;
}
```

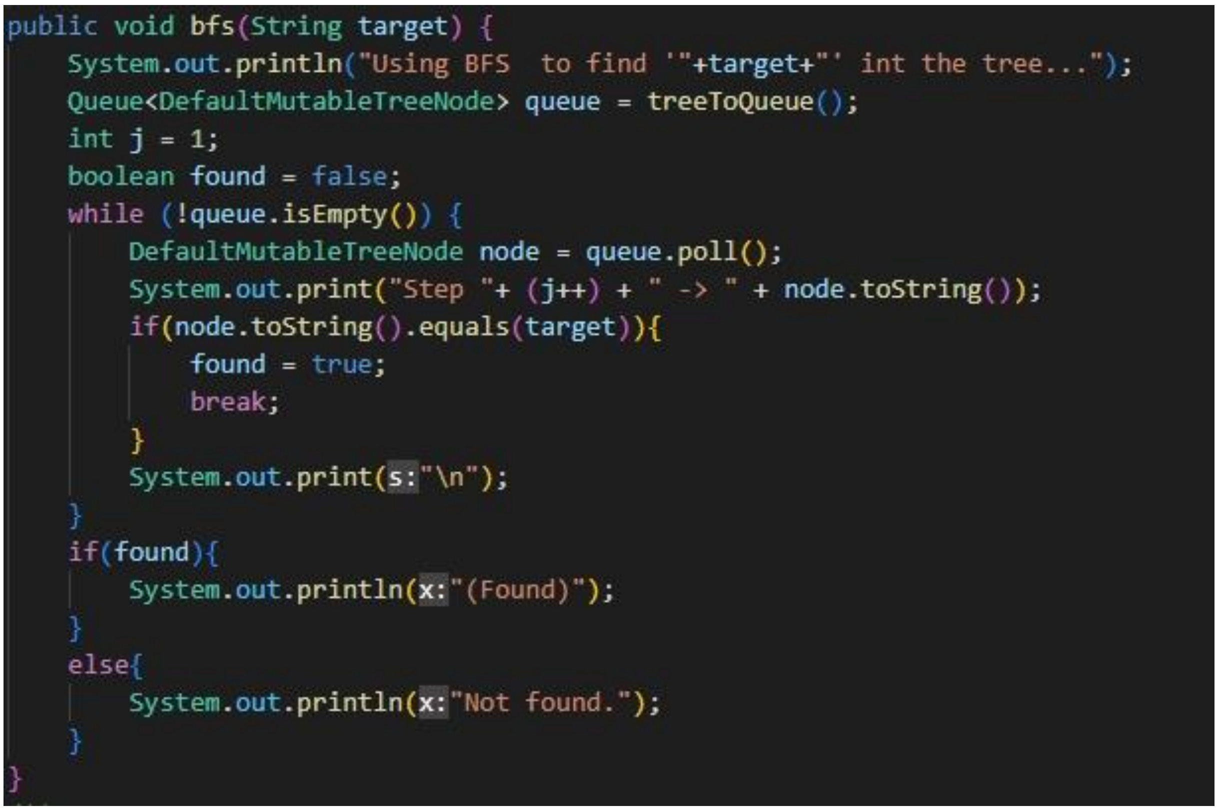
- 2D string size increase operations are done with the "resize" method.



```java
public String[][] resize(String[][] arr, int rowsToAdd, int colsToAdd) {
    String[][] newArr = new String[arr.length + rowsToAdd][arr[0].length + colsToAdd];

    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[0].length; j++) {
            newArr[i][j] = arr[i][j];
        }
    }

    return newArr;
}
```

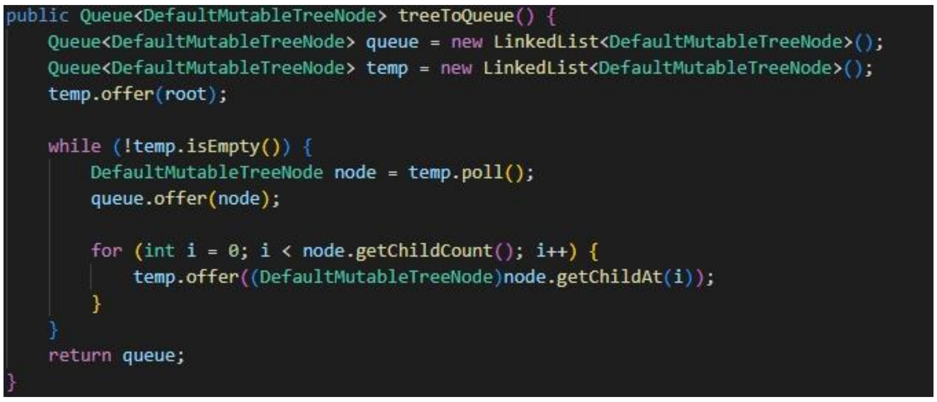
2. BFS

- I used Queue data structure to implement BFS algorithm. Because for BFS, starting with the root node, first the parent nodes and then the child nodes should be searched, and for this job, I created the "treeToQueue" method using the queue's FIFO principle. First, I put all the nodes in the tree into the queue, then I searched for the target node starting from the beginning by decreasing its size.



```java
public void bfs(String target) {
    System.out.println("Using BFS  to find '"+target+"' int the tree...");
    Queue<DefaultMutableTreeNode> queue = treeToQueue();
    int j = 1;
    boolean found = false;
    while (!queue.isEmpty()) {
        DefaultMutableTreeNode node = queue.poll();
        System.out.print("Step "+ (j++) + " -> " + node.toString());
        if(node.toString().equals(target)){
            found = true;
            break;
        }
        System.out.print(s:"\n");
    }
    if(found){
        System.out.println(x:"(Found)");
    }
    else{
        System.out.println(x:"Not found.");
    }
}
```

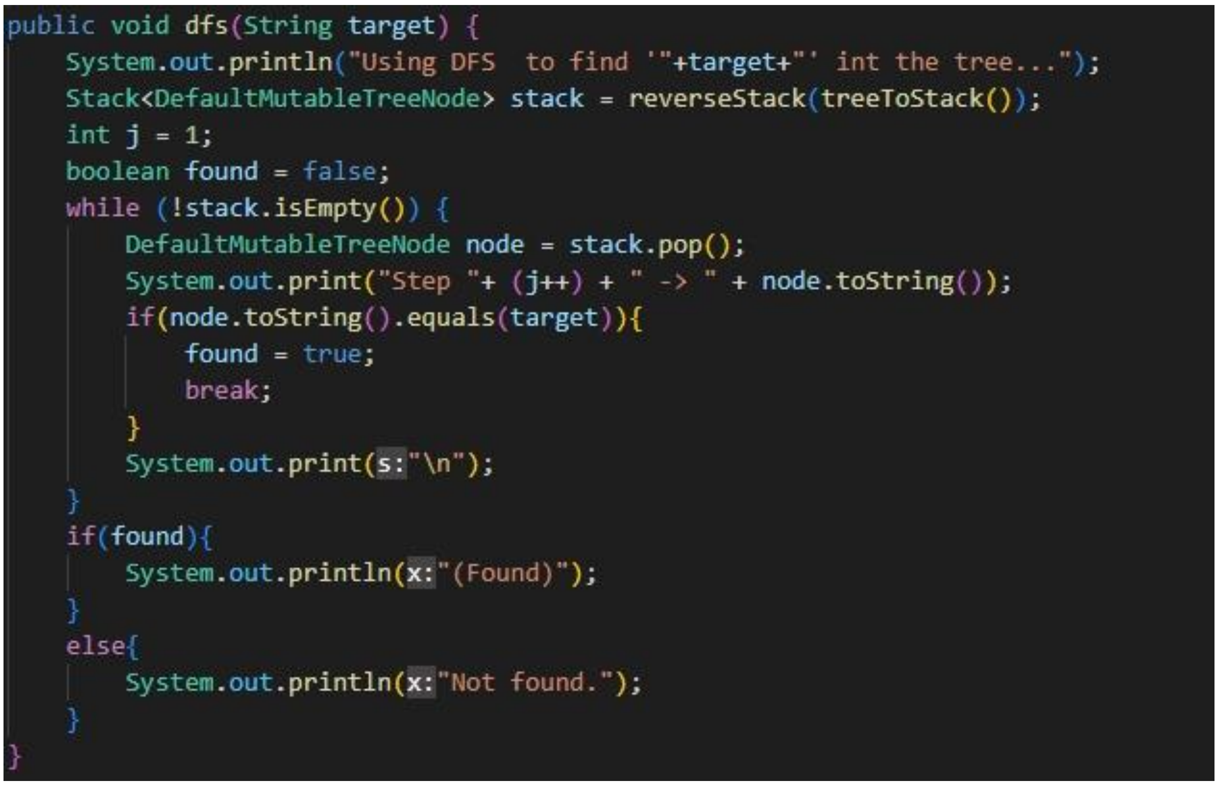
- This method begins by adding the root node of the tree to temp. Then, the algorithm enters a loop that runs while temp is not empty. During each iteration of the loop, the algorithm removes the first node from temp using the poll() method and adds it to the end of queue using the offer() method. The algorithm then examines each of the children of the node that was just removed from temp. For each child, the algorithm adds the child to the end of temp using the offer() method.



```java
public Queue<DefaultMutableTreeNode> treeToQueue() {
    Queue<DefaultMutableTreeNode> queue = new LinkedList<DefaultMutableTreeNode>();
    Queue<DefaultMutableTreeNode> temp = new LinkedList<DefaultMutableTreeNode>();
    temp.offer(root);

    while (!temp.isEmpty()) {
        DefaultMutableTreeNode node = temp.poll();
        queue.offer(node);

        for (int i = 0; i < node.getChildCount(); i++) {
            temp.offer((DefaultMutableTreeNode)node.getChildAt(i));
        }
    }
    return queue;
}
```

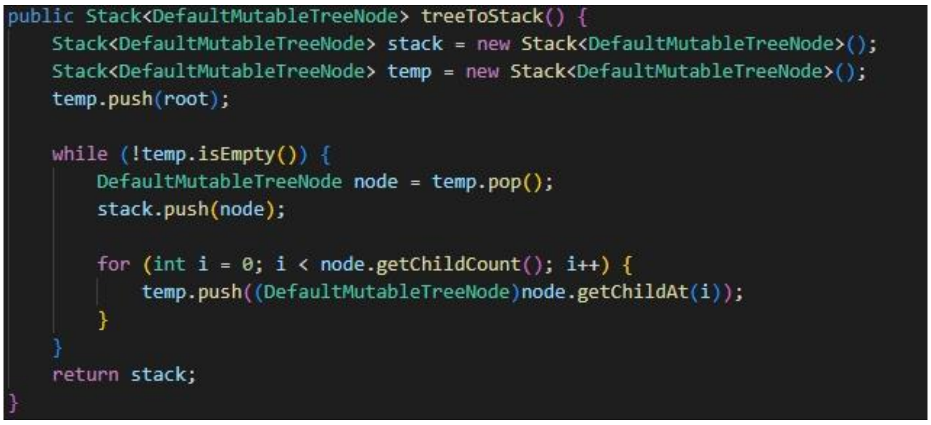
3. DFS

- I used Stack data structure to implement DFS algorithm. Because for DFS, it is necessary to start with root and continue with the last child node of the root and check the child nodes of those nodes(LIFO). When I applied this, I saw that the element at the top of the stack was the last element I needed to look at. To fix this, I reversed the stack that I obtained by using the "treeToStack" method. After this operations I search the target node by decreasing stack size.



```java
public void dfs(String target) {
    System.out.println("Using DFS  to find '"+target+"' int the tree...");
    Stack<DefaultMutableTreeNode> stack = reverseStack(treeToStack());
    int j = 1;
    boolean found = false;
    while (!stack.isEmpty()) {
        DefaultMutableTreeNode node = stack.pop();
        System.out.print("Step "+ (j++) + " -> " + node.toString());
        if(node.toString().equals(target)){
            found = true;
            break;
        }
        System.out.print(s:"\n");
    }
    if(found){
        System.out.println(x:"(Found)");
    }
    else{
        System.out.println(x:"Not found.");
    }
}
```

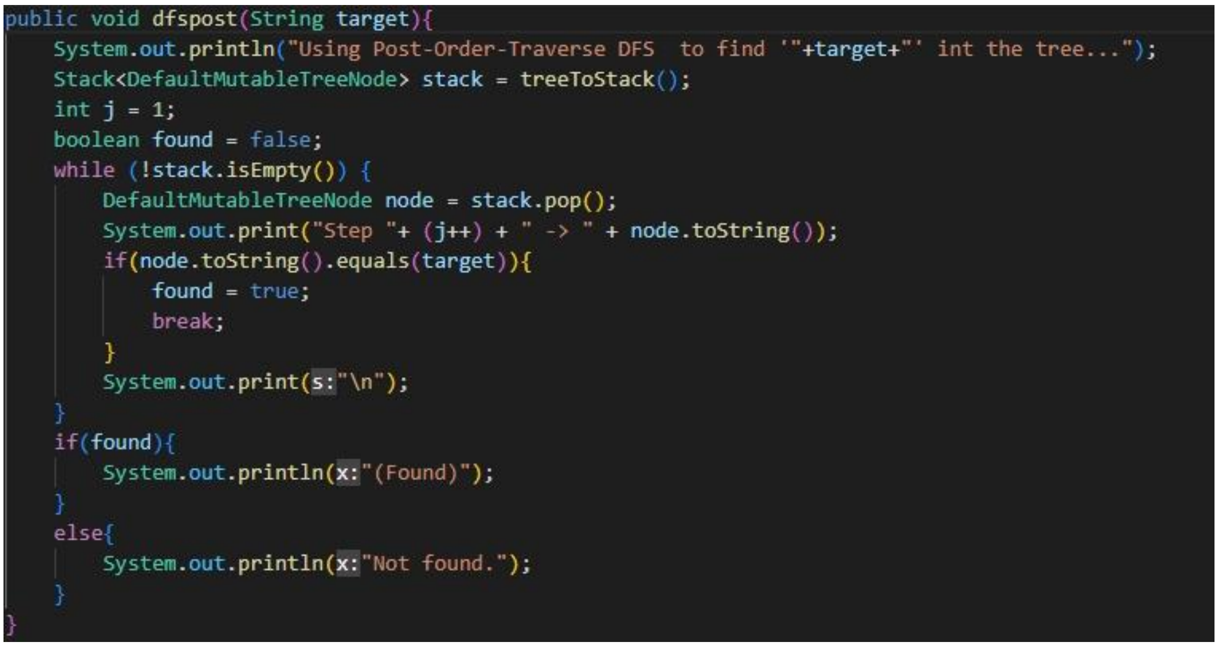
- This method begins by adding the root node of the tree to temp. Then, the algorithm enters a loop that runs while temp is not empty. During each iteration of the loop, the algorithm removes the first node from temp using the pop() method and adds it to the head of stack using the push() method. The algorithm then examines each of the children of the node that was just removed from temp. For each child, the algorithm adds the child to the head of temp using the push() method.



```java
public Stack<DefaultMutableTreeNode> treeToStack() {
    Stack<DefaultMutableTreeNode> stack = new Stack<DefaultMutableTreeNode>();
    Stack<DefaultMutableTreeNode> temp = new Stack<DefaultMutableTreeNode>();
    temp.push(root);

    while (!temp.isEmpty()) {
        DefaultMutableTreeNode node = temp.pop();
        stack.push(node);

        for (int i = 0; i < node.getChildCount(); i++) {
            temp.push((DefaultMutableTreeNode)node.getChildAt(i));
        }
    }
    return stack;
}
```

4. Post-Order DFS

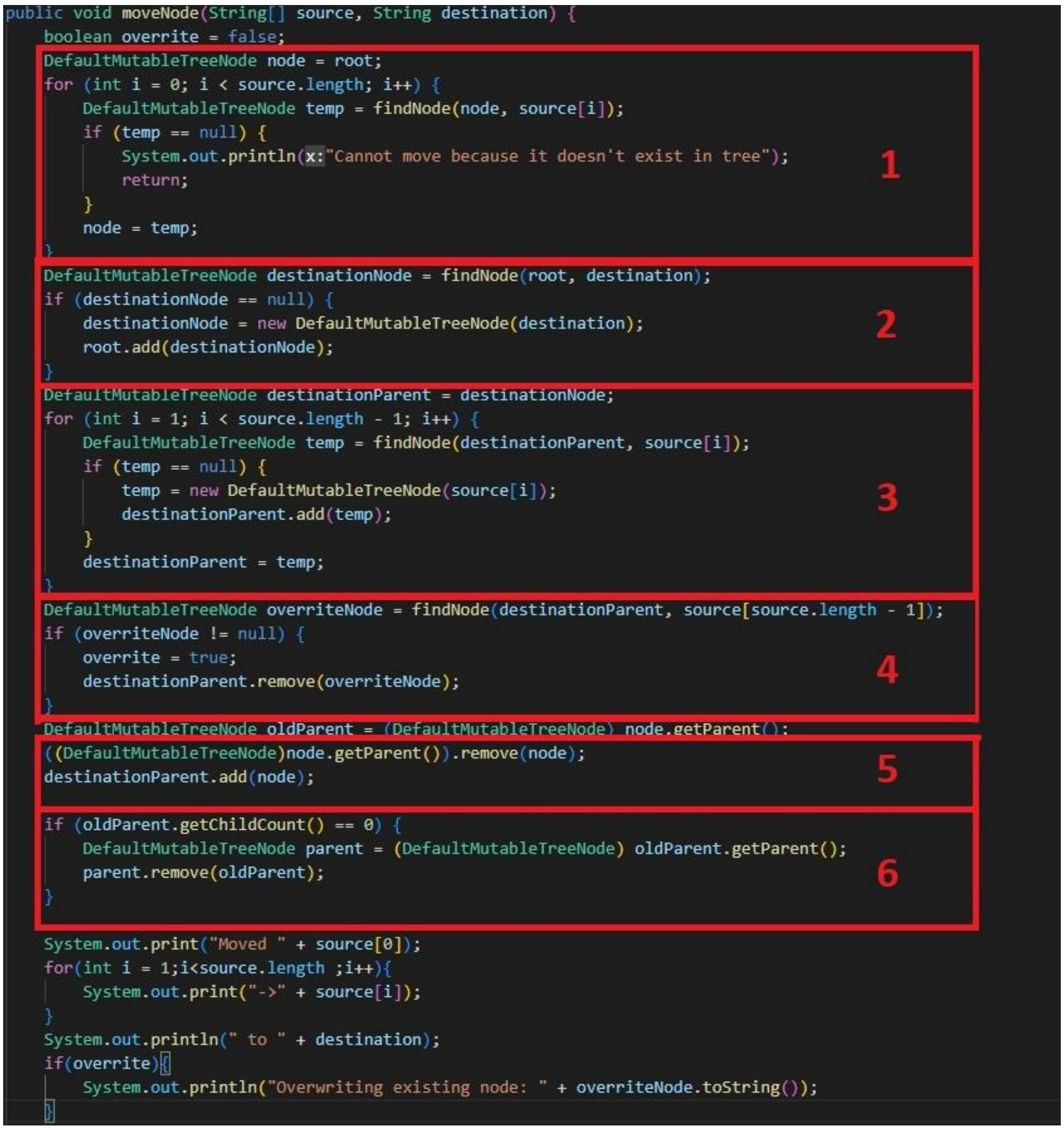
- In this method, I did exactly what I did in the previous DFS algorithm. However, this time I did not invert the tree that turned into a stack.



```java
public void dfspost(String target){
    System.out.println("Using Post-Order-Traverse DFS  to find '"+target+"' int the tree...");
    Stack<DefaultMutableTreeNode> stack = treeToStack();
    int j = 1;
    boolean found = false;
    while (!stack.isEmpty()) {
        DefaultMutableTreeNode node = stack.pop();
        System.out.print("Step "+ (j++) + " -> " + node.toString());
        if(node.toString().equals(target)){
            found = true;
            break;
        }
        System.out.print(s:"\n");
    }
    if(found){
        System.out.println(x:"(Found)");
    }
    else{
        System.out.println(x:"Not found.");
    }
}
```

5. Move Node

- I will explain this method by parts
  1 – In this part I check if source node exists. If its not prints error
  2 – In this part I check if destination node exists. If its not creates a new node and adds it to root
  3 – In this part, I check whether the parent nodes (Not year nodes) of the node we will move are at the destination. If not, it is created.
  4 – I am checking that the node we will be moving in this part is not already at the destination If there is, I delete the one at the destination and add a flag to print "Overritten" message later.
  5 – In this part I delete the node that we will be moving from its parent and adds it to destination node.
  6 – In this part, I check if the old parent node has still a child node. If its not I delete it.



```java
public void moveNode(String[] source, String destination) {
    boolean override = false;
    DefaultMutableTreeNode node = root;
    for (int i = 0; i < source.length; i++) {
        DefaultMutableTreeNode temp = findNode(node, source[i]);
        if (temp == null) {
            System.out.println(x:"Cannot move because it doesn't exist in tree");
            return;
        }
        node = temp;
    }
    DefaultMutableTreeNode destinationNode = findNode(root, destination);
    if (destinationNode == null) {
        destinationNode = new DefaultMutableTreeNode(destination);
        root.add(destinationNode);
    }
    DefaultMutableTreeNode destinationParent = destinationNode;
    for (int i = 1; i < source.length - 1; i++) {
        DefaultMutableTreeNode temp = findNode(destinationParent, source[i]);
        if (temp == null) {
            temp = new DefaultMutableTreeNode(source[i]);
            destinationParent.add(temp);
        }
        destinationParent = temp;
    }
    DefaultMutableTreeNode overriteNode = findNode(destinationParent, source[source.length - 1]);
    if (overriteNode != null) {
        override = true;
        destinationParent.remove(overriteNode);
    }
    DefaultMutableTreeNode oldParent = (DefaultMutableTreeNode) node.getParent();
    ((DefaultMutableTreeNode)node.getParent()).remove(node);
    destinationParent.add(node);

    if (oldParent.getChildCount() == 0) {
        DefaultMutableTreeNode parent = (DefaultMutableTreeNode) oldParent.getParent();
        parent.remove(oldParent);
    }

    System.out.print("Moved " + source[0]);
    for(int i = 1;i<source.length ;i++){
        System.out.print("->" + source[i]);
    }
    System.out.println(" to " + destination);
    if(override){
        System.out.println("Overwriting existing node: " + overriteNode.toString());
    }
```