CSE 222
Data Structures
Report

Yavuz Selim İkizler
1901042617

Code Design

General Class Structures

Class MyJtree:

I have created the MyJtree class in the program to build the JTree structure. This class holds DefaultMutableTreeNode root, Jframe frame, and int step variables as private.

Private variables:

```
private DefaultMutableTreeNode root;
private int step;
private JFrame frame;
```

The root variable represents the root node of the tree structure, and the step variable holds the current step as an integer in search algorithms. There is only one constructor in the program, which takes a string as a parameter. The public search methods are BFS_search, DFS_search, and DFS_traversal_search. In addition to these algorithms, there are helper methods. The move_node method is used to move nodes, and there are also helper methods for this. Below is a detailed explanation of all methods. This is the general class structure.

Class Main:

In the main class, I created a MyJtree object by passing the file name as a parameter and then called the visualize method. This will read the tree from the file and display it on the GUI by calling the visualize function. Then, I created a menu structure with 5 options, which call the main methods in the class. The methods are called in a while loop with a switch case structure, based on the option number entered by the user. The necessary strings are requested from the user in the methods to be called.

Options:

```
System.out.println("Please select an option:");
        System.out.println("1. Option 1 BFS");
        System.out.println("2. Option 2 DFS");
        System.out.println("3. Option 3 DFS Traversal");
        System.out.println("4. Move Node");
        System.out.println("5. Exit");
        System.out.print("Enter your choice: ");
```

MyJtree Class Methods

Constructure:

```
public MyJtree(String file_name)
```

The MyJtree constructor takes a string file name as a parameter. It initializes the step variable to 0 and creates a root node with the name "Root" using the DefaultMutableTreeNode class. It then creates a 2D dynamic array data to store the tree data read from the file. The data is read from the file using a Scanner and split into strings using ";" as the delimiter. The array is dynamically resized as new lines are read.

The constructor then loops through the data array and constructs the tree structure using a nested loop. The outer loop iterates over each row in the data array, while the inner loop iterates over each element in the current row. The add_node boolean variable is used to keep track of whether a new node needs to be created or an existing node needs to be updated.

For each element in the row, the constructor checks if there is an existing child node with the same name as the current element. If there is, it updates the parent node to the existing child node and sets the add_node flag to false. If there is no existing child node, it creates a new child node with the name of the current element and adds it to the parent node. The parent node is then set to the newly created child node

Visualize:

```
public void visualize
```

The visualize method creates a JTree object using the root node and a JScrollPane to allow for scrolling if the tree structure exceeds the size of the window. It then creates a JFrame object with the title "Tree Viewer", sets the default close operation to exit the application when the window is closed, and adds the JScrollPane containing the JTree object to the JFrame. The pack method is called on the JFrame to ensure that the size of the JFrame is set to fit the JTree object. Finally, the JFrame is set to visible, displaying the tree structure on the GUI.

BFS_search:

```
public void BFS_search(String search_element)
```

The BFS_search method performs a breadth-first search on the tree structure starting from the root node, searching for the specified search_element parameter. The method initializes a queue data structure to keep track of nodes to be visited during the search. The root node is added to the queue to start the search, and a flag is initialized to determine if the search_element has been found.

The method then enters a while loop that continues until either the queue is empty or the search_element is found. In each iteration of the loop, the head of the queue is removed and checked if it matches the search_element. If it matches, the flag is set to true and the method prints a message indicating the node has been found. Otherwise, the method prints a message indicating the current node being searched.

The method then increments the step_num counter and adds all the children of the current node to the queue for further exploration. If the search_element is not found after all nodes have been explored, the method prints a message indicating that it was not found.

DFS_search:

```
public void DFS_search(String search_element)
```

The search algorithm contains two methods, DFS_search and DFS_supporter. DFS_search is a public method that takes a search element as an argument and performs a depth-first search on the tree to find that element. It initializes a counter variable called "step" to 1 and prints the first node in the tree as the starting point of the search. It then calls the DFS_supporter method with the root node and the search element as arguments.

DFS_supporter is a support method and it is a private recursive method that takes a parent node and a search element as arguments. It first checks if the parent node has any children. If it does not, it returns false, indicating that the search element was not found. If it has children, it iterates over them in reverse order and for each child, it increments the step counter and checks if the child node's label matches the search element. If it does, it prints the node and returns true, indicating that the search element was found. If it does not match, it prints the node and recursively calls itself with the child node as the new parent node. If the search element is found in any of the child nodes, the method returns true; otherwise, it returns false.

Overall, the DFS_search method provides a way to perform a depth-first search on the tree, and the DFS_supporter method implements the actual search logic. The step counter is used to keep track of the progress of the search and print the visited nodes along with their order.

DFS_traversal_search:

main method:

```
 public void DFS_traversal_search(String search_element)
```

helper:

```
private boolean DFS_traversal_search_supporter(DefaultMutableTreeNode parent,
String search_element)
```

This code implements a depth-first search (DFS) algorithm to traverse the tree and search for a specific element. The DFS_traversal_search method takes in a search_element parameter and calls the DFS_traversal_search_supporter method with the root node of the tree.

The DFS_traversal_search_supporter method recursively traverses the tree in a depth-first manner,
This method recursively assigns the child node to the parent node until it reaches a leaf node. If a leaf node is reached, it checks whether the desired node has been reached, and returns true if it has been reached and false otherwise. After each recursive call, the value of the parent node is compared with the desired node, and if they match, true is returned. This method allows for checking the child nodes first, followed by the parent nodes, until the root node is reached.
During the traversal, the method also keeps track of the current step number, which starts at 0 and is incremented each time a node is visited. If the current node is not the root node, the method prints the current step number and the label of the node being visited.

Move_node:

```
        public void move_node(String node_string,String destination)
```

The provided code implements a method called move_node which moves a specified node from one location to another. The method takes two arguments: node_string which is a comma-separated string representing the path to the node to be moved, and destination which is the destination folder for the node.

The method starts by checking whether the destination argument is a valid integer representing a year value. If the argument is not valid, the method prints an error message and returns without performing any action.
Helper:IsInteger()

```
private boolean isInteger(String input)
```

If the destination argument is valid, the method converts the node_string argument to a queue of node names using the stringToQueue helper method. The method then initializes several variables such as search_node, temp, remove_node, destination_node, destQueue, print_queu, node_exist, node_exist2, and is_override to be used in the following steps.

Helper:StringToQueue()

```
  private  Queue<String> stringToQueue(String s)
```

The method then checks whether the node to be moved has at least two layers. If not, the method prints an error message and returns without performing any action.

The method then searches for the destination node in the tree. If the destination node does not exist, the method creates it and adds it to the tree.

"destination_node" represents the parent node of the node to be added.
The method then starts moving the node from its current location to the destination. It does this by iterating over the elements in the destQueue queue which represents the path to the destination node. For each element, the method checks whether the element already exists as a child node of the current destination node. If it does, the method sets the current destination node to be that child node. Otherwise, the method creates a new node with that element as the name and adds it as a child node of the current destination node.Then assign the child node to the destination node.

After moving the node to the destination, the method checks whether the destination node already has a child node with the same name as the last element in the destQueue queue. If it does, the method sets the is_override flag to true.

The method then starts moving the node from its current location to the destination node. It does this by iterating over the elements in the node_queu queue which represents the path to the node to be moved. "search_node" represents the parent of the node to be moved. At each step, the nodes in "node_queue" are compared with the child nodes of "search_node". If there is a match, the matching child node is assigned to "search_node" until an unmatched node is found or there are no more elements left in "node_queue". If an unmatched node is found, it means that the node to be moved is not present in the tree. If there are no more elements left in "node_queue", it means that the node to be moved has been reached.

After moving the node to the destination, the method checks whether the node_queu queue is empty. If it is not empty, the method prints an error message indicating that the node do not exist in the tree. If the is_override flag is set to true, the method prints a message indicating that the node has been overwritten. If the queue is empty and the is_override flag is not set, the method adds the remove_node to the destination_node and updates the visual representation of the tree.

NOTE:
If there are no remaining children in the parent node after a node is removed, the deletion process is not performed to that node. I did not include that.