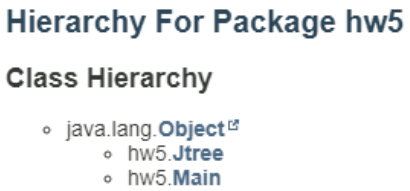
GTU Department of Computer Engineering
CSE 222 - Homework 5- Report

Doğukan Taştan 1901042627

General Structure



The structure consists of the Main class that reads the file into a 2d array and prints the user menu, and the Jtree class where all other operations are performed.

Explain Of Jtree

It accepts a 2D string array in its constructor, this array is the array to which the values read from the file in the mainde are added. Then it first sends the array to the fileArrayToTree function, which converts the array into tree nodes and adds them to the tree.

The childNode in the fileArrayToTree function runs the findNode function which checks if the given values already exist.

```
DefaultMutableTreeNode childNode = findNode(parentNode, itemName);
```

Then, if the childNode value is null, which means that there is no such value, the addNode function is executed to add the childNode, and our new parentNode is changed to childNode since the childNode is now added.

```
if (childNode == null)
{childNode = addNode(parentNode, itemName);}
parentNode = childNode;
```

```
public void writeTree()
```
function sets the output to be printed to the screen using swing Jtree

```
public void exit(){ f.dispose();}
```
This function is used to turn off the f label on the screen at exit time

Move Function

```
/** Our function that performs move and delete operations */
public void move(String[] src,String[] dest) {
     DefaultMutableTreeNode srcNode = root;
     for (String itemName : src) {
         DefaultMutableTreeNode childNode = findNode(srcNode, itemName); /** find source location */
         if (childNode == null) {
             System.out.print("Cannot move ");
             for(int i=0;i<src.length;i++)
                 System.out.print(src[i]+(i!=src.length-1 ? "->":""));
             System.out.print(" because it doesn't exist in the tree .");
             return;
         }
         srcNode = childNode;
```

In this first part of the Move function, we try to find the source by navigating through the incoming src string array. If it is not found, we end the function by suppressing the necessary error messages.

```
    }
    DefaultMutableTreeNode destNode = root;
    for (String itemName : dest) {
        DefaultMutableTreeNode childNode = findNode(destNode, itemName);  /** find destination location */
        if (childNode == null) {
            childNode = addNode(destNode, itemName);
        }
        destNode = childNode;
    }
    /** so far we have found the file to copy and where to copy it */
```

Then we do the same for the destination location. Due to the design of the function, we can do the migration even if the destination folder is not only the year but also other locations.

```
/** @param list We kept this list of names in the section will be moved */
LinkedList<String[]> list = new LinkedList<String[]>();
/** @param stack Stack used for additional Dfs search operation */
Stack<DefaultMutableTreeNode> stack = new Stack<>();
stack.add(srcNode);
/**This value is initially 2 because we want to include the folder 1 above in
int initialLevel=2;
while (!stack.isEmpty()) {
    DefaultMutableTreeNode currentNode = stack.pop();

    String[] currentpath= new String[currentNode.getLevel()-initialLevel+1]; //
    DefaultMutableTreeNode iter = currentNode;
    int i = currentNode.getLevel()-initialLevel;

    while(iter.getParent() != null && i > -1){ // adding the name of the files
        currentpath[i] = iter.toString();
        iter = (DefaultMutableTreeNode) iter.getParent();
        i--;
    }
    list.add(currentpath);
    for (int k = 0; k < currentNode.getChildCount(); k++) { // It's part of the
        stack.push((DefaultMutableTreeNode) currentNode.getChildAt(k));
    }
}
```

In this section, we keep the names with an auxiliary linkedlist, I would like to emphasize that I do not keep Node. Then we start by assigning SrcNode to the stack for the search operation. initialLevel is set to 2 because we want to copy what we are moving with its parent folder. We pop the item in the stack to the currentNode, set the currentpath string array. In our loop we assign the parents to our path array by setting how much depth is above with the i value. The bottom part is part of our search algorithm that pushes the children to the stack.

```
        for (Object location : list.toArray()) {           // Adds to the destination by
                DefaultMutableTreeNode parentNode = destNode;
                for (String itemName : (String[])location) {
                    DefaultMutableTreeNode childNode = findNode(parentNode, itemName);
                    if (childNode == null) {
                        childNode = addNode(parentNode, itemName);
                    }
                    else {
                        DefaultMutableTreeNode iter = childNode;
                         while(iter.getParent() != null ){ // adding the name of the files
                             System.out.print(iter.toString()+"->");
                             iter = (DefaultMutableTreeNode) iter.getParent();
                         }
                        System.out.println("has been overwritten.");
                    }
                    parentNode = childNode;
                }
            }

        DefaultMutableTreeNode parentNode=(DefaultMutableTreeNode) srcNode.getParent();
        srcNode.removeFromParent();
        while(parentNode.getChildCount() == 0) {
            DefaultMutableTreeNode temp=(DefaultMutableTreeNode) parentNode.getParent();
            parentNode.removeFromParent();
            parentNode=temp;
        }
        /*for(String[] x : list)
            System.out.println(Arrays.toString(x));*/
    }
```

We are doing the insertion operations in the target node, here we first check the similar name and then add it, if there is an item with the same name, we print the overwritten error on the screen. At the bottom, while deleting, if the folder is empty, we ensure that the folder is deleted.

Search Functions

```java
public void bfsSearch(String target) {
    /** While doing bfs search, we print what we travel by throwing it to Queue */
    Queue<DefaultMutableTreeNode> queue = new LinkedList<>();
    queue.add(root);
    int step = 0;

    while (!queue.isEmpty()) {
        /** remove the existing node from the queue */
        DefaultMutableTreeNode currentNode = queue.poll();
        step++;

        System.out.println("Step " + step + ": Checking " + currentNode.getUserObject().toString());

        if (currentNode.getUserObject().toString().equals(target)) {
            System.out.println("Found " + target + " in step " + step);
            return;
        }
        /** we add the children of the existing node to the queue */
        for (int i = 0; i < currentNode.getChildCount(); i++) {
            queue.add((DefaultMutableTreeNode) currentNode.getChildAt(i));
        }
    }
    System.out.println(target + " not found in the tree.");
}
```

Bfs search function uses Queue structure while searching. If the queue is not empty, it adds to the currentNode by polling, then increases the step value and gives output to the screen, then it is checked whether it is found or not. Then, based on the number of children of the CurrentNode, the children of the CurrentNode are added to the queue in a loop and the process continues like this.Queue structure adapts to our BFS search algorithm since it will be first in first out.

```java
public void dfsSearch(String target) {
    /** The stack we use for dfs search */
    Stack<DefaultMutableTreeNode> stack = new Stack<>();
    stack.push(root);
    int step = 0;

    while (!stack.isEmpty()) {
        DefaultMutableTreeNode currentNode = stack.pop();
        step++;

        System.out.println("Step " + step + ": Checking " + currentNode.getUserObject().toString());

        if (currentNode.getUserObject().toString().equals(target)) {
            System.out.println("Found " + target + " in step " + step);
            return;
        }

        for (int i = 0; i <=currentNode.getChildCount() - 1 ; i++) {
            stack.push((DefaultMutableTreeNode) currentNode.getChildAt(i));
        }
    }
    System.out.println(target + " not found in the tree.");
}
```

In our Dfs search algorithm, we did very similar operations to Bfs, only we used the Stack data structure instead of Queue.

```java
    /** In the recuirsive structure I split it into 2 functions to be able to use it by taking only String  */
    public void postOrderTraversalSearch(String target) {
        int[] step = {0};
        if (postOrderTraversal(root, target, step)) {
            System.out.println("Found " + target + " in step " + step[0]);
        } else {
            System.out.println(target + " not found in the tree.");
        }
    }
    private boolean postOrderTraversal(DefaultMutableTreeNode currentNode, String target, int[] step) {
        for (int i = 0; i < currentNode.getChildCount(); i++) {
            if (postOrderTraversal((DefaultMutableTreeNode) currentNode.getChildAt(i), target, step)) {
                return true;
            }
        }
        step[0]++;
        System.out.println("Step " + step[0] + ": Checking " + currentNode.getUserObject().toString());
        return currentNode.getUserObject().toString().equals(target);
    }
}
```

Since our Post order Traversal function cannot work by taking only String due to its Recuirsive structure, I divided this function into 2 parts. The first part takes only String and then runs our recuirsive function in an if statement and prints the results on the screen according to the value returned by our Recuirsive function. The other part sends the children of the CurrentNode back to itself using the recuirsive structure, and at the bottom, the step value increases and then checks are made.