Daniel Pulido-Alaniz
7/18/2025

Report

Program Information

My program simulates managing/navigating a file system through the terminal in a similar manner to how it is done in Linux. The program was written entirely by me, and I did not use any tutorials or AI. I implemented simplified versions of various Linux commands like cd, ls, pwd, mkdir, touch, find, clear, delete, and help. One way I made a simplified version is that you can't cd through multiple directories by giving a parameter like 'home/work/data'. My own addition to it was implementing a stats command that counts the number of files, folder, and depth of the entire file system. Although not entirely similar to Linux, it can be used in a limited fashion to practice some Linux commands commonly used to navigate and manage folders and files.

Algorithms

One algorithm that I coded for my program is the merge sort algorithm. This was used for a '-s' that I added to my ls command. It would sort the files and folders in alphabetical order, but it would not change their order permanently. I would be sorting the names, so I processed them as strings and used .compareTo() for comparison. I broke down the merge sort into two different functions. The mergeSort function recursively split the array in half, and the merge function merged arrays back into order. The complexity of this algorithm is O(nlogn). The n part comes from the merging which takes linear time. The log(n) comes from continually splitting the arrays in half.

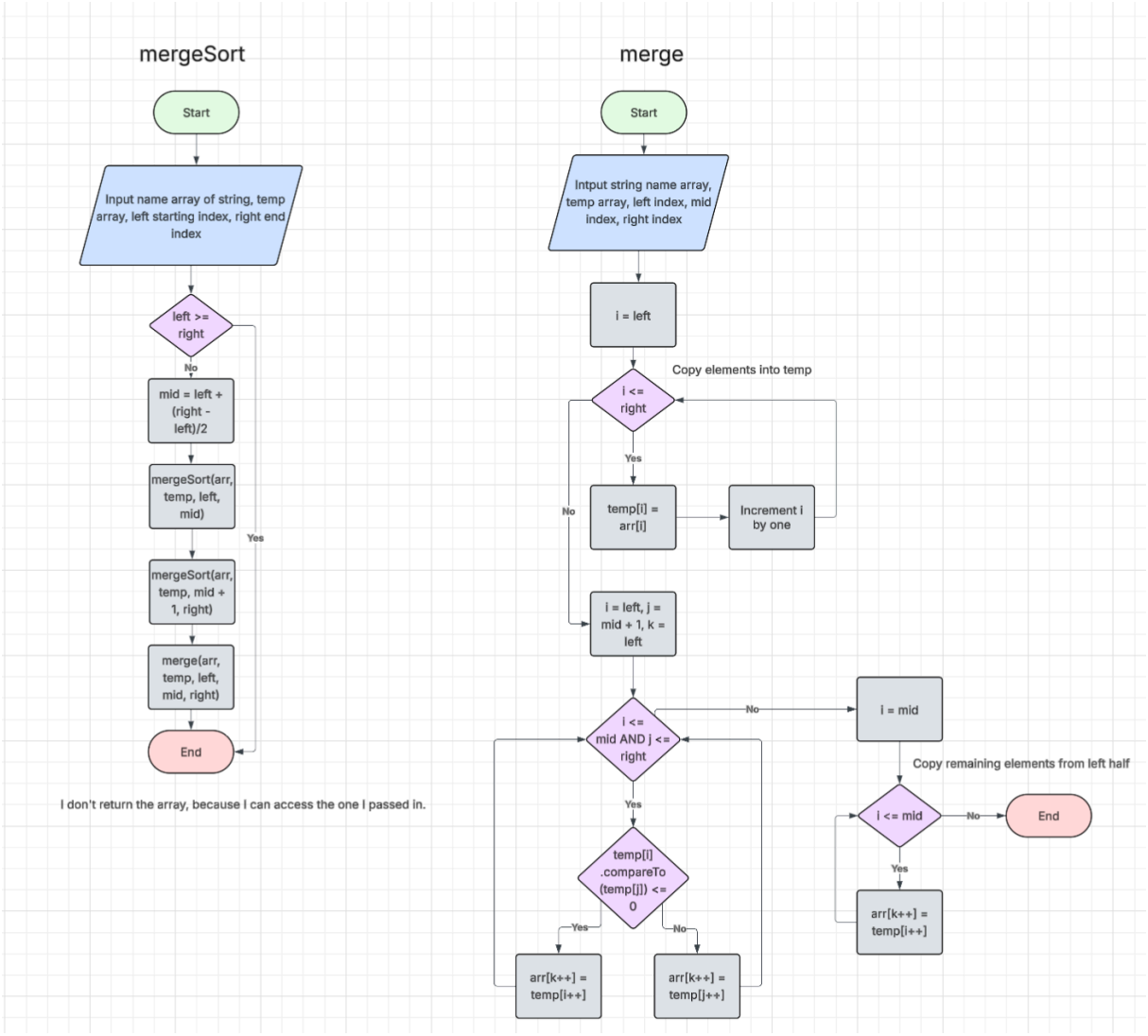
Code Snapshots for Merge Sort:

```java
// Recursive Merge Sort for string arrays
private void mergeSort(String[] arr, String[] temp, int left, int right){
    if (left >= right){
        return; // Base case: one element
    }

    int mid = left + (right - left) / 2;
    mergeSort(arr, temp, left, mid);  // Sort left half
    mergeSort(arr, temp, mid + 1, right); // Sort right half
    merge(arr, temp, left, mid, right);  // Merge sorted halves
}
```

```java
// Merge two sorted halves of the array
private void merge(String[] arr, String[] temp, int left, int mid, int right){
    // Copy elements into temp for merging
    for (int i = left; i <= right; i++){
        temp[i] = arr[i];
    }

    int i = left;
    int j = mid + 1;
    int k = left;

    // Compare and place elements back into arr in sorted order
    while (i <= mid && j <= right){
        if (temp[i].compareTo(temp[j]) <= 0){
            arr[k++] = temp[i++]; // Take from left
        } else {
            arr[k++] = temp[j++]; // Take from right
        }
    }

    // Copy any remaining elements from left half
    while (i <= mid){
        arr[k++] = temp[i++];
    }
}
```

Daniel Pulido-Alaniz
7/18/2025

Flowchart for Merge Sort:



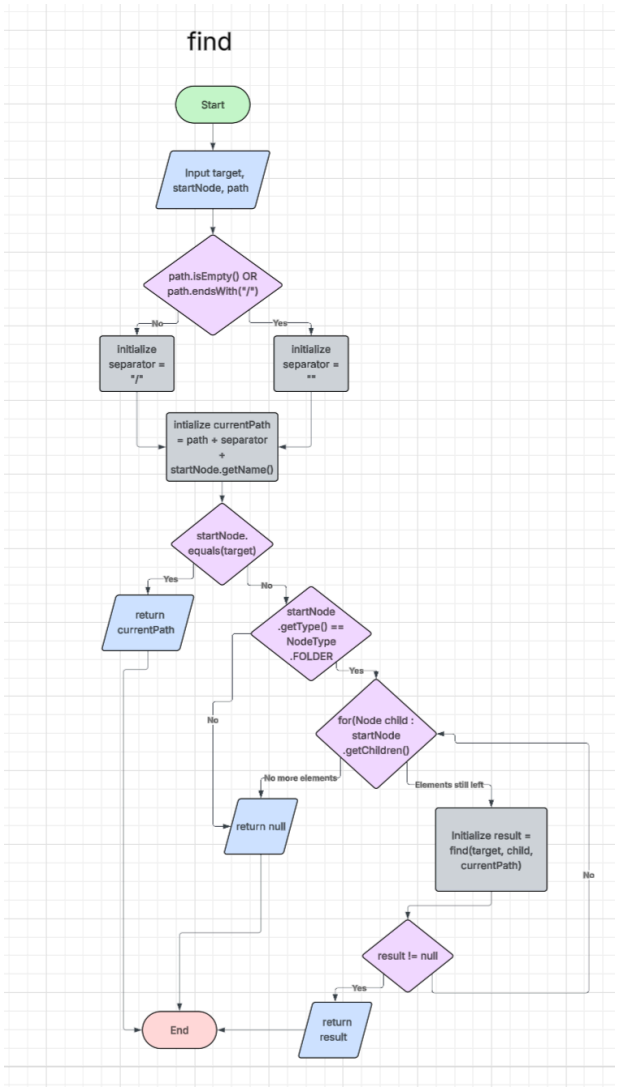
The next algorithm that I implemented was a depth first search algorithm. I used this to find a node that matches the name that the user passes. If the file/folder is found, the path to it is displayed to the user. My file system is represented as a tree, so I recursively search through each folder for any potential match. Since it's a depth first search, I go as deep as I can before backtracking and checking other children. I pass the path along each call to keep track of where I am at for when I need to return it. I also check whether a node is of type file or folder since file nodes don’t have children. If no file/folder matches, the function returns null. The time complexity for this algorithm is O(n). This is because in the worst case scenario, the algorithm will go through every file and folder.

Code Snapshot for Depth First Search

Daniel Pulido-Alaniz
7/18/2025

```java
private String find(FileSystemNode target, FileSystemNode startNode, String path){
    String separator = path.isEmpty() || path.endsWith(suffix:"/") ? "" : "/";
    String currentPath = path + separator + startNode.getName();

    if (startNode.equals(target)){
        return currentPath; // Target found
    }

    // Recurse into folders
    if (startNode.getType() == NodeType.FOLDER){
        for(FileSystemNode child : startNode.getChildren()){
            String result = find(target, child, currentPath);
            if (result != null){
                return result; // Found in subtree
            }
        }
    }

    return null; // Not found
}
```

Flowchart for Depth First Search



Daniel Pulido-Alaniz
7/18/2025

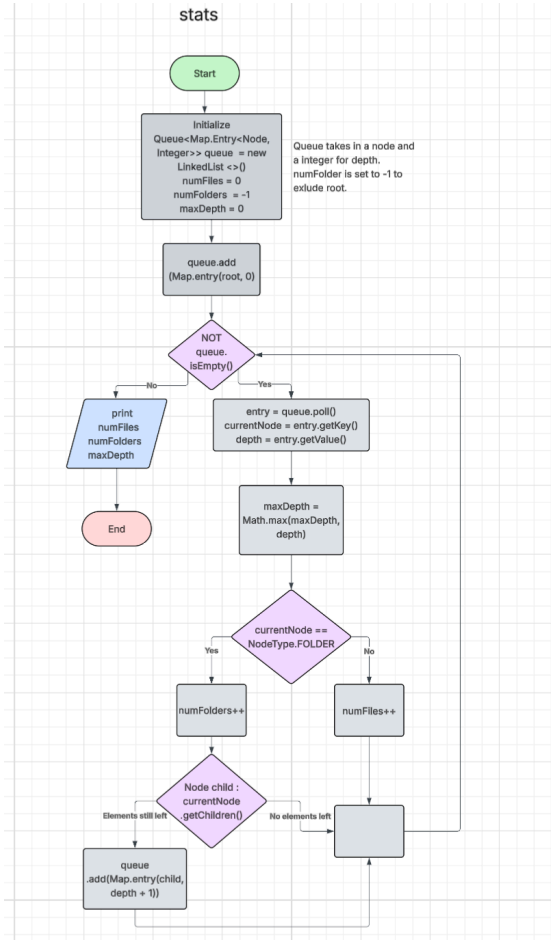
The third algorithm that I implemented was an iterative breadth first traversal for my stats command. The stats command counts the total number of files and folders in the system. It also displays the maximum depth of the file system. To implement this, I used a queue to keep track of the nodes that I still had to process. Everytime I accessed a folder, I pushed all its children to the queue. Each queue entry also contained an integer to keep track of the depth, and I updated the max depth every time I encountered a bigger depth. A while loop was used to keep iterating while the queue still had nodes to go through. The time complexity of this algorithm is O(n) because it always goes through each node.

Code Snapshot for Breadth First Traversal

```java
public void stats(){
    Queue<Map.Entry<FileSystemNode, Integer>> queue = new LinkedList<>();

    int numFiles = 0;
    int numFolders = -1; // Start at -1 to exclude root
    int maxDepth = 0;

    queue.add(Map.entry(path.firstElement(), v:0)); // Start from root at depth 0

    while(!queue.isEmpty()){
        Map.Entry<FileSystemNode, Integer> entry = queue.poll();
        FileSystemNode current = entry.getKey();
        int depth = entry.getValue();

        maxDepth = Math.max(maxDepth, depth); // Track deepest level

        if (current.getType() == NodeType.FOLDER){
            numFolders++;
            for(FileSystemNode child : current.getChildren()){
                queue.add(Map.entry(child, depth + 1)); // Enqueue children with updated depth
            }
        } else {
            numFiles++;
        }
    }

    System.out.println(x:"Statistics:");
    System.out.println("Number of Files: " + numFiles);
    System.out.println("Number of Folders: " + numFolders);
    System.out.println("Maximum Depth: " + maxDepth);
}
```

Daniel Pulido-Alaniz
7/18/2025

Flowchart for Breadth First Traversal



Data Structures Used

I used several data structures to implement the functionality of my file manager. First, I used a stack to keep track of the current directory path. Each time the user navigates into a folder using the cd command, that folder node is pushed onto the stack, representing moving deeper into the file system. When the user types ‘cd ..’ to go up one directory, the top element is popped from the stack, which brings the user back one level towards the root directory. This approach was chosen because it accurately models the hierarchical nature of file systems and provides efficient operations for adding and removing directories from the path.

Next, I utilized a HashMap in two different ways. The first HashMap stores the mapping between command names and their corresponding command objects. When the user enters a command, I split the input string, take the first word, and use the HashMap to quickly retrieve the appropriate command object to execute. The second HashMap is used for the help command. This map associates each command with its help description, allowing the program to instantly provide relevant usage instructions when the user types ‘help <command>’. I used HashMaps

because they offer constant-time lookup, making the help command quick even if I added many commands.

Finally, I used an ArrayList in the FileSystemNode class to store the children of each folder. Since a folder can contain an unknown and potentially large number of files and subfolders, an ArrayList provides a flexible and efficient way to manage this dynamic collection. You can easily traverse through the elements which is necessary for various commands like ls and cd.

One Step in the Design Process and Two Errors I Encountered

One step during the development process was deciding how to handle command execution from user input. At first, I considered using a long series of if-else statements or switch statements to check which command the user entered. However, I saw the opportunity to make the system more scalable and maintainable by using the Command design pattern. This approach allowed me to create a separate class for each command, encapsulating its behavior inside the execute method. By doing this, I made the system highly extensible; adding a new command only requires writing a new class and adding it to the command map, without modifying any existing code. This design choice simplified future improvements and adhered to object-oriented principles.

One issue I encountered was with the touch command, where I initially forgot to validate file names for extensions. Without this check, users could create files without proper formats, which was unrealistic. It also made it difficult for the user to distinguish between folder and file. I resolved this by splitting the file name on the period character and ensuring it had at least two parts.

Another error occurred in the find command, where I repeatedly struggled to prevent duplicate slashes from appearing when building paths during recursion. The issue happened because I was always appending a backslash between segments without checking whether the existing path already ended with one. To fix this, I added a conditional that checks if the path is empty or already ends with a backslash before deciding whether to append another slash.

Changes for the Next Version

In the next version, I would improve the cd command to allow users to navigate directly to a full path like /home/user/documents instead of moving one folder at a time. This would make navigation faster and more user-friendly. Additionally, I would add features like support for relative paths and autocomplete suggestions for folder and file names. These enhancements would make the file manager feel more like a real-world terminal experience.