Design and Analysis of a Virtual File System Using N-ary Trees with a Depth-First Search Traversal Algorithm

Kurt Mikhael Purba – 13524065
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10, Kota Bandung, 40132, Indonesia
E-mail: 13524065@std.stei.itb.ac.id, kurtmikhael123@gmail.com

***Abstract*—File system is a component in an operating system that functions to organize data. This paper will try to implement a tree to represent the virtual file system data structure. The programming language used to realize this idea is the python programming language which has an object oriented programming paradigm. By using this programming language, each node can have its own features. The implementation of the DFS algorithm is made to help search for a file and directory. The analysis section demonstrates the system's operational correctness and evaluates the performance of its core functions. The time complexity for the DFS-based search is shown to be linear with respect to the total number of files and directories in the system. This work serves as a practical demonstration of applying fundamental tree theory and graph traversal algorithms to solve a classic computer science problem.**

***Keywords*—graph,tree,depth-first search,node.**

I. Introduction

The file system is the most fundamental component in modern computing. It is responsible for organizing, storing, and managing data with a hierarchical and logical structure, so that users and applications can access and manipulate data efficiently. This hierarchical structure can be represented as a tree. Each directory and file can be represented as a node.

Although users interact with file systems every day, the process of modeling and implementing this structure from the ground up involves core concepts from discrete mathematics and computer science. One of the most common operational challenges is how to efficiently search for files in a directory structure that can be very large and deep. Without a systematic algorithm, the search process will be slow and unreliable.

This paper proposes the modeling and implementation of a simple virtual file system using an N-ary tree data structure. The N-ary tree was chosen because a directory can have an unlimited number of children, unlike a binary tree. To overcome the challenge of file searching, the Depth First Searching (DFS) search algorithm will be implemented. This algorithm was chosen because it intuitively mimics the way a deep search works in a directory hierarchy.

Through this implementation, the paper aims to practically demonstrate the application of tree theory in solving real problems in the field of informatics. In addition, a performance analysis will be carried out on the implemented DFS algorithm to provide an overview of its efficiency in the context of file systems [1].

II. Theoritical Basis

A. Graph

A graph G is defined as G = (V, E), where V is a non-empty set of vertices,defined as V = { v_1 , v_2 , ... , v_n }, the set V can't be empty, meaning that the graph can't contain no vertices , and E is an edge connecting a pair of vertices ,defined as E = {e_1 , e_2 , ... , e_n }, the set E can be empty, meaning the graph can't contain a single edge [2].

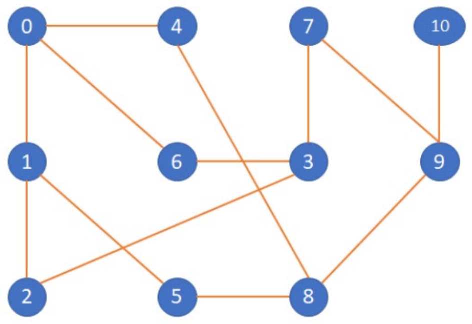


Fig 2.1 Example of Graph [2]

Based on whether there are rings or double edges in the graph, then graphs are classified into two types [2]:

a. *Simple Graph*

Simple graph is a graph that does not contain loops or multiple edges.

b. *Unsimple Graph*

Unsimple graph is a graph that contains multiple edges or rings.

Non-simple graphs are further divided into:

- Multi-graph: Graph containing multiple edges
- Pseudo-graph: Graph containing loop edges

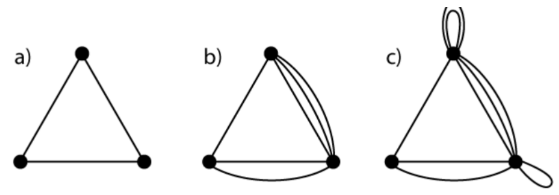


Fig 2.2 a) Simple Graph, b) Multi-Graph, and c) Pseudo-Graph [2]

B. Tree

A tree is an undirected graph that is connected and does not contain circuits (cycles).

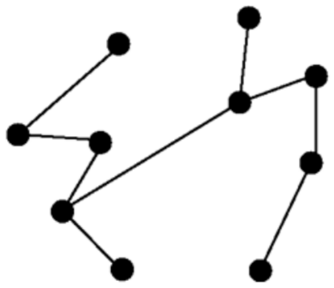


Fig 2.3 Example of Tree [3]

A forest is a collection of disjoint trees or a disconnected graph that contains no circuits. Each component in the connected graph is a tree.

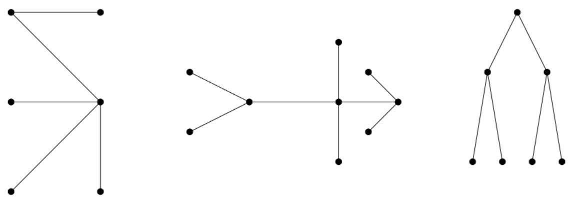


Fig 2.4 Examples of Forest [3]

A tree has some basic properties. Let G = (V, E) be a simple undirected graph and the number of vertices is n. Then, all the following statements are equivalent [3]:

- G is a tree.
- Every pair of vertices in G is connected by a single path.
- G is connected and has m = n – 1 edges.
- G doesn't contain circuits and has m = n – 1 edges.
- G doesn't contain circuits and adding one edge to the graph will create only one circuit.
- G is connected and all its edges are bridges.

C. Depth-First Searching

Depth-First Search (DFS) is a fundamental algorithm used for traversing or searching graph and tree data structures. As the name implies, the main working principle of DFS is to explore depth first. This algorithm will explore one branch of the data structure as deep as possible until it reaches the end point (the furthest node) before returning (backtracking) to explore other branches that have not been visited.

This "depth-first" philosophy fundamentally distinguishes it from other search algorithms such as Breadth-First Search (BFS), which explores in a wide layer by layer. The most intuitive analogy to understand DFS is the way someone solves a maze: we will choose one path and continue to follow it until we find a dead end or reach the goal. If the path is a dead end, we will backtrack to the last intersection and try another path that has never been passed. This "depth-ward motion" search movement is very suitable for handling hierarchical structures.

This suitability makes DFS a very relevant choice in the context of virtual file systems. The hierarchical structure of a file system, where a directory can contain multiple subdirectories, is a perfect real-world model for a DFS search strategy. The algorithm naturally mimics the way a user or program traverses a deep directory path before returning to explore a lower directory level.

Conceptually, the DFS process is inherently recursive. The core operation of DFS is from the current node, visit one of its unvisited neighbors. This newly visited neighbor then becomes the current node, and the same operation is repeated. This pattern of repeating the same operation over smaller parts of the problem (i.e., the subgraph rooted at a neighbor node) is the definition of recursion. When a path has been exhausted, the algorithm must return to the previous node to explore the remaining neighbors. This return process is the natural behavior of a recursive function call that has completed and returned control to its caller. Thus, the logic of DFS that exhaustively explores a subproblem before returning is inherently recursive, making an iterative implementation a manual simulation of the process [4].

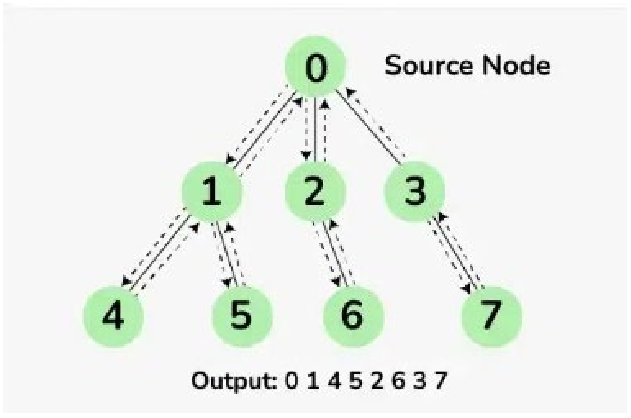


Fig 2.5 DFS Search Sequence Example [5]

The efficiency of the DFS algorithm is measured through time and space complexity. This analysis depends heavily on how the graph is represented [4].

a. Times Complexity

Time complexity measures how the execution time of an algorithm grows as the input size (number of vertices and edges) increases [4].

1. Adjacency List Representation

The analysis is based on two main operations:

- Vertex Visit: Each vertex (V) is visited and processed exactly once thanks to the use of visited arrays or sets. Operations on each vertex (such as marking) take constant time. So, the total time to visit all vertices is proportional to the number of vertices, which is O(V) [4].
- Edge Traversal: The algorithm traverses the adjacency list of each vertex exactly once during the entire process. The total length of all adjacency lists in a graph is equal to the number of edges (E) for a directed graph, or twice the number of edges (2E) for an undirected graph. In both cases, it is proportional to E. So, the total time to check all edges is O(E) [4].

Thus, the total time complexity is the sum of these two components,

$$O(V)+O(E)=O(V+E)$$

2. Adjacency Matrix Representation

When a graph is represented as a V×V matrix, to find all neighbors of a vertex, the algorithm must scan all rows of the corresponding matrix, which contains V entries. This operation must be performed for each vertex. As a result, the total time for edge exploration becomes [4]:

$$O(V\times V)=O(V^2)$$

b. Space Complexity

Space complexity measures the additional memory (auxiliary space) required by the algorithm, beyond the space to store the graph itself.

1. Additional Space Complexity: O(V)

The space requirement of DFS is dominated by two components:

- Recursion or iterative stack: In the worst case, such as in a linear chain graph, the recursion depth can be V. This means the call stack will store up to V function frames. Similarly, the explicit stack in the iterative version can store up to V vertices. This contributes O(V) to the space complexity [4].
- Visited data structure: To keep track of the vertices that have been visited, an array or set whose size is proportional to the number of vertices is required, which is O(V).

Since both components are O(V), the additional space complexity of DFS is O(V) [4].

D. *File System*

In the context of this research, a virtual file system is a software model that simulates the structure and functionality of a real file system on an operating system. The mapping of the file system concept into tree theory is as follows:

- Directory (Folder): Represented as an internal node in an N-ary Tree, because a directory can contain files or other directories (have children).
- File: Represented as a leaf node in a tree, because a file cannot contain other items (it has no children).
- Parent-Child Relationship: The relationship between a directory and the files or sub-directories within it is represented as an edge in the tree.
- Root Directory (/): Represents the root of the entire file system structure [1].

III. IMPLEMENTATION

This chapter explains the design and implementation of a virtual file system. This system is built using the object oriented programming (OOP) paradigm in Python to wrap data structures and their functionality logically.

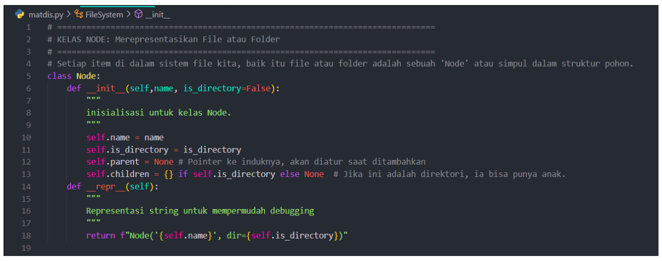
A. *Object Definition*

- Node: This class is designed as a representation of every single unit in the file system, it can be a file or a directory. Each node object has essential attributes to define its status and position in the hierarchy. The following are the attributes that each node has.

Attributes	Explanation
Name	Each file and directory has a name as its identity. Each name must be unique, it cannot be the same between one node and another.
Is_Directory	Is_Directory is a marker whether a node is a directory/folder or not, if it is a directory, then the node can have children, if is_directory is false, then the node cannot have children nodes.
Parent	Parent is an attribute that points to the parent node of a directory.
Children	Children is an attribute that states the child nodes of a directory. If a node

	has an is_directory value of false, then the node does not have any child nodes.

Table 3.1 Attributes of Node



```
matdis.py > FileSystem > __init__
# ==========================================================================
# KELAS NODE: Merepresentasikan File atau Folder
# ==========================================================================
# Setiap item di dalam sistem file kita, baik itu file atau folder adalah sebuah 'Node' atau simpul dalam struktur pohon.
class Node:
    def __init__(self,name, is_directory=False):
        """
        inisialisasi untuk kelas Node.
        """
        self.name = name
        self.is_directory = is_directory
        self.parent = None # Pointer ke induknya, akan diatur saat ditambahkan
        self.children = {} if self.is_directory else None  # Jika ini adalah direktori, ia bisa punya anak.
    def __repr__(self):
        """
        Representasi string untuk mempermudah debugging
        """
        return f"Node('{self.name}', dir={self.is_directory})"
```

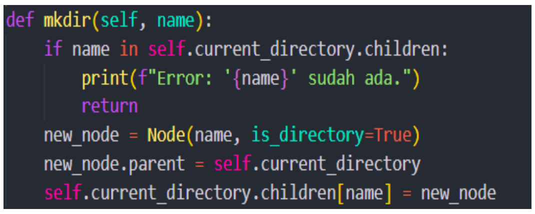
Fig 3.1 Node Class Code

- File System: This class acts as the main manager of the entire tree structure. It is responsible for initializing the system with a root directory, keeping track of the current directory (current_directory), and providing an interface for all file operations.

B. Function and Procedure Definition

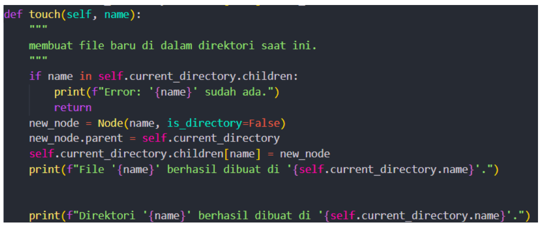
The core functions are implemented as methods in the FileSystem class. Here is an explanation of each key method:

- Mkdir(name) and touch(name): These two methods add a new node to the tree. They check whether the given name already exists in the current directory. If not, a new Node object (either a directory or a file) is created and added to the children dictionary of current_directory.



```
def mkdir(self, name):
    if name in self.current_directory.children:
        print(f"Error: '{name}' sudah ada.")
        return
    new_node = Node(name, is_directory=True)
    new_node.parent = self.current_directory
    self.current_directory.children[name] = new_node
```

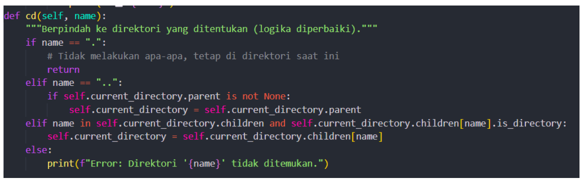
Fig 3.2 Implementation of Mkdir



```
def touch(self, name):
    """
    membuat file baru di dalam direktori saat ini.
    """
    if name in self.current_directory.children:
        print(f"Error: '{name}' sudah ada.")
        return
    new_node = Node(name, is_directory=False)
    new_node.parent = self.current_directory
    self.current_directory.children[name] = new_node
    print(f"File '{name}' berhasil dibuat di '{self.current_directory.name}'.")


    print(f"Direktori '{name}' berhasil dibuat di '{self.current_directory.name}'.")
```

Fig 3.3 Implementation of Touch

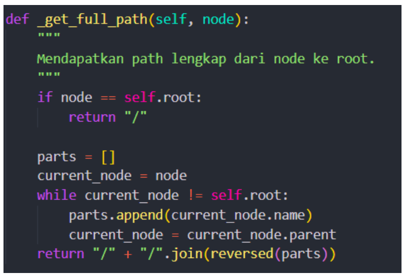
- Cd(name): This method handles navigation between directories. Its logic handles three main cases: moving to a child directory, moving to a parent directory (..), and staying in the current directory (.).



```
def cd(self, name):
    """Berpindah ke direktori yang ditentukan (logika diperbaiki)."""
    if name == ".":
        # Tidak melakukan apa-apa, tetap di direktori saat ini
        return
    elif name == "..":
        if self.current_directory.parent is not None:
            self.current_directory = self.current_directory.parent
    elif name in self.current_directory.children and self.current_directory.children[name].is_directory:
        self.current_directory = self.current_directory.children[name]
    else:
        print(f"Error: Direktori '{name}' tidak ditemukan.")
```

Fig 3.4 Implementation of cd

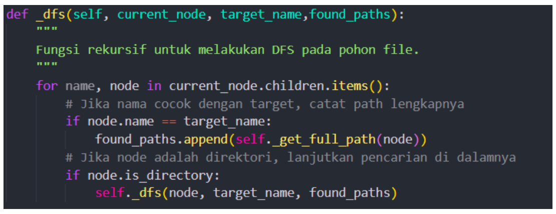
- Get_full_path(node): This is an internal helper function used to reconstruct the absolute path of a node. It works by traversing the hierarchy upwards from the given node through its parent attributes until it reaches the root.



```
def _get_full_path(self, node):
    """
    Mendapatkan path lengkap dari node ke root.
    """
    if node == self.root:
        return "/"

    parts = []
    current_node = node
    while current_node != self.root:
        parts.append(current_node.name)
        current_node = current_node.parent
    return "/" + "/".join(reversed(parts))
```

Fig 3.5 Implementation of get_full_path

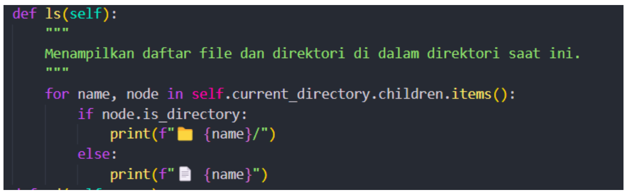
- DFS(current_node, target_name, found_paths): This recursive function is the heart of the DFS algorithm. It takes a node as a starting point, then recursively visits all the nodes below it. If a node with a name matching target_name is found, its full path will be recorded.



```
def _dfs(self, current_node, target_name,found_paths):
    """
    Fungsi rekursif untuk melakukan DFS pada pohon file.
    """
    for name, node in current_node.children.items():
        # Jika nama cocok dengan target, catat path lengkapnya
        if node.name == target_name:
            found_paths.append(self._get_full_path(node))
        # Jika node adalah direktori, lanjutkan pencarian di dalamnya
        if node.is_directory:
            self._dfs(node, target_name, found_paths)
```

Fig 3.6 Implementation of DFS

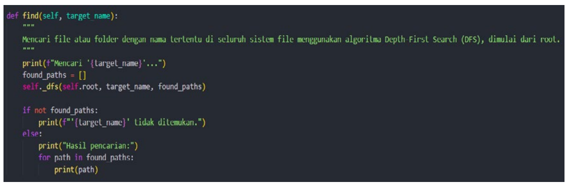
- Ls : Ls is a feature where the program will search for files and directories that are child nodes of a directory.



```
def ls(self):
    """
    Menampilkan daftar file dan direktori di dalam direktori saat ini.
    """
    for name, node in self.current_directory.children.items():
        if node.is_directory:
            print(f"📁 {name}/")
        else:
            print(f"📄 {name}")
```

Fig 3.7 Implementation of Ls

- Find(Name) : Find is a function that will search for a node in the form of a file or directory. This search process will use the dfs function that has been created previously. The find search process always starts from the root node. If a file/directory is found, the program will provide its full path to the user, conversely, if not found, the program will provide a message that the file/directory is not found in the file system.



```python
def find(self, target_name):
    """
    Mencari file atau folder dengan nama tertentu di seluruh sistem file menggunakan algoritma Depth-First Search (DFS), dimulai dari root.
    """
    print(f"Mencari '{target_name}'...")
    found_paths = []
    self._dfs(self.root, target_name, found_paths)

    if not found_paths:
        print(f"'{target_name}' tidak ditemukan.")
    else:
        print("Hasil pencarian:")
        for path in found_paths:
            print(path)
```

Fig 3.8 Implementation of Find

C. *Program Flow and User Interaction*

The main program is located in the if __name__ == '__main__' block, which means this block is executed if and only if the program is run from that file. This program has the following workflow,

1) A file system object is initialized, which automatically creates a root directory.
2) The program will also initialize some initial files and directories to demonstrate the initial structure of the file system.
3) The program then enters an interactive loop, where it displays a prompt showing the current directory path.
4) The user can enter commands (such as ls, cd, mkdir, find), which the program will parse to call the appropriate methods on the file system object, the input structure being:
    a) Mkdir,touch,find,cd: [command] [filename].
    b) Ls : [command] .
5) The loop will continue until the user types the exit command.

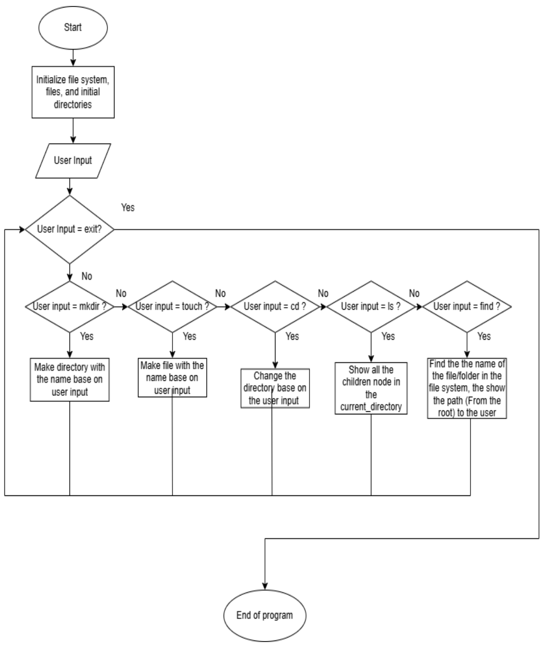
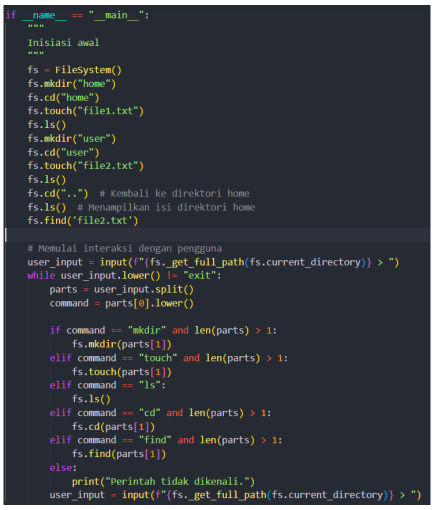

Start
Initialize file system, files, and initial directories
User Input
User Input = exit?
Yes
No
User input = mkdir ?
No
User input = touch ?
No
User input = cd ?
No
User input = ls ?
No
User input = find ?
Yes
Yes
Yes
Yes
Yes
Make directory with the name base on user input
Make file with the name base on user input
Change the directory base on the user input
Show all the children node in the current_directory
Find the the name of the file/folder in the file system, the show the path (From the root) to the user
End of program

Fig 3.9 Flowchart of The Main Program

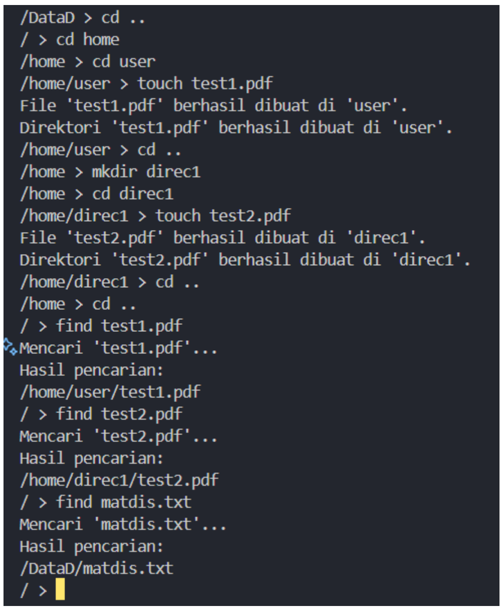


```python
if __name__ == "__main__":
    """
    Inisiasi awal
    """
    fs = FileSystem()
    fs.mkdir("home")
    fs.cd("home")
    fs.touch("file1.txt")
    fs.ls()
    fs.mkdir("user")
    fs.cd("user")
    fs.touch("file2.txt")
    fs.ls()
    fs.cd("..")  # Kembali ke direktori home
    fs.ls()  # Menampilkan isi direktori home
    fs.find('file2.txt')

    # Memulai interaksi dengan pengguna
    user_input = input(f"{fs._get_full_path(fs.current_directory)} > ")
    while user_input.lower() != "exit":
        parts = user_input.split()
        command = parts[0].lower()

        if command == "mkdir" and len(parts) > 1:
            fs.mkdir(parts[1])
        elif command == "touch" and len(parts) > 1:
            fs.touch(parts[1])
        elif command == "ls":
            fs.ls()
        elif command == "cd" and len(parts) > 1:
            fs.cd(parts[1])
        elif command == "find" and len(parts) > 1:
            fs.find(parts[1])
        else:
            print("Perintah tidak dikenali.")
        user_input = input(f"{fs._get_full_path(fs.current_directory)} > ")
```

Fig 3.10 Code of The Main Program

IV. Results and Discussion

After implementing the code part. Next, this paper will provide an explanation of the results of running the program that has been created.

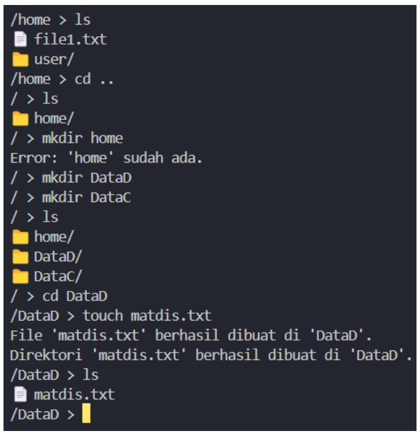


```
/DataD > cd ..
/ > cd home
/home > cd user
/home/user > touch test1.pdf
File 'test1.pdf' berhasil dibuat di 'user'.
Direktori 'test1.pdf' berhasil dibuat di 'user'.
/home/user > cd ..
/home > mkdir direc1
/home > cd direc1
/home/direc1 > touch test2.pdf
File 'test2.pdf' berhasil dibuat di 'direc1'.
Direktori 'test2.pdf' berhasil dibuat di 'direc1'.
/home/direc1 > cd ..
/home > cd ..
/ > find test1.pdf
Mencari 'test1.pdf'...
Hasil pencarian:
/home/user/test1.pdf
/ > find test2.pdf
Mencari 'test2.pdf'...
Hasil pencarian:
/home/direc1/test2.pdf
/ > find matdis.txt
Mencari 'matdis.txt'...
Hasil pencarian:
/DataD/matdis.txt
/ >
```

Fig 4.1 Terminal view when you want to create a new directory and file

As can be seen in Fig 4.1 that the file system has been given several directories and files for its initial initiation. In Fig 4.1 it can be seen that when the user wants to create a new node in the form of a directory named home, the program will check the child node belonging to root (marked as '/'), because the directory named home already exists in the root then the program gives a message that the directory already exists in the root. The user then creates directories named DataD and DataC in the root directory and the program successfully creates both directories because both directories are still not found in the child node belonging to root.

The same thing also happens when the user wants to add a file to a directory. The program will first check the user's input file name is found in the child nodes of the current directory, if not then the program will successfully create the file in the directory.



```
/home > ls
📄 file1.txt
📁 user/
/home > cd ..
/ > ls
📁 home/
/ > mkdir home
Error: 'home' sudah ada.
/ > mkdir DataD
/ > mkdir DataC
/ > ls
📁 home/
📁 DataD/
📁 DataC/
/ > cd DataD
/DataD > touch matdis.txt
File 'matdis.txt' berhasil dibuat di 'DataD'.
Direktori 'matdis.txt' berhasil dibuat di 'DataD'.
/DataD > ls
📄 matdis.txt
/DataD > █
```

Fig 4.2 Terminal view when you want to find a file

In Fig 4.2, it can be seen that the user wants to find out the path of a file. The program will start searching for files with DFS from the root node. If a node does not have the file being searched for, the program will move to the next node. One example is when the user wants to search for test2.pdf. It can be seen in Fig 4.2 that test2.pdf is created in the direc1 directory. The program will start searching from the root node, then it will enter the home node. After entering the home node, the program will enter the user directory node, because test2.pdf is not found in the user node, the program will move from the child node of the home node, namely the direc1 node. The program will start searching for the test2.pdf node in the direc1 node. It turns out that test2.pdf is in the direc1 node so the program will provide a path from the root to the test2.pdf file.

V. CONCLUSION

The conclusion of this paper is that a virtual file system can be represented as an N-ary tree. Each node represents a file/directory where a file is a leaf node and a directory can be a leaf node or a parent node. The DFS algorithm also helps the process of searching for a node. With this algorithm, the program can search for the deepest node of a parent node before moving to another parent node. Of course, there are still faster search algorithms than DFS, but DFS is a suitable algorithm because someone often puts a file in a very deep place, so this algorithm is very suitable for searching for the file.

VI. APPENDIX

The following is the source code for implementing an N-ary tree on a virtual file system and the DFS algorithm for searching a file/directory :

https://github.com/Kurt-Mikhael/Virtual-File-System-Using-N-ary-Trees-with-a-Depth-First-Search-Traversal-Algorithm

VII. ACKNOWLEDGMENT

The author wishes to express gratitude to God Almighty for His blessings and guidance during the preparation of this paper. The author also thanks Dr. Ir. Rinaldi Munir, M.T., the lecturer for the IF1220 Discrete Mathematics course, for the knowledge and guidance he provided. Gratitude is also extended to the author's family for their constant support during the studies.

REFERENCES

[1] B.L. Pratama, “Representasi pohon dalam hierarki Linux filesystem dan manajemen direktori/file,” *Makalah IF2120 Matematika Diskrit*, Institut Teknologi Bandung, Bandung, Indonesia, 2015.

[2] R. Munir, ”Graf: Bagian 1 [course material],” *Bahan Kuliah IF1220 Matematika Diskrit*. Bandung, Indonesia: Program Studi Teknik Informatika STEI-ITB, 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [accessed 17 June 2025]

[3] R. Munir, ”Pohon: Bagian 1 [course material],” *Bahan Kuliah IF1220 Matematika Diskrit*. Bandung, Indonesia: Program Studi Teknik Informatika STEI-ITB, 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. [accessed 18 June 2025].

[4] L.E. Zen and D.U. Iswavigra, “Penggunaan algoritma depth first search dalam sistem pakar: studi literatur,” *J. Inf. dan Teknol.*, vol. 5, no. 2, pp. 85–90, Jun. 2023.

[5] GeeksforGeeks, “DFS traversal of a tree using recursion,” *GeeksforGeeks*, https://www.geeksforgeeks.org/dsa/dfs-traversal-of-a-tree-using-recursion/. [accessed 19 June 2025].

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, 19th June 2025

[illegible]

Kurt Mikhael Purba-13524065