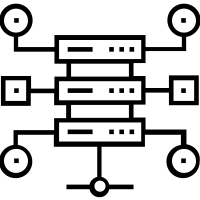
**Assignment No: \_2\_\_**

**Date: 17/ 03 /2025**

**Title: Implementation and Performance Comparison of Tree Data Structures for Efficient File System Management**

(Title based on the application domain and the data structure you will be implementing)

|  |  |  |  |
| --- | --- | --- | --- |
| **Assignment Type of Submission:** |  |  |  |
| **Group** | Yes | Yuxuan Song  24207239  Yijun Liu  24202574  Deepak Shelke  24208478 | **33.33**  **33.33**  **33.33** |

****

1. **Problem Domain Description:**

This project addresses a real-world problem in file system management by leveraging the strengths of tree-based data structures to achieve efficient file storage, retrieval, and prioritization. The problem domain is inspired by the need for hierarchical file organization in modern computing systems, where files and directories must be managed in a way that supports fast insertion, deletion, and searching operations. Traditional linear data structures like arrays and linked lists are insufficient for this purpose due to their O(n) time complexity for search and insertion operations. In contrast, tree-based structures provide O(log n) time complexity for these operations, making them ideal for managing large-scale file systems.

In a real-world scenario, consider a cloud-based file storage system (e.g., Google Drive, Dropbox) where millions of files are stored and accessed by users daily. Each file has a unique name and size, and users frequently perform operations such as inserting new files into the system, searching for specific files by their names, and prioritizing frequently accessed files for faster retrieval.

To simulate this problem, we designed a file management system using three types of tree data structures:

Binary Search Tree (BST): A basic tree structure that organizes files hierarchically. While efficient for small datasets, it can become unbalanced, leading to degraded performance.

AVL Tree: A self-balancing BST that maintains optimal tree height, ensuring O(log n) performance for all operations even with large datasets.

Splay Tree: A self-adjusting tree that moves frequently accessed files closer to the root, optimizing repeated access patterns and improving performance for real-world usage scenarios.

To test the system, we generated a dataset of 10,000 files stored in an Excel file (files.csv). Each file has:

A filename: A random 6-digit number followed by .txt (e.g., 123456.txt).

A size: A random integer between 1 and 200, representing the file size in KB.

This dataset simulates a real-world file system with a large number of files, allowing us to stress-test the performance of each tree structure. We read the files.csv dataset and inserted all 10,000 files into the BST, AVL Tree, and Splay Tree. Then we randomly selected 5 filenames from the dataset and used the search method of each tree to retrieve their sizes.

All trees successfully returned the correct file sizes, demonstrating their ability to handle real-world queries efficiently.

1. **Theoretical Foundations of the Data Structure(s) utilised**

This project models a hierarchical file system using tree-based data structures, ensuring efficient organization, retrieval, and access prioritization.

Binary Search Tree (BST) Implementation

Concept: A tree structure where each node represents a file/folder, ensuring a sorted order.

Operations:

insert(): Adds a file to the appropriate position (O(log n) average, O(n) worst case if unbalanced).

delete(): Removes a file while maintaining structure (O(log n) average, O(n) worst case).

search(): Finds a file based on its name or path (O(log n) average, O(n) worst case).

Advantages: Simple and efficient when balanced, making it useful for structured storage.

AVL Tree Implementation

Concept: A BST with automatic rebalancing to maintain efficiency.

Operations:

insert(): Adds a file and balances the tree (O(log n)).

delete(): Removes a file and rebalances (O(log n)).

search(): Finds a file efficiently (O(log n)).

Advantages: Ensures balanced height, reducing lookup inefficiencies of BST.

Splay Tree Implementation

Concept: A self-adjusting BST that moves frequently accessed files closer to the root.

Operations:

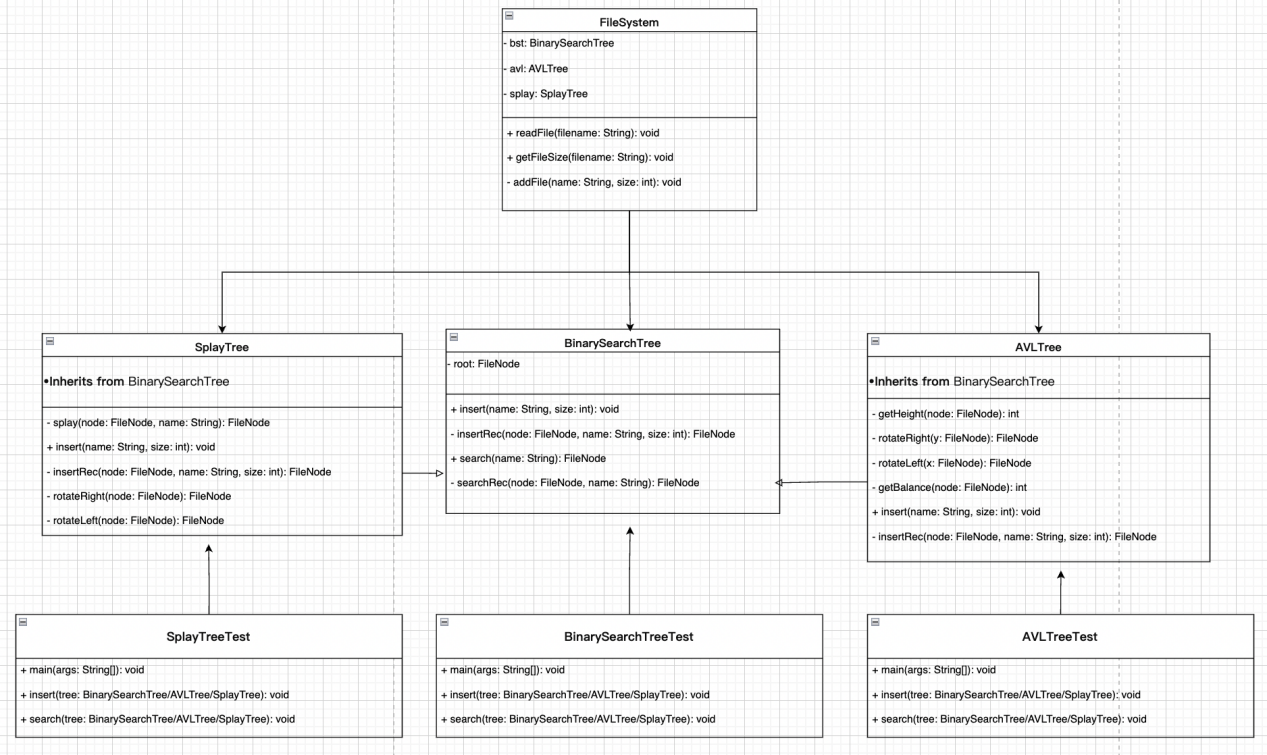
insert(): Adds a file and splay-balances the tree (O(log n) amortized).

delete(): Removes a file and maintains splay properties (O(log n) amortized).

search(): Brings frequently accessed files near the root (O(log n) amortized).

Advantages: Improves efficiency for repeated accesses by dynamically adjusting structure.

1. **Analysis/Design (UML Diagram(s))**

****

1. **Code Implementation (please add your TA -** [Furqan.rustam1@gmail.com](mailto:Furqan.rustam1@gmail.com) **– as a collaborator)**

GitHub (link):https://github.com/WolfClarence/Group9\_Assignment2

BinarySearchTree Implementation:

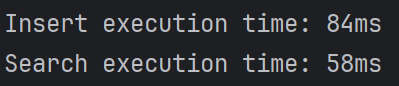
The BinarySearchTree class implements a binary search tree designed to store and retrieve files efficiently. In this structure, each file is represented as a node, with the filename serving as the key for insertion and search operations. The binary search tree ensures an average time complexity of O(log n) for both insertion and search operations. However, in the worst-case scenario, such as when files are inserted in a sorted order, the tree can become unbalanced, leading to a degraded time complexity of O(n).

The FileNode class represents a file within the tree. It includes fields for the filename, which is used as the key for insertion and search, the size of the file, and pointers to the left and right child nodes. Additionally, it contains a height field, which is utilized for balancing in more advanced tree structures like AVL trees.

The insertion method adds a new file to the tree while maintaining the binary search tree property. If the tree is empty, the new file becomes the root node. If the tree is not empty, the method recursively traverses the tree to find the appropriate position for the new file based on the filename. The search method retrieves a file by its filename by recursively traversing the tree and comparing the target filename with the current node's filename to determine the direction of the search.

To evaluate the performance of the binary search tree, a stress test was conducted involving 100,000 insertions and 100,000 searches. The filenames used in the test were randomly generated numbers between 0 and 99,999. While some search operations may return null results due to the randomness of the filenames, this does not affect the evaluation, as the primary focus is on measuring the execution time of the operations. The results of the stress test demonstrated that the binary search tree achieved efficient performance for both insertion and search operations under random data conditions, with the insert execution time of 84ms and search execution time of 58 ms.

Execution Photo:



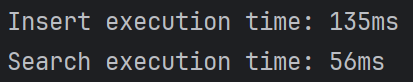
AVL Tree Implementation:

The AVLTree class extends the BinarySearchTree to implement a self-balancing binary search tree known as an AVL tree. While it shares many similarities with the binary search tree, such as the structure of the FileNode class and the basic insertion and search logic, the key difference lies in its ability to maintain balance through rotations. This ensures that the tree remains balanced, guaranteeing O(log n) time complexity for insertion, search, and deletion operations even in the worst-case scenarios.

The AVL tree achieves balance by calculating a balance factor for each node, which is the difference in height between its left and right subtrees. If the balance factor of any node exceeds the allowed threshold (either greater than 1 or less than -1), the tree performs rotations to restore balance. The rotations include right rotations for left-heavy trees and left rotations for right-heavy trees, as well as combinations like left-right and right-left rotations for more complex imbalances. These rotations ensure that the tree height remains logarithmic relative to the number of nodes, optimizing performance for all operations.

To evaluate the performance of the AVL tree, a stress test was conducted involving 100,000 insertions and 100,000 searches, using randomly generated filenames between 0 and 99,999. The results demonstrated that the AVL tree achieved efficient performance, with an insertion execution time of 135 ms and a search execution time of 56 ms. While the insertion time is slightly higher compared to the binary search tree (84 ms), this is expected due to the additional overhead of maintaining balance through rotations. However, the search performance remains comparable (56 ms for AVL tree vs. 58 ms for BST), highlighting the AVL tree's ability to provide consistent and efficient search operations even under large datasets. This makes the AVL tree particularly suitable for applications where balanced tree structures are critical for maintaining optimal performance.

Execution Photo:



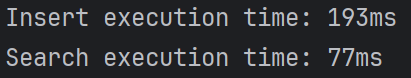
Splay Tree Implementation:

The SplayTree class extends the BinarySearchTree to implement a self-adjusting binary search tree known as a splay tree. Unlike the AVL tree, which maintains balance through strict height constraints, the splay tree optimizes for locality of reference by moving frequently accessed nodes closer to the root through a process called splaying. This self-adjusting property ensures that recently accessed nodes can be accessed more quickly in subsequent operations, making the splay tree particularly efficient for workloads with repetitive access patterns.

The key feature of the splay tree is the splay operation, which restructures the tree to bring a specific node to the root. This operation is performed during both insertion and search operations. The splay operation uses a combination of rotations, including zig, zag, zig-zig, zag-zag, zig-zag, and zag-zig, to move the target node to the root while maintaining the binary search tree property. While this restructuring does not guarantee a balanced tree, it ensures that frequently accessed nodes remain near the root, optimizing access times for repetitive queries.

To evaluate the performance of the splay tree, a stress test was conducted involving 100,000 insertions and 100,000 searches, using randomly generated filenames between 0 and 99,999. The results demonstrated that the splay tree achieved an insertion execution time of 193 ms and a search execution time of 77 ms. While the insertion time is higher compared to both the binary search tree (84 ms) and the AVL tree (135 ms), this is due to the additional overhead of the splay operation during each insertion. However, the search performance remains efficient (77 ms), particularly for workloads where certain nodes are accessed repeatedly, as the splay operation brings frequently accessed nodes closer to the root. This makes the splay tree well-suited for applications with non-uniform access patterns, where optimizing for frequently accessed data is critical.

Execution Photo:



1. **Video of the Implementation running**

youtube link: https://www.youtube.com/watch?v=deLd4JLdNz

Comments:

This is a brief overview of the code execution and result display. The results are shown in the console.

**Please save as pdf and submit on Brightspace**

**Students belonging to the same group** please **submit the same file .**