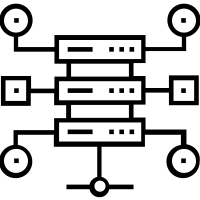
**Assignment No: \_3\_\_**

**Date: 31/ 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 report addresses the problem of selecting the top 10 most viewed posts from a large dataset of posts on a website. The website hosts a large number of posts for users to browse, and to enhance the user experience, it recommends the 10 posts with the highest view counts on the homepage. Given the large dataset size of 10,000 posts, an efficient algorithm is required to identify and extract the top 10 posts without sorting the entire dataset, which would be computationally expensive.

To simulate a real-world scenario, a dataset containing 10,000 posts was generated. Each postId was assigned a unique integer between 1 and 10,000, and each view count was a randomly generated integer between 0 and 99,999. The dataset was stored in a CSV file named posts.csv, with each row containing a postId and a view count. A naive approach would be to sort all 10,000 posts by view count and pick the top 10. However, sorting has a time complexity of O(n log n), which is inefficient for large datasets. Instead, a minimum heap (priority queue) was used because it allows efficient selection of the top 10 elements while scanning through the dataset in a single pass. The heap maintains only 10 elements at a time, ensuring efficient memory usage and reducing the overall time complexity to approximately O(n).

The algorithm follows these steps. First, a minimum heap of size 10 was created to store posts with their postId and viewCount. Then, the CSV file was read line by line, extracting the postId and viewCount from each row. Each post was then inserted into the heap. If the heap contained fewer than 10 elements, the new post was added. If the heap already had 10 elements, the new post’s view count was compared with the smallest view count in the heap. If the new post had more views than the smallest element in the heap, the smallest element was removed, and the new post was inserted. After processing all posts, the heap contained the 10 most viewed posts. Finally, the posts in the heap were retrieved and displayed, representing the top 10 most popular posts on the website.

The algorithm successfully processed the entire dataset and extracted the top 10 posts with the highest view counts. The execution time for processing all 10,000 posts and retrieving the results was approximately 18 milliseconds, demonstrating the efficiency of this method.

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

1. **Analysis/Design (UML Diagram(s))**
2. **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\_Assignment3

This is a new repository, please join as the collaborator.

MyHeap Implementation:

MyHeap is a custom binary heap that functions similarly to Java’s PriorityQueue. It supports generic types and relies on a Comparator to determine ordering, allowing for both min-heaps and max-heaps. Internally, it uses an ArrayList for dynamic storage and implements heap operations like insertion (offer), removal (poll), and retrieval (peek).

To maintain the heap property, it employs heapify operations:

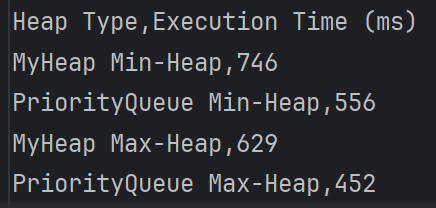
Sift-up restores order when adding elements.

Sift-down ensures order when removing elements.

This implementation provides an efficient O(log n) time complexity for insertions and deletions, making it suitable for priority-based applications.

The performance of both heaps was evaluated over 1,000,000 operations, and the execution times were logged into a CSV file. The test results show that the custom MyHeap implementation is slightly slower than Java's built-in PriorityQueue. For the Min-Heap, MyHeap took 746 ms, while PriorityQueue took 556 ms. For the Max-Heap, MyHeap took 629 ms, compared to PriorityQueue's 452 ms. This performance difference can be attributed to the fact that PriorityQueue is highly optimized with a well-structured internal implementation, ensuring efficient memory usage and optimized heap operations. Despite these differences in performance, the custom heap's execution time remains within acceptable bounds for most use cases, as both heaps maintain the expected O(log n) time complexity for insertion and deletion operations.

Sample output:



1. **Video of the Implementation running**

youtube link:

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 .**