CMPE 211: Data Structures and Algorithms Project Report

Name: Emre Şentürk
Student ID: 123200133
Submission Date: January 19, 2025

1. Introduction :

In this report, we compare the performance of sorting algorithms (Merge Sort and Quick Sort) and search algorithms (Binary Search Tree and Hash Table) across different input instances and operation sequences. The goal is to analyze how these algorithms perform under varying conditions, providing insights into their strengths and weaknesses. This helps in selecting the most appropriate algorithm for different problem settings.

Objectives:

- To compare the performance of two sorting algorithms (Merge Sort and Quick Sort).
- To compare the performance of two search algorithms (Binary Search Tree and Hash Table).
- To analyze the reasons behind performance differences across different inputs and operations.

2. Sorting Algorithms:

The two sorting algorithms chosen for this project are **Merge Sort** and **Quick Sort**. The comparison is based on the execution times of these algorithms on two different input instances.

Instance 1: Shuffled Input

- Example: [8, 3, 7, 4, 2, 6, 1, 5]
- This instance represents a randomly shuffled array, which is typically favorable for algorithms that have consistent time complexity, like Merge Sort.

Instance 2: Partially Sorted Input

- [1, 2, 3, 4, 5, 6, 7, 8]
- This instance represents a sorted array, where Quick Sort might perform better due to its pivot selection mechanism.

3. Search Algorithms

The two search algorithms evaluated are **Binary Search Tree (BST)** and **Hash Table**. The performance of these algorithms is analyzed using two different sequences of operations.

Sequence 1:

- Operations: Put(5), Put(10), Put(3), Get(10), Get(5), Get(7)
- This sequence includes a mixture of insertions and retrievals.

Sequence 2:

- Operations: Put(2), Put(4), Put(1), Get(3), Get(4), Put(6), Get(6)
- This sequence includes a different set of operations to assess the impact of operation frequency and key distribution.

4. Results:

Sorting Algorithm Comparison

The following execution times were recorded for each sorting algorithm on the two input instances.

Instance 1: Shuffled Input

- **Merge Sort** Time: 0.210554s
- **Quick Sort** Time: 0.154535s

Instance 2: Partially Sorted Input

- **Merge Sort** Time: 0.157513s
- **Quick Sort** Time: 0.109016s

Analysis:

- **Instance 1 (Shuffled Input)**: Merge Sort performs better when the array is shuffled because its time complexity remains consistent regardless of the input order, while Quick Sort may degrade to $O(n^2)$ in the worst case.
- **Instance 2 (Sorted Input)**: Quick Sort outperforms Merge Sort due to its better average-case performance in scenarios where the array is sorted, thanks to its efficient pivot-based partitioning.

Search Algorithm Comparison

The following execution times were recorded for each search algorithm on the two operation sequences:

Sequence 1:

- **Binary Search Tree** Time: 0.000013s
- **Hash Table** Time: 0.000007s

Sequence 2:

- **Binary Search Tree** Time: 0.000008s
- **Hash Table** Time: 0.000003s

Analysis:

- **Sequence 1**: The Hash Table performs better because it allows constant-time lookup and insertion O(1), whereas Binary Search Tree operations typically take logarithmic time O(log n).
- **Sequence 2**: Depending on the sequence, BST might perform better if the keys are inserted in a balanced manner. However, Hash Tables tend to perform well due to the quick access times even in cases of random or scattered inserts.

5. Conclusion

In conclusion, the choice of algorithm for sorting or searching depends largely on the characteristics of the input data and the operations being performed. **Merge Sort** and **Quick Sort** both have their strengths and weaknesses depending on the input's structure. Similarly, **Binary Search Trees** and **Hash Tables** each excel under different conditions, with Hash Tables generally being faster for random access patterns, and Binary Search Trees performing well with balanced data.

This experiment demonstrates the importance of understanding the underlying characteristics of algorithms and their performance implications based on input types and operation sequences.