Name: Darshit Shah

Priority Queue Comparison: Binary Heap vs. Binary Search Tree

1. Introduction

This project explores and compares two classic data structures for implementing priority queues: the binary heap and the binary search tree (BST). Priority queues are essential in real-world systems such as operating system schedulers, simulation engines, and shortest path algorithms. Although both structures support logarithmic-time operations in theory, their practical performance often differs due to input patterns and structural behavior.

2. Background and Research

To guide this project, I reviewed recent literature including the 2025 study titled "Evaluation of Priority Queues in the Priority Flood Algorithm for Hydrological Modelling." This paper compared several PQ implementations including binary heaps, AVL trees, skip lists, pairing heaps, and a novel hash-augmented structure called HHeap. The study concluded that while binary heaps remain robust and efficient for general use, other structures like pairing heaps or HHeap can outperform them in specialized large-scale workloads such as terrain modeling and flood simulation.

My goal was to conduct a foundational comparison between two widely taught and understood data structures: binary heaps and unbalanced BSTs. This serves as a stepping stone to understanding the performance characteristics of more advanced PQ variants.

Relevant time complexities:

- - Binary Heap: O(log n) insert/delete_min, O(1) find_min
- - BST (Unbalanced): Average O(log n), Worst-case O(n)

3. Implementation Summary

I implemented two priority queue classes in Python:

- - HeapPriorityQueue using Python's built-in heapq
- - BSTPriorityQueue using a custom binary search tree

Each implementation includes insert, find_min, and delete_min operations. Code was structured with clean interfaces and tested with random input sequences.

4. Experimental Setup

To evaluate performance, I ran three different workloads:

- - 70% insert / 30% delete
- - 50% insert / 50% delete
- - 30% insert / 70% delete

For each workload, I executed 1000 operations on fresh instances of each data structure, recording the average time per insert and delete operation using Python's perf_counter().

5. Results

The benchmark revealed that HeapPriorityQueue consistently performed faster and more stably across all workloads. BSTPriorityQueue's insert operation slowed significantly in insert-heavy scenarios due to structural imbalance, confirming theoretical expectations.

The results were visualized using bar and line plots, along with a tabular summary of operation timings.

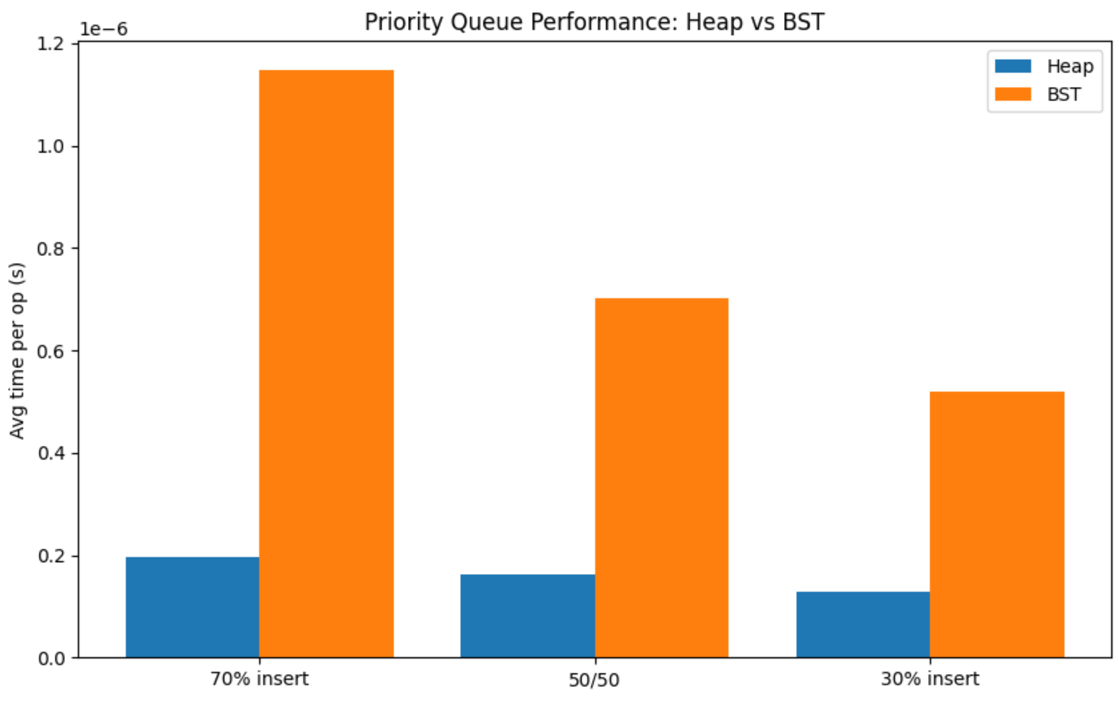

Fig 5.1 : Average time per operation of Priority Queues

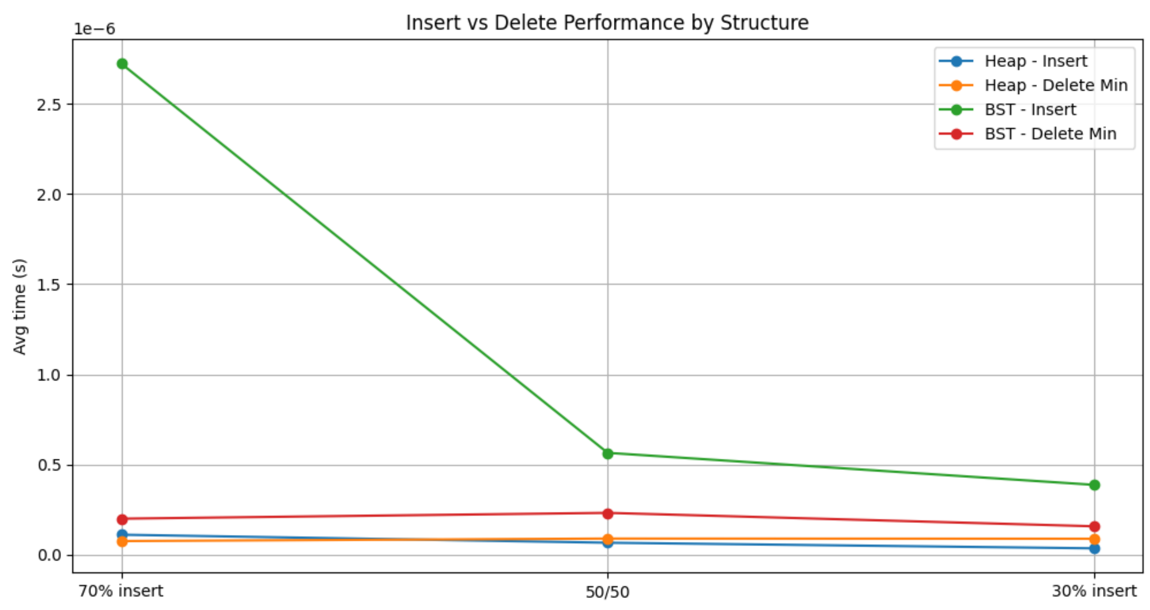

Fig 5.2 : Insert vs Delete Performance chart for Heap and BST

6. Analysis

Binary heaps maintained stable performance due to their compact structure and heap property, even in insert-heavy workloads. On the other hand, unbalanced BSTs degraded to near-linear performance during frequent inserts, especially with sorted or semi-sorted inputs.

Interestingly, delete_min operations in BSTs showed more stable timing, indicating that the leftmost-path traversal is less sensitive to imbalance. Heaps, however, remained faster across the board.

7. Extension and Improvement Opportunities

Building on recent research findings, several improvements can be considered:

- Replace the BST with a self-balancing variant like AVL or Red-Black Tree for consistent O(log n) performance.
- Experiment with pairing heaps, as they performed very well in the 2025 study under large workloads.
- Explore HHeap, a hash-augmented heap variant that showed excellent performance in large, real-world terrain modeling datasets.
- Use cache-aware or d-ary heaps to improve performance under specific hardware constraints or CPU cache conditions.
- Implement input reshuffling or batching techniques to reduce imbalance in BSTs and optimize bulk insert operations in heaps.

8. Conclusion

This project reaffirmed the robustness of binary heaps for implementing priority queues under general-purpose workloads. While unbalanced BSTs serve as useful educational tools, they quickly degrade in skewed input scenarios. With emerging research pointing toward cache-friendly and hybrid data structures, future work can explore integrating these advanced variants into practical systems.

9. References

1) Ma, Lejun & Yuan, Yue & Wang, Huan & Liu, Huihui & Wu, Qiuling. (2025). Evaluation of Priority Queues in the Priority Flood Algorithm for Hydrological Modelling. Water. 17. 10.3390/w17223202.

2) Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

3) Grammarly Software for writing.