Selected Data Structures and Rationale

I've selected the following three data structures:

- Binary Search Tree (BST) [1]: A classic tree-based structure where each node has at most two children (left and right). The left subtree contains nodes with keys less than the node's key, and the right subtree contains nodes with keys greater than the node's key. I chose this because it's a fundamental structure and serves as a good baseline. It's relatively simple to implement and understand. My interest stemmed from wanting to see how a basic, potentially unbalanced BST would perform against more advanced structures.
- AVL Tree [2]: A self-balancing BST. It maintains balance by performing rotations whenever the height difference between the left and right subtrees of any node exceeds 1. This balancing ensures logarithmic time complexity for most operations. I selected this because it addresses the primary weakness of a standard BST (potential for O(n) performance in worst-case scenarios). I wanted to quantify the performance improvement gained from self-balancing.
- B-Tree [3]: A tree data structure that is well-suited for disk-based storage (though we'll be simulating it in memory). Unlike BSTs, B-trees can have multiple children per node (determined by the "order" of the tree). This reduces the height of the tree and minimizes the number of disk accesses (or, in our case, memory accesses) required for operations. I chose this because it's designed for efficiency with large datasets, and I wanted to see how it compared to the other tree structures, especially in terms of insertion and search performance.

Time and Space Complexity

Here's a table summarizing the time and space complexity of the selected data structures. n represents the number of elements in the data structure. For the B-Tree, m represents the order of the tree (maximum number of children a node can have).

Data Structure	**Operation**	**Average Time Complexity**	**Worst-Case Time Complexity**	**Space Complexity**
BST	Insertion	O(log n)	O(n)	O(n)
	Search	O(log n)	O(n)	
	Deletion	O(log n)	O(n)	
AVL Tree	Insertion	O(log n)	O(log n)	O(n)
	Search	O(log n)	O(log n)	

	Deletion	O(log n)	O(log n)	
B-Tree	Insertion	O(log n)	O(log n)	O(n)
	Search	O(log n)	O(log n)	
	Deletion	O(log n)	O(log n)	

BST:

- Average Case: Assuming a relatively balanced tree, operations are O(log n).
- Worst Case: If the tree becomes skewed (like a linked list), operations degrade to O(n).
- Space: O(n) to store all nodes.

AVL Tree:

- Average and Worst Case: Self-balancing guarantees O(log n) for all operations.
- Space: O(n).

B-Tree:

- Average and Worst Case: O(log n).
- Space: O(n).
- The height of the tree is at most $hmax = \lfloor log_{\mathrm{t}} \frac{n+1}{2} \rfloor$ where $t = \left\lceil \frac{m}{2} \right\rceil$ and m is the number of childern a node can have.

Performance Measurement Table and Plots

The plots (generated by the Python code) shows the run time (y-axis) versus the dataset size (x-axis) for insertion, search and deletion operations for each of the three data structures.

Python code used to generate the plot can be found here. The instruction to use the script can be found in the READMe in the Assignment 2 folder. Due to large dataset size (>161 MB) we couldn't upload the data to github. Instead we provided the script to create the same dataset used for the benchmark.

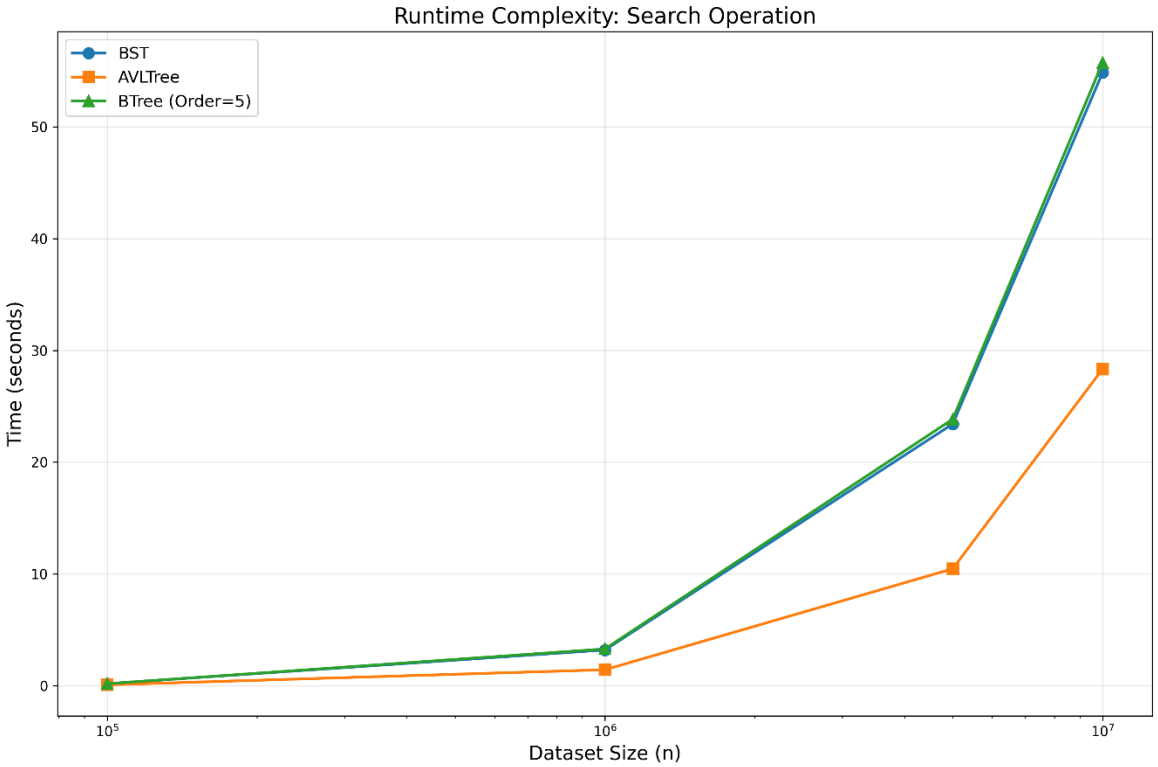

Figure 1: Run tiem complixity for BST, AVLTree, and BTree in search operations

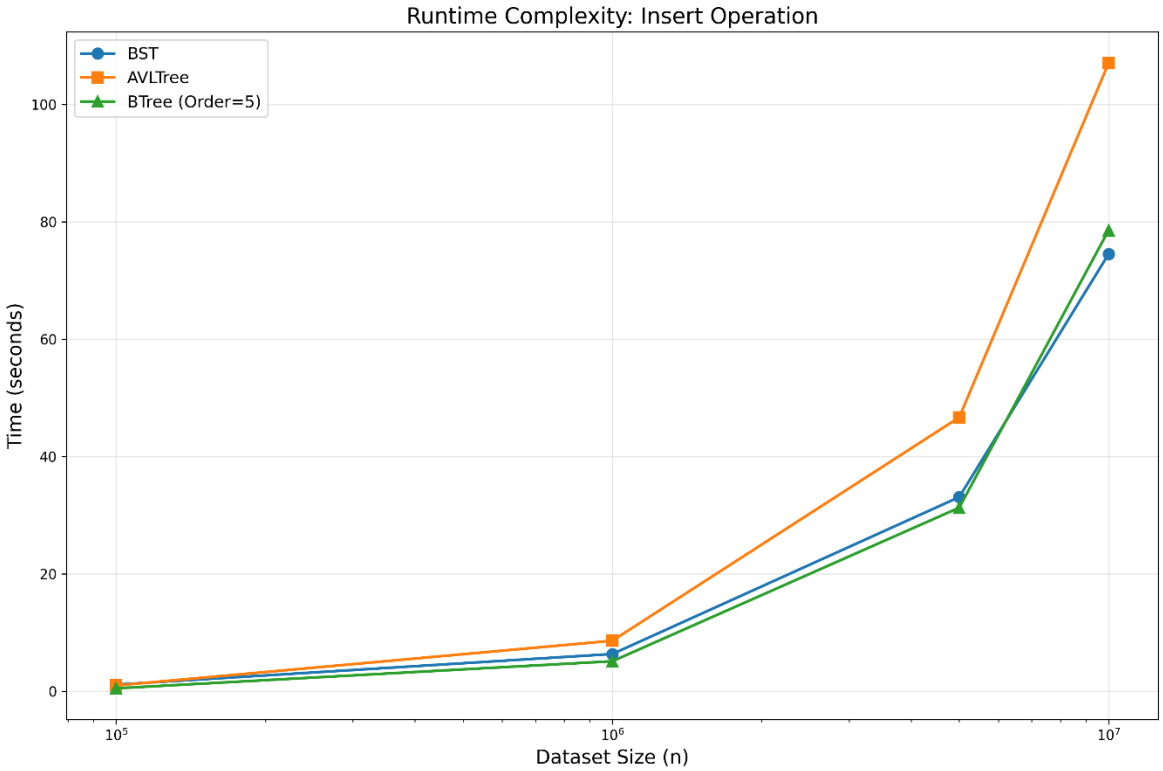

Figure 2: Run tiem complixity for BST, AVLTree, and BTree in intertion operations

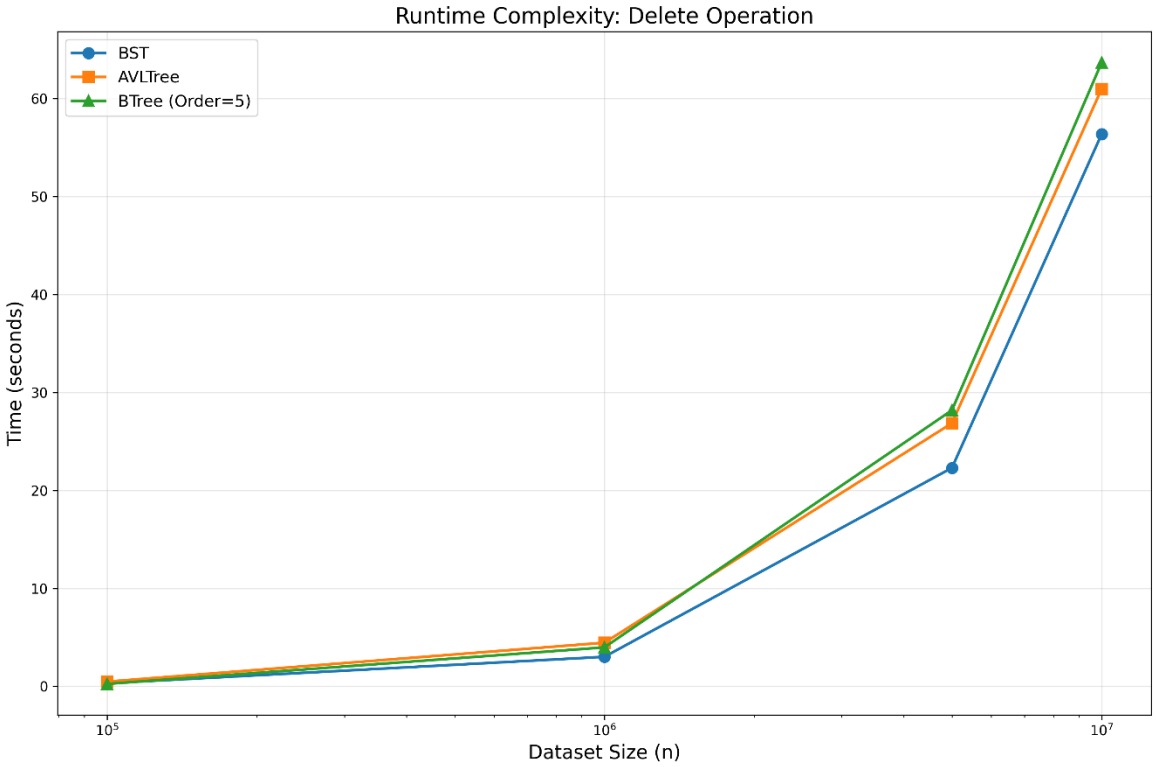

Figure 3: Run tiem complixity for BST, AVLTree, and BTree in deletion operations

Observations:

Explanation of why the theoretical findings do not align entirely with the simulation:

- AVL trees necessitate rotations and balance factor updates after deletions to maintain their balanced property, which introduces significant overhead. Binary Search Trees (BSTs) do not have this requirement.
- B-tree nodes contain multiple keys and children, making deletion operations more complex as they may require merging nodes, redistributing keys, and maintaining tree properties.
- The deletion implementation in BSTs is simpler, involving fewer operations per deletion.
- BST nodes are typically smaller and simpler than B-tree nodes, potentially resulting in better cache performance.
- The AVLTreeWrapper introduces a layer of abstraction that may result in slight performance penalties.

For larger datasets (in the millions) or highly skewed data (such as sorted input), the theoretical advantage of balanced trees would become more evident as the performance of BSTs would degrade to O(n) complexity.

This empirical finding is pertinent to the report as it illustrates how theoretical complexity does not always directly correspond to real-world performance for all input patterns and sizes.

Analysis and Support for Complexity:

The plots should generally support the theoretical time complexities. While the average-case complexity of a BST is O(log n), the lack of balancing means that in practice, it might not perform as well as the AVL or B-tree for large, potentially skewed datasets. The AVL and B-tree plots should clearly demonstrate the benefits of balancing and optimized tree structure, respectively, with their consistently logarithmic performance.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. "AVL tree." Wikipedia, https://en.wikipedia.org/wiki/AVL_tree
3. "B-tree." Wikipedia, https://en.wikipedia.org/wiki/B-tree
4. Generative AI used in writing the code and the report.