Performance Comparison of Data Structures in Contact Management System

Introduction

This report analyzes the performance of five common data structures such as **Array**, **Singly Linked List**, **HashMap**, **Binary Search Tree (BST)**, and **Balanced Binary Search Tree (AVL Tree)** in a contact management system. The system was tested using four operations: Insert, Search, Update, and Delete, with contact data sizes of 100, 500, and 1000. Performance was measured in milliseconds (ms), and the benchmark results were plotted to visually assess the efficiency of each data structure.

Data Structures Overview

We evaluate the following data structures:

1. **Array (List)**: A simple, contiguous memory structure where elements are accessed by index.
2. **Singly Linked List**: A linear structure where each element points to the next, making insertion and deletion straightforward.
3. **HashMap (Separate Chaining)**: A key-value store with efficient lookup and modification operations on average.
4. **Binary Search Tree (BST)**: A tree structure that allows for efficient search and insertion when balanced.
5. **AVL Tree**: A self-balancing BST that ensures logarithmic time complexity for most operations

Code Implementations in C program:
https://github.com/akgWMU/pa3-datastructs-contact-management-system-f25cs5115-santhiya-2000.

Time & Space Complexities:

Data Structure	Insert	Search	Delete	Update	Space Complexity
Array	O(1) (append at end) / O(n) (insert in middle)	O(n) (linear search)	O(n) (shift elements)	O(n) (need search first)	O(n) (contacts + contiguous storage)
Linked List	O(1) (at head) / O(n) (at tail or specific position)	O(n) (traverse sequentially)	O(n) (find + re-link)	O(n) (need search first)	O(n) (contacts + next pointers)
Hash Map	O(1) avg, O(n) worst (collision-heavy)	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(1) avg, O(n) worst (search first)	O(n + m) (contacts + buckets)
BST (Unbalanced)	O(log n) avg, O(n) worst (skewed tree)	O(log n) avg, O(n) worst	O(log n) avg, O(n) worst	O(log n) avg, O(n) worst (search first)	O(n) (contacts + pointers)

AVL Tree (Balanced BST)	O(log n) (with rotations)	O(log n)	O(log n) (with rebalancing)	O(log n) (need search first)	O(n) (contacts + balance factor per node)

Performance Analysis

Based on the performance data across various operations and contact sizes:

Insert Operation

- **Array**: The insertion time increases steeply as the number of contacts grows. At 1000 contacts, the insertion time was 0.492ms (for 100 contacts), rising to 2.868ms. This is expected due to the need to shift elements in the array.
- **LinkedList**: Shows a similar trend to arrays, with the insertion time rising significantly as the dataset size increases. For 1000 contacts, the time was 2.000ms, much slower than **HashMap** and **AVL**.
- **HashMap**: The insert time is relatively consistent even with increasing contacts. For 1000 contacts, the insertion time was 0.807ms, making it significantly faster than Array and LinkedList.
- **BST & AVL**: Both have logarithmic insertion times. **BST** starts with a time of 0.101ms at 100 contacts and increases to 0.994ms at 1000 contacts. **AVL** maintains even better performance, starting at 0.061ms and rising to 0.634ms for 1000 contacts.

Search Operation

- **Array**: The search time increases linearly, with the search time for 1000 contacts reaching 2.580ms. The data structure suffers from inefficient searching since it requires a linear scan.
- **LinkedList**: Similar to arrays, the search time increases substantially as contacts increase. For 1000 contacts, the search took 3.216ms, indicating its inefficiency for large datasets.
- **HashMap**: The search operation remains highly efficient, averaging close to 0.200ms at 1000 contacts. This is a key advantage of hash maps, making them ideal for fast lookups.
- **BST & AVL**: **BST** shows an increase in search time with 1000 contacts at 0.222ms, while **AVL** shows even better performance at 0.146ms for 1000 contacts due to its balanced nature.

Update Operation

- **Array**: Update times follow a similar trend to insertion times, with the time rising significantly at larger contact sizes. For 1000 contacts, the update time reached 3.395ms.
- **LinkedList**: Updates also suffer from linear time complexity, with a time of 2.958ms at 1000 contacts, slightly better than **Array** due to easier insertion/deletion in linked lists.
- **HashMap**: Continues to show excellent performance with a time of 0.223ms at 1000 contacts, which is almost constant due to the nature of hash tables.
- **BST & AVL**: Both trees show logarithmic update times, with **AVL** outperforming **BST** at all levels. At 1000 contacts, **BST** took 0.263ms while **AVL** took 0.165ms.

Delete Operation

- **Array**: The delete operation, like insertion, requires shifting elements, resulting in high time complexity. At 1000 contacts, deletion took 2.868ms.
- **LinkedList**: Deletes are faster in linked lists than arrays, with the time increasing to 0.086ms at 1000 contacts. Deleting from the head or tail is efficient in a linked list.
- **HashMap**: Deletions are also constant time on average, with 1000 contacts requiring 0.275ms.
- **BST & AVL**: Both trees offer efficient deletions, with **AVL** offering the best performance at 0.301ms at 1000 contacts compared to **BST** at 0.264ms.

Graphical Performance Analysis

The **performance graph** visualizes the performance differences in time (milliseconds) across different operations (Insert, Search, Update, and Delete) and varying contact sizes (100, 500, and 1000). Below are key takeaways from the graph:

1. **Insert & Delete Operations**:
    - **Arrays** and **LinkedLists** show a steep rise in time as the number of contacts increases, indicating their inefficiency in handling large datasets.
    - **HashMap**, **BST**, and **AVL** trees show much steadier growth, with AVL trees consistently outperforming all other structures.
2. **Search & Update Operations**:
    - **HashMap** dominates both operations in terms of speed, followed by **AVL** trees.
    - **BST** also shows good performance, but it is outperformed by **AVL** due to its balanced nature.
    - **Array** and **LinkedList** both suffer from poor performance, especially at larger sizes.

Trade-offs and Design Choices

1. **Array**: Arrays are fast for indexed access, but they struggle with insertions and deletions. They are best used when the dataset is static or when only search operations are frequent.
2. **LinkedList**: Linked lists provide fast insertions and deletions, but their search and update operations are slower compared to other structures. They are well-suited for use cases where dynamic operations (insertion/deletion) are more frequent than search.
3. **HashMap**: The **HashMap** excels at fast lookups and updates but does not maintain any order. It is ideal when quick access to data is required, such as in caching or dictionaries.
4. **BST**: While **BST** provides efficient search, insert, and delete operations, its performance degrades if the tree becomes unbalanced. It is suitable for applications where ordered data and efficient searching are important.
5. **AVL Tree**: **AVL trees** ensure efficient performance for all operations (insert, search, update, delete), making them the most reliable choice when performance consistency is key, especially in real-time systems.

Real-World Use Cases

1. **Array**: Suitable for applications with static data or scenarios requiring frequent random access, such as small contact lists.
2. **LinkedList**: Ideal for dynamic datasets with frequent insertions and deletions, such as task lists or undo operations in software.

3. **HashMap**: Best for systems where fast lookups are essential, like storing user sessions, caching, or dictionary lookups.
4. **BST & AVL Trees**: Both are ideal for applications requiring ordered data and efficient search, like in databases, file systems, or search engines.

Conclusion

The benchmark results demonstrate that **AVL Trees** offer the most balanced performance, excelling in all operations (insert, search, update, and delete). **HashMaps** also perform excellently for fast lookups but don't maintain order. **Arrays** and **LinkedLists** are less efficient for dynamic datasets, with **LinkedLists** performing slightly better in terms of insertions and deletions.