Performance Comparison of Data Structures in a Contact Management System

Objective: To analyze and compare arrays, linked lists, hash maps, BSTs, and AVL trees in storing and managing contact information.

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

Data Structures & Design Choices:

- **Array**: simple contiguous storage, best for small datasets.
- **Linked List**: supports fast insert/delete but costly search.
- **Hash Map**: direct access via hashing, best for large datasets.
- **BST**: keeps order, but can degrade if unbalanced.
- **AVL Tree**: balanced BST with guaranteed O(log n).

Performance Analysis:
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 for each operations:

- **Insert/Delete**: Fastest in Linked List & Hash Map.
- **Search/Update**: Hash Map is best, followed by AVL Tree.
- **Memory**: Array uses least overhead. Linked List & AVL need extra pointers.
- **Scalability**: Hash Map and AVL scale well. Array/Linked List degrade linearly.

Trade-Offs in Data Structures for Contact Management

When implementing a contact management system, the choice of data structure impacts speed, memory, and usability. Each option comes with advantages and disadvantages, and the best choice depends on workload and requirements.

1. Array

- **Advantages:**
  - Very simple to implement.
  - Fast sequential access.
  - Minimal memory overhead.
- **Disadvantages:**
  - Insertions/deletions in the middle require shifting **O(n)**.
  - Searching is **O(n)** unless sorted.
  - Fixed size unless reallocated.
- **Trade-off:**
  Best for **small, read-heavy datasets** where simplicity matters. Poor for dynamic updates.

2. Singly Linked List

- **Advantages:**
  - Insertions/deletions at the head are **O(1)**.
  - Dynamic memory allocation and no resizing needed.
- **Disadvantages:**
  - Searching requires traversal **O(n)**.
  - Extra memory needed for pointers.
  - Poor cache performance due to scattered memory.
- **Trade-off:**
  Good when **frequent insertions/deletions** are required, but **slow lookups** make it unsuitable for large datasets.

3. Hash Map

- **Advantages:**
  - Average-case **O(1)** for insert, search, delete, and update.
  - Excellent scalability for large datasets.

- **Disadvantages:**
    - Performance depends on a good hash function.
    - Worst-case collisions → **O(n)**.
    - Consumes more memory for buckets and resizing.
    - Does not preserve ordering of contacts.
- **Trade-off:**
  Ideal for **fast, direct lookups** (like searching by name), but **cannot return ordered contact lists** without extra steps.

4. Binary Search Tree (Unbalanced)

- **Advantages:**
    - Stores contacts in sorted order.
    - Enables range queries and in-order traversal.
- **Disadvantages:**
    - Can degrade into a linked list in the worst case (**O(n)**).
    - Insertions/deletions can become inefficient if the tree is unbalanced.
- **Trade-off:**
  Useful if **ordering is important**, but **unreliable without balancing**.

5. Balanced BST (AVL Tree)

- **Advantages:**
    - Guarantees **O(log n)** for insert, search, update, and delete.
    - Always balanced, predictable performance.
    - Supports ordered traversal.
- **Disadvantages:**
    - More complex to implement than other structures.
    - Rotations add slight overhead to insertions/deletions.
    - Requires extra memory to store balance factors.

Trade-off:
Best for **ordered datasets with guaranteed performance**, but more complex and slightly slower than a hash map for raw lookups.

Realworld Implications - Application based Analysis:

- **Small datasets** - Array is fine (low memory, simple).
- **Frequent insert/delete** - Linked List.
- **Fast lookups** - Hash Map.
- **Ordered queries (range, sorted lists)** - AVL Tree.
- **BST** alone is not reliable unless balanced.