**RIPHAH INTERNATIONAL UNIVERSITY LAHORE CAMPUS**

***Riphah School of Computing & Innovation (RSCI)***

**Design & Analysis of Algorithm**

**BS Computer Science (5B) – Fall 2024**

**Semester Project**

|  |  |
| --- | --- |
| **Student Information** | |
| **Name:** | Muhammad Talha, Hafiza Shujia Yousaf |
| **SAP ID:** | 46287, 47798 |
| **Section:** | 5-B |
| **Project Name:** | Banking Application |

**Project Report**

**Project Title: Banking Application**

* **Introduction:**

In a banking application, efficient data processing is crucial to ensure fast retrieval and manipulation of customer records. The application must handle frequent operations, such as searching for customer information and sorting transaction data. Choosing the right algorithms for searching and sorting can significantly improve the application’s performance and responsiveness. In this report, we will discuss the most suitable algorithms for these tasks, analyze their efficiency, and consider the consequences of using alternative algorithms. Detailed explanations of best, worst, and average case complexities will be provided to highlight the trade-offs between different algorithms.

**1. Searching Algorithms for Customer Records:**

To efficiently search customer records in a database, the goal is to find an algorithm with the lowest possible time complexity, ensuring that customer data retrieval is fast, especially with a large database.

* 1. **Hash Table:**
* **Description**: Hash tables use a hashing function to directly map keys (e.g., customer IDs) to specific indexes, allowing for near-instantaneous data retrieval.
* **Best Case Complexity**: *O(1)* - Direct lookup without collisions.
* **Average Case Complexity**: *O(1)* - Generally constant, as collisions are minimized with a good hashing function.
* **Worst Case Complexity**: O(n) - In case of hash collisions, particularly if poorly managed or with an uneven hash distribution.
* **Why Hash Table?** For unique keys, such as customer IDs, a hash table is ideal. It allows for *O(1)* average case retrieval, making it the fastest option for accessing individual records. Hash tables are used frequently in database indexing and are a staple in high-performance applications. In a banking context, where customer searches may happen constantly, the constant time lookup makes hash tables optimal.

**1.2. Binary Search Tree (BST):**

* + **Description**: A BST keeps data sorted, allowing for a balanced search by traversing nodes. Balanced variants, like AVL or Red-Black trees, ensure consistent performance.
  + **Best Case Complexity**: *O(1)* - Target is at the root of the tree.
  + **Average Case Complexity**: *O(log n)* - Average traversal time for balanced trees.
  + **Worst Case Complexity**: *O(log n)* for balanced trees; *O(n)* if unbalanced.
  + **Why Binary Search Tree?** BSTs are effective if the data is continuously modified, as they support both quick insertion and deletion operations in *O(log n)* time for balanced structures. In a dynamic database environment, BSTs can be helpful where range-based searches (e.g., fetching customers based on account balance ranges) are necessary.

**2. Sorting Algorithms for Customer Data:**

Sorting is often required after retrieving records, especially when displaying results in a specific order, such as by name or account balance. The following sorting algorithms are optimal choices based on efficiency and stability.

**1.1. Quick Sort:**

* **Description**: A divide-and-conquer algorithm that selects a pivot and partitions the list into elements less than and greater than the pivot.
* **Best and Average Case Complexity**: *O(n log n)* - Efficient partitioning with an ideal pivot.
* **Worst Case Complexity**: *O(n^2)* - Occurs with poor pivot choices (e.g., already sorted data with first or last element as pivot).
* **Why Quick Sort?** Quick Sort is highly efficient for in-memory sorting, handling large datasets quickly. Its average case complexity of *O(n log n)* makes it optimal for sorting customer data in real-time applications. It’s well-suited for large datasets typical in banking, although caution is needed for pivot selection to avoid *O(n^2)* behavior.

**1.2. Merge Sort:**

* **Description**: A stable, divide-and-conquer algorithm that splits the list in half, recursively sorts each half, and merges them.
* **Best, Average, and Worst Case Complexity**: *O(n log n)* - Consistent performance across cases.
* **Why Merge Sort?** Merge Sort provides consistent performance and is stable, meaning it preserves the order of equal elements. Stability is useful in banking, as it ensures consistency in the ordering of records with identical values. However, Merge Sort requires additional space for merging, making it slower for extremely large datasets compared to Quick Sort.
* **Impact of Using Alternative Searching Algorithms in Searching:**

**1.1. Linear Search:**

* **Description**: Sequentially checks each element until the target is found.
* **Best Case**: *O(1)* - Target is the first element.
* **Worst and Average Case**: *O(n)* - Entire list may need to be scanned.
* **Why Not Linear Search?** Linear search becomes inefficient for large databases as it requires traversing each record, resulting in poor performance with larger data sets. For customer records in a banking application, where searches are frequent, linear search could cause slowdowns, increasing user wait times.

**1.2. Binary Search on Sorted Array**

* **Description**: Divides the array in half repeatedly to locate the element.
* **Best Case**: *O(1)* - Target is in the middle.
* **Worst and Average Case**: *O(log n)*
* **Limitations**: While binary search is efficient on sorted data, it requires the array to remain sorted, which can be costly to maintain with frequent insertions or deletions. This makes binary search better suited for static datasets rather than dynamic banking records.
* **Impact of Using Alternative Searching Algorithms in Sorting:**
  1. **Bubble Sort**
  + **Description**: Repeatedly steps through the list, swapping adjacent elements if they are in the wrong order.
  + **Best Case**: *O(n)* - When the list is already sorted.
  + **Worst and Average Case**: *O(n^2)* - Requires multiple passes.
  + **Why Not Bubble Sort?** Bubble Sort is inefficient for large datasets due to its *O(n^2)* complexity, making it a poor choice for a banking application. Sorting large numbers of records with Bubble Sort would lead to significant delays, negatively impacting the user experience.

**1.2. Selection Sort:**

* **Description**: Selects the smallest element from the unsorted list and places it in the sorted portion.
* **Best, Worst, and Average Case Complexity**: *O(n^2)* - Inefficient for all cases.
* **Why Not Selection Sort?** Selection Sort has consistently high time complexity and does not improve with partially sorted data, making it impractical for high-performance applications. It would result in slow processing speeds in a banking application.

**1.3. Insertion Sort:**

* **Description**: Builds a sorted list one element at a time by inserting each element into its correct position.
* **Best Case**: *O(n)* - Efficient for nearly sorted data.
* **Worst and Average Case Complexity**: *O(n^2)*
* **Why Not Insertion Sort?** Insertion Sort is practical only for small or nearly sorted datasets, as its complexity scales poorly with larger datasets. In a banking application with numerous customer records, it would perform slower than Quick Sort or Merge Sort.
* **Conclusion:**

For a banking application, efficient searching and sorting are critical. Using **Hash Tables** for searching provides fast *O(1)* lookups, making it ideal for unique customer ID retrieval. For sorting, **Quick Sort** and **Merge Sort** are suitable for their fast average case performance and stability. Using inefficient algorithms like Linear Search, Bubble Sort, or Selection Sort would significantly degrade application performance, especially as the dataset grows. These choices ensure a responsive user experience and scalable performance, meeting the high demands of a banking system.