**Randomized Quicksort and Hashing with Chaining**

**Introduction**

The efficiency and scalability of algorithms significantly influence their practical usability. In this report, Randomized Quicksort and Hashing with Chaining were analyzed to assess their performance under various conditions. Randomized Quicksort was compared with Deterministic Quicksort to evaluate their efficiency across different array types, while the Hashing with Chaining implementation was analyzed for its collision-handling capabilities and the impact of load factors. This analysis aimed to deepen understanding of algorithm behavior and provide insights into their theoretical and empirical performance.

**Part 1: Randomized Quicksort Analysis**

**Rigorous Analysis of Time Complexity**

Randomized Quicksort was implemented by selecting a pivot uniformly at random from the subarray during each partition. The algorithm was analyzed using the recurrence *T(n)=T(k)+T(n−k−1)+O(n)* where k represents the size of the smaller partition (Fatima, 2023). On average, the partitions were balanced, resulting in *T(n)=2T(n/2)+O(n).* Solving this recurrence relation using the Master Theorem confirmed that the average-case time complexity was *O(n log n).*

This efficiency stemmed from the random pivot selection, which minimized the likelihood of unbalanced partitions and avoided the worst-case behavior *(O( n^2)* seen in Deterministic Quicksort with certain input arrays.

**Empirical Comparison**

The performance of Randomized Quicksort and Deterministic Quicksort was compared empirically using arrays of sizes 100, 1000, 5000, and 10,000. These arrays included randomly generated, already sorted, reverse-sorted, and arrays with repeated elements.

For smaller arrays (e.g., size 100), both algorithms performed comparably, with execution times in the microsecond range. Randomized Quicksort consistently maintained low execution times regardless of input distribution (Fatima, 2023). However, Deterministic Quicksort showed noticeable inefficiencies when handling already sorted and reverse-sorted arrays, particularly for larger input sizes. For instance, with an array size of 10,000, Deterministic Quicksort took 5.47 seconds for sorted input and 2.69 seconds for reverse-sorted input, while Randomized Quicksort completed the same tasks in just 0.02 seconds.

The differences became more pronounced with larger arrays. Randomized Quicksort exhibited stable performance across all input types due to its random pivot selection, which avoided the pitfalls of deterministic pivoting strategies. In contrast, Deterministic Quicksort’s reliance on the first element as the pivot resulted in unbalanced partitions for sorted and reverse-sorted arrays, leading to longer execution times.

For arrays with repeated elements, both algorithms performed slower as the size increased, with Deterministic Quicksort showing higher execution times due to its deterministic pivoting. Randomized Quicksort handled these arrays more efficiently, though the increased comparisons in repeated elements did lead to some delays (e.g., 4.72 seconds for an array of size 10,000).

**Discussion of Results**

The empirical results aligned well with the theoretical analysis. Randomized Quicksort’s *O(n log n)* average-case complexity was reflected in its consistent performance across different input types and sizes (Youvan, 2024). Deterministic Quicksort, however, showed **O(n^2)** behavior in scenarios with unbalanced partitions, such as sorted and reverse-sorted arrays. These discrepancies highlighted the importance of algorithm design in ensuring scalability and efficiency across diverse input conditions.

**Part 2: Hashing with Chaining**

**Rigorous Analysis of Operations**

A hash table with chaining was implemented using a universal hash function to distribute keys uniformly across buckets. This design efficiently supported insertion, search, and deletion operations (Fatima, 2023). The expected time for these operations was O(1+α), where α (load factor) is the ratio of elements to slots. Under the assumption of simple uniform hashing, α was minimized, ensuring near-constant time for all operations.

**Empirical Results**

The hash table’s performance was tested by inserting key-value pairs ("Alice", 25), ("Bob", 30), and ("Charlie", 35) into a table with 10 buckets. After all insertions, the buckets showed minimal collisions, confirming the effectiveness of the hash function (Fatima, 2023). Searching for keys yielded expected results (e.g., retrieving 25 for "Alice" and returning None for "David"). Deleting "Bob" correctly updated the hash table without affecting other entries.

**Discussion of Results**

The load factor was kept low (α=0.3) during the experiment, which contributed to the efficiency of operations. Strategies to maintain a low load factor included using a larger number of buckets and dynamically resizing the hash table when α\alphaα exceeded a predefined threshold. These strategies ensured minimal collisions and optimized performance.

**Conclusion**

The analysis of Randomized Quicksort and Hashing with Chaining demonstrated the critical role of algorithm design in achieving efficiency and scalability. Randomized Quicksort’s performance remained stable across diverse input types, validating its *O (n log n)* time complexity. In contrast, Deterministic Quicksort struggled with sorted and reverse-sorted arrays due to its deterministic pivoting strategy. Similarly, Hashing with Chaining showed near-constant operation times, emphasizing the importance of a low load factor and uniform hashing. This study highlighted the importance of theoretical and empirical evaluation in selecting algorithms suitable for practical applications.

**References**

Fatima, P. (2023). Optimizing Algorithm Efficiency through Advanced Data Structures in C++: A Comparative Analysis of Performance, Scalability, and Complexity. *International Journal of Computations, Information and Manufacturing (IJCIM)*, *3*(2), 66-72.

Youvan, D. C. (2024). Computational Sequences for Enhanced Efficiency: A Novel Approach to Data Handling, Security, and Performance Optimization.