Analysis, Comparisons, and Discussion of Results

Avinna Bhattarai

005028547

MSCS-532-A01

Assignment 3

Dr. Vanessa Ruth Cooper

09/15/2024

Analysis, Comparisons, and Discussion of Results

This report provides an in-depth analysis of two algorithms: Randomized Quicksort and Hashing with Chaining. The report includes theoretical analysis, empirical comparisons, and discussions of observed results. Implementations for these algorithms are provided in separate files in the repository.

**Implementation Files**

* Randomized Quicksort: ***randomized\_quicksort.py***
* Deterministic Quicksort Comparison: ***deterministic\_quicksort\_comparison.py***
* Hash Table with Chaining: ***hash\_table.py***

# Randomized Quicksort Analysis

* The Randomized Quicksort algorithm is implemented in ***randomized\_quicksort.py***. The algorithm works by selecting a pivot element uniformly at random from the subarray being partitioned, which helps in achieving better average-case performance compared to deterministic methods.

## Theoretical Analysis

**Average-Case Time Complexity:**

* Recurrence Relation: The time complexity of Randomized Quicksort can be analyzed using the recurrence relation:  
  where is the size of one of the partitions. On average, the pivot divides the array into two approximately equal parts, leading to:  
  This recurrence relation is solved using the Master Theorem, yielding **.**
* Indicator Random Variables: The pivot selection process is random, and the expected size of the partitions is balanced, which supports the average-case complexity of .

The Randomized Quicksort algorithm achieves time complexity on average due to the probabilistic choice of pivots, which ensures that the partitions are balanced, leading to efficient recursive sorting.

1. **Empirical Comparison**

The empirical comparison between Randomized Quicksort and Deterministic Quicksort is performed using ***deterministic\_quicksort\_comparison.py***. The following observations were made:

* Randomly Generated Arrays: Randomized Quicksort generally performed better than Deterministic Quicksort due to its ability to handle various input distributions more effectively.
* Already Sorted Arrays: Deterministic Quicksort showed significantly poorer performance on already sorted arrays due to its poor pivot choice strategy (always picking the first element).
* Reverse-Sorted Arrays: Like sorted arrays, Deterministic Quicksort performed worse on reverse-sorted arrays.
* Arrays with Repeated Elements: Both algorithms handled arrays with repeated elements reasonably well, but Randomized Quicksort maintained better overall performance.

1. **Discussion**

The empirical results align with the theoretical analysis that Randomized Quicksort should outperform Deterministic Quicksort in most cases. Discrepancies between theoretical expectations and empirical results can arise due to factors such as constant factors in the complexity or implementation-specific overheads.

# Hashing with Chaining

The hash table with chaining is implemented in ***hash\_table.py***. The hash table uses chaining for collision resolution and supports operations such as insert, search, and delete. A suitable hash function is used to minimize collisions.

## Theoretical Analysis

**Expected Time Complexity:**

* Insert, Search, Delete: Under the assumption of simple uniform hashing, the average time complexity for these operations is . The performance is influenced by the load factor α\alphaα, which is the ratio of the number of elements to the number of slots in the hash table.

**Effect of Load Factor:**

* Load Factor: As the load factor increases, the performance of hash table operations can degrade. A high load factor leads to more collisions, causing longer chains and thus slower operations.
* Dynamic Resizing: To maintain efficient performance, dynamic resizing of the hash table is implemented (not shown in the basic implementation). Resizing typically involves increasing the number of slots and rehashing existing elements to ensure a low load factor and efficient operation.
  1. **Discussion**

The hash table with chaining provides efficient average-case performance for insert, search, and delete operations. Keeping the load factor low is crucial for maintaining performance, and dynamic resizing is a common strategy to manage the load factor effectively.

Summary of Findings: The analysis and empirical testing of Randomized Quicksort and Hashing with Chaining have yielded insightful results. Randomized Quicksort consistently outperforms Deterministic Quicksort, particularly on various input distributions, due to its probabilistic pivot selection, which helps achieve better average-case performance. The theoretical analysis supports this observation with an average-case time complexity of , which aligns with the empirical results showing that Randomized Quicksort is generally more efficient across different types of arrays, including random, sorted, reverse-sorted, and those with repeated elements. On the other hand, Hashing with Chaining demonstrates efficient average-case performance for insert, search, and delete operations, with an average time complexity of under simple uniform hashing. The performance of the hash table is influenced by the load factor, and strategies such as dynamic resizing are crucial for maintaining operational efficiency by managing collisions and keeping the load factor low. Overall, the theoretical and empirical analyses confirm that both Randomized Quicksort and Hashing with Chaining are effective algorithms when applied under the right conditions, highlighting the importance of choosing appropriate algorithms and data structures based on specific performance needs.

(Last Name, Year)

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). The MIT Press.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

Knuth, D. E. (1998). *The art of computer programming: Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.