Richa Lahoti
005039141
MSCS-532-M20
Assignment 3

Part 1: Randomized Quicksort Analysis

Randomized Quicksort Algorithm Implementation

The randomized quicksort algorithm implementation is in the github repo linked here: https://github.com/rlahoti-cb/MSCS532_Assignment3 . This implements randomized quicksort where the pivot element is chosen uniformly at random from the subarray that is being partitioned. The README.md covers how to run this code to generate the data that is used in the analysis section below. Note, that your numbers may vary slightly due to running the code on a different machine and other variables.

Analysis

We can begin by providing a rigorous analysis of the average-case time complexity of randomized quicksort. This will be a key reference to section 7.4.2 which discusses the expected runtime of the randomized quicksort algorithm (Cormen, Leiserson, Rivest, & Stein, 2009). In the randomized quicksort variant, there will be O(log n) levels that the algorithm will recurse. At each level, there will be O(n) work done. Given that we are using randomized pivot selection, we know that there will be a constant fraction of elements on one side of the partition, for every level of recursion. This results in a runtime of O(n log n) for the randomized quicksort algorithm.

Comparison

I conducted a real world analysis that compares the runtime of the randomized quicksort and deterministic quicksort algorithms on various types of input data. The input data included random data, sorted data, reversed sorted data, and repeated random data. Here is a table of the results of the execution of the code. The code outputs the results to the terminal and I formatted the output into this table for easier viewing.

Input Data Type	Pivot Type	Time Taken (s)
random	random	0.019147158
sorted	random	0.019935131
reversed sorted	random	0.019943237
repeated random	random	1.978361130

random	deterministic	0.02201390266
sorted	deterministic	2.078945637
reversed sorted	deterministic	3.607861042
repeated random	deterministic	1.302836895

We can compare both the random and deterministic quicksort algorithms for each input data type that was used.

For random input data, the results were fairly similar. We note that the real-world data supports the idea that pivot selection does not significantly affect the performance on randomized data. This implies that regardless of the pivot selection approach both pivot selection approaches result in the similar runtime which matches the average case of O(n log n).

For sorted input data, we note that the deterministic pivot selection approach (2.0789s) performs much worse than the randomized pivot selection approach (0.0199s). This aligns with our theoretical expectation that if a bad pivot is chosen, this will lead to the worst-case runtime of $O(n^2)$. A poor pivot selection (through deterministic pivot selection) creates highly imbalanced partitions. Randomized pivot selection prevents this imbalance which leads to the average case runtime of O(n log n).

For reverse sorted input data we again note that the deterministic pivot selection approach (3.6079s) performs much worse than the randomized pivot selection approach (0.0199s). This again aligns with our theoretical expectation that if a bad pivot is chosen, this will lead to the worst-case runtime of $O(n^2)$. Again, a poor pivot selection (through deterministic pivot selection) creates highly imbalanced partitions. Randomized pivot selection prevents this imbalance which leads to the average case runtime of O(n log n).

For random repeated elements, we see that the randomized pivot selection (1.9784s) takes longer than the deterministic pivot selection (1.3028s). I would have expected that for random repeated elements, randomized pivot selection would theoretically perform better than deterministic pivot selection.

In summary, randomized quicksort generally performs the same across input types. Deterministic quicksort on the other hand is incredibly sensitive to the type of data that we are operating on (especially due to heavily imbalanced partitions leading to poor performance).

Part 2: Hashing with Chaining

Implementation

The hash table with chaining implementation is in the github repo linked here: https://github.com/rlahoti-cb/MSCS532_Assignment3 . This implements a hash table with chaining for collisions (GeeksforGeeks, 2025). The README.md covers how to run this code to generate the data that is used in the analysis section below. Note, that your numbers may vary slightly due to running the code on a different machine and other variables.

Analysis

In this section, we will analyze the insertion, search, and deletion times of data in a hash table on the assumption of simple uniform hashing. We will also take a look at how the load factor can affect these times.

In order to run this experiment, I ended up creating a hash table of 10,000 elements for each load factor. This allows us to empirically compare the performance of the three operations (insert, search, delete) across various load factors.

Here is the data collected which shows the performance of the insertion, search, and deletion operations on a hash table with chaining across load factors of 0.1, 0.25, 1.0, 2.5, and 5.0.

Load Factor	Insert Time (s)	Search Time (s)	Delete Time (s)
0.1	0.0025880	0.0024178	0.0004911
0.25	0.0109739	0.0109539	0.0010178
1	0.1985040	0.1910610	0.0043910
2.5	1.3875043	1.3630888	0.0140102
5	8.2866299	9.4632919	0.0418370

The data clearly shows that as the load factor goes up, the time it takes each operation increases. We see noticeable increases for both insert and delete for example when we go from a load factor of 2.5 to 5.0. This shows that as the load factor goes up (the number of elements in each bucket), it takes a *lot* longer to insert and search. This makes sense from a theoretical perspective – as the load factor goes up, that means that there are more elements in each bucket which means that for a bucket that operates as a linked list or an array, the time to insert or delete goes up.

In order to maintain a low load factor, we can dynamically resize the hash table (GeeksforGeeks, 2023). This involves increasing the number of buckets of the table (such as doubling the number of buckets) and then rehashing all of the data to fit into the new table. This allows us to reduce the load factor of the hash table and improve the performance of the operations.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

GeeksforGeeks. (2023, March 28). Load factor and rehashing.
https://www.geeksforgeeks.org/load-factor-and-rehashing/

GeeksforGeeks. (2025, March 4). Separate chaining collision handling technique in hashing.
https://www.geeksforgeeks.org/dsa/separate-chaining-collision-handling-technique-in-hashing/

GeeksforGeeks. (2023, September 14). Quicksort using random pivoting.
https://www.geeksforgeeks.org/dsa/quicksort-using-random-pivoting/