Name: Brendan Yeong

Student ID: 005025797

MSCS 532

Assignment 3

Part 1: Randomized Quicksort Analysis

2. Analysis

Let’s look at why the average-case time complexity of Randomized Quicksort is by examining the steps of the algorithm:

Partitioning and recursive calls: In each level of recursive call, the array is divided into two subarrays. This step takes time, where is the number of elements in the array.

Random pivot selection: The pivot gets chosen randomly which is different than then standard quick sort. This step makes sure that the subarrays are divided into two roughly equal lengths. The recurrence relation for the average-case running time is:

Using the master theorem, we can solve the recurrence to give us .

Indicator random variables:

This method also allows us to analyze the average number of comparisons by calculating the expected number of comparisons during the partitioning process.

Comparison

if and only if or is chosen as a pivot before any element between them is chosen as a pivot. The probability that and are compared is , as there are elements between and , and pivot point is chosen as random.

Therefore, the expected number of comparisons,

This shows that the expected number of comparisons in Randomized Quicksort is , which relates to the overall time complexity of the algorithm.

3. Comparison

A graph with numbers and dots

Description automatically generated

Observations:

In this graph we can clearly see that with randomizing most of the time the run time in seconds is slower.

Random generated arrays: Both the randomized and deterministic quicksort perform similarly in my computer but in theory if the datasets were to go even larger, the randomized quicksort should have a slight advantage due to better average pivot selection.

Sorted in ascending: The deterministic quicksort performs poorly here with a due to poor pivot selection, and the randomized maintains at due to random pivot selection. We can see that the data points at the graph clumps together with the others.

Sorted in descending: This is almost like the ascending array where the deterministic degrades to while randomized maintains it’s efficiency. We can see that in the graph data point.

Random with repeated elements arrays: Both algorithms perform almost the same but with an even larger array the randomized algorithm does have a slight advantage on average due to a better average pivot selection.

The difference in empirical results mainly stems from pivot selection. While Deterministic Quicksort can perform in cases like already sorted or reverse-sorted arrays, Randomized Quicksort avoids this scenario by randomly selecting pivots.

Part 2: Hashing with Chaining.

Analysis

1. Expected time complexity of each operation:

* Search: on average. However, with the case of chaining, insertions could be in constant time but in rare cases where keys are hash to the same index, this would cause a degrade in performance.
* Insert: on average, but worst case could be degraded to if all keys collide into the same bucket.
* Delete: on average for deleting from a small chain but degrade to if all elements collide.

2. Load factor affects

Hash table T with m slots that stores n elements with the load factor  plays a crucial role in the performance of the hash table.

A higher load factor, means more collision as there are more elements per bucket and can degrade the performance of searches, inserts, and deletions operations.

A low load fact around means that fewer elements per slot which ensures that most operations remain complexity on average.

The performance of the hash table degrades as the load factor increases, with each operation needs to traverse a chain (for search, insert, or delete) will take more time as the chains grow longer.

3. Strategies for minimizing collisions

To maintain efficient performance, we must keep the load factor low. So here is two strategies:

When the load factor increases, we can do dynamic resizing by resizing the hash table doubling and rehash all existing elements. Doing this would maintain low collision probabilities and keeps the average time operations at . Another strategy is to choose a good hash function. A well-design hash functions will help ensure that keys are distributed uniformly across the table and reducing the occurrence of a collision.

Reference

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>

Deterministic and randomized quicksort. (n.d.). https://people.engr.tamu.edu/andreas-klappenecker/csce411-s15/csce411-random3.pdf

GeeksforGeeks. (2023, January 10). *Implementation of hashing with chaining in python*. https://www.geeksforgeeks.org/implementation-of-hashing-with-chaining-in-python/