MSCS532_ASSIGNMENT 3

DATE: 10TH NOVEMBER 2024

Shafan Nazeer Ahmed
005030047

Randomized Quicksort Analysis

- A pivot element is selected at random from the array by Randomized Quicksort.
- It divides the array into two subarrays, one containing elements that are smaller than the pivot and the other containing elements that are larger.
- The partition step compares each array element to the pivot in each recursive call, taking O(n) time for an array of size n.

Recurrence Relation

For an array of size n, let T(n) show how long Randomized Quicksort takes to run. There are two recursive calls that deal with subarrays whose sizes depend on where the pivot is. The partition step takes O(n) time. To find the recurrence relation, let's say that the pivot evenly splits the array in the best case scenario.

$$T(n)=T(n/2)+T(n/2)+O(n)$$

$$T(n)=2T(n/2)+O(n)$$

Applying the Master Theorem

Then it becomes,

T(n)=aT(n/b)+O(n^d)

Where:

- a = 2 (since the problem is divided into two subarrays),
- b = 2 (the array is halved in each step),
- d = 1 (the partition step takes O(n) time).

a = b^d

So solution to *Recurrence* relation is

T(n)=O(nlogn)

COMPARISON

1. Deterministic Quicksort Implementation

The difference between Deterministic and the Randomized quicksort is that, in the Deterministic version the first element is always chosen as the pivot.

Below is the implementation of Deterministic Quicksort:-

```
def deterministic_partition(A, low, high):
    pivot = A[low]  # Always choose the first element as the pivot
    i = low + 1
    j = high
    while True:
        while i <= j and A[i] <= pivot:
            i += 1
        while A[j] >= pivot and j >= i:
            j -= 1
        if j < i:
            break
        A[i], A[j] = A[j], A[i]
    A[low], A[j] = A[j], A[low]
    return j

def deterministic_quicksort(A, low, high):
    if low < high:
        pivot_idx = deterministic_partition(A, low, high)
```

```
    deterministic_quicksort(A, low, pivot_idx - 1)
    deterministic_quicksort(A, pivot_idx + 1, high)
```

2. Randomized Quicksort Implementation

```
import random

def randomized_partition(A, low, high):
    pivot_idx = random.randint(low, high)
    A[pivot_idx], A[high] = A[high], A[pivot_idx]
    return partition(A, low, high)

def randomized_quicksort(A, low, high):
    if low < high:
        pivot_idx = randomized_partition(A, low, high)
        randomized_quicksort(A, low, pivot_idx - 1)
        randomized_quicksort(A, pivot_idx + 1, high)
```

3. Time Measurement

Using Python's time module to measure the running time for both sorting methods.

```
import time

# Helper function to measure execution time

def measure_time(algorithm, A):

    start_time = time.time()

    algorithm(A, 0, len(A) - 1)

    return time.time() - start_time
```

4. Test Arrays

Here are the different types of arrays

1. Randomly Generated Arrays

```
random_array = [random.randint(0, 10000) for _ in range(1000)]
```

2. Already Sorted Array

```
sorted_array = list(range(1000))
```

3. Reverse Sorted Array

```
reverse_sorted_array = list(range(1000, 0, -1))
```

4. Arrays with Repeated Elements

repeated_elements_array = [5] * 500 + [10] * 500

Hashing With Chaining

Search Time

The linked list length at the key hashing index determines search time. The optimal scenario involves traversing the entire list without the key, with a time complexity of $O(1 + \alpha)$, where α is the load factor (the ratio of elements to slots).

Insert Time

The insert time is like search time, as it necessitates the traversal of the linked list to identify the position for the new key value pair. The time complexity is $O(1 + \alpha)$.

Delete Time

The delete time is comparable to the search time, as it necessitates traversing the linked list to identify the key value pair that is to be deleted. The time complexity is $O(1 + \alpha)$.

The Load Factor

The performance of these operations is significantly influenced by the load factor (α). The length of the linked lists at each index increases as the load factor increases, resulting in longer search, insert, and delete times. The time complexity approaches O(n) as the load factor approaches 1, where n is the number of elements in the hash table. Also a higher load factor increases the probability of collisions, which can result in slower performance and longer linked lists.

Strategies for maintaining a low load factor

Below are few strategies maintain low load factor:-

- The hash table is resized when the load factor surpasses a specific threshold. This includes the rehashing of current key value pairs and their redistribution across the new expanded table.
- Can opt for open addressing over chaining. This method investigates additional indices in the table to identify an empty slot, which can enhance performance and minimize collisions.
- Utilize hash functions from universal hash function families to reduce the load factor and minimize collisions.