2024

Assignment 6: Medians and Order Statistics & Elementary Data Structures

- NAME: **ABDUL RAHEMAN GOTORI**
- STUDENT ID: **005029919**

- COURSE & TITLE: **ALGORITHMS AND DATA STRUCTURES (MSCS-532-A01)**
- DATE: **29ND SEPTEMBER 2024**

Table of Contents

Selection Algorithms

PERFORMANCE ANALYSIS

1. Time complexity

Deterministic Algorithm

Time Complexity = O(n) in the worst case.

The algorithm ensures that the pivot divides the array into fairly balanced partitions, ensuring linear time even in the worst-case scenario.

Randomised Algorithm

Time Complexity = O(n).

The algorithm performs well in the average case because the pivot is chosen at random. The worst-case time complexity is O(n^2), but this occurs with a low probability.

2. Space Complexity

Deterministic algorithm

Due to the recursive calls and the creation of sublists the space complexity is O(n).

Randomised Algorithm

Space Complexity = O(n), the same as deterministic algorithms.

3. Additional Overheads

Deterministic algorithm

Sorting each group of five elements introduces new constant factors, but it remains linear overall.

Randomised Algorithm

There is little additional overhead because pivot selection is random and does not require sorting.

EMPIRICAL ANALYSIS

To compare the performance of both algorithms its run on different input sizes and distributions. For empirical analysis, I considered using larger datasets to account for variability, especially for the randomized algorithm.

```
import time

import random

def test_selection_algorithms():
```

```
# Generate different types of input arrays
random_array = random.sample(range(1, 1000000), 10000)
sorted_array = list(range(1, 10001))
reverse_sorted_array = list(range(10000, 0, -1))
duplicate_array = [5] * 10000 + list(range(1, 1001))

test_cases = {
    "Random": random_array,
    "Sorted": sorted_array,
    "Reverse Sorted": reverse_sorted_array,
    "With Duplicates": duplicate_array
}

k = 5000  # Middle element

for name, arr in test_cases.items():
    print(f"\nTest Case: {name}")

    # Test Deterministic Algorithm
    start_time = time.time()
    deterministic_result = deterministic_select(arr, k)
    deterministic_time = time.time() - start_time
```

```
    print(f"Deterministic Select: Element={deterministic_result},
Time={deterministic_time:.4f} seconds")


    # Test Randomized Algorithm
    start_time = time.time()
    randomized_result = randomized_select(arr, k)
    randomized_time = time.time() - start_time
    print(f"Randomized Select: Element={randomized_result}, Time={randomized_time:.4f}
seconds")


if __name__ == "__main__":
   test_selection_algorithms()
```

Expected results:-

- Deterministic Algorithm - All test cases should run linearly.
- Randomized Algorithm - Should perform well, but pivot selection may affect performance.

Elementary Data Structures

PERFORMANCE ANALYSIS

Time Complexity

1. Arrays

- Time Complexity = O(n)
- Deletion

  At Specific Index = O(n) due to shifting of elements.
- Access

  By Index = O(1).

2. Matrices

- Insertion, Deletion, Access = O(1).
- O(rows×cols).

3. Stacks

- Push = O(1).
- Pop = O(1).
- Peek = O(1).
- Size Check = O(1).

4. Queues

- Enqueue = O(1).
- Dequeue = O(n)
- Front = O(1).
- Size Check = O(1).

5. Singly Linked Lists

- Insertion
    - At Beginning = O(1)O(1)O(1).
    - At End = O(n)
- Deletion
    - By Key = O(n).
- Traversal = O(n)O(n)O(n).
- Access = O(n)O(n)O(n).

2. Space Complexity

- Arrays = O(n) for storage.
- Matrices = O(rows×cols).
- Stacks and Queues = O(n).
- Singly Linked Lists = O(n).

DISCUSSION & PRACTICAL APPLICATIONS

Arrays are great for managing static data and lookup tables that need to access the index often. For fast manipulation of image data stored in a 2D array their constant time access is ideal. Insertions and deletions at random positions are inefficient in arrays because shifting elements takes linear time.

Linked lists are better for frequent insertions and deletions, like systems programming dynamic memory allocation or media player playlist management. They perform better when they can add or remove elements without shifting the structure. Pointer storage and inefficient random access increase memory usage in linked lists, making them unsuitable for index-based element access applications.

Stacks' Last-In-First-Out (LIFO) behavior makes them essential for managing function calls in programming languages, implementing undo mechanisms in software applications, and evaluating expressions in compilers and calculators. First-In-First-Out (FIFO) queues are crucial for operating system task scheduling, printer print job handling, and graph algorithm breadth-first search (BFS).

The application requirements determine whether arrays or linked lists are used for stacks and queues. Dynamic arrays simplify stack implementations and improve cache performance, allowing constant-time push and pop operations.

Array based queues can take a linear amount of time to dequeue if they are not optimized with

data structures like collections.deque. Linked lists let you change the size of a queue and add and remove items from it at constant time but they need more pointer memory and take longer to traverse.

In conclusion, these data structures have many practical applications, each with unique benefits for specific problem domains. Memory usage, speed, and implementation ease can affect software system performance and efficiency when choosing a data structure. Understanding these basic data structures prepares you for complex algorithms and scalable, high-performance applications.