Assignment 6: Medians and Order Statistics & Elementary Data Structures

Overview:

Part 1: Implementation and Analysis of Selection Algorithms

Deterministic algorithm

```python
## Deterministic Selection
def deterministic_select(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k - 1]

    # Step 1: Divide into groups of five
    groups = [arr[i:i + 5] for i in range(0, len(arr), 5)]
    medians = [sorted(group)[len(group) // 2] for group in groups]

    # Step 2: Recursively find the median of medians
    pivot = deterministic_select(medians, len(medians) // 2 + 1)

    # Step 3: Partition the array
    low = [x for x in arr if x < pivot]
    high = [x for x in arr if x > pivot]
    equals = [x for x in arr if x == pivot]

    # Step 4: Determine which part contains the k-th element
    if k <= len(low):
        return deterministic_select(low, k)
    elif k <= len(low) + len(equals):
        return pivot
    else:
        return deterministic_select(high, k - len(low) - len(equals))
```

Randomized Selection Algorithm

```python
#Randomized Algorithm
import random

def randomized_select(arr, k):
    if len(arr) == 1:
        return arr[0]

    # Step 1: Randomly select a pivot
    pivot = random.choice(arr)

    # Step 2: Partition the array
    low = [x for x in arr if x < pivot]
    high = [x for x in arr if x > pivot]
    equals = [x for x in arr if x == pivot]

    # Step 3: Determine which part contains the k-th element
    if k <= len(low):
        return randomized_select(low, k)
    elif k <= len(low) + len(equals):
        return pivot
    else:
        return randomized_select(high, k - len(low) - len(equals))

```

2. Performance Analysis

Time Complexity:

- Deterministic Algorithm:
    - Worst-case: O(n), achieved by selecting a pivot that divides the array into roughly equal parts. This is achieved through the Median of Medians approach, which ensures balanced partitions.

- Randomized Algorithm:
    - Expected-case: O(n), due to the probability of selecting a good pivot.
    - Worst-case: O(n^2), occurs when the pivot divides the array poorly in multiple recursive calls.

Space Complexity:

- Both algorithms are O(n) due to the additional memory used during partitioning.

- Deterministic selection has slightly higher constant overhead due to the computation of medians (Cormen et al, 2009).

3. Empirical Analysis:

Testing on Random Array:

Deterministic Selection Result Time:	0.083853 seconds
Randomized Selection Result Time:	0.033855 seconds

Testing on Sorted Array:

Deterministic Selection Result Time:	0.052135 seconds
Randomized Selection Result Time:	0.054980 seconds

Testing on Reverse Sorted Array:

Deterministic Selection Result Time:	0.062798 seconds
Randomized Selection Result Time:	0.050198 seconds

Observations:

1. Random Array:
    - The randomized algorithm's faster execution shows its efficiency when pivot selection works well without significant imbalance in partitions.
    - The deterministic algorithm takes longer due to the overhead of computing the median-of-medians pivot.
2. Sorted and Reverse-Sorted Arrays:
    - The deterministic algorithm maintains stable performance because its pivot selection mechanism avoids pathological cases like always selecting the smallest or largest element as the pivot.
    - Randomized selection also performs well, though it is slightly slower on sorted arrays compared to random arrays. This is likely due to occasional poor random pivot choices.

The deterministic algorithm ensures balanced partitions, achieving O(n) time complexity in all cases.

The randomized algorithm achieves O(n) on average but may encounter suboptimal partitions leading to O(n^2 in the worst case). However, the observed results demonstrate its robustness, as the likelihood of consistently poor pivots is low.

The empirical results validate the theoretical expectations; the Randomized algorithm is generally faster due to its simplicity and probabilistic efficiency.

Part 2: Elementary Data Structures Implementation and Discussion

Instructions:

1. Implementation:

```python
# Array
class Array:
    def __init__(self):
        self.data = []

    def insert(self, index, value):
        if 0 <= index <= len(self.data):
            self.data.insert(index, value)
        else:
            print("Index out of bounds.")

    def delete(self, index):
        if 0 <= index < len(self.data):
            return self.data.pop(index)
        else:
            print("Index out of bounds.")
            return None

    def access(self, index):
        if 0 <= index < len(self.data):
            return self.data[index]
        else:
            print("Index out of bounds.")
            return None
```

```python
# Matrix
class Matrix:
    def __init__(self, rows, cols):
        self.data = [[0] * cols for _ in range(rows)]

    def insert(self, row, col, value):
        if 0 <= row < len(self.data) and 0 <= col < len(self.data[0]):
            self.data[row][col] = value
        else:
            print("Index error.")

    def access(self, row, col):
        if 0 <= row < len(self.data) and 0 <= col < len(self.data[0]):
            return self.data[row][col]
        else:
            print("Index error.")
            return None

    def display(self):
        for row in self.data:
            print(row)
```

```python
# Stack
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if self.stack:
            return self.stack.pop()
        else:
            print("Stack is empty.")
            return None

    def peek(self):
        if self.stack:
            return self.stack[-1]
        else:
            print("Stack is empty.")
            return None

    def is_empty(self):
        return len(self.stack) == 0

```

```python
# Queue
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, value):
        self.queue.append(value)

    def dequeue(self):
        if self.queue:
            return self.queue.pop(0)
        else:
            print("Queue is empty.")
            return None

    def is_empty(self):
        return len(self.queue) == 0

```

2. Performance Analysis:

Time Complexity of Basic Operations

- Arrays

    - Insertion:
        - At a specific index: O(n) (shifting elements).
        - At the end: O(1)
    - Deletion:
        - From a specific index: O(n) (shifting elements).
    - Access: O(1) (direct indexing).
- Matrices
    - Insertion: O(1) (direct access via row and column indices).
    - Access: O(1)
- Stacks
    - Push: O(1)
    - Pop: O(1)
    - Peek: O(1)
- Queues
    - Enqueue: O(1) if implemented with dynamic resizing; O(n) if shifting is required for dequeueing.
    - Dequeue: O(n) with shifting; O(1) with circular buffer or linked list implementation.
- Singly Linked Lists
    - Insertion at the beginning: O(1)
    - Insertion at the end: O(n) without a tail pointer, O(1) with a tail pointer.
    - Deletion: O(n) (search for the element).
    - Traversal: O(n) (Weiss, 2011)

Trade-Offs Between Arrays and Linked Lists for Stacks and Queues

- Memory Usage:
    - Arrays: Require contiguous memory, which may cause allocation failures for large stacks or queues.
    - Linked Lists: Dynamically allocate memory, which allows flexible size adjustments but includes memory overhead for storing pointers.

- Performance:
    - Arrays:
        - Faster for stacks due to constant-time push and pop operations.
        - For queues, shifting elements during dequeue can be slow (O(n)O(n)O(n)) unless a circular buffer is used.
    - Linked Lists:
        - Provide constant-time enqueue and dequeue operations for queues without requiring shifting.
- Implementation Complexity:
    - Arrays: Simple to implement.
    - Linked Lists: More complex due to pointer manipulation (Knuth, 1997)

Efficiency of Different Data Structures in Specific Scenarios

- Arrays:
  Best suited for applications where random access is frequent, such as indexing large datasets or implementing look-up tables.
- Matrices:
  Ideal for numerical computations, image processing, or any application requiring multi-dimensional data representation.
- Stacks:
    - Use arrays when memory requirements are predictable and operations are limited to the stack's top.
    - Use linked lists for dynamic stacks with unpredictable sizes.
- Queues:
    - Arrays are efficient for queues in fixed-size scenarios or with a circular buffer for O(1) enqueue/dequeue.
    - Linked lists are better for dynamic queues with variable sizes.
- Singly Linked Lists:
  Suitable for applications requiring frequent insertions and deletions, such as implementing undo features in text editors or chaining in hash tables.

3. Discussion:

Practical Applications

Arrays and Matrices

Arrays and matrices are fundamental data structures widely used in various domains due to their simplicity and efficiency in indexing operations. Arrays are commonly applied in scenarios requiring sequential storage and direct access, such as storing lookup tables, managing fixed-size datasets, and implementing heaps for priority queues. Matrices are indispensable in fields like scientific computing, image processing, and machine learning for representing multi-dimensional data, such as grids, adjacency matrices in graph representations, and weight matrices in neural networks (Weiss, 2011)..

On the other hand, arrays require contiguous memory, which may limit their use in systems with constrained memory. Their fixed size can be a disadvantage in dynamic scenarios, where linked lists may be a better option

Stacks

Stacks are pivotal in applications requiring a last-in-first-out (LIFO) approach. They are used in function call management (recursion), syntax parsing, backtracking algorithms, and implementing undo operations in software. Arrays often back stack implementations when memory usage is predictable, as they provide faster operations due to lower overhead. Linked lists are more suitable for dynamic stacks where the size is not known in advance (Weiss, 2011).

Queues

Queues operate on a first-in-first-out (FIFO) basis, have applications like task scheduling , for example CPU job scheduling, message passing, breadth-first search (BFS) in graphs, and real-time data buffering like for example in streaming services. Circular queues implemented with arrays are efficient for fixed-size buffers, such as in network routers. Linked lists are favored for dynamic queue implementations, such as event-driven architectures, where the size of the queue fluctuates(Weiss, 2011) .

Singly Linked Lists

Linked lists excel in applications requiring dynamic memory usage and frequent updates, such as implementing hash table chaining, dynamic graph structures, or software undo features. Their ability to efficiently insert and delete elements makes them ideal for use cases like dynamic memory allocation in operating systems (Weiss, 2011).

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*

Weiss, M. A. (2011). *Data Structures and Algorithm Analysis in Java*. Pearson.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.