11/23/2024

Assignment 6: Medians and Order Statistics & Elementary Data

Rutu Shah
005026421

Table of Contsnts

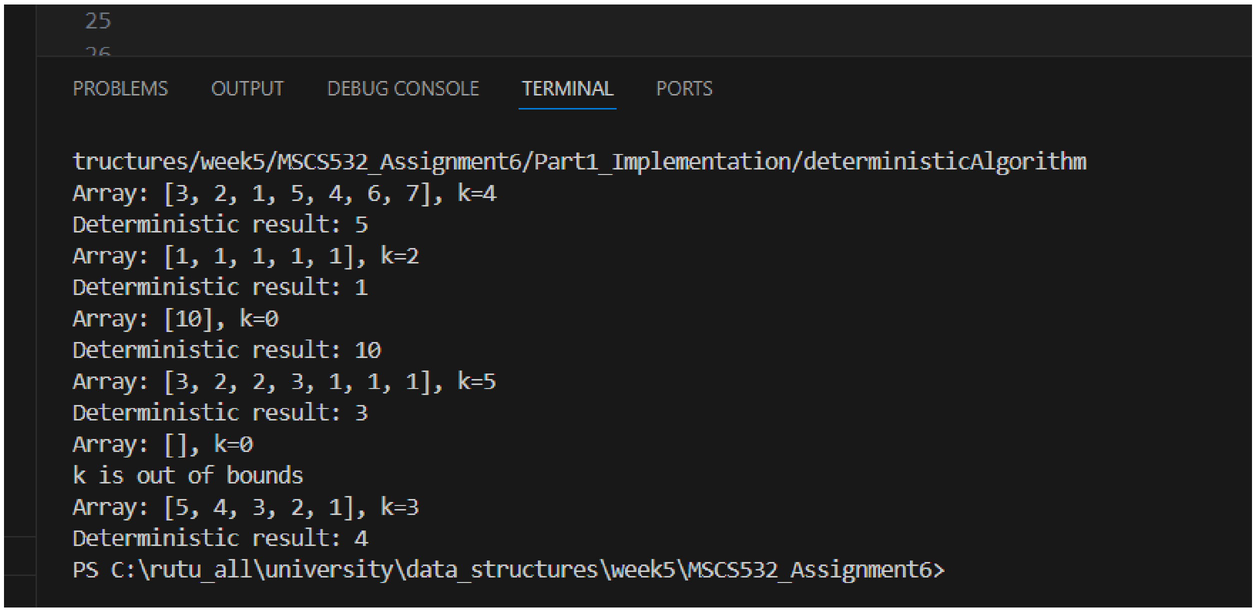
Part 1: Implementation and Analysis of Selsection Algorithms

1. Implementation

1.1 Implementation of the deterministic algorithm for selection in worst-case linear time (e.g., Median of Medians).

```python
def median_of_medians(arr, k):
    if k < 0 or k >= len(arr):
        raise ValueError("k is out of bounds")

    if len(arr) <= 5:
        return sorted(arr)[k]

    # Divide the array into groups of 5 and find the medians
    medians = [sorted(arr[i:i + 5])[len(arr[i:i + 5]) // 2] for i in range(0, len(arr), 5)]

    # Find the median of the medians
    pivot = median_of_medians(medians, len(medians) // 2)

    # Partition the array around the pivot
    low = [x for x in arr if x < pivot]
    high = [x for x in arr if x > pivot]
    pivots = [x for x in arr if x == pivot]

    if k < len(low):
        return median_of_medians(low, k)
    elif k < len(low) + len(pivots):
        return pivots[0]
    else:
        return median_of_medians(high, k - len(low) - len(pivots))


# Test cases
test_arrays = [
    ([3, 2, 1, 5, 4, 6, 7], 4),  # Normal case
    ([1, 1, 1, 1, 1], 2),        # Array with duplicates
    ([10], 0),                   # Single element array
    ([3, 2, 2, 3, 1, 1, 1], 5),  # Duplicates and unsorted
    ([], 0),                     # Empty array
    ([5, 4, 3, 2, 1], 3)         # Reversed array
]

for arr, k in test_arrays:
    try:
        print(f"Array: {arr}, k={k}")
        print(f"Deterministic result: {median_of_medians(arr, k)}")
    except ValueError as e:
        print(e)
```

Output:



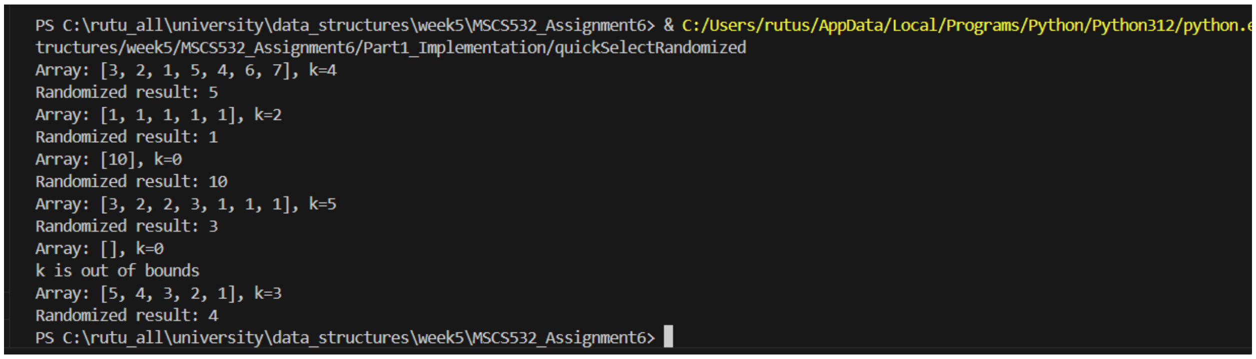
```
tructures/week5/MSCS532_Assignment6/Part1_Implementation/deterministicAlgorithm
Array: [3, 2, 1, 5, 4, 6, 7], k=4
Deterministic result: 5
Array: [1, 1, 1, 1, 1], k=2
Deterministic result: 1
Array: [10], k=0
Deterministic result: 10
Array: [3, 2, 2, 3, 1, 1, 1], k=5
Deterministic result: 3
Array: [], k=0
k is out of bounds
Array: [5, 4, 3, 2, 1], k=3
Deterministic result: 4
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6>
```

1.2 Implementation of randomized algorithm for selection in expected linear time (e.g., Randomized Quickselect).

```
import random


def randomized_quickselect(arr, k):
    if k < 0 or k >= len(arr):
        raise ValueError("k is out of bounds")


    if len(arr) == 1:
        return arr[0]


    # Choose a random pivot
    pivot = random.choice(arr)


    # Partition the array around the pivot

    low = [x for x in arr if x < pivot]

    high = [x for x in arr if x > pivot]

    pivots = [x for x in arr if x == pivot]


    if k < len(low):

        return randomized_quickselect(low, k)

    elif k < len(low) + len(pivots):

        return pivots[0]

    else:

        return randomized_quickselect(high, k - len(low) - len(pivots))


# Test cases

test_arrays = [

    ([3, 2, 1, 5, 4, 6, 7], 4),  # Normal case

    ([1, 1, 1, 1, 1], 2),         # Array with duplicates

    ([10], 0),                    # Single element array

    ([3, 2, 2, 3, 1, 1, 1], 5),  # Duplicates and unsorted

    ([], 0),                      # Empty array

    ([5, 4, 3, 2, 1], 3)          # Reversed array

]
```

```
for arr, k in test_arrays:
    try:
        print(f"Array: {arr}, k={k}")
        print(f"Randomized result: {randomized_quickselect(arr, k)}")
    except ValueError as e:
        print(e)
```

Output:



```
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.e
tructures/week5/MSCS532_Assignment6/Part1_Implementation/quickSelectRandomized
Array: [3, 2, 1, 5, 4, 6, 7], k=4
Randomized result: 5
Array: [1, 1, 1, 1, 1], k=2
Randomized result: 1
Array: [10], k=0
Randomized result: 10
Array: [3, 2, 2, 3, 1, 1, 1], k=5
Randomized result: 3
Array: [], k=0
k is out of bounds
Array: [5, 4, 3, 2, 1], k=3
Randomized result: 4
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> 
```

2. Performance Analysis

2.1 Time Complexity:

2.1.1 Deterministic algorithms (Median of Medians):

In deterministic algorithms, the median of medians is guaranteed to have O(n) time complexity in the worst case. Below are the pointers of why?

- The input array is divided into 5 groups, and the median of each group is calculated. Sorting a group of 5 items takes constant time () of O (1) per group), and there are n/5 groups. Hence, this step takes O(n).

- The algorithm selects median of the medians recursively. The problem size is lowered to n/5 and this recursive search is done with T(n/5).
- Once the pivot (median of medians) is computed, array gets split into smaller than, equal to, and larger than pivot. Partitioning takes O(n).
- The algorithm chooses recursively the K^{th} element in the left partition (or the right partition, depending on k), where maximum partition size is 7n/10 (since minimum 30% of elements are deleted in each operation).

Hence the recurrence relation is calculated as follows,

- $T(n) = T(n/5)+T(7n/10)+O(n)$

Solving the recurrence using the substitution method or the recursion tree approach gives

- $T(n) = O(n)$

Best Case and Average Case: Even in the best case, the algorithm follows the same steps, making the time complexity O(n).

2.1.2 Randomized Algorithm for Quick Select:

The expected time complexity of randomized Quick select is O(n):

- A random pivot is chosen, and the array is partitioned into elements less than, equal to, and greater than the pivot. Partitioning takes O(n).
- On average, the pivot divides the array into two roughly equal parts, reducing the size of the problem to n/2 in each step

The recurrence relation for the expected case is:

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

Solving this gives:

- $T(n)=O(n)$

2.2 Space Complexity

2.2.1 Deterministic Algorithms (Median of Medians)

The deterministic Algorithm is implemented using recursion. Here the maximum recursion depth is proportional to the logarithm of the input size, as the problem size decreases geometrically by

n->n/5->7n/10

Hence, the space complexity is O (log n)

2.2.2 Randomized Quick Select algorithm

- The same goes for the depth of recursion based on each step reduction in size. Defaults the size of the problem is divided on every step to give a depth of O (log n).
- At worst, the depth of recursion could be O(n) (if the pivot always gives the smallest possible size decrease).
- Space Complexity of O (log n) is an expectation, O(n) in the worst-case scenario.

2.3 Overhead and Practical Considerations

- The deterministic algorithm the computational overhead is higher because it involves finding the median of medians, that requires sorting small groups and performing additional recursive calls.

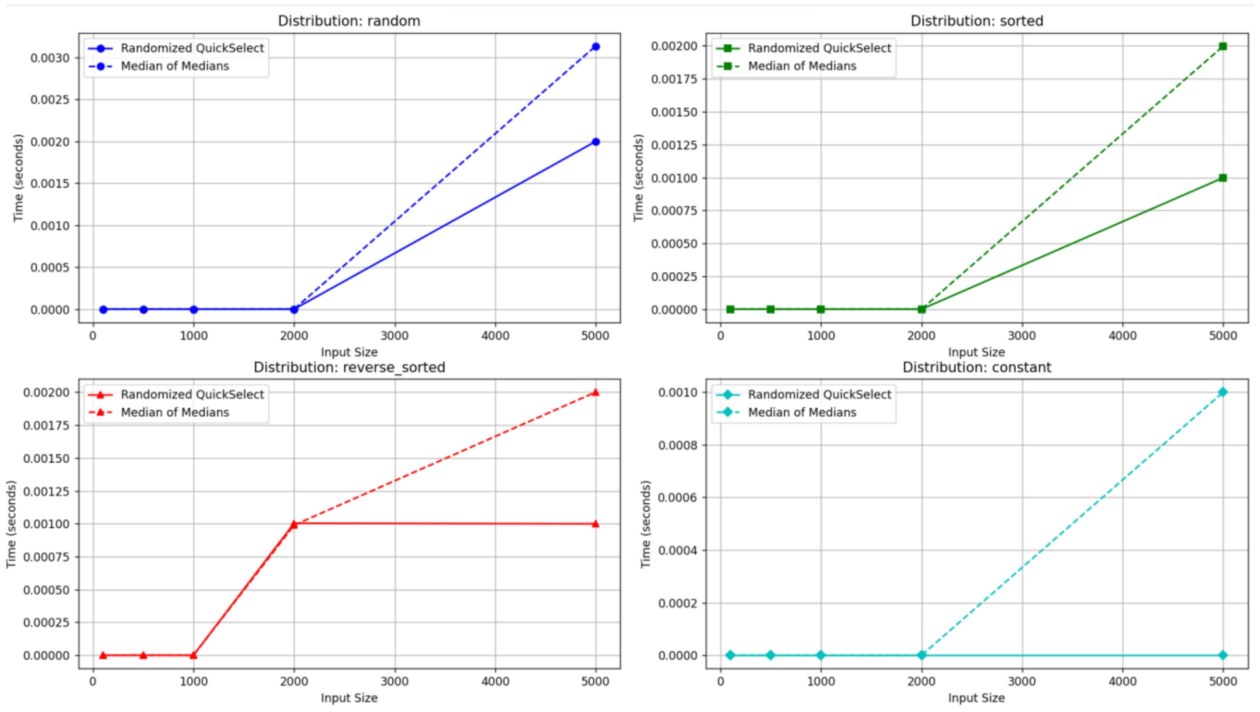
- The randomized algorithm has lower overhead but may occasionally perform poorly if the random pivot consistently leads to unbalanced partitions.

3. Empirical Analysis

3.1 Implementation of empirical analysis

```python
# Standard library imports

import time

import statistics

import copy

from typing import List, Tuple, Callable

import random


# Third-party imports

import numpy as np

from matplotlib import pyplot as plt


def generate_test_cases(size: int, distribution: str) -> Tuple[List[int], int]:

    """Generate test arrays of different distributions."""

    if distribution == "random":

        arr = [random.randint(1, size * 10) for _ in range(size)]

    elif distribution == "sorted":

        arr = list(range(1, size + 1))

    elif distribution == "reverse_sorted":

        arr = list(range(size, 0, -1))

    elif distribution == "constant":

        arr = [1] * size
    else:
        raise ValueError("Invalid distribution type")

    k = random.randint(0, size - 1)
    return arr, k

def measure_time(func: Callable, arr: List[int], k: int, num_trials: int = 5) -> float:
    """Measure average execution time over multiple trials."""
    times = []
    for _ in range(num_trials):
        arr_copy = copy.deepcopy(arr)
        start_time = time.time()
        func(arr_copy, k)
        end_time = time.time()
        times.append(end_time - start_time)

    return statistics.median(times)

def run_benchmarks(sizes: List[int], distributions: List[str]) -> dict:
    """Run comprehensive benchmarks and return results."""
    results = {
        "random_select": {dist: [] for dist in distributions},
        "median_select": {dist: [] for dist in distributions}
    }


    for size in sizes:
        print(f"Testing size {size}...")
        for dist in distributions:
            arr, k = generate_test_cases(size, dist)

            # Measure randomized quickselect
            random_time = measure_time(randomized_quickselect, arr, k)
            results["random_select"][dist].append(random_time)

            # Measure median of medians
            median_time = measure_time(median_of_medians, arr, k)
            results["median_select"][dist].append(median_time)

    return results


def plot_results(sizes: List[int], results: dict, distributions: List[str]):
    """Plot the benchmark results."""
    plt.figure(figsize=(15, 10))
    colors = ['b', 'g', 'r', 'c']
    markers = ['o', 's', '^', 'D']
    for i, dist in enumerate(distributions):
        plt.subplot(2, 2, i + 1)
        plt.plot(sizes, results["random_select"][dist], f'{colors[i]}{markers[i]}-',
                 label='Randomized QuickSelect')
        plt.plot(sizes, results["median_select"][dist], f'{colors[i]}{markers[i]}--',
                 label='Median of Medians')
        plt.title(f'Distribution: {dist}')
        plt.xlabel('Input Size')
        plt.ylabel('Time (seconds)')
        plt.legend()
        plt.grid(True)
    plt.tight_layout()
    plt.show()


def print_results(sizes: List[int], results: dict, distributions: List[str]):
    """Print numerical results and performance ratios."""
    print("\nNumerical Results (time in seconds):")


    # Print results for each algorithm
    for algo in ["Randomized QuickSelect", "Median of Medians"]:
        print(f"\n{algo}:")
        key = "random_select" if algo == "Randomized QuickSelect" else "median_select"
        for dist in distributions:
            print(f"\n{dist} distribution:")
            for size, time in zip(sizes, results[key][dist]):
                print(f"Size {size}: {time:.6f}")

    # Calculate and print performance ratios
    print("\nPerformance Ratio (Median of Medians / Randomized QuickSelect):")
    for dist in distributions:
        ratios = [m/r for m, r in zip(results["median_select"][dist],
                                      results["random_select"][dist])]
        avg_ratio = sum(ratios) / len(ratios)
        print(f"{dist}: {avg_ratio:.2f}x")

def main():
    # Set up test parameters
    sizes = [100, 500, 1000, 2000, 5000]
    distributions = ["random", "sorted", "reverse_sorted", "constant"]

    # Run benchmarks
    results = run_benchmarks(sizes, distributions)

    # Plot results
    plot_results(sizes, results, distributions)
    # Print numerical results

    print_results(sizes, results, distributions)



if __name__ == "__main__":

    main()
```

Output:



3.2 Discussion of the results observed from empirical analysis

The above attached graph shows that Randomized Quick select and Median of Medians are both linear with input size under all distributions (random, sorted, reverse-sorted, and constant). But Median of Medians always takes longer because it has overhead to find the pivot and Randomized

Quick select is faster because it is so simple. Algorithms can handle sorted and reverse-sorted inputs effectively and heavy inputs with duplications reduce the partitioning load.

For most real-world use cases Randomized Quick select is the fastest and easiest to use algorithm that promises O(n) performance. Median of Medians on the other hand, can be used for worst-case assurances and runs faster albeit longer.

Insights and Recommendations

1. Randomized Quick select is suitable for most real-world use cases where average performance is less important than worst-case guarantee. It is very easy and has low overhead, so it is faster in all distributions.
2. Median of Medians can be used in the worst-case oriented applications, for example real time system or enemy inputs.
3. Empirical evidence suggests that both algorithms are suitable for all kinds of input distributions and scale linearly with input size when predicted.

Part 2: Elementary Data Structures Implementation and Discussion

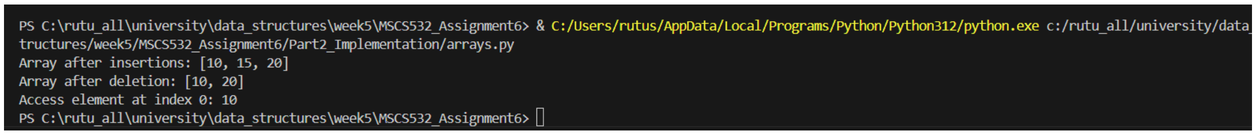
1. Implementation

1.1 Implementation of Arrays

```
class Array:

    def __init__(self, capacity):

        self.capacity = capacity

        self.size = 0

        self.data = [None] * capacity  # Fixed-size array
```

```python
def insert(self, index, value):
    if self.size >= self.capacity:
        raise Exception("Array is full")
    if index < 0 or index > self.size:
        raise IndexError("Index out of bounds")

    # Shift elements to the right
    for i in range(self.size, index, -1):
        self.data[i] = self.data[i - 1]
    self.data[index] = value
    self.size += 1

def delete(self, index):
    if index < 0 or index >= self.size:
        raise IndexError("Index out of bounds")

    # Shift elements to the left
    for i in range(index, self.size - 1):
        self.data[i] = self.data[i + 1]
    self.data[self.size - 1] = None
    self.size -= 1
    def access(self, index):
        if index < 0 or index >= self.size:
            raise IndexError("Index out of bounds")
        return self.data[index]

    def __str__(self):
        return str([self.data[i] for i in range(self.size)])

# Creating an array with capacity 5
array = Array(5)
array.insert(0, 10)  # Insert 10 at index 0
array.insert(1, 20)  # Insert 20 at index 1
array.insert(1, 15)  # Insert 15 at index 1 (shifts others)
print("Array after insertions:", array)

array.delete(1)  # Delete the element at index 1
print("Array after deletion:", array)

print("Access element at index 0:", array.access(0))
```

Output:



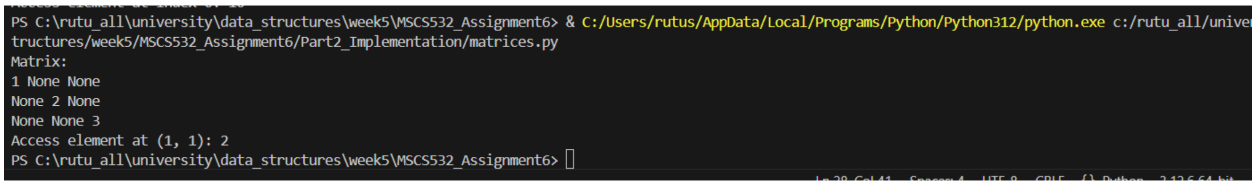
```
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_
tructures/week5/MSCS532_Assignment6/Part2_Implementation/arrays.py
Array after insertions: [10, 15, 20]
Array after deletion: [10, 20]
Access element at index 0: 10
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> 
```

1.2 Implementation of Matrices

```
class Matrix:

    def __init__(self, rows, cols):

        self.rows = rows

        self.cols = cols

        self.data = [[None] * cols for _ in range(rows)]


    def insert(self, row, col, value):

        if row < 0 or row >= self.rows or col < 0 or col >= self.cols:

            raise IndexError("Index out of bounds")

        self.data[row][col] = value


    def access(self, row, col):

        if row < 0 or row >= self.rows or col < 0 or col >= self.cols:

            raise IndexError("Index out of bounds")

        return self.data[row][col]


    def __str__(self):

        return "\n".join([" ".join(map(str, row)) for row in self.data])



# Creating a 3x3 matrix
```

```python
matrix = Matrix(3, 3)

matrix.insert(0, 0, 1)  # Insert 1 at (0, 0)

matrix.insert(1, 1, 2)  # Insert 2 at (1, 1)

matrix.insert(2, 2, 3)  # Insert 3 at (2, 2)

print("Matrix:")

print(matrix)


print("Access element at (1, 1):", matrix.access(1, 1))
```

Output:



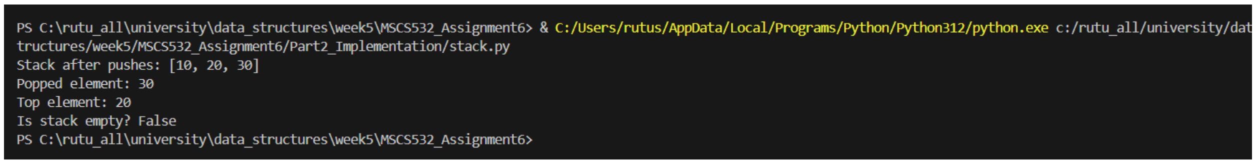
```
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/univer
tructures/week5/MSCS532_Assignment6/Part2_Implementation/matrices.py
Matrix:
1 None None
None 2 None
None None 3
Access element at (1, 1): 2
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> 
```

1.3 Stack Implementation

```python
class Stack:

    def __init__(self):

        self.data = []


    def push(self, value):

        self.data.append(value)


    def pop(self):
        if not self.data:
            raise Exception("Stack is empty")
        return self.data.pop()

    def peek(self):
        if not self.data:
            raise Exception("Stack is empty")
        return self.data[-1]

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

    def __str__(self):
        return str(self.data)

# Creating a stack
stack = Stack()
stack.push(10)  # Push 10 onto the stack
stack.push(20)  # Push 20
stack.push(30)  # Push 30
print("Stack after pushes:", stack)

print("Popped element:", stack.pop())  # Pop top element
```

```python
print("Top element:", stack.peek())  # Peek at the top element

print("Is stack empty?", stack.is_empty())
```

Output:

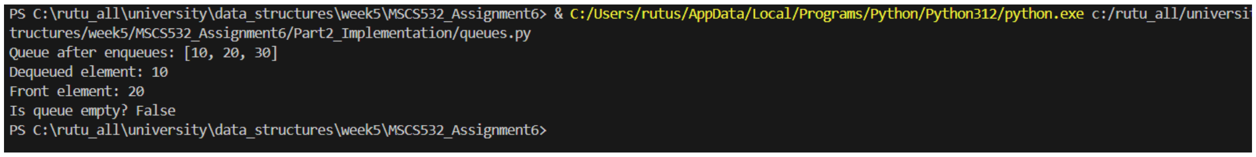


```
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/dat
tructures/week5/MSCS532_Assignment6/Part2_Implementation/stack.py
Stack after pushes: [10, 20, 30]
Popped element: 30
Top element: 20
Is stack empty? False
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6>
```

1.4 Queue Implementation

```python
class Queue:

    def __init__(self):

        self.data = []



    def enqueue(self, value):

        self.data.append(value)



    def dequeue(self):

        if not self.data:

            raise Exception("Queue is empty")

        return self.data.pop(0)



    def peek(self):

        if not self.data:
            raise Exception("Queue is empty")
        return self.data[0]

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

    def __str__(self):
        return str(self.data)

# Creating a queue
queue = Queue()
queue.enqueue(10)  # Enqueue 10
queue.enqueue(20)  # Enqueue 20
queue.enqueue(30)  # Enqueue 30
print("Queue after enqueues:", queue)

print("Dequeued element:", queue.dequeue())  # Dequeue front element
print("Front element:", queue.peek())  # Peek at the front element
print("Is queue empty?", queue.is_empty())
```

Output:



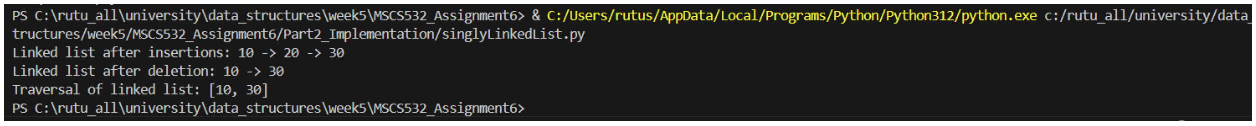
```
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/universi
tructures/week5/MSCS532_Assignment6/Part2_Implementation/queues.py
Queue after enqueues: [10, 20, 30]
Dequeued element: 10
Front element: 20
Is queue empty? False
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6>
```

1.5 Singly- linked list implementation

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None


class SinglyLinkedList:
    def __init__(self):
        self.head = None


    def insert(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node


    def delete(self, value):
        if not self.head:
            raise Exception("List is empty")
        if self.head.value == value:
            self.head = self.head.next
            return
        current = self.head
        while current.next and current.next.value != value:
            current = current.next
        if current.next:
            current.next = current.next.next
        else:
            raise ValueError("Value not found in the list")

    def traverse(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.value)
            current = current.next
        return elements

    def __str__(self):
        return " -> ".join(map(str, self.traverse()))

# Creating a singly linked list
```

```python
linked_list = SinglyLinkedList()

linked_list.insert(10)  # Insert 10

linked_list.insert(20)  # Insert 20

linked_list.insert(30)  # Insert 30

print("Linked list after insertions:", linked_list)



linked_list.delete(20)  # Delete node with value 20

print("Linked list after deletion:", linked_list)



print("Traversal of linked list:", linked_list.traverse())
```

Output:



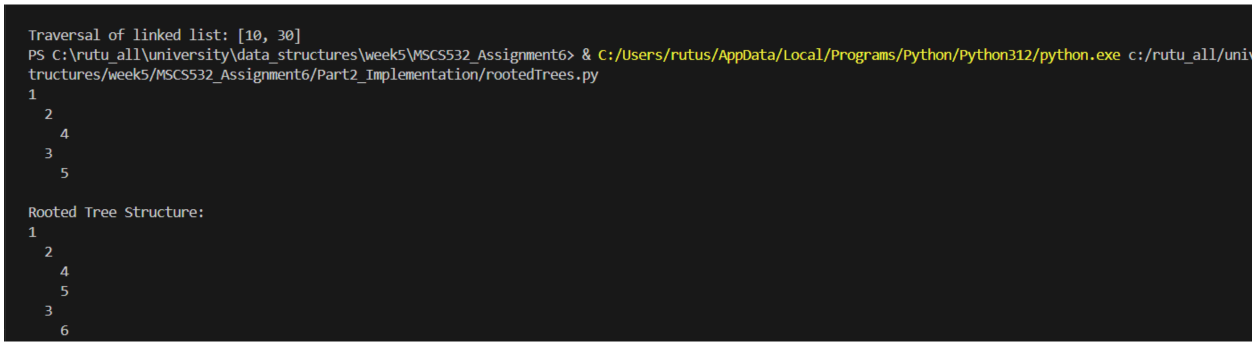
```
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_
tructures/week5/MSCS532_Assignment6/Part2_Implementation/singlyLinkedList.py
Linked list after insertions: 10 -> 20 -> 30
Linked list after deletion: 10 -> 30
Traversal of linked list: [10, 30]
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6>
```

1.6 Rooted Trees.py implementation

```python
class TreeNode:

    def __init__(self, value):

        self.value = value

        self.children = []



    def add_child(self, child_node):

        self.children.append(child_node)



    def __str__(self, level=0):
        result = " " * level + str(self.value) + "\n"

        for child in self.children:

            result += child.__str__(level + 2)

        return result


# Example usage:

root = TreeNode(1)

child1 = TreeNode(2)

child2 = TreeNode(3)

root.add_child(child1)

root.add_child(child2)

child1.add_child(TreeNode(4))

child2.add_child(TreeNode(5))

print(root)


# Creating a rooted tree

root = TreeNode(1)

child1 = TreeNode(2)

child2 = TreeNode(3)


# Adding children to the root

root.add_child(child1)

root.add_child(child2)
```

```python
# Adding grandchildren
child1.add_child(TreeNode(4))
child1.add_child(TreeNode(5))
child2.add_child(TreeNode(6))


# Printing the tree
print("Rooted Tree Structure:")
print(root)
```

Output:



```
Traversal of linked list: [10, 30]
PS C:\rutu_all\university\data_structures\week5\MSCS532_Assignment6> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/univ
tructures/week5/MSCS532_Assignment6/Part2_Implementation/rootedTrees.py
1
  2
    4
  3
    5

Rooted Tree Structure:
1
  2
    4
    5
  3
    6
```

2. Performance Analysis

2.1 Time complexity analysis for basic operations of each defined data structure in implementation part.

2.1 Arrays and Matrices

In arrays, the insertion at a specific index requires shifting elements to right, taking O(n) in worst case when inserting at beginning.

In matrices, specific cell accessing time takes O (1) time complexity, however modifying a matrix row or column is O(n), where n is the matrix size.

2.1.2 Deletion

Array deletion also has shifting elements to the left, taking the time complexity of O(n) in worst case.

2.1.3 Access

Time complexity to access the element in matrix is O (1)

Please see the detailed time complexity is:

Operation	**Array (1D)**	**Matrices (2D)**
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)
Access	O(1)	O(1)

2.1.4 Stacks

In stacks, insertion operations known as push is done at the end of stack and pop i.e removal of the element from the end in a stack with an array takes time complexity of O (1). Moreover, if the stack is implemented using linked list, operations involve creating or deleting a node, taking time of array complexity of O (1).

2.1.5 Queues

Enqueue (insertion to the end) in array-backed queue is O (1), and dequeue (removal from the front) requires shifting of all elements, hence time complexity is O(n). With linked lists, enqueue and dequeue are O (1) since pointers are being used.

Operation	**Stack (Arrays)**	**Queue (Array)**	**Stack (Linked List)**	**Queue (Linked List)**
Push/ Enqueue	O (1)	O (1)	O (1)	O (1)
Pop / Dequeue	O (1)	O (n)	O (1)	O (1)

2.1 6 Singly – linked list

- In singly- linked list, insertion at either head or tail takes O (1). but inserting at a specific position takes O (n) time.
- Deletion also takes O (1) for already known node, whereas finding a node to delete takes O (n).
- Traversal of all nodes takes O (n) time complexity

2.1.7 Root Trees

In rooted trees, addition of a child node takes O (1) and traversal of entire tree takes O (n), where n is the total number of nodes.

2.2. Trade-offs between array vs linked – list for implementing stacks and queues.

Memory Usage:

Arrays used shared memory location, and this can be a memory waster if the allocated memory size is larger than needed. Whereas in Linked list, the memory allocation is dynamic, however they still need more memory for pointers.

Performance:

Arrays allow O (1) access to elements, and they are more performant with small sets due to cache locality. But shrinking arrays (say during stack overflow) is costly because elements are copied. Linked lists have monotonic O (1) insert/delete time, but O(n) traversal to get some arbitrary item.

Use Case:

- Arrays are great for stacks where you know what the maximum size is, and when you rarely resize.
- Linked lists are better for queues as they have O (1) insertion and deletion at both ends.

2.3 Comparison between the efficiency of different data structure into specific scenario

Arrays and Matrices:

Arrays are efficient when the size is fixed and predictable, since they allow for O(1) access. Matrices are good for mathematical calculations, like graph representations, because elements are stored in a regular manner and access is predictable.

Stacks and Queues:

The array implementations of stacks are efficient in cases where there is a restriction to push and pop operations. The linked list should be considered where dynamic resizing or/and memory constraints are an issue, whereas in the case of queues, for which size can grow dynamically, linked lists have better performance than arrays, since array dequeuing has to shift all elements down, which takes O(n) time.

Linked Lists

Linked lists are ideal for systems where insertion and removal occur frequently, such as in applications requiring the maintenance of dynamic task lists or implementing "undo" functionality in software. However, their O(n) traversal makes them inefficient compared to arrays for random access.

Rooted Trees:

Rooted trees are optimal for hierarchical data structures (e.g., file systems, organization charts). They efficiently model relationships but require careful implementation for traversal and balancing in applications like binary search trees.

3. Discussion

3.1 Practical Applications of these data structures in real world scenarios.

3.1.1 Arrays and Matrices

Arrays are one of the fundamental data structures used in many applications. They find their common usage in storing collections of data whose size is known well in advance, and random access is important, for example, pixel data in images, storage of tables in databases, and lookup tables for algorithms. Matrices find their application as two-dimensional arrays in scientific computations, computer graphics-for example, image transformations and 3-D rendering-neural networks, where weight matrices are very significant parts of deep learning models.

3.1.2 Stacks and Queues

Stacks are widely used in those cases when a program needs to implement a structure following the LIFO principle, such as maintaining the call stack of functions in a program, parsing expressions in compilers, and performing a backtracking algorithm-concepts such as maze solving or depth-first search. Queues, on the other hand, are integral in FIFO applications, including job scheduling in operating systems, task queues in web servers, and breadth-first search in graph traversal. The circular queue will be beneficial in real-time systems such as printer task buffers or video steaming buffers.

3.1.3 Linked Lists

Some common applications of linked lists are in dynamic memory allocation, for example, in implementations of stacks, queues, or hashing (chaining), where records need to be inserted and deleted dynamically. They also find many applications in music players or image viewers, where there is a need to sequentially access elements, such as songs or images, but where insertion and deletion at arbitrary positions needs to be supported.

3.1.4 Rooted Trees

Rooted trees are crucial in applications where data has a natural hierarchy. Examples include file systems, where directories and files form a tree structure; XML/HTML parsing; and organizational charts. Applications of binary search trees include searching and sorting, while heaps-a form of trees-are also utilized to implement priority queues. Other applications include decision trees, which are algorithms from machine learning, and tries or prefix trees used for text search and auto-completion.

3.2 Scenarios for preferring one data structure over another.

3.2.1 Memory Usage

Arrays are highly memory-efficient for static datasets because they store only the data, with no additional overhead for pointers. However, they require contiguous memory allocation, which may be challenging for very large datasets.

Linked lists are more memory intensive as they store pointers alongside the data, which can result in significant overhead for large datasets. Trees, being a more complex variant of linked structures, have similar overhead concerns.

3.2.2 Speed

Arrays provide O (1) random access to elements due to their contiguous memory layout, making them ideal for situations where speed of retrieval is paramount, such as in indexing or searching through small datasets.

Linked lists, with their sequential access, are slower for retrieval (O(n)) but excel in dynamic insertion and deletion at O(1) if the pointer to the node is available. Similarly, stacks and queues implemented using linked lists avoid the resizing overhead faced by arrays.

Trees strike a balance in scenarios requiring fast lookups and dynamic data management. Binary search trees (if balanced) provide O(logn) search, insertion, and deletion operations.

3.2.3 Ease of Implementation

Arrays are straightforward to implement and work well for problems requiring simple storage and access.

Linked lists, while flexible, are more challenging to implement due to the need for pointer management and extra care in handling edge cases such as empty lists or tail nodes.

Trees involve even greater complexity, as they require recursive operations for traversal and additional logic for balancing in structures like AVL trees or red-black trees.

4. Conclusion

Each data structure has a different set of strengths and weaknesses, making it more applicable in certain scenarios than in others:

- Use arrays for datasets that are static and require fast access.
- Use linked lists for dynamic datasets where insertions and deletions occur frequently.
- Employ stacks and queues to manage data linearly but with access patterns.
- Use trees for hierarchical data or scenarios requiring efficient dynamic organization, such as priority-based processing or prefix-based searching.

The choice of data structure should always align with the problem's requirements, balancing memory usage, speed, and ease of implementation.

5. GitHub Repository

https://github.com/rutus-code/MSCS532_Assignment6

6. References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
- Goodrich, M. T., & Tamassia, R. (2011). *Data structures and algorithms in Python*. Wiley.
- Weiss, M. A. (2012). *Data structures and algorithm analysis in Java* (3rd ed.). Pearson.
- Knuth, D. E. (1998). *The art of computer programming, Volume 1: Fundamental algorithms* (3rd ed.). Addison-Wesley.
- Lafore, R. (2002). *Data structures and algorithms in Java*. Sams Publishing.
- Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.
- Mitzenmacher, M., & Upfal, E. (2017). *Probability and computing: Randomized algorithms and probabilistic analysis* (2nd ed.). Cambridge University Press.