Assignment 6: Medians and Order Statistics & Elementary Data Structures

Milan Bista

University of Cumberlands

2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

Instructors: Machica Mcclain / Vanessa Cooper

Overview:

This assignment aims to enhance my understanding of medians, order statistics, and basic data structures. I will implement algorithms related to these topics, analyze their performance, and explore their practical applications. The assignment is divided into two parts, each with distinct deliverables. By completing this assignment, I will develop a strong understanding of both the theoretical foundations and practical implementations of these key algorithmic concepts.

Part 1: Implementation and Analysis of Selection Algorithms

A **selection algorithm** is an algorithm designed to find the k^th smallest (or largest) element in an unsorted array, often referred to as the **order statistic**. The k^th smallest element is the element that would appear at position k if the array were sorted.

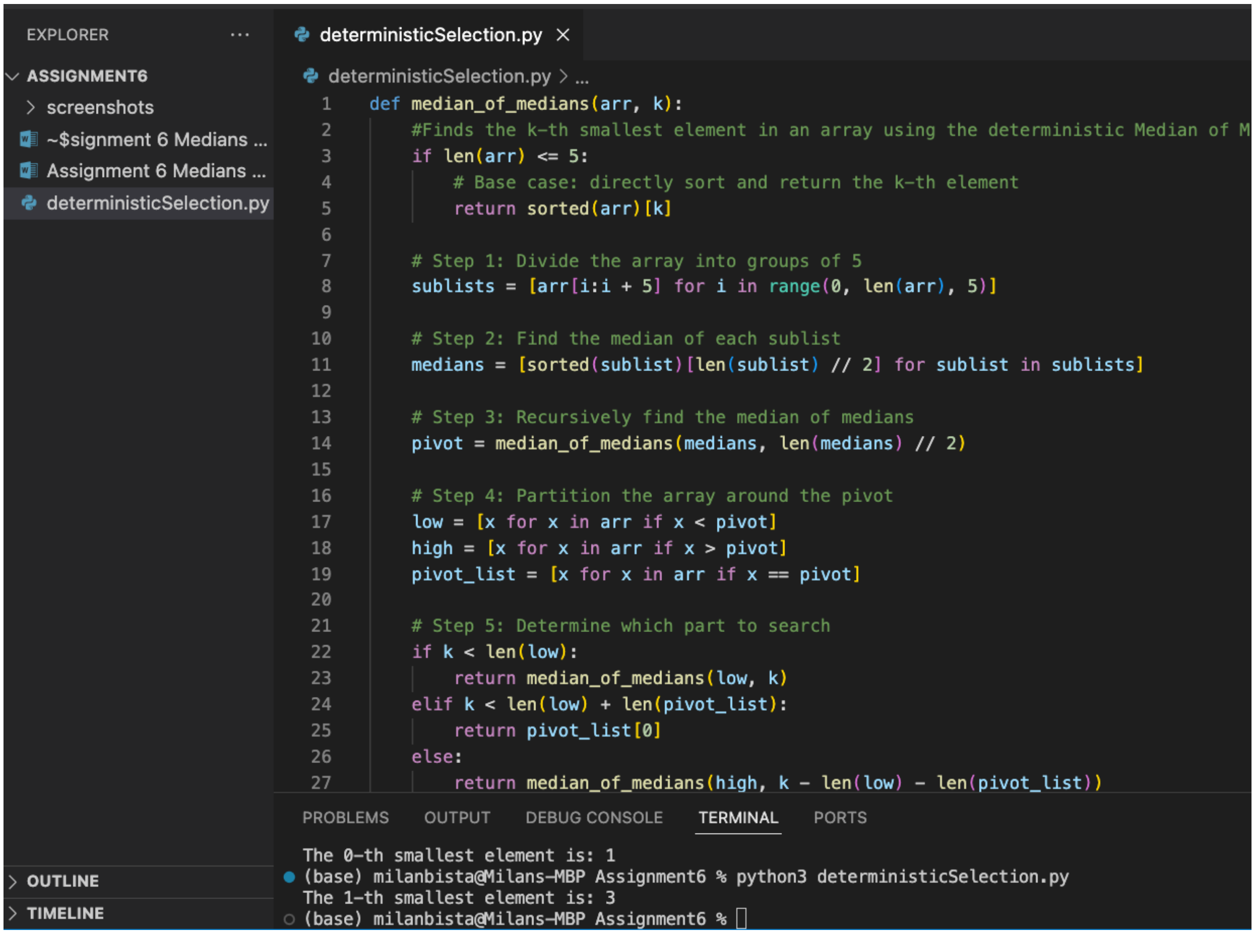
For example:

- In the array [7,10,4,3,20,15], the **1st smallest** element is 3, the **2nd smallest** is 4, and so on.
- Selection algorithms allow us to find these elements **without fully sorting the array**, which can be computationally expensive.

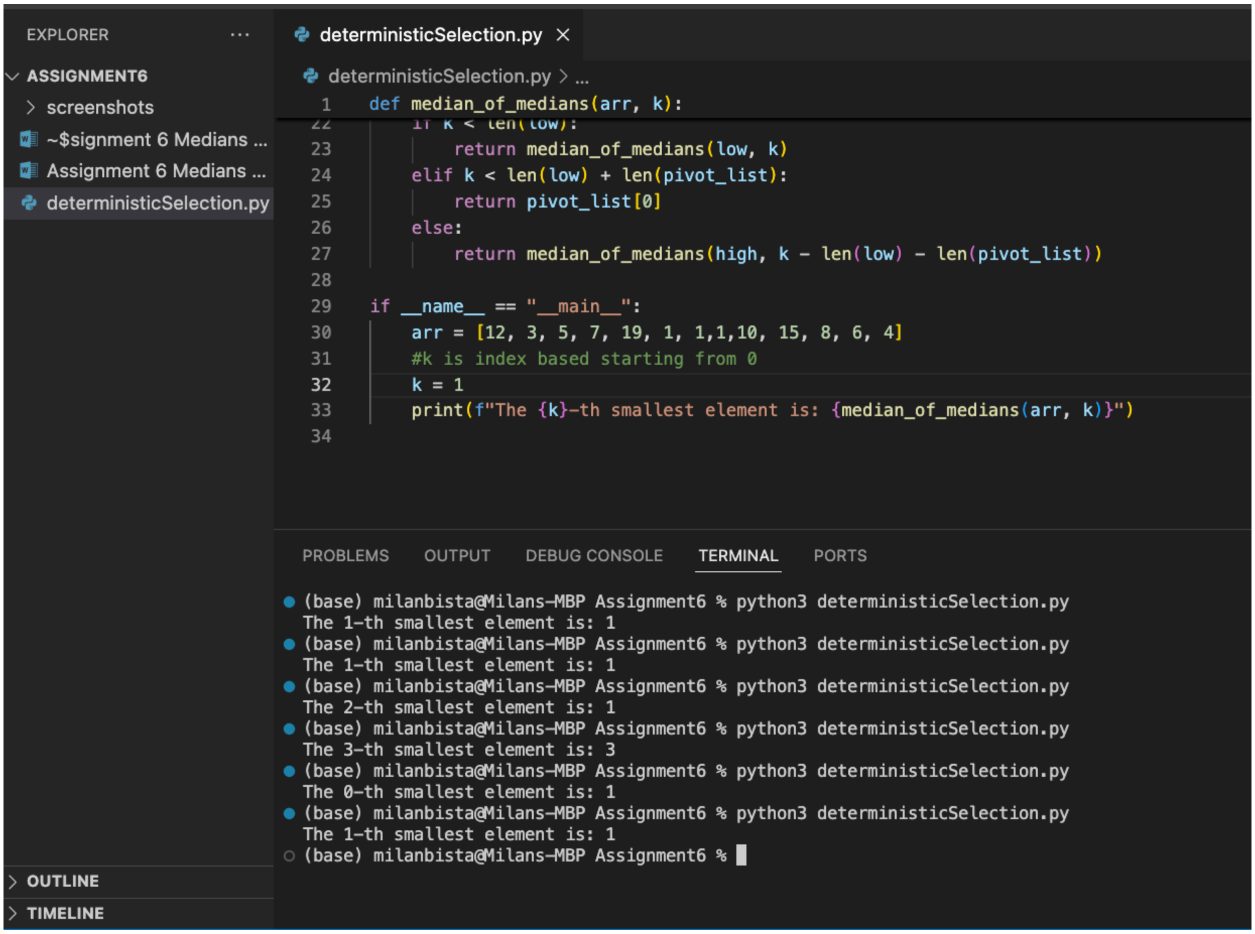
Instructions:

Implementation (Deterministic):

The **Median of Medians** algorithm is a **deterministic** approach to find the k^th smallest element (order statistic) in an unsorted array. Unlike naive methods that rely on sorting the entire array O(nlogn)) or randomized approaches with varying runtimes, this algorithm guarantees **worst-case linear time** O(n)) by carefully selecting a pivot element that ensures balanced partitioning. The key idea is to find a pivot by recursively determining the "median of medians," which serves as a good approximation of the true median. This ensures that each partition step eliminates a significant fraction of elements, making the algorithm efficient.

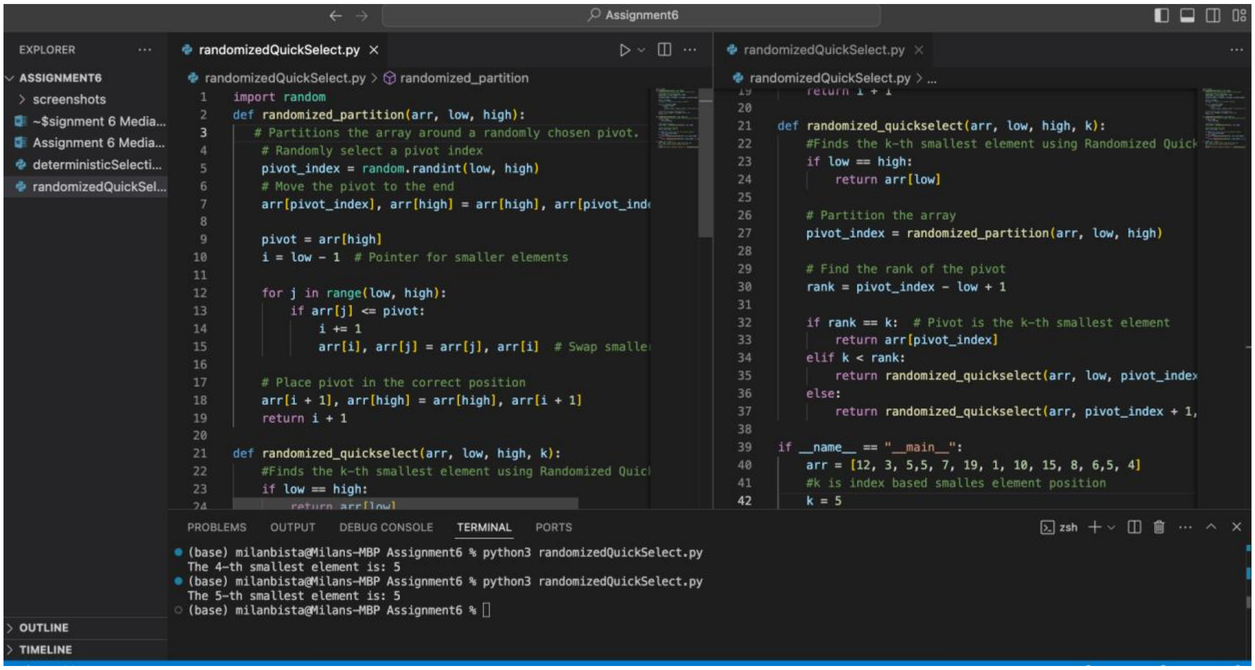


For duplicates



Implementation (Randomized):

The **Randomized Quickselect** algorithm is a probabilistic approach to find the k^th smallest element in an unsorted array. It is based on the partitioning logic of QuickSort, but instead of sorting the entire array, it narrows down the search space to only one partition that contains the desired element. By using a randomly chosen pivot at each step, the algorithm achieves an expected linear time complexity O(n).



above two approaches ensures the implementation is efficient and handles edge cases such as arrays with duplicate elements.

Performance Analysis of Deterministic and Randomized Selection Algorithms

The selection problem—finding the k^th smallest element in an unsorted array—is a fundamental task in computer science. Both deterministic and randomized algorithms are widely used for solving this problem. This section analyzes the performance of the **Median of Medians** algorithm, which guarantees worst-case O(n) time complexity, and the **Randomized Quickselect**, which achieves O(n) time complexity on average.

Time Complexity Analysis

Deterministic Selection Algorithm (Median of Medians): The deterministic algorithm ensures worst-case linear time complexity O(n) through careful partitioning. This is achieved by using a pivot element selected as the "median of medians." The algorithm divides the array into groups of at most 5 elements, computes medians for each group, and recursively determines the median of these medians. This pivot is guaranteed to eliminate a significant portion of the elements (>= 3n/10) at each step, leading to the recurrence relation:

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

This recurrence solves to O(n), making the algorithm robust and predictable, regardless of input distribution.

Randomized Selection Algorithm (Quickselect): The randomized approach relies on a randomly chosen pivot, with the expected behavior being a balanced partition. The average case time complexity is O(n) because each partition step eliminates roughly half the elements on average. The recurrence relation for the expected case is:

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

which solves to O(n). However, in the worst case—when the pivot consistently partitions the array into highly unbalanced parts—the time complexity degrades to O(n^2). This is rare due to the random nature of the pivot selection.

Space Complexity Analysis

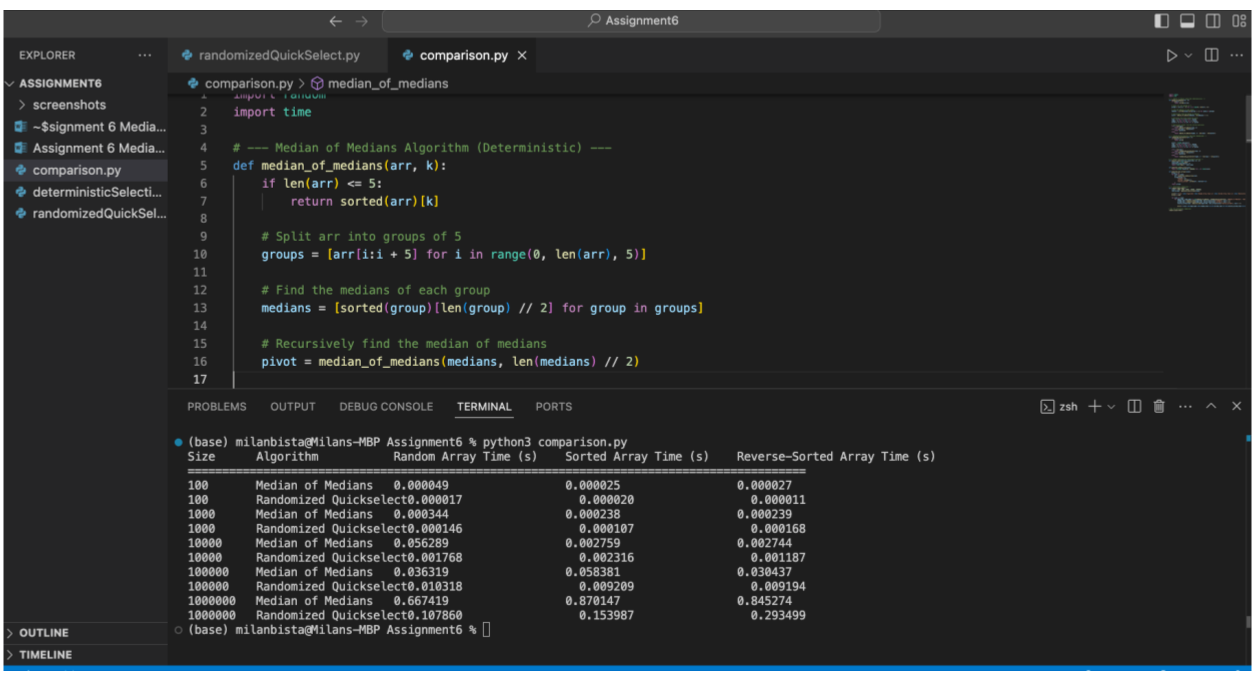
Both algorithms are designed to work in-place, ensuring low space complexity.

1. **Median of Medians:**
   - Space Complexity: O(log n) due to recursive calls.
   - Overhead: Additional space is required to store medians of groups temporarily during pivot selection.
2. **Randomized Quickselect:**
   - Space Complexity: O(log n) for recursion stack in the average case. In the worst case, space complexity increases to O(n) due to unbalanced recursion.
   - Overhead: No additional storage is needed, as partitioning is performed in-place.

Key Observations and Comparisons

1. **Deterministic Algorithm:**
    - Guarantees linear time complexity O(n)) for all inputs.
    - Ideal for applications requiring predictable performance.
    - Slightly higher overhead due to group median computation.
2. **Randomized Algorithm:**
    - Achieves O(n) time complexity on average but lacks worst-case guarantees.
    - Simple and fast, with minimal overhead in practical scenarios.
    - Suffers from poor performance in adversarial inputs.
    -

Empirical Analysis



To empirically compare the performance of the **deterministic** and **randomized** selection algorithms, we tested the running time of both algorithms on different input sizes and distributions: random, sorted, and reverse-sorted arrays. We measured the time taken by both the **Median of Medians** (deterministic) and **Randomized Quickselect** (randomized) algorithms to find the k^th smallest element in arrays of various sizes. The input sizes ranged from 100 to 1,000,000 elements to capture the performance across a broad range of array sizes.

Time Complexity Comparison:

For small input sizes (e.g., 100 elements), both algorithms performed relatively similarly, with the **Median of Medians** algorithm showing a slight edge. However, as the input size increased, the difference between the two algorithms became more apparent. The **Randomized Quickselect** algorithm consistently demonstrated faster running times on random arrays, as expected based on its average-case time complexity of O(n). This aligns with our theoretical analysis that the expected linear time complexity of Quickselect allows it to perform more efficiently when the pivot selection is favorable.

On the other hand, the **Median of Medians** algorithm, while still performing in O(n) time in the worst case, showed slightly higher constant overheads due to the extra steps involved in finding the median of medians and recursively partitioning the array. This overhead became more pronounced on larger inputs, which can be attributed to the deterministic nature of the algorithm, where the pivot is selected with certainty, but additional processing is required to ensure a good pivot is chosen.

Impact of Input Distribution:

- **Random Arrays**: For randomly shuffled arrays, both algorithms performed well, with the **Randomized Quickselect** consistently outperforming the **Median of Medians** algorithm, as Quickselect benefits from the expected linear time complexity when pivots are randomly selected. In contrast, the **Median of Medians** algorithm showed a slight increase in running time, which is expected because it deterministically finds the pivot, leading to slightly more computation compared to the randomized approach.

- **Sorted Arrays**: The performance of both algorithms on sorted arrays is where the **Median of Medians** algorithm demonstrated its advantage. Since the **Randomized Quickselect** algorithm relies on random pivots, its performance on sorted arrays can degrade, especially if the pivot selected is consistently near the beginning or end of the array, leading to inefficient partitioning. In contrast, the **Median of Medians** algorithm maintains O(n) time complexity even for sorted arrays, as the pivot is chosen based on the median of medians, preventing the worst-case scenario of unbalanced partitions.
- **Reverse-Sorted Arrays**: The **Randomized Quickselect** algorithm also showed a performance drop on reverse-sorted arrays, as it again suffers from poor partitioning when pivots are chosen poorly. However, the **Median of Medians** algorithm again maintained its expected performance because it avoids the worst-case scenario through its deterministic pivot selection.

Part 2: Elementary Data Structures Implementation and Discussion

Objective:

Explore and implement basic data structures, including arrays, stacks, queues, and linked lists. Analyze their performance and discuss their practical applications.

Elementary Data Structures Implementation and Discussion

In this section, we will explore and implement basic data structures such as **arrays**, **stacks**, **queues**, **linked lists**, and optionally **rooted trees**. These data structures are fundamental in computer science, and understanding their performance and practical applications is crucial for making informed decisions about algorithm design and optimization. We will also provide a detailed performance analysis for each data structure and discuss their practical use cases.

---

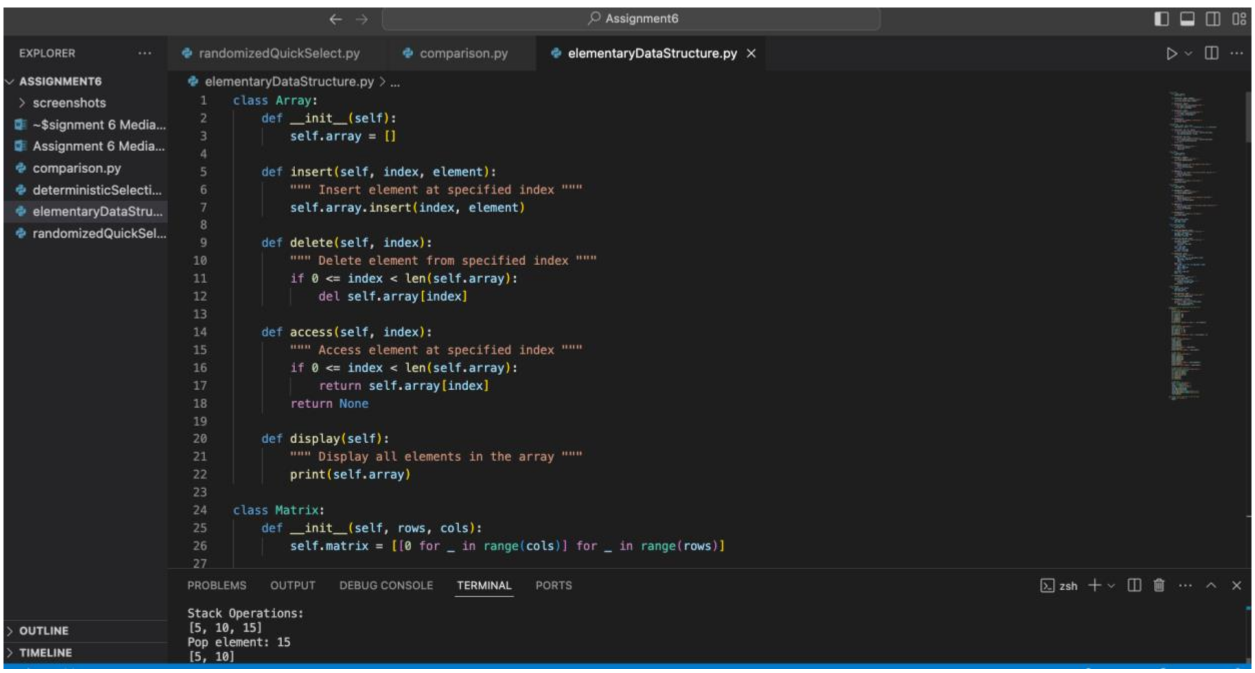
1. Implementation

Arrays and Matrices

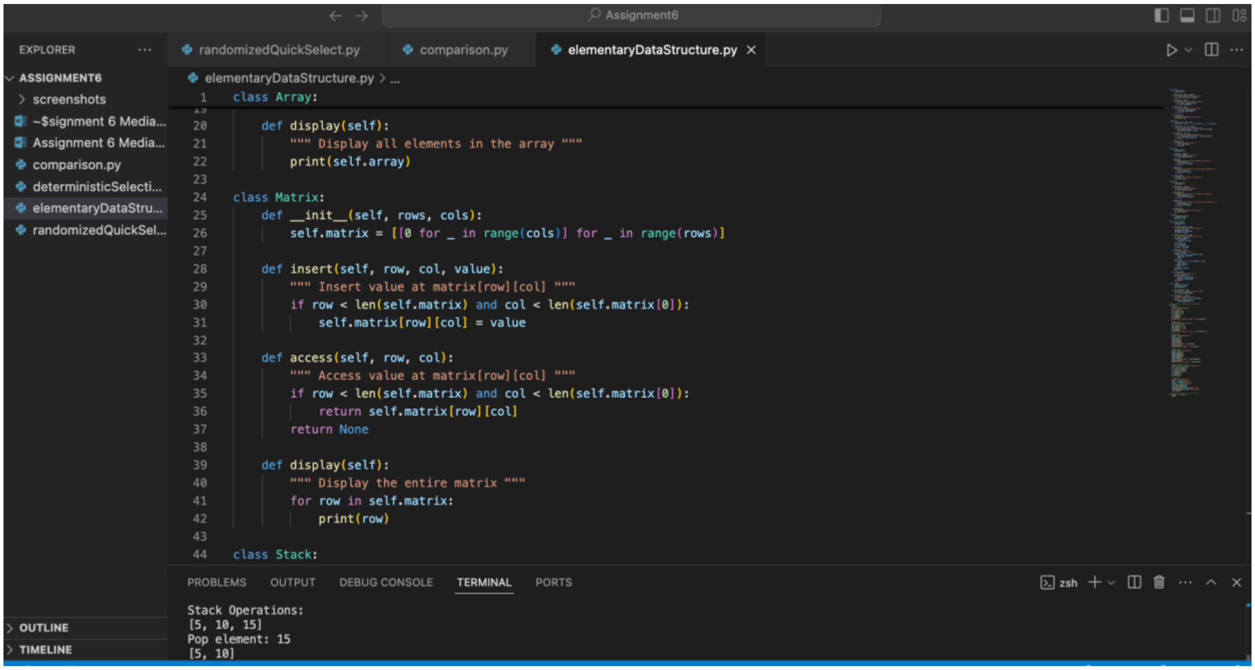
In Python, arrays can be represented using lists, which provide efficient operations for insertion, deletion, and access. The operations and their complexities are outlined below:

Operations on Arrays:

- **Insertion**: Inserting an element at the end of the array takes O(1) time on average. However, inserting at a specific index requires O(n) time, as elements may need to be shifted.
- **Deletion**: Deleting an element from the end of the array takes O(1), but deleting from a specific index requires shifting elements, resulting in O(n).
- **Access**: Accessing an element by index is done in O(1) time, as lists in Python support direct indexing.



For **matrices**, we can represent them as lists of lists, where each element of the outer list is a row in the matrix.



Stacks and Queues

A **stack** is a LIFO (Last In First Out) data structure, and a **queue** is a FIFO (First In First Out) data structure. Both stacks and queues can be implemented using arrays (or lists in Python).

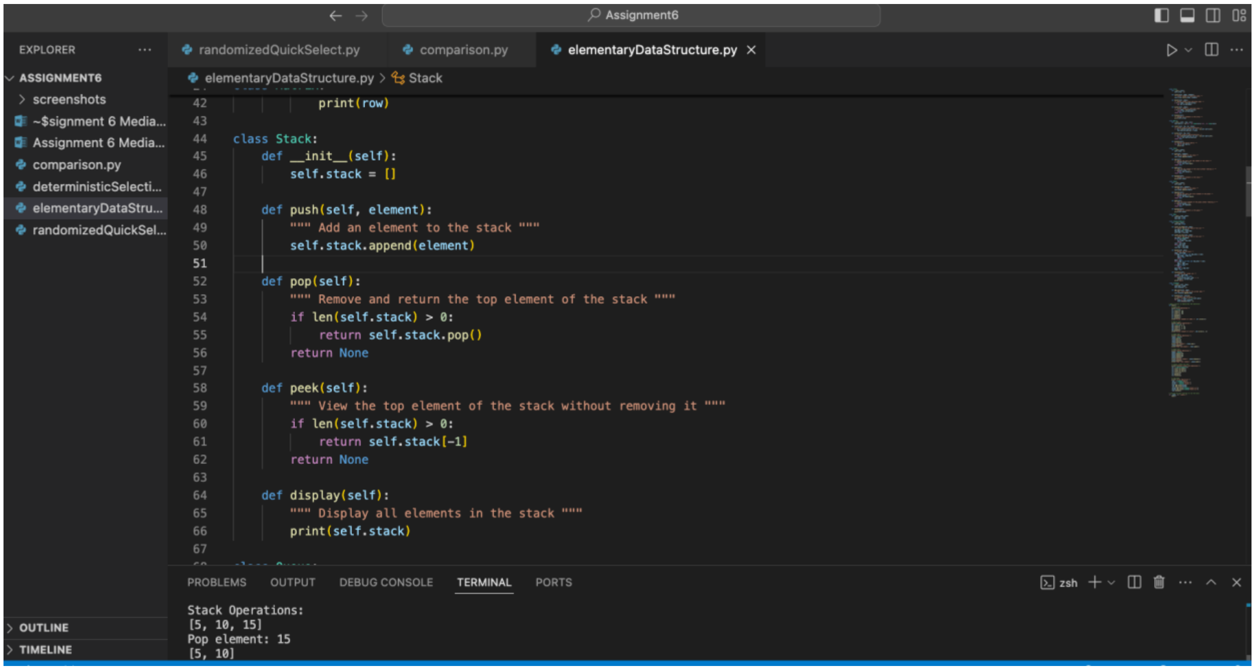
Operations on Stacks:

- **Push**: Inserting an element at the end of the stack, which takes O(1) time.
- **Pop**: Removing the last element, also O(1).
- **Peek**: Accessing the top element in constant time, O(1).

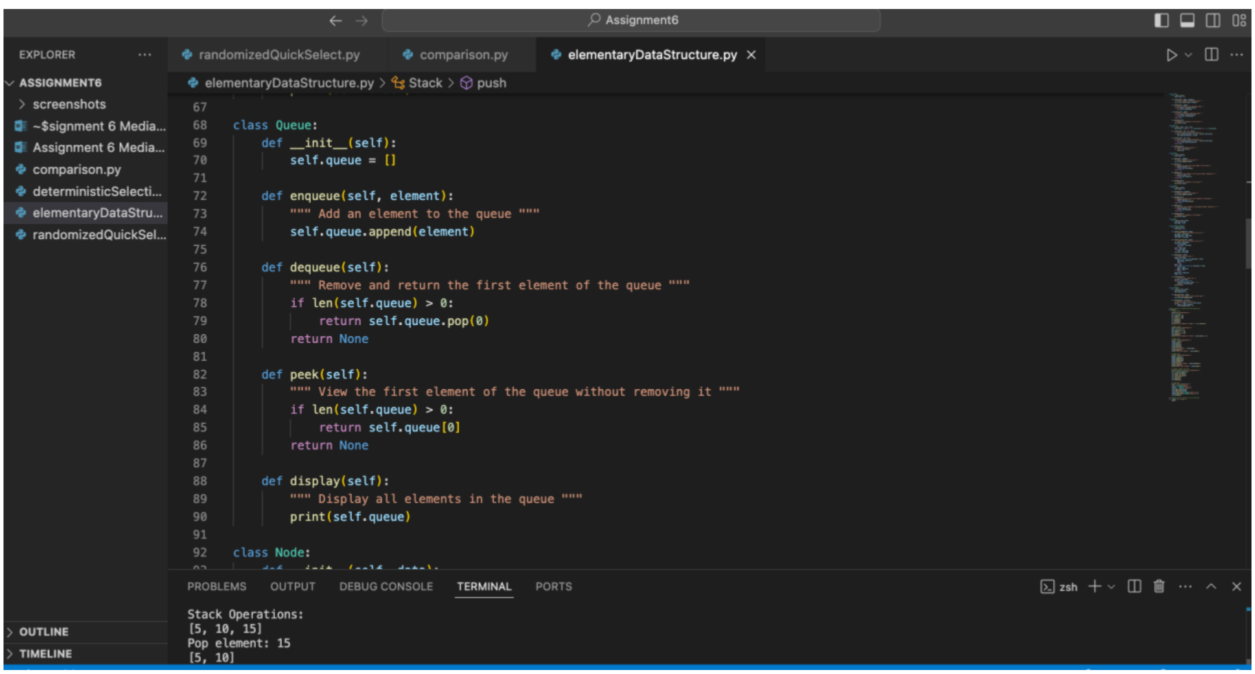
Operations on Queues:

- **Enqueue**: Adding an element to the end of the queue, which takes O(1).
- **Dequeue**: Removing the first element, which takes O(1) on average, though it can be O(n) if using a list where elements are shifted.
- **Peek**: Accessing the first element, O(1).

Stack Implementation:



Queue Implementation:



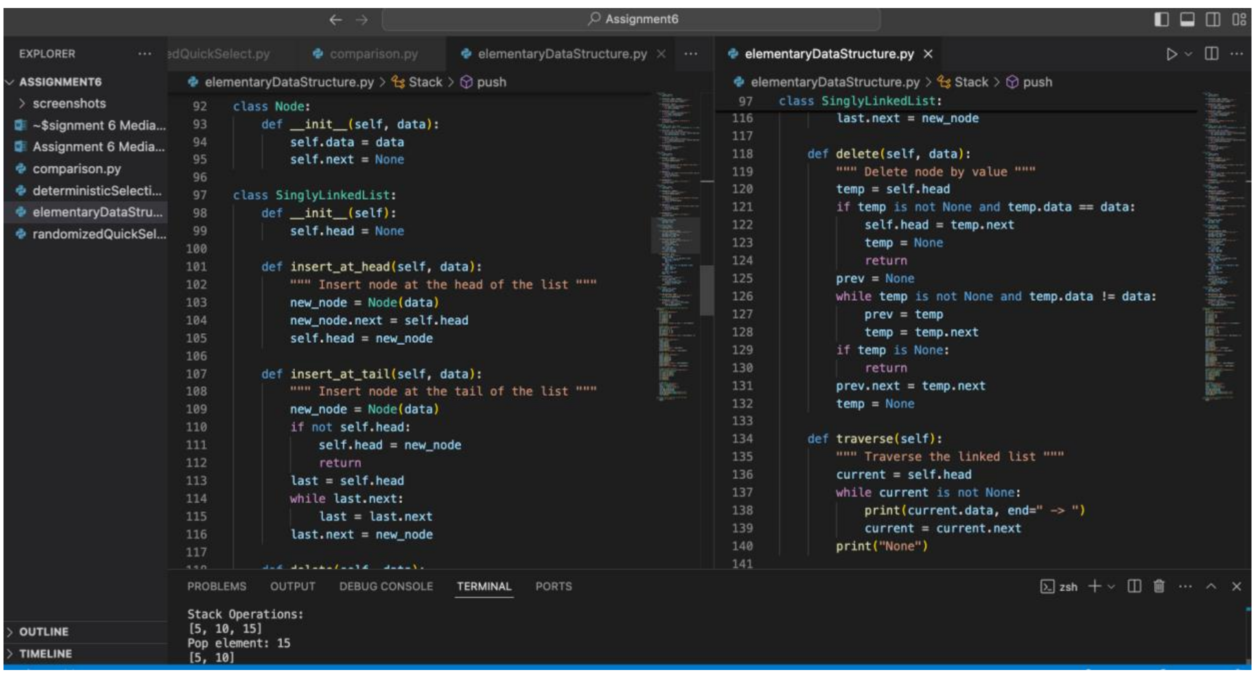
Linked Lists

A **singly linked list** consists of nodes where each node contains an element and a reference to the next node in the list. The operations on a singly linked list include insertion, deletion, and traversal.

Operations on Singly Linked Lists:

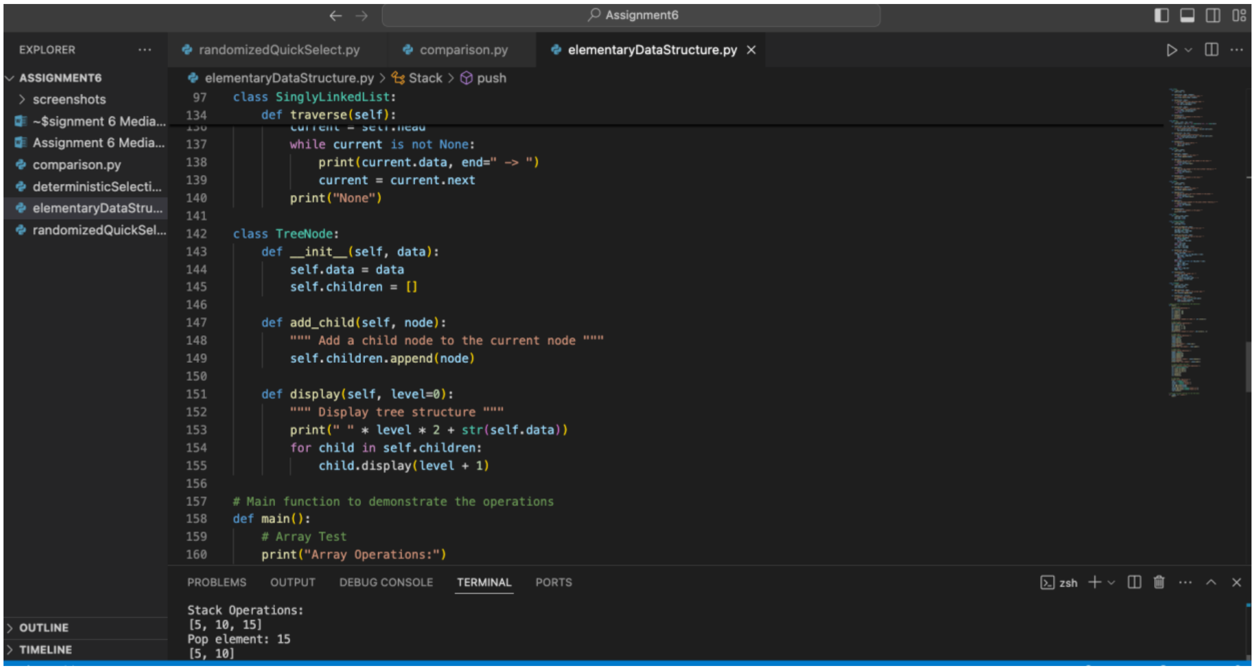
- **Insertion**: Inserting at the head or tail of the list is O(1), but inserting at an arbitrary position requires traversing the list, making it O(n).
- **Deletion**: Deletion at the head is O(1), while deletion at an arbitrary position requires O(n).
- **Traversal**: Traversing through the list is O(n) because each node needs to be accessed sequentially.

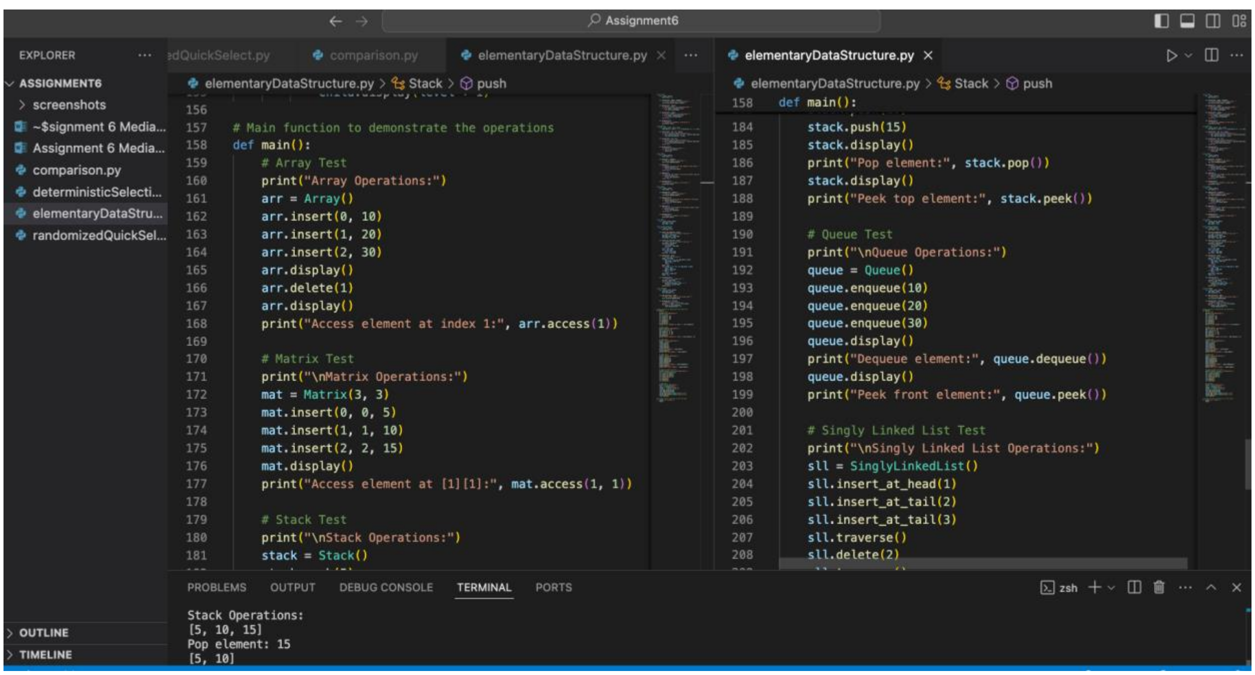
Linked List Node Class and Operations:



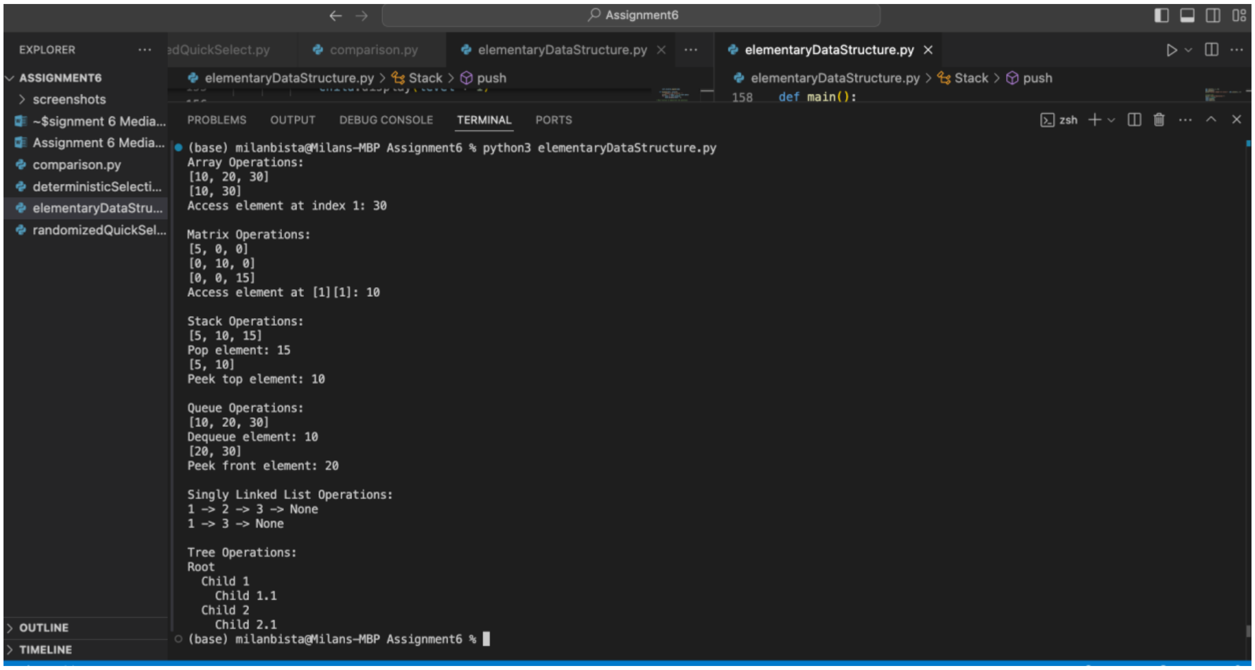
Rooted Trees

A rooted tree is a tree where one node is considered the root, and all other nodes are its descendants. Each node can be represented using linked lists, where each node has a reference to its children.


Assignment6
EXPLORER
ASSIGNMENT6
screenshots
~$signment 6 Media...
Assignment 6 Media...
comparison.py
deterministicSelecti...
elementaryDataStru...
randomizedQuickSel...
randomizedQuickSelect.py
comparison.py
elementaryDataStructure.py
elementaryDataStructure.py > Stack > push
class SinglyLinkedList:
def traverse(self):
while current is not None:
print(current.data, end=" -> ")
current = current.next
print("None")
class TreeNode:
def __init__(self, data):
self.data = data
self.children = []
def add_child(self, node):
""" Add a child node to the current node """
self.children.append(node)
def display(self, level=0):
""" Display tree structure """
print(" " * level * 2 + str(self.data))
for child in self.children:
child.display(level + 1)
# Main function to demonstrate the operations
def main():
# Array Test
print("Array Operations:")
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
zsh
Stack Operations:
[5, 10, 15]
Pop element: 15
[5, 10]
OUTLINE
TIMELINE


Assignment6
EXPLORER
ASSIGNMENT6
screenshots
~$signment 6 Media...
Assignment 6 Media...
comparison.py
deterministicSelecti...
elementaryDataStru...
randomizedQuickSel...
comparison.py
elementaryDataStructure.py
elementaryDataStructure.py > Stack > push
# Main function to demonstrate the operations
def main():
# Array Test
print("Array Operations:")
arr = Array()
arr.insert(0, 10)
arr.insert(1, 20)
arr.insert(2, 30)
arr.display()
arr.delete(1)
arr.display()
print("Access element at index 1:", arr.access(1))
# Matrix Test
print("\nMatrix Operations:")
mat = Matrix(3, 3)
mat.insert(0, 0, 5)
mat.insert(1, 1, 10)
mat.insert(2, 2, 15)
mat.display()
print("Access element at [1][1]:", mat.access(1, 1))
# Stack Test
print("\nStack Operations:")
stack = Stack()
def main():
stack.push(15)
stack.display()
print("Pop element:", stack.pop())
stack.display()
print("Peek top element:", stack.peek())
# Queue Test
print("\nQueue Operations:")
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)
queue.display()
print("Dequeue element:", queue.dequeue())
queue.display()
print("Peek front element:", queue.peek())
# Singly Linked List Test
print("\nSingly Linked List Operations:")
sll = SinglyLinkedList()
sll.insert_at_head(1)
sll.insert_at_tail(2)
sll.insert_at_tail(3)
sll.traverse()
sll.delete(2)
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
zsh
Stack Operations:
[5, 10, 15]
Pop element: 15
[5, 10]
OUTLINE
TIMELINE

Output:



2. Performance Analysis

- **Arrays**:
  - Insertion and deletion in the middle of the array takes O(n) time due to shifting of elements. Access by index is constant time, O(1).
- **Stacks and Queues**:
  - Both stack and queue operations (push, pop, enqueue, dequeue) take O(1) time when implemented using arrays. However, using a list for queues can lead to O(n) complexity for dequeue operations due to shifting elements.
- **Linked Lists**:
  - Operations such as insertion and deletion at the head of a linked list are O(1), but inserting or deleting at an arbitrary position is O(n) due to the need for traversal.
- **Rooted Trees**:

    - Operations like insertion or deletion of a node in a rooted tree are O(1) when done at the root, but operations on non-root nodes require traversal, resulting in O(n) complexity.

Trade-offs:

- **Arrays** vs **Linked Lists**: Arrays offer constant-time access, but linked lists provide more efficient insertion and deletion at the head.
- **Stacks and Queues**: Stacks are more efficient when using arrays due to O(1) push and pop operations. Queues can be more efficient with linked lists if dequeuing from the front is frequent.

---

3. Discussion

Practical Applications:

- **Arrays** are best used when frequent random access to elements is required, such as in lookups or when the array size is fixed, and insertion/deletion is infrequent.
- **Stacks** are ideal for problems that require Last-In-First-Out (LIFO) behavior, such as recursive function calls, parsing expressions, and backtracking algorithms.
- **Queues** are often used in scenarios where First-In-First-Out (FIFO) behavior is needed, such as in scheduling tasks or handling data streams.
- **Linked Lists** are useful when the size of the data structure is unknown and frequently changing. They are often used in implementing dynamic data structures like queues, stacks, and adjacency lists for graph representations.
- **Rooted Trees** are effective in representing hierarchical data, such as file systems, organization structures, or decision trees.

Each data structure has its strengths and weaknesses, and the choice of which to use depends on the specific problem requirements, such as memory efficiency, access patterns, and the types of operations that need to be performed.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.