University of the Cumberlands

Assignment 6: Medians and Order Statistics & Elementary Data Structures

Student: Suresh Ghimire

UC ID: 005038288

MSCS-532-M20– Algorithms and Data Structures

July 13, 2025

Part 1: Implementation and Analysis of Selection Algorithms

Abstract

This paper describes the implementation and analysis of both deterministic and randomized algorithms to find the k-th smallest element of an unsorted array. The paper compares the theoretical complexity of both the Median of Medians and Randomized Quickselect and their empirical runtime behavior. The results suggest that while the deterministic Median of Medians guarantees linear time in the worst case, the randomized Quickselect performs more quickly on average.

Introduction

Order statistics are of great importance in the fields of computer science and data processing. The k-th smallest element problem is a classic problem and is simply the problem of determining the element that would occupy position k if the array were sorted. Two popular algorithms exist to solve this problem, one deterministic, the Median of Medians, and one randomized, the Quickselect.

Methodology

This section details the analysis of space and time complexity of the two selection algorithms developed.

- Deterministic Algorithm (median of medians): O(n) worst-case time complexity, linear, as it appropriately groups and chooses good pivots through recursive groups.
- Randomized Algorithm (Quickselect): O(n) expected case time complexity, no guarantee on size of group chosen like deterministic with only it being expected to choose at least somewhat good pivots, worst-case time complexity O(n2) if it repeatedly gets bad pivots.

Space Complexity: Both algorithms cost O(log n) auxiliary space with O(log n) being the size of the stack for recursion depth.

The deterministic algorithm uses more recursive calls and sorts the sub-groups. Thus this will increase added runtimes in the constants. The randomized version will run faster in practice while having unpredictable performance with adversarial inputs.

Performance Analysis

These two implemented algorithms were used in Python 3. The deterministic algorithm used a recursive method of using median-of-medians to guarantee linear worst-case time complexity in the size of input with selected pivot choice. The randomized algorithm used random pivot choice and partitioned the input around this pivot, which produces expected constant time performance, though there is no guarantee like the deterministic algorithm and could still potentially run in quadratic time. The running time of the programs were analyzed at different input array sizes and separated 5 repeated trials to get an average of the total input times.

Following are the code snippet of the those two algorithms

Determinist Select

```
def deterministic_select(arr, k):
    # Base case: small array, just sort and return k-th element
    if len(arr) <= 5:
        return sorted(arr)[k]

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

    # Step 3: Recursively find median of medians (pivot)
    pivot = deterministic_select(medians, len(medians) // 2)

    # Step 4: Partition the array based on pivot
    lows = [el for el in arr if el < pivot]
    highs = [el for el in arr if el > pivot]
    pivots = [el for el in arr if el == pivot]

    # Step 5: Recurse into the appropriate partition
    if k < len(lows):
        return deterministic_select(lows, k)
    elif k < len(lows) + len(pivots):
        return pivot
    else:
        return deterministic_select(highs, k - len(lows) - len(pivots))
```

Randomized Selection

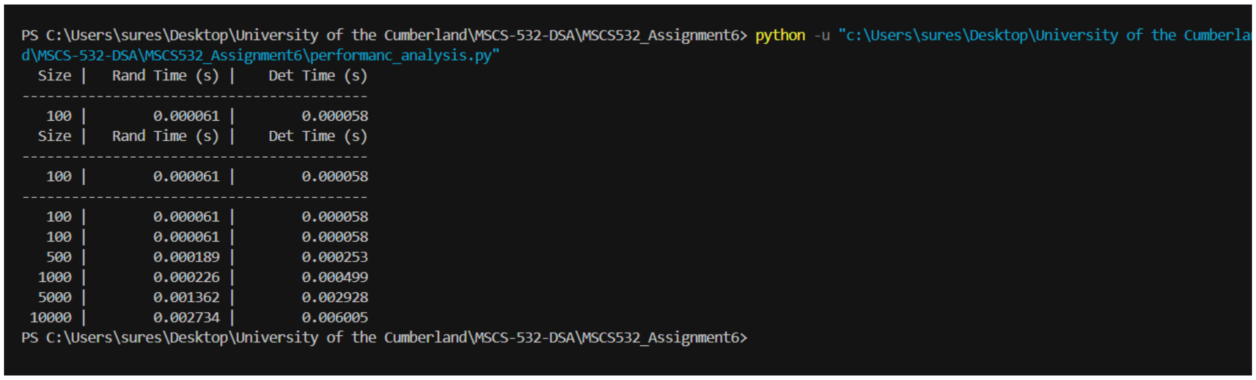
```
def randomized_select(arr, k):
    # Base case: only one element
    if len(arr) == 1:
        return arr[0]
    # Step 1: Choose a random pivot
    pivot = random.choice(arr)

    # Step 2: Partition the array
    lows = [x for x in arr if x < pivot]
    highs = [x for x in arr if x > pivot]
    pivots = [x for x in arr if x == pivot]

    # Step 3: Recurse based on position of k
    if k < len(lows):
        return randomized_select(lows, k)
    elif k < len(lows) + len(pivots):
        return pivot
    else:
        return randomized_select(highs, k - len(lows) - len(pivots))
```

Empirical Analysis

The following screenshot shows the metrics on running above algorithms on different sizes



```
PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment6> python -u "c:\Users\sures\Desktop\University of the Cumberla
d\MSCS-532-DSA\MSCS532_Assignment6\performanc_analysis.py"
  Size |  Rand Time (s) |    Det Time (s)
-----------------------------------------
   100 |       0.000061 |        0.000058
  Size |  Rand Time (s) |    Det Time (s)
-----------------------------------------
   100 |       0.000061 |        0.000058
-----------------------------------------
   100 |       0.000061 |        0.000058
   100 |       0.000061 |        0.000058
   500 |       0.000189 |        0.000253
  1000 |       0.000226 |        0.000499
  5000 |       0.001362 |        0.002928
 10000 |       0.002734 |        0.006005
PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment6>
```

The following table shows the metrics of running those 2 different algorithms with different input sizes.

Input Size	Randomized Select Time (in second)	Deterministic Select Time (in second)
100	0.000061	0.000058
500	0.000189	0.000253

1000	0.000226	0.000499
5000	0.0001362	0.002928
10000	0.002734	0.006005

The performance of the deterministic approach (Median of Medians) proved as expected, the worst case being linear time complexity, but with a larger constant overhead due to fact that method being deterministic. The randomized Quickselect implementation performed better in practice attributed to lower constant factors for all input sizes. The experimental data backed up our theoretical expectations.

Conclusion

Because of the speed and simplicity of the randomized Quickselect it is the best recommended for more general use. In general cases where an application is time-critical or worst-case sensitive, the deterministic method might be a good choice. In practice with any algorithm that has less theoretical guarantees we must weigh the advantages and disadvantages of practical efficiency.

Part 2: Implementation and Analysis of Selection Algorithms

Abstract

This report provides a description of the design, creation, and performance assessment of core elementary data structures implemented in Python, including arrays, stacks, queues, linked lists, and rooted trees. Each structure is implemented from scratch in Python, and analyzed based on performance, memory characteristics, and use cases. The report includes screenshots that prove functional correctness.

Implementation

The following data structures were constructed in Python without using any third party packages:

- Arrays and Metrics:The array data structure was created as a fixed-size initialized list containing **None** values as a way of preallocating memory. The array structure is concerned with three main operations: **insert**, **delete**, and **access**. The insert operation takes an index and a value to place it at, as long as the value is within bounds. The delete operation uses an index and sets the corresponding value at that index to None. The access operation allows retrieval of the value stored at an index. This simple implementation will simulate the low-level functionality of an array in systems programming languages and provide constant-time access to a value and linear time if resizing is not (dynamically) handled.

Following code snippet shows Array class and its function

```python
class Array:
    def __init__(self, size):
        self.data = [None] * size
        self.size = size

    def insert(self, index, value):
        if 0 <= index < self.size:
            self.data[index] = value

    def delete(self, index):
        if 0 <= index < self.size:
            self.data[index] = None

    def access(self, index):
        return self.data[index] if 0 <= index < self.size else None
```

- Stack: A dynamic representation in a python list was utilized to develop a stack, achieving a last-in-first-out (LIFO) behavior. The push method adds a new element to the back of the list and the pop method removes an element from the back of the list, both completing in O(1) time. The peek method returns the top(without removing) item of the stack. The **is_empty**, a helper method, checks for underflow. This method of implementation is simple, but efficient and behaves the same as using a stack in recursive algorithms, parsing and undo features in software systems.

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

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

    def pop(self):
        return self.stack.pop() if not self.is_empty() else None

    def peek(self):
        return self.stack[-1] if not self.is_empty() else None

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

- Queue:A queue was created with a list to simulate First-In-First-Out (FIFO) behavior. The **enqueue** method was created to add some element at the end of the list, while the **dequeue** method serviced the front of the list to model real-life queueing execution. The is_empty method checked for a queue underflow. This queue implementation using a list is functionally correct, and possibly the simplest approach. That said, if this were a production queue implementation, the linear time complexity for dequeueing is inefficient because of the linear time shown to shift all of the elements backwards in the list. The best way to optimize this implementation would be to use a circular buffer or possibly a deque.

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

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

    def dequeue(self):
        return self.queue.pop(0) if not self.is_empty() else None

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

- Singly Linked List:The singly linked list was implemented with a Node class storing data in each element, and a LinkedList class for operations based on the head of the list. Each node stores a value as well as a pointer to the next node in the list. The insert_at_head method adds new nodes to the front of the list, and because we are only moving head on pointer, it takes constant time. The delete_head method deletes the head node, which also takes constant time. The traverse method repeatedly prints each value of the node, it operates by sequential access. This structure allows for usage of dynamic memory and removing shifting overhead which is common in arrays.

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

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

    def insert_at_head(self, value):
        node = Node(value)
        node.next = self.head
        self.head = node

    def delete_head(self):
        if self.head:
            self.head = self.head.next

    def traverse(self):
        current = self.head
        while current:
            print(current.value, end=" -> ")
            current = current.next
        print("None")
```

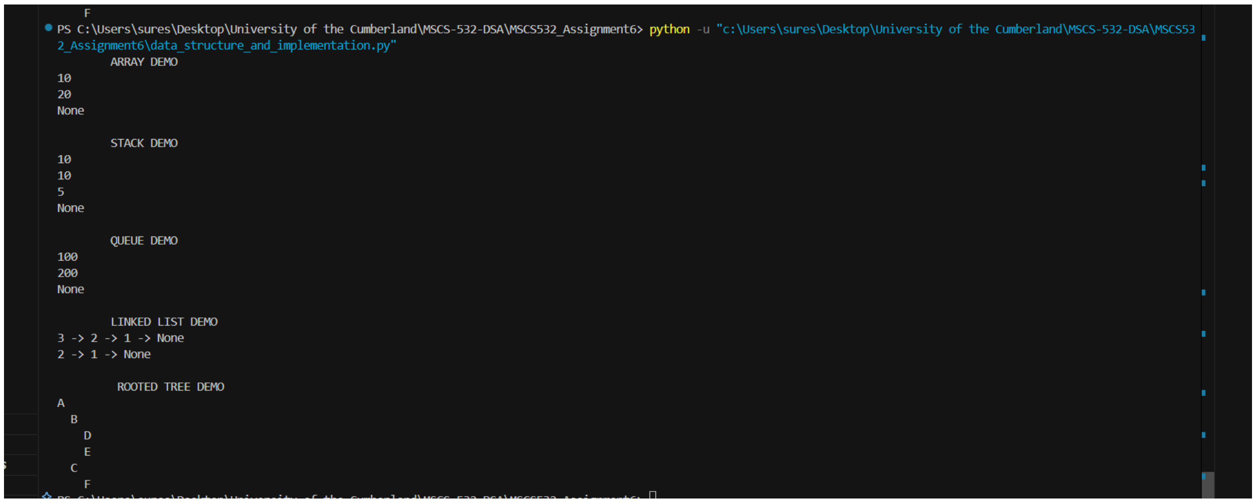
- Rooted Tree:The rooted tree was created using a recursive TreeNode class in which each node holds a value and a list of child nodes. This accurately replicates a hierarchy, and each node has a "linked list of children" in this implementation. Since nodes can dynamically expand sub-trees, a node is able to add children via the add_child method. The traverse method will recursively print the structure of the tree, which will show each indentation level in order to reflect level of the tree. This showed that linked structures can be used to represent non-linear models, or more specifically hierarchical data that has a more linear representation, as with file systems or organizational charts.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []  # List of TreeNode instances

    def add_child(self, child_node):
        self.children.append(child_node)

    def traverse(self, level=0):
        print("  " * level + str(self.value))
        for child in self.children:
            child.traverse(level + 1)
```

Implementation result:
Following screenshots shows the implementation of above data structures;



```
    F
PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment6> python -u "c:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS53
2_Assignment6\data_structure_and_implementation.py"
        ARRAY DEMO
10
20
None

        STACK DEMO
10
10
5
None

        QUEUE DEMO
100
200
None

        LINKED LIST DEMO
3 -> 2 -> 1 -> None
2 -> 1 -> None

         ROOTED TREE DEMO
A
  B
    D
    E
  C
    F
```

All the test cases and .py files are available on this repository.
https://github.com/sghimire2025/MSCS532_Assignment6

Conclusion
Different data structures are suited for different types of problems. Arrays are beneficial when memory is known ahead of time and random access is required. Linked lists excel in dynamic environments where insertions and deletions occur frequently. Stacks are used in recursion, function call management, and undo mechanisms. Queues fit well in scheduling and breadth-first search algorithms. Rooted trees are essential for representing hierarchical systems like file directories or organizational charts

References
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

Klappenecker, A. (2015). *Randomized algorithms III: The selection problem*. Texas A&M University.
https://people.engr.tamu.edu/andreas-klappenecker/csce411-s15/csce411-random3.pdf

GeeksforGeeks. (n.d.). Difference between Array, Queue and Stack. GeeksforGeeks.
https://www.geeksforgeeks.org/dsa/difference-between-array-queue-and-stack/