Assignment 6: Medians and Order Statistics and Elementary Data Structures

Alexandria Roberts

School of Computer and Information Sciences

MSCS531-A01 Computer Architecture and Design

Professor Bass

11/23/2025

Introduction

This assignment focuses on the implementation and analysis of algorithms related to order statistics, more specifically determining the k-th smallest element in an unsorted array. Two approaches are explored: Deterministic Selection using the Median of Medians method and Randomized Selection using Quickselect. Algorithms like this are useful for when the objective is to locate a specific ranked element without sorting the entire dataset, which is not as efficient for larger inputs.

Along with this, the assignment should demonstrate the implementations of elementary data structures including arrays, queues, stacks, and singly linked lists. These structures serve as core building blocks in computer science and are essential for managing data efficiently in real-world applications. (Cormen et al, 2022)

Part 1: Selection Algorithms

Deterministic Selection (Median of Medians)

The deterministic selection algorithm ensures linear time complexity O(n) in the worst case. It does this by dividing the input array into groups of five elements, while finding the median of each group, then looping and selecting the median of those medians to use as the pivot.

The key implementation used in this assignment was this function below:

```
22
23  def deterministic_select(arr: List[int], k: int) -> int:
24      """Median-of-medians select. Returns k -th smallest (1-based)."""
25
```

This function confirms the input, then calls upon the internal looped function select() that then performs the Median of Medians logic. As stated above, the array then divides into groups

of five; medians are extracted, and a pivot is selected to partition the array into lower high values.

Randomized Selection (Quickselect)

Randomized Selection picks a pivot by chance rather than a more structured one. Even though it's worst-case complexity can degrade to $O(n^2)$, instead its expected performance is O(n), and usually performs better than the deterministic method in practice.

The main function implemented is:

```
51
52  def randomized_select(arr: List[int], k: int) -> int:
53      """Randomized Quickselect. Returns k -th smallest (1-based)."""
54
```

The function above used the helper function quickselect() and selects a pivot index at random using random.randint(). The partitioned array and recursion continued onuntil the k-th smallest position can be obtained.

Experimental Results

Several timed experiments were conducted using different input sizes and distributions. The function run_selection_experiments() created results for:

- Sorted arrays
- Random arrays
- Reverse sorted arrays

Timing results then showed that the randomized selection consistently performed faster than deterministic selection, demonstrating its efficiency for practical use cases even though it lacks worst-case guarantees.

Part 2: Elementary Data Structures

The following array and matrices operations were implented:

- Insertion using array_insert()
- Direct access using array_access()
- Deletion using array_delete()
- Matrix access and modification using matrix_get() and matrix_set()

Example:

```
131  def array_insert(a: List[Any], index: int, value: Any) -> None:
132      """Inserts values at index in array a (list). O(n -index)."""
133      a.insert(index, value)
```

Stack Implementation

I used a stack implementing using Python's list structure. This stack followed Last-In and First-Out (LIFO) based principles. The core operations include:

- Pop()
- Push()
- Peek()
- Is_empty()

Example:

```
152
153 class Stack:
154     """Stack using Python list as array."""
155
156     def __init__(self):
157         self.data: List[Any] = []
158
159     def push(self, item: Any) -> None:
160         self.data.append(item)
161
162     def pop(self) -> Any:
163         if not self.data:
164             raise IndexError("pop from empty stack")
165         return self.data.pop()
166
167     def peek(self) -> Any:
168         if not self.data:
169             raise IndexError("peek from empty stack")
170         return self.data[-1]
171     def is_empty(self) -> bool:
172         return len(self.data) == 0
```

Queue Implementation

First-In First-Out (FIFO) behavior is used in the queue. This allows it to use the list-based storage with enqueue and dequeue operations.

Example:

```
175
176  class Queue:
177      """
178      Simple queue using list.
179      Note: dequeue using pop(0) is O(n), which you can mention in analysis.
180      """
181
182      def __init__(self):
183          self.data: List[Any] = []
184
185      def enqueue(self, item: Any) -> None:
186          self.data.append(item)
187
188      def dequeue(self) -> Any:
189          if not self.data:
190              raise IndexError("dequeue from empty queue")
191          return self.data.pop(0)
192
193      def peek(self) -> Any:
194          if not self.data:
195              raise IndexError("peek from empty queue")
196          return self.data[0]
197
198      def is_empty(self) -> bool:
199          return len(self.data) == 0
```

Singly Linked List

When it comes to singly linked lists, it was implemented inorder to demonstrate pointer-based data structure behavior. This includes:

- Insert_at_head
- Insert_at_tail
- Delete_value
- Traverse

Example:

```
class SinglyLinkedList:
    def __init__(self):
        self.head: Optional[ListNode] = None

    def insert_at_head(self, value: Any) -> None:
        self.head = ListNode(value, self.head)

    def insert_at_tail(self, value: Any) -> None:
        new_node = ListNode(value)
        if self.head is None:
            self.head = new_node
            return
        curr = self.head
        while curr.next:
            curr = curr.next
        curr.next = new_node

    def delete_value(self, value: Any) -> bool:
        curr = self.head
        prev = None
        while curr:
            if curr.value == value:
                if prev is None:
                    self.head = curr.next
                else:
                    prev.next = curr.next
                return True
            prev = curr
            curr = curr.next
        return False

    def traverse(self) -> List[Any]:
        result = []
        curr = self.head
        while curr:
            result.append(curr.value)
            curr = curr.next
        return result
```

Discussion

As shown above, the experiments results show that randomized selection is continually faster for the tested input sizes. However, deterministic selection allows theoretical reliability by guaranteeing linear rime complexity. This showcases the trade-offs between predictability and performance, which is considered an important part of algorithm design. These data structures

illustrate how different storage models affect performance efficiency. Array-based stacks and the queues are basic but can help introduce overhead, particularly with dequeue operations that require more memory due to node references.

Conclusion

This assignment helped me to successfully understand and demonstrate both order statistics algorithms and fundamental data structures. The deterministic and randomized selection algorithms help showcase different approaches to solving the same problem with a variety of efficiency trade-offs. At the same time, the implementation of stacks, queues, and linked lists help reinforce continuous understanding of core data management concepts essential when it comes to software development.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.