Medians and Order Statistics & Elementary Data Structures

Avinna Bhattarai

005028547

Course Number: MSCS-532-A01

Assignment6: Medians and Order Statistics & Elementary Data Structures

Dr. Vanessa Ruth Cooper

09/29/2024

**Overview**

This assignment is focused on two main areas:

1. Implementing selection algorithms for finding order statistics (specifically the smallest element).
2. Exploring and implementing basic data structures such as arrays, stacks, queues, and linked lists.

**Part 1: Implementation and Analysis of Selection Algorithms**

**Implementation**

**Deterministic Algorithm: Median of Medians**

The Median of Medians algorithm is a deterministic selection algorithm that finds an approximate median in linear time. This is achieved by recursively selecting the median of medians as the pivot for partitioning. Please refer to the repository for implementation where the file name is *deterministic\_alg.py.*

**Randomized Algorithm: Randomized Quickselect**

Randomized Quickselect is a selection algorithm that works similarly to QuickSort but focuses on the element. It uses a random pivot to partition the array, leading to an expected linear runtime. Please refer to the repository for implementation where the file name is *randomized\_alg.py.*

**Performance Analysis**

**Time Complexity:**

* **Deterministic Algorithm (Median of Medians):**
  + Worst-case time complexity:
* **Randomized Algorithm (Randomized Quickselect):**
  + Expected time complexity:

**Space Complexity:**

* Both algorithms require: space for input, with recursive depth potentially reaching in the worst case.

**Empirical Analysis**

To compare running times empirically, please refer to *comparison\_alg.py*

**Discussion of Results**

Observation might suggest that for smaller input sizes, the difference in time is negligible, but as the size increases, the deterministic algorithm’s overhead might lead to longer execution times due to the complexity of the partitioning process. The randomized algorithm tends to perform faster in practice due to its simplicity, though the worst-case scenario can be quite slow.

**Part 2: Elementary Data Structures Implementation and Discussion**

**Implementation**

Note: Please find implementation of corresponding data structures in file named *data\_structures.py*

**Performance Analysis**

* **Arrays:**
  + Access:
  + Insertion: (amortized)
  + Deletion: (due to shifting elements) (Knuth, 1997).
* **Stacks:**
  + Push:
  + Pop:
  + Peek:
* **Queues:**
  + Enqueue:
  + Dequeue: (if implemented with an array)
* **Linked Lists:**
  + Insertion (if inserting at head)
  + Deletion: (to find the element)
  + Traversal:

**Discussion of Elementary Data Structures**

**1. Arrays**

Arrays are foundational data structures with fixed sizes and fast access times, making them ideal for:

* **Static Data**: When the dataset size is known and won’t change, arrays are optimal due to their access time.
* **Mathematical Computations**: Many numerical algorithms leverage arrays for efficient data manipulation.

However, arrays have limitations, particularly with dynamic datasets. The need to shift elements during deletions can be inefficient, which is why linked lists or dynamic arrays (like Python’s list) are often used in such cases.

**2. Stacks**

Stacks, operating on a Last-In-First-Out (LIFO) principle, are crucial in several areas:

* **Function Call Management**: In programming, the call stack keeps track of active functions. When functions are called, their context is pushed onto the stack and popped off once execution completes.
* **Undo Mechanisms**: Applications like text editors utilize stacks to implement undo functionality, where the last action taken can be reversed.

**3. Queues**

Queues operate on a First-In-First-Out (FIFO) basis and are essential for:

* **Task Scheduling**: Operating systems use queues to manage tasks. Processes are queued for CPU time, ensuring fairness and order.
* **Breadth-First Search (BFS)**: Queues are integral to graph algorithms like BFS, where nodes are explored in layers.

**4. Linked Lists**

Linked lists provide flexibility and efficiency in dynamic scenarios:

* **Dynamic Memory Management**: Linked lists are particularly useful when the number of elements isn’t known in advance, allowing for efficient insertions and deletions.
* **Complex Data Structures**: Many complex structures, such as hash tables and adjacency lists for graphs, rely on linked lists for efficient management of data.

**Conclusion**

Understanding and implementing selection algorithms and elementary data structures is vital for efficient algorithm design and problem-solving. The trade-offs between different approaches highlight the importance of context in selecting the right algorithm or data structure for a given application. From real-time systems to dynamic memory management, these concepts play crucial roles in computer science and software development.

**References**

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

Hoare, C. A. R. (1961). "Quicksort". *The Computer Journal*, 5(1), 10-16.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.