Richa Lahoti
005039141
MSCS-532-M20
Assignment 6

Part 1: Deterministic and Randomized Selection Analysis

Implementation

The deterministic and randomized selection algorithm implementation is in the github repo linked here: https://github.com/rlahoti-cb/MSCS532_Assignment6 . The README.md covers how to run this code to generate the data that is used in the analysis section below. Note, that your numbers may vary slightly due to running the code on a different machine and other variables. The implementation exists in "assignment_6_part_1.py". The results exist in "results_part_1.txt".

Performance Analysis

We can begin by providing a detailed analysis of the time complexity for the deterministic and randomized selection algorithms.

For the randomized pivot value selection approach, the worst-case runtime is $O(n^2)$. In the worst case, a poor pivot selection for the randomized algorithm leads to a highly imbalanced split of the input array. This can occur when the pivot is repeatedly chosen as the largest or smallest value in the input array. Each level of recursion leads to O(n) work with n recursive levels. This leads to a worst case runtime of $O(n^2)$.

For the deterministic algorithm, the worst case runtime is O(n). With the median of median approach, this guarantees that at least ~30% of elements are guaranteed to be less than or greater than the pivot. Recursively selecting the median of medians as a pivot allows for a linear time execution of this algorithm.

We can now discuss the space complexity analysis for the deterministic and randomized selection algorithms.

The randomized selection algorithm takes O(1), or constant, space. This is due to the pivot selection being random and the algorithm not needing to store additional space. The deterministic selection algorithm however, in the worst case, takes O(n) additional space. This is due to storing the medians at each recursive call.

Empirical Analysis

In this section, we can run an empirical analysis comparing the deterministic and randomized selection algorithms on varying input sizes and varying distributions of input data. This will allow us to compare in the real world how these algorithms perform on varying inputs.

The table below captures the results from the execution of both algorithms. The input data sizes tested were 10,000 elements, 100,000 elements, and 1,000,000 elements. The input arrays data types were randomized arrays, sorted arrays, and reverse sorted arrays.

Algorithm	Data Type	Input Size	Time Taken (s)
deterministic	random	10000	0.006901
randomized	random	10000	0.006086
deterministic	sorted	10000	0.004797
randomized	sorted	10000	0.003047
deterministic	reversed sorted	10000	0.005643
randomized	reversed sorted	10000	0.001439
deterministic	random	100000	0.076646
randomized	random	100000	0.097707
deterministic	sorted	100000	0.067547
randomized	sorted	100000	0.036881
deterministic	reversed sorted	100000	0.068763
randomized	reversed sorted	100000	0.031978
deterministic	random	1000000	1.153323
randomized	random	1000000	0.696423
deterministic	sorted	1000000	0.883223
randomized	sorted	1000000	0.90387
deterministic	reversed sorted	1000000	0.796653
randomized	reversed sorted	1000000	0.832028

There are quite a few patterns that we can identify when comparing these two algorithms in a real-world experiment.

For when the data distribution is random, we notice that the randomized algorithm is generally faster. For example, as the input size increases, we see that the randomized algorithm performs noticeably better than the deterministic algorithm on random data. With the test run evaluating 1 million elements, we see that the deterministic algorithm on random data took 1.15s and the randomized algorithm took 0.70s. This is a significant difference. This is unexpected compared to what we expect to occur from a theoretical perspective. Theoretically, the deterministic algorithm should take O(n) time in the worst case and the randomized algorithm in the worst case will take $O(n^2)$ time. Another pattern that is evident is that when the input data distribution

is sorted or reverse sorted, randomized performs well at smaller scales and the deterministic algorithm catches up to match performance at larger scales.

It is quite interesting to note how these algorithms perform on different input sizes and distributions of data. Running this experiment on various input data sizes and data distributions truly allows us to empirically compare the algorithms even though the results may not reflect our expectations based on theory!

Part 2: Elementary Data Structures

Implementation

The implementation of these elementary data structures is in the github repo linked here: https://github.com/rlahoti-cb/MSCS532_Assignment6 . The README.md covers how to run this code to generate a demonstration of the usage of these data structures. The implementation exists in "assignment_6_part_2.py". The results exist in "results_part_2.txt".

Performance Analysis

First, we can begin with analyzing the time complexity for each operation for each data structure in the worst case.

Array – insert O(n), delete O(n), access O(1)
Matrix – insert O(n), delete O(n), access O(1)
Stack – push O(1), pop O(1)
Queue – enqueue O(1), dequeue O(1)
Linked List – insert O(1), delete O(n), traverse O(n)

We can now analyze the tradeoff when choosing the underlying data structures for queues and stacks. When implementing stacks and queues, we can choose between two underlying data structures – arrays or linked lists. This tradeoff is with respect to performance and behavior. Given that arrays are fixed in size, if we know that our input data is of a fixed size then it is preferable to use an array data structure. In the case that our input size is unbounded, it is generally preferable to use a linked list which allows for cheap dynamic resizing at the cost of additional space for storing pointers. Both data structures however allow for constant, O(1), time access operations for a stack's push and pop method.

With respect to the queue data structure, it requires careful implementation to handle enqueue and dequeue operations in constant time. Arrays require management of a circular buffer or element shifts to ensure that these operations maintain O(1) time complexity. Linked lists on the other hand allow for very simple and cheap pointers to the head and tail of a linked list allowing for simple O(1) time complexity the enqueue and dequeue operations.

Finally, we can compare the efficiency of these data structures in different scenarios. In the case that you would like fast, O(1), access to data then arrays and matrices are generally best. For example, an image is represented as a matrix since it allows for highly performant access to the

underlying data. In the case where a user desires behavior to frequently insert or delete elements and the beginning or end then a linked list can generally be helpful. The linked list allows for pointers to the tail or head which enables good performance for this behavior. In the case where we need last in first out behavior, stacks can be well-suited. For example, in an operating system the call stack uses a literal stack. In the case where the desired behavior is handling tasks in order then a queue is a good choice for a data structure. For example, if there is a task queue in an operating system, a queue is a great choice.

Discussion

We can now discuss the practical applications or use cases for each of these data structures. We can also comment on when each of these data structures are preferred due to factors like memory, usage, speed, and ease of implementation.

The use case for arrays or matrices is generally when there is a desire for fast access or data lookups. Matrices do a fantastic job representing 2D or 3D data allowing for matrix operations and calculations. An example of this is image processing where the image is represented as a matrix which allow for highly performant lookups and access to the data. Arrays are generally preferred when the data size is known or bounded. In the case where data size is unbounded and arrays are used, this will lead to dynamic resizing of the array or matrix which is an expensive operation.

The use case for stacks is recursive algorithms which require backtracking. Stacks operate on a LIFO so the last element in is the first element out. This is ideal for backtracking. Another use case of LIFO-based data structures such as stacks are the call stack in an operating system. The last functional call that pops is the first function call that was inserted. Stacks are also the preferred data structure when the latest element matters.

The use case for queues is scheduling algorithms. Queues allow for processing of requests or tasks in a FIFO order. This is a first in first out order. The first request that is ingested is the first request that is processed. The first task that is scheduled is the first task that is processed. These behaviors make the FIFO-based data structures like queues preferred.

Linked lists use case include adjacency lists in a graph representation. A graph composed of nodes and edges can be represented as an adjacency list where each node in the graph is represented as a linked list containing all of the neighbors of that node. This allows for efficient add or removal operations. Linked lists are also generally preferred when the data size for the input is not bounded. This is true in the case of a graph where there can be an unbounded number of nodes and edges added. Linked lists require more space than the other data structures since we have to store not only the value but also pointers to other nodes.

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

GeeksforGeeks. (2025, July). Array Data Structure Guide. https://www.geeksforgeeks.org/dsa/array-data-structure-guide/

GeeksforGeeks. (2025, July). Matrix Data Structure. https://www.geeksforgeeks.org/dsa/matrix/

GeeksforGeeks. (2025, July). Linked List Data Structure. https://www.geeksforgeeks.org/dsa/linked-list-data-structure/

GeeksforGeeks. (2025, July). Queue Data Structure. https://www.geeksforgeeks.org/dsa/queue-data-structure/

GeeksforGeeks. (2025, July). Stack Data Structure. https://www.geeksforgeeks.org/dsa/stack-data-structure/