Bishal Thapa

005030505

MSCS532 - A01: Algorithms and Data Structures

Assignment 6

Median of Medians vs Quickselect

The Median of Medians algorithm finds the kth smallest element in an unsorted array in worst-case linear time, O(n). It works by dividing the array into small groups, finding their medians, and using the median of medians as a pivot. This helps create balanced partitions and avoids worst-case scenarios. On the other hand, Quickselect is a randomized algorithm with expected linear time, O(n). It picks a random pivot, partitions the array, and searches the relevant partition. While it is faster on average, Quickselect can degrade to O(n^2) in the worst case.

Despite this, Quickselect is often faster in practice because it has less overhead. Quickselect uses O(1) extra space for space complexity since it works in place. Median of Medians requires O(logn) space due to recursion. Quickselect is usually preferred because it is faster, but the Median of Medians is helpful when you need guaranteed worst-case performance. Both algorithms solve the selection problem efficiently without needing to sort the array.

Here is the performance Comparison of Median of Medians vs Quickselect for sample size n = 10000.

Data Type	**Median of Medians (seconds)**	**Quickselect (seconds)**
Sorted Data	0.003987	0.000987
Sorted Descending Data	0.003012	0.001634
Random Data	0.004546	0.000993

Random Data with Duplicates	0.003834	0.001539

Quickselect is much faster than the Median of Medians for all data types at an input size of 10,000. It runs about 3–4 times quicker, with the best time of 0.000987 seconds for sorted data. The Median of Medians is more stable across different inputs but takes longer due to extra steps. Quickselect is more efficient in most cases, especially for random data. However, in the worst case, Quickselect can be much slower. In such cases, the Median of Medians is a better choice because it always guarantees a good performance.

Here is the performance Comparison of Median of Medians vs Quickselect for sample size n = 20000.

Data Type	**Median of Medians (seconds)**	**Quickselect (seconds)**
Sorted Data	0.006212	0.001666
Sorted Descending Data	0.005506	0.001879
Random Data	0.007705	0.002875
Random Data with Duplicates	0.006907	0.002412

For an input size of 20,000, Quickselect again performs significantly better than the Median of Medians, running up to four times faster. Its best time is 0.001666 seconds for sorted data. The Median of Medians, though slower, remains stable across different data types due to its structured approach. After running two different datasets, we can confirm that Quickselect is more efficient for most cases, especially with random data. However, it can slow down significantly in the worst case. In such situations, the Median of Medians is the better choice because it guarantees consistent performance regardless of the input.

Implementing Data Structures

The complexity of basic operations in data structures depends on how they are implemented. For arrays, accessing an element by index is O(1) because arrays allow direct access in constant time. However, inserting or deleting elements in the middle requires shifting other elements, which takes O(n) time.

For matrices, which are 2D arrays, operations like accessing, inserting, or deleting an element at a fixed index also have O(1) complexity. In stacks, both push (add) and pop (remove) operations are O(1) because elements are only added or removed from the top. For queues, adding an element (enqueue) is O(1), but removing an element (dequeue) can take O(n) in Python lists because elements need to be shifted.

In singly linked lists, inserting or deleting at the head is O(1). However, inserting or deleting at other positions requires traversing the list, which takes O(n) time.
For binary trees (or rooted trees), insertion and searching are O(log n) in balanced trees. In unbalanced trees, these operations can become O(n) because the tree may resemble a linear structure.

There are trade-offs between using arrays and linked lists when implementing stacks and queues. Both structures allow O(1) time for push and pop operations for stacks. Arrays have better cache locality because elements are stored in continuous memory, which can improve performance. However, linked lists use dynamic memory to avoid resizing problems when elements are added or removed. For queues, the differences are more apparent. Arrays can enqueue (add) in O(1) time, but dequeue (remove) in Python lists takes O(n) time because elements must be shifted. Linked lists, on the other hand, handle both enqueue and dequeue in O(1) time. However, they use extra memory for storing pointers, which increases overhead.
In summary, linked lists are more flexible for efficient operations, while arrays can be faster for operations limited to one end, like in stacks.

The efficiency of data structures depends on the situation they are used in. For example, arrays work well when you often need to access elements by index, like in a fixed-size list. They are also good when the data size is known beforehand and does not change much. However, if you need to insert or delete elements in the middle often, arrays can slow down because shifting elements takes O(n) time. Linked lists, on the other hand, are better for cases where you need dynamic memory and frequent insertions or deletions at any position. They handle these operations in O(1) time without needing to resize or shift elements. However, linked lists use extra memory for pointers and do not allow efficient random access, making them less suitable for scenarios where you often need to access elements by index.

Stacks and queues work well with linked lists when dynamic resizing is needed, especially for queues where elements are often removed from the front. However, arrays are more space-efficient and cache-friendly if you only need to push/pop or enqueue/dequeue at one end and want to save memory.

In real-world situations, different data structures are suited for different problems based on performance, memory use, and ease of implementation. Arrays are commonly used in tasks that need fast, indexed access, like image processing, where pixel data is accessed quickly for transformations or filters. They are also used in sorting algorithms like quicksort and mergesort because their memory structure allows efficient element access. However, arrays struggle with dynamic operations like insertions and deletions, especially when the data size changes often. This is where linked lists perform better. They are useful in circumstances where memory needs to be flexible. Some examples are real-time systems or event-driven simulations, as they allow efficient insertions and deletions without shifting elements. This makes them ideal for tasks with unpredictable data changes.

Choosing what data structure to use often depends on balancing memory efficiency, speed, and ease of use. Arrays are faster for random access due to their memory structure. However, they can be inefficient when resizing or shifting elements during insertions and deletions. This makes them less suitable for dynamic data. On the other hand, linked lists manage memory flexibly by allocating it only when needed. This makes them better for memory-constrained environments or tasks requiring frequent data changes. However, linked lists use extra memory for pointers, and accessing elements sequentially can be slower than with arrays. In short, arrays are best when quick access and low memory use are important, while linked lists are better for tasks needing dynamic memory and frequent changes.