**PART 1:**

Implementation:

The implementation includes two algorithms which are the deterministic selection the Median of Medians approach and the randomized selection with Quickselect. In the deterministic algorithm, the given array is split into groups of five, and find the median of each group. The above medians are then processed Recursive to derive the final median of medians which is used to create the partitioning. It achieves an optimal partition and also gives a worst-case linear time. As for the algorithm, unauthorized at each recursive step a random pivot in the array is sorted upon it, dividing the array in two, thus it has an expected time complexity of O(n). Both implementations care about such situations, for example, when the array contains elements that are the same in some places, and they use in-place partitioning to save space. This makes the algorithms more powerful to give kth smallest element for various types of input arrays.

Performance Analysis:

There are two varieties of sorting which are deterministic and randomized though both of them have the same time complexity of O(n) that is required for selecting the kth smallest element in the list. The deterministic algorithm (Median of Medians) also ensures we get worst-case linear time O(n) by use of pivot elements to balance partitions. This will make sure that the number of tasks which are to be completed reduces by a half through each call made recursively. On the other hand, the randomized algorithm, Quickselect, is presented, which takes linear expected time O(n) because it always chooses a pivot randomly, which gives good partitions on average but can make a worst-case partition of O(n^2). However, since both partition and quick sorting happen in place, all three algorithms require O(1) extra space, any extra space required may come from the recursion stack.

Empirical Analysis:

The analysis of empirical results was done to determine the effectiveness of the deterministic and randomized algorithms. When testing the presented deterministic and randomized algorithms, it was observed that both the algorithms always found out the correct kth smallest element. The big O of observed performance for both implementations was consistent with the scale of the inputs used and the deterministic algorithm did not take longer to implement the grouping and finding of the median steps because it worked on a much smaller scale. Specifically for the larger arrays, the randomized algorithm is likely to be faster, though sometimes this does not include the cost of finding the median as the pivot. However, as a deterministic algorithm, it does not suffer from skewed partitions and variations in performance depending on the input distribution and is better for worst-case utilization.

**PART 2:**

Performance Analysis:

We discuss the time complexity for operations in each of the data structures that we have implemented. Arrays, where we have to know the index number of the element we wish to access, are good when we want to access the elements because they take O(1), meaning constant time. Nonetheless, insertion and deletion may require elevated time at O(n) due to shifts in memory storage. Implementations of stacks and queues with arrays have O(1) time complexity for push (to a stack), pop (from a stack), enqueue (to a queue), and dequeue (from a queue), provided that we don’t have to resize the array. While in a linked list insertion and deletion in the beginning take O(1) time for insertion and deletion and for search or deletion anywhere in the linked list takes O(n) time. The tree structure based on linked lists used successfully in the design principle provides a powerful possibility of developing hierarchic relations with operations in which time complexity is equal to О(n) and which designates the number of nodes in the tree.

Trade-offs Between Arrays and Linked Lists:

It is true for most instances that if the indices of the elements to be retrieved are known, then arrays can respond in constant time. The other type Random linked lists allow insertion and deletion operations to take place at any position but has a better time complexity of O (n) to access any element within it. Comparing the Stacks and Queues, the array as well as the linked list perform equally well as far as the fundamental operations are concerned that is insertion at the top, deletion at the top and insertion at the back, deletion at the front. However, when the size of the stack or queue is predetermined then the arrays have many advantages in implementation, but in case when the size of the stack or queue is not specified then linked lists are better in implementation because of the property of dynamic memory allocation.

Practical Applications:

The arrays are used where it is necessary to access the elements including in memory management and systems level. The queues are applied in breadth-first search and the scheduling systems in operating systems. Most commonly randomized linked lists are used in the situation where many insertions and deletions are expected such as dynamic storage management, and lists as the basis for other abstract types of data like hash tables or graphs. Linked lists as the implementation of rooted trees are necessary to depict a simple form of computational structures such as file systems and organization charts. As it has been implemented, arrays are favoured when the size of the required dataset can be predetermined and, at the same time, the time for accessing a specific element is of crucial importance. Linked lists are preferred where a large volume of data is anticipated and there are numerous insertions and deletions anticipated.