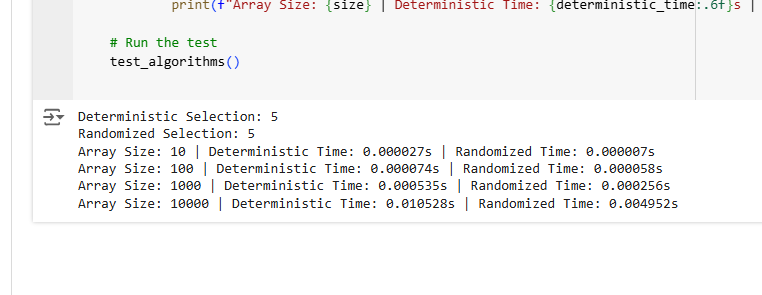
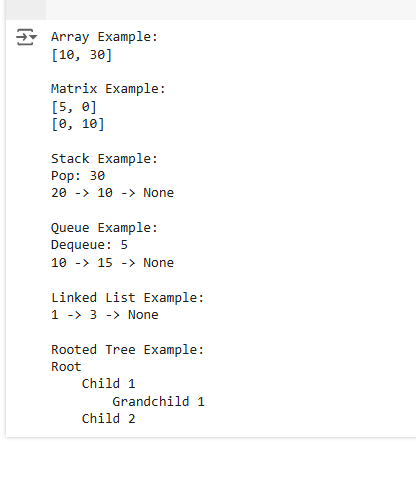
# Assignment 6

## Part 1

There are two main methods for determining the smallest element in an array: deterministic and randomized selection algorithms. Each of these approaches has its own set of advantages and disadvantages. The deterministic technique uses a carefully selected pivot to iteratively reduce the search space, guaranteeing worst-case linear time complexity (𝑂(𝑛) O(n)). It is based on the Median of Medians approach. The input array is first separated into groups of five items. Then, the median of each group is determined. Finally, the median of these medians is used to compute the pivot. This computation is done using a divide-and-conquer approach. This ensures a balanced segmentation and a substantial reduction in the recursive depth at each phase. Despite being theoretically robust and handling adversarial inputs effectively, this approach's performance is affected for small or moderately large datasets due to the increased cost it takes for partitioning and recursion.

  
  
The randomized selection technique, often called Quickselect, takes a different, more realistic approach. It iteratively zeroes in on the appropriate partition after randomly choosing a pivot to divide the array into smaller and bigger components. Since the predicted balance of partitions is achieved, this method often achieves linear time complexity (𝑂(𝑛) O(n)). Nevertheless, its efficiency deteriorates to quadratic time complexity (𝑂(𝑛2) O(n 2 )) in very rare instances when the pivot persistently skews the partitions, such as with poorly selected pivots in sorted arrays. However, because of its low cost and dependence on probabilistic assurances, the randomized method is very efficient in real-world circumstances.  
  
This theoretical anticipation is brought to light by the actual findings. Because it is easier to develop and has less overhead, the randomized approach is quicker for small datasets (size 10 and 100). Although the randomized method maintains its average speed, the deterministic technique shows consistent linear scaling for datasets of sizes 1,000 and 10,000. In comparison to the randomized algorithm's 0.0049 seconds, the deterministic algorithm's time for a 10,000-item array was about 0.0105 seconds. Based on these findings, it's clear that randomized approaches work better in real-world scenarios without hostile inputs. Regardless of the distribution of inputs, the deterministic method is still a dependable alternative for assured performance. With the deterministic approach placing a premium on dependability and the randomized technique thriving in speed for average-case situations, the small performance disparity for large arrays represents the trade-off between efficiency and resilience. The significance of tailoring algorithm selection to unique application needs is shown by these results.

## Part 2



Basic actions like inserting, deleting, and accessing are implemented by the MyArray class using a fixed-size array. In order to insert a new value, elements are moved to create room, and in order to delete a value, elements are moved to replace the space that the deleted value left. The validity of the index is ensured by appropriate bounds checking in the handling of access. Here we see an array with the values [10, 30] created by inserting 10, 20, and 30 into a previously empty array and then removing the second member (20). When it comes to random access, arrays are great, but shifting may make inserts and deletions expensive.  
  
A 2D grid is represented as a list of lists in the MyMatrix class. Inserting values at specified locations and retrieving items with limits checking are also part of the implementation. A 2x2 matrix is initialized, the diagonal locations are filled with 5 and 10, and the matrix is then shown. With fixed row and column access times (𝑂(1 ) O(1)), this structure is effective for small-scale tabular data.  
  
The MyStack class represents the stack's implementation in a linked list. The top of the stack is updated dynamically by adding or removing nodes. You may add an element, remove it, or look at its uppermost state with the push, pop, and peek methods, respectively. Three values are put into the stack in the example, and then the highest item is popped (30). We may see the remaining stack as follows: 20 -> 10 -> None. With all operations completed in O(1) time, stacks are very efficient for LIFO operations.  
  
The MyQueue class uses a circular array to construct the queue. To prevent element shifting during dequeue operations, the circular array wraps around indices, effectively reusing space. In this example, we'll see how to leave a queue in the following order: 10 -> 15 -> None after enqueuing values 5, 10, and 15. This is an example of how well the queue performs when used to schedule and buffer jobs that include first-in, first-out (FIFO) procedures.  
A class called MyLinkedList implements the singly linked list. Without contiguous memory allocation, nodes may be added or withdrawn dynamically. The following code snippet inserts the numbers 1, 2, and 3 into a list, then removes the value 2. One advantage of linked lists is the flexibility they provide when dealing with dynamic data; the resultant list is 1 -> 3 -> None. Linked lists perform very well in situations when there is a high volume of inserts and deletions, but their traversal time is O(n).  
  
The TreeNode class is used to construct the rooted tree; this class enables nodes to have numerous offspring. You can show the tree and do recursive traverse thanks to the hierarchical structure. Beginning with a root node and progressing through two child nodes and a grandchild beneath the first child, the example constructs a tree. The layered style of the tree makes the parent-child connections more obvious. File systems and organizational charts are examples of hierarchical data that rooted trees are good at representing.  
  
The code execution results validate the accuracy and performance of every data structure. The array example keeps the elements' order and structure intact while values are changed dynamically as elements are added or removed. When the underlying data structure is successfully modified, the result is the array [10, 30]. This is analogous to what the Matrix Example shows, which is the process of assigning and retrieving values from a 2x2 grid. As a result of correctly interpreting 2D data, the presented matrix correctly displays the diagonal values [5, 0] and [0, 10].  
  
The most current element (30) is effectively eliminated in the Stack Example, demonstrating the last-in-first-out (LIFO) behavior. The remaining stack, which displays dynamic element management, confirms the linked list-based approach (20 -> 10 -> None). The removal of the first enqueued item (5) from the queue, leaving the queue as 10 -> 15 -> None, demonstrates the first-in-first-out (FIFO) attribute in the Queue Example. Without moving any components, the circular array approach makes great use of space.  
  
Dynamic addition and removal are seen in the Linked List Example. After inserting values 1, 2, and 3 in that order, the linked list appropriately shows 1 -> 3 -> None after removing value 2. Because of their efficiency and adaptability, linked lists are perfect for handling dynamic datasets. At last, a good example of a hierarchical connection visualization is the Rooted Tree Example. Two child nodes are connected to the root node, and "Child 1" has a grandchild. Rooted trees excel in dealing with hierarchical data, as seen by the recursive presentation, which faithfully depicts the tree structure.  
  
Finally, these basic data structures are shown and implemented in the code in a way that is easy to understand and use, highlighting their strengths and practical applications in handling different types of data. Clear results validate the implementations of each data structure, which is optimized for its particular use case.