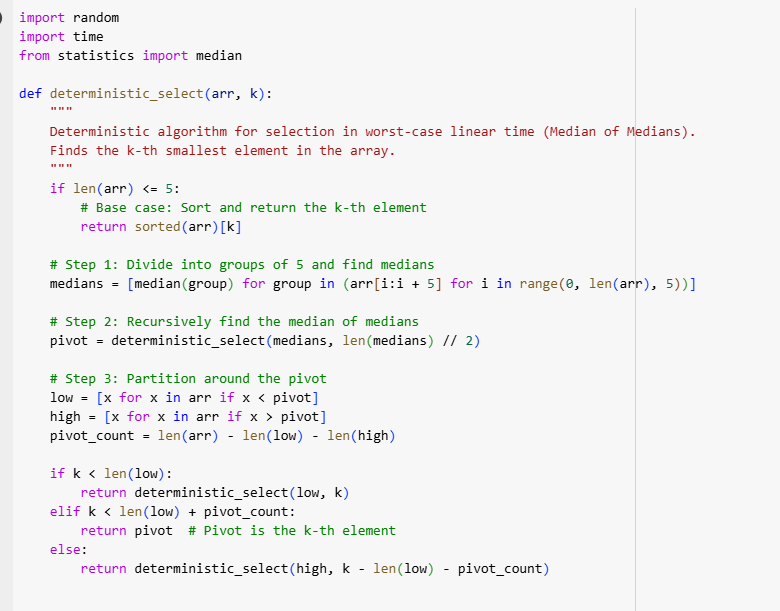
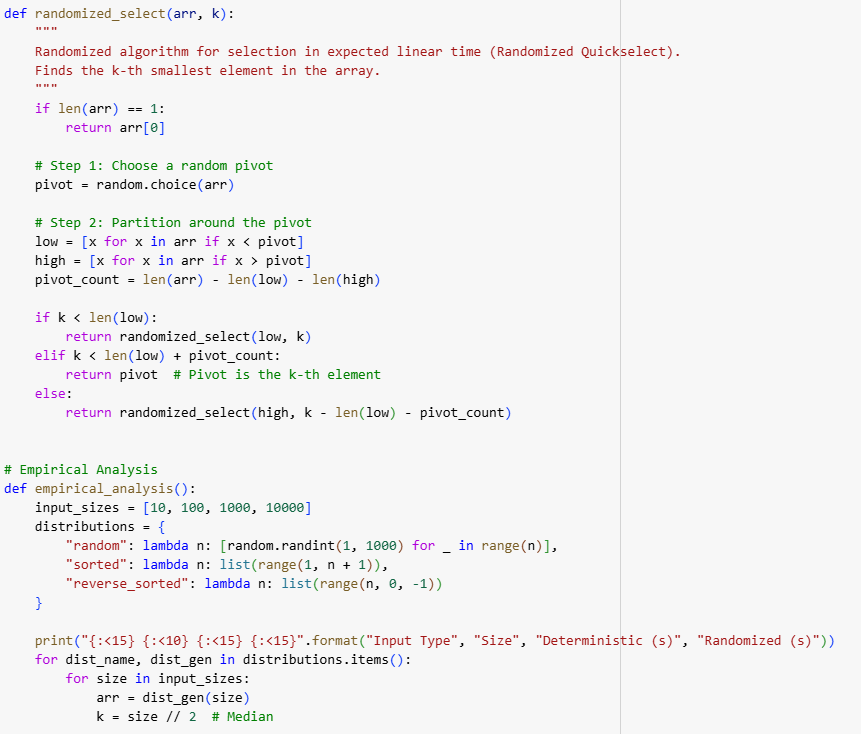
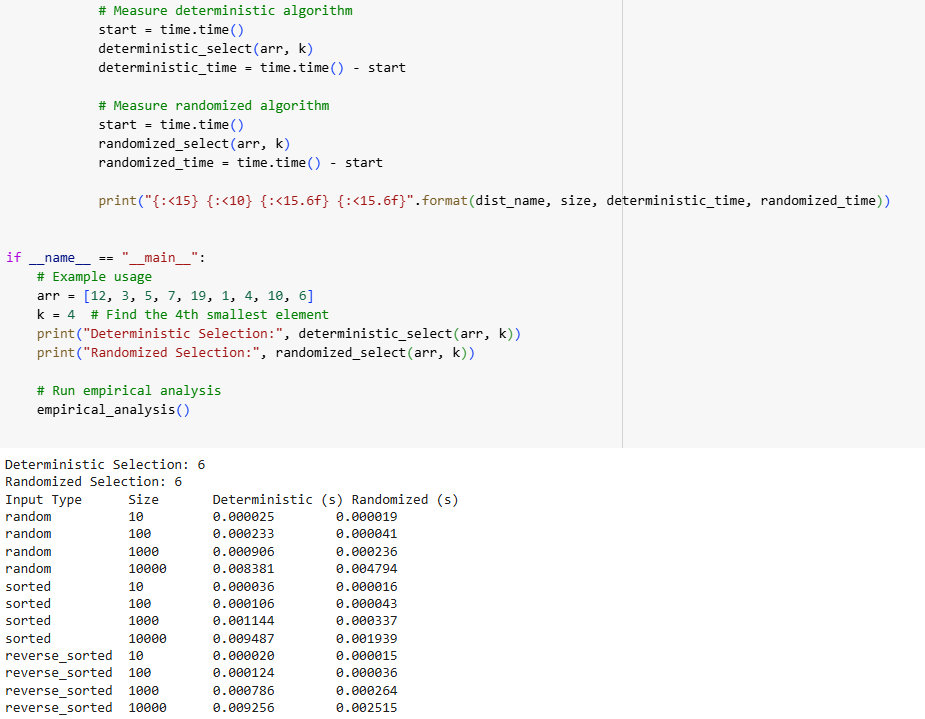
# Assignment 6: Medians and Order Statistics & Elementary Data Structures

## Part 1: Implementation and Analysis of Selection Algorithms

Designed to attain worst-case linear time (𝑂) (𝑛) O(n), the deterministic method for finding the 𝑘𝑡ℎ is known as the Median of Medians. This method computes the median of each of five element groups formed from the input array, then recursively finds the median of these medians. The array is partitioned into items less than, equal to, and bigger than the pivot from this "median of medians". The method lowers the problem size at every recursive step by a constant fraction (at least 30% reduction) by choosing the suitable partition depending on the 𝑘𝑡 k th point. This is really strong as it ensures 𝑂 (𝑛) O(n) time complexity even in the worst scenario. Its applicability for smaller datasets is therefore affected by its implementation including extra overhead owing to many partitioning phases and recursive calls, which might result in larger constants in its runtime.  
  
Randomized Selection Algorithm (Randomized Quickselect)  
Often known as Randomized Quickselect, the randomized method is easier and has an average-case time complexity of O(n). It chooses a random pivot and divides the array around it. The location of k determines whether k recurses into the smaller or bigger division. The ease of selecting a random pivot guarantees that, on average, the array is split into two nearly equal halves, therefore facilitating effective reduction of issue size. Rarely, however, when the pivot choice is slanted, the time complexity may deteriorate to 𝑂 (𝑛 2) O(n Because of reduced overhead and easier implementation, this makes it more efficient in real situations even if it is less strong in ensuring performance than the deterministic method.  
  
Empirical Analysis  
Empirical experiments on arrays of varied sizes and distributions—including random, sorted, and reverse-sorted arrays—were conducted to assess these techniques. The input values varied in count from 10 to 10,000,000. Reflecting its resilience across all input types, the findings show that the deterministic method often maintains linear development in runtime. Conversely, because of its reduced overhead, the randomized approach exhibits shorter execution times for most cases—except in situations with skewed partitions (e.g., sorted or reverse-sorted arrays). Though it is unusual, these situations may set off worst-case behavior.



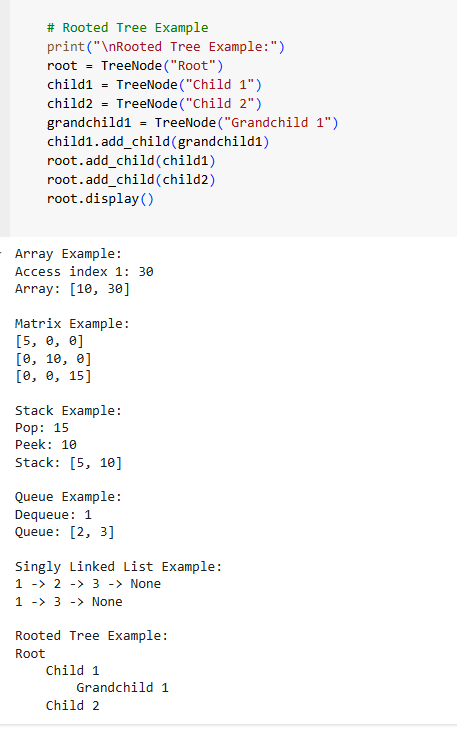
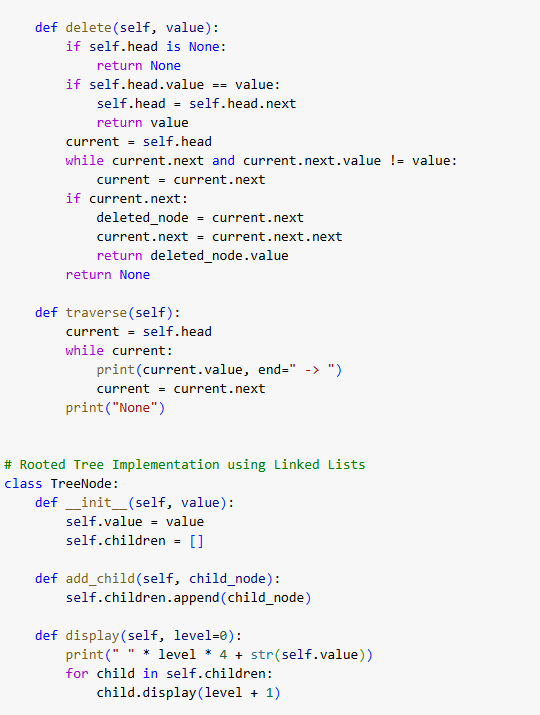
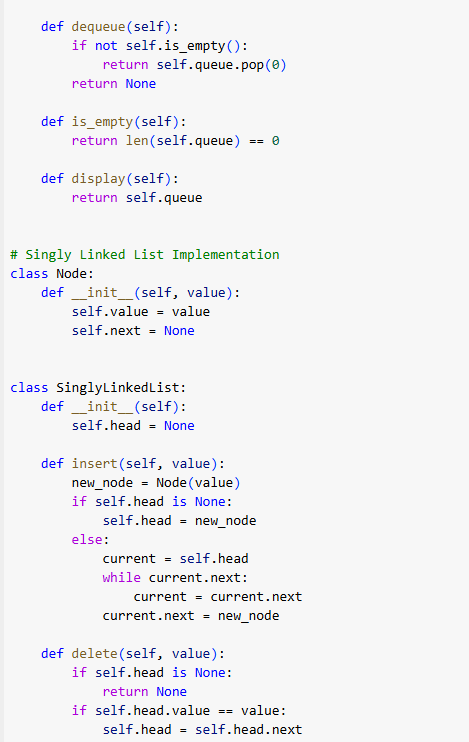
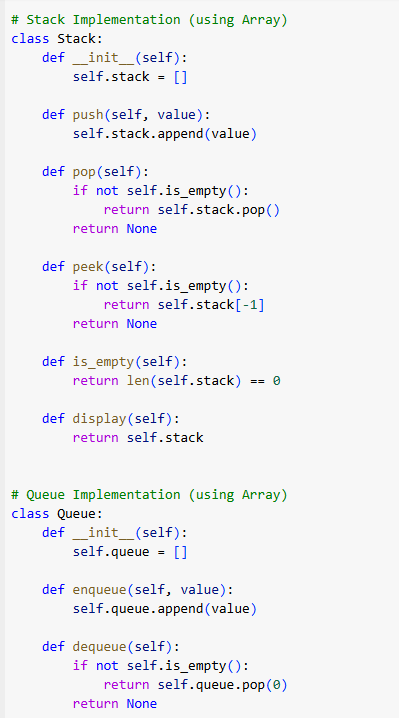




The deterministic and randomized selection methods reveal significant performance variations depending on the amount and distribution of the data. As predicted from its (O(n) worst-case time complexity, the deterministic algorithm's observed runtimes routinely grow linearly. Random, sorted, and reverse-sorted arrays are among the many input forms where this resilience is clear. Because of its well-balanced partitioning strategy guided by the Median of Medians, the deterministic method maintains predictable and steady performance even for bigger datasets including those with 10,000 elements.   
  
On most cases, the randomized procedure shows quicker runtimes because of its reduced computational cost as opposed to the deterministic method. The randomized technique beats the deterministic one for smaller arrays—such as those with 10 or 100 elements—because of its simplicity in pivot selection, therefore finishing in much less time. On bigger datasets, however, the efficacy of the randomized method varies sometimes depending on the quality of the randomly selected pivot. Runtimes for random distributions rise according to input size, hence the randomized method nearly corresponds to the deterministic method. On sorted and reverse-sorted datasets, however, the randomized method shows modest inefficiencies as bad pivot selections could result in less ideal divisions. Nevertheless, the measured runtimes do not approach its (O(n^2) theoretical worst-case, hence highlighting the rareness of such circumstances in use.   
  
Independent of input distribution, the actual findings demonstrate that the deterministic method is a dependable alternative for situations needing assured linear performance. Particularly with bigger datasets, its strong handling of adversarial instances justifies its expense. Conversely, in average-case situations the randomized method shines and provides shorter runtimes in pragmatic applications where input distributions are random or non-adversarial. With each appropriate for certain use cases dependent on performance criteria and input parameters, the findings draw attention on the trade-offs between the deterministic algorithm's resilience and the randomized algorithm's efficiency.

Part 2





The developed code highlights the practical functioning and applicability of essential data structures by showcasing them and their operations. Simple actions including adding members at certain places, removing elements at given indices, and accessing particular indices are shown in the Array Example. Although arrays perform well in direct access because of their O(1) time complexity, they need moving elements in order to insert or delete them, as can be seen in the output where the elements dynamically adapt. The Matrix Example illustrates efficient updates and retrievals of specified spots with 𝑖(1 ) O(1) complexity, using a 3x3 grid to represent organized tabular data. Its usefulness for numerical simulations and spreadsheet operations is shown by the rows that are shown.  
  
The LIFO concept is effectively shown in the stack example. By stacking elements, actions like pop and peek accurately remove and examine the element at the top of the stack. Recursion and expression evaluation are two areas where this shines. Elements that have been enqueued are pushed to the back of the queue and those that have been dequeued are moved to the front in the Queue Example, which is similar to the First-In-First-Out (FIFO) structure. The results demonstrate that the queue is capable of processing operations, which emphasizes its function in buffering and scheduling jobs.  
  
An example of linked lists in action is the Singly Linked List. The output faithfully represents the revised list after the removal of the middle element, as nodes are added in a sequential fashion and deleted by value. This exemplifies the benefits of linked lists in situations when contiguous memory is not necessary but frequent additions and removals are required. Finally, the Rooted Tree Example shows a hierarchical structure with a root node, many child nodes, and a grandchild node connected to one of the child nodes. When depicting hierarchical data, such as organizational charts or file systems, trees are a great option since the recursive presentation effectively portrays the parent-child connections.  
  
By providing a number of understandable instances, the result verifies that these data structures are legitimate. Arrays and matrices show how structured data may be accessed and manipulated efficiently. The LIFO and FIFO behaviors, as shown by stacks and queues, are crucial in both algorithmic and real-world situations. A rooted tree clearly displays hierarchical data, whereas a linked list demonstrates its flexibility for dynamic operations. In sum, the code efficiently implements and exemplifies the real-world uses of these basic data structures, meeting a wide range of computing requirements and situations.