

Assignment 2

Prepared by MahmoudSoliman

1.

<u>Feature</u>	<u>Priority Queue (Heap-based - Binary Heap)</u>	<u>Dynamic Array (Resizable Array)</u>	<u>Dynamic Stack (Linked List)</u>
Description	Data structure that allows prioritized access to elements, where the element with the highest priority is served first. Implemented using a binary heap (a complete binary tree satisfying the heap property).	Contiguous block of memory that dynamically resizes as needed to accommodate more elements. Provides direct access to elements by index.	A LIFO (Last-In, First-Out) data structure where elements are added (pushed) and removed (popped) from the top. Implemented using a linked list.
Implementation	Typically implemented using a binary heap. Parent node's priority is always higher (or lower, depending on min/max heap) than its children. Uses an array internally but maintains the heap property.	Array-based. When the array is full, a new array with a larger capacity is allocated, and elements are copied over.	Linked list-based. Each element is stored in a node that contains the data and a pointer to the next node.

- the time and space computational complexity of your data structures

<u>Feature</u>	<u>Priority Queue (Heap-based - Binary Heap)</u>	<u>Dynamic Array (Resizable Array)</u>	<u>Dynamic Stack (Linked List)</u>
Time Complexity - Insertion (Enqueue)	$O(\log n)$ [Insertion into the heap and maintaining the heap property]	$O(1)$ Amortized [Typically $O(1)$, but $O(n)$ when resizing]	$O(1)$ [Adding a new node to the front of the list]

Time Complexity - Deletion (Dequeue - Highest Priority)	$O(\log n)$ [Removing the root and re-heapifying]	$O(n)$ [Removing from the beginning requires shifting all elements.] $O(1)$ at the end.	$O(1)$ [Removing the top node]
Time Complexity - Access by Index	N/A [Priority Queues are not designed for access by index. If random access is necessary, it is a sign that this might not be the correct structure]	$O(1)$ [Direct access to the element at the specified index]	$O(n)$ [Requires traversing the linked list from the head]
Time Complexity - Search	$O(n)$ [Not a primary operation for a standard heap-based priority queue. Requires iterating through all elements.]	$O(n)$ [Requires iterating through the elements]	$O(n)$ [Requires traversing the linked list]
Space Complexity	$O(n)$ [Stores all n elements in the heap]	$O(n)$ [Stores all n elements in the array. Can have some unused space allocated due to resizing.]	$O(n)$ [Stores n elements in linked list nodes, plus overhead for pointers]

- a table summarizing the measured performance

--- Analysis Table (Random Data) ---

Data Size	Dynamic Array Time (s)	Dynamic Stack Time (s)	Priority Queue Time (s)
100	0.000052	0.000087	0.000118
1,000	0.000174	0.044329	0.001013
10,000	0.001731	0.005937	0.012018
100,000	0.023777	0.071539	0.291992
250,000	0.053460	0.864560	1.003779
500,000	0.107475	1.756041	2.482018
750,000	0.187502	1.536199	4.352568
1,000,000	0.229306	1.887859	5.867494

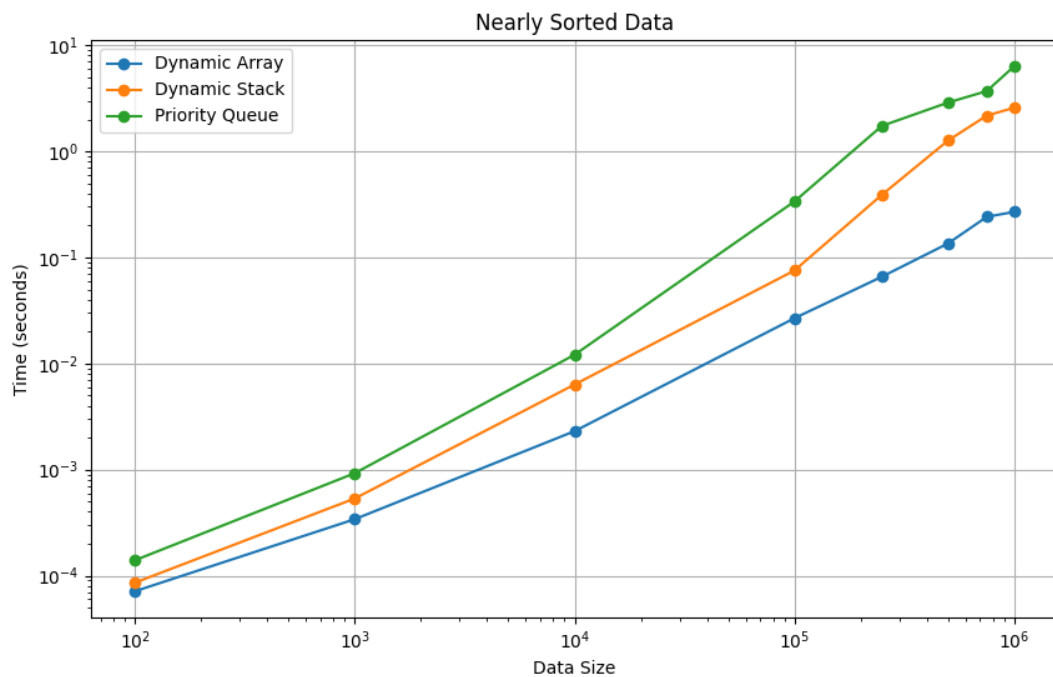
--- Analysis Table (Sorted Data) ---

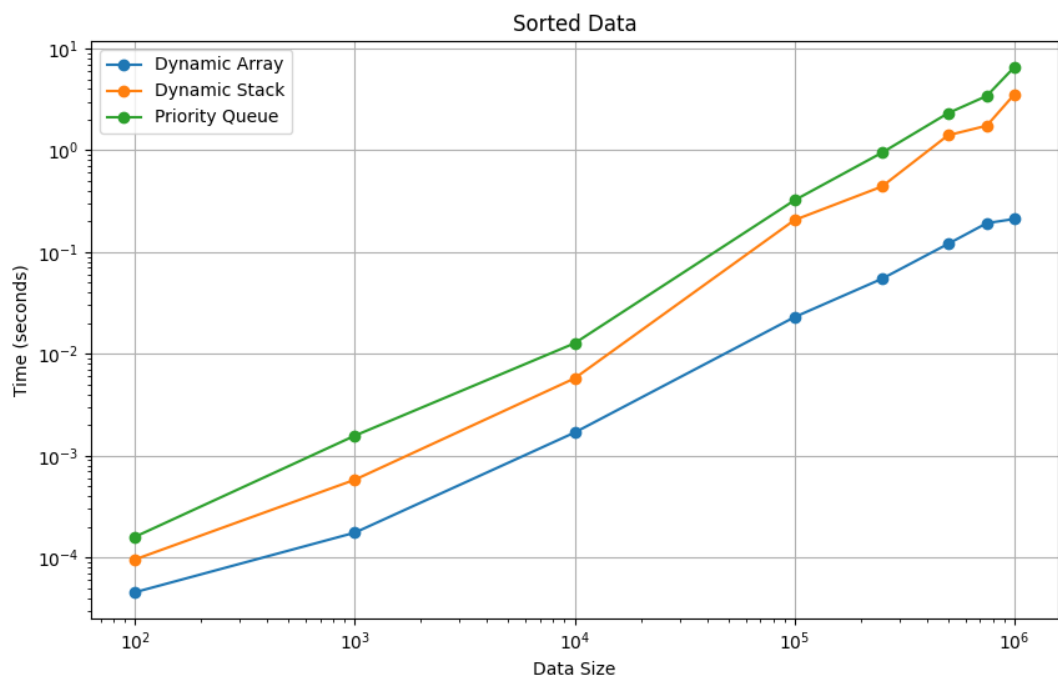
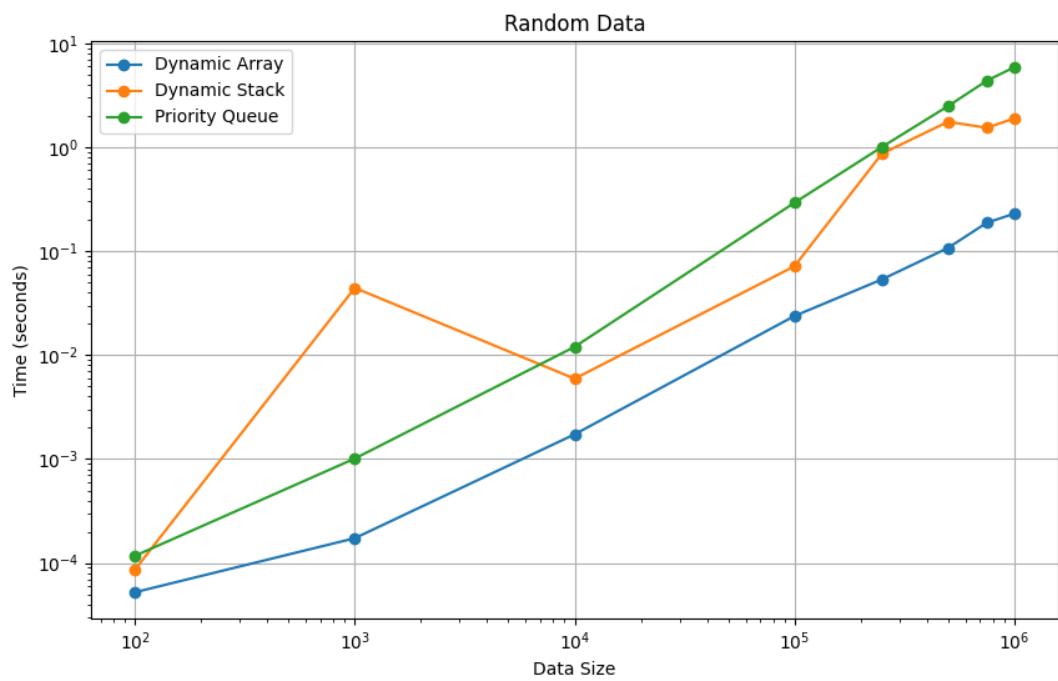
Data Size	Dynamic Array Time (s)	Dynamic Stack Time (s)	Priority Queue Time (s)
100	0.000046	0.000096	0.000160
1,000	0.000175	0.000580	0.001571
10,000	0.001687	0.005749	0.012712
100,000	0.022945	0.206252	0.322420
250,000	0.054612	0.440210	0.946135
500,000	0.120412	1.404718	2.315122
750,000	0.192317	1.743445	3.419249
1,000,000	0.211398	3.528738	6.541796

--- Analysis Table (Nearly Sorted Data) ---

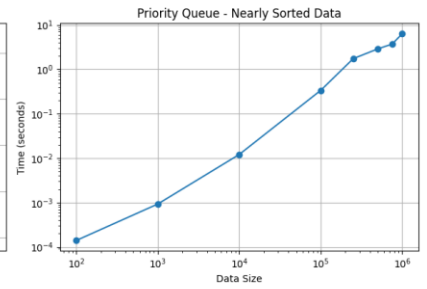
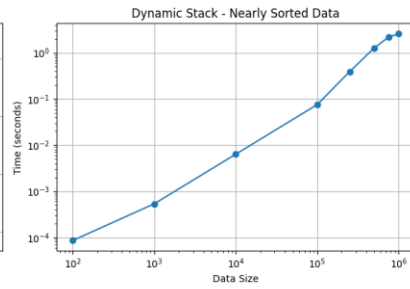
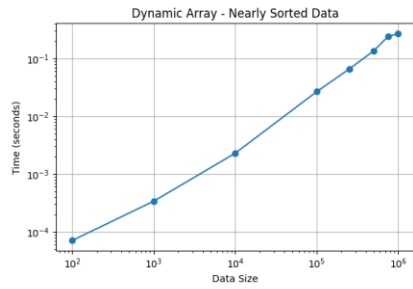
Data Size	Dynamic Array Time (s)	Dynamic Stack Time (s)	Priority Queue Time (s)
-----	-----	-----	-----
100	0.000071	0.000086	0.000140
1,000	0.000341	0.000534	0.000925
10,000	0.002303	0.006340	0.012103
100,000	0.026740	0.075292	0.336963
250,000	0.065826	0.389778	1.739223
500,000	0.135630	1.266640	2.887046
750,000	0.241516	2.174072	3.704250
1,000,000	0.269179	2.581679	6.351282

- lots showing the run time complexity for large enough data. You can generate synthetic dataset using Python functions. Aim for at least one million data points. Then, run your program with different implementations and compare their run time in a plot. Include this analysis in your pdf file and explain if it supports your analysis. Remember to upload your dataset on the web and include a link to the dataset in the pdf file.

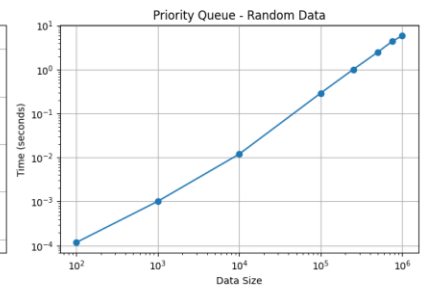
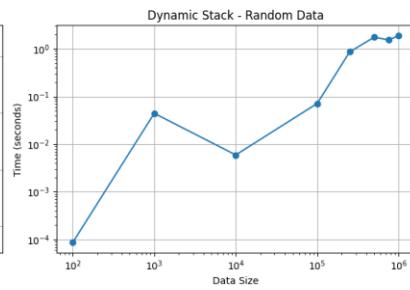
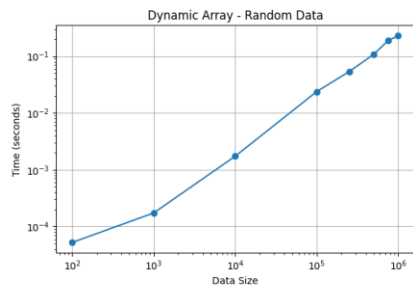




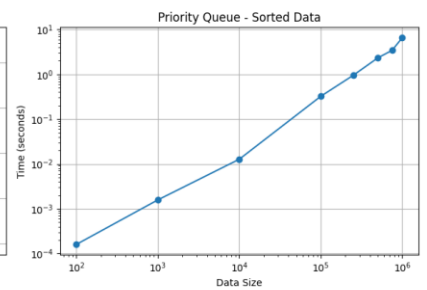
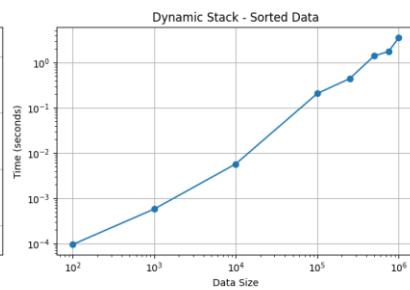
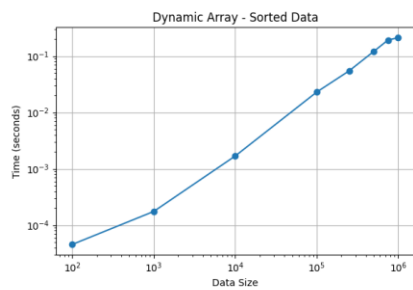
Nearly Sorted Data



Random Data



Sorted Data



- **an explanation of why you selected those data structures (what attracted them; this is totally subjective and personal).**

For this assignment, I selected the Priority Queue, Dynamic Array, and Dynamic Stack because they represent a compelling trio of fundamental and widely applicable data structures. My attraction to them stems from a combination of their inherent elegance, their versatility, and the interesting trade-offs they embody.

- **Priority Queue (Heap-based): The Allure of Prioritization**

What draws me to the Priority Queue is its ability to efficiently manage data based on importance. In a world where resources are often limited and tasks need to be prioritized, the concept of a data structure that intrinsically handles prioritization is deeply appealing. The heap-based implementation, in particular, fascinates me. The way it maintains the heap property through clever algorithms like heapify is a testament to efficient design. I am particularly interested in exploring the performance differences between binary heaps (simplicity) and Fibonacci heaps (theoretical efficiency), especially in scenarios with frequent priority updates. It feels like a powerful tool for orchestrating complex systems. Beyond its practical applications in task scheduling, graph algorithms, and event handling, the Priority Queue embodies a principle of order and efficiency that resonates with my own problem-solving approach.

- **Dynamic Array (Resizable Array): The Foundation of Flexibility**

The Dynamic Array, while seemingly simple, is the bedrock upon which many other data structures are built. It's the workhorse of the collection world. My fascination lies in its balance between providing direct access to elements ($O(1)$ lookup) and its ability to adapt to changing data sizes. The resizing mechanism, while potentially expensive ($O(n)$ in the worst case), is a clever way to overcome the limitations of fixed-size arrays. I find the amortized analysis of insertion ($O(1)$ amortized) particularly interesting, as it demonstrates how occasional costly operations can be balanced out by frequent inexpensive ones. Understanding the Dynamic Array is crucial for understanding how higher-level data structures like Python lists work under the hood. It's a reminder that even the most sophisticated tools are often built upon surprisingly simple foundations. Its fundamental nature makes it an essential element in any programmer's toolkit, making it highly relevant for this assignment.

- **Dynamic Stack (Linked List based): The Elegance of Simplicity and Order**

The Dynamic Stack, implemented using a linked list, appeals to me through its clean, LIFO (Last-In, First-Out) behavior. It's the embodiment of a "stack of plates" – the last plate placed on top is the first one removed. What I appreciate most is its consistent $O(1)$ push and pop operations, regardless of the number of elements in the stack. The linked list implementation ensures that there are no resizing costs, unlike array-based stacks. The Stack's importance

extends far beyond simple data storage. It plays a vital role in function call management, expression evaluation, and backtracking algorithms. It's a fundamental tool for managing program execution flow, and its simplicity makes it easy to reason about and implement. For these reasons, it forms an excellent comparison point against the Dynamic Array, which has different performance trade-offs.

In summary, I chose these three data structures because they are fundamental building blocks of computer science. They are relevant, versatile, and offer a fascinating glimpse into the trade-offs involved in algorithm design. Exploring them in detail provides a solid understanding of the principles behind data structure selection and implementation. Moreover, by comparing them, the strengths and weaknesses of different approaches become clear.

- **Use appropriate parameters to describe the complexity. You do not need to justify the complexity; only to explain each parameters used and provide the formula with appropriate references**

1. Priority Queue (Heap-based - Binary Heap)

- **Parameters:**
 - n : Represents the number of elements currently stored in the priority queue (heap).
- **Time Complexity:**
 - Insertion (Enqueue): $O(\log n)$
 - Deletion (Dequeue - Highest Priority): $O(\log n)$
 - Search (for an arbitrary element): $O(n)$ - Inefficient for heap-based priority queues.
- **Space Complexity:**
 - $O(n)$ - The heap stores all n elements.
- **Explanation:**
 - The logarithmic time complexity ($\log n$) for insertion and deletion arises from the heap property maintenance. After inserting or deleting an element, the heap structure needs to be adjusted to ensure that the parent node always has higher (or lower, for a min-heap) priority than its children. This adjustment typically involves traversing a path from the root to a leaf, or vice versa, which takes logarithmic time in a balanced binary heap.
 - The base of the logarithm is typically 2, reflecting the binary tree structure of the heap.
 - Searching in a heap is generally $O(n)$ because the heap property doesn't provide any ordering guarantee between siblings, making a targeted search impossible.
- **Reference:**
 - Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Specifically, Chapter 6: Heapsort)

2. Dynamic Array (Resizable Array)

- **Parameters:**
 - n : Represents the number of elements currently stored in the dynamic array.
- **Time Complexity:**
 - Access by Index: $O(1)$
 - Insertion at End (Append): $O(1)$ amortized, $O(n)$ worst-case
 - Deletion at End: $O(1)$
 - Insertion/Deletion at Beginning or Middle: $O(n)$
- **Space Complexity:**
 - $O(n)$ - Stores n elements. Can have some overhead due to allocated but unused space.
- **Explanation:**

- $O(1)$ access by index is a key advantage, enabled by the contiguous memory allocation.
- The amortized $O(1)$ insertion at the end deserves explanation. When the array is full, it needs to be resized, which involves allocating a new, larger array and copying all existing elements. This resizing operation takes $O(n)$ time. However, resizing doesn't happen with every insertion. The resizing is infrequent, and the cost is spread out over many insertions, resulting in an average cost of $O(1)$ per insertion. This is what "amortized" means. The worst-case time complexity for append is $O(n)$ when resizing occurs.
- Insertion or deletion at the beginning or middle requires shifting all subsequent elements to make room or close the gap, hence the $O(n)$ complexity.
- **Reference:**
 - Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional. (Specifically, Section 1.4: Analysis of Algorithms)

3. Dynamic Stack (Linked List based)

- **Parameters:**
 - n : Represents the number of elements currently stored in the stack.
- **Time Complexity:**
 - Push (Add to Top): $O(1)$
 - Pop (Remove from Top): $O(1)$
- **Space Complexity:**
 - $O(n)$ - Each element is stored in a node, and there are n nodes.
- **Explanation:**
 - The $O(1)$ time complexity for push and pop is a direct result of using a linked list. Adding or removing a node at the head of a linked list only requires updating a few pointers, which takes constant time. No shifting or resizing is involved.
 - Space complexity is $O(n)$ because each element in the stack requires a separate node in the linked list, consuming memory proportional to the number of elements. Each node stores the data and a pointer to the next node.
- **Reference:**
 - Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Python*. John Wiley & Sons. (Specifically, Chapter 3: Stacks, Queues, and Deques)

- **Python code comparing the data structures selected. You are responsible for generating appropriate data sets for the performance profile. Provide any justification of your choices in the report. Do not hesitate to contact the instructor (a quick email can save you hours for misguided work!). Add the GitHub link of your implementations in the pdf file.**

Link: <https://github.com/MahmoudOsama97/AdvancedDataStructuresRepo>

General Steps to Use the project:

Option1:

1. Clone the Repository: git clone

<https://github.com/MahmoudOsama97/AdvancedDataStructuresRepo>

2. Run the Code: python3 main/main.py

Option2:

Use colab notebook directly by uploading it to google colab or by running it in VS Code
























Note: code is a combination from:

1) Being manually written

2) browsed from github repos and geeksforgeeks site

3) used Gemini in some parts to understand the algorithms and have example code.

Folder Structure:

- ▼  **20.7 MB Dataset**
 -  **7.5 MB random_data.txt**
 -  **6.6 MB nearly_sorted_data.txt**
 -  **6.6 MB sorted_data.txt**
 -  **4.0 KB DatasetGenerator.py**
- >  **8.8 MB .git**
- ▼  **1.2 MB Notebook**
 -  **1.2 MB Assignment2.ipynb**
- >  **908.0 KB [3 Files]**
- ▼  **464.0 KB Plots**
 -  **68.0 KB merged_performance_nearly_sort...**
 -  **64.0 KB merged_performance_random.png**
 -  **64.0 KB merged_performance_sorted.png**
 -  **60.0 KB data_structure_performance_rand...**
 -  **56.0 KB data_structure_performance_nearl...**
 -  **56.0 KB data_structure_performance_sort...**
 -  **32.0 KB dynamic_array_performance_near...**
 -  **32.0 KB dynamic_array_performance_rand...**
 -  **32.0 KB dynamic_array_performance_sort...**
- ▼  **12.0 KB Main**
 -  **12.0 KB Main.py**
- ▼  **12.0 KB UnitTesting**
 -  **12.0 KB UnitTesting.py**