Assignment No: 3
Date: 27/03/2024

ADVANCED DATA STRUCTURES

ASSIGNMENT - 3

COMPARISON OF INSERTION AND DELETION TIMES IN SORTED SEQUENCE, UNSORTED SEQUENCE AND HEAP BASED PRIORITY QUEUES

Assignment Type of Submission:			
Group **COMP47500 Advanced Data Structures** **Group Assignment- 3**	Yes/No Yes	**List all group members' details:** Student Name: Avantika Sivakumar Student ID: 23205789 Student Name: Ria Rahul Asgekar Student ID: 23203987 Student Name: Jeevan Raju Ravi Govind Raju Student ID: 23207314	**% Contribution Assignment Workload** 33.33% (Code implementation, Result evaluation, video explanation) 33.33% (Code Implementation, material gathered for report, video explanation) 33.33% (Code implementation, material gathered for report, video explanation)

1. **Problem Domain Description:**

This report aims to compare the efficiency of insertion and deletion operations across three distinct data structures: sorted sequences, unsorted sequences, and priority queues implemented using the heap data structure. The analysis focuses on evaluating the time complexities of insertion and deletion operations to understand the performance characteristics of each data structure.

1. Sorted Sequences:

1.1. Overview:

Sorted sequences maintain elements in ascending or descending order, ensuring that elements are arranged based on their values. Insertion and deletion operations in sorted sequences involve maintaining this sorted order, which impacts their time complexities.

1.2. Insertion Time:

Inserting an element into a sorted sequence requires finding the correct position based on its value and shifting existing elements to accommodate the new element while preserving the sorted order. The time complexity of insertion in a sorted sequence is O(n) in the worst case, where n is the number of elements in the sequence.

1.3. Deletion Time:

Deleting an element from a sorted sequence involves locating the element and removing it from the sequence. Since the sequence is sorted, binary search can be employed to find the element, resulting in a time complexity of O(log n). However, removing the element and shifting subsequent elements may take O(n) time in the worst case, resulting in a total time complexity of O(n).

2. Unsorted Sequences:

2.1. Overview:

Unsorted sequences do not maintain any specific order among elements, allowing elements to be stored without consideration of their values. Insertion and deletion operations in unsorted sequences are typically simpler compared to sorted sequences.

2.2. Insertion Time:

Inserting an element into an unsorted sequence involves appending the element to the end of the sequence, resulting in constant-time insertion with a time complexity of O(1).

2.3. Deletion Time:

Deleting an element from an unsorted sequence requires locating the element and removing it. Since the sequence is unsorted, linear search may be necessary, resulting in a time complexity of O(n) in the worst case.

3. Priority (Heap) Queues:

3.1. Overview:

Priority queues implemented using the heap data structure maintain elements based on their priority, ensuring that higher priority elements are processed first. Heaps, particularly binary heaps, are commonly used to implement priority queues due to their efficient insertion and deletion operations.

3.2. Insertion Time:

Inserting an element into a priority queue implemented using a heap involves adding the element to the end of the heap and then performing heapify operations to maintain the heap property. The time complexity of insertion in a heap-based priority queue is O(log n), where n is the number of elements in the queue.

3.3. Deletion Time:

Deleting an element from a heap-based priority queue entails removing the element from the root of the heap and then restoring the heap property through heapify operations. Similar to insertion, the time complexity of deletion in a heap-based priority queue is O(log n).

In conclusion, the efficiency of insertion and deletion operations varies across sorted sequences, unsorted sequences, and priority queues implemented using heaps. Sorted sequences offer efficient deletion but slower insertion times due to maintaining sorted order. Unsorted sequences provide constant-time insertion but have slower deletion times, especially for large sequences. Priority queues implemented using heaps offer a balanced approach with logarithmic time complexity for both insertion and deletion operations, making them suitable for managing prioritised elements efficiently. The choice of data structure depends on the specific requirements of the application and the trade-offs between insertion and deletion efficiency.

2. Theoretical Foundations of the Data Structure(s) utilised

2.1. Sorted Sequences:

Sorted sequences are based on the fundamental concept of maintaining elements in a specific order, typically either ascending or descending based on their values. The primary theoretical foundation of sorted sequences lies in algorithms and data structures related to sorting, searching, and maintaining order.

- Sorting Algorithms:
  Sorted sequences rely on various sorting algorithms such as merge sort, quicksort, or insertion sort to arrange elements in the desired order efficiently. These algorithms

leverage principles of comparison and partitioning to sort elements in ascending or descending order, providing a foundation for maintaining sorted sequences.

- Search Algorithms:
  Search algorithms such as binary search play a crucial role in sorted sequences by enabling efficient retrieval of elements based on their values. Binary search, in particular, relies on the sorted nature of the sequence to quickly locate elements by repeatedly dividing the search interval in half, demonstrating the theoretical synergy between sorting and searching.

- Order Maintenance:
  Theoretical concepts related to maintaining order, such as insertion and deletion techniques while preserving sortedness, are fundamental to sorted sequences. Algorithms for inserting elements into sorted sequences while preserving order, such as binary insertion or maintaining balance in self-adjusting data structures, contribute to the theoretical foundation of sorted sequences.

2.2. Unsorted Sequences:

Unsorted sequences deviate from the constraints of maintaining order, focusing on simplicity and flexibility in element arrangement. The theoretical foundation of unsorted sequences is rooted in data structures and algorithms that prioritise ease of insertion and removal over maintaining a specific order.

- Linear Data Structures:
  Unsorted sequences often utilise linear data structures such as arrays or linked lists, emphasising sequential storage of elements without enforcing any particular order. The theoretical foundation of unsorted sequences draws from concepts of dynamic memory allocation, pointer manipulation, and efficient traversal of linear structures.

- Insertion and Deletion Efficiency:
  Theoretical considerations for unsorted sequences revolve around optimising insertion and deletion operations. Algorithms and data structures that enable constant-time insertion, such as dynamic arrays or resizable arrays, contribute to the theoretical underpinnings of unsorted sequences.

2.3. Priority (Heap) Queues:

Priority queues implemented using the heap data structure combine the theoretical principles of both sorting and efficient element retrieval based on priority. The theoretical foundation of heap-based priority queues encompasses concepts from heap data structures, priority queues, and binary trees.

- Heap Data Structure:
  Theoretical foundations of priority queues implemented using heaps are grounded in heap data structures, particularly binary heaps. Binary heaps leverage the properties

of complete binary trees and heap order to enable efficient insertion, deletion, and retrieval of elements with logarithmic time complexity.

- Priority Queue Abstraction:
  Priority queues abstract the notion of prioritised elements, allowing higher priority elements to be processed before lower priority ones. Theoretical considerations for priority queues revolve around defining and maintaining priority order, ensuring that elements are processed according to their assigned priorities.

- Binary Trees:
  Theoretical concepts from binary trees, such as parent-child relationships and tree traversal algorithms, contribute to the foundation of heap-based priority queues. Binary tree properties, such as complete binary trees or balanced binary trees, influence the efficiency and performance of priority queues implemented using heaps.

In summary, the theoretical foundations of sorted sequences, unsorted sequences, and priority queues implemented using heaps encompass a diverse range of concepts from sorting algorithms, linear data structures, priority queue abstractions, and binary tree properties. Understanding these theoretical principles is essential for designing, implementing, and analysing the performance of these data structures in various applications.

3. Applications:

3.1. Sorted Sequences:

- Database Indexing:
  Sorted sequences find extensive use in database indexing, where data is organised in sorted order based on indexed columns. This facilitates efficient searching, retrieval, and sorting of data, improving the performance of database queries.

- Symbol Tables:
  In compilers and interpreters, sorted sequences are utilised in symbol tables to store identifiers such as variables, functions, and keywords. Organising symbols in sorted order enables quick lookup and resolution during compilation or interpretation processes.

- Financial Data Analysis:
  Sorted sequences are employed in financial data analysis applications to organise time-series data, such as stock prices or economic indicators. Sorting data based on timestamps allows for efficient analysis, trend identification, and statistical computations.

3.2. Unsorted Sequences:

- Task Queuing in Web Servers:
  Unsorted sequences are commonly used in web servers to manage incoming HTTP requests in the order they are received. Tasks or requests are appended to the end of the queue and processed sequentially, ensuring fair handling of client requests.

- Buffering in Network Communication:
  Unsorted sequences serve as buffers in networking equipment, such as routers and switches, to temporarily store data packets before forwarding them to their destination. This helps regulate data flow and prevent packet loss during network congestion.

- Background Job Processing:
  Many software applications use unsorted sequences to manage background tasks or asynchronous operations. Background tasks, such as data processing or file uploads, are added to the queue and processed sequentially, allowing the main application thread to remain responsive.

3.3. Priority (Heap) Queues:

- Task Scheduling in Operating Systems:
  Priority queues implemented using heaps are integral to operating system schedulers for managing processes or threads. Tasks with higher priority levels, such as system-critical processes, are scheduled for execution before lower priority tasks, ensuring efficient resource utilisation.

- Network Traffic Management:
  Priority queues are utilised in network routers and switches to prioritise traffic based on Quality of Service (QoS) requirements. Time-sensitive data, such as voice or video streams, are given higher priority to ensure low latency and reliable transmission.

- Real-Time Systems:
  In industries like aerospace and automotive, priority queues are used in real-time systems for task scheduling and event handling. Critical tasks and events are assigned higher priorities to meet stringent timing constraints and ensure system reliability.

In summary, sorted sequences, unsorted sequences, and priority queues implemented using heaps find diverse applications across various industries, ranging from database management and web servers to network communication and real-time systems. Each type of data structure offers specific advantages and is tailored to meet the unique requirements of different domains, contributing to the efficient operation of systems in the industry.

4. Time Complexity:

4.1. Sorted Sequences:

4.1.1. Insertion: O(n)

- Inserting an element into a sorted sequence requires finding the correct position to maintain the sorted order. This typically involves traversing the sequence linearly to locate the insertion point, resulting in a time complexity proportional to the number of elements already in the sequence.

4.1.2. Deletion: O(n)

- Deleting an element from a sorted sequence also requires locating the element within the sequence, which can be done efficiently using binary search with a time complexity of O(log n). However, removing the element and shifting subsequent elements to fill the gap may take O(n) time in the worst case.

4.2. Unsorted Sequences:

4.2.1. Insertion: O(1)

- Inserting an element into an unsorted sequence involves appending the element to the end of the sequence, resulting in constant-time insertion regardless of the number of elements already in the sequence.

4.2.2. Deletion: O(n)

- Deleting an element from an unsorted sequence may require traversing the entire sequence to locate the element to be deleted. Therefore, the time complexity of deletion is O(n) in the worst case.

4.3. Priority (Heap) Queues:

4.3.1. Insertion: O(log n)

- Inserting an element into a priority queue implemented using a heap involves adding the element to the end of the heap and then performing heapify operations to maintain the heap property. The time complexity of insertion in a heap-based priority queue is logarithmic, as it depends on the height of the heap tree, which is O(log n) for a heap with n elements.

4.3.2. Deletion: O(log n)

- Deleting an element from a priority queue implemented using a heap entails removing the element from the root of the heap and then restoring the heap property through heapify operations. Similar to insertion, the time complexity of deletion in a heap-based priority queue is logarithmic, with a worst-case complexity of O(log n).

In summary, the time complexity of operations varies across different data structures. Sorted sequences exhibit linear time complexity for both insertion and deletion due to the need for maintaining sorted order. Unsorted sequences offer constant-time insertion but linear-time

deletion. Priority queues implemented using heaps provide logarithmic time complexity for both insertion and deletion, making them suitable for efficient priority-based operations.

5. Advantages:

5.1. Sorted Sequences:

5.1.1. Efficient Searching:
Sorted sequences facilitate efficient searching using algorithms like binary search, which has a time complexity of O(log n). This makes searching for elements significantly faster compared to unsorted sequences.

5.1.2. Predictable Order:
Elements in sorted sequences are arranged in a predictable order, making it easier to reason about and predict the position of elements relative to each other.

5.1.3. Support for Range Queries:
Sorted sequences allow for efficient range queries, such as finding all elements within a specified range. Algorithms like binary search can be adapted to efficiently locate the boundaries of the range.

5.2. Unsorted Sequences:

5.2.1. Constant-Time Insertion:
Unsorted sequences offer constant-time insertion, as elements are simply appended to the end of the sequence. This makes them suitable for scenarios where rapid insertion is a priority.

5.2.2. Simple Implementation:
Unsorted sequences are straightforward to implement and manage, requiring minimal overhead for maintaining order. This simplicity can lead to better performance in scenarios with frequent insertions.

5.2.3. Preservation of Insertion Order:
Unsorted sequences preserve the order in which elements are inserted, making them suitable for applications where maintaining the insertion order is important.

5.3. Priority (Heap) Queues:

5.3.1. Efficient Priority-Based Operations:
Priority queues implemented using heaps excel at priority-based operations, such as retrieving the highest (or lowest) priority element efficiently. This makes them ideal for scenarios where elements need to be processed based on their priority levels.

5.3.2. Logarithmic Time Complexity:

Both insertion and deletion operations in heap-based priority queues have a logarithmic time complexity of O(log n). This ensures efficient performance even for large datasets.

5.3.3. Dynamic Operations:
Priority queues can dynamically adjust to changes in priority levels, allowing for efficient insertion and deletion of elements while maintaining the heap property. This adaptability makes them suitable for dynamic environments.

In summary, sorted sequences offer efficient searching and support for range queries, unsorted sequences provide constant-time insertion and simplicity in implementation, and priority queues implemented using heaps excel at priority-based operations with logarithmic time complexity. The choice of data structure depends on the specific requirements of the application, balancing factors such as search efficiency, insertion speed, and priority-based processing.

6. History:

6.1. Sorted Sequences:

- Early Computing: Sorted sequences have been utilised since the early days of computing for tasks such as searching and sorting. Algorithms for sorting, like bubble sort and insertion sort, were among the first to be developed and implemented on early computers.

- Algorithmic Developments: Over time, more efficient sorting algorithms were developed, such as quicksort and mergesort, which significantly improved the efficiency of sorting operations on sorted sequences.

- Database Systems: In the development of database systems, sorted sequences played a crucial role in organising and indexing data for efficient retrieval. B-tree and B+-tree data structures were introduced to maintain sorted order efficiently, leading to the development of modern database systems.

6.2. Unsorted Sequences:

- Early Data Structures: Unsorted sequences have been used in various forms since the early days of programming. Simple arrays and lists were often used to store collections of elements without any specific order.

- Dynamic Data Structures: With the advent of dynamic data structures like linked lists, unsorted sequences gained popularity for their simplicity and flexibility. Linked lists allowed for constant-time insertion and deletion operations, making them suitable for dynamic applications.

- Algorithmic Analysis: The study of algorithmic complexity led to a deeper understanding of the performance characteristics of unsorted sequences. Algorithms like linear search and traversal operations became fundamental in analysing the efficiency of operations on unsorted sequences.

6.3. Priority (Heap) Queues:

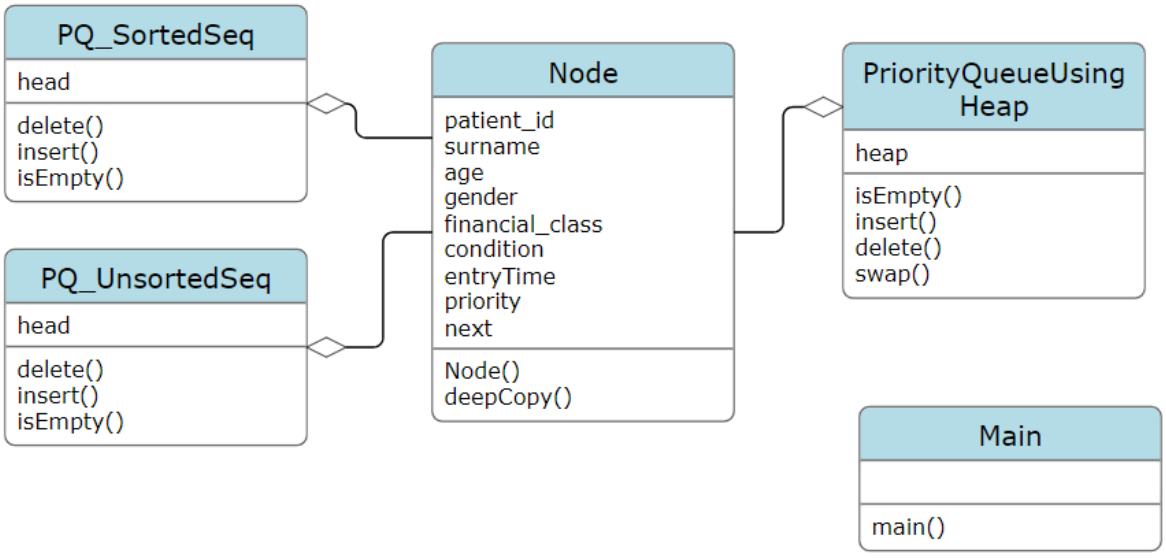
- Heap Data Structure: The heap data structure, essential for implementing priority queues, has a rich history dating back to the 1960s. Heaps were initially used in the implementation of priority queues and heap sort algorithms.

- Priority Queue Applications: Priority queues found applications in various fields, including operating systems, task scheduling, and network protocols. Early implementations focused on efficient priority-based processing in scenarios where tasks or events needed to be prioritised.

- Algorithmic Optimizations: Over time, optimizations were made to priority queue implementations, leading to efficient data structures like binary heaps and Fibonacci heaps. These optimizations improved the performance of priority queue operations in practical applications.

In summary, the history of sorted sequences, unsorted sequences, and priority queues is intertwined with the evolution of computing and algorithmic developments. From early sorting algorithms to modern database systems, these data structures have played vital roles in organising, processing, and managing data in various applications throughout history.

7. Analysis/Design (UML Diagram(s))



PQ_SortedSeq: This class implements a priority queue using sorted sequence. It has a class variable head of type Node. Its class methods are delete(), insert() and isEmpty().

PQ_UnsortedSeq: This class implements a priority queue using an unsorted sequence. It has a class variable head. Its class methods are delete(), insert() and isEmpty().

PriorityQueueUsingHeap: This class implements a priority queue using a heap data structure. It has a class variable heap. Its class methods are delete(), insert(), swap() and isEmpty().

Relationships:
The 3 priority queue classes have a 'has a' relationship(aggregation) with the Node class, as the Node class objects are used as the fundamental unit to make the queues. The main class has an instance of each of the queues.

Algorithms:

1. Priority queue using heap:
    a. Insert:
    Add the new node to the heap.
    Compare the priority of the newly added node with its parent.
    If the priority of the newly added node is greater than its parent's priority, swap them and update the index to the parent's index.
    Repeat the comparison and swap process until either the parent's priority becomes greater than or equal to the current node's priority or until the root is reached.
    b. Delete:
    Replace the root node (node with the highest priority) with the last node in the heap.
    Remove the last node from the heap.
    Reorganise the heap by comparing the priority of the current node with its children.
    If the priority of the current node is less than any of its children, swap it with the larger child and update the index to the larger child's index.
    Repeat the comparison and swap process until either the current node's priority becomes greater than or equal to its children's priorities or until it has no children.

2. Priority queue using sorted sequence:
    a. Insert:
    Traverse the linked list until finding the appropriate position to insert the new node based on its priority.
    If the linked list is empty or the new node has higher priority than the current head, insert the new node as the new head.
    Otherwise, iterate through the list until finding a node with lower priority than the new node.
    Insert the new node after this node, maintaining the order of priorities in the queue.

b. Delete:
   Save the reference to the current head node in a temporary variable.
   Move the head pointer to the next node, effectively removing the current head from the queue.
   Return the temporary variable which holds the removed node.

3. Priority queue using unsorted sequence:
   a. Insert:
      Check if the queue is empty. If it is, set the head to the new node.
      Traverse the queue until reaching the last node.
      Set the next reference of the last node to point to the new node, effectively inserting the new node at the end of the queue.
   b. Delete:
      Check if the queue is empty. If it is, return null to indicate an empty queue.
      Initialize two pointers, prev and current, both starting at the head of the queue.
      Initialise two more pointers, minPrev and minCurrent, both initially pointing to the head as well. These pointers will track the node with the lowest priority.
      Iterate through the queue, comparing the priorities of nodes. If a node with a lower priority is found, update minPrev and minCurrent accordingly.
      After the iteration, remove the node with the lowest priority from the queue:
      If the node with the lowest priority is the head, update the head to point to the next node.
      Otherwise, adjust the next reference of the previous node to skip over the node with the lowest priority.
      Return the removed node.

8. Time and Space Complexities:

8.1. Sorted Sequences:

8.1.1. Time Complexity:
- Insertion: O(n)
- Deletion: O(n)

8.1.2. Space Complexity:
- O(n) to store n elements

8.2. Unsorted Sequences:

8.2.1. Time Complexity:
- Insertion: O(1)
- Deletion: O(n)

8.2.2. Space Complexity:

- O(n) to store n elements

8.3. Priority (Heap) Queues:

8.3.1. Time Complexity:
- Insertion: O(log n)
- Deletion: O(log n)

8.3.2. Space Complexity:
- O(n) to store n elements

In summary, sorted sequences offer efficient searching with O(log n) time complexity but slower insertion and deletion operations with O(n) time complexity. Unsorted sequences provide constant-time insertion but have linear-time complexity for deletion. Priority queues implemented using heaps offer efficient insertion and deletion operations with O(log n) time complexity, making them suitable for priority-based processing. However, they require O(n) space to store elements.

9. IMPLEMENTATION:

9.1. Sorted Sequences:

9.1.1. Data Structure:
- Arrays or linked lists can be used to implement sorted sequences.
- Arrays offer efficient random access but may require shifting elements during insertion and deletion.
- Linked lists provide constant-time insertion and deletion at the cost of slower searching.

9.1.2. Insertion:
- To insert an element into a sorted sequence, binary search can be used to find the correct position, followed by shifting elements to accommodate the new element.
- If using an array, resizing may be necessary if the array is full.

9.1.3. Deletion:
- Deleting an element from a sorted sequence involves finding the element and then removing it from the sequence.
- After removal, elements may need to be shifted to fill the gap.

9.2. Unsorted Sequences:

9.2.1. Data Structure:
- Arrays or linked lists are commonly used to implement unsorted sequences.
- Arrays offer efficient random access but may require resizing if full.
- Linked lists provide constant-time insertion and deletion but slower searching.

9.2.2. Insertion:

- Inserting an element into an unsorted sequence involves simply appending the element to the end of the sequence.
- If using an array, resizing may be necessary if the array is full.

9.2.3. Deletion:

- Deleting an element from an unsorted sequence requires finding the element and then removing it from the sequence.
- Elements after the deleted element may need to be shifted to fill the gap.

9.3. Priority (Heap) Queues:

9.3.1. Data Structure:

- Priority queues are commonly implemented using binary heaps or other heap variants.
- Binary heaps are complete binary trees where every parent node has a priority higher than or equal to its children.

9.3.2. Insertion:

- To insert an element into a priority queue, the new element is added to the end of the heap and then "bubbled up" or "percolated up" to maintain the heap property.
- This involves swapping the element with its parent until the heap property is restored.

9.3.3. Deletion:

- Deleting the highest priority element from a priority queue involves removing the root element (which is the highest priority) and then restoring the heap property by "bubbling down" or "percolating down" the new root.
- This process entails swapping the root with one of its children until the heap property is restored.

9.3.4. Space Complexity:

- Priority queues implemented using heaps have a space complexity of O(n) to store n elements in the heap.

In summary, implementations of sorted sequences, unsorted sequences, and priority queues differ in their underlying data structures and operations. Sorted sequences require maintaining order during insertion and deletion, while unsorted sequences offer simpler insertion but slower searching. Priority queues, implemented using heaps, provide efficient insertion and deletion operations suitable for priority-based processing. The choice of implementation depends on the specific requirements of the application, balancing factors such as time complexity, space complexity, and ease of implementation.

10. RUN:

```
Console ×
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe  (23 Mar 2024, 21:37:42 – 21:37:43) [pid: 4360]
Average insertion time for priority queue using heap: 784.7 nanoseconds
Average insertion time for sorted sequence priority queue: 1199.1 nanoseconds
Average insertion time for unsorted sequence priority queue: 2698.6 nanoseconds
Average deletion time for priority queue using heap: 2362.6 nanoseconds
Average deletion time for sorted sequence priority queue: 149.3 nanoseconds
Average deletion time for unsorted sequence priority queue: 3717.5 nanoseconds
```

Dataset:

In order to run the experiments, we used a dataset containing patient records.

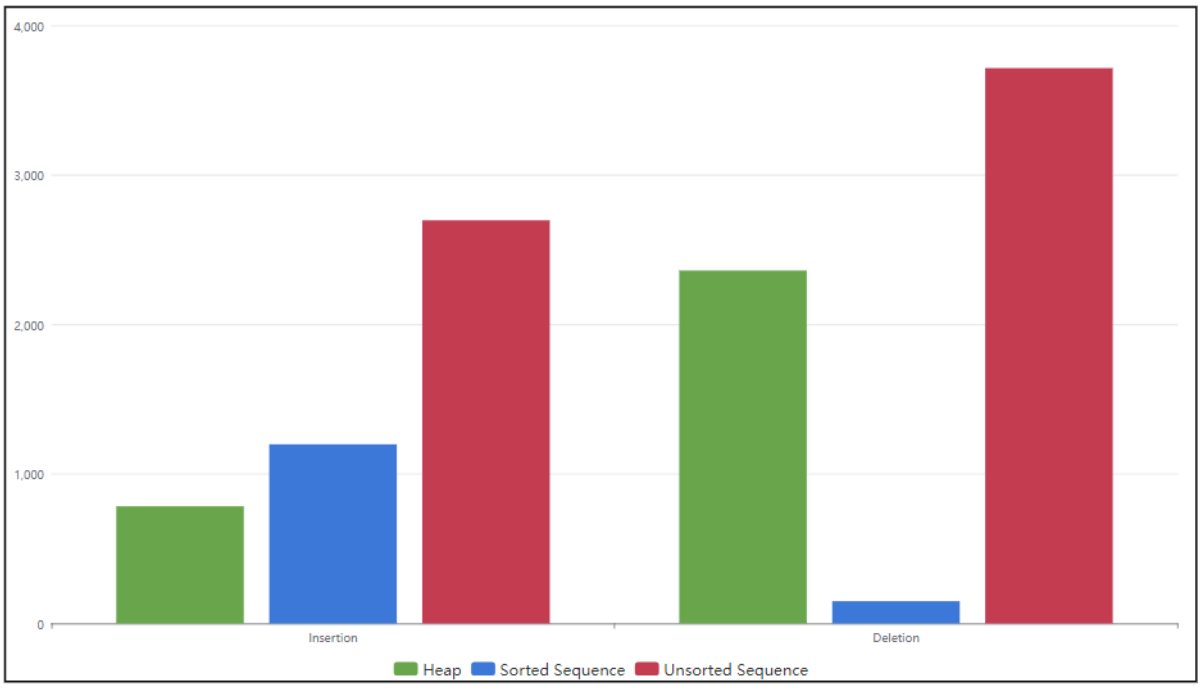
https://www.kaggle.com/datasets/jennahchen/mock-up-patient-data-for-priority-queue-practice

From this dataset, we used the name, age, patient ID, medical condition, financial condition, gender, and entry time to insert and remove patients from the queues. The program assigns priority to each patient based on their condition and adds to/removes from the queue accordingly.

Dataset size: 1000 rows and varied data types(int, string, datetime).

Results:

The records from the dataset are inserted into each of the queues and removed from each of the queues. The average time taken for each is plotted below.



For insertion, the heap data structure takes the least time on average. The sorted sequence data structure takes a longer time because it has to traverse through the list comparing each item till it finds the right position to insert the patient record. The unsorted sequence takes

the longest time on average because it always has to traverse till the end of the queue to add the element. If the unsorted sequence was implemented using a double ended queue, it would take the least time.
Deletion is very quick in the sorted sequence as it just has to remove the very first element without comparing its priority to any others. The heap data structure takes longer as it needs to adjust the heap accordingly (heapify function) to maintain the properties of the heap. The unsorted sequence has the longest deletion times as it has to go through the entire queue each time to find which element is of the highest priority.

Code Implementation GitHub (link):
https://github.com/RiaAsgekar/COMP47500_Assignment3

Set of Experiments run and results:
https://github.com/RiaAsgekar/COMP47500_Assignment3/blob/main/assignment3/Main.java

Video of the Implementation running
Zoom (link & password):
https://ucd-ie.zoom.us/rec/share/9e8v_l2uOyacQHLeezAoQdl9Bkk-rkBffeKc8llTrPcF5yCxXxhDz8XuvbGvsd4.z23g-7PuEJsF4IeV?startTime=1711575661000

Passcode: u7Y5y=#k

Comments:
Ria gives an overview of our chosen problem domain, explains priority queues, principle and applications involved, program structure. Jeevan explains about the sorted sequence and heap based priority queues implementations. Avantika explains about the unsorted sequence implementation , the Main class and talks about the results.

References:

1. https://www.baeldung.com/java-merge-sorted-sequences
2. https://www.csl.mtu.edu/cs2321/www/newLectures/03_Sequences_And_Iterators.html
3. https://stackoverflow.com/questions/62029063/ordered-or-unordered-sequence-problem-in-java
4. https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#:~:text=An%20unbounded%20priority%20queue%20based,does%20not%20permit%20null%20elements.
5. https://www.geeksforgeeks.org/priority-queue-class-in-java/
6. https://www.geeksforgeeks.org/how-to-implement-priority-queue-using-heap-or-array/
7. https://stackoverflow.com/questions/18144820/inserting-into-sorted-linkedlist-java
8. https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap6.pdf

9. https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/
10. https://www.mcobject.com/docs/Content/Users_Guides/Core/Using_Sequences/Using_Sequences_Java.htm#:~:text=The%20Java%20Sequence%20class%20is,equivalent%20role%20in%20Java%20applications