COMP47500 – Advanced Data Structures in Java

Assignment No: 1

Date: 26/02/24

Title: Leveraging Queue and Dequeue to handle bulk orders in an Order Management System

Code Link : https://github.com/COMP47500-Adv-Data-Structures-in-Java/Assignment-1

Execute: OrderManagementSystem.java inside ie/ucd/csnl/comp47500

Assignment Type of Submission:			
Group	Yes/No	**List all of group members' details:**	**% Contribution Assignment Workload**
	Yes	Student Name : Prakash Jha Student ID : 2320077	25%
		Student Name: Satish Gaikwad Student ID: 23204142	25%
		Student Name: Chungman Lee Student ID: 23205535	25%
		Student Name: Abhijith Sathyendran Student ID: 23200124	25%

1. Problem Domain Description:

An Order Management System (OMS) serves as a critical tool for businesses operating in sales, e-commerce, or retail sectors. Its primary function is to facilitate the smooth handling of orders from inception to fulfilment, ensuring prompt processing and customer contentment. Implementing an OMS using queue and dequeue data structures in Java presents a structured methodology to efficiently manage orders.

The task at hand involves conceptualising and constructing an Order Management System (OMS) in Java, leveraging **queue and dequeue** data structures. This system should simplify the creation, modification, and processing of orders, thereby streamlining order management and fulfilment processes. It encompasses features for order tracking, handling orders at millions, processing of orders based on their types(priority vs normal orders).

Key Features:

- Order Queue Management:
  Utilisation of queue data structure to oversee the order queue, guaranteeing First-In-First-Out (FIFO) order processing. Implementation of enqueue and dequeue operations to seamlessly add new orders to the queue and process them sequentially.

- Order Creation and Modification:
  Provision for users to generate new orders by furnishing pertinent order details like customer information, product quantities, and shipping address. Support for order modification functionalities, allowing users to amend order specifics or annul orders before they undergo processing.

- Order Processing Workflow:
  Establishment of a structured workflow for order processing encompassing order validation, inventory assessment, and fulfilment. Incorporation of logic to transition orders through various processing stages, ensuring prompt fulfilment and delivery.

- Reporting and Analytics:
  Generation of reports and analytics on various metrics such as order volumes, order processing durations, inventory turnover, and customer satisfaction. Provision of insights to fine-tune operations, pinpoint bottlenecks, and enhance overall efficiency in order processing.

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

Queues and deques are ubiquitous data structures in computer science, employed in diverse applications from task scheduling to network communication. Their fundamental operations and theoretical characteristics are crucial for efficient implementation and utilisation.

Queue

A queue adheres to the **First-In-First-Out (FIFO)** principle, an ADT where elements are enqueued at the rear and dequeued from the front. This linear ordering behaviour is analogous to a waiting line, where the first individual in line receives service first. Queues can be implemented using various data structures - arrays, linkedLists etc. - each offering distinct advantages and trade-offs.

Arrays: Simple and efficient for enqueue operations, but dequeuing can become expensive as elements need to be shifted down the array to maintain order. Additionally, array size needs to be predefined, limiting flexibility for dynamic data sets.

Linked Lists: Offer dynamic memory allocation, making them suitable for queues of varying sizes. Both enqueue and dequeue operations can be performed efficiently, with constant time complexity (O(1)) in most cases. However, random access within the queue is not as efficient compared to arrays.

The standard operations available in queue are:

- **add(E e)**: This method inserts the specified element into the queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to offer(E e), which can fail to insert an element only by throwing an exception.
- **offer(E e)**: This method inserts the specified element into the queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to add(E e), which can fail to insert an element only by throwing an exception.
- **remove()**: This method retrieves and removes the head of the queue. This method throws an exception if the queue is empty.
- **poll()**: This method retrieves and removes the head of the queue, or returns null if the queue is empty.
- **element(**): This method retrieves, but does not remove, the head of the queue. This method throws an exception if the queue is empty.
- **peek()**: This method retrieves, but does not remove, the head of the queue, or returns null if the queue is empty.
- **iterator()**: This method returns an iterator over the elements in the queue. The elements are returned in no particular order (unless this queue is an instance of some class that provides a guarantee). The time complexity of this operation is O(1) for obtaining the iterator, but O(n) for iterating over the entire queue.
- **isEmpty()**: This method returns true if the queue contains no elements.

- **contains(Object o)**: This method returns true if the queue contains the specified element. More formally, returns true if and only if the queue contains at least one element e such that o.equals(e).

Time complexity table for Queue operations:

add(E e)	O(1)
offer(E e)	O(1)
remove()	O(1)
poll()	O(1)
element()	O(1)
peek()	O(1)
iterator()	O(n)
isEmpty()	O(1)
contains(Object o)	O(n)

Dequeue

A dequeue is a more versatile structure, extending the queue by **allowing element insertion and removal from both the front and rear**. Deques are used when we need to insert and delete elements from both ends, such as in a palindrome checker where we need to compare the front and rear elements together. This flexibility broadens its applicability compared to the unidirectional nature of queues. Similar to queues, deques can leverage various data structures for implementation, with their inherent flexibility leading to a wider range of applications beyond traditional queue usage.

The standard operations available in dequeue are:

- **addFirst(E element)**: This method adds the specified element to the beginning of the deque. It creates a new node with the given element and inserts it as the new first node. If the deque is empty, the new node becomes both the first and last node. The size of the deque is incremented by 1.
- **addLast(E element)**: This method adds the specified element to the end of the deque. It creates a new node with the given element and inserts it as the new last node. If the deque is empty, the new node becomes both the first and last node. The size of the deque is incremented by 1.
- **removeFirst()**: This method removes and returns the element at the beginning of the deque. It updates the first node to the next node in the deque and sets the previous node of the new first node to null. If the deque becomes empty after removing the element, both first and last nodes are set to null. The size of the deque is decremented by 1. If the deque is already empty, a NoSuchElementException is thrown.
- **removeLast()**: This method removes and returns the element at the end of the deque. It updates the last node to the previous node in the deque and sets the next node of the new last node to null. If the deque becomes empty after removing the element, both first and last nodes are set to null. The size of the deque is decremented by 1. If the deque is already empty, a NoSuchElementException is thrown.
- **getFirst()**: This method returns the element at the beginning of the deque without removing it. If the deque is empty, a NoSuchElementException is thrown.
- **getLast()**: This method returns the element at the end of the deque without removing it. If the deque is empty, a NoSuchElementException is thrown.
- **size()**: This method returns the current size of the deque.
- isEmpty(): This method checks if the deque is empty. It returns true if the deque is empty, otherwise false.
- **contains(Object obj)**: This method checks if the deque contains the specified object. It iterates through the deque and compares each element with the specified object using the equals() method. It returns true if the object is found, otherwise false.
- **iterator()**: This method returns an iterator over the elements in the deque. It allows iterating through the elements in the order they were added.
- **peek()**: This method retrieves, but does not remove, the head (first element) of the queue represented by this deque (double-ended queue), or returns null if this deque is empty. It's equivalent to peekFirst().
- **peekFirst()**: This method retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty.
- **peekLast():** This method retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.
- **remove()**: This method retrieves and removes the head (first element) of the queue represented by this deque. If the deque is empty, it throws a NoSuchElementException. This method is equivalent to removeFirst().
- **poll()**: This method retrieves and removes the head (first element) of the queue represented by this deque, or returns null if this deque is empty. This method is equivalent to pollFirst().

Time complexity table for dequeue operations

addFirst(E element)	O(1)
addLast(E element)	O(1)
removeFirst()	O(1)
removeLast()	O(1)
getFirst()	O(1)
getLast()	O(1)
size()	O(1)
isEmpty()	O(1)
contains(Object obj)	O(n)
iterator()	O(n)
peek()	O(1)
peekFirst()	O(1)
peekLast()	O(1)
remove()	O(1)
poll()	O(1)

In summary, the choice between a queue and a deque depends on the specific requirements of the problem you are trying to solve. If you need to maintain a FIFO order, a queue is the appropriate data structure. If you need to add or remove elements from both ends, a deque is the appropriate data structure.

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

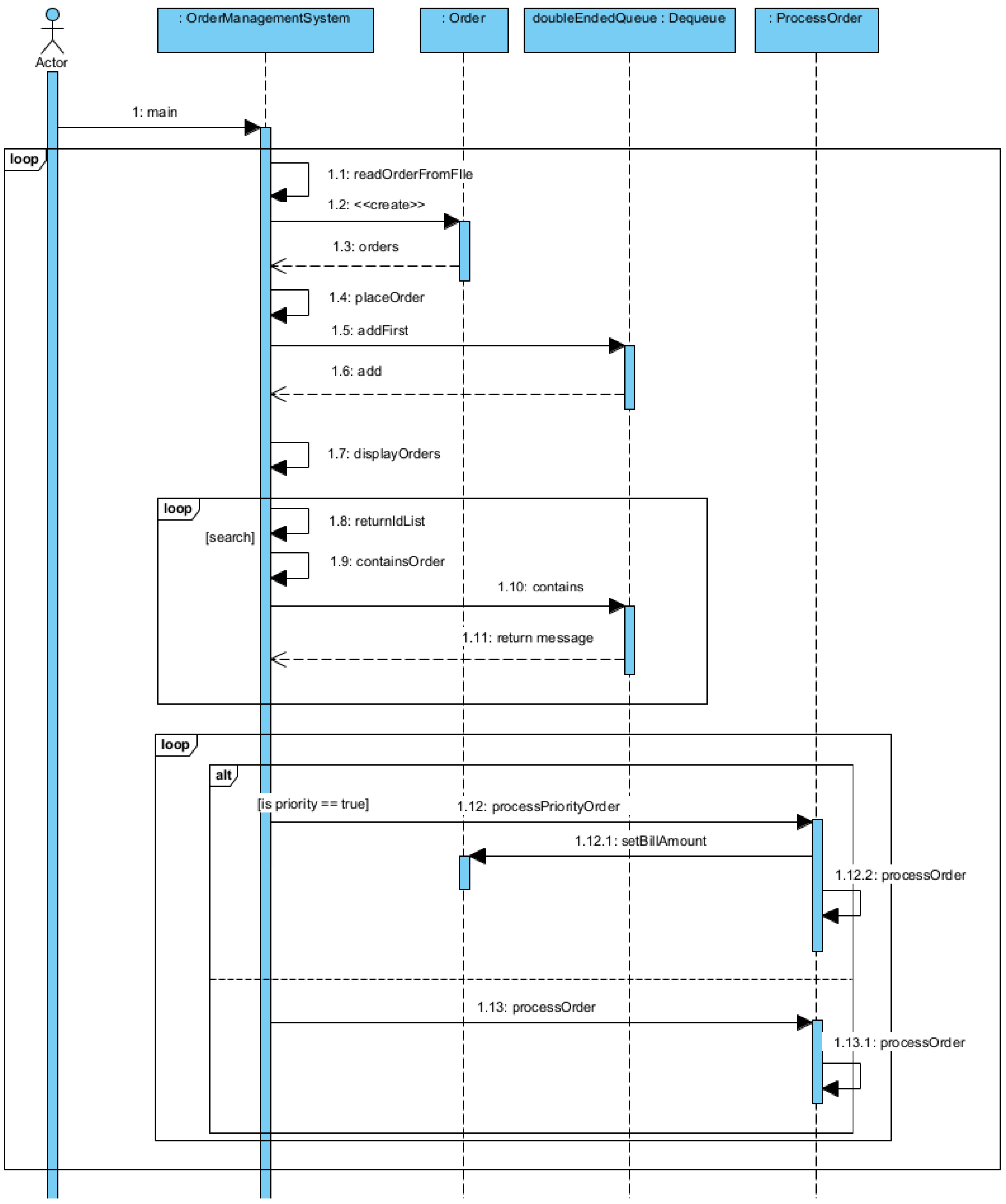
For the solution implemented by the current system there are 2 main objectives. Handle the vast amounts of incoming orders on a first come first serve basis, but with a special case of handling priority orders. A Queue would be ideal for any such waiting list implementations. It offers access to the head of the queue from where you can process orders and a tail of the queue where orders can be appended to.

Queues however can create an overhead when there are priority orders to be executed much faster before the normal orders. An order arriving on priority would be required to be added at the head of the list so that it can be served immediately. The queue has no such capability and hence any priority order would have to be served like a normal order. The workaround would be to create separate queues, one for priority and one for normal orders. This would come at the cost of memory and hence the overhead.

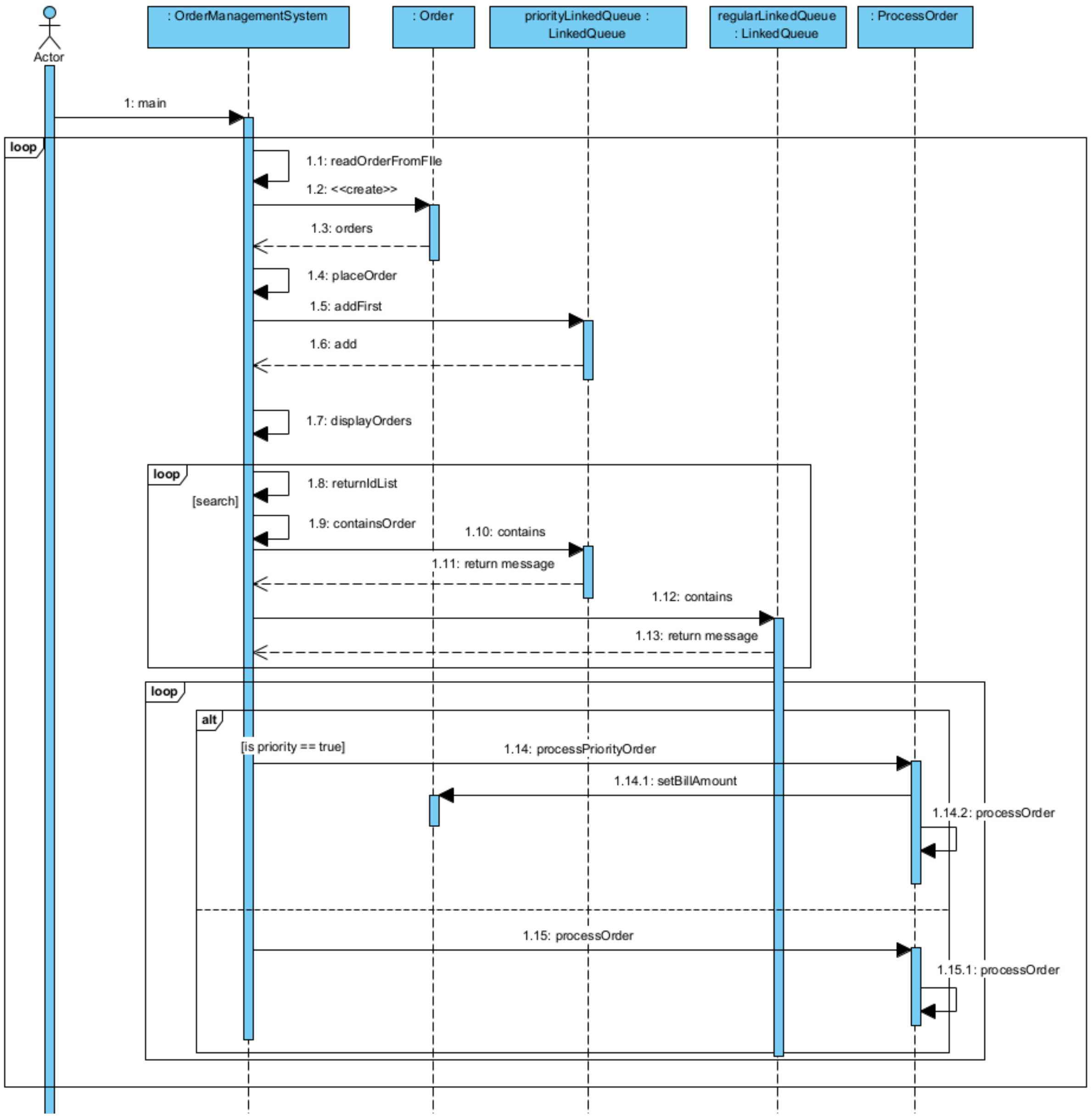
Dequeue offers more flexibility in such situations. It can be used to insert the incoming priority orders at the head where they get processed immediately and hence the time to process a priority order in a dequeue from the vast incoming list would be much smaller compared to one in a single queue.

All orders to be processed are added to the queue and sent to a processing logic which handles the priority orders and normal orders as needed.

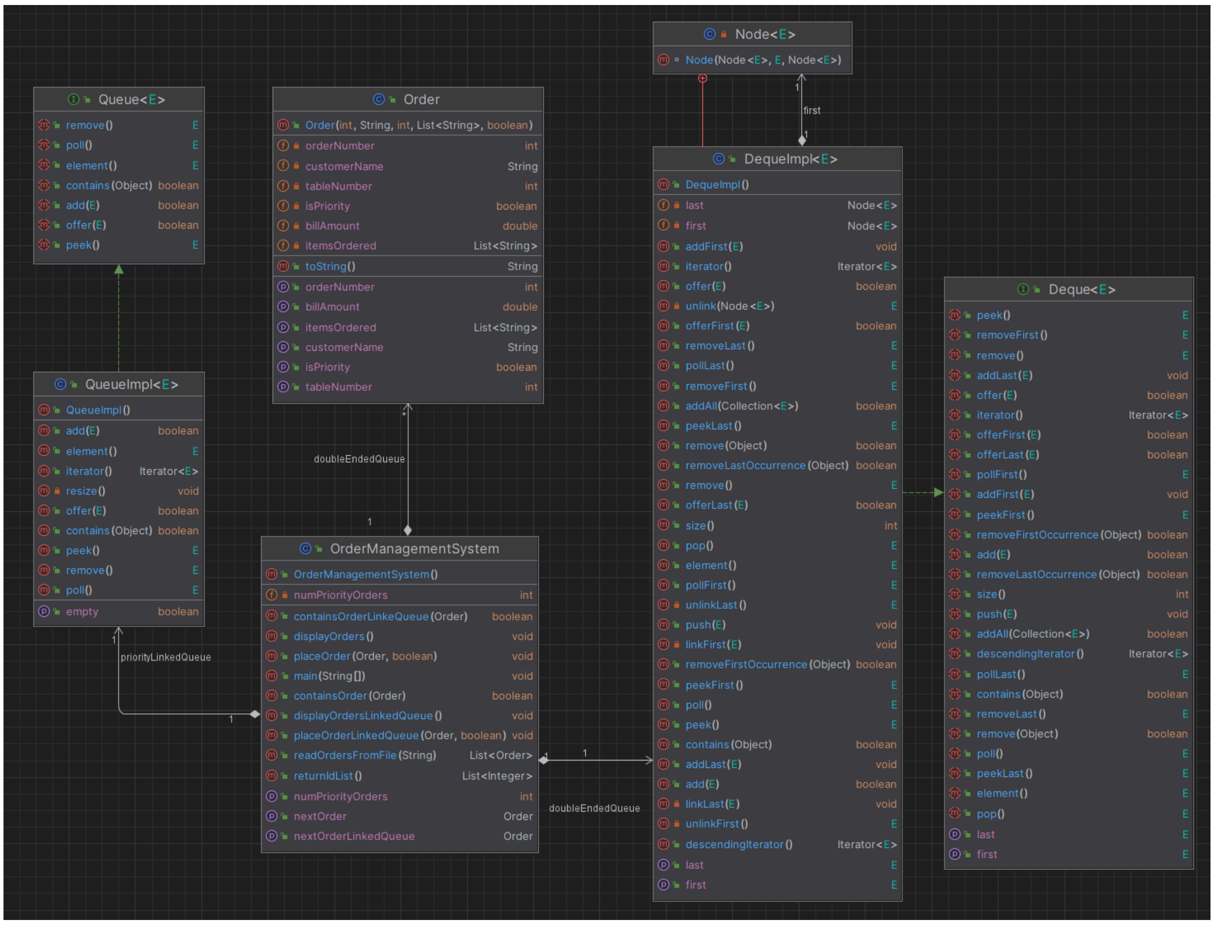
Sequence Diagram(Double Ended Queue)



Sequence Diagram(Linked Queue)



Class Diagram

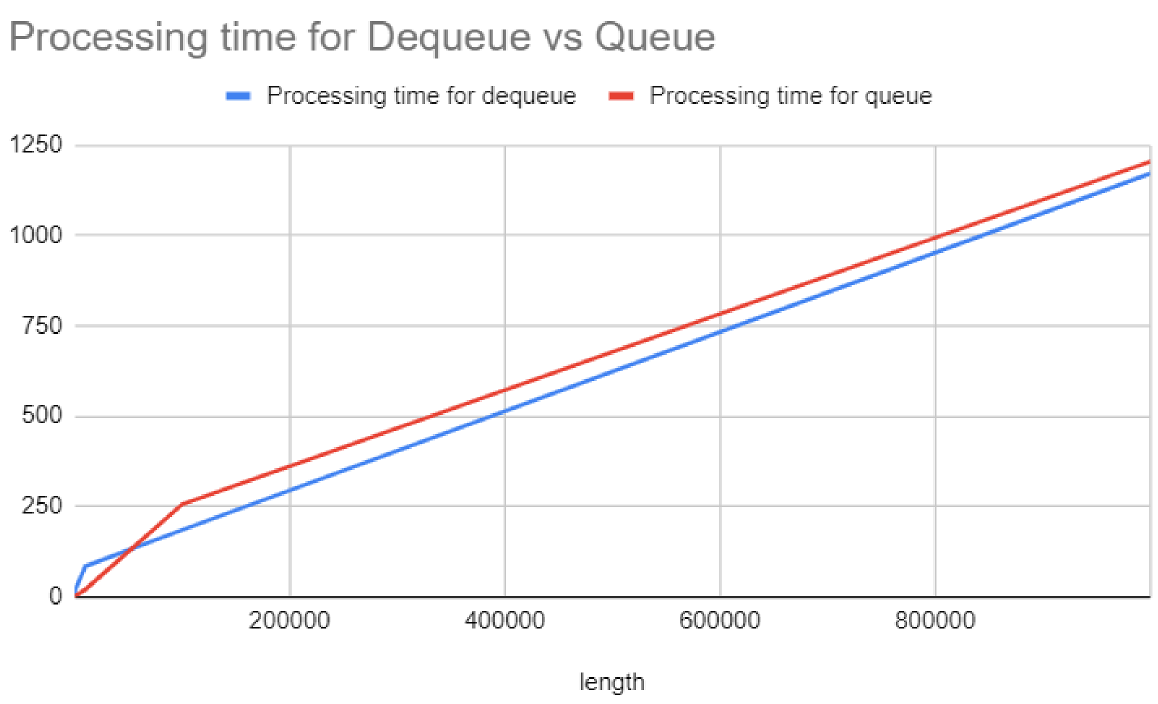


4. Code Implementation

GitHub (link): https://github.com/COMP47500-Adv-Data-Structures-in-Java/Assignment-1

5. Set of Experiments run and results

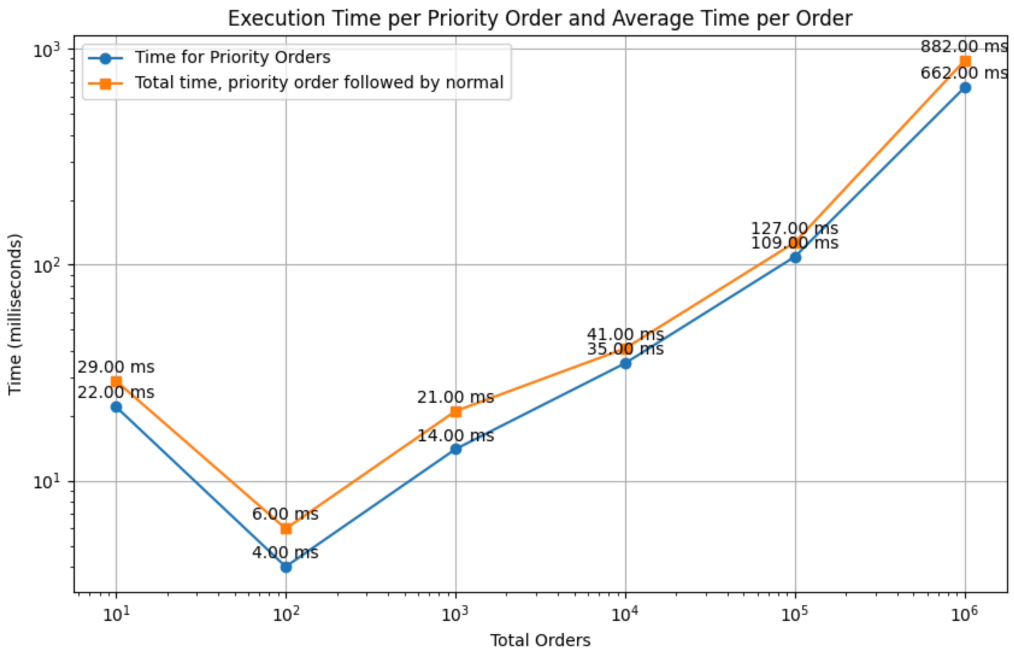
GRAPH1 : The below graph shows the total processing time of different data set against queue and dequeue.



Comments:

- From Graph1 we can observe that the processing of orders almost take simital time for both queue and dequeue as their complexity is same, the only difference is of accessing elements from both front and rear part of dequeue

GRAPH2 : The below graph shows the time taken to execute a priority order in a Deque



Comments:

- From Graph2 using a deque helps in much faster and efficient processing of priority orders.

6. Video of the Implementation running

Recording link: https://drive.google.com/file/d/1BNTCPtew62hQubMrU8zZtDwb-B5Jj3kp/view?usp=sharing

Comments: uploaded video recording in google drive.