Student Name: _____ Roll No: _____ Section: _____

Experiment No. 08

Lab 08 – Working with Queues and Operations on queues in JAVA.

Java Queue Interface

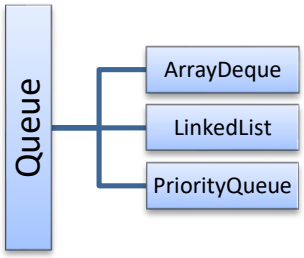
In this lab, we will learn about the Java Queue interface and its methods.

The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.

Classes that Implement Queue

Since the Queue is an interface, we cannot provide the direct implementation of it. In order to use the functionalities of Queue, we need to use classes that implement it:

- ArrayDeque
- LinkedList
- PriorityQueue



Array Deque, LinkedList and Priority Queue implements the Queue interface in Java.

Interfaces that Extend Queue

The Queue interface is also extended by various sub-interfaces:

- Deque
- BlockingQueue
- BlockingDeque

Deque, Blocking Queue and Blocking Deque extends the Queue interface.

Working of Queue Data Structure

In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front.

Working of queue data structure: first in first out.

Student Name: _____ Roll No: _____ Section: _____

How to use Queue?

In Java, we must import java.util.Queue package in order to use Queue.

```
// LinkedList implementation of Queue
Queue<String> student1 = new LinkedList<>();

// Array implementation of Queue
Queue<String> student2 = new ArrayDeque<>();

// Priority Queue implementation of Queue
Queue<String> student3 = new PriorityQueue<>();
```

Output:

Here, we have created objects student1, student2 and student3 of classes LinkedList, ArrayDeque and PriorityQueue respectively. These objects can use the functionalities of the Queue interface.

Methods of Queue

The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.

Some of the commonly used methods of the Queue interface are:

add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.
offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
element() - Returns the head of the queue. Throws an exception if the queue is empty.
peek() - Returns the head of the queue. Returns null if the queue is empty.
remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.
poll() - Returns and removes the head of the queue. Returns null if the queue is empty.
Implementation of the Queue Interface

1. Implementing the LinkedList Class

```
import java.util.Queue;
import java.util.LinkedList;
class Main {
    public static void main(String[] args) {
```

Student Name: _____ Roll No: _____ Section: _____

```
        // Creating Queue using the LinkedList class
        Queue<Integer> numbers = new LinkedList<>();

        // offer elements to the Queue
        numbers.offer(1);
        numbers.offer(2);
        numbers.offer(3);
        System.out.println("Queue: " + numbers);

        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " + accessedNumber);

        // Remove elements from the Queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

Output:

2. Implementing the Queues using Array

```
// A class to represent a queue
class Queue
{
    private int[] arr;      // array to store queue elements
    private int front;      // front points to the front element in the queue
    private int rear;       // rear points to the last element in the queue
    private int capacity;   // maximum capacity of the queue
    private int count;      // current size of the queue

    // Constructor to initialize a queue
    Queue(int size)
    {
        arr = new int[size];
        capacity = size;
        front = 0;
        rear = -1;
        count = 0;
```

Student Name: _____ Roll No: _____ Section: _____

```
    }

    // Utility function to dequeue the front element
    public int dequeue()
    {
        // check for queue underflow
        if (isEmpty())
        {
            System.out.println("Underflow\nProgram Terminated");
            System.exit(-1);
        }

        int x = arr[front];

        System.out.println("Removing " + x);

        front = (front + 1) % capacity;
        count--;

        return x;
    }

    // Utility function to add an item to the queue
    public void enqueue(int item)
    {
        // check for queue overflow
        if (isFull())
        {
            System.out.println("Overflow\nProgram Terminated");
            System.exit(-1);
        }

        System.out.println("Inserting " + item);

        rear = (rear + 1) % capacity;
        arr[rear] = item;
        count++;
    }

    // Utility function to return the front element of the queue
    public int peek()
    {
        if (isEmpty())
        {
            System.out.println("Underflow\nProgram Terminated");
            System.exit(-1);
        }
```

Student Name: _____ Roll No: _____ Section: _____

```
        return arr[front];
    }

    // Utility function to return the size of the queue
    public int size() {
        return count;
    }

    // Utility function to check if the queue is empty or not
    public boolean isEmpty() {
        return (size() == 0);
    }

    // Utility function to check if the queue is full or not
    public boolean isFull() {
        return (size() == capacity);
    }
}

class Main
{
    public static void main (String[] args)
    {
        // create a queue of capacity 5
        Queue q = new Queue(5);

        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);

        System.out.println("The front element is " + q.peek());
        q.dequeue();
        System.out.println("The front element is " + q.peek());

        System.out.println("The queue size is " + q.size());

        q.dequeue();
        q.dequeue();

        if (q.isEmpty()) {
            System.out.println("The queue is empty");
        }
        else {
            System.out.println("The queue is not empty");
        }
    }
}
```

Student Name: _____ Roll No: _____ Section: _____

Output:

3. Implementing the Priority Queue Class

```
import java.util.Queue;
import java.util.PriorityQueue;
class Main {
    public static void main(String[] args) {
        // Creating Queue using the PriorityQueue class
        Queue<Integer> numbers = new PriorityQueue<>();

        // offer elements to the Queue
        numbers.offer(5);
        numbers.offer(1);
        numbers.offer(2);
        System.out.println("Queue: " + numbers);

        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " + accessedNumber);

        // Remove elements from the Queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

Output:

Student Name: _____ Roll No: _____ Section: _____

Creating Priority Queue

In order to create a priority queue, we must import the java.util.PriorityQueue package. Once we import the package, here is how we can create a priority queue in Java.

```
PriorityQueue<Integer> numbers = new PriorityQueue<>();
```

Here, we have created a priority queue without any arguments. In this case, the head of the priority queue is the smallest element of the queue. And elements are removed in ascending order from the queue.

Methods of Priority Queue

The Priority Queue class provides the implementation of all the methods present in the Queue interface.

Insert Elements to Priority Queue

add() - Inserts the specified element to the queue. If the queue is full, it throws an exception.
offer() - Inserts the specified element to the queue. If the queue is full, it returns false.

```
import java.util.PriorityQueue;

class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();

        // Using the add() method
        numbers.add(4);
        numbers.add(2);
        System.out.println("PriorityQueue: " + numbers);

        // Using the offer() method
        numbers.offer(1);
        System.out.println("Updated PriorityQueue: " + numbers);
    }
}
```

Output:

Student Name: _____ Roll No: _____ Section: _____

Access Priority Queue Elements

To access elements from a priority queue, we can use the peek() method. This method returns the head of the queue. For example,

```
import java.util.PriorityQueue;

class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        System.out.println("PriorityQueue: " + numbers);

        // Using the peek() method
        int number = numbers.peek();
        System.out.println("Accessed Element: " + number);
    }
}
```

Output:

```
import java.util.PriorityQueue;

class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        System.out.println("PriorityQueue: " + numbers);

        // Using the remove() method
        boolean result = numbers.remove(2);
        System.out.println("Is the element 2 removed? " + result);

        // Using the poll() method
        int number = numbers.poll();
```

Student Name: _____ Roll No: _____ Section: _____

```
        System.out.println("Removed Element Using poll(): " + number);
    }
}
```

Output:

```
import java.util.PriorityQueue;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        System.out.print("PriorityQueue using iterator(): ");

        //Using the iterator() method
        Iterator<Integer> iterate = numbers.iterator();
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output:

Student Name: _____ Roll No: _____ Section: _____

Iterating Over a Priority Queue

To iterate over the elements of a priority queue, we can use the iterator() method. In order to use this method, we must import the java.util.Iterator package. For example,

PriorityQueue using iterator(): 1, 4, 2,

Other PriorityQueue Methods

Methods Descriptions

contains(element) Searches the priority queue for the specified element. If the element is found, it returns true, if not it returns false.

size() Returns the length of the priority queue.

toArray() Converts a priority queue to an array and returns it.

PriorityQueue Comparator

In all the examples above, priority queue elements are retrieved in the natural order (ascending order). However, we can customize this ordering.

For this, we need to create our own comparator class that implements the Comparator interface. For example,

```
import java.util.PriorityQueue;
import java.util.Comparator;
class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>(new CustomComparator());
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        numbers.add(3);
        System.out.print("PriorityQueue: " + numbers);
    }
}

class CustomComparator implements Comparator<Integer> {

    @Override
    public int compare(Integer number1, Integer number2) {
        int value =  number1.compareTo(number2);
        // elements are sorted in reverse order
        if (value > 0) {
            return -1;
        }
        else if (value < 0) {
            return 1;
        }
        else {
```

Student Name: _____ Roll No: _____ Section: _____

```
            return 0;
        }
    }
}
```

Output:

Student Name: _____ Roll No: _____ Section: _____

Programming Exercise

1. Write the application in which you think queues can be used.
2. Write the code of the application you have discussed in question 1.
3. In bank when old aged people come, no matter the token number you can give them priority how can you managed it using priority queues. Code the application.
4. Write an algorithm and then program that can show the number of users who can use printer for their printing After certain time a manager can access for printing his priority is high how can you manage this application using priority queue.
5. Write an algorithm and then program to clear all the elements from the queues, and priority queues
6. Discuss in details the operations you can perform on queues and priority queues