Data Structure and Algorithms (JAVA)



**6th Lab**

**Semester: Spring 2025**

**Software Engineering**

**Faculty of Information Technology, UCP Lahore, Pakistan**

Table of Contents

[**Session 1: Queue as a Data Structure** 4](#_Toc196292114)

[1.1 Queue using Array-based implementation 4](#_Toc196292115)

[1.2 Queue implementation using Linked List Class 4](#_Toc196292116)

[1.3 Priority Queue 5](#_Toc196292117)

[**Session 2: Custom Implementation of Queue and Double-Ended Queue** 6](#_Toc196292118)

[2.2 Double-Ended Queue using Doubly Linked List 7](#_Toc196292119)

[Session 3: Complexity Analysis 8](#_Toc196292120)

[Session 4: Scenario-Based Case Studies 9](#_Toc196292121)

[Case Study 1: Print Job Management System (Queues using Linked List) 9](#_Toc196292122)

[Case Study 2: Call Center Customer Support (Circular Queue Using Array) 9](#_Toc196292123)

[Conclusion 9](#_Toc196292124)

**Lab Manual: Data Structures and Algorithms using Java**

Objective

In this lab, students will implement queues using various approaches in Java. This includes developing an understanding of queues using arrays, singly and doubly circular lists, and different methods for implementing stack operations. The focus will be on practical use cases and time complexity analysis.

## **Session 1: Queue as a Data Structure**

### **1.1 Queue using Array-based implementation**

class ArrayQueue {

int front, rear, capacity;

int[] queue;

ArrayQueue(int size) {

capacity = size;

front = rear = 0;

queue = new int[capacity];

}

void enqueue(int data) {

if (rear == capacity) {

System.out.println("Queue is full");

return;

}

queue[rear++] = data;

}

void dequeue() {

if (front == rear) {

System.out.println("Queue is empty");

return;

}

for (int i = 0; i < rear - 1; i++) {

queue[i] = queue[i + 1];

}

rear--;

}

void display() {

if (front == rear) {

System.out.println("Queue is empty");

return;

}

for (int i = front; i < rear; i++) {

System.out.print(queue[i] + " ");

}

System.out.println();

}

}

### **1.2 Queue implementation using Linked List Class**

import java.util.LinkedList;

import java.util.Queue;

public class LinkedListQueue {

public static void main(String[] args) {

Queue<Integer> queue = new LinkedList<>();

queue.offer(10);

queue.offer(20);

queue.offer(30);

System.out.println("Queue: " + queue);

System.out.println("Removed: " + queue.poll());

System.out.println("Queue after removal: " + queue);

}

}

## **1.3 Priority Queue**

import java.util.PriorityQueue;

public class PriorityQueueExample {

public static void main(String[] args) {

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.offer(30);

pq.offer(10);

pq.offer(20);

System.out.println("PriorityQueue: " + pq);

System.out.println("Polled: " + pq.poll()); // Smallest element

System.out.println("After poll: " + pq);

}

}

**1.4 ArrayDeque**

import java.util.ArrayDeque;

import java.util.Queue;

public class DequeQueue {

public static void main(String[] args) {

Queue<Integer> deque = new ArrayDeque<>();

deque.offer(1);

deque.offer(2);

deque.offer(3);

System.out.println("Queue: " + deque);

System.out.println("Removed: " + deque.poll());

System.out.println("After removal: " + deque);

}

}

## **Session 2: Custom Implementation of Queue and Double-Ended Queue**

**2.1 Queue using Singly Linked List**

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

class LinkedQueue {

Node front, rear;

public LinkedQueue() {

front = rear = null;

}

public void enqueue(int value) {

Node newNode = new Node(value);

if (rear == null) {

front = rear = newNode;

return;

}

rear.next = newNode;

rear = newNode;

}

public int dequeue() {

if (front == null) {

System.out.println("Queue is empty!");

return -1;

}

int val = front.data;

front = front.next;

if (front == null) rear = null;

return val;

}

public void display() {

Node current = front;

if (current == null) {

System.out.println("Queue is empty!");

return;

}

while (current != null) {

System.out.print(current.data + " ");

current = current.next;

}

System.out.println();

}

}

### **2.2 Double Ended Queue using Doubly Linked List**

class Node {

int data;

Node prev, next;

public Node(int data) { this.data = data;

} }

class DequeDoublyLinkedList {

private Node front, rear;

public DequeDoublyLinkedList() {

front = rear = null;

}

public void addFront(int data) {

Node newNode = new Node(data);

if (front == null) {

front = rear = newNode;

} else {

newNode.next = front;

front.prev = newNode;

front = newNode;

} }

public void addRear(int data) {

Node newNode = new Node(data);

if (rear == null) {

front = rear = newNode;

} else {

rear.next = newNode;

newNode.prev = rear;

rear = newNode;

} }

public int removeFront() {

if (front == null) {

System.out.println("Deque is empty");

return -1;

}

int val = front.data;

front = front.next;

if (front != null)

front.prev = null;

else

rear = null; // if deque becomes empty

return val;

}

public int removeRear() {

if (rear == null) {

System.out.println("Deque is empty");

return -1; }

int val = rear.data;

rear = rear.prev;

if (rear != null)

rear.next = null;

else

front = null;

return val; }

public void display() {

Node temp = front;

while (temp != null) {

System.out.print(temp.data + " ");

temp = temp.next; }

System.out.println();

}

}

## **Session 3: Complexity Analysis**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Method (Array / LL / JCF)** | **Time Complexity** |
| Enqueue element | enqueue(element) | O(1) |
| Dequeue element | dequeue() | O(1) |
| Peek front element | peek() / peekFirst() (JCF) | O(1) |
| Check if queue is empty | isEmpty() | O(1) |
| Traverse the queue | Loop through queue | O(n) |
| Search element (JCF only) | contains(element) | O(n) |
| Create queue (array) | Constructor with size | O(1) |
| Create queue (linked list) | Add nodes manually | O(1) per node |
| Create queue (JCF) | Queue<Type> q = new ArrayDeque<>(); | O(1) |
| Capacity check (array) | (rear == size - 1) | O(1) |

## **Session 4: Scenario-Based Case Studies**

### **Case Study 1: Print Job Management System (Queues using Linked List)**

**Scenario:**

A print server maintains a queue of print jobs submitted by users. Jobs are printed in the order they arrive (FIFO). The system allows adding, removing, and viewing print jobs in the queue.

**Tasks:**

* Add a new print job to the queue
* Process (dequeue) the next print job
* View the current job at the front of the queue
* Display all jobs in the print queue

## **Case Study 2:** Call Center Customer Support (Circular Queue Using Array)

**Scenario:**

A customer support call center routes incoming calls into a circular queue. Once a support agent is available, the next caller is dequeued. The circular nature helps reuse the array space efficiently.

**Tasks:**

* Add a new caller to the queue
* Connect the next caller to an agent (dequeue)
* Check who is next in line
* Display all waiting callers
* Handle queue overflow when queue is full

### **Conclusion**

In this lab, students developed a core understanding of queues, deques, and circular queues using Java arrays, linked lists, and Java’s Collection Framework. Real-world scenarios such as print job scheduling and call center management were explored to demonstrate the practical applications of these data structures in handling sequential data and resource allocation.