Assignment – 10

Prakash Manikanta Irrinki
Prakashnaidu9494@gmail.com

Task 1:

Implementing a Linked List:

1) Write a class CustomLinkedList that implements a singly linked list with methods for InsertAtBeginning, InsertAtEnd, InsertAtPosition, DeleteNode, UpdateNode, and DisplayAllNodes. Test the class by performing a series of insertions, updates, and deletions.

Program:

```
public class Node {
	int data;
	Node next;
	public Node(int data) {
		super();
		this.data = data;
		this.next = null;
	}
}
public class CustomLinkedList {
	private Node head;

	public void insertAtBegining(int data) {
		Node newnode=new Node(data);
		newnode.next=head;
		head=newnode;
	}
	public void insertAtEnd(int data) {
		Node newnode=new Node(data);
		if(head==null) {
			head=newnode;
			return;
		}
		Node current=head;
		while(current.next != null) {
			current=current.next;
		}
		current.next=newnode;
	}
	public void insertAtPosition(int data,int position) {
		if(position==0) {
			insertAtBegining(data);
			return;
		}
		Node newnode=new Node(data);
		Node current=head;
		for(int i=0;i<position-1 && current != null; i++) {
			current=current.next;
		}
		if(current == null) {
			return;
```

```
            }
            newnode.next=current.next;
            current.next=newnode;
      }
      public void deleteNode(int key) {
            Node temp=head;
            Node prev=null;
            if(temp != null && temp.data == key) {
                  head=temp.next;
                  return;
            }
            while(temp != null && temp.data != key) {
                  prev = temp;
                  temp=temp.next;
            }
            if(temp == null) {
                  return;
            }
            prev.next=temp.next;
      }
      public void updateNode(int data,int position) {
            Node current=head;
            for(int i=0;i<position && current!=null; i++) {
                  current=current.next;
            }
            if(current!=null) {
                  current.data=data;
            }
      }
      public void displayAllNodes() {
            Node current=head;
            while(current!=null) {
                  System.out.print(current.data+" ");
                  current=current.next;
            }
            System.out.println();
      }
      public static void main(String[] args) {
            CustomLinkedList list=new CustomLinkedList();
            list.insertAtBegining(16);
            list.insertAtEnd(18);
            list.insertAtPosition(14,1);
            list.displayAllNodes();
            list.deleteNode(14);
            list.insertAtPosition(20,1);
            list.displayAllNodes();
            list.updateNode(142,1);
            list.displayAllNodes();
      }
}
```

Output:

```
16 14 18
16 20 18
16 142 18
```

Task 2:

Stack and Queue Operations:

1) Create a CustomStack class with operations Push, Pop, Peek, and IsEmpty. Demonstrate its LIFO behavior by pushing integers onto the stack, then popping and displaying them until the stack is empty.

Program:

```
import java.util.ArrayList;
import java.util.EmptyStackException;
public class CustomStack {
	private ArrayList<Integer> stack;
	CustomStack(){
		stack=new ArrayList<>();
	}
	public void push(int data) {
		stack.add(data);
	}
	public int pop() {
		if(!isEmpty()) {
			int lastindex=stack.size()-1;
			int item=stack.get(lastindex);
			stack.remove(lastindex);
			return item;
		}
		else {
			throw new EmptyStackException();
		}
	}
	public int peek() {
		if(!isEmpty()) {
			return stack.get(stack.size()-1);
		}
```

```
            else {
                throw new EmptyStackException();
            }
        }
        public boolean isEmpty() {
            return stack.isEmpty();
        }
        public static void main(String[] args) {

            CustomStack cs=new CustomStack();

            cs.push(16);
            cs.push(143);
            cs.push(18);

            while(!cs.isEmpty()) {
                System.out.println(cs.pop());
            }

        }

}
```

Output:

```
18
143
16
```

2) Develop a CustomQueue class with methods for Enqueue, Dequeue, Peek, and IsEmpty. Show how your queue can handle different data types by enqueuing strings and integers, then dequeuing and displaying them to confirm FIFO order.

Program:

```
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.Queue;
```

```
public class CustomQueue {
	private Queue<Integer> queue;
	private Queue<String> queue1;
	public CustomQueue() {
		queue = new LinkedList<>();
		queue1=new LinkedList<>();
	}
	public void enqueue(Integer data,String str) {
		queue.offer(data);
		queue1.offer(str);
	}
	public Object dequeue() {
		if(!isEmpty()) {
			return queue.poll()+", "+ queue1.poll();
		}
		else {
			throw new NoSuchElementException("Queue is empty");
		}
	}
	public Integer peek() {
		if(!isEmpty()) {
			return queue.peek();
		}
		else {
			throw new NoSuchElementException("Queue is empty");
		}
	}
	public boolean isEmpty() {
		return queue.isEmpty();
	}
	public static void main(String[] args) {
	   CustomQueue q = new CustomQueue();
	   q.enqueue(30,"Apple");
	  q.enqueue(25,"Orange");
	  q.enqueue(40,"Banana");
	  while(!q.isEmpty()) {
	     System.out.println(q.dequeue());
	     System.out.println(q.queue1.poll());
	  }

	}
}
```

Output:

```
30
Apple
25
Orang
40
Banana
```

Task 3:

Priority Queue Scenario:

a) Implement a priority queue to manage emergency room admissions in a hospital. Patients with higher urgency should be served before those with lower urgency.

Program:

```
import java.util.PriorityQueue;

public class EmergencyRoomAdmissionsPriority {

	static class Patient implements Comparable<Patient> {

		private String name;
		private int urgency;

		public Patient(String name, int urgency) {
			super();
			this.name = name;
			this.urgency = urgency;
		}
		@Override
		public int compareTo(Patient o) {
			return Integer.compare(this.urgency, o.urgency);
		}

		@Override
		public String toString() {
			return "Patient [name=" + name + ", urgency=" +
				urgency + "]";
		}
	}
	public static void main(String[] args) {
		PriorityQueue<Patient> eq = new PriorityQueue<>();
		eq.offer(new Patient("Prakash", 3));
		eq.offer(new Patient("Manohar", 5));
		eq.offer(new Patient("Feroze", 4));
		eq.offer(new Patient("Teja", 1));
		eq.offer(new Patient("Pavan", 2));
		while (!eq.isEmpty()) {
			Patient np = eq.poll();
			System.out.println("Treating Patient: " + np);
		}
	}
}
```

Output:

```
Treating Patient: Patient [name=Teja, urgency=1]
Treating Patient: Patient [name=Pavan, urgency=2]
Treating Patient: Patient [name=Prakash, urgency=3]
Treating Patient: Patient [name=Feroze, urgency=4]
Treating Patient: Patient [name=Manohar, urgency=5]
```