Day 16 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@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.**

**package** linkedlist;

**import** org.w3c.dom.Node;

**public** **class** CustomLinkedList {

**private** Node head;

**private** **static** **class** Node{

**int** data;

Node next;

**public** Node(**int** data) {

**this**.data = data;

**this**.next = **null**;

}

}

**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) {

**throw** **new** IllegalArgumentException("Invalid position: cannot be negatiove");

}

**if**(position == 1) {

Node newNode = **new** Node(data);

newNode.next = head;

head = newNode;

**return**;

}

Node newNode = **new** Node(data);

Node current = head;

**for**(**int** i = 1; i < position -1 && current != **null**; i++) {

current = current.next;

}

**if**(current == **null**) {

**throw** **new** IllegalArgumentException("Invalid position: exceeds linked list size");

}

newNode.next = current.next;

current.next = newNode;

}

**public** **void** deleteNode(**int** data) {

**if**(head == **null**) {

**throw** **new** IllegalStateException("List is Empty");

}

**if**(head.data == data) {

head = head.next;

**return**;

}

Node current = head;

**while**(current.next != **null** && current.next.data != data){

current = current.next;

}

**if**(current.next == **null**) {

**throw** **new** IllegalArgumentException("Data not found in the list");

}

current.next = current.next.next;

}

**public** **void** updateNode(**int** oldData, **int** newData) {

**if**(head == **null**) {

**throw** **new** IllegalStateException("List is Empty");

}

**if**(head.data == oldData) {

head.data = newData;

**return**;

}

Node current = head;

**while**(current != **null** && current.data != oldData) {

current = current.next;

}

**if**(current == **null**) {

**throw** **new** IllegalArgumentException("Old Data not found in the list");

}

current.data = newData;

}

**public** **void** displayAllNodes() {

Node current = head;

**while**(current != **null**) {

System.***out***.print(current.data + "-> ");

current = current.next;

}

System.***out***.println("null");

}

**public** **static** **void** main(String[] args) {

CustomLinkedList list = **new** CustomLinkedList();

list.insertAtBegining(2);

list.insertAtEnd(3);

list.insertAtPosition(1, 1);

list.insertAtPosition(4, 4);

System.***out***.println("Original Linked List:");

list.displayAllNodes();

list.updateNode(1,5);

list.updateNode(4,6);

System.***out***.println("Updated Linked List:");

list.displayAllNodes();

list.deleteNode(5);

list.deleteNode(6);

System.***out***.println("Updated Linked List:");

list.displayAllNodes();

}

}

**Output:**

Original Linked List:

1-> 2-> 3-> 4-> null

Updated Linked List:

5-> 2-> 3-> 6-> null

Updated Linked List: 2-> 3-> null

**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.**

**package** stack;

**import** java.util.EmptyStackException;

**public** **class** CustomStack {

**private** **static** **class** Node{

**int** data;

Node next;

Node(**int** data){

**this**.data = data;

}

}

**private** Node top;

**public** **void** push(**int** data) {

Node newNode = **new** Node(data);

newNode.next = top;

top = newNode;

}

**public** **int** pop() {

**if**(top == **null**)

**throw** **new** EmptyStackException();

**int** data = top.data;

top = top.next;

**return** data;

}

**public** **int** peek() {

**if**(top == **null**)

**throw** **new** EmptyStackException();

**return** top.data;

}

**public** **boolean** isEmpty() {

**return** top == **null**;

}

**public** **static** **void** main(String[] args) {

CustomStack stack = **new** CustomStack();

stack.push(1);

stack.push(2);

stack.push(3);

**while**(!stack.isEmpty()) {

System.***out***.println(stack.pop());

}

}

}

**Output:**

3

2

1

**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.**

**package** Queue;

**public** **class** CustomQueue<T> {

**private** T[] data;

**private** **int** front, rear;

**public** CustomQueue(**int** capacity) {

data = (T[]) **new** Object[capacity];

front = rear = -1;

}

**public** **boolean** isEmpty() {

**return** front == -1;

}

**public** **boolean** isFull() {

**return** (rear + 1) % data.length == front;

}

**public** **void** enqueue(T element) {

**if**(isFull()) {

**throw** **new** IllegalStateException("Queue Overflow");

}

**if**(isEmpty()) {

front = 0;

}

rear = (rear + 1) % data.length;

data[rear] = element;

}

**public** T dequeue() {

**if**(isEmpty()) {

**throw** **new** IllegalStateException("Queue Underflow");

}

T element = data[front];

**if**(front == rear) {

front = rear = -1;

}

**else** {

front = (front + 1) % data.length;

}

**return** element;

}

**public** T peek() {

**if**(isEmpty()) {

**throw** **new** IllegalStateException("Queue is Empty");

}

**return** data[front];

}

**public** **static** **void** main(String[] args) {

CustomQueue<Object> queue = **new** CustomQueue<Object>(5);

queue.enqueue("Hello");

queue.enqueue(10);

queue.enqueue(**true**);

**while**(!queue.isEmpty()) {

System.***out***.println(queue.dequeue());

}

}

}

**Output:**

Hello

10

True

**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.**

**package** priorityqueue;

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**public** **class** Patient **implements** Comparable<Patient> {

**private** **final** **int** id;

**private** **final** **int** Urgency;

**public** Patient(**int** id, **int** Urgency) {

**this**.id = id;

**this**.Urgency = Urgency;

}

**public** **int** getId() {

**return** id;

}

**public** **int** getUrgency() {

**return** Urgency;

}

@Override

**public** **int** compareTo(Patient other) {

**return** Integer.*compare*(**this**.Urgency, other.Urgency);

}

@Override

**public** String toString() {

**return** "Patient{" + "id=" + id + ", Urgency=" + Urgency + '}';

}

}

**public** **class** EmergencyRoom {

**public** **static** **void** main(String[] args) {

PriorityQueue<Patient> patientQueue = **new** PriorityQueue<Patient>(Comparator.*comparingInt*(Patient :: getUrgency));

patientQueue.add(**new** Patient(1, 3));

patientQueue.add(**new** Patient(2, 1));

patientQueue.add(**new** Patient(3, 5));

patientQueue.add(**new** Patient(4, 2));

**while**(!patientQueue.isEmpty()) {

System.***out***.println("Serving patient: " + patientQueue.poll());

}

}

}

**Output:**

Serving patient: Patient{id=2, Urgency=1}

Serving patient: Patient{id=4, Urgency=2}

Serving patient: Patient{id=1, Urgency=3}

Serving patient: Patient{id=3, Urgency=5}