FA21-BCS-035  
M. Barum Khan  
  
  
  
  
DSA LAB ASSIGNMENT 2

# Question1:

public class Question1 {

    private Node head;

    private class Node {

        int data;

        Node next;

        Node(int data) {

            this.data = data;

            next = null;

        }

    }

    public void insert(int data) {

        Node newNode = new Node(data);

        if (head == null) {

            head = newNode;

        } else {

            newNode.next = head;

            head = newNode;

        }

    }

    public void display() {

        Node current = head;

        while (current != null) {

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

            current = current.next;

        }

        System.out.println();

    }

    public int get(int index) {

        if (index < 0) {

            throw new IllegalArgumentException("Index cannot be negative: " + index);

        }

        Node current = head;

        int currentIndex = 0;

        while (current != null) {

            if (currentIndex == index) {

                return current.data;

            }

            current = current.next;

            currentIndex++;

        }

        throw new IllegalArgumentException("Index is out of bounds: " + index);

    }

    // this method has a complexity of O(n)

    public void insertAt(int data, int index) {

        if (index < 0) {

            throw new IllegalArgumentException("Index cannot be negative: " + index);

        }

        if (index == 0) {

            insert(data);

        } else {

            Node newNode = new Node(data);

            Node current = head;

            int currentIndex = 0;

            while (current != null) {

                if (currentIndex == index - 1) {

                    newNode.next = current.next;

                    current.next = newNode;

                    return;

                }

                current = current.next;

                currentIndex++;

            }

            throw new IllegalArgumentException("Index is out of bounds: " + index);

        }

    }

    // The complexity of this method is O(n)

    public void removeAt(int index) {

        if (index < 0) {

            throw new IllegalArgumentException("Index cannot be negative: " + index);

        }

        if (head == null) {

            throw new IllegalArgumentException("Index is out of bounds: " + index);

        }

        if (index == 0) {

            head = head.next;

        } else {

            Node current = head;

            int currentIndex = 0;

            while (current != null) {

                if (currentIndex == index - 1) {

                    if (current.next == null) {

                        throw new IllegalArgumentException("Index is out of bounds: " + index);

                    }

                    current.next = current.next.next;

                    return;

                }

                current = current.next;

                currentIndex++;

            }

            throw new IllegalArgumentException("Index is out of bounds: " + index);

        }

    }

    // The complexity of this method is O(n)

    public void reverse() {

        Node previous = null;

        Node current = head;

        while (current != null) {

            Node next = current.next;

            current.next = previous;

            previous = current;

            current = next;

        }

        head = previous;

    }

    public static void main(String[] args) {

        Question1 list = new Question1();

        list.insert(5);

        list.insert(6);

        list.insert(7);

        list.display();

        list.reverse();

        list.display();

    }

}

# Question2:

import java.util.Arrays;

class RollerCoasterRide {

    private static final int MAX\_CAPACITY = 20;

    private static final int MAX\_RIDERS = 100;

    private int[] ridersQueue;

    private int front;

    private int rear;

    private int adultCount;

    private int childCount;

    public RollerCoasterRide() {

        ridersQueue = new int[MAX\_RIDERS];

        front = -1;

        rear = -1;

        adultCount = 0;

        childCount = 0;

    }

    public void enqueue(int riderId, boolean isAdult, boolean isVIP) {

        if (isFull()) {

            System.out.println("Riders queue is full. Cannot enqueue more riders.");

            return;

        }

        if (front == -1) {

            front = 0;

        }

        if (isVIP) {

            front = (front - 1 + MAX\_RIDERS) % MAX\_RIDERS;

            ridersQueue[front] = riderId;

        } else {

            rear = (rear + 1) % MAX\_RIDERS;

            ridersQueue[rear] = riderId;

        }

        if (isAdult) {

            adultCount++;

        } else {

            childCount++;

        }

    }

    public int dequeue() {

        if (isEmpty()) {

            System.out.println("Riders queue is empty. No riders to dequeue.");

            return -1;

        }

        int dequeuedRider = ridersQueue[front];

        ridersQueue[front] = 0;

        if (front == rear) {

            front = -1;

            rear = -1;

        } else {

            front = (front + 1) % MAX\_RIDERS;

        }

        return dequeuedRider;

    }

    public boolean isFull() {

        return (rear + 1) % MAX\_RIDERS == front;

    }

    public boolean isEmpty() {

        return front == -1;

    }

    public void processRiders() {

        while (!isEmpty()) {

            int ridersCount = Math.min(size(), MAX\_CAPACITY);

            int adultRidersCount = 0;

            int childRidersCount = 0;

            for (int i = 0; i < ridersCount; i++) {

                int riderId = dequeue();

                boolean isAdult = determineIsAdult(riderId);

                if (isAdult) {

                    adultRidersCount++;

                } else {

                    childRidersCount++;

                }

            }

            System.out.println("Processing riders: " + Arrays.toString(ridersQueue));

            System.out.println("Ride started with " + ridersCount + " riders.");

            System.out.println("Adult riders: " + adultRidersCount);

            System.out.println("Child riders: " + childRidersCount);

            System.out.println();

            adultCount -= adultRidersCount;

            childCount -= childRidersCount;

        }

    }

    public int size() {

        if (isEmpty()) {

            return 0;

        }

        if (front <= rear) {

            return rear - front + 1;

        } else {

            return MAX\_RIDERS - front + rear + 1;

        }

    }

    public int getAdultCount() {

        return adultCount;

    }

    public int getChildCount() {

        return childCount;

    }

    private boolean determineIsAdult(int riderId) {

        return riderId % 2 == 0;

    }

}

public class Question2 {

    public static void main(String[] args) {

        RollerCoasterRide ride = new RollerCoasterRide();

        ride.enqueue(1, true, true);

        ride.enqueue(2, false, false);

        ride.enqueue(3, true, false);

        ride.enqueue(4, false, true);

        ride.enqueue(5, true, true);

        ride.processRiders();

        System.out.println("Adult count: " + ride.getAdultCount());

        System.out.println("Child count: " + ride.getChildCount());

    }

}

# Question3:

class CircularQueue {

    private int capacity;

    private int front;

    private int rear;

    private int[] queue;

    public CircularQueue(int capacity) {

        this.capacity = capacity;

        this.front = -1;

        this.rear = -1;

        this.queue = new int[capacity];

    }

    public boolean isEmpty() {

        return front == -1;

    }

    public boolean isFull() {

        return (rear + 1) % capacity == front;

    }

    public void enqueue(int data) {

        if (isFull()) {

            System.out.println("Circular Queue is full. Data cannot be inserted.");

        } else if (isEmpty()) {

            front = 0;

            rear = 0;

            queue[rear] = data;

            System.out.println("Data inserted: " + data);

        } else {

            rear = (rear + 1) % capacity;

            queue[rear] = data;

            System.out.println("Data inserted: " + data);

        }

    }

    public void dequeue() {

        if (isEmpty()) {

            System.out.println("Circular Queue is empty. No data to dequeue.");

        } else if (front == rear) {

            int data = queue[front];

            front = -1;

            rear = -1;

            System.out.println("Data dequeued: " + data);

        } else {

            int data = queue[front];

            front = (front + 1) % capacity;

            System.out.println("Data dequeued: " + data);

        }

    }

    public void display() {

        if (isEmpty()) {

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

        } else {

            System.out.println("Circular Queue:");

            if (rear >= front) {

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

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

                }

            } else {

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

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

                }

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

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

                }

            }

            System.out.println();

        }

    }

}

public class Question3 {

    public static void main(String[] args) {

        CircularQueue circularQueue = new CircularQueue(5);

        circularQueue.enqueue(1);

        circularQueue.enqueue(2);

        circularQueue.enqueue(3);

        circularQueue.display();

        circularQueue.dequeue();

        circularQueue.display();

        circularQueue.enqueue(4);

        circularQueue.enqueue(5);

        circularQueue.display();

        circularQueue.enqueue(6);

        circularQueue.dequeue();

        circularQueue.dequeue();

        circularQueue.dequeue();

        circularQueue.display();

    }

}