```java
//Write a program to demonstrate stack operations using an array.(i.e Push, Pop, Peep,Change,display)
import java.util.Scanner;

class Stack {
    private int maxSize;
    private int top;
    private int[] stackArray;

    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;
    }

    public void push(int value) {
        if (top < maxSize - 1) {
            stackArray[++top] = value;
            System.out.println("Pushed " + value + " onto the stack.");
        } else {
            System.out.println("Stack is full. Cannot push " + value + ".");
        }
    }

    public void pop() {
        if (top >= 0) {
```

```
        int poppedValue = stackArray[top--];
        System.out.println("Popped " + poppedValue + " from the stack.");
    } else {
        System.out.println("Stack is empty. Cannot pop.");
    }
}

public void peek() {
    if (top >= 0) {
        System.out.println("Top element of the stack is: " + stackArray[top]);
    } else {
        System.out.println("Stack is empty. Cannot peek.");
    }
}

public void change(int index, int newValue) {
    if (index >= 0 && index <= top) {
        stackArray[index] = newValue;
        System.out.println("Changed element at index " + index + " to " + newValue);
    } else {
        System.out.println("Invalid index. Cannot change element.");
    }
}

public void display() {
        if (top >= 0) {
            System.out.println("Stack elements:");
            for (int i = top; i >= 0; i--) {
                System.out.println(stackArray[i]);
            }
        } else {
            System.out.println("Stack is empty. Nothing to display.");
        }
    }
}

public class StackDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the size of the stack: ");
        int size = scanner.nextInt();

        Stack stack = new Stack(size);

        while (true) {
            System.out.println("\nStack Operations:");
            System.out.println("1. Push");
            System.out.println("2. Pop");
            System.out.println("3. Peek");
            System.out.println("4. Change");
System.out.println("5. Display");

System.out.println("6. Exit");


System.out.print("Enter your choice: ");

int choice = scanner.nextInt();


switch (choice) {

    case 1:

        System.out.print("Enter the value to push: ");

        int valueToPush = scanner.nextInt();

        stack.push(valueToPush);

        break;

    case 2:

        stack.pop();

        break;

    case 3:

        stack.peek();

        break;

    case 4:

        System.out.print("Enter the index to change: ");

        int indexToChange = scanner.nextInt();

        System.out.print("Enter the new value: ");

        int newValue = scanner.nextInt();

        stack.change(indexToChange, newValue);

        break;
```

```
                case 5:
                    stack.display();
                    break;
                case 6:
                    scanner.close();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}



//Write a program to demonstrate queue operations using an array.(i.e Enqueue, Dequeue, display, getfront, getrear)
import java.util.Scanner;

class Queue {
    private int maxSize;
    private int front;
    private int rear;
    private int[] queueArray;

    public Queue(int size) {
        maxSize = size;
    queueArray = new int[maxSize];
    front = 0;
    rear = -1;
}

public void enqueue(int value) {
    if (rear < maxSize - 1) {
        queueArray[++rear] = value;
        System.out.println("Enqueued " + value + " into the queue.");
    } else {
        System.out.println("Queue is full. Cannot enqueue " + value + ".");
    }
}

public void dequeue() {
    if (front <= rear) {
        int dequeuedValue = queueArray[front++];
        System.out.println("Dequeued " + dequeuedValue + " from the queue.");
    } else {
        System.out.println("Queue is empty. Cannot dequeue.");
    }
}

public void display() {
    if (front <= rear) {
        System.out.println("Queue elements:");
        for (int i = front; i <= rear; i++) {
            System.out.println(queueArray[i]);
        }
    } else {
        System.out.println("Queue is empty. Nothing to display.");
    }
}

public int getFront() {
    if (front <= rear) {
        return queueArray[front];
    } else {
        System.out.println("Queue is empty. Cannot get front.");
        return -1;
    }
}

public int getRear() {
    if (front <= rear) {
        return queueArray[rear];
    } else {
        System.out.println("Queue is empty. Cannot get rear.");
        return -1;
    }
    }
}

public class QueueDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the size of the queue: ");
        int size = scanner.nextInt();

        Queue queue = new Queue(size);

        while (true) {
            System.out.println("\nQueue Operations:");
            System.out.println("1. Enqueue");
            System.out.println("2. Dequeue");
            System.out.println("3. Display");
            System.out.println("4. Get Front");
            System.out.println("5. Get Rear");
            System.out.println("6. Exit");

            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
```

```
    System.out.print("Enter the value to enqueue: ");
    int valueToEnqueue = scanner.nextInt();
    queue.enqueue(valueToEnqueue);
    break;
case 2:
    queue.dequeue();
    break;
case 3:
    queue.display();
    break;
case 4:
    int frontValue = queue.getFront();
    if (frontValue != -1) {
        System.out.println("Front element of the queue is: " + frontValue);
    }
    break;
case 5:
    int rearValue = queue.getRear();
    if (rearValue != -1) {
        System.out.println("Rear element of the queue is: " + rearValue);
    }
    break;
case 6:
    scanner.close();
    System.exit(0);
```

```
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}



//Write a program to demonstrate deque operations using an array.(i.e insert from front, delete from rear, display, getfront, getrear)


import java.util.Scanner;


class Deque {
    private int maxSize;
    private int front;
    private int rear;
    private int[] dequeArray;

    public Deque(int size) {
        maxSize = size;
        dequeArray = new int[maxSize];
        front = -1;
        rear = -1;
    }
```

```java
public void insertFront(int value) {
    if ((front == 0 && rear == maxSize - 1) || (front == rear + 1)) {
        System.out.println("Deque is full. Cannot insert from front: " + value);
    } else {
        if (front == -1) {
            front = 0;
            rear = 0;
        } else if (front == 0) {
            front = maxSize - 1;
        } else {
            front--;
        }
        dequeArray[front] = value;
        System.out.println("Inserted " + value + " from the front of the deque.");
    }
}

public void deleteRear() {
    if (front == -1) {
        System.out.println("Deque is empty. Cannot delete from rear.");
    } else {
        int deletedValue = dequeArray[rear];
        if (front == rear) {
            front = -1;
            rear = -1;
        } else if (rear == 0) {
            rear = maxSize - 1;
        } else {
            rear--;
        }
        System.out.println("Deleted " + deletedValue + " from the rear of the deque.");
    }
}

public void display() {
    if (front == -1) {
        System.out.println("Deque is empty. Nothing to display.");
    } else {
        System.out.println("Deque elements:");
        int i = front;
        do {
            System.out.println(dequeArray[i]);
            if (i == rear) {
                break;
            }
            if (i == maxSize - 1) {
                i = 0;
            } else {
                i++;
            }
            } while (i != front);
        }
    }

    public int getFront() {
        if (front != -1) {
            return dequeArray[front];
        } else {
            System.out.println("Deque is empty. Cannot get front.");
            return -1;
        }
    }

    public int getRear() {
        if (rear != -1) {
            return dequeArray[rear];
        } else {
            System.out.println("Deque is empty. Cannot get rear.");
            return -1;
        }
    }
}

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

```
Scanner scanner = new Scanner(System.in);

System.out.print("Enter the size of the deque: ");

int size = scanner.nextInt();



Deque deque = new Deque(size);



while (true) {

   System.out.println("\nDeque Operations:");

   System.out.println("1. Insert from Front");

   System.out.println("2. Delete from Rear");

   System.out.println("3. Display");

   System.out.println("4. Get Front");

   System.out.println("5. Get Rear");

   System.out.println("6. Exit");



   System.out.print("Enter your choice: ");

   int choice = scanner.nextInt();



   switch (choice) {

      case 1:

         System.out.print("Enter the value to insert from front: ");

         int valueToInsert = scanner.nextInt();

         deque.insertFront(valueToInsert);

         break;

      case 2:
```

```
                deque.deleteRear();
                break;
            case 3:
                deque.display();
                break;
            case 4:
                int frontValue = deque.getFront();
                if (frontValue != -1) {
                    System.out.println("Front element of the deque is: " + frontValue);
                }
                break;
            case 5:
                int rearValue = deque.getRear();
                if (rearValue != -1) {
                    System.out.println("Rear element of the deque is: " + rearValue);
                }
                break;
            case 6:
                scanner.close();
                System.exit(0);
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    }
}
```

```
}


//Write a program to demonstrate circular queue operations using an array. (i.e Enqueue, Dequeue, display, getfront, getrear)
import java.util.Scanner;


class CircularQueue {
    private int maxSize;
    private int front;
    private int rear;
    private int[] queueArray;

    public CircularQueue(int size) {
        maxSize = size;
        queueArray = new int[maxSize];
        front = -1;
        rear = -1;
    }

    public boolean isEmpty() {
        return front == -1;
    }

    public boolean isFull() {
        return (front == 0 && rear == maxSize - 1) || (front == rear + 1);
    }
```

```java
public void enqueue(int value) {
    if (isFull()) {
        System.out.println("Circular Queue is full. Cannot enqueue " + value + ".");
    } else {
        if (isEmpty()) {
            front = 0;
        }
        rear = (rear + 1) % maxSize;
        queueArray[rear] = value;
        System.out.println("Enqueued " + value + " into the Circular Queue.");
    }
}

public void dequeue() {
    if (isEmpty()) {
        System.out.println("Circular Queue is empty. Cannot dequeue.");
    } else {
        int dequeuedValue = queueArray[front];
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % maxSize;
        }
        System.out.println("Dequeued " + dequeuedValue + " from the Circular Queue.");
    }
}

public void display() {
    if (isEmpty()) {
        System.out.println("Circular Queue is empty. Nothing to display.");
    } else {
        System.out.println("Circular Queue elements:");
        int i = front;
        do {
            System.out.println(queueArray[i]);
            if (i == rear) {
                break;
            }
            i = (i + 1) % maxSize;
        } while (i != front);
    }
}

public int getFront() {
    if (!isEmpty()) {
        return queueArray[front];
    } else {
        System.out.println("Circular Queue is empty. Cannot get front.");
```

```
            return -1;
        }
    }

    public int getRear() {
        if (!isEmpty()) {
            return queueArray[rear];
        } else {
            System.out.println("Circular Queue is empty. Cannot get rear.");
            return -1;
        }
    }
}

public class CircularQueueDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the size of the Circular Queue: ");
        int size = scanner.nextInt();

        CircularQueue circularQueue = new CircularQueue(size);

        while (true) {
            System.out.println("\nCircular Queue Operations:");
            System.out.println("1. Enqueue");
System.out.println("2. Dequeue");

System.out.println("3. Display");

System.out.println("4. Get Front");

System.out.println("5. Get Rear");

System.out.println("6. Exit");


System.out.print("Enter your choice: ");

int choice = scanner.nextInt();


switch (choice) {

   case 1:

      System.out.print("Enter the value to enqueue: ");

      int valueToEnqueue = scanner.nextInt();

      circularQueue.enqueue(valueToEnqueue);

      break;

   case 2:

      circularQueue.dequeue();

      break;

   case 3:

      circularQueue.display();

      break;

   case 4:

      int frontValue = circularQueue.getFront();

      if (frontValue != -1) {

         System.out.println("Front element of the Circular Queue is: " + frontValue);
```

```
                    }
                    break;
                case 5:
                    int rearValue = circularQueue.getRear();
                    if (rearValue != -1) {
                        System.out.println("Rear element of the Circular Queue is: " + rearValue);
                    }
                    break;
                case 6:
                    scanner.close();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}


//import java.util.Stack;

public class PostfixExpressionEvaluator {
    public static int evaluatePostfix(String expression) {
        Stack<Integer> stack = new Stack<>();
```

```
for (int i = 0; i < expression.length(); i++) {
    char ch = expression.charAt(i);

    if (Character.isDigit(ch)) {
        // If the character is a digit, push it onto the stack
        stack.push(ch - '0');
    } else {
        // If the character is an operator, pop the top two operands from the stack,
        // perform the operation, and push the result back onto the stack
        int operand2 = stack.pop();
        int operand1 = stack.pop();

        switch (ch) {
            case '+':
                stack.push(operand1 + operand2);
                break;
            case '-':
                stack.push(operand1 - operand2);
                break;
            case '*':
                stack.push(operand1 * operand2);
                break;
            case '/':
                stack.push(operand1 / operand2);
                break;
                }
            }
        }

        // The result of the postfix expression will be at the top of the stack
        return stack.pop();
    }

    public static void main(String[] args) {
        String postfixExpression = "23*5+"; // Example postfix expression: 2 * 3 + 5
        int result = evaluatePostfix(postfixExpression);
        System.out.println("Result of the postfix expression is: " + result);
    }
}

//Write a program to evaluate prefix expression using stack

import java.util.Stack;

public class PrefixExpressionEvaluator {
    public static int evaluatePrefix(String expression) {
        Stack<Integer> stack = new Stack<>();

        // Scan the expression from right to left
        for (int i = expression.length() - 1; i >= 0; i--) {
```

```
char ch = expression.charAt(i);


if (Character.isDigit(ch)) {
   // If the character is a digit, push it onto the stack
   stack.push(ch - '0');
} else {
   // If the character is an operator, pop the top two operands from the stack,
   // perform the operation, and push the result back onto the stack
   int operand1 = stack.pop();
   int operand2 = stack.pop();


   switch (ch) {
      case '+':
         stack.push(operand1 + operand2);
         break;
      case '-':
         stack.push(operand1 - operand2);
         break;
      case '*':
         stack.push(operand1 * operand2);
         break;
      case '/':
         stack.push(operand1 / operand2);
         break;
   }
```

```
            }
        }

        // The result of the prefix expression will be at the top of the stack
        return stack.pop();
    }

    public static void main(String[] args) {
        String prefixExpression = "+*23*549"; // Example prefix expression: + * 2 3 * 5 4 9
        int result = evaluatePrefix(prefixExpression);
        System.out.println("Result of the prefix expression is: " + result);
    }
}

//Write a program to demonstrate a method insert at first to add node in first position of a singly LinkedList

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
```

```
}


class LinkedList {
    Node head;


    // Method to insert a node at the first position of the linked list
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }


    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}


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

        LinkedList list = new LinkedList();

        // Insert nodes at the first position
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);

        // Display the linked list
        System.out.println("Linked List after inserting at first:");
        list.display();
    }
}
```

```
//Write a program to demonstrate a method insert at last to add node in last position of a singly LinkedList

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```
class LinkedList {

  Node head;



  // Method to insert a node at the last position of the linked list

  public void insertAtLast(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;

  }



  // Method to display the linked list

  public void display() {

    Node current = head;

    while (current != null) {

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

      current = current.next;
```

```
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedList list = new LinkedList();

        // Insert nodes at the last position
        list.insertAtLast(1);
        list.insertAtLast(2);
        list.insertAtLast(3);

        // Display the linked list
        System.out.println("Linked List after inserting at last:");
        list.display();
    }
}
```

//Write a program to demonstrate a method insert before particular value to add node in before the value entered by user in a singly LinkedList

```
import java.util.Scanner;
```

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedList {
    Node head;

    // Method to insert a node before a particular value
    public void insertBeforeValue(int valueToInsertBefore, int data) {
        Node newNode = new Node(data);

        if (head == null) {
            // If the list is empty, set the new node as the head
            head = newNode;
            return;
        }

        if (head.data == valueToInsertBefore) {
```

```
        // If the value to insert before is the head's data, insert the new node at the beginning
        newNode.next = head;
        head = newNode;
        return;
    }

    Node current = head;
    while (current.next != null) {
        if (current.next.data == valueToInsertBefore) {
            // If the next node has the value to insert before, insert the new node before it
            newNode.next = current.next;
            current.next = newNode;
            return;
        }
        current = current.next;
    }

    // If the value to insert before is not found, do nothing (or you can handle it as needed)
}

// Method to display the linked list
public void display() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " -> ");
```

```
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes into the linked list
        list.insertBeforeValue(2, 1);
        list.insertBeforeValue(4, 2);
        list.insertBeforeValue(6, 3);

        // Display the linked list
        System.out.println("Linked List:");
        list.display();

        // Ask the user for a value to insert before
        System.out.print("Enter a value to insert before: ");
        int valueToInsertBefore = scanner.nextInt();

        // Ask the user for the value to insert
        System.out.print("Enter the value to insert: ");
        int valueToInsert = scanner.nextInt();

        // Insert the new node before the specified value
        list.insertBeforeValue(valueToInsertBefore, valueToInsert);

        // Display the updated linked list
        System.out.println("Linked List after insertion:");
        list.display();

        scanner.close();
    }
}
```

//Write a program to demonstrate a method insert after particular value to add node in after the value entered by user in a singly LinkedList.

```
import java.util.Scanner;

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedList {
    Node head;

    // Method to insert a node after a particular value
    public void insertAfterValue(int valueToInsertAfter, int data) {
        Node newNode = new Node(data);
        Node current = head;

        while (current != null) {
            if (current.data == valueToInsertAfter) {
                newNode.next = current.next;
                current.next = newNode;
                return;
            }
            current = current.next;
        }

        // If the value to insert after is not found, do nothing (or you can handle it as needed)
    }

    // Method to display the linked list
    public void display() {
        Node current = head;

        while (current != null) {

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

            current = current.next;

        }

        System.out.println("null");

    }

}


public class Main {

    public static void main(String[] args) {

        LinkedList list = new LinkedList();

        Scanner scanner = new Scanner(System.in);


        // Insert nodes into the linked list

        list.insertAfterValue(1, 0);

        list.insertAfterValue(3, 2);


        // Display the linked list

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

        list.display();


        // Ask the user for a value to insert after

        System.out.print("Enter a value to insert after: ");

        int valueToInsertAfter = scanner.nextInt();
        // Ask the user for the value to insert
        System.out.print("Enter the value to insert: ");
        int valueToInsert = scanner.nextInt();

        // Insert the new node after the specified value
        list.insertAfterValue(valueToInsertAfter, valueToInsert);

        // Display the updated linked list
        System.out.println("Linked List after insertion:");
        list.display();

        scanner.close();
    }
}
```

```
//Write a program to demonstrate a method to insert a node in ordered way in a singly LinkedList


import java.util.Scanner;

class Node {
    int data;
    Node next;
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedList {
    Node head;

    // Method to insert a node in an ordered way
    public void insertOrdered(int data) {
        Node newNode = new Node(data);

        if (head == null || data < head.data) {
            // If the list is empty or the new data is smaller than the head's data,
            // insert the new node at the beginning
            newNode.next = head;
            head = newNode;
            return;
        }

        Node current = head;
        while (current.next != null && current.next.data < data) {
            current = current.next;
        }
        // Insert the new node after the current node
        newNode.next = current.next;
        current.next = newNode;
    }

    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes into the linked list in an ordered way
        list.insertOrdered(5);
        list.insertOrdered(2);
        list.insertOrdered(7);
        list.insertOrdered(4);

        // Display the ordered linked list
        System.out.println("Ordered Linked List:");
        list.display();

        // Ask the user for a value to insert in an ordered way
        System.out.print("Enter a value to insert in an ordered way: ");
        int valueToInsert = scanner.nextInt();

        // Insert the new node in an ordered way
        list.insertOrdered(valueToInsert);

        // Display the updated ordered linked list
        System.out.println("Ordered Linked List after insertion:");
        list.display();

        scanner.close();
    }
}

//Write a program to demonstrate a method to delete the node at first position in a singly LinkedList.

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedList {
    Node head;

    // Method to delete the node at the first position (head) of the linked list
    public void deleteAtFirst() {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node temp = head;
        head = head.next;
        temp.next = null; // Disconnect the deleted node from the list
    }

    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedList list = new LinkedList();

        // Insert nodes into the linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);

        // Display the linked list
        System.out.println("Linked List before deletion:");
        list.display();

        // Delete the node at the first position
        list.deleteAtFirst();
        // Display the updated linked list
        System.out.println("Linked List after deletion:");
        list.display();
    }
}



//Write a program to demonstrate a method to delete the node at last position in a singly LinkedList


class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedList {
    Node head;

    // Method to delete the node at the last position of the linked list
    public void deleteAtLast() {
```

```
    if (head == null) {
        System.out.println("List is empty. Nothing to delete.");
        return;
    }

    if (head.next == null) {
        // If there is only one node in the list, delete it
        head = null;
        return;
    }

    Node current = head;
    Node previous = null;

    while (current.next != null) {
        previous = current;
        current = current.next;
    }

    previous.next = null; // Disconnect the last node
}

// Method to insert a node at the end of the linked list
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;
    }

    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}
```

```
public class Main {

   public static void main(String[] args) {

      LinkedList list = new LinkedList();


      // Insert nodes at the end of the linked list

      list.insertAtEnd(1);

      list.insertAtEnd(2);

      list.insertAtEnd(3);


      // Display the linked list before deletion

      System.out.println("Linked List before deletion:");

      list.display();


      // Delete the node at the last position

      list.deleteAtLast();


      // Display the updated linked list

      System.out.println("Linked List after deletion:");

      list.display();

   }

}
```

//Write a program to demonstrate a method to delete the node with a value entered by user in a singly LinkedList.

```
import java.util.Scanner;
```

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node current = head;
        Node previous = null;
        boolean found = false;
        do {
            if (current.data == valueToDelete) {
                found = true;
                break;
            }
            previous = current;
            current = current.next;
        } while (current != head);

        if (found) {
            // If found, remove the node
            if (current == head) {
                // If the node to delete is the head, update the head and previous node
                head = current.next;
                previous.next = head;
            } else {
                previous.next = current.next;
            }
        } else {
            System.out.println("Value not found in the list. Nothing to delete.");
        }
    }

    // Method to insert a node at the beginning
    public void insertAtFirst(int data) {
```

```
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        head.next = head; // Circular reference
    } else {
        Node current = head;
        while (current.next != head) {
            current = current.next;
        }
        newNode.next = head;
        head = newNode;
        current.next = head; // Update the last node's reference to the new head
    }
}

// Method to insert a node at the end
public void insertAtLast(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        head.next = head; // Circular reference
    } else {
        Node current = head;
        while (current.next != head) {
            current = current.next;
            }
            current.next = newNode;
            newNode.next = head;
        }
    }

    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular Linked List is empty.");
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println(" (Head)");
    }
}

public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
```

```
Scanner scanner = new Scanner(System.in);

// Insert nodes at the beginning and end of the circular linked list
list.insertAtFirst(3);
list.insertAtFirst(2);
list.insertAtFirst(1);
list.insertAtLast(4);
list.insertAtLast(5);

// Display the circular linked list
System.out.println("Circular Linked List:");
list.display();

// Ask the user for a value to delete
System.out.print("Enter a value to delete: ");
int valueToDelete = scanner.nextInt();

// Delete the node with the specified value
list.deleteNode(valueToDelete);

// Display the updated circular linked list
System.out.println("Circular Linked List after deletion:");
list.display();

scanner.close();
```

```
    }
}

//Write a program to demonstrate a method insert at first to add node in first position of a circular LinkedList

import java.util.Scanner;

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
```

```
}

Node current = head;
Node previous = null;
boolean found = false;

do {
    if (current.data == valueToDelete) {
        found = true;
        break;
    }
    previous = current;
    current = current.next;
} while (current != head);

if (found) {
    // If found, remove the node
    if (current == head) {
        // If the node to delete is the head, update the head and previous node
        head = current.next;
        previous.next = head;
    } else {
        previous.next = current.next;
    }
} else {
        System.out.println("Value not found in the list. Nothing to delete.");
    }
}

// Method to insert a node at the beginning
public void insertAtFirst(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        head.next = head; // Circular reference
    } else {
        Node current = head;
        while (current.next != head) {
            current = current.next;
        }
        newNode.next = head;
        head = newNode;
        current.next = head; // Update the last node's reference to the new head
    }
}

// Method to insert a node at the end
public void insertAtLast(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        head.next = head; // Circular reference
    } else {
        Node current = head;
        while (current.next != head) {
            current = current.next;
        }
        current.next = newNode;
        newNode.next = head;
    }
}

// Method to display the circular linked list
public void display() {
    if (head == null) {
        System.out.println("Circular Linked List is empty.");
        return;
    }

    Node current = head;
    do {
        System.out.print(current.data + " -> ");
        current = current.next;
    } while (current != head);
    System.out.println(" (Head)");
    }
}

public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert a node at the first position of the circular linked list
        System.out.print("Enter a value to insert at first: ");
        int valueToInsert = scanner.nextInt();
        list.insertAtFirst(valueToInsert);

        // Display the updated circular linked list
        System.out.println("Circular Linked List after insertion at first:");
        list.display();

        scanner.close();
    }
}
```

//Write a program to demonstrate a method insert at last to add node in last position of a circular LinkedList.

```
import java.util.Scanner;
```

```java
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node current = head;
        Node previous = null;
        boolean found = false;
        do {
            if (current.data == valueToDelete) {
                found = true;
                break;
            }
            previous = current;
            current = current.next;
        } while (current != head);

        if (found) {
            // If found, remove the node
            if (current == head) {
                // If the node to delete is the head, update the head and previous node
                head = current.next;
                previous.next = head;
            } else {
                previous.next = current.next;
            }
        } else {
            System.out.println("Value not found in the list. Nothing to delete.");
        }
    }

    // Method to insert a node at the beginning
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            newNode.next = head;
            head = newNode;
            current.next = head; // Update the last node's reference to the new head
        }
    }

    // Method to insert a node at the end
    public void insertAtLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head;
        }
    }

    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular Linked List is empty.");
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println(" (Head)");
    }
}
public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        Scanner scanner = new Scanner(System.in);
        // Insert a node at the last position of the circular linked list
        System.out.print("Enter a value to insert at last: ");
        int valueToInsert = scanner.nextInt();
        list.insertAtLast(valueToInsert);

        // Display the updated circular linked list
        System.out.println("Circular Linked List after insertion at last:");
        list.display();

        scanner.close();
    }
}
```

//Write a program to demonstrate a method insert before particular value to add node in before the value entered by user in a circular LinkedList.

```
import java.util.Scanner;

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node current = head;
        Node previous = null;
        boolean found = false;

        do {
            if (current.data == valueToDelete) {
                found = true;
                break;
            }
            previous = current;
            current = current.next;
        } while (current != head);

        if (found) {
            // If found, remove the node
            if (current == head) {
                // If the node to delete is the head, update the head and previous node
                head = current.next;
                previous.next = head;
            } else {
                previous.next = current.next;
            }
        } else {
            System.out.println("Value not found in the list. Nothing to delete.");
        }
    }

    // Method to insert a node at the beginning
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            newNode.next = head;
            head = newNode;
            current.next = head; // Update the last node's reference to the new head
        }
    }

    // Method to insert a node at the end
    public void insertAtLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head;
        }
    }
```

```
    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular Linked List is empty.");
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println(" (Head)");
    }
}

public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes at the beginning and end of the circular linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);
}

        list.insertAtLast(4);
        list.insertAtLast(5);

        // Display the circular linked list
        System.out.println("Circular Linked List:");
        list.display();

        // Ask the user for a value to insert before
        System.out.print("Enter a value to insert before: ");
        int valueToInsertBefore = scanner.nextInt();

        // Ask the user for the value to insert
        System.out.print("Enter the value to insert: ");
        int valueToInsert = scanner.nextInt();

        // Insert the new node before the specified value
        list.insertBeforeValue(valueToInsertBefore, valueToInsert);

        // Display the updated circular linked list
        System.out.println("Circular Linked List after insertion before the value:");
        list.display();

        scanner.close();
    }
}
```

```
//Write a program to demonstrate a method insert after particular value to add node in after the value entered by user in a circular LinkedList

import java.util.Scanner;

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }
```

```
Node current = head;
Node previous = null;
boolean found = false;

do {
    if (current.data == valueToDelete) {
        found = true;
        break;
    }
    previous = current;
    current = current.next;
} while (current != head);

if (found) {
    // If found, remove the node
    if (current == head) {
        // If the node to delete is the head, update the head and previous node
        head = current.next;
        previous.next = head;
    } else {
        previous.next = current.next;
    }
} else {
    System.out.println("Value not found in the list. Nothing to delete.");
    }
}

// Method to insert a node at the beginning
public void insertAtFirst(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        head.next = head; // Circular reference
    } else {
        Node current = head;
        while (current.next != head) {
            current = current.next;
        }
        newNode.next = head;
        head = newNode;
        current.next = head; // Update the last node's reference to the new head
    }
}

// Method to insert a node at the end
public void insertAtLast(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        head.next = head; // Circular reference
    } else {
        Node current = head;
        while (current.next != head) {
            current = current.next;
        }
        current.next = newNode;
        newNode.next = head;
    }
}

// Method to display the circular linked list
public void display() {
    if (head == null) {
        System.out.println("Circular Linked List is empty.");
        return;
    }

    Node current = head;
    do {
        System.out.print(current.data + " -> ");
        current = current.next;
    } while (current != head);
    System.out.println(" (Head)");
}
```

```
}



public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes at the beginning and end of the circular linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);
        list.insertAtLast(4);
        list.insertAtLast(5);

        // Display the circular linked list
        System.out.println("Circular Linked List:");
        list.display();

        // Ask the user for a value to insert after
        System.out.print("Enter a value to insert after: ");
        int valueToInsertAfter = scanner.nextInt();

        // Ask the user for the value to insert
        System.out.print("Enter the value to insert: ");
        int valueToInsert = scanner.nextInt();

        // Insert the new node after the specified value
        list.insertAfterValue(valueToInsertAfter, valueToInsert);

        // Display the updated circular linked list
        System.out.println("Circular Linked List after insertion after the value:");
        list.display();

        scanner.close();
    }
}
```

```
//Write a program to demonstrate a method to insert a node in ordered way in a circular LinkedList
import java.util.Scanner;

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
```

```
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node current = head;
        Node previous = null;
        boolean found = false;

        do {
            if (current.data == valueToDelete) {
                found = true;
                break;
            }
            previous = current;
            current = current.next;
        } while (current != head);
        if (found) {
            // If found, remove the node
            if (current == head) {
                // If the node to delete is the head, update the head and previous node
                head = current.next;
                previous.next = head;
            } else {
                previous.next = current.next;
            }
        } else {
            System.out.println("Value not found in the list. Nothing to delete.");
        }
    }

    // Method to insert a node at the beginning
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            newNode.next = head;
            head = newNode;
            current.next = head; // Update the last node's reference to the new head
        }
    }

    // Method to insert a node at the end
    public void insertAtLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head;
        }
    }

    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular Linked List is empty.");
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println(" (Head)");
    }
}


public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes into the circular linked list in an ordered way
        list.insertOrdered(5);
        list.insertOrdered(2);
        list.insertOrdered(7);
        list.insertOrdered(4);
        // Display the ordered circular linked list
        System.out.println("Ordered Circular Linked List:");
        list.display();

        // Ask the user for a value to insert in an ordered way
        System.out.print("Enter a value to insert in an ordered way: ");
        int valueToInsert = scanner.nextInt();

        // Insert the new node in an ordered way
        list.insertOrdered(valueToInsert);

        // Display the updated ordered circular linked list
        System.out.println("Ordered Circular Linked List after insertion:");
        list.display();

        scanner.close();
    }
}
```

//Write a program to demonstrate a method to delete the node at first position in a circular LinkedList.

```
import java.util.Scanner;

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node current = head;
        Node previous = null;
        boolean found = false;

        do {
            if (current.data == valueToDelete) {
                found = true;
                break;
            }
            previous = current;
            current = current.next;
        } while (current != head);

        if (found) {
            // If found, remove the node
            if (current == head) {
                // If the node to delete is the head, update the head and previous node
                head = current.next;
                previous.next = head;
            } else {
                previous.next = current.next;
            }
        } else {
            System.out.println("Value not found in the list. Nothing to delete.");
        }
    }

    // Method to insert a node at the beginning
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference

        } else {

            Node current = head;

            while (current.next != head) {

                current = current.next;

            }

            newNode.next = head;

            head = newNode;

            current.next = head; // Update the last node's reference to the new head

        }

    }


    // Method to insert a node at the end

    public void insertAtLast(int data) {

        Node newNode = new Node(data);

        if (head == null) {

            head = newNode;

            head.next = head; // Circular reference

        } else {

            Node current = head;

            while (current.next != head) {

                current = current.next;

            }

            current.next = newNode;

            newNode.next = head;
        }
    }

    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular Linked List is empty.");
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println(" (Head)");
    }
}

public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();

        // Insert nodes at the beginning and end of the circular linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);
        list.insertAtLast(4);
        list.insertAtLast(5);

        // Display the circular linked list before deletion
        System.out.println("Circular Linked List before deletion:");
        list.display();

        // Delete the node at the first position
        list.deleteAtFirst();

        // Display the updated circular linked list
        System.out.println("Circular Linked List after deletion at first:");
        list.display();
    }
}
```

```
//Write a program to demonstrate a method to delete the node at last position in a circular LinkedList.
import java.util.Scanner;

class Node {
    int data;
    Node next;
```

```java
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node current = head;
        Node previous = null;
        boolean found = false;

        do {
            if (current.data == valueToDelete) {
                found = true;
                break;
            }
            previous = current;
            current = current.next;
        } while (current != head);

        if (found) {
            // If found, remove the node
            if (current == head) {
                // If the node to delete is the head, update the head and previous node
                head = current.next;
                previous.next = head;
            } else {
                previous.next = current.next;
            }
        } else {
            System.out.println("Value not found in the list. Nothing to delete.");
        }
    }

    // Method to insert a node at the beginning
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            newNode.next = head;
            head = newNode;
            current.next = head; // Update the last node's reference to the new head
        }
    }

    // Method to insert a node at the end
    public void insertAtLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head;
        }
    }

    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular Linked List is empty.");
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println(" (Head)");
    }
}


public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();

        // Insert nodes at the beginning and end of the circular linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);
        list.insertAtLast(4);
        list.insertAtLast(5);

        // Display the circular linked list before deletion
        System.out.println("Circular Linked List before deletion:");
        list.display();

        // Delete the node at the last position
        list.deleteAtLast();

        // Display the updated circular linked list
        System.out.println("Circular Linked List after deletion at last:");
        list.display();
    }
}
```

//Write a program to demonstrate a method to delete the node with a value entered by user in a circular LinkedList

```
import java.util.Scanner;

class Node {
    int data;
    Node next;
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head;

    // Method to delete a node with a specific value
    public void deleteNode(int valueToDelete) {
        if (head == null) {
            System.out.println("List is empty. Nothing to delete.");
            return;
        }

        Node current = head;
        Node previous = null;
        boolean found = false;

        do {
            if (current.data == valueToDelete) {
                found = true;
                break;
            }
            previous = current;
            current = current.next;
        } while (current != head);

        if (found) {
            // If found, remove the node
            if (current == head) {
                // If the node to delete is the head, update the head and previous node
                head = current.next;
                previous.next = head;
            } else {
                previous.next = current.next;
            }
        } else {
            System.out.println("Value not found in the list. Nothing to delete.");
        }
    }

    // Method to insert a node at the beginning
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            newNode.next = head;
            head = newNode;
            current.next = head; // Update the last node's reference to the new head
        }
    }

    // Method to insert a node at the end
    public void insertAtLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular reference
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head;
        }
    }
```

```
    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular Linked List is empty.");
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println(" (Head)");
    }
}
// Use the same CircularLinkedList class and Main class as provided in a previous response.

public class Main {
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes into the circular linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);

        list.insertAtFirst(1);

        list.insertAtLast(4);

        list.insertAtLast(5);


        // Display the circular linked list before deletion

        System.out.println("Circular Linked List before deletion:");

        list.display();


        // Ask the user for a value to delete

        System.out.print("Enter a value to delete: ");

        int valueToDelete = scanner.nextInt();


        // Delete the node with the specified value

        list.deleteNode(valueToDelete);


        // Display the updated circular linked list

        System.out.println("Circular Linked List after deletion:");

        list.display();


        scanner.close();

    }

}
```

//Write a program to demonstrate a method insert at first to add node in first position of a doubly LinkedList

```
class Node {
    int data;
    Node prev;
    Node next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}

class DoublyLinkedList {
    Node head;

    // Method to insert a node at the first position
    public void insertAtFirst(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
        }
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();

        // Insert nodes at the beginning of the doubly linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);

        // Display the doubly linked list
        System.out.println("Doubly Linked List:");
        list.display();
    }
}
```

//Write a program to demonstrate a method insert at last to add node in last position of a doubly LinkedList.

```
class Node {
    int data;
    Node prev;
    Node next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}

class DoublyLinkedList {
    Node head;
    Node tail;

    // Method to insert a node at the last position
    public void insertAtLast(int data) {
        Node newNode = new Node(data);
        if (tail == null) {
            head = tail = newNode;
        } else {
            newNode.prev = tail;
            tail.next = newNode;
            tail = newNode;
        }
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();

        // Insert nodes at the end of the doubly linked list
        list.insertAtLast(1);
        list.insertAtLast(2);
        list.insertAtLast(3);

        // Display the doubly linked list
        System.out.println("Doubly Linked List:");
        list.display();
    }
}
```

```
//Write a program to demonstrate a method insert before particular value to add node in before the value entered by user in a doubly LinkedList

import java.util.Scanner;

class Node {
    int data;
    Node prev;
    Node next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
```

```
}

class DoublyLinkedList {
    Node head;

    // Method to insert a node before a particular value
    public void insertBeforeValue(int valueToInsertBefore, int data) {
        Node newNode = new Node(data);

        if (head == null) {
            head = newNode;
            return;
        }

        if (head.data == valueToInsertBefore) {
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
            return;
        }

        Node current = head;
        while (current != null) {
            if (current.data == valueToInsertBefore) {
                newNode.prev = current.prev;
                newNode.next = current;
                current.prev.next = newNode;
                current.prev = newNode;
                return;
            }
            current = current.next;
        }
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        Scanner scanner = new Scanner(System.in);
        // Insert nodes into the doubly linked list

        list.insertBeforeValue(2, 1);

        list.insertBeforeValue(4, 2);

        list.insertBeforeValue(6, 3);


        // Display the doubly linked list

        System.out.println("Doubly Linked List:");

        list.display();


        scanner.close();

    }

}
```

//Write a program to demonstrate a method insert after particular value to add node in after the value entered by user in a doubly LinkedList

```
import java.util.Scanner;


class Node {

    int data;

    Node prev;

    Node next;


    public Node(int data) {

        this.data = data;
        this.prev = null;
        this.next = null;
    }
}

class DoublyLinkedList {
    Node head;

    // Method to insert a node after a particular value
    public void insertAfterValue(int valueToInsertAfter, int data) {
        Node newNode = new Node(data);

        if (head == null) {
            head = newNode;
            return;
        }

        Node current = head;
        while (current != null) {
            if (current.data == valueToInsertAfter) {
                newNode.prev = current;
                newNode.next = current.next;
                if (current.next != null) {
                    current.next.prev = newNode;
                }
                current.next = newNode;
                return;
            }
            current = current.next;
        }
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes into the doubly linked list
        list.insertAfterValue(1, 2);
        list.insertAfterValue(2, 3);
        list.insertAfterValue(3, 4);

        // Display the doubly linked list
        System.out.println("Doubly Linked List:");
        list.display();

        scanner.close();
    }
}

//Write a program to demonstrate a method to insert a node in ordered way in a doubly LinkedList

import java.util.Scanner;

class Node {
    int data;
    Node prev;
    Node next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
```

```
}


class DoublyLinkedList {
  Node head;


  // Method to insert a node in an ordered way
  public void insertOrdered(int data) {
    Node newNode = new Node(data);


    if (head == null || data < head.data) {
      // If the list is empty or the new data is smaller than the head's data,
      // insert the new node at the beginning
      newNode.next = head;
      if (head != null) {
        head.prev = newNode;
      }
      head = newNode;
      return;
    }


    Node current = head;
    while (current.next != null && current.next.data < data) {
      current = current.next;
    }
```

```
        // Insert the new node after the current node
        newNode.next = current.next;
        newNode.prev = current;
        if (current.next != null) {
            current.next.prev = newNode;
        }
        current.next = newNode;
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        Scanner scanner = new Scanner(System.in);
        // Insert nodes into the doubly linked list in an ordered way
        list.insertOrdered(5);
        list.insertOrdered(2);
        list.insertOrdered(7);
        list.insertOrdered(4);

        // Display the ordered doubly linked list
        System.out.println("Ordered Doubly Linked List:");
        list.display();

        scanner.close();
    }
}

//Write a program to demonstrate a method to delete the node at first position in a doubly LinkedList
import java.util.Scanner;

class Node {
    int data;
    Node prev;
    Node next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}

class DoublyLinkedList {
    Node head;

    // Method to insert a node in an ordered way
    public void insertOrdered(int data) {
        Node newNode = new Node(data);

        if (head == null || data < head.data) {
            // If the list is empty or the new data is smaller than the head's data,
            // insert the new node at the beginning
            newNode.next = head;
            if (head != null) {
                head.prev = newNode;
            }
            head = newNode;
            return;
        }

        Node current = head;
        while (current.next != null && current.next.data < data) {
            current = current.next;
        }

        // Insert the new node after the current node
        newNode.next = current.next;
        newNode.prev = current;
        if (current.next != null) {
            current.next.prev = newNode;
        }
        current.next = newNode;
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}
public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        // Insert nodes at the beginning of the doubly linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);

        // Display the doubly linked list before deletion
        System.out.println("Doubly Linked List before deletion at first:");
        list.display();

        // Delete the node at the first position
        list.deleteAtFirst();

        // Display the updated doubly linked list
        System.out.println("Doubly Linked List after deletion at first:");
        list.display();
    }
}

//Write a program to demonstrate a method to delete the node at last position in a doubly LinkedList
import java.util.Scanner;

class Node {
    int data;
    Node prev;
    Node next;
    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}

class DoublyLinkedList {
    Node head;

    // Method to insert a node in an ordered way
    public void insertOrdered(int data) {
        Node newNode = new Node(data);

        if (head == null || data < head.data) {
            // If the list is empty or the new data is smaller than the head's data,
            // insert the new node at the beginning
            newNode.next = head;
            if (head != null) {
                head.prev = newNode;
            }
            head = newNode;
            return;
        }
        Node current = head;
        while (current.next != null && current.next.data < data) {
            current = current.next;
        }

        // Insert the new node after the current node
        newNode.next = current.next;
        newNode.prev = current;
        if (current.next != null) {
            current.next.prev = newNode;
        }
        current.next = newNode;
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();

        // Insert nodes at the end of the doubly linked list
        list.insertAtLast(1);
        list.insertAtLast(2);
        list.insertAtLast(3);

        // Display the doubly linked list before deletion
        System.out.println("Doubly Linked List before deletion at last:");
        list.display();

        // Delete the node at the last position
        list.deleteAtLast();

        // Display the updated doubly linked list
        System.out.println("Doubly Linked List after deletion at last:");
        list.display();
    }
}
```

//Write a program to demonstrate a method to delete the node with a value entered by user in a doubly LinkedList

```
import java.util.Scanner;

class Node {
  int data;
  Node prev;
  Node next;

  public Node(int data) {
    this.data = data;
    this.prev = null;
    this.next = null;
  }
}

class DoublyLinkedList {
  Node head;

  // Method to insert a node in an ordered way
  public void insertOrdered(int data) {
    Node newNode = new Node(data);

    if (head == null || data < head.data) {
      // If the list is empty or the new data is smaller than the head's data,
      // insert the new node at the beginning
      newNode.next = head;
```

```
            if (head != null) {
                head.prev = newNode;
            }
            head = newNode;
            return;
        }

        Node current = head;
        while (current.next != null && current.next.data < data) {
            current = current.next;
        }

        // Insert the new node after the current node
        newNode.next = current.next;
        newNode.prev = current;
        if (current.next != null) {
            current.next.prev = newNode;
        }
        current.next = newNode;
    }

    // Method to display the doubly linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        Scanner scanner = new Scanner(System.in);

        // Insert nodes into the doubly linked list
        list.insertAtFirst(3);
        list.insertAtFirst(2);
        list.insertAtFirst(1);
        list.insertAtLast(4);
        list.insertAtLast(5);

        // Display the doubly linked list before deletion
        System.out.println("Doubly Linked List before deletion:");
        list.display();

        // Ask the user for a value to delete
        System.out.print("Enter a value to delete: ");
        int valueToDelete = scanner.nextInt();


        // Delete the node with the specified value
        list.deleteNode(valueToDelete);


        // Display the updated doubly linked list
        System.out.println("Doubly Linked List after deletion:");
        list.display();


        scanner.close();
    }
}



//Write a program to delete duplicate values from a given singly LinkedList
class Node {
    int data;
    Node next;


    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```
class SinglyLinkedList {
    Node head;

    // Method to delete duplicate values from the linked list
    public void deleteDuplicates() {
        if (head == null || head.next == null) {
            return; // No duplicates in an empty or single-node list
        }

        Node current = head;
        while (current != null) {
            Node runner = current;
            while (runner.next != null) {
                if (runner.next.data == current.data) {
                    runner.next = runner.next.next; // Remove duplicate node
                } else {
                    runner = runner.next;
                }
            }
            current = current.next;
        }
    }

    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        // Insert nodes with duplicate values into the linked list
        list.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(2);
        Node fourth = new Node(3);
        Node fifth = new Node(3);

        list.head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;
        // Display the linked list before deleting duplicates
        System.out.println("Linked List before deleting duplicates:");
        list.display();

        // Delete duplicate values
        list.deleteDuplicates();

        // Display the updated linked list
        System.out.println("Linked List after deleting duplicates:");
        list.display();
    }
}

//Write a program to delete only even values from a given singly LinkedLis

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```
class SinglyLinkedList {

  Node head;


  // Method to delete only even values from the linked list

  public void deleteEvenValues() {

    if (head == null) {

      return;

    }


    while (head != null && head.data % 2 == 0) {

      head = head.next; // Remove even value from the beginning

    }


    Node current = head;

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

      if (current.next.data % 2 == 0) {

        current.next = current.next.next; // Remove even value

      } else {

        current = current.next;

      }

    }

  }


  // Method to display the linked list

  public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        // Insert nodes with even and odd values into the linked list
        list.head = new Node(2);
        Node second = new Node(4);
        Node third = new Node(1);
        Node fourth = new Node(6);
        Node fifth = new Node(8);

        list.head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;
        // Display the linked list before deleting even values
        System.out.println("Linked List before deleting even values:");
        list.display();

        // Delete even values
        list.deleteEvenValues();

        // Display the updated linked list
        System.out.println("Linked List after deleting even values:");
        list.display();
    }
}

//Write a program to delete only odd values from a given singly LinkedList

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```
class SinglyLinkedList {
    Node head;

    // Method to delete only odd values from the linked list
    public void deleteOddValues() {
        if (head == null) {
            return;
        }

        while (head != null && head.data % 2 != 0) {
            head = head.next; // Remove odd value from the beginning
        }

        Node current = head;
        while (current != null && current.next != null) {
            if (current.next.data % 2 != 0) {
                current.next = current.next.next; // Remove odd value
            } else {
                current = current.next;
            }
        }
    }

    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        // Insert nodes with even and odd values into the linked list
        list.head = new Node(2);
        Node second = new Node(4);
        Node third = new Node(1);
        Node fourth = new Node(6);
        Node fifth = new Node(8);

        list.head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;
        // Display the linked list before deleting odd values
        System.out.println("Linked List before deleting odd values:");
        list.display();

        // Delete odd values
        list.deleteOddValues();

        // Display the updated linked list
        System.out.println("Linked List after deleting odd values:");
        list.display();
    }
}

//Write a program to delete odd positioned nodes from a given singly LinkedList

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```
class SinglyLinkedList {
    Node head;

    // Method to delete odd-positioned nodes from the linked list
    public void deleteOddPositionedNodes() {
        Node current = head;

        while (current != null && current.next != null) {
            current.next = current.next.next; // Remove odd-positioned node
            current = current.next;
        }
    }

    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
```

```
public static void main(String[] args) {
    SinglyLinkedList list = new SinglyLinkedList();

    // Insert nodes into the linked list
    list.head = new Node(1);
    Node second = new Node(2);
    Node third = new Node(3);
    Node fourth = new Node(4);
    Node fifth = new Node(5);

    list.head.next = second;
    second.next = third;
    third.next = fourth;
    fourth.next = fifth;

    // Display the linked list before deleting odd-positioned nodes
    System.out.println("Linked List before deleting odd-positioned nodes:");
    list.display();

    // Delete odd-positioned nodes
    list.deleteOddPositionedNodes();

    // Display the updated linked list
    System.out.println("Linked List after deleting odd-positioned nodes:");
    list.display();
    }
}

//Write a program to delete even positioned nodes from a given singly LinkedList

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class SinglyLinkedList {
    Node head;

    // Method to delete even-positioned nodes from the linked list
    public void deleteEvenPositionedNodes() {
        if (head == null || head.next == null) {
            return; // No even-positioned nodes in an empty or single-node list
        }

        Node current = head;
        while (current != null && current.next != null) {
            current.next = current.next.next; // Remove even-positioned node
            current = current.next;
        }
    }

    // Method to display the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}

public class Main {
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        // Insert nodes into the linked list
        list.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);
        Node fourth = new Node(4);
        Node fifth = new Node(5);

        list.head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;

        // Display the linked list before deleting even-positioned nodes
        System.out.println("Linked List before deleting even-positioned nodes:");
        list.display();

        // Delete even-positioned nodes
        list.deleteEvenPositionedNodes();

        // Display the updated linked list
        System.out.println("Linked List after deleting even-positioned nodes:");
        list.display();
    }
}
```