```
//Saket M Kharche
// Implementation of Doubly Linked List with all operations
class DoublyLinkedList {
    // Node class representing each element in the doubly linked list
    // This is an inner class that's only used within the DoublyLinkedList class
    private class Node {
        int data;  // Stores the actual value/data of the node
        Node prev; // Reference to the previous node in the list
        Node next; // Reference to the next node in the list

        // Constructor to initialize a new node with data
        // When we create a node, it starts disconnected (prev and next are null)
        public Node(int data) {
            this.data = data;
            this.prev = null;
            this.next = null;
        }
    }

    // Member variables for the DoublyLinkedList class
    private Node head;  // Reference to the first node in the list
    private Node tail;  // Reference to the last node in the list

    // Constructor to initialize an empty list
    // When we create a new list, it starts with no nodes
    public DoublyLinkedList() {
        head = null;
        tail = null;
    }

    // Insert a new node at the beginning of the list
    public void insertAtBeginning(int value) {
        // Step 1: Create a new node with the given value
        Node newNode = new Node(value);

        // Step 2: Check if list is empty
        if (head == null) {
            // For an empty list, the new node becomes both head and tail
            head = newNode;
            tail = newNode;
        } else {
            // Step 3: For non-empty list, connect the new node to the current head
            newNode.next = head;  // New node points forward to current head
            head.prev = newNode;  // Current head points backward to new node
            head = newNode;       // Update head to point to the new node
        }
        System.out.println(value + " inserted at beginning.");
    }

    // Insert a new node at the end of the list
    public void insertAtEnd(int value) {
        // Step 1: Create a new node with the given value
        Node newNode = new Node(value);

        // Step 2: Check if list is empty
        if (head == null) {
            // For an empty list, the new node becomes both head and tail
            head = newNode;
            tail = newNode;
        } else {
            // Step 3: For non-empty list, connect the new node to the current tail
            tail.next = newNode;  // Current tail points forward to new node
            newNode.prev = tail;  // New node points backward to current tail
            tail = newNode;       // Update tail to point to the new node
        }
        System.out.println(value + " inserted at end.");
    }

    // Insert a node after a specified node
    public void insertAfterNode(int value, int afterValue) {
        // Step 1: Start from the head node
        Node current = head;

        // Step 2: Traverse the list to find the node that contains `afterValue`
        // Keep moving forward until we reach the end or find the target value
        while (current != null && current.data != afterValue) {
            current = current.next;
        }

        // Step 3: If the specified node with `afterValue` is not found
        if (current == null) {
            System.out.println("Node with value " + afterValue + " not found.");
            return; // Exit the function
        }

        // Step 4: Create a new node with the given value
        Node newNode = new Node(value);

        // Step 5: Set up the connections for the new node
        // This is the crucial part of insertion in a doubly linked list

        // Link the new node's next to current node's next
        newNode.next = current.next;
        // Link the new node's prev to current node
        newNode.prev = current;

        // Step 6: Update connections from adjacent nodes
        // If current is not the tail, update the next node's prev pointer
        if (current.next != null) {
            current.next.prev = newNode;
        }
        // Update the current node's next pointer
        current.next = newNode;

        // Step 7: If we inserted after the tail, update the tail pointer
        if (current == tail) {
            tail = newNode;
        }

        System.out.println(value + " inserted after " + afterValue + ".");
    }

    // Delete a node from the beginning
    public void deleteFromBeginning() {
        // Step 1: Check if list is empty
        if (head == null) {
            System.out.println("List is empty!");
            return;
        }

        // Step 2: Print what we're deleting
        System.out.println("Deleted: " + head.data);

        // Step 3: Move head to the next node
        head = head.next;

        // Step 4: Update connections based on the new state
        if (head != null) {
            // If there's still a node after deletion, remove its backward link
            head.prev = null;
        } else {
            // If list becomes empty after deletion, update tail as well
            tail = null;
        }
    }

    // Delete a node from the end
    public void deleteFromEnd() {
        // Step 1: Check if list is empty
        if (tail == null) {
            System.out.println("List is empty!");
            return;
        }

        // Step 2: Print what we're deleting
        System.out.println("Deleted: " + tail.data);

        // Step 3: Move tail to the previous node
        tail = tail.prev;

        // Step 4: Update connections based on the new state
        if (tail != null) {
            // If there's still a node after deletion, remove its forward link
            tail.next = null;
        } else {
            // If list becomes empty after deletion, update head as well
            head = null;
        }
    }

    // Delete a node with a specific value
    public void deleteByValue(int value) {
        // Step 1: Start from the head node
        Node current = head;

        // Step 2: Traverse the list to find the node with the target value
        while (current != null && current.data != value) {
            current = current.next;
        }

        // Step 3: If node not found, print message and return
        if (current == null) {
            System.out.println("Node with value " + value + " not found.");
            return;
        }

        // Step 4: Print what we're deleting
        System.out.println("Deleted: " + value);

        // Step 5: Update the connections to bypass the node being deleted

        // If not the head node, update the previous node's next pointer
        if (current.prev != null) {
            current.prev.next = current.next;
        } else {
            // If it's the head node, update head pointer
            head = current.next;
        }

        // If not the tail node, update the next node's prev pointer
        if (current.next != null) {
            current.next.prev = current.prev;
        } else {
            // If it's the tail node, update tail pointer
            tail = current.prev;
        }

        // The node 'current' is now disconnected and will be garbage collected
    }

    // Search for a node with a given value
    public boolean search(int value) {
        // Step 1: Start from the head node
        Node current = head;

        // Step 2: Traverse the list
        while (current != null) {
            // Step 3: Check if current node has the target value
            if (current.data == value) {
                System.out.println("Found: " + value);
                return true;
            }
            // Move to the next node
            current = current.next;
        }

        // Step 4: If we reach here, the value wasn't found
        System.out.println("Not found: " + value);
        return false;
    }

    // Display list forward (from head to tail)
    public void displayForward() {
        // Step 1: Start from the head node
        Node current = head;

        // Step 2: Print the starting message
        System.out.print("List (forward): ");

        // Step 3: Traverse the list from head to tail
        while (current != null) {
            // Print each node's data
            System.out.print(current.data + " ");
            // Move to the next node
            current = current.next;
        }
        System.out.println();
    }

    // Display list backward (from tail to head)
    // This demonstrates the advantage of a doubly linked list over a singly linked list
    public void displayBackward() {
        // Step 1: Start from the tail node
        Node current = tail;

        // Step 2: Print the starting message
        System.out.print("List (backward): ");

        // Step 3: Traverse the list from tail to head
        while (current != null) {
            // Print each node's data
            System.out.print(current.data + " ");
            // Move to the previous node
            current = current.prev;
        }
        System.out.println();
    }
}

// Main class to test Doubly Linked List
public class Main {
    public static void main(String[] args) {
        // Create a new empty doubly linked list
        DoublyLinkedList dll = new DoublyLinkedList();

        // Insert operations demonstration
        dll.insertAtBeginning(10);  // List: 10
        dll.insertAtEnd(30);        // List: 10 -> 30
        dll.insertAtBeginning(5);   // List: 5 -> 10 -> 30
        dll.insertAtEnd(40);        // List: 5 -> 10 -> 30 -> 40
        dll.insertAfterNode(20, 10); // List: 5 -> 10 -> 20 -> 30 -> 40

        // Display the list in both directions
        dll.displayForward();  // Expected Output: 5 10 20 30 40
        dll.displayBackward();  // Expected Output: 40 30 20 10 5

        // Delete operations demonstration
        dll.deleteFromBeginning();  // Removes 5, List: 10 -> 20 -> 30 -> 40
        dll.displayForward();       // Expected Output: 10 20 30 40

        dll.deleteFromEnd();        // Removes 40, List: 10 -> 20 -> 30
        dll.displayForward();       // Expected Output: 10 20 30

        dll.deleteByValue(20);      // Removes 20, List: 10 -> 30
        dll.displayForward();       // Expected Output: 10 30

        // Search operation demonstration
        dll.search(30);             // Should find 30
        dll.search(50);             // Should not find 50
    }
}
```