class Node {

int data;

Node next;

Node prev;

Node(int data) {

this.data = data;

this.next = null;

this.prev = null;

}

}

class Question8 {

private Node head;

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

newNode.prev = temp;

}

}

public void deleteByValue(int value) {

if (head == null) return;

if (head.data == value) {

head = head.next;

if (head != null) {

head.prev = null;

}

return;

}

Node current = head;

while (current != null && current.data != value) {

current = current.next;

}

if (current != null) {

if (current.next != null) {

current.next.prev = current.prev;

}

if (current.prev != null) {

current.prev.next = current.next;

}

}

}

public void reverse() {

Node current = head;

Node temp = null;

while (current != null) {

temp = current.prev;

current.prev = current.next;

current.next = temp;

current = current.prev;

}

if (temp != null) {

head = temp.prev;

}

}

public String traverse() {

StringBuilder result = new StringBuilder();

Node current = head;

result.append("[");

while (current != null) {

result.append(current.data);

current = current.next;

if (current != null) {

result.append(", ");

}

}

result.append("]");

return result.toString();

}

public static void main(String[] args) {

// Test Case 1

Question8 list1 = new Question8();

list1.insert(5);

list1.insert(10);

list1.insert(15);

list1.insert(20);

System.out.println("Input: List = " + list1.traverse());

list1.reverse();

System.out.println("Output: List = " + list1.traverse());

// Test Case 2

Question8 list2 = new Question8();

list2.insert(4);

list2.insert(8);

list2.insert(12);

System.out.println("Input: List = " + list2.traverse());

list2.reverse();

System.out.println("Output: List = " + list2.traverse());

}

}  
  
  
explanation  
The provided Java implementation of a doubly linked list allows for insertion of nodes, deletion of nodes by value, and reversal of the list. Each node consists of three components: the data value, a pointer to the next node, and a pointer to the previous node, enabling bidirectional traversal. The insert method adds nodes to the end of the list, while the deleteByValue method removes a node based on its data value. The reverse method swaps the next and prev pointers of each node, effectively reversing the order of the list. The traverse method constructs a string representation of the list's contents, facilitating easy output of the current state of the list.  
  
time and space complexity  
The time complexity for insertion and deletion operations in a doubly linked list is O(1) for adding/removing nodes at the head or tail, while it becomes O(n) for operations that require traversal, such as deleting a node by value. The reversal operation has a time complexity of O(n) because it processes each node exactly once. The space complexity is O(n) as well, where n is the number of nodes in the list, due to the storage required for the nodes themselves. Additionally, the algorithm uses a constant amount of extra space (O(1)) for pointer manipulation during operations  
  
output  
