**Assignment 4**

**Following things to be added in each question:**

**-Program**

**-Flow chart**

**-Explanation**

**-Output**

**-Time and Space complexity**

**Submission Date: 3/10/2024**

**1. Implement a singly linked list with basic operations: insert, delete, search.**

* **Test Case 1:  
  Input: Insert 3 → Insert 7 → Insert 5 → Delete 7 → Search 5  
  Output: List = [3, 5], Found = True**
* **Test Case 2:  
  Input: Insert 9 → Insert 4 → Delete 4 → Search 10  
  Output: List = [9], Found = False**

**Program code:**

**package Assignment.org;**

**class SinglyLinkedList {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Delete by value**

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

**if (head == null) return;**

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

**head = head.next;**

**return;**

**}**

**Node temp = head;**

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

**temp = temp.next;**

**}**

**if (temp.next != null) {**

**temp.next = temp.next.next;**

**}**

**}**

**// Search by value**

**public boolean search(int data) {**

**Node temp = head;**

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

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

**return true;**

**}**

**temp = temp.next;**

**}**

**return false;**

**}**

**// Display the list**

**public void display() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**SinglyLinkedList list = new SinglyLinkedList();**

**// Test Case 1**

**list.insert(3);**

**list.insert(7);**

**list.insert(5);**

**list.delete(7);**

**list.display(); // Output: List = [3, 5]**

**System.*out*.println("Found = " + list.search(5)); // Output: Found = True**

**// Test Case 2**

**list = new SinglyLinkedList();**

**list.insert(9);**

**list.insert(4);**

**list.delete(4);**

**list.display(); // Output: List = [9]**

**System.*out*.println("Found = " + list.search(10)); // Output: Found = False**

**}**

**}**

**Output:**

**List = [3, 5]**

**Found = true**

**List = [9]**

**Found = false**

**Explanation:**

* **Insert: Adds a node with the specified data at the end of the list.**
* **Delete: Removes the node containing the given value if found.**
* **Search: Looks for the value in the list and returns true if found, otherwise false.**
* **Display: Prints the current elements of the list in a formatted way.**

**Time Complexity:**

* **Insert: O(n) for inserting at the end (n is the number of nodes).**
* **Delete: O(n) for searching and deleting a node.**
* **Search: O(n) for traversing the list.**
* **Display: O(n) for printing all elements.**

**Space Complexity:**

* **O(n) due to the space required for storing nodes.**

**2. Reverse a singly linked list.**

* **Test Case 1:  
  Input: List = [1, 2, 3, 4, 5]  
  Output: List = [5, 4, 3, 2, 1]**
* **Test Case 2:  
  Input: List = [10, 20, 30]  
  Output: List = [30, 20, 10]**

**Program code:**

**package Assignment.org;**

**class SinglyLinkedListReverse {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Reverse the linked list**

**public void reverse() {**

**Node prev = null;**

**Node current = head;**

**Node next = null;**

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

**next = current.next; // Store next node**

**current.next = prev; // Reverse the current node's pointer**

**prev = current; // Move prev to current**

**current = next; // Move to next node**

**}**

**head = prev; // Update the head to the new first node**

**}**

**// Display the list**

**public void display() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**SinglyLinkedListReverse list = new SinglyLinkedListReverse();**

**// Test Case 1**

**list.insert(1);**

**list.insert(2);**

**list.insert(3);**

**list.insert(4);**

**list.insert(5);**

**System.*out*.print("Original ");**

**list.display(); // Output: List = [1, 2, 3, 4, 5]**

**list.reverse();**

**System.*out*.print("Reversed ");**

**list.display(); // Output: List = [5, 4, 3, 2, 1]**

**// Test Case 2**

**list = new SinglyLinkedListReverse();**

**list.insert(10);**

**list.insert(20);**

**list.insert(30);**

**System.*out*.print("Original ");**

**list.display(); // Output: List = [10, 20, 30]**

**list.reverse();**

**System.*out*.print("Reversed ");**

**list.display(); // Output: List = [30, 20, 10]**

**}**

**}**

**Output:**

**Original List = [1, 2, 3, 4, 5]**

**Reversed List = [5, 4, 3, 2, 1]**

**Original List = [10, 20, 30]**

**Reversed List = [30, 20, 10]**

**Explanation:**

* **Insert: Adds a node to the end of the list.**
* **Reverse: Reverses the linked list by changing the direction of the next pointers of the nodes. Starting from the head, it traverses through each node and reverses the link, maintaining the previous node for reference.**
* **Display: Prints the current elements of the list after reversing.**

**Time Complexity:**

* **Insert: O(n) for inserting at the end.**
* **Reverse: O(n) since we traverse the entire list once to reverse it.**
* **Display: O(n) to print the elements.**

**Space Complexity:**

* **O(1) for reversing the list (only a few extra pointers are used).**
* **O(n) for storing the list nodes.**

**3. Detect a cycle in a linked list.**

* **Test Case 1:  
  Input: List = [1 → 2 → 3 → 4 → 5 → 3 (cycle)]  
  Output: Cycle Detected**
* **Test Case 2:  
  Input: List = [6 → 7 → 8 → 9]  
  Output: No Cycle**

**Program code:**

**package Assignment.org;**

**class SinglyLinkedListCycleDetection {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Create a cycle in the linked list for testing**

**public void createCycle(int pos) {**

**if (head == null) return;**

**Node temp = head;**

**Node cycleNode = null;**

**int index = 0;**

**while (temp.next != null) {**

**if (index == pos) {**

**cycleNode = temp;**

**}**

**temp = temp.next;**

**index++;**

**}**

**temp.next = cycleNode; // Creating a cycle**

**}**

**public boolean detectCycle() {**

**Node slow = head;**

**Node fast = head;**

**while (fast != null && fast.next != null) {**

**slow = slow.next; // Move slow pointer one step**

**fast = fast.next.next; // Move fast pointer two steps**

**if (slow == fast) { // Cycle detected**

**return true;**

**}**

**}**

**return false; // No cycle**

**}**

**// Display the list (without handling cycle display)**

**public void display() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(" → ");**

**temp = temp.next;**

**}**

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

**}**

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

**SinglyLinkedListCycleDetection list = new SinglyLinkedListCycleDetection();**

**// Test Case 1: Cycle exists**

**list.insert(1);**

**list.insert(2);**

**list.insert(3);**

**list.insert(4);**

**list.insert(5);**

**list.createCycle(2); // Creating a cycle: 5 → 3**

**if (list.detectCycle()) {**

**System.*out*.println("Cycle Detected");**

**} else {**

**System.*out*.println("No Cycle");**

**}**

**// Test Case 2: No cycle**

**list = new SinglyLinkedListCycleDetection();**

**list.insert(6);**

**list.insert(7);**

**list.insert(8);**

**list.insert(9);**

**if (list.detectCycle()) {**

**System.*out*.println("Cycle Detected");**

**} else {**

**System.*out*.println("No Cycle");**

**}**

**}**

**}**

**Output:**

**Cycle Detected**

**No Cycle**

**Explanation:**

* **Insert: Adds a node to the end of the list.**
* **Cycle Creation: For testing, we create a cycle manually by connecting the last node to an earlier node.**
* **Cycle Detection: Uses Floyd’s cycle detection algorithm, also known as the "Tortoise and Hare" method. If slow and fast pointers meet at some point, there's a cycle. If fast reaches the end (null), there's no cycle.**
* **Display: Displays the list without handling cycles (used only for normal lists).**

**Time Complexity:**

* **Insert: O(n) for inserting at the end.**
* **Cycle Detection: O(n) because both pointers traverse the list, with fast pointer covering the list twice as fast.**
* **Display: O(n) (but not safe for cyclic lists).**

**Space Complexity:**

* **O(1) for detecting the cycle since it uses constant space for pointers.**
* **O(n) for storing the list nodes.**

**4. Merge two sorted linked lists.**

* **Test Case 1:  
  Input: List1 = [1, 3, 5], List2 = [2, 4, 6]  
  Output: Merged List = [1, 2, 3, 4, 5, 6]**
* **Test Case 2:  
  Input: List1 = [10, 15, 20], List2 = [12, 18, 25]  
  Output: Merged List = [10, 12, 15, 18, 20, 25]**

**Program code:**

**package Assignment.org;**

**class SinglyLinkedListMerge {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Merge two sorted linked lists**

**public Node mergeLists(Node head1, Node head2) {**

**if (head1 == null) return head2;**

**if (head2 == null) return head1;**

**if (head1.data < head2.data) {**

**head1.next = mergeLists(head1.next, head2);**

**return head1;**

**} else {**

**head2.next = mergeLists(head1, head2.next);**

**return head2;**

**}**

**}**

**// Display the list**

**public void display(Node head) {**

**Node temp = head;**

**System.*out*.print("Merged List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**SinglyLinkedListMerge list1 = new SinglyLinkedListMerge();**

**SinglyLinkedListMerge list2 = new SinglyLinkedListMerge();**

**// Test Case 1**

**list1.insert(1);**

**list1.insert(3);**

**list1.insert(5);**

**list2.insert(2);**

**list2.insert(4);**

**list2.insert(6);**

**Node mergedHead = list1.mergeLists(list1.head, list2.head);**

**list1.display(mergedHead); // Output: Merged List = [1, 2, 3, 4, 5, 6]**

**// Test Case 2**

**list1 = new SinglyLinkedListMerge();**

**list2 = new SinglyLinkedListMerge();**

**list1.insert(10);**

**list1.insert(15);**

**list1.insert(20);**

**list2.insert(12);**

**list2.insert(18);**

**list2.insert(25);**

**mergedHead = list1.mergeLists(list1.head, list2.head);**

**list1.display(mergedHead); // Output: Merged List = [10, 12, 15, 18, 20, 25]**

**}**

**}**

**Output:**

**Merged List = [1, 2, 3, 4, 5, 6]**

**Merged List = [10, 12, 15, 18, 20, 25]**

**Explanation:**

* **Insert: Adds a node to the end of the list.**
* **Merge: This function recursively merges two sorted lists by comparing their nodes. It attaches the smaller node first and continues until one list is exhausted.**
* **Display: Prints the elements of the merged linked list in sorted order.**

**Time Complexity:**

* **Insert: O(n) for inserting at the end.**
* **Merge: O(m + n), where m and n are the lengths of the two linked lists. Each node is processed once.**
* **Display: O(m + n) for printing the merged list.**

**Space Complexity:**

* **O(m + n) due to the recursive calls on each node.**

**5. Find the nth node from the end of a linked list.**

* **Test Case 1:  
  Input: List = [10, 20, 30, 40, 50], n = 2  
  Output: 40**
* **Test Case 2:  
  Input: List = [5, 15, 25, 35], n = 4  
  Output: 5**

**Program code:**

**package Assignment.org;**

**class ListNthFromEnd {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Find the nth node from the end**

**public int findNthFromEnd(int n) {**

**Node first = head;**

**Node second = head;**

**int count = 0;**

**// Move first pointer `n` steps ahead**

**while (count < n) {**

**if (first == null) return -1; // n is larger than the number of nodes**

**first = first.next;**

**count++;**

**}**

**// Move both pointers until first reaches the end**

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

**first = first.next;**

**second = second.next;**

**}**

**return second.data; // Second is at the nth node from the end**

**}**

**// Display the list**

**public void display() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**ListNthFromEnd list = new ListNthFromEnd();**

**// Test Case 1**

**list.insert(10);**

**list.insert(20);**

**list.insert(30);**

**list.insert(40);**

**list.insert(50);**

**System.*out*.print("Original ");**

**list.display(); // Output: List = [10, 20, 30, 40, 50]**

**int nthNode = list.findNthFromEnd(2);**

**System.*out*.println("Nth node from end: " + nthNode); // Output: 40**

**// Test Case 2**

**list = new ListNthFromEnd();**

**list.insert(5);**

**list.insert(15);**

**list.insert(25);**

**list.insert(35);**

**System.*out*.print("Original ");**

**list.display(); // Output: List = [5, 15, 25, 35]**

**nthNode = list.findNthFromEnd(4);**

**System.*out*.println("Nth node from end: " + nthNode); // Output: 5**

**}**

**}**

**Output:**

**Original List = [10, 20, 30, 40, 50]**

**Nth node from end: 40**

**Original List = [5, 15, 25, 35]**

**Nth node from end: 5**

**Explanation:**

* **Insert: Adds a node to the end of the list.**
* **Find nth Node: Two pointers (first and second) are used. The first pointer moves n steps ahead, then both move one step at a time until the first reaches the end. The second pointer will be at the nth node from the end.**
* **Display: Prints the elements of the list.**

**Time Complexity:**

* **Insert: O(n) for inserting at the end.**
* **Find nth Node: O(n), as the list is traversed once.**
* **Display: O(n) to print the elements.**

**Space Complexity:**

* **O(1), as only two pointers are used.**

**6. Remove duplicates from a sorted linked list.**

* **Test Case 1:  
  Input: List = [1, 1, 2, 3, 3, 4]  
  Output: List = [1, 2, 3, 4]**
* **Test Case 2:  
  Input: List = [7, 7, 8, 9, 9, 10]  
  Output: List = [7, 8, 9, 10]**

**Program code:**

**package Assignment.org;**

**class SinglyLinkedListRemoveDuplicates {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Remove duplicates from a sorted linked list**

**public void removeDuplicates() {**

**Node current = head;**

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

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

**current.next = current.next.next; // Skip the duplicate node**

**} else {**

**current = current.next; // Move to the next node**

**}**

**}**

**}**

**// Display the list**

**public void display() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**SinglyLinkedListRemoveDuplicates list = new SinglyLinkedListRemoveDuplicates();**

**// Test Case 1**

**list.insert(1);**

**list.insert(1);**

**list.insert(2);**

**list.insert(3);**

**list.insert(3);**

**list.insert(4);**

**System.*out*.print("Original ");**

**list.display(); // Output: List = [1, 1, 2, 3, 3, 4]**

**list.removeDuplicates();**

**System.*out*.print("After Removing Duplicates ");**

**list.display(); // Output: List = [1, 2, 3, 4]**

**// Test Case 2**

**list = new SinglyLinkedListRemoveDuplicates();**

**list.insert(7);**

**list.insert(7);**

**list.insert(8);**

**list.insert(9);**

**list.insert(9);**

**list.insert(10);**

**System.*out*.print("Original ");**

**list.display(); // Output: List = [7, 7, 8, 9, 9, 10]**

**list.removeDuplicates();**

**System.*out*.print("After Removing Duplicates ");**

**list.display(); // Output: List = [7, 8, 9, 10]**

**}**

**}**

**Output:**

**Original List = [1, 1, 2, 3, 3, 4]**

**After Removing Duplicates List = [1, 2, 3, 4]**

**Original List = [7, 7, 8, 9, 9, 10]**

**After Removing Duplicates List = [7, 8, 9, 10]**

**Explanation:**

* **Insert: Adds a node to the end of the list.**
* **Remove Duplicates: Traverses the list and removes consecutive duplicate nodes by skipping over them.**
* **Display: Prints the elements of the list after removing duplicates.**

**Time Complexity:**

* **Insert: O(n) for inserting at the end.**
* **Remove Duplicates: O(n) since each node is processed once.**
* **Display: O(n) for printing the list.**

**Space Complexity:**

* **O(1), since no additional space is used except for a few pointers.**

**7. Implement a doubly linked list with insert, delete, and traverse operations.**

* **Test Case 1:  
  Input: Insert 10 → Insert 20 → Insert 30 → Delete 20  
  Output: List = [10, 30]**
* **Test Case 2:  
  Input: Insert 1 → Insert 2 → Insert 3 → Delete 1  
  Output: List = [2, 3]**

**Program code:**

**package Assignment.org;**

**class DoublyLinkedList {**

**class Node {**

**int data;**

**Node prev, next;**

**Node(int data) {**

**this.data = data;**

**this.prev = null;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Delete a node by value**

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

**Node temp = head;**

**if (temp == null) return;**

**// If the head is the node to be deleted**

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

**head = temp.next;**

**if (head != null) head.prev = null;**

**return;**

**}**

**// Traverse to find the node to delete**

**while (temp != null && temp.data != data) {**

**temp = temp.next;**

**}**

**// If the node is found, delete it**

**if (temp != null) {**

**if (temp.next != null) {**

**temp.next.prev = temp.prev;**

**}**

**if (temp.prev != null) {**

**temp.prev.next = temp.next;**

**}**

**}**

**}**

**// Traverse the list**

**public void traverse() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**DoublyLinkedList list = new DoublyLinkedList();**

**// Test Case 1**

**list.insert(10);**

**list.insert(20);**

**list.insert(30);**

**list.traverse(); // Output: List = [10, 20, 30]**

**list.delete(20);**

**list.traverse(); // Output: List = [10, 30]**

**// Test Case 2**

**list = new DoublyLinkedList();**

**list.insert(1);**

**list.insert(2);**

**list.insert(3);**

**list.traverse(); // Output: List = [1, 2, 3]**

**list.delete(1);**

**list.traverse(); // Output: List = [2, 3]**

**}**

**}**

**Output:**

**List = [10, 20, 30]**

**List = [10, 30]**

**List = [1, 2, 3]**

**List = [2, 3]**

**8. Reverse a doubly linked list.**

* **Test Case 1:  
  Input: List = [5, 10, 15, 20]  
  Output: List = [20, 15, 10, 5]**
* **Test Case 2:  
  Input: List = [4, 8, 12]  
  Output: List = [12, 8, 4]**

**Program code:**

**package Assignment.org;**

**class ReverseDoublylinkedList{**

**class Node {**

**int data;**

**Node prev, next;**

**Node(int data) {**

**this.data = data;**

**this.prev = null;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert at the end**

**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;**

**}**

**}**

**// Reverse the list**

**public void reverse() {**

**Node temp = null;**

**Node current = head;**

**// Swap the next and prev pointers for all nodes**

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

**temp = current.prev;**

**current.prev = current.next;**

**current.next = temp;**

**current = current.prev;**

**}**

**// Adjust head to the last element**

**if (temp != null) {**

**head = temp.prev;**

**}**

**}**

**// Traverse the list**

**public void traverse() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**ReverseDoublylinkedList list = new ReverseDoublylinkedList();**

**// Test Case 1**

**list.insert(5);**

**list.insert(10);**

**list.insert(15);**

**list.insert(20);**

**list.traverse(); // Output: List = [5, 10, 15, 20]**

**list.reverse();**

**list.traverse(); // Output: List = [20, 15, 10, 5]**

**// Test Case 2**

**list = new ReverseDoublylinkedList();**

**list.insert(4);**

**list.insert(8);**

**list.insert(12);**

**list.traverse(); // Output: List = [4, 8, 12]**

**list.reverse();**

**list.traverse(); // Output: List = [12, 8, 4]**

**}**

**}**

**Output:**

**List = [5, 10, 15, 20]**

**List = [20, 15, 10, 5]**

**List = [4, 8, 12]**

**List = [12, 8, 4]**

**9. Add two numbers represented by linked lists.**

* **Test Case 1:  
  Input: List1 = [2 → 4 → 3], List2 = [5 → 6 → 4] (243 + 465)  
  Output: Sum List = [7 → 0 → 8]**
* **Test Case 2:  
  Input: List1 = [9 → 9 → 9], List2 = [1] (999 + 1)  
  Output: Sum List = [0 → 0 → 0 → 1]**

**Program code:**

**package Assignment.org;**

**class AddTwoNumbers {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head1 = null, head2 = null, result = null;**

**// Insert into the first list**

**public void insertList1(int data) {**

**head1 = insert(head1, data);**

**}**

**// Insert into the second list**

**public void insertList2(int data) {**

**head2 = insert(head2, data);**

**}**

**// Insert a new node at the end of the given list**

**private Node insert(Node head, int data) {**

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

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

**return newNode;**

**} else {**

**Node temp = head;**

**while (temp.next != null) {**

**temp = temp.next;**

**}**

**temp.next = newNode;**

**}**

**return head;**

**}**

**// Add two numbers represented by the two lists**

**public void addTwoLists() {**

**Node l1 = head1, l2 = head2;**

**int carry = 0;**

**while (l1 != null || l2 != null) {**

**int sum = carry + (l1 != null ? l1.data : 0) + (l2 != null ? l2.data : 0);**

**carry = sum / 10;**

**result = insert(result, sum % 10);**

**if (l1 != null) l1 = l1.next;**

**if (l2 != null) l2 = l2.next;**

**}**

**// If there's any carry left**

**if (carry > 0) {**

**result = insert(result, carry);**

**}**

**}**

**// Traverse and print a list**

**public void traverseResult() {**

**Node temp = result;**

**System.*out*.print("Sum List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(" → ");**

**temp = temp.next;**

**}**

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

**}**

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

**AddTwoNumbers obj = new AddTwoNumbers();**

**// Test Case 1: List1 = [2 → 4 → 3], List2 = [5 → 6 → 4]**

**obj.insertList1(2);**

**obj.insertList1(4);**

**obj.insertList1(3);**

**obj.insertList2(5);**

**obj.insertList2(6);**

**obj.insertList2(4);**

**obj.addTwoLists();**

**obj.traverseResult(); // Output: Sum List = [7 → 0 → 8]**

**// Test Case 2: List1 = [9 → 9 → 9], List2 = [1]**

**obj = new AddTwoNumbers();**

**obj.insertList1(9);**

**obj.insertList1(9);**

**obj.insertList1(9);**

**obj.insertList2(1);**

**obj.addTwoLists();**

**obj.traverseResult(); // Output: Sum List = [0 → 0 → 0 → 1]**

**}**

**}**

**Output:**

**Sum List = [7 → 0 → 8]**

**Sum List = [0 → 0 → 0 → 1]**

**10. Rotate a linked list by k places.**

* **Test Case 1:  
  Input: List = [10, 20, 30, 40, 50], k = 2  
  Output: List = [30, 40, 50, 10, 20]**
* **Test Case 2:  
  Input: List = [5, 10, 15, 20], k = 3  
  Output: List = [20, 5, 10, 15]**

**Program code:**

**package Assignment.org;**

**class RotateLinkedList {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head = null;**

**// Insert a new node at the end of the list**

**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;**

**}**

**}**

**// Rotate the list by k places**

**public void rotate(int k) {**

**if (head == null || k == 0) return;**

**Node current = head;**

**int length = 1;**

**// Find the length of the list**

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

**current = current.next;**

**length++;**

**}**

**// Make the list circular**

**current.next = head;**

**// Find the new head after rotating k places**

**k = k % length;**

**int skip = length - k;**

**current = head;**

**for (int i = 1; i < skip; i++) {**

**current = current.next;**

**}**

**// Set the new head and break the circular link**

**head = current.next;**

**current.next = null;**

**}**

**// Traverse and print the list**

**public void traverse() {**

**Node temp = head;**

**System.*out*.print("List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(", ");**

**temp = temp.next;**

**}**

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

**}**

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

**RotateLinkedList list = new RotateLinkedList();**

**// Test Case 1: List = [10, 20, 30, 40, 50], k = 2**

**list.insert(10);**

**list.insert(20);**

**list.insert(30);**

**list.insert(40);**

**list.insert(50);**

**System.*out*.print("Original ");**

**list.traverse();**

**list.rotate(2);**

**System.*out*.print("After Rotation ");**

**list.traverse(); // Output: List = [30, 40, 50, 10, 20]**

**// Test Case 2: List = [5, 10, 15, 20], k = 3**

**list = new RotateLinkedList();**

**list.insert(5);**

**list.insert(10);**

**list.insert(15);**

**list.insert(20);**

**System.*out*.print("Original ");**

**list.traverse();**

**list.rotate(3);**

**System.*out*.print("After Rotation ");**

**list.traverse(); // Output: List = [20, 5, 10, 15]**

**}**

**}**

**Output:**

**Original List = [10, 20, 30, 40, 50]**

**After Rotation List = [40, 50, 10, 20, 30]**

**Original List = [5, 10, 15, 20]**

**After Rotation List = [10, 15, 20, 5]**

**11. Flatten a multilevel doubly linked list.**

* **Test Case 1:  
  Input: List = [1 → 2 → 3, 3 → 7 → 8, 8 → 10 → 12]  
  Output: Flattened List = [1 → 2 → 3 → 7 → 8 → 10 → 12]**
* **Test Case 2:  
  Input: List = [1 → 2 → 3, 2 → 5 → 6, 6 → 7 → 9]  
  Output: Flattened List = [1 → 2 → 5 → 6 → 7 → 9 → 3]**

**Program code:**

**package Assignment.org;**

**class FlattenDoublyLinkedList {**

**class Node {**

**int data;**

**Node next, child;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**this.child = null;**

**}**

**}**

**private Node head;**

**// Insert a node at the end of the main list**

**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;**

**}**

**}**

**// Flatten a multilevel doubly linked list**

**public Node flatten(Node head) {**

**if (head == null) return null;**

**Node current = head;**

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

**if (current.child != null) {**

**Node temp = current.child;**

**while (temp.next != null) {**

**temp = temp.next;**

**}**

**temp.next = current.next;**

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

**current.next = temp;**

**}**

**current.next = current.child;**

**current.child = null;**

**}**

**current = current.next;**

**}**

**return head;**

**}**

**// Traverse and print the list**

**public void traverse(Node head) {**

**Node temp = head;**

**System.*out*.print("Flattened List = [");**

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

**System.*out*.print(temp.data);**

**if (temp.next != null) System.*out*.print(" → ");**

**temp = temp.next;**

**}**

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

**}**

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

**FlattenDoublyLinkedList list = new FlattenDoublyLinkedList();**

**// Test Case 1: List = [1 → 2 → 3, 3 → 7 → 8, 8 → 10 → 12]**

**list.head = list.new Node(1);**

**list.head.next = list.new Node(2);**

**list.head.next.next = list.new Node(3);**

**list.head.next.next.child = list.new Node(7);**

**list.head.next.next.child.next = list.new Node(8);**

**list.head.next.next.child.next.child = list.new Node(10);**

**list.head.next.next.child.next.child.next = list.new Node(12);**

**list.traverse(list.flatten(list.head)); // Output: Flattened List = [1 → 2 → 3 → 7 → 8 → 10 → 12]**

**// Test Case 2: List = [1 → 2 → 3, 2 → 5 → 6, 6 → 7 → 9]**

**list = new FlattenDoublyLinkedList();**

**list.head = list.new Node(1);**

**list.head.next = list.new Node(2);**

**list.head.next.next = list.new Node(3);**

**list.head.next.child = list.new Node(5);**

**list.head.next.child.next = list.new Node(6);**

**list.head.next.child.next.child = list.new Node(7);**

**list.head.next.child.next.child.next = list.new Node(9);**

**list.traverse(list.flatten(list.head)); // Output: Flattened List = [1 → 2 → 5 → 6 → 7 → 9 → 3]**

**}**

**}**

**Output:**

**Flattened List = [1 → 2 → 3 → 7 → 8 → 10 → 12]**

**Flattened List = [1 → 2 → 5 → 6 → 7 → 9 → 3]**

**12. Split a circular linked list into two halves.**

* **Test Case 1:  
  Input: Circular List = [1 → 2 → 3 → 4 → 5 → 6 → (back to 1)]  
  Output: List1 = [1 → 2 → 3], List2 = [4 → 5 → 6]**
* **Test Case 2:  
  Input: Circular List = [10 → 20 → 30 → 40 → (back to 10)]  
  Output: List1 = [10 → 20], List2 = [30 → 40]**

**Program code:**

**package Assignment.org;**

**class CircularLinkedList {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head;**

**// Insert a node at the end of the circular linked list**

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

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

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

**head = newNode;**

**head.next = head; // Points to itself (circular)**

**} else {**

**Node temp = head;**

**while (temp.next != head) {**

**temp = temp.next;**

**}**

**temp.next = newNode;**

**newNode.next = head; // Make it circular**

**}**

**}**

**// Split the circular linked list into two halves**

**public void splitList() {**

**if (head == null || head.next == head) {**

**return;**

**}**

**Node slow = head;**

**Node fast = head;**

**// Using slow and fast pointers to find the midpoint**

**while (fast.next != head && fast.next.next != head) {**

**slow = slow.next;**

**fast = fast.next.next;**

**}**

**// For even number of nodes**

**if (fast.next.next == head) {**

**fast = fast.next;**

**}**

**// First half**

**Node head1 = head;**

**// Second half**

**Node head2 = slow.next;**

**// Make first half circular**

**slow.next = head;**

**// Make second half circular**

**fast.next = head2;**

**// Display the two halves**

**System.*out*.print("First Half: ");**

**traverse(head1);**

**System.*out*.print("Second Half: ");**

**traverse(head2);**

**}**

**// Traverse and print the circular linked list**

**public void traverse(Node head) {**

**if (head == null) return;**

**Node temp = head;**

**do {**

**System.*out*.print(temp.data + " ");**

**temp = temp.next;**

**} while (temp != head);**

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

**}**

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

**CircularLinkedList list = new CircularLinkedList();**

**// Test Case 1: Circular List = [1 → 2 → 3 → 4 → 5 → 6 → (back to 1)]**

**list.insert(1);**

**list.insert(2);**

**list.insert(3);**

**list.insert(4);**

**list.insert(5);**

**list.insert(6);**

**list.splitList(); // Output: First Half = [1 2 3], Second Half = [4 5 6]**

**// Test Case 2: Circular List = [10 → 20 → 30 → 40 → (back to 10)]**

**list = new CircularLinkedList();**

**list.insert(10);**

**list.insert(20);**

**list.insert(30);**

**list.insert(40);**

**list.splitList(); // Output: First Half = [10 20], Second Half = [30 40]**

**}**

**}**

**Output:**

**First Half: 1 2 3**

**Second Half: 4 5 6**

**First Half: 10 20**

**Second Half: 30 40**

**13. Insert a node in a sorted circular linked list.**

* **Test Case 1:  
  Input: Circular List = [10 → 20 → 30 → 40 → (back to 10)], Insert 25  
  Output: Circular List = [10 → 20 → 25 → 30 → 40 → (back to 10)]**
* **Test Case 2:  
  Input: Circular List = [5 → 15 → 25 → (back to 5)], Insert 10  
  Output: Circular List = [5 → 10 → 15 → 25 → (back to 5)]**

**Program code:**

**package Assignment.org;**

**class SortedCList {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head;**

**// Insert a node in a sorted circular linked list**

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

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

**// Case 1: List is empty**

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

**head = newNode;**

**newNode.next = head; // Make it circular**

**}**

**// Case 2: Insert before the head**

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

**Node temp = head;**

**while (temp.next != head) { // Traverse to the last node**

**temp = temp.next;**

**}**

**newNode.next = head;**

**temp.next = newNode;**

**head = newNode; // Update the head to the new node**

**}**

**// Case 3: Insert at the correct sorted position**

**else {**

**Node current = head;**

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

**current = current.next;**

**}**

**newNode.next = current.next;**

**current.next = newNode;**

**}**

**}**

**// Traverse and print the circular linked list**

**public void traverse() {**

**if (head == null) return;**

**Node temp = head;**

**do {**

**System.*out*.print(temp.data + " ");**

**temp = temp.next;**

**} while (temp != head);**

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

**}**

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

**SortedCList list = new SortedCList();**

**// Test Case 1: Circular List = [10 → 20 → 30 → 40 → (back to 10)], Insert 25**

**list.insert(10);**

**list.insert(20);**

**list.insert(30);**

**list.insert(40);**

**list.insert(25);**

**list.traverse(); // Output: Circular List = [10 20 25 30 40]**

**// Test Case 2: Circular List = [5 → 15 → 25 → (back to 5)], Insert 10**

**list = new SortedCList();**

**list.insert(5);**

**list.insert(15);**

**list.insert(25);**

**list.insert(10);**

**list.traverse(); // Output: Circular List = [5 10 15 25]**

**}**

**}**

**Output:**

**10 20 25 30 40**

**5 10 15 25**

**14. Check if two linked lists intersect, and find the intersection point if they do.**

* **Test Case 1:  
  Input: List1 = [1 → 2 → 3 → 4 → 5], List2 = [6 → 7 → 4 → 5]  
  Output: Intersection Point = 4**
* **Test Case 2:  
  Input: List1 = [10 → 20 → 30 → 40], List2 = [15 → 25 → 35]  
  Output: No Intersection**

**Program code:**

**package Assignment.org;**

**class LinkedListIntersection {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head;**

**// Add nodes to the list**

**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;**

**}**

**}**

**// Get the head of the list**

**public Node getHead() {**

**return head;**

**}**

**// Find the intersection point of two linked lists**

**public static Node getIntersectionNode(Node head1, Node head2) {**

**int length1 = *getLength*(head1);**

**int length2 = *getLength*(head2);**

**// Adjust starting point of the longer list**

**if (length1 > length2) {**

**head1 = *moveForward*(head1, length1 - length2);**

**} else {**

**head2 = *moveForward*(head2, length2 - length1);**

**}**

**// Traverse both lists together to find the intersection**

**while (head1 != null && head2 != null) {**

**if (head1 == head2) {**

**return head1; // Intersection point**

**}**

**head1 = head1.next;**

**head2 = head2.next;**

**}**

**return null; // No intersection**

**}**

**// Get the length of a linked list**

**private static int getLength(Node head) {**

**int length = 0;**

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

**length++;**

**head = head.next;**

**}**

**return length;**

**}**

**// Move forward by a given number of steps in a linked list**

**private static Node moveForward(Node head, int steps) {**

**while (steps > 0 && head != null) {**

**head = head.next;**

**steps--;**

**}**

**return head;**

**}**

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

**// Test Case 1: List1 = [1 → 2 → 3 → 4 → 5], List2 = [6 → 7 → 4 → 5], Intersection at 4**

**LinkedListIntersection list1 = new LinkedListIntersection();**

**list1.insert(1);**

**list1.insert(2);**

**list1.insert(3);**

**list1.insert(4);**

**list1.insert(5);**

**LinkedListIntersection list2 = new LinkedListIntersection();**

**list2.insert(6);**

**list2.insert(7);**

**list2.getHead().next.next = list1.getHead().next.next.next; // Creating intersection at node 4**

**Node intersection = *getIntersectionNode*(list1.getHead(), list2.getHead());**

**if (intersection != null) {**

**System.*out*.println("Intersection Point = " + intersection.data);**

**} else {**

**System.*out*.println("No Intersection");**

**}**

**// Test Case 2: List1 = [10 → 20 → 30 → 40], List2 = [15 → 25 → 35], No intersection**

**list1 = new LinkedListIntersection();**

**list1.insert(10);**

**list1.insert(20);**

**list1.insert(30);**

**list1.insert(40);**

**list2 = new LinkedListIntersection();**

**list2.insert(15);**

**list2.insert(25);**

**list2.insert(35);**

**intersection = *getIntersectionNode*(list1.getHead(), list2.getHead());**

**if (intersection != null) {**

**System.*out*.println("Intersection Point = " + intersection.data);**

**} else {**

**System.*out*.println("No Intersection");**

**}**

**}**

**}**

**Output:**

**Intersection Point = 4**

**No Intersection**

**15. Find the middle element of a linked list in one pass.**

* **Test Case 1:  
  Input: List = [1, 2, 3, 4, 5]  
  Output: Middle = 3**
* **Test Case 2:  
  Input: List = [11, 22, 33, 44, 55, 66]  
  Output: Middle = 44**

**Program code:**

**package Assignment.org;**

**class Middleelement {**

**class Node {**

**int data;**

**Node next;**

**Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**private Node head;**

**// Insert a node at the end of the list**

**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;**

**}**

**}**

**// Find and return the middle element of the list**

**public int findMiddle() {**

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

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

**}**

**Node slow = head;**

**Node fast = head;**

**// Fast pointer moves two steps at a time, slow pointer moves one step**

**while (fast != null && fast.next != null) {**

**slow = slow.next;**

**fast = fast.next.next;**

**}**

**return slow.data;**

**}**

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

**// Test Case 1: List = [1, 2, 3, 4, 5], Middle = 3**

**Middleelement list1 = new Middleelement();**

**list1.insert(1);**

**list1.insert(2);**

**list1.insert(3);**

**list1.insert(4);**

**list1.insert(5);**

**System.*out*.println("Middle = " + list1.findMiddle()); // Output: 3**

**// Test Case 2: List = [11, 22, 33, 44, 55, 66], Middle = 44**

**Middleelement list2 = new Middleelement();**

**list2.insert(11);**

**list2.insert(22);**

**list2.insert(33);**

**list2.insert(44);**

**list2.insert(55);**

**list2.insert(66);**

**System.*out*.println("Middle = " + list2.findMiddle()); // Output: 44**

**}**

**}**

**Output:**

**Middle = 3**

**Middle = 44**