**Assignment 4**

**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

class SinglyLinkedList {

// Node class to represent each node in the list

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head; // Head of the list

// Insert a new node 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 a node with the given value

public void delete(int data) {

if (head == null) return; // Empty list

if (head.data == data) {

head = head.next; // Delete head node

return;

}

Node temp = head;

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

temp = temp.next;

}

if (temp.next != null) {

temp.next = temp.next.next; // Remove the node

}

}

// Search for a node with the given value

public boolean search(int data) {

Node temp = head;

while (temp != null) {

if (temp.data == data) return true;

temp = temp.next;

}

return false;

}

// Print the list

public void printList() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(", ");

}

}

System.out.println("]");

}

// Test Cases

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.printList(); // Output: List = [3, 5]

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

// Test Case 2

list = new SinglyLinkedList(); // Create a new list

list.insert(9);

list.insert(4);

list.delete(4);

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

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

}

}

**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]

class SinglyLinkedList {

// Node class to represent each node in the list

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head; // Head of the list

// Insert a new node 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 singly linked list

public void reverse() {

Node previous = null;

Node current = head;

Node next = null;

while (current != null) {

next = current.next; // Store next node

current.next = previous; // Reverse the link

previous = current; // Move previous to current

current = next; // Move to the next node

}

head = previous; // Update head to the new first node

}

// Print the list

public void printList() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(", ");

}

}

System.out.println("]");

}

// Test Cases

public static void main(String[] args) {

// Test Case 1

SinglyLinkedList list1 = new SinglyLinkedList();

list1.insert(1);

list1.insert(2);

list1.insert(3);

list1.insert(4);

list1.insert(5);

System.out.print("Input: ");

list1.printList(); // Output: List = [1, 2, 3, 4, 5]

list1.reverse();

System.out.print("Output: ");

list1.printList(); // Output: List = [5, 4, 3, 2, 1]

// Test Case 2

SinglyLinkedList list2 = new SinglyLinkedList();

list2.insert(10);

list2.insert(20);

list2.insert(30);

System.out.print("Input: ");

list2.printList(); // Output: List = [10, 20, 30]

list2.reverse();

System.out.print("Output: ");

list2.printList(); // Output: List = [30, 20, 10]

}

}

**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

class SinglyLinkedList {

// Node class to represent each node in the list

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head; // Head of the list

// Insert a new node 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 position) {

if (head == null) return;

Node cycleNode = head;

for (int i = 1; i < position && cycleNode != null; i++) {

cycleNode = cycleNode.next;

}

if (cycleNode != null) {

Node lastNode = head;

while (lastNode.next != null) {

lastNode = lastNode.next;

}

lastNode.next = cycleNode; // Create the cycle

}

}

// Detect a cycle in the linked list using Floyd’s Tortoise and Hare algorithm

public boolean detectCycle() {

Node slow = head;

Node fast = head;

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

slow = slow.next; // Move slow pointer by 1

fast = fast.next.next; // Move fast pointer by 2

if (slow == fast) {

return true; // Cycle detected

}

}

return false; // No cycle

}

// Print the list (for visual purposes; will stop if a cycle is detected)

public void printList() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(" → ");

}

}

System.out.println("]");

}

// Test Cases

public static void main(String[] args) {

// Test Case 1: Cycle Detected

SinglyLinkedList list1 = new SinglyLinkedList();

list1.insert(1);

list1.insert(2);

list1.insert(3);

list1.insert(4);

list1.insert(5);

list1.createCycle(3); // Creates a cycle pointing to node with data 3

System.out.print("Input: ");

list1.printList(); // Will show list until it reaches the cycle

System.out.println("Output: " + (list1.detectCycle() ? "Cycle Detected" : "No Cycle"));

// Test Case 2: No Cycle

SinglyLinkedList list2 = new SinglyLinkedList();

list2.insert(6);

list2.insert(7);

list2.insert(8);

list2.insert(9);

System.out.print("Input: ");

list2.printList(); // Output: List = [6 → 7 → 8 → 9]

System.out.println("Output: " + (list2.detectCycle() ? "Cycle Detected" : "No Cycle"));

}

}

**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]

class SinglyLinkedList {

// Node class to represent each node in the list

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head; // Head of the list

// Insert a new node 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;

}

}

// Print the list

public void printList() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(", ");

}

}

System.out.println("]");

}

// Merge two sorted linked lists

public static SinglyLinkedList merge(SinglyLinkedList list1, SinglyLinkedList list2) {

SinglyLinkedList mergedList = new SinglyLinkedList();

Node current1 = list1.head;

Node current2 = list2.head;

while (current1 != null && current2 != null) {

if (current1.data <= current2.data) {

mergedList.insert(current1.data);

current1 = current1.next;

} else {

mergedList.insert(current2.data);

current2 = current2.next;

}

}

// Add remaining nodes from list1

while (current1 != null) {

mergedList.insert(current1.data);

current1 = current1.next;

}

// Add remaining nodes from list2

while (current2 != null) {

mergedList.insert(current2.data);

current2 = current2.next;

}

return mergedList; // Return the merged list

}

// Test Cases

public static void main(String[] args) {

// Test Case 1

SinglyLinkedList list1 = new SinglyLinkedList();

list1.insert(1);

list1.insert(3);

list1.insert(5);

SinglyLinkedList list2 = new SinglyLinkedList();

list2.insert(2);

list2.insert(4);

list2.insert(6);

SinglyLinkedList mergedList1 = merge(list1, list2);

System.out.print("Input: List1 = [1, 3, 5], List2 = [2, 4, 6] \n");

System.out.print("Output: Merged List = ");

mergedList1.printList(); // Output: Merged List = [1, 2, 3, 4, 5, 6]

// Test Case 2

SinglyLinkedList list3 = new SinglyLinkedList();

list3.insert(10);

list3.insert(15);

list3.insert(20);

SinglyLinkedList list4 = new SinglyLinkedList();

list4.insert(12);

list4.insert(18);

list4.insert(25);

SinglyLinkedList mergedList2 = merge(list3, list4);

System.out.print("Input: List1 = [10, 15, 20], List2 = [12, 18, 25] \n");

System.out.print("Output: Merged List = ");

mergedList2.printList(); // Output: Merged List = [10, 12, 15, 18, 20, 25]

}

}

**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

class SinglyLinkedList {

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

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;

}

}

public int findNthFromEnd(int n) {

Node mainPtr = head;

Node refPtr = head;

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

if (refPtr == null) {

return -1; // n is greater than the number of nodes

}

refPtr = refPtr.next;

}

while (refPtr != null) {

mainPtr = mainPtr.next;

refPtr = refPtr.next;

}

return mainPtr.data;

}

// Test Cases

public static void main(String[] args) {

SinglyLinkedList list = new SinglyLinkedList();

list.insert(10);

list.insert(20);

list.insert(30);

list.insert(40);

list.insert(50);

System.out.println("Output: " + list.findNthFromEnd(2));

SinglyLinkedList list2 = new SinglyLinkedList();

list2.insert(5);

list2.insert(15);

list2.insert(25);

list2.insert(35);

System.out.println("Output: " + list2.findNthFromEnd(4));

}

}

**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]

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

} else {

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

}

}

}

// Test Cases

public static void main(String[] args) {

SinglyLinkedList list = new SinglyLinkedList();

list.insert(1);

list.insert(1);

list.insert(2);

list.insert(3);

list.insert(3);

list.insert(4);

list.removeDuplicates();

System.out.print("Output: ");

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

SinglyLinkedList list2 = new SinglyLinkedList();

list2.insert(7);

list2.insert(7);

list2.insert(8);

list2.insert(9);

list2.insert(9);

list2.insert(10);

list2.removeDuplicates();

System.out.print("Output: ");

list2.printList(); // Output: List = [7, 8, 9, 10]

}

**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]

class DoublyLinkedList {

class Node {

int data;

Node next;

Node prev;

public Node(int data) {

this.data = data;

this.next = null;

this.prev = null;

}

}

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 delete(int data) {

Node temp = head;

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

temp = temp.next;

}

if (temp != null) {

if (temp.prev != null) {

temp.prev.next = temp.next;

} else {

head = temp.next; // Delete head

}

if (temp.next != null) {

temp.next.prev = temp.prev;

}

}

}

public void printList() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(", ");

}

}

System.out.println("]");

}

// Test Cases

public static void main(String[] args) {

DoublyLinkedList list = new DoublyLinkedList();

list.insert(10);

list.insert(20);

list.insert(30);

list.delete(20);

System.out.print("Output: ");

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

DoublyLinkedList list2 = new DoublyLinkedList();

list2.insert(1);

list2.insert(2);

list2.insert(3);

list2.delete(1);

System.out.print("Output: ");

list2.printList(); // Output: 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]

public void reverse() {

Node temp = null;

Node current = head;

while (current != null) {

temp = current.prev;

current.prev = current.next;

current.next = temp;

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

}

if (temp != null) {

head = temp.prev; // Update head to the new first node

}

}

// Test Cases

public static void main(String[] args) {

DoublyLinkedList list = new DoublyLinkedList();

list.insert(5);

list.insert(10);

list.insert(15);

list.insert(20);

list.reverse();

System.out.print("Output: ");

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

DoublyLinkedList list2 = new DoublyLinkedList();

list2.insert(4);

list2.insert(8);

list2.insert(12);

list2.reverse();

System.out.print("Output: ");

list2.printList(); // Output: 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]

public static SinglyLinkedList addTwoNumbers(SinglyLinkedList list1, SinglyLinkedList list2) {

SinglyLinkedList result = new SinglyLinkedList();

Node ptr1 = list1.head;

Node ptr2 = list2.head;

int carry = 0;

while (ptr1 != null || ptr2 != null || carry != 0) {

int sum = carry;

if (ptr1 != null) {

sum += ptr1.data;

ptr1 = ptr1.next;

}

if (ptr2 != null) {

sum += ptr2.data;

ptr2 = ptr2.next;

}

result.insert(sum % 10); // Insert last digit of sum

carry = sum / 10; // Update carry

}

return result;

}

// Test Cases

public static void main(String[] args) {

SinglyLinkedList list1 = new SinglyLinkedList();

list1.insert(2);

list1.insert(4);

list1.insert(3);

SinglyLinkedList list2 = new SinglyLinkedList();

list2.insert(5);

list2.insert(6);

list2.insert(4);

SinglyLinkedList sumList = addTwoNumbers(list1, list2);

System.out.print("Output: ");

sumList.printList();

SinglyLinkedList list3 = new SinglyLinkedList();

list3.insert(9);

list3.insert(9);

list3.insert(9);

SinglyLinkedList list4 = new SinglyLinkedList();

list4.insert(1);

SinglyLinkedList sumList2 = addTwoNumbers(list3, list4);

System.out.print("Output: ");

sumList2.printList();

}

**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]

public void rotate(int k) {

if (head == null || head.next == null || k == 0) return;

Node current = head;

int length = 1;

while (current.next != null) {

current = current.next;

length++;

}

current.next = head; // Make it circular

k = k % length; // In case k is greater than length

int skipLength = length - k;

Node lastNode = head;

for (int i = 1; i < skipLength; i++) {

lastNode = lastNode.next;

}

head = lastNode.next; // Update head

lastNode.next = null; // Break the circular link

}

// Test Cases

public static void main(String[] args) {

SinglyLinkedList list = new SinglyLinkedList();

list.insert(10);

list.insert(20);

list

**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]

class MultiLevelDoublyLinkedList {

class Node {

int data;

Node next;

Node down;

public Node(int data) {

this.data = data;

this.next = null;

this.down = null;

}

}

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;

}

}

public Node flatten(Node node) {

Node current = node;

Node tail = current;

while (current != null) {

if (current.down != null) {

Node downTail = flatten(current.down);

downTail.next = current.next;

current.next = current.down;

current.down = null; // Clear the down link

tail = downTail;

}

tail = current;

current = current.next;

}

return tail;

}

public void flatten() {

flatten(head);

}

public void printList() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(" → ");

}

}

System.out.println("]");

}

// Test Cases

public static void main(String[] args) {

MultiLevelDoublyLinkedList list = new MultiLevelDoublyLinkedList();

list.insert(1);

list.insert(2);

list.insert(3);

list.head.down = new MultiLevelDoublyLinkedList().new Node(7);

list.head.down.next = new MultiLevelDoublyLinkedList().new Node(8);

list.head.down.next.next = new MultiLevelDoublyLinkedList().new Node(10);

list.head.down.next.next.down = new MultiLevelDoublyLinkedList().new Node(12);

list.flatten();

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

MultiLevelDoublyLinkedList list2 = new MultiLevelDoublyLinkedList();

list2.insert(1);

list2.insert(2);

list2.insert(3);

list2.head.down = new MultiLevelDoublyLinkedList().new Node(5);

list2.head.down.next = new MultiLevelDoublyLinkedList().new Node(6);

list2.head.down.next.down = new MultiLevelDoublyLinkedList().new Node(7);

list2.head.down.next.down.next = new MultiLevelDoublyLinkedList().new Node(9);

list2.flatten();

list2.printList(); // Output: 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]

class CircularLinkedList {

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head;

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

newNode.next = head; // Point to itself

} else {

Node temp = head;

while (temp.next != head) {

temp = temp.next;

}

temp.next = newNode;

newNode.next = head; // Make it circular

}

}

public void split() {

if (head == null) return;

Node slow = head;

Node fast = head;

// Use the fast and slow pointer technique

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

slow = slow.next;

fast = fast.next.next;

}

// Now slow is at the end of the first half

Node head1 = head;

Node head2 = slow.next;

slow.next = head1; // End the first half

fast.next = head2; // End the second half

System.out.print("List1 = [");

printList(head1);

System.out.print("List2 = [");

printList(head2);

}

public void printList(Node start) {

Node temp = start;

while (temp.next != start) {

System.out.print(temp.data + " → ");

temp = temp.next;

}

System.out.print(temp.data + "]"); // Print last node

}

// Test Cases

public static void main(String[] args) {

CircularLinkedList list = new CircularLinkedList();

list.insert(1);

list.insert(2);

list.insert(3);

list.insert(4);

list.insert(5);

list.insert(6);

list.split(); // Output: List1 = [1 → 2 → 3] List2 = [4 → 5 → 6]

CircularLinkedList list2 = new CircularLinkedList();

list2.insert(10);

list2.insert(20);

list2.insert(30);

list2.insert(40);

list2.split(); // Output: List1 = [10 → 20] List2 = [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)]

public void insertInSortedOrder(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

newNode.next = head; // Point to itself

} else {

Node current = head;

Node prev = null;

do {

prev = current;

current = current.next;

} while (current != head && current.data < data);

prev.next = newNode;

newNode.next = current;

// If new node is inserted before head, update head

if (data < head.data) {

head = newNode;

}

}

}

// Test Cases

public static void main(String[] args) {

CircularLinkedList list = new CircularLinkedList();

list.insert(10);

list.insert(20);

list.insert(30);

list.insert(40);

list.insertInSortedOrder(25);

list.printList(list.head); // Output: Circular List = [10 → 20 → 25 → 30 → 40 → (back to 10)]

CircularLinkedList list2 = new CircularLinkedList();

list2.insert(5);

list2.insert(15);

list2.insert(25);

list2.insertInSortedOrder(10);

list2.printList(list2.head); // Output: Circular List = [5 → 10 → 15 → 25 → (back to 5)]

}

**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

public static Node findIntersection(Node head1, Node head2) {

Set<Node> nodesSet = new HashSet<>();

Node current = head1;

while (current != null) {

nodesSet.add(current);

current = current.next;

}

current = head2;

while (current != null) {

if (nodesSet.contains(current)) {

return current; // Intersection point

}

current = current.next;

}

return null; // No intersection

}

// Test Cases

public static void main(String[] args) {

SinglyLinkedList list1 = new SinglyLinkedList();

list1.insert(1);

list1.insert(2);

list1.insert(3);

list1.insert(4);

list1.insert(5);

SinglyLinkedList list2 = new SinglyLinkedList();

list2.insert(6);

list2.insert(7);

list2.head.next.next = list1.head.next; // Intersect at node with value 4

Node intersectionPoint = findIntersection(list1.head, list2.head);

if (intersectionPoint != null) {

System.out.println("Intersection Point = " + intersectionPoint.data); // Output: Intersection Point = 4

} else {

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

}

SinglyLinkedList list3 = new SinglyLinkedList();

list3.insert(10);

list3.insert(20);

list3.insert(30);

SinglyLinkedList list4 = new SinglyLinkedList();

list4.insert(15);

list4.insert(25);

list4.insert(35);

intersectionPoint = findIntersection(list3.head, list4.head);

if (intersectionPoint != null) {

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

} else {

System.out.println("No Intersection"); // Output: 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

class SinglyLinkedList {

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head;

// Method to insert a new node 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;

}

}

// Method to find the middle element in one pass

public int findMiddle() {

if (head == null) return -1; // Return -1 if the list is empty

Node slow = head;

Node fast = head;

// Move slow by one and fast by two

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

slow = slow.next;

fast = fast.next.next;

}

return slow.data; // Return the middle element

}

// Method to print the list (for verification)

public void printList() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(", ");

}

}

System.out.println("]");

}

// Test Cases

public static void main(String[] args) {

SinglyLinkedList list1 = new SinglyLinkedList();

list1.insert(1);

list1.insert(2);

list1.insert(3);

list1.insert(4);

list1.insert(5);

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

SinglyLinkedList list2 = new SinglyLinkedList();

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: Middle = 44

}

}