Assignment 4

1. Singly Linked List with Insert, Delete, Search Operations

Program:

```
class Node {
   int data;
   Node next;
   public Node(int data) {
      this.data = data;
      this.next = null;
   }
}
class SingleLinked {
   Node head;
   public void insert(int data) {
      Node newNode = new Node(data);
      if (head == null) {
         head = newNode;
      } else {
         Node current = head;
         while (current.next != null) {
            current = current.next;
         }
         current.next = newNode;
      }
   }
   public void delete(int data) {
      if (head == null) return;

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

      Node current = head;
      while (current.next != null && current.next.data != data) {
         current = current.next;
      }
      if (current.next != null) {
         current.next = current.next.next;
      }
   }
   public boolean search(int data) {
      Node current = head;
      while (current != null) {
         if (current.data == data) {
            return true;
         }
```

```
            current = current.next;
        }
        return false;
    }
    public void printList() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        SingleLinked list = new SingleLinked();
        list.insert(3);
        list.insert(7);
        list.insert(5);
        list.delete(7);
        list.printList();
        System.out.println(list.search(5));
        list = new SingleLinked();
        list.insert(9);
        list.insert(4);
        list.delete(4);
        list.printList();
        System.out.println(list.search(10));
    }
}
```

Time Complexity:

- Insert: O(n) (because we traverse to the end)
- Delete: O(n)
- Search: O(n)

Space Complexity:

- O(n) for storing the list.

2. Reverse a Singly Linked List

Program:

```
class ReverseSinglyLinkedList {
    Node head;
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
```

```java
            current.next = newNode;
        }
    }
    public void reverse() {
        Node prev = null, current = head, next;
        while (current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
        }
        head = prev;
    }
    public void printList() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        ReverseSinglyLinkedList list = new ReverseSinglyLinkedList();

        list.insert(1);
        list.insert(2);
        list.insert(3);
        list.insert(4);
        list.insert(5);
        list.reverse();
        list.printList();
        list = new ReverseSinglyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.reverse();
        list.printList();
    }
}
```

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

Test Case 2:
List = [30, 20, 10]
Time Complexity:

- O(n) for traversing and reversing the list.

Space Complexity:

- O(1) because we only use a few extra pointers.

3. Detect a Cycle in a Linked List

Program:

```
public class CycleDetection {
    Node head;

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

    boolean detectCycle() {
        Node slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) return true; // Cycle detected
        }
        return false; // No cycle
    }

    void createCycle(int position) {
        if (position < 0) return;
        Node temp = head, cycleNode = null;
        int index = 0;
        while (temp.next != null) {
            if (index == position) cycleNode = temp;
            temp = temp.next;
            index++;
        }
        temp.next = cycleNode; // Creating cycle
    }

    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 static void main(String[] args) {
        CycleDetection list = new CycleDetection();
        list.insert(1);
        list.insert(2);
```

```
        list.insert(3);
        list.insert(4);
        list.insert(5);
        list.createCycle(3);
        if (list.detectCycle()) {
            System.out.println("Cycle detected.");
        } else {
            System.out.println("No cycle detected.");
        }
    }
}
```

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
Time Complexity:

- O(n), where n is the number of nodes in the linked list.

Space Complexity:

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

4. Merge Two Sorted Linked Lists
Program:

```
class MergeSortedLists {
    static class Node {
        int data;
        Node next;
        Node(int data) { this.data = data; next = null; }
    }

    Node merge(Node list1, Node list2) {
        if (list1 == null) return list2;
        if (list2 == null) return list1;
        Node result;

        if (list1.data <= list2.data) {
            result = list1;
            result.next = merge(list1.next, list2);
        } else {
            result = list2;
            result.next = merge(list1, list2.next);
        }
        return result;
    }

    void printList(Node head) {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
```

```
            temp = temp.next;
        }
        System.out.println();
    }
}
```

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]

Time Complexity:

- O(n + m), where n and m are the lengths of the two lists.

Space Complexity:

- O(n + m), as each recursive call adds to the call stack (can be reduced to O(1) with an iterative approach).

5. Find the nth Node from the End of a Linked List

Program:

```
class NthNodeFromEnd {
    Node head;

    static class Node {
        int data;
        Node next;
        Node(int data) { this.data = data; next = null; }
    }
    int findNthFromEnd(int n) {
        Node mainPtr = head, refPtr = head;
        int count = 0;
          while (count < n) {
            if (refPtr == null) return -1; // n is larger than the number of nodes
            refPtr = refPtr.next;
            count++;
        }
        while (refPtr != null) {
            mainPtr = mainPtr.next;
            refPtr = refPtr.next;
        }
        return mainPtr.data;
    }

    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;
        }
```

```
    }
}
```
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
Time Complexity:

- O(n), where n is the length of the linked list.

Space Complexity:

- O(1), only constant space is used.

6. Remove Duplicates from a Sorted Linked List
Program:
```
class RemoveDuplicates {
    Node head;

    static class Node {
        int data;
        Node next;
        Node(int data) { this.data = data; next = null; }
    }
    void removeDuplicates() {
        Node current = head;
        while (current != null && current.next != null) {
            if (current.data == current.next.data) {
                current.next = current.next.next;
            } else {
                current = current.next;
            }
        }
    }

    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;
        }
    }

    void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
```

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

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]
Time Complexity:

- O(n), where n is the number of nodes in the list.

Space Complexity:

- O(1), constant space is used.

7. Implement a Doubly Linked List with Insert, Delete, and Traverse Operations
Program:

```
class DoublyLinkedList {
    Node head;

    static class Node {
        int data;
        Node prev, next;
        Node(int data) { this.data = data; prev = next = null; }
    }
    void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null) temp = temp.next;
        temp.next = newNode;
        newNode.prev = temp;
    }
    void delete(int data) {
        Node temp = head;
        while (temp != null && temp.data != data) temp = temp.next;
        if (temp == null) return; // Node not found
        if (temp.prev != null) temp.prev.next = temp.next;
        if (temp.next != null) temp.next.prev = temp.prev;
        if (temp == head) head = temp.next; // Deleting head node
    }

    void traverse() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
```

```
            temp = temp.next;
        }
        System.out.println();
    }
}
```

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]
Time Complexity:

- Insert: O(n)
- Delete: O(n)
- Traverse: O(n)

Space Complexity:

- O(1), constant space for each operation.

8. Reverse a Doubly Linked List
Program:

```
class ReverseDoublyLinkedList {
    Node head;

    static class Node {
        int data;
        Node prev, next;
        Node(int data) { this.data = data; prev = next = null; }
    }
        void reverse() {
        Node temp = null;
        Node current = head;
        while (current != null) {
            // Swap prev and next
            temp = current.prev;
            current.prev = current.next;
            current.next = temp;
            current = current.prev;        }
        if (temp != null) head = temp.prev; // New head after reversal
    }

    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;
    }
  }

  void traverse() {
    Node temp = head;
    while (temp != null) {
      System.out.print(temp.data + " ");
      temp = temp.next;
    }
    System.out.println();
  }
}
```

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]
Time Complexity:

- O(n), where n is the number of nodes.

Space Complexity:

- O(1), constant space.

9. Add Two Numbers Represented by Linked Lists
Program:

```
class AddTwoNumbers {
  static class Node {
    int data;
    Node next;
    Node(int data) { this.data = data; next = null; }
  }

  Node addTwoLists(Node list1, Node list2) {
    Node dummy = new Node(0); // Result list
    Node current = dummy;
    int carry = 0;

    while (list1 != null || list2 != null) {
      int sum = carry;
      if (list1 != null) {
        sum += list1.data;
        list1 = list1.next;
      }
      if (list2 != null) {
        sum += list2.data;
        list2 = list2.next;
      }
```

```
            carry = sum / 10;
            current.next = new Node(sum % 10);
            current = current.next;
        }

        if (carry > 0) current.next = new Node(carry);

        return dummy.next;
    }

    void printList(Node head) {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
```

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]
Time Complexity:

- O(n + m), where n and m are the lengths of the two lists.

Space Complexity:

- O(n + m), new list created for the result.

10. Rotate a Linked List by K Places
Program:

```
class RotateLinkedList {
    Node head;

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

    void rotate(int k) {
        if (head == null || k == 0) return;

        Node current = head;
        int length = 1;
            while (current.next != null) {
            current = current.next;
```

```
            length++;
        }
        current.next = head;
        k = k % length;
        int stepsToNewHead = length - k;
        Node newTail = head;
        for (int i = 1; i < stepsToNewHead; i++) {
            newTail = newTail.next;
        }
        head = newTail.next;
        newTail.next = null;
    }

    void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null) temp = temp.next;
        temp.next = newNode;
    }

    void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
```

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]
Time Complexity:

- O(n)

Space Complexity:

- O(1)

11. Flatten a Multilevel Doubly Linked List
Program:

```
class FlattenDoublyLinkedList {
    Node head;
```

```
    static class Node {
        int data;
        Node next, child;
        Node(int data) { this.data = data; next = child = null; }
    }

    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.child = temp;

                current.next = current.child;
                current.child = null;
            }
            current = current.next;
        }
        return head;
    }

    void insert(Node parent, int data) {
        Node newNode = new Node(data);
        if (parent.child == null) parent.child = newNode;
        else {
            Node temp = parent.child;
            while (temp.next != null) temp = temp.next;
            temp.next = newNode;
        }
    }

    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
        System.out.println();
    }
}
```

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]
Time Complexity:

- O(n)

Space Complexity:

- O(1)

12. Split a Circular Linked List into Two Halves
Program:

```
class SplitCircularLinkedList {
  Node head;

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

  void splitList() {
    if (head == null || head.next == head) return;

    Node slow = head;
    Node fast = head;

    // Find the middle of the circular linked list
    while (fast.next != head && fast.next.next != head) {
      slow = slow.next;
      fast = fast.next.next;
    }

    Node head1 = head;
    Node head2 = slow.next;

    slow.next = head1; // End first half
    Node temp = head2;
    while (temp.next != head) temp = temp.next;
    temp.next = head2; // End second half

    printList(head1);
    printList(head2);
  }

  void insert(int data) {
    Node newNode = new Node(data);
    if (head == null) {
      head = newNode;
      newNode.next = head;
      return;
```

```
        }
        Node temp = head;
        while (temp.next != head) temp = temp.next;
        temp.next = newNode;
        newNode.next = head;
    }

    void printList(Node head) {
        Node temp = head;
        if (head != null) {
            do {
                System.out.print(temp.data + " ");
                temp = temp.next;
            } while (temp != head);
        }
        System.out.println();
    }
}
```

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]
Time Complexity:

- O(n)

Space Complexity:

- O(1)

13. Insert a Node in a Sorted Circular Linked List
Program:

```
class InsertInSortedCircularList {
    Node head;

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

    void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            newNode.next = head;
            return;
        }
```

```
        Node current = head;
        Node prev = null;
        do {
            prev = current;
            current = current.next;
            if (data >= prev.data && data <= current.data) {
                break;
            }
            if (current == head && data < head.data) { // Insert before head
                break;
            }
        } while (current != head);

        newNode.next = current;
        prev.next = newNode;
        if (data < head.data) head = newNode; // New node becomes head
    }

    void printList() {
        Node temp = head;
        if (head != null) {
            do {
                System.out.print(temp.data + " ");
                temp = temp.next;
            } while (temp != head);
        }
        System.out.println();
    }
}
```

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)]
Time Complexity:

- O(n)

Space Complexity:

- O(1)

14. Check if Two Linked Lists Intersect and Find the Intersection Point
Program:

```
class LinkedListIntersection {
    Node head;

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

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

int getLength(Node head) {
    int length = 0;
    Node temp = head;
    while (temp != null) {
        length++;
        temp = temp.next;
    }
    return length;
}

Node getIntersection(Node head1, Node head2) {
    int len1 = getLength(head1);
    int len2 = getLength(head2);

    Node longer = len1 > len2 ? head1 : head2;
    Node shorter = len1 > len2 ? head2 : head1;
    int diff = Math.abs(len1 - len2);
        while (diff-- > 0) {
        longer = longer.next;
    }

    while (longer != null && shorter != null) {
        if (longer == shorter) {
            return longer; // Intersection point
        }
        longer = longer.next;
        shorter = shorter.next;
    }

    return null; // No intersection
}

void insert(Node head, int data) {
    Node newNode = new Node(data);
    if (head == null) {
        this.head = newNode;
        return;
    }
    Node temp = head;
    while (temp.next != null) temp = temp.next;
    temp.next = newNode;
}

void printList(Node head) {
    Node temp = head;
```

```
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
```

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
Time Complexity:

- O(m + n), where m and n are the lengths of the two lists.

Space Complexity:

- O(1)

15. Find the Middle Element of a Linked List in One Pass
Program:

```
class MiddleOfLinkedList {
    Node head;

    static class Node {
        int data;
        Node next;
        Node(int data) { this.data = data; next = null; }
    }
    Node findMiddle() {
        if (head == null) return null;

        Node slow = head;
        Node fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        return slow; // Slow pointer will be at the middle
    }
    void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
```

```
            return;
        }
        Node temp = head;
        while (temp.next != null) temp = temp.next;
        temp.next = newNode;
    }

    void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
```

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
Time Complexity:

- O(n), where n is the length of the list.

Space Complexity:

- O(1)