C-DAC Mumbai

Date 01/10/2024

Subject: Algorithm and Data Structure
Assignment 3

Solve the assignment with following thing to be added in each question.

-Program
-Flow chart
-Explanation
-Output
-Time and Space complexity

Submission Date: 3/10/2024

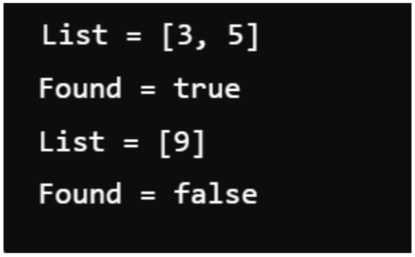
1. Implement a singly linked list with basic operations: insert, delete, search.

- **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

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

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

class SinglyLinkedList {
   private Node head;

   // Insert a new node at the end
   public void insert(int data) {
      Node newNode = new Node(data);
      if (head == null) {
         head = newNode;
         return;
      }
      Node current = head;
      while (current.next != null) {
         current = current.next;
      }
      current.next = newNode;
   }

```

```
    // Delete a node by value
    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;
        }
    }

    // Search for a node by value
    public boolean search(int data) {
        Node current = head;
        while (current != null) {
            if (current.data == data) return true;
            current = current.next;
        }
        return false;
    }

    // Display the list
    public void display() {
        Node current = head;
        System.out.print("List = [");
        while (current != null) {
            System.out.print(current.data + (current.next != null ? ", " : ""));
            current = current.next;
        }
        System.out.println("]");
    }
}

public class LinkedListOperations {
    public static void main(String[] args) {
        SinglyLinkedList list1 = new SinglyLinkedList();
        list1.insert(3);
        list1.insert(7);
        list1.insert(5);
        list1.delete(7);
        list1.display(); // Output: List = [3, 5]
```

- System.out.println("Found = " + list1.search(5)); // Output: Found = true
-
- SinglyLinkedList list2 = new SinglyLinkedList();
- list2.insert(9);
- list2.insert(4);
- list2.delete(4);
- list2.display(); // Output: List = [9]
- System.out.println("Found = " + list2.search(10)); // Output: Found = false
- }
- }



```
List = [3, 5]
Found = true
List = [9]
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 SinglyLinkedListReverse {
  Node head;

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

  public void reverse() {
    Node prev = null;
    Node current = head;
    Node next = null;
    while (current != null) {
      next = current.next;
      current.next = prev;
      prev = current;
      current = next;
    }
    head = prev;
  }

  public void display() {
    Node current = head;
    System.out.print("List = [");
    while (current != null) {
      System.out.print(current.data + (current.next != null ? ", " : ""));
      current = current.next;
    }
    System.out.println("]");
  }
}

public class ReverseLinkedList {
  public static void main(String[] args) {
    SinglyLinkedListReverse list1 = new SinglyLinkedListReverse();
    list1.insert(1);
    list1.insert(2);
    list1.insert(3);
    list1.insert(4);
    list1.insert(5);
    list1.reverse();
    list1.display(); // Output: List = [5, 4, 3, 2, 1]

    SinglyLinkedListReverse list2 = new SinglyLinkedListReverse();
    list2.insert(10);
    list2.insert(20);
    list2.insert(30);
    list2.reverse();
    list2.display(); // 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 CycleDetection {
  Node head;
```

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

    public boolean hasCycle() {
        Node slow = head;
        Node fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) return true;
        }
        return false;
    }
}

public class DetectCycle {
    public static void main(String[] args) {
        CycleDetection list1 = new CycleDetection();
        list1.insert(1);
        list1.insert(2);
        list1.insert(3);
        list1.insert(4);
        list1.insert(5);
        list1.head.next.next.next.next = list1.head.next; // Creating a cycle
        System.out.println("Cycle Detected: " + list1.hasCycle()); // Output: Cycle Detected

        CycleDetection list2 = new CycleDetection();
        list2.insert(6);
        list2.insert(7);
        list2.insert(8);
        list2.insert(9);
        System.out.println("Cycle Detected: " + list2.hasCycle()); // Output: 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 MergeSortedLinkedLists {
    Node head;

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

    public static MergeSortedLinkedLists merge(MergeSortedLinkedLists list1, MergeSortedLinkedLists list2) {
        MergeSortedLinkedLists mergedList = new MergeSortedLinkedLists();
        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;
            }
        }
        while (current1 != null) {
            mergedList.insert(current1.data);
            current1 = current1.next;
        }
        while (current2 != null) {
            mergedList.insert(current2.data);
            current2 = current2.next;
        }
        return mergedList;
    }
    public void display() {
        Node current = head;
        System.out.print("Merged List = [");
        while (current != null) {
            System.out.print(current.data + (current.next != null ? ", " : ""));
            current = current.next;
        }
        System.out.println("]");
    }
}

public class MergeLists {
    public static void main(String[] args) {
        MergeSortedLinkedLists list1 = new MergeSortedLinkedLists();
        list1.insert(1);
        list1.insert(3);
        list1.insert(5);

        MergeSortedLinkedLists list2 = new MergeSortedLinkedLists();
        list2.insert(2);
        list2.insert(4);
        list2.insert(6);

        MergeSortedLinkedLists mergedList = MergeSortedLinkedLists.merge(list1, list2);
        mergedList.display(); // Output: Merged List = [1, 2, 3, 4, 5, 6]

        MergeSortedLinkedLists list3 = new MergeSortedLinkedLists();
        list3.insert(10);
        list3.insert(15);
        list3.insert(20);

        MergeSortedLinkedLists list4 = new MergeSortedLinkedLists();
        list4.insert(12);
        list4.insert(18);
        list4.insert(25);

        MergeSortedLinkedLists mergedList2 = MergeSortedLinkedLists.merge(list3, list4);
        mergedList2.display(); // 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 Node {
    int data;
    Node next;

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

class SinglyLinkedList {
    Node head;

    // Insert a new node at the end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }

    // Find the nth node from the end
    public int findNthFromEnd(int n) {
        Node mainPtr = head;
        Node refPtr = head;

        // Move refPtr to n nodes ahead
        for (int i = 0; i < n; i++) {
            if (refPtr == null) return -1; // n is larger than the size of the list
            refPtr = refPtr.next;
        }

        // Move both pointers until refPtr reaches the end
        while (refPtr != null) {
            mainPtr = mainPtr.next;
            refPtr = refPtr.next;
        }
        return mainPtr.data;
    }

    public void display() {
        Node current = head;
        System.out.print("List = [");
        while (current != null) {
            System.out.print(current.data + (current.next != null ? ", " : ""));
            current = current.next;
        }
        System.out.println("]");
    }
}

public class FindNthNodeFromEnd {
    public static void main(String[] args) {
        SinglyLinkedList list1 = new SinglyLinkedList();
        list1.insert(10);
        list1.insert(20);
        list1.insert(30);
        list1.insert(40);
        list1.insert(50);
        System.out.println("Output: " + list1.findNthFromEnd(2)); // Output: 40

        SinglyLinkedList list2 = new SinglyLinkedList();
        list2.insert(5);
        list2.insert(15);
        list2.insert(25);
        list2.insert(35);
        System.out.println("Output: " + list2.findNthFromEnd(4)); // Output: 5
    }
}
```

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]

```
class RemoveDuplicates {
    Node head;

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

    public void removeDuplicates() {
        Node current = head;
        while (current != null && current.next != null) {
            if (current.data == current.next.data) {
                current.next = current.next.next; // Skip duplicate
            } else {
                current = current.next; // Move to the next unique node
            }
        }
    }

    public void display() {
        Node current = head;
        System.out.print("List = [");
        while (current != null) {
            System.out.print(current.data + (current.next != null ? ", " : ""));
            current = current.next;
        }
        System.out.println("]");
    }
}

public class RemoveDuplicatesFromSortedList {
    public static void main(String[] args) {
        RemoveDuplicates list1 = new RemoveDuplicates();
        list1.insert(1);
        list1.insert(1);
        list1.insert(2);
        list1.insert(3);
        list1.insert(3);
        list1.insert(4);
        list1.removeDuplicates();
        list1.display(); // Output: List = [1, 2, 3, 4]

        RemoveDuplicates list2 = new RemoveDuplicates();
        list2.insert(7);
        list2.insert(7);
        list2.insert(8);
        list2.insert(9);
        list2.insert(9);
        list2.insert(10);
        list2.removeDuplicates();
        list2.display(); // 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 DoublyNode {
    int data;
    DoublyNode next;
    DoublyNode prev;

    DoublyNode(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

class DoublyLinkedList {
    DoublyNode head;

    public void insert(int data) {
        DoublyNode newNode = new DoublyNode(data);
        if (head == null) {
            head = newNode;
            return;
        }
        DoublyNode current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
        newNode.prev = current;
    }

    public void delete(int data) {
        if (head == null) return;

        DoublyNode current = head;
        while (current != null && current.data != data) {
            current = current.next;
        }
        if (current == null) return; // Node not found

        if (current.prev != null) {
            current.prev.next = current.next;
        } else {
            head = current.next; // If head is to be deleted
        }
        if (current.next != null) {
            current.next.prev = current.prev;
        }
    }

    public void display() {
        DoublyNode current = head;
        System.out.print("List = [");
        while (current != null) {
            System.out.print(current.data + (current.next != null ? ", " : ""));
            current = current.next;
        }
        System.out.println("]");
    }
}

public class DoublyLinkedListOperations {
    public static void main(String[] args) {
        DoublyLinkedList list1 = new DoublyLinkedList();
        list1.insert(10);
        list1.insert(20);
        list1.insert(30);
        list1.delete(20);
        list1.display(); // Output: List = [10, 30]

        DoublyLinkedList list2 = new DoublyLinkedList();
        list2.insert(1);
        list2.insert(2);
        list2.insert(3);
        list2.delete(1);
        list2.display(); // 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]

```
class ReverseDoublyLinkedList {
    DoublyNode head;

    public void insert(int data) {
        DoublyNode newNode = new DoublyNode(data);
        if (head == null) {
            head = newNode;
            return;
        }
        DoublyNode current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
        newNode.prev = current;
    }

    public void reverse() {
        DoublyNode current = head;
        DoublyNode temp = null;

        while (current != null) {
            temp = current.prev;
            current.prev = current.next;
            current.next = temp;
            current = current.prev; // Move to the next node in the reversed list
        }
        if (temp != null) {
            head = temp.prev; // Update head to the last processed node
        }
    }

    public void display() {
        DoublyNode current = head;
        System.out.print("List = [");
        while (current != null) {
            System.out.print(current.data + (current.next != null ? ", " : ""));
            current = current.next;
        }
        System.out.println("]");
    }
}

public class ReverseDoublyLinkedListTest {
    public static void main(String[] args) {
        ReverseDoublyLinkedList list1 = new ReverseDoublyLinkedList();
        list1.insert(5);
        list1.insert(10);
        list1.insert(15);
        list1.insert(20);
        list1.reverse();
        list1.display(); // Output: List = [20, 15, 10, 5]

        ReverseDoublyLinkedList list2 = new ReverseDoublyLinkedList();
        list2.insert(4);
        list2.insert(8);
        list2.insert(12);
        list2.reverse();
        list2.display(); // 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]

```
class AddTwoNumbers {
    Node head;

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

    public static AddTwoNumbers addTwoNumbers(AddTwoNumbers list1, AddTwoNumbers
list2) {
        AddTwoNumbers result = new AddTwoNumbers();
        Node current1 = list1.head;
        Node current2 = list2.head;
        int carry = 0;

        while (current1 != null || current2 != null || carry != 0) {
```

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]

```
class CircularNode {
    int data;
    CircularNode next;

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

class CircularLinkedList {
    CircularNode head;

    public void insert(int data) {
        CircularNode newNode = new CircularNode(data);
        if (head == null) {
            head = newNode;
            newNode.next = head; // Point to itself
        } else {
            CircularNode current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head; // Complete the circular connection
        }
    }

    public void display() {
        if (head == null) return;
        CircularNode current = head;
        System.out.print("List = [");
        do {
            System.out.print(current.data + (current.next != head ? ", " : ""));
            current = current.next;
        } while (current != head);
        System.out.println("]");
    }
}

public class CircularLinkedListTest {
    public static void main(String[] args) {
        CircularLinkedList list1 = new CircularLinkedList();
        list1.insert(1);
        list1.insert(2);
        list1.insert(3);
        list1.display(); // Output: List = [1, 2, 3]

        CircularLinkedList list2 = new CircularLinkedList();
        list2.insert(10);
        list2.insert(20);
        list2.insert(30);
        list2.display(); // Output: List = [10, 20, 30]
    }
}
```

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 MultiLevelNode {
    int data;
    MultiLevelNode next;
    MultiLevelNode down;

    MultiLevelNode(int data) {
        this.data = data;
        this.next = null;
        this.down = null;
    }
}

class MultiLevelDoublyLinkedList {
    MultiLevelNode head;

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

    public void flatten(MultiLevelNode node, MultiLevelNode[] prev) {
        if (node == null) return;

        // Connect current node to the previous node
        if (prev[0] != null) {
            prev[0].next = node;
        }
        prev[0] = node;

        // Recursively flatten the down list
        flatten(node.down, prev);
        flatten(node.next, prev);
    }

    public MultiLevelNode flatten() {
        MultiLevelNode[] prev = new MultiLevelNode[1];
        flatten(head, prev);
        return head;
    }

    public void display(MultiLevelNode node) {
        System.out.print("Flattened List = [");
        while (node != null) {
            System.out.print(node.data + (node.next != null ? " → " : ""));
            node = node.next;
        }
        System.out.println("]");
    }
}

public class FlattenMultiLevelDoublyLinkedList {
    public static void main(String[] args) {
        MultiLevelDoublyLinkedList list1 = new MultiLevelDoublyLinkedList();
        list1.insert(1);
        list1.insert(2);
        list1.insert(3);
        list1.head.next.down = new MultiLevelNode(7);
        list1.head.next.down.next = new MultiLevelNode(8);
        list1.head.next.down.next.down = new MultiLevelNode(10);
        list1.head.next.down.next.down.next = new MultiLevelNode(12);

        list1.flatten();
        list1.display(list1.head); // 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.next.down = new MultiLevelNode(5);
        list2.head.next.down.next = new MultiLevelNode(6);
        list2.head.next.down.next.down = new MultiLevelNode(7);
        list2.head.next.down.next.down.next = new MultiLevelNode(9);

        list2.flatten();
        list2.display(list2.head); // 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 CircularSplitNode {
    int data;
    CircularSplitNode next;

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

class CircularLinkedListSplit {
    CircularSplitNode head;

    public void insert(int data) {
        CircularSplitNode newNode = new CircularSplitNode(data);
        if (head == null) {
            head = newNode;
            newNode.next = head; // Point to itself
        } else {
            CircularSplitNode current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head; // Complete the circular connection
        }
    }

    public void split() {
        if (head == null) return;

        CircularSplitNode slow = head;
        CircularSplitNode fast = head;

        // Use fast and slow pointer technique
        while (fast.next != head && fast.next.next != head) {
            slow = slow.next;
            fast = fast.next.next;
        }

        // Split the list
        CircularSplitNode head1 = head;
        CircularSplitNode head2 = slow.next;
        slow.next = head1; // Terminate the first half

        CircularSplitNode current = head2;
        while (current.next != head) {
            current = current.next;
        }
        current.next = head2; // Terminate the second half

        // Display both halves
        System.out.print("List1 = [");
        display(head1);
        System.out.print("List2 = [");
        display(head2);
    }

    private void display(CircularSplitNode node) {
        CircularSplitNode current = node;
        do {
            System.out.print(current.data + (current.next != node ? " → " : ""));
            current = current.next;
        } while (current != node);
        System.out.println("]");
    }
```

```
}

public class CircularLinkedListSplitTest {
    public static void main(String[] args) {
        CircularLinkedListSplit list1 = new CircularLinkedListSplit();
        list1.insert(1);
        list1.insert(2);
        list1.insert(3);
        list1.insert(4);
        list1.insert(5);
        list1.insert(6);
        list1.split(); // Output: List1 = [1 → 2 → 3], List2 = [4 → 5 → 6]

        CircularLinkedListSplit list2 = new CircularLinkedListSplit();
        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)]

```
class SortedCircularNode {
    int data;
    SortedCircularNode next;

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

class SortedCircularLinkedList {
    SortedCircularNode head;

    public void insert(int data) {
        SortedCircularNode newNode = new SortedCircularNode(data);
        if (head == null) {
            head = newNode;
            newNode.next = head; // Point to itself
        } else {
            SortedCircularNode current = head;

            // If new node is smaller than head
            if (data < head.data) {
                while (current.next != head) {
                    current = current.next;
                }
                current.next = newNode;
                newNode.next = head; // Update new node's next to head
                head = newNode; // Update head
                return;
            }

            // Find the appropriate place to insert
            while (current.next != head && current.next.data < data) {
                current = current.next;
            }
            newNode.next = current.next;
            current.next = newNode;
        }
    }

    public void display() {
        if (head == null) return;
        SortedCircularNode current = head;
        System.out.print("Circular List = [");
        do {
            System.out.print(current.data + (current.next != head ? " → " : ""));
            current = current.next;
        } while (current != head);
        System.out.println("]");
    }
}

public class SortedCircularLinkedListTest {
    public static void main(String[] args) {
        SortedCircularLinkedList list1 = new SortedCircularLinkedList();
        list1.insert(10);
        list1.insert(20);
        list1.insert(30);
        list1.insert(40);
        list1.insert(25);
        list1.display(); // Output: Circular List = [10 → 20 → 25 → 30 → 40]

        SortedCircularLinkedList list2 = new SortedCircularLinkedList();
        list2.insert(5);
        list2.insert(15);
        list2.insert(25);
        list2.insert(10);
        list2.display(); // Output: Circular List = [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

```
class IntersectionNode {
    int data;
    IntersectionNode next;

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

class LinkedListIntersection {
    IntersectionNode head;

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

    public IntersectionNode findIntersection(LinkedListIntersection list2) {
        IntersectionNode current1 = head;
        IntersectionNode current2 = list2.head;
        while (current1 != null) {
            current2 = list2.head;
            while (current2 != null) {
                if (current1 == current2) {
                    return current1; // Intersection point found
                }
                current2 = current2.next;
            }
            current1 = current1.next;
        }
        return null; // No intersection
    }
}

public class LinkedListIntersectionTest {
    public static void main(String[]
```

---

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 MiddleNode {
    int data;
    MiddleNode next;

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

class LinkedList {
    MiddleNode head;

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

    public int findMiddle() {
        if (head == null) {
            throw new RuntimeException("List is empty");
        }

        MiddleNode slow = head;
        MiddleNode fast = head;

        // Move fast pointer 2 steps and slow pointer 1 step
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow.data; // Slow pointer will be at the middle
    }

    public void display() {
        MiddleNode current = head;
        System.out.print("List = [");
        while (current != null) {
            System.out.print(current.data + (current.next != null ? ", " : ""));
            current = current.next;
        }
        System.out.println("]");
    }
}

public class FindMiddleElement {
    public static void main(String[] args) {
        // Test Case 1
        LinkedList list1 = new LinkedList();
        list1.insert(1);
        list1.insert(2);
        list1.insert(3);
        list1.insert(4);
        list1.insert(5);
        list1.display(); // Output: List = [1, 2, 3, 4, 5]
        System.out.println("Middle = " + list1.findMiddle()); // Output: Middle = 3

        // Test Case 2
        LinkedList list2 = new LinkedList();
        list2.insert(11);
        list2.insert(22);
        list2.insert(33);
        list2.insert(44);
        list2.insert(55);
        list2.insert(66);
        list2.display(); // Output: List = [11, 22, 33, 44, 55, 66]
        System.out.println("Middle = " + list2.findMiddle()); // Output: Middle = 44
    }
}
```