Amal Al-Dubai /Software Engineering

1. develop an implementation of the equals method in the context of the SinglyLinkedList class.
   public class SinglyLinkedList<T> {

```
private static class Node<T> {
    T data;
    Node<T> next;

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

private Node<T> head;
private int size;

public SinglyLinkedList() {
    this.head = null;
    this.size = 0;
}

public void add(T data) {
    Node<T> newNode = new Node<>(data);
    if (head == null) {
        head = newNode;
    } else {
        Node<T> current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
    size++;
}

public int size() {
    return size;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    SinglyLinkedList<?> other = (SinglyLinkedList<?>) obj;

    if (this.size != other.size) {
```

```
            return false;
        }

        Node<T> current1 = this.head;
        Node<?> current2 = other.head;

        while (current1 != null && current2 != null) {
            if (!current1.data.equals(current2.data)) {
                return false;
            }
            current1 = current1.next;
            current2 = current2.next;
        }

        return current1 == null && current2 == null;
    }

    public static void main(String[] args) {
        SinglyLinkedList<Integer> list1 = new SinglyLinkedList<>();
        SinglyLinkedList<Integer> list2 = new SinglyLinkedList<>();

        list1.add(1);
        list1.add(2);
        list1.add(3);

        list2.add(1);
        list2.add(2);
        list2.add(3);

        System.out.println(list1.equals(list2)); // Should print: true

        list2.add(4);

        System.out.println(list1.equals(list2)); // Should print: false
    }
}
```

2. Give an algorithm for finding the second-to-last node in a singly linked list in which the last node is indicated by a null next reference.

```
public class SinglyLinkedList<T> {

  private static class Node<T> {
    T data;
    Node<T> next;

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

  private Node<T> head;
  private int size;

  public SinglyLinkedList() {
    this.head = null;
    this.size = 0;
  }
```

```
    public void add(T data) {
        Node<T> newNode = new Node<>(data);
        if (head == null) {
            head = newNode;
        } else {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        size++;
    }

    public int size() {
        return size;
    }

    public T findSecondToLast() {
        if (head == null || head.next == null) {
            throw new IllegalStateException("List must have at least two nodes.");
        }

        Node<T> current = head;
        while (current.next.next != null) {
            current = current.next;
        }
        return current.data;
    }

    public static void main(String[] args) {
        SinglyLinkedList<Integer> list1 = new SinglyLinkedList<>();

        list1.add(1);
        list1.add(2);
        list1.add(3);
        list1.add(4);
        System.out.println("Second to last element in  the list: " + list1.findSecondToLast());
    }
}
```

3. Give an implementation of the size( ) method for the SingularlyLinkedList class, assuming that we did not maintain size as an instance variable.

```
public class SinglyLinkedList<T> {

    private static class Node<T> {
        T data;
        Node<T> next;

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

```
    private Node<T> head;

    public SinglyLinkedList() {
        this.head = null;
    }

    public int size() {
        int count = 0;
        Node<T> current = head;
        while (current != null) {
            count++;
            current = current.next;
        }
        return count;
    }

    public static void main(String[] args) {
        SinglyLinkedList<Integer> list = new SinglyLinkedList<>();
        list.head = new Node<>(1);
        list.head.next = new Node<>(2);
        list.head.next.next = new Node<>(3);
        list.head.next.next.next = new Node<>(4);
        System.out.println("Size of the list: " + list.size());
    }
}
```

4. Implement a rotate( ) method in the SinglyLinkedList class, which has semantics equal to addLast(removeFirst( )), yet without creating any new node.

```
public class SinglyLinkedList<T> {

    private static class Node<T> {
        T data;
        Node<T> next;

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

    private Node<T> head;
    private Node<T> tail;

    public SinglyLinkedList() {
        this.head = null;
        this.tail = null;
    }

    public void add(T data) {
        Node<T> newNode = new Node<>(data);
        if (head == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
```

```
    }

    public void rotate() {
        if (head == null || head.next == null) {
            return;
        }

        Node<T> first = head;
        head = head.next;
        first.next = null;
        tail.next = first;
        tail = first;
    }

    public void printList() {
        Node<T> current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        SinglyLinkedList<Integer> list = new SinglyLinkedList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);

        System.out.println("Original list:");
        list.printList();

        list.rotate();
        System.out.println("After the rotation:");
        list.printList();

    }
}
```

5. Describe an algorithm for concatenating two singly linked lists L and M, into a single list L′ that contains all the nodes of L followed by all the nodes of M.

```
package ds.lab3;


public class SinglyLinkedList<T> {

    private static class Node<T> {
        T data;
        Node<T> next;

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

```
}

private Node<T> head;
private Node<T> tail;

public SinglyLinkedList() {
    this.head = null;
    this.tail = null;
}

public void add(T data) {
    Node<T> newNode = new Node<>(data);
    if (head == null) {
        head = tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
}

public void concatenate(SinglyLinkedList<T> otherList) {
    if (this.head == null) {
        this.head = otherList.head;
        this.tail = otherList.tail;
    } else if (otherList.head != null) {
        this.tail.next = otherList.head;
        this.tail = otherList.tail;
    }

}

public void printList() {
    Node<T> current = head;
    while (current != null) {
        System.out.print(current.data + " -> ");
        current = current.next;
    }
    System.out.println("null");
}

public static void main(String[] args) {
    SinglyLinkedList<Integer> list1 = new SinglyLinkedList<>();
    list1.add(1);
    list1.add(2);
    list1.add(3);

    SinglyLinkedList<Integer> list2 = new SinglyLinkedList<>();
    list2.add(4);
    list2.add(5);
    list2.add(6);

    System.out.println("List 1:");
    list1.printList();

    System.out.println("List 2:");
    list2.printList();

    list1.concatenate(list2);

    System.out.println("Concatenated List:");
    list1.printList();
```

```
    }
}
```

6. Describe in detail an algorithm for reversing a singly linked list L using only a constant amount of additional space.

```
public class SinglyLinkedList<T> {

    private static class Node<T> {
        T data;
        Node<T> next;

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

    private Node<T> head;

    public SinglyLinkedList() {
        this.head = null;
    }

    public void add(T data) {
        Node<T> newNode = new Node<>(data);
        if (head == null) {
            head = newNode;
        } else {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }

    public void reverse() {
        Node<T> prev = null;
        Node<T> current = head;
        Node<T> next;

        while (current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
        }

        head = prev;
    }

    public void printList() {
        Node<T> current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
```

```
            current = current.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        SinglyLinkedList<String> list = new SinglyLinkedList<>();
        list.add("A");
        list.add("M");
        list.add("A");
        list.add("L");

        System.out.println("Original list:");
        list.printList();

        list.reverse();

        System.out.println("Reversed list:");
        list.printList();
    }
}
```