Data Structure Lab4 : Singly Linked List 2022-2023

Topics

1. Implement Node Class
2. Generics
3. Implement SinglyLinkedList Class
4. Implement Basic Methods of SinglyLinkedList
    - isEmpty()
    - size()
    - first()
    - last()
    - addFirst()
    - addLast()
    - removeFirst()

Homework

1. develop an implementation of the equals method in the context of the SinglyLinkedList class.
   import java.util.Objects;

```
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;
```

Data Structure Lab4 : Singly Linked List 2022-2023

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

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

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

    Node<T> currentThis = this.head;
    Node<?> currentOther = otherList.head;

    while (currentThis != null && currentOther != null) {
        if (!Objects.equals(currentThis.data, currentOther.data)) {
            return false;
        }
        currentThis = currentThis.next;
        currentOther = currentOther.next;
    }
    return currentThis == null && currentOther == null;
}
```

Data Structure Lab4 : Singly Linked List 2022-2023

```
    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.append(1);
        list1.append(2);
        list1.append(3);

        SinglyLinkedList<Integer> list2 = new SinglyLinkedList<>();
        list2.append(1);
        list2.append(2);
        list2.append(3);

        SinglyLinkedList<Integer> list3 = new SinglyLinkedList<>();
        list3.append(1);
        list3.append(2);
        list3.append(4);

        SinglyLinkedList<Integer> list4 = new SinglyLinkedList<>();
        list4.append(1);
        list4.append(2);

        System.out.println("list1 equals list2: " + list1.equals(list2));
        System.out.println("list1 equals list3: " + list1.equals(list3));
        System.out.println("list1 equals list4: " + list1.equals(list4));
    }
}
```

Data Structure Lab4 : Singly Linked List 2022-2023

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.
   Handle Edge Cases: If the list is empty (head is null) or has only one node (head's next is null), there is no second-to-last node. Return an appropriate result (e.g., null or throw an exception).
   Traverse the List: Use a pointer to traverse the list starting from the head. Maintain a reference to the **previous node** (second-to-last) and the **current node**.
   Stop Before the Last Node: Continue traversing until the next node of the current node is null. At this point:

- The previous node is the **second-to-last node**.
- Return this node.

```
Algorithm FindSecondToLastNode(head):
    if head is null OR head.next is null:
        Return null  // No second-to-last node exists

    Initialize two pointers:
        previous = null
        current = head

    While current.next is not null:
        previous = current
        current = current.next

    Return previous
```

Data Structure Lab4 : Singly Linked List 2022-2023

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

   Start at the head of the list.
   Use a counter variable count initialized to 0.
   Traverse through the list using a pointer, incrementing count for each node encountered.
   Stop when the pointer reaches null.
   Return the value of count as the size of the list.

   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 append(T data) {
    if (head == null) {
        head = new Node<>(data);
        return;
    }
```

Data Structure Lab4 : Singly Linked List 2022-2023

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

    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.append(10);
        list.append(20);
        list.append(30);
        list.append(40);

        System.out.println("Size of the list: " + list.size()); // Output: 4

        SinglyLinkedList<Integer> emptyList = new SinglyLinkedList<>();
        System.out.println("Size of the empty list: " + emptyList.size()); //
Output: 0
    }
}
```

Data Structure Lab4 : Singly Linked List 2022-2023

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;

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

public void append(T data) {
    if (head == null) {
        head = new Node<>(data);
        return;
    }
    Node<T> current = head;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new Node<>(data);
```

```
    }

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

        Node<T> oldHead = head;
        head = head.next;

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

        current.next = oldHead;
        oldHead.next = null;
    }

    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.append(10);
        list.append(20);
        list.append(30);
```

```
        list.append(40);

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

        list.rotate();

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

        list.rotate();

        System.out.println("After another 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.

   Check for Edge Cases:
   If either list is empty (null), return the other list as the result.
   Find the Last Node of L:
   Traverse list **L** until the last node (where next is null).
   Link the Last Node of L to the Head of M:
   Set the next pointer of the last node in **L** to point to the head of **M**.
   This effectively appends **M** to the end of **L**.
   Return the Head of L:
   The head of the concatenated list is the same as the head of **L**, since all the nodes of **L** will precede the nodes of **M**.

   Algorithm Concatenate(L, M):

```
If L is null:
    Return M
If M is null:
    Return L

current = L
While current.next is not null:
    current = current.next

current.next = M
Return L
```

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

Initialization

- **Previous Node (prev):** Initialize prev to None. This will point to the previous node during the traversal.
- **Current Node (curr):** Initialize curr to the head of the linked list (L).

2. Iteration

- **Loop:** While curr is not None:
    - **Next Node (next):** Store the next pointer of the curr node in a temporary variable next. This is crucial because we'll be modifying the next pointer of curr in the next step.
    - **Reverse Link:** Update the next pointer of the curr node to point to the prev node.
    - **Move Pointers:**
        - Update prev to point to the current node (curr).
        - Update curr to the next node (which was stored earlier).

3. Return Result

- After the loop completes, prev will be pointing to the new head of the reversed linked list.

Data Structure Lab4 : Singly Linked List 2022-2023

- Return prev.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def reverse_linked_list(head):
    prev = None
    curr = head

    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node

    return prev
```