## 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.

public boolean equals(Object obj) {  
 if (this == obj) {  
 return true; *// Identical objects* }  
  
 if (obj == null || !(obj instanceof SinglyLinkedList)) {  
 return false; *// Not a SinglyLinkedList instance* }  
  
 SinglyLinkedList<?> other = (SinglyLinkedList<?>) obj;  
  
 *// Check lengths first for efficiency* if (this.size() != other.size()) {  
 return false;  
 }  
  
 Node<?> n1 = this.head;  
 Node<?> n2 = other.head;  
  
 *// Compare nodes recursively, checking elements and references* while (n1 != null && n2 != null) {  
 if (!n1.data.equals(n2.data)) {  
 return false;  
 }  
  
 *// Prevent infinite loops due to cycles (recursive case)* if (n1.data == this.head || n2.data == other.head) {  
 return false; *// Cycle detected* }  
  
 n1 = n1.next;  
 n2 = n2.next;  
 }  
  
 *// Both lists traversed without mismatch: equal* return true;  
 }

1. 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 Node<T> findSecondToLast(Node<T> head) {  
 if (head == null || head.next == null) {  
 *// Handle error case (empty list or list with only one node)* return null;  
 }  
  
 Node<T> current = head;  
 Node<T> secondLast = head;  
  
 while (current.next != null) {  
 if (current == secondLast) {  
 *// Handle error case (list with less than two nodes)* return null;  
 }  
  
 secondLast = current;  
 current = current.next;  
 }  
  
 return secondLast;  
 }

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

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

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

public void rotate() {  
 if (head == null || head.next == null) {  
 *// Handle empty list or list with one node* return;  
 }  
  
 Node<T> tempHead = head;  
 Node<T> tempTail = head;  
  
 while (tempTail.next != null) {  
 tempTail = tempTail.next;  
 }  
  
 tempTail.next = tempHead;  
 tempHead.next = null;  
  
 head = head.next;  
 }

1. 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.

public static Node<T> concatenate(Node<T> headL, Node<T> headM) {  
 if (headL == null) {  
 return headM;  
 }  
  
 if (headM == null) {  
 return headL;  
 }  
  
 Node<T> currentL = headL;  
 while (currentL.next != null) {  
 currentL = currentL.next;  
 }  
  
 currentL.next = headM;  
  
 return headL;  
 }

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

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