## 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.
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

### Algorithm:

1. **Initialize two pointers**:  
   Create two pointers, previous and current, and initialize them both to the head of the linked list.
2. **Traverse the list**:  
   Iterate through the list. Move previous to the current node and current to the next node. Continue this until current.next is null, which indicates that current is the last node in the list.
3. **Stop before the last node**:  
   Once current is pointing to the last node (i.e., current.next == null), the previous node will be the second-to-last node. Return the previous node.
4. **Edge case**:  
   If the list has fewer than two nodes (i.e., the list is empty or contains only one node), return null as there is no second-to-last node

### Pseudocode:

text

Copy code

function findSecondToLastNode(head):

if head == null or head.next == null:

return null

previous = head

current = head.next

while current.next != null:

previous = current

current = current.next

1. Give an implementation of the size( ) method for the SingularlyLinkedList class, assuming that we did not maintain size as an instance variable.
2. Implement a rotate( ) method in the SinglyLinkedList class, which has semantics equal to addLast(removeFirst( )), yet without creating any new node.
3. 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.

### **Algorithm:**

1. **Check if L is empty:**
   * If L is empty, simply return M as the resulting list L′.
2. **Check if M is empty:**
   * If M is empty, return L as the resulting list L′.
3. **Find the tail of L:**
   * Start at the head of L and traverse the list until you reach the last node (where node.next == null).
4. **Link the last node of L to the head of M:**
   * Set tail.next = M.head, where tail is the last node of L.
5. **Return the modified list L:**
   * The head of L remains the head of L′, with all nodes from M appended.

### **Pseudocode:**

text

Copy code

function concatenate(L, M):

if L.head == null:

return M // If L is empty, return M

if M.head == null:

return L // If M is empty, return L

tail = L.head

while tail.next != null:

tail = tail.next // Traverse to the end of L

tail.next = M.head // Link last node of L to the head of M

return L // Return L as the concatenated list

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

### **Algorithm:**

To reverse the singly linked list, we iteratively reverse the direction of the next pointers for all nodes in the list.

1. **Initialize Pointers**:
   * Use three pointers:
     + prev (initially null): Tracks the previous node.
     + current (initially head): Tracks the current node being processed.
     + next (initially null): Temporarily stores the next node in the original list to avoid losing track during reversal.
2. **Iterate Through the List**:
   * While current is not null:
     + Save the next node in the next pointer (next = current.next).
     + Reverse the link by setting current.next = prev.
     + Move prev one step forward (prev = current).
     + Move current one step forward (current = next).
3. **Update the Head**:
   * After the loop ends, prev will be pointing to the new head of the reversed list. Update the head of the list to prev.
4. **Return the Reversed List**:
   * The list is now reversed, and the head points to the new first node.

### **Pseudocode**:

text

Copy code

function reverseList(head):

prev = null

current = head

while current != null:

next = current.next // Save the next node

current.next = prev // Reverse the link

prev = current // Move prev forward

current = next // Move current forward

head = prev // Update head to the new first node

return head