develop an implementation of the equals method in the context of the SinglyLinkedList class.?

public class SinglyLinkedList<T> {

private Node<T> head;

// ... other methods for the SinglyLinkedList class

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

SinglyLinkedList<T> otherList = (SinglyLinkedList<T>) obj;

Node<T> currentNode = head;

Node<T> otherCurrentNode = otherList.head;

while (currentNode != null && otherCurrentNode != null) {

if (!currentNode.data.equals(otherCurrentNode.data)) {

return false;

}

currentNode = currentNode.next;

otherCurrentNode = otherCurrentNode.next;

}

// Check if both lists reached the end at the same time

return currentNode == null && otherCurrentNode == null;

}

private static class Node<T> {

T data;

Node<T> next;

Node(T data) {

this.data = data;

this.next = null;

}

}

}

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 Node<T> head;

// ... other methods for the SinglyLinkedList class

public Node<T> findSecondToLastNode() {

if (head == null || head.next == null) {

System.out.println("List is too short to find the second-to-last node.");

return null;

}

Node<T> current = head.next;

Node<T> previous = head;

while (current.next != null) {

previous = current;

current = current.next;

}

return previous;

}

private static class Node<T> {

T data;

Node<T> next;

Node(T data) {

this.data = data;

this.next = null;

}

}

}

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

public class SinglyLinkedList<T> {

private Node<T> head;

// ... other methods for the SinglyLinkedList class

public int size() {

int count = 0;

Node<T> current = head;

while (current != null) {

count++;

current = current.next;

}

return count;

}

private static class Node<T> {

T data;

Node<T> next;

Node(T data) {

this.data = data;

this.next = null;

}

}

}

1. 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 Node<T> head;

// ... other methods for the SinglyLinkedList class

public void rotate() {

if (head == null || head.next == null) {

// List is empty or has only one node, no rotation needed

return;

}

Node<T> secondNode = head.next;

Node<T> current = head;

while (current.next != null) {

current = current.next;

}

// Set the last node's next reference to the head

current.next = head;

// Update the head to the second node

head = secondNode;

// Set the new last node's next reference to null

current.next.next = null;

}

private static class Node<T> {

T data;

Node<T> next;

Node(T data) {

this.data = data;

this.next = null;

}

}

}

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.

public class SinglyLinkedList<T> {

private Node<T> head;

// ... other methods for the SinglyLinkedList class

public SinglyLinkedList<T> concatenate(SinglyLinkedList<T> otherList) {

if (head == null) {

return otherList;

}

if (otherList.head == null) {

return this;

}

SinglyLinkedList<T> concatenatedList = new SinglyLinkedList<>();

concatenatedList.head = head;

Node<T> currentNode = concatenatedList.head;

while (currentNode.next != null) {

currentNode = currentNode.next;

}

currentNode.next = otherList.head;

return concatenatedList;

}

private static class Node<T> {

T data;

Node<T> next;

Node(T data) {

this.data = data;

this.next = null;

}

}

}

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 Node<T> head;

// ... other methods for the SinglyLinkedList class

public SinglyLinkedList<T> reverse() {

if (head == null || head.next == null) {

// List is empty or has only one node, no reversal needed

return this;

}

Node<T> current = head;

Node<T> previous = null;

Node<T> next = null;

while (current != null) {

next = current.next;

current.next = previous;

previous = current;

current = next;

}

head = previous;

return this;

}

private static class Node<T> {

T data;

Node<T> next;

Node(T data) {

this.data = data;

this.next = null;

}

}