class Node {

public int data;

public Node next;

public Node(int value) {

data = value;

next = null;

}

}

class SinglyLL {

public Node first;

public SinglyLL() {

first = null;

}

public void insertLast(int no) {

Node newn = new Node(no);

if (first == null) {

first = newn;

} else {

Node temp = first;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newn;

}

}

public void display() {

System.out.print("[ ");

Node temp = first;

while (temp != null) {

System.out.print(temp.data);

temp = temp.next;

if (temp != null) {

System.out.print(", ");

}

}

System.out.print(" ]");

}

public void removeDuplicates() {

Node temp = first;

while (temp != null && temp.next != null) {

if (temp.data == temp.next.data) {

// Skip the duplicate node

temp.next = temp.next.next;

} else {

temp = temp.next; // Move to the next node

}

}

}

}

class Question6 {

public static void main(String[] args) {

// Test case 1: Create a sorted linked list with duplicates

SinglyLL obj1 = new SinglyLL();

obj1.insertLast(1);

obj1.insertLast(1);

obj1.insertLast(2);

obj1.insertLast(3);

obj1.insertLast(3);

obj1.insertLast(4);

System.out.print("Input: List = ");

obj1.display();

obj1.removeDuplicates();

System.out.print("\nOutput: List = ");

obj1.display();

System.out.println();

// Test case 2:

SinglyLL obj2 = new SinglyLL();

obj2.insertLast(7);

obj2.insertLast(7);

obj2.insertLast(8);

obj2.insertLast(9);

obj2.insertLast(9);

obj2.insertLast(10);

System.out.print("Input: List = ");

obj2.display();

obj2.removeDuplicates();

System.out.print("\nOutput: List = ");

obj2.display();

System.out.println();

}

}  
explanation  
The algorithm for removing duplicates from a sorted linked list iteratively traverses the list using a single pointer. It compares each node with the next node to check for duplicates. When a duplicate is found, it adjusts the next pointer of the current node to skip over the duplicate node. This process continues until all nodes are checked, effectively removing duplicates while maintaining the sorted order of the list. The approach ensures optimal performance with linear time complexity and constant space usage, making it both efficient and straightforward.  
  
time and space  
o(n)…0(1)  
  
output  
