Name: Swarnali Ghosh

Java Pre-Skilling Training Session

Assignment -5.1(Task2 to Task 8)

Module-5 (DAY 1 TO 6)

Mail-id: swarnalighosh666@gmail.com

TASK-2

Linked List Middle Element Search

You are given a singly linked list. Write a function to find the middle element without using any extra space and only one traversal through the linked list.

SOLUTION:

Here's the Java function to find the middle element of a singly linked list without using any extra space and with only one traversal:

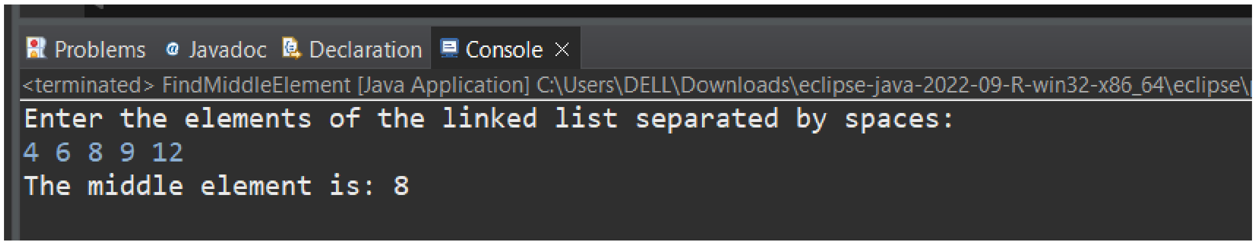
```
class ListNode {
  int val;
  ListNode next;
  ListNode(int val) {
    this.val = val;
    this.next = null;
  }
}
public class FindMiddleElement {
    public static ListNode findMiddle(ListNode head) {
        if (head == null) return null;
        ListNode slowPointer = head;
        ListNode fastPointer = head;

        while (fastPointer != null && fastPointer.next != null) {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next.next;
        }
        return slowPointer;
    }
}
```

Here, I have implemented the function in creating the entire code→

```
package com.wipro.swarnali;

import java.util.Scanner;

class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class FindMiddleElement {

    public static ListNode findMiddle(ListNode head) {
        if (head == null) return null;

        ListNode slowPointer = head;
```

```java
        ListNode fastPointer = head;

        // Move slowPointer one step at a time and fastPointer two
steps at a time
        // When fastPointer reaches the end, slowPointer will be at
the middle
        while (fastPointer != null && fastPointer.next != null) {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next.next;
        }

        return slowPointer;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the elements of the linked list
separated by spaces:");
        String[] elements = scanner.nextLine().split(" ");

        ListNode head = null;
        ListNode tail = null;

        // Creating the linked list
        for (String element : elements) {
            int val = Integer.parseInt(element);
            ListNode newNode = new ListNode(val);
            if (head == null) {
                head = newNode;
                tail = newNode;
            } else {
                tail.next = newNode;
                tail = tail.next;
            }
        }
        // Finding the middle element
        ListNode middle = findMiddle(head);

        if (middle != null) {
            System.out.println("The middle element is: " +
middle.val);
        } else {
            System.out.println("The list is empty.");
        }
        scanner.close();
    }
}
```

Output: -



Problems Javadoc Declaration Console

```
<terminated> FindMiddleElement [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-x86_64\eclipse\
Enter the elements of the linked list separated by spaces:
4 6 8 9 12
The middle element is: 8
```

TASK-3

Queue Sorting with Limited Space

You have a queue of integers that you need to sort. You can only use additional space equivalent to one stack. Describe the steps you would take to sort the elements in the queue.

SOLUTION:

Sorting a queue with limited space, equivalent to just one stack, in Java requires a creative approach. Here's a simple algorithm to achieve this→

1. **Input Queue:** This is the queue we want to sort.
2. **Output Queue:** This is the queue where we'll store elements sorted in ascending order.
3. **Temporary Stack:** This stack will be used for temporary storage during the sorting process.

Here's a step-by-step explanation of how the algorithm works:

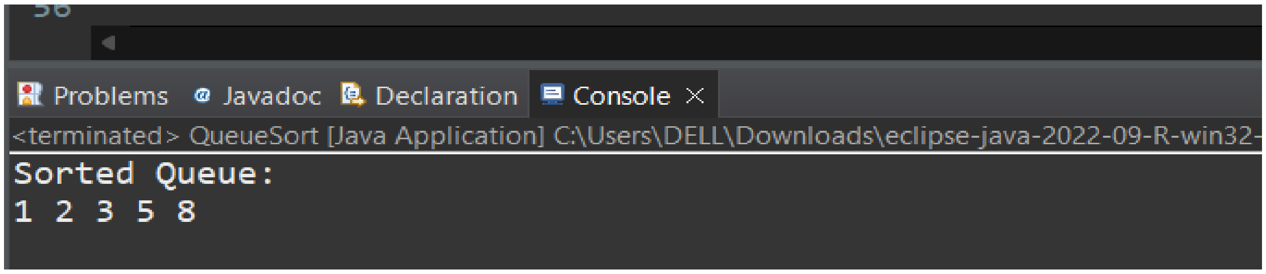
1. **Populating the Temporary Stack:** - We'll iterate through the input queue and push each element onto the temporary stack.
2. **Finding Minimum Element:** - While pushing elements from the input queue to the temporary stack, we'll keep track of the minimum element encountered so far.
3. **Emptying the Temporary Stack:** - Once all elements are pushed onto the temporary stack, we'll start popping elements from the stack and enqueueing them back into the input queue, except for the minimum element.
4. **Repeating the Process:** - We'll repeat steps 1-3 until the temporary stack is empty.
5. **Enqueueing the Minimum Element:** - After the temporary stack is empty, we'll enqueue the minimum element (which we kept track of earlier) into the output queue.
6. **Final Output:** - The output queue will now contain elements sorted in ascending order.

Here is the Java code implementation: -

```java
package com.wipro.swarnali;

import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class QueueSort {
    public static Queue<Integer> sortQueue(Queue<Integer> inputQueue)
{
        Queue<Integer> outputQueue = new LinkedList<>();
        Stack<Integer> tempStack = new Stack<>();

        while (!inputQueue.isEmpty()) {
            int minElement = inputQueue.poll();

            // Finding the minimum element and populating the
temporary stack
            while (!inputQueue.isEmpty()) {
                int currentElement = inputQueue.poll();
                if (currentElement < minElement) {
                    tempStack.push(minElement);
                    minElement = currentElement;
                } else {
                    tempStack.push(currentElement);
                }
            }
            // Enqueueing elements back into the input queue except
for the minimum element
            while (!tempStack.isEmpty()) {
                int currentElement = tempStack.pop();
                if (currentElement != minElement) {
                    inputQueue.offer(currentElement);
                }
            }

            // Enqueueing the minimum element into the output queue
            outputQueue.offer(minElement);
        }

        return outputQueue;
    }

    public static void main(String[] args) {
        Queue<Integer> inputQueue = new LinkedList<>();
        inputQueue.offer(5);
        inputQueue.offer(3);
        inputQueue.offer(8);
        inputQueue.offer(1);
        inputQueue.offer(2);

        Queue<Integer> sortedQueue = sortQueue(inputQueue);
        System.out.println("Sorted Queue:");
        while (!sortedQueue.isEmpty()) {
            System.out.print(sortedQueue.poll() + " ");
        }
    }
}
```

This implementation sorts the input queue and prints the sorted elements. It uses only one stack for additional space, satisfying the limited space requirement.

Output: -



```
<terminated> QueueSort [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-
Sorted Queue:
1 2 3 5 8
```

TASK-4

Stack Sorting In-Place

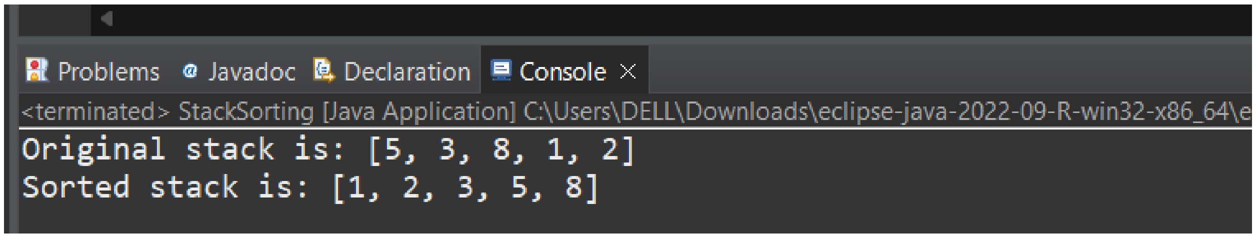
You must write a function to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure such as an array. The stack supports the following operations: push, pop, peek, and isEmpty.

SOLUTION:

To sort a stack in ascending order such that the smallest items are on top, and using an additional temporary stack without copying elements into any other data structure, you can follow a simple algorithm.

```java
package com.wipro.swarnali;

import java.util.Stack;

public class StackSorting {
    public static void sortStack(Stack<Integer> stack) {
        Stack<Integer> tempStack = new Stack<>();

        while (!stack.isEmpty()) {
            int temp = stack.pop();

            // Move elements from tempStack to stack until the correct
position for temp is found
            while (!tempStack.isEmpty() && tempStack.peek() < temp) {
                stack.push(tempStack.pop());
            }

            tempStack.push(temp);
        }

        // Copy elements from tempStack back to stack to have them in
ascending order
        while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
        }
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(5);
        stack.push(3);
        stack.push(8);
        stack.push(1);
        stack.push(2);

        System.out.println("Original stack is: " + stack);
        sortStack(stack);
        System.out.println("Sorted stack is: " + stack);
    }
}
```

Output: -



```
<terminated> StackSorting [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-x86_64\e
Original stack is: [5, 3, 8, 1, 2]
Sorted stack is: [1, 2, 3, 5, 8]
```

Here's a summary of the steps involved in sorting the stack in ascending order (smallest item on top):

1. **Initialization**: Create a temporary stack.
2. **Sorting Process**:
    - Pop elements from the original stack one by one.
    - While the temporary stack is not empty and its top element is greater than the current element being popped, push elements from the temporary stack back into the original stack.
    - Push the current element into the temporary stack.
3. **Copying Back**: After sorting, copy elements from the temporary stack back into the original stack without utilizing any other data structure such as an array. This ensures adherence to the constraint of not copying elements into any other data structure.

4. **Result**: The original stack will now have elements sorted in ascending order, with the smallest item on top.

This process ensures that the smallest item is continuously pushed to the top of the temporary stack, resulting in the sorted order when elements are copied back to the original stack.

TASK-5

Removing Duplicates from a Sorted Linked List

A sorted linked list has been constructed with repeated elements. Describe an algorithm to remove all duplicates from the linked list efficiently.

SOLUTION:

Algorithm Explanation:

1. **Initialization**:
    - Starting with a pointer, current, set to the head of the linked list. This pointer will be used to traverse the list.
2. **Traversal and Removal of Duplicates**:
    - While current is not null and current.next is not null, do the following:
        - Compare the value of the current node (current.val) with the value of the next node (current.next.val).
        - If the values are equal, it means there is a duplicate:
            - Skip the next node by setting current.next to current.next.next. This effectively removes the duplicate node from the list.
        - If the values are not equal, move the current pointer to the next node (current = current.next).
    - Continue this process until the end of the list is reached.
3. **End of List**:

- When the traversal is complete, all duplicate nodes have been removed, and the list contains only unique elements.

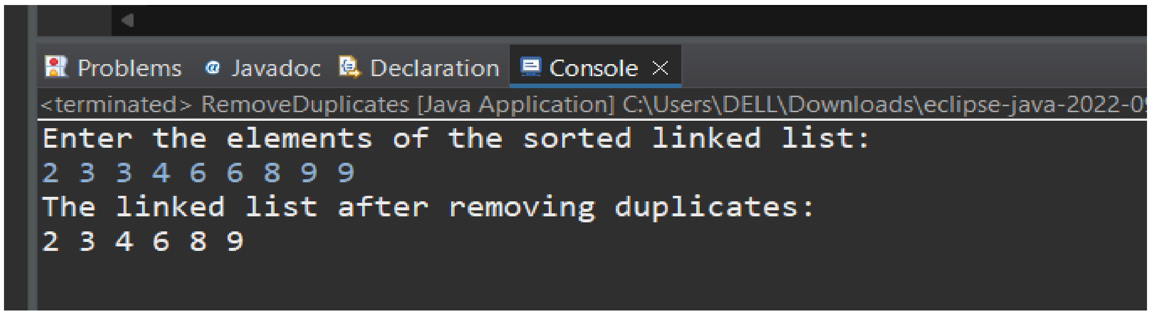
Key Points:

- **Single Pass**: The algorithm only makes one pass through the linked list. This makes it efficient with a time complexity of O(n), where n is the number of nodes in the list.
- **In-Place Modification**: The algorithm modifies the list in place, meaning no extra space is needed for another list. This results in a space complexity of O(1).

Java Code Implementation:

```java
package com.wipro.swarnali;

import java.util.Scanner;

class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public class RemoveDuplicates {
    public static void removeDuplicates(ListNode head) {
        // Pointer to the current node
        ListNode current = head;

        // Traverse the list until the end
        while (current != null && current.next != null) {
            // If current node's value is the same as the next node's
value
            if (current.val == current.next.val) {
                // Skip the next node
                current.next = current.next.next;
            } else {
                // Move to the next node
                current = current.next;
            }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the elements of the sorted linked 
list: ");
        String[] elements = scanner.nextLine().split(" ");

        // Check if the input is not empty
        if (elements.length == 0) {
            System.out.println("The list is empty.");
            return;
        }

        // Create the head of the linked list
        ListNode head = new ListNode(Integer.parseInt(elements[0]));
        ListNode current = head;

        // Create the rest of the linked list
        for (int i = 1; i < elements.length; i++) {
            current.next = new 
ListNode(Integer.parseInt(elements[i]));
            current = current.next;
        }

        // Remove duplicates
        removeDuplicates(head);

        // Print the list to verify duplicates are removed
        current = head;
        System.out.println("The linked list after removing 
duplicates:");
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
    }
}
```

Output: -



```
<terminated> RemoveDuplicates [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-0
Enter the elements of the sorted linked list:
2 3 3 4 6 6 8 9 9
The linked list after removing duplicates:
2 3 4 6 8 9
```

TASK-6

Searching for a Sequence in a Stack

Given a stack and a smaller array representing a sequence, write a function that determines if the sequence is present in the stack. Consider the sequence present if, upon popping the elements, all elements of the array appear consecutively in the stack.

SOLUTION:

Problem Statement:

Given a stack and a sequence (array), we need to check if the sequence appears consecutively in the stack when popping the elements. The sequence should be matched in the reverse order because elements are popped from the top of the stack.

Improved Algorithm:

1. **Initialization**:
   - Create a pointer `i` that starts at the last index of the sequence (i.e., `i = sequence.length - 1`). This allows us to match the sequence in reverse order.
2. **Traverse the Stack**:
   - While the stack is not empty:
     - Pop the top element from the stack.
     - Compare the popped element with the current element of the sequence (`sequence[i]`).
     - If the elements match:
       - Decrement `i` to move to the next element of the sequence.
       - If `i` goes below 0, it means we have matched the entire sequence. Return `true`.
     - If the elements do not match:
       - Continue popping the next element from the stack without resetting `i`.
3. **End of Stack**:

- If the stack is exhausted before finding the full sequence, return `false`.

Key Points:

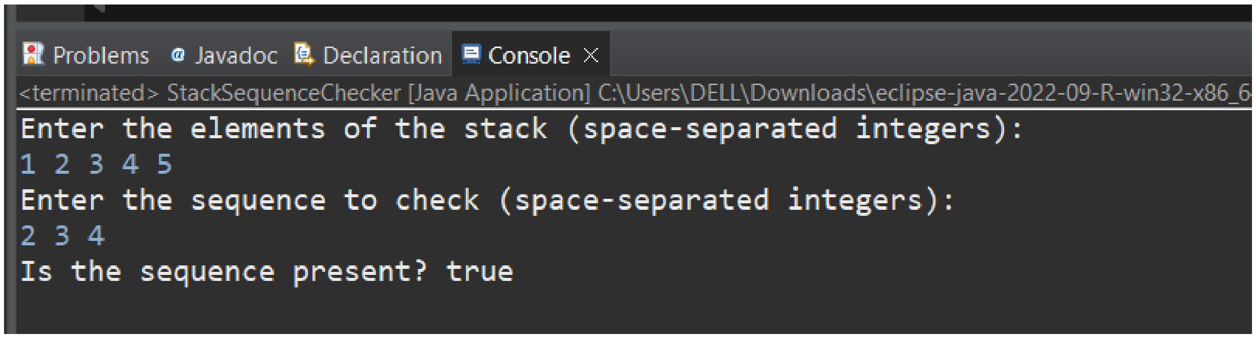
- **Reverse Matching**: We match the sequence in reverse order because stacks follow the Last-In-First-Out (LIFO) principle. The last element pushed onto the stack is the first one to be popped.
- **Single Pass**: The algorithm makes a single pass through the stack, ensuring an efficient check.
- **No Reset on Mismatch**: Unlike some previous versions, the pointer `i` is not reset on a mismatch. This allows the algorithm to continue checking subsequent elements in the stack.

Java code Implementation:

```
package com.wipro.swarnali;

import java.util.Scanner;
import java.util.Stack;

public class StackSequenceChecker {

    public static boolean isSequencePresent(Stack<Integer> stack,
int[] sequence) {
        // Pointer to track position in the sequence array from end
        int i = sequence.length - 1;

        // Loop through the stack elements
        while (!stack.isEmpty()) {
            int top = stack.pop();

            // Check if the current stack top matches the current
sequence element
            if (top == sequence[i]) {
                i--;
                // Check if we have matched the entire sequence
                if (i < 0) {
                    return true;
                }
            }
        }
```

```java
        // Return false if the sequence was not fully matched
        return false;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Read the stack elements from the user
        System.out.println("Enter the elements of the stack (space-
separated integers):");
        String[] stackElements = scanner.nextLine().split(" ");
        Stack<Integer> stack = new Stack<>();
        for (String element : stackElements) {
            stack.push(Integer.parseInt(element));
        }

        // Read the sequence to check from the user
        System.out.println("Enter the sequence to check (space-
separated integers):");
        String[] sequenceElements = scanner.nextLine().split(" ");
        int[] sequence = new int[sequenceElements.length];
        for (int i = 0; i < sequenceElements.length; i++) {
            sequence[i] = Integer.parseInt(sequenceElements[i]);
        }

        // Check if the sequence is present in the stack
        boolean result = isSequencePresent(stack, sequence);
        System.out.println("Is the sequence present? " + result);

        scanner.close();
    }
}
```

Output: -



```
<terminated> StackSequenceChecker [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-x86_6
Enter the elements of the stack (space-separated integers):
1 2 3 4 5
Enter the sequence to check (space-separated integers):
2 3 4
Is the sequence present? true
```

TASK-7

Merging Two Sorted Linked Lists

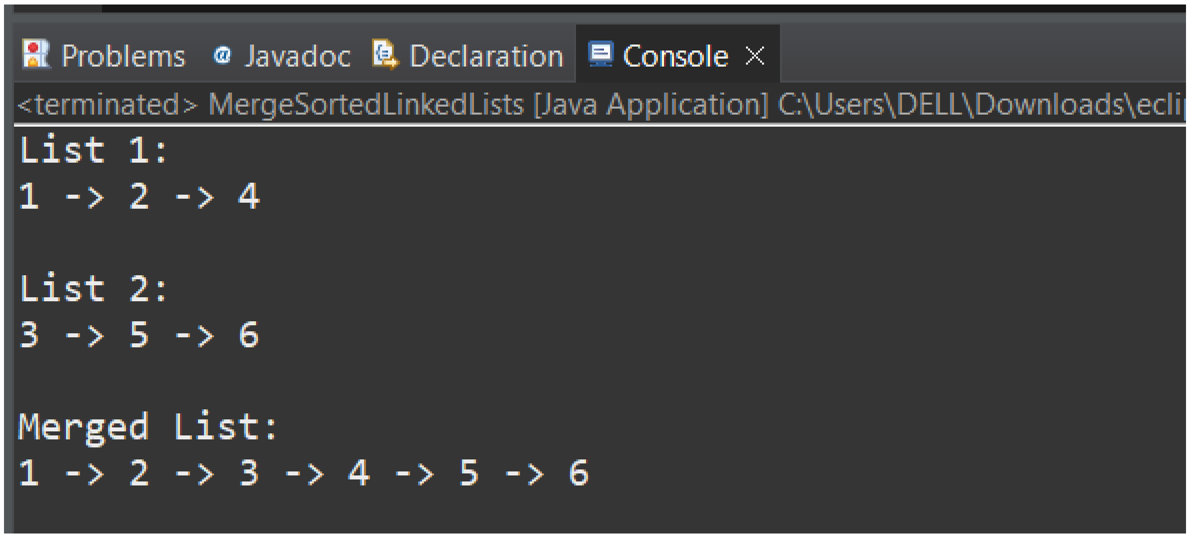
You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).

SOLUTION:

```
package com.wipro.swarnali;

class List_Node {
    int val;
    List_Node next;
    List_Node(int x) { val = x; }
}

public class MergeSortedLinkedLists {

    public static List_Node mergeTwoLists(List_Node l1, List_Node l2)
{
        // If either list is empty, return the other list
        if (l1 == null) return l2;
        if (l2 == null) return l1;

        // Make sure l1 is the list starting with the smaller element
        if (l1.val > l2.val) {
           List_Node temp = l1;
            l1 = l2;
            l2 = temp;
        }

        // Keep the head of the merged list
        List_Node head = l1;

        while (l1 != null && l2 != null) {
           List_Node prev = null;

            // Traverse l1 as long as its elements are smaller than or
equal to l2's elements
```

```java
            while (l1 != null && l1.val <= l2.val) {
                prev = l1;
                l1 = l1.next;
            }

            // Link the last node of l1 to the current node of l2
            prev.next = l2;

            // Swap l1 and l2 so that l1 points to the list starting
with the smaller element
            List_Node temp = l1;
            l1 = l2;
            l2 = temp;
        }

        return head;
    }

    public static void printList(List_Node head) {
      List_Node current = head;
        while (current != null) {
            System.out.print(current.val);
            if (current.next != null) {
                System.out.print(" -> ");
            }
            current = current.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {

        // Creating first sorted linked list: 1 -> 2 -> 4
      List_Node l1 = new List_Node(1);
        l1.next = new List_Node(2);
        l1.next.next = new List_Node(4);

        // Creating second sorted linked list: 3 -> 5 -> 6
        List_Node l2 = new List_Node(3);
        l2.next = new List_Node(5);
        l2.next.next = new List_Node(6);

        System.out.println("List 1: ");
        printList(l1);

        System.out.println("\nList 2: ");
        printList(l2);

        // Merging the lists
        List_Node mergedList = mergeTwoLists(l1, l2);

        System.out.println("\nMerged List: ");
        printList(mergedList);
    }
}
```

Output: -



```
<terminated> MergeSortedLinkedLists [Java Application] C:\Users\DELL\Downloads\ecli
List 1:
1 -> 2 -> 4

List 2:
3 -> 5 -> 6

Merged List:
1 -> 2 -> 3 -> 4 -> 5 -> 6
```

Explanation:

1. **List_Node class**:

   This is a simple definition of a node in a linked list, containing an integer value and a reference to the next node.

2. **mergeTwoLists method**:
   1. The method starts by checking if either of the input lists is null and returns the other list if one is null.
   2. It ensures that l1 starts with the smaller element. If not, l1 and l2 are swapped.
   3. The head variable keeps track of the start of the merged list.
   4. The while loop continues until one of the lists is exhausted.

    5. Inside the loop, it uses a prev pointer to keep track of the last node in the merged portion of l1. It traverses l1 until it finds a node that is greater than l2.val.
    6. It then links prev.next to l2 and swaps l1 and l2 to ensure l1 always points to the list with the smaller current element.
    7. This process continues until one of the lists is fully traversed.
    8. The method returns the head of the merged list.
3. **printList method**:
    1. The method prints each node's value followed by " -> " only if the current node is not the last node. This ensures clean output without "null".
4. **main method**:
    1. Creates two sample sorted linked lists, prints them, merges them, and then prints the merged list.

This implementation merges the two input linked lists in place without creating any new nodes, meeting the requirement of not using any extra space.

TASK-8

Circular Queue Binary Search

Consider a circular queue (implemented using a fixed-size array) where the elements are sorted but have been rotated at an unknown index. Describe an approach to perform a binary search for a given element within this circular queue.

SOLUTION:

To perform a binary search on a rotated sorted array (which can be seen as a circular queue), we can adapt the standard binary search algorithm by incorporating checks to determine the sorted portion of the array at each step.

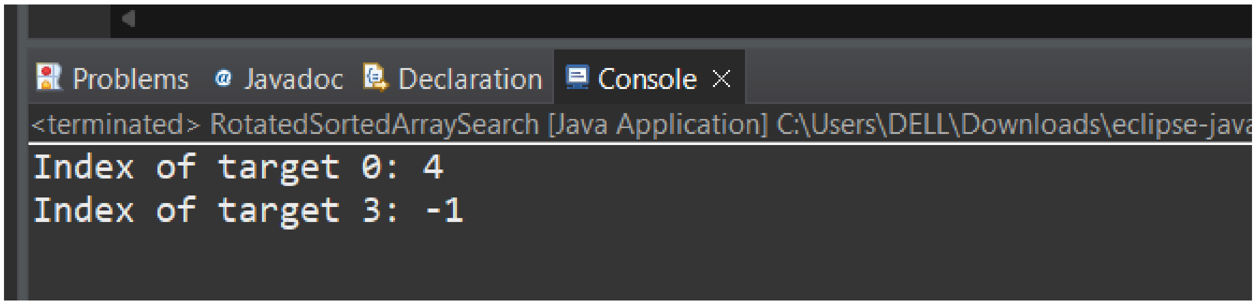
Here is a step-by-step approach to achieve this:

1. **Initialize Pointers**: Set two pointers, low and high, to the start and end of the array respectively.
2. **Binary Search Loop**: Loop while low is less than or equal to high:
   - Calculate the middle index mid as low + (high - low) / 2.
   - Check if the element at mid is the target element. If it is, return mid.
   - Determine which half of the array (from low to mid or from mid to high) is properly sorted.
     - If the left half is sorted (arr[low] <= arr[mid]):
       - Check if the target is within the range of the sorted half (arr[low] <= target < arr[mid]).
         - If it is, narrow the search to the left half by setting high to mid - 1.
         - Otherwise, narrow the search to the right half by setting low to mid + 1.
     - If the right half is sorted (arr[mid] <= arr[high]):
       - Check if the target is within the range of the sorted half (arr[mid] < target <= arr[high]).
         - If it is, narrow the search to the right half by setting low to mid + 1.
         - Otherwise, narrow the search to the left half by setting high to mid - 1.
3. **Element Not Found**: If the loop exits without finding the target, return an indication that the element is not present (e.g., -1).

Java Code Implementation: -

```java
package com.wipro.swarnali;


public class RotatedSortedArraySearch {

    public static int search(int[] arr, int target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            // Check if mid is the target element
            if (arr[mid] == target) {
                return mid;
            }

            // Determine if the left half is sorted
            if (arr[low] <= arr[mid]) {
                // Check if the target is in the left half
                if (arr[low] <= target && target < arr[mid]) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            }
            // Otherwise, the right half must be sorted
            else {
                // Check if the target is in the right half
                if (arr[mid] < target && target <= arr[high]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }
        }

        // Target element not found
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {4, 5, 6, 7, 0, 1, 2};
        int target = 0;
        int result = search(arr, target);
        System.out.println("Index of target " + target + ": " +
result);

        target = 3;
        result = search(arr, target);
        System.out.println("Index of target " + target + ": " +
result);
    }
}
```

Output: -



```
<terminated> RotatedSortedArraySearch [Java Application] C:\Users\DELL\Downloads\eclipse-java
Index of target 0: 4
Index of target 3: -1
```

This approach ensures that we correctly handle the rotated sorted array by leveraging the properties of sorted subarrays within the rotation.