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.

```
package com.wipro.linear;

public class LinkedList {

	private Node head;
	private Node tail;
	private int length;

	class Node {
		int value;
		Node next;

		public Node(int value) {
			super();
			this.value = value;
		}
	}
	public LinkedList(int value) {
		super();
```

```
        Node newNode = new Node(value);
        //System.out.println("Node :" +newNode);
        head = newNode;
        tail = newNode;
        length = 1;
    }

    public void getHead() {
        System.out.println("Head :" + head.value);
    }
    public void getTail() {
        System.out.println("Tail : " + tail.value);
    }
    public void getLength() {
        System.out.println("Length  :" +  length);
    }

    public void printList() {
        Node temp =head;
        System.out.println("\n");
        getHead();
        getTail();
```

```
        getLength();
        System.out.println("Items in list :");
        while(temp !=null) {
            System.out.print("--->" +temp.value + " \t");
            temp=temp.next;
        }
    }

    public void append(int value) {
        Node newNode = new Node(value);
        if(length == 0) {

            head=newNode;
            tail=newNode;
        }else {
            tail.next = newNode;
            tail = newNode;
        }
        length++;
    }

    public Node removeLast() {
```

```
        if(length == 0) {
             return null;
        }

        Node temp = head;
        Node pre = head;
        while(temp.next !=null) {
          pre=temp;
          temp=temp.next;
        }

        tail =pre;
        tail.next = null;
        length--;

        if(length == 0) {
             head =null;
             tail =null;
        }
        return temp;
    }
```

```
public void prepend(int value) {
     Node newNode = new Node(value);
     if(length == 0) {

          head=newNode;
          tail=newNode;
     }else {
          newNode.next = head;
          head = newNode;
     }
     length++;
}
public Node removeFirst() {

     if(length==0) {
          return null;
     }
     Node temp =head;
     head=head.next;
     temp.next=null;
     length --;
     if(length==0) {
```

```
            head=null;
            tail=null;
        }
        return temp;
    }

    public Node get(int index) {
        if(index<0 || index >=length) {
            return null;
        }
        Node temp=head;
        for
        (int i=0;i<index;i++) {
            temp=temp.next;
        }
        return temp;

    }
    public boolean set(int index,int value) {
        Node temp =get(index);
        if(temp != null) {
            temp.value=value;
```

```
            return true;
        }
        return false;
    }
    public boolean insert(int index, int value) {
        if(index<0 || index >=length) {
            return false;
        }
        if(index==0) {
            prepend(value);
            return true;
        }
        if(index == length) {
            append(value);
            return true;
        }
        Node newNode = new Node(value);
        Node temp = get(index-1);
        newNode.next=temp.next;
        temp.next=newNode;
        length++;
        return true;
```

```
}

public Node searchMiddle() {
      if (head == null) {
            return null;
        }
      int mid=length/2;
          Node current=get(mid);
          if(length%2==0) {
               current=get(mid-1);
          }

     return current;
}
public static void main(String[] args) {
     LinkedList myll = new LinkedList(11);
     myll.printList();

     myll.append(3);
     myll.append(23);

     myll.printList();
```

```
        System.out.println("Removed " +
myll.removeLast().value);

        myll.printList();

        myll.prepend(1);

        myll.printList();

        System.out.println("Removed " +
myll.removeFirst().value);

        myll.printList();

        System.out.println("Item at index 1 is " +
myll.get(1).value);

        myll.append(3);

        myll.append(7);


        System.out.println("Replace item at index 1
:"+myll.set(1, 33));

        myll.printList();

        System.out.println("\n Insert betweeb 1 and 2: "+
myll.insert(2,20));

        myll.printList();

        System.out.println();

        System.out.println("middle element is :"
+myll.searchMiddle().value);
```

```
    }

}
```

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.

```
import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;


public class QueueSorter {

    public static void sortQueue(Queue<Integer> queue) {

        Stack<Integer> stack = new Stack<>();


        // Get the size of the queue

        int n = queue.size();


        // Outer loop to process all elements

        while (n > 0) {

            int minIndex = -1;
```

```
            int minValue = Integer.MAX_VALUE;

            // Find the minimum element in the queue
            for (int i = 0; i < n; i++) {
                int current = queue.poll();
                if (current < minValue) {
                    minValue = current;
                    minIndex = i;
                }
                queue.offer(current);
            }

            // Remove elements from the queue and push to the
stack except the minimum one
            for (int i = 0; i < n; i++) {
                int current = queue.poll();
                if (i != minIndex) {
                    stack.push(current);
                }
            }

            // Push the minimum element directly to the stack
            stack.push(minValue);
```

```
            // Move elements back from the stack to the queue
            while (!stack.isEmpty()) {
                queue.offer(stack.pop());
            }

            // Reduce the range of unsorted elements
            n--;
        }

        // Now the queue is sorted in reverse order, move back to stack to get correct order
        while (!queue.isEmpty()) {
            stack.push(queue.poll());
        }
        while (!stack.isEmpty()) {
            queue.offer(stack.pop());
        }
    }

    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(3);
```

```
        queue.offer(1);
        queue.offer(4);
        queue.offer(1);
        queue.offer(5);
        queue.offer(9);
        queue.offer(2);
        queue.offer(6);
        queue.offer(5);
        queue.offer(3);
        queue.offer(5);

        System.out.println("Original queue: " + queue);
        sortQueue(queue);
        System.out.println("Sorted queue: " + queue);
    }
}
```

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

```
import java.util.Stack;

public class StackSorter {

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

        while (!stack.isEmpty()) {
            // Pop out the first element
            int current = stack.pop();

            // While temporary stack is not empty and
            // top of tempStack is greater than current
            while (!tempStack.isEmpty() && tempStack.peek() >
current) {
                // pop from tempStack and push it into the input
stack
                stack.push(tempStack.pop());
            }

            // push current into tempStack
            tempStack.push(current);
        }
```

```
        // Transfer the sorted elements back to the original
stack
        while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
        }
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(34);
        stack.push(3);
        stack.push(31);
        stack.push(98);
        stack.push(92);
        stack.push(23);

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

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class LinkedList {

    public static void removeDuplicates(ListNode head) {
        ListNode current = head;

        // Traverse the list until the end
        while (current != null && current.next != null) {
            if (current.val == current.next.val) {
                // Skip the next node if it's a duplicate
                current.next = current.next.next;
            } else {
                // Move to the next node if it's not a duplicate
```

```
            current = current.next;
        }
    }
}

public static void main(String[] args) {
    // Example usage
    ListNode head = new ListNode(1);
    head.next = new ListNode(1);
    head.next.next = new ListNode(2);
    head.next.next.next = new ListNode(3);
    head.next.next.next.next = new ListNode(3);

    System.out.println("Original list:");
    printList(head);

    removeDuplicates(head);

    System.out.println("List after removing duplicates:");
    printList(head);
}
```

```
    public static void printList(ListNode head) {
        ListNode current = head;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println();
    }
}
```

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.

```
import java.util.Stack;


public class StackSequenceSearch {


    public static boolean isSequenceInStack(Stack<Integer>
stack, int[] sequence) {
        if (sequence.length == 0) {
            return true;  // An empty sequence is always present
        }
```

```
        Stack<Integer> tempStack = new Stack<>();
        int seqIndex = 0;

        // Traverse the stack
        while (!stack.isEmpty()) {
            int element = stack.pop();
            tempStack.push(element);

            // Check if the current element matches the current
sequence element
            if (element == sequence[seqIndex]) {
                seqIndex++;
                // Check if the entire sequence has been found
                if (seqIndex == sequence.length) {
                    return true;
                }
            } else {
                // Reset the sequence index if there is a
mismatch
                seqIndex = 0;
            }
        }
```

```
        // Restore the original stack order if needed
        while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
        }

        return false;
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
        stack.push(5);

        int[] sequence1 = {3, 4, 5};
        int[] sequence2 = {2, 4};

        System.out.println("Sequence {3, 4, 5} in stack: " +
isSequenceInStack(stack, sequence1)); // true
        System.out.println("Sequence {2, 4} in stack: " +
```

```
isSequenceInStack(stack, sequence2)); // false
    }
}
```

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).

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}


public class MergeSortedLists {


    public static ListNode mergeTwoLists(ListNode l1, ListNode
l2) {
        // Create a dummy node to act as the start of the merged
list
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;


        // Traverse both lists
```

```
        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) {
                current.next = l1;
                l1 = l1.next;
            } else {
                current.next = l2;
                l2 = l2.next;
            }
            current = current.next;
        }

        // If there are remaining nodes in either list, append
them
        if (l1 != null) {
            current.next = l1;
        } else {
            current.next = l2;
        }

        // The merged list is next to the dummy node
        return dummy.next;
    }
```

```
public static void printList(ListNode head) {
    ListNode current = head;
    while (current != null) {
        System.out.print(current.val + " ");
        current = current.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    // Example usage
    ListNode l1 = new ListNode(1);
    l1.next = new ListNode(2);
    l1.next.next = new ListNode(4);

    ListNode l2 = new ListNode(1);
    l2.next = new ListNode(3);
    l2.next.next = new ListNode(4);

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

```
        System.out.println("List 2:");
        printList(l2);

        ListNode mergedList = mergeTwoLists(l1, l2);

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

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.

```
public class CircularQueueBinarySearch {

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

        while (low <= high) {
            int mid = low + (high - low) / 2;
```

```
            if (arr[mid] == target) {
                return mid;
            }

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

```
        return -1; // Target not found
    }

    public static void main(String[] args) {
        int[] arr = {13, 18, 25, 2, 8, 10};
        int target = 8;

        int result = circularQueueBinarySearch(arr, target);
        if (result != -1) {
            System.out.println("Target found at index: " +
result);
        } else {
            System.out.println("Target not found.");
        }
    }
}
```