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.

Sol:

To find the middle element of a singly linked list in Java with only one traversal and without using extra space, you can use the two-pointer technique. One pointer (let's call it `slow`) will move one step at a time, while the other pointer (let's call it `fast`) will move two steps at a time. When the `fast` pointer reaches the end of the list, the `slow` pointer will be at the middle element.

Program:

```
class ListNode {
   int val;
   ListNode next;
   ListNode(int x) {
      val = x;
      next = null;
   }
}
public class LinkedListMiddleElement {
         public static ListNode findMiddleElement(ListNode head) {
      if (head == null) {
         return null;
      }
      ListNode slow = head;
      ListNode fast = head;
      while (fast != null && fast.next != null) {
         slow = slow.next;
         fast = fast.next.next;
      }
      return slow;
   }
```

```
    public static void main(String[] args) {
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);

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

==========

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.

Sol:

Steps to Sort the Queue

Initialize the Stack:

- Create an empty stack to be used for sorting.

Sorting Process:

- Find the minimum element in the queue and push it onto the stack.
- Remove the found minimum element from the queue.
- Repeat the above steps until the queue is empty.

Restoring Sorted Elements to the Queue:

- Pop elements from the stack back into the queue. Since the stack has elements in sorted order (smallest at the top), the queue will now be sorted.

Program:

```
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<>();


     while (!queue.isEmpty()) {

        int min = Integer.MAX_VALUE;

        int n = queue.size();

        for (int i = 0; i < n; i++) {

           int current = queue.poll();

           if (current < min) {

              min = current;

           }

           queue.offer(current);

        }


        for (int i = 0; i < n; i++) {

           int current = queue.poll();

           if (current != min) {

              stack.push(current);

           }

        }
```

```
            queue.offer(min);

            while (!stack.isEmpty()) {
                queue.offer(stack.pop());
            }
        }

    }
        public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(6);
        queue.offer(4);
        queue.offer(1);
        queue.offer(7);
        queue.offer(3);

        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 is Empty.

Sol:

Steps to Sort the Stack Using an Additional Temporary Stack

1. **Initialize the Temporary Stack**:

- Create an empty temporary stack to hold the sorted elements.

2. **Sorting Process**:

- Pop an element from the original stack.
- While the temporary stack is not empty and the top of the temporary stack is greater than the popped element, move elements from the temporary stack back to the original stack.
- Push the popped element onto the temporary stack.
- Repeat the process until the original stack is empty.

3. **Transfer Sorted Elements Back to Original Stack**:

- After sorting is complete, transfer the elements from the temporary stack back to the original stack so that the smallest items are on top.

Program:

```
import java.util.Stack;

public class StackSorter {

    public static void sortStack(Stack<Integer> stack) {

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


     while (!stack.isEmpty()) {

        int current = stack.pop();


        while (!tempStack.isEmpty() && tempStack.peek() > current) {

           stack.push(tempStack.pop());

        }

       tempStack.push(current);

     }


     while (!tempStack.isEmpty()) {

        stack.push(tempStack.pop());

     }
```

```
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(33);
        stack.push(21);
        stack.push(43);
        stack.push(15);
        stack.push(92);
        stack.push(25);

        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.

Sol:

Steps to Remove Duplicates from a Sorted Linked List

1. **Initialize Pointers**:

- Use a pointer to traverse the linked list, starting from the head.

2. **Traversal and Removal**:

- For each node, check if its value is the same as the next node's value.
- If they are the same, bypass the next node by pointing the current node's next to the node after the next node.
- If they are not the same, move the pointer to the next node.
- Continue this process until you reach the end of the list.

Program:

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

public class RemoveDuplicatesFromSortedList {

    public static ListNode deleteDuplicates(ListNode head) {
        ListNode current = head;

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

        return head;
    }
```

```
    public static void main(String[] args) {
      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);
     head = deleteDuplicates(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.

Sol:

Program:

```
import java.util.Stack;

 public class SequenceInStack {

    public static boolean isSequenceInStack(Stack<Integer> stack, int[] sequence) {

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



       while (!stack.isEmpty()) {

          if (stack.peek() == sequence[0]) {

             tempStack.push(stack.pop());

             if (tempStack.  size() == sequence.length) {

                boolean sequenceFound = true;

                for (int i = sequence.length - 1; i >= 0; i--) {

                   if (tempStack.pop() != sequence[i]) {

                      sequenceFound = false;

                      break;

                   }

                }

                if (sequenceFound)

                   return true;

             }

          } else {
```

```
                tempStack.push(stack.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);
        stack.push(6);
      int[] sequence = {3, 4, 5};
       System.out.println(isSequenceInStack(stack, sequence)); // Output: 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).

Sol:

Program:

```
class ListNode {
    int val;
```

```
    ListNode next;
    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}
public class MergeSortedLinkedLists {
    public static ListNode mergeLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        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 (l1 != null) {
            current.next = l1;
        } else {
```

```
            current.next = l2;
        }
      return dummy.next;
    }
      public static void main(String[] args) {
        ListNode l1 = new ListNode(1);
        l1.next = new ListNode(3);
        l1.next.next = new ListNode(5);

        ListNode l2 = new ListNode(2);
        l2.next = new ListNode(4);
        l2.next.next = new ListNode(6);
        ListNode mergedList = mergeLists(l1, l2);
        while (mergedList != null) {
            System.out.print(mergedList.val + " ");
            mergedList = mergedList.next;
        }
    }
}
```

= = = = = = = = =

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.

Sol:

Program:

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

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

            if (nums[mid] == target) {
                return mid; // Element found
            }

            if (nums[low] <= nums[mid]) {
                if (nums[low] <= target && target < nums[mid]) {
                    high = mid – 1;
                } else {
                    low = mid + 1;
                }
            }
            else {
                if (nums[mid] < target && target <= nums[high]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
```

```
            }
        }


        return -1;
    }
     public static void main(String[] args) {
        int[] nums = {4, 5, 6, 7, 0, 1, 2};
        int target = 0;
        int index = search(nums, target);
        if (index != -1) {
            System.out.println("Element found at index: " + index);
        } else {
            System.out.println("Element not found");
        }
    }
}
```