

## 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.

Ans.

```
public Node findMiddle()
{
    if(head==null)
        return null;

    Node current=head;
    Node current1=head;

    while(current1!=null && current1.next!=null)
    {
        current=current.next;
        current1=current1.next.next;
    }

    return current;
}
```

OutPUT:

```
231     myll.append(3);
232     myll.append(5);
233
234     myll.printList();
235     //myll.removeFirst();
236     //myll.printList();
237     //myll.prepend(3);
238     //System.out.println("Replace item at index 1:"+myll.set(1, 33));
```

<

Problems Javadoc Declaration Console X

<terminated> LinkedList [Java Application] C:\Program Files\Java\jdk-22\bin\javaw.exe (Jun 1, 2024, 11:23:47 PM - 11:23:47 PM)

Node: DS.LinkedList\$Node@4517d9a3

Head: 4

tail: 5

Length: 3

--->4

--->3

--->5

Middle: 3

### 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.

Ans.

1. **Iterate:** Multiple times (n iterations, n being queue size).
2. **Find Minimum:** Dequeue and enqueue all elements, tracking the smallest found (store on stack - optional).
3. **Enqueue Minimum:** Dequeue all again, enqueue the minimum (from stack or comparison), then remaining elements.

Repeat steps 2-3 progressively places the smallest elements at the front, achieving a sorted queue.

**Trade-off:** Using the stack simplifies minimum tracking but might overflow. Not using the stack saves space but requires comparing with the previous minimum.

**Complexity:**

- Time Complexity:  $O(n^2)$ . Each iteration requires dequeuing and enqueueing all elements, leading to a quadratic runtime.
- Space Complexity:  $O(1)$  or  $O(\log n)$  depending on the approach. Without using the stack for minimum tracking, it's constant. Using the stack introduces a space complexity of  $O(\log n)$  in the worst case (when the stack holds all minimum elements during initial iterations for a large 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.

Ans:

```
package Practice2;

import java.util.Stack;

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

        while (!stack.isEmpty()) {
            // Pop the top element from the original stack
            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(34);
stack.push(3);
stack.push(31);
stack.push(98);
stack.push(92);
stack.push(23);

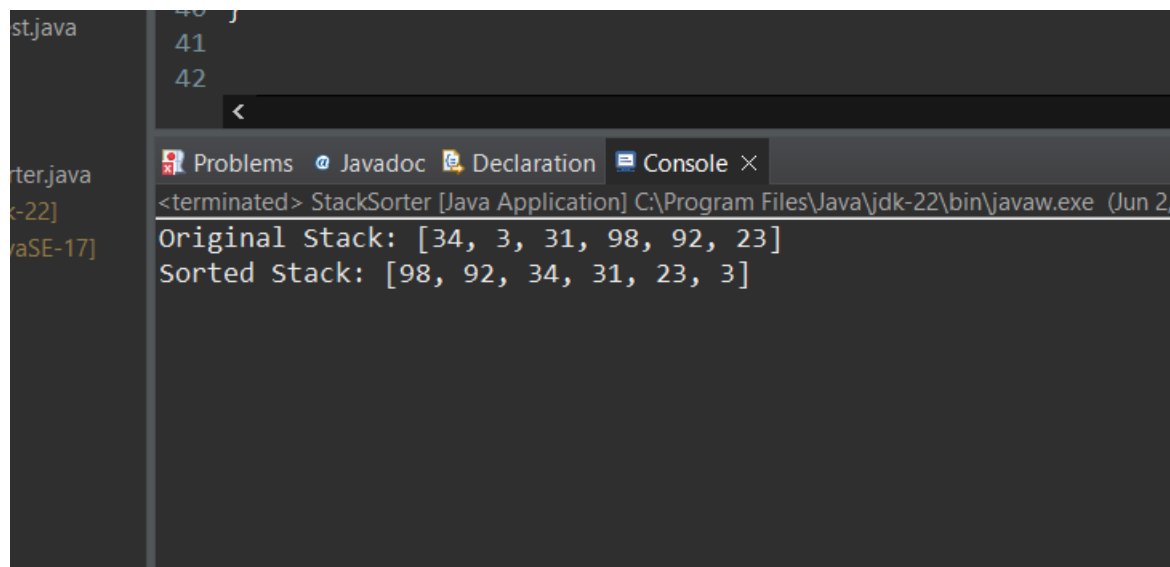
System.out.println("Original Stack: " + stack);

sortStack(stack);

System.out.println("Sorted Stack: " + stack);
}
}

```

OUTPUT:



```

<terminated> StackSorter [Java Application] C:\Program Files\Java\jdk-22\bin\javaw.exe (Jun 2
Original Stack: [34, 3, 31, 98, 92, 23]
Sorted Stack: [98, 92, 34, 31, 23, 3]

```

### 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.

Ans:

```
package List;
```

```

class ListNode1 {
    int val;
    ListNode1 next;
}

```

ListNode1() {}

```
ListNode1(int val) { this.val = val; }  
ListNode1(int val, ListNode1 next) { this.val = val; this.next = next; }  
}
```

```
public class RemoveDuplicates {  
    public ListNode1 removeDuplicates(ListNode1 head) {  
        if (head == null || head.next == null) {  
            return head;  
        }  
        ListNode1 prev = head;  
        ListNode1 curr = head.next;  
  
        while (curr != null) {  
            if (prev.val == curr.val) {  
                // Duplicate found, remove current node  
                prev.next = curr.next;  
            } else {  
                // Move both pointers  
                prev = curr;  
            }  
            curr = curr.next;  
        }  
        return head;  
    }  
}
```

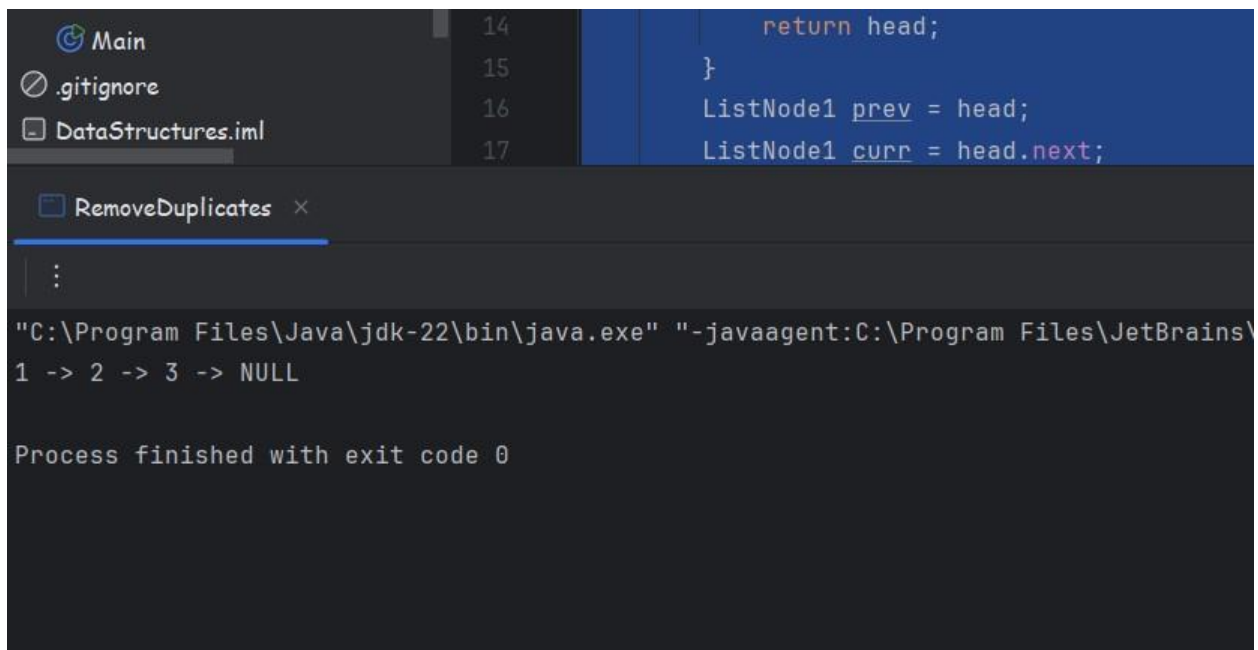
```
public static void main(String[] args) {  
    RemoveDuplicates solution = new RemoveDuplicates();  
  
    // Sample linked list with duplicates  
    ListNode1 head = new ListNode1(1);  
    head.next = new ListNode1(1);  
    head.next.next = new ListNode1(2);  
    head.next.next.next = new ListNode1(3);  
    head.next.next.next.next = new ListNode1(3);  
  
    // Remove duplicates  
    head = solution.removeDuplicates(head);  
  
    // Print the modified list  
    ListNode1 temp = head;  
    while (temp != null) {
```

```

        System.out.print(temp.val + " -> ");
        temp = temp.next;
    }
    System.out.println("NULL");
}
}

```

OutPut:



The screenshot shows an IDE with a file explorer on the left containing 'Main', '.gitignore', and 'DataStructures.iml'. The main editor displays Java code with line numbers 14 to 17. Line 14 is 'return head;', line 15 is '}', line 16 is 'ListNode1 prev = head;', and line 17 is 'ListNode1 curr = head.next;'. Below the editor is a 'Remove Duplicates' dialog box. At the bottom is a terminal window showing the command 'C:\Program Files\Java\jdk-22\bin\java.exe' and its output: '1 -> 2 -> 3 -> NULL' and 'Process finished with exit code 0'.

```

14      return head;
15  }
16  ListNode1 prev = head;
17  ListNode1 curr = head.next;

```

Remove Duplicates x

⋮

```

"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\
1 -> 2 -> 3 -> NULL

Process finished with exit code 0

```

### 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.

Ans:

```
package Stack;
```

```
import java.util.Stack;
```

```
public class StackSequence {
```

```

public static boolean searchSequence(Stack<Integer> stack, int[] sequence) {
    if (stack.isEmpty() || sequence.length > stack.size()) {
        return false;
    }

    int expectedIndex = sequence.length - 1;

    for (int element : stack) {
        if (element == sequence[expectedIndex]) {
            expectedIndex--;
            if (expectedIndex < 0) {
                return true;
            }
        } else {
            expectedIndex = sequence.length - 1; // Reset the search when mismatch occurs
        }
    }

    return false; // Sequence not found if loop finishes
}

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


    int[] sequence1 = {1, 2, 3};
    int[] sequence2 = {4, 2};

    System.out.println("Sequence 1 found: " + searchSequence(stack, sequence1));
    System.out.println("Sequence 2 found: " + searchSequence(stack, sequence2));
}
}

```



OutPut:



The screenshot shows an IDE with a project named 'StackSequence'. The code editor displays the following Java code:

```
14 int expectedIndex = sequence.length - 1;
15
16 for (int element : stack) {
17     if (element == sequence[expectedIndex]) {
```

The output console shows the following text:

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Commur
Sequence 1 found: false
Sequence 2 found: false

Process finished with exit code 0
```

## 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).

Ans:

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

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

```
public class MergeSortedLinkedLists {

    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {

        // Create a dummy node to act as the starting point of the merged list
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        // Traverse both lists and link nodes in ascending order
        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;
        }

        // Link the remaining nodes, if any
        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 main(String[] args) {
        // Create first sorted linked list: 1 -> 3 -> 5
        ListNode l1 = new ListNode(1);
        l1.next = new ListNode(3);
        l1.next.next = new ListNode(5);

        // Create second sorted linked list: 2 -> 4 -> 6
        ListNode l2 = new ListNode(2);
        l2.next = new ListNode(4);
        l2.next.next = new ListNode(6);

        // Merge the two lists
        ListNode mergedList = mergeTwoLists(l1, l2);

        // Print the merged list
        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.

Ans:

```
package Practice2;

public class CircularQueueBinarySearch {
    public static int circularBinarySearch(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;
            }

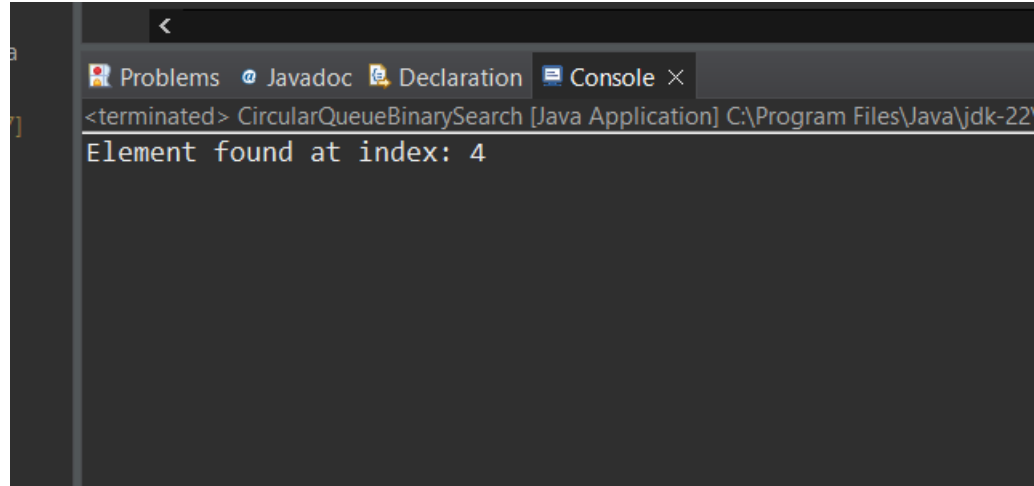
            if (arr[low] <= arr[mid]) {
                if (target >= arr[low] && target < arr[mid]) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            } else {
                if (target > arr[mid] && target <= arr[high]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }
        }

        return -1;
    }

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

```
}  
}
```

Output:



The screenshot shows an IDE's console window with a dark theme. The title bar of the console tab reads "<terminated> CircularQueueBinarySearch [Java Application] C:\Program Files\Java\jdk-22". The console output displays the text "Element found at index: 4". The IDE interface includes tabs for "Problems", "Javadoc", "Declaration", and "Console".

```
<terminated> CircularQueueBinarySearch [Java Application] C:\Program Files\Java\jdk-22  
Element found at index: 4
```