

Searching, Sorting - Assignment

Task 1: Real-time Data Stream Sorting

A stock trading application requires real-time sorting of trade transactions by price. Implement a heap sort algorithm that can efficiently handle continuous incoming data, adding and sorting new trades as they come.

Code:

```
package heapsort;
import java.util.*;

class Trade {
    private String symbol;
    private double price;
    public Trade(String symbol, double price) {
        this.symbol = symbol;
        this.price = price;
    }

    public String getSymbol() {
        return symbol;
    }

    public double getPrice() {
        return price;
    }

    @Override
    public String toString() {
        return "Trade: Symbol=" + symbol + ", Price=" + price;
    }
}
```

```

public class heapSorting {
private static void heapify(Trade[] trades, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && trades[left].getPrice() > trades[largest].getPrice()) {
        largest = left;
    }
    if (right < n && trades[right].getPrice() > trades[largest].getPrice()) {
        largest = right;
    }

    if (largest != i) {
        Trade temp = trades[i];
        trades[i] = trades[largest];
        trades[largest] = temp;

        heapify(trades, n, largest);
    }
}

private static void heapSort(Trade[] trades) {
    int n = trades.length;

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(trades, n, i);
    }

    for (int i = n - 1; i > 0; i--) {
        Trade temp = trades[0];
        trades[0] = trades[i];
        trades[i] = temp;
        heapify(trades, i, 0);
    }
}

```

```
}  
}
```

```
public static void main(String[] args) {  
    Trade[] trades = new Trade[]{  
        new Trade("ABC", 150.50),  
        new Trade("DEF", 2700.75),  
        new Trade("MNO", 300.20),  
        new Trade("PQR", 650.80)  
    };  
    heapSort(trades);  
    for (Trade trade : trades) {  
        System.out.println(trade);  
    }  
}
```

Output:

```
Trade: Symbol=ABC, Price=150.5  
Trade: Symbol=MNO, Price=300.2  
Trade: Symbol=PQR, Price=650.8  
Trade: Symbol=DEF, Price=2700.75
```

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.

Code:

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

```

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

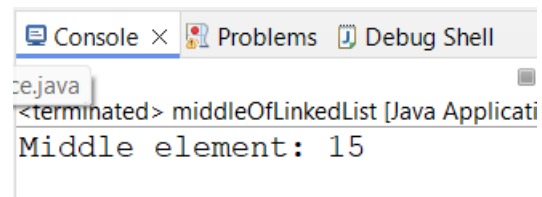
public class middleOfLinkedList {
    public static ListNode findMiddle(ListNode head) {
        if (head == null) {
            return null;
        }
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

    public static void main(String[] args) {
        ListNode head = new ListNode(5);
        head.next = new ListNode(10);
        head.next.next = new ListNode(15);
        head.next.next.next = new ListNode(20);
        head.next.next.next.next = new ListNode(25);

        ListNode middle = findMiddle(head);
        System.out.println("Middle element: " + middle.val);
    }
}

```

Output:



```
<terminated> middleOfLinkedList [Java Applicati
Middle element: 15
```

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.

Code:

```
package queuesorting;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class queueSortingWithLimitedSpace {

    public static void sortQueue(Queue<Integer> queue) {
        Stack<Integer> stack = new Stack<>();
        while (!queue.isEmpty()) {
            int current = queue.poll();
            while (!stack.isEmpty() && stack.peek() < current) {
                queue.offer(stack.pop());
            }
            stack.push(current);
        }
        while (!stack.isEmpty()) {
            queue.offer(stack.pop());
        }
    }

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

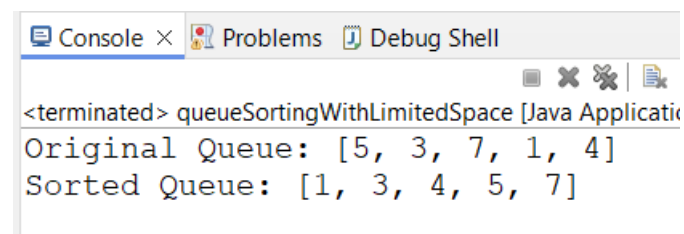
```

queue.offer(5);
queue.offer(3);
queue.offer(7);
queue.offer(1);
queue.offer(4);

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

```

Output:



```

<terminated> queueSortingWithLimitedSpace [Java Applicati
Original Queue: [5, 3, 7, 1, 4]
Sorted Queue: [1, 3, 4, 5, 7]

```

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.

Code:

```

package stacksorting;

import java.util.Stack;

public class stackSortingInPlace {

    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(30);
    stack.push(10);
    stack.push(50);
    stack.push(90);
    stack.push(20);
    stack.push(60);

```

```

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

```

Output:

```

Original Stack: [30, 10, 50, 90, 20, 60]
Sorted Stack: [10, 20, 30, 50, 60, 90]

```

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.

Code:

```
package removeduplicatesfromsortedlist;
```

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

```
public class RemoveDuplicatesFromSortedList {
```

```
    public static ListNode removeDuplicates(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 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) {
```



```

ListNode head = new ListNode(10);
head.next = new ListNode(10);
head.next.next = new ListNode(20);
head.next.next.next = new ListNode(20);
head.next.next.next.next = new ListNode(30);
head.next.next.next.next.next = new ListNode(40);
head.next.next.next.next.next.next = new ListNode(40);
head.next.next.next.next.next.next.next = new ListNode(50);

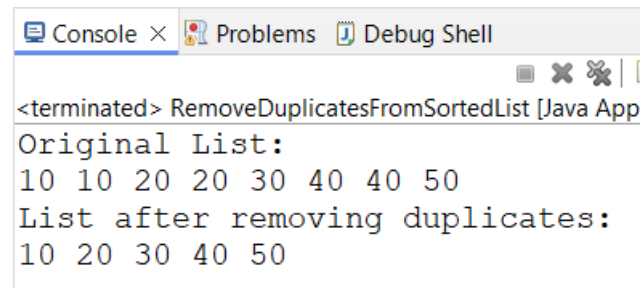
```

```

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

```

Output:



```

<terminated> RemoveDuplicatesFromSortedList [Java App
Original List:
10 10 20 20 30 40 40 50
List after removing duplicates:
10 20 30 40 50

```

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.

Code:

```

package searchingsequenceinstack;
import java.util.Stack;
public class searchingSequenceInStack {

```

```

public static boolean isSequenceInStack(Stack<Integer> stack, int[] sequence) {
    int seqIndex = sequence.length-1;
    boolean isSequenceFound = false;
    while (!stack.isEmpty()) {
        int current = stack.pop();
        if (seqIndex >=0 && current == sequence[seqIndex]) {
            seqIndex--;
        } else {
            seqIndex = sequence.length-1;
        }
        if (seqIndex < 0) {
            isSequenceFound = true;
            break;
        }
    }
    return isSequenceFound;
}

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);
    stack.push(7);
    int[] sequence = {4, 5, 6};
    System.out.println("Stack: " + stack);
    System.out.println("Sequence: " + java.util.Arrays.toString(sequence));
    boolean result = isSequenceInStack(stack, sequence);
    System.out.println("Is the sequence present? " + result);
}

```

Output:

```
Console × Problems Debug Shell
<terminated> searchingSequenceInStack [Java Applicat
Stack: [1, 2, 3, 4, 5, 6, 7]
Sequence: [4, 5, 6]
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).

Code:

```
package mergetwosortedlinkedlists;
```

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

```
public class mergeTwoSortedLinkedLists {
```

```
    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        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 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) {
    ListNode l1 = new ListNode(10);
    l1.next = new ListNode(30);
    l1.next.next = new ListNode(50);

    ListNode l2 = new ListNode(20);
    l2.next = new ListNode(40);
    l2.next.next = new ListNode(60);

    System.out.println("List 1:");
    printList(l1);
    System.out.println("List 2:");
    printList(l2);
}

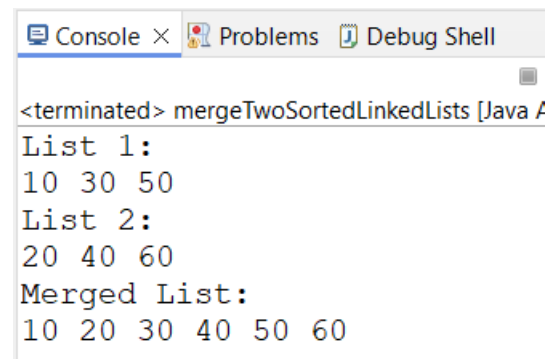
```

```

ListNode mergedList = mergeTwoLists(l1, l2);
System.out.println("Merged List:");
printList(mergedList);
}
}

```

Output:



```

<terminated> mergeTwoSortedLinkedLists [Java /
List 1:
10 30 50
List 2:
20 40 60
Merged List:
10 20 30 40 50 60

```

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.

Code:

```

package circularqueuebinarysearch;

public class circularQueueBinarySearch {

    public static int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right]) {
                left = mid + 1;
            }
        }
    }
}

```

```

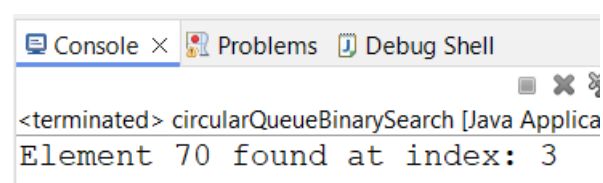
    } else {
        right = mid;
    }
}

int rotationIndex = left;
left = 0;
right = nums.length - 1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    int rotatedMid = (mid + rotationIndex) % nums.length;
    if (nums[rotatedMid] == target) {
        return rotatedMid;
    } else if (nums[rotatedMid] < target) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
return -1;
}

public static void main(String[] args) {
    int[] nums = {40, 50, 60, 70, 10, 20, 30};
    int target = 70;
    int index = search(nums, target);
    System.out.println("Element " + target + " found at index: " + index);
}
}

```

Output:



The screenshot shows an IDE window with three tabs: 'Console', 'Problems', and 'Debug Shell'. The 'Console' tab is active and displays the output of the program. The output consists of two lines: the first line is a prompt '<terminated> circularQueueBinarySearch [Java Applica' and the second line is 'Element 70 found at index: 3'.

```

<terminated> circularQueueBinarySearch [Java Applica
Element 70 found at index: 3

```