Day 6

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.

To implement a heap sort algorithm that can efficiently handle continuous incoming data in a stock trading application, we can use a data structure known as a priority queue (min-heap or max-heap).

Here's how we can implement this:

1. **Min-Heap for Sorting Trades by Price:** We'll use a min-heap to keep the trades sorted by price in real-time. Every new trade will be added to this heap.
2. **Inserting Trades:** As new trades come in, they will be inserted into the min-heap.
3. **Retrieving Sorted Trades:** At any point, we can retrieve the sorted trades by continuously popping elements from the heap.

```
import java.util.PriorityQueue;

import java.util.Comparator;


class Trade {

   double price;

   String details;


   public Trade(double price, String details) {
```

```
        this.price = price;
        this.details = details;
    }

    public double getPrice() {
        return price;
    }

    public String getDetails() {
        return details;
    }

    @Override
    public String toString() {
        return "Trade{price=" + price + ", details='" + details + "'}";
    }
}

class TradeHeap {
    private PriorityQueue<Trade> minHeap;

    public TradeHeap() {
        minHeap = new PriorityQueue<>(Comparator.comparingDouble(Trade::getPrice));
    }
```

```
    public void addTrade(Trade trade) {
        minHeap.add(trade);
    }

    public Trade peekTopTrade() {
        return minHeap.peek();
    }

    public Trade pollTopTrade() {
        return minHeap.poll();
    }

    public void getSortedTrades() {
        while (!minHeap.isEmpty()) {
            System.out.println(minHeap.poll());
        }
    }
}

public class Main {
    public static void main(String[] args) {
        TradeHeap tradeHeap = new TradeHeap();

        tradeHeap.addTrade(new Trade(100.0, "Trade1"));
        tradeHeap.addTrade(new Trade(95.0, "Trade2"));
```

```
        tradeHeap.addTrade(new Trade(110.0, "Trade3"));
        tradeHeap.addTrade(new Trade(105.0, "Trade4"));
        System.out.println("Top trade: " + tradeHeap.peekTopTrade());
        System.out.println("All sorted trades:");
        tradeHeap.getSortedTrades();
    }
}
```

Explanation:

1. **Trade Class:** This class represents a trade with price and details attributes. It includes a constructor, getters, and a toString method for easy printing.
2. **TradeHeap Class:** This class contains a PriorityQueue (min-heap) to manage the trades.
    - The TradeHeap constructor initializes the PriorityQueue with a custom comparator to sort trades by price.
    - The addTrade method adds a new trade to the heap.
    - The peekTopTrade method returns the trade with the smallest price without removing it.
    - The pollTopTrade method removes and returns the trade with the smallest price.
    - The getSortedTrades method removes and prints all trades in sorted order.
3. **Main Class:** This class demonstrates how to use the TradeHeap class.
    - Trades are added to the TradeHeap.
    - The top trade is peeked at and printed.
    - All trades are printed in sorted order by continuously polling from the heap.

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.

To find the middle element of a singly linked list in one traversal and without using extra space, you can use **the "Tortoise and Hare" algorithm**. This approach uses two pointers: one (the "slow" pointer) moves one step at a time, and the other (the "fast" pointer) moves two steps at a time. When the fast pointer reaches the end of the list, the slow pointer will be at the middle element.

```
public class LinkedListMiddle {

 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; }
```

findMiddle Method:

- Takes the head of the linked list as input.
- Initializes two pointers, slow and fast, both pointing to the head.
- Iterates through the list with slow moving one step at a time and fast moving two steps at a time.

- When fast reaches the end of the list (or the node before the end if the list has an even number of elements), slow will be at the middle node.
- Returns the slow pointer, which now points to the middle element.

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.

Steps to Sort a Queue using One Additional Stack

1. **Initialize Structures:**
    - Let queue be your input queue.
    - Initialize an empty stack stack.
2. **Iterate through the Queue:**
    - While the queue is not empty:
        1. Remove the front element from the queue.
        2. While the stack is not empty and the top element of the stack is greater than the front element:
            - Pop the top element from the stack and enqueue it back to the queue.
        3. Push the front element onto the stack.
3. **Move Elements Back to Queue:**
    - Once all elements are processed and the queue is empty, move all elements from the stack back to the 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.

To sort a stack such that the smallest items are on the top using an additional temporary stack, you can follow these steps:

1. **Initialize Structures:**
    - Let originalStack be the input stack.
    - Initialize an empty stack tempStack.
2. **Sorting Process:**
    - While the originalStack is not empty:
        1. Pop the top element from the originalStack.
        2. While the tempStack is not empty and the top element of the tempStack is greater than the popped element:
            - Pop the top element from the tempStack and push it back to the originalStack.
        3. Push the popped element onto the tempStack.
3. **Move Elements Back to Original Stack:**
    - Once the originalStack is empty and the elements in the tempStack are sorted in descending order (smallest at the bottom), move all elements back from the tempStack to the originalStack.

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.

Initialize Pointers:

- Use a pointer (current) to traverse the list, starting from the head of the list.

Traverse the List:

- While current is not null and current.next is not null:
    - Compare current node's value with current.next node's value.

- If they are the same, skip the next node by changing the current.next pointer to current.next.next.
- If they are different, move the current pointer to the next node.

End of List:

- Once the traversal is complete, the linked list will have all duplicates removed.

```
public class RemoveDuplicates
{
public static ListNode removeDuplicates(ListNode head)
 {
 // Initialize the current pointer
 ListNode current = head;
 // Traverse the list
while (current != null && current.next != null)
{ if (current.val == current.next.val)
 {                                                    // Skip the duplicate node
          current.next = current.next.next; }
 else {                                               // Move to the next node
 current = current.next; }
}
return head;
}
```

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.

Initialize Structures:

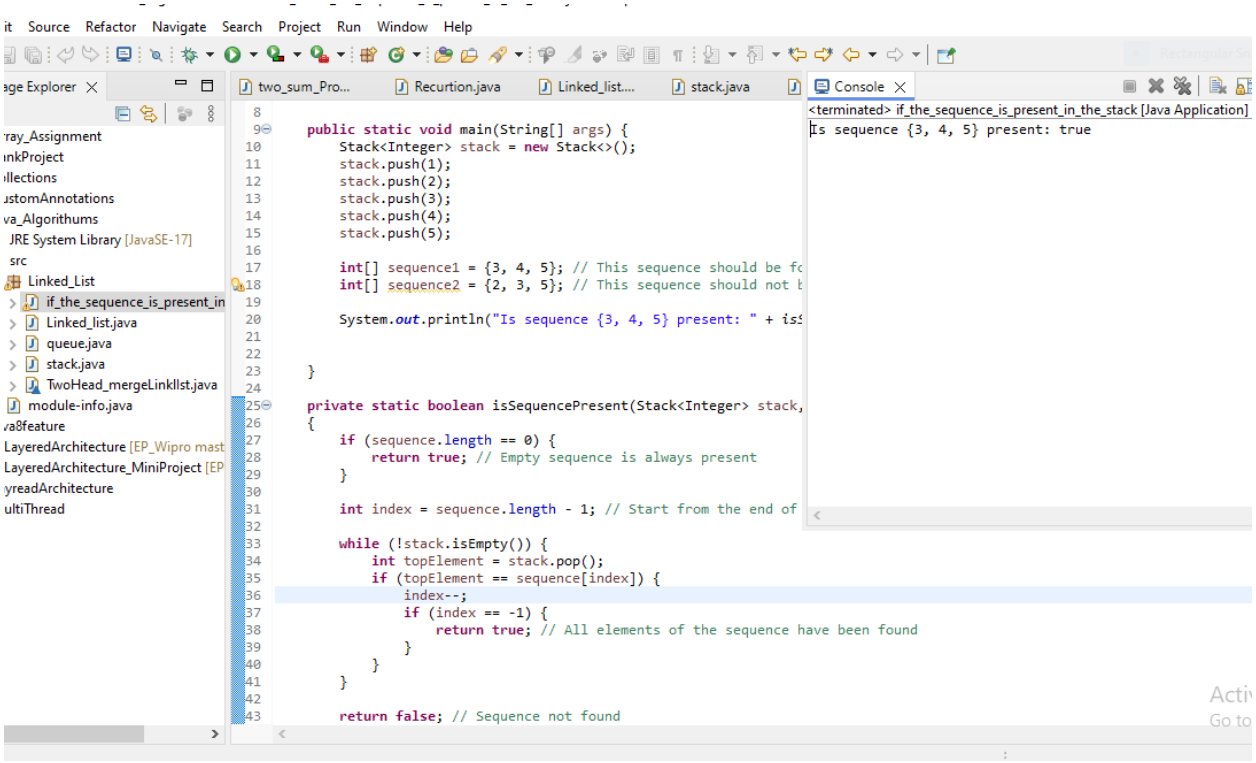
- Use a Stack for the main stack.
- Use an integer index to track the position in the sequence array starting from the last element.

Iterate Through the Stack:

- While the stack is not empty:
    - Pop an element from the stack.
    - Compare the popped element with the current element in the sequence array (tracked by index).
    - If they match, decrement index.
    - If index becomes -1, it means all elements of the sequence have been found consecutively in the stack.

End of Iteration:

- After the iteration, if index is -1, return true, indicating that the sequence is present in the stack.
- Otherwise, return 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).

To merge two sorted linked lists into one sorted linked list without using any extra space (i.e., reusing the nodes from the original lists), you can follow these steps:

1. **Initialize Pointers:**
    - Use two pointers, l1 and l2, to traverse the two input linked lists.
2. **Merge Process:**

- Compare the nodes pointed by l1 and l2.
- Attach the smaller node to the merged list and move the respective pointer forward.
- Repeat until one of the lists is exhausted.

3. **Append Remaining Nodes:**
    - Attach the remaining nodes from the non-exhausted list to the merged list.

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.

Explanation:

1. **ListNode Class:**
    - Represents a node in the linked list with an integer value val and a reference to the next node next.
2. **mergeTwoLists Method:**
    - Uses a dummy node to simplify the edge cases.
    - Iterates through both input lists, comparing nodes and appending the smaller node to the merged list.
    - After exiting the loop, attaches any remaining nodes from either list to the merged list.
    - Returns the merged list, which starts from dummy.next.
3. **printList Method:**
    - Utility method to print the linked list.
4. **main Method:**
    - Demonstrates creating two sorted linked lists, merging them, and printing the merged list.