

Manjula Nannuri

Day-6_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.

```
package sortings;

import java.util.Comparator;
import java.util.PriorityQueue;

class Trade {

    private String tradeId;

    private double price;

    public Trade(String tradeId, double price) {

        this.tradeId = tradeId;

        this.price = price;

    }

    public String getTradeId() {

        return tradeId;

    }

    public double getPrice() {

        return price;

    }

    @Override

    public String toString() {

        return "Trade{tradeId='" + tradeId + "', price=" + price + "'}";

    }

}

public class RealTimeTradeProcessor_Test {

    private PriorityQueue<Trade> minHeap;
```

```

public RealTimeTradeProcessor_Test() {

    this.minHeap = new
    PriorityQueue<>(Comparator.comparingDouble(Trade::getPrice));

}

public void addTrade(Trade trade) {

    minHeap.add(trade);

}

public Trade getHighestPriorityTrade() {

    if (minHeap.isEmpty()) {

        throw new IllegalStateException("No trades available");

    }

    return minHeap.poll();

}

public void displayAllTrades() {

    if (minHeap.isEmpty()) {

        System.out.println("No trades available");

        return;

    }

    System.out.println("All trades in ascending order of price:");

    PriorityQueue<Trade> copy = new PriorityQueue<>(minHeap);

    while (!copy.isEmpty()) {

        System.out.println(copy.poll());

    }

}

public static void main(String[] args) {

    RealTimeTradeProcessor_Test tradeProcessor = new
    RealTimeTradeProcessor_Test();

    // Adding some trades

    tradeProcessor.addTrade(new Trade("T1", 100.5));

```

```

tradeProcessor.addTrade(new Trade("T2", 150.75));

tradeProcessor.addTrade(new Trade("T3", 99.99));

tradeProcessor.addTrade(new Trade("T4", 200.0));

System.out.println("Initial state of the trade queue:");

tradeProcessor.displayAllTrades();

// Processing trades based on priority

System.out.println("\nProcessing trades:");

while (!tradeProcessor.minHeap.isEmpty()) {

    System.out.println("Processing: " +
        tradeProcessor.getHighestPriorityTrade());

}

System.out.println("\nFinal state of the trade queue:");

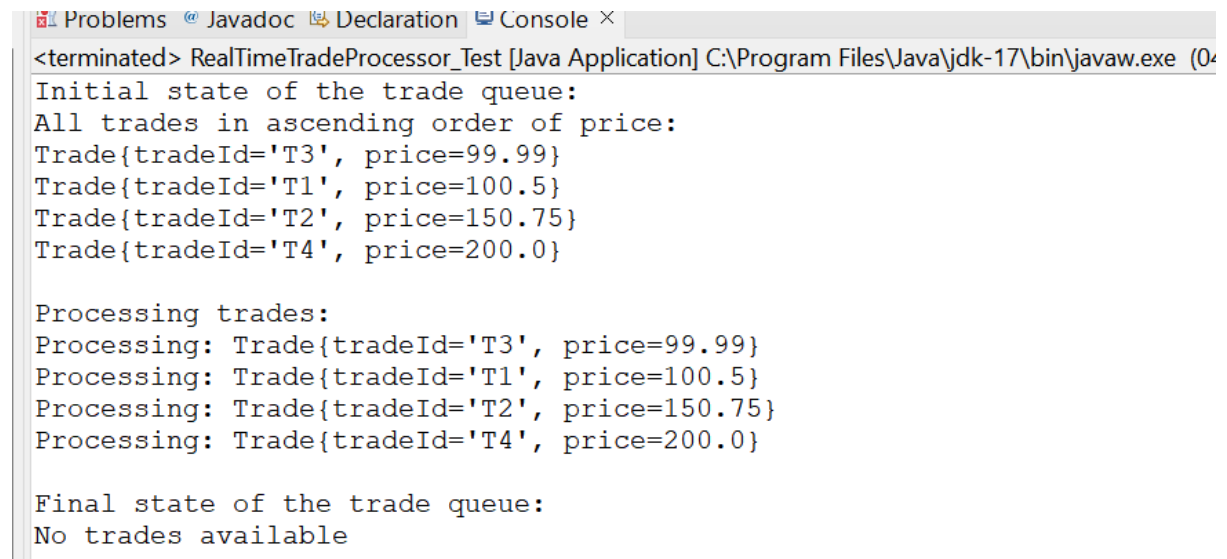
tradeProcessor.displayAllTrades();

}

}

```

OUTPUT:



```

<terminated> RealTimeTradeProcessor_Test [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (0
Initial state of the trade queue:
All trades in ascending order of price:
Trade{tradeId='T3', price=99.99}
Trade{tradeId='T1', price=100.5}
Trade{tradeId='T2', price=150.75}
Trade{tradeId='T4', price=200.0}

Processing trades:
Processing: Trade{tradeId='T3', price=99.99}
Processing: Trade{tradeId='T1', price=100.5}
Processing: Trade{tradeId='T2', price=150.75}
Processing: Trade{tradeId='T4', price=200.0}

Final state of the trade queue:
No trades available

```

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 sortings;
```

```
class LinkedList {

    int val;

    LinkedList next;

    LinkedList(int val) {

        this.val = val;

        this.next = null;

    }

    public static LinkedList findMiddle(LinkedList head) {

        if (head == null) {

            return null;

        }

        LinkedList slow = head;

        LinkedList fast = head;

        while (fast != null && fast.next != null) {

            slow = slow.next;

            fast = fast.next.next;

        }

        return slow;

    }

    public static void main(String[] args) {

        LinkedList head = new LinkedList(1);

        head.next = new LinkedList(2);

        head.next.next = new LinkedList(3);

        head.next.next.next = new LinkedList(4);

        head.next.next.next.next = new LinkedList(5);

        LinkedList middle = findMiddle(head);

        if (middle != null) {

            System.out.println("The middle element is: " + middle.val);

        }

    }

}
```

```

} else {

System.out.println("The linked list is empty.");

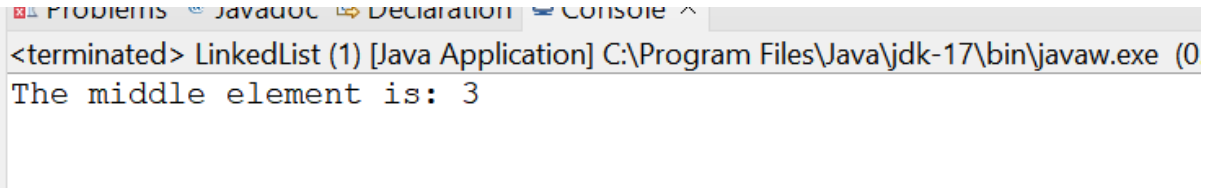
}

}

}

```

OUTPUT:



```

<terminated> LinkedList (1) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (0
The middle element is: 3

```

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 the Queue using One Stack:

1. **Initialize a Stack:** Create an empty stack to use as auxiliary space.
2. **Extract Elements from the Queue:** Remove elements from the queue one by one and insert them into the stack while maintaining sorted order in the stack.
3. **Insert in Sorted Order:**
 - If the stack is empty or the top element of the stack is less than or equal to the current element from the queue, push the element onto the stack.
 - If the top element of the stack is greater than the current element, pop elements from the stack back into the queue until you find the correct position for the current element, then push the current element onto the stack.
4. **Move Sorted Elements back to the Queue:** Once all elements are processed and the stack contains the elements in sorted order, transfer them 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.

```
package sortings;

import java.util.Stack;

public class StackSorting {

    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(5);

        stack.push(3);

        stack.push(8);

        stack.push(1);

        stack.push(4);

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

        sortStack(stack);

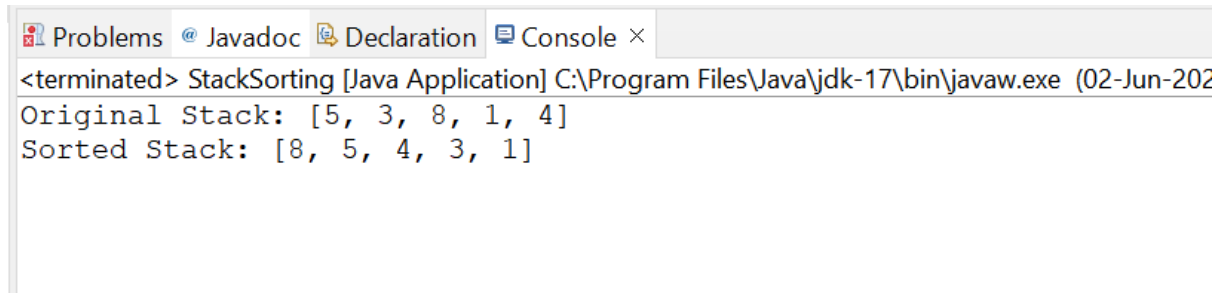
        System.out.println("Sorted Stack: " + stack);

    }

}
```

```
}
```

OUTPUT:



```
<terminated> StackSorting [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (02-Jun-2022)
Original Stack: [5, 3, 8, 1, 4]
Sorted Stack: [8, 5, 4, 3, 1]
```

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.

```
package sortings;

class ListNode {

    int val;

    ListNode next;

    ListNode(int val) {

        this.val = val;

        this.next = null;

    }

    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(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);
    head.next.next.next.next.next = new ListNode(4);

    System.out.println("Original List:");

    printList(head);

    ListNode newList = removeDuplicates(head);

    System.out.println("List after removing duplicates:");

    printList(newList);
}
}

```

OUTPUT:


```
<terminated> ListNode [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (02-Jun-2023 14:02:00)
Original List:
1 1 2 3 3 4
List after removing duplicates:
1 2 3 4
```

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.

```
package sortings;

import java.util.Stack;

public class StackSequenceChecker {

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

        if (sequence.length == 0) {

            return true;

        }

        Stack<Integer> originalStack = (Stack<Integer>) stack.clone();

        while (!originalStack.isEmpty()) {

            Stack<Integer> tempStack = (Stack<Integer>) originalStack.clone();

            int current = tempStack.pop();

            if (current == sequence[0]) {

                boolean sequenceFound = true;

                for (int i = 1; i < sequence.length; i++) {

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

                        sequenceFound = false;

                        break;

                    }

                }

            }

        }

    }

}
```

```

    if (sequenceFound) {

        return true;

    }

}

originalStack.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, 3, 5};

    System.out.println("Sequence 1 present in stack: " +
        isSequencePresent(stack, sequence1));

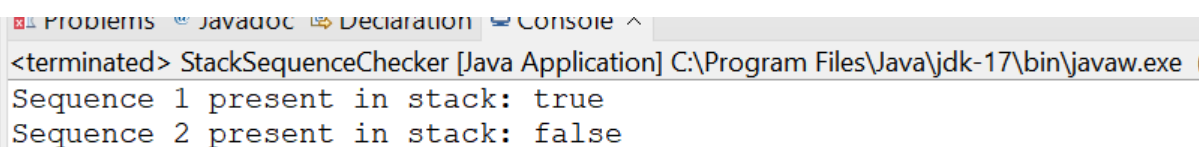
    System.out.println("Sequence 2 present in stack: " +
        isSequencePresent(stack, sequence2));

}

}

```

OUTPUT:



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the Java application. The output consists of two lines: "Sequence 1 present in stack: true" and "Sequence 2 present in stack: false". The first line is highlighted in blue, and the second line is highlighted in yellow.

```

<terminated> StackSequenceChecker [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe
Sequence 1 present in stack: true
Sequence 2 present in stack: 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).

```
package sortings;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class MergingSort {

    int key;

    MergingSort next;

    public MergingSort(int key) {

        this.key = key;

        next = null;

    }

    public static MergingSort newNode(int key) {

        return new MergingSort(key);

    }

    public static void main(String[] args) {

        MergingSort a = new MergingSort(5);

        a.next = new MergingSort(10);

        a.next.next = new MergingSort(15);

        a.next.next.next = new MergingSort(40);

        MergingSort b = new MergingSort(2);

        b.next = new MergingSort(3);

        b.next.next = new MergingSort(20);

    }

}
```

```

List<Integer> v = new ArrayList<>();

while (a != null) {

v.add(a.key);

a = a.next;

}

while (b != null) {

v.add(b.key);

b = b.next;

}

Collections.sort(v);

MergingSort result = new MergingSort(-1);

MergingSort temp = result;

for (int i = 0; i < v.size(); i++) {

result.next = new MergingSort(v.get(i));

result = result.next;

}

temp = temp.next;

System.out.print("Resultant Merge Linked List is : ");

while (temp != null) {

System.out.print(temp.key + " ");

temp = temp.next;

}

}

}

```

OUTPUT:

```

<terminated> MergingSort [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (02-Jun-2024, 1:39:08 pm - 1:3
Resultant Merge Linked List is : 2 3 5 10 15 20 40

```

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.

```
package sortings;

public class CircularQueue {

    public static int circularBinarySearch(int[] arr, int target) {

        int start = 0;

        int end = arr.length - 1;

        while (start <= end) {

            int mid = start + (end - start) / 2;

            if (arr[mid] == target) {

                return mid;

            }

            if (arr[start] <= arr[mid]) {

                if (arr[start] <= target && target < arr[mid]) {

                    end = mid - 1;

                } else {

                    start = mid + 1;

                }

            }

            else {

                if (arr[mid] < target && target <= arr[end]) {

                    start = mid + 1;

                } else {

                    end = mid - 1;

                }

            }

        }

    }

}
```

```

}

return -1;

}

public static void main(String[] args) {

    int[] circularQueue = {4, 5, 6, 7, 0, 1, 2}; // Example circular queue

    int target = 6;

    int index = circularBinarySearch(circularQueue, target);

    if (index != -1) {

        System.out.println("Element " + target + " found at index " + index + " in
the circular queue.");

    } else {

        System.out.println("Element " + target + " not found in the circular
queue.");

    }

}

}

}

```

OUTPUT:

```

<terminated> CircularQueue [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (02-Jun-
Element 6 found at index 2 in the circular queue.

```