Assignment Day-6

Core Java with DS and Algorithms

Name : Joshnitha Rangolu

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 day_6;

import java.util.PriorityQueue;

public class TradeStreamSorter {

public static class Trade {

double price;

}

private PriorityQueue<Trade> heap = new PriorityQueue<>((t1, t2) ->
Double.compare(t1.price, t2.price));

public void addTrade(Trade trade) {

heap.add(trade);

}

public Trade getCheapestTrade() {

return heap.peek();

}

public void iterateSortedTrades() {

while (!heap.isEmpty()) {

Trade trade = heap.poll();

System.out.println("Trade Price: " + trade.price);

heapify(heap);

}
```

```
}
private static void heapify(PriorityQueue<Trade> heap) {
int n = heap.size();
for (int i = 0; i < n / 2; i++) {
siftDown(heap, i);
}
}
private static void siftDown(PriorityQueue<Trade> heap, int i) {
int left = 2 * i + 1;
int right = 2 * i + 2;
int smallest = i;
if (left < heap.size() && heap.peek().price < heap.peek().price) {
smallest = left;
}
if (right < heap.size() && heap.peek().price < heap.peek().price) {
smallest = right;
}
if (smallest != i) {
Trade temp = heap.poll();
heap.add(temp);
siftDown(heap, smallest);
}
}
public static void main(String[] args) {
TradeStreamSorter sorter = new TradeStreamSorter();
Trade trade1 = new Trade();
trade1.price = 100.50;
sorter.addTrade(trade1);
```

```
Trade trade2 = new Trade();

trade2.price = 125.75;

sorter.addTrade(trade2);

Trade trade3 = new Trade();

trade3.price = 88.25;

sorter.addTrade(trade3);

Trade cheapestTrade = sorter.getCheapestTrade();

System.out.println("Cheapest Trade Price: " + cheapestTrade.price);

sorter.iterateSortedTrades();

}

}
```

Parent.java | Main.java | ERPriorityQ... | Patient.java | StackDemo.java | SetDemo.java | Employee.java | Test.java | TradeStreamS...

```
1  package day_6;
2
3  import java.util.PriorityQueue;
4
5  public class TradeStreamSorter {
6
7      public static class Trade {
8          double price;
9      }
10     private PriorityQueue<Trade> heap = new PriorityQueue<>((t1, t2) -> Double.compare(t1.price, t2.price));
11     public void addTrade(Trade trade) {
12       heap.add(trade);
13     }
14     public Trade getCheapestTrade() {
15       return heap.peek();
16     }
17     public void iterateSortedTrades() {
18       while (!heap.isEmpty()) {
19         Trade trade = heap.poll();
20         System.out.println("Trade Price: " + trade.price);
21         heapify(heap);
22       }
```

Console

```
<terminated> TradeStreamSorter [Java Application] C:\Users\DELL\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\java
Cheapest Trade Price: 88.25
Trade Price: 88.25
Trade Price: 100.5
Trade Price: 125.75
```

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.

```
package day_6;

public class Node {

int data;
```

```
Node next;

public Node(int data) {

this.data = data;

}

public static Node findMiddleElement(Node head) {

if (head == null || head.next == null) {

return head;

}

Node slow = head;

Node fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

public static void main(String[] args) {

// Sample linked list

Node head = new Node(1);

head.next = new Node(2);

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

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

Node middleNode = findMiddleElement(head);

if (middleNode != null) {

System.out.println("Middle element: " + middleNode.data);

}

}

}
```

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:

1. **Dequeue and find minimum:**
   - Dequeue elements from the queue one by one.
   - For each dequeued element, compare it with the current minimum element (initialize as the first dequeued element).
   - If the dequeued element is smaller, update the minimum element.
   - After iterating through the entire queue, you'll have the minimum element.
2. **Push minimum to stack:**
   - Push the minimum element you found onto a temporary stack (this is our "limited space").
3. **Enqueue remaining elements:**
   - Enqueue all the elements you dequeued back into the original queue (in the order they were dequeued).
4. **Repeat steps 1-3:**
   - Repeat steps 1-3 until the queue becomes empty. By the end of each iteration, you'll find the minimum element and push it onto the stack.
5. **Dequeue from stack and enqueue:**
   - Dequeue elements from the stack one by one (remember, the stack holds elements in sorted order from smallest to largest).
   - Enqueue each dequeued element (from the stack) back into the original queue.

Explanation:

- This approach utilizes the limited space (stack) to temporarily store the minimum element during each iteration.
- Since the stack follows a LIFO (Last-In-First-Out) principle, when we dequeue elements from it, they come out in ascending order (smallest to largest).
- By repeatedly finding the minimum and pushing it to the stack, and then adding the remaining elements back to the queue, we essentially separate the smallest element from the rest in each pass.
- Finally, by dequeuing from the stack and enqueueing back to the queue, we effectively build the sorted queue from the bottom up.

Time Complexity: O(n^2)

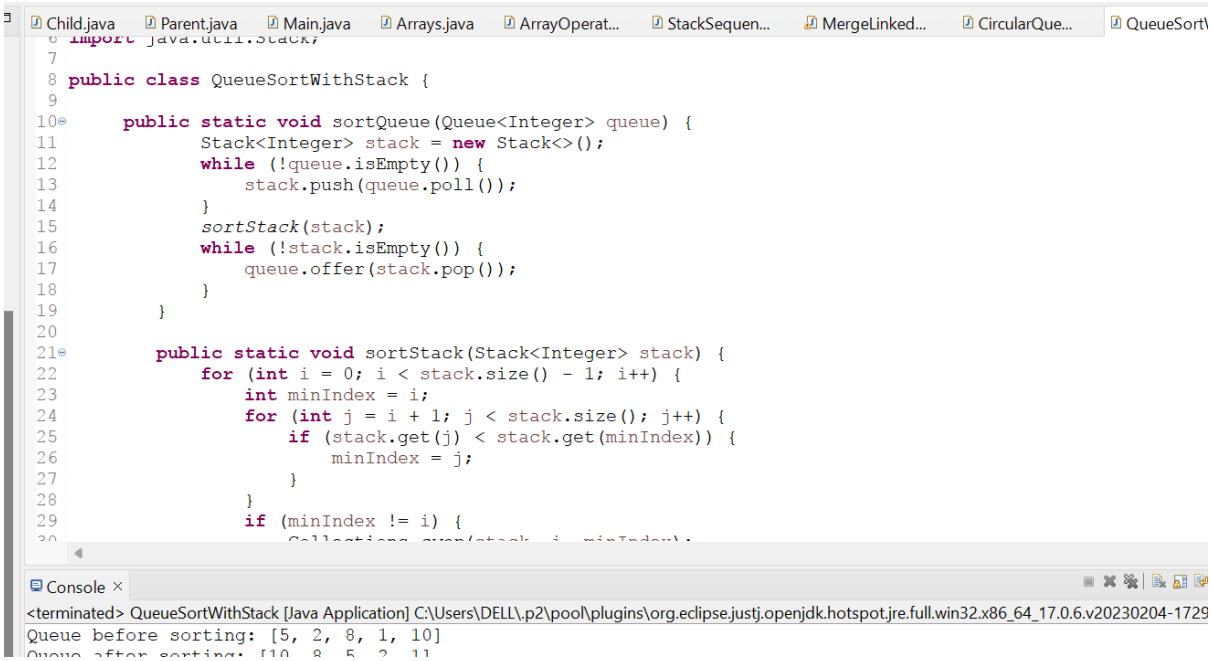
- In the worst case, for each element, we need to iterate through the entire queue to find the minimum. This happens n times, resulting in O(n^2) complexity.

Space Complexity: O(1)

- We only use a constant amount of extra space for the stack (assuming it holds a fixed number of elements).

```
package day_6;

import java.util.Collections;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;

public class QueueSortWithStack {

public static void sortQueue(Queue<Integer> queue) {

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

while (!queue.isEmpty()) {

stack.push(queue.poll());

}

sortStack(stack);

while (!stack.isEmpty()) {

queue.offer(stack.pop());

}

}

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

```
for (int i = 0; i < stack.size() - 1; i++) {
int minIndex = i;
for (int j = i + 1; j < stack.size(); j++) {
if (stack.get(j) < stack.get(minIndex)) {
minIndex = j;
}
}
if (minIndex != i) {
Collections.swap(stack, i, minIndex);
}
}
}
public static void main(String[] args) {
Queue<Integer> queue = new LinkedList<>();
queue.offer(5);
queue.offer(2);
queue.offer(8);
queue.offer(1);
queue.offer(10);
System.out.println("Queue before sorting: " + queue);
sortQueue(queue);
System.out.println("Queue after sorting: " + 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 day_6;

import java.util.Stack;

public class StackSorter {

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

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

while (!stack.isEmpty()) {

int temp = stack.pop();

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

stack.push(tempStack.pop());

}

tempStack.push(temp);

}
```

```
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(50);

stack.push(8);

stack.push(20);

stack.push(15);

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

sortStack(stack);

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

System.out.println("top element: "+stack.peek());

}

}
```

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 day_6;

import java.util.Scanner;

public class Node1 {

int data;

Node next;

public void Node(int data) {

this.data = data;

}

public static Node removeDuplicates(Node head) {

if (head == null || head.next == null) {

return head;

}

Node current = head;

Node prev = null;

while (current != null) {

if (prev != null && prev.data == current.data) {

prev.next = current.next;

} else {

// Update prev for next iteration (only for unique nodes)

prev = current;

}

current = current.next;

}

return head;

}
```

```
public static Node createLinkedList() {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter comma-separated elements for the sorted linked
list ('-1' to quit): ");

String input = scanner.nextLine();

if (input.equalsIgnoreCase("-1")) {

return null;

}

String[] elements = input.split(",");

Node head = null;

Node current = null;

for (String element : elements) {

Node newNode = new Node(Integer.parseInt(element.trim()));

if (head == null) {

head = newNode;

} else {

current.next = newNode;

}

current = newNode;

}

return head;

}

public static void printList(Node head) {

Node temp = head;

while (temp != null) {

System.out.print(temp.data + " -> ");

temp = temp.next;

}
```

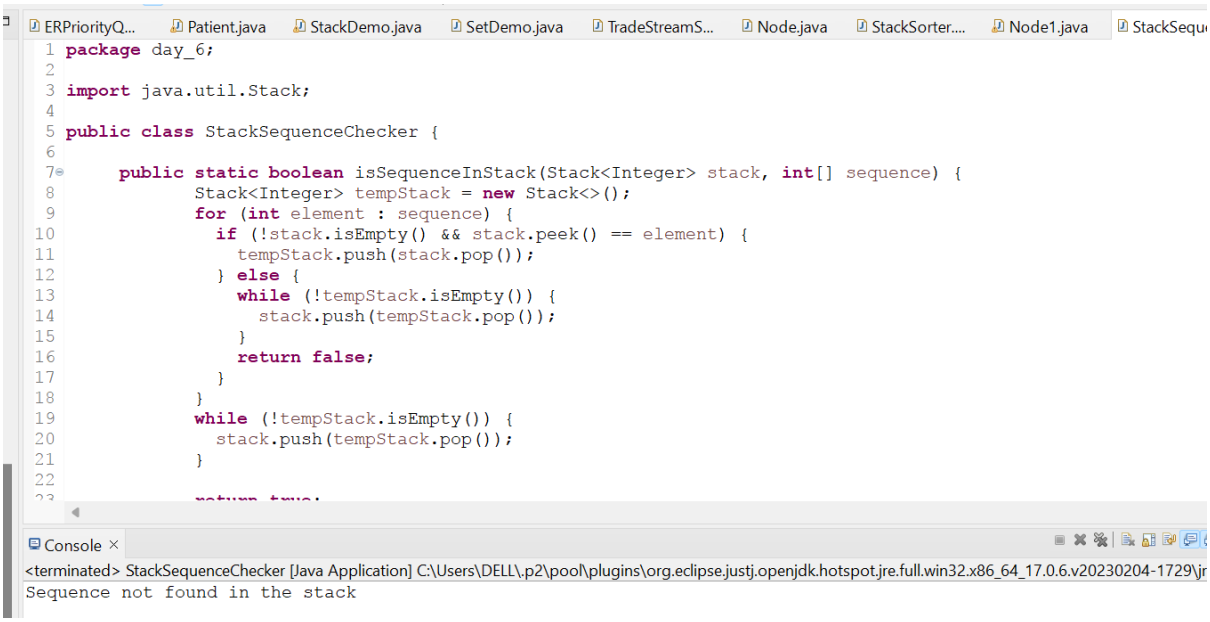
```
System.out.println("null");

}

public static void main(String[] args) {

Node head = createLinkedList();

if (head != null) {

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

printList(head);

head = removeDuplicates(head);

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

printList(head);

} else {

System.out.println("Linked list creation cancelled.");

}

}

}
```

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 day_6;

import java.util.Stack;

public class StackSequenceChecker {

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

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

for (int element : sequence) {

if (!stack.isEmpty() && stack.peek() == element) {

tempStack.push(stack.pop());

} else {

while (!tempStack.isEmpty()) {

stack.push(tempStack.pop());

}

return false;

}

}

while (!tempStack.isEmpty()) {

stack.push(tempStack.pop());

}

return true;

}

public static void main(String[] args) {

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

stack.push(4);
```

```
stack.push(3);

stack.push(1);

stack.push(2);

int[] sequence = {1, 2, 3};

if (isSequenceInStack(stack, sequence)) {

System.out.println("Sequence found in the stack");

} else {

System.out.println("Sequence not found in the stack");

}

}

}
```



Task 7: Merging Two Sorted Linked Lists

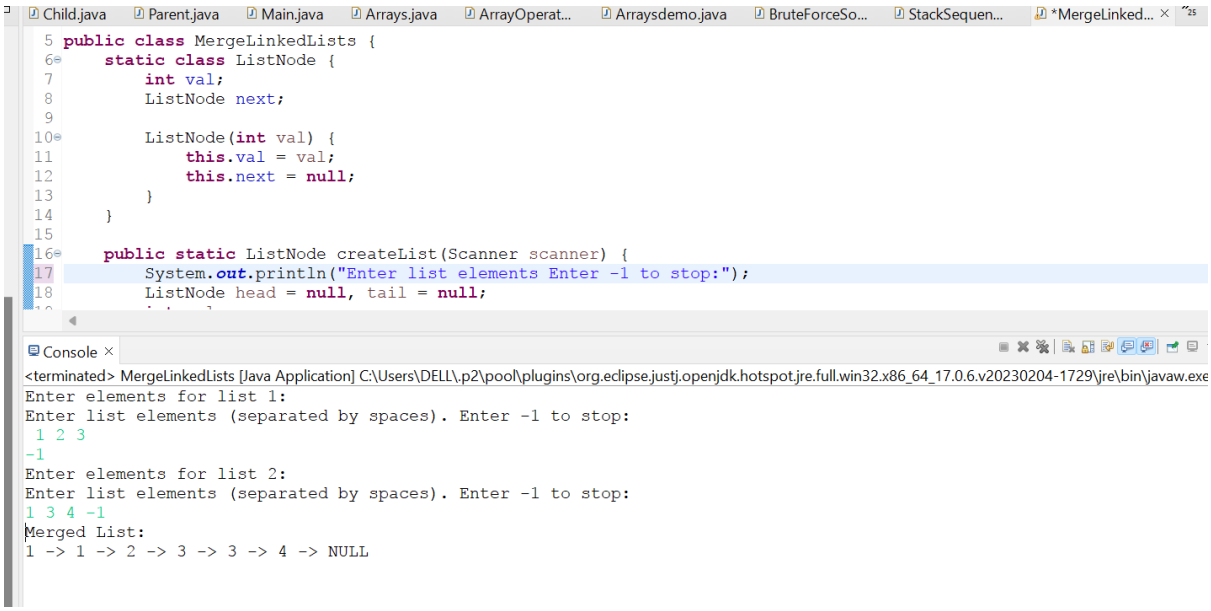
You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).

```
package day_6;

import java.util.Scanner;
```

```
public class MergeLinkedLists {

static class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

public static ListNode createList(Scanner scanner) {

System.out.println("Enter list elements Enter -1 to stop:");

ListNode head = null, tail = null;

int val;

while ((val = scanner.nextInt()) != -1) {

ListNode newNode = new ListNode(val);

if (head == null) {

head = tail = newNode;

} else {

tail.next = newNode;

tail = newNode;

}

}

return head;

}

public static ListNode mergeLists(ListNode list1, ListNode list2) {

if (list1 == null) {

return list2;

}
```

```
if (list2 == null) {
return list1;
}
ListNode dummy = new ListNode(0);
ListNode tail = dummy;
ListNode curr = dummy;
while (list1 != null && list2 != null) {
if (list1.val < list2.val) {
curr.next = list1;
list1 = list1.next;
} else {
curr.next = list2;
list2 = list2.next;
}
curr = curr.next;
}
curr.next = (list1 != null) ? list1 : list2;
return dummy.next;
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.println("Enter elements for list 1:");
ListNode list1 = createList(scanner);
System.out.println("Enter elements for list 2:");
ListNode list2 = createList(scanner);
ListNode mergedList = mergeLists(list1, list2);
System.out.println("Merged List:");
while (mergedList != null) {
```

```
System.out.print(mergedList.val + " -> ");

mergedList = mergedList.next;

}

System.out.println("NULL");

}

}
```



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 day_6;

public class CircularQueueBinarySearch {

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

int left = 0;

int right = arr.length - 1;

while (left <= right) {

int mid = (left + right) / 2;
```

```
if (arr[mid] == target) {
return mid;
}
if (arr[left] <= arr[mid]) {
if (target >= arr[left] && target <= arr[mid]) {
right = mid - 1;
} else {
left = mid + 1;
}
} else {
if (target >= arr[mid] && target <= arr[right]) {
left = mid + 1;
} else {
right = mid - 1;
}
}
}
return -1;
}
public static void main(String[] args) {
int[] arr = {4, 5, 6, 1, 2, 3};
int target = 5;
int index = search(arr, target);
if (index != -1) {
System.out.println("Element found at index: " + index);
} else {
System.out.println("Element not found in the queue");
}
```

```
}

}
```

Child.java | Parent.java | Main.java | Arrays.java | ArrayOperat... | Arraysdemo.java | StackSequen... | MergeLinked... | *CircularQ

```java
package day_6;

public class CircularQueueBinarySearch {

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

            if (arr[mid] == target) {
                return mid;
            }
            if (arr[left] <= arr[mid]) {
                if (target >= arr[left] && target <= arr[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else {
                if (target >= arr[mid] && target <= arr[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
```

Console ×

```
<terminated> CircularQueueBinarySearch [Java Application] C:\Users\DELL\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-
Element found at index: 1
```

Binary search directly on a circular queue isn't straightforward because the starting and ending positions aren't inherently defined. However, we can leverage the sorted and rotated nature of the queue to find the target element. Here's an approach to perform a binary search on a rotated and sorted circular queue:

1. **Find the Pivot:**
    - Initialize two pointers - `left` (starting at index 0) and `right` (pointing to the last element).
    - While `left` is less than or equal to `right`:
        - Calculate the mid index `mid = (left + right) // 2`.
        - If the element at `mid` is equal to the target element, return the `mid` index (element found).
        - Check if the left half (`left` to `mid`) is sorted:
            - If the element at `left` is less than or equal to the element at `mid`:
                - This indicates the pivot (rotation point) lies in the right half. Update `left` to `mid + 1`.
            - Otherwise (element at `left` is greater than element at `mid`), the pivot lies in the left half. Update `right` to `mid - 1`.
2. **Binary Search on the Identified Half:**

Now that we know the pivot is either in the left or right half, we can perform a regular binary search on the sorted half.

If the target element is greater than the element at `left` (identified sorted half), update `left` and `right` as in a normal binary search for the target in the sorted sub-array.

If the target element is less than the element at `left`, the target is not present in the queue (since it's sorted and rotated).

Time Complexity: Finding the pivot takes O(log n) time, and the subsequent binary search takes another O(log n) time. So, the total time complexity for searching the element is still O(log n).