Day 17 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@gmail.com

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 real_time_data_stream_sorting;

import java.util.PriorityQueue;

public class TradeTransaction  implements
Comparable<TradeTransaction> {
		double price;
		public TradeTransaction(double price) {
			this.price = price;
		}

		@Override
		public int compareTo(TradeTransaction other) {
			return Double.compare(this.price, other.price);
		}
	}

public class RealTimeSorting {
	private PriorityQueue<TradeTransaction> minHeap;

	public RealTimeSorting() {
		minHeap = new PriorityQueue<>();
	}

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

	public TradeTransaction getNextTrade() {
		return minHeap.poll();
	}

	public static void main(String[] args) {
		RealTimeSorting sorting = new RealTimeSorting();

		sorting.addTrade(new TradeTransaction(100.5));
		sorting.addTrade(new TradeTransaction(50.2));
		sorting.addTrade(new TradeTransaction(75.8));
```

```
            while (true) {
                  TradeTransaction trade = sorting.getNextTrade();
                  if (trade == null)
                        break;
                  System.out.println("Next trade price: " +
trade.price);
            }
      }
}
```

Output:

```
Next trade price: 50.2
Next trade price: 75.8
Next trade price: 100.5
```

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 linkedlist_mid_element_search;

public class ListNode {

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


public class SinglyLinkedList {

      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 printMiddle(ListNode head) {
                        ListNode mid = findMiddle(head);
                        if (mid != null)
```

```
                        System.out.println("Middle Element is :"
+ mid.val);
                else
                        System.out.println("The List is Empty!");
        }
        public static void main(String[] args) {
                ListNode head = new ListNode(55);
                head.next = new ListNode(45);
                head.next.next = new ListNode(22);
                head.next.next.next = new ListNode(35);
                head.next.next.next.next = new ListNode(14);
                printMiddle(head);
        }
}
```

Output:

```
Middle Element is :22
```

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.

```
package queuesorting;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;


public class QueueSort {


        public static void sortQueue(Queue<Integer> queue) {
                Stack<Integer> stack = new Stack<>();


                while (!queue.isEmpty()) {
                        int temp = queue.poll();


                        while (!stack.isEmpty() && stack.peek() <
temp) {
```

```
                        queue.add(stack.pop());
                    }
                    stack.push(temp);
                }

                while (!stack.isEmpty()) {
                    queue.add(stack.pop());
                }
            }
                public static void printQueue(Queue<Integer> queue)
{
                    while (!queue.isEmpty()) {
                        System.out.print(queue.poll() + " ");
                    }
                    System.out.println();
                }
                public static void main(String[] args) {
                    Queue<Integer> queue = new
LinkedList<Integer>();

                        queue.add(48);
                        queue.add(63);
                        queue.add(41);
                        queue.add(32);
                        queue.add(19);
                        queue.add(73);
                        queue.add(56);
                        queue.add(39);
                        queue.add(17);
                        queue.add(95);
                        queue.add(8);
```

```
                System.out.println("Original Queue: ");
                printQueue(new LinkedList<Integer>(queue));

                sortQueue(queue);

                System.out.println("Sorted Queue: ");
                printQueue(queue);
            }
}
```

Output:

```
Original Queue:
48 63 41 32 19 73 56 39 17 95 8
Sorted Queue:
8 17 19 32 39 41 48 56 63 73 95
```

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 stacksorting;

import java.util.Stack;

public class StackSort {

    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 printStack(Stack<Integer> stack) {
            for (Integer item : stack) {
                System.out.print(item + " ");
            }
            System.out.println();
        }

        public static void main(String[] args) {
            Stack<Integer> stack = new Stack<>();

            stack.push(32);
            stack.push(15);
            stack.push(41);
            stack.push(21);
            stack.push(56);
            stack.push(90);
            stack.push(26);
            stack.push(67);
            stack.push(85);

            System.out.println("Original stack:");
            printStack(stack);

            sortStack(stack);

            System.out.println("Sorted stack:");
            printStack(stack);
        }
}
```

Output:

```
Original stack:
32 15 41 21 56 90 26 67 85
Sorted stack:
15 21 26 32 41 56 67 85 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.

```java
package removing_duplicates;

import java.util.LinkedList;

public class ListNode {

	int val;
	ListNode next;

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

public class RemoveDuplicates {

	public static void removeDuplicates(ListNode head) {
		if (head == null) return;
		ListNode current = head;

		while (current != null && current.next != null) {
			if (current.val == current.next.val) {
				current.next = current.next.next;
			}
			else {
					current = current.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 head = new ListNode(10);

		head.next = new ListNode(10);
		head.next.next = new ListNode(21);
		head.next.next.next = new ListNode(13);
		head.next.next.next.next = new ListNode(13);
```

```
            System.out.println("Original list:");
            printList(head);

            removeDuplicates(head);

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

}
```

Output:

```
Original list:
10 10 21 13 13
List after removing duplicates:
10 21 13
```

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 com.stacksequence;

import java.util.Arrays;

import java.util.Stack;


public class StackInSequence {


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

sequence) {

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

                  int index = sequence.length - 1;


                  while (!stack.isEmpty()) {

                        int element = stack.pop();

                        if (element == sequence[index]) {
```

```
                        index--;
                }
                tempStack.push(element);

                if (index < 0)
                        break;
                }

            while (!tempStack.isEmpty()) {
                stack.push(tempStack.pop());
            }

            return index < 0;
        }

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

            int[] sequence1 = { 3, 4, 5 };
            System.out.println("Sequence: " +
Arrays.toString(sequence1));
            System.out.println("Is the sequence present in the
stack? " +
```

```
      isSequencePresent(stack, sequence1));


                  int[] sequence2 = { 5, 4, 1 };

                  System.out.println("Sequence: " +
Arrays.toString(sequence2));

                  System.out.println("Is the sequence present in the
stack? " +

      isSequencePresent(stack, sequence2));

                  }


}
```

Output:

```
Original Stack: [1, 2, 3, 4, 5, 6]
Sequence: [3, 4, 5]
Is the sequence present in the stack? true
Sequence: [5, 4, 1]
Is the sequence present in the 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 merging_linkedlist;

public class ListNode {

       int val;
       ListNode next;

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

public class MergeSortedLinkedList {

      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(1);
        l1.next = new ListNode(2);
        l1.next.next = new ListNode(4);

        ListNode l2 = new ListNode(1);
        l2.next = new ListNode(3);
        l2.next.next = new ListNode(4);

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

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

        ListNode mergedHead = mergeTwoLists(l1, l2);
```

```
            System.out.println("Merged Sorted list:");
            printList(mergedHead);
            }
}
```

Output:

```
List 1:
1 2 4
List 2:
1 3 4
Merged Sorted list:
1 1 2 3 4 4
```

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 circular_queue;

import java.util.Arrays;

import java.util.Scanner;


public class BinarySearchInCricularQueue {


      public static int searchInCircularQueue(int[] arr, int target)
{
                  int low = 0, 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 (arr[low] <= target && target <
arr[mid]) {
                                          high = mid - 1;
                                    } else {
                                                low = mid + 1;
                                          }
                              } else {
                                          if (arr[mid] < target &&
target <= arr[high]) {
                                                low = mid + 1;
                                          } else {
                                                high = mid - 1;
                                          }
                              }
                    }
                    return -1;
              }

      public static void main(String[] args) {
              Scanner sc = new Scanner(System.in);

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

              System.out.println("Circular Queue: " +
      Arrays.toString(circularQueue));

              System.out.print("Enter the Target Value: ");
              int target = sc.nextInt();

              int result = searchInCircularQueue(circularQueue,
target);
```

```
            if (result != -1) {
            System.out.println("Element " + target + " found at
index: " +
      result);
            } else {
            System.out.println("Element " + target + " not found in
the "
+ "circular queue.");
                  }
                  sc.close();
            }
}
```

Output:

```
Circular Queue: [4, 5, 6, 7, 0, 1, 2]
Enter the Target Value: 7
Element 7 found at index: 3
```