Assignment – 11

Prakash Manikanta Irrinki
Prakashnaidu9494@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.

Program:

```
import java.util.PriorityQueue;
public class RealTimeTradeSorting {
    public static void main(String[] args) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b)
-> b - a);
        maxHeap.add(100);
        maxHeap.add(150);
        maxHeap.add(120);
        while (!maxHeap.isEmpty()) {
            System.out.println(maxHeap.poll());
        }
    }
}
```

Output :

```
150
120
100
```

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.

Program:

```
class Node {
   int data;
   Node next;

   Node(int data) {
      this.data = data;
      this.next = null;
   }
}
class LinkedList {
   Node head;
   void findMiddleElement() {
      if (head == null) {
         System.out.println("The list is empty.");
         return;
      }
      Node slow = head;
      Node fast = head;

      while (fast != null && fast.next != null) {
         slow = slow.next;
         fast = fast.next.next;
      }

      System.out.println("The middle element is: " + slow.data);
   }
}
public class Main {
   public static void main(String[] args) {
      LinkedList list = new LinkedList();
      list.head = new Node(1);
      list.head.next = new Node(2);
      list.head.next.next = new Node(3);
      list.head.next.next.next = new Node(4);
      list.head.next.next.next.next = new Node(5);
      list.findMiddleElement();
   }
}
```

Output:

The middle element is: 3

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.

Program:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;
public class QueueSorting {
    public static void sortQueue(Queue<Integer> queue) {
        Stack<Integer> stack = new Stack<>();
        while (!queue.isEmpty()) {
            stack.push(queue.poll());
        }
        while (!stack.isEmpty()) {
            int temp = stack.pop();

            while (!queue.isEmpty() && queue.peek() < temp) {
                stack.push(queue.poll());
            }
      queue.add(temp);
        }
    }
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(5);
        queue.add(3);
        queue.add(8);
        queue.add(1);
       sortQueue(queue);
       System.out.println("Sorted Queue:");
        while (!queue.isEmpty()) {
            System.out.print(queue.poll() + " ");
        }
    }
}
```

Output :

```
Sorted Queue:
8 1 3 5
```

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.

Program:

```
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(5);
        stack.push(1);
        stack.push(3);
        stack.push(2);
        stack.push(4);
        System.out.println("Original Stack:");
        System.out.println(stack);
        sortStack(stack);
        System.out.println("Sorted Stack:");
        System.out.println(stack);
    }
}
```

Output:

```
Original Stack:
[5, 1, 3, 2, 4]
Sorted Stack:
[5, 4, 3, 2, 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.

Program:

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

    ListNode(int val) {
        this.val = val;
    }
}
class Solution {
    public ListNode removeDuplicates(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode current = head;
        while (current.next != null) {
            if (current.val == current.next.val) {
                current.next = current.next.next;
            } else {
                current = current.next;
            }
        }
        return head;
    }
}
public class Main {
    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);
        Solution solution = new Solution();
        ListNode newHead = solution.removeDuplicates(head);
        while (newHead != null) {
            System.out.print(newHead.val + " ");
            newHead = newHead.next;
        }
    }
}
```

Output:

```
1 2 3
```

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.

Program:

```
import java.util.Stack;
public class SequenceChecker {
    public static boolean isSequenceInStack(Stack<Integer> stack,
int[] sequence) {
        Stack<Integer> tempStack = new Stack<>();
        for (int num : stack) {
            tempStack.push(num);
        }
        for (int i = 0; i < sequence.length; i++) {
            while (!tempStack.isEmpty() && tempStack.peek() !=
sequence[i]) {
                tempStack.pop(); // Remove elements from the top of
the temporary stack
            }
            if (tempStack.isEmpty()) {
                return false;
            }
            tempStack.pop();
        }
        return true;
    }
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(5);
        stack.push(6);
        stack.push(7);
        stack.push(8);
        stack.push(9);
        stack.push(10);
        int[] sequence1 = {7, 8, 9};
        int[] sequence2 = {6, 7, 8};
        System.out.println("Is sequence [7, 8, 9] in the stack? " +
isSequenceInStack(stack, sequence1));
        System.out.println("Is sequence [6, 7, 8] in the stack? " +
isSequenceInStack(stack, sequence2));
    }
}
```

Output:

```
Is sequence [7, 8, 9] in the stack? false

Is sequence [6, 7, 8] 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).

Program:

```
public class ListNode {

      int element;
    ListNode next;

    ListNode(int val) {
        this.element = val;
    }
}
public class MergeTwoLinkedLists {
     public static ListNode merge(ListNode list1, ListNode list2) {
           if (list1 == null) {
            return list2;
        }
        if (list2 == null) {
            return list1;
        }
        ListNode mergedlist;
        if (list1.element < list2.element) {
            mergedlist = list1;
            list1 = list1.next;
        } else {
            mergedlist = list2;
            list2 = list2.next;
        }

        ListNode currentlist = mergedlist;

        while (list1 != null && list2 != null) {
            if (list1.element < list2.element) {
                currentlist.next = list1;
                list1 = list1.next;
            } else {
                currentlist.next = list2;
                list2 = list2.next;
            }
            currentlist = currentlist.next;
        }

        if (list1 != null) {
            currentlist.next = list1;
        }
        else if (list2 != null) {
            currentlist.next = list2;
        }
```

```
        return mergedlist;
      }
      public static void displayList(ListNode mergedlist) {
        ListNode currentlist = mergedlist;
        while (currentlist != null) {
            System.out.print(currentlist.element + " ");
            currentlist = currentlist.next;
        }
        System.out.println();
    }
      public static void main(String[] args) {
           ListNode list1 = new ListNode(1);
        list1.next = new ListNode(3);
        list1.next.next = new ListNode(5);

        ListNode list2 = new ListNode(2);
        list2.next = new ListNode(4);
        list2.next.next = new ListNode(6);

        displayList(list1);
        displayList(list2);

        ListNode mergedlist = merge(list1, list2);
        displayList(mergedlist);

      }
}
```

Output:

```
1 3 5
2 4 6
1 2 3 4 5 6
```

Explanation:

To merge two sorted linked lists without using extra space, you can manipulate the pointers of the existing nodes to create the merged list. Here's how you can do it:

1. Compare the heads of the two input linked lists.

2. Assign the smaller head as the head of the merged list.

3. Move to the next node of the smaller list and compare it with the current node of the other list.

4. Repeat this process until one of the lists becomes empty.

5. Append the remaining nodes of the non-empty 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.

Program:

```
public class CircularQueueBinarySearch {
	public static int search(int[] arr, int key) {
		int start=0;
		int end=arr.length-1;
		while(start <= end) {
			int mid=start + (end-start)/2;
			if(arr[mid] == key) {
				return mid;
			}
			if(arr[start] <= arr[mid]) {
				if(arr[start] <= key && key < arr[mid]) {
					end=mid-1;
				}
				else {
					start=mid+1;
				}
			}
			else {
				if(arr[mid] < key && key <= arr[end]) {
					start=mid+1;
				}
				else {
					end=mid-1;
				}
			}
		}
		return -1;
	}
	public static void main(String[] args) {

		int[] arr= {2,5,8,4,1,0,3};
		int key=0;
		int index=search(arr,key);

		if(index!=-1) {
			System.out.println("Element found at index:"+index);
		}
		else {
			System.out.println("Element not found.");
		}
	}
}
```

Output:

```
Element found at index:5
```

Explanation:

Performing a binary search on a rotated sorted array, like the one represented by a circular queue, requires modifying the traditional binary search algorithm slightly to accommodate the rotation. Here's an approach to perform a binary search for a given element within a circular queue:

Find the Pivot Index:

First, you need to find the pivot index where the rotation occurred. You can achieve this by performing a modified binary search to find the smallest element in the array, which indicates the rotation point.

Determine Search Range:

Once you have the pivot index, you can determine which half of the array to search based on comparing the target element with the first and last elements of the array.

Perform Binary Search:

Perform a standard binary search within the determined search range.

Handle Wraparound:

Since the array is circular, you need to handle the wraparound condition when adjusting the mid index during the binary search.

step-by-step algorithm:

- Initialize variables for the start and end indices of the array.

- Find the pivot index using a modified binary search to locate the smallest element.

- Determine which half of the array to search based on the target value and the values at the start and end indices.

- Perform a standard binary search within the determined search range, adjusting for wraparound.

- If the target element is found, return its index. Otherwise, return -1 to indicate that the element is not present in the circular queue.