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.

```
import java.util.Arrays;


public class RealTimeTradingApp {
    private static int[] trades = new int[100];
    private static int numTrades = 0;


    public static void addTrade(int trade) {
        if (numTrades == trades.length) {
            trades = Arrays.copyOf(trades, trades.length * 2);
        }
        trades[numTrades++] = trade;
        heapSort(trades, numTrades);
    }


    public static void heapSort(int[] arr, int size) {
        for (int i = size / 2 - 1; i >= 0; i--)
            heapify(arr, size, i);
                for (int i = size - 1; i > 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
```

```
        heapify(arr, i, 0);
    }
}

public static void heapify(int[] arr, int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left = 2*i + 1
    int right = 2 * i + 2; // right = 2*i + 2

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;



    if (largest != i) {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        heapify(arr, n, largest);
    }
}

public static void main(String[] args) {
    // Example of adding trades
    addTrade(12);
    addTrade(11);
```

```
        addTrade(13);

        System.out.println("Sorted trades after initial additions: " +
Arrays.toString(Arrays.copyOf(trades, numTrades)));

        addTrade(5);
        addTrade(8);
        addTrade(6);

        System.out.println("Sorted trades after continuous additions: " +
Arrays.toString(Arrays.copyOf(trades, numTrades)));
    }
}
```

Explanation:

- RealTimeTradingApp class: This class manages the continuous addition of trades and sorting them using heap sort.
- addTrade method: Adds a new trade to the trades array. If the array is full, it doubles its size. After adding a new trade, it calls heapSort to sort the array of trades.
- heapSort method: This method sorts the array using the heap sort algorithm. It first builds a max heap from the array and then repeatedly extracts the maximum element to sort the array in ascending order.
- heapify method: This method is used to maintain the heap property while building the heap or performing heap sort.
- main method: This method demonstrates how to add trades and print the sorted trades array.

Output:

- **Sorted trades after initial additions: [11, 12, 13]**
- **Sorted trades after continuous additions: [5, 6, 8, 11, 12, 13]**

Explanation:

- Initially, we add trades with prices 12, 11, and 13. The heapSort method sorts them, and we print the sorted array.
- Next, we add more trades with prices 5, 8, and 6. Again, after sorting using heapSort, we print the sorted array.

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.

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

    ListNode(int x) {
        val = x;
        next = null;
    }
}

public class FindMiddleElement {

    public ListNode findMiddle(ListNode head) {
        if (head == null) {
            return null;
        }

        ListNode slow = head;
        ListNode fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;        // Move slow pointer by one step
            fast = fast.next.next;   // Move fast pointer by two steps
        }

        return slow;
```

```
    }

    public static void main(String[] args) {
        // Example usage:
        FindMiddleElement finder = new FindMiddleElement();

        // Create a linked list: 1 -> 2 -> 3 -> 4 -> 5
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);

        ListNode middle = finder.findMiddle(head);

        if (middle != null) {
            System.out.println("Middle element: " + middle.val);  // Output: Middle element: 3
        } else {
            System.out.println("List is empty.");
        }
    }
}
```

1 **Single Pass with Two Pointers**: Use two pointers, slow and fast, initialized to the head of the linked list.

- slow pointer moves one step at a time.
- fast pointer moves two steps at a time.

2 Finding the Middle Element:

- Traverse the linked list with fast pointer moving two steps ahead and slow pointer moving one step ahead in each iteration.
- When fast pointer reaches the end of the list (fast.next is null), slow pointer will be at the middle element of the linked list.

3 Edge Cases:

- If the linked list is empty, return null.
- If there is only one element in the list, that element is the middle.

ListNode class: Represents each node of the singly linked list.

FindMiddleElement class:

findMiddle method: This method takes the head of the linked list as input and returns the middle node. It uses the two-pointer technique to find the middle node with one pass through the list.

main method: Demonstrates the usage by creating a sample linked list and printing the middle element.

Complexity:

Time Complexity:O(n) where n is the number of nodes in the linked list. This is because we traverse the list once.

Space Complexity: O(1) as we use only two pointers (slow and fast) and no extra space proportional to the input size.

Output:

Middle element: 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.

- **Steps to Sort a Queue using One Stack:**
    1. Initialize the queue and stack**:**
- Create a Queue<Integer> to hold the elements.
- Create a Stack<Integer> to assist with sorting.

```
Queue<Integer> queue = new LinkedList<>();
Stack<Integer> stack = new Stack<>();
```

2. **Sort the elements using the stack:**

- While the queue is not empty:
- Dequeue an element from the queue.
- While the stack is not empty and the top of the stack is greater than the dequeued element, dequeue elements from the stack and enqueue them into the queue.
- Enqueue the dequeued element into the stack.

```
while (!queue.isEmpty()) {
   int temp = queue.poll();
   while (!stack.isEmpty() && stack.peek() > temp) {
      queue.offer(stack.pop());
   }
   stack.push(temp);
}
```

3. **Rebuild the queue from the stack:**

- While the stack is not empty, dequeue elements from the stack and enqueue them back into the queue.

```
while (!stack.isEmpty()) {
   queue.offer(stack.pop());
}
```

4. **Final sorted queue:**

At this point, the queue should contain the sorted elements.

// Now 'queue' contains the sorted elements

```
import java.util.*;
public class QueueSortUsingStack {
   public static void sortQueue(Queue<Integer> queue) {
```

```
        if (queue == null || queue.isEmpty()) return;
        Stack<Integer> stack = new Stack<>();
        while (!queue.isEmpty()) {
            int temp = queue.poll();

            while (!stack.isEmpty() && stack.peek() > temp) {
                queue.offer(stack.pop());
            }
            stack.push(temp);
        }

        while (!stack.isEmpty()) {
            queue.offer(stack.pop());
        }
    }

    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(5);
        queue.add(3);
        queue.add(1);
        queue.add(7);
        queue.add(2);

        System.out.println("Original queue: " + queue);
        sortQueue(queue);
        System.out.println("Sorted queue: " + queue);
    }
}
```

Explanation:

- **sortQueue(Queue<Integer> queue):** This method takes a queue of integers as input and sorts it using one additional stack.
- **main(String[] args):** In the main method, we create a queue with some integers, sort it using sortQueue, and then print the sorted queue**.**

Output:

Original queue: [5, 3, 1, 7, 2]

Sorted queue: [1, 2, 3, 5, 7]

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.

```
import java.util.Stack;

public class SortStack {

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

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

      while (!stack.isEmpty()) {

         // Pop the top element from the original stack

         int temp = stack.pop();


         // While temporary stack is not empty and top of stack is greater than temp,

         // pop from temporary stack and push it to the original stack

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

            stack.push(tempStack.pop());

         }

         // Push temp in tempStack

         tempStack.push(temp);

      }
```

```
        // Copy the elements from tempStack to stack
        while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
        }
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(5);
        stack.push(2);
        stack.push(8);
        stack.push(1);
        stack.push(3);
        System.out.println("Stack before sorting:");
        System.out.println(stack);
        sortStack(stack);
        System.out.println("Stack after sorting:");
        System.out.println(stack);
    }
}
```

Explanation:

1 sortStack function: This function sorts the input stack in ascending order.

- We use an additional temporary stack (tempStack) to assist with sorting.
- We iterate through each element in the original stack:
- Pop the top element from the original stack and store it in temp.
- While tempStack is not empty and the top of tempStack is greater than temp, pop from tempStack and push it back onto the original stack.
- Push temp onto tempStack.
- After the loop, all elements are in tempStack in ascending order.

- Finally, we move elements from tempStack back to the original stack**.**

Main function: Demonstrates the usage of **sortStack.**

- We create a sample stack with unsorted elements.
- Print the stack before sorting.
- Sort the stack using sortStack.
- Print the stack after sorting.

Output:

Stack before sorting:

[5, 2, 8, 1, 3]

Stack after sorting:

[8, 5, 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.

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

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

public class RemoveDuplicates {

   public ListNode deleteDuplicates(ListNode head) {
      if (head == null || head.next == null) {
         return 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) {
    // Sorted linked list: 1 -> 2 -> 2 -> 3 -> 3 -> 3 -> 4 -> 5 -> 5
    ListNode head = new ListNode(1);
    head.next = new ListNode(2);
    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(3);
        head.next.next.next.next.next.next = new ListNode(4);
        head.next.next.next.next.next.next.next = new ListNode(5);
        head.next.next.next.next.next.next.next.next = new ListNode(5);

        System.out.println("Linked list before removing duplicates:");
        printList(head);

        RemoveDuplicates solution = new RemoveDuplicates();
        ListNode newHead = solution.deleteDuplicates(head);

        System.out.println("Linked list after removing duplicates:");
        printList(newHead);
    }
}
```

Explanation:

- **ListNode Class**: Represents each node in the linked list.
- **deleteDuplicates** Method: This method takes the head of the sorted linked list as input and removes duplicates in-place.
- It initializes current to traverse the list.
- Inside the loop, it checks if current and **current.next** have the same value. If they do, it skips **current.next** to remove the duplicate node.
- If they do not have the same value, it moves current to **current.next** to process the next node.
- **printLis**t Method: Utility method to print the linked list for verification.
- Main Method: Sets up an example sorted linked list, calls **deleteDuplicates**, and prints the modified list.

Output:

Linked list before removing duplicates:

1 2 2 3 3 3 4 5 5

Linked list after removing duplicates:

1 2 3 4 5

Time Complexity:

- The time complexity of this algorithm is O(n), where n is the number of nodes in the linked list. This is because each node is processed exactly once.

Space Complexity:

- The space complexity is O(1), as we are modifying the linked list in-place and using only a constant amount of extra space for pointers.

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.

```
import java.util.Stack;


public class SequenceInStack {
   public static boolean sequenceInStack(Stack<Integer> stack, int[] sequence) {
      Stack<Integer> tempStack = new Stack<>();
      for (int i = 0; i < sequence.length; i++) {
         int current = sequence[i];
         while (!stack.isEmpty() && stack.peek() != current) {
            tempStack.push(stack.pop());
         }
         if (stack.isEmpty()) {
            return false;
         }
         stack.pop();


         // Push back elements from tempStack to the original stack
         while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
         }
```

```
        }

        // If all elements are found and removed, return true
        return true;
    }

    // Example usage
    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 = {1, 2, 3};
        int[] sequence2 = {4, 5, 3};

        System.out.println(sequenceInStack(stack, sequence1));
        System.out.println(sequenceInStack(stack,  sequence2));
    }
}
```

Explanation:

1 Function sequenceInStack:

- **stack**: The original stack where we want to check the sequence.
- **sequence:** The sequence we want to check for in the stack.
- **tempStack:** A temporary stack used to hold elements temporarily while searching for the sequence.

2 Algorithm:

- Iterate through each element of the sequence array.
- For each element in the sequence:
- Pop elements from stack to tempStack until the top element of stack matches the current element in sequence.
- If stack becomes empty before finding the current element in sequence, return false.
- Pop the element from stack.
- Push elements back from tempStack to stack.
- If all elements of sequence are found and removed from stack, return true.

Main Method:

Example usage demonstrates how to use sequenceInStack function with different sequences and outputs the result (true or false).

Output:

true

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).

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // Handle cases where one of the lists is null
        if (l1 == null) return l2;
        if (l2 == null) return l1;
```

```
        // Ensure l1 is the smaller list
        if (l1.val > l2.val) {
            ListNode temp = l1;
            l1 = l2;
            l2 = temp;
        }
        ListNode head = l1;
        while (l1 != null && l2 != null) {
            ListNode next1 = l1.next;
            ListNode next2 = l2.next;

            if (next1 != null && next1.val <= l2.val) {
                l1 = next1;
            } else {
                l1.next = l2;
                l1 = l2;
                l2 = next1;
            }
        }

        if (l1 == null) {
            l1 = l2;
        }

        return head;
    }
}
```

Explanation:

1 ListNode Definition: Defines the structure of a node in the linked list.

```
class ListNode {
   int val;
   ListNode next;
   ListNode(int x) { val = x; }
}
```

2 mergeTwoLists Function:

- This function takes two sorted linked lists l1 and l2 as input and returns a merged list.
- The function modifies the existing nodes of the input lists to create the merged list without using extra space.

3 Edge Cases:

If either l1 or l2 is null, simply return the other list because a merged list would be the same as the non-null list.

```
if (l1 == null) return l2;
if (l2 == null) return l1;
```

4 Initialization:

Ensure l1 is the smaller list by swapping if necessary.

```
if (l1.val > l2.val) {
   ListNode temp = l1;
   l1 = l2;
   l2 = temp;
}
```

5 Merging Process:

- Traverse through both lists (l1 and l2) simultaneously and rearrange their pointers.
- At each step, move the pointer from the smaller node to the next node.

```
ListNode head = l1;
while (l1 != null && l2 != null) {
   ListNode next1 = l1.next;
   ListNode next2 = l2.next;
```

```
    if (next1 != null && next1.val <= l2.val) {
        l1 = next1;
    } else {
        l1.next = l2;
        l1 = l2;
        l2 = next1;
    }
}
```

6 Final Adjustment:

If one of the lists (l1 or l2) is exhausted, append the rest of the other list to the end of the merged list.

```
if (l1 == null) {
    l1 = l2;
}
```

7 Return:

Return the head of the merged list.

```
return head;
```

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.

1 Identify the Rotation Point:

First, identify the index at which the array is rotated. This can be done using a modified binary search. Once you know the rotation point, you can treat the array as if it were sorted from the rotation point to the end and from the beginning up to the rotation point.

2 Perform Binary Search:

- Perform a binary search on the rotated array. You can determine which part of the array to search based on the target value and the values at the start, middle, and end of the array.

```
public class CircularQueueBinarySearch {
   public int search(int[] nums, int target) {
      int n = nums.length;
      if (n == 0) return -1;
      int pivot = findPivot(nums);
      if (pivot == -1) {
         return binarySearch(nums, 0, n - 1, target);
      }

      if (nums[pivot] == target) {
         return pivot;
      }
      if (target >= nums[0] && target <= nums[pivot - 1]) {
         return binarySearch(nums, 0, pivot - 1, target);
      } else {
         return binarySearch(nums, pivot + 1, n - 1, target);
      }
   }
   private int findPivot(int[] nums) {
      int low = 0;
      int high = nums.length - 1;

      while (low <= high) {
         int mid = (low + high) / 2;
```

```
        if (mid > 0 && nums[mid] < nums[mid - 1]) {
            return mid;
        }
        if (nums[mid] >= nums[low]) {
            // Left part is sorted, pivot must be in the right
            low = mid + 1;
        } else {
            // Right part is sorted, pivot must be in the left part
            high = mid - 1;
        }
    }
    // If no pivot found, the array is not rotated
    return -1;
}

// Function to perform binary search in a sorted array
private int binarySearch(int[] nums, int low, int high, int target) {
    while (low <= high) {
        int mid = (low + high) / 2;
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
```

```
    }

    public static void main(String[] args) {
        CircularQueueBinarySearch solution = new CircularQueueBinarySearch();

        int[] nums1 = {4, 5, 6, 7, 0, 1, 2};
        int target1 = 0;
        System.out.println("Index of " + target1 + " in nums1: " + solution.search(nums1, target1));

        int[] nums2 = {4, 5, 6, 7, 0, 1, 2};
        int target2 = 3;
        System.out.println("Index of " + target2 + " in nums2: " + solution.search(nums2, target2));
    }
}
```

Explanation:

search Method:

This method initiates the search process by finding the pivot (rotation point) of the rotated sorted array and then performing a binary search on the correct segment of the array based on the target value.

findPivot Method:

Uses binary search to find the pivot element (smallest element in the rotated array). If the array is not rotated, it returns -1.

binarySearch Method:

Performs a standard binary search on the sorted segment of the array identified by the low and high indices.

Main Method:

Demonstrates how to use the search method to find the indices of target values in two different example arrays.