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.

```
public class RealtimeTradeSorter {
    private Trade[] trades;
    private int capacity;
    private int size;

    public RealtimeTradeSorter(int capacity) {
        this.capacity = capacity;
        trades = new Trade[capacity];
    }

    public void addTrade(Trade trade) {
        if (size == capacity) {
            throw new IllegalStateException("Trade sorter capacity reached");
        }
        trades[size] = trade;
        size++;
        siftUp(size - 1);
    }

    private void swap(int index1, int index2) {
        Trade temp = trades[index1];
        trades[index1] = trades[index2];
        trades[index2] = temp;
    }
```

```
private void siftUp(int index) {
    while (index > 0 && trades[parent(index)].getPrice() < trades[index].getPrice()) {
        swap(index, parent(index));
        index = parent(index);
    }
}

private int parent(int index) {
    return (index - 1) / 2;
}

public Trade[] getTopKTrades(int k) {
    if (k > size) {
        throw new IllegalArgumentException("k cannot be greater than the number of trades");
    }
    Trade[] topK = new Trade[k];
    System.arraycopy(trades, 0, topK, 0, k);
    return topK;
}
public Trade[] getFullySortedTrades() {
    for (int i = size / 2 - 1; i >= 0; i--) {
        siftDown(i);
    }
    return trades;
}
private void siftDown(int index) {
    int leftChild = leftChild(index);
    int rightChild = rightChild(index);
    int largest = index;
    if (leftChild < size && trades[leftChild].getPrice() > trades[largest].getPrice()) {
```

```
            largest = leftChild;
        }
        if (rightChild < size && trades[rightChild].getPrice() > trades[largest].getPrice()) {
            largest = rightChild;
        }
        if (largest != index) {
            swap(index, largest);
            siftDown(largest);
        }
    }
    private int leftChild(int index) {
        return 2 * index + 1;
    }
    private int rightChild(int index) {
        return 2 * index + 2;
    }
}
public class Trade {
    String symbol;
    double price;
    int quantity;
}
```

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.

```
public static Node findMiddle(Node head) {
  if (head == null || head.next == null) {
    return head;
  }
  SlowFastPointers slow = new SlowFastPointers(head);
  while (fast != null && fast.next != null) {
    slow.slow = slow.slow.next;
    fast = fast.next.next;
  }
  return slow.slow;
}

private static class SlowFastPointers {
  public Node slow;
  public SlowFastPointers(Node head) {
    this.slow = head;
  }
}
```

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 push to stack:**
    - While the queue is not empty:
        - Dequeue an element from the queue and push it onto the stack.
2. **Sort the stack:**
    - **(Optional):** If you have a built-in sorting function that can handle stacks, you can use it here. However, let's explore a manual sorting approach using recursion.
    - Define a recursive function `sortStack(stack)`.
    - **Base case:** If the stack is empty or has only one element, it's already sorted (do nothing).
    - **Recursive call:** Pop the top element from the stack (`temp`).
    - Call `sortStack(stack)` to sort the remaining elements on the stack.
    - Now, we need to insert `temp` back into the sorted stack. This involves iteratively popping elements from the stack, comparing them with `temp`, and pushing the smaller elements back onto the stack. Once you encounter an element larger than `temp`, push `temp` onto the stack and then push the remaining popped elements back in the reverse order.
3. **Push back to queue:**
    - While the stack is not empty:
        - Pop an element from the stack and enqueue it back into 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.

```
public static void sortStack(Stack<Integer> dataStack, Stack<Integer> tempStack) {
  while (!dataStack.isEmpty()) {
    int temp = dataStack.pop();
      while (!tempStack.isEmpty() && temp < tempStack.peek()) {
      dataStack.push(tempStack.pop());
    }
    tempStack.push(temp);
  }
  while (!tempStack.isEmpty()) {
    dataStack.push(tempStack.pop());
  }
}
```

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.

Steps:

1. **Initialize pointers:**
    - Set two pointers: `current` to the head of the linked list and `prev` to `null`.
2. **Iterate through the list:**
    - While `current` is not null:
        - Check if the value of `current` is the same as the value of the node pointed to by `prev`.
            - If the values are the same (duplicate), skip the current node by setting `current.next` to `current.next.next`.
            - If the values are different (unique), update `prev` to point to the current node.
        - Move `current` to the next node (`current = current.next`).
3. **Return the head:**
    - After the loop, the `prev` pointer will be pointing to the last unique node in the list (or null if the list was empty or all elements were duplicates).
    - Return the head of the modified list, which can be either `null` (if the original list was empty) or the node pointed to by `prev`.

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.

```
public static boolean sequenceInStack(Stack<Integer> stack, int[] sequence) {
  Stack<Integer> tempStack = new Stack<>();
    for (int element : sequence) {
    found:
    while (!stack.isEmpty()) {
      int topElement = stack.pop();
      tempStack.push(topElement);
      if (topElement == element) {
        break found;
      }
    }

    while (!tempStack.isEmpty()) {
      stack.push(tempStack.pop());
    }
    return false;
  }
  while (!tempStack.isEmpty()) {
    stack.push(tempStack.pop());
  }
  return true;
}
```

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).

```
public static ListNode mergeTwoLists(ListNode list1, ListNode list2) {
  if (list1 == null) return list2;
  if (list2 == null) return list1;

  ListNode mergedHead = (list1.val <= list2.val) ? list1 : list2;
  ListNode current = mergedHead;
  ListNode list1Pointer = list1;
  ListNode list2Pointer = list2;

  while (list1Pointer != null && list2Pointer != null) {
    if (list1Pointer.val <= list2Pointer.val) {
      current.next = list1Pointer;
      list1Pointer = list1Pointer.next;
    } else {
      current.next = list2Pointer;
      list2Pointer = list2Pointer.next;
    }
    current = current.next;
  }
  current.next = (list1Pointer != null) ? list1Pointer : list2Pointer;

  return mergedHead;
}

class ListNode {
```

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

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.

Steps:

1. **Find the Pivot:**
    - Initialize two pointers: `low` pointing to the beginning of the array (`queue[0]`) and `high` pointing to the end of the array (`queue[queue.length - 1]`).
    - While `low` is less than or equal to `high`:
        - Calculate the mid index: `mid = (low + high) / 2`.
        - If the middle element (`queue[mid]`) is greater than both the low (`queue[low]`) and high (`queue[high]`) elements, it means the pivot (point of rotation) lies before the middle element. This is because in a sorted and rotated array, elements before the pivot are greater than elements after the pivot.
            - Update `high` to `mid - 1`.
        - If the middle element is less than or equal to the low element, it means the pivot lies after or at the middle element.
            - Update `low` to `mid + 1`.
    - After the loop terminates, `low` will point to the first element after the pivot (or the pivot itself if the array has only one element).

2. **Binary Search on the Correct Half:**
    - Check if the target element is equal to the low element (`queue[low]`). If yes, return the low index (found).
    - Depending on the position of the pivot identified in step 1:
        - If the target element is greater than or equal to the low element (`queue[low]`) and less than or equal to the high element (`queue[queue.length - 1]`), perform a normal binary search on the sub-array from `low` to `queue.length - 1` (including the high element).
        - If the target element is less than the low element (`queue[low]`), perform a binary search on the sub-array from `low + 1` to `high` (excluding the low element).