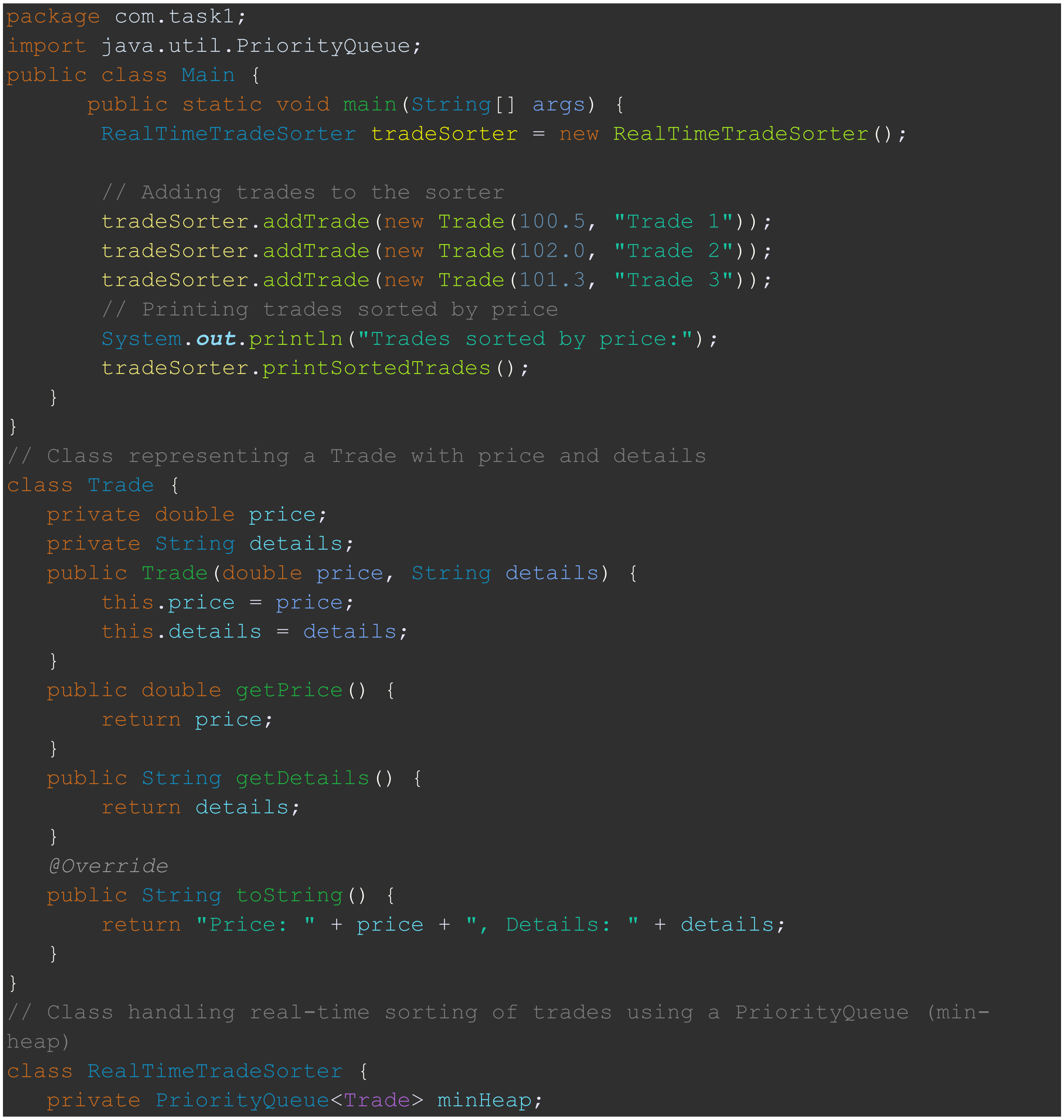
Core_Java_Day 6(digvijaythakare2017@gmail.com)

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.

Solution:



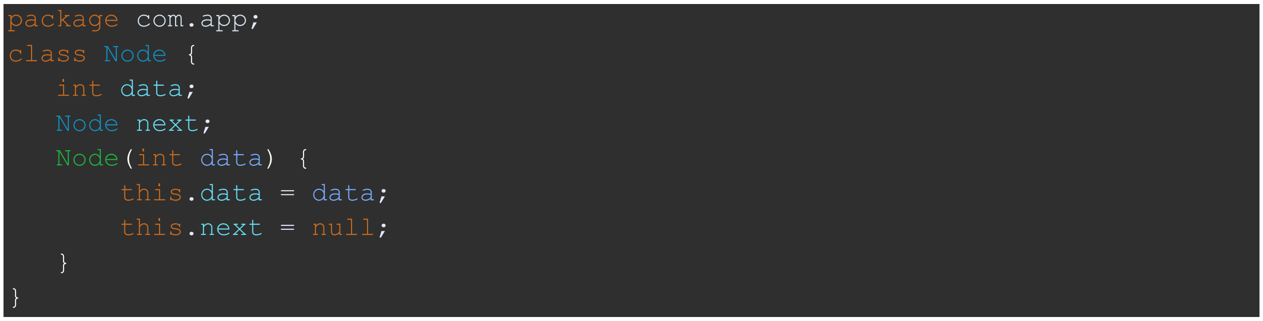
```
package com.task1;
import java.util.PriorityQueue;
public class Main {
      public static void main(String[] args) {
       RealTimeTradeSorter tradeSorter = new RealTimeTradeSorter();

       // Adding trades to the sorter
       tradeSorter.addTrade(new Trade(100.5, "Trade 1"));
       tradeSorter.addTrade(new Trade(102.0, "Trade 2"));
       tradeSorter.addTrade(new Trade(101.3, "Trade 3"));
       // Printing trades sorted by price
       System.out.println("Trades sorted by price:");
       tradeSorter.printSortedTrades();
   }
}
// Class representing a Trade with price and details
class Trade {
   private double price;
   private String details;
   public Trade(double price, String details) {
       this.price = price;
       this.details = details;
   }
   public double getPrice() {
       return price;
   }
   public String getDetails() {
       return details;
   }
   @Override
   public String toString() {
       return "Price: " + price + ", Details: " + details;
   }
}
// Class handling real-time sorting of trades using a PriorityQueue (min-heap)
class RealTimeTradeSorter {
   private PriorityQueue<Trade> minHeap;
   public RealTimeTradeSorter() {
       // Initializing the min-heap with a custom comparator for Trade
objects
       minHeap = new PriorityQueue<>((trade1, trade2) ->
Double.compare(trade1.getPrice(), trade2.getPrice()));
   }
   // Method to add a new trade to the heap
   public void addTrade(Trade trade) {
       minHeap.offer(trade);
   }
   // Method to get the trade with the minimum price without removing it
   public Trade getMinPriceTrade() {
       return minHeap.peek();
   }
   // Method to remove and return the trade with the minimum price
   public Trade removeMinPriceTrade() {
       return minHeap.poll();
   }
   // Method to print all trades sorted by price
   public void printSortedTrades() {
       while (!minHeap.isEmpty()) {
           System.out.println(minHeap.poll());
       }
   }
}
```

Output:

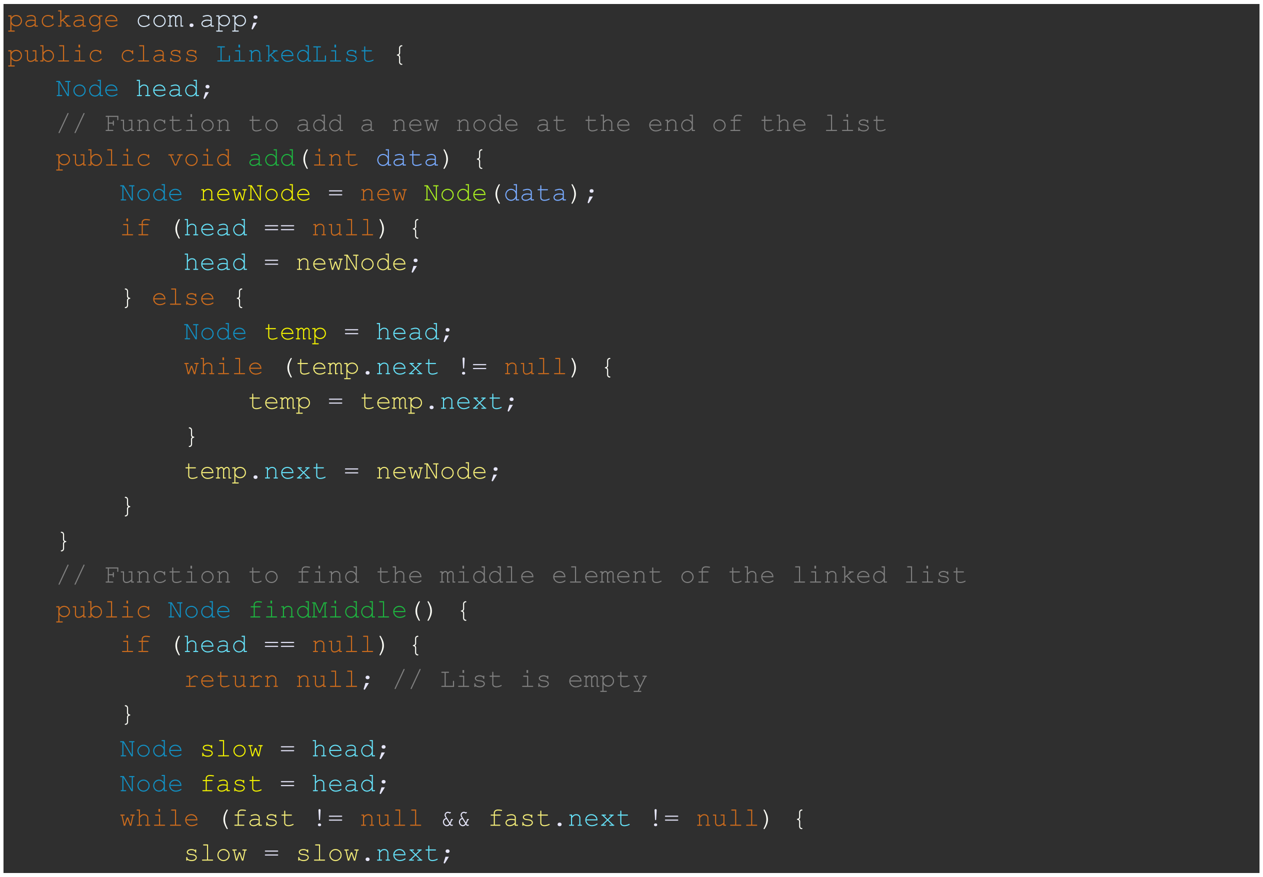
```
Trades sorted by price:
Price: 100.5, Details: Trade 1
Price: 101.3, Details: Trade 3
Price: 102.0, Details: Trade 2
```

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.

Solution:



```java
package com.app;
class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```



```java
package com.app;
public class LinkedList {
    Node head;
    // Function to add a new node at the end of the list
    public void add(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }
    // Function to find the middle element of the linked list
    public Node findMiddle() {
        if (head == null) {
            return null; // List is empty
        }
        Node slow = head;
        Node fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow; // slow is now at the middle node
    }
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        Node middle = list.findMiddle();
        if (middle != null) {
            System.out.println("The middle element is: " + middle.data);
        } else {
            System.out.println("The list is empty.");
        }
    }
}
```

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.

Solution:

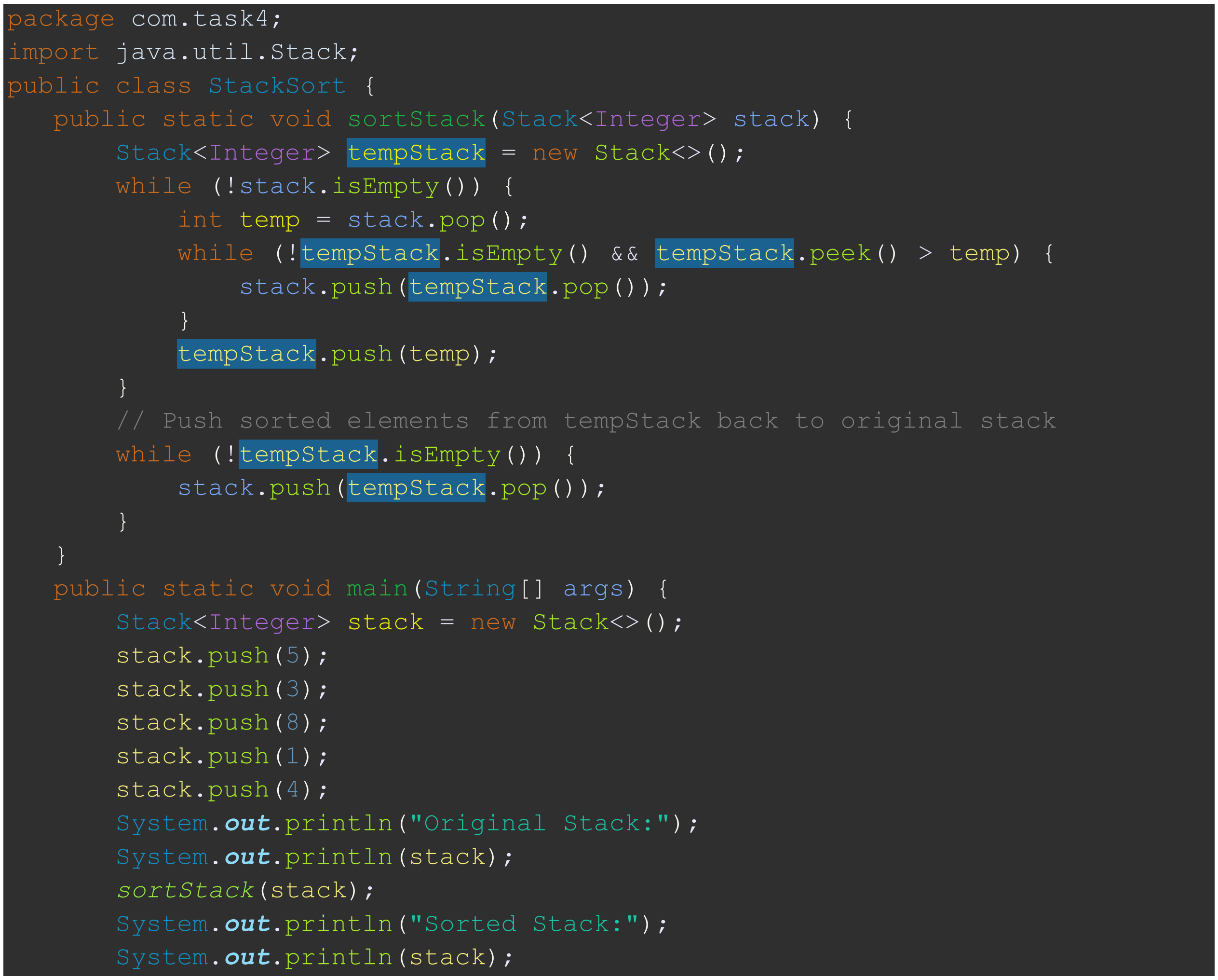
```java
package com.task3;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;
public class QueueSort {
    public static void sort(Queue<Integer> queue) {
        if (queue == null || queue.size() <= 1) {
            return; // Base case: already sorted or empty queue
        }
        Stack<Integer> stack = new Stack<>();
        // Divide the queue into two subqueues
        Queue<Integer> subqueue1 = new LinkedList<>();
        Queue<Integer> subqueue2 = new LinkedList<>();
        boolean toggle = true; // To alternate between subqueues
        while (!queue.isEmpty()) {
            if (toggle) {
                subqueue1.offer(queue.poll());
            } else {
                subqueue2.offer(queue.poll());
            }
            toggle = !toggle;
        }
        // Recursively sort the subqueues
        sort(subqueue1);
        sort(subqueue2);
        // Merge the sorted subqueues back into the original queue
        while (!subqueue1.isEmpty() && !subqueue2.isEmpty()) {
            if (subqueue1.peek() < subqueue2.peek()) {
                queue.offer(subqueue1.poll());
            } else {
                queue.offer(subqueue2.poll());
            }
        }
        // Enqueue any remaining elements from subqueue1
        while (!subqueue1.isEmpty()) {
            queue.offer(subqueue1.poll());
        }
        // Enqueue any remaining elements from subqueue2
        while (!subqueue2.isEmpty()) {
            queue.offer(subqueue2.poll());
        }
    }
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(5);
        queue.offer(3);
        queue.offer(8);
        queue.offer(1);
        queue.offer(4);
        System.out.println("Original Queue:");
        System.out.println(queue);
        sort(queue);
        System.out.println("Sorted Queue:");
        System.out.println(queue);
    }
}
```

Output:

```
Original Queue:
[5, 3, 8, 1, 4]
Sorted Queue:
[1, 3, 4, 5, 8]
```

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.

Solution:



```java
package com.task4;
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);
       }
       // Push sorted elements from tempStack back to original stack
       while (!tempStack.isEmpty()) {
           stack.push(tempStack.pop());
       }
   }
   public static void main(String[] args) {
       Stack<Integer> stack = new Stack<>();
       stack.push(5);
       stack.push(3);
       stack.push(8);
       stack.push(1);
       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, 3, 8, 1, 4]
Sorted Stack:
[8, 5, 4, 3, 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.

Solution:

```
package com.task5;
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}
```

```
package com.task5;
public class RemoveDuplicates {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode current = head;
        while (current != null && current.next != null) {
            if (current.val == current.next.val) {
                // Skip the next node
                current.next = current.next.next;
            } else {
                // Move to the next node
                current = current.next;
            }
        }
        return head;
    }
    public static void main(String[] args) {
        RemoveDuplicates remover = new RemoveDuplicates();
        // Example usage
        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);
        System.out.println("Original List:");
        printList(head);
        ListNode result = remover.deleteDuplicates(head);
        System.out.println("List after removing duplicates:");
        printList(result);
    }
    private static void printList(ListNode head) {
        ListNode current = head;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println();
    }
}
```

Output:

```
Original List:
1 1 2 3 3
List after removing duplicates:
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.

Solution:

```
package com.task6;
import java.util.Stack;
public class SequenceSearch {
   public static boolean isSequencePresent(Stack<Integer> stack, int[]
sequence) {
       Stack<Integer> tempStack = new Stack<>();
       int sequenceIndex = sequence.length - 1; // Start from the last
element of the sequence
       // Iterate through the stack
       while (!stack.isEmpty()) {
           int current = stack.pop();
           // If current element matches the next element in the sequence
           if (current == sequence[sequenceIndex]) {
               sequenceIndex--; // Move to the previous element in the
sequence
               // If all elements in the sequence have been found
               if (sequenceIndex < 0) {
                   return true; // Sequence found
               }
           } else {
               // Push unmatched elements back to the temporary stack
               tempStack.push(current);
               // Reset sequence index to the last element
               sequenceIndex = sequence.length - 1;
           }
       }
```

```
        // Push unmatched elements back to the original stack
        while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
        }
        return false; // Sequence not found
    }
    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);
        stack.push(7);
        int[] sequence1 = {3, 4, 5}; // Present in the stack
        int[] sequence2 = {5, 6, 7, 8}; // Not present in the stack
        System.out.println("Sequence 1 present in stack: " +
isSequencePresent(stack, sequence1));
        System.out.println("Sequence 2 present in stack: " +
isSequencePresent(stack, sequence2));
    }
}
```

Output:

```
Sequence 1 present in stack: true
Sequence 2 present in 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).

Solution:

```java
package com.task7;
class ListNode {
   int val;
   ListNode next;
   ListNode(int val) {
       this.val = val;
       this.next = null;
   }
}
```

```java
package com.task7;
public class MergeSortedList {
   public static ListNode mergeLists(ListNode l1, ListNode l2) {
       // Base cases
       if (l1 == null) {
           return l2;
       }
       if (l2 == null) {
           return l1;
       }
       // Choose the smaller node as the head of the merged list
       if (l1.val < l2.val) {
           l1.next = mergeLists(l1.next, l2);
           return l1;
       } else {
           l2.next = mergeLists(l1, l2.next);
           return l2;
       }
   }
   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) {
       // Example usage
       ListNode l1 = new ListNode(1);
       l1.next = new ListNode(3);
       l1.next.next = new ListNode(5);
       ListNode l2 = new ListNode(2);
       l2.next = new ListNode(4);
       l2.next.next = new ListNode(6);
       System.out.println("List 1:");
       printList(l1);
       System.out.println("List 2:");
       printList(l2);
       ListNode mergedList = mergeLists(l1, l2);
       System.out.println("Merged List:");
       printList(mergedList);
   }
}
```

Output:

```
List 1:
1 3 5
List 2:
2 4 6
Merged List:
1 2 3 4 5 6
```

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.

Solution:

```
package com.task8;
public class CircularQueueBinarySearch {
   public static int binarySearch(int[] nums, int target) {
       int left = 0;
       int right = nums.length - 1;
       // Find the pivot point (rotation index)
       while (left < right) {
           int mid = left + (right - left) / 2;
           if (nums[mid] > nums[right]) {
               left = mid + 1;
           } else {
               right = mid;
           }
       }
       int pivot = left;
       // Perform binary search
       left = 0;
       right = nums.length - 1;
       while (left <= right) {
           int mid = left + (right - left) / 2;
           int adjustedMid = (mid + pivot) % nums.length; // Adjusted mid index for circular array
           if (nums[adjustedMid] == target) {
               return adjustedMid;
```

```
            } else if (nums[adjustedMid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1; // Element not found
    }
    public static void main(String[] args) {
        int[] nums = {4, 5, 6, 7, 0, 1, 2}; // Example circularly sorted array
        int target = 0; // Target element to search
        int index = binarySearch(nums, target);
        if (index != -1) {
            System.out.println("Element " + target + " found at index " +
index);
        } else {
            System.out.println("Element " + target + " not found");
        }
    }
}
```

Output:

```
Element 0 found at index 4
```