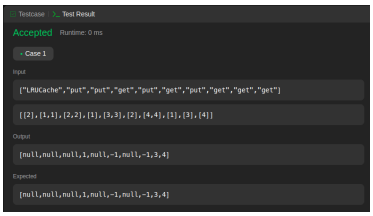
Question.1

```
class LRUCache {
    class Node{
      int key;
      int value;

      Node prev;
      Node next;

      Node(int key, int value){
         this.key= key;
         this.value= value;
      }
   }

   public Node[] map;
   public int count, capacity;
   public Node head, tail;

   public LRUCache(int capacity) {

      this.capacity= capacity;
      count= 0;

      map= new Node[10_000+1];

      head= new Node(0,0);
      tail= new Node(0,0);

      head.next= tail;
      tail.prev= head;

      head.prev= null;
      tail.next= null;
   }
    public void deleteNode(Node node){
      node.prev.next= node.next;
      node.next.prev= node.prev;

      return;
   }

   public void addToHead(Node node){
      node.next= head.next;
      node.next.prev= node;
      node.prev= head;
```

```
    head.next= node;

    return;
}

public int get(int key) {
    if( map[key] != null ){

        Node node= map[key];

        int nodeVal= node.value;

        deleteNode(node);

        addToHead(node);

        return nodeVal;
    }
    else
        return -1;

}

public void put(int key, int value) {

    if(map[key] != null){

        Node node= map[key];

        node.value= value;

        deleteNode(node);

        addToHead(node);

    } else {

        Node node= new Node(key,value);

        map[key]= node;

        if(count < capacity){
            count++;
            addToHead(node);
        }
        else {

            map[tail.prev.key]= null;
```

```
            deleteNode(tail.prev);

            addToHead(node);
        }
    }

    return;

  }
}
```

Output:

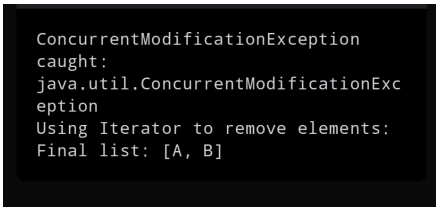


Question 2.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ConcurrentModificationDemo {
   public static void main(String[] args) {

       List<String> myList = new ArrayList<>();
       myList.add("A");
       myList.add("B");
       myList.add("C");


       try {
           for (String item : myList) {
               if (item.equals("B")) {
                   myList.remove(item);
               }
           }
       } catch (ConcurrentModificationException e) {
           System.out.println("ConcurrentModificationException caught: " + e);
       }


       System.out.println("Using Iterator to remove elements:");
       Iterator<String> iterator = myList.iterator();
       while (iterator.hasNext()) {
           String item = iterator.next();
```

```
            if (item.equals("C")) {
                iterator.remove();  // Proper way to remove element
            }
        }

        System.out.println("Final list: " + myList);
    }
}
```

Output:



The for-each loop in Java is syntactic sugar for an iterator. When using this loop, if the underlying collection is modified directly it causes a ConcurrentModificationException because the iterator's internal state becomes inconsistent with the collection's state. The iterator detects this inconsistency and throws the exception to prevent undefined behavior.

Question.3

Defining the Annotation

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface LogExecutionTime{
}
```

Creating an Annotation Processor

```
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.TypeElement;
```

```
import javax.tools.Diagnostic;
import java.util.Set;

@SupportedAnnotationTypes("your.package.LogExecutionTime")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class LogExecutionTimeProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
        for (Element element : roundEnv.getElementsAnnotatedWith(LogExecutionTime.class))
{
            if (element.getKind() == ElementKind.METHOD) {
                ExecutableElement method = (ExecutableElement) element;
                processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE, "Processing
method: " + method.getSimpleName());

            }
        }
        return true;
    }
}
```

Question.4

```
   public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class Codec {

    // Serialize a tree to a single string
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }

    // Helper function for serialization
    private void serializeHelper(TreeNode root, StringBuilder sb) {
```

```
        if (root == null) {
            sb.append("null,");
            return;
        }
        sb.append(root.val).append(",");
        serializeHelper(root.left, sb);
        serializeHelper(root.right, sb);
    }

    // Deserialize your encoded data to tree
    public TreeNode deserialize(String data) {
        List<String> nodes = new LinkedList<>(Arrays.asList(data.split(",")));
        return deserializeHelper(nodes);
    }

    // Helper function for deserialization
    private TreeNode deserializeHelper(List<String> nodes) {
        if (nodes.get(0).equals("null")) {
            nodes.remove(0);
            return null;
        }
        TreeNode root = new TreeNode(Integer.parseInt(nodes.remove(0)));
        root.left = deserializeHelper(nodes);
        root.right = deserializeHelper(nodes);
        return root;
    }

    public static void main(String[] args) {
        Codec codec = new Codec();
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.right.left = new TreeNode(4);
        root.right.right = new TreeNode(5);

        String serialized = codec.serialize(root);
        System.out.println("Serialized: " + serialized);

        TreeNode deserialized = codec.deserialize(serialized);
        System.out.println("Deserialized Tree Root Value: " + deserialized.val);
    }
}
```

Output:

```
Serialized:
1,2,null,null,3,4,null,null,5,null,
null,
Deserialized Tree Root Value: 1
```

Question 5.

```
class TrieNode {
    public TrieNode[] children;
    public boolean isEndOfWord;

    public TrieNode() {
        children = new TrieNode[26]; // Since inputs are lowercase letters a-z
        isEndOfWord = false;
    }
}

class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.isEndOfWord = true;
    }

    // Returns if the word is in the trie.
    public boolean search(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return false;
            }
            node = node.children[index];
        }
        return node.isEndOfWord;
    }

    // Returns if there is any word in the trie that starts with the given prefix.
    public boolean startsWith(String prefix) {
```

```
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return false;
            }
            node = node.children[index];
        }
        return true;
    }

    public static void main(String[] args) {
        Trie trie = new Trie();

        trie.insert("apple");
        System.out.println(trie.search("apple"));   // Returns true
        System.out.println(trie.search("app"));     // Returns false
        System.out.println(trie.startsWith("app")); // Returns true
        trie.insert("app");
        System.out.println(trie.search("app"));     // Returns true
    }
}
```

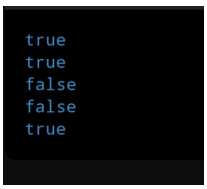
```
true
false
true
true
```

Question 6.

```
  import java.util.Stack;

public class ValidParentheses {
    public static boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

        for (char c : s.toCharArray()) {

            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else {

                if (stack.isEmpty()) {
                    return false;
                }

                char top = stack.pop();
```

```
                if ((c == ')' && top != '(') ||
                    (c == '}' && top != '{') ||
                    (c == ']' && top != '[')) {
                    return false;
                }
            }
        }


        return stack.isEmpty();
    }

    public static void main(String[] args) {
        String test1 = "()";
        String test2 = "()[]{}";
        String test3 = "(]";
        String test4 = "([)]";
        String test5 = "{[]}";

        System.out.println(isValid(test1));
        System.out.println(isValid(test2));
        System.out.println(isValid(test3));
        System.out.println(isValid(test4));
        System.out.println(isValid(test5));
    }
}
```

Output:

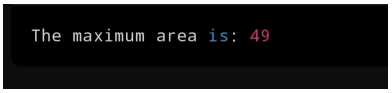


Question.7

```
    public class ContainerWithMostWater {

    public static int maxArea(int[] height) {
        int maxArea = 0;
        int left = 0;
        int right = height.length - 1;

        while (left < right) {
            int currentHeight = Math.min(height[left], height[right]);
            int currentWidth = right - left;
            int currentArea = currentHeight * currentWidth;
            maxArea = Math.max(maxArea, currentArea);
```

```
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return maxArea;
    }

    public static void main(String[] args) {
        int[] height = {1,8,6,2,5,4,8,3,7};
        System.out.println("The maximum area is: " + maxArea(height));
    }
}
```

Output:



Question.8

```
import java.util.PriorityQueue;

public class KthLargestElement {

    public static int findKthLargest(int[] nums, int k) {
        // Create a min-heap (PriorityQueue) with initial capacity of k
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(k);

        // Add the first k elements to the heap
        for (int i = 0; i < k; i++) {
            minHeap.add(nums[i]);
        }

        // Iterate through the remaining elements
        for (int i = k; i < nums.length; i++) {
            if (nums[i] > minHeap.peek()) {
                // Remove the smallest element and add the current element
                minHeap.poll();
                minHeap.add(nums[i]);
            }
        }

        // The root of the heap is the k-th largest element
        return minHeap.peek();
    }
```

```
    public static void main(String[] args) {
        int[] nums = {3, 2, 1, 5, 6, 4};
        int k = 2;
        System.out.println("The " + k + "th largest element is " + findKthLargest(nums, k)); //
Output: 5
    }
}
```

Output:

```
The 2th largest element is 5
```

Question.9

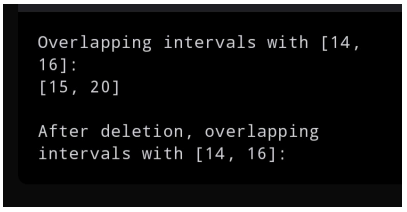
```
     class Interval {
    int start;
    int end;

    public Interval(int start, int end) {
        this.start = start;
        this.end = end;
    }
}
    import java.util.*;

class IntervalTree {
    static class Node {
        Interval interval;
        int maxEnd; // Maximum end value in the subtree rooted at this node
        Node left;
        Node right;

        public Node(Interval interval) {
            this.interval = interval;
            this.maxEnd = interval.end;
            this.left = null;
            this.right = null;
        }
    }

    private Node root;

    public IntervalTree() {
        this.root = null;
    }

    // Helper function to update maxEnd value of a node
    private void updateMaxEnd(Node node) {
```

```
    if (node != null) {
        int maxChildEnd = 0;
        if (node.left != null) {
            maxChildEnd = Math.max(maxChildEnd, node.left.maxEnd);
        }
        if (node.right != null) {
            maxChildEnd = Math.max(maxChildEnd, node.right.maxEnd);
        }
        node.maxEnd = Math.max(node.interval.end, maxChildEnd);
    }
}

// Insert a new interval into the interval tree
public void insertInterval(int start, int end) {
    Interval newInterval = new Interval(start, end);
    this.root = insertInterval(this.root, newInterval);
}

private Node insertInterval(Node node, Interval newInterval) {
    if (node == null) {
        return new Node(newInterval);
    }

    // Insert in BST manner based on start value
    if (newInterval.start < node.interval.start) {
        node.left = insertInterval(node.left, newInterval);
    } else {
        node.right = insertInterval(node.right, newInterval);
    }

    // Update maxEnd for current node after insertion
    updateMaxEnd(node);

    return node;
}

// Delete an interval from the interval tree
public void deleteInterval(int start, int end) {
    Interval toDelete = new Interval(start, end);
    this.root = deleteInterval(this.root, toDelete);
}

private Node deleteInterval(Node node, Interval toDelete) {
    if (node == null) {
        return null;
    }

    // Find the interval to delete
```

```
    if (toDelete.start < node.interval.start) {
        node.left = deleteInterval(node.left, toDelete);
    } else if (toDelete.start > node.interval.start) {
        node.right = deleteInterval(node.right, toDelete);
    } else {
        // Found the node to delete
        if (toDelete.end == node.interval.end) {
            // Case 1: Node to delete has no children or only one child
            if (node.left == null) {
                return node.right;
            } else if (node.right == null) {
                return node.left;
            }

            // Case 2: Node to delete has two children
            Node minNode = findMin(node.right);
            node.interval = minNode.interval;
            node.right = deleteInterval(node.right, minNode.interval);
        } else {
            // Recursively delete based on end value
            if (toDelete.end < node.interval.end) {
                node.left = deleteInterval(node.left, toDelete);
            } else {
                node.right = deleteInterval(node.right, toDelete);
            }
        }
    }

    // Update maxEnd for current node after deletion
    updateMaxEnd(node);

    return node;
}

// Helper function to find the node with minimum value in a subtree
private Node findMin(Node node) {
    while (node.left != null) {
        node = node.left;
    }
    return node;
}

// Find all intervals that overlap with the given interval [start, end]
public List<Interval> findOverlappingIntervals(int start, int end) {
    List<Interval> result = new ArrayList<>();
    findOverlappingIntervals(root, start, end, result);
    return result;
}
```

```
    private void findOverlappingIntervals(Node node, int start, int end, List<Interval> result) {
        if (node == null) {
            return;
        }

        // Check if current node's interval overlaps with [start, end]
        if (node.interval.start <= end && node.interval.end >= start) {
            result.add(node.interval);
        }

        // Traverse left subtree if necessary
        if (node.left != null && node.left.maxEnd >= start) {
            findOverlappingIntervals(node.left, start, end, result);
        }

        // Traverse right subtree if necessary
        if (node.right != null && node.right.interval.start <= end) {
            findOverlappingIntervals(node.right, start, end, result);
        }
    }
}
```

Output:



Question 10.

```
import java.util.Scanner;

public class PalindromeChecker {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a string: ");
        String input = scanner.nextLine();
        scanner.close();

        if (isPalindrome(input)) {
            System.out.println("The string is a palindrome.");
        } else {
            System.out.println("The string is not a palindrome.");
        }
    }
```

```
    public static boolean isPalindrome(String str) {
        // Remove all non-alphanumeric characters and convert to lower case
        String cleanedStr = str.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

        // Initialize two pointers
        int left = 0;
        int right = cleanedStr.length() - 1;

        // Compare characters from both ends
        while (left < right) {
            if (cleanedStr.charAt(left) != cleanedStr.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }

        return true;
    }
}
```

Output: