W3 - Data Structure & Algorithms

Problem 1: Remove a Loop from a Singly Linked List

Overview: Detect and remove a loop in a singly linked list.

Algorithm Steps:

1. **Detect Loop**:
   - Use Floyd's Cycle detection algorithm (Tortoise and Hare approach) where two pointers move through the list at different speeds.
   - If there's a loop, they will eventually meet inside the loop.
2. **Find Loop Start**:
   - Once a loop is detected, initialize one pointer to the start of the list and keep the other at the meeting point.
   - Move both pointers one step at a time. The point where they meet again is the start of the loop.
3. **Remove Loop**:
   - Keep one pointer fixed at the start of the loop and move the other around the loop until it reaches the node just before the start of the loop.
   - Set the next pointer of this node to null to remove the loop.
4. **Code**

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

    Node(int d) {
        data = d;
        next = null;
    }
}
public class LinkedList {
    Node head; // head of the list

    // Function to detect and remove loop in a linked list
    public void removeLoop() {
        Node slow = head, fast = head;
        boolean loopExists = false;

        // Detect loop
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                loopExists = true;
                break;
            }
        }

        // Remove loop if exists
        if (loopExists) {
            slow = head;
            while (slow.next != fast.next) {
                slow = slow.next;
                fast = fast.next;
            }
            fast.next = null; // Remove loop
        }
    }
}
```

Problem 2: Implement Circular Linked List for Josephus Problem

Overview: Implement a circular linked list and use it to solve the Josephus problem, where people are arranged in a circle and eliminated every kth person until one remains.

Algorithm Steps:

1. **Build Circular Linked List**:
    - Create a circular linked list with nodes representing each person in the problem.
2. **Josephus Solution**:
    - Start from the first person and iterate through the list, counting up to k.
    - Eliminate the kth person by removing the node from the list and close the gap by linking the previous node to the next node.
    - Continue the process until only one node remains in the list.
    - The remaining node represents the position that should be taken to avoid elimination.

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

    Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    Node head = null;
    Node tail = null;

    // Add new node to the list
    public void add(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            tail.next = newNode;
        }
        tail = newNode;
        tail.next = head;
    }

    // Solve Josephus problem
    public int solveJosephus(int k) {
        Node curr = head;
        Node prev = tail;

        while (curr != prev) {
            int count = 1;
            while (count != k) {
                prev = curr;
                curr = curr.next;
                count++;
            }
            prev.next = curr.next; // Remove k-th node
            curr = prev.next;
        }
        return curr.data;
    }
}
```

Problem 3: Queue Simulation for ATM

Overview: Calculate the maximum and average waiting time of people using an ATM, given their arrival and service times.

Algorithm Steps:

1. **Initialize Variables**:

    - Keep a variable for the current time, maximum waiting time, total waiting time, and a queue to represent the line.

2. **Process Each Person**:
    - Iterate over each person. If the ATM is free, serve them immediately. If not, add them to the queue.
    - Update the current time based on service duration.
    - Calculate the waiting time for each person (current time - arrival time).
    - Update the maximum and total waiting times.
3. **Calculate Results**:
    - Maximum waiting time is the highest waiting time recorded.
    - Average waiting time is the total waiting time divided by the number of people.

```
import java.util.PriorityQueue;

public class ATMQueueSimulation {
    public static void main(String[] args) {
        int[] A = {1, 2, 4, 6, 7}; // Arrival times
        int[] D = {3, 5, 1, 2, 4}; // Duration times
        System.out.println("Maximum and Average waiting time: " + calculateWaitTimes(A, D));
    }

    private static String calculateWaitTimes(int[] A, int[] D) {
        PriorityQueue<Integer> queue = new PriorityQueue<>();
        int maxWait = 0;
        int totalWait = 0;
        int currentTime = 0;

        for (int i = 0; i < A.length; i++) {
            currentTime = Math.max(currentTime, A[i]);
            int waitTime = currentTime - A[i];
            maxWait = Math.max(maxWait, waitTime);
            totalWait += waitTime;
            currentTime += D[i];
        }

        double averageWait = (double) totalWait / A.length;
        return "Max: " + maxWait + ", Avg: " + averageWait;
    }
}
```

Problem 4: Check for Balanced Parentheses, Brackets, and Curly Braces

Overview: Determine if a sequence of characters has balanced parentheses, brackets, and curly braces.

Algorithm Steps:

1. **Use a Stack**:
    - Iterate through each character in the sequence.
    - When an opening bracket (`(`, `[`, `{`) is encountered, push it onto the stack.
    - When a closing bracket (`)`, `]`, `}`) is encountered, pop from the stack and check if it matches the corresponding opening bracket. If it doesn't match or the stack is empty, the sequence is unbalanced.
2. **Check Stack at End**:
    - After processing all characters, if the stack is not empty, then the sequence is unbalanced.

```
import java.util.Stack;

public class BalancedParentheses {
    public static boolean isBalanced(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[] testStrings = {"[]", "{}", "()", "(){{[]}}()", "{)", "{{[[}}]]"};
        for (String test : testStrings) {
            System.out
```