Backend Engineer Home Test – Manjusri Pavuluri

1.JSON is a document format used to encode information that is both human-readable and machine-readable. JSON format is explained at http://json.org. Please write a JSON parser that accepts an input JSON string and produces a Map output structure. Do not use any existing library to do the parsing.
Example input JSON:

{“debug”: "on",

"window”:

{“title”: "sample",

"size": 500}}

The parser may be a static function:

public class JSONParser

{ public static Map parse(String json) { ... } }

The corresponding output for the input JSON should be:

Map output = JsonParser.parse(input);

assert output.get("debug").equals("on");

assert (Map(output.get("window")).get("title").equals("sample")

assert (Map(output.get("window")).get("size").equals(500)?

Solution:

```
package defaultpackage;

import java.util.*;
 public class JSONParser
 {
       public static Map parseJson(String jsonString)
       {
             Map resultMap = new HashMap<>();
       // Remove curly braces and split JSON into key-value pairs
       jsonString = jsonString.replace("{", "").replace("}", "");
       String[] keyValuePairs = jsonString.split(",");
       for (String pair : keyValuePairs)
       {
             String[] keyValue = pair.split(":");
             String key = keyValue[0].trim();
             String value = keyValue[1].trim();
             // Check if the value is a string, number, or nested JSON
             if (value.startsWith("\"") && value.endsWith("\""))
             {
                   resultMap.put(key, value.substring(1, value.length() -
1));
                   }
             else if (value.matches("\\d+"))
```

```
                {
                        resultMap.put(key, Integer.parseInt(value));
                        }
            else if (value.startsWith("{") && value.endsWith("}"))
            {
                  resultMap.put(key, parseJson(value));
                  } }
      return resultMap; }
      public static void main(String[] args)
      {
            String jsonString = "{\"name\": \"Manju\", \"age\": 20,
\"address\": {\"city\": \"New York\", \"zip\": \"10001\"}}";
      Map resultMap = parseJson(jsonString);
      System.out.println(resultMap); }


      }
```

2. Assuming you have a binary tree of integers, come up with an algorithm to serialize and deserialize it. Assumptions: a. Serialization should be to a String and from a String. b. There are no cyclic connections in the tree Model: public class Node { Node left; Node right; int num; } public interface TreeSerializer { String serialize(Node root); Node deserialize(String str); }

I. Implement a TreeSerializer given the above assumptions.

Solution:

```
package defaultpackage;

import java.util.*;

//Node class represents a node in the binary tree
class Node {
 int num;
 Node left;
 Node right;

 // Constructor to initialize a node with a value
 Node(int num) {
     this.num = num;
 }
}

//TreeSerializer5 interface defines the methods for tree serialization and
deserialization
interface TreeSerializer5 {
 String serialize(Node root);
 Node deserialize(String str);
}

//NonCyclicTreeSerializer implements TreeSerializer5 to serialize and
deserialize a binary tree
public class NonCyclicTreeSerializer implements TreeSerializer5 {
 // Serialize the tree to a string
 public String serialize(Node root) {
     if (root == null) {
         return "null";
```

```
        }
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }

    // Recursive helper method to serialize the tree
    private void serializeHelper(Node node, StringBuilder sb) {
        if (node == null) {
            sb.append("null").append(",");
        } else {
            sb.append(node.num).append(","); // Append the node's value
            serializeHelper(node.left, sb); // Recursively serialize the left
subtree
            serializeHelper(node.right, sb); // Recursively serialize the right
subtree
        }
    }

    // Deserialize the string to a tree
    public Node deserialize(String str) {
        Deque<String> nodes = new LinkedList<>(Arrays.asList(str.split(",")));
        return deserializeHelper(nodes);
    }

    // Recursive helper method to deserialize the tree
    private Node deserializeHelper(Deque<String> nodes) {
        String val = nodes.poll(); // Get the next value from the deque
        if (val.equals("null")) {
            return null;
        }
        Node node = new Node(Integer.parseInt(val)); // Create a new node with
the parsed value
        node.left = deserializeHelper(nodes); // Recursively deserialize the
left subtree
        node.right = deserializeHelper(nodes); // Recursively deserialize the
right subtree
        return node;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter space-separated tree values: ");
        String[] values = scanner.nextLine().split(" ");
        Node root = buildTree(values, 0);

        NonCyclicTreeSerializer serializer = new NonCyclicTreeSerializer();

        // Serialize and print the serialized tree
        String serialized = serializer.serialize(root);
        System.out.println("Serialized: " + serialized);

        // Deserialize the serialized string and print the root value
        Node deserialized = serializer.deserialize(serialized);
        System.out.println("Deserialized root value: " + deserialized.num);
```

```
        scanner.close();
    }

    // Recursive method to build the binary tree from input values
    private static Node buildTree(String[] values, int index) {
        if (index >= values.length || values[index].equals("null")) {
            return null;
        }

        Node node = new Node(Integer.parseInt(values[index])); // Create a new
node with the parsed value
        node.left = buildTree(values, 2 * index + 1); // Recursively build the
left subtree
        node.right = buildTree(values, 2 * index + 2); // Recursively build the
right subtree

        return node;
    }
}
```

II.Implement a TreeSerializer that takes into account a "cyclic tree". Your implementation should throw a RuntimeException when a cyclic connection is found in the Tree. Bonus: create an implementation that works even with a "cyclic tree". See "cyclic tree" example: Solution:

```
package defaultpackage;

import java.util.*;

class Node5 {
    int num;
    Node5 left;
    Node5 right;

    Node5(int num) {
        this.num = num;
    }
}

interface TreeSerializer {
    String serialize(Node5 root);
    Node5 deserialize(String str);
}

public class CyclicTreeSerializer implements TreeSerializer {
    private Set<Node5> visitedNodes = new HashSet<>();
    private Set<Node5> currentPath = new HashSet<>(); // To track current
path during traversal

    public String serialize(Node5 root) {
        visitedNodes.clear();
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }

    private void serializeHelper(Node5 node, StringBuilder sb) {
        if (node == null) {
```

```
            sb.append("null").append(",");
            return;
        }

        if (currentPath.contains(node)) {
            throw new RuntimeException("Cyclic connection found in the
tree");
        }

        currentPath.add(node);
        sb.append(node.num).append(",");
        serializeHelper(node.left, sb);
        serializeHelper(node.right, sb);
        currentPath.remove(node);
        visitedNodes.add(node);
    }

    public Node5 deserialize(String str) {
        Deque<String> nodes = new
LinkedList<>(Arrays.asList(str.split(",")));
        visitedNodes.clear();
        return deserializeHelper(nodes);
    }

    private Node5 deserializeHelper(Deque<String> nodes) {
        String val = nodes.poll();
        if (val.equals("null")) {
            return null;
        }
        Node5 node = new Node5(Integer.parseInt(val));

        if (visitedNodes.contains(node)) {
            throw new RuntimeException("Cyclic connection found in the
tree");
        }

        visitedNodes.add(node);
        currentPath.add(node);
        node.left = deserializeHelper(nodes);
        node.right = deserializeHelper(nodes);
        currentPath.remove(node);
        return node;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter space-separated tree values: ");
        String[] values = scanner.nextLine().split(" ");
        Node5 root = buildTree(values, 0);

        CyclicTreeSerializer serializer = new CyclicTreeSerializer();

        try {
            String serialized = serializer.serialize(root);
            System.out.println("Serialized: " + serialized);
        } catch (RuntimeException e) {
```

```
            System.out.println("Error during serialization: " +
e.getMessage());
        }

        try {
            Node5 deserialized =
serializer.deserialize(serializer.serialize(root));
            System.out.println("Deserialized root value: " +
deserialized.num);
        } catch (RuntimeException e) {
            System.out.println("Error during deserialization: " +
e.getMessage());
        }

        scanner.close();
    }

    private static Node5 buildTree(String[] values, int index) {
        if (index >= values.length || values[index].equals("null")) {
            return null;
        }

        Node5 node = new Node5(Integer.parseInt(values[index]));
        node.left = buildTree(values, 2 * index + 1);
        node.right = buildTree(values, 2 * index + 2);

        return node;
    }
}
```

III. Suggest changes that should be done in order to support any data type (as opposed to only an int data type)

```
package defaultpackage;

import java.util.*;

//Update the Node class to use a generic type parameter T
class Node6<T> {
 T data;
 Node6<T> left;
 Node6<T> right;

 Node6(T data) {
     this.data = data;
 }
}

//Update the TreeSerializer interface to use a generic type parameter T
interface TreeSerializer7<T> {
 String serialize(Node6<T> root);
 Node6<T> deserialize(String str);
}

//Update the CyclicTreeSerializer class to use a generic type parameter T
```

```java
class CyclicTreeSerializer3<T> implements TreeSerializer7<T> {
 private Set<Node6<T>> visitedNodes = new HashSet<>();

 // Serialize the tree to a string
 public String serialize(Node6<T> root) {
     visitedNodes.clear();
     StringBuilder sb = new StringBuilder();
     serializeHelper(root, sb);
     return sb.toString();
 }

 // Recursive helper for serialization
 private void serializeHelper(Node6<T> node, StringBuilder sb) {
     if (node == null) {
         sb.append("null").append(",");
         return;
     }

     if (visitedNodes.contains(node)) {
         throw new RuntimeException("Cyclic connection found in the tree");
     }

     visitedNodes.add(node);
     sb.append(node.data).append(",");
     serializeHelper(node.left, sb);
     serializeHelper(node.right, sb);
 }

 // Deserialize the string to a tree
 public Node6<T> deserialize(String str) {
     Deque<String> nodes = new LinkedList<>(Arrays.asList(str.split(",")));
     visitedNodes.clear();
     return deserializeHelper(nodes);
 }

 // Recursive helper for deserialization
 private Node6<T> deserializeHelper(Deque<String> nodes) {
     String val = nodes.poll();
     if (val.equals("null")) {
         return null;
     }
     Node6<T> node = new Node6<>(null); // Replace null with actual data type
     // Parse val to the appropriate data type T and assign to node.data

     if (visitedNodes.contains(node)) {
         throw new RuntimeException("Cyclic connection found in the tree");
     }

     visitedNodes.add(node);
     node.left = deserializeHelper(nodes);
     node.right = deserializeHelper(nodes);
     return node;
 }

 // Main method for testing
 public static void main(String[] args) {
     Scanner scanner = new Scanner(System.in);
```

```
        System.out.print("Enter space-separated tree values: ");
        String[] values = scanner.nextLine().split(" ");
        Node6<String> root = buildTree(values, 0); // Use the appropriate data
type

        CyclicTreeSerializer3<String> serializer = new
CyclicTreeSerializer3<>();

        try {
            String serialized = serializer.serialize(root);
            System.out.println("Serialized: " + serialized);
        } catch (RuntimeException e) {
            System.out.println("Error during serialization: " + e.getMessage());
        }

        try {
            Node6<String> deserialized =
serializer.deserialize(serializer.serialize(root));
            System.out.println("Deserialized root value: " + deserialized.data);
        } catch (RuntimeException e) {
            System.out.println("Error during deserialization: " +
e.getMessage());
        }

        scanner.close();
    }

    // Recursive method to build the tree from input values
    private static Node6<String> buildTree(String[] values, int index) {
        if (index >= values.length || values[index].equals("null")) {
            return null;
        }

        Node6<String> node = new Node6<>(values[index]); // Use the appropriate
data type
        node.left = buildTree(values, 2 * index + 1);
        node.right = buildTree(values, 2 * index + 2);

        return node;
    }
}
```