DATA STRUCTURE AND ALGORITHMS (DSA711S)

ASSIGNMENT ONE 2024

GROUP MEMBERS

MODE OF STUDY	STUDENT NUMBER	STUDENT NAME
PART TIME	211089427	JUSTINA NGHINYENGWA
FULL TIME	222023805	PAULUS S. NANGUDI
FULL TIME	218017758	TANGENI DANIEL
FULL TIME	222088001	PENEYAMBEKO HAINGHUMBI
FULL TIME	222076690	ULAMBA MATHEUS

QUESTIONS

A. Write a java class (call it ServerNode) for the server node.

```
class ServerNode {

    private List<ClientNode> clients;

    private String serverName;

    public ServerNode(String name) {

        this.serverName = name;

        this.clients = new ArrayList<>();

    }
    public void addClient(ClientNode client) {
        clients.add(client);
    }
    public void brokerMessage(String message, ClientNode sender) {
        for (ClientNode client : clients) {
            client.receiveMessage(message, sender);
        }
    }
```

B. CLIENTNODE CLASS

```
class ClientNode {
    private String clientId;
    public ClientNode(String id) {
        this.clientId = id;
    }
    public void send(String message, ServerNode server) {
        server.brokerMessage(message, this);
    }
    public void receiveMessage(String message, ClientNode sender) {
        System.out.println(clientId + " received: '" + message + "' from " + sender.clientId);
    }
}
```

```
}
```

C. STARNODE CLASS

```
class Star {
    private List<ClientNode> clients;
    private ServerNode server;

    public Star() {
        this.clients = new ArrayList<>();
        this.server = new ServerNode("CentralServer");
    }

    public void insertNode(ClientNode client) {
        clients.add(client);
        server.addClient(client);
    }

    public void deleteNode(ClientNode client) {
        clients.remove(client);
    }

    public ServerNode getServer() {
        return server;
    }
}
```

Main Class

```
import java.util.ArrayList;

import java.util.List;


public class NetworkSimulation {

    public static void main(String[] args) {

        Star star = new Star();


        // Creating instances of ClientNode and adding them to the star network

        ClientNode device1 = new ClientNode("Device1");

        ClientNode device2 = new ClientNode("Device2");

        ClientNode device3 = new ClientNode("Device3");


        star.insertNode(device1);

        star.insertNode(device2);

        star.insertNode(device3);


        // Sending messages between clients

        device1.send("Hello from Device1", star.getServer());

        device2.send("Hi from Device2", star.getServer());

        device3.send("Greetings from Device3", star.getServer());
    }

}
```

D. I) COMPRESSION ALGORITHM: LZW (LEMPEL-ZIV-WELCH)

Efficient Compression: LZW compression can achieve good compression ratios for a wide range of data types, especially for text data with repetitive patterns. In communication between clients, text data is often exchanged (e.g., messages, documents), making LZW a suitable choice.

Fast Compression and Decompression: LZW compression and decompression algorithms are relatively simple and can be implemented efficiently. This makes them suitable for scenarios where speed is a concern, such as real-time communication between clients.

No Need for Predefined Dictionary: Unlike some other compression algorithms, LZW builds its dictionary dynamically as it encounters new strings in the input data. This means that both the sender and receiver can use the same compression and decompression algorithms without needing to synchronize a predefined dictionary.

ii. Time complexity:

```
import java.util.HashMap;

import java.util.Map;


public class LZWCompression {

    public static void main(String[] args) {

        String message = "Hello from Device1";
        // Compress message
        String compressedMessage = compress(message);
        System.out.println("Compressed message: " + compressedMessage);

        // Decompress message
        String decompressedMessage = decompress(compressedMessage);
        System.out.println("Decompressed message: " + decompressedMessage);
    }

    public static String compress(String message) {
        Map<String, Integer> dictionary = new HashMap<>();
        int dictSize = 256;
        for (int i = 0; i < 256; i++) {
            dictionary.put("" + (char)i, i);
        }

        String w = "";
        StringBuilder result = new StringBuilder();
        for (char c : message.toCharArray()) {
            String wc = w + c;
            if (dictionary.containsKey(wc)) {
                w = wc;
            } else {
                result.append(dictionary.get(w)).append(" ");
                dictionary.put(wc, dictSize++);
                w = "" + c;
            }
        }

        if (!w.equals("")) {
            result.append(dictionary.get(w)).append(" ");
        }

        return result.toString();
    }

    public static String decompress(String compressedMessage) {
        String[] parts = compressedMessage.split(" ");
        Map<Integer, String> dictionary = new HashMap<>();
        int dictSize = 256;
        for (int i = 0; i < 256; i++) {
            dictionary.put(i, "" + (char)i);
        }

        StringBuilder result = new StringBuilder();
        int prevCode = Integer.parseInt(parts[0]);
```

```
        result.append(dictionary.get(prevCode));
        for (int i = 1; i < parts.length; i++) {
            int code = Integer.parseInt(parts[i]);
            String entry;
            if (dictionary.containsKey(code)) {
                entry = dictionary.get(code);
            } else if (code == dictSize) {
                entry = dictionary.get(prevCode) + dictionary.get(prevCode).charAt(0);
            } else {
                throw new IllegalArgumentException("Invalid compressed message");
            }

            result.append(entry);
            dictionary.put(dictSize++, dictionary.get(prevCode) + entry.charAt(0));
            prevCode = code;
        }

        return result.toString();
    }
}
```

Let's analyze the time complexity of the provided code:

COMPRESSION:

1. Initialization of Dictionary: - This part has a constant time complexity since it initializes the dictionary with 256 entries, which is independent of the size of the input message. Thus, it has a time complexity of O(1).

2. Compression Loop: - The loop iterates over each character in the message, and for each character, it performs constant-time operations such as dictionary lookups and updates.

- The worst-case scenario is that each character in the message is unique and not present in the dictionary, causing the dictionary size to grow linearly with the size of the message.

- Therefore, the time complexity of the compression loop is O(n), where n is the size of the input message.

DECOMPRESSION:

1. Initialization of Dictionary: - Similar to compression, this part initializes the dictionary with 256 entries, resulting in a constant time complexity of O(1).

2. Decompression Loop: - The loop iterates over each code in the compressed message, performing dictionary lookups and updates, and appending characters to the result string.

- Similar to compression, in the worst-case scenario, each code corresponds to a new entry in the dictionary, causing the dictionary size to grow linearly with the size of the compressed message.

- Thus, the time complexity of the decompression loop is also O(n), where n is the size of the compressed message.

Overall Time Complexity: Since both compression and decompression have a time complexity of O(n), where n is the size of the input (or compressed) message, the overall time complexity of the code is O(n) for both compression and decompression.