Bahria University,

Karachi Campus

COURSE: CSC-221 DATA STRUCTURES AND ALGORITHM
TERM: FALL 2021, CLASS: BSE- 3 (A/B)

SFL(STUDENT FACULTY LINK)

Engr. Laraib Siddique/ Engr. Ayesha Khan

Signed **Remarks:** **Score:**

SUBMITTED BY:

Ayesha Ahmed
Muhammad Askari
Hamza Zafar

Table of Content

INTRODUCTION & PROBLEM

Student Faculty Link is a type of learning management system. It's software to help educational institutions create and organize learning materials in one place and track the progress of their students. It gives a source of connection to the teacher and the student to communicate with each other in a comprehensive environment.Learning management systems were designed to address problems like training and learning gaps, using analytical data and reporting. LMSs are focused on online learning delivery but support a range of uses, acting as a platform for online content, including courses, both asynchronous based and synchronous based.

PARADIGMS

The paradigms used in SFL are OOP(Object Oriented Programming) . Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability, in which code is divided into classes each having their attributes and functions. we have made use of functions like upload, download and hashing to carry out the procedures of uploading, downloading, encryption and decryption.

ALGORITHM & EXPLANATION

Huffman Coding:

Huffman coding is an efficient method of compressing data without losing information. In computer science, information is encoded as bits—1's and 0's. Strings of bits encode the information that tells a computer which instructions to carry out.

Explanation:

Huffman coding provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message. Symbols that appear more often will be encoded as a shorter-bit string while symbols that aren't used as much will be encoded as longer strings. Since the frequencies of symbols vary across messages, there is no one Huffman coding that will work for all messages. This means that the Huffman coding for sending message X may differ from the Huffman coding used to send message Y. There is an algorithm for generating the Huffman coding for a given message based on the frequencies of symbols in that particular message.

Suppose the string below is to be sent over a network.

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of `8 * 15 = 120` bits are required to send this string.

Using the Huffman Coding technique, we can compress the string to a smaller size.

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

Huffman Coding prevents any ambiguity in the decoding process using the concept of **prefix code** ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.

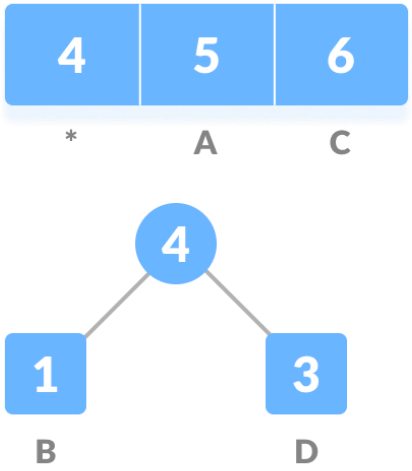
Huffman coding is done with the help of the following steps:

1. Calculate the frequency of each character in the string.

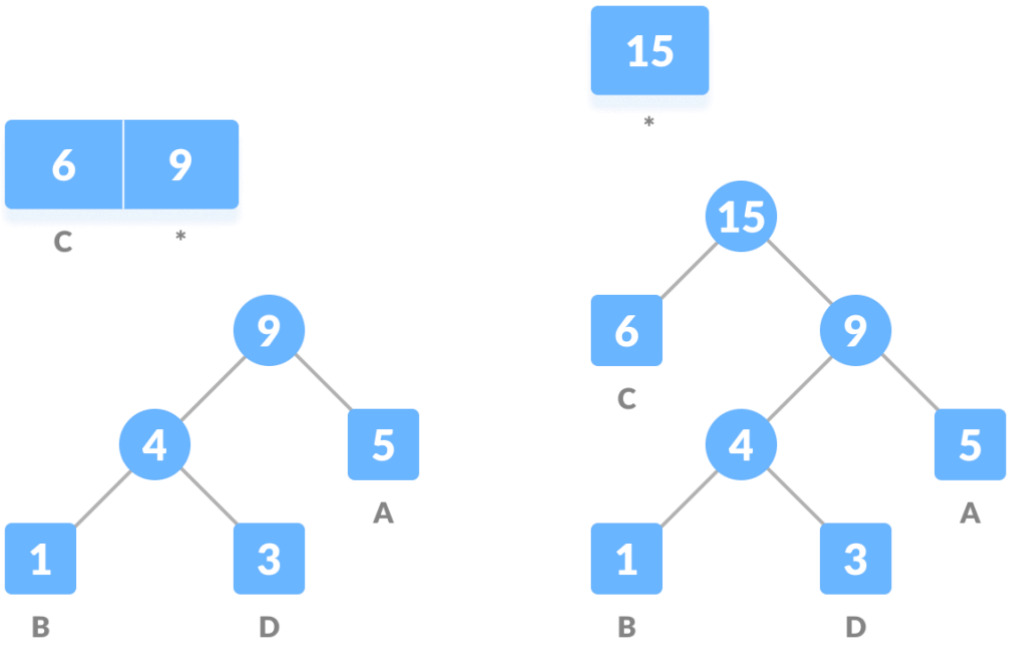
2. Sort the characters in increasing order of the frequency. These are stored in a priority queue *Q*.

3. Make each unique character as a leaf node.

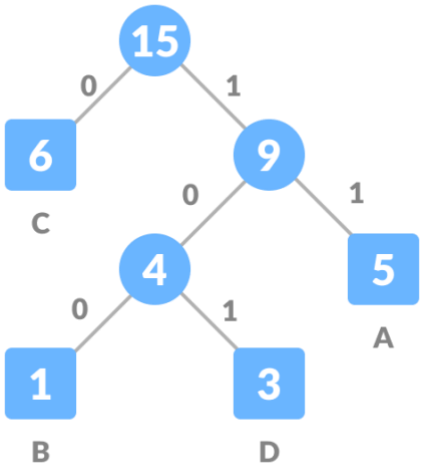
4. Create an empty node *z*. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of *z*. Set the value of the *z* as the sum of the above two minimum



5. Remove these two minimum frequencies from *Q* and add the sum into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node *z* into the tree.
7. Repeat steps 3 to 5 for all the characters. Repeat steps 3 to 5 for all the



8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

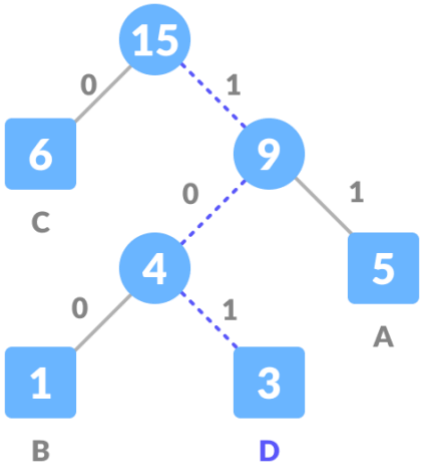
Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to `32 + 15 + 28 = 75`.

Decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



ALGORITHM CODE

Hashing:

```
static int HashFunction(string s)
{
        int total = 0;
        char[] c;
        c = s.ToCharArray();

        // Summing up all the ASCII values
       // of each alphabet in the string
for (int k = 0; k <= c.GetUpperBound(0); k++)
         total += (int)c[k];

        //1000 ~ number of total rows possible in table

return total % 1000;
}

public void EncryptPass(ref string password)
        {
            var sha = SHA256.Create();
            var ByteArr = Encoding.Default.GetBytes(password);
            var encryptPass = sha.ComputeHash(ByteArr);

            password = Convert.ToBase64String(encryptPass);
        }
```

Huffman Algorithm:

Huffman Tree:

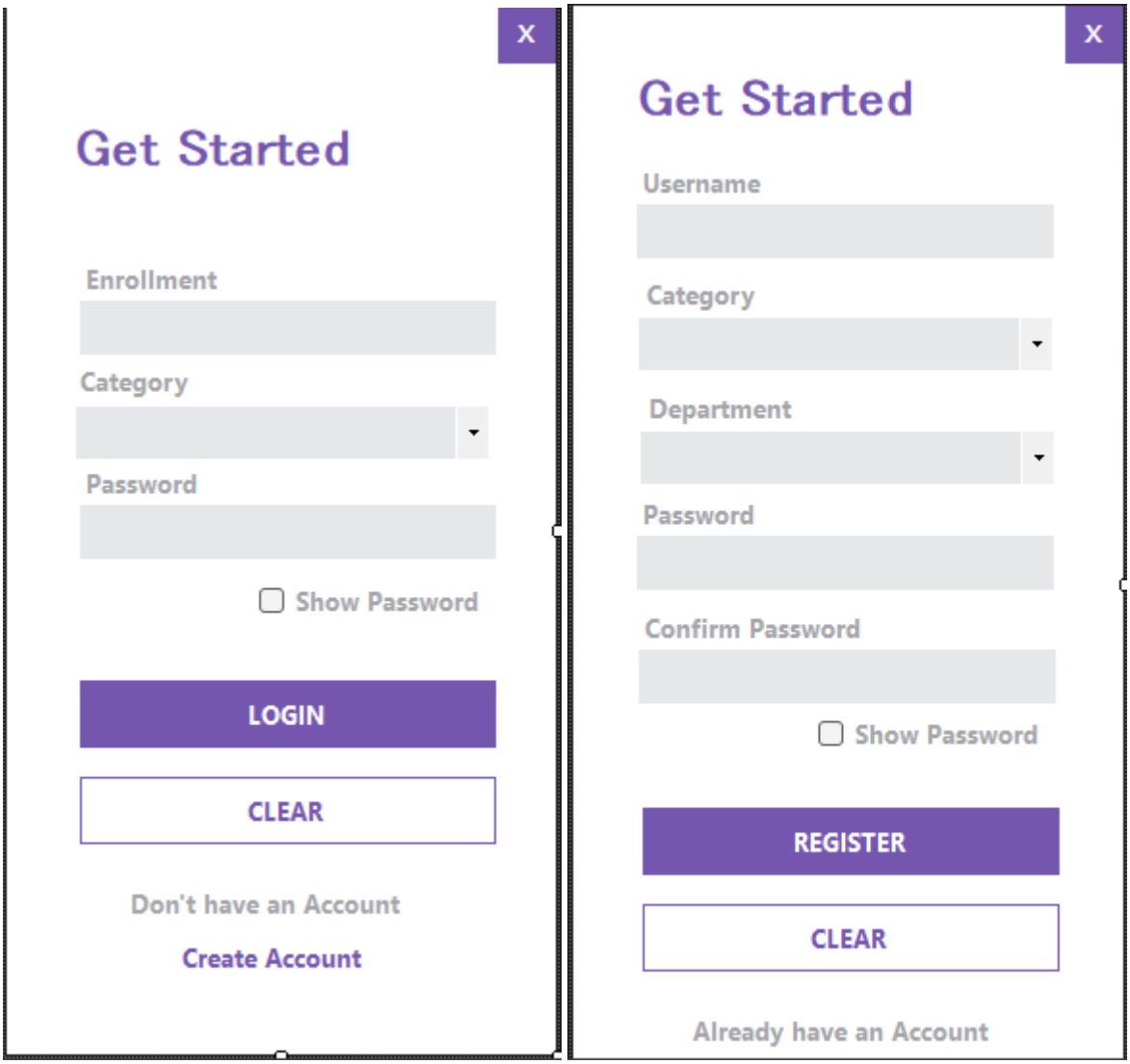
```
public class HuffmanTree
    {
        private List<Node> node = new List<Node>();
        public Node rootNode { get; set; }
        public Dictionary<char, int> frequency = new Dictionary<char, int>();


        // build the huffman tree of the input file
        public void Build_Tree(string input)
        {
            for (int i = 0; i < input.Length; i++)
            {
                // check the tree contains character or not
                if (!frequency.ContainsKey(input[i]))
                {
                    // add the character in tree
                    frequency.Add(input[i], 0);
                }

                frequency[input[i]]++;
            }

            foreach (KeyValuePair<char, int> symbol in frequency)
            {
                node.Add(new Node() { character = symbol.Key, frequency = symbol.Value });
            }



            while (node.Count > 1)
            {
                // ordering the nodes on the basis of frequency
                List<Node> orderedNodes = node.OrderBy(node => node.frequency).ToList<Node>();

                if (orderedNodes.Count >= 2)
                {
                    // Take first two items
                    List<Node> takenNode = orderedNodes.Take(2).ToList<Node>();

                    // Create a parent node by combining the frequencies
                    Node parent = new Node()
                    {
                        character = '*',
                        frequency = takenNode[0].frequency + takenNode[1].frequency,
                        leftNode = takenNode[0],
                        rightNode = takenNode[1]
                    };

                    node.Remove(takenNode[0]);
                    node.Remove(takenNode[1]);
                    node.Add(parent);
                }

                this.rootNode = node.FirstOrDefault();

            }
```

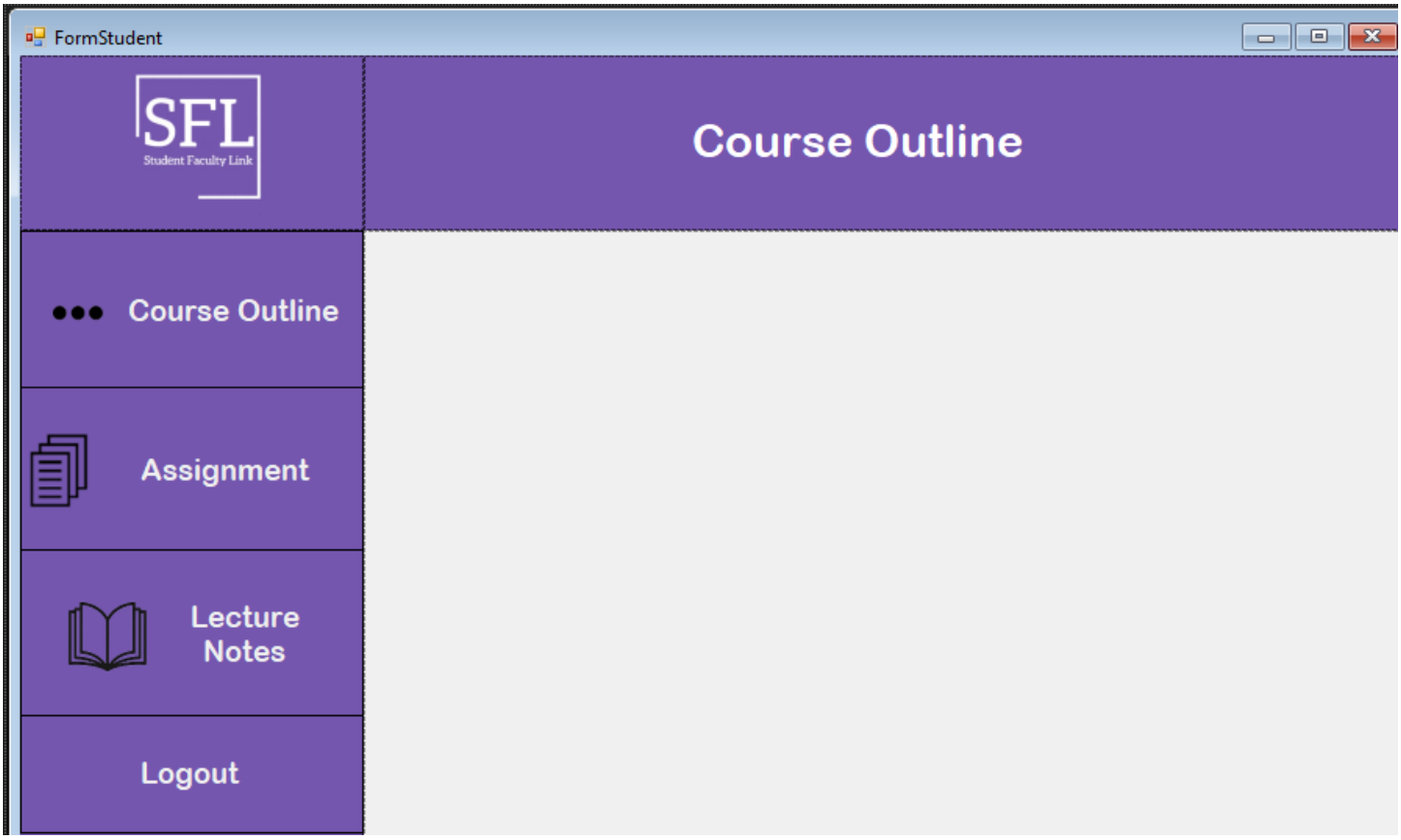
```
        }

        // Encode function for encoding the characters in binary form
        public BitArray Encode(string input)
        {
            List<bool> encodedInput = new List<bool>();

            for (int i = 0; i < input.Length; i++)
            {
                List<bool> encodedCharacter = this.rootNode.Traverse_Tree(input[i], new List<bool>());
                encodedInput.AddRange(encodedCharacter);
            }

            BitArray BitArray = new BitArray(encodedInput.ToArray());

            return BitArray;
        }

        public string Decode(BitArray BitArray)
        {
            Node currentNode = this.rootNode;
            string decoded = "";

            foreach (bool bit in BitArray)
            {
                if (bit)
                {
                    if (currentNode.rightNode != null)
                    {
                        currentNode = currentNode.rightNode;
                    }
                }
                else
                {
                    if (currentNode.leftNode != null)
                    {
                        currentNode = currentNode.leftNode;
                    }
                }

                if (IsLeaf(currentNode))
                {
                    decoded += currentNode.character;
                    currentNode = this.rootNode;
                }
            }

            return decoded;
        }

        public bool IsLeaf(Node node)
        {

            return (node.leftNode == null && node.rightNode == null);

        }
    }

}
```

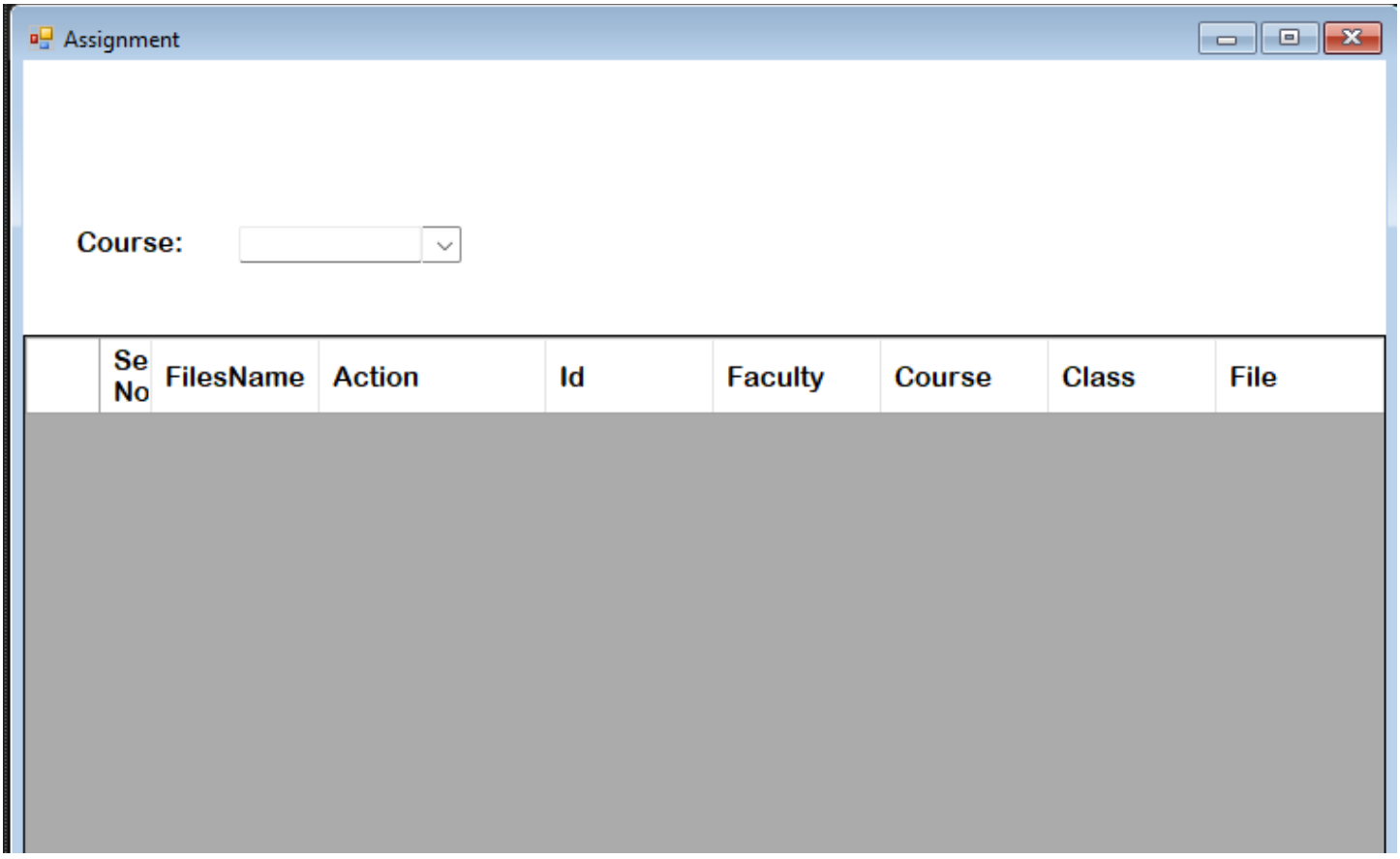
Node Class:

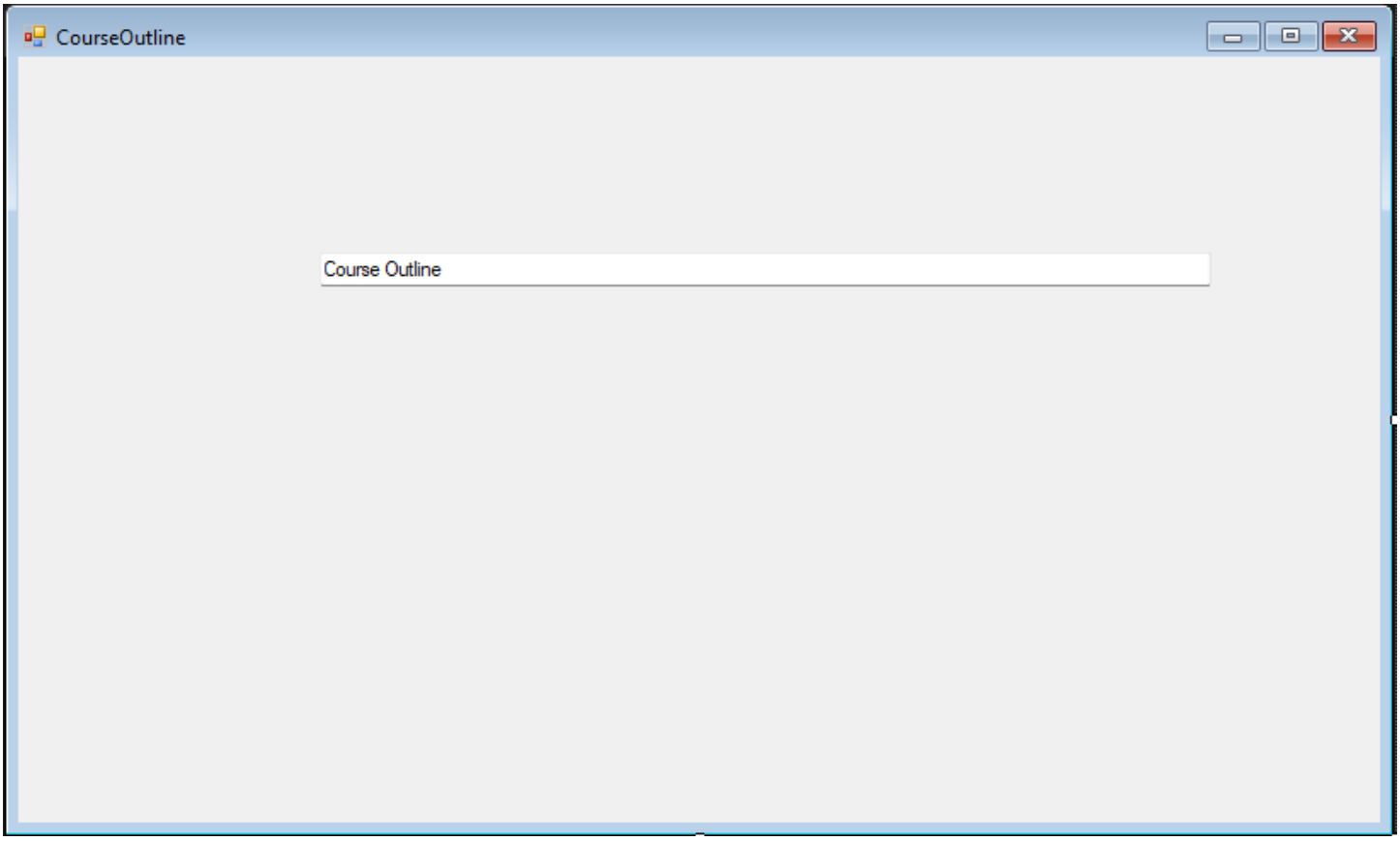
```
public class Node
    {
        public char character { get; set; }
        public int frequency { get; set; }
        public Node rightNode { get; set; }
        public Node leftNode { get; set; }

        public List<bool> Traverse_Tree(char symbol, List<bool> data)
        {
            // Leaf
            if (rightNode == null && leftNode == null)
            {
                if (symbol.Equals(this.character))
                {
                    return data;
                }
                else
                {
                    return null;
                }
            }
            else
            {
                List<bool> left = null;
                List<bool> right = null;

                if (leftNode != null)
                {
                    List<bool> leftPath = new List<bool>();
                    leftPath.AddRange(data);
                    leftPath.Add(false);

                    left = leftNode.Traverse_Tree(symbol, leftPath);
                }

                if (rightNode != null)
                {
                    List<bool> rightPath = new List<bool>();
                    rightPath.AddRange(data);
                    rightPath.Add(true);
                    right = rightNode.Traverse_Tree(symbol, rightPath);
                }

                if (left != null)
                {
                    return left;
                }
                else
                {
                    return right;
                }
            }
        }
    }
}
```

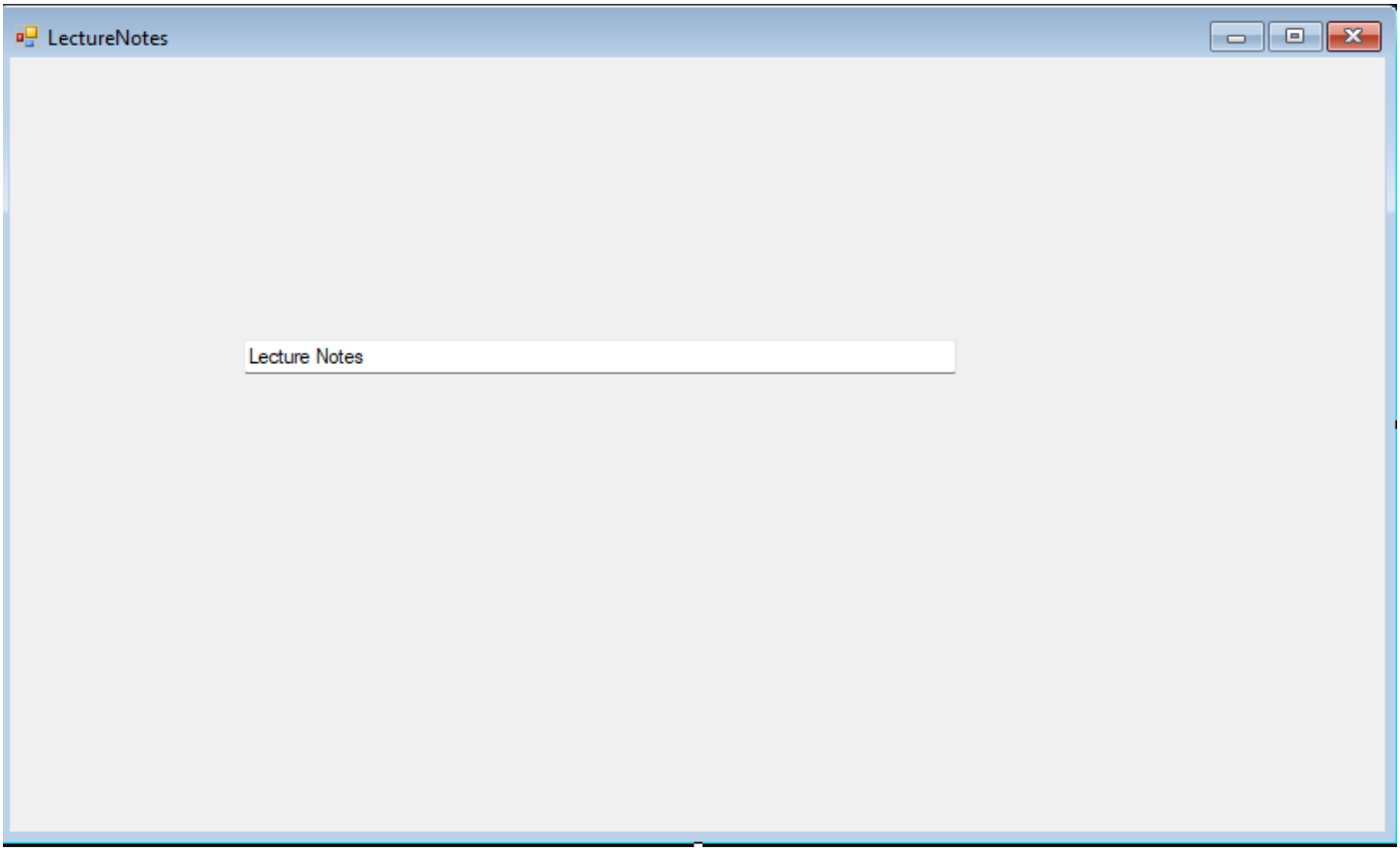
INTERFACES


X
Get Started
Enrollment
Category
Password
Show Password
LOGIN
CLEAR
Don't have an Account
Create Account
X
Get Started
Username
Category
Department
Password
Confirm Password
Show Password
REGISTER
CLEAR
Already have an Account


FormStudent
SFL
Student Faculty Link
Course Outline
Course Outline
Assignment
Lecture Notes
Logout


Assignment
Course:
Se No
FilesName
Action
Id
Faculty
Course
Class
File


CourseOutline
Course Outline


LectureNotes
Lecture Notes

CONCLUSION

SFL is software that gives students access to learning material in one place from any device and the ability to return to the content as needed. It allows you to consolidate all of the learning materials into one place so that students can conveniently access them. Also, you can assign these materials based on specific needs. For example, if you're assigning a group project, you may assign students a different learning material for each student based on their role in the group. For educational institutions, online training poses a cost-effective method of learning. With a learning management system, any organization can eliminate expenses such as classroom rentals, transport and accommodations, instructor's salary, textbooks, supplies, and physical printouts.