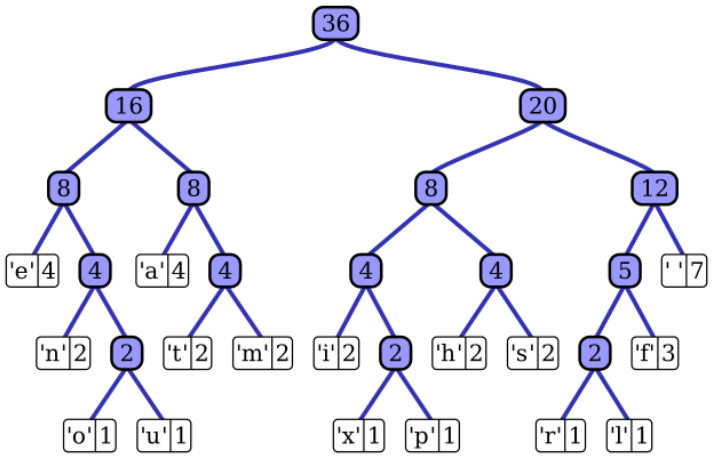
Huffman coding – Huffman's Algorithm

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-legth codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods - it is replaced with arithmetic coding or asymmetric numeral systems if better compression ratio is required.



Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Node.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HuffmanTest
{
    public class Node
    {
        public char Symbol { get; set; }
        public int Frequency { get; set; }
        public Node Right { get; set; }
        public Node Left { get; set; }

        public List<bool> Traverse(char symbol, List<bool> data)
        {
            // Leaf
            if (Right == null && Left == null)
            {
                if (symbol.Equals(this.Symbol))
                {
                    return data;
                }
                else
                {
                    return null;
                }
            }
            else
            {
                List<bool> left = null;
                List<bool> right = null;

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

                    left = Left.Traverse(symbol, leftPath);
                }

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

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

HuffmanTree.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace HuffmanTest
{
    public class HuffmanTree
    {
        private List<Node> nodes = new List<Node>();
        public Node Root { get; set; }
        public Dictionary<char, int> Frequencies = new
Dictionary<char, int>();

        public void Build(string source)
        {
            for (int i = 0; i < source.Length; i++)
            {
                if (!Frequencies.ContainsKey(source[i]))
                {
                    Frequencies.Add(source[i], 0);
                }

                Frequencies[source[i]]++;
            }

            foreach (KeyValuePair<char, int> symbol in Frequencies)
            {
                nodes.Add(new Node() { Symbol = symbol.Key, Frequency
= symbol.Value });
            }

            while (nodes.Count > 1)
            {
```

```
                List<Node> orderedNodes = nodes.OrderBy(node =>
node.Frequency).ToList<Node>();

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

                    // Create a parent node by combining the
frequencies
                    Node parent = new Node()
                    {
                        Symbol = '*',
                        Frequency = taken[0].Frequency +
taken[1].Frequency,
                        Left = taken[0],
                        Right = taken[1]
                    };

                    nodes.Remove(taken[0]);
                    nodes.Remove(taken[1]);
                    nodes.Add(parent);
                }

                this.Root = nodes.FirstOrDefault();

            }

        }

        public BitArray Encode(string source)
        {
            List<bool> encodedSource = new List<bool>();

            for (int i = 0; i < source.Length; i++)
            {
                List<bool> encodedSymbol =
this.Root.Traverse(source[i], new List<bool>());
                encodedSource.AddRange(encodedSymbol);
            }

            BitArray bits = new BitArray(encodedSource.ToArray());

            return bits;
        }

        public string Decode(BitArray bits)
        {
            Node current = this.Root;
            string decoded = "";
```

```
            foreach (bool bit in bits)
            {
                if (bit)
                {
                    if (current.Right != null)
                    {
                        current = current.Right;
                    }
                }
                else
                {
                    if (current.Left != null)
                    {
                        current = current.Left;
                    }
                }

                if (IsLeaf(current))
                {
                    decoded += current.Symbol;
                    current = this.Root;
                }
            }

            return decoded;
        }

        public bool IsLeaf(Node node)
        {
            return (node.Left == null && node.Right == null);
        }

    }
}
```

Program to test the algorithm

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace HuffmanTest
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Please enter the string:");
            string input = Console.ReadLine();
            HuffmanTree huffmanTree = new HuffmanTree();
```

```
            // Build the Huffman tree
            huffmanTree.Build(input);

            // Encode
            BitArray encoded = huffmanTree.Encode(input);

            Console.Write("Encoded: ");
            foreach (bool bit in encoded)
            {
                Console.Write((bit ? 1 : 0) + "");
            }
            Console.WriteLine();

            // Decode
            string decoded = huffmanTree.Decode(encoded);

            Console.WriteLine("Decoded: " + decoded);

            Console.ReadLine();
        }
    }
}
```