**Lab 5**

**Topic: Coding information using the Huffman algorithm**

**The purpose of the work** : to study the Huffman coding method.

**Huffman Method Algorithm**

's algorithm algorithm ) — an algorithm for optimal prefix coding of the alphabet.

One of the first algorithms for efficient information coding was proposed by Huffman in 1952. This algorithm became the basis for a large number of information compression programs. For example, Huffman coding is used in ARJ, ZIP, RAR compression programs, in the JPEG lossy graphic image compression algorithm, and is also built into modern fax machines.

Efficient Huffman coding consists of representing the most probable (frequently encountered) letters by binary codes of the shortest length, and the less probable ones by codes of a longer length (if all code words of a shorter length have already been exhausted). This is done in such a way that the average code length per letter of the original message is minimal.

Huffman coding is a simple algorithm for constructing variable-length codes that have a minimum average length. This very popular algorithm is the basis of many computer programs for compressing text and graphics data. Some of them use the Huffman algorithm directly, while others use it as one of the stages of a multi-level compression process. The Huffman method produces perfect compression (that is, compresses the data to its entropy) if the probabilities of the symbols are exactly equal to negative powers of 2. The algorithm starts building the code tree from the bottom up, then slides down the tree to build each individual code from right to left (from the least significant bit to the most significant).

The algorithm begins by compiling a list of alphabet symbols in descending order of their probabilities. Then a tree is constructed from the root, with these symbols as leaves. This is done in steps, with the two symbols with the lowest probabilities being selected at each step, added to the top of the partial tree, removed from the list, and replaced by an auxiliary symbol representing these two symbols. The auxiliary symbol is assigned a probability equal to the sum of the probabilities of the symbols selected at that step. When the list is reduced to one auxiliary symbol representing the entire alphabet, the tree is declared complete. The algorithm ends by descending the tree and constructing codes for all symbols.

To illustrate the Huffman algorithm, let's look at a graphical way of constructing a coding tree. A Huffman coding tree is a binary tree in which each node has a weight, and the weight of a parent is equal to the total weight of its children.

**Coding algorithm**

1. The letters of the input alphabet form a list of free nodes of the future coding tree. Each node in this list has a weight equal to the probability of the corresponding letter appearing in the message.
2. Two free tree nodes with the smallest weights are selected. If there are more than two free nodes with the smallest weights, then any pair can be taken.
3. Their parent is created with a weight equal to their combined weight.
4. The parent is added to the list of free nodes, and its two children are removed from this list.
5. One arc coming out of the parent node is assigned a bit 1, the other one is assigned a bit 0.
6. Steps 2, 3, 4, 5 are repeated until there is only one node left in the list of free nodes. This node will be the root of the tree. Its weight is equal to one -- the total probability of all the letters in the message.
7. Now, moving along the code tree from top to bottom and sequentially writing out the binary digits corresponding to the arcs, we can obtain the codes of the letters of the input alphabet.

**Example of coding using the Huffman algorithm**

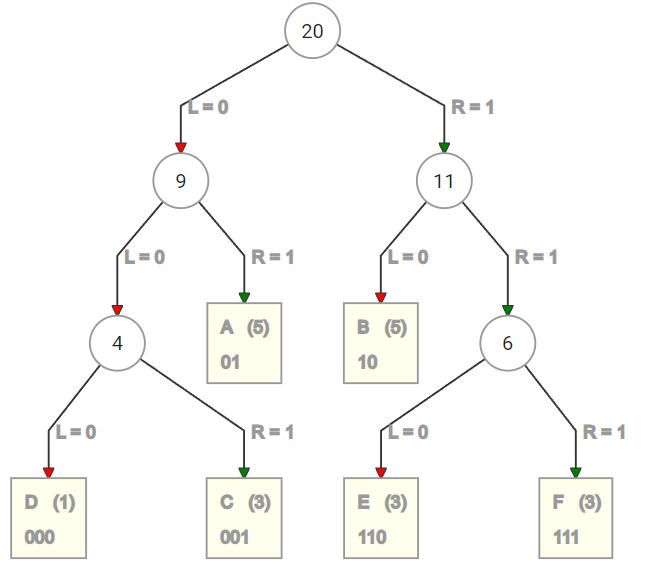
Let's say we have a string "AABBCCDEFABCAEFBABFE". Total - 20 characters. Here's how we can encode this string using the Huffman algorithm:

1. **Let's calculate the frequency of each character in the string:**

* A: 5 times
* B: 5 times
* C: 3 times
* D: 1 time
* E: 3 times
* F: 3 times

1. **Let's create a Huffman tree:**
   * Let's create a Huffman tree starting with the symbols with the lowest frequency. We'll merge the nodes with the lowest frequencies until we create a complete tree.

Approximate tree construction:



1. **We assign a unique code to each symbol**
2. **Use assigned codes to encode strings**

That is, the main point of the algorithm is to assign a shorter code to symbols that occur most frequently, and vice versa, symbols that occur less frequently have a longer code.

**Lab assignment:**

Write a program that implements information encoding using the Huffman method.

The program must accept text either from the input window on the main form or read from a file as input information. Calculate and output the compression ratio.

Example of tree construction (online):

https://huffman.ooz.ie

**Application**

**Example of implementation of the Huffman method**

**1. Main program**

using System;

using System.Collections.Generic ;

using System.Linq ;

using System.Text ;

using System.Collections ;

namespace HuffmanTest

{

**class program**

{

**static void Main(string[] args )**

{

string input = "String to encode ";

HuffmanTree huffmanTree = new HuffmanTree ();

// Building a Huffman tree

huffmanTree.Build (input);

// Coding

BitArray encoded = huffmanTree.Encode (input);

Console.Write ("Encoded: ");

foreach (bool bit in encoded)

{

Console.Write ((bit ? 1 : 0) + "");

}

Console.WriteLine ();

// Decoding

string decoded = huffmanTree.Decode (encoded);

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

Console.ReadLine ();

}

}

}

**2. Class HuffmanTree**

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)

{

List<Node> taken = orderedNodes.Take (2). ToList <Node>();

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);

}

}

}

**3. Class Node**

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)**

{

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 ;

}

}

}

}

}