|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| Министерство науки и высшего образования Российской Федерации | | | | | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  «МИРЭА – Российский технологический университет»  РТУ МИРЭА | | | | | | |
| **Институт** | | ИКБСП | | | | |
|  | | | | |  | |
| **Специальность (направление):** | | | | Информационные системы и технологии | | |
|  | | | | | |  |
| **Кафедра:** | | КБ4 «Интеллектуальные системы информационной безопасности» | | | | |
|  | | | | | |  |
| **Дисциплина:** | | | «Технологии программирования» | | | |

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе на тему

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Разработка приложения сжатия данных методом Хаффмана | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| Студент: | |  | |  |  | | |  | | Быховер. И.С. | |
|  | | | *подпись* |  | *дата* | | |  | | *инициалы и фамилия* | |
| Группа: | БСБО-02-19 | | | | | |  | | Шифр: | | 09.03.02 |
| Работа защищена на оценку: | | | | | |  | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Руководитель работы: |  |  |  |  | к.т.н., доц. Ильиченкова З.В. |
|  | *подпись* |  | *дата* |  | *инициалы и фамилия* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Члены комиссии: |  |  |  |  |  |
|  | *подпись* |  | *дата* |  | *инициалы и фамилия* |
|  |  |  |  |  |  |
|  | *подпись* |  | *дата* |  | *инициалы и фамилия* |

**Москва 2020 г.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| Министерство науки и высшего образования Российской Федерации | | | | | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  «МИРЭА – Российский технологический университет»  РТУ МИРЭА | | | | | | |
| **Институт** | | ИКБСП | | | | |
|  | | | | |  | |
| **Специальность (направление):** | | | | Информационные системы и технологии | | |
|  | | | | | |  |
| **Кафедра:** | | КБ4 «Интеллектуальные системы информационной безопасности» | | | | |
|  | | | | | |  |
| **Дисциплина:** | | | «Технологии программирования» | | | |

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студент: | | | Быховер Илья Сергеевич | | | | | | | |
| Группа: | | БСБО-02-19 | | | |  | Шифр: | | 09.03.02 | |
| 1)Тема: | | | | Разработка приложения сжатия данных методом Хаффмана | | | | | | |
|  | | | |  | | | | | | |
| 2)Срок предоставления работы к защите: | | | | | | | | 02 июня 2020г. | | |
| 3)Исходные данные для разработки: | | | | |  | | | | | |
| 4)Содержание пояснительной записки: | | | | | | | | | |  | |
|  | -Титульный лист | | | | | | | | | |
|  | -Содержание | | | | | | | | | |
|  | -Введение | | | | | | | | | |
|  | -Глава … | | | | | | | | | |
|  | -Глава … | | | | | | | | | |
|  | -Заключение | | | | | | | | | |
|  | -Список литературы | | | | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Руководитель работы: |  | |  |  | | |  | к.т.н., доц. Кашкин Е.В. | | |
|  | *подпись* | |  | *дата* | | |  | *инициалы и фамилия* | | |
| Задание принял к исполнению: | |  | | |  |  | | |  | Быховер И.С. |
|  | | *подпись* | | |  | *дата* | | |  | *инициалы и фамилия* |

**Москва 2020 г.**

Оглавление

[Введение 4](#_Toc41674378)

[Глава 1. Методы реализации программ сжатия данных 6](#_Toc41674379)

[Глава 2. Предложения по реализации программы сжатия данных 7](#_Toc41674380)

[Глава 3. Описание и реализация программы сжатия данных 11](#_Toc41674381)

[Заключение 25](#_Toc41674382)

[Список литературы 26](#_Toc41674383)

# Введение

На сегодняшний день часто требуется не только шифровать данные, но и сделать так, чтобы их можно было хранить в меньшем объеме и передавать как можно быстрее и эффективнее. Сжатие данных – это алгоритмическое преобразование, производимое с целью уменьшения занимаемого ими объема. Все методы сжатия делятся на два основных класса: без потерь (обратимое сжатие) и с потерями (необратимое сжатие).

Метод сжатия без потерь не совсем эффективный, так как часть данных несущественно теряется или изменяется. В основном этот метод используют для сжатия видео-, аудио-данных, а также фотографий. Сжатие без потерь обычно применяют для передачи объемных данных или в случаях, когда нам нужно восстановить данные без каких-либо искажений. Данный тип сжатия используется во всех файловых архиваторах.

С высоким ростом развития вычислительной техники, по-прежнему остается проблема скорости передачи данных по каналам связи. Существующей пропускной способности не хватает, чтобы передавать без потерь данные больших объемов и за малый промежуток времени. Поэтому разработка новых и эффективных методом сжатия данных является актуальной задачей в наше время.

В соответствии с вышеизложенным, цель работы: разработать программу для сжатия данных методом Хаффмана с использованием графического интерфейса.

Для того, чтобы достичь данной цели, необходимо решить следующие задачи:

1. Изучить алгоритм Хаффмана
2. Создать графический интерфейс с помощью платформы .NET
3. Реализовать алгоритм Хаффмана на языке программирования C# для кодирования текстовой информации

# Глава 1. Методы реализации программ сжатия данных

Общий принцип всех алгоритмов сжатия заключается в поиске избыточной информации в данных и последующем ее кодированием с целью получения более компактного выходного потока данных. Любой алгоритм сжатия включает в себя два преобразования обратных друг другу – это сжатие и восстановление данных. Таким образом, при разработке программ сжатия данных, не зависимо от алгоритма сжатия, появляется необходимость описания процедуры кодирования, а также декодирования для восстановления исходных данных.

# Глава 2. Предложения по реализации программы сжатия данных

Для реализации нашей программы, мы будем использовать алгоритм Хаффмана. Это один из первых эффективных методов сжатия информации, который был впервые разработан в 1952 году. Данный алгоритм до сих пор используется во многих программах сжатия данных.

Сам принцип алгоритма состоит в том, что мы кодируем часто встречающейся символы в сообщении кодом наименьшей длины, а менее частые символы – кодами большей длины. Это позволяет нам сильнее сжать данные.

Я выделил следующие основные предложения по реализации своей программы: с помощью языка программирования C# разработаем

1. Графический интерфейс
2. Методы кодирования и декодирования данных

Ниже опишем алгоритм каждого метода.

**Алгоритм кодирования:**

Возьмем для примера слово effervescence. Для начала, нужно составить таблицу частот символов в данном сообщении:

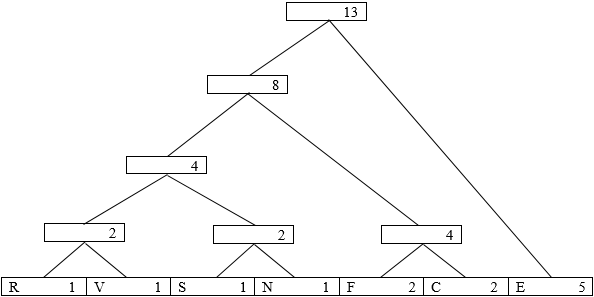


Сортируем словарь по частоте символов, получаем:



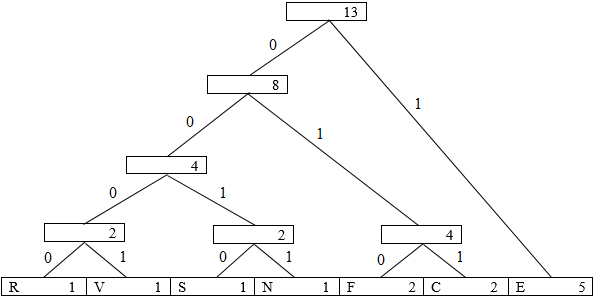
По полученной таблице нужно построить бинарное дерево (дерево Хаффмана). Бинарное дерево – это дерево, в котором каждый узел имеет не более двух потомков. Обычно, первого называют родительским узлом, а потомков называют левым и правым наследником. Итак, для того, чтобы построить дерево Хаффмана нужно взять первые два минимальных узла и от них создать новый узел, который будет родительским по отношению к этим двум узлам. Также в новом узле будет храниться сумма частот первых двух узлов.

Построение дерева Хаффмана:

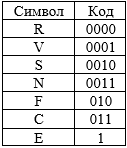


После того, как мы построили дерево, нам нужно пройтись по нему и записать кодировку каждого символа. Для этого обычно используется рекурсивная функция. Начиная с вершины дерева, мы записываем в кодировку 0, каждый раз, когда мы идем налево, и 1, когда мы идем направо. Конечный ряд и будет кодом Хаффмана для выбранного нами символа. Далее будем проходиться по дереву Хаффмана и записывать коды символов в таблицу.

Кодовое дерево Хаффмана для слова effervescence:



Запишем в словарь все получившиеся кодировки символов:



Каждая из получившихся кодировок обладает своей уникальностью (то есть ни одно кодовое слово не является началом другого), что позволяет однозначно их декодировать.

Следует обратить внимание, для того, чтобы закодировать строки, необходимо дважды пройтись по ним, поэтому данный метод называется двухпроходный. Первый раз мы проходим по строке для того, чтобы узнать частоту символов и построить по ним дерево. Затем мы проходимся по второму разу и зашифровываем каждый символ кодом Хаффмана. В результате мы получим сжатые данные в виде закодированной строки.

В случае нашего примера, получим:

101001010000000110010011100110111

**Алгоритм декодирования:**

Продемонстрируем данный метод на нашем примере. У нас есть закодированная строка: 101001010000000110010011100110111.

Чтобы восстановить данные, нам понадобится дерево Хаффмана, которое использовали для кодирования. Мы начинаем побитно считывать закодированную строку и одновременно проходиться по узлам дерева, начиная с вершины. Например, мы считываем первый бит, равный 1, который приведет нас к букве E. Затем мы возвращаемся к вершине дерева. Считываем следующий бит, равный 0, который ведет нас налево, потом к биту 1, затем к биту 0, что приводит нас к букве F. Мы возвращаемся к вершине дерева и повторяем процесс до тех пор, пока не восстановим полностью закодированную строку.

В итоге получаем исходное слово:

101001010000000110010011100110111 => effervescence

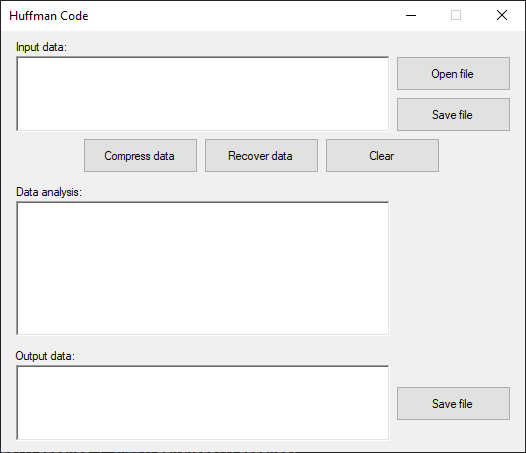
Для создания графических интерфейсов с помощью платформы .NET применяются разные технологии – Windows Forms, WPF и так далее. Для разработки будем использовать Windows Forms.

Реализация программы с использованием данного алгоритма будет показана в третьей главе.

# Глава 3. Описание и реализация программы сжатия данных

Данная программа реализована на языке программирования C# с использованием графического интерфейса.

С помощью средств Windows Forms разместим на форме все необходимые элементы управления программы:



В первое текстовое поле **inputTextBox**, чтобы добавить данные, пользователь должен либо ввести их с клавиатуры, либо, нажав на кнопку **Open file**, открыть файл. Ниже будет представлен метод **openButton\_Click** открытия файла и добавления данных оттуда в текстовое поле **inputTextBox**.

private void openButton\_Click(object sender, EventArgs e)

{

OpenFileDialog openFile = new OpenFileDialog();

openFile.Filter = "Text Document (\*.txt)|\*.txt|BIN File (\*.bin)|\*.bin|All files (\*.\*)|\*.\*";

if (openFile.ShowDialog().Equals(DialogResult.OK))

{

inputTextBox.Clear();

outputTextBox.Clear();

using (FileStream fstream = File.Open(openFile.FileName, FileMode.Open))

{

if (openFile.FileName.Contains(".bin"))

{

try

{

if (analysisTextBox.TextLength.Equals(0))

throw new Exception("Cannot perform this operation!");

compressButton.Enabled = false;

recoverButton.Enabled = true;

if (!check)

countOfBits = outputBits.Count();

inputBits.Clear();

byte[] data = new byte[fstream.Length];

fstream.Read(data, 0, data.Length);

for (int i = 0; i < data.Length; i++)

{

byte b = data[i];

for (int j = 0; j < 8; j++)

{

--countOfBits;

if (!(b & (1 << j)).Equals(0))

inputBits.Add(1);

else inputBits.Add(0);

if (countOfBits.Equals(0))

break;

}

}

inputBits.Reverse();

foreach (var bit in inputBits)

inputTextBox.AppendText(bit.ToString());

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

else

{

compressButton.Enabled = true;

recoverButton.Enabled = false;

byte[] data = new byte[fstream.Length];

fstream.Read(data, 0, data.Length);

inputTextBox.Text = System.Text.Encoding.Default.GetString(data);

}

}

}

}

Данный метод описывает событие нажатия на кнопку **Open file** (**openButton**). В зависимости от выбранного файла будет выполнена определенная часть метода. Если пользователь открывает текстовый документ с целью сжатия, то программа просто считывает побайтно каждый символ в файле и затем добавляет в текстовое поле. В случае, если пользователь открыл бинарный файл, программа побайтно добавит все данные в массив **data**. Далее, перебирая каждый байт в массиве, программа с помощью побитовых операций побитно разбирает каждый байт и проверяет, если бит равен нулю, то в список **inputBits** добавляется ноль, иначе единица. После этого, в текстовое поле **inputTextBox** программа выводит получившийся список. Также в данном методе учитываются ошибки пользователя, если пользователь попытается открыть файл с целью восстановления с учетом того, что второе текстовое поле **analysisTextBox** пустое, то программа выдаст ошибку о том, что нельзя выполнить данное действие.

Перед тем как приступить к кодированию данных, нам понадобятся некоторые структуры данных для хранения информации:

List<int> outputBits = new List<int>(); – для хранения закодированной строки, с целью вывода в третье текстовое поле **outputTextBox**, и в дальнейшем для сохранения в бинарный файл

List<int> inputBits = new List<int>(); – для хранения закодированной строки из бинарного файла

Dictionary<string, int> symbols = new Dictionary<string, int>(); – для хранения символов и их количества из текстового поля **inputTextBox**

Dictionary<string, List<int>> codeTable = new Dictionary<string, List<int>>(); – для хранения символов и их кодировок

После того, как пользователь ввел данные, при нажатии на кнопку **Compress data** происходит сам процесс кодирования введенной строки.

private void compressButton\_Click(object sender, EventArgs e)

{

symbols.Clear();

codeTable.Clear();

try

{

if (inputTextBox.TextLength.Equals(0))

throw new Exception("Cannot perform this operation!");

analysisTextBox.Clear();

outputTextBox.Clear();

symbolsFrequency();

createTree();

showAnalysis();

compressionData();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

В данном методе вызываются сразу четыре метода. Разберем по отдельности каждый из них.

Рассмотрим метод **symbolsFrequency**:

public void symbolsFrequency()

{

for (int i = 0; i < inputTextBox.TextLength; i++)

{

string symbol = inputTextBox.Text[i].ToString();

if (symbol.Equals("\n")) symbol = "Newline";

if (symbol.Equals(" ")) symbol = "Space";

if (!symbols.ContainsKey(symbol))

symbols.Add(symbol, 1);

else symbols[symbol]++;

}

}

Данный метод берет из строки символ и проверяет, если символ не содержится в словаре **symbols**, то добавляет его, иначе увеличивает на единицу счетчик, показывающий, сколько раз встречался данный символ в строке. Это пригодится для построения бинарного дерева.

С помощью следующего метода **createTree**, построим бинарное дерево Хаффмана:

public void createTree()

{

List<Node> nodes = new List<Node>();

foreach (var item in symbols.OrderBy(item => item.Value))

{

Node node = new Node();

node.symbol = item.Key;

node.frequence = item.Value;

nodes.Add(node);

}

while (!nodes.Count().Equals(1))

{

nodes = new List<Node>(nodes.OrderBy(item => item.frequence));

Node leftNode = new Node(); leftNode = nodes.First(); nodes.Remove(nodes.First());

Node rightNode = new Node(); rightNode = nodes.First(); nodes.Remove(nodes.First());

Node sourceNode = new Node(leftNode, rightNode);

nodes.Add(sourceNode);

}

Node rootNode = new Node();

rootNode = nodes.First();

code.Clear();

createCodeTable(rootNode);

}

Перед тем, как строить дерево, опишем класс **Node** для дальнейшей работы:

class Node

{

public string symbol;

public int frequence;

public Node leftNode;

public Node rightNode;

public Node() { }

public Node(Node leftNode, Node rightNode)

{

this.leftNode = leftNode;

this.rightNode = rightNode;

this.frequence = this.leftNode.frequence + this.rightNode.frequence;

}

}

Класс **Node** содержит два основных поля: символ и частота. И так как каждый узел в бинарном дереве может иметь только два узла–наследника, то нам нужны еще две переменные типа **Node** – это левый и правый потомок.

В методе **createTree** создается список **nodes** типа **Node**, для хранения узлов. Затем создаем узел **node**, передаем ему значение символа и частоты и кладем в этот список. Заполнив весь список узлами, мы начинаем выстраивать дерево. Для этого, мы берем два узла с минимальными частотами, создаем от них родительский узел **sourceNode** и передаем в новый узел сумму частот этих двух узлов. После этого, удаляем эти два узла, а новый узел кладем в список. Для того, чтобы определить, какие узлы в списке имеют минимальные частоты, нужно каждый раз сортировать этот список по возрастанию. Так получается, что первые два узла имеют минимальные частоты. Повторяем это все до тех пор, пока в списке не останется один узел. Когда в списке останется один узел, мы создаем переменную **rootNode** класса **Node** для хранения корневого узла и положим в него оставшийся узел в списке.

В итоге, мы получим бинарное дерево, в котором каждый узел указывает на своего левого и правого наследника.

Далее опишем рекурсивный метод **createCodeTable** для прохода по дереву, вызываемый из метода **createTree**:

private void createCodeTable(Node node)

{

if (node.leftNode != null)

{

code.Add(0);

createCodeTable(node.leftNode);

}

if (node.rightNode != null)

{

code.Add(1);

createCodeTable(node.rightNode);

}

if (node.symbol != null)

codeTable.Add(node.symbol, code.ToList());

if (!code.Count.Equals(0))

{

code.Reverse();

code.Remove(code.First());

code.Reverse();

}

}

Для начала объявим временный список **code** для хранения кодировки одного символа. Начиная проходить дерево с корневого узла, проверяем, если у узла есть левый потомок (**leftNode**), то добавляем в список **code** ноль, если у узла есть правый потомок (**rightNode**), то добавляем единицу. И так проходимся по ветвям до тех пор, пока у узла не окажется ни левого, ни правого потомка. Если такое случилось, то это значит, что мы дошли до узла (листа дерева), который содержит в себе символ. Имея символ и его код, сохраним их в словарь **codeTable**.

Получив все кодировки символов, мы можем зашифровать каждый символ получившимся кодом:

private void compressionData()

{

outputBits = new List<int>();

string symbol;

for (int i = 0; i < inputTextBox.TextLength; i++)

{

if (inputTextBox.Text[i].ToString().Equals(" "))

symbol = "Space";

else if (inputTextBox.Text[i].ToString().Equals("\n"))

symbol = "Newline";

else symbol = inputTextBox.Text[i].ToString();

if (codeTable.ContainsKey(symbol))

{

foreach (var item in codeTable)

{

if (item.Key.Equals(symbol))

{

List<int> code = new List<int>(item.Value);

foreach (var element in code)

outputBits.Add(element);

}

}

}

}

foreach (var bit in outputBits)

outputTextBox.AppendText(bit.ToString());

}

Используя метод **compressionData**, будем посимвольно проверять текстовое поле **inputTextBox**, если словарь **codeTable** содержит кодировку символа, то в список **outputBits** добавляем кодировку данного символа. Затем получившийся список выводим в текстовом поле **outputTextBox**.

Далее закодированную строку можно сохранить в файл:

private void saveOutputButton\_Click(object sender, EventArgs e)

{

SaveFileDialog saveFile = new SaveFileDialog();

saveFile.Filter = "BIN File (\*.bin)|\*.bin|Text Document (\*.txt)|\*.txt|All files (\*.\*)|\*.\*";

if (saveFile.ShowDialog().Equals(DialogResult.OK))

{

using (FileStream fstream = File.Open(saveFile.FileName, FileMode.Create))

{

if (saveFile.FileName.Contains(".bin"))

{

check = false;

outputBits.Reverse();

byte result;

int count;

for (int i = 0; i < outputBits.Count(); i++)

{

count = 0; result = 0;

for (byte b = 1; i < outputBits.Count(); b <<= 1, i++)

{

++count;

if (outputBits[i].Equals(1))

result |= b;

if (count.Equals(8))

break;

}

fstream.WriteByte(result);

}

outputBits.Reverse();

}

else

{

byte[] data = new byte[fstream.Length];

data = System.Text.Encoding.Default.GetBytes(outputTextBox.Text);

fstream.Write(data, 0, data.Length);

}

}

}

}

Данный метод разбивает закодированную строку на блоки по восемь символов и получившиеся байты записывает в файл.

Для того, чтобы восстановить закодированную строку используем метод **recoveryData**, вызываемый в методе **recoverButton\_Click** по нажатию кнопки **Recover data**:

private void recoveryData()

{

inputBits.Clear();

for (int i = 0; i < inputTextBox.TextLength; i++)

{

if (inputTextBox.Text[i].ToString().Equals("1"))

inputBits.Add(1);

else if (inputTextBox.Text[i].ToString().Equals("0")) inputBits.Add(0);

}

int index = 0;

List<int> checkCode = new List<int>();

for (; index < inputBits.Count(); index++)

{

checkCode.Add(inputBits[index]);

foreach (var item in codeTable)

{

List<int> code = new List<int>(item.Value);

if (checkCode.SequenceEqual(code))

{

if (item.Key.Equals("Space"))

outputTextBox.AppendText(" ");

else if (item.Key.Equals("Newline"))

outputTextBox.AppendText("\n");

else outputTextBox.AppendText(item.Key);

checkCode.Clear();

}

}

}

}

Закодированную строку вводим с клавиатуры или получаем из файла в текстовое поле **inputTextBox** и записываем ее содержимое в список **inputBits**. Затем создаем временный список **checkCode**. Посимвольно заполняем список **checkCode** элементами из списка **inputBits** и сравниваем его с кодами символов из словаря **codeTable**. Если коды совпали, то в текстовое поле **outputTextBox** записываем символ, которому принадлежит данная кодировка, а список **checkCode** очищаем. Повторяем это до тех пор, пока не расшифруем полностью сообщение.

Также для удобства выведем в текстовом поле **analysisTextBox** анализ данных:

private void showAnalysis()

{

symbols = symbols.OrderBy(item => item.Key).ToDictionary(item => item.Key, item => item.Value);

codeTable = codeTable.OrderBy(item => item.Key).ToDictionary(item => item.Key, item => item.Value);

analysisTextBox.AppendText("Symbol\t\tFrequency\tCode\n");

string[] codes = new string[codeTable.Count()];

int index = 0;

foreach (var item in codeTable)

{

List<int> code = new List<int>(item.Value);

foreach (var element in code)

codes[index] += element.ToString();

++index;

}

index = 0;

foreach (var item in symbols)

analysisTextBox.AppendText($"{item.Key}\t\t{item.Value}\t\t{codes[index++]}\n");

}

# Заключение

При выполнении курсовой работы я узнал, для чего нужна компрессия данных, а также убедился в том, что в современном мире никак не обойтись без алгоритмов сжатия.

В соответствии с этим в курсовой работе были решены следующие задачи:

1. Изучена информация о том, что такое сжатие данных, какие существуют алгоритмы, а также как программно реализовать алгоритм сжатия.
2. Предложены решения для реализации программного продукта.
3. Создан программный продукт для сжатия данных методом Хаффмана с использованием графического интерфейса. Также на примере текста “Big cities of the United Kingdom” провели сравнение исходного и сжатого файлов и убедились в работоспособности программы.

Программный продукт может быть использован, например, в различных веб–приложениях для более быстрой и эффективной передачи объемных данных. В будущем в программе следует добавить возможность сжимать файлы других форматов и доработать сохранение дерева Хаффмана как часть сжатого файла.

# Список литературы

1. Алгоритмы для начинающих. Теория и практика для разработчика. Луридас Панос. Изд-во Бомбора. 2018. С.86-107.
2. OpenFileDialog и SaveFileDialog [Электронный ресурс] – Режим доступа: <https://metanit.com/sharp/windowsforms/4.20.php> (Дата обращения 21.05.2020)
3. System Linq Пространство имен [Электронный ресурс] – Режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/api/system.linq?view=netcore-3.1> (Дата обращения 24.04.2020)
4. Алгоритм Хаффмана. Построение кодового дерева Хаффмана [Электронный ресурс] – Режим доступа: <https://studme.org/187066/informatika/algoritm_haffmana> (Дата обращения 17.04.2020)
5. Введение в Windows Forms. Создание графического приложения [Электронный ресурс] – Режим доступа: <https://metanit.com/sharp/windowsforms/1.1.php> (Дата обращения 19.04.2020)
6. Код Хаффмана [Электронный ресурс] – Режим доступа: <https://en.wikipedia.org/wiki/Huffman_coding> (Дата обращения 17.04.2020)
7. Коллекции C# [Электронный ресурс] – Режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/collections> (Дата обращения 24.04.2020)
8. Поразрядные операции [Электронный ресурс] – Режим доступа: <https://metanit.com/sharp/tutorial/2.22.php> (Дата обращения 27.04.2020)
9. Сжатие данных [Электронный ресурс] – Режим доступа: <https://en.wikipedia.org/wiki/Data_compression> (Дата обращения 24.04.2020).
10. Сжатие данных в примерах. Кодирование по методу Хаффмана [Электронный ресурс] – Режим доступа: <https://www.ibm.com/developerworks/ru/library/l-compr/index.html> (Дата обращения 18.04.2020).