**СОДЕРЖАНИЕ**

1 ПОСТАНОВКА ЗАДАЧИ…………………………………………… 5

1.1 Обзор методов и алгоритмов решения поставленной задачи.. 5

1.2 Обоснование выбранных методов и алгоритмов…………….. 7

2 ВЫБОР СРЕДЫ РАЗРАБОТКИ…………………………………….. 8

3 СТРУКТУРА ВХОДНЫХ И ВЫХОДНЫХ ДАННЫХ……………. 9

4 ДИАГРАММА КЛАССОВ…………………………………………... 105 ОПИСАНИЕ КЛАССОВ…………………………………………….. 11

6 БЛОК-СХЕМА АЛГОРИТМА……………………………………… 14 6.1 Метод TreeClass createTree(Dictionary<byte, int> occurTable) 14

7 АЛГОРИТМЫ ПО ШАГАМ………………………………………... 15

7.1 Метод void createByteCodeTable(treeNode root, string startStr) 15

8 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ……………………………. 16

ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПРОГРАММЫ……………………... 21

**ВВЕДЕНИЕ**

C# — элегантный, типобезопасный объектно-ориентированный язык, предназначенный для разработки разнообразных безопасных и мощных приложений, выполняемых в среде .NET Framework. С помощью языка C# можно создавать обычные приложения Windows, XML-веб-службы, распределенные компоненты, приложения "клиент-сервер", приложения баз данных и т. д. Visual C# предоставляет развитый редактор кода, конструкторы с удобным пользовательским интерфейсом, встроенный отладчик и множество других средств, упрощающих разработку приложений на базе языка C# и .NET Framework.

Синтаксис C# очень выразителен, но прост в изучении. Все, кто знаком с языками C, C++ или Java с легкостью узнают синтаксис с фигурными скобками, характерный для языка C#. Разработчики, знающие любой из этих языков, как правило, смогут добиться эффективной работы с языком C# за очень короткое время. Синтаксис C# делает проще то, что было сложно в C++, и обеспечивает мощные возможности, такие как типы значений Nullable, перечисления, делегаты, лямбда-выражения и прямой доступ к памяти, чего нет в Java.

C# поддерживает универсальные методы и типы, обеспечивая более высокий уровень безопасности и производительности, а также итераторы, позволяющие при реализации коллекций классов определять собственное поведение итерации, которое может легко использоваться в клиентском коде. Выражения LINQ делают строго типизированный запрос очень удобной языковой конструкцией.

Как объектно-ориентированный язык, C# поддерживает понятия инкапсуляции, наследования и полиморфизма. Все переменные и методы, включая метод Main – точку входа приложения – инкапсулируются в определения классов. Класс может наследовать непосредственно из одного родительного класса, но может реализовывать любое число интерфейсов. Для методов, которые переопределяют виртуальные методы в родительском классе, необходимо ключевое слово override, чтобы исключить случайное повторное определение. В языке C# структура похожа на облегченный класс: это тип, распределяемый в стеке, реализующий интерфейсы, но не поддерживающий наследование.

Программа на языке C# выполняется в среде .NET Framework – интегрированном компоненте Windows, содержащем виртуальную систему выполнения (среда CLR) и унифицированный набор библиотек классов.

**1 ПОСТАНОВКА ЗАДАЧИ**

Разработать классический вариант архиватора файлов для операционных систем семейства Windows, используя собственный алгоритм сжатия и разжатия.

К реализации предъявляются следующие требования:

1. Удобный интерфейс.
2. Возможность из программы выбирать файл для работы.
3. Процент сжатия не менее 15%
   1. **Обзор методов и алгоритмов решения поставленной задачи**

Для реализации приложения используется алгоритм Хаффмана – адаптивный жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при написании им курсовой работы. В настоящее время используется во многих программах сжатия данных.

Идея алгоритма состоит в следующем: зная вероятности символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана.

Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.

Выбираются два свободных узла дерева с наименьшими весами.

Создается их родитель с весом, равным их суммарному весу.

Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.

Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0.

Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Допустим, у нас есть следующая таблица частот:

15 7 6 6 5

А Б В Г Д

Этот процесс можно представить как построение дерева, корень которого — символ с суммой вероятностей объединенных символов, получившийся при объединении символов из последнего шага, его n0 потомков — символы из предыдущего шага и т. д.

Чтобы определить код для каждого из символов, входящих в сообщение, мы должны пройти путь от листа дерева, соответствующего текущему символу, до его корня, накапливая биты при перемещении по ветвям дерева (первая ветвь в пути соответствует младшему биту). Полученная таким образом последовательность битов является кодом данного символа, записанным в обратном порядке.

Для данной таблицы символов коды Хаффмана будут выглядеть следующим образом:

А Б В Г Д

0 100 101 110 111

Поскольку ни один из полученных кодов не является префиксом другого, они могут быть однозначно декодированы при чтении их из потока. Кроме того, наиболее частый символ сообщения А закодирован наименьшим количеством бит, а наиболее редкий символ Д — наибольшим.

Классический алгоритм Хаффмана имеет ряд существенных недостатков. Во-первых, для восстановления содержимого сжатого сообщения декодер должен знать таблицу частот, которой пользовался кодер. Следовательно, длина сжатого сообщения увеличивается на длину таблицы частот, которая должна посылаться впереди данных, что может свести на нет все усилия по сжатию сообщения. Кроме того, необходимость наличия полной частотной статистики перед началом собственно кодирования требует двух проходов по сообщению: одного для построения модели сообщения (таблицы частот и Н-дерева), другого для собственно кодирования. Во-вторых, избыточность кодирования обращается в ноль лишь в тех случаях, когда вероятности кодируемых символов являются обратными степенями числа 2. В-третьих, для источника с энтропией, не превышающей 1 (например, для двоичного источника), непосредственное применение кода Хаффмана бессмысленно.

* 1. **Обоснование выбранных методов и алгоритмов**

С тех пор, как Д. А. Хаффман опубликовал в 1952 году свою работу "Метод построения кодов с минимальной избыточностью", его алгоритм кодирования стал базой для огромного количества дальнейших исследований в этой области.

По сей день в компьютерных журналах можно найти большое количество публикаций, посвященных как различным реализациям алгоритма Хаффмана, так и поискам его лучшего применения. Кодирование Хаффмана используется в коммерческих программах сжатия (например в PKZIP и LHA), встроено в некоторые телефаксы и даже используется в алгоритме JPEG сжатия графических изображений с потерями.

Алгоритм обладает рядом преимуществ перед другими:

1. Прост – легко запоминается и сравнительно легко программируется.
2. Подходит для всех схем сжатия(статической, полуадаптивной и адаптивной)
3. Быстро работает
4. Генерирует таблицу кодов элементов

**2 ВЫБОР СРЕДЫ РАЗРАБОТКИ**

Для реализации приложения был выбран язык разработки – C#.

Сами разработчики языка описывают его, как простой, современный, объектно-ориентированный и безопасный язык программирования. Синтаксически C# напоминает C++ и Java, что позволило программистам за достаточно короткое время изучить тонкости нового языка.

Несмотря на то, что C# и .NET предназначены в первую очередь для веб-разработки, их также активно применяют для создания приложений, которые должны устанавливаться на машине конечного пользователя, где и будет выполняться вся обработка данных. Разработку таких приложений обеспечивает библиотека Windows Forms, позволяющая проектировать графический интерфейс.

Приложение, описанное в данной работе, разработано именно с помощью библиотеки Windows Forms.

Язык был выбран по следующим причинам:

1. Претендует на подлинную объектную ориентированность.
2. Призван реализовать компонентно-ориентированный подход к программированию, который способствует меньшей машинно-архитектурной зависимости результирующего программного кода, большей гибкости, переносимости и легкости повторного использования программ.
3. Отличием от предшественников является изначальная ориентация на безопасность кода.
4. Расширенная поддержка событийно-ориентированного программирования.
5. Язык является «родным» для создания приложений в среде Microsoft .NET, поскольку наиболее тесно и эффективно интегрирован с ней.

**3 СТРУКТУРА ВХОДНЫХ**  **И ВЫХОДНЫХ ДАННЫХ**

Входные данные:

openFileD - экземпляр класса OpenFileDialog, содержит данные о файле;

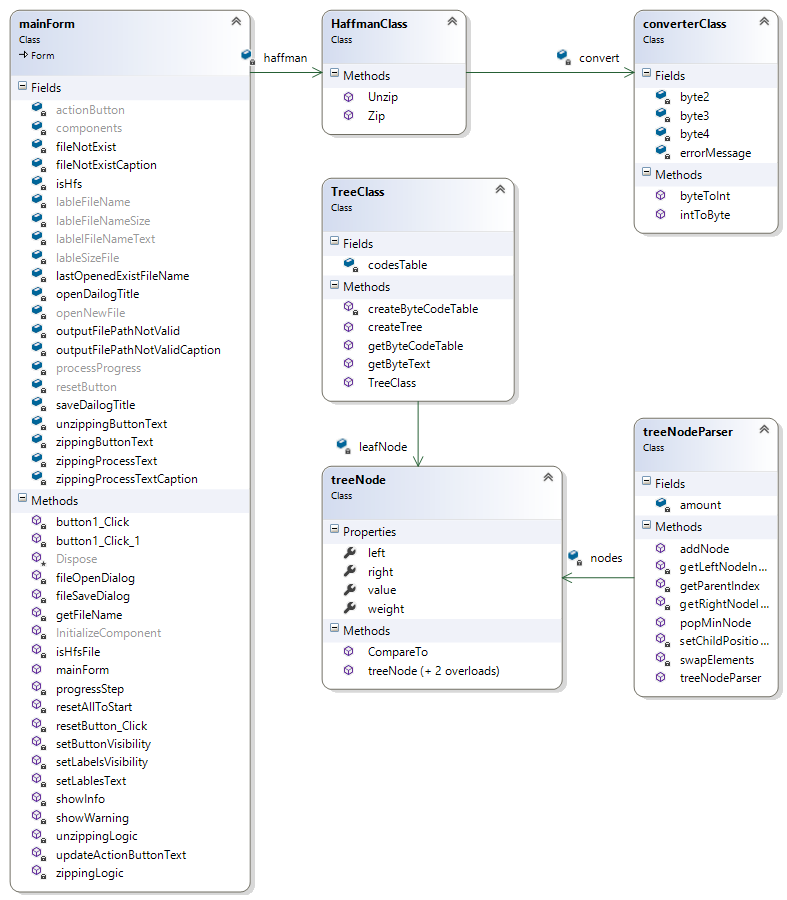
zip – метод класса HaffmanClass, производит сжатие файла и сохранение результатов в выходной файл ;

unzip - метод класса HaffmanClass, производит разжатие файла и сохранение результатов в выходной файл;

Выходные данные:

saveFileD – - экземпляр класса SaveFileDialog, содержит данные о файле преобразованном методами класса HaffmanClass.

**4 ДИАГРАММА КЛАССОВ**

****

**5 ОПИСАНИЕ КЛАССОВ**

|  |  |
| --- | --- |
| Класс для выполнения функций сжатия/разжатия | HaffmanClass |
| Класс для конвертации byte в int и обратно | ConverterClass |
| Класс для работы с деревом | TreeClass |
| Класс представляющий собой один элемент дерева | treeNode |
| Класс для работы с элементами дерева | treeNodeParser |
| Главный класс для работы с приложением | MainForm |

HaffmanClass – класс для выполнения функций сжатия/разжатия:

class HaffmanClass

{

private:

converterClass convert; // class for converting byte to in and back

public:

void Zip(string readFile, string writeFile);

//

// Method for zipping file

//

bool Unzip( string readFile, string writeFile ) = 0;

//

// Method for unzipping file

//

};

ConverterClass – класс для конвертации byte в int и обратно

class ConverterClass

{

private:

int byte2 = 256; //values for converting

int byte3 = 65536;

int byte4 = 16777216;

public:

int byteToInt( byte[] bytes);

//

// Method for converting byte to int

//

byte[] intToByte( int number);

//

// Method for converting int to byte

//

};

treeNode – класс представляющий собой один элемент дерева

class treeNode

{

public:

byte? value //value of node

int weight //weight of node

treeNode right //link to left node

treeNode left //link to right node

treeNode() //defaul constructor

treeNode(int nodeWeight, byte nodeValue)

treeNode(treeNode leftNode, treeNode rightNode, int cammonWeight)

int CompareTo(treeNode anotherNode) //compare current node with another one

};

TreeClass – класс для работы с деревом

class treeNode

{

private:

treeNode leafNode; //a leaf node of the tree

Dictionary<byte, string> codesTable; //table with code for each symbol

void createByteCodeTable(treeNode root, string startStr)

//

// Method for fill up table with code each symbol

//

public:

TreeClass(treeNode node) //deafault constructor

static TreeClass createTree(Dictionary<byte, int> occurTable)

//

// Method for parsing occurances table

//

Dictionary<byte, string> getByteCodeTable()

//

// Method for getting code for each symbol

//

List<byte> getByteText //method to create list with original text

};

treeNodeParser– класс для работы с элементами дерева

class treeNodeParser

{

private:

treeNode[] nodes; //an array of nodes

int amount; //amoutn of nodes

int getParentIndex(int index)

//

// method to get parent index

//

int getLeftNodeIndex(int index) // method to get left index

int getRightNodeIndex(int index) // method to get right index

void swapElements(ref treeNode node1, ref treeNode node2)

//

// method to swap two nodes

//

void setChildPositionsFor(int index)

//

// method to recalculate minimal values in nodes list

//

public:

treeNodeParser() //deafault constructor

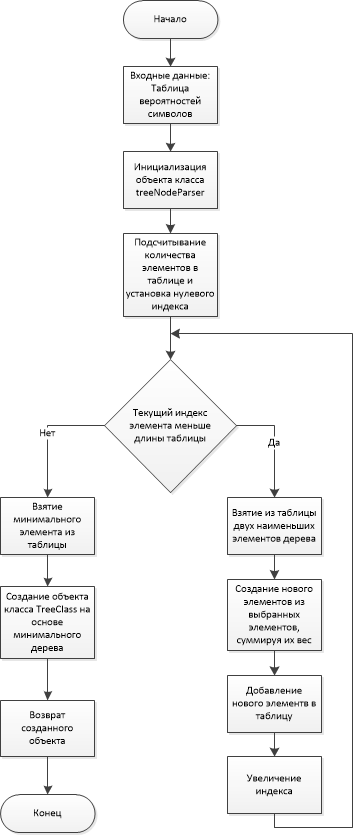
void addNode(treeNode newNode) //add a new node to list

treeNode popMinNode() //pop a minimal node from list

};

**6 БЛОК-СХЕМА АЛГОРИТМА**

**6.1 Метод TreeClass createTree(Dictionary<byte, int> occurTable) класса TreeClass–** метод построения дерева элементов на основе таблицы вероятностей

****

**7 АЛГОРИТМЫ ПО ШАГАМ**

**7.1 Метод void createByteCodeTable(treeNode root, string startStr) класса TreeClass–** метод построения таблицы кодов для каждого символа

1. Начало.
2. Входные данные:

root– указатель на элемент дерева;

startStr– строка с кодов символа;

Выходные данные:

codesTable–таблица кодов для каждого символа;

1. Если root.value не равен null
   1. Начинаем проверять элементы поддерева
      1. Если root.left не равен null
         1. Переходи в левую ветвь и прибавляем к startStr «0» createByteCodeTable(root.left, startStr + "0") функция вызывает саму себя, с другими параметрами
      2. Если root.right не равен null
         1. Переходи в правую ветвь и прибавляем к startStr «1» createByteCodeTable(root.left, startStr + "1") функция вызывает саму себя, с другими параметрами

3.2 Иначе заносим в таблицы кодов текущее значение значение startString - codesTable[Convert.ToByte(root.value)] = startStr.ToString()

1. Конец.

**8 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ**

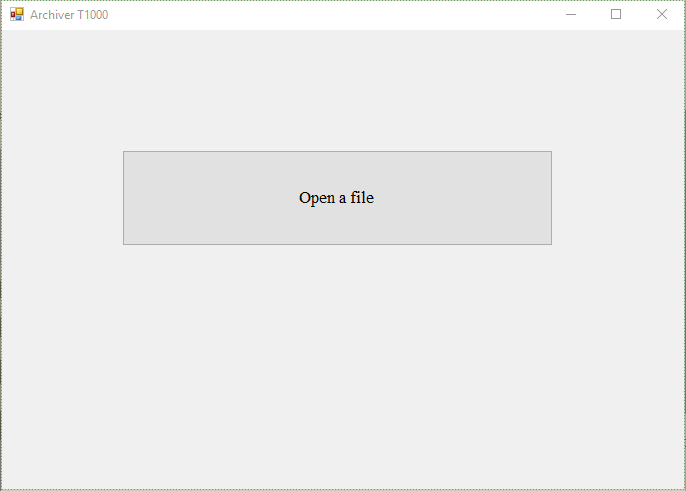
****

Рисунок 8.1 – Начало программы

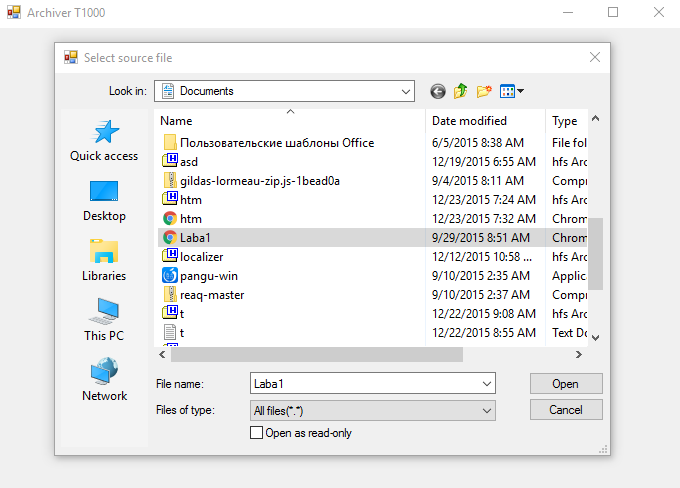


Рисунок 8.2 – Выбор файла для работы

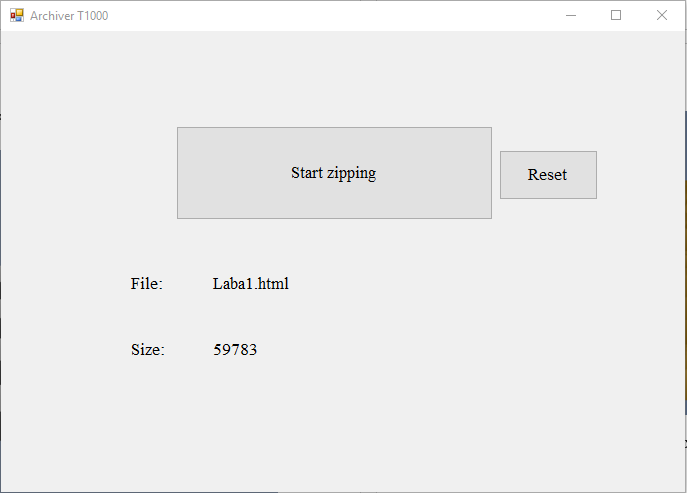


Рисунок 8.3 – Выбор операции с файлом

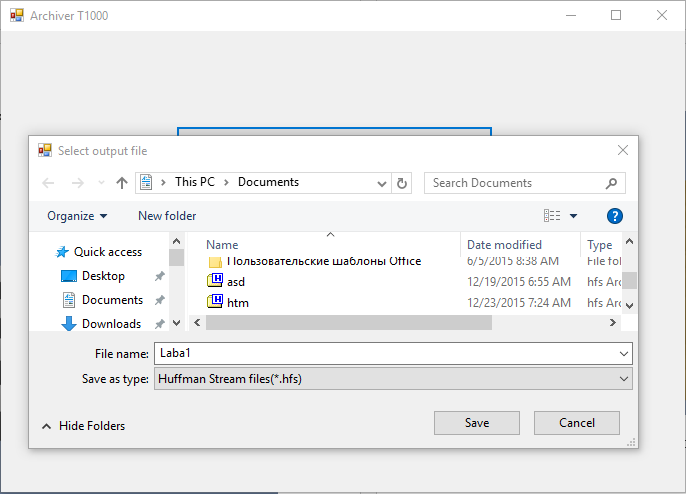


Рисунок 8.4 – Указание пути для архивации файла

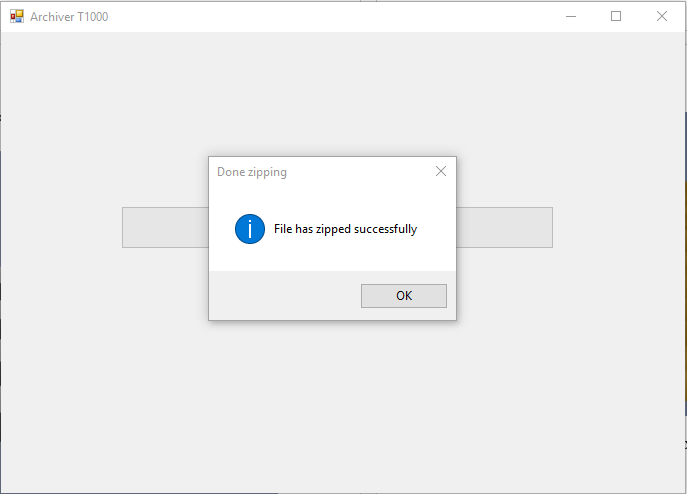


Рисунок 8.5 – Вывод сообщения об успешной архивации

**ЗАКЛЮЧЕНИЕ**

В рамках данной работы был разработан архиватор файлов. Учитывая простой интерфейс, она будет доступна всем возрастным группам.

В будущем этот проект можно усовершенствовать, обновив интерфейс, добавив возможность архивации папок, архивации с паролем, а также возможность собирать в интерфейс программы любое количество интересующих папок и файлов с последующей архивацией в один файл. В зависимости от файла коэффициент сжатия может составлять 20-45%.

Поддерживаемые операционные системы: Microsoft Windows XP, Vista, 7, 8, 10, а также Windows Server.

**СПИСОК ЛИТЕРАТУРЫ**

1. Gornakov S. «CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#» 4-е изд
2. А. Хейлсберг, М. Торгерсен, С. Вилтамут, П. Голд. «Язык программирования C#. Классика Computers Science. 4-е изд.». СПб.: Питер, 2005 – 896 стр.; ил. ISBN 5-7502-0064-7
3. Э. Стиллмен, Дж. Грин. «Изучаем C#» СПб.:БХВ-Петербург, 2003. – 688 с. ISBN 978-5-496-00867-9
4. “Структуры данных и алгоритмы сжатия информации без потерь” – Методическое пособие, Пантеллев Е.Р. – Иваново 2001г.
5. “Алгоритмы: построение и анализ”, Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. – 2е издание, Издательский дом “Вильямс”, 2007г.

**ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПРОГРАММЫ**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.IO;

//using JadBenAutho.Tools;

namespace WindowsFormsApplication1

{

public partial class mainForm : Form

{

public mainForm()

{

InitializeComponent();

resetAllToStart();

}

//private HuffmanAlgorithm huffmanAlg = new HuffmanAlgorithm();

private HaffmanClass haffman = new HaffmanClass();

//texts

private string fileNotExist = "Invalid file path",

fileNotExistCaption = "File doesn't exist",

outputFilePathNotValid = "Invalid output file path",

outputFilePathNotValidCaption = "Ouptput file doesn't exist",

saveDailogTitle = "Select output file",

openDailogTitle = "Select source file",

zippingButtonText = "Start zipping",

unzippingButtonText = "Start unzipping",

zippingProcessText = "File has zipped successfully",

zippingProcessTextCaption = "Done zipping";

private bool isHfs;

private string lastOpenedExistFileName;

private void resetAllToStart(){

setButtonVisibility(actionButton, false);

processProgress.Visible = false;

processProgress.Value = 0;

setLabelsVisibility(false);

setButtonVisibility(openNewFile, true);

setButtonVisibility(resetButton, false);

}

private void setLabelsVisibility(bool isShowLables){

lableFileName.Enabled = isShowLables;

lableFileName.Visible = isShowLables;

lableSizeFile.Enabled = isShowLables;

lableSizeFile.Visible = isShowLables;

lablelFileNameText.Enabled = isShowLables;

lablelFileNameText.Visible = isShowLables;

lableFileNameSize.Enabled = isShowLables;

lableFileNameSize.Visible = isShowLables;

}

private void setLablesText(string text, int size) {

lablelFileNameText.Text = text;

lableFileNameSize.Text = size.ToString();

}

private string getFileName(string fullFileName) {

Array arr = fullFileName.Split('\\');

int lastIndex = arr.Length - 1;

return arr.GetValue(lastIndex).ToString();

}

private void showWarning(string text, string caption)

{

MessageBox.Show(text, caption, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

private void showInfo(string text, string caption)

{

MessageBox.Show(text, caption, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

private void setButtonVisibility(System.Windows.Forms.Button button, bool isShow = false)

{

button.Enabled = isShow;

button.Visible = isShow;

}

private OpenFileDialog fileOpenDialog()

{

OpenFileDialog openFileD = new OpenFileDialog();

openFileD.FileName = "";

openFileD.Filter = "All files(\*.\*)|\*.\*";

openFileD.Multiselect = false;

openFileD.CheckPathExists = true;

openFileD.ShowHelp = false;

openFileD.Title = openDailogTitle;

openFileD.ShowReadOnly = true;

return openFileD;

}

private SaveFileDialog fileSaveDialog()

{

SaveFileDialog saveFileD = new SaveFileDialog();

saveFileD.DefaultExt = isHfs? "" : "\*.hfs";

saveFileD.Filter = isHfs? "All files(\*.\*)|\*.\*" : "Huffman Stream files(\*.hfs)|\*.hfs";

saveFileD.CheckPathExists = false;

saveFileD.ShowHelp = false;

saveFileD.Title = saveDailogTitle;

return saveFileD;

}

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

{

resetAllToStart();

}

private void unzippingLogic(SaveFileDialog saveDialog)

{

string outputFile = saveDialog.FileName;

haffman.Unzip(lastOpenedExistFileName, outputFile);

showInfo("unzip done", zippingProcessTextCaption);

resetAllToStart();

}

private void zippingLogic(SaveFileDialog saveDialog)

{

string outputFile = saveDialog.FileName;

//huffmanAlg.PercentCompleted += new PercentCompletedEventHandler(progressStep);

//FileStream D = new FileStream(outputFile, FileMode.Create);

//D.Close();

// create stream to a zipped file

//FileStream zipStream = new FileStream(lastOpenedExistFileName, FileMode.Open);

haffman.Zip(lastOpenedExistFileName, outputFile);

//huffmanAlg.ShrinkWithProgress(zipStream, outputFile, null);

//zipStream.Close();

showInfo(zippingProcessText, zippingProcessTextCaption);

resetAllToStart();

}

private bool isHfsFile( string fileName) {

string type = "hfs";

Array arr = fileName.Split('.');

int lastIndex = arr.Length - 1;

return arr.GetValue(lastIndex).ToString() == type;

}

private void updateActionButtonText(bool isZipping)

{

actionButton.Text = !isZipping ? zippingButtonText : unzippingButtonText;

}

private void button1\_Click(object sender, EventArgs e)

{

//create a dialog to open file

OpenFileDialog dialog = fileOpenDialog();

DialogResult result = dialog.ShowDialog();

//if dialog closed and selected file exists -> continue

if (result == DialogResult.OK && File.Exists(dialog.FileName)) {

String fileName = dialog.FileName;

lastOpenedExistFileName = fileName;

isHfs = isHfsFile(dialog.SafeFileName);

setButtonVisibility(openNewFile, false);

setButtonVisibility(resetButton, true);

//update action button state

setButtonVisibility(actionButton, true);

updateActionButtonText(isHfs);

//update file info labels

String text = File.ReadAllText(fileName);

setLabelsVisibility(true);

setLablesText(getFileName(fileName), text.Length);

} else if (result != DialogResult.Cancel && result != DialogResult.Abort){

//selected file doesn't exist :(

showWarning(fileNotExist, fileNotExistCaption);

}

}

private void progressStep()

{

processProgress.PerformStep();

}

private void button1\_Click\_1(object sender, EventArgs e)

{

if (lastOpenedExistFileName != "")

{

setLabelsVisibility(false);

SaveFileDialog saveDialog = fileSaveDialog();

DialogResult result = saveDialog.ShowDialog();

if (result == DialogResult.OK && Path.GetFileName(saveDialog.FileName).Length != 0)

{

setButtonVisibility(actionButton, false);

setButtonVisibility(resetButton, false);

//show progress bar

processProgress.Visible = true;

if (isHfs){

unzippingLogic(saveDialog);

}

else {

zippingLogic(saveDialog);

}

} else if(result != DialogResult.Cancel && result != DialogResult.Abort) {

//selected output file doesn't exist :(

showWarning(outputFilePathNotValid, outputFilePathNotValidCaption);

}

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApplication1

{

class converterClass

{

private int byte2 = 256;

private int byte3 = 65536;

private int byte4 = 16777216;

private string errorMessage = "Cannot convert";

public int byteToInt(byte[] bytes)

{

if (bytes.Length != 4)

{

throw new Exception(errorMessage);

}

return bytes[0] + bytes[1] \* byte2 + bytes[2] \* byte3 + bytes[3] \* byte4;

}

public byte[] intToByte(int number)

{

if (number > 0)

{

int temp = number,

i;

byte[] bytes= new byte[4];

for(i = 0; i < 4; i++)

{

bytes[i] = (byte)(temp % 256);

temp /= 256;

}

return bytes;

} else

{

throw new Exception(errorMessage);

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApplication1

{

class treeNodeParser

{

private treeNode[] nodes = new treeNode[2000];

private int amount;

public treeNodeParser()

{

nodes = new treeNode[2000];

amount = 0;

}

//function for getting indexes

private int getParentIndex(int index)

{

return index / 2;

}

//left node will be have index like current node index\*2

private int getLeftNodeIndex(int index)

{

return index \* 2;

}

//right node will be have index like current node index\*2 + 1, because (index\*2) - it's left node

private int getRightNodeIndex(int index)

{

return (index \* 2) + 1;

}

private void swapElements(ref treeNode node1, ref treeNode node2)

{

treeNode temp = node1;

node1 = node2;

node2 = temp;

}

public void addNode(treeNode newNode)

{

amount++;

int i = amount;

nodes[i] = newNode;

while( i > 1 && nodes[getParentIndex(i)].CompareTo(nodes[i]) > 0)

{

int parentIndex = getParentIndex(i);

swapElements(ref nodes[i], ref nodes[parentIndex]);

i = parentIndex;

}

}

public treeNode popMinNode()

{

treeNode minNode = nodes[1];

nodes[1] = nodes[amount];

amount--;

setChildPositionsFor(1);

return minNode;

}

private void setChildPositionsFor(int index) {

int leftIndex = getLeftNodeIndex(index),

rightIndex = getRightNodeIndex(index),

smallestIndex;

if(leftIndex <= amount && nodes[leftIndex].CompareTo(nodes[index]) < 0) {

smallestIndex = leftIndex;

} else

{

smallestIndex = index;

}

if (rightIndex <= amount && nodes[rightIndex].CompareTo(nodes[smallestIndex]) < 0)

{

smallestIndex = rightIndex;

}

if (smallestIndex != index)

{

swapElements(ref nodes[index], ref nodes[smallestIndex]);

setChildPositionsFor(smallestIndex);

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApplication1

{

class treeNode

{

public byte? value { get; set; }

public int weight { get; set; }

public treeNode right { get; set; }

public treeNode left { get; set; }

//def constructor

public treeNode() {

value = null;

weight = 0;

}

public treeNode(int nodeWeight, byte nodeValue)

{

weight = nodeWeight;

value = nodeValue;

}

public treeNode(treeNode leftNode, treeNode rightNode, int cammonWeight)

{

left = leftNode;

right = rightNode;

weight = cammonWeight;

}

public int CompareTo(treeNode anotherNode)

{

if(weight > anotherNode.weight)

{

return 1;

} else if (weight < anotherNode.weight)

{

return -1;

}

return 0;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApplication1

{

class TreeClass

{

private treeNode leafNode;

private Dictionary<byte, string> codesTable;

//default constructor

public TreeClass(treeNode node)

{

leafNode = node;

codesTable = new Dictionary<byte, string>();

}

public static TreeClass createTree(Dictionary<byte, int> occurTable)

{

//need create something like list of treeNodes

treeNodeParser nodeParser = new treeNodeParser();

//add each occurance to the ordered list of nodes

foreach(var occur in occurTable)

{

nodeParser.addNode(new treeNode(occur.Value, occur.Key));

}

int amount = occurTable.Count;

for (int i = 0; i < amount -1; ++i)

{

//get last to nodes, that have minimally weight

treeNode leftNode = nodeParser.popMinNode();

treeNode rightNode = nodeParser.popMinNode();

//create from it a new node with a new weight and add new node to ordered list

nodeParser.addNode(new treeNode(leftNode, rightNode, leftNode.weight + rightNode.weight));

}

//should return full tree node

//after all cyrcles, when we get min node, it will have links to sub notes, the notes to itself subnotes and etc. :)

return new TreeClass(nodeParser.popMinNode());

}

public Dictionary<byte, string> getByteCodeTable()

{

if (leafNode == null)

{

throw new Exception("Tree' leaf node shouldn't be empty");

} else

{

//fill up for each original byte it new code

createByteCodeTable(leafNode, "");

//return calculated table

return codesTable;

}

}

private void createByteCodeTable(treeNode root, string startStr)

{

//if node doesn't have value

if (root.value == null)

{

//and have left node

if (root.left != null)

{

//add to output string '0'

createByteCodeTable(root.left, startStr + "0");

}

//and have right node

if (root.right != null)

{

//add to output string '1'

createByteCodeTable(root.right, startStr + "1");

}

} else

{

//if node has value -> add for the key value output string calculated before

codesTable[Convert.ToByte(root.value)] = startStr.ToString();

}

}

public List<byte> getByteText(string text)

{

//we have string read form zipped file

int i, len = text.Length;

//create list of bytes

List<byte> resultText = new List<byte>();

for (i = 0; i < len; i++)

{

//set node leaf node of calculated tree

treeNode node = leafNode;

while (true)

{

if (i >= len)

{

break;

}

//if value exists -> we find original byte, add it to list

if (node.value != null)

{

resultText.Add(Convert.ToByte(node.value));

i--;

break;

} else

{

//else if currant symbol equl to 0 -> go to left node, otherwise - to right node

node = (text[i] == '0') ? node.left : node.right;

i++;

}

}

}

//return list of original bytes

return resultText;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.IO;

namespace WindowsFormsApplication1

{

class HaffmanClass

{

private converterClass convert = new converterClass();

public void Zip(string readFile, string writeFile)

{

//vars

int i, j;

byte writeByte, pow;

long len;

//1) read file and fill up occurances

//open file for reading

FileStream readStream = new FileStream(readFile, FileMode.Open, FileAccess.Read);

//fill up occurance of elements dictionary

Dictionary<byte, int> symbolOccurrences = new Dictionary<byte, int>();

List<byte> input = new List<byte>();

len = readStream.Length;

for (i = 0; i < len; i++)

{

//read byte from file

byte readByte = (byte)readStream.ReadByte();

//add it to hte input list

input.Add(readByte);

//increment counter for this byte in occurances table

if (symbolOccurrences.ContainsKey(readByte))

{

symbolOccurrences[readByte]++;

} else

{

symbolOccurrences.Add(readByte, 1);

}

}

//close file

readStream.Close();

//2) write length of occurances and occurances to output file: create functions for convertion int to byte and back

FileStream writeStream = new FileStream(writeFile, FileMode.Create, FileAccess.Write);

//write occurances length

//convert size of table to bytes and write those bytes to output file

byte[] countInByte = convert.intToByte(symbolOccurrences.Count);

for (i = 0; i < 4; i++)

{

writeStream.WriteByte(countInByte[i]);

}

//write occurances

foreach (var occur in symbolOccurrences)

{

//write occurance to output file

writeStream.WriteByte(occur.Key);

//convert amount of occurances fot the bytes and write to file

byte[] occurInByte = convert.intToByte(occur.Value);

for (i = 0; i < 4; i++)

{

writeStream.WriteByte(occurInByte[i]);

}

}

//3) create tree from occurances: create Tree class

TreeClass tree = TreeClass.createTree(symbolOccurrences);

//4) create code for each element from occurance dictionary

Dictionary<byte, string> codes = tree.getByteCodeTable();

//use stringBuilder to create output string for writting to file

StringBuilder resultString = new StringBuilder("");

len = input.Count;

for (i = 0; i < len; i++)

{

resultString.Append(codes[input[i]].ToString());

}

//5) write symbol's code to output file

//write to file size of last row

writeStream.WriteByte(Convert.ToByte(resultString.Length % 8));

len = resultString.Length;

writeByte = 0;

j = 0;

pow = 1;

//write output string using easy logic

for (i = 0; i < len; i++)

{

if(resultString[i] == '1')

{

writeByte += pow;

}

if(j == 7)

{

writeStream.WriteByte(writeByte);

pow = 1;

writeByte = 0;

} else

{

pow \*= 2;

}

j = (j + 1) % 8;

}

//if last row not full -> write last byte

if(resultString.Length % 8 != 0)

{

writeStream.WriteByte(writeByte);

}

//close output file

writeStream.Close();

}

public void Unzip(string readFile, string writeFile)

{

byte[] byteArr = new byte[4];

byte tableKey;

int i, j, tableSize;

long len = 4

//1) read file, length of occurances and fill up occurances

FileStream readStream = new FileStream(readFile, FileMode.Open, FileAccess.Read);

//create occurances table

Dictionary<byte, int> symbolOccurrences = new Dictionary<byte, int>();

//read 4 byte in conver it to occurance table size

for (i = 0; i < len; i++)

{

byteArr[i] = (byte)readStream.ReadByte();

}

tableSize = convert.byteToInt(byteArr);

byteArr = new byte[4];

//read bytes and fill up occurances table

for (i = 0; i < tableSize; i++)

{

//read key first

tableKey = (byte)readStream.ReadByte();

//then read 4 bytes and covert it to int(amount of accorances)

for (j = 0; j < len; j++)

{

byteArr[j] = (byte)readStream.ReadByte();

}

symbolOccurrences.Add(tableKey, convert.byteToInt(byteArr));

}

//2) create tree using occurances and then recognize file text

TreeClass tree = TreeClass.createTree(symbolOccurrences);

//read last row length from file

byte lastRow = (byte)readStream.ReadByte();

//calculate table size: file length - 4Bytes(table size) - 1Byte(length last row) - 5\* table size(1Byte key + 4Bytes occurances)

len = readStream.Length - 1 - 4 - tableSize \* 5;

//create stringBuilder to fill up all text

StringBuilder input = new StringBuilder("");

//use easy logic for reading text

for (i = 0; i < len; i++)

{

tableKey = (byte)readStream.ReadByte();

if (i == (readStream.Length - 2) && lastRow != 0)

{

for (j = 0; j <= lastRow; j++)

{

input.Append((tableKey % 2).ToString());

tableKey /= 2;

}

} else

{

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

{

input.Append((tableKey % 2).ToString());

tableKey /= 2;

}

}

}

//close file

readStream.Close();

//3) write to output file

//parse text to get bytes of original text

List<byte> text = tree.getByteText(input.ToString());

//open output file for writting

FileStream writeStream = new FileStream(writeFile, FileMode.Create, FileAccess.Write);

//write original bytes to it

writeStream.Write(text.ToArray(), 0, text.Count);

//close output file

writeStream.Close();

}

}

}