Лабораторная работа №1

**Сжатие данных**

Написать программу, сжимающую данные из некоторого файла по алгоритму Хаффмана.

Программа должна выполнять следующие действия:

Сжатие данных

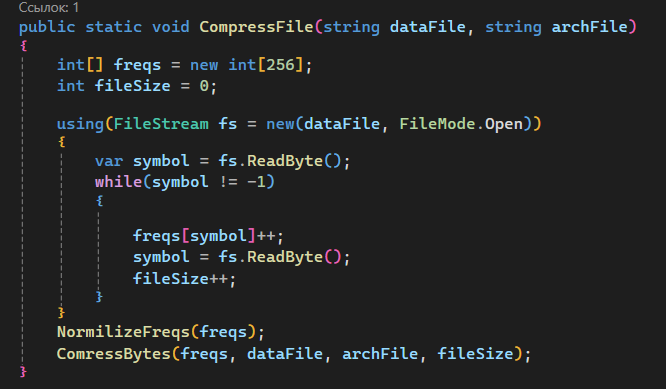
1. Открывать файл, подлежащий сжатию, как бинарный файл байтов.
2. Подсчитывать частоту вхождения каждого байта в файле.
3. По имеющимся частотам строить дерево Хаффмана.
4. Создавать новый файл-архив.
5. Записывать в файл-архив заголовок – байты с частотами их вхождения.
6. Последовательно считывать байты исходного файла, кодировать их и записывать в файл-архив.

Распаковка данных

1. Открывать файл-архив.
2. Считывать из файла-архива заголовок и строить по нему дерево Хаффмана.
3. Последовательно считывать байты файла-архива, анализировать байты побитно с помощью дерева Хаффмана и записывать найденные в дереве байты в новый файл.

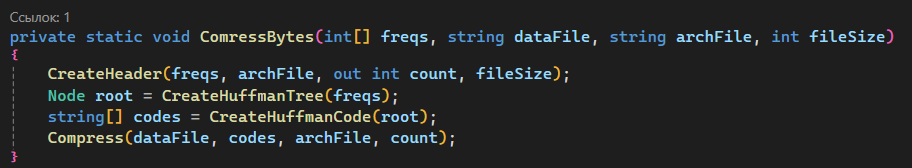
Алгоритм:

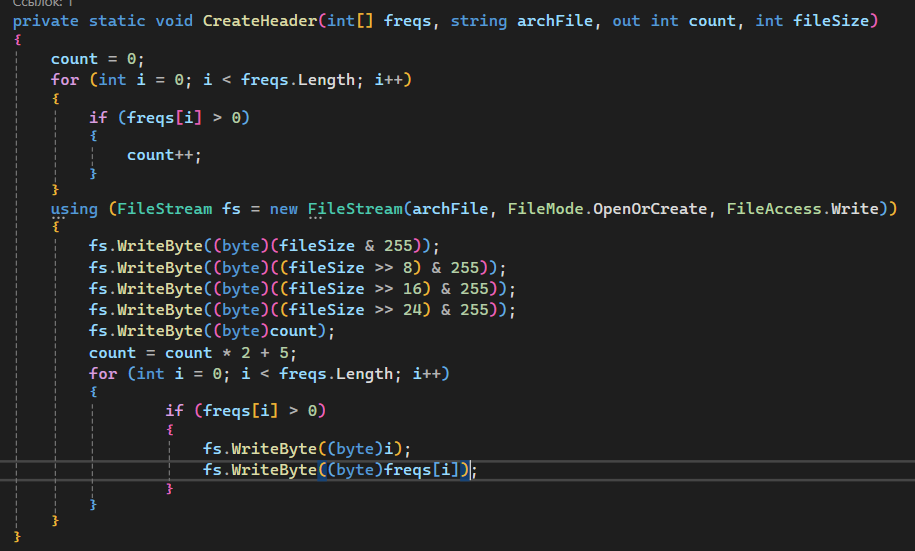
Работа сжатия данных:

В методе CompressFile сначала создаем массив частот размером 256(256 это всевозможные индексы символов ASCII кода). Так как в языке C# созданный массив инициализируется значениями по умолчанию, то наш массив уже заполнен нулями, поэтому можно не заполнять массив нулями самостоятельно. Так же создаем переменную int fileSize, в которой будем хранить размер исходного файла. Через конструкцию using открываем файл и считываем байты(т.е. символ из файла). Пока метод ReadByte не вернет -1, то в массиве частот увеличиваем на 1 по индексу символа и увеличиваем fileSize на 1. После этого вызываем внутренний метод NormilizeFreqs, который проверяет нужно ли нормализовать частоты. Нормализовать частоты символов нужно тогда, когда максимальная частота является больше 255(255 это предел байта, поэтому если частота символа будет больше 255, то при распаковке файла эта частота будет равна 255, а не изначальной частоте и будет неверная распаковка). Нормализуется частота по формуле:

Добавляем 1 потому, что частота символа может быть маленькой изначально и при нормализации будет близка к нулю. В знаменателе максимальную частоту увеличиваем на 1 потому, что чтобы частоты не могли пересечь 255.

В конце метода вызываем CompressBytes.

В методе CompressBytes сначала вызываем метод CreateHeader, который запишет в заархивированный файл заголовок файла.

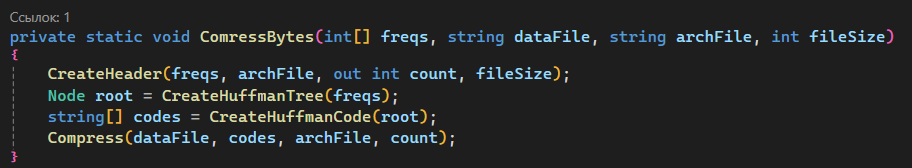
Сначала считаем через цикл сколько у нас вообще символов и записываем в переменную count.

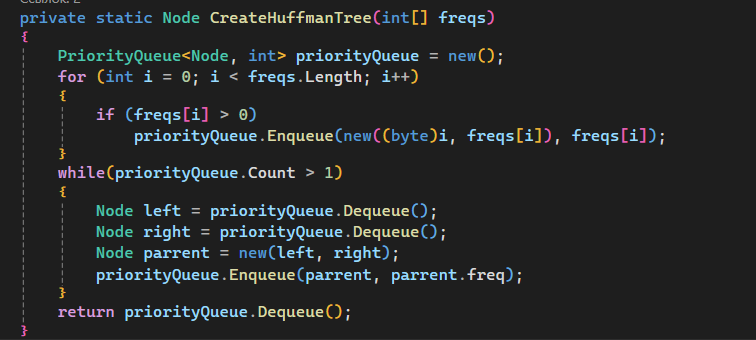
Далее открываем файл через конструкцию using. Первые четыре строчки будут занимать размер файла(4 строчки потому, что если все записать все в одну, то при распаковки нам вернет только 255)

В пятую строку заголовка мы мы запишем количество символов в исходном файле. Затем увеличиваем count в 2 раза и увеличиваем на 5, чтобы потом могли начать записывать код с этого места, а не перезаписать размер.

Далее записываем символ и его частоту.

Перейдем к коду. После открытия файла записываем первые 8 бит, потом следующие 8 бит и так еще 2 раза. После записи размера файла, запишем сколько символов в файле. После всего этого в список добавляем через цикл символ и его частоту, если частота больше 0.

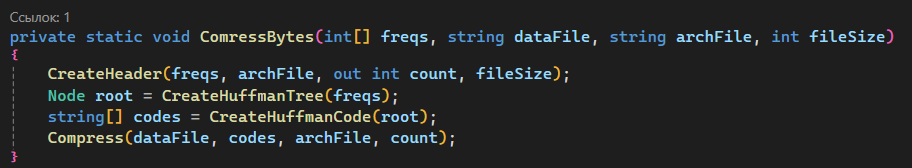
После создания заголовка нужно построить дерево Хаффмана. Сделаем это через метод CreateHuffmanTree, которая принимает частоты и возвращает корень дерева.



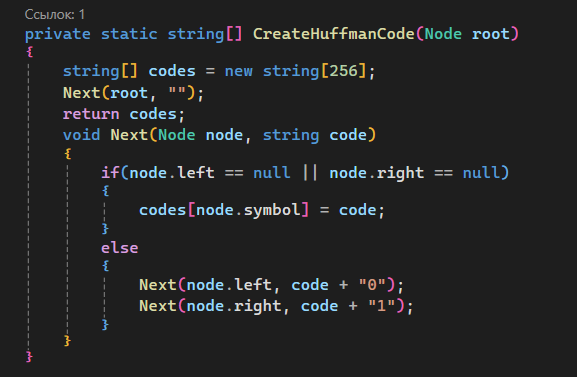
Строить дерево будем через очередь с приоритетом, в которой будет хранится узел и частота.

Очередь с приоритетом – это структура данных, которая является очередью, но удаления элемента работает немного по другому. Удаляется сначала тот элемент, у которого минимальный элемент, но если приоритет у всех одинаковый, то она работает как стандартная очередь.

Через цикл заполняем очередь теми символами, у которых частота больше нуля. После заполнения очереди начнем создавать дерево. Пока в очереди больше 1 элемента будем удалять из очереди два элемента и создавать родителя этих узлов, в котором будут эти два узла и новая частота(сумма двух частот данных узлов), и добавляем его в очередь. Как в очереди останется один элемент, т.е. корень дерева, возвращаем его в конце метода.



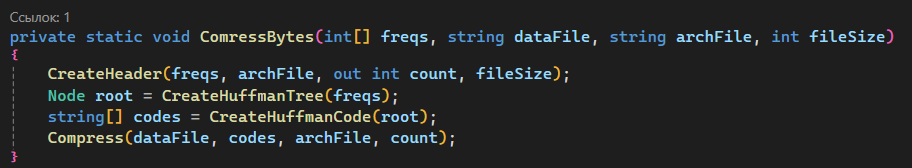
После построения дерева нам нужно построить коды символов. Сделаем это через метод CreateHuffmanCode, которая принимает корень дерева и вернет массив строк с кодами.



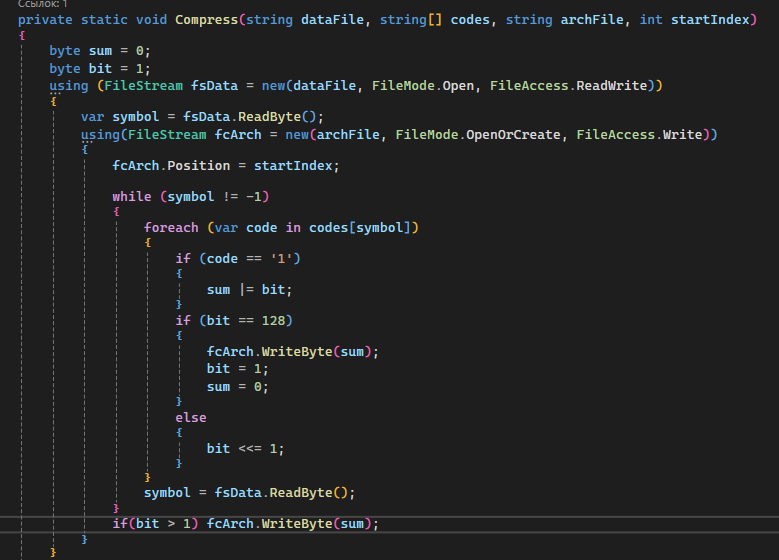
Создаем массив строк размеров 256 и вызываем рекурсивную функцию Next, которая принимает корень дерева и строку после выполнения возвращаем массив кодов.

В функции Next проверяем узел на крайний случай, а именно, что у узла больше нет листьев. Если да, то записываем в массив строк код символа.

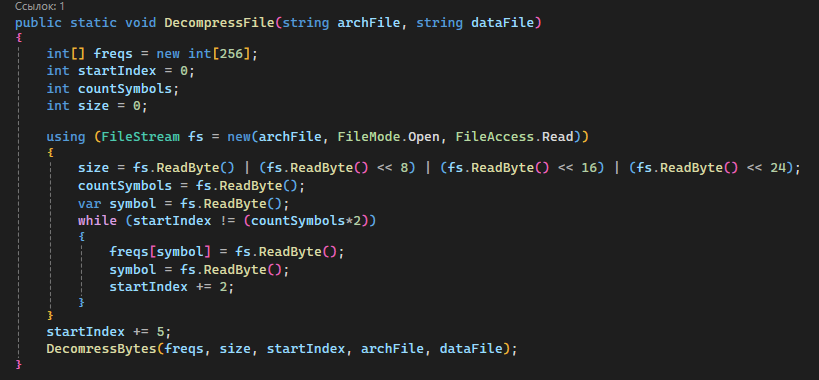
Если нет, то вызываем функцию для левого листа и добавляем к коду 0 и для правого листа и добавляем к коду 1.



После полученных кодов нужно сжать исходный массив байт. Сделаем это через метод Compress.

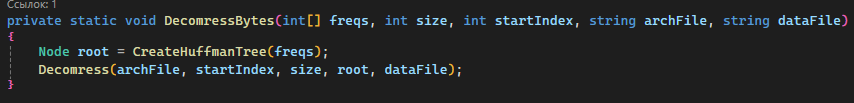
Создаем переменную sum, в которой будем хранить полученную сумму из кода. Так же создаем переменную bit, через нее мы будем увеличивать переменную sum в цикле. В цикле считываем байты исходных данных, внутри цикла проходимся коду символа через вложенный цикл. внутри вложенного цикла проверяем равен ли символ 1, если да, то увеличиваем sum через бинарное или. Если bit равен 128, то записываем sum в файл-архив с позиции startIndex и задаем sum = 0, а bit = 1. Если не равно 128, то увеличиваем bit по сдвигу влево. После цикла проверяем больше ли bit 1. Если да, то значит мы записали не все, что нужно и записываем в файл-архив остаток.

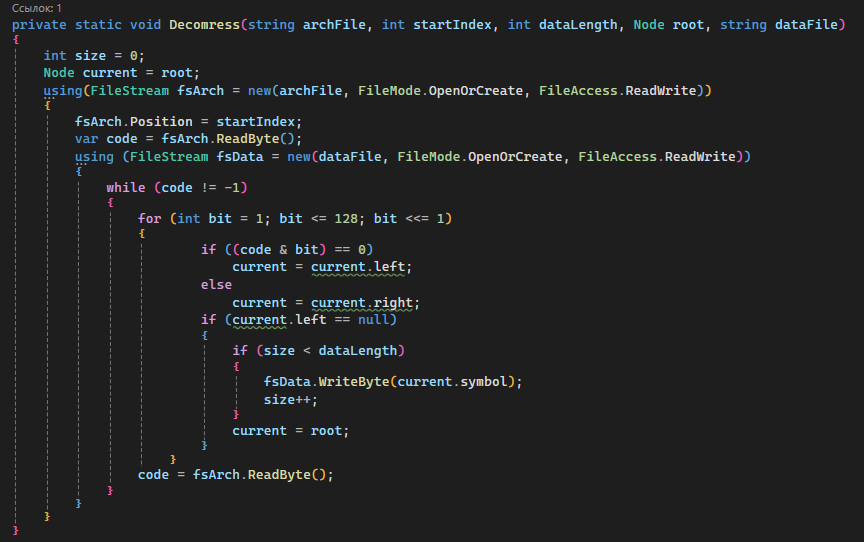
Работа распаковки:



Для начала создаем необходимые переменные: массив частот, кол-во символов, начальный индекс, размер. Распаковка файлов начинается с того, что считываем сначала заголовок. Для начала прочитаем длину данных. Это делается так же, как и запись, только наоборот. Теперь мы берем с первого элемента по четвертый и складываем их через логическое или. Со второго элемента делаем сдвиг влево по 8n, где n от 1 до 3.

После получение длины данных создаем массив частот с размеров 256. После этого считываем с заархивированных данных кол-во элементов. Считываем данные пока startIndex не будет равен кол-во символов умноженного на 2. Это сделано так, чтобы мы не прочитали сжатые коды, которые находятся ниже заголовка. Внутри цикла увеличиваем startIndex на 2, а после цикла добавляем 5. Затем вызываем метод DecompressBytes.

После считывания заголовка у нас есть массив частот, поэтому мы можем построить дерево Хаффмана через метод CreateHuffmanTree. После получения дерева вызываем метод Decompress.



Сначала создаем переменную size, через которую будем обращаться к распакованному массиву. Создаем переменную current, в которую записываем корень дерева. Через нее будем проходить по дереву. В файл-архив проходимся по заархивированным данным с начального индекса, т. е. с переменной startIndex. Внутри цикла проходим по еще одному массиву, в котором будет находиться до того момента, как bit станет больше 128. Внутри вложенного цикла проверяем равен ли символ 0 через логическое и. Если да, то идем по дереву влево, иначе вправо. Если дальше по дереву ничего нет(нет больше листов), то проверяем меньше ли переменная size длины данных. Если да, то записываем символ в файл и увеличиваем size на 1. Сбрасываем current до root.

Код:

using System.Drawing;

namespace Lr6

{

public class HuffmanTree

{

private class Node

{

public readonly int freq;

public readonly byte symbol;

public readonly Node? left;

public readonly Node? right;

public Node(byte symbol, int freq)

{

this.symbol = symbol;

this.freq = freq;

}

public Node(Node left, Node right)

{

this.freq= left.freq + right.freq;

this.left = left;

this.right = right;

}

}

public static void CompressFile(string dataFile, string archFile)

{

int[] freqs = new int[256];

int fileSize = 0;

using(FileStream fs = new(dataFile, FileMode.Open))

{

var symbol = fs.ReadByte();

while(symbol != -1)

{

freqs[symbol]++;

symbol = fs.ReadByte();

fileSize++;

}

}

NormilizeFreqs(freqs);

ComressBytes(freqs, dataFile, archFile, fileSize);

}

private static void NormilizeFreqs(int[] freqs)

{

int max = freqs.Max();

if (max <= 255) return;

for (int i = 0; i < 256; i++)

{

if (freqs[i] > 0)

{

freqs[i] = 1 + (freqs[i] \* 255) / (max + 1);

}

}

}

private static void ComressBytes(int[] freqs, string dataFile, string archFile, int fileSize)

{

CreateHeader(freqs, archFile, out int count, fileSize);

Node root = CreateHuffmanTree(freqs);

string[] codes = CreateHuffmanCode(root);

Compress(dataFile, codes, archFile, count);

}

private static void Compress(string dataFile, string[] codes, string archFile, int startIndex)

{

byte sum = 0;

byte bit = 1;

using (FileStream fsData = new(dataFile, FileMode.Open, FileAccess.ReadWrite))

{

var symbol = fsData.ReadByte();

using(FileStream fcArch = new(archFile, FileMode.OpenOrCreate, FileAccess.Write))

{

fcArch.Position = startIndex;

while (symbol != -1)

{

foreach (var code in codes[symbol])

{

if (code == '1')

{

sum |= bit;

}

if (bit == 128)

{

fcArch.WriteByte(sum);

bit = 1;

sum = 0;

}

else

{

bit <<= 1;

}

}

symbol = fsData.ReadByte();

}

if(bit > 1) fcArch.WriteByte(sum);

}

}

}

private static string[] CreateHuffmanCode(Node root)

{

string[] codes = new string[256];

Next(root, "");

return codes;

void Next(Node node, string code)

{

if(node.left == null || node.right == null)

{

codes[node.symbol] = code;

}

else

{

Next(node.left, code + "0");

Next(node.right, code + "1");

}

}

}

private static Node CreateHuffmanTree(int[] freqs)

{

PriorityQueue<Node, int> priorityQueue = new();

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

{

if (freqs[i] > 0)

priorityQueue.Enqueue(new((byte)i, freqs[i]), freqs[i]);

}

while(priorityQueue.Count > 1)

{

Node left = priorityQueue.Dequeue();

Node right = priorityQueue.Dequeue();

Node parrent = new(left, right);

priorityQueue.Enqueue(parrent, parrent.freq);

}

return priorityQueue.Dequeue();

}

private static void CreateHeader(int[] freqs, string archFile, out int count, int fileSize)

{

count = 0;

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

{

if (freqs[i] > 0)

{

count++;

}

}

using (FileStream fs = new FileStream(archFile, FileMode.OpenOrCreate, FileAccess.Write))

{

fs.WriteByte((byte)(fileSize & 255));

fs.WriteByte((byte)((fileSize >> 8) & 255));

fs.WriteByte((byte)((fileSize >> 16) & 255));

fs.WriteByte((byte)((fileSize >> 24) & 255));

fs.WriteByte((byte)count);

count = count \* 2 + 5;

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

{

if (freqs[i] > 0)

{

fs.WriteByte((byte)i);

fs.WriteByte((byte)freqs[i]);

}

}

}

}

public static void DecompressFile(string archFile, string dataFile)

{

int[] freqs = new int[256];

int startIndex = 0;

int countSymbols;

int size = 0;

using (FileStream fs = new(archFile, FileMode.Open, FileAccess.Read))

{

size = fs.ReadByte() | (fs.ReadByte() << 8) | (fs.ReadByte() << 16) | (fs.ReadByte() << 24);

countSymbols = fs.ReadByte();

var symbol = fs.ReadByte();

while (startIndex != (countSymbols\*2))

{

freqs[symbol] = fs.ReadByte();

symbol = fs.ReadByte();

startIndex += 2;

}

}

startIndex += 5;

DecomressBytes(freqs, size, startIndex, archFile, dataFile);

}

private static void DecomressBytes(int[] freqs, int size, int startIndex, string archFile, string dataFile)

{

Node root = CreateHuffmanTree(freqs);

Decomress(archFile, startIndex, size, root, dataFile);

}

private static void Decomress(string archFile, int startIndex, int dataLength, Node root, string dataFile)

{

int size = 0;

Node current = root;

using(FileStream fsArch = new(archFile, FileMode.OpenOrCreate, FileAccess.ReadWrite))

{

fsArch.Position = startIndex;

var code = fsArch.ReadByte();

using (FileStream fsData = new(dataFile, FileMode.OpenOrCreate, FileAccess.ReadWrite))

{

while (code != -1)

{

for (int bit = 1; bit <= 128; bit <<= 1)

{

if ((code & bit) == 0)

current = current.left;

else

current = current.right;

if (current.left == null)

{

if (size < dataLength)

{

fsData.WriteByte(current.symbol);

size++;

}

current = root;

}

}

code = fsArch.ReadByte();

}

}

}

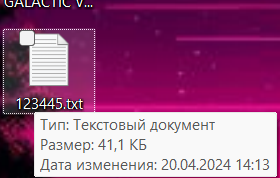
}

}

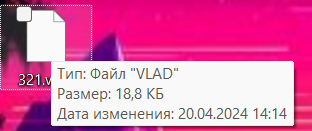
}

Тест:

Файл до сжатия:



После:



Проверка на потерю данных после распаковки:

