Лабораторная работа №1

**Сжатие данных**

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

Программа должна выполнять следующие действия:

Сжатие данных

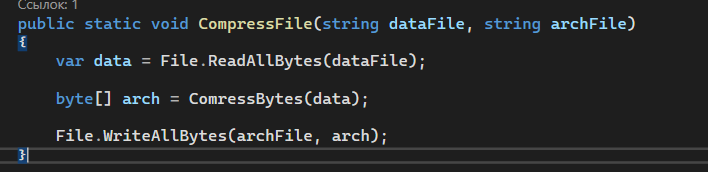
1. Открывать файл, подлежащий сжатию, как бинарный файл байтов.
2. Подсчитывать частоту вхождения каждого байта в файле.
3. По имеющимся частотам строить дерево Хаффмана.
4. Создавать новый файл-архив.
5. Записывать в файл-архив заголовок – байты с частотами их вхождения.
6. Последовательно считывать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

Распаковка данных

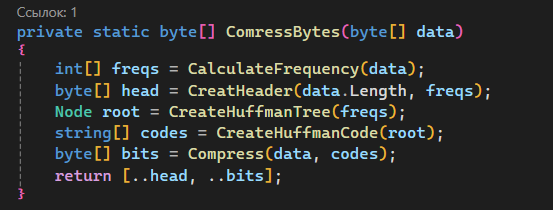
1. Открывать файл-архив.
2. Считывать из файла-архива заголовок и строить по нему дерево Хаффмана.
3. Последовательно считывать байты файла-архива, анализировать байты побитно с помощью дерева Хаффмана и записывать найденные в дереве байты в новый файл.

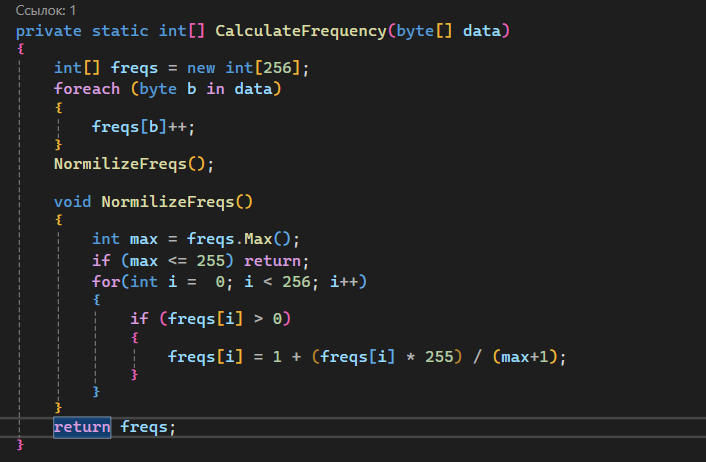
Алгоритм:

Работа сжатия данных:



В методе CompressFile считываем все байты из исходного файла и вызываем метод CompressBytes, который принимает массив байт исходного файла и вернет нам сжатые данные и записываем их в новый файл.

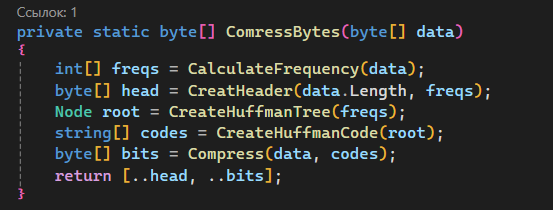
  
В методе CompressBytes сначала считаем частоту символов через метод CalculateFrequency, которая принимает исходные байты и вернет нам массив с частотами.



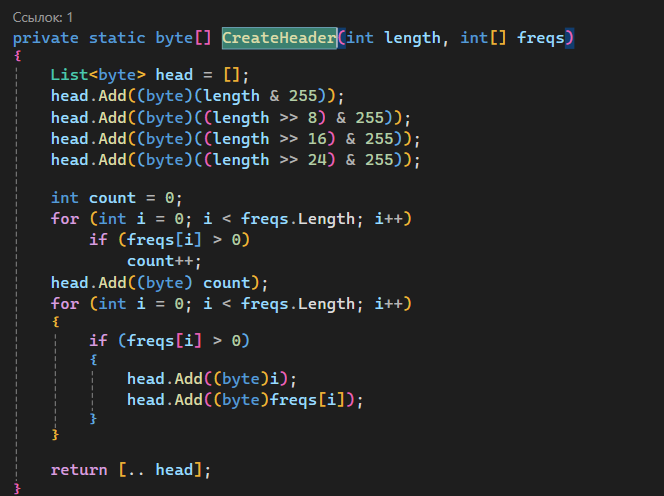
В методе CalculateFrequency сначала создаем массив частот размером 256(256 это всевозможные индексы символов ASCII кода). Так как в языке C# созданный массив инициализируется значениями по умолчанию, то наш массив уже заполнен нулями, поэтому можно не заполнять массив нулями самостоятельно. Через цикл foreach проходим по исходным данным и увеличиваем частоту каждого символа на 1 в массиве freqs. 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 CalculateFrequency возвращаем частоты символов.



После вычисления частот создаем заголовок для файла через метод CreateHeader, чтобы потом могли распаковать файл. Метод принимает длину массива исходных данных и массив частот.



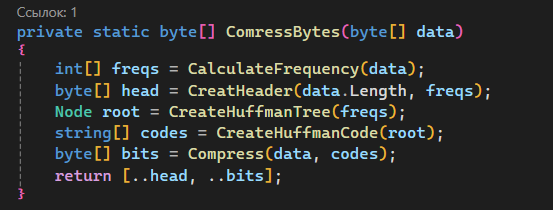
Создаем список байт, в который будем записывать все необходимые данные для заголовка файла. Перед самим описанием алгоритма стоит описать, что будем записывать и как.

Первые четыре строчки будут занимать размер файла(4 строчки потому, что если все записать все в одну, то при распаковки нам вернет только 255)

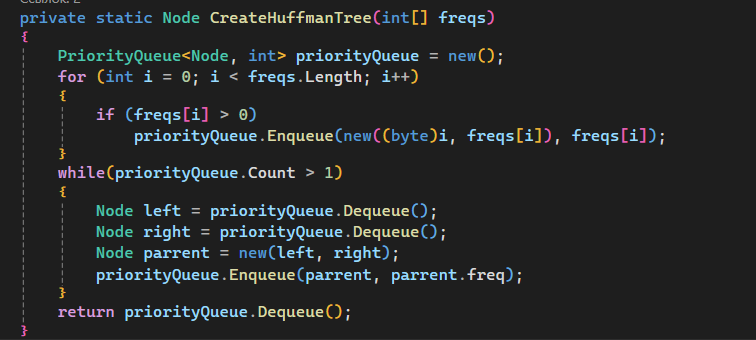
В пятую строку заголовка мы мы запишем количество символов в исходном файле.

Далее записываем символ и его частоту.

Перейдем к коду. После создания списка добавляем в список первые 8 бит, потом следующие 8 бит и так еще 2 раза. После записи размера файла, запишем сколько символов в файле. Сначала посчитаем сколько их, создав счетчик. Сделаем это через цикл, в котором будем проверять если частота больше 0, то символ в файле есть, поэтому увеличиваем счетчик. После цикла добавляем в список счетчик. После всего этого в список добавляем через цикл символ и его частоту, если частота больше 0. Возвращаем готовый заголовок.



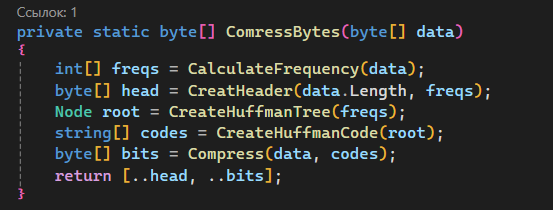
После создания заголовка нужно построить дерево Хаффмана. Сделаем это через метод CreateHuffmanTree, которая принимает частоты и возвращает корень дерева.



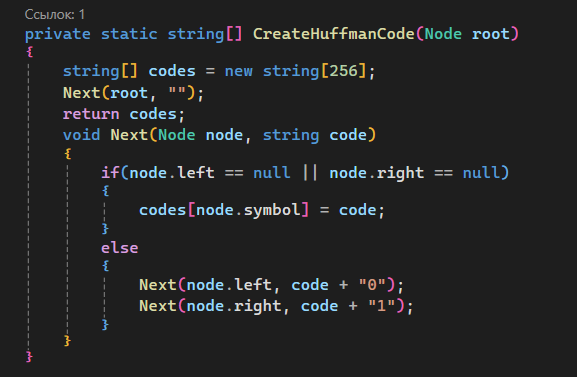
Строить дерево будем через очередь с приоритетом, в которой будет хранится узел и частота.

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.



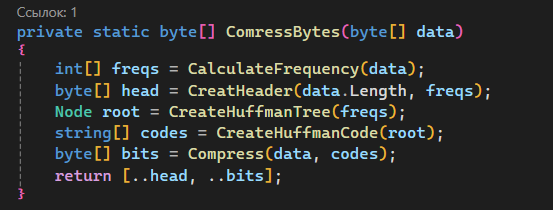
После построения дерева нам нужно построить коды символов. Сделаем это через метод CreateHuffmanCode, которая принимает корень дерева и вернет массив строк с кодами.



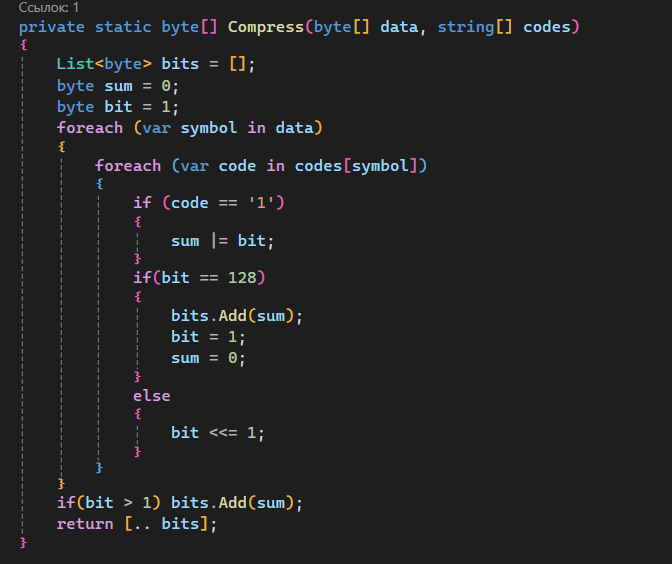
Создаем массив строк размеров 256 и вызываем рекурсивную функцию Next, которая принимает корень дерева и строку после выполнения возвращаем массив кодов.

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.

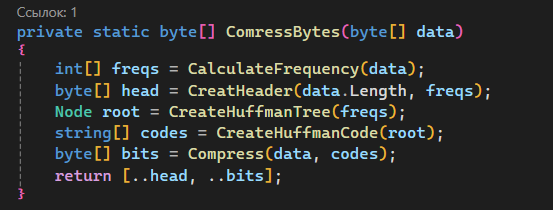
Если нет, то вызываем функцию для левого листа и добавляем к коду 0 и для правого листа и добавляем к коду 1.



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

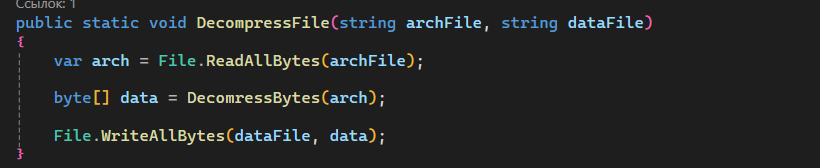


Для начала создаем список байтов, в который будем добавлять сжатый код. Создаем переменную sum, в которой будем хранить полученную сумму из кода. Так же создаем переменную bit, через нее мы будем увеличивать переменную sum в цикле. В цикле проходимся по исходным данным, внутри цикла проходимся коду символа через вложенный цикл. внутри вложенного цикла проверяем равен ли символ 1, если да, то увеличиваем sum через бинарное или. Если bit равен 128, то добавляем sum в список 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 добавляем в список остаток. В конце метода возвращаем список.

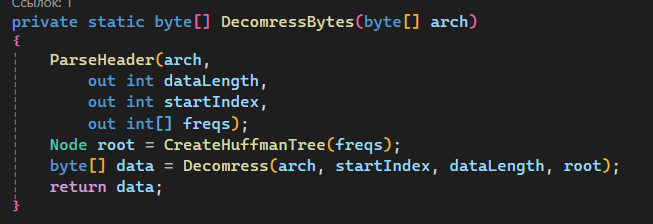


В конце CompressBytes возвращаем массив, в котором идет сначала заголовок, а потом сжатые данные.

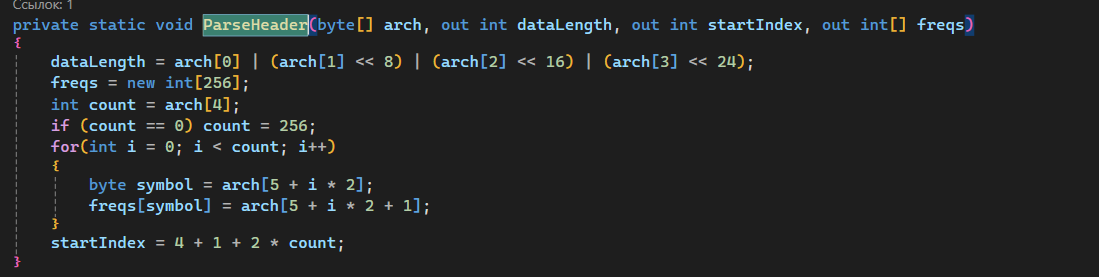
Работа распаковки:



Распаковка файлов начинается с того, что читаем все байты заархивированного файла и вызываем метод DecomressBytes, который вернет распакованные данные. После запишим данные в новый файл.



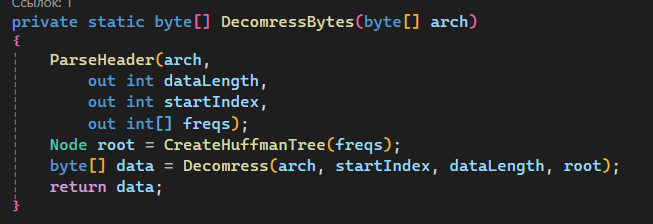
Перед распаковкой данных нужно прочитать заголовок файла. Сделаем это через метод ParseHeader. функция вернет нам длину данных, начальный индекс, с которого нужно начать читать массив, массив частот.



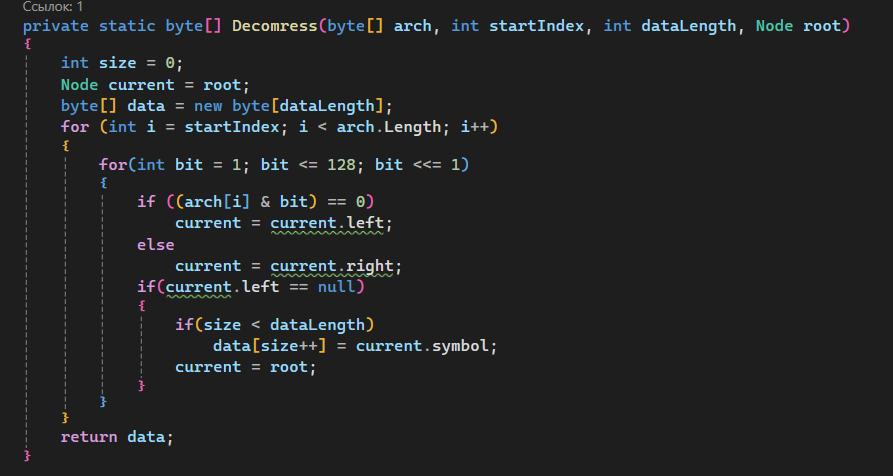
Для начала прочитаем длину данных. Это делается так же, как и запись, только наоборот. Теперь мы берем с первого элемента по четвертый и складываем их через логическое или. Со второго элемента делаем сдвиг влево по 8n, где n от 1 до 3.

После получение длины данных создаем массив частот с размеров 256. После этого считываем с заархивированных данных кол-во элементов. Если число 0, то кол-во 256. В цикле записываем частоты символа с заархивированных данных. Символ считается находится по индексу 5+i\*2, потому что символ в файле находится на каждом 6 индексе, а его частота на каждом следующем индексе после символа.

Начальный индекс находится по формуле: ;



После считывания заголовка у нас есть массив частот, поэтому мы можем построить дерево Хаффмана через метод CreateHuffmanTree. После получения дерева вызываем метод Decompress, которая принимает заархивированные данные, начальный индекс, длину данных, дерево и возвращает распакованные данные.



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. Создаем массив data размером длины данных, в него мы будем записывать распакованные данные. В массиве проходим по заархивированным данным с начального индекса до конца массива. 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 массив данных и увеличиваем size на 1. Сбрасываем current до root. В конце метода возвращаем распакованные данные.

Код:

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)

{

var data = File.ReadAllBytes(dataFile);

byte[] arch = ComressBytes(data);

File.WriteAllBytes(archFile, arch);

}

private static byte[] ComressBytes(byte[] data)

{

int[] freqs = CalculateFrequency(data);

byte[] head = CreateHeader(data.Length, freqs);

Node root = CreateHuffmanTree(freqs);

string[] codes = CreateHuffmanCode(root);

byte[] bits = Compress(data, codes);

return [..head, ..bits];

}

private static byte[] Compress(byte[] data, string[] codes)

{

List<byte> bits = [];

byte sum = 0;

byte bit = 1;

foreach (var symbol in data)

{

foreach (var code in codes[symbol])

{

if (code == '1')

{

sum |= bit;

}

if(bit == 128)

{

bits.Add(sum);

bit = 1;

sum = 0;

}

else

{

bit <<= 1;

}

}

}

if(bit > 1) bits.Add(sum);

return [.. bits];

}

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 byte[] CreateHeader(int length, int[] freqs)

{

List<byte> head = [];

head.Add((byte)(length & 255));

head.Add((byte)((length >> 8) & 255));

head.Add((byte)((length >> 16) & 255));

head.Add((byte)((length >> 24) & 255));

int count = 0;

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

if (freqs[i] > 0)

count++;

head.Add((byte) count);

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

{

if (freqs[i] > 0)

{

head.Add((byte)i);

head.Add((byte)freqs[i]);

}

}

return [.. head];

}

private static int[] CalculateFrequency(byte[] data)

{

int[] freqs = new int[256];

foreach (byte b in data)

{

freqs[b]++;

}

NormilizeFreqs();

void NormilizeFreqs()

{

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);

}

}

}

return freqs;

}

public static void DecompressFile(string archFile, string dataFile)

{

var arch = File.ReadAllBytes(archFile);

byte[] data = DecomressBytes(arch);

File.WriteAllBytes(dataFile, data);

}

private static byte[] DecomressBytes(byte[] arch)

{

ParseHeader(arch,

out int dataLength,

out int startIndex,

out int[] freqs);

Node root = CreateHuffmanTree(freqs);

byte[] data = Decomress(arch, startIndex, dataLength, root);

return data;

}

private static byte[] Decomress(byte[] arch, int startIndex, int dataLength, Node root)

{

int size = 0;

Node current = root;

byte[] data = new byte[dataLength];

for (int i = startIndex; i < arch.Length; i++)

{

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

{

if ((arch[i] & bit) == 0)

current = current.left;

else

current = current.right;

if(current.left == null)

{

if(size < dataLength)

data[size++] = current.symbol;

current = root;

}

}

}

return data;

}

private static void ParseHeader(byte[] arch, out int dataLength, out int startIndex, out int[] freqs)

{

dataLength = arch[0] | (arch[1] << 8) | (arch[2] << 16) | (arch[3] << 24);

freqs = new int[256];

int count = arch[4];

if (count == 0) count = 256;

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

{

byte symbol = arch[5 + i \* 2];

freqs[symbol] = arch[5 + i \* 2 + 1];

}

startIndex = 5 + 2 \* count;

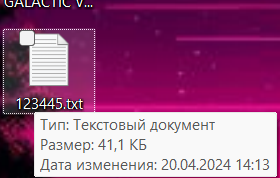
}

}

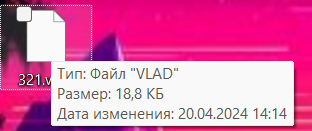
}

Тест:

Файл до сжатия:



После:



Проверка на потерю данных после распаковки:

