Контрольная работа выполнена в www.MatBuro.ru

©МатБюро – Консультации по математике, программированию, экономике, праву, естественным наукам

Поможем вам с написанием программ: www.matburo.ru/sub_subject.php?p=pz

Цели работы

Разработать программу архивации (компрессии) и разархивации (декомпресии) файла любого размера по алгоритмам RLE и LZW.

Листинг программы

Основной файл программы (Program.cs)

```
using System;
using System.IO;

namespace lab4
{
    class Program
    {
        /// <summary>
        /// Расширение файла архива
        /// </summary>
        private const String ENC_FILE_EXT = ".arh";

        /// <summary>
        /// Алгоритм архивации
        /// </summary>
        enum Algo
        {
            RLE, LZW
        }

        /// <summary>
        /// Отображение справки
        /// </summary>
        private static void ShowCmdHelp()
        {
            Console.WriteLine(
                "Использование программы:\n" +
                "lab4.exe [ключ_запуска] [алгоритм] [имя_файла]\n" +
                "Ключ запуска:\n" +
                "-c - архивация\n" +
                "-d - разархивация\n" +
                "-t - тест\n" +
                "Алгоритм:\n" +
                "RLE\n" +
                "LZW\n" +
                "Имя файла - полный или относительный путь к файлу. При разархивации файл должен иметь расширение .arh\n\n" +
                "Пример командной строки:\n" +
                "lab4.exe -c RLE input.txt"
            );
        }

        /// <summary>
        /// Подбор имени архива
        /// </summary>
        /// <param name="decompressedFileName">Имя исходного файла</param>
        /// <returns>Имя архива</returns>
```

www.matburo.ru

```
        private static String GetCompressedFileName(String decompressedFileName)
        {
            // Добавляем к имени исходного файла расширение архива
            return decompressedFileName + ENC_FILE_EXT;
        }

        /// <summary>
        /// Архивация файла
        /// </summary>
        /// <param name="decompressedFileName">Имя исходного файла</param>
        /// <param name="algFactory">Фабрика алгоритмов</param>
        /// <returns>Имя архива</returns>
        private static String Compress(String decompressedFileName, Func<Stream, Stream>
algFactory)
        {
            // Подбираем имя архива
            String compressedFileName = GetCompressedFileName(decompressedFileName);
            // Открываем файлы на чтение/запись, создаем прокси-поток архивации
            using (Stream src = File.OpenRead(decompressedFileName), dst =
File.Create(compressedFileName), enc = algFactory.Invoke(dst))
            {
                // Копируем данные одного файла в другой через прокси-поток архивации
                src.CopyTo(enc);
            }
            return compressedFileName;
        }

        /// <summary>
        /// Подбор имени разархивированного файла
        /// </summary>
        /// <param name="compressedFileName">Имя архива</param>
        /// <returns>Подобранное имя</returns>
        private static String GetDecompressedFileName(String compressedFileName)
        {
            // Убираем расширение .arh
            String decompressedFileName = Path.ChangeExtension(compressedFileName, null);
            // Проверяем существут ли файл с получившимя именем
            if (File.Exists(decompressedFileName))
            {
                // Если существует, будем в цикле добавлять индекс в скобках к имени
файла
                // Начинаем с 1
                int i = 1;
                // Отделяем расширение файла
                String ext = Path.GetExtension(decompressedFileName);
                // И основную часть
                String path = Path.ChangeExtension(decompressedFileName, null);
                do
                {
                    // Строим новое имя с индексом в скобках
                    decompressedFileName = String.Format("{0}({1}){2}", path, i, ext);
                    // Переходим к следующему индексу
                    i++;
                }
                // Выход из цикла, если подобрали имя несуществующего файла
                while (File.Exists(decompressedFileName));
            }
            return decompressedFileName;
```

```
        }

        /// <summary>
        /// Разархивация
        /// </summary>
        /// <param name="compressedFileName">Имя архива</param>
        /// <param name="algFactory">Фабрика алгоритмов</param>
        /// <returns>Имя разархивированного файла</returns>
        private static String Decompress(String compressedFileName, Func<Stream, Stream>
algFactory)
        {
            // Проверяем расширение
            if (!Path.GetExtension(compressedFileName).Equals(ENC_FILE_EXT,
StringComparison.OrdinalIgnoreCase))
                throw new Exception("Сжатый файл должен иметь расширение " +
ENC_FILE_EXT);
            // Подбираем имя разархивированного файла
            String decompressedFileName = GetDecompressedFileName(compressedFileName);
            // Открываем файлы на чтение/запись, создаем прокси-поток разархивации
            using (Stream src = File.OpenRead(compressedFileName), dst =
File.Create(decompressedFileName), enc = algFactory.Invoke(src))
            {
                // Копируем данные одного файла в другой через прокси-поток разархивации
                enc.CopyTo(dst);
            }
            return decompressedFileName;
        }

        /// <summary>
        /// Функция побайтового сравнения файлов
        /// </summary>
        /// <param name="fileName1">Имя файла 1</param>
        /// <param name="fileName2">Имя файла 2</param>
        /// <returns>Позиция первого отличающегося элемента</returns>
        private static int FindFirstDifference(String fileName1, String fileName2)
        {
            // Открываем файлы на чтение
            using (Stream s1 = File.OpenRead(fileName1), s2 = File.OpenRead(fileName2))
            {
                // Выделяем память под буферы
                const int bufSize = 1024;
                byte[] buf1 = new byte[bufSize];
                byte[] buf2 = new byte[bufSize];
                int pos = 0;
                int c1, c2;
                // Читаем в цикле, пока не закончился первый файл
                do
                {
                    // Считываем порцию данных из каждого файла
                    c1 = s1.Read(buf1, 0, buf1.Length);
                    c2 = s2.Read(buf2, 0, buf2.Length);
                    // Вычисляем размер, который присутствует в каждом из буферов
                    int minBuf = Math.Min(c1, c2);
                    // В этой части сравниваем побайтово
                    for (int i = 0; i < minBuf; i++)
                        if (buf1[i] != buf2[i])
                            return pos + i;
                    // Если считался разный размер, файлы отличаются
```

```
            if (c1 != c2)
                return pos + minBuf;
            // Двигаемся дальше
            pos += c1;
        }
        while (c1 > 0);
    }
    // Если дошли до этой точки, файлы одинаковые
    return -1;
}

/// <summary>
/// Главная функция программы
/// </summary>
/// <param name="args"></param>
static void Main(string[] args)
{
    // Проверяем количество аргументов
    if (args.Length != 3)
    {
        // Если не совпадает, выводим справку
        ShowCmdHelp();
        return;
    }
    // Запоминаем имя файла
    String fileName = args[2];
    // Создаем фабрики алгоритмов в зависимости от параметров командной строки
    Func<Stream, Stream> compressAlgo;
    Func<Stream, Stream> decompressAlgo;
    switch (args[1])
    {
        case "RLE":
            compressAlgo = dest => new RLECompressor(dest);
            decompressAlgo = source => new RLEDecompressor(source);
            break;
        case "LZW":
            compressAlgo = dest => new LZWCompressor(dest);
            decompressAlgo = source => new LZWDecompressor(source);
            break;
        default:
            // Неизвестный алгоритм - выводим справку
            ShowCmdHelp();
            return;
    }
    // Выполняем операцию в зависимости от ключа
    switch (args[0])
    {
        // Архивация
        case "-c":
            Compress(fileName, compressAlgo);
            break;
        // Разархивация
        case "-d":
            Decompress(fileName, decompressAlgo);
            break;
        // Тест
        case "-t":
            // Архивируем файл
```

www.matburo.ru

```
                    String compressedFileName = Compress(fileName, compressAlgo);
                    // Разархивируем файл
                    String decompressedFileName = Decompress(compressedFileName,
decompressAlgo);
                    // Сравниваем результат с исходным файлом
                    int p = FindFirstDifference(fileName, decompressedFileName);
                    if (p == -1)
                    {
                        // Если все хорошо, удаляем файлы
                        File.Delete(compressedFileName);
                        File.Delete(decompressedFileName);
                        Console.WriteLine("Исходный и распакованный файл совпадают");
                    }
                    else
                        Console.WriteLine("Исходный и распакованный файл отличаются в
позиции {0}", p);
                    break;
                default:
                    // Неизвестный ключ - выводим справку
                    ShowCmdHelp();
                    return;
            }
        }
    }
}
```

Файл с реализацией алгоритма RLE (RLE.cs)

```
using System;
using System.IO;

namespace lab4
{
    /// <summary>
    /// Класс прокси-потока для записи сжатых алгоритмом RLE данных в поток назначения
    /// </summary>
    class RLECompressor : Stream
    {
        /// <summary>
        /// Вспомогательная переменная для записи в поток назначения
        /// </summary>
        private readonly BinaryWriter writer;
        /// <summary>
        /// Буфер для хранения последовательности неповторяющихся символов
        /// </summary>
        private readonly byte[] buf = new byte[-sbyte.MinValue];
        /// <summary>
        /// Указатель внутри буфера для записи следующего символа
        /// </summary>
        private int bufPtr = 0;
        /// <summary>
        /// Количество повторений последнего символа
        /// </summary>
        private int repeatCnt = 0;

        /// <summary>
        /// Конструктор
```

www.matburo.ru

```
        /// <param name="dest">Конечный поток, в который записываются сжатые
данные</param>
        /// </summary>
        public RLECompressor(Stream dest) => writer = new BinaryWriter(dest);

        /// <summary>
        /// Поток не поддерживает чтение
        /// </summary>
        public override bool CanRead => false;

        /// <summary>
        /// Поток не поддерживает локацию
        /// </summary>
        public override bool CanSeek => false;

        /// <summary>
        /// Поток поддерживает запись
        /// </summary>
        public override bool CanWrite => true;

        /// <summary>
        /// Длина данных в потоке неизвестна
        /// </summary>
        public override long Length => throw new NotImplementedException();

        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Position { get => throw new NotImplementedException(); set
=> throw new NotImplementedException(); }

        /// <summary>
        /// Сброс буфера записи в конечный поток
        /// </summary>
        public override void Flush()
        {
            FlushBuf();
            writer.Flush();
        }

        /// <summary>
        /// Чтение не реализовано
        /// </summary>
        public override int Read(byte[] buffer, int offset, int count)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Seek(long offset, SeekOrigin origin)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Установка размера не реализована
```

www.matburo.ru

```
        /// </summary>
        public override void SetLength(long value)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Закрытие потока. Сбрасывает содержимое буфера в поток назначения
        /// </summary>
        protected override void Dispose(bool disposing)
        {
            base.Dispose(disposing);
            Flush();
        }

        /// <summary>
        /// Запись текущей последовательности в поток назначения
        /// </summary>
        private void FlushBuf()
        {
            // Если у нас повторяющася последовательность:
            while (repeatCnt >= RLEUtil.MIN_REPEAT)
            {
                // Разбиваем ее на куски длиной не более 129 байт (127 влазит в sbyte,
плюс 2 - минимальная длина)
                int chunk = Math.Min(repeatCnt, sbyte.MaxValue + RLEUtil.MIN_REPEAT);
                // Записываем длину последовательности, учитывая, что минимальная длина -
2 байта. Таким образом,
                // длину 2 кодируем значением 0, 3 - значением 1 и т.д.
                writer.Write((sbyte)(chunk - RLEUtil.MIN_REPEAT));
                // Записываем сам повторяющийся символ
                writer.Write(buf[0]);
                // Уменьшаем количество повторений на длину записанного куска
                repeatCnt -= chunk;
            }
            // Если еще что-то осталось (последовательность длины 1):
            if (repeatCnt > 0)
            {
                // Записываем длину последовательности, кодируя ее со знаком "-"
                writer.Write((sbyte) -bufPtr);
                // Записываем саму последовательность
                writer.Write(buf, 0, bufPtr);
                // Обнуляем количество повторений
                repeatCnt = 0;
            }
            // Сбрасываем длину буфера
            bufPtr = 0;
        }

        /// <summary>
        /// Функция записи в поток
        /// </summary>
        /// <param name="buffer">Буфер для записи</param>
        /// <param name="offset">Смещение в буфере</param>
        /// <param name="count">Количество байт для записи</param>
        public override void Write(byte[] buffer, int offset, int count)
        {
            // В цикле записываем данные побайтово
```

```
            for (int i = offset; i < offset + count; i++)
            {
                // Сохраняем байт для записи
                byte b = buffer[i];
                // Если у нас пустой буфер или текущий байт не равен последнему символу
буфера,
                // то мы имеем дело с неповторяющейся последовательностью
                if (bufPtr == 0 || b != buf[bufPtr - 1])
                {
                    // Если мы превысили размер буфера или в буфере была повторяющаяся
последовательность
                    if (bufPtr == buf.Length || repeatCnt >= RLEUtil.MIN_REPEAT)
                        // Сбрасываем буфер
                        FlushBuf();
                    // Записываем в буфер текущий байт
                    buf[bufPtr++] = b;
                    // Устанавливаем число повторений 1, т.е. неповторяющаяся
последовательность
                    repeatCnt = 1;
                }
                // Иначе текущий символ совпал с предыдущим, т.е. началась повторяющаяся
последовательность
                else
                {
                    // Если в буфере была последовательность длины более 1 (т.е.
неповторяющаяся)
                    if (bufPtr > 1)
                    {
                        // Сбросим из буфера все символы, кроме последнего
                        bufPtr--;
                        FlushBuf();
                        // В буфере теперь один символ
                        bufPtr = 1;
                        // И этот символ - текущий
                        buf[0] = b;
                        // А количество повторов - 2: последний символ из буфера и
текущий
                        repeatCnt = 2;
                    }
                    // В буфере последовательность длины 1 (повторяющаяся)
                    else
                        // Увеличим количество повторений
                        repeatCnt++;
                }
            }
        }
    }

    /// <summary>
    ///  Класс прокси-потока для чтения сжатых алгоритмом RLE данных из исходного потока.
    /// </summary>
    class RLEDecompressor : Stream
    {
        /// <summary>
        /// Вспомогательная переменная для чтения из исходного потока
        /// </summary>
        private readonly BinaryReader reader;
        /// <summary>
```

www.matburo.ru

```
        /// Буфер для хранения распакованных данных
        /// </summary>
        private readonly byte[] buf = new byte[-sbyte.MinValue];
        /// <summary>
        /// Указатель текущей позиции буфера
        /// </summary>
        private int bufPtr = 0;
        /// <summary>
        /// Текущий размер буфера
        /// </summary>
        private int bufSize = 0;
        /// <summary>
        /// Признак повторяющейся последовательности в буфере
        /// </summary>
        private bool repeating = false;

        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="source">Исходный поток</param>
        public RLEDecompressor(Stream source) => reader = new BinaryReader(source);

        /// <summary>
        /// Поток поддерживает чтение
        /// </summary>
        public override bool CanRead => true;

        /// <summary>
        /// Поток не поддерживает локацию
        /// </summary>
        public override bool CanSeek => false;

        /// <summary>
        /// Поток не поддерживает запись
        /// </summary>
        public override bool CanWrite => false;

        /// <summary>
        /// Длина данных в потоке неизвестна
        /// </summary>
        public override long Length => throw new NotImplementedException();

        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Position { get => throw new NotImplementedException(); set
=> throw new NotImplementedException(); }

        /// <summary>
        /// Буфер на запись отсутствует
        /// </summary>
        public override void Flush()
        {
        }

        /// <summary>
        /// Функция чтения очередной последовательности в буфер
        /// </summary>
```

www.matburo.ru

```
private void ReadBuf()
{
    // Сбрасываем указатель на начало буфера
    bufPtr = 0;
    sbyte sizeCode;
    try
    {
        // Читаем код длины последовательности
        sizeCode = reader.ReadSByte();
    }
    catch (EndOfStreamException)
    {
        // Если исходный поток кончился, то данные закончились, размер буфера 0
        bufSize = 0;
        return;
    }
    // Код длины положительный - повторяющаяся последовательность
    if (sizeCode >= 0)
    {
        // Декодируем размер последовательности (код + 2)
        bufSize = sizeCode + RLEUtil.MIN_REPEAT;
        // Считываем повторяющийся символ. При возникновении исключения оно будет
        // выброшено наружу, т.к. это будет означать некорректные данные в потоке
        buf[0] = reader.ReadByte();
        // Устанавливаем признак повторяющейся последовательности
        repeating = true;
    }
    // Код длины положительный - неповторяющаяся последовательность
    else
    {
        // Декодируем размер последовательности (-код)
        bufSize = -sizeCode;
        // Считываем последовательность в буфер
        if (reader.Read(buf, 0, bufSize) < bufSize)
            throw new EndOfStreamException();
        // Устанавливаем признак неповторяющейся последовательности
        repeating = false;
    }
}

/// <summary>
/// Функция чтения из потока
/// </summary>
/// <param name="buffer">Буфер для чтения</param>
/// <param name="offset">Смещение в буфере</param>
/// <param name="count">Количество байт для чтения</param>
/// <returns>Возвращает количество прочитанных байт</returns>
public override int Read(byte[] buffer, int offset, int count)
{
    // Устанавливаем указатель в буфере для чтения
    int destPtr = offset;
    // В цикле считываем требуемое количество байт
    while (count > 0)
    {
        // Если указатель в буфере равен его длине, прочитан весь буфер
        if (bufPtr == bufSize)
            // Считаем в буфер следующую последовательность
            ReadBuf();
```

```
                // Если данных больше нет, завершаем чтение
                if (bufPtr == bufSize)
                    break;
                // Определим размер следующей порции данных: минимум из того, сколько
осталось
                // прочитать, и остатка буфера
                int chunk = Math.Min(count, bufSize - bufPtr);
                // Если последовательность повторяющаяся
                if (repeating) {
                    // Запишем в цикле повторяющийся символ требуемое количество раз
                    for (int i = 0; i < chunk; i++)
                        buffer[destPtr++] = buf[0];
                    // При этом указатель в буфере у нас остается на месте, а мы
уменьшаем
                    // "размер" буфера, который в этом случае служит для хранения числа
повторений
                    bufSize -= chunk;
                }
                // Если последовательность неповторяющаяся
                else
                {
                    // Скопируем из буфера последовательности нужное количество байт
данных
                    Array.Copy(buf, bufPtr, buffer, destPtr, chunk);
                    // Продвинем указатель в буфере последовательности
                    bufPtr += chunk;
                    // Продвинем указатель в буфере чтения
                    destPtr += chunk;
                }
                // Уменьшим требуемое количество данных на длину прочитанной порции
                count -= chunk;
            }
            // Количество прочитанных байт - разница текущего указателя в буфере с его
            // исходным значением
            return destPtr - offset;
        }

        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Seek(long offset, SeekOrigin origin)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Установка размера не реализована
        /// </summary>
        public override void SetLength(long value)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Запись не реализована
        /// </summary>
        public override void Write(byte[] buffer, int offset, int count)
        {
```

```
            throw new NotImplementedException();
        }
    }

    /// <summary>
    /// Вспомогательный класс для алгоритма RLE
    /// </summary>
    class RLEUtil
    {
        /// <summary>
        /// Минимальная длина последовательсти повторяющихся символов.
        /// Длина 0 не нужна, длина 1 кодируется как неповторяющаяся последовательность.
        /// </summary>
        public const int MIN_REPEAT = 2;
    }
}
```

Файл с реализацией алгоритма LZW (LZW.cs)

```
using System;
using System.Collections.Generic;
using System.Diagnostics.CodeAnalysis;
using System.IO;

namespace lab4
{
    /// <summary>
    /// Класс прокси-потока для записи сжатых алгоритмом LZW данных в поток назначения
    /// </summary>
    class LZWCompressor : Stream
    {
        /// <summary>
        /// Вспомогательная переменная для записи в поток назначения
        /// </summary>
        private readonly BinaryWriter writer;
        /// <summary>
        /// Словарь
        /// </summary>
        private readonly Dictionary<byte[], uint> dictionary = new Dictionary<byte[],
uint>(new ByteArrayComparer());
        /// <summary>
        /// Текущее количество бит, необходимых для кодирования слова, увеличивающееся с
наполнением словаря
        /// </summary>
        private int wordBits;
        /// <summary>
        /// Битовая маска для выделения кода слова из uint
        /// </summary>
        private uint wordMask;
        /// <summary>
        /// Количестве использованных бит в буфере записи
        /// </summary>
        private int bufBits = 0;
        /// <summary>
        /// Буфер записи
        /// </summary>
        private uint buf = 0;
        /// <summary>
```

```
        /// Текущее слово
        /// </summary>
        private byte[] word;

        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="dest">Конечный поток, в который записываются сжатые
данные</param>
        public LZWCompressor(Stream dest)
        {
            writer = new BinaryWriter(dest);
            // Записываем в словарь все 256 возможных значений байта
            byte b = byte.MinValue;
            do
            {
                dictionary.Add(new byte[] { b }, b);
                b++;
            }
            while (b != byte.MinValue);
            // Устанавливаем начальный размер кода в битах. Он увеличивается при занятии
максимального
            // значения, представимого текущим количеством бит. Так как значение для 8
бит (255) занято,
            // берем 9
            setWordBits(9);
        }

        /// <summary>
        /// Функция устанавливаем текущий размер кода в битах и обновляет маску
        /// </summary>
        /// <param name="bits"></param>
        private void setWordBits(int bits)
        {
            wordBits = bits;
            wordMask = LZWUtil.calcBitMask(bits);
        }

        /// <summary>
        /// Поток не поддерживает чтение
        /// </summary>
        public override bool CanRead => false;

        /// <summary>
        /// Поток не поддерживает локацию
        /// </summary>
        public override bool CanSeek => false;

        /// <summary>
        /// Поток поддерживает запись
        /// </summary>
        public override bool CanWrite => true;

        /// <summary>
        /// Длина данных в потоке неизвестна
        /// </summary>
        public override long Length => throw new NotImplementedException();
```

www.matburo.ru

```
        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Position { get => throw new NotImplementedException(); set
=> throw new NotImplementedException(); }

        /// <summary>
        /// Сброс буфера невозможен, так как минимальная единица записи - байт, а у нас
данные записываются битами,
        /// т.е. сброс буфера повредит выходной поток
        /// </summary>
        public override void Flush()
        {
        }

        /// <summary>
        /// Чтение не реализовано
        /// </summary>
        public override int Read(byte[] buffer, int offset, int count)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Seek(long offset, SeekOrigin origin)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Установка размера не реализована
        /// </summary>
        public override void SetLength(long value)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Запись кода словаря в поток назначения
        /// </summary>
        /// <param name="code">Код</param>
        private void WriteCode(uint code)
        {
            // Несуществующий код не пишем
            if (code == LZWUtil.NULL_CODE)
                return;
            // Добавляем биты кода в буфер
            buf |= code << bufBits;
            // Вычисляем новое количество бит в буфере
            int newBufBits = bufBits + wordBits;
            // Если произошло переполнение буфера
            if (newBufBits >= 32)
            {
                // Сбрасываем буфер в поток
                writer.Write(buf);
                // Записываем в буфер оставшиеся биты
```

```
                buf = code >> 32 - bufBits;
                // Устанавливаем новый размер буфера
                bufBits = newBufBits - 32;
            }
            // Переполнения не было
            else
                // Устанавливаем новый размер буфера
                bufBits = newBufBits;
        }

        /// <summary>
        /// Проверка наличия слова в словаре и добавление при отсутствии
        /// </summary>
        /// <param name="word"></param>
        /// <returns></returns>
        private bool CheckAddNewWord(byte[] word)
        {
            // Если слово отсутствует
            if (!dictionary.ContainsKey(word))
            {
                // Добавляем слово
                dictionary.Add(word, (uint)dictionary.Count);
                // Проверяем, требуется ли увеличить длину кода
                if (dictionary.Count - 1 > wordMask)
                    // Увеличиваем длину кода
                    setWordBits(wordBits + 1);
                return true;
            }
            return false;
        }

        /// <summary>
        /// Запись байта
        /// </summary>
        /// <param name="b"></param>
        private void WriteByte(byte b)
        {
            // Код текущего слова
            uint code;
            // Новое слово
            byte[] newWord;
            // Если слово пусто (при записи в поток первого байта)
            if (word == null)
            {
                // Код слова пустой
                code = LZWUtil.NULL_CODE;
                // Новое слово состоит из добавляемого байта
                newWord = new byte[] { b };
            }
            // Если слово не пусто
            else
            {
                // Находим код слова в словаре
                code = dictionary.GetValueOrDefault(word, LZWUtil.NULL_CODE);
                // Новое слово получаем путем добавления к текущему слову записываемого
байта
                newWord = LZWUtil.AppendByte(word, b);
            }
```


www.matburo.ru

```
            // Проверяем, надо ли добавить новое слово в словарь
            if (CheckAddNewWord(newWord))
            {
                // Если надо - записываем в выходной поток код текущего слово
                WriteCode(code);
                // Текущее слово теперь состоит из добавляемого байта
                word = new byte[] { b };
            }
            // Если новое слово уже есть в словаре
            else
                // Новое слово становится текущим словом
                word = newWord;
        }

        /// <summary>
        /// Функция записи в поток
        /// </summary>
        /// <param name="buffer">Буфер для записи</param>
        /// <param name="offset">Смещение в буфере</param>
        /// <param name="count">Количество байт для записи</param>
        public override void Write(byte[] buffer, int offset, int count)
        {
            // Записываем в выходной поток по одному байту
            for (int i = offset; i < offset + count; i++)
                WriteByte(buffer[i]);
        }

        /// <summary>
        /// Закрытие потока. Сбрасывает содержимое буфера в поток назначения
        /// </summary>
        protected override void Dispose(bool disposing)
        {
            base.Dispose(disposing);
            // Записываем в поток назначения код текущего слова
            WriteCode(word == null ? LZWUtil.NULL_CODE :
dictionary.GetValueOrDefault(word, LZWUtil.NULL_CODE));
            // Записывае в поток назначения значение буфера. Последние биты в файле не
будут
            // использованы, но так как минимальная длина кода равна 9, а количество
неиспользованных
            // бит в последнем байте не превышает 7, мы сможем использовать признак конца
файла как
            // окончание потока при расжатии
            while (bufBits > 0)
            {
                // Записываем первый байт кода в поток назначения
                writer.Write((byte)buf);
                // Сдвигаем буфер на один байт вправо
                buf >>= 8;
                // Уменьшаем количество бит в буфере
                bufBits -= 8;
            }
            bufBits = 0;
        }
    }

    /// <summary>
    ///  Класс прокси-потока для чтения сжатых алгоритмом RLE данных из исходного потока.
```

```
    /// </summary>
    class LZWDecompressor : Stream
    {
        /// <summary>
        /// Вспомогательная переменная для чтения из исходного потока
        /// </summary>
        private readonly BinaryReader reader;
        /// <summary>
        /// Словарь
        /// </summary>
        private readonly List<byte[]> dictionary;
        /// <summary>
        /// Текущее количество бит, необходимых для кодирования слова, увеличивающееся с
наполнением словаря
        /// </summary>
        private int wordBits;
        /// <summary>
        /// Битовая маска для выделения кода слова из uint
        /// </summary>
        private uint wordMask;
        /// <summary>
        /// Буфер, содержащий последнее раскодированное слово
        /// </summary>
        private byte[] buf;
        /// <summary>
        /// Указатель внутри буфера на последний использованный байт слова
        /// </summary>
        private int bufPtr;
        /// <summary>
        /// Буфер кодов, читаемых из исходного потока
        /// </summary>
        private uint codeBuf = 0;
        /// <summary>
        /// Количество задействованных бит в буфере кодов
        /// </summary>
        private int codeBufBits = 0;

        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="source">Исходный поток</param>
        public LZWDecompressor(Stream source)
        {
            reader = new BinaryReader(source);
            // Создаем словарь и записываем в него все 256 возможных значений байта
            dictionary = new List<byte[]>();
            byte b = byte.MinValue;
            do
            {
                dictionary.Add(new byte[] { b });
                b++;
            }
            while (b != byte.MinValue);
            // Как и в случае с потоком сжатия, изначально длина кода 9 бит
            setWordBits(9);
        }

        /// <summary>
```

www.matburo.ru

```
        /// Функция устанавливаем текущий размер кода в битах и обновляет маску
        /// </summary>
        /// <param name="bits"></param>
        private void setWordBits(int bits)
        {
            wordBits = bits;
            wordMask = LZWUtil.calcBitMask(bits);
        }

        /// <summary>
        /// Поток поддерживает чтение
        /// </summary>
        public override bool CanRead => true;

        /// <summary>
        /// Поток не поддерживает локацию
        /// </summary>
        public override bool CanSeek => false;

        /// <summary>
        /// Поток не поддерживает запись
        /// </summary>
        public override bool CanWrite => false;

        /// <summary>
        /// Длина данных в потоке неизвестна
        /// </summary>
        public override long Length => throw new NotImplementedException();

        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Position { get => throw new NotImplementedException(); set
=> throw new NotImplementedException(); }

        /// <summary>
        /// Буфер на запись отсутствует
        /// </summary>
        public override void Flush()
        {
        }

        /// <summary>
        /// Чтение следующего кода из исходного потока
        /// </summary>
        /// <returns>Возвращает код слова</returns>
        private uint ReadCode()
        {
            // В переменную b будет побайтово считывать данные из входного потока
            uint b = 0;
            // Читаем, пока не получили требуемое количество бит
            while (codeBufBits < wordBits)
            {
                try
                {
                    // Считываем следующий байт
                    b = reader.ReadByte();
                }
```


www.matburo.ru

```
            catch (EndOfStreamException)
            {
                // Конец потока означает, что данные закончились, вернем пустой код
                return LZWUtil.NULL_CODE;
            }
            // Добавим к буферу считанное значение
            codeBuf |= b << codeBufBits;
            // Увеличим размер буфера
            codeBufBits += 8;
        }
        // Теперь, когда буфер содержит нужное количество бит, выделим код по маске
        uint res = codeBuf & wordMask;
        // Сдвинем буфер вправо на взятое количество бит
        codeBuf >>= wordBits;
        // Если в процессе произошло переполнение буфера
        if (codeBufBits > 32)
        {
            // Восстановим потерянные биты из последнего считанного байта
            codeBuf |= b >> (codeBufBits - 32) << (32 - wordBits);
        }
        // Пометим, что из буфера взяли код
        codeBufBits -= wordBits;
        return res;
    }

    /// <summary>
    /// Добавление слова в словарь
    /// </summary>
    /// <param name="word">Слово</param>
    private void AddWord(byte[] word)
    {
        // Добавляем слово
        dictionary.Add(word);
        // Увеличиваем длину кода при заполнении словаря
        if (dictionary.Count + 1 > wordMask)
            setWordBits(wordBits + 1);
    }

    /// <summary>
    /// Функция чтения из потока
    /// </summary>
    /// <param name="buffer">Буфер для чтения</param>
    /// <param name="offset">Смещение в буфере</param>
    /// <param name="count">Количество байт для чтения</param>
    /// <returns>Возвращает количество прочитанных байт</returns>
    public override int Read(byte[] buffer, int offset, int count)
    {
        // Устанавливаем указатель в буфере для чтения
        int destPtr = offset;
        // В цикле считываем требуемое количество байт
        while (count > 0)
        {
            // Если буфер пуст (до считывания первого кода) или полностью использован
            if (buf == null || bufPtr >= buf.Length)
            {
                // Читаем следующий код
                uint code = ReadCode();
                // Если код пустой, данных больше нет, выходим из цикла
```

```
                    if (code == LZWUtil.NULL_CODE)
                        break;
                    byte[] next;
                    // Если кода нет в словаре
                    if (code >= dictionary.Count)
                    {
                        // Следующее слово равно конкатенации предыдущего с его первым
байтом
                        next = LZWUtil.AppendByte(buf, buf[0]);
                        // Добавляем слово в словарь
                        AddWord(next);
                    }
                    // Если код есть в словаре
                    else
                    {
                        // Следующее слово равно конкатенации предыдущего с первым байтом
нового слова
                        next = dictionary[(int)code];
                        if (buf != null)
                        {
                            byte[] w = LZWUtil.AppendByte(buf, next[0]);
                            AddWord(w);
                        }
                    }
                    // Записываем в буфер следующее слово
                    buf = next;
                    // Устанавливаем указатель буфера на начало слова
                    bufPtr = 0;
                }
                // Записываем в буфер чтения очередной байт слова и продвигаем указатели
обоих буферов
                buffer[destPtr++] = buf[bufPtr++];
                // Уменьшаем количество байт для чтения
                count--;
            }
            // Количество прочитанных байт - разница текущего указателя в буфере с его
            // исходным значением
            return destPtr - offset;
        }

        /// <summary>
        /// Локация не реализована
        /// </summary>
        public override long Seek(long offset, SeekOrigin origin)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Установка размера не реализована
        /// </summary>
        public override void SetLength(long value)
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// Запись не реализована
```


www.matburo.ru

```
        /// </summary>
        public override void Write(byte[] buffer, int offset, int count)
        {
            throw new NotImplementedException();
        }
    }

    /// <summary>
    /// Вспомогательный класс для алгоритма LZW
    /// </summary>
    class LZWUtil
    {
        /// <summary>
        /// Специальное значение кода, означающее несуществующий код
        /// </summary>
        public const uint NULL_CODE = uint.MaxValue;

        /// <summary>
        /// Добавление байта к массиву байт
        /// </summary>
        /// <param name="to">Массив</param>
        /// <param name="b">Байт</param>
        /// <returns>Результат добавления</returns>
        public static byte[] AppendByte(byte[] to, byte b)
        {
            byte[] res = new byte[to.Length + 1];
            Array.Copy(to, res, to.Length);
            res[to.Length] = b;
            return res;
        }

        /// <summary>
        /// Вычисление битовой маски для заданного числа бит
        /// </summary>
        /// <param name="bits">Число бит</param>
        /// <returns>Битовая маска</returns>
        public static uint calcBitMask(int bits)
        {
            return ((uint)1 << bits) - 1;
        }
    }

    /// <summary>
    /// Вспомогательный класс сравнения байтовых массивов
    /// для их использования в качестве ключа словаря
    /// </summary>
    class ByteArrayComparer : IEqualityComparer<byte[]>
    {
        private const uint V = 0x811c9dc5;

        /// <summary>
        /// Сравнение байтовых массивов
        /// </summary>
        /// <param name="x">Массив 1</param>
        /// <param name="y">Массив 2</param>
        /// <returns></returns>
        public bool Equals([AllowNull] byte[] x, [AllowNull] byte[] y)
        {
```

www.matburo.ru

```
            // Проверяем, что массивы не пусты, и их длины равны
            if (x == null || y == null || x.Length != y.Length)
                return false;
            // Сравниваем массивы равной длины побайтово
            for (int i = 0; i < x.Length; i++)
                // Массивы не равны, если отличаются хотя бы в одном байте
                if (x[i] != y[i])
                    return false;
            // Если все совпало - равны
            return true;
        }

        /// <summary>
        /// Функция расчета хэша массива байтов по алгоритму Adler32
        /// https://ru.wikipedia.org/wiki/Adler-32
        /// </summary>
        /// <param name="obj"></param>
        /// <returns></returns>
        public int GetHashCode([DisallowNull] byte[] obj)
        {

            uint s1 = 1;
            uint s2 = 0;
            foreach (byte b in obj)
            {
                s1 = (s1 + b) % 65521;
                s2 = (s2 + s1) % 65521;
            }
            return (int)((s2 << 16) + s1);
        }
    }
}
```

Тестирование

Взаимодействие с программой осуществляется через параметры командной строки с выводом отладочных сообщений на консоль. При запуске программы без параметров отображается справка с перечислением требуемых параметров и их возможных значений (рис. 1).

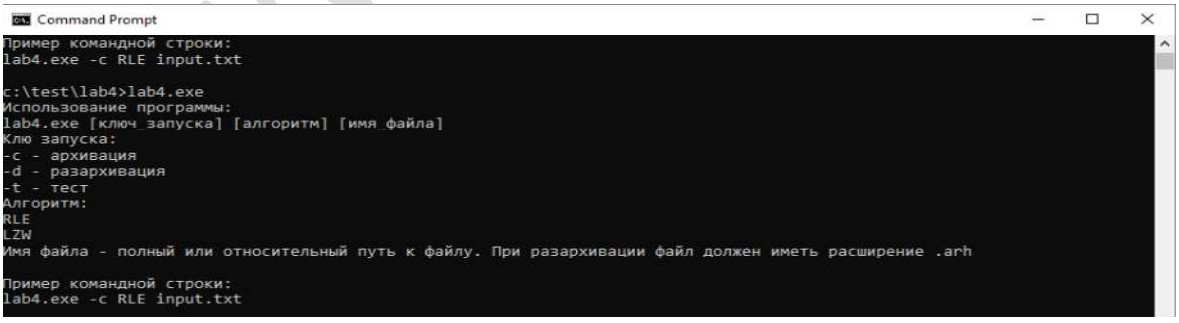

Рис. 1

Для облегчения тестирования помимо реализации функции архивации (ключ -c) и разархивации (ключ -d) была добавления функция проверки соответствия файла, восстановленного из архива, исходному файлу (ключ -t).

Контрольная работа выполнена в www.MatBuro.ru

©МатБюро – Консультации по математике, программированию, экономике, праву, естественным наукам

Поможем вам с написанием программ: www.matburo.ru/sub_subject.php?p=pz

Проведенное тестирование показало, что для всех входных данных разархивирование архива приводит к созданию файла, идентичного исходного для обоих алгоритмов, т.е. программа работает корректно. Также алгоритм RLE показал более высокий коэффициент сжатия для повторяющихся последовательностей из одного символа, в то время как LZW лучше показал себя на более длинных повторяющихся послевательностях.

www.matburo.ru