Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу «Дискретный анализ»: Методы сжатия данных

Студент: А. М. Титеев
Преподаватель: Н. А. Зацепин
Группа: М8О-408Б
Дата:
Оценка:
Подпись:

Москва, 2020

Условие

Необходимо реализовать два известных метода сжатия данных для сжатия одного файла.

Формат запуска должен быть аналогичен формату запуска программы gzip, должны быть поддержаны следующие ключи: c, d, k, l, r, t, 1, 9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Метод решения

Как и требуется в условии запуск программы аналогичен запуску утилиты gzip: ./main <ключи> <файлы> <ключи> <файлы> ...

Препроцессинг

На первом этапе работы программа определяет наличие в поступившей строке ключей, директорий и файлов. При обработке ключей учитывается их взаимоперекрываение, как в утилите gzip: l и r имеют наибольший приоритет, далее идёт ключ t, после чего остальные. В случае, если новый ключ перекрывает по логике утилиты некоторые из уже имеющихся, то эти ключи деактивируются.

Если полученное слово из стандартного ввода не является ключом, то программа проверяет наличие директории с таким именем. Если такой директории нет, то считается, что это имя файла, и оно заносится в список файлов. Если директория с таким именем существует и подключён ключ r, то все файлы внутри этой директории добавляются в список.

После с файлами ведётся работа согласно введённым ключам.

Арифметический алгоритм

В этой программе реализованная целочисленная арифметика. Причина проста - машинам лучше и проще работать с целыми числами, чем с числами с плавающими точками. Например, наличие машинного эпсилон.

Перед объяснением алгоритма стоит упомянуть ряд моментов. Во-первых, так как мы имеем дело с целыми числами, это значит, что у нас нет возможности бесконечно переходить к меньшим отрезкам. Поэтому в данной реализации работа с отрезками сводится к их масштабированию, параллельно записывая нужные биты. Во-вторых, для работы внутри рассматриваемого отрезка используется таблица частот, которая содержит в себе информацию о том, насколько часто встречается тот или иной символ. Это необходимо для нужного разбиения на ещё более мелкие отрезки.

Данная реализация делится на несколько фаз:

- Подготовительная фаза - производится инициализация буфера, куда будут записываться биты, счётчиков, начального отрезка и таблицы частот. Отдельно стоит упомянуть, что начальные значения частот у каждого символа изначально стоят

как единицы. Это необходимо корректной работы с отрезком. Помимо этого также в файл заносится служебная информация, где указано, каким алгоритмом сжали и какой был размер исходного файла.

- Основная фаза - производится кодировка символов. Каждому символу в начале сопоставляется соответствующий маленький отрезок, с которым в дальнейшем происходит масштабирование и смещение, дабы избежать переполнения. В процессе заносятся соответствующие биты. После того, как дальнейшее масштабирование становится невозможным, кодировка символа завершается. Обновляется таблица частот в соответствии с символом.

- Завершающая фаза - когда символы закодированы, программа кодирует EOF и дозаписывает недостающие биты. На этом работа с файлом завершается.

В процессе декомпрессии алгоритм будет часто обращаться к таблице частот, поэтому надо максимально сильно сократить работу с ней, чтобы алгоритм не просел по времени. Для этого есть довольно простое решение - в процессе декомпрессии во время обновления таблицы будет производиться её небольшая сортировка. Сортировка сводится к тому, чтобы часто попадающиеся символы были ближе к началу таблицы. Это и ускоряет декомпрессию.

Аналогично компрессии, декомпрессию можно разделить на несколько фаз:

- Подготовительная фаза - почти аналогично компрессии, но с некоторыми отличиями. Инициализируется специальная переменная, которая будет содержать в себе биты, которые были в сжатом файле. Также в файл ничего не записывается, а производится лишь чтение.

- Основная фаза - при помощи специальной переменной определяем символ. Проводим такие же манипуляции с отрезком, как и в сжатии, параллельно загружая новые биты из файла и отбрасывая уже лишние. После этого аналогично компрессии обновляется таблица частот.

- Завершающая фаза - после получения кода об EOF декомпрессия завершается. Весь результат декомпрессии по ходу заносится в новый файл.

LZ77

Описание файлов программы

Код программы разбит на 9 файлов:

1. Arithmetic.h - Описывает класс Arithmetic, в котором заключён соответствующий алгоритм кодирования.

2. Arithmetic.cpp - Реализует класс Arithmetic.

3. Globals.h - Файл с общими библиотеками.
4. LZ77.h - Описывает класс LZ77, в котором заключён соответствующий алгоритм кодирования.
5. LZ77.cpp - Реализует класс LZ77.
6. main.cpp - Основной файл, отвечающий за чтение входных данных и принятие действий относительно каждого файла.
7. Makefile - Файл для сборки программы.
8. preprocessing.h - Содержит прототипы функций, необходимых для обработки данных перед компрессией/декомпрессией.
9. preprocessing.cpp - Реализует все функции из соответствующей библиотеки.

Основные типы данных

1. Arithmetic - Реализует соответствующий алгоритм.
2. LZ77 - Реализует соответствующий алгоритм.

Описание методов и функций программы

Основные свойства и методы класса Arithmetic

public:

1. bool Compress (const char*, const char*) - сжатие файла.
2. bool Decompress (const char*, const char*) - распаковка файла.
3. Arithmetic() - конструктор, в котором задаются начальные значения для последующей работы со сжатием/распаковкой файла.

private:

1. bool chError - флаг ошибки при распаковке файла.
2. unsigned char indexToChar [NO_OF_SYMBOLS] - таблица перевода из индексов к символам.
3. int charToIndex [NO_OF_CHARS] - таблица перевода из символов в индексы.
4. int cumFreq [NO_OF_SYMBOLS + 1] - массив накопленных частот. Нужен для определения границ.

5. int freq [NO_OF_SYMBOLS + 1] - массив частот. В нём хранится число появлений тех или иных символов.
6. long low - нижняя граница отрезка.
7. long high - верхняя граница отрезка.
8. long value - число, которое лежит в отрезке.
9. long bitsToFollow - количество бит, которые надо пустить в след за следующим выставляемым битом.
10. int buffer - буффер для работы с файлом.
11. int bitsToGo - число битов, которые ещё можно загрузить в буфер.
12. int garbageBits - счётчик плохих битов при распаковке файла. Как только их становится слишком много - распаковка отменяется и выводится сообщение об этом.
13. FILE *out - файл, в который мы записываем.
14. FILE *in - файл, из которого мы считываем.
15. void UpdateModel (int) - обновление модели под новый символ.
16. void StartInputingBits() - подготовка к побитовому вводу.
17. void StartOutputingBits() - подготовка к побитовому выводу.
18. void EncodeSymbol (int) - кодировка символа.
19. void StartEncoding() - подготовка к сжатию.
20. void DoneEncoding() - завершение кодирования. Загрузка последних битов в буфер.
21. void StartDecoding() - подготовка к распаковке.
22. int DecodeSymbol() - распаковка символа.
23. int InputBit() - получение одного бита из файла.
24. void OutputBit(int) - отправление одного бита в файл.
25. void DoneOutputingBits() - отправление последних битов в файл.
26. void OutputBitPlusFollow(int) - вывод указанного бита и отложенных ранее.

Основные свойства и методы класса TLZ77

public:

1. LZ77() - стандартный конструктор.
2. LZ77(IStruct s) - конструктор через вспомогательную структуру IStruct.
3. InitEncode() - инициализирует данные необходимые для сжатия.
4. Compress(std::string in_str, std::string out_str) - сжатие файла.
5. Decompress(std::string in_str, std::string out_str) - распаковка файла.
6. ∼LZ77() - деструктор.

private:

1. LoadDict(unsigned int dictpos) - загрузка словаря из файла в циклический буфер на позиции dictpos.
2. DeleteData(unsigned int dictpos) - удаления всех ссылок на удаляемый сектор с началом в dictpos.
3. HashData(unsigned int dictpos, unsigned int bytestodo) - хэширование и запись ссылок на возможное преведущее совпадение в словаре.
4. FindMatch(unsigned int dictpos, unsigned int startlen) - поиск максимального совпадения в словаре с позицией dictpos, не меньше чем startlen.
5. DictSearch(unsigned int dictpos, unsigned int bytestodo) - кодирование считанного сектора с началом в dictpos и длинной bytestodo.
6. SendChar(unsigned int character) - кодирование символа character.
7. SendMatch(unsigned int matchlen, unsigned int matchdistance) - кодирование пары <matchlen, matchdistance>.
8. ReadBits(unsigned int numbits) - считывание numbits битов из файла.
9. SendBits(unsigned int bits, unsigned int numbits) - отправка numbits битов записанных в bits в файл.
10. const int compressFloor - минимальное совпадение, для записи в виде<length,offset>.
11. const int comparesCeil - максимальное число раз которое ищется совпадение в FindMatch.
12. const int CHARBITS - сколькими битами кодируется символ.

13. const int MATCHBITS - сколькими битами кодируется длина совпадения.
14. const int DICTBITS - сколькими битами кодируется длина словаря(offset).
15. const int HASHBITS - сколько бит в хэше.
16. const int SECTORBITS - сколько бит в секторе.
17. const unsigned int MAXMATCH - максимальная кодируемая длина совпадения.
18. const unsigned int DICTSIZE - размер словаря.
19. const unsigned int HASHSIZE - размер хэша.
20. const unsigned int SHIFTBITS - на сколько происходит сдвиг при хэшировании.
21. const unsigned int SECTORLEN - размер сектора.
22. const unsigned int SECTORAND - нужен для определения к какому сектору относится то или иное место в словаре.
23. unsigned char* dict - ссылка на словарь размером DICTSIZE.
24. unsigned int *hash - ссылка на хэш размером HASHSIZE.
25. unsigned int *nextlink - ссылка на массив, на каждой позиции которого хранится позиция предыдущего вхождения подстроки с совпадающим хэшем.
26. unsigned int matchlength - длина совпадения, применяется в FindMatch, DictSearch.
27. unsigned int matchpos - позиция совпадения, применяется там же.
28. unsigned int bitbuf - буфер, который используется для записи и чтения бит из файла.
29. unsigned int bitsin - сколько битов находится в буфере в данный момент.
30. unsigned int masks[17] - маски для побитового чтения/записи.
31. FILE *infile, *outfile; - файлы из которых идёт считывание/запись.

Прочие функции

1. void Compress(std::string) - Создает необходимые для компрессии данные (временные файлы).
2. void Decompress(std::string) - Создает необходимые для декомпрессии данные (классы и временные файлы).

3. bool ActivateKeys(std::string) - Активирует или деактивирует указанные в аргументах ключи. Из-за некорректного ключа функция возвращает false и завершает работу программы.

4. void ArchiveInfo(std::string) - Выводит информацию об архиве.

5. unsigned long long int CompressA(std::string) - Создаёт класс для арифметического кодирования и работает с ним. При некорректной работе алгоритма возвращает 0, иначе количество байт в получившемся файле.

6. unsigned long long int CompressL(std::string) - Создаёт класс для кодирования LZ77 и работает с ним. При некорректной работе алгоритма возвращает 0, иначе количество байт в получившемся файле.

7. bool ArchiveCheck(std::string) - Проверка на наличие у файла суффикса «.gz».

8. bool DirectoryCheck(std::string, bool) - Проверка на работоспособность директории.

9. void GetFiles(std::string, std::map<std::string, int>*) - Записывает все имена файлов директории в отдельное красно-чёрное дерево.

10. void SaveBest(std::string, unsigned long long int, unsigned long long int) - При отсутствии ключей 1 и 9 выбирает наименьший из полученных компрессией файлов и удаляет наибольший.

11. void ShowErrors(std::string) - Вывод сообщений о возникших ошибках во время подготовки данных для алгоритмов.

12. void Mv(std::string, std::string) - Выполняет команду mv для переименования временного файла.

13. void Rm(std::string) - Выполняет команду rm для удаления файла получившегося в процессе компрессии/декомпрессии и содержащего битые данные из-за ошибки алгоритма или удаляет поступивший алгоритму файл в случае отсутствия ключа k.

Исходный код

Arithmetic.cpp

```
#include "Arithmetic.h"

Arithmetic::Arithmetic ()
{

    int i;
```

```
    for ( i = 0; i < NO_OF_CHARS; ++i)
    {
        charToIndex [i] = i + 1;
        indexToChar [i + 1] = i;
    }

    for ( i = 0; i <= NO_OF_SYMBOLS; ++i)
    {
        freq [i] = 1;
        cumFreq [i] = NO_OF_SYMBOLS - i;
    }

    in = nullptr;
    out = nullptr;
    chError = 0;
    freq [0] = 0;
}
void Arithmetic::UpdateModel (int symbol)
{

    int i;
    int chI, chSymbol;
    int cum;

    if (cumFreq [0] == MAX_FREQUENCY)
    {
        cum = 0;
        for ( i = NO_OF_SYMBOLS; i >= 0; --i)
        {
            freq [i] = (freq [i] + 1) / 2;
            cumFreq [i] = cum;
            cum += freq [i];
        }
    }

    for (i = symbol; freq [i] == freq [i - 1]; --i);

    if (i < symbol)
    {
        chI                         = indexToChar [i];
        chSymbol                    = indexToChar [symbol];
```

```
            indexToChar [i]             = chSymbol;
            indexToChar [symbol]        = chI;
            charToIndex [chI]           = symbol;
            charToIndex [chSymbol]      = i;
        }

        freq [i] += 1;
        while (i > 0)
        {
            --i;
            cumFreq [i] += 1;
        }
}
void Arithmetic::StartInputingBits ()
{
    bitsToGo = 0;
    garbageBits = 0;
}
int Arithmetic::InputBit ()
{

    int t;

    if (bitsToGo == 0)
    {
        buffer = getc (in);
        if (buffer == EOF)
        {
            ++garbageBits;
            if (garbageBits > BITS_IN_REGISTER - 2)
            {
                printf ("ERROR: Incorrect compress file!\n");
                chError = true;
                return 0;
            }
        }
        bitsToGo = 8;
    }

    t = buffer & 1;
    buffer >>= 1;
    --bitsToGo;
```

```
    return t;
}
void Arithmetic::StartOutputingBits ()
{
    buffer = 0;
    bitsToGo = 8;
}
void Arithmetic::OutputBit (int bit)
{

    buffer >>= 1;

    if (bit) {
        buffer |= 0x80;
    }

    --bitsToGo;

    if (bitsToGo == 0)
    {
        if(keys[0])
        {
            std::cout << buffer;
        }
        else
        {
            putc(buffer, out);
        }
        bitsToGo = 8;
    }
}
void Arithmetic::DoneOutputingBits ()
{
    if(keys[0])
    {
        std::cout << (buffer >> bitsToGo);
    }
    else
    {
        putc(buffer >> bitsToGo, out);
    }
```

```
}
void Arithmetic::OutputBitPlusFollow (int bit)
{
    OutputBit (bit);
    while (bitsToFollow > 0)
    {
        OutputBit (!bit);
        --bitsToFollow;
    }
}
void Arithmetic::StartEncoding ()
{
    low             = 0l;
    high            = TOP_VALUE;
    bitsToFollow    = 0l;
}
void Arithmetic::DoneEncoding ()
{

    ++bitsToFollow;
    if (low < FIRST_QTR)
    {
        OutputBitPlusFollow(0);
    }
    else
    {
        OutputBitPlusFollow(1);
    }
}
void Arithmetic::StartDecoding ()
{

    value = 0l;
    for ( int i = 0; i < BITS_IN_REGISTER; ++i)
    {
        value = 2 * value + InputBit ();
    }
    low = 0l;
    high = TOP_VALUE;
}
void Arithmetic::EncodeSymbol (int symbol)
{
```

```
    long range;

    range = (long) (high - low) + 1;
    high  = low + (range * cumFreq [symbol - 1]) / cumFreq [0]
       - 1;
    low   = low + (range * cumFreq [symbol]    ) / cumFreq [0];
    for (;;)
    {
        if (high < HALF)
        {
            OutputBitPlusFollow (0);
        }
        else if (low >= HALF)
        {
            OutputBitPlusFollow (1);
            low -= HALF;
            high -= HALF;
        }
        else if (low >= FIRST_QTR && high < THIRD_QTR)
        {
            ++bitsToFollow;
            low -= FIRST_QTR;
            high -= FIRST_QTR;
        }
        else
            break;

        low = 2 * low;
        high = 2 * high + 1;
    }
}
int Arithmetic::DecodeSymbol ()
{

    long range;
    int cum, symbol;

    range = (long) (high - low) + 1;

    cum = (int) ((((long) (value - low) + 1) * cumFreq [0] - 1)
        / range);
```

```
    for (symbol = 1; cumFreq [symbol] > cum; symbol++);

    high = low + (range * cumFreq [symbol - 1]) / cumFreq [0] -
        1;
    low  = low + (range * cumFreq [symbol]    ) / cumFreq [0];

    for (;;)
    {

        if (high < HALF) {}
        else if (low >= HALF)
        {
            value -= HALF;
            low   -= HALF;
            high  -= HALF;
        }
        else if (low >= FIRST_QTR && high < THIRD_QTR)
        {
            value -= FIRST_QTR;
            low   -= FIRST_QTR;
            high  -= FIRST_QTR;
        }
        else
            break;

        low   = 2 * low;
        high  = 2 * high + 1;
        value = 2 * value + InputBit ();
        if(chError)
        {
            return 0;
        }
    }
    return symbol;
}
bool Arithmetic::Compress (const char *infile, const  char *
   outfile)
{
    int ch, symbol;
    char tmp = 'A';
```

```
in = fopen (infile, "r+b");
if(!keys[0])
{
    out = fopen (outfile, "w+b");
}

if (in == nullptr || (out == nullptr && !keys[0]))
{
    return false;
}

if(!keys[0])
{
    fwrite(&tmp, sizeof(char), 1, out);
}
else
{
    std::cout << tmp;
}

unsigned long long savePos, sizeOfFile;
savePos = ftell(in);
fseek(in, 0, SEEK_END);
sizeOfFile = ftell(in);
fseek(in, savePos, SEEK_SET);
if(!keys[0])
{
    fwrite(&sizeOfFile, sizeof(long long), 1, out);
}
else
{
    std::cout << sizeOfFile;
}

StartOutputingBits ();
StartEncoding ();
for (;;)
{
    ch = getc (in);
    if (ch == EOF)
    {
        break;
```

```
        }
        symbol = charToIndex [ch];
        EncodeSymbol (symbol);
        UpdateModel (symbol);
    }
    EncodeSymbol (EOF_SYMBOL);
    DoneEncoding ();
    DoneOutputingBits ();
    fclose (in);
    if(!keys[0])
    {
        fclose (out);
    }
    return true;
}
bool Arithmetic::Decompress (const char *infile, const char *
   outfile)
{

    int symbol;
    unsigned char ch;
    char typeC = 0;
    unsigned long long oldSize = 0;
    in = fopen (infile, "r+b");
    if(!keys[0] && !keys[5])
    {
        out = fopen (outfile, "w+b");
    }
    if (in == nullptr || (out == nullptr && !keys[0]))
    {
        return false;
    }

    fread(&typeC, sizeof(char), 1, in);
    fread(&oldSize, sizeof(long long), 1, in);

    StartInputingBits ();
    StartDecoding ();
    for (;;)
    {
        symbol = DecodeSymbol ();
```

```
            if(chError)
            {
                return false;
            }

            if (symbol == EOF_SYMBOL)
            {
                break;
            }

            ch = indexToChar [symbol];

            if(!keys[5])
            {
                if(keys[0])
                {
                    std::cout << ch;
                }
                else
                {
                    putc(ch, out);
                }
            }

            UpdateModel (symbol);
        }
        fclose (in);
        if(!keys[0])
        {
            fclose (out);
        }
        return true;
}
```

Arithmetic.h

```
#ifndef ARITHMETIC_H
#define ARITHMETIC_H

#include <iostream>
#include <fstream>
#include <cstdio>
```

```
#include <cstdlib>
#include "Globals.h"

const long BITS_IN_REGISTER = 16;
const long TOP_VALUE            = ((long) 1 << 16) - 1;
const long FIRST_QTR            = (TOP_VALUE / 4) + 1;
const long HALF                 = 2 * FIRST_QTR;
const long THIRD_QTR            = 3 * FIRST_QTR;
const long NO_OF_CHARS          = 256;
const long EOF_SYMBOL           = NO_OF_CHARS + 1;
const long NO_OF_SYMBOLS        = NO_OF_CHARS + 1;
const long MAX_FREQUENCY        = 16383;

class Arithmetic {
public:

    bool Compress (const char*, const char*);

    bool Decompress (const char*, const char*);

    Arithmetic();

private:

    bool chError;

    unsigned char indexToChar [NO_OF_SYMBOLS];
    int charToIndex [NO_OF_CHARS];

    int cumFreq [NO_OF_SYMBOLS + 1];
    int freq [NO_OF_SYMBOLS + 1];

    long low, high;
    long value;

    long bitsToFollow;
    int buffer;
    int bitsToGo;
    int garbageBits;

    FILE *out, *in;
```

```
    void UpdateModel (int);

    void StartInputingBits ();

    void StartOutputingBits ();

    void EncodeSymbol (int);

    void StartEncoding ();

    void DoneEncoding ();

    void StartDecoding ();

    int DecodeSymbol ();

    int InputBit ();

    void OutputBit (int);

    void DoneOutputingBits ();

    void OutputBitPlusFollow (int);
};

#endif
```

Globals.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

#include <vector>
#include <map>
#include <dirent.h>
#include <ctime>
#include <errno.h>
#include <iostream>
#include <sstream>

extern std::vector<bool> keys;
```

```
#endif
```

LZ77.cpp

```
#include "LZ77.h"
#include "LZ77.h"
#include <exception>
#include <iostream>
#include <stdexcept>
#include <stdio.h>
#include <cstdio>

#define KB      1024
#define MB      1024*KB
#define NIL 0xFFFF
#define REM(x,y) ((double)((x)%(y)))/(y)

LZ77::LZ77()
{
        dict=(unsigned char*)calloc(dictSize_c + maxMatch_c,
           sizeof(char));
        hash=(unsigned int*)calloc(hashSize_c, sizeof(unsigned
           int));
        nextlink=(unsigned int*)calloc(dictSize_c, sizeof(
           unsigned int));
}
LZ77::LZ77(IStruct s):
threshold_c     (s.threshold_c),
maxCompares_c   (s.maxCompares_c),
charBits_c              (s.charBits_c),
lengthBits_c            (s.lengthBits_c),
dictBits_c              (s.dictBits_c),
hashBits_c              (s.hashBits_c),
sectorBits_c            (s.sectorBits_c),
maxMatch_c              (s.maxMatch_c),
dictSize_c              (s.dictSize_c),
hashSize_c              (s.hashSize_c),
shiftBits_c             (s.shiftBits_c),
sectorSize_c            (s.sectorSize_c),
sectorAND_c             (s.sectorAND_c)
{
```

```
        dict=(unsigned char*)calloc(dictSize_c + maxMatch_c,
           sizeof(char));
        hash=(unsigned int*)calloc(hashSize_c, sizeof(unsigned
           int));
        nextlink=(unsigned int*)calloc(dictSize_c, sizeof(
           unsigned int));
}
void LZ77::PutBits(unsigned int bits, unsigned int numbits)
{

        bitbuf |= (bits << bitsin);

        bitsin += numbits;

        while (bitsin >= 8)
        {
                if (fputc(bitbuf & 0xFF, outfile) == EOF)
                {
                        printf("\nerror␣writing␣to␣output␣file"
                           );
                        throw std::runtime_error("Error␣while␣
                           writing␣to␣file\n");
                }
                bitbuf >>= 8;
                bitsin -= 8;
        }

}
unsigned int LZ77::GetBits(unsigned int numbits)
{

        unsigned int i;

        i = bitbuf >> (8 - bitsin);

        while (numbits > bitsin)
        {
                if ((bitbuf = getc(infile)) == EOF)
                {
                        printf("\nerror␣reading␣from␣input␣file
                           ");
```

```
                        throw std::runtime_error("Error while
                           reading from file\n");
                }
                i |= (bitbuf << bitsin);
                bitsin += 8;
        }

        bitsin -= numbits;

        return (i & masks[numbits]);

}
void LZ77::PutMatch(unsigned int matchlen, unsigned int
   matchdistance)
{
        PutBits(1, 1);

        PutBits(matchlen - (threshold_c + 1), lengthBits_c);

        PutBits(matchdistance, dictBits_c);
}
void LZ77::PutChar(unsigned int character)
{
        PutBits(0, 1);

        PutBits(character, charBits_c);
}
void LZ77::InitEncode()
{
        register unsigned int i;

        for (i = 0; i < hashSize_c; i++) hash[i] = NIL;

        nextlink[dictSize_c] = NIL;

}
unsigned int LZ77::LoadDict(unsigned int dictpos)
{
        register unsigned int i, j;

        if ((i = fread(&dict[dictpos], sizeof(char),
           sectorSize_c, infile)) == EOF)
```

```
        {
                printf("\nerror reading from input file");
                throw std::runtime_error("Error while loading
                   dictionary from file\n");
        }

        if (dictpos == 0)
        {
                for (j = 0; j < maxMatch_c; j++) dict[j +
                   dictSize_c] = dict[j];
        }

        return i;
}
void LZ77::DeleteData(unsigned int dictpos)
{

        register unsigned int i, j;

        j = dictpos;

        for (i = 0; i < dictSize_c; i++)
                if ((nextlink[i] & sectorAND_c) == j) nextlink[
                   i] = NIL;

        for (i = 0; i < hashSize_c; i++)
                if ((hash[i] & sectorAND_c) == j) hash[i] = NIL
                   ;

}
void LZ77::HashData(unsigned int dictpos, unsigned int
   bytestodo)
{
        register unsigned int i, j, k;

        if (bytestodo <= threshold_c)
                for (i = 0; i < bytestodo; i++) nextlink[
                   dictpos + i] = NIL;
        else
        {
                for (i = bytestodo - threshold_c; i < bytestodo
                   ; i++)
```

```
                        nextlink[dictpos + i] = NIL;
                j = (((unsigned int)dict[dictpos]) <<
                   shiftBits_c) ^ dict[dictpos + 1];
                k = dictpos + bytestodo - threshold_c;

                for (i = dictpos; i < k; i++)
                {
                        nextlink[i] = hash[j = (((j <<
                           shiftBits_c) & (hashSize_c - 1)) ^
                           dict[i + 2])];
                        hash[j] = i;
                }
        }
}
void LZ77::FindMatch(unsigned int dictpos, unsigned int
   startlen)
{
        register unsigned int i, j, k;
        unsigned char l;

        i = dictpos; matchlength = startlen; k = maxCompares_c;
        l = dict[dictpos + matchlength];

        do
        {
                if ((i = nextlink[i]) == NIL) return;

                if (dict[i + matchlength] == l)
                {
                        for (j = 0; j < maxMatch_c; j++)
                                if (dict[dictpos + j] != dict[i
                                    + j]) break;

                        if (j > matchlength)
                        {
                                matchlength = j;
                                matchpos = i;
                                if (matchlength == maxMatch_c)
                                   return;
                                l = dict[dictpos + matchlength
                                   ];
                        }
```

```
                }
        } while (--k);

}
void LZ77::DictSearch(unsigned int dictpos, unsigned int
   bytestodo)
{

        register unsigned int i, j;

        unsigned int matchlen1, matchpos1;

        i = dictpos; j = bytestodo;

        while (j)
        {
                FindMatch(i, threshold_c);

                if (matchlength > threshold_c)
                {
                        matchlen1 = matchlength;
                        matchpos1 = matchpos;

                        for (; ; )
                        {
                                FindMatch(i + 1, matchlen1);

                                if (matchlength > matchlen1)
                                {
                                        matchlen1 = matchlength
                                           ;
                                        matchpos1 = matchpos;
                                        PutChar(dict[i++]);
                                        j--;
                                }
                                else
                                {
                                        if (matchlen1 > j)
                                        {
                                                matchlen1 = j;
                                                if (matchlen1
                                                   <=
```

```
                                                        threshold_c)
                                                         { PutChar(
                                                        dict[i++]);
                                                        j--; break;
                                                        }
                                                }

                                                PutMatch(matchlen1, (i
                                                   - matchpos1) & (
                                                   dictSize_c - 1));
                                                i += matchlen1;
                                                j -= matchlen1;
                                                break;
                                        }
                                }
                        }
                        else
                        {
                                PutChar(dict[i++]);
                                j--;
                        }
                }

}
bool LZ77::Compress(std::string in_str, std::string out_str) {
        FILE* temp_input=infile, *temp_output=outfile;
        try{

                if((infile = fopen(in_str.c_str(), "rb")) ==
                   NULL){
                        std::cerr<<"Error:␣can't␣open␣read␣file
                           ";
                        infile=temp_input, outfile=temp_output;
                        return false;
                }
                if(keys[0]){
                        outfile=stdout;
                }
                else{
                        if((outfile = fopen(out_str.c_str(), "
                           wb")) == NULL){
```

```
                std::cerr<<"Error: can't open
                   write file";
                fclose(infile);
                infile=temp_input, outfile=
                   temp_output;
                return false;
        }
}
char t_char='L';
fwrite(&t_char, sizeof(char), 1, outfile);
unsigned long long savePos, sizeOfFile;
savePos = ftell(infile);
fseek(infile, 0, SEEK_END);
sizeOfFile = ftell(infile);
fseek(infile, savePos, SEEK_SET);
fwrite(&sizeOfFile, sizeof(long long), 1,
   outfile);

unsigned int dictpos, deleteflag, sectorlen;
unsigned long bytescompressed;

InitEncode();

dictpos = deleteflag = 0;

bytescompressed = 0;

while (1)
{
        if (deleteflag) DeleteData(dictpos);

        if ((sectorlen = LoadDict(dictpos)) ==
           0) break;

        HashData(dictpos, sectorlen);

        DictSearch(dictpos, sectorlen);

        bytescompressed += sectorlen;

        dictpos += sectorSize_c;
```

```
                        if (dictpos == dictSize_c)
                        {
                                dictpos = 0;
                                deleteflag = 1;
                        }
                }

                PutMatch(maxMatch_c + 1, 0);

                if (bitsin) PutBits(0, 8 - bitsin);

                if(fclose(infile)){
                        std::cerr<<"Warning: input file closure
                           failed.\n";
                }
                if(fclose(outfile)){
                        std::cerr<<"Warning: output file
                           closure failed. Data loss may occure
                           .\n";
                }
                infile=temp_input, outfile=temp_output;

                return true;
        }
        catch(const std::exception& e){
                std::cerr << "Caught exception \"" << e.what()
                   << "\"\n";
                fclose(infile); fclose(outfile);
                infile=temp_input, outfile=temp_output;
                return false;
        }
}
bool LZ77::Decompress(std::string in_str, std::string out_str)
   {
        FILE* temp_input=infile, *temp_output=outfile;
        try{
                if((infile = fopen(in_str.c_str(), "rb")) ==
                   NULL){
                        std::cerr<<"Error: can't open read file
                           ";
                        infile=temp_input, outfile=temp_output;
                        return false;
```

```
        }
        if(!keys[5])
        {
    if(keys[0])
    {
        outfile=stdout;
    }
    else{
        if((outfile = fopen(out_str.c_str(), "wb")) ==
           NULL)
        {
            std::cerr<<"Error: can't open write file";
            fclose(infile);
            infile=temp_input, outfile=temp_output;
            return false;
        }
    }
}

        char typeC = 0;
        unsigned long long oldSize = 0;
        fread(&typeC, sizeof(char), 1, infile);
        fread(&oldSize, sizeof(long long), 1, infile);

        register unsigned int i, j, k;
        unsigned long bytesdecompressed;
        i = 0;
        bytesdecompressed = 0;
        int64_t countC=0,countM=0,countL=0;
        while(1)
        {
                if (GetBits(1) == 0)
                {
                        countC++;
                        dict[i++] = GetBits(charBits_c)
                           ;
                        if (i == dictSize_c)
                        {
                                if(!keys[5]){
                                        if (fwrite(dict
                                           , sizeof(
                                           char),
```

```
                                        dictSize_c,
                                        outfile) ==
                                        EOF)
                                    {
                                            printf(
                                        "\
                                        nerror
                                        ␣
                                        writing
                                        ␣to␣
                                        output
                                        ␣
                                        file
                                        ");
                                        throw
                                        std
                                        ::
                                        runtime_error
                                        ("
                                        Error
                                        ␣
                                        while
                                        ␣
                                        writing
                                        ␣to␣
                                        output
                                        ␣
                                        file
                                        \n")
                                        ;
                                    }
                                }
                                i = 0;
                                bytesdecompressed +=
                                    dictSize_c;
                        }
                }
                else
                {
                        k = (threshold_c + 1) + GetBits
                            (lengthBits_c);
```

```
if (k == (maxMatch_c + 1))
{
        if(!keys[5]){
                if (fwrite(dict
                   , sizeof(
                   char), i,
                   outfile) ==
                   EOF)
                {
                        printf(
                           "\
                           nerror
                           ␣
                           writing
                           ␣to␣
                           output
                           ␣
                           file
                           ");
                        throw
                           std
                           ::
                           runtime_error
                           ("
                           Error
                           ␣
                           while
                           ␣
                           writing
                           ␣to␣
                           output
                           ␣
                           file
                           \n")
                           ;
                }
        }
        bytesdecompressed += i;
        return true;
}
countM++;
countL+=k;
```

```
j = ((i - GetBits(dictBits_c))
   & (dictSize_c - 1));

do
{
        dict[i++] = dict[j++];
        j &= (dictSize_c - 1);
        if (i == dictSize_c)
        {
                if(!keys[5]){
                        if (
                           fwrite
                           (
                           dict
                           ,
                           sizeof
                           (
                           char
                           ),
                           dictSize_c
                           ,
                           outfile
                           ) ==
                            EOF
                           )
                        {
                                printf
                                   (
                                   "
                                   \
                                   nerror
                                   ␣
                                   writing
                                   ␣
                                   to
                                   ␣
                                   output
                                   ␣
                                   file
                                   "
                                   )
                                   ;
```

```
                                                                throw
                                                                   std
                                                                   ::
                                                                   runtime
                                                                   (
                                                                   "
                                                                   Error
                                                                   ␣
                                                                   while
                                                                   ␣
                                                                   writing
                                                                   ␣
                                                                   to
                                                                   ␣
                                                                   output
                                                                   ␣
                                                                   file
                                                                   \
                                                                   n
                                                                   "
                                                                   )
                                                                   ;
                                                        }
                                                    }
                                                    i = 0;
                                                    bytesdecompressed
                                                        +=
                                                       dictSize_c;
                                                }
                                            } while (--k);
                                        }
                                }
                        }
                catch(const std::exception& e){
                        std::cerr << "Caught␣exception␣\"" << e.what()
                           << "\"\n";
                        fclose(infile); fclose(outfile);
                        infile=temp_input, outfile=temp_output;
                        return false;
```

```
        }
}
```

LZ77.h

```
#pragma once
#include "Globals.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <string>
#include <fstream>
#include <iostream>
#include <memory>
#include <bitset>
#include <string>

class LZ77
{

public:

        typedef struct InitStruct{
                int threshold_c                 = 2;
                int maxCompares_c               = 75;
                int lengthBits_c                = 4;
                int dictBits_c                  = 13;
                int hashBits_c                  = 10;
                int sectorBits_c                = 10;
                int charBits_c                  = 8;
                unsigned int maxMatch_c         = (1 <<
                   lengthBits_c) + threshold_c - 1;
                unsigned int dictSize_c = 1 << dictBits_c;
                unsigned int hashSize_c         = 1 <<
                   hashBits_c;
                unsigned int shiftBits_c        = (hashBits_c +
                    threshold_c) / (threshold_c + 1);
                unsigned int sectorSize_c       = 1 <<
                   sectorBits_c;
                unsigned int sectorAND_c        = (0xFFFF <<
                   sectorBits_c) & 0xFFFF;
```

```
        } IStruct;
        LZ77();
        LZ77(IStruct s);
        void InitEncode();
        bool Compress(std::string in_str, std::string out_str);
        bool Decompress(std::string in_str, std::string out_str
           );

        virtual ~LZ77(){free(dict);free(hash);free(nextlink);};
private:
        unsigned int LoadDict(unsigned int dictpos);
        void DeleteData(unsigned int dictpos);
        void HashData(unsigned int dictpos, unsigned int
           bytestodo);
        void FindMatch(unsigned int dictpos, unsigned int
           startlen);
        void DictSearch(unsigned int dictpos, unsigned int
           bytestodo);

        void PutChar(unsigned int character);
        void PutMatch(unsigned int matchlen, unsigned int
           matchdistance);

        unsigned int GetBits(unsigned int numbits);
        void PutBits(unsigned int bits, unsigned int numbits);

        const int threshold_c           = 2;
        const int maxCompares_c                         = 75;
        const int charBits_c                            = 8;
        const int lengthBits_c                  = 4;
        const int dictBits_c                            = 13;
        const int hashBits_c                            = 10;
        const int sectorBits_c                  = 10;
        const unsigned int maxMatch_c   = (1 << lengthBits_c) +
            threshold_c - 1;
        const unsigned int dictSize_c           = 1 <<
           dictBits_c;
        const unsigned int hashSize_c   = 1 << hashBits_c;
        const unsigned int shiftBits_c  = (hashBits_c +
           threshold_c) / (threshold_c + 1);
        const unsigned int sectorSize_c = 1 << sectorBits_c;
```

```
        const unsigned int sectorAND_c  = (0xFFFF <<
           sectorBits_c) & 0xFFFF;
        unsigned char* dict;
        unsigned int *hash, *nextlink;
        unsigned int
                matchlength=0,
                matchpos=0,
                bitbuf=0,
                bitsin=0,
                masks[17] = {
                   0,1,3,7,15,31,63,127,255,511,1023,2047,4095,8191,16383
                    };

                FILE *infile, *outfile;
};
```

main.cpp

```
#include "preprocessing.h"

std::vector<bool> keys;

int main(int argc, char *argv[])
{
        clock_t t0 = clock();
        std::map<std::string, int> files, filesInDirectories;
        std::map<std::string, int>::iterator finder;
        keys = {false, false, false, false, false, false, false
           , false, false};

        for (int i = 1; i < argc; ++i)
        {
                std::string keyFile;
                std::stringstream tmp(argv[i]);

                tmp >> keyFile;

                if (keyFile[0] == '-')
                {
                        if (!ActivateKeys(keyFile))
                                return -1;
                }
```

```
        else
        {
                if (keyFile.size() > 2)
                        if (keyFile[0] == '.' &&
                           keyFile[1] == '/')
                                keyFile.erase(0, 2);

                if (DirectoryCheck(keyFile, false))
                        GetFiles(keyFile, &
                           filesInDirectories);
                else
                {
                        bool got = false;
                        finder = files.find(keyFile);

                        if (finder != files.end())
                                got = true;

                        if (!got)
                                files.insert({keyFile,
                                   i});
                }
        }
}

if (files.empty() && filesInDirectories.empty())
{
        std::cout << "Compressed data not written to a
           terminal." << std::endl;
        return 0;
}

for (finder = filesInDirectories.begin(); finder !=
   filesInDirectories.end() && keys[4]; ++finder)
        if (finder->first == "main")
                continue;
        else
                if (keys[3])
                        ArchiveInfo(finder->first);
                else if (keys[1] || keys[5])
                        Decompress(finder->first);
                else
```

```
                                Compress(finder->first);

        for (finder = files.begin(); finder != files.end(); ++
           finder)
                if (finder->first == "main")
                        continue;
                else
                        if (keys[3])
                                ArchiveInfo(finder->first);
                        else if (keys[1] || keys[5])
                                Decompress(finder->first);
                        else
                                Compress(finder->first);

        clock_t t1 = clock();

        if (keys[8])
                std::cout << std::endl << ((double) t1 - t0) /
                   CLOCKS_PER_SEC << "␣seconds" << std::endl;

        return 0;
}
```

preprocessing.cpp

```
#include "preprocessing.h"

void Compress(std::string fileName)
{
        std::string compressFile = fileName + ".gz";

        if (ArchiveCheck(fileName))
        {
                std::cout << fileName << "␣already␣has␣.gz␣
                   suffix␣--␣unchanged" << std::endl;
                return;
        }

        FILE *file;

        if (!keys[0])
                if(file = fopen(compressFile.c_str(), "r"))
```

```
        {
                std::cout << compressFile << " is
                   already exists; do you wish to
                   overwrite (y or n)?" << std::endl;
                char save;
                std::cin >> save;
                fclose(file);

                if (save != 'Y' && save != 'y')
                {
                        std::cout << "\tnot overwritten
                           " << std::endl;
                        return;
                }
        }

unsigned long long int sizeLZ, sizeAr;

if (!keys[6] && !keys[7])
{
        sizeLZ = CompressL(fileName);

        if (sizeLZ == 0)
        {
                if (!keys[0])
                        Rm(fileName + ".LZ7");

                return;
        }

        sizeAr = CompressA(fileName);

        if (sizeAr == 0)
        {
                if (!keys[0])
                {
                        Rm(fileName + ".LZ7");
                        Rm(fileName + ".ARI");
                }

                return;
        }
```

```
        }
        else if (keys[6])
        {
                sizeAr = 0;
                sizeLZ = CompressL(fileName);

                if (sizeLZ == 0)
                {
                        if (!keys[0])
                                Rm(fileName + ".LZ7");

                        return;
                }
        }
        else
        {
                sizeLZ = 0;
                sizeAr = CompressA(fileName);

                if (sizeAr == 0)
                {
                        if (!keys[0])
                                Rm(fileName + ".ARI");

                        return;
                }
        }

        if (!keys[0])
                SaveBest(fileName, sizeLZ, sizeAr);
}
void Decompress(std::string fileName)
{
        if (keys[1] && !keys[0])
                if (!ArchiveCheck(fileName))
                {
                        std::cout << fileName << ": unknown
                           suffix -- ignored" << std::endl;
                        return;
                }
```

```
std::string tmp = fileName + ".tmp";
std::string decompresFile = fileName;

decompresFile.pop_back();
decompresFile.pop_back();
decompresFile.pop_back();

FILE* file = NULL;

if (!keys[0] && !keys[5])
        if (file = fopen(decompresFile.c_str(), "r"))
        {
                std::cout << decompresFile << "␣is␣
                   already␣exists;␣do␣you␣wish␣to␣
                   overwrite␣(y␣or␣n)?" << std::endl;
                char save;
                std::cin >> save;
                fclose(file);

                if (save != 'Y' && save != 'y')
                {
                        std::cout << "\tnot␣overwritten
                           " << std::endl;
                        return;
                }
        }

char algo = 0;
bool complete;

if (!(file = fopen(fileName.c_str(), "r")))
{
        std::cout << "Can\'t␣open␣file␣" << fileName <<
            std::endl;
        return;
}

if (!fread(&algo, sizeof(char), 1, file))
{
        std::cout << fileName << ":␣can't␣transfer␣data
           " << std::endl;
        return;
```

```
}
fclose(file);
if (algo == 'A')
{
        Arithmetic* algorithm = new Arithmetic;

        if (algorithm == nullptr)
        {
                std::cout << fileName << ": unexpected
                   memory error" << std::endl;
                return;
        }

        complete = algorithm->Decompress(fileName.c_str
           (), tmp.c_str());
        delete algorithm;
}
else if (algo == 'L')
{
        LZ77* algorithm = new LZ77;

        if (algorithm == nullptr)
        {
                std::cout << fileName << ": unexpected
                   memory error" << std::endl;
                return;
        }

        complete = algorithm->Decompress(fileName, tmp)
           ;
        delete algorithm;
}
else
{
        std::cout << fileName << ": not compressed data
           " << std::endl;
        return;
}

if (!complete)
{
        if (!keys[5])
```

```
                        std::cout << "\t\tdecompressing failed"
                            << std::endl;

                if (!keys[0] && !keys[5])
                        Rm(tmp);

                return;
        }

        if (!keys[0] && !keys[2] && !keys[5])
                Rm(fileName);

        if (!keys[0] && !keys[5])
                Mv(tmp, decompresFile);
}
bool ActivateKeys(std::string argvKeys)
{
    for (int j = 1; j < argvKeys.size(); ++j)
    {
        switch (argvKeys[j])
        {
            case 'a':
                keys[8] = true;
                break;
            case 'c':
                if (!keys[3] || !keys[5])
                    keys[0] = true;
                if (keys[0])
                    keys[2] = false;
                break;
            case 'd':
                if (!keys[3] || !keys[5])
                    keys[1] = true;
                if (keys[1])
                {
                    keys[6] = false;
                    keys[7] = false;
                }
                break;
            case 'k':
                if (!keys[0] || !keys[3] || !keys[5])
                    keys[2] = true;
```

```
            break;
        case 'l':
            keys[3] = true;
            keys[0] = false;
            keys[1] = false;
            keys[2] = false;
            keys[5] = false;
            keys[6] = false;
            keys[7] = false;
            break;
        case 'r':
            keys[4] = true;
            break;
        case 't':
            if (!keys[3])
                keys[5] = true;
            if (keys[5])
            {
                keys[0] = false;
                keys[1] = false;
                keys[2] = false;
                keys[6] = false;
                keys[7] = false;
            }
            break;
        case '1':
            if (!keys[1] && !keys[3] && !keys[5])
                keys[6] = true;
            if (keys[6])
                keys[7] = false;
            break;
        case '9':
            if (!keys[1] && !keys[3] && !keys[5])
                keys[7] = true;
            if (keys[7])
                keys[6] = false;
            break;
        default:
            std::cout << "invalid option -- '" << argvKeys[
               j] << "'" << std::endl;
            return false;
    }
```

```
    }
    return true;
}
void ArchiveInfo(std::string fileName)
{
    char algo = 0;
        FILE *file;
        if(!(file = fopen(fileName.c_str(), "r")))
        {
                std::cout << "Can't open file " << fileName <<
                   std::endl;
                return;
        }

    if (!fread(&algo, sizeof(char), 1, file))
        {
                std::cout << "Can't read file " << fileName <<
                   std::endl;
                return;
        }

    if (algo != 'L' && algo != 'A')
        {
                std::cout << fileName << ": wrong format" <<
                   std::endl;
                return;
        }

    unsigned long long int big, cut;

    if (!fread(&big, sizeof(unsigned long long int), 1, file))
        {
                std::cout << "Can't read file " << fileName <<
                   std::endl;
                return;
        }

        if (fseek(file, 0L, SEEK_END))
        {
                std::cout << "Can't read file " << fileName <<
                   std::endl;
                return;
```

```
        }

    cut = ftell(file);

        fclose(file);

    double ratio = (double) cut / big;
    ratio = 1 - ratio;

    if (big == 0)
        ratio = 0.0;
    else if (ratio < -1)
        ratio = -1;

    std::cout << "          compressed          uncompressed  
       ratio uncompressed_name" << std::endl;

    printf("%19llu %19llu %5.1lf", cut, big, ratio * 100);

    if (ArchiveCheck(fileName))
    {
        fileName.pop_back();
        fileName.pop_back();
        fileName.pop_back();
    }

    std::cout << "% " << fileName << std::endl;
}
unsigned long long int CompressA(std::string fileName)
{
    std::string tmp  = fileName + ".ARI";
    Arithmetic* algorithm = new Arithmetic;

    if (algorithm == nullptr)
    {
        std::cout << fileName << ": unexpected memory error" <<
            std::endl;
        return 0;
    }

    if (!algorithm->Compress(fileName.c_str(), tmp.c_str()))
    {
```

```
        delete algorithm;
        std::cout << "\t\tcompression␣failed" << std::endl;
        return 0;
    }

    delete algorithm;

    if (keys[0])
        return 1;

        FILE *file;

        if (!(file = fopen(tmp.c_str(), "r")))
        {
                std::cout << fileName << ":␣can't␣read␣file" <<
                    std::endl;
        return 0;
        }

        if (fseek(file, 0L, SEEK_END))
        {
                std::cout << "Can't␣read␣file␣" << fileName <<
                   std::endl;
                return 0;
        }

    unsigned long long int size = ftell(file);

        fclose(file);

    return size;
}
unsigned long long int CompressL(std::string fileName)
{
    std::string tmp = fileName + ".LZ7";

    LZ77* algorithm = new LZ77;

    if (algorithm == nullptr)
    {
        std::cout << fileName << ":␣unexpected␣memory␣error" <<
            std::endl;
```

```
        return 0;
    }

    if (!algorithm->Compress(fileName, tmp))
    {
        delete algorithm;
        std::cout << "\t\tcompression failed" << std::endl;
        return 0;
    }

    delete algorithm;

    if (keys[0])
        return 1;

    FILE *file;

        if (!(file = fopen(tmp.c_str(), "r")))
        {
                std::cout << fileName << ": can't read file" <<
                    std::endl;
        return 0;
        }

        if (fseek(file, 0L, SEEK_END))
        {
                std::cout << "Can't read file " << fileName <<
                   std::endl;
                return 0;
        }

    unsigned long long int size = ftell(file);

        fclose(file);

    return size;
}
bool ArchiveCheck(std::string fileName)
{
    int length = fileName.length() - 1;
    return (fileName[length - 2] == '.' && fileName[length - 1]
        == 'g' && fileName[length] == 'z');
```

```
}
bool DirectoryCheck(std::string directoryName, bool help)
{
        DIR* directory = opendir(directoryName.c_str());
        if (directory == NULL)
        {
                if (help && errno != ENOTDIR)
                        ShowErrors(directoryName);
                return false;
        }

        closedir(directory);

        if (errno == EBADF)
        {
                if (help)
                        ShowErrors(directoryName);
                return false;
        }

        return true;
}
void GetFiles(std::string directoryName, std::map<std::string,
   int>* filesInDirectories)
{
        DIR *directory = opendir(directoryName.c_str());

        if (directory == NULL)
        {
                ShowErrors(directoryName);
                return;
        }

        short int depth = 0;

        for (int i = 0; i < directoryName.size(); ++i)
        {
                if (directoryName[i] == '/')
                        ++depth;

                if (depth > 1)
                        break;
```

```
}

struct dirent *directoryFile = readdir(directory);
std::map<std::string, int>::iterator finder;

while (directoryFile)
{
        if (errno == EBADF)
        {
                std::cout << directoryName << ": 
                   something wrong" << std::endl;
                return;
        }

        std::string tmp = std::string(directoryFile->
           d_name);

        if (tmp == "." || tmp == ".." || (tmp == "main"
            && depth == 1 && directoryName[1] == '.'
                && (directoryName[0] == directoryName
                   [1])))
        {
                directoryFile = readdir(directory);
                continue;
        }

        if (directoryName.back() == '/')
                tmp = directoryName + tmp;
        else
                tmp = directoryName + "/" + tmp;

        if (tmp == "./main")
        {
                directoryFile = readdir(directory);
                continue;
        }

        if (DirectoryCheck(tmp, true))
                GetFiles(tmp, filesInDirectories);
        else
        {
                bool got = false;
```

```
                        finder = filesInDirectories->find(tmp);

                        if (finder != filesInDirectories->end()
                           )
                                got = true;

                        if (!got)
                                filesInDirectories->insert({tmp
                                   , filesInDirectories->size()
                                    + 1});
                }

                directoryFile = readdir(directory);
        }

        closedir(directory);

        if (errno == EBADF)
                ShowErrors(directoryName);
}
void SaveBest(std::string fileName, unsigned long long int
   sizeLZ, unsigned long long int sizeAr)
{
    std::string gz;

    if (sizeLZ > 0 && sizeAr > 0)
    {
        if (sizeLZ > sizeAr)
        {
            gz = fileName + ".ARI";
            Rm(fileName + ".LZ7");
        }
        else
        {
            gz = fileName + ".LZ7";
            Rm(fileName + ".ARI");
        }

        Mv(gz, fileName + ".gz");
    }
    else
        if (sizeLZ == 0)
```

```
            Mv(fileName + ".ARI", fileName + ".gz");
        else
            Mv(fileName + ".LZ7", fileName + ".gz");

    if (!keys[2])
        Rm(fileName);
}
void ShowErrors(std::string directoryName)
{
    switch (errno)
    {
        case EACCES:
            std::cout << "No permission for directory " <<
               directoryName
                    << "\t Try it next time" << std::endl;
            break;
        case EBADF:
            std::cout << "Not a valid descriptor for directory
               " << directoryName
                    << "\t Try it next time" << std::endl;
            break;
        case EMFILE:
            std::cout << "Too many files opened in system. Can'
               t open directory" << directoryName
                    << "\t Try it next time" << std::endl;
            break;
        case ENOMEM:
            std::cout << "Not enought memory for opening
               directory " << directoryName
                    << "\t Try it next time" << std::endl;
            break;
        case ENOENT:
            std::cout << "No file or directory with name " <<
               directoryName << std::endl;
            break;
    }
}
void Mv(std::string oldName, std::string newName)
{
    std::string command = "mv ./" + oldName + " ./" + newName;
    system(command.c_str());
}
```

```
void Rm(std::string fileName)
{
    std::string command = "rm␣./" + fileName;
    system(command.c_str());
}
```

preprocessing.h

```
#ifndef MAIN_HELP_H
#define MAIN_HELP_H

#include "Globals.h"
#include "LZ77.h"
#include "Arithmetic.h"
#include <cstdio>
#include <cstdlib>

void Compress(std::string);
void Decompress(std::string);

bool ActivateKeys(std::string);

void ArchiveInfo(std::string);

unsigned long long int CompressA(std::string);
unsigned long long int CompressL(std::string);

bool ArchiveCheck(std::string);
bool DirectoryCheck(std::string, bool);
void GetFiles(std::string, std::map<std::string, int>*);

void SaveBest(std::string, unsigned long long int, unsigned
   long long int);

void ShowErrors(std::string);

void Mv(std::string, std::string);
void Rm(std::string);

#endif
```

Тест производительности

Файл	Размер исходного файла	Алгоритм	Время сжатия (с)	Время декомпрессии (с)	Размер сжатого файла	Коэффициент сжатия
world95.txt		LZ77				
enwik8		LZ77				
enwik9		LZ77				
world95.txt		Арифметика				
enwik8		Арифметика				
enwik9		Арифметика				
world95.txt		gzip				
enwik8		gzip				
enwik9		gzip				

- Центральный процессор -
- Графический адаптер -
- Оперативная память -

Выводы

Благодаря выполнению данного проекта я научился основным принципам работы с файлами и директориями во время компрессии и декомпрессии. Так же я освоил основные правила и принципы работы компрессии и декомпрессии данных. Два построенных алгоритма дали мне необходимый базис навыков для работы с кастомными буферами. Так же я значительно улучшил свои навыки написания комплексных программ построения утилит, к примеру я научился реализовывать поддержку ключей и возможность работы нескольких алгоритмов.

Помимо этого стоит отметить, что статическое арифметическое кодирование будет проигрывать зачастую динамическому. Причина кроется в том, что в процессе чтения файла будут складываться ситуации, когда какой-то символ попадается достаточно часто, так что по сути его можно будет закодировать меньшим числом бит. Но если у нас статическое арифметическое кодирование, то возможна ситуация, когда под такой символ понадобится больше битов для кодирования, чем нужно. По сути динамический вариант может хорошо адаптироваться под какие-то временные особенности текста, что благоприятно влияет на качество, но взамен немного бьёт по времени работы.

Список литературы

1. Арифметическое кодирование - Arithmetic coding [Электронный ресурс]: ru.qwe.wiki URL: https://ru.qwe.wiki/wiki/Arithmetic_coding (дата обращения 28.08.2020)

2. Arithmetic Coding [Электронный ресурс]: users.cs.cf.ac.uk URL: https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node213.html (дата обращения 16.09.2020)

3. LZ77 на C, реализация алгоритма LZ77 на C [Электронный ресурс]: algor.skyparadise.org URL: https://algor.skyparadise.org/read/14 (дата обращения 16.09.2020)