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: О. Р. Лисовский
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> ...

Обработка входных данных

Программа начинается с обработки строки стандартного ввода. Строка обрабатывается по словам. Если слово начинается с символа «-», то предполагается что это набор ключей и ключи передаются в специальную функцию, чтобы исключить противоречия работы ключей. Возможные сочетания и главенство одних ключей над другими описано в таблице ниже.

	d	k	l	r	t	1	9
c	нет блока	c блокирует k	l блокирует c	нет блока	t блокирует c	нет блока	нет блока
d	-	нет блока	l блокирует d	нет блока	t блокирует d	d блокирует 1	d блокирует 9
k	-	-	l блокирует k	нет блока	t блокирует k	нет блока	нет блока
l	-	-	-	нет блока	l блокирует t	l блокирует 1	l блокирует 9
r	-	-	-	-	нет блока	нет блока	нет блока
t	-	-	-	-	-	t блокирует 1	t блокирует 9
1	-	-	-	-	-	-	последний полученный блокирует прошлые

В случае если полученное слово начинается с другого символа, то программа предполагает что это имя файла или директории и добавляет его в список для дальнейшей обработки.

Интерфейс

После установления активных ключей и заполнения списка объектов компрессии/-декомпрессии, программа начинает работу этим самым списком. В случае отсутствия ключа -r все директории не рассматриваются. При активации ключа всё содержимое директории рекурсивно обрабатывается программой. При работе с файлами проверяется наличие/отсутствие (в зависимости от ключа -d) файла с расширением .gz и в случае необходимости программа спрашивает у пользователя право на перезапись соответствующего файла.

При подготовке непосредственно компрессии проверяется наличие ключа -1 или -9 для определения необходимого алгоритма. В случае их отсутствия используются оба алгоритма и выбирается лучший результат.

При подготовке непосредственно декомпрессии читается первый байт файла для установления алгоритма декодирования.

Постобработка

При окончании работы компрессии/декомпрессии программа получает сигнал об их завершении. Если этот сигнал соответствует ошибке то работа с конкретным файлом аварийно прекращается и обрабатывается следующий файл. Дальнейшие действия обусловлены введёнными ключами.

Далее незакодированный файл будет упоминаться как файл, а закодированный файл как архив.

Арифметическая компрессия

В данной курсовой работе реализована целочисленная компрессия, так как реализовать компрессию на числах с плавающей точкой крайне проблематично в силу ограничений. Например, тот же машинный эпсилон или точность в целом. Целочисленная компрессия довольно неплохо отличается от арифметики с плавающей точкой, однако ниже будет описано всё наиболее подробно.

В первую очередь настраивается таблица частот для последующей работы и кодировки символов. Частоты символов выставляются на единицы, чтобы накопленные частоты высчитывались правильно. В соответствии с ними выставляются и накопленные частоты. Они необходимы для определения отрезков, которые как раз отвечают за кодировку символов.

После этого открывается новый временный файл, куда заносится тип кодировки и размер исходного файла. Задаются стартовые значения для счётчиков и инициализируется буфер.

Далее начинается кодировка символов. В соответствии каждому символу выставляется нужный отрезок. Цель заключается в том, чтобы записать старшие биты, по которым будет достаточно определить отрезок, в файл. Проводятся некоторые манипуляции с отрезком, благодаря которым можно записать нужные старшие биты. После

кодирования символа отрезок остаётся тем же, и продолжаем кодировать при помощи него и следующие символы.

После кодирования символов обновляется и оптимизируется таблица частот.

После кодирования символов мы кодируем EOF и дописываем недостающие биты. Завершаем работу с файлом.

Арифметическая декомпрессия

Производится подготовка таблицы частот, аналогично компрессии.

На вход поступает сжатый файл, в котором закодировано число. Мы заносим в наш буфер первые 16 бит. Самые старшие из них по сути и дают нам информацию об отрезке, а следовательно и символе. После определения символа мы проводим аналогичные манипуляции с отрезком, как и в случае с компрессией, отбрасывая уже ненужные биты и загружая новые.

После этого также обновляется и оптимизируется таблица частот. Оптимизация в первую очередь нужна именно во время декомпрессии. Она заключается в том, чтобы наиболее часто встречающиеся символы оказывались ближе к началу таблицы частот. Это ускоряет поиск, а значит и распаковку.

LZ77 компрессия

LZ77 использует уже просмотренную часть сообщения как словарь. Чтобы добиться сжатия, он пытается заменить очередной фрагмент сообщения на указатель в содержимое словаря.

В качестве модели данных LZ77 использует “скользящее” по сообщению окно, разделенное на две неравные части. Первая, большая по размеру, включает уже просмотренную часть сообщения и называется словарём. Вторая, намного меньшая, является буфером, содержащим еще не закодированные символы входного потока. Обычно размер окна составляет несколько килобайтов. Буфер намного меньше, обычно не более ста байтов. Алгоритм пытается найти в словаре фрагмент, совпадающий с содержимым буфера.

Алгоритм LZ77 выдает коды, состоящие из трех элементов:<offset,length,symbol>

1. offset - смещение в словаре относительно его начала подстроки, совпадающей с содержимым буфера;

2. length - длина подстроки;

3. symbol - первый символ в буфере, следующий за подстрокой.

Если совпадение не обнаружено, то алгоритм выдает код <0, 0, первый символ в буфере> и продолжает свою работу. Хотя такое решение неэффективно, но оно гарантирует, что алгоритм сможет закодировать любое сообщение. Можно улучшить алгоритм для данного случая, если вместо <offset,length,symbol> записывать либо нулевой бит, а

затем <offset,length>, либо единичный бит, а затем <symbol>.Такая версия алгоритма называется LZS и тратит на одиночные символы гораздо меньше бит. Также для ускорения сжатия для поиска совпадения в словаре использовалась хэш таблица и словарь был представлен циклическим буфером.

LZ77 декомпрессия

Декодер LZ77 тривиально прост и может работать со скоростью, приближающейся к скорости процедуры обычного копирования информации. В стандартной версии LZ77 мы считываем количество бит, которым кодируется <offset,length,symbol>, после чего находим нужную позицию в тексте с помощью offset и копируем посимвольно length символов, после чего пишем символ symbol. В улучшенной версии LZS сначала декодер считывает один бит, чтобы определить, закодирована ли пара <offset,length> или <symbol>. Если это символ, то следующие 8 битов выдаются как раскодированный символ и помещаются в скользящее окно. Иначе, если это не закодированный конец файла, то соответствующее количество символов словаря помещается в окно и выдается в раскодированном виде. Для ускорения работы запись раскодированных данных ведётся не сразу в выходной файл, а в буфер, который при заполнении записывается в файл. Поскольку это все, что делает декодер, понятно, почему процедура декодирования работает так быстро.

По окончании чтения архива, количество байт, которое было в изначальном файле, сверяется с тем, сколько было записано в его новую версию. При несовпадении выводится соответствующее сообщение, и декомпрессия завершается неудачно.

Описание файлов программы

Код программы разбит на 9 файлов:

1. ACC.h - Содержит базовую информацию о классе TACC, необходимом для работы компрессии и декомпрессии соответствующего алгоритма.
2. ACC.cpp - Содержит реализацию класса TACC.
3. BFile.h - Содержит базовую информацию о классах TOutBinary и класса TInBinary, необходимых для работы с файлами.
4. BFile.cpp - Содержит реализацию классов TOutBinary и TInBinary.
5. interface.h - Содержит в себе перечисление и описание всех функций необходимых для взаимодействия программы и алгоритмов сжатия данных.
6. interface.cpp - Содержит реализацию всех функций, описанных в файле interface.h.
7. Library.h - Содержит в себе ключи, необходимые для работы алгоритмов, и библиотеки для работы всей программы.

8. LZ77.h - Содержит базовую информацию о классе TLZ77, необходимом для работы компрессии и декомпрессии соответствующего алгоритма.
9. LZ77.cpp - Содержит реализацию класса TLZ77.
10. main.cpp - Файл запуска.
11. Makefile - Сборочный файл.

Основные типы данных

1. TOutBinary - класс, обеспечивающий запись необходимого количества байт в файл.
2. TInBinary - класс обеспечивающий считывание необходимого количества байт из файла.
3. TLZ77 - класс, описывающий работу алгоритма LZ77.
4. TACC - класс, описывающий работу арифметического алгоритма.

Описание методов и функций программы

Основные свойства и методы класса TACC

public:

1. bool Compress (const char*, const char*) - сжатие файла.
2. bool Decompress (const char*, const char*) - распаковка файла.
3. TACC() - 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 EncodeSymbol (int) - кодировка символа.
17. void InputFileInfo() - запись информации о сжимаемом файле.
18. void StartEncoding() - подготовка к сжатию.
19. void DoneEncoding() - завершение кодирования. Загрузка последних битов в буфер.
20. void StartDecoding() - подготовка к распаковке.
21. int DecodeSymbol() - распаковка символа.
22. int InputBit() - получение одного бита из файла.
23. void OutputBit(int) - отправление одного бита в файл.
24. void OutputBitPlusFollow(int) - вывод указанного бита и отложенных ранее.

Основные свойства и методы класса TOutBinary

public:

1. TOutBinary() - задаёт начальные значения. Файл не будет открыт.
2. bool Open(std::string*) - открывает файл.
3. bool Close() - закрывает файл.

4. bool Write(const char*, size_t) - запись в файл.
5. bool WriteBin(size_t bit) - запись бита в файл.
6. unsigned long long SizeFile() - подсчёт размера файла.
7. friend bool operator « (TOutBinary& file, size_t const &bit) - запись бита в файл.

private:

1. std::ofstream out - файл вывода.
2. std::string name - имя файла.
3. unsigned char head - маска для заноса бита в block.
4. unsigned char block - временный буфер для хранения и записи битов в файл.

Основные свойства и методы класса TInBinary

public:

1. TInBinary() - задаёт начальные значения. Файл не будет открыт.
2. bool Open(std::string*) - открывает файл.
3. bool Close() - закрывает файл.
4. bool Read(char*, size_t) - считывает из файла некоторое количество байт.
5. bool ReadBin(char* bit) - считывает из файла один бит.
6. unsigned long long SizeFile() - подсчёт размера файла.
7. friend bool operator » (TInBinary& iFile, char &bit) - получение бита из файла.

private:

1. std::ifstream in - файл вывода.
2. std::string name - имя файла.
3. unsigned char head - маска для заноса бита в block.
4. unsigned char block - временный буфер для хранения битов из файла, через него получают биты.

Основные свойства и методы класса TLZ77

public:

1. TLZ77() - стандартный конструктор.
2. TLZ77(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. LoadDict(unsigned int dictpos) - загрузка словаря из файла в циклический буфер на позиции dictpos.
7. DeleteData(unsigned int dictpos) - удаления всех ссылок на удаляемый сектор с началом в dictpos.
8. HashData(unsigned int dictpos, unsigned int bytestodo) - хэширование и запись ссылок на возможное преведущее совпадение в словаре.
9. FindMatch(unsigned int dictpos, unsigned int startlen) - поиск максимального совпадения в словаре с позицией dictpos, не меньше чем startlen.
10. DictSearch(unsigned int dictpos, unsigned int bytestodo) - кодирование считанного сектора с началом в dictpos и длинной bytestodo.
11. SendChar(unsigned int character) - кодирование символа character.
12. SendMatch(unsigned int matchlen, unsigned int matchdistance) - кодирование пары <matchlen, matchdistance>.
13. ReadBits(unsigned int numbits) - считывание numbits битов из файла.
14. SendBits(unsigned int bits, unsigned int numbits) - отправка numbits битов записанных в bits в файл.
15. ∼TLZ77() - деструктор.

private:

1. const int compressFloor - минимальное совпадение, для записи в виде<length,offset>.
2. const int comparesCeil - максимальное число раз которое ищется совпадение в FindMatch.
3. const int CHARBITS - сколькими битами кодируется символ.

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

Прочие функции

1. void FileIterator(std::map<std::string, int>) - Осуществляет проход по всем папкам и файлам для их компрессии/декомпрессии.
2. bool Parser(std::map<std::string, int>*, std::string) - фильтрует полученные при вводе аргументы. При получении некорректного аргумента возвращает false.

3. bool AskDir(std::string, bool) - Проверка на существование директории. При существовании возвращает true, в любом ином случае false.

4. void DirectoryWork(std::string) - В случае наличия ключа -r осуществляет работу с внутренними файлами и директориями указанной директории.

5. void DeComPress(std::string) - Помогает определить действия по отношению к указанному файлу: совершить компрессию, декомпрессию или посмотреть информацию об архиве.

6. bool Rewrite(std::string) - В случае возможного повторения имён файлов при компрессии/декомпрессии принимает решение о перезаписи.

7. void ErrorNotes(std::string) - Показывает сообщения об ошибках, возникших при работе с указанной директорией.

8. bool KeyL(TInBinary*, std::string) - Осуществляет работу ключа -l - вывод информации об архиве.

9. void PreCompress(TInBinary*, std::string) - Осуществляет подготовку указанного файла к сжатию в соответствии с указанными ключами.

10. unsigned long long int Compress(std::string, TInBinary*, bool) - Непосредственно активирует указанный алгоритм сжатия. Возвращает размер полученного архива или 0 в случае ошибки.

11. void PreDecompress(TInBinary*, std::string) - Осуществляет подготовку указанного файла к разжатию в соответствии с указанными ключами.

Исходный код

ACC.h

```
#ifndef ACC_H
#define ACC_H

#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstdlib>
#include "Library.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 ACC {
public:
    bool Compress (const char*, const char*);
    bool Decompress (const char*, const char*);
    ACC();
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 EncodeSymbol (int);
    void InputFileInfo ();
    void StartEncoding ();
    void DoneEncoding ();
    void StartDecoding ();
    int DecodeSymbol ();
    int InputBit ();
    void OutputBit (int);
    void OutputBitPlusFollow (int);
};

#endif
```

ACC.cpp

```
#include "ACC.h"
```

```
ACC::ACC () {
    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 ACC::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;
```

```
    }
    return;
}

void ACC::InputFileInfo() {
    char tmp = 'A';
    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;
    }
    return;
}

int ACC::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 ACC::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;
    }
    return;
}

void ACC::OutputBitPlusFollow (int bit) {
    OutputBit (bit);
    while (bitsToFollow > 0) {
        OutputBit (!bit);
        --bitsToFollow;
    }
    return;
}

void ACC::StartEncoding () {
    buffer = 0;
    bitsToGo = 8;
    low = 0l;
    high = TOP_VALUE;
    bitsToFollow = 0l;
    return;
}

void ACC::DoneEncoding () {
    ++bitsToFollow;
```

```
    if (low < FIRST_QTR) {
        OutputBitPlusFollow(0);
    }
    else {
        OutputBitPlusFollow(1);
    }
    if(keys[0]) {
        std::cout << (buffer >> bitsToGo);
    }
    else {
        putc(buffer >> bitsToGo, out);
    }
    return;
}

void ACC::StartDecoding () {
    bitsToGo = 0;
    garbageBits = 0;
    value = 0l;
    for ( int i = 0; i < BITS_IN_REGISTER; ++i) {
        value = 2 * value + InputBit ();
    }
    low = 0l;
    high = TOP_VALUE;
    return;
}

void ACC::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;
    }
    return;
}

int ACC::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 ACC::Compress (const char *infile, const  char *outfile) {
    int ch, symbol;
    in = fopen (infile, "r+b");
    if(!keys[0]) {
        out = fopen (outfile, "w+b");
    }
    if (in == nullptr || (out == nullptr && !keys[0])) {
        return false;
    }
        InputFileInfo();
    StartEncoding ();
    for (;;) {
        ch = getc (in);
        if (ch == EOF) {
            break;
        }
        symbol = charToIndex [ch];
        EncodeSymbol (symbol);
        UpdateModel (symbol);
    }
    EncodeSymbol (EOF_SYMBOL);
    DoneEncoding ();
    fclose (in);
    if(!keys[0]) {
        fclose (out);
    }
    return true;
}

bool ACC::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);
    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] && !keys[5]) {
        fclose (out);
    }
    return true;
}
```

BFile.h

```
#ifndef BFILE_H
#define BFILE_H

#include <iostream>
#include <fstream>
#include <string>
```

```
class TOutBinary {
public:
    TOutBinary();
    bool Open(std::string* name);
    bool Close();
    bool Write(const char* obj, size_t size);
    bool WriteBin(size_t bit);
    unsigned long long SizeFile();
    friend bool operator << (TOutBinary& file, size_t const &
       bit);
private:
    std::ofstream out;
    std::string name;
    unsigned char head;
    unsigned char block;
};

class TInBinary {
public:
    TInBinary();
    bool Open(std::string* name);
    bool Close();
    bool Read(char* obj, size_t size);
    bool ReadBin(char* bit);
    unsigned long long SizeFile();
    friend bool operator >> (TInBinary& iFile, char &bit);
private:
    std::ifstream in;
    std::string name;
    unsigned char head;
    unsigned char block;
};

#endif
```

BFile.cpp

```
#include "BFile.h"

TOutBinary::TOutBinary() {
    head = 1 << 7;
```

```
    block = 0;
}

bool TOutBinary::Open(std::string* name) {
    if(out.is_open()) {
        return false;
    }
    else {
        out.open(name->c_str(), std::ofstream::out);
        if(!out) {
            return false;
        }
        else {
            this->name = *name;
            head = 1 << 7;
            block = 0;
            return true;
        }
    }
}

bool TOutBinary::Close() {
    if(out.is_open()) {
        if(!(head & (1 << 7))) {
            out << block;
        }
        out.close();
        if(out.fail()) {
            return false;
        }
        return true;
    }
    else {
        return false;
    }
}

bool TOutBinary::Write(const char* obj, size_t size) {
    if(out.is_open()) {
        if(!(head & (1 << 7))) {
            out << block;
        }
```

```
        out.write(obj, size);
        return true;
    }
    else {
        return false;
    }
}

bool TOutBinary::WriteBin(size_t bit) {
    if(out.is_open()) {
        if(bit) {
            block |= head;
        }
        head >>= 1;
        if(!head) {
            out << block;
            block = 0;
            head = 1 << 7;
        }
        return true;
    }
    else {
        return false;
    }
}

unsigned long long TOutBinary::SizeFile() {
    std::ifstream in(name, std::ifstream::ate | std::ifstream::
       binary);
    return in.tellg();
}

bool operator << (TOutBinary& file, size_t const &bit) {
    if(file.out.is_open()) {
        if(bit) {
            file.block |= file.head;
        }
        file.head >>= 1;
        if(!file.head) {
            file.out << file.block;
            file.block = 0;
            file.head = 1 << 7;
```

```
        }
        return true;
    }
    else {
        return false;
    }
}

TInBinary::TInBinary() {
    head = 0;
    block = 0;
}

bool TInBinary::Open(std::string* name) {
    if(in.is_open()) {
        return false;
    }
    else {
        in.open(name->c_str(), std::ofstream::in);
        if(!in) {
            return false;
        }
        else {
            this->name = *name;
            head = 0;
            block = 0;
            return true;
        }
    }
}

bool TInBinary::Close() {
    in.close();
    if(in.fail()) {
        return false;
    }
    return true;
}

bool TInBinary::Read(char* obj, size_t size) {
    if(!in.eof()) {
        in.read(obj, size);
```

```
        return true;
    }
    else {
        return false;
    }
}

bool TInBinary::ReadBin(char* bit) {
    if(!head) {
        if(in >> block) {
            head = 1 << 7;
        }
        else {
            return false;
        }
    }
    ((block & head) != 0) ? (*bit = 1) : (*bit = 0);
    head >>= 1;
    return true;
}

unsigned long long TInBinary::SizeFile() {
    std::ifstream in(name, std::ifstream::ate | std::ifstream::
       binary);
    return in.tellg();
}

bool operator >> (TInBinary& iFile, char& bit) {
    if(!iFile.head) {
        if(iFile.in >> iFile.block) {
            iFile.head = 1 << 7;
        }
        else {
            return false;
        }
    }
    ((iFile.block & iFile.head) != 0) ? (bit = 1) : (bit = 0);
    iFile.head >>= 1;
    return true;
}
```

interface.h

```
#ifndef MAIN_HELP_H
#define MAIN_HELP_H

#include "Library.h"
#include "ACC.h"
#include "LZ77.h"
#include "BFile.h"
#include <cstdio>
#include <cstdlib>
#include <string>

void FileIterator(std::map<std::string, int>);

bool Parser(std::map<std::string, int>*, std::string);

bool AskDir(std::string, bool);

void DirectoryWork(std::string);

void DeComPress(std::string);

bool Rewrite(std::string);

void ErrorNotes(std::string);

bool KeyL(TInBinary*, std::string);

void PreCompress(TInBinary*, std::string);

unsigned long long int Compress(std::string, TInBinary*, bool);

void PreDecompress(TInBinary*, std::string);

#endif
```

interface.cpp

```
#include "interface.h"

void FileIterator(std::map<std::string, int> files) {
```

```
        std::map<std::string, int>::iterator i;
        bool directory;
        for (i = files.begin(); i != files.end(); ++i) {
                directory = AskDir(i->first, false);
                if (!keys[4] && directory) {
                        std::cout << i->first << "␣is␣a␣
                           directory␣--␣ignored\n";
                }
                else if (directory) {
                        DirectoryWork(i->first);
                }
                else if (i->first == "main") {
                        continue;
                }
                else if (errno == ENOTDIR) {
                        DeComPress(i->first);
                }
                else {
                        ErrorNotes(i->first);
                }
        }
        return;
}

bool Parser(std::map<std::string, int>* fileNames, std::string
   argv) {
        if (argv[0] != '-') {
                if (argv[0] == '.' && argv[1] == '/' && argv.
                   size() > 2) {
                        argv.erase(0, 2);
                }
                if (fileNames->find(argv) == fileNames->end())
                   {
                        fileNames->insert({argv, fileNames->
                           size() + 1});
                }
                return true;
        }
    for (int j = 1; j < argv.size(); ++j) {
        switch (argv[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 -- '" << argv[j]
                   << "'\n";
                return false;
        }
    }
    return true;
}

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

```
                }
                return false;
        }
        return true;
}

void DirectoryWork(std::string dirName) {
        short int directoriesIn = 0;
        for (int i = 0; i < dirName.size(); ++i) {
                if (dirName[i] == '/') {
                        ++directoriesIn;
                }
                if (directoriesIn > 1) {
                        break;
                }
        }
        DIR *directory = opendir(dirName.c_str());
        if (directory == NULL) {
                ErrorNotes(dirName);
                return;
        }
        struct dirent *file = readdir(directory);
        bool programm;
        while (file) {
                if (errno == EBADF) {
                        std::cout << dirName << ": something
                           wrong\n";
                        return;
                }
                std::string fileName = std::string(file->d_name
                   );
                programm = fileName == "main" && directoriesIn
                   == 1 && dirName[0] == dirName[1] && dirName
                   [1] == '.';
                if (fileName == "." || fileName == ".." ||
                   programm) {
                        file = readdir(directory);
                        continue;
                }
                if (dirName.back() == '/') {
                        fileName = dirName + fileName;
                }
```

```
                else {
                        fileName = dirName + "/" + fileName;
                }
                if (fileName == "./main") {
                        file = readdir(directory);
                        continue;
                }
                if (AskDir(fileName, true)) {
                        DirectoryWork(fileName);
                }
                else {
                        DeComPress(fileName);
                }
                file = readdir(directory);
        }
        closedir(directory);
        if (errno == EBADF) {
                ErrorNotes(dirName);
        }
        return;
}

void DeComPress(std::string fileName) {
    TInBinary* file = new TInBinary;
    if (file == nullptr) {
        std::cout << fileName << ": unexpected memory error\n";
        exit(1);
    }
    if (!file->Open(&fileName)) {
        std::cout << fileName << ": can't read file\n";
        delete file;
        return;
    }
    if (keys[3]) {
        if(!KeyL(file, fileName)) {
            std::cout << fileName << ": wrong format\n";
        }
    }
    else if (keys[1] || keys[5]) {
        PreDecompress(file, fileName);
    }
    else {
```

```
        PreCompress(file, fileName);
    }
    file->Close();
    delete file;
    return;
}

bool Rewrite(std::string file) {
        std::cout << file << "␣is␣already␣exists;␣do␣you␣wish␣
           to␣overwrite␣(y␣or␣n)?\n";
        char input;
        std::cin >> input;
        if (input != 'Y' && input != 'y') {
                std::cout << "\tnot␣overwritten\n";
                return false;
        }
        return true;
}

void ErrorNotes(std::string dir) {
        std::string tmp = "␣directory␣" + dir + "\t␣Try␣it␣next
           ␣time\n";
    switch (errno) {
        case EACCES:
            std::cout << "No␣permission␣for" << tmp;
            break;
        case EBADF:
            std::cout << "Not␣a␣valid␣descriptor␣for" << tmp;
            break;
        case EMFILE:
            std::cout << "Too␣many␣files␣opened␣in␣system.␣Can'
               t␣open" << tmp;
            break;
        case ENOMEM:
            std::cout << "Not␣enought␣memory␣for␣opening" <<
               tmp;
            break;
        case ENOENT:
            std::cout << "No␣file␣or␣directory␣with␣name␣" <<
               dir << "\n";
            break;
    }
```

```
    return;
}

bool KeyL(TInBinary* archive, std::string archiveName) {
    char letter = 0;
    if (!archive->Read(&letter, sizeof(char))) {
        return false;
    }
    if (letter != 'L' && letter != 'A') {
        return false;
    }
    unsigned long long int original, compressed;
    if (!archive->Read((char*)&original, sizeof(unsigned long
       long int))) {
        return false;
    }
    compressed = archive->SizeFile();
    double coef = 1.0 - (double) compressed / original;
    if (original == 0) {
        coef = 0.0;
    }
    else if (coef < -1) {
        coef = -1.0;
    }
    std::cout << "␣␣␣␣␣␣␣␣␣compressed␣␣␣␣␣␣␣␣uncompressed␣␣
       ratio␣uncompressed_name\n";
    printf("%19llu␣%19llu␣%5.1lf", compressed, original, coef *
        100);
    if (archiveName.substr(archiveName.length() - 3) == ".gz")
       {
                archiveName.erase(archiveName.size() - 3, 3);
    }
    std::cout << "%␣" << archiveName << "\n";
    return true;
}

void PreCompress(TInBinary* file, std::string fileName) {
        std::string archiveName = fileName + ".gz";
        file->Close();
        if (fileName.substr(fileName.length() - 3) == ".gz") {
                std::cout << fileName << "␣already␣has␣.gz␣
                   suffix␣--␣unchanged\n";
```

```
        return;
}
if (!keys[0] && file->Open(&archiveName)) {
        file->Close();
        if (!Rewrite(archiveName)) {
                return;
        }
}
file->Close();
unsigned long long int LZ77, arithmetic;
if (!keys[6] && !keys[7]) {
        arithmetic = Compress(fileName, file, false);
        if (arithmetic == 0) {
                if (!keys[0]) {
                        archiveName = "rm␣./" +
                           fileName + ".A";
                        system(archiveName.c_str());
                }
                return;
        }
        file->Close();
        LZ77 = Compress(fileName, file, true);
        file->Close();
        if (LZ77 == 0) {
                if (!keys[0]) {
                        archiveName = "rm␣./" +
                           fileName + ".L␣" + fileName
                           + ".A";
                        system(archiveName.c_str());
                }
                return;
        }
}
else if (keys[6]) {
        arithmetic = 0;
        LZ77 = Compress(fileName, file, true);
        file->Close();
        if (LZ77 == 0) {
                if (!keys[0]) {
                        archiveName = "rm␣./" +
                           fileName + ".L";
                        system(archiveName.c_str());
```

```
                        }
                        return;
                }
        }
        else {
                LZ77 = 0;
                arithmetic = Compress(fileName, file, false);
                if (arithmetic == 0) {
                        if (!keys[0]) {
                                archiveName = "rm␣./" +
                                   fileName + ".A";
                                system(archiveName.c_str());
                        }
                        return;
                }
        }
        if (keys[0]) {
                return;
        }
        archiveName = "";
        if (!keys[2]) {
                archiveName += "rm␣./" + fileName + "\n";
    }
    if (LZ77 == 0) {
                archiveName += "mv␣./" + fileName + ".A␣./" +
                   fileName + ".gz";
    }
    else if (arithmetic == 0) {
                archiveName += "mv␣./" + fileName + ".L␣./" +
                   fileName + ".gz";
        }
        else if (LZ77 > arithmetic) {
                archiveName += "mv␣./" + fileName + ".A␣./" +
                   fileName + ".gz\nrm␣./" + fileName + ".L";
        }
        else {
                archiveName += "mv␣./" + fileName + ".L␣./" +
                   fileName + ".gz\nrm␣./" + fileName + ".A";
        }
        system(archiveName.c_str());
        return;
}
```

```
unsigned long long int Compress(std::string fileName, TInBinary
   * file, bool LZ) {
    std::string tmpName;
    if (LZ) {
                tmpName = fileName + ".L";
                TLZ77* algorithm = new TLZ77;
                if (algorithm == nullptr) {
                        std::cout << fileName << ":␣unexpected␣
                           memory␣error\n";
                        exit(1);
                }
                if (!algorithm->Compress(fileName, tmpName)) {
                        delete algorithm;
                        std::cout << "\t\tcompression␣failed\n"
                           ;
                        return 0;
                }
                delete algorithm;
        }
        else {
                tmpName = fileName + ".A";
                ACC* algorithm = new ACC;
                if (algorithm == nullptr) {
                        std::cout << fileName << ":␣unexpected␣
                           memory␣error\n";
                        exit(1);
                }
                if (!algorithm->Compress(fileName.c_str(),
                   tmpName.c_str())) {
                        delete algorithm;
                        std::cout << "\t\tcompression␣failed\n"
                           ;
                        return 0;
                }
                delete algorithm;
        }
        if (keys[0]) {
                return 1;
        }
    if (!file->Open(&tmpName)) {
        std::cout << fileName << ":␣can't␣read␣file\n";
```

```
        return 0;
    }
    return file->SizeFile();
}

void PreDecompress(TInBinary* archive, std::string archiveName)
    {
        if (keys[1] && !keys[0]) {
                if (!(archiveName.substr(archiveName.length() -
                    3) == ".gz")) {
                        std::cout << archiveName << ": unknown
                           suffix -- ignored\n";
                        return;
                }
        }
        std::string decompressName = archiveName + ".t";
        archive->Close();
        std::string fileName = archiveName;
        fileName.erase(fileName.size() - 3, 3);
        if (!keys[0] && !keys[5] && archive->Open(&fileName)) {
                archive->Close();
                if (!Rewrite(fileName)) {
                        return;
                }
        }
        archive->Close();
        if (!archive->Open(&archiveName)) {
                std::cout << archiveName << ": can't read file\
                   n";
                return;
        }
        char code = 0;
        bool decompression;
        if (!archive->Read(&code, sizeof(char))) {
                std::cout << archiveName << ": can't transfer
                   data\n";
                return;
        }
        if (code == 'A') {
                ACC* algorithm = new ACC;
                if (algorithm == nullptr) {
```

```
                std::cout << archiveName << ": unexpected memory error\n";
                exit(1);
        }
        decompression = algorithm->Decompress(archiveName.c_str(), decompressName.c_str());
        delete algorithm;
}
else if (code == 'L') {
        TLZ77* algorithm = new TLZ77;
        if (algorithm == nullptr) {
                std::cout << archiveName << ": unexpected memory error\n";
                exit(1);
        }
        decompression = algorithm->Decompress(archiveName, decompressName);
        delete algorithm;
}
else {
        std::cout << archiveName << ": not compressed data\n";
        return;
}
if (!decompression) {
        if (!keys[5]) {
                std::cout << "\t\tdecompressing failed\n";
        }
        if (!keys[0] && !keys[5]) {
                decompressName = "rm ./" + decompressName;
                system(decompressName.c_str());
        }
        return;
}
if (!keys[0] && !keys[2] && !keys[5]) {
        archiveName = "rm ./" + archiveName;
        system(archiveName.c_str());
}
if (!keys[0] && !keys[5]) {
```

```
                decompressName = "mv␣./" + decompressName + "␣
                    ./" + fileName;
                system(decompressName.c_str());
        }
        return;
}
```

Library.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

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

extern std::vector<bool> keys;

#endif
```

LZ77.h

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

class TLZ77 {
```

```
public:
        typedef struct InitStruct{
                const int compressFloor                 = 2;
                const int comparesCeil                  = 75;
                const int CHARBITS
                          = 8;
                const int MATCHBITS                     = 4;
                const int DICTBITS
                          = 13;
                const int HASHBITS
                          = 10;
                const int SECTORBITS                    = 10;
                const unsigned int MAXMATCH     = (1 <<
                   MATCHBITS) + compressFloor - 1;
                const unsigned int DICTSIZE             = 1 <<
                   DICTBITS;
                const unsigned int HASHSIZE     = 1 << HASHBITS
                   ;
                const unsigned int SHIFTBITS    = (HASHBITS +
                   compressFloor) / (compressFloor + 1);
                const unsigned int SECTORLEN    = 1 <<
                   SECTORBITS;
                const unsigned int SECTORAND    = (0xFFFF <<
                   SECTORBITS) & 0xFFFF;
        } IStruct;
        TLZ77();
        TLZ77(IStruct s){
                dict = (unsigned char*)calloc(DICTSIZE +
                   MAXMATCH, sizeof(char));
                hash = (unsigned int*)calloc(HASHSIZE, sizeof(
                   unsigned int));
                nextlink = (unsigned int*)calloc(DICTSIZE,
                   sizeof(unsigned int));
        };
        void InitEncode();
        bool Compress(std::string in_str, std::string out_str);
        bool Decompress(std::string in_str, std::string out_str
           );
        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 SendChar(unsigned int character);
        void SendMatch(unsigned int matchlen, unsigned int
           matchdistance);
        unsigned int ReadBits(unsigned int numbits);
        void SendBits(unsigned int bits, unsigned int numbits);
        virtual ~TLZ77(){free(dict); free(hash); free(nextlink)
           ;};
        const int compressFloor                 = 2;
        const int comparesCeil                  = 75;
        const int CHARBITS                              = 8;
        const int MATCHBITS                     = 4;
        const int DICTBITS                              = 13;
        const int HASHBITS                              = 10;
        const int SECTORBITS                    = 10;
        const unsigned int MAXMATCH     = (1 << MATCHBITS) +
           compressFloor - 1;
        const unsigned int DICTSIZE             = 1 << DICTBITS
           ;
        const unsigned int HASHSIZE     = 1 << HASHBITS;
        const unsigned int SHIFTBITS    = (HASHBITS +
           compressFloor) / (compressFloor + 1);
        const unsigned int SECTORLEN    = 1 << SECTORBITS;
        const unsigned int SECTORAND    = (0xFFFF << SECTORBITS
           ) & 0xFFFF;
        unsigned char* dict;
        unsigned int *hash, *nextlink;
        unsigned int
                counter = 0,
                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, 32767,
                   65535};
        FILE *infile, *outfile;
};
```

LZ77.cpp

```
#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)

TLZ77::TLZ77() {
        dict = (unsigned char*)calloc(DICTSIZE + MAXMATCH,
           sizeof(char));
        hash = (unsigned int*)calloc(HASHSIZE, sizeof(unsigned
           int));
        nextlink = (unsigned int*)calloc(DICTSIZE, sizeof(
           unsigned int));
}

void TLZ77::SendBits(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;
                counter++;
        }
        return;
}

unsigned int TLZ77::ReadBits(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 TLZ77::SendMatch(unsigned int matchlen, unsigned int
   matchdistance) {
        SendBits(1, 1);
        SendBits(matchlen - (compressFloor + 1), MATCHBITS);
        SendBits(matchdistance, DICTBITS);
        return;
}

void TLZ77::SendChar(unsigned int character) {
        SendBits(0, 1);
        SendBits(character, CHARBITS);
        return;
}

void TLZ77::InitEncode() {
        register unsigned int i;
        for (i = 0; i < HASHSIZE; i++)
                hash[i] = NIL;
        nextlink[DICTSIZE] = NIL;
        return;
}

unsigned int TLZ77::LoadDict(unsigned int dictpos) {
        register unsigned int i, j;
        if ((i = fread(&dict[dictpos], sizeof(char), SECTORLEN,
            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; j++)
                        dict[j + DICTSIZE] = dict[j];
        }
        return i;
}

void TLZ77::DeleteData(unsigned int dictpos) {
        register unsigned int i, j;
        j = dictpos;
        for (i = 0; i < DICTSIZE; i++)
                if ((nextlink[i] & SECTORAND) == j)
                        nextlink[i] = NIL;
        for (i = 0; i < HASHSIZE; i++)
                if ((hash[i] & SECTORAND) == j)
                        hash[i] = NIL;
        return;
}

void TLZ77::HashData(unsigned int dictpos, unsigned int
    bytestodo) {
        register unsigned int i, j, k;
        if (bytestodo <= compressFloor)
                for (i = 0; i < bytestodo; i++)
                        nextlink[dictpos + i] = NIL;
        else {
                for (i = bytestodo - compressFloor; i <
                    bytestodo; i++)
                        nextlink[dictpos + i] = NIL;
                j = (((unsigned int)dict[dictpos]) << SHIFTBITS
                    ) ^ dict[dictpos + 1];
                k = dictpos + bytestodo - compressFloor;

                for (i = dictpos; i < k; i++) {
                        nextlink[i] = hash[j = (((j <<
                            SHIFTBITS) & (HASHSIZE - 1)) ^ dict[
                            i + 2])];
                        hash[j] = i;
                }
```

```
        }
        return;
}

void TLZ77::FindMatch(unsigned int dictpos, unsigned int
   startlen) {
        register unsigned int i, j, k;
        unsigned char l;
        i = dictpos; matchlength = startlen; k = comparesCeil;
        l = dict[dictpos + matchlength];
        do {
                if ((i = nextlink[i]) == NIL)
                        return;
                if (dict[i + matchlength] == l) {
                        for (j = 0; j < MAXMATCH; j++)
                                if (dict[dictpos + j] != dict[i
                                    + j])
                                        break;
                        if (j > matchlength) {
                                matchlength = j;
                                matchpos = i;
                                if (matchlength == MAXMATCH)
                                        return;
                                l = dict[dictpos + matchlength
                                   ];
                        }
                }
        } while (--k);
        return;
}

void TLZ77::DictSearch(unsigned int dictpos, unsigned int
   bytestodo) {
        register unsigned int i, j;
        i = dictpos; j = bytestodo;
        while (j) {
                FindMatch(i, compressFloor);
                if (matchlength > j)
                        matchlength = j;
                if (matchlength > compressFloor) {
                        SendMatch(matchlength, (i - matchpos) &
                            (DICTSIZE - 1));
```

```
                        i += matchlength;
                        j -= matchlength;
                }
                else {
                        SendChar(dict[i]);
                        ++i;
                        j--;
                }
        }
        return;
}

bool TLZ77::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 tmp_char = 'L';
                fwrite(&tmp_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 += SECTORLEN;
                    if (dictpos == DICTSIZE) {
                            dictpos = 0;
                            deleteflag = 1;
                    }
            }
            SendMatch(MAXMATCH + 1, 0);
            if (bitsin)
                    SendBits(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 TLZ77::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;
```

```
for (;;) {
        if (ReadBits(1) == 0) {
                countC++;
                dict[i++] = ReadBits(CHARBITS);
                if (i == DICTSIZE) {
                        if (!keys[5]) {
                                if (fwrite(dict
                                   , sizeof(
                                   char),
                                   DICTSIZE,
                                   outfile) ==
                                   EOF) {
                                        printf(
                                           "\
                                           nerror
                                           ␣
                                           writing
                                           ␣to␣
                                           output
                                           ␣
                                           file
                                           ");
                                        throw
                                           std
                                           ::
                                           runtime_error
                                           ("
                                           Error
                                           ␣
                                           while
                                           ␣
                                           writing
                                           ␣to␣
                                           output
                                           ␣
                                           file
                                           \n")
                                           ;
                                }
                        }
                        i = 0;
```

```
                        bytesdecompressed +=
                           DICTSIZE;
                }
        }
        else {
                k = (compressFloor + 1) +
                   ReadBits(MATCHBITS);
                if (k == (MAXMATCH + 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 - ReadBits(DICTBITS)) &
    (DICTSIZE - 1));
do {
        dict[i++] = dict[j++];
        j &= (DICTSIZE - 1);
        if (i == DICTSIZE) {
                if (!keys[5]) {
                        if (
                           fwrite
                           (
                           dict
                           ,
                           sizeof
                           (
                           char
                           ),
                           DICTSIZE
                           ,
                           outfile
                           ) ==
                            EOF
                           ) {
                                printf
                                   (
                                   "
                                   \
                                   nerror
                                   ␣
                                   writing
                                   ␣
                                   to
                                   ␣
                                   output
                                   ␣
                                   file
                                   "
                                   )
```

```
                                                                ;
                                                            throw
                                                                std
                                                                ::
                                                                runtime
                                                                (
                                                                "
                                                                Error
                                                                ␣
                                                                while
                                                                ␣
                                                                writing
                                                                ␣
                                                                to
                                                                ␣
                                                                output
                                                                ␣
                                                                file
                                                                \
                                                                n
                                                                "
                                                                )
                                                                ;
                                                        }
                                                    }
                                                    i = 0;
                                                    bytesdecompressed
                                                        += DICTSIZE
                                                       ;
                                                }
                                            } 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;
        }
}
```

main.cpp

```
#include "interface.h"

std::vector<bool> keys;

int main(int argc, char *argv[]) {
        clock_t t0 = clock();
        std::map<std::string, int> files;
        keys = {false, false, false, false, false, false, false
           , false, false};
        for (int i = 1; i < argc; ++i) {
                std::string str;
                std::stringstream tmp(argv[i]);
                tmp >> str;
                if (!Parser(&files, str)) {
                        return 1;
                }
        }
        if (files.empty()) {
                std::cout << "Compressed␣data␣not␣written␣to␣a␣
                   terminal.\n";
                return 0;
        }
        FileIterator(files);
        clock_t t1 = clock();
        if (keys[8]) {
                std::cout << "\n" << ((double) t1 - t0) /
                   CLOCKS_PER_SEC << "␣seconds\n";
        }
        return 0;
}
```

Тест производительности

Файл	Размер исходного файла (B)	Алгоритм	Время сжатия (c)	Время декомпрессии (c)	Размер сжатого файла (B)	Коэффициент сжатия
world95.txt	3005020	LZ77	0.5	0.1	1502185	2
world95.txt	3005020	Арифметика	1.3	1.5	1917592	1.6
world95.txt	3005020	оба	1.7	0.1	1502185	2
world95.txt	3005020	gzip	0.4	2.5	878248	3.4
enwik8	100000000	LZ77	14.5	2.8	46965090	2.1
enwik8	100000000	Арифметика	43.3	49.3	62762905	1.6
enwik8	100000000	оба	56.1	2.8	46965090	2.1
enwik8	100000000	gzip	11.2	3.8	36518329	2.7
enwik9	1000000000	LZ77	138	26.4	432608909	2.3
enwik9	1000000000	Арифметика	439.9	543.5	635524001	1.6
enwik9	1000000000	оба	559.9	26.3	432608909	2.3
enwik9	1000000000	gzip	99.1	31.8	323742886	3.1

- Центральный процессор - Mobile DualCore Intel Celeron 1017U, 1600 MHz (16 x 100)
- Графический адаптер - Intel(R) HD Graphics (834742 KB)
- Оперативная память - DDR3-1333 DDR3 SDRAM 2 GB

Выводы

В процессе выполнения данной работы я освоил 2 вида кодирования: арифметическое и LZ77. Это два совершенно разных по сути алгоритма. К примеру, LZ77 стремится сжать файл за счёт каких-либо повторений в нём. Арифметическое кодирование в свою очередь полностью опирается на чтение отдельных символов и их частоту появления.

Зачастую арифметика оказывалась хуже LZ77. На такой результат могло повлиять недостаточно большое количество повторений в тексте. Однако вполне возможно если речь пойдёт о картинках, которые, как мы знаем, сохраняются в виде зачастую повторяющихся последовательностей. В теории это должно сильно повлиять на качество работы алгоритмов, и тогда скорее всего LZ77 отработает лучше арифметики.

Благодаря освоению двух алгоритмов сразу у меня появились представления о рабоче прочих алгоритмов кодирования и стали очевидны различные требования к их работе и результату. Были существенно улучшены навыки работы с файлами: проверка наличия, запись, чтение, перепись.

Список литературы

1. Алгоритм LZ77 [Электронный ресурс]: mf.grsu.by URL: http://mf.grsu.by/UchProc/livak/po/comprsite/theory_lz77.html (дата обращения 10.08.2020)

2. Алгоритмы LZW, LZ77 и LZ78 [Электронный ресурс]: habr.com URL: https://habr.com/ru/post/132683/ (дата обращения 23.08.2020)

3. Арифметическое кодирование [Электронный ресурс]: mf.grsu.by URL: http://mf.grsu.by/UchProc/livak/po/comprsite/theory_arithmetic.html (дата обращения 30.08.2020)

4. Идея арифметического кодирования [Электронный ресурс]: algolist.ru URL: http://algolist.ru/compress/standard/arithm.php (дата обращения 02.09.2020)

5. Arithmetic coding - integer implementation [Электронный ресурс]: stringology.org URL: http://www.stringology.org/DataCompression/ak-int/index_en.html (дата обращения 26.09.2020)