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: В. В. Гринин
Преподаватель: Н. А. Зацепин
Группа: М8О-408Б
Дата:
Оценка:
Подпись:

Москва, 2020

Условие

Необходимо реализовать два известных метода сжатия данных: LZW и Арифметическое кодирование.

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

Обработка входных данных

Для начала программа обрабатывает аргументы командной строки. Таковыми могут быть ключи или наименования файлов и директорий. Чтобы отделять ключи от обычных наименований, в качестве первого символа для ключей используется «-». В ходе изучения работы ключей программы gzip, были выявлены закономерности в их поведении. А именно:

1. Если был введён ключ -t, то далее ключи -c, -k, -d, -1, -9 программа учитывать не будет.

2. Если был введён ключ -l, то он делает ключи -c, -k, -d, -t, -1, -9 недействительными как сейчас, так и при их будущих вводах.

3. В то же время сочетания -cd, -c1, -c9, -kd, -k1, -k9, а так же сочетания ключа -r со всеми остальными ключами не являются взаимоисключающими.

4. Если же во время обработки ключей встречается неизвестный ключ, то программа прекращает свою работу с соответствующей ошибкой, как и утилита gzip.

В случае если начальным символом не является «-», то оно заносится в красное дерево имён файлов/директорий.

Работа с файлами

Проверяется пустота дерева имён файлов/папок – если дерево пустое то программа завершается.

Если дерево не пустое, каждый его элемент сначала рассматривается как директория, затем как файл. Чтобы отделить дерикторию от файла, используется функция `opendir` из библиотеки `dirent.h`.

Если элемент является директорией, то проверяется активность ключа -r: ключ активирован – идёт работа с директорией, нет – пропуск файла. Если элемент является файлом, то никаких проверок не проводится и начинается непосредственная работа с ним.

Подготовка к сжатию

Если у имени файла есть суффикс «.gz», то, если отсутствует ключ -c, файл не обрабатывается. Программа по итогу выведет соответствующее сообщение. Если этого суффикса нет, то, если отсутствует ключ -c, проверяется наличие файла с тем же именем, но при этом ещё и с суффиксом. Если такой файл существует – пользователю предлагают перезаписать этот файл. Если пользователь откажется, то работа с данным файлом прекращается.

Если отсутствуют ключи -1 и -9, то файл сжимается сразу двумя методами. По итогу сжатия получаем два временных файла. Далее сравниваются размеры файлов, где выбирается наименьший, который впоследствии и становится результатом работы программы. Тот файл, который больше, просто удаляется. Если применяется один из этих ключей, то программа использует один из двух алгоритмов. Для ключа -1 это LZW, а для ключа -9 это арифметическое кодирование.

В случае если какой-то из алгоритмов дал сбой, то прекращается работа с файлом и выводится соответствующая ошибка.

Подготовка к распаковке

Для распаковки используется ключ -d. Проверяется наличие у файла суффикса «.gz». Если он не имеется, то, если отсутствует ключ -c, работа с файлом прекращается и выведится соответствующее сообщение. Если имеется, то при отсутствии -t и -c проверяется наличие файла с тем же именем, но без суффикса «.gz». При наличии такого файла, программа запрашивает разрешение на перезапись. В случае отказа завершается работа с файлом. Когда такого файла нет или пользователь дал согласие, то работа продолжается. Каждый архив, сжатый этой программой имеется первые несколько служебных байт, которые содержат в себе информацию о алгоритме сжатия и размере исходного файла. При считывании первого байта определяется алгоритм сжатия. Для LZW это символ «L», для арифметического кодирования это «A». Если это другой символ, то работа с файлом прекращается и выводится уведомление об ошибке. В случае неудачного завершения алгоритма, выводится соответствующее сообщение. При отсутствии ключей -t и -c удаляется файл, в который записывались данные после распаковки, и работа с файлом прекращается. Далее при отсутствии ключей -t, -c и -k, удаляется изначальный архив, а при отсутствии ключей -t и -c временный файл для распаковки переименовывается и получает имя изначального архива без расширения «.gz».

Получение информации об архиве

В случае указания ключа -l производятся следующие действия. Для начала программа считывает первый байт. В случае, если он не совпадает с символами, указывающими на метод сжатия, то выводится сообщение об ошибке и завершается работа с файлом.

Далее читается 8 байт, в которые помещается размер файла до сжатия. После этого программа считывает размер архива, и вычисляется процент сжатия. Далее выво-

дятся размер сжатого файла, размер до компрессии, процент сжатия в полуинтервале $[-100\%; 100\%)$ и имя файла до архивации (если файл имеет расширение «.gz», то имя выводится без этого расширения, в противном случае выводится имя архива).

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

LZW компрессия

Компрессия методом LZW происходит по следующему принципу: из размера файла определяется верхняя граница буфера, в котором будут храниться слова, и каким количеством байт будут кодироваться слова, после чего строится префиксное дерево из всех односимвольных слов–символов ASCII. В архив записывается 9 байт информации, первый из которых - это указание метода архивации, а остальные 8 - размер изначального файла. Далее читается первый символ файла, и в архив записывается код соответствующего слова. Полученный символ указывает на узел, в который будет добавлен следующий символ. Затем символы считываются до создания новой вершины в префиксном дереве, а в архив записывается код вершины, предшествующей новой. Последняя буква, полученная до добавления вершины заносится в буфер. После чего из корня ищется вершина, к которой ведёт эта буква, и процесс повторяется вплоть до окончания символов в файле или создания максимального количества вершин, которое сможет прочитать декомпрессор. Если произошло второе, то в архив записывается 0, (никак иначе на этом этапе он быть записан не может), ранее установленным количеством байт, из префиксного дерева удаляются все вершины кроме корневой и потомков первого рода, после чего процесс компрессии начинается заново, но позиции в исходном файле и архиве не получают откат. Если на каком либо этапе компрессии возникает ошибка, его работа прекращается, и выводится соответствующая ошибка.

LZW декомпрессия

Декомпрессия начинается с прочтения размера изначального файла из архива, который необходим для определения нужно количества байт для прочтения слова и проверки на безошибочность декомпрессии. Далее, из архива считываются только коды слов определённого ранее размера. После считывания первого слова создаётся красно-чёрное дерево, и в него записываются все односимвольные слова из ASCII символов. Далее, по полученному коду в дереве находится необходимая строка, и она записывается в файл. После чего этот символ записывается во временное слово. Далее алгоритм считывает коды из архива. При обработке кодов возможны 4 ситуации:

1. Код входит в список полученных слов. В этом случае в красно-чёрном дереве ищется слово с необходимым кодом, и это слово записывается в файл, после чего в красно-чёрное дерево записывается новое слово, которое является предыдущим словом, к которому добавили первую букву только что полученного. Декомпрессия продолжается.

2. Код не входит в красно-чёрное дерево, но он является следующим по счёту, следовательно это слово можно интерпретировать как предыдущее, к которому дописали в конце букву, с которой оно начинается. Полученное слово записывается в файл и в красно-чёрное дерево, после чего декомпрессия продолжается.

3. Код не входит в красно-чёрное дерево и не является следующим на подходе, следовательно архив повреждён. Декомпрессия прекращается, и выводится соответствующее сообщение.

4. Код равен 0. Красно-чёрное дерево очищается, и процесс декомпрессии начинается сначала, но позиции в файле и архиве не получают откат.

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.

Арифметическое сжатие

В теории арифметическое сжатие описывается достаточно просто. У нас имеется промежуток от 0 до 1. Имеется таблица частот, в которой содержится информация о том, как часто встречается тот или иной символ. Промежуток разделяется на множество отрезков, каждый из которых представляет собой какой-либо символ. При считывании символа, мы переходим к его отрезку. Далее цикл повторяется, но уже с новыми границами, заданными этим символом. На практике же мы неизбежно сталкиваемся с машинным эпсилон, поэтому следует попробовать написать всё в целых числах.

На вход мы получаем файл. Создаём свой файл, в который мы заносим первые 9 байт. 1 байт обозначит тип сжатия, остальные 8 байт - размер исходного файла. По умолчанию у каждого символа частота выставлена на единицу.

Каждый символ кодируется по следующей схеме:

1. Рассчёт границ символа по частоте его появления;

2. Кодировка символа посредством цепочки манипуляций над границами. Если отрезок лежит в верхней половине допустимых значений - пишем бит равный единице. Если лежит в нижней половине - пишем бит равный нулю. Если лежит где-то по центру - увеличиваем счётчик битов, которые будут выставлены вслед за следующим битом с отличным от него значением. Если не выполняется ни одно из этих условий, т.е. получившийся отрезок достаточно большой, то кодировка завершается. Иначе - увеличиваем границы в 2 раза. По сути это аналогично побитовому сдвигу влево.

3. Обновление таблицы частот. Если случилось переполнение, то масштабируем частоты, деля их на два и пересчитывая накопленные частоты. После этого производится сортировка таблицы, чтобы ускорить работу с ней.

Арифметическая распаковка

В теории мы получаем число, в котором закодированы символы. Далее мы определяем в каком отрезке лежит это число, благодаря чему узнаём о закодированном символе. Далее мы выбираем новые границы, а именно границы того отрезка. Разбиваем этот отрезок также на несколько частей, следуя таблице частот и аналогично узнаём следующий символ. Однако в текущей реализации используются целые числа, поэтому и декодирование немного отличается.

В первую очередь мы получаем такое же число. По нему мы также определяем границы, однако способ их нахождения несколько отличается - вместо того, чтобы сразу их узнать, мы сначала находим накопленную частоту и уже по ней определяем границы и закодированный символ. После этого мы по аналогичной схеме, как в кодировании символа, проводим манипуляции над границами и таким образом убираем ненужные биты. Далее мы начинаем декодировать следующий символ.

После декодирования символа мы также обновляем таблицу частот.

Описание файлов программы

Код программы разбит на 13 файлов:

1. ACC.h - Содержит перечисление методов и описание класса TArithmetic, необходимого для работы арифметической компрессии и декомпрессии.
2. ACC.cpp - Содержит реализацию всех методов класса TArithmetic.
3. BFile.h - Содержит перечисление методов и описание класса TOutBinary и класса TInBinary, необходимых для записи в файл и чтения из файла соответственно.
4. BFile.cpp - Содержит реализацию всех методов классов TOutBinary и TInBinary.
5. Globals.h - Содержит в себе все необходимые глобальные переменные и библиотеки используемые несколькими файлами.
6. LZW.h - Содержит перечисление методов и описание класса TLZW, необходимого для работы алгоритма LZW.
7. LZW.cpp - Содержит реализацию всех методов класса TLZW.
8. main_help.h - Содержит в себе перечисление и описание всех функций необходимых для препроцессинга перед началом работы алгоритмов компрессии и декомпрессии.
9. main_help.cpp - Содержит реализацию всех функций, необходимых для препроцессинга, описанных в файле main_help.h.
10. Prefix.h - Содержит перечисление методов и описание класса TPrefix, необходимого для работы LZW компрессии.

11. Prefix.cpp - Содержит реализацию всех методов класса TPrefix.
12. main.cpp - Содержит в себе алгоритм чтения файлов и ключей.
13. Makefile - Файл для сборки программы.

Основные типы данных

1. TACC - класс, описывающий работу арифметического алгоритма компрессии и декомпрессии.
2. TOutBinary - класс обеспечивающий запись необходимого количества байт в файл.
3. TInBinary - класс обеспечивающий считывание необходимого количества байт из файла.
4. TLZW - класс, описывающий работу алгоритма LZW.
5. TPrefix - класс, обеспечивающий построение префиксного дерева для LZW сжатия.

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 буффер;
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 буффер;
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 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 буффер 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 буффер для хранения битов из файла, через него получают биты;

Основные свойства и методы класса TLZW

public:

1. TLZW(TInBinary*, TOutBinary*) - Конструктор класса. Передаются файл для чтения и файл для записи.
2. bool Compress(std::string) - Производит компрессию данных. На вход получает имя файла для компрессии. В случае успешного выполнения возвращает true, иначе false.
3. bool Decompress(std::string) - Производит декомпрессию данных. На вход получает имя файла для декомпрессии. В случае успешного выполнения возвращает true, иначе false.
4. ∼TLZW() - Стандартный деконструктор.

private:

1. TInBinary* ForRead - Файл для чтения.
2. TOutBinary* ForWrite - Файл для записи.
3. TPrefix* CompressionTree - Префиксное дерево для хранения слов при компрессии.
4. std::map<unsigned long long int, std::string> DecompressionTree - Красно-чёрное дерево для хранения слов при декомпрессии.

Основные свойства и методы класса TPrefix

public:

1. TPrefix(TInBinary*, TOutBinary*) - Конструктор для корневой вершины. Передаются файл для чтения и файл для записи.
2. TPrefix() - Конструктор для всех прочих вершин.
3. int Update(char) - Добавление вершины из других вершин. Возвращает коды ошибок или успехов.

4. int UpdateForRoot() - Добавление вершины из корня. Возвращает коды ошибок или успехов.
5. void Clear(bool) - Очистка дерева после переполнения.
6. ~TPrefix() - Стандартный деконструктор.

private:

1. std::vector<std::pair<char, TPrefix*>> Next - Вектор потомков вершины и путей в них.
2. unsigned long long int NumberOfWord - Номер слова в данном узле.
3. static char LastLetter - Последняя прочитанная буква. Необходима для построения нового слова.
4. static unsigned long long int NeedToRead - Вспомогательная переменная для чтения нужного кол-ва символов.
5. static unsigned long long int LastNumber - Номер следующего добавленного слова.
6. static unsigned long long int Border - Максимальная граница количества слов перед очисткой дерева.
7. static TInBinary* ForRead - Файл для чтения.
8. static TOutBinary* ForWrite - Файл для записи
9. static unsigned short int Bites - Количество байт, необходимое для кодирования слова.

Прочие функции

1. bool KeyManager(std::string) - Обрабатывает полученные ключи. В случае получения неизвестного ключа возвращает false, иначе true.
2. bool DifferensOfSizes(TInBinary*, std::string) - вывод для каждого файла размера сжатого, оригинального, коэффициента сжатия(%) и имя оригинального файла(ключ l). В случае повреждения. архива возвращает false, иначе true.
3. void WorkWithDirectory(std::string) - работает с директорией (ключ r).
4. void WorkWithFile(std::string) - работает с файлом (определяет наличие файла, принимает решение о компрессии или декомпрессии, выполняет прочие ключи).
5. bool IsDirectory(std::string, bool) - Проверяет, является ли файл директорией. Если файл является директорией, возвращает true, иначе false.

6. void PrintDirectoryErrors(std::string) - Уведомляет об ошибках.
7. bool IsArchive(std::string) - Проверяет, является ли файл архивом. Если файл является архивом, возвращает true, иначе false.
8. void Rename(std::string, std::string) - Изменяет название файла после успешной компрессии или декомпрессии.
9. void Delete(std::string) - Удаляет временный файл.
10. void MainDecompress(TInBinary*, std::string) - Отвечает за подготовку декомпрессинга.
11. void MainCompress(TInBinary*, std::string) - Отвечает за подготовку компрессинга.
12. unsigned long long int LZWCompress(TInBinary*, std::string, TOutBinary*) - Подготавливает LZW компрессию. Возвращает размер нового файла.
13. unsigned long long int ArithmeticCompress(TInBinary*, std::string) - Подготавливает арифметический компрессию. Возвращает размер нового файла.
14. void KeepSmall(unsigned long long int, unsigned long long int,unsigned long long int, std::string) - Сохраняет архив самого малого размера.
15. int main(int, char*) - Осуществляет чтение входных данных.

Исходный код

ACC.cpp

```
#include "ACC.h"

TACC::TACC () {

    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 TACC::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 TACC::StartInputingBits () {
    bitsToGo = 0;
    garbageBits = 0;
}

//-----------------------------------------------------------------
//
int TACC::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 TACC::StartOutputingBits () {
    buffer = 0;
    bitsToGo = 8;
}

//-----------------------------------------------------------------
```

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

//-----------------------------------------------------------------
//
void TACC::OutputBitPlusFollow (int bit) {
    OutputBit (bit);
    while (bitsToFollow > 0) {
        OutputBit (!bit);
        --bitsToFollow;
    }
}
```

```
//-----------------------------------------------------------------
//
void TACC::StartEncoding () {
    low             = 0l;
    high            = TOP_VALUE;
    bitsToFollow    = 0l;
}

//-----------------------------------------------------------------
//
void TACC::DoneEncoding () {

    ++bitsToFollow;
    if (low < FIRST_QTR) {
        OutputBitPlusFollow(0);
    }
    else {
        OutputBitPlusFollow(1);
    }
}

//-----------------------------------------------------------------
/*

*/
void TACC::StartDecoding () {

    value = 0l;
    for ( int i = 0; i < BITS_IN_REGISTER; ++i) {
        value = 2 * value + InputBit ();
    }
    low = 0l;
    high = TOP_VALUE;
}

//-----------------------------------------------------------------
//
void TACC::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 TACC::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 TACC::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 TACC::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]) {
        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;
}
```

ACC.h

```
#ifndef ACC_H
#define ACC_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;
//                                      EOF
const long          EOF_SYMBOL  = NO_OF_CHARS + 1;
//
const long          NO_OF_SYMBOLS = NO_OF_CHARS + 1;

//
const long          MAX_FREQUENCY = 16383;
```

```
class TACC {
    public:

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

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

        //
        TACC();


    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 /* ACC_H */
```

BFile.cpp

```
/* 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;
}
```

BFile.h

```
/* BFile.h */
#ifndef BFILE_H
#define BFILE_H

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

/*                          .
 */
class TOutBinary {
    public:

        /*                      :
         */
        TOutBinary();

        /*                      :                       .
         *                              - true.         - false.
         */
        bool        Open(std::string* name);

        /*                      :                       .
         *                              - true.         - false.
         */
        bool        Close();

        /*                      :
         *                              - true.         - false.
         */
        bool        Write(const char* obj, size_t size);
```

```
        /*                      :
         *                              - true.          - false.
         */
        bool        WriteBin(size_t bit);

        /*                      :
         */
        unsigned long long SizeFile();

        /*                      :
         *                              - true.          - false.
         */
        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();

        /*                      :                              .
         *                              - true.
         *            - false.
         */
        bool        Open(std::string* name);

        /*                      :                              .
         *                              - true.
```

```
         *             - false.
         */
        bool         Close();

        /*                    :                                    size
         *                                 -                      true.
         *                                   -                      false.
         */
        bool         Read(char* obj, size_t size);

        /*                    :
         *                                 -                      false.
         */
        bool         ReadBin(char* bit);

        /*                    :
         */
        unsigned long long SizeFile();

        /*                    :
         *                                 - true.
         *             - false.
         */
        friend bool operator >> (TInBinary& iFile, char &bit);
    private:

        std::ifstream    in;

        std::string      name;
        unsigned char    head;
        unsigned char    block;
};

#endif
```

Globals.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
```

LZW.cpp

```
#include "LZW.h"

TLZW::TLZW(TInBinary* from, TOutBinary* to) {
        this->ForRead = from;
        this->ForWrite = to;
        this->CompressionTree = new TPrefix(from, to);
        return;
}

bool TLZW::Compress(std::string fileName) {
        char method = 'L';
        unsigned long long int size = this->ForRead->SizeFile();
        if (keys[0]) {
                std::cout << method << size;
        }
        else if (!keys[5]) {
                if (!this->ForWrite->Write(&method, sizeof(char))) {
                        std::cout << std::endl << "\t\t" << fileName << "
                        return false;
                }
                if (!this->ForWrite->Write((char*)&size, sizeof(unsigned
                        std::cout << std::endl << "\t\t" << fileName << "
                        return false;
                }
        }
        if (size == 0) {
                return true;
        }
        int bufferState = this->CompressionTree->UpdateForRoot();
        while (bufferState == FULL) {
```

```
                this->CompressionTree->Clear(true);
                bufferState = this->CompressionTree->UpdateForRoot();
        }
        if (bufferState == MEMORY_ERROR) {
                std::cout << std::endl << "\t\t" << fileName << ": unexpe
                return false;
        }
        else if (bufferState == WRITE_ERROR) {
                std::cout << std::endl << "\t\t" << fileName << ": can't
                return false;
        }
        else if (bufferState == READ_ERROR) {
                std::cout << std::endl << "\t\t" << fileName << ": bad in
                return false;
        }
        return true;
}

bool TLZW::Decompress(std::string fileName) {
        unsigned long long int letter;
        std::string previousWord, presentWord;
        unsigned short int bites;
        if (!this->ForRead->Read((char*)&letter, sizeof(unsigned long lon
                std::cout << fileName << ": can't read file" << std::endl
                return false;
        }
        if (letter < std::numeric_limits<unsigned short int>::max()) {
                bites = sizeof(short int);
        }
        else if (letter < std::numeric_limits<unsigned int>::max()) {
                bites = sizeof(int);
        }
        else {
                bites = sizeof(long long int);
        }
        const unsigned long long int fileSize = letter;
        if (fileSize == 0) {
                return true;
        }
        std::map<unsigned long long int, std::string>::iterator finder;
        unsigned long long int wordCounter = CHAR_HAS;
        unsigned long long int alreadyRead = 0;
```

```
while (this->ForRead->Read((char*)&letter, bites)) {
        for (int i = 0; i < CHAR_HAS; ++i) {
                this->DecompressionTree.insert({i + 1, std::string
        }
        ++alreadyRead;
        finder = this->DecompressionTree.find(letter);
        previousWord = finder->second;
        if (keys[0]) {
                std::cout << previousWord;
        }
        else if (!keys[5]) {
                if (!this->ForWrite->Write((char*)previousWord.c_
                        std::cout << fileName << ": can't write i
                        return false;
                }
        }
        if (alreadyRead == fileSize) {
                return true;
        }
        while (this->ForRead->Read((char*)&letter, bites)) {
                if (letter == 0) {
                        break;
                }
                finder = this->DecompressionTree.find(letter);
                if (finder != this->DecompressionTree.end()) {
                        if (keys[0]) {
                                std::cout << finder->second;
                        }
                        else if (!keys[5]) {
                                if (!this->ForWrite->Write((char*
                                        std::cout << fileName <<
                                        return false;
                                }
                        }
                        alreadyRead += finder->second.size();
                        if (alreadyRead == fileSize) {
                                return true;
                        }
                        presentWord = previousWord + finder->seco
                        previousWord = finder->second;
                }
                else if (letter == wordCounter + 1) {
```

```
                                presentWord = previousWord + previousWord
                                alreadyRead += presentWord.size();
                                if (keys[0]) {
                                        std::cout << presentWord;
                                }
                                else if (!keys[5]) {
                                        if (!this->ForWrite->Write((char*
                                                std::cout << fileName <<
                                                return false;
                                        }
                                }
                                if (alreadyRead == fileSize) {
                                        return true;
                                }
                                previousWord = presentWord;
                        }
                        else {
                                std::cout << fileName << ":␣invalid␣compr
                                return false;
                        }
                        this->DecompressionTree.insert({wordCounter + 1, 
                        ++wordCounter;
                }
                this->DecompressionTree.clear();
                if (letter != 0) {
                        letter = 1;
                        break;
                }
                wordCounter = CHAR_HAS;
        }
        if (alreadyRead < fileSize) {
                std::cout << fileName << ":␣unexpected␣end␣of␣file" << st
                return false;
        }
        return true;
}

TLZW::~TLZW() {
        delete this->CompressionTree;
}
```

LZW.h

```
#pragma once
#include "Prefix.h"
#include "Globals.h"

class TLZW {
public:

        TLZW(TInBinary*, TOutBinary*);

        bool Compress(std::string);

        bool Decompress(std::string);

        ~TLZW();

private:

        TInBinary* ForRead;

        TOutBinary* ForWrite;

        TPrefix* CompressionTree;

        std::map<unsigned long long int, std::string> DecompressionTree;
};
```

Prefix.cpp

```
#include "Prefix.h"

unsigned long long int TPrefix::Border;
TInBinary* TPrefix::ForRead;
TOutBinary* TPrefix::ForWrite;
unsigned char TPrefix::LastLetter;
unsigned long long int TPrefix::LastNumber;
unsigned long long int TPrefix::NeedToRead;
unsigned short int TPrefix::Bites;

TPrefix::TPrefix(TInBinary* from, TOutBinary* to) {
        if (keys[1]) {
```

```
                return;
        }
        std::map<unsigned long long int, std::string> tree;
        unsigned long long int highBorder = tree.max_size();
        this->Border = highBorder;
        if (highBorder == 0) {
                return;
        }
        this->ForRead = from;
        this->ForWrite = to;
        highBorder = this->ForRead->SizeFile();
        this->NeedToRead = highBorder;
        if (highBorder < std::numeric_limits<unsigned short int>::max())
                this->Bites = sizeof(unsigned short int);
        }
        else if (highBorder < std::numeric_limits<unsigned int>::max()) {
                this->Bites = sizeof(unsigned int);
        }
        else {
                this->Bites = sizeof(unsigned long long int);
        }
        this->NumberOfWord = 0;
        this->LastLetter = 0;
        this->LastNumber = 1;
        for (int i = 0; i < CHAR_HAS; ++i) {
                this->Next.push_back({(unsigned char) i, new TPrefix()});
        }
}

TPrefix::TPrefix() {
        this->NumberOfWord = this->LastNumber;
        ++this->LastNumber;
}

int TPrefix::Update(unsigned char letter) {
        bool needNew = true;
        for (int i = 0; i < this->Next.size(); ++i) {
                if (letter == this->Next[i].first) {
                        needNew = false;
                        if (this->NeedToRead > 0) {
                                if (!this->ForRead->Read((char*)&letter,
                                        return READ_ERROR;
```

```
                                }
                                --this->NeedToRead;
                                return this->Next[i].second->Update(lette
                        }
                        else {
                                if (keys[0]) {
                                        std::cout << (char*)&this->Next[i
                                }
                                if (!keys[5]) {
                                        if (!this->ForWrite->Write((char*
                                                return WRITE_ERROR;
                                        }
                                }
                                return GOT_EOF;
                        }
                }
        }
        if (needNew) {
                this->Next.push_back({letter, new TPrefix()});
                if (this->Next.back().second == nullptr) {
                        return MEMORY_ERROR;
                }
                this->LastLetter = letter;
                if (keys[0]) {
                        std::cout << (char*)&this->NumberOfWord;
                }
                else if (!keys[5]) {
                        if (!this->ForWrite->Write((char*)&this->NumberOfW
                                return WRITE_ERROR;
                        }
                }
        }
        return OK;
}

int TPrefix::UpdateForRoot() {
        this->LastNumber = CHAR_HAS + 1;
        unsigned char letter, startLetter;
        unsigned long long int tmpInt = 0;
        if (!this->ForRead->Read((char*)&letter, sizeof(unsigned char))) {
                if (this->NeedToRead != 0) {
                        return READ_ERROR;
```

```
        }
        return GOT_EOF;
}
--this->NeedToRead;
while (true) {
        if (this->NeedToRead > 0) {
                startLetter = letter;
                if (this->LastNumber < this->Border) {
                        if (!this->ForRead->Read((char*)&letter,
                                return READ_ERROR;
                        }
                        --this->NeedToRead;
                        tmpInt = this->Next[startLetter].second->
                        if (tmpInt != OK) {
                                return tmpInt;
                        }
                        letter = this->LastLetter;
                }
                else {
                        tmpInt = 0;
                        if (keys[0]) {
                                std::cout << (char*)&this->Next[l
                        }
                        else if (!keys[5]) {
                                if (!this->ForWrite->Write((char*
                                        return WRITE_ERROR;
                                }
                                if (!this->ForWrite->Write((char*
                                        return WRITE_ERROR;
                                }
                        }
                }
        }
        else {
                if (keys[0]) {
                        std::cout << (char*)&this->Next[letter].s
                }
                else if (!keys[5]) {
                        if (!this->ForWrite->Write((char*)&this->
                                return WRITE_ERROR;
                        }
                }
```

```
                        return GOT_EOF;
                }
        }
}

void TPrefix::Clear(bool root) {
        for (int i = 0; i < this->Next.size(); ++i) {
                if (root) {
                        this->Next[i].second->Clear(false);
                }
                else {
                        delete this->Next[i].second;
                }
        }
        if (!root) {
                this->Next.clear();
                this->Next.shrink_to_fit();
        }
        return;
}

TPrefix::~TPrefix() {
        for (int i = 0; i < this->Next.size(); ++i) {
                delete this->Next[i].second;
        }
}
```

Prefix.h

```
#ifndef PREFIX_H
#define PREFIX_H

#include "Globals.h"
#include "BFile.h"
#include <limits>

const short int CHAR_HAS = 256;

enum UpdateResult {
    GOT_EOF,
    OK,
    FULL,
```

```
    MEMORY_ERROR,
    WRITE_ERROR,
    READ_ERROR
};

class TPrefix {
public:

        TPrefix(TInBinary*, TOutBinary*);

        TPrefix();

        int Update(unsigned char);

        int UpdateForRoot();

        void Clear(bool);

        ~TPrefix();

private:

        std::vector<std::pair<unsigned char, TPrefix*>> Next;

        unsigned long long int NumberOfWord;

        static unsigned char LastLetter;

        static unsigned long long int NeedToRead;

        static unsigned long long int LastNumber;

        static unsigned long long int Border;

        static TInBinary* ForRead;

        static TOutBinary* ForWrite;

        static unsigned short int Bites;
};

#endif
```

main.cpp

```
#include "main_help.h"

std::vector<bool> keys;

int main(int argc, char *argv[]) {
        clock_t t0 = clock();
        std::map<std::string, int> fileNames;
        std::map<std::string, int>::iterator finder;
        keys = {false, false, false, false, false, false, false, false, f
        for (int i = 1; i < argc; ++i) {
                std::string keyFile;
                std::stringstream tmp(argv[i]);
                tmp >> keyFile;
                if (keyFile[0] == '-') {
                        if (!KeyManager(keyFile)) {
                                return -1;
                        }
                }
                else {
                        bool got = false;
                        if (keyFile.size() > 2) {
                                if (keyFile[0] == '.' && keyFile[1] == '/
                                        keyFile.erase(0, 2);
                                }
                        }
                        finder = fileNames.find(keyFile);
                        if (finder != fileNames.end()) {
                                got = true;
                        }
                        if (!got) {
                                fileNames.insert({keyFile, i});
                        }
                }
        }
        if (fileNames.empty()) {
                std::cout << "Compressed␣data␣not␣written␣to␣a␣terminal."
                return 0;
        }
        for (finder = fileNames.begin(); finder != fileNames.end(); ++fin
                bool directory = IsDirectory(finder->first, false);
                if (!keys[4] && directory) {
```

```
                        std::cout << finder->first << " is a directory --
                }
                else if (directory) {
                        WorkWithDirectory(finder->first);
                }
                else if (finder->first == "main") {
                        continue;
                }
                else {
                        if (errno == ENOTDIR) {
                                WorkWithFile(finder->first);
                        }
                        else {
                                PrintDirectoryErrors(finder->first);
                        }
                }
        }
        clock_t t1 = clock();
        if (keys[8]) {
                std::cout << std::endl << ((double) t1 - t0) / CLOCKS_PER
        }
        return 0;
}
```

main_help.cpp

```
#include "main_help.h"

bool KeyManager(std::string gotKeys) {
    for (int j = 1; j < gotKeys.size(); ++j) {
        switch (gotKeys[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 -- '" << gotKeys[j] << "'" <
                return false;
        }
    }
    return true;
}

bool DifferensOfSizes(TInBinary* file, std::string fileName) {
    char trash = 0;
    if (!file->Read(&trash, sizeof(char))) {
        return false;
    }
    if (trash != 'L' && trash != 'A') {
        return false;
    }
    unsigned long long int uncompressed, compressed;
    if (!file->Read((char*)&uncompressed, sizeof(unsigned long long int))
        return false;
    }
    compressed = file->SizeFile();
    double ratio = (double) compressed / uncompressed;
    ratio = 1 - ratio;
    if (uncompressed == 0) {
        ratio = 0.0;
    }
    else if (ratio < -1) {
        ratio = -1;
    }
    std::cout << "          compressed          uncompressed  ratio uncompre
```

```
    printf("%19llu␣%19llu␣%5.1lf", compressed, uncompressed, ratio * 100)
    if (IsArchive(fileName)) {
        fileName.pop_back();
        fileName.pop_back();
        fileName.pop_back();
    }
    std::cout << "%␣" << fileName << std::endl;
    return true;
}

void WorkWithDirectory(std::string directoryName) {
        DIR *directory = opendir(directoryName.c_str());
        if (directory == NULL) {
                PrintDirectoryErrors(directoryName);
                return;
        }
        short int countOfDirs = 0;
        for (int i = 0; i < directoryName.size(); ++i) {
                if (directoryName[i] == '/') {
                        ++countOfDirs;
                }
                if (countOfDirs > 1) {
                        break;
                }
        }
        struct dirent *directoryFile;
        directoryFile = readdir(directory);
        while (directoryFile) {
                if (errno == EBADF) {
                        std::cout << directoryName << ":␣something␣wrong"
                        return;
                }
                std::string tmp = std::string(directoryFile->d_name);
                if (tmp == "." || tmp == ".." || (tmp == "main" && countO
                        && (directoryName[0] == directoryName[1]) && dire
                {
                        directoryFile = readdir(directory);
                        continue;
                }
                if (directoryName.back() == '/') {
                        tmp = directoryName + tmp;
                }
```

```
                else {
                        tmp = directoryName + "/" + tmp;
                }
                if (tmp == "./main") {
                        directoryFile = readdir(directory);
                        continue;
                }
                if (IsDirectory(tmp, true)) {
                        WorkWithDirectory(tmp);
                }
                else {
                        WorkWithFile(tmp);
                }
                directoryFile = readdir(directory);
        }
        closedir(directory);
        if (errno == EBADF) {
                PrintDirectoryErrors(directoryName);
        }
        return;
}

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

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

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

void PrintDirectoryErrors(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 " << direc
                      << "\t Try it next time" << std::endl;
            break;
        case EMFILE:
            std::cout << "Too many files opened in system. Can't open dir
                      << "\t Try it next time" << std::endl;
            break;
        case ENOMEM:
            std::cout << "Not enought memory for opening directory " << d
                      << "\t Try it next time" << std::endl;
```

```
            break;
        case ENOENT:
            std::cout << "No␣file␣or␣directory␣with␣name␣" << directoryNa
            break;
    }
    return;
}

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

void Rename(std::string oldName, std::string nextName) {
    std::string command = "mv␣./" + oldName + "␣./" + nextName;
    system(command.c_str());
    return;
}

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

void MainDecompress(TInBinary* file, std::string fileName) {
        if (keys[1] && !keys[0]) {
                if (!IsArchive(fileName)) {
                        std::cout << fileName << ":␣unknown␣suffix␣--␣ign
                        return;
                }
        }
        std::string tmpName = fileName + ".tmp";
        file->Close();
        std::string nextName = fileName;
        nextName.pop_back();
        nextName.pop_back();
        nextName.pop_back();
        if (!keys[0] && !keys[5]) {
                if (file->Open(&nextName)) {
                        std::cout << nextName << "␣is␣already␣exists;␣do␣
                        char choise;
```

```
                std::cin >> choise;
                file->Close();
                if (choise != 'Y' && choise != 'y') {
                        std::cout << "\tnot overwritten" << std::
                        return;
                }
        }
}
if (!file->Open(&fileName)) {
        std::cout << fileName << ": can't read file" << std::endl
        return;
}
TOutBinary* decompressedFile = nullptr;
if (!keys[0] && !keys[5]) {
        decompressedFile = new TOutBinary;
        if (decompressedFile == nullptr) {
                std::cout << fileName << ": unexpected memory err
                return;
        }
        if (!decompressedFile->Open(&tmpName)) {
                std::cout << fileName << ": can't transfer data"
                delete decompressedFile;
                return;
        }
}
char algorithm = 0;
bool success;
if (!file->Read(&algorithm, sizeof(char))) {
        std::cout << fileName << ": can't transfer data" << std::
        if (!keys[0] && !keys[5]) {
                decompressedFile->Close();
                Delete(tmpName);
                delete decompressedFile;
        }
        return;
}
if (algorithm == 'A') {
        TACC* method = new TACC;
        if (method == nullptr) {
                std::cout << fileName << ": unexpected memory err
                if (!keys[0] && !keys[5]) {
                        decompressedFile->Close();
```

```
                        Delete(tmpName);
                        delete decompressedFile;
                }
                return;
        }
        success = method->Decompress(fileName.c_str(), tmpName.c_
        delete method;
}
else if (algorithm == 'L') {
        TLZW* method = new TLZW(file, decompressedFile);
        if (method == nullptr) {
                std::cout << fileName << ": unexpected memory err
                if (!keys[0] && !keys[5]) {
                        decompressedFile->Close();
                        Delete(tmpName);
                        delete decompressedFile;
                }
                return;
        }
        success = method->Decompress(fileName);
        delete method;
}
else {
        std::cout << fileName << ": not compressed data" << std::
        if (!keys[0] && !keys[5]) {
                decompressedFile->Close();
                Delete(tmpName);
                delete decompressedFile;
        }
        return;
}
if (!keys[0] && !keys[5]) {
        decompressedFile->Close();
        delete decompressedFile;
}
if (!success) {
        if (!keys[5]) {
                std::cout << "\t\tdecompressing failed" << std::e
        }
        if (!keys[0] && !keys[5]) {
                Delete(tmpName);
        }
```

```
                return;
        }
        if (keys[5]) {
                if (algorithm != 'L') {
                        Delete(tmpName);
                }
                return;
        }
        if (!keys[0] && !keys[2]) {
                Delete(fileName);
        }
        if (!keys[0]) {
                Rename(tmpName, nextName);
        }
        return;
}

void MainCompress(TInBinary* file, std::string fileName) {
        std::string nextName = fileName + ".gz";
        file->Close();
        if (IsArchive(fileName)) {
                std::cout << fileName << " already has .gz suffix -- unch
                return;
        }
        if (!keys[0]) {
                if (file->Open(&nextName)) {
                        std::cout << nextName << " is already exists; do
                        char choise;
                        std::cin >> choise;
                        file->Close();
                        if (choise != 'Y' && choise != 'y') {
                                std::cout << "\tnot overwritten" << std::
                                return;
                        }
                }
        }
        TOutBinary* compressionFile = nullptr;
        if (!keys[0]) {
                compressionFile = new TOutBinary;
                if (compressionFile == nullptr) {
                        std::cout << fileName << ": unexpected memory err
                        return;
```

```
        }
}
unsigned long long int LZWSize, arithmeticSize;
if (!keys[6] && !keys[7]) {
        LZWSize = LZWCompress(file, fileName, compressionFile);
        file->Close();
        if (LZWSize == 0) {
                if (!keys[0]) {
                        delete compressionFile;
                        Delete(fileName + ".LZW");
                }
                return;
        }
        arithmeticSize = ArithmeticCompress(fileName, file);
        if (arithmeticSize == 0) {
                if (!keys[0]) {
                        delete compressionFile;
                        Delete(fileName + ".LZW");
                        Delete(fileName + ".ARI");
                }
                return;
        }

}
else if (keys[6]) {
        arithmeticSize = 0;
        LZWSize = LZWCompress(file, fileName, compressionFile);
        file->Close();
        if (LZWSize == 0) {
                if (!keys[0]) {
                        delete compressionFile;
                        Delete(fileName + ".LZW");
                }
                return;
        }
}
else {
        LZWSize = 0;
        arithmeticSize = ArithmeticCompress(fileName, file);
        if (arithmeticSize == 0) {
                if (!keys[0]) {
                        delete compressionFile;
```

```
                                        Delete(fileName + ".ARI");
                                }
                                return;
                        }
                }
                if (!keys[0]) {
                        delete compressionFile;
                        KeepSmall(LZWSize, arithmeticSize, fileName);
                }
                return;
}

unsigned long long int LZWCompress(TInBinary* file, std::string fileName,
    std::string LZWName = fileName + ".LZW";
    if (!file->Open(&fileName)) {
        std::cout << fileName << ": can't read file" << std::endl;
        return 0;
    }
    if (!keys[0]) {
        if (!compressionFile->Open(&LZWName)) {
            std::cout << fileName << ": can't transfer data" << std::endl
            return 0;
        }
    }
    TLZW* method = new TLZW(file, compressionFile);
    if (method == nullptr) {
        std::cout << fileName << ": unexpected memory error" << std::endl
        return 0;
    }
    if (!method->Compress(fileName)) {
        delete method;
        std::cout << "\t\tcompression failed" << std::endl;
        return 0;
    }
    delete method;
    if (!keys[0]) {
        compressionFile->Close();
    }
    if (keys[0]) {
        return 1;
    }
    file->Close();
```

```
    if (!file->Open(&LZWName)) {
        std::cout << fileName << ": can't read file" << std::endl;
        return 0;
    }
    return file->SizeFile();
}

unsigned long long int ArithmeticCompress(std::string fileName, TInBinary
    std::string arithmeticName  = fileName + ".ARI";
    TACC* method = new TACC;
    if (method == nullptr) {
        std::cout << fileName << ": unexpected memory error" << std::endl
        return 0;
    }
    if (!method->Compress(fileName.c_str(), arithmeticName.c_str())) {
        delete method;
        std::cout << "\t\tcompression failed" << std::endl;
        return 0;
    }
    delete method;
    if (keys[0]) {
        return 1;
    }
    if (!file->Open(&arithmeticName)) {
        std::cout << fileName << ": can't read file" << std::endl;
        return 0;
    }
    return file->SizeFile();
}

void KeepSmall(unsigned long long int LZWSize, unsigned long long int ari
    std::string nextName;
    if (LZWSize > 0 && arithmeticSize > 0) {
        if (LZWSize > arithmeticSize) {
            nextName = fileName + ".ARI";
            Delete(fileName + ".LZW");
        }
        else {
            nextName = fileName + ".LZW";
            Delete(fileName + ".ARI");
        }
        Rename(nextName, fileName + ".gz");
```

```
    }
    else {
        if (LZWSize == 0) {
            Rename(fileName + ".ARI", fileName + ".gz");
        }
        else {
            Rename(fileName + ".LZW", fileName + ".gz");
        }
    }
    if (!keys[2]) {
        Delete(fileName);
    }
    return;
}
```

main_help.h

```
#ifndef MAIN_HELP_H
#define MAIN_HELP_H

#include "Globals.h"
#include "LZW.h"
#include "ACC.h"
#include <cstdio>
#include <cstdlib>

bool KeyManager(std::string);

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

void WorkWithDirectory(std::string);

void WorkWithFile(std::string);

bool IsDirectory(std::string, bool);

void PrintDirectoryErrors(std::string);

bool IsArchive(std::string);

void Rename(std::string, std::string);
```

```
void Delete(std::string);

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

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

unsigned long long int LZWCompress(TInBinary*, std::string, TOutBinary*);

unsigned long long int LZ77Compress(TInBinary* file, std::string, TOutBina

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

void KeepSmall(unsigned long long int, unsigned long long int, std::string

#endif
```

Тест производительности

Для начала рассмотрим эффективность работы утилиты gzip.

Файл	Размер исходного файла	Алгоритм	Время сжатия (с)	Время «разжатия» (с)	Размер сжатого файла	Коэффициент сжатия
world95.txt	3006464	gzip	0.158	0.072	880640	3.414
enwik8	100003840	gzip	6.417	0.887	36519936	2.738
enwik9	1000005632	gzip	50.881	11.889	323743744	3.089

Пока что-то сказать сложно, но ясно одно - утилита выполняет свою работу быстро и корректно. Теперь к курсовой работе.

Файл	Размер исходного файла	Алгоритм	Время сжатия (с)	Время «разжатия» (с)	Размер сжатого файла	Коэффициент сжатия
world95.txt	3006464	Оба алгоритма	1.504	0.897	1716224	1.752
enwik8	100003840	Оба алгоритма	65.907	37.150	48697344	2.054
enwik9	1000005632	Арифметика	282.655	280.363	635531264	1.573

Небольшое пояснение по алгоритмам. В случае первых двух файлов программа проходила по двум алгоритмам и выбирала тот, что лучше отработал. Однако там в обоих случаях хорошо себя показала арифметика. В первую очередь тут могло сказаться, что в тексте последовательности не так часто повторяются, как хотелось бы. Для арифметики наличие каких-то повторяющихся последовательностей не важно, однако ей выгодно встречать как можно больше и чаще относительно маленький набор симво-

лов. Думаю, LZW лучше будет работать для сжатия изображений, так как там больше шансов встретить повторяющиеся последовательности.

Помимо этого можно заметить, что на последнем файле используется только арифметика. Причина в том, что LZW использует дерево, которое неплохо так разрослось, из-за чего ОС в целом тормозила. Пришлось экстренно прекратить программу и сжать только при помощи арифметики, так как она не использует динамические структуры.

Если сравнивать с самим gzip, то очевидно, что он сжимает гораздо лучше и быстрее. Особенно это чувствуется с файлами огромного размера. Также из интересного можно заметить, что на файле enwik8 распаковка у gzip проходила крайне быстро, особенно на фоне программы из курсовой работы.

Тесты проводились на ноутбуке со следующими техническими характеристиками:

- Центральный процессор: Intel® Core™ i5-7200U CPU @ 2.50GHz x 4
- Графический адаптер: Mesa Intel® HD Graphics 620 (KBL GT2)
- Оперативная память: 8 ГБ.

Выводы

В ходе выполнения работы была написана программа, взаимодействующая с файловой системой, а также реализующая LZW и арифметическое сжатие.

Оба алгоритма с точки зрения сложности концепции одинаково просты. Однако когда речь доходит до реализации, то картина меняется.

В случае арифметического кодирования если мы будем пытаться написать его на основе чисел с плавающей точкой, то мы сталкиваемся с двумя проблемами:

- Числа с плавающей точкой сами по себе занимают большое количество бит, что отражается на том минимуме символов, который необходим, чтобы файл в целом стал меньше.
- Может сильно помешать машинный эпсилон. Из-за него порой файлы не получалось нормально сжать.

По этим причинам пришлось искать целочисленный способ кодирования, который в свою очередь уже сложнее в реализации и менее очевиден для понимания. Но это в лучшую сторону сказалось на эффективности сжатия.

В случае LZW единственную сложность вызвала реализация префиксного дерева. А по остальному всё довольно просто.

Список литературы

1. Арифметическое кодирование [Электронный ресурс]: mf.grsu.by URL: http://mf.grsu.by/UchProc/livak/po/comprsite/theory_arithmetic.html (дата обращения 14.07.2020)

2. Арифметическое кодирование [Электронный ресурс]: habr.com URL: https://habr.com/ru/post/130531/ (дата обращения 26.08.2020)

3. Алгоритм LZW [Электронный ресурс]: mf.grsu.by URL: http://mf.grsu.by/UchProc/livak/po/comprsite/theory_lzw.html (дата обращения 26.08.2020)