Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Разработка архиватора (Huffman + LZ77)

Студент: А. В. Скворцов
Преподаватель: А. А. Журавлёв
Группа: М8О-208Б
Дата:
Оценка:
Подпись:

Москва, 2018

1 Цели и задачи

Цели курсового проекта:

1. Закрепление теоретического материала, полученного в ходе курса
2. Проведение исследования в выбранной предметной области
3. Приобретение практических навыков в использовании знаний

Задание: Реализовать архиватор файлов, использующий комбинацию методов сжатия Huffman и LZ77. 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.

2 Теоретические сведения

LZ77 - словарный метод сжатия без потерь, разработанный Абрахамом Лемпелем и Якобом Зивом в 1977 году.

В его основе лежит простая идея: в тексте часто встречаются повторяющиеся подстроки(например слова). Если попытаться от них избавиться и делать ссылки на первую такую подстроку, можно заметно сократить размер исходного файла. Такие ссылки можно задать парой чисел (a, b), где a говорит, сколько символов назад было предыдущее упоминание этой подстроки, а b указывает на длину этого повторения.

Весь процесс сжатия выглядит следующим образом: считываем строку до тех пор, пока она повторяется в обработанном тексте, и если её длина больше определенного значения L , записываем ссылку в выходной поток, иначе — саму строку. Тогда закодированный текст состоит из ссылок и символов.

Декодирование же максимально просто: если считали символ, то записываем его в выходной поток, а если ссылку, то возвращаемся назад в раскодированной строке и выводим заданное количество символов на выход.

Очевидно, что для быстрой работы нужно ограничить размер текста, в котором мы будем искать повторения, а значение L нужно, чтобы размер ссылки не превышал размер закодированного текста. Если поиск подстроки осуществляется за константу, то общее время работы алгоритма линейное.

Алгоритм Хаффмана — алгоритм оптимального префиксного кодирования алфавита, разработанный в 1952 году Дэвидом Хаффманом.

Идея алгоритма заключается в следующим: каждый символ текста кодируется одинаковым количеством бит, однако некоторые из них встречаются чаще других, и зная такую статистику, мы можем дать символам с большей вероятностью появления более короткие коды, а с меньшей вероятностью — более длинные. Доказано,

что можно найти новые коды так, чтобы общая длина текста по крайней мере не увеличилась.

Особенностью таких кодов является то, что ни один из них не является началом другого, что позволяет нам однозначно декодировать сжатый текст.

Вычислить новые последовательности битов можно по следующему алгоритму:

1. Составим список кодируемых символов с весом, равным частоте появления символа в строке
2. Из списка выберем два узла с наименьшим весом
3. Сформируем новый узел с весом, равным сумме весов выбранных узлов, и присоединим к нему два выбранных узла в качестве детей
4. Добавим к списку только что сформированный узел вместо двух объединенных узлов
5. Добавим к списку только что сформированный узел вместо двух объединенных узлов
6. Если в списке больше одного узла, то повторим пункты со второго по пятый

Если в построенном дереве считать переход к левому сыну за 0, а к правому за 1, то путь, пройденный от вершины к символу-листу и есть новый код этого символа.

LZ77 и Huffman прекрасно дополняют друг друга. Можно заметить, что обработанный LZ77 текст всего лишь расширяет исходный алфавит, добавляя в него длину смещения и длину подстроки, поэтому мы можем дополнительно прогнать новый текст через алгоритм Хаффмана, увеличивая таким образом коэффициент сжатия.

На такой комбинации работает известный алгоритм **DEFLATE**, который используется архиватором gzip и png-изображениями.

3 Реализация программы

Несмотря на кажущуюся простоту описанных алгоритмов, при их реализации возникают некоторые проблемы. Самой сложной и неочевидной частью была организация поиска подстроки в LZ77. К сожалению, алгоритм не описывает этот процесс. Я осуществил его следующим образом.

Чтобы общее время архивации было линейным, поиск подстроки для каждого символа текста должен осуществляться за константное время. Для этого ограничим размеры словаря до p и собственно самой подстроки до q и организуем их в виде кольцевого буфера, так называемого "скользящего окна". Тогда они будут хранить информацию о последних $p + q$ символах, и даже наивный алгоритм будет работать за $O(p * q) = O(1)$.

Однако, как правило, значение p достаточно велико, поэтому поиск не должен зависеть от этого параметра, чтобы общее время работы нас утраивало. Для этой задачи подходит суффиксное дерево, позволяющее находить образец за его длину.

Класс TSuffTree — иплиментация суффиксного дерева, способного поддерживать свой размер. Так при добавлении нового символа в заполненное дерево, мы удаляем в нем самый старый суффикс (метод RemoveLongestSuffix), уменьшая тем самым обрабатываемую строку на единицу. Такой суффикс мы находим с помощью указателя LongestSuffix, а затем просто удаляем лист, соответствующий ему, а при необходимости и его родителя, если у него остается единственный сын. Заметим, что суффиксы добавляются в дерево в порядке длины — от самого длинного к самому короткому. Это позволяет легко находить новую позицию LongestSuffix (каждый суффикс хранит указатель на следующий добавленный).

Может случится так, что какая-то вершина в дереве будет оставаться очень долго, и её ссылка в словарь станет недействительной, так как он все время меняется. Чтобы решить эту проблему, суффиксное дерево размера p должно хранить строку для последних $2p$ символов. Тогда каждые p новых символов, мы обновляем все метки в дереве обходом в глубину (метод UpdateLabels), и они будут действительны ещё как минимум p добавлений, т.е. находиться в пределах размера строки $2p$.

Класс THuffmanTree — реализация дерева Хаффмана. К счастью, его программирование никаких проблем не вызвало, всё делается в полном соответствии с известным алгоритмом. Однако вместо списка кодируемых символов используется очередь с приоритетами std::priority_queue. Предоставляется следующий интерфейс:

1. const std::string& GetCode(alpChar key) — возвращает новое двоичное представления ключа в виде строки;
2. alpChar GetChar(IBitStream &ibfs) — возвращает символ по его двоичному представлению из битового потока
3. void SaveTo(OBitStream &obfs) — сохраняет дерево в двоичный поток
4. LoadFrom(IBitStream &ibfs) — загружает дерево из двоичного потока

Само дерево строится по статистике, которую мы собираем ещё при проходе lz77.

В процессе архивации происходит частая работа с последовательностью битов — запись/чтение. Для реальной компрессии данных эти последовательности должны плотно прилегать друг к другу, борьба идет за каждый бит! Поэтому для удобства работы были реализованы отдельные классы — битовые потоки IBitStream и OBitStream. Внутри себя они имеют буфер размера 1 байт. Когда IBitStream получает на вход строку из нулей и единиц, он в соответствии с ними побитово дополняет буфер, а в случае необходимости сбрасывает его либо в стандартный, либо в файловый поток методом write. Аналогичным образом, но в обратную сторону работает

OBitStream.

Совместно работать Huffman и lz77 начинают в функции encode файла compressor.cpp. Сначала создается экземпляр класса TLZ77 с определенными размерами буферов, который затем осуществляет первичное сжатие во временный файл temp.bin и возвращает статистику частоты появления символов расширенного алфавита(в него добавляется длина сдвига). По этой статистике строится и записывается во итоговый файл дерево хаффмана ht. Затем работа сводится к чтению символов нового алфавита из temp.bin и запись соответствующих им новых кодов в новый файл. Когда процесс сжатия заканчивается, в конец дописывается информация о начальном и конечном весе, а также значение контрольной суммы, которое вычисляется простым xor алгоритмом (checksum ^= newSymbol).

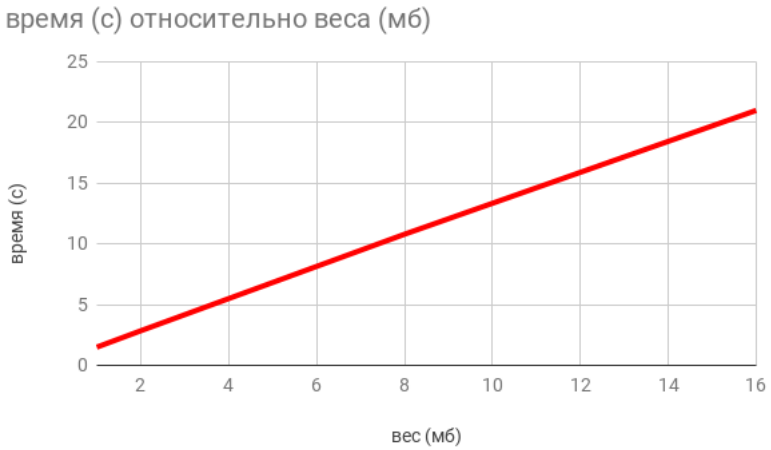
Работа с файловой системой осуществляется с помощью библиотеки experimental/filesystem 17-го стандарта c++.

4 Тестирование

Сравним производительность самописного архиватора с gzip на случайно сгенерированных файлах различных размеров:

1. **Архивация**

исходный файл (мб)		1	2	4	8	16
время (с)	gzip	0.075	0.159	0.31	0.52	0.97
	myzip	1.5	2.85	5.5	10.8	21
сжатый файл (мб)	gzip	0.6661	1.3	2.7	5.3	10.7
	myzip	0.6326	1.3	2.5	5.1	10.1



Как видно из таблицы, gzip работает быстрее приблизительно в 20 раз, однако сжимает чуть-чуть хуже. Выше представлен график зависимости времени сжатия myzip от веса входного файла.

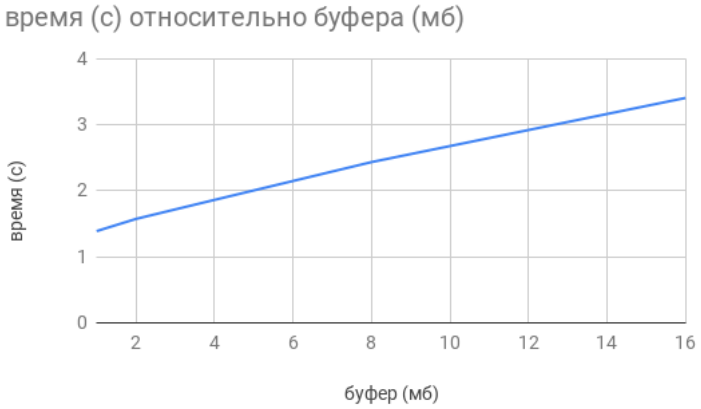
2. **Разархивация**

исходный файл (мб)		1	2	4	8	16
время (с)	gzip	0.026	0.050	0.097	0.171	0.324
	myzip	0.225	0.39	0.757	1.4	2.81

Тут ситуация получше — gzip работает быстрее всего в 8 раз!

Посмотрим, как размер словаря влияет на производительность программы. В качестве тестового файла воспользуемся tar архивом на 2мб.

размер буфера (мб)	2048	4096	8192	16384	32762
время (с)	1.388	1.573	1.862	2.436	3.41
сжатый файл (мб)	0.44	0.42	0.413	0.4	0.388



Несмотря на то, что поиск осуществляется за длину образца, мы, тем не менее, на каждой итерации совершаем какую-то работу для поддержания размера дерева. Этим и объясняется такой рост времени сжатия.

5 Дневник отладки

№	Статус	Комментарий
1	Compilation error	Неверно составленный makefile
2	Compilation error	Некорректное имя makefile

3	Runtime error	Компаратор для std::priority_queue выполнял сравнение >= вместо >, что иногда приводило к ошибкам. Баг удалось обнаружить с помощью ключа компиляции -D_GLIBCXX_DEBUG.
4	Time limit exceeded	Поиск подстроки в lz77 осуществлялся наивным алгоритмом. Несмотря на его линейность в нашем случае, он производил слишком много вычислений.
5	Time limit exceeded	Наивный алгоритм был заменен на Кнута-Морриса-Пратта, однако поиск в нем пропорционален длине словаря, что не сильно уменьшает константу в асимптотике.
6	Accepted	КМП был заменен на суффиксное дерево, поиск в котором зависит от длины образца, которая в разы меньше длины словаря

6 Недочёты

Несмотря на корректную работу программы, лично у меня есть к ней несколько претензий. К примеру, во время программирования курсового проекта я сначала разработал классы TLZ77 и THuffmanTree и только потом начал думать, как их соединить вместе. Аналогичная ситуация с битовыми потоками, а также переключением потока вывода на экран/файл. Все держится на костылях и компромиссах. Кодстайл также оставляет желать лучшего. Единственный выход — разработать архитектуру и декомпозицию программы с самого начала!

7 Выводы

Программирование курсового проекта стало целым испытанием для меня. Только я начинал думать, что все готово, как тут же приходилось все переделывать. И дело тут даже не в багах!

На лабораторных работах мы решаем отдельные небольшие задачи. В курсовом проекте же пришлось эти частички собирать во что-то единое и законченное. Сделать это можно не одним путем, но найти оптимальный из них — непростая задача. Как оказалось, сначала нужно продумать структуру и декомпозицию программы, и только потом приступать к реализации! В противном случае большая часть кода будет состоять из различного рода костылей.

Алгоритмы lz77 и huffman coding на первый взгляд кажутся довольно простыми. Дерево Хаффмана действительно легко запрограммировать, и оно быстро работает,

потому находит широкое применение на практике(DEFLATE, JPEG и др). Наоборот, детали реализации и скорость работы lz77 зависят от программиста.

Чтобы закончить курсовой проект пришлось в буквальном смысле повторить весь курс дискретного анализа. Это и алгоритмы сжатия, и алгоритмы поиска подстроки(суффиксное дерево, КМП) и алгоритмы поиска на графах(поиск в глубину в суффиксном дереве), и жадные алгоритмы(дерево хаффмана), а также общие навыки программирования. А еще я на личном опыте убедился, зачем нужны системы контроля версий.

8 Исходный код

compressor.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdint>
#include <map>
#include <experimental/filesystem>
#include <iomanip>

#include "tlz77.h"
#include "tbitstream.h"
#include "thuffmantree.h"

namespace fs = std::experimental::filesystem;

const char FAIL[] = "\0";
enum {COMPRESS, STDOUT = 'c', DECOMPRESS = 'd', LIST = 'l', TEST = 't', KEEP = 'k',
    RECURSION = 'r'};
bool recursion_flag = false;
bool keep_flag = false;
bool stdout_flag = false;
char action_flag = COMPRESS;
uint8_t speed_level = 2;

std::string uncompName(const std::string &name)
{
    if (name == "-") {
        return name;
    } else if (name.find(".myzip", name.size() - 6) != std::string::npos) {
        return name.substr(0, name.size() - 6);
    } else {
        return std::string(FAIL);
    }
}

std::pair<uint8_t, uint8_t> checksum(const std::string &name)
{
    std::ifstream ifs(name, std::ios::binary);
    uint8_t ret, temp, tempret = 0;

    while(ifs.peek() != EOF) {
        ret = tempret;
        ifs.read((char *) &temp, sizeof(char));
        tempret ^= temp;
    }
    return std::pair<uint8_t, uint8_t>(ret, tempret);
}
```

```
46
47  void encode(const std::string &from, const std::string &to)
48  {
49      TLZ77 lz77(255, 16384 / speed_level);
50      std::map<std::pair<bool, char>, uint32_t> statistics = lz77.Encode(from.data(), "
            temp.bin");
51
52      THuffmanTree ht(statistics);
53      IBitStream ibfs("temp.bin");
54      OBitStream obfs(to.data());
55      char c;
56
57      ht.SaveTo(obfs);
58
59      while(!ibfs.Eof()) {
60          ibfs >> c;
61          if (c == '0') {
62              std::pair<dis_t, len_t> matching;
63
64              ibfs.Read((char *) &matching.first, sizeof(dis_t));
65              ibfs.Read((char *) &matching.second, sizeof(len_t));
66
67              obfs << ht.GetCode(std::pair<bool, char>(false, (char) matching.second));
68              obfs.Write((char *) &matching.first, sizeof(dis_t));
69          } else {
70              ibfs.Read((char *) &c, sizeof(char));
71              obfs << ht.GetCode(std::pair<bool, char>(true, c));
72          }
73      }
74      obfs << ht.GetCode(TERM_PAIR);
75
76      obfs.Close();
77      ibfs.Close();
78
79      fs::remove(fs::path("temp.bin"));
80
81      uintmax_t unzip_size = (from != "-" ? fs::file_size(fs::path(from)) : 0);
82      uintmax_t zip_size = (to != "-" ? fs::file_size(fs::path(to)) + 2 * sizeof(
            uintmax_t) + sizeof(uint8_t) : 0);
83
84      if (to != "-") {
85          std::ofstream ofs(to, std::ios::binary | std::ios_base::app);
86          ofs.write((char *) &unzip_size, sizeof(uintmax_t));
87          ofs.write((char *) &zip_size, sizeof(uintmax_t));
88          ofs.close();
89      } else {
90          std::cout.write((char *) &unzip_size, sizeof(uintmax_t));
91          std::cout.write((char *) &zip_size, sizeof(uintmax_t));
92      }
```

```
93
94      uint8_t chcksm = (to != "-" ? checksum(to) : std::pair<uint8_t, uint8_t>(0, 0)).
            second;
95      if (to != "-") {
96          std::ofstream ofs(to, std::ios::binary | std::ios_base::app);
97          ofs.write((char *) &chcksm, sizeof(uint8_t));
98          ofs.close();
99      } else {
100         std::cout.write((char *) &chcksm, sizeof(uint8_t));
101     }
102 }
103
104 void decode(const std::string &from, const std::string &to)
105 {
106     THuffmanTree ht;
107     IBitStream ibfs(from.data());
108     OBitStream obfs("temp.bin");
109     std::pair<bool, char> temp;
110     uintmax_t unzip_size;
111
112     ht.LoadFrom(ibfs);
113
114     while ((temp = ht.GetChar(ibfs)) != TERM_PAIR) {
115         if (temp.first == true) {
116             obfs << '1';
117             obfs.Write((char *) &temp.second, sizeof(char));
118         } else {
119             dis_t distance;
120             ibfs.Read((char *) &distance, sizeof(dis_t));
121             obfs << '0';
122             obfs.Write((char *) &distance, sizeof(dis_t));
123             obfs.Write((char *) &temp.second, sizeof(len_t));
124         }
125     }
126
127     ibfs.Close();
128     obfs.Close();
129
130     TLZ77 lz77;
131     lz77.Decode("temp.bin", to.data());
132
133     fs::remove(fs::path("temp.bin"));
134 }
135
136 uint8_t parseFlags(char const *flags)
137 {
138     if (flags[0] != '-') {
139         return 1;
140     }
```

```
141     if (flags[1] == '\0') {
142         return 1;
143     }
144 
145     for(uint8_t i = 1; flags[i] != '\0'; ++i) {
146         char c = flags[i];
147         if (c == RECURSION) {
148             recursion_flag = true;
149         } else if (c == KEEP) {
150             keep_flag = true;
151         } else if (c == STDOUT) {
152             stdout_flag = true;
153             keep_flag = true;
154         } else if (c == '1') {
155             speed_level = 4;
156         } else if (c == '9') {
157             speed_level = 1;
158         } else if (c == DECOMPRESS || c == LIST || c == TEST) {
159             action_flag = c;
160         } else {
161             std::cout << "./myzip: invalid option -- '" << c << "'\n";
162             return 0;
163         }
164     }
165 
166     return 2;
167 }
168 
169 bool process(const std::string &filename)
170 {
171     bool ret = true;
172 
173     if (action_flag == COMPRESS) {
174         std::string to((filename == "-" || stdout_flag) ? "-" : (std::string(filename)
                + ".myzip"));
175         encode(filename, to);
176 
177         if (keep_flag == false) {
178             fs::remove(fs::path(filename));
179         }
180     } else if (action_flag == DECOMPRESS) {
181         std::string to(uncompName(filename));
182         if (to != FAIL) {
183             decode(filename, (stdout_flag ? "-" : to));
184 
185             if (keep_flag == false) {
186                 fs::remove(fs::path(filename));
187             }
188         } else {
```

```
189             std::cout << "./myzip: " << filename << ": unknown suffix -- ignore\n";
190         }
191     } else if (action_flag == LIST) {
192         std::string to(uncompName(filename));
193 
194         if (to != "-" && to != FAIL) {
195             std::ifstream ifs(filename, std::ios::binary);
196             uintmax_t unzip_size, zip_size;
197 
198             ifs.seekg(0, std::ios_base::end);
199             ifs.seekg(uintmax_t(ifs.tellg()) - 2 * sizeof(uintmax_t) - sizeof(uint8_t),
                      std::ios_base::beg);
200 
201             ifs.read((char *) &unzip_size, sizeof(uintmax_t));
202             ifs.read((char *) &zip_size, sizeof(uintmax_t));
203             ifs.close();
204 
205             std::cout.precision(3);
206             std::cout << std::right << std::setw(19) << zip_size << std::setw(20) <<
                    unzip_size
207                       << std::setw(6) << (100.0 * zip_size / unzip_size) << std::setw
                          (2) << "% "
208                       << std::left << std::setw(17) << to << std::endl;
209 
210         }
211     } else if (action_flag == TEST) {
212         uint8_t cur_chcksum = checksum(filename).first;
213         uint8_t chcksm;
214 
215         std::ifstream ifs(filename, std::ios::binary);
216 
217         ifs.seekg(0, std::ios_base::end);
218         ifs.seekg(uintmax_t(ifs.tellg()) - sizeof(uint8_t), std::ios_base::beg);
219 
220         ifs.read((char *) &chcksm, sizeof(uint8_t));
221         ifs.close();
222 
223         if (cur_chcksum != chcksm) {
224             ret = false;
225         }
226     }
227 
228     return ret;
229 }
230 
231 int main(int argc, char const *argv[])
232 {
233     char c;
234     if (argc == 1) {
```

```
235         std::cout << "For help, type: ./myzip -h\n";
236     } else {
237         uint8_t start = parseFlags(argv[1]);
238         if (!start) {
239             std::cout << "Try `./myzip -h' for more information.\n";
240         } else {
241             if (action_flag == LIST) {
242                 std::cout << std::right << std::setw(19) << "compressed" << std::setw
                        (20) << "uncompressed"
243                           << std::setw(7) << "ratio" << ' ' << std::left << std::setw(17)
                              << "uncompressed_name" << std::endl;
244             }
245             for (uint8_t i = start; i < argc; ++i) {
246                 bool stdin_flag = std::string("-") == argv[i];
247                 if (fs::exists(argv[i]) || stdin_flag) {
248                     if (recursion_flag && fs::is_directory(fs::path(argv[i]))) {
249                         fs::recursive_directory_iterator begin(argv[i]);
250                         fs::recursive_directory_iterator end;
251 
252                         for (auto path = begin; path != end; ++path) {
253                             if (fs::is_regular_file(*path)) {
254                                 process(path->path().string());
255                             }
256                         }
257                     } else {
258                         if (fs::is_regular_file(argv[i])) {
259                             if (process(std::string(argv[i])) == false) {
260                                 return 1;
261                             }
262                         } else {
263                             std::cout << "./myzip: " << argv[i] << " is a directory --
                                    ignored\n";
264                         }
265                     }
266                 } else {
267                     std::cout << "./myzip: " << argv[i] << ": No such file or directory\
                            n";
268                 }
269             }
270         }
271     }
272 
273     return 0;
274 }
```

tbitstream.h

```
1 #ifndef TBITSTREAM_H
2 #define TBITSTREAM_H
3 
```

```
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7  #include <cstdint>
8
9  class OBitStream
10 {
11 private:
12     bool stdout;
13     std::ostream *ofs;
14     char buff;
15     uint8_t size;
16     const uint8_t capacity = 8;
17
18 public:
19     OBitStream();
20     OBitStream(const char* filename);
21     OBitStream(std::string &&filename);
22
23     OBitStream& operator<<(char c);
24     OBitStream& operator<<(const std::string &str);
25     void Write(const char *str, uint16_t size);
26     bool Eof();
27     void Flush();
28     void Close();
29
30     ~OBitStream();
31
32 };
33
34 class IBitStream
35 {
36 private:
37     bool stdin;
38     std::istream *ifs;
39     char buff;
40     uint8_t size;
41     const uint16_t capacity = 8;
42
43 public:
44     IBitStream();
45     IBitStream(const char* filename);
46
47     IBitStream& operator>>(char &c);
48     void Read(char *str, uint16_t size);
49     bool Eof();
50     void Close();
51
52     ~IBitStream();
```

```
53
54 };
55
56 #endif
```

tbitstream.cpp

```
1  #include "tbitstream.h"
2
3  OBitStream::OBitStream(): stdout(true), buff(0), size(0)
4  {
5      ofs = new std::ostream(nullptr);
6      ofs->rdbuf(std::cout.rdbuf());
7  }
8
9  OBitStream::OBitStream(const char* filename): stdout(false), buff(0), size(0)
10 {
11     if (std::string("-") == filename) {
12         ofs = new std::ostream(nullptr);
13         ofs->rdbuf(std::cout.rdbuf());
14     } else {
15         ofs = new std::ofstream(filename, std::ios::binary);
16     }
17 }
18
19 OBitStream::OBitStream(std::string &&filename): OBitStream(filename.data()) {}
20
21
22 OBitStream& OBitStream::operator<<(char c)
23 {
24     if (size == capacity) {
25         Flush();
26     }
27     buff = (buff << 1) | (c - '0');
28     ++size;
29
30     return *this;
31 }
32
33 OBitStream& OBitStream::operator<<(const std::string &str)
34 {
35     for (uint16_t i = 0; i < str.size(); ++i) {
36         if (size == capacity) {
37             Flush();
38         }
39         buff = (buff << 1) | (str[i] - '0');
40         ++size;
41     }
42
43     return *this;
```

```
44 }
45
46 void OBitStream::Write(const char *str, uint16_t length)
47 {
48     for (uint16_t i = 0; i < length; ++i) {
49         char c = str[i];
50         uint8_t temp_size = size;
51         buff = (buff << (capacity - size)) | (((unsigned char) c) >> size);
52         size = capacity;
53         Flush();
54         buff = c & (((unsigned char) 255) >> (capacity - temp_size));
55         size = temp_size;
56     }
57 }
58
59 bool OBitStream::Eof()
60 {
61     return ofs->eof();
62 }
63
64 void OBitStream::Flush()
65 {
66     if (size != 0) {
67          buff = buff << (capacity - size);
68         ofs->write((char *) &buff, sizeof(char));
69         buff = 0;
70         size = 0;
71     }
72 }
73
74 void OBitStream::Close()
75 {
76     if (ofs != nullptr) {
77         Flush();
78         buff = 0;
79         size = 0;
80         delete ofs;
81         ofs = nullptr;
82     }
83 }
84
85 OBitStream::~OBitStream()
86 {
87     Close();
88 }
89
90 //-----------------------------------------------------------
91
92 IBitStream::IBitStream(): buff(0), size(0)
```

```
93  {
94      ifs = new std::istream(nullptr);
95      ifs->rdbuf(std::cin.rdbuf());
96  }
97
98  IBitStream::IBitStream(const char* filename): buff(0), size(0)
99  {
100     if (std::string("-") == filename) {
101         ifs = new std::istream(nullptr);
102         ifs->rdbuf(std::cin.rdbuf());
103     } else {
104         ifs = new std::ifstream(filename, std::ios::binary);
105     }
106 }
107
108 IBitStream& IBitStream::operator>>(char &c)
109 {
110     if (size == 0) {
111         ifs->read((char *) &buff, sizeof(char));
112         size = capacity;
113     }
114     c = (buff & 128) == 128 ? '1' : '0';
115     buff <<= 1;
116     --size;
117
118     return *this;
119 }
120
121 void IBitStream::Read(char *str, uint16_t length)
122 {
123     for (uint16_t i = 0; i < length; ++i) {
124         str[i] = buff;
125         ifs->read((char *) &buff, sizeof(char));
126         str[i] = str[i] | (((unsigned char) buff) >> size);
127         buff = buff << (capacity - size);
128     }
129 }
130
131 bool IBitStream::Eof()
132 {
133     return ifs->eof();
134 }
135
136 void IBitStream::Close()
137 {
138     if (ifs != nullptr) {
139         buff = 0;
140         size = 0;
141         delete ifs;
```

```
142         ifs = nullptr;
143     }
144 }
145
146 IBitStream::~IBitStream()
147 {
148     Close();
149 }
```

thuffmantree.h

```
1  #ifndef THUFFMANTREE_H
2  #define THUFFMANTREE_H
3
4  #include <iostream>
5  #include <fstream>
6  #include <queue>
7  #include <vector>
8  #include <map>
9  #include <string>
10
11 #include "tbitstream.h"
12
13 typedef std::pair<bool, char> alpChar;
14
15 struct TNode
16 {
17     alpChar item;
18     uint32_t freq;
19     TNode *left;
20     TNode *right;
21
22     TNode(alpChar _item, uint32_t _freq, TNode *_left = nullptr, TNode *_right =
           nullptr):
23         item(_item), freq(_freq), left(_left), right(_right) {}
24
25     ~TNode()
26     {
27         if (left != nullptr) {
28             delete left;
29         }
30         if (right != nullptr) {
31             delete right;
32         }
33     }
34 };
35
36 class THuffmanTree
37 {
38 private:
```

```
39      TNode *root = nullptr;
40      std::map<alpChar, std::string> table;
41
42      void BuildTable(TNode *node, std::string &path);
43      void SaveTreeTo(OBitStream &obfs, TNode *node);
44      TNode * LoadTreeFrom(IBitStream &ibfs);
45
46  public:
47      THuffmanTree(std::map<alpChar, uint32_t> &statistics);
48      THuffmanTree();
49
50      const std::string& GetCode(alpChar key);
51      alpChar GetChar(IBitStream &ibfs);
52      void BuildFromStat(std::map<alpChar, uint32_t> &statistics);
53      void SaveTo(OBitStream &obfs);
54      void LoadFrom(IBitStream &ibfs);
55
56      ~THuffmanTree();
57  };
58
59  #endif
```

thuffmantree.cpp

```
1   #include "thuffmantree.h"
2
3   struct comparator
4   {
5       bool operator() (const TNode *p1, const TNode *p2)
6       {
7           return p1->freq > p2->freq;
8       }
9   };
10
11  THuffmanTree::THuffmanTree(std::map<alpChar, uint32_t> &statistics)
12  {
13      BuildFromStat(statistics);
14  }
15
16  THuffmanTree::THuffmanTree()
17  {
18
19  }
20
21  const std::string& THuffmanTree::GetCode(alpChar key)
22  {
23      return table[key];
24  }
25
26  alpChar THuffmanTree::GetChar(IBitStream &ibfs)
```

```
27 {
28     TNode *node = root;
29     char c;
30 
31     while(node->left != nullptr || node->right != nullptr) {
32         ibfs >> c;
33         node = (c == '1' ? node->right : node->left);
34     }
35 
36     return node->item;
37 }
38 
39 void THuffmanTree::BuildTable(TNode *node, std::string &path)
40 {
41     if (node != nullptr) {
42         if (node->right == nullptr && node->left == nullptr) {
43             table[node->item] = path;
44         }
45         path.push_back('1');
46         BuildTable(node->right, path);
47         path.pop_back();
48 
49         path.push_back('0');
50         BuildTable(node->left, path);
51         path.pop_back();
52     }
53 }
54 
55 void THuffmanTree::BuildFromStat(std::map<alpChar, uint32_t> &statistics)
56 {
57     if (root != nullptr) {
58         delete root;
59         table.clear();
60     } else {
61         uint16_t alpSize = statistics.size();
62         std::priority_queue<TNode*, std::vector<TNode*>, comparator> pq;
63 
64         for(auto i: statistics) {
65             pq.push(new TNode(i.first, i.second));
66         }
67 
68         for(uint16_t i = 1; i < alpSize; ++i) {
69             TNode *l = pq.top();
70             pq.pop();
71             TNode *r = pq.top();
72             pq.pop();
73             pq.push(new TNode(alpChar(), l->freq + r->freq, l, r));
74         }
75 
```

```
76          root = pq.top();
77          pq.pop();
78
79          std::string path;
80          BuildTable(root, path);
81      }
82  }
83
84  void THuffmanTree::SaveTreeTo(OBitStream &obfs, TNode *node)
85  {
86      if (node->right == nullptr && node->left == nullptr) {
87          obfs << '1';
88          obfs << (node->item.first == true ? '1' : '0');
89          obfs.Write((char *) &(node->item.second), sizeof(char));
90      } else {
91          obfs << '0';
92          SaveTreeTo(obfs, node->left);
93          SaveTreeTo(obfs, node->right);
94      }
95  }
96
97  void THuffmanTree::SaveTo(OBitStream &obfs)
98  {
99      SaveTreeTo(obfs, root);
100 }
101
102 TNode * THuffmanTree::LoadTreeFrom(IBitStream &ibfs)
103 {
104     char c;
105     TNode *node;
106
107     ibfs >> c;
108     if (c == '1') {
109         alpChar item;
110         ibfs >> c;
111         item.first = (c == '1' ? true : false);
112         ibfs.Read((char *) &(item.second), sizeof(char));
113         node = new TNode(item, 0, nullptr, nullptr);
114     } else {
115         node = new TNode(alpChar(), 0, nullptr, nullptr);
116         node->left = LoadTreeFrom(ibfs);
117         node->right = LoadTreeFrom(ibfs);
118     }
119
120     return node;
121 }
122
123 void THuffmanTree::LoadFrom(IBitStream &ibfs)
124 {
```

```
125     root = LoadTreeFrom(ibfs);
126 }
127
128 THuffmanTree::~THuffmanTree()
129 {
130     delete root;
131 }
```

tlz77.h

```
1  #ifndef LZ77_H
2  #define LZ77_H
3
4  #include <fstream>
5  #include <iostream>
6  #include <map>
7  #include <string>
8  #include <vector>
9  #include "tbitstream.h"
10 #include "tringarray.h"
11 #include "tsufftree.h"
12
13 typedef uint8_t len_t;
14 typedef uint16_t dis_t;
15
16 const len_t MAX_INPUT_CAPACITY = 255;
17 const dis_t MAX_SEARCH_CAPACITY = 32768;
18 const char TERM = '\0';
19 const std::pair<bool, char> TERM_PAIR = {false, 0};
20
21 class TLZ77
22 {
23 private:
24     len_t input_cap;
25     dis_t search_cap;
26
27 public:
28     TLZ77(len_t icap = MAX_INPUT_CAPACITY, dis_t scap = MAX_SEARCH_CAPACITY);
29
30     std::map<std::pair<bool, char>, uint32_t> Encode(char const *from, char const *to);
31     void Decode(char const *from, char const *to);
32
33     ~TLZ77();
34 };
35
36 #endif
```

tlz77.cpp

```
1  #include "tlz77.h"
2
```

```
3  TLZ77::TLZ77(len_t icap, dis_t scap): input_cap(icap), search_cap(scap) {}
4
5  std::istream * initistream(char const *name)
6  {
7      std::istream *ifs;
8
9      if (name[0] == '-' && name[1] == '\0') {
10         ifs = new std::istream(nullptr);
11         ifs->rdbuf(std::cin.rdbuf());
12     } else {
13         ifs = new std::ifstream(name, std::ios::binary);
14     }
15
16     return ifs;
17 }
18
19 std::ostream * initostream(char const *name)
20 {
21     std::ostream *ofs;
22
23     if (name[0] == '-' && name[1] == '\0') {
24         ofs = new std::ostream(nullptr);
25         ofs->rdbuf(std::cout.rdbuf());
26     } else {
27         ofs = new std::ofstream(name, std::ios::binary);
28     }
29
30     return ofs;
31 }
32
33 std::map<std::pair<bool, char>, uint32_t> TLZ77::Encode(char const *from, char const *
       to)
34 {
35     std::istream *ifs = initistream(from);
36     OBitStream ofs(to);
37     std::map<std::pair<bool, char>, uint32_t> statistics;
38     TRingArray<char> input_buff(input_cap);
39     TSuffTree search_buff(search_cap);
40     char c;
41
42     while(input_buff.size != input_cap && ifs->peek() != EOF) {
43         ifs->read((char *) &c, sizeof(char));
44         input_buff.Push(c);
45     }
46
47     while (input_buff.size != 0) {
48         std::pair<dis_t, len_t> matching = search_buff.Find(input_buff);
49
50         if (matching.second == 0) {
```

```
51              matching.second = 1;
52          }
53
54          if (matching.second > 3) {
55              ofs << "0";
56              ofs.Write((char *) &matching.first, sizeof(dis_t));
57              ofs.Write((char *) &matching.second, sizeof(len_t));
58              ++statistics[std::pair<bool, char>(false, (char) matching.second)];
59          } else {
60              for (len_t i = 0; i < matching.second; ++i) {
61                  ofs << "1";
62                  c = input_buff[input_buff.pos + i];
63                  ofs.Write((char *) &c, sizeof(char));
64                  ++statistics[std::pair<bool, char>(true, c)];
65              }
66          }
67
68          for (len_t i = 0; i < matching.second; ++i) {
69              c = input_buff[input_buff.pos];
70              search_buff.Extend(c);
71              input_buff.PopFront();
72          }
73
74          while(input_buff.size != input_cap && ifs->peek() != EOF) {
75              ifs->read((char *) &c, sizeof(char));
76              input_buff.Push(c);
77          }
78
79      }
80
81      statistics[TERM_PAIR] = 1;
82
83      //input_buff.Reset();
84      //search_buff.Reset();
85
86      delete ifs;
87      ofs.Close();
88
89      return statistics;
90  }
91
92  void TLZ77::Decode(char const *from, char const *to)
93  {
94      IBitStream ifs(from);
95      std::ostream *ofs = initostream(to);
96      TRingArray<char> search_buff(16384);
97      char c;
98
99      while (!ifs.Eof()) {
```

```
100         ifs >> c;
101         if (c == '0') {
102             std::pair<dis_t, len_t> matching;
103 
104             ifs.Read((char *) &matching.first, sizeof(dis_t));
105             ifs.Read((char *) &matching.second, sizeof(len_t));
106 
107             for (len_t i = 0; i < matching.second; ++i) {
108                 c = search_buff[search_buff.DistanceToEnd(0) - matching.first];
109                 ofs->write((char *) &c, sizeof(char));
110                 search_buff.Push(c);
111             }
112         } else {
113             ifs.Read((char *) &c, sizeof(char));
114             ofs->write((char *) &c, sizeof(char));
115             search_buff.Push(c);
116         }
117     }
118 
119     ifs.Close();
120     delete ofs;
121 }
122 
123 TLZ77::~TLZ77()
124 {
125 
126 }
```

tringarray.h

```
1  #ifndef TRINGARRAY_H
2  #define TRINGARRAY_H
3  
4  template<typename T> struct TRingArray
5  {
6      T *arr;
7      int size, cap, pos;
8  
9      TRingArray(int _cap): cap(_cap), size(0), pos(0)
10     {
11         arr = new T[cap];
12     }
13 
14     int RealIndex(int index)
15     {
16         int newIndex = index % cap;
17         return newIndex + (newIndex >= 0 ? 0 : cap);
18     }
19 
20     T& operator[](int index)
```

```
21      {
22          return arr[RealIndex(index)];
23      }
24
25      void Push(T el)
26      {
27          arr[(pos + size) % cap] = el;
28
29          if (size < cap) {
30              ++size;
31          } else {
32              pos = ++pos % cap;
33          }
34      }
35
36      void PopFront()
37      {
38          pos = ++pos % cap;
39          --size;
40      }
41
42      int EndPos()
43      {
44          return (pos + size - 1) % cap;
45      }
46
47      int DistanceToEnd(int from)
48      {
49          int end = (pos + size - 1) % cap;
50          return end - from + 1 + (end < from ? size : 0);
51      }
52
53      ~TRingArray()
54      {
55          delete[] arr;
56      }
57  };
58
59  #endif
```

tsufftree.h

```
1   #ifndef TSUFFTREE_H
2   #define TSUFFTREE_H
3
4   #include <iostream>
5   #include <map>
6   #include <vector>
7   #include <string>
8
```

```
9  #include "tringarray.h"
10
11 const int inf = 100000;
12
13 struct TSTNode
14 {
15     TSTNode* parent;
16     TSTNode* suffRef;
17     std::map<char, TSTNode*> edges;
18     int beg, len;
19
20     TSTNode() :
21         parent(this), suffRef(this), beg(0), len(0) {}
22
23     TSTNode(TSTNode *_parent, int _beg, int _len) :
24         parent(_parent), suffRef(nullptr), beg(_beg), len(_len) {}
25
26     ~TSTNode()
27     {
28         for (auto i: edges) {
29             delete i.second;
30         }
31     }
32 };
33
34 class TSuffTree
35 {
36 private:
37     TSTNode *root;
38     TSTNode *longestSuffix;
39     TSTNode *lastLeaf;
40
41     TRingArray<char> str;
42     const int capacity;
43     int size, updateCount;
44
45     //pointers to current position in the tree
46     TSTNode *pos; // <<<---
47     int depth; // <<<---
48
49     //functions to control position movements and to modify the tree
50     bool Go(char c);
51     void FastGo(int beg, int len); //we use it when we sure the path exists
52     void FindSuffRefPos();
53     void GoSuffRef();
54
55     //functions to modify the tree
56     void Split();
57     TSTNode* Fork(char c);
```

```
58      void RemoveLeaf(TSTNode *n);
59      void RemoveLongestSuffix();
60
61      //update edges' labels to maintain their validaty
62      void UpdateLabels();
63      int UpdateLabel(TSTNode *n);
64  public:
65      TSuffTree(int _capacity);
66
67      void Extend(char c);
68      std::pair<int, int> Find(TRingArray<char> &pattern); //(indexes (i, len))
69
70      ~TSuffTree();
71  };
72
73  #endif
```

tsufftree.cpp

```
1   #include "tsufftree.h"
2
3   TSuffTree::TSuffTree(int _capacity) :
4       root(new TSTNode()), longestSuffix(nullptr), lastLeaf(nullptr), pos(root), depth(1)
            ,
5       capacity(_capacity), size(0), str(2 * _capacity), updateCount(0)
6   {
7
8   }
9
10  bool TSuffTree::Go(char c)
11  {
12      if (depth < pos->len && depth < str.DistanceToEnd(pos->beg)) { //position on the
            edge
13          if (str[pos->beg + depth] == c) {
14              ++depth;
15              return true;
16          }
17      } else { //position on the node
18          if (pos->edges.count(c)) {
19              pos = pos->edges[c];
20              depth = 1;
21              return true;
22          }
23      }
24
25      return false;
26  }
27
28  void TSuffTree::FastGo(int beg, int len)
29  {
```

```
30      while (len > pos->len) {
31          len -= pos->len;
32          beg += pos->len;
33          pos = pos->edges[str[beg]];
34      }
35
36      depth = len;
37  }
38
39  void TSuffTree::Split()
40  {
41      if (depth < pos->len) { //position on the edge
42          TSTNode *mid = new TSTNode(pos->parent, pos->beg, depth);
43          pos->parent->edges[str[pos->beg]] = mid;
44          mid->edges[str[mid->beg + mid->len]] = pos;
45          pos->parent = mid;
46          pos->beg = mid->beg + mid->len;
47          pos->len = pos->len - mid->len;
48
49          pos = mid;
50      }
51  }
52
53  TSTNode* TSuffTree::Fork(char c)
54  {
55      Split();
56
57      return (pos->edges[c] = new TSTNode(pos, str.EndPos(), capacity));
58  }
59
60  void TSuffTree::FindSuffRefPos()
61  {
62      int shift = (pos->parent == root ? 1 : 0);
63      int fastGoBeg = pos->beg + shift;
64      int fastGoLen = depth - shift;
65
66      if (fastGoLen == 0) {
67          pos = root;
68      } else {
69          pos = pos->parent->suffRef->edges[str[fastGoBeg]];
70          FastGo(fastGoBeg, fastGoLen);
71      }
72  }
73
74  void TSuffTree::GoSuffRef()
75  {
76      if (pos->suffRef == nullptr) {
77          TSTNode *from = pos;
78
```

```
79          FindSuffRefPos();
80          Split();
81
82          from->suffRef = pos;
83      } else {
84          pos = pos->suffRef;
85      }
86  }
87
88  void TSuffTree::RemoveLeaf(TSTNode *n)
89  {
90      if (n == nullptr || n->edges.size() > 0) {
91          return;
92      }
93
94      char c = str[n->beg]; //c is the first letter leading to leaf
95      n = n->parent; //now n is internal node leading to leaf
96      delete n->edges[c];
97      n->edges.erase(c);
98
99      if (n->edges.size() == 1 && n != root) { //remove internal node with the only child
100         TSTNode *onlyLeaf = n->edges.begin()->second;
101
102         n->parent->edges[str[n->beg]] = onlyLeaf;
103         onlyLeaf->parent = n->parent;
104         onlyLeaf->beg -= n->len;
105         onlyLeaf->len += n->len;
106
107         if (pos == onlyLeaf) {
108             depth += n->len;
109         }
110
111         if (pos == n) {
112             pos = onlyLeaf;
113         }
114
115         n->edges.begin()->second = nullptr;
116         delete n;
117     }
118 }
119
120 void TSuffTree::RemoveLongestSuffix()
121 {
122     if (longestSuffix != nullptr) {
123         if (pos != longestSuffix) {
124             TSTNode *n = longestSuffix;
125             longestSuffix = longestSuffix->suffRef;
126             RemoveLeaf(n);
127         } else { //builder pos is on the longest suffix, special case
```

```
            //repoint longest suffix pos
            lastLeaf->suffRef = pos;
            lastLeaf = pos;
            longestSuffix = longestSuffix->suffRef;

            //relabel current node as a new one and go to the next one
            pos->beg = str.EndPos() + 1 - depth;
            pos->len = capacity;

            FindSuffRefPos(); //go to new pos
        }
    }
}

int TSuffTree::UpdateLabel(TSTNode *n)
{
    if (n->edges.empty()) {
        return str.DistanceToEnd(n->beg);
    }

    int minDist = capacity;
    int newEnd = n->beg;

    for (auto i: n->edges) {
        int dist = UpdateLabel(i.second);
        if (dist < minDist) {
            minDist = dist;
            newEnd = i.second->beg;
        }
    }

    n->beg = str.RealIndex(newEnd - n->len);
    return minDist + n->len;
}

void TSuffTree::UpdateLabels()
{
    for (auto i: root->edges) {
        UpdateLabel(i.second);
    }
}

void TSuffTree::Extend(char c)
{
    if (size == capacity) {
        RemoveLongestSuffix();
        --size;
    }
    str.Push(c);
```

```
177     ++size;
178     ++updateCount;
179 
180     if (size > 1) {
181         while (!Go(c)) { //while rule 2
182             TSTNode *newLeaf = Fork(c);
183 
184             //save pointers to the next longest suffix after the previous one
185             lastLeaf->suffRef = newLeaf;
186             lastLeaf = newLeaf;
187 
188             if (pos == root) {
189                 break;
190             }
191             GoSuffRef();
192         }
193     } else {
194         longestSuffix = Fork(c);
195         lastLeaf = longestSuffix;
196     }
197 
198     if (updateCount == capacity) {
199         UpdateLabels();
200         updateCount;
201         updateCount = 0;
202     }
203 }
204 
205 std::pair<int, int> TSuffTree::Find(TRingArray<char> &pattern)
206 {
207     // we are saving the builder position
208     // because we will use tree's position pointers for searching
209     TSTNode *savedPos = pos;
210     int savedDepth = depth;
211 
212     pos = root;
213     depth = 1;
214 
215     int len;
216     for (len = 0; len < pattern.size && Go(pattern[pattern.pos + len]); ++len) {}
217 
218     std::pair<int, int> ret(str.DistanceToEnd(pos->beg + depth - len), len);
219 
220     pos = savedPos;
221     depth = savedDepth;
222 
223     return ret;
224 }
225 
```

```
226 TSuffTree::~TSuffTree()
227 {
228     delete root;
229 }
```