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 + LZW)

Студент: Е. А. Кондратьев
Преподаватель: С. А. Сорокин
Группа: М8О-306Б-19

Дата:
Оценка:
Подпись:

Москва, 2021

Курсовая работа

Задание: Необходимо разработать и реализовать архиватор, который использует указанные методы сжатия данных для сжатия одного файла.

В качестве примера возьмем программу gzip.

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

1 Описание

Реализация курсового проекта сводится к решению следующих задач:

1. Изучите теоретическую часть приведенных алгоритмов сжатия данных.

2. Ознакомьтесь с существующими реализациями и определитесь с конкретными деталями реализации.

3. Обработка особых случаев и исключений.

2 Теоретическая часть

Алгоритм LZW – Lempel-Ziv-Welch – Алгоритм Лемпеля-Зива-Велча – это алгоритм сжатия данных без потерь на основе словаря. Важной особенностью и преимуществом этого алгоритма является то, что алгоритм разработан таким образом, что его достаточно легко реализовать как программно, так и аппаратно.

Это очень наглядно и удобно, так как не требует расчета частот символов. При кодировании не нужно сохранять словарь. Хотя для следующего алгоритма - полустатического кодирования Хаффмана, необходимо будет сохранить кодовое дерево.

В общих чертах процесс сжатия: символы входного потока считываются последовательно, и проверяется, существует ли такая строка чтения в словаре. Пока такая строка существует, считывается следующий символ, в противном случае код найденной строки отправляется на выход, а новая строка вводится в словарь.

Поскольку сжатие данных осуществляется без потерь, этот алгоритм можно использовать для сжатия текстовых и растровых данных. Существующие реализации используются в файлах TIFF, PDF, GIF, PostScript.

Интересным моментом в истории алгоритма LZW являются патенты на него. При разработке формата GIF CompuServe не знала о патенте. В декабре 1994 года, когда Unisys стало известно об использовании LZW в широко используемом графическом формате, компания обнародовала свои планы по сбору лицензионных сборов с коммерческого программного обеспечения, которое могло бы создавать файлы GIF. В то время формат уже был настолько распространен, что большинству софтверных компаний не оставалось ничего другого, кроме как платить. Эта ситуация стала одной из причин развития графического формата PNG, который стал третьим по распространенности после GIF и JPEG.o распространённости, после GIF и JPEG.

Аналогичным образом была создана утилита *gzip*, поскольку в *compress* был использован LZW. Как говорится на сайте GNU «The superior compression ratio of gzip is just a bonus»

На текущий момент сроки всех патентов истекли.

Алгоритм кодирования Хаффмана – весь алгоритм сводится к вычислению кодов

Хаффмана, и последующим сопоставлением входного кода и символа.

Коды Хаффмана – оптимальные префиксные коды. Ни один код не является префиксом другого, это позволяет однозначно интерпритировать входные данные, считывая их последовательно.

Вычисление кодов Хаффмана в полустатическом кодировании сводится к проходу по файлу для вычисления частот появления символов, построению по этим частотам дерева Хаффмана и сопоставлению входным данным полученных кодов.

Кодирование Хаффмана прекрасно логически дополняет кодирование LZW: LZW выделяет частые подстроки, алгоритм Хаффмана кодирует часто встречающиеся элемент более короткими кодами по сравнению с редко встречающимися.

3 Реализация

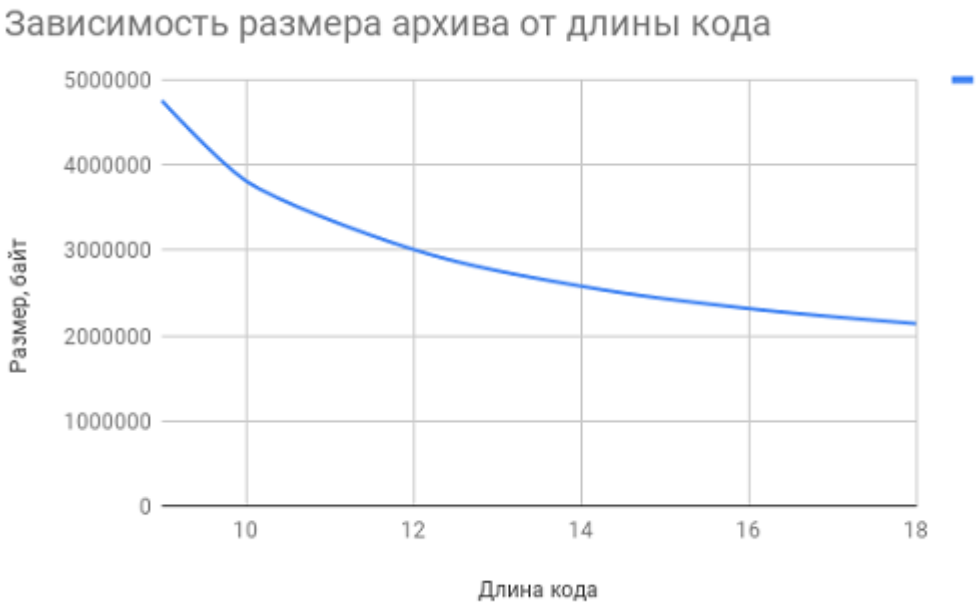
Сердцем или просто важной частью алгоритма LZW можно назвать словарь. От реализации словаря зависит время архивации. Структура словаря в декодере довольно очевидна – доступ по коду к конкретным строкам – массив или вектор. Совсем другая ситуация с кодером: необходимо по строке определять код и нахождение в словаре. Для решения этой проблемы было реализовано префиксное дерево. С каждым входным символом опускатся от корня, если пути нет – слова нет, небходимо его добавить.

Во время решения второй задачи, кроме посещения лекций по дискретному анализу, я изучил материалы по алгоритму Лемпеля — Зива — Велча, и заметил, что выбор того каким образом реализовывать конкретные особенности алгоритма ложится непосредственно на разработчика, конкретных спецификаций тут нет.

В LZW как в прочих словарных алгоритмах нужно заранее предусматривать каких размеров будет словарь. Взаимосвязь между памятью затрачиваемой на словарь и степенью сжатия довольно очевидна: больше памяти тратится на словарь – лучше сжатие. Поскольку в архиваторе нужно предусмотреть ключи, отвечающие за соотношение скорости к качеству сжатия, этими ключами будет управляться размер памяти на заданный словарь. Моё внимание привлекла модификация алгоритма LZC, в которой длины кодов растут от 9 до 16 битов. Эта модификация была реализована в некогда популярной утилите compress. После того как длина кода кода достигает размера в 17 бит, словарь сбрасывется и в нём остаются только символы с 0x00 по 0xff, то есть такие же какие были в словаре в самом начале кодирования. Декодер должен учитывать такую ситуацию и также сбрасывать словарь. В моём алгоритме декодер также как и кодер подсчитывает длину кода и исходя из неё принимает решение о сбросе словаря.

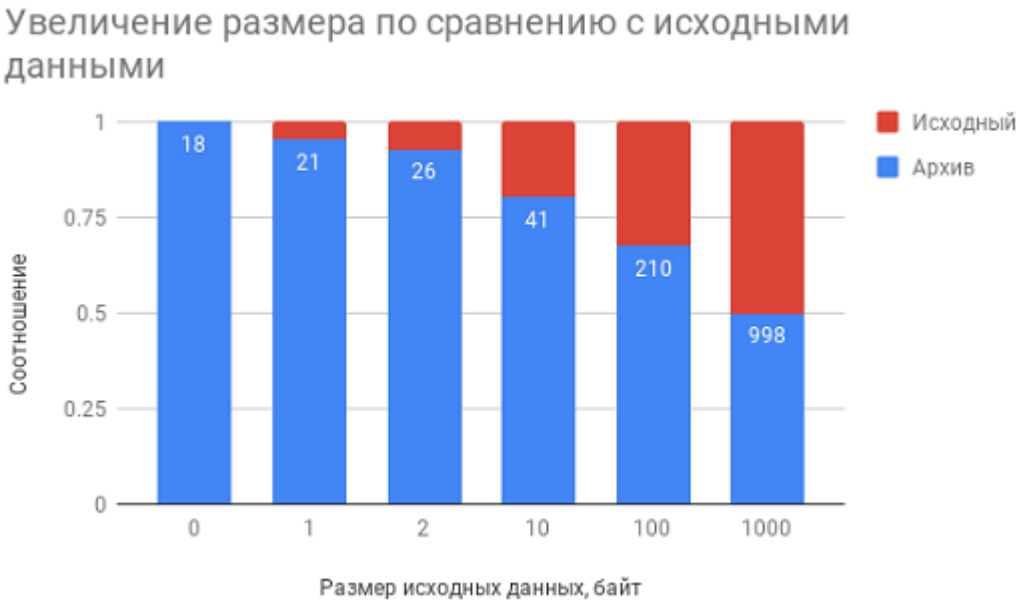
Пример для файла pdf весом 12,2 МБ

Для нескольких книг собранных в архив .tar 12,3 МБ



Однако маленький словарь не обязательно значит быстрое кодирование, например для уже сжатых файлов: изображения .img, видео и аудио, большинство кодов будут записываться в словарь и не использоваться более, просто занимая место, вследствие этого словарь будет сбрасываться слишком часто и станут заметны затраты по вре-

мени на это сбрасывание и уменьшение данных в размере будет почти незаметно, более того, поскольку алгоритму нужно записывать дерево Хаффмана (компактную сериализацию) возможно, что размер архива будет больше размера исходного файла. То же самое происходит и с очень малыми файлами.



Но архиваторы испольуют всё же для сжатия больших данных, поэтому такой особенностью можно пренебречь.

Для решения задачи со сбросом словаря можно также было использовать специальный управляющий код, прочитав который декодер сделает заранее заданное действие. Один такой код в программе присутствует: когда кодер заканчивает со входными данными, он отправляет код 0x00 (которому в словаре соответствует пустая строка), прочитав такой код декодер завершит декодирование не зависимо от такого сколько бит осталось во входном потоке. Для такого подхода есть несколько причин. Во-первых поскольку длина кода измеряется в битах, то при побайтовой записи в файл, может остаться незаполненный до конца битами байт, а минимальное элемент который записать в файл равен полному байту – 8 бит. Решением этого является добавление нулевых битов до заполнения байта. Но полученные таким образом незначащие биты декодер вполне может принять за часть закодированных данных, тем самым неправильно интерпретируя архив.

Во-вторых идея отделить часть с закодированным файлом от остальных данных довольно логична. Для возможности проверки валидности архива в его конец записывается число 0xee и, для того чтобы была возможность оценить размер исходных данных без непосредственно декодирования в конец файла также записывается раз-

мер незакодированного файла.

После того как входные данные были закодированы алгоритмом LZW во временный файл, к нему применяется полустатический алгоритм кодирования Хаффмана. При первом проходе по файлу подсчитывается частота для каждого значения байта. После этого по полученным данным стоится дерево кодов Хаффмана. Построение реализовано через очередь с приоритетами. Хотя существует алгоритм реализации построения кодов за линейное время основанный на двух обычных очередях, первый метод имеет решающие для этого проекта особенности: удобочитаем и легкореализуем и, как следствие, его просто отлаживать. Для линейнго алгоритма же необходимо просматривать по два верхних элемента из очередей, для этого пришлось бы использовать несколько вставок удалений из дека, поскольку у классической очереди отсутствует возможность просматривать два элемента сверху, или же возвращать элементы не в конец очереди, а в начало. Почему некритична потеря времени при замене алгоритма работающего за линейное время? Построение дерева кодов лишь подготовительная работа перед кодированием, максимальное число листов в дереве - 0x100, то есть 256. По сравнению с дальнейшим кодированием, задача построения кодов Хаффмана пренебрежительно мала.

В реализации кодирования Хаффмана так же применяется символ выхода. Причём тут он особенно важен, так как если в LZW длина кода больше байта и понятно когда после нескольких (меньше 8) нулевых битов идёт конец файла, эти биты нужно игнорировать. А коды хаффмана могут быть длиной и два бита и один, при таких условиях отделить закодированные данные от фиктивных нулевый не представляется возможным. Для разрешения этой проблемы в дереве на этапе составления кодов вставляется символ выхода. Таким образом ему сопоставляется некоторый код в дереве. После конца входных данных кодер пишет этот код, и декодер приняв его, завершает процесс декодирования.

Разархивация – самая простая часть процесса, дерево Хаффмана уже записано в файле, нужно только его десериализовать, и воспользоваться, декодирование LZW использует в качестве словаря вектор, сопоставление коду слова, происходит посредством доступа по индексу.

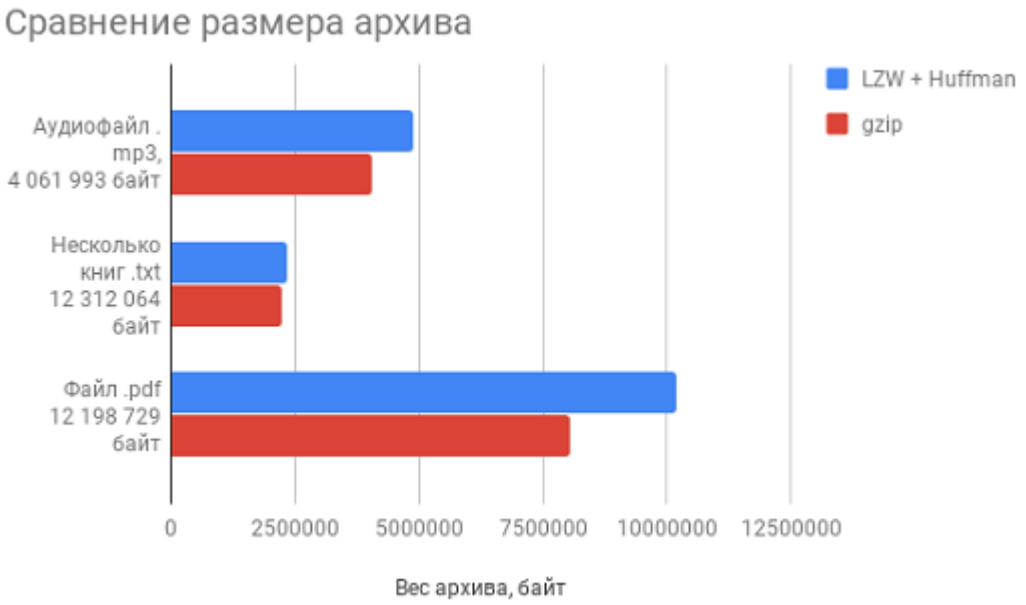
4 Особые случаи

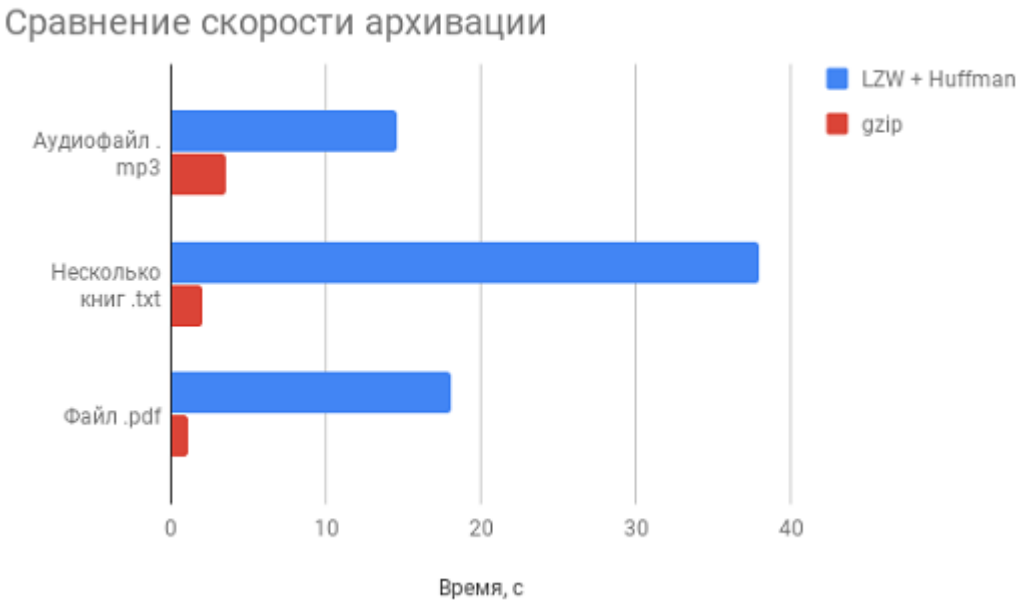
Отдельной темой является корректность алгоритма на пустом файле и на файлах содержащих одинаковые байты. Энтропия таких данных неопределена, и корректноть работы придётся гарантировать уже особенностями реализации. Коды выхода решают все проблемы. LZW закодирует пустой файл своим кодом выхода, таким образом на кодирование алгоритмом Хаффмана, пустые данные поступить не могут, но данные содержащие одинаковые байты вполне могут. Однако, поскольку в дерево

добавляется свой собственный ещё один символ выхода – в дереве гарантированно будет два символа. Ни один из этих символов не в корне, следовательно никакому символу не соответствует пустого кода. Пустой код при кодировании Хаффмана возможен, если в нём содержится ровно один узел – корень.

5 Сравнение с аналогами

Получившаяся программа сжимает хорошо, но работает в разы медленнее gzip.





Стоит отметить, что на уже сжатом .mp3 gzip архив получился на один килобайт меньше исходных данных, а архив LZW+Huffman на 24 КБ больше, связано это с сохранённым в архиве деревом Хаффмана.

В целом результаты не такие плохие, учитывая то, что gzip очень оптимизированная программа, написанная профессионалами.

6 Выводы

Выполнив курсовой проект по курсу «Дискретный анализ», я приобрёл практические и теоретические навыки в использовании знаний, полученных в течении курса и провел исследование в выбранной предметной области, разработал архиватор и изучил методы их создания и сжатия данных.

Реализация архиватора была очень интересным проектом, я получил ценные знания и опыт в процессе.

Алгоритм LZW оказался неожиданно очень удобным и в плане реализации и плане сжатия данных, я обязательно буду использовать его в дальнейшем.

Реализация декодирования была в разы проще по сравнению с кодированием: и в LZW и в алгоритме Хаффмана, оно сводилось к сопоставлению кодам символов и строк. Значительно больше усилий ушло на реализацию кодирования.

Использование кода выхода и решение проблемы с пустыми файлами было очень интересной практикой. Именно тут нужно использовать практический подход, поскольку математические и теоретические модели не до конца учитывают особенности реальности.

7 Исходный код

menu.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <experimental/filesystem>
#include <algorithm>
#include <iterator>

#include "lzw.hpp"
#include "huffman.hpp"

namespace fs = std::experimental::filesystem;

int Compress(std::string &filename, bool fromStdin, bool toStdout, bool keepFiles,
    bool fastest)
{
  std::ifstream inputData;
  std::ofstream outputTmp;
  std::ifstream inputTmp;
  std::ofstream outputData;

  uint32_t maxCodeLen;
  if (fastest)
  {
    maxCodeLen = 16;
  }
  else
  {
    maxCodeLen = 12;
  }
  std::string tmpname = filename + "~.tmp";
  outputTmp.open(tmpname, std::ofstream::out | std::ofstream::binary);
  if (!(outputTmp.is_open() && outputTmp.good()))
  {
    std::cerr << "Error while creating temporary file" << std::endl;
    outputTmp.close();
    return 1;
  }

  if (fromStdin)
  {
    std::istream &inputStd = std::cin;
    EncodeLZW(inputStd, outputTmp, maxCodeLen);
    outputTmp.close();
  }
  else
```

```
46  {
47    inputData.open(filename, std::ifstream::in | std::ifstream::binary);
48    if (!(inputData.is_open()))
49    {
50      std::cerr << "Error while opening file " << filename << std::endl;
51      inputData.close();
52      return 1;
53    }
54    EncodeLZW(inputData, outputTmp, maxCodeLen);
55    outputTmp.close();
56    inputData.close();
57  }
58
59  inputTmp.open(tmpname, std::ifstream::in | std::ifstream::binary);
60  if (!(inputTmp.is_open()))
61  {
62    std::cerr << "Error while opening temporary file" << std::endl;
63    inputTmp.close();
64    return 1;
65  }
66  if (toStdout)
67  {
68    std::ostream &outputStd = std::cout;
69    uint8_t byte = 0xee;
70    outputStd.write((char *)&byte, 1);
71    EncodeHuffman(inputTmp, outputStd);
72    inputTmp.close();
73  }
74  else
75  {
76    outputData.open(filename + ".Z", std::ofstream::out | std::ofstream::binary);
77    if (!(outputData.is_open() && outputData.good()))
78    {
79      std::cerr << "Error while creating archive file" << std::endl;
80      outputData.close();
81      return 1;
82    }
83    uint8_t specialByte = 0xee;
84    outputData.write((char *)&specialByte, 1);
85
86    EncodeHuffman(inputTmp, outputData);
87
88    //write down uncompressed size
89    outputData.write((char *)&specialByte, 1);
90    uint64_t uncompressedSize = fs::file_size(filename);
91    outputData.write((char *)&uncompressedSize, 8);
92
93    inputTmp.close();
94    outputData.close();
```

```
95    }
96    if (remove(tmpname.data()))
97    {
98      std::cerr << "Error while removing temporary file" << std::endl;
99      return 1;
100   }
101   if (!keepFiles && !fromStdin)
102   {
103     if (remove(filename.data()))
104     {
105       std::cerr << "Error while removing file " << filename << std::endl;
106       return 1;
107     }
108   }
109
110   return 0;
111 }
112 int Decompress(std::string &filename, bool fromStdin, bool toStdout, bool keepFiles)
113 {
114   std::ifstream inputData;
115   std::ofstream outputTmp;
116   std::ifstream inputTmp;
117   std::ofstream outputData;
118   uint32_t ret = 0;
119
120   std::string tmpname = filename + "~.tmp";
121   std::string unpackedname = "";
122   outputTmp.open(tmpname, std::ofstream::out | std::ofstream::binary);
123   if (!(outputTmp.is_open() && outputTmp.good()))
124   {
125     std::cerr << "Error while creating temporary file" << std::endl;
126     outputTmp.close();
127     return 1;
128   }
129
130   if (fromStdin)
131   {
132     std::istream &inputStd = std::cin;
133     uint8_t specialByte = 0xee;
134     uint8_t byte;
135     inputData.read((char *)&byte, 1);
136     if (byte == specialByte)
137     {
138       DecodeHuffman(inputStd, outputTmp);
139     }
140     else
141     {
142       ret = 1;
143     }
```

```
144     outputTmp.close();
145   }
146   else
147   {
148     unpackedname = filename.substr(0, filename.length() - 2);
149     if (filename.substr(filename.length() - 2, filename.length() - 1) != ".Z")
150     {
151       std::cerr << "File does not have extension .Z :" << filename << std::endl;
152       return 1;
153     }
154     inputData.open(filename, std::ifstream::in | std::ifstream::binary);
155     if (!(inputData.is_open()))
156     {
157       std::cerr << "Error while opening archive file " << filename << std::endl;
158       inputData.close();
159       return 1;
160     }
161
162     uint8_t specialByte = 0xee;
163     uint8_t byte;
164     inputData.clear();
165     inputData.read((char *)&byte, 1);
166     if (byte == specialByte)
167     {
168       DecodeHuffman(inputData, outputTmp);
169     }
170     else
171     {
172       ret = 1;
173     }
174     inputData.close();
175     outputTmp.close();
176   }
177   if (ret)
178   {
179     return ret;
180   }
181   inputTmp.open(tmpname, std::ifstream::in | std::ifstream::binary);
182   if (!(inputTmp.is_open()))
183   {
184     std::cerr << "Error while opening temporary file" << std::endl;
185     inputTmp.close();
186     return 1;
187   }
188   if (toStdout)
189   {
190     std::ostream &outputStd = std::cout;
191     ret = DecodeLZW(inputTmp, outputStd);
192     inputTmp.close();
```

```
193   }
194   else
195   {
196     outputData.open(unpackedname, std::ofstream::out | std::ofstream::binary);
197     if (!(outputData.is_open() && outputData.good()))
198     {
199       std::cerr << "Error while creating file: " << unpackedname << std::endl;
200       outputData.close();
201       return 1;
202     }
203     ret = DecodeLZW(inputTmp, outputData);
204     outputData.close();
205     inputTmp.close();
206   }
207 
208   if (remove(tmpname.data()))
209   {
210     std::cerr << "Error while removing temporary file" << std::endl;
211     return 1;
212   }
213 
214   if (!keepFiles && !fromStdin)
215   {
216     if (remove(filename.data()))
217     {
218       std::cerr << "Error while removing file " << filename << std::endl;
219       return 1;
220     }
221   }
222 
223   if (ret)
224   {
225     std::cerr << "Wrong archive structure:" << std::endl;
226   }
227   return ret;
228 }
229 void checkIntegrity(std::string &filename)
230 {
231   //so there is special byte 0xee in the beggining
232   //next goes bytes is serialized huffman tree
233   //basically any input data can be interpreterted as tree
234   std::ifstream inputData;
235   if (filename.substr(filename.length() - 2, filename.length() - 1) != ".Z")
236   {
237     std::cerr << filename << " is not a .Z archive" << std::endl;
238     return;
239   }
240   inputData.open(filename, std::ifstream::in | std::ifstream::binary);
241   uint8_t byte;
```

```
242   bool corruption = false;
243   inputData.read((char *)&byte, 1);
244   if (byte != 0xee)
245     corruption = true;
246   else
247   {
248     inputData.seekg(-9, inputData.end);
249     inputData.read((char *)&byte, 1);
250     if (byte != 0xee)
251       corruption = true;
252   }
253
254   inputData.close();
255   if (corruption)
256     std::cout << "Archive file " << filename << " corrupted" << std::endl;
257   else
258     std::cout << "Archive file " << filename << " is consistent" << std::endl;
259   //archive also should end with 0xee and uncomressed size
260 }
261 void getInfo(std::string &filename)
262 {
263
264   uint64_t uncompressedSize = 0, compressedSize;
265   std::string unpackedname = filename.substr(0, filename.length() - 2);
266   if (filename.substr(filename.length() - 2, filename.length() - 1) != ".Z")
267   {
268     std::cerr << filename << " is not a .Z archive" << std::endl;
269   }
270   else
271   {
272     std::cout << filename << ":" << std::endl;
273     compressedSize = fs::file_size(filename);
274     std::cout << "\tcompressed " << compressedSize << std::endl;
275
276     std::ifstream inputData;
277     inputData.open(filename, std::ifstream::in | std::ifstream::binary);
278     if (!(inputData.is_open()))
279     {
280       std::cerr << "Error while opening file " << filename << std::endl;
281       inputData.close();
282     }
283
284     inputData.seekg(-8, inputData.end);
285     inputData.read((char *)&uncompressedSize, 8);
286     inputData.close();
287     std::cout << "\tuncompressed " << uncompressedSize << std::endl;
288     std::cout << "\tuncompressed_name " << unpackedname << std::endl;
289   }
290 }
```

```
291 int main(int argc, char const *argv[])
292 {
293   bool labelWriteToStdout = false;
294   bool labelDecompress = false;
295   bool labelKeepFiles = false;
296   bool labelListProperties = false;
297   bool labelRecursive = false;
298   bool labelTestIntegrity = false;
299 
300   bool labelFastest = true; //default -9
301 
302   bool labelCompress = false;
303   bool labelReadFromStdin = false;
304   for (int i = 1; i < argc - 1; ++i)
305   {
306     std::string arg = argv[i];
307     if (arg == "-c")
308     {
309       labelWriteToStdout = true;
310     }
311     else if (arg == "-d")
312     {
313       labelDecompress = true;
314     }
315     else if (arg == "-k")
316     {
317       labelKeepFiles = true;
318     }
319     else if (arg == "-l")
320     {
321       labelListProperties = true;
322     }
323     else if (arg == "-r")
324     {
325       labelRecursive = true;
326     }
327     else if (arg == "-t")
328     {
329       labelTestIntegrity = true;
330     }
331     else if (arg == "-1")
332     {
333       labelFastest = true;
334     }
335     else if (arg == "-9")
336     {
337       labelFastest = false;
338     }
339   }
```

```
340   std::string filename = argv[argc - 1];
341   if (filename == "-")
342   {
343     labelWriteToStdout = true;
344     labelReadFromStdin = true;
345   }
346
347   if (!labelTestIntegrity && !labelDecompress && !labelListProperties)
348     labelCompress = true;
349
350   if (labelRecursive)
351   {
352     fs::recursive_directory_iterator dir(filename), end;
353     std::vector<std::string> files;
354     while (dir != end)
355     {
356       if (fs::is_regular_file(dir->path()))
357       {
358         files.push_back(dir->path());
359       }
360       ++dir;
361     }
362
363     if (labelCompress)
364     {
365       for (std::string &file : files)
366         if (Compress(file, false, false, labelKeepFiles, labelFastest))
367           return 1;
368     }
369     else if (labelDecompress)
370     {
371       for (std::string &file : files)
372         if (Decompress(file, false, false, labelKeepFiles))
373           return 1;
374     }
375     else if (labelListProperties && !labelReadFromStdin)
376     {
377       getInfo(filename);
378     }
379     else if (labelTestIntegrity)
380     {
381       checkIntegrity(filename);
382     }
383     return 0;
384   }
385   if (labelCompress)
386   {
387     return Compress(filename, labelReadFromStdin, labelWriteToStdout, labelKeepFiles,
          labelFastest);
```

```
388   }
389   else if (labelDecompress)
390   {
391     return Decompress(filename, labelReadFromStdin, labelWriteToStdout, labelKeepFiles)
          ;
392   }
393   else if (labelListProperties && !labelReadFromStdin)
394   {
395     getInfo(filename);
396   }
397   else if (labelTestIntegrity)
398   {
399     checkIntegrity(filename);
400   }
401   return 0;
402 }
```

huffman.hpp

```
1  #include <fstream>
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>
5  #include <queue>
6  #include <bitset>
7
8  #include "bitsBuffer.hpp"
9
10 class TreeNode
11 {
12 public:
13   TreeNode *left;
14   TreeNode *right;
15   uint16_t symbol;
16   uint32_t freq;
17   bool leaf;
18
19   //ENCODING
20   //no fields will be changed later
21   //leafs
22   TreeNode(uint16_t sym, uint32_t fr) : left(nullptr),
23                     right(nullptr),
24                     symbol(sym),
25                     freq(fr),
26                     leaf(true)
27   {
28   }
29
30   //internal
31   TreeNode(TreeNode *leftNode, TreeNode *rightNode) : left(leftNode),
```

```
32                          right(rightNode),
33                          symbol(0),
34                          freq(leftNode->freq + rightNode->freq),
35                          leaf(false)
36    {
37    }
38
39    //DECODING
40    //fields with links will be changed
41    //leaf
42    TreeNode(uint16_t sym) : left(nullptr), //empty
43                right(nullptr), //empty
44                symbol(sym),
45                freq(0), //does not matter
46                leaf(true)
47    {
48    }
49
50    //internal
51    TreeNode() : left(nullptr), //will be init later
52          right(nullptr), //also
53          symbol(0),
54          freq(0), //does not matter
55          leaf(false)
56    {
57    }
58
59    virtual ~TreeNode()
60    {
61      if (!leaf)
62      {
63        delete left;
64        delete right;
65      }
66    }
67  };
68
69  void DFS(const TreeNode *node, std::vector<std::vector<bool>> &table, std::vector<bool
        > &path)
70  {
71    if (node->leaf)
72    {
73      table[node->symbol] = path;
74      return;
75    }
76    path.push_back(false);
77    DFS(node->left, table, path);
78    path.pop_back();
79    path.push_back(true);
```

```
80    DFS(node->right, table, path);
81    path.pop_back();
82  }
83
84  void GetCodes(TreeNode *root, std::vector<std::vector<bool>> &table)
85  {
86    std::vector<bool> path;
87    DFS(root, table, path);
88  }
89
90  void DFSSerialize(const TreeNode *node, BitsOutputBuffer &buffer)
91  {
92    if (node->leaf)
93    {
94      buffer.AddToBuffer(true);
95      buffer.AddToBuffer(node->symbol);
96      return;
97    }
98    buffer.AddToBuffer(false);
99    DFSSerialize(node->left, buffer);
100   DFSSerialize(node->right, buffer);
101 }
102
103 void SerializeTree(const TreeNode *root, BitsOutputBuffer &buffer)
104 {
105   DFSSerialize(root, buffer);
106 }
107
108 ////////////////
109
110 TreeNode *DFSDeserialize(BitsInputBuffer &buffer)
111 {
112   if (buffer.GetBits(1))
113   {
114     //input bit is 1
115     //leaf
116     return new TreeNode((uint16_t)buffer.GetBits(16));
117   }
118   else
119   {
120     //input bit is 0
121     //internal node
122     TreeNode *node = new TreeNode();
123     node->left = DFSDeserialize(buffer);
124     node->right = DFSDeserialize(buffer);
125     return node;
126   }
127 }
128
```

```
129 TreeNode *DeserializeTree(BitsInputBuffer &buffer)
130 {
131   TreeNode *ret = DFSDeserialize(buffer);
132   return ret;
133 }
134
135 struct Comp
136 {
137   bool operator()(const TreeNode *a, const TreeNode *b)
138   {
139     return a->freq > b->freq;
140   }
141 };
142
143 int EncodeHuffman(std::istream &inputData, std::ostream &outputData)
144 {
145   //HUFFMAN CODES
146
147   //preparing codes
148   std::vector<uint32_t> freqStats(0xff + 1, 0);
149   uint8_t byte;
150   while (!inputData.eof())
151   {
152     inputData.read((char *)&byte, 1);
153     freqStats[byte]++;
154     inputData.peek();
155   }
156   std::priority_queue<TreeNode *, std::vector<TreeNode *>, Comp> pq;
157   uint16_t sym = 0;
158   for (uint32_t i = 0; i < freqStats.size(); ++i, ++sym)
159   {
160     if (freqStats[i])
161       pq.push(new TreeNode(sym, freqStats[i]));
162   }
163   uint16_t exitCode = 0xff + 1;
164   pq.push(new TreeNode(exitCode, 0));
165   freqStats.clear(); //no longer needed
166   TreeNode *left, *rigth;
167   while (pq.size() > 1)
168   {
169     left = pq.top();
170     pq.pop();
171     rigth = pq.top();
172     pq.pop();
173     pq.push(new TreeNode(left, rigth));
174   }
175   TreeNode *root = pq.top();
176   pq.pop();
177   std::vector<std::vector<bool>> codeTable(0xff + 2);
```

```
178   //some elements will be empty, buut [] works in O(1)
179   GetCodes(root, codeTable);
180   codeTable.shrink_to_fit(); //no more changes in codeTable
181
182   BitsOutputBuffer outputBuffer(outputData);
183   SerializeTree(root, outputBuffer);
184   delete root; //tree no longer needed - there is codeTable
185
186   //second run through file
187   inputData.clear();
188   inputData.seekg(0, std::ios::beg); //seekg == seek get position
189   //zero is offset from position
190
191   while (!inputData.eof())
192   {
193     inputData.read((char *)&byte, 1);
194     outputBuffer.AddToBuffer(codeTable[byte]);
195     inputData.peek(); //to trigger eof
196   }
197   outputBuffer.AddToBuffer(codeTable[exitCode]);
198   outputBuffer.Flush();
199   codeTable.clear();
200 }
201
202 ////////////////////////////////////////////////////////////////////////
203
204 void DecodeHuffman(std::istream &encodedData, std::ostream &decodedData)
205 {
206
207   BitsInputBuffer inputBuffer(encodedData);
208   //get tree from file
209   TreeNode *root = DeserializeTree(inputBuffer);
210   TreeNode *currNode = root;
211   uint16_t exitCode = 0xff + 1;
212   while (!inputBuffer.Empty())
213   {
214     //if (encodedData.eof()) {
215     // byte += 2;
216     //}
217     if (inputBuffer.GetBits(1))
218       currNode = currNode->right;
219     else
220       currNode = currNode->left;
221     if (currNode->leaf)
222     {
223       if (currNode->symbol == exitCode)
224         return;
225       decodedData.write((char *)&currNode->symbol, 1);
226       currNode = root;
```

```
227     }
228   }
229 }
```

lzw.hpp

```
1  #include <iostream>
2  #include <vector>
3  #include <map>
4  #include <string>
5  #include <math.h>
6  #include <bitset>
7  #include "bitsBuffer.hpp"
8  class TrieNode
9  {
10   friend class Trie;
11   TrieNode(uint32_t num) : code(num)
12   {
13   }
14   void Clear()
15   {
16     for (std::pair<uint8_t, TrieNode *> elem : path)
17       delete elem.second;
18     path.clear();
19   }
20
21   virtual ~TrieNode()
22   {
23     for (std::pair<uint8_t, TrieNode *> elem : path)
24       delete elem.second;
25   }
26
27 private:
28   std::map<uint8_t, TrieNode *> path;
29   uint32_t code;
30 };
31
32 class Trie
33 {
34 public:
35   //next code to give
36   //also used in calculating of codeLen
37   uint32_t maxCode;
38
39   //current lenght of code in bits
40   uint32_t codeLen;
41
42   //next value of maxCode that will increase codeLen
43   uint32_t nextIncrease;
44   Trie() : path(0xff + 1),
```

```
45        maxCode(1),
46        nextIncrease(2),
47        codeLen(1),
48        currentNode(nullptr)
49    {
50      uint8_t sym = 0x0;
51      for (uint32_t i = 0; i <= 0xff; ++sym, ++i)
52      {
53        //conversion from char to string
54        path[sym] = new TrieNode(maxCode);
55        if (maxCode == nextIncrease)
56        {
57          nextIncrease *= 2;
58          ++codeLen;
59        }
60        ++maxCode;
61      }
62    }
63
64    void Clear()
65    {
66      //root array stays in place
67      maxCode = 0xff + 2;
68      nextIncrease = 512;
69      codeLen = 9;
70      currentNode = nullptr;
71      for (TrieNode *node : path)
72        node->Clear();
73    }
74
75    uint32_t GetCode(std::string &str)
76    {
77      TrieNode *currNode = path[(uint8_t)str[0]];
78      for (int i = 1; i < str.length(); ++i)
79      {
80        currNode = currNode->path[(uint8_t)str[i]];
81      }
82      return currNode->code;
83    }
84
85    TrieNode *CheckWord(uint8_t sym)
86    {
87      if (currentNode)
88      {
89        if (currentNode->path.count(sym))
90        {
91          currentNode = currentNode->path[sym];
92        }
93        else
```

```
94        {
95          currentNode = nullptr;
96        }
97      }
98      else
99      {
100       currentNode = this->path[sym];
101     }
102     return currentNode;
103   }
104
105   void AddWord(std::string &str)
106   {
107     TrieNode *currNode = path[(uint8_t)str[0]];
108     for (int i = 1; i < str.length() - 1; ++i)
109     {
110       currNode = currNode->path[(uint8_t)str[i]];
111     }
112     currNode->path[(uint8_t)str[str.length() - 1]] = new TrieNode(maxCode);
113     if (maxCode == nextIncrease)
114     {
115       nextIncrease *= 2;
116       ++codeLen;
117     }
118     ++maxCode;
119   }
120   virtual ~Trie()
121   {
122     for (TrieNode *node : path)
123       delete node;
124   }
125
126 private:
127   TrieNode *currentNode;
128   std::vector<TrieNode *> path;
129 };
130
131 int32_t GetCodeLen(const uint32_t maxCode)
132 {
133   return (uint32_t)ceil(log2(maxCode));
134 }
135
136 void EncodeLZW(std::istream &inputData, std::ostream &outputData, uint32_t maxCodeLen)
137 {
138   //LZW ENCODING
139   Trie trie;
140   uint32_t exitCode = 0;
141   std::string prev;
142   std::string full;
```

```
143   uint8_t curr;
144   BitsOutputBuffer buffer(outputData);
145   //write in file maxCodeLen
146   buffer.AddToBuffer(maxCodeLen, 32);
147   inputData.peek(); //check for eof
148   while (!inputData.eof())
149   {
150     inputData.read((char *)&curr, 1);
151     full += curr;
152     if (trie.CheckWord(curr))
153     {
154       prev += curr;
155     }
156     else
157     {
158       buffer.AddToBuffer(trie.GetCode(prev), trie.codeLen);
159       trie.AddWord(full);
160       if (trie.codeLen > maxCodeLen)
161       {
162         trie.Clear();
163       }
164       prev = curr;
165       full = curr;
166       trie.CheckWord(curr);
167     }
168     inputData.peek();
169   }
170   if (!(prev.empty()))
171   {
172     buffer.AddToBuffer(trie.GetCode(prev), trie.codeLen);
173   }
174   buffer.AddToBuffer(exitCode, trie.codeLen);
175   buffer.Flush();
176 }
177
178 int DecodeLZW(std::istream &encodedData, std::ostream &decodedData)
179 {
180   BitsInputBuffer buffer(encodedData);
181   uint32_t maxCodeLen = buffer.GetBits(32);
182   if (!(maxCodeLen == 12 || maxCodeLen == 16))
183   {
184     return 1;
185   }
186   std::vector<std::string> dict;
187   dict.push_back("");
188   uint8_t sym = 0x0;
189   for (uint32_t i = 0; i <= 0xff; ++sym, ++i)
190   {
191     //conversion from char to string
```

```
192     dict.push_back(std::string(1, sym));
193   }
194
195   std::string currDecode;
196   //get codeLen bits from buffer
197   uint32_t exitCode = 0;
198
199   uint32_t code = buffer.GetBits(GetCodeLen(dict.size()));
200   std::string prev = dict[code]; //first code processes separatly
201   std::string full = prev;
202   while (!buffer.Empty())
203   {
204     code = buffer.GetBits(GetCodeLen(dict.size() + 1));
205
206     if (code == exitCode)
207     {
208       break;
209     }
210
211     if (code == dict.size())
212     {
213       //code that is not in dictionary
214       //previous decoded string + its first letter
215       currDecode = prev[0];
216       full += currDecode;
217       dict.push_back(full);
218       decodedData.write(prev.data(), prev.length());
219       prev = dict.back();
220       full = prev;
221     }
222     else
223     {
224       currDecode = dict[code];
225       full += currDecode[0];
226       dict.push_back(full);
227       decodedData.write(prev.data(), prev.length());
228       prev = currDecode;
229       full = prev;
230     }
231     //one more than currently in dict is significant
232     //its part of decoding
233     if (GetCodeLen(dict.size() + 1) > maxCodeLen)
234     {
235       decodedData.write(prev.data(), prev.length());
236       dict.resize(0xff + 2);
237       prev = dict[buffer.GetBits(GetCodeLen(dict.size() + 1))];
238       full = prev;
239     }
240   }
```

```
241   decodedData.write(prev.data(), prev.length());
242
243   return 0;
244 }
```

bitsBuffer.hpp

```
1  #ifndef BITSBUFFER_HPP
2  #define BITSBUFFER_HPP
3
4  #include <fstream>
5  #include <iostream>
6  #include <vector>
7  #include <algorithm>
8
9  class BitsInputBuffer
10 {
11 public:
12   BitsInputBuffer(std::istream &inputFile) : file(inputFile)
13   {
14   }
15
16   void Flush()
17   {
18     buffer = 0;
19     bufferCount = 0;
20   }
21   bool Empty()
22   {
23     file.peek();
24     return (bufferCount == 0) && file.eof();
25   }
26
27   uint32_t GetBits(uint32_t len)
28   {
29     uint32_t symbol = 0;
30     //fill buffer
31     if (bufferCount < len)
32       ReadBits(len);
33
34     //still not enough
35     if (bufferCount < len)
36     {
37       //input data ended
38       //return what have
39       uint32_t ret = buffer;
40       file.peek();
41       buffer = 0;
42       bufferCount = 0;
43       return ret;
```

```
44      }
45
46      //get len bits from begin of bufferCount
47      bufferCount -= len;
48      symbol = (uint32_t)(buffer >> bufferCount);
49      //delete first len bits stored
50      if (bufferCount == 0)
51      {
52        //shifts for full size do not work
53        //left shift count >= width of type [-Wshift-count-overflow]
54        symbol = buffer;
55        buffer = 0;
56      }
57      else
58      {
59        buffer = buffer << (64 - bufferCount);
60        buffer = buffer >> (64 - bufferCount);
61      }
62      return symbol;
63    }
64
65  private:
66    void ReadBits(uint32_t len)
67    {
68      uint8_t byte = 0;
69      while (bufferCount < len)
70      {
71        byte = 0;
72        file.read((char *)&byte, 1);
73        buffer = (buffer << 8) | (uint64_t)(byte);
74        bufferCount += 8;
75        file.peek(); //to trigger eof
76        if (file.eof())
77        {
78          return;
79        }
80      }
81    }
82
83    std::istream &file;
84    uint64_t buffer = 0;
85    uint32_t bufferCount = 0;
86  };
87
88  class BitsOutputBuffer
89  {
90  public:
91    BitsOutputBuffer(std::ostream &outputFile) : file(outputFile) {}
92
```

```
93  void AddToBuffer(uint32_t code, uint32_t len)
94  {
95    //LZW special
96    buffer = (buffer << len) | (uint64_t)code;
97    bufferCount += len;
98    if (bufferCount >= 24)
99      WriteBits();
100 }
101
102 void AddToBuffer(std::vector<bool> &code)
103 {
104   for (bool k : code)
105   {
106     //put in code
107     if (k)
108       buffer = (buffer << 1) | 1;
109     else
110       buffer = (buffer << 1);
111     ++bufferCount;
112     if (bufferCount >= 32)
113       WriteBits();
114   }
115 }
116
117 void AddToBuffer(uint8_t code)
118 {
119   buffer = (buffer << 8) | (uint64_t)code;
120   bufferCount += 8;
121   WriteBits();
122 }
123
124 void AddToBuffer(uint16_t code)
125 {
126   buffer = (buffer << 16) | (uint64_t)code;
127   bufferCount += 16;
128   WriteBits();
129 }
130
131 void AddToBuffer(bool code)
132 {
133   if (code)
134     buffer = (buffer << 1) | 1;
135   else
136     buffer = (buffer << 1);
137
138   ++bufferCount;
139   if (bufferCount >= 24)
140   {
141     WriteBits();
```

```
142     }
143   }
144
145   void Flush()
146   {
147     WriteBits();
148     uint8_t byte = 0;
149     //fill with 0 empty space
150     if (bufferCount == 0)
151       //no incomplete bytes
152       //so no filler needed
153       return;
154     byte = (uint8_t)(buffer << (8 - bufferCount));
155     file.write((char *)&byte, 1);
156     buffer = 0;
157     bufferCount = 0;
158   }
159
160   virtual ~BitsOutputBuffer() {}
161
162 private:
163   std::ostream &file;
164   uint64_t buffer = 0;
165   uint32_t bufferCount = 0;
166   void WriteBits()
167   {
168     uint8_t byte = 0;
169     while (bufferCount >= 8)
170     {
171       //8 bits from begin of buffer
172       bufferCount -= 8;
173       byte = (uint8_t)(buffer >> (bufferCount));
174       //delete first 8 bits stored
175       if (bufferCount == 0)
176       {
177         //shifts for full size do not work
178         //left shift count >= width of type [-Wshift-count-overflow]
179         buffer = 0;
180       }
181       else
182       {
183         buffer = buffer << (64 - bufferCount);
184         buffer = buffer >> (64 - bufferCount);
185       }
186       file.write((char *)&byte, 1);
187     }
188   }
189 };
190
```

```
191
192 #endif
```