Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

Сжатие данных помощью кодов Хаффмана

по дисциплине: «Объектно-ориентированное программирование»

Выполнил студент:
гр. в3530904/00030 Дружинин Н.В.

Руководитель:
доцент Круглов С.К.

Санкт-Петербург
2022

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

Задание на курсовую работу

Сжатие данных помощью кодов Хаффмана

по дисциплине: «Объектно-ориентированное программирование»

ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Студенту группы № в3530904/00030 Дружинину Н.В.
1. Тема работы:
Сжатие данных помощью кодов Хаффмана
2. Исходные данные для работы:
"Алгоритмы: построение и анализ"Томас Кормен, Чарльз Эрик Лейзерсон, Рональд Линн Ривест, Клиффорд Штайн
Википедия https://ru.wikipedia.org
3. Вопросы, подлежащие проработке:
Принцип префиксного кодирования информации, построение оптимального кодового дерева, построение отображения код-символ на основе построенного дерева.

Руководитель: доцент Круглов С.К.
Задание принял к исполнению: студент Дружинин Н.В.

Содержание

1 Введение

1.1 Коды Хаффмана

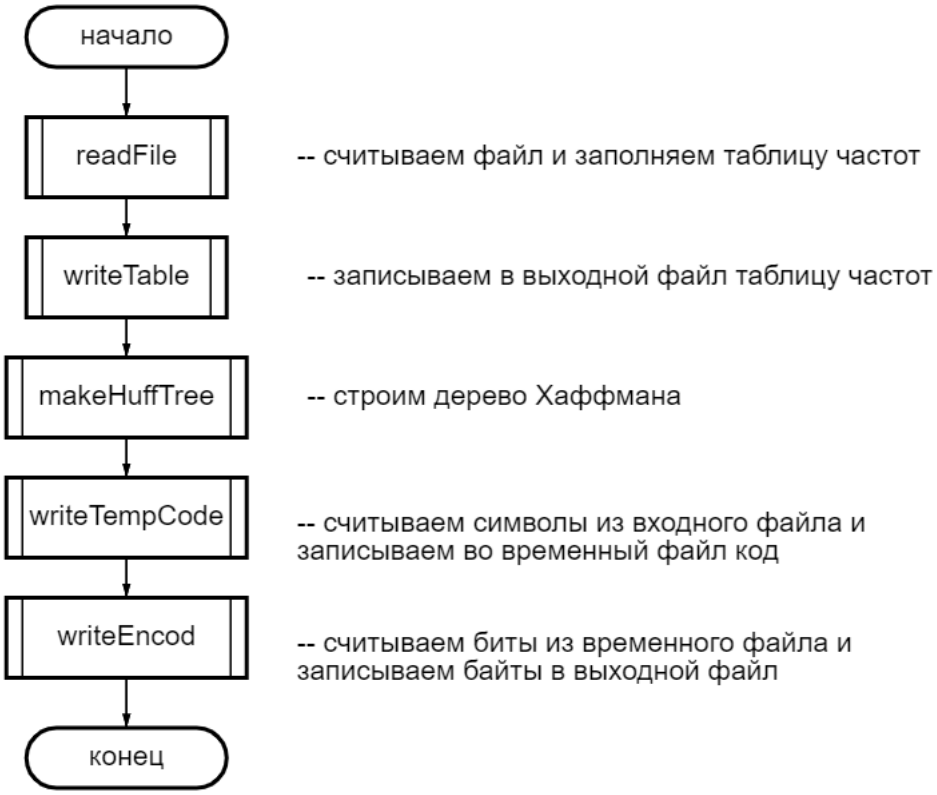
Коды Хаффмана (Huffman codes) - эффективный метод сжатия данных, который, в зависимости от характеристик этих данных, обычно позволяет сэкономить от 20 до 90% объема.
Данные рассматриваются как последовательность символов. В жадном алгоритме Хаффмана используется таблица, содержащая частоты появления тех или инных символов. С помощью этой таблицы определяется оптимальное представление каждого символа в виде бинарной строки.
Наиболее лучшие результаты кодирования удается получить используя коды переменной длины или неравномерные коды, поскольку наиболее часто встречающимся символам сопоставляются короткие кодовые слова, а редко встречающимся - длинные.
В данной работе используются **префиксные коды**, т.е. такие коды, в которых никакое кодовое слово не является префиксом какого-то другого кодового слова. Это связано с тем, что префиксное кодирование позволяет упростить процесс декодирования и позволяет однозначно идентифицировать символ по кодовому слову.
Оптимальный код всегда представлен *полным* бинарным деревом, каждый узел которого (кроме листьев) имеет по два дочерних узла. Если C - алфавит, из которого извлекаются кодируемые символы, и все частоты, с которыми встречаются символы, положительны, то дерево, представляющее оптимальный префиксный код, содержит ровно $\mid C \mid$ листьев, по одному для каждого символа из множества C, и ровно $\mid C \mid -1$ внутренних узлов.
Если имеется дерево T, соответствеющее префиксонму коду, легко подсчитать количество битов, которые потребуются для кодирования файла:

$$B(T) = \sum_{c \in C} c.freq * d_T(c)$$

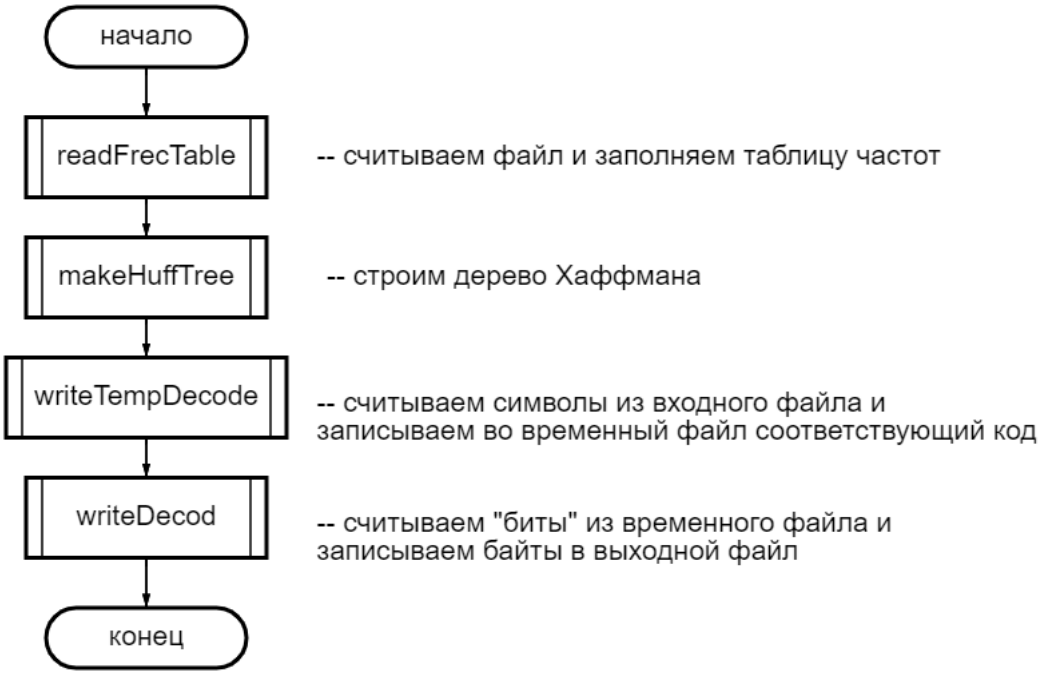
- $c.freq$ - частота появления символа c в файле
- $d_T(c)$ - глубина листа, представляющего символ c в дереве, а так же длина слова, кодирующего символ c

2 Блок-схема алгоритма

2.1 Кодирование файла



2.2 Декодирование файла



3 Описание программы

3.1 Ввод-вывод файла

Для считывания информации из файла в программе используется функция:

```
FILE* fp = fopen(filename, "rb");
```

которая возвращает указатель на файл для считывания байтов. Для составления таблицы частот символов используется ассоциативный контейнер *map* из стандартной библиотеки:

```
std::map<unsigned char, size_t> freq;
```

При кодировании и декодировании информации используется временный файл, в котором каждый символ представлен соответствующим кодом из "0"и "1". Поскольку минимальной единицей информации при чтении или записи является байт, то для записи битов используется структура *union*, которая позволяет переиспользовать одну и ту же область памяти для хранения разных полей данных:

```
    union code
    {
        unsigned char chh;
        struct byte
        {
            unsigned b1:1;
            unsigned b2:1;
            unsigned b3:1;
            unsigned b4:1;
            unsigned b5:1;
            unsigned b6:1;
            unsigned b7:1;
            unsigned b8:1;
        } byte;
    };
```

3.2 Построение дерева Хаффмана

Для построения дерева Хаффмана используется очередь с приоритетами *std* :: *priority_queue* из стандартной библиотеки:

```
std::priority_queue<nodePtr, std::vector<nodePtr>, compare> q;
```

которая реализована через вектор умных указателей *std* :: *shared_ptr* на структуру *huffnode*:

```
    struct huffnode
    {
        unsigned char ch;
        size_t freq;
        bool node;
        nodePtr left, right;
        huffnode(unsigned char ch_, size_t freq_)
        {
            left = right = nullptr;
            this->ch = ch_;
            this->node = false;
            this->freq = freq_;
        }
        huffnode(size_t freq_)
        {
            left = right = nullptr;
            this->node = true;
            this->freq = freq_;
        }
    };
```

Для сортировки элементов очереди *q* используется предикат *compare*:

```
    struct compare
    {
        bool operator()(nodePtr left, nodePtr right)
        {
            return(left->freq > right->freq );
        }
    };
```

Для составления полного бинарного дерева используется умный указатель на указатель *std :: shared_ptr* на структуру *huffnode*:

```
using structNodePtr = std::shared_ptr<struct huffnode>;
//...
while (q.size() != 1)
        {
            left = q.top();
            q.pop();
            right = q.top();
            q.pop();
            top = std::make_shared<huffnode>(left->freq + right->freq);
            top->left = left;
            top->right = right;
            q.push(top);
        }
```

Рекурсивная функция *makeBitCode* проходится по дереву и составляет кодовые слова для каждого символа:

```
//...
    void makeBitCode(structNodePtr root, std::string str)
    {
        if (root == nullptr)
        {
            return;
        }
        if (root->node == false)
        {
            inoutput::bitcode[root->ch] = str;
        }

        makeBitCode(root->left, str + "0");
        makeBitCode(root->right, str + "1");
    }
//...
```

4 Листинг программмы

4.1 main.cpp

```cpp
#include "input_output.h"
#include "encode_decode.h"

using namespace inoutput;
using namespace encode_decode;

int main()
{
    try
    {

        char oper;
        std::string inFile, outFile, tempFile{"temp.txt"}, suff;
        std::cout << "To archivate press 1, to unzip press 2\n";
        std::cin >> oper;
        switch (oper)
        {
        case '1':
            {
                //getfilename
                std::cout << "Please enter file name ";
                std::cin >> inFile;
                outFile = inFile;
                suff = "_encode";
                outFile.insert(outFile.begin() + outFile.find_last_of('.'), std
                    ::begin(suff), std::end(suff));

                //open and read input file
                size_t filesize;
                readFile(inFile.c_str());

                //write frequencies table
                writeTable(outFile.c_str());
                printMap(freq);

                //make Huffman tree
                makeHuffTree();
                printMap(bitcode);

                //write bit codes in temporary file
                writeTempCode(inFile.c_str(), tempFile.c_str(), filesize); //
                    pif.txt

                //get encode file
                writeEncod(tempFile.c_str(), outFile.c_str(), filesize);
                break;
            }
        case '2':
            {
                //getfilename
                std::cout << "Please enter file name ";
                std::cin >> inFile;
                outFile = inFile;
                suff = "_decode";
                outFile.insert(outFile.begin() + outFile.find_last_of('.'), std
                    ::begin(suff), std::end(suff));

                //decode
                //get frquencies table
                long int offset;
                readFrecTable(inFile.c_str(), offset);
                //printMap(freq);
```

```
60
61                  //create Huffman tree
62                  makeHuffTree();
63                  //printMap(bitcode);
64
65                  writeTempDecode(inFile.c_str(), tempFile.c_str(), offset);
66                  writeDecod(tempFile.c_str(), outFile.c_str());
67                  break;
68              }
69
70          }
71
72          return 0;
73      }
74      catch (Exception* e)
75      {
76          std::cout << e->what() << "\n";
77      }
78      catch (...)
79      {
80          std::cout << "Unknown error \n";
81      }
82  }
```

4.2 input_output.h

```
1  #ifndef INOUT_H
2  #define INOUT_H
3
4  #include <iostream>
5  #include <cstdio>
6  #include <map>
7  #include <bitset>
8  #include <string>
9  #include <algorithm>
10 #include "exception.h"
11
12
13 namespace inoutput
14 {
15     union code
16     {
17         unsigned char chh;
18         struct byte
19         {
20             unsigned b1:1;
21             unsigned b2:1;
22             unsigned b3:1;
23             unsigned b4:1;
24             unsigned b5:1;
25             unsigned b6:1;
26             unsigned b7:1;
27             unsigned b8:1;
28         } byte;
29     };
30
31     std::map<unsigned char, size_t> freq;
32     std::map<unsigned char, std::string> bitcode;
33
34     void readFile(const char* filename)
35     {
36         FILE* fp = fopen(filename, "rb");
37         if (!fp)
38         {
39             throw Exception("Can not open file ", filename);
```

```
40          }
41
42          int ch;
43          while ((ch = fgetc(fp)) != EOF)
44          {
45              freq[ch]++;
46          }
47
48          fclose(fp);
49      }
50
51      template<class T, class N>
52      void printMap(std::map<T, N> m)
53      {
54          for (const auto& it : m)
55          {
56              std::cout << it.first << " " << std::bitset<8>(it.first) << " = "
57              << std::dec << it.second << " - " << std::hex << it.second << "\n";
58          }
59      }
60
61      void writeTable(const char* filename)
62      {
63          FILE* fp = fopen(filename, "wb");
64
65          if (!fp)
66          {
67              throw Exception("Can not open file ", filename);
68          }
69
70          int numOfUnicElems = freq.size();
71          fwrite(&numOfUnicElems, sizeof(int), 1, fp);
72          for(const auto it : freq)
73          {
74              fwrite(&it.first, sizeof(char), 1, fp);
75              fwrite(&it.second, sizeof(int), 1, fp);
76          }
77
78          fclose(fp);
79      }
80
81      void writeTempCode(const char* infile, const char* outfile, size_t&
            filesize)
82      {
83          FILE* outfp = fopen(outfile, "wb"); //ab
84
85          if (!outfp)
86          {
87              throw Exception("Can not open file ", outfile);
88          }
89
90          FILE* infp = fopen(infile, "rb");
91
92          if (!infp)
93          {
94              throw Exception("Can not open file ", infile);
95          }
96
97          int ch;
98          size_t counter = 0;
99          while ((ch = fgetc(infp)) != EOF)
100         {
101             std::string tmp = bitcode[ch];
102             for (size_t i = 0; i < tmp.size(); ++i)
103             {
104                 fputc(tmp[i], outfp);
```

```
105                 ++counter;
106             }
107         }
108         filesize = counter;
109 
110         fclose(outfp);
111         fclose(infp);
112     }
113 
114     void writeEncod(const char* infile, const char* outfile, const size_t
        filesize)
115     {
116         FILE* outfp = fopen(outfile, "ab"); //ab
117 
118         if (!outfp)
119         {
120             throw Exception("Can not open file ", outfile);
121         }
122 
123         FILE* infp = fopen(infile, "rb");
124 
125         if (!infp)
126         {
127             throw Exception("Can not open file ", infile);
128         }
129 
130         char byte_[8];
131         char ch;
132         union code byteCode;
133         for (int i = 0; i < filesize; ++i)
134         {
135             fread(&ch, sizeof(char), 1, infp);
136             byte_[(i % 8)] = ch;
137             if((i % 8) == 7)
138             {
139                 byteCode.byte.b1 = byte_[0] - '0';
140                 byteCode.byte.b2 = byte_[1] - '0';
141                 byteCode.byte.b3 = byte_[2] - '0';
142                 byteCode.byte.b4 = byte_[3] - '0';
143                 byteCode.byte.b5 = byte_[4] - '0';
144                 byteCode.byte.b6 = byte_[5] - '0';
145                 byteCode.byte.b7 = byte_[6] - '0';
146                 byteCode.byte.b8 = byte_[7] - '0';
147                 fwrite(&byteCode.chh, sizeof(unsigned char), 1, outfp);
148             }
149         }
150 
151         fclose(infp);
152         fclose(outfp);
153     }
154 
155     void readFrecTable(const char* filename, long int& offset)
156     {
157         FILE* fp = fopen(filename, "rb");
158         if (!fp)
159         {
160            throw Exception("Can not open file ", filename);
161         }
162 
163         int numOfUnicElems = 0;
164         fread(&numOfUnicElems, sizeof(int), 1, fp);
165 
166         inoutput::freq.clear();
167         int counter = 0, frequency;
168         char ch;
169         while (counter < numOfUnicElems)
```

```
170         {
171             fread(&ch, sizeof(char), 1, fp);
172             fread(&frequency, sizeof(int), 1, fp);
173             inoutput::freq[ch] = frequency;
174             ++counter;
175         }
176         offset = sizeof(char) * numOfUnicElems + sizeof(int) * numOfUnicElems +
                sizeof(int);
177         fclose(fp);
178     }
179
180     char findCharInBitcode(const std::string& str, bool& flag)
181     {
182         for(auto& elem : bitcode)
183         {
184             if (elem.second == str)
185             {
186                 //std::cout << "I found symbol " << elem.first << "\n";
187                 flag = true;
188                 return elem.first;
189             }
190         }
191         return '0';
192     }
193
194     void writeDecod(const char* infile, const char* outfile)
195     {
196         FILE* outfp = fopen(outfile, "wb"); //ab
197
198         if (!outfp)
199         {
200             throw Exception("Can not open file ", outfile);
201         }
202
203         FILE* infp = fopen(infile, "rb");
204
205         if (!infp)
206         {
207             throw Exception("Can not open file ", infile);
208         }
209
210         char ch;
211         std::string str;
212         bool flag;
213         while (!feof(infp))
214         {
215             flag = false;
216             while(str.size() < 3)
217             {
218                 fread(&ch, sizeof(char), 1, infp);
219                 str += ch;
220             }
221
222             ch = findCharInBitcode(str, flag);
223             if (flag)
224             {
225                 fputc(ch, outfp);
226                 str.clear();
227             }
228             else
229             {
230                 while (!flag && !feof(infp))
231                 {
232                     fread(&ch, sizeof(char), 1, infp);
233                     str += ch;
234                     ch = findCharInBitcode(str, flag);
```

```
235                     if (flag)
236                     {
237                         fputc(ch, outfp);
238                         str.clear();
239                     }
240                 }
241             }
242         }
243         fclose(infp);
244         fclose(outfp);
245     }
246 
247     void writeTempDecode(const char* infile, const char* outfile, long int
        offset)
248     {
249         FILE* outfp = fopen(outfile, "wb");
250 
251         if (!outfp)
252         {
253             throw Exception("Can not open file ", outfile);
254         }
255 
256         FILE* infp = fopen(infile, "rb");
257 
258         if (!infp)
259         {
260             throw Exception("Can not open file ", infile);
261         }
262 
263         unsigned char ch;
264         if(fseek(infp, offset, 0) != 0)
265         {
266             throw Exception("Can not make an ofset");
267         }
268 
269         union code byteCode;
270         std::string strByte = "";
271         while (!feof(infp))
272         {
273             fread(&ch, sizeof(char), 1, infp);
274             byteCode.chh = ch;
275             strByte += (byteCode.byte.b1 + '0');
276             strByte += (byteCode.byte.b2 + '0');
277             strByte += (byteCode.byte.b3 + '0');
278             strByte += (byteCode.byte.b4 + '0');
279             strByte += (byteCode.byte.b5 + '0');
280             strByte += (byteCode.byte.b6 + '0');
281             strByte += (byteCode.byte.b7 + '0');
282             strByte += (byteCode.byte.b8 + '0');
283             for_each (strByte.begin(), strByte.end(), [&outfp](const char& c) {
                    fputc(c, outfp); });
284             strByte.clear();
285         }
286 
287         fclose(outfp);
288         fclose(infp);
289     }
290 }
291 
292 
293 
294 #endif // INOUT_H
```

4.3 encode_decode.h

```cpp
#ifndef ENCODE_H
#define ENCODE_H

#include <memory>
#include <queue>
#include <vector>

namespace encode_decode
{
    struct huffnode;
    using nodePtr = std::shared_ptr<huffnode>;
    using structNodePtr = std::shared_ptr<struct huffnode>;

    struct huffnode
    {
        unsigned char ch;
        size_t freq;
        bool node;
        nodePtr left, right;
        huffnode(unsigned char ch_, size_t freq_)
        {
            left = right = nullptr;
            this->ch = ch_;
            this->node = false;
            this->freq = freq_;
        }
        huffnode(size_t freq_)
        {
            left = right = nullptr;
            this->node = true;
            this->freq = freq_;
        }
    };

    struct compare
    {
        bool operator()(nodePtr left, nodePtr right)
        {
            return(left->freq > right->freq );
        }
    };

    void makeBitCode(structNodePtr root, std::string str)
    {
        if (root == nullptr)
        {
            return;
        }
        if (root->node == false)
        {
            inoutput::bitcode[root->ch] = str;
        }

        makeBitCode(root->left, str + "0");
        makeBitCode(root->right, str + "1");
    }

    template <class T>
    void printQue(T t)
    {
        while(!t.empty())
        {
            auto tt = t.top();
            std::cout << std::bitset<8>(tt->ch) << " " << tt->freq<< "\n";
            t.pop();
```

```
        }
        std::cout << "\n";
    }

    void makeHuffTree()
    {
        structNodePtr left, right, top;
        std::priority_queue<nodePtr, std::vector<nodePtr>, compare> q;

        for(auto i = inoutput::freq.begin(); i != inoutput::freq.end(); ++i)
        {
            nodePtr tmp = std::make_shared<huffnode>(i->first, i->second);
            q.push(tmp);
        }
        //printQue(q);

        while (q.size() != 1)
        {
            left = q.top();
            q.pop();
            right = q.top();
            q.pop();
            top = std::make_shared<huffnode>(left->freq + right->freq);
            top->left = left;
            top->right = right;
            q.push(top);
        }
        makeBitCode(q.top(), "");
    }
}

#endif // ENCODE_H
```

4.4 exception.h

```
#ifndef EXCEPTION_H
#define EXCEPTION_H

#include <exception>

class Exception : public std::exception
{
public:
    Exception(const std::string& msg) : _msg(msg) {}
    Exception(const std::string& msg, const std::string& add) : _msg(msg + add)
        {}
    const char* what() const noexcept
    {
        return _msg.c_str();
    }
private:
    std::string _msg;
};

#endif // EXCEPTION_H
```

5 Вывод

В данной курсовой работе был рассмотрен способ сжатия файлов, путем кодирования информации, используя алгоритм Хаффмана. Алгоритм показал наибольшую эффективность при сжатии текстовых файлов 42%. При сжатии графических файлов наблюдается невысокая эффективность $5 - 10\%$.