

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

КУРСОВАЯ РАБОТА

Сжатие данных с помощью кодов Хаффмана
по дисциплине: «Объектно-ориентированное программирование»

Выполнил студент:
гр. в3530904/00030

Дружинин Н.В.

Руководитель:
доцент

Круглов С.К.

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

Задание на курсовую работу

Сжатие данных с помощью кодов Хаффмана
по дисциплине: «Объектно-ориентированное программирование»

ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Студенту группы № в3530904/00030 Дружинину Н.В.

1. Тема работы:

Сжатие данных с 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	4
1.1	Коды Хаффмана	4
2	Блок-схема алгоритма	5
2.1	Кодирование файла	5
2.2	Декодирование файла	5
3	Описание программы	6
3.1	Ввод-вывод файла	6
3.2	Построение дерева Хаффмана	6
4	Листинг программы	8
4.1	main.cpp	8
4.2	input_output.h	9
4.3	encode_decode.h	13
4.4	exception.h	15
5	Вывод	16

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 иных 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 $|C|$ листьев, по одному для каждого символа из множества C , и ровно $|C| - 1$ внутренних узлов.

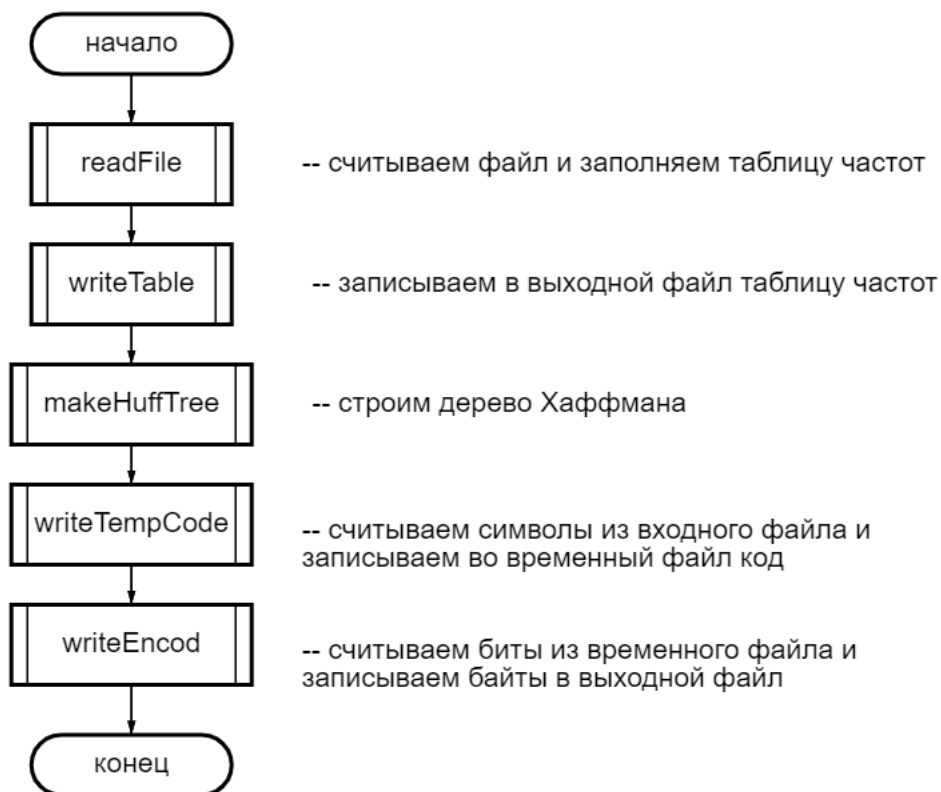
Если имеется дерево T , соответствующее префиксному коду, легко подсчитать количество битов, которые потребуются для кодирования файла:

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

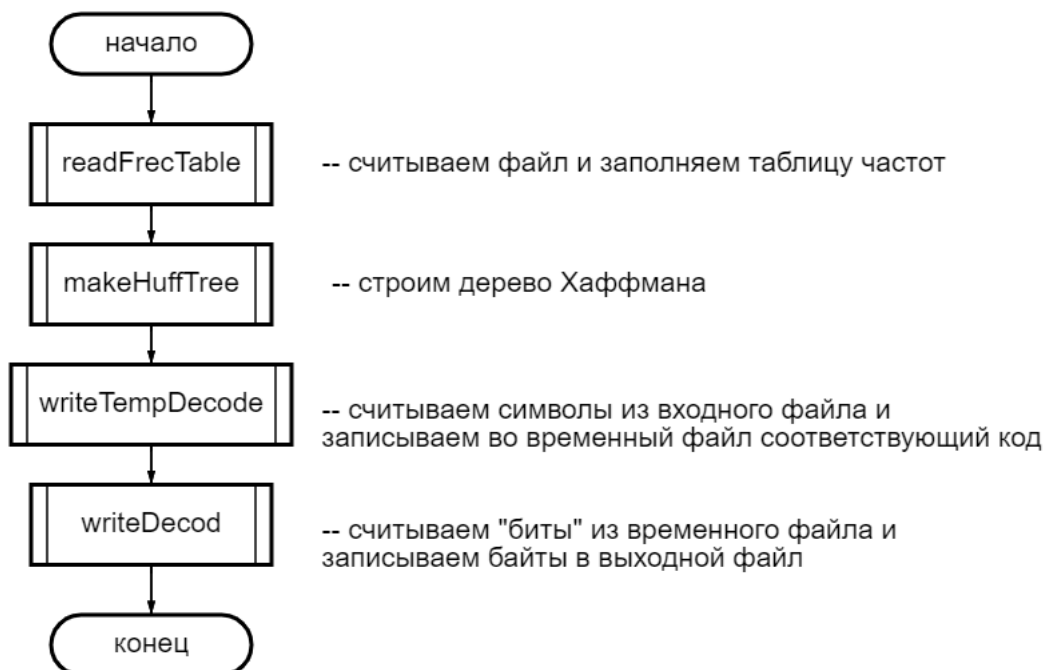
- $c.freq$ - частота появления символа c в файле
- $d_T(c)$ - глубина листа, представляющего символ c в дереве, а так же длина слова, кодирующего символ c

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:

```
1 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:

```
1 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:

```
1 union code
2 {
3     unsigned char chh;
4     struct byte
5     {
6         unsigned b1:1;
7         unsigned b2:1;
8         unsigned b3:1;
9         unsigned b4:1;
10        unsigned b5:1;
11        unsigned b6:1;
12        unsigned b7:1;
13        unsigned b8:1;
14    } byte;
15 };
```

3.2 Построение дерева Хаффмана

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

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

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

```
1 struct huffnode
2 {
3     unsigned char ch;
4     size_t freq;
5     bool node;
6     nodePtr left, right;
7     huffnode(unsigned char ch_, size_t freq_)
8     {
9         left = right = nullptr;
10        this->ch = ch_;
11        this->node = false;
12        this->freq = freq_;
13    }
14    huffnode(size_t freq_)
15    {
16        left = right = nullptr;
17        this->node = true;
18        this->freq = freq_;
19    }
20 };
```

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

```

1 struct compare
2 {
3     bool operator()(nodePtr left , nodePtr right)
4     {
5         return (left->freq > right->freq );
6     }
7 };

```

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

```

1 using structNodePtr = std::shared_ptr<struct huffnode>;
2 //...
3 while (q.size() != 1)
4 {
5     left = q.top();
6     q.pop();
7     right = q.top();
8     q.pop();
9     top = std::make_shared<huffnode>(left->freq + right->freq);
10    top->left = left;
11    top->right = right;
12    q.push(top);
13 }

```

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

```

1 //...
2 void makeBitCode(structNodePtr root , std::string str)
3 {
4     if (root == nullptr)
5     {
6         return;
7     }
8     if (root->node == false)
9     {
10        input::bitcode[root->ch] = str;
11    }
12
13    makeBitCode(root->left , str + "0");
14    makeBitCode(root->right , str + "1");
15 }
16 //...

```

4 Листинг программы

4.1 main.cpp

```
1 #include "input_output.h"
2 #include "encode_decode.h"
3
4 using namespace inoutput;
5 using namespace encode_decode;
6
7 int main()
8 {
9     try
10    {
11
12        char oper;
13        std::string inFile, outFile, tempFile{"temp.txt"}, suff;
14        std::cout << "To archivate press 1, to unzip press 2\n";
15        std::cin >> oper;
16        switch (oper)
17        {
18            case '1':
19                {
20                    //getfilename
21                    std::cout << "Please enter file name ";
22                    std::cin >> inFile;
23                    outFile = inFile;
24                    suff = "_encode";
25                    outFile.insert(outFile.begin() + outFile.find_last_of('.'), std
                        ::begin(suff), std::end(suff));
26
27                    //open and read input file
28                    size_t filesize;
29                    readFile(inFile.c_str());
30
31                    //write frequencies table
32                    writeTable(outFile.c_str());
33                    printMap(freq);
34
35                    //make Huffman tree
36                    makeHuffTree();
37                    printMap(bitcode);
38
39                    //write bit codes in temporary file
40                    writeTempCode(inFile.c_str(), tempFile.c_str(), filesize); //
                        pif.txt
41
42                    //get encode file
43                    writeEncod(tempFile.c_str(), outFile.c_str(), filesize);
44                    break;
45                }
46            case '2':
47                {
48                    //getfilename
49                    std::cout << "Please enter file name ";
50                    std::cin >> inFile;
51                    outFile = inFile;
52                    suff = "_decode";
53                    outFile.insert(outFile.begin() + outFile.find_last_of('.'), std
                        ::begin(suff), std::end(suff));
54
55                    //decode
56                    //get frquencies table
57                    long int offset;
58                    readFrecTable(inFile.c_str(), offset);
59                    //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&
82     filesize)
83 {
84     FILE* outfp = fopen(outfile, "wb"); //ab
85
86     if (!outfp)
87     {
88         throw Exception("Can not open file ", outfile);
89     }
90
91     FILE* infp = fopen(infile, "rb");
92
93     if (!infp)
94     {
95         throw Exception("Can not open file ", infile);
96     }
97
98     int ch;
99     size_t counter = 0;
100     while ((ch = fgetc(infp)) != EOF)
101     {
102         std::string tmp = bitcode[ch];
103         for (size_t i = 0; i < tmp.size(); ++i)
104         {
105             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         input::freq[ch] = frequency;
174         ++counter;
175     }
176     offset = sizeof(char) * numOfUnicElems + sizeof(int) * numOfUnicElems +
177             sizeof(int);
178     fclose(fp);
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 offset");
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

```

1 #ifndef ENCODE_H
2 #define ENCODE_H
3
4 #include <memory>
5 #include <queue>
6 #include <vector>
7
8 namespace encode_decode
9 {
10     struct huffnode;
11     using nodePtr = std::shared_ptr<huffnode>;
12     using structNodePtr = std::shared_ptr<struct huffnode>;
13
14     struct huffnode
15     {
16         unsigned char ch;
17         size_t freq;
18         bool node;
19         nodePtr left, right;
20         huffnode(unsigned char ch_, size_t freq_)
21         {
22             left = right = nullptr;
23             this->ch = ch_;
24             this->node = false;
25             this->freq = freq_;
26         }
27         huffnode(size_t freq_)
28         {
29             left = right = nullptr;
30             this->node = true;
31             this->freq = freq_;
32         }
33     };
34
35     struct compare
36     {
37         bool operator()(nodePtr left, nodePtr right)
38         {
39             return (left->freq > right->freq);
40         }
41     };
42
43     void makeBitCode(structNodePtr root, std::string str)
44     {
45         if (root == nullptr)
46         {
47             return;
48         }
49         if (root->node == false)
50         {
51             inoutput::bitcode[root->ch] = str;
52         }
53
54         makeBitCode(root->left, str + "0");
55         makeBitCode(root->right, str + "1");
56     }
57
58     template <class T>
59     void printQue(T t)
60     {
61         while (!t.empty())
62         {
63             auto tt = t.top();
64             std::cout << std::bitset<8>(tt->ch) << " " << tt->freq << "\n";
65             t.pop();
66         }
67     }
68 }

```

```

66     }
67     std::cout << "\n";
68 }
69
70 void makeHuffTree()
71 {
72     structNodePtr left, right, top;
73     std::priority_queue<nodePtr, std::vector<nodePtr>, compare> q;
74
75     for(auto i = inoutput::freq.begin(); i != inoutput::freq.end(); ++i)
76     {
77         nodePtr tmp = std::make_shared<huffnode>(i->first, i->second);
78         q.push(tmp);
79     }
80     //printQue(q);
81
82     while (q.size() != 1)
83     {
84         left = q.top();
85         q.pop();
86         right = q.top();
87         q.pop();
88         top = std::make_shared<huffnode>(left->freq + right->freq);
89         top->left = left;
90         top->right = right;
91         q.push(top);
92     }
93     makeBitCode(q.top(), "");
94 }
95 }
96
97 #endif // ENCODE_H

```

4.4 exception.h

```

1  #ifndef EXCEPTION_H
2  #define EXCEPTION_H
3
4  #include <exception>
5
6  class Exception : public std::exception
7  {
8  public:
9      Exception(const std::string& msg) : _msg(msg) {}
10     Exception(const std::string& msg, const std::string& add) : _msg(msg + add)
11     {}
12     const char* what() const noexcept
13     {
14         return _msg.c_str();
15     }
16 private:
17     std::string _msg;
18 };
19 #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%.