МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
КАФЕДРА САПР

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Алгоритмы кодирования

Студентка гр. 9301 _____ Синицкая В. А.

Преподаватель _____ Тутуева А. В.

Санкт-Петербург
2021

Оглавление

1 Постановка задачи

20 вариант => 2 вариант:

1. Реализовать кодирование и декодирование по алгоритму Шеннона-Фано входной строки, вводимой через консоль.
2. Посчитать объем памяти, который занимает исходная и закодированная строки
3. Выводить на экран таблицу частот и кодов, результат кодирования и декодирования, коэффициент сжатия
4. Стандартные структуры данных C++ использовать нельзя. Необходимо использовать структуры данных из предыдущих лабораторных работ

2 Реализуемые классы и методы

В проекте содержатся классы Element, L1List, содержится заголовочный файл Shannon-Fano_Coding.h, в котором написаны функции кодирования и декодирования, а также функция console_program, с помощью которой осуществляется взаимодействие с пользователем и исполнение кодирования и декодирования данной строки, а так же вывод требуемой информации.

Класс Element содержит поле next (указатель на следующий элемент), поле symbol, хранящее буквенное значение, поле code, хранящее пустую строку или код символа после произведенного кодирования, поле appearance, которое говорит о том, сколько раз в тексте встретилась данная буква.

L1List был модифицирован и дополнен функциями вывода, поиска элемента по заданному символу, функцией дописывания кода в конец кода элемента по данному индексу этого элемента, функцией сортировки слиянием по полю appearance от большего к меньшему.

Файл Shannon-Fano_Coding.h содержит функции text_processing_from_file, shannon_fano_code_from_list, shannon_fano_code, code_from_file, decode_from_file, console_program.

text_processing_from_file открывает файл по заданному имени и считывает текст, анализируя его. В результаты возвращает L1List, в котором отсортированы все буквы по количеству их встречаемости в данном тексте.

shannon_fano_code_from_list имея лист элементов, начальный индекс и количество элементов, которые нужно брать начиная с этого индекса, изменяет их коды, находя их по алгоритму Шеннона-Фано. Так как лист отсортирован, то элементы распределяются по частям довольно просто, если сумма левых меньше или равна сумме правых, новый элемент идет в левую часть, если наоборот, то в правую. Вместо элементов хранятся их индексы и операции разделения и рекурсивного вызова производятся с индексами. На каждом шаге создается массив беззнаковых целых чисел размером в количество сортируемых элементов, левая часть для рекурсивного вызова это левая часть нового массива, туда элементы добавляются по порядку, в правую часть они впихиваются с конца, проталкивая элементы на один влево. Рекурсивный вызов происходит от левой и правой части такого массива.

shannon_fano_code принимает имя файла и возвращает список элементов, с помощью которого можно кодировать и декодировать текст, использует для этого функции text_processing_from_file и shannon_fano_code_from_list.

code_from_file возвращает строку, закодированную по найденному листу. Находя каждый элемент в списке, прибавляет его код к итоговой строке. Операция будет работать оптимальнее из-за того, что лист отсортирован по встречаемости символов.

decode_from_file открывает заданный файл, в котором написан закодированный текст, имея лист элементов, раскодирует и возвращает раскодированную строку. Сначала по алгоритму находится минимальная длин кода, потом из закодированного текста считывается такое количество символов, если нашелся такой элемент, то переходим к поиску следующего, если нашелся такой, который начинается также, но не этот, из-за префиксности кода нет необходимости начинать проход опять с начала списка, алгоритм идет дальше. Если код не был найден, генерируется исключение.

console_program считывает ввод с консоли, записывает его в файл и делает все необходимые действия и обработки, выводя пользователю в консоль кодированный и декодированный текст, символы, их коды и их встречаемость, исходный размер текста в битах и закодированный, коэффициент сжатия данных, который получился в этом примере.

3 Оценка временной сложности каждого метода

1) L1List Element* search_on_symbol(char) — O(N).
2) L1List void write_end_of_code(unsigned int, string) — O(N) — как метод at.
3) L1List void merge_sort_by_appearance(unsigned int, unsigned int) — O(NlogN) — logN рекурсивных вызовов, по N операций в каждом.
4) text_processing_from_file — O(N^2) — проход по всем символам текста и поиск символа в листе.
5) shannon_fano_code_from_list — O(NlogN) — logN рекурсивных вызовов и N действий для подсчета суммы встречаемости и дописывания кода.
6) shannon_fano_code — O(N^2) — как сложности 4 и 5 пунктов.
7) decode_from_file — O(N^2) — в среднем. Вообще говоря, сильно зависит от длины строки, встречаемости каждой буквы, зависит от минимальной длины кода, не будет хуже, чем длина строки делить на минимальную длину кода и умножить это на N, где N — это количество различных символов в данном коде. Можно пойти дальше и подумать о длине кода, но не факт, что длина минимального кода не меньше, чем logN, или что не больше, чем log(N+1). Если один символ встречается ну очень часто по сравнению с другими, то он может быть закодирован кодом длины 1, а остальные — более длинными кодами.
8) console_program — O(N^2) — как худшая сложность из содержащихся функций.

4 Описание реализованных Unit test

Была проверена работа функции обработки текста, которая включала в себя проверку сортировки списка, работа функций кодирования и декодирования, были также сделаны проверки работы с консолью и выполнение console_program путем переключения работы с юнит-тестов на приложение.

5 Пример работы программы

Примером работы послужит выполнение функции console_program для трех цитат:

1) Life is like a box of chocolate. You never know what you’re gonna get. (Рис. 5.1)

2) Be yourself; everyone else is already taken. (Рис. 5.2)

3) To be, or not to be, that is the question (Рис. 5.3).

```
write some text to code:
Life is like a box of chocolate. You never know what you’re gonna get.

coded text:
0101011001110001001111110011001011111010110011010010011111101111111001001111
0110111011110001111100000000101111000001111010110111010110011110111111011000
1110100011100010011001000011101001110100100010111000001110000000001101110100
1111111010011101000010100101000011111110001110001000110111111110000110101111
1011
decoded text:
Life is like a box of chocolate. You never know what you're gonna get.
table of frequences and codes:
symbol =  , frequency = 13, code = 111
symbol = o, frequency = 8, code = 0111
symbol = e, frequency = 7, code = 0011
symbol = a, frequency = 4, code = 10111
symbol = n, frequency = 4, code = 0001
symbol = i, frequency = 3, code = 10011
symbol = t, frequency = 3, code = 01011
symbol = f, frequency = 2, code = 10001
symbol = l, frequency = 2, code = 10101
symbol = k, frequency = 2, code = 01001
symbol = c, frequency = 2, code = 10000
symbol = h, frequency = 2, code = 00001
symbol = ., frequency = 2, code = 11011
symbol = u, frequency = 2, code = 01000
symbol = r, frequency = 2, code = 10100
symbol = w, frequency = 2, code = 00000
symbol = g, frequency = 2, code = 1100
symbol = L, frequency = 1, code = 010101
symbol = s, frequency = 1, code = 00101
symbol = b, frequency = 1, code = 10010
symbol = x, frequency = 1, code = 0110
symbol = Y, frequency = 1, code = 10110
symbol = v, frequency = 1, code = 00100
symbol = y, frequency = 1, code = 11010
symbol = ', frequency = 1, code = 010100
original memory size = 560 bit
zipped memory size = 304 bit
compression ratio = 0.542857
```

Рис. 5.1 — работа программы при вводе цитаты 1.

```
write some text to code:
Be yourself; everyone else is already taken.

coded text:
10001111011001110101000011011000111110011101010010111111000
00111101100111010100101111011111100100011110110000000010110
10110011011111010111000011011010000101101001110010100100
decoded text:
Be yourself; everyone else is already taken.
table of frequences and codes:
symbol = e, frequency = 9, code = 111
symbol =  , frequency = 6, code = 011
symbol = y, frequency = 3, code = 0011
symbol = r, frequency = 3, code = 1011
symbol = s, frequency = 3, code = 0001
symbol = l, frequency = 3, code = 1001
symbol = a, frequency = 3, code = 0101
symbol = o, frequency = 2, code = 10101
symbol = n, frequency = 2, code = 00101
symbol = B, frequency = 1, code = 10001
symbol = u, frequency = 1, code = 00001
symbol = f, frequency = 1, code = 1101
symbol = ;, frequency = 1, code = 01001
symbol = v, frequency = 1, code = 10000
symbol = i, frequency = 1, code = 00000
symbol = d, frequency = 1, code = 1100
symbol = t, frequency = 1, code = 01000
symbol = k, frequency = 1, code = 10100
symbol = ., frequency = 1, code = 00100
original memory size = 352 bit
zipped memory size = 173 bit
compression ratio = 0.491477
```

Рис. 5.2 — работа программы при вводе цитаты 2.

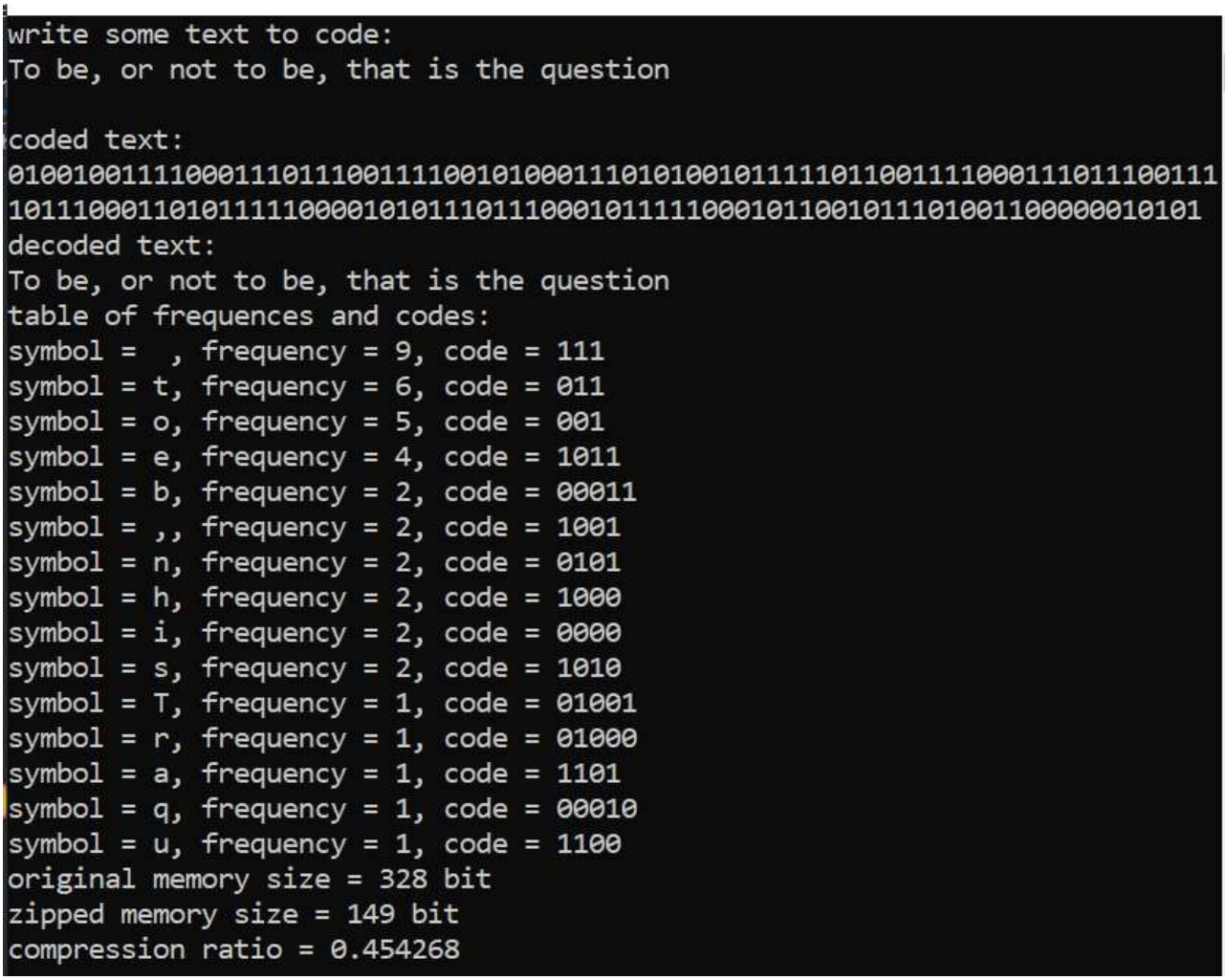

Рис. 5.3 — работы программы при вводе цитаты 3.

6 Листинг

6.1 Element.h

```
1  #include <iostream>
2  #include <stdexcept>
3  using namespace std;
4  class Element
5  {
6    Element* next;
7    string code = ""; // Shannon-Fano code of the symbol
8    char symbol;
9    unsigned int appearance_times = 0; // how many times current sybol was in the text
10   void set_next(Element*);
11   void set_symbol(char);
12   void set_appearance(unsigned int);
13   void set_code(string);
14   Element(); // to forbid creating with uncknown symbol
15 public:
16   Element(char);
17   Element(char, unsigned int);
18   void plus_appearance(unsigned int); // pluses to appearence_times data number
19   unsigned int get_appearance();
20   string get_code();
```

```
21   char get_symbol();
22   friend class L1List;
23   Element* get_next();
24   ~Element();
25 };
```

6.2 L1List.h

```
1  #pragma once
2  #include "Element.h"
3  #include <fstream>
4  class L1List
5  {
6    Element* head = nullptr;
7    Element* tail = nullptr;
8    void set_head(Element*);
9    void set_tail(Element*);
10   void set_next(Element*, Element*); // change next regarding to cur
11   //static void set_symbol(Element*, char); // chage cur data
12 public:
13   L1List();
14   Element* get_head();
15   Element* get_tail();
16   char get_symbol(Element*); // get cur data
17   Element* get_next(Element*); // get next regarding to cur
18   bool isEmpty(); // check for empty list
19   void push_back(Element*); // adding to the end of the list
20   void push_front(Element*); // adding to the begining of the list
21   void pop_back(); // delete last element
22   void pop_front(); // delete first element
23   unsigned int get_size(); // get size of the list
24   void insert(Element*, unsigned int); // adding element on index (before the element, th
   at had this index lately)
25   Element* at(unsigned int); // return element on index
26   void remove(unsigned int); // delete element on index
27   void print_to_console(); // print elements to console without using at()
28   void print_to_file(string filename = "out.txt"); // print all to file
29   void clear(); // delete all elements of the list
30   void set(unsigned int, char); // change data of element on index
31   void push_front(L1List*); // insertion another list into begining of the data-list
32   Element* search_on_symbol(char); // finds element with data symbol
33   void merge_sort_by_appearance(unsigned int, unsigned int); // sorts list by appearence
   from bigger to smaller
34   void write_end_of_code(unsigned int, string); // write data string to the end of the co
   de of the element with data index
35   ~L1List();
36
37 };
```

6.3 Shannon-Fano_Coding.h

```
1  #pragma once
2  #include "L1List.h"
3
4  L1List* text_processing_from_file(string); // reading from file, parameter = filename
5  void shannon_fano_code_from_list(unsigned int*, unsigned int, L1List*);
6  L1List* shannon_fano_code(string); // when read from file
7  string code_from_file(L1List*, string);
8  string decode_from_file(L1List*, string);
9  void console_program(); // user-session
```

6.4 Element.cpp

```
#pragma once
#include "Element.h"
Element::Element()
{
  symbol = ' ';
  code = "";
  appearance_times = 0;
  next = nullptr;
}
Element::Element(char new_symbol)
{
  symbol = new_symbol;
  code = "";
  appearance_times = 0;
  next = nullptr;
}
Element::Element(char new_symbol, unsigned int times)
{
  symbol = new_symbol;
  code = "";
  appearance_times = times;
  next = nullptr;
}
Element* Element::get_next() { return next; }
void Element::set_next(Element* new_element) { next = new_element; }
void Element::set_symbol(char new_symbol) { symbol = new_symbol; }
void Element::set_code(string new_code) { code = new_code; }
void Element::set_appearance(unsigned int new_appearance) { appearance_times = new_appea
rance; }
void Element::plus_appearance(unsigned int times)
{ appearance_times += times; }
unsigned int Element::get_appearance() { return appearance_times; }
string Element::get_code() { return code; }
char Element::get_symbol() { return symbol; }
Element::~Element() { }
```

6.5 L1List.cpp

```
#pragma once
#include "L1List.h"
L1List::L1List()
{
  head = nullptr;
  tail = nullptr;
}
Element* L1List::get_head() { return head; }
Element* L1List::get_tail() { return tail; }
void L1List::set_head(Element* head_element) { head = head_element; }
void L1List::set_tail(Element* tail_element) { tail = tail_element; }
void L1List::set_next(Element* now_element, Element* next_element) { now_element-
>set_next(next_element); }
//static void L1List::set_symbol(Element* now_element, char new_symbol) { now_element-
>symbol = new_symbol; }
Element* L1List::get_next(Element* now_element) { return (now_element->get_next()); }
char L1List::get_symbol(Element* now_element) { return (now_element->symbol); }

void L1List::write_end_of_code(unsigned int index, string code_end)
{
  if (index >= get_size())
```

```
20            throw new invalid_argument("Invalid index");
21    at(index)->code += code_end;
22  }
23
24  bool L1List::isEmpty()
25  {
26    if (head != nullptr)
27            return false; // list is not empty
28    return true;
29  }
30
31  void L1List::push_back(Element* new_element)
32  {
33    if (isEmpty())
34            head = tail = new_element;
35    else
36    {
37            tail->set_next(new_element);
38            tail = tail->get_next();
39    }
40  }
41
42  void L1List::push_front(Element* new_element)
43  {
44    if (isEmpty())
45            head = tail = new_element;
46    else
47    {
48            Element* current = new_element; // cur = new elem
49            current->set_next(head); // cur->next = head
50            head = current; // head = cur
51    }
52  }
53
54  void L1List::pop_back()
55  {
56    Element* current = get_head();
57    if (!isEmpty())
58    {
59            if (get_next(current) == nullptr) // delete head
60            {
61                    Element* element_to_delete = head;
62                    current = head = tail = nullptr;
63                    delete element_to_delete;
64            }
65            else
66            {
67                    while (get_next(current)->get_next() != nullptr) // while cur->next-
    >next exists
68                    {
69                            current = get_next(current); // cur = cur->next
70                    } // cur = the element before the last existing element
71                    Element* element_to_delete = get_next(current);
72                    current->set_next(nullptr);
73                    tail = current;
74                    delete element_to_delete;
75            }
76    }
77    else
78            throw out_of_range("The List is empty");
79  }
```

```
80
81 void L1List::pop_front()
82 {
83   Element* current = get_head();
84   if (!isEmpty())
85   {
86           if (get_next(current) == nullptr) // delete head
87           {
88                   Element* element_to_delete = head;
89                   head = current = tail = nullptr;
90                   delete element_to_delete;
91           }
92           else
93           {
94                   Element* element_to_delete = head;
95                   current = head->get_next();// cur = head->next
96                   set_head(current); // head = cur
97                   delete element_to_delete;
98           }
99   }
100  else
101          throw out_of_range("The List is empty");
102}
103
104unsigned int L1List::get_size()
105{
106  unsigned int List_size = 0;
107  Element* current = get_head();
108  if (!isEmpty())
109  {
110          List_size = 1;
111          while (get_next(current) != nullptr) // while cur->next exists
112          {
113                  current = get_next(current); // cur = cur->next
114                  List_size++;
115          } // cur = last existing element
116  }
117  return List_size;
118}
119
120void L1List::insert(Element* new_element, unsigned int index) // first index = 0
121{
122  if (index == get_size())
123          push_back(new_element);
124  else if (index == 0)
125          push_front(new_element);
126  else if (index > get_size())
127          throw out_of_range("Invalid index");
128  else
129  {
130          Element* current = get_head(); // now = head
131          while (index > 1)
132          {
133                  index--;
134                  current = get_next(current); // cur = cur->next
135          } // cur is the element before the future new element
136          new_element->set_next(get_next(current)); //e->next = cur->next
137          set_next(current, new_element); // cur->next = e
138  }
139}
140
```

```
141 Element* L1List::at(unsigned int index) // first index = 0
142 {
143   if (index >= get_size())
144           throw out_of_range("Invalid index");
145   else if (index == 0)
146           return head;
147   else
148   {
149           Element* current = head;
150           while (index > 0)
151           {
152                   index--;
153                   current = get_next(current); // cur = cur->next
154           } // cur is the element with data index
155           return current;
156   }
157 }
158
159 void L1List::remove(unsigned int index)
160 {
161   if (index >= get_size())
162           throw out_of_range("Invalid index");
163   else if (index == 0) // delete head
164           pop_front();
165   else if (index == get_size() - 1)
166           pop_back();
167   else
168   {
169           Element* current = head;
170           while (index > 1)
171           {
172                   index--;
173                   current = get_next(current); // cur = cur->next
174           } // cur is the element before the deleting element
175           Element* element_to_delete = new Element;
176           element_to_delete = get_next(current);
177           set_next(current, element_to_delete->get_next()); //cur->next = cur->next-
    >next
178           if (element_to_delete == nullptr)
179                   tail = current;
180           delete element_to_delete;
181   }
182 }
183
184 void L1List::print_to_console()
185 {
186   if (isEmpty())
187           throw new invalid_argument("can't print empty list");
188   Element* current = get_head(); // now = head
189   while (current != nullptr)
190   {
191
      cout << "symbol = " << get_symbol(current) << ", appearence_times = " << current-
    >get_appearance() << ", code = " << current->get_code() << endl;
192           current = get_next(current); // cur = cur->next
193   }
194 }
195
196 void L1List::print_to_file(string filename)
197 {
198   ofstream out(filename);
```

```
199  if (isEmpty())
200          throw new invalid_argument("can't print empty list");
201  Element* current = get_head(); // now = head
202  while (current != nullptr)
203  {
204
     out << "symbol = " << get_symbol(current) << ", appearence_times = " << current-
   >get_appearance() << ", code = " << current->get_code() << endl;
205          current = get_next(current); // cur = cur->next
206  }
207}
208
209void L1List::clear()
210{
211  if (!isEmpty())
212  {
213          Element* current = get_head(); // cur = head
214          while (get_next(current) != nullptr) // while next exists
215          {
216                  current = get_next(current); // cur = cur->next
217                  delete get_head();
218                  set_head(current); // head = cur
219
220          } //cur - the last in the list
221          Element* element_to_delete = get_head();
222          head = current = tail = nullptr;
223          delete element_to_delete;
224  }
225  else
226          throw exception("The List is empty");
227}
228
229void L1List::set(unsigned int index, char new_symbol) // change data on index element
230{
231  if (index >= get_size())
232          throw out_of_range("Invalid index");
233  else if (index == 0)
234          head->symbol = new_symbol;
235  else if (index == get_size() - 1)
236          tail->symbol = new_symbol;
237  else
238  {
239          Element* current = get_head(); // now = head
240          while (index > 0)
241          {
242                  index--;
243                  current = get_next(current); // cur = cur->next
244          } // cur is the element with data index
245          current->symbol = new_symbol;
246  }
247}
248
249void L1List::push_front(L1List* L) // insertion another list into the begining of data
250{
251  if (!L->isEmpty())
252  {
253          push_front(L->get_head()); // first in this is the same as the first in L now
254
     Element* head_element = head; // now head is a new element because that's how my push_f
   ront works
255          Element* L_cur = L->get_head();
```

```
256           while (L->get_next(L_cur) != nullptr) // while cur->next exists
257           {
258                   L_cur = L->get_next(L_cur);
259                   insert(L_cur, 1);
260                   head = head-
   >get_next(); // head is an inserted element - this is made for next iteration
261           }
262           set_head(head_element);
263   }
264}
265
266Element* L1List::search_on_symbol(char s_symbol)
267{ // SPECIALLY returns nullptr when searched for not-existing symbol
268   if (isEmpty())
269           throw new out_of_range("search in empty list");
270   Element* current = get_head();
271   while (current != nullptr)
272   {
273           if (current->symbol == s_symbol)
274                   return current;
275           current = current->get_next();
276   }
277   return current;
278}
279
280void L1List::merge_sort_by_appearance(unsigned int start_index, unsigned int end_index)
281{
282   if (start_index == end_index)
283           return;
284   unsigned int middle_index = (start_index + end_index) / 2;
285   merge_sort_by_appearance(start_index, middle_index);
286   merge_sort_by_appearance(middle_index + 1, end_index);
287   unsigned int left_index = start_index;
288   unsigned int right_index = middle_index + 1;
289   L1List* merge_list = new L1List();
290   while ((left_index < middle_index + 1) && (right_index < end_index + 1))
291   {
292           if (at(left_index)->get_appearance() >= at(right_index)->get_appearance())
293           {
294                   Element* new_element = new Element(this->at(left_index)-
   >get_symbol(), this->at(left_index)->get_appearance());
295                   merge_list->push_back(new_element);
296                   left_index++;
297           }
298           else
299           {
300                   Element* new_element = new Element(this->at(right_index)-
   >get_symbol(), this->at(right_index)->get_appearance());
301                   merge_list->push_back(new_element);
302                   right_index++;
303           }
304   }
305   while (left_index < middle_index + 1)
306   {
307           Element* new_element = new Element(this->at(left_index)->get_symbol(), this-
   >at(left_index)->get_appearance());
308           merge_list->push_back(new_element);
309           left_index++;
310   }
311   while (right_index < end_index + 1)
312   {
```

```
313             Element* new_element = new Element(this->at(right_index)->get_symbol(), this-
    >at(right_index)->get_appearance());
314             merge_list->push_back(new_element);
315             right_index++;
316     }
317     Element* current = at(start_index); // for going through elements of the original list
318     Element* current_merged = merge_list->at(0);
319     for (int i = 0; i <= end_index - start_index; i++)
320     {
321             current->set_symbol(current_merged->get_symbol());
322             current->set_appearance(current_merged->get_appearance());
323             //current->set_code(current_merged->get_code());
324             current = current->get_next();
325             current_merged = current_merged->get_next();
326     }
327 }
328
329 L1List::~L1List()
330 {
331     if (!isEmpty())
332             clear();
333 }
```

6.6 Shannon-Fano_Coding.cpp

```
1   // Shannon-Fano_Coding.cpp : Определяет функции для статической библиотеки.
2   //
3   #pragma once
4   #include "framework.h"
5   #include "Shannon-Fano_Coding.h"
6
7   L1List* text_processing_from_file(string filename)
8   {
9     ifstream in(filename);
10    if (!in.is_open())
11            throw new invalid_argument("the data file doesn't exist");
12    L1List* list = new L1List();
13    // не заходим в while wtf
14    while (in.peek() != EOF)
15    {
16            char now_char = in.get();
17            if (list->isEmpty())
18                    list->push_back(new Element(now_char, 1));
19            else
20            {
21                    Element* current = list->search_on_symbol(now_char);
22                    if (current != nullptr)
23                            current->plus_appearance(1);
24                    else
25                            list->push_back(new Element(now_char, 1));
26            }
27    }
28    in.close();
29    // now we want to sort from biggest to smaller by appearence
30    list->merge_sort_by_appearance(0, list->get_size() - 1);
31    return list;
32  }
33
34  void shannon_fano_code_from_list(unsigned int* index_array, unsigned int size, L1List* l
    ist)
```

```
35 {
36   if (size == 1)
37           return;
38   unsigned int left_sum = 0, right_sum = 0, count_left = 0, count_right = 0; // elements
   in tree that goes to right are marked by 0, to left - by 1
39   unsigned int* splited = new unsigned int[size];
40   for (unsigned int i = 0; i < size; i++)
41   {
42           if (left_sum <= right_sum) // adding element to the left
43           { // adding in splited in straight order
44                   splited[i - count_right] = index_array[i];
45                   count_left++;
46                   left_sum += list->at(index_array[i])->get_appearance();
47                   list->write_end_of_code(index_array[i], "1");
48           }
49           else // adding element to the right
50           {// adding in splited in reverse order to save this part sorted
51
     for (unsigned int j = size - 1 - count_right; j < size - 1; j++) // shifting right elem
   ents on one to the left
52                           splited[j] = splited[j + 1];
53                   splited[size - 1] = index_array[i]; // adding element like last
54                   right_sum += list->at(index_array[i])->get_appearance();
55                   list->write_end_of_code(index_array[i], "0");
56                   count_right++;
57           }
58   }
59   // now we have an array where in the left part indexses of elements of the left branch
60   // we going to add next number to the codes by recursive calls
61   shannon_fano_code_from_list(splited, count_left, list);
62   shannon_fano_code_from_list(&splited[count_left], size - count_left, list);
63 }
64
65 L1List* shannon_fano_code(string filename) // when read from file
66 {
67   L1List* list = text_processing_from_file(filename);
68   if (list->get_size() == 1)
69   {
70           list->write_end_of_code(0, "1");
71           return list;
72   }
73   unsigned int* index_array = new unsigned int[list->get_size()];
74   for (unsigned int i = 0; i < list->get_size(); i++)
75           index_array[i] = i;
76   shannon_fano_code_from_list(&index_array[0], list->get_size(), list);
77   return list;
78 }
79
80 string decode_from_file(L1List* list, string filename) // list with chars and their code
   s, name of the file
81 { // file "filename" contains code to decode
82   ifstream in(filename);
83   if (!in.is_open())
84           throw new invalid_argument("read from not-existing file");
85   string decoded_text = "";
86   // we are going to find min length of the code
87   Element* current = list->get_head();
88   unsigned int min_code_length = current->get_code().size();
89   while (current != nullptr)
90   {
91           if (current->get_code().size() < min_code_length)
```

```
92                      min_code_length = current->get_code().size();
93              current = current->get_next();
94      }
95      // processing input
96      while (in.peek() != EOF)
97      {
98              string supposed_code = ""; // get from console
99              for (unsigned i = 0; i < min_code_length; i++)
100                     supposed_code += char(in.get());
101             current = list->get_head();
102             while (current != nullptr)
103             {
104                     string part_of_current_code = current->get_code();
105                     if (supposed_code.size() < part_of_current_code.size())
106
    part_of_current_code = part_of_current_code.substr(0, supposed_code.size());
107
    while ((supposed_code == part_of_current_code) && (part_of_current_code != current-
  >get_code()))
108
    { // if we found similar part it is the searched code or supposed code should be longer
109                             supposed_code += char(in.get());
110                             part_of_current_code += current-
  >get_code()[part_of_current_code.size()];
111                     }
112                     if (supposed_code == current->get_code())
113                     {
114                             decoded_text += current->get_symbol();
115                             break;
116                     }
117                     current = current->get_next();
118             }
119             if (current == nullptr)
120                     throw new out_of_range("char by code not found");
121     }
122     return decoded_text;
123 }
124
125 string code_from_file(L1List* list, string filename)
126 { // file "filename" contains some text to code
127     ifstream in(filename);
128     if (!in.is_open())
129             throw new invalid_argument("read from not-existing file");
130     string coded_text = "";
131     while (in.peek() != EOF)
132     {
133             Element* now = list->search_on_symbol(in.get());
134             coded_text += now->get_code();
135     }
136     return coded_text;
137 }
138
139 void console_program() // user-session
140 {
141     string filename = "out.txt";
142     cout << "write some text to code:" << endl;
143     ofstream out(filename);
144     while (cin.peek() != '\n')
145     {
146             char now_char = cin.get();
147             out << now_char;
```

```
148  }
149  out.close();
150  L1List* list = shannon_fano_code(filename);
151  string coded_text = code_from_file(list, filename);
152  cout << endl << "coded text: " << endl;
153  cout << coded_text << endl;
154  out.open(filename);
155  out << coded_text;
156  out.close();
157  cout << "decoded text: \n" << decode_from_file(list, filename) << endl;
158  Element* current = list->get_head();
159  unsigned int original_memory = 0;
160  cout << "table of frequences and codes:" << endl;
161  while (current != nullptr)
162  {
163          original_memory += current->get_appearance();
164          cout << "symbol = " << current->get_symbol() << ", frequency = "
165                  << current->get_appearance() << ", code = " << current-
   >get_code() << endl;
166          current = current->get_next();
167  }
168  const int bites_in_byte = 8;
169  cout << "original memory size = " << original_memory*sizeof(char)* bites_in_byte << " b
   it" << endl;
170  cout << "zipped memory size = " << coded_text.size() << " bit" << endl;
171  cout << "compression ratio = " << (double)coded_text.size() / (original_memory * sizeof
   (char) * 8.0) << endl;
172}
173
174int main()
175{
176  console_program();
177  return 0;
178}
```

6.7 UnitTest_Shannon-Fano_Coding.cpp

```
1  #include "pch.h"
2  #include "CppUnitTest.h"
3  #include "..\\Shannon-Fano_Coding\Shannon-Fano_Coding.h"
4
5  using namespace Microsoft::VisualStudio::CppUnitTestFramework;
6
7  namespace UnitTestShannonFanoCoding
8  {
9    TEST_CLASS(UnitTestShannonFanoCoding)
10   {
11   public:
12           TEST_METHOD(Test_text_processing_one)
13           {
14                   ofstream out("in.txt");
15                   out << "s";
16                   out.close();
17                   L1List* list = text_processing_from_file("in.txt");
18                   list->print_to_file("out.txt");
19                   Assert::AreEqual(list->at(0)->get_symbol(), 's');
20           }
21           TEST_METHOD(Test_code_one)
22           {
23                   ofstream out("in.txt");
```

```
24                  out << "s";
25                  out.close();
26                  L1List* list = shannon_fano_code("in.txt");
27                  list->print_to_file("out.txt");
28                  Assert::AreEqual(list->at(0)->get_code(), string("1"));
29          }
30          TEST_METHOD(Test_text_processing_two)
31          {
32                  ofstream out("in.txt");
33                  out << "su";
34                  out.close();
35                  L1List* list = text_processing_from_file("in.txt");
36                  list->print_to_file("out.txt");
37                  Assert::AreEqual(list->at(0)->get_symbol(), 's');
38                  Assert::AreEqual(list->at(1)->get_symbol(), 'u');
39          }
40          TEST_METHOD(Test_code_two)
41          {
42                  ofstream out("in.txt");
43                  out << "su";
44                  out.close();
45                  L1List* list = shannon_fano_code("in.txt");
46                  list->print_to_file("out.txt");
47                  Assert::AreEqual(list->at(0)->get_code(), string("1"));
48                  Assert::AreEqual(list->at(1)->get_code(), string("0"));
49          }
50          TEST_METHOD(Test_text_processing_three)
51          {
52                  ofstream out("in.txt");
53                  out << "ssunnn";
54                  out.close();
55                  L1List* list = text_processing_from_file("in.txt");
56                  list->print_to_file("out.txt");
57                  Assert::AreEqual(list->at(0)->get_symbol(), 'n');
58                  Assert::AreEqual(list->at(1)->get_symbol(), 's');
59                  Assert::AreEqual(list->at(2)->get_symbol(), 'u');
60          }
61
62          TEST_METHOD(Test_code_three)
63          {
64                  ofstream out("in.txt");
65                  out << "ssunnn";
66                  out.close();
67                  L1List* list = shannon_fano_code("in.txt");
68                  list->print_to_file("out.txt");
69                  Assert::AreEqual(list->at(0)->get_code(), string("1"));
70                  Assert::AreEqual(list->at(1)->get_code(), string("01"));
71                  Assert::AreEqual(list->at(2)->get_code(), string("00"));
72          }
73          TEST_METHOD(Test_text_processing)
74          {
75                  ofstream out("in.txt");
76                  out << "aaabbbbbccdddd";
77                  out.close();
78                  L1List* list = text_processing_from_file("in.txt");
79                  list->print_to_file("out.txt");
80                  Assert::AreEqual(list->at(0)->get_symbol(), 'b');
81                  Assert::AreEqual(list->at(1)->get_symbol(), 'd');
82                  Assert::AreEqual(list->at(2)->get_symbol(), 'a');
83                  Assert::AreEqual(list->at(3)->get_symbol(), 'c');
84          }
```

```
85          TEST_METHOD(Test_code)
86          {
87                  ofstream out("in.txt");
88                  out << "aaabbbbbccdddd";
89                  out.close();
90                  L1List* list = shannon_fano_code("in.txt");
91                  list->print_to_file("out.txt");
92                  Assert::AreEqual(list->at(0)->get_code(), string("11"));
93                  Assert::AreEqual(list->at(1)->get_code(), string("01"));
94                  Assert::AreEqual(list->at(2)->get_code(), string("00"));
95                  Assert::AreEqual(list->at(3)->get_code(), string("10"));
96          }
97          TEST_METHOD(Test_text_processing_sentence)
98          {
99                  ofstream out("in.txt");
100                 out << "the sun will shine soon";
101                 out.close();
102                 L1List* list = text_processing_from_file("in.txt");
103                 list->print_to_file("out.txt");
104                 Assert::AreEqual(list->at(0)->get_symbol(), ' ');
105                 Assert::AreEqual(list->at(1)->get_symbol(), 's');
106                 Assert::AreEqual(list->at(2)->get_symbol(), 'n');
107                 Assert::AreEqual(list->at(3)->get_symbol(), 'h');
108                 Assert::AreEqual(list->at(4)->get_symbol(), 'e');
109                 Assert::AreEqual(list->at(5)->get_symbol(), 'i');
110                 Assert::AreEqual(list->at(6)->get_symbol(), 'l');
111                 Assert::AreEqual(list->at(7)->get_symbol(), 'o');
112                 Assert::AreEqual(list->at(8)->get_symbol(), 't');
113                 Assert::AreEqual(list->at(9)->get_symbol(), 'u');
114                 Assert::AreEqual(list->at(10)->get_symbol(), 'w');
115         }
116
117         TEST_METHOD(Test_code_sentence)
118         {
119                 ofstream out("in.txt");
120                 out << "the sun will shine soon";
121                 out.close();
122                 L1List* list = shannon_fano_code("in.txt");
123                 list->print_to_file("out.txt");
124                 Assert::AreEqual(list->at(0)->get_code(), string("111"));
125                 Assert::AreEqual(list->at(1)->get_code(), string("011"));
126                 Assert::AreEqual(list->at(2)->get_code(), string("001"));
127                 Assert::AreEqual(list->at(3)->get_code(), string("1011"));
128                 Assert::AreEqual(list->at(4)->get_code(), string("1001"));
129                 Assert::AreEqual(list->at(5)->get_code(), string("0101"));
130                 Assert::AreEqual(list->at(6)->get_code(), string("110"));
131                 Assert::AreEqual(list->at(7)->get_code(), string("000"));
132                 Assert::AreEqual(list->at(8)->get_code(), string("1010"));
133                 Assert::AreEqual(list->at(9)->get_code(), string("0100"));
134                 Assert::AreEqual(list->at(10)->get_code(), string("1000"));
135         }
136
137         TEST_METHOD(Test_code_test_string)
138         {
139                 ofstream out("in.txt");
140                 out << "it is test string";
141                 out.close();
142                 L1List* list = shannon_fano_code("in.txt");
143                 list->print_to_file("out.txt");
144                 Assert::AreEqual(list->at(0)->get_symbol(), 't');
145                 Assert::AreEqual(list->at(1)->get_symbol(), 'i');
```

```
146                     Assert::AreEqual(list->at(2)->get_symbol(), ' ');
147                     Assert::AreEqual(list->at(3)->get_symbol(), 's');
148                     Assert::AreEqual(list->at(4)->get_symbol(), 'e');
149                     Assert::AreEqual(list->at(5)->get_symbol(), 'r');
150                     Assert::AreEqual(list->at(6)->get_symbol(), 'n');
151                     Assert::AreEqual(list->at(7)->get_symbol(), 'g');
152                     Assert::AreEqual(list->at(0)->get_code(), string("111"));
153                     Assert::AreEqual(list->at(1)->get_code(), string("011"));
154                     Assert::AreEqual(list->at(2)->get_code(), string("001"));
155                     Assert::AreEqual(list->at(3)->get_code(), string("101"));
156                     Assert::AreEqual(list->at(4)->get_code(), string("010"));
157                     Assert::AreEqual(list->at(5)->get_code(), string("100"));
158                     Assert::AreEqual(list->at(6)->get_code(), string("000"));
159                     Assert::AreEqual(list->at(7)->get_code(), string("110"));
160             }
161             TEST_METHOD(Test_code_my)
162             {
163                     ofstream out("in.txt");
164
    out << "I'm trying to find a sentense that will break my thoughts about the length of t
  he code";
165                     out.close();
166                     L1List* list = shannon_fano_code("in.txt");
167                     list->print_to_file("out.txt");
168             }
169
170
171             TEST_METHOD(Test_decode_test_string)
172             {
173                     string filename = "out.txt";
174                     string input = "it is test string";
175                     ofstream out(filename);
176                     out << input;
177                     out.close();
178                     L1List* list = shannon_fano_code(filename);
179                     string coded_text = code_from_file(list, filename);
180                     out.open(filename);
181                     out << coded_text;
182                     out.close();
183                     string decoded_text = decode_from_file(list, filename);
184                     Assert::AreEqual(input, decoded_text);
185             }
186             TEST_METHOD(Test_decode_test_char)
187             {
188                     string filename = "out.txt";
189                     string input = "s";
190                     ofstream out(filename);
191                     out << input;
192                     out.close();
193                     L1List* list = shannon_fano_code(filename);
194                     string coded_text = code_from_file(list, filename);
195                     out.open(filename);
196                     out << coded_text;
197                     out.close();
198                     string decoded_text = decode_from_file(list, filename);
199                     Assert::AreEqual(input, decoded_text);
200             }
201             TEST_METHOD(Test_decode_test_two_letters)
202             {
203                     string filename = "out.txt";
204                     string input = "it is test string";
```

```
205                     ofstream out(filename);
206                     out << input;
207                     out.close();
208                     L1List* list = shannon_fano_code(filename);
209                     string coded_text = code_from_file(list, filename);
210                     out.open(filename);
211                     out << coded_text;
212                     out.close();
213                     string decoded_text = decode_from_file(list, filename);
214                     Assert::AreEqual(input, decoded_text);
215             }
216
217             TEST_METHOD(Test_decode_test_two_sentece)
218             {
219                     string filename = "out.txt";
220                     string input = "The sun will shine soon!";
221                     ofstream out(filename);
222                     out << input;
223                     out.close();
224                     L1List* list = shannon_fano_code(filename);
225                     string coded_text = code_from_file(list, filename);
226                     out.open(filename);
227                     out << coded_text;
228                     out.close();
229                     string decoded_text = decode_from_file(list, filename);
230                     Assert::AreEqual(input, decoded_text);
231             }
232     };
233 }
```