**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 3](#_Toc68535691)

[2 Реализуемые классы и методы 3](#_Toc68535692)

[3 Оценка временной сложности каждого метода 5](#_Toc68535693)

[4 Описание реализованных Unit test 5](#_Toc68535694)

[5 Пример работы программы 5](#_Toc68535695)

[6 Листинг 8](#_Toc68535696)

[6.1 Element.h 8](#_Toc68535697)

[6.2 L1List.h 9](#_Toc68535698)

[6.3 Shannon-Fano\_Coding.h 9](#_Toc68535699)

[6.4 Element.cpp 10](#_Toc68535700)

[6.5 L1List.cpp 10](#_Toc68535701)

[6.6 Shannon-Fano\_Coding.cpp 16](#_Toc68535702)

[6.7 UnitTest\_Shannon-Fano\_Coding.cpp 19](#_Toc68535703)

# Постановка задачи

20 вариант => 2 вариант:

Реализовать кодирование и декодирование по алгоритму Шеннона-Фано входной строки, вводимой через консоль.

Посчитать объем памяти, который занимает исходная и закодированная строки

Выводить на экран таблицу частот и кодов, результат кодирования и декодирования, коэффициент сжатия

Стандартные структуры данных C++ использовать нельзя. Необходимо использовать структуры данных из предыдущих лабораторных работ

# 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.

# 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.

# 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.

# 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).

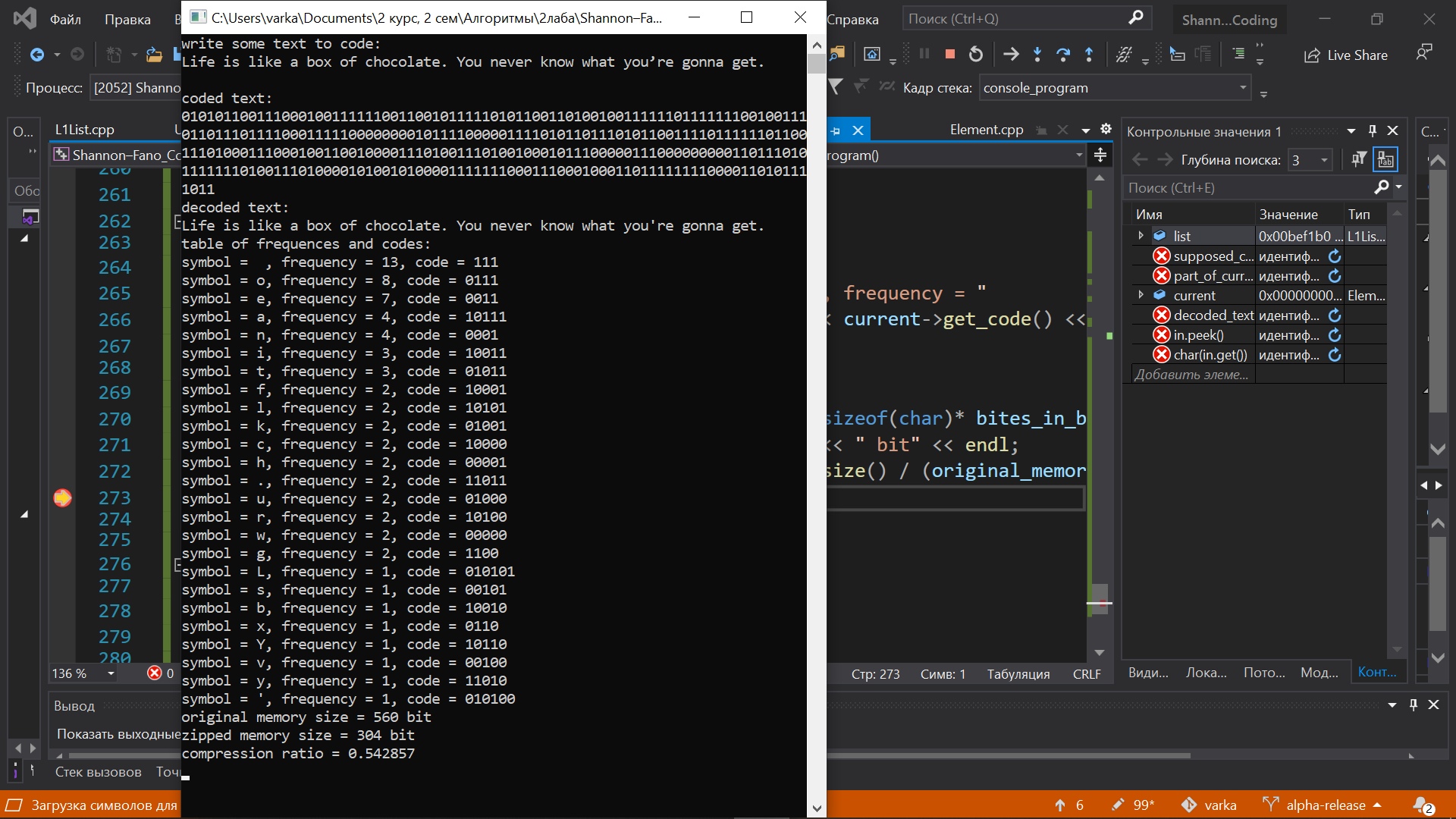


Рис. 5.1 — работа программы при вводе цитаты 1.

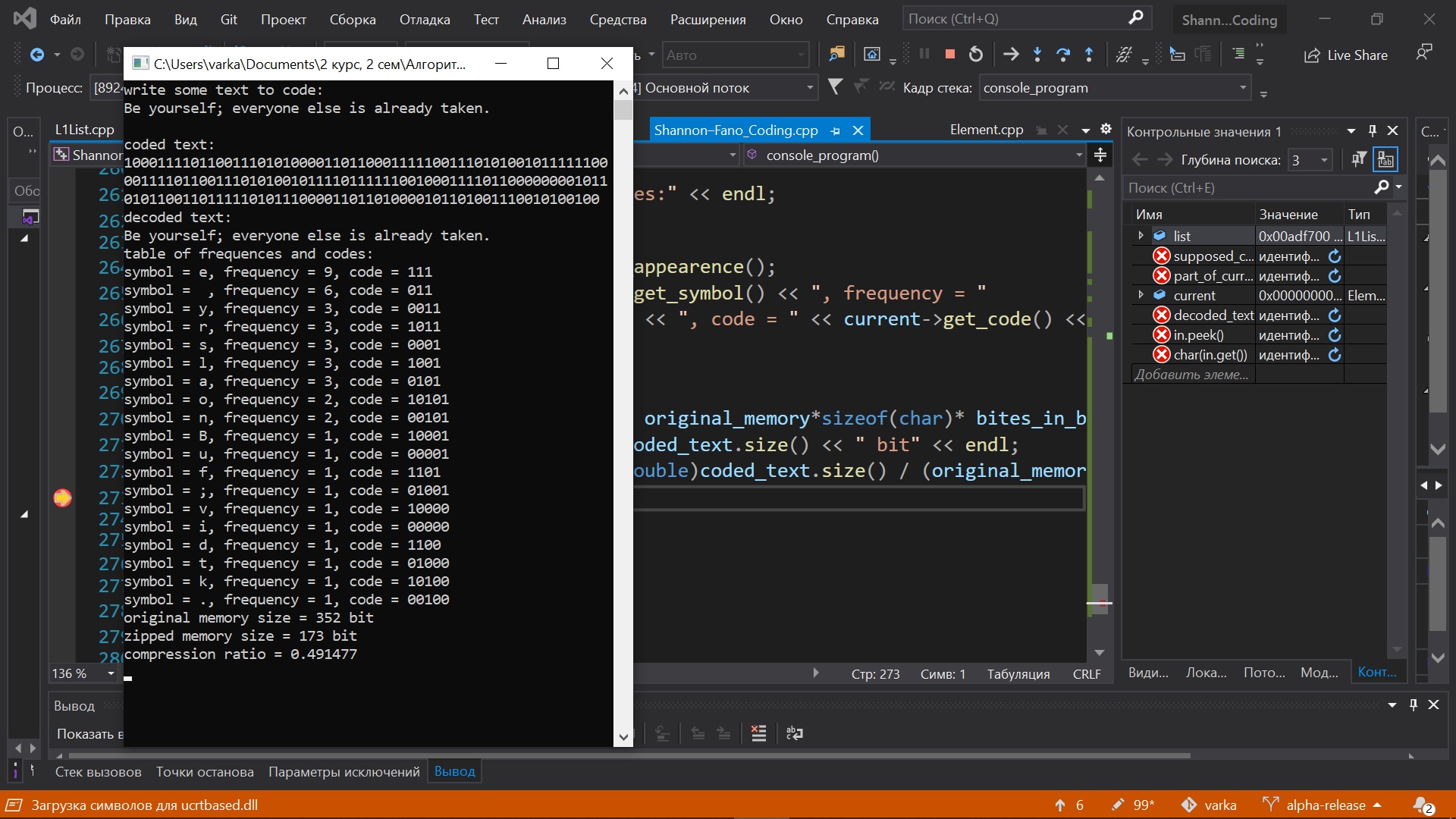


Рис. 5.2 — работа программы при вводе цитаты 2.

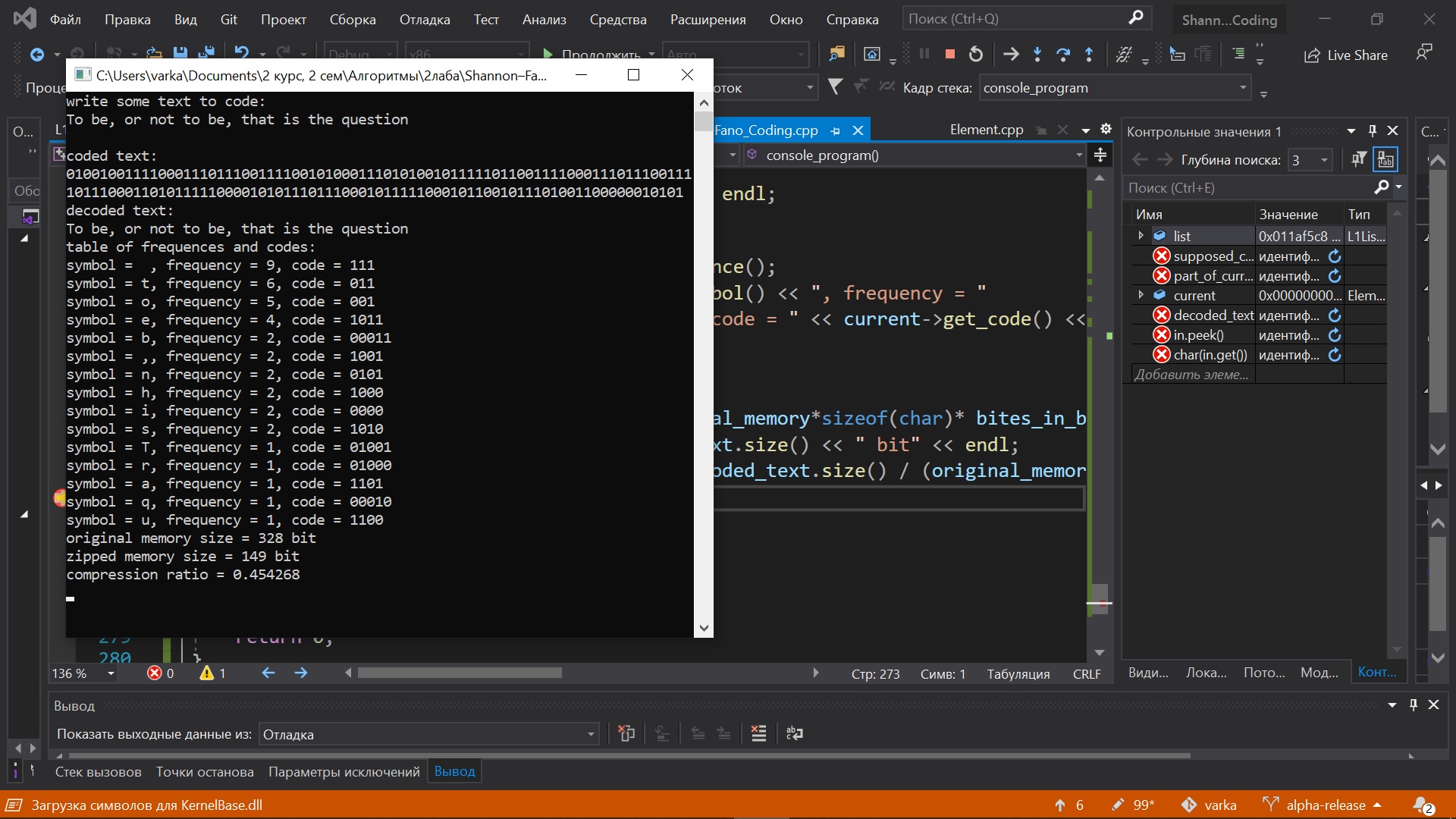


Рис. 5.3 — работы программы при вводе цитаты 3.

# Листинг

## 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. };

## 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, that 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 code of the element with data index
35. ~L1List();
37. };

## Shannon-Fano\_Coding.h

1. #pragma once
2. #include "L1List.h"
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

## Element.cpp

1. #pragma once
2. #include "Element.h"
3. Element::Element()
4. {
5. symbol = ' ';
6. code = "";
7. appearance\_times = 0;
8. next = nullptr;
9. }
10. Element::Element(char new\_symbol)
11. {
12. symbol = new\_symbol;
13. code = "";
14. appearance\_times = 0;
15. next = nullptr;
16. }
17. Element::Element(char new\_symbol, unsigned int times)
18. {
19. symbol = new\_symbol;
20. code = "";
21. appearance\_times = times;
22. next = nullptr;
23. }
24. Element\* Element::get\_next() { return next; }
25. void Element::set\_next(Element\* new\_element) { next = new\_element; }
26. void Element::set\_symbol(char new\_symbol) { symbol = new\_symbol; }
27. void Element::set\_code(string new\_code) { code = new\_code; }
28. void Element::set\_appearance(unsigned int new\_appearance) { appearance\_times = new\_appearance; }
29. void Element::plus\_appearance(unsigned int times)
30. { appearance\_times += times; }
31. unsigned int Element::get\_appearance() { return appearance\_times; }
32. string Element::get\_code() { return code; }
33. char Element::get\_symbol() { return symbol; }
34. Element::~Element() { }

## L1List.cpp

1. #pragma once
2. #include "L1List.h"
3. L1List::L1List()
4. {
5. head = nullptr;
6. tail = nullptr;
7. }
8. Element\* L1List::get\_head() { return head; }
9. Element\* L1List::get\_tail() { return tail; }
10. void L1List::set\_head(Element\* head\_element) { head = head\_element; }
11. void L1List::set\_tail(Element\* tail\_element) { tail = tail\_element; }
12. void L1List::set\_next(Element\* now\_element, Element\* next\_element) { now\_element->set\_next(next\_element); }
13. //static void L1List::set\_symbol(Element\* now\_element, char new\_symbol) { now\_element->symbol = new\_symbol; }
14. Element\* L1List::get\_next(Element\* now\_element) { return (now\_element->get\_next()); }
15. char L1List::get\_symbol(Element\* now\_element) { return (now\_element->symbol); }
17. void L1List::write\_end\_of\_code(unsigned int index, string code\_end)
18. {
19. if (index >= get\_size())
20. throw new invalid\_argument("Invalid index");
21. at(index)->code += code\_end;
22. }
24. bool L1List::isEmpty()
25. {
26. if (head != nullptr)
27. return false; // list is not empty
28. return true;
29. }
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. }
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. }
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. }
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. }
104. unsigned 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. }
120. void 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. }
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. }
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. }
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. }
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. }
209. void 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
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. }
229. void 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. }
249. void 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\_front 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. }
266. Element\* 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. }
280. void 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. }
329. L1List::~L1List()
330. {
331. if (!isEmpty())
332. clear();
333. }

## 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"
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. }
34. void shannon\_fano\_code\_from\_list(unsigned int\* index\_array, unsigned int size, L1List\* list)
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 elements 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. }
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. }
80. string decode\_from\_file(L1List\* list, string filename) // list with chars and their codes, 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. }
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. }
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 << " bit" << 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. }
174. int main()
175. {
176. console\_program();
177. return 0;
178. }

## UnitTest\_Shannon-Fano\_Coding.cpp

1. #include "pch.h"
2. #include "CppUnitTest.h"
3. #include "..\\Shannon–Fano\_Coding\Shannon-Fano\_Coding.h"
5. using namespace Microsoft::VisualStudio::CppUnitTestFramework;
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. }
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. }
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. }
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 the code";
165. out.close();
166. L1List\* list = shannon\_fano\_code("in.txt");
167. list->print\_to\_file("out.txt");
168. }

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. }
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. }