**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»**

**Вариант 2**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Сиротин П.О. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

Оглавление

[**1.** **Постановка задачи. Краткое описание реализуемого алгоритма и используемых структур данных. Оценка временной сложности каждого метода** 3](#_Toc74002841)

[2. Описание реализованных unit-тестов 6](#_Toc74002842)

[3. Пример работы. 6](#_Toc74002843)

[4. Листинг. 8](#_Toc74002844)

[5. Вывод 19](#_Toc74002845)

1. **Постановка задачи. Краткое описание реализуемого алгоритма и используемых структур данных. Оценка временной сложности каждого метода**

* 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

Некоторые классы с реализацией были взяты из прошлой лабораторной работы (например, класс stack с его вложенным классом st\_node и класс map). В класс map были внесены изменения:

* Из-за особенности алгоритма необходимо было добавить перегрузку таких операторов, как = и [];
* Из-за особенности алгоритма пришлось добавить пару строк в метод map::insert(), чтобы в дальнейшем можно было правильно закодировать сообщение;
* Был добавлен рекурсивный метод заполнения массива – это необходимо для создания массива символов и массива частот появления этих символов во входной строке, чтобы в дальнейшем можно было реализовать алгоритм Шеннона-Фано;

Кроме этого, был создан отдельный файл, в котором реализованы такие алгоритмы, как сортировка двух массивов (сортировка по частотам, но сортируются одновременно и символы, и частоты), функция Swap, которая пригодилась в алгоритме сортировки, функция разделения массива на две равные по значениям части и некоторые другие.

Поскольку в данной работе приходится снова взаимодействовать с парами «ключ-значение», было принято решение использовать КЧ-дерево для реализации ассоциативного массива, но заполнять его не привычными парами «int - char», а парами типа «char - int», «char - string» и «string - char».

Выбор «char - int» обусловлен тем, что в КЧ-дерево надо будет добавлять пары значений, забирая символы сразу из введённой пользователем строки (и, разумеется, для дальнейшей работы необходимы какие-то массивы с символами и их частотами, именно в этом месте создаются эти пары). Если символ не встречался ранее, то в дерево добавляется пара «символ – 1», если же он встречается не первый раз, то нужно просто увеличить частоту («символ – 2» и т.д.). Именно для этого были добавлены перегрузки операторов = и []. Почему тут не подходит реализация через пары «int - char»? Пары должны быть составлены так, что в одном узле должен находиться символ и соответствующая ему частота появления во входной строке. Если вставлять через ключ int, как взаимодействовать с этими частотами? В целом, это реализуемо, но заставит сделать несколько дополнительных действий, но я остановился на, как мне кажется, самой оптимальной реализации.

Пара «char - string» выбрана так, поскольку с её помощью мы сможем составить ассоциативный массив, в котором будут храниться символы и их представление в двоичной системе счисления. Данный выбор поможет в будущем правильно закодировать сообщение. Аналогично и с парой «string - char», только эта пара необходима для расшифровки сообщения и перевода его из двоичного кода в текст.

Алгоритм Шеннона-Фано – рекурсивный, поскольку в нем приходится постоянно делить массивы пополам, пока мы не получим отдельные символы и соответствующие им частоты. После того, как алгоритм Шеннона-Фано закончит свою работу, мы получаем готовый ассоциативный массив, с помощью которого удобно составить закодированную строку, чтобы затем вывести её в консоль.

template <typename key\_t, typename value\_t>

Class Node

|  |  |
| --- | --- |
| Элемент | Описание |
| key\_t key; | Ключ узла |
| value\_t value; | Информация, хранящаяся в узле |
| Node\* left; | Указатель на левого потомка данного узла |
| Node\* right; | Указатель на правого потомка данного узла |
| Node\* parent; | Указатель на родителя данного узла |
| ColorName color | Цвет узла (ColorName – это перечисление enum, где BLACK == 0, RED == 1) |

template <typename key\_t, typename value\_t>

Class map

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| Node\* root; | Указатель на корень КЧ-дерева | | |
| Node\* nil; | Пустой лист КЧ-дерева | | |
| size\_t size; | Размер дерева (для тестирования) | | |
| Метод | | Описание | Оценка временной сложности |
| size\_t give\_size() | | Возвращает размер дерева (для тестирования) | О(1) |
| void rightRotate(Node\* curX) | | Правый поворот – вспомогательный алгоритм (для вставки нового элемента в дерево) | О(1) |
| void leftRotate(Node\* curX) | | Левый поворот – вспомогательный алгоритм (для вставки нового элемента в дерево) | О(1) |
| void insertFixup(Node\* to\_insert) | | Восстановление свойств КЧ-дерева после вставки нового элемента | О(logn) |
| void transplant(Node\* x, Node\* y) | | Перемещение поддеревьев | О(1) |
| void removeFixup(Node\* x) | | Восстановление свойств КЧ-дерева после удаления элемента | О(logn) |
| void printHelp(Node\* current) | | Вспомогательный рекурсивный метод (для вывода информации о дереве в консоль) | O(logn) |
| void recursiveGet\_Keys(Node\* current, stack<key\_t>\* keys\_list) | | Вспомогательный рекурсивный метод (для получения списка ключей дерева) | О(logn) |
| void recursiveGet\_Values(Node\* current, stack<value\_t>\* values\_list) | | Вспомогательный рекурсивный метод (для получения списка значений дерева) | O(logn) |
| void insert(key\_t key, value\_t value) | | Вставка нового узла в КЧ-дерево (по ключу и информации) | O(logn) |
| void remove(key\_t key) | | Удаления узла из КЧ-дерева по ключу | O(logn) |
| void clear() | | Очистка (удаление) КЧ-дерева | O(nlogn) |
| stack<key\_t>\* get\_keys() | | Получение списка ключей дерева | O(logn) |
| stack<value\_t>\* get\_values() | | Получение списка значений дерева | O(logn) |
| void print() | | Вывод информации о дереве (о каждом узле, начиная с корня: ключ, значение и цвет) | O(logn) |
| value\_t find(key\_t key) | | Поиск значения по ключу | O(logn) |
| key\_t give\_root\_key() | | Вспомогательный метод для удобного тестирования: возвращает ключ корня дерева | O(1) |
| value\_t give\_root\_value() | | Вспомогательный метод для удобного тестирования: возвращает информацию из корня дерева | O(1) |
| template<class array\_t>  void MakeAnArray(Node\* x, array\_t\* newArray, bool flag) | | Рекурсивно реализованный метод, проходящий по элементам КЧ-дерева и заполняющий массив требуемой информацией | O(n) |

template <class T>

Class st\_node

|  |  |
| --- | --- |
| Элемент | Описание |
| T info | Информация внутри узла стека |
| st\_node\* prev | Указатель на предыдущий элемент стека (на элемент «снизу») |
| st\_node\* next | Указатель на следующий элемент стека (на элемент «сверху») |

template <class T>

Class stack

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| st\_node\* top | Указатель на первый элемент стека | | |
| Метод | | Описание | Оценка временной сложности |
| T pop() | | Удаление элемента из стека, возвращает информацию удаленного элемента | О(1) |
| void push(T to\_push) | | Вставка нового элемента в стек | О(1) |
| void print() | | Вывод информации из элементов стека в консоль | О(n) |
| T what\_at(int index) | | Возвращает данные, находящиеся в элементе, имеющим данный индекс в списке | O(n) |

Реализованные функции

|  |  |  |
| --- | --- | --- |
| Функция | Описание | Оценка временной сложности |
| template<class array\_t>  int SliceItUp(array\_t\* freqs, int freq\_start, int freq\_end) | Деление массива на две равные по значению части. Если дан массив {6, 5, 4, 3, 2, 2}, то функция вернёт индекс 1 (т.е. элемент массива со значением 5). При проверке получается, что 6+5 = 4+3+2+2, значит, метод делит всё верно. | О(n) |
| template <class array\_t>  void Swap(array\_t\* arr, int first, int second) | Вспомогательный элемент для удобной перестановки местами двух элементов массива | О(1) |
| template<class array\_t1, class array\_t2>  void SortTwoArrays(array\_t1\* freqs, array\_t2\* symbs, int arrays\_size) | Сортировка двух массивов одновременно (используется функция Swap). Поскольку соответствующие элементы массивов составляют пару «частота – символ», то сравниваются значения массива частот, но происходят перестановки сразу в двух массивах | О(n2) |
| void Shannon\_Fano\_Algo(int\* symbs\_freqs, char\* symbs, string& left\_or\_right, string& fullCode, int i\_start, int i\_end, map<char, string>\* mapWithCodes, unsigned& memory\_after) | Алгоритм Шеннона-Фано, реализован рекурсивно | O(n2) |
| void Summing(string usersText, string encodeResult, string decodeResult) | Функция кодирования и декодирования строки; в функции используется алгоритм Шеннона-Фано | O(n2) |

# Описание реализованных unit-тестов

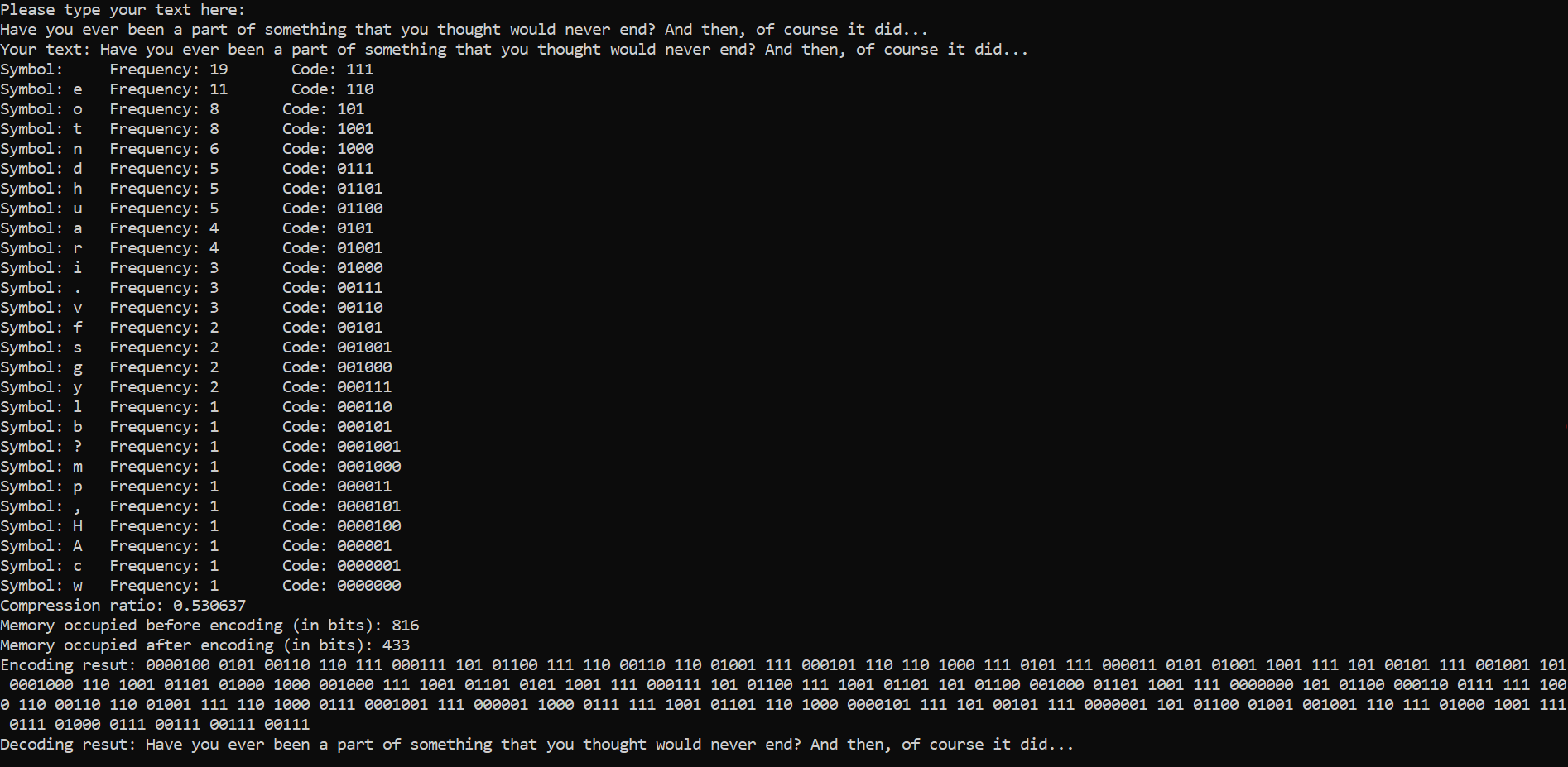
|  |  |
| --- | --- |
| SliceTest | Проверка функции SliceItUp, правильно ли делится массив с числами на две части |
| DoubleSort | Проверка функции SortTwoArrays, правильно ли сортируются оба массива |
| EncodeDecode | Проверка функции Summing, правильно ли кодируется и декодируется сообщение |

# Пример работы.

Первая цитата:

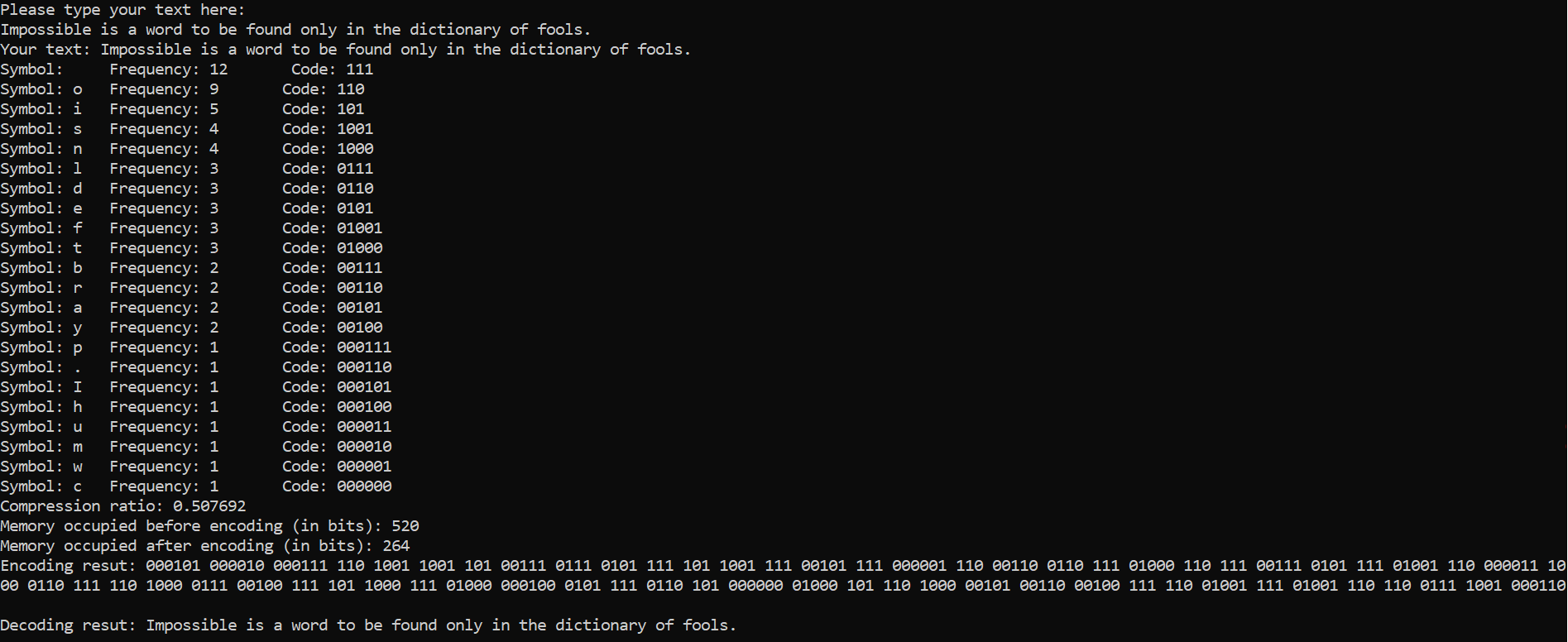
Have you ever been a part of something that you thought would never end? And then, of course it did...

(c) Rise Against - Everchanging



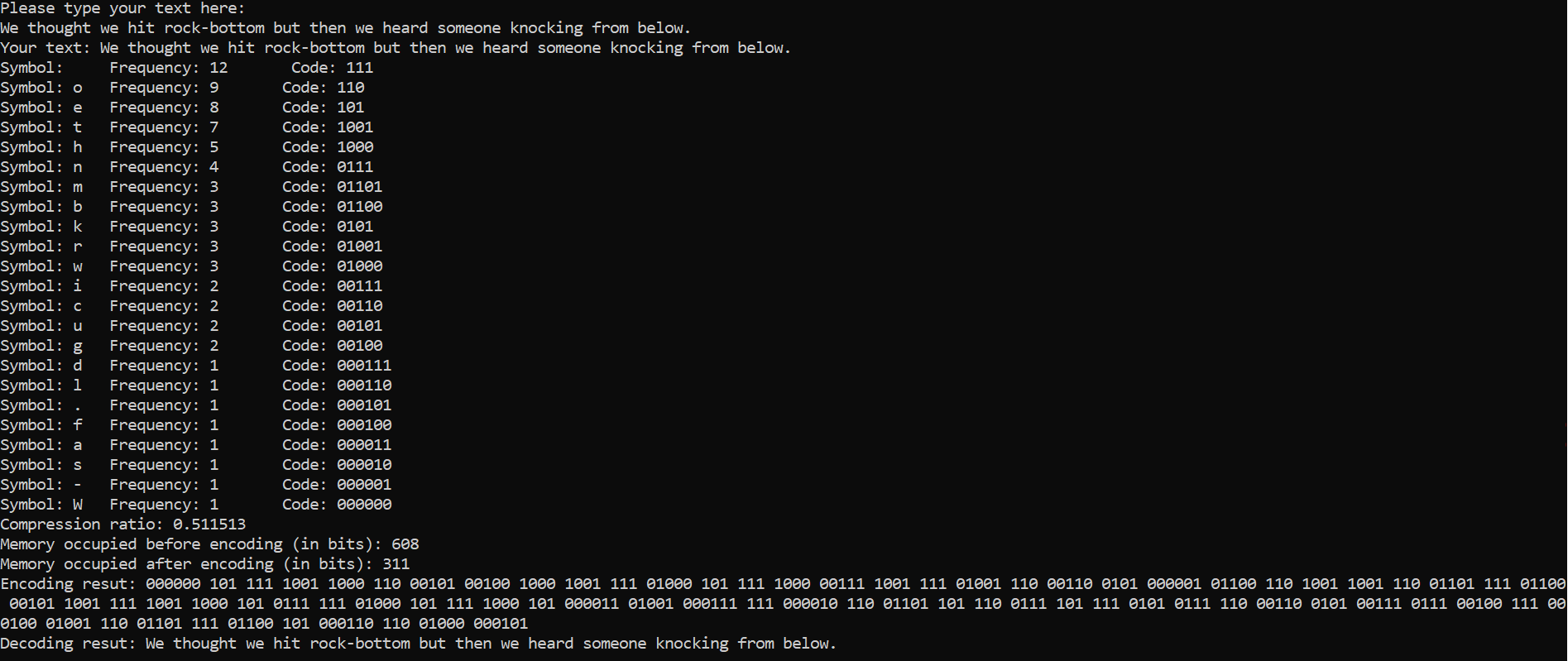
Вторая цитата:

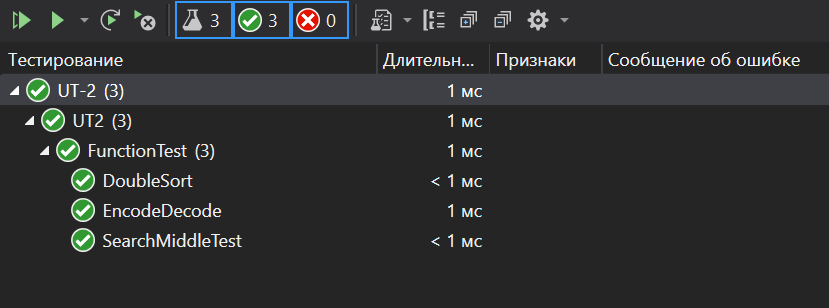
Impossible is a word to be found only in the dictionary of fools.  
(c) Napoleon Bonaparte



Третья цитата:

We thought we hit rock-bottom but then we heard someone knocking from below.  
(c) Stanisław Jerzy Lec





# Листинг.

|  |
| --- |
| Stack.h |
| #pragma once  #include <stdexcept>  #include <iostream>  using namespace std;  template <class T>  class stack  {  private:  class st\_node  {  public:  st\_node(T info, st\_node\* prev = nullptr, st\_node\* next = nullptr)  {  this->info = info;  this->prev = prev;  this->next = next;  }  ~st\_node() {};  T info;  st\_node\* prev;  st\_node\* next;  };  public:  st\_node\* top;  stack()  {  top = nullptr;  }  void push(T to\_push)  {  if (top)  {  st\_node\* buf = new st\_node(to\_push);  buf->prev = top;  top->next = buf;  top = buf;  }  else top = new st\_node(to\_push);  }  T pop()  {  if (!top)  throw out\_of\_range("An error occured.");  if (top->prev == nullptr)  {  T tmp = top->info;  top = nullptr;  delete top;  return tmp;  }  else  {  st\_node node\_temp = top;  T tmp = node\_temp->info;  top = top->prev;  top->next = nullptr;  node\_temp->prev = nullptr;  delete node\_temp;  return tmp;  }  }  void print()  {  st\_node\* current = top;  if (top)  {  while (current->prev)  {  cout << current->info << " ";  current = current->prev;  }  cout << current->info;  cout << endl;  }  else throw out\_of\_range("An error occured: stack is empty.");  }  T what\_at(int index)  {  st\_node\* current = top;  int i = 0;  while (i != index)  {  if (current->prev)  current = current->prev;  else throw "No such element.";  i++;  }  return current->info;  }  ~stack()  {  T tmp;  while (top)  {  tmp = pop();  }  }  }; |
| Map.h |
| #pragma once  #include "Stack.h"  #include <iostream>  #include <stdexcept>  using namespace std;  enum ColorName  {  BLACK,  RED  };  template <typename key\_t, typename value\_t>  class map  {  private:  class Node  {  public:  key\_t key;  value\_t value;  ColorName color;  Node\* left;  Node\* right;  Node\* parent;  Node(key\_t key, value\_t value, ColorName color, Node\* left, Node\* right, Node\* parent)  {  this->key = key;  this->value = value;  this->color = color;  this->left = left;  this->right = right;  this->parent = parent;  }  Node() {}  ~Node() {}  };  private:  Node\* root;  Node\* nil;  size\_t size; // it is necessary to remember size (number of nodes) of tree to make "get\_keys()" and "get\_values()" with iterator correctly  // some helper methods  void rightRotate(Node\* curX)  {  Node\* curY = curX->left; // Y is a left son for X!  curX->left = curY->right;  if (curY->right != nil) // if right son of Y exists  curY->right->parent = curX;  if (curY != nil)  curY->parent = curX->parent; // parent of x is a parent of y  if (curX->parent == nil) // no parent for X  root = curY; // Y is root of tree  else  {  if (curX == curX->parent->left)  curX->parent->left = curY;  else  curX->parent->right = curY;  }  curY->right = curX; // now x is right son of y  if (curX != nil)  curX->parent = curY;  }  void leftRotate(Node\* curX)  {  Node\* curY = curX->right; // Y is a right son for X!  curX->right = curY->left;  if (curY->left != nil) // if left son of Y exists  curY->left->parent = curX;  if (curY != nil)  curY->parent = curX->parent; // parent of x is a parent of y  if (curX->parent == nil) // no parent for X  root = curY; // Y is root of tree  else  {  if (curX == curX->parent->left)  curX->parent->left = curY;  else  curX->parent->right = curY;  }  curY->left = curX; // now x is left son of y  if (curX != nil)  curX->parent = curY;  }  void insertFixup(Node\* to\_insert) // fix the tree after the "insert" operation  {  while ((to\_insert->parent->color == RED)&&(to\_insert != root))  {  if (to\_insert->parent == to\_insert->parent->parent->left)  {  Node\* uncle = to\_insert->parent->parent->right;  if (uncle->color == RED)  {  // CASE I  to\_insert->parent->color = BLACK;  uncle->color = BLACK;  to\_insert->parent->parent->color = RED;  to\_insert = to\_insert->parent->parent;  }  else  {  if (to\_insert == to\_insert->parent->right)  {  // CASE II  to\_insert = to\_insert->parent;  leftRotate(to\_insert);  }  // CASE III  to\_insert->parent->color = BLACK;  to\_insert->parent->parent->color = RED;  rightRotate(to\_insert->parent->parent);  }  }  else  {  Node\* uncle = to\_insert->parent->parent->left;  if (uncle->color == RED)  {  // CASE I  to\_insert->parent->color = BLACK;  uncle->color = BLACK;  to\_insert->parent->parent->color = RED;  to\_insert = to\_insert->parent->parent;  }  else  {  if (to\_insert == to\_insert->parent->left)  {  // CASE II  to\_insert = to\_insert->parent;  rightRotate(to\_insert);  }  // CASE III  to\_insert->parent->color = BLACK;  to\_insert->parent->parent->color = RED;  leftRotate(to\_insert->parent->parent);  }  }  }  root->color = BLACK;  }  // helpers for "remove" method  void transplant(Node\* x, Node\* y) // swap x and y  {  if (x->parent == nil)  root = y;  else  if (x == x->parent->left)  x->parent->left = y;  else x->parent->right = y;  y->parent = x->parent;  }  void removeFixup(Node\* x) // fix the tree after the "remove" operation  {  Node\* w;  while ((x != root) && (x->color == BLACK))  {  if (x == x->parent->left)  {  w = x->parent->right;  if (w->color == RED) // brother of x is RED  {  // CASE I  w->color = BLACK;  x->parent->color = RED;  leftRotate(x->parent);  w = x->parent->right;  }  if ((w->left->color == BLACK) && (w->right->color == BLACK)) // w is BLACK, his sons are BLACK too  {  // CASE II  w->color = RED;  x = x->parent;  }  else  {  // CASE III  if (w->right->color == BLACK) // w, brother of x, is BLACK, left son of w is RED, right - BLACK  {  w->left->color = BLACK;  w->color = RED;  rightRotate(w);  w = x->parent->right;  }  // CASE IV  w->color = x->parent->color; // w, brother of x, is BLACK, his right son is RED  x->parent->color = BLACK;  w->right->color = BLACK;  leftRotate(x->parent);  x = root;  }  }  else  {  w = x->parent->left;  if (w->color == RED)  {  // CASE I  w->color = BLACK;  x->parent->color = RED;  rightRotate(x->parent);  w = x->parent->left;  }  if ((w->right->color == BLACK) && (w->left->color == BLACK))  {  // CASE II  w->color = RED;  x = x->parent;  }  else  {  // CASE III  if (w->left->color == BLACK)  {  w->right->color = BLACK;  w->color = RED;  leftRotate(w);  w = x->parent->left;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->left->color = BLACK;  rightRotate(x->parent);  x = root;  }  }  }  x->color = BLACK;  }  void printHelp(Node\* current) // recursive helper method  { // can be done with iterators or with recursion  //dftIterator iter(root, nil); // iterator starts from root!  //Node\* current;  string color;  /\*while (iter.has\_next())  {  current = iter.next();  if (current->color)  color = "RED";  else color = "BLACK";  cout << "Key: " << current->key << " Data: " << current->value << " Color: " << color << endl;  }\*/  if (current->color)  color = "RED";  else color = "BLACK";  if (current != nil)  {  cout << "Key: " << current->key << " Data: " << current->value << " Color: " << color << endl;  if (current->right != nil)  printHelp(current->right);  if (current->left != nil)  printHelp(current->left);  }  else return; // there's no node  }  void recursiveGet\_Keys(Node\* current, stack<key\_t>\* keys\_list) // recursive helper method (get\_keys)  {  if (current == nil)  return; // end of branch  else  {  keys\_list->push(current->key);  if (current->right != nil)  recursiveGet\_Keys(current->right, keys\_list);  }  if (current->left != nil)  recursiveGet\_Keys(current->left, keys\_list);  }  void recursiveGet\_Values(Node\* current, stack<value\_t>\* values\_list) // recursive helper method (get\_values)  {  if (current == nil)  return; // end of branch  else  {  values\_list->push(current->value);  if (current->right != nil)  recursiveGet\_Values(current->right, values\_list);  }  if (current->left != nil)  recursiveGet\_Values(current->left, values\_list);  }  public:  map()  {  this->nil = new Node();  this->root = nil;  this->size = 0;  }  ~map()  {  clear();  }  void createAtree()  {  this->nil = new Node();  this->root = nil;  this->size = 0;  }  void insert(key\_t key, value\_t value) // inserts a node to the map  {  // new node is ALWAYS RED  // in some cases it will become BLACK  Node\* to\_insert = new Node();  to\_insert->key = key;  to\_insert->value = value;  Node\* y = nil;  Node\* x = root;  while (x != nil)  {  y = x;  if (key < x->key)  x = x->left;  else if (key > x->key)  x = x->right;  else if (key == x->key) // it is necessary to make operator overloading correctly  throw -1;  }  to\_insert->parent = y;  if (y == nil)  {  root = to\_insert;  }  else  if (to\_insert->key < y->key)  y->left = to\_insert;  else  y->right = to\_insert;  to\_insert->left = nil;  to\_insert->right = nil;  to\_insert->color = RED;  size++;  insertFixup(to\_insert);  }  void remove(key\_t key) // removes a node from map by key  {  if (root == nil)  throw "Nothing to remove: tree is empty.";  Node\* toBeDeleted = root;  Node\* x, \* min;  while (toBeDeleted->key != key)  {  if (key > toBeDeleted->key)  toBeDeleted = toBeDeleted->right;  else  toBeDeleted = toBeDeleted->left;  if (toBeDeleted == nil)  throw "There's no node in RB-Tree with that key.";  }  ColorName originalColor = toBeDeleted->color;  if (toBeDeleted->left == nil)  {  x = toBeDeleted->right;  transplant(toBeDeleted, toBeDeleted->right);  }  else  if (toBeDeleted->right == nil)  {  x = toBeDeleted->left;  transplant(toBeDeleted, toBeDeleted->left);  }  else  {  min = toBeDeleted->right;  while (min->left != nil)  min = min->left;  originalColor = min->color;  x = min->right;  if (min->parent == toBeDeleted)  x->parent = min;  else  {  transplant(min, min->right);  min->right = toBeDeleted->right;  min->right->parent = min;  }  transplant(toBeDeleted, min);  min->left = toBeDeleted->left;  min->left->parent = min;  min->color = toBeDeleted->color;  }  if (originalColor == BLACK)  removeFixup(x);  size--;  }  void clear() // clears map  {  while (root != nil)  remove(root->key);  cout << "Map is empty." << endl;  }  stack<key\_t>\* get\_keys() // returns array of keys (need to use size and iterator)  {  if (root == nil)  throw "Nothing to get: map is empty.";  stack<key\_t>\* out\_list = new stack<key\_t>();  recursiveGet\_Keys(root, out\_list);  return out\_list;  }  stack<value\_t>\* get\_values() // returns array of values (need to use size and iterator)  {  // i could do that with iterators but i suck on the topic of iteratos :c  if (root == nil)  throw "Nothing to get: map is empty.";  stack<value\_t>\* out\_list = new stack<value\_t>();  recursiveGet\_Values(root, out\_list);  return out\_list;  }  void print()  {  if (root == nil)  throw "Nothing to print: map is empty.";  else  {  cout << "Root: ";  printHelp(root);  cout << endl;  }  }  Node\* find(key\_t key) // finds an element by key and returns value from this element  {  if (root == this->nil)  throw "Nothing to find: tree is empty.";  else  {  Node\* toFind = root;  while (toFind->key != key)  {  if (toFind->parent == nullptr)  throw "There's no such element.";  if (toFind == this->nil)  throw "There's no node in RB-Tree with that key.";  if (key > toFind->key)  toFind = toFind->right;  else  toFind = toFind->left;  }  return toFind;  }  }  size\_t give\_size() // this method just gives us size  {  return size;  }  value\_t& operator [] (const key\_t key)  {  return find(key)->value;  }  void operator = (value\_t new\_data)  {  this->value = new\_data;  }  template<class array\_t>  void MakeAnArray(Node\* x, array\_t\* newArray, bool flag)  {  static int i = 0;  // newArray is the one we need to fill with nodes: MakeAnArray to symbols just to fill an array with symbols  // and then MakeAnArray to frequencies of these symbols to make pairs symbol-frequency  if (x->parent == nullptr)  throw "Tree is empty.";  if (x == nil)  return;  else  {  if (flag == 1) // freqs  {  newArray[i++] = x->key; // that's the purpose why there is STATIC INT  if (x->left != nil)  MakeAnArray(x->left, newArray, flag); // that's the purpose why there is STATIC INT  }  else // symbs  {  newArray[i++] = x->value;  if (x->left != nil)  MakeAnArray(x->left, newArray, flag);  }  }  if (x->right != nil)  MakeAnArray(x->right, newArray, flag);  }  Node\* give\_root()  {  return root; // this is because our map root is private and we need pointer  }  }; |
| DeCodesAndEnCodes.h |
| #pragma once  #include "Map.h"  #include <stdexcept>  #include <iostream>  using namespace std;  template<class array\_t>  int SliceItUp(array\_t\* freqs, int freq\_start, int freq\_end)  {  // we have 2 arrays: freqs[] and symbs[]  // FOR EXAMPLE: freqs = { 6, 5, 4, 3, 2, 2 }  // FOR EXAMPLE: symbs = { a, b, c, d, e, f }  // because our algorythm is done recursively, we should have two indexes: start and end  double average = 0;  int i = 0;  for (i = freq\_start; i < freq\_end; i++)  average = average + freqs[i];  // average = 22 for our example  average = average / 2; // now it's 11...  int sum = 0, median = freq\_start;  i = freq\_start;  while ((i < freq\_end) && (sum + freqs[i] < average))  {  sum += freqs[i];  i++;  median++;  }  // i = 0, median = 0;  // 0 + 6 < 11 - true  // i = 1; median = 1;  // 6 + 5 < 11 - false  // median = 1  // now we can slice our { 6 [5] 4 3 2 2 } and { a [b] c d e f }  // and make 4 arrays - {6 5} and {a b}, {4 3 2 2} and {c d e f}  // and then slice every pair again while arrays\_size != 1  return median;  }  template <class array\_t>  void Swap(array\_t\* arr, int first, int second)  {  array\_t tmp;  tmp = arr[first];  arr[first] = arr[second];  arr[second] = tmp;  }  template<class array\_t1, class array\_t2>  void SortTwoArrays(array\_t1\* freqs, array\_t2\* symbs, int arrays\_size)  {  // we have 2 arrays: freqs[] and symbs[]  // freqs size == arrays\_size  // symbs size == arrays\_size  // FOR EXAMPLE: freqs = { 2, 6, 3, 5, 2, 4 }  // FOR EXAMPLE: symbs = { f, a, d, b, e, c }  // we are sorting freqs  for (int i = 0; i < arrays\_size; ++i)  {  for (int j = i + 1; j < arrays\_size; ++j)  {  if (freqs[i] < freqs[j])  {  Swap(freqs, i, j);  Swap(symbs, i, j);  }  }  }  // freqs = { 6, 5, 4, 3, 2, 2 }  // symbs = { a, b, c, d, e, f }  }  // The main algorithm of our program  void Shannon\_Fano\_Algo(int\* symbs\_freqs, char\* symbs, string& left\_or\_right, string& fullCode, int i\_start, int i\_end, map<char, string>\* mapWithCodes, unsigned& memory\_after)  {  // couldn't make it with the char\*, i'm sorry  string tmpCode;  tmpCode = fullCode + left\_or\_right; // e.g. "1001" + '0'  if (i\_start == i\_end) // end of an algorithm (symbs[i\_start] == symbs[i\_end] == 'a')  {  memory\_after += tmpCode.length() \* symbs\_freqs[i\_start];  mapWithCodes->insert(symbs[i\_start], tmpCode); // filling a map array with the pair like ("A", "10011")  cout << "Symbol: " << symbs[i\_start] << " Frequency: " << symbs\_freqs[i\_start] << " Code: " << tmpCode << endl;  return;  }  int array\_median = SliceItUp(symbs\_freqs, i\_start, i\_end); // {6 [5] 4 3 2 2}  string one = "1"; // i can't just write Shannon\_Fano\_Algo(..., ..., "1", ...), because he can't transform const char[2] in string type  string zero = "0";  Shannon\_Fano\_Algo(symbs\_freqs, symbs, one, tmpCode, i\_start, array\_median, mapWithCodes, memory\_after); // example is given in the comments in SliceItUp(): {6 5}  Shannon\_Fano\_Algo(symbs\_freqs, symbs, zero, tmpCode, array\_median + 1, i\_end, mapWithCodes, memory\_after); // {4 3 2 2}  }  // analog of int main() but as a function (to combine it all together and then test it)  void Summing(string usersText, string &encodeResult, string &decodeResult)  {  // we need usersText only to read from console, find memory, make symbols arrays and that's all  cout << "Your text: " << usersText << endl;  unsigned memBe4 = usersText.length() \* 8, memAfter = 0; // it's needed to make final statistics  map<char, int>\* charAndFreqs;  charAndFreqs = new map<char, int>();  for (size\_t i = 0; i < usersText.length(); i++)  {  try  {  charAndFreqs->insert(usersText[i], 1); // we are making pairs ('A', 1)  }  catch (int ex)  {  // for that purpose we need an operator overloading  if (ex == -1)  (\*charAndFreqs)[usersText[i]]++; // if 'A' meets 2+ times: ('A', 2), ('A', 3), ...  }  }  int\* symb\_freqs = new int[charAndFreqs->give\_size()];  char\* symbs = new char[charAndFreqs->give\_size()];  charAndFreqs->MakeAnArray(charAndFreqs->give\_root(), symb\_freqs, 0);  charAndFreqs->MakeAnArray(charAndFreqs->give\_root(), symbs, 1);  SortTwoArrays(symb\_freqs, symbs, charAndFreqs->give\_size()); // to make {6 5 4 3 2 2} and { a b c d e f} by sorting two arrays at the same time  // Encoding process  string temp = "";  map<char, string>\* charsAndCodes;  charsAndCodes = new map<char, string>();  Shannon\_Fano\_Algo(symb\_freqs, symbs, temp, temp, 0, charAndFreqs->give\_size() - 1, charsAndCodes, memAfter);  string Encoding\_Result = "";  for (int i = 0; i < usersText.length(); i++)  {  Encoding\_Result += (\*charsAndCodes)[usersText[i]] + ' '; // string will look like "1010 1001 001 11 11 11 ..."  }  encodeResult = Encoding\_Result;  cout << "Compression ratio: " << (float)memAfter / (double)memBe4 << endl;  cout << "Memory occupied before encoding (in bits): " << memBe4 << endl;  cout << "Memory occupied after encoding (in bits): " << memAfter << endl;  cout << "Encoding resut: " << encodeResult << endl;  // Decoding process  map<string, char>\* codesAndChars;  codesAndChars = new map<string, char>();  for (int i = 0; i < charAndFreqs->give\_size(); ++i)  {  codesAndChars->insert((\*charsAndCodes)[symbs[i]], symbs[i]);  }  string Decoding\_Result = "", Code;  for (int i = 0; i < Encoding\_Result.length(); i++)  {  Code = "";  while (Encoding\_Result[i] != ' ')  {  Code += Encoding\_Result[i++];  }  Decoding\_Result += (\*codesAndChars)[Code];  }  decodeResult = Decoding\_Result;  cout << "Decoding resut: " << decodeResult << endl;  delete[] symb\_freqs;  delete[] symbs;  } |
| UT-2.cpp |
| #include "CppUnitTest.h"  #include <stdexcept>  #include "../Term4\_LabWork2/DeCodesAndEnCodes.h"  #include "../Term4\_LabWork2/Stack.h"  #include "../Term4\_LabWork2/Map.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UT2  {  TEST\_CLASS(FunctionTest)  {  public:  TEST\_METHOD(SearchMiddleTest)  {  int freqs[6]{ 6, 5, 4, 3, 2, 2 };  int median = SliceItUp(freqs, 0, 6);  Assert::IsTrue(median == 1);  }  TEST\_METHOD(DoubleSort)  {  int\* freqs = new int[6] { 2, 6, 3, 5, 2, 4 };  char\* symbs = new char[6]{ 'f', 'a', 'd', 'b', 'e', 'c'};  char\* toControl = new char[6]{ 'a', 'b' , 'c', 'd','e','f'};  SortTwoArrays(freqs, symbs, 6);  for (int i = 0; i < 6; i++)  {  Assert::IsTrue(symbs[i] == toControl[i]);  }  }  TEST\_METHOD(EncodeDecode)  {  string str = "to be or not to be";  string test = "011 10 11 001 010 11 10 0000 11 0001 10 011 11 011 10 11 001 010";  string encoding\_result, decoding\_result;  Summing(str, encoding\_result, decoding\_result);  for (int i = 0; i < encoding\_result.length() - 1; i++)  {  Assert::IsTrue(encoding\_result[i] == test[i]);  }  Assert::IsTrue(decoding\_result == str);  }  };  } |

# Вывод

Я изучил и научился реализовывать алгоритм Шеннона-Фано с помощью красно-чёрного дерева и работать с ним.