**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Алгоритмы кодирования

**Вариант 2**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Кнауб К.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

# Постановка задачи

Реализовать кодирование и декодирование по алгоритму Шеннона-Фано входной стоки, вводимой через консоль. Посчитать объем памяти, который занимает исходная и закодированная строки. Вывести на экран таблицу частот и кодов, результат кодирования и декодирования, коэффициент сжатия.

# Описание реализуемого алгоритма и структур данных

В данной лабораторной работе был использован созданный раннее ассоциативный массив на основе красно-черного дерева.

|  |  |
| --- | --- |
| **Название** | **Описание** |
| void push\_back() | добавление в конец списка |
| void push\_front() | добавление в начало списка |
| void pop\_back() | удаление последнего элемента |
| void pop\_front() | удаление первого элемента |
| void insert() | добавление элемента по индексу |
| data\_T at() | получение элемента по индексу |
| void remove() | удаление элемента по индексу |
| size\_t GetSize() | получение размера списка |
| void print\_to\_console() | вывод элементов в консоль через разделитель |
| bool isEmpty() | проверка на пустоту списка |
| void clear() | удаление всех элементов списка |
| void reverse | изменение порядка элементов на обратный |
| void set(size\_t, data\_T) | замена элемента по индексу |
| data\_T top() | получение первого элемента в списке |

|  |  |
| --- | --- |
| **Название** | **Назначение** |
| void insert | добавление элемента с ключом и значением |
| void remove | удаление элемента дерева по ключу |
| Tree\_Node\* find | поиск элемента по ключу |
| void clear | очищение ассоциативного массива |
| void get\_keys | возвращает список ключей |
| void get\_values | возвращает список значений |
| void print | вывод в консоль |
| void Insert\_fixup | восстанавливает свойства дерева после вставки нового узла в дерево |
| void Transplant | перемещает поддеревья в дереве |
| void Remove\_fixup | восстанавливает свойства дерева после удаления узла из дерева |
| void NIL\_Root | создает новый объект NIL |
| void Left\_turn | метод левого поворота |
| void Right\_turn | метод правого поворота |
| Tree\_Node\* get\_Root | передаёт методам необходимых указателей |
| Tree\_Node\* get\_Parent | передает методам необходимых указателей |
| Tree\_Node\* Tree\_min | возвращает указатель на минимальный элемент |

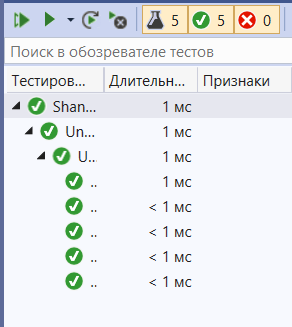
|  |  |
| --- | --- |
| **Название** | **Описание** |
| void Bubble\_Sort() | сортировка массивов |
| void Search\_Border() | поиск середины массива |
| void Shannon\_Fano() | алгоритм шеннона-фано |
| Void Fill\_Array() | работа с массивом |
| void Decoding() | кодирование и декодирование |

# Оценка временной сложности каждого алгоритма

|  |  |
| --- | --- |
| void Bubble\_Sort() |  |
| void Search\_Border() |  |
| void Algorithm\_Fano() |  |
| void Fill\_Array() |  |
| void Decoding() |  |

# Описание реализованных unit-тестов

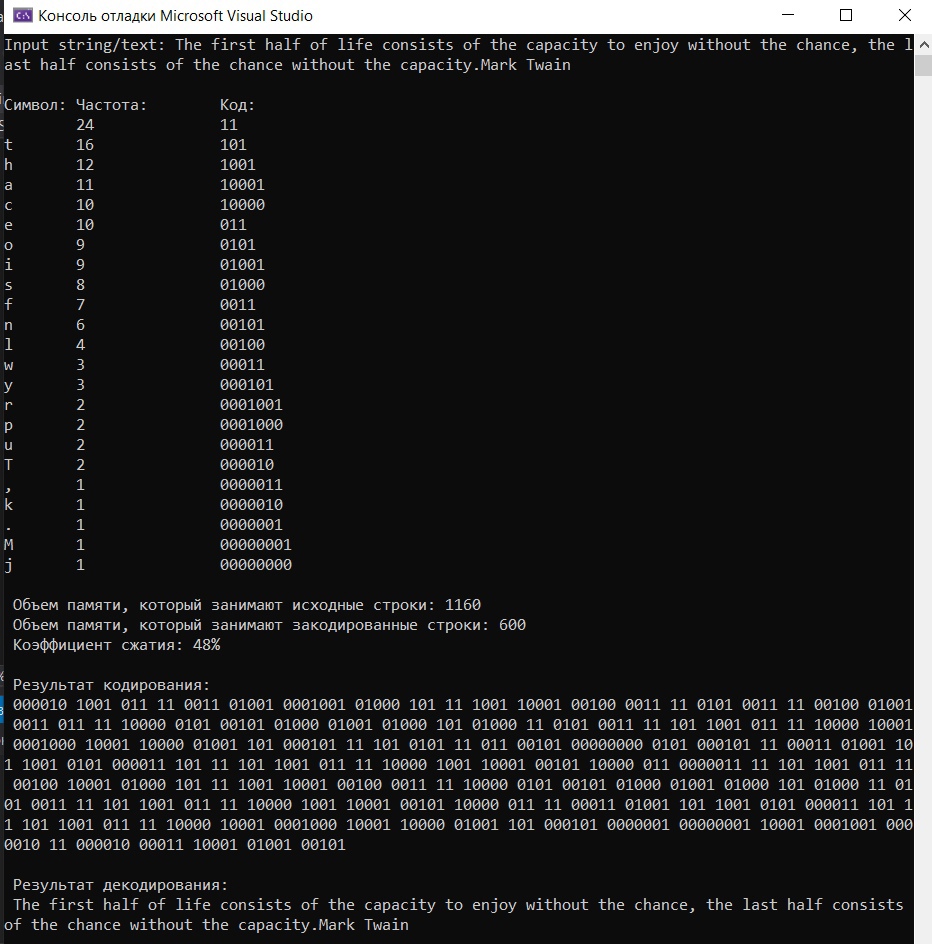
|  |
| --- |
| **Название теста** |
| Bubble\_Sorting\_1 |
| Bubble\_Sorting\_2 |
| Border\_Search\_1 |
| Border\_Search\_2 |
| Algorithm \_Fano |



# Пример работы

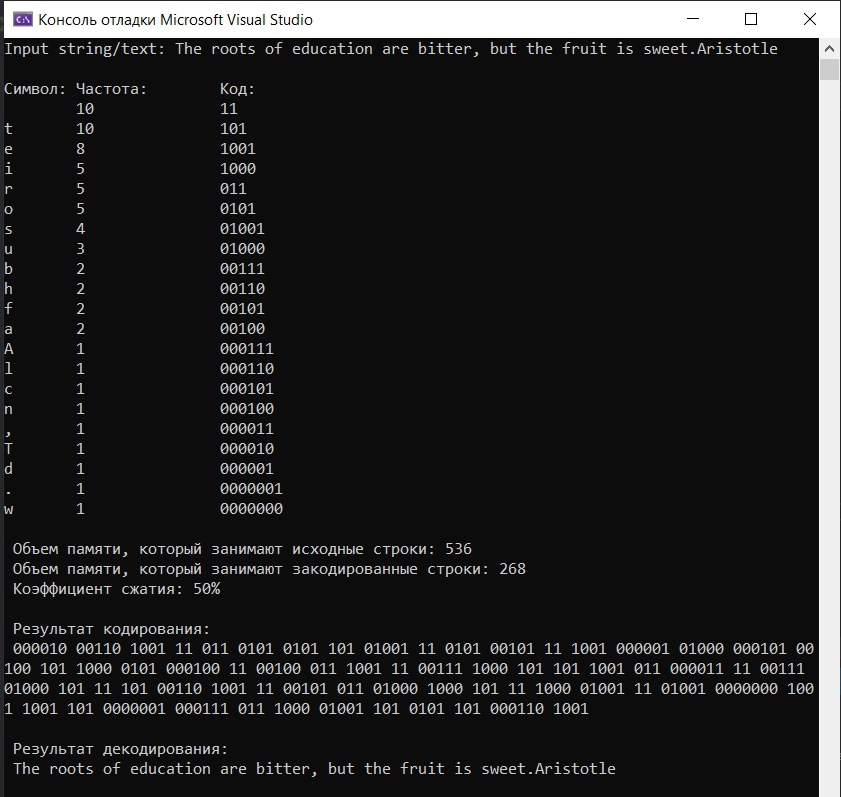
The first half of life consists of the capacity to enjoy without the chance, the last half consists of the chance without the capacity.

Mark Twain



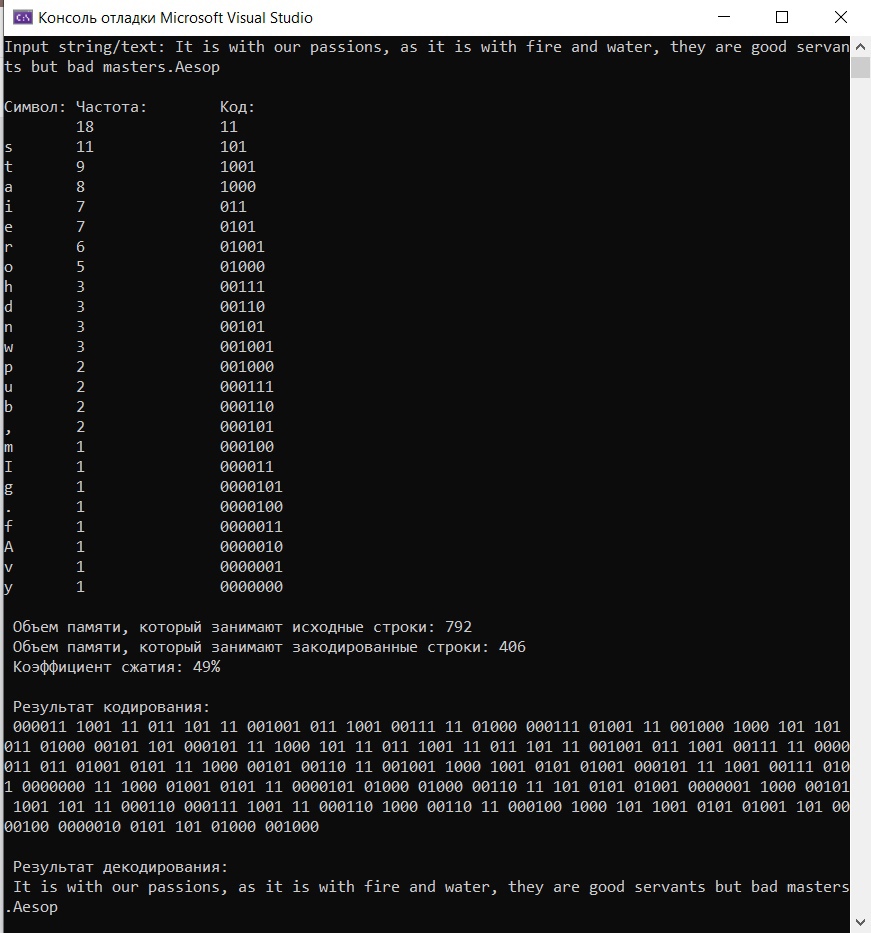
The roots of education are bitter, but the fruit is sweet.

Aristotle



It is with our passions, as it is with fire and water, they are good servants but bad masters.

Aesop



# Листинг

|  |
| --- |
| **LinkedList.h** |
| #pragma once  template<class NODE\_TYPE>  class LinkedList  {  public:    LinkedList();  ~LinkedList();  void push\_back(NODE\_TYPE newEl);  void push\_front(NODE\_TYPE newEl);  void pop\_back();  void pop\_front();  void insert(NODE\_TYPE newEl, size\_t index);  int at(size\_t index) const;  void remove(size\_t index);  size\_t get\_size();  void print\_to\_console();  void clear();  void set(size\_t index, NODE\_TYPE newData);  bool isEmpty();  void sort();  size\_t find\_first(LinkedList \*list);  bool isPrinted();  private:  class Node  {  public:  Node(NODE\_TYPE data, Node\* next = nullptr, Node\* prev = nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  NODE\_TYPE data;  Node\* next;  Node\* prev;  };    Node\* head;  Node\* tail;  size\_t size;  };  #include "LinkedList.inl" |
| **LinkedList.inl** |
| #include "LinkedList.h"  #include <iostream>  #include <stdexcept>  #include <string>  #include <iomanip>  using namespace std;  template<class NODE\_TYPE>  LinkedList<NODE\_TYPE>::LinkedList()  {  head = nullptr;  tail = nullptr;  size = 0;  }  template<class NODE\_TYPE>  LinkedList<NODE\_TYPE>::~LinkedList()  {  clear();  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::push\_back(NODE\_TYPE newEl)  {  Node\* temp;  if (size == 0)  {  head = new Node(newEl);  tail = head;  }  else  {  temp = tail;  tail->next = new Node(newEl);  tail = tail->next;  tail->prev = temp;  }  size++;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::push\_front(NODE\_TYPE newEl)  {  Node\* temp;  if (size == 0)  {  head = new Node(newEl);  tail = head;  }  else  {  temp = head;  head = new Node(newEl, head);  temp->prev = head;  }  size++;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::pop\_back()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = nullptr;  tail = nullptr;  }  else  {  Node\* cur = tail;  cur = cur->prev;  cur->next = nullptr;  delete tail;  tail = cur;  cur->prev = tail->prev;  }  size--;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::pop\_front()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = nullptr;  tail = nullptr;  }  else  {  Node\* cur = head;  cur = cur->next;  cur->prev = nullptr;  delete head;  head = cur;  }  size--;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::insert(NODE\_TYPE newEl, size\_t index)  {  Node\* temp\_pointer = nullptr, \* cur;  Node\* temp = new Node(newEl);  cur = head;  int i = 0;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  while (cur != NULL && i<int(index))  {  cur = cur->next;  i++;  }  if (cur == NULL)  {  if (head == NULL)  {  cout << "List is empty\n";  return;  }  return;  }  if (cur != head)  {  temp\_pointer = cur->prev;  cur->prev = temp;  temp->next = cur;  temp->prev = temp\_pointer;  }  else  push\_front(newEl);  if (temp\_pointer != NULL)  temp\_pointer->next = temp;  size++;  }  template<class NODE\_TYPE>  int LinkedList<NODE\_TYPE>::at(size\_t index) const  {  if (size == 0)  cout << "List is empty\n";  if (index >= size || index < 0) {  throw out\_of\_range("Index is greater than list size");  }  if ((index < size) && (size != 0))  {  size\_t counter = 0;  Node\* cur = head;  while (counter != index)  {  cur = cur->next;  counter++;  }  return cur->data;  }  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::remove(size\_t index)  {  size\_t counter = 0;  Node\* cur = head;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  while (counter != index)  {  cur = cur->next;  counter++;  }  if (cur == head)  {  pop\_front();  return;  }  if (cur == tail)  {  pop\_back();  return;  }  Node\* prev, \* next;  prev = cur->prev;  next = cur->next;  if (prev != NULL)  prev->next = cur->next;  if (next != NULL)  next->prev = cur->prev;  delete cur;  size--;  }  template<class NODE\_TYPE>  size\_t LinkedList<NODE\_TYPE>::get\_size()  {  return size;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::print\_to\_console()  {  Node\* cur = head;  cout << "List elements:\t";  while (cur != nullptr)  {  cout <<setw(3)<< cur->data <<' ';  cur = cur->next;  }  cout << '\n';  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::clear()  {  while (head)  {  tail = head->next;  delete head;  head = tail;  size--;  }  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::set(size\_t index, NODE\_TYPE newData)  {  Node\* cur = head;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  size\_t counter = 1;  while (counter <= index)  {  counter++;  cur = cur->next;  }  cout << "Element " << cur->data;  cur->data = newData;  cout << " replaced by " << cur->data << '\n';  }  template<class NODE\_TYPE>  bool LinkedList<NODE\_TYPE>::isEmpty()  {  if (head != NULL)  return true;  else  return false;  }  template<class NODE\_TYPE>  inline void LinkedList<NODE\_TYPE>::sort()  {  Node\* cur = head;  while (cur->next)  {  if ((cur->data) < (cur->next->data))  {  NODE\_TYPE temp = cur->next->data;  cur->next->data = cur->data;  cur->data = temp;  cur = cur->next;  sort();  }  else  {  cur = cur->next;  }  }  }  template<class NODE\_TYPE>  bool LinkedList<NODE\_TYPE>::isPrinted()  {  Node\* cur = head;  while (cur != nullptr)  {  cout << cur->data << ' ';  cur = cur->next;  }  cout << '\n';  if (cur == nullptr)  return true;  else  return false;  }  template<class NODE\_TYPE>  size\_t LinkedList<NODE\_TYPE>::find\_first(LinkedList\* list)  {  int counter = 0;  Node\* curL1, \* curL2;  curL1 = (\*list).head;  curL2 = head;  if (curL1 == nullptr || curL2 == nullptr)  return 0;  if ((\*list).get\_size() > size)  {  while (curL2->data != curL1->data && curL1->next != NULL)  {  curL1 = curL1->next;  counter++;  }  while (curL2->data == curL1->data && curL2->next != NULL && curL1->next != NULL)  {  curL2 = curL2->next;  curL1 = curL1->next;  }  if (curL2->next == NULL)  {  return counter;  }  else  return 0;  }  else  {  if ((\*list).get\_size() == size)  if (curL1->data == curL2->data)  goto analysis;  else  return 0;  while (curL1->data != curL2->data && curL2->next != NULL)  {  curL2 = curL2->next;  counter++;  }  analysis:  while (curL1->data == curL2->data && curL1->next != NULL && curL2->next != NULL)  {  curL1 = curL1->next;  curL2 = curL2->next;  }  if (curL1->next == NULL)  {  return counter;  }  else  return 0;  }  } |
| **Tree.h** |
| #pragma once  #include "LinkedList.h"  #include <iostream>  enum NodeColor { BLACK, RED };  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  class RB\_TREE\_Node  {  public:  ~RB\_TREE\_Node();  void NIL\_N\_Root();  void Left\_Rotate(RB\_TREE\_Node\* node\_x);  void Right\_Rotate(RB\_TREE\_Node\* node\_y);  void Insert(NODE\_TYPE\_1 key, NODE\_TYPE\_2 data);  void Insert\_Fixup(RB\_TREE\_Node\* node\_z);  void Transplant(RB\_TREE\_Node\* u, RB\_TREE\_Node\* v);  void Remove(NODE\_TYPE\_1 key);  void Remove\_Fixup(RB\_TREE\_Node\* node\_x);  RB\_TREE\_Node\* Find(NODE\_TYPE\_1 key, RB\_TREE\_Node\* node\_x);  void Clear(RB\_TREE\_Node\* tree);  void Get\_Keys(RB\_TREE\_Node\* node\_x, LinkedList<NODE\_TYPE\_1>\* List\_of\_Keys);  void Get\_Values(RB\_TREE\_Node\* node\_x, LinkedList<NODE\_TYPE\_2>\* List\_of\_Values);  void Print(RB\_TREE\_Node\* node\_x);  RB\_TREE\_Node\* Get\_Root() { return root; }  RB\_TREE\_Node\* Get\_Parent() { return parent; }  template<class ARRAY\_TYPE>  void Fill\_Array(RB\_TREE\_Node\* node\_x, ARRAY\_TYPE\* array, const char\* key\_or\_data)  {  static int i = 0;  if (key\_or\_data == "data")  {  if (node\_x->Get\_Parent() == NULL)  throw "RB Tree is empty";  if (node\_x == NIL)  return;  else  {  array[i++] = node\_x->data;  if (node\_x->left != NIL)  Fill\_Array(node\_x->left, array, key\_or\_data);  }  if (node\_x->right != NIL)  Fill\_Array(node\_x->right, array, key\_or\_data);  }  else if (key\_or\_data == "key")  {  if (node\_x->Get\_Parent() == NULL)  throw "RB Tree is empty";  if (node\_x == NIL)  return;  else  {  array[i++] = node\_x->key;  if (node\_x->left != NIL)  Fill\_Array(node\_x->left, array, key\_or\_data);  }  if (node\_x->right != NIL)  Fill\_Array(node\_x->right, array, key\_or\_data);  }  }    NODE\_TYPE\_2& operator[] (const NODE\_TYPE\_1 key)  {  RB\_TREE\_Node\* node = Find(key, root);  return node->data;  }  void operator = (NODE\_TYPE\_2 new\_data)  {  data = new\_data;  }  void operator ++ () { ++data; }  unsigned Get\_Size() { return size; }  RB\_TREE\_Node\* Tree\_Minimum(RB\_TREE\_Node\* node\_x);  private:  RB\_TREE\_Node\* left, \* right, \* parent;  RB\_TREE\_Node\* root;  RB\_TREE\_Node\* NIL;  NodeColor color;  NODE\_TYPE\_1 key;  NODE\_TYPE\_2 data;  unsigned size;  };  #include "Tree.inl" |
| **Tree.inl** |
| #include "LinkedList.h"  #include "Tree.h"  #include <iostream>  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::~RB\_TREE\_Node()  {  Clear(root);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::NIL\_N\_Root()  {  NIL = new RB\_TREE\_Node;  NIL->color = BLACK;  root = NIL;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Left\_Rotate(RB\_TREE\_Node\* node\_x)  {  RB\_TREE\_Node\* node\_y = node\_x->right;  node\_x->right = node\_y->left;  if (node\_y->left != NIL)  node\_y->left->parent = node\_x;  if (node\_y != NIL)  node\_y->parent = node\_x->parent;  if (node\_x->parent == NIL)  root = node\_y;  else if (node\_x == node\_x->parent->left)  node\_x->parent->left = node\_y;  else  node\_x->parent->right = node\_y;  node\_y->left = node\_x;  if (node\_x != NIL)  node\_x->parent = node\_y;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Right\_Rotate(RB\_TREE\_Node\* node\_x)  {  RB\_TREE\_Node\* node\_y = node\_x->left;  node\_x->left = node\_y->right;  if (node\_y->right != NIL)  node\_y->right->parent = node\_x;  if (node\_y != NIL)  node\_y->parent = node\_x->parent;  if (node\_x->parent == NIL)  root = node\_y;  else if (node\_x == node\_x->parent->left)  node\_x->parent->left = node\_y;  else  node\_x->parent->right = node\_y;  node\_y->right = node\_x;  if (node\_x != NIL)  node\_x->parent = node\_y;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Insert(NODE\_TYPE\_1 key, NODE\_TYPE\_2 data)  {  RB\_TREE\_Node\* node\_z = new RB\_TREE\_Node;  node\_z->key = key;  node\_z->data = data;  RB\_TREE\_Node\* node\_x = root;  RB\_TREE\_Node\* node\_y = NIL;  while (node\_x != NIL)  {  node\_y = node\_x;  if (node\_z->key == node\_x->key)  throw -1;  if (node\_z->key < node\_x->key)  node\_x = node\_x->left;  else  node\_x = node\_x->right;  }  node\_z->parent = node\_y;  if (node\_y == NIL)  root = node\_z;  else if (node\_z->key < node\_y->key)  node\_y->left = node\_z;  else  node\_y->right = node\_z;  node\_z->left = NIL;  node\_z->right = NIL;  node\_z->color = RED;  Insert\_Fixup(node\_z);  size++;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Insert\_Fixup(RB\_TREE\_Node\* node\_z)  {  while (node\_z->parent->color == RED && node\_z != root)  {  if (node\_z->parent == node\_z->parent->parent->left)  {  RB\_TREE\_Node\* node\_y = node\_z->parent->parent->right;  if (node\_y->color == RED)  {  node\_z->parent->color = BLACK;  node\_y->color = BLACK;  node\_z->parent->parent->color = RED;  node\_z = node\_z->parent->parent;  }  else  {  if (node\_z == node\_z->parent->right)  {  node\_z = node\_z->parent;  Left\_Rotate(node\_z);  }  node\_z->parent->color = BLACK;  node\_z->parent->parent->color = RED;  Right\_Rotate(node\_z->parent->parent);  }  }  else  {  RB\_TREE\_Node\* node\_y = node\_z->parent->parent->left;  if (node\_y->color == RED)  {  node\_z->parent->color = BLACK;  node\_y->color = BLACK;  node\_z->parent->parent->color = RED;  node\_z = node\_z->parent->parent;  }  else  {  if (node\_z == node\_z->parent->left)  {  node\_z = node\_z->parent;  Right\_Rotate(node\_z);  }  node\_z->parent->color = BLACK;  node\_z->parent->parent->color = RED;  Left\_Rotate(node\_z->parent->parent);  }  }  }  root->color = BLACK;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Transplant(RB\_TREE\_Node\* node\_u, RB\_TREE\_Node\* node\_v)  {  if (node\_u->parent == NIL)  root = node\_v;  else if (node\_u == node\_u->parent->left)  node\_u->parent->left = node\_v;  else  node\_u->parent->right = node\_v;  node\_v->parent = node\_u->parent;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Remove(NODE\_TYPE\_1 key)  {  RB\_TREE\_Node\* node\_z = Find(key, root);  RB\_TREE\_Node\* node\_y = node\_z, \* node\_x;  NodeColor node\_y\_original\_color = node\_y->color;  if (node\_z->left == NIL)  {  node\_x = node\_z->right;  Transplant(node\_z, node\_z->right);  }  else if (node\_z->right == NIL)  {  node\_x = node\_z->left;  Transplant(node\_z, node\_z->left);  }  else  {  node\_y = Tree\_Minimum(node\_z->right);  node\_y\_original\_color = node\_y->color;  node\_x = node\_y->right;  if (node\_y->parent == node\_z)  node\_x->parent = node\_y;  else  {  Transplant(node\_y, node\_y->right);  node\_y->right = node\_z->right;  node\_y->right->parent = node\_y;  }  Transplant(node\_z, node\_y);  node\_y->left = node\_z->left;  node\_y->left->parent = node\_y;  node\_y->color = node\_z->color;  }  if (node\_y\_original\_color == BLACK)  Remove\_Fixup(node\_x);  size--;  }  template <class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Remove\_Fixup(RB\_TREE\_Node\* node\_x)  {  while (node\_x != root && node\_x->color == BLACK)  {  if (node\_x == node\_x->parent->left)  {  RB\_TREE\_Node\* node\_w = node\_x->parent->right;  if (node\_w->color == RED)  {  node\_w->color = BLACK;  node\_x->parent->color = RED;  Left\_Rotate(node\_x->parent);  node\_w = node\_x->parent->right;  }  if (node\_w->left->color == BLACK && node\_w->right->color == BLACK)  {  node\_w->color = RED;  node\_x = node\_x->parent;  }  else  {  if (node\_w->right->color == BLACK)  {  node\_w->left->color = BLACK;  node\_w->color = RED;  Right\_Rotate(node\_w);  node\_w = node\_x->parent->right;  }  node\_w->color = node\_x->parent->color;  node\_x->parent->color = BLACK;  node\_w->right->color = BLACK;  Left\_Rotate(node\_x->parent);  node\_x = root;  }  }  else  {  RB\_TREE\_Node\* node\_w = node\_x->parent->left;  if (node\_w->color == RED)  {  node\_w->color = BLACK;  node\_x->parent->color = RED;  Right\_Rotate(node\_x->parent);  node\_w = node\_x->parent->left;  }  if (node\_w->right->color == BLACK && node\_w->left->color == BLACK)  {  node\_w->color = RED;  node\_x = node\_x->parent;  }  else  {  if (node\_w->left->color == BLACK)  {  node\_w->right->color = BLACK;  node\_w->color = RED;  Left\_Rotate(node\_w);  node\_w = node\_x->parent->left;  }  node\_w->color = node\_x->parent->color;  node\_x->parent->color = BLACK;  node\_w->left->color = BLACK;  Right\_Rotate(node\_x->parent);  node\_x = root;  }  }  }  node\_x->color = BLACK;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>\* RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Find(NODE\_TYPE\_1 key, RB\_TREE\_Node\* node\_x)  {  if (node\_x->Get\_Parent() == NULL)  throw "Element not found";  if (node\_x == NIL || node\_x->key == key)  return node\_x;  if (key < node\_x->key)  return Find(key, node\_x->left);  else  return Find(key, node\_x->right);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Clear(RB\_TREE\_Node\* tree)  {  if (tree == NIL)  return;  else  {  if (tree->left)  Clear(tree->left);  if (tree->right)  Clear(tree->right);  }  delete tree;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Get\_Keys(RB\_TREE\_Node\* node\_x, LinkedList<NODE\_TYPE\_1>\* List\_of\_Keys)  {  if (node\_x->Get\_Parent() == NULL)  throw "RB Tree is empty";  if (node\_x == NIL)  return;  else  {  List\_of\_Keys->push\_back(node\_x->key);  if (node\_x->left != NIL)  Get\_Keys(node\_x->left, List\_of\_Keys);  }  if (node\_x->right != NIL)  Get\_Keys(node\_x->right, List\_of\_Keys);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Get\_Values(RB\_TREE\_Node\* node\_x, LinkedList<NODE\_TYPE\_2>\* List\_of\_Values)  {  if (node\_x->Get\_Parent() == NULL)  throw "RB Tree is empty";  if (node\_x == NIL)  return;  else  {  List\_of\_Values->push\_back(node\_x->data);  if (node\_x->left != NIL)  Get\_Values(node\_x->left, List\_of\_Values);  }  if (node\_x->right != NIL)  Get\_Values(node\_x->right, List\_of\_Values);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Print(RB\_TREE\_Node\* node\_x)  {  if (node\_x->Get\_Parent() == NULL)  throw "RB Tree is empty";  if (node\_x == NIL)  return;  else  {  std::cout << "Key = " << node\_x->key << ", data = " << node\_x->data << std::endl;  if (node\_x->left != NIL)  Print(node\_x->left);  }  if (node\_x->right != NIL)  Print(node\_x->right);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>\* RB\_TREE\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Tree\_Minimum(RB\_TREE\_Node\* node\_x)  {  if (node\_x->Get\_Parent() == NULL)  throw "RB Tree is empty";  while (node\_x->left != NIL)  node\_x = node\_x->left;  return node\_x;  } |
| **Decode.h** |
| #pragma once  #include <iostream>  #include "LinkedList.h"  #include "Tree.h"  #include <string>  #include <iomanip>  using namespace std;  template<class ARRAY\_TYPE\_1, class ARRAY\_TYPE\_2>  void Bubble\_Sort(ARRAY\_TYPE\_1\* array\_1, ARRAY\_TYPE\_2\* array\_2, int Array\_Size)  {  ARRAY\_TYPE\_1 temp\_1;  ARRAY\_TYPE\_2 temp\_2;  for (int i = 0; i < Array\_Size; ++i)  {  for (int j = i + 1; j < Array\_Size; ++j)  {  if (array\_1[i] < array\_1[j])  {  temp\_1 = array\_1[i];  array\_1[i] = array\_1[j];  array\_1[j] = temp\_1;  temp\_2 = array\_2[i];  array\_2[i] = array\_2[j];  array\_2[j] = temp\_2;  }  }  }  }  template<class ARRAY\_TYPE>  unsigned Search\_Border(ARRAY\_TYPE\* array, int start, int end)  {  unsigned from\_begin = 0, from\_end = 0;  int j = end - 1;  for (int i = start; ; i++)  {  if (i == j)  {  return i;  }  from\_begin += array[i];  while (true)  {  if (from\_end < from\_begin)  from\_end += array[j--];  else  break;  if (i == j)  {  return i;  }  }  if (i == j)  {  return i;  }  }  }  void Shannon\_Fano(unsigned\* freq, char\* sym, string& Branch, string& Full\_Branch, int start, int end, RB\_TREE\_Node<char, string>\* Symbols\_Code, unsigned& Memory\_Size\_After)  {  unsigned m;  string T\_Branch = "";  T\_Branch = Full\_Branch + Branch;  if (start == end)  {  Memory\_Size\_After += T\_Branch.length() \* freq[start];  Symbols\_Code->Insert(sym[start], T\_Branch);  cout << sym[start] << "\t" << freq[start] << "\t\t" << T\_Branch << endl;  return;  }  m = Search\_Border(freq, start, end);  string zero = "0", one = "1";  Shannon\_Fano(freq, sym, one, T\_Branch, start, m, Symbols\_Code, Memory\_Size\_After);  Shannon\_Fano(freq, sym, zero, T\_Branch, m + 1, end, Symbols\_Code, Memory\_Size\_After);  }  void Decoding(string& str)  {    cout << "Input string/text: " << str << "\n\n";  unsigned Memory\_Size\_Before = str.length() \* 8, Memory\_Size\_After = 0;  RB\_TREE\_Node<char, unsigned> Frequency;  Frequency.NIL\_N\_Root();  for (int i = 0; i < str.length(); ++i)  {  try  {  Frequency.Insert(str[i], 1);  }  catch (int ex)  {  if (ex == -1)  Frequency[str[i]]++;  }  }  unsigned Array\_Size = Frequency.Get\_Size();  unsigned\* A = new unsigned[Array\_Size];  char\* B = new char[Array\_Size];  Frequency.Fill\_Array(Frequency.Get\_Root(), A, "data");  Frequency.Fill\_Array(Frequency.Get\_Root(), B, "key");  Bubble\_Sort(A, B, Array\_Size);  string zero = "";  RB\_TREE\_Node<char, string> Symbols\_Code;  Symbols\_Code.NIL\_N\_Root();  setlocale(LC\_ALL, "Russian");  cout << "Символ:\tЧастота:\tКод:\n";  Shannon\_Fano(A, B, zero, zero, 0, Array\_Size - 1, &Symbols\_Code, Memory\_Size\_After);  cout << "\n Oбъем памяти, который занимают исходные строки: " << Memory\_Size\_Before << "\n Объем памяти, который занимают закодированные строки: " << Memory\_Size\_After << endl;  cout << " Коэффициент сжатия: " << round((1 - ((double)Memory\_Size\_After / (double)Memory\_Size\_Before)) \* 100) << "%\n";  string Encoding\_Result = "";  for (int i = 0; i < str.length(); i++)  {  Encoding\_Result += Symbols\_Code[str[i]] + " ";  }  cout << "\n Результат кодирования:\n " << Encoding\_Result << endl;  RB\_TREE\_Node<string, char> Codes\_Symbol;  Codes\_Symbol.NIL\_N\_Root();  for (int i = 0; i < Array\_Size; ++i)  {  Codes\_Symbol.Insert(Symbols\_Code[B[i]], B[i]);  }  string Decoding\_Result = "", Code;  for (int i = 0; i < Encoding\_Result.length(); i++)  {  Code = "";  while (true)  {  if (Encoding\_Result[i] != ' ')  Code += Encoding\_Result[i++];  else  break;  }  Decoding\_Result += Codes\_Symbol[Code];  }  cout << "\n Результат декодирования:\n " << Decoding\_Result << endl;  delete[] A;  delete[] B;  } |
| **UnitTest\_2.cpp** |
| #include "pch.h"  #include "CppUnitTest.h"  #include "..\Shannon-Fano\LinkedList.h"  #include "..\Shannon-Fano\Tree.h"  #include "..\Shannon-Fano\Decode.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest\_2  {  TEST\_CLASS(UnitTest\_2)  {  public:  TEST\_METHOD(Bubble\_Sorting\_1)  {  int\* temp1 = new int[5];  double\* temp2 = new double[5]{ 1.01,1.02,1.03,1.04,1.05 };  double\* check = new double[5]{ 1.05,1.04,1.03,1.02,1.01 };  for (int i = 0; i < 5; i++)  temp1[i] = double(i);  Bubble\_Sort(temp1, temp2, 5);  for (int i = 0; i < 5; ++i)  {  Assert::IsTrue(temp1[i] == double(4.0 - i));  Assert::IsTrue(temp2[i] == check[i]);  }  }  TEST\_METHOD(Bubble\_Sorting\_2)  {  unsigned\* temp1 = new unsigned[10]{ 1,2,3,4,5,6,7,8,9,10 };  char\* temp2 = new char[10]{ 'a', 'b', 'c', 'd', 'e','f','g','h','i','j' };  unsigned\* temp1\_check = new unsigned[10]{ 10,9,8,7,6,5,4,3,2,1 };  char\* temp2\_check = new char[10]{ 'j','i','h','g','f','e', 'd', 'c', 'b', 'a' };  Bubble\_Sort(temp1, temp2, 10);  for (int i = 0; i < 10; ++i)  {  Assert::AreEqual(temp1[i], temp1\_check[i]);  Assert::AreEqual(temp2[i], temp2\_check[i]);  }  }  TEST\_METHOD(Border\_Search\_1)  {  int\* temp = new int[5]{ 100,40,30,20,10 };  int border = Search\_Border(temp, 0, 5);  Assert::AreEqual(border, 0);  }  TEST\_METHOD(Border\_Search\_2)  {  int\* temp = new int[7]{ 10,6,5,4,3,2,1 };  int border = Search\_Border(temp, 0, 7);  Assert::AreEqual(border, 1);  }  TEST\_METHOD(Algorithm\_Fano)  {  string str = "Happy Holidays!";  unsigned Memory\_Size\_Before = str.length() \* 8, Memory\_Size\_After = 0;  RB\_TREE\_Node<char, unsigned> Frequency;  Frequency.NIL\_N\_Root();  for (int i = 0; i < str.length(); ++i)  {  try  {  Frequency.Insert(str[i], 1);  }  catch (int ex)  {  if (ex == -1)  Frequency[str[i]]++;  }  }  unsigned Array\_Size = Frequency.Get\_Size();  unsigned\* temp1 = new unsigned[Array\_Size];  char\* temp2 = new char[Array\_Size];  Frequency.Fill\_Array(Frequency.Get\_Root(), temp1, "data");  Frequency.Fill\_Array(Frequency.Get\_Root(), temp2, "key");  Bubble\_Sort(temp1, temp2, Array\_Size);  string zero = "";  RB\_TREE\_Node<char, string> Symbols\_Code;  Symbols\_Code.NIL\_N\_Root();  Shannon\_Fano(temp1, temp2, zero, zero, 0, Array\_Size - 1, &Symbols\_Code, Memory\_Size\_After);  RB\_TREE\_Node<char, string> Symbols\_Code\_Check;  Symbols\_Code\_Check.NIL\_N\_Root();  Symbols\_Code\_Check.Insert('a', "111");  Symbols\_Code\_Check.Insert('H', "110");  Symbols\_Code\_Check.Insert('p', "101");  Symbols\_Code\_Check.Insert('y', "100");  Symbols\_Code\_Check.Insert('l', "011");  Symbols\_Code\_Check.Insert('i', "0101");  Symbols\_Code\_Check.Insert('d', "0100");  Symbols\_Code\_Check.Insert('!', "0011");  Symbols\_Code\_Check.Insert('o', "0010");  Symbols\_Code\_Check.Insert(' ', "0001");  Symbols\_Code\_Check.Insert('s', "0000");  char keys[] = { 'a', 'H', 'p', 'y' , 'l', 'i', 'd', '!', 'o', ' ','s' };  Assert::AreEqual(Memory\_Size\_Before, (unsigned)120);  Assert::AreEqual(Memory\_Size\_After, (unsigned)51);  for (int i = 0; i < 10; ++i)  Assert::AreEqual(Symbols\_Code[keys[i]], Symbols\_Code\_Check[keys[i]]);  delete[] temp1;  delete[] temp2;  }  };  } |