**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

**ОТЧЁТ**

**по лабораторной работе №2**

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

**Тема: «Алгоритмы кодирования»**

**Вариант 1.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Горбанев А.В |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

**Постановка задачи и описание реализуемого класса и методов.**

Задание: Реализовать кодирование и декодирование по алгоритму Хаффмана входной строки, вводимой через консоль.

Для этого мне понадобились классы “RBl\_Tree”, “List”,”Queue”,"HaffmanTree".

Оценка временной сложности каждого метода:

|  |  |
| --- | --- |
| Функция | Сложность |
| Insert(key,data) | O(log(n)) |
| Remove(key) | O(log(n)) |
| Find(key) | O(log(n)) |
| Clear | O(n) |
| Get\_keys | O(n) |
| Get\_values | O(n) |
| print | O(n) |

**Описание реализованных unit-тестов.**

|  |  |
| --- | --- |
| Проверка методов класса «Tree\_Node» | |
| Название теста | Что проверяет |
| RLK\_Test\_plus\_toRbTree\_Test | Проверка кодирования |
| decode\_Test | Проверка декодирования |

**Результат выполнения всех unit-тестов**



**Пример работы программы**



**Листинг**

|  |
| --- |
| List.h |
| #ifndef LIST\_H  #define LIST\_H  #include <string>  using namespace std;  template<class T>  class list {  private:  class nodeList;  nodeList\* head;  int size;  public:  list();  ~list();  T pop();  T pop(int);  void push(T value);  int getSize();  void sort();  string getString();  };  template <class T>  class list<T>::nodeList {  private:  T value;  nodeList\* pNext;  public:  nodeList() : pNext(nullptr) {};  nodeList(T \_value, nodeList\* next) :  value(\_value), pNext(next) {}  T getValue() {  return value;  }  void setValue(T \_value) {  value = \_value;  }  nodeList\* getNext() {  return pNext;  }  void setNext(nodeList\* next) {  pNext = next;  }  };  template <class T>  list<T>::list() : head(nullptr), size(0) {}  template<class T>  list<T>::~list()  {  while (size > 0)  this->pop();  }  template <class T>  T list<T>::pop() {  if (size == 0)  return 0;  if (size == 1) {  T tempNextVal = head->getValue();  delete head;  head = nullptr;  this->size = 0;  return tempNextVal;  }  nodeList\* temp = head;  while (temp->getNext()->getNext() != nullptr) temp = temp->getNext();  T tempNextVal = temp->getNext()->getValue();  delete temp->getNext();  size--;  temp->setNext(nullptr);  return tempNextVal;  }  template <class T>  T list<T>::pop(int index) {  if (size == 1 || size == 0 || index == size - 1) {  return this->pop();  }  if (index == 0) {  nodeList\* temp = head->getNext();  T tempvalue = head->getValue();  head = temp;  size--;  return tempvalue;  }  nodeList\* temp = head;  for (int i = 0; i < index - 1; i++)  temp = temp->getNext();  T tempNextVal = temp->getNext()->getValue();  nodeList\* tempNext = temp->getNext()->getNext();  size--;  temp->setNext(tempNext);  return tempNextVal;  }  template <class T>  void list<T>::push(T value) {  if (size == 0) {  head = new nodeList(value, nullptr);  size++;  return;  }  nodeList\* temp = head;  while (temp->getNext() != nullptr) temp = temp->getNext();  temp->setNext(new nodeList(value, nullptr));  size++;  return;  }  template <class T>  int list<T>::getSize() {  return size;  }  template <class T>  void list<T>::sort() {  if (size == 1 || size == 0)  return;  nodeList\* temp = head;  bool f = false;  for (int i = 0; i < size - 1; i++) {  temp = head;  f = false;  for (int j = 0; j < size - 1 - i; j++) {  if (temp->getValue()->getPower() > temp->getNext()->getValue()->getPower()) {  T tempVal = temp->getValue();  temp->setValue(temp->getNext()->getValue());  temp->getNext()->setValue(tempVal);  f = true;  }  temp = temp->getNext();  }  if (!f)  return;  }  }  template <class T>  string list<T>::getString() {  nodeList\* temp = head;  string out = "";  while (temp != NULL) {  out += temp->getValue() + '0';  temp = temp->getNext();  }  return out;  }  #endif |
| RBTree.h |
| #include <iostream>  #include <string>  #include "queue.h"  #include "list.h"  using namespace std;  template <class T\_key, class T\_value>  class RBTree {  class Node;  protected:  Node\* nil;  Node\* root;  private:  enum class Color {  Black,  Red  };  class Node {  public:  Color color;  T\_key key;  Node\* pLeft;  Node\* pRight;  Node\* parent;  T\_value value;  Node() : color(Color::Black), key(0),  pLeft(nullptr), pRight(nullptr),  parent(nullptr) {}  Node(Node\* node) : color(node->color), key(node->key),  pLeft(node->pLeft), pRight(node->pRight),  parent(node->parent), value(node->value) {}  Node(Color \_color, T\_key \_key, Node\* left, Node\* right, Node\* \_parent, T\_value \_value) :  color(\_color), key(\_key),  pLeft(left), pRight(right),  parent(\_parent), value(\_value) {}  };  void leftRotate(Node\* x) {  Node\* y = x->pRight;  x->pRight = y->pLeft;  if (y->pLeft != nil)  y->pLeft->parent = x;  if (y != nil)  y->parent = x->parent;  if (x->parent == nil)  root = y;  else if (x == x->parent->pLeft)  x->parent->pLeft = y;  else  x->parent->pRight = y;  y->pLeft = x;  if (x != nil)  x->parent = y;  }  void rightRotate(Node\* y) {  Node\* x = y->pLeft;  y->pLeft = x->pRight;  if (x->pRight != nil)  x->pRight->parent = y;  if (x != nil)  x->parent = y->parent;  if (y->parent == nil)  root = x;  else if (y == y->parent->pRight)  y->parent->pRight = x;  else  y->parent->pLeft = x;  x->pRight = y;  if (y != nil)  y->parent = x;  }  void rbInsertFixup(Node\* z) {  while (z != root && z->parent->color == Color::Red) {  if (z->parent == z->parent->parent->pLeft) {  Node\* y = z->parent->parent->pRight;  if (y->color == Color::Red) {  z->parent->color = Color::Black;  y->color = Color::Black;  z->parent->parent->color = Color::Red;  z = z->parent->parent;  }  else {  if (z == z->parent->pRight) {  z = z->parent;  leftRotate(z);  }  z->parent->color = Color::Black;  z->parent->parent->color = Color::Red;  rightRotate(z->parent->parent);  }  }  else {  Node\* y = z->parent->parent->pLeft;  if (y->color == Color::Red) {  z->parent->color = Color::Black;  y->color = Color::Black;  z->parent->parent->color = Color::Red;  z = z->parent->parent;  }  else {  if (z == z->parent->pLeft) {  z = z->parent;  rightRotate(z);  }  z->parent->color = Color::Black;  z->parent->parent->color = Color::Red;  leftRotate(z->parent->parent);  }  }  }  root->color = Color::Black;  }  void rbInsert(Node\* z) {  Node\* y = nil;  Node\* x = root;  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->pLeft;  else  x = x->pRight;  }  z->parent = y;  if (y == nil)  root = z;  else if (z->key < y->key)  y->pLeft = z;  else  y->pRight = z;  z->pLeft = nil;  z->pRight = nil;  z->color = Color::Red;  rbInsertFixup(z);  }  void deleteFixup(Node\* x) {  while (x != root && x->color == Color::Black) {  if (x == x->parent->pLeft) {  Node\* w = x->parent->pRight;  if (w->color == Color::Red) {  w->color = Color::Black;  x->parent->color = Color::Red;  leftRotate(x->parent);  w = x->parent->pRight;  }  if (w->pLeft->color == Color::Black && w->pRight->color == Color::Black) {  w->color = Color::Red;  x = x->parent;  }  else {  if (w->pRight->color == Color::Black) {  w->pLeft->color = Color::Black;  w->color = Color::Red;  rightRotate(w);  w = x->parent->pRight;  }  w->color = x->parent->color;  x->parent->color = Color::Black;  w->pRight->color = Color::Black;  leftRotate(x->parent);  x = root;  }  }  else {  Node\* w = x->parent->pLeft;  if (w->color == Color::Red) {  w->color = Color::Black;  x->parent->color = Color::Red;  rightRotate(x->parent);  w = x->parent->pLeft;  }  if (w->pRight->color == Color::Black && w->pLeft->color == Color::Black) {  w->color = Color::Red;  x = x->parent;  }  else {  if (w->pLeft->color == Color::Black) {  w->pRight->color = Color::Black;  w->color = Color::Red;  leftRotate(w);  w = x->parent->pLeft;  }  w->color = x->parent->color;  x->parent->color = Color::Black;  w->pLeft->color = Color::Black;  rightRotate(x->parent);  x = root;  }  }  }  x->color = Color::Black;  }  void deleteNode(Node\* z) {  Node\* x, \* y;  if (z == nil)  return;  if (z->pLeft == nil || z->pRight == nil)  y = z;  else {  y = z->pRight;  while (y->pLeft != nil) y = y->pLeft;  }  if (y->pLeft != nil)  x = y->pLeft;  else  x = y->pRight;  x->parent = y->parent;  if (y->parent != nil) {  if (y == y->parent->pLeft)  y->parent->pLeft = x;  else  y->parent->pRight = x;  }  else  root = x;  if (y != z) {  z->key = y->key;  z->value = y->value;  }  if (y->color == Color::Black)  deleteFixup(x);  delete y;  }  Node\* findNode(T\_key key) {  Node\* it = root;  while (it->key != key && it != nil) {  if (it->key < key)  it = it->pRight;  else  it = it->pLeft;  }  if (it == nil)  return(nullptr);  return it;  }  void recursionForClear(Node\* x) {  if (x->PLeft != nil)  recursionForClear(x->pLeft);  if (x->pRight != nil)  recursionForClear(x->pRight);  delete x;  }  public:  class iterator : public RBTree {  private:  queue<Node\*> queueIt;  T\_value value;  T\_key key;  Node\* it;  Node\* nil;  public:  iterator(RBTree\* tree) : it(NULL),  value(tree->root->value), key(tree->root->key), nil(tree->nil) {  queueIt.push(tree->root);  }  const T\_value& operator++() {  if (queueIt.getSize() == 0)  return NULL;  it = queueIt.front();  queueIt.pop();  if (it->pLeft != nil)  queueIt.push(it->pLeft);  if (it->pRight != nil)  queueIt.push(it->pRight);  value = it->value;  key = it->key;  return it->value;  }  const T\_key& operator\*() {  return key;  }  const T\_value& getValue() {  return value;  }  };  RBTree() {  nil = new Node();  root = nil;  }  void insert(T\_key \_key, T\_value \_value) {  Node\* node = new Node(Color::Red, \_key, nullptr, nullptr, nullptr, \_value);  rbInsert(node);  }  T\_value find(T\_key \_key) {  if (findNode(\_key))  return findNode(\_key)->value;  return 0;  }  void remove(T\_key \_key) {  deleteNode(findNode(\_key));  }  void clear() {  recursionForClear(root);  root = nil;  }  list<T\_key> get\_keys() {  list<T\_key> out;  iterator it(this);  while (++it != NULL) {  out.push(\*it);  }  return out;  }  list<T\_value> get\_values() {  list<T\_value> out;  iterator it(this);  while (++it != NULL) {  out.push(it.getValue());  }  return out;  }  void print() {  iterator it(this);  cout << '\n';  while (++it != NULL) {  cout << \*it << " ";  }  }  int getSize() {  int i = 0;  iterator it(this);  while (++it != NULL) {  i++;  }  return i;  }  };  #pragma once |
| 2labs.cpp |
| #include "list.h"  #include "queue.h"  #include "RBTree.h"  #include "HaffmanTree.h"  #include <string>  #include <iostream>  using namespace std;  string to\_binary\_string(char n)  {  string result;  do  {  result += ('0' + (n % 2));  n = n / 2;  } while (n > 0);  return result;  }  int main() {  string input;  list<HaffmanTree\*> listOfHaffmanTrees;  RBTree<char, int> rbtree;  cin >> input;  int tempVal;  for (int i = 0; i < input.length(); i++) {  if (rbtree.find(input[i]) == 0) {  rbtree.insert(input[i], 1);  }  else {  tempVal = rbtree.find(input[i]);  rbtree.remove(input[i]);  rbtree.insert(input[i], tempVal + 1);  }  }  RBTree<char, int>::iterator it(&rbtree);  ++it;  cout << "symbol rate: \n";  for (int i = 0; i < rbtree.getSize(); i++) {  listOfHaffmanTrees.push(new HaffmanTree(\*it, it.getValue()));  cout << \*it << " " << it.getValue() << endl;  ++it;  }  while (listOfHaffmanTrees.getSize() != 1) {  listOfHaffmanTrees.sort();  listOfHaffmanTrees.push(new HaffmanTree(listOfHaffmanTrees.pop(0), listOfHaffmanTrees.pop(1)));  }  HaffmanTree\* haffmanTree = listOfHaffmanTrees.pop(0);    cout << "First string: ";  for (int i = 0; i < input.length(); i++) {  for (int j = 0; j < (8 - to\_binary\_string(input[i]).length()); j++)  cout << "0";  cout << to\_binary\_string(input[i]) + " ";  }  haffmanTree->RLK(haffmanTree->getRoot());  RBTree<char, string> haffmanTable;  haffmanTree->toRbTree(&haffmanTable, haffmanTree->getRoot());  cout << "\n\nstring weight: " << 8 \* input.length() << " bits";  string encodeInput = "";  cout << "\n\nSecond string: ";  for (int i = 0; i < input.length(); i++) {  encodeInput += haffmanTable.find(input[i]);  cout << haffmanTable.find(input[i]);  cout << " ";  }  cout << "\n\nencode weight: " << encodeInput.length() << " bits";    cout << "\n\nDecode string: ";  cout << haffmanTree->decode(encodeInput);    cout << "\n\nCompression ratio: ";  cout << (float(encodeInput.length()) / float((8 \* input.length()))) \* 100 << " %\n";  return 0;  } |
| queue.h |
| #ifndef QUEUE\_H  #define QUEUE\_H  using namespace std;  template<class T>  class queue {  private:  class nodeQueue;  nodeQueue\* head;  nodeQueue\* tail;  int size;  public:  queue();  ~queue();  T pop();  void push(T value);  int getSize();  T front();  };  template <class T>  class queue<T>::nodeQueue {  private:  T value;  nodeQueue\* pNext;  public:  nodeQueue() : pNext(nullptr) {};  nodeQueue(T \_value, nodeQueue\* next) :  value(\_value), pNext(next) {}  T getValue() {  return value;  }  void setValue(T \_value) {  value = \_value;  }  nodeQueue\* getNext() {  return pNext;  }  void setNext(nodeQueue\* next) {  pNext = next;  }  };  template <class T>  queue<T>::queue() : head(nullptr), tail(nullptr), size(0) {}  template<class T>  queue<T>::~queue()  {  while (size > 0)  this->pop();  }  template <class T>  T queue<T>::pop() {  if (size == 0)  throw "queue error";  nodeQueue\* temp = head;  T tempVal = head->getValue();  head = head->getNext();  delete temp;  size--;  if (head == nullptr)  tail = head;  return tempVal;  }  template <class T>  void queue<T>::push(T value) {  if (size == 0) {  head = new nodeQueue(value, nullptr);  tail = head;  size++;  return;  }  tail->setNext(new nodeQueue(value, nullptr));  tail = tail->getNext();  size++;  return;  }  template <class T>  int queue<T>::getSize() {  return size;  }  template <class T>  T queue<T>::front() {  return this->head->getValue();  }  #endif |
| **HaffmanTree.h** |
| #ifndef HAFFMAN\_H  #define HAFFMAN\_H  #include <stdlib.h>  #include <string>  #include "list.h"  #include "queue.h"  #include "RBTree.h"  using namespace std;  class HaffmanTree {  private:  class nodeHaffman {  public:  nodeHaffman\* parent;  nodeHaffman\* pLeft;  nodeHaffman\* pRight;  string code;  char value;  nodeHaffman(nodeHaffman\* parent, nodeHaffman\* left, nodeHaffman\* right, char value) :  parent(parent), pLeft(left), pRight(right), value(value), code("") {};  void setParent(nodeHaffman\* parent) {  parent = parent;  }  void setCode(string code) {  code = code;  }  };  int power;  nodeHaffman\* root;  public:  nodeHaffman\* getRoot() {  return root;  }  int getPower() {  return power;  }  HaffmanTree(char value, int power) :  power(power) {  root = new nodeHaffman(nullptr, nullptr, nullptr, value);  }  HaffmanTree(HaffmanTree\* leftTree, HaffmanTree\* rightTree) {  power = leftTree->getPower() + rightTree->getPower();  root = new nodeHaffman(nullptr, leftTree->getRoot(), rightTree->getRoot(), NULL);  root->pLeft->setParent(root);  root->pRight->setParent(root);  }  void RLK(nodeHaffman\* node)  {  if (node != nullptr)  {  if (node->pRight != nullptr) {  node->pRight->code = node->code + "1";  RLK(node->pRight);  }  if (node->pLeft != nullptr) {  node->pLeft->code = node->code + "0";  RLK(node->pLeft);  }  }  }  void toRbTree(RBTree<char, string>\* tree, nodeHaffman\* node)  {  if (node != nullptr)  {  if (node->pRight != nullptr) {  toRbTree(tree, node->pRight);  }  if (node->pLeft != nullptr) {  toRbTree(tree, node->pLeft);  }  if (node->value != 0)  tree->insert(node->value, node->code);  }  }  string decode(string encodeInput) {  string out = "";  nodeHaffman\* node = root;  for (int i = 0; i < encodeInput.length(); i++) {  if (encodeInput[i] == '0')  node = node->pLeft;  else if (encodeInput[i] == '1')  node = node->pRight;  if (node->value != 0) {  out += node->value;  node = root;  }  }  return out;  }  };  #endif |

Вывод:

В ходе лабораторной работы научился кодировать и декодировать строку с помощью алгоритма Хаффмана.