**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 (1 вариант)

|  |  |  |
| --- | --- | --- |
| Студент гр. 8309 |  | Ильин Д.Д. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**Исходная формулировка задания:**

Реализовать кодирование Хаффмана с использованием самостоятельно написанных структур данных

**Цель работы:**

Научиться реализовывать код Хаффмана.

**Описание реализуемых и вспомогательных классов**

Класс Tree\_Hoffman, содержит вложенный класс Node, который в свою очередь имеет три публичных поля: указатели на Node\*Left (Указатель на левый элемент), Node\*Right (Указатель на правый элемент) и переменную value класса Pair (вспомогательный класс состоящий из двух публичных шаблонных полей). А сам класс имеет поля Node\* Top (вершина дерева) и List<Pair<char,int>>\* list\_symbol (список символов в дереве).Сам класс подразумевает собой дерево кодирования построенное на основе алгоритма кодирования Хаффмана. Класс содержит следующие методы:

* Конструктор – который получает на вход строку, и строит список типа Node символов и их частоты появления в строке. Потом этот список передаётся в функцию *build\_tree*.
* Деструктор – реализован деструктор, который вызывает метод clear (на основе обычного удаления двоичного дерева).
* *void build\_tree (List<Node>\*& list\_for\_build\_tree)* – функция построения дерева кодирования. Само добавление проходит по следующему алгоритму: сортируем список в порядке возрастания, берём первые два элемента, удаляем их из списка и создаем дерево, корнем которого становится элемент с суммой частот этих элементов и пустым полем, а листьями становятся элементы, частоты которых складывались: слева с меньшей частотой, справа – с большей. Вставляем это дерево в список так, чтобы соблюдался порядок возрастания. И так далее в цикле, пока список не будет состоять из одного элемента.
* Map<char,string>\*& get\_tree\_with\_code() – функция получения ассоциативного массива символов и их кодов. Для этого вызывается функция *fill\_tree\_with\_code*
* void fill\_tree\_with\_code (Map<char, string>\* hoffman, Node\* root, string cur) – функция заполнения ассоциативного массива, ключом которого является символ, присутствующий в строке, а значением код символа (каждый символ – это элемент, не имеющий сыновей). Методом Хаффмана мы определяем код каждого элемента и записываем его в ассоциативный массив.
* List<Pair<char, int>>\* get\_list\_symbol () – функция получения списка пар символов и их частот появления в строке.

String Decoding\_hoffman\_tree (string& coding\_str) – функция декодирования закодированной строки. Мы идём по закодированной строке и, если встречаем 0, то идём влево, иначе – вправо, и, если встречаем символ, то возвращаемся в корень, не меняя позиции продвижения по строке, тем самым декодируем закодированную строку.

**Оценка временной сложности**

* *build\_tree(List<Node>\*& list\_for\_build\_tree)* – O(NlogN)
* *get\_list\_symbol()* – O(1)
* *get\_tree\_with\_code()* – O(N)
* *fill\_tree\_with\_code(Map<char, string>\* hoffman, Node\* root, string cur)* – O(N)
* *string Decoding\_hoffman\_tree(string& coding\_str)* – O(NlogN)

**Описание реализованных unit-тестов**

Все юнит-тесты выполнены с помощью класса Assert и его функции AreEqual.

-Test\_count – проверка правильности подсчета частот букв в заданной строке

-Test\_count2 – проверка правильности подсчета частот букв в заданной строке, 2 случай.

-Test\_check – проверка правильности работы функции, которая определяет нахождения элемента в строке.

-Test\_tree – проверка правильности работы функции построения очереди из элементов дерева.

-Test\_tree2 – проверка правильности работы функции построения очереди из элементов дерева во 2 случае.

-Test\_sort – проверка правильности работы функции преобразования строки (в строку, в которой каждый элемент встречается по 1 разу)

-Test\_coding – проверка правильности работы функции построение дерева Хаффмана.

-Test\_get – проверка правильности работы обхода по листьям дерева.

-reverse – проверка правильности работы функции реверсирования строки.

-Test\_get2 – проверка правильности работы обхода по всем элементам дерева.

-Test\_final – проверка построения строки из кодов букв исходной строки.

-Remove\_5 – проверка пятого случая удаления

- Find\_1 – проверка метода поиска элемента.

-Find-exception – проверка корректности работы исключения в функции.

-Clear – создается дерево, потом вызывается функция очистки, проверяется корректность работы.

-get\_keys – создается дерево, потом вызывается тестируемая функция и проверяется правильность списка, который вернула функция.

-get\_values – создается дерево, потом вызывается тестируемая функция и проверяется правильность списка, который вернула функция.

**Программа**

**laba4.2.cpp**

#include"iostream"

#include"Map.h"

#include"string"

#include"Hoffman.h"

using namespace std;

int main() {

string str;

string coding\_str;

cout << "Input: ";

getline(cin, str);

Tree\_Hoffman\* Tree\_Hoffman\_tree = new Tree\_Hoffman(str);

Map<char, string>\* hoffman = Tree\_Hoffman\_tree->get\_tree\_with\_code();

int counter = 0;

List<Pair<char, int>>\* list\_symbol = Tree\_Hoffman\_tree->get\_list\_symbol();

for (int i = 0; i < list\_symbol->get\_size(); i++)

cout << '{' << list\_symbol->at(i).first << '}' << "==" << '{' << hoffman->find(list\_symbol->at(i).first) << "} with frequency: " << list\_symbol->at(i).second << endl;

cout << "old string: " << str << endl;

cout << "new string: ";

for (int i = 0; i < str.size(); i++) {

coding\_str += hoffman->find(str[i]);

cout << hoffman->find(str[i]);

counter += hoffman->find(str[i]).size();

}

cout << endl << "number of characters : " << counter << endl;

string decoding\_str = Tree\_Hoffman\_tree->Decoding\_hoffman\_tree(coding\_str);

cout << "Compression ratio: x: " << (((double)(decoding\_str.size() \* 8) / (coding\_str.size())));

delete Tree\_Hoffman\_tree;

return 0;

}

**Hoffman.h**

#pragma once

#include"List.h"

#include"Map.h"

#include"Pair.h"

#include<string>

class Tree\_Hoffman

{

class Node {

public:

Node(Pair<char, int> value = Pair<char, int>(), Node\* left = NULL, Node\* right = NULL) :value(value), left(left), right(right) {}

Pair<char, int> value;

Node\* left;

Node\* right;

};

public:

~Tree\_Hoffman() {

this->clear\_tree(Top);

}

Tree\_Hoffman(string str) {

Map<char, int>\* map\_symbol = new Map<char, int>();

list\_symbol = new List<Pair<char, int>>();

for (int i = 0; i < str.size(); i++) {

if (!map\_symbol->find\_is(str[i]))//if the symbol is not in the list then add the value to the symbol map otherwise increase the number

map\_symbol->insert(str[i], 1);

else

map\_symbol->increment\_value(str[i]);

}

list\_symbol = map\_symbol->get\_pairs();

map\_symbol->clear();

list\_symbol->sort();

///////////////////////////////////////////////////////////////////////////building the encoding tree

List<Node>\* list\_for\_build\_tree = new List<Node>();

for (int i = 0; i < list\_symbol->get\_size(); i++)

list\_for\_build\_tree->push\_back(Node(list\_symbol->at(i)));

build\_tree(list\_for\_build\_tree);

///////////////////////////////////////////////////////////////////////////

}

List<Pair<char, int>>\* get\_list\_symbol() {

return list\_symbol;

}

Map<char, string>\*& get\_tree\_with\_code() {

Map<char, string>\* hoffman = new Map<char, string>();

string cur;

fill\_tree\_with\_code(hoffman, Top, cur);

return hoffman;

}

string Decoding\_hoffman\_tree(string& coding\_str) {

string decoding\_str;

int pos = 0;

Decoding(Top, coding\_str, decoding\_str, pos);

return decoding\_str;

}

private:

void Decoding(Node\* root, string& coding\_str, string& decoding\_str, int& position) {

if (coding\_str.size() > position) {

while (root->right != NULL && root->left != NULL) {

if (coding\_str[position] == '0')

root = root->left;

else

root = root->right;

position++;

}

decoding\_str += root->value.first;

if (Top->left == NULL && Top->right == NULL)

position++;

Decoding(Top, coding\_str, decoding\_str, position);

}

}

void build\_tree(List<Node>\*& list\_for\_build\_tree) {

Top = NULL;

if (list\_for\_build\_tree->get\_size() > 1) {

while (list\_for\_build\_tree->get\_size() != 0) {

Node\* cur = new Node();

cur->left = new Node(list\_for\_build\_tree->at(0));

cur->right = new Node(list\_for\_build\_tree->at(1));

cur->value.second = list\_for\_build\_tree->at(0).value.second + list\_for\_build\_tree->at(1).value.second;

list\_for\_build\_tree->pop\_front();

list\_for\_build\_tree->pop\_front();

int i = 0;

for (; i < list\_for\_build\_tree->get\_size() && list\_for\_build\_tree->at(i).value.second < cur->value.second; i++);

if (list\_for\_build\_tree->get\_size() != 0 && list\_for\_build\_tree->get\_size() != i)

list\_for\_build\_tree->insert(\*cur, i);

else

if (list\_for\_build\_tree->get\_size() == i && list\_for\_build\_tree->get\_size() != 0)

list\_for\_build\_tree->push\_back(\*cur);

if (list\_for\_build\_tree->get\_size() == 0) {

Top = cur;

}

}

}

else {

Top = new Node(Pair<char, int>(list\_for\_build\_tree->at(0).value.first, list\_for\_build\_tree->at(0).value.second));

}

//////////////////////////////////////////////////////////////////////////

}

void fill\_tree\_with\_code(Map<char, string>\* hoffman, Node\* root, string cur) {//filling in the code tree

if (Top->left != NULL && Top->right != NULL) {

if (root->left != NULL && root->right != NULL) {

fill\_tree\_with\_code(hoffman, root->left, cur + '0');

fill\_tree\_with\_code(hoffman, root->right, cur + '1');

}

else

{

hoffman->insert(root->value.first, cur);

}

}

else {

hoffman->insert(root->value.first, cur + '0');

}

}

void clear\_tree(Node\* tree) {

if (tree != NULL) {

clear\_tree(tree->left);

clear\_tree(tree->right);

delete tree;

}

}

Node\* Top;

List<Pair<char, int>>\* list\_symbol;

};

**Map.h**

#pragma once

#define COLOR\_RED 1

#define COLOR\_BLACK 0

#include"List.h"

#include"Pair.h"

using namespace std;

template<class T, class T1>

class Map {

public:

class Node

{

public:

Node(bool color = COLOR\_RED, T key = T(), Node\* parent = NULL, Node\* left = NULL, Node\* right = NULL, T1 value = T1()) :color(color), key(key), parent(parent), left(left), right(right), value(value) {}

T key;

T1 value;

bool color;

Node\* parent;

Node\* left;

Node\* right;

};

~Map() {

if (this->Top != NULL)

this->clear();

Top = NULL;

delete TNULL;

TNULL = NULL;

}

Map(Node\* Top = NULL, Node\* TNULL = new Node(0)) :Top(TNULL), TNULL(TNULL) {}

void increment\_value(T key) {

Node\* cur = this->find\_value(key);

cur->value++;

}

void printTree()

{

if (Top)

{

print\_Helper(this->Top, "", true);

}

else throw std::out\_of\_range("Tree is empty!");

}

void insert(T key, T1 value)

{

if (this->Top != TNULL) {

Node\* node = NULL;

Node\* parent=NULL;

/\* Search leaf for new element \*/

for (node = this->Top; node != TNULL; )

{

parent = node;

if (key < node->key)

node = node->left;

else if (key > node->key)

node = node->right;

else if (key == node->key)

throw std::out\_of\_range("key is repeated");

}

node = new Node(COLOR\_RED, key, TNULL, TNULL, TNULL, value);

node->parent = parent;

if (parent != TNULL) {

if (key < parent->key)

parent->left = node;

else

parent->right = node;

}

rbtree\_fixup\_add(node);

}

else {

this->Top = new Node(COLOR\_BLACK, key, TNULL, TNULL, TNULL, value);

}

}

List<T>\* get\_keys() {

List<T>\* list = new List<T>();

this->ListKey(Top,list);

return list;

}

List<T1>\* get\_values() {

List<T1>\* list = new List<T1>();

this->ListValue(Top, list);

return list;

}

List<Pair<T, T1>>\* get\_pairs() {

List<Pair<T, T1>>\* list = new List<Pair<T,T1>>();

this->ListPairs(Top, list);

return list;

}

T1 find(T key) {

Node\* node = Top;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node->value;

else

throw std::out\_of\_range("Key is missing");

}

bool find\_is(T key) {

Node\* node = Top;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return true;

else

return false;

}

void remove(T key) {

this->deleteNodeHelper(this->find\_key(key));

}

void clear() {

this->clear\_tree(this->Top);

this->Top = NULL;

}

private:

Node\* Top;

Node\* TNULL;

Node\* find\_value(T key) {

Node\* node = Top;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node;

}

/////////////////////////////////////////////////////////////////////////

//delete functions

////////////////////////////////////////////////////////////////////////

void deleteNodeHelper(Node\* find\_node)

{

Node\* node\_with\_fix, \* cur\_for\_change;

cur\_for\_change = find\_node;

bool cur\_for\_change\_original\_color = cur\_for\_change->color;

if (find\_node->left == TNULL)

{

node\_with\_fix = find\_node->right;

Transplant(find\_node, find\_node->right);

}

else if (find\_node->right == TNULL)

{

node\_with\_fix = find\_node->left;

Transplant(find\_node, find\_node->left);

}

else

{

cur\_for\_change = minimum(find\_node->right);

cur\_for\_change\_original\_color = cur\_for\_change->color;

node\_with\_fix = cur\_for\_change->right;

if (cur\_for\_change->parent == find\_node)

{

node\_with\_fix->parent = cur\_for\_change;

}

else

{

Transplant(cur\_for\_change, cur\_for\_change->right);

cur\_for\_change->right = find\_node->right;

cur\_for\_change->right->parent = cur\_for\_change;

}

Transplant(find\_node, cur\_for\_change);

cur\_for\_change->left = find\_node->left;

cur\_for\_change->left->parent = cur\_for\_change;

cur\_for\_change->color = find\_node->color;

}

delete find\_node;

if (cur\_for\_change\_original\_color == COLOR\_BLACK)

{

this->rbtree\_fixup\_add(node\_with\_fix);

}

}

//swap links(parent and other) for rotate

void Transplant(Node\* cur, Node\* cur1)

{

if (cur->parent == TNULL)

{

Top = cur1;

}

else if (cur == cur->parent->left)

{

cur->parent->left = cur1;

}

else

{

cur->parent->right = cur1;

}

cur1->parent = cur->parent;

}

void clear\_tree(Node\* tree) {

if (tree != TNULL) {

clear\_tree(tree->left);

clear\_tree(tree->right);

delete tree;

}

}

/////////////////////////////////////////////////////////////////////////////////////////

//all find functions

////////////////////////////////////////////////////////////////////////////////////////

Node\* minimum(Node\* node)

{

while (node->left != TNULL)

{

node = node->left;

}

return node;

}

Node\* maximum(Node\* node)

{

while (node->right != TNULL)

{

node = node->right;

}

return node;

}

Node\* grandparent(Node\* cur)

{

if ((cur != TNULL) && (cur->parent != TNULL))

return cur->parent->parent;

else

return TNULL;

}

Node\* uncle(Node\* cur)

{

Node\* cur1 = grandparent(cur);

if (cur1 == TNULL)

return TNULL; // No grandparent means no uncle

if (cur->parent == cur1->left)

return cur1->right;

else

return cur1->left;

}

Node\* sibling(Node\* n)

{

if (n == n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

Node\* find\_key(T key) {

Node\* node = this->Top;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node;

else

throw std::out\_of\_range("Key is missing");

}

/////////////////////////////////////////////////////////////////////////////////////

//all print function

/////////////////////////////////////////////////////////////////////////////////////

void print\_Helper(Node\* root, string indent, bool last)

{

if (root != TNULL)

{

cout << indent;

if (last)

{

cout << "R----";

indent += " ";

}

else

{

cout << "L----";

indent += "| ";

}

string sColor = !root->color ? "BLACK" : "RED";

cout << root->key << "(" << sColor << ")" << endl;

print\_Helper(root->left, indent, false);

print\_Helper(root->right, indent, true);

}

}

void ListValue(Node\* tree, List<T1>\* list){

if (tree != TNULL) {

ListValue(tree->left, list);

list->push\_back(tree->value);

ListValue(tree->right, list);

}

}

void ListKey(Node\* tree, List<T>\* list) {

if (tree != TNULL) {

ListKey(tree->left, list);

list->push\_back(tree->key);

ListKey(tree->right, list);

}

}

void ListPairs(Node\* tree, List<Pair<T,T1>>\* list) {

if (tree != TNULL) {

ListPairs(tree->left, list);

Pair<T, T1> cur(tree->key, tree->value);

list->push\_back(cur);

ListPairs(tree->right, list);

}

}

///////////////////////////////////////////////////////////////////////////////////////

//fix before add

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

void rbtree\_fixup\_add(Node\* node)

{

Node\* uncle;

/\* Current node is COLOR\_RED \*/

while (node != this->Top && node->parent->color == COLOR\_RED)//

{

/\* node in left tree of grandfather \*/

if (node->parent == this->grandparent(node)->left)//

{

/\* node in left tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == COLOR\_RED) {

/\* Case 1 - uncle is COLOR\_RED \*/

node->parent->color = COLOR\_BLACK;

uncle->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

node = this->grandparent(node);

}

else {

/\* Cases 2 & 3 - uncle is COLOR\_BLACK \*/

if (node == node->parent->right) {

/\*Reduce case 2 to case 3 \*/

node = node->parent;

this->left\_rotate(node);

}

/\* Case 3 \*/

node->parent->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

this->right\_rotate(this->grandparent(node));

}

}

else {

/\* Node in right tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == COLOR\_RED) {

/\* Uncle is COLOR\_RED \*/

node->parent->color = COLOR\_BLACK;

uncle->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

node = this->grandparent(node);

}

else {

/\* Uncle is COLOR\_BLACK \*/

if (node == node->parent->left) {

node = node->parent;

this->right\_rotate(node);

}

node->parent->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

this->left\_rotate(this->grandparent(node));

}

}

}

this->Top->color = COLOR\_BLACK;

}

///////////////////////////////////////////////////////////////////////////

//Rotates

////////////////////////////////////////////////////////////////////////

//(left rotate)

void left\_rotate(Node\* node)

{

Node\* right = node->right;

/\* Create node->right link \*/

node->right = right->left;

if (right->left != TNULL)

right->left->parent = node;

/\* Create right->parent link \*/

if (right != TNULL)

right->parent = node->parent;

if (node->parent != TNULL) {

if (node == node->parent->left)

node->parent->left = right;

else

node->parent->right = right;

}

else {

this->Top = right;

}

right->left = node;

if (node != TNULL)

node->parent = right;

}

//(right rotate)

void right\_rotate(Node\* node)

{

Node\* left = node->left;

/\* Create node->left link \*/

node->left = left->right;

if (left->right != TNULL)

left->right->parent = node;

/\* Create left->parent link \*/

if (left != TNULL)

left->parent = node->parent;

if (node->parent != TNULL) {

if (node == node->parent->right)

node->parent->right = left;

else

node->parent->left = left;

}

else {

this->Top = left;

}

left->right = node;

if (node != TNULL)

node->parent = left;

}

};

**List.h**

#pragma once

using namespace std;

template<class T>

class List

{

private:

class Node {

public:

Node(T data = T(), Node\* Next = NULL) {

this->data = data;

this->Next = Next;

}

Node\* Next;

T data;

};

public:

void push\_back(T obj) { // add to end bc

if (head != NULL) {

this->tail->Next = new Node(obj);

tail = tail->Next;

}

else {

this->head = new Node(obj);

this->tail = this->head;

}

Size++;

}

void push\_front(T obj) { // add to start bc

if (head != NULL) {

Node\* current = new Node;

current->data = obj;

current->Next = this->head;

this->head = current;

}

else {

this->head = new Node(obj);

}

this->Size++;

}

void pop\_back() { // delete last item bc

if (head != NULL) {

Node\* current = head;

while (current->Next != tail)//looking for the penultimate

current = current->Next;

delete tail;

tail = current;

tail->Next = NULL;

Size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void pop\_front() { // delete the first item bc-+

if (head != NULL) {

Node\* current = head;

head = head->Next;

current->Next = NULL;

delete current;

Size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void insert(T obj, size\_t k) {// adding an element by index (insertion before an element that was previously available at this index) bc

if (k >= 0 && this->Size > k) {

if (this->head != NULL) {

if (k == 0)

this->push\_front(obj);

else

{

Node\* current = new Node;// to add an item

Node\* current1 = head;// to search for the total item

for (int i = 0; i < k - 1; i++) {

current1 = current1->Next;

}

current->data = obj;

current->Next = current1->Next;// retells on the trail element

current1->Next = current;

Size++;

}

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

T at(size\_t k) {// getting an item by index bc

if (this->head != NULL && k >= 0 && k <= this->Size - 1) {

if (k == 0)

return this->head->data;

else

if (k == this->Size - 1)

return this->tail->data;

else

{

Node\* current = head;

for (int i = 0; i < k; i++) {

current = current->Next;

}

return current->data;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

void remove(int k) { // delete item by index bc

if (head != NULL && k >= 0 && k <= Size - 1) {

if (k == 0) this->pop\_front();

else

if (k == this->Size - 1) this->pop\_back();

else

if (k != 0) {

Node\* current = head;

for (int i = 0; i < k - 1; i++) {// go to pre-element

current = current->Next;

}

Node\* current1 = current->Next;

current->Next = current->Next->Next;

delete current1;

Size--;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

size\_t get\_size() { // getting list size bc

return Size;

}

void print\_to\_console() { // output of list items to the console through a separator, do not use at bc

if (this->head != NULL) {

Node\* current = head;

for (int i = 0; i < Size; i++) {

cout << current->data << ' ';

current = current->Next;

}

}

}

void clear() { // delete all list items

if (head != NULL) {

Node\* current = head;

while (head != NULL) {

current = current->Next;

delete head;

head = current;

}

Size = 0;

}

}

void set(size\_t k, T obj) // replacement of an element by index with a transmitted element

{

if (this->head != NULL && this->get\_size() >= k && k >= 0) {

Node\* current = head;

for (int i = 0; i < k; i++) {

current = current->Next;

}

current->data = obj;

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

void sort() {

T\* array = new T[this->Size];

Node\* cur = head;

int i = 0;

while (cur != NULL) {

array[i] = cur->data;

cur = cur->Next;

i++;

}

QuickSort(array, 0, this->Size - 1);

cur = head;

i = 0;

while (cur != NULL) {

cur->data = array[i];

cur = cur->Next;

i++;

}

}

bool isEmpty() { // check for empty list bc

return (bool)(head);

}

void reverse() { // reorders items in a list

int Counter = Size;

Node\* HeadCur = NULL;

Node\* TailCur = NULL;

for (int j = 0; j < Size; j++) {

if (HeadCur != NULL) {

if (head != NULL && head->Next == NULL) {

TailCur->Next = head;

TailCur = head;

head = NULL;

}

else {

Node\* cur = head;

for (int i = 0; i < Counter - 2; i++)

cur = cur->Next;

TailCur->Next = cur->Next;

TailCur = cur->Next;

cur->Next = NULL;

tail = cur;

Counter--;

}

}

else {

HeadCur = tail;

TailCur = tail;

Node\* cur = head;

for (int i = 0; i < Size - 2; i++)

cur = cur->Next;

tail = cur;

tail->Next = NULL;

Counter--;

}

}

head = HeadCur;

tail = TailCur;

}

public:

List(Node\* head = NULL, Node\* tail = NULL, int Size = 0) :head(head), tail(tail), Size(Size) {}

~List() {

if (head != NULL) {

this->clear();

}

};

private:

Node\* head;

Node\* tail;

int Size;

void swap(T& a, T& b) {

T c = a;

a = b;

b = c;

}

void QuickSort(T\* arr, int first, int last)

{

T mid;

int f = first, l = last;

mid = arr[(f + l) / 2];

do

{

while (arr[f] < mid) f++;

while (arr[l] > mid) l--;

if (f <= l)

{

swap(arr[f], arr[l]);

f++;

l--;

}

} while (f < l);

if (first < l) QuickSort(arr, first, l);

if (f < last) QuickSort(arr, f, last);

}

};

**Pair.h**

#pragma once

template<class T, class T1>

class Pair {

public:

Pair(T first = T(), T1 second = T1()) :first(first), second(second) {}

T first;

T1 second;

template<class T, class T1> friend bool operator >(const Pair<T, T1>& a, const Pair<T, T1>& b);

template<class T, class T1> friend bool operator <(const Pair<T, T1>& a, const Pair<T, T1>& b);

};

template<class T, class T1>

bool operator> (const Pair<T, T1>& a, const Pair<T, T1>& b) {

return a.second > b.second;

}

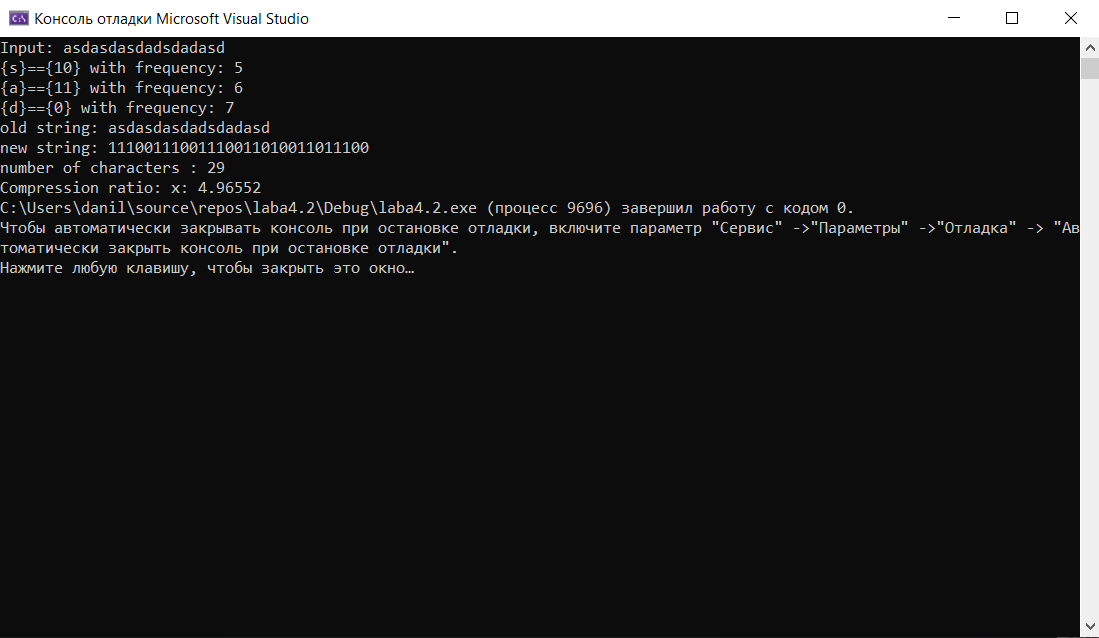
template<class T, class T1>

bool operator< (const Pair<T, T1>& a, const Pair<T, T1>& b) {

return a.second < b.second;

}

**Пример работы**



**Вывод**

При написании программы были улучшены знания ООП, а также изучен алгоритм кодирования Хаффмана.