**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №1**

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

**Тема: «Списки и их реализация»**

**Вариант 18(2)**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Плюснина Е.Ю. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**Постановка задачи**

Реализовать кодирование и декодирование по алгоритму Шеннона-Фано (2 вариант) входной строки, вводимой через консоль.

# Описание реализуемого класса, оценка временной сложности

class ShannonFano – класс реализации алгоритма Шеннона-Фано.

char\* input – входная строка.

int inputLength – длина входной строки.

bool\* markString – закодированная строка.

int markLength – закодированная строка.

class Node

BList<char>\* symb – массив символов.

int all – количество символов.

bool leaf – является ли элемент «листом».

Node\* left – левый элемент.

Node\* right – правый элемент.

Node\* parent – «предок».

**Функции:**

void Tree(Node\*) – построение дерева. Оценка временной сложности: О(n\*n)

void MarkF(Node\*) – построение кода. Оценка временной сложности: О(n\*n)

bool\* MarkString() – кодирование строки. Оценка временной сложности: О(n\*logn)

int GetMarkLength() – возвращение длины строки.

double Coefficient() - возвращение коэффициента сжатия.

char\* DecodeString() - декодирование строки. Оценка временной сложности: О(n\*logn)

void CodesOut() – вывод кодов в консоль.

# Описание реализованных unit-тестов

EasyString – кодирование простой строки (4 элемента).

String\_1 – кодирование строки.

String\_2 – кодирование строки.

String\_3 – кодирование строки (49 элементов)

nullString – кодирование пустой строки.

# Листинг

**Queue.cpp**

#include <iostream>

using namespace std;

template <class T>

class BList

{

private:

class Node

{

public:

Node(T this\_branch, int priority = 0, Node\* previous\_branch = nullptr, Node\* next\_branch = nullptr)

{

this->this\_branch = this\_branch;

this->priority = priority;

this->previous\_branch = previous\_branch;

this->next\_branch = next\_branch;

};

~Node()

{ }

T this\_branch;

int priority;

Node\* next\_branch;

Node\* previous\_branch;

private:

};

Node\* first;

Node\* last;

size\_t size;

public:

BList(Node\* first = nullptr, Node\* last = nullptr) {

this->first = first;

this->last = last;

size = 0;

};

void push(T, int);

size\_t get\_size();

T pop\_back();

T get\_front();

T pop\_front();

int first\_priority();

int last\_priority();

bool IsEmpty();

void clear();

BList<T>\* Copy();

~BList()

{

while (first->next\_branch != nullptr) {

first = first->next\_branch;

delete first->previous\_branch;

}

size = 0;

delete first;

};

};

template <class T>

void BList<T>::push(T add, int priority)

{

if (size == 0)

{

first = new Node(add, priority);

last = first;

}

else

{

Node\* NewNode = new Node(add, priority);

Node\* current = first;

while ((current->next\_branch != NULL) && (NewNode->priority < current->priority))

{

current = current->next\_branch;

}

if ((NewNode->priority <= current->priority) && (current->next\_branch == NULL)) {

NewNode->previous\_branch = current;

NewNode->next\_branch = NULL;

current->next\_branch = NewNode;

last = NewNode;

}

else if ((NewNode->priority >= current->priority) && (current->previous\_branch == NULL))

{

NewNode->previous\_branch = NULL;

NewNode->next\_branch = current;

current->previous\_branch = NewNode;

first = NewNode;

}

else {

NewNode->next\_branch = current;

NewNode->previous\_branch = current->previous\_branch;

current->previous\_branch->next\_branch = NewNode;

current->previous\_branch = NewNode;

}

}

size++;

}

template <class T>

size\_t BList<T>::get\_size()

{

return size;

}

template <class T>

T BList<T>::pop\_front()

{

T data;

if (size == 1)

{

data = first->this\_branch;

size = 0;

first = NULL;

last = NULL;

}

else if (size > 1)

{

data = first->this\_branch;

first = first->next\_branch;

delete first->previous\_branch;

first->previous\_branch = NULL;

size--;

}

else {

throw out\_of\_range("Error");

}

return data;

}

template <class T>

T BList<T>::pop\_back()

{

T data;

if (size == 1)

{

data = last->this\_branch;

size = 0;

first = NULL;

last = NULL;

}

else if (size > 1)

{

data = last->this\_branch;

last = last->previous\_branch;

delete last->next\_branch;

last->next\_branch = NULL;

size--;

}

else {

throw out\_of\_range("Error");

}

return data;

}

template <class T>

int BList<T>::first\_priority() {

return first->priority;

}

template <class T>

int BList<T>::last\_priority() {

return last->priority;

}

template <class T>

void BList<T>::clear()

{

if (first != NULL)

{

while (first->next\_branch != NULL)

{

first = first->next\_branch;

delete first->previous\_branch;

}

size = 0;

first = NULL;

last = NULL;

}

}

template <class T>

bool BList<T>::IsEmpty()

{

if (first == NULL)

return true;

return false;

}

template <class T>

BList<T>\* BList<T>::Copy() {

BList<T>\* NewQueue = new BList<T>();

Node\* current= first;

while (current!= nullptr) {

if (current== NULL) break;

NewQueue->push(current->this\_branch, current->priority);

current= current->next\_branch;

}

return NewQueue;

}

template <class T>

T BList<T>::get\_front() {

return first->this\_branch;

}

**Shannon-Fano.h**

#pragma once

#include "RB-Tree.cpp"

#include "Queue.cpp"

class ShannonFano

{

private:

class Node {

public:

BList<char>\* symb;

int all;

bool leaf;

Node\* left;

Node\* right;

Node\* parent;

Node() {

this->left = nullptr; this->right = nullptr; this->parent = nullptr;

this->symb = new BList<char>();

this->leaf = false;

this->all = 0;

}

};

class Mark {

public:

bool\* mark;

int length;

Mark(bool\* mark = nullptr, int length = 0) {

this->mark = mark;

this->length = length;

}

};

Node\* root;

char\* input;

int inputLength;

RedBlackTree<char, Mark\*>\* marks;

bool\* markString;

int markLength;

BList<char>\* probability(char\*, int\*);

void Tree(Node\*);

void MarkF(Node\*);

public:

ShannonFano(char\* string, int stringLength) {

this->markString = NULL;

this->markLength = 0;

this->input = string;

this->inputLength = stringLength;

if (stringLength == 0) {

root = nullptr;

}

else {

root = new Node();

root->symb = probability(string, &stringLength);

root->all = stringLength;

if (root->symb->get\_size() == 1) {

root->leaf = true;

}

else {

root->leaf = false;

Tree(root);

}

}

MarkF(root);

}

bool\* MarkString();

int GetMarkLength();

double Coefficient();

char\* DecodeString();

void CodesOut();

~ShannonFano() {};

};

**Shannon-Fano.cpp**

#include "Shannon-Fano.h"

#include"Stack.h"

BList<char>\* ShannonFano::probability(char\* string, int\* length) {

BList<char>\* symb = new BList<char>();

int chars[256] = { 0 };

int count = 0;

for (int i = 0; (string[i] != i < \*length && '\n'); i++)

{

chars[string[i]]++; count++;

}

for (int i = 0; i < 256; i++) {

if (chars[i] != 0) symb->push((char)i, chars[i]);

}

\*length = count;

return symb;

}

void ShannonFano::Tree(Node\* current) {

if (current->symb->get\_size() == 1) {

current->leaf = true;

return;

}

BList<char>\* symb = new BList<char>();

symb = current->symb->Copy();

Node\* left = new Node();

Node\* right = new Node();

int pLeft, pRight;

pLeft = symb->first\_priority();

left->symb->push(symb->pop\_front(), pLeft);

pRight = symb->last\_priority();

right->symb->push(symb->pop\_back(), pRight);

while (!symb->IsEmpty()) {

int pCurrent;

if (pLeft > pRight) {

pCurrent = symb->last\_priority();

right->symb->push(symb->pop\_back(), pCurrent);

pRight += pCurrent;

}

else {

pCurrent = symb->first\_priority();

left->symb->push(symb->pop\_front(), pCurrent);

pLeft += pCurrent;

}

}

left->all = pLeft;

right->all = pRight;

current->left = left;

current->right = right;

left->parent = current;

right->parent = current;

Tree(left);

Tree(right);

}

void ShannonFano::MarkF(Node\* root) {

if (root == nullptr) {

this->marks = nullptr;

return;

}

Stack<Node\*>\* stack = new Stack<Node\*>();

stack->push(root);

RedBlackTree<char, Mark\*>\* marks = new RedBlackTree<char, Mark\*>();

bool\* CurrentMark = new bool[root->all / 2 + 1];

Node\* current = root;

int deep = 0;

if (root->leaf) {

deep = 1;

CurrentMark[0] = 0;

}

while (!stack->IsEmpty()) {

if (current->left != NULL) {

CurrentMark[deep] = 1;

deep++;

if (current->right != NULL) {

stack->push(current->right);

}

current = current->left;

}

else if (current->right != NULL) {

CurrentMark[deep] = 0;

deep++;

current = current->right;

}

else {

bool\* finalCode = new bool[deep];

for (int i = 0; i < deep; i++) finalCode[i] = CurrentMark[i];

Mark\* Code = new Mark(finalCode, deep);

marks->insert((int)current->symb->get\_front(), Code);

while (stack->get\_top() != current->right) {

current = current->parent;

deep--;

if (current == nullptr) break;

}

current = stack->pop();

CurrentMark[deep] = 0;

deep++;

}

}

this->marks = marks;

}

bool\* ShannonFano::MarkString() {

if (root == nullptr) return NULL;

if (markString != NULL) return markString; //if we did this already

BList<char>\* symb = new BList<char>();

symb = root->symb->Copy();

int prior, codeLength = 0; char CurrentSymbol;

while (!symb->IsEmpty()) {

prior = symb->first\_priority();

CurrentSymbol = symb->pop\_front();

Mark\* CurrentMark = marks->find(CurrentSymbol);

codeLength += prior \* CurrentMark->length;

}

this->markLength = codeLength;

markString = new bool[codeLength];

int currenti = 0, j = 0;

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

CurrentSymbol = input[j];

Mark\* CurrentMark = marks->find(CurrentSymbol);

for (int i = 0; i < CurrentMark->length; i++) {

markString[currenti] = CurrentMark->mark[i]; currenti++;

}

}

return markString;

}

char\* ShannonFano::DecodeString() {

if (markString == NULL) return NULL;

int curLength = 0;

Node\* current = root;

char\* string = new char[markLength];

for (int i = 0; curLength < markLength && i < markLength;) {

while (!current->leaf) {

if (markString[i] == 1) {

current = current->left;

}

else {

current = current->right;

}

i++;

}

string[curLength] = current->symb->get\_front();

curLength++;

current = root;

}

for (int i = 0; i < curLength + 1; i++) cout << string[i];

char\* finalString = new char[curLength + 1];

for (int i = 0; i < curLength + 1; i++) finalString[i] = string[i];

return finalString;

}

void ShannonFano::CodesOut() {

char\* chars = marks->get\_keys();

Mark\*\* Symbs = marks->get\_values();

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

{

cout << endl << chars[i] << " ";

for (int j = 0; j < Symbs[i]->length; j++)

cout << Symbs[i]->mark[j];

}

}

int ShannonFano::GetMarkLength() {

return markLength;

}

double ShannonFano::Coefficient() {

return (double)sizeof(char) \* inputLength \* 8 / markLength;

}

**UnitTest\_4sem\_lab2.cpp**

#include "CppUnitTest.h"

#include "..\AlgStrD\_4sem\_lab2\Shannon-Fano.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Sem4Lab1test

{

TEST\_CLASS(Sem4Lab1test)

{

public:

TEST\_METHOD(EasyString)

{

char\* string = new char[4]{ 'c','o','o','l' };

ShannonFano \* test = new ShannonFano(string, 4);

bool\* code = test->MarkString();

char\* decode = test->DecodeString();

for (size\_t i = 0; i < 4; i++)

{

Assert::AreEqual(string[i], decode[i]);

}

}

TEST\_METHOD(String\_1)

{

char\* string = new char[1]{ 'h' };

ShannonFano\* test = new ShannonFano(string, 1);

bool\* code = test->MarkString();

Assert::AreEqual(test->GetMarkLength(), 1);

char\* decode = test->DecodeString();

for (size\_t i = 0; i < 1; i++)

{

Assert::AreEqual(string[i], decode[i]);

}

}

TEST\_METHOD(String\_2)

{

char\* string = new char[3]{ '.','.','.' };

ShannonFano\* test = new ShannonFano(string, 3);

bool\* code = test->MarkString();

Assert::AreEqual(test->GetMarkLength(), 3);

char\* decode = test->DecodeString();

for (size\_t i = 0; i < 3; i++)

{

Assert::AreEqual(string[i], decode[i]);

}

}

TEST\_METHOD(String\_3)

{

char\* string = new char[49]{ 'T','h','e',' ','s','h','a','d','o','w','s',' ','o','f',' ','t','h','e',' ','b','o','n','f','i','r','e','s',' ','a','r','e',' ','h','i','d','d','e','n',' ','f','r','o','m',' ','s','i','g','h','t' };

ShannonFano\* test = new ShannonFano(string, 49);

bool\* code = test->MarkString();

char\* decode = test->DecodeString();

for (size\_t i = 0; i < 16; i++)

{

Assert::AreEqual(string[i], decode[i]);

}

}

TEST\_METHOD(nullString)

{

ShannonFano\* test = new ShannonFano(NULL, 0);

bool\* code = test->MarkString();

Assert::AreEqual(NULL, test->DecodeString());

}

};

}

# Вывод

Я научилась реализовывать алгоритм Шеннона-Фано.