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: Stack

Студент гр. 8301 _____ Пчёлко В.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2020

Оглавление

Цель работы

Реализовать алгоритм Хаффмана.

Ход работы

Описание реализуемых классов:

Класс: haffman_algo

Класс представляет собой дерево для «постройки» кодов Хаффмана для каждого символа строки. Содержит указатели на левый и правый потомок, а также поле данных Node, представляющее из себя пару (вспомогательный класс affman_pair) символ + количество вхождений в строку.

Методы:

- Конструктор – использует функцию build_tree для постройки дерева Хаффмана. Перед запуском функции постройки дерева преобразует строку в список пар символ + количество вхождений.
- Деструктор – использует функцию clear для удаления дерева.
- build_tree – функция строит дерево по принципу: сортировка списка по возрастанию количества вхождений символа в строку. Далее по алгоритму Хаффмана добавляем элементы в дерево и удаляем их из списка пока список не опустеет.
- get_codes_map – возвращает ассоциативный массив символ + код Хаффмана, построенный по принципу Хаффмана используя метод add_codes.
- add_codes – заполняет ассоциативный массив символ + код кодами символов по принципу Хаффмана.
- get_list_of_chars – возвращает список пар.
- decode_string – декодирует строку путем полного обхода всех символов. Одновременно с обходом символов происходит обход дерева (0 – левый потомок, 1 - правый) пока не получим символ. Далее возвращаемся в корень дерева.

Временная сложность методов:

- build_tree – O(NlogN)
- get_list_of_chars– O(const)
- add_codes – O(N)
- get_codes_map– O(N)
- decode_string – O(NlogN)

Логарифмическая сложность методов обусловлена использованием дерева. Метод возвращающий список работает за константу ввиду того что мы храним список. Заполнение кодами имеет сложность N ввиду необходимости построить код для каждого символа без исключения.

Unit-Тесты

Реализованные юнит тесты проверяют кодирование строк, содержащих разные символы, повторяющиеся символы, а также алгоритм декодирования.

Пример работы программы

```
Введите строку для кодирования: qqqqqqqqqq
Таблица кодов:
Символ <---> Код
   q   <---> 0 <---> вхождений в строку: 10
старая строка: qqqqqqqqqq
новая строка: 0000000000
Размер декодированной(исходной) строки = 80 бит
Размер закодированной строки = 10 бит
Коэффициент сжатия: x8
```

```
Введите строку для кодирования: The plastic world has won.
Таблица кодов:
Символ <---> Код
   i   <---> 10100 <---> вхождений в строку: 1
   e   <---> 10101 <---> вхождений в строку: 1
   n   <---> 11010 <---> вхождений в строку: 1
   r   <---> 11011 <---> вхождений в строку: 1
   p   <---> 11000 <---> вхождений в строку: 1
   t   <---> 11001 <---> вхождений в строку: 1
   .   <---> 0010 <---> вхождений в строку: 1
   T   <---> 0011 <---> вхождений в строку: 1
   c   <---> 0000 <---> вхождений в строку: 1
   d   <---> 0001 <---> вхождений в строку: 1
   s   <---> 1011 <---> вхождений в строку: 2
   w   <---> 1000 <---> вхождений в строку: 2
   l   <---> 1001 <---> вхождений в строку: 2
   h   <---> 0110 <---> вхождений в строку: 2
   o   <---> 0111 <---> вхождений в строку: 2
   a   <---> 010 <---> вхождений в строку: 2
       <---> 111 <---> вхождений в строку: 4
старая строка: The plastic world has won.
новая строка: 0011011010101111110001001010101111001101000000111100001111101110010001111011001010111111000011110100010
Размер декодированной(исходной) строки = 208 бит
Размер закодированной строки = 104 бит
Коэффициент сжатия: x2
```

```
Введите строку для кодирования: The plastic world has won. A mock-up turned out to be the stronger one. The last little ship has cooled down. The last little lantern has tired
Таблица кодов:
Символ <---> Код
   m   <---> 0111000 <---> вхождений в строку: 1
   b   <---> 0111001 <---> вхождений в строку: 1
   k   <---> 0010110 <---> вхождений в строку: 1
   g   <---> 0010111 <---> вхождений в строку: 1
   -   <---> 0010100 <---> вхождений в строку: 1
   A   <---> 0010101 <---> вхождений в строку: 1
   u   <---> 011101 <---> вхождений в строку: 3
   w   <---> 101100 <---> вхождений в строку: 3
   p   <---> 101101 <---> вхождений в строку: 3
   T   <---> 101010 <---> вхождений в строку: 3
   .   <---> 101011 <---> вхождений в строку: 3
   c   <---> 00100 <---> вхождений в строку: 3
   d   <---> 01111 <---> вхождений в строку: 5
   i   <---> 10100 <---> вхождений в строку: 5
   r   <---> 10111 <---> вхождений в строку: 6
   n   <---> 0011 <---> вхождений в строку: 7
   a   <---> 1110 <---> вхождений в строку: 7
   s   <---> 1111 <---> вхождений в строку: 8
   h   <---> 0110 <---> вхождений в строку: 8
   o   <---> 1000 <---> вхождений в строку: 10
   l   <---> 1001 <---> вхождений в строку: 10
   e   <---> 000 <---> вхождений в строку: 13
   t   <---> 010 <---> вхождений в строку: 14
       <---> 110 <---> вхождений в строку: 26
старая строка: The plastic world has won. A mock-up turned out to be the stronger one. The last little ship has cooled down. The last little lantern has tired
новая строка: 1010100110000110101101100111101111010101000010011010110010001011110010111111001101110111111010110010000011101011110001010111001110001000001000010110001010001110110
1101110010011101101110011000011111101000011101010110010100011001110010001100100110000110111101010111100000110010111000101111101000001100010101111010101001100001101001111101111010
1101001101000100101001000110111101101010010110111001101110111111000100100010001001000011111100111110001011000011101011110101010011000011010011110111101011010011010001001010001000
110100111100011010000101110011110011011101111110010101001011100001111
Размер декодированной(исходной) строки = 1144 бит
Размер закодированной строки = 586 бит
Коэффициент сжатия: x1.95222
```

Выводы

В ходе лабораторной работы я ознакомился и реализовал алгоритм Хаффмана для кодирования и декодирования строк.

Код программы
haffman_algo.h

```
#pragma once
#include"list.h"
#include"map.h"
#include"haffman_pair.h"
#include<string>
class HaffmanTree
{
	class Node {
	public:
		Node(haffman_pair<char, int> h_pair = haffman_pair<char, int>(), Node* left = 
NULL, Node* right = NULL) :h_pair(h_pair), left(left), right(right) {}
		haffman_pair<char, int> h_pair;
		Node* left;
		Node* right;
	};
public:
	~HaffmanTree() {
		this->clear_tree(Top);
	}
	HaffmanTree(string str) {
		Map<char, int>* map_of_chars = new Map<char, int>();
		list_of_chars = new List<haffman_pair<char, int>>();
		for (int i = 0; i < str.size(); i++) {
			if (!map_of_chars->is_in_map(str[i]))  //if the symbol is not in map 
then add symbol to map else increase the number of entries
				map_of_chars->insert(str[i], 1);
			else
				map_of_chars->add_entry(str[i]);
		}
		list_of_chars = map_of_chars->get_pairs();
		map_of_chars->clear();
		list_of_chars->sort();
		//----------------build haffman tree------------------------
		List<Node>* list_of_haffman_nodes = new List<Node>();
		for (int i = 0; i < list_of_chars->get_size(); i++)
			list_of_haffman_nodes->push_back(Node(list_of_chars->at(i)));
		build_tree(list_of_haffman_nodes);
	}
	List<haffman_pair<char, int>>* get_list_of_chars() {
		return list_of_chars;
	}
	Map<char, string>*& get_codes_map() {
		Map<char, string>* haffman_codes_map = new Map<char, string>();
		string current;
		add_codes(haffman_codes_map, Top, current);
		return haffman_codes_map;
	}
	string decode_string(string& encrypted) {
		string decoded;
		int pos = 0;
		decode(Top, encrypted, decoded, pos);
		return decoded;
	}
private:
	void decode(Node* root, string& encrypted, string& decoded, int& position) {
		if (encrypted.size() > position) {
			while (root->right != NULL && root->left != NULL) {
				if (encrypted[position] == '0')
					root = root->left;
				else
					root = root->right;
```

```
                        position++;
                }
                decoded += root->h_pair.symbol;
                if (Top->left == NULL && Top->right == NULL)
                        position++;
                decode(Top, encrypted, decoded, position);
        }
    }
    void build_tree(List<Node>*& list_of_haffman_nodes) {
        Top = NULL;
        if (list_of_haffman_nodes->get_size() > 1) {
                while (list_of_haffman_nodes->get_size() != 0) {
                        Node* current = new Node();
                        current->left = new Node(list_of_haffman_nodes->at(0));
                        current->right = new Node(list_of_haffman_nodes->at(1));
                        current->h_pair.num_of_entry = list_of_haffman_nodes-
>at(0).h_pair.num_of_entry + list_of_haffman_nodes->at(1).h_pair.num_of_entry;
                        list_of_haffman_nodes->pop_front();
                        list_of_haffman_nodes->pop_front();
                        int i = 0;
                        for (; i < list_of_haffman_nodes->get_size() &&
list_of_haffman_nodes->at(i).h_pair.num_of_entry < current->h_pair.num_of_entry; i++);
                        if (list_of_haffman_nodes->get_size() != 0 &&
list_of_haffman_nodes->get_size() != i)
                                list_of_haffman_nodes->insert(*current, i);
                        else
                                if (list_of_haffman_nodes->get_size() == i &&
list_of_haffman_nodes->get_size() != 0)
                                        list_of_haffman_nodes->push_back(*current);
                        if (list_of_haffman_nodes->get_size() == 0) {
                                Top = current;
                        }
                }
        }
        else {
                Top = new Node(haffman_pair<char, int>(list_of_haffman_nodes-
>at(0).h_pair.symbol, list_of_haffman_nodes->at(0).h_pair.num_of_entry));
        }
    }
    void add_codes(Map<char, string>* haffman_codes_map, Node* root, string current) {
        if (Top->left != NULL && Top->right != NULL) {
                if (root->left != NULL && root->right != NULL) {
                        add_codes(haffman_codes_map, root->left, current + '0');
                        add_codes(haffman_codes_map, root->right, current + '1');
                }
                else
                {
                        haffman_codes_map->insert(root->h_pair.symbol, current);
                }
        }
        else {
                haffman_codes_map->insert(root->h_pair.symbol, current + '0');
        }
    }
    void clear_tree(Node* tree) {
        if (tree != NULL) {
                clear_tree(tree->left);
                clear_tree(tree->right);
                delete tree;
        }
    }
    Node* Top;
    List<haffman_pair<char, int>>* list_of_chars;
    };
```

Unit-tests

```
#include "stdafx.h"
#include "CppUnitTest.h"
#include "..//AaDSLab2/haffman_algo.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace HaffmanTests
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(different_chars)
		{
			string str = "Hello";
			string encrypted;
			HaffmanTree* haffmanTree = new HaffmanTree(str);
			Map<char, string>* hoffman = haffmanTree->get_codes_map();
			List<haffman_pair<char, int>>* list_symbol = haffmanTree-
>get_list_of_chars();
			for (int i = 0; i < str.size(); i++)
				encrypted += hoffman->find(str[i]);
			Assert::AreEqual(encrypted, string("1001111100"));
		}
		TEST_METHOD(same_chars)
		{
			string str = "fff";
			string encrypted;
			HaffmanTree* haffmanTree = new HaffmanTree(str);
			Map<char, string>* hoffman = haffmanTree->get_codes_map();
			List<haffman_pair<char, int>>* list_symbol = haffmanTree-
>get_list_of_chars();
			for (int i = 0; i < str.size(); i++)
				encrypted += hoffman->find(str[i]);
			Assert::AreEqual(encrypted, string("000"));
		}
		TEST_METHOD(text_encryption_decoding)
		{
			string str = "i love coding";
			string encrypted;
			HaffmanTree* haffmanTree = new HaffmanTree(str);
			Map<char, string>* hoffman = haffmanTree->get_codes_map();
			List<haffman_pair<char, int>>* list_symbol = haffmanTree-
>get_list_of_chars();
			for (int i = 0; i < str.size(); i++)
				encrypted += hoffman->find(str[i]);
			string decoded = haffmanTree->decode_string(encrypted);
			Assert::AreEqual(str, decoded);
		}

	};
	}
```