МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Кодирование

Студент гр. 9384 _____ Гурин С.Н.

Преподаватель _____ Ефремов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить способ реализации алгоритма кодирования Хаффмана на языке C++.

Задание.

ВАРИАНТ 3.

Кодирование: статическое Хаффмана

Выполнение работы.

При запуске программы пользователь должен ввести в файл input.txt строку для кодирования по алгоритму Хафмана. Если файл оказывается пустым, то пользователю выводится сообщение об ошибке и программа завершает свою работу. Если же строка введена верно, то программа сначала выводит введенную строку, затем с помощью метода класса Coder _CountOfLetters() идет подсчет и запись в массив количество различных букв.

Далее происходит определение и создание с помощью метода класса Coder _MakePriority() структуры priority, которая включает в себя символы и их приоритеты. Запись приоритетов происходит происходит с помощью удаления всех одинаковых символов в строке для того, чтобы обезопасить структуру от повтоения символов. Затем происходит сортировка по возрастания полученного массива структур. После выполнения этого метода происходит вывод всех приоритетов

Затем с помощью метода _MakeNodes() происходит создание из полученной ранее структуры двухсвязный список узлов вхождения символов. Реализация этого списка происходит с помощью класса List. Этот класс включает в себя методы _Add() - добавление элемента в список, _Pop() - удаления двух первых элементов списка, и _SortNodes() - сортировка элементов списка по возрастанию.

После выполнения этого метода происходит кодирование элементов с помощью метода _Huffman(). Кодирование выполняется рекурсивно созданием

из первых двух минимальных элементов списка бинарного дерева, их удаления и добавления полученного дерева. Этот метод выполняется до тех пор, пока не останется один элемент списка.

Далее полученный элемент списка записывается в элемент класса Coder. Затем происходит определение массива структур code, который включает в себя бинарные коды символа и сами символы.

После этого с помощью метода _WriteCodedValue() происходит ассоциация элементов этого дерева и кодов этих элементов. Код элемента строится таким образом, что при обходе по бинарному дереву записывается путь к данному элементу(left (zero) = 0, right (one) = 1)

Затем происходит вывод полученной структуры и закодированная строка.

Тестирование.

№ теста	ввод input.txt	вывод console
1		`Empty String`

2	aab	`Your string: aab`
		`Priorities: b-1; a-2;`
		`Code: b-0; a-1;`
		`Coded string: 1 1 0`

3	(a(v)c)	`Your string: (a(v)c)`
		`Priorities: a-1; v-1; c-1; (-2; )-2;`
		`Code: c-00; (-01; )-010; a-0110; v-0111;`
		`Coded string: 01 0110 01 0111 010 00 010`

4	abcd	`Your string: abcd`
		`Priorities: a-1; b-1; c-1; d-1;`
		`Code: a-00; b-01; c-010; d-011;`
		`Coded string: 00 01 010 011`

Выводы.

При выполнении данной лабораторной работы был изучен алгоритм кодирования Хаффмана на языке C++.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "Coder.h"
#include "List.h"

int main(){
    Coder<char> b;
    List<Coder<char>*> ptr;
    std::string tmp;
    std::ifstream  file;
    file.open("input.txt");
    getline(file, tmp);
    file.close();
    if (tmp.empty()) {
        std::cout << "Empty String\n";
        exit(1);
    }
    std::cout << "Your string: " << tmp << "\n";
    int amount = b._CountOfLetters(tmp);
    Coder<char>::priority* prior = new Coder<char>::priority[amount];
    b._MakePriority(prior, tmp, amount);
    std::cout << "Priorities: ";
    for (int i = 0; i < amount; i++) {
        if (prior[i].letter != ' ') {
            std::cout << prior[i].letter << "-" << prior[i].count << ";
";
        }
        else {
            std::cout << "SPACE" << "-" << prior[i].count << "; ";
        }
    }
    b._MakeNodes(prior, amount, ptr);
    b._Huffman(ptr);
    Coder<char>* Tree = ptr.Head->x;
    Coder<char>::CodedValue* code = new Coder<char>::CodedValue[amount];
    b._WriteCodedValue(Tree, code);
    std::cout << "\nCode: ";
    for (int i = 0; i < amount; i++) {
        if (code[i].letter != ' ') {
            std::cout << code[i].letter << "-";
            for (size_t n = 0; n < code[i].code.size(); n++) {
                std::cout << code[i].code[n];
            }
            std::cout << "; ";
        }
        else {
            std::cout << "SPACE" << "-";
            for (size_t n = 0; n < code[i].code.size(); n++) {
                std::cout << code[i].code[n];
            }
```

```
            std::cout << "; ";
        }
    }
    std::cout << "\nCoded string: ";
    for (size_t i = 0; i < tmp.size(); i++) {
        for (int j = 0; j < amount; j++) {
            if (tmp[i] == code[j].letter) {
                for (size_t n = 0; n < code[j].code.size(); n++) {
                    std::cout << code[j].code[n];
                }
            }
        }
        std::cout << " ";
    }

    return 0;
}
```

Название файла: Coder.h

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include "List.h"
template <typename T>
class Coder {
public:
     Coder() {
          this->freq = 0;
          this->root = NULL;
          this->zero = nullptr;
          this->one = nullptr;
     }

     Coder(Coder* Left, Coder* Right) {
          this->zero = Left;
          this->one = Right;
          this->freq = Left->GetFreq() + Right->GetFreq();
     }

     struct priority {
          int count = 0;
          char letter = NULL;
     };

     struct CodedValue {
          std::vector<bool> code;
          char letter = NULL;
     };

     int GetFreq() {
          return this->freq;
     }

     Coder* GetZero() {
          return this->zero;
     }
```

```
      Coder* GetOne() {
            return this->one;
      }

      T GetRoot() {
            return this->root;
      }

    void _MakeNodes(priority prior[], int amount, List<Coder*>& ptr) {
        MakeNodes(prior, amount, ptr);
    }

      void _Huffman(List<Coder*>& ptr) {
            Huffman(ptr);
      }

    int _CountOfLetters(std::string tmp){
        return CountOfLetters(tmp);
    }

    void _MakePriority(priority prior[], std::string tmp, int amount) {
        MakePriority(prior, tmp, amount);
    }

    void _WriteCodedValue(Coder<char>* Tree, CodedValue code[]) {
        WriteCodedValue(Tree, code);
    }

private:
      int freq;
      T root;
      Coder* zero;
      Coder* one;

    void MakeNodes(priority prior[], int amount, List<Coder*>& ptr) {
        for (int i = 0; i < amount; i++) {
            Coder* p = new Coder;
            p->root = prior[i].letter;
            p->freq = prior[i].count;
            ptr._Add(p);
        }
    }

      void Huffman(List<Coder*>& ptr) {
            if (ptr.GetCount() != 1) {
                  ptr._SortNodes();
                  Coder* Left = ptr.Head->x;
                  Coder* Right = ptr.Head->Next->x;
                  ptr._Pop();
                  Coder* p = new Coder(Left, Right);
                  ptr._Add(p);
                  Huffman(ptr);
            }
      }

    void CountingPriority(std::string tmp, int arr[]) {
        for (size_t i = 0; i < tmp.size(); i++)
```

```
    {
        arr[(int)tmp[i] - 32]++;
    }
}


void DeleteLeter(char letter, std::string& tmp) {
    for (size_t i = 0; i <= tmp.length(); i++) {
        if (tmp[i] == letter) {
            tmp.erase(i, 1);
            i--;
        }
    }
}

int Count(char letter, std::string tmp) {
    int count = 0;
    for (size_t i = 0; i <= tmp.length(); i++) {
        if (tmp[i] == letter) {
            count++;
        }
    }
    return count;
}

int CountOfLetters(std::string tmp) {
    int arr[95] = { 0 };
    CountingPriority(tmp, arr);
    int amount = 0;
    for (int i = 0; i < 95; i++)
    {
        if (arr[i] != 0) {
            amount++;
        }
    }
    return amount;
}

void Sort(priority prior[], int amount) {
    int temp1;
    char temp2;
    for (int i = 0; i < amount - 1; i++) {
        for (int j = 0; j < amount - i - 1; j++) {
            if (prior[j].count > prior[j + 1].count) {
                temp1 = prior[j].count;
                temp2 = prior[j].letter;
                prior[j].count = prior[j + 1].count;
                prior[j].letter = prior[j + 1].letter;
                prior[j + 1].count = temp1;
                prior[j + 1].letter = temp2;
            }
        }
    }
}

void MakePriority(priority prior[], std::string tmp, int amount) {
    for (int i = 0; i < amount; i++) {
```

```
            prior[i].letter = tmp[0];
            prior[i].count = Count(prior[i].letter, tmp);
            DeleteLeter(prior[i].letter, tmp);
        }
        Sort(prior, amount);
    }

    int index_1 = 0;
    std::vector<bool> temp;

    void WriteCodedValue(Coder<char>* Tree, CodedValue code[]) {
        if (Tree->GetZero() != NULL) {
            temp.push_back(0);
            WriteCodedValue(Tree->GetZero(), code);
        }
        if (Tree->GetOne() != NULL) {
            temp.push_back(1);
            WriteCodedValue(Tree->GetOne(), code);
        }
        if (Tree->GetZero() == NULL && Tree->GetOne() == NULL) {
            code[index_1].letter = Tree->GetRoot();
            code[index_1].code = temp;
            index_1++;
            temp.pop_back();
        }
    }

};
```

Название файла: List.h

```
#pragma once
#include <stdlib.h>
#include <iostream>
#include "Coder.h"

int count = 0;

template <typename P>
class List{
private:
    void Pop() {
        Node* tmp = Head->Next;
        delete Head;
        Head = tmp;
        count--;
    }

    void Add(P x) {
        Node* temp = new Node;
        temp->Next = NULL;
        temp->x = x;
        count++;

        if (Head != NULL)
        {
            temp->Prev = Tail;
            Tail->Next = temp;
```

```
            Tail = temp;
        }
        else
        {
            temp->Prev = NULL;
            Head = Tail = temp;
        }
    }

    void SortNodes() {
        Node* left = Head;
        Node* right = Head->Next;
        Node* temp = new Node;
        while (left->Next) {
            while (right) {
                if ((left->x->GetFreq()) > (right->x->GetFreq())) {
                    temp->x = left->x;
                    left->x = right->x;
                    right->x = temp->x;
                }
                right = right->Next;
            }
            left = left->Next;
            right = left->Next;
        }
    }

public:
    struct Node {
        P x;
        Node* Next, * Prev;
    };
    Node* Head, * Tail;

    List() {
        this->Head = NULL;
        this->Tail = NULL;
    };

    int GetCount() {
        return count;
    }

    void _Pop() {
        Pop();
        Pop();
    }

    void _Add(P(x)) {
        Add(x);
    }

    void _SortNodes() {
        SortNodes();
    }
};
```

Название файла: Makefile

```
all: goal
	./start

goal: main.o
	g++ main.o -o start

main.o: main.cpp Coder.h List.h
	g++ -c main.cpp

clean:
	rm -f *.o
```