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: Динамическое кодирование Хаффмана

Студент гр. 9382 _____ Юрьев С.Ю.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить алгоритм динамического кодирования Хаффмана. Получить навыки реализации алгоритмов динамического кодирования и динамического построения строго упорядоченного дерева Хаффмана, используя алгоритм Виттера.

Основные теоретические положения.

Дерево Хаффмана – строго бинарное дерево, в котором по расположению листьев в дереве можно получить код Хаффмана

Лист — узел множество поддеревьев которого пусто.

Строго бинарное дерево — дерево, в котором содержится ровно 2n - 1 узлов, n из которых являются листьями

Блок — все узлы одного типа (лист/внутренний) одного веса

Лидер блока — узел с максимальным номером внутри блока

Задание.

5. Кодирование: динамическое Хаффмана

Описание структуры данных для реализации дерева Хаффмана.

Для реализации дерева Хаффмана был создан класс Node. В нем определены 8 полей:

m_parent — для хранения указателя на «родителя» узла;

m_right - для хранения указателя на правого «ребенка» узла;

m_left - для хранения указателя на левого «ребенка» узла

m_weight — для хранения веса узла

m_number — для хранения номера узла

m_symbol — для хранения символа содержащегося в листе

m_isLeaf — для хранения флага, указывающего на то, является ли этот узел листом.

m_isNYT — для хранения флага, указывающего на то, является ли этот узел специальным нулевым узлом.

Также определено 3 конструктора для создания узла нужного типа:

Node() - для создания первого нулевого узла, который станет корнем дерева

Node(Node* parent) — для создания последующих нулевых узлов

Node(Node* parent, char symbol) — для создания листьев

Описание алгоритма.

Принимается ввод пользователя, после чего последовательно обрабатывается каждый символ.

На каждой итерации идет проверка на то, встречался ли символ раньше. Если это первое появление такого символа, то символ кодируется, как путь в дереве до нулевого узла + общий код символа, а из нулевого узла создаются 2 новых узла: новый нулевой символ и лист соответствующего символа, а старый нулевой узел становится внутренним узлом. Листом для увеличения становится новосозданный лист. Если же символ встречался ранее, то он кодируется путём до соответствующего ему листа в дереве. Сам же лист меняется местами с лидером блока.

После такой проверки циклично выполняется увеличение веса узла и его родителей и прародителей вплоть до корня, параллельно с этим производятся необходимые перестройки дерева.

При увеличении со сдвигом внутреннего узла в начале он меняется местами с лидером блока листьев веса большего на 1. После этого увеличивается вес самого внутреннего узла, а цикл переходит к его старому родителю (он мог измениться при обмене местами).

При увеличении со сдвигом листа в начале он меняется местами с лидером блока внутренних узлов того же веса, что и лист. После этого увеличивается его собственный вес и цикл переходит к его новому родителю.

Описание функций.

void pushFirst(Node* a) — вносит в список листьев самый первый пустой узел.

Node* findSymbol(char symbol, std::vector<Node*> *leafs) — ищет в переданном векторе лист соответствующий переданному символу и возвращает его, если лист найден не был, то возвращается указатель на нынешний пустой узел.

void swapNodes(Node *a, Node *b) — меняет местами 2 узла

Node* findBlockLeader(std::vector<Node*> *vec, unsigned int weight) — находит в векторе лидера блока переданного веса.

void encodeSymbol(std::string* emptyStr, Node* p) — записывает в строку путь до переданного узла (1 — вправо, 0 - налево)

void splitHafTree(Node* root, std::vector<Node*> *vec, short int level) — записывает в переданный вектор все узлы дерева по уровням, разделяя уровни с помощью нулевых указателей

void remakeNumeration(Node* p) — пересоздает правильную нумерацию узлов в дереве, начиная с переданного узла.

void deleteAllNodes() - освобождает всю динамически выделенную память из под дерева

void fillSymbolCode(char symbol, std::string *str) — записывает в переданную строку ascii-код переданного символа.

Node* slideAndIncrement(Node* p) — производит увеличение веса переданного узла и совершает необходимые перестройки дерева из-за этого увеличения

std::string vitterCoder(char symbol) — основная функция, реализующая кодирование символа. Управляет работой других функций и использует результаты их выполнения.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 — Результаты тестирования

No	Входные данные	Выходные данные	Комментарии
1	aaaa	Enter symbol line aaaa Start coding symbol 'a' It is the first meeting with this symbol. Way to the NYT = - Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with	Ввод множества одинаковых символов подряд

		slidings. Start coding symbol 'a' We saw this symbol earlier. Way to the leaf with this symbol = 1 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'a' We saw this symbol earlier. Way to the leaf with this symbol = 1 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'a' We saw this symbol earlier. Way to the leaf with this symbol = 1 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Final coded line: 01100001111	
2	abccaaff	Enter symbol line abccaaff Start coding symbol 'a' It is the first meeting with this symbol. Way to the NYT = - Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings.	Ввод разных символов с повторениями в разных частях строки

		Start coding symbol 'b' It is the first meeting with this symbol. Way to the NYT = 0 Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'c' It is the first meeting with this symbol. Way to the NYT = 10 Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'c' We saw this symbol earlier. Way to the leaf with this symbol = 01 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'a' We saw this symbol earlier. Way to the leaf with this symbol = 10 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'a' We saw this symbol earlier. Way to the leaf with this symbol = 11 Swap this leaf with leader of the	

		block. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'f' It is the first meeting with this symbol. Way to the NYT = 100 Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'f' We saw this symbol earlier. Way to the leaf with this symbol = 1111 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Final coded line: 01100001001100000100110001101101110001100100111	
3	~	Enter symbol line ~ Start coding symbol '~' It is the first meeting with this symbol. Way to the NYT = - Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Final coded line: 01111100	Ввод одного единственного символа

4	ab c c a	Enter symbol line	Ввод пустых

		ab c c a Start coding symbol 'a' It is the first meeting with this symbol. Way to the NYT = - Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol 'b' It is the first meeting with this symbol. Way to the NYT = 0 Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Final coded line: 01100001001100000	символов между значащими символами
5		Enter symbol line Nothing was entered. Finishing program...	Пустой ввод
6	123311	Enter symbol line 123311 Start coding symbol '1' It is the first meeting with this symbol.	Ввод числа

		Way to the NYT = - Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol '2' It is the first meeting with this symbol. Way to the NYT = 0 Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol '3' It is the first meeting with this symbol. Way to the NYT = 10 Adding 2 new nodes from NYT to the tree. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol '3' We saw this symbol earlier. Way to the leaf with this symbol = 01 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Start coding symbol '1' We saw this symbol earlier. Way to the leaf with this symbol = 10 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings.	

		Start coding symbol '1' We saw this symbol earlier. Way to the leaf with this symbol = 11 Swap this leaf with leader of the block. Increment weight to needed leaf and to all his parents with slidings. Final coded line: 00110001000110000100011001 1011011	

Выводы.

Были изучены и опробованы методы работы и реализации динамического кодирования Хаффмана на языке C++. Была создана программа для реализации динамического кодирования Хаффмана, использующая, как рекурсивные функции, так и функции не рекурсивной природы.

ПРИЛОЖЕНИЕ С КОДОМ

main.cpp :

```
#include "headers.h"

int main()
{
    using namespace std;

    Node *a = new Node;
    pushFirst(a); // вставили начальный NYT
    string line = "", newLine = "";
    char ch;

    cout << "Enter symbol line" << endl;

     ch = cin.peek();    // записываем следующий знак из cin в переменную, не
изменяя cin
    if ((ch == '\n') || (ch == ' ') || (ch == '\t'))      // проверка на пустой
ввод
    {
        cout << "Nothing was entered. Finishing program..." << endl;
        deleteAllNodes();
        return 0;      // выход из программы
    }

    cin >> line; // считывается строка
    cin.ignore(32767,'\n'); // cin очищается от остатков

     for (int i = 0; i < line.size(); i++) // и для каждого символа вызывается
кодер
    {
        cout << '\n' << endl;
        string a = vitterCoder(line[i]);
        newLine.append(a);
    }

    cout << '\n' << endl;
     cout << "Final coded line:\n" << newLine << endl; // выводится итоговая
закодированная строка

    deleteAllNodes(); // освобождает память выделенную под дерево хаффмана
    return 0;
}
```

funcs.cpp:

```
#include "headers.h"

unsigned int count = 187; // ( = 2*кол-во символов алфавита - 1) нужно для
нумерования элементов дерева (94 печатных символа первой половины ascii)
bool showSymbolCodes = true; // false - чтобы, вместо кода символа выводилось: |
symbol| (легче проверять)

std::vector<Node*> leafs; // здесь хранятся все адреса на узлы-листья + NYT (на
нулевом месте)

std::vector<Node*> internalNodes; // здесь хранятся все адреса на внутренние
узлы (корень на нулевом месте)
```

```
void pushFirst(Node* a)
{
    leafs.push_back(a);
};

Node* findSymbol(char symbol, std::vector<Node*> *leafs) // ищет в переданном
векторе из уникальных листьев тот, что содержит нужный символ
{
     if (symbol == '\0') // \0 - обозначение для внутренних узлов, поэтому нет
уникального элемента с таким символом
        return nullptr;

    int lastInd = leafs->size();
     for (int i = 1; i < lastInd; i++) // идем по списку листьев и проверяем
значения символа внутри
    {
            if(((leafs->at(i))->m_symbol) == symbol) // если переданный символ
встречался раньше
        {
            return leafs->at(i);
        }
    }
    return leafs->at(0); // если же это новый символ, то вернем NYT
};

void swapNodes(Node *a, Node *b) // меняет 2 узла/листа местами с помощью замены
их собвственных указателей и указателей их родителей
{
      if (a->m_parent == nullptr || b->m_parent == nullptr) // если одно из
значений - корень
    {
        std::cout << "You can`t swap root." << std::endl;
        return;
    }

    if (a == b)
    {
        std::cout << "You can`t swap one element with itself." << std::endl;
        return;
    }


    if ((a->m_parent) == (b->m_parent)) // если у элементов один родитель
    {
        Node* parent = a->m_parent;
        if(a->m_parent->m_left == a) // если a - левый ребенок, a b - правый
        {
            a->m_parent->m_left = b;
            a->m_parent->m_right = a;
        }
        else // если же a - правый ребенок, a b - левый
        {
            a->m_parent->m_left = a;
            a->m_parent->m_right = b;
        }
        // менять указатели на родителей переданным элементам не нужно
    }
    else // если у элементов разные родители
    {
        // меняем указатели на детей в их родителях

        Node *tempVal; // временное место хранения значения
```

```
        tempVal = a->m_parent;
           if( (tempVal)->m_right == a) // если a - правый ребенок, то меняем
указатель родителя на правого ребенка
            (tempVal)->m_right = b;
        else // иначе меняем указатель на левого ребенка
            (tempVal)->m_left = b;


         tempVal = b->m_parent; // ошибка!!! У родителя b сейчас оба ребенка = b
ПОПРАВЬ!
        if( (tempVal)->m_right == b) // аналогично поступаем с родителем второго
узла
            (tempVal)->m_right = a;
        else
            (tempVal)->m_left = a;


        // меняем указатели на родителей в детях

        b->m_parent = a->m_parent;
        a->m_parent = tempVal;
    };

      if(b->m_isLeaf && a->m_isLeaf) // если это 2 листа, то можно спокойно
поменять у них номера на друг друга
    {
        unsigned int k = a->m_number;
        a->m_number = b->m_number;
        b->m_number = k;
    }
};

Node* findBlockLeader(std::vector<Node*> *vec, unsigned int weight) // находит
лидера блока (лидер блока: тот же вес, тот же тип (лист/вн.узел), максимальный
номер)
{ // после вызова сделай проверку на то, не является ли тот же элемент и лидером
(для лист-лист) и на nullptr (надо ли вообще делать swap)
    Node* p = nullptr;
    unsigned int number = 0;

    for (int i = 0; i < vec->size(); i++) // проходимся по всем элементам
    {
          if((vec->at(i))->m_weight == weight) // если вес элемента == нужному
весу
        {
            if(p == nullptr) // первый подходящий элемент
            {
                p = vec->at(i);
                number = p->m_number;
            }
             else if( (vec->at(i))->m_number > number) // если у элемента номер
больше
            {
                p = vec->at(i);
                number = p->m_number;
            }
        }
    }
    return p;
};
```

```
void encodeSymbol(std::string* emptyStr, Node* p) // записывает в переданную строку путь до листа/NYT
{
    Node* prevNode = p;

    while(prevNode->m_parent != nullptr) // пока не находимся в корне
    {
        if(prevNode->m_parent->m_left == prevNode) // если это левый ребенок
        {
            emptyStr->insert(0,"0"); // вставляем в начало 0
            prevNode = prevNode->m_parent;
        }
           else if(prevNode->m_parent->m_right == prevNode) // если это правый ребенок
        {
            emptyStr->insert(0,"1"); // вставляем в начало 1
            prevNode = prevNode->m_parent;
        }
        else
        {
                 std::cerr << "There is strange error in encodeSymbol()." << std::endl;
            return;
                // если это вывелось, значит где-то ошибка в создании дерева (указатель на ребенка пуст)
        }

    }
};

void splitHafTree(Node* root, std::vector<Node*> *vec, short int level) // запишет все листы и узлы в вектор по порядку КПЛ
{
    if (level == 1) // если это самый первый проход
    {
        vec->push_back(root); // то просто создаем первый блок
             vec->push_back(nullptr); // nullptr служат для разделения блоков (уровней дерева)
    }
    else
    {
        short int countOfNulls = 0; // для подсчета уровней
        auto k = vec->begin();
        while (k != vec->end()) // проходимся по всему вектору
        {
            if ((*k) == nullptr) // если встретили разделитель
            {
                countOfNulls += 1;
                   if (countOfNulls == level) // если количество разделителей = уровню функции
                {
                        vec->insert(k, root); // то записываем значение в конец нужного блока
                    break; // и останавливаем итерацию
                }
            }
            k++;
        }
         if(countOfNulls < level) // если в векторе меньше блоков, чем уровень у функции, то добавим еще один блок
        {
            vec->push_back(root);
            vec->push_back(nullptr);
```

```
        }
    }

       if ((root->m_left) != nullptr) // если узел внутренний, то делаем
рекурсивный вызов функции для детей узла
    {
        splitHafTree(root->m_right, vec, level + 1);
        splitHafTree(root->m_left, vec, level + 1);
    }
    return;
};

void remakeNumeration(Node* p) // передаем этой функции указатель на корень и
она расставит номера по КПЛ
{
    std::vector<Node*> vec;
     splitHafTree(p, &vec, 1); // теперь в vec упорядоченно лежат все узлы и
листья дерева
    unsigned int count = 187;

    auto iter = vec.begin();
      while(iter != vec.end()) // проходимся по всем узлам и выставляем там
соответствующие номера
    {
        if (*iter != nullptr) // пропускаем все разделители
        {
            (*iter)->m_number = count;
            count -= 1;
        }
        iter++;
    }

};

void deleteAllNodes()
{
    std::vector<Node*> *allLeafs = &leafs;
    std::vector<Node*> *allIntNodes = &internalNodes;

    auto l = allLeafs->begin();
    auto iN = allIntNodes->begin();

    while(l != allLeafs->end())
    {
        delete *l;
        l++;
    }
    while(iN != allIntNodes->end())
    {
        delete *iN;
        iN++;
    }
};

void fillSymbolCode(char symbol, std::string *str) // записывает ascii-номер
символа в строку
{
    int c = (int)symbol;
    str->push_back('0');
    for(int j = 64; j > 1; j = j/2)
    {
        if(c > j)
        {
```

```
            c -= j;
            str->push_back('1');
        }
        else
        {
            str->push_back('0');
        }
    }
    if(c == 1)
        str->push_back('1');
    else
        str->push_back('0');
};


Node* slideAndIncrement(Node* p)
{
    if(!(p->m_isLeaf)) // если это внутренний узел
    {
        Node* previousP = p->m_parent;

        // сдвигаем p в дереве выше, чем узлы-листья с весом w+1
        Node* a = findBlockLeader(&leafs, (p->m_weight) + 1);
        if(a != nullptr) // если такие листья есть
        {
              swapNodes(a, p); // т.к. здесь swap листа с внутренним узлом, то
swap не поменяет автоматически внутреннюю нумерацию

                  Node* p = internalNodes.at(0); // на первом месте в списке
внутренних узлов стоит корень
            remakeNumeration(p); // пересоздаем нумерацию искуственно, начиная с
корня
        }

        p->m_weight += 1;
        return previousP;
    }
    else // если же это лист
    {
        // сдвигаем p в дереве выше, чем внутренние узлы с весом w
        Node* b = findBlockLeader(&internalNodes, p->m_weight);
        if (b != nullptr)
        {
            swapNodes(p, b);
            remakeNumeration(internalNodes.at(0));
        }

        p->m_weight += 1;
        return p->m_parent;
    }
};

std::string vitterCoder(char symbol)
{
    std::cout << "Start coding symbol \'" << symbol << "\'" << std::endl;
    std::string code;
    Node* nodeForIncrease = nullptr;
    Node* p = findSymbol(symbol, &leafs); // смотрим не встречался ли нам такой
символ раньше

    if (p->m_isNYT) // если этот символ встретился впервые
    {
        std::cout << "It is the first meeting with this symbol." << std::endl;
```

```
        // Кодирование символа:

        encodeSymbol(&code, p); // записывает в строку путь до NYT

        std::cout << "Way to the NYT = " << code;
        if (code == "")
        {
            std::cout << '-';
        }
        std::cout << std::endl;

          if(!(::showSymbolCodes)) // записывает в строку |symbol| вместо кода
символа, если это указано пользователем
        {
            std::string symbolCode = "|";
            symbolCode.push_back(symbol);
            symbolCode.push_back('|');

               code.append(symbolCode); // добавляет в конец строки кодирования
символа |symbol|
        }
        else // записывает в строку соответствующий код символа
        {
            std::string symbolCode = "";
            fillSymbolCode(symbol, &symbolCode);
            code.append(symbolCode);
        }


        // Перестройка дерева:
        std::cout << "Adding 2 new nodes from NYT to the tree." << std::endl;
        p->m_left = new Node(p); // создаем новый NYT
        p->m_right = new Node(p, symbol); // и создаем новый лист
           p->m_isNYT = false; // в узле, где мы находимся, указываем, что он
больше не NYT
           nodeForIncrease = p->m_right; // указываем, что лист был только что
создан, и его вес нужно увеличить

          leafs.push_back(p->m_right); // записываем новосозданный лист в конец
вектора листьев
        leafs.at(0) = p->m_left; // меняем указатель на новый NYT в векторе

         internalNodes.push_back(p); // так как старый NYT стал внутренним узлом
сохраним его в соответствующий вектор
    }
    else // если же символ встречался ранее
    {
        std::cout << "We saw this symbol earlier." << std::endl;
        // Кодирование символа:

        encodeSymbol(&code, p); // записывает в строку путь до листа
        std::cout << "Way to the leaf with this symbol = " << code << std::endl;

        // Перестройка дерева:

        std::cout << "Swap this leaf with leader of the block." << std::endl;
        // swap лист с тем же элементов с листом-лидером блока (блок - один тип,
один вес)
        Node* k = findBlockLeader(&leafs, p->m_weight);
         if (k != p) // на nullptr проверка здесь не нужна т.к. гарантированно
есть хотя бы 1 элемент такого веса (сам p)
           swapNodes(p, k);
```

```
          if ( p->m_parent->m_left->m_isNYT ) // если p после перемещения все равно брат NYT (это нужно во избежания пары проблем с slideAndIncrement() )
        {
            nodeForIncrease = p; // лист будет увеличен в конце отдельно
                p = p->m_parent;     // а цикличное увеличение начнется с его родителя
        }
    }

     std::cout << "Increment weight to needed leaf and to all his parents with slidings." << std::endl;
    while (p != nullptr) // увеличиваем вес и делаем необходимые сдвиги в дереве от p до корня
    {
        p = slideAndIncrement(p);
    };

     if (nodeForIncrease != nullptr) // делаем еще одно увеличение и сдвиг в дереве, если это необходимо
    {
        slideAndIncrement(nodeForIncrease);
    };

    return code; // возвращаем строку по значению
};
```

headers.h :

```
#ifndef HEADERS_H
#define HEADERS_H

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

// описывает узел дерева
class Node
{
public:
    // обеспечивают построение дерева Хаффмана с другими узлами
    Node* m_parent = nullptr;
    Node* m_right = nullptr;
    Node* m_left = nullptr;

    // характеристики узла
    unsigned int m_weight = 0; // вес узла
    unsigned int m_number = 187; // 94*2 - 1
    char m_symbol = '\0'; // символ который хранится в листе
```

```
    bool m_isLeaf = false; // флаг листа (TRUE, если лист)
    bool m_isNYT = true; // является ли узел NYT

    Node() = default; // базовый конструктор (исп. для корня)
     Node(Node* parent): m_parent{parent}, m_number{(parent->m_number) - 2} //
конструктор для NYT с родителем
    {};
        Node(Node* parent, char symbol): m_parent{parent}, m_symbol{symbol},
m_isNYT{false}, m_isLeaf{true}, m_number{(parent->m_number) - 1} // конструктор
для листьев
    {};
};

void pushFirst(Node* a);
void deleteAllNodes();

std::string vitterCoder(char symbol);

#endif
```