МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
КАФЕДРА МОЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Динамическое кодирование и декодирование по Хаффману - демонстрация

Студент гр. 9303 _____ Дюков В.А.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург
2020

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Дюков В.А.

Группа 9303

Тема работы: Динамическое кодирование и декодирование по Хаффману – демонстрация

Исходные данные:
Произвольный поток символов

Предполагаемый объем пояснительной записки:
Не менее 10 страниц.

Дата выдачи задания: 6.11.2020

Дата сдачи реферата: 25.12.2020

Дата защиты реферата: 25.12.2020

Студент _____ Дюков В.А.

Преподаватель _____ Филатов А.Ю.

АННОТАЦИЯ

В данной работе были реализованы динамическое кодирование и динамической декодирование по Хаффману. Были созданны методы реализующие демонстрацию работы программы на разных этапах её выполнения, позволяющие лучше понять работу программы. Были представленны тесты программы.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цели исследования:

- Реализовать алгоритмы кодирования и декодирования динамическим методом Хаффмана.
-

План экспериментального исследования:

1. Реализовать динамический метод кодирования и декодирования Хаффмана.
2. Реализовать методы демонстрирующие промежуточные состояние программы.

1. ПЕРВЫЙ РАЗДЕЛ

1.1. Общие сведения

Динамического кодирование Хаффмана — адаптивный метод, основанный на кодировании Хаффмана. Он позволяет строить кодовую схему в поточном режиме (без предварительного сканирования данных), не имея никаких начальных знаний из исходного распределения, что позволяет за один проход сжать данные. Данный метод позволяет динамически регулировать дерево Хаффмана, не имея начальных частот. В дереве кодировании Хаффмана есть особый внешний узел, называемый 0-узел, используемый для идентификации входящих символов. То есть, всякий раз, когда встречается новый символ — его путь в дереве начинается с нулевого узла. Самое важное — то, что нужно усекать и балансировать дерево кодирования Хаффмана при необходимости, и обновлять частоту связанных узлов. Как только частота символа увеличивается, частота всех его родителей должна быть тоже увеличена. Это достигается путём последовательной перестановки узлов, поддеревьев или и тех и других. Важной особенностью дерева кодированияв в динамическом методе Хаффмана является принцип братства (или соперничества): каждый узел имеет два потомка (узлы без потомков называются листами) и веса идут в порядке убывания.

2. ВЫПОЛНЕНИЕ РАБОТЫ

2.1. Общие методы

Для выполнения работы были созданны классы:

- *Tree* – класс дерева
- *TreeList* – класс – интерфейс, отвечающий за обработку дерева: добавление ветвей, перестроение дерева, изменение весов и отрисовку дерева. Методы print() и add() сделанны виртуальными и определены в классах в следующих классах.
- *EncodeTreeList* – класс, реализующий работу интерфейса *TreeList*. Его работа описана в разделе 2.2.
- *DecodeTreeList* – класс, реализующий работу интерфейса *TreeList*. Его работа описана в разделе 2.3.

Метод класса *TreeList* – *rebuild()* отвечает за перестройку дерева. В списке (в котором хранится все узлы дерева в поядке убывания), происходит обход всех узлов с конца. Если текущий элемент списка меньше предыдущего (1), то условие упорядоченности нарушено. В таком случае, обход в списке продолжается до тех пор, пока не найдётся первый элемент, который не меньше, чем (1), и элемент (1) меняются данными с предыдущим, только-что найденного.

Метод класса *TreeList* – *drawTree()* отвечает за вывод дерева в текущем виде в консоль. Метод изображает состояние узлов дерева, выводя символ, который они содержат и вес узла.

2.2. Кодирование данных

За кодирование данных отвечает метод *add()* переопределённый в классе *EncodeTreeList* , наследуемом от *TreeList*.

На вход метод принимает ссылку на входной и выходной потоки данных. Метод считывает один символ из входного потока и проверяет, встречался ли этот символ ранее. Если да, то вес узла, содержащего этот символ, и его родителей, возрастает на один, до самого корня дерева. Если ранее символ не встречался, то метод находит в списке (в котором хранится все узлы дерева в поядке убывания) esc-символ и добавляет ему правую и левую ветви, где в правой ветви будет хранится новый символ, а в левой – esc-символ.

Далее в методе происходит вызов метода *rebuild()* и затем метода *print(),* который записывет либо код символа в дереве (если сивол ранее встречался), либо код до esc-символа куда был добавлен новый символ, и ASCII код самого символа (если сивол встретился впервые).

2.3. Декодирование данных

За кодирование данных отвечает метод *add()* переопределённый в классе *DecodeTreeList* , наследуемом от *TreeList*.

На вход метод принимает ссылку на входной и выходной потоки данных. Если дерево пустое, то метод считывает 8 символов из потока, преобразует их в символ и добавляет в дерево. Если дерево не пусто, метод считывает символы из входного потока и прохит по дереву в соответствии с ними, пока не упрётся в лист. Если код листа – не esc-символ, то вес этого листа, и его родителей, возрастает на один, до самого корня дерева. Иначе, метод считывает 8 символов из потока, преобразует их в символ. Затем добавляет найденному листу правую и левую ветви, где в правой ветви будет хранится новый символ, а в левой – esc-символ.

Далее в методе происходит вызов метода *rebuild()* и затем метода *print(),* который записывет декодированный символ в поток.

ТЕСТИРОВАНИЕ

№	Входные данные	Результат	Вывод
1	shhooowwweel	0111001100110100001000110111 1001111000111011110011111100 0110010100010100011011 00	верно
2	hello	0110100000110010100011011001 0111001101111	верно
3	0111001100110100001000110111 1100111100011101111001111111 0001100101000101000110110 0	shhooowwweel	верно
4	0011001111000110010010000110 0011001001	333221311	верно

заключение

В ходе выполнения работы были реализованы динамическое кодирование и декодирование методом Хаффмана. Были реализованны методы, демонстрирующие работу программы. В ходе выполнения программы, были изучены методы обработки бинарных деревьев и способы вывода бинарных деревьев в удобном виде.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Файл main.cpp

```
#include <fstream>
#include <conio.h>
#include <Windows.h>
#include "TreeList.h"

int button_get(std::string* buttons, int size) {

 int count = 0;

 while (1) {

	std::cout << '\r';

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

		if (i == count) std::cout << '<' << buttons[i] << '>' << '\t';
		else std::cout << ' ' << buttons[i] << ' ' << '\t';
	}

	unsigned char key = _getch();
	if (key == 224) key = _getch();

	switch (key) {
	case 75:
		count--;
		break;
	case 77:
		count++;
		break;
	case 13:
```

```
            return count + 1;
        case 27:
            return 0;
        }
        if (count > size - 1) count = 0;
        if (count < 0) count = size - 1;
  }
}

//

int main() {

  std::ifstream in;
  std::ofstream out;
  TreeList* huffman;
  std::string str;
  std::string empty[] = { "" };
  std::string coding[] = { "Encode", "Decode" };

  while (1) {

        huffman = nullptr;

        system("cls");
        std::cout << "Choose what you want to do:\n\n";
        int ch = button_get(coding, 2);

        if (ch == 1) huffman = new EncodeTreeList;
        if (ch == 2) huffman = new DecodeTreeList;

        if (huffman) {

            std::cout << "\n\nEnter the source data file:\n\n";
            std::cin >> str;
            in.open(str);
```

```
        }

        if (in.is_open()) {

              std::cout << "\n\nEnter the file where the result will be
saved:\n\n";
              std::cin >> str;
              out.open(str);
        }

        while (huffman && in.is_open() && out.is_open()) {

              system("cls");
              huffman->drawTree();
              button_get(empty, 0);
              if (!huffman->add(in, out)) break;
        }

        if (huffman) delete huffman;
        if (in.is_open()) in.close();
        if (out.is_open()) out.close();

        std::cout << "\n\nPress 'Esc' to exit / 'Enter' to continue.\n";
        if (!button_get(empty, 0)) break;
  }

  return 0;
}
```

Файл Tree.h

```
#pragma once

class Tree {
```

```
public:
  Tree* left;
  Tree* right;
  Tree* parent;
  char data;
  char code;
  int weight;
  int depth;

  Tree(char _d, char _c, Tree* _p, int _t) : data(_d), weight(0), left(nullptr),
right(nullptr), parent(_p), code(_c), depth(_t) {}

  void operator++ (int);
  void operator-- (int);
  void inc_dep(int);
  void dec_dep(int);
};
```

Файл Tree.cpp

```
#include "Tree.h"

void Tree::operator++ (int) {

  weight++;
  if (parent) (*parent)++;
}

void Tree::operator-- (int) {

  weight--;
  if (parent) (*parent)--;
}

void Tree::inc_dep(int a) {
```

```
 depth += a;
 if (left) left->inc_dep(a);
 if (right) right->inc_dep(a);
}

void Tree::dec_dep(int a) {
 depth -= a;
 if (left) left->dec_dep(a);
 if (right) right->dec_dep(a);
}
```

Файл TreeList.h

```
#pragma once

#include <iostream>
#include <list>

#define ESC 0

class Tree;

class TreeList {

public:
 std::list<Tree*> list;

 TreeList();
 virtual ~TreeList();

 virtual bool add(std::istream&, std::ostream&) = 0;
 void rebuild();
 void drawTree();
 virtual void print(std::ostream& out, char sym, bool first = false) = 0;
```

```
};

class EncodeTreeList : public TreeList {

public:
 EncodeTreeList() : TreeList() {};

 virtual bool add(std::istream&, std::ostream&);
 virtual void print(std::ostream& out, char sym, bool first = false);
};

class DecodeTreeList : public TreeList {

public:
 DecodeTreeList() : TreeList() {};

 virtual bool add(std::istream&, std::ostream&);
 virtual void print(std::ostream& out, char sym, bool first = false);
};
```

Файл TreeList.cpp

```
#include "TreeList.h"
#include "Tree.h"

std::string ItoS(int num) {

 int sym;
 std::string str;

 if (!num) str = "0";

 while (num) {

       sym = num % 10;
```

```
        if (sym == 0) str += '0';
        if (sym == 1) str += '1';
        if (sym == 2) str += '2';
        if (sym == 3) str += '3';
        if (sym == 4) str += '4';
        if (sym == 5) str += '5';
        if (sym == 6) str += '6';
        if (sym == 7) str += '7';
        if (sym == 8) str += '8';
        if (sym == 9) str += '9';

        num = num / 10;
  }

  for (int i = 0; i < str.length() / 2; i++) {

        char reserve = str[i];
        str[i] = str[str.length() - i - 1];
        str[str.length() - i - 1] = reserve;
  }

  return str;
}

// Interface

TreeList::TreeList() {

  list.push_back(new Tree(ESC, '.', nullptr, 1));
}

TreeList::~TreeList() {

  for (std::list<Tree*>::iterator i = list.begin(); i != list.end(); i++)
        delete* i;
```

```
}

void TreeList::rebuild() {

  Tree* prev = *(--list.end());
  Tree* max = nullptr;

  for (std::list<Tree*>::iterator i = --list.end(); true; i--) {

        if (prev->weight > (*i)->weight) max = prev;
        if (max && (max->weight <= (*i)->weight)) {

              Tree test = *max;

              // max restore
              max->left = prev->left;
              if (max->left) max->left->parent = max;
              max->right = prev->right;
              if (max->right) max->right->parent = max;
              max->data = prev->data;
              max->dec_dep(max->depth - prev->depth);
              (*max)--;

              // prev restore
              prev->left = test.left;
              if (prev->left) prev->left->parent = prev;
              prev->right = test.right;
              if (prev->right) prev->right->parent = prev;
              prev->data = test.data;
              prev->inc_dep(prev->depth - test.depth);
              (*prev)++;

              rebuild();

              break;
        }
```

```
        prev = *i;

        if (i == list.begin()) break;
  }
}

void TreeList::drawTree() {

  int tree_on_lvl = 1;
  Tree** trees = new Tree*[1];
  trees[0] = *list.begin();
  int depth = (*(--list.end()))->depth;

  while (depth) {

        int num = 0;
        for (int i = 0; i < depth - 1; i++)
              num = num + 3 * pow(2, i);

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


              for (int j = 0; j < num; j++) std::cout << ' ';
              if (trees[i] != nullptr) {

                    if (trees[i]->data == ESC) std::cout << "[~ ";
                    else std::cout << "[" << trees[i]->data << " ";
                    std::cout << trees[i]->weight << "] ";
              }
              else std::cout << "      ";
              for (int j = 0; j < num; j++) std::cout << ' ';
        }
        depth--;
        std::cout << "\n\n\n";

        tree_on_lvl = tree_on_lvl * 2;
```

```
        Tree** n_trees = new Tree * [tree_on_lvl];
        for (int i = 0; i < tree_on_lvl; i++) {

              if (trees[i / 2] == nullptr) n_trees[i] = nullptr;
              else {

                    if (i % 2) n_trees[i] = trees[i / 2]->right;
                    else n_trees[i] = trees[i / 2]->left;
              }
        }
        delete[] trees;
        trees = n_trees;
  }
}

// Encode

bool EncodeTreeList::add(std::istream& in, std::ostream& out) {

  bool new_symb = true;
  char symb;
  in.get(symb);
  if (in.eof()) return false;

  for (std::list<Tree*>::iterator i = list.begin(); i != list.end(); i++)
        if ((*i)->data == symb) {

              (*(*i))++;
              new_symb = false;
        }

  if (new_symb) {

        Tree* esc = *(--list.end());
        esc->right = new Tree(symb, '1', esc, esc->depth + 1);
        (*esc->right)++;
```

```
        esc->left = new Tree(ESC, '0', esc, esc->depth + 1);
        list.push_back(esc->right);
        list.push_back(esc->left);
  }

  print(out, symb, new_symb);
  rebuild();

  return true;
}

void EncodeTreeList::print(std::ostream& out, char sym, bool first) {

  std::string code = "";
  Tree* end = nullptr;
  for (std::list<Tree*>::iterator i = list.begin(); i != list.end(); i++) {

        if ((*i)->data == sym) {

              end = (*i);
              break;
        }
  }

  if (end) {

        if (first) end = end->parent;

        for (end; end->parent; end = end->parent) {

              code += end->code;
        }

        for (int i = 0; i < code.length() / 2; i++) {

              char reserve = code[i];
```

```
            code[i] = code[code.length() - i - 1];
            code[code.length() - i - 1] = reserve;
        }

        out << code;

        if (first) {

            int k = 128;
            for (int i = 7; i > -1; i--) {

                out << ((sym & k) >> i);
                k = k >> 1;
            }
        }
  }
}

// Decode

bool DecodeTreeList::add(std::istream& in, std::ostream& out) {

  char res = 0;
  Tree* tree = *list.begin();

  while (tree && tree->left) {

        in.get(res);
        if (in.eof()) return false;
        switch (res) {
        case 49:
            tree = tree->right;
            break;
        case 48:
            tree = tree->left;
            break;
```

```
        default:
                tree = nullptr;
                break;
        }
  }

  if (!tree) return false;
  if (tree->data != 0) (*tree)++;
  else {

        res = 0;
        for (int i = 0; i < 8; i++) {

                res = res * 2 + in.get() - 48;
                if (in.eof()) return false;
        }

        Tree* esc = *(--list.end());
        esc->right = new Tree(res, '1', esc, esc->depth + 1);
        (*esc->right)++;
        esc->left = new Tree(ESC, '0', esc, esc->depth + 1);
        list.push_back(esc->right);
        list.push_back(esc->left);
        tree = tree->right;
  }

  print(out, tree->data, 0);
  rebuild();

  return true;
}

void DecodeTreeList::print(std::ostream& out, char sym, bool first) {

  out << sym;
}
```