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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных

Тема: Сравнительное исследование динамического и статического метода Хаффмана

Студент гр. 9303 _____ Ахримов А.М.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Ахримов А.М.

Группа 9303

Тема работы: Динамическое кодирование и декодирование по Хаффману – сравнительное исследование со “статическим” методом.

Исходные данные: произваольный набор символов.

Предполагаемый объем пояснительной записки:
Не менее 10 страниц.

Дата выдачи задания: 6.11.2020

Дата сдачи реферата: 18.12.2020

Дата защиты реферата: 18.12.2020

Студент _____ Ахримов А.М.

Преподаватель _____ Филатов А.Ю.

АННОТАЦИЯ

В данной работе проводилось сравнительное исследование двух методов кодирования и декодирования по Хаффману: статического и динамического. Была экспериментально установлена сложность алгоритмов на основе подсчета базовых операций. В работе представлены графики, основанные на подсчете базовых операций. Было реализовано динамическое кодирование и декодирование по Хаффману, а также программы для генерации наборов входных данных.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цели исследования:

- Реализовать алгоритмы кодирования и декодирования динамическим методом Хаффмана.

- Провести сравнительный анализ динамического и статического алгоритмов.

Исследование проводилось со случайно сгенерированными наборами данных.

План экспериментального исследования:

1) Реализовать динамический метод кодирования и декодирования Хаффмана.

2) Реализовать программу для генерации набора данных для среднего и худшего случая.

3) Выбрать базовые операции и реализовать автоматический подсчёт выбранных операций в статическом и динамическом алгоритме кодирования по Хаффману.

4) На основе полученных результатов составить графики зависимости количества базовых операций от набора входных данных.

5) Сделать выводы по полученным графикам, оценить сложность алгоритмов, сопоставить с теоретической оценкой.

1. ОПИСАНИЕ ДИНАМИЧЕСКОГО МЕТОДА ХАФФМАНУ

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.

1.2. Реализация

В данной реализации созданы два класса: Node – узлы дерева и BinTree – бинарное дерево кодирования. Объект класса Node может хранить в себе вес узла, символ(если узел является листом бинарного дерева), указатели на левое поддерево, правое поддерево и родителя. Также в этом классе реализованы конструкторы и метод swap, меняющий местами два узла.

Класс BinTree может хранить указатель на корень бинарного дерева и на 0-узел. Основные методы данного класса – coding и decoding. Данные методы соотвесственно кодируют и декодируют сообщение. Рассмотрим данные методы.

Метод coding при использовании консоли берёт символы из терминала до первого перевода строки. Цикл работы метода такой:

1. Проверить, встречался ли раньше символ, функцией check_sym. Если символ уже встречался функция check_sym увеличить вес соотвествующего узла.

2. Если символа нет в бинарном дереве, то функция find_zero возвращает код пути до 0-узла, а функция addNode создаёт новый узел с новым символом.

3. Функция refresh_tree() проверяет веса остальных узлов и при необходимости обновляет веса родителей и меняет местами узлы, чтобы дерево оставалось упорядоченным.

Метод decoding берёт символы из терминала до певого символа не являющегося 0 или 1. Цикл работы метода такой:

1. Вызывается функция follow_code, которая по поступающим символам ведёт поиск по дереву. Если программа дошла до листа дерева, то вес данного узла увеличивается, в декодированную строку добавляется символ, хранящийся в узле. Если программа дошла до 0-узла, создаётся новый узел, а следуюшие 8 символов интерпритируется как код символа из ascii.

2. Выполняется функция refresh_tree, которая проверяет веса остальных узлов и при необходимости обновляет веса родителей и меняет местами узлы, чтобы дерево оставалось упорядоченным.

Методы coding и decoding возвращают строку — закодированное или декодированное сообщение.

2. СРАВНИТЕЛЬНОЕ ИССЛЕДОВАНИЕ ДИНАМИЧЕСКОГО И СТАТИЧЕСКОГО МЕТОДА ХАФФМАНА

2.1. Теоретическая оценка сложности алгоритмов

Основные затраты статического метода Хаффмана связаны с сортировкой массива. В данной реализации алгоритма используется быстрая сортировка, у которой в среднем случае сложность $O(nlogn)$, а в худшем - $O(n^2)$.

В динамическом алгоритме Хаффмана основные затраты связаны со скрутурой хранения кодов. В каждой итерации требуется запускать несколько поисков в глубину, чтобы проверить упорядоченность дерева или чтобы обновить веса узлов. Так как сложность поиска в глубину равна $O(n)$, то общая сложность алгоритма будет составлять $O(n^2)$ (для худшего и среднего случая).

2.2. Генерация представительного набора входных данных

Для оценки среднего случая брался набор символов с кодами ascii от 65 до 122, длиной от 2 до 1000. Для генерации таких наборов данных была написана программа random. Программа использует функции time() из библиотеки ctime, srand() и rand() из cstdlib для генерации набора символов заданной длины n. Реализацию данной программы см. в приложении Б.

Так как от количества уникальных символов напрямую увеличивается деревья кодирования, то для динамического и статического кодирования худшим случаем будет тот, когда в наборе данных нет одинаковых элементов. При этом для усложнения алгоритма сортировки в статическом методе Хаффмана символы идут в порядке убывания их кодов из ascii. Для генерации такого набора была написана функция random_worst. Реализацию данной программы см. в приложении В.

2.3. Практическая оценка алгоритмов

Для практического анализа алгоритмов выберем базовые операции и реализуем в алгоритмах подсчёт данных операций. Выбранные базовые операции: операции сравнения (кроме счетчиков в циклах), добавления в конец/начало массива, удаление элемента из начала/конца массива, доступ к элементу массива по индексу (массив – vector из STL). Для подсчёта была введена глобальная переменная counter_procedure.

Далее приведены некоторые графики, полученные на основе результатов алгоритмов при среднем и худшем случае.

Рассмотрим статический метод Хаффмана. График зависимости числа базовых операций от количества наборов данных представлен на рис.1. Черная пунктирная линия представляет собой график по полученным результатам алгоритма. Синяя линия - график $xlogx$, красная - x^2. Исходя из графика, можно оценить сложность алгоритма для среднего случая как $O(nlogn)$.

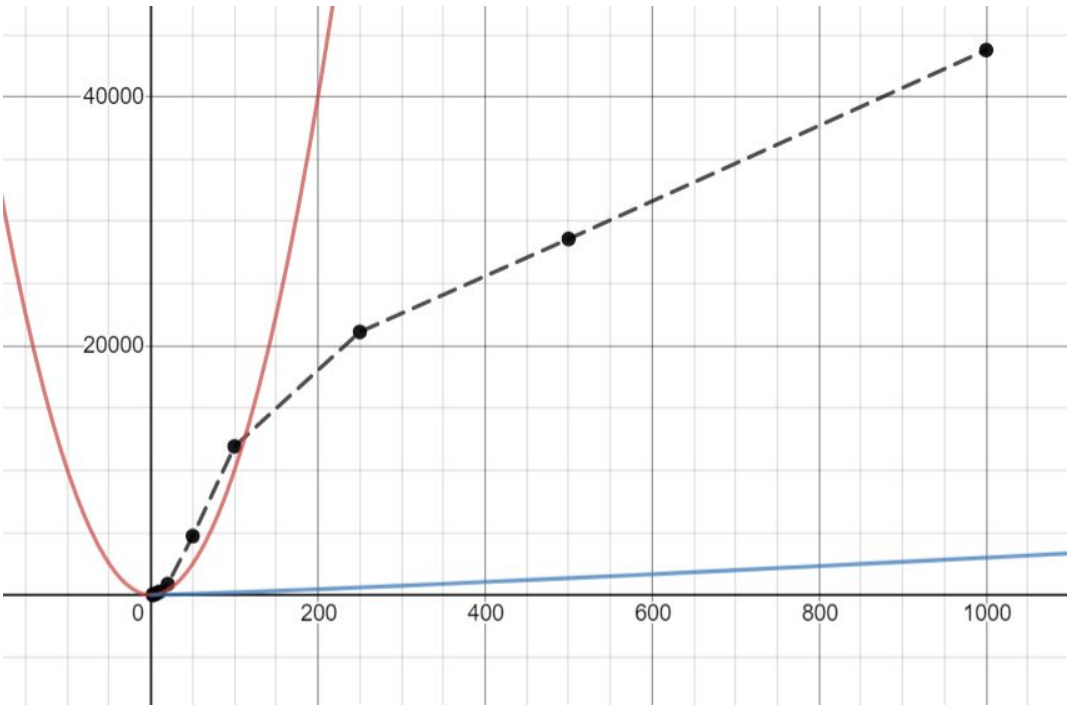

Рисунок 1 – График для статического метода Хаффмана.

Для худшего случая график представлен на рис. 2. Черная линия — результат при худшем случаи, пунктирная зеленая — средний. По этому графику можно убедиться, что сложность алгоритма Хаффмана в худшем случае - $O(n^2)$.

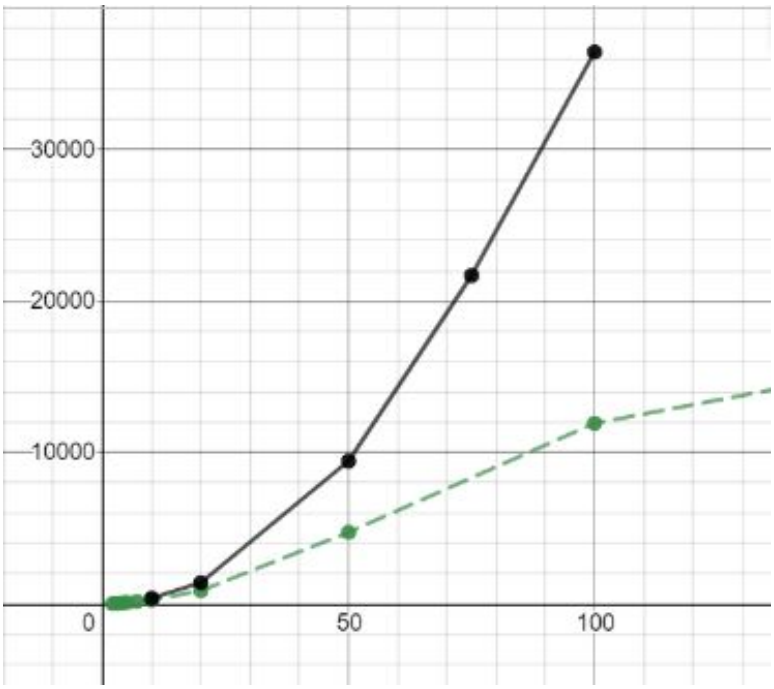

Рисунок 2 — График сравнения худшего и среднего случая статического метода Хаффмана

Далее рассмотрим динамический метод Хаффмана. График зависимости числа базовых операций от количества наборов данных представлен на рис. 3. Черная пунктирная линия представляет собой график по полученным результатам алгоритма. Фиолетовая линия - график x^3 , красная - x^2 . Исходя из графика, можно оценить сложность алгоритма для среднего случая как $O(n^2)$.

Для худшего случая график представлен на рис. 2. Черная линия — результат при худшем случаи, пунктирная зеленая — средний. По этому

графику можно убедиться, что сложность алгоритма Хаффмана в худшем случае также $O(n^2)$.

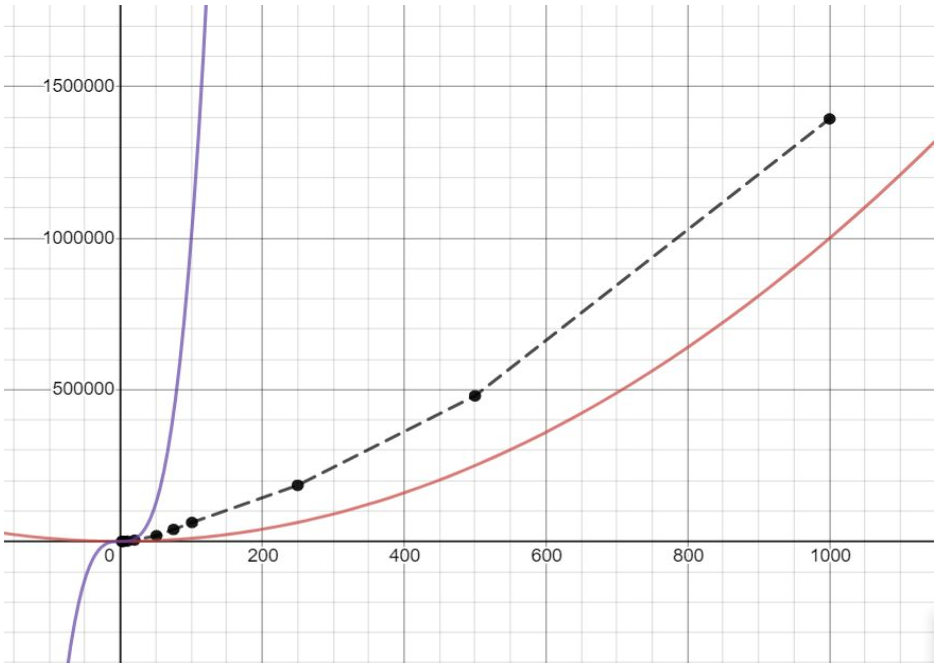

Рисунок 3 – График для динамического метода Хаффмана.

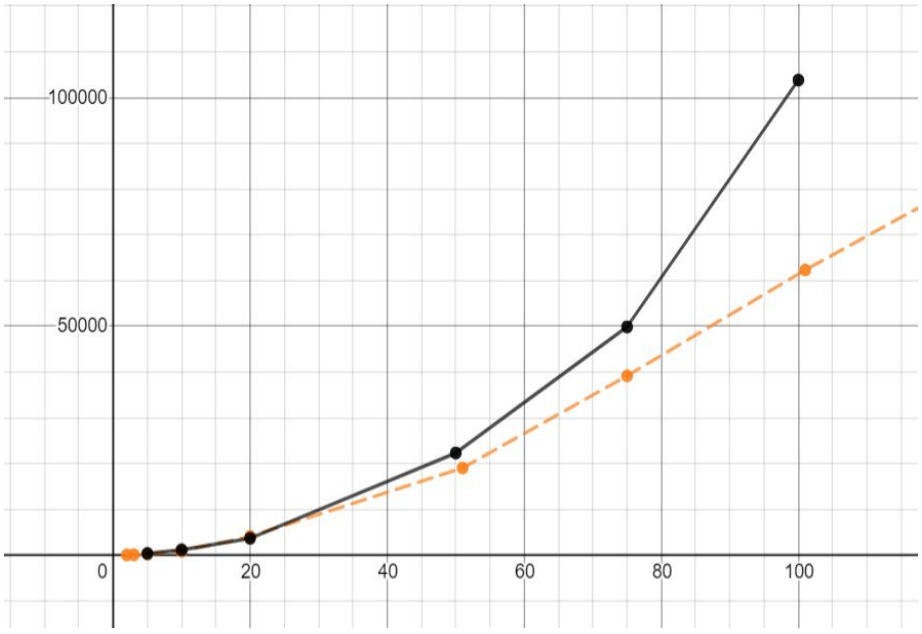

Рисунок 4 – Сравнения наихудшего и среднего случая дли динамического метода Хаффмана.

Сравним графики динамического и статического кодирования Хаффмана и убедимся, что статический алгоритм в среднем случае работает эффективнее, чем динамический (зеленая линия — статический метод, оранжевая — динамический).

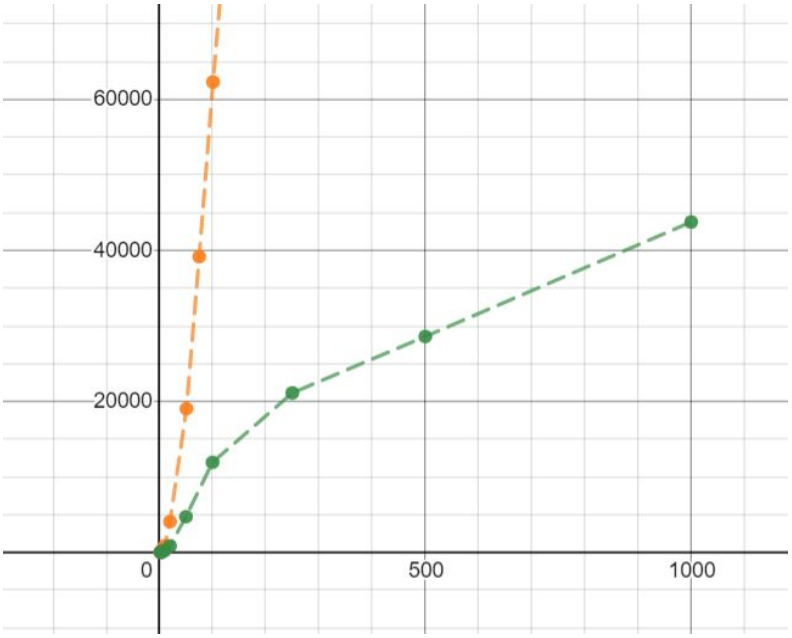

Рисунок 5 — Сравнение динамического и статического алгоритма Хаффмана.

Теперь рассмотрим декодирование по статическому методу Хаффмана. Так как у нас уже есть готовое дерево, все что остаёться программе — построить дерево по заднной форме и пройтись по нему в зависимости от закодированного сообщения. Поиск в глубину имеет сложность порядка $O(n)$, следовательно алгоритм декодирования иммет схожую сложность. График иллюстрирующий зависимость количетсва базовых операций от размеров входного набора см. на рис. 6.

Для динамического метода Хаффмана сложность декодирования будет совпадать со сложностью кодирования, так как декодировщику приходиться делать ту же работу, что и кодировщику. График с декодированием по Хаффману см. на рис.7.

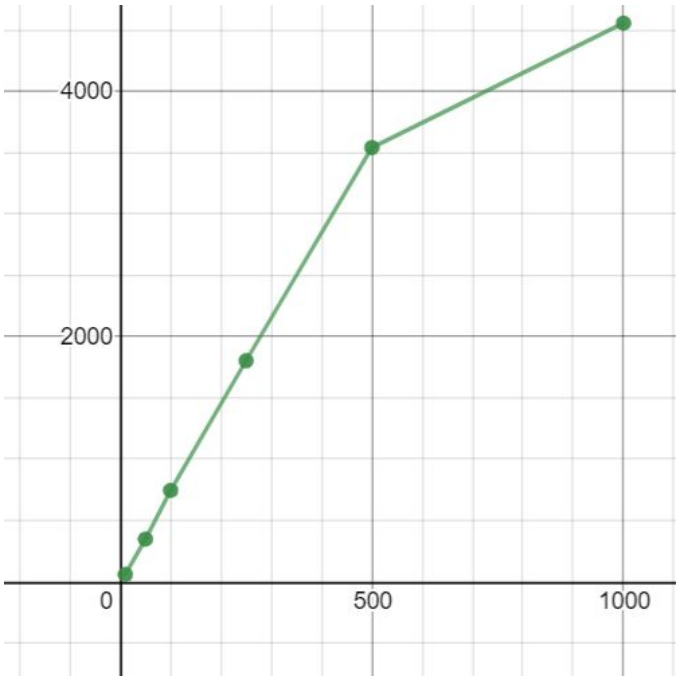

Рисунок 6 — График декодирования статического метода Хаффмана.

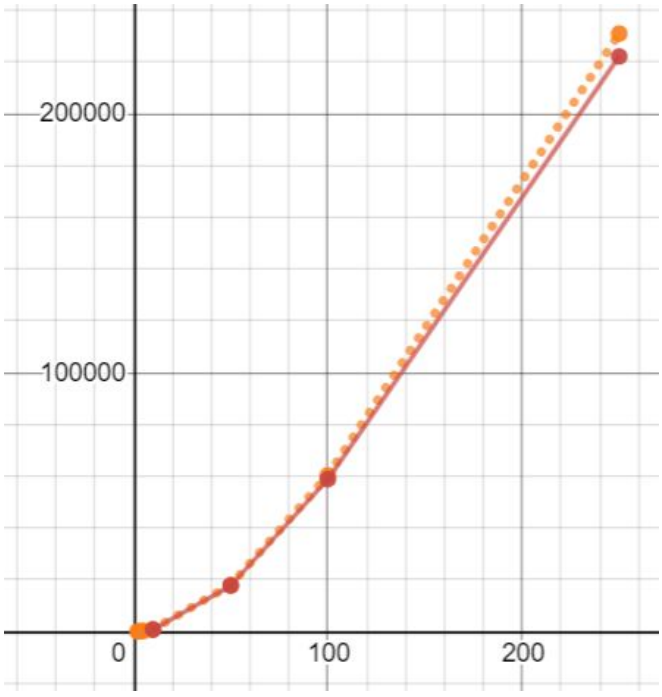

Рисунок 7 — График декодирования динамического метода Хаффмана.

Тестирование.

Результаты тестирования для динамического кодирования методом Хаффмана представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1.	abadab!	abadab! \1/ zero = 0 a = 1 //////// \2/ \1/ zero = 0 b = 1 a = 1 //////// \3/ \1/ zero = 0 b = 1 a = 2 //////// \4/ \2/ \1/ zero = 0 d = 1

		b = 1 a = 2 //////// \5/ \2/ \1/ zero = 0 d = 1 b = 1 a = 3 //////// \6/ \3/ \1/ zero = 0 d = 1 b = 2 a = 3 //////// \7/ a = 3 \4/ \2/ \1/ zero = 0 ! = 1 d = 1 b = 2

		////////// time = 19630.2 0110000100110001010001100100101000 00100001 counter = 207
2.	0110000100110001010001100100101000 0100000100001	\1/ zero = 0 a = 1 ////////// \2/ \1/ zero = 0 b = 1 a = 1 ////////// \3/ \1/ zero = 0 b = 1 a = 2 ////////// \4/ \2/ \1/ zero = 0 d = 1 b = 1 a = 2 ////////// \5/ \2/

		\1/ zero = 0 d = 1 b = 1 a = 3 //////// \6/ \3/ \1/ zero = 0 d = 1 b = 2 a = 3 \7/ a = 3 \4/ \2/ \1/ zero = 0 ! = 1 d = 1 b = 2 \7/ a = 3 \4/ \2/ \1/ zero = 0 ! = 1 d = 1 b = 2 abadab! counter = 184

Заключение

В ходе выполнения работы было проведено сравнительное исследование динамического и статического метода Хаффмана. Был реализован динамический метод Хаффмана, а также создана программа для генерации случайных наборов данных. На основе результатов подсчёта базовых операций эксперимательно оценены сложности данных алгоритмов, которые совпали с теоретическими оценками.

По полученным результатам можно сделать вывод о том, что динамический метод менее эффективный, чем статический. Тем не менее динамический метод поддерживает кодирование в поточном режиме, что можеть быть необходимо в некоторых ситуациях.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <queue>
#include <limits>
#include <cmath>


using namespace std;

int counter_procedure;

string reverse_string(string s) {
string copy = s;
for (int i = s.size() - 1, j = 0; i >= 0; i--, j++)
      copy[j] = s[i];
return copy;
}

string binary_code(int a) {
string s;
while (a > 0) {
      if (a % 2)
            s += "1";
      else
            s += "0";
      a = a / 2;
}
if (s.size() < 8) {
      int size = s.size();
      for (int i = 0; i < (8 - size); i++)
            s += "0";
}
return reverse_string(s);
}

int from_binary(string s) {
int a = 0;
for (int i = s.size() - 1; i >= 0; i--) {
      if (s[s.size() - i - 1] == '1')
            a += pow(2,i);
}
return a;
}

class Node
{
public:
Node(char c) {
```

```
      this->c = c;
      weight = 1;
}

int weight;
char c;
Node* left = nullptr;
Node* right = nullptr;
Node* parent = nullptr;
};

void swap(Node* a, Node* b) {
if (a->parent != nullptr && b->parent != nullptr) {
      Node* buff = a->parent;
      a->parent = b->parent;
      b->parent = buff;
      if (a->parent == b->parent) {
            if (a->parent->left == a) {
                  a->parent->left = b;
                  a->parent->right = a;
            }
            if (a->parent->right == a) {
                  a->parent->right = b;
                  a->parent->left = a;
            }
      }
      else {
            if (a->parent->left == b) {
                  a->parent->left = a;
            }
            else if (a->parent->right == b) {
                  a->parent->right = a;
            }
            if (b->parent->left == a)
                  b->parent->left = b;
            else if (b->parent->right == a)
                  b->parent->right = b;
      }
}
}

class BinTree
{
public:
BinTree() {
      Node* nd = new Node('\0');
      nd->weight = 0;
      zero_node = nd;
      root = nd;
}

void addNode(const char c) {
      Node* nd = new Node(c);
      Node* new_zero = new Node('\0');
      new_zero->weight = 0;
```

```
            zero_node->right = nd;
            zero_node->left = new_zero;
            zero_node->weight = 1;
            nd->parent = zero_node;
            new_zero->parent = zero_node;
            zero_node = new_zero;
            counter_procedure += 2;
      }

      void find_zero(Node* nd, string s) {
            counter_procedure += 1;
            if (nd->weight == 0) {
                  current_coding_message += s;
                  return;
            }
            else if(nd->left != nullptr && nd->right != nullptr){
                  find_zero(nd->left, (s + "0"));
                  find_zero(nd->right, (s + "1"));
            }
      }

      bool check_sym(const char c, Node* nd, string s) {
            counter_procedure += 1;
            if (nd->left == nullptr && nd->right == nullptr) {
                  if (nd->c == c) {
                        nd->weight += 1;
                        current_coding_message += s;
                        return true;
                  }
                  return false;
            }
            else
                  return check_sym(c, nd->left, s + "0") || check_sym(c,
nd->right, s + "1");
      }

      int refresh_weight(Node* nd) {
            if (nd == nullptr)
                  return 0;
            counter_procedure += 1;
            if (nd->left == nullptr && nd->right == nullptr)
                  return nd->weight;
            else {
                  nd->weight      =      refresh_weight(nd->left)      +
refresh_weight(nd->right);
                  return nd->weight;
            }
      }

      void check_weight(queue<Node*> que, int size, Node** bad_node,
int current_weight) {
            int new_size = 0;
            while (size > 0) {
                  Node* nd = que.front();
                  que.pop();
```

```
            size -= 1;
            counter_procedure += 1;
            if (nd->right != nullptr) {
                  if (nd->right->weight > current_weight)
                        *bad_node = nd->right;
                  else
                        current_weight = nd->right->weight;
                  que.push(nd->right);
                  new_size += 1;
                  counter_procedure += 1;
            }
            if (nd->left != nullptr) {
                  if (nd->left->weight > current_weight)
                        *bad_node = nd->left;
                  else
                        current_weight = nd->left->weight;
                  que.push(nd->left);
                  new_size += 1;
                  counter_procedure += 1;
            }
      }
      if (!que.empty())
            check_weight(que, new_size, bad_node, current_weight);
}

void swap_nodes(queue<Node*>& que, int size, Node* bad_node) {
      int new_size = 0;
      while (size > 0) {
            Node* nd = que.front();
            que.pop();
            size -= 1;
            counter_procedure += 1;
            if (nd->right != nullptr) {
                  if (nd->right->weight < bad_node->weight) {
                        swap(nd->right, bad_node);
                        return;
                  }
                  que.push(nd->right);
                  new_size += 1;
                  counter_procedure += 1;
            }
            if (nd->left != nullptr) {
                  if (nd->left->weight < bad_node->weight) {
                        swap(nd->left, bad_node);
                        return;
                  }
                  que.push(nd->left);
                  new_size += 1;
                  counter_procedure += 1;
            }
      }
      if (!que.empty())
            swap_nodes(que, new_size, bad_node);
}
```

```
void refresh_tree() {
	refresh_weight(root);
	int flag;
	do {
		flag = 0;
		Node* bad_node = nullptr;
		queue<Node*> que;
		que.push(root);
		check_weight(que, 1, &bad_node, root->weight);
		if (bad_node != nullptr) {
			queue<Node*> que;
			que.push(root);
			swap_nodes(que, 1, bad_node);
			refresh_weight(root);
			flag = 1;
		}
	} while (flag);
}


string coding() {
	current_coding_message = "";
	char b;
	cin.get(b);
	while (b != '\n') {
		string s = "";
		if (!check_sym(b, root, s)){
			find_zero(root, s);
			current_coding_message += binary_code(b);
			addNode(b);
		}
		refresh_tree();
		cin.get(b);
		print();
	}

	return current_coding_message;
}

string coding(ifstream& file) {
	current_coding_message = "";
	char b;
	while (file.get(b)) {
		string s = "";
		if (!check_sym(b, root, s)) {
			find_zero(root, s);
			current_coding_message += binary_code(b);
			addNode(b);
		}
		refresh_tree();
		print();
	}

	return current_coding_message;
}
```

```
char decode_symbol() {
	string s = "";
	char b;
	for (int i = 0; i < 7; i++) {
		cin.get(b);
		s += b;
	}
	return from_binary(s);
}

bool follow_code(Node* nd) {
	counter_procedure += 1;
	if (nd->weight == 0) {
		char b = (char) decode_symbol();
		current_decoding_message += b;
		addNode(b);
		return true;
	}
	if (nd->left == nullptr) {
		nd->weight += 1;
		current_decoding_message += nd->c;
		return true;
	}
	char b;
	cin.get(b);
	if (b == '0') {
		return follow_code(nd->left);;
	}
	else if (b == '1') {
		return follow_code(nd->right);
	}
	 return false;
}

string decoding() {
	current_decoding_message = "";
	while (follow_code(root)) {
		refresh_tree();
		print();
	}
	refresh_weight(root);
	print();
	return current_decoding_message;
}

char decode_symbol(ifstream& file) {
	string s = "";
	char b;
	for (int i = 0; i < 7; i++) {
		file.get(b);
		s += b;
	}
	return from_binary(s);
```

```
}


bool follow_code(Node* nd, ifstream& file) {

	if (nd->weight == 0) {
		char b = (char)decode_symbol(file);
		current_decoding_message += b;
		addNode(b);
		return true;
	}
	if (nd->left == nullptr) {
		nd->weight += 1;
		current_decoding_message += nd->c;
		return true;
	}
	char b;
	file.get(b);
	if (b == '0') {
		return follow_code(nd->left, file);
	}
	else if (b == '1') {
		return follow_code(nd->right, file);
	}
	return false;
}

string decoding(ifstream& file) {
	current_decoding_message = "";
	while (follow_code(root, file)) {
		refresh_tree();
		print();
	}
	refresh_weight(root);
	print();
	return current_decoding_message;
}


int count_level(Node* nd, int n) {
	n += 1;
	if (nd->left == nullptr)
		return n;
	else {
		int a = count_level(nd->left, n);
		int b = count_level(nd->right, n);
		return a > b ? a : b;
	}
}


void printNode(Node* a, int k) {
	for (int i = 0; i < k; i++)
		cout << "   ";
	if (a->c == '\0' && a->weight != 0)
```

```
                cout << "\\" << a->weight << "/" << endl;
            else if (a->weight == 0)
                cout << "zero = 0" << endl;
            else
                cout << a->c << " = " << a->weight << endl;
            if (a->left != nullptr) {
                printNode(a->left, k + 1);
                printNode(a->right, k + 1);
            }
        }

        void print() {
            printNode(root, 0);
            cout <<"/////////" <<endl;
        }

        Node* root;
        Node* zero_node;
        string current_coding_message;
        string current_decoding_message;
        };



        int main(int argc, char* argv[]) {
        string message;
        if (argc == 1) {
            int flag = 0;
            int start;
            int end;
            cout << "Hello there!\n";
            while (!flag) {
                cout << "Continue?[y/n]\n";
                counter_procedure = 0;
                BinTree bt;
                char b;
                string s;
                cin >> b;
                switch (b) {
                case 'y':
                    break;
                case 'n':
                    return 0;
                }
                cout << "Coding/Decoding?[c/d]\n";
                cin >> b;
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                switch (b)
                {
                case 'c':
                    s = bt.coding();
                    cout << s << endl;
                    cout << "counter = " << counter_procedure <<
endl;
                    break;
```

```
                case 'd':
                      cout << bt.decoding() << endl;
                      cout << "counter = " << counter_procedure <<
endl;
                      break;
                }
                cout << "//////" << endl;
          }
     }
     else if (argc == 2) {
                ifstream file(argv[1]);
                char b;
                BinTree bt;
                file.get(b);
                if (b == '0') {

                      cout << bt.coding(file) << endl;

                }
                else if (b == '1') {
                      cout << bt.decoding(file) << endl;
                }
                file.close();
          }
     else if (argc == 3) {
          ifstream input(argv[1]);
          ofstream output(argv[2]);
          char b;
          BinTree bt;
          input.get(b);
          if (b == '0') {
                output << '1';
                output << bt.coding(input);
          }
          else if (b == '1') {
                output << '0';
                output << bt.decoding(input);
          }
          input.close();
          output.close();
     }
     else
          return 0;



     return 0;
     }
```

ПРИЛОЖЕНИЕ Б.

КОД ПРОГРАММЫ ДЛЯ ГЕНЕРАЦИИ НАБОРА ВХОДНЫХ ДАННЫХ (СРЕДНИЙ СЛУЧАЙ)

```
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>
#include <vector>

using namespace std;

int main(int argc, char* argv[]) {
    srand(static_cast<unsigned int>(time(0)));
    int n;
    cin >> n;
    char buff;
    vector<char> a;
    for (int i = 0; i < n; i++) {
        buff = (char)rand() % 122;
        while (buff < 62 || buff > 122) {
            buff = (char)rand() % 122;
            }
            a.push_back(buff);
        }
    for (int i = 0; i < a.size(); i++)
        cout << a[i];
    cout << endl;
    return 0;

}
```

ПРИЛОЖЕНИЕ В.

КОД ПРОГРАММЫ ДЛЯ ГЕНЕРАЦИИ НАБОРА ВХОДНЫХ ДАННЫХ (ХУДШИЙ СЛУЧАЙ)

```
#include <ctime>
#include <cstdlib>
#include <fstream>
#include <vector>
#include <iostream>

using namespace std;


int main(int argc, char* argv[]) {
    int n;
    srand(static_cast<unsigned int>(time(0)));

    cin >> n;
    char buff;
    vector<char> a;
    int m;
    while (n > 0) {
        m = n % 95;
        if (m == 0) m = 1;
        for (int i = m; i >= 0; i--) {
            a.push_back((char)(i + 32));
        }
        n -= m;
    }

    for (int i = 0; i < a.size(); i++)
        cout << a[i];
    cout << endl;
    return 0;

}
```