МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждения высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практической работы № 8.1

Тема:

«Кодирование и сжатие данных методами без потерь»

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Туляшева А.Т.

Группа: ИКБО-42-23

Москва - 2024

Содержание

ЦЕЛЬ РАБОТЫ

Цель: Получение практических навыков и знаний по выполнению сжатия данных рассматриваемыми методами.

1 ЗАДАНИЕ 1

Исследование алгоритмов сжатия на примерах.

1) Выполнить каждую задачу варианта, представив алгоритм решения в виде таблицы и указав результат сжатия.

2) Описать процесс восстановления сжатого текста.

3) Сформировать отчет, включив задание, вариант задания, результаты выполнения задания варианта.

Вариант 1:

Вариант	Сжатие данных по методу Лемпеля–Зива LZ77 Используя двухсимвольный алфавит (0, 1) закодировать следующую фразу:	Закодировать следующую фразу, используя код LZ78	Закодировать фразу методами Шеннона–Фано
	1	2	3
1	0001010010101001101	кукуркукурекурекун	Ана, дэус, рики, паки, Дормы кормы констунтаки, Дэус дэус канадэус – бац!

1.1 Алгоритм 1

1.1.1 Постановка задачи

Применение алгоритма группового сжатия текста RLE.

Сжать текст, используя метод RLE (run length ecoding/кодирование длин серий/групповое кодирование).

Требования к выполнению заданию

1) Описать процесс сжатия алгоритмом RLE.

2) Придумать текст, в котором есть длинные (в разумных пределах) серии из повторяющихся символов. Выполнить сжатие текста. Рассчитать коэффициент сжатия.

3) Придумать текст, в котором много неповторяющихся символов и между ними могут быть серии. Выполнить групповое сжатие, показать коэффициент сжатия. Применить алгоритм разделения текста при групповом кодировании, позволяющий повысить эффективность сжатия этого текста. Рассчитать коэффициент сжатия после применения алгоритма.

1.1.2 Решение задания

Метод RLE (run-length encoding) основан на замене последовательных повторяющихся символов единичным символом и его счетчиком. Данный метод эффективен для текстов с повторяющимися символами, таких как строки, содержащие много пробелов, нулей или других повторяющихся элементов.

Шаги работы алгоритма RLE:

1. Инициализация: Создаем пустую строку для хранения сжатого результата и счётчик для отслеживания количества повторений.

2. Итерация: Пробираем входной текст посимвольно. Если текущий символ совпадает с предыдущим, увеличиваем счётчик. Иначе добавляем количество повторений и предыдущий символ в сжатую строку и сбрасываем счётчик для нового символа.

3. Добавление завершающей пары: После завершения итерации добавляем последнюю пару "символ-счётчик" в сжатую строку.

4. Возвращение результата: Возвращаем полученную сжатую строку.

Пример 1: Текст с длинными сериями повторяющихся символов:

AAAAAAAAAAABBBBBBBBBCCCCC

Текст можно представить в виде:

11A9B5C

Расчет коэффициента сжатия:

$$K = \frac{\text{Исходная длина}}{\text{Сжатая длина}}$$

Для данного текста:

- Исходная длина: 26 символа
- Сжатая длина: 10 символов

Тогда K = 26 / 10 ≈ 2.6

Пример 2: Текст с большим количеством уникальных символов:

abcdabcdabcdabbbccdddddccccbba

Сжатие текста (используя алгоритм RLE):

abcdabcdabcda3b2c2d5c5b2a

Коэффициент сжатия:

- Исходная длина: 26 символов
- Сжатая длина: 18 символов
- K = 26 / 18 ≈ 1.44(4)

Улучшение сжатия с разделением текста. Разделим текст на сегменты для RLE и обработаем повторяющиеся блоки:

(abcd)3a3b2c2d5c5b2a

Этот подход позволяет записать повторяющийся блок (abcd)3 один раз, указав его количество.

- Новая сжатая длина: 13 символов
- Новый коэффициент сжатия: K = 26 / 13 = 2.

Таким образом, использование RLE с разделением текста позволило улучшить коэффициент сжатия.

1.2 Алгоритм 2

1.2.1 Постановка задачи

Исследование алгоритмов группового сжатия (методы Лемпеля –Зива: LZ77, LZ78) на примерах.

Тексты для сжатия:

Сжатие данных по методу Лемпеля–Зива LZ77 Используя двухсимвольный алфавит (0, 1) закодировать следующую фразу:	Закодировать следующую фразу, используя код LZ78
1	2
0001010010101001101	кукуркукурекурекун

Требования к выполнению заданию

1) Выполнить каждую задачу варианта задания, представив алгоритм решения в виде таблицы и указав результат сжатия.

2) Описать процесс восстановления сжатого текста.

3) Сформировать отчет, включив задание, вариант задания, результаты выполнения задания варианта (таблица).

1.2.2 Решение задачи

Методы Лемпеля-Зива (LZ) — это алгоритмы сжатия данных, которые используют подход, основанный на нахождении повторяющихся подстрок. Основное отличие между LZ77 и LZ78 заключается в способе хранения информации о найденных повторениях.

- LZ77: Использует скользящее окно, чтобы находить повторяющиеся подстроки и записывать их как ссылку на начало строки и длину совпадения. Формат записи: (дистанция, длина, символ).

Шаги сжатия LZ77:

1. Начинаем с пустого окна и читаем символы.

2. Запоминаем максимальную длину совпадения среди уже закодированных подстрок.

3. Кодируем текущий символ на основе найденных совпадений.

- LZ78: Сжимает данные, создавая словарь, в который добавляются уникальные подстроки. Каждое новое значение хранит индекс предыдущей подстроки и следующий символ.

Шаги сжатия LZ78:

1. Создаем пустой словарь.

2. Считываем символы и создаем подстроки, обновляя словарь. Сжатие двоичного кода методом Лемпеля –Зива LZ77:

Исходный текст	0001010010101001101
LZ-код	0.00.01.010.010.100.1101
R	1 2 3 4
Вводимые коды	- 10 11 100 101 110 111

Где LZ – сжатый текст (в данном примере в связи с небольшим размером исходного текста размер текста не уменьшился). R отмечает шаги кодирования, после которых происходит переход на представление кодов A увеличенным числом разрядов R. Так, на первом шаге вводится код 10 для комбинации 00, и поэтому на следующих двух шагах R = 2, после третьего шага R = 3, после седьмого шага R = 4, т.е. в общем случае R = K после шага 2^{K-1} –1.

Сжатие текста методом Лемпеля –Зива LZ78. Слово: кукуркукурекурекун.

Содержимое словаря		Содержимое считанной строки	Код
	1	к	<0,к>
к	2	у	<0,у>
к, у	3	ку	<1,у>
к, у, ку	4	р	<0,р>
к, у, ку, р	5	кук	<3,к>
к, у, ку, р, кук	6	ур	<2,р>
к, у, ку, р, кук, ур	7	е	<0,е>
к, у, ку, р, кук, ур, е	8	кур	<3,р>
к, у, ку, р, кук, ур, е, кур	9	ек	<7,к>
к, у, ку, р, кук, ур, е, кур, ек		ун	<2,н>

Результат сжатия: 0к0у1у0р3к2р0е3р7к2н

Алгоритм восстановления сжатого текста:

Процесс восстановления:

1. Инициализация:

   - Создаем пустой словарь.

   - Выходная строка тоже изначально пуста.

2. Чтение сжатой строки по блокам (index, next_char):

   - 0к:

     - index = 0, добавляем символ к.

     - Выходная строка: к.

     - Добавляем к в словарь: {1: "к"}.

   - 0у:

     - index = 0, добавляем символ у.

     - Выходная строка: ку.

     - Добавляем у в словарь: {1: "к", 2: "у"}.

   - 1у:

     - index = 1, берем из словаря значение "у", добавляем к.

     - Выходная строка: куку.

     - Добавляем ку в словарь: {1: "к", 2: "у", 3: "ку"}.

   - 0р:

     - index = 0, добавляем символ р.

     - Выходная строка: кукур.

     - Добавляем р в словарь: {1: "к", 2: "у", 3: "ку", 4: "р"}.

   - 3к:

- index = 3, берем из словаря значение "ку", добавляем к.

- Выходная строка: кукуркук.

- Добавляем кук в словарь: {1: "к", 2: "у", 3: "ку", 4: "р", 5: "кук"}.

- 2р:

- index = 2, добавляем ур.

- Выходная строка: кукуркукур.

- Добавляем р в словарь: {1: "к", 2: "у", 3: "ку", 4: "р", 5: "кук", 6: "ур"}.

- 0е:

- index = 0, добавляем е.

- Выходная строка: кукуркукуре.

- Добавляем е в словарь: {1: "к", 2: "у", 3: "ку", 4: "р", 5: "кук", 6: "ур", 7: "е"}.

- 3р:

- index = 3, берем из словаря значение "ку", добавляем р.

- Выходная строка: кукуркукурекур.

- Добавляем кур в словарь: {1: "к", 2: "у", 3: "ку", 4: "р", 5: "кук", 6: "ур", 7: "е", 8: "кур"}.

- 7к:

- index = 7, берем из словаря значение "е", добавляем к.

- Выходная строка: кукуркукурекурек.

- Добавляем ек в словарь: {1: "к", 2: "у", 3: "ку", 4: "р", 5: "кук", 6: "ур", 7: "е", 8: "кур", 9: "ек"}.

- 2н:

- index = 2, берем из словаря значение "у", добавляем н.

- Выходная строка: кукуркукурекурекун.

Итак, восстановленная строка выглядит как кукуркукурекурекун, что соответствует заданной строке.

1.3 Алгоритм 3, 4

1.3.1 Постановка задачи

Разработать на псевдокоде или описать словесно алгоритмы сжатия и восстановления текста методами Шеннона-Фано и Хаффмана.

Представить процесс выполнения алгоритма для задачи варианта. Все результаты представить в отчете.

Требования к выполнению заданию

1. Сформировать отчет по разработке каждого алгоритма в соответствии с требованиями.

1.1. По методу Шеннона-Фано.

1) Данными для выполнения задания является текст.

2) Привести постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования.

3) Представить таблицу формирования кода для текста варианта задания.

4) Изобразить префиксное дерево формирования кода.

5) Рассчитать коэффициент сжатия.

1.2. По методу Хаффмана.

1) Данные для выполнения задания: ваша фамилия имя отчество.

2) Привести постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования.

3) Построить таблицу частот встречаемости символов в исходной строке для чего сформировать алфавит исходной строки и посчитать количество вхождений (частот) символов и их вероятности появления.

4) Изобразить префиксное дерево Хаффмана.

5) Упорядочить построенное дерево слева-направо (при необходимости) и изобразить его.

6) Провести кодирование исходной строки по аналогии с примером.

7) Рассчитать коэффициенты сжатия относительно кодировки ASCII и относительно равномерного кода.

8) Рассчитать среднюю длину полученного кода и его дисперсию.

9) По результатам выполненной работы сделать выводы и сформировать отчет. Отобразить результаты выполнения всех требований (с 1 по 8), предъявленных в задании и оформить разработку программы: постановка, подход к решению, код, результаты тестирования.

1.3.2 Решение задачи

Алгоритм Шеннона-Фано основан на частоте повторения символов. Так, часто встречающийся символ кодируется кодом меньшей длины, а редко встречающийся — кодом большей длины.

В свою очередь, коды, полученные при кодировании, префиксные. Это и позволяет однозначно декодировать любую последовательность кодовых слов. Но все это вступление. Для работы оба алгоритма должны иметь таблицу частот элементов алфавита.

Принцип работы метода Шеннона-Фано:

1. Подсчет частоты символов: Сначала вычисляем количество вхождений каждого символа в исходном тексте. Это поможет определить, какие символы встречаются чаще, а какие реже.
2. Сортировка по частоте: Символы сортируются по убыванию их частоты (или вероятности появления).
3. Разделение на группы: Сортированные символы разделяются на две группы, так чтобы сумма частот в обеих группах была как можно более равной. Это делается на каждом шаге рекурсивно.
4. Присваивание кодов: Каждой группе символов присваиваются биты: 0 для одной группы и 1 для другой. Таким образом, каждый символ

получает уникальный код. Код каждого символа будет комбинацией 0 и 1, и для символов с большей частотой код будет короче.

5. Кодировка: После того как символы получили свои коды, их частота умножается на количество бит в коде, чтобы посчитать объем закодированной фразы.

По варианту: «Ана, дэус, рики, паки, Дормы кормы констунтаки, Дэус дэус канадэус – бац»

Символ	Частота	1-я цифра	2-я цифра	3-я цифра	4-я цифра	5-я цифра	6-я цифра	7-я цифра	Код	Кол-во бит
пробел	10	0	0	0					000	30
а	6	0	0	1					001	18
к	6	0	1	0					010	18
,	5	0	1	1	0				0110	20
у	5	0	1	1	1				0111	20
с	5	1	0	0	0				1000	20
д	5	1	0	0	1				1001	20
о	3	1	0	1	0				1010	12
т	3	1	0	1	1	0			10110	15
э	3	1	0	1	1	1			10111	15
А	2	1	1	0	0	0			11000	10
р	2	1	1	0	0	1			11001	10
и	2	1	1	0	1	0			11010	10
п	2	1	1	0	1	1			11011	10
Д	2	1	1	1	0	1			11100	10
м	2	1	1	1	0	1			11101	10
н	2	1	1	1	1	0			11110	10

-	1	1	1	1	1	1	0		111110	6
б	1	1	1	1	1	1	1	0	1111110	7
ц	1	1	1	1	1	1	1	1	1111111	7

Объем незакодированной фразы – 73 * 8 бит = 584 бит.

Объем закодированной фразы – 278 бит.

Коэффициент сжатия: 278 / 584 = 0.48

Дерево:

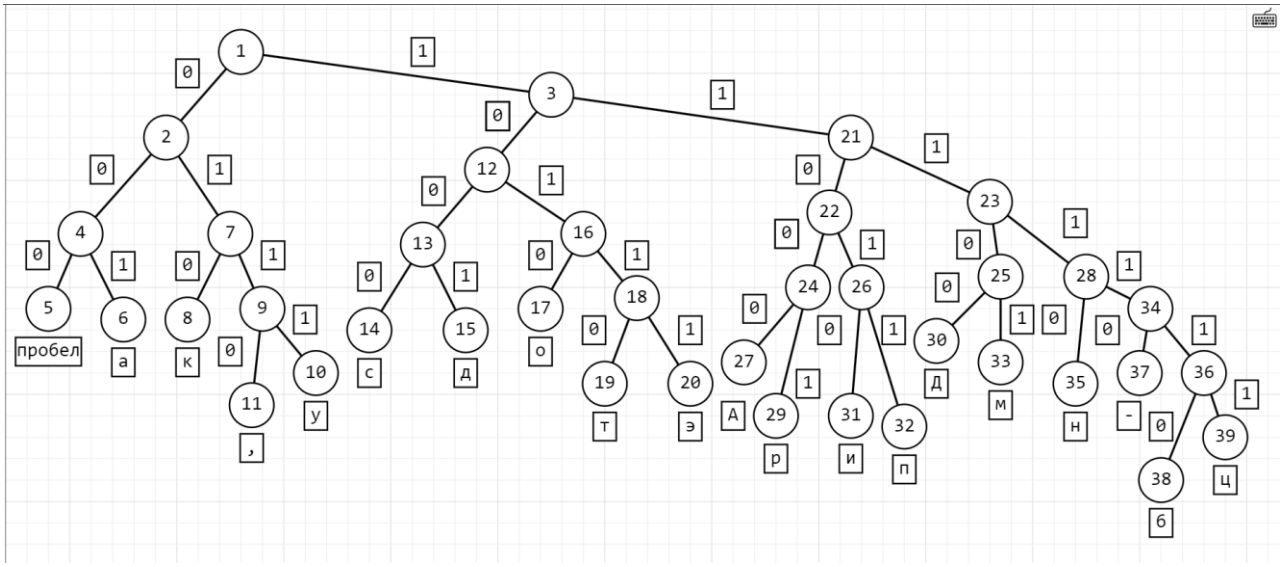

Рисунок 1 - Дерево префиксного кода Фано

Код Хаффмана - это особый тип оптимального префиксного кода, который обычно используется для сжатия данных без потерь. Он сжимает данные, очень эффективно экономя от 20% до 90% памяти, в зависимости от характеристик сжатых данных.

Кодовый символ – это наименьшая единица данных, подлежащая сжатию.

Кодовое слово – это последовательность кодовых символов из алфавита кода.

Префиксный код – это код, в котором никакое кодовое слово не является префиксом любого другого кодового слова.

Оптимальный префиксный код – это префиксный код, имеющий минимальную среднюю длину.

Кодовое дерево (дерево кодирования Хаффмана) – это бинарное дерево, у которого: листья помечены символами, для которых разрабатывается кодировка; узлы (в том числе корень) помечены суммой вероятностей появления всех символов, соответствующих листьям поддерева, корнем которого является соответствующий узел.

Мой пример – туляшева арина тимуровна (24). Вероятность появления символа: количество вхождений символа в текст/количество символов в тексте.

Таблица в обычном порядке:

Алфавит	т	у	л	я	ш	е	в	а	«»
Кол. вх.	2	2	1	1	1	1	2	4	2
Вероятн.	0.083	0.083	0.042	0.042	0.042	0.042	0.083	0.167	0.083

Алфавит	р	и	н	т	м	о
Кол. вх.	2	2	2	2	1	1
Вероятн.	0.083	0.083	0.083	0.083	0.042	0.042

Таблица данных, отсортированная по убыванию частот:

Алфавит	а	т	у	в	«»	р	и	н	т
Кол. вх.	4	2	2	2	2	2	2	2	2
Вероятн.	0.167	0.083	0.083	0.083	0.083	0.083	0.083	0.083	0.083

Алфавит	л	я	ш	е	м	о
Кол. вх.	1	1	1	1	1	1
Вероятн.	0.042	0.042	0.042	0.042	0.042	0.042

Дерево Хаффмана:

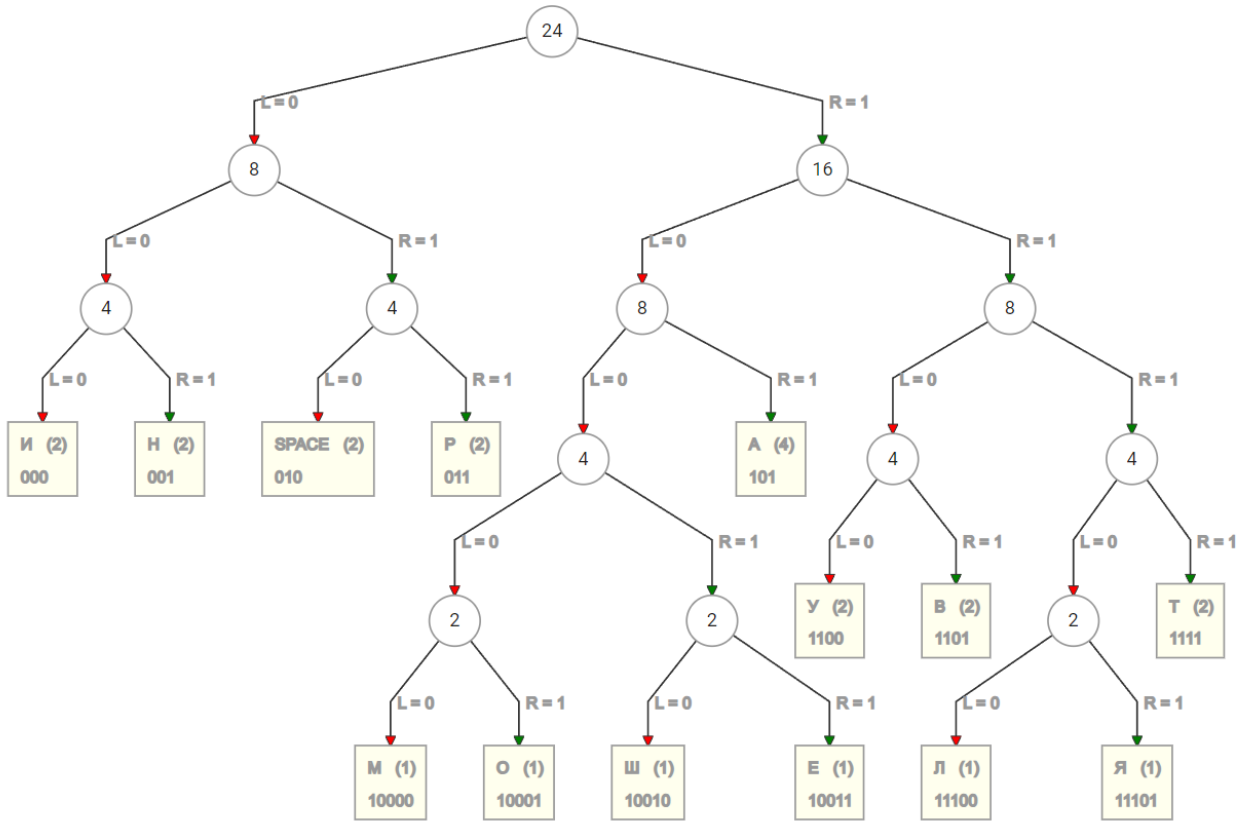

Рисунок 2 - Дерево Хаффмана

Кодирование исходной строки по аналогии с примером:

111111001110011101100101001111011101 101011000001101 111100010000110001110001110100110 1

Общее число битов в заданном коде = 84 бита

Коэффициент сжатия = 24*8/84 = 2.286

2 ЗАДАНИЕ 2

2.1 Алгоритм Шеннона-Фано

2.1.1 Постановка задачи

Разработать алгоритм и реализовать программу сжатия текста алгоритмом Шеннона – Фано. Разработать алгоритм и программу восстановления сжатого текста. Выполнить тестирование программы на текстовом файле. Определить процент сжатия.

2.1.2 Решение задачи

Для решения задания была разработана программа (рис. 3 – 5). На рисунке 3 показаны структура Symbol, которая хранит символ (в виде строки) и его

частоту, функция compareFrequency для сортировки символов по их частоте в порядке убывания и функция buildShannonFanoCode, которая рекурсивно строит коды для каждого символа, деля список символов на две части с приблизительно одинаковой общей частотой.

```
#include ...
using namespace std;

struct Symbol { //структура хранения символа и частоты
    string character;
    int frequency;
};

bool compareFrequency(const Symbol& a, const Symbol& b) {
    return a.frequency > b.frequency; //сортировка символов по частоте (по убыванию)
}

//для построения кода Шеннона-Фано
void buildShannonFanoCode(vector<Symbol>& symbols, map<string, string>& codes, string code = "") {
    if (symbols.size() == 1) {
        codes[symbols[0].character] = code; //присвоение кода символу
        return;
    }
    //деление символов на две группы с ~ одинаковой общей частотой
    int totalFrequency = 0;
    for (const auto& s : symbols) {
        totalFrequency += s.frequency; //сумма частот всех символов
    }
    int halfFrequency = 0;
    size_t splitIndex = 0;
    for (size_t i = 0; i < symbols.size(); ++i) {
        halfFrequency += symbols[i].frequency; //точка раздела по частоте
        if (halfFrequency >= totalFrequency / 2) {
            splitIndex = i + 1; //индекс деления массива
            break;
        }
    }
    //два подмассива для рекурсии
    vector<Symbol> left(symbols.begin(), symbols.begin() + splitIndex);
    vector<Symbol> right(symbols.begin() + splitIndex, symbols.end());
    buildShannonFanoCode(left, codes, code + "0"); //для левых 0 к коду
    buildShannonFanoCode(right, codes, code + "1"); //для правых 1
}
```

Рисунок 3 – Программа

На рисунке 4 показаны функция compressText, которая сжимает текст, заменяя каждый символ его кодом из таблицы кодов и функция decompressText, которая восстанавливает исходный текст из сжатого, используя обратные коды символов. А на рисунке 5 изображен main.

```cpp
//для сжатия текста
string compressText(const string& text, const map<string, string>& codes) {
    string compressedText;
    for (size_t i = 0; i < text.size(); ++i) {
        string character(1, text[i]);  //строка из одного символа
        compressedText += codes.at(character); //добавление кода символа в сжатый текст
    } //замена каждого символа его кодом из таблицы кодов
    return compressedText; //сжатый текст
}

//для восстановления текста
string decompressText(const string& compressedText, const map<string, string>& reverseCodes) {
    string decompressedText;
    string currentCode;
    for (char c : compressedText) {
        currentCode += c; //текущий код из симовлов сжатого текста
        if (reverseCodes.find(currentCode) != reverseCodes.end()) { //код найден в обратной таблице
            decompressedText += reverseCodes.at(currentCode); //добавление символа
            currentCode = "";
        }
    }
    return decompressedText; //востановленный текст
}
```

Рисунок 4 – Программа

```cpp
int main() {
    setlocale(LC_ALL, "ru");
    ifstream inputFile("input.txt");
    string text((istreambuf_iterator<char>(inputFile)), istreambuf_iterator<char>());
    inputFile.close();
    //подсчет частоты появления каждого символа
    map<string, int> frequencyMap;
    for (size_t i = 0; i < text.size(); ++i) {
        string character(1, text[i]);
        frequencyMap[character]++;
    }
    //сортировка символов по частоте
    vector<Symbol> symbols;
    for (const auto& entry : frequencyMap) {
        symbols.push_back({ entry.first, entry.second });
    } //символы с частотой
    sort(symbols.begin(), symbols.end(), compareFrequency);

    map<string, string> codes;
    buildShannonFanoCode(symbols, codes);

    map<string, string> reverseCodes;
    for (const auto& entry : codes) {
        reverseCodes[entry.second] = entry.first;
    } //создание обратной таблицы

    string compressedText = compressText(text, codes);
    cout << "Сжатый текст: " << endl;
    for (char c : compressedText) {
        cout << c;
    } //сжатый текст в бинарном виде
    cout << endl;

    string decompressedText = decompressText(compressedText, reverseCodes);
    cout << "Восстановленный текст: " << endl;
    cout << decompressedText << endl;

    double compressionRatio = (double)compressedText.size() / (text.size() * 8) * 100;
    cout << "Процент сжатия: " << compressionRatio << "%" << endl;
}
```

Рисунок 5 – main

2.1.3 Тестирование

Было проведено тестирование на тексте, заданного моим вариантом (1) для задания 1: «Ана, дэус, рики, паки, Дормы кормы констунтаки, Дэус дэус канадэус – бац». Коэффициенты сжатия совпадают, значит задание выполнено успешно (рис. 5).

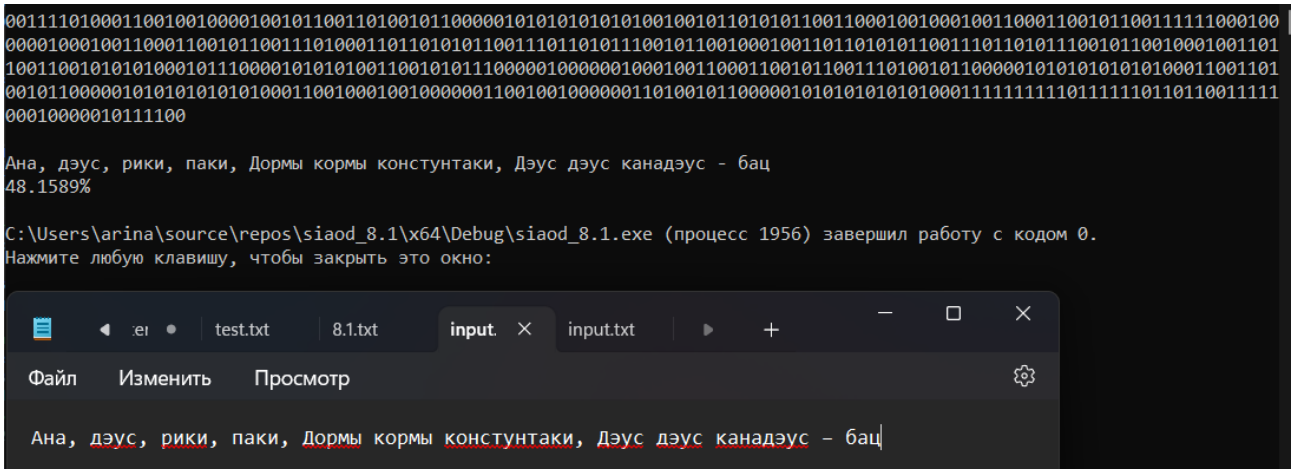

Рисунок 5 – Тестирование (на русском)

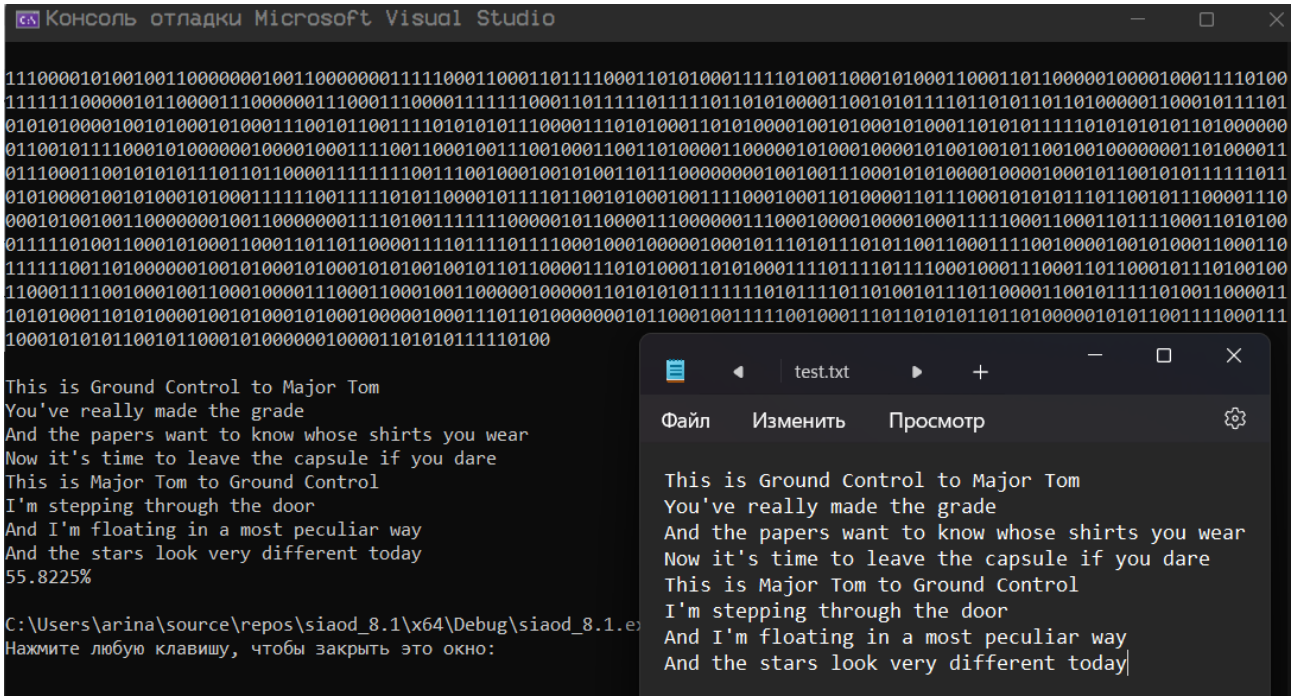

Рисунок 6 – Тестирование (на английском)

2.2 Алгоритм Хаффмана

2.2.1 Постановка задачи

Применить алгоритм Хаффмана для архивации данных текстового файла. Выполнить практическую оценку сложности алгоритма Хаффмана. Провести

архивацию этого же файла любым архиватором. Сравнить коэффициенты сжатия разработанного алгоритма и архиватора.

2.2.2. Решение задачи

Для решения задания была разработана программа (рис. 7 – 9). На рисунке 7 изображены структура Node, где каждый узел представляет символ с частотой его появления и указателями на левый и правый дочерние узлы, структура Compare, которая задает, как узлы сравниваются в очереди, функция generateCodes, что рекурсивно строит код Хаффмана для каждого символа, добавляя 0 и 1 для левого и правого поддеревьев соответственно, функция compressText для сжатия и функция decompressText для восстановления текста.

```
struct Node { //структура для представления узла дерева Хаффмана
    char ch; //символ, хранящийся в узле
    int freq; //частота
    Node* left; //указатель на левый дочерний узел
    Node* right; //на правый
    Node(char ch, int freq) : ch(ch), freq(freq), left(nullptr), right(nullptr) {}
}; //создание узла с символом и частотой

struct Compare { //сравнение узлов по частоте
    bool operator()(Node* left, Node* right) {
        return left->freq > right->freq;  //сравнение частот
    }
};

 //для построения кодов Хаффмана
void generateCodes(Node* root, const string& str, map<char, string>& codes) {
    if (!root) return; //узел пустой
    if (!root->left && !root->right) { //листовой узел (символ)
        codes[root->ch] = str; //добавление кода для этого символа
    }
    generateCodes(root->left, str + "0", codes); //0 к коду для левого поддерева
    generateCodes(root->right, str + "1", codes); //1 для правого
}

 //сжатие
string compressText(const string& text, map<char, string>& codes) {
    string compressedText = ""; //строка для сжатого текста
    for (char ch : text) { //по каждому символу в тексту
        compressedText += codes[ch];  //добавление кода символа в сжатый текст
    }
    return compressedText; //сжатый текст
}

 //восстановление
string decompressText(const string& compressedText, map<string, char>& reverseCodes) {
    string decompressedText = ""; //строка для восст. текста
    string currentCode = ""; //хранение тек. кода
    for (char bit : compressedText) { //по каждому биту в сжатом тексте
        currentCode += bit; //+бит к тек. коду
        if (reverseCodes.find(currentCode) != reverseCodes.end()) { //есть код в обратной таблице
            decompressedText += reverseCodes[currentCode];  //добавление символа в восст. текст
            currentCode = ""; //очищение кода для след. символа
        }
    }
    return decompressedText; //восстановленный текст
}
```

Рисунок 7 – Программа

На рисунке 8 представлена функция buildHuffmanTree, которая строит дерево Хаффмана, подсчитывая частоты символов и объединяя их в минимальном порядке. А на рисунке 9 – main.

```
//для построения дерева Хаффмана и получения кодов
void buildHuffmanTree(const string& text, map<char, string>& codes, map<string, char>& reverseCodes) {
    map<char, int> frequencyMap; //подсчет частоты каждого символа
    for (char ch : text) {
        frequencyMap[ch]++;
    }
    //очередь для узлов дерева (по частоте)
    priority_queue<Node*, vector<Node*>, Compare> minHeap;
    for (auto pair : frequencyMap) { //по частотам
        Node* node = new Node(pair.first, pair.second); //новый узел
        minHeap.push(node); //+ узел в очередь
    }
    while (minHeap.size() > 1) { //строим дерева хаффмана
        Node* left = minHeap.top(); //извлекаем узел с min частотой
        minHeap.pop();
        Node* right = minHeap.top(); //извдекаем след. узел
        minHeap.pop();

        Node* merged = new Node('\0', left->freq + right->freq);  //новый узел: два предыдущих
        merged->left = left; //левый потомок - 1 узел
        merged->right = right; //правый потомок - 2 узел

        minHeap.push(merged); //+ объед. узел в очередь
    }
    //генерация кодов хаффмана из корня дерева
    generateCodes(minHeap.top(), "", codes);
    for (auto pair : codes) { //обратная таблица
        reverseCodes[pair.second] = pair.first;
    } //ключ - код, значение - символ
}
```

Рисунок 8 - Программа

```
int main() {
    setlocale(LC_ALL, "ru_RU.UTF-8");
    ifstream inputFile("input.txt");
    string text((istreambuf_iterator<char>(inputFile)), istreambuf_iterator<char>());
    inputFile.close();

    map<char, string> huffmanCodes; //таблица кодов хаффмана
    map<string, char> reverseCodes; //обратная таблица
    buildHuffmanTree(text, huffmanCodes, reverseCodes); //дерева хаффмана

    string compressedText = compressText(text, huffmanCodes);
    cout << "Сжатый текст: " << endl;
    cout << compressedText << endl;
    string decompressedText = decompressText(compressedText, reverseCodes);
    cout << "Восстановленный текст: " << endl;
    cout << decompressedText << endl;

    double compressionRatio = (double)compressedText.size() / (text.size() * 8) * 100;
    cout << "Процент сжатия: " << compressionRatio << "%" << endl;
}
```

Рисунок 9 – main

2.2.3 Тестирование

Было проведено тестирование на большом тексте (рис. 10). Размер исходного текста - 9024 бита, размер сжатого текста - 5220 бит, размер архива – 5264 бита.

```
Ground Control to Major Tom
Ground Control to Major Tom
Take your protein pills and put your helmet on

Ground Control to Major Tom
Commencing countdown, engines on
Check ignition and may God's love be with you

Ten, Nine, Eight, Seven,
Six, Five, Four, Three, Two, One, Liftoff

This is Ground Control to Major Tom
You've really made the grade
And the papers want to know whose shirts you wear
Now it's time to leave the capsule if you dare

This is Major Tom to Ground Control
I'm stepping through the door
And I'm floating in a most peculiar way
And the stars look very different today

For here
Am I sitting in a tin can
Far above the world
Planet Earth is blue
And there's nothing I can do

Though I'm past one hundred thousand miles
I'm feeling very still
And I think my spaceship knows which way to go
Tell my wife I love her very much she knows

Ground Control to Major Tom
Your circuit's dead, there's something wrong
Can you hear me, Major Tom?
Can you hear me, Major Tom?
Can you hear me, Major Tom?
Can you....

Here am I floating round my tin can
Far above the Moon
Planet Earth is blue
And there's nothing I can do

9024
5220

57.8457%
```

Рисунок 10 – Тестирование

ВЫВОДЫ

Алгоритмы сжатия данных играют важную роль в оптимизации хранения и передачи информации. RLE (Run-Length Encoding) — один из самых простых методов, который эффективен для данных с последовательными повторениями, таких как графические изображения. Этот алгоритм заменяет последовательности одинаковых символов их значением и количеством, что позволяет значительно снизить объем данных в случаях, когда такие последовательности часто встречаются. Однако RLE неэффективен для данных с высокой энтропией, где нет длинных последовательностей повторяющихся символов.

Алгоритмы LZ77 и LZ78 представляют собой более сложные методы сжатия, использующие концепцию словарного кодирования. Эти алгоритмы работают путем поиска повторяющихся подстрок и замены их ссылками на первое появление в тексте, что позволяет достичь хороших результатов при сжатии текстовых и бинарных данных. LZ77 использует скользящее окно для поиска повторений, в то время как LZ78 строит словарь из уникальных подстрок, что делает его более гибким для разнообразных данных. Оба алгоритма широко применяются в современных форматах сжатия, таких как ZIP и PNG, обеспечивая значительное уменьшение размера файлов.

Методы Шеннона-Фано и Хаффмана относятся к алгоритмам сжатия на основе кодирования переменной длины. Оба подхода основываются на частоте встречаемости символов: чем чаще символ встречается, тем короче его код. Алгоритм Хаффмана, в частности, оптимален для минимизации средней длины кода, что делает его эффективным для текстовых данных. Эти методы обеспечивают высокую степень сжатия и часто используются в сочетании с другими алгоритмами для достижения лучших результатов. В заключение, выбор алгоритма сжатия зависит от характера данных и требований к скорости и степени сжатия.

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона.
2. Практические занятия Сорокин А.В. - РТУ МИРЭА 2024
3. Лекции Бузыкова Ю.С. – РТУ МИРЭА 2024
4. Практические занятия преподавателя Муравьевой Е. А., 2024.
5. Лекции Лозовский В.В. – РТУ МИРЭА 2024