ГУАП

КАФЕДРА № 51

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| канд. техн. наук |  |  |  | Е. М. Линский |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О КУРСОВОЙ РАБОТЕ |
| АЛГОРИТМ ХАФФМАНА |
| по курсу: ОСНОВЫ ПРОГРАММИРОВАНИЯ |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 5013 |  |  |  | А. В. Гончаров |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2021

Оглавление

[1 Постановка задачи 3](#_Toc90902786)

[2 Алгоритм Хаффмана 3](#_Toc90902787)

[Описание алгоритма 3](#_Toc90902788)

[Структуры данных, используемые в программе 3](#_Toc90902790)

[Псевдокод 3](#_Toc90902791)

[3 Инструкция пользователя 5](#_Toc90902792)

[Запуск программы 5](#_Toc90902793)

[Формат входного файла 5](#_Toc90902794)

[Формат выходного файла 5](#_Toc90902795)

[4 Тестовые примеры 5](#_Toc90902796)

[Тест 1 6](#_Toc90902797)

[Тест 2 6](#_Toc90902798)

[Тест 3 6](#_Toc90902799)

[Список литературы 6](#_Toc90902800)

# Постановка задачи

Задачей данной курсовой работы является разработка программы-архиватора и разархиватора английских букв, использующей для сжатия алгоритм Хаффмана. Разархивированный текст должен соответствовать исходному.

# Алгоритм Хаффмана

## Описание алгоритма

Алгоритм Хаффмана - жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при написании им курсовой работы. В настоящее время используется во многих программах сжатия данных.

Алгоритм кодирует наиболее часто встречаемые символы кодами меньшей длины, а редко встречаемые символы кодами большей длины. Таким образом, заменяя символы на их соответствующие коды, полученные в ходе обхода дерева Хаффмана можно сжать файл.

При декодировании восстанавливается бинарное дерево, при обходе которого идентифицируются закодированные символы, после чего текст декодируется.

Пример:

Считать исходный текст из файла, после чего сжать его, используя алгоритм Хаффмана. Затем полученный сжатый текст сохранить в отдельный файл вместе с деревом Хаффмана для последующего декодирования:

Исходный текст:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

Алгоритм начинается с получения исходного текста из файла. Затем считается частота повторения каждого уникального символа, и пара символ-частота записывается в приоритетную очередь (Таблица 1):

Таблица 1 – Приоритетная очередь

|  |  |  |
| --- | --- | --- |
| **Символ** | **A** | **B** |
| Частота | 59 | 35 |

Создаётся новый узел двоичного дерева, частота которого равна сумме частот двух символов с наивысшей частотой, которые затем становятся его листьями и удаляются из приоритетной очереди. Процесс повторяется до тех пор, пока все вершины не станут листьями дерева (Рисунок 1).

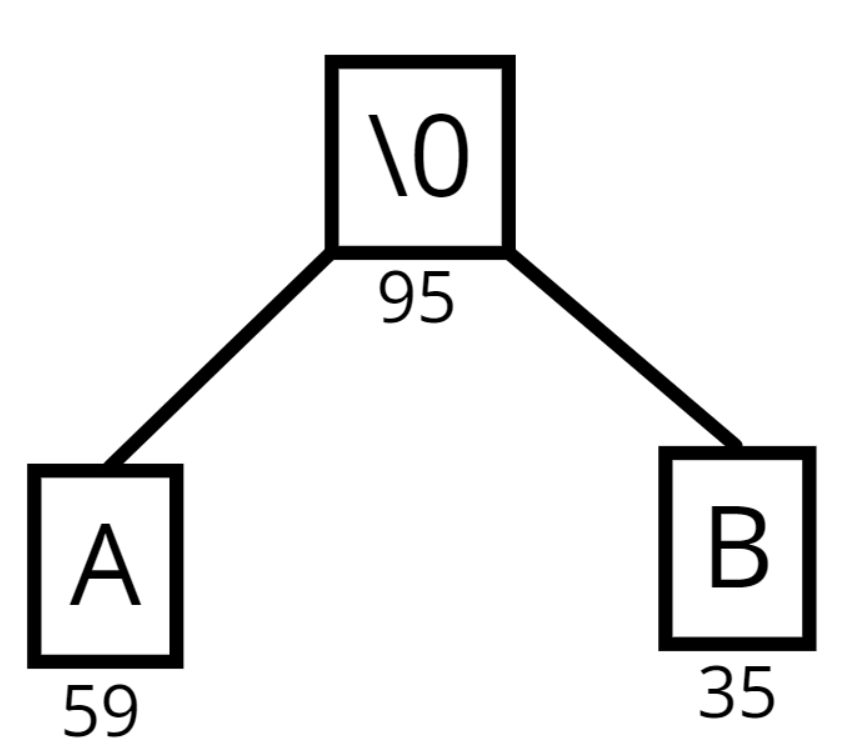


Рисунок 1 – Дерево Хаффмана

После завершения работы цикла выполняется обход бинарного дерева, во время которого для каждого символа определяется соответствующая последовательность нулей и единиц.

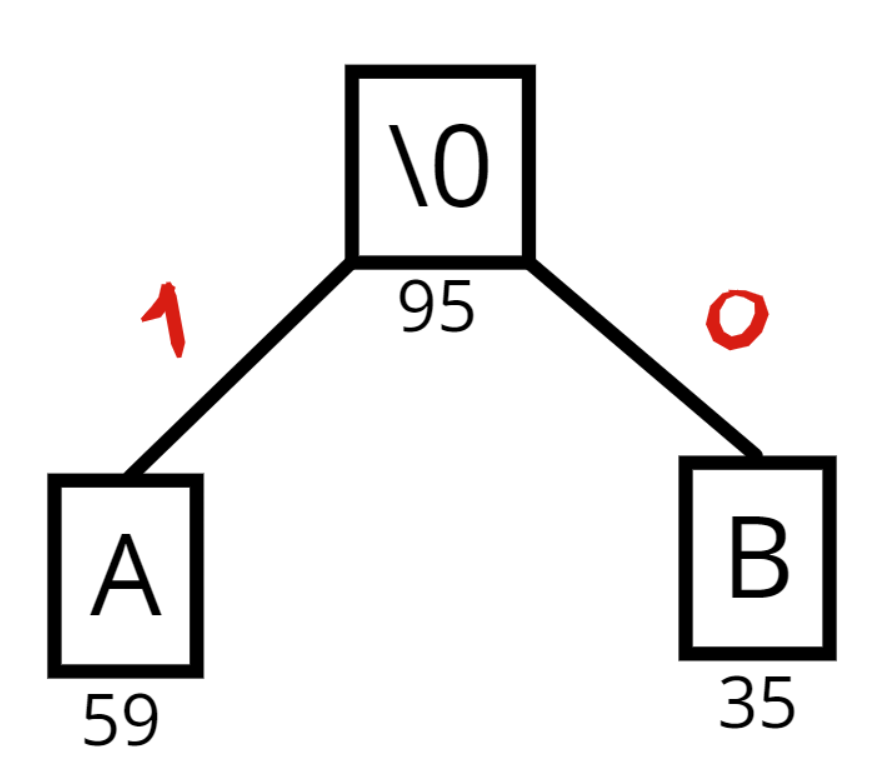


Рисунок 2 – Определение кодов

В результате получаем коды: A – 1, B – 0. Для того, чтобы сжать исходный текст, каждый из его символов заменяется соответствующим кодом. В результате получается последовательность нулей и единиц:

1111111111111111111111111111111111111111111111111111111111100000000000000000000000000000000000

Недостающие для заполнения байта биты заполняются нулями. Количество дополнительных нулей (< 8) записывается в начало бинарного файла. Затем записывается длина строковой репрезентации дерева, разделительный символ ‘#’, само дерево (01B1A) и, сжатая в байты, последовательность битов.

При декодировании сначала считывается кол-во дополнительных нулей, затем генерируется дерево. Последовательность битов, считанная из файла, используется для обхода дерева и идентификации закодированных символов. Ранее установленные дополнительные нули при декодировании не учитываются.

Результат декодирования:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

## Структуры данных, используемые в программе

В данном алгоритме используются структуры данных:

* struct Node – структура, описывающая узел дерева
* std::unordered\_map<char, int> freq – структура данных для хранения частоты символов
* std::priority\_queue<Node\*, std::vector<Node\*>, comp> node\_queue matrix – структура данных приоритетной очереди, в которой хранятся узлы дерева.
* struct comp – структура, описывающая операцию сравнения элементов приоритетной очереди

## Псевдокод

Алгоритм архивации:

1. Считываем текст из файла
2. Считаем частоту появления символов: find\_frequency(text);
3. Вызываем функцию для создания бинарного дерева: Node\* root = build\_tree(freq);
4. Создаём структуру для хранения пар символ-код std::unordered\_map<char, std::string> huffmanCode;
5. Определим коды символов: encode(root, “”, huffmanCode);
6. Заменяем символы исходного текста их кодами
7. Записываем в бинарный файл кол-во дополнительных нулей: bits = 8 – (str.size % 8);
8. Сгенерируем строковую репрезентацию бинарного дерева: writeBinaryTree(root, tree);
9. Запишем длину строковой репрезентации дерева в файл
10. Запишем строковую репрезентацию дерева в файл, отделив её символом «#»
11. Запишем биты и дополнительные нули в файл: writeBinaryString(outfile, str); Flush\_Bits(std::ostream& outfile);

Генерация дерева Хаффмана:

1. Создаём приоритетную очередь для хранения узлов.
2. Заполним очередь узлами: node\_queue.push(addNode(pair.first, pair.second, nullptr, nullptr));
3. While(в очереди не останется один узел) {
4. Запоминаем два узла с наивысшей частотой
5. Создаём новый узел с частотой, равной сумме их частот, после чего их удаляем.
6. Записываем созданный узел в очередь
7. }
8. Возвращаем указатель на оставшийся узел – корень дерева

Определение кодов символов encode():

1. Первичный вызов функции с указателем на корень дерева, пустой строкой и структурой для хранения значений символ-код
2. if(текущий указатель на узел пуст) { выход из функции }
3. if(текущий узел – лист) { сопоставить символу этого узла код из строки}
4. Рекурсивный вызов функции для левого потомка, копирование в неё строки с добавленным нулём
5. Рекурсивный вызов функции для правого потомка, копирование в неё строки с добавленной единицей

Генерация строковой репрезентации бинарного дерева:

1. Первичный вызов функции с указателем на корень дерева и ссылкой на пустую строку
2. If(текущий узел – лист) { вписать единицу и символ узла }
3. Else { вписать ноль и осуществить рекурсивный вызов для левого и правого потомков }

Запись битов в файл:

1. Объявим статическую переменную буфера: static unsigned char bit\_buffer;
2. Выполним побитовый сдвиг влево с присваиванием

bit\_buffer <<= 1;

1. If(бит – единица) { выполним побитовое «или» с присваиванием bit\_buffer |= 0x1; }
2. If(считали восемь битов) { запишем буфер в бинарный файл, обнулим буфер }

Декодирование:

1. Считаем кол-во дополнительных нулей int zeros;
2. Считаем строковую репрезентацию дерева parse\_tree(infile, tree)
3. Считаем закодированный текст в двоичной форме parse\_binary\_text(infile, text, tree\_size, zeros);
4. Сгенерируем дерево Хаффмана: readBinaryTree(tree, index);
5. While(индекс не достигнет конца текста) {
6. Вызов рекурсивной функции декодирования
7. }
8. Запись результатов декодирования в файл

Парсинг битов:

1. Объявляем переменную-буфер char c;
2. Считываем из файла до знака «#»
3. Зная размер дерева, пропускаем его
4. Считываем текст: while(успешно считали символ) {
5. Сохранение в строку двоичного кода считанного символа
6. }
7. Удаляем с конца строки дополнительные нули

# Инструкция пользователя

## Запуск программы

Запуск программы происходит из командной строки:

*Адрес\_к\_исполняемому\_файлу\Huffmans\_Algorithm*.exe *Адрес\_к\_входному\_файлу\название\_входного\_файла.txt [-en | -encode | -de | -decode]*

## Формат входного файла

Текст на английском языке.

## Формат выходного файла

* При кодировании:

Бинарный файл, содержащий строковую репрезентацию бинарного дерева и закодированный текст

* При декодировании:

Файл с декодированным текстом.

# Тестовые примеры

## Тест 1

До сжатия: 20.1 Кб. После – 11.1 Кб

|  |  |
| --- | --- |
| До выполнения | После выполнения |
| “input6.txt”:  In my younger and more vulnerable years… have control of things." | “input6 – encoded.txt”:  6197#00001r001b1,01p00001S01;0001(1)1101в1Ђ01B1M1'01-01W01q1N1e001s1i01n1o0000001k1"00001A1E01D001L00191501Љ1R01!1:001x01Y1C01j1H1v1l1a001d01f1y1t00001c1g01w01.001T001z1G01?001O001V1ґ01Г11F1I001m1u1h1 ПoҐ~·Х±!е©фp… N>KэМ3рv°xч«ЮХ±LР@ |
| Cmd:  Source text: In my younger and more vulnerable years… have control of things."  Found characters frequency:  9 1  y 347  I 119  . 172  …  h 11011  111  Binary code: 11001111011011111…000001  ‘0’-bits added: 6  Tree transcription: 000001r001b1…u1h1 |

## Тест 2

До разархивации: 11.1 Кб. После – 19.7 Кб

|  |  |
| --- | --- |
| До выполнения | После выполнения |
| “input6 – encoded.txt”:  6197#00001r001b1,01p00001S01;0001(1)1101в1Ђ01B1M1'01-01W01q1N1e001s1i01n1o0000001k1"00001A1E01D001L00191501Љ1R01!1:001x01Y1C01j1H1v1l1a001d01f1y1t00001c1g01w01.001T001z1G01?001O001V1ґ01Г11F1I001m1u1h1 ПoҐ~·Х±!е©фp… N>KэМ3рv°xч«ЮХ±LР@ | “input6 – decoded.txt”:  In my younger and more vulnerable years… have control of things." |
| Cmd:  Tree origin: 000001r001b1…u1h1  Binary code: 11001111011011111…000001  Decoded text: In my younger and more vulnerable years… have control of things." |

## Тест 3

До сжатия: 20.1 Кб. После – 11.1 Кб

|  |  |
| --- | --- |
| До выполнения | После выполнения |
| “input1.txt”:  BCAADDDCCACACACBCAADDDCCACACAC | “input1 - encoded.txt”:  811#01C001B1D1AЏ¶›hыi¶ |
| Source text: BCAADDDCCACACACBCAADDDCCACACAC Found characters frequency:  B 2  C 12  A 10  D 6  Binary code: 1000111110…0110  ‘0’-bits added: 8  Tree transcription: 01C001B1D1A |

# Список литературы

1. a. Т. Кормен, Ч. Лейзерсон, Р. Ривест, “Алгоритмы: построение и анализ”

Год, код-во страниц редакция