МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»
КАФЕДРА № 25

ПРЕПОДАВАТЕЛЬ
Доцент, канд. техн. наук Линский Е.М.

ОТЧЕТ О КУРСОВОЙ РАБОТЕ

по курсу: **ОСНОВЫ ПРОГРАММИРОВАНИЯ**

РАБОТУ ВЫПОЛНИЛ
СТУДЕНТ ГР. № 2355 Чадов А.А.

Санкт-Петербург 2024

Содержание

1 Постановка задачи

Задачей данной курсовой работы является разработка программы, реализующей алгоритм Хаффмана. Программа должна эффективно строить дерево Хаффмана и использовать его для сжатия и разжатия одиночных файлов. Особое внимание уделяется оптимальному хранению и передаче дерева, чтобы минимизировать дополнительные накладные расходы на хранение метаданных.

Алгоритм Хаффмана является одним из классических алгоритмов сжатия данных без потерь. Его основная идея заключается в кодировании символов текста переменной длиной кодов, где более часто встречающиеся символы получают более короткие коды. Такой подход позволяет значительно уменьшить размер файла.

1.1 Формулировка задачи

Разработать программу, которая:

1. Архивирует текстовые файлы, используя алгоритм Хаффмана:
   - строит дерево кодирования на основе частот символов в файле;
   - создает сжатый файл, содержащий закодированные данные и информацию о дереве.
2. Разархивирует файлы, сжатые с использованием данного алгоритма:
   - восстанавливает исходное дерево кодирования;
   - декодирует данные и восстанавливает оригинальный файл.
3. Эффективно передает дерево кодирования в сжатом виде.

1.2 Требования к программе

1. Программа должна поддерживать архивирование и разархивирование одиночных текстовых файлов.
2. Интерфейс программы должен быть удобным, а взаимодействие с пользователем — интуитивно понятным.
3. Кодирование дерева:

- дерево должно быть представлено минимально возможным объемом данных, чтобы сократить общий размер архивированного файла.

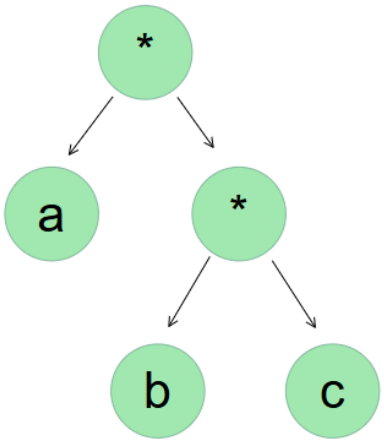
4. Сжатие данных:
   - программа должна продемонстрировать экономию памяти на реальных данных.
5. Кодирование и декодирование должны быть надежными и корректными: при разархивировании содержимое файла должно быть идентично исходному.

1.3 Примеры

Рассмотрим строку для сжатия: `"aaabbc"`.
Частоты символов:

- `a` — 3
- `b` — 2
- `c` — 1

На основе этих частот строится дерево Хаффмана. Пример дерева:



Коды символов:

- `a` → 0

- `b` → 10
- `c` → 11

Сжатый файл будет содержать:

1. Закодированные данные: `0001011`.
2. Данные о дереве, позволяющие восстановить структуру для декодирования.

Декодирование сжатого файла должно привести к восстановлению строки `"aaabbc"`.

1.4 Пример значимости задачи

Сжатие данных используется в повседневной жизни: при передаче файлов через интернет, хранении данных на жестких дисках, обработке мультимедиа. Например, программы сжатия ZIP или алгоритмы в аудио- и видеокодеках основываются на подобных принципах.

2 Алгоритм

2.1 Идея алгоритма

Алгоритм Хаффмана основан на принципе построения префиксного кодирования. Идея заключается в том, чтобы символы с более высокой частотой появления в тексте кодировались более короткими двоичными кодами, а менее частые символы — более длинными. Таким образом достигается компрессия: общее количество битов, необходимых для представления текста, уменьшается.

2.2 Подробное описание алгоритма

Алгоритм состоит из следующих этапов:

1. **Подсчёт частот символов:**
    - На вход подаётся строка, из которой составляется таблица частот появления каждого символа.

2. **Построение дерева Хаффмана:**
    - Используется приоритетная очередь (min-heap), где хранятся узлы дерева, отсортированные по частоте.
    - Каждый узел представляет либо символ, либо объединение нескольких узлов.
3. **Генерация кодов:**
    - После построения дерева каждому символу присваивается уникальный код, основанный на пути от корня дерева:
        - Левый переход в дереве добавляет 0 в код.
        - Правый переход добавляет 1 в код.
4. **Кодирование текста:**
    - Символы исходного текста заменяются их двоичными кодами.
5. **Декодирование текста:**
    - Закодированный текст раскодируется, проходя по дереву Хаффмана от корня к листьям.

2.3 Используемые структуры данных

- **Приоритетная очередь (min-heap):**
    - Для хранения узлов дерева Хаффмана.
    - Каждый узел содержит:
        * Символ (если это листовой узел).
        * Частоту появления символа.
        * Указатели на левого и правого потомков.
- **Дерево Хаффмана:**
    - Бинарное дерево, где:
        * Листовые узлы содержат символы.
        * Внутренние узлы содержат сумму частот своих дочерних узлов.
- **Хэш-таблица:**
    - Хранит соответствие символов их двоичным кодам.

2.4 Иллюстрация алгоритма

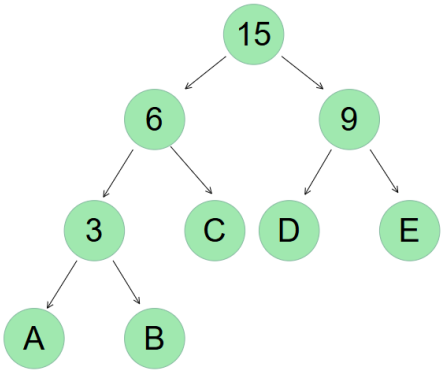
Пример Текст: `"ABBCCCDDDDEEEEE"`
Шаг 1: Подсчёт частот символов

A	1
B	2
C	3
D	4
E	5

Шаг 2: Построение дерева Хаффмана

1. Создаём узлы для каждого символа: `A:1`, `B:2`, `C:3`, `D:4`, `E:5`.
2. Объединяем два узла с наименьшими частотами: `A:1 + B:2 → AB:3`. Результат: `[AB:3, C:3, D:4, E:5]`.
3. Повторяем объединение: `AB:3 + C:3 → ABC:6`. Результат: `[ABC:6, D:4, E:5]`.
4. Далее: `D:4 + E:5 → DE:9`. Результат: `[ABC:6, DE:9]`.
5. Наконец: `ABC:6 + DE:9 → ABCDE:15`.

Итоговое дерево:



Шаг 3: Генерация кодов

A	000
B	001
C	01
D	10
E	11

Шаг 4: Кодирование текста Текст `"ABBCCCDDDDEEEEE"` преобразуется в: `000 001 001 01 01 01 10 10 10 10 11 11 11 11 11`

Шаг 5: Декодирование текста Путём обхода дерева из закодированного текста восстанавливаем исходный текст.

2.5 Псевдокод

```
// Загрузка текста из файла
// Чтение всего содержимого файла в строку
// Подсчёт частот символов

// Построение дерева Хаффмана
priority_queue<Node*, vector<Node*>, Comparator> pq;
for (auto& [ch, count] : freq) {
   // Создать листовой узел и добавить в очередь
}
while (pq.size() > 1) {
   // Извлечь узел с наименьшей частотой
   // Извлечь следующий узел с наименьшей частотой
   // Создать новый узел
   // Поместить новый узел обратно в очередь
}
// Корень дерева

// Кодирование символов
unordered_map<char, string> huffmanCode;
function<void(Node*, string)> encode = [&](Node* node, string str) {
   // Базовый случай
   if (!node->left && !node->right) {
       // Записать код для листового узла
        return;
    }
```

```
    // Рекурсивно идти влево
    // Рекурсивно идти вправо
};
encode(root, "");

// Кодирование текста
for (char ch : text) {
    // Заменить каждый символ его кодом
}

// Запись закодированного текста в бинарный файл
// Записать длину строки
// Преобразовать строку битов в байты и записать

// Декодирование текста из бинарного файла
// Прочитать длину строки
while (encodedFile.peek() != EOF) {
    // Прочитать байт
    for (int i = 7; i >= 0; --i) {
        // Преобразовать байт в строку битов
    }
}
// Обрезка до реальной длины

// Раскодирование текста
Node* currentNode = root;
for (char bit : binaryStr) {
    // Перемещение по дереву
    if (!currentNode->left && !currentNode->right) {
        // Найден символ
        // Вернуться к корню дерева
    }
}

// Запись раскодированного текста в файл
```

3 Сложность алгоритма

- **Подсчёт частот:** $O(n)$, где n — длина текста.
- **Построение дерева:** $O(m \log m)$, где m — число уникальных символов (вставка в приоритетную очередь).
- **Генерация кодов:** $O(m)$, так как каждый символ посещается один раз.
- **Кодирование текста:** $O(n)$.
- **Декодирование текста:** $O(n)$.

Итоговая сложность: $O(n + m \log m)$,
где n — длина текста, m — число уникальных символов.

4 Инструкция пользователя

После запуска программы выводится предложение ввести путь до входного файла, после чего отображается результат работы программы.
Формат входного файла: `.txt`, в котором хранится текст в любом виде.
Выходные файлы: закодированный текст в формате `.bin` и декодированный текст в формате `.txt`.

5 Дополнительное задание

Сравнивать размеры исходного/сжатого файлов и текстов, выводить среднюю длину символа.

6 Тестовые примеры

Пример 1

```
   Enter file path and name:
I:\testt.txt
```
Не удалось открыть файл: `I:\testt.txt`
Ошибка: входной файл пуст или не найден!

Пример 2

```
   Enter file path and name:
I:\test\input.txt
```

Закодированная строка:

```
1101101101110010001010001011100110101000001101011111001110011011111010110101110101
```

Huffman Codes (Character → Code):

- 'o' → 000
- 'f' → 0010
- 't' → 0011
- 's' → 0101
- 'c' → 0100
- 'a' → 011
- 'm' → 10001
- 'e' → 10000
- '.' → 111010
- 'n' → 1001
- ' ' → 101
- 'l' → 110000
- 'p' → 110001
- 'g' → 11001
- 'd' → 11010
- 'H' → 110110
- 'u' → 110111
- 'r' → 11100
- 'h' → 111011

- 'i' → 1111

Процесс завершён. Проверьте файлы encoded.bin и decoded.txt.
File Compression Statistics:

- Input file size: 46 bytes
- Encoded file size: 32 bytes
- Compression ratio: 30.4348%

Text Compression Statistics:

- Original text size: 368 bits
- Encoded text size: 190 bits
- Compression ratio: 48.3696%
- Average code length: 4.13043 bits/symbol

Пример 3

```
   Enter file path and name:
I:\test\input.txt
```

Закодированная строка:

```
11111111111111111111111111111110111111111111111111111111111111111111111
```

Huffman Codes (Character → Code):

- 'd' → 0
- 'a' → 1

Процесс завершён. Проверьте файлы encoded.bin и decoded.txt.
File Compression Statistics:

- Input file size: 76 bytes
- Encoded file size: 18 bytes
- Compression ratio: 76.3158%

Text Compression Statistics:

- Original text size: 608 bits
- Encoded text size: 76 bits
- Compression ratio: 87.5%
- Average code length: 1 bit/symbol

7 Список литературы

Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн — *Алгоритмы. Построение и анализ. Издание 3-е (2013).*