Lab 2

Status	Not Started

Лабораторная работа: Реализация алгоритма Хаффмана для сжатия и восстановления файлов

Цель работы

Освоить принципы кодирования информации с переменной длиной кода на примере алгоритма Хаффмана. Научиться использовать структуры данных для построения эффективных алгоритмов сжатия и восстановления файлов.

Постановка задачи

Необходимо разработать программу, реализующую алгоритм сжатия и последующего восстановления данных с использованием метода кодирования Хаффмана.

Вариант 1 (оценка 1-7)

Программа должна:

1. Анализ исходного файла и определение частоты появления символов.
2. Построение кодов символов на основе дерева Хаффмана.
3. Кодирование исходного файла с использованием построенных кодов.
4. Сохранение сжатого файла в отдельный бинарный файл.
5. Декодирование полученного сжатого файла и восстановление исходных данных.
6. Проверка корректности восстановления (сравнение исходного и декодированного файлов).

Вариант 2 (оценка 1-10)

Программа должны работать в двух режимах: encode и decode.
В режиме encode программа должна:

1. Проанализировать исходный файл и определить частоты символов.
2. Построить таблицу кодов символов на основе алгоритма Хаффмана.
3. Закодировать исходный файл с использованием построенных кодов.
4. Записать в результирующий файл информацию о дереве Хаффмана или о частотах символов.
5. Сохранить в результирующем файле закодированную последовательность бит.

В режиме decode программа должна:

1. Прочитать из данного файла информацию о дереве Хаффмана или о частотах символов.
2. Построить дерево Хаффмана на основе полученной информации.
3. Декодировать последовательность бит, записанную в файле.
4. Сохранить результат в результирующий файл.

Входные и выходные данные

- **Входной файл:** произвольный текстовый файл.
- **Выходные файлы:**
  - файл сжатых данных (в бинарном формате);
  - файл, полученный после декодирования (для проверки совпадения с исходным).

Требования к программе

- Программа должна обеспечивать корректную обработку любого входного файла.

- **(только во 2 варианте)** Сжатый файл должен содержать необходимую информацию для восстановления исходного текста.
- **(только во 2 варианте)** Должна быть обеспечена возможность повторного декодирования без потери данных.
- **(только во 2 варианте)** Пользователь должен иметь возможность выбрать режим работы программы: **кодирование** или **декодирование**.
- Необходимо предусмотреть вывод краткой информации о ходе работы программы (например, статистика частот, длина кодов, коэффициент сжатия и т.д.).

Результаты работы

В результате выполнения лабораторной работы студент должен:

- продемонстрировать работу программы на тестовом примере;
- подтвердить совпадение исходного и восстановленного файлов;

Теоретическая часть: Алгоритм Хаффмана

Введение

Алгоритм Хаффмана — это классический метод **оптимального префиксного кодирования** символов, разработанный Дэвидом Хаффманом в 1952 году. Он используется для **сжатия данных без потерь**, что означает возможность полного восстановления исходной информации после декодирования.

Основная идея алгоритма заключается в том, что символам, которые встречаются чаще, присваиваются **короткие двоичные коды**, а редким символам — **более длинные коды**. Это позволяет значительно уменьшить общий объём закодированного сообщения по сравнению с кодированием фиксированной длины.

Основные этапы алгоритма Хаффмана

1. Подсчёт частот символов

На первом этапе производится анализ входного файла и подсчёт частоты появления каждого символа. Для этого:

- Программа последовательно считывает все символы из файла.
- Для каждого уникального символа подсчитывается количество его вхождений.
- Результат сохраняется в виде таблицы частот (например, в виде словаря или массива).

Пример: Для строки `AABBBCCCC` частоты будут: A=2, B=3, C=4.

2. Построение дерева Хаффмана

2.1 С помощью min-heap (сложнее)

Дерево Хаффмана строится снизу вверх с использованием **жадного алгоритма**. Процесс построения выглядит следующим образом:

1. Для каждого уникального символа создаётся узел (лист дерева) с весом, равным его частоте.
2. Все созданные узлы помещаются в **приоритетную очередь** (min-heap), упорядоченную по возрастанию весов.
3. Из очереди извлекаются два узла с наименьшими весами.
4. Создаётся новый внутренний узел, который становится родителем для извлечённых узлов. Вес нового узла равен **сумме весов дочерних узлов**.
5. Новый узел помещается обратно в приоритетную очередь.
6. Шаги 3-5 повторяются до тех пор, пока в очереди не останется один узел — это и будет **корень дерева Хаффмана**.

Важно: Узел с меньшим весом обычно становится левым потомком, а с большим — правым (хотя порядок может варьироваться в зависимости от реализации).

2.2 С помощью упорядоченного по частотам списка (проще)

1. Для каждого уникального символа создаётся узел (лист дерева) с весом, равным его частоте.

2. Все созданные узлы помещаются в **связный список**, упорядоченный по возрастанию весов.
3. Из списка извлекаются два узла, стоящих в начале.
4. Создается новый внутренний узел, который становится “родителем” для извлеченных узлов (извлеченные узлы больше не стоят в списке, а свисают из-под него), вес нового узла равен **сумме весов дочерних узлов.**
5. Новый узел помещается обратно в связный список так, чтобы полученный список оказался упорядоченным по возрастанию весов.
6. Шаги 3-5 повторяются до тех пор, пока в очереди не останется один узел — это и будет **корень дерева Хаффмана**.

3. Формирование кодов символов

После построения дерева необходимо сформировать двоичные коды для каждого символа. Это делается путём **обхода дерева от корня к листьям**:

- При переходе к **левому потомку** к текущему коду добавляется бит `0`.
- При переходе к **правому потомку** добавляется бит `1`.
- Когда достигается лист дерева (узел, содержащий символ), накопленная последовательность битов становится кодом этого символа.

Обход можно реализовать рекурсивно (depth-first search) или итеративно с использованием стека.

Пример кодов: Для строки `AABBBCCCC` возможные коды могут быть: A=`00`, B=`01`, C=`1`.

4. Кодирование данных

На этапе кодирования исходный текст преобразуется в последовательность битов:

1. Каждый символ исходного файла заменяется на его двоичный код согласно таблице кодов Хаффмана.
2. Все коды объединяются в единый **битовый поток**.

3. Полученный битовый поток записывается в выходной файл.

5. Декодирование данных

Процесс декодирования восстанавливает исходный текст из сжатого битового потока:

1. Последовательно считываются биты из битового потока.
2. Начиная от корня дерева, программа спускается по дереву: при чтении `0` — переход к левому потомку, при `1` — к правому.
3. Когда достигается лист дерева, извлекается соответствующий символ и добавляется к декодированному тексту.
4. Процесс начинается заново от корня дерева для следующей последовательности битов.
5. Декодирование продолжается до конца битового потока.

Особенности и свойства алгоритма Хаффмана

Префиксное кодирование

Коды Хаффмана являются **префиксными** (prefix-free), что означает, что ни один код не является началом (префиксом) другого кода. Это важное свойство обеспечивает **однозначность декодирования** без использования разделителей между кодами.

Пример: Если коды символов A=`0`, B=`10`, C=`11`, то последовательность `01011` однозначно декодируется как `ABCC`.

Оптимальность кодирования

Алгоритм Хаффмана обеспечивает **оптимальное кодирование** при условии, что вероятности (частоты) появления символов известны заранее. Это означает, что никакой другой метод префиксного кодирования не может дать более короткое среднее представление символов для заданного распределения частот.

Коэффициент сжатия

Эффективность сжатия зависит от распределения частот символов:

- Чем больше различие в частотах, тем выше коэффициент сжатия.
- Для равномерно распределённых символов сжатие может быть минимальным или отсутствовать.
- В худшем случае (все символы встречаются с одинаковой частотой) коды могут оказаться такой же длины, как и при фиксированном кодировании.

Применение алгоритма

Алгоритм Хаффмана широко используется в современных технологиях сжатия данных:

- **ZIP и GZIP** — архиваторы файлов использует комбинацию алгоритма Хаффмана и LZ77.
- **JPEG** — формат изображений применяет кодирование Хаффмана на этапе энтропийного кодирования.
- **MP3** — аудиокодек использует модифицированный алгоритм Хаффмана для сжатия данных.
- **PNG** — формат изображений использует алгоритм DEFLATE, который включает кодирование Хаффмана.
- **Факс-передача (CCITT Group 3)** — стандарт сжатия факсимильных сообщений.

Сложность алгоритма

Временная сложность алгоритма Хаффмана составляет **O(n log n)**, где n — количество уникальных символов. Это обусловлено операциями с **приоритетной очередью** при построении дерева. Пространственная сложность — **O(n)** для хранения дерева и таблицы кодов.

Ограничения

Несмотря на свою эффективность, алгоритм Хаффмана имеет некоторые ограничения:

- Требуется два прохода по данным: один для подсчёта частот, второй для кодирования.

- Необходимо сохранять дерево или таблицу частот вместе со сжатыми данными для возможности декодирования.
- Для файлов с равномерным распределением символов эффективность сжатия низкая.
- Не учитывает контекстные зависимости между символами (в отличие от более сложных алгоритмов типа арифметического кодирования).

Варианты алгоритма

Существуют различные модификации алгоритма Хаффмана:

- **Адаптивный Хаффман** — динамически обновляет дерево в процессе кодирования, не требуя предварительного подсчёта частот.
- **Canonical Huffman** — использует специальную форму кодов, которая позволяет более эффективно хранить таблицу кодов.
- **Length-limited Huffman** — ограничивает максимальную длину кодов, что полезно для аппаратной реализации.

Рекомендации по реализации

Для реализации алгоритма Хаффмана рекомендуется использовать модульную структуру программы и заранее определить основные структуры данных и функции, обеспечивающие удобное построение и использование дерева кодирования.

---

1. Структуры данных

1. **Узел дерева (Node)**

   Каждый узел дерева Хаффмана может содержать:

   - символ (для листьев);
   - частоту (вес);
   - указатели на левый и правый дочерние узлы.

```
typedef struct Node {
    unsigned char symbol;
```

```
    unsigned int freq;
    struct Node *left, *right;
} Node;
```

2. **Минимальная куча (Min-Heap)**

   Используется для эффективного выбора двух узлов с минимальной частотой при построении дерева.

   Можно реализовать собственную кучу или воспользоваться стандартными средствами языка (например, `priority_queue` в C++, `heapq` в Python).

```
typedef struct {
    int size;
    int capacity;
    Node **array;
} MinHeap;
```

---

2. Основные этапы программы и функции

Программа обычно делится на две основные части — **кодирование** и **декодирование**.

Рекомендуется реализовать следующие функции:

Функция	Назначение
`build_frequency_table()`	Подсчёт частот символов во входном файле
`create_min_heap()`	Инициализация пустой кучи
`insert_heap()` / `extract_min()`	Добавление и извлечение узлов с минимальной частотой
`build_huffman_tree()`	Построение дерева Хаффмана на основе частот
`generate_codes()`	Рекурсивное формирование кодов для символов при обходе дерева
`encode_file()`	Преобразование исходного файла в последовательность бит и запись в бинарный файл

Функция	Назначение
`decode_file()`	Восстановление исходных данных по дереву или таблице кодов
`free_tree()`	Освобождение памяти, занятой структурой дерева

3. Работа с файлами

- Для корректной побитовой записи и чтения файлов необходимо открывать их в **бинарном режиме** (`"rb"` и `"wb"`).

  Это предотвращает автоматические преобразования символов перевода строки и других управляющих байтов.
- При записи кодированного потока удобно использовать **буфер для битов**: накапливать отдельные биты в байт и записывать его, когда буфер заполнен.
- В сжатом файле следует хранить не только закодированные данные, но и **служебную информацию** — например, таблицу частот или длину кодов, чтобы можно было восстановить дерево при декодировании.

4. Проверка корректности

После реализации алгоритма необходимо убедиться, что:

1. Декодированный файл полностью совпадает с исходным.
2. Программа корректно работает с разными типами данных и размером файлов.
3. При повторном кодировании и декодировании данные не искажаются.
4. Вычисляемый коэффициент сжатия соответствует ожиданиям (меньше 1 для обычного текста).

Минимальная куча: теория и советы по реализации

1. Понятие минимальной кучи

Минимальная куча (min-heap) — это структура данных, представляющая собой **практически полное бинарное дерево**, в котором выполняется **свойство кучи**:

> Для любого узла значение (или приоритет) не меньше, чем значение его родителя.

Таким образом, **наименьший элемент всегда находится в корне дерева**, что позволяет быстро извлекать минимум за логарифмическое время.

---

2. Структура и представление

Минимальную кучу удобно хранить **в виде массива**, а не в виде явного дерева.

Для элемента с индексом `i`:

- **левый потомок** имеет индекс `2*i + 1`;
- **правый потомок** имеет индекс `2*i + 2`;
- **родитель** — индекс `(i - 1) / 2`.

Пример:

```
Индексы:   0   1   2   3   4
Значения: [3,  5,  8,  9,  7]
```

Здесь корень (3) — минимальный элемент, а структура соответствует почти полному дереву.

---

3. Основные операции

Операция	Назначение	Сложность
`insert()`	Добавление нового элемента с сохранением свойства кучи	O(log n)
`extractMin()`	Извлечение и удаление минимального элемента (корня)	O(log n)
`heapify()`	Восстановление свойства кучи после изменений	O(log n)

Операция	Назначение	Сложность
`buildHeap()`	Построение кучи из неупорядоченного массива	O(n)

4. Применение в алгоритме Хаффмана

При построении дерева Хаффмана на каждом шаге нужно выбирать **два узла с минимальной частотой** и объединять их в новый узел.

Использование минимальной кучи позволяет:

- быстро находить два наименьших узла (`extractMin()` дважды),
- добавлять новый объединённый узел обратно (`insert()`),
- выполнять все операции за `O(log n)`.

Без кучи построение дерева было бы гораздо медленнее (`O(n²)` при линейном поиске минимума).

5. Советы по реализации

⚙️ Структура элемента

Элементом кучи должен быть **указатель на узел дерева**, который содержит частоту символа (вес).

Например:

```
typedef struct Node {
    unsigned char symbol;
    unsigned int freq;
    struct Node *left, *right;
} Node;
```

⚙️ Основные функции

```
void insertHeap(Node* node);      // вставка нового узла
Node* extractMin();               // извлечение минимального узла
void heapify(int index);          // восстановление свойств кучи
```

⚙️ Приоритет

Сравнение элементов должно выполняться **по частоте** (`freq`).

Если частоты равны, можно использовать дополнительное условие (например, по символу) для стабильности.

⚙️ Построение кучи

- После заполнения массива узлов вызовите `buildHeap()` для формирования корректной структуры.
- При вставке и удалении узлов всегда поддерживайте инвариант:

  `parent.freq ≤ children.freq` .

---

6. Пример логики работы

Пусть даны частоты символов:

`A:5, B:9, C:12, D:13, E:16, F:45`

Построение:

1. Все узлы помещаются в кучу.
2. Извлекаются два минимальных (A:5 и B:9) → создаётся новый узел (freq=14).
3. Новый узел добавляется обратно.
4. Повторяется, пока не останется один узел (корень дерева).

---

7. Типичные ошибки

1. Неправильное индексирование потомков или родителя.
2. Потеря связи между узлами после извлечения минимума.
3. Неверное сравнение при одинаковых частотах.
4. Неинициализированные указатели при объединении узлов.
5. Нарушение структуры после вставки (отсутствие `heapify`).