Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

КУРСОВОЙ ПРОЕКТ

по курсу

«Структуры и алгоритмы обработки данных»

Вариант 45

Выполнил: студент группы ИП-413

Цыренов Ч.Х.

Проверил: доцент кафедры ПМиК

Янченко Е.В.

Новосибирск – 2025

СОДЕРЖАНИЕ

1. ПОСТАНОВКА ЗАДАЧИ

Цель проекта: Разработка программного комплекса на языке С для выполнения комплексной обработки структурированных данных, хранящихся в бинарном файле. Проект направлен на практическое применение фундаментальных алгоритмов и структур данных, оценку их эффективности и особенностей реализации.

Конкретный вариант задания (B=4, C=3, S=1, D=4, E=1):

1. **Источник данных (B=4):** База данных «Населенный пункт», содержащаяся в файле testBase4.dat. Объем базы — 4000 записей.
2. **Структура записи:** Каждая запись в файле имеет фиксированный размер 64 байта и следующую структуру:

Структура записи:

ФИО гражданина: текстовое поле 32 символа

формат <Фамилия>_<Имя>_<Отчество>

Название улицы: текстовое поле 18 символов

Номер дома: целое число

Номер квартиры: целое число

Дата поселения: текстовое поле 10 символов

формат дд-мм-гг

Пример записи из БД:

Петров_Иван_Федорович_____

Ленина_____

10

67

29-02-65

3. **Задачи по обработке:**

   - **Загрузка и отображение:** Динамически загрузить всю базу данных в оперативную память и реализовать постраничный вывод содержимого на экран.
   - **Сортировка (S=1):** Упорядочить данные с использованием **метода Вильямса-Флойда (пирамидальная сортировка)**. Сортировка должна выполняться косвенно, через массив указателей, без физического перемещения записей.
   - **Ключ сортировки (C=3):** Упорядочивание производится по **Дате поселения** (первичный ключ, по возрастанию) и **Названию улицы** (вторичный ключ, лексикографически).
   - **Быстрый поиск (C=3):** Реализовать **двоичный поиск** по отсортированному массиву. Ключом поиска является **год поселения** (две последние цифры). Все найденные записи должны быть помещены в динамическую очередь.
   - **Дерево поиска (D=4):** Из записей, помещенных в очередь,

построить **Дерево оптимального поиска, используя приближенный алгоритм А2**. Ключом для построения и поиска в дереве является **номер квартиры**.

- **Кодирование (E=1):** Выполнить статическое кодирование исходного файла базы данных с помощью **алгоритма Хаффмена**. Необходимо предварительно рассчитать вероятности всех символов, вывести результирующую кодовую таблицу, а также вычислить энтропию источника и среднюю длину кодового слова для оценки эффективности сжатия.

2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

2.1. Метод сортировки: Пирамидальная сортировка (Метод Вилльямса-Флойда)

Пирамидальная сортировка основана на алгоритме построения пирамиды.

Последовательность a_i, a_{i+1}, …, a_k называется (i,k)-пирамидой, если неравенство $a_j \leq min(a_{2j}, a_{2j+1})$(*) выполняется для каждого j = i, …, k, для которого хотя бы один из элементов a_{2j}, a_{2j+1} существует.

Например, массив A является пирамидой, а массив B не является. A = (a_2, a_3, a_4, a_5, a_6, a_7, a_8) = (3, 2, 6, 4, 5, 7) B = (b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (3, 2, 6, 4, 5, 7)

Свойства пирамиды

1. Если последовательность a_i, a_{i+1}, …, a_{k-1}, a_k является (i, k)-пирамидой, то последовательность a_{i+1}, …, a_{k-1}, полученная усечением элементов с обоих концов последовательности, является (i+1, k-1)-пирамидой.
2. Если последовательность a_1 … a_n — (1, n)-пирамида, то a_1 — минимальный элемент последовательности.
3. Если a_1, a_2 … $a_{\lfloor n/2 \rfloor}$, $a_{\lfloor n/2 \rfloor+1}$, … a_n — произвольная последовательность, то последовательность $a_{\lfloor n/2 \rfloor+1}$, …, a_n является $(\lfloor n/2 \rfloor + 1, \text{n})$-пирамидой.

Процесс построения пирамиды выглядит следующим образом. Дана последовательность a_{s+1}, …, a_k, которая является (s+1, k)-пирамидой. Добавим новый элемент x и поставим его на s-тую позицию в последовательности, т.е. пирамида всегда будет расширяться влево. Если выполняется (*), то полученная последовательность — (s, k)-пирамида. Иначе найдутся элементы a_{2s+1}, a_{2s}такие, что либо $a_{2s} < a_s$, либо $a_{2s+1} < a_s$.

Пусть имеет место первый случай, второй случай рассматривается аналогично. Поменяем местами элементы a_s и a_{2s}. В результате получим новую последовательность a'_s, a_{s+1}, …, a'_{2s}, …, a_k. Повторяем все действия для элемента a'_{2s}и т.д., пока не получим (s, k)-пирамиду.

Пирамидальная сортировка производится в два этапа. Сначала строится пирамида из элементов массива. По свойству (3) правая часть массива является

($\lfloor n/2 \rfloor + 1$, n)-пирамидой. Будем добавлять по одному элементу слева, расширяя пирамиду, пока в неё не войдут все элементы массива. Тогда по свойству (2) первый элемент последовательности — минимальный.

Произведём двустороннее усечение: уберём элементы a_1 , a_n . По свойству (1) оставшаяся последовательность является (2, n-1)-пирамидой. Элемент a_1поставим на последнее место, а элемент a_nдобавим к пирамиде a_2, …, a_{n-1}слева. Получим новую (1, n-1)-пирамиду. В ней первый элемент является минимальным.

Поставим первый элемент пирамиды на позицию n-1, а элемент a_{n-1}добавим к пирамиде a_2, …, a_{n-2}, и т.д. В результате получим обратно отсортированный массив.

Общее количество операций сравнений и пересылок для пирамидальной сортировки: C ≤ $2nlogn + n + 2$, M ≤ $nlogn + 6.5n - 4$. Таким образом, C = $O(nlogn)$, M = $O(nlogn)$при $n \to \infty$.

Отметим некоторые свойства пирамидальной сортировки. Метод пирамидальной сортировки неустойчив и не зависит от исходной отсортированности массива.

2.2. Двоичный поиск

Алгоритм двоичного поиска в упорядоченном массиве сводится к следующему. Берём средний элемент отсортированного массива и сравниваем его с ключом X. Возможны три варианта:

- Выбранный элемент равен X. Поиск завершён.
- Выбранный элемент меньше X. Продолжаем поиск в правой половине

массива.

- Выбранный элемент больше X. Продолжаем поиск в левой половине массива.

Из-за необходимости найти все элементы, соответствующие заданному ключу поиска, в курсовой работе использовалась вторая версия двоичного поиска, которая из подходящих элементов находит самый левый. В результате для поиска остальных требуется просматривать лишь оставшуюся правую часть массива.

Верхняя оценка трудоёмкости алгоритма двоичного поиска такова. На каждой итерации поиска необходимо два сравнения для первой версии, одно сравнение для второй версии. Количество итераций не больше $\lceil \log_2(n+1) \rceil$. Таким образом, трудоёмкость двоичного поиска в обоих случаях составляет $O(\log n), \quad n \to \infty$

2.3. Дерево и поиск по дереву: Дерево оптимального поиска (Алгоритм A2)

Второй алгоритм (A2) использует предварительно упорядоченный набор вершин. В качестве корня выбирается такая вершина, что разность весов левого и правого поддеревьев была минимальна. Для этого путём последовательного суммирования весов определим вершину V_k, для которой справедливы неравенства:

$$\sum_{i=1}^{k-1} w_i < \frac{W}{2} \text{ и } \sum_{i=1}^{k} w_i \geq \frac{W}{2}.$$

Тогда в качестве «центра тяжести» может быть выбрана вершина V_k, V_{k-1}или V_{k+1}, т. е. вершина, для которой разность весов левого и правого поддерева минимальна. Далее действия повторяются для каждого поддерева.

Поскольку число возможных конфигураций из n вершин растёт экспоненциально с ростом n, то решение задачи построения ДОП при больших n методом перебора нерационально. Однако деревья оптимального поиска обладают свойствами, которые позволяют получить алгоритм построения ДОП, начиная с отдельных вершин с последовательным включением новых вершин в дерево. Далее будем считать, что множество вершин, входящих в дерево, упорядочено. Поскольку вес дерева остаётся неизменным, то вместо средневзвешенной высоты будем рассматривать взвешенную высоту дерева: $P = h_1w_1 + h_2w_2 + \cdots + h_nw_n$.

2.4. Метод кодирования: Код Хаффмена

Рассмотрим источник с алфавитом $A = \{a_1, a_2, \dots, a_n\}$ и вероятностями $p_1, \dots, p_n$. Пусть символы алфавита некоторым образом упорядочены, например, $a_1 \le a_2 \le \cdots \le a_n$. Алфавитным называется код, в котором кодовые слова лексикографически упорядочены, т.е. $\phi(a_1) \le \phi(a_2) \le \cdots \le \phi(a_n)$.

Побуквенный разделимый код называется оптимальным, если средняя длина кодового слова минимальна для данного распределения вероятностей символов. Избыточностью кода называется разность между средней длиной кодового слова и энтропией источника сообщений: $r = L_{\text{ср}} - H$. Задача эффективного неискажающего сжатия заключается в построении кодов с

наименьшей избыточностью, у которых средняя длина кодового слова близка к энтропии источника. К таким кодам относятся классические коды Хаффмена, Шеннона, Фано, Гильберта–Мура.

Побуквенное кодирование пригодно для любых сообщений. Однако на практике часто доступна дополнительная информация о вероятностях символов исходного алфавита. С использованием этой информации решается задача оптимального побуквенного кодирования.

Пусть имеется дискретный вероятностный источник, порождающий символы алфавита $A = \{a_1, \dots, a_n\}$ с вероятностями $p_i = p(a_i)$, где $\sum_{i=1}^{n} p_i = 1$. Основной характеристикой источника является его энтропия, которая представляет собой среднее количество информации в сообщении источника и определяется выражением (для двоичного случая):

$$H(p_1, \dots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i.$$

Энтропия характеризует меру неопределённости выбора для данного источника. Например, если $A = \{a_1, a_2\}$, $p_1 = 0$, $p_2 = 1$, т.е. источник может породить только символ a_2, то неопределённости нет и энтропия $H(p_1, p_2) = 0$. Максимальная энтропия достигается, если все символы равновероятны, например, $A = \{a_1, a_2\}$, $p_1 = 1/2$, $p_2 = 1/2$, тогда $H(p_1, p_2) = 1$.

Для практических применений важно, чтобы коды сообщений имели по возможности наименьшую длину. Основной характеристикой неравномерного кода является количество символов, затрачиваемых на кодирование одного

сообщения. Пусть имеется разделимый побуквенный код для источника, порождающего символы алфавита $A = \{a_1, \dots, a_n\}$ с вероятностями $p_i = p(a_i)$, $\sum_{i=1}^{n} p_i = 1$, состоящий из n кодовых слов с длинами $L_1, \dots, L_n$ в алфавите {0,1}. Средней длиной кодового слова называется величина

$$L_{\text{ср}} = \sum_{i=1}^{n} p_i L_i,$$

или среднее число кодовых букв на одну букву источника.

Алгоритм построения оптимального кода Хаффмена

1. Упорядочим символы исходного алфавита $A = \{a_1, \dots, a_n\}$ по убыванию их вероятностей: $p_1 \geq p_2 \geq \cdots \geq p_n$.
2. Если $A = \{a_1, a_2\}$, то $a_1 \to 0$, $a_2 \to 1$.
3. Если известны коды для $\{a_1, \dots, a_n\}$ в виде $\langle a_j \to b_j \rangle$, $j = 1, \dots, n$, то объединяем два символа с наименьшими вероятностями a'_j и a''_j в новый «символ» с вероятностью $p(a_j) = p(a'_j) + p(a''_j)$, назначая $a'_j \to b_j 0$, $a''_j \to b_j 1$, и повторяем процесс на уменьшенном множестве.

Пример. Пусть дан алфавит $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$ с вероятностями $p_1 = 0.36$, $p_2 = 0.18$, $p_3 = 0.18$, $p_4 = 0.12$, $p_5 = 0.09$, $p_6 = 0.07$. Будем складывать две наименьшие вероятности и включать суммарную вероятность на соответствующее место в упорядоченном списке вероятностей до тех пор, пока в списке не останется два символа. Тогда закодируем эти два символа 0 и

1. Далее кодовые слова достраиваются рекурсивно (как показано в типичном дереве Хаффмена).

Посчитаем среднюю длину построенного кода Хаффмена:

$$L_{\text{ср}} = 1 \cdot 0.36 + 3 \cdot 0.18 + 3 \cdot 0.18 + 3 \cdot 0.12 + 4 \cdot 0.09 + 4 \cdot 0.07 = 2.44,$$

при этом энтропия данного источника равна

$$H = -(0.36\log_2 0.36 + 2 \cdot 0.18\log_2 0.18 + 0.12\log_2 0.12 + 0.09\log_2 0.09 + 0.07\log_2 0.07) \approx 2.37.$$

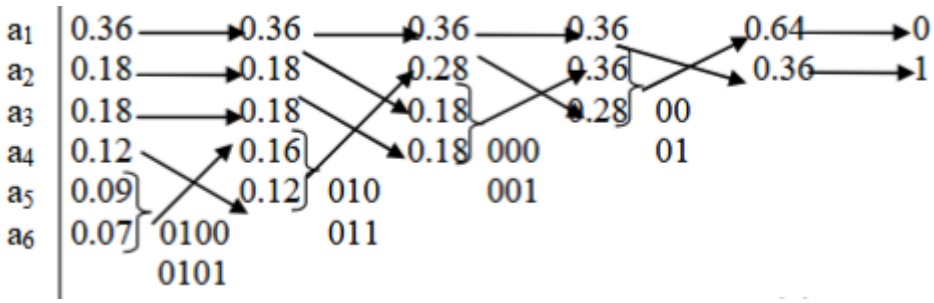

Рисунок 1. Процесс построения кода Хаффмена

a_i	P_i	L_i	кодовое слово
a_1	0.36	2	1
a_2	0.18	3	000
a_3	0.18	3	001
a_4	0.12	4	011
a_5	0.09	4	0100
a_6	0.07	4	0101

Таблица 1. Код Хаффмена

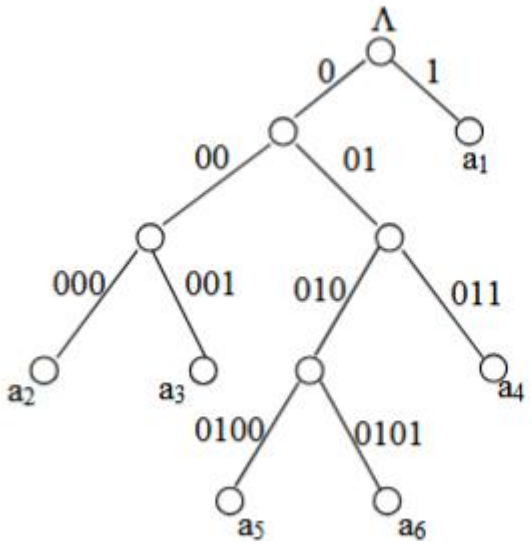

Рисунок 2. Кодовое дерево для кода Хаффмана

Код Хаффмена обычно строится и хранится в виде двоичного дерева, в листьях которого находятся символы алфавита, а на ветвях — 0 или 1. Тогда уникальным кодом символа является путь от корня дерева к этому символу, по которому все 0 и 1 собираются в одну уникальную последовательность.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

Загрузка и хранение данных:

Данные из файла считываются в один большой блок памяти, выделенный функцией malloc. Для реализации косвенной сортировки создается **индексный массив** index_arr, содержащий указатели на записи в основном массиве database. Такой подход позволяет избежать медленных операций копирования целых структур (64 байта) при обменах в процессе сортировки, заменяя их быстрыми обменами указателей (4 или 8 байт).

Реализация сортировки:

Функция сравнения compare_records реализует сложный ключ. Для сравнения дат из формата ДД-ММ-ГГ извлекаются компоненты. Год преобразуется в 4-значный формат по правилу: значения от 00 до 24 интерпретируются как 2000-е годы, а от 25 до 99 — как 1900-е. При равенстве дат сравнение продолжается по полю улицы с помощью стандартной функции strcmp.

Реализация поиска и очереди:

После выполнения двоичного поиска и нахождения индекса первой подходящей записи, запускается цикл, который движется вправо по index_arr, пока год в записи совпадает с ключом поиска. Указатели на найденные записи добавляются в очередь. Очередь реализована как односвязный список (struct QNode) с указателями на начало (q_head) и конец (q_tail), что обеспечивает добавление элементов за константное время

$$O(1)$$

.

Построение и поиск в дереве:

Перед вызовом рекурсивной функции build_tree_A2 все элементы из очереди переносятся во временный массив, который затем сортируется по номеру квартиры. Это является необходимым условием для корректной работы алгоритма А2.

Функция поиска в дереве search_tree_recursive реализована таким образом, чтобы находить **все** записи с искомым ключом. Поскольку в дереве А2

дубликаты могут оказаться в разных поддеревьях, после нахождения одного совпадения поиск рекурсивно продолжается как в левом, так и в правом поддереве.

Реализация кодирования:

Реализация кодирования:

Для реализации алгоритма Хаффмена база данных интерпретируется не как набор структур, а как сплошной поток байтов. Это достигается путем приведения указателя на массив записей к типу unsigned char*.

Процесс кодирования разделен на несколько этапов:

1. **Частотный анализ:** Сначала выполняется линейный проход по всей области памяти, занимаемой базой данных. В массиве счетчиков размером 256 элементов (по числу возможных значений байта) накапливается количество вхождений каждого символа. На основе полученных частот вычисляются вероятности появления каждого символа.
2. **Подготовка структур:** Данные о символах (значение байта, вероятность, будущее кодовое слово) упаковываются в массив вспомогательных структур SymbolInfo. Массив сортируется по убыванию вероятностей с использованием стандартной функции быстрой сортировки qsort.
3. **Построение кодов:** Используется рекурсивный подход. Задача кодирования n символов сводится к задаче для $n-1$ символов путем объединения двух элементов с наименьшими вероятностями в один фиктивный элемент с суммарной вероятностью.

- Функция находит правильную позицию для вставки суммарной вероятности в отсортированный массив, сохраняя порядок убывания.
- После рекурсивного вызова, на этапе «раскручивания» рекурсии (возврата), происходит формирование кодов: текущие кодовые слова сдвигаются, а последним двум символам (которые были объединены) присваиваются суффиксы '0' и '1'.

4. **Анализ эффективности:** После генерации кодов для всех символов программа вычисляет энтропию источника по формуле Шеннона и сравнивает её со средневзвешенной длиной полученных кодовых слов, что позволяет математически подтвердить оптимальность построенного кода.

Для корректного вывода кириллических символов в консоли Windows используется функция WinAPI SetConsoleCP(866).

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. Основные переменные и структуры

- const int DB_SIZE: Константа, определяющая количество записей в базе (4000).
- struct Record: Описывает структуру одной записи в базе данных.
- Record *database: Глобальный указатель на динамический массив, где хранятся все записи.

- Record **index_arr: Глобальный указатель на массив указателей, используемый для сортировки.
- struct QNode: Структура узла односвязного списка, реализующего очередь.
- QueueNode *q_head, *q_tail: Указатели на начало и конец очереди.
- struct TNode: Структура узла бинарного дерева поиска.
- TreeNode *tree_root: Указатель на корень построенного дерева.
- struct SymbolInfo: Вспомогательная структура для кодирования Хаффмена, хранит символ, его вероятность и сгенерированный код.

4.2. Описание подпрограмм

Управление данными:

- read_database(): Чтение бинарного файла в память.
- view_database(): Постраничный вывод содержимого базы данных на экран.
- clear_queue(), enqueue(), queue_length(): Функции для управления очередью.
- free_tree(): Рекурсивное освобождение памяти, занятой деревом.

Сортировка:

- heap_sort(): Основная функция, управляющая процессом пирамидальной сортировки.
- sift(): Реализация операции просеивания элемента в пирамиде.

- compare_records(): Функция сравнения двух записей по составному ключу.

Поиск:

- binary_search_and_queue(): Выполняет двоичный поиск и формирует очередь.

Дерево:

- build_tree_A2(): Рекурсивная функция построения дерева по алгоритму A2.
- tree_logic(): Оркестрирует процесс создания, вывода и поиска в дереве.
- search_tree_recursive(): Рекурсивный поиск всех вхождений ключа в дереве.

Кодирование:

- huffman_coding(): Главная функция, управляющая всем процессом кодирования.
- Huffman(), Up(), Down(): Функции, реализующие алгоритм Хаффмена согласно псевдокоду.
- compare_symbols(): Функция сравнения для qsort, упорядочивает символы по убыванию вероятности.

5. ТЕКСТ ПРОГРАММЫ

```
#define _CRT_SECURE_NO_WARNINGS
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <windows.h>
#include <math.h>

#define DB_SIZE 4000
#define PAGE_SIZE 20


typedef struct {
	char fio[32];
	char street[18];
	short int house;
	short int flat;
	char date[10];
} Record;
```

```
typedef struct QNode {
        Record* data;
        struct QNode* next;
} QueueNode;

typedef struct TNode {
        Record* data;
        struct TNode *left;
        struct TNode *right;
} TreeNode;

// For Huffman
typedef struct {
        unsigned char symbol;
        double probability;
        char code[256];
        int length;
} SymbolInfo;
```

```
Record *database;

Record **index_arr;


QueueNode *q_head = NULL;

QueueNode *q_tail = NULL;


TreeNode *tree_root = NULL;


int get_day(char* date) { return (date[0] - '0') * 10 + (date[1] - '0'); }

int get_month(char* date) { return (date[3] - '0') * 10 + (date[4] - '0'); }


int get_full_year(char* date) {

    int y = (date[6] - '0') * 10 + (date[7] - '0');

    if (y < 25) return 2000 + y;

    else return 1900 + y;

}


int get_short_year(char* date) {
```

```
    return (date[6] - '0') * 10 + (date[7] - '0');
}

void read_database() {
    FILE *fp = fopen("testBase4.dat", "rb");
    if (!fp) {
        printf("Error: File testBase4.dat not found!¥n");
        exit(1);
    }
    database = (Record*)malloc(DB_SIZE * sizeof(Record));
    fread(database, sizeof(Record), DB_SIZE, fp);
    fclose(fp);

    index_arr = (Record**)malloc(DB_SIZE * sizeof(Record*));
    for (int i = 0; i < DB_SIZE; i++) index_arr[i] = &database[i];
}

void print_header() {
    printf("¥n%-4s %-32s %-18s %-4s %-4s %-10s¥n", "No", "FIO", "Street",
"Dom", "Kv", "Date");
```

```
        printf("--------------------------------------------------------------------------------
¥n");
}

void print_record(Record *rec, int i) {
        // i is passed as 0-based index, so we print i+1
        printf("%-4d %-32.32s %-18.18s %-4d %-4d %-10.10s¥n",
                i + 1, rec->fio, rec->street, rec->house, rec->flat, rec->date);
}

int view_database(const char* title, int allow_sort_command) {
        int i = 0;
        while (i < DB_SIZE) {
                system("cls");
                printf("=== %s ===¥n", title);
                print_header();
                int limit = i + PAGE_SIZE;
                if (limit > DB_SIZE) limit = DB_SIZE;
                for (int j = i; j < limit; j++) print_record(index_arr[j], j);
                i = limit;
```

```
            printf("¥nCommands: [Enter] Next Page, [Esc] Main Menu");
            if (allow_sort_command) printf(", [S] SORT NOW");
            printf(": ");

            int ch = getch();
            if (ch == 27) return 0;
            if (allow_sort_command && (ch == 's' || ch == 'S')) return 1;
        }
        printf("¥nEnd of list. Press any key...");
        getch();
        return 0;
}

int compare_records(Record *a, Record *b) {
        int ya = get_full_year(a->date);
        int yb = get_full_year(b->date);
        if (ya != yb) return ya - yb;
```

```
    int ma = get_month(a->date);
    int mb = get_month(b->date);
    if (ma != mb) return ma - mb;

    int da = get_day(a->date);
    int db = get_day(b->date);
    if (da != db) return da - db;

    return strcmp(a->street, b->street);
}

void sift(Record **arr, int L, int R) {
    int i = L;
    int j = 2 * i + 1;
    Record *x = arr[L];
    while (j <= R) {
        if (j < R && compare_records(arr[j + 1], arr[j]) > 0) j++;
        if (compare_records(x, arr[j]) >= 0) break;
        arr[i] = arr[j];
```

```
            i = j;
            j = 2 * i + 1;
        }
        arr[i] = x;
}

void heap_sort() {
        int L = DB_SIZE / 2 - 1;
        while (L >= 0) { sift(index_arr, L, DB_SIZE - 1); L--; }
        int R = DB_SIZE - 1;
        while (R > 0) {
            Record *temp = index_arr[0];
            index_arr[0] = index_arr[R];
            index_arr[R] = temp;
            R--;
            sift(index_arr, 0, R);
        }
}
```

```
void clear_queue() {
    QueueNode *current = q_head;
    while (current != NULL) {
        QueueNode *temp = current;
        current = current->next;
        free(temp);
    }
    q_head = NULL;
    q_tail = NULL;
}

void enqueue(Record* rec) {
    QueueNode *new_node = (QueueNode*)malloc(sizeof(QueueNode));
    new_node->data = rec;
    new_node->next = NULL;
    if (q_tail == NULL) { q_head = new_node; q_tail = new_node; }
    else { q_tail->next = new_node; q_tail = new_node; }
}
```

```
int queue_length() {
    int count = 0;
    QueueNode *curr = q_head;
    while(curr) { count++; curr = curr->next; }
    return count;
}

void binary_search_and_queue() {
    int key;
    system("cls");
    printf("=== STEP: BINARY SEARCH ===¥n");
    printf("Enter Year (2 digits, e.g. 96): ");
    if (scanf("%d", &key) != 1) {
        while(getchar()!='¥n'); return;
    }
    while(getchar()!='¥n');

    int L = 0, R = DB_SIZE - 1;
    while (L < R) {
```

```
        int m = (L + R) / 2;
        int current = get_short_year(index_arr[m]->date);
        if (current < key) L = m + 1;
        else R = m;
    }

    if (get_short_year(index_arr[R]->date) == key) {
        printf("Records found. Building Queue...¥n");
        clear_queue();
        int i = R;
        while (i < DB_SIZE && get_short_year(index_arr[i]->date) == key) {
            enqueue(index_arr[i]);
            i++;
        }

        printf("¥n--- Queue Contents ---¥n");
        QueueNode *curr = q_head;
        int idx = 1;
        print_header();
```

```
            while(curr) {
                print_record(curr->data, idx++ - 1);
                curr = curr->next;
            }
        } else {
            printf("No records found for year %02d.¥n", key);
            clear_queue();
        }
        printf("¥nPress any key to continue...");
        getch();
}

void free_tree(TreeNode* node) {
        if (node == NULL) return;
        free_tree(node->left);
        free_tree(node->right);
        free(node);
}
```

```
void sort_temp_array(Record** arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j]->flat > arr[j+1]->flat) {
                Record* temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

TreeNode* build_tree_A2(Record** arr, int L, int R) {
    if (L > R) return NULL;
    double wes = 0;
    for (int k = L; k <= R; k++) wes += 1.0;

    double summa = 0;
    int i;
```

```
    for (i = L; i < R; i++) {
        if (summa < wes / 2.0 && (summa + 1.0) >= wes / 2.0) break;
        summa += 1.0;
    }

    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    node->data = arr[i];
    node->left = build_tree_A2(arr, L, i - 1);
    node->right = build_tree_A2(arr, i + 1, R);
    return node;
}

void print_tree_recursive(TreeNode* node, int *counter) {
    if (node == NULL) return;
    print_tree_recursive(node->left, counter);

    print_record(node->data, (*counter));
    (*counter)++;
```

```
    print_tree_recursive(node->right, counter);
}

void search_tree_recursive(TreeNode* node, int key, int *found_count) {
    if (node == NULL) return;

    if (key < node->data->flat) {
        search_tree_recursive(node->left, key, found_count);
    }
    else if (key > node->data->flat) {
        search_tree_recursive(node->right, key, found_count);
    }
    else {
        print_record(node->data, (*found_count));
        (*found_count)++;

        // Continue searching both subtrees for potential duplicates
        search_tree_recursive(node->left, key, found_count);
        search_tree_recursive(node->right, key, found_count);
```

```
    }
}

void tree_logic() {
    if (q_head == NULL) {
        printf("¥nQueue is empty. Cannot build tree.¥nPress any key...");
        getch();
        return;
    }

    if (tree_root) { free_tree(tree_root); tree_root = NULL; }

    int n = queue_length();
    Record** temp_arr = (Record**)malloc(n * sizeof(Record*));
    QueueNode* curr = q_head;
    for(int k=0; k<n; k++) { temp_arr[k] = curr->data; curr = curr->next; }

    sort_temp_array(temp_arr, n);
    tree_root = build_tree_A2(temp_arr, 0, n - 1);
```

```
free(temp_arr);


system("cls");
printf("=== STEP: OPTIMAL TREE (A2) ===¥n");
printf("Tree built by Flat Number.¥n¥n");
print_header();


int counter = 0;
print_tree_recursive(tree_root, &counter);


// Search in Tree
int key_flat;
printf("¥n>>> Tree Search <<<¥n");
printf("Enter Flat Number to search: ");
if (scanf("%d", &key_flat) == 1) {
    printf("Searching...¥n");
    print_header();


    int found_count = 0;
```

```
            search_tree_recursive(tree_root, key_flat, &found_count);


            if (found_count == 0) printf("Flat %d not found in the tree.¥n",
key_flat);
            else printf("¥nTotal found: %d¥n", found_count);
        }
        while(getchar()!='¥n');
        printf("¥nPress any key to continue...");
        getch();
}


char C_matrix[256][256];
int L_arr[256];


void Up(int n, double q, double *P, int *j_out) {
        int i;
        int j = 0;
        for (i = n - 2; i >= 1; i--) {
            if (P[i-1] <= q) {
                P[i] = P[i-1];
```

```
            } else {
                j = i;
                goto found;
            }
        }
        j = i;
found:
        P[j] = q;
        *j_out = j;
}

void Down(int n, int j) {
        char S[256];
        strcpy(S, C_matrix[j]);

        for (int i = j; i < n - 1; i++) {
            strcpy(C_matrix[i], C_matrix[i+1]);
            L_arr[i] = L_arr[i+1];
        }
```

```
        strcpy(C_matrix[n-2], S);
        strcpy(C_matrix[n-1], S);

        strcat(C_matrix[n-2], "0");
        strcat(C_matrix[n-1], "1");

        L_arr[n-2] = strlen(C_matrix[n-2]);
        L_arr[n-1] = strlen(C_matrix[n-1]);
}

void Huffman(int n, double *P) {
        if (n == 2) {
                strcpy(C_matrix[0], "0"); L_arr[0] = 1;
                strcpy(C_matrix[1], "1"); L_arr[1] = 1;
        } else {
                double q = P[n-2] + P[n-1];
                int j;
                Up(n, q, P, &j);
```

```
            Huffman(n - 1, P);
            Down(n, j);
        }
}

int compare_symbols(const void *a, const void *b) {
        SymbolInfo *sa = (SymbolInfo*)a;
        SymbolInfo *sb = (SymbolInfo*)b;
        if (sb->probability > sa->probability) return 1;
        if (sb->probability < sa->probability) return -1;
        return 0;
}

void huffman_coding() {
        system("cls");
        printf("=== STEP: HUFFMAN CODING ===¥n");
        printf("Calculating probabilities...¥n");

        long counts[256] = {0};
```

```
long total_bytes = 0;


unsigned char *ptr = (unsigned char*)database;
size_t total_size = DB_SIZE * sizeof(Record);
for (size_t i = 0; i < total_size; i++) {
    counts[ptr[i]]++;
    total_bytes++;
}


SymbolInfo symbols[256];
int n = 0;
for (int i = 0; i < 256; i++) {
    if (counts[i] > 0) {
        symbols[n].symbol = (unsigned char)i;
        symbols[n].probability = (double)counts[i] / total_bytes;
        n++;
    }
}
```

```
    qsort(symbols, n, sizeof(SymbolInfo), compare_symbols);

    double *P = (double*)malloc(n * sizeof(double));
    for(int i=0; i<n; i++) P[i] = symbols[i].probability;

    Huffman(n, P);

    printf("¥n%-10s %-12s %-20s %-10s¥n", "Symbol", "Prob(Pi)", "Code",
"Length(Li)");
    printf("----------------------------------------------------¥n");

    double avg_len = 0;
    double entropy = 0;

    for (int i = 0; i < n; i++) {
        strcpy(symbols[i].code, C_matrix[i]);
        symbols[i].length = L_arr[i];

        char display_char = symbols[i].symbol;
        if (display_char < 32) display_char = '.';
```

```
        if (i < 15 || i > n - 5) {
            printf("'%c' (0x%02X)    %.6f          %-20s %d¥n",
                display_char, symbols[i].symbol, symbols[i].probability,
                symbols[i].code, symbols[i].length);
        } else if (i == 15) {
            printf("... (skipping middle rows) ...¥n");
        }

        avg_len += symbols[i].probability * symbols[i].length;
        if (symbols[i].probability > 0) {
            entropy -= symbols[i].probability * log2(symbols[i].probability);
        }
    }

    printf("-------------------------------------------------------¥n");
    printf("Average Code Length (L_avg): %.4f¥n", avg_len);
    printf("Entropy (H):                 %.4f¥n", entropy);
    printf("Comparison: L_avg %s H + 1¥n", (avg_len < entropy + 1) ? "<" :
">=");
```

```
    free(P);
    printf("¥nPress any key to finish...");
    getch();
}

int main() {
    SetConsoleCP(866);
    SetConsoleOutputCP(866);

    read_database();

    int sort_requested = view_database("Unsorted Database (Load)", 1);

    if (sort_requested) {
        printf("¥nSorting (Heap Sort)... Please wait.¥n");
        heap_sort();
        printf("Sorted! Press any key to view.¥n");
        getch();
```

```
        view_database("Sorted Database (Date+Street)", 0);

        binary_search_and_queue();

        tree_logic();

        huffman_coding();
    } else {
        printf("¥nExited without sorting.¥n");
    }

    if (database) free(database);
    if (index_arr) free(index_arr);
    clear_queue();
    free_tree(tree_root);

    return 0;
}
```

6. РЕЗУЛЬТАТЫ

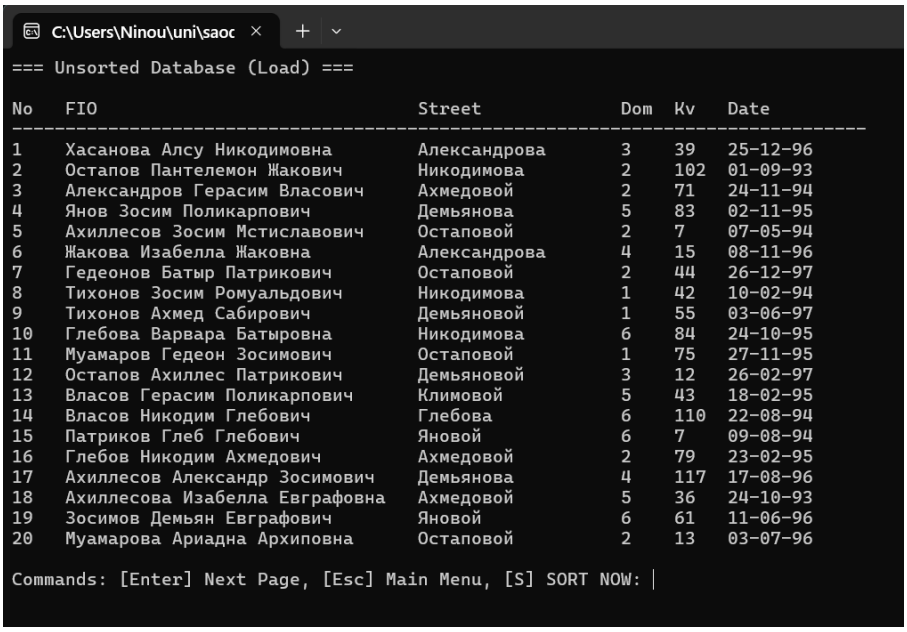

Рисунок 1. Неотсортированная база данных

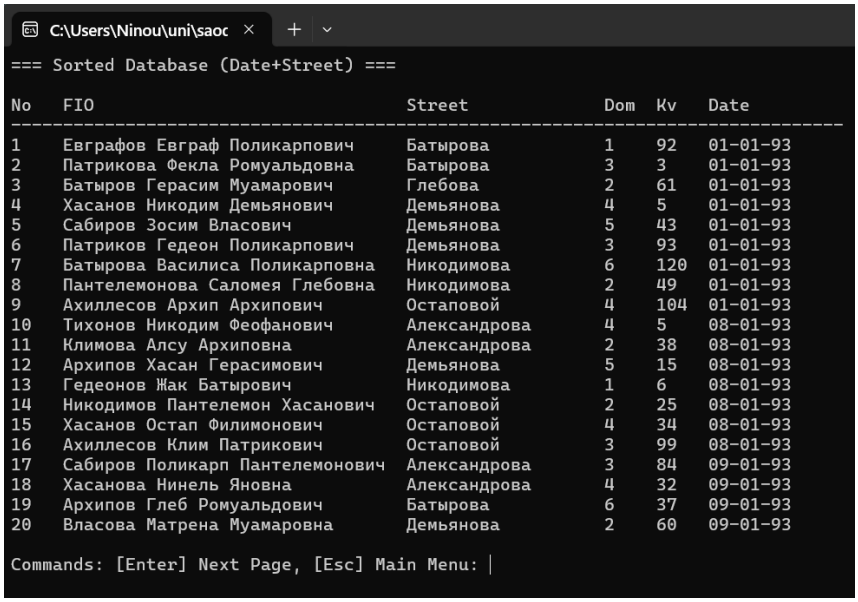

Рисунок 2. Отсортированная по дате поселения и названии улицы база данных.

Рисунок 3. Очередь из записей, полученных в результате поиска (Год 96)

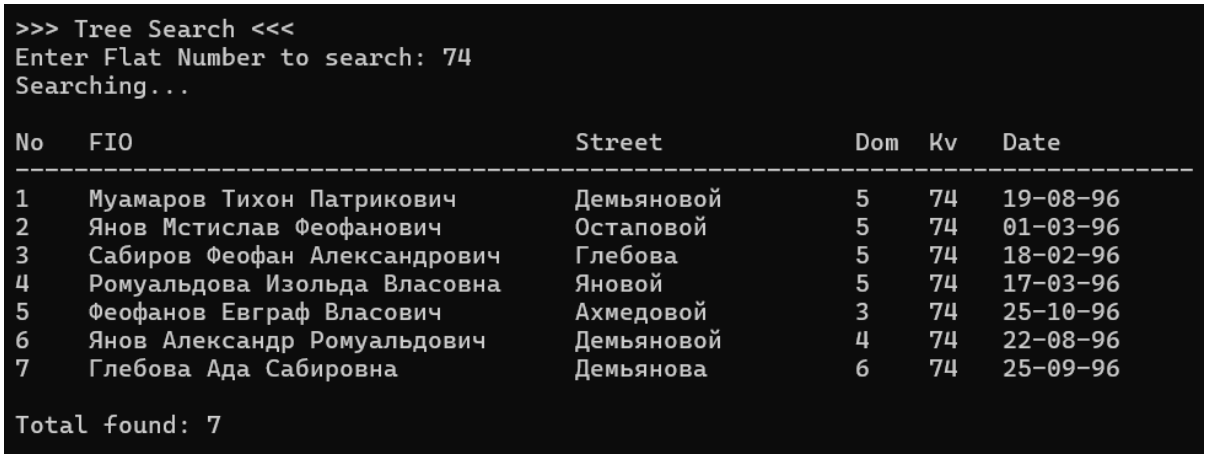

Рисунок 4. Построено дерево, поиск в дереве по номеру квартиры (74 квартира)

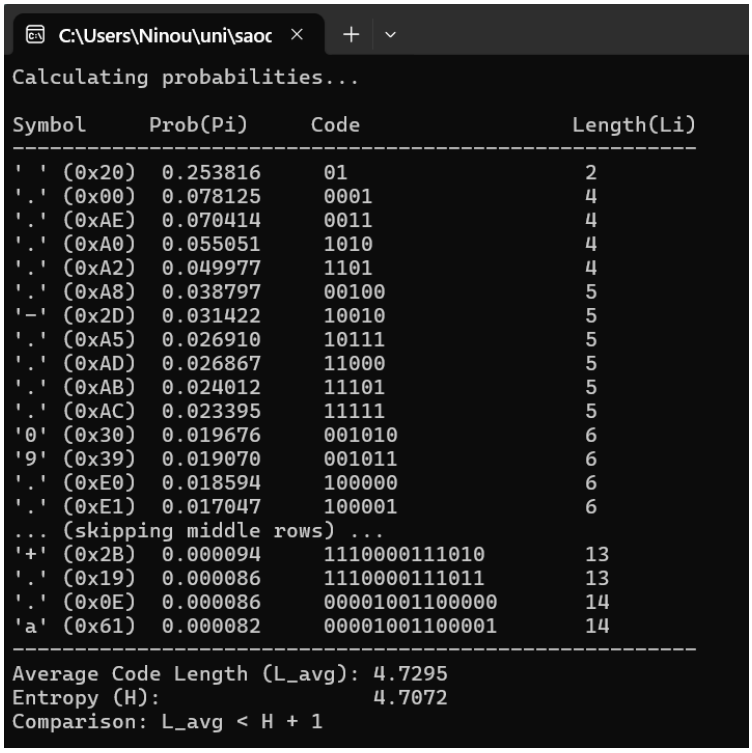

Рисунок 5. Результаты кодирования базы данных (начальный и конечный фрагменты), средняя длина кодового слова и энтропия источника:

7. ВЫВОДЫ

В результате выполнения курсового проекта была успешно разработана и протестирована программа, реализующая все поставленные задачи по обработке базы данных «Населенный пункт».

Основные результаты:

1. Освоены и применены на практике методы работы с динамической памятью и файловыми потоками на языке С.

2. Реализована эффективная сортировка больших объемов данных с использованием метода Вилльямса-Флойда $O(n \log n)$ и техники

косвенной адресации через массив указателей, что позволило минимизировать накладные расходы на перемещение данных.

3. Продемонстрирована работа алгоритма двоичного поиска для быстрой выборки данных из упорядоченного массива.
4. Применены динамические структуры данных (односвязный список и двоичное дерево) для решения задач промежуточного хранения и организации данных для быстрого доступа.
5. Изучен и реализован алгоритм построения сбалансированного дерева (А2), обеспечивающий логарифмическую сложность поиска по вторичному ключу.
6. Выполнено статическое кодирование данных методом Хаффмена. Расчеты энтропии и средней длины кодового слова подтвердили теоретическую эффективность алгоритма как метода оптимального префиксного кодирования.