Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №7 ПО ДИСЦИПЛИНЕ: ТИПЫ И СТРУКТУРЫ ДАННЫХ

Деревья, хеш-таблицы.

Студент **Зинин Артём Сергеевич**

Группа **ИУ7-36Б**

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

Студент _____ **Зинин А.С.**

Преподаватель _____ **Никульшина Т.А.**

2024

Описание условия задачи

Задание: Построить хеш-таблицу и AVL-дерево по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице (используя открытую и закрытую адресацию). Вывести на экран деревья и хеш-таблицы. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если количество сравнений при поиске/добавлении больше указанного. Оценить эффективность использования этих структур (по времени и по памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий и при различных методах их разрешения.

Построить дерево поиска из слов текстового файла (задача No6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Удалить все слова, начинающиеся на указанную букву, в исходном и сбалансированном дереве. Построить хеш-таблицу из слов текстового файла. Вывести построенную таблицу слов на экран. Осуществить поиск и удаление введенного слова. Выполнить программу для различных размерностей таблицы и сравнить время удаления, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц.

Описание ТЗ

Исходные данные:

- Пункт меню (Меню обозначено в пункте “Задача, реализуемая программой”)
- Значения деревьев
- Файлы

Выходные данные:

1. Графическое представления дерева, отображение хэш-таблиц
2. Текстовый обход дерева
3. Адреса элементов при их поиске.
4. Вывод результатов сравнения эффективности программы при различных замерах

Задача, реализуемая программой:

1. Добавить слово в дерево
2. Удалить слово из дерева
3. Найти слово в дереве
4. Совершить обход дерева
5. Прочесть слова из файла
6. Отобразить дерево графически
7. Сравнить время удаления, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц.

0. Выход из программы

Способ обращения к программе:

Программа запускается следующим образом: ./app.exe

Возможные аварийные ситуации и ошибки пользователя:

- Неверный тип вводимых данных
- Пустой ввод
- Попытка удалить отсутствующий элемент

Описание внутренних СД

```
typedef struct tree
{
    char *value;
    struct tree *left;
    struct tree *right;
} tree_t;

typedef struct avl_tree
{
    unsigned char height;
    char *value;
    struct avl_tree *left;
    struct avl_tree *right;
} avl_tree_t;

typedef struct hashset_close
{
    char **value;
    int size;
} hashset_close_t;

typedef struct hashset_open_node
{
    char *value;
    struct hashset_open_node *next;
} hashset_open_node_t;

typedef struct hashset_open
{
    hashset_open_node_t **table;
    int size;
} hashset_open_t;
```

```
// Из файла menu.h
void print_menu(); // Выводит меню пользователя
int get_type(int); // Получает тип выбранного действия от
пользователя

// Из файла tree.h
tree_t *allocate_tree(); // Выделяет память для нового
узла дерева
tree_t *find_tree(tree_t*, tree_t*, int (*)(void*,
void*)); // Находит узел в дереве
tree_t *find_tree_with_parent(tree_t*, tree_t*, tree_t
**, int (*)(void*, void*)); // Находит узел в дереве с
родителем
void add_node(tree_t**, tree_t*, int (*)(void*,
void*)); // Добавляет узел в дерево
int remove_node(tree_t**, tree_t*, int (*)(void*,
void*)); // Удаляет узел из дерева
void replace_ressult_tree(tree_t*, tree_t*, tree_t*); //
Заменяет результат дерева
void print_tree(tree_t*); // Выводит дерево
size_t read_file(tree_t**, FILE*); // Читает файл и
создает дерево
void free_tree(tree_t*); // Освобождает память дерева
int comparator_tree(void*, void*); // Сравнивает узлы
дерева
void dont_print_sort_tree(tree_t*); // Применяет
сортировку к дереву без вывода

// Из файла avl_tree.h
avl_tree_t *allocate_avl_tree(); // Выделяет память для
узла AVL-дерева
avl_tree_t *find_avl_tree(avl_tree_t*, avl_tree_t*, int
(*)(void*, void*)); // Находит узел в AVL-дереве
avl_tree_t *add_avl_node(avl_tree_t*, avl_tree_t*, int
(*)(void*, void*)); // Добавляет узел в AVL-дерево
avl_tree_t *find_min_avl_node(avl_tree_t*); // Находит
минимальный узел в AVL-дереве
avl_tree_t *remove_min_avl_node(avl_tree_t*); // Удаляет
минимальный узел из AVL-дерева
avl_tree_t *remove_avl_node(avl_tree_t*, avl_tree_t*, int
(*)(void*, void*)); // Удаляет узел из AVL-дерева
void free_avl_node(avl_tree_t*); // Освобождает память
узла AVL-дерева
void print_avl_tree(avl_tree_t*); // Выводит AVL-дерево
size_t read_file_avl_tree(avl_tree_t**, FILE*); // Читает
AVL-дерево из файла
void free_avl_tree(avl_tree_t*); // Освобождает память
AVL-дерева
```

```
int comparator_avl_tree(void*, void*); // Сравнивает узлы
AVL-дерева
void dont_print_sort_avl_tree(avl_tree_t*); // Сортирует
AVL-дерево без вывода
int height(avl_tree_t*); // Возвращает высоту AVL-дерева
int calc_balance(avl_tree_t*); // Вычисляет баланс AVL-
дерева
avl_tree_t *balance_avl_tree(avl_tree_t*); // Балансирует
AVL-дерево
avl_tree_t *rotate_right(avl_tree_t*); // Поворачивает
AVL-дерево вправо
avl_tree_t *rotate_left(avl_tree_t*); // Поворачивает
AVL-дерево влево
void fix_height(avl_tree_t*); // Фиксирует высоту AVL-
дерева

// Из файла compare_time.h
unsigned long long calc_elapsed_time(const struct
timespec*, const struct timespec*); // Вычисляет разницу
во времени
void comparison_code(char*); // Сравнивает
производительность структур данных
int find_avl_tree_count(avl_tree_t*, avl_tree_t*, int(*)
(void*, void*)); // Считает количество узлов в AVL-дереве

// Из файла functions.h
void menu_add(tree_t**, avl_tree_t**, hashset_close_t**,
hashset_open_t**); // Добавляет элемент в структуры
данных
void menu_remove(tree_t**, avl_tree_t**,
hashset_close_t*, hashset_open_t*); // Удаляет элемент из
структур данных
void menu_find(tree_t*, avl_tree_t*, hashset_close_t*,
hashset_open_t*); // Ищет элемент в структурах данных
void menu_print(tree_t*, avl_tree_t*, hashset_close_t*,
hashset_open_t*); // Выводит содержимое структур данных
void menu_show_tree(tree_t*, avl_tree_t*); // Показывает
деревья графически
void menu_read(tree_t**, avl_tree_t**, hashset_close_t**,
hashset_open_t**); // Читает данные из файла
void menu_efficiency_comparison_code(); // Сравнивает
эффективность структур данных
void signal_remove_tmp_files(int); // Удаляет временные
файлы при сигнале
int find_word_starts_with_symbol_file(FILE*, char); //
Считает слова, начинающиеся с символа в файле
int find_word_starts_with_symbol_tree(tree_t*, char); //
Считает слова, начинающиеся с символа в дереве
```

```
void print_remove_result(int); // Выводит результат
удаления
void print_razd(int); // Выводит разделитель

// Из файла tree_gui.h
int show_tree(void*, char, int, void (*)(FILE*, void*),
int (*)(FILE*, void*, char, char*)); // Показывает дерево
графически
void print_tree_structure(FILE*, void*); // Выводит
структуру дерева в файл
int print_tree_color(FILE*, void*, char, char*); //
Выводит цветное представление дерева
void print_avl_tree_structure(FILE*, void*); // Выводит
структуру AVL-дерева в файл
int print_avl_tree_color(FILE*, void*, char, char*); //
Выводит цветное представление AVL-дерева
char *create_tmpfilename(); // Создает имя временного
файла
int initialize_tmpfiles(char**, char**, char**); //
Инициализирует временные файлы
void remove_tmp_files(); // Удаляет временные файлы

// Из файла hashset_close.h
hashset_close_t *create_hashset(int); // Создает хеш-
таблицу с закрытой адресацией
int add_str_hashset(hashset_close_t **, char *); //
Добавляет строку в хеш-таблицу с закрытой адресацией
void add_strdup_hashset(hashset_close_t **, char *); //
Добавляет копию строки в хеш-таблицу с закрытой
адресацией
void restruct_hashset(hashset_close_t **); //
Перестраивает хеш-таблицу с закрытой адресацией
void del_str_hashset(hashset_close_t *, char *); //
Удаляет строку из хеш-таблицы с закрытой адресацией
void find_str_hashset(hashset_close_t *, char *); // Ищет
строку в хеш-таблице с закрытой адресацией
int find_str_hashset_count(hashset_close_t*, char*); //
Считает количество итераций при поиске в хеш-таблице с
закрытой адресацией
void del_hashset(hashset_close_t **); // Удаляет хеш-
таблицу с закрытой адресацией
void read_file_hashset(hashset_close_t**, FILE*); //
Читает хеш-таблицу с закрытой адресацией из файла
void print_node_close(hashset_close_t*, int, int); //
Выводит узел хеш-таблицы с закрытой адресацией
int count_max_str_len(hashset_close_t*); // Считает
максимальную длину строки в хеш-таблице с закрытой
адресацией
```

```
// Из файла hashset_open.h
hashset_open_t *create_hashset_open(int); // Создает хеш-таблицу с открытой адресацией
int add_str_hashset_open(hashset_open_t**, char*); // Добавляет строку в хеш-таблицу с открытой адресацией
void add_strdup_hashset_open(hashset_open_t**, char*); // Добавляет копию строки в хеш-таблицу с открытой адресацией
void restruct_hashset_open(hashset_open_t**); // Перестраивает хеш-таблицу с открытой адресацией
void restruct_node_open(hashset_open_t**,
hashset_open_node_t*); // Перестраивает узел хеш-таблицы с открытой адресацией
void del_str_hashset_open(hashset_open_t*, char*); // Удаляет строку из хеш-таблицы с открытой адресацией
void find_str_hashset_open(hashset_open_t*, char*); // Ищет строку в хеш-таблице с открытой адресацией
int find_str_hashset_open_count(hashset_open_t*,
char*); // Считает количество итераций при поиске в хеш-таблице с открытой адресацией
void del_hashset_open(hashset_open_t**); // Удаляет хеш-таблицу с открытой адресацией
void free_node_open(hashset_open_t*,
hashset_open_node_t*); // Освобождает память узла хеш-таблицы с открытой адресацией
void read_file_hashset_open(hashset_open_t**, FILE*); // Читает хеш-таблицу с открытой адресацией из файла
void print_node_open(hashset_open_t*, int); // Выводит узел хеш-таблицы с открытой адресацией
int count_node_len_open(hashset_open_t*, int); // Считает длину узла хеш-таблицы с открытой адресацией

// Из файла io.h
void get_numeric(char*, void*, int); // Получает числовое значение от пользователя
bool isNumeric(const char*, bool); // Проверяет, является ли строка числом
ssize_t getword(char**, size_t*, FILE*); // Читает слово из файла
size_t len_staff(char**, size_t*, size_t); // Увеличивает размер массива для чтения слова
bool is_prime(int); // Проверяет, является ли число простым
int prime_before(int); // Находит ближайшее простое число меньше заданного
```

Описание алгоритма

Хэш-функция общего случая выглядит следующим образом:

```
static int calc_hash(char *str, int key, int size)
{
    unsigned long long result = 0;
    while (*str)
        result = result * key + *(str++);
    return result % size;
}
```

1-я хэш функция (а также хэш функция для хэш таблицы с открытой адресацией) в качестве значения key получает число 2887
2-я хэш функция в качестве значения key получает число 2909
Size - размер хэ-таблицы

Хэш-таблица с закрытой адресацией:

1. Создание таблицы
2. Добавление элемента:
    1. Вычисление 1-го и 2-го хэша
    2. Если ячейка на месте , куда указывает 1-й хэш - занята, прибавление 2-го и так 6 раз, пока не будет найдено место.
    3. В случае, если за 6 проходов место не было найдено - увеличение размера хэш таблицы до ближайшего меньшего простого числа к (размер таблицы * 2)
3. Поиск элемента:
    1. Вычисление 1-го и 2-го хэша
    2. Если значение на месте , куда указывает 1-й хэш - иное, прибавление 2-го и так 6 раз, пока не будет найдено необходимое значение.
    3. В случае, если за 6 проходов значение не было найдено - прекращение поиска.
4. Удаление элемента:
    1. Вычисление 1-го и 2-го хэша
    2. Если значение на месте , куда указывает 1-й хэш - иное, прибавление 2-го и так 6 раз, пока не будет найдено необходимое значение.
        1. В случае, если за 6 проходов значение не было найдено - прекращение поиска.

2. В случае, если было найдено - замена значения на заранее подготовленное, обозначающее удалённый элемент.

Хэш-таблица с открытой адресацией:

1. Создание таблицы
2. Добавление элемента:
    1. Вычисление хэша
    2. Если ячейка на месте , куда указывает хэш - занята, переход к следующему элементу в односвязном списке, находящемся в этой ячейка и добавление нужного элемента следующим.
    3. В случае, если за 3 прохода место не было найдено - увеличение размера хэш таблицы до (размер таблицы * 2)
3. Поиск элемента:
    1. Вычисление хэша
    2. Если ячейка на месте , куда указывает хэш - иная, переход к следующему элементу в односвязном списке, находящемся в этой ячейка, пока не будет найдено необходимое значение.
    3. В случае, если за 3 прохода значение не было найдено - прекращение поиска.
4. Удаление элемента:
    1. Вычисление хэша
    2. Если ячейка на месте , куда указывает хэш - иная, переход к следующему элементу в односвязном списке, находящемся в этой ячейка, пока не будет найдено необходимое значение.
        1. В случае, если за 3 прохода значение не было найдено - прекращение поиска.
        2. В случае, если было найдено - удаление этого элемента и установление указателя для предыдущего элемента на следующий = следующий элемент после удаляемого.

Авл-дерево:

1. Создание дерева:
    1. Инициализация узла AVL-дерева.
    2. Если переданная строка не равна NULL, выделяется память для строки и копируется её значение.
    3. Поля left и right устанавливаются в NULL, а height задаётся равным 1.
2. Добавление элемента:
    1. Если корень дерева равен NULL, создаётся новый узел, который становится корнем.
    2. Сравнение нового узла с текущим:
        - Если узел меньше текущего, рекурсивно добавляется в левое поддерево.
        - Если больше, рекурсивно добавляется в правое поддерево.
        - Если равен текущему узлу, добавление отклоняется.
    3. После добавления узел балансируется:
        - Вычисляется баланс-фактор текущего узла (разница высот правого и левого поддеревьев).
        - Если баланс-фактор равен 2 или -2, выполняются повороты (левый или правый) для восстановления баланса.
3. Поиск элемента:
    1. Если корень дерева равен NULL, поиск прекращается.
    2. Сравнение искомого узла с текущим:
        - Если они равны, возвращается текущий узел.
        - Если меньше, поиск продолжается в левом поддереве.
        - Если больше, поиск продолжается в правом поддереве.
    3. Если узел не найден, возвращается NULL.
4. Удаление элемента:
    1. Если дерево пустое, операция прекращается.
    2. Сравнение удаляемого узла с текущим:
        - Если узел меньше текущего, удаление продолжается в левом поддереве.
        - Если больше, в правом.

- Если узел найден:
  - Если у него нет потомков, он просто удаляется.
  - Если есть только одно поддерево, заменяется этим поддеревом.
  - Если есть оба поддерева:
    - Находится минимальный узел в правом поддереве.
    - Удаляется минимальный узел из правого поддерева.
    - Заменяется текущий узел минимальным.

3. После удаления узел балансируется аналогично добавлению.

5. Балансировка узла:
   1. Вычисляется высота узла как 1 + max(высота_левого, высота_правого).
   2. Рассчитывается баланс-фактор баланс-фактор = высота_правого – высота_левого
   3. Если баланс-фактор равен 2:
      1. Если баланс-фактор правого дочернего узла < 0:
         - Выполняется **правый поворот** для правого дочернего узла.
      2. Выполняется **левый поворот** для текущего узла.
   4. Если баланс-фактор равен -2:
      1. Если баланс-фактор левого дочернего узла больше 0:
         - Выполняется **левый поворот** для левого дочернего узла.
      2. Выполняется **правый поворот** для текущего узла.
6. Повороты:
   1. Левый поворот:
      - Узел заменяется своим правым дочерним узлом.
      - Правый узел становится левым дочерним для нового корня.
      - Высота обновляется.
   2. Правый поворот:
      - Узел заменяется своим левым дочерним узлом.
      - Левый узел становится правым дочерним для нового корня.
      - Высота обновляется.

7. Освобождение памяти:
    1. Если узел имеет потомков, память освобождается рекурсивно для всех поддеревьев.
    2. Для каждого узла освобождаются значение (value) и сам узел.

Набор тестов

Позитивные тесты

Описание теста	Входные данные	Выходные данные
Проверка добавления слова	1 test	Слово успешно добавлено
Проверка удаления слова	1 test 2 test	Слово успешно удалено
Проверка поиска слова	1 test 3 test	Слово успешно найдено
Проверка обхода дерева	1 test 4	Обход дерева успешно сделан
Проверка чтения слов из файла	5 func_tests/data/test7.txt	Деревья и хэш-таблицы успешно построены
Проверка отображения деревьев графически	5 func_tests/data/test7.txt 6	Деревья, отображенны графически
Проверка отображения деревьев графически, выделяя слова на заданную букву цветом	5 func_tests/data/test7.txt 6 b	Деревья, отображенны графически, слова начинающиеся на букву b выделены цветом
Проверка корректного выхода из программы	0	Программа корректно завершает работу без ошибок

Негативное тесты

Описание теста	Входные данные	Выходные данные
Проверка удаления слова из пустого дерева	2 test	Невозможно удалить слово, дерево пусто
Проверка удаления несуществующего слова из дерева	1 test 2 badword	Слово не найдено
Проверка чтения слов из несуществующего файла в дерево	5 strange_path/bad_file.txt	Failed to open file: No such file or directory

Сравнение эффективности

Время удаления, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц.

filename	type	size, bytes	avg find count	avg find t, ns	avg rm count	avg dell t, ns
test20.txt	BIN_tree	480	4.050000	126.050000	-	-
test20.txt	AVL_tree	640	2.400000	115.950000	3.750000	274.150000
test20.txt	Close_table	360	1.550000	123.250000	1.550000	166.600000
test20.txt	Open_table	680	1.600000	95.100000	1.600000	155.150000
test50.txt	BIN_tree	1200	5.740000	143.800000	-	-
test50.txt	AVL_tree	1600	3.460000	129.580000	4.920000	308.040000
test50.txt	Close_table	1320	1.180000	100.840000	1.180000	137.500000
test50.txt	Open_table	2120	1.220000	71.520000	1.220000	128.040000
test100.txt	BIN_tree	2400	7.640000	180.950000	-	-
test100.txt	AVL_tree	3200	4.300000	151.330000	5.850000	337.290000
test100.txt	Close_table	2552	1.170000	101.770000	1.170000	163.910000
test100.txt	Open_table	4152	1.340000	74.930000	1.340000	126.640000
test1000.txt	BIN_tree	24000	11.848000	271.965000	-	-
test1000.txt	AVL_tree	32000	7.362000	233.727000	9.196000	493.604000
test1000.txt	Close_table	20040	1.310000	120.341000	1.310000	135.535000
test1000.txt	Open_table	36040	1.108000	92.257000	1.108000	131.082000
test10000.txt	BIN_tree	240000	16.249800	488.515200	-	-
test10000.txt	AVL_tree	320000	10.648500	371.198900	12.562700	749.752000
test10000.txt	Close_table	318968	1.152000	121.139300	1.152000	147.116200
test10000.txt	Open_table	478968	1.076000	114.129400	1.076000	175.826300

Выводы по проделанной работе

Бинарные деревья поиска удобно использовать если нужно всегда иметь доступ к отсортированным данным. Для поиска же деревья лучше использовать, либо в тех случаях, когда из исходных данных оно получится сбалансированным по высоте, либо использовать деревья специально для этого предназначенные (например красно-чёрное дерево)

Бинарные деревья поиска занимают больше места чем файлы, но использования их эффективнее чем, использование файла, в большинстве файлов.

Бинарные деревья поиска, по факту их существования являются отсортированными, поэтому вывод отсортированного списка будет иметь сложность O(n), n - кол-во элементов в дереве

Среднее время поиска в бинарных деревьях поиска зависит от ветвистости этого дерева.

Сбалансированные двоичные деревья - всегда, лучше чем обычное бинарное дерево поиска

Хэш-таблицы, практически всегда лучше.

Ответы на контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от АВЛ-дерева?

- Идеально сбалансированное дерево: дерево, в котором разница высот левого и правого поддеревьев любого узла равна 0, а количество узлов в левых и правых поддеревьях максимально равное. Такое дерево обеспечивает минимальную высоту, но добиться такого состояния после каждого изменения данных практически невозможно.
- АВЛ-дерево: двоичное дерево поиска, где разница высот левого и правого поддеревьев любого узла (баланс-фактор) составляет не более 1. Оно поддерживает балансировку с помощью вращений при добавлении или удалении узлов, но не всегда идеально сбалансировано.

2. Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?

- В АВЛ-дереве: гарантирована O(log n) сложность поиска благодаря поддержанию сбалансированности дерева. Поиск осуществляется через последовательное сравнение значения с узлами дерева.
- В обычном дереве двоичного поиска: в худшем случае (например, для вырожденного дерева) сложность может достигать O(n), если дерево несбалансировано, превращаясь в линейный список.

3. Что такое хеш-таблица, каков принцип ее построения?

Хеш-таблица — структура данных, которая использует хеш-функцию для преобразования ключа в индекс массива, где хранятся значения. Принципы:

1. Хеш-функция: принимает ключ и вычисляет его хеш (индекс в массиве).
2. Коллизии: если два ключа имеют одинаковый хеш, применяется метод разрешения (например, цепочки или открытая адресация).
3. Простота доступа: операции вставки, удаления и поиска выполняются в среднем за O(1).

4. Что такое коллизии? Каковы методы их устранения?

Коллизия — ситуация, когда разные ключи дают одинаковое значение хеш-функции. Методы устранения:

1. Метод цепочек: каждый индекс массива хранит список всех элементов, соответствующих данному хешу.
2. Открытая адресация: в случае коллизии осуществляется поиск следующей свободной ячейки (например, линейная, квадратичная адресация или двойное хеширование).
3. Перехеширование: изменение хеш-функции при переполнении.

5. В каком случае поиск в хеш-таблицах становится неэффективен?

- Слишком большое количество коллизий приводит к ухудшению эффективности. В случае цепочек — рост длины списка, в случае открытой адресации — увеличение числа проходов.
- Неэффективная хеш-функция, которая создает неравномерное распределение значений.
- Высокий коэффициент заполнения (>0.7) без увеличения размера таблицы.

6. Эффективность поиска в структурах данных:

Структура данных	Средняя сложность поиска	Худшая сложность поиска
АВЛ-дерево	$O(\log n)$	$O(\log n)$
Дерево двоичного поиска	$O(\log n)$	$O(n)$
Хеш-таблица	$O(1)$	$O(n)$
Файл (линейный поиск)	$O(n)$	$O(n)$
Файл (индексированный)	$O(\log n)$	$O(\log n)$

Примечания:

- Хеш-таблицы эффективны для быстрого доступа при хорошем распределении ключей.
- АВЛ-деревья полезны, если требуется упорядоченность данных.
- Файлы используются, если данные должны быть сохранены на диске.