Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное 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»

Двоичные деревья и хеш-таблицы

Вариант № 5

Студент: **Бондарева В. А.**

Группа: **ИУ7-34Б**

Преподаватель: **Барышникова М. Ю.**

2025 г.

Условие задачи

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.

Описание ТЗ

Исходные данные

Исходными данными программы являются:

1. Строка, содержащая единственное целое положительное число, являющееся пунктом меню.
   Пользователю предоставляются следующие опции меню:

```
0. Выход
1. Считать дерево из файла (bst и avl)
2. Распечатать bst дерево в виде дерева
3. Распечатать avl дерево в виде дерева
4. Распечатать bst дерево в виде картинки
5. Распечатать avl дерево в виде картинки
6. Добавить слово в дерево
7. Удалить слово из дерева
8. Найти слово в дереве
9. Считать хэш-таблицу из файла
10. Распечатать chaining таблицу
11. Распечатать таблицу с открытой адресацией
12. Показать картинку chaining таблицы
13. Показать картинку open таблицы
14. Добавить слово в таблицы
15. Удалить слово из таблиц
16. Найти слово в таблицах
17. Провести оценку эффективности поиска
```

Листинг 1. Опции меню.

2. Строка, содержащая имя файла, над которым необходимо произвести какие-либо операции.
3. Строка, содержащее число.

Результат

Результатом выполнения программы являются:

1. Вывод меню для выбора опции.

2. Графическое отображение бинарного дерева поиска, avl-дерева, хеш-таблицы.
3. Отображение бинарного дерева поиска, avl-дерева, хеш-таблицы в консоли.
4. Таблица оценки эффективности.

Способ обращения к программе

Пользователь обращается к программе при помощи исполняемого файла app.exe.

Возможные аварийные ситуации и ошибки пользователя

Все возможные аварийные ситуации об ошибке пользователя описаны типом status_t, включающим в себя коды возврата программы:

1. INPUT_ERR_CODE: ошибка ввода.
2. INVALID_PTR_CODE: ошибка работы с указателем.
3. MEMORY_ERR_CODE: ошибка работы с памятью.
4. FILE_OPEN_ERR_CODE: ошибка открытия файла.
5. FILE_READ_ERR_CODE: ошибка чтения файла.
6. NOTHING_TO_DELETE_CODE: ошибка удаления элемента из пустой структуры.
7. EMPTY_TREE_CODE: ошибка работы с пустым деревом.
8. NOT_FOUND_CODE: не найден необходимый элемент.

Описание внутренних структур данных

```
struct bst_node
{
    char *word;
    int count;
    bst_node_t *left;
    bst_node_t *right;
};

typedef struct avl_node_t
{
    char *word;
    int count;
    int height;
    struct avl_node_t *left;
    struct avl_node_t *right;
} avl_node_t;

typedef struct hash_node_t
{
    char *word;
    int count;
    struct hash_node_t *next;
    bool is_deleted;
```

```
} hash_node_t;

typedef struct
{
    hash_node_t **table;
    int size;
    int count;
    int collisions;
    int total_comparisons;
    int searches_count;
} hst_chaining_t;

typedef struct
{
    char **keys;
    int *counts;
    bool *occupied;
    bool *deleted;
    int size;
    int count;
    int collisions;
    int total_comparisons;
    int searches_count;
} hst_open_t;
```

Листинг 2. Внутренние структуры данных.

В данной лабораторной работе используются несколько необходимых структур данных.

Структура bst_node представляет собой узел двоичного дерева поиска, где word – это слово «частотного словаря», count – количество его «повторений», то есть то, насколько данное слово часто встречается, a left и right – указатели на «потомков» узла.

Структура avl_node_t – это узел avl-дерева. По аналогии с bst_node word и count – это слово и соответствующая этому слово частота его нахождений, a left и right – указатели на потомков. Поле height – высота узла.

Структура hash_node_t – узел для хеш-таблицы. Данная структура является универсальной для любой реализации хеш-таблицы. Поле next — указатель на следующую ноду списка, is_deleted — флаг, показывающий, была ли эта нода удалена из таблицы. Остальные поля структуры аналогичны по функционалу полям структур avl_node_t и bst_node.

Структура hst_chaining_t представляет собой хеш-таблицу с цепочками. Поле table — указатель на указатель на голову списка с узлами таблицы, size — размер таблицы, count — количество элементов в таблице, collisions — количество коллизий в таблице, total_comparisons — количество сравнений при поиске в таблице, searches_count — количество выполненных операций поиска.

Структура hst_open_t является хеш-таблицей с открытой адресацией, где keys — массив целых чисел, occupied — массив для проверки, занята ли ячейка таблицы, deleted — массив для проверки, была ли ячейка удалена, size — размер таблицы, count — количество элементов в таблице, collisions — количество коллизий в таблице, total_comparisons — количество сравнений при поиске в таблице, searches_count — количество выполненных операций поиска.

В листинге №3 представлены сигнатуры основных функций.

```
hst_chaining_t *create_hash_table_chaining(int size);
void free_hash_table_chaining(hst_chaining_t *ht);
unsigned int hash_function(const char *word, int table_size);
void hash_table_insert_chaining(hst_chaining_t *ht, const char *word);
hash_node_t *hash_table_search_chaining(hst_chaining_t *ht, const char
*word, int *comparisons);
void hash_table_delete_chaining(hst_chaining_t *ht, const char *word);
void display_hash_table_chaining(hst_chaining_t *ht);
void hash_table_statistics_chaining(hst_chaining_t *ht);
void rehash_chaining(hst_chaining_t *ht);

hst_open_t *create_hash_table_open(int size);
void free_hash_table_open(hst_open_t *ht);
void hash_table_insert_open(hst_open_t *ht, const char *word);
int hash_table_search_open(hst_open_t *ht, const char *word, int
*comparisons);
void hash_table_delete_open(hst_open_t *ht, const char *word);
void display_hash_table_open(hst_open_t *ht);
void hash_table_statistics_open(hst_open_t *ht);
void rehash_open(hst_open_t *ht);

status_t make_bst_from_file(bst_node_t **root, const char *filename);
status_t insert_bst_node(bst_node_t **root, const char *word);
status_t delete_bst_node(bst_node_t **root, const char *word, bool
*to_del_found);
status_t find_word_in_bst(bst_node_t **root, const char *word, bool flag,
int *comparisons);
status_t define_nodes_quantity_on_each_level(bst_node_t **root);
status_t balance_tree(bst_node_t **root);

status_t make_avl_from_file(avl_node_t **root, const char *filename);
size_t avl_height(avl_node_t *root);
status_t insert_avl_node(avl_node_t **root, const char *word);
status_t delete_avl_node(avl_node_t **root, const char *word, bool
*to_del_found);
status_t find_word_in_avl(avl_node_t **root, const char *word, bool flag,
int *comparisons);
size_t count_nodes_avl(avl_node_t *root);
```

Листинг 3. Сигнатуры основных функций.

Алгоритм

1. Циклически (пока не будет обнаружена ошибка ввода опции) выводится меню и предлагается выбрать один из пунктов.
2. Пользователь вводит номер необходимой опции (согласно нумерации в самом меню).
3. Происходит обработка ввода опции меню, если ввод корректен, согласно указанному пункту, выполняются конкретные операции.

4. В случае успешного завершения обработки выводится результат выполнения операций (если вывод результата предполагается данной опцией меню). Если в процессе выполнения возникла ошибка, соответствующее сообщение выводится в консоль.
5. Когда цикл завершается, программа возвращает код своего завершения.

Листинг 4. Общий алгоритм работы программы.

В программе используются две хеш-функции. Первая — использует умножение на «константу Кнута», вторая использует метод умножения с дробной частью.

```
// djb2
unsigned int hash_function1(const char *word, int table_size)
{
    unsigned long hash = 5381;
    int c;

    while ((c = (unsigned char)*word++))
        hash = ((hash << 5) + hash) + c;

    return (unsigned int)(hash % table_size);
}

// sdbm
unsigned int hash_function2(const char *word, int table_size)
{
    unsigned long hash = 0;
    int c;

    while ((c = (unsigned char)*word++))
        hash = c + (hash << 6) + (hash << 16) - hash;

    return (unsigned int)(hash % table_size);
}
```

Листинг 5. Хеш-функции, используемые в программе.

Если для поиска элемента в таблице требуется больше 4х сравнений, то хеш-функция меняется и таблица перестраивается. Изначально размер хеш-таблицы составляет 10 ячеек, если в ходе выполнения программы загруженность таблицы превышает 70%, то она перестраивается. При каждом перестроении размер таблицы увеличивается в 1.3 раза.

1. Если узел пуст – создание нового узла
2. Если символ меньше – рекурсивная вставка в левое поддерево
3. Если символ больше – рекурсивная вставка в правое поддерево
4. Если символ равен – увеличить счетчик
5. Вернуть указатель на корень поддерева

Листинг 6. Алгоритм вставки в BST.

1. Если узел пуст – создание нового узла
2. Если символ меньше – рекурсивная вставка в левое поддерево
3. Если символ больше – рекурсивная вставка в правое поддерево
4. Если символ равен – увеличить счетчик
5. Вернуть указатель на корень поддерева

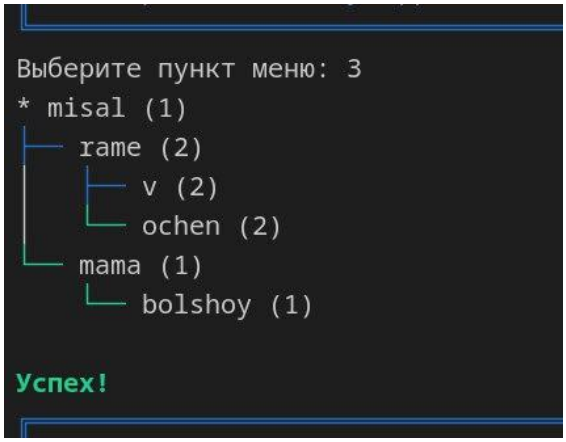
Листинг 7. Алгоритм вставки в AVL.

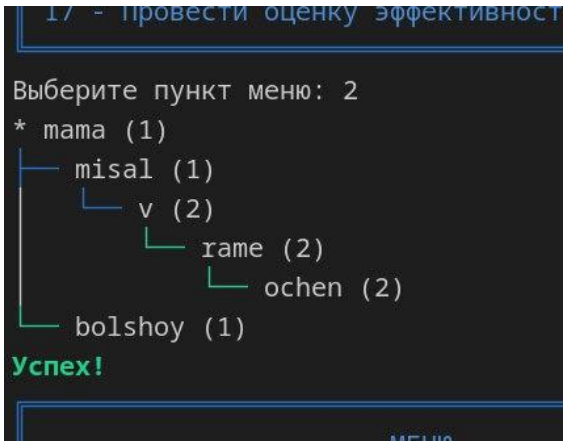
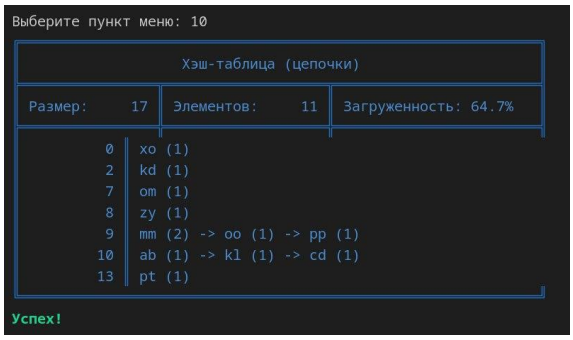
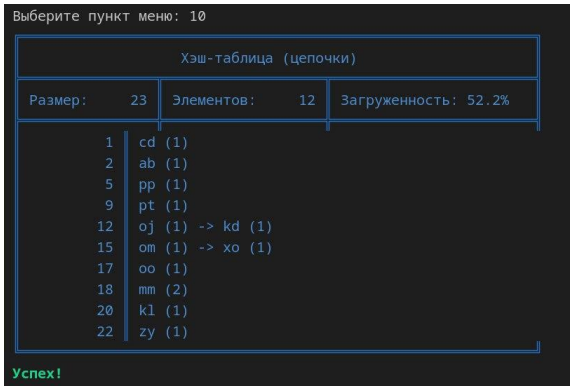
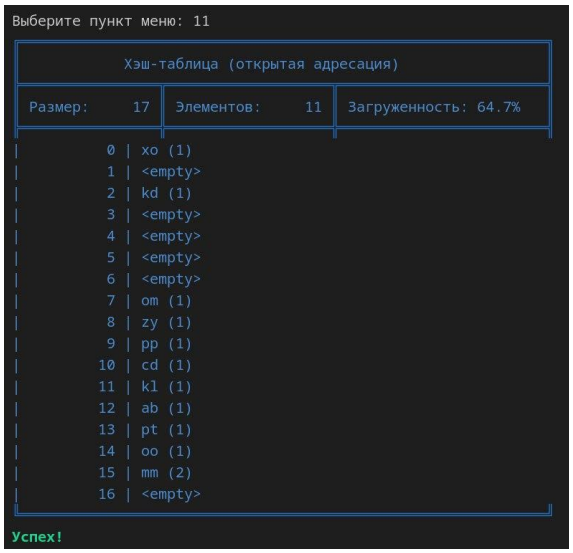
1. Если баланс узла становится больше единицы (левое поддерево глубже) и новый элемент вставлен влево от левого потомка, выполняется правый поворот вокруг текущего узла.
2. Если баланс узла становится меньше минус единицы (правое поддерево глубже) и новый элемент вставлен вправо от правого потомка, выполняется левый поворот вокруг текущего узла.
3. Если баланс узла больше единицы и новый элемент вставлен вправо от левого потомка, сначала выполняется левый поворот для левого потомка, а затем правый поворот для текущего узла.
4. Если баланс узла меньше минус единицы и новый символ вставлен влево от правого потомка, выполняется симметричный двойной поворот: правый поворот для правого потомка, левый поворот для текущего узла.
5. Возвращается обновленный указатель на текущий узел (корень поддерева после балансировки).
6. Шаги 1-5 рекурсивно применяются для каждого элемента дерева.
7. В результате работы алгоритма возвращается корень построенного AVL.

Листинг 8. Алгоритм построения AVL (балансировки BST).

Тесты

Позитивные тесты

Номер теста	Описание	Ожидаемый результат
1	Корректный ввод пункта меню.	
2	Ввод существующего файла.	
3	Вывод avl-дерева.	

4	Вывод двоичного дерева поиска.	
5	Исходная хеш-таблица с открытой адресацией.	
6	Хеш-таблица с открытой адресацией после реструктуризации (добавлен элемент oj).	
7	Исходная хеш-таблица с цепочками.	

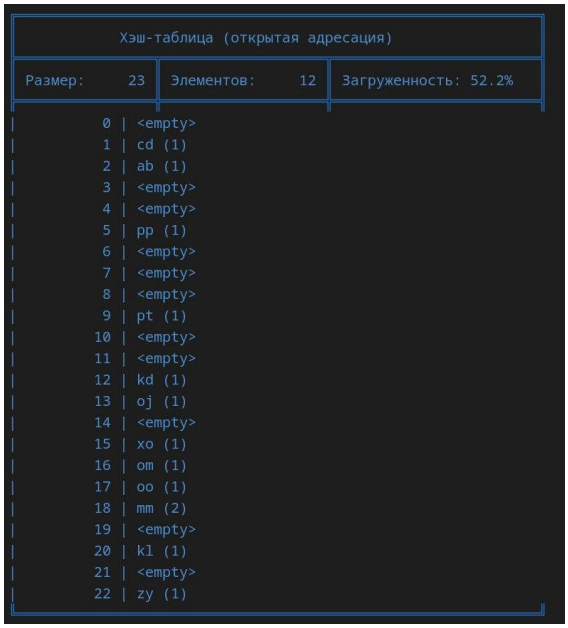
8	Хеш-таблица с цепочками после реструктуризации (добавлен элемент oj).	

Таблица 1. Позитивные тесты.

Негативные тесты

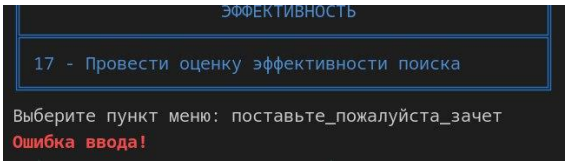
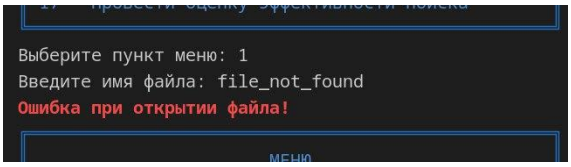
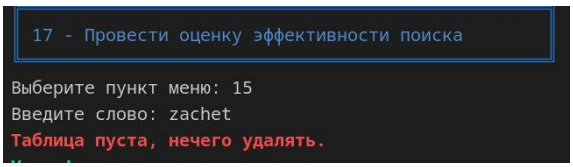
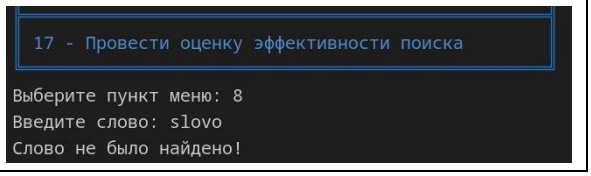
Номер теста	Описание	Ожидаемый результат
1	Некорректный ввод пункта меню.	
2	Ввод несуществующего файла.	
3	Удаление несуществующего элемента.	
4	Поиск в пустом дереве.	

Таблица 2. Негативные тесты.

Оценка эффективности

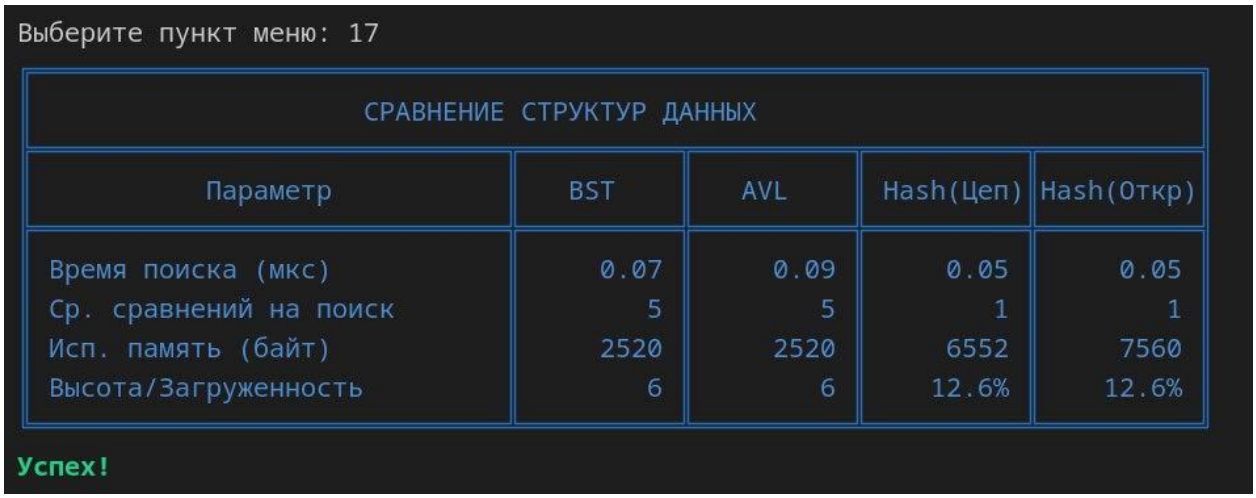


Рис. 1. Сравнение эффективности поиска слов в зависимости от структуры данных (63 слова, сбалансированное дерево).

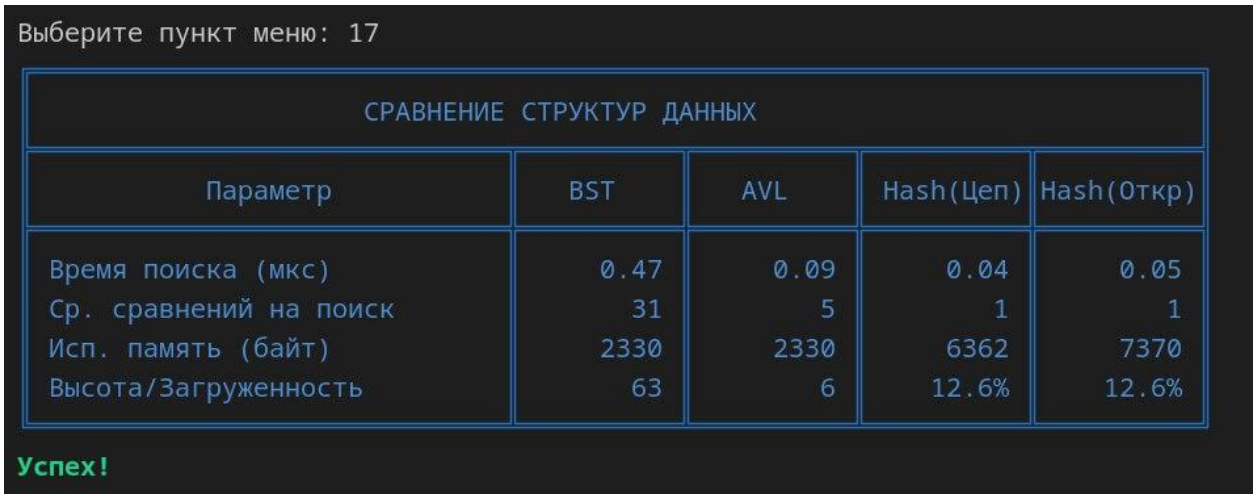


Рис. 2. Сравнение эффективности поиска слов в зависимости от структуры данных (63 слова, вырожденное дерево).

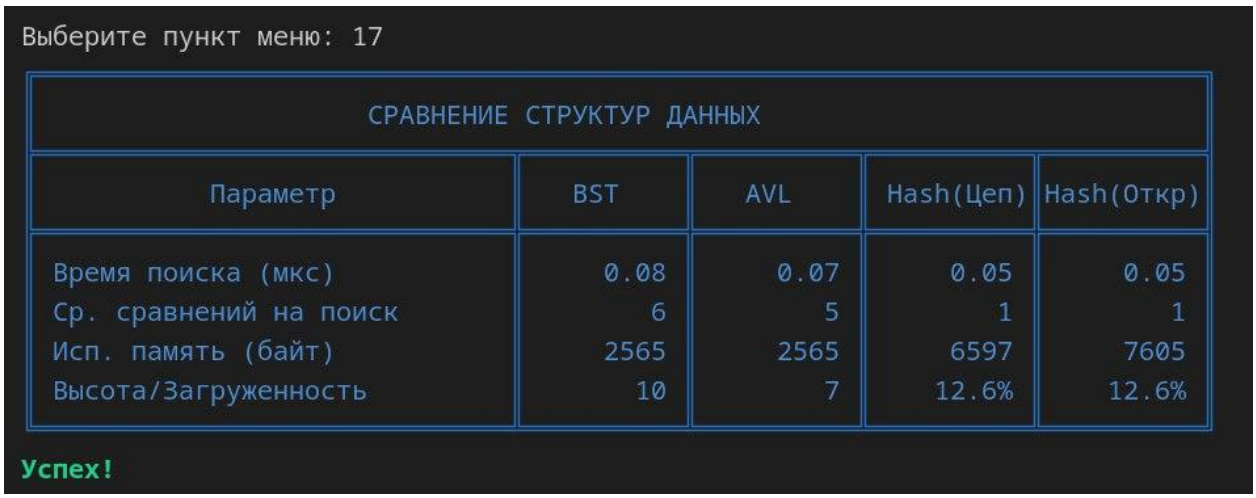


Рис. 3. Сравнение эффективности поиска слов в зависимости от структуры данных (63 слова, случайные слова).

Для каждой структуры поиск осуществлялся 1000 раз, после чего вычислялось среднее арифметическое время поиска. В трех файлах с разным содержанием было 63 слова (сбалансированное дерево, вырожденное и случайное).

Из данных рисунков 1, 2 и 3 очевидно, что поиск элемента в хеш-таблицах работает быстрее, чем поиск элемента в деревьях. Причем хеш-таблица с открытой адресацией и хеш-таблица с цепочками одинаково эффективны по времени поиска. AVL дерево работает быстрее чем BST, уступая ему только в том случае, когда дерево является сбалансированным. Если дерево вырожденное, BST уступает AVL дереву значительно (примерно в 5.2 раза медленнее).

В результате сравнения эффективности данных структур по памяти AVL и BST всегда показывают одинаковые результаты, что закономерно, так как они имеют равное количество листьев. В случае с хеш-таблицами очевидно, что хеш-таблица с открытой адресацией занимает больше памяти, чем хеш-таблица с цепочками ввиду больших накладных расходов при небольшой выборке данных (63 элемента).

Вывод

Выбор хеш-таблиц для решения задач, связанных с хранением и поиском элементов в структуре, оправдан в случае высоких требований к скорости поиска, отсутствия строгих ограничений на потребляемую память, а также большого объема обрабатываемых данных, так как использование алгоритмов хеширования, расширения хеш-таблиц, разрешения коллизий не являются целесообразным для небольших наборов данных или в условиях, где важнее предсказуемость времени выполнения и простота реализации.

В свою очередь, выбор деревьев (особенно AVL) обоснован, когда стабильность времени выполнения операций важна даже в худшем случае, данные динамически изменяются и объем данных относительно невелик.

Контрольные вопросы

1. *Чем отличается идеально сбалансированное дерево от АВЛ дерева?*
   В идеально сбалансированном дереве (Perfectly Balanced Tree) каждый уровень дерева полностью заполнен узлами, и высота дерева минимальна. Такие деревья обеспечивают оптимальное время выполнения операций, но в практике редко встречаются из-за ограничений на количество элементов в дереве. AVL-дерево - это форма сбалансированного дерева двоичного поиска, в котором разница в высоте между левым и правым поддеревьями для каждого узла ограничена (высота различается не более чем на 1). Это обеспечивает быстрое выполнение операций вставки, удаления и поиска.
2. *Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?*

В AVL-дереве поиск выполняется так же, как и в обычном дереве двоичного поиска. Разница заключается в том, что AVL-дерево поддерживает балансировку после каждой операции вставки или удаления, чтобы сохранять свою структуру сбалансированной.

3. *Что такое хеш-таблица, каков принцип ее построения?*
   Хеш-таблица - это структура данных, позволяющая эффективно выполнять операции вставки, удаления и поиска. Она использует хеш-функцию для преобразования ключа в индекс массива, где хранятся значения. Принцип построения: Выбор хеш-функции. Выделение массива определенного размера. Разрешение коллизий (в случае, если два ключа хешируются в один и тот же индекс).
4. *Что такое коллизии? Каковы методы их устранения.*
   Коллизии возникают, когда два различных ключа хешируются в один и тот же индекс. Методы разрешения коллизий включают:
   - Цепочки: каждый индекс массива представляет собой связанный список.
   - Открытое хеширование: при коллизии производится поиск следующего свободного слота в массиве.
   - Двойное хеширование: используются две хеш-функции для определения следующего индекса при коллизии.
5. *В каком случае поиск в хеш-таблицах становится неэффективен?*
   Поиск в хеш-таблицах становится неэффективным при большом количестве коллизий, что может привести к увеличению длины цепочек или увеличению размера открытого адреса.
6. *Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хеш-таблицах и в файле.*
   - В AVL-деревьях и деревьях двоичного поиска поиск выполняется за время, пропорциональное логарифму числа элементов в дереве.
   - В хеш-таблицах, при эффективном хешировании, поиск может быть выполнен за постоянное время O (1).