



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 **Попов Ю.А.**

Группа **ИУ7-32Б**

Вариант 1

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

Студент _____ **Попов Ю.А.**

Преподаватель _____ **Барышникова М.Ю.**

Оглавление

Оглавление	2
Описание условия задачи	3
Описание технического задания	3
Входные данные:	3
Выходные данные:	4
Действие программы:	4
Обращение к программе:	4
Аварийные ситуации:	5
Описание структуры данных	5
Описание алгоритма	7
Удаление узла в двоичном дереве поиска:	8
Удаление узла в AVL:	8
Перебалансировка AVL:	8
Хэш-таблица	9
Описание основных функций	9
Набор тестов	11
Оценка эффективности	15
Объем занимаемой памяти (байты)	15
Временные замеры (нс)	16
Ответы на контрольные вопросы	19
1. Чем отличается идеально сбалансированное дерево от AVL дерева?	19
2. Чем отличается поиск в AVL-дереве от поиска в дереве двоичного поиска	20
3. Что такое хеш-таблица, каков принцип ее построения?	20
4. Что такое коллизии? Каковы методы их устранения.	20
5. В каком случае поиск в хеш-таблицах становится неэффективен?	21
6. Эффективность поиска в AVL деревьях, в дереве двоичного поиска, в хеш-таблицах и в файле.	21
Вывод	21

Описание условия задачи

Построить двоичное дерево поиска из букв вводимой строки. Сбалансировать дерево (задача №6) после удаления повторяющихся букв. Вывести его на экран в виде дерева. Составить хеш-таблицу, содержащую буквы и количество их вхождений во введенной строке. Вывести таблицу на экран. Реализовать операции добавления и удаления введенной буквы во всех структурах. Осуществить поиск введенной буквы в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

Описание технического задания

Входные данные:

При работе с программой на экран выводится меню:

- 0 - Выход
- 1 - Тестирование двоичного дерева поиска
- 2 - Тестирование AVL дерева
- 3 - Тестирование хэш таблицы с открытой адресацией
- 4 - Тестирование хэш таблицы с закрытой (цепочной) адресацией
- 5 - Решение задачи
- 6 - Тестирование эффективности

Пользователь вводит целое число-команду от 0 до 6. Программа выполняет операции, в зависимости от выбранной опции. Если пользователь выбрал операции 1-4, то пользователь попадает в подпрограмму для тестирования, где должен ввести номер операции и элемент, один символ, для вставки в случае, если выбрана операция “добавление” или “поиск”. Выбрав пятую команду, пользователь должен ввести путь к файлу с данными.

Остальные операции не требуют входных данных. Чтобы корректно завершить программу, необходимо воспользоваться соответствующей командой.

Выходные данные:

Если пользователь выбрал 1-2 операцию, то он получит изображение, построенного дерева. Если выбраны операции 3-4, то на экран будет выведено представление хэш таблицы. При выборе 5 операции, будет выведены изображения деревьев, на основе строки, а также сравнение эффективности двоичного дерева поиска и авл-дерева. При выборе 6 операции, на экран будут выведены результаты замерного эксперимента.

При тестировании дерева и хэш-таблицы, в зависимости от операции на экран выводится либо дерево, либо его удаленный элемент.

Действие программы:

Программа работает до тех пор, пока пользователь не введет число 0. Программное обеспечение позволяет ввести команды от 0 до 6, в случае если команда выходит за этот диапазон, или не подходит по типу данных, выводится соответствующее сообщение, и программа завершает работу.

Обращение к программе:

Программа запускается через терминал, в командной строке, с помощью команды ./app.exe. После запуска. пользователю будет доступно меню программы.

Аварийные ситуации:

1. ERR_STRING – Ошибка при вводе строки;
2. ERR_OPERATION – Неверный выбор операции;
3. ERR_MEMORY_ALLOCATION - Ошибка при выделении памяти;
4. ERR_DATA_INPUT – Ошибка при вводе данных;
5. ERR_FILE - Неверный файл для построения графика;
6. WARNING_NO_EL - Нет нужного элемента для поиска или удаления
7. WARNING_REPEAT - Предупреждение о том, что элемент уже есть в дереве или хэш-таблице

Во всех нештатных ситуациях, программа уведомляет о месте, где произошла ошибка.

Описание структуры данных

В качестве данных, которые будет хранить дерево был выбран символ. Это обусловлено условием задачи.

Структура для хранения одного элемента дерева представлена на листинге 1.

value - 1 символ для хранения

repeated - Количество повторов этого символа

```
typedef struct _data_type_  
{  
    char value;  
    int repeat;  
} data_t;
```

листинг 1. Структура data_t

Структура для хранения узла двоичного дерева поиска представлена на листинге 2.

data - данные

left - Указатель на левого потомка

right - Указатель на правого потомка

```
struct _binary_search_tree_t_
{
    data_t data;
    struct _binary_search_tree_t_ *left; // Левый потомок
    struct _binary_search_tree_t_ *right; // Правый потомок
};
```

листинг 2. Структура bst_tree_t

Структура для хранения узла авл-дерева поиска представлена на листинге 3.

left - Указатель на левого потомка

right - Указатель на правого потомка

height - высота узла

data - данные

```
struct _avl_tree_type_
{
    struct _avl_tree_type_ *left; // Левый потомок
    struct _avl_tree_type_ *right; // Правый потомок

    int height; // Высота
    data_t data; // Данные
};
```

листинг 3. Структура avl_tree_t

Структура для хэш-таблицы с открытой адресацией представлена на листинге 4.

state_t - Перечисляемый тип данных для, описания состояния узла

data - данные в узле

size - Размер хэш-таблицы

```
typedef enum
{
    STATE_EMPTY,
```

```

        STATE_REMOVED,
        STATE_BUSY
    } state_t;

    // Один элемент хэш-таблицы
    struct _open_ht_item_
    {
        state_t state; // Состояние
        data_t data;   // Данные
    };

    // Основная структура для Хэш-таблицы
    struct _open_ht_type_
    {
        struct _open_ht_item_ *table; // Массив данных
        size_t size;                  // Размер
    };

```

листинг 4. Структура для описания хэш-таблицы с открытой адресацией

Структура для хэш-таблицы с закрытой адресацией представлена на листинге 5.

table - Массив списков

size - Размер хэш-таблицы

```

struct _close_ht_type_
{
    list_t **table; // Массив списков
    size_t size;    // размер
};

```

листинг 5. Структура для описания хэш-таблицы с закрытой адресацией

Описание алгоритма

1. После запуска программы выводит приглашение к вводу и меню.
2. Программа принимает номер операции, и запускает соответствующие действия в соответствии с номером команды
3. При решении задачи пользователь вводит строку
4. Строка поэлементно добавляется в двоичное дерево поиска
5. Повторяющиеся элементы строки удаляются из двоичного дерева поиска

6. Получившееся дерево, конвертируется в АВЛ-дерево, результат выводится на экран.

Удаление узла в двоичном дереве поиска:

1. Если узел - лист, просто удаляем его.
2. Если у узла один потомок, заменить узел на его потомка.
3. Если два потомка, ищем минимум в правом поддереве (или максимум в левом), заменяем значение удаляемого узла на найденное и удаляем узел-замену рекурсивно.

Удаление узла в АВЛ:

1. Удаление такое же как в ддп.
2. После удаления пересчитываем баланс-фактор для всех узлов на пути вверх от удаленного.
3. Если баланс-фактор узла меньше -1 или больше 1, выполняем вращение для восстановления баланса.

Перебалансировка АВЛ:

1. LL (левый левый): правое вращение.
2. RR (правый правый): левое вращение.
3. LR (левый правый): сначала левое вращение для левого поддерева, затем правое вращение.
4. RL (правый левый): сначала правое вращение для правого поддерева, затем левое вращение.

После вращения пересчитываются высоты узлов.

Хэш-таблица

Хэш-функция: преобразует данные в хэш. Я использовал простую хэш-функцию, которая возвращает остаток от деления номера символа в таблице ASCII на размер таблицы.

Открытая адресация: В реализации таблица создается с полями для хранения данных, статуса ячеек (0 — свободна, 1 — занята, 2 — удалена). Для вставки сначала вычисляется индекс через хэш-функцию, затем проверяется, свободна ли ячейка. Если занята, используется линейное пробирование: идём к следующей ячейке, пока не найдём свободную. Если количество превышает установленный размер, то выполняется реструктуризация. Аналогично при поиске, проверяем индекс от хэш-функции и продолжаем искать, если не нашли совпадение. Для удаления, как в поиске ищем нужные данные, освобождаем память, помечаем ячейку как удалённую.

Закрытая адресация: В этой реализации таблица хранит массив указателей на связанные списки (цепочки). Для вставки сначала вычисляется индекс через хэш-функцию, после чего создается новый узел и добавляется в конец списка в соответствующей ячейке. При поиске идём по списку в ячейке, сравниваем каждое слово с искомым. Для удаления идём по списку и ищем нужный узел. Если нашли, то удаляем элемент из списка, и освобождаем память.

Описание основных функций

Функции для работы с двоичным деревом поиска

```
bst_tree_t *bin_tree_create_node(data_t data)
int bin_tree_insert(bst_tree_t **root, data_t data)
bst_tree_t *bin_tree_search(bst_tree_t *root, data_t data)
int bin_tree_remove(bst_tree_t **root, data_t data)
int bin_tree_show(bst_tree_t *tree);
void bin_tree_free(bst_tree_t *tree);
void tree_remove_repeat(bst_tree_t **tree);
```

```
void bin_tree_inorder_traversal(const bst_tree_t *root, int is_head,
int is_color);
```

Создание элемента, вставка, удаление, поиск, обход. Вывод дерева на экран, освобождение памяти, удаление дубликатов из строки.

Функции для работы с авл деревом

```
void avl_tree_test(void);
// Функции для создания элемента
avl_tree_t *avl_tree_create_node(data_t data);
// Очистка памяти из-под всего дерева
void avl_tree_free(avl_tree_t *root);
// Функции для работы с деревом
// Вставка элемента в дерево
error_t avl_tree_insert(avl_tree_t **tree, data_t data);
// Удаление элемента из AVL дерева
avl_tree_t *avl_tree_remove(avl_tree_t *tree, data_t element);
// Поиск элемента в AVL дереве
avl_tree_t *avl_tree_search(avl_tree_t *root, data_t element);
// обходы дерева
void avl_tree_apply_preorder(avl_tree_t *tree, avl_tree_apply_fn_t
apply_fn, void *arg);
void avl_tree_apply_inorder(avl_tree_t *tree, avl_tree_apply_fn_t
apply_func, void *arg);
void avl_tree_apply_postorder(avl_tree_t *tree, avl_tree_apply_fn_t
apply_
```

Создание элемента, вставка, удаление, поиск, обходы. Вывод дерева на экран, освобождение памяти.

Функции для работы с хэш таблицей с открытой адресацией

```
// Функция для тестирования Hash таблицы на открытой адресации
void open_ht_test(void);
// Создание таблицы ht;
open_ht_t *open_ht_create(size_t size);
// Освобождение памяти из-под хэш таблицы
void open_ht_free(open_ht_t **ht);

// реструктуризация
int open_ht_restruct(close_ht_t **ht);

error_t open_ht_insert(open_ht_t **ht, data_t element, bool
*is_restructured);
int open_ht_remove(open_ht_t *ht, data_t data);
```

```
int open_ht_search(open_ht_t *ht, data_t data, size_t *cmp);

void open_ht_print(const close_ht_t *ht)
```

Создание элемента, вставка, удаление, поиск, вывод на экран, реструктуризация, освобождение памяти.

Функции для работы с хэш таблицей с закрытой адресацией

```
// Создание таблицы
close_ht_t *close_ht_create(size_t size);
// Освобождение памяти и удаление всех элементов
void close_ht_free(close_ht_t **ht);
// Очистка памяти из-под всех элементов
int close_ht_clear(close_ht_t *ht);

// реструктуризация
int close_ht_restruct(close_ht_t **ht);

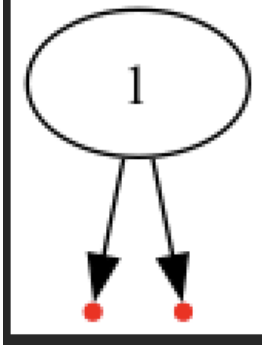
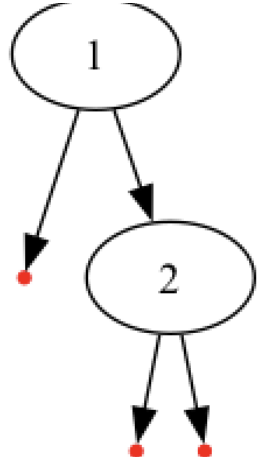
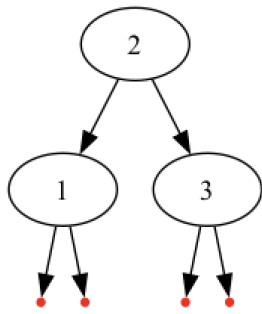
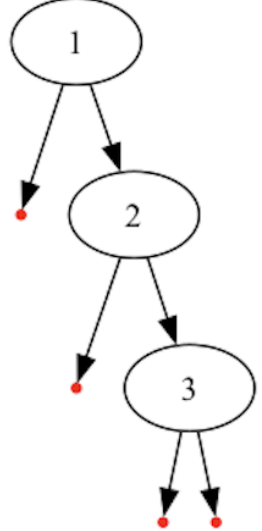
// Основные функции для работы с хэш таблицей
int close_ht_insert(close_ht_t **ht, data_t *data, bool *is_restruct);
int close_ht_remove(close_ht_t *ht, data_t *data);
int close_ht_search(const close_ht_t *ht, data_t *data, size_t *cmp);

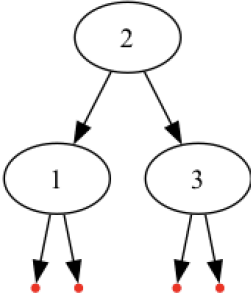
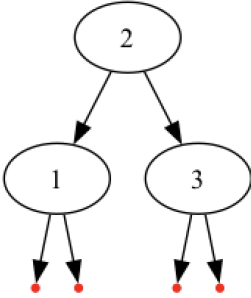
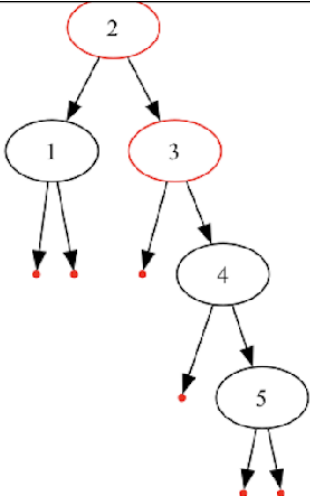
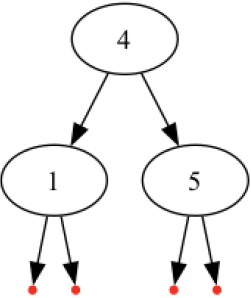
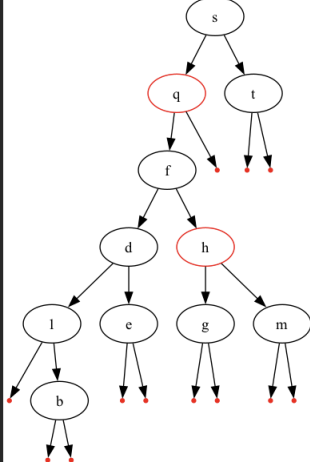
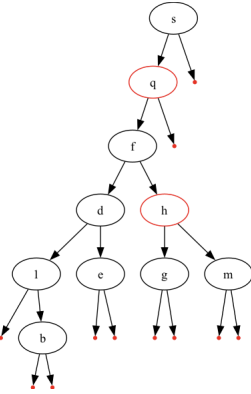
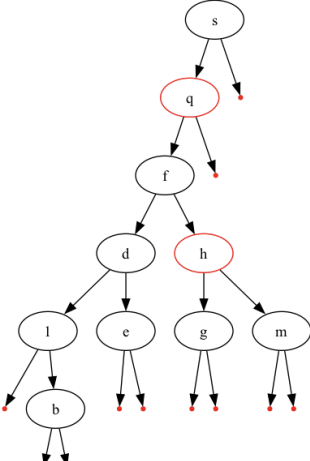
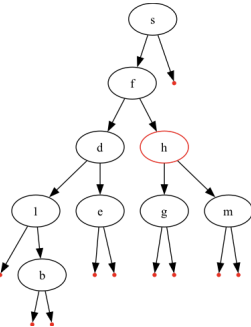
// Вывод на экран
void close_ht_print(const close_ht_t *ht);
```

Создание элемента, вставка, удаление, поиск, вывод на экран, реструктуризация, освобождение памяти.

Набор тестов

№	Описание теста	Ввод	Вывод
1	Некорректный ввод операции	abc	Ошибка при выборе операции.
2	Некорректный ввод операции. Выход за диапазон	10000	Ошибка при выборе операции.
3	Некорректный ввод операции. Отрицательное число	-100	Ошибка при выборе операции
6	Пустая строка		Ошибка при вводе строки
7	Неверный ввод данных для дерева.	aaa	Ошибка,

	Ввод строки		неверный ввод элемента
8	Добавление одного элемента в дерево	1	
9	Добавление одного потомка в двоичное дерево поиска	1 2	
10	Добавление двух потомков (в виде дерева) в двоичное дерево поиска	2 1 3	
11	Добавление двух потомков (в виде списка) в двоичное дерево поиска	1 2 3	

12	Удаление дубликатов (нет дубликатов)		
13	Удаление дубликатов (символы дубликаты из исходной строки) из двоичного дерева поиска		
14	Удаление элемента без потомков из двоичного дерева поиска		
15	Удаление элемента с одним потомком из двоичного дерева поиска		

	реструктуризация	4 5	2 [Занято] 'a' 3 [Занято] '4' 4 [Занято] '5'
23	Добавление элемента в пустую хэш-таблицу с закрытой адресацией	a	00. [a] -> [NULL]
24	Добавление элемента в хэш-таблицу с закрытой адресацией, требуется реструктуризация	a 2 3 4 5	00. [2] -> [4] -> [NULL] 01. [a] -> [3] -> [5] -> [NULL]
25	Поиск элемента в хэш-таблице с открытой адресацией	2	Элемент найден. Время поиска: 0.45 Количество сравнение 1
26	Поиск элемента в хэш-таблице с закрытой адресацией	5	Элемент найден. Время поиска: 0.89 Количество сравнение 3

Оценка эффективности

Объем занимаемой памяти (байты)

Объем, памяти, занимаемой структурами, в зависимости от размера

Количество элементов	Память, занимаемая деревом (байт)	Память, занимаемая авл деревом(байт)	Память, занимаемая хэш-таблицей с открытой адресацией байт)	Память, занимаемая хэш-таблицей с закрытой адресацией байт)
1	32	32	28	32
10	320	320	148	176
50	1600	1600	628	848
100	3200	3200	1228	1872

Таблица 1. Оценка занимаемой памяти, относительно размера.

Один элемент обоих деревьев занимает 32 байта, следовательно можно легко рассчитать размер всего дерева. Хэш таблица с открытой адресацией занимает: $16 + \langle \text{Ближайшее большее целое число} \rangle * 12$. Хэш таблица с закрытой адресацией занимает $16 + N * 16 * n$, где N - количество ячеек, а n число элементов списка в соответствующей ячейке. Таким образом эффективнее всего получилась хэш-таблица с открытой адресацией из-за того что каждый ее элемент занимает в памяти минимум места.

Временные замеры (нс)

Временные замеры осуществлялись с помощью библиотеки time.h. Ниже представлены таблица 2, сравнение времени поиска в различных структурах данных. Каждый эксперимент запускался 40 раз, и значение усреднялось. Замер хеш-таблиц был проведен при количестве коллизий не больше 4.

Количество уникальных элементов	Поиск элементов в бинарном дерева (мкс)	Поиск элементов в AVL-дереве (мкс)	Поиск элементов в хеш-таблице с открытой адресацией (мкс)	Поиск элементов в хеш-таблице с закрытой адресацией (мкс)
5	0.055	0.051	0.056	0.093
10	0.101	0.062	0.053	0.095
20	0.084	0.078	0.053	0.084
40	0.097	0.095	0.078	0.088

Таблица 2. Среднее время поиска во всех структурах данных

По таблице видно, что авл-дерево всегда быстрее чем бинарное дерево.

Кроме этого, видно что таблица с открытой адресацией оказалась быстрее за счет последовательного доступа.

В таблице 3, представлено количество сравнений для поиска в различных структурах данных. Замер хеш-таблиц был проведен при количестве коллизий не больше 4.

Количество уникальных элементов	Количество сравнений в бинарном дерева (мкс)	Количество сравнений в AVL-дерева (мкс)	Количество сравнений в хеш-таблице с открытой адресацией (мкс)	Количество сравнений в хеш-таблице с закрытой адресацией (мкс)
5	2	2	4	2
10	6	3	1	4
20	4	3	1	4
40	5	4	1	1

Таблица 3. Количество сравнений во время поиска.

Протестируем зависимости эффективности хеш-таблиц от количества максимальных коллизий. В таблице 4 представлены результаты среднего времени работы поиска, в зависимости от максимального количества коллизий. В таблице 5 демонстрируется среднее количество сравнений при поиске в зависимости от максимального количества коллизий.

Количество уникальных элементов	Максимальное количество коллизий	Поиск элементов в хеш-таблице с открытой адресацией (мкс)	Поиск элементов в хеш-таблице с закрытой адресацией (мкс)
10	1	0.046	0.047
20		0.047	0.050
40		0.056	0.063
10	4	0.050	0.079
20		0.050	0.072
40		0.050	0.052
10	10	0.064	0.072
20		0.073	0.074
40		0.050	0.059

Таблица 4. Среднее время работы поиска, в зависимости от максимального количества коллизий

Количество уникальных элементов	Максимальное количество коллизий	Количество сравнений в хеш-таблице с открытой адресацией	Количество сравнений в хеш-таблице с закрытой адресацией
10	1	1	1
20		1	1
40		1	1
10	4	1	4
20		1	4
40		1	1
10	10	1	10
20		11	6
40		1	8

Таблица 5. Количество сравнений, в зависимости от максимального количества коллизий

Ответы на контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от AVL дерева?

Идеально сбалансированное дерево — это дерево, в котором высота левого и правого поддеревьев каждого узла не превышает 1, что обеспечивает минимальную высоту дерева. AVL дерево — это сбалансированное дерево,

где разница высот поддеревьев любого узла не более 1. Идеально сбалансированное дерево всегда сбалансировано по всем уровням, в то время как AVL дерево поддерживает баланс только при добавлении или удалении узлов, и для этого могут понадобиться дополнительные операции перебалансировки.

2. Чем отличается поиск в AVL-дереве от поиска в дереве двоичного поиска

Поиск в AVL дереве всегда эффективнее потому что AVL-дерево всегда сбалансированное, в то время как двоичное дерево поиска может оказаться несбалансированным, а в худшем случае вырожденным.

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

Хеш-таблица — это структура данных, которая использует хеш-функцию для преобразования ключа в индекс массива, где хранится соответствующее значение. Принцип построения заключается в том, что для каждого ключа вычисляется хеш-значение, и этот индекс указывает на место хранения данных в таблице. Хеш-таблица позволяет выполнять операции вставки, поиска и удаления за амортизированное время $O(1)$, если коллизии минимальны.

4. Что такое коллизии? Каковы методы их устранения.

Коллизии — это ситуации, когда два различных элемента хешируются в одну и ту же ячейку хеш-таблицы. Методы устранения коллизий включают открытую и закрытую адресацию. В открытой адресации элементы размещаются в следующей свободной ячейке при столкновении, а в закрытой адресации для каждого индекса создается список, и элементы с одинаковым хешем добавляются в этот список.

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

Поиск в хеш-таблицах становится неэффективным, когда таблица сильно заполнена или хеш-функция неудачна, что приводит к большому числу коллизий. Это может значительно увеличить время поиска, так как приходится искать в длинных цепочках (в случае закрытой адресации) или использовать методы линейного поиска (в случае открытой адресации).

6. Эффективность поиска в AVL деревьях, в дереве двоичного поиска, в хеш таблицах и в файле.

Эффективность поиска: в AVL-деревьях поиск имеет сложность $O(\log n)$, в дереве двоичного поиска — от $O(\log n)$ до $O(h)$, где h — высота дерева. В хеш-таблицах поиск в идеале выполняется за $O(1)$, но в случае сильных коллизий может быть $O(n)$. В файле поиск обычно выполняется за $O(n)$, так как приходится проверять все элементы последовательности.

Вывод

AVL и ДДП:

структура AVL и ДДП деревьев одинаковая, но дополнительно хранится поле высоты, чтобы контролировать баланс.

Времена выполнения операций:

в ДДП — от $O(\log(n))$, до $O(n)$

в AVL поиск всегда $O(\log n)$, так как дерево всегда сбалансировано.

Таким образом AVL-дерево занимает немного больше памяти, но значительно выигрывает в скорости работы, однако оно обладает более сложной реализацией.

Хэш-таблицы:

Открытая адресация и закрытая адресация — методы обработки коллизий в хэш-таблицах. Основное различие в том, что при открытой адресации все 25 элементы хранятся в массиве, а при закрытой — элементы с одним и тем же хэшем хранятся в списках.

Времена выполнения операций:

Поиск: В идеале $O(1)$ для обеих реализаций, однако при открытой адресации поиск может занять $O(n)$ в случае высокой заполненности таблицы. В закрытой адресации — $O(1 + k)$, где k — длина списка.

Вставка: В открытой адресации — $O(1)$ в идеале, но $O(n)$ при реструктуризации. В закрытой — $O(1 + k)$.

Удаление: Открытая адресация сложнее, так как ячейки помечаются как удалённые, чтобы не нарушить работу поиска. Это $O(1)$ в идеале. В закрытой удаление проще и занимает $O(1 + k)$.

Память: Открытая адресация требует меньше памяти, так как использует только массив для хранения данных. Закрытая требует дополнительной памяти для списков, но это упрощает управление коллизиями. Открытая адресация плохо работает при заполненности таблицы более 70%. В закрытой адресации удобнее работать с коллизиями и легче удалять элементы, производительность меньше зависит от заполненности.

Вывод: Открытая адресация подходит для таблиц с низкой заполняемостью и ограниченным размером. Закрытая адресация эффективнее для больших таблиц с частыми коллизиями.