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»

Отчёт

по лабораторной работе №6

Название «Двоичные деревья поиска и хеш-таблицы»

Дисциплина «Типы и структуры данных»

Вариант 4

Студент ИУ7-31Б _____ Корниенко К.Ю.
(подпись, дата) (Фамилия И.О.)

Преподователь _____ Никульшина Т.А.
(подпись, дата) (Фамилия И.О.)

Москва, 2021

Содержание

Введение

Большинство современных информационных систем содержат в себе базы данных объем которых может превышать несколько гигабайт и к поиску данных в таких базах предъявляются особые требования, такие как максимальная скорость, прогнозируемость времени поиска и точность нахождения информации.

Простые алгоритмы перебора не способны обеспечить максимальную скорость и предоставить оценку времени выполнения операции поиска ввиду чего повсеместно заменяются на алгоритмы поиска с использованием двоичных деревьев. Доказательством этого служит сравнение скорости поиска в современных СУБД. Именно СУБД, использующие индексацию и двоичные деревья поиска показывают наибольшую производительность при поиске данных по таблице, содержащей большой объем данных. [1]

Целью данной работы является получение навыков применения двоичных деревьев, реализация основных операций над деревьями; построение и обработка хеш-таблицы; сравнение эффективности сбалансированных деревьев, двоичных деревьев поиска и хеш-таблиц.

Для выполнения поставленной цели необходимо решить следующие задачи:

- исследовать структуры данных: деревья и хеш-таблицы;
- сформулировать условие задачи;
- описать используемые структуры данных;
- привести схемы алгоритма балансировки и описание реализуемых хеш-функций;
- сформулировать требования к разрабатываемому программному обеспечению;
- определить средства программной реализации;
- реализовать алгоритм, решающий поставленную задачу;
- сравнить эффективность реализованного алгоритма для различных структур данных: сбалансированного дерева, двоичного дерева поиска, хеш-таблицы и файла.

1 Аналитический раздел

В данном разделе представлены основные теоритические сведения о двоичных деревьях и хеш-таблицах, а также описано техническое задание.

1.1 Основные теоритические сведения

Ниже представлены основные теоритические сведения, касаемые таких структур данных как двоичное дерево поиска, сбалансированное двоичное дерево поиска и хеш-таблица.

1.1.1 Двоичное дерево поиска

Дерево – это нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».

К основным операциям, выполняемым с различными структурами данных, можно отнести поиск среди набора данных. Один из широко используемых для этого методов — построение двоичного дерева поиска, которое ускоряет и упрощает задачу поиска данных в определенном наборе информации, структурирует заданную информацию для более эффективного ее дальнейшего использования, а при отсутствии необходимой информации возвращает указатель на пустой элемент [2].

Дерево двоичного поиска – это такое дерево, в котором все левые потомки моложе предка, а все правые – старше. Это свойство называется характеристическим свойством дерева двоичного поиска и выполняется для любого узла, включая корень. С учетом этого свойства поиск узла в двоичном дереве поиска можно осуществить, двигаясь от корня в левое или правое поддерево в зависимости от значения ключа поддерева.

1.1.2 Сбалансированное двоичное дерево поиска

Если при построении дерева поочередно располагать узлы слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более чем на единицу. Такое дерево называется идеально сбалансированным.

N элементов можно организовать в бинарное дерево с высотой не более $log_2(N)$, поэтому для поиска среди N элементов может потребоваться не больше $log_2(N)$ сравнений, если дерево идеально сбалансировано.

Адельсон-Вельский и Ландис сформулировали менее жесткий критерий сбалансированности таким образом: двоичное дерево называется сбалансированным, если у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. Такое дерево называется АВЛ-деревом.

Использование этого критерия приводит к легко выполняемой балансировке. При этом средняя длина поиска остается практически такой же, как и у идеально сбалансированного дерева. При включении узла в сбалансированное дерево возможны 3 случая: (рассматриваем включение в левое поддерево):

1) Левое и правое поддеревья становятся неравной высоты, но критерий сбалансированности не нарушается.

2) Левое и правое поддерево приобретают равную высоту и, таким образом, сбалансированность даже улучшается.

3) Критерий сбалансированности нарушается, и дерево надо перестраивать.

Алгоритм включения и балансировки существенно зависит от способа хранения информации о сбалансированности дерева. Одно из решений – хранить в каждой вершине показатель ее сбалансированности. В этом случае сбалансированность будет определяться как разность между высотой правого и левого поддеревьев

Отсюда следует, что дерево – это более подходящая структура для организации поиска, чем, например, линейный список.

1.1.3 Хеш-таблицы

Массив, заполненный в порядке, определенным хеш-функцией, называется *хеш-таблицей*. Хеш-функции позволяют по значению ключа определять сразу индекс элемента массива, в котором хранится информация. Минимальная трудоемкость поиска в хеш-таблице равна $O(1)$.

Принято считать, что хорошей является такая функция, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;
- функция должна распределять ключи в хеш-таблице наиболее равномерно.

Сформулируем понятие коллизии следующим образом. Пусть есть хеш-функция H. Если найдены две различных строки U, W со свойством $H(U) = H(W)$, то обнаружена коллизия при хешировании с помощью функции H [3].

Существует несколько возможных вариантов разрешения коллизий, которые имеют свои достоинства и недостатки.

Первый метод: внешнее (открытое) хеширование (метод цепочек).

В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором, так как при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким.

Второй метод: внутреннее (закрытое) хеширование (открытая адресация).

Состоит в том, чтобы полностью отказаться от ссылок. В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку (с шагом 1), до тех пор, пока не будет найден ключ K или пустая позиция в таблице. При этом, если индекс следующего просматриваемого элемента определяется добавлением

какого-то постоянного шага (от 1 до n), то данный способ разрешения коллизий называется линейной адресацией. Для вычисления шага можно также применить формулу:

$$h = h + a^2 \tag{1.1}$$

где a – это номер попытки поиска ключа. Этот вид адресации называется квадратичной или произвольной адресацией.

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3–4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента

1.2 Техническое задание

1.2.1 Общее задание

Построить дерево в соответствии с заданным вариантом задания. Вывести его на экран в виде дерева. Реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов. Сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты дерева и степени его ветвления. Построить хеш-таблицу по указанным данным. Вывести на экран деревья и хеш-таблицу. Сравнить эффективность поиска в двоичном дереве поиска, в сбалансированном дереве поиска и в хеш-таблице. Вывести на экран измененные структуры. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи.

1.2.2 Задание варианта

Вариант 4:

В текстовом файле содержатся целые числа. Построить ДДП из чисел файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Построить хеш-таблицу из чисел файла. Использовать закрытое хеширование для устранения коллизий. Осуществить удаление введенного целого числа в ДДП, в сбалансированном дерево, в хеш-таблице и в файле. Сравнить время удаления, объем памяти и количество сравнений при использовании различных (4-х) структур данных. Если количество сравнений в хеш-таблице больше указанного, то произвести реструктуризацию таблицы, выбрав другую функцию.

2 Конструкторский раздел

В данном разделе представлена информация об используемых структурах данных, а также приведены схемы реализуемых алгоритмов.

2.1 Функции хеширования

В рамках данной лабораторной работы реализованы две функции хеширования, реализованные по следующим схемам:

- метод деления;
- мультипликативная схема.

2.1.1 Метод деления

Метод деления основан на простой операции – взятии остатка от деления. Для данного метода возьмем остаток от деления на M:

$$h(K) = K \ mod \ M \tag{2.1}$$

2.1.2 Мультипликативная схема

Ниже представлена функция хеширования для мультипликативной формы.

Пусть ω — размер машинного слова. Метод состоит в выборе некоторой целой константы A, взаимно простой с ω, после чего можно положить:

$$h(K) = \lfloor M \left(\left(\frac{A}{\omega} K \right) \ mod \ 1 \right) \rfloor \tag{2.2}$$

Мультипликативная схема хеширования реализуется на основе хеширования Фиббоначи: $\frac{A}{\omega}$ приблизительно равно золотому сечению $\phi = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$.

2.2 Разрешение коллизий

Для разрешения коллизий используется схема адресации, известная как линейное исследование, которая использует циклическую последовательность проверок:

$$h(K),\ h(K)+1,\ \ldots,\ M-2,\ M-1,\ 0,\ 1,\ \ldots,\ h(K)-1 \tag{2.3}$$

2.3 Схемы алгоритмов

Ниже, на рисунке 2.1, представлен алгоритм балансировки для поддерева двоичного дерева поиска. Для полной балансировки необходимо рекурсивно применить этот алгоритм ко всем узлам дерева.

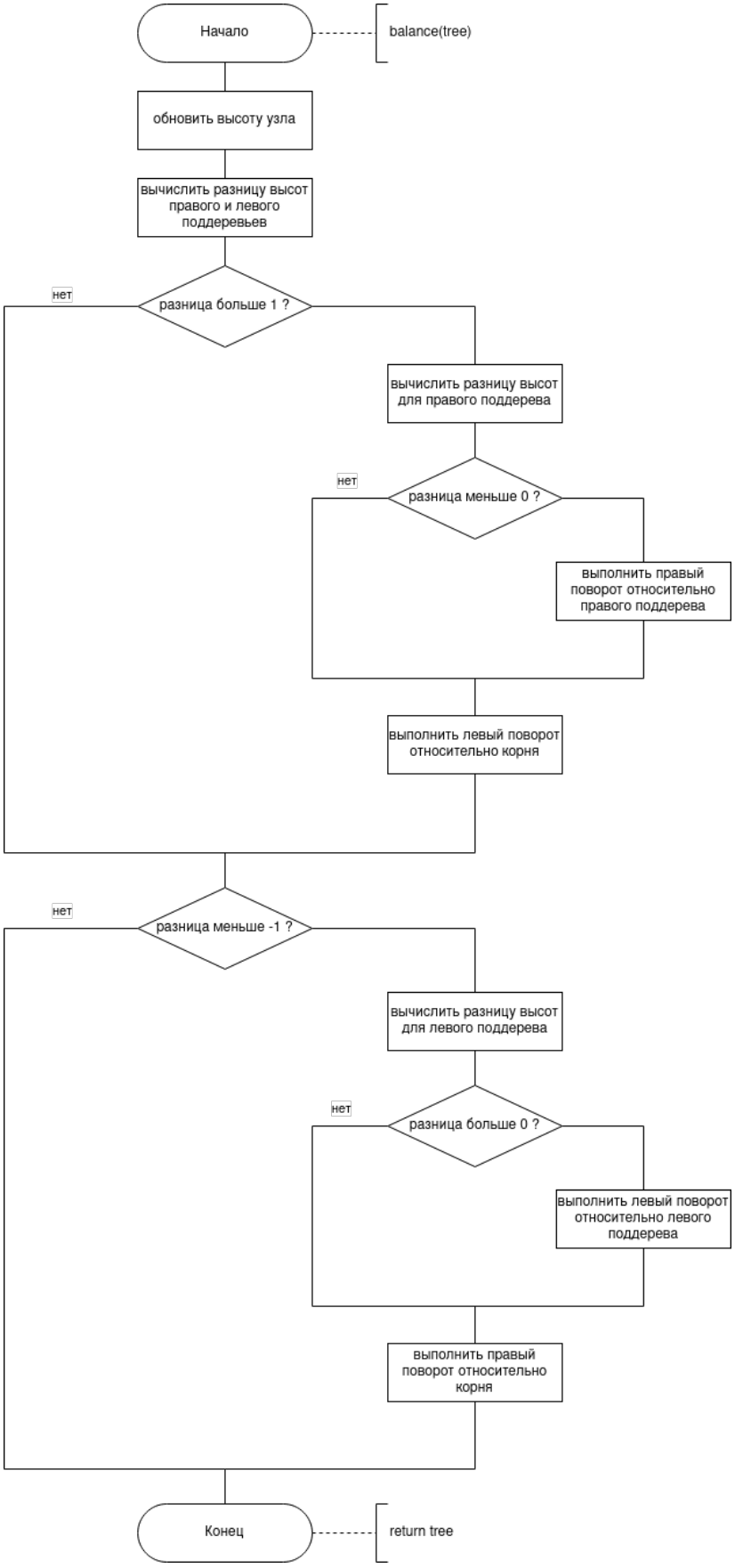

Рисунок 2.1 — Алгоритм балансировки

2.4 Структуры данных

Ниже, на листинге 2.1, представлена структура АВЛ-дерева.

Листинг 2.1 — Структура АВЛ-дерева

```
typedef struct avl_node
{
    data_t key;
    uint8_t height;

    struct avl_node *left;
    struct avl_node *right;
} node_t;
```

Ниже, на листинге 2.2, представлена структура хеш-таблицы.

Листинг 2.2 — Структура хеш-таблицы

```
typedef struct
{
    data_t key;
    bool has_value;
    size_t cmp_count;
} hash_data_t;

typedef struct
{
    size_t size;
    size_t step;
    hash_data_t *data;
    hf_t func;
} hash_table_t;
```

2.5 Реализуемые функции

Ниже, на листинге 2.3, представлены функции для работы с двоичными деревьями поиска и АВЛ-деревьями.

Листинг 2.3 — Функции для работы с деревьями

```
// ================================================================================
// ДДП
// ================================================================================

/**
    * @brief Ввод бинарного дерева поиска (не сбалансированного)
    *
    * @param filename имя файла, содержащего целые числа
    * @return node_t* (указатель на корень дерева)
    */
node_t *bst_input(const char *filename);

/**
    * @brief Вывод бинарного дерева поиска в графическом виде
    *
    * @param root корень ДДП
    */
void bst_output(const node_t *root, const char *graphname);

/**
    * @brief Освобождение памяти из-под бинарного дерева
    *
    * @param root корень ДДП
    */
void bst_destroy(node_t *root);

/**
    * @brief Поиск элемента в ДДП
    *
    * @param root корень ДДП
    * @param key искомый элемент
    * @return node_t* (искомый узел)
    */
const node_t *bst_search(const node_t *root, data_t key, size_t *cmp_count);

/**
    * @brief Удаление элемента из дДП
    *
    * @param root корень ДДП
    * @param key удаляемый элемент
    * @return node_t* (искомый узел)
```

```
42      */
43  node_t *bst_remove(node_t *root, data_t key, size_t *cmp_count);
44
45  double bst_get_mean_cmp_count(const node_t *root);
46
47  /**
48      * @brief Вычисляет размер занимаемой деревом памяти
49      */
50  size_t bst_sizeof(const node_t *root);
51
52  /**
53      * @brief Преобразование ДДП в АВЛ-дерево
54      *
55      * @param node узел дерева
56      * @return node_t* (корень сбалансированного АВЛ дерева)
57      */
58  node_t *bst_to_avl(node_t *node);
59
60  // ============================================================================
61  // АВЛ-деревья
62  // ============================================================================
63
64  /**
65      * @brief Выделяет память под узел дерева
66      *
67      * @param key значение узла
68      * @return node_t* (указатель на выделенную область памяти)
69      */
70  node_t *node_init(data_t key);
71
72  /**
73      * @brief Обёртка поля height, которая может работать с нулевыми указателями
74      *
75      * @param node узел дерева
76      * @return uint8_t (высота поддерева)
77      */
78  uint8_t height(node_t *node);
79
80  /**
81      * @brief Вычисление показателя балансировки узла
82      *
83      * @param node узел дерева, балансировка которого вычисляется
84      * @return int (показатель балансировки узла)
85      */
86  int bfactor(node_t *node);
87
88  /**
```

```
89      * @brief Восстанавливает корректное значения поля height
90      *
91      * @param node узел, высота которого исправляется
92      */
93  void fix_height(node_t *node);
94
95  /**
96      * @brief Балансировка при вставке в отбалансированное дерево
97      *
98      * @param node узел дерева
99      * @return node_t* (корень сбалансированного дерева)
100     */
101 node_t *balance(node_t *node);
102
103 /**
104     * @brief Вставить значение в дерево, сохранив балансировку
105     *
106     * @param root корень дерева
107     * @param key значение узла
108     * @return node_t* (корень сбалансированного дерева)
109     */
110 node_t *avl_insert(node_t *root, data_t key);
111
112 /**
113     * @brief Удаления ключа key из данного дерева
114     *
115     * @param root корень дерева
116     * @param key ключ, подлежащий удалению
117     * @param[out] cmp_count количество сравнений
118     * @return node_t* (указатель на новое дерево)
119     */
120 node_t *avl_remove(node_t *root, data_t key, size_t *cmp_count);
```

Ниже, на листинге 2.4, представлены функции для работы с хеш-таблицами.

Листинг 2.4 — Функции для работы с хеш-таблицами

```
1  /**
2  * @brief Нулевая хеш-таблица
3  *
4  * @return hash_table_t
5  */
6  hash_table_t hash_null(void);
7
8  /**
9  * @brief Выделение памяти под хеш-таблицу
10 *
11 * @param size размер хеш-таблицы
```

```
12 * @param step шаг для устранения коллизий
13 * @param func хеш-функция
14 * @return hash_table_t
15 */
16 hash_table_t hash_init(size_t size, size_t step, hf_t func);
17
18 /**
19 * @brief Освобождение памяти из-под хеш-таблицы
20 *
21 * @param table хеш-таблица
22 */
23 void hash_destroy(hash_table_t *table);
24
25 /**
26 * @brief Ввод хеш-таблицы
27 *
28 * @param table хеш-таблицы
29 * @param size максимальный размер хеш-таблицы
30 * @param filename имя входного файла
31 * @return int
32 */
33 int hash_input(hash_table_t *table, size_t size, const char *filename);
34
35 /**
36 * @brief Вывод хеш-таблицы
37 *
38 * @param table хеш-таблица
39 */
40 void hash_output(const hash_table_t *table);
41
42 /**
43 * @brief Поиск в хеш-таблице
44 *
45 * @param table хеш-таблица
46 * @param data искомый ключ
47 * @return true
48 * @return false
49 */
50 bool hash_find(hash_table_t *table, data_t data);
51
52 /**
53 * @brief Вставка в хеш-таблицу
54 *
55 * @param table хеш-таблица
56 * @param data ключ
57 * @return int
58 */
```

```
59  int hash_insert(hash_table_t *table, data_t data);
60
61  /**
62  * @brief Удаление из хеш-таблицы
63  *
64  * @param table хеш-таблица
65  * @param data удаляемый ключ
66  * @param cmp_count максимальное число сравнений
67  * @return int
68  */
69  int hash_remove(hash_table_t *table, data_t data, size_t *cmp_count);
70
71  /**
72  * @brief Вычисление среднего числа сравнений
73  *
74  * @param table хеш-таблица
75  * @return double
76  */
77  double hash_get_mean_cmp_count(const hash_table_t *table);
78
79  /**
80  * @brief Вычисление максимального числа сравнений
81  *
82  * @param table хеш-таблица
83  * @return size_t
84  */
85  size_t hash_get_max_cmp_count(const hash_table_t *table);
86
87  /**
88  * @brief Размер занимаемый под хеш-таблицу памяти
89  *
90  * @param table хеш-таблица
91  * @return size_t
92  */
93  size_t hash_sizeof(const hash_table_t *table);
94
95  /**
96  * @brief Реструктуризация хеш-таблицы
97  *
98  * @param table хеш-таблица
99  * @return int
100 */
101 int hash_restruct(hash_table_t *table);
102
103 /**
104 * @brief Хеш-функция на основе деления
105 *
```

```
106 * @param data данные, подлежащие хешированию
107 * @param table_size размер хеш-таблицы
108 * @return size_t
109 */
110 size_t hasher1(data_t data, size_t table_size);
111
112 /**
113 * @brief Хеш-функция на основе хеширования Фиббоначи (мультипликативная форма)
114 *
115 * @param data данные, подлежащие хешированию
116 * @param table_size размер хеш-таблицы
117 * @return size_t
118 */
119 size_t hasher2(data_t data, size_t table_size);
```

3 Технологический раздел

В данном разделе представлены требования к разрабатываемому ПО, а также представлены листинги кодов реализованных алгоритмов.

3.1 Средства реализации

Выбранным языком для реализации алгоритмов является структурный язык C, в соответствии с требованиями, предъявляемыми курсом «Типы и структуры данных».

Для получения исполняемого файла используется компилятор gcc [4]. В качестве инструмента сборки используется утилита make [5].

3.2 Требования к ПО

Ниже представлены требования к разрабатываемому ПО.

- на вход программа получает имя файла, в котором содержатся целые числа, элемент, который необходимо вставить в дерево, размерность хеш-таблицы и максимальное количество сравнений для хеш-функции;
- на выходе – дерево из чисел файла, сбалансированное дерево, дерево после удаления элемента, сбалансированное дерево после удаления элемента, хеш-таблица, время удаления в двоичном дереве поиска, АВЛ-дереве, хеш-таблице и файле, а также объемы занимаемой памяти каждой структурой.

3.3 Реализация алгоритмов

3.3.1 Функции для работы с деревом двоичного поиска

Ниже на листинге 3.1 приведена реализация алгоритма вставки в двоичное дерево поиска.

Листинг 3.1 — Реализация алгоритма вставки в двоичное дерево поиска

```
node_t *bst_insert(node_t *root, data_t key)
{
    if (root == NULL)
        return node_init(key);
    else if (key < root->key)
    {
        node_t *tmp = bst_insert(root->left, key);

        if (tmp == NULL)
            return tmp;

        root->left = tmp;
    }
    else if (key > root->key)
    {
        node_t *tmp = bst_insert(root->right, key);

        if (tmp == NULL)
            return tmp;

        root->right = tmp;
        tmp = NULL;
    }

    fix_height(root);
    return root;
}
```

На листинге 3.2 представлена реализация алгоритма удаления ключа из двоичного дерева поиска.

Листинг 3.2 — Реализация алгоритма удаления ключа из двоичного дерева поиска

```
node_t *bst_remove(node_t *root, data_t key, size_t *cmp_count)
{
    if (root == NULL)
        return NULL;

    *cmp_count++;
    if (key < root->key)
        root->left = bst_remove(root->left, key, cmp_count);
    else if (key > root->key)
        root->right = bst_remove(root->right, key, cmp_count);
    else
    {
        node_t *left = root->left;
        node_t *right = root->right;
        free(root);
        if (right == NULL)
            return left;
        node_t *min = find_min(right);
        min->right = remove_min(right);
        min->left = left;

        return min;
    }

    return root;
}
```

3.3.2 Функции для работы с АВЛ-деревом

Листинг 3.3 содержит реализацию алгоритма построения АВЛ-дерева из дерева двоичного поиска. На листингах 3.4 и 3.5 представлены левый и правый повороты дерева соответственно. Данные функции используются как вспомогательные в функции, реализующей алгоритм балансировки АВЛ-дерева. На листинге 3.6 представлена реализация алгоритма балансировки АВЛ-дерева.

Листинг 3.3 — Реализация алгоритма построения АВЛ-дерева из дерева двоичного поиска

```
1  // dest - avl, source - bst
2  static node_t *__bst_to_avl(node_t *dest, node_t *source)
3  {
4      if (source != NULL)
5      {
6          dest = avl_insert(dest, source->key);
7          dest = __bst_to_avl(dest, source->left);
8          dest = __bst_to_avl(dest, source->right);
9      }
10
11     return dest;
12 }
13
14 node_t *bst_to_avl(node_t *node)
15 {
16     return __bst_to_avl(NULL, node);
17 }
```

Листинг 3.4 — Реализация алгоритма поворота дерева влево

```
1  node_t *rotate_left(node_t *node)
2  {
3      node_t *root = node->right;
4      node->right = root->left;
5      root->left = node;
6
7      fix_height(node);
8      fix_height(root);
9
10     return root;
11 }
```

Листинг 3.5 — Реализация алгоритма поворота дерева вправо

```
node_t *rotate_right(node_t *node)
{
    node_t *root = node->left;
    node->left = root->right;
    root->right = node;

    fix_height(node);
    fix_height(root);

    return root;
}
```

Листинг 3.6 — Реализация алгоритма балансировки АВЛ-дерева

```
node_t *balance(node_t *node)
{
    fix_height(node);

    if (bfactor(node) > 1)
    {
        if (bfactor(node->right) < 0)
            node->right = rotate_right(node->right);
        return rotate_left(node);
    }

    if (bfactor(node) < -1)
    {
        if (bfactor(node->left) > 0)
            node->left = rotate_left(node->left);
        return rotate_right(node);
    }

    return node;
}
```

3.3.3 Функции для работы с хеш-таблицей

На листинге 3.7 представлена реализация алгоритма вставки ключа в хеш-таблицу.

Листинг 3.7 — Реализация алгоритма вставки ключа в хеш-таблицу

```
1  int hash_insert(hash_table_t *table, data_t data)
2  {
3      size_t index = table->func(data, table->size) % table->size;
4      hash_data_t tdata = table->data[index];
5      size_t increments = 0;
6
7      while (tdata.has_value && tdata.key != data && increments < table->size)
8      {
9          index = (index + table->step) % table->size;
10         tdata = table->data[index];
11         increments++;
12     }
13
14     if (tdata.key == data)
15         return SUCCESS;
16
17     if (!tdata.has_value)
18     {
19         tdata.key = data;
20         tdata.has_value = true;
21         tdata.cmp_count = increments + 1;
22         table->data[index] = tdata;
23         return SUCCESS;
24     }
25
26     return MEM_ERR;
27 }
```

Листинг 3.8 содержит реализацию алгоритма удаления ключа из хеш-таблицы.

Листинг 3.8 — Реализация алгоритма удаления ключа из хеш-табллицы

```
1  int hash_remove(hash_table_t *table, data_t data, size_t *cmp_count)
2  {
3      size_t index = table->func(data, table->size) % table->size;
4      hash_data_t tdata = table->data[index];
5      size_t increments = 0;
6
7      while (tdata.key != data && increments < table->size)
8      {
9          index = (index + table->step) % table->size;
10         tdata = table->data[index];
11         increments++;
12     }
13
14     *cmp_count = increments + 1;
15
16     if (tdata.has_value && tdata.key == data)
17     {
18         tdata.has_value = false;
19         tdata.cmp_count = increments + 1;
20         table->data[index] = tdata;
21         return SUCCESS;
22     }
23
24     return NO_ELEM;
25 }
```

3.4 Тестовые данные

Ниже, в таблице 3.1, представлены функциональные тесты к разрабатываемому ПО.

Таблица 3.1 — Функциональные тесты.

Название теста	**входные данные**	**фактический результат**	**ожидаемый результат**
Неверное имя файла	^Z	сообщение о неверном вводе	сообщение о неверном вводе
Пустой входной файл	*«пустой файл»*	сообщение о неверном вводе	сообщение о неверном вводе
Вставка некорректного символа	1 2 f 4	сообщение о неверном вводе	сообщение о неверном вводе
удаление несуществующего ключа	4 13 21 *ключ:* 40	сообщение: ключ не существует	сообщение: ключ не существует
Вставка ключа в заполненную хеш-таблицу	1 2 ... 49 50 *размер х.т.:* 10	сообщение: таблица переполнена	сообщение: таблица переполнена

Все тесты пройдены успешно.

4 Экспериментальный раздел

В данном разделе представлены результаты сравнения эффективности по времени реализованного алгоритма, при использования различных структур данных, таких как двоичное дерево поиска, АВЛ-дерево, хеш-таблица и файл.

4.1 Постановка эксперимента

В данном эксперименте проводится сравнительный анализ временной эффективности и эффективности по памяти при удалении элемента из ДДП, АВЛ-дерева, хеш-таблицы и файла.

4.1.1 Технические характеристики

- операционная система: Fedora Workstation 35 Linux 64-bit;
- оперативная память: 8 GB;
- Intel Core i5-7300HQ @ 4x 3.5GHz.

4.1.2 Описание экспериментальных данных

Тестирование произведено на основе хеш-таблицы, максимальный размер которой 50 элементов. Для тестирования временной эффективности будет произведено 10 замеров удаления, после чего результат будет усреднен.

4.2 Результат эксперимента

В результате подсчета эффективности по времени и по памяти для удаления из четырех структур данных, были получены следующие данные:

4.2.1 Время работы

Таблица 4.1 — Время удаления

Структура данных	Время, тики	Среднее число сравнений
Двоичное дерево поиска	5864	4.7
АВЛ-дерево	1540	2.9
Хеш-таблица	1216	1.1
Файл	258382	5.5

4.2.2 Объем занимаемой памяти

Таблица 4.2 — Время удаления

Структура данных	Память, байт
Двоичное дерево поиска	240
АВЛ-дерево	240
Хеш-таблица	832
Файл	36

4.3 Вывод

В результате эксперимента можно сделать вывод, что использование файла для хранения и удаления элементов наиболее выгодная по памяти, но значительно проигрывает по времени выполнения удаления, в сравнении с другими структурами данных. Использование файла обосновано в том случае, когда объем хранимых данных значителен и не особо важна эффективность по времени.

Использование двоичных деревьев поиска позволяет сократить время удаления примерно в 40 раз, и уменьшить среднее число сравнений на 14 %. Справедливо отметить, что использование сбалансированного дерева (в данном случае АВЛ-дерева) эффективнее по времени удаления примерно в 4 раза, по сравнению с ДДП, причем среднее количество сравнений примерно в 1.5 раза меньше. Использование деревьев является эффективным и по времени, и по памяти.

Наиболее эффективной по времени оказалась хеш-таблица, среднее число сравнений в которой, в лучшем случае, может не превышать 1. Но хеш-таблица оказалась наиболее неэффективной по памяти. Использование хеш-таблицы выгодно в том случае, когда необходимо наименьшее время вставки/поиска/удаления, при этом без особых ограничений по памяти.

5 Контрольные вопросы

1) Что такое дерево?

Дерево — рекурсивная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим». Эта структура описывается рекурентно как узел, у которого есть указатели на другие узлы (поддеревья).

2) Как выделяется память под представление деревьев?

Память для представления в виде связного списка выделяется динамически в момент добавления новых ключей.

3) Какие стандартные операции возможны над деревьями?

Стандартные операции над деревьями включают в себя вставку узла в дерево, поиск узла, балансировка дерева. Также возможно отделить поддерево в отдельное дерево.

4) Что такое дерево двоичного поиска?

Дерево двоичного поиска - это дерево, в котором для каждого узла задано отношение порядка таким образом, что этот узел меньше одного своего поддерева, но больше другого поддерева.

5) Чем отличается идеально сбалансированное дерево от АВЛ-дерева?

Идеально сбалансированное дерево определяется как дерево двоичного поиска, в котором у каждого узла количество узлов в обоих его поддеревьях отличается не более чем на единицу. В АВЛ деревьях это требование ослаблено. В них у каждого узла высоты обоих его поддеревьев отличаются не более чем на единицу.

6) Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?

Поиск в сбалансированном дереве зачастую происходит быстрее, так как высота несбалансированного дерева как правило превосходит высоту того же сбалансированного дерева.

7) Что такое хеш-таблица, каков принцип ее построения?

Хеш-таблица это структура для данных с произвольным доступом к ним. Принцип построения хеш-таблицы основан на особой функции, называемой хеш-функцией, которая сопоставляет уникальный ключ с его местом в таблице. Идеальная хеш-функция - это инъекция множества ключей во множество мест в таблице.

8) Что такое коллизии? Каковы методы их устранения.

Коллизии — это ситуации, когда для разных ключей выбранная хеш-функция возвращает одно и то же значение.

Коллизии могут возникать на этапе "упаковки"расчитанного большого хеша в размерность таблицы. То есть хеш-значения разных ключей могут быть разными, но при упаковке они получают одно и то же место в таблице. Такого рода коллизии могут быть устранены изменением размерности таблицы.

9) В каком случае поиск в хеш-таблицах становится неэффективен?

Поиск в хеш-таблице может становиться неэффективным в случаях большого числа коллизий, из-за которых нужно будет производить дополнительный последовательный поиск по ключам, имеющим одинаковый хеш.

10) Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах. В хеш-таблице минимальное время поиска $O(1)$. В АВЛ: $O(log_2 n)$. В дереве двоичного поиска $O(h)$, где h - высота дерева (от $log_2 n$ до n).

Заключение

Решение о выборе той или иной структуры данных для произведения операций над данными сильно зависит от поставленных задач. В рамках данной лабораторной работы была выполнена операция удаления элемента из файла, ДДП, АВЛ-дерева и хеш-таблицы.

Самым эффективным по памяти оказался файл, однако его использование выгодно лишь в том случае, когда время выполнения операции не играет такой важной роли, как объем памяти.

Использование деревьев: двоичного дерева поиска и АВЛ-дерева, обеспечивает баланс между временем произведения операций и объемом хранимых данных. В случае с АВЛ-деревом наиболее заметен прирост производительности, но необходимо сохранять сбалансированность дерева для получения столь эффективного по времени результата.

Самой эффективной по времени структурой данных оказалась хеш-таблица. При подходящем выборе максимального размера хеш таблицы и хеш-функции, можно получить среднее количество сравнений немногим превосходящее 1, а в лучшем случае равным 1. Но для хранения хеш-таблицы потребуется в 3-4 раза больше памяти, чем для хранения деревьев.

Список использованных источников

1. Сергеев М.И., Янишевская А.Г. АВЛ-деревья, выполнение операций над ними // ИВД. 2016. №2 (41). — (дата обращения: 04.12.2021). https://cyberleninka.ru/article/n/avl-derevya-vypolnenie-operatsiy-nad-nimi.

2. *Вирт, Н.* Алгоритмы и структуры данных / Н. Вирт. — ДМК Пресс, 2010. — С. 272.

3. Исканцев Н. В. Математическая теория стойкости хеш-функций к коллизиям // Наука и современность. 2012. №16-2. — (дата обращения: 04.12.2021). https://cyberleninka.ru/article/n/matematicheskaya-teoriya-stoykosti-hesh-funktsiy-k-kolliziyam.

4. GCC, the GNU Compiler Collection. — (дата обращения: 04.12.2021). https://gcc.gnu.org/.

5. GNU Make. — (дата обращения: 04.12.2021). https://www.gnu.org/software/make/.