|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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 №6 «ДЕРЕВЬЯ, ХЕШ – ТАБЛИЦЫ»**

Студент Иванов Павел Александрович

Группа ИУ7 – 35Б

Вариант №6

Проверил Барышникова М.Ю.

**Оглавление**

[Описание условия задачи 2](#_Toc89336767)

[Описание технического задания 2](#_Toc89336768)

[Описание структуры данных 5](#_Toc89336769)

[Основные функции 6](#_Toc89336770)

[Описание алгоритма 8](#_Toc89336771)

[Набор тестов 9](#_Toc89336772)

[Сравнение эффективности 12](#_Toc89336773)

[Ответы на контрольные вопросы 14](#_Toc89336774)

[Вывод 15](#_Toc89336775)

# **Описание условия задачи**

Построить ДДП, сбалансированное двоичное дерево (АВЛ) и хеш- таблицу по указанным данным. Сравнить эффективность поиска в ДДП в АВЛ дереве и в хеш-таблице (используя открытую или закрытую адресацию) и в файле. Вывести на экран деревья и хеш-таблицу. 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. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий.

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 метод цепочек для устранения коллизий. Осуществить поиск введенного целого числа в ДДП, в сбалансированном дереве, в хеш-таблице и в файле. Сравнить время поиска, объем памяти и количество сравнений при использовании различных (4-х) структур данных. Если количество сравнений в хеш-таблице больше указанного (вводить), то произвести реструктуризацию таблицы, выбрав другую функцию.

# **Описание технического задания**

Входные данные:

*Целое число*: номер команды меню, число для поиска в структурах, число для добавления в структуру или удаления из неё, максимальное число сравнений в хеш-таблице.

*Строка*: имя файла.

Выходные данные:

*Изображение*: граф в формате PNG.

*Целое число*: элементы хэш-таблицы, число сравнений, объём памяти в байтах, число тактов, процент коллизий.

Действие программы:

Чтение файла целых чисел в следующие структуры: дерево двоичного поиска (далее — ДДП), АВЛ-дерево, хэш-таблицу. С этими структурами производятся операции: создание структуры, добавление элемента, удаление элемента, просмотр в графическом (для деревьев) и текстовом (для хеш-таблицы) форматах, реструктуризация при слишком большом числе сравнений (для хеш-таблицы). Измерение времени поиска во всех структурах и в файле, объема затраченной памяти и эффективности поиска при различном проценте коллизий в хеш-таблице.

Обращение к программе:

Программа открывается через команду ./app.exe. Если исполняемый файл отсутствует, вызвать make из папки проекта.

Аварийные ситуации:

1. Отсутствует указанный файл.

Сообщение: “No file!”

1. Построение дерева из пустого / несуществующего файла.

Сообщение: “No file!”

1. Добавление элемента в пустое дерево (и ДДП, и АВЛ).

Сообщение: “No tree!”

1. Удаление элемента из пустого дерева (и ДДП, и АВЛ).

Сообщение: “No tree!”

1. Просмотр несуществующего дерева (и ДДП, и АВЛ).

Сообщение: “No tree!”

1. Построение АВЛ-дерева из несуществующего ДДП.

Сообщение: “Error!”

1. Ввод ненатурального максимального числа сравнений.

Сообщение: “Wrong input!”

1. Неправильный ввод команды.

Сообщение: “Wrong input!”

1. Неправильный ввод числа для поиска или удаления.

Сообщение: “Wrong input!”

1. Поиск в не построенном дереве.

Сообщение: “No tree!”

1. Поиск в не построенной хеш-таблице.

Сообщение: “No table!”

1. Добавление в таблицу / удаление из хеш-таблицы неверно введенного элемента.

Сообщение: “Wrong input!”

1. Поиск в несуществующем файле.

Сообщение: “No file!”

**Меню программы:**

1 – Ввод имени файла

2 – Построение ДДП

3 – Добавление элемента в ДДП

4 – Удаление элемента из ДДП

5 – Просмотр ДДП

6 – Построение АВЛ-дерева из ДДП

7 – Добавление элемента в АВЛ-дерево

8 – Удаление элемента из АВЛ-дерева

9 – Просмотр АВЛ-дерева

10 – Ввод максимального числа сравнений в хеш-таблице

11 – Построение хеш-таблицы на основе файла

12 – Просмотр хеш-таблицы

13 – Добавление элемента в хеш-таблицу

14 – Удаление элемента из хеш-таблицы

15 – Поиск данных в ДДП

16 – Поиск данных в АВЛ-дереве

17 – Поиск данных в хеш-таблице

18 – Поиск данных в файле

19 – Вывод сравнение эффективности структур

0 – Выход

# **Описание структуры данных**

Структура данных для ДДП:

typedef struct bst\_node\_t bst\_node\_t;

struct bst\_node\_t

{

    int num;

    bst\_node\_t \*left;

    bst\_node\_t \*right;

};

ДДП хранится как указатель на голову дерева. В каждом узле дерева хранятся данные (целое число) и два указателя: на левого и правого сына данной вершины.

Структура для хранения АВЛ-дерева:

typedef struct avl\_node\_t avl\_node\_t;

struct avl\_node\_t

{

    int num;

    avl\_node\_t \*left;

    avl\_node\_t \*right;

    int height;

};

АВЛ-дерево хранится как указатель на голову дерева. В каждом узле дерева хранятся данные (целое число), указатели на правого и левого сыновей, а также высота данной вершины. Высота хранится для того, чтобы постоянно поддерживать дерево сбалансированным.

Хеш-таблица представляет собой структуру, в которой хранится число элементов и массив указателей на начала списков цепочек элементов, которые хеш-функция определила к конкретной позиции.

typedef struct hash\_t hash\_t;

typedef struct node\_t node\_t;

struct hash\_t

{

    int size;

    node\_t \*\*array;

};

Нода списка задаётся обычной структурой:

struct node\_t

{

    int data;

    node\_t \*next;

};

Сама хеш-функция задается как остаток деления на размер массива:

int hash\_function(int key, int m)

{

    return (abs(key) % m);

}

# **Основные функции**

// Функция добавления вершины в ДДП

bst\_node\_t \*bst\_add\_node(bst\_node\_t \*head, bst\_node\_t \*node);

// Функция поиска элемента в ДДП

bst\_node\_t \*bst\_search\_node(bst\_node\_t \*head, int num, int \*comp\_number);

// Функция обхода ДДП

void bst\_traverse(bst\_node\_t \*tree, void (\*f)(bst\_node\_t\*, void\*), void \*arg, char \*order);

// Функция удаление элемента из ДДП

bst\_node\_t \*bst\_delete\_node(bst\_node\_t \*head, int num);

// Функция очистки ноды ДДП

void bst\_free\_node(bst\_node\_t \*node, void \*arg);

// Функция освобождения памяти из под ДДП

void bst\_free\_tree(bst\_node\_t \*head);

// Функция создания ячейки АВЛ-дерева

avl\_node\_t \*avl\_create\_node(int num);

// Функция добавления элемента в АВЛ-дерево

avl\_node\_t \*avl\_add\_node(avl\_node\_t \*head, avl\_node\_t \*node);

// Функция поиска ноды в АВЛ-дереве

avl\_node\_t \*avl\_search\_node(avl\_node\_t \*head, int num, int \*comp\_number);

// Функция левого-левого поворота

avl\_node\_t \*avl\_ll\_rotation(avl\_node\_t \*tree);

// Функция правого-правого поворота

avl\_node\_t \*avl\_rr\_rotation(avl\_node\_t \*tree);

// Функция левого-правого поворота

avl\_node\_t \*avl\_lr\_rotation(avl\_node\_t \*tree);

// Функция правого-левого поворота

avl\_node\_t \*avl\_rl\_rotation(avl\_node\_t \*tree);

// Функция обхода АВЛ-дерева

void avl\_traverse(avl\_node\_t \*tree, void (\*f)(avl\_node\_t\*, void\*), void \*arg, char \*order);

// Функция удаления ноды АВЛ-дерева

avl\_node\_t \*avl\_delete\_node(avl\_node\_t \*head, int num);

// Функция, определяющая балансировку в вершине head

int get\_balance(avl\_node\_t \*head);

// Функция создания АВЛ-дерева из ДДП

int avl\_create\_from\_bst(bst\_node\_t \*head, avl\_node\_t \*\*src);

// Функция освобождения памяти из-под вершины АВЛ-дерева

void avl\_free\_node(avl\_node\_t \*node, void \*arg);

// Функция очистки АВЛ-дерева

void avl\_free\_tree(avl\_node\_t \*head);

// Функция создания хеш-таблицы

hash\_t \*hash\_create(int size);

// Функция освобвождения памяти из под хеш-таблицы

void hash\_free(hash\_t \*table);

// Функция добавления элемента в хеш-таблицу

void hash\_add(hash\_t \*table, int num);

// Функция удаления элемента из хеш-таблицы

void hash\_delete(hash\_t \*table, int num);

// Функция реструктуризации хеш-таблицы

void hash\_restructure(hash\_t \*\*table);

// Функция поиска в хеш-таблице

int hash\_search(hash\_t \*table, int num, int \*comp\_number);

// Функция просмотра хеш-таблицы

void hash\_read(hash\_t \*table);

// Функция чтения списка в хеш-таблицу

int read\_file\_to\_hash(hash\_t \*\*table, FILE \*f);

// Функция подсчёта числа коллизий в хеш-таблице

int hash\_count\_collisions(hash\_t \*table);

// Функция чтения файла в структуру ДДП

int read\_file\_to\_bst(bst\_node\_t \*\*bst, FILE \*f);

# **Описание алгоритма**

*Добавление в дерево:*

Сравнение числа с номером в вершине. Если число больше числа в вершине, вызывается та же функция поиска из правого сына корня, если меньше — из левого. Если вызываемый сын NULL – возвращаем ноду со вставляемым элементом. При выходе из рекурсии корни поддеревьев переприсваиваются. Таким образом, тот самый нулевой сын теперь будет ссылаться на вставленный элемент. Если это АВЛ-дерево, требуется также балансировка по выходе из рекурсии. Если перевешивает левая ветвь и число добавили в левое поддерево левого сына данной вершины, совершается LL-поворот. Аналогично определяются и выполняются RR, LR и RL повороты. Повороты производятся сменой указателей. Высота при это добавляется при каждом добавлении и удалении рекурсивно, как максимум из высот сыновей + 1.

*Удаление из дерева:*

Углубляемся рекурсивно вглубь дерева аналогично добавлению. Если вершина на очередном вызове имеет заданное число, то её необходимо удалить. Здесь возможно нескольких ситуаций:

1. У вершины нет сыновей: тогда возвращаем NULL
2. У вершины только один сын: тогда возвращаем указатель на него
3. У вершины оба сына: тогда меняем вершину на самый правый элемент в левом поддереве

В случае, если это АВЛ-дерево, также производится балансировка, аналогичная добавлению.

Поиск реализуется точно так же, без балансировки: постоянное углубление внутрь дерева.

Хеш-таблица строится как массив из простого числа элементов, наиболее близкого к и превосходящего 0,72 \* (объём данных).

*Добавление в хеш-таблицу:*

Вычисление позиции элемента по хеш-функции. Если она пустая и никуда не указывает, создаем ячейку списка – теперь она начинает список цепочек. Если на эту позицию снова попадает какой-то элемент, то проходим до конца списка цепочек и вставляем в конец.

*Удаление из хеш-таблицы:*

Вычисление позиции элемента по хеш-функции. Если она пустая и никуда не указывает, то элемента даже нет. Иначе, идём по списку цепочек и затираем ноду списка, в которой найден необходимый элемент.

Поиск происходит аналогично удалению, только без процедуры затирания ноды списка.

Поиск в файле происходит проходом с начала списка и сравнением требуемого числа со всеми элементами файла.

Реструктуризация таблицы производится выбором другой хеш-функции, то есть увеличением размера массива до тех пор, пока количество коллизий не уменьшится. Поиск идет до тех пор, пока не будет найдено первое простое число, превышающее изначальное, при котором число коллизий меньше заданного.

# **Набор тестов**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тестовый случай | Ввод | Результат |
| 1 | Неверный ввод команды | -1 | Wrong input! |
| 2 | Неверный ввод команды | a | Wrong input! |
| 3 | Отсутствует указанный файл. | 1 | No file! |
| 4 | Построение дерева из пустого / несуществующего файла. | 2 | No file! |
| 5 | Добавление элемента в пустое дерево (и ДДП, и АВЛ). | 3 / 7 | No tree! |
| 6 | Просмотр несуществующего дерева (и ДДП, и АВЛ). | 5 / 9 | No tree! |
| 7 | Построение АВЛ-дерева из несуществующего ДДП. | 6 | No tree! |
| 8 | Ввод ненатурального максимального числа сравнений | 10, -2 | Wrong input! |
| 9 | Неправильный ввод числа для поиска или удаления. | aaaa | Wrong input! |
| 10 | Поиск в не построенном дереве. | 15 / 16 | No tree! |
| 11 | Поиск в не построенной хеш-таблице. | 17 | No table! |
| 12 | Добавление в таблицу / удаление из хеш-таблицы неверно введенного элемента. | 13 / 14, aaa | Wrong input! |
| 13 | Поиск в несуществующем файле. | 18 | No file! |
| 14 | Успешный ввод файла | 1 | - |
| 15 | Построение ДДП по файлу | 2 | Success! |
| 16 | Добавление элемента в ДДП | 3, -5 | Success! |
| 17 | Удаление элемента из ДДП | 4, -5 | Success! |
| 18 | Просмотр ДДП | 5 | Открылась картинка, визуализирующая граф |
| 19 | Построение АВЛ-дерева по файлу | 6 | Success! |
| 20 | Добавление элемента в ДДП | 7, -5 | Success! |
| 21 | Удаление элемента из ДДП | 8, -5 | Success! |
| 22 | Просмотр ДДП | 9 | Открылась картинка, визуализирующая граф |
| 23 | Ввод максимального числа сравнений | 10, 111 | - |
| 24 | Построение хеш-таблицы | 11 | Success! |
| 25 | Просмотр хеш-таблицы | 12 | Вертикально выведен массив, каждый элемент которого список |
| 26 | Добавление элемента в хеш-таблицу | 13, -5 | Success! |
| 27 | Удаление элемента из хэш-таблицы | 14, -5 | Success! |
| 28 | Поиск данных в ДДП | 15, -5 | Found! (Not found!) |
| 29 | Поиск данных в АВЛ-дереве | 16, -5 | Found! (Not found!) |
| 30 | Поиск данных в хэш-таблице | 17, -5 | Found! (Not found!) |
| 28 | Поиск данных в файле | 18, -5 | Found! (Not found!) |
| 29 | Вывод информации по сравнению эффективности | 19 | Выведены сравнительные таблицы времени поиска, занимаемой памяти |

# **Сравнение эффективности**

По заданию требуется сравнить время поиска в структурах. Я получил следующие данные. Время измерялось в тактах процессора, «сравн.» - среднее число сравнений для поиска в структуре при данном количестве.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ДДП | | АВЛ-дерево | | Хеш-таблица\* | | Файл | |
| Кол-во | Время | Сравн. | Время | Сравн. | Время | Сравн. | Время | Сравн. |
| 10 | 42 | 3 | 26 | 3 | 22 | 1 | 5401 | 5 |
| 50 | 44 | 6 | 34 | 4 | 19 | 1 | 7973 | 25 |
| 100 | 51 | 8 | 39 | 5 | 24 | 1 | 14110 | 50 |
| 500 | 113 | 11 | 88 | 8 | 24 | 1 | 68274 | 250 |
| 1000 | 151 | 12 | 126 | 9 | 25 | 1 | 139801 | 500 |

\* - рассматривается таблица, в которой процент коллизий не превышает 10%

Оценка памяти (в байтах):

|  |  |  |  |
| --- | --- | --- | --- |
| Количество | ДДП | АВЛ-дерево | Хеш-таблица\* |
| 10 | 240 | 320 | 504 |
| 50 | 1200 | 1600 | 3256 |
| 100 | 2400 | 3200 | 5432 |
| 500 | 12000 | 16000 | 27496 |
| 1000 | 24000 | 32000 | 48392 |

\* - в данных хеш-таблицах процент коллизий не превышает 10%

По количеству сравнений по временной характеристике видны некоторые закономерности. Число сравнений в АВЛ-дереве, как правило, логарифм из числа ключей. В ДДП ситуация действительно хуже, чем в АВЛ-дереве: там требуется больше сравнений и больше времени. Объясняется это несбалансированностью дерева: из-за этого поиск может «затянуться». Что касается хэш-таблицы (практически) без коллизий, то время поиска – константное, что и должно было получится. Хуже всего ситуация с файлом, в котором отсутствует произвольный доступ — поэтому число сравнений и поиска большое.

Чтобы сделать выводы по памяти, рассмотрим производительность поиска в хеш-таблице при разном проценте коллизий:

|  |  |  |  |
| --- | --- | --- | --- |
| Процент | Время, такты | Среднее число сравнений | Память, байт |
| 45% | 42 | 1 | 21816 |
| 39% | 38 | 1 | 22824 |
| 35% | 35 | 1 | 23976 |
| 32% | 33 | 1 | 25304 |
| 26% | 36 | 1 | 26888 |
| 23% | 33 | 1 | 28776 |
| 22% | 33 | 1 | 30936 |
| 17% | 32 | 1 | 33432 |
| 14% | 31 | 1 | 36344 |
| 12% | 28 | 1 | 39752 |
| 8% | 30 | 1 | 43704 |
| 7% | 28 | 1 | 48392 |
| 6% | 25 | 1 | 53784 |
| 4% | 26 | 1 | 60152 |
| 3% | 21 | 1 | 67592 |
| 2% | 25 | 1 | 76184 |

По результатам измерений можно наблюдать интересную зависимость: при уменьшении процента коллизий с 45% до 2% затраты по памяти увеличиваются почти в 4 раза (это объясняется увеличением размеров таблицы, и следственным ему наличием большого числа пустых указателей), но зато затраты по времени – снижаются почти в 2 раза. Этим можно объяснить и смысл реструктуризации хеш-таблицы — кратный прирост производительности по времени.

Тем не менее, даже при большом проценте коллизий (45%) время поиска в таблице, хранящей 1000 элементов, в 3 раза меньше времени поиска тех же самых данных в ДДП или АВЛ-дереве.

Таким образом, сказать, что лучше использовать в любой ситуации — трудно. Наименьшие затраты по памяти вызывает использование файла: в таком случае в памяти данные не хранятся, но это сильно сказывается на времени работы. Наибольшие же затраты по памяти, в среднем, требует хеш-таблица, но, как установили ранее, это помогает ей быть самой быстрой в отношении поиска структурой. Вариант ДДП не требует большого количества памяти, но по времени выигрывает разве что лишь у файла. АВЛ-дерево — средний вариант в отношении быстродействия и занимаемой памяти.

# **Ответы на контрольные вопросы**

***1. Что такое дерево?***

С точки зрения математики, дерево (в случае программирования рассматривается ориентированное) — бесконтурный граф, каждая вершина которого имеет степень полузахода, равную 1, кроме одной — корня, который имеет степень полузахода 0.

Как правило, дерево определяется рекурсивно либо как пустая структура, либо как узел типа дерева с конечным числом древовидных структур этого же типа.

***2. Как выделяется память под представление деревьев?***

Выделение памяти под дерево зависит от способа представления дерева.

а) Если дерево представляется массивом связей с предками, то память выделяется единым блоком под массив: либо на стеке (если изначально известно максимальное число вершин), либо в куче.

б) Если дерево представляется «массивом лидеров», в котором i-ый элемент указывает на всех сыновей узла i, то память под массив выделяется либо на стеке, либо в куче, а далее – память под списки сыновей выделяются динамически.

в) Если дерево определено рекурсивно (см. вопрос 1), то память под него выделяется динамически (при добавлении в дерево выделяется память, указатель на неё записывается в нового родителя вершины).

***3. Какие стандартные операции возможны над деревьями?***

Над деревьями возможны следующие операции: обход дерева, включение элемента, добавление элемента, поиск элемента.

***4. Что такое дерево двоичного поиска?***

Дерево двоичного поиска — бинарное дерево, в котором все правые потомки больше предка, а все левые потомки — меньше. Это свойство выполняется для любого узла дерева.

***5. Чем отличается идеально сбалансированное дерево от АВЛ дерева?***

Идеально сбалансированное дерево — дерево, у которого *число равномерно распределённых вершин* в левом и правом поддеревьях отличается не более чем на единицу.

АВЛ-дерево — дерево, для каждого узла которого верно, что *высота двух его поддеревьев* отличается не более чем на единицу.

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

Высота дерева в ДДП оценивается от log2n до n. В случае АВЛ-дерева же, строго доказано, что высота дерева находится в интервале от log(n + 1) до 1.44 \* log(n + 2) — 0.328 что гораздо меньше, чем худший случай обычного двоичного дерева поиска. Таким образом, в среднем поиск в АВЛ-дереве будет быстрее, чем поиск в ДДП.

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

Массив, заполненный в порядке, определенным хеш-функцией, называется хеш-таблицей. Хеш-функция — функция, ставящая некоторому ключу в соответствие индекс ячейки. По-другому, хеш-функция — отображение строки в число (позже соответствующее индексу в массиве), где под строкой понимается любой объект.

***8. Что такое коллизии? Каковы методы их устранения.***

Коллизия — ситуация «наслаивания» значений в хеш-таблице, когда разным ключам соответствует одно и то же значение хеш-функции. Существует несколько методов борьбы с коллизиями:

Открытое хеширование — так называемый метод цепочек, когда при появлении коллизий элементы выстраиваются в связные списки, каждый из которых соответствует определенному значению хеш-функции.

Закрытое хеширование — полный отказ от ссылок, когда после появления коллизии просматриваются все дальнейшие ячейки.

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

Когда требуется более 3-4 сравнений для поиска элемента, поиск в хеш-таблице становится неэффективным.

10. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах.

Временная эффективность поиска:

|  |  |  |
| --- | --- | --- |
|  | Лучший случай | Худший случай |
| ДДП | O(logn) | O(n) |
| АВЛ-дерево | O(logn) | O(logn) |
| Хеш-таблица | O(1) | O(n) |

# **Вывод**

В ходе данной работы я познакомился с такими структурами данных, как Двоичное Дерево Поиска, АВЛ-дерево, хеш-таблица, а также сравнил эффективность поиска в этих структурах и в файле. Я получил следующие результаты: Сказать, что лучше использовать в любой ситуации — трудно. Наименьшие затраты по памяти вызывает использование файла: в таком случае в дополнительной памяти данные не занимают, но это сильно сказывается на времени работы. Наибольшие же затраты по памяти, в среднем, требует хеш-таблица с маленьким число коллизий, но, как установили ранее, это помогает ей быть самой быстрой в отношении поиска структурой. Вариант ДДП не требует большого количества памяти, но по времени выигрывает разве что лишь у файла. АВЛ-дерево — средний вариант в отношении быстродействия и занимаемой памяти.