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 №07 ПО ДИСЦИПЛИНЕ: ТИПЫ И СТРУКТУРЫ ДАННЫХ

Сбалансированные деревья, хеш–таблицы

Студент **Батуев А.Г.**

Группа **ИУ7-36Б**

Вариант **2**

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

Студент _____ **Батуев А.Г.**

Преподаватель _____ **Никульшина Т.А.**

2024 г.

Условие задачи

Цель работы – освоение работы с хеш-таблицами, сравнение эффективности-поиска в сбалансированных (AVL) деревьях, в двоичных деревьях поиска и в хештаблицах. Сравнение эффективности устранения коллизий при внешнем ивнутреннем хешировании.

Задание по варианту.
Используя предыдущую программу (задача №6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из чисел файла. Реализовать операции добавления и удаления введенного числа во всех структурах. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

Техническое задание

На вход **программа получает**:

- опция из меню (цифра от 0 до 10)
- файл с данными
- данные от пользователя

Программа выводит:

- дерево в формате .dot
- элементы дерева в отсортированном виде
- кол-во узлов на каждом уровне
- хэш-таблицы
- результаты тестирования

Программа должна реализовывать создание, хранение, вывод дерева и хэш-таблиц, а также операции добавления, удаления и поиск элемента из дерева и хэш-таблиц, экспорт дерева в формат dot и тестирование основных функций (добавление, удаление, поиск).

Обращение к программе осуществляется по указанию названия программы (./app.exe).

Необходимо внимательно **обработать возможные аварийные ситуации**, которые могут включать:

- неправильный ввод данных
- переполнение памяти
- удаление несуществующего элемента

Описание внутренних структур данных

vertex_t — это структура, описывающая узел бинарного дерева. Узел содержит:

- value - целое число, представляющее значение узла.
- bal – целое число, представляющий баланс правого и левого поддерева
- height – целое число, представляющее высоты в конкретной вершине
- left - указатель на левый дочерний узел.
- right - указатель на правый дочерний узел.

tree_t — это структура, описывающая бинарное дерево. Она включает:

- root - указатель на корень дерева, который является узлом типа vertex_t.
- size - целое число, определяющее количество узлов в дереве.
- height - целое число, представляющее высоту дерева (длина самого длинного пути от корня до листа).

```
typedef struct node
{
    int value;               // Значение узла
    int bal;                 // Баланс вершины
    int height;              // Высота вершины
    struct node *left;       // Левый потомок
    struct node *right;      // Правый потомок
} vertex_t;

// Структура дерева
typedef struct
{
    vertex_t *root;          // Корень дерева
    int size;                // Количество узлов
    int height;              // Высота дерева
} tree_t;
```

hash_chain — это структура, описывающая хеш-таблицу с использованием закрытой адресации (цепочек). Она включает:

- table — указатель на массив указателей типа node_t*, представляющий хеш-таблицу, где каждый элемент массива является началом связного списка (цепочки).
- size — целое число, определяющее размер хеш-таблицы.

hash_open — это структура, описывающая хеш-таблицу с использованием открытой адресации. Она включает:

- table — указатель на массив указателей типа int*, представляющий хеш-таблицу, где каждый элемент массива может хранить значение ключа.
- Status — статус ячейки (занята, свободна, удалена)
- size — целое число, определяющее размер хеш-таблицы.

node_t — это структура, представляющая узел связного списка для реализации цепочек. Она включает:

- key — целое число, представляющее ключ узла.

- next — указатель на следующий узел типа node_t, что позволяет формировать цепочку.

node_status_t — перечисление статусов:

- FREE — свободна
- OCCUPIED — занята
- DELETED - удалена

```
extern unsigned int TABLE_SIZE_CHAIN;
extern unsigned int TABLE_SIZE_OPEN;
extern unsigned int HASH_PERFOMANCE_LIMIT_OPEN;
extern unsigned int HASH_PERFOMANCE_LIMIT_CHAIN;

// Закрытая адресация (цепочки)
typedef struct hash_node
{
    int key;
    struct hash_node *next;
} node_t;

typedef struct
{
    node_t** table;
    unsigned int size;
} hash_chain;

typedef enum
{
    FREE,
    OCCUPIED,
    DELETED
} node_status_t;

// Открытая адресация
typedef struct
{
    int** table;
    node_status_t *status;
    unsigned int size;
} hash_open;
```

Основные функции программы

Программа реализует следующие основные функции:

- void action(int option, tree_t *tree hash_chain *chain, hash_open *open); Функция меню:
    1. Прочитать элементы из файла
    2. Добавить элемент
    3. Удалить элемент
    4. Найти элемент
    5. Сохранить в формате dot
    6. Обход
    7. Отсортировать
    8. Протестировать
    9. Изменить размер хэш-таблицы

0. Выход

- void write_into_file(int value, char *filename); Записывает число в конец файла.
- tree_t *init_tree(); Создает и инициализирует пустое дерево. Возвращает указатель на структуру дерева.
- void free_tree(tree_t *tree); Освобождает память, выделенную под дерево, включая все его узлы.
- void add_node(tree_t *tree, char *filename); Добавляет узел в дерево с данными, прочитанными из файла.
- void delete_node(tree_t *tree); Удаляет узел из дерева.
- void read_tree(tree_t *tree, char *filename); Читает данные дерева из файла и заполняет дерево.
- void export_tree_to_dot(vertex_t *root, char *filename);
- Экспортирует дерево в файл формата DOT для визуализации с помощью Graphviz.
- void travel(tree_t *tree); Постфиксный обход дерева
- void find(vertex_t *root, int value, int steps); Ищет значение в дереве, начиная с указанного узла root. Также подсчитывает количество шагов, сделанных для поиска.
- void sort(tree_t *tree); Выводит элементы дерева в отсортированном виде
- tree_t *balance_tree(tree_t *tree); Балансирует дерево
- void init_open(hash_open* ht, unsigned int size); Инициализация хэш-таблицы
- int insert_open(hash_open* ht, int key); Добавление в хэш-таблицу
- int search_open(hash_open* ht, int key); Поиск в хэш-таблице
- int delete_open(hash_open* ht, int key, char verbose); Удаление из хэш-таблицы
- void resize_open(hash_open* ht, unsigned int new_size); Реформирование хэщ-таблицы

Описание основных алгоритмов дерева

Алгоритм добавления узла.

- Если дерево пустое, создается корень.
- Если значение меньше текущего узла, рекурсивно добавляется в левое поддерево.
- Если значение больше текущего узла, рекурсивно добавляется в правое поддерево.

Алгоритм удаления узла.

- Если значение узла меньше текущего узла, то продолжается рекурсивный поиск в левом поддереве.
- Если значение узла больше текущего узла, то продолжается рекурсивный поиск в правом поддереве.

Если узел найден:

- Узел-лист удаляется.
- Узел с одним потомком заменяется на своего потомка.
- Узел с двумя потомками заменяется минимальным узлом из правого поддерева.

Алгоритм поиска узла.
Аналогичен алгоритму удаления, но без удаления.

Алгоритм сортировки узла.
Выполняется симметричный (in-order) обход дерева:

- Рекурсивный обход левого поддерева.
- Вывод значения текущего узла.
- Рекурсивный обход правого поддерева.

Описание основных алгоритмов хэш-таблиц

Открытая адресация

Алгоритм добавления:

1. Вычисляется хэш ключа.
2. Если ячейка по пуста (стоит статус FREE или OCCUPIED), значение вставляется.
3. Если ячейка занята, происходит линейное пробирование — индекс увеличивается до нахождения пустой ячейки.
    a. Если таблица заполнена, вставка невозможна.

Алгоритм удаления:

1. Вычисляется хэш ключа.
2. Линейным пробированием проверяются ячейки до нахождения ключа.
3. Ключ удаляется, ячейка помечается как пустая.
    i. Если ключ не найден, удаление невозможно.

Алгоритм поиска:

1. Вычисляется хэш ключа.
2. Линейным пробированием проверяются ячейки.
3. Если ключ найден, возвращается число проверок.
    a. Если достигнут пустой индекс или начальный индекс, ключ отсутствует.

Закрытая (цепочная) адресация

Алгоритм добавления:

1. Вычисляется хэш ключа.
2. Если в индексе нет цепочки, создаётся новый узел.
3. Если цепочка есть, новый узел добавляется в конец.

Алгоритм удаления:

1. Вычисляется хэш ключа.
2. В цепочке по индексу выполняется поиск узла.
3. Если узел найден, он удаляется из цепочки. Если цепочка пуста, ключ отсутствует.

Алгоритм поиска:

1. Вычисляется хэш ключа.
2. В цепочке по индексу выполняется последовательный поиск узла.
3. Если узел найден, возвращается число проверок, иначе ключ отсутствует.

Набор тестов

Дерево.

Функция добавления

Тестовый сценарий	**Входные данные**	**Ожидаемый результат**
Добавление узла в пустое дерево	tree = NULL, value = 10	Узел с value = 10 становится корнем дерева.
Добавление узла, меньшего корня	tree.root = 10, value = 5	Узел с value = 5 добавлен как левый потомок корня.
Добавление узла, большего корня	tree.root = 10, value = 15	Узел с value = 15 добавлен как правый потомок корня.
Добавление узла, меньшего узла уровня 2	Дерево: 10 → 5, value = 2	Узел с value = 2 добавлен как левый потомок узла 5.
Добавление узла, большего узла уровня 2	Дерево: 10 → 15, value = 20	Узел с value = 20 добавлен как правый потомок узла 15.

Функция удаления

Тестовый сценарий	**Входные данные**	**Ожидаемый результат**
Удаление листа	Дерево: 10 → 5, удаляется value = 5	Узел с value = 5 удален, узел 10 остается корнем.
Удаление узла с одним потомком (слева)	Дерево: 10 → 5 → 3, удаляется value = 5	Узел с value = 5 удален, узел 3 становится левым потомком корня.
Удаление узла с одним потомком (справа)	Дерево: 10 → 15 → 20, удаляется value = 15	Узел с value = 15 удален, узел 20 становится правым потомком корня.
Удаление узла с двумя потомками	Дерево: 10 → 5, 15, удаляется value = 10	Узел с value = 10 заменяется минимальным элементом правого поддерева (15), узел 15 удален.
Удаление отсутствующего узла	Дерево: 10 → 5, 15, удаляется value = 8	Дерево остается неизменным.

Функция поиска

Тестовый сценарий	**Входные данные**	**Ожидаемый результат**
Поиск значения в корне	Дерево: 10 → 5, 15, value = 10	Узел с value = 10 найден за 1 шаг.
Поиск значения в левом поддереве	Дерево: 10 → 5, 15, value = 5	Узел с value = 5 найден за 2 шага.
Поиск значения в правом	Дерево: 10 → 5,	Узел с value = 15 найден за 2

поддереве	15, value = 15	шага.
Поиск отсутствующего значения	Дерево: 10 → 5, 15, value = 8	Узел с value = 8 не найден, поиск завершен за 2 шага.
Поиск в пустом дереве	tree = NULL, value = 10	Узел с value = 10 не найден.

Хэш-таблица

Открытая адресация

Тестовый сценарий	**Входные данные**	**Ожидаемый результат**
Добавление нового ключа	Ключ: 10, Таблица: пустая	Ключ 10 добавлен, возврат числа проверок 1.
Добавление при коллизии	Ключи: 10, 20, Таблица пустая	Оба ключа добавлены, коллизия разрешена, проверки: 1, 2.
Добавление в заполненную таблицу	Ключи: все ячейки заняты	Сообщение об ошибке, ключ не добавлен.
Удаление существующего ключа	Ключ: 10, Таблица: содержит 10	Ключ 10 удалён, возврат числа проверок 1.
Удаление несуществующего ключа	Ключ: 30, Таблица: пустая	Сообщение об отсутствии ключа, возврат -1.
Удаление ключа после пробирования	Ключ: 20, Таблица: 10 -> 20	Ключ 20 удалён, возврат числа проверок 2.
Поиск существующего ключа	Ключ: 10, Таблица: содержит 10	Ключ найден, возврат числа проверок 1.
Поиск ключа после пробирования	Ключ: 20, Таблица: 10 -> 20	Ключ найден, возврат числа проверок 2.
Поиск несуществующего ключа	Ключ: 30, Таблица: пустая	Ключ не найден, возврат -1.

Закрытая адресация

Тестовый сценарий	**Входные данные**	**Ожидаемый результат**
Добавление нового ключа	Ключ: 10, Таблица: пустая	Ключ 10 добавлен в цепочку, возврат 1.
Добавление при коллизии	Ключи: 10, 20, Таблица пустая	Оба ключа добавлены в одну цепочку, проверки: 1, 2.
Удаление существующего ключа	Ключ: 10, Таблица: содержит 10	Ключ 10 удалён, возврат числа проверок 1.

Удаление последнего узла в цепочке	Ключ: 20, Таблица: 10 -> 20	Ключ 20 удалён, возврат числа проверок 2.
Удаление из пустой цепочки	Ключ: 30, Таблица: пустая	Сообщение об отсутствии ключа, возврат -1.
Удаление ключа из середины цепочки	Ключ: 15, Таблица: 10 -> 15 -> 20	Ключ 15 удалён, цепочка корректна, проверки 2.
Поиск существующего ключа	Ключ: 10, Таблица: содержит 10	Ключ найден, возврат числа проверок 1.
Поиск ключа в середине цепочки	Ключ: 15, Таблица: 10 -> 15 -> 20	Ключ найден, возврат числа проверок 2.
Поиск несуществующего ключа	Ключ: 30, Таблица: пустая	Ключ не найден, возврат -1.

Аналитическая часть

Оборудование.
Все замеры происходили не ноутбуке с подключенным питанием и в режиме высокой производительности.
Характеристики ноутбука:

- ОС: Ubuntu LTS 22.04
- Оперативная память: 16 Гб DDR4
- Процессор: Intel core i5-12500H

Результаты исследования.

В представленном тестировании оценивалась производительность бинарного дерева, вырожденного дерева, сбалансированного дерева, хэш-таблицы с закрытой и открытой адресаицей. Были протестированы операции вставки, поиска, удаления. Ниже приведен анализ результатов.

Размерность 500

Структура данных	**Высота**	**Среднее время добавления (нс)**	**Среднее время удаления (нс)**	**Среднее время поиска (нс)**	**Занимаемая память (байт)**
Вырожденное дерево	500	472699721.50	1818624.00	3186.00	16000
Сбалансированное дерево	11	31616498.50	12178.30	187.20	16000
Бинарное дерево	17	4197767.60	11921.90	190.40	16000
Цепочная хэш-таблица	-	94299.10	70.40	46.50	10536
Открытая хэш-таблица	-	300968.60	54.40	55.70	12312

317 - ограничение цепочной хеш-таблицы
1297 - ограничение открытой хеш-таблицы

Размерность 750

Структура данных	**Высота**	**Среднее время добавления (нс)**	**Среднее время удаления (нс)**	**Среднее время поиска (нс)**	**Занимаемая память (байт)**
Вырожденное дерево	750	1615361804.30	4592858.20	3592.90	24000

Сбалансированное дерево	11	77575338.30	15165.80	172.70	24000
Бинарное дерево	21	10344042.40	19876.50	171.00	24000
Цепочная хэш-таблица	-	148770.50	61.50	50.30	15592
Открытая хэш-таблица	-	596612.90	52.40	40.90	19552

449 - ограничение цепочной хеш-таблицы
2069 - ограничение открытой хеш-таблицы

Размерность 1000

Структура данных	**Высота**	**Среднее время добавления (нс)**	**Среднее время удаления (нс)**	**Среднее время поиска (нс)**	**Занимаемая память (байт)**
Вырожденное дерево	1000	3942834432.60	9462844.20	7623.10	32000
Сбалансированное дерево	12	149336742.30	31094.30	165.20	32000
Бинарное дерево	19	16163802.00	22554.70	154.90	32000
Цепочная хэш-таблица	-	295966.30	63.90	41.70	21816
Открытая хэш-таблица	-	1083626.70	61.00	50.20	28088

727 - ограничение цепочной хеш-таблицы
3011 - ограничение открытой хеш-таблицы

Размерность 1250

Структура данных	**Высота**	**Среднее время добавления (нс)**	**Среднее время удаления (нс)**	**Среднее время поиска (нс)**	**Занимаемая память (байт)**
Вырожденное дерево	1250	7710830903.70	16131187.60	7579.90	40000
Сбалансированное дерево	13	242282654.00	28324.40	200.30	40000
Бинарное дерево	21	26411821.50	29490.10	176.60	40000
Цепочная хэш-таблица	-	459864.40	65.30	49.20	27496
Открытая хэш-таблица	-	1509824.70	71.40	59.30	33984

937 - ограничение цепочной хеш-таблицы
3643 - ограничение открытой хеш-таблицы

Размерность 1500

Структура данных	**Высота**	**Среднее время добавления (нс)**	**Среднее время удаления (нс)**	**Среднее время поиска (нс)**	**Занимаемая память (байт)**
Вырожденное дерево	1500	12918852259.40	13838766.00	7725.30	48000
Сбалансированное дерево	13	316923557.20	26657.30	164.90	48000
Бинарное дерево	21	33507391.10	36741.60	157.10	48000
Цепочная хэш-таблица	-	440456.30	63.30	37.00	31864
Открытая хэш-таблица	-	1980634.90	56.80	47.60	42376

983 - ограничение цепочной хеш-таблицы
4597 - ограничение открытой хеш-таблицы

Анализ памяти
Для всех представлений деревьев характерен линейный рост используемой памяти в зависимости от размера данных. Вырожденное дерево потребляет тот же объем памяти, что и сбалансированное или бинарное дерево, поскольку затраты в основном связаны с хранением узлов и их указателей. Хэш-таблицы показывают меньшую зависимость используемой памяти от размера данных благодаря иной организации структуры. Цепочная хэш-таблица при размере 500 занимает 10 536 байт, увеличиваясь до 31 864 байт при размере 1500. Открытая хэш-таблица использует немного больше памяти — 12 312 байт для размера 500 и 42 376 байт для размера 1500. Основной прирост связан с увеличением числа ячеек для разрешения коллизий, однако даже при максимальном размере данные хэш-таблицы остаются менее затратными по памяти по сравнению с деревьями.
Сравнение деревьев и хэш-таблиц показывает, что хэш-таблицы являются более экономичными в плане использования памяти.

Анализ времени
Для деревьев временные характеристики сильно зависят от их структуры. Вырожденное дерево демонстрирует наихудшие показатели из-за максимальной высоты, равной количеству узлов, что приводит к линейному времени выполнения всех операций. В сбалансированных деревьях время операций значительно меньше благодаря минимальной высоте. Бинарные деревья занимают промежуточное положение, демонстрируя приемлемую

производительность, но их временные затраты увеличиваются с ростом данных из-за менее оптимальной высоты по сравнению со сбалансированными деревьями.

Хэш-таблицы, напротив, демонстрируют стабильные и низкие временные затраты на все операции. Открытая хэш-таблица имеет несколько более высокое время добавления, что связано с процессами перераспределения данных и уменьшением числа коллизий. Временные затраты на поиск и удаление в обеих хэш-таблицах остаются близкими к константе и значительно ниже, чем у деревьев.

Сравнение деревьев и хэш-таблиц показывает, что последние значительно превосходят деревья по скорости выполнения операций, особенно при большом объеме данных.

Контрольные вопросы

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

Идеально сбалансированное дерево - дерево, в котором разница высот левого и правого поддеревьев любого узла равна 0.
АВЛ-дерево – дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу.

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

В АВЛ-дереве гарантированно, что поиск одного элемента не займет более чем O(log n) по сложности, а в дереве двоичного поиска в худшем случае это займет O(n), из-за того, что худший случай – односвязный список.

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

Хеш-таблица — структура данных, основанная на хеш-функции, которая преобразует ключи в индексы массива. Принцип: ключ передается через хеш-функцию, которая вычисляет индекс в массиве, где и хранятся данные.

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

Коллизии — ситуации, когда разные ключи приводят к одному и тому же индексу в хеш-таблице. Методы устранения:

- Открытая адресация (поиск следующего свободного места).
- Закрытая адресация (связные списки).

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

Поиск в хеш-таблицах становится неэффективен, если хеш-функция сильно несбалансирована, то есть, если для поиска элемента требуется пройти значительное расстояние по массиву, иными словами в ситуации, когда хэш-таблица начинает вести себя как обычный массив.

6. *Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хеш-таблицах и в файле.*

- АВЛ-деревья: O(log n)
- Дерево двоичного поиска: O(n) в худшем случае, O(log n) в лучшем случае
- Хеш-таблицы: O(1) в среднем случае, O(n) в худшем случае
- Файл: O(n).

Заключение

Анализ временных и пространственных характеристик структур данных (деревьев и хэш-таблиц) показал, что выбор структуры данных должен основываться на характере задачи, объеме данных и требуемой производительности. Деревья и хэш-таблицы предлагают разные преимущества и имеют свои ограничения, которые необходимо учитывать при проектировании систем.

Деревья, несмотря на большую занимаемую память, предоставляют важное преимущество в виде упорядоченности данных, что делает их полезными в задачах, требующих поддержания порядка или выполнения диапазонных запросов.

Хэш-таблицы, напротив, предоставляют минимальные затраты времени на выполнение операций поиска, добавления и удаления благодаря их константной сложности в среднем случае.

Таким образом, в задачах, требующих быстродействия и малых затрат памяти, предпочтение следует отдавать хэш-таблицам. Однако для задач с упорядоченными данными или при необходимости минимизации временных затрат на редкие сложные запросы деревья остаются эффективным решением.