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

Вариант 2

Студент **Абдуллаев Ш. В.**

Группа **ИУ7-34Б**

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

Студент _____ **Абдуллаев Ш. В.**

Преподаватель _____**Силантьева А. В.**

2024

Описание условия задачи

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.

Описание входных данных

Программа принимает на вход текстовый файл, содержащий целочисленные значения. На основе этих данных создаются структуры данных: бинарное дерево, сбалансированное дерево и хеш-таблица. Эти структуры могут служить входными данными для выполнения различных операций. Также программа принимает на вход целочисленное значение для добавления в различные структуры.

Описание исходных данных

Программа представляет собой консольное приложение для работы с бинарным деревом поиска, сбалансированным деревом (AVL), а также хэш-таблицей. Основные возможности работы со структурами включают:

- Вывод элементов: элементы структур можно отобразить в консоли.
- Поиск элементов: пользователь вводит число для поиска. Программа выводит результат поиска.
- Добавление элементов: пользователь вводит число для добавления. Если число уже существует, программа уведомляет об этом.
- Удаление элементов: пользователь вводит число для удаления. Если число отсутствует, программа уведомляет, что удаление невозможно.
- Загрузка данных из файла: чтение чисел из указанного файла в структуру.

Имеются следующие ограничения:

- Ожидаемый формат данных: только целые числа. Ввод некорректных данных (например, букв или символов) завершает программу с сообщением об ошибке.
- Работа с пустыми структурами: некоторые операции (поиск, удаление) недоступны, если структуры пустые.

Допустимый ввод	**Недопустимый ввод**
Выберите команду: 1	Выберите команду: q
Введите число для поиска: 42	Введите число для поиска: abc
Введите число для добавления: 15	Введите число для поиска: 15abc
Введите число для удаления: -8	Введите число для поиска: 7x

Таблица 1. Примеры ввода

Описание результатов

1. Выйти из программы: Завершение работы программы с освобождением всех выделенных ресурсов. После этого программа завершает выполнение.
2. Считать дерево из файла чисел: Открытие указанного файла и чтение чисел для построения обычного двоичного дерева. В случае успеха, дерево загружается и готово к дальнейшим операциям.
3. Вывести дерево: Вывод пользователю текущее состояние двоичного дерева.
4. Проверить наличие числа в дереве: Поиск числа в двоичном дереве. Если число присутствует в дереве, программа выводит сообщение о его нахождении. Если число отсутствует, выводится сообщение об этом.
5. Добавить число в дерево: Добавление нового числа в двоичное дерево. Если число уже существует в дереве, программа выведет сообщение о невозможности добавления.
6. Удалить число из дерева: Удаление числа из дерева. Если число найдено, оно удаляется. Если число не найдено, выводится сообщение о невозможности удаления.

7. Считать сбалансированное дерево из файла чисел: Чтение чисел из файла и создание сбалансированного двоичного дерева.
8. Вывести сбалансированное дерево: Отображение пользователю текущее состояние сбалансированного дерева.
9. Проверить наличие числа в сбалансированном дереве: Поиск числа в сбалансированном дереве. Программа сообщает, найдено ли число, или его отсутствие в дереве.
10. Добавить число в сбалансированное дерево: Вставка нового числа в сбалансированное дерево. Если число уже присутствует в дереве, программа выведет сообщение о невозможности добавления.
11. Удалить число из сбалансированного дерева: Удаление числа из сбалансированного дерева. Если число найдено, оно удаляется. Если число не найдено, программа сообщит о невозможности удаления.
12. Считать хеш-таблицу из файла чисел: Чтение чисел из файла для создания хеш-таблицы.
13. Вывести хеш-таблицу: Вывод текущего состояния хеш-таблицы, где будут показаны элементы и их позиции в таблице.
14. Проверить наличие числа в хеш-таблице: Поиск числа в хеш-таблице. Программа сообщает, если число найдено, или выводит сообщение о его отсутствии.
15. Добавить число в хеш-таблицу: Вставка нового числа в хеш-таблицу. Если число уже существует, программа выведет сообщение о невозможности добавления.
16. Удалить число из хеш-таблицы: Удаление числа из хеш-таблицы. Если число найдено, оно удаляется. Если число не найдено, программа сообщит о невозможности удаления.
17. Сравнить эффективность поиска в различных структурах: Сравнение времени выполнения поиска чисел в различных структурах данных, таких как двоичное дерево, сбалансированное дерево и хеш-таблица. Результаты сравнения будут выведены.

Описание задачи, реализуемой программой

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

Способ обращения к программе

Обращения к программе пользователем происходит с помощью вызова исполняемого файла (app.exe).

Описание возможных аварийных ситуаций и ошибок пользователя

- NO_DATA_ERROR: ошибка отсутствия файла данных. Возникает, если в программу не передан путь к файлу при запуске.
- INPUT_COMMAND_ERR: ошибка ввода команды. Возникает, если пользователь вводит некорректное значение при выборе команды из меню.
- INPUT_FOR_FIND_ERR: ошибка ввода числа для поиска. Возникает, если пользователь ввел некорректное значение при попытке найти число в дереве.
- INPUT_FOR_INSERT_ERR: ошибка ввода числа для добавления в дерево. Возникает, если пользователь вводит некорректное значение при добавлении нового узла.
- INPUT_FOR_REMOVE_ERR: ошибка ввода числа для удаления из дерева. Возникает, если пользователь ввел некорректное значение при удалении узла.
- INPUT_FOR_APPEND_TO_FILE_ERR: ошибка ввода числа для записи в файл. Возникает, если пользователь вводит некорректное значение при попытке добавить новое число в файл.

Описание внутренних структур данных

Программа содержит структуру, которая используется для хранения узлов бинарного дерева, представленную в Листинге 1.

```
typedef struct Node
{
    int value;
    struct Node *left;
    struct Node *right;
} Node;
```

Листинг 1. Структура Node

Рассмотрим каждое поле структуры:

1. value: целочисленное поле типа int, которое хранит значение текущего узла дерева. Это значение представляет данные, связанные с узлом.
2. left: указатель на структуру Node, указывающий на левое поддерево текущего узла. Если левое поддерево отсутствует, это поле принимает значение NULL.
3. right: указатель на структуру Node, указывающий на правое поддерево текущего узла. Если правое поддерево отсутствует, это поле принимает значение NULL.

Еще программа содержит структуру, которая используется для хранения узлов сбалансированного бинарного дерева, представленную в Листинге 2.

```
typedef struct AVLNode
{
    int value;
    int height;
    struct AVLNode *left;
    struct AVLNode *right;
} AVLNode;
```

Листинг 2. Структура AVLNode

Рассмотрим каждое поле структуры:

1. value: целочисленное поле типа int, которое хранит значение текущего узла дерева. Это значение представляет данные, связанные с узлом.

2. height: целочисленное поле, которое хранит высоту узла в контексте его поддеревьев. Высота узла используется для балансировки AVL-дерева, чтобы убедиться, что дерево остается сбалансированным. Высота равна максимальному уровню из поддеревьев плюс один.
3. left: указатель на структуру AVLNode, указывающий на левое поддерево текущего узла. Если левое поддерево отсутствует, это поле принимает значение NULL.
4. right: указатель на структуру AVLNode, указывающий на правое поддерево текущего узла. Если правое поддерево отсутствует, это поле принимает значение NULL.

Также программа содержит структуру, которая используется для хранения хеш-таблицы, представленную в Листинге 3.

```
typedef struct HashTableNode
{
    int value;
    struct HashTableNode *next;
} HashTableNode;

typedef struct HashTable
{
    HashTableNode **buckets;
    size_t size;
    size_t count;
} HashTable;
```

Листинг 3. Структура HashTable

Рассмотрим каждое поле структуры:

1. value: целочисленное поле, которое хранит значение, связанное с текущим узлом хеш-таблицы. Это основная информация, которую мы ищем или сохраняем в таблице.
2. next: указатель на следующий узел в цепочке. Это поле используется для обработки коллизий в хеш-таблице методом цепочек. Если для определенного хеш-ключа возникают коллизии они будут храниться в цепочке узлов, связанных через этот указатель. Если узел является последним в цепочке, это поле будет равно NULL.
3. buckets: указатель на массив указателей на HashTableNode, представляющий собой контейнер (или бакеты) хеш-таблицы. Каждый элемент массива buckets — это указатель на первую ноду цепочки для данного индекса хеш-таблицы. Если в контейнере несколько элементов, они хранятся в виде списка узлов, связанных через поле next.
4. size: размер хеш-таблицы, который определяет количество контейнеров.
5. count: количество элементов, которые хранятся в хеш-таблице.

Описание алгоритма

1. Инициализация системы и выделение памяти.
2. Основной цикл программы начинается. Пользователю предоставляется меню с различными опциями, и программа ожидает ввода выбора пользователя.
3. В зависимости от выбора пользователя, программа выполняет следующие действия:
   - Выйти из программы
   - Считать дерево из файла чисел
   - Вывести дерево
   - Проверить наличие числа в дереве
   - Добавить число в дерево
   - Удалить число из дерева
   - Считать сбалансированное дерево из файла чисел
   - Вывести сбалансированное дерево

- Проверить наличие числа в сбалансированном дереве
- Добавить число в сбалансированное дерево

  Сначала, если дерево пустое, создается новый узел с указанным значением. Далее, значение сравнивается с текущим узлом, и рекурсивно выбирается левое или правое поддерево для вставки. После добавления нового узла обновляется высота текущего узла, и вычисляется баланс дерева. Если баланс нарушен (разница высот поддеревьев больше 1), выполняются соответствующие вращения для восстановления сбалансированности: либо одно вращение (влево или вправо), либо два вращения (сначала влево, затем вправо, или наоборот). В итоге дерево остается сбалансированным после каждой вставки.
- Удалить число из сбалансированного дерева

  Если узел имеет одного или нет поддерева, он заменяется своим дочерним элементом (или удаляется). Если у узла два поддерева, его значение заменяется минимальным элементом правого поддерева. После удаления обновляется высота узлов и проверяется баланс дерева. При нарушении баланса выполняются соответствующие вращения для восстановления сбалансированности.
- Считать хеш-таблицу из файла чисел
- Вывести хеш-таблицу
- Проверить наличие числа в хеш-таблице

  Сначала вычисляется индекс с помощью хеш-функции. Затем, начиная с контейнера, по индексу проверяется каждый узел цепочки. Для каждого узла увеличивается счетчик сравнений (cmpCnt). Если значение найдено, возвращается 1. Если узел не содержит искомое значение, поиск продолжается в следующем узле цепочки. Если значение не найдено после обхода всей цепочки, возвращается 0.
- Добавить число в хеш-таблицу

  Сначала вычисляется индекс, по которому будет добавлено значение, используя хеш-функцию. Затем функция проверяет длину цепочки

коллизий по этому индексу; если длина цепочки превышает 3, вызывается функция перехеширования таблицы. Далее создается новый узел для вставки, и, если выделение памяти прошло успешно, новый узел добавляется в начало цепочки по рассчитанному индексу, увеличивая счетчик элементов в таблице. В случае ошибки выделения памяти функция возвращает код ошибки.

- Удалить число из хеш-таблицы

  Сначала вычисляется индекс с помощью хеш-функции, и начинается обход цепочки в контейнере по этому индексу. Если узел с искомым значением найден, флаг found устанавливается в 1. Если узел не первый в цепочке, его предыдущий узел обновляет указатель на следующий элемент, в противном случае удаляется первый узел цепочки. Узел освобождается, и уменьшается количество элементов в таблице. Если значение не найдено в цепочке, операция завершится без изменений.
- Сравнить эффективность поиска в различных структурах

4. После выполнения каждой операции программа возвращает пользователя в главное меню, где он может выбрать следующее действие.

Набор тестов

Описание	**Результат**
Добавить элемент в сбалансированное дерево Элемент: 3	Успешное добавление элемента в сбалансированное дерево Добавленный элемент: 3
Удалить элемент из сбалансированного дерева Элемент: 5	Успешное удаление элемента в сбалансированное дерево Удаленный элемент: 5
Проверить наличие числа в сбалансированном дереве Элемент: 123	Число 123 найдено в сбалансированном дереве
Вывести хеш-таблицу	Хеш-таблица:

	Контейнер 0: 0 -> NULL Контейнер 1: 1 -> NULL Контейнер 2: 10 -> NULL Контейнер 3: 123 -> NULL Контейнер 4: 12 -> NULL Контейнер 5: NULL Контейнер 6: NULL Контейнер 7: 15 -> 7 -> NULL
Добавить элемент в хеш-таблице Элемент: 3	Успешное добавление элемента в хеш-таблицу Добавленный элемент: 3
Удалить элемент из хеш-таблицы Элемент: 5	Успешное удаление элемента из хеш-таблицы Удаленный элемент: 5
Проверить наличие числа в хеш-таблице Элемент: 123	Число 123 найдено в хеш-таблице

Таблица 2. Набор позитивных тестов

Описание	**Результат**
Попытка не передавать датасет	Возврат ошибки NO_DATA_ERROR
Попытка ввести символы вместо команды	Возврат ошибки INPUT_COMMAND_ERR
Попытка ввести не целое число при поиске элемента	Возврат ошибки INPUT_FOR_FIND_ERR
Попытка ввести не целое число при добавлении элемента	Возврат ошибки INPUT_FOR_INSERT_ERR
Попытка ввести не целое число при удалении элемента	Возврат ошибки INPUT_FOR_REMOVE_ERR

Попытка ввести не целое число при добавлении элемента в файл	Возврат ошибки INPUT_FOR_APPEND_TO_FILE_ERR

Таблица 3. Набор негативных тестов

Оценка эффективности

При запуске программы N = 500 раз, были получены следующие данные:

Таблица 4. Сравнение эффективности для 4 элементов

Реализация	**Время поиска (нс)**	**Кол. сравнений**	**Память (байт)**
Дерево	2640.60	1	156
Сбаланс. дерево	2486.14	1	168
Хеш-таблица	2313.24	1	96

Таблица 5. Сравнение эффективности для 8 элементов

Реализация	**Время поиска (нс)**	**Кол. сравнений**	**Память (байт)**
Дерево	3854.49	3	364
Сбаланс. дерево	2738.64	2	392
Хеш-таблица	2341.62	1	208

Таблица 6. Сравнение эффективности для 32 элементов

Реализация	**Время поиска (нс)**	**Кол. сравнений**	**Память (байт)**
Дерево	6364.60	9	1612
Сбаланс. дерево	3742.77	4	1736
Хеш-таблица	2279.65	1	688

Таблица 7. Сравнение эффективности для 64 элементов

Реализация	**Время поиска (нс)**	**Кол. сравнений**	**Память (байт)**
Дерево	8706.63	15	3276

Сбаланс. дерево	4701.86	5	3528
Хеш-таблица	2662.15	1	1392

Таблица 8. Сравнение эффективности для 128 элементов

Реализация	**Время поиска (нс)**	**Кол. сравнений**	**Память (байт)**
Дерево	16197.91	26	6604
Сбаланс. дерево	4990.65	6	7112
Хеш-таблица	2603.60	1	2800

Таблица 9. Сравнение эффективности для 512 элементов

Реализация	**Время поиска (нс)**	**Кол. сравнений**	**Память (байт)**
Дерево	50739.71	95	26572
Сбаланс. дерево	5552.97	8	28616
Хеш-таблица	2356.49	1	11344

Таблица 10. Сравнение эффективности для 1024 элементов

Реализация	**Время поиска (нс)**	**Кол. сравнений**	**Память (байт)**
Дерево	100839.84	171	53196
Сбаланс. дерево	8893.04	9	57288
Хеш-таблица	2664.81	1	22736

Выводы

В ходе выполнения лабораторной работы была разработана программа для работы с деревом, сбалансированным деревом и хеш-таблицей, предоставляющая пользователю удобный интерфейс для взаимодействия с данными.

В результате тестирования эффективности программы на различных объемах данных, были получены временные показатели для операций поиска с различными структурами.

Согласно результатам тестов, для представленных данных сбалансированное дерево выигрывает обычное дерево по скорости за счет сбалансированности расположения узлов, но проигрывает по памяти для хранения высоты в узле, чтобы как раз и достигать сбалансированности. Однако хеш-таблица выигрывает как по скорости, так и по памяти деревьям. Также можно заметить: скорость доступа к элементам примерно равна, что выполняет концепцию доступа за O(1) — просто обращение к определённой ячейке в массиве, где хранится значение. Хеш-таблица требует меньше памяти из-за своей природы: она использует простую структуру данных (массив). При правильной настройке таблицы и минимальном количестве коллизий память тратится только на массив и значения, что подтверждает хорошую реализацию структуры и подбора хеш-функции. Даже если данные в хеш-таблице занимают больше памяти, на уровне структуры хеш-таблица экономит ресурсы.

В целом, разработанная программа успешно решает поставленную задачу и предоставляет эффективные средства для работы с бинарным деревом поиска, с сбалансированным AVL деревом и с хеш-таблицей.

Контрольные вопросы

1. Идеально сбалансированное дерево и AVL-дерево

Идеально сбалансированное дерево: Каждый уровень дерева полностью заполнен узлами, и высота дерева минимальна. Такие деревья обеспечивают оптимальное время выполнения операций, но в практике редко встречаются из-за ограничений на количество элементов в дереве.

AVL-дерево: это форма сбалансированного дерева двоичного поиска, в котором разница в высоте между левым и правым поддеревьями для каждого узла

ограничена (высота различается не более чем на 1). Это обеспечивает быстрое выполнение операций вставки, удаления и поиска.

2. Поиск в AVL-дереве и дереве двоичного поиска

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

3. Хеш-таблица и её принцип построения

Хеш-таблица — это структура данных, позволяющая эффективно выполнять операции вставки, удаления и поиска. Она использует хеш-функцию для преобразования ключа в индекс массива, где хранятся значения.

Принцип построения: Выбор хеш-функции. Выделение массива определенного размера. Разрешение коллизий (в случае, если два ключа хешируются в один и тот же индекс).

4. Коллизии и методы их устранения

Коллизии возникают, когда два различных ключа хешируются в один и тот же индекс. Методы разрешения коллизий включают:

- Цепочки: Каждый индекс массива представляет собой связанный список.

- Открытое хеширование: при коллизии производится поиск следующего свободного слота в массиве.

- Двойное хеширование: используются две хеш-функции для определения следующего индекса при коллизии.

5. Неэффективность поиска в хеш-таблицах

Поиск в хеш-таблицах становится неэффективным при большом количестве коллизий, что может привести к увеличению длины цепочек или увеличению размера открытого адреса.

6. Эффективность поиска

- В AVL-деревьях и деревьях двоичного поиска поиск выполняется за время, пропорциональное логарифму числа элементов в дереве.

- В хеш-таблицах, при эффективном хешировании, поиск может быть выполнен за постоянное время O(1).