|  |  |
| --- | --- |
| Изображение выглядит как герб, эмблема, символ, нашивка  Автоматически созданное описание | **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 №2 ПО ДИСЦИПЛИНЕ:**

**ТИПЫ И СТРУКТУРЫ ДАННЫХ**

***«Хеш-таблицы и деревья»***

Студент **Демирел Э.А.**

**Вариант 2**

Группа **ИУ7-31Б**

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

|  |  |
| --- | --- |
| Студент | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  **Демирел Э.А.** |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Барышникова М.Ю.** |

**2024**

**Условие задачи**

Используя предыдущую программу (лабораторная работа №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**

**Входные данные:**

Файл, содержащий числа, которые могут храниться в следующих структурах:

* бинарное дерево поиска;
* AVL-дерево;
* Хэш-таблица с закрытой адресацией;
* Хэш-таблица с открытой адресацией.

Строка – имя файла, из которого будут считываться числа.

Целое число – действие, выбранное пользователем (от 0 до 12).

**Выходные данные:**

В результате операций можно получить изображение (файл формата .png), содержащий визуальное представление полученного бинарного или AVL – дерева.

Также на экран выводится содержание хеш-таблиц.

При выборе замерного эксперимента выводится результат сравнения структур, описанных выше.

**Описание задачи:**

Программа обязана мочь строить двоичное дерево поиска и АВЛ-дерево по числам из файла, а также считывать эти числа в хеш-таблицу.

Должны быть функции добавления и удаления чисел из этих структур, а также визуализация деревьев. Программа обязана предоставлять возможность проведения замерного эксперимента.

**Способ обращения к программе:**

Обращение к программе происходит через запуск исполняемого файла app.exe. После запуска пользователем с клавиатуры вводятся желаемые действия и имя файла.

**Аварийные ситуации:**

Все аварийные ситуации, связанные с неправильным вводом пользователя, обработаны. Единственными аварийными ситуациями может быть ошибка выделения памяти и ошибка создания dot файла и изображения, зачастую из-за отсутствия на компьютере пользователя утилиты Graphwiz.

**Используемые структуры данных**

#define TABLE\_SIZE 101

typedef struct HashNode

{

int key;

struct HashNode \*next;

} HashNode;  
  
typedef struct HashTable

{

HashNode \*\*table;

size\_t occupied;

int size;

} HashTable;

TABLE\_SIZE – начальный размер хеш-таблицы при её создании.

HashNode – узел хеш-таблицы с закрытой адресацией.

key – значение узла хеш-таблицы.

next – адрес на следующий узел.

HashTable – структура хеш-таблицы, где table – массив узлов хеш-таблицы, occupied – количество элементов, size – размер хеш-таблицы.

typedef struct TreeNode

{

int value;

struct TreeNode \*left;

struct TreeNode \*right;

} TreeNode;

typedef struct AVLNode

{

int value;

int height;

struct AVLNode \*left, \*right;

} AVLNode;

TreeNode – описание узла бинарного дерева, которое может выступать сразу в качестве корня, так что дополнительной «абстракции» не требуется.

value – значение узла, left и right – адреса левого и правого узла.

AVLNode – описание узла AVL-дерева. Здесь, аналогично, не требуется «абстракция». value – значение узла, height – его высота, left и right – адреса левого и правого узла.

typedef struct HashNodeLinear

{

int key;

int is\_occupied;

} HashNodeLinear;

typedef struct HashTableLinear

{

HashNodeLinear \*table;

size\_t size;

size\_t occupied;

} HashTableLinear;

HashNodeLinear – структура, содержащая значение и флаг занятости ключа (0 – свободно, 1 – занято): key – ключ, is\_occupied – флаг занятости.

HashTableLinear – структура хеш-таблицы с открытой адресацией. table – массив структур HashNodeLinear, size – размер таблицы, occupied – количество элементов.

**Алгоритм программы**

**Общий алгоритм работы программы:**

1. Отображается меню с доступными операциями, затем программа ожидает ввод от пользователя.
2. Пользователь вводит имя файла, в котором содержатся числа.
3. Каждое число потом сохраняется в соответствующей структуре данных.

В зависимости от выбора пользователя производится одна из следующих операций:

1. Добавление элемента
2. Удаление элемента
3. Вывод содержимого структуры
4. Поиск элемента
5. Проведение замерного эксперимента (для оценки производительности)
6. Балансировка двоичного дерева
7. Произвести рехеширование хеш-таблиц
8. Считать данные из файла в хеш-таблицу

**Двоичное дерево поиска (BST)**

Хранит данные в форме двоичного дерева, упорядоченного по ключам.

Поиск в BST:

1. Начинаем с корневого узла.
2. Сравниваем искомый ключ с ключом текущего узла.
3. Если искомый ключ меньше, идём в левое поддерево.
4. Если больше — в правое поддерево.
5. Если равен — нашли нужный узел и возвращаем результат.
6. Продолжаем процесс, пока не найдём ключ или не достигнем конца дерева.

Добавление:

1. Начинаем с корневого узла.
2. Сравниваем добавляемый ключ с ключом текущего узла.
3. Если меньше — переходим к левому поддереву, если больше — к правому.
4. Когда достигаем пустого места, создаём новый узел.

Удаление:

1. Находим узел с заданным ключом.
2. Рассматриваем три ситуации:
3. Узел не имеет потомков: просто удаляем его.
4. Узел имеет одного потомка: удаляем узел и ставим на его место потомка.
5. Узел имеет двух потомков: находим наименьший ключ в правом поддереве, заменяем ключ удаляемого узла этим значением, а затем удаляем дубликат в правом поддереве.

**AVL-дерево**

AVL-дерево — это BST с автоматической балансировкой.

Добавление и удаление аналогичны операциям в BST.

Балансировка:

1. После каждой операции вставки или удаления проверяем балансировку узлов.
2. Если разница в высотах левого и правого поддеревьев для какого-либо узла больше 1, выполняем одно или двойное вращение:
3. LL: Правое вращение
4. RR: Левое вращение
5. LR: Сначала левое, затем правое вращение
6. RL: Сначала правое, затем левое вращение

**Хеш-таблица: Реализуется двумя способами.**

С закрытой адресацией (метод цепочек), где в каждой ячейке хранится связный список.

С открытой адресацией (метод линейного пробирования), где при коллизии ищется следующая свободная ячейка.

Хеш-таблица с **цепочками**:

Поиск:

1. Вычисляем хеш ключа: index = hash(key) % size.
2. Находим соответствующий список в ячейке.
3. Просматриваем элементы списка, пока не найдём совпадающий ключ.

Добавление:

1. Вычисляем индекс по хэшу.
2. Если ключ уже существует, обновляем значение.
3. Если нет — добавляем новый элемент в начало списка.

Удаление:

1. Вычисляем индекс по хэшу.
2. Находим элемент в списке с заданным ключом.
3. Удаляем его, корректируя ссылки в списке.

Хеш-таблица с **линейным пробированием**:

Поиск:

1. Вычисляем начальный индекс: index = hash(key) % size.
2. Если ячейка пуста, поиск неудачен.
3. Если ячейка занята, сравниваем ключи.
4. Если ключ совпадает — возвращаем значение.
5. Если нет — идём к следующей ячейке: (index + 1) % size.
6. Повторяем, пока не найдём нужный элемент или не убедимся, что его нет.

Добавление:

1. Вычисляем индекс по хэшу.
2. Если ячейка пустая — вставляем элемент.
3. Если ячейка занята, но с тем же ключом — обновляем значение.
4. Если ключи разные, ищем следующую пустую ячейку, двигаясь по формуле (index + 1) % size.
5. При необходимости производится рехэширование.

Удаление:

1. Находим элемент по принципу поиска.
2. Если элемент найден, помечаем ячейку как удалённую.
3. После удаления возможно потребуется перестановка (пробирование) других элементов, чтобы сохранить корректность поиска.

Рехэширование

Выполняется, если:

* Таблица заполнена на 75% или более.
* Пользователь выбрал операцию рехэширования.

Процесс рехэширования:

1. Увеличиваем размер таблицы в 2 раза.
2. Повторно вычисляем хеши для всех ключей.
3. Перемещаем элементы в новую, более крупную таблицу.

**Примеры работы программы**

1. Чтение данных (чисел) из файла.

Файл test.txt:

1 -9 3 -1 7 0 10 11 12

Программа для работы с бинарным деревом, AVL-деревом и хеш-таблицей.

Введите имя файла для чтения: test.txt

Деревья успешно построены.

2. Вывод бинарного дерева (на основании п.1)

Выберите действие:

1. Вывести BST дерево (визуализация в Graphviz)

2. Вывести AVL дерево (визуализация в Graphviz)

3. Добавить число в BST и AVL дерево

4. Добавить число в хеш-таблицу

5. Удалить число из BST и AVL дерева

6. Удалить число из хеш-таблицы

7. Найти число в BST, AVL дереве и хеш-таблице

8. Провести замерный эксперимент

9. Сбалансировать BST дерево

10. Выполнить рехеширование хеш-таблицы

11. Показать хеш-таблицу (метод цепочек)

12. Считать данные из файла в хеш-таблицу

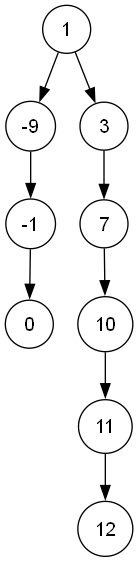
0. Выйти

Ваш выбор: 1

BST дерево экспортировано в bst\_tree.dot.

Изображение BST дерева создано.

Полученное изображение:



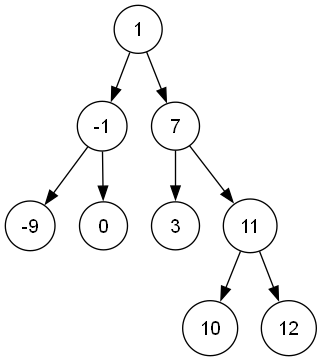
3. Вывод AVL-дерева (на основании п.1)

Ваш выбор: 2

AVL дерево экспортировано в avl\_tree.dot.

Изображение AVL дерева создано.

Полученное изображение:



4. Добавление числа в BST и AVL дерева (сразу с визуализацией)

Ваш выбор: 3

Введите число для добавления: 5

Число добавлено в BST и AVL дерево.

Полученные изображения:

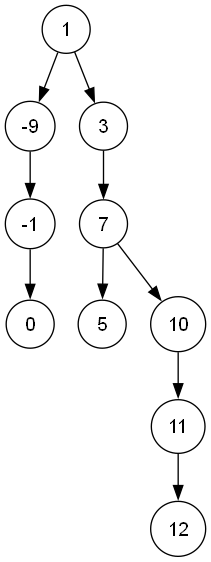


Рисунок 1. BST дерево

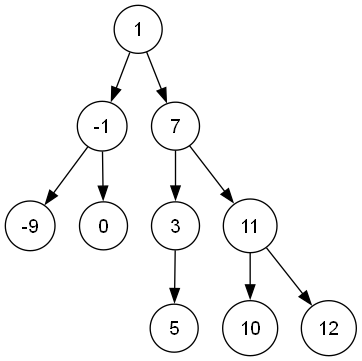


Рисунок 2. AVL дерево

5. Удаление числа из BST и AVL дерева (сразу с визуализацией)

Ваш выбор: 5

Введите число для удаления: 3

Число удалено из BST и AVL дерева.

Полученные изображения:

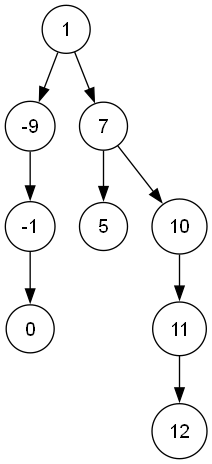


Рисунок 3. BST дерево

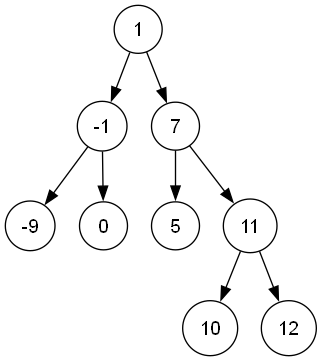


Рисунок 4. AVL дерево

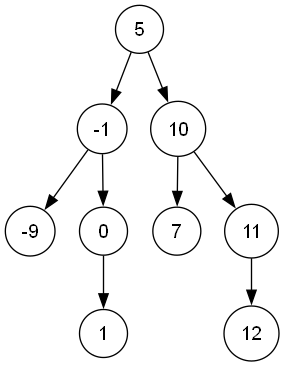
6. Балансировка BST дерева (сразу с визуализацией)

Ваш выбор: 9

Сбалансируем BST дерево.

BST дерево успешно сбалансировано.

Полученное изображение:



7. Чтение данных из файла в хеш-таблицу и её вывод

Ваш выбор: 11

Содержимое хеш-таблицы (закрытая адресация):

[0]: 0 -> NULL

[1]: 1 -> NULL

[2]: NULL

[3]: 3 -> NULL

[4]: NULL

[5]: NULL

[6]: NULL

[7]: 7 -> NULL

[8]: NULL

[9]: NULL

[10]: 10 -> NULL

[11]: 11 -> NULL

[12]: 12 -> NULL

…

8. Добавление числа в хеш-таблицу и проверка работы коллизий.

Ваш выбор: 4

Введите число для добавления в хеш-таблицу: 108

Число добавлено в хеш-таблицу.

Выберите действие:

1. Вывести BST дерево (визуализация в Graphviz)

2. Вывести AVL дерево (визуализация в Graphviz)

3. Добавить число в BST и AVL дерево

4. Добавить число в хеш-таблицу

5. Удалить число из BST и AVL дерева

6. Удалить число из хеш-таблицы

7. Найти число в BST, AVL дереве и хеш-таблице

8. Провести замерный эксперимент

9. Сбалансировать BST дерево

10. Выполнить рехеширование хеш-таблицы

11. Показать хеш-таблицу (метод цепочек)

12. Считать данные из файла в хеш-таблицу

0. Выйти

Ваш выбор: 11

Содержимое хеш-таблицы (закрытая адресация):

[0]: 0 -> NULL

[1]: 1 -> NULL

[2]: NULL

[3]: 3 -> NULL

[4]: NULL

[5]: NULL

[6]: NULL

[7]: 7 -> 108 -> NULL // коллизия

…

9. Удаление числа из хеш-таблицы (на основании п.8)

Ваш выбор: 6

Введите число для удаления из хеш-таблицы: 7

Число удалено из хеш-таблицы.

Выберите действие:

1. Вывести BST дерево (визуализация в Graphviz)

2. Вывести AVL дерево (визуализация в Graphviz)

3. Добавить число в BST и AVL дерево

4. Добавить число в хеш-таблицу

5. Удалить число из BST и AVL дерева

6. Удалить число из хеш-таблицы

7. Найти число в BST, AVL дереве и хеш-таблице

8. Провести замерный эксперимент

9. Сбалансировать BST дерево

10. Выполнить рехеширование хеш-таблицы

11. Показать хеш-таблицу (метод цепочек)

12. Считать данные из файла в хеш-таблицу

0. Выйти

Ваш выбор: 11

Содержимое хеш-таблицы (закрытая адресация):

[0]: 0 -> NULL

[1]: 1 -> NULL

[2]: NULL

[3]: 3 -> NULL

[4]: NULL

[5]: NULL

[6]: NULL

[7]: 108 -> NULL

[8]: NULL

…

10. Рехеширование хеш-таблицы (на основании п.8)

Ваш выбор: 10

Выполняется рехеширование хеш-таблицы.

Рехеширование завершено. Новый размер таблицы: 202

Выберите действие:

1. Вывести BST дерево (визуализация в Graphviz)

2. Вывести AVL дерево (визуализация в Graphviz)

3. Добавить число в BST и AVL дерево

4. Добавить число в хеш-таблицу

5. Удалить число из BST и AVL дерева

6. Удалить число из хеш-таблицы

7. Найти число в BST, AVL дереве и хеш-таблице

8. Провести замерный эксперимент

9. Сбалансировать BST дерево

10. Выполнить рехеширование хеш-таблицы

11. Показать хеш-таблицу (метод цепочек)

12. Считать данные из файла в хеш-таблицу

0. Выйти

Ваш выбор: 11

Содержимое хеш-таблицы (закрытая адресация):

[0]: 0 -> NULL

[1]: 1 -> NULL

[2]: NULL

[3]: 3 -> NULL

[4]: NULL

[5]: NULL

[6]: NULL

[7]: 7 -> NULL

[8]: NULL

…

[106]: NULL

[107]: NULL

[108]: 108 -> NULL

[109]: NULL

…

11. Ввод не числа в меню

Ваш выбор: число

Ошибка: Введите число.

12. Ввод выходит за рамки допустимых чисел

Ваш выбор: 13

Неверный выбор. Попробуйте снова.

13. Буква в добавлении числа

Ваш выбор: 3

Введите число для добавления: п

Ошибка ввода числа.

14. Буква в удалении числа

Ваш выбор: 5

Введите число для удаления: ф

Ошибка ввода числа.

15. Поиск числа в AVL-дереве, бинарном дереве поиска и хеш-таблице

Ваш выбор: 7

Введите число для поиска: 0

Результаты поиска:

BST: найдено кол-во сравнений: 3

AVL: найдено кол-во сравнений: 3

Хеш-таблица: найдено, кол-во сравнений: 1

**Замерный эксперимент**

Измерения проводились для каждой структуры один раз, размеры данных от 1000 до 10000, с шагом в 1000 элементов.

Бинарное дерево поиска:

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер** | **Время поиска (тики)** | **Кол-во сравнений** | **Используемая память, байт** |
| **1000** | 0 | 14 | 12000 |
| **2000** | 0 | 17 | 24000 |
| **3000** | 0 | 11 | 36000 |
| **4000** | 0 | 16 | 48000 |
| **5000** | 0 | 15 | 60000 |
| **6000** | 0 | 20 | 72000 |
| **7000** | 0 | 24 | 84000 |
| **8000** | 0 | 21 | 96000 |
| **9000** | 0 | 13 | 108000 |
| **10000** | 0 | 19 | 120000 |

Хеш-таблица с цепочками:

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер** | **Время поиска (тики)** | **Кол-во сравнений** | **Используемая память, байт** |
| **1000** | 0 | 1 | 16012 |
| **2000** | 0 | 1 | 32012 |
| **3000** | 0 | 1 | 48012 |
| **4000** | 0 | 1 | 64012 |
| **5000** | 0 | 1 | 80012 |
| **6000** | 0 | 1 | 96012 |
| **7000** | 0 | 1 | 112012 |
| **8000** | 0 | 1 | 128012 |
| **9000** | 0 | 1 | 144012 |
| **10000** | 0 | 1 | 160012 |

AVL-дерево:

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер** | **Время поиска (тики)** | **Кол-во сравнений** | **Используемая память, байт** |
| **1000** | 0 | 10 | 16000 |
| **2000** | 0 | 12 | 32000 |
| **3000** | 0 | 11 | 48000 |
| **4000** | 0 | 11 | 64000 |
| **5000** | 0 | 9 | 80000 |
| **6000** | 0 | 13 | 96000 |
| **7000** | 0 | 13 | 112000 |
| **8000** | 0 | 11 | 128000 |
| **9000** | 0 | 13 | 144000 |
| **10000** | 0 | 14 | 160000 |

Хеш-таблица с линейной адресацией:

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер** | **Время поиска (тики)** | **Кол-во сравнений** | **Используемая память, байт** |
| **1000** | 0 | 3 | 16012 |
| **2000** | 0 | 5 | 32012 |
| **3000** | 0 | 1 | 48012 |
| **4000** | 0 | 1 | 64012 |
| **5000** | 0 | 1 | 80012 |
| **6000** | 0 | 1 | 96012 |
| **7000** | 0 | 1 | 112012 |
| **8000** | 0 | 1 | 128012 |
| **9000** | 0 | 1 | 144012 |
| **10000** | 0 | 1 | 160012 |

**Количество сравнений:**

* **Бинарное дерево поиска (BST):** Количество сравнений варьируется от 11 до 24, что отражает непредсказуемость бинарного дерева. Оно растёт нерегулярно при увеличении размера. Это указывает на то, что BST может ухудшаться до почти линейного поиска, если данные вставляются в определённом порядке.
* **AVL-дерево:** Количество сравнений колеблется в районе 9–14, что намного стабильнее, чем у обычного BST. Это ожидаемо, так как AVL-деревья остаются сбалансированными, обеспечивая поиск примерно за O(log n) операций, что отражается в относительно небольшом и почти постоянном числе сравнений.
* **Хеш-таблица с цепочками:** Количество сравнений всегда равно 1. Это указывает на отсутствие или минимум коллизий и очень быстрый доступ, что характерно для хорошего хэширования и равномерного распределения элементов. Поиск фактически сводится к константному времени.
* **Хеш-таблица с линейной адресацией:** В большинстве случаев сравнений 1, за исключением нескольких случаев (1000 и 2000 элементов — 3 и 5 сравнений соответственно), что может говорить о небольшом количестве коллизий, но затем с ростом таблицы или корректным масштабированием число сравнений возвращается к 1. Это также говорит о близком к O(1) доступе.

**Используемая память:**

* Во всех структурах данных память увеличивается линейно с ростом размера.
* Для BST и AVL: память растёт пропорционально количеству узлов. В примере BST требуется 12 байт на узел (примерно), у AVL — чуть больше (в данных примерах 16000 байт на 1000 элементов, то есть около 16 байт на элемент, что может включать дополнительные поля).
* Для хеш-таблиц: память также растёт линейно, но величина и прирост другой.

**Время:**

Почти все структуры, несмотря на разную асимптотическую сложность по времени (O(1) в среднем у хеш-таблиц и O(log(n)) в среднем у деревьев), работают весьма быстро, что даже в тестовой среде в большим количеством данных тики получаются нулевые.

**Контрольные вопросы**

**1. Отличие идеально сбалансированного дерева от AVL-дерева:**  
Идеально сбалансированное дерево – это дерево, в котором все листья расположены на одинаковой или почти одинаковой глубине, и каждая вершина делит множество узлов на примерно равные части. Это обеспечивает минимально возможную высоту и высокую эффективность операций, однако такое дерево сложно поддерживать.  
AVL-дерево — это сбалансированное двоичное дерево поиска, в котором разница высот левого и правого поддерева у любого узла не превышает 1. Оно не столь идеально, как идеально сбалансированное дерево, но достигает примерно минимальной высоты благодаря локальным балансировкам, что делает его реализацию и обслуживание проще.

**2. Различия между поиском в AVL-дереве и обычном двоичном дереве поиска:**  
Поиск в AVL-дереве осуществляется так же, как и в обычном двоичном дереве поиска — сравниваем ключи и идём влево или вправо. Однако, поскольку AVL-дерево поддерживает балансировку, его высота остаётся примерно равной O(log n). Это гарантирует, что поиск всегда будет эффективным. В обычном двоичном дереве поиска в худшем случае (когда дерево вырождается в цепочку) время поиска может достигать O(n).

**3. Что такое хеш-таблица и принцип её работы:**  
Хеш-таблица — это структура данных для хранения пар «ключ-значение», обеспечивающая быстрый доступ к данным.  
Работает она следующим образом:

* Хеш-функция преобразует ключ в числовой индекс.
* По этому индексу значение сохраняется в массиве.
* При поиске ключ снова пропускают через хеш-функцию, и по полученному индексу быстро находят соответствующее значение.

**4. Что такое коллизии и методы их разрешения:**  
Коллизия возникает, когда два разных ключа дают один и тот же хеш-индекс.  
Способы решения:

* Цепочки (закрытая адресация): В каждой ячейке таблицы хранится список, в котором сохраняются все ключи с одинаковым хешем.
* Рехеширование (увеличение размера таблицы и пересчёт хеша для всех ключей), что уменьшает вероятность коллизий.

**5. В каких случаях поиск в хеш-таблице становится неэффективным:**  
Поиск в хеш-таблице ухудшается, если:

* Происходит множество коллизий, что вынуждает просматривать несколько элементов списка или последовательность пробирований.
* Таблица часто пересоздаётся и перераспределяется (рехешируется) из-за высокой заполненности или плохого хеширования.

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

|  |  |  |
| --- | --- | --- |
| Структура данных | Худшее время поиска | Среднее время поиска |
| Файл | O(n) | O(n) |
| Хэш-таблица | O(n) | O(1) |
| BST | O(n) | O(log(n)) |
| AVL-дерево | O(log(n)) | O(log(n)) |

**Вывод**

В ходе лабораторной работы экспериментальным образом было показано, что ъеш-таблицы показывают почти постоянное количество сравнений (практически 1), что соответствует теоретической O(1) сложности. AVL-деревья сохраняют баланс и обеспечивают сравнительно стабильный (около O(log n)) поиск, а обычные двоичные деревья поиска могут работать нестабильно и иногда деградировать до линейной сложности.