



**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)**

Лабораторная Работа №7 «Сбалансированные деревья, хеш–таблицы » Вариант №6

Студент **Шахнович Дмитрий Сергеевич**

Группа **ИУ7-22Б**

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

Студент **Шахнович Д.С.**

Оценка _____

2023 г.

Описание условия задачи

Построить хеш-таблицу по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице (используя открытую и закрытую адресацию). 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 по памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий и при различных методах их разрешения.

Используя предыдущую программу (задача No6), построить дерево, например, для следующего выражения: $9+(8*(7+(6*(5+4)-(3-2))+1))$. При постфиксном обходе дерева, вычислить значение каждого узла и результат записать в его вершину. Получить массив, используя инфиксный обход полученного дерева. Построить для этих данных дерево двоичного поиска (ДДП), сбалансировать его. Построить хеш-таблицу для значений этого массива. Осуществить поиск указанного значения. Сравнить время поиска, объем памяти и количество сравнений при использовании ДДП, сбалансированных деревьев и хеш-таблиц.

Техническое задание

Исходные данные:

Для всех структур исходными данными являются целые числа.

Выходные данные:

Программа должна выдавать введенные деревья, хэш-таблицу. При поиске выдавать количество совершенных сравнений. Также должна выдавать замеры поиска числа в различных структурах данных и размеры этих структур.

Описание задания:

Реализация основных методов для ДДП, АВЛ и хэш-таблиц, реализация перевода дерева выражения в выше написанные структуры. Сравнить эффективность структур в поиске.

Способы обращения к программе:

Запуск программы через терминал, затем управление программой с помощью меню. Пункты меню:

- 1 — Добавить число в ДДП;
- 2 — Найти число в ДДП;
- 3 — Удалить число из ДДП;
- 4 — Вывести ДДП;

- 5 — Добавить число в АВЛ;
- 6 — Найти число в АВЛ;
- 7 — Удалить число из АВЛ;
- 8 — Вывести АВЛ;

- 9 — Добавить число в хэш-таблицу;
- 10 — Найти число в хэш-таблице;
- 11 — Удалить число из хэш-таблицы;
- 12 — Реструктуризовать хэш-таблицу;
- 13 — Вывести хэш-таблицу;

- 14 — Изменить переменные выражения;
- 15 — Подсчитать выражение в дереве и занести в массив;
- 16 — Добавить числа из массива в ДДП;
- 17 — Добавить числа из массива в АВЛ;
- 18 — Добавить числа из массива в хэш-таблицу;
- 19 — Сравнить;
- 0 — Выход.

Аварийные ситуации:

- 1) Ввод несуществующей команды в меню;
Сообщение: «Ошибка: Некорректная команда.»
- 2) Ошибка ввода/вывода;
Сообщение: «Ошибка функций ввода/вывода.»
- 3) Неудачная попытка работы с файлом
Сообщение: «Ошибка при работе с файлом.»
- 4) Неудачная попытка выделения памяти;
Сообщение: «Ошибка выделения памяти.»
- 5) Ввод литералов или чисел вне запрашиваемого диапазона;
Сообщение: «Ошибка: Некорректный формат ввода.»
- 6) Удаление элемента дерева или хэш-таблицы, которого не существует;
Сообщение: «Ошибка: Элемент не найден.»
- 7) Добавление существующего элемента в дерево или хэш-таблицу;
Сообщение: «Ошибка: Элемент уже существует.»
- 8) Деление на ноль в дереве;
Сообщение: «Ошибка: Деление на ноль.»

Описание структур данных

```

/// @brief Структура узла бинарного дерева выражения
struct btree_node
{
    btree_node *parent; /// Указатель на родителя узла, для корня - NULL
    btree_node *left; /// Указатель на левого потомка
    btree_node *right; /// Указатель на правого потомка
    int data_id; /// Переменная для определения информационной части
    узла.
    /// Если VALUE_ID, то хранится число, Если OPERATION_ID, то хранится
    символ операции
    union
    {
        int value;
        char op;
    } data; /// Информационная часть узла
};

```

```

/// @brief Структура узла бинарного дерева поиска целых
/// Также используется для АВЛ
struct int_node_t
{
    int_node_t *left; /// Указатель на левого потомка
    int_node_t *right; /// Указатель на правого потомка
    int data; /// Число узла
    int height; /// Высота дерева
};

```

```

/**
 * @brief Узел списка, хранящий целые числа
 *
 */
struct int_listnode_t

```

```

{
    int data; /// Значение - целое
    int_listnode_t *next; /// Указатель на следующий элемент
};

/**
 * @brief Список со данными узла в виде ключ-строка - целое число
 *
 */
struct int_list
{
    int_listnode_t *start; /// Указатель на голову списка
    int_listnode_t *end; /// Указатель на хвост списка
};

```

```

/// Хеш таблица с открытой адресацией.
/// В качестве хэша числа используется остаток от деления на размер
таблицы
struct ohash_table_t
{
    int_list *arr; /// Массив списков целых чисел
    size_t size; /// Размер таблицы/массива
};

```

Описание алгоритма

1. Вывести пользователю меню и ожидать ввода номера команды;
2. В зависимости от выбора пункта провести операцию с текущим деревом символов или провести вычисления на дереве выражений.

При возникновении ошибок или завершения пункта вернуться к выбору пункта меню.

Алгоритм реструктуризации хэш-таблицы.

При реструктуризации таблицы изменяется её размер и все элементы заново распределяются по новым индексам.

Мой алгоритм реструктуризации для таблицы с открытой адресацией следующий:

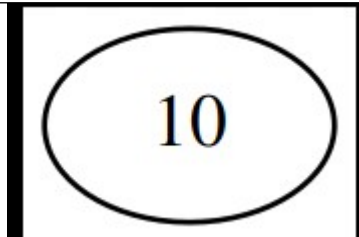
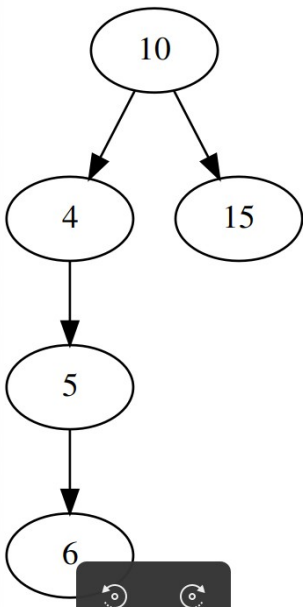
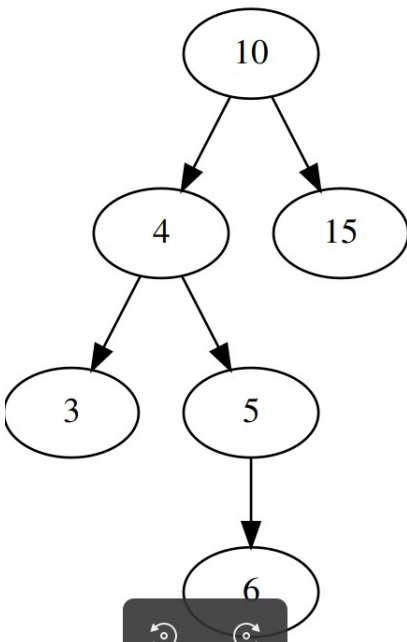
1) Из исходной таблица переносим все числа в один линейный список(так как при открытой адресации числа и так хранятся в списках, то это позволяет не выделять заново память, а использовать те же списки.)

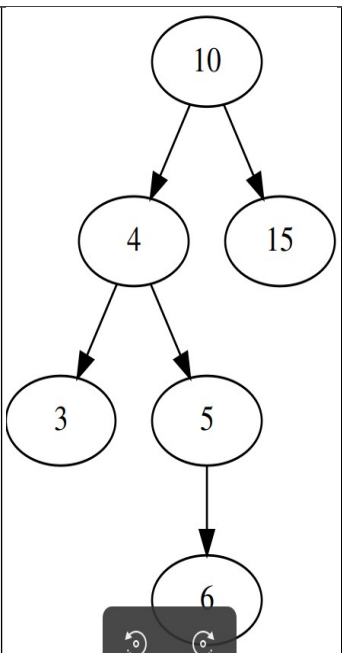
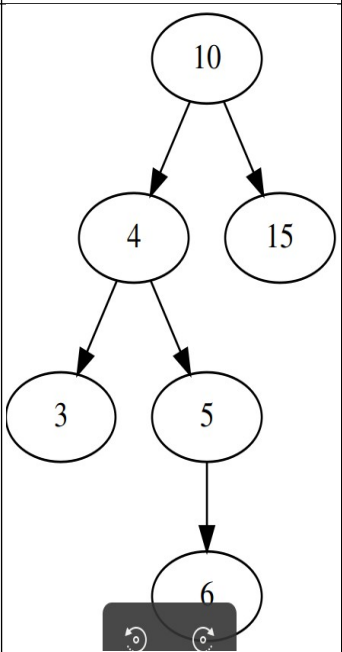
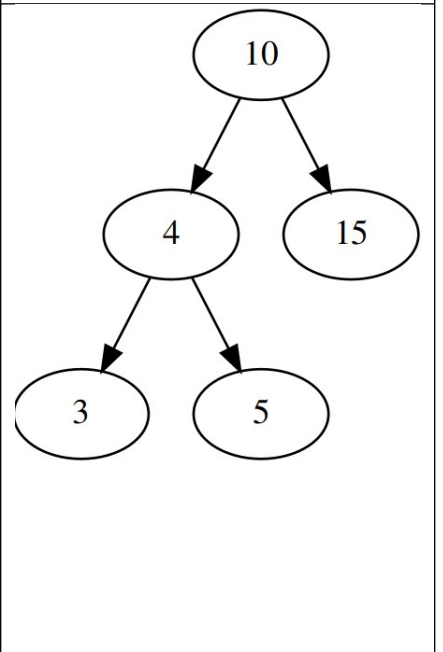
2) Изменяем размер таблицы на нужный.

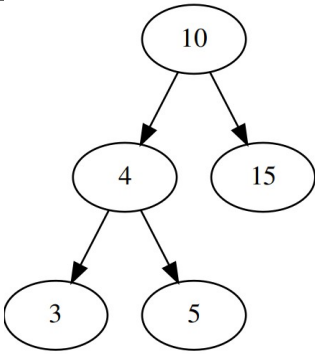
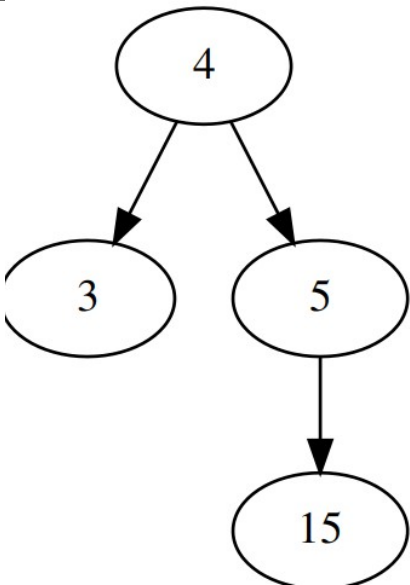
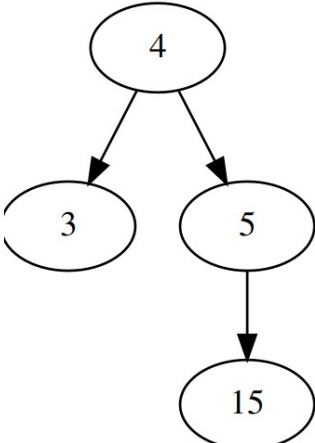
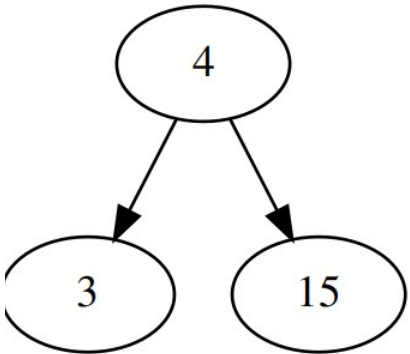
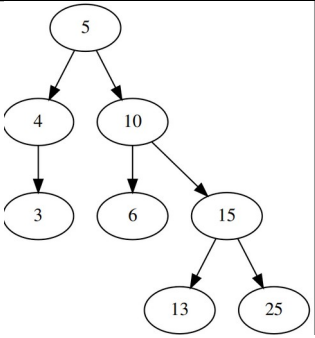
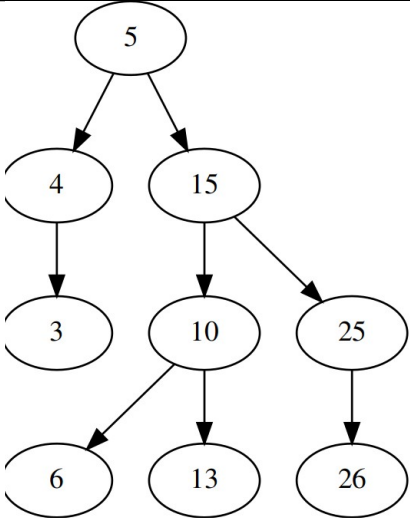
3) Расставляем все числа из собранного списка в измененную хэш-таблицу по правилам хэш-функции.

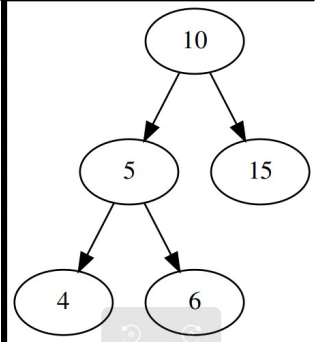
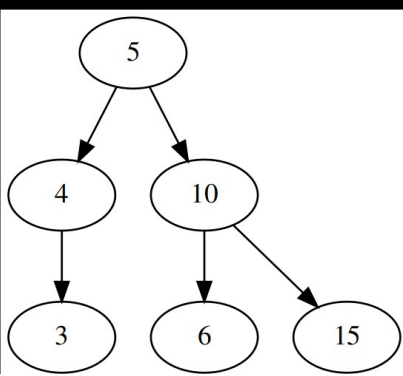
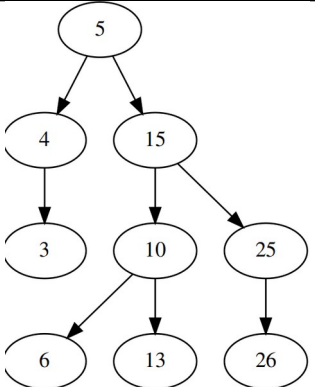
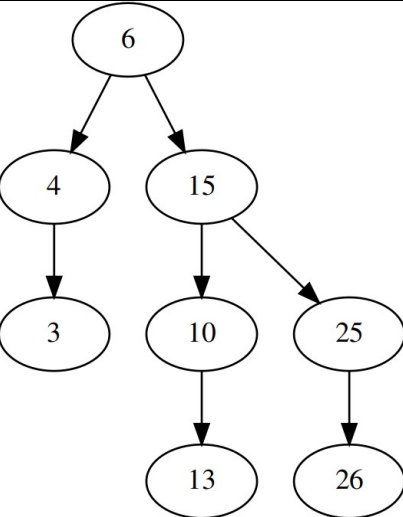
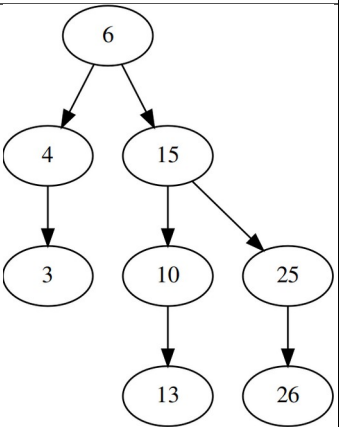
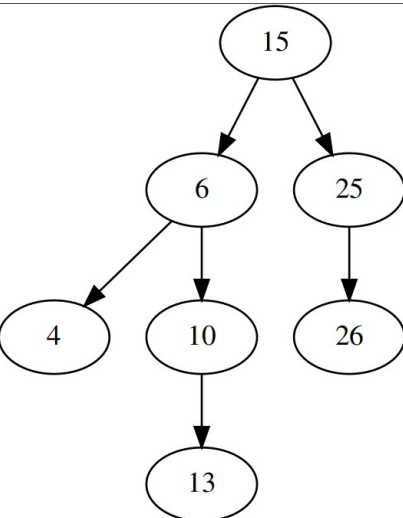
В итоге получаем таблицу нужного размера с корректно расставленными элементами.

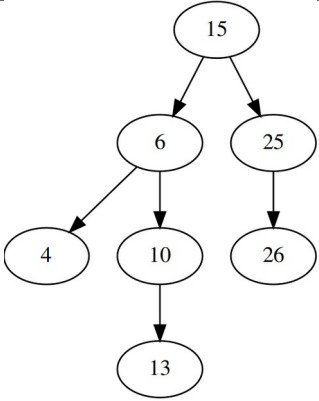
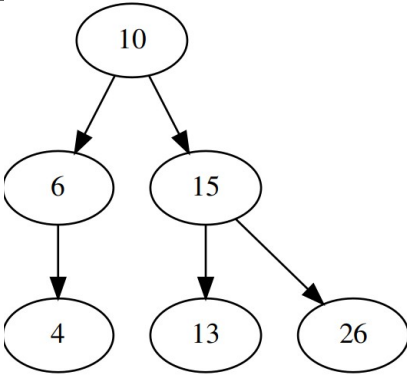
Тестовые данные

Позитивные тесты			
№	Описание	Вход	Выход
1	Добавить число в пустое ДДП	Граф: Пустой 10	
2	Добавить числа в ДДП с несколькими элементами	 3	

3	Найти существующее число в ДДП	 <p>6</p>	Число найдено в дереве.
4	Удалить лист ДДП	 <p>6</p>	

5	Удалить корень ДДП	 <p>10</p>	
6	Удалить потомка из середины ДДП	 <p>5</p>	
7	Добавить элемент в AVL с нарушением баланса	 <p>26</p>	

8	Добавить элемент в AVL с нарушением баланса до корня	 <p>3</p>	
9	Удалить корень AVL	 <p>5</p>	
10	Удалить лист AVL с нарушением баланса	 <p>3</p>	

11	Удалить потомка из середины AVL с нарушением баланса	 <p>25</p>	
12	Добавить элемент в хэш таблицу	<p>Таблица:</p> <p>0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 3 - 123 4 - 64 -> 4 22</p>	<p>0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 123 4 - 64 -> 4</p>
13	Добавить элемент в хэш таблицу в пустой индекс	<p>0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 4 - 64 -> 4 88</p>	<p>0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 88 4 - 64 -> 4</p>
14	Удалить элемент из хэш таблицы	<p>0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 88 4 - 64 -> 4 4</p>	<p>0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 88 4 - 64</p>
15	Удалить элемент из хэш	0 - 25 -> 1245 -> 65	0 - 25 -> 1245 -> 65 -> 5

	таблицы, опустошив индекс	-> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 88 4 — 64 64	1 - 1 2 - 52 -> 32 -> 22 3 - 88
16	Удалить элемент из хэш таблицы из середины списка.	0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 88 4 - 64 -> 4 1245	0 - 25 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 88 4 - 64 -> 4
17	Реструктуризовать таблицу с увеличением размера	0 - 25 -> 1245 -> 65 -> 5 1 - 1 2 - 52 -> 32 -> 22 3 - 88 4 - 64 -> 4 9	1 - 1 -> 64 2 - 65 3 - 1245 4 - 22 -> 4 5 - 5 -> 32 7 - 25 -> 52 -> 88
18	Реструктуризовать таблицу с уменьшением размера	1 - 1 -> 64 2 - 65 3 - 1245 4 - 22 -> 4 5 - 5 -> 32 7 - 25 -> 52 -> 88 2	0 - 64 -> 22 -> 4 -> 32 -> 52 -> 88 1 - 1 -> 65 -> 1245 -> 5 -> 25
Негативные тесты			
1	Ввести некорректный	16	Ошибка: Некорректная

	код в меню.		команда.
2	При запросе числа ввести литерал.	sda	Ошибка: Некорректный формат ввода.
3	При запросе символа ввести несколько	sdw	Ошибка: Некорректный формат ввода.
4	Добавить число в дерево или хэш-таблицу, где уже есть это число	Дерево: «a» a	Ошибка: Элемент уже существует.
5	Удалить не существующее в дереве или хэш-таблице число	Дерево: «a» b	Ошибка: Элемент не найден.
6	Заполнить пустым массивом дерева или хэш-таблицу		Пустой массив
7	Задать выражение в дереве с делением на ноль	10 / 0	Ошибка: Деление на ноль.

Замеры эффективности

Замеры выполнения операции проводились 1000000 раз для каждого метода хранения со случайными значениями чисел в выражении. В качестве результата бралось среднее.

Время выполнения вычислений

Время поиска в ДДП, нс	Время поиска в АВЛ, нс	Время поиска в таблице размера 5, нс	Время поиска в таблице размера 10, нс	Время поиска в таблице размера 15, нс	Время поиска в таблице размера 20, нс
47	39	38	33	31	31

Среднее количество сравнений

Поиск в ДДП, нс	Поиск в АВЛ, нс	Поиск в таблице размера 5, нс	Поиск в таблице размера 10, нс	Поиск в таблице размера 15, нс	Поиск в таблице размера 20, нс
4	3	2	1	1	1

Объем памяти

ДДП, байт	АВЛ, байт	Хэш-таблица размера 5, байт	Хэш-таблица размера 10, байт	Хэш-таблица размера 15, байт	Хэш-таблица размера 20, байт
408	408	352	432	512	592

Как видно самым быстрым оказался метод хранения Хэш-таблицей, при этом даже с размером, в котором возникают коллизии. По памяти также

выиграла хэш-таблица, если брать размер в 5 элементов. Также как и ожидалось AVL-дерево выигрывает обычное дерево за счет сбалансированности.

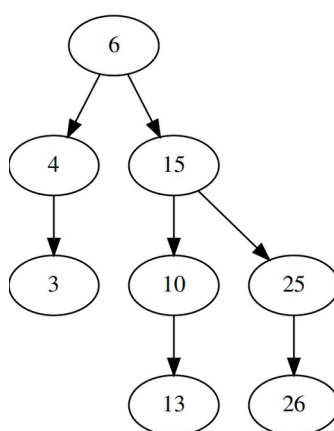
Интересно заметить, что при увеличении размера таблицы ускоряется поиск, но только до 20 элементов. Это связано с тем, что в результате выражения получается ~ 17 чисел и при увеличении размера от 15 до 20 количество коллизий почти не уменьшается и любой поиск происходит за константу.

Ответы на вопросы

1. Чем отличается идеально сбалансированное дерево от AVL дерева?
Идеально сбалансированное дерево предполагает, что все его уровни полностью заполнены, кроме, может быть последнего.

В AVL деревьях могут заполнены не все уровни, однако должно выполняться правило, что для любого узла отличие высот его поддеревьев не больше 1.

Например:



Данное дерево не является идеально-сбалансированным, так как 2-ой уровень(начиная с 0) заполнен не полностью, однако оно является AVL.

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

Алгоритм поиска ничем не отличается, так как AVL — подвид ДДП, однако поиск в AVL может выполняться быстрее, так как оно сбалансировано.

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

Хэш-таблицей называется массив, который заполнен по принципу, что индекс некоторого элемента есть значение некоторой функции, которая называется хэш-функцией. Это позволяет находить место элемента в массиве за константу.

4. Что такое коллизии? Каковы методы их устранения.

Коллизией называется совпадение значений хэш-функции для различных элементов. Это приводит к тому, что в одной ячейке массива должно лежать два различных значения.

Существует несколько методов разрешения коллизий, например внутреннее хеширование, при котором если ячейка в массиве уже занята, то идем далее с некоторым шагом, пока не найдем свободную.

Еще одним методом разрешения коллизий является метод цепочек, при котором к уже существующему элементу в массиве мы цепляем ссылку на следующий. Таким образом у нас получается массив односвязных списков. Преимуществом этого метода по сравнению с предыдущим является неограниченность количества элементов в таблице, однако такой метод тратит больше памяти.

5. В каком случае поиск в хэш-таблицах становится неэффективен? При большом числе коллизий поиск в хэш-таблице превращается из константного в линейный и становится неэффективным. Также он становится неэффективным, если хэш-функция вычисляется дольше, чем занял бы поиск в другой структуре.

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

Как показала практика, самой эффективной оказалась Хэш-таблица. АВЛ обогнала ДДП. И так как в файле происходит линейный поиск, то вероятно все вышеописанные методы будут эффективнее него.

Выводы

Во многих областях возникает задача хранения и быстрого поиска данных. В таких случаях для максимальной оптимизации программист должен выбрать оптимальную структуру, которая будет эффективно выполнять эти задачи по памяти и времени.