**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

# Отчёт по лабораторной работе № 6
# «Обработка деревьев и хеш-таблиц»

Студент Шелия София Малхазовна

Группа ИУ7 – 35Б

*2020 г.*

**Цель работы:** построить дерево, вывести его на экран в виде дерева, реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов, сбалансировать дерево, сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления; построить хеш-таблицу и вывести ее на экран, устранить коллизии, если они достигли указанного предела, выбрав другую хеш-функцию и реструктуризировав таблицу; сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска (ДДП), в хеш-таблицах и в файлах. Сравнить эффективность реструктуризации таблицы для устранения коллизий и поиска в ней с эффективностью поиска в исходной таблице.

## 1. Описание условия задачи.

### Вариант №6

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.

## 2. Техническое задание

**Исходные данные и результат**

**Ввод:**

1. Пользователь вводит целое положительно число – число сравнений после которых происходит реструктуризация хэш-таблицы.
2. Пользователь вводит цифру 1 или 2 (1 – чтение данных для структур из файла, 2 – работа с пустыми структурами)
3. Пользователь вводит цифру
   - 1 - Обновить файл для текущего состояния дерева
   - 2 - Обновить файл для текущего состояния avl-дерева
   - 3 - Вывести текущее состояние хэш-таблицы
   - 4 - Добавление элемента

   *Пользователь вводит целое число – значение элемента для добавления.*
   - 5 - Удаление элемента

   *Пользователь вводит целое число – значение элемента для удаления.*
   - 6 - Сменить количество сравнений для реструктуризации хэш-таблицы

   *Пользователь вводит целое число – количество сравнений после которых должна происходить реструктуризация*
   - 7 - Сменить значение ключа для хэш-таблицы

   *Пользователь вводит целое положительное простое число – новый ключ для хэш-таблицы*

- 8 - Оценить эффективность поиска в разных структурах данных
- 0 - Завершить программу

**Вывод: (для пунктов 1 - 8)**

1. Вывода нет, данные из обычного дерева передаются в файл gv
2. Вывода нет, данные из avl-дерева передаются в файл gv
3. Выводится текущее состояние хэш-таблицы
4. Вывода нет
5. Вывода нет
6. Вывода нет
7. Вывода нет
8. Выводится сравнительная таблица времени поиска элемента для разных структур при различном количестве элементов в них

### Описание задачи, реализуемой программой

1. Программа создает дерево, avl-дерево, хэш-таблицу
2. Программа производит операции добавления/удаления элемента одновременно для всех структур.
3. Программа изменяет значение коллизии и ключа для хэш-таблицы.
4. Программа сравнивает время поиска элемента для разных структур при различном количестве элементов в них.

### Способ обращения к программе

Запуск приложения возможен через терминал MSYS2, а именно.

```
1. gcc -std=c99 -Wall -Werror -c *.c
2.gcc -o main.exe *.o
3. ./main.exe
```

## Описание возможных аварийных ситуаций и ошибок пользователя

**Ошибки пользователя при выборе пункта меню**

- Пустой ввод
- Вводятся посторонние символы (например, буквы)
- Вводится отрицательное число
- Вводится не целое число
- Вводится отсутствующий пункт меню

**Ошибки пользователя при вводе элемента для добавления/удаления**

- Пустой ввод
- Вводятся посторонние символы (например, буквы)
- Вводится не целое число

**Ошибки пользователя при вводе ключа**

- Пустой ввод
- Вводятся посторонние символы (например, буквы)
- Вводится не целое число
- Вводится не простое число

**Ошибки пользователя при вводе нового значения коллизии**

- Пустой ввод
- Вводятся посторонние символы (например, буквы)
- Вводится не целое число

### Аварийные ситуации.

- Попытка удаление элемента из пустой структуры

## 3. Описание внутренних структур данных

Для хранения дерева в обычном виде я использовала следующую структуру:

```
typedef struct tree_node
{
        int data;
        struct tree_node *left;
        struct tree_node *right;
        struct tree_node *parent;
}tree_node;
```

Для хранения avl-дерева я использовала следующую структуру:

```
typedef struct avl_tree_node
{
   int data;
   int height;
   struct avl_tree_node *left;
   struct avl_tree_node *right;

}avl_tree_node;
```

Для хранения хэш-таблицы я использовала массив указателей на элементы следующего структурного типа

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

## 4. Алгоритм

## Обычное дерево

Как происходит заполнение дерева при считывании данных из файла?
Сначала из файла считывается значение нового элемента, далее идет проход по дереву: если значение элемента больше текущего, то мы переходим на правую ветку, а иначе на левую, пока не найдем свободное место для вставки нового элемента. (если элемент равен текущему, то мы прерываем добавление, потому что элемент с данным значением уже существует в дереве)

Добавление:
Добавление происходит по описанному выше алгоритму, то есть с помощью спуска по дереву и поиску свободного узла.

Удаление:
Сначала по дереву ищется элемент, который нужно удалить (если его нет никаких дальнейших действие не происходит, так как удалять нечего)
Если элемент все же найден, то далее возможны три ситуации:

1. Удаляемый элемент является листом.

Происходит освобождение памяти удаляемого узла, а его родитель начинает указывать на NULL.

2. Удаляемый элемент имеет правый или левый подузел.

В этом случае из-под узла освобождается память, а его родитель начинает указывать на узел, на который указывал удаляемый элемент.

3. Удаляемый элемент имеет и правый и левый подузел.

В этом случае происходит поиск максимума в левом поддереве (относительно удаляемого узла), и место удаляемого элемента занимает этот левый максимум, то есть меняются местами указатели и из-под удаляемого узла освобождается память

## AVL – дерево

Отличительное особенность AVL-дерева от обычного является балансировка, которая осуществляется левыми и правами поворотами вокруг узла.
То есть после добавление нового элемента происходит проверка на соблюдение баланса (у каждого узла высота двух поддеревьев отличается не более чем на единицу), если данное условие нарушается, то происходит балансировка.

## hash-таблица

Для хеш-таблицы изначально выбираются два значения: ключ хеш-функции и кол-во сравнений, после превышения которого таблица должно провести реструктуризацию.

Что представляет из себя ключ хеш-функции?
Это какое-то простое число, с помощью которого определяется индекс ячейки для добавления/поиска/удаления элемента (Индекс ячейки = значение элемента % ключ)

Что представляет из себя количество сравнений, после которого требуется реструктуризация?

Если, например, при поиске или удалении элемента программа определила ячейку, где хранится нужный элемент, но, чтобы дойти до него ей нужно сделать еще несколько шагов по связному списку, и кол-во этих шагов превышает заданное, то это значит, что требуется реструктуризация таблицы для сокращения кол-ва шагов при поиске нужного элемента (следовательно, сокращении времени поиска).

Как происходит считывание данных в хеш-таблицу?
Изначально у нас есть массив, элементами которого являются указатели на связные списки.
При считывании нового элемента из файла, программа сначала определяет адрес его ячейки, а затем, добавляет его в связной список, находящийся в данной ячейке. Причем, если при добавлении кол-во сравнений превысило заданное, то происходит реструктуризация и только потом продолжается считывание данных.

Как происходит реструктуризация?
Для начала выбирается новое значение ключа для хеш-функции: самое близкое к нынешнему значению простое число, большее по значению.
Дальше по старому принципу идет перенос значений в новую пустую хеш-таблицу с новым значением ключа, если в процессе переноса кол-во сравнений вновь превысило максимальное, то перенос останавливается, новая хеш-таблица очищается и вновь ищется значение ключа. Если перенос значений прошел успешно, и новая реструктуризация не требуется, то освобождается память для старой хеш-таблицы.

## 5. Тесты

**Негативные тесты**

| **Входные данные** | **Результат** | **Условие проверки** |
|---|---|---|
| **Для выбора пункта меню** | | |
| Выберите действие:<br>1 - Обновить файл для текущего состояния дерева<br>2 - Обновить файл для текущего состояния avl-дерева<br>3 - Вывести текущее состояние хэш-таблицы<br>4 - Добавление элемента<br>5 - Удаление элемента<br>6 - Сменить количество сравнений для реструктуризации хэш-таблицы<br>7 - Сменить значение ключа для хэш-таблицы<br>8 - Оценить эффективность поиска в разных структурах данных<br>0 - Завершить программу<br>Выбор: | Некорректный ввод. | Пустой ввод |

| | | |
|---|---|---|
| Выберите действие:<br>1 - Обновить файл для текущего состояния дерева<br>2 - Обновить файл для текущего состояния avl-дерева<br>3 - Вывести текущее состояние хэш-таблицы<br>4 - Добавление элемента<br>5 - Удаление элемента<br>6 - Сменить количество сравнений для реструктуризации хэш-таблицы<br>7 - Сменить значение ключа для хэш-таблицы<br>8 - Оценить эффективность поиска в разных структурах данных<br>0 - Завершить программу<br>Выбор: skd | Некорректный ввод. | Вводятся посторонние символы (например, буквы) |
| Выберите действие:<br>1 - Обновить файл для текущего состояния дерева<br>2 - Обновить файл для текущего состояния avl-дерева<br>3 - Вывести текущее состояние хэш-таблицы<br>4 - Добавление элемента<br>5 - Удаление элемента<br>6 - Сменить количество сравнений для реструктуризации хэш-таблицы<br>7 - Сменить значение ключа для хэш-таблицы<br>8 - Оценить эффективность поиска в разных структурах данных<br>0 - Завершить программу<br>Выбор: -43 | Некорректный ввод. | Вводится отрицательное число |
| Выберите действие:<br>1 - Обновить файл для текущего состояния дерева<br>2 - Обновить файл для текущего состояния avl-дерева<br>3 - Вывести текущее состояние хэш-таблицы<br>4 - Добавление элемента<br>5 - Удаление элемента<br>6 - Сменить количество сравнений для реструктуризации хэш-таблицы<br>7 - Сменить значение ключа для хэш-таблицы<br>8 - Оценить эффективность поиска в разных структурах данных<br>0 - Завершить программу<br>Выбор: 8789 | Некорректный ввод. | Вводится отсутствующий пункт меню |
| Выберите действие:<br>1 - Обновить файл для текущего состояния дерева | Некорректный ввод. | Вводится не целое число |

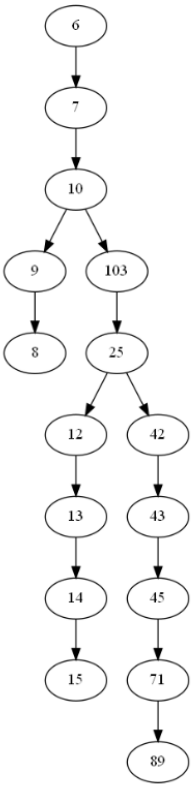
| | | |
|---|---|---|
| 2 - Обновить файл для текущего состояния avl-дерева<br>3 - Вывести текущее состояние хэш-таблицы<br>4 - Добавление элемента<br>5 - Удаление элемента<br>6 - Сменить количество сравнений для реструктуризации хэш-таблицы<br>7 - Сменить значение ключа для хэш-таблицы<br>8 - Оценить эффективность поиска в разных структурах данных<br>0 - Завершить программу<br>Выбор: 8789.43 | | |
| **Для ввода значения элемента** | | |
| Введите целое число - значение элемента для добавления/удаления: | Некорректный ввод.<br>Введите целое число: | Пустой ввод |
| Введите целое число - значение элемента для добавления/удаления: 1ава | Некорректный ввод.<br>Введите целое число: | Вводятся посторонние символы (например, буквы) |
| Введите целое число - значение элемента для добавления/удаления: 34.12 | Некорректный ввод.<br>Введите целое число: | Вводится не целое число |
| **Для ввода значения ключа** | | |
| Введите новый ключ(целое положительное простое число < 100): | Некорректный ввод.<br>Введите целое число: | Пустой ввод |
| Введите новый ключ(целое положительное простое число < 100): шуща | Некорректный ввод.<br>Введите целое число: | Вводятся посторонние символы (например, буквы) |
| Введите новый ключ(целое положительное простое число < 100): 90.3 | Некорректный ввод.<br>Введите целое число: | Вводится не целое число |
| Введите новый ключ(целое положительное простое число < 100): 16 | Некорректный ввод.<br>Введите целое число: | Вводится не простое число |

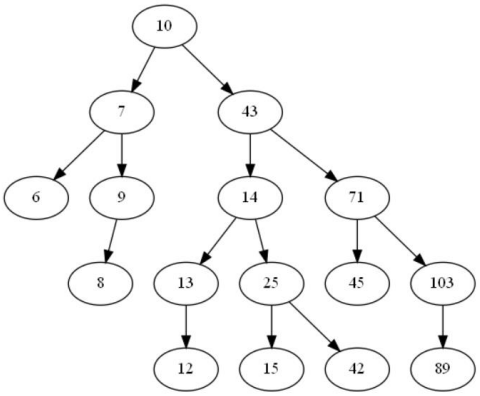
| Входные данные | Результат | Условие проверки |
|---|---|---|
| | Попытка удаления из пустых структур! | Попытка удаления из пустых структур |
| | Элемент с таким значением отсутствует. | Попытка удаления несуществующего элемента |
| | Данный элемент отсутствует. | Поиск несуществующего элемента |

## Позитивные тесты

## 1. Обычное добавление

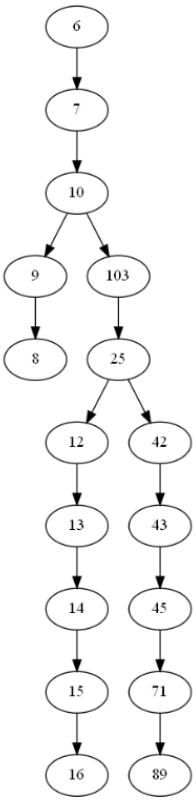
Обычное дерево AVL-дерево Хеш-таблица

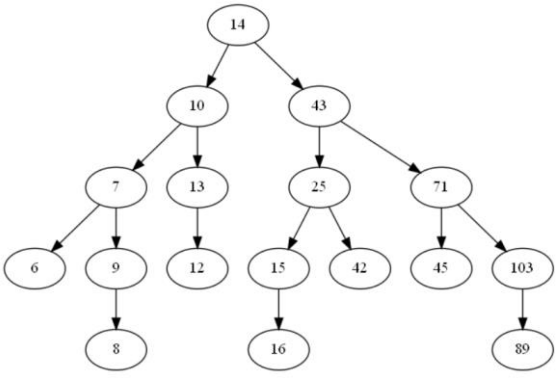



```
Hash key = 7
0: 7 -> 14 -> 42 -> NULL
1: 8 -> 15 -> 43 -> 71 -> NULL
2: 9 -> NULL
3: 10 -> 45 -> NULL
4: 25 -> NULL
5: 12 -> 89 -> 103 -> NULL
6: 6 -> 13 -> NULL
```

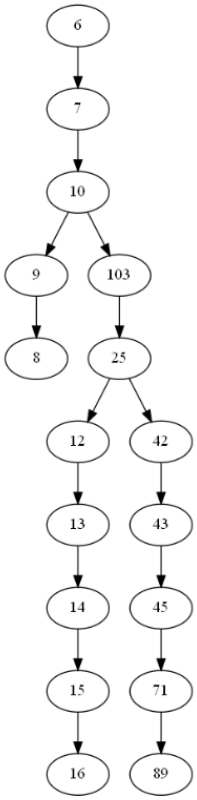
```
Введите целое число - значение элемента для добавления: 16
Добавление прошло успешно!
Для обычного дерева понадобилось: 10 сравн.
Для AVL-дерева понадобилось: 6 сравн.
Для хеш-таблицы понадобилось: 1 сравн.
```

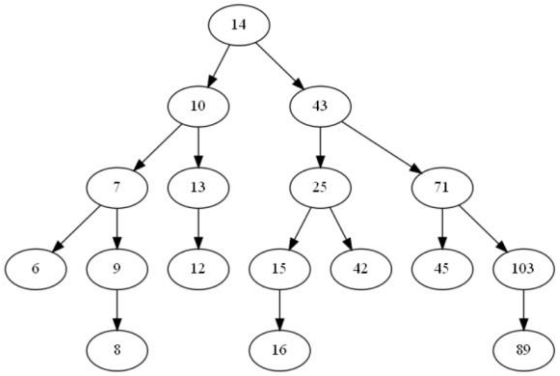



```
Hash key = 7
0: 7 -> 14 -> 42 -> NULL
1: 8 -> 15 -> 43 -> 71 -> NULL
2: 9 -> 16 -> NULL
3: 10 -> 45 -> NULL
4: 25 -> NULL
5: 12 -> 89 -> 103 -> NULL
6: 6 -> 13 -> NULL
```

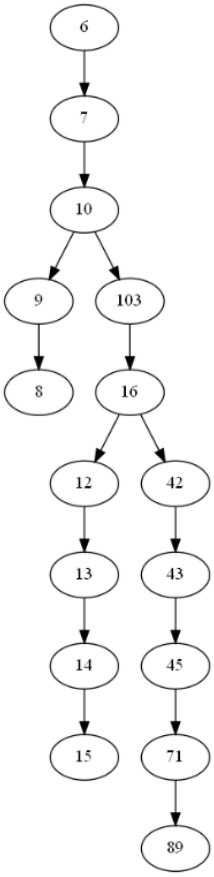
## 2. Обычное удаление

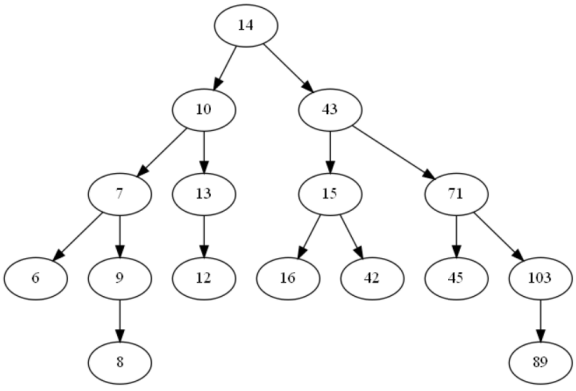



```
Hash key = 7
0: 7 -> 14 -> 42 -> NULL
1: 8 -> 15 -> 43 -> 71 -> NULL
2: 9 -> 16 -> NULL
3: 10 -> 45 -> NULL
4: 25 -> NULL
5: 12 -> 89 -> 103 -> NULL
6: 6 -> 13 -> NULL
```

```
Введите целое число - значение элемента для удаления: 25
Удаление прошло успешно!
Для обычного дерева понадобилось: 5 сравн.
Для AVL-дерева понадобилось: 3 сравн.
Для хеш-таблицы понадобилось: 1 сравн.
```



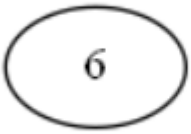



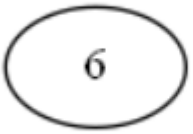

```
Hash key = 7
0: 7 -> 14 -> 42 -> NULL
1: 8 -> 15 -> 43 -> 71 -> NULL
2: 9 -> 16 -> NULL
3: 10 -> 45 -> NULL
4: NULL
5: 12 -> 89 -> 103 -> NULL
6: 6 -> 13 -> NULL
```

## 3. Добавление в пустые структуры

```
Hash key = 7
0: NULL
1: NULL
2: NULL
3: NULL
4: NULL
5: NULL
6: NULL
```

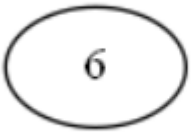
```
Введите целое число - значение элемента для добавления: 6
Добавление прошло успешно!
Для обычного дерева понадобилось: 1 сравн.
Для AVL-дерева понадобилось: 1 сравн.
Для хеш-таблицы понадобилось: 1 сравн.
```

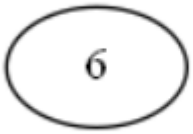



```
Hash key = 7
0: NULL
1: NULL
2: NULL
3: NULL
4: NULL
5: NULL
6: 6 -> NULL
```

## 4. Удаление единственного элемента




```
Hash key = 7
0: NULL
1: NULL
2: NULL
3: NULL
4: NULL
5: NULL
6: 6 -> NULL
```

```
Введите целое число - значение элемента для удаления: 6
Удаление прошло успешно!
Для обычного дерева понадобилось: 1 сравн.
Для AVL-дерева понадобилось: 1 сравн.
Для хеш-таблицы понадобилось: 1 сравн.
```

```
Hash key = 7
0: NULL
1: NULL
2: NULL
3: NULL
4: NULL
5: NULL
6: NULL
```

## 5. Реструктуризация при ручной смене ключа

```
Hash key = 19
0: NULL
1: 1 -> NULL
2: NULL
3: 3 -> NULL
4: 4 -> 42 -> NULL
5: 5 -> 43 -> NULL
6: 6 -> 25 -> NULL
7: 7 -> 45 -> NULL
8: 8 -> 103 -> NULL
9: 9 -> NULL
10: 10 -> NULL
11: NULL
12: 12 -> NULL
13: 13 -> 89 -> NULL
14: 14 -> 71 -> NULL
15: 15 -> 72 -> NULL
16: 73 -> NULL
17: NULL
18: NULL
```

```
Текущее значение ключа: 19
Введите новый ключ(целое положительное простое число < 100): 13
```

При выбранном ключе кол-во сравнений превысило заданное (3), поэтому ключ сменился повторно

```
Hash key = 17
0: NULL
1: 1 -> 103 -> NULL
2: NULL
3: 3 -> 71 -> NULL
4: 4 -> 72 -> 89 -> NULL
5: 5 -> 73 -> NULL
6: 6 -> NULL
7: 7 -> NULL
8: 8 -> 25 -> 42 -> NULL
9: 9 -> 43 -> NULL
10: 10 -> NULL
11: 45 -> NULL
12: 12 -> NULL
13: 13 -> NULL
14: 14 -> NULL
15: 15 -> NULL
16: NULL
```

6. Автоматическая реструктуризация при добавлении элемента

Установленное кол-во сравнений – 4

```
Hash key = 11
0: NULL
1: 1 -> 12 -> 45 -> 89 -> NULL
2: 13 -> NULL
3: 3 -> 14 -> 25 -> NULL
4: 4 -> 15 -> 103 -> NULL
5: 5 -> 71 -> NULL
6: 6 -> 72 -> NULL
7: 7 -> 73 -> NULL
8: 8 -> NULL
9: 9 -> 42 -> NULL
10: 10 -> 43 -> NULL
```

Добавили 100

```
Hash key = 13
0: 13 -> NULL
1: 1 -> 14 -> NULL
2: 15 -> NULL
3: 3 -> 42 -> NULL
4: 4 -> 43 -> NULL
5: 5 -> NULL
6: 6 -> 45 -> 71 -> NULL
7: 7 -> 72 -> NULL
8: 8 -> 73 -> NULL
9: 9 -> 100 -> NULL
10: 10 -> NULL
11: 89 -> NULL
12: 12 -> 25 -> 103 -> NULL
```

7. Реструктуризация при смене максимального кол-ва сравнений (изначально 20)

```
Hash key = 7
0: 7 -> 14 -> 42 -> NULL
1: 1 -> 8 -> 15 -> 43 -> 71 -> NULL
2: 9 -> 72 -> NULL
3: 3 -> 10 -> 45 -> 73 -> NULL
4: 4 -> 25 -> NULL
5: 5 -> 12 -> 89 -> 103 -> NULL
6: 6 -> 13 -> NULL
```

```
Введите количество сравнений, после превышения которых необходима
переструктуризация хэш-таблицы(целое положительное число < 100): 6
```

```
Hash key = 11
0: NULL
1: 1 -> 12 -> 45 -> 89 -> NULL
2: 13 -> NULL
3: 3 -> 14 -> 25 -> NULL
4: 4 -> 15 -> 103 -> NULL
5: 5 -> 71 -> NULL
6: 6 -> 72 -> NULL
7: 7 -> 73 -> NULL
8: 8 -> NULL
9: 9 -> 42 -> NULL
10: 10 -> 43 -> NULL
```

## 6. Сравнение эффективности

*Количество итераций 1000000*
**Время**

### Поиск элемента

| Количество элементов | Обычное дерево | AVL-дерево | Hash-таблица | Файл |
|---|---|---|---|---|
| 50 | 0.00000200 | 0.00000100 | 0.00000100 | 0.00001800 |
| 100 | 0.00000200 | 0.00000200 | 0.00000100 | 0.00001500 |
| 500 | 0.00001800 | 0.00001400 | 0.00000400 | 0.00005700 |
| 1000 | 0.00002500 | 0.00002700 | 0.00000700 | 0.00010700 |
| 5000 | 0.00021100 | 0.00014600 | 0.00003800 | 0.01293600 |

Занимаемая память

| Количество элементов | Обычное дерево | AVL-дерево | Hash-таблица |
|---|---|---|---|
| 50 | 1600 | 1200 | 40000 |
| 100 | 3200 | 2400 | 40000 |
| 500 | 16000 | 12000 | 40000 |
| 1000 | 32000 | 24000 | 40000 |
| 5000 | 160000 | 120000 | 40000 |

Теперь проанализируем время поиска для хеш-таблицы для разного количества максимальных сравнений.

Количество элементов 100, ключ - 7

| Кол-во сравнений | Время |
|---|---|
| 1 | 0.0000010000 |
| 10 | 0.0000010000 |
| 30 | 0.0000020000 |
| 50 | 0.0000010000 |
| 100 | 0.0000020000 |

**Выводы по проделанной работе**

Хеш-таблицы являются самыми эффективными по времени, но сильно проигрывают по памяти, если количество элементов не близко к максимальному (если близко начинают выигрывать). При небольшом количестве элементов AVL-дерево выигрывает у остальных по памяти и является более быстрым по сравнению с обычным деревом.

## Ответы на контрольные вопросы

### 1. Что такое дерево?

Дерево – структура данных, представляющая собой древовидную структуру в виде набора связанных узлов.

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

Память выделяется динамически при появлении нового узла.

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

Добавление/удаление элемента, разные варианты обхода и поиск.

### 4. Что такое дерево двоичного поиска?

Это деревья, которые отличаются от обычных тем, что имеют не более двух потомков и тем, что на них определено отношение порядка.

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

У идеально сбалансированного дерева (которое может получиться, если при построении дерева располагать узлы слева и справа), число вершин в левом и правом поддеревьях отличается не более чем на единицу.

АВЛ-дерево - дерева, у которого у каждого узла высота двух поддеревьев отличается не более чем на единицу.

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

При поиске в АВЛ-дереве происходит меньше сравнений.

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

Хэш-таблица - это массив элементами которого являются связные списки. Принцип их действия таков, что выбирается определенный ключ, с помощью которого, определяется в какую ячейку массива нужно добавлять элемент.

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

Коллизия - это ситуация, когда разным ключам соответствует одно значение хеш-функции.

Основные методы устранения: метод цепочек, метод закрытого хеширования.

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

При плохо подобранной хеш-функции, количество элементов в одной цепочке может стать очень большим, и поиск по ней будет осуществляться дольше. В этом случае будет необходима реструктуризация таблицы, то есть смена хэш-функции и заполнение хэш-функции с начала.

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

Дерево: O(h), где h – глубина дерева

АВЛ-дерево: O(logN)

Хеш–таблица: O(1)