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

Студент **Звягин Даниил Олегович**

Группа **ИУ7-33Б**

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

Студент _____ **Звягин Д.О.**

Преподаватель _____ **Барышникова М. Ю.**

Преподаватель _____ **Никульшина Т.А.**

Оценка _____

2023 г.

Условие задачи

Построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах. Сравнить эффективность устранения коллизий при внешнем и внутреннем хешировании

Техническое задание

Используя предыдущую программу (задача №6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из чисел файла. 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.

Входные данные:

Пользователю будет предложено меню.
Пункты меню:

0. Выход
1. Вывести дерево на экран
2. Выбрать файл для работы
3. Считать дерево из выбранного файла
4. Добавить число в дерево и в файл
5. Посчитать количество узлов на каждом уровне дерева
6. Вывести среднее время добавление числа в дерево
7. Удалить число из дерева
8. Преобразовать дерево в AVL
9. Вывести открытую хэш таблицу
10. Вывести закрытую хэш таблицу
11. Протестировать поиск числа

Выходные данные:

В зависимости от пункта меню: Изображение дерева на экране (в формате .png, сгенерированного с помощью graphviz); Информация об ошибках; Замеры времени по добавлению элементов в разные структуры;
Изображения открываются с помощью утилиты sxiv, т.к. это приложение, которое я использую для отображения изображений, команда может быть заменена на любой обработчик в 377 строке файла my_tree.c;
Хэш таблицы; Время поиска, количество сравнений и объём памяти, занимаемый разными структурами.

Возможные аварийные ситуации:
Нехватка памяти

Способ обращения к программе
Собрать программу с помощью make release (/debug);
Запустить программу с помощью файла app.exe;

Структуры данных

```
// перечисление пунктов меню
enum menu
{
	EXIT,
	SHOW_TREE,
	CHOOSE_FILE,
	PARSE_FILE,
	ADD_NUMBER,
	COUNT_NODES_ON_LEVELS,
	AVG_ADD_TIME,
	DELETE_NUMBER,
	MAKE_AVL,
	OPEN_HASH,
	CLOSED_HASH,
	FIND_NUMBER,
};

// перечисление ошибок при работе с деревом
enum tree_errors
{
	TREE_NO_MEMORY = 1,
	TREE_BAD_DOT,
};

// тип данных узла дерева
typedef struct tree_node *tree_node_t;
struct tree_node
{
	void *data;
	size_t count;
	struct tree_node *parent;
	struct tree_node *left;
	struct tree_node *right;
};

// структура данных дерева (оформлено в видео АТД)
typedef struct tree *tree_t;
struct tree
{
	tree_node_t root;
```

```
	compataror_t cmp;
	const char *format;
};


#define MAX_COLLISIONS 4

// Структура данных хэш таблицы с открытым хэшированием
typedef struct open_hash *open_hash_t;
struct open_hash
{
	node_t **data;
	int datalen;
};

// Структура данных хэш таблицы с закрытым хэшированием
typedef struct closed_hash *closed_hash_t;
struct closed_hash
{
	int *data;
	int datalen;
};
```

Принцип заполнения хэш-таблиц

Хэш-функция

```
static size_t hash_get_index(int num, int m)
{
	return (size_t)((num % m + m) % m);
}
```

Функция принимает число и размерность хэш-таблицы, возвращает остаток от деления числа на размерность; Остаток - математический (то есть всегда больше 0)

Реструктуризация:
Для обоих типов таблиц:
Получаем новую размерность (у меня - следующее простое число)
Проходим по всей старой таблице и добавляем её элементы в новую, затем старую таблицу удаляем, а новую ставим на её место.

Добавление:
Для хэш-таблицы с открытым хэшированием:
Получаем индекс элемента с помощью хэш-функции
Получаем указатель на начало списка данной ячейки (или NULL - тогда он пустой)
Если число уже в списке, ничего не делаем
Если длина списка больше лимита коллизий, происходит реструктуризация таблицы, а затем алгоритм добавления начинается сначала,

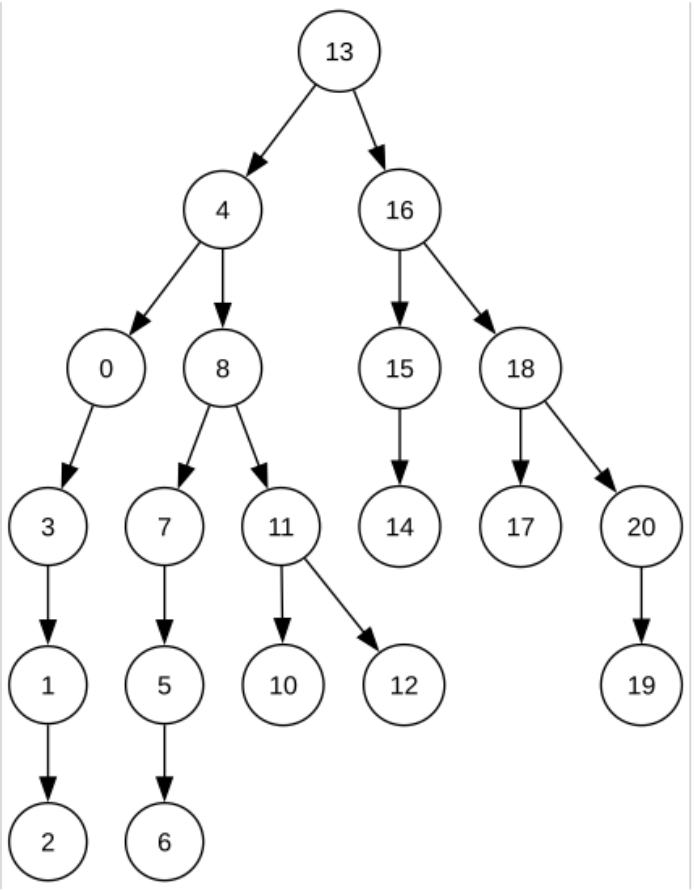
Иначе - в конец списка добавляется узел с данным значением

Для хэш-таблицы с закрытым хэшированием:
Если дан 0, считается, что он уже записан

Получаем индекс элемента с помощью хэш-функции
Начинаем цикл с i от 0 до лимита коллизий -1
смотрим на число в ячейке с индексом (index + i * i) % m, где m - размерность таблицы
Если оно совпало с данным, ничего не делаем
Иначе если в ячейке 0 и номер ячейки не 0 - записываем данное число
Иначе, если цикл завершился и число не было записано, производим реструктуризацию и начинаем алгоритм добавления заново.

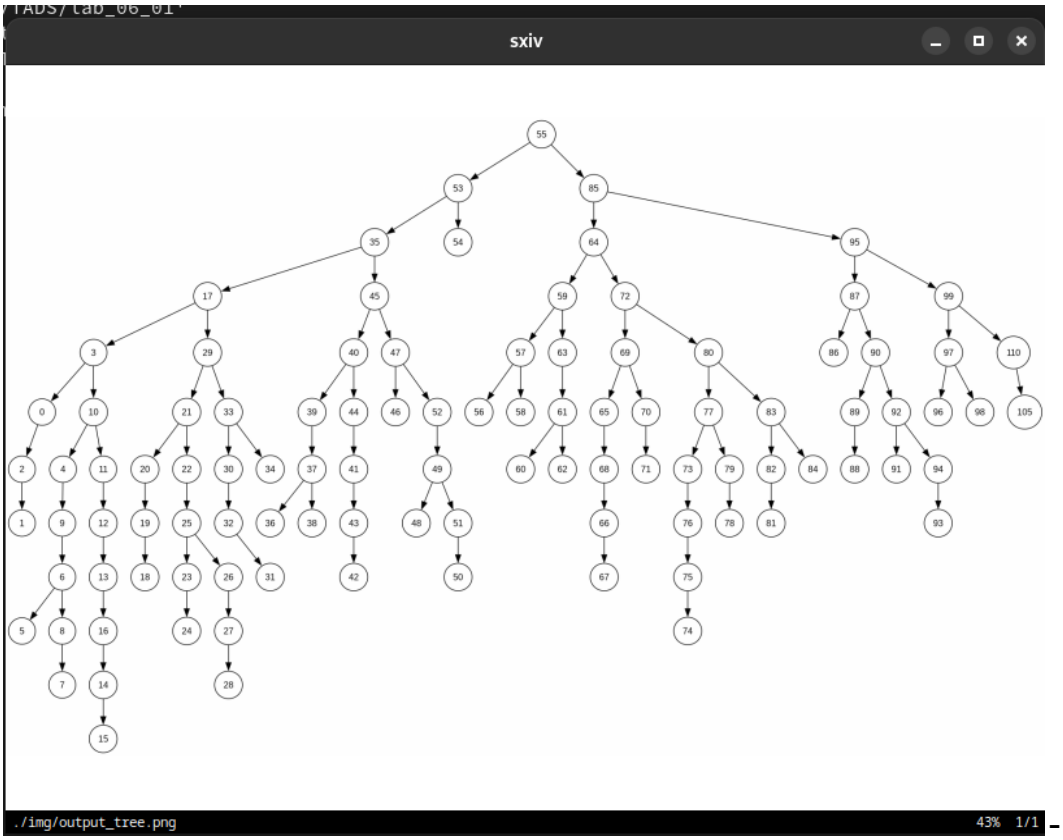
Пример работы

Обход дерева:
Я использую обход в глубину влево (префиксно), то есть для дерева на рисунке ниже порядок будет следующим:



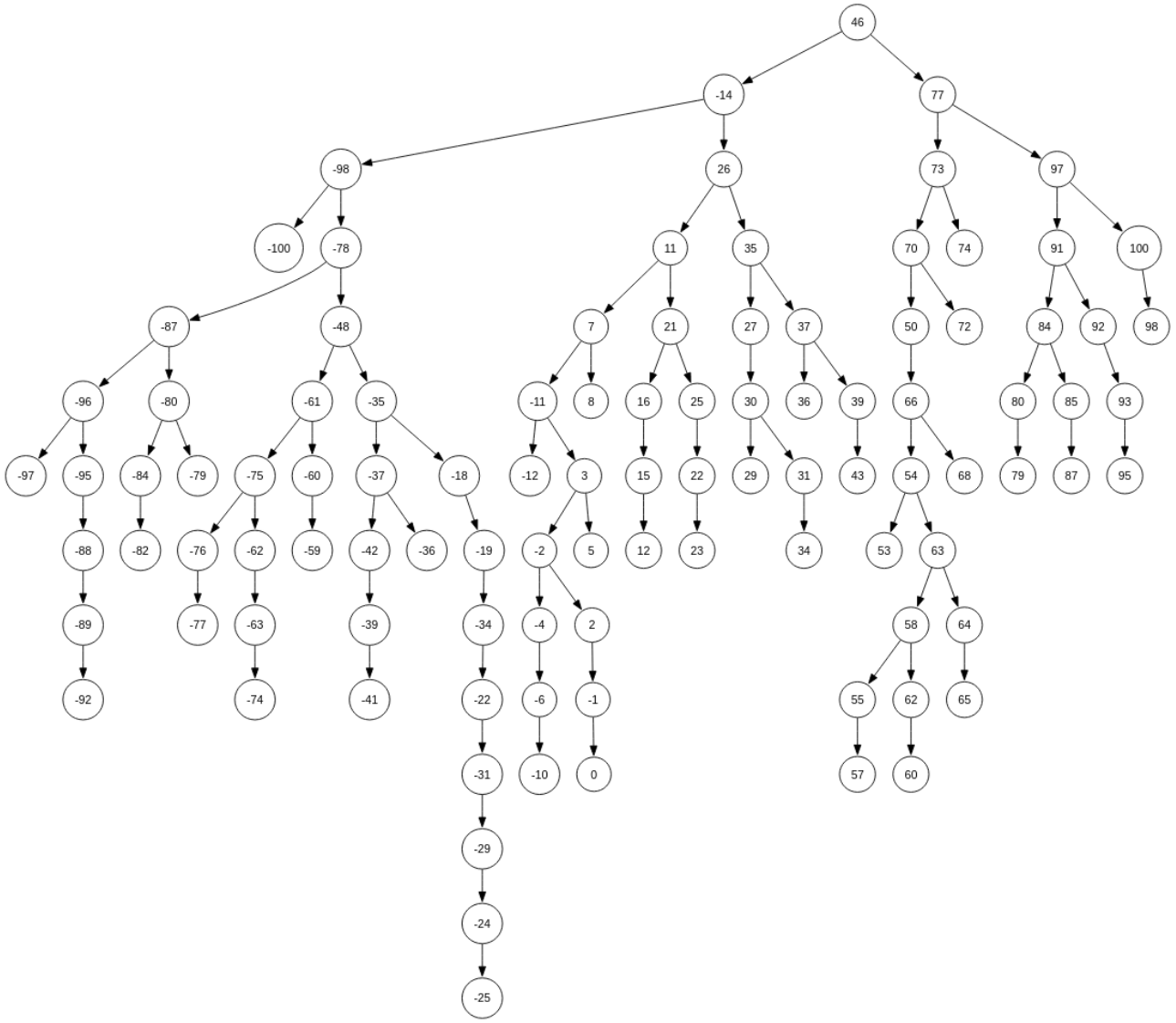
13, 4, 0, 3, 1, 2, 8, 7, 5, 6, 11, 10, 12, 16, 15, 14, 18, 17, 20, 19

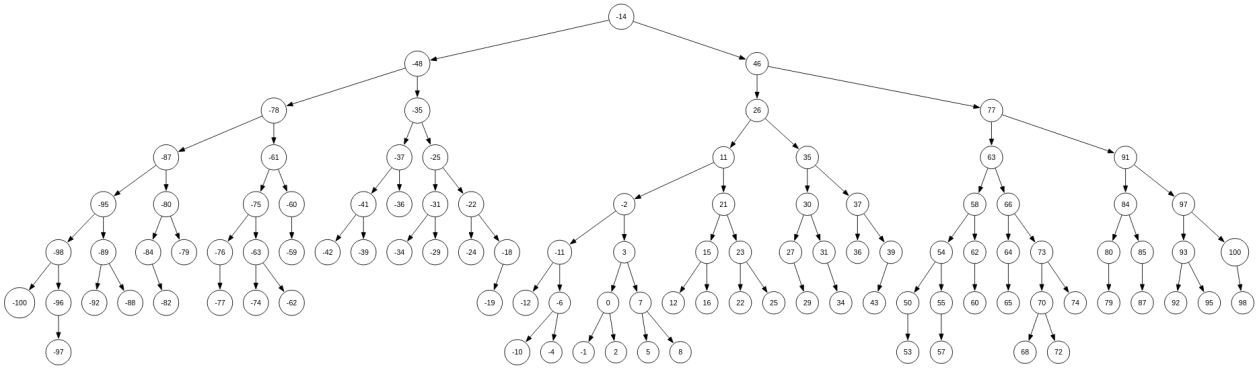
Пример работы программы:


sxiv
55
53
85
35
54
64
95
17
45
59
72
87
99
3
29
40
47
57
63
69
80
86
90
97
110
0
10
21
33
39
44
46
52
56
58
61
65
70
77
83
89
92
96
98
105
2
4
11
20
22
30
34
37
41
49
60
62
68
71
73
79
82
84
88
91
94
1
9
12
19
25
32
36
38
43
48
51
66
76
78
81
93
6
13
18
23
26
31
42
50
67
75
5
8
16
24
27
74
7
14
28
15
./img/output_tree.png
43% 1/1

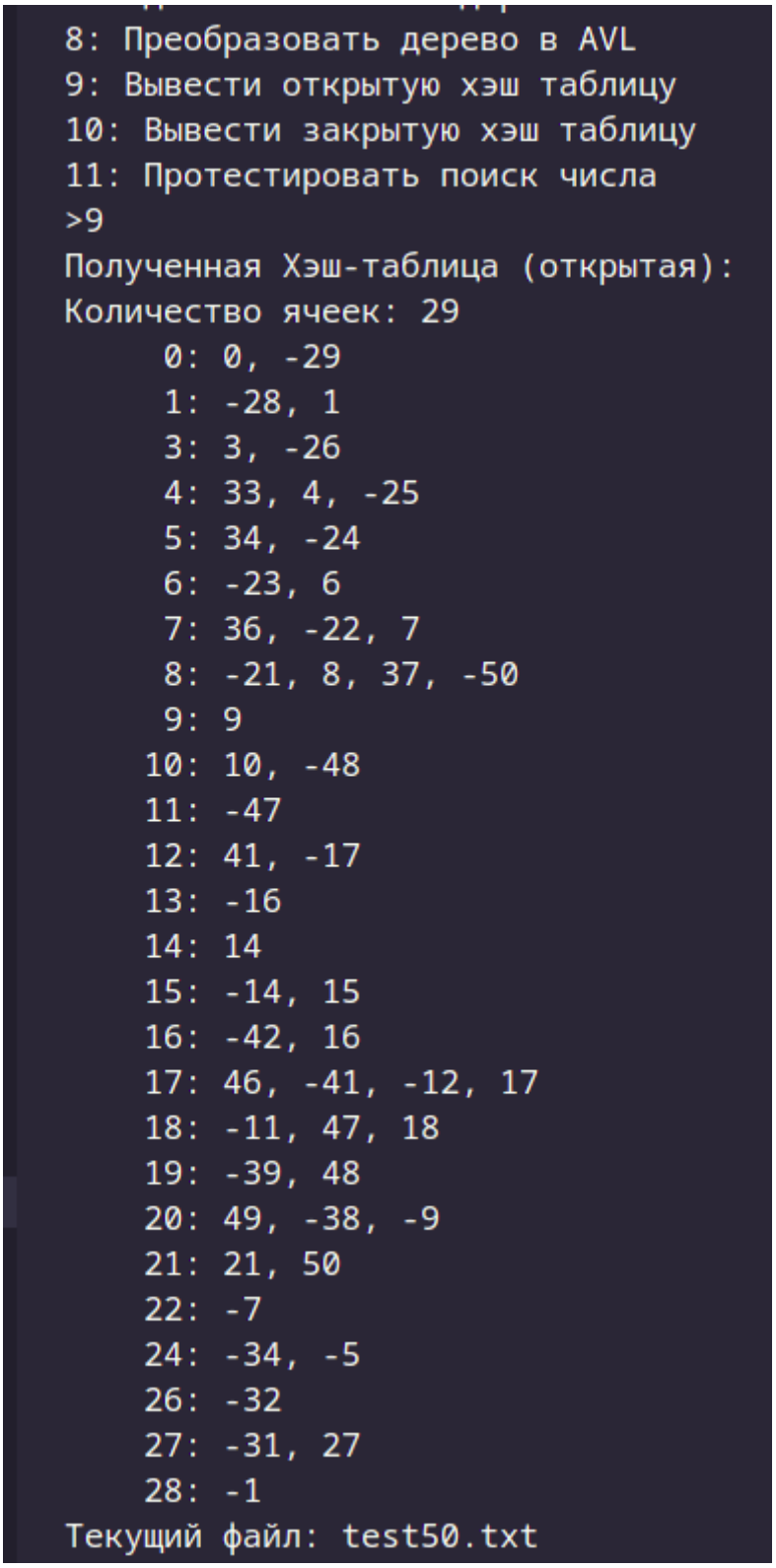
```
3: Считать дерево из файла
4: Добавить число в дерево
5: Посчитать количество узлов дерева на каждом уровне
6: Вывести среднее время добавления числа в дерево
>2
Введите название файла
>test100pos.txt
Текущий файл: test100pos.txt
Меню:
0: Выход
1: Вывести дерево на экран
2: Выбрать файл для работы
3: Считать дерево из файла
4: Добавить число в дерево
5: Посчитать количество узлов дерева на каждом уровне
6: Вывести среднее время добавления числа в дерево
>3
Всего считано 102 чисел
Текущий файл: test100pos.txt
Меню:
0: Выход
1: Вывести дерево на экран
2: Выбрать файл для работы
3: Считать дерево из файла
4: Добавить число в дерево
5: Посчитать количество узлов дерева на каждом уровне
6: Вывести среднее время добавления числа в дерево
>1
Текущий файл: test100pos.txt
Меню:
0: Выход
1: Вывести дерево на экран
2: Выбрать файл для работы
3: Считать дерево из файла
4: Добавить число в дерево
5: Посчитать количество узлов дерева на каждом уровне
6: Вывести среднее время добавления числа в дерево
>5
На слое 0 : 1
На слое 1 : 2
На слое 2 : 4
На слое 3 : 6
На слое 4 : 12
На слое 5 : 20
На слое 6 : 21
На слое 7 : 16
На слое 8 : 10
На слое 9 : 6
На слое 10 : 3
На слое 11 : 1
```

Преобразование дерева в AVL-дерево (первое изображение - обычное, второе - AVL)





Открытая хэш-таблица для файла с 50 числами



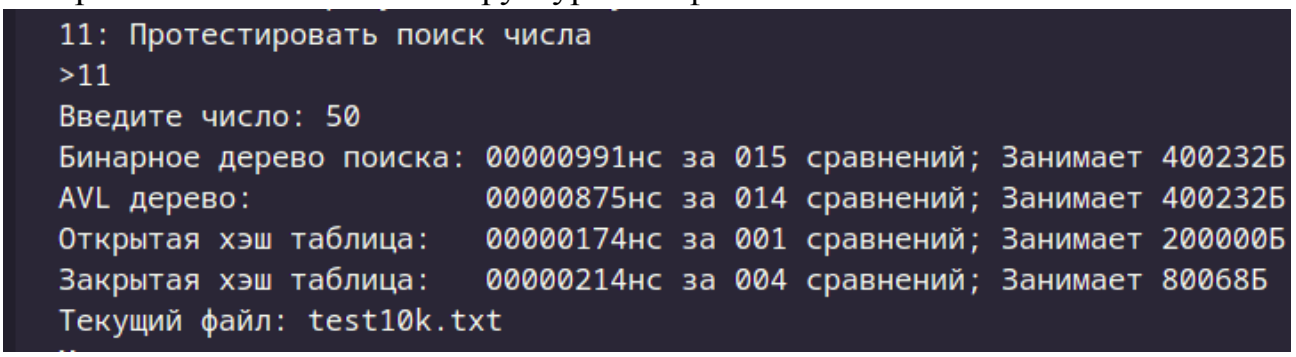
Закрытая хэш-таблица для файла с 50 числами

```
4: Добавить число в дерево
5: Посчитать количество узлов дерева на каждом уровне
6: Вывести среднее время добавления числа в дерево
7: Удалить число из дерева
8: Преобразовать дерево в AVL
9: Вывести открытую хэш таблицу
10: Вывести закрытую хэш таблицу
11: Протестировать поиск числа
>10
Полученная Хэш-таблица (закрытая):
Количество ячеек: 89
    0: 0
    1: 1
    3: 3
    4: 4
    6: 6
    7: 7
    8: 8
    9: 9
   10: 10
   14: 14
   15: 15
   16: 16
   17: 17
   18: 18
   21: 21
   27: 27
   33: 33
   34: 34
   36: 36
   37: 37
   39: -50
   41: -48
   42: -47
   45: 41
   46: 46
   47: -42
   37: 37
   39: -50
   41: -48
   42: -47
   45: 41
   46: 46
   47: -42
   48: -41
   49: 49
   50: -39
   51: -38
   52: 48
   54: 50
   55: -34
   56: 47
   57: -32
   58: -31
   60: -29
   61: -28
   63: -26
   64: -25
   65: -24
   66: -23
   67: -22
   68: -21
   72: -17
   73: -16
   75: -14
   77: -12
   78: -11
   80: -9
   82: -7
   84: -5
   88: -1
Текущий файл: test50.txt
Меню:
0: Выход
```

Замеры поиска элемента по структурам из файла с 50 элементами

```
11: Протестировать поиск числа
>11
Введите число: 35
Бинарное дерево поиска: 00000507нс за 008 сравнений; Занимает 2152Б
AVL дерево:             00000348нс за 006 сравнений; Занимает 2152Б
Открытая хэш таблица:   00000347нс за 002 сравнений; Занимает 1104Б
Закрытая хэш таблица:   00000280нс за 004 сравнений; Занимает 380Б
Текущий файл: test50.txt
```

Замеры поиска элемента по структурам из файла с 10 тысячами элементов



Замеры

Сравним скорость поиска элемента по деревьям
Скорость поиска элементов в AVL деревьях всегда не дольше поиска по Бинарному дереву (что можно видеть и по сложности поиска в последнем контрольном вопросе), однако для моих тестов прирост в скорости значителен

Скорость поиска по хэш-таблицам гораздо быстрее скорости по деревьям и хэш таблицы занимают гораздо меньше места (из-за того, что каждый узел дерева гораздо больше даже узла списка)
Если сравнивать их между собой, то поиск по хэш-таблице с закрытым хэшированием происходит стабильно быстрее, даже несмотря на коллизии, так как проход по элементам списка занимает некоторое время.

Выводы по проделанной работе

Для быстрого поиска данных можно использовать такие структуры, как деревья поиска и хэш-таблицы.
Деревья являются хорошим вариантом поиска, но они занимают (сравнительно) большое количество памяти и поиск по ним происходит с минимальной сложностью log(n)

Недостаток AVL дерева в сравнении с деревом поиска заключается в том, что добавление элементов в него требует балансировки дерева, что сильно замедляет добавление элементов и усложняет программу.

Для ещё более быстрого поиска данных можно использовать хэш-таблицы, которые позволяют не только ускорить поиск, но и уменьшить количество занимаемой памяти.

В хэш-таблицах могут возникать коллизии, для избавления от которых можно использовать списки или алгоритмы поиска новых мест в массиве.

Хэш-таблица, выполненная с помощью массива (с закрытым хэшированием) выигрывает как по памяти (даже если размер массива больше, int занимает гораздо меньше места, чем узел списка), так и по скорости (за счёт свойств массива (однородность и т.д.))

Поэтому для хранения данных, в которых нужно будет выполнять быстрый поиск, выгоднее всего использовать хэш-таблицы с закрытым хэширванием, единственный недостаток которых - необходимость в реструктуризации при большом количестве коллизий.

Контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?
Идеально сбалансированное дерево заполняется путём равномерного распределения вершин по ветвям и, в отличие от АВЛ дерева не является отсортированным (то есть не (по крайней мере не всегда) является деревом поиска)

2. Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?
Алгоритм поиска - ничем. Просто в среднем сравнений придётся произвести меньше.

3. Что такое хеш-таблица, каков принцип ее построения?
Массив, позиция элементов в котором определяется хэш-функцией, которая позволяет определить позицию каждого элемента.

4. Что такое коллизии? Каковы методы их устранения.
Коллизия - это ситуация, когда хэш-функция присваивает двум разным элементам одинаковый индекс. В зависимости от того, закрытая или открытая хэш-таблица, коллизии устраняются
*для первого случая - поиском новой позиции чуть дальше (index + i*i), где i - номер попытки поиска нового места.*

для второго случая - записью элемента в конец списка, который содержится в ячейке по хэш индексу.

5. В каком случае поиск в хеш-таблицах становится неэффективен?
В случаях, когда количество коллизий становится слишком большим. В таких случаях хэш-таблицу обычно реструктуризируют.

6. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хеш-таблицах и в файле.
В хэш таблицах минимальное - O(1)
В АВЛ - O(log n)
В Двоичном дереве поиска - O(h), где h от log n до n