Министерство образования и науки 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 (ИУ)

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии (ИУ7)

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

«Обработка деревьев, хеш-функций»

Студент, группа **ИУ7-33Б** **Рядинский К.В.,**

2020 г.

Описание условия задачи

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 закрытое хеширование для устранения коллизий. Осуществить удаление введенного целого числа в ДДП, в сбалансированном дерево, в хеш-таблице и в файле. Сравнить время удаления, объем памяти и количество сравнений при использовании различных (4-х) структур данных. Если количество сравнений в хеш-таблице больше указанного, то произвести реструктуризацию таблицы, выбрав другую функцию.

Техническое задание

Входные данные:

1. Имя файла, содержащего числа
2. Максимальное значение допустимых коллизий — число
3. Число для поиска

Выходные данные

1. Псевдографическое изображение дерева
2. Хэш-таблица
3. Сравнение эффективности поиска в различных структурах данных (деревья, хэш-таблица).

Функции программы

1. Вывод бинарного дерева
2. Вывод сбалансированного дерева
3. Вывод хэш-таблицы
4. Сравнение эффективности

Обращение к программе

запускается из терминала с аргументом в виде файла, содержащего числа

Аварийные случаи

1. Ввод несуществующего файла
2. Ввод пустого или содержащего некорректные данные (буквы, вещественные числа) файла
3. Некорректный ввод максимально допустимого кол-ва коллизий (буквы, отрицательные числа)

Структуры данных

Typedef struct tnode_s
{
	int data;
	tnode_t *left; — левое поддерево
	tnode_t *right; — правое поддерево
	Int fact; — разница высот левого и правого поддерева
} tnode_t;

Typedef struct hashelem_s
{
	int val — значение элемента
	int free; — свободна ли ячейка
} hashelem_t;

Typedef struct hashtable_s
{
	hashelem_t *arr; — массив
	unsigned len len; — длинна массива
} hashtable_t;

Структура программы

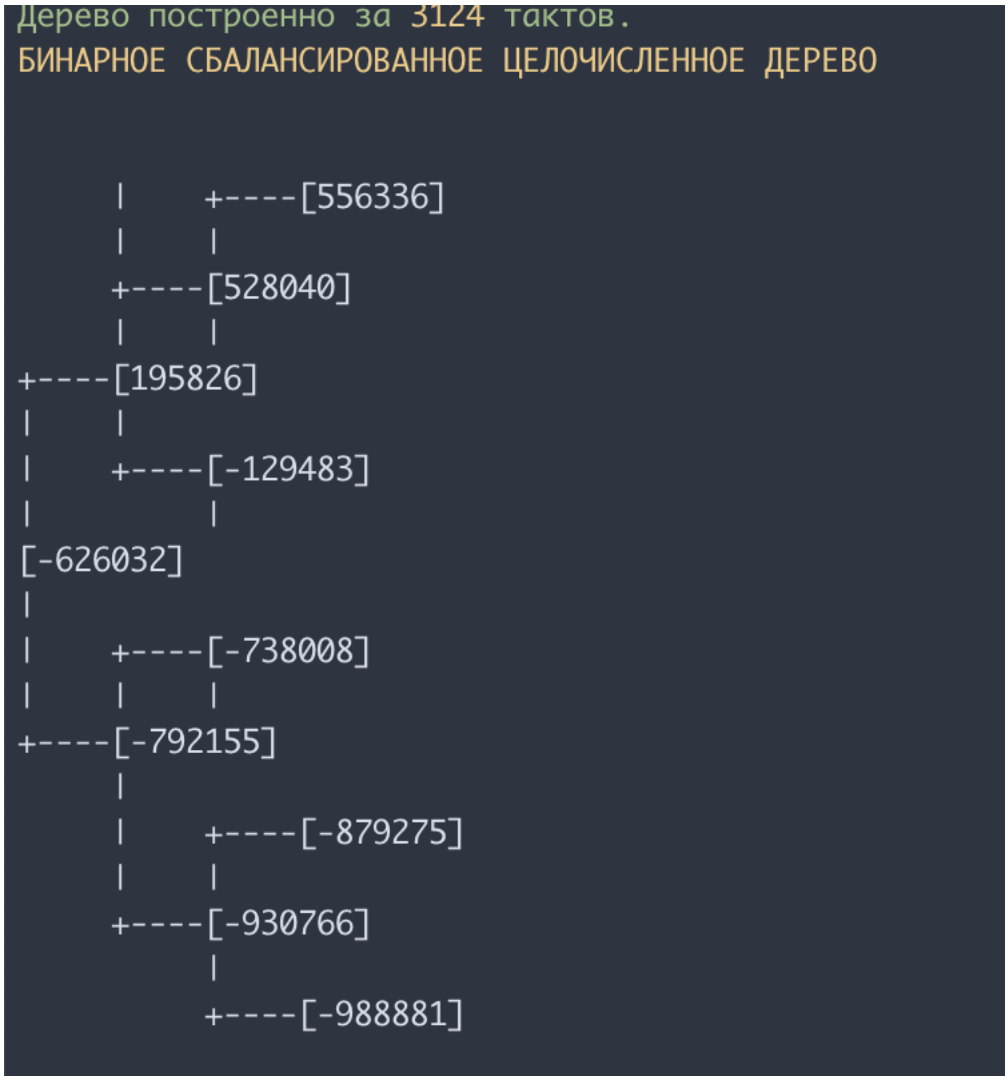
int hash_insert(hashtable_t *ht, const int val, const hashtype_t type); — вставка в хэш таблицу
int hash_find(const hashtable_t *ht, const int key, const hashtype_t type); — поиск в хэш-таблице

hashtable_t *hash_init(unsigned int len); — инициализация таблицы
void hash_clean(hashtable_t **ht); — очистка таблицы

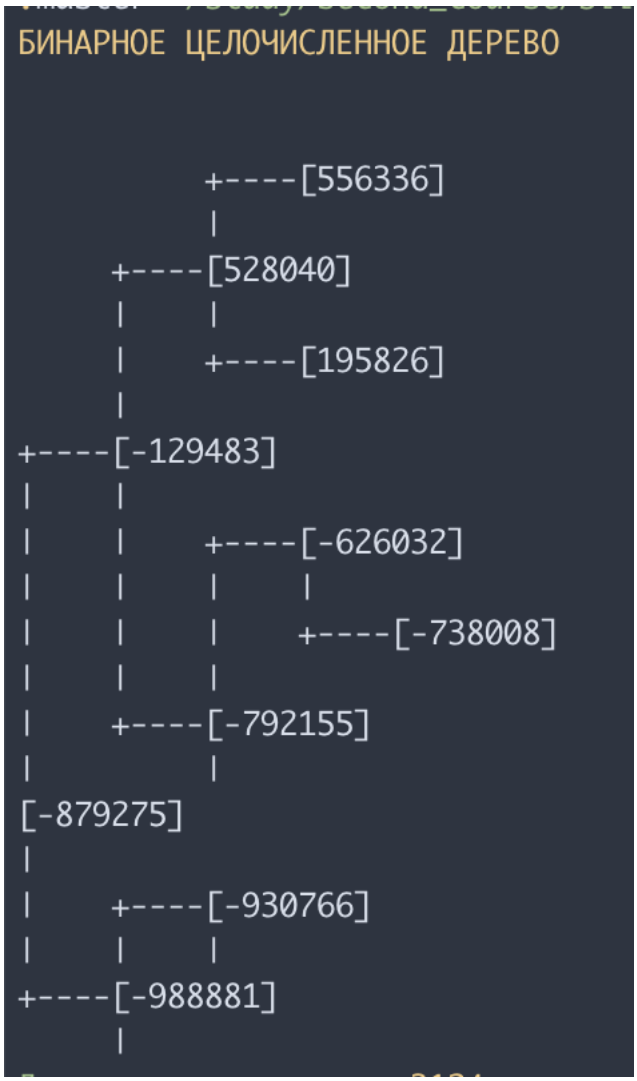
tnode_t *tree_insert(tnode_t *tree, int val); — вставка элемента в дерево
tnode_t *tree_insert_b(tnode_t *tree, int val); — сбалансированная вставка в дерево
tnode_t *init_node(int val); — инициализация узла дерева
void tree_clean(tnode_t *tree); — очистка дерева

tnode_t *tree_find(tnode_t *tree, int key); — поиск в дереве

Визуализация



БИНАРНОЕ ЦЕЛОЧИСЛЕННОЕ ДЕРЕВО



ХЭШ	ДАННЫЕ
0	556336
1	-988881
2	-129483
3	-879275
4	-792155
5	-626032
6	195826
7	528040
8	-738008
9	-930766

Алгоритм

Сначала происходит чтение из заданного файла в массив, после этого строится бинарное дерево из чисел, считанных из массива. Далее дерево балансируется. Далее строится хэш-таблица на простой хэш функции. Далее у пользователя запрашивается максимально допустимое число коллизий, если числа коллизий у текущей хэш таблицы больше, то она перестраивается по другой хэш функции. После этого выводятся результаты проверки эффективности поиска по заданному числу в разных структурах данных.

Реструктуризация таблицы проводится по следующему алгоритму:

1. Выбирается новая «сложная» функция
2. Если со «сложной» функцией кол-во коллизий не удовлетворяет условию, то размер таблицы увеличивается (берется следующее простое число)

Было проверено несколько сложных хэш-функций

Первая

```
52
53
54      key = val;
55
56      key += ~(key << 16);
57      key ^= (key >> 5);
58      key += (key << 3);
59      key ^= (key >> 13);
60      key += ~(key << 9);
61      key ^= (key >> 17);
62
63
64      return key % len;
65  }
66
```

Вторая

```
{
    int key = abs(val);
    int N = 1024 * 1024 * 128;
    double A = 0.618033;
    int h = N * fmod(key * A , 1);
```

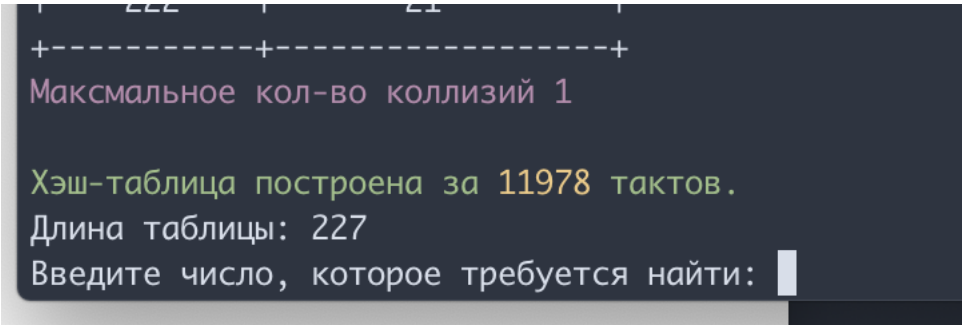
Третья

```
{
    unsigned int new = val;

    return new % len;
}
```

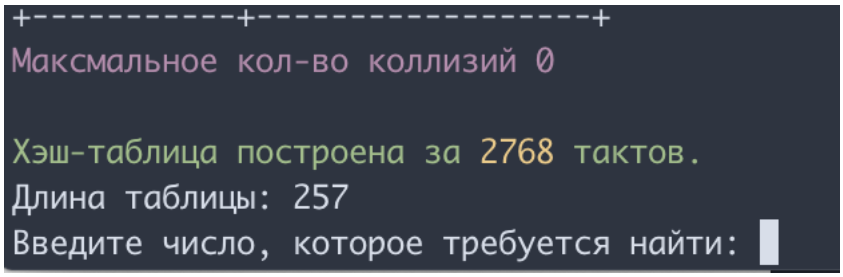
Для первой функции для того, чтобы добиться 0 кол-во коллизий для файла из 100 элементов с разбросом от -100 до 100, потребовалась длина 1109

```
+-----------+------------------+
Максмальное кол-во коллизий 0

Хэш-таблица построена за 2458 тактов.
Длина таблицы: 1109
Введите число, которое требуется найти:
```

Для второй функции для того, чтобы добиться 1 (так как 0 за адекватное время добиться не получилось) кол-во коллизий для файла из 100 элементов с разбросом от -100 до 100, потребовалась длина 227



Для третьей функции для того, чтобы добиться 0 кол-во коллизий для файла из 100 элементов с разбросом от -100 до 100, потребовалась длина 1109



Таким образом, самой эффективной функцией оказалась самая простая функция. Третья функция позволяет достичь нуля коллизий за меньшее кол-во элементов и работает она достаточно быстро.

Тесты

	Тест	Ввод	Вывод
1	Несуществующий файл	test.test	Ошибка, неправильное имя файл
2	Пустой файл	empty.empty	Ошибка, пустой файл
3	Файл содержит не только целые числа	badfile.txt	Ошибка чтения из файла.

4	Некорректный ввод коллизий	-3	Ошибка, введено неправильное кол-во коллизий

Оценка эффективности

Поиск числа (в тактах процессора) (Среднее по всему файлу)

Кол-во элементов	Бинарное дерево	АВЛ дерево	Хэш-таблица	Файл
10	135	125	57	5606
100	114	88	39	33429
1000	135	126	33	165643
10000	226	211	60	1628041

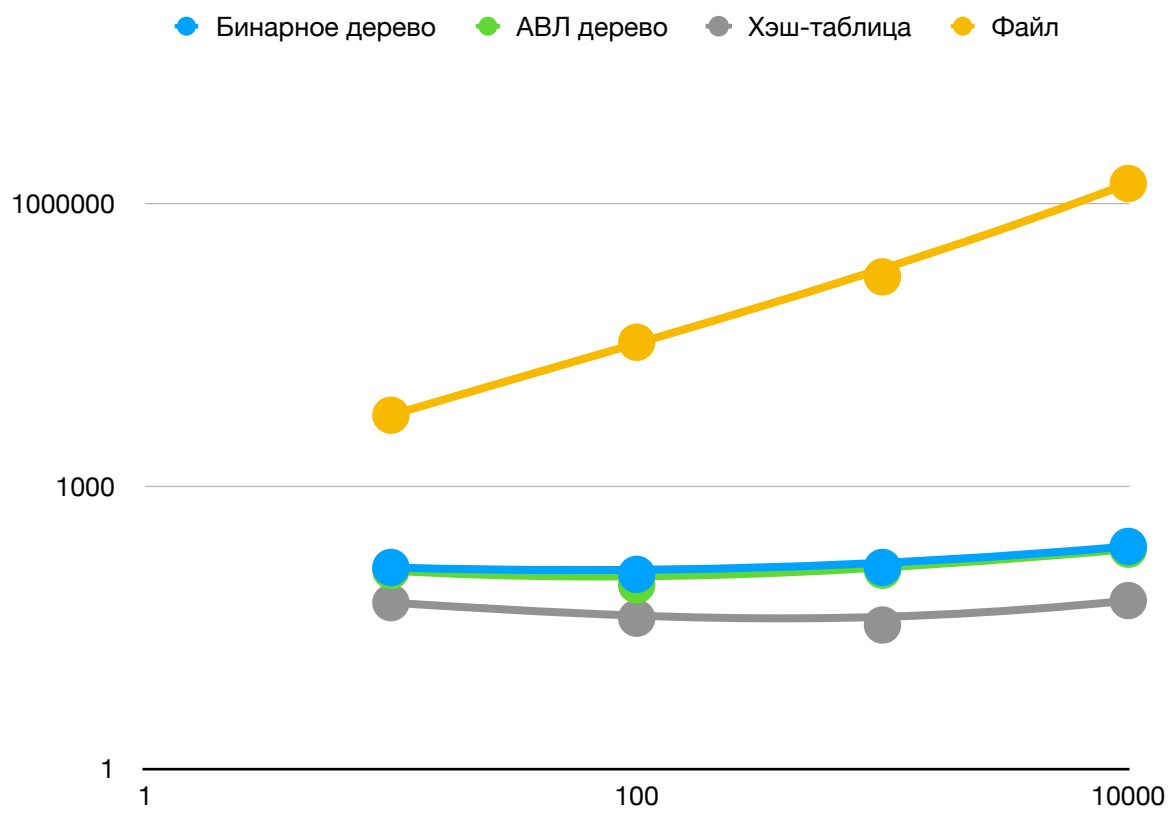
Объем занимаемой памяти (Байты)

Кол-во элементов	Бинарное дерево	АВЛ дерево	Хэш-таблица	Файл
10	320	320	1320	77
100	3200	3200	3368	335
1000	32000	32000	18152	4392
10000	320000	320000	2070792	73834

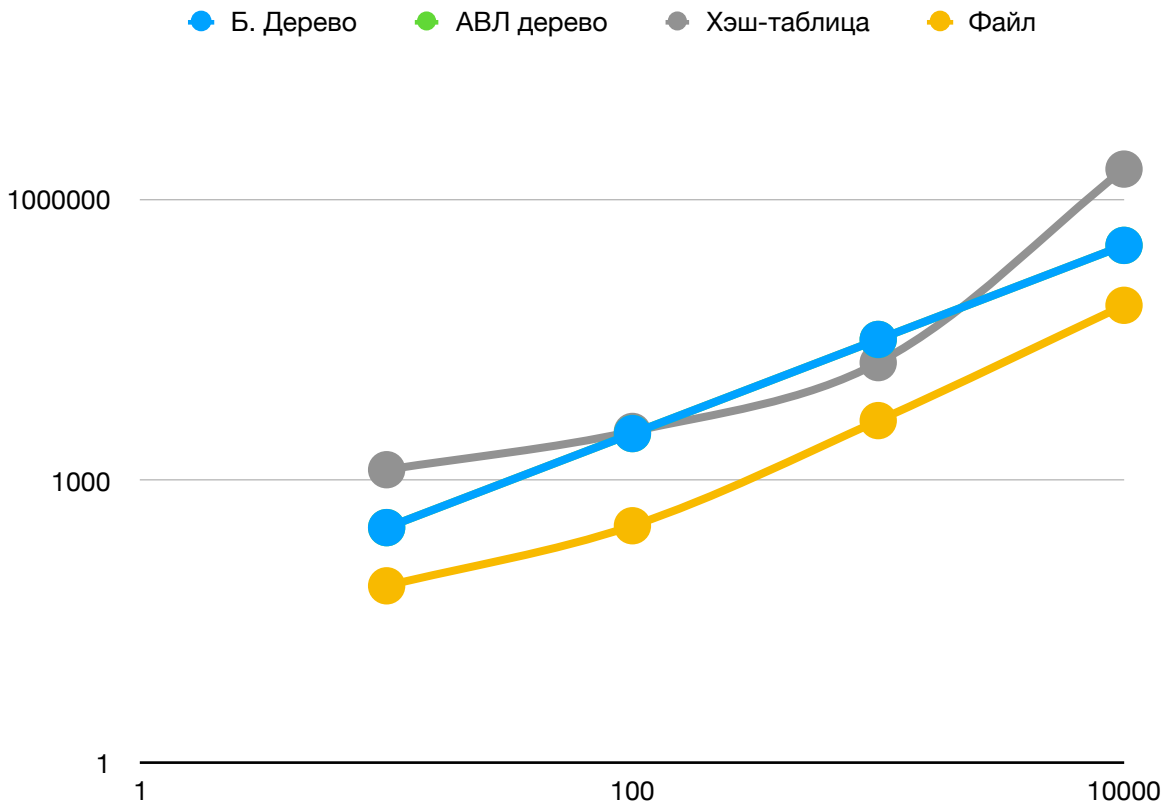
Кол-во сравнений

Кол-во элементов	Бинарное дерево	АВЛ дерево	Хэш-таблица	Файл
10	5	3	1	10
25	8	4	1	25
50	10	5	1	500
100	12	6	1	100

Время поиска



Объем памяти



Контрольные вопросы

1. Что такое дерево?

Дерево – это рекурсивная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».

2. Как выделяется память под представление деревьев?

В виде связного списка — динамически под каждый узел.

3. Какие стандартные операции возможны над деревьями?

Обход дерева, поиск по дереву, включение в дерево, исключение из дерева.

4. Что такое дерево двоичного поиска?
Двоичное дерево поиска - двоичное дерево, для каждого узла которого сохраняется условие: левый потомок больше или равен родителю, правый потомок строго меньше родителя (либо наоборот).

5. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

У АВЛ дерева для каждой его вершины высота двух её поддеревьев различается не более чем на 1, а у идеально сбалансированного дерева различается количество вершин в каждом поддереве не более чем на 1.

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

Поиск в АВЛ дереве происходит быстрее, чем в ДДП.

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

Хеш-таблицей называется массив, заполненный элементами в порядке, определяемом хеш-функцией. Хеш-функция каждому элементу таблицы ставит в соответствие некоторый индекс. Функция должна быть простой для вычисления, распределять ключи в таблице равномерно и давать минимум коллизий.

8. Что такое коллизии? Каковы методы их устранения?

Коллизия – ситуация, когда разным ключам хеш-функция ставит в соответствие один и тот же индекс. Основные методы устранения коллизий:
открытое и закрытое хеширование. При открытом хешировании к ячейке по данному ключу прибавляется связанны список, при закрытом – новый элемент кладется в ближайшую свободную ячейку после данной.

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

Поиск в хеш-таблице становится неэффективен при большом числе коллизий – сложность поиска возрастает по сравнению с O(1). В этом случае требуется реструктуризация таблицы – заполнение её с использованием новой хеш-функции.

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

В хеш-таблице минимальное время поиска O(1). В АВЛ: O(log2n). В дереве двоичного поиска O(h), где h - высота дерева (от log2n до n).

Вывод

Использование хеш-таблицы всегда эффективно по времени, но не всегда эффективно по памяти (в случае хорошей дистрибуции функции распределение будет не плотным), так как требует выделенной памяти под каждый хеш (если отсутствуют коллизии, хорошая дистрибуция). В случае деревьев, АВЛ дерево не всегда выигрывает по времени поиска у несбалансированного дерева, так как порядок вершин при балансировке меняется, но всегда выигрывает по среднему значению количества сравнений и среднему времени поиска по дереву.