Лабораторная работа №7 по дисциплине “Типы и структуры данных”

Подготовил: Ширяев Алексей

Группа: ИУ7-33Б

Цель работы — построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска, в хештаблицах и в файлах. Сравнить эффективность реструктуризации таблицы для устранения коллизий и поиска в ней с эффективностью поиска в исходной таблице.

Условие задачи

Вариант 3

Построить хеш-таблицу и AVL-дерево 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-таблицы. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если количество сравнений при поиске/добавлении 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.

Построить дерево поиска из слов текстового файла (задача №6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Удалить все слова, начинающиеся на указанную букву, в исходном и сбалансированном дереве. Построить хеш-таблицу из слов текстового файла. Вывести построенную таблицу слов на экран. Осуществить поиск и удаление введенного слова. Выполнить программу для различных размерностей таблицы и сравнить время удаления, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц.

ПРИМЕЧАНИЕ! В отчёте будут использоваться обозначения для действий:

<usual read> - Чтение дерева из файла

<avl copy> - Глубокое копирование из обычного дерева в сбалансированное

<usual add>/<avl add> - Добавление элемента в дерево

<usual remove>/<avl remove> - Удаление элемента из дерева по ключу

<usual search>/<avl search> - Поиск дерева по ключу

<usual pre_order_output>/<avl pre_order_output> - Префиксный обход дерева и создание файла с визуализацией дерева

<usual in_order_output>/<avl in_order_output> - Инфиксный обход дерева и создание файла с визуализацией дерева

<usual post_order_output>/<avl post_order_output> - Постфиксный обход дерева и

создание файла с визуализацией дерева

<usual find>/<avl find> - Нахождение элементов, начинающихся на букву и их удаление

<open limit>/<close limit> - Изменения лимита до реструктуризации

<open read>/<close read> - Чтение хэш-таблицы из файла

<open add>/<close add> - Добавление элемента в хэш-таблицу

<open remove>/<close remove> - Удаление элемента из хэш-таблицы по ключу

<open search>/<close search> - Поиск хэш-таблицы по ключу

<open output>/<close output> - Вывод хэш-таблицы

<stat> - Вывод статистики

<exit>- Выход из программы

(ПРИМЕЧАНИЕ! Под usual деревом подразумевается двоичное дерево поиска, в вершинах которого слова.)

Описание техзадачи

Описание исходных данных

Данные на входе: Меню. Код действия. Далее для каждого действия:

<usual read> - Файл с данными

<avl copy> - Usual-дерево

<usual add>/<avl add> - Usual-дерево/AVL-дерево, Ключ

<usual remove>/<avl remove> - Usual-дерево/AVL-дерево, Ключ

<usual search>/<avl search> - Usual-дерево/AVL-дерево, Ключ

<usual pre_order_output>/<avl pre_order_output> - Usual-дерево/AVL-дерево

<usual in_order_output>/<avl in_order_output> - Usual-дерево/AVL-дерево

<usual post_order_output>/<avl post_order_output> - Usual-дерево/AVL-дерево

<usual find>/<avl find> - Usual-дерево/AVL-дерево, ключ

<open limit>/<close limit> - Новое значение предела до реструктизацией

<open read>/<close read> - Файл с данными

<open add>/<close add> - Хэш-таблица, ключ

<open remove>/<close remove> - Хэш-таблица, ключ

<open search>/<close search> - Хэш-таблица, ключ

<open output>/<close output> - Хэш-таблица

<stat> - -

<exit>- Выход из программы

Данные на выходе:

<usual read> - Usual-дерево

<avl copy> - AVL-дерево

<usual add>/<avl add> - Usual-дерево/AVL-дерево

<usual remove>/<avl remove> - Usual-дерево/AVL-дерево

<usual search>/<avl search> - Usual-дерево/AVL-дерево

<usual pre_order_output>/<avl pre_order_output> - Usual-дерево/AVL-дерево, изображение-визуализация дерева

<usual in_order_output>/<avl in_order_output> - Usual-дерево/AVL-дерево, изображение-визуализация дерева

<usual post_order_output>/<avl post_order_output> - Usual-дерево/AVL-дерево, изображение-визуализация дерева

<usual find>/<avl find> - Usual-дерево/AVL-дерево

<open limit>/<close limit> - Предел реструктизации

<open read>/<close read> - Хэш-таблица

<open add>/<close add> - Хэш-таблица

<open remove>/<close remove> - Хэш-таблица

<open search>/<close search> - Хэш-таблица

<open output>/<close output> - Хэш-таблица

<stat> Статистика

<exit>- --

Описание задачи, реализуемой программой

Программа реализует ряд действий:

<usual read> - Чтение дерева из файла

<avl copy> - Глубокое копирование из обычного дерева в сбалансированное

<usual add>/<avl add> - Добавление элемента в дерево

<usual remove>/<avl remove> - Удаление элемента из дерева по ключу

<usual search>/<avl search> - Поиск дерева по ключу

<usual pre_order_output>/<avl pre_order_output> - Префиксный обход дерева и создание файла с визуализацией дерева

<usual in_order_output>/<avl in_order_output> - Инфиксный обход дерева и создание файла с визуализацией дерева

<usual post_order_output>/<avl post_order_output> - Постфиксный обход дерева и создание файла с визуализацией дерева

<usual find>/<avl find> - Нахождение элементов, начинающихся на букву и их удаление

<open limit>/<close limit> - Изменения лимита до реструктуризации

<open read>/<close read> - Чтение хэш-таблицы из файла

<open add>/<close add> - Добавление элемента в хэш-таблицу

<open remove>/<close remove> - Удаление элемента из хэш-таблицы по ключу

<open search>/<close search> - Поиск хэш-таблицы по ключу

<open output>/<close output> - Вывод хэш-таблицы

<stat> - Вывод статистики

<exit>- Выход из программы

Способ обращения к программе

Для обращения к программе запускается файл app.exe.

Описание возможных аварийных ситуаций и ошибок пользователя

Программа может не вывести результат, а вывести сообщение об ошибке. Данная ситуация может произойти при условии:

Ошибки пользователя

1. Неверный код действия
2. Выполнение действия при нехватке данных (Удаления, вывод, поиск)

Аварийные ситуации

1. Программа не смогла выделить необходимую память для работы (для добавления элемента в дерево/хэш-таблицу)
2. Программа создала некорректный файл с деревом (если в статистике слишком много элементов)

Описание внутренних СД

Информация о вершине дерева

Информация о вершине представляет собой структуру на языке Си, состоящую из:

```
struct node
{
    char *data;
    color_t color;

    node_t *left;
    node_t *right;
};
```

*data - указатель на строку вершины

color - значение цвета

*left - указатель на левого потомка вершины

*right - указатель на правого потомка вершины

Информация о хэш-таблице с закрытой адресацией

Информация о хэш-таблице с закрытой адресацией представляет собой структуру на языке Си, содержащую:

```
typedef struct
{
    char *data[TABLE_MAX_SIZE];
    size_t (*hash_function)(char *, size_t);

    size_t size;
} close_hash_table_t;
```

data – массив размера TABLE_MAX_SIZE, содержащий слова

```
#define TABLE_MAX_SIZE 40000
```

hash_function – Хэш-функция

size – Максимальный размер хэш-таблицы

Информация о хэш-таблице с открытой адресацией

Информация о хэш-таблице с открытой адресацией представляет собой структуру на языке Си, содержащую:

```c
struct open_hash_table
{
    data_t *data[TABLE_MAX_SIZE];
    size_t (*hash_function)(char *, size_t);

    size_t size;
};
```

data – массив размера TABLE_MAX_SIZE, содержащий слова

```c
#define TABLE_MAX_SIZE 40000
```

hash_function – Хэш-функция

size – Максимальный размер хэш-таблицы

Информация о ключе хэш-таблицы с открытой адресацией

Информация о ключе хэш-таблицы с открытой адресацией представляет собой структуру на языке Си, содержащую:

```c
struct data
{
    char *str;

    data_t *next;
};

typedef struct open_hash_table open_hash_table_t;
```

str – строка, содержащее слово

next – информация о следующем элементе с данным хэшем.

Описание алгоритма

Для начала считывается код действия. Далее в зависимости от выбранного действия:

<usual read>

```c
int node_read_by_file(char *filedata, node_t **root)
```

- С клавиатуры считывается название файла.
- Файл проверяется на корректность
- Программа читает информацию в файле, выделяет память под элемент, заполняет его и добавляет в новое дерево.

<avl copy>

```c
int tree_to_avl_cpy(node_t *node, node_t **avl_tree)
```

- Программа проходится по Usual-дереву
- Выполняется глубокое копирование вершины
- Скопированная вершина добавляется в AVL-дерево

<usual add>/<avl add>

```c
node_t *node_add(node_t *node, node_t *elem)
```

```
node_t *avl_node_add(node_t *node, node_t *elem)
```

- Считывается строка-ключ нового элемента
- Выделяется память под новый элемент дерева
- Проверяется наличие элемента с такой же строкой-ключом
- Новый элемент добавляется в дерево (при отсутствии эквивалентного)
- (ДЛЯ AVL-дерева) Выполняется балансировка поддерева

<usual remove>/<avl remove>

```
void node_delete(node_t **node, char *data)
```

```
void avl_node_delete(node_t **node, char *data)
```

- Считывается строка-ключ элемента
- Проверяется наличие элемента с такой же строкой-ключом
- Программа удаляет элемент (при наличии)
- (ДЛЯ AVL-дерева) Выполняется балансировка поддерева

<usual search>/<avl search>

```
node_t *node_search(node_t *node, char *data, int *compares)
```

- Считывается строка-ключ элемента
- Производится поиск элемента с такой же строкой-ключом
- Выводится результат поиска
- Выводится кол-во сравнений

<usual pre_order_output>/<avl pre_order_output> <usual in_order_output>/<avl in_order_output> <usual post_order_output>/<avl post_order_output>

```
void node_output_pre_order(node_t *node, FILE *f)
```

```
void node_output_in_order(node_t *node, FILE *f)
```

```
void node_output_post_order(node_t *node, FILE *f)
```

- Выполняется соответствующий обход дерева
- Программа преобразует информацию о дереве в код на языке DOT
- Создается изображение дерева

```
void node_export_to_dot_eli(FILE *f, const char *node_data, node_t *node)
```

<usual find>/<avl find>

```
void node_delete_by_char(node_t **node, char c)
```

```
void avl_node_delete_by_char(node_t **node, char c)
{
```

- Считывается символ, с которого должна начинаться строка
- Программа проходится по дереву, закрашивая вершины, строка которых начинается с указанного символа

<open limit>/<close limit>

```
case CODE_OPEN_RESTRUCT_SIZE:
    if (! open_hash_table)
    {
        printf("\nNO HASH TABLE\n");

        break;
    }

    printf("Enter restruct value (> 1) (or enter invalid value to skip changes): ");
    if (scanf("%d", &restruct_limit_tmp) == 1 && restruct_limit_tmp > 1)
        open_restruct_limit = restruct_limit_tmp;

    break;
```

```
case CODE_CLOSE_RESTRUCT_SIZE:
    if (! close_hash_table)
    {
        printf("\nNO HASH TABLE\n");

        break;
    }

    printf("Enter restruct value (> 1) (or enter invalid value to skip changes): ");
    if (scanf("%d", &restruct_limit_tmp) == 1 && restruct_limit_tmp > 1)
        close_restruct_limit = restruct_limit_tmp;

    break;
```

- Считывается предел реструктуризации
- Предел реструктуризации меняется

<open read>/<close read>

```
int open_hash_table_read_by_file(char *filedata, open_hash_table_t *hash_table);
```

```
int close_hash_table_read_by_file(char *filedata, close_hash_table_t *hash_table);
```

- С клавиатуры считывается название файла.
- Файл проверяется на корректность
- Программа читает информацию в файле, выделяет память под элемент, заполняет его и добавляет в новую хэш-таблицу.

<open add>/<close add>

```
int open_hash_table_add(open_hash_table_t *hash_table, char *str, int *comp);
```

```c
int close_hash_table_add(close_hash_table_t *hash_table, char *str, int *comp);
```

- Считывается строка-ключ нового элемента
- Выделяется память под новый элемент хэш-таблицы
- Ищется место для добавления (при наличии коллизии)
- Новый элемент добавляется в хэш-таблицу (при отсутствии эквивалентного)

<open remove>/<close remove>

```c
int open_hash_table_delete(open_hash_table_t *hash_table, char *str);
```

```c
int close_hash_table_delete(close_hash_table_t *hash_table, char *str);
```

- Считывается строка-ключ элемента
- Проверяется наличие элемента с такой же строкой-ключом
- Программа удаляет элемент (при наличии)

<open search>/<close search>

```c
int open_hash_table_search(open_hash_table_t *hash_table, char *str, int *comp);
```

```c
int close_hash_table_search(close_hash_table_t *hash_table, char *str, int *comp);
```

- Считывается строка-ключ элемента
- Производится поиск элемента с такой же строкой-ключом
- Выводится результат поиска
- Выводится кол-во сравнений

<open output>/<close output> - Хэш-таблица,

```
void open_hash_table_output(open_hash_table_t *hash_table);
```

```
void close_hash_table_output(close_hash_table_t *hash_table);
```

- Выполняется обход хэш-таблицы по её размеру (параметру size)
- Непустые хэш-ячейки выводятся на экран

<stat>

```
hashstat();
avl_hash_stat();

total_stat_search();
total_stat_compares();
```

- HASHSTAT – Сравниваются хэш-таблицы с открытой и закрытой адресацией в обычной работе при наличии коллизии и при реструктуризации.
- AVL_HASH_STAT – Сравниваются скорости удаления, объём памяти, кол-во сравнений хэш-таблиц и AVL-деревьев
- TOTAL_STAT_SEARCH – Сравниваются Usual-дерево, AVL-дерево, open хэш-таблица и close хэш-таблица в поставленной задаче — поиске максимально удалённого элемента при самом неудачном развитии событий.
- TOTAL_STAT_COMPARES – Сравниваются все структуры данных средним количеством сравнений при поиске при случайном заполнении структур данных

<exit>

```
case CODE_EXIT:
    tree = node_free(tree);
    avl_tree = node_free(avl_tree);
    open_hash_table_free(&open_hash_table);
    close_hash_table_free(&close_hash_table);

    str_free(&filename, &filename_size);
    str_free(&data, &data_size);

    flag = false;

    break;
```

- Освобождается память, выделенная под динамически выделенные данные программы (Деревья и строки, необходимые для выполнения действий с вводом строк, хэш-таблицы)
- Программа завершает свою работу

Набор тестов

В ходе выполнения лабораторной работы были написаны тесты для проверки работы программы

Позитивные тесты

Номер теста	Входные данные	Ожидаемые выходные данные
01 - <usual read> Чтение корректного файла	1 test2.txt 6	IN-ORDER: 0 1 1 214

		124126 1243124 15 26 351 421 48 5 531 573 6 8p076 9 975 maepet mamaprivet
02 - <usual add> Добавление вершины в дерево	2 a 2 b 6	IN-ORDER: a b
03 - <usual remove> Исключение вершины из	2	IN-ORDER:

дерева	a 2 b 3 a 6	b
04 - <usual search> Поиск вершины по указанному ключу	1 test1.txt 4 573	DATA WAS FOUNDED SUCCESSFULLY Total compares: *Кол-во сравнений*
05 - <usual pre-order_output> Вывод префиксного обхода дерева	1 test1.txt 5	PRE-ORDER: mamaprivet 5 1 0 1243124 1 214 124126 421 351 26 15 48

		6 573 531 9 8p076 975 maepet
06 - <usual post_order_output> Выход из программы	1 test1.txt 7	POST-ORDER: 0 124126 1 214 15 26 351 48 421 1243124 1 531 573 8p076 maepet

		975 9 6 5 mamaprivet
07 - <usual find> Нахождение элементов, начинающихся на данный символ и их удаление	1 test1.txt 8 1 6	*Файл с деревом, где удалены найденные элементы*
08 - <avl copy> - Копирование обычного дерева в AVL-дерево	1 test1.txt 9 13	«AVL-дерево с элементами Usual-дерево»
09 - < avl add> Добавление вершины в дерево	1 test1.txt 9 10 opewr 13	«Файл с деревом, имеющим добавленный элемент»
10 - < avl remove> Исключение вершины из дерева	1 test1.txt	«Файл с деревом, не имеющим элемент «1» »

	9 11 1 13	
11 - <avl search> Поиск вершины по указанному ключу	1 test1.txt 9 12 1	DATA WAS FOUNDED SUCCESSFULLY Total compares: *Кол-во сравнений*
12 - < avl in-order_output> Вывод префиксного обхода дерева	1 test1.txt 9 14	PRE-ORDER: mamaprivet 5 1 0 1243124 1 214 124126 421 351 26 15 48 6

		573 531 9 8p076 975 maepet
13 - < avl post_order_output> Выход из программы	1 test1.txt 9 15	POST-ORDER: 0 124126 1 214 15 26 351 48 421 1243124 1 531 573 8p076 maepet 975

		9 6 5 mamaprivet
14 - <avl find> Нахождение элементов, начинающихся на данный символ и их удаление	1 test1.txt 9 16 1 13	*Файл с деревом, где удалены найденные элементы*
15 - <close read> Чтение файла и построение хэш-таблицы	24 test1.txt	«Хэш-таблица с данными из файла»
16 - <close add> Добавление в хэш-таблицу	24 test1.txt 25 powpm 28	«Хэш-таблица с добавленным элементом»
17 - <close delete> Удаление из хэш-таблицы	24 test1.txt 26 1 28	«Хэш-таблица с удалённым элементом»
18 - <close search> Поиск в	24	DATA WAS FOUNDED

хэш-таблице	test1.txt 27 1	SUCCESSFULLY Total compares: *Кол-во сравнений*
19 <close limit> изменение предела реструктуризации	24 test1.txt 23 55	restruct limit: 55
20 - <open read> Чтение файла и построение хэш-таблицы	18 test1.txt	«Хэш-таблица с данными из файла»
21 - <open add> Добавление в хэш-таблицу	18 test1.txt 19 powpm 22	«Хэш-таблица с добавленным элементом»
22 - < open delete> Удаление из хэш-таблицы	18 test1.txt 20 1 22	«Хэш-таблица с удалённым элементом»
23 - < open search> Поиск в хэш-таблице	18 test1.txt	DATA WAS FOUNDED SUCCESSFULLY Total compares: *Кол-во

	21 1	сравнений*
24 <open limit> изменение предела реструктуризации	18 test1.txt 17 55	restruct limit: 55
25 - <stat> Вывод статистики	29	*Статистика*
26 - <exit> Выход из программы	30	-

Негативные тесты

Номер теста	Входные данные	Выходные данные
01 - Код неправильный	15	INVALID CODE
02 - Код с иными символами	1.1	INVALID CODE
03 - <usual remove> Нет дерева	3	NO DATA
04 - <usual add> Найден эквивалентный элемент	1 test1.txt 2	THIS VALUE IS ALREADY IN TREE

	1	
05 - <usual search> Нет дерева	4	NO DATA
06 - <usual remove> Нет элемента с данным ключом	1 test1.txt 3 yotumidore	ELEMENT IS NOT FOUND
07 - <usual pre_order_output> Нет дерева	5	NO DATA
08 - <usual in_order_output> Нет дерева	6	NO DATA
09 - <usual post_order_output> Нет дерева	7	NO DATA
10 - <avl remove> Нет дерева	11	NO DATA
11 - <avl add> Найден эквивалентный элемент	1 test1.txt 9 10 1	THIS VALUE IS ALREADY IN TREE
12 - <avl search> Нет дерева	12	NO DATA
13 - <avl remove> Нет элемента с данным ключом	1 test1.txt 9 11 yotumidore	ELEMENT IS NOT FOUND

14 - <avl pre_order_output> Нет дерева	13	NO DATA
15 - <avl in_order_output> Нет дерева	14	NO DATA
16 - <avl post_order_output> Нет дерева	15	NO DATA
17 - <open limit> Нет хэш-таблицы	17	NO DATA
18 - <open remove> Нет хэш-таблицы	20	NO DATA
19 - <open search> Нет хэш-таблицы	21	NO DATA
20 - <open output> Нет хэш-таблицы	22	NO DATA
21 - <close limit> Нет хэш-таблицы	23	NO DATA
22 - <close remove> Нет хэш-таблицы	26	NO DATA
23 - <close search> Нет хэш-таблицы	27	NO DATA
24 - <close output> Нет хэш-таблицы	28	NO DATA

Сравнение времени удаления, объема памяти и количества сравнений при использовании сбалансированных деревьев и хеш-таблиц.

Ниже представлена статистика по зависимости времени удаления, объема памяти и количества сравнений от количества элементов при использовании сбалансированных деревьев и хеш-таблиц.(для получения значения использовалось 1500 итераций (Значение ITER_COUNT = 1500))

Статистика

HASHES VS AVL STATISTICS (time in nsec) (total iteration: 1500)
(P.S. count is (collision count + 1) for hash tables)

BINARY HASH FUNCTION; SIZE = 3000

count	T DELETE (OP)	T DELETE (CL)	T DELETE (AVL)	COMPARES (OP)	COMPARES (CL)	COMPARES (AVL)	SIZE (OP)	SIZE (CL)	SIZE (AVL)
1	68.890000	64.705000	43.249000	1	1	1	320051	320035	46
2	69.330000	67.355000	114.482000	2	2	2	320068	320041	84
4	77.060000	69.650000	212.114000	4	4	3	320112	320053	160
8	87.186000	93.625000	311.061000	8	8	3	320200	320077	312
16	122.862000	126.578000	435.654000	16	16	4	320376	320125	616
32	195.027000	201.934000	592.680000	32	32	5	320728	320221	1224
64	343.648000	360.943000	774.287000	64	64	6	321432	320413	2440
128	645.196000	653.238000	980.945000	128	128	7	322840	320797	4872
256	1271.141000	1291.824000	1221.967000	256	256	8	325656	321565	9736
512	2512.828000	2561.637000	1482.546000	512	512	9	331288	323101	19464
1024	4899.247000	5107.047000	1781.625000	1024	1024	10	342552	326173	38920

На основе полученных данных можно сделать следующие выводы:

- Время поиска в хэш-таблице при большом количестве коллизий возрастает сильнее, чем время поиска в AVL-дереве при росте количества элементов (Это связано с тем, что сложность поиска среди коллизий — O(N), а у AVL-дерева — O(logN))
- Память, затраченная на хэш-таблицы, во много раз превосходит память AVL-дерева, причём память на таблицу с закрытой адресацией меньше, чем на таблицу с открытой адресацией
- При одинаковых количествах сравнений поиск в таблице быстрее, чем в AVL-дереве (Это связано с тем, что поиск в AVL-дереве осуществляется рекурсивно, что довольно затратно)

Сравнение эффективности поиска в хеш-таблице с разными адресациями при различном количестве коллизий и при различных методах их разрешения.

Ниже представлена статистика по эффективности поиска в хеш-таблице с разными адресациями при различном количестве коллизий и при различных методах их разрешения.(для получения значения использовалось 1500 итераций (Значение ITER_COUNT = 1500))

Статистика

```
HASHES STATISTICS (time in nsec) (total iteration: 1500)

BINARY HASH FUNCTION; SIZE = 3000
collision|HASH FIND (OPEN)|HASH FIND(CLOSE)|COMPARES  (OPEN)|COMPARES (CLOSE)|   best time    |
0        |65.906000       |68.636000       |1              |1               |     OPEN       |
1        |71.662000       |52.669000       |2              |2               |     CLOSE      |
3        |58.467000       |59.325000       |4              |4               |     OPEN       |
7        |72.714000       |75.128000       |8              |8               |     OPEN       |
15       |94.683000       |111.521000      |16             |16              |     OPEN       |
31       |158.575000      |184.544000      |32             |32              |     OPEN       |
63       |286.158000      |345.673000      |64             |64              |     OPEN       |
127      |557.448000      |640.654000      |128            |128             |     OPEN       |
255      |1064.739000     |1246.570000     |256            |256             |     OPEN       |
511      |2134.573000     |2505.454000     |512            |512             |     OPEN       |
1023     |4207.096000     |5135.235000     |1024           |1024            |     OPEN       |

TERNARY HASH FUNCTION; SIZE = 6000
collision|HASH FIND (OPEN)|HASH FIND(CLOSE)|COMPARES  (OPEN)|COMPARES (CLOSE)|   best time    |
0        |55.035000       |52.803000       |1              |1               |     CLOSE      |
1        |53.815000       |51.567000       |1              |1               |     CLOSE      |
3        |54.139000       |50.785000       |1              |1               |     CLOSE      |
7        |53.897000       |50.626000       |1              |1               |     CLOSE      |
15       |54.018000       |51.593000       |1              |1               |     CLOSE      |
31       |58.533000       |52.090000       |1              |1               |     CLOSE      |
63       |54.447000       |51.470000       |1              |1               |     CLOSE      |
127      |53.906000       |51.280000       |1              |1               |     CLOSE      |
255      |53.023000       |51.731000       |1              |1               |     CLOSE      |
511      |53.152000       |56.224000       |1              |1               |     OPEN       |
1023     |127.356000      |51.826000       |1              |1               |     CLOSE      |

BINARY HASH FUNCTION; SIZE = 6000
collision|HASH FIND (OPEN)|HASH FIND(CLOSE)|COMPARES  (OPEN)|COMPARES (CLOSE)|   best time    |
0        |58.578000       |53.740000       |1              |1               |     CLOSE      |
1        |55.431000       |53.859000       |2              |2               |     CLOSE      |
3        |59.895000       |62.853000       |4              |4               |     OPEN       |
7        |71.043000       |82.818000       |8              |8               |     OPEN       |
15       |97.619000       |109.055000      |16             |16              |     OPEN       |
31       |162.321000      |187.528000      |32             |32              |     OPEN       |
63       |292.614000      |361.101000      |64             |64              |     OPEN       |
127      |568.179000      |676.682000      |128            |128             |     OPEN       |
255      |1077.450000     |1257.261000     |256            |256             |     OPEN       |
511      |2321.888000     |2441.344000     |512            |512             |     OPEN       |
1023     |4202.408000     |4908.747000     |1024           |1024            |     OPEN       |
```

- Время поиска (Таблица 1) в открытой хэш-таблице меньше, чем в закрытой хэш-таблице (Это связано с тем, что работа с указателями быстрее работы с индексами)
- При реструктуризации хэш-таблиц (Таблица 2) можно также увидеть, что теперь скорость доступа к элементу без коллизии в хэш-таблице с закрытой адресации выше, чем в хэш-таблице с открытой адресацией (Это связано с тем, что для работы с хэш-таблицей с открытой адресацией используются дополнительные ресурсы для работы со структурой)
- При реструктуризации хэш-таблиц (Увеличение размера и смены хэш-функции) (Таблица 2) можно увидеть константное количество сравнений — 1 (Коллизии устранились).
- При изменении лишь размера при реструктуризации (таблица 3) можно заметить, что увеличение размера коллизию не исправил (Это произошло из-за того, что хэш элементов ИЗНАЧАЛЬНО < 3000 (прежнее кол-во элементов), поэтому при увеличении ситуация осталась прежней)

Сравнение эффективности использования структур (по времени и по памяти) для поставленной задачи (поиск максимального по глубине элемента).

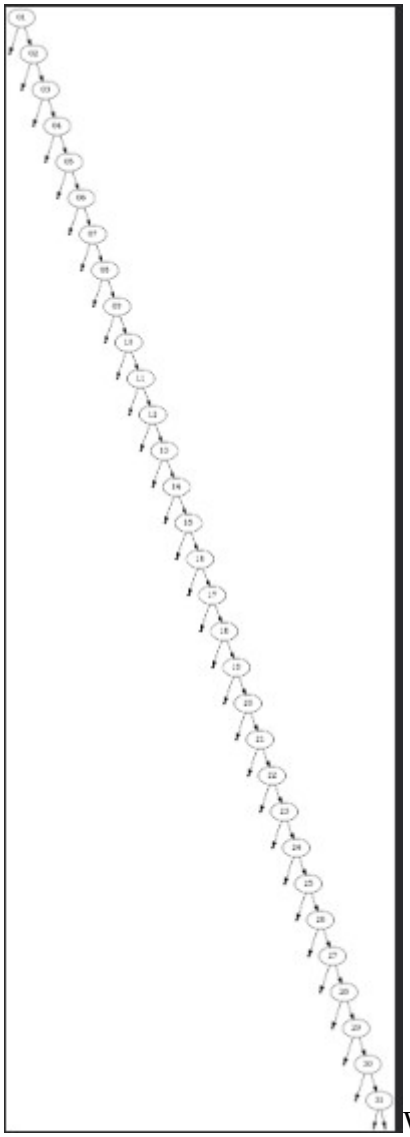
Ниже представлена статистика по эффективности использования структур (по времени и по памяти) для поставленной задачи (поиск максимального по глубине элемента). (для получения значения использовалось 1500 итераций (Значение ITER_COUNT = 1500))

```
TOTAL STATISTICS: SEARCHING MAX ELEMENT (time in nsec) (total iteration: 1500)
(P.S. count is (collision count + 1) for hash tables)

BINARY HASH FUNCTION; SIZE = 3000
 count |  WORTH TREE  |    AVL TREE  |   OPEN HASH  |  CLOSE HASH  | COMP (WORST) | COMP   (AVL) | COMP  (OPEN) | COMP (CLOSE) |
1      |30.948000     |30.345000     |54.561000     |55.689000     |1             |1             |1             |1             |
2      |44.508000     |30.709000     |56.499000     |53.710000     |2             |1             |2             |2             |
4      |47.770000     |34.566000     |59.023000     |61.902000     |4             |2             |4             |4             |
8      |82.088000     |40.497000     |71.114000     |79.000000     |8             |3             |8             |8             |
16     |153.280000    |48.103000     |100.186000    |110.771000    |16            |4             |16            |16            |
32     |331.720000    |56.436000     |163.387000    |186.923000    |32            |5             |32            |32            |
64     |625.259000    |61.807000     |298.971000    |348.991000    |64            |6             |64            |64            |
128    |1282.192000   |69.455000     |558.435000    |676.808000    |128           |7             |128           |128           |
256    |2819.954000   |86.681000     |1096.903000   |1331.170000   |256           |8             |256           |256           |
512    |5399.910000   |90.455000     |2201.842000   |2712.286000   |512           |9             |512           |512           |
1024   |11523.426000  |100.323000    |4397.809000   |5638.495000   |1024          |10            |1024          |1024          |

TOTAL STATISTICS: SIZES (size in bytes)
 count |  WORTH SIZE  |    AVL SIZE  |   OPEN SIZE  |  CLOSE SIZE  |
1      |42            |46            |320051        |320035        |
2      |76            |84            |320068        |320041        |
4      |144           |160           |320112        |320053        |
8      |280           |312           |320200        |320077        |
16     |568           |616           |320376        |320125        |
32     |1128          |1224          |320728        |320221        |
64     |2248          |2440          |321432        |320413        |
128    |4616          |4872          |322840        |320797        |
256    |9224          |9736          |325656        |321565        |
512    |18440         |19464         |331288        |323101        |
1024   |37896         |38920         |342552        |326173        |
```

- При большом количестве коллизий и максимальной высоте быстрее всех ищет AVL-дерево
- Деревья являются выгоднями по памяти, причём WORTH-дерево по памяти чуть меньше (из-за отсутствия необходимости в хранении высоты)
- Хэш-таблица с закрытыми адресами выгоднее по памяти, чем хэш-таблица с открытыми адресами (это связано с дополнительными расходами для реализации связного списка при коллизияв у хэш-таблицы с открытой адресацией)
- Несмотря на аналогичный алгоритм поиска (по сути в связном списке) WORTH-дерево почти в 2 раза медленнее, чем хэш-таблица с открытой адресацией (Это связано с рекурсивной реализацией поиска в WORTH-дереве)

 WORST дерево при высоте 31 и количеством элементов 31. Степень ветвления 1

Сравнение среднего количества сравнений для поиска данных в указанных структурах.

Ниже представлена статистика по среднему количеству сравнений для поиска данных в указанных структурах. (!ИНФОРМАЦИЯ СЛУЧАЙНАЯ!) (для получения значения использовалось 1500 итераций (Значение ITER_COUNT = 1500))

```
TOTAL STATISTICS: AVERAGE COMPARES FROM RANDOM DATA (time in nsec) (total iteration: 1500)
(P.S. count is (collision count + 1) for hash tables)

BINARY HASH FUNCTION; SIZE = 3000
 count |  USUAL COMP  |    AVL COMP  |   OPEN COMP  |  CLOSE COMP  |
1      |1.000000      |1.000000      |1.000000      |1.000000      |
2      |1.500000      |1.500000      |1.000000      |1.000000      |
4      |2.000000      |2.000000      |1.000000      |1.000000      |
8      |4.000000      |2.625000      |1.000000      |1.000000      |
16     |3.812500      |3.437500      |1.000000      |1.000000      |
32     |6.125000      |4.375000      |1.000000      |1.000000      |
64     |6.546875      |5.265625      |1.031250      |1.000000      |
128    |7.070312      |6.164062      |1.039062      |1.023438      |
256    |8.875000      |7.257812      |1.031250      |1.035156      |
512    |10.880859     |8.246094      |1.085938      |1.119141      |
1024   |11.904297     |9.251953      |1.185547      |1.283203      |
2048   |13.062500     |10.223145     |1.339844      |2.142578      |
4096   |14.114014     |11.260010     |1.696533      |28.736816     |
```

- Среднее количество сравнений в хэш-таблицах, в отличие от деревьев, лежит в окрестности 1.
- (В ДАННОМ СЛУЧАЕ МАКСИМАЛЬНЫЙ РАЗМЕР ХЭШ-ТАБЛИЦ С ЗАКРЫТОЙ АДРЕСАЦИЕЙ = 3000) При 4096 элементах можем увидеть, что в хэш-таблице с закрытой адресацией поиск происходит в среднем при довольно большом количестве сравнений. Это произошло потому, что хэш-таблица переполнена, и среднее количество сравнений непредсказуемо (В данном случае реструктуризация хэш-таблиц не проводилась в целях фиксации данного явления).
- Среднее количество сравнений в AVL-дереве постепенно становится меньше, чем среднее количество сравнений в обычном дереве

Выводы по проделанной работе

Хэш-таблицы - структура данных, позволяющая быстро выполнять вставку/удаление/поиск элементов. Единственный недостаток данного способа представления данных — коллизия. При возникновении коллизии работа хэш-таблицы замедляется. Проблему замедления решает реструктуризация хэш-таблицы (Например, изменение хэш-функции или изменение размера таблицы), однако при изменении параметров не всегда решается проблема коллизии.

AVL-дерево — двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её поддеревьев не должна различаться более чем на 1 (модуль разницы <= 1). Данная структура данных является модификацией двоичного дерева поиска и позволяет компактно расположить данные для работы с ними. Несмотря на наличие существенной разницы во времени выполнения операций по сравнению с хэш-таблицей, AVL-дерево позволяет выполнять базовые операции со стабильно маленькой скоростью, а также расходует намного меньше памяти, чем хэш-таблицы.

Контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

Идеально сбалансированное дерево не является двоичным деревом поиска (при построении такого дерева значение узла НЕ учитывается).

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

АВЛ-дерево хранит данные так, что поиск выполняется эффективно. O(logN) – максимальная сложность поиска в такой таблице.

В дереве двоичного поиска данные хранятся в зависимости от построение, поэтому данные могут искаться с максимальной сложностью O(N).

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

Хэш-таблица — структура данных, определяющая место хранения элемента по хэш-функции — функция, при помощи которой вычисляется индекс элемента. При построении:

1. Вычисляется хэш-функция элементам
2. По индексу располагается элемент хэш-таблицы

4. Что такое коллизии? Каковы методы их устранения.

Коллизия — явление, при котором нескольким ключам соответствует одна и та же хэш-функция. Для устранения коллизии производится реструктуризация таблицы (Например, меняется хэш-функция или меняется максимальный размер таблицы).

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

Поиск в хэш-таблице становится неэффективным, когда в таблице находится большое число коллизий. В таком случае после вычисления хэш-функции элемента необходимо будет перебирать и элементы, имеющие равный хэш, но имеющие совершенно другой ключ

6. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хештаблицах и в файле

Эффективность поиска в хэш-таблице одна из самых высоких среди предложенных структур данных, но при большом числе коллизий эффективность падает, и поиск может быть не таким эффективным. В лучшем случае сложность поиска — O(1).

Эффективность поиска в АВЛ-дереве ниже, чем у хэш-таблицы, однако является стабильной. Максимальная сложность поиска в АВЛ-дереве — O(logN).

Эффективность поиска в двоичном дереве поиска ниже, чем у АВЛ-дерева. Максимальная сложность поиска в двоичном дереве поиска может достигать O(N).

Эффективность поиска в файле одна из самых низких среди предложенных структур данных, так как помимо максимальной сложности O(N) требует дополнительных ресурсов для работы с файлом.