|  |  |
| --- | --- |
|  | **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**

Студент **Паламарчук А.Н.**

Группа **ИУ7-33Б**

Предмет **Типы и структуры данных**

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

|  |  |
| --- | --- |
| Студент | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Паламарчук А.Н.** |
| Преподаватель | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Никульшина Т. А.** |
| Преподаватель | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Барышникова М. Ю.** |

*2023 г.*

# Условие задачи

Построить хеш-таблицу по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице. Вывести на экран деревья и хеш-таблицу. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Оценить эффективность использования этих структур (по времени и по памяти) для поставленной задачи.

# Техническое задание

Используя предыдущую программу (задача №6), 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.

## Входные данные

* Команда (число) из меню программы (см. меню программы) – обозначение необходимой операции.

## Выходные данные

* Таблицы эффективности
* Файл в DOT формате
* Файлы png с деревьями
* Хеш-таблицы
* Результат поиска в двоичном дереве поиска
* Результат поиска в открытой хеш-таблице
* Результаты поиска в закрытой хеш-таблице

## Возможные аварийные ситуации

* Некорректный исходный файл
* Ошибка открытия файла
* Отсутствие исходного файла
* Некорректная команда

## Описание внутренних структур данных

typedef struct tree\_node

{

const char \*name;

int val;

// родитель

struct tree\_node \*parent;

// меньшие

struct tree\_node \*left;

// большие

struct tree\_node \*right;

} tree\_node\_t;

typedef struct hash\_node

{

int key;

int value;

struct hash\_node \*node;

} hash\_node\_t;

typedef struct hash\_table

{

size\_t size;

hash\_node\_t \*\*array;

} hash\_table\_t;

**Константы**

#define \_POSIX\_C\_SOURCE 199309L

#define COUNT\_TESTS 100

#define FILE\_DOT "graph.gv"

#define FILE\_PNG "graph.png"

#define FILE\_BST\_DOT "graph\_BST.gv"

#define FILE\_BST\_PNG "graph\_BST.png"

**Используемые функции**

int fill\_tree(struct val\_nodes \*tree);

Заполнение дерева значениями пользователя

void export\_to\_dot(FILE \*f, const char \*tree\_name, struct tree\_node \*tree);

Перевод дерева в DOT формат

void apply\_pre(tree\_node\_t \*tree, void (\*f)(tree\_node\_t\*, void\*), void \*arg);

Префиксный обход дерева

void apply\_in(tree\_node\_t \*tree, void (\*f)(tree\_node\_t\*, void\*), void \*arg);

Инфиксный обход дерева

void apply\_post(tree\_node\_t \*tree, void (\*f)(tree\_node\_t\*, void\*), void \*arg);

Постфиксный обход дерева

tree\_node\_t\* balance\_BST(int \*arr, size\_t asize);

Построение сбалансированного дерева

int create\_node(tree\_node\_t \*\*new\_node, int data);

Создает элемент хеш-таблицы

void fill\_hash\_table(hash\_table\_t \*table, int \*a, size\_t asize);

Заполнение закрытой хеш-таблицы

void fill\_hash\_table\_open(hash\_table\_t \*table, int \*a, size\_t asize);

Заполнение открытой хеш-таблицы

tree\_node\_t\* search\_from\_tree(tree\_node\_t \*node, int data, size\_t \*cmp);

Поиск в двоичном дереве поиска

hash\_node\_t\* search\_from\_hash\_table(hash\_table\_t\* table, int key, size\_t \*cmp);

Поиск в закрытой хеш-таблице

hash\_node\_t\* open\_htable\_search(hash\_table\_t\* table, int key, size\_t \*cmp);

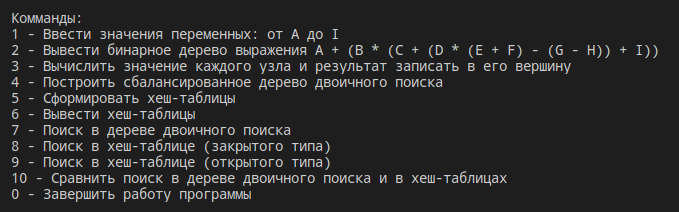
Поиск в открытой хеш-таблице

int cmp\_tree\_vs\_table(void);

Сравнение двоичного дерева поиска и хеш-таблиц

**Меню программы**

Программа обрабатывает нужную команду:

****

**Хеш-функция**

// key – значение для хеширования

// size – размер хеш-таблицы

int hash(int key, int size)

{

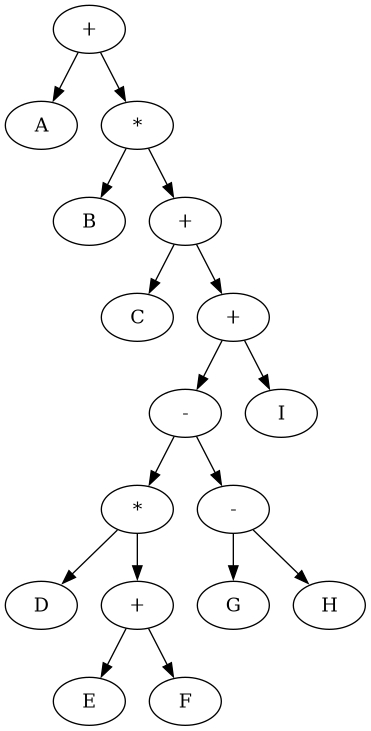
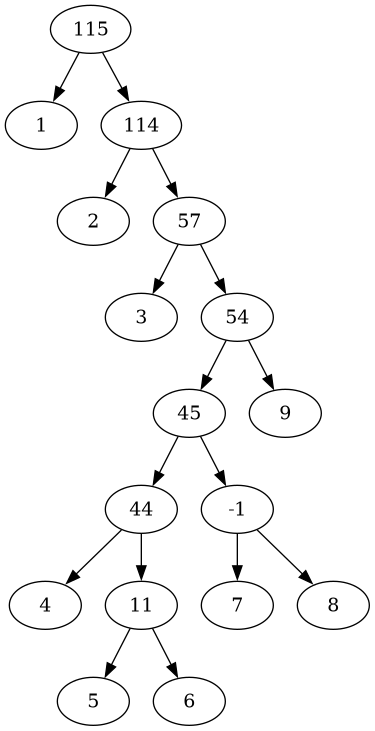
return key % size;

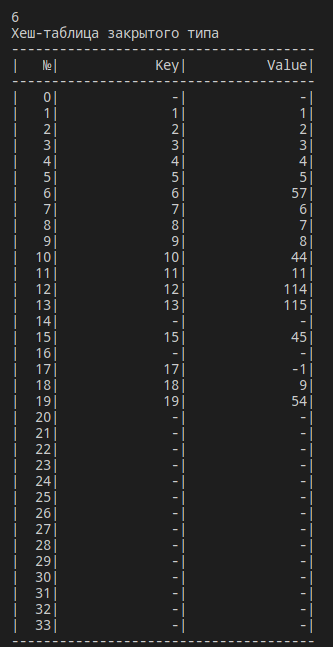
}

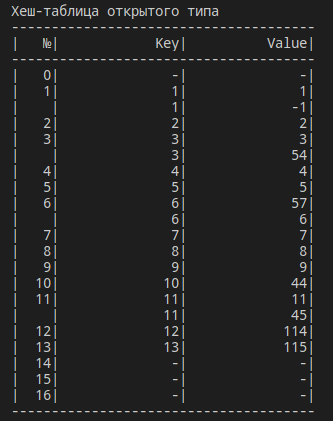
**Реструктуризация**

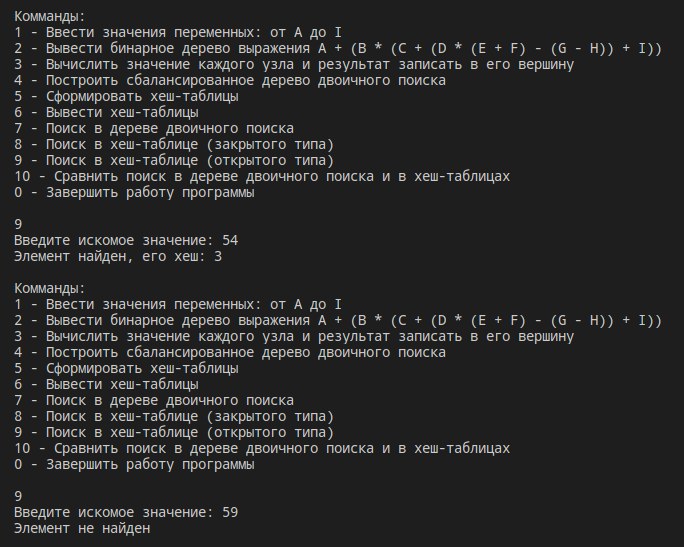
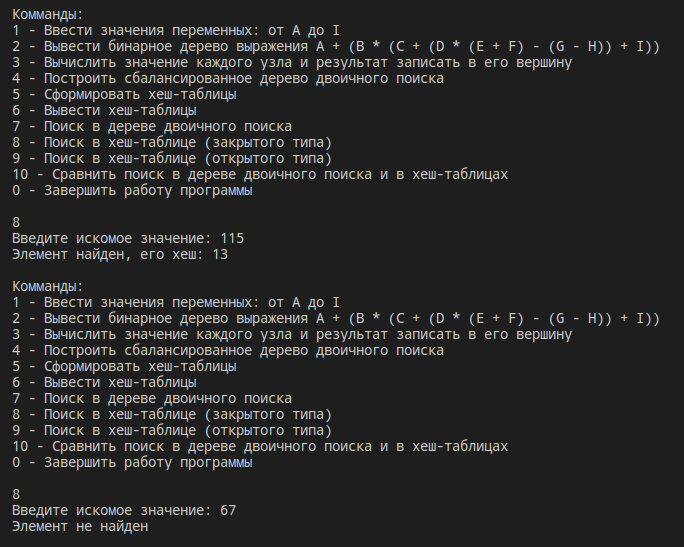
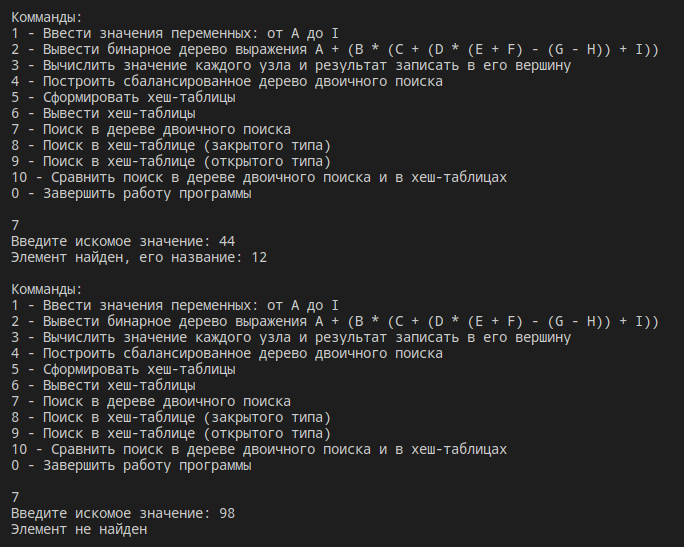
При коллизии (более 10 элементов с одним хешем) в хеш-таблице, выполняется реструктуризация. Увеличивается размер хеш-таблицы в 2 раза и перезаписывается в новую хеш-таблицу.

**Пример работы программы**

****

****

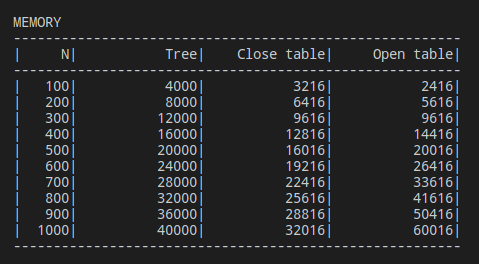
****

****

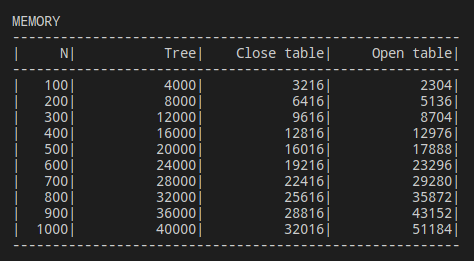
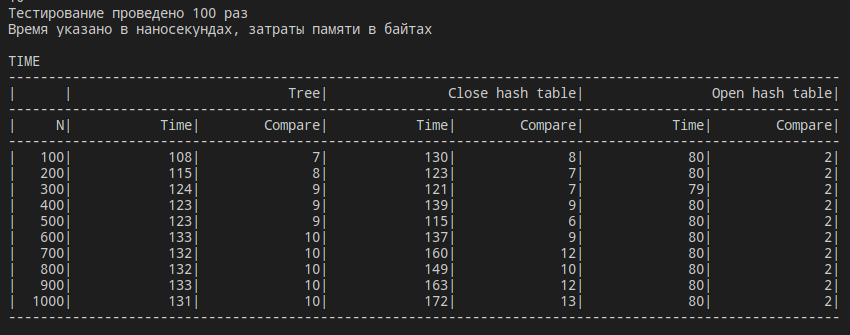
**Производительность**

* Замер для структур, построенных с помощью массива, упорядоченного по возрастанию.

# C:\Users\Natalie\Downloads\image (27).png



* Замер для структур, построенных с помощью неупорядоченного массива.



Сравним скорость поиска элемента. Скорость поиска по хеш-таблицам выше скорости поиска по бинарному дереву поиска и хеш-таблицы занимают меньше места (каждый узел дерева больше узла списка). Однако при возникновении достаточно большого числа коллизий закрытая хеш-таблица начинает уступать бинарному дереву поиска.

# Выводы по проделанной работе

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

Для ещё более быстрого поиска данных можно использовать хеш-таблицы, которые позволяют не только ускорить поиск, но и уменьшить количество занимаемой памяти. Однако в хеш-таблицах могут возникать коллизии, для избавления от которых можно использовать списки или алгоритмы поиска новых мест в массиве.

Поэтому для хранения данных, в которых нужно будет выполнять быстрый поиск, выгоднее всего использовать хеш-таблицы, 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 всегда является деревом поиска)

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

Алгоритм поиска ничем не отличается. Просто в среднем сравнений придётся произвести меньше.

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

Массив, позиция элементов в котором определяется хеш-функцией, которая позволяет определить позицию каждого элемента.

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

Коллизия - это ситуация, когда хеш-функция присваивает двум разным элементам одинаковый индекс. В зависимости от того, закрытая или открытая хеш-таблица, коллизии устраняются

для первого случая - поиском новой позиции чуть дальше (index + k^2), где k - номер попытки поиска нового места.

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

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

В случаях, когда количество коллизий становится слишком большим. В таких случаях хеш-таблицу реструктуризируют.

**6. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хеш-таблицах и в файле.**

* в хеш-таблицах минимальное - O(1)
* в АВЛ - O(log n)
* в двоичном дереве поиска от O(log n) до O(n)