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

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ 6

По дисциплине «Типы и структуре данных»

### Название Деревья и Хеш-таблицы

### Студент Мансуров Владислав Михайлович

*фамилия, имя, отчество*

### Группа ИУ7-36Б

Тип лабораторной работы Учебная

### Название

предприятия

|  |  |  |
| --- | --- | --- |
| Студент |  | Мансуров В.М. |
| Преподаватель | *подпись, дата* | *фамилия, и.о.*  Никульшина Т.А. |
|  | *подпись, дата* | *фамилия, и.о.* |

*2021 г.*

***Содержание***

### Условие задачи 4

### Техническое задание… 4

### Описание алгоритма 5

### Анализ алгоритмов… 13

### Контрольные вопросы… 17

### Заключение… 19

***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.*

***Условия задачи***

***Вариант 2***

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 добавление введенного целого числа, если его там нет, в ДДП, в сбалансированное дерево, в хеш-таблицу и в файл. Сравнить время добавления, 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

***Техническое задание***

*Построить ДДП, сбалансированное двоичное дерево (АВЛ) и хештаблицу по указанным данным. Сравнить эффективность поиска в ДДП в АВЛ дереве и в хеш-таблице (используя открытую или закрытую адресацию) и в файле. Вывести на экран деревья и хеш-таблицу. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий.*

***Входные данные***

В качестве указания пункта меню – целое число.

В подпунктах – по 1-2 целых числа

***Выходные данные***

При выводе Хеш-таблицы – вывод данных из Хеш-таблицы, содержащие целые числа.

При выводе Дерева – вывод дерева, содержащий целые числа.

Данные об эффективности.

***Описание алгоритма:***

В реализации программы были созданы две структуры для хранения Дерева и Хеш-таблицы:

*Дерево*

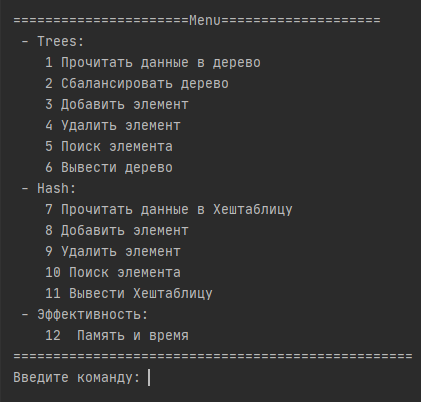
typedef struct st\_tree\_branch  
{  
 int number; //целое число  
 int height; // высота узла  
 struct st\_tree\_branch \*left; // левый узел вершины  
 struct st\_tree\_branch \*right; // правый узел вершины  
} branch\_t;

typedef struct st\_tree  
{  
 struct st\_tree\_branch \*head; // указатель на первую вершину дерева  
 int size; // количество элементов  
 int total\_compare; // общее количество сравнений  
} tree\_t;

*Хеш-таблица*

typedef struct st\_hash\_elem  
{  
 int number; // целое число  
 char \_is\_full; // заполненность элемента  
} hash\_elem\_t;  
  
typedef struct st\_hash\_table  
{  
 hash\_elem\_t \*array; // массив данных таблицы  
 int size; // размер массива  
 int divider; // делитель, по которому находится индекс числа  
 int elements; // количество элементов   
 int total\_compare; // общее количество сравнений  
} hash\_table\_t;

Для взаимодействия с программой было создано консольное меню:



Название файл передается через командную строку, если оно корректно или такой файл существует и не пуст, то пользователю выводится меню для взаимодействие с программой.

При вводе корректного значения выполняются определенные действия или операции, при некорректном выводится в консоль сообщение об ошибке и предоставляется ввести пункт еще раз. Так происходит пока не будет введен корректное значение меню.

Для реализации программы, была использована статическая и динамическая типизация, поэтому были написаны функции выделения памяти и очищения памяти, чтобы при работе не возникали утечки памяти.

Если дерево пусто и запуске – 4, 5, 6 пунктах – Выводится сообщение об этом.

Если хеш-таблица пуста и запуске – 9, 10, 11 пунктах – Выводится сообщение об этом.

Алгоритм хеш-функции: Для нахождения ключа используются модуль от числа – num % size,

где num – число

size - размер

*Пункта 1 консольного меню:*

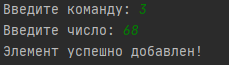
В этом пункте данные читаются из файл в бинарное дерево. При успешном чтении дерева, получаем сообщение об этом, при повторном чтении файл не считывается.

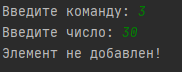
*Пункта 2 консольного меню:*

В этом пункте происходит балансировка дерева.

*Пункта 3 консольного меню:*

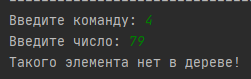
Добавления целого числа в дерево.

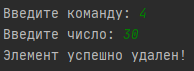




*Пункта 4 консольного меню:*

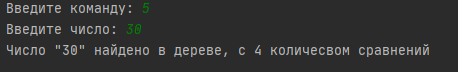
Удаление целого числа из дерева

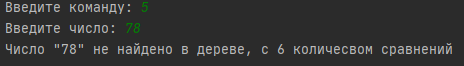




*Пункта 5 консольного меню:*

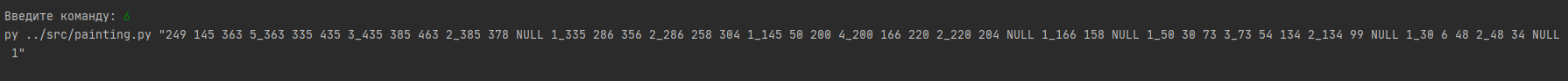
Поиск целого числа в дереве

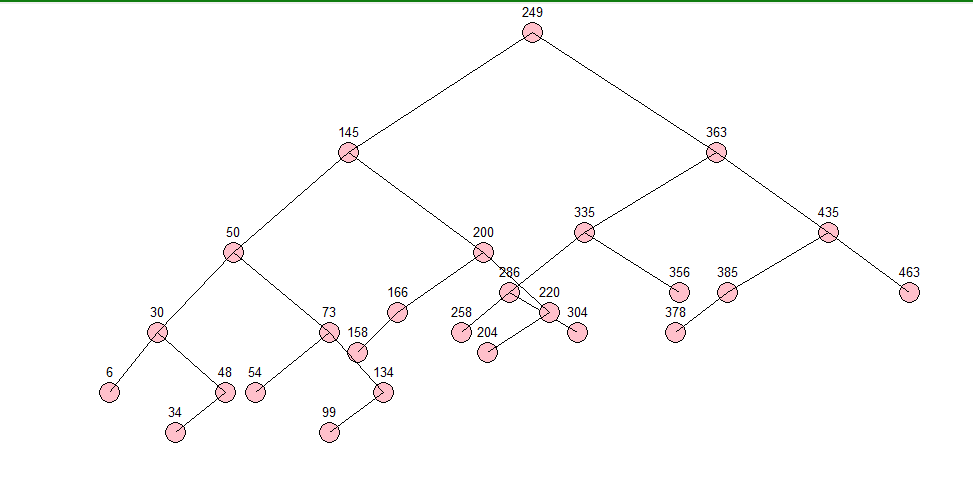




*Пункта 6 консольного меню:*

Вывод дерева, запуская питон скрип, который через свои графические библиотеки стоит дерево. При этом выводится команда запуска этого скрипта.





*Пункта 7 консольного меню:*

Чтение из файла целых чисел в хеш-таблицу, при успешном чтении дерева, получаем, сообщении об этом, при повторном чтении не считывает.



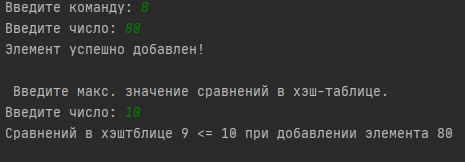
*Пункта 8 консольного меню:*

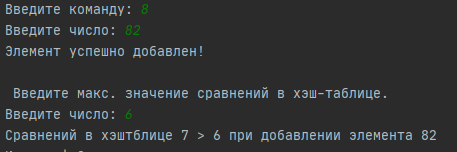
Добавление элемента в хеш-таблицу. Добавление происходит следующим образом:

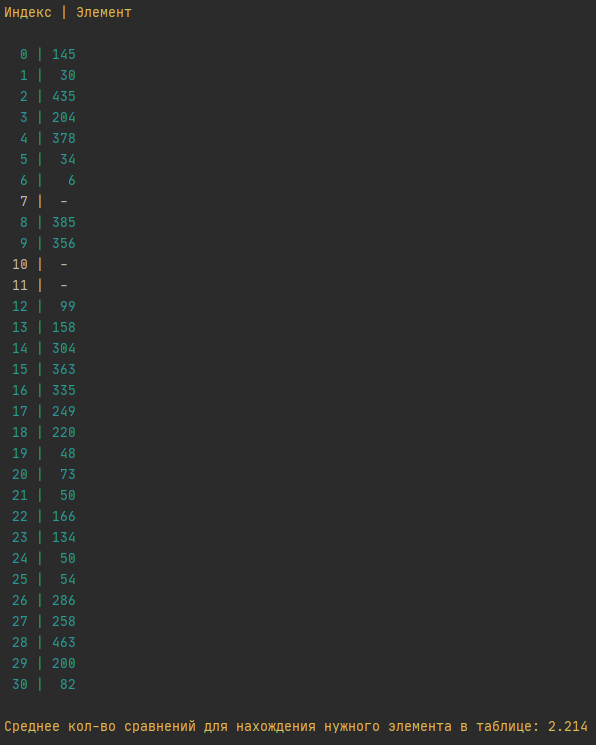
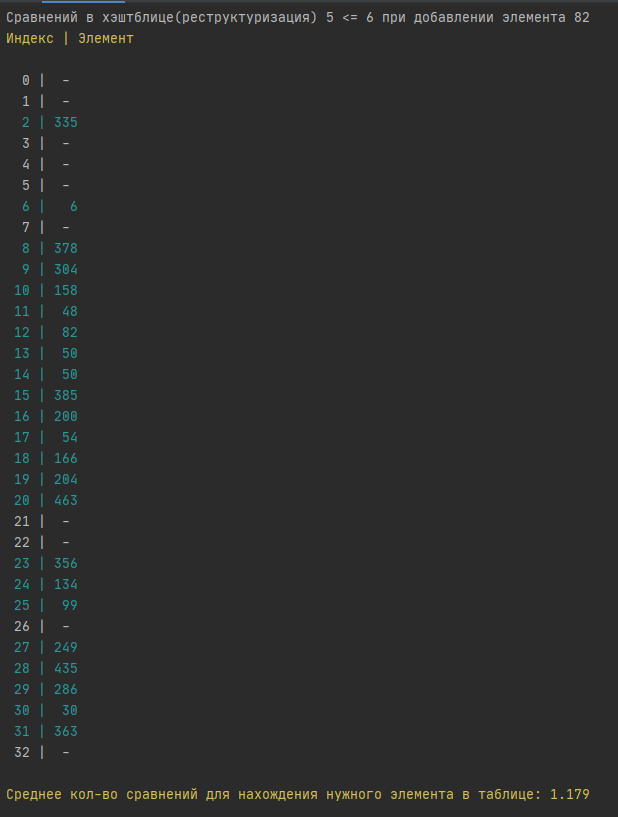
Используя заданную хеш-функцию находится индекс для добавление в хеш-таблицу, если этот место занято, то смотрится следующее место в хеш-таблице, до тех пор пока не будет найдено свободное место и оно будет заполнено.

Затем вводится число сравнений, если число сравнений при добавлений больше указанного, то производится рекструктуризация хеш-таблицы, то есть ведется поиск новой хеш-функции, пока не будет найден, ток количество сравнений, которое меньше или равно указанному.

После выводится результат, прошлой хеш-функции и новой хеш-функции:





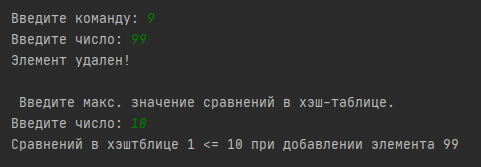
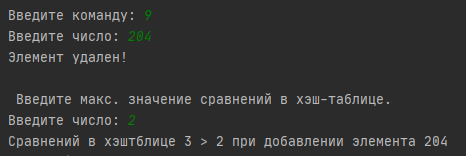
 

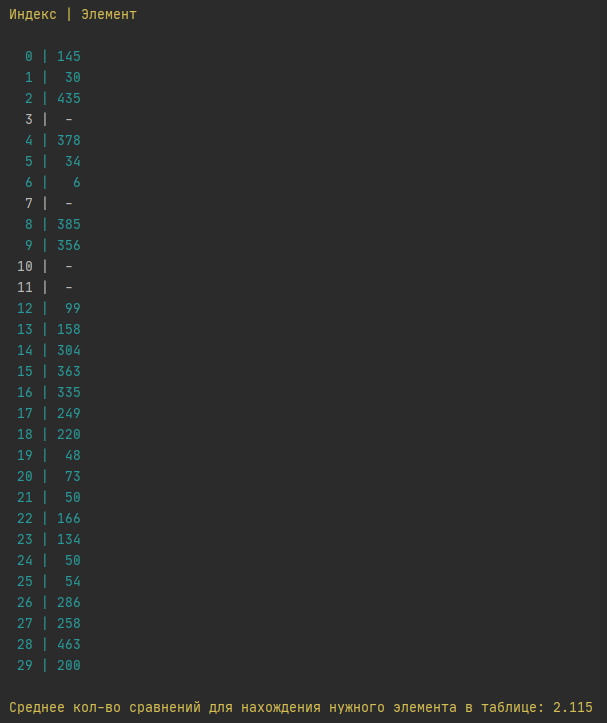
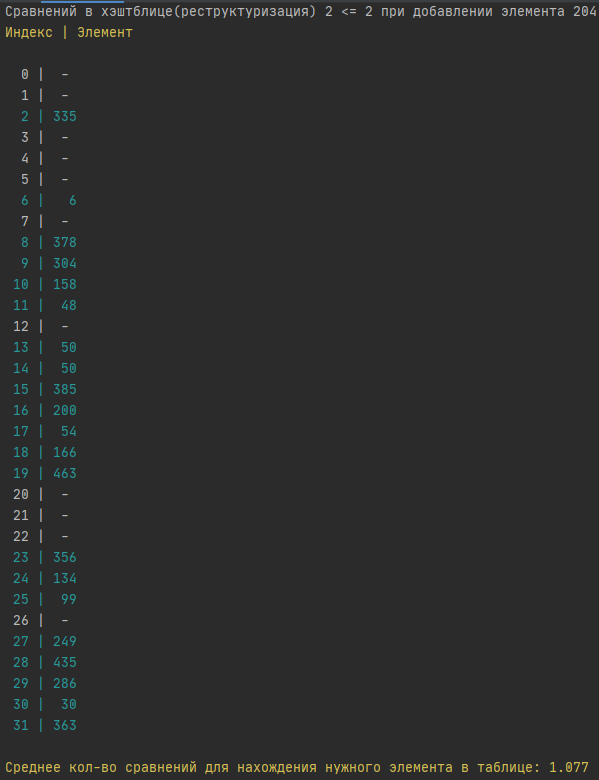
*Пункта 9 консольного меню:*

Удаление элемента в хеш-таблицу. Удаление происходит следующим образом:

Используя заданную хеш-функцию находится индекс для удаления элемента из хеш-таблицы, затем если это место занято и элемент нужный для удаления, то удаляется элемент и место становится свободным, в противном случае, проходим до конца хеш-функции, пока не будет найден элемент для удаления.

Затем, как при пункте добавления, вводится число сравнений и проводится рекструктруризация.

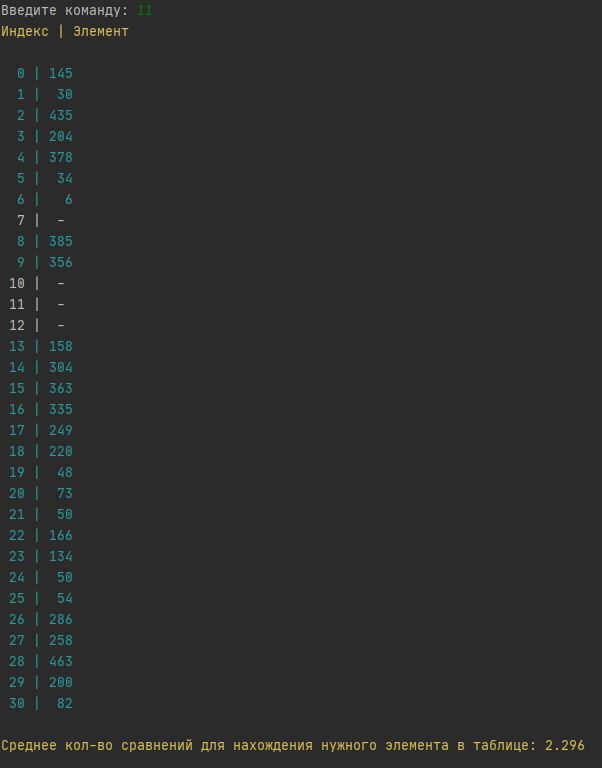
 

*Пункта 10 консольного меню:*

Поиск элемента в хеш-таблице, если найден выводится сообщение о том, что элемент найден, с определенным количеством сравнений.

*Пункта 11 консольного меню:*

Вывод хеш-таблицы.



*Пункт 0 консольного меню* Выход из программы и очищение памяти дерева и хеш-таблицы

Функции для работы с деревьями

//функция устанавливает значения структуры по умолчанию

void default\_tree(tree\_t \*tree);

// создания узла дерева  
branch\_t \*create\_element\_tree(int element);

// добавления узла в дерево (ДДП и АВЛ)  
int push\_tree(branch\_t \*\*head, int number, \_Bool \_isbalance);

// чтения файла с целыми числами как дерево (ДДП и АВЛ)  
int fread\_tree(FILE \*f, tree\_t \*tree, \_Bool \_isbalance);

//очищения дерева   
void free\_branch(branch\_t \*head);  
void free\_tree(tree\_t \*tree);

// нахождение высоты начиная узла  
int find\_branch\_height(branch\_t \*head);

// нахождение всех высот дерева  
void find\_all\_height(branch\_t \*head);  
int height(branch\_t \*head);

// нахождение разницу между высотами правого и левого узла  
int bfactor(branch\_t \*head);  
void fixheight(branch\_t \*p);

// поворот в право узел  
branch\_t\* rotateright(branch\_t \*right);

// поворот в право узел  
branch\_t\* rotateleft(branch\_t \*left);

// балансировка узла  
branch\_t\* balance\_branch(branch\_t \*head);

// балансировка дерева  
branch\_t \*balance\_tree(branch\_t \*head);

// нахождение минимального элемента дерева  
branch\_t \*findmin(branch\_t \*head);

// удаления минимального элемента  
branch\_t\* pop\_min(branch\_t \*head, \_Bool \_isbalance) ;

// удаление элемента их дерева  
branch\_t \*pop\_branch\_tree(branch\_t \*head, int pop\_number, \_Bool \_isbalance);

// поиск элемента в деревк  
int search\_element(branch\_t \*head, int number);  
// вывод дерева  
void print\_tree(tree\_t tree);

Функции работы с хеш-таблицей

//функция устанавливает значения структуры по умолчанию

void default\_hash(hash\_table\_t \*hash);

// хеш-функция  
int hash\_func(int num, int divide);

// установления делителя для хеш-функции  
int select\_prime(int max);

// добавления числа в хеш-таблицу  
int push\_hash(hash\_table\_t \*hash\_table, int num);

// чтение целых чисел из файла в хеш-таблицу  
int fread\_hash(FILE \*f, hash\_table\_t \*hash\_table);

// удаления элемента из жеш-таблицы  
int pop\_hash(hash\_table\_t \*hash\_table, int num);

// поиск элемента в хеш-таблице  
int search\_hash(hash\_table\_t \*hash\_table, int search);

// реструктуризация хеш-таблицы (поиск новой хеш-функции)  
hash\_table\_t restruct(hash\_table\_t table, int search, int max);  
// вывод хеш-таблицы  
void print\_hash(hash\_table\_t \*table);

***Аварийные выходы из программы***

Выходом из программы является некорректное чтение файла, несуществующий файл или пустой файл.

При взаимодействии с меню – ошибка выделения памяти под дерево или хеш-таблицу.

Выходов аварийных нет, в том случае, если программа не будет принудительна закрыта.

Выход осуществляется только пунктом меню «0»

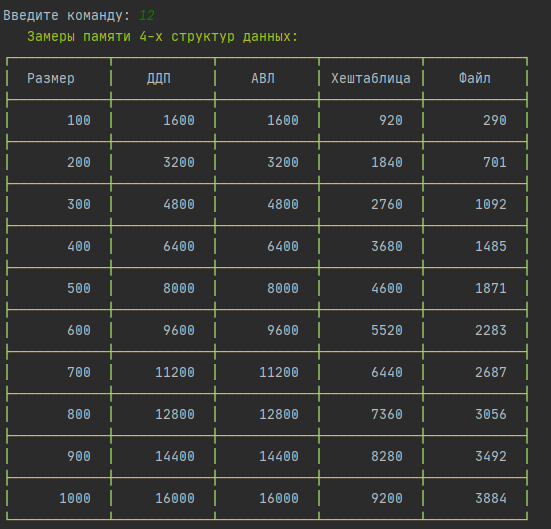
В ошибочных случаях выводится сообщение об ошибке и ввод повторяется, пока программа не получит корректные данные.

Если в результате операции – умножения или вводе матрицы и векторов, память не была выделена, то пользователь возвращается в главное меню и выводится сообщение об не выделении память в определенной операции.

***Анализ результатов эффективности***

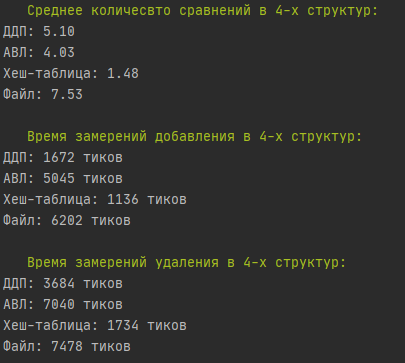
*Запускается пунктом меню – 12*

*Замеры по памяти:*



Замерив память работы от 100 – 1000 элементов, где из 4-х структур данных наиболее работы с памятью – это работа с файлом(60-70% эффективнее хеш-таблицы и 65 – 85% эффективнее деревьев), затем уже с хеш-таблицей(35-50% эффективнее деревьев), а затем деревья, между ДДП и АВЛ, различий нет, так их структуры одинаковы.

*Замеры времени и сравнений*



Результаты средних значений сравнений 4-х структур данных, что наиболее эффективное структура является хеш-таблица, в среднем значение у этой структуры 2-3 сравнений при 100 элементов заполненности.

Самым менее эффективным оказалось работа с файлами, где сравнений в среднем оказалось 50 при 100 элементов заполненности.

Время добавления и удаления элементов из структур, показало, что эффективнее работать с деревьями или хеш-таблицами, чем с файлом(эффективность 40%)

Эффективность ДДП при добавлении и удаления элементов, наиболее эффективнее, чем в АВЛ. Но поиск элементов в АВЛ является наиболее быстрым, чем в ДДП.

***Контрольные вопросы***

**1.Что такое дерево?**

Дерево – нелинейная структура данных, которая используется для представления иерархических связей «один ко многим». Дерево с базовым типом Т определяется рекурсивно: это либо пустая структура (пустое дерево), либо узел типа Т с конечным числом древовидных структур того же типа – поддеревьев.

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

Выделение памяти под деревья определяется типом их представления. Это может быть таблица связей с предками (№ вершины - № родителя), или связный список сыновей. Оба представления можно реализовать как с помощью матрицы, так и с помощью списков. При динамическом представлении деревьев (когда элементы можно удалять и добавлять) целесообразнее использовать списки – т.е. выделять память под каждый элемент динамически.

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

Обход, поиск, добавление и удаление элемента.

**4. Что такое дерево двоичного поиска?**

Дерево двоичного поиска – дерево, в котором все левые потомки «моложе» предка, а все правые – «старше». Это свойство выполняется для любого узла, включая корень.

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

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем

на 2.

Идеально сбалансированное: для каждой его вершины количество вершин в левом и правом поддереве различаются не более чем на 1.

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

Алгоритм поиска одинаковый.

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

Массив, заполненный в порядке, определенным хеш-функцией, называется хеш-таблицей.

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

Может возникнуть ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда h(K1) = h(K2), в то время как K1 ≠ K2. Такая ситуация называется коллизией. Для устранения коллизий нужно выбрать другую хеш-функцию (или эту же с другими параметрами) и перегенерировать таблицу.

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

При большом количестве коллизий.

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

- В хеш-таблице минимальное время поиска О(1).

- В АВЛ дереве: О(log\_2n).

- В ДДП О(h), где h - высота дерева (от log\_2n до n).

***Вывод***

В ходе выполнения работы была реализована программа, работающая с двумя типами деревьев и хеш-таблицами. Были реализованы функции добавления, удаления и поиска для деревьев и хеш-таблицы.

Добавление и удаление элемента в хеш-таблице происходит на 80% быстрее, чем в обычном дереве, и на 35% быстрее, чем в АВЛ-дереве.

ДДП и АВЛ-дерево требуют примерно одинаковое количество памяти. а Хеш-таблица требует на 50% меньше памяти.

Эффективность хеш-таблицы зависит от среднего количества допустимых коллизий. При большом значении этого показателя, хеш-таблицы становятся не эффективными, увеличивается время удаления и среднее количество необходимых сравнений.

Таким образом, представление данных в виде хеш-таблицы эффективнее, чем в виде деревьев, а АВЛ-дерево эффективнее, чем дерево двоичного поиска, так как оно сбалансированное и требует меньшее количество сравнений.

В задачах связанных с быстром поиском случайных данных лучше всего подходит Хэш-таблица с правильно подобранной хэш-функцией для данного типа данных. АВЛ дерево разумно использовать при большом кол-ве коллизий в хэш-таблице (т.е. если подобрать хэш-функцию не удалось).