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

### Студент Блохин Артем Олегович

*фамилия, имя, отчество*

### Группа ИУ7-34Б

Тип лабораторной работы Учебная

### Название

предприятия

|  |  |  |
| --- | --- | --- |
| Студент |  | Блохин А.О. |
| Преподаватель | *подпись, дата* | *фамилия, и.о.*  Силантьева А.В. |
|  | *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*** *- получить навыки применения двоичных деревьев, реализовать основные операции над деревьями: обход деревьев, включение, исключение и поиск узлов.*

***Условие задачи***

***Вариант 2***

Построить дерево в соответствии со своим вариантом задания. Вывести его на экран в виде дерева. Реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов. Сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления.

***Техническое задание***

*Построить двоичное дерево поиска, в вершинах которого находятся слова из текстового файла. Вывести его на экран в виде дерева. Определить количество вершин дерева, содержащих слова, начинающиеся на указанную букву. Выделить эти вершины цветом. Сравнить время поиска начинающихся на указанную букву слов в дереве и в файле.*

***Входные данные***

В качестве указания пункта меню – целое число (0 – 7).

В подпунктах – по 0-1 целых числа

***Выходные данные***

При выводе Дерева – вывод дерева, содержащий целые числа.

Данные об эффективности.

***Описание алгоритма:***

В реализации программы была создана структура для хранения Дерева:

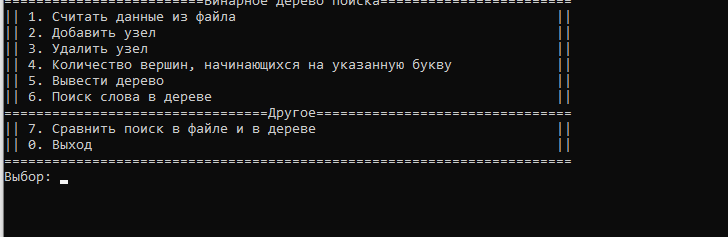
*Дерево*

typedef struct branch  
{  
 char \*word; // строка

struct branch \*parent; // родительский узел вершины  
 struct branch \*left; // левый узел вершины  
 struct branch \*right; // правый узел вершины  
} branch\_t;

typedef struct st\_tree  
{  
 struct branch \*head; // указатель на первую вершину дерева  
 int size; // количество элементов  
} binary\_tree\_t;

Для взаимодействия с программой было создано консольное меню:



Файл уже подготовлен для работы с программой. Файл содержит 80 элементов.

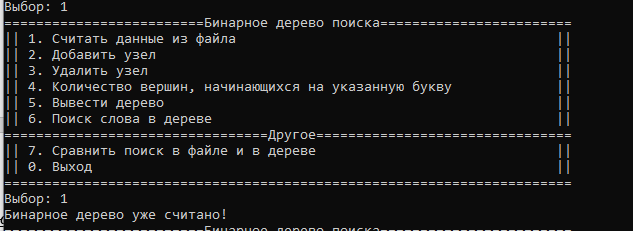
При вводе корректного значения выполняются определенные действия или операции, при некорректном - выводится в консоль сообщение об ошибке, и программа завершается.

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

Если дерево пусто и при выборе пунктов: 2, 3, 4, 5, 6 – Выводится сообщение об этом.

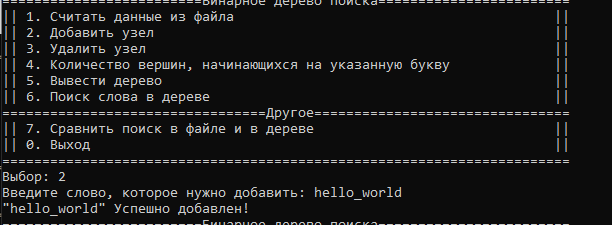
*Пункта 1 консольного меню:*

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



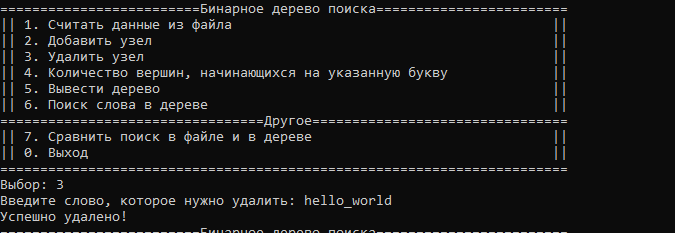
*Пункта 2 консольного меню:*

В этом пункте происходит добавление узла в дерево. Если дерево еще не проинициализировано, то программа выведет сообщение об этом.



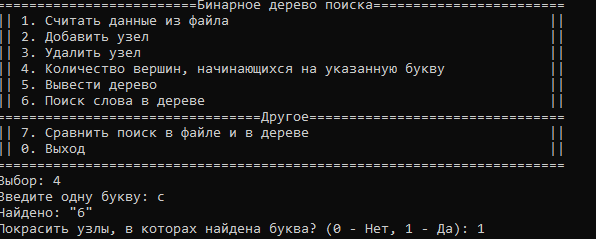
*Пункта 3 консольного меню:*

Удаление узла из дерева. Если дерево еще не проинициализировано, то программа выведет сообщение об этом.

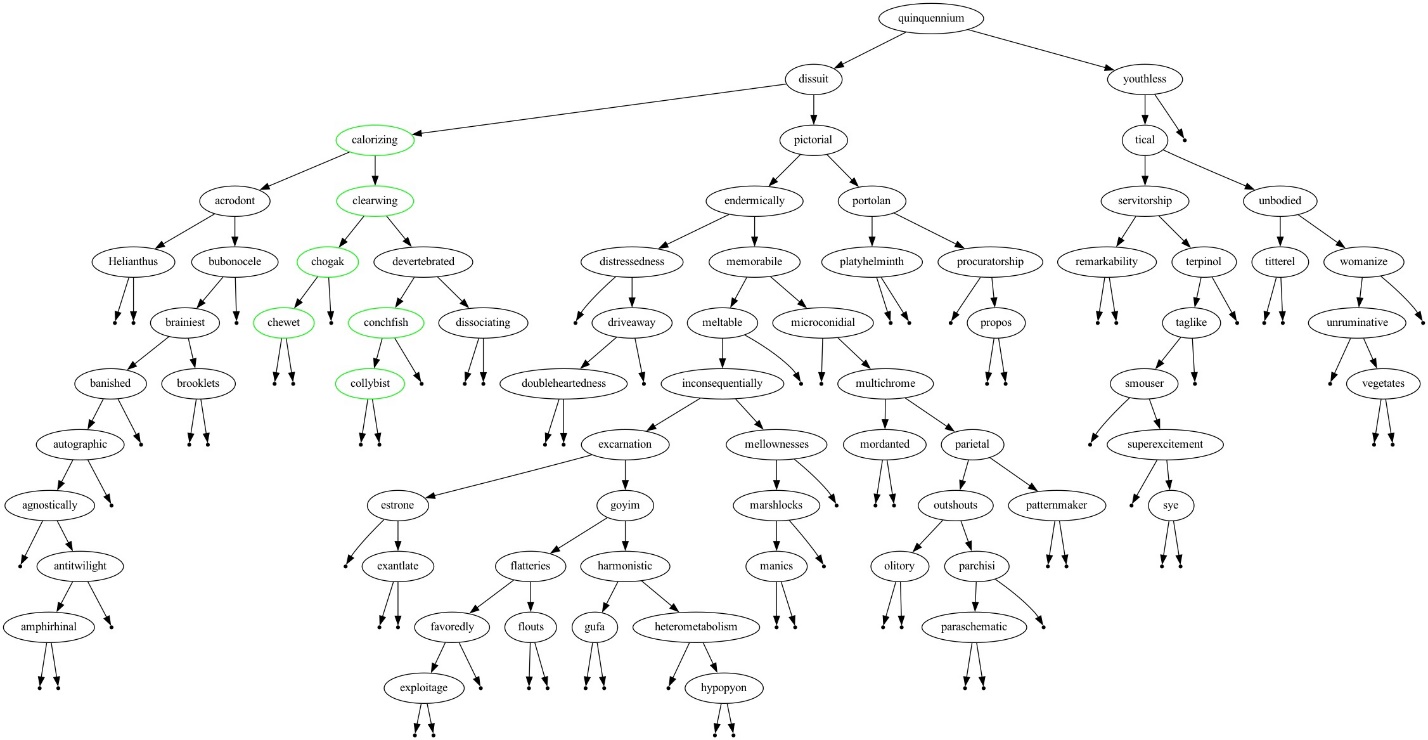


*Пункта 4 консольного меню:*

Выполнение пункта по заданию: “Количество вершин, начинающихся на указанную букву”. Если дерево еще не проинициализировано, то программа выведет сообщение об этом.

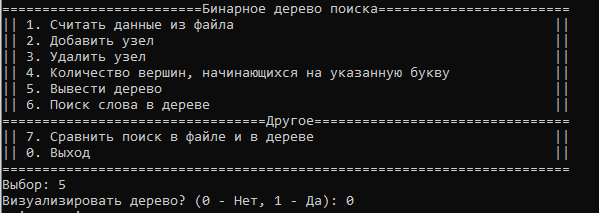


Также, присутствует, возможность визуализировать найденные узлы:

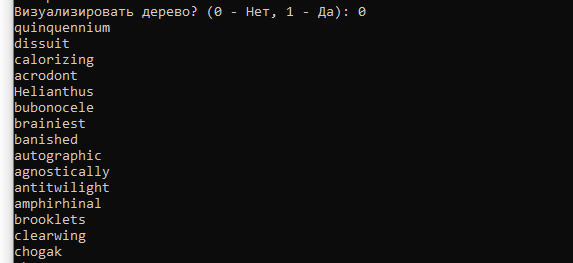


*Пункта 5 консольного меню:*

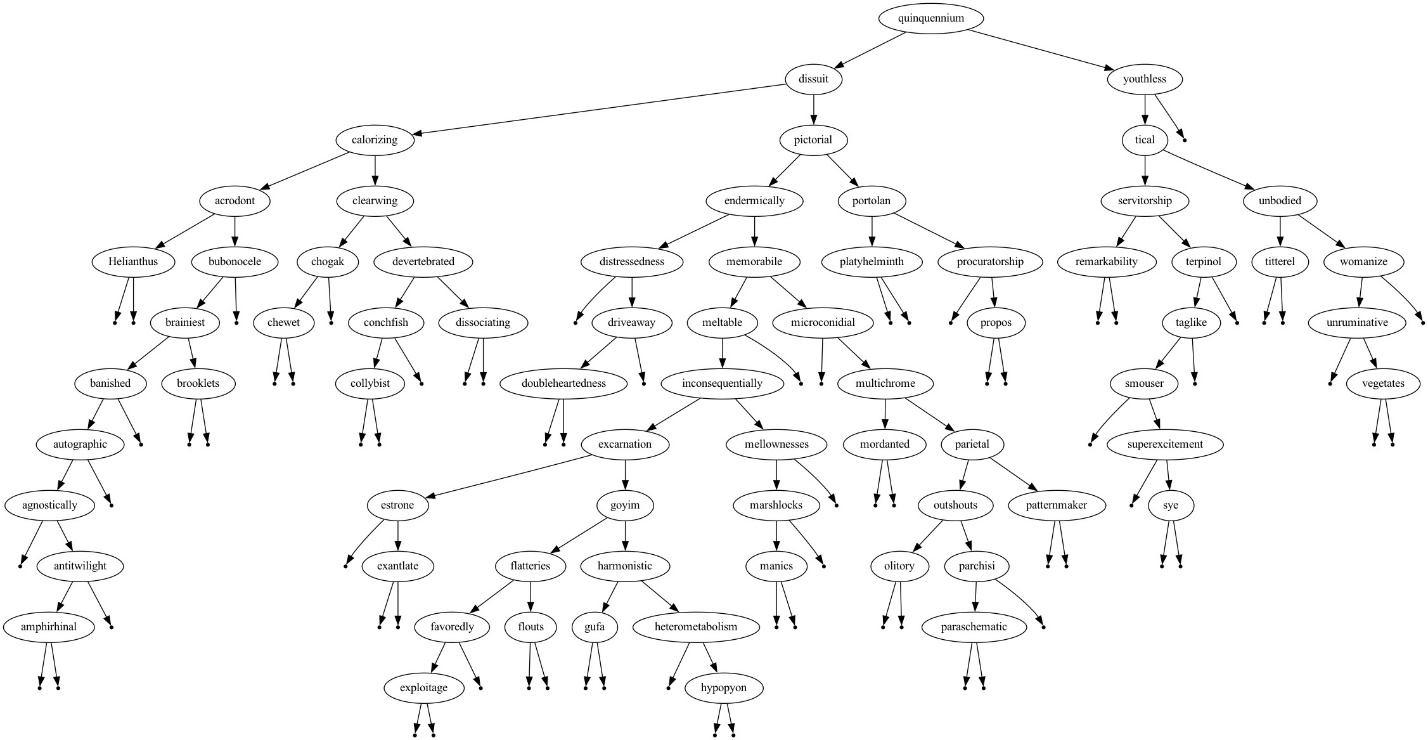
Вывод дерева. Если дерево еще не проинициализировано, то программа выведет сообщение об этом.



Если дерево не визуализировать, то выведутся все узлы дерева в строчку.

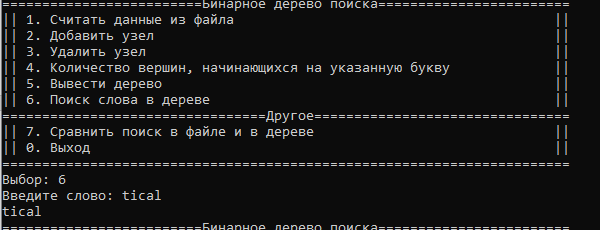


Если будет выбран пункт меню “визуализировать”, то будет создан png файл:



*Пункта 6 консольного меню:*

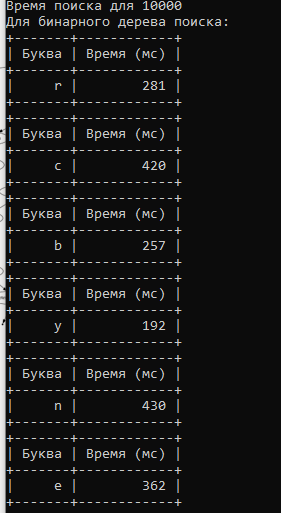
Поиск слова в дереве. Если в дереве не существует данного слова, то выводится сообщение об ошибке.

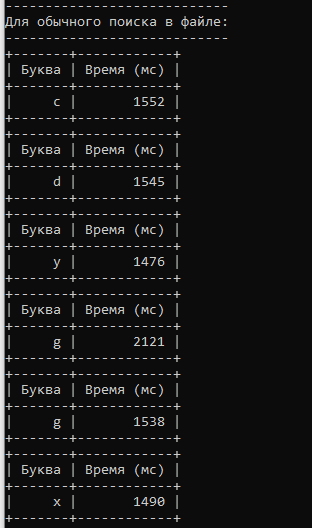


*Пункта 7 консольного меню:*

Сравнение времени поиска в файле и в дереве.

Сравнение происходит при количестве записей в файле: 50, 500, 1000, 1500, 10000. И случайной буквой.





*Пункт 0 консольного меню*

Выход из программы и очищение памяти дерева.

Функции для работы с деревьями

// добавления узла в дерево (ДДП)

void insert(branch\_t \*\*head, char \*value);

// удаление узла из дерева (ДДП)

int deleteValue(branch\_t \*root, char \*value);

// чтения файла со строками в дерево (ДДП)

int file\_read(binary\_tree\_t \*binary\_tree, FILE \*f);

// поиск узла в дереве (ДДП)  
branch\_t \*getNodeByValue(branch\_t \*root, char \*value);

// вычисление количества слов, начинающихся на букву  
size\_t get\_quantity\_by\_letter(branch\_t \*root, char value);   
// вывод дерева

void print\_tree(branch\_t \*head);

***Аварийные выходы из программы***

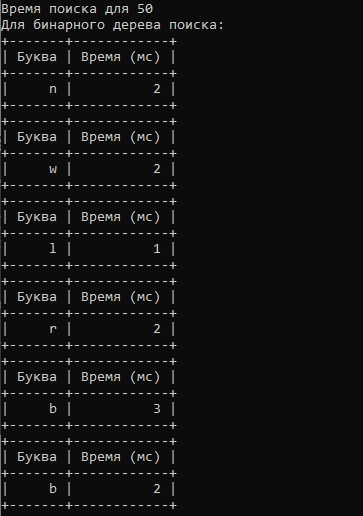
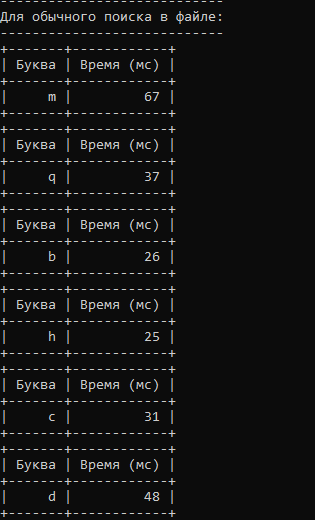
Выходов аварийных нет, в том случае, если программа не будет принудительна закрыта.

Выход осуществляется только пунктом меню «0»

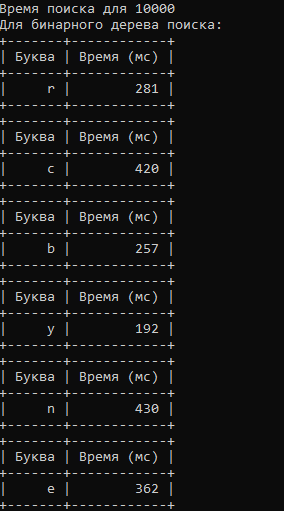
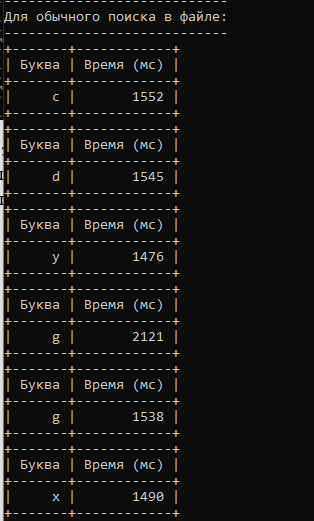
***Анализ результатов эффективности***

*Запускается пунктом меню – 7*

*Замеры времени и сравнений*

Замерив время поиска от 50 – 10000 элементов, видно, что бинарное дерево эффективнее по скорости поиска. Для небольшого количества время не столь значительно. Но, если увеличить количество слов до 10000, то результат уже будет значителен.

  ***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 бывают типы деревьев?**

* N-арное дерево;
* Сбалансированное дерево;
* Бинарное дерево;
* Бинарное дерево поиска;
* Дерево AVL;
* Красное-черное дерево;
* 2-3 дерево.

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

Обход, поиск, добавление и удаление элемента.

**5. 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.

***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).