1. Описание условия задачи.

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.

2. Описание технического задания.

Построить дерево поиска из слов текстового файла, сбалансировать полученное дерево. Вывести его на экран в виде дерева. Удалить все слова, начинающиеся на указанную букву, в исходном и сбалансированном дереве. Сравнить время удаления, объем памяти. Построить хеш-таблицу из слов текстового файла. Вывести построенную таблицу слов на экран. Осуществить поиск и удаление введенного слова, вывести таблицу. Выполнить программу для различных размерностей таблицы и сравнить время удаления, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц

Входные данные.

Целое число: пункт меню (от 0 до 13).

```
1) Построить бинарное дерево из файла
2) Добавить слово в бинарное дерево
3) Извлечь элемент из бинарного дерева
4) Найти элемент в дереве.
5) Удалить все слова, начинающиеся на указанную букву в бинарном дереве
6) Вывести бинарное дерево на экран
7) Построить хеш-таблицу из файла
8) Добавить слово в хеш-таблицу
9) Удалить слово из хеш-таблицы
10) Найти слово в хеш-таблице
11) Удалить все слова, начинающиеся на указанную букву в хеш-таблице
12) Вывести хеш-таблицу
13) Вывести замеры эффективности
0) Завершить работу программы
```

Строка: путь к файлу
Строка: вставляемый элемент
Строка: удаляемый элемент

Строка: буква для удаления слов.

Выходные данные.
Сообщение об успехе или ошибке выполнения пункта, изображение дерева, хеш-таблица, таблица замеров по времени и по памяти.

Способ обращения к программе.
Запускается через терминал командой *./app*

Возможные аварийные ситуации и ошибки пользователя.
Некорректный ввод данных, удаление из пустого дерева или из пустой хеш-таблицы. Ошибка выделения памяти.

3. Описание внутренних структур данных.

Структура вершины дерева.
```
typedef struct tree_node
{
   char *str; //Данные
   struct tree_node *parent; // Указатель на родителя
   struct tree_node *left; // Указатель на левого потомка
   struct tree_node *right; // Указатель на правого потомка
   unsigned char height; // Высота поддерева
} tree_node_t;
```

Структура дерева.
```
typedef struct {
   tree_node_t *root;
   int is_avl; // 1 для AVL, 0 для обычного дерева
} tree_t;
```

Структура узла открытой хеш-таблицы.
```
typedef char *ht_key_t;

typedef struct node
{
   ht_key_t key; // Строка
   struct node *next; // Указатель на следующий узел
} key_node_t;
```

Структура открытой хеш-таблицы.

```
typedef struct
{
   key_node_t **keys; // Массив списков ключей
   int size; // Размер хеш-таблицы
   int count; // Кол-во элементов в хеш-таблице
} open_hash_table_t;
```

Структура закрытой хеш-таблицы.

```
typedef struct
{
   ht_key_t *keys; // Массив ключей
   int size; // Размер хеш-таблицы
   int count; // Кол-во элементов в хеш-таблице
} closed_hash_table_t;
```

Интерфейс для работы со структурами.

Для работы с деревьями

```
// Очищение памяти, ввыделенной под дерево
void free_tree(tree_node_t *tree_node);

// Создание вершины дерева
tree_node_t *create_node(char *str);

// Вставка вершины в дерево
tree_node_t *tree_node_insert(tree_node_t *root, tree_node_t *node, int
is_balanced);

// Чтение дерева из файла
int tree_read(FILE *file, tree_node_t **root_bst, tree_node_t **root_avl);

// Поиск вершины в дереве
tree_node_t *tree_node_search(tree_node_t *root, const char *str);

int tree_node_search_cmps(tree_node_t *root, const char *str);

// Удаление вершины из дерева
tree_node_t *tree_node_remove(tree_node_t *root, tree_node_t *removed_node, int
is_avl);

// Удаление всех вершин, начинающихся на задааную букву
```

```
tree_node_t *tree_remove_by_first_letter(tree_node_t *root, char letter, size_t
*removed_elems, int is_avl);

// Вывод дерева на экран
int open_dot_img(const char *file_name, tree_node_t *root);
```

Для работы с открытой хеш-таблицей

```
// Очищение данных хеш-таблицы
void clear_open_ht_data(open_hash_table_t *hash_table);

// Инициализация хеш-таблицы
int create_open_ht(open_hash_table_t *hash_table);

// Вставка в хеш-таблицу
void insert_open_ht(open_hash_table_t *hash_table, char *str);

// Поиск в хеш-таблице
char *search_open_ht(open_hash_table_t *hash_table, char *str);

// Кол-во сравнений для поиска
int cmps_search_open_ht(open_hash_table_t *hash_table, char *str);

// Удлаение элемента из хеш-таблицы
void delete_open_ht(open_hash_table_t *hash_table, char *str);

// Удаление всех элементов, начинающихся на заданную букву
void del_by_first_letter_open_ht(open_hash_table_t *hash_table, char letter);

// Чтение хеш-таблицы из файла
int fread_open_ht(FILE *file, open_hash_table_t *hash_table);

// Вывод хеш-таблицы
void print_open_ht(open_hash_table_t *hash_table);

// Реструктуризация хеш-таблицы
int restruct_open_ht(open_hash_table_t *hash_table);
```

Для работы с закрытой хеш-таблицей

```
// Очищение данных хеш-таблицы
void clear_closed_ht_data(closed_hash_table_t *hash_table);

// Инициализация хеш-таблицы
int create_closed_ht(closed_hash_table_t *hash_table);
```

```
// Вставка в хеш-таблицу
void insert_closed_ht(closed_hash_table_t *hash_table, char *str);

// Поиск в хеш-таблице
char *search_closed_ht(closed_hash_table_t *hash_table, char *str);

// Кол-во сравнений для поиска
int cmps_search_closed_ht(closed_hash_table_t *hash_table, char *str);

// Удлаение элемента из хеш-таблицы
void delete_closed_ht(closed_hash_table_t *hash_table, char *str);

// Удаление всех элементов, начинающихся на заданную букву
void del_by_first_letter_closed_ht(closed_hash_table_t *hash_table, char letter);

// Чтение хеш-таблицы из файла
int fread_closed_ht(FILE *file, closed_hash_table_t *hash_table);

// Вывод хеш-таблицы
void print_closed_ht(closed_hash_table_t *hash_table);

// Реструктуризация хеш-таблицы
int restruct_closed_ht(closed_hash_table_t *hash_table);
```

4. Описание алгоритма.

Для двоичного дерева поиска:

Алгоритм поиска:
При поиске элемента на каждом шаге сравнивается значение ключа и значение в текущем листе. Если значение ключа больше, то переходим в правое поддерево, если меньше – в левое, если одинаковы – элемент найден. Если нужного поддерева нет, то нужного элемента не существует. Максимальная сложность поиска O(n).

Алгоритм вставки:
При вставке сначала идёт поиск. Если элемент найден, то он заменяется, иначе к последнему просмотренному элементу присоединяется новый элемент в ту сторону, в которой он должен был бы искаться при обходе. сложность как при поиске.

Алгоритм удаления:

При удалении тоже сначала идёт поиск, а потом либо просто удаляется (если нет потомков), либо заменяется на единственное поддерево, либо (при наличии двух поддеревьев) заменяется на самого маленького (самого левого) потомка справа. Сложность как при поиске.

Для АВЛ дерева:
Для АВЛ дерева аналогично с двоичным деревом поиска, но на обратном пути перерасчитываются высоты поддеревьев (для каждого поддерева определяется как увеличенная на единицу максимальная из двух высот поддеревьев потомка), после чего при нарушении критерия сбалансированности совершается нужный поворот. Максимальная сложность поиска при этом сокращаеттся до O(logn).

Для хеш-таблицы с открытым хешированием:

Алгоритм поиска:

При поиске элемента высчитывается значение хеш-функции, после чего идёт просмотр цепочки для соответствующего элемента таблицы. Если элемента нет, то его не будет в данной цепочке. Так как длина цепочек ограничена 3 элементами, то сложность поиска O(1).

Алгоритм удаления:

При удалении сначала идёт поиск, потом найденный элемент исключается из цепочки, а ссылка в предыдущем элементе или в таблице переставляется на следующий или обнуляется при отсутствии следующего. Сложность удаления O(1).

Алгоритм вставки:

При вставке тоже сначала ищется место, если элемент существовал, то просто меняется значение, иначе добавляется в конец цепочки. Если длина цепочки больше 3, то происходит реструктуризация таблицы. Без реструктуризации сложность поиска тоже O(1).

Алгоритм реструктуризации:

При реструктуризации размер таблицы увеличивается на следующее простое число, после чего все элементы перемещаются в новый массив. Реструктуризация будет продолжаться до тех пор, пока длины всех цепочек не окажутся меньше 4. Сложность реструктуризации O(n).

Для хеш-таблицы с закрытым хешированием:

Сложности всех функций аналогичны.

При поиске вместо обхода по цепочки будут просмотрены следующие индексы вычисленные квадратичным шагом, т.е $h = h + a^2$, где a — номер попытки. Для избежания выхода за границы таблицы индексы берутся по модулю размера таблицы.

При удалении просто удаляется найденный элемент.

При вставке вычисляется индекс в таблице, и если этот индекс занят, то будут просматриваться следующие индексы (аналогично в поиске), пока не будет найден свободный индекс.

Реструктуризация аналогична предыдущей таблице.

5. Замеры.

Количество итераций = 10000.

Таблица среднего времени поиска

Количество элементов	Время для обычного дерева	Время для АВЛ дерева	Время для открытой хеш-таблицы	Время для закрытой хеш-таблицы
10	0.03	0.0323	0.047	0.0484
50	0.0449	0.0408	0.0433	0.0446
75	0.05	0.0461	0.0446	0.0485
100	0.0613	0.0456	0.0421	0.0503
150	0.0555	0.0496	0.043	0.0501
250	0.0756	0.0561	0.0499	0.0593
300	0.0833	0.0639	0.0553	0.059

По таблице можно увидеть, что поиск в хеш-таблице (как в открытой так и закрытой) не зависит, от количества элементов, и всегда выполняется примерно за одно и то же время, а в древовидных структурах, заметен рост времени поиска при увелечении их размера, особо сильно а обычном, несбалансированном.

Таблица среднего количества сравнений для поиска

Количество элементов	Количество сравнений для обычного дерева	Количество сравнений для АВЛ дерева	Количество сравнений для открытой хеш-таблицы	Количество сравнений для закрытой хеш-таблицы
10	3.105	2.9028	1.2905	2.0989
50	5.8506	5.0045	1.3542	1.3672
75	6.6228	5.5071	1.3069	1.924
100	8.0557	5.8678	1.3525	2.0285
150	8.1382	6.4304	1.2551	2.064
250	9.1245	7.176	1.4263	2.6856
300	10.3065	7.555	1.3551	2.2882

По этой таблице, можно увидеть, что наименьшее количество сравнений требует открытая хеш-таблица, закрытая хеш-таблицы в среднем требует чуть больше сравнений, из-за того, что не всегда сразу удается найти пустую ячейку таблицы. Количество сравнений в дереве в свою очередь напрямую зависит от количества элементов (для АВЛ это log(n), для обычного колеблется от log(n) до n).

Таблица времени удаления начинающихся на заданную букву

Количество элементов	Время для обычного дерева	Время для АВЛ дерева	Время для открытой хеш-таблицы	Время для закрытой хеш-таблицы
10	0.21	0.25	0.21	0.23
50	0.32	0.33	0.22	0.2
75	0.63	0.68	0.51	0.5
100	0.86	0.87	0.91	0.82
150	1.0	1.02	1.13	0.93

250	2.06	1.93	2.18	2.53
300	3.1	2.8	2.7	3.39

По этой таблице, можно увидеть, что скорость удаления элементов не зависит от структуры, а зависит от количества элементов в структуре, так как нам нужно обойти все данные и сравнить, подходит ли слово для удаления. Скорость же удаления в закрытой хеш-таблице компенсируется её размером.

Таблица занимаемой памяти

Количество элементов	Обычное дерево	АВЛ дерево	Открытая хеш-таблица	Закрытая хеш-таблица
10	400	400	160	40
50	2000	2000	800	200
75	3000	3000	1200	300
100	4000	4000	1600	400
150	6000	6000	2400	600
250	10000	10000	4000	1000
300	12000	12000	4800	1200

По этой таблице видно, что древовидные структуры занимают больше всего памяти, так как они хранят ссылки на своих предшественников. Самая экономичная же по памяти, оказалась закрытая хеш-таблица, так как в отличие от открытой, которая хранит массив списков, она хранит лишь массив ключей.

6. Выводы по проделанной работе.

Деревья являются хорошим вариантом поиска, но они занимают (сравнительно) большое количество памяти и сложность поиска зависит от высота дерева. Недостаток AVL дерева в сравнении с обычным деревом поиска заключается в том, что добавление элементов в него требует балансировки дерева, что сильно замедляет добавление элементов и усложняет программу. Также один из плюсов деревьев поиска, это то, что в них можно достатчоно быстро выолнить обход по возрастанию или убыванию ключей, чего нельзя добиться в хеш-таблице.

Одной из самых оптимальных структур данных для поиска является хеш-таблица, так как поиск выполняется всегда за константное время O(1). Также хеш-таблица является достатчно экономичной по памяти, особенно хеш-

таблица с закрытой адресацией (по сравнению с деревьями). Однако у хеш-таблицы есть существенные минусы в виде коллизий, которые могут замедлить время обработки, если их не устранять правильным образом. При увелечении коллизий, требуется реструктуризация, что достаточно ресурсоёмко по памяти и по времени. Также, на мой взгляд, реализация хеш-таблицы сложнее нежели чем реализация двоичного дерева поиска.

Контрольные вопросы.

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

У ИСД число вершин в левом и правом поддеревьях отличается не более, чем на единицу и при построении ИСД не учитывается значение узла, то есть такое дерево не является двоичным деревом поиска, в АВЛ у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу, и оно является деревом поиска.

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

С точки зрения алгоритма, поиск ничем не отличается. При поиске в АВЛ-дереве количество сравнений будет обычно меньше, чем в дереве двоичного поиска.

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

Хеш-таблица — это массив, позиция элементов в котором определяется хэш-функцией, которая позволяет определить позицию каждого элемента.

4. Что такое коллизии? Каковы методы их устранения?

Коллизия — это ситуация, когда хеш-функция определяет один и тот же индекс для двух разных ключей. Основные методы устранения: метод цепочек и открытая адресация. Коллизия первым методом устраняется добавлением элемента в конец связанного списка, который содержится в ячейке по хеш индексу. Вторым методом — просматриваются следующие записи таблицы по порядку с некоторым шагом. Если шаг постоянный, то такой вид адресации называют линейным, а если же берётся квадрат, от попытки поиска ключа, то квадратичной адресацией.

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

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

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

В хэш таблицах - O(1)
В АВЛ - O(log n)
В Двоичном дереве поиска — зависит от высоты, поэтому от O(log n) до O(n).