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

Группа ИУ7-35Б

Тема работы **Деревья, хеш-таблицы**

Студент Баранов Николай Алексеевич

Преподаватель Силантьева Александра Васильевна

2023 г.

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

1. Описание условия задачи

Построить хеш-таблицу для зарезервированных слов языка C++, содержащую HELP для каждого слова. Выдать на экран подсказку по введенному слову. Выполнить программу для различных размерностей таблицы и сравнить время поиска и количество сравнений. Для указанных данных создать сбалансированное дерево. Добавить подсказку по вновь введённому слову, используя при необходимости реструктуризацию таблицы. Сравнить эффективность добавления ключа в таблицу или её реструктуризацию для различной степени заполненности таблицы.

2. Описание ТЗ

2.1. Описание исходных данных и результатов.

При чтении ключевого слова или его описания программа принимает на вход произвольную непустую строку.

При чтении числа программа считывает число в указанном диапазоне.

Выбор действия происходит в меню, где вводится число – номер действия.

Любое дерево при необходимости сохраняется в файле в виде графа на языке описания графов DOT. Далее с помощью утилиты graphviz граф можно визуализировать.

Хеш-таблицы можно сохранить в файл формата csv.

При сравнении эффективности поиска выводится общее время работы, среднее количество сравнений и затраченная память на хранение структуры.

При сравнении эффективности добавления выводится общее время работы и среднее количество сравнений. Для хеш-таблиц дополнительно выводится число сделанных реструктуризаций.

2.2. Описание задачи, реализуемой программой.

Программа позволяет ввести информацию о ключевых словах языка C++ и сохранить их в двоичном дереве поиска, АВЛ-дереве, хеш-таблицах с открытой и закрытой адресациями. Также программа сохранить вышеуказанные структуры в файлах.

2.3. Способ обращения к программе.

Запуск программы:

./app.exe

В этом случае программа начнёт выполнять свою задачу. Ввод осуществляется вручную с клавиатуры.

2.4. Описание возможных аварийных ситуаций и ошибок пользователя.

2.4.1. Переполнение буфера. Возникает, если строка при чтении оказалась длиннее чем максимальное количество цифр в числе (вместо этого также может случится ошибка выхода числа из диапазона, если число пометилось в буфер).

2.4.2. Неожиданный символ. Возникает, если в строке присутствуют символы, не предусмотренные программой, или из-за неправильного формата входных данных.

2.4.3. Число не входит в диапазон. Возникает, если пользователь ввёл число, которое не входит в диапазон допустимых значений.

2.4.4. Ошибка при выделении памяти. Практически не зависит от действий пользователя. Возникает только при неудачной попытке выделить память на куче, поэтому вероятность возникновения крайне мала.

3. Описание внутренних структур данных.

Основные используемые структуры:

1) Двоичное дерево поиска:

```
typedef struct node_tree node_tree;

struct tree
{
    node_tree *start;
    size_t size;
    int lastcmp;
};

struct node_tree
{
    char *key;
    char *value;
    node_tree *left, *right;
};
```

tree – дерево.

Её поля:

size – целое число. Хранит размер дерева.

start – указатель на структуру node_tree. Хранит вершину дерева.

lastcmp – целое число. Хранит количество сравнений при последнем поиске или вставке.

node_tree – лист дерева.

Её поля:

key – указатель на строку. Содержит ключевое слово.

value – указатель на строку. Содержит описание ключевого слова.

left, right – указатели на node_tree. Содержат левого и правого потомков листа.

2) АВЛ дерево:

```
typedef struct node_tree node_tree;
```

```
struct avl_tree
{
    node_tree *start;
    size_t size;
    int lastcmp;
};

struct node_tree
{
    char *key;
    char *value;
    node_tree *left, *right;
    int hleft, hright;
};
```

avl_tree – дерево.
Её поля:
size – целое число. Хранит размер дерева.
start – указатель на структуру node_tree. Хранит вершину дерева.
lastcmp – целое число. Хранит количество сравнений при последнем поиске или вставке.
node_tree – лист дерева.
Её поля:
key – указатель на строку. Содержит ключевое слово.
value – указатель на строку. Содержит описание ключевого слова.
left, right – указатели на node_tree. Содержат левого и правого потомков листа.
hleft, hright – целые числа. Содержат высоты левого и правого поддеревьев.

3) Хеш-таблица (с открытым хешированием):

```
typedef struct leaf leaf;

struct hasho
{
    size_t size, bufsize;
    leaf **table;
    int lastcmp;
};

struct leaf
{
    leaf *next;
    char *key;
    char *value;
};
```

hasho – хеш-таблица.
Её поля:
size – целое число. Хранит количество элементов в таблице.
bufsize – целое число. Хранит размер таблицы.
table – массив указателей на структуру leaf. Хранит элементы таблицы.
lastcmp – целое число. Хранит количество сравнений при последнем поиске или вставке.

leaf – элемент таблицы.
Её поля:
key – указатель на строку. Содержит ключевое слово.
value – указатель на строку. Содержит описание ключевого слова.
next – указатель на leaf. Содержит следующий элемент в этой ячейке таблицы.

4) Хеш-таблица (с закрытым хешированием):

```
typedef struct record record;

struct hashc
{
    size_t size, bufsize;
    record *table;
    int lastcmp;
};

struct record
{
    char *key;
    char *value;
};
```

hasho – хеш-таблица.
Её поля:
size – целое число. Хранит количество элементов в таблице.
bufsize – целое число. Хранит размер таблицы.
table – массив указателей на структуру record. Хранит элементы таблицы.
lastcmp – целое число. Хранит количество сравнений при последнем поиске или вставке.
record – элемент таблицы.
Её поля:
key – указатель на строку. Содержит ключевое слово.
value – указатель на строку. Содержит описание ключевого слова.

Вспомогательные структуры:

1) structs_t. Содержит указатели на каждую из 4 вышеуказанных структур. Нужна для сокращения количества передаваемых в функцию параметров.

```
typedef struct
{
    tree *bt;
    avl_tree *bta;
    hasho *hto;
    hashc *htc;
} structs_t;
```

2) times_t. Содержит времена добавления (названия начинаются на t, дальше идёт имя из предыдущей структуры), количество сравнений

(именование аналогично, только a вместо t), для таблиц количество реструктуризаций (rhto и rhtc)? а также количество добавлений в структуры в переменной adds.

```
typedef struct
{
    double tbt, tbta, thto, thtc;
    double abt, abta, ahto, ahtc;
    int rhto, rhtc;
    int adds;
} times_t;
```

4. Описание алгоритма.

Для двоичного дерева 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).

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.

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 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:

Используемая хеш-функция – остаток от деления суммы кодов всех букв на размер таблицы.

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).

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).

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).

При реструктуризации размер таблицы увеличивается в 1.6 раз, 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 дополнительно просмотрены 2 последующих элемента. Для избежания выхода за границы таблицы индексы берутся по модулю размера таблицы.

При удалении просто удаляется найденный элемент.

При вставке сначала идёт проверка на существование элемента в таблице, и если его нет, то ищется первое пустое место. Если подходящее место найдено более чем за 3 сравнения, таблица реструктурируется.

Реструктуризация аналогична предыдущей таблице.

5. Сравнительный анализ разных способов хранения и перемножения.

Для замера времени и памяти при поиске программа запускала 1000 итераций, на каждой из которых измерялось время работы поиска случайно сгенерированного слова. Время работы приведено суммарное.

В таблицах ниже приведены результаты работы программы для разных размеров деревьев и хеш-таблиц.

Для 20 элементов:

Структура	Время, мкс	Память, байт	Среднее количество сравнений
Двоичное дерево поиска	1146	664	19.02
АВЛ дерево	857	824	5.89
Хеш-таблица (открытое хеширование)	667	496	2.73
Хеш-таблица (закрытое хеширование)	764	1120	4

Для 40 элементов:

Структура	Время, мкс	Память, байт	Среднее количество сравнений
Двоичное дерево	1756	1304	27.98

поиска			
АВЛ дерево	1100	1624	6.66
Хеш-таблица (открытое хеширование)	703	976	2.43
Хеш-таблица (закрытое хеширование)	741	2784	4

Здесь видно, что хуже всего по времени и количеству сравнений оказалось двоичное дерево поиска. В худшем случае поиск идёт за O(n), где n – количество элементов. При этом даже по памяти ДДП выигрывает только у АВЛ дерева. В свою очередь, поиск в АВЛ дереве оказался быстрее, чем в ДДП, при этом при увеличении массива вдвое в АВЛ дереве в среднем начало требоваться на 1 сравнение больше, в то время как в ДДП их количество выросло на 47%, так как сложность поиска в АВЛ дереве близится к O(logn). Используемая память при этом в каждом из случаев увеличилась почти вдвое. На хеш-таблицы изменение размера почти никак не повлияло на время работы. При закрытом хешировании используемая память увеличилась примерно на 150%, при открытом – примерно на 100%. Количество сравнений для закрытого хеширования везде 4, так как всегда моделировался худший случай – отсутствие значения в таблице. Однако из-за того, что переход по ссылке дольше, чем переход к следующему элементу, их время поиска примерно одинаковое.

Время добавления замеряется в программе во время работы каждый раз, когда пользователь добавляет элемент в структуры или при начальной инициализации.

В таблице ниже приведены результаты при добавлении 40 элементов.

Структура	Время, мкс	Среднее количество сравнений	Количество реструктуризаций
Двоичное дерево поиска	62	16.78	
АВЛ дерево	52	5.43	
Хеш-таблица (открытое хеширование)	75	2.5	3
Хеш-таблица (закрытое хеширование)	60	5.4	4

Время добавления в структуры неточное из-за относительно малых размеров самих структур. Однако двоичное дерево поиска опять оказалось наихудшим по количеству сравнений. Для АВЛ дерева среднее количество сравнений будет расти, но при этом оно тоже будет близиться к O(logn). У хеш-таблиц среднее количество сравнений изменяться практически не будет, зато

число реструктуризаций будет расти вместе с ростом количества элементов в таблице. Количество реструктуризаций будет расти логарифмически, как и количество сравнений в АВЛ дереве, так как реструктуризация проводится путём увеличения размера хеш-таблицы в 1.6 раза, то есть количество вставок элементов между двумя реструктуризациями будет расти как степенная функция, из-за чего вставка в хеш-таблицу впоследствии будет осуществляться примерно с той же скоростью, что и вставка в АВЛ-дерево. В хеш-таблице с открытым хешированием всегда будет меньше количество реструктуризаций, но сами они сложнее из-за организации новых цепочек, а также из-за более долгого перехода по ссылкам, хеш-таблицы с открытым хешированием оказались даже быстрее при вставке, несмотря на большее количество сравнений.

6. Набор тестов.

Тесты описаны только для ввода значений. Отсутствие выходных данных означает в данном случае что тест позитивный.

В меню проверяется только то что после каждого действия запускается нужная функция.

Входные данные	**Выходные данные**	**Что и где проверялось**
0		Ввод минимального целого числа
3		Ввод целого числа не на границе
6		Ввод целого числа максимального размера
const		Ввод строки
	Ошибка. Пустой ввод.	Ввод пустой строки при вводе числа
	Ошибка. Не удалось считать строку.	Ввод пустой строки при вводе строки
what	Ошибка. Встречен неожиданный символ.	Ввод не числа, если нужно целое число
1234567890	Ошибка. Переполнение буфера при вводе.	Переполнение буфера (при чтении целого числа)
9	Ошибка. Число выходит за пределы допустимого диапазона.	Ввод числа, не входящего в допустимый диапазон чисел.

7. Выводы по проделанной работе.

В ходе выполнения работы я изучил операции с АВЛ-деревом и хеш-таблицами. Также я реализовал двоичное древо поиска, АВЛ-дерево и хеш-таблицы с открытым и закрытым хешированием и сравнил в них эффективность вставки и поиска элемента.

8. Ответы на вопросы.

8.1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

В АВЛ дереве менее жесткий критерий сбалансированности – для каждого узла дерева высота поддеревьев отличается не более чем на 1. В идеально сбалансированном, помимо этого, только нижний уровень может быть не полностью заполнен.

8.2. Чем отличается поиск в АВЛ дереве от поиска в двоичном дереве поиска?

Алгоритм поиска одинаковый (если не идёт вставка и удаление), отличается только сложность поиска в зависимости от того, как построилось дерево. В худшем случае поиск в обычном двоичном дереве поиска будет работать за O(n), а в АВЛ дереве поиск всегда будет приближаться к O(logn).

Если идёт вставка или удаление, то для АВЛ дерева на обратном пути проверяется разница в высотах поддеревьев, и если она больше 1, то придётся несколько раз переприсвоить ссылки, чтобы не нарушать критерий сбалансированности.

8.3. Что такое хеш-таблица, какой принцип её построения?

Хеш-таблица – массив, заполненный в порядке, определяемым хеш-функцией, которая ставит в соответствие каждому ключу индекс ячейки, где расположен элемент с этим ключом.

8.4. Что такое коллизии? Каковы методы их устранения?

Коллизии – ситуации, когда одному индексу ячейки соответствует несколько ключей. Первый метод решения – метод цепочек (открытое хеширование). Здесь к каждой ячейке таблицы присоединяется связный список, который хранит значения для каждого из ключей, добавленных в эту ячейку. Второй способ – открытая адресация (внутреннее хеширование). Здесь нет ссылок, но при этом идёт поиск по таблице другой подходящей ячейке. Ищут обычно либо с шагом 1, либо с шагом n^2, где n – номер попытки поиска.

8.5. В каком случае поиск в хеш-таблице становится неэффективен?

Поиск становится неэффективен, когда для него начинает требоваться более 3-4 сравнений.

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

Сложности поиска для деревьев указаны в ответе на 2 вопрос. Для таблиц поиск должен осуществляться за O(1). Для файлов сложность поиска зависит от того, как он составлен. Если это обычный текстовый файл с невыровненными данными и без каких-либо меток, позволяющих найти нужные данный, то сложность поиска будет O(n).