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
Лабораторная работа №7
«Сбалансированные деревья, хэш-таблицы»

Группа ИУ7-34Б

Дисциплина Типы и структуры данных

Вариант 3

Студент Писаренко Д.П.

Преподаватель Рыбкин Ю.С.

Москва 2022 г.

Оглавление

Описание условия задачи

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

Техническое задание

a. Описание исходных данных

Элементом любого контейнера является слово.

Правила ввода:

- Каждое слово с новой строки
- Пункты меню – целые числа.

b. Описание задачи, реализуемой программой

Реализована программа, позволяющая:

- Добавлять слова в любой контейнер из файла
- Удалять слова по полному совпадению в любом контейнере
- Удалить слова, начинающиеся на определенную букву в любом контейнере
- Вывести дерево в файл в DOT формате
- Сравнить время удаления слов в различных контейнерах и объем занимаемой памяти.

c. Способ обращения к программе

После запуска исполняемого файла app.exe, пользователю предлагается выбор опций из меню. Для выбора какого-то пункта меню необходимо ввести целое число после приглашения к вводу.

d. Описание возможных аварийных ситуаций и ошибок пользователя

- Вместо пункта меню вводится строка

- Введена пустая строка
- Слово уже содержится в контейнере (в данном случае это слово не будет добавлено повторно).

Описание внутренних СД

```
typedef struct tree_node tree_node_t;

struct tree_node
{
    tree_node_t *left;
    tree_node_t *right;
    char *val;
};
```

Листинг 1. Структура дерева

Значение полей структуры tree_node_t:

- left – левый ребенок
- right – правый ребенок
- val – строка

```
typedef struct avl_tree_node avl_tree_node_t;

struct avl_tree_node
{
    int height;
    avl_tree_node_t *left;
    avl_tree_node_t *right;
    char *val;
};
```

Листинг 2. Структура АВЛ-дерева

Значение полей структуры tree_node_t:

- left – левый ребенок
- right – правый ребенок
- val – строка
- height – высота поддерева

```
typedef enum { EMPTY, DELETED, FILLED } NodeStatus_t;

typedef struct {
    NodeStatus_t nodeStatus;
    char *key;
} node_arr_t;

typedef struct {
    size_t capacity;
    size_t size;
```

```
    size_t (*hasher)(const char *);
    node_arr_t *arr;
} hashtable_arr_t;
};
```

Листинг 3. Структура хеш-таблицы с открытой адресацией

Значение полей структуры node_arr_t:

- nodeStatus – текущее состояние узла (занята, свободна, удалена)
- key – строка

Значение полей структуры hashtable_arr_t:

- capacity – размер выделенной памяти для хеш-таблицы
- size – количество элементов, хранимых в хеш-таблице
- hasher – хеш-функция
- arr – выделенный под хеш-таблицу массив

```
typedef struct node_list node_list_t;

struct node_list
{
    char *key;
    node_list_t *next;
};

typedef struct
{
    node_list_t **buckets;
    size_t buckets_count;

    size_t buckets_sizes_list_idx;
    size_t items_count;

    size_t (*hasher)(const char *);
} hashtable_list_t;
```

Листинг 4. Структура хеш-таблицы с закрытой адресацией

Значение полей структуры node_list_t:

- next – следующий узел
- key – строка

Значение полей структуры hashtable_list_t:

- buckets – массив указателей на «цепочки»
- buckets_count – размер массива buckets
- buckets_sizes_list_idx – текущий индекс размера
- items_count – реальное количество элементов в хеш-таблице
- hasher – хеш-функция

Описание алгоритма

Алгоритм:

1. Пользователь вводит номер команды из меню.
2. Пока пользователь не введет 0 (выход из программы), ему будет предложено выполнять действия с меню.

Описание функций:

tree_node_t *tree_insert(tree_node_t *root, char *val) – добавляет строку в дерево, возвращает новый корень.

tree_node_t *find_min(tree_node_t *node) – находит минимальный элемент в дереве с корнем в node.

tree_node_t *find_max(tree_node_t *node) – находит максимальный элемент в дереве с корнем в node.

tree_node_t *remove_startswith(tree_node_t *root, char start) – удаляет все вершины дерева, элемент которых начинается с символа start, возвращает новый корень.

void tree_save_as_dot(FILE *f, tree_node_t *root) – сохраняет в файл f представление дерева в DOT-формате.

void tree_clear(tree_node_t *root) – освобождает выделенную под дерево память.

avl_tree_node_t *avl_tree_insert(avl_tree_node_t *root, char *val) - добавляет строку в авл-дерево, возвращает новый корень.

avl_tree_node_t *find_min(avl_tree_node_t *node) – находит минимальный элемент в авл-дереве с корнем в node.

avl_tree_node_t *find_maxavl_tree_node_t *node) – находит максимальный элемент в авл-дереве с корнем в node.

avl_tree_node_t *remove_startswith(avl_tree_node_t *root, char start) – удаляет все вершины авл-дерева, элемент которых начинается с символа start, возвращает новый корень.

void avl_tree_save_as_dot(FILE *f, avl_tree_node_t *root) – сохраняет в файл f представление авл-дерева в DOT-формате.

void avl_tree_clear(avl_tree_node_t *root) – освобождает выделенную под авл-дерево память.

hashtable_arr_t createHashTable_arr(size_t (*hasher)(const char *)) - создает хеш-таблицу с заданной хеш-функцией.

void deleteHashTable_arr(hashtable_arr_t *hashtable) — очищает выделенную под хеш-таблицу память.

int addHashTable_arr(hashtable_arr_t *hashtable, const char *) - добавляет в хеш-таблицу новую строку.

void removeStartswith_arr(hashtable_arr_t *hashtable, char target) — удаляет строки, начинающиеся на символ target из хеш-таблицы.

void printHashTable_arr(hashtable_arr_t *hashtable) — распечатывает хеш-таблицу на экран.

bool hasHashTable_arr(hashtable_arr_t *hashtable, const char *key) — проверяет, есть ли элемент в хеш-таблице

void remove_arr(hashtable_arr_t *hashtable, const char *key) — удаляет элемент key, если он есть в таблице

size_t gorner_hasher(const char *str, size_t coeff) — хеш-функция полиномиального хеширования hash(str) = ((str[0] * coeff) + str[1] * coeff) + … , где коэффициент хеш-функции

Реструктуризация хеш-таблицы

Реструктуризация хеш-таблицы с открытой адресацией происходит в том случае, если количество элементов в ней больше, чем ¾ от размера самой таблицы. Выбор такой константы обусловлена тем, что при такой максимальной средней длине цепочки амортизированная сложность сильно стремится к O(1). При этом размер хеш-таблицы увеличивается в два раза и для каждого элемента пересчитывается хеш-функция.

Реструктуризация хеш-таблицы с закрытой адресацией происходит в том случае, если средняя длина цепочки превышает 7. Выбор такой константы обусловлена тем, что при такой максимальной средней длине цепочки амортизированная сложность сильно стремится к O(1). При этом в качестве размера хеш-таблицы берется такое простое число, которое примерно в 2 раза больше, чем предыдущий размер таблицы.

Набор тестов

Негативные тесты:

Описание теста	Сообщение
Вводится буква вместо пункта меню	Введен неверный пункт меню
Файл для считывания данных не существует	Ошибка при попытке считывания из файла
Слишком большое количество слов в файле	Ошибка при попытке выделения памяти

Позитивные тесты

Описание теста	Действие	Сообщение
Ввод корректного файла при добавлении слов в файл	Ввод правильного файла	Слова успешно были добавлены
Удаление слов из дерева начинающихся на определнную букву	Ввода буквы	Удаление узлов прошло успешно
Удаление слов из хеш-таблицы, начинающихся на определнную букву	Ввод буквы	Удаление в хеш-таблице прошло успешно
Вывести дерево в файл DOT	Ввод названия файла «tree.dot»	Файл tree.dot успешно создан
Замерный эксперимент	-	Вывод таблицы, содержащей время выполнения Удаления слов в различных контейнерах
Выход	Выбор необходимого	Завершение

	пункта меню	программы

Оценка эффективности удаления

Удаляем по латинской букве «a»

Количество слов в файле	Удаление в АВЛ-дереве(такты)	Удаление в дереве (такты)
100	9360	13988
500	9178	21112
1000	14040	15054

Таблица 1. Удаление в сбалансированном дереве и в ДДП

АВЛ-дерево выигрывает по эффективности, так как более сбалансировано.

Эффективность удаления в правосторонних деревьях:

Количество узлов в дереве	Удаление в АВЛ-дереве(такты)	Удаление в правостороннем дереве (такты)
100	7722	5096
500	7566	5278
1000	14612	9672

Таблица 2. Удаление в правосторонних деревьях

В данном случае удаление несбалансированном дереве будет быстрее т. к. все удаляемые элементы лежат близко к корню.

Эффективность удаления в левосторонних деревьях:

Количество узлов в дереве	Удаление в АВЛ-дереве(такты)	Удаление в правостороннем дереве (такты)
100	6630	6240
500	11648	22230
1000	19682	35750

Таблица 3. Удаление в левосторонних деревьях

Как можно заметить, время удаления в левостороннем дереве сильно увеличилось, так как все удаляемые элементы лежат в конце «бамбука». В АВЛ-дереве результаты чуть хуже, чем на случайных данных, так как приходится много балансировать.

Эффективность удаления в хеш-таблице и в АВЛ-дереве:

Количество узлов в дереве	Удаление в АВЛ-дереве(такты)	Удаление в хеш-таблице с открытой адресацией	Удаление в хеш-таблице с закрытой адресацией
100	5720	4758	4004
500	11362	9438	9386
1000	17524	19838	15340

Оценка эффективности по памяти

Если сравнивать обычное дерево двоичного поиска, то оно эффективнее по памяти ровно на столько, сколько занимает памяти атрибут узла АВЛ-дерева, хранящий высоту текущего поддерева. В моем случае этот атрибут имеет тип int и занимает 4 байта.

Размер хеш-таблиц и в АВЛ-дерева:

Количество узлов в дереве	Размер АВЛ-дерева(байты)	Размер хеш-таблицы с открытой адресацией(байты)	Размер хеш-таблицы с закрытой адресацией(байты)
100	3200	2048	1704
500	16000	8192	7080
1000	32000	19838	11320

Наиболее эффективным контейнером по памяти в данном случае оказалась хеш-таблица с закрытой адресацией, благодаря тому, что памяти выделяется примерно столько же, сколько и требуется для хранения полезной информации

Вывод

В результате проделанной работы, мы выяснили, что операция удаления в двоичном дереве поиска выполняется очень быстро, как и в хеш-таблицах. В моем случае оказалось наиболее эффективным контейнером по времени и по памяти хеш-таблица с открытой адресацией. Так же важно отметить, что выбор подходящего контейнера зависит от типа хранимых данных.

Контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

Идеально сбалансированное дерево - дерево, у которого вес левого и правого поддерева отличается не более, чем на единицу.

АВЛ дерево - дерево, у которого высота левого и правого поддерева отличается не более, чем на единицу.

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

Сам процесс поиска ничем не отличается, но из-за того, что АВЛ-дерево балансирует высоту дерева, то и поиск происходит более стабильно.

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

Хеш-таблица - структура данных, которая позволяет по значению ключа сразу определять индекс элемента массива, в котором хранится информация.

Для этого необходимо создать такую функцию, по которой можно вычислить этот индекс. Такая функция называется хеш-функцией и она ставит в соответствие каждому ключу индекс ячейки

4. Что такое коллизии? Каковы методы их устранения.

Коллизия - ситуация, при которой два разных ключа дают одинаковое значение хэш-функции, т. е. одинаковый индекс.

Различают два основных метода разрешения коллизий: открытое и закрытое хеширование.

Открытое (метод цепочек) - каждому индексу соответствует связный список значений.

Закрытое (открытая адресация) - при возникновении коллизии элемент ставится выполняется поиск, пока не найдется данный элемент или свободная ячейка таблицы.

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

Поиск в хэш-таблицах становится неэффективен при достаточном количестве коллизий (3-4). Так как для поиска элемента требуется производить больше сравнений.

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

Эффективность поиска зависит от поставленной задачи, в каких-то случаях можно ограничиваться деревья в пользу других преимуществ, а в каких-то стоит отдать предпочтение хэш-таблицей.

Данная структура является наиболее эффективной для поиска.