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 «СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ, ХЕШ-ТАБЛИЦЫ»

Студент Равашдех Фадей Хешамович

Группа ИУ7 – 35Б

Преподаватель Никульшина Т. А.

2023

Оглавление

Описание условия задачи

Построить хеш-таблицу для слов текстового файла. Осуществить поиск указанного слова в двоичном дереве поиска (ДДП) и в хеш-таблице, если его нет, то добавить его (по желанию пользователя) в дерево и, соответственно, в таблицу. При необходимости использовать реструктуризацию таблицы. Сбалансировать дерево. Сравнить время поиска, объем памяти и количество сравнений при использовании ДДП, сбалансированных деревьев и хештаблиц. Сравнить эффективность добавления ключа в таблицу или ее реструктуризацию для различной степени заполненности таблицы.

Описание технического задания

Входные данные: Номер пункта меню, строка пути к файлу с данными, слова для их поиска/добавления/удаления в дереве/хеш-таблице.

Выходные данные: Считаное из файла двоичное дерево поиска, сбалансированное двоичное дерево поиска, хеш таблицы с открым и закрытым хешированием, результат поиска в дереве и хеш-таблицах, сравнение количества сравнений, времени поиска и занимаемой памяти между файлом и деревом.

Обращение к программе:

Запускается через терминал командой: ./app.exe.

Функции меню программы:

0. Информация о программе
1. Считать данные из файла
2. Вывести данные
3. Найти слово
4. Удалить слово
5. Сбалансировать дерево
6. Сравнение деревьев и хеш-таблиц
7. Сравнение хеш-таблиц
8. Выйти

Аварийные ситуации:

1. Ошибка выделения памяти;
2. Прекращение ввода (EOF);
3. Некорректный ввод;
4. Некорректный выбор действия меню программы;
5. Пустая строка;
6. Некорректный файл.

Описание структур данных

Основные структуры дерева

```
// Структура данных узла дерева: слово и его высота при балансировке
typedef struct
{
    char *key;
    int height;
} pair_t;
// Структура узла дерева
struct bintree
{
    pair_t data;
    bin_tree_t left;
    bin_tree_t right;
};
```

Структуры хеш-таблиц

```
//  Структура цепочки на списке для открытого хеширования
struct hash_list
{
    char *key;
    node_t *node;
};
// Структура хеш-таблицы открытого хеширования
struct hash_table_open
{
    node_t **arr;
    size_t n;
};
// Структура хеш-таблицы закрытого хеширования
struct hash_table_closed
{
    char **arr;
    size_t n;
};
```

Вспомогательные структуры для обхода по дереву

```
// Структура для замера времени поиска
struct measure_times
{
    double tree_timer;
    double file_timer;
    char *filename;
    bin_tree_t tree;
};
// Структура для списка для стека для вывода дерева
typedef struct list *list_t;
struct list
{
    char ch;
    list_t next;
};
```

Описание функций программы

Функции основного файла программы

```
// Контроллер программы
int controller(int option, bin_tree_t *tree, char **filename);
// Основная функция
int main(void);
```

Функции двоичного дерева поиска

```
// Перевод ДДТ в АВЛ-дерево
bt_rc_t bst_to_avl_tree(bin_tree_t *tree);
// Очищение дерева
void bin_tree_clear(bin_tree_t *tree);
// Вставка
bt_rc_t bin_tree_insert(bin_tree_t *tree, const char *key);
// Поиск
bt_rc_t bin_tree_find(const bin_tree_t tree, const char *key,
int **num, int *cmp_count);
// Удаление
bt_rc_t bin_tree_remove(bin_tree_t *tree, char *key);
// Обход (префиксный)
void bin_tree_each(const bin_tree_t tree, void (*action)(const
char *key, int *num, void *param), void *param);
```

Функции хеш-таблицы открытого хеширования

```
// Конструктор
hto_t hto_create(void);
// Деструктор
void hto_destroy(hto_t *table);
// Очистка
ht_rc_t hto_clear(hto_t table);
// Вставка
ht_rc_t hto_insert(hto_t *table, const char *key);
// Поиск
ht_rc_t hto_find(hto_t table, const char *key, int *cmp_count);
// Удаление
ht_rc_t hto_remove(hto_t table, char *key);
// Вывод
void hto_print(hto_t table);
// Реструктуризация
ht_rc_t hto_restruct(hto_t *table);
```

Функции хеш-таблицы закрытого хеширования

```
// Конструктор
htc_t htc_create(void);
// Деструктор
void htc_destroy(htc_t *table);
// Очистка
ht_rc_t htc_clear(htc_t table);
// Вставка
ht_rc_t htc_insert(htc_t *table, const char *key);
// Поиск
ht_rc_t htc_find(htc_t table, const char *key, int *cmp_count);
// Удаление
ht_rc_t htc_remove(htc_t table, char *key);
```

```
// Вывод
void htc_print(htc_t table);
// Реструктуризация
ht_rc_t htc_restruct(htc_t *table);
```

Функции-контроллеры

```
// Чтение из файла
int read(bin_tree_t *tree, hto_t *hto_table, htc_t *htc_table,
char **filename);
// Вывод
void print(bin_tree_t tree, hto_t hto_table, htc_t htc_table);
// Поиск (добавление)
int find(bin_tree_t *tree, hto_t *hto_table, htc_t *htc_table);
// Удаление
int remove_word(bin_tree_t *tree, hto_t hto_table, htc_t
htc_table);
// Балансировка
int balance(bin_tree_t *tree);
```

Функции сравнения

```
// Общее сравнение
int compare(char *filename);
// Сравнение времени
void measure_search(const char *key, int *value, void *n);
int compare_time(bin_tree_t tree, bin_tree_t avl, hto_t hto,
htc_t htc);
// Сравнение количества сравнений
void measure_cmps(const char *key, int *value, void *n);
int compare_cmps(bin_tree_t tree, bin_tree_t avl, hto_t hto,
htc_t htc);
// Сравнение памяти
void measure_mems(const char *key, int *value, void *n);
int compare_mem(bin_tree_t tree, bin_tree_t avl, hto_t hto,
htc_t htc)
```

Функции ввода/вывода

```
// Вывод меню программы
void print_menu(void);
// Вывод информации о программе
void print_info(void);
// Чтение строки неопределен. длины из стандартного потока ввода
int get_input(char **str, FILE *f);
// Чтение строки определенной длины из стандартного потока ввода
int get_n_input(char pch[], size_t n);
// Получение выбранного пользователем пункта
int get_option(size_t *option);
// Вывод сообщения, пояснящего ошибку
void print_error_msg(int rc);
// Вывод сообщения, пояснящего ошибку, и возврат того же кода
возврата
int print_error(int rc);
```

Описание алгоритма

1. Поиск в дереве двоичного поиска: сравнение искомого слова со словом данного узла. Если искомое слово равно, то слово найдено, если меньше, то вызывается функция поиска от левой вершины, иначе от правой. Если в итоге узел пуст, то слово не найдено.
2. Вставка элемента в ДДП: проход рекурсивно по дереву как при поиске, но если находит искомый узел, то увеличивает его чатоту на единицу, а если узел пуст, то создает элемент с переданным словом и вставляет его на место пустого.
3. Удаление элемента из ДДП : Если узел пуст, то возврат. Если узел не имеет потомков, то удаление узла. Если у узла один потомок, то узел удаляется, а на его место встает его потомок. Если у узла два потомка, то находится самый правый потомок у левого потомка и данные заменяются на найденные, а найденый потомок удаляется.
4. Обход дерева: Обход выполняется префиксно, сначала обрабатывается узел дерева с помощью переданной функции и переменной возврата, потом рекурсивный обход левого и правого потомков.
5. Балансировка: Создается АВЛ-дерево, в которое вставляются элементы из обычного ДДП, которые следят за своей высотой и балансируются при необходимости поворотами, если разница высот поддеревьев равна двум.
6. Хеш-функция для строк: берется первый символ, добавляется в сумму по модулю размера таблицы. Берется второй элемент, складывается с суммой, умноженной на константу (простое число), помещается в сумму по модулю размера таблицы. И далее следующие символы. Тогда итоговое значение хеш-функции будет равно: c1 * k^n + c2 * k^(n-1) + … + cn (mod N), где ci — i-ый символ строки, k — константа, n — количество символов в строке, N — размер таблицы.
7. Вставка в хеш-таблицу с открытым хешированием: вычисляется значение хеш-функции, в начало списка с соответствующим индексом вставляется новый узел, если не коллизия не превысит определенное значение.
8. Поиск в хеш-таблице с открытым хешированием: вычисляется значение хеш-функции, в списке с соответствующим индексом ищется ключ.
9. Удаление из хеш-таблицы с открытым хешированием: вычисляется значение хеш-функции, из списка с соответствующим индексом удаляется узел с соответствующим ключом.

10. Вставка в хеш-таблицу с закрытым хешированием: вычисляется значение хеш-функции, если значение по индексу пусто, то запись, если нет, то индекс увеличивается на единицу пока не найдется пустого места или коллизия не превысит определенное значение.

11. Поиск в хеш-таблице с закрытым хешированием: вычисляется значение хеш-функции, ищется ключ аналогично вставке.

12. Удаление из хеш-таблицы с закрытым хешированием: ищется ключ аналогично вставке, когда находит, заменяет его на DEL — значение ключа, который считается свободным для вставки, но занятым для поиска.

13. Реструктуризация хеш-таблицы происходит при превышении коллизией определенного значения: создается новый массив, больше старого, в который вставляются все элементы из старого.

НАБОР ТЕСТОВ

№	Название теста	Пользовательский ввод	Вывод
1	Некорректнрое действие	10	Некорректный номер пункта меню. Действия не существует. Ошибка: Действие с введенным пунктом отсутствует.
2	Некорректный ввод файла	1 \n	Ошибка: Некорректный ввод.
3	Работа с некорректным файлом	1 not_existing_file.txt 4	Ошибка: некорректный файл.
4	Некорректный ввод	1 data/10 \n	Ошибка: Некорректный ввод.
5	Корректный ввод и вывод дерева	1 data/10 2	Вывод хеш-таблицы с открытым хешированием: … Вывод хеш-таблицы с закрытым хешированием: …

Оценка среднего количества сравнений

Количество ключей	ДДП	Сбалансированное ДДП	Хеш-таблица с открытым хешированием	Хеш-таблица с закрытым хешированием
1	1.000000	1.000000	1.000000	1.000000
10	3.100000	2.900000	1.000000	1.000000
100	7.340000	5.810000	1.890000	1.000000
1000	13.560000	9.175000	1.889000	1.000000
10000	16.166900	12.440500	1.793300	1.000000

Сбалансированное ДДП делает меньше сравнений, чем обычное ДДП, т.к. средняя глубина сбалансированного ДДП ниже, чем средняя глубина обычного ДДП, и соответствует своей ассимптотике log2(N). Количество сравнений в хеш-таблице с открытым хешированием находится между 1 и (C+1)/2, где C – максимальная допустимая коллизия. При этом коллизии в хеш-таблице с закрытым хешированием, в отличие от хеш-таблицы с открытым хешированием, влияют на следующие записи, поэтому часто возникают ситуации, когда даже при неполной заполненности таблицы невозможна вставка записи без большой коллизии, поэтому происходит реструктуризация таблицы и среднее количество сравнений около единицы. Хеш-таблицы имеют среднее количество сравнений меньшее чем у ДДП.

Оценка эффективности поиска (секунды)

Количество ключей	ДДП	Сбалансированное ДДП	Хеш-таблица с открытым хешированием	Хеш-таблица с закрытым хешированием
1	0.000004	0.000003	0.000006	0.000011
10	0.000120	0.000110	0.000060	0.000117
100	0.001762	0.001470	0.000466	0.000882
1000	0.009760	0.006758	0.001311	0.002327
10000	0.104646	0.078042	0.014229	0.024264

Сбалансированное ДДП в следствие меньшего среденего количества сравнений работает быстрее, чем обычное ДДП. Хеш-таблица с открытым хешированием работает быстрее хеш-таблицы с закрытым хешированием примерно в 2 раза, т.к. сама таблица меньше в чуть менее, чем количество максимальной допустимой коллизии раз. Хеш-таблицы эффективнее деревьев в поиске, т.к. имеют меньшее количество сравнений.

Память (байт)

Количество ключей	ДДП	Сбалансированное ДДП	Хеш-таблица с открытым хешированием	Хеш-таблица с закрытым хешированием
1	26	30	106	90
10	260	300	268	108
100	2674	3074	2170	1146
1000	27298	31298	22330	12906
10000	279298	319298	232314	138282

Сбалансированное ДДП занимает больше места в памяти, чем обычное ДДП, т.к. состоит из узлов с добавленной высотой. Хеш-таблица с открытым хешированием занимает примерно в 2 раза больше места в памяти, чем хеш-таблица с закрытым хешированием, т.к. узлы списка занимают больше памяти чем элемент массива. Хеш-таблицы занимают меньше памяти, чем деревья, т.к. узел дерева больше узла списка, узел дерева > элемент массива.

Эффективность реструктуризации (секунды)

Заполненность хеш-таблицы с		Хеш-таблица с	
открытым хешированием	закрытым хешированием	открытым хешированием	закрытым хешированием
1/33	1/11	0.000012	0.000012
10/33	10/11	0.000021	0.000009
100/111	100/109	0.000053	0.000026
1000/1137	1000/1201	0.000542	0.000289
10000/12381	10000/12373	0.005741	0.000708

Видно, что при равном размере и равной заполненности реструктуризация хеш-таблицы с закрытым хешированием быстрее, чем реструктуризация хеш-таблицы с открытым хешированием. Одна и та же таблица при большей заполненности реструктурируется дольше.

Ответы на контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева? В идеально сбалансированном дереве кол-во элементов в правом и левом поддереве отличается не более чем на единицу. В АВЛ дереве высоты правого и левого поддерева отличается не более чем на единицу.

2. Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска? Алгоритм тот же, но в АВЛ-дереве поиск быстрее.

3. Что такое хеш-таблица, каков принцип ее построения? Структура данных позволяющая получать по ключу элемент массива называется хеш-таблицей. Для доступа по ключу используется хеш-функция, которая по ключу получает нужный индекс массива. Она должна возвращать равные индексы для одного ключа, использовать все индексы с одинаковой вероятностью и быстро считаться.

4. Что такое коллизии? Каковы методы их устранения. Ситуация, когда из разных ключей хеш-функция выдаёт один и тот же индекс, называется коллизией. Метод цепочек – при коллизии элемент добавляется в список элементов этого индекса. Линейная адресация – при коллизии ищется следующая незаполненная ячейка. Произвольная адресация - используется заранее сгенерированный список случайных чисел для получения последовательности. Двойное хеширование – вычитать из значения первой значение второй при необходимости.

5. В каком случае поиск в хеш-таблицах становится неэффективен? При большом количестве коллизий.

6. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах Скорость поиска в хеш-таблице зависит от числа коллизий. При небольшом числе коллизий для поиска элемента совершается мало сравнений и поиск быстрее чем в деревьях. АВЛ-дерево быстрее при поиске за счёт более равномерного распределения элементов чем в ДДП.

Вывод

В ходе лабораторной работы я научился реализовывать двоичное дерево поиска, АВЛ-дерево и хеш-таблицы с различным хешированием.

АВЛ-деревья более сбалансированы, чем обычные деревья двоичного поиска, поэтому их среднее коилчество сравнений меньше, поэтому и скорость поиска выше. Хеш-таблица с открытым хешированием примерно в 2 раза быстрее, чем хеш-таблица с закрытым хешированием, но при этом занимает в 2 раза больше памяти. При этом коллизия больше в хеш-таблице с открытым хешированием, что и обеспечивает меньший размер таблицы-массива. Реструктуризация быстрее у хеш-таблицы с закрытым хешированием, т.к перенос элементов в новый массив быстрее, чем перенос элементов в массив списков.