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 № 6

«Деревья. Хеш-таблицы.»

Студент Маслова Марина Дмитриевна
фамилия, имя, отчество

Группа ИУ7-33Б

2020 г.

Оглавление

Техническое задание

Условие задачи

Построить ДДП, в вершинах которого находятся слова из текстового файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Добавить указанное слово, если его нет в дереве (по желанию пользователя) в исходное и сбалансированное дерево. Сравнить время добавления и объем памяти. Построить хеш-таблицу из слов текстового файла, задав размерность таблицы с экрана, используя метод цепочек для устранения коллизий. Вывести построенную таблицу слов на экран. Осуществить добавление введенного слова, вывести таблицу. Сравнить время добавления, объем памяти и количество сравнений при использовании ДДП, сбалансированных деревьев, хеш-таблиц и файла.

Входные данные

Имя файла в качестве аргумента при запуске.

Целое положительное число, задающее размер хеш-таблицы.

Строка, задающее слово для добавления в структуры.

Выходные данные

Построенные ДДП и сбалансированное дерево, полученная хеш-таблица, время добавления, количество сравнений и используемая память при добавлении в каждую из структур данных.

Задачи, реализуемые программой

1. Построение ДДП слов по данным из файла.
2. Балансировка дерева.
3. Построение хеш-таблицы.
4. Реструктуризация хеш-таблицы.
5. Добавление слова в каждую из структур.

Возможные аварийные ситуации и ошибки пользователя

Неверно заданные аргументы при вызове приложения.

Несуществующий файл.

Неверно заданный размер таблицы.

Неверно заданный параметр выбора.

Ошибка выделения памяти.

Описание внутренних структур данных

Для хранения вершин дерева используется структура:

```
typedef struct node node_t;
struct node
{
    int height;      //высота дерева с корнем в данной вершине
    node_t *left;    //указатель на левого потомка
    node_t *right;   //указатель на правого потомка
    char *value;     //слово в данной вершине
};
```

Для хранения элементов хеш-таблицы используется структура:

```
typedef struct word_list word_list_t;
struct word_list
{
    char *word;         //слово
    word_list_t *next; //указатель на следующий элемент
};
```

Для хранения самой хеш-таблицы:

```
typedef struct hash_table
{
    word_list_t **arr;//массив указателей на списки элементов
    int table_size;   //размер таблицы
} hash_table_t;
```

Описание функций

```
/**
* Создает ДДП result, вершинами которого являются слова
* из файла file
*/
int create_binary_tree(FILE *file, node_t **result);

/**
```

```
	* Инициализирует вершину дерева node с высотой height,
	* левым потомком left, правым потомком right и словом word
	*/
	void node_init(node_t *node, const int height, node_t *left, node_t
*right, char *word);

	/**
	* Добавляет вершину со словом word в ДДП result
	*/
	int add_node_to_bintree(node_t **result, char *word);

	/**
	* Печатает дерево с корнем root с положением от края экрана в
	* place пробелов
	*/
	void print_tree(node_t *root, int place);

	/**
	* Создает АВЛ-дерево balanced_tree, по ДДП tree
	*/
	int create_balanced_tree(node_t *tree, node_t **balanced_tree);

	/**
	* Добавляет вершину со словом word в АВЛ-дерево tree
	*/
	int add_node_to_balansed_tree(node_t **tree, char *word);

	/**
	* Инициализирует хеш-таблицу table с размером table_size
	*/
	int hash_table_init(hash_table_t *table, const int table_size);

	/**
	* Вычисляет хеш-функцию по заданному слову word и размеру
	* таблицы table_size
	*/
	int hash_functon(const char *const word, const int table_size);

	/**
	* Создает хеш-таблицу result со словами из файла file
	*/
	int create_hash_table(FILE *file, hash_table_t *result);
```

```
/**
* Печатает хеш-таблицу table
*/
void print_table(hash_table_t *table);

/**
* Добавляет слово word в хеш-таблицу result
*/
int add_word_to_hash_table(hash_table_t *result, char *word);
```

Описание алгоритма

При добавлении элемента в ДДП используется рекурсия: на каждой вершине, начиная с корня, выбирается правая или левая ветвь в зависимости от того больше или меньше добавляемое слово слова, находящегося в текущей вершине. Если ветвь не существует, то создается новая вершина с заданным словом, а также ветвь к этой вершине.

При добавлении элемента в сбалансированное дерево используется аналогичный алгоритм, только после добавления происходит последовательное балансирование выше лежащих узлов, а следовательно, и дерева в целом.

При добавлении элемента в хеш-таблицу вычисляется хеш-функция слова по формуле $f = s_0 + s_1 * P + s_2 * P^2 + \ldots + s_i * P^i + \ldots$, где s_i– i-ый символ текущего слова, P– простое число (в реализации используется $P = 61$), далее находится остаток от деления найденного значения на размер таблицы, и слово добавляется в массив под соответствующим индексом, если значения функций для двух различных слов совпадают, используется метод цепочек.

Балансировка дерева происходит посредством балансировки поддеревьв дерева, начиная с листов.

Формирование деревьев и таблицы реализуется с помощью последовательного добавления слов в дерево или таблицу.

Реструктуризация хеш-таблицы производится с помощью увеличения её размера до ближайшего простого числа до тех пор, пока не будет достигнуто необходимое количество сравнений.

Тесты

№	Что проверяется	Входные данные	Результат
1	Неверное количество аргументов	./app.exe	Неверное количество аргументов
2	Неверное имя файла	test.txt	Не удалось открыть файл
3	Неверный ввод размера хеш-таблицы (не число)	jkjkj	Неверный размер! Повторите попытку!
4	Неверный ввод размера хеш-таблицы (отрицательное число)	-1	Неверный размер! Повторите попытку!
5	Неверный ввод размера хеш-таблицы (отрицательное число)	0	Неверный размер! Повторите попытку!
6	Неверный выбор (не число)	jk	Несоответствующие данные! Повторите попытку!
7	Неверный выбор (больше 1)	10	Несоответствующие данные! Повторите попытку!
8	Неверный выбор (меньше 0)	-10	Несоответствующие данные! Повторите попытку!

Оценка эффективности

Добавление элементов (тики):

Количество элементов	ДДП	АВЛ	Хеш-таблица (без коллизий)	Хеш-таблица (1 / четверть от количества)**	Файл
5	3241	10073	2307 (5)	2607 (2) / 2802 (1)	13344
50	4733	53027	2013 (137)	3233 (13) / 7642 (2)	55352
500	7439	128712	1561 (2137)	2373 (139) / 16423 (2)	103656

* для хеш-таблицы в скобках указан её размер

** количество сравнений

Среднее количество сравнений:

Количество элементов	ДДП	Сбалансированное	Хеш-таблица	Файл
5	2	2	1	5
50	5	5	1	50
500	11	8	1	500

Используемая память (байт):

Количество элементов	ДДП	Сбалансированное	Хеш-таблица (без коллизий)	Файл
5	228	228	268	35
50	1983	1983	2285	305
500	19533	19533	28645	3005

Контрольные вопросы

Что такое дерево?

Дерево – это нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».

Как выделяется память под представление деревьев?

Память выделяется под каждую вершину дерева с учетом хранения в нем необходимых данных и указателей на потомков.

Какие стандартные операции возможны над деревьями?

Обход дерева, поиск элемента, добавление элемента, удаление элемента.

Что такое дерево двоичного поиска?

Дерево двоичного поиска – это такое дерево, в котором у каждого узла два потомка и все левые потомки моложе предка, а все правые – старше.

Чем отличается идеально сбалансированное дерево от АВЛ дерева?

В идеально сбалансированном дереве количество вершин в каждом поддереве различается не больше, чем на 1, в АВЛ же дереве для каждой его вершины высота ее двух поддеревьев различается не более, чем на 1.

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

В АВЛ-дереве поиск работает быстрее за счет меньшей высоты поддеревьев, а следовательно и меньшего количества сравнений.

Что такое хеш-таблица, каков принцип ее построения?

Хеш-таблицей называется массив, заполняемый в порядке, определяемым функцией, которая каждому элементу ставит в соответствие его индекс в этом массиве. Принцип построения скрыт в её определении: для каждого добавляемого элемента вычисляется его индекс, по которому элемент и добавляется в массив.

Что такое коллизии? Каковы методы их устранения.

Коллизии – это ситуации, когда хеш-функция различным элементам ставит в соответствие одинаковые индексы в хеш-таблице. Устраняются они

методом цепочек (открытое хеширование) или открытой регистрацией (закрытое хеширование)

В каком случае поиск в хеш-таблицах становится неэффективен?

При большой количестве коллизий увеличивается количество сравнений, что сказывается на временной эффективности поиска в хеш-таблице.

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

Эффективности в худшем случае.

ДДП – $O(h)$, где h – высота дерева.

АВЛ –$O(\log_2 n)$

Хеш-таблица – без коллизий $O(1)$, с коллизиями в худшем случае $O(n)$.

Вывод

Для добавления элементов самым эффективным по времени является хеш-таблица. Медленнее работает добавление элемента в несбалансированное дерево двоичного поиска (чем больше количество элементов, тем больше разрыв по времени от хеш-таблицы), добавление элемента в сбалансированное дерево в 5-10 раз медленнее, чем при добавлении в обычное ДДП, несмотря на то, что место добавления находится за меньшее число сравнений, затрачивается время на то, чтобы сбалансировать дерево. Добавление уникального элемента в файл в 10-100 раз больше времени по сравнению с другими структурами, так как происходит сравнение вставляемой строки с каждой строкой в файле, что определяет большее количество сравнений. Однако файл более эффективен по памяти, требует ее в 4 раза меньше, чем другие структуры. Память используемая хеш-таблицей в данной реализации зависит от её размера, чем меньше таблица, тем меньше памяти она использует, однако тем больше коллизий возникает, тем больше сравнений происходит при добавлении элемента, тем медленнее происходит процесс. При этом допущение малого количества сравнений в хеш-таблице (1-2) в значительной мере (в 2-15 раз в зависимости от количества элементов) уменьшает необходимую память, практически не изменяя время обработки.

Таким образом, при недостатке памяти алгоритм добавления уникального элемента в некоторый «список» эффективнее реализовывать в файле, при желании ускорить обработку данных при небольших длинах «списка» необходимо использовать деревья двоичного поиска, при обработке же большого количества данных (500 и более элементов) правильным будет использовать хеш-таблицу с размером, который в необходимом количестве минимизирует коллизии, что требует большего количества памяти, но в значительной мере ускоряет процесс обработки.