Оберган Татьяна ИУ7-35Б

Вариант 18

**Лабораторная работа №6**

Деревья, хеш-таблицы

**Цель работы:** построить дерево, вывести его на экран в виде дерева, реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов, сбалансировать дерево, сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления; построить хеш-таблицу и вывести ее на экран, устранить коллизии, если они достигли указанного предела, выбрав другую хеш-функцию и реструктуризировав таблицу; сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска (ДДП) и в хеш-таблицах. Сравнить эффективность реструктуризации таблицы для устранения коллизий с эффективностью поиска в исходной таблице.

**Условие задачи (вар 3):**

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. Удалить указанное слово в исходном и сбалансированном дереве. Сравнить время удаления 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 удаление введенного слова, вывести таблицу. Сравнить время удаления, объем памяти и количество сравнений при использовании ДДП, сбалансированных деревьев, хеш-таблиц и файла.

**Входные данные:**Текстовый файл с данными. Минимальный размер хеш-таблицы.

**Выходные данные:**Вывод в консоль обычного дерева/сбалансированного дерева, созданную хеш-таблицу.

**Функция программы:**Загрузка обычного/сбалансированного дерева, хеш-таблицы из файла. Удаление элементов их этих структур. При удалении элемента из сбалансированного дерева происходит его автоматическая балансировка.

**Обращение к программе** осуществляется через консоль.

**Аварийные ситуации:**Отсутствие файлов с данными, ошибки выделения памяти.

**Структуры данных:**

Узел дерева:**struct** tree\_node  
{  
 **char** \*str;  
 **int** height;  
 **struct** tree\_node \*left;  
 **struct** tree\_node \*right;  
};

Узел хеш-таблицы:

**struct** table\_node  
{  
 **char** \*word;  
 **struct** table\_node \*next;  
};

**Функции:**

Функции работы с деревьями:

**struct** tree\_node\* create\_node(**char** \*word);  
**void** count\_nodes(node\_t \*tree, **int** \*sum);

**int** height(node\_t\* p);  
**void** go\_through(**struct** tree\_node \*tree, **void** (\*f)(**struct** tree\_node\*, **void**\*, **int** flag), **void** \*arg, **int** flag);

**void** print\_tree(**struct** tree\_node \*tree, **int** flag);  
**void** export\_to\_dot(FILE \*f, **const char** \*tree\_name, **struct** tree\_node \*tree, **int** flag);  
  
node\_t\* create\_tree(node\_t \*root, FILE \*f);  
  
**void** check\_height(node\_t\* p);  
node\_t\* rotateright(node\_t\* p);  
node\_t\* rotateleft(node\_t\* q);  
node\_t\* balance(node\_t\* p);  
node\_t\* create\_balanced\_tree(node\_t \*root, FILE \*f);  
**struct** tree\_node\* insert(**struct** tree\_node \*tree, **struct** tree\_node \*node);

**struct** tree\_node\* balanced\_insert(**struct** tree\_node \*root, **struct** tree\_node \*node);  
node\_t \*remove\_in\_tree(**char** \*rem, node\_t \*tree, **int** \*done, **int** \*flag);  
node\_t \*remove\_in\_balanced\_tree(**char** \*rem, node\_t \*tree, **int** \*done, **int** \*flag);

**void** check\_balance(node\_t \*root);

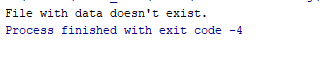
Функции работы с хеш-таблицей:

**int** next\_prime\_number(**int** number);  
**int** check\_tbl(**int** \*tbl, **int** len, **int** max);  
**int** count\_min\_len(**char** file\_name[], **int** stop\_len\_list);  
**int** len\_list(node\_hash \*head);  
**int** next\_prime\_number(**int** number);  
**int** key(**char** \*word, **int** tbl\_len);  
**void** insert\_node(node\_hash \*\*head, node\_hash \*node);  
node\_hash\* create\_hash\_node(**char** \*word);  
node\_hash \*\*create\_hash\_table(**char** file\_name[], **int** \*len, **int** stop\_len\_list);  
**void** remove\_from\_hash\_table(**char** \*del\_word, node\_hash \*\*table, **int** len\_table, **int** \*done);  
  
**void** print\_table(node\_hash \*\*table, **int** len\_table);

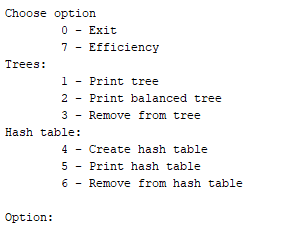
Функции проверки эффективности:  
**void** efficiency(**char** data\_file[], **int** len\_table, **int** stop\_len\_list);  
**unsigned long int** tick(**void**);

**Интерфейс:**

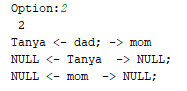
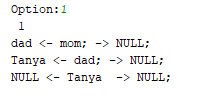
Проверка на отсутствие файла:



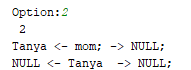
Меню:



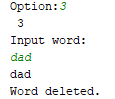
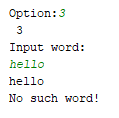
Вывод обычного и сбалансированного дерева:



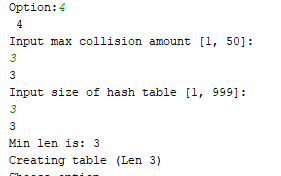
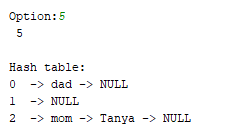
Удаляем слова – dad, mom, Tanya

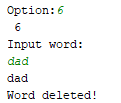
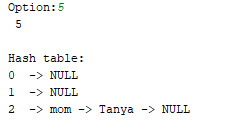
  

Само удаление:

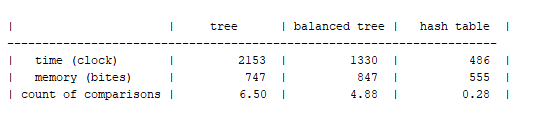
Хеш-таблица:

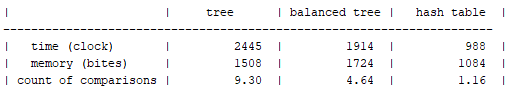
 

**Анализ эффективности (по памяти и времени):**

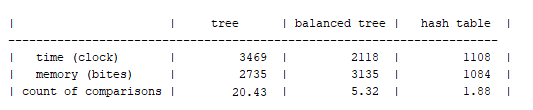
25 элементов. Хеш-таблица длины 30.



50 элементов. Хеш-таблица длины 41.

****

100 элементов в алфавитном порядке. Хеш-таблица длины 41.



**Контрольные вопросы:**

**1.Что такое дерево?**

Дерево – нелинейная структура данных, которая используется для представления иерархических связей «один ко многим». Дерево с базовым типом Т определяется рекурсивно: это либо пустая структура (пустое дерево), либо узел типа Т с конечным числом древовидных структур того же типа – поддеревьев.

**2. Как выделяется память под представление деревьев?**

Выделение памяти под деревья определяется типом их представления. Это может быть таблица связей с предками (№ вершины - № родителя), или связный список сыновей. Оба представления можно реализовать как с помощью матрицы, так и с помощью списков. При динамическом представлении деревьев (когда элементы можно удалять и добавлять) целесообразнее использовать списки – т.е. выделять память под каждый элемент динамически.

**3. Какие стандартные операции возможны над деревьями?**

Обход, поиск, добавление и удаление элемента.

**4. Что такое дерево двоичного поиска?**

Дерево двоичного поиска – дерево, в котором все левые потомки «моложе» предка, а все правые – «старше». Это свойство выполняется для любого узла, включая корень.

**5. Чем отличается идеально сбалансированное дерево от АВЛ дерева?**

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Идеально сбалансированное: для каждой его вершины количество вершин в левом и правом поддереве различаются не более чем на 1.

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

Алгоритм поиска одинаковый.

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

Массив, заполненный в порядке, определенным хеш-функцией, называется хеш-таблицей.

**8. Что такое коллизии? Каковы методы их устранения.**

Может возникнуть ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда h(K1) = h(K2), в то время как K1 ≠ K2. Такая ситуация называется коллизией. Для устранения коллизий нужно выбрать другую хеш функцию (или эту же с другими параметрами) и перегенерировать таблицу.

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

При большом количестве коллизий.

**Вывод:**

Самой оптимальной структурой по времени для поиска данных является хеш-таблица. Подобранные хорошо изначально параметры для хеш-таблицы позволяет уменьшить временные затраты на ее перегенерацию и в дальнейшем уменьшить среднее количество сравнений при поиске. Поиск в сбалансированном дереве быстрее, чем в обычном.