УДК 004.021

Романова Полина Сергеевна
Romanova Polina Sergeevna
Студент
Student
Никульшина Татьяна Александровна
Nikulshina Tatiana Alexandrovna
Старший преподаватель
Senior lecturer
Погорелов Дмитрий Александрович
Pogorelov Dmitriy Alexandrovich
Ассистент
Assistant
Московский государственный технический университет имени Н.Э. Баумана
Bauman Moscow State Technical University

СРАВНЕНИЕ ВРЕМЕНИ ВЫПОЛНЕНИЯ, ОБЪЁМА ПАМЯТИ И КОЛИЧЕСТВА СРАВНЕНИЙ ПРИ УДАЛЕНИИ СЛОВА В ДВОИЧНОМ ДЕРЕВЕ ПОИСКА, СБАЛАНСИРОВАННОМ ДЕРЕВЕ, ХЕШ-ТАБЛИЦЕ И ФАЙЛЕ

COMPARISON OF EXECUTION TIME, MEMORY SIZE, AND NUMBER OF COMPARISONS WHEN DELETING A WORD IN A BINARY SEARCH TREE, BALANCED TREE, HASH TABLE, AND FILE

Аннотация: Рассмотрено построение двоичного дерева поиска (ДДП), в вершинах которого находятся слова из текстового файла, произведена балансировка полученного дерева. Произведено сравнение времени удаления указанного слова и объём выделенной памяти. Аналогичная работа проведена с хеш-таблицей и файлом.

Abstract: The construction of a binary search tree (BST) with words from a text file at its vertices is considered, and the resulting tree is balanced. The time of deleting the specified word and the amount of allocated memory are compared. Similar work was done with the hash table and the file.

Ключевые слова: удаление файла, двоичное дерево поиска, хеш-таблицы, сбалансированный деревья.

Keywords: file deletion, binary search tree, hash tables, balanced trees.

При работе с предложениями и текстами, составленными из этих предложений, необходимо иметь возможность не только добавлять слова и изменять их, а также удалять, что является не менее важной задачей.

При организации работы с предложениями и текстами, можно работать сразу со всем текстом, как с файлом, а можно первоначально разбить текст на слова, а далее построить:

двоичное дерево поиска (ДДП);

сбалансированное дерево;

хеш-таблицу.

Рассмотрим организацию каждой структуры и оценим время удаления и объём памяти под каждый вариант.

Двоичное дерево поиска

Структура узла дерева состоит из указателя на данные, хранимые в данном узле, указателя на левый потомок (такой же узел) и указателя на правый потомок. При этом слева находится элемент, меньший данного, а справа – больший (рис. 1).

```
typedef struct Node {
    elem_t *data;
    struct Node *left;
    struct Node *right;
} tree_node_t;
```

Рис. 1. Структура узла двоичного дерева поиска на языке С

У данной структуры есть следующие особенности:

вид дерева зависит от порядка элементов, в котором они добавлялись (при добавлении отсортированных данных дерево выродится в односвязный список);

при добавлении элемента каждый раз запрашивается память под узел, а значит данные располагаются в произвольном порядке;

при удалении элемента память под узлом освобождается.

Сбалансированное двоичное дерево (АВЛ-дерево)

Структура узла дерева состоит из указателя на данные, хранимые в данном узле, указателя на левый потомок (такой же узел), указателя на правый потомок и высоту узла (листья имеют высоту 0, их непосредственные родители – 1 и т.д.). При этом слева находится элемент, меньший данного, а справа – больший. Кроме того, выполняется условие балансировки: для каждой вершины (узла) высота её двух поддеревьев различается не более чем на 1 (рис. 2).

```
typedef struct BNode {
    elem_t *data;
    struct BNode *left;
    struct BNode *right;
    short height;
} balanced_tree_node_t;
```

Рис.2. Структура узла сбалансированного двоичного дерева на языке С

У данной структуры есть следующие особенности:

за счёт балансировки дерево не вырождается в односвязный список в случае отсортированных данных, его высота пропорциональна логарифму;

при добавлении элемента каждый раз запрашивается память под узел, а значит данные располагаются в произвольном порядке;

при удалении элемента память под узлом освобождается.

Хеш-таблица

Описание хеш-таблицы на языке С приведено на рисунке 3.

```
typedef struct hash_table hash_table_t;
typedef size_t (* hash_func_type_t)(elem_t *elem, hash_table_t
*table);
struct hash_table {
    list_t *data;
    size_t size;
    double mul_const;
    hash_func_type_t hash_func_type;
    double av_conflicts;
    xor_rands_t xor_rands;
};
```

Рис.3. Структура хеш-таблицы на языке С

Вспомогательной структурой данных является односвязный список. Описание узла на языке С (рис. 4).

```
typedef struct
{
    node_t *head;
    size_t count;
} list_t;
typedef struct node node_t;
struct node
{
    void *data;
    node_t *next;
};
```

Рис. 4. Структура вспомогательного односвязного списка

Структура хеш-таблицы состоит из ее размера, массива указателей на головы односвязных списков и данных, необходимых для определения хеш-функции. Тип хеширования, который реализуется данной хеш-таблицей – открытый (устранение коллизий с помощью метода цепочек: если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором).

Описание алгоритма для дерева

Идея удаления следующая: находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.

Рекурсивный алгоритм описан ниже.

Если на текущем шаге указатель на узел дерева пуст, то мы пришли в потомок листа, а значит надо вернуть нулевой указатель.

Если указатель на узел дерева не пуст, то сравним значение в этом узле с переданными данными: если значение узла больше, то продолжим поиск элемента для удаления в левом поддереве, иначе в правом. Выйдя из рекурсии, вернём указатель на текущий узел дерева (с изменённым левым или правым потомком). При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.

Если значение узла равно переданным данным, то мы нашли элемент, который необходимо удалить. Сохраним указатели на потомков и удалим данный узел.

Если правого потомка не существует, то вернём указатель на левого потомка (для ДДП это очевидный шаг, а для АВЛ это справедливо благодаря его свойству: поскольку правый потомок отсутствует, то левый потомок либо вообще не существует, либо является листом).

Если же правый существует, то найдём минимум в правом поддереве (очевидно нужно двигаться по левым потомкам). Извлекаем минимум и присваиваем его правому потомку указатель на правого потомка исходного узла, который может быть получен после извлечения минимума (для АВЛ в процессе получения надо также выполнять условие балансировки). Левому потомку минимума присваиваем указатель на левый потомок исходного узла. Возвращаем минимум. При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.

Возврат текущего узла дерева (с изменённым левым или правым потомком). При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.

Балансировка (для АВЛ)

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины. Используются 4 типа вращений.

Малое левое вращение (рис. 5). Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева <= высота R.

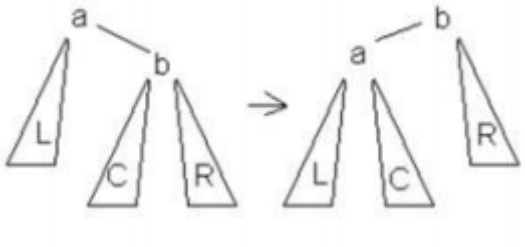

Рис. 5. Малое левое вращение

Большое левое вращение (рис. 6).

Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева > высота R.

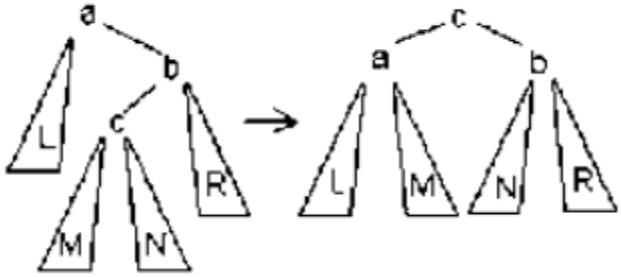

Рис. 6. Большое левое вращение

Малое правое вращение (рис. 7).

Данное вращение используется тогда, когда (высота b-поддерева — высота R) = 2 и высота C <= высота L.

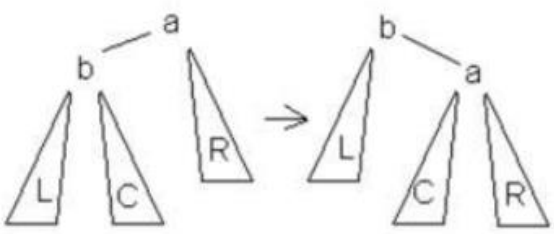

Рис. 7. Малое правое вращение

Большое правое вращение (рис. 8). Данное вращение используется тогда, когда (высота b-поддерева — высота R) = 2 и высота c-поддерева > высота L.

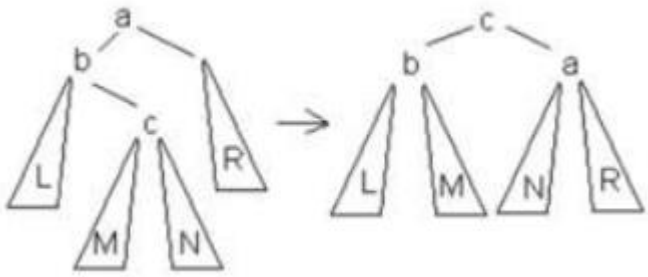

Рис. 8. Большое правое вращение

Балансировка реализуется с помощью переприсвоения указателей.

Оценка по времени

АВЛ-дерево

Г. М. Адельсон-Вельский и Е. М. Ландис доказали теорему, согласно которой высота АВЛ-дерева с N внутренними вершинами заключена между log2(N+1) и 1.4404*log2(N+2)-0.328, то есть высота АВЛ-дерева никогда не превысит высоту идеально сбалансированного дерева более, чем на 45 %. Для больших N имеет место оценка 1.04*log2(N). Высота дерева влияет на количество сравнений при поиске (удалении, вставке), а значит и на время выполнения.

ДДП

Эта структура данных имеет оценку от O(log2N) до O(n). Оценка зависит от порядка, в котором поступали элементы при добавлении: если поступили отсортированные данные, то дерево превращается в лево/правостороннее дерево, то есть односвязный список, в котором поиск (удаление/добавление) осуществляется в худшем случае за O(n).

Оценка по памяти

Поскольку выбрана реализация дерева на списочной структуре, то объём памяти будет пропорционален количеству элементов (узлов) в дереве. Каждый узел сбалансированного дерева в отличие от обычного содержит дополнительно

высоту узла, а значит АВЛ-дерево при такой реализации всегда проиграет несбалансированному.

Хеш-таблица

Оценка удаления по времени

Поскольку для устранения коллизий используется метод цепочек (внешний тип хеширования), то вставка элементов сводится к вычислению значения хеш-функции и поиске в односвязном списке. При хорошо подобранной хеш-функции удаление элементов по сложности будет близко к O(1) (на самом деле сложность зависит от количества элементов в списке, соответствующем индексу, который был вычислен для удаляемого элемента и, как следствие, от положения удаляемого элемента в списке).

Оценка по памяти

В случае хеш-таблицы (при статическом методе хеширования) объём будет определяться таким образом: (размер узла списка) * (количество добавленных элементов) + (размер указателя на узел в списке) * (размер хеш-таблицы) + (размер структуры таблицы - незначительно)

Экспериментальная проверка сделанных выводов

Количество элементов в файле = 500. Результаты представлены на рисунке 9.

data struct	time	memory	comparisons
file	941.00	0	500.00
hash table	0.29	16088	1.76
binary tree (BT)	0.37	9960	9.59
balanced BT	0.58	13280	7.81

Рис. 9. Результаты эксперимента

Время при работе с файлом очевидным образом превысило все остальные, поскольку происходит обращение к внешнему устройству. Хеш-таблица (имеющая размер 500) показала самый лучший результат по времени и по сравнениям, что тоже ожидаемо: выбранная хеш-функция оказалась достаточно хорошей, чтобы в среднем доступ к элементам осуществляется за примерно O(1). Сбалансированное дерево выиграло по сравнениям у обычного, но проиграло по памяти и по времени: по памяти из-за дополнительного поля (высоты) в структуре каждого узла дерева , по времени из-за выполнения функции балансировки и других вспомогательных рекурсивных функций.

Теперь посмотрим, что произойдет, если поступят отсортированные данные (рис. 10). Количество элементов = 500, размер хеш-таблицы = 500.

data struct	time	memory	comparisons
file	796.00	0	500.00
hash table	0.34	16088	6.32
binary tree (BT)	2.85	12000	250.50
balanced BT	0.55	16000	8.00

Рис. 10. Результаты эксперимента с отсортированными данными

Можем заметить, что ДДП резко проиграло по времени в среднем сбалансированному дереву (в 5 раз) и хеш-таблице (в 8.4 раз). А хеш-таблица и сбалансированное дерево вне зависимости от данных сохранили результаты по времени и памяти.

Исходя из полученных экспериментальных данных и теоретических оценок, можно сделать вывод, что в нашей задаче (удаления элементов) целесообразно использовать хеш-таблицу. Эта структура данных, в отличие от деревьев, имеющих оценку в O(log2N) в лучшем случае, имеет оценку, близкую к O(1). Однако здесь следует заметить, что при большом количестве коллизий

хеш-таблица станет менее эффективной. В этом случае необходимо произвести реструктуризацию хеш-таблицы путём увеличения размера или выбора другой хеш-функции. Кроме того, если сравнивать ДДП и сбалансированное дерево, то использование сбалансированного дерева является более оптимальным, поскольку его эффективность не зависит от входных данных в отличие от ДДП. Если же сравнивать ДДП, АВЛ, хеш-таблицу и файл, то мы видим, что файл сильно проигрывает другим структурам данных по времени за счет того, что происходит обращение к внешнему устройству, однако это позволяет практически не ограничиваться по памяти (так как размер ОП, отведённый программе сильно меньше, чем количество памяти на внешнем устройстве).

Библиографический список:

1. Вирт Н. Алгоритмы и структуры данных: Пер. с англ.- СПб.: Невский диалект, 2001. С. 69–71, 205–235

2. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы.: Пер. с англ. М.: Издат. дом «Вильямс», 2000. С. 58–76