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

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6
Деревья, хеш –таблицы

Название предмета: Типы и структуры данных

Студент: Варламова Екатерина Алексеевна

Группа: ИУ7-31Б

2020г.

I. Описание условия 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. 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.

Техническое задание

1. Описание исходных данных

Программа ожидает:

- название файла со словами (в аргументах командной строки)
- номер действия (подробнее о том, какой номер соответствует определённому действию, будет выведено при запуске программы).
- слово (по запросу удалить его в указанную структуру данных)
- слово (по запросу добавить его в указанную структуру данных)
- размер хеш-таблицы и максимальное количество коллизий
- хеш-функция (по запросу реструктуризации)

2. Описание результата программы

Результатом работы программы могут являться (в зависимости от введённого действия):

1. Двоичное дерево (его графическая интерпретация и список дуг)
2. Сбалансированное двоичное дерево (его графическая интерпретация и список дуг)
3. Хеш-таблица
4. Вывод содержимого файла

5. Статистика по времени выполнения операции удаления указанного слова, объёму затрачиваемой при этом памяти и количестве операций сравнения при обращении к файлу и использовании разных структур данных: деревьев (обычного и сбалансированного) и хеш-таблицы.

Формат вывода:

1. Вывод дерева
    - Список дуг

      “слово” -> “слово”

      “слово” -> “слово”

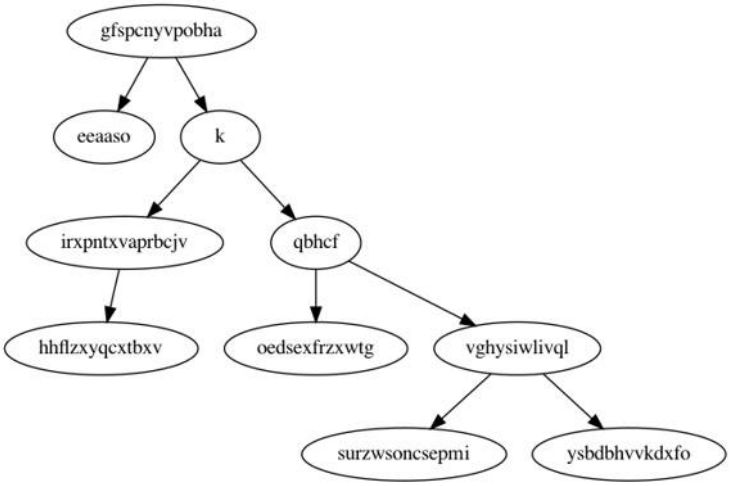
      Пример:

      gfspcnyvpobha -> eeaaso

      gfspcnyvpobha -> k

      k -> irxpntxvaprbcjv

      k -> qbhcf
    - Графическая интерпретация



2. Вывод сбалансированного дерева
    - Список дуг

      “слово” -> “слово”

      “слово” -> “слово”

      Пример:

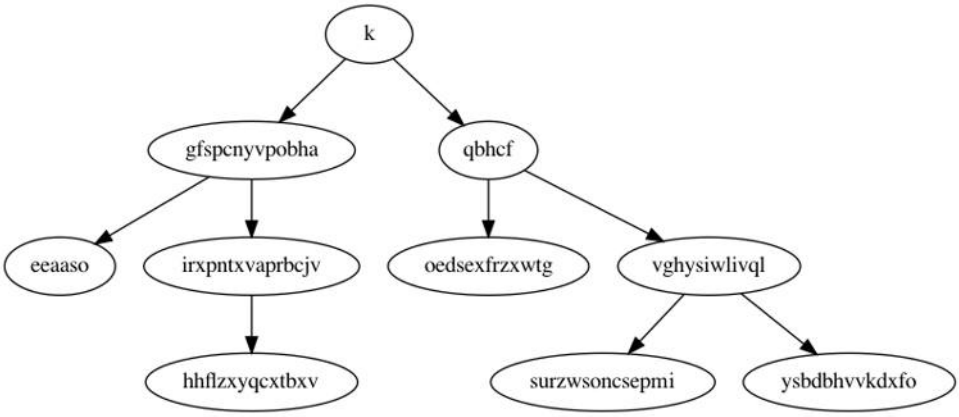
      gfspcnyvpobha -> eeaaso

      gfspcnyvpobha -> k

k -> irxpntxvaprbcjv

k -> qbhcf

- Графическая интерпретация



3. Хеш-таблица (например, из 10 элементов и при размере таблицы 5)

average conflicts: <float num>

current size: <size of table>

current hash func: <sum/mul/xor>

index: <num>

values in list: “слово”

index: <num>

values in list: “слово” “слово” “слово” “слово” “слово”

index: <num>

values in list: “слово” “слово” “слово”

4. Статистика выводится в виде таблиц.

3. *Описание задачи, реализуемой программой*

В программе производится операция удаления (поиска) в различных структурах данных и проводится сравнение их эффективности.

4. Способ обращения к программе

Способ обращения к программе - консольный. Дальнейшие инструкции будут выведены после запуска.

5. Описание возможных аварийных ситуаций и ошибок пользователя

- ошибка ввода действия;

- ошибка в названии файла или его отсутствие (физическое и/или в аргументах командной строки);
- отказ операционной системы выделить запрашиваемую память;

Во всех указанных случаях программа сообщит об ошибке

II. Описание внутренних структур данных

В программе есть 3 основных структуры данных:

1. Двоичное дерево поиска

Описание на языке С узла дерева выглядит таким образом:

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

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. Поэтому 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;

2. Сбалансированное двоичное дерево (АВЛ-дерево)

Описание на языке С выглядит таким образом:

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

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.

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;

3. Хеш-таблица

Описание таблицы на языке С выглядит таким образом:

```
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;
};
```

Вспомогательной структурой данных является односвязный список. Описание узла на языке С выглядит следующим образом:

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

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).

III. Описание алгоритма

1. Деревья

Обход

Основной рекурсивный подход для обхода (непустого) бинарного дерева: Начиная с узла N делаем следующее:

(L) Рекурсивно обходим левое поддерево. Этот шаг завершается при попадании опять в узел N.

(R) Рекурсивно обходим правое поддерево. Этот шаг завершается при попадании опять в узел N.

(N) Обрабатываем сам узел N.

Эти шаги могут быть проделаны в любом порядке:

– сверху вниз: N,L,R.

– слева направо: L,N,R

– снизу вверх: L,R,N

Вставка

Рекурсивный алгоритм:

1. Если на текущем шаге указатель на узел дерева пуст, то мы нашли место вставки, а значит можем создать новый узел и присвоить ему значение данных, переданных для вставки, а затем вернуть этот узел.
2. Если указатель на узел дерева не пуст, то сравним значение в этом узле с переданными данными: если значение узла больше, то продолжим поиск места вставки в левом поддереве, иначе в правом (случай равенства зависит от конкретной реализации, например, можно не включать узел, если такой уже есть).
3. Возврат текущего узла дерева (с изменённым левым или правым потомком). При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.

Удаление

Идея 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:

1. Если на текущем шаге указатель на узел дерева пуст, то мы пришли в потомок листа, а значит надо вернуть нулевой указатель.
2. 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.
3. 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).
4. 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

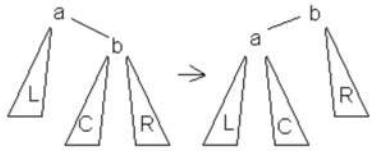
реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.

5. 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)

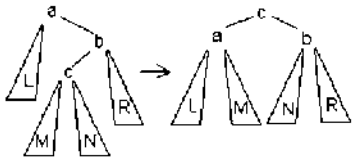
Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной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

Малое левое вращение



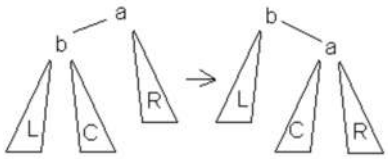
Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева <= высота R.

Большое левое вращение



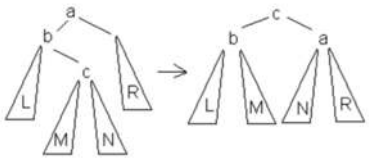
Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева > высота R.

Малое правое вращение



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

Большое правое вращение



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

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.

2. **Хеш-таблица**

Оценка по времени

Вставка

Поскольку для устранения коллизий используется метод цепочек (внешний тип хеширования), то вставка элементов сводится к вычислению значения хеш-функции и добавлению в односвязный список (O(1)).

Удаление

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)

Проверка выводов экспериментально

Количество элементов в файле = 500. В среднем получаются такие результаты:

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

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. Количество элементов = 500, размер хеш-таблицы = 500. Результаты в среднем:

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

Видим, 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.

Итог

Мы экспериментально подтвердили сделанные ранее оценки.

IV. Выводы по проделанной работе

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).

V. Ответы на вопросы

1. Что такое дерево?

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

2. Как выделяется память под представление деревьев?

Если дерево имеет списочную структуру, то память выделяется под каждый узел отдельно, а если дерево представлено массивом, то один раз на какой-то фиксированный размер.

3. Какие стандартные операции возможны над деревьями?

Обход вершин, поиск по дереву, включение узла в дерево, удаление узла.

4. Что такое дерево двоичного поиска?

Дерево двоичного поиска – это такое дерево, в котором все левые потомки моложе предка, а все правые – старше.

5. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

ИДС - дерево, у которого количество вершин в левом и правом поддеревьях отличается не более, чем на 1.

АВЛ дерево - двоичное дерево, у которого высота двух поддеревьев каждого узла дерева отличается не более чем на 1.

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

Из-за условия балансировки средняя длина поиска в АВЛ-дереве меньше, чем в ДДП.

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

Это структура данных, в основе которой лежит массив, но индекс, по которому располагается элемент, зависит непосредственно от значения элемента. Функция, которая реализует отображение из множества значений элементов в множество индексов называется хеш-функцией.

8. Что такое коллизии? Каковы методы их устранения.

Коллизия - ситуация, когда разным ключам (ключ вычисляется из значения элемента) соответствует одно значение хеш-функции. Для устранения или минимизации числа коллизий можно попробовать подобрать другую хеш-функцию. Если коллизия всё же возникла, то используется открытое или закрытое хеширование: при открытом для каждого индекса выстраивается цепочка из элементов, ключ которых соответствует данному индексу (то есть элементы помещаются в список, а указатель на голову хранится в хеш-таблице); при закрытом - если ячейка с вычисленным индексом занята, то просматриваются следующие записи таблицы по порядку (с шагом 1), до тех пор, пока не будет найден ключ или пустая позиция в таблице. При этом, если индекс следующего просматриваемого элемента определяется добавлением какого-то постоянного шага (от 1 до n), то данный способ разрешения коллизий называется линейной адресацией, существует также квадратичная адресация, при которой для вычисления шага применяется формула: h=h+a^2, где a – это номер попытки поиска ключа.

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

При большом количестве коллизий (в частности, когда количество элементов меньше размера таблицы)

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

См. «Описание алгоритма»