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: Мицевич Максим Дмитриевич

Группа: ИУ7-31Б

2020г.

IV. Описание условия задачи

В текстовом файле содержатся целые числа. Построить ДДП из чисел файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Построить хеш-таблицу из чисел файла. Использовать закрытое хеширование для устранения коллизий. Осуществить добавление введенного целого числа,если его там нет, в ДДП, в сбалансированноедерево, в хеш-таблицуи в файл.Сравнить время добавления, объем памяти и количество сравнений при использовании различных(4-х)структур данных. Если количество сравнений в хеш-таблице больше указанного(вводить),то произвести реструктуризацию таблицы, выбрав другую функцию.

II. Техническое задание

1. *Описание исходных данных*

Программа ожидает выбор от пользователя одного из пунктов меню:

1 – работа с файлом

2- работа с бинарным деревом

3 – работа с АВЛ деревом

4 – работа с хеш-таблицей

5 – вывод информации об эффективности алгоритмов

Возможные способы обработки структур данных, описанных выше:

1 – добавление элемента

2 – поиск элемента

3 – удаление элемента

4 - печать

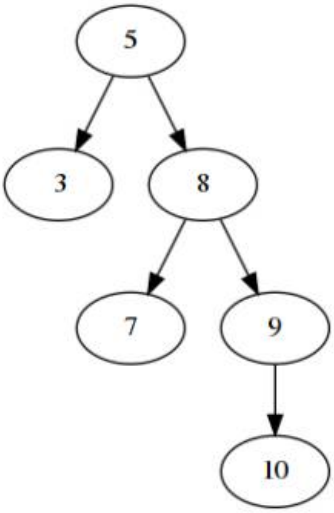
2. Описание результата программы

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

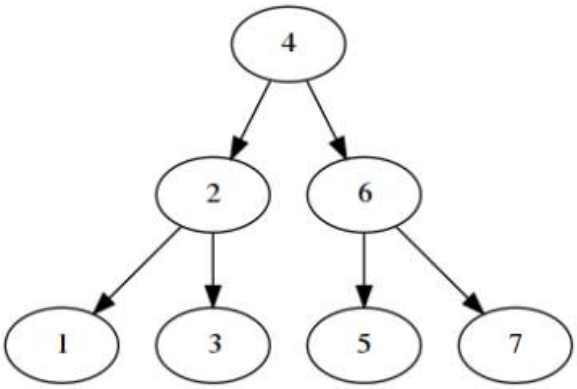
1. Двоичное дерево (его графическая интерпретация)
2. Сбалансированное двоичное дерево (его графическая интерпретация)
3. Хеш-таблица
4. Вывод содержимого файла
5. Статистика по времени выполнения операциям добавления и поиска числа в разных структурах данных с разным содержимым числа .

Формат вывода:

1. Вывод дерева



2. Вывод сбалансированного дерева



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

<func_const>

| index | data | type |

| 0 | <data> | <type> |

……

Возможные типы элемента:

-1 – элемент удален

0 – пустой элемент

1 – элемент используется

4. Статистика выводится в виде таблиц.

| struct type | time add | time search | avg comparisons | memory |

3. Описание задачи, реализуемой программой

Программа выполняет опреации добавления, поиска, удаления элемента в структуру данных и печать этой структры. Также возможна оценка эффективности алгоритмов добавления и поиска элемента в этих структурах.

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;
- неправильный ввод какого-либо параметра
- переполнение хеш-таблицы
- отказ операционной системы выделить запрашиваемую память;

Во всех указанных случаях программа сообщит об ошибке

III. 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 node_t;
struct node
{
    elem_t data;
    node_t *left;
    node_t *right;
};
```

Поле data отвечает за хранение ключа

left и right – указатели на левого и правого потомков

2. ***Сбалансированное двоичное дерево (АВЛ-дерево)***

Описание на языке С выглядит таким образом:

```
typedef struct balance_node balance_node_t;
struct balance_node
{
    elem_t data;
    unsigned char height;
    balance_node_t *right;
    balance_node_t *left;
}
```

Поле data отвечает за хранение ключа

height – высота дерева

right и left – указатели на правого и левого потомков

3. ***Хеш-таблица***

Описание таблицы на языке С выглядит таким образом:

```
typedef size_t (*hash_func_t) (elem_t, func_const_t, size_t);
typedef func_const_t (*gen_const_func_t) (void);

typedef enum FIELD_TYPE
{
```

```
    DELETED = -1,
    EMPTY   =  0,
    USE     =  1
}field_type_t;

typedef struct
{
    elem_t elem;
    field_type_t field_type;
}field_t;

typedef struct
{
    field_t *table;
    size_t size;
    func_const_t func_const;
    hash_func_t hash_func;
}hash_table_t;
```

В ячейке таблицы хранится ее тип и ключ

table – указатель на таблицу

size – рзамер таблицы

func_const – константа, требуемая для хеш-функции

hash_func - хеш-функция

IV. Описание алгоритма

1. ДДП и АВЛ-дерево

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.

Поиск

Рекурсивный алгоритм:

1. Если указатель на текущий узел равен нулевому, то выполняем выход из функции (искомый элемент найти не удалось)
2. Если значение текущего узла равно требуемому элементу, то выполняем выход из функции (требуемый элемент найден)
3. Если значение ключа текущего узла больше искомого, то запускаем алгоритм из левого потомка, иначе – из правого.

Удаление

Рекурсивный алгоритм:

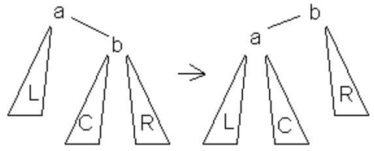
1. Выполняем поиск требуемого элемента. Если элемент не найден, то выполняем выход из функции (удалять нечего). Если элемент найден, то возможны разные варианты дальнейшего развитияю
2. Если найденный элемент находится в листе, то указатели на этот узел присваиваем ноль и очищаем под ним память

3. Если у найденного узла только один поток, то указателю на найденный узел присваиваем значение его потомка и очищаем память под узлом.
4. Если у найденного узла два потомка, то заменяем значение его ключа либо на наибольший элемент в левом поддереве, либо на наименьший в правом, а сам узел с наибольшим (наименьшим) элементом удаляем. Данный элемент будет находиться в листе, поэтому удаление происходит согласно пункту 2.
5. В случае АВЛ дерева, требуется балансировка узлов после выхода из рекурсии.

Балансировка узла

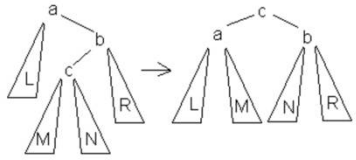
Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной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

Малое левое вращение



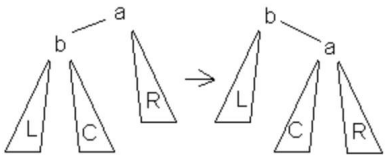
Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева <= высота R.

Большое левое вращение



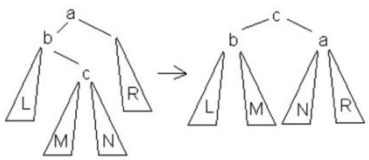
Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева > высота R.

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

Для больших N имеет место оценка высоты дерева 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**

Вставка

- вычисляется значение хеш-функции для требуемого ключа
- если ячейка с полученным индексом пустая или удаленная, то в нее вставляется элемент и меняется статус ячейки на “используемая”
- если нет, то просматривается следующая ячейка, до тех пор, пока мы не найдем требуемую или не вернемся к изначальной
- если мы вернулись к изначальной, то таблица переполнена

Поиск

- вычисляется хеш-функция для требуемого ключа
- если текущая ячейка используется и значение ее ключа равно требуемому, то выполняется возврат из фукнции (элемент найден)
- если нет, то идем по следующим ячейкам, пока не встретим пустую или не вернемся к исходной
- если мы нашли пустую или вернулись к исходной, то выходим из функции (элемент не найден)

Удаление

- Выполняется поиск элемента
- Если он найден, то его ячейке присваивается статус удаленной

Оценка по времени

Вставка

Вставка происходит с сложностью в промежутке от O(1) до O(N) в зависимости от количества коллизий.

Удаление

Сложность удаление аналогична сложности вставки.

Оценка по памяти

Количество занимаемой таблицей памяти зависит не от количества элементов в таблице, а от количества элементов, под которые выделена память. То есть если в таблице много свободных ячеек, то память рассходуется неэффективно.

Проверка выводов экспериментально

Количество элементов в файле = 1000, элементы выбираются случайным образом. В среднем получаются такие результаты:

Хеш-функция – взятие остатка деления на размер таблицы

struct type	time add	time search	avg comparisons	memory
file	1.067330s	0.917470s	4957.958000	8b
binary tree	0.000116s	0.000131s	11.744000	24000b
balance tree	0.000396s	0.000140s	9.218000	32000b
hash table	0.000043s	0.000060s	1.950000	23248b

Время работы с файлом ожидаемо оказалось хуже по сравнению со всеми остальными. Это связано с тем, что при работе с файлом постоянно происходит обращение к внешнему ресурусу. Время добавления элемента в хэш-таблицу лучше времени добавления в деревья, так как деревья основаны на списках и при добавлении элемента происходит выделение памяти, в то время как память под таблицу выделяется заранее. Время добавления в АВЛ-дерево больше, чем в обычное, так как после добавления элемента нередко происходит балансировка дерева. Касательно времени поиска, лучшее время показала хэш-таблица, так как хэш-функция и размер таблицы подбирались под входные данные изначально и средняя сложность поиска оказалась O(1). Время поиска в сбалансированном дереве ожидаемо меньше времени поиска в обычном дереве. По памяти лучше всего себя показала хэш-таблица, так как память требуемая под 1 элемент у нее меньше, чем у деревьев.

Теперь посмотрим, что произойдет, если поступят отсортированные данные. Количество элементов = 1000. Результаты в среднем:

Хеш-функция – взятие остатка деления на размер таблицы

```
elements - 1000
|  struct type |   time add | time search | avg comparisons |   memory |
|         file |  1.300990s |   1.163890s |     5527.374000 |       8b |
|  binary tree |  0.004549s |   0.002652s |      500.500000 |   24000b |
| balance tree |  0.000693s |   0.000071s |        8.987000 |   32000b |
|   hash table |  0.000030s |   0.000046s |        1.659000 |   16304b |
```

Видим, что ДДП резко проиграло по времени в среднем сбалансированному дереву (в 30 раз) и хеш-таблице (в 40 раз). А хеш-таблица и сбалансированное дерево вне зависимости от данных сохранили результаты по времени и памяти.

При использования хеш-функции умножения (берется дробная часть произведения ключа на константу и умножается на размер массива) получаются похожие результаты. Связано это с тем, что операция деления более затратная, чем операция умножения, но в то же время в функции умножения больше операций и есть вызов функций.

Итог

Мы экспериментально подтвердили сделанные ранее оценки.

V. Выводы по проделанной работе

Если входные данные заранее известны, то самой эффективной структурой будет хэш-таблица, так как есть возможность заранее подобрать правильную хэш-функцию и размер таблицы (в таком случае сложность поиска будет O(1)). Если входные данные заранее не известны, то в таблице может возникнуть много коллизий, что приведет к снижению ее эффективности. В таком случае, нужно будет реструктизировать таблицу, что может занять не мало времени. Возможно, если входные данные заранее не известны, то эффективнее будет использовать АВЛ-дерево. Оно проиграет хэш-таблице по памяти, но при определенном раскаладе может дать выгрышь по времени.

Функции

Добавление, чтение из файла, удаление, поиск в сбалансированном дереве

int add_to_btree(balance_node_t **root, elem_t elem);

int fread_btree(char *file_name, balance_node_t **root);

void delete_from_btree(balance_node_t **root, elem_t elem)

void search_in_btree(balance_node_t *root, elem_t elem, size_t *comparisons, int *find);

Добавление, чтение из файла, удаление, поиск в дереве

int add_to_tree(node_t **root, elem_t elem);

int fread_tree(char *file_name, node_t **root);

void delete_from_tree(node_t **root, elem_t elem);

void search_in_tree(node_t *root, elem_t elem, size_t *comparisons, int *find)

Чтение из файал, удаление, подсчет среднего количества сравнений, реструктизация, поиск в хеш-таблице

int fread_table(char *file_name, hash_table_t *table);

void delete_from_table(hash_table_t *table, elem_t elem);

double count_avg_comparisons(hash_table_t table);

int restruct_table(hash_table_t *table, hash_func_t h_func, gen_const_func_t gen_const);

void search_in_table(hash_table_t table, elem_t elem, size_t *comparisons, int *find)

Тесты

Позитивные тесты

Входные данные	Что проверяем	Ожидаемый выходной результат
Выбор: Работа с сбалансированным деревом Выбор: добавление и печать 1 2 3 4 5 6 7 8 9 10	Проверка балансировки узлов	Дерево должно быть сбалансированным
Выбор: Работа с обычным деревом Выбор: добавление и поиск 5 3 7 4 6 8 Элемент 3	Проверка поиска	Элемент 3 найден за 2 сравнения
Выбор: Работа с обычным деревом Выбор: добавление и поиск 5 3 7 4 6 8 Элемент 9	Проверка поиска	Не найден
Выбор: работа с хеш-таблицей Добавление элемента в таблицу и превышение максимального среднего количества сравнений	Проверка реструктизации таблица	Таблица должна перестроится с другой функцией и размером

Негативные тесты

Входные данные	Что проверяем	Ожидаемый выходной результат
Выбор: работа с хеш-таблицей Добавление элемента в таблицу и превышение количества элементов в таблице	Переполнение таблицы	Table overflow

VI. 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»