

Хеш-таблицы, деревья

Юрий Литвинов

yurii.litvinov@gmail.com

06.04.2018г

Абстрактные типы данных

- ▶ АТД — некоторая математическая модель и набор операций, определённый в рамках этой модели
 - ▶ Обобщение понятия “тип”
- ▶ Состоит из типа данных и операций, выполняющих над ним преобразования
 - ▶ Внутреннее устройство типа данных невидимо для остальной программы (принцип сокрытия деталей реализации)
 - ▶ Работа с АТД — только с помощью связанных с ним функций
 - ▶ Тип данных и операции для работы с ним лежат рядом, так, чтобы все изменения в АТД были локализованы и не затрагивали остальную программу (принцип инкапсуляции)
- ▶ В объектно-ориентированных языках АТД реализуется через классы

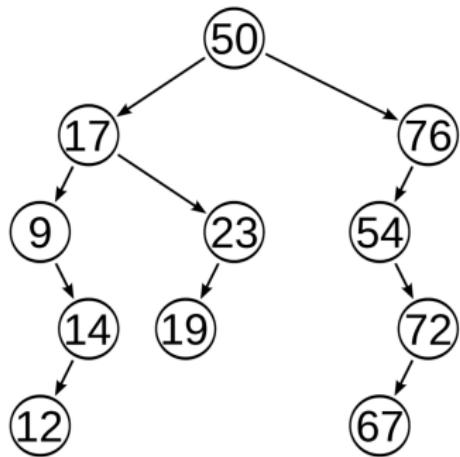
Пример — стек

- ▶ Функции:
 - ▶ createStack()
 - ▶ deleteStack()
 - ▶ push()
 - ▶ pop()
 - ▶ isEmpty()
- ▶ **Внешнему миру вообще всё равно, как стек устроен внутри**
 - ▶ Может быть на массиве
 - ▶ Может быть на указателях

Дерево

Ещё один абстрактный тип данных, используемый в программировании повсеместно

- ▶ Файловая система
- ▶ Абстрактное синтаксическое дерево
 - ▶ Дерево разбора арифметического выражения
- ▶ Двоичное дерево поиска
- ▶ Дерево контроллов (или виджетов) в пользовательском интерфейсе
- ▶ ...



Определения

Дерево — совокупность элементов, называемых **узлами** (один из которых — **корень**), и отношений, образующих иерархическую структуру узлов.

- ▶ Узел является деревом, он же — корень дерева
- ▶ Есть узел n и деревья T_1, T_2, \dots, T_k — деревья с корнями n_1, n_2, \dots, n_k соответственно. Тогда можно построить новое дерево, с корнем n и поддеревьями T_1, T_2, \dots, T_k . Узлы n_1, n_2, \dots, n_k называются **сыновьями** узла n .

Нулевое дерево — дерево без узлов.

Дерево — связный ациклический граф.

Несвязный ациклический граф — лес.

Ещё определения

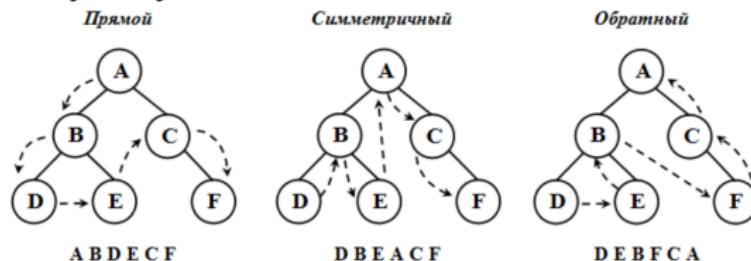
- ▶ Путь из n_1 в n_k — последовательность узлов n_1, \dots, n_k , в которой каждый узел является родителем следующего
- ▶ Длина пути — число, на единицу меньшее количества узлов, составляющих путь
- ▶ Путь нулевой длины — путь из узла к самому себе
- ▶ Узел a называется предком узла b , если существует путь из a в b , b в этом случае — потомок a
 - ▶ Каждый узел — предок и потомок самого себя
- ▶ Потомок, не являющийся самим узлом, называется истинным потомком
 - ▶ С предком аналогично
- ▶ Узел, не имеющий истинных потомков, называется листом
- ▶ Поддерево какого-либо дерева — узел вместе со всеми потомками

И ещё определения

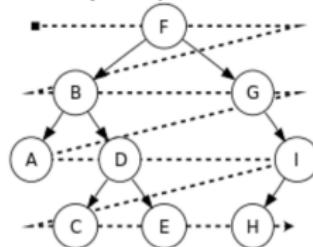
- ▶ Высота узла — длина самого длинного пути из узла до какого-либо листа
- ▶ Глубина узла — длина пути от узла до корня
- ▶ Высота дерева — высота корня
- ▶ Деревья бывают упорядоченными и неупорядоченными
 - ▶ Можно упорядочить узлы дерева, не связанные отношением предок-потомок (слева-справа)
- ▶ Деревья бывают помеченными (каждой вершине сопоставлено значение)

Обходы

В глубину:



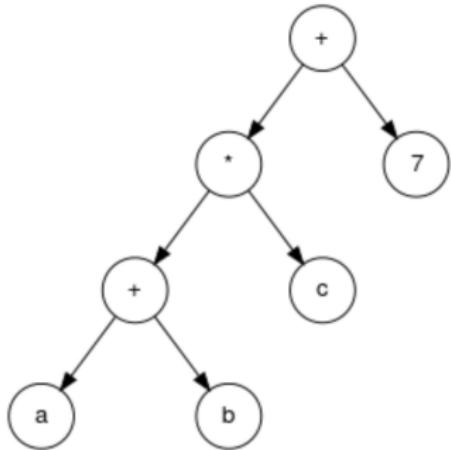
В ширину:



Деревья выражений

$$(a + b) * c + 7$$

- ▶ Прямой порядок — префиксная запись
 - ▶ $+ * + a b c 7$
- ▶ Обратный порядок — постфиксная запись
 - ▶ $a b + c * 7 +$
- ▶ Симметричный порядок — инфиксная запись
 - ▶ $a + b * c + 7$



АТД “Дерево”

- ▶ parent(n, t)
- ▶ leftmostChild(n, t)
- ▶ rightSibling(n, t)
- ▶ label(n, t)
- ▶ create(n, t₁, ..., t_i)
- ▶ root(t)
- ▶ makenull(t)

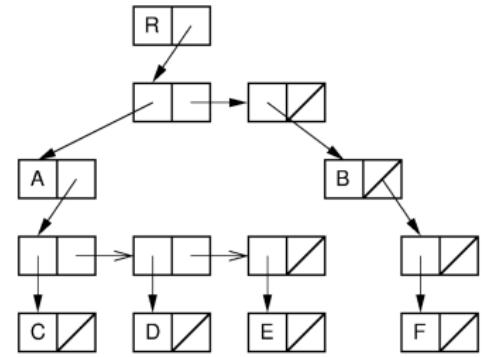
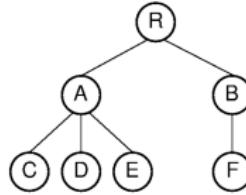
```
void preorder(Node *n)
{
    cout << label(n);
    Node *child = leftmostChild(n);
    while (child != nullptr)
    {
        preorder(child);
        child = rightSibling(child);
    }
}
```

Нерекурсивный обход в прямом порядке

```
void nonRecursivePreorder(Node *n) {  
    stack<Node*> s;  
    Node *current = n;  
    while (true) {  
        if (current != nullptr) {  
            cout << label(current) << " ";  
            s.push(current);  
            current = leftmostChild(current);  
        } else {  
            if (s.empty())  
                return;  
            current = rightSibling(s.top());  
            s.pop();  
        }  
    }  
}
```

Реализация списком сыновей

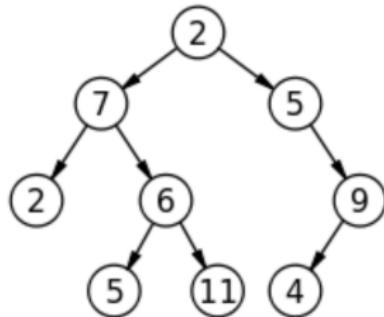
```
struct Node
{
    ElementType value;
    Node *sibling;
    Node *child;
};
```



Двоичные деревья

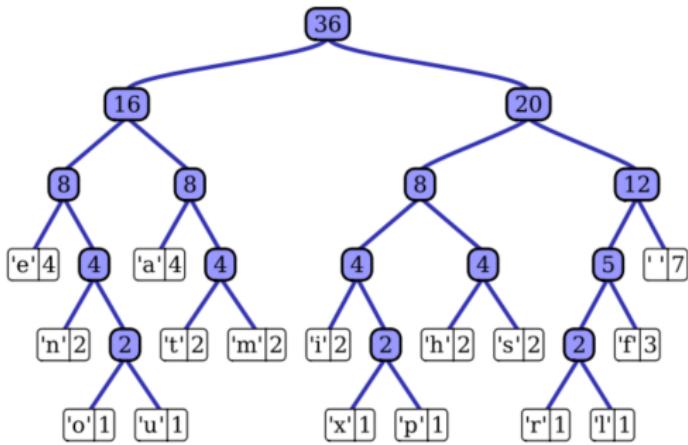
Деревья, у которых есть левый и правый сын, и это разные вещи

```
struct Node
{
    ElementType value;
    Node *leftChild;
    Node *rightChild;
};
```



Пример: алгоритм Хаффмана

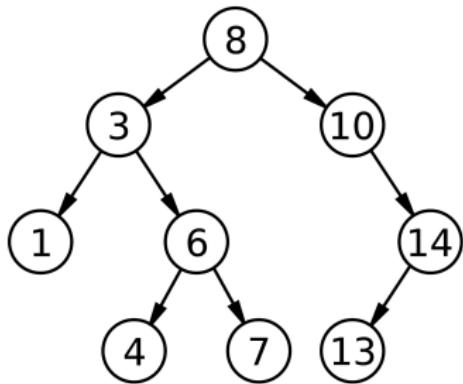
- ▶ Алгоритм сжатия, вычисляющий кратчайшую кодовую последовательность для символа
 - ▶ Если в тексте одни буквы “A”, нет смысла кодировать A 16-ю битами
- ▶ Префиксные коды
- ▶ Дерево частот символов



Пример: “this is an example of a huffman tree”

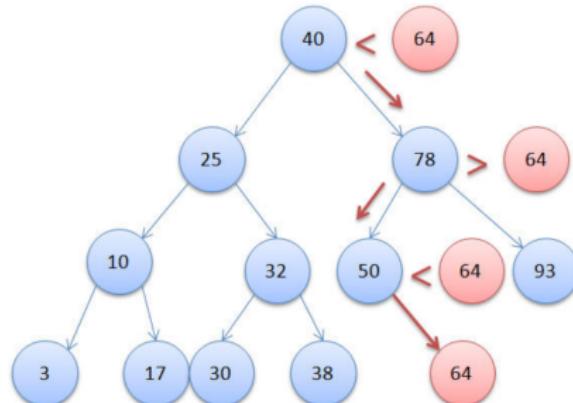
Двоичное дерево поиска

- ▶ Двоичное дерево, у которого для каждого узла в левом поддереве элементы, меньшие значения в узле, в правом — элементы, большие значения в узле
- ▶ Используется для представления множеств и ассоциативных массивов
 - ▶ Если дерево сбалансировано (т.е. высота примерно логарифм количества вершин), операции вставки, удаления и поиска выполняются за $\log(n)$

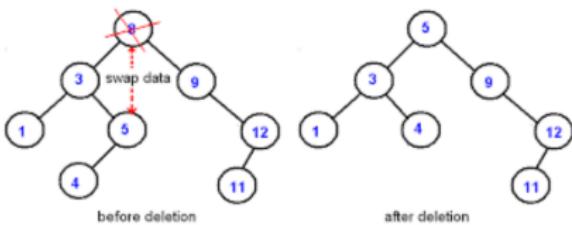


Операции

Вставка:

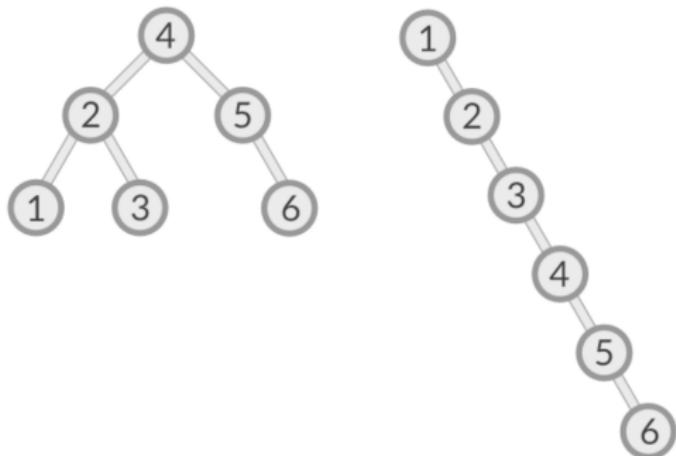


Удаление:



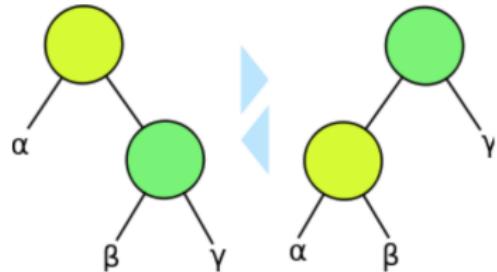
Проблема

- ▶ При неудачном порядке вставки дерево может выродиться в список
- ▶ $n \leq 2^{h+1} - 1$, поэтому $h \geq \log_2(n+1) - 1 \geq \lfloor \log_2(n) \rfloor$
- ▶ Трудоёмкости всех операций сразу станут линейными



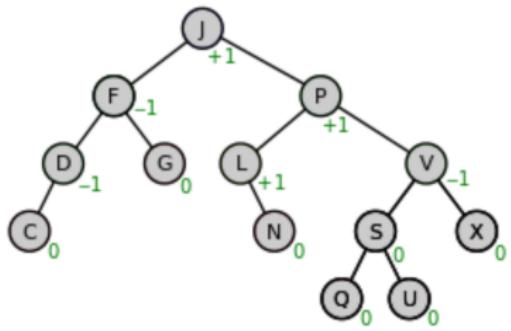
Балансировка

- ▶ Перестраиваем дерево каждый раз после вставки и удаления, чтобы сохранить высоту дерева возможно меньшей
- ▶ Основная операция — поворот
 - ▶ Сохраняет свойства двоичного дерева поиска
 - ▶ Возможно, уменьшает его общую высоту
- ▶ Конкретных алгоритмов балансировки очень много



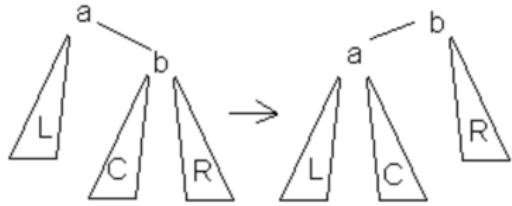
АВЛ-дерево

- ▶ 1962г., Г.М. Адельсон-Вельский и Е.М. Ландис
- ▶ В каждой вершине хранится разность высот левого и правого поддерева
- ▶ Вставка и удаление гарантируют, что разность высот будет не больше 1
- ▶ Теоретически лучшая балансировка из популярных деревьев, но относительно большой оверхэд

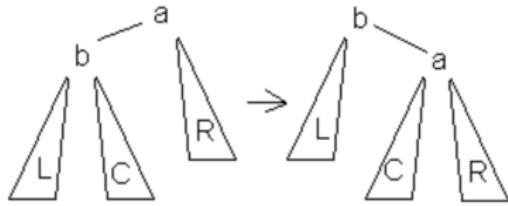


Балансировка

Малое левое вращение



Малое правое вращение

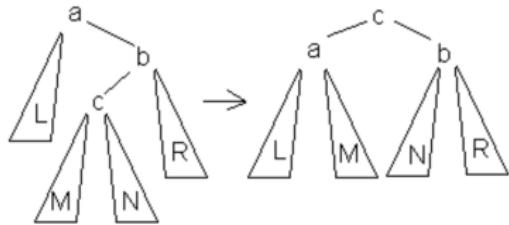


Проводится в случае, если высота $b >$ высота $L + 1$, и высота $C \leq$ высоте R

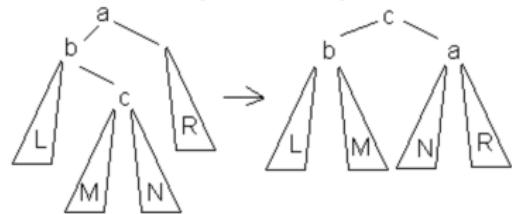
- ▶ При повороте важно не забыть обновить значения баланса
- ▶ И не запутаться в указателях

Балансировка

Большое левое вращение



Большое правое вращение



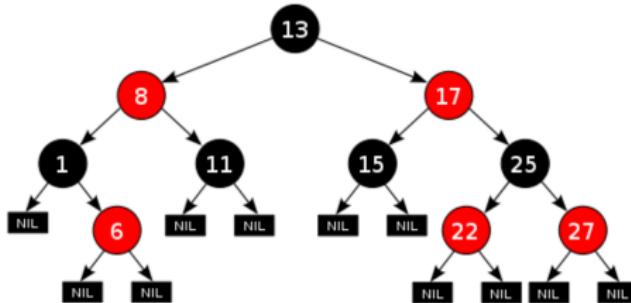
Балансировка выполняется на обратном проходе рекурсии при вставке и удалении, если есть потребность

Красно-чёрные деревья

- ▶ 1972г., Р. Байер
- ▶ Хуже сбалансированы, чем АВЛ-деревья, зато не придуманы в Советском Союзе требуют константного количества поворотов на каждую операцию (в отличие от $O(\log(n))$ для АВЛ-деревьев)
 - ▶ Поэтому используются практически во всех стандартных библиотеках

Красно-чёрные деревья

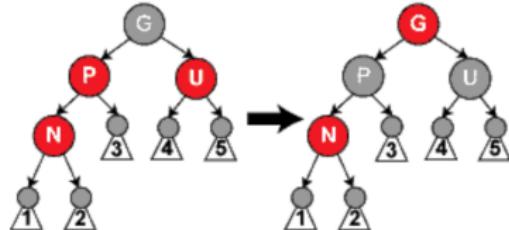
- ▶ В каждой вершине хранится цвет (красный или чёрный)
 - ▶ Корень чёрный
 - ▶ Все листья чёрные
 - ▶ Оба потомка красного узла — чёрные
 - ▶ Всякий путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов
 - ▶ Высота поддеревьев не может отличаться более, чем вдвое



Красно-чёрное дерево, добавление

Случаи 1-3

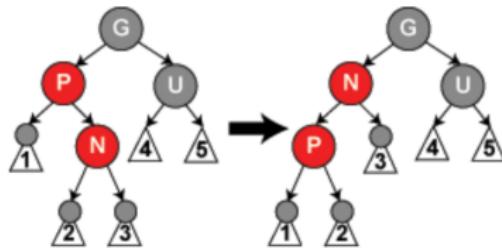
- ▶ Добавляем в корень — ок, красим его в чёрный
- ▶ Добавляем как сына чёрному узлу — ок, красим в красный
- ▶ Если родитель и “дядя” красные, перекрашиваем их и добавляем наш узел как красный. Дедушка может нарушить ограничения, так что, возможно, его тоже придётся перекрасить (выполнив перекрашивание рекурсивно до корня)



Красно-чёрное дерево, добавление

Случай 4

- Родитель красный, дядя чёрный, узел справа от родителя. Выполняем поворот пары “родитель-сын”.

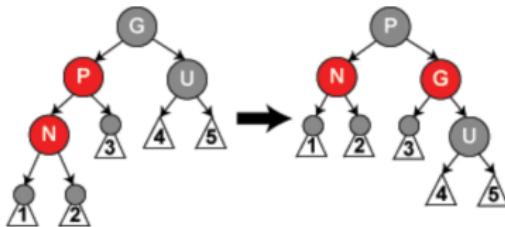


Ограничение “оба потомка красного узла чёрные” всё ещё
нарушается, но об этом позаботится случай 5.

Красно-чёрное дерево, добавление

Случай 5

- ▶ Родитель красный, дядя чёрный, узел слева от родителя. Выполняем поворот относительно пары “родитель-дедушка”, который и восстанавливает балансировку.



Опять-таки, надо не забыть перекрасить узлы

Красно-чёрное дерево, удаление

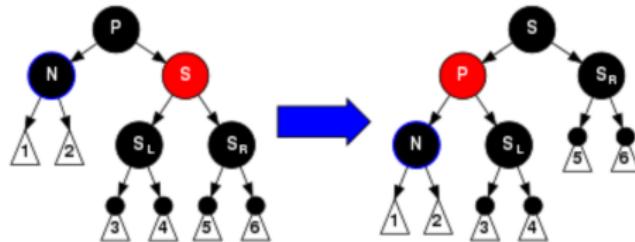
Сначала делаем как обычно — кладём значение самого большого узла в левом поддереве в удаляемый узел и... надо удалить тот узел, откуда мы взяли значение, но не всё так просто.

- ▶ Если он красный, то оба его потомка — чёрные листы. Удаляем красный узел и ставим на его место лист (они не хранят значений, поэтому не важно, какой)
- ▶ Если он чёрный, а его единственный нелистовой потомок красный, то ставим потомка на его место и перекрашиваем его в чёрный
- ▶ Если он чёрный и его потомок чёрный, то его оба потомка листы, но если кого-то просто удалить, то число чёрных узлов в поддереве изменится, так что надо перебалансировать дерево

Красно-чёрное дерево, удаление

Случаи 1-2

- ▶ Самый простой случай, когда удаляемый узел корень: просто удаляем
- ▶ У удалённого узла был красный брат: делаем поворот по ребру “отец-брать”

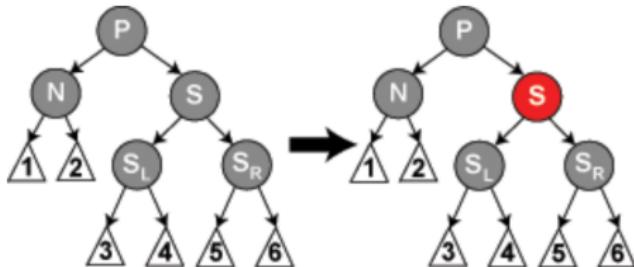


Сильно лучше не стало, потому что поддеревья всё ещё имеют разную чёрную высоту, но теперь можно применить правила 4, 5 или 6

Красно-чёрные деревья, удаление

Случай 3

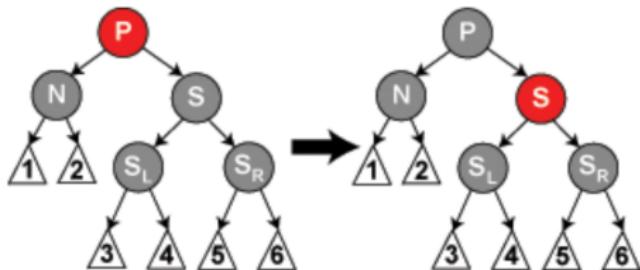
Если родитель чёрный, брат и его сыновья чёрные: перекрашиваем брата в красный. Поскольку из левого поддерева мы только что удалили один чёрный узел, а в правом поддереве один чёрный узел покрасили в красный, баланс восстановлен. Но только в поддереве, потому как оно стало на 1 чёрный узел короче, надо перебалансировать родителей.



Красно-чёрные деревья, удаление

Случай 4

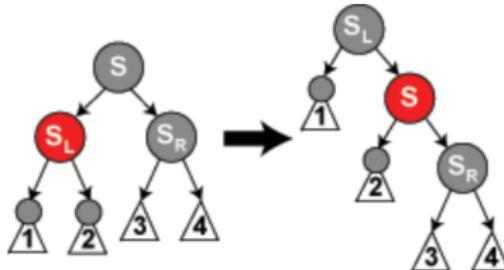
Брат и сыновья брата чёрные, но родитель красный — просто перекрасить брата нельзя. А вот перекрасить одновременно брата и родителя можно, это восстановит баланс (причём, во всём дереве сразу, потому что его чёрная высота не изменится).



Красно-чёрные деревья, удаление

Случай 5

Левый сын брата красный, правый — чёрный. Выполняем поворот относительно брата и левого сына, одновременно перекрашивая брата и левого сына:

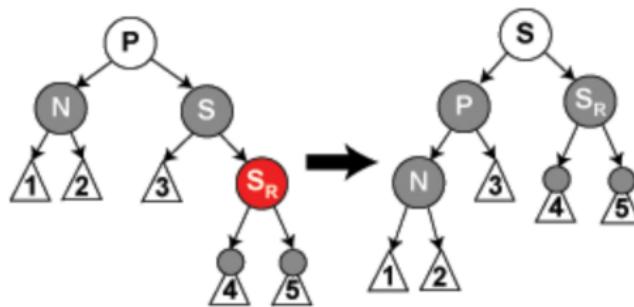


Глобально ничего не поменялось, но теперь можно применить случай 6.

Красно-чёрные деревья, удаление

Случай 6

Брат чёрный, его правый сын красный, левый — чёрный: выполняем поворот вокруг ребра “родитель-брать” и перекрашиваем узлы:



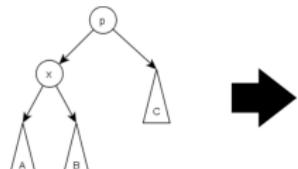
Баланс восстановлен (в левом поддереве на один чёрный узел больше), при этом можно доказать, что это всё ещё красно-чёрное дерево.

Splay-деревья

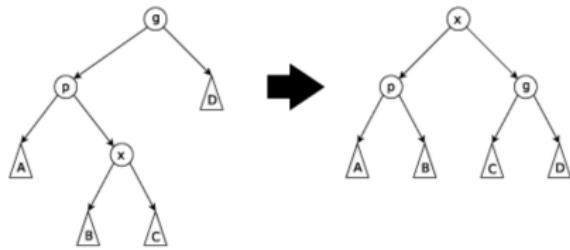
- ▶ 1985г., Д. Слитор и Р.А. Тарьян
- ▶ Продвигает узлы, к которым часто происходит обращение, ближе к корню, поэтому может быть быстрее остальных деревьев
- ▶ Не хранит дополнительных данных в узлах
- ▶ Не гарантирует сбалансированности
- ▶ Не дружит с параллельными алгоритмами
- ▶ Проще в реализации

Splay-деревья, splaying

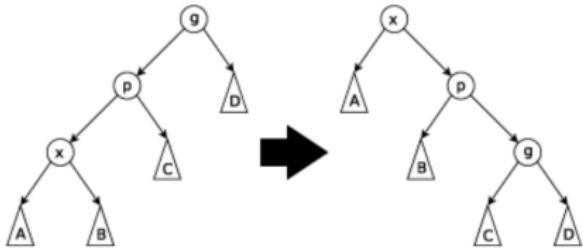
Zig:



Zig-zag:



Zig-zig:

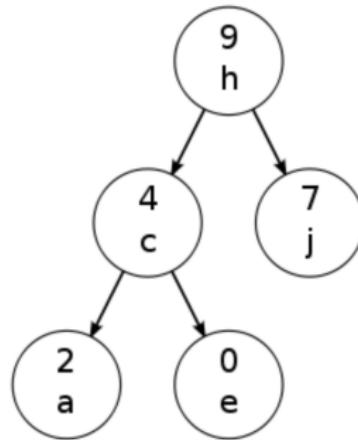


Splay-деревья, операции

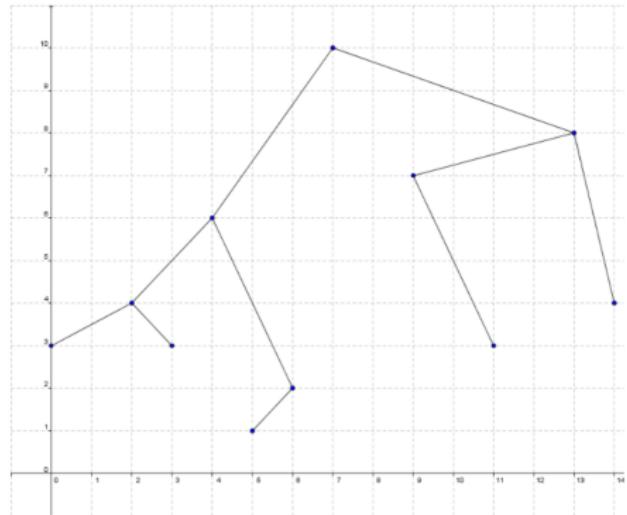
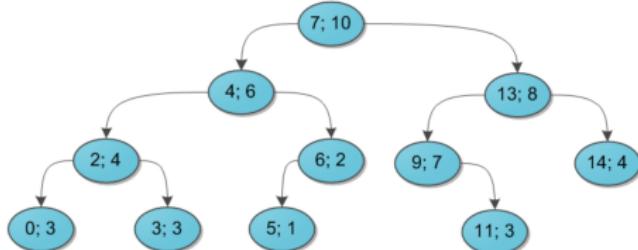
- ▶ Поиск:
 - ▶ Ищем узел как в обычном двоичном дереве поиска
 - ▶ Выполняем серию splaying-ов до тех пор, пока найденный узел не окажется корнем
- ▶ Вставка:
 - ▶ Вставляем узел как обычно в двоичное дерево поиска
 - ▶ Выполняем серию splaying-ов до тех пор, пока вставленный узел не окажется корнем
- ▶ Удаление:
 - ▶ Удаляем узел как обычно
 - ▶ Ташим родителя удалённого узла в корень дерева

Декартовы деревья

- ▶ Бинарное дерево поиска и куча одновременно
 - ▶ Храним ключ и “приоритет”
 - ▶ Приоритет выбирается случайно (!) при добавлении ключа
 - ▶ Куча по приоритету
- ▶ Тоже лишь примерно сбалансировано
- ▶ Легко пишется
 - ▶ Поэтому любимо олимпиадниками



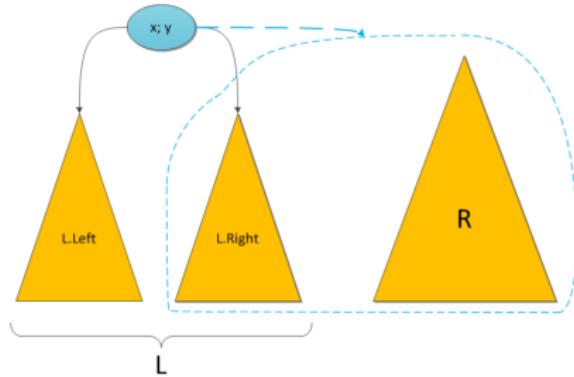
Декартово дерево и плоскость



© <https://habrahabr.ru/post/101818/>

Merge

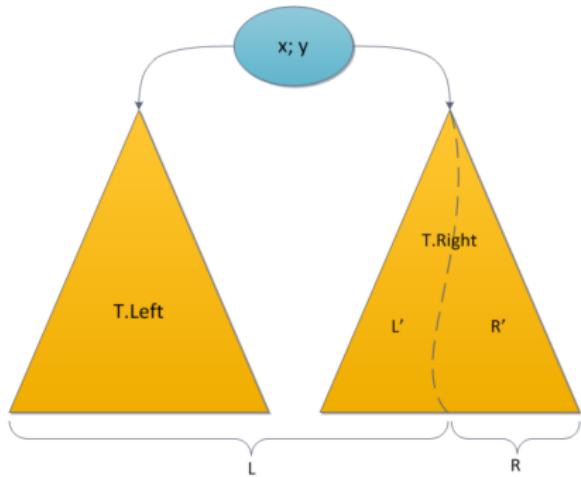
- ▶ Сливает два декартовых поддерева в одно
- ▶ Ключи в левом поддереве должны быть меньше ключей в правом



© <https://habrahabr.ru/post/101818/>

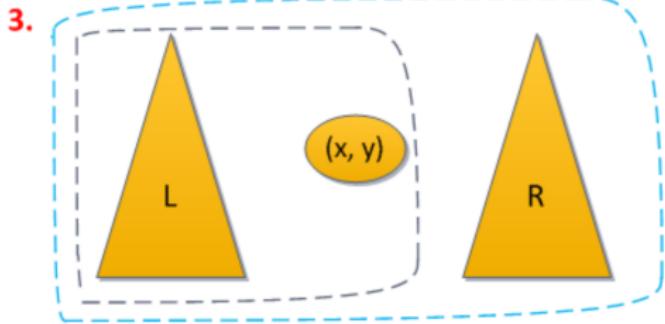
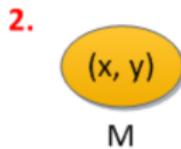
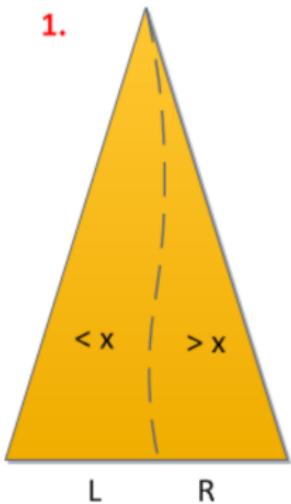
Split

- ▶ Разделяет декартово дерево на два
- ▶ Ключи в левом меньше заданного, ключи в правом больше



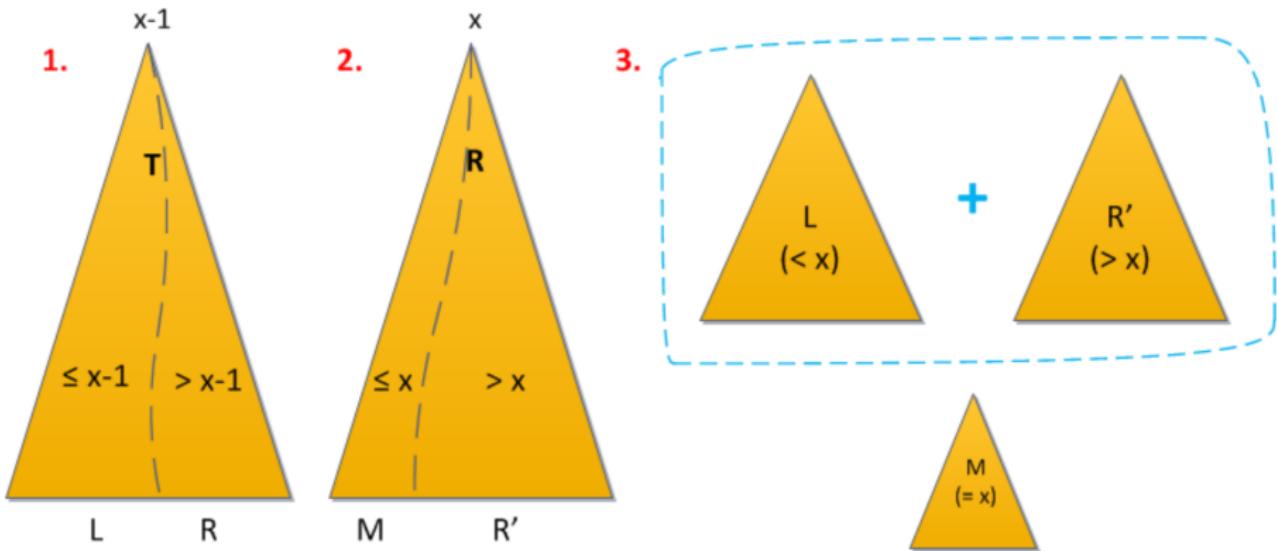
© <https://habrahabr.ru/post/101818/>

Добавление



© <https://habrahabr.ru/post/101818/>

Удаление



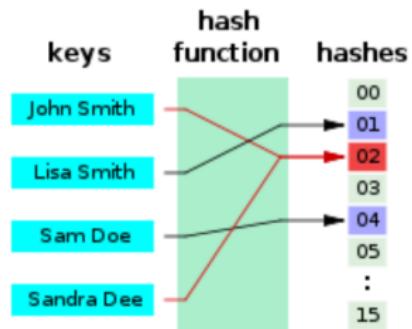
© <https://habrahabr.ru/post/101818/>

Хеш-таблицы

- ▶ Реализация абстрактного типа данных “множество” или “ассоциативный массив”
- ▶ Требует в среднем константного времени для операций вставки, удаления и поиска
 - ▶ Очень похожа на массив
- ▶ Хранит значения неупорядоченными
- ▶ Сильно зависит от качества хеш-функции

Хеш-функция

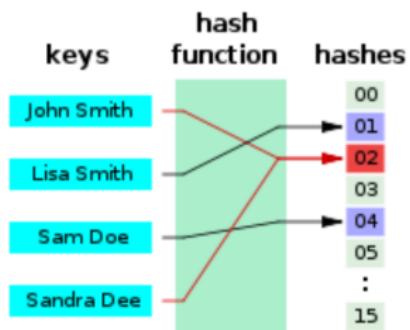
- ▶ Некоторая функция, отображающая большое (потенциально бесконечное) множество **ключей** в конечное (и маленькое) множество **хеш-значений**
 - ▶ Не инъективна
- ▶ Чем “случайнее” она это делает, тем лучше
 - ▶ Немного разным ключам должны соответствовать сильно разные значения



Хеш-функция

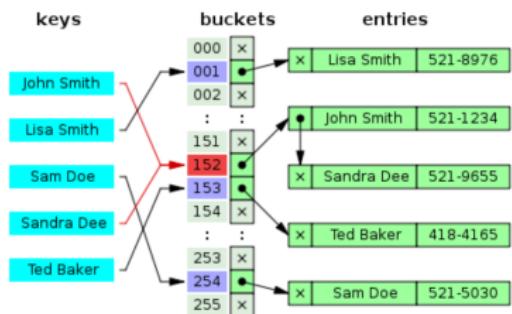
- ▶ Зачем

- ▶ Факторизуем множество ключей по классам эквивалентности, образованным ключами с равными хеш-значениями, будем хранить в массиве фактор-множества
- ▶ Хеш-значения можно использовать как индексы массива, где лежит что-то, что позволяет найти ключ (**сегменты**), и чем лучше хеш-функция перемешает ключи, тем меньше вероятность коллизии



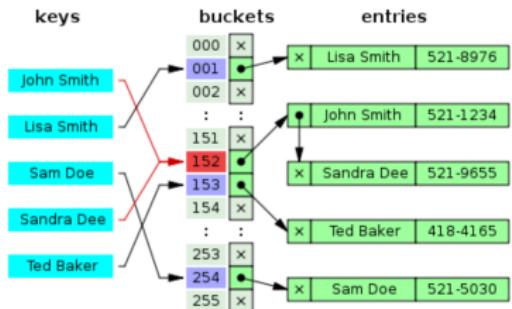
Хеш-таблица со списками значений

- ▶ Парадокс дней рождения:
В группе, состоящей из 23 или более человек, вероятность совпадения дней рождения (число и месяц) хотя бы у двух людей превышает 50 %
- ▶ Будем хранить в массиве список ключей с одинаковым хеш-значением



Хеш-таблица, “открытая адресация”

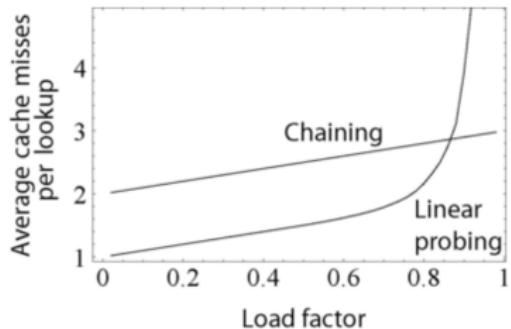
- ▶ Второй способ: будем хранить в хеш-таблице сами ключи со значениями, а если ячейка уже занята, брать следующую (может быть, по какому-нибудь сложному правилу)
- ▶ Удаление требует дополнительной информации
 - ▶ Пустая ячейка будет воспринята как конец цепочки
 - ▶ Обычно делают флаг “ячейка была удалена”
 - ▶ При вставке — вставляют
 - ▶ При поиске и удалении — идут дальше



Коэффициент заполнения

- ▶ Пусть n — число элементов в хеш-таблице, k — число сегментов (в английской литературе сегменты называются buckets). Коэффициент заполнения хеш-таблицы $L = n/k$

- ▶ Коэффициент заполнения должен быть примерно равен 1 для хеш-таблиц со списками и < 0.7 для хеш-таблиц с открытой адресацией
 - ▶ Динамическое изменение размеров массива



Выбор хеш-функции

- ▶ Должна быть возможно более случайной
- ▶ Должна зависеть только от ключа
- ▶ Должна считаться быстро
- ▶ Например:

```
int h(char *value) {  
    int result = 0;  
    for (int i = 0; value[i] != '\0'; ++i)  
        result = (result + value[i]) % hashSize;  
    return result;  
}
```

- ▶ Обычно хеш-функция возвращает просто целое число, а хеш-таблица сама “загоняет” его в нужный диапазон значений

Ещё про хеш-функции

- ▶ Для целых чисел вполне сойдёт id
- ▶ “Совершенная хеш-функция” — инъективна
- ▶ Универсальная хеш-функция — семейство функций
- ▶ Криптографические хеш-функции
 - ▶ MD5
 - ▶ SHA1
 - ▶ Небыстро считаются, поэтому не подходят
- ▶ Хеш-функции для сложных типов данных
 - ▶ Сумма или произведение хеш-функций элементов, как для строки
 - ▶ Значение полинома $a[0] * p^n + a[1] * p^{n-1} + \dots + a[n]$, особенно если p — простое
 - ▶ Rolling hash
 - ▶ xor хеш-функций элементов