

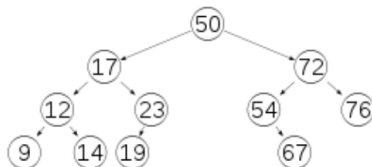
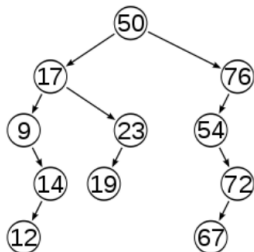
# Самобалансирующие деревья

Юрий Литвинов  
yurii.litvinov@gmail.com

03.11.2020

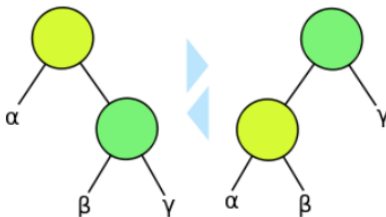
# Проблема

- ▶ Если в обычное двоичное дерево поиска вставлять элементы в возрастающем (или убывающем) порядке, оно выродится в список
- ▶  $n \leq 2^{h+1} - 1$ , поэтому  $h \geq \log_2(n + 1) - 1 \geq \text{floor}(\log_2(n))$



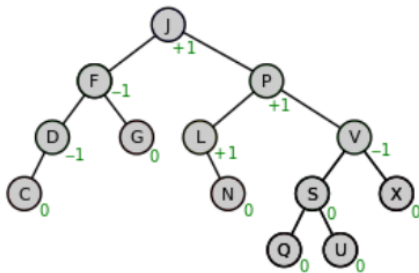
# Балансировка

- ▶ 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



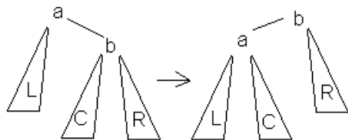
# АВЛ-дерево

- ▶ 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

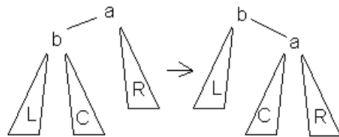


# Балансировка

## Малое левое вращение



## Малое правое вращение

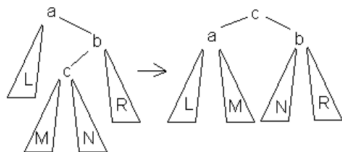


Проводится в случае, если высота  $b >$  высота  $L + 1$ , и высота  $C \leq$  высоте  $R$

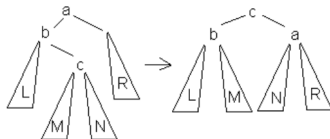
- ▶ При повороте важно не забыть обновить значения баланса
- ▶ И не запутаться в указателях

# Балансировка

Большое левое вращение



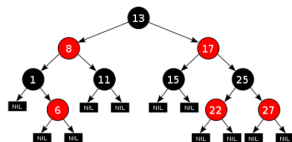
Большое правое вращение



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

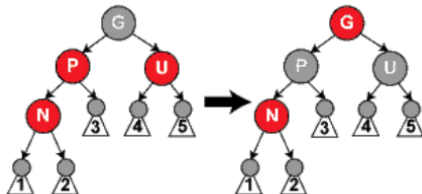
# Красно-чёрные деревья

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

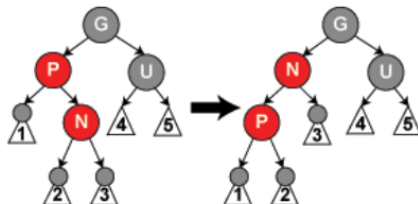
1. Добавляем в корень — ок, красим его в чёрный
2. Добавляем как сына чёрному узлу — ок, красим в красный
3. Если родитель и “дядя” красные, перекрашиваем их и добавляем наш узел как красный. Дедушка может нарушить ограничения, так что, возможно, его тоже придётся перекрасить (выполнив перекрашивание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





## Красно-чёрное дерево, добавление(2)

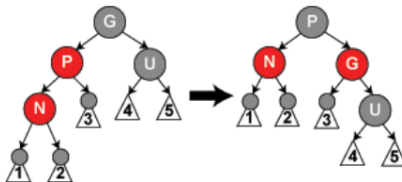
4. Родитель красный, дядя чёрный, узел справа от родителя. Выполняем поворот пары “родитель-сын”.



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

## Красно-чёрные деревья, добавление (3)

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

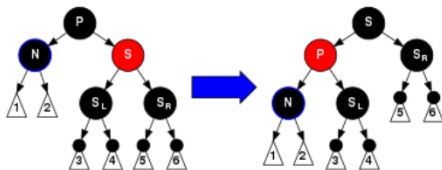
# Красно-чёрные деревья, удаление

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-чёрные деревья, удаление (2)

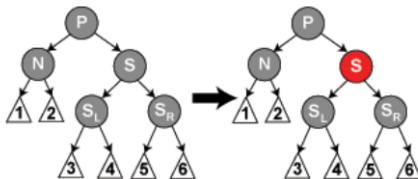
1. Самый простой случай, когда удаляемый узел корень: просто удаляем
2. У удалённого узла был красный брат: делаем поворот по ребру “отец-брат”



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

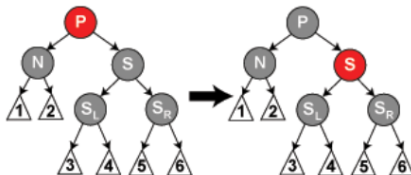
## Красно-чёрные деревья, удаление (3)

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.



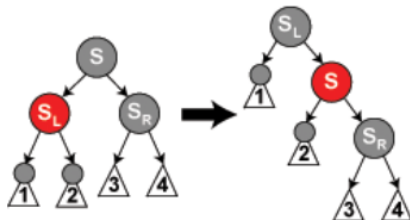
## Красно-чёрные деревья, удаление (4)

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 (5)

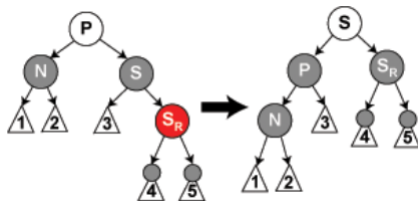
5. Левый сын брата красный, правый — чёрный. Выполняем поворот относительно брата и левого сына, одновременно перекрашивая брата и левого сына:



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

## Красно-чёрные деревья, удаление (6)

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



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

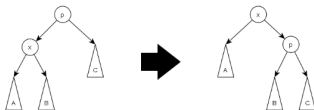


# Splay-деревья

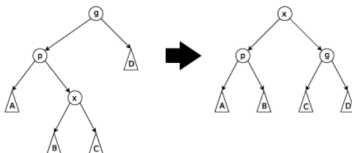
- ▶ 1985г., Д. Слитор и Р.А. Тарьян
- ▶ Продвигает узлы, к которым часто происходит обращение, ближе к корню, поэтому может быть быстрее остальных деревьев
- ▶ Не хранит дополнительных данных в узлах
- ▶ Не гарантирует сбалансированности
- ▶ Не дружит с параллельными алгоритмами
- ▶ Проще в реализации

# Splay-деревья, splaying

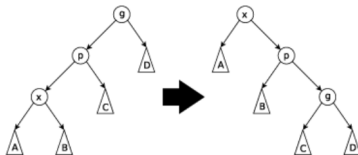
## ► Zig



## ► Zig-zag



## ► Zig-zig

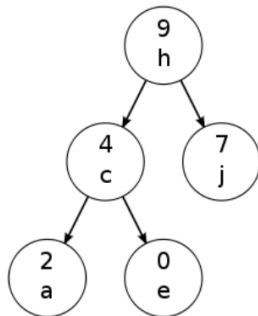


# Splay-деревья, операции

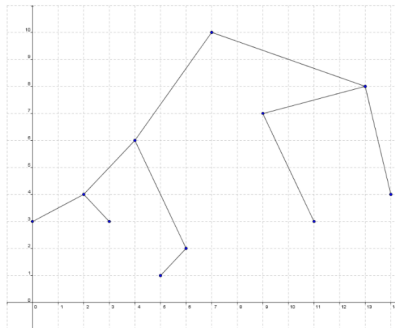
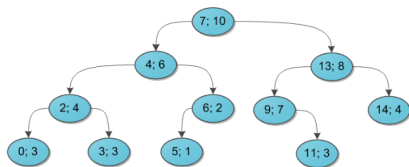
- ▶ Поиск:
  - ▶ Ищем узел как в обычном двоичном дереве поиска
  - ▶ Выполняем серию splaying-ов до тех пор, пока найденный узел не окажется корнем
- ▶ Вставка:
  - ▶ Вставляем узел как обычно в двоичное дерево поиска
  - ▶ Выполняем серию splaying-ов до тех пор, пока вставленный узел не окажется корнем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

# Декартовы деревья

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

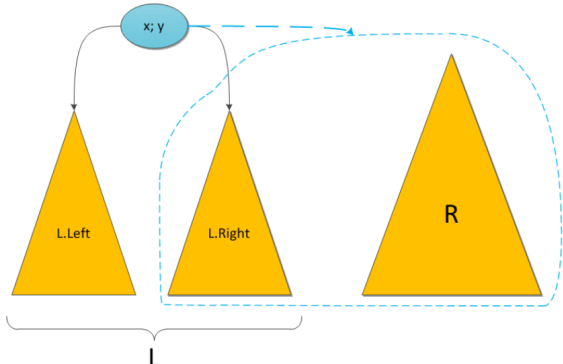


# Декартово дерево и плоскость



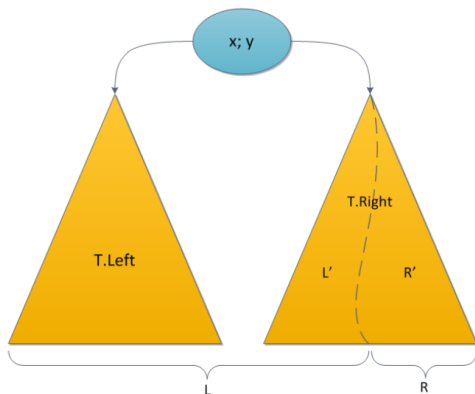
© (c) <https://habrahabr.ru/post/101818/>

# Merge

- ▶ Сливают два декартовых поддерева в одно
  - ▶ Ключи в левом поддереве должны быть меньше ключей в правом
- 
- ▶ Рекурсивно — сравниваем приоритеты вершин поддеревьев, если второе меньше, сливаем правое поддерево первого и второе, иначе наоборот

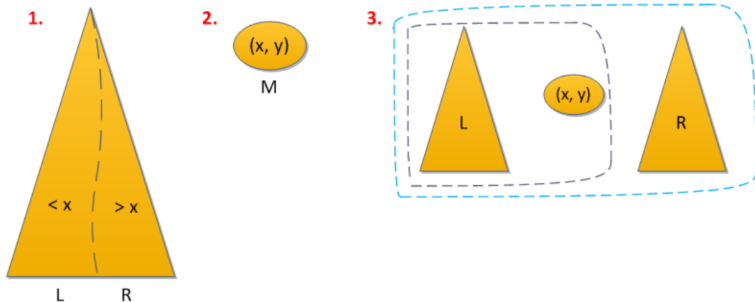
# Split

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



- ▶ Тоже рекурсивно — если ключ в корне меньше заданного, добавляем его и его левое поддерево в L, а правое поддерево — результат split от его корня

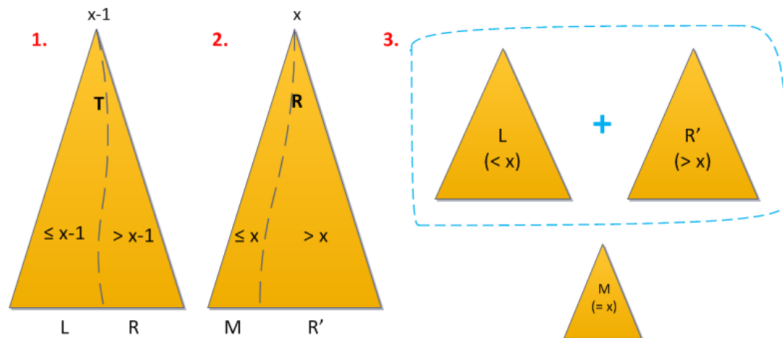
# Добавление



- ▶ Делаем split по ключу добавляемого элемента
- ▶ Делаем merge L и M
- ▶ Делаем merge того, что получилось, и R



# Удаление



- ▶ Делаем split по ключу удаляемого элемента
- ▶ Выкидываем удаляемый элемент
- ▶ Делаем merge остатков дерева