Rank-Balanced Trees

Локис Василий
КН-301

Екатеринбург, 2010

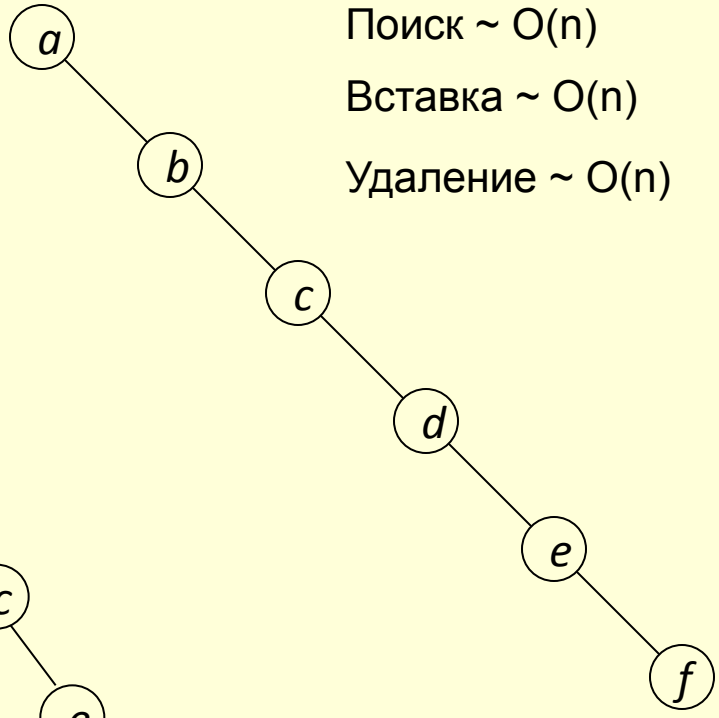
Binary Trees

Операции над элементами в деревьях:

- *поиск*
- *вставка*
- *удаление*

Binary Trees

Основная проблема в бинарных деревьях поиска (BST) – несбалансированность

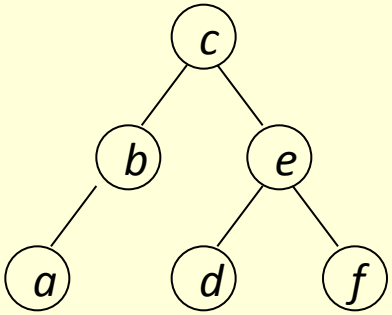


Поиск ~ O(n)

Вставка ~ O(n)

Удаление ~ O(n)

Хочется, чтобы дерево было всегда идеально сбалансированным



Поиск ~ O(log n)

Вставка ~ O(log n)

Удаление ~ O(log n)

Binary Trees

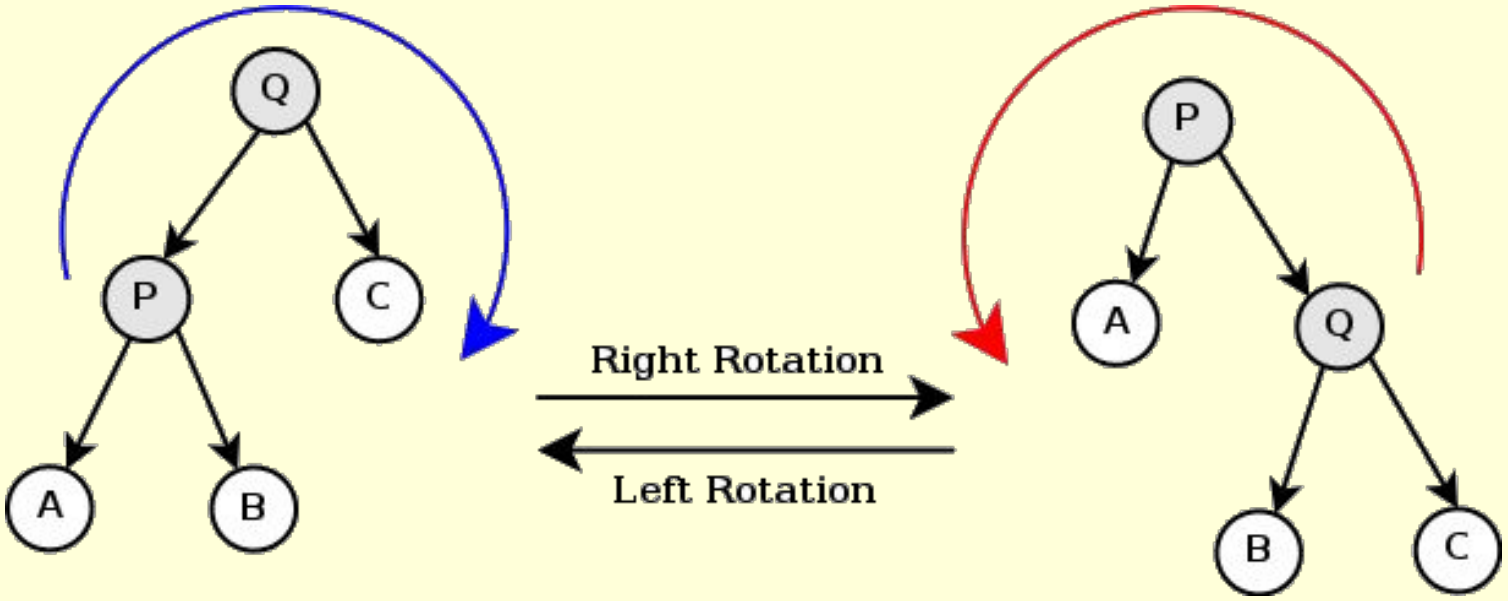
Решение:

1) Хранить в каждой вершине информацию о сбалансированности ее поддеревьев
2) После каждой вставки или удаления, если это необходимо, изменять структуру дерева так, чтобы восстановить баланс. Далее пробегать по дереву и обновлять информацию о сбалансированности

Вопрос: Как быстро и безопасно изменять структуру дерева?

Повороты

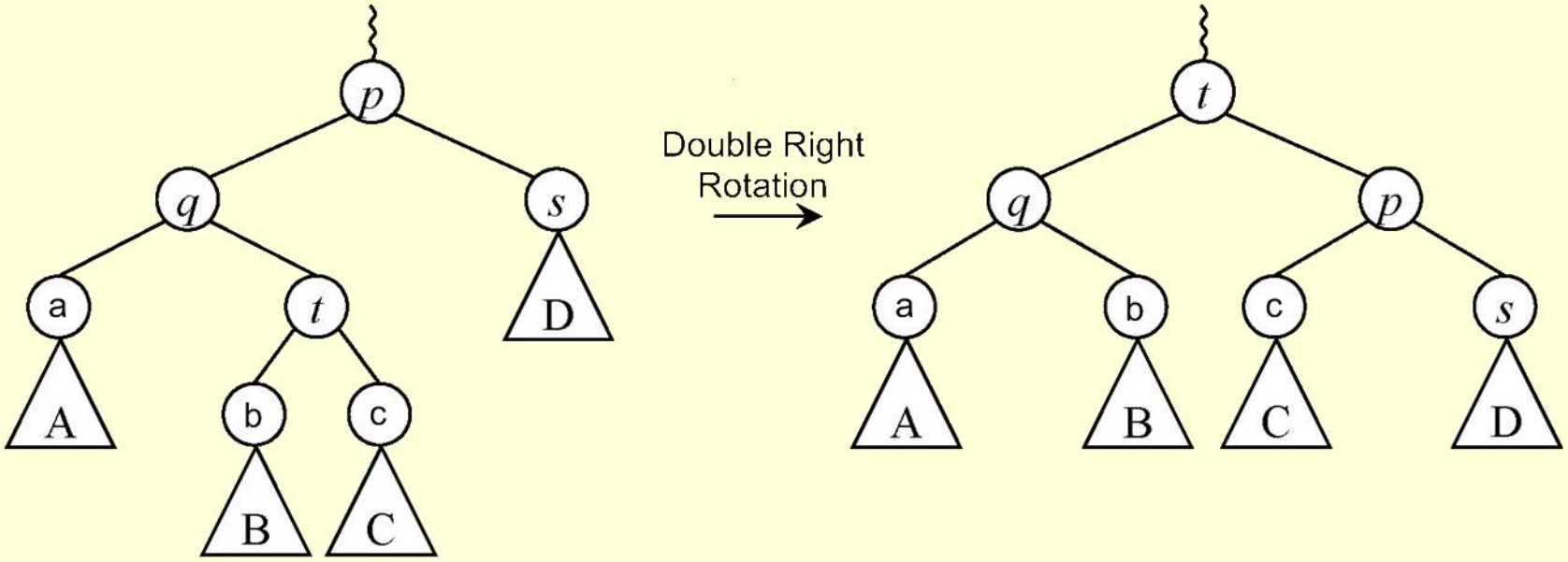
Одинарные левый и правый повороты:



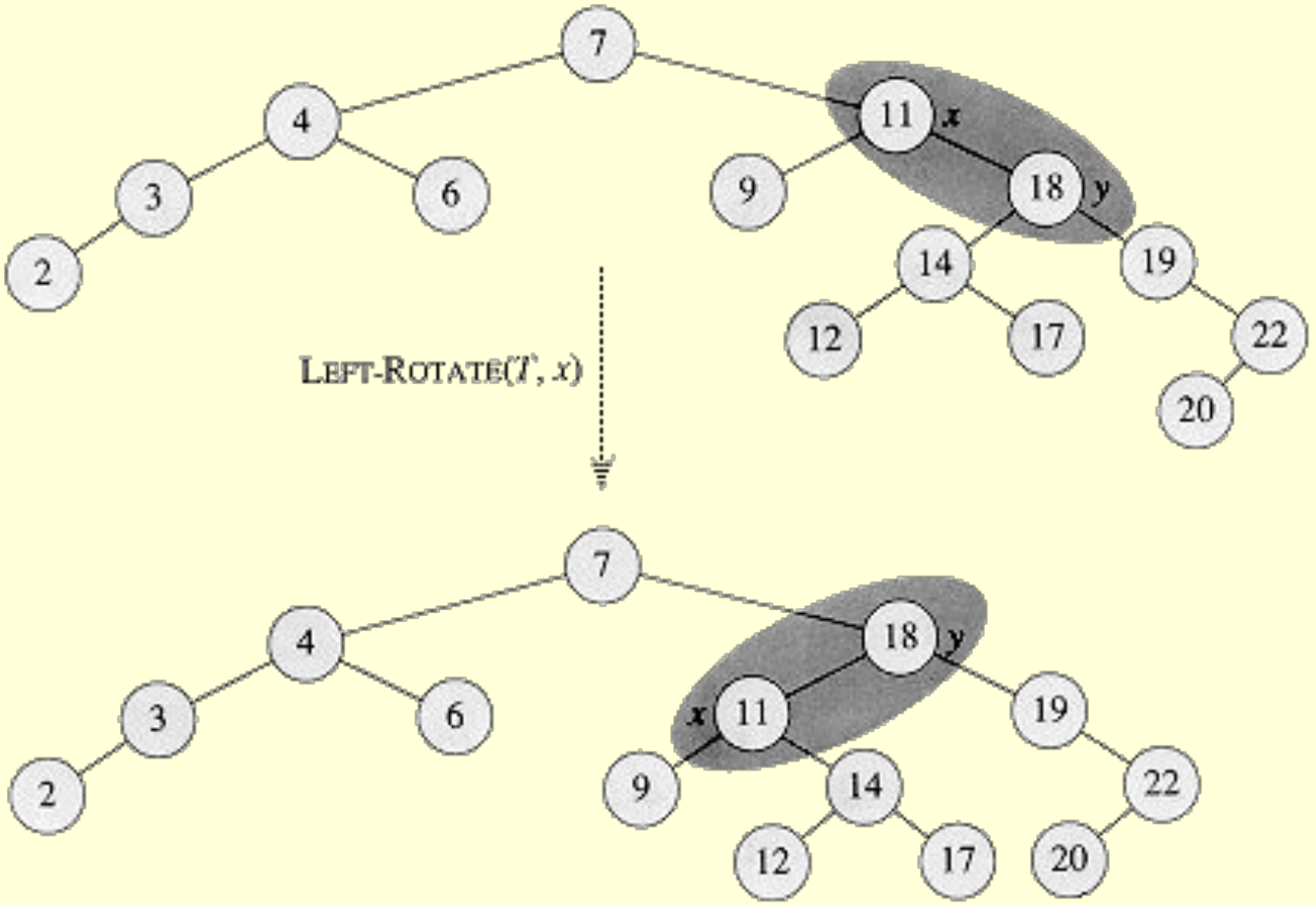
- сохраняют симметрический порядок;
- изменяют высоту;
- работают за O(1).

Повороты

Пример двойного правого поворота
(двойной левый точно так же только симметрично)



Повороты


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

Balanced Search Trees

Binary

Multiway

Red-Black trees
AVL trees
Weight Balanced trees
LLRB trees

2,3 trees
B trees

Red-Black Trees

Красно-черное дерево – самобалансирующееся бинарное дерево поиска. Сбалансированность достигается за счет введения дополнительного атрибута узла дерева — «цвет».

Если у узла нет потомков или родителя, то соответствующий указатель принимает значение NIL. Эти значения NIL мы будем рассматривать как указатели на внешние узлы (листья) дерева. Остальные узлы – внутренние.

Red-Black Trees

Красно-черное дерево обладает следующими **свойствами**:

1) Каждый узел является красным или черным.

2) Корень дерева является черным.

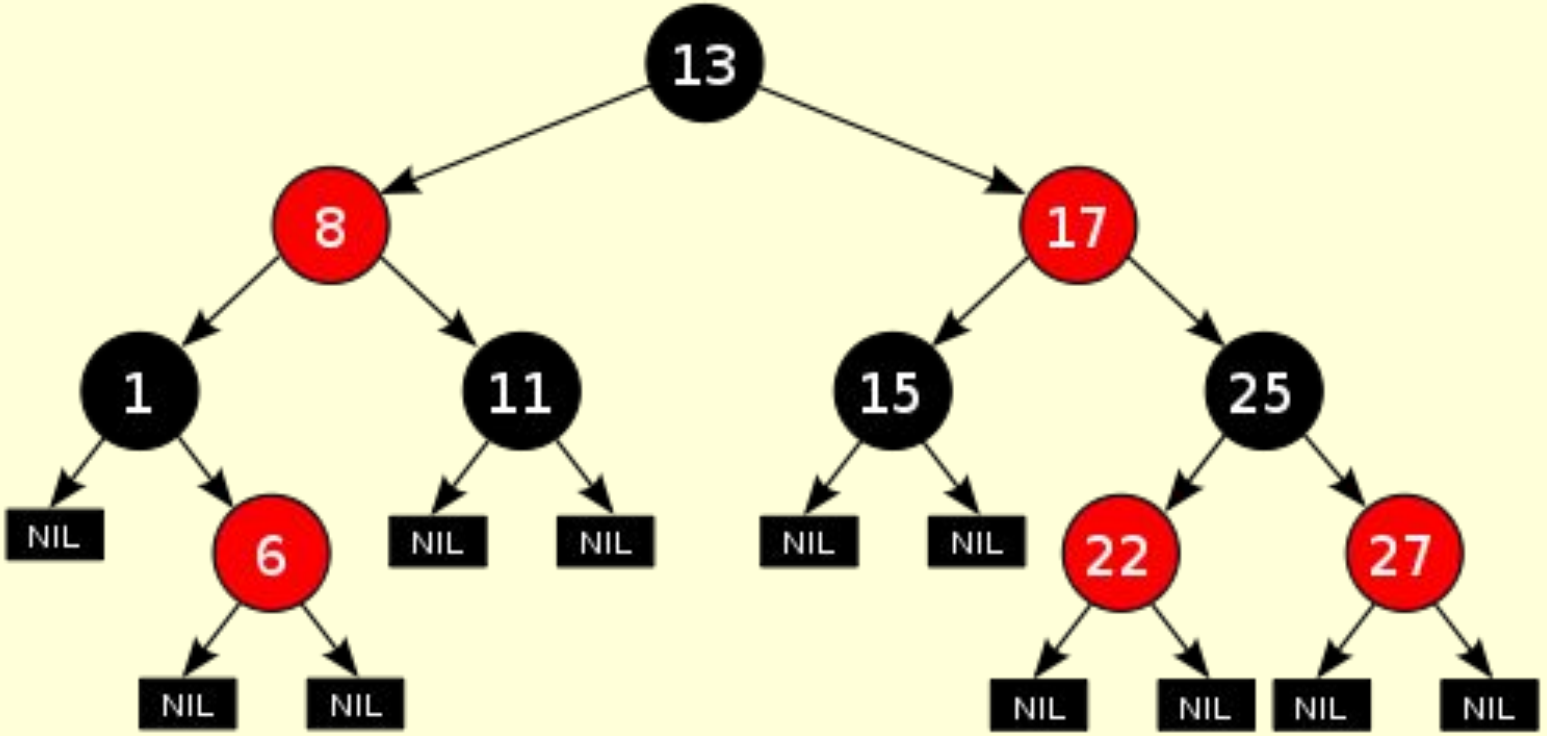
3) Каждый лист дерева (NIL) является черным.

4) Если узел — красный, то оба его дочерних узла — черные.

5) Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

Red-Black Trees

Пример красно-черного дерева:



Red-Black Trees

Если в красно-черном дереве N узлов, то можно доказать, что его высота не превосходит $2logN + 1$, следовательно, максимальная высота растёт как $O(logN)$.

Операция поиска, а также операции вставки и удаления (это мы увидим чуть позднее) выполняются за O(h), где h – высота дерева, т. е. за O(logN).

Red-Black Trees

Вставка:

Вставляем узел в дерево, как если бы это было обычное бинарное дерево поиска, а затем окрашиваем его в красный цвет.

Какие красно-черные свойства могут нарушиться при этом?

1) Каждый узел является красным или черным.
2) Корень дерева является черным.
3) Каждый лист дерева (NIL) является черным.
4) Если узел — красный, то оба его дочерних узла — черные.
5) Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

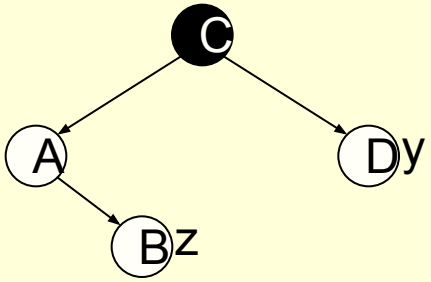
Только свойства 2 и 4. Если нарушилось свойство 2, то значит вставляемый узел – единственный в дереве, и он является корнем. Тогда просто покрасим его в **черный** цвет.

Red-Black Trees

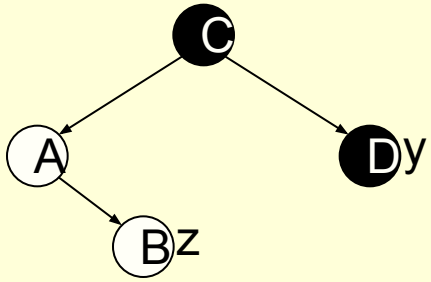
Рассмотрим ситуацию, когда нарушается свойство 4 (*Никакие два красных узла не могут непосредственно следовать друг за другом*):

Пусть вставляем узел z. Тогда родитель z – красный. Обозначим y – «дядя» z.

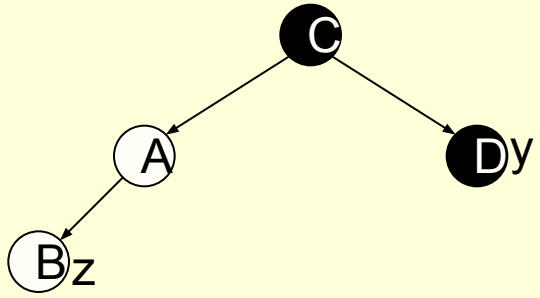
Существует 3 случая (на самом деле их 6, но они симметричны):



узел y красный.

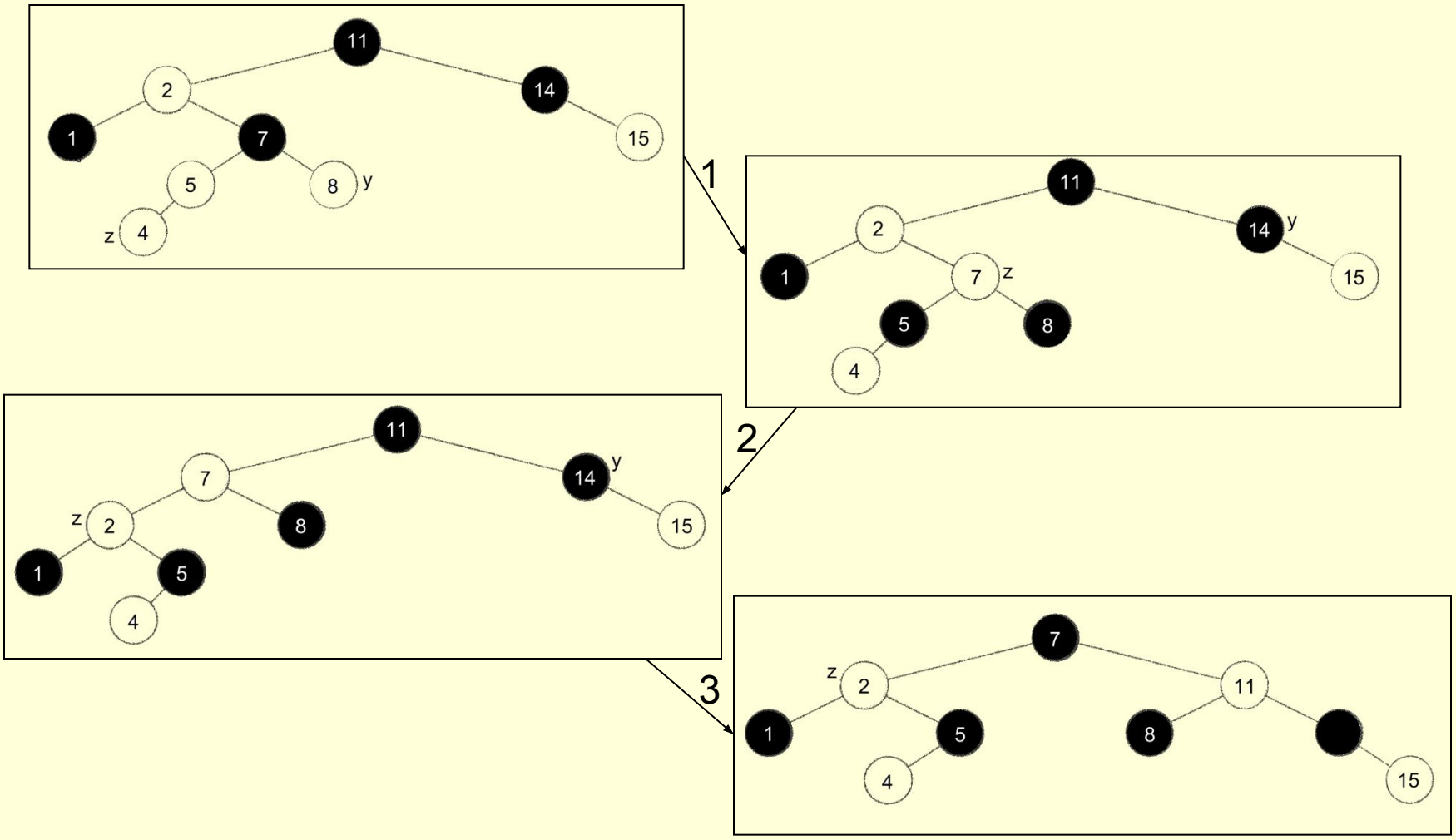


узел y черный
и z — правый



узел y черный
и z — левый

Red-Black Trees



Red-Black Trees

Анализ:

Оценим сложность алгоритма вставки:

Поиск места для вставки нового узла - O(log n)
Восстановление красно-черных свойств - O(log n):

В случаях 2 и 3 завершение работы происходит после выполнения постоянного числа изменений цвета и **не более двух поворотов**. В случае 1 указатель z перемещается вверх по дереву сразу на два уровня (т.е. не более чем O(log n) операций), и никакие повороты при этом не выполняются.

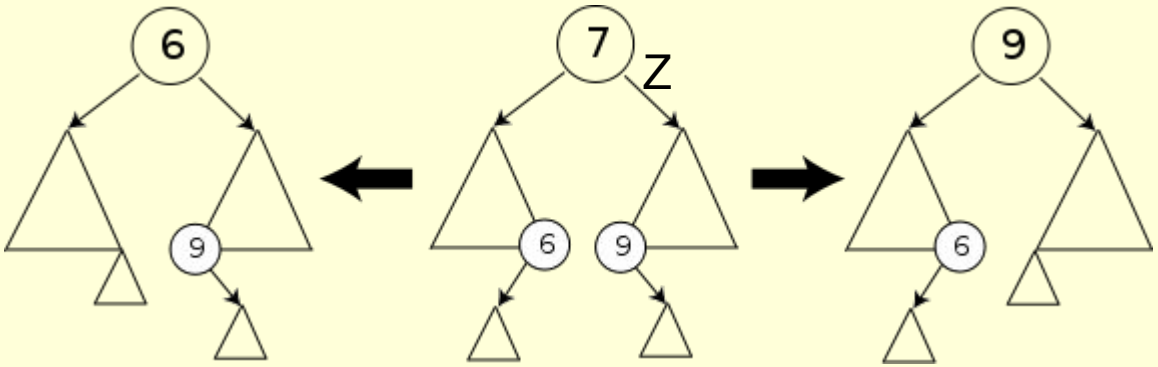
Общее время работы - O(log n).

Red-Black Trees

Удаление:

Производится так же, как в обычном бинарном дереве:

1. Если у удаляемого узла Z нет детей, то просто удаляем его.
2. Если у Z ровно один ребенок, то удаляем Z, ребенка ставим на его место, добавляя соответствующие связи.

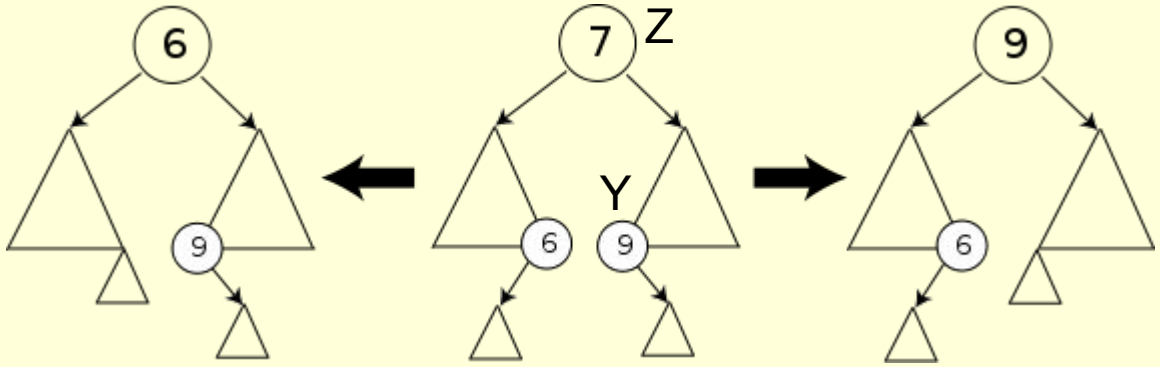


Red-Black Trees

Удаление:

3. Если у Z два ребенка, то поступаем так:

Ищем узел Y, значение которого больше (или меньше) чем значение Z, но так, чтобы между значениями Z и Y не было других значений в дереве. Другими словами, если отсортировать все элементы дерева, то узлы Z и Y будут стоять рядом.

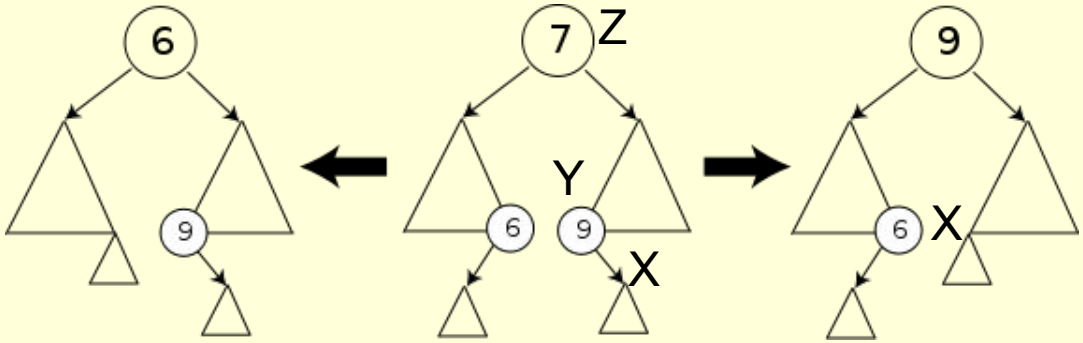


Red-Black Trees

Удаление:

3. (продолжение):

Далее заменяем значение узла Z значением найденного Y. Так как копируется только значение, то нарушения красно-черных свойств нет. Проблемы возникают дальше, когда нам нужно извлечь узел Y из дерева. Далее, когда речь пойдет о удалении узла, мы будем иметь в виду именно узел Y. Потомка Y – обозначим X.



Red-Black Trees

Удаление:

Если удаляемый узел (Y) **красного** цвета, то все хорошо. (Нарушения красно-черных свойств не происходит)

Если же он **черного** цвета, то могут возникнуть три проблемы:

- если удаляемый узел Y был корнем, а теперь корнем стал его красный потомок X, нарушается свойство 2. (Корень - черный)
- может возникнуть ситуация двух подряд идущих красных узлов (нарушение свойства 4)
- уменьшается черная высота для всего поддерева удаляемого узла (нарушение свойства 5)

Red-Black Trees

Удаление:

Попытаемся восстановить свойство 5:

Будем считать, что при удалении Y как бы отдает свою «черноту» X, тогда X – «сверхчерный». Это значит, что при прохождении через X, мы будем добавлять дополнительную 1 к количеству черных узлов.

Получается, что узел X окрашен либо "дважды черным", либо "красно-черным" цветом, что дает при подсчете черных узлов на пути, содержащем X, вклад, равный, соответственно, 2 или 1.
(Нарушается 1 красно-черное свойство)

Red-Black Trees

Удаление:

Цель: переместить дополнительную черноту вверх по дереву до тех пор, пока не выполнится одно из перечисленных условий:

1) X указывает на красно-черный узел — в этом случае мы просто делаем узел X "единожды черным".

2) X указывает на корень — в этом случае мы просто убираем излишнюю черноту.

3) Можно выполнить некоторые повороты и перекраску, после которых двойная чернота будет устранена.

Red-Black Trees

Удаление:

Пусть W – «брат» X, т.е. второй потомок отца X.

Будем рассматривать 4 возможных случая:

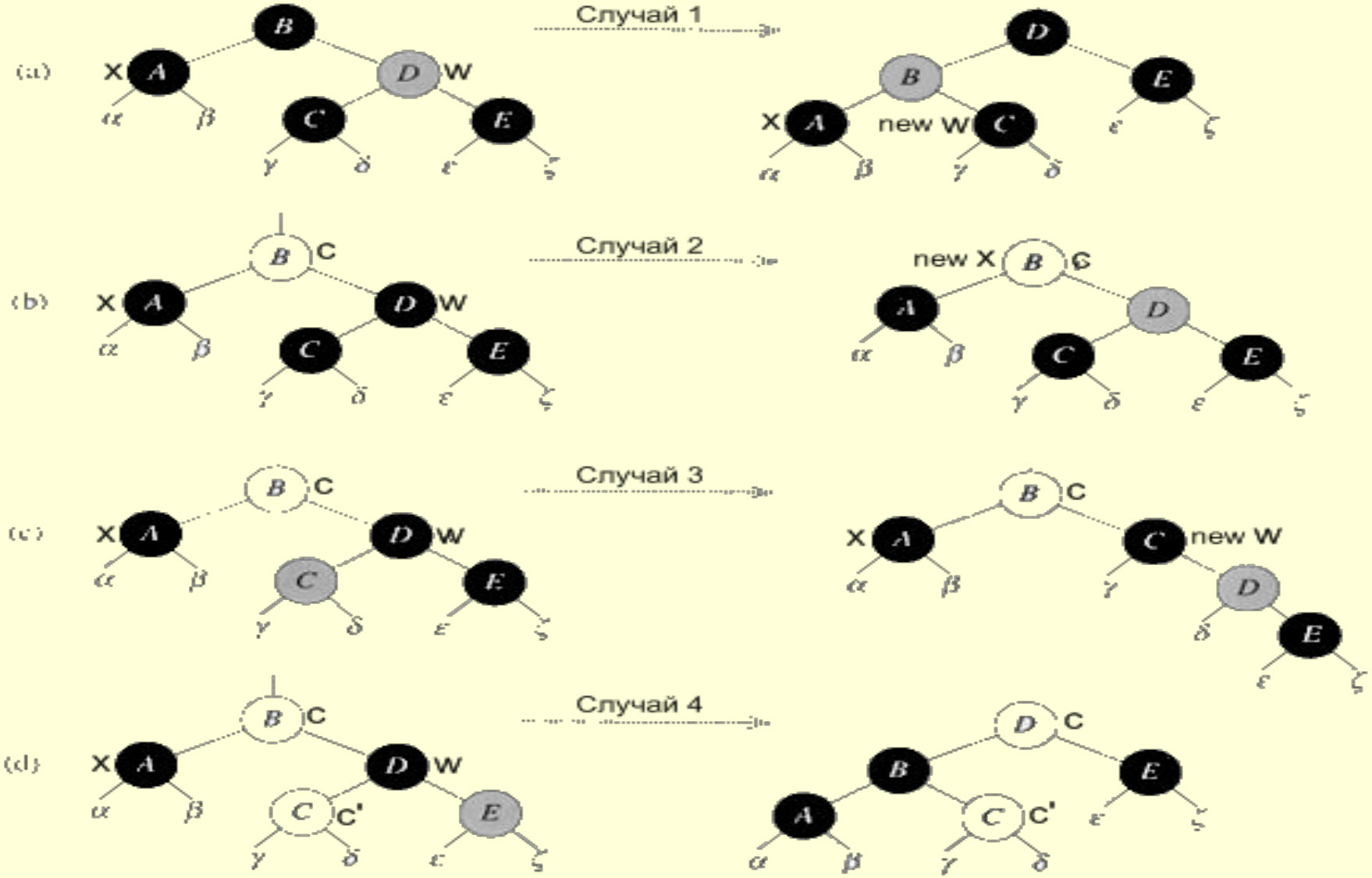
Случай 1: узел w красный.

Случай 2: узел w черный, оба его дочерних узла черные.

Случай 3: узел w черный, его левый дочерний узел красный, а правый — черный.

Случай 4: узел w черный, его правый дочерний узел красный.

Red-Black Trees



Red-Black Trees

Анализ:

Оценим сложность алгоритма удаления:

Поиск удаляемого узла Z - O(log n)
Поиск «ближайшего к нему» Y - O(log n)

Восстановление красно-черных свойств - O(log n):

В случаях 1, 3 и 4 завершение работы происходит после выполнения постоянного числа изменений цвета и **не более трех поворотов**. В случае 2 указатель X перемещается вверх по дереву не более чем O (log n) раз, и никакие повороты при этом не выполняются.

Общее время работы - O(log n).

Red-Black Trees

Где используются?

- Контейнер map в реализации библиотеки STL языка C++;
- Класс TreeMap языка Java;
- Многие другие реализации ассоциативного массива в различных библиотеках
- в ядре Linux (для организации очередей запросов, в ext3)

AVL-trees

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Высота узла :

- Высота листа равна 1. Высота нулевого указателя – 0.
- Высота внутреннего узла есть максимум из высот его поддеревьев плюс 1.

Для каждого узла хранится **коэффициент симметрии** (balance factor), имеющий три значения (-1, 0, 1) для обозначения, если высота левого поддерева <, = или > правого соответственно. При всех остальных значениях узел (а значит и все дерево) считается несбалансированным.

AVL-trees

Оценка высоты дерева:

Обозначим N_h – минимальное количество узлов в АВЛ-дереве высотой h

Можно доказать, что дерево с высотой h должно содержать как минимум Fh вершин, где Fi — i-ое число Фибоначчи. Так как $F_{k+2} > \varphi^k$, то $k \le \log_\varphi N \approx 1.44 \log N$

Таким образом, h = O(log N).

$\varphi = (1 + \sqrt{5})/2$ – золотое сечение

Операции поиска, а также вставки и удаления (про них мы еще поговорим) над деревом выполняются за O(log N).

AVL-trees

Вставка:

Заметим, что при вставке или удалении узла, высота некоторого поддерева может измениться максимум на 1. Следовательно, если свойство дерева нарушается, то это возможно только если коэффициент симметрии равен 2 (или -2).

Чтобы восстановить свойство АВЛ-дерева воспользуемся поворотами.

AVL-trees

Вставка:

- Вставим новый узел в качестве листа, как это делается в обычном бинарном дереве
- Будем рассматривать все вершины на пути от нового листа к корню дерева. Будем проверять, верно ли, что высоты левого и правого поддеревьев отличаются не больше чем на 1.
- Если да, то перейдем к предку рассматриваемого узла. Иначе, восстановим свойство, применяя либо одинарный, либо двойной поворот
- Известно, что для вставки требуется максимум один поворот, т.е. если сделали поворот, то дальше можно не подниматься по дереву.

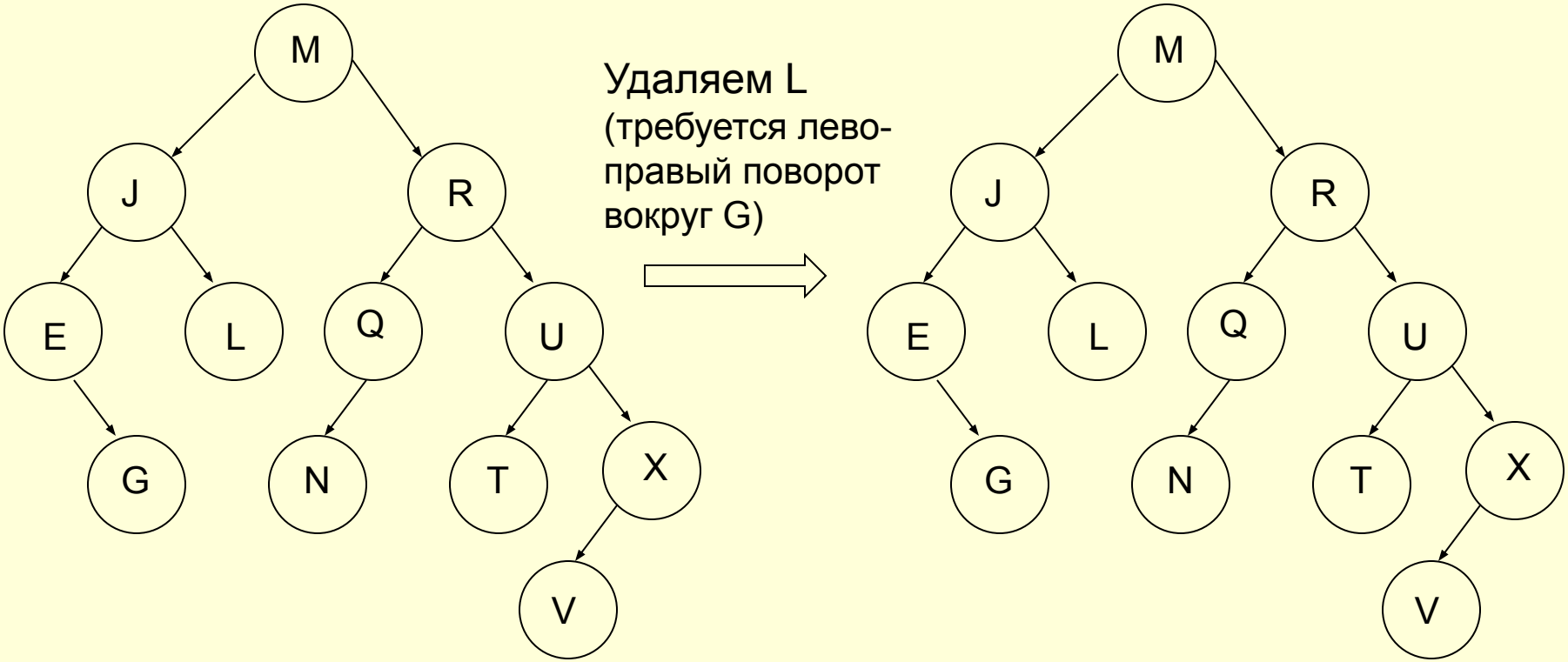
AVL-trees

Удаление:

- Удаляем узел как это делается в обычном бинарном дереве.
- Будем рассматривать все вершины на пути от удаляемого листа к корню дерева. Будем проверять, верно ли, что высоты левого и правого поддеревьев отличаются не больше чем на 1.
- Если да, то перейдем к предку рассматриваемого узла. Иначе, восстановим свойство, применяя либо одинарный, либо двойной поворот
- При удалении может потребоваться больше чем один поворот, следовательно продолжаем, пока не достигнем корня дерева.

AVL-trees

Пример, когда при удалении может потребоваться не один поворот:



После восстановления J, узел M все еще остается несбалансированным

AVL-trees

Наглядная демонстрация работы АВЛ-дерева:

http://www.strille.net/works/media_technology_projects/avl-tree_2001/

http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

http://www.cs.jhu.edu/~goodrich/dsa/trees/avltree.html

AVL-trees

Эффективность

Г.М.Адельсон-Вельский и Е.М.Ландис доказали теорему, согласно которой высота АВЛ-дерева с N внутренними вершинами заключена между log(N+1) и 1.4404*log(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*log(N). Таким образом, выполнение базовых операций требует порядка log(N) сравнений.

Экспериментально выяснено, что одна балансировка приходится на каждые два включения и на каждые пять исключений.

AVL-trees

Где используются?

- АВЛ-деревья используются, когда нужен быстрый доступ к элементам дерева
- Могут использоваться для сортировки данных

AVL vs Red-Black

Сравнение красно-черных и АВЛ-деревьев:

И у тех, и у других все базовые операции над бинарными деревьями имеют сложность – O(log N)

В худшем случае высота составляет:

АВЛ-деревья - 1.44 * log(N+2) - 0.33

Красно-черные деревья - 2 * log(N+1)

Следовательно, при поиске АВЛ-деревья работают быстрее красно-черных

При вставке красно-черные деревья выполняют балансировку за O(1), АВЛ же за O(log N)

Касательно удаления, здесь также выигрывают красно-черные, так как им потребуется O(1) (максимум 3 поворота), тогда как для АВЛ может потребоваться O(log N)

AVL vs Red-Black

Сравнение красно-черных и АВЛ-деревьев:

Таким образом,

АВЛ-деревья рекомендуется использовать, когда хочется быстрого поиска элемента в фиксированных данных

Если же данные динамические, т.е. много операций вставки и удаления, то лучше использовать *красно-черные деревья*

Rank-Balanced Trees

Вывод:

Хочется совместить преимущества АВЛ и красно-черных деревьев:

- Маленькая высота
- Малое количество балансировок
- Простой алгоритм

Ranks

Каждый узел имеет **ранг** – целое число, оценивающее высоту поддерева данного узла

Будем считать, что листья имеют ранг 0, Отсутствующие узлы имеют ранг -1

Ранг-разность для ребенка - это разность между рангом родителя и рангом ребенка

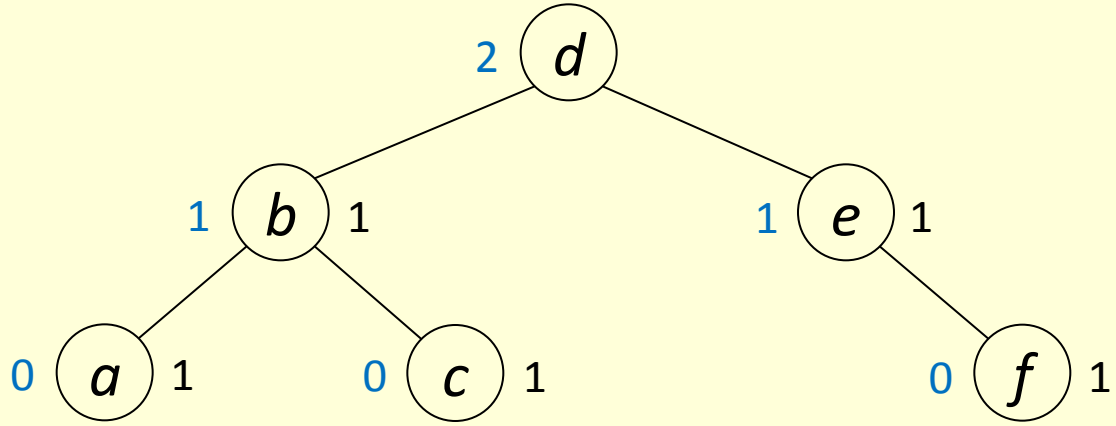
i-ребенок: узел с ранг-разностью i

i,j-узел: дети этого узла имеют ранги i и j

Рангом дерева будем называть ранг его корня

Ranked Binary Trees

Пример бинарного дерева с рангом:



Синие цифры – ранг

Черные – ранг-разность

Ranked Binary Trees

АВЛ-деревья: каждый узел либо 1,1-, либо 1,2-

Красно-черные: все ранг-разности либо 0, либо 1, никакой 0-ребенок не может быть отцом другого 0-ребенка

RB-деревья: каждый узел 1,1-, 1,2-, или 2,2-узел (ранг-разность либо 1, либо 2)

В каждом случае требуется дополнительный бит сбалансированности

Оценка высот

n_k = минимальное n для ранга k

АВЛ-деревья:

$n_0 = 1, n_1 = 2, n_k = n_{k-1} + n_{k-2} + 1$

$n_k = F_{k+3} - 1 \Rightarrow k \le \log_\varphi n \approx 1.44 \log n$

RB-деревья:

$n_0 = 1, n_1 = 2, n_k = 2n_{k-2}$,

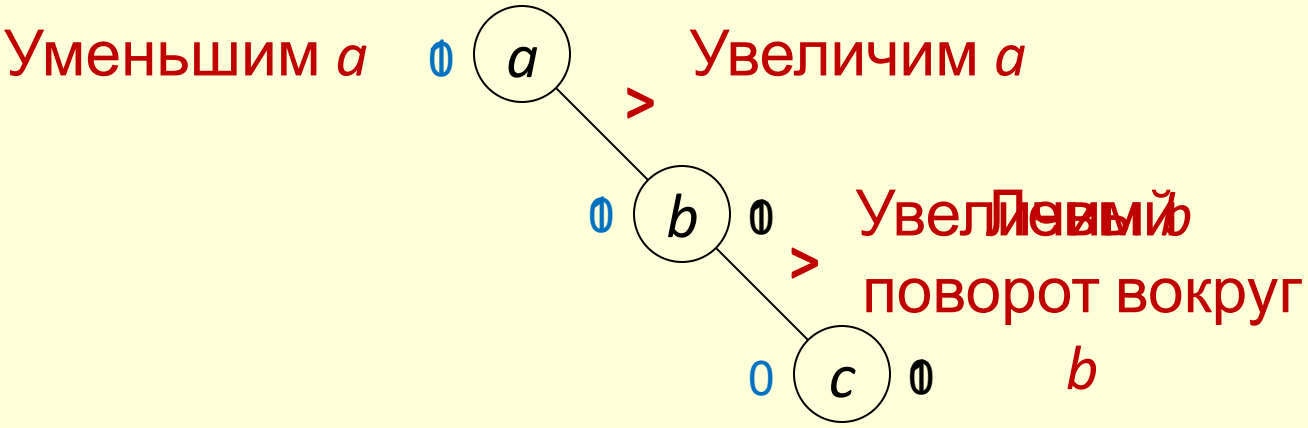
$n_k = 2^{\lceil k/2 \rceil} \Rightarrow k \le 2 \log n$

F_k = k-ое число Фибоначчи
φ – золотое сечение
$F_{k+2} > \varphi^k$

Та же высота и для красно-черных деревьев

Rank-Balanced Trees

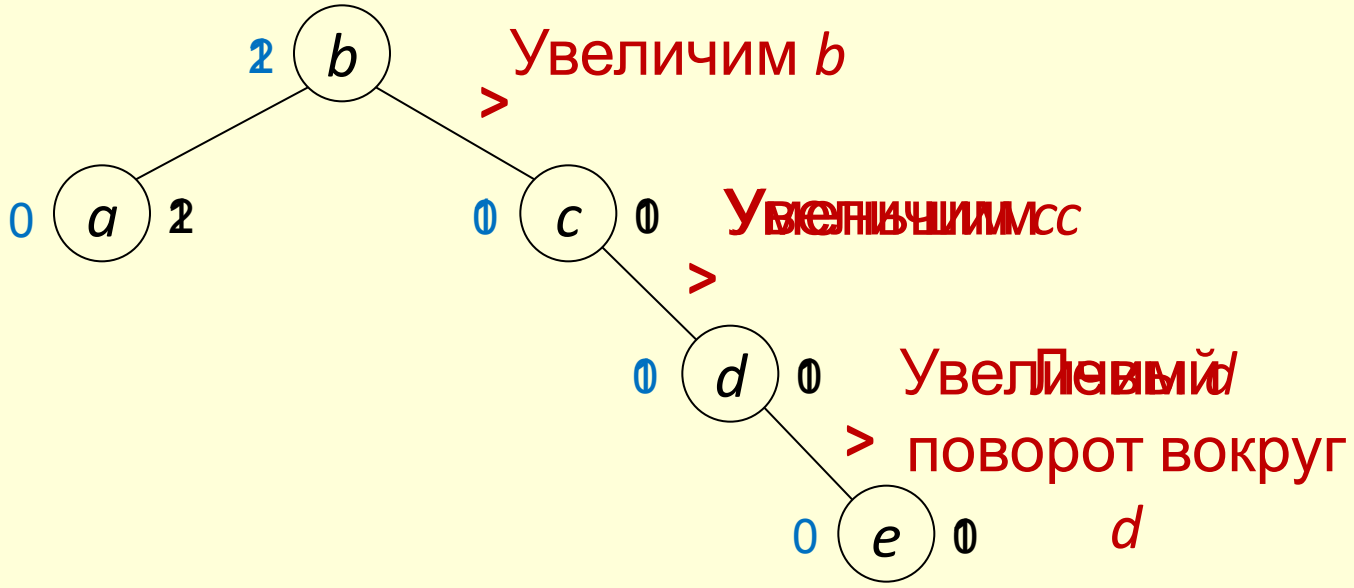
Вставка:



Вставим *b*

Rank-Balanced Trees

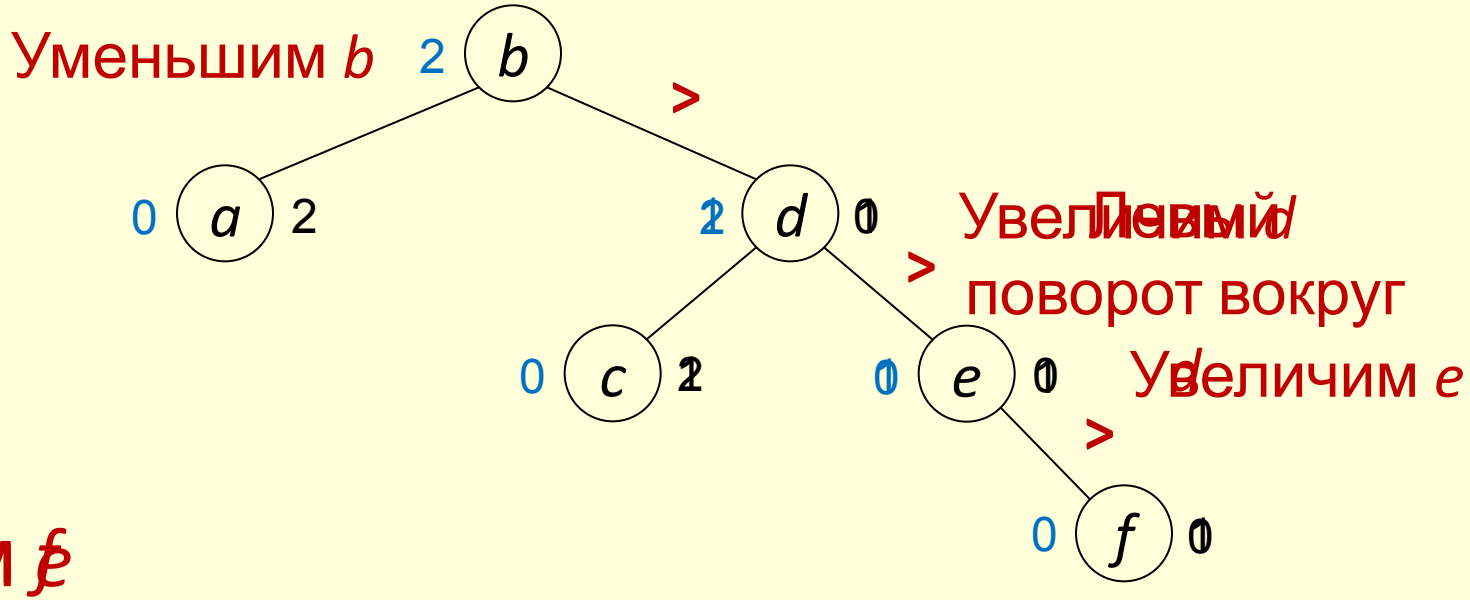
Вставка:



Вставим *e*

Rank-Balanced Trees

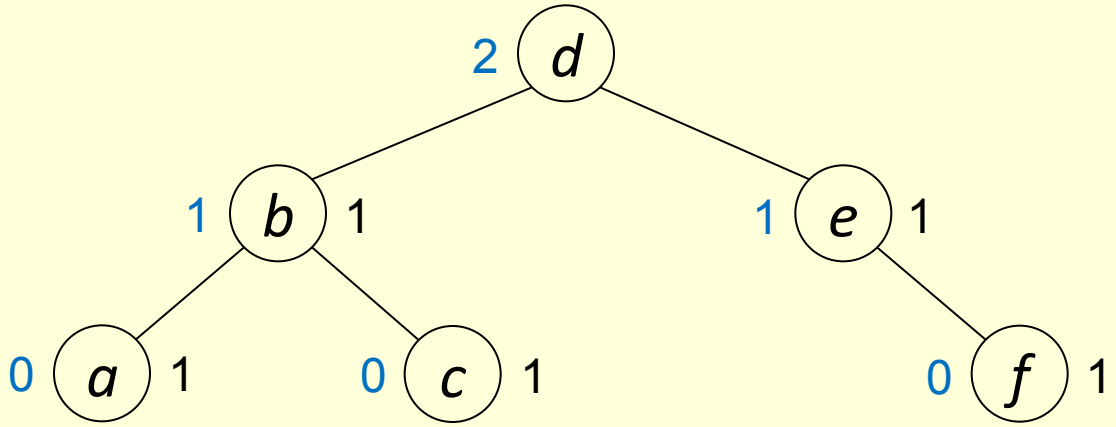
Вставка:



Вставим f

Rank-Balanced Trees

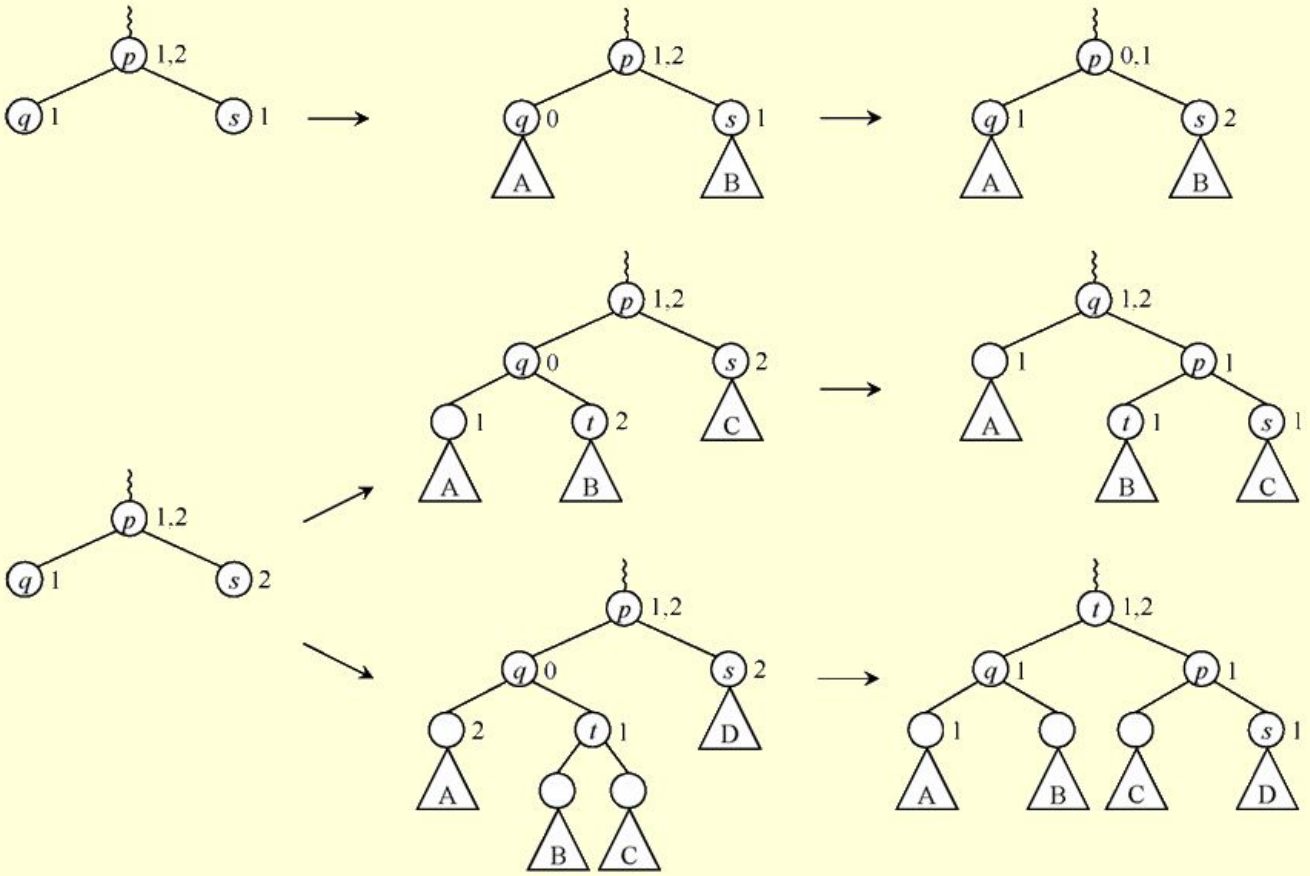
Вставка:



Вставим *f*

Rank-Balanced Trees

Балансировка при вставке:



Все числа здесь – ранг-разности. Так как нет 2,2-узлов, то ведет себя в точности как АВЛ-дерево. Выполнится не более двух поворотов.

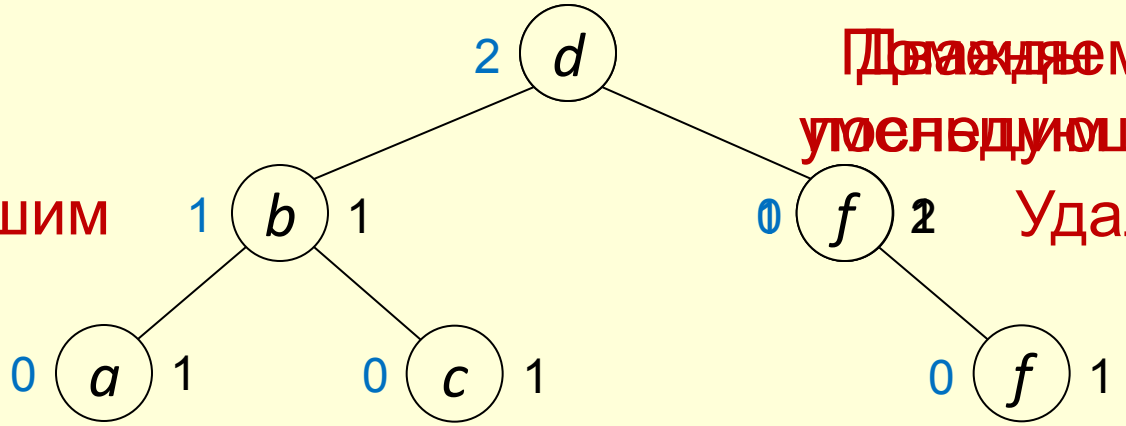
Rank-Balanced Trees

Удаление:

Если узел *q* встает на место удаляемой вершины, а *p* ее родитель, то нарушение происходит, если *p* – лист ранга 1 или если *q* – 3-ребенок.

Rank-Balanced Trees

Удаление:



Уменьшим *b*

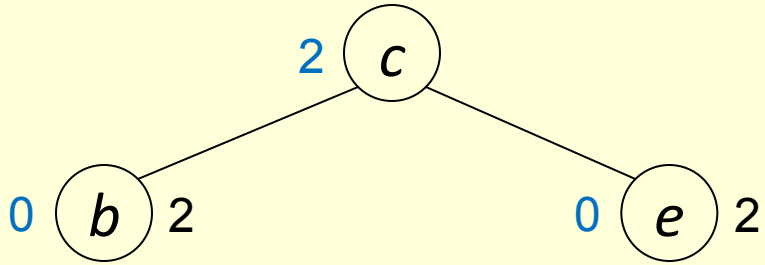
Удалим *f*

Двойной поворот вокруг *c*

Дважды увеличим *c*

Rank-Balanced Trees

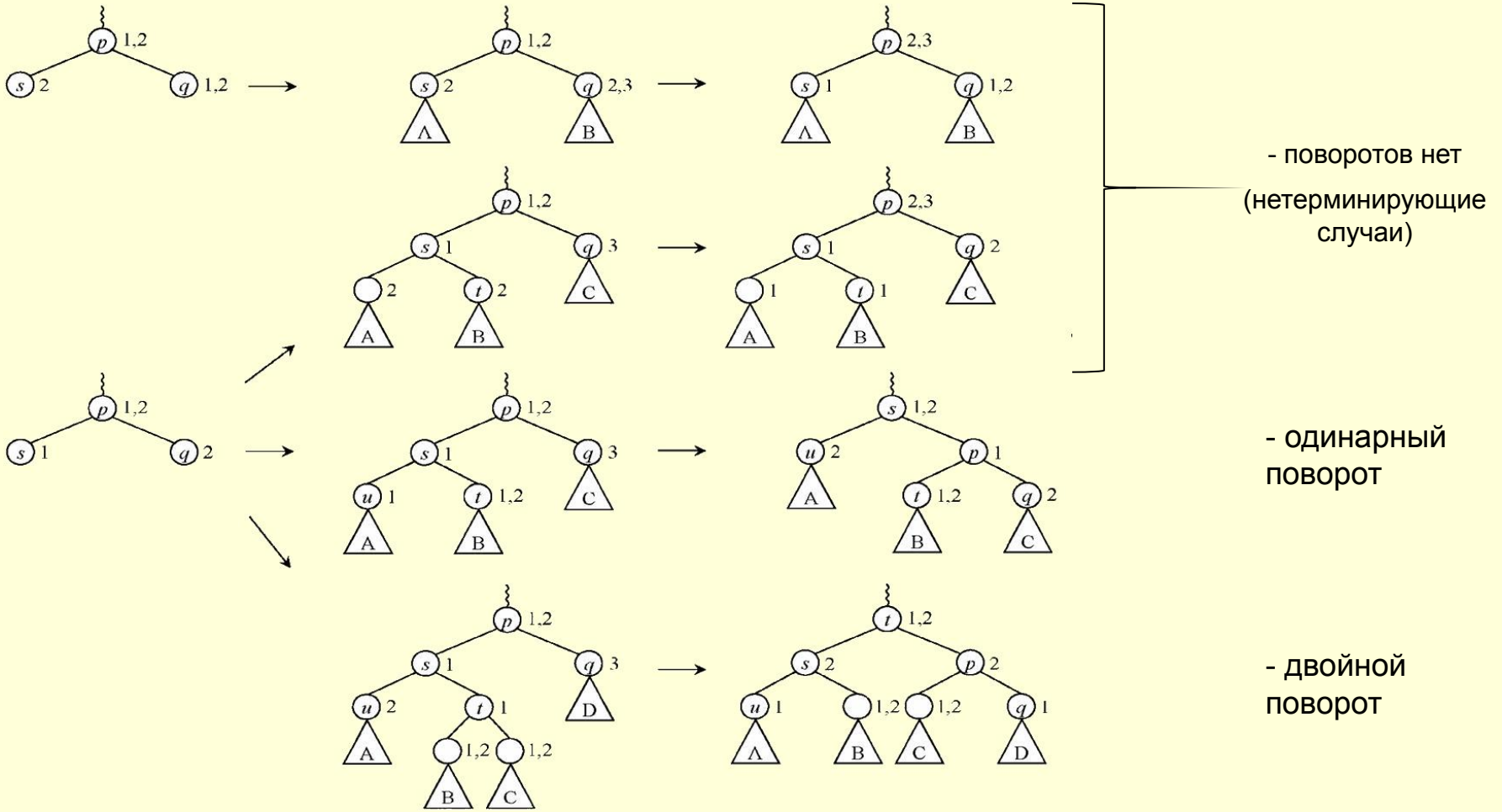
Удаление:



Удалим *f*

Rank-Balanced Trees

Балансировка при удалении:



Выполнится не более двух поворотов!

Rank-Balanced Trees

Существует теорема, что высота сбалансированного по рангу дерева не превосходит $\log_{\varphi} m$, где m – количество вставок в дерево. (d – количество удалений)

Общая высота RB-дерева – $\min\{2\log n, \log_{\varphi} m\}$

В сравнении с другими деревьями:

Если m=n, то в точности как АВЛ-дерево, т.е. $O(\log_{\varphi} m) \sim O(\log N)$

Если d и m примерно равны, то высота приближается к $O(2\log N)$ – как красно-черное дерево

Rank-Balanced Trees

Таким образом, RB-деревья выполняют меньше поворотов, чем красно-черные деревья, а также достигают лучшей оценки высоты дерева (приближается к высоте АВЛ-дерева)

Совместили преимущества АВЛ и красно-черных деревьев

Ссылки

- Много примеров взято из работ Siddhartha Sen, Bernhard Haeupler, Robert E. Tarjan (Princeton University)
- http://en.wikipedia.org/wiki/AVL_tree
- http://en.wikipedia.org/wiki/Red-black_tree
- http://www.strille.net/works/media_technology_projects/avl-tree_2001/

И другие


Спасибо
за внимание