Сбалансированные деревья

Дискретный анализ 2012/13

Андрей Калинин, Татьяна Романова

17 сентября 2012 г.

Литература

- Д. Кнут. Искусство программирования, том 3, Сортировка и поиск, 2-е издание, М.:Вильямс, 2003, стр. 492–509, п. 6.2.3. «Сбалансированные деревья».
- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.. Алгоритмы: построение и анализ, 2-е издание, М.:Вильямс, 2005, стр. 336-359, глава 13, «Красно-черные деревья».

Поиск

- Различные подходы

AVL-деревья

- Определение, высота дерева
- Вставка
- Удаление

Красно-черные деревья

- Определение, высота дерева
- Вставка
- Удаление

Раздел

Поиск

- Различные подходы

AVL-деревья

- Определение, высота дерева
- Вставка
- Удаление

Красно-черные деревья

- Определение, высота дерева
- Вставка
- Удаление

Где, что и как ищем

- Храним пары ключ-значение. Хотим найти значения или просто факт наличия для ...
  - Точных значений ключей.
  - Ключей в интервалах.
  - Префиксов ключей.
  - Ключей с подстроками.
  - Ключей, удовлетворяющих маскам.
- Какова скорость доступа к данным?
  - Равномерный (случайный) доступ.
  - Скорость неравномерна, есть локальность доступа.
- Как часто ...
  - Ищем?
  - Добавляем?
  - Изменяем?
  - Удаляем?

Поиск

- Основные алгоритмы:
  - Последовательный поиск — $O(n)$.
  - Бинарный поиск в статическом отсортированном массиве — $O(\log n)$.
  - Поиск в сбалансированных и сильноветвящихся деревьях — $O(\log n)$.
  - Поиск с использованием хеширования — $O(1)$ при наличии «хорошей» хеш-функции.
  - Цифровой поиск. Время работы $O(|key|)$.
- Специализированные алгоритмы:
  - Оптимальные деревья поиска.
  - String B-tree.
  - Суффиксные деревья.

Дерево поиска

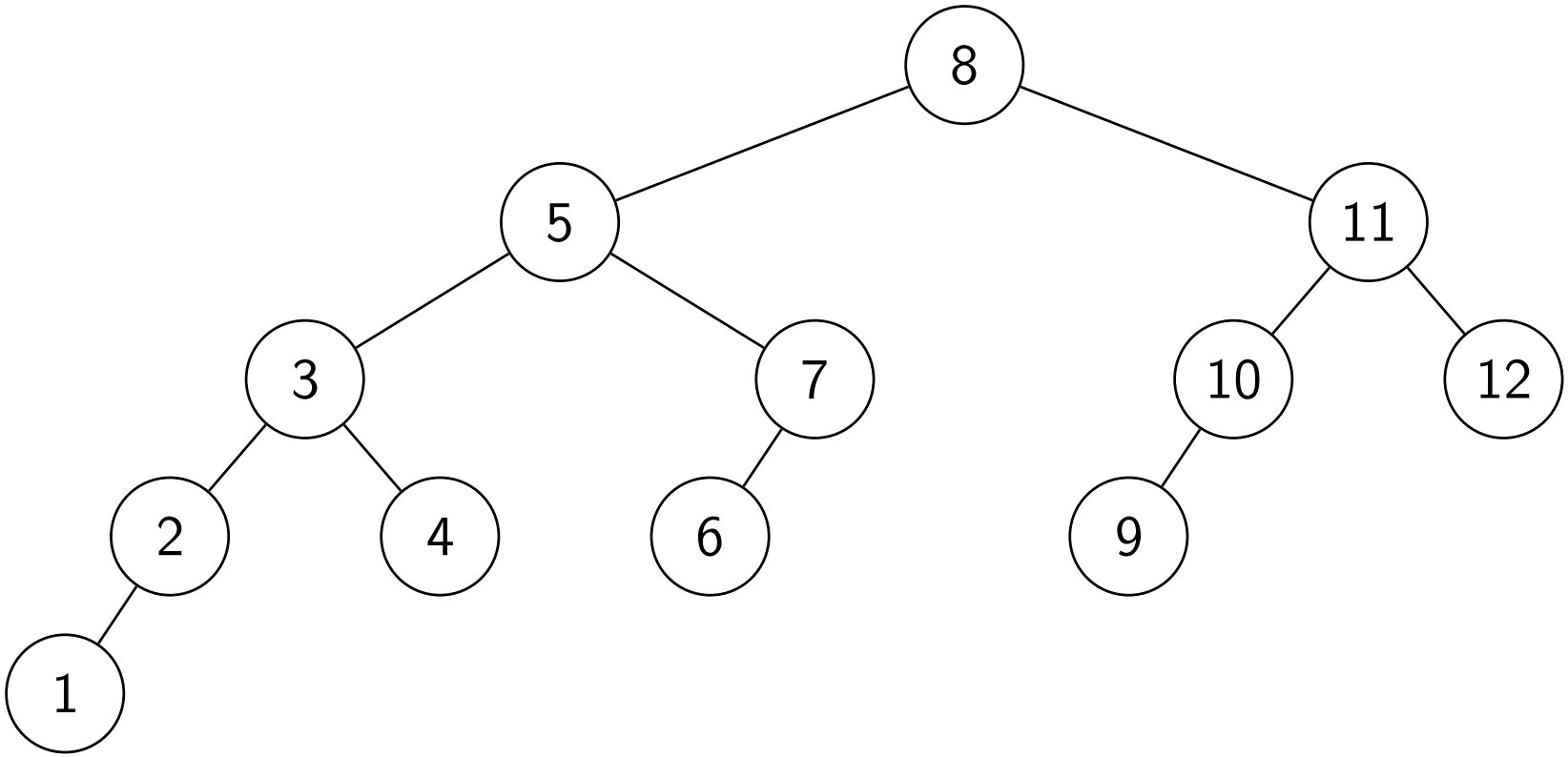
- Левое поддерево содержит меньшие ключи, правое — большие.
- Может выродиться в линейный список.
- Идеально сбалансированное дерево поиска — дерево наименьшей высоты ($\log_2 n$).
- Сбалансированное дерево — дерево, для которого выполняется какое-то условие баланса.

Раздел

AVL-деревья

- Обычное дерево поиска при «плохих» данных вырождается в линейный список.
- Высота дерева — длина самого длинного пути от корня к листу.
- AVL-дерево — это то, у которого высота левого поддерева любого узла отличается от высоты правого не более, чем на 1.
- Удаление, добавление и поиск элемента в AVL-дереве производится за $O(\log n)$.

Минимальное AVL-дерево



Высота AVL-дерева

Теорема

Высота AVL-дерева есть $O(\log n)$, где n — количество узлов в дереве.

Доказательство.

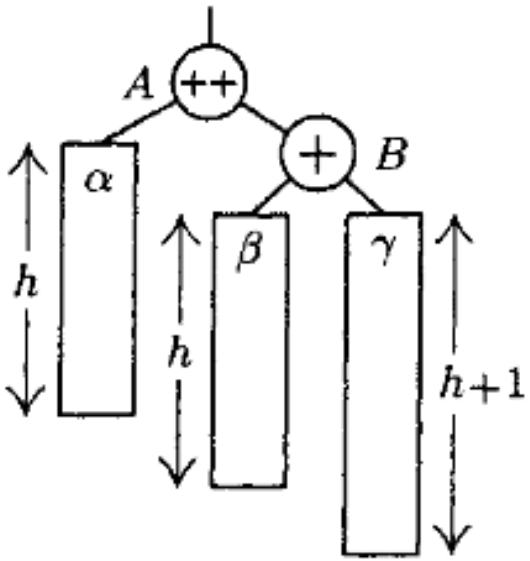
Пусть T_h — минимальное AVL-дерево. Высота одного из его поддеревьев (например, левого) равна $h-1$, другого — $h-2$. Высоты поддеревьев T_{h-1} будут равны $h-2$ и $h-3$ и т. п.
$n_{min} = 1 + 1 + 2 + \cdots + F_h$, где F_h — h-е число Фибоначчи.
$n \geq \sum_{i=1}^{h} F_i = F_{i+2} - 1 \sim \phi^{h+2}/\sqrt{5}$.
$h = O(\log n)$. □

Раздел

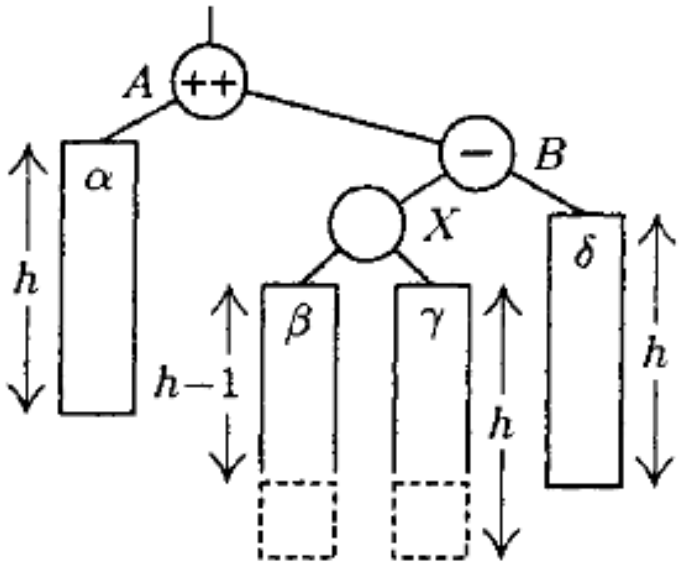
Вставка в AVL-дерево

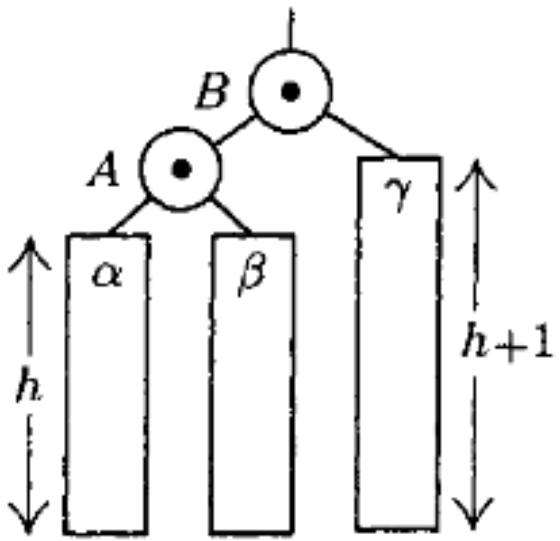
1. Поиск ключа в дереве.
2. Вставка нового ключа.
3. Перебалансировка дерева при необходимости:
   - 3.1 Перевычисляем баланс по пути наверх.
   - 3.2 Если баланс нарушен, выполняем один из поворотов.

Балансировка при вставке

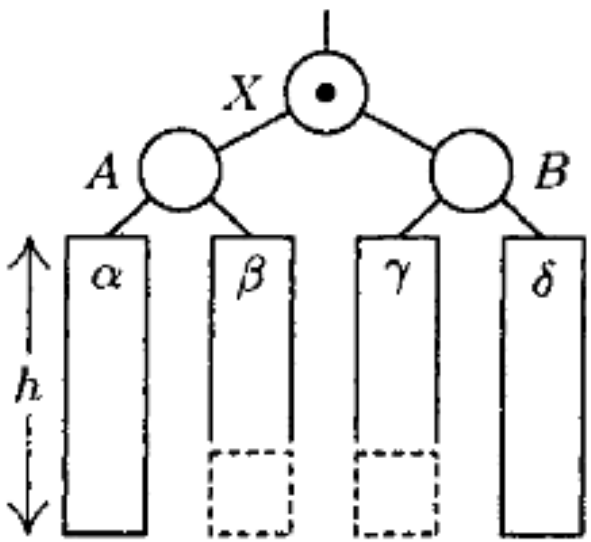


Случай 2





Случай 2



Раздел

Удаление из AVL-дерева

- Найти узел X с ключом, который нужно удалить.
- Если $X.r = NIL$ или $X.l = NIL$, то $R = X$, иначе найти узел Y, следующий по порядку за узлом X (самый левый в правом поддереве) и заместить ключ в X ключом в Y и $R = Y$.
  $P_0 = head, D_0 = +1, link(D_i, P_i) = P_{i+1}$,
- При поиске узла R класть в стек пары (P_k, D_k), $k = 0 \dots l$.
  $P_l = R, link(D_l, P_l) = NIL$.
- $link(D_{l-1}, P_{l-1}) = link(-D_l, P_l)$.
- Скорректировать фактор сбалансированности P_{l-1}.

Корректировка баланса P_k

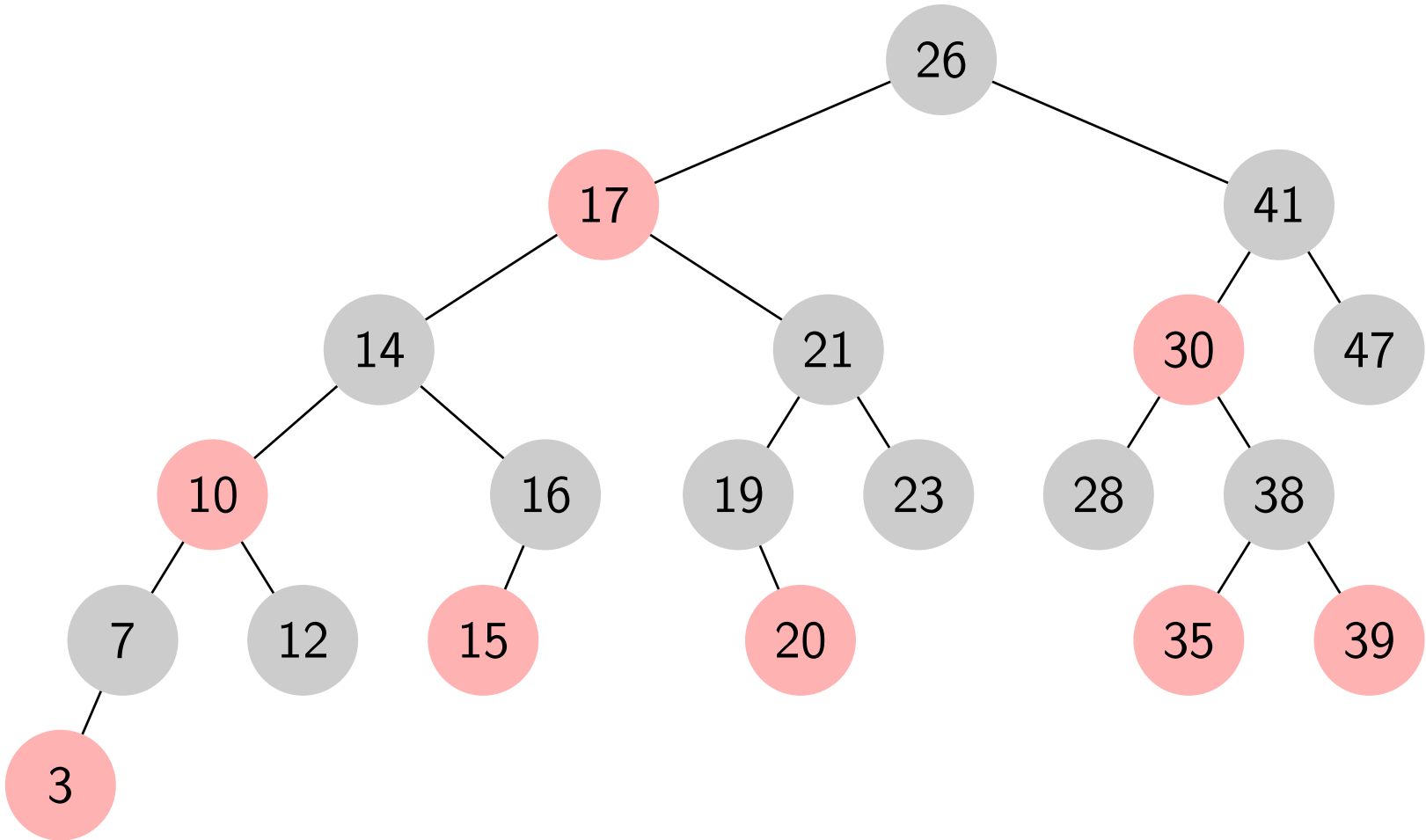
- Если $k = 0$, завершить выполнение алгоритма.
- $bal(P_k) = D_k \Longrightarrow bal(P_k) \leftarrow 0, k \leftarrow k - 1$, повторить корректировку для нового k.
- $bal(P_k) = 0 \Longrightarrow bal(P_k) \leftarrow -D_k$, завершить выполнение алгоритма.
- $bal(P_k) = -D_k \Longrightarrow$ требуется перебалансировка:
  - Аналогично добавлению: рассматриваем поддерево, в котором не удаляли элемент, как дерево с добавленным элементом.
  - Добавляется третий случай, аналогичный первому но с высотой β равной $h + 1$.
- Важное отличие от добавления элемента: может потребоваться $\log n$ поворотов, тогда как для добавления не требуется больше 1.

Раздел

Красно-черные деревья

1. Каждый узел является либо красным либо черным.
2. Корень дерева является черным.
3. Каждый лист дерева (NIL) является черным.
4. Если узел дерева красный, то оба дочерних узла — черные.
5. Для каждого узла все пути от него до листьев-потомков содержат одинаковое количество черных узлов.

Пример



Высота красно-черного дерева

Теорема

Красно-чёрное дерево с n внутренними узлами имеет высоту не более чем $2\log_2(n+1)$.

Доказательство.

1. $bh(x)$ — черная высота узла x (количество черных узлов на пути от x к листу, не считая x).
2. Максимальная и минимальная высоты поддеревьев отличаются не больше, чем в 2 раза (из свойств 4 и 5).
3. Минимальное красно-черное дерево состоит только из черных узлов.
   Следовательно, $n \geq 2^{h_{min}} - 1$,
   $h_{max} = 2h_{min} \leq 2\log_2(n+1)$
4. Вывод: высота красно-черного дерева $h = O(\log n)$.

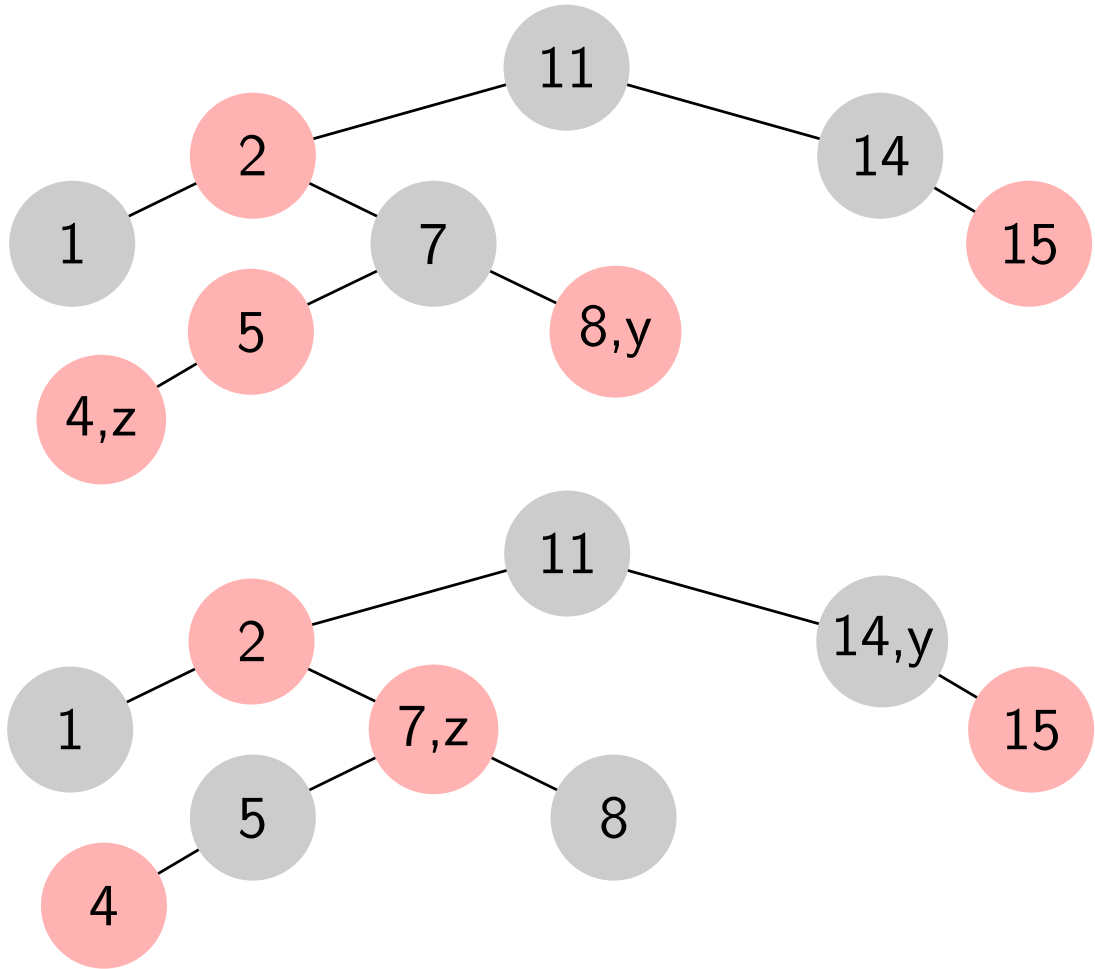
Раздел

Поиск
- Различные подходы

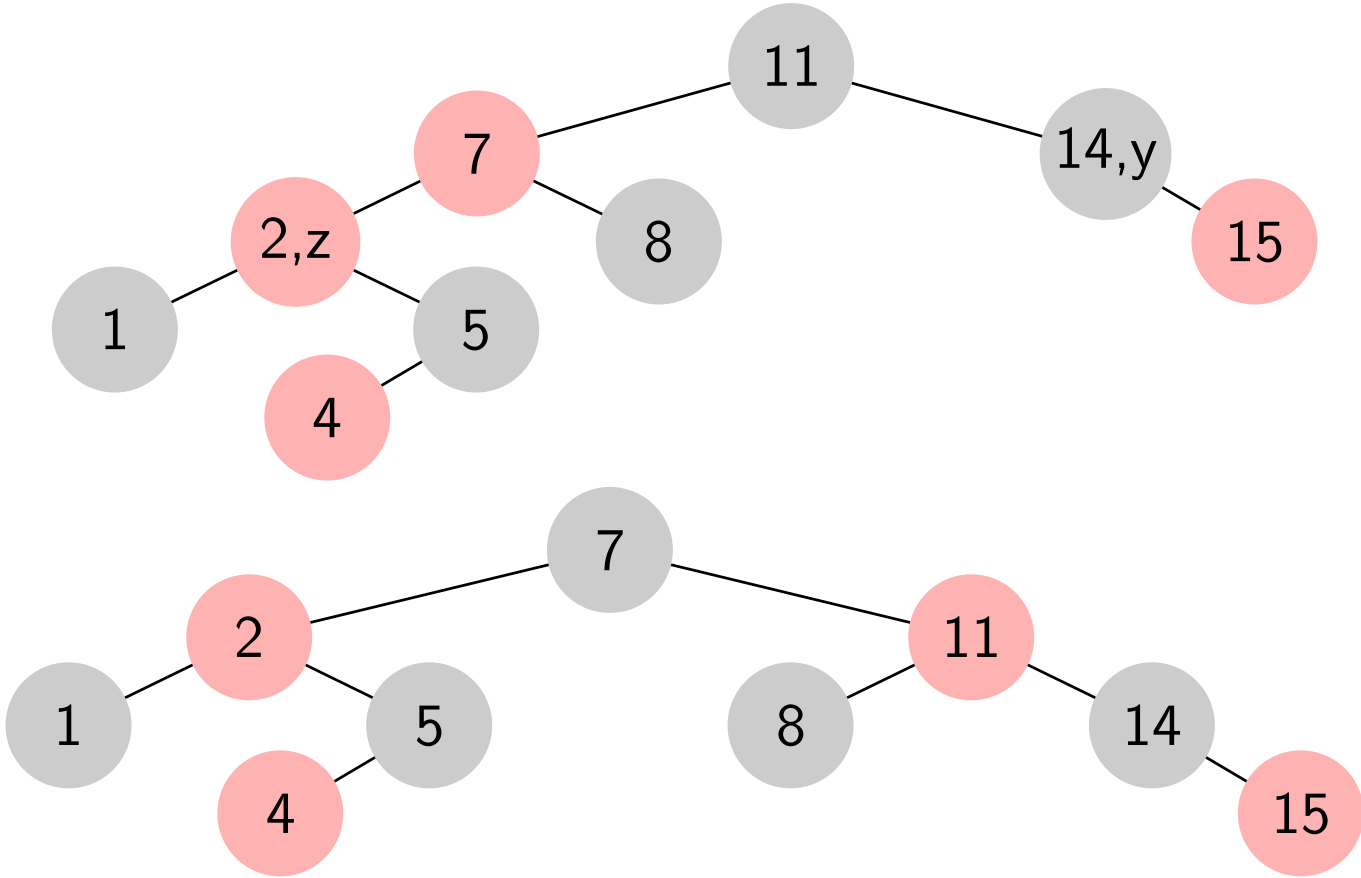
AVL-деревья
- Определение, высота дерева
- Вставка
- Удаление

Красно-черные деревья
- Определение, высота дерева
- Вставка
- Удаление

Вставка. Иллюстрация



Вставка. Иллюстрация



Вставка. Общая идея

- Выполняется вставка узла z в дерево T (как в обычное дерево поиска).
- Вставленный узел красится в красный цвет, $color(z) = red$. Возможно нарушено свойство 4.
- Вызывается процедура, гарантирующая корректность нового дерева. До тех пор, пока родитель узла z красный:
  - Пусть y — «дядя» узла z, если дядя красный, то нужно перекрасить его и отца в черный цвет, а дедушку в красный, и переместить указатель z на дедушку.
  - Если дядя черный, а узел z — левый сын, то нужно повернуть вправо отца и деда, отца покрасить в черный, деда в красный (у него гарантированно 2 черных сына).
  - Если дядя черный, а узел z — правый сын, то нужно повернуть z и отца влево и переместить указатель z на бывшего отца z. Задача свелась к предыдущей.

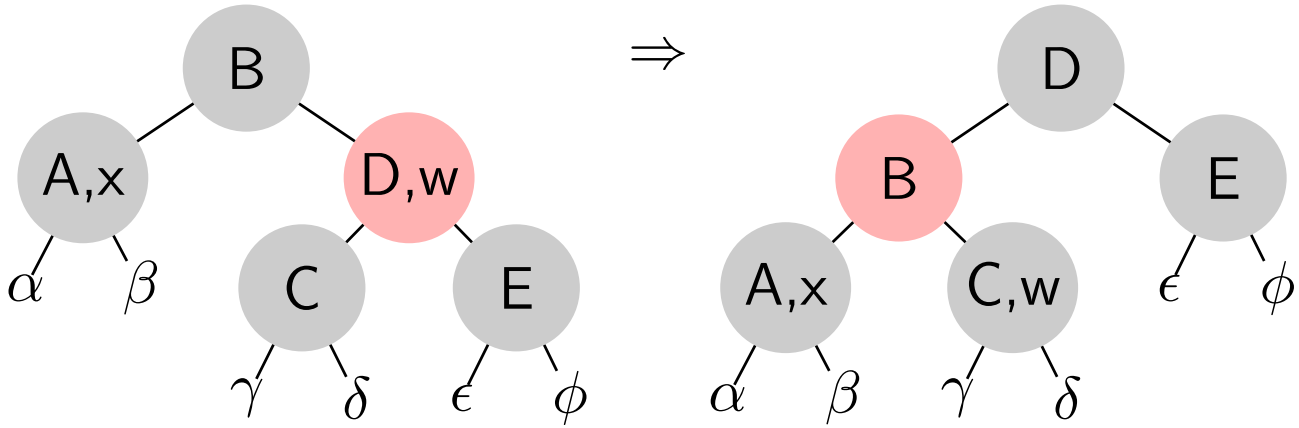
Вставка. Алгоритм

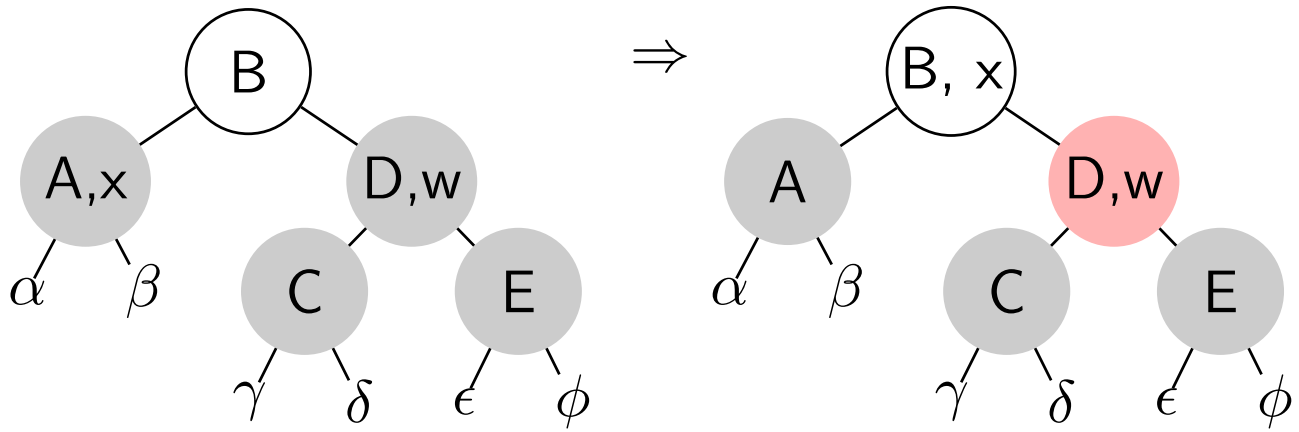
RB-INSERT-FIXUP(T, z)

```
 1  while color[p[z]] = RED
 2        if p[z] = left[p[p[z]]]
 3              y := right[p[p[z]]]
 4              if color[y] = RED
 5                    color[p[z]] := color[y] := BLACK
 6                    color[p[p[z]]] := RED, z := p[p[z]]
 7              else if z = right[p[z]]
 8                          LEFT-ROTATE(z, p[z])
 9                          z := left[z]
10                    color[p[z]] := BLACK, color[p[p[z]]] := RED
11                    RIGHT-ROTATE(p[z], p[p[z]])
12        else (то же, с заменой left на right и наоборот)
13  color[root[T]] := BLACK
```

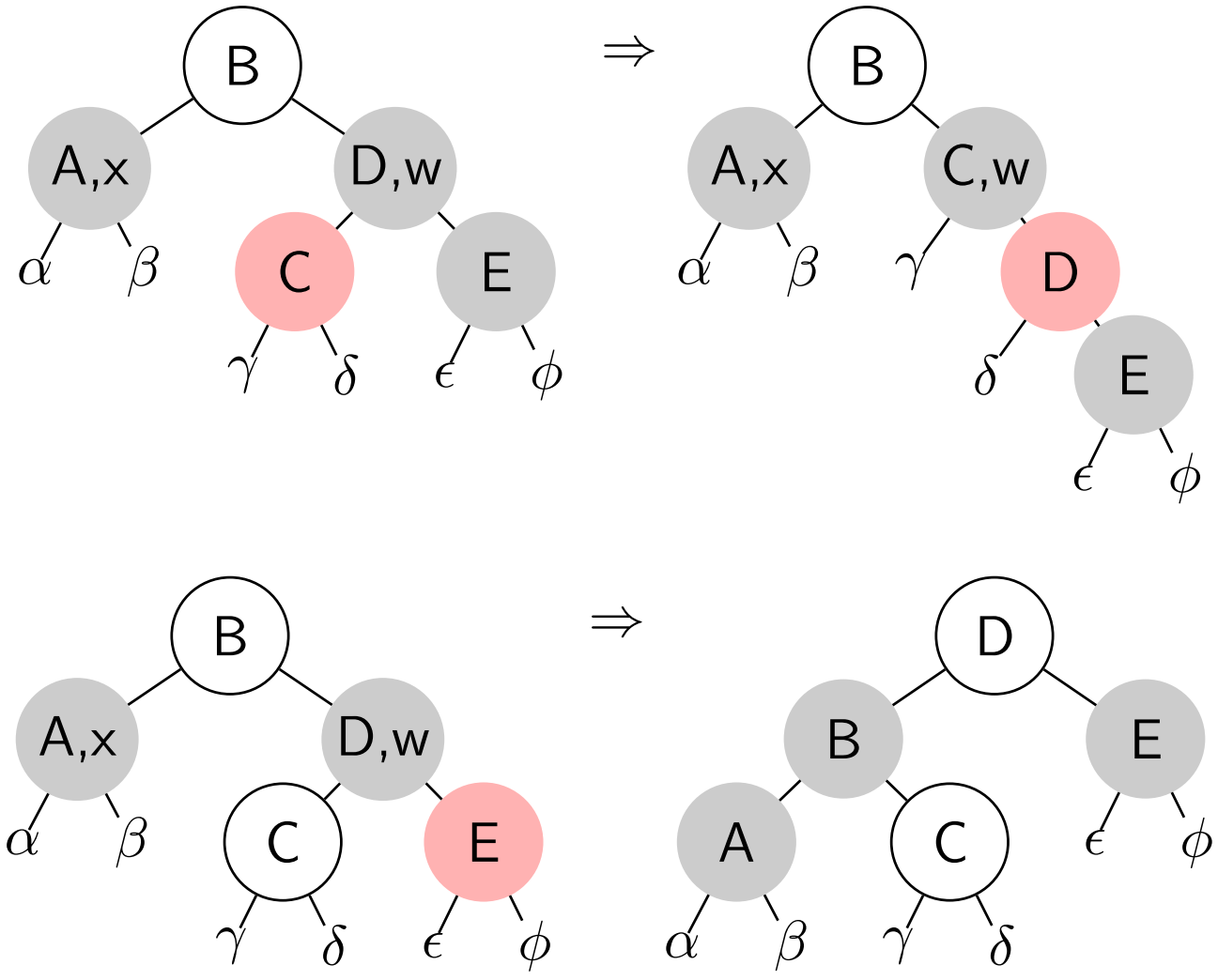
Раздел

Удаление. Иллюстрация





Удаление. Иллюстрация



Удаление. Общая идея

- Выполняется удаление узла z из дерева T (как из обычного дерево поиска).
- Если у узла z есть оба сына, то находится узел y — следующий по порядку, у которого нет одного из сыновей.
- Существующего сына узла y (или z) назовем x.
- Если удалялся черный узел, то оказалось нарушено свойство 5.
- Вызывается процедура, гарантирующая корректность нового дерева. До тех пор, пока x не корень, и пока он черный (если x — красный, то можно перекрасить его, и дерево станет корректным), выполнять некоторые действия.

Удаление. Алгоритм

RB-DELETE-FIXUP(T, x)

```
 1  while x ≠ root[T] и color[x] = BLACK
 2        if x = left[p[x]]
 3              w := right[p[x]]
 4              if color[w] = RED
 5                    color[w] := BLACK, color[p[x]] := RED
 6                    LEFT-ROTATE(w, p[x])
 7                    w := right[p[x]]
 8              if color[left[w]] = BLACK и color[right[w]] = BLACK
 9                    color[w] := RED
10                    x := p[x]
```

Удаление. Алгоритм

RB-DELETE-FIXUP(T, x)

```
 1  while x ≠ root[T] и color[x] = BLACK
 2        if x = left[p[x]]
 3        else if color[right[w]] = BLACK
 4                  color[left[w]] := BLACK, color[w] := RED
 5                  RIGHT-ROTATE(left[w], w)
 6                  w := right[p[x]]
 7              color[w] := color[p[x]]
 8              color[p[x]] := BLACK
 9              color[right[w]] := BLACK
10              LEFT-ROTATE(w, p[x])
11              x := root[T]
12        else (то же, с заменой left на right и наоборот)
13  color[x] := BLACK
```