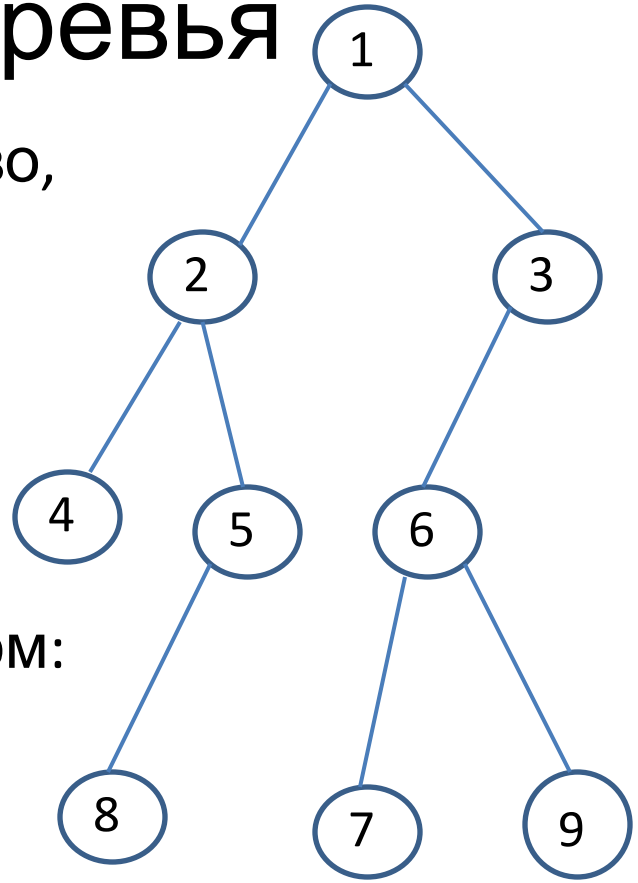
Алгоритмы и структуры данных

Лекция 7

Деревья поиска

Бинарные деревья

Упорядоченное дерево – это дерево, в котором множество сыновей каждой вершины упорядочено слева направо.



Бинарное дерево – это упорядоченное дерево, в котором:

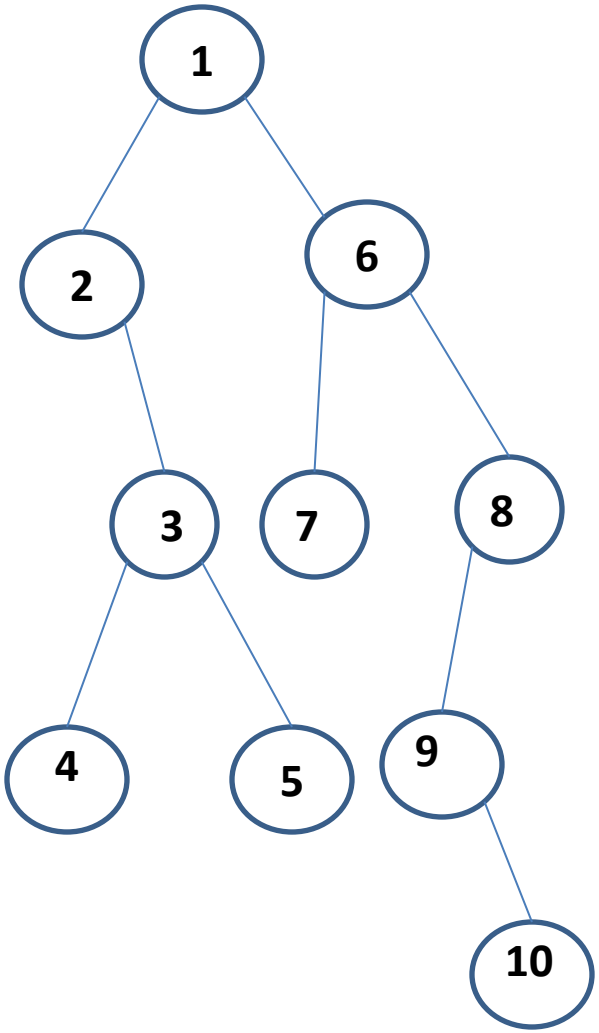
1) любой сын – либо левый либо правый,
2) любой узел имеет не более одного левого и не более одного правого сына.

Обходы деревьев в глубину

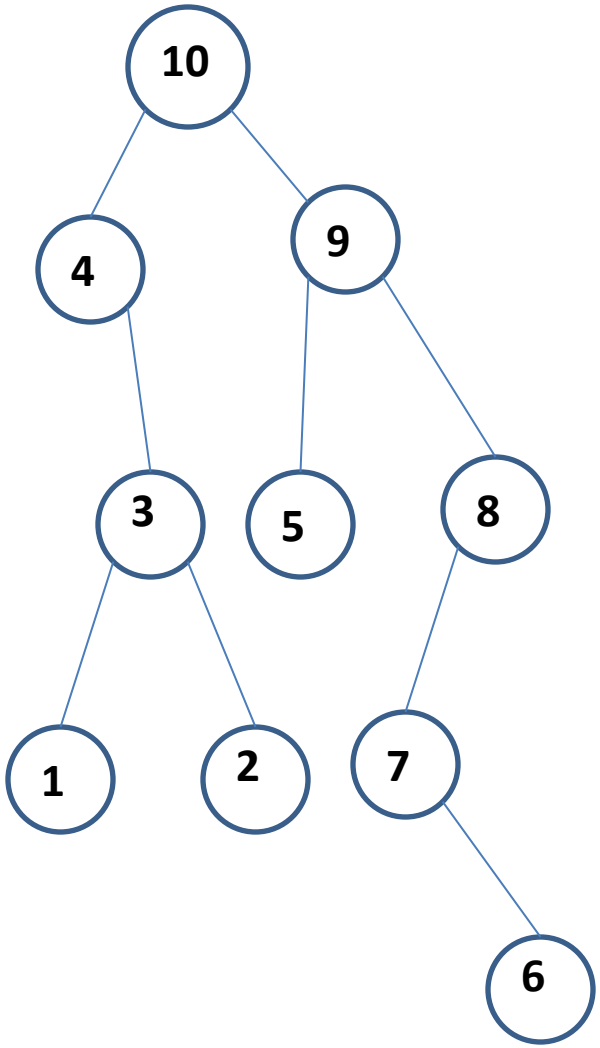
Пусть T – дерево, r- корень, $v_1, v_2, \ldots, v_n$ – сыновья вершины r.

1. Прямой (префиксный) обход:
   - посетить корень *r;*
   - посетить в прямом порядке поддеревья с корнями $v_1, v_2, \ldots, v_n$.
2. Обратный (постфиксный) обход:
   - посетить в обратном порядке поддеревья с корнями $v_1, v_2, \ldots, v_n$;
   - посетить корень *r.*
3. Внутренний (инфиксный) обход для бинарных деревьев:
   - посетить во внутреннем порядке левое поддерево корня *r* (*если существует*);
   - посетить корень *r;*
   - посетить во внутреннем порядке правое поддерево корня *r* (*если существует*).

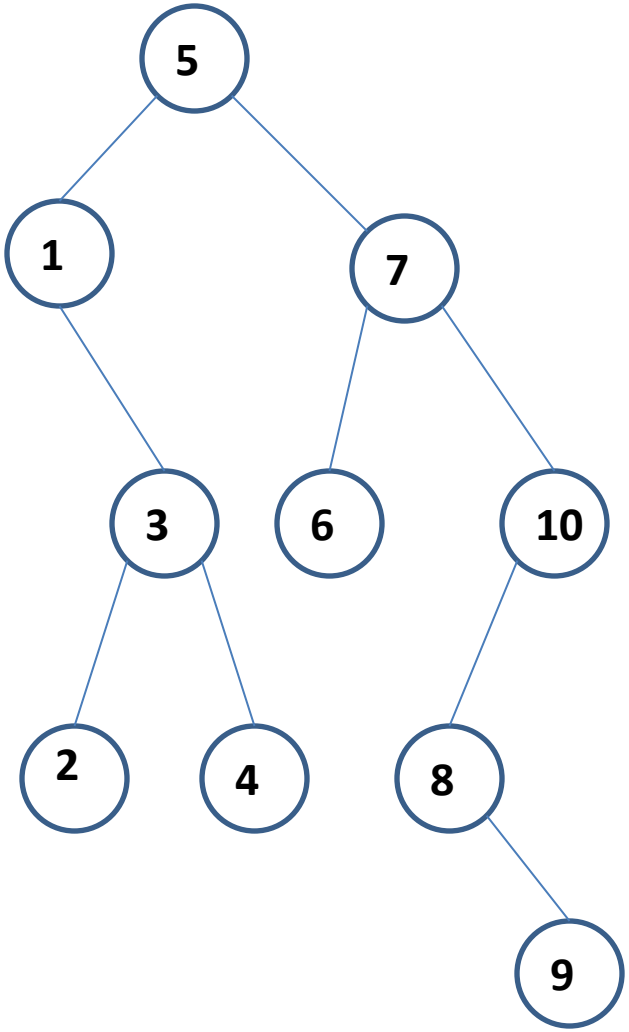
Обходы деревьев в глубину. Пример 1.



Прямой

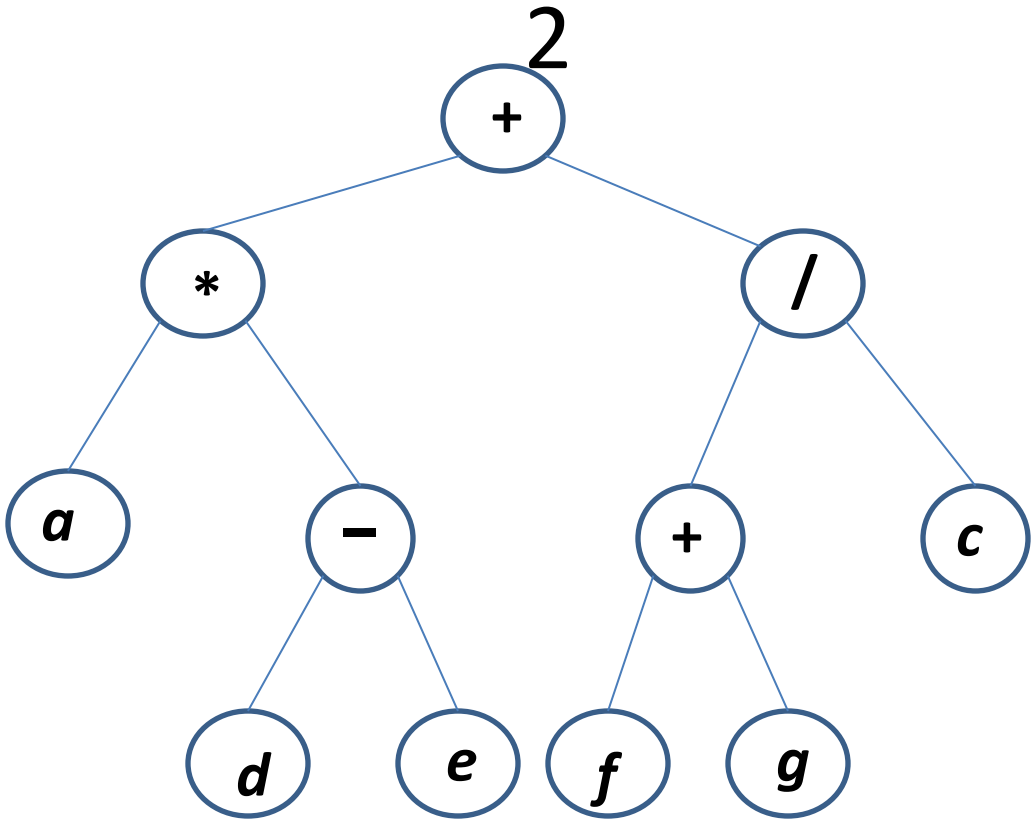


Обратный



Внутренний

Обходы деревьев в глубину. Пример 2



*+ * a – d e / + f g c* - *префиксный обход*

*a d e – * f g + c / +* - *постфиксный обход*

*a * (d – e)+ (f + g) / c* - *инфиксный обход*

Реализация обхода

```
typedef  struct node {
    char * word;
    struct node * left;
    struct node * right;
} tree;

 void print_tree ( tree * t )
{
    if ( !t )
      return;
    print_tree( t->left );
    printf ( “%s\n”, t->word );
    print_tree( t->right );
}
```

Обход дерева в ширину

- это обход вершин дерева по уровням, начиная от корня, слева *направо* (или справа налево).

Алгоритм обхода дерева в ширину

Шаг 0:

Поместить в очередь корень дерева.

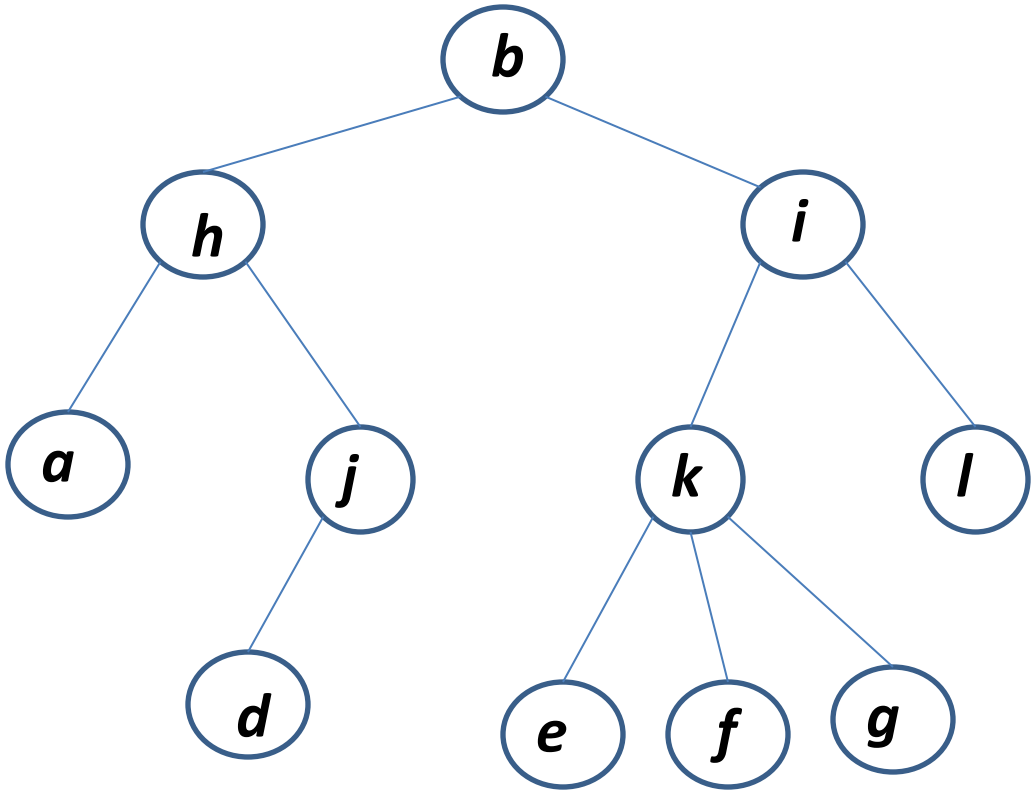
Шаг 1:

Взять из очереди очередную вершину. Поместить в очередь всех ее сыновей по порядку слева направо (справа налево).

Шаг 2:

Если очередь пуста, то конец обхода, иначе перейти на Шаг 1.

Обход дерева в ширину. Пример



b	h	i	a	j	k	l	d	e	f	g

Представления деревьев

Определение. *Левое скобочное представление* дерева *T* (обозначается *Lrep*(*T*)) можно получить, применяя к нему следующие рекурсивные правила:

(1) Если корнем дерева T служит вершина *a* с поддеревьями T_1, T_2, . . ., T_n, расположенными в этом порядке (их корни — прямые потомки вершины a), то

Lrep(*T*) = *a* (*Lrep* (T_1), *Lrep* (T_2) , . . ., *Lrep* (T_n))

(2) Если корнем дерева *T* служит вершина a, не имеющая прямых потомков, то *Lrep* (T) = *a*.

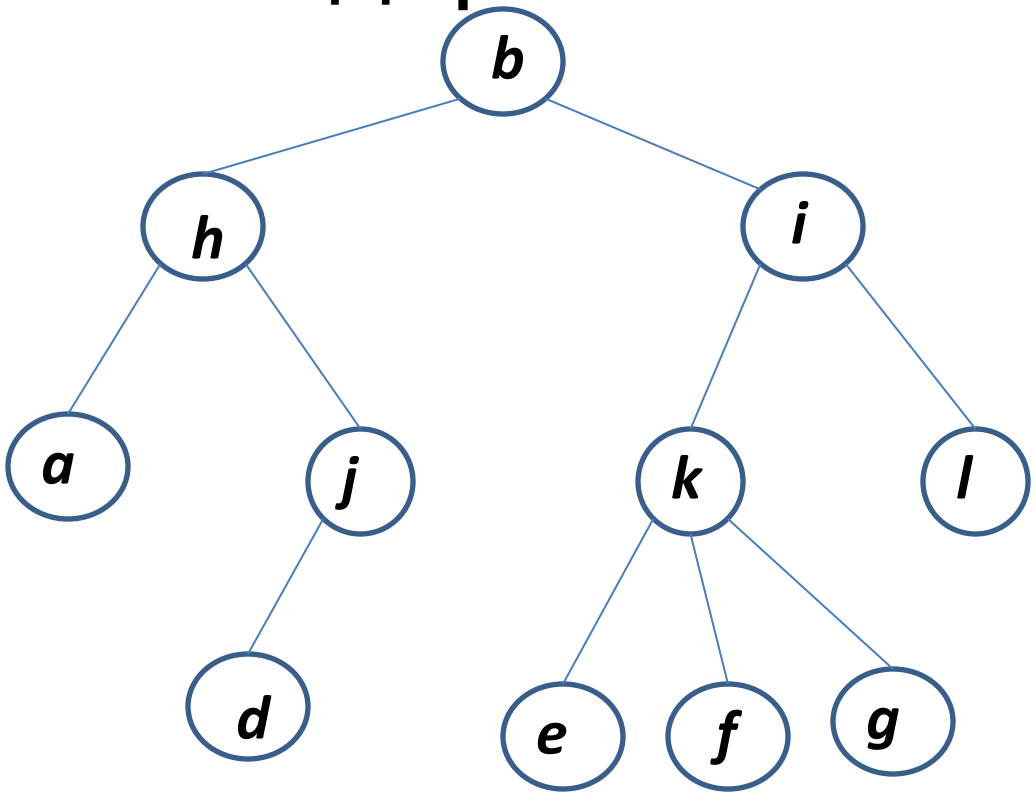
Определение. *Правое скобочное представление* *Rrep*(*T*) дерева *T:*

(1) Если корнем дерева *T* служит вершина *a* с поддеревьями T_1, T_2, . . ., T_n, то

Rrep(*T*) = (*Rrep*(T_1), *Rrep*(T_2), . . ., *Rrep* (T_n))*a*.

(2) Если корнем дерева *T* служит вершина *a,* не имеющая прямых потомков, то *Rrep*(*T*) = *a*.

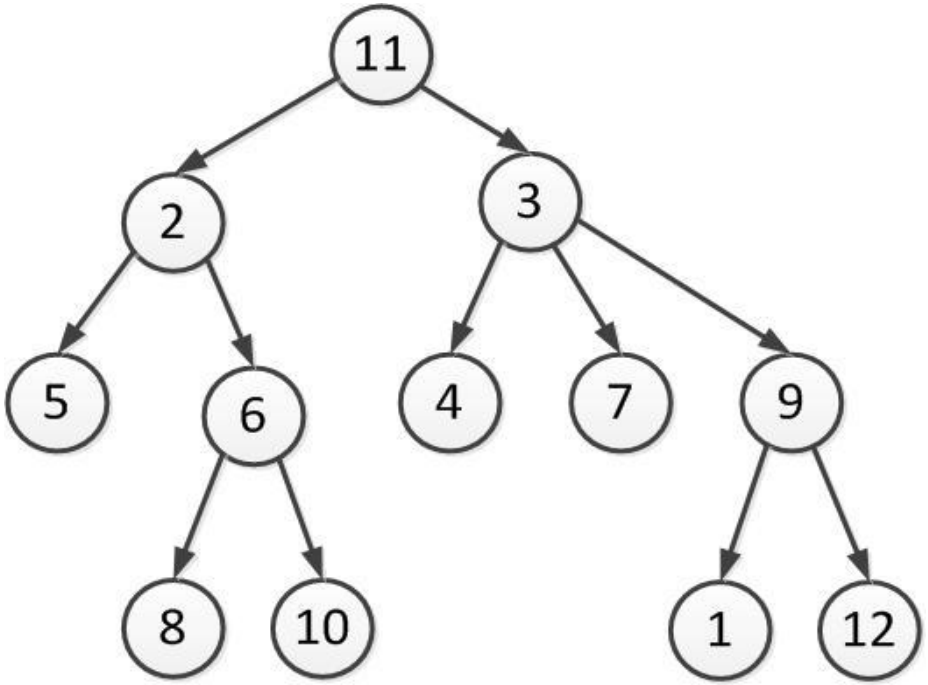
Скобочные представления деревьев



- *Lrep*(*T*) = *b* (*h* (*a*, *j* (*d*)), *i* (*k* (*e*, *f*, *g*), *l*))
- *Rrep*(*T*) = ((*a*, (*d*) *j*) *h*, ((*e*, *f*, *g*) *k*, *l*) *i*) *b*

Представление дерева списком прямых предков

Составляется список прямых предков для вершин дерева с номерами 1, 2, ..., *n* (именно в этом порядке). Чтобы опознать корень, будем считать, что его предок—это 0.



9	11	11	3	2	2	3	6	3	6	0	9
1	2	3	4	5	6	7	8	9	10	11	12

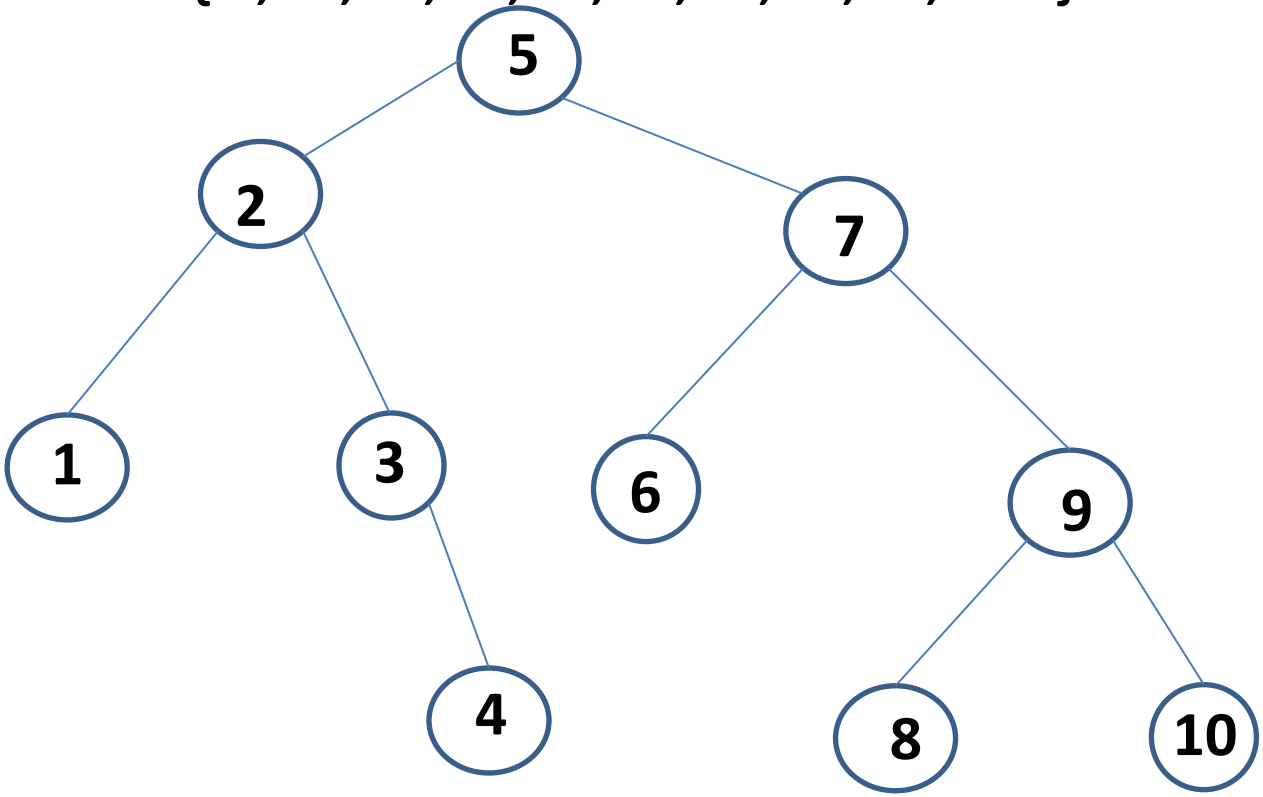
Дерево двоичного поиска

Определение. Деревом двоичного поиска для множества S называется помеченное двоичное дерево, каждый узел v которого помечен элементом $l(v) \in S$ так, что

1) $l(u) < l(v)$ для каждого узла u из левого поддерева узла v,
2) $l(w) > l(v)$ для каждого узла w из правого поддерева узла v,
3) для любого элемента $a \in S$ существует единственный узел v , такой что $l(v) = a$.

Дерево двоичного поиска. Пример

Пусть S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}



Алгоритм просмотра дерева двоичного поиска

Вход: Дерево *T* двоичного поиска для множества *S*, элемент *a*.

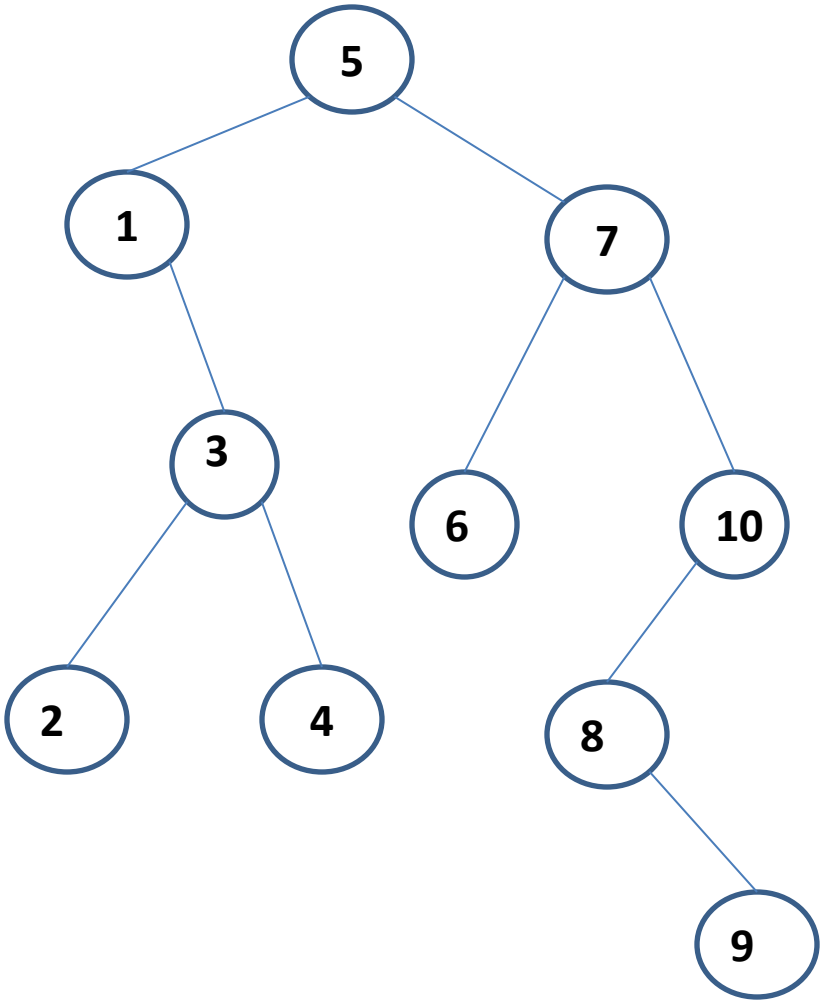
Выход: *true* если $a \in S$, *false* - в противном случае.

Метод: Если $T = \varnothing$, то выдать *false,* иначе выдать ПОИСК (*a*, *r*), где *r* – корень дерева *T*.

```
функция ПОИСК (a, v) : boolean
{
    если a = l(v) то выдать true
    иначе
      если a < l(v)  то
            если v имеет левого сына w
            то выдать ПОИСК (a, w)
            иначе выдать false;
      иначе
            если v имеет правого сына w
            то выдать ПОИСК (a, w)
            иначе выдать false;
}
```

Пример построения дерева двоичного поиска

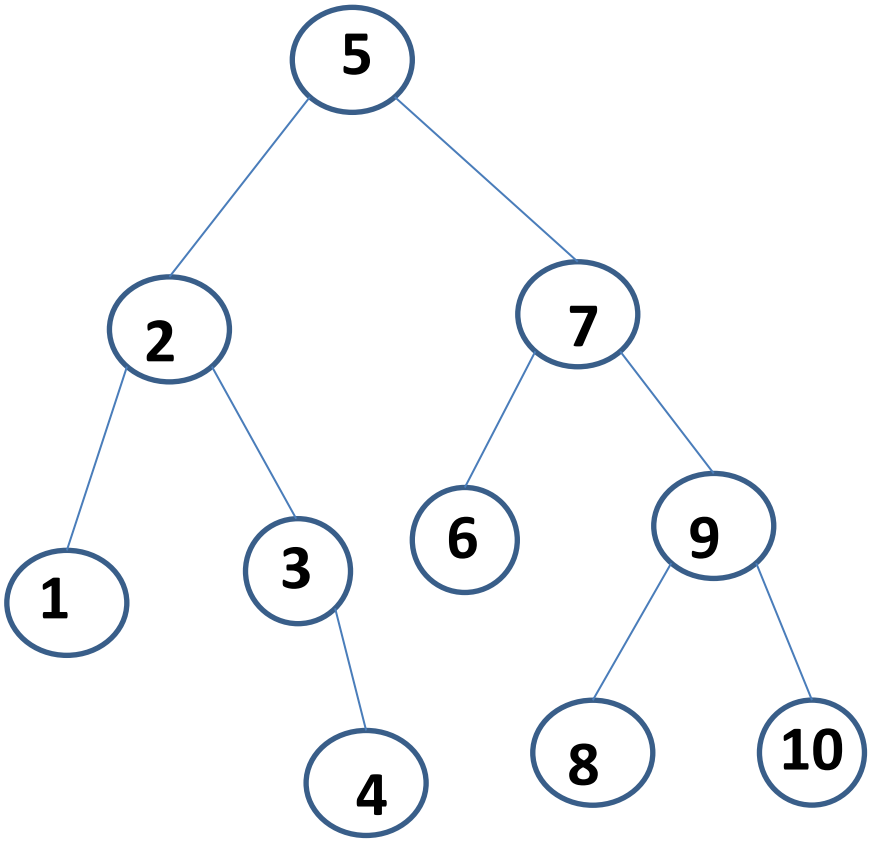
Пусть на вход подаются числа в следующем порядке:
5, 1, 7, 6, 3, 2, 10, 8, 4, 9



Операции нахождения минимума и максимума

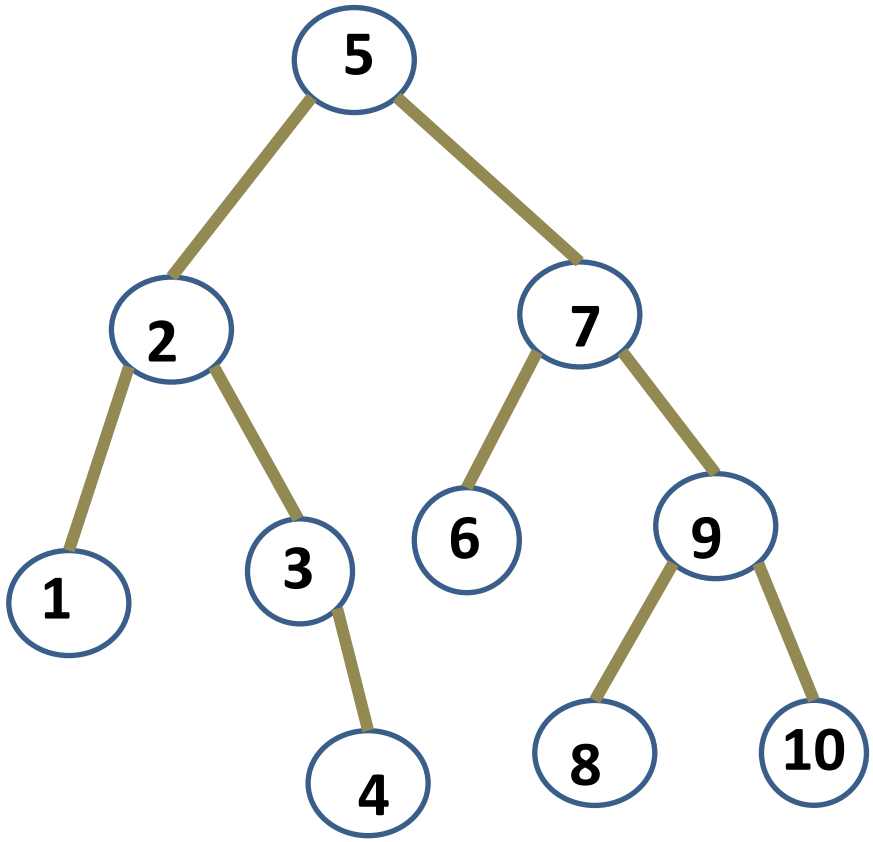
```
min ( v )
  v ← root( T );
  while left(v) ≠ NIL
   v ← left(v)
  return v
```

```
max ( v )
  v ← root(T);
  while right(v) ≠ NIL
   v ← right(v)
  return v
```



Операция поиска следующего

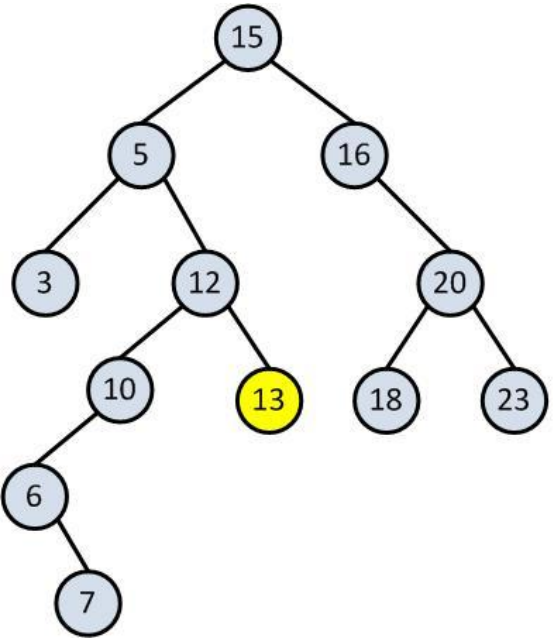
```
successor (x)
  if right[x] ≠ NIL then
   return min (right [x])
  y ← p[x]
  while y ≠ NIL and x = right [y]
   do
      x ← y
      y ← p[y]
  return y
```

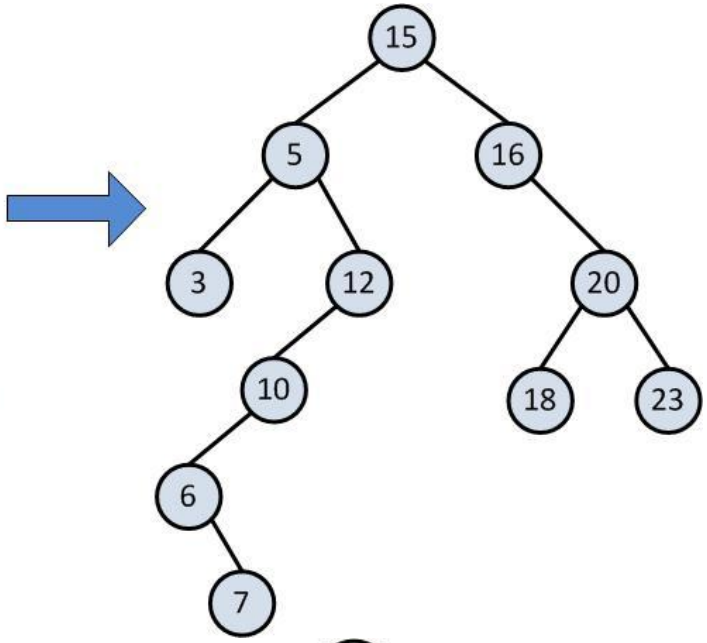


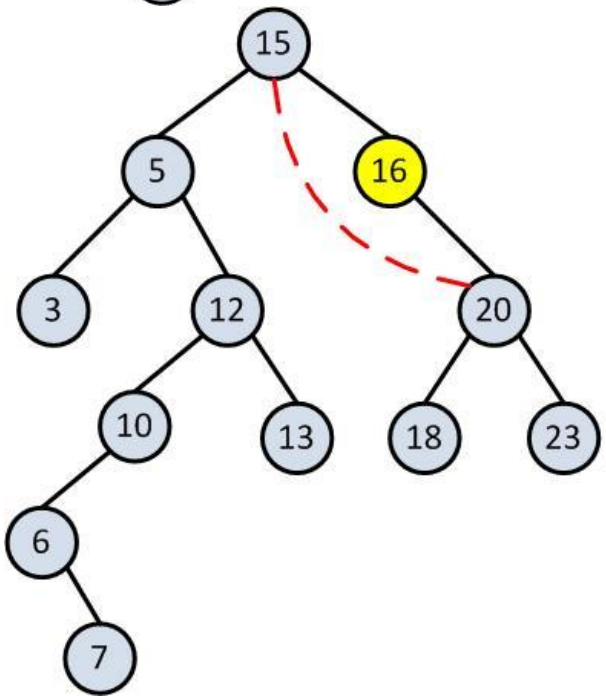
Операция удаления элемента из дерева поиска

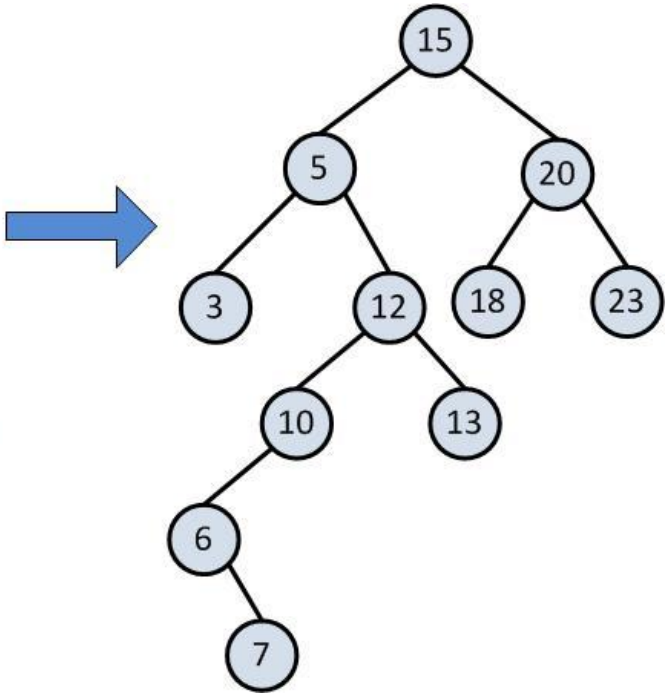
```
Delete(T, z)
  if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← successor (z)
  if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
  if x ≠ NIL
   p[x] ← p[y]
  if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
         then left[p[y]] ← x
         else right[p[y]] ← x
  if y ≠ z
   then key[z] ← key[y]
      Копирование сопутствующих данных в z
  return y
```

Пример удаления элемента из дерева поиска

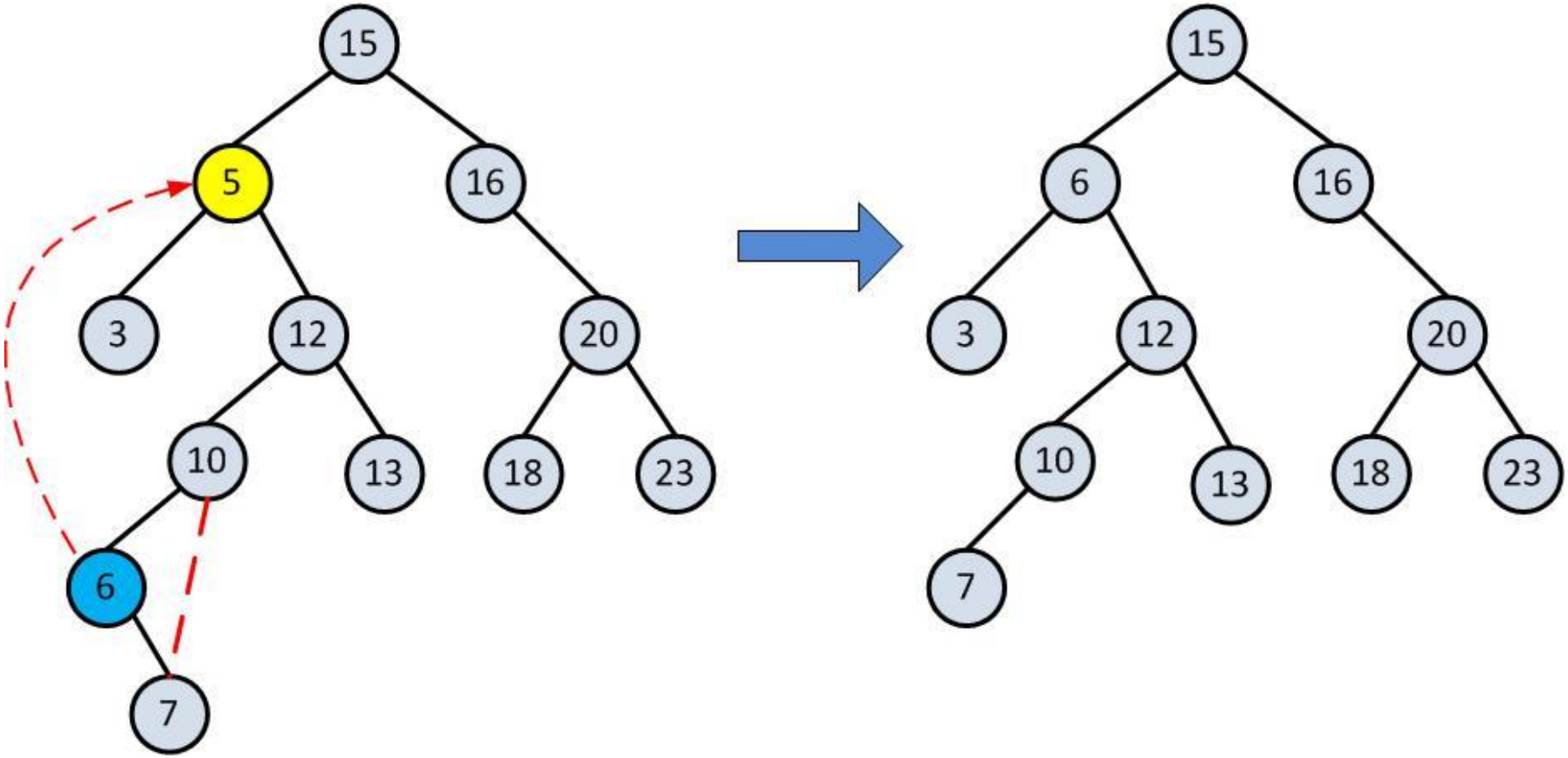








Пример удаления элемента из дерева поиска (окончание)



Теорема (*Т. Кормен и др.*)

Математическое ожидание высоты случайного бинарного дерева поиска с *n* ключами равно $O(\log_2 n)$

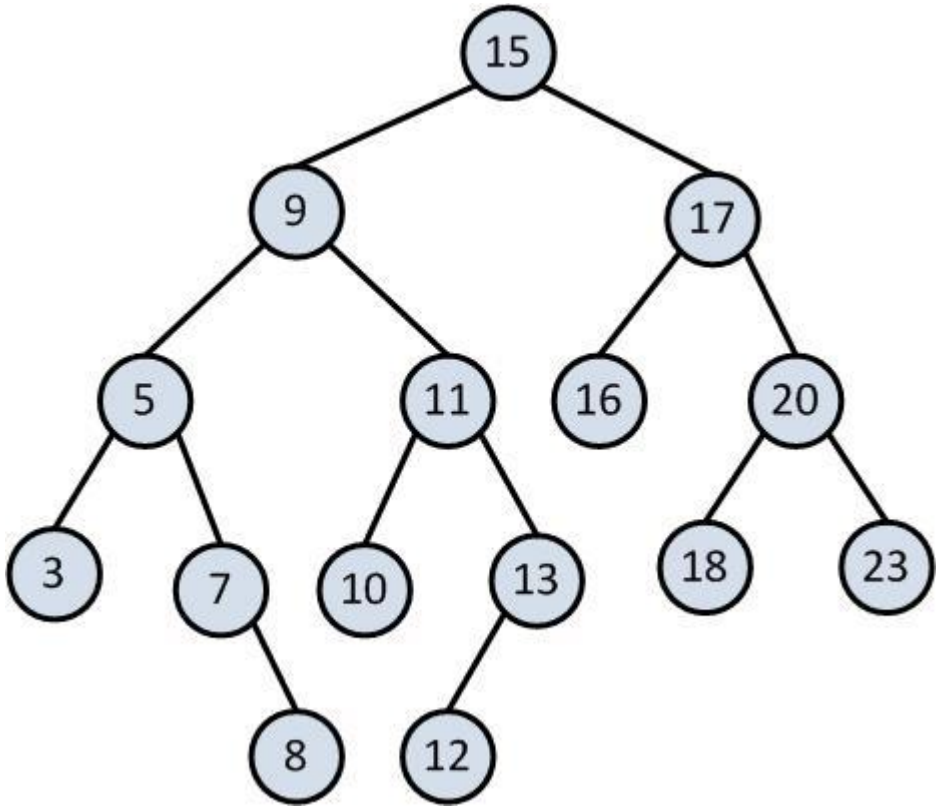
Сбалансированные деревья

Определение

Дерево называется сбалансированным тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на единицу.

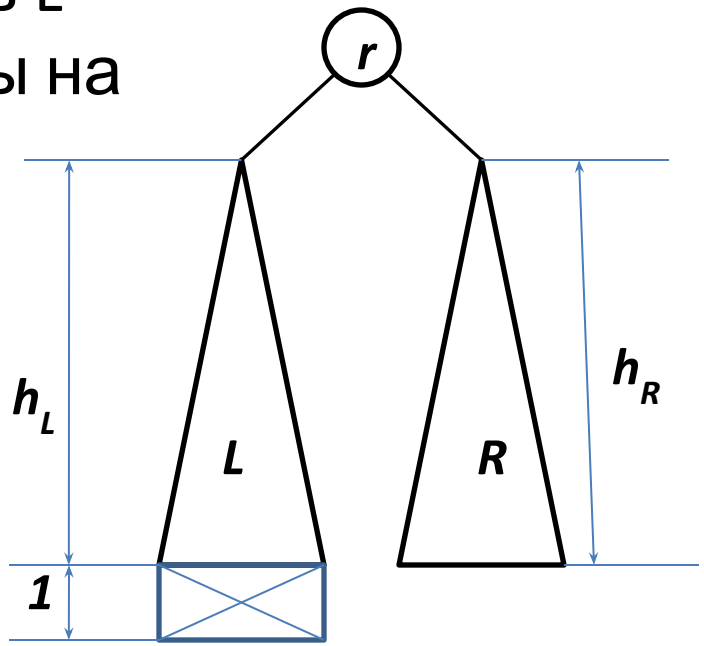
АВЛ-деревья

(1964 г. - Г.М.Адельсон-Вельский, Е.М. Ландис)



Вставка элемента в сбалансированное дерево

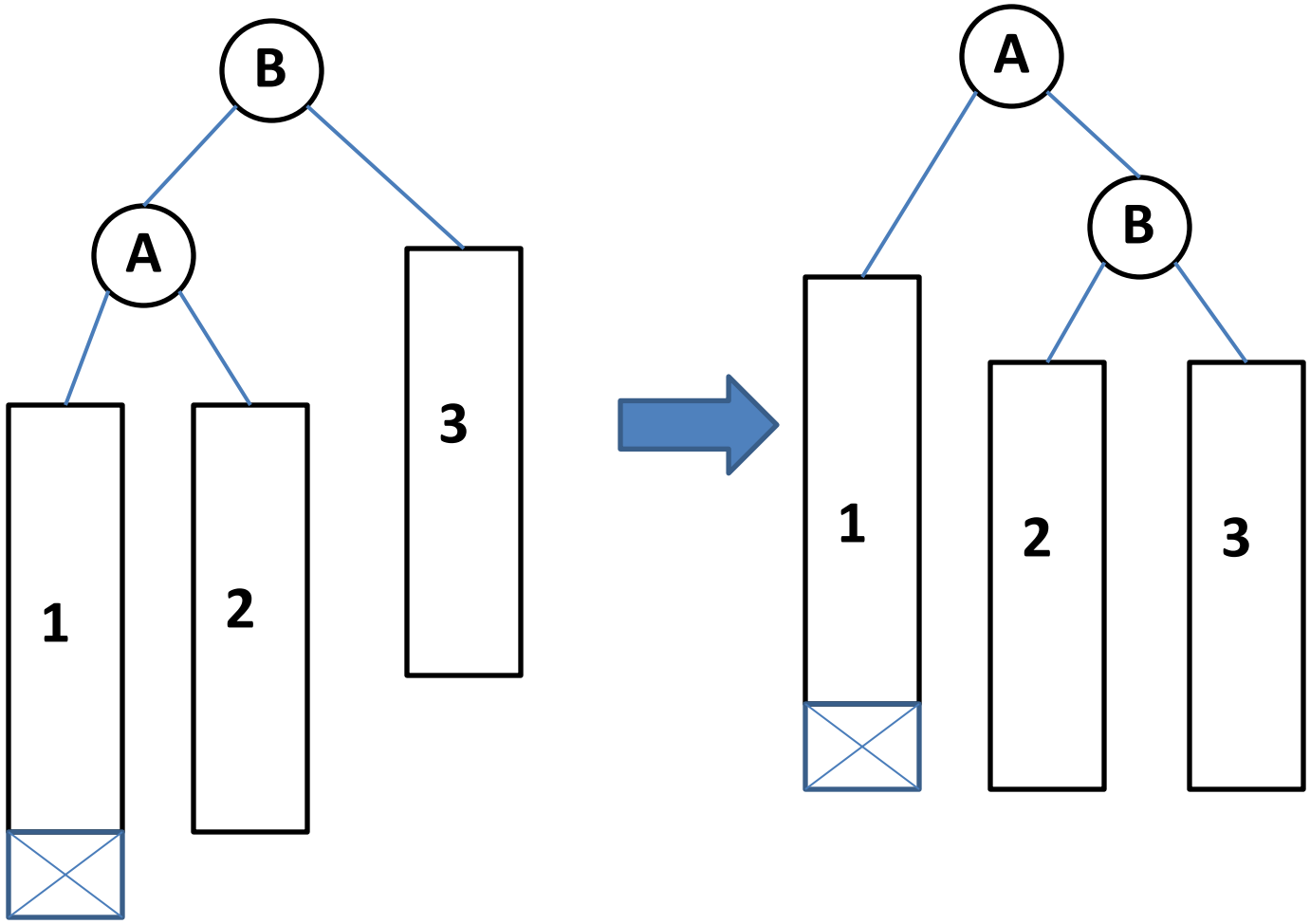
Пусть r – корень, L – левое поддерево, R – правое поддерево. Предположим, что включение в L приведет к увеличению высоты на 1.



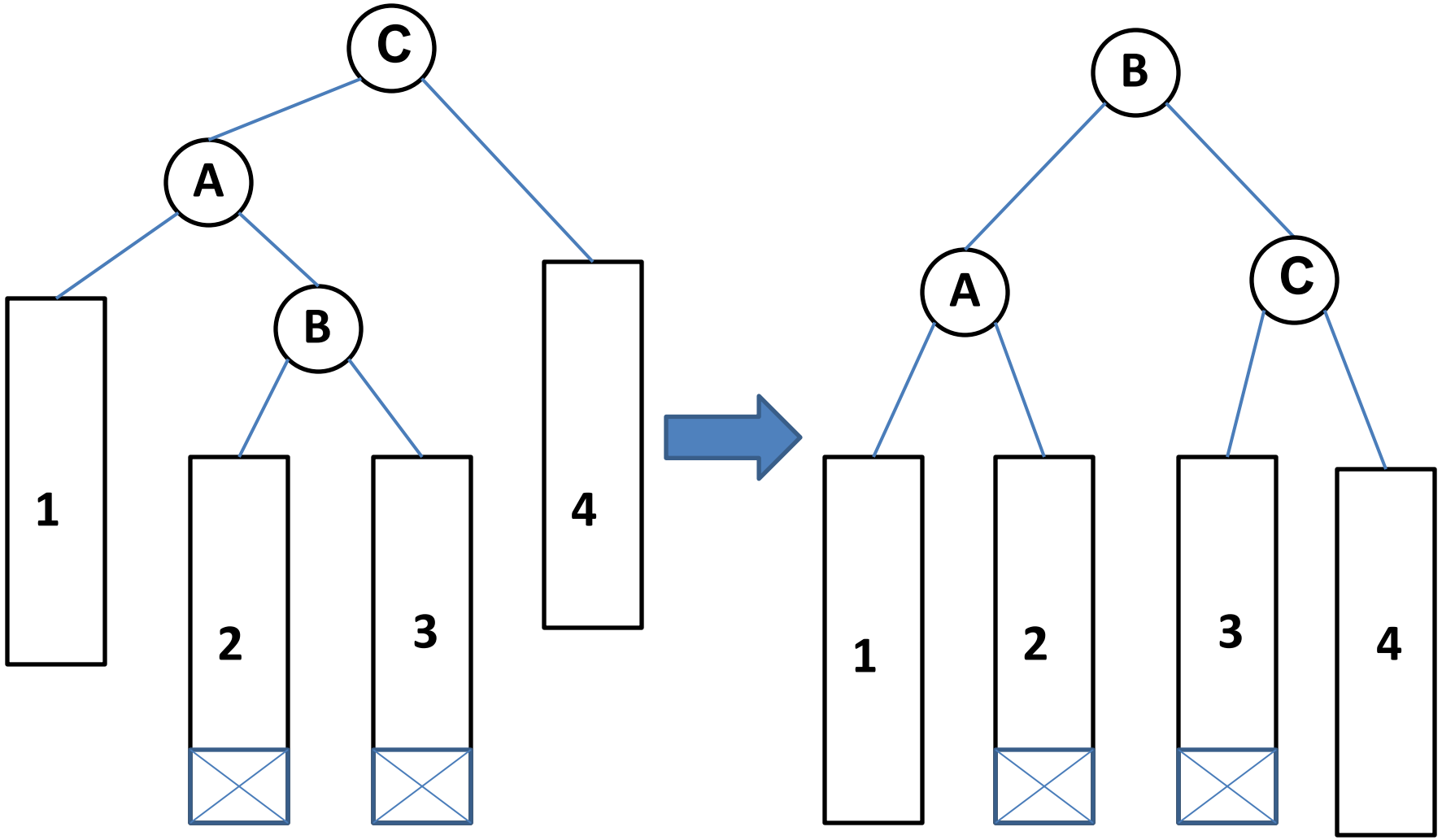
Возможны три случая:

1. $h_L = h_R$
2. $h_L < h_R$
3. $h_L > h_R$ *→нарушен принцип сбалансированности, дерево нужно перестраивать*

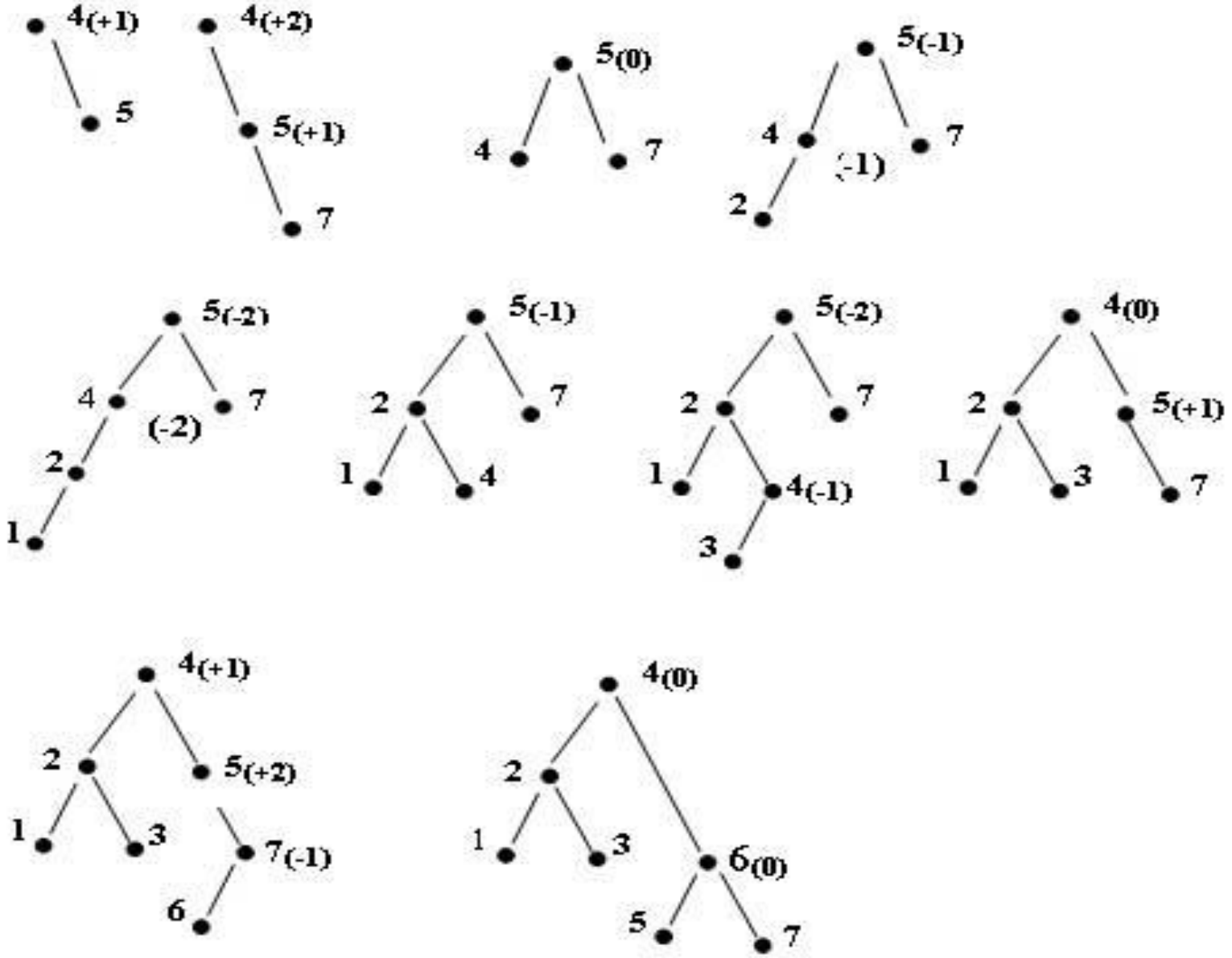
Вставка в левое поддерево



Вставка в правое поддерево



Пример построения АВЛ-дерева



Красно-чёрное дерево (Red-Black-Tree, RB-Tree)

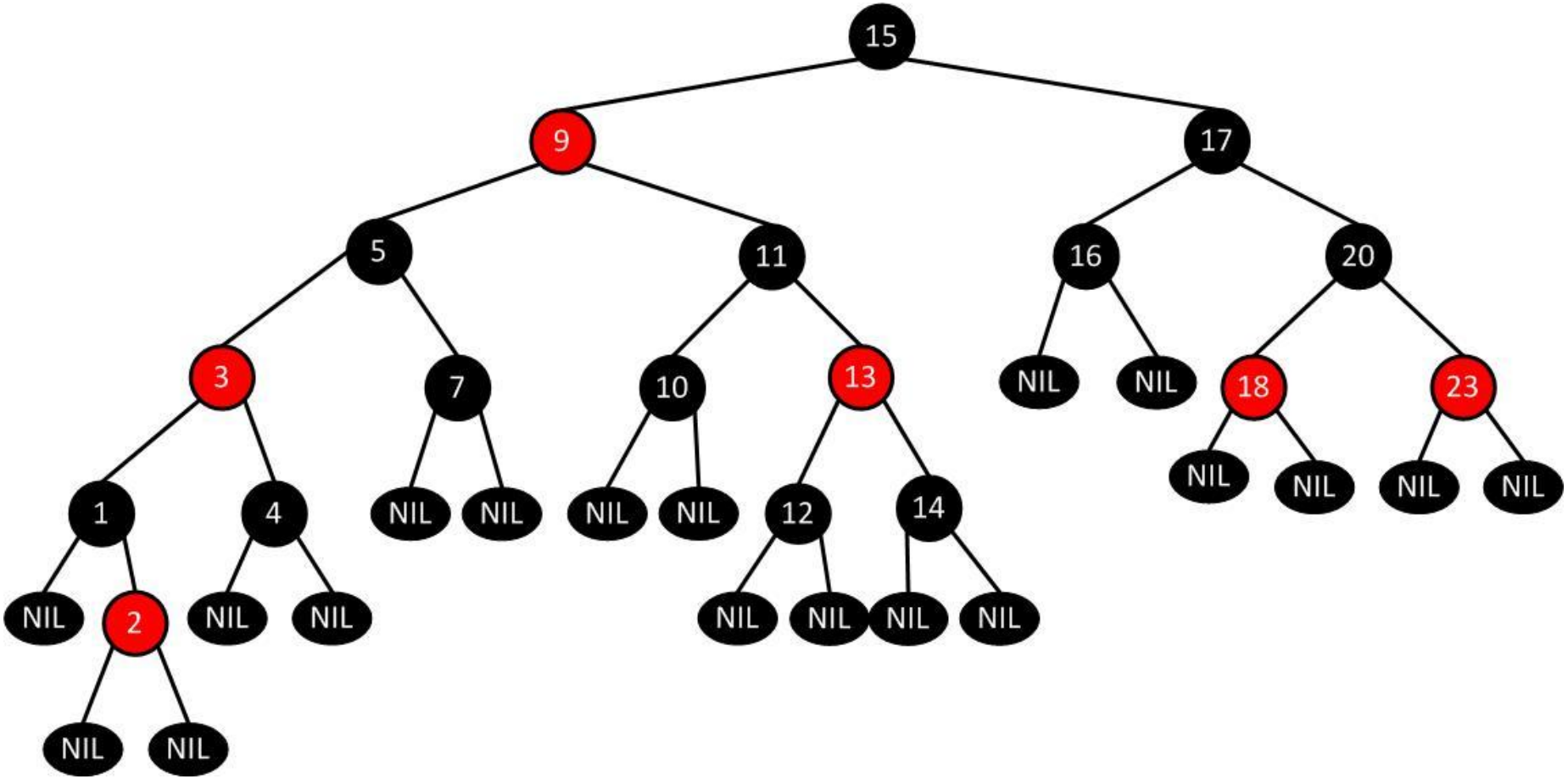
Красно-чёрное дерево обладает следующими свойствами.

1. Каждый узел либо красный либо чёрный
2. Все листья черны.
3. Корень дерева – чёрный
4. Все сыновья красных узлов черны.
5. Для каждого узла все пути от него до листьев, являющихся его потомками, содержат одно и то же количество черных узлов.

Для корня число чёрных узлов до листьев называется чёрной высотой дерева.

Для удобства листьями красно-чёрного дерева считаются фиктивные «нулевые» узлы, не содержащие данных (NIL).

Пример


15
9
17
5
11
16
20
3
7
10
13
NIL
NIL
18
23
1
4
NIL
NIL
NIL
NIL
12
14
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
2
NIL
NIL
NIL
NIL

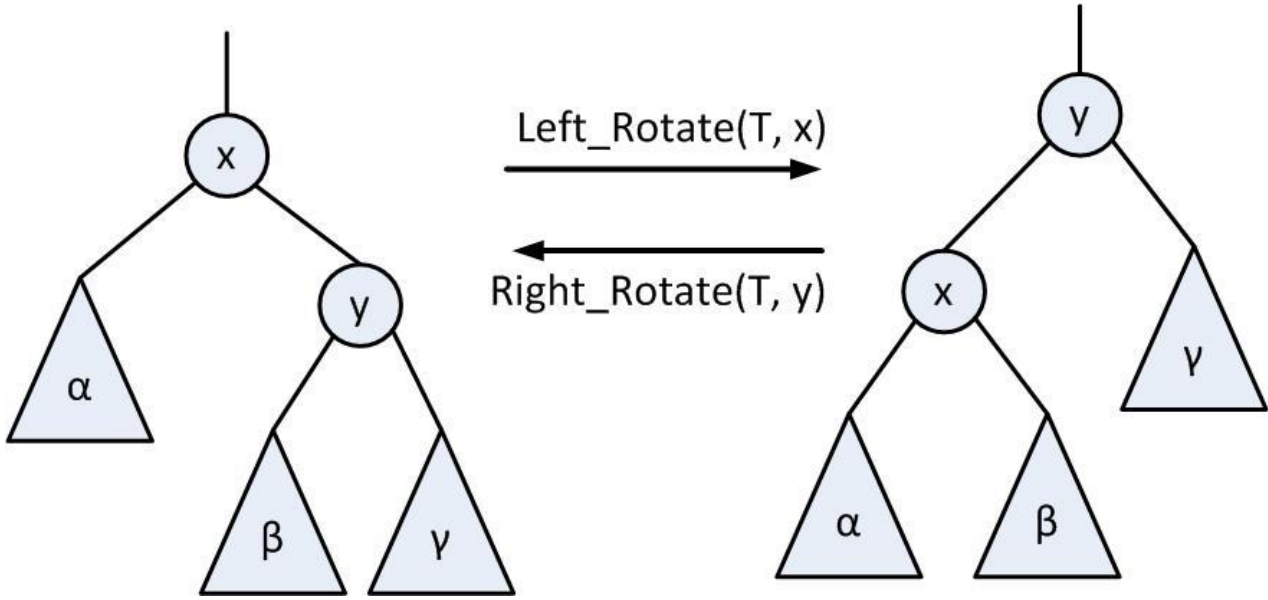
Свойства красно-чёрного дерева

1. Если h - чёрная высота дерева, то количество листьев не менее 2^{h-1}.

2. Если h - высота дерева, то количество листьев не менее $2^{(h-1)/2}$.

3. Если количество листьев N, высота дерева не больше $2log_2(N+1)$

Повороты




```
Left_Rotate(T, x)
    y ← right[x]
    right[x] ← left[y]
    if left[y] ≠ nil [T]
        then p[left[y]] ← x
    p[y] ← p[x]
    if p[x] = nil [T]
        then root[T] ← y
        else if x = left[p[x]]
            then left[p[x]] ← y
            else right[p[x]] ← y
    left[y] ← x
    p[x] ← y
```

Операция вставки

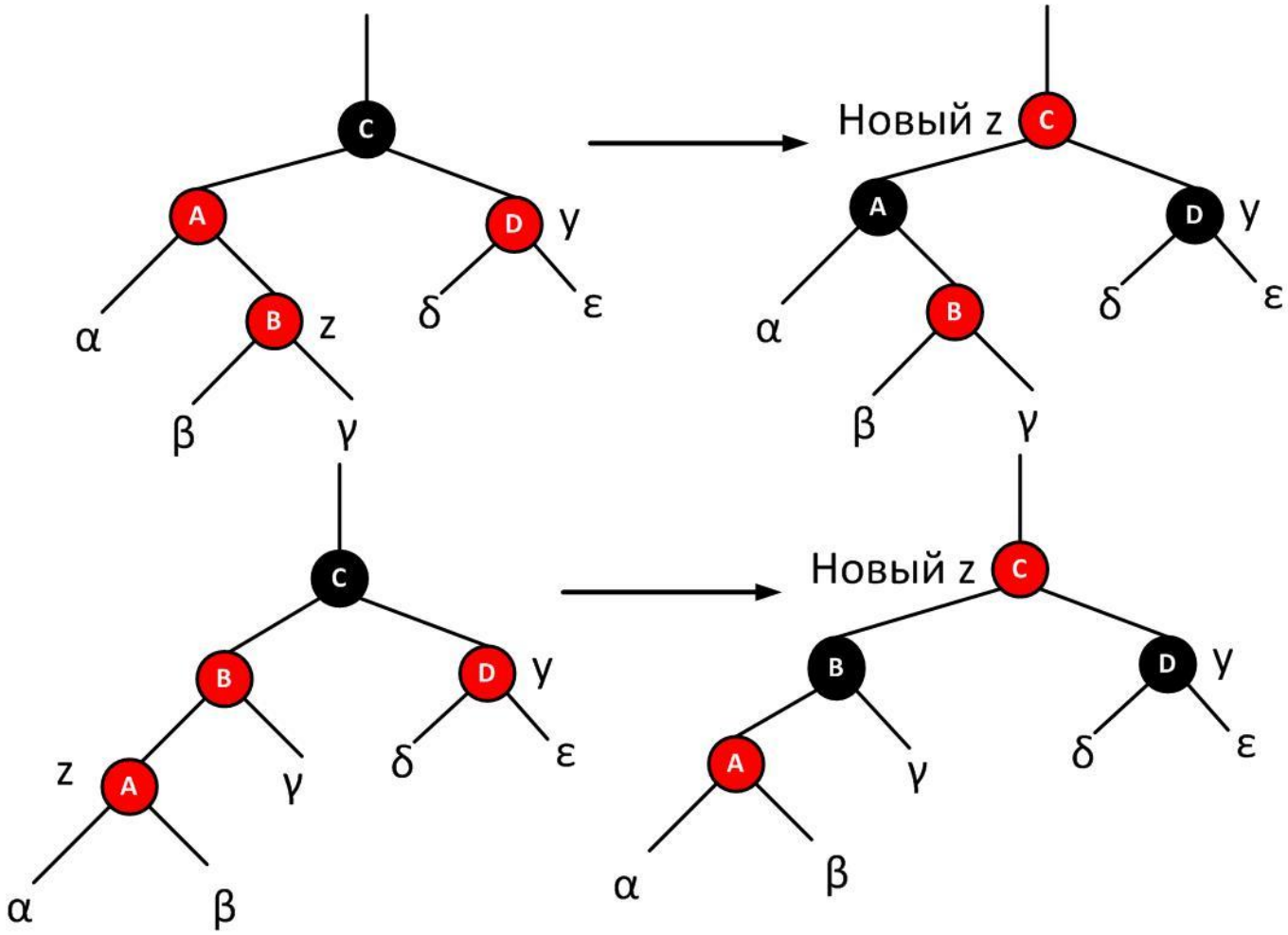
Чтобы вставить узел, мы сначала

- ищем в дереве место, куда его следует добавить.
- Новый узел всегда добавляется как лист, поэтому оба его потомка являются **NULL**-узлами и предполагаются черными.
- После вставки красим узел в красный цвет.
- После этого применяем процедуру Fixup:

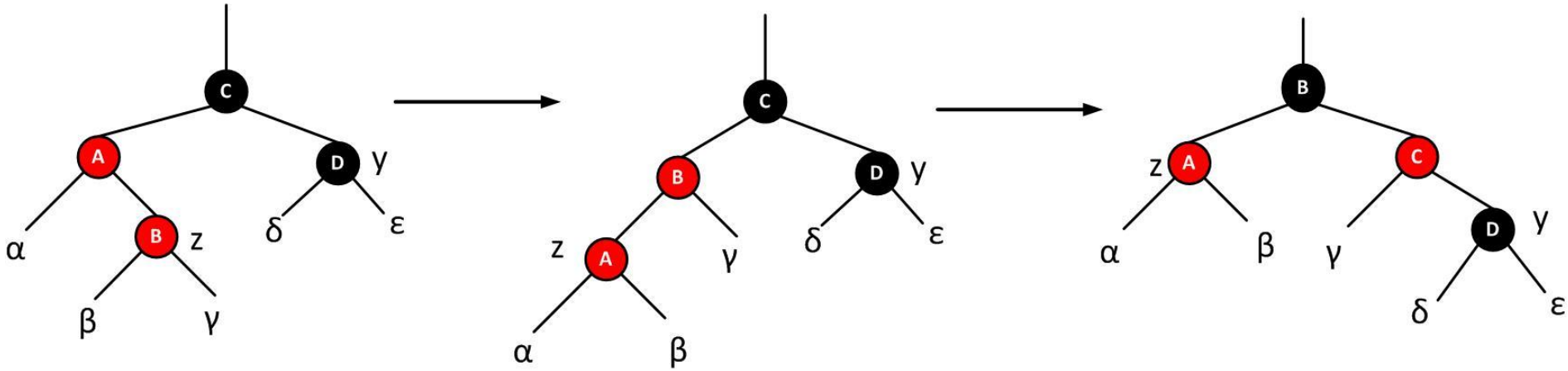
```
Fixup (T, z)
    while color[p[x]] = RED
        do   if p[z] = left[p[p[z]]]
             then y ←right[p[p[z]]]
                  if color [y] = RED                    //Случай 1
                  then [p[z]] ← BLACK
                       color [p[p[z]]] ← RED
                       z ←p[p[z]]
                  else if z = right[p[z]]               // Случай 2
                       then z ←p[z]
                            Left_Rotate (T, z)
                            color[p[z]] ← BLACK         // Случай 3
                            color [p[p[z]]] ← RED
                            Right_Rotate(T, p[p[z]])
             else (симметрично с then)
    color[root[T]] ← RLACK
```

Красный предок, красный «дядя» – случай 1

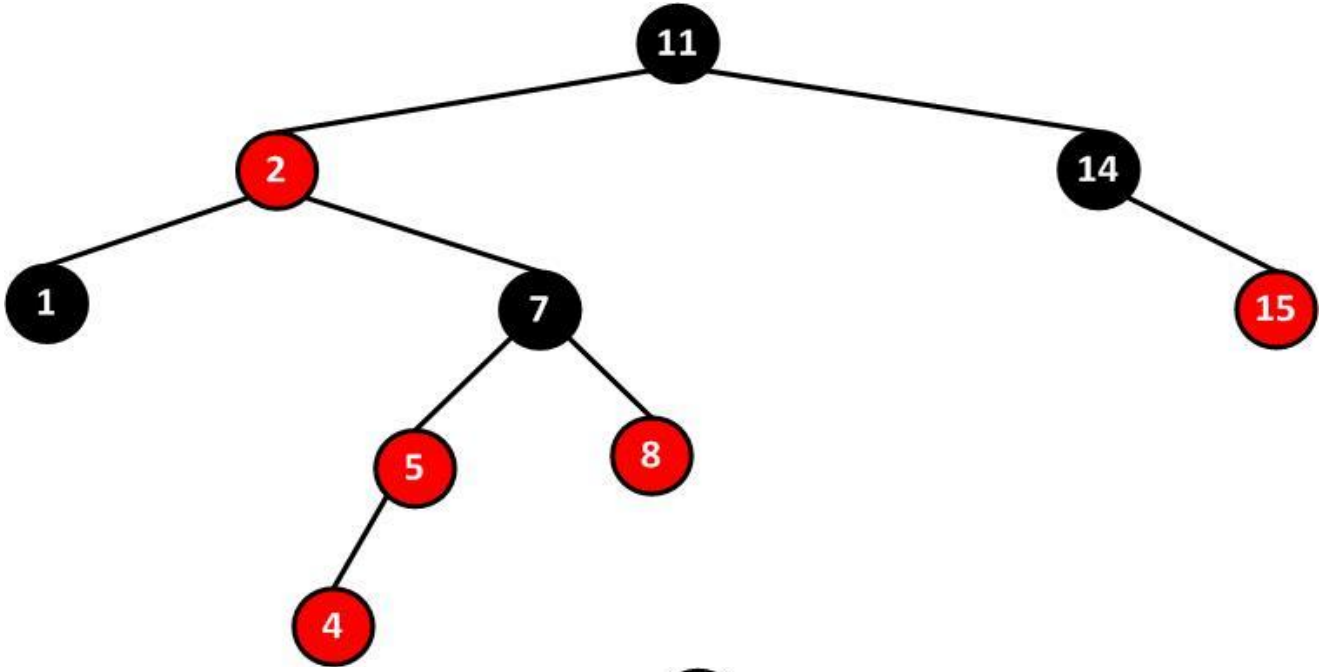
Простое перекрашивание избавляет нас от красно-красного нарушения. После перекраски нужно проверить "дедушку" нового узла (узел C), поскольку он может оказаться красным.



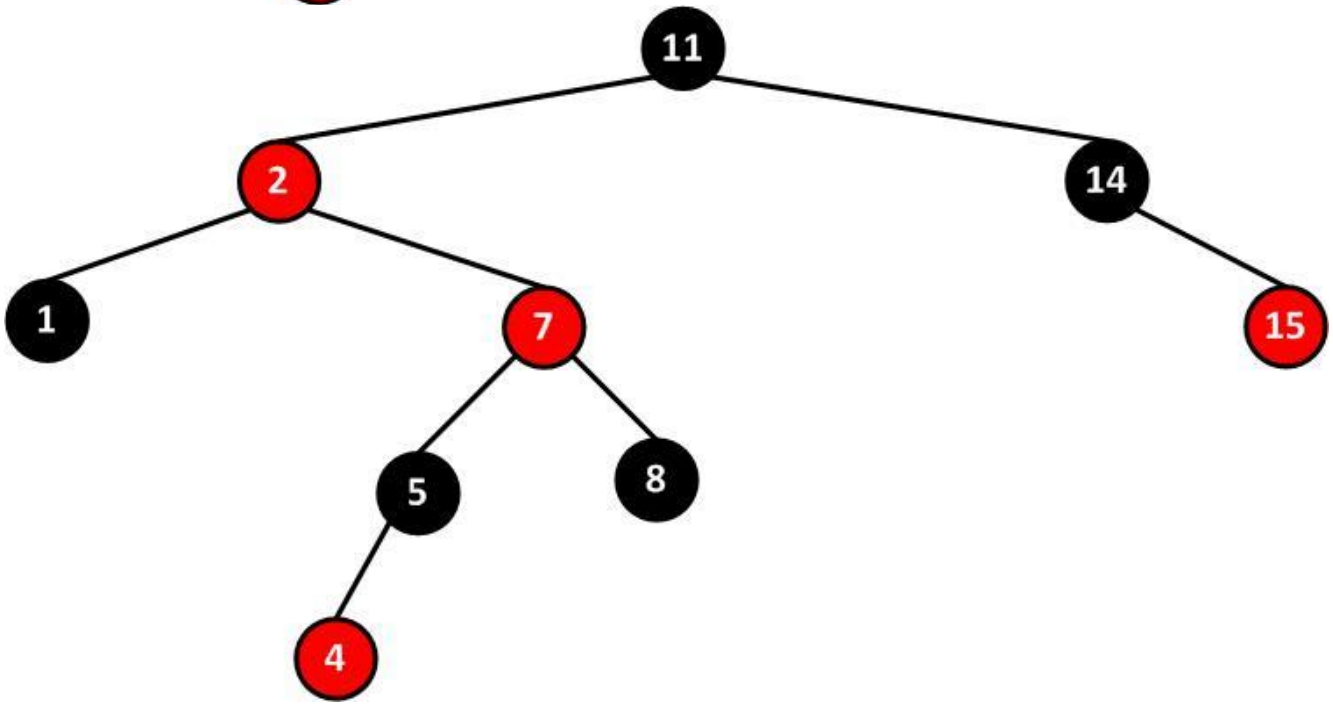
Красный предок, черный «дядя» - случаи 2 и 3



Пример



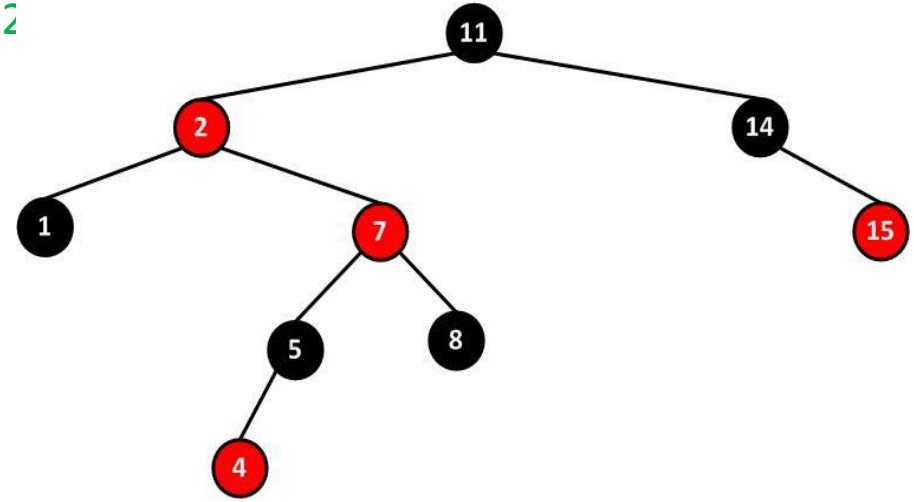
Случай 1

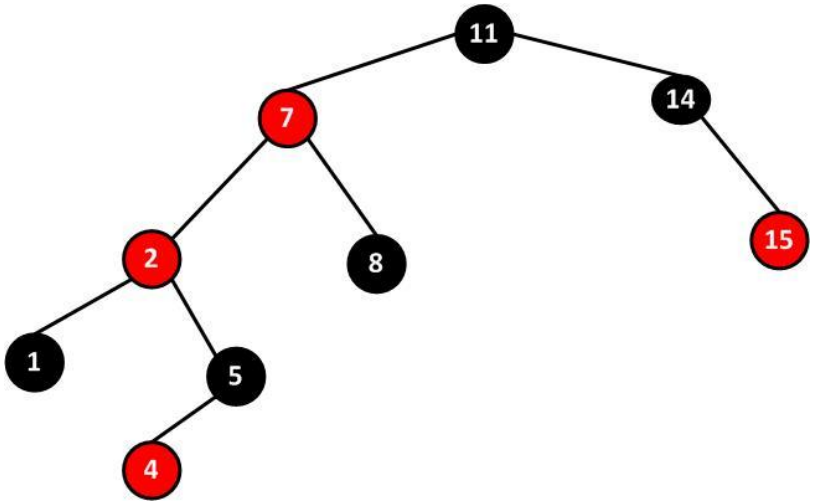


Случай 2

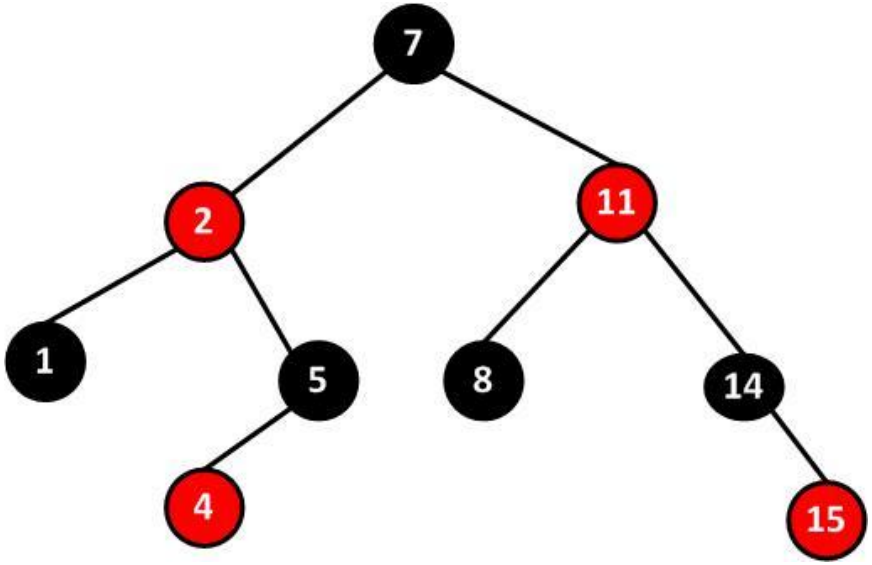
Пример, окончание

Случай 2





Случай 3



Удаление узла

Если удаляемый узел красный все правила сохраняются и все прекрасно.

Если же удаляемый узел черный, требуется значительное количество кода, для поддержания дерева частично сбалансированным.

Как и в случае вставки в красно-черное дерево, здесь также существует несколько различных случаев.

Сравнение с АВЛ-деревом

Пусть высота дерева *h*, минимальное количество листьев *N*.

Тогда:

- для АВЛ-дерева $N(h) = N(h-1) + N(h-2)$.

  Поскольку $N(0) = 1$, $N(1) = 1$, N(h) растёт как последовательность Фибоначчи, следовательно, $N(h) = \Theta(\lambda^h)$, где $\lambda = (\sqrt{5}+1)/2 \approx 1,62$

- для красно-чёрного дерева $N(h) \geq 2^{(h-1)/2} = \Theta(\sqrt{2}^h)$

Следовательно, при том же количестве листьев красно-чёрное дерево может быть выше АВЛ-дерева, но не более чем $\log \lambda / \log \sqrt{2} \approx 1,388$ раз.

Поиск, вставка, удаление

Поиск. Поскольку красно-чёрное дерево, в худшем случае, выше, поиск в нём медленнее, но проигрыш по времени не превышает 40%.

Вставка. Вставка требует до 2 поворотов в обоих видах деревьев. Однако из-за большей высоты красно-чёрного дерева вставка может занимать больше времени.

Удаление. Удаление из красно-черного дерева требует до 3 поворотов, в АВЛ-дереве оно может потребовать числа поворотов до корня. Поэтому удаление из красно-чёрного дерева быстрее, чем из АВЛ-дерева.

Память. АВЛ-дерево в каждом узле хранит высоту (целое число). Красно-чёрное дерево в каждом узле хранит цвет (1 бит). Таким образом, красно-чёрное дерево может быть экономичнее.

Splay деревья

Сплей-дерево (Splay-tree) — это двоичное дерево поиска.

Оно позволяет находить быстрее те данные, которые использовались недавно.

Относится к разряду сливаемых и самобалансирующихся деревьев.

Сплей-дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году.

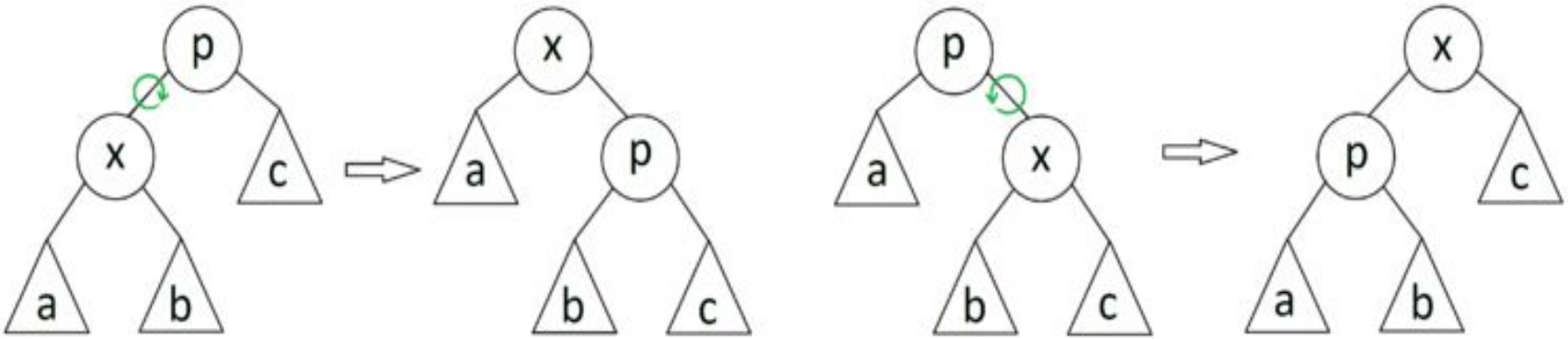
Основная операция Splay(Tree, x):

1. x становится корнем
2. $T_{real} = \Theta(depth(x))$
3. $T_{amort} = O(log(n))$

Splay(Tree, x)

Zig

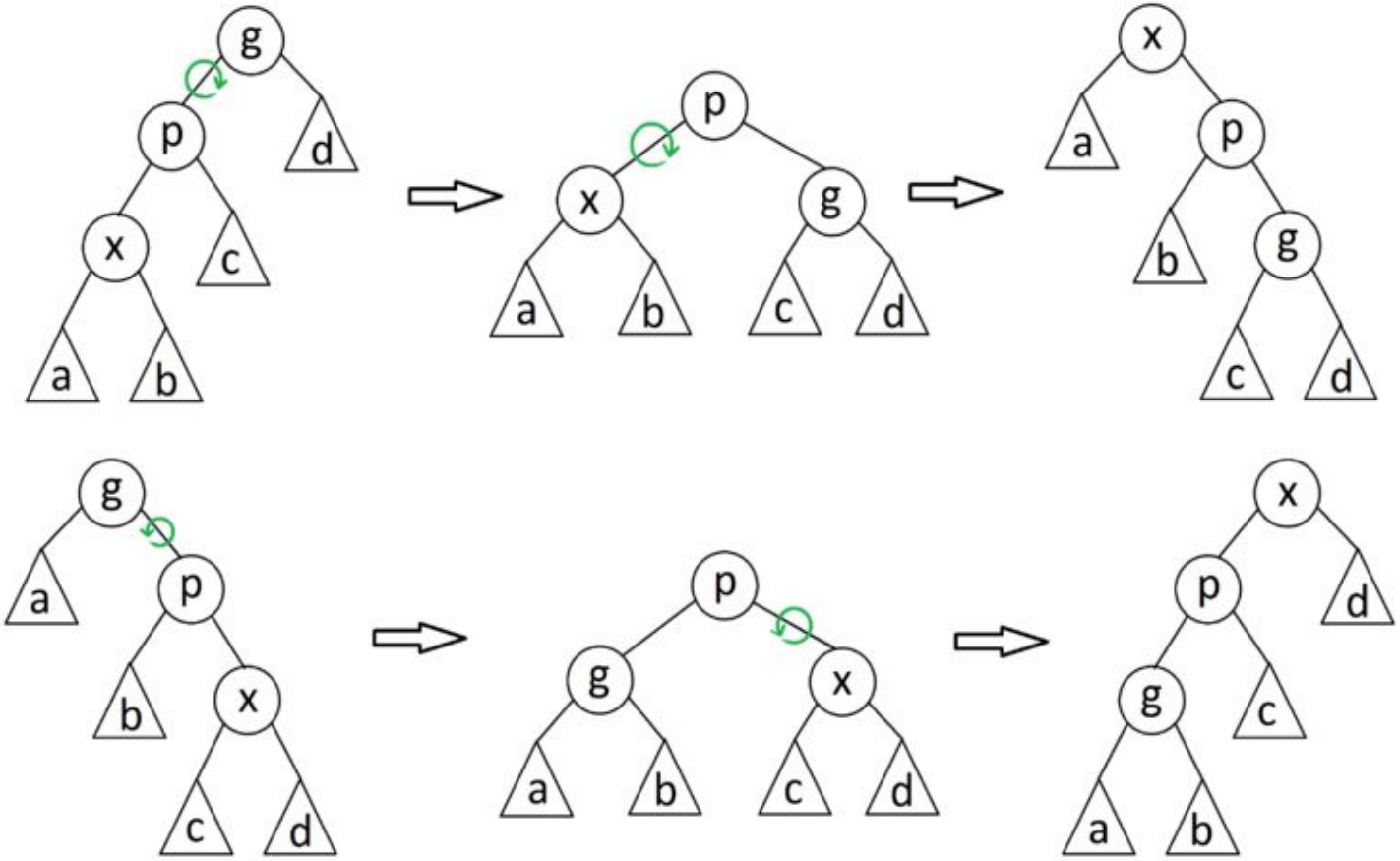
Если p — корень дерева с сыном x, то совершаем один поворот вокруг ребра (x, p), делая x корнем дерева. Данный случай является крайним и выполняется только один раз в конце, если изначальная глубина x была нечетной.



Splay(Tree, x)

Zig-Zig

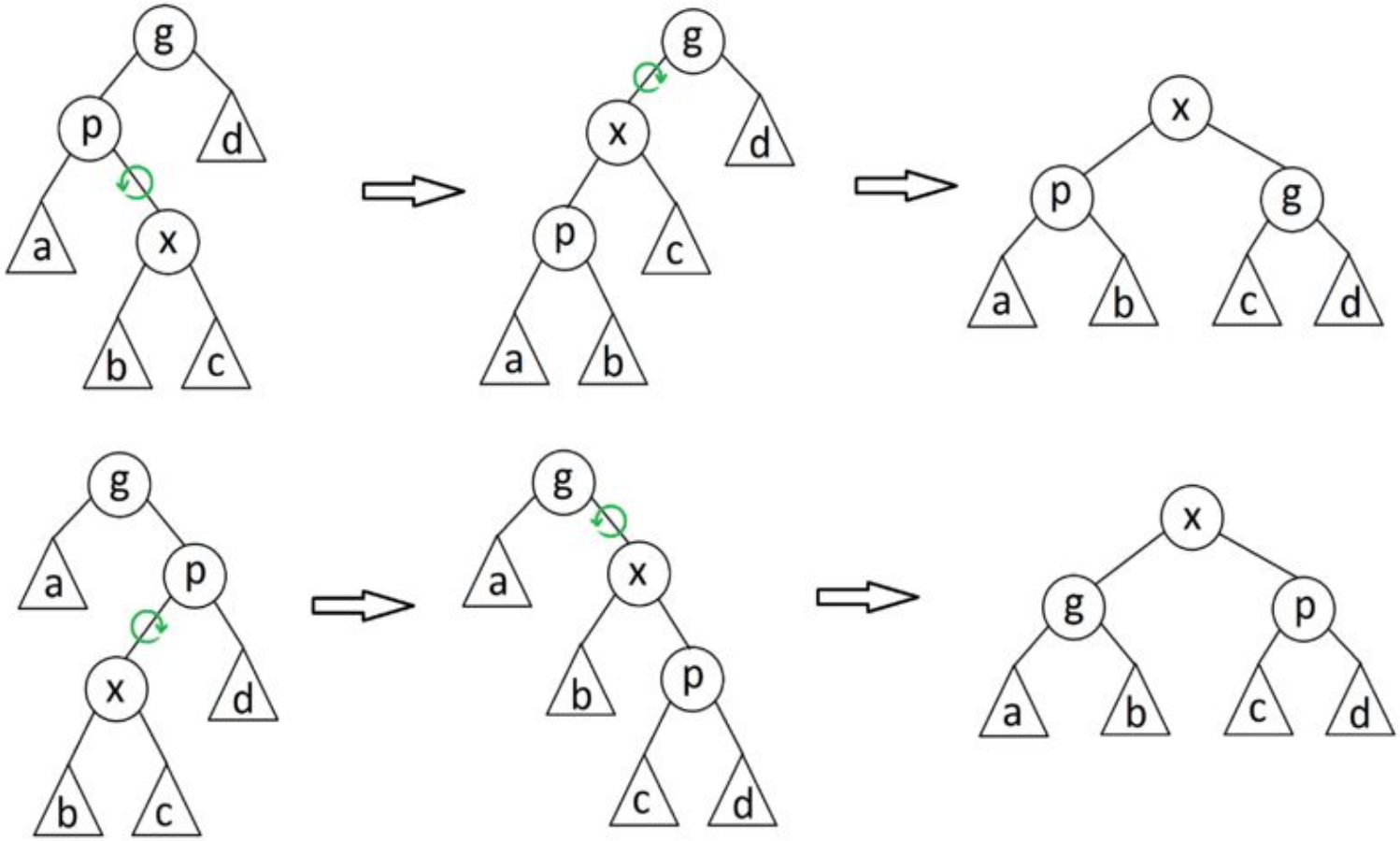
Если p — не корень дерева, а x и p — оба левые или оба правые дети, то делаем поворот ребра (p, g), где g отец p, а затем поворот ребра (x, p).



Splay(Tree, x)

Zig-Zag

Если *p* — не корень дерева и *x* — левый ребенок, а *p* — правый, или наоборот, то делаем поворот вокруг ребра (*x*, *p*), а затем поворот нового ребра (*x*, *g*), где *g* – бывший



Операции

Find (Tree, x)

Эта операция выполняется как для обычного бинарного дерева, только после нее запускается операция Splay.

Merge (Tree1, Tree2)

У нас есть два дерева $Tree1$ и $Tree2$, причём подразумевается, что все элементы первого дерева меньше элементов второго.

Запускаем Splay от самого большого элемента в дереве $Tree1$ (пусть это элемент i). После этого корень $Tree1$ содержит элемент i, при этом у него нет правого ребёнка. Делаем $Tree2$ правым поддеревом i и возвращаем полученное дерево.

Split (Tree, x)

Запускаем Splay от элемента x и возвращаем два дерева, полученные отсечением правого или левого поддерева от корня, в зависимости от того, содержит корень элемент больше или не больше, чем x.

Add (Tree, x)

Запускаем Split(Tree, x), который нам возвращает деревья $Tree1$ и $Tree2$, их подвешиваем к x как левое и правое поддеревья соответственно.

Remove(Tree, x)

Запускаем Splay от x элемента и возвращаем Merge от его детей.

Анализ Splay

Амортизационный анализ сплей-дерева проводится с помощью метода потенциалов.

Потенциалом рассматриваемого дерева назовём сумму рангов его вершин.

Ранг вершины — $r(x) = \log_2 w(x)$,
$w(x)$ — количество вершин в поддереве с корнем x.

Лемма

Амортизированное время операции Splay вершины x в дереве с корнем t не превосходит

$$3r(t) - 3r(x) + 1$$

Доказательство леммы

Пусть r' и r — ранги вершин после шага и до него соответственно, p — предок вершины x, а g — предок p, если есть.

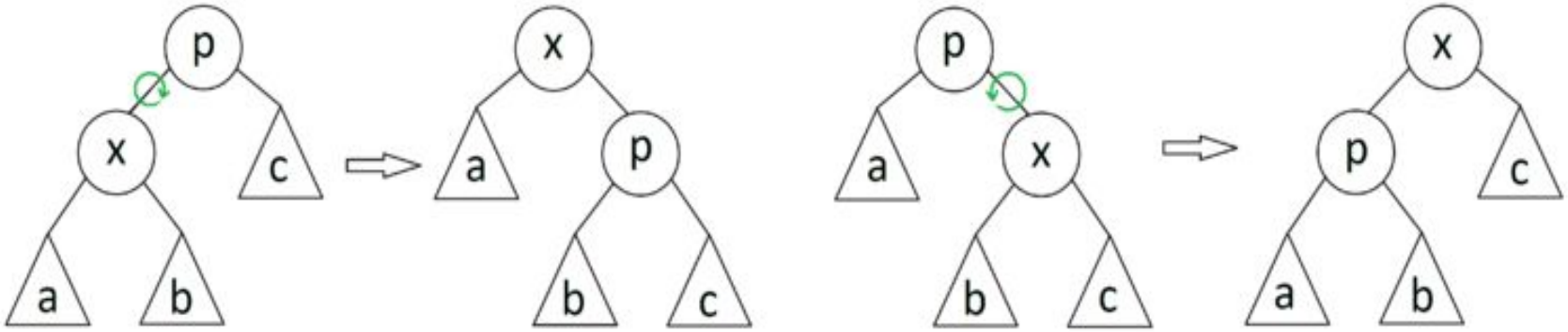
Zig. Выполнен один поворот ⇒

$$T_{amort} = 1 + r'(x) + r'(p) - r(x) - r(p)$$

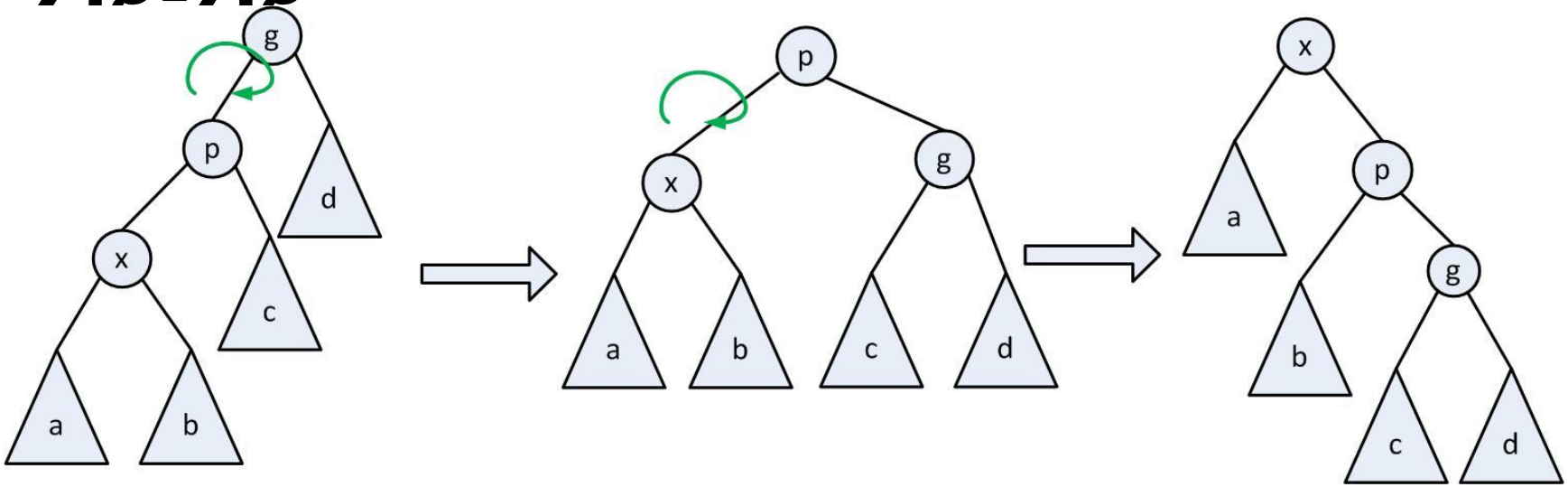
Ранг p уменьшился ⇒ $T_{amort} \le 1 + r'(x) - r(x)$.

Ранг x увеличился ⇒ $r'(x) - r(x) \ge 0$ ⇒

$$T_{amort} \le 1 + 3r'(x) - 3r(x) .$$



Доказательство леммы - Zig-zig



Выполнено два поворота $\Rightarrow T_{amort} = 2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g)$

$r'(x) = r(g) \Rightarrow T_{amort} = 2 + r'(p) + r'(g) - r(x) - r(p)$

$r(x) \le r(p) \Rightarrow T_{amort} \le 2 + r'(p) + r'(g) - 2r(x)$

$r'(p) \le r'(x) \Rightarrow T_{amort} \le 2 + r'(x) + r'(g) - 2r(x)$

Докажем, что эта сумма не превосходит $3(r'(x) - r(x))$, т.е., что $r(x) + r'(g) - 2r'(x) \le -2$.

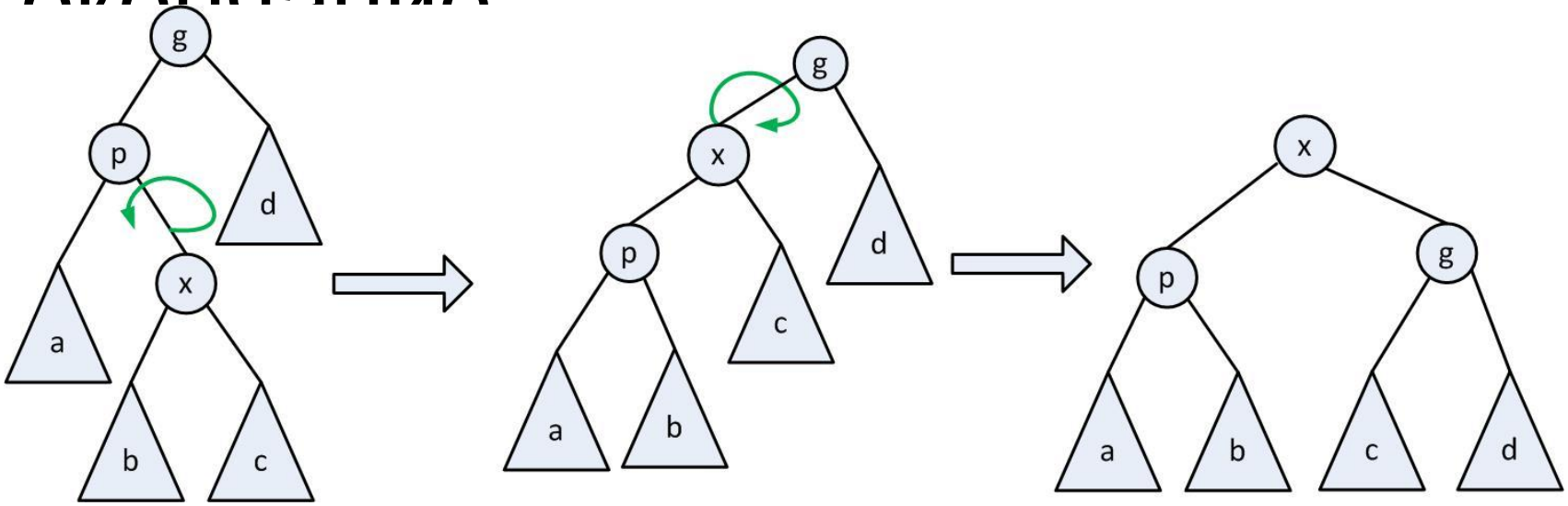
$$(r(x) - r'(x)) + (r'(g) - r'(x)) = log_2 \frac{w(x)}{w'(x)} + log_2 \frac{w'(g)}{w'(x)}$$

Из рисунка видно, что w′(g) + w(x) ≤ w′(x), значит, сумма выражений под логарифмами не превосходит единицы.

$\log_2 a + \log_2 b = \log_2 ab$, $a+b \le 1$, логарифм — функция возрастающая $\Rightarrow ab \le ¼ \Rightarrow$

$\log_2 ab \le -2$

Доказательство леммы - окончание



Zig-zag. Выполнено два поворота $\Rightarrow T_{amort} = 2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g)$

$r'(x) = r(g) \Rightarrow T_{amort} = 2 + r'(p) + r'(g) - r(x) - r(p)$

$r(x) \le r(p) \Rightarrow T_{amort} \le 2 + r'(p) + r'(g) - 2r(x) \le 2(r'(x) - r(x))$, – надо доказать

т.е. $r'(p) + r'(g) - 2r'(x) \le -2$.

$$(r(p) + r'(g)) - 2r'(x)) = log_2 \frac{w(p)}{w'(x)} + log_2 \frac{w'(g)}{w'(x)} \le -2$$

$$T_{amort} \le 2(r'(x) - r(x)) \le 3(r'(x) - r(x))$$

$$T_{splay} \le 3r(t) - 3r(x) + 1$$

$$T_{splay} \le 3\log_2 N - 3\log_2 w(x) + 1 = O(\log_2 N)$$