Деревья

Лекция 10

Обходы дерева

Обход дерева – это способ методичного исследования узлов дерева, при котором каждый узел проходится только один раз.

Обходы
- в глубину
- в ширину

Обходы деревьев в глубину

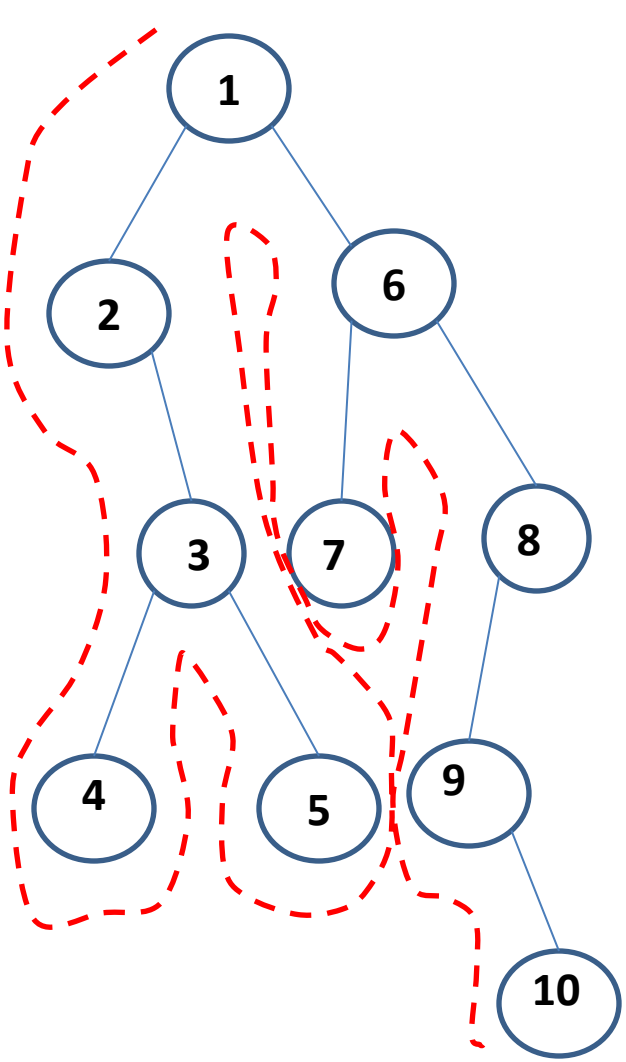
Пусть T – дерево, r- корень, $v_1, v_2, \ldots, v_n$ – сыновья вершины r.

1. Прямой (префиксный) обход:
   - посетить корень *r;*
   - посетить в прямом порядке поддеревья с корнями $v_1, v_2, \ldots, v_n$.

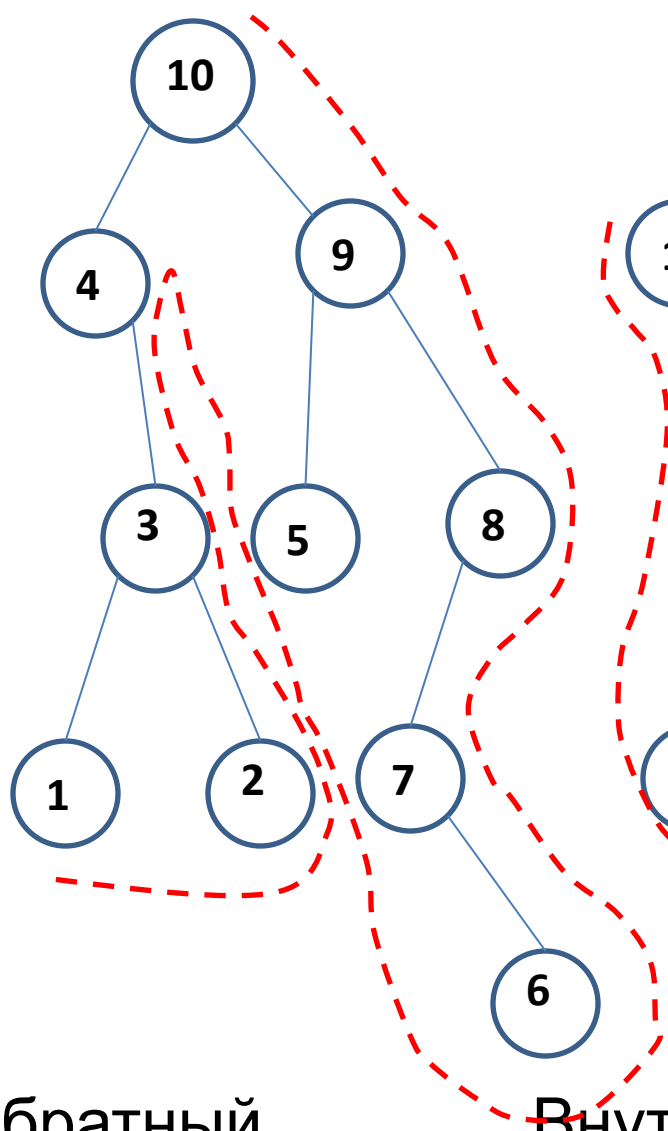
2. Обратный (постфиксный) обход:
   - посетить в обратном порядке поддеревья с корнями $v_1, v_2, \ldots, v_n$;
   - посетить корень *r.*

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*).

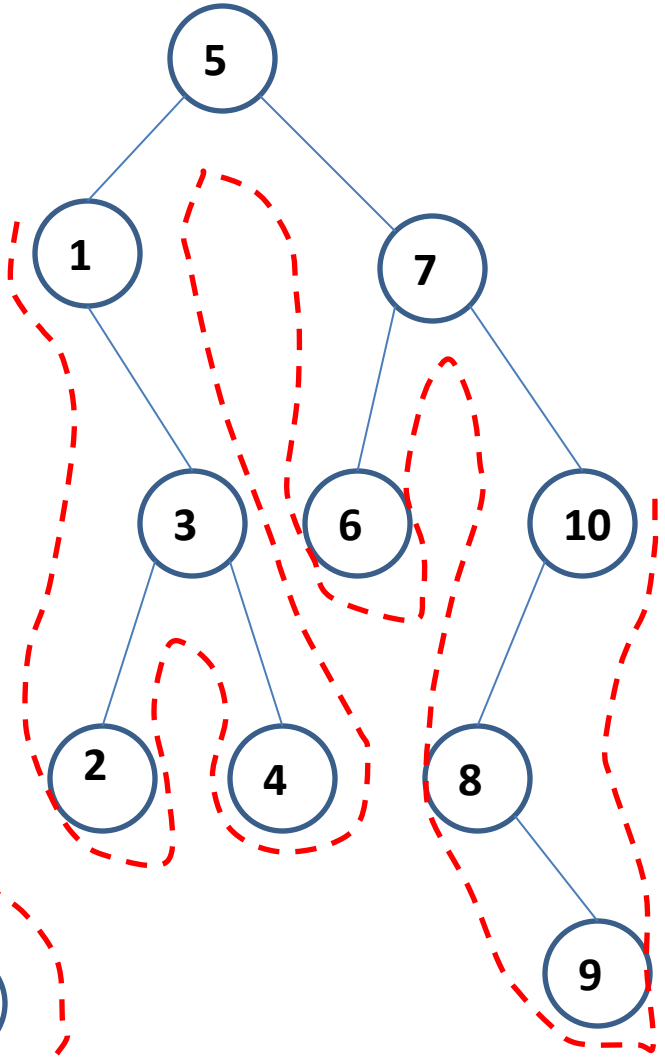
Обходы деревьев в глубину. Пример 1.



Прямой

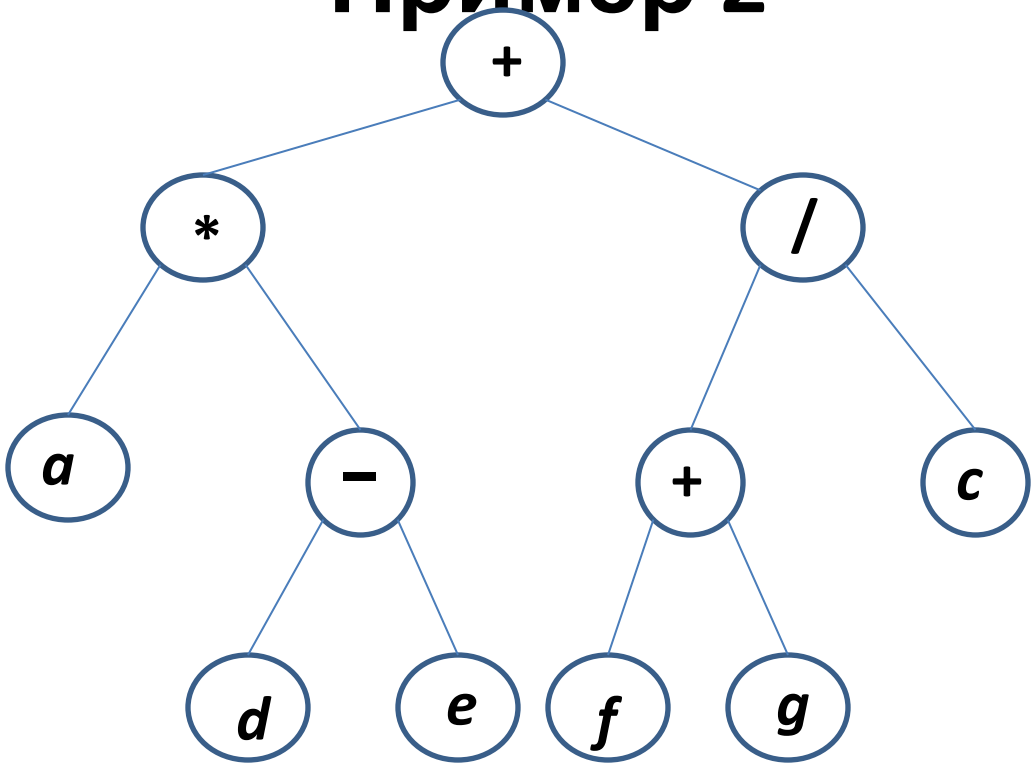


Обратный



Внутренний

Обходы деревьев в глубину. Пример 2



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

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

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

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.

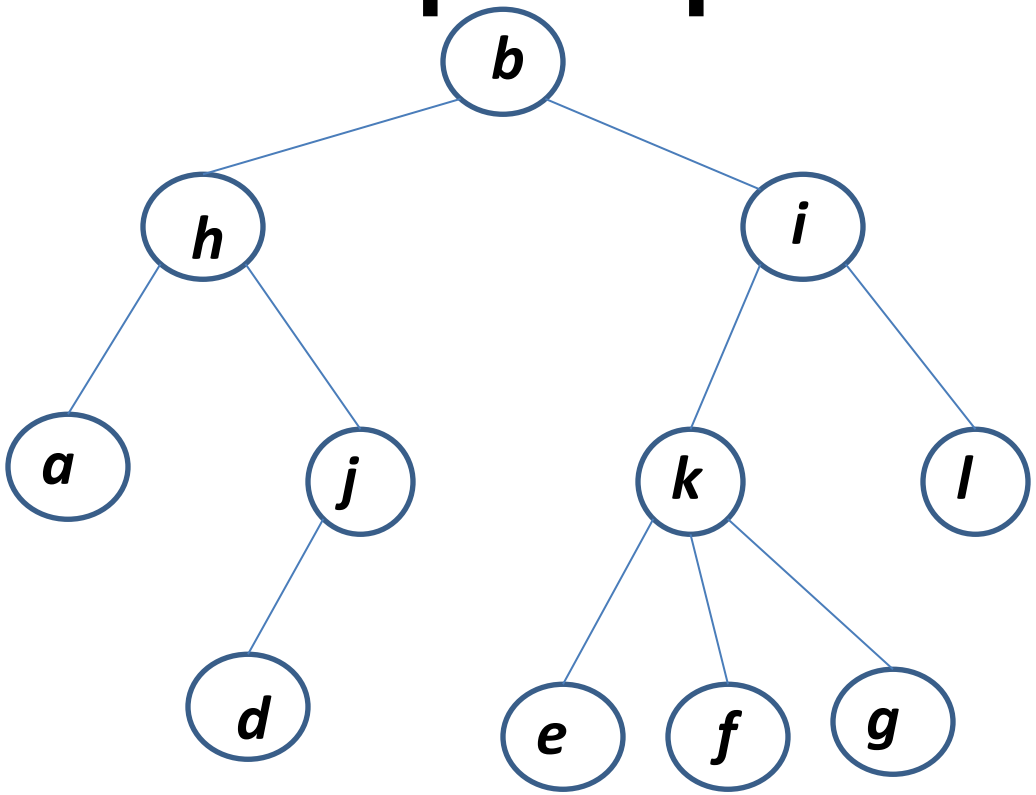
Шаг 1:

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

$$Lrep(T) = a\,(Lrep\,(T_1), Lrep\,(T_2), \ldots, Lrep\,(T_n))$$

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

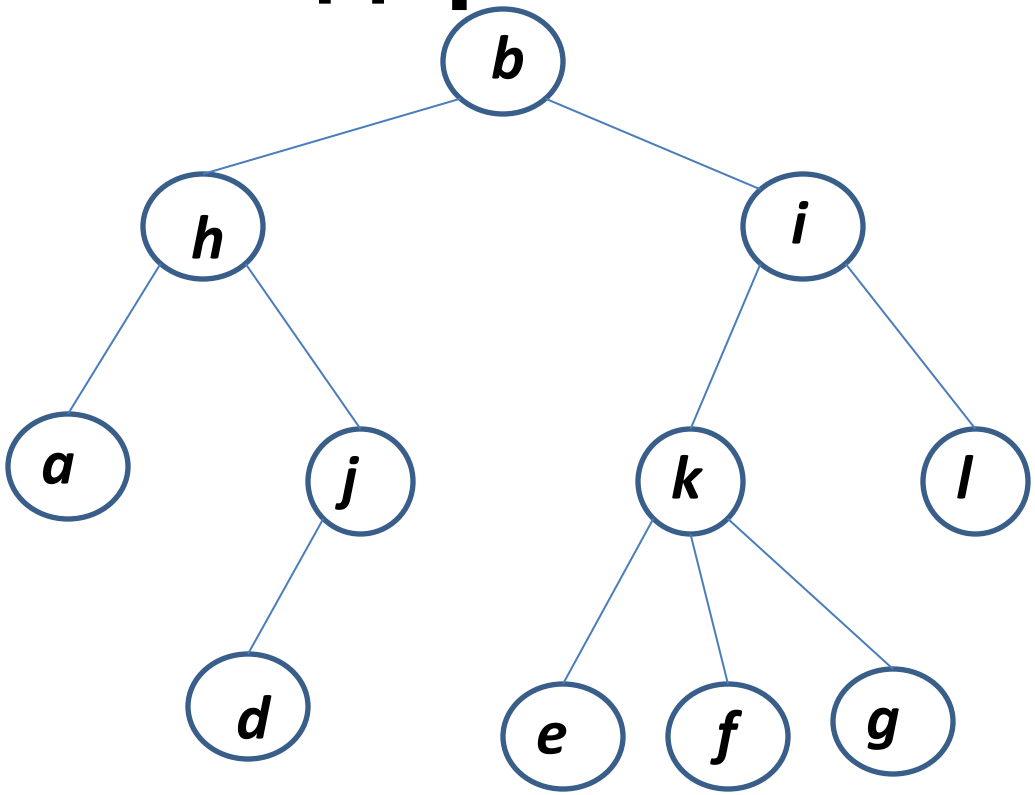
Определение. *Правое скобочное представление* *Rrep*(*T*) дерева *T:*

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

$$Rrep(T) = (Rrep(T_1), Rrep(T_2), \ldots, Rrep\,(T_n))a.$$

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

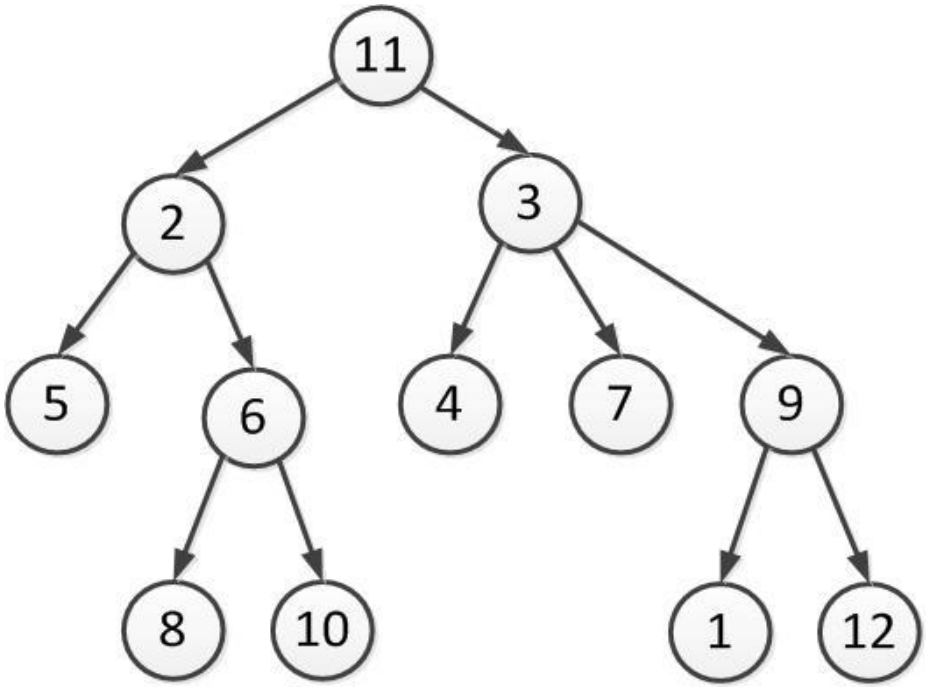
Скобочные представления деревьев



- *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

Бинарное дерево

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде.

При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Бинарное дерево

Бинарное дерево поиска– это один из способов сохранения отсортированного списка с возможностью постоянного добавления или удаления элементов из этого списка.

Бинарное дерево поиска является структурой данных, которое соблюдает три основных правила:

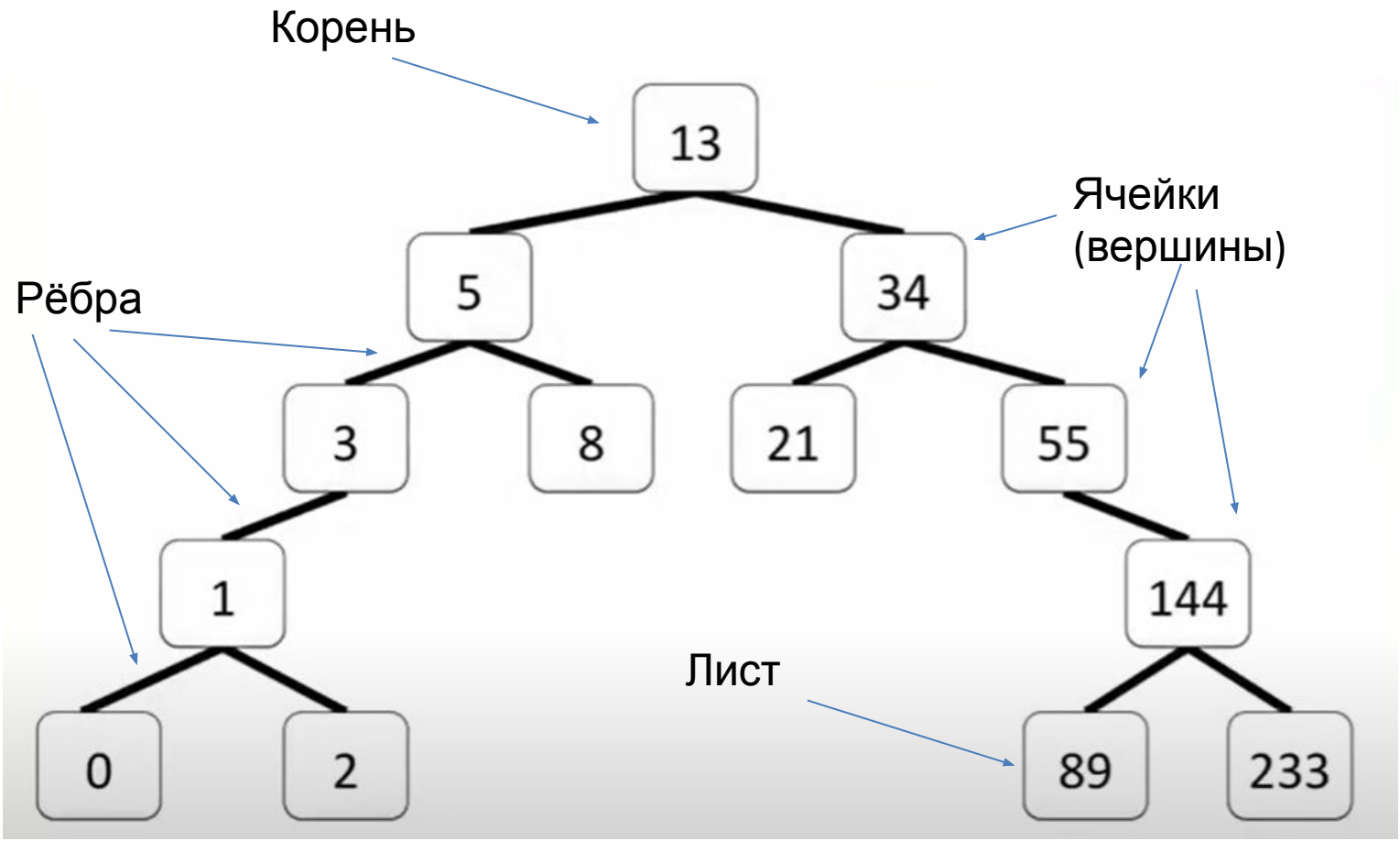
1. **Левое** поддерево любой вершины может иметь значение, которое **меньше** или равно значению вершины

2. **Правое** поддерево любой вершины может иметь значение, которое **больше** или равно значению вершины

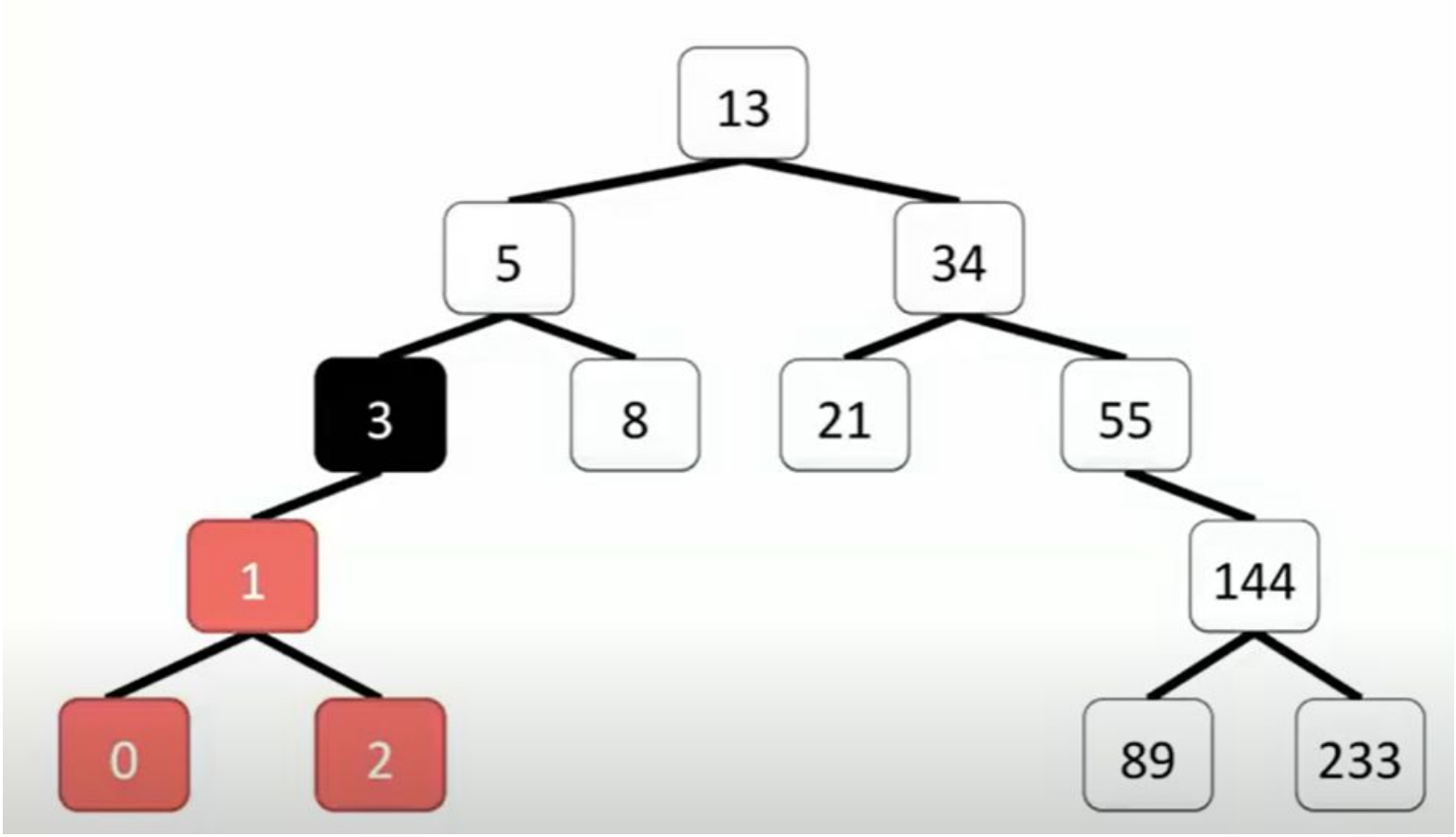
3. Правые и левые поддеревья всех вершин, так же являются бинарными деревьями поиска

Бинарное дерево ещё называют «Деревом программистов» и растёт оно сверху вниз.

Бинарное дерево

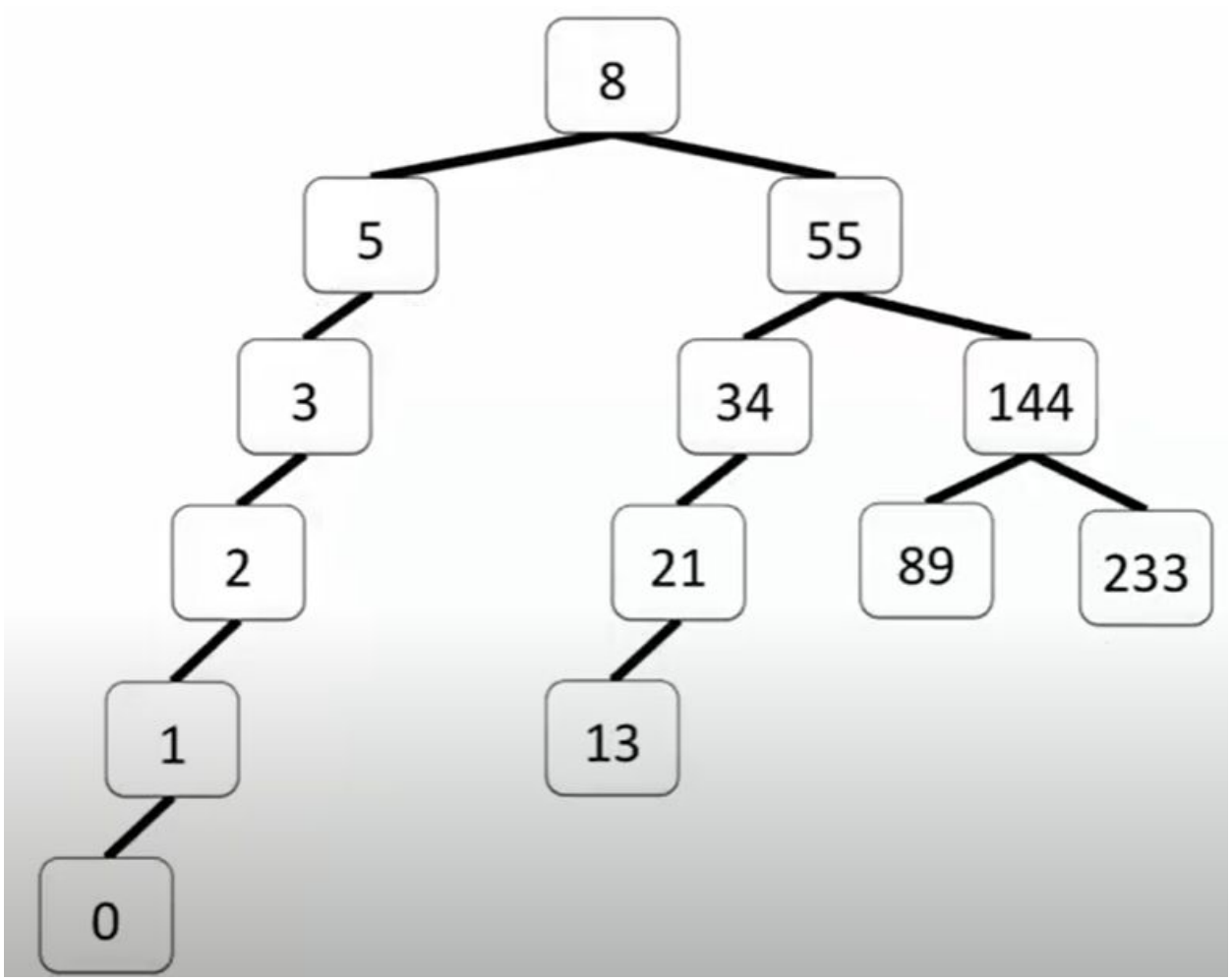


Бинарное дерево



Корень этого дерева имеет значение 13, ниже – находятся две вершины со значениями 5 и 34 . Поддерево являет собой дерево, которое состоит из частей целого дерева. Например левое поддерево вершины 3, это дерево созданное из вершин со значениями 0,1 и 2.

Бинарное дерево

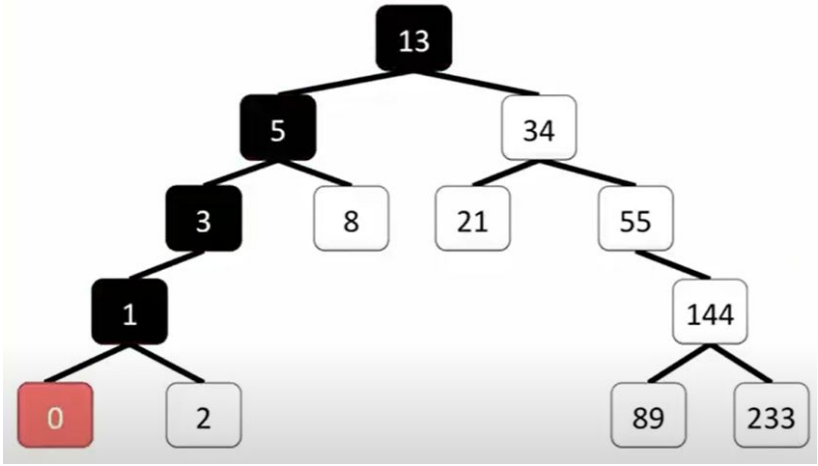


Если вернуться к основным правилам бинарного дерева поиска, то мы видим, что каждое поддерево отвечает всем трём основным правилам, а именно: левое поддерево состоит из значений, которые меньше или равны значению вершины. Правое поддерево состоит из значений, которые больше или равны вершине. И, наконец вершины обоих поддеревьев , так же являются бинарными деревьями

Бинарное дерево

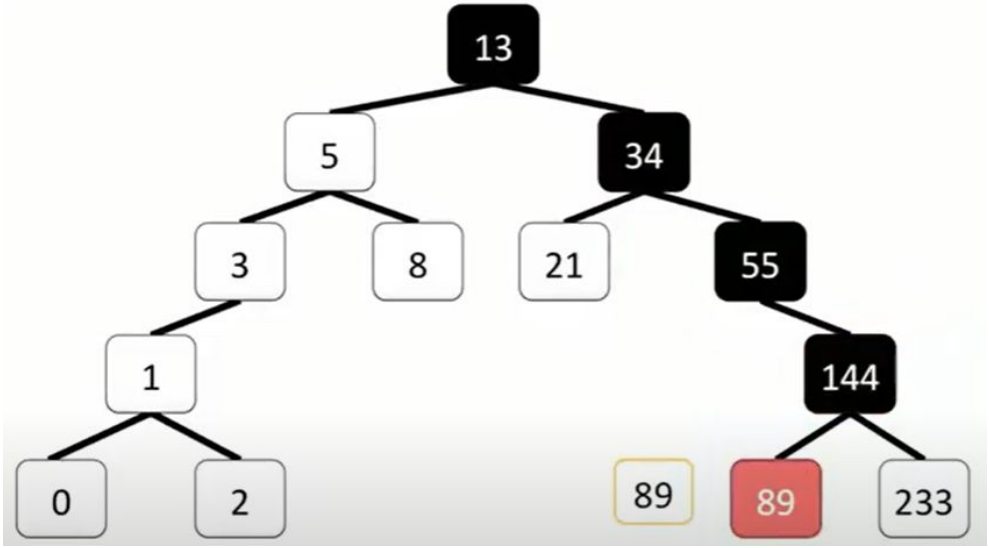
Существует много способов построения бинарного дерева поиска целых чисел.

С помощью бинарного дерева можно быстро найти максимальный и минимальный элемент массива



Минимальный элемент можно найти постоянно двигаясь влево, до тех пор, пока не закончатся вершины.

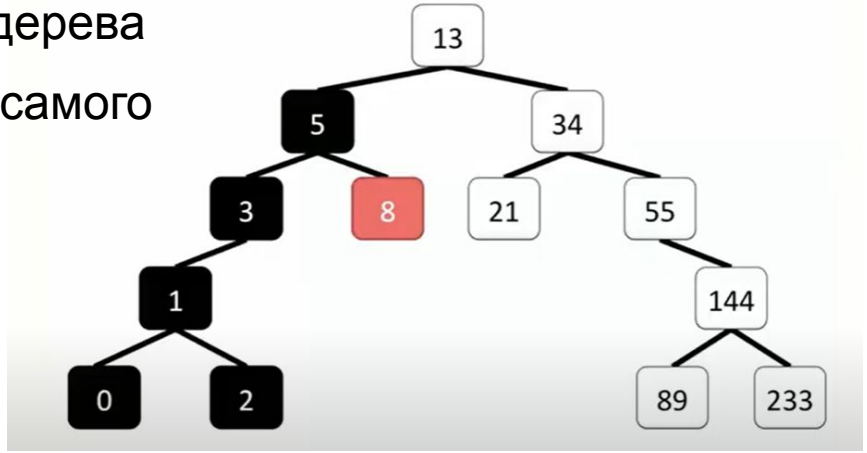
Максимальный элемент можно найти постоянно двигаясь вправо, снова пока не кончатся вершины



Чтобы найти любой элемент, например, 89. Нужно сравнить его со значением вершины и переходить в лево или вправо в зависимости больше или меньше число от значения вершины

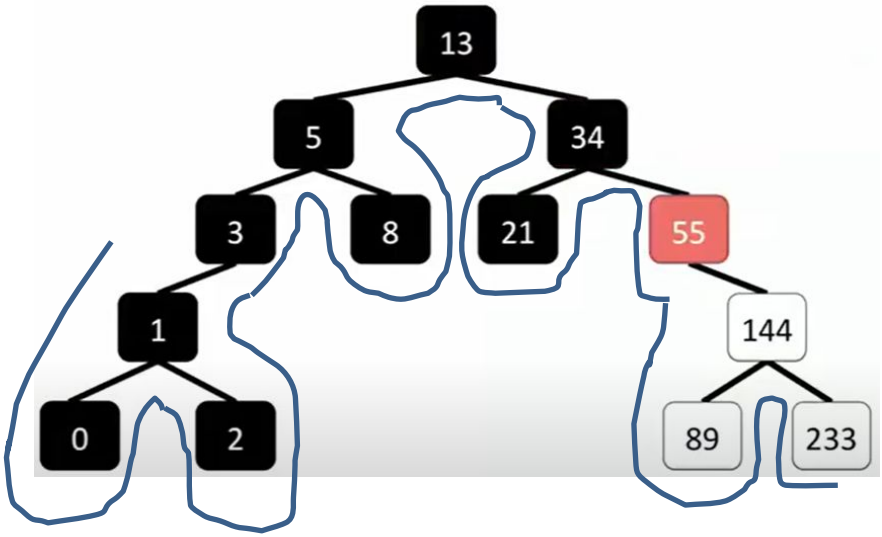
Бинарное дерево

Так же с помощью бинарного дерева поиска мы можем вывести все числа дерева, делая обход дерева во **внутреннем порядке (инфиксный обход)**. Это означает вывод всех чисел левого поддерева, потом числа из вершин, потом выводятся все числа правого поддерева



Сначала спускаемся с корня 13 влево до самого маленького значения (5 -> 3 -> 1 ->0)
От 0 начинаем подниматься:

0 1 2 3 5 8 ...



Теперь выводим значение корня 13, потом значения правого поддерева, но перед выводом вершины 34 сначала выводим его левое поддерево 21, потом 34, потом 55. Оно не имеет левого поддерева, по этому сразу переходим к его правому поддереву 144, но перед его выводом снова выводим его левое поддерево 89, потом 144 и 233

0 1 2 3 5 8 13 21 34 55 89 144 233

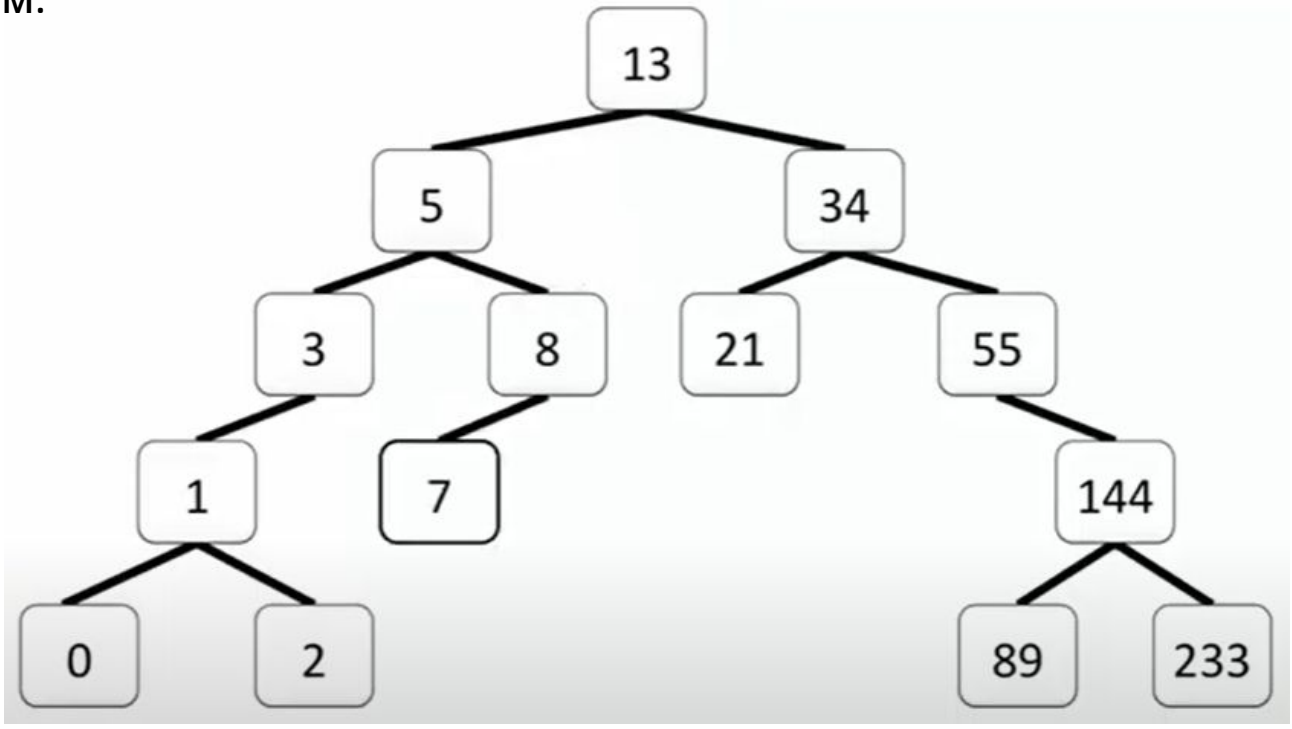
Бинарное дерево

Так же легко и добавить число к дереву. Нужно снова соблюдать три основных правила, просто добавляем значение туда где есть место.

Так же легко и добавить число к дереву. Нужно снова соблюдать три основных правила, просто добавляем значение туда где есть место.

Допустим, что мы хотим добавить 7.

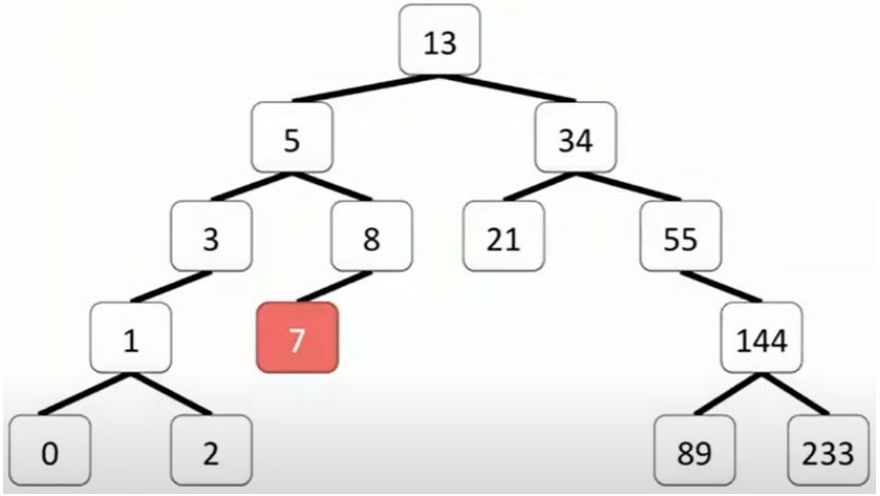
1) Т.к. 7 < 13 , то мы идём влево
2) Т.к. 7 > 5, то мы идём её правую ветвь
3) Т.к. 7< 8 и у этой вершины нет левой ветви, то ставим 7 тут. Левым поддеревом.

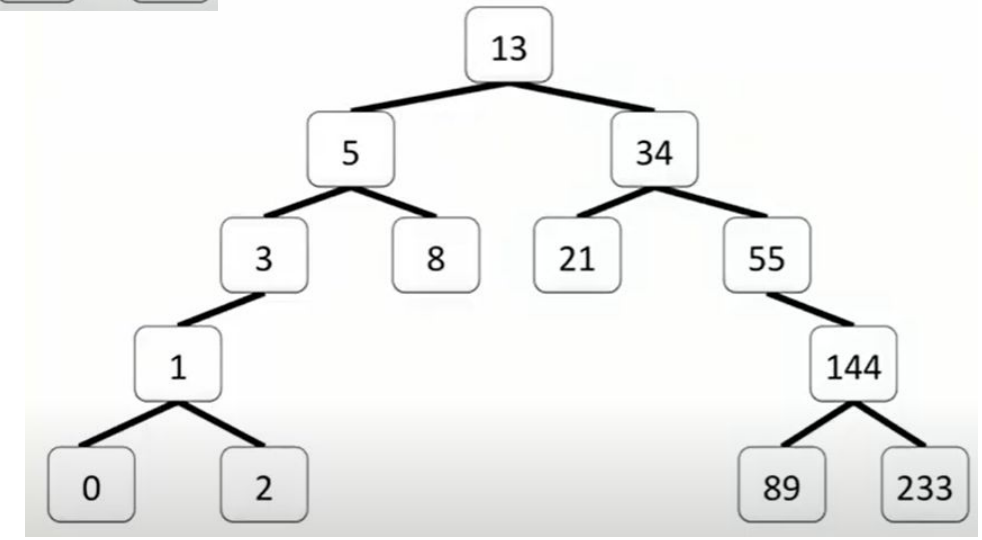


Бинарное дерево

Процесс удаления тоже возможен, но несколько сложнее. Но тут возможны три варианта:

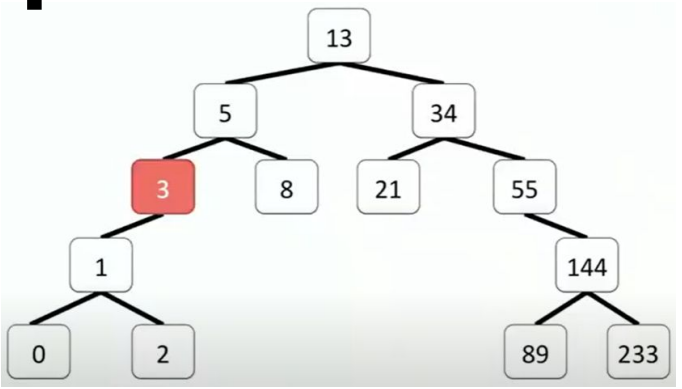
1) Самый простой способ – когда элемент не имеет поддеревьев. В этом случае мы просто находим этот элемент и его удаляем. Допустим удалим 7.

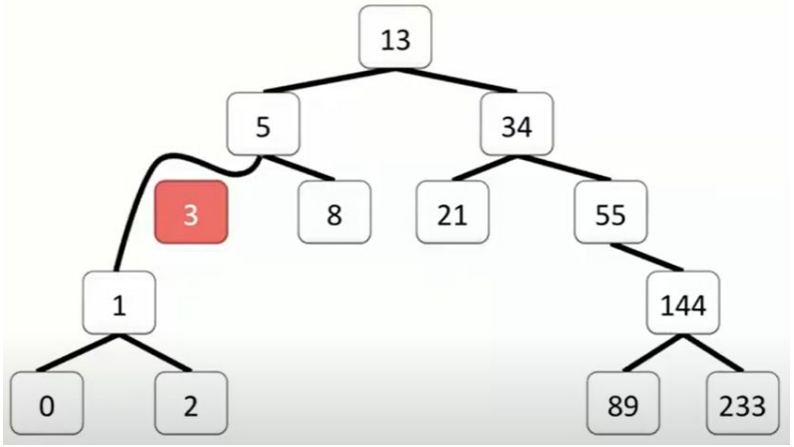




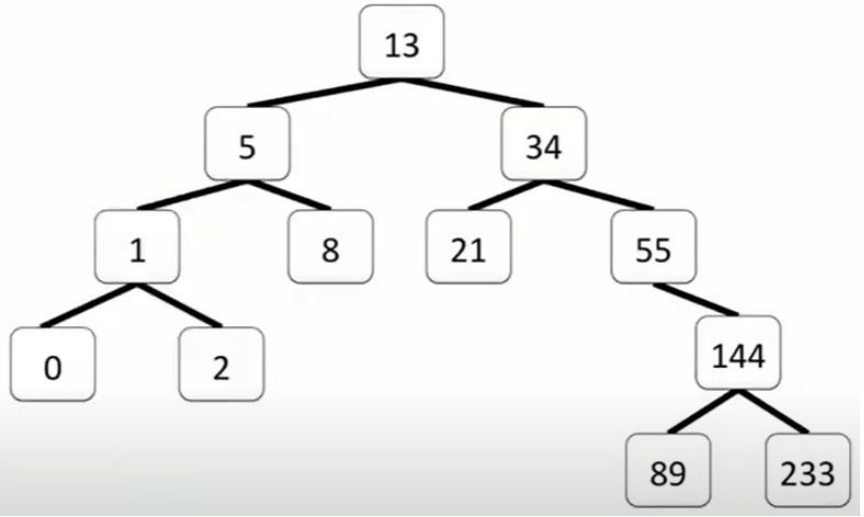
Бинарное дерево

2) Второй способ используется когда вершина имеет одно поддерево. ВЫ этом случае вершина удаляется, а вместо неё ставится поддерево только что удалённой вершины. Допустим, что хотим удалить число 3.



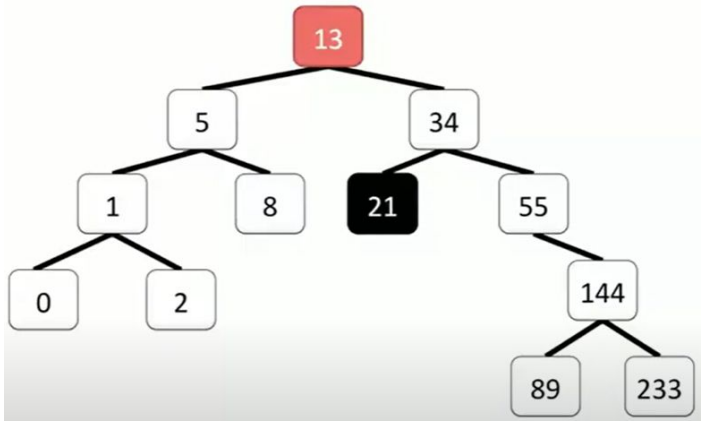


Берём поддерево этой вершины и крепим его к родительской вершине числа 3 (в данном случае – 1 крепится 5). Это легко сделать, т.к. по первому правилу все элементы левого поддерева 3

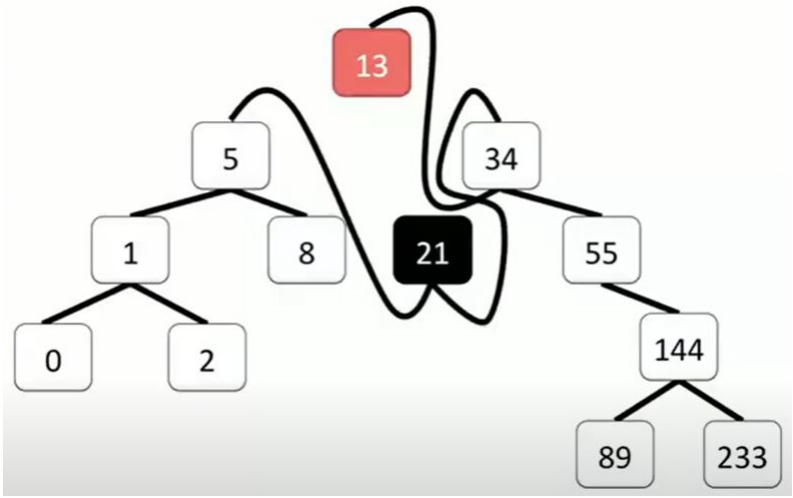


Бинарное дерево

3) Третий способ самый сложный это случай, когда удаляемая вершина имеет два поддерева.
Сначала нам нужно найти вершину, которая имеет следующее по возрастанию значение, потом мы меняем эти вершины местами, а потом удаляем ненужную вершину.
Обратите внимание, что у следующей по возрастанию вершины должно быть два поддерева.



Давайте удалим корневую вершину 13. Ищем следующее по величине значение. В нашем случае – это 21.



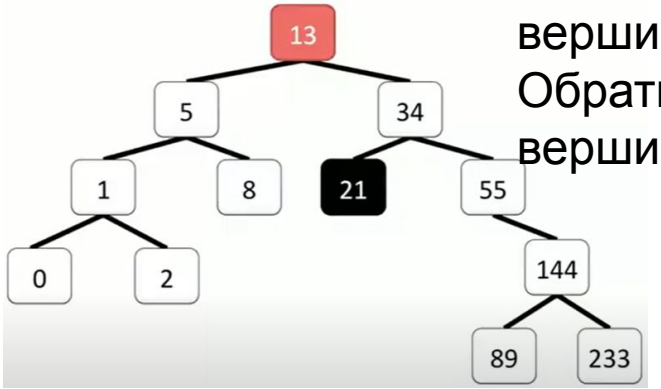
Потом меняем местами эти вершины, оставляя вершину 13 без подвершин.

Бинарное дерево

3) Третий способ самый сложный это случай, когда удаляемая вершина имеет два поддерева.

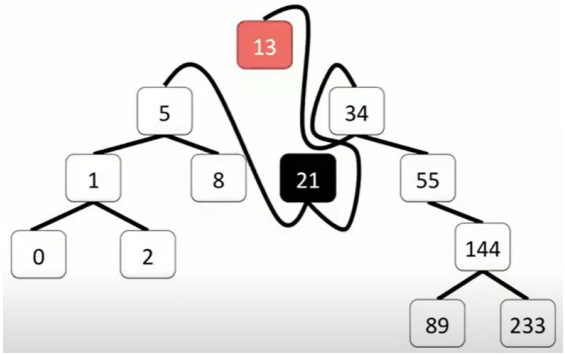
Сначала нам нужно найти вершину, которая имеет следующее по возрастанию значение, потом мы меняем эти вершины местами, а потом удаляем ненужную вершину.

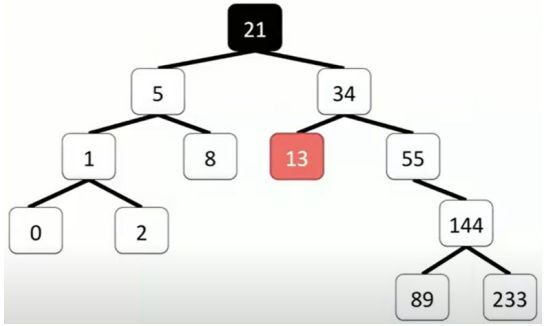
Обратите внимание, что у следующей по возрастанию вершины должно быть два поддерева.

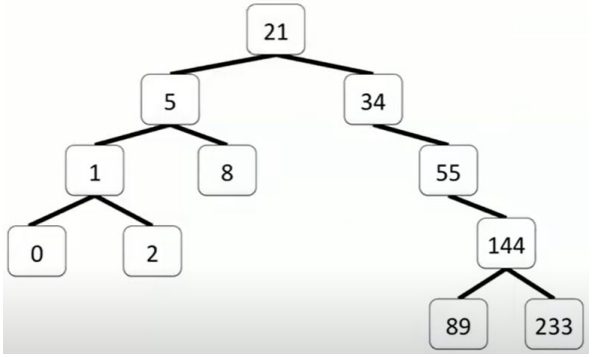


Давайте удалим корневую вершину 13. Ищем следующее по величине значение. В нашем случае – это 21.

Потом меняем местами эти вершины, оставляя вершину 13 без подвершин. И, наконец, удаляем число 13







Бинарное дерево

Таким образом сортировка бинарным деревом, за счёт дополнительной памяти, быстро решает вопрос с добавлением очередного элемента в отсортированную часть массива. Причём в роли отсортированной части массива выступает бинарное дерево. Дерево формируется буквально на лету при переборе элементов.

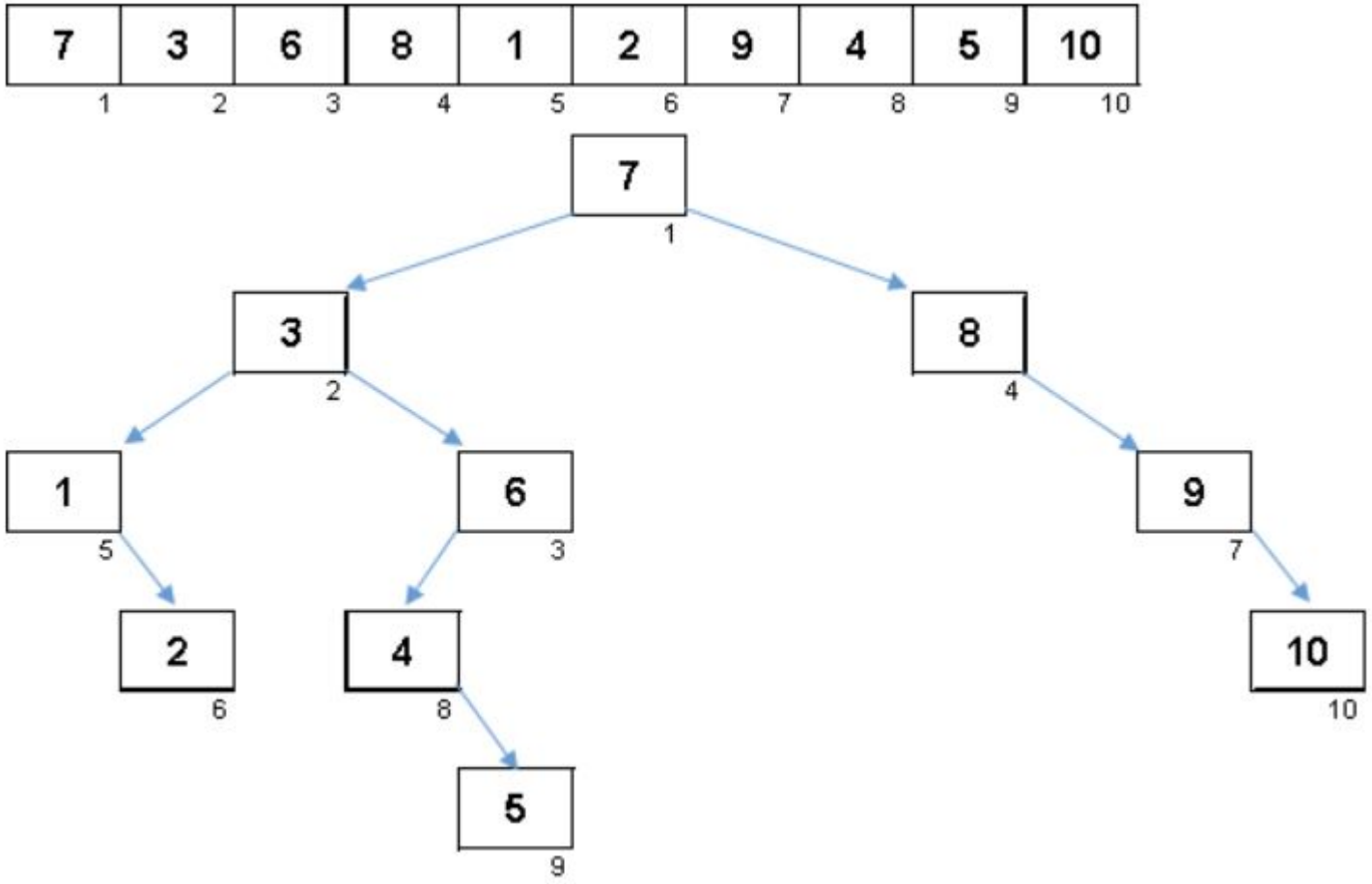
Элемент сравнивается сначала с корнем, а потом и с более вложенными узлами по принципу: если элемент меньше чем узел — то спускаемся по левой ветке, если не меньше — то по правой. Построенное по такому правилу дерево затем можно легко обойти так, чтобы двигаться от узлов с меньшими значениями к узлам с большими значениями (и таким образом получить все элементы в возрастающем порядке).

Основная загвоздка сортировок вставками (затраты на вставку элемента на своё место в отсортированной части массива) здесь решена, построение происходит вполне оперативно. Во всяком случае для освобождения точки вставки не нужно медленно передвигать караваны элементов как в предыдущих алгоритмах. Казалось бы, вот она, наилучшая из сортировок вставками. Но есть проблема.

Когда получается красивая симметричная ёлочка (так называемое идеально сбалансированное дерево), то вставка происходит быстро, поскольку дерево в этом случае имеет минимально возможную вложенность уровней. Но сбалансированная (или хотя бы близкая к таковой) структура из случайного массива получается редко. И дерево, скорее всего, будет неидеальное и

Бинарное дерево

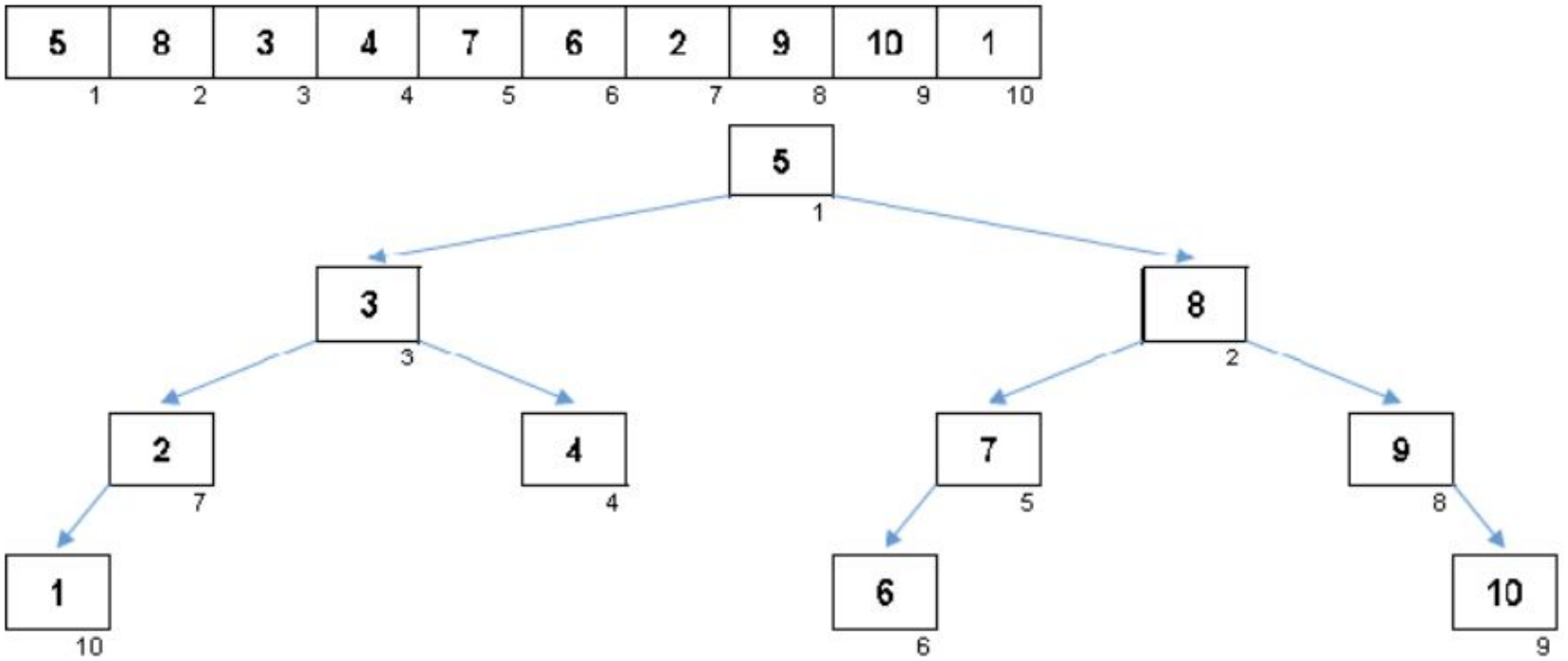
Рандомный массив со значениями от 1 до 10. Элементы в таком порядке генерируют несбалансированное двоичное дерево:



Бинарное дерево

Дерево мало построить, его ещё нужно обойти. Чем больше несбалансированности — тем сильнее будет буксовать алгоритм по обходу дерева. Тут как скажут звёзды, рандомный массив может породить как уродливую корягу (что более вероятно) так и древовидный фрактал.

Значения элементов те же, но порядок другой. Генерируется сбалансированное двоичное дерево:



Проблему несбалансированных деревьев решает сортировка выворачиванием, которая использует особую разновидность бинарного дерева поиска — splay tree. Это замечательное древо-трансформер, которое после каждой операции перестраивается в сбалансированное состояние. Но это уже – другая

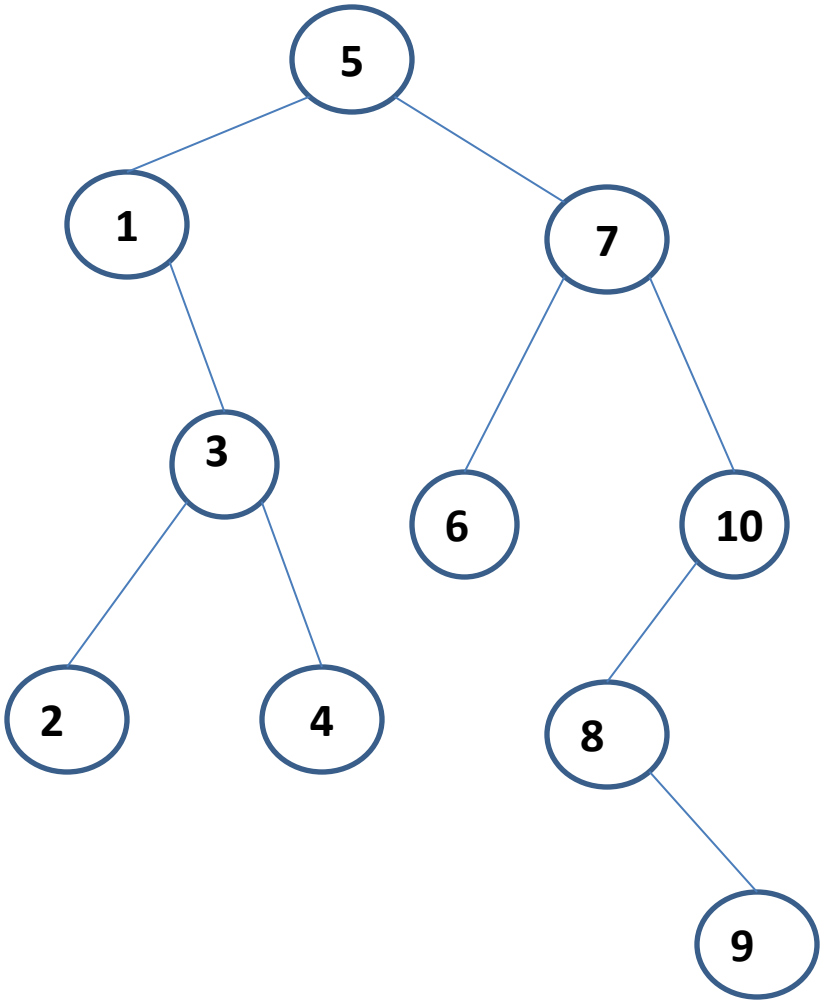
Дерево двоичного поиска

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*.

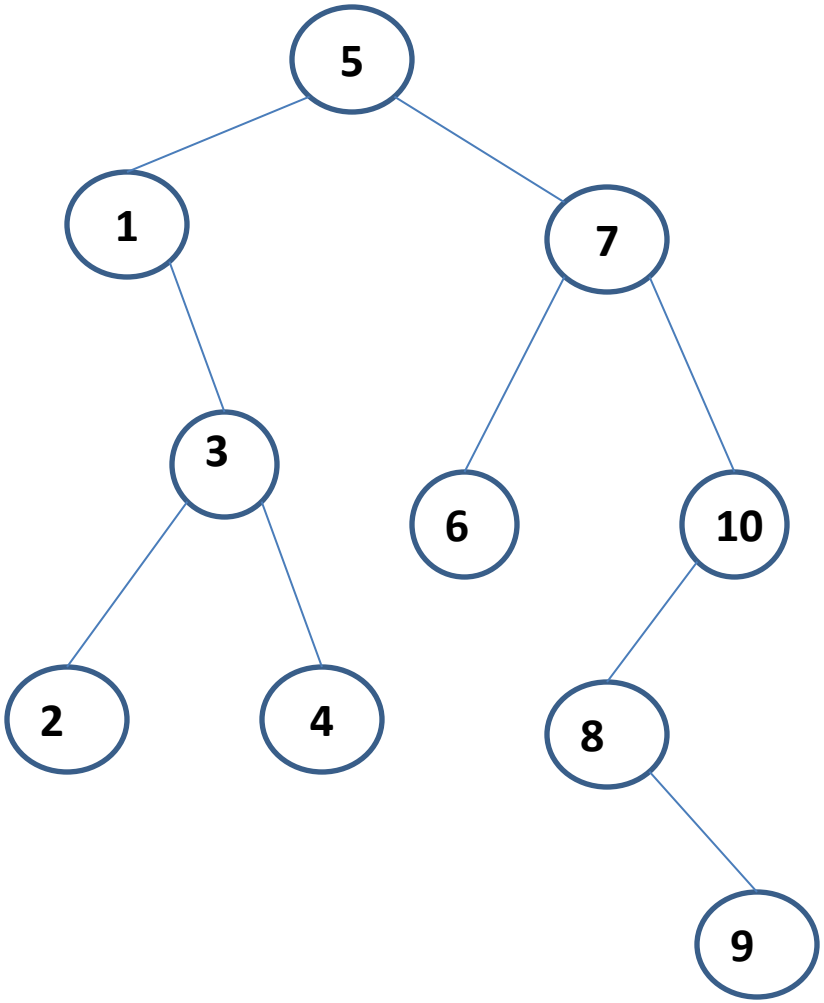
Выход: *true* если $a \in S$, *false* - в противном случае.

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;
}
```

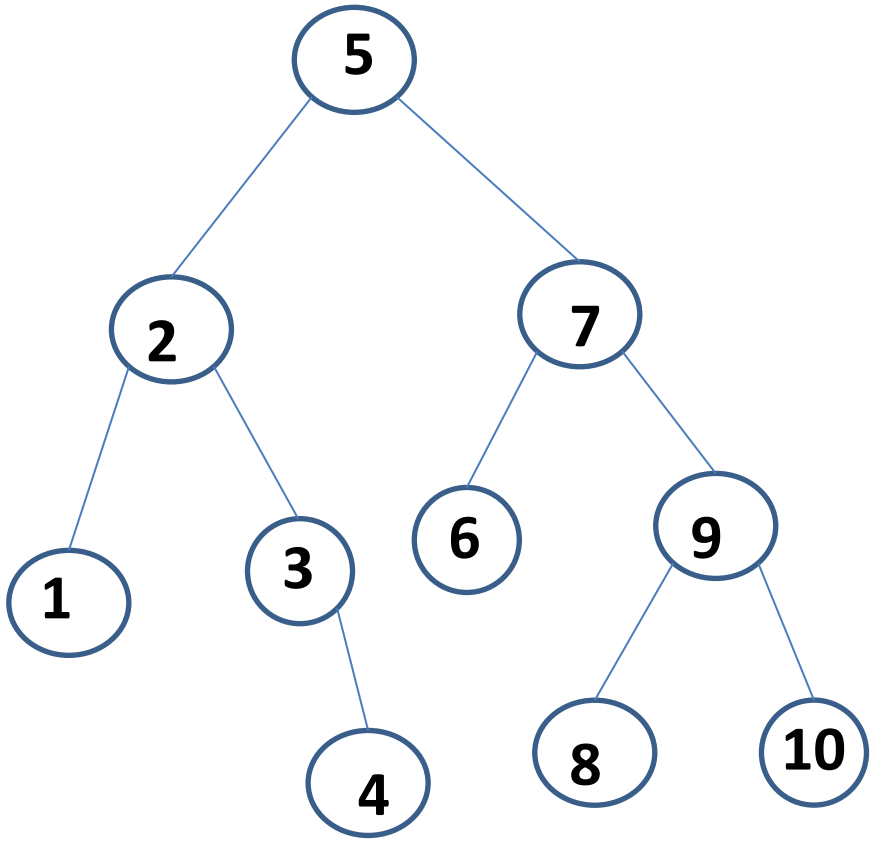
Пример построения дерева двоичного поиска

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



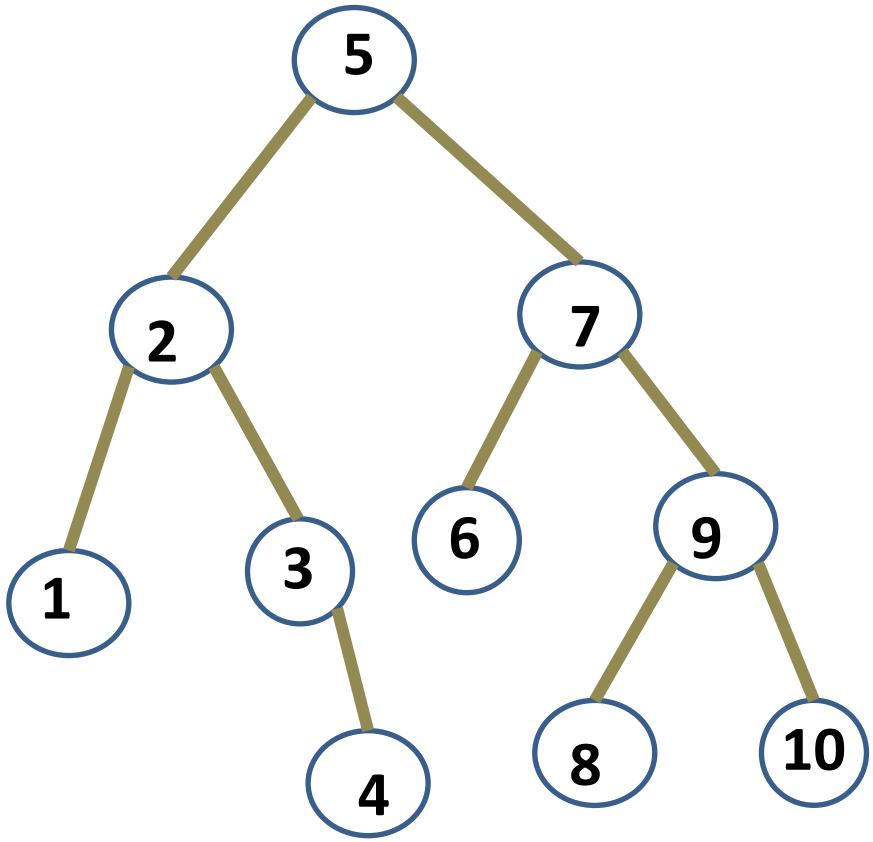
Операции нахождения минимума и максимума

```
T         – дерево
root [T]  – корень дерева
left[v]   – левый сын v
right[v]  – правый сын v
p[v]      – отец вершины v

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

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



Операция поиска следующего

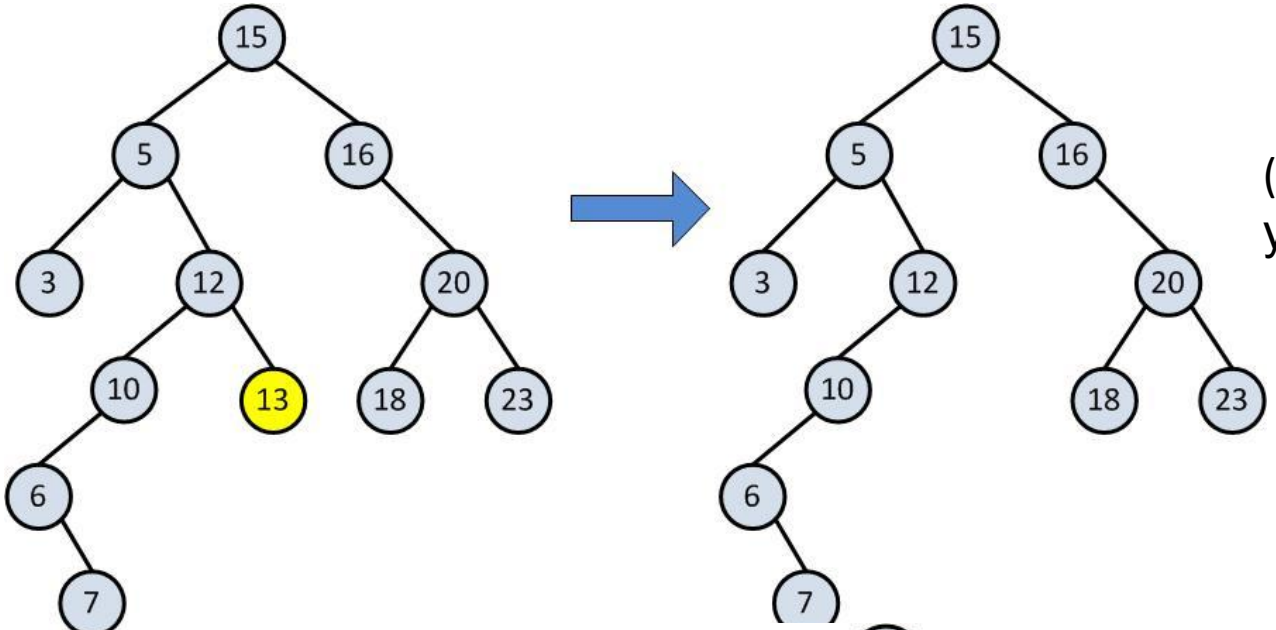
```
successor (x)
  if right[x] ≠ NIL then
   return min (right [x])
  y ← p[x]
  while y ≠ NIL and x = right [y]
   do
      x ← y
      y ← p[y]
  return y
```



Операция удаления элемента из дерева поиска

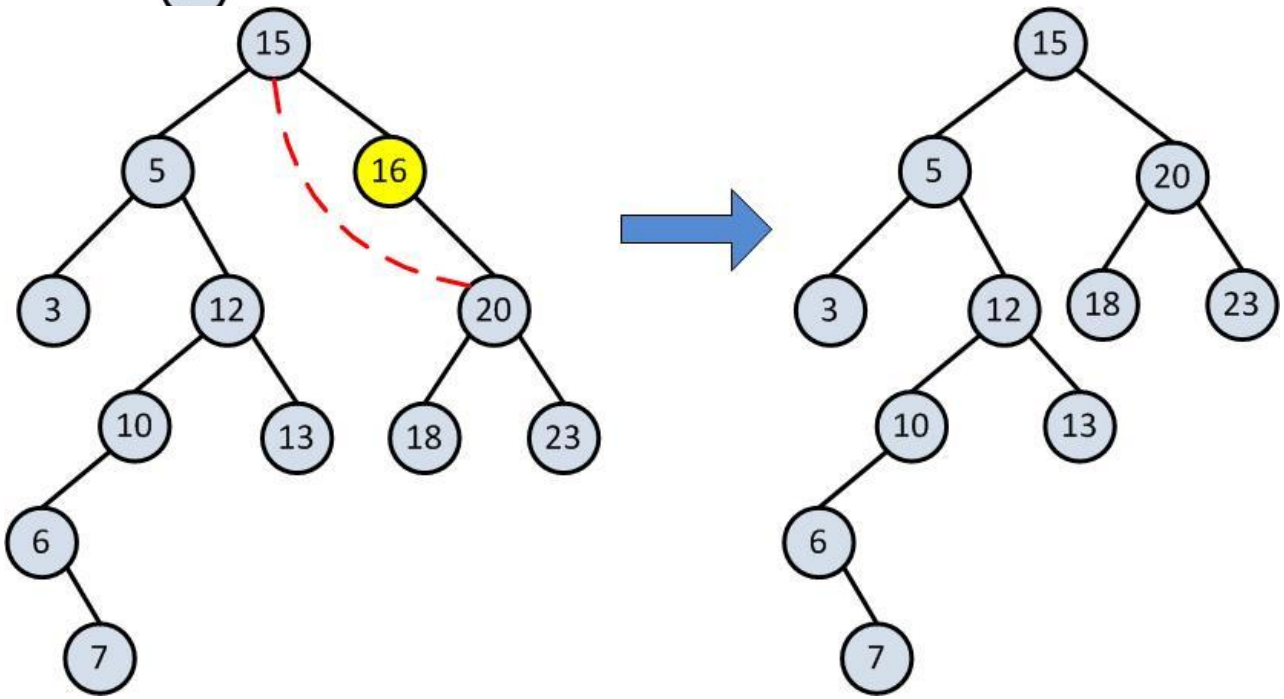
```
Delete(T, z)
  if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← successor (z)
  if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
  if x ≠ NIL
   p[x] ← p[y]
  if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
         then left[p[y]] ← x
         else right[p[y]] ←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

Пример удаления элемента из дерева поиска

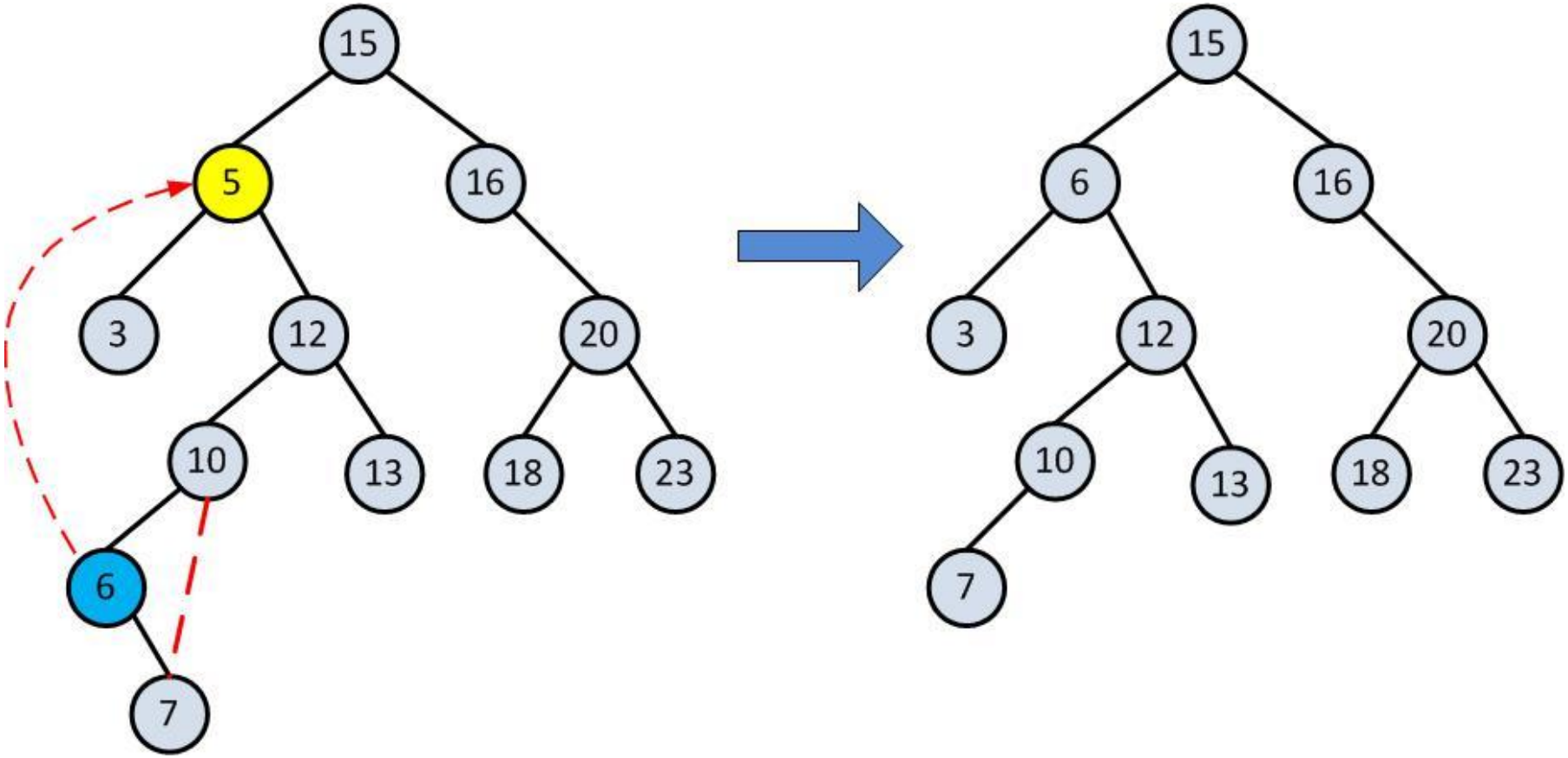


Первый случай (самый простой) – удаляем лист

Второй случай – удаляем промежуточный элемент у которого только правый/левый потомок



Пример удаления элемента из дерева поиска (окончание)



Третий случай – удаляем узел у которого есть оба потомка. В этом случае, находим минимальный элемент в правом поддереве этого узла и перестраиваем узлы.

Теорема

Среднее число сравнений, необходимых для вставки n случайных элементов в дерево двоичного поиска, пустое в начале, равно $O(n \log_2 n)$ для $n \geq 1$.

(без доказательства).

Максимальное число сравнений $O(n^2)$ – для вырожденных деревьев.

Сбалансированные деревья

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

Дерево называется **идеально сбалансированным**, если все его уровни, за исключением, может быть, последнего, полностью заполнены. В бинарном дереве полностью заполненный уровень n содержит 2^n узлов. В идеально сбалансированном дереве при полностью заполненных уровнях на последнем уровне находится больше половины узлов дерева.

Если дерево поиска близко к сбалансированному, то даже в худшем случае за время порядка O(log_2 n) в нем можно:

- Найти вершину (узел) с заданным значением или выяснить, что такой вершины нет.
- Включить (добавить) новую вершину.
- Исключить (удалить) вершину.
- Стандартные операции:
- -поиск максимального/минимального элемента
- -поиск предыдущего/следующего по величине элемента
- -удаление элемента
- -различные варианты обхода дерева – для его вывода, записи в файл или удаления

Сбалансированные деревья

АВЛ–дерево (или AVL tree) — древовидная структура данных с быстрым доступом к информации. Она представляет собой бинарное дерево — иерархическую схему из вершин и путей между ними, где у одной вершины может быть не более двух потомков. АВЛ-дерево – модифицированное дерево двоичного поиска, у него оптимизирована структура - это структура данных, изобретенная советскими математиками: Г.М. Андельсон-Вельским и Е.М. Ландисом в 1962 году.

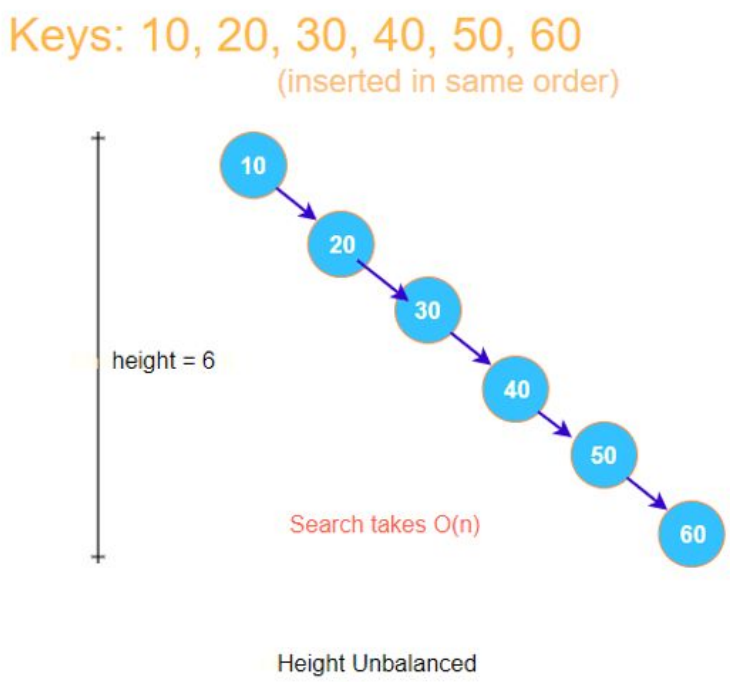
АВЛ – дерево является частным случаем двоичного дерева поиска, для которого определено следующее условие: каждый узел дерева имеет коэффициент сбалансированности, который должен удовлетворять условию $|h(Rtree) - h(Ltree)| \leq 1$, где h(Ltree) и h(Rtree) – высоты левого и правого поддеревьев соответственно. То есть, коэффициент сбалансированности для каждого узла дерева должен принимать одно из трех значений {-1,0,1}. Если, хотя бы для одного узла это условие не выполняется, то дерево не является сбалансированным по АВЛ и необходима его балансировка.

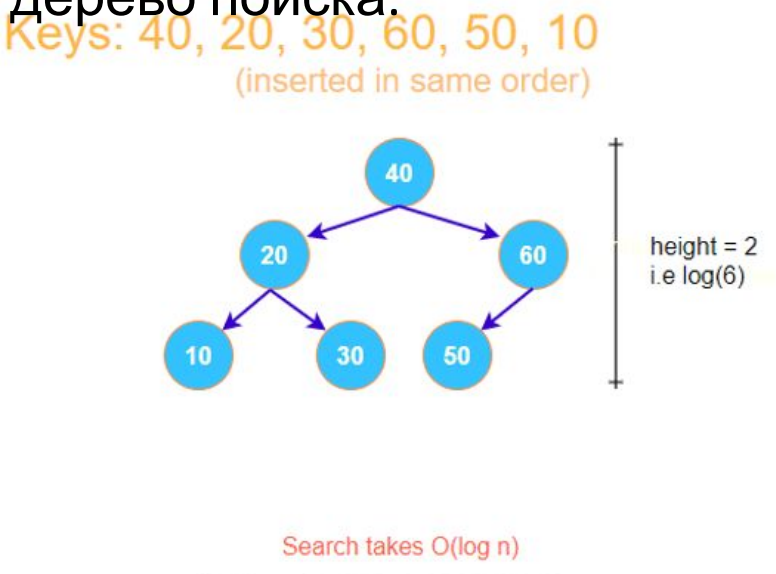
Сложность операций поиска, добавления, удаления узла в АВЛ-дереве составляет порядка O(log N), где N – количество элементов

Сбалансированные деревья

Рассмотрим следующее ключи, вставленные в заданном порядке в бинарное дерево поиска. Высота дерева растет линейно, когда мы вставляем ключи в порядке возрастания их значений. Таким образом, поиск, в худшем случае, занимает O(n).

Для поиска элемента требуется линейное время; следовательно, нет смысла использовать Двоичное дерево поиска.

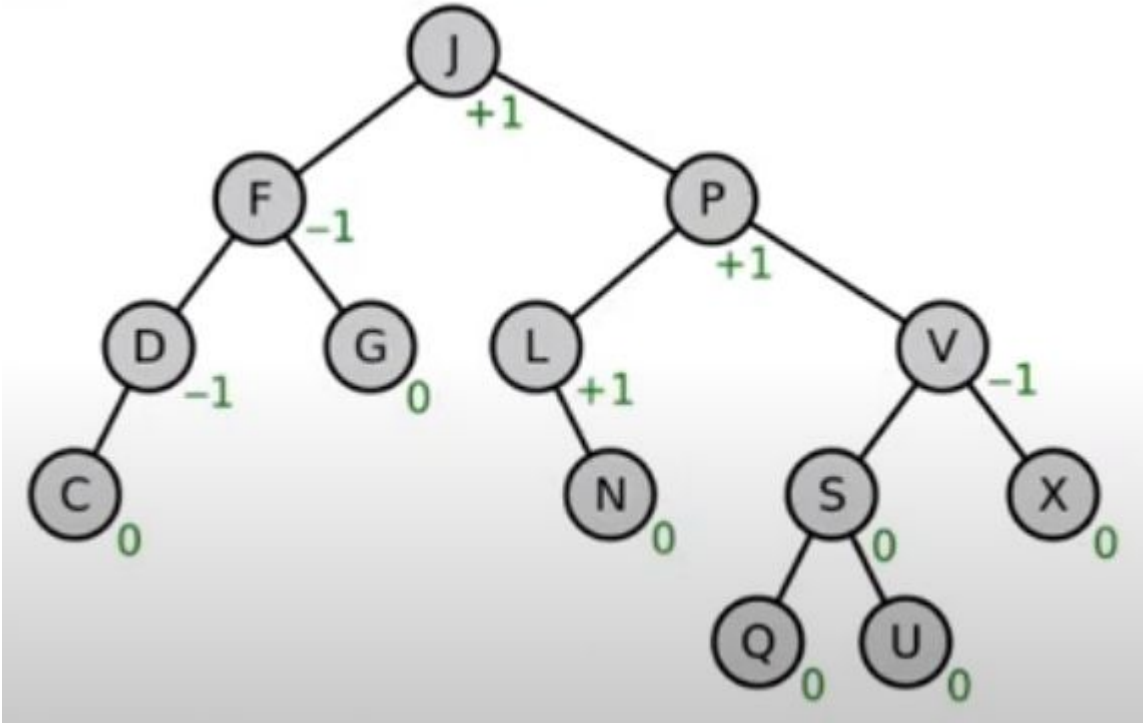




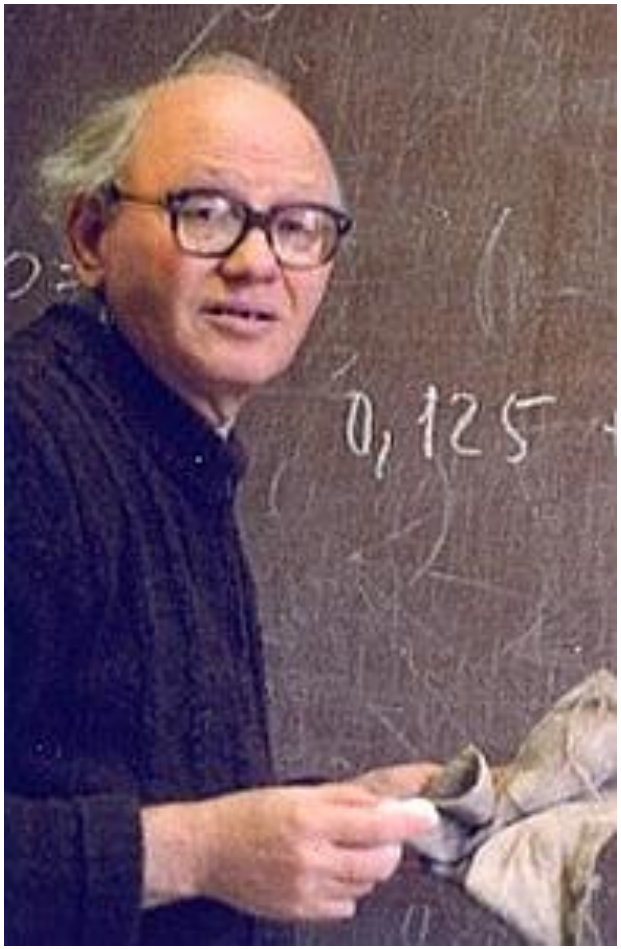
С другой стороны, если высота дерева сбалансирована, мы получим лучшее время для поиска элемента.

Сбалансированные деревья

АВЛ – дерево является частным случаем двоичного дерева поиска, для которого определено следующее условие: каждый узел дерева имеет коэффициент сбалансированности, который должен удовлетворять условию | h(Rtree) - h(Ltree) | ≤ 1, где h(Ltree) и h(Rtree) – высоты левого и правого поддеревьев соответственно. То есть, коэффициент сбалансированности для каждого узла дерева должен принимать одно из трех значений {-1,0,1}. Если, хотя бы для одного узла это условие не выполняется, то дерево не является сбалансированным по АВЛ и необходима его балансировка.



Авторы структуры АВЛ- деревьев



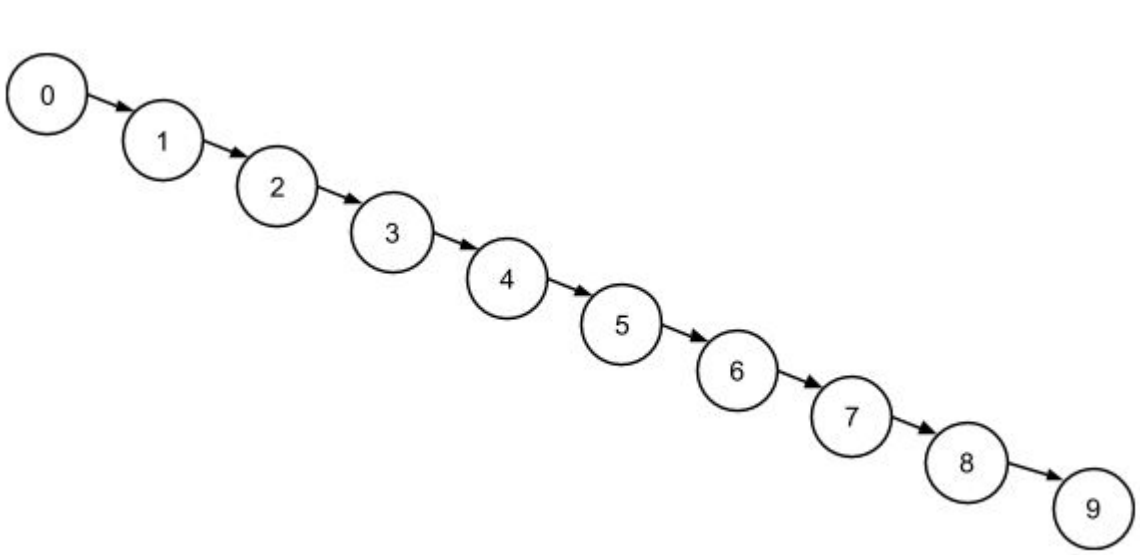
Г. М.Адельсон-Вельский

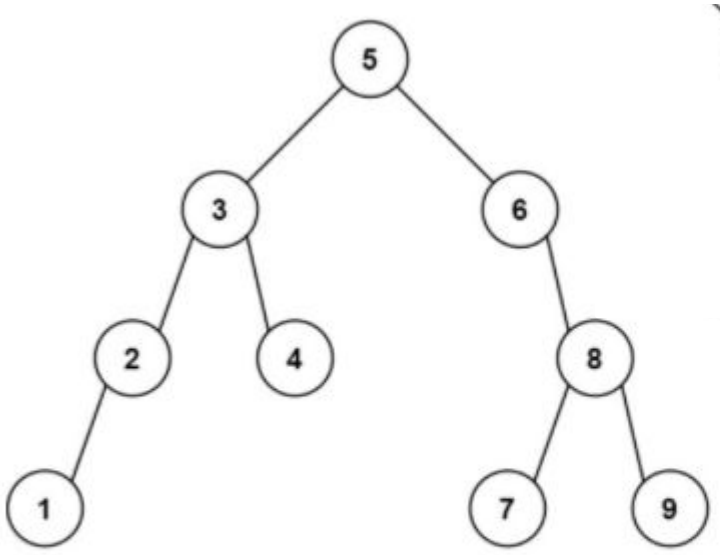
Е. М. Ландис

Сбалансированные деревья

АВЛ–дерево - это модификация классического бинарного дерева поиска, благодаря которой структура лучше сбалансирована и практически не может выродиться. Вырождением называют ситуацию, когда у каждого узла оказывается только по одному потомку и структура фактически становится линейной — это неоптимально.

Благодаря сбалансированности и борьбе с вырождением дерева информация в нем хранится более эффективно. Поэтому доступ к данным оказывается быстрее и найти их становится легче.



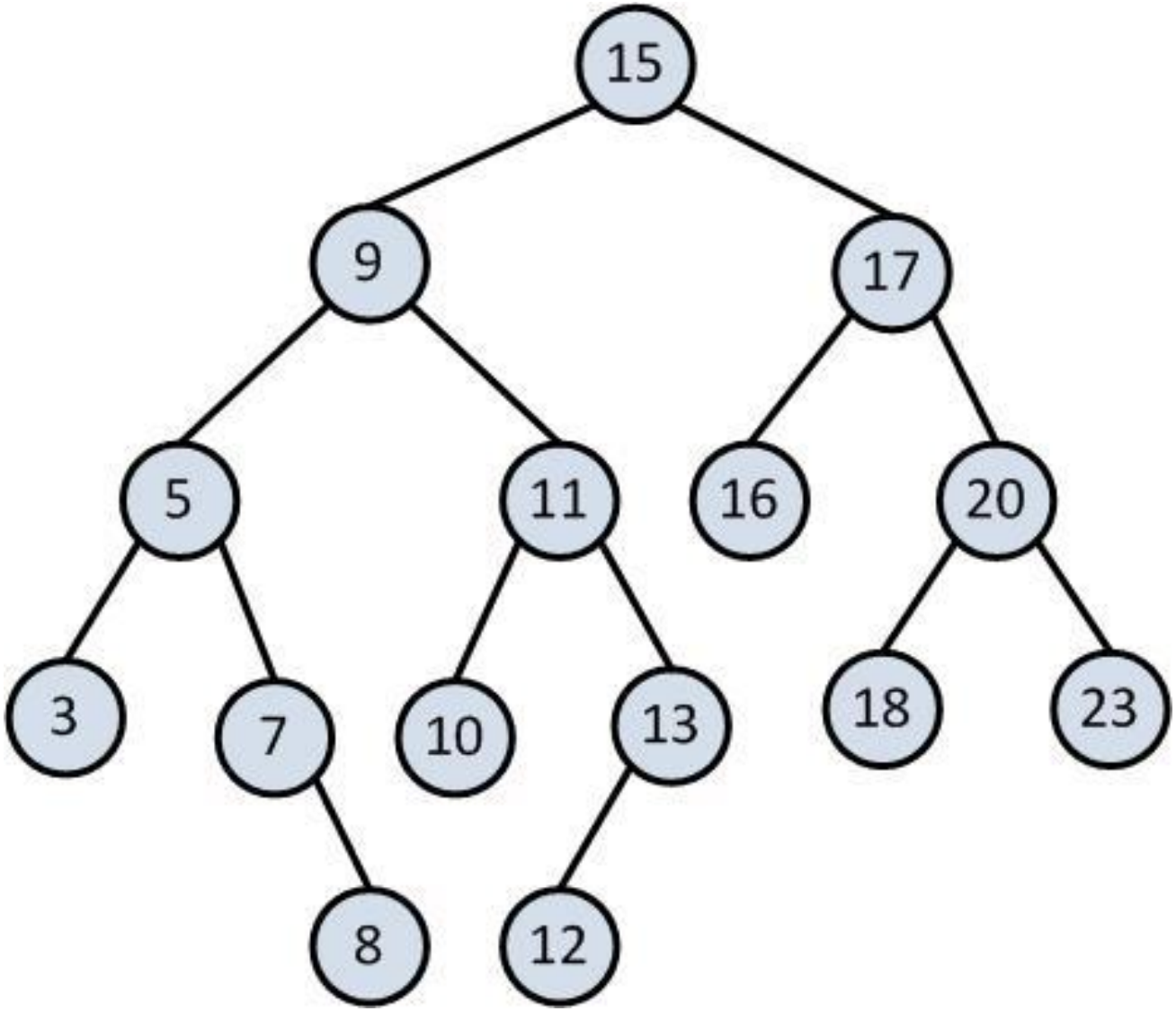


Сбалансированные деревья

Кто пользуется АВЛ-деревьями

- Разработчики, которые работают с алгоритмами сортировки, хранения и поиска информации, реализуют те или иные сложные структуры. Сфера применения деревьев довольно широка, и про умение ими пользоваться могут спрашивать на собеседованиях в соответствующих отраслях.
- Аналитики данных, для которых работа с информацией — профессиональная сфера деятельности. АВЛ-деревья же — это один из методов эффективно хранить информацию и быстро получать из структуры конкретные данные. Также они могут пригодиться при реализации алгоритмов работы с данными.
- Математики, которые решают фундаментальные и практические задачи. АВЛ-деревья — подвид бинарных деревьев поиска, которые, в свою очередь, являются своеобразными графами. Все это относится к области дискретной математики: она частично пересекается с Computer Science и схожими направлениями.

Пример АВЛ-дерева



Для чего нужны АВЛ-деревья

- Для хранения данных. Эта структура позволяет хранить информацию в «узлах» дерева и перемещаться по ней с помощью путей, которые соединяют между собой узлы. Благодаря особому алгоритму данные хранятся относительно эффективно и с ними довольно удобно работать. Мы подробнее поговорим об этом ниже.
- Для поисковых алгоритмов. АВЛ-деревья и бинарные деревья поиска в принципе — важная составная часть разнообразных алгоритмов поиска информации. Их применяют при построении поисковых систем и интеллектуальных сервисов.
- Для сортировки. Хранение информации в АВЛ-дереве позволяет быстрее отсортировать данные, а задача сортировки часто встречается в IT. С помощью деревьев можно хранить и сортировать информацию в базах данных, в особых участках памяти, в хэшах и других структурах.

Для чего нужны АВЛ-деревья (продолжение)

- Для программных проверок. АВЛ-дерево может использоваться для решения некоторых стандартных задач, например для быстрой проверки существования элемента в структуре.

- Для построения сложных структур. Дерево может быть составной частью более сложной структуры данных или какого-либо алгоритма, например используемого для поиска, хранения или принятия решений.

- Для других задач. Деревья могут понадобиться везде, где нужны связные структуры данных, оптимизированные под определенные операции. Например, они могут быть более функциональной альтернативой связанному списку — линейной структуре данных, где в каждом элементе есть ссылка на предыдущий и/или следующий.

Структура АВЛ-дерева

В рассматриваемых примерах программная структура АВЛ-дерева будет иметь следующий вид:

```
struct Tree
{
        int Value;
        Tree *Right, *Left;
};
```

Возможен и такой вариант представления:

```
struct node // структура для представления узлов дерева
{
        int key;
        unsigned char height;
        node* left;
        node* right;
        node(int k) { key = k; left = right = 0; height = 1; }
};
```

Поле key хранит ключ узла, поле height — высоту поддерева с корнем в данном узле, поля left и right — указатели на левое и правое поддеревья. Простой конструктор создает новый узел (высоты 1) с заданным ключом k.

АВЛ-деревья (AVL-деревья)

Приведение уже существующего дерева к идеально сбалансированному - процесс сложный. Проще **балансировать дерево в процессе его роста (после включения каждого узла).**

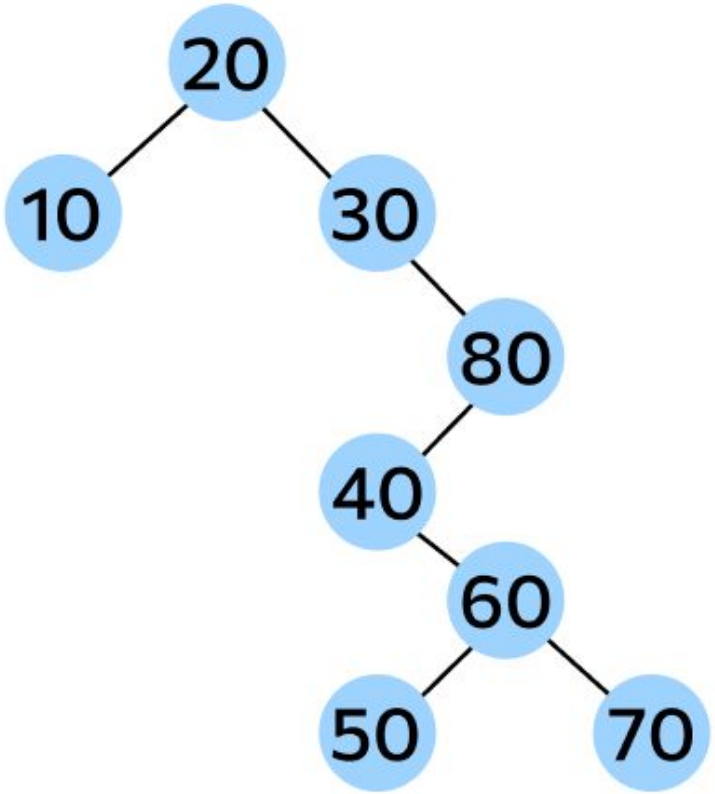
Однако требование **идеальной сбалансированности** делает и этот процесс достаточно сложным, способным затрагивать все узлы дерева.

В **1962 году** советские математики **Адельсон-Вельский Г. М.** и **Ландис Е.А.** предложили **метод балансировки,** требующий после включения или исключения узла лишь **локальные изменения вдоль пути от корня к данному узлу, то есть времени не более O(log_2 n).**

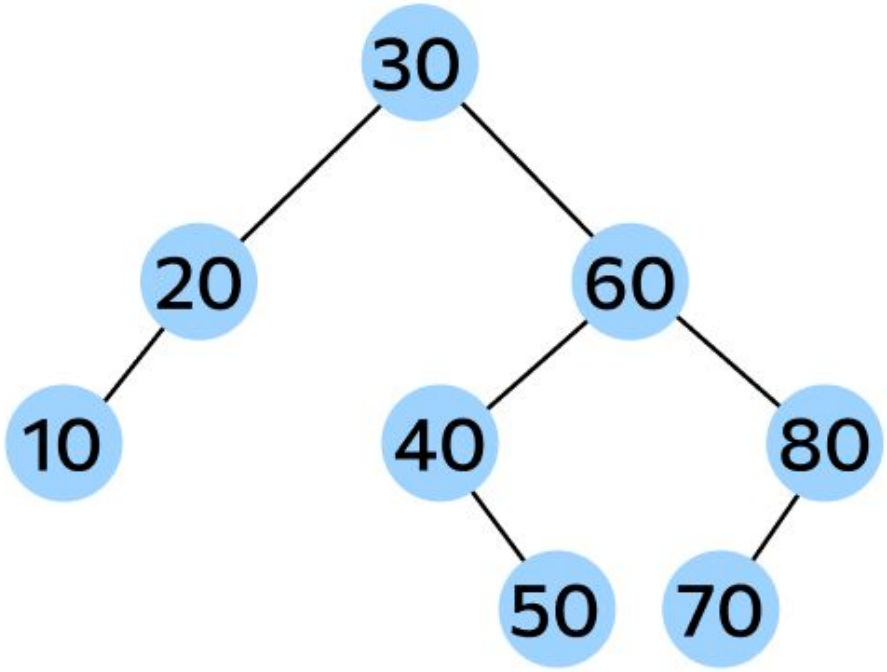
При этом деревьям разрешается отклоняться от идеальной сбалансированности, но в небольшой степени, чтобы среднее время доступа к узлам было лишь немногим больше, чем в идеально сбалансированном дереве. Такие деревья получили

АВЛ-деревья (AVL-деревья)

Бинарное дерево поиска



AVL-дерево



Особенности АВЛ-деревьев

АВЛ-дерево отличается от обычного бинарного дерева поиска несколькими особенностями:

- оно сбалансировано по высоте. Поддеревья, которые образованы левым и правым потомками каждого из узлов, должны различаться длиной не более чем на один уровень;

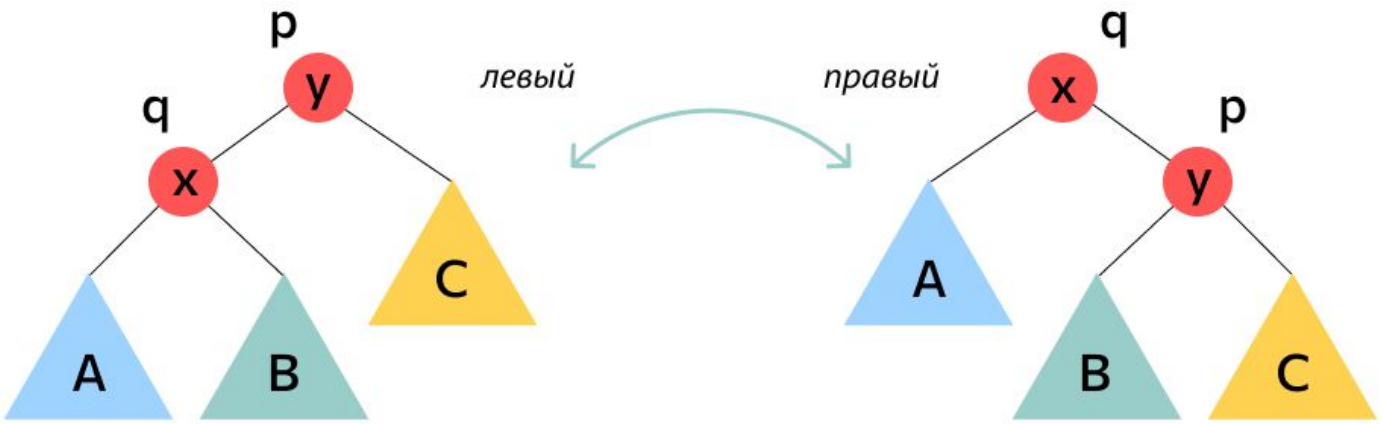
- из первой особенности вытекает еще одна — общая длина дерева и, соответственно, скорость операций с ним зависят от числа узлов логарифмически и гарантированно;

- вероятность получить сильно несбалансированное АВЛ-дерево крайне мала, а риск, что оно выродится, практически отсутствует.

Сбалансированность такого дерева гарантирована, в отличие от так называемых рандомизированных деревьев, которые сбалансированы вероятностно — то есть с небольшой вероятностью структура все еще может выродиться

Что такое балансировка

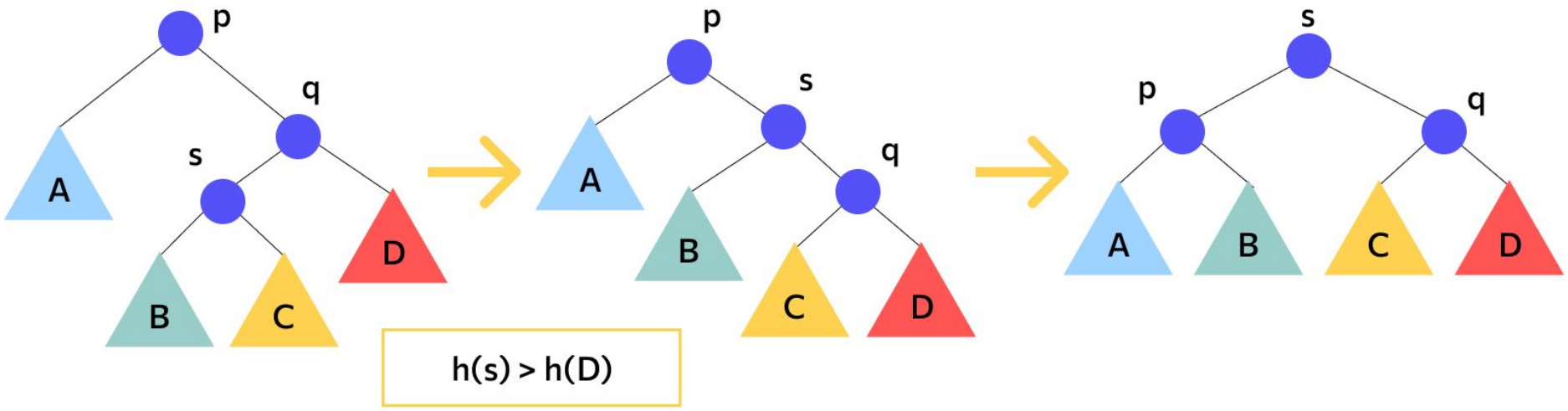
Балансировкой называют операцию, которая делает дерево более сбалансированным. В случае с АВЛ-деревьями ее применяют, если нарушается главное правило структуры: поддеревья-потомки одного узла начинают различаться больше чем на один уровень. Если разница в количестве уровней становится равна 2 или –2, запускается балансировка: связи между предками и потомками изменяются и перестраиваются так, чтобы сохранить правильную структуру.



$A < x < B < y < C$

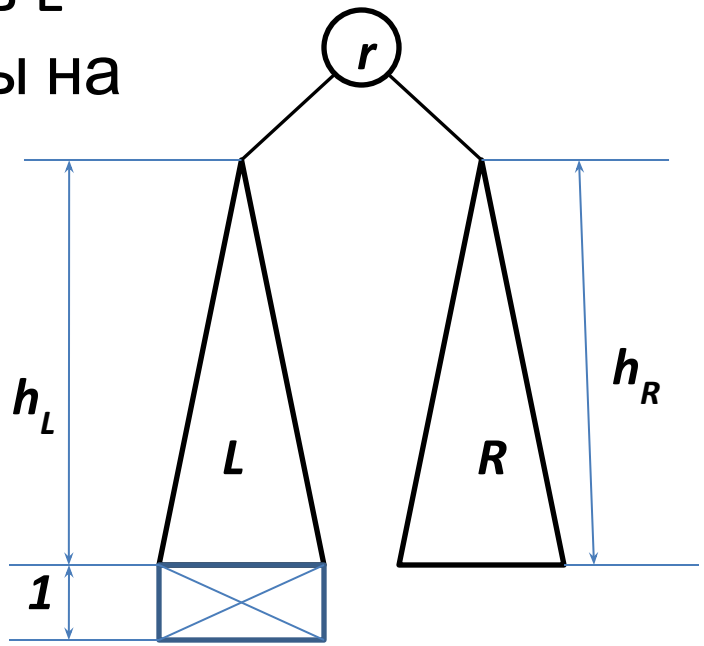
Что такое балансировка

Обычно для этого какой-либо из узлов «поворачивается» влево или вправо, то есть меняет свое расположение. Поворот может быть простым, когда расположение изменяет только один узел, или большим: при нем два узла разворачиваются в разные стороны. Большой поворот эффективен там, где не сработает обычный.



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*

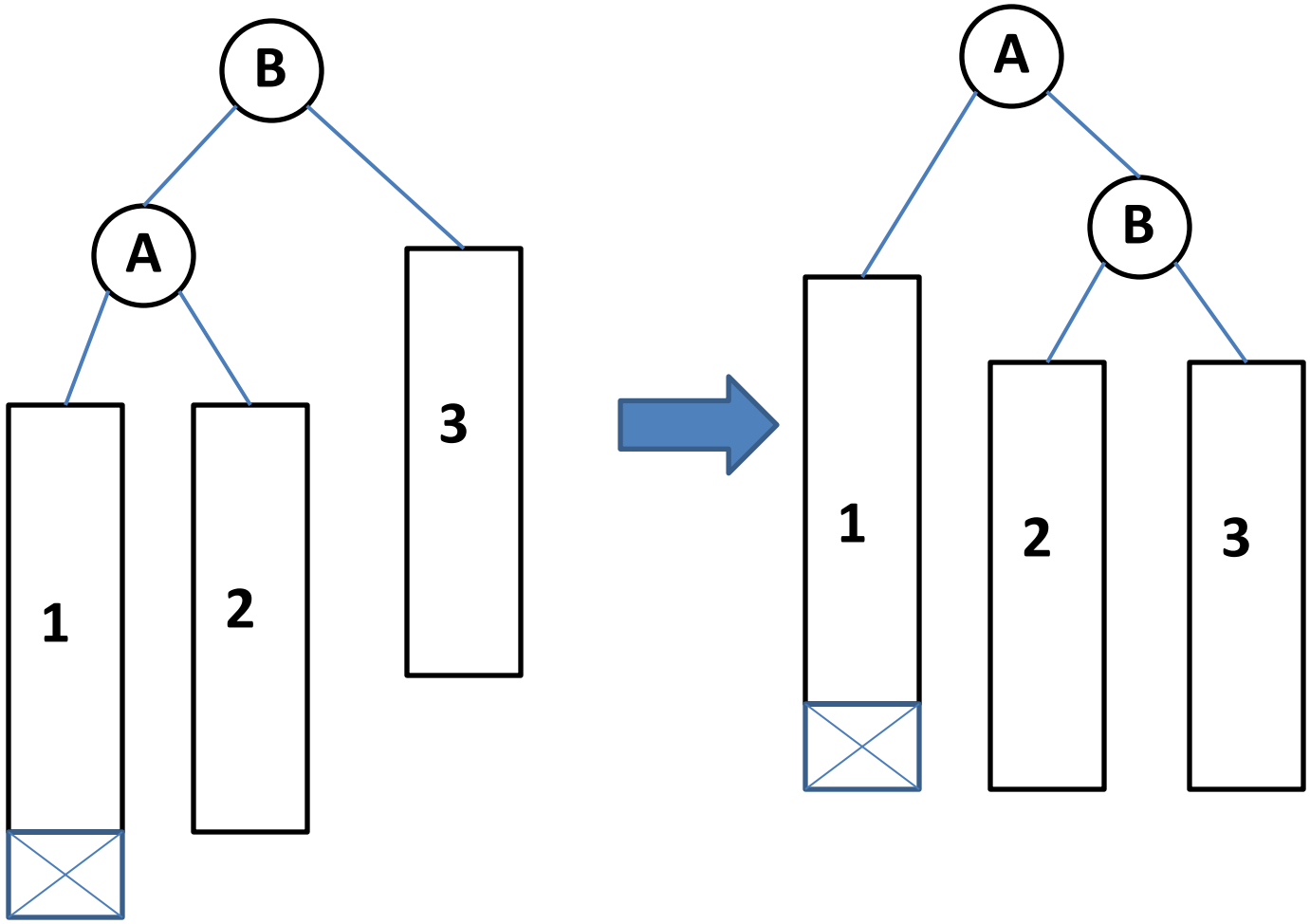
Добавление узла к AVL-дереву

Процедура, которая **вставляет в дерево новый узел**, **рекурсивно спускается вниз по дереву** (начиная путь с корня), **чтобы найти местоположение узла** (нисходящая рекурсия).

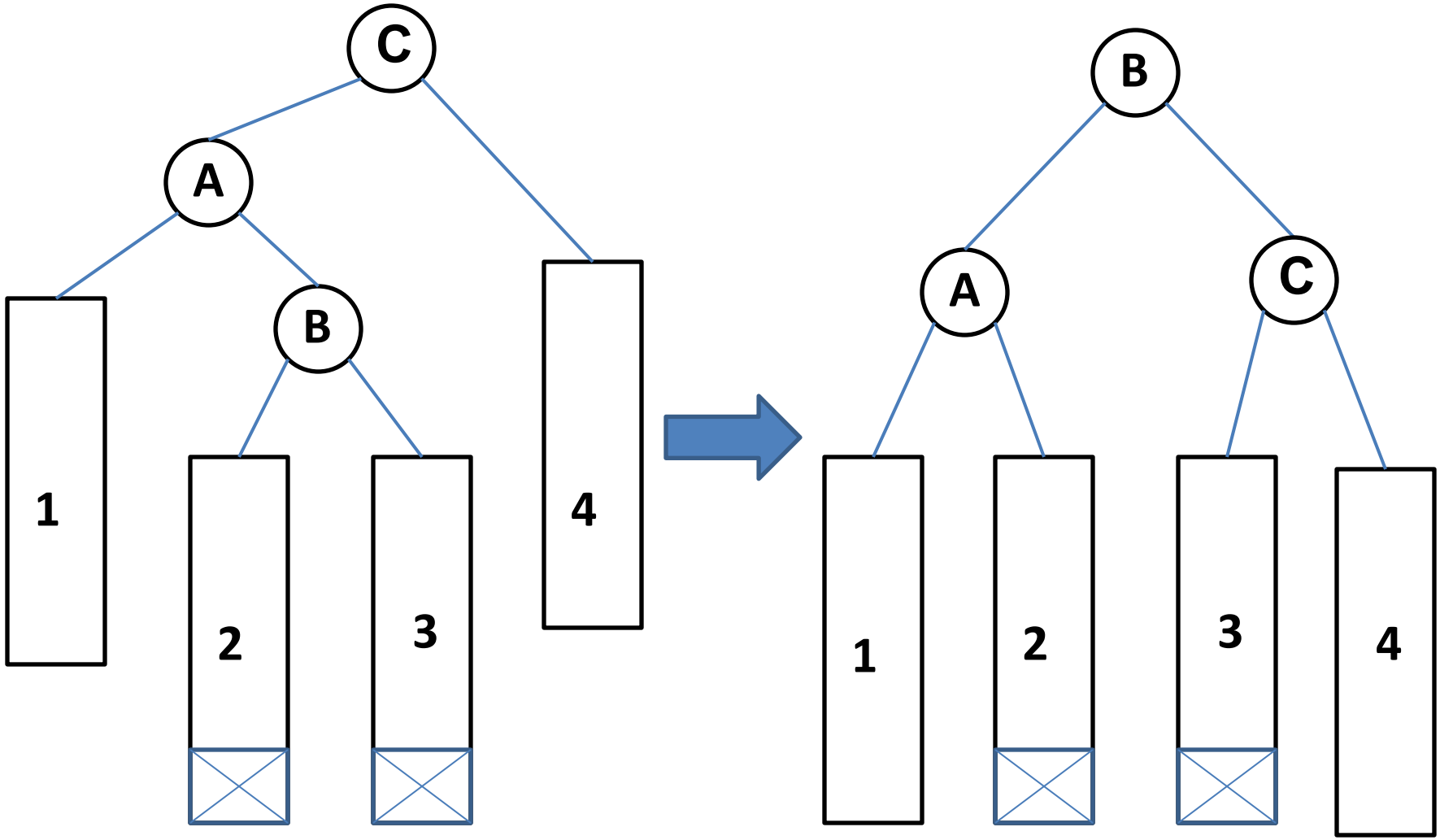
После вставки вершины, **она исследует узлы в обратном порядке – к корню** (восходящая рекурсия)**, проверяя, чтобы высота поддеревьев отличалась не более чем на единицу.**

Если найдена вершина, где это **условие не выполняется**, то **элементы сдвигаются по кругу** (как - будет показано далее),чтобы сохранить выполняемость условия AVL-дерева.

Вставка в левое поддерево



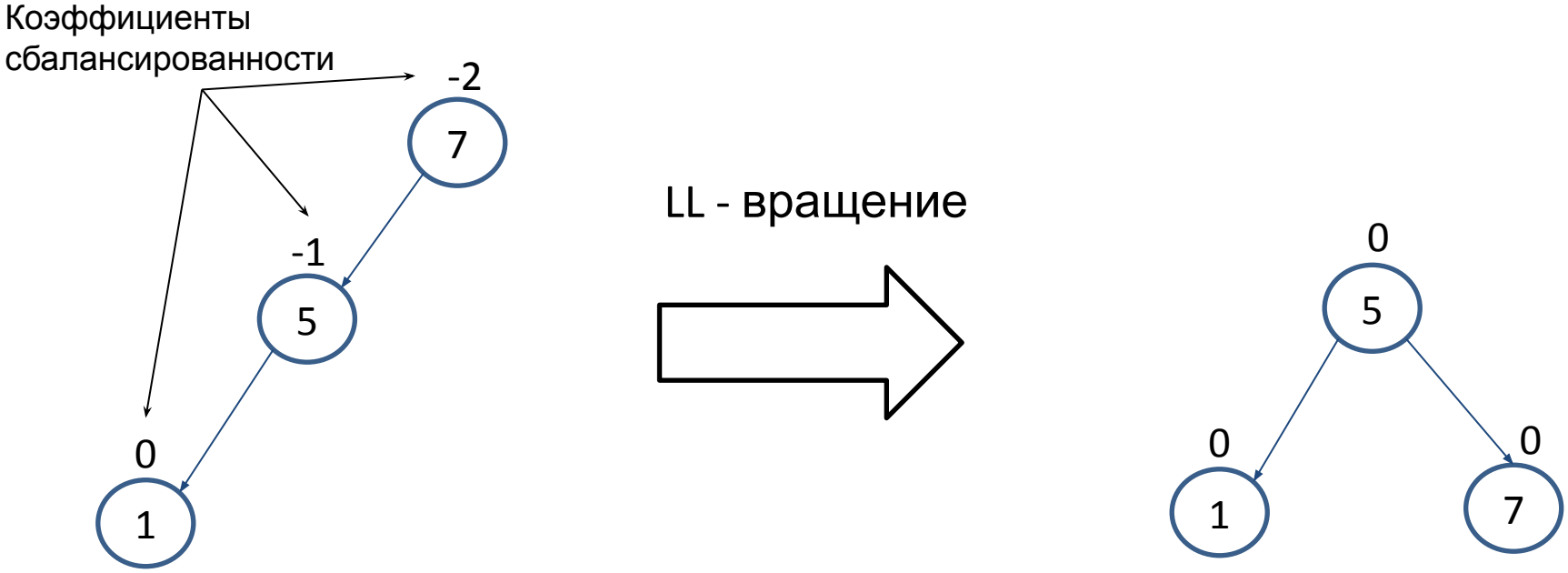
Вставка в правое поддерево



Балансировка АВЛ-дерева
LL - вращение

Существует 4 вида ситуаций в которых необходима балансировка АВЛ-дерева, как правило, такие ситуации возникают по причине добавления или удаления узла из АВЛ-дерева. Рассмотрим первую из них.

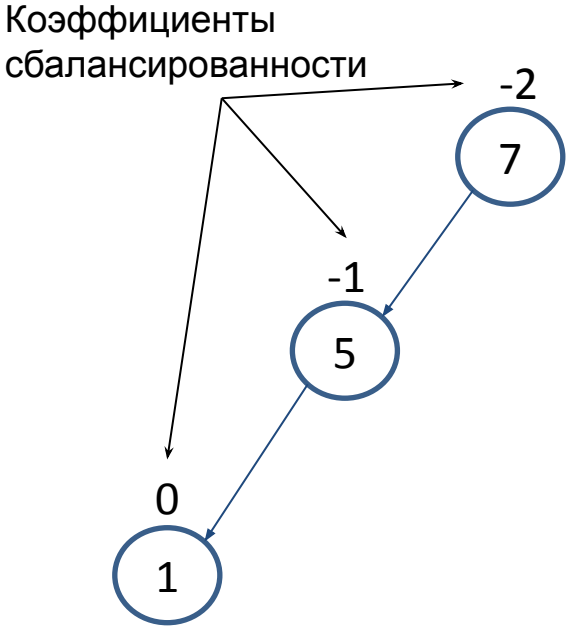
Данная ситуация возникает когда дерево имеет «перевес» в левом поддереве левого потомка узла дерева, в этом случае необходимо провести балансировку малым левым вращением или LL-вращением (Left-Left Rotation).



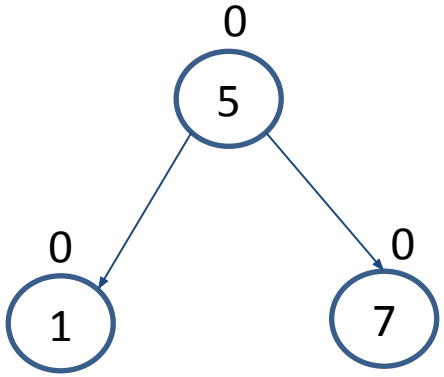
Балансировка АВЛ-дерева
LL - вращение

Существует 4 вида ситуаций в которых необходима балансировка АВЛ-дерева, как правило, такие ситуации возникают по причине добавления или удаления узла из АВЛ-дерева. Рассмотрим первую из них.

Данная ситуация возникает когда дерево имеет «перевес» в левом поддереве левого потомка узла дерева, в этом случае необходимо провести балансировку малым левым вращением или LL-вращением (Left-Left Rotation).




```
Tree* LL_Rotation(Tree* T2 ){
Tree* T1;
T1 = T2->Left;
T2->Left = T1->Right;
T1->Right = T2;
return T1; }
```

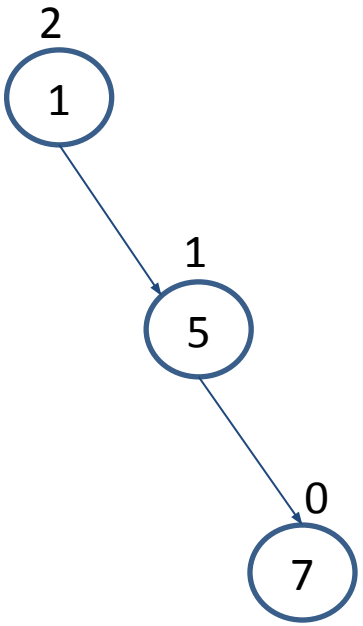


Балансировка АВЛ-дерева
RR - вращение

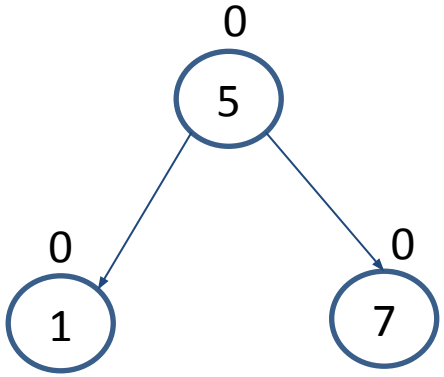
Данная ситуация возникает когда дерево имеет «перевес» в правом поддереве правого потомка узла дерева, в этом случае необходимо провести балансировку малым правым вращением или RR-вращением (Right-Right Rotation).

Балансировка АВЛ-дерева
RR - вращение

Данная ситуация возникает когда дерево имеет «перевес» в правом поддереве правого потомка узла дерева, в этом случае необходимо провести балансировку малым правым вращением или RR-вращением (Right-Right Rotation).

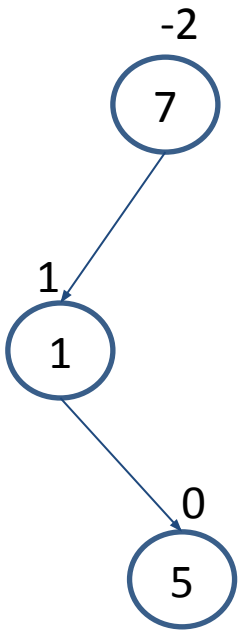



```
Tree* RR_Rotation (Tree* T1 ){
Tree* T2;
T2 = T1->Right;
T1->Right = T2->Left;
T2->Left = T1;
return T2; }
```



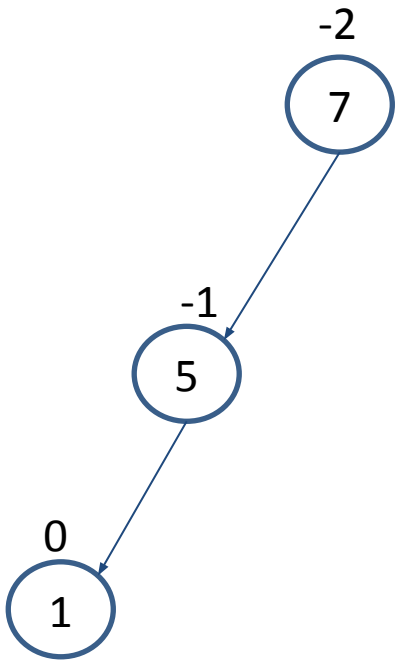
Балансировка АВЛ-дерева
LR - вращение

Данная ситуация возникает когда дерево имеет «перевес» в правом поддереве левого потомка узла дерева, в этом случае необходимо провести балансировку комбинацией малого правого (RR) и малого левого (LL) вращений, такой способ балансировки называется лево-правым вращением (Left-Right-Rotation)

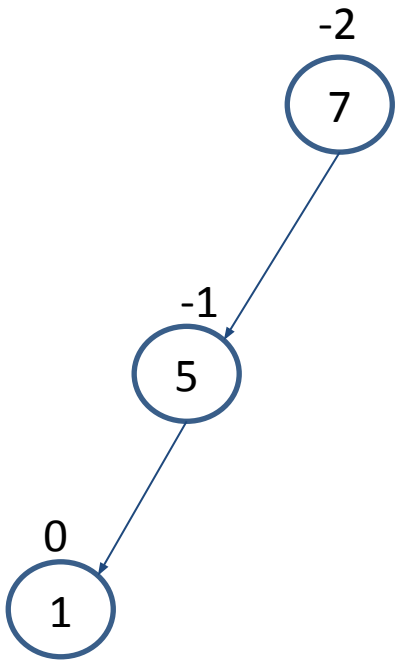


Шаг 1: RR - вращение

```
Tree* LR_Rotation(Tree* T3)
{
  T3->Left = RR_Rotation(T3->Left );
  return LL_Rotation(T3);
}
```

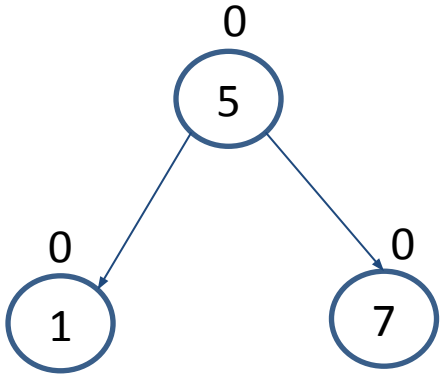


Балансировка АВЛ-дерева
LR – вращение (продолжение)



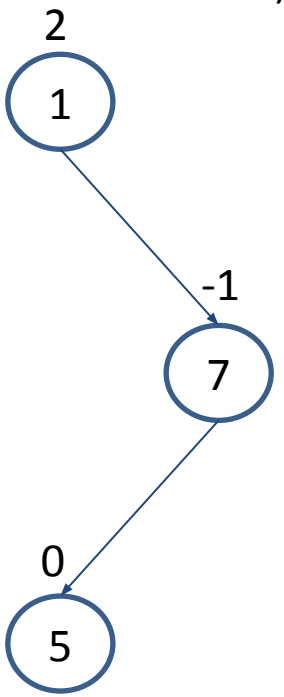
Шаг 2: LL - вращение

```
Tree* LR_Rotation(Tree* T3)
{
  T3->Left = RR_Rotation(T3->Left );
  return LL_Rotation(T3);
}
```



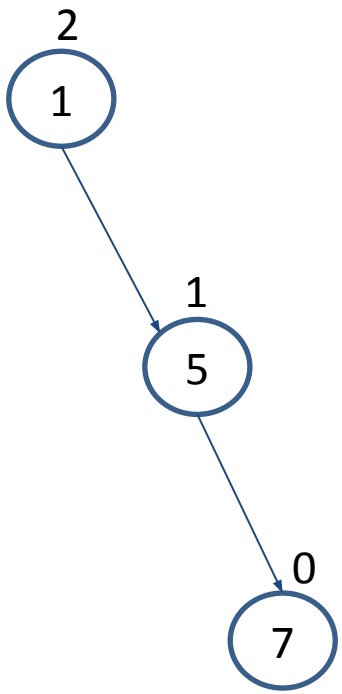
Балансировка АВЛ-дерева
RL - вращение

Данная ситуация возникает когда дерево имеет «перевес» в левом поддереве правого потомка узла дерева, в этом случае необходимо провести балансировку комбинацией малого левого (LL) и малого правого (RR) вращений, такой способ балансировки называется право-левым вращением (Right-Left-Rotation)

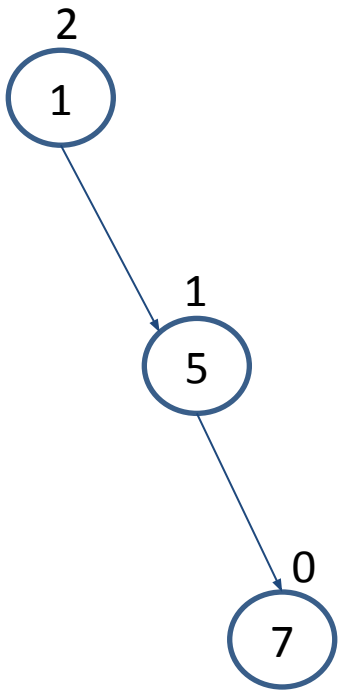


Шаг 1: LL - вращение

```
Tree* RL_Rotation(Tree* T1) {
  T1->Right = LL_Rotation(T1->Right );
  return RR_Rotation(T1);
}
```

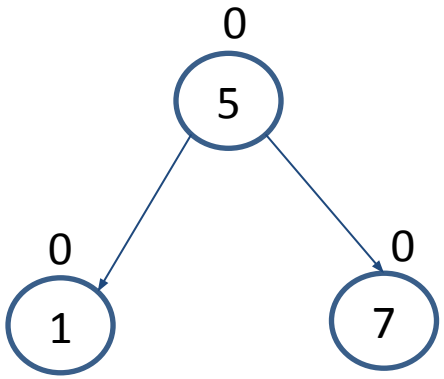


Балансировка АВЛ-дерева
RL – вращение (продолжение)



Шаг 2: RR - вращение

```
Tree* RL_Rotation(Tree* T1) {
  T1->Right = LL_Rotation(T1->Right );
  return RR_Rotation(T1);
}
```



Структура узлов

Будем представлять узлы АВЛ-дерева следующей структурой:

```
struct node // структура для представления узлов дерева
{
	int key;
	unsigned char height;
	node* left;
	node* right;
	node(int k) { key = k; left = right = 0; height = 1; }
};
```

Напомню, поле key хранит ключ узла, поле height — высоту поддерева с корнем в данном узле, поля left и right — указатели на левое и правое поддеревья. Простой конструктор создает новый узел (высоты 1) с заданным ключом k.

Традиционно, **узлы АВЛ-дерева хранят не высоту, а разницу высот правого и левого поддеревьев** (так называемый **balance factor**), которая может принимать только три значения -1, 0 и 1. Однако, заметим, что эта разница все равно хранится в переменной, размер которой равен минимум одному байту (если не придумывать каких-то хитрых схем «эффективной» упаковки таких величин). Вспомним, что высота $h < 1.44 \log_2(n + 2)$, это значит, например, что при $n=10^9$ (один миллиард ключей, больше 10 гигабайт памяти под хранение узлов) высота дерева не превысит величины h=44, которая с успехом помещается в тот же один байт памяти, что и balance factor. Таким образом, хранение высот с одной стороны не увеличивает объем памяти, отводимой под узлы дерева, а с другой стороны существенно упрощает реализацию некоторых операций.

Структура узлов

Определим три вспомогательные функции, связанные с высотой. Первая является оберткой для поля height, она может работать и с нулевыми указателями (с пустыми деревьями):

```
unsigned char height(node* p)
{
    return p?p->height:0;
}
```

Вторая вычисляет balance factor заданного узла (и работает только с ненулевыми указателями):

```
int bfactor(node* p)
{
    return height(p->right)-height(p->left);
}
```

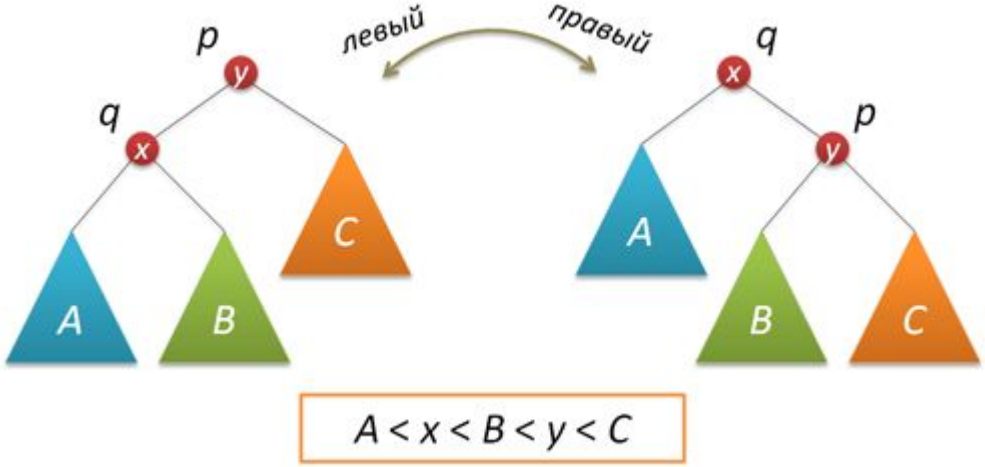
Третья функция восстанавливает корректное значение поля height заданного узла (при условии, что значения этого поля в правом и левом дочерних узлах являются корректными):

```
void fixheight(node* p)
{
    unsigned char hl = height(p->left);
    unsigned char hr = height(p->right);
    p->height = (hl>hr?hl:hr)+1;
}
```

Заметим, что все три функции являются нерекурсивными, т.е. время их работы есть величина O(1).

Балансировка узлов

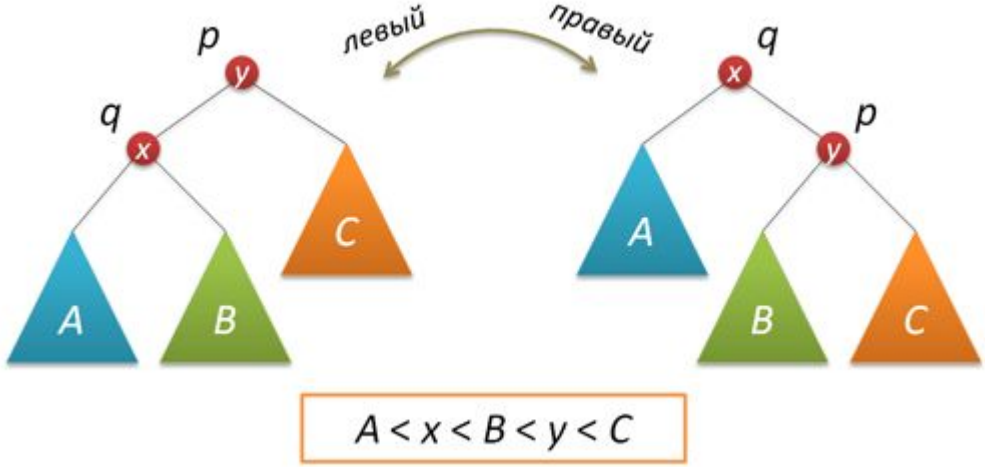
В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает расбалансировка поддерева. Для выправления ситуации применяются хорошо нам известные повороты вокруг тех или иных узлов дерева. Напомню, что простой поворот вправо (влево) производит следующую трансформацию дерева:



Код, реализующий правый поворот, выглядит следующим образом (как обычно, каждая функция, изменяющая дерево, возвращает новый корень полученного дерева):

```
node* rotateright(node* p) // правый поворот вокруг p
{
    node* q = p->left;
    p->left = q->right;
    q->right = p;
    fixheight(p);
    fixheight(q);
    return q;
```

Балансировка узлов

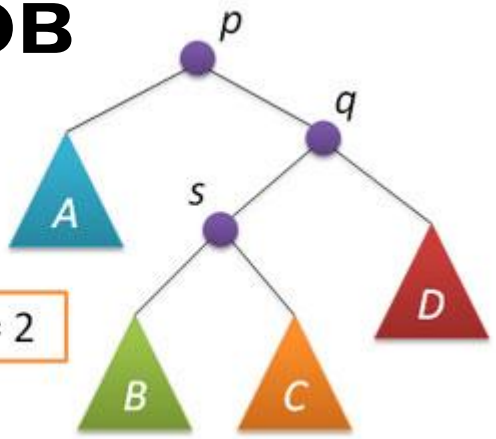


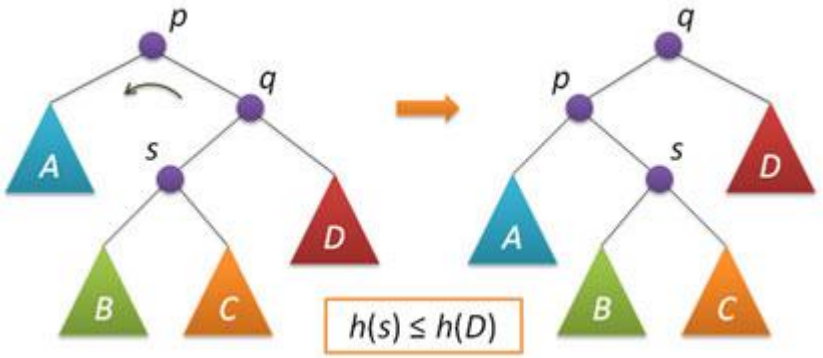
Левый поворот является симметричной копией правого:

```
node* rotateleft(node* q) // левый поворот вокруг q
{
  node* p = q->right;
  q->right = p->left;
  p->left = q;
  fixheight(q);
  fixheight(p);
  return p;
}
```

Балансировка узлов

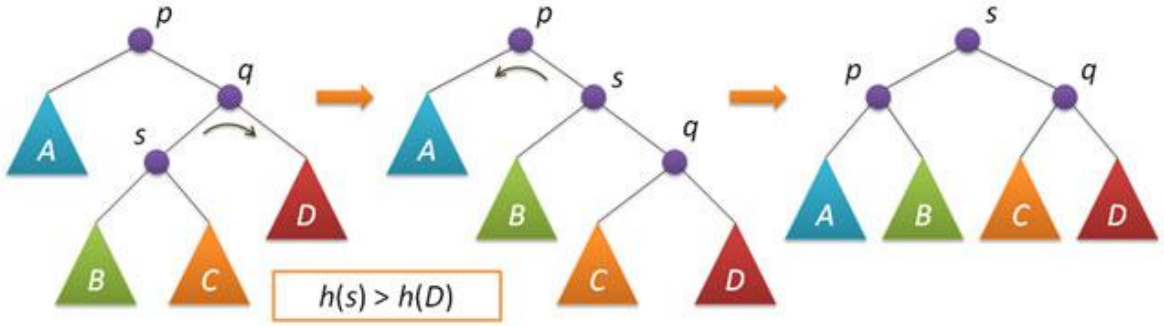
Рассмотрим теперь ситуацию дисбаланса, когда высота правого поддерева узла p на 2 больше высоты левого поддерева (обратный случай является симметричным и реализуется аналогично). Пусть q — правый дочерний узел узла p, а s — левый дочерний узел узла q.





Анализ возможных случаев в рамках данной ситуации показывает, что для исправления расбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо так называемый большой поворот влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q больше высоты его правого поддерева: h(s)≤h(D)

Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p.



Балансировка узлов

Код, выполняющий балансировку, сводится к проверке условий и выполнению поворотов:

```
node* balance(node* p) // балансировка узла p
{
    fixheight(p);
    if( bfactor(p)==2 )     {
        if( bfactor(p->right) < 0 )
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if( bfactor(p)==-2 )    {
        if( bfactor(p->left) > 0  )
            p->left = rotateleft(p->left);
        return rotateright(p);    }
    return p; // балансировка не нужна
}
```

Описанные функции поворотов и балансировки также не содержат ни циклов, ни рекурсии, а значит выполняются за постоянное время, не зависящее от размера АВЛ-дерева.

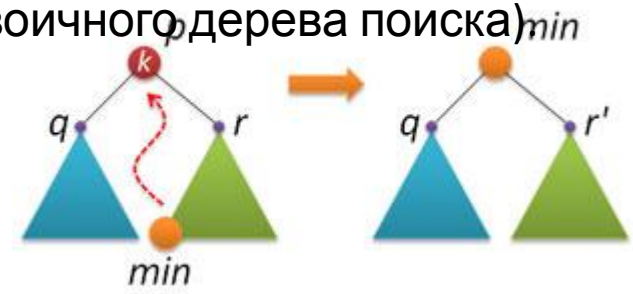
Вставка ключей

Вставка нового ключа в АВЛ-дерево выполняется, по большому счету, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным.

```
node* insert(node* p, int k) // вставка ключа k в дерево с
           //корнем p
{
 if( !p ) return new node(k);
 if( k<p->key )
    p->left = insert(p->left,k);
 else
    p->right = insert(p->right,k);
 return balance(p);
}
```

Удаление узлов

С удалением узлов из АВЛ-дерева, к сожалению, все не так просто, как с рандомизированными деревьями поиска. Способа, основанного на слиянии (join) двух деревьев, пока, ни найти, ни придумать никому не удалось. Поэтому за основу был взят вариант, описываемый практически везде (и который обычно применяется и при удалении узлов из стандартного двоичного дерева поиска).



Идея следующая: находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min. При реализации возникает несколько нюансов. Прежде всего, если у найденный узел p не имеет правого поддерева, то по свойству АВЛ-дерева слева у этого узла может быть только один единственный дочерний узел (дерево высоты 1), либо узел p вообще лист. В обоих этих случаях надо просто удалить узел p и вернуть в качестве результата указатель на левый дочерний узел узла p.

Пусть теперь правое поддерево у p есть. Находим минимальный ключ в этом поддереве. По свойству двоичного дерева поиска этот ключ находится в конце левой ветки, начиная от корня дерева. Применяем рекурсивную функцию:

```
node* findmin(node* p) // поиск узла с минимальным ключом в дереве p
{
    return p->left?findmin(p->left):p;
}
```

Удаление узлов

Еще одна служебная функция у нас будет заниматься удалением минимального элемента из заданного дерева. Опять же, по свойству АВЛ-дерева у минимального элемента справа либо подвешен единственный узел, либо там пусто. В обоих случаях надо просто вернуть указатель на правый узел и по пути назад (при возвращении из рекурсии) выполнить балансировку. Сам минимальный узел не удаляется, т.к. он нам еще пригодится.

```
node* removemin(node* p) // удаление узла с минимальным ключом из дерева p
{
 if( p->left==0 )
    return p->right;
 p->left = removemin(p->left);
 return balance(p);
}
```

Теперь все готово для реализации удаления ключа из АВЛ-дерева. Сначала находим нужный узел, выполняя те же действия, что и при вставке ключа:

```
node* remove(node* p, int k) // удаление ключа k из дерева p
{
 if( !p ) return 0;
 if( k < p->key )
    p->left = remove(p->left,k);
 else if( k > p->key )
    p->right = remove(p->right,k);
```

Удаление узлов

Как только ключ k найден, переходим к плану Б: запоминаем корни q и r левого и правого поддеревьев узла p; удаляем узел p; если правое поддерево пустое, то возвращаем указатель на левое поддерево; если правое поддерево не пустое, то находим там минимальный элемент min, потом его извлекаем оттуда, слева к min подвешиваем q, справа — то, что получилось из r, возвращаем min после его балансировки.

```
else //  k == p->key
{
    node* q = p->left;
    node* r = p->right;
    delete p;
    if( !r ) return q;
    node* min = findmin(r);
    min->right = removemin(r);
    min->left = q;
    return balance(min);
}
```

При выходе из рекурсии не забываем выполнить балансировку:

```
 return balance(p);
}
```

Удаление узлов

Вот собственно и все! Поиск минимального узла и его извлечение, в принципе, можно реализовать в одной функции, при этом придется решать (не очень сложную) проблему с возвращением из функции пары указателей. Зато можно сэкономить на одном проходе по правому поддереву.

Очевидно, что операции вставки и удаления (а также более простая операция поиска) выполняются за время пропорциональное высоте дерева, т.к. в процессе выполнения этих операций производится спуск из корня к заданному узлу, и на каждом уровне выполняется некоторое фиксированное число действий. А в силу того, что АВЛ-дерево является сбалансированным, его высота зависит логарифмически от числа узлов. Таким образом, время выполнения всех трех базовых операций гарантированно логарифмически зависит от числа узлов дерева.

Программа для АВЛ-деревьев

```
struct node // структура для представления узлов дерева
{
 int key;
 unsigned char height;
 node* left;
 node* right;
 node(int k) { key = k; left = right = 0; height = 1; }
};

unsigned char height(node* p)
{
 return p?p->height:0;
}

int bfactor(node* p)
{
 return height(p->right)-height(p->left);
}

void fixheight(node* p)
{
 unsigned char hl = height(p->left);
 unsigned char hr = height(p->right);
 p->height = (hl>hr?hl:hr)+1;
}
```

Программа для АВЛ-деревьев

```
node* rotateright(node* p) // правый поворот вокруг p
{
 node* q = p->left;
 p->left = q->right;
 q->right = p;
 fixheight(p);
 fixheight(q);
 return q;
}

node* rotateleft(node* q) // левый поворот вокруг q
{
 node* p = q->right;
 q->right = p->left;
 p->left = q;
 fixheight(q);
 fixheight(p);
 return p;
}
```

Программа для АВЛ-деревьев

```
node* balance(node* p) // балансировка узла p
{
 fixheight(p);
 if( bfactor(p)==2 )
 {
     if( bfactor(p->right) < 0 )
        p->right = rotateright(p->right);
     return rotateleft(p);
 }
 if( bfactor(p)==-2 )
 {
     if( bfactor(p->left) > 0  )
        p->left = rotateleft(p->left);
     return rotateright(p);
 }
 return p; // балансировка не нужна
}

node* insert(node* p, int k) // вставка ключа k в дерево с
корнем p
{
 if( !p ) return new node(k);
 if( k<p->key )
     p->left = insert(p->left,k);
 else
     p->right = insert(p->right,k);
 return balance(p);
}
```

Программа для АВЛ-деревьев

```
node* findmin(node* p) // поиск узла с минимальным ключом в дереве p
{
 return p->left?findmin(p->left):p;
}

node* removemin(node* p) // удаление узла с минимальным ключом из дерева p
{
 if( p->left==0 )
     return p->right;
 p->left = removemin(p->left);
 return balance(p);
}
```

Программа для АВЛ-деревьев

```
node* remove(node* p, int k) // удаление ключа k из дерева p
{
 if( !p ) return 0;
 if( k < p->key )
    p->left = remove(p->left,k);
 else if( k > p->key )
    p->right = remove(p->right,k);
 else //  k == p->key
 {
    node* q = p->left;
    node* r = p->right;
    delete p;
    if( !r ) return q;
    node* min = findmin(r);
    min->right = removemin(r);
    min->left = q;
    return balance(min);
 }
 return balance(p);
}
```

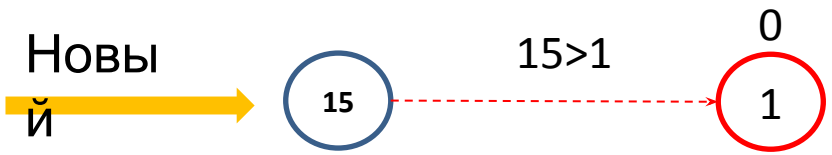
Построение АВЛ-дерева

Рассмотрим процесс построения АВЛ-дерева, состоящего из последовательности чисел:

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13

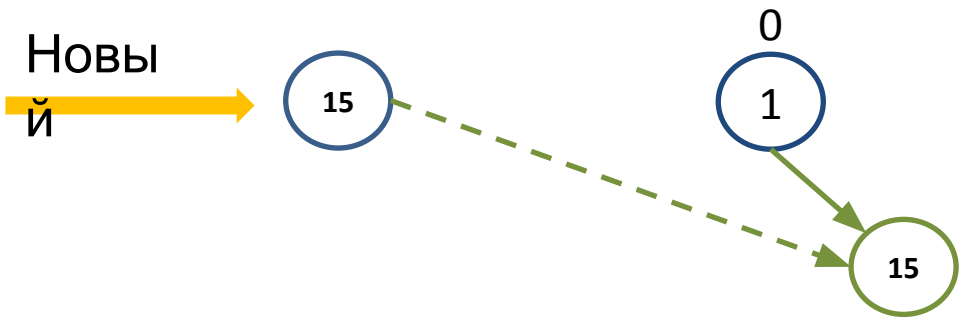
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



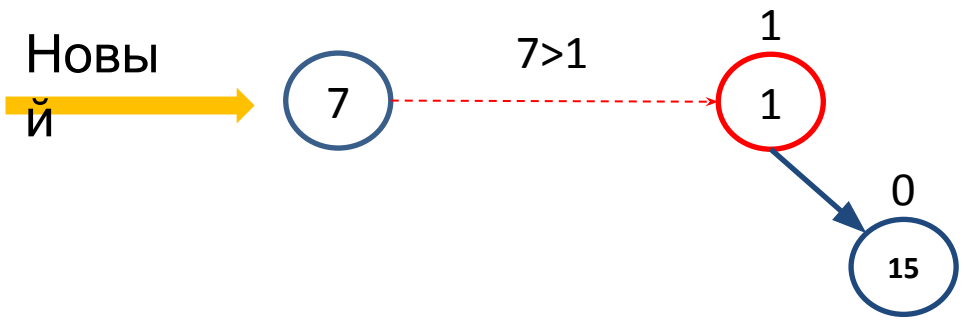
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



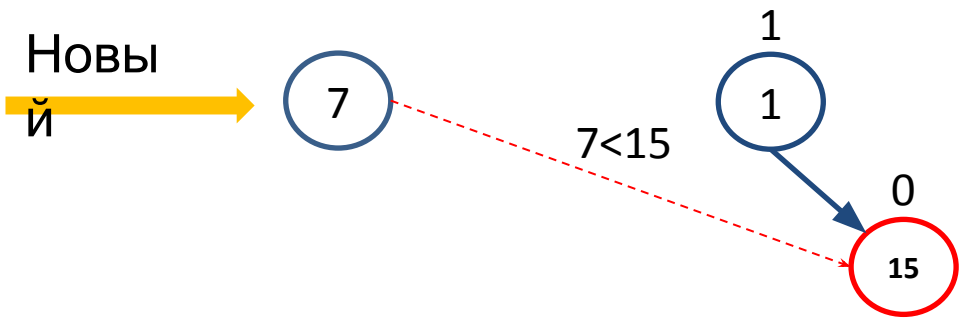
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



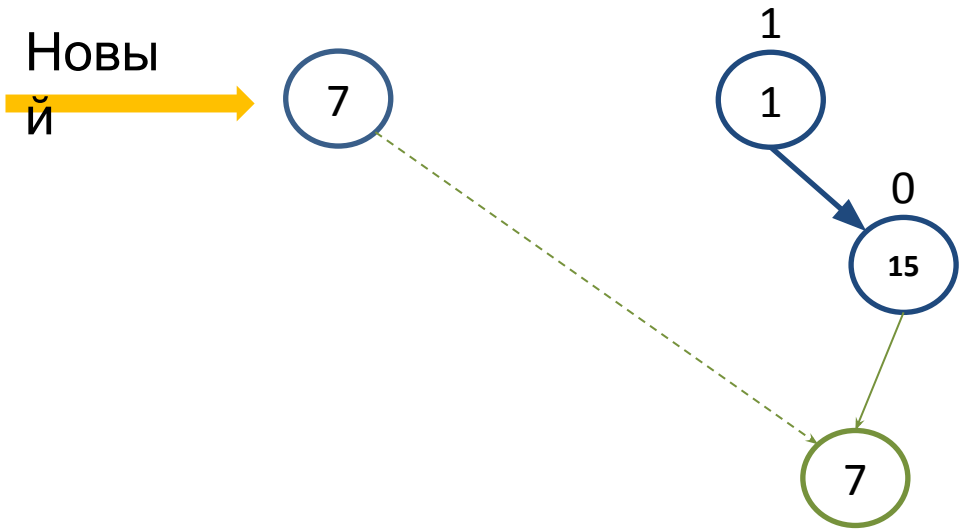
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

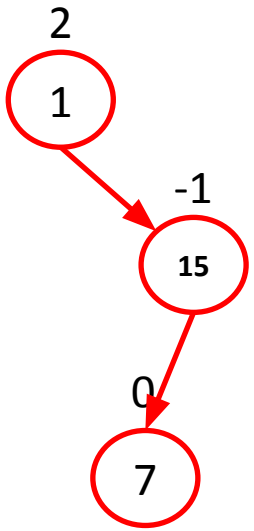
1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

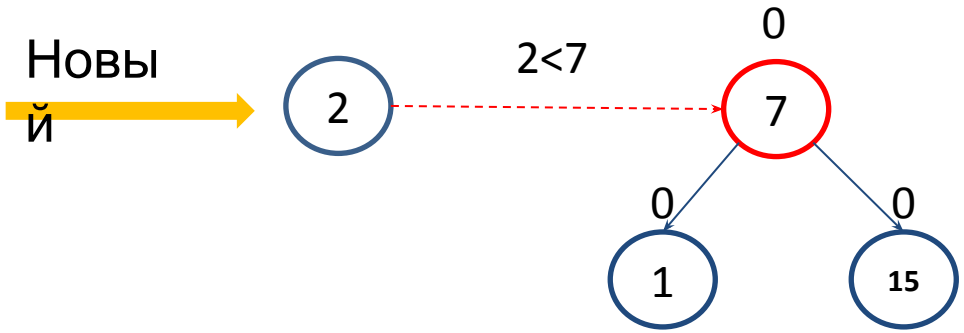
Разбалансировка в узле 1

Выполняем RL-поворот.



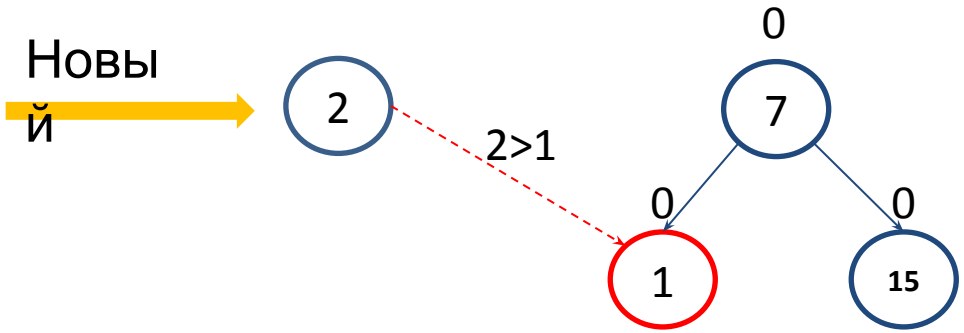
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



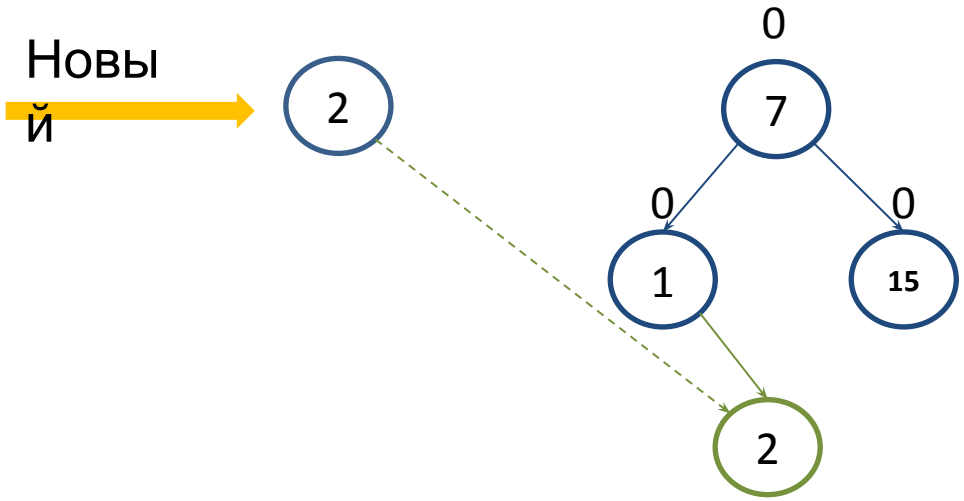
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



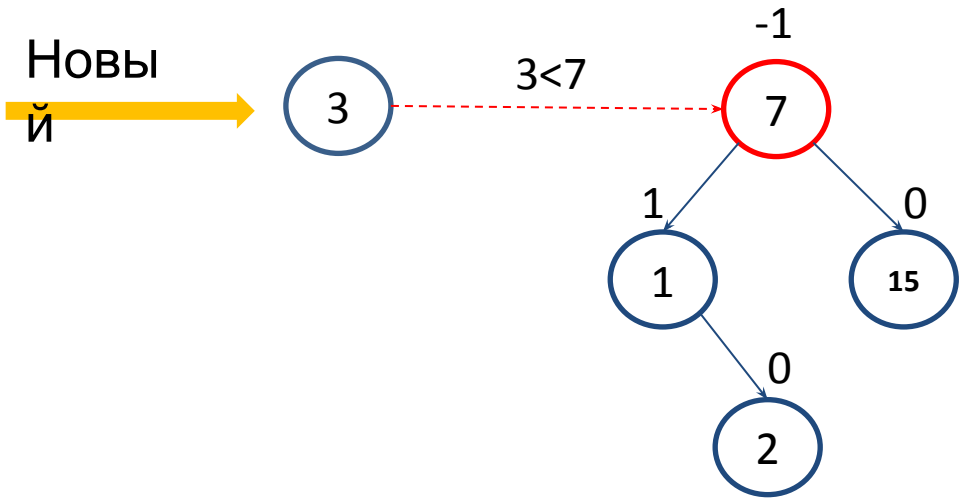
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



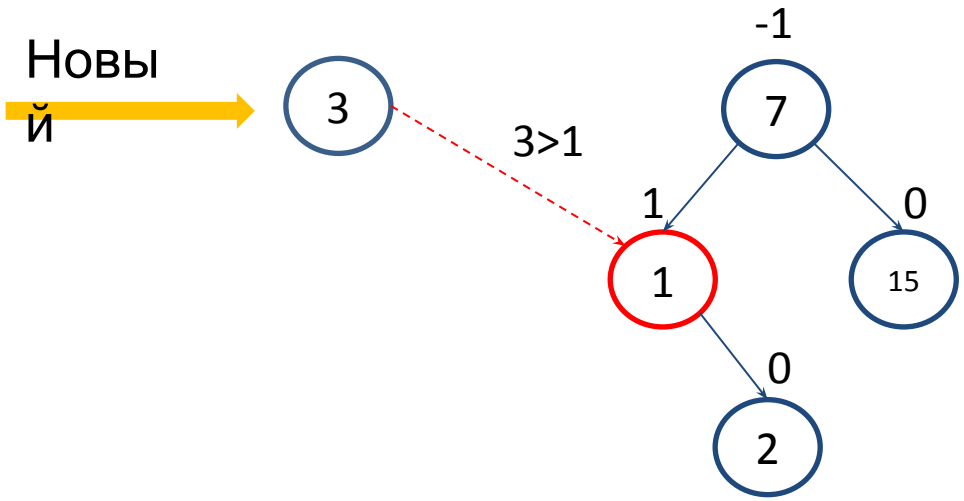
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



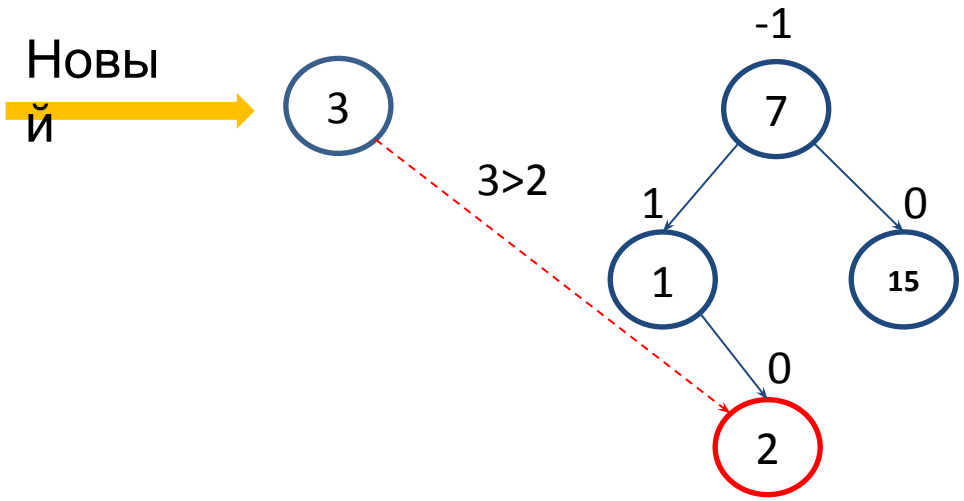
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



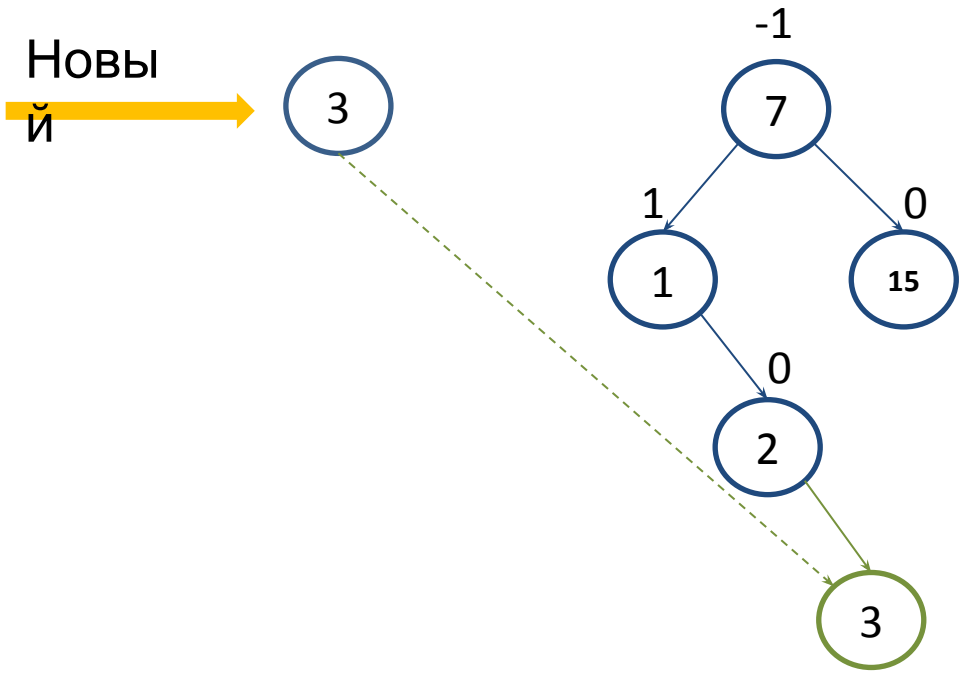
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

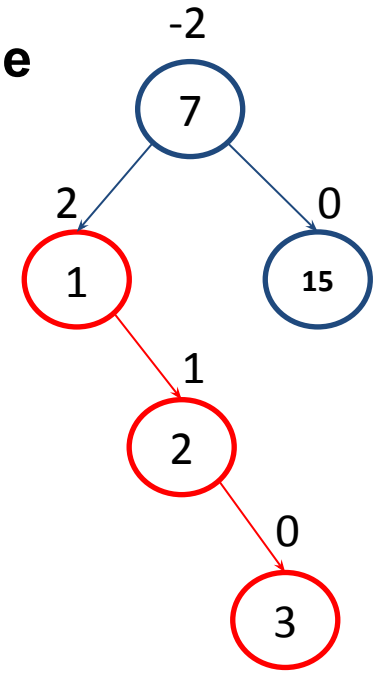
1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

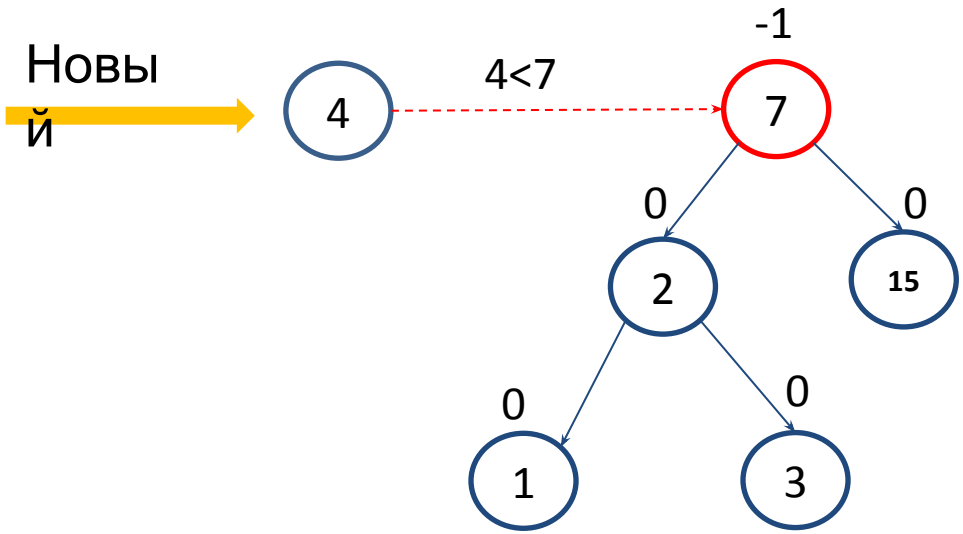
Разбалансировка в узле 1

Выполняем RR-поворот.



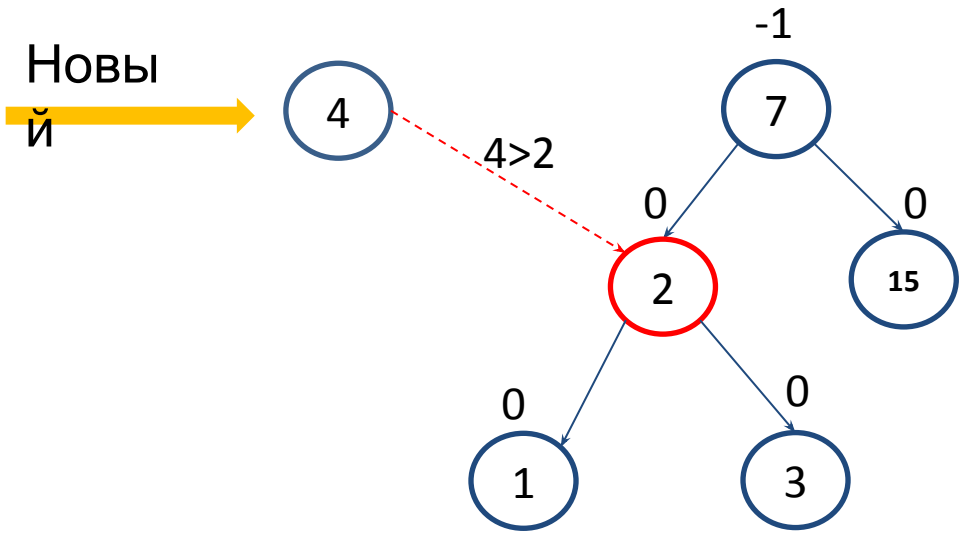
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



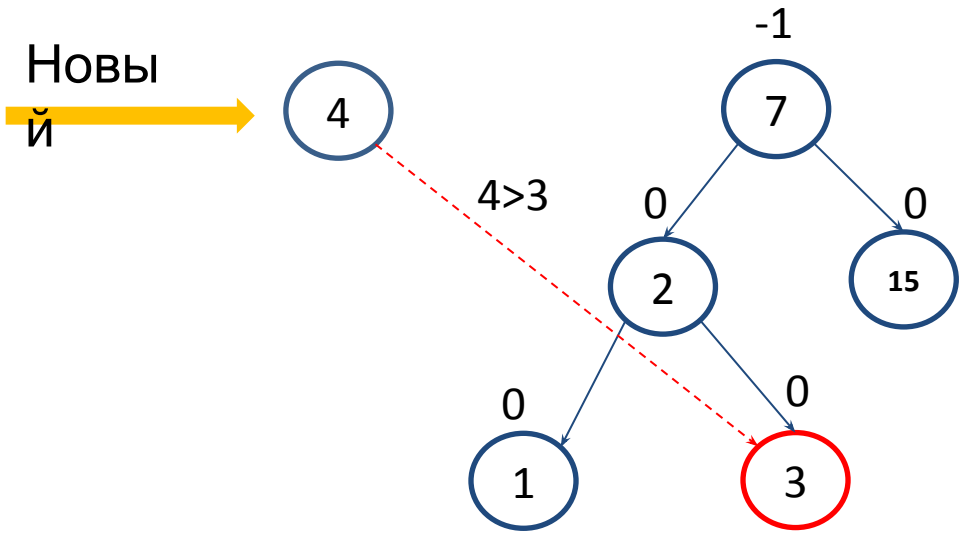
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



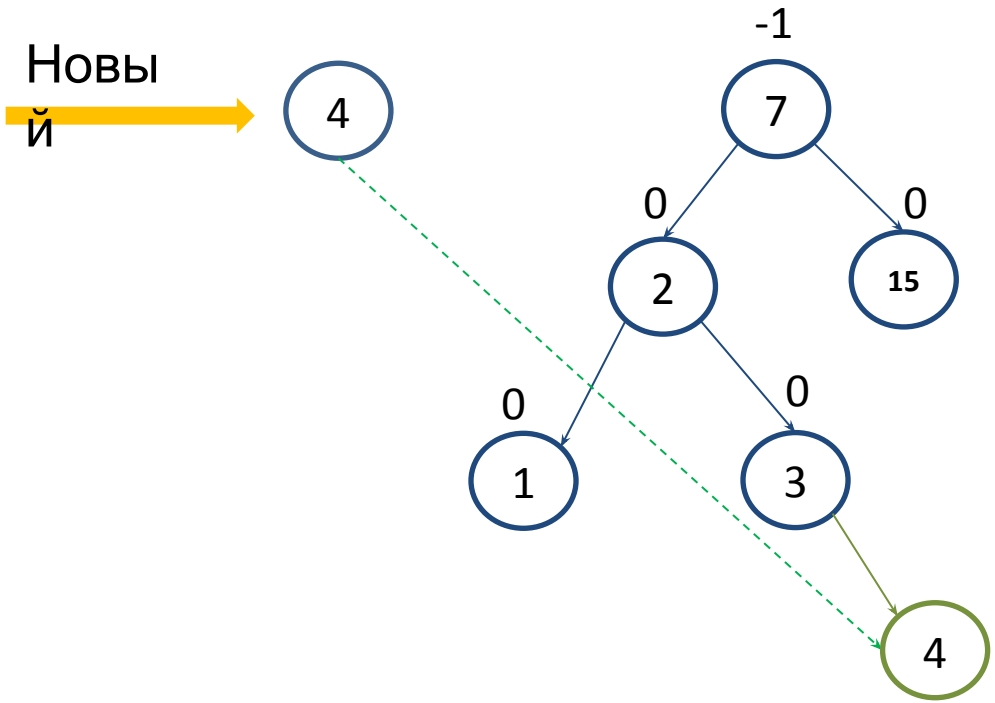
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

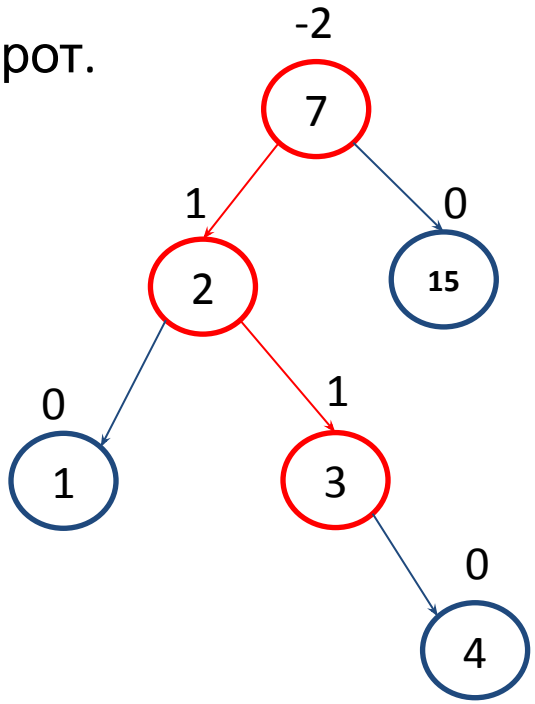
1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

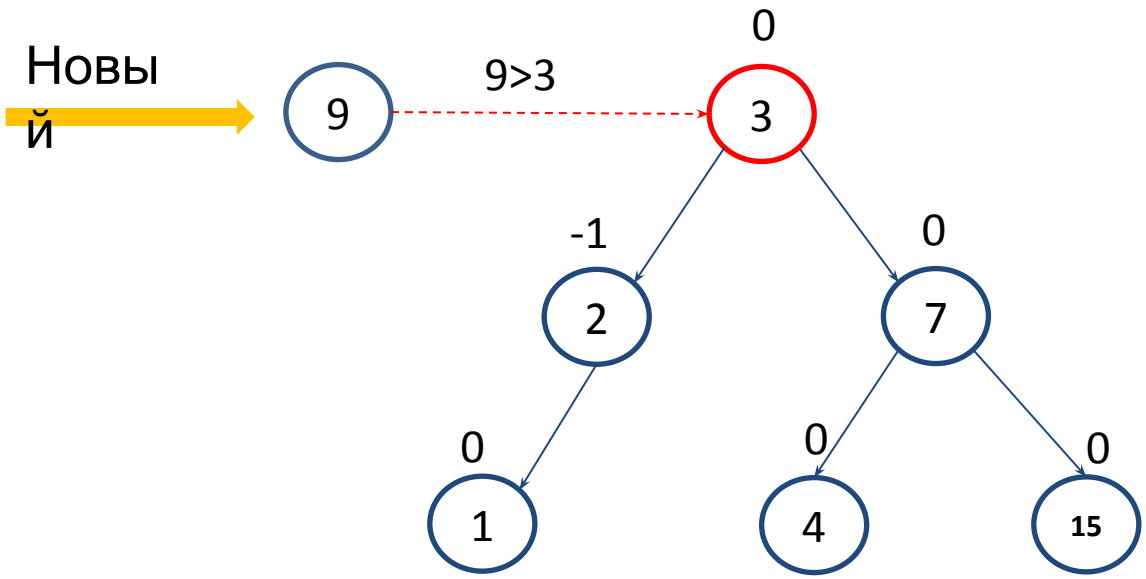
Разбалансировка в узле 7

Выполняем LR-поворот.



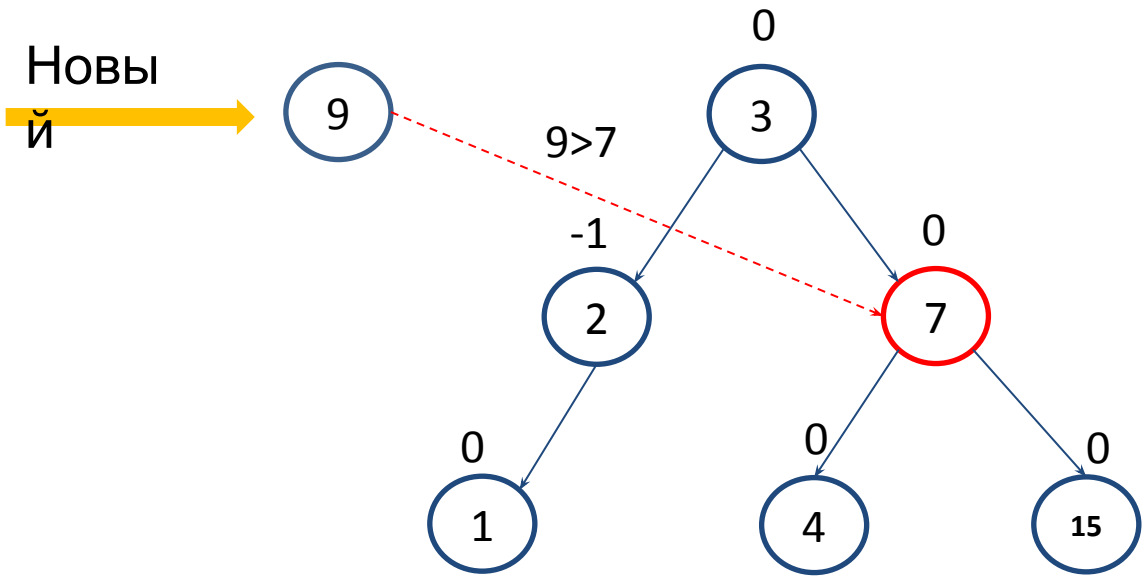
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



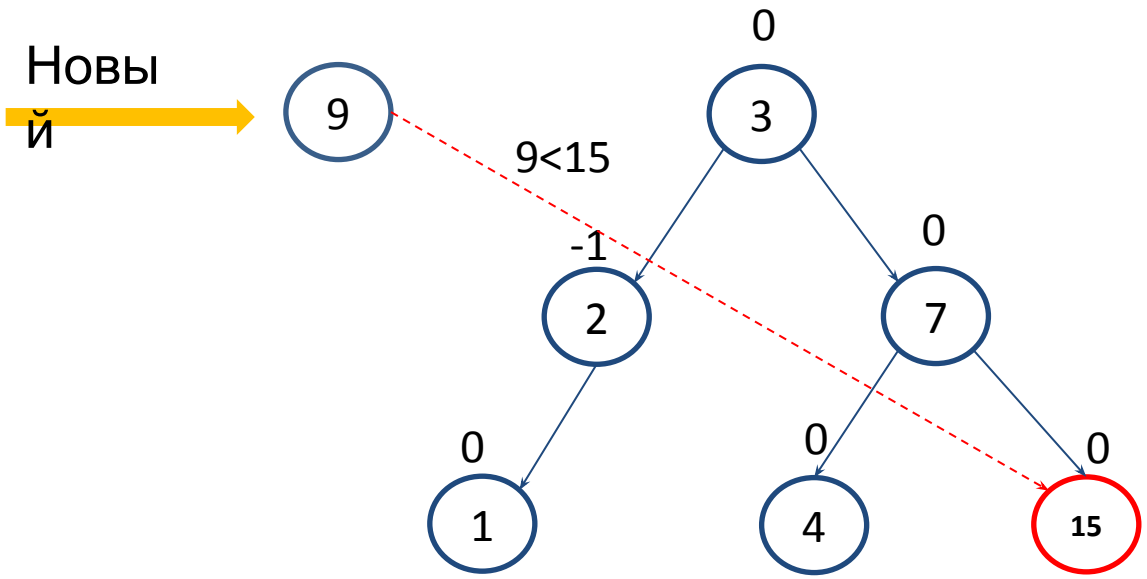
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



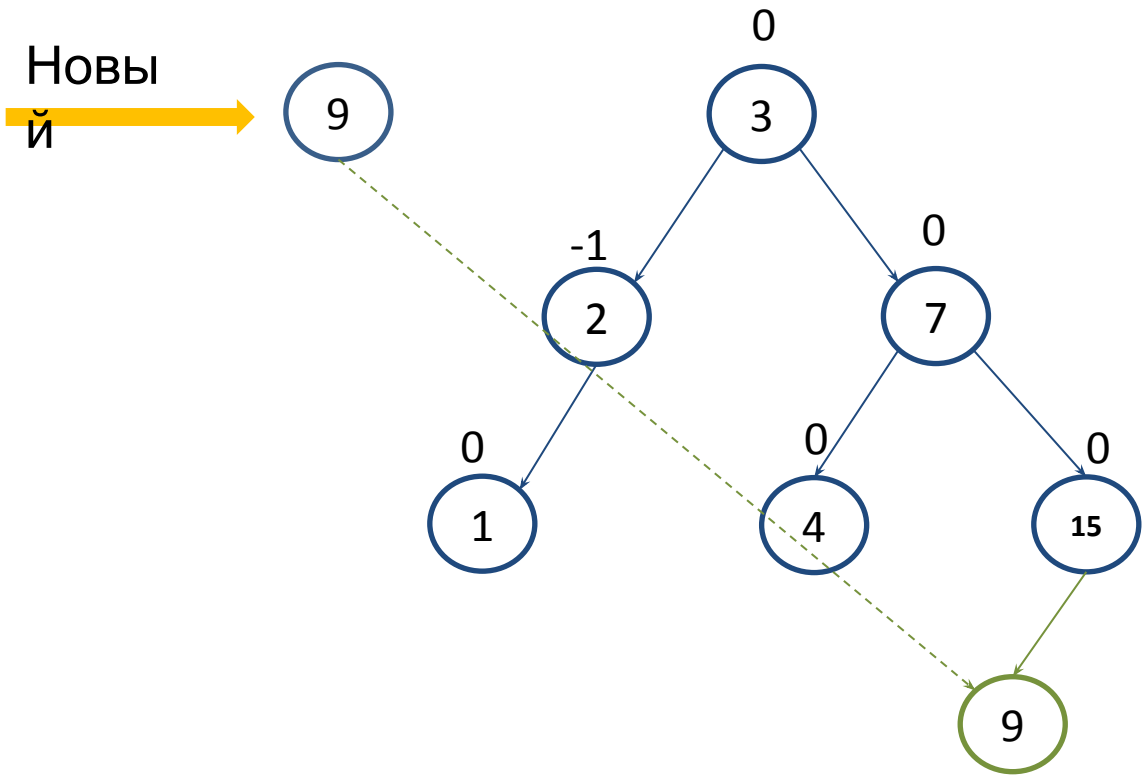
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



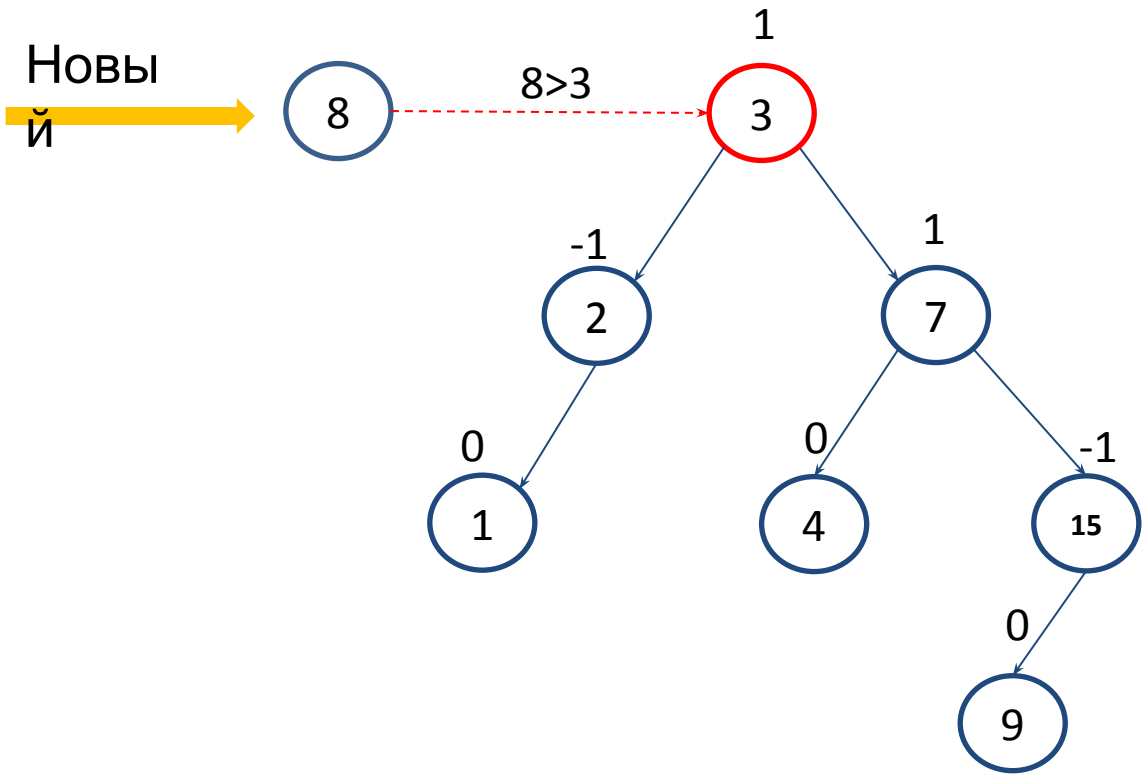
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



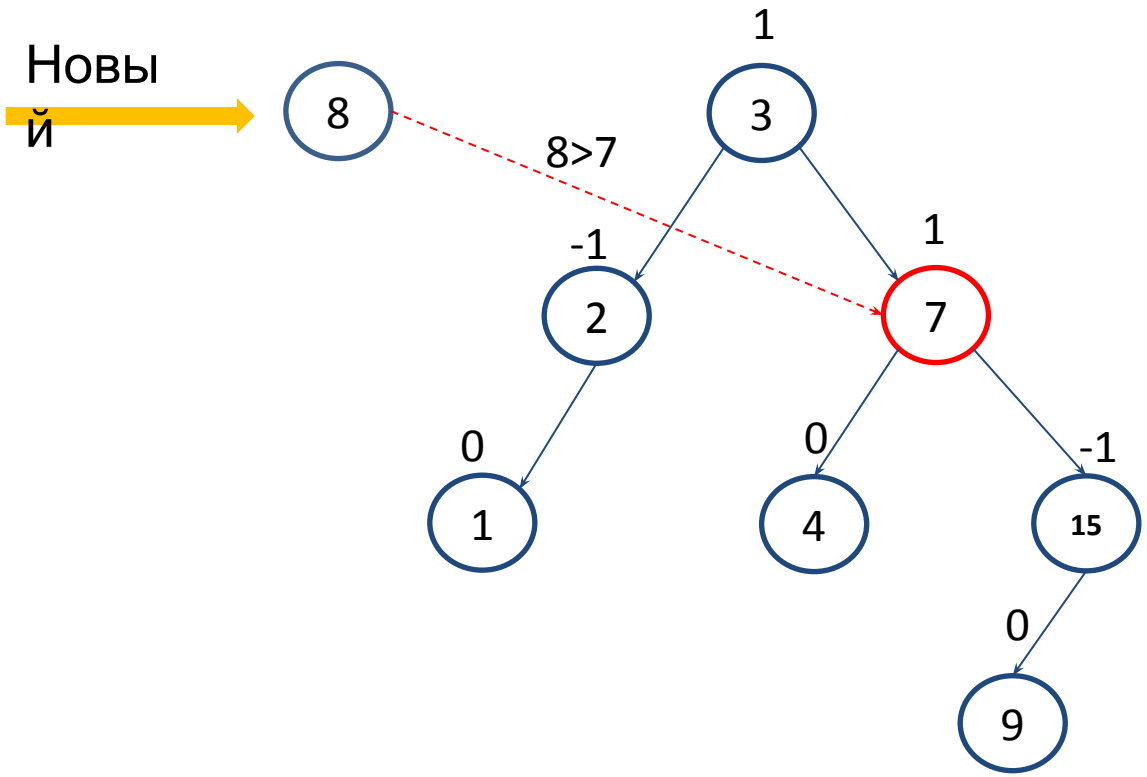
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



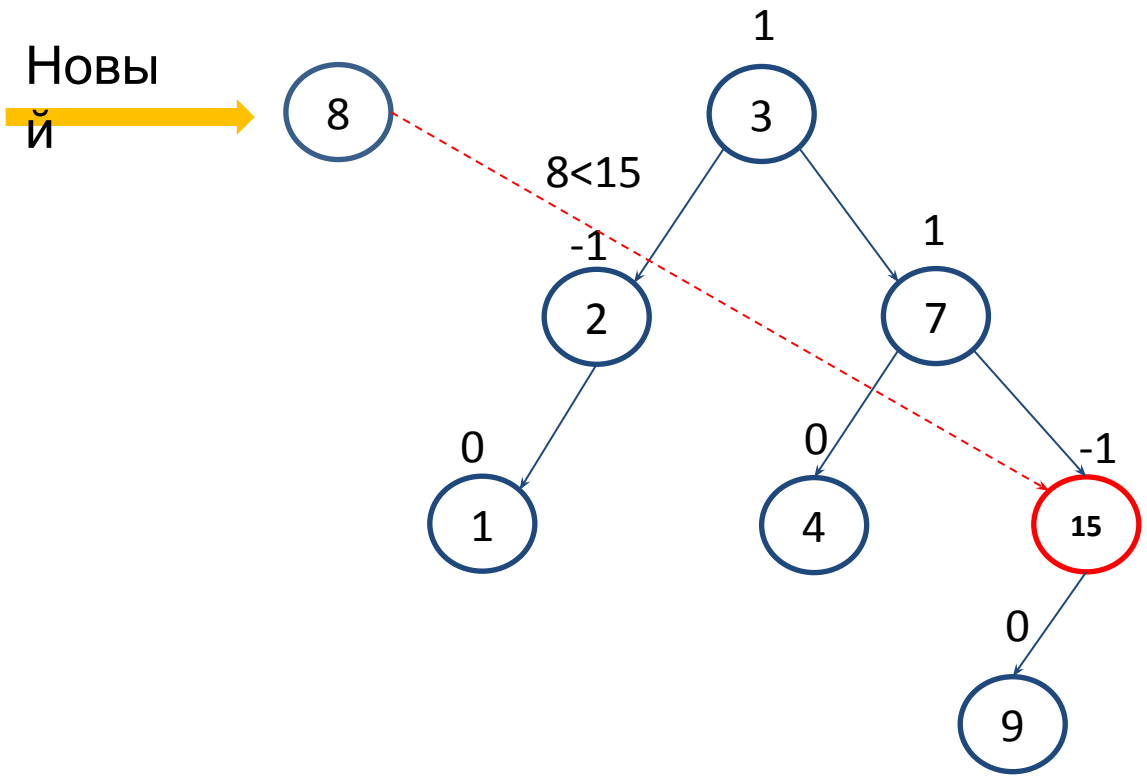
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



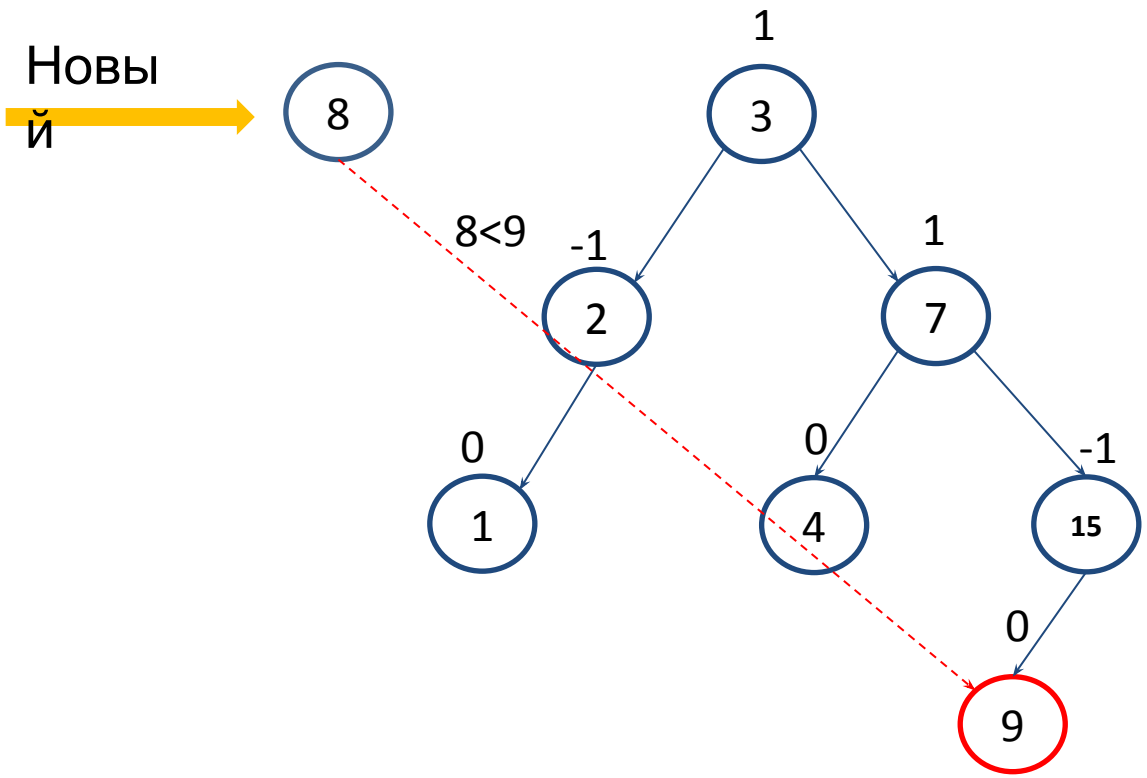
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



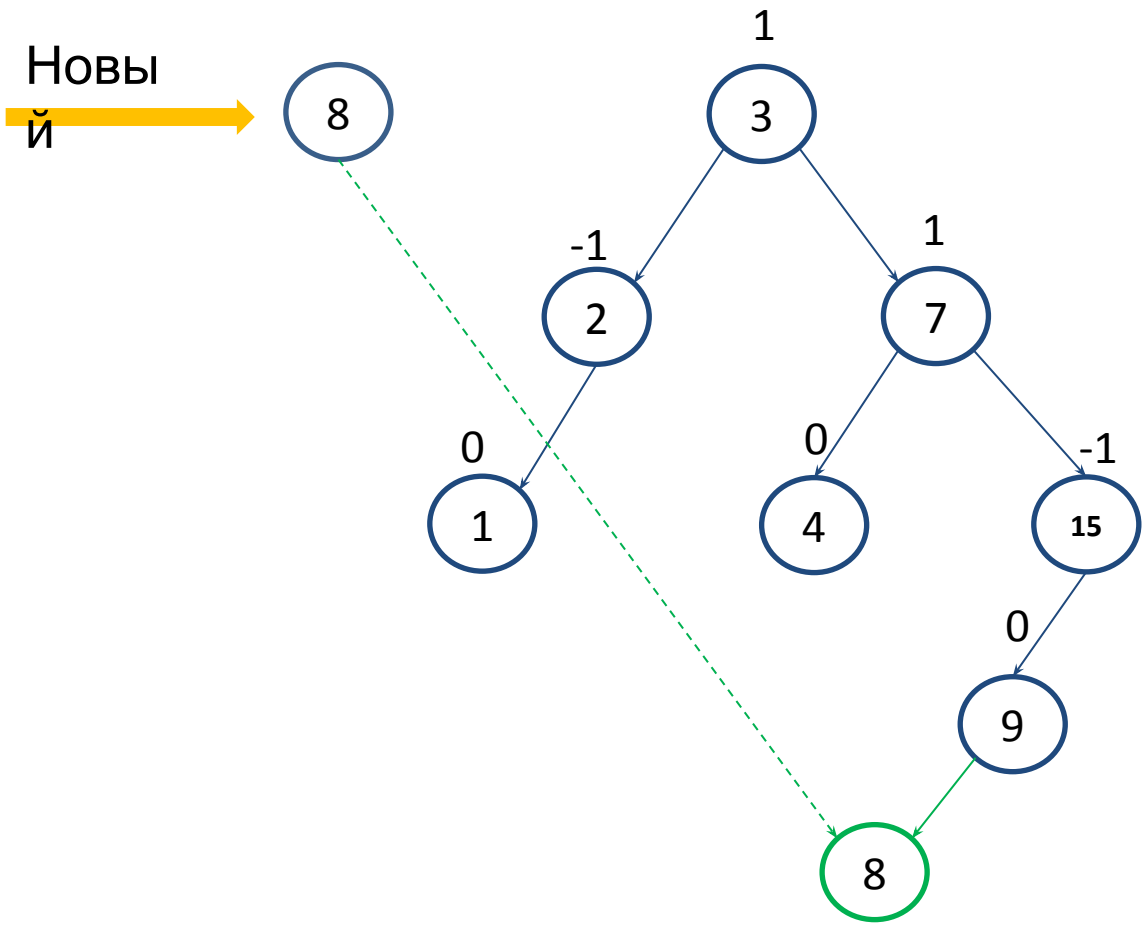
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

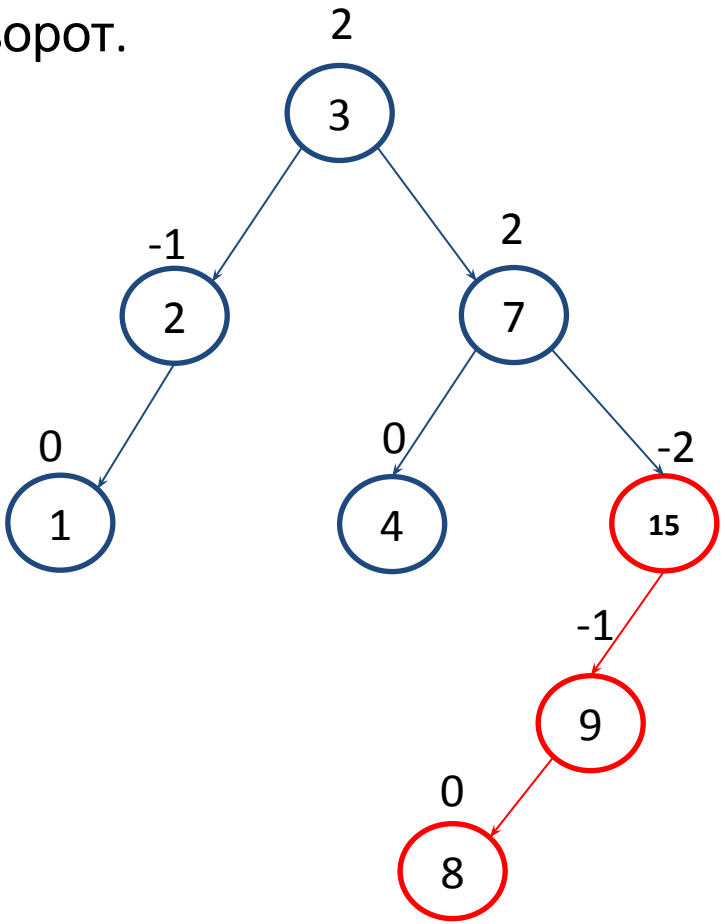
1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

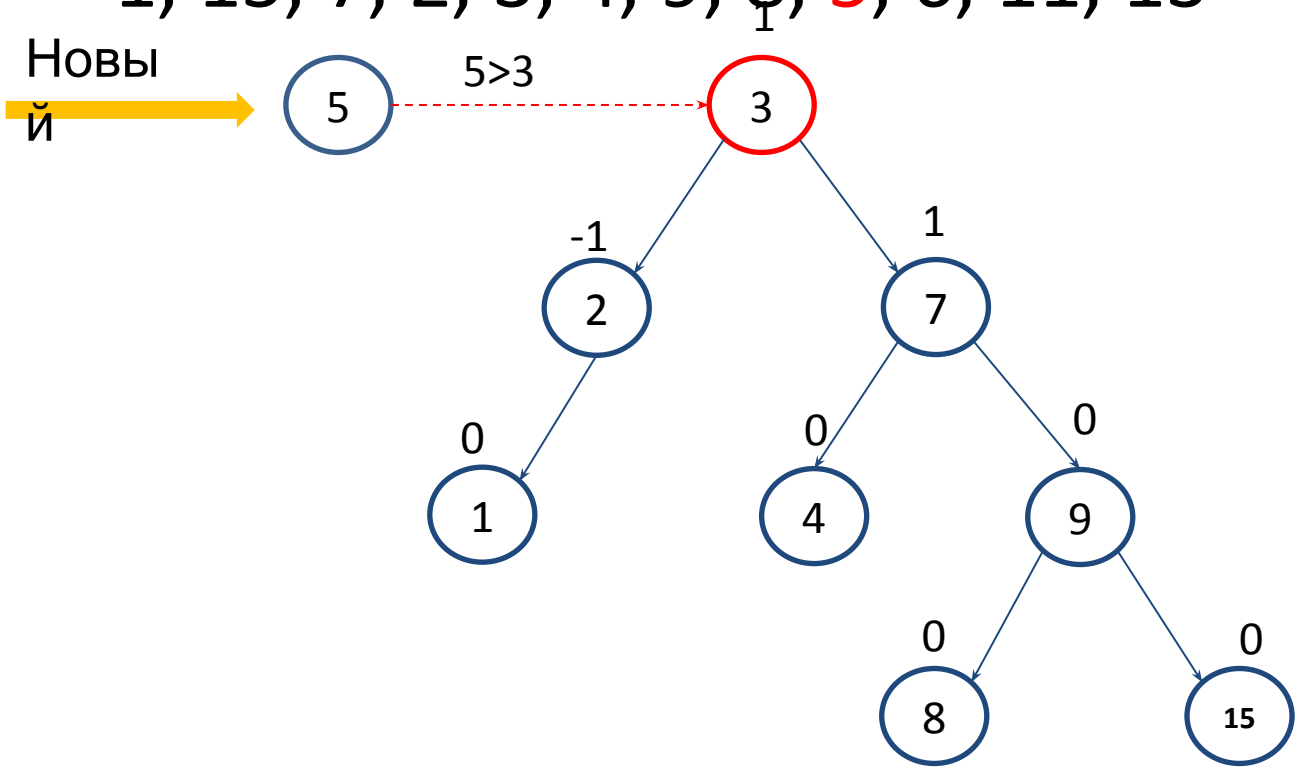
Разбалансировка в узле 15

Выполняем LL-поворот.



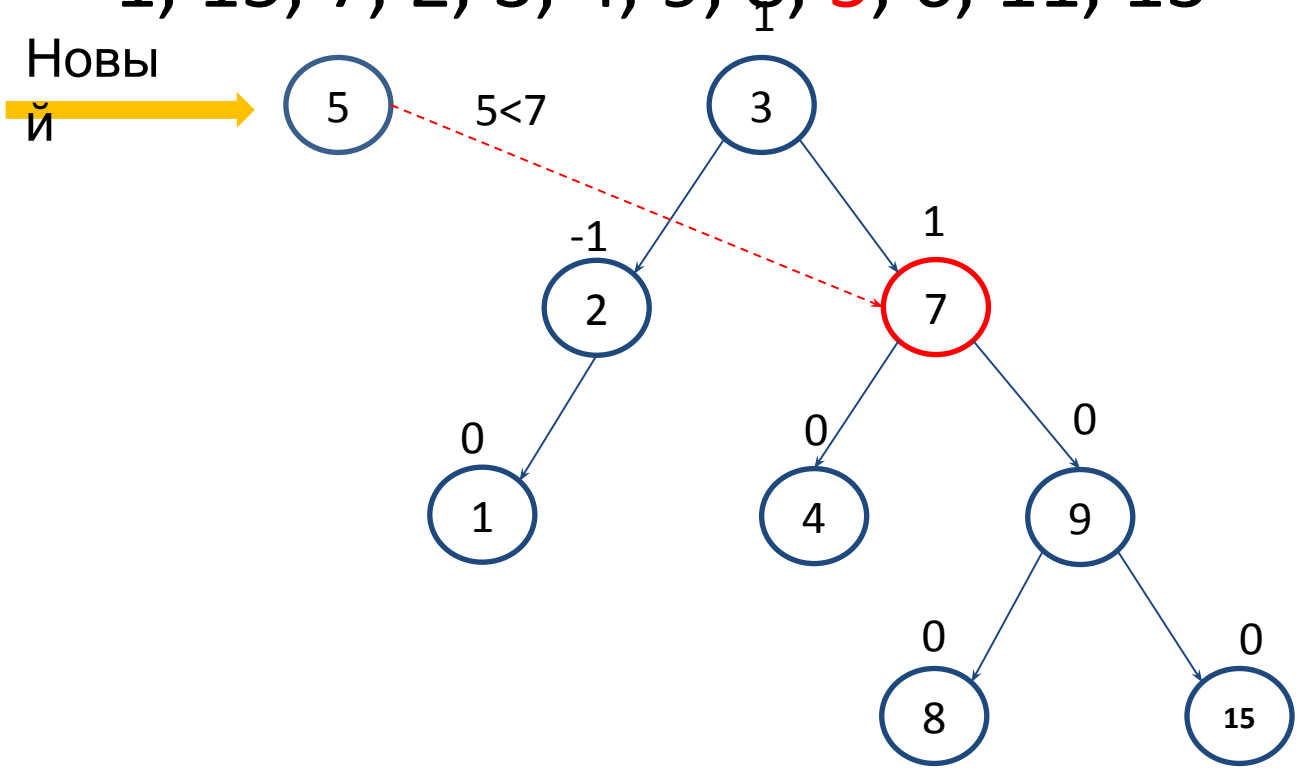
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



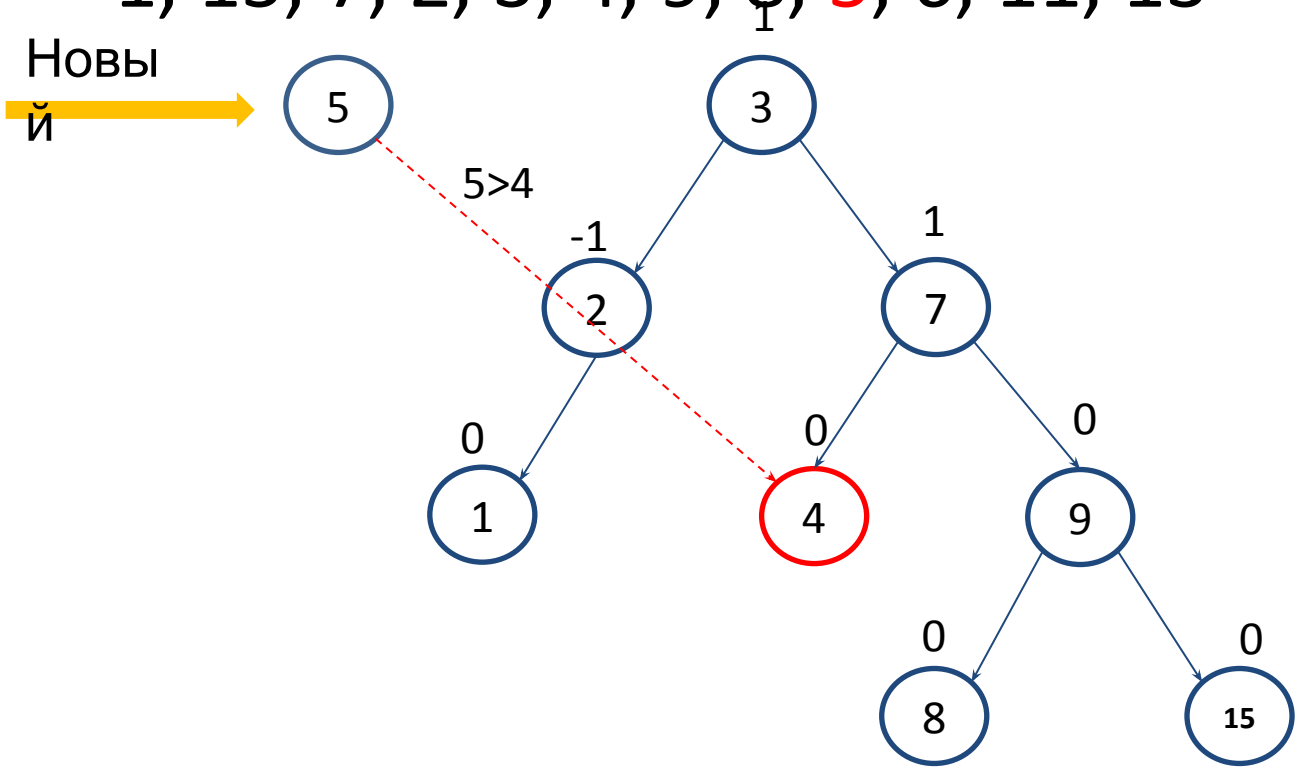
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



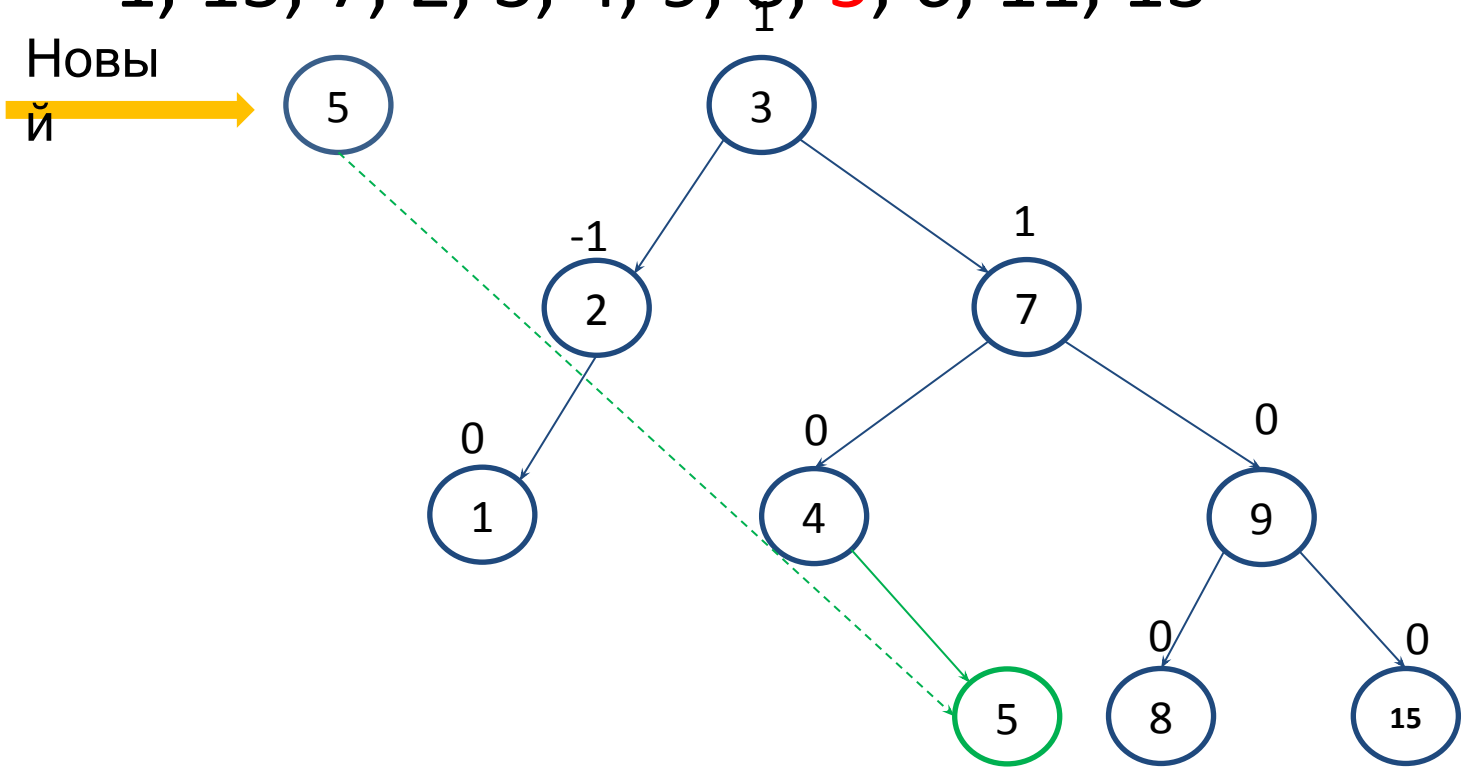
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



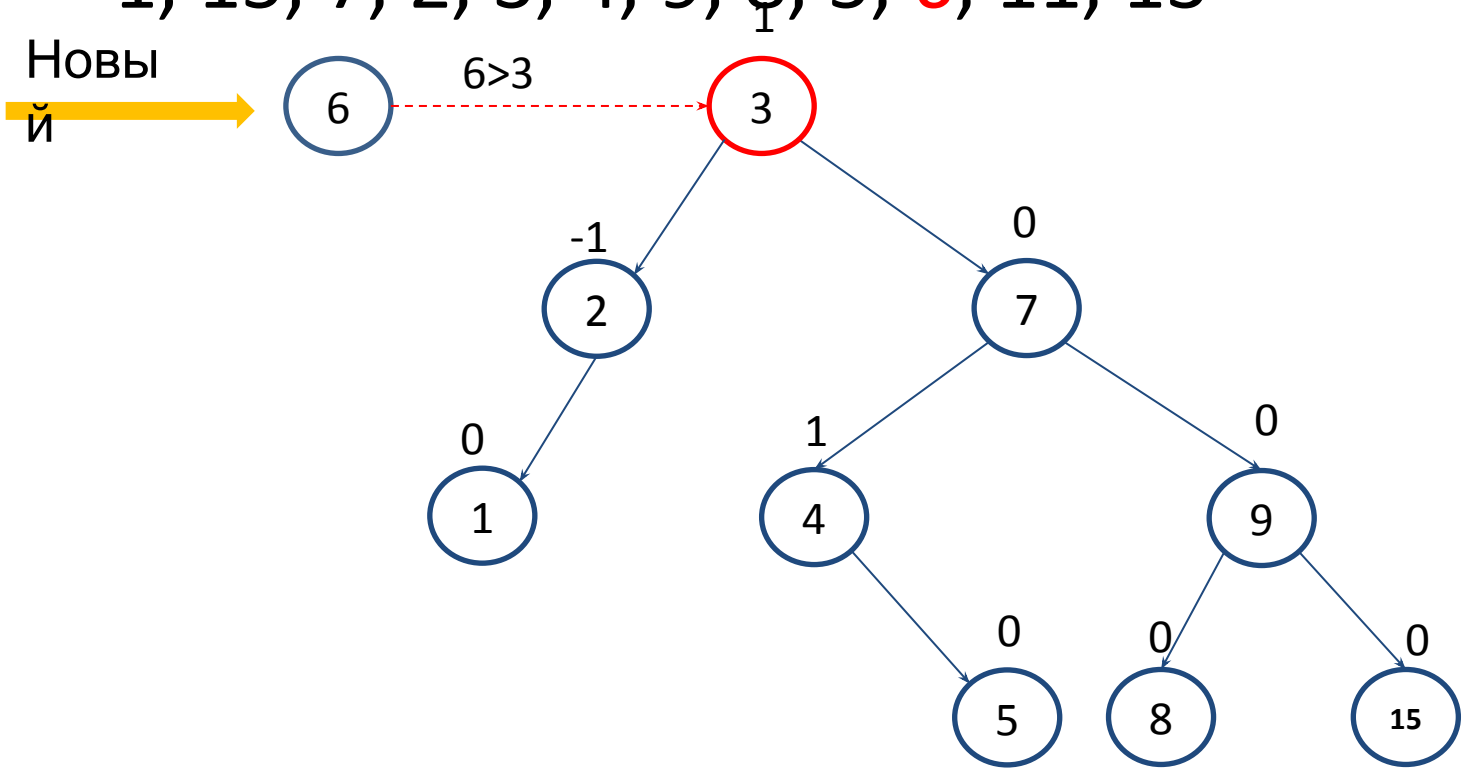
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



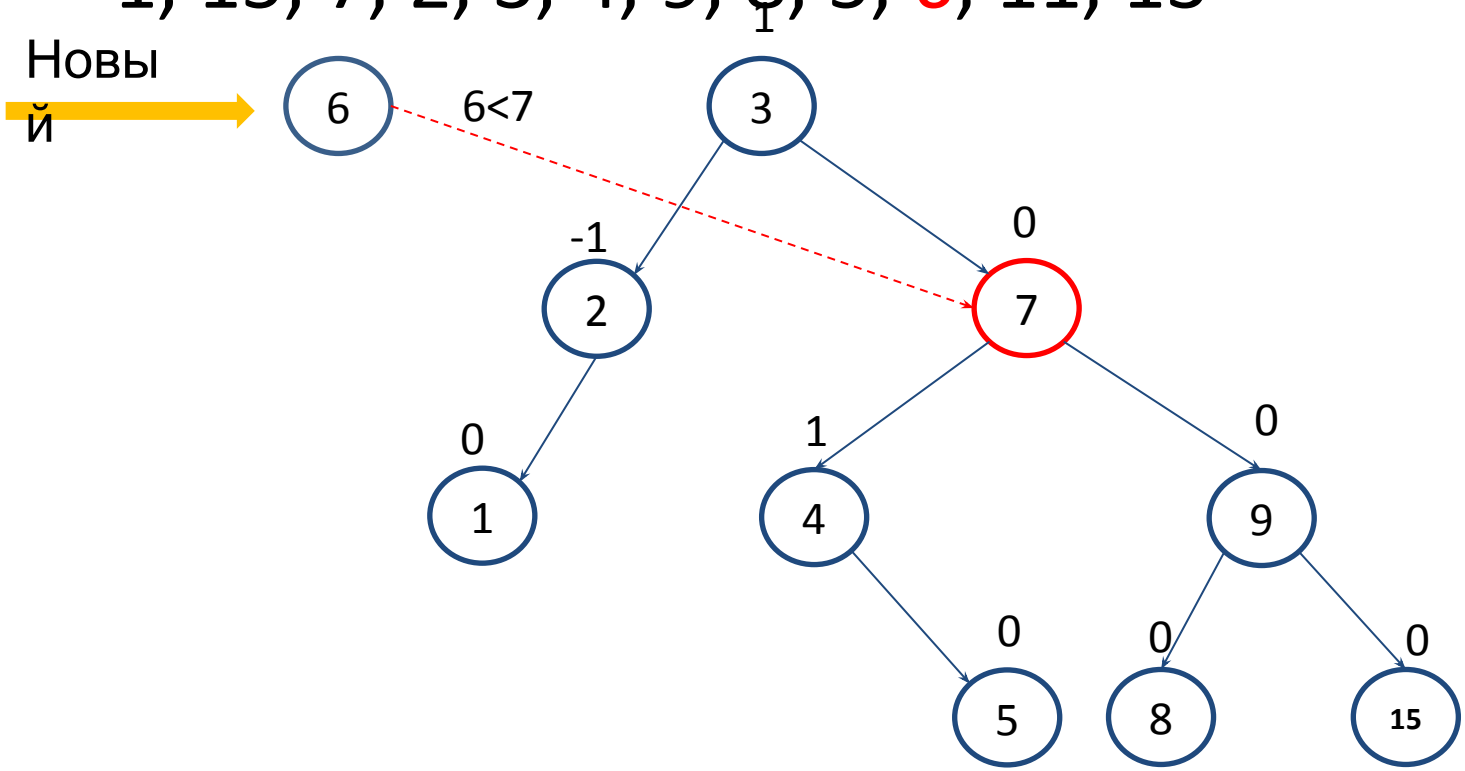
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



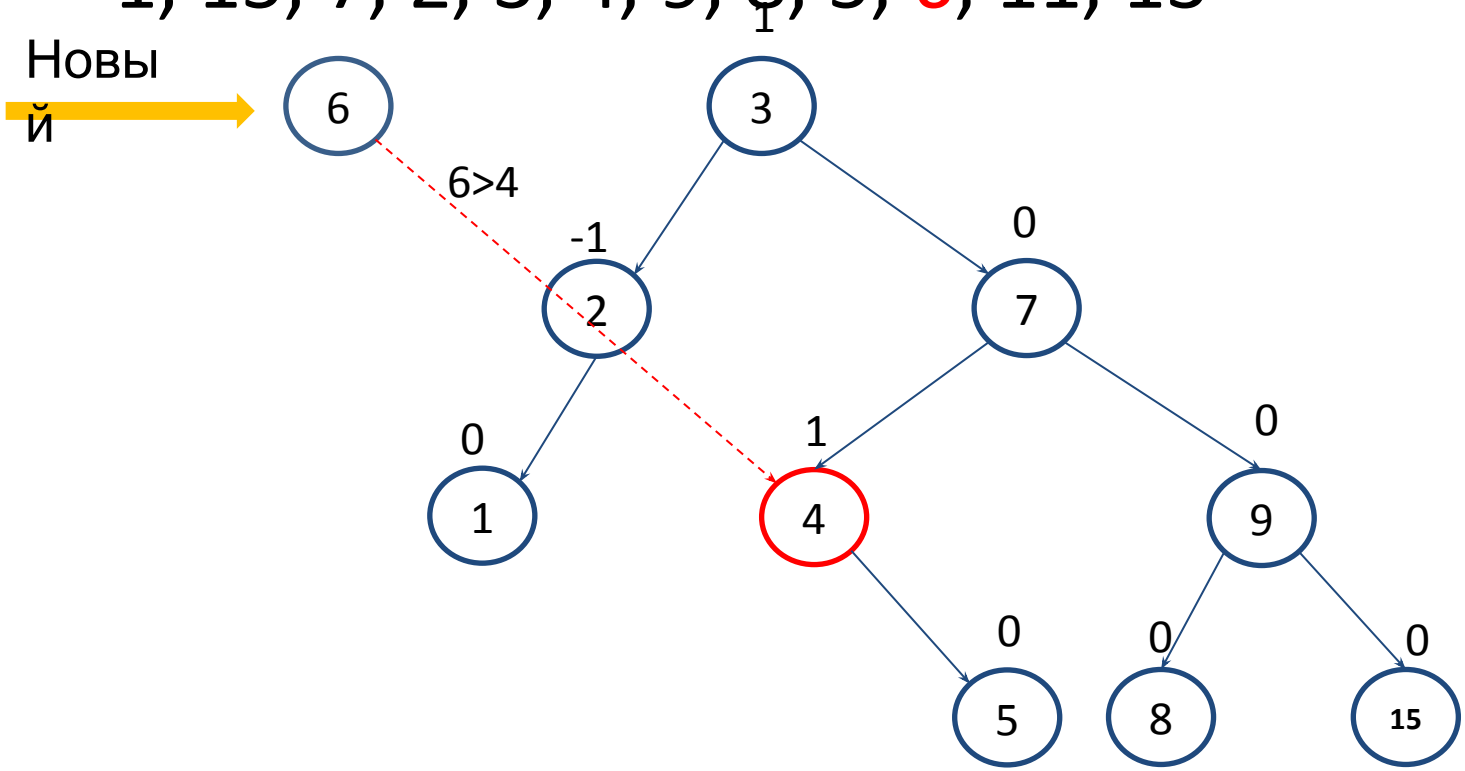
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



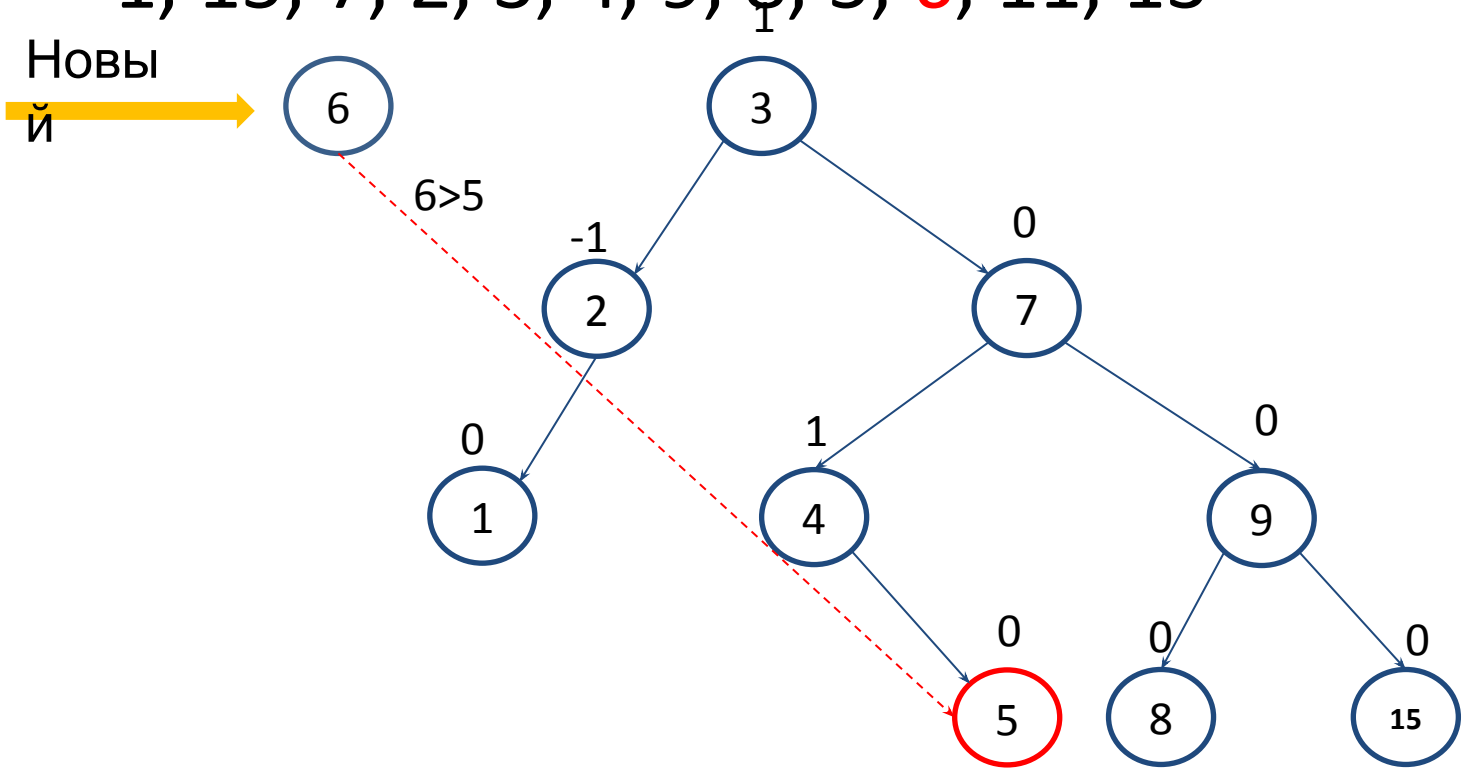
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



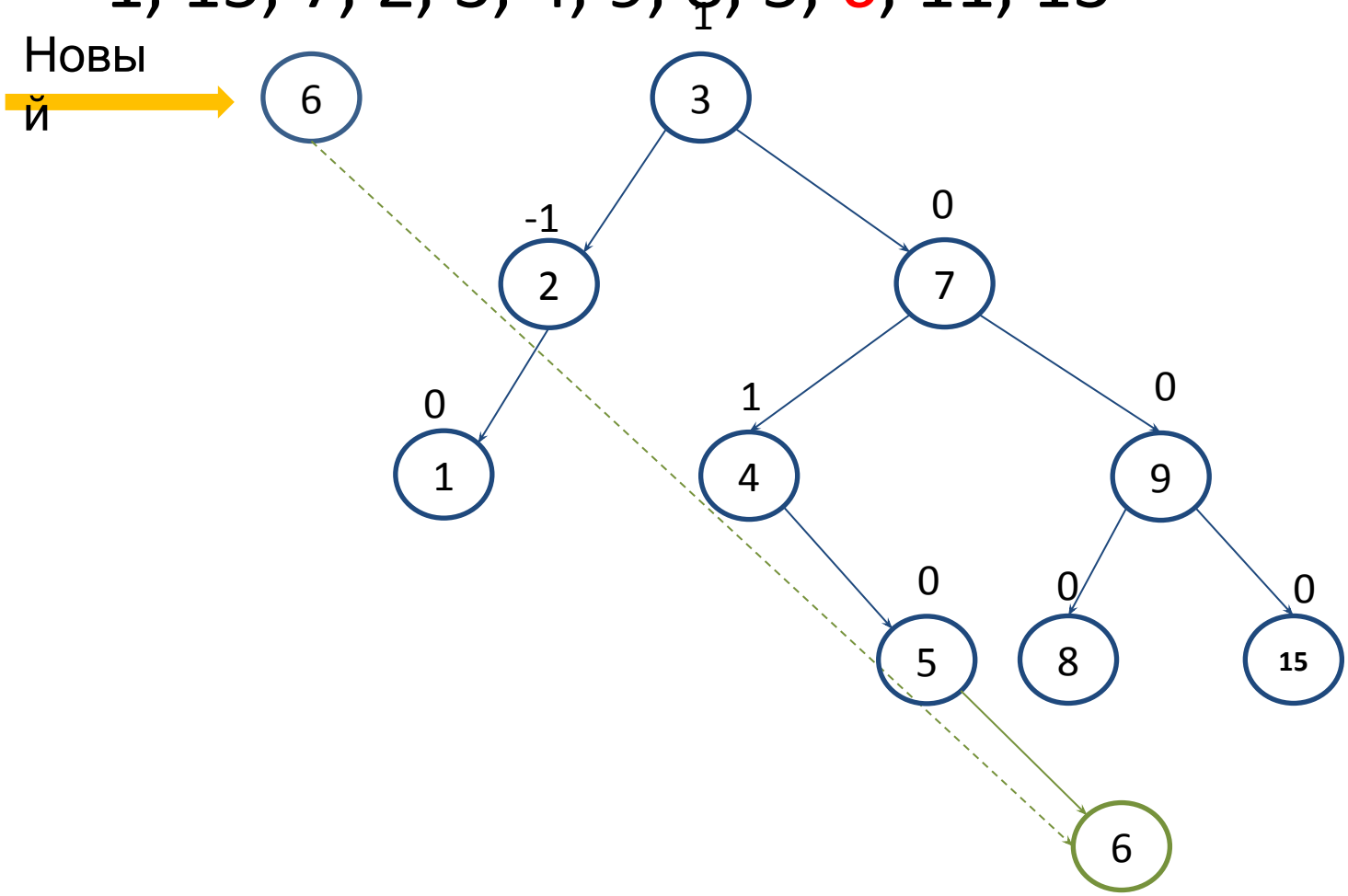
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

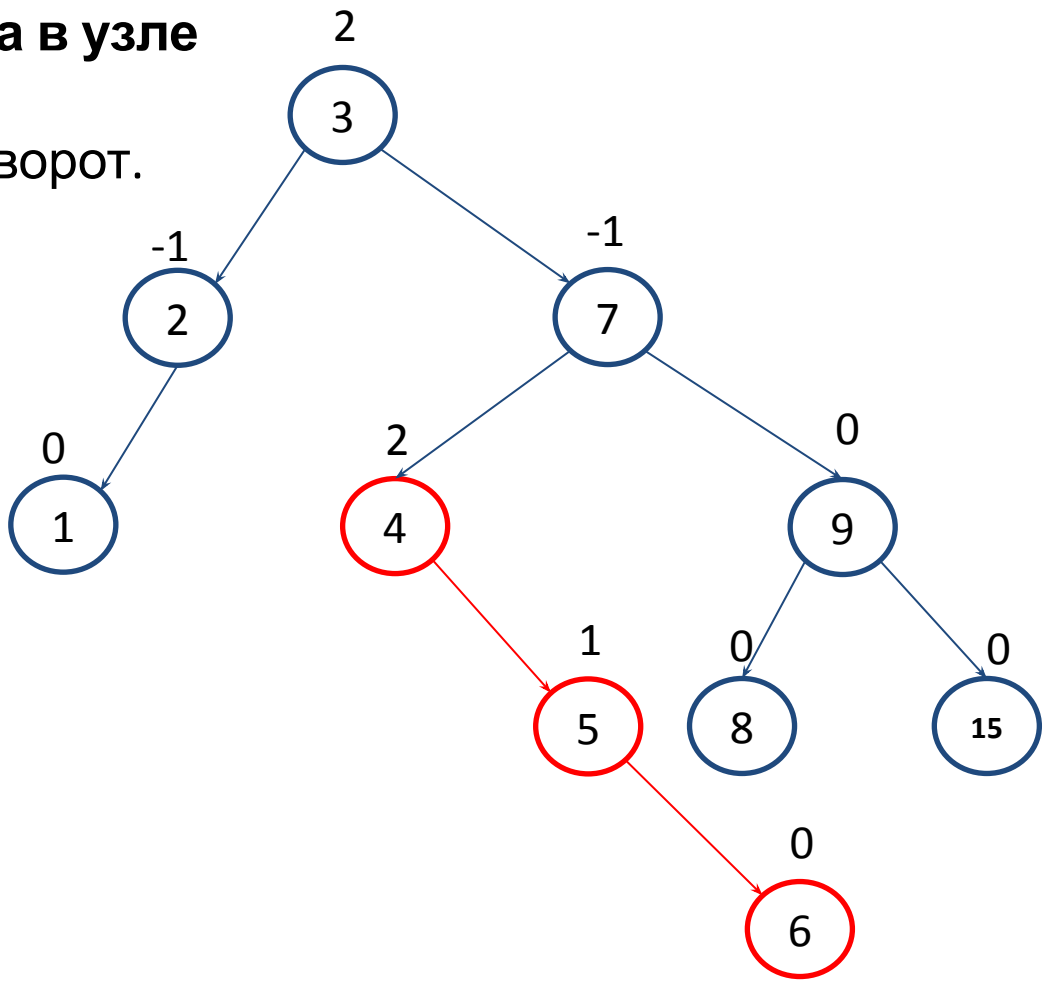
1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

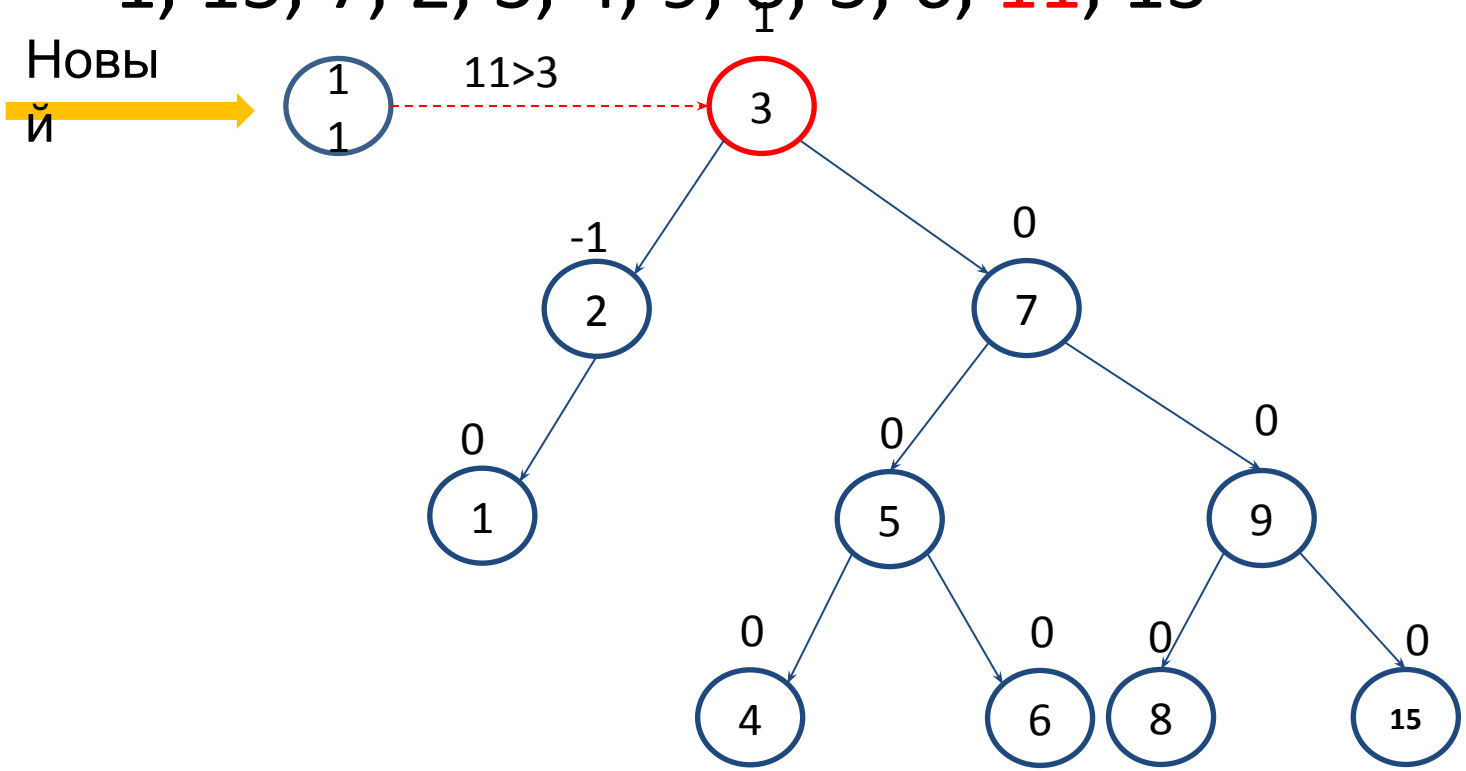
Разбалансировка в узле 4

Выполняем RR-поворот.



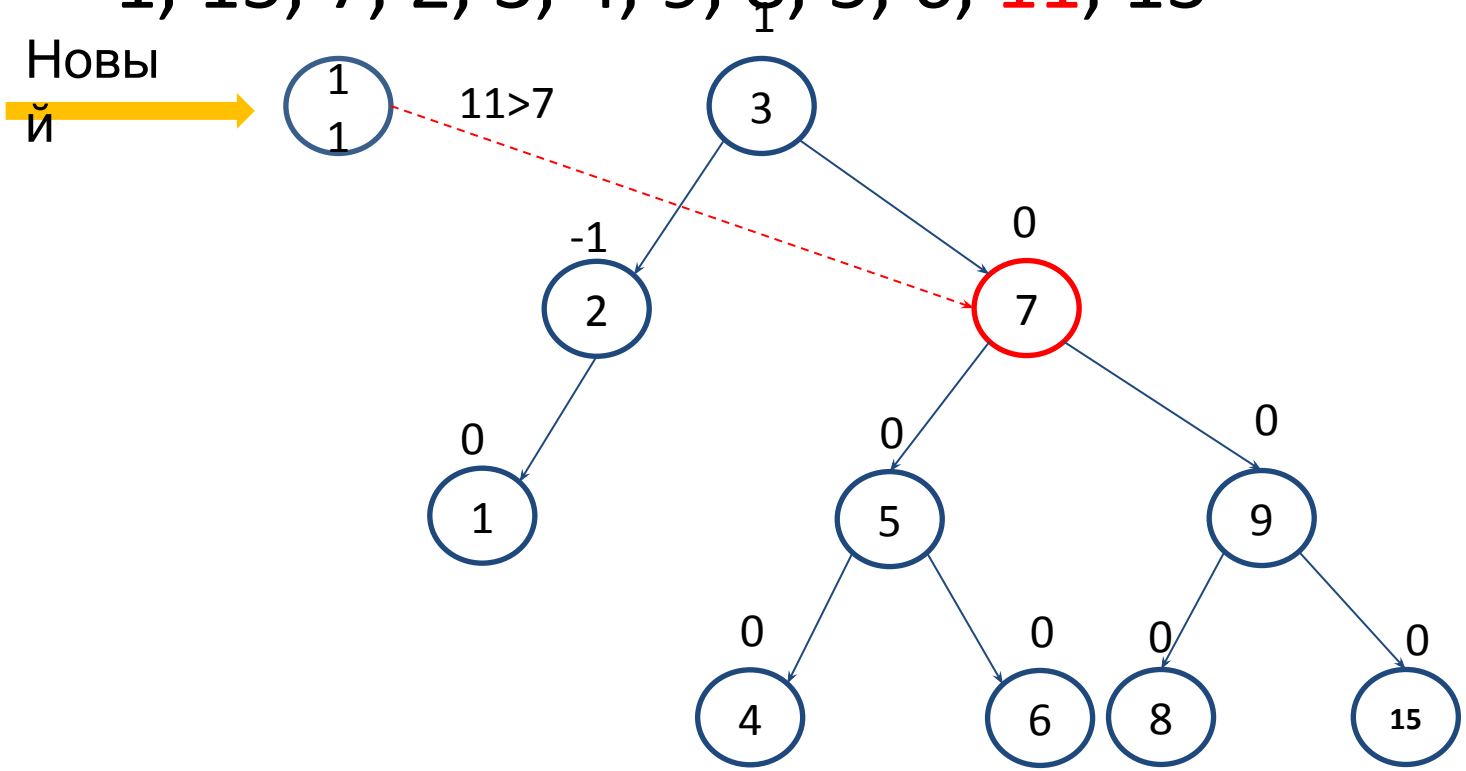
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



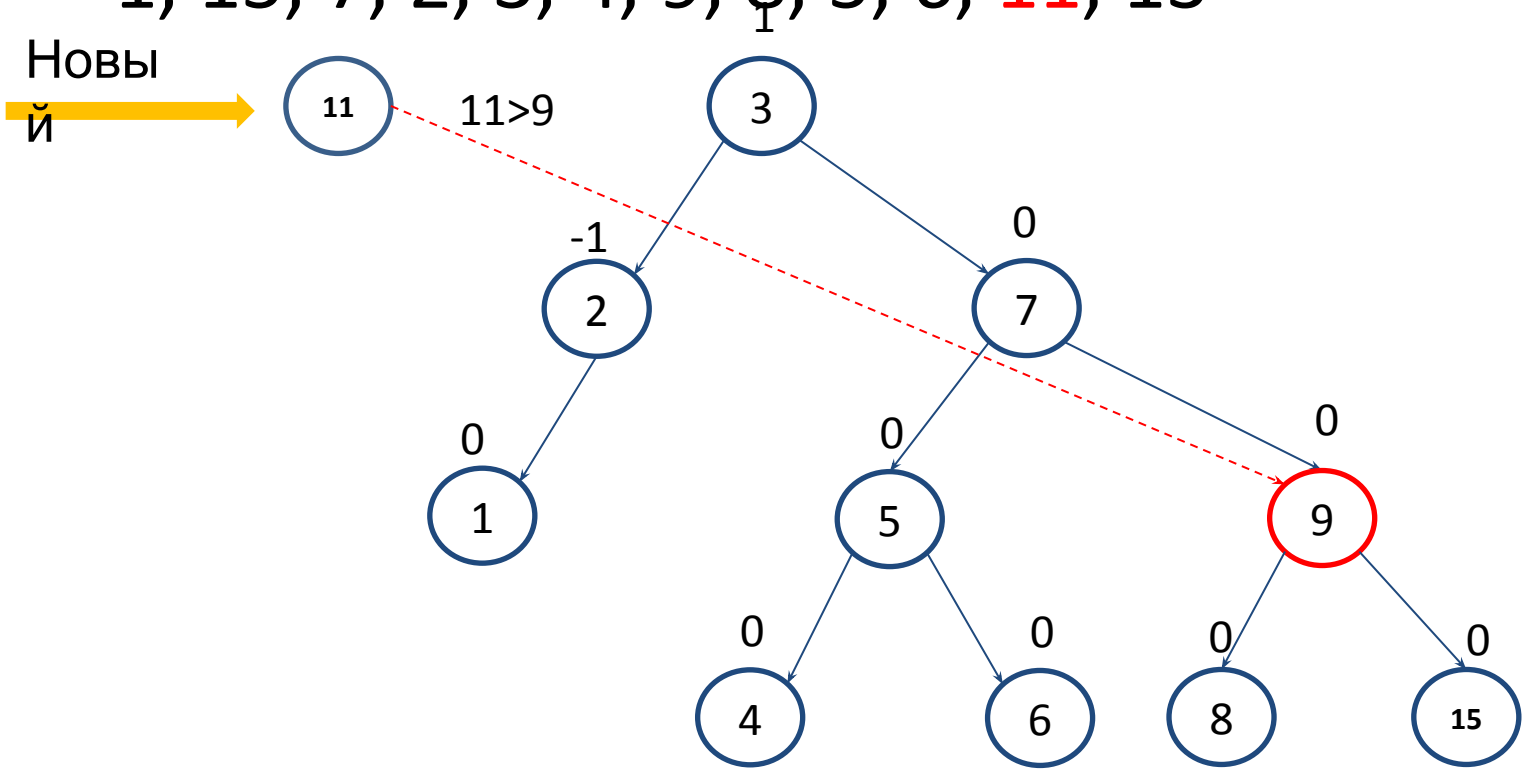
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



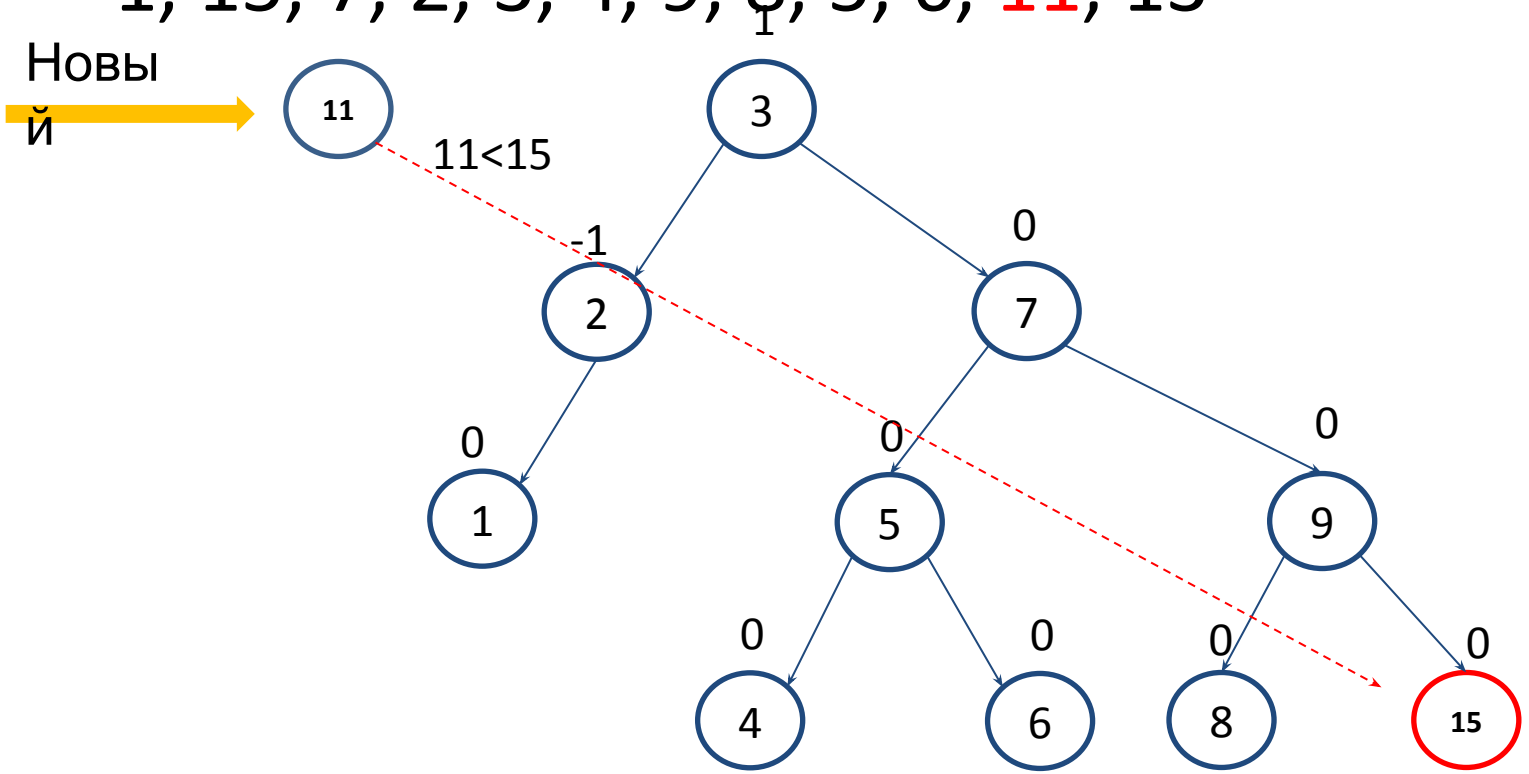
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



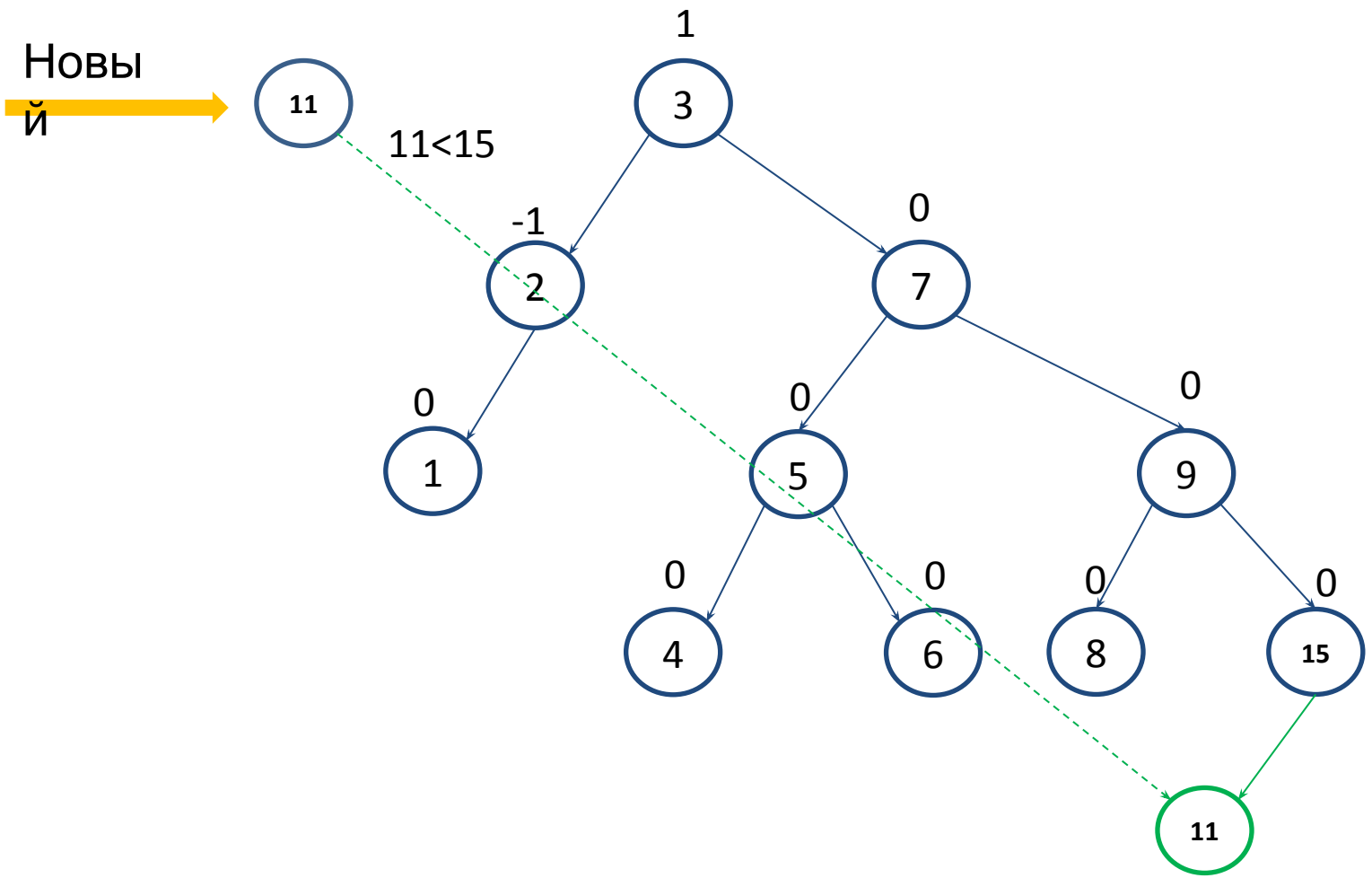
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

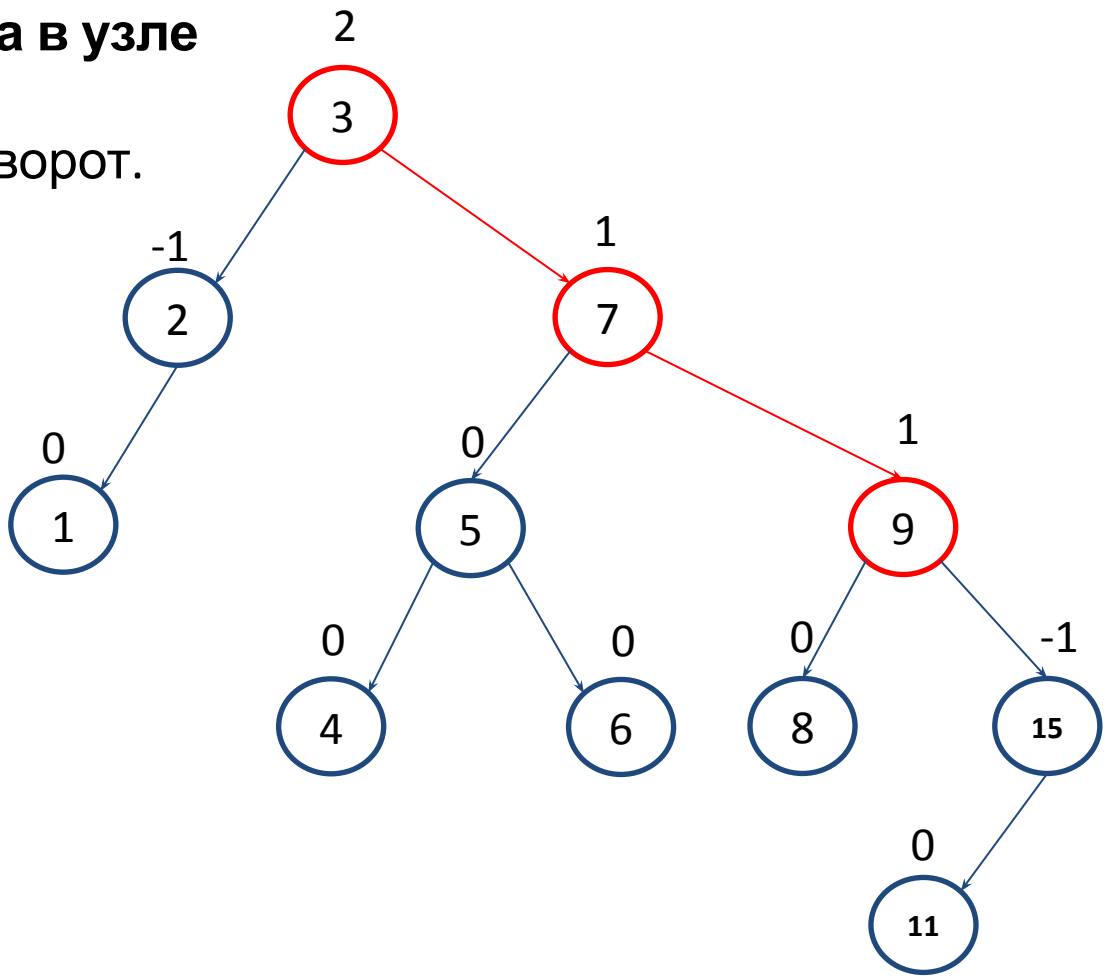
1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

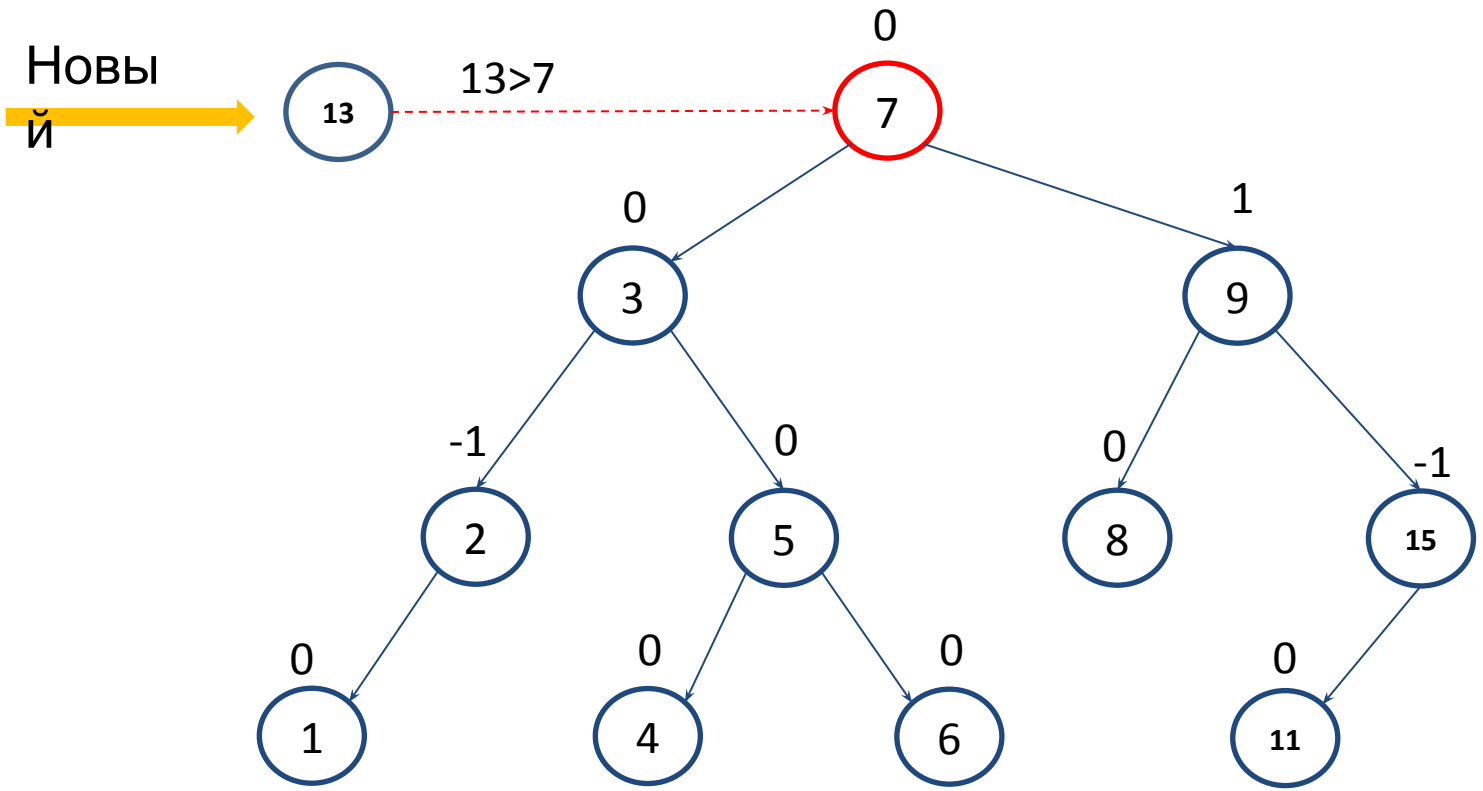
Разбалансировка в узле 3

Выполняем RR-поворот.



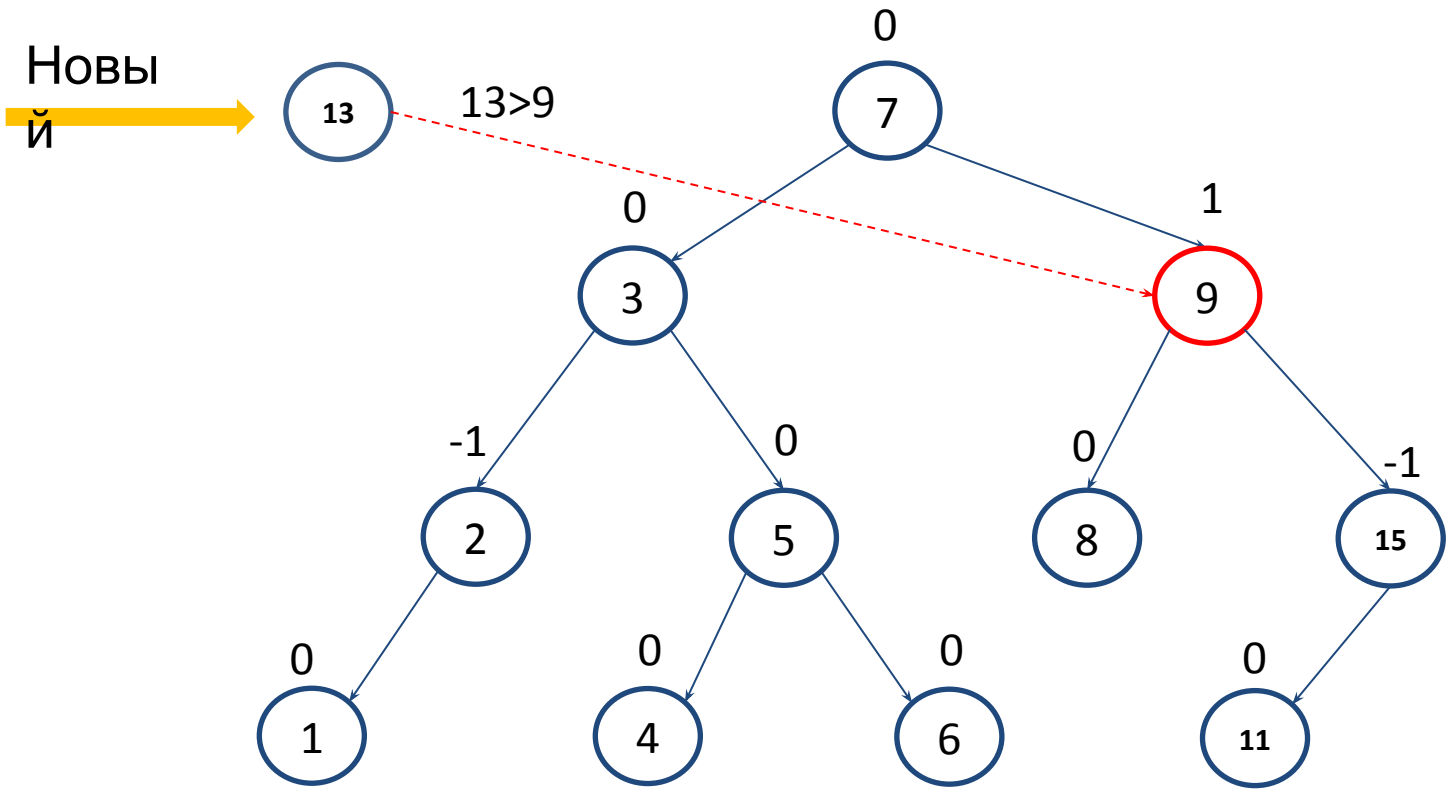
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



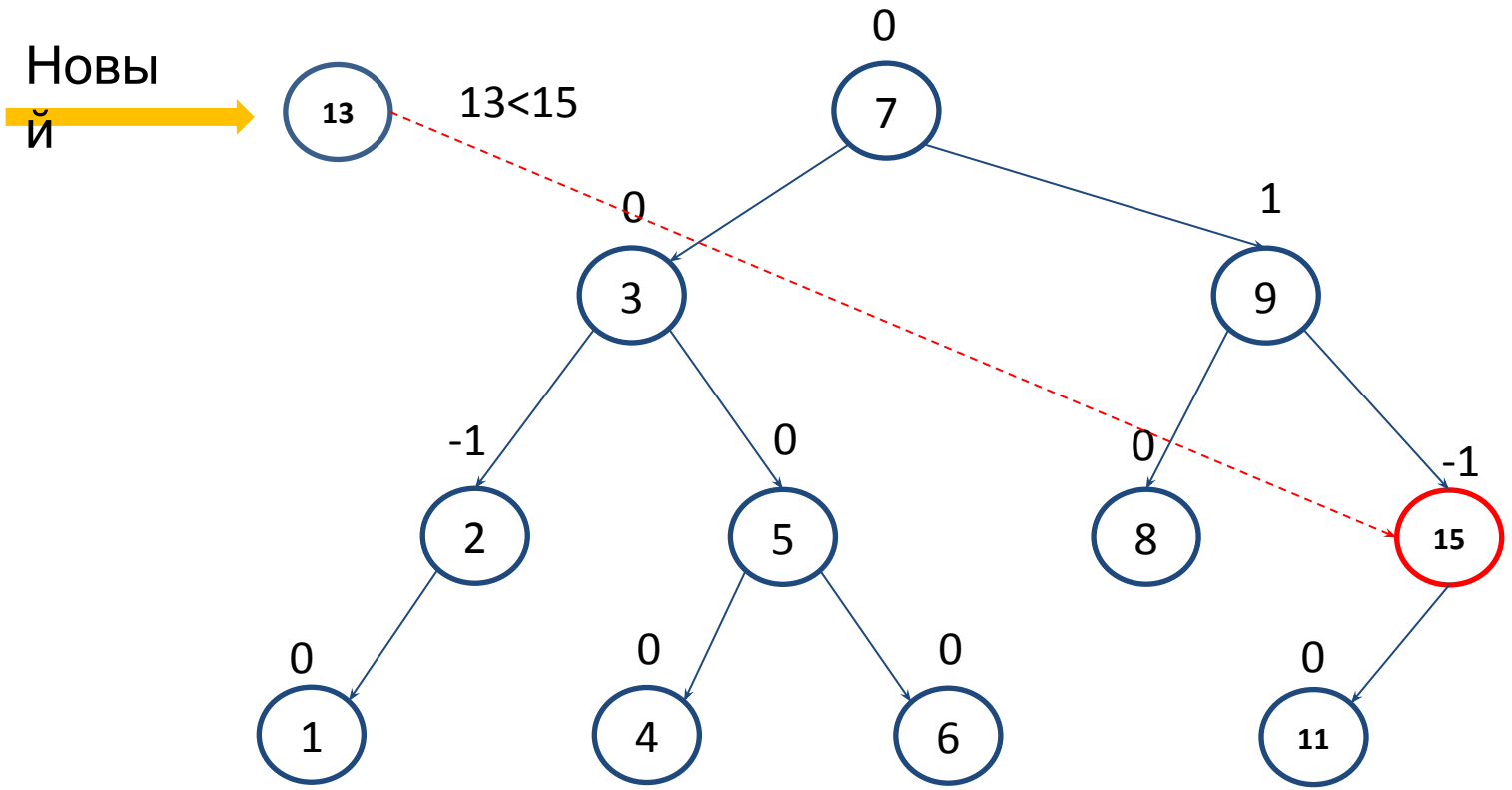
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



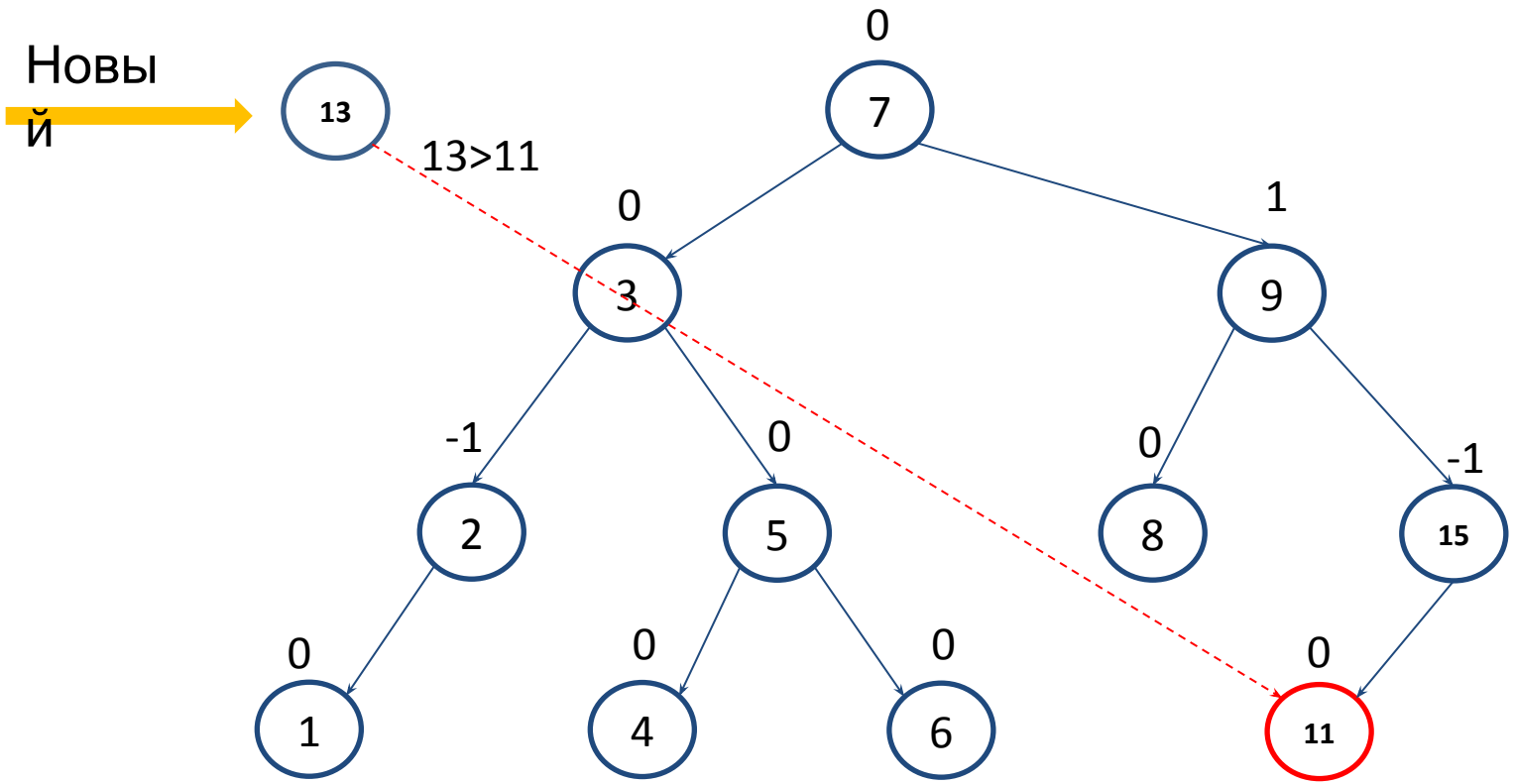
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



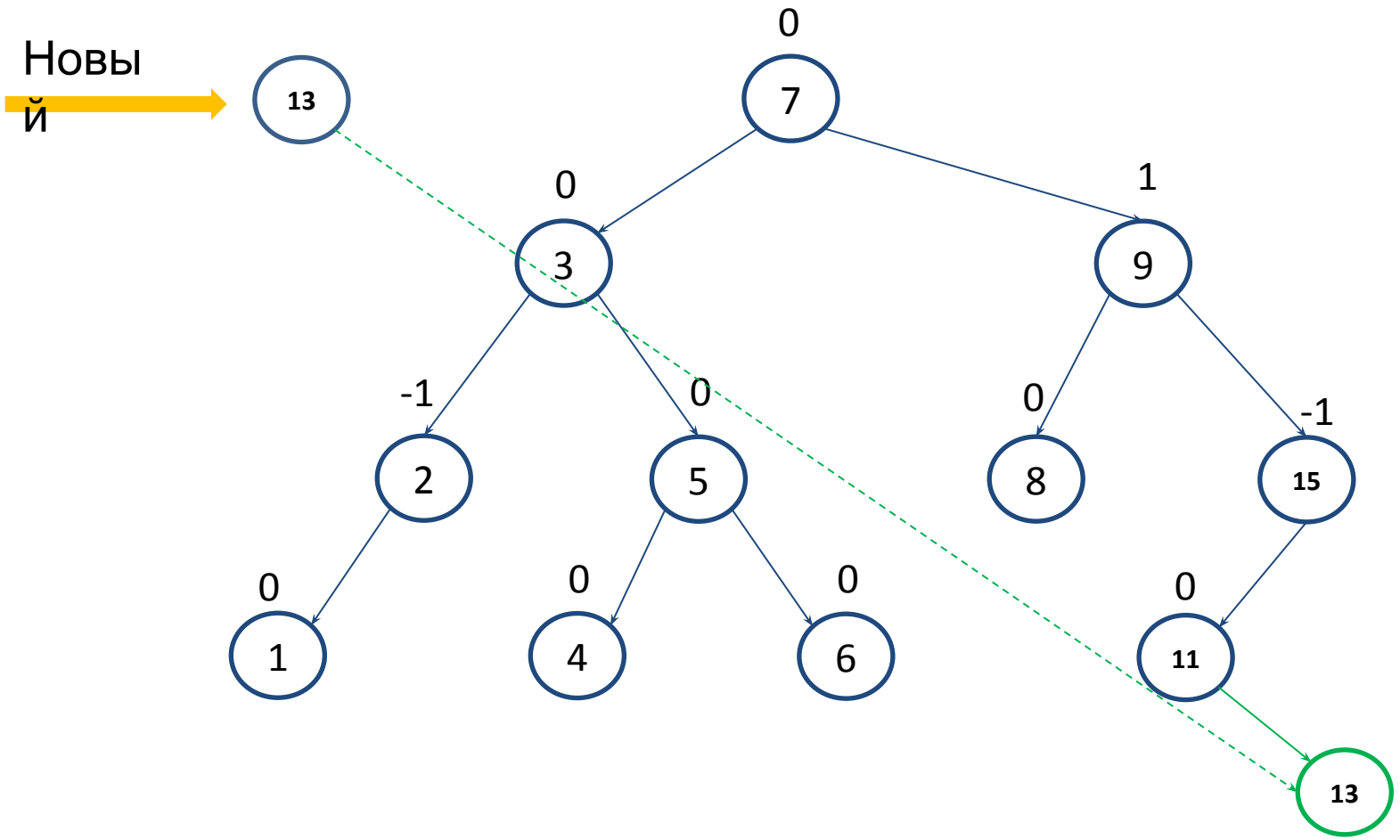
Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

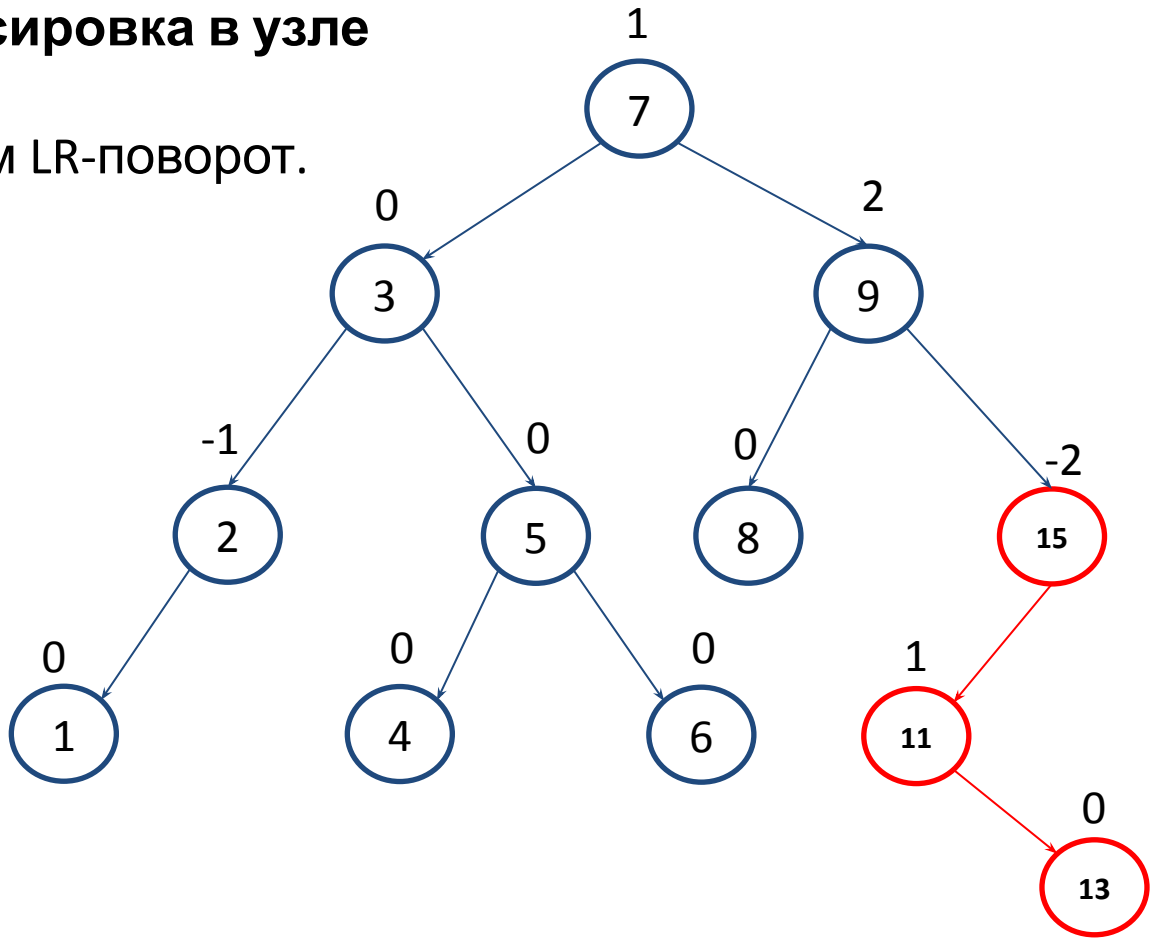
1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13



Построение АВЛ-дерева

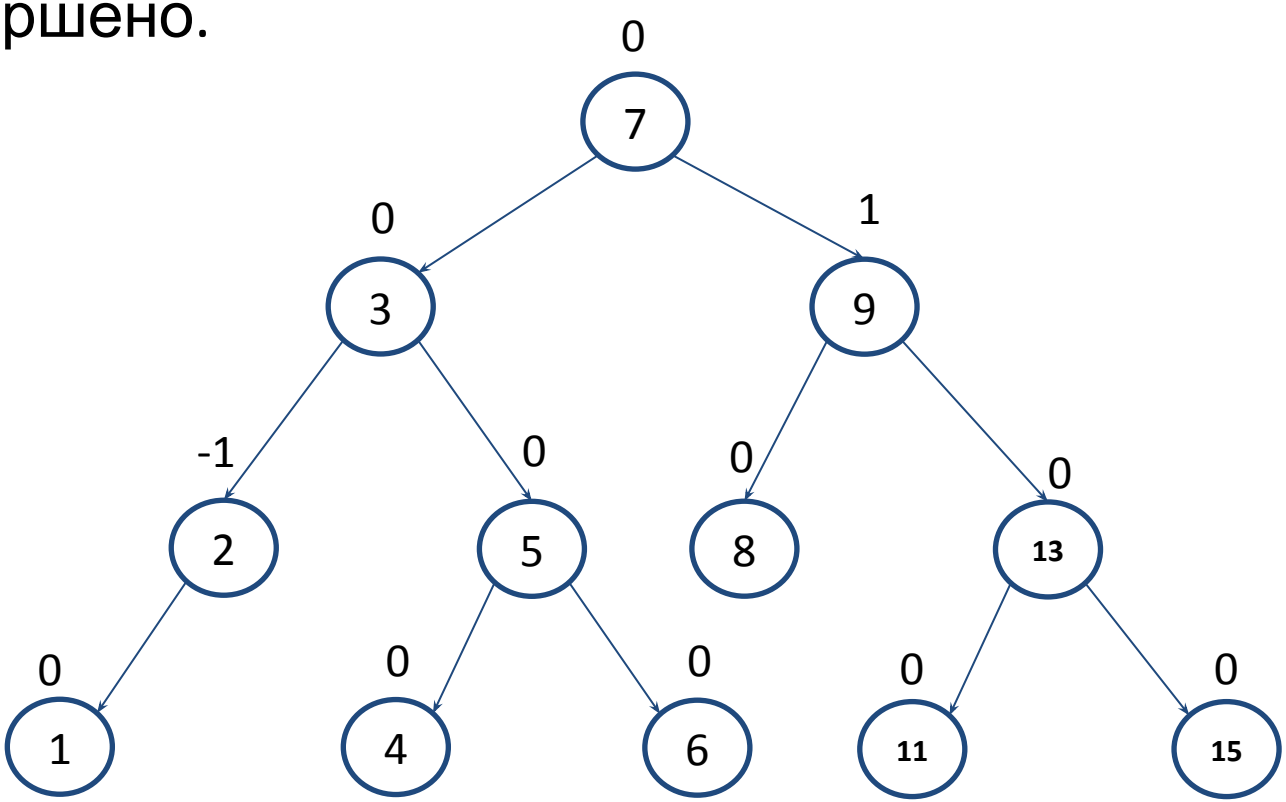
Разбалансировка в узле 15

Выполняем LR-поворот.



Построение АВЛ-дерева

1, 15, 7, 2, 3, 4, 9, 8, 5, 6, 11, 13. Построение завершено.

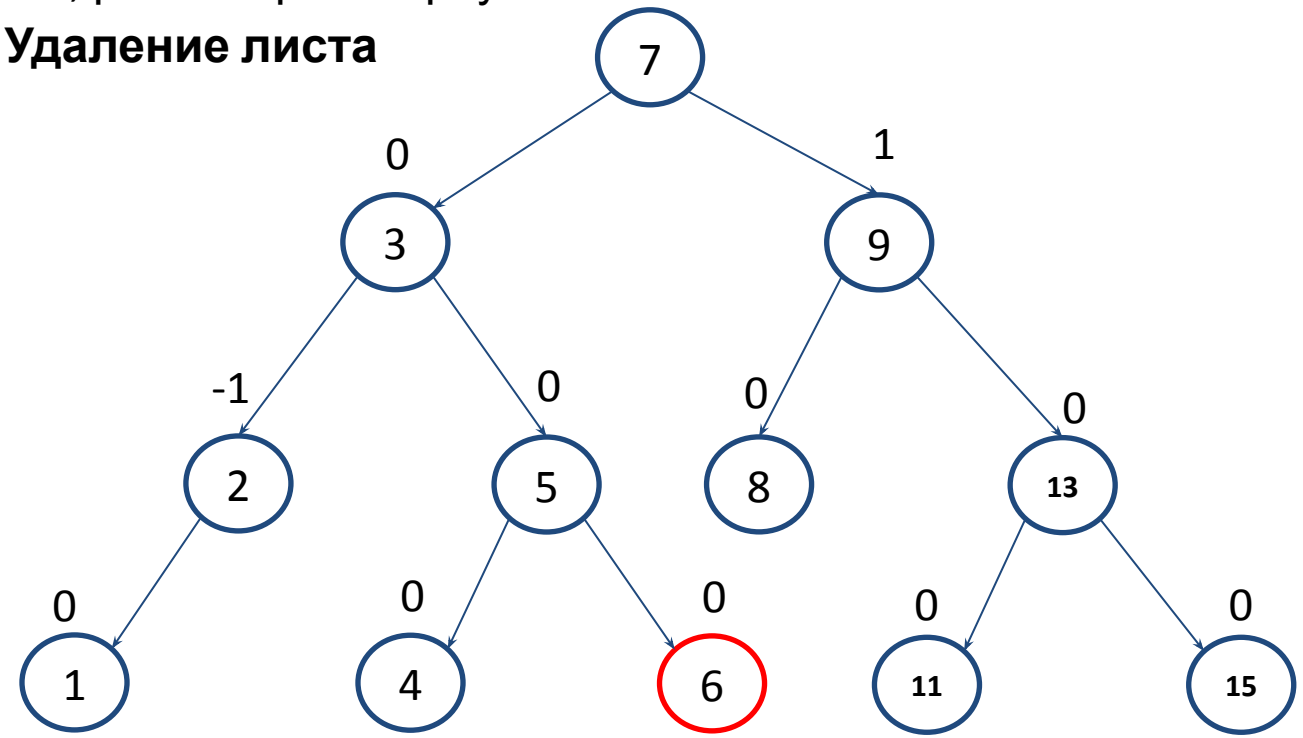


Удаление узла из АВЛ-дерева

Алгоритм удаления узла из АВЛ-дерева схож с алгоритмом удаления узла из обычного бинарного дерева поиска, с единственным отличием: в АВЛ-дереве, после удаления узла может потребоваться дополнительная балансировка.

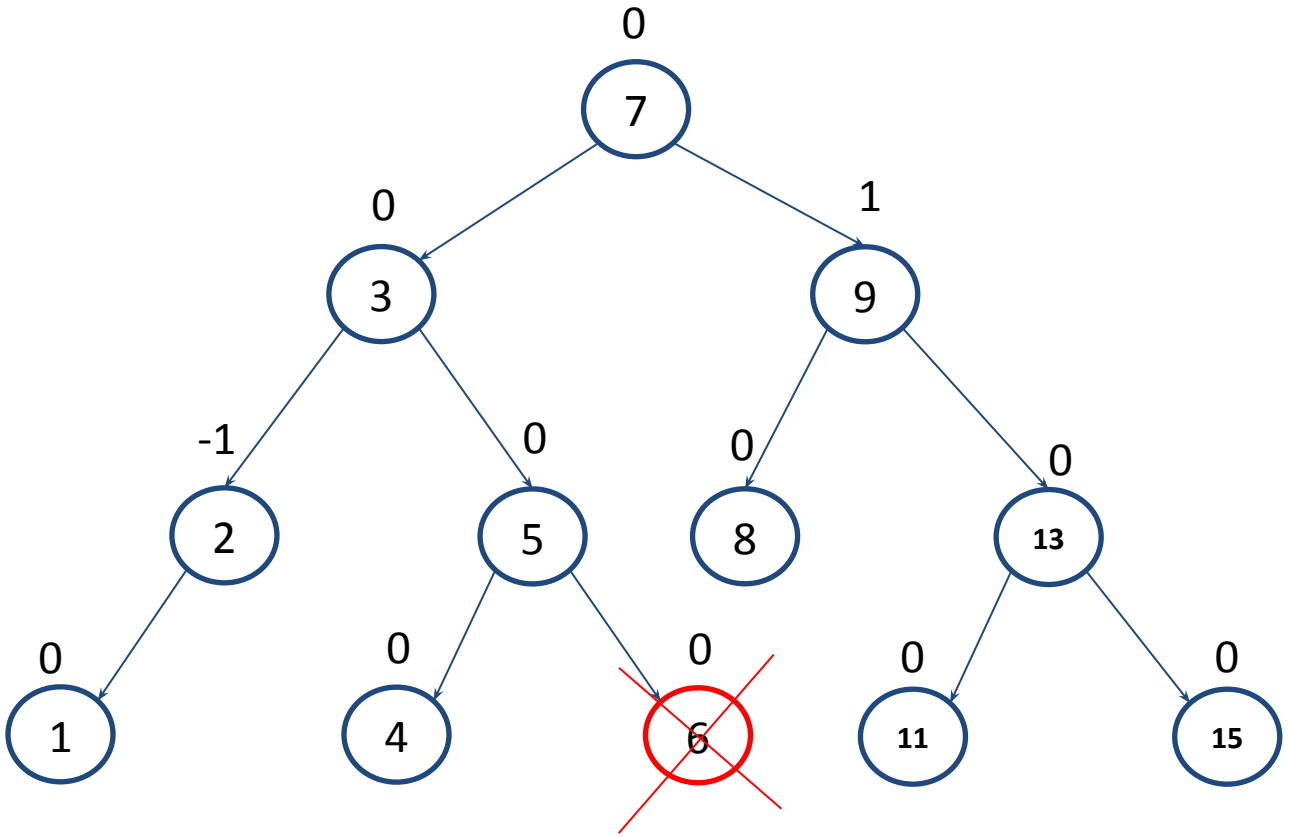
Существует 4 основных ситуации, возникающих при удаления узла из АВЛ – дерева, рассмотрим первую из них:

Случай 1: Удаление листа



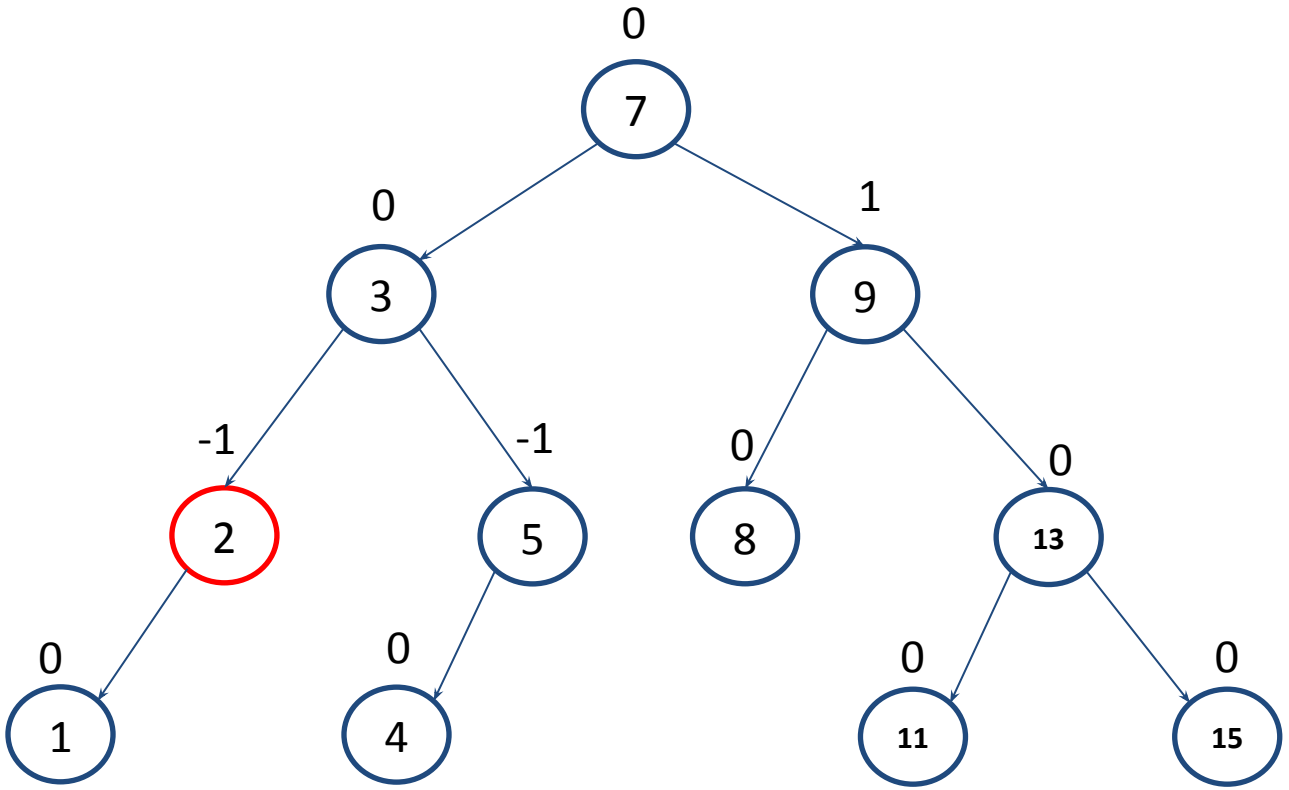
Удаление узла из АВЛ-дерева (случай 1)

Если узел не имеет потомков, т.е. является листом, то он просто удаляется, без каких-либо дополнительных операций. Удалим узел 6.



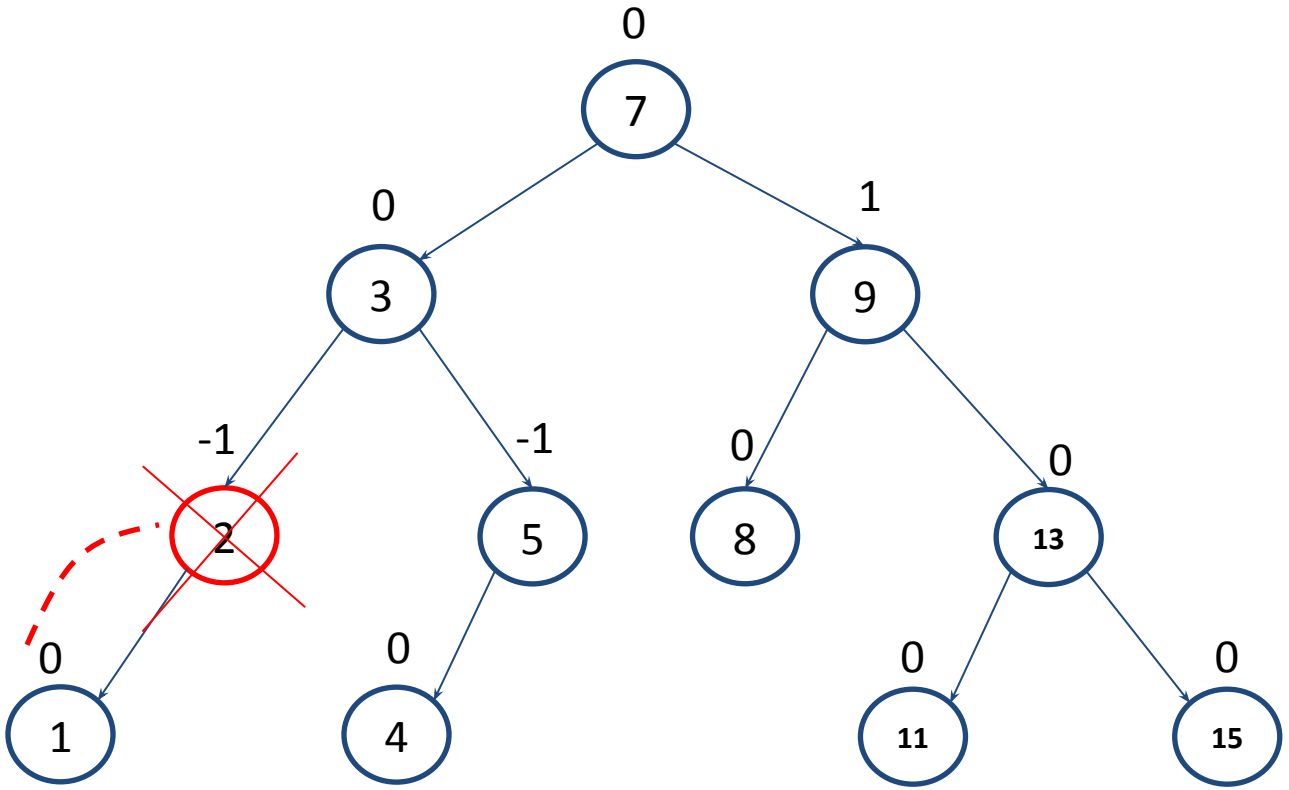
Удаление узла из АВЛ-дерева (случай 2)

Если удаляемый узел имеет только одно поддерево, то потомственный ему узел, включая все поддерево просто переносится (перевешивается) на место удаляемого узла. Удалим узел 2.



Удаление узла из АВЛ-дерева (случай 2)

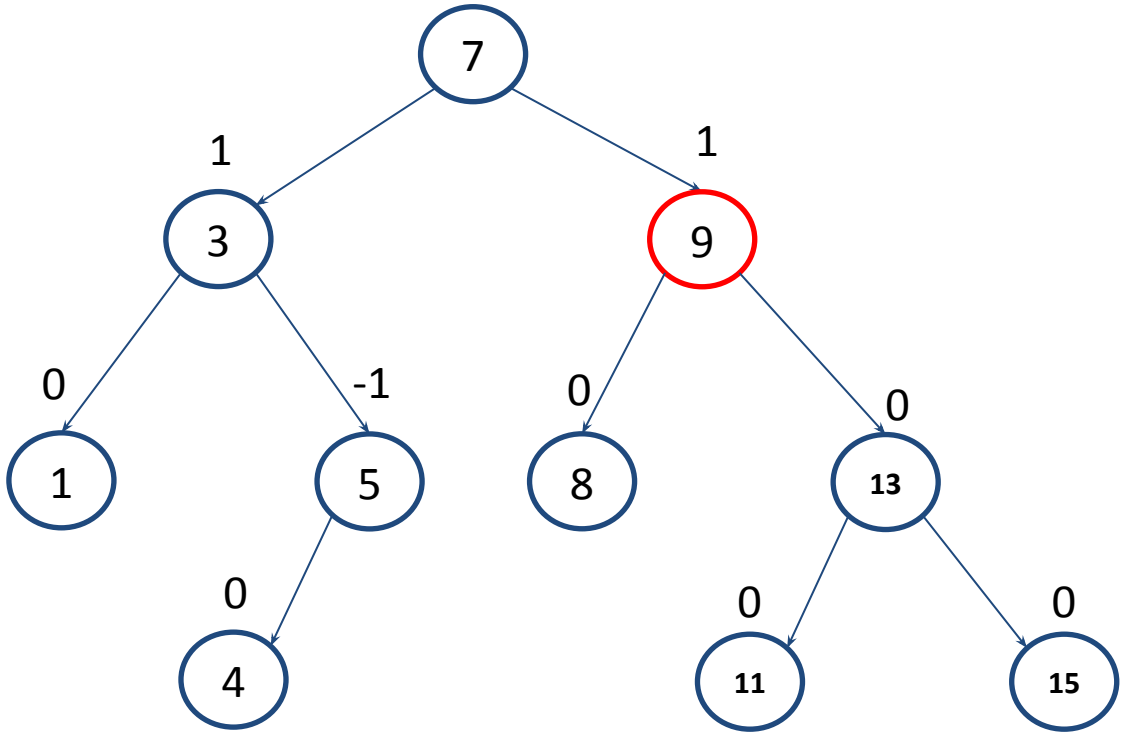
Удаляем узел 2 и переносим на его место узел 1.



Удаление узла из АВЛ-дерева (случай 3)

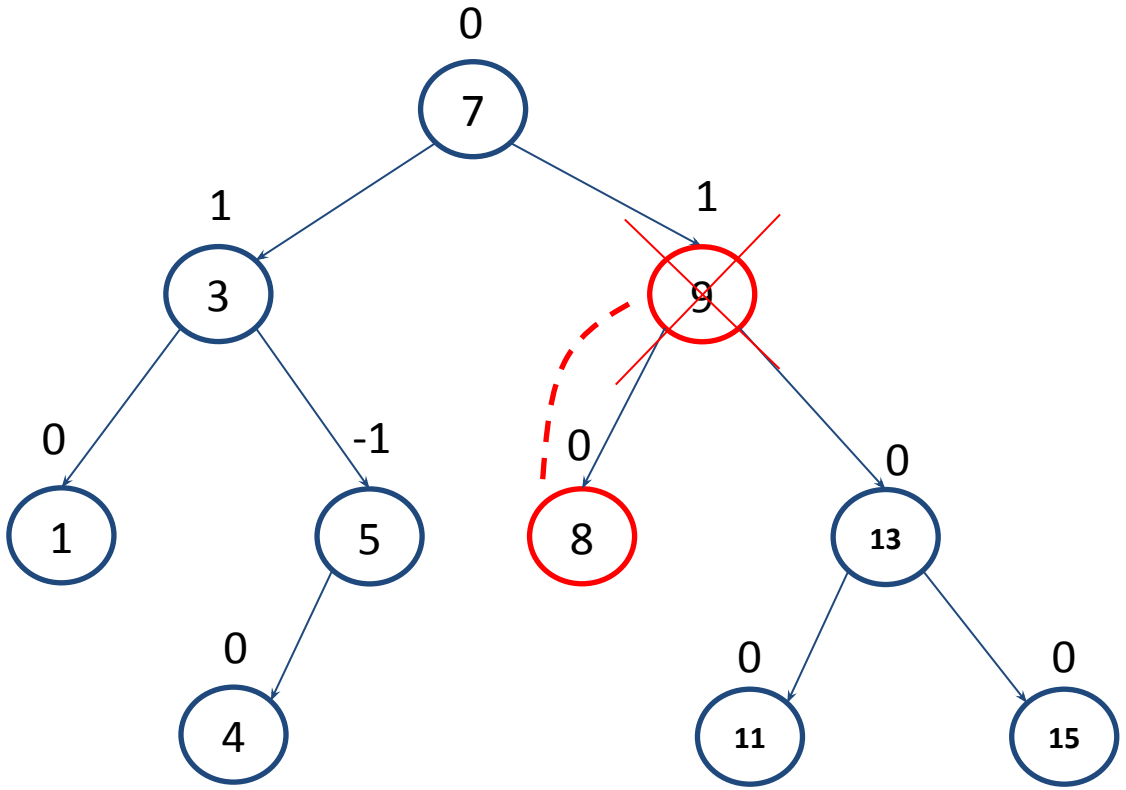
Если удаляемый узел имеет два поддерева, то в левом поддереве, удаляемого узла, ищется узел с наибольшим значением (самый правый узел). Если найденный узел является листом, то он просто переносится на место удаляемого узла. Удалим узел 9_0

.



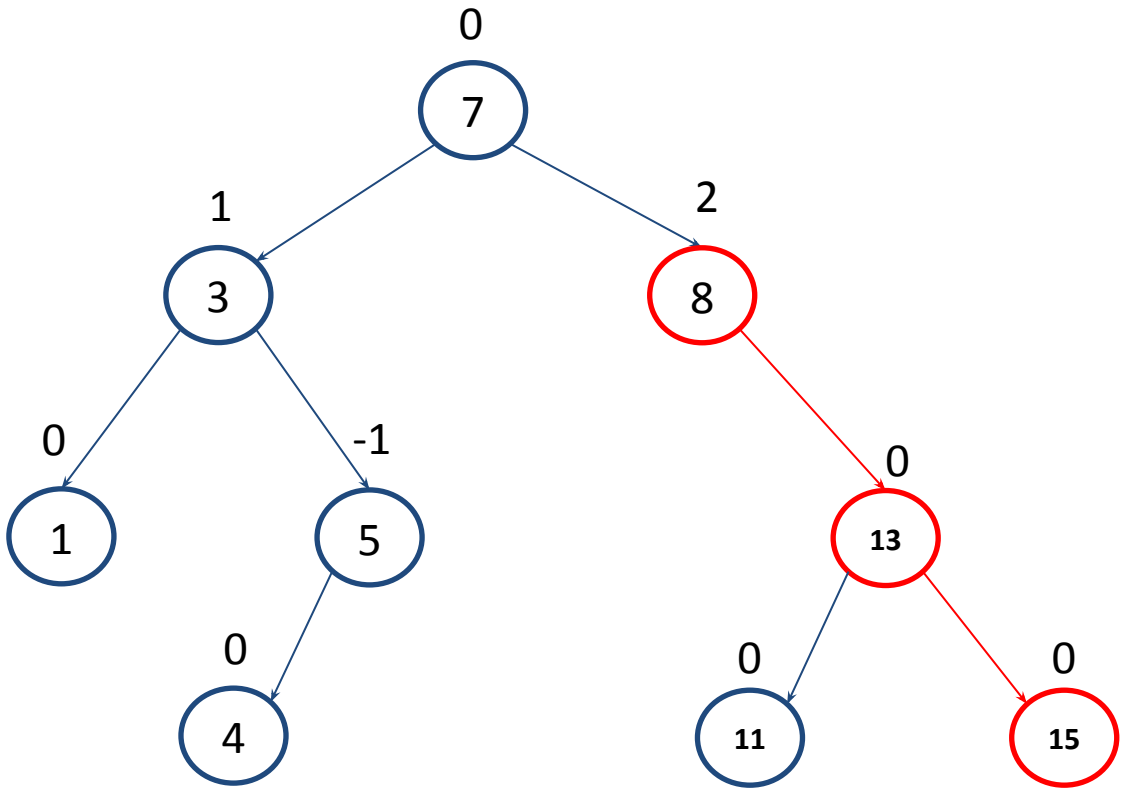
Удаление узла из АВЛ-дерева (случай 3)

Самый правый узел в левом поддереве узла 9 - это 8. Узел 8 является листом, поэтому просто переносим узел 8 на место удаляемого узла 9.



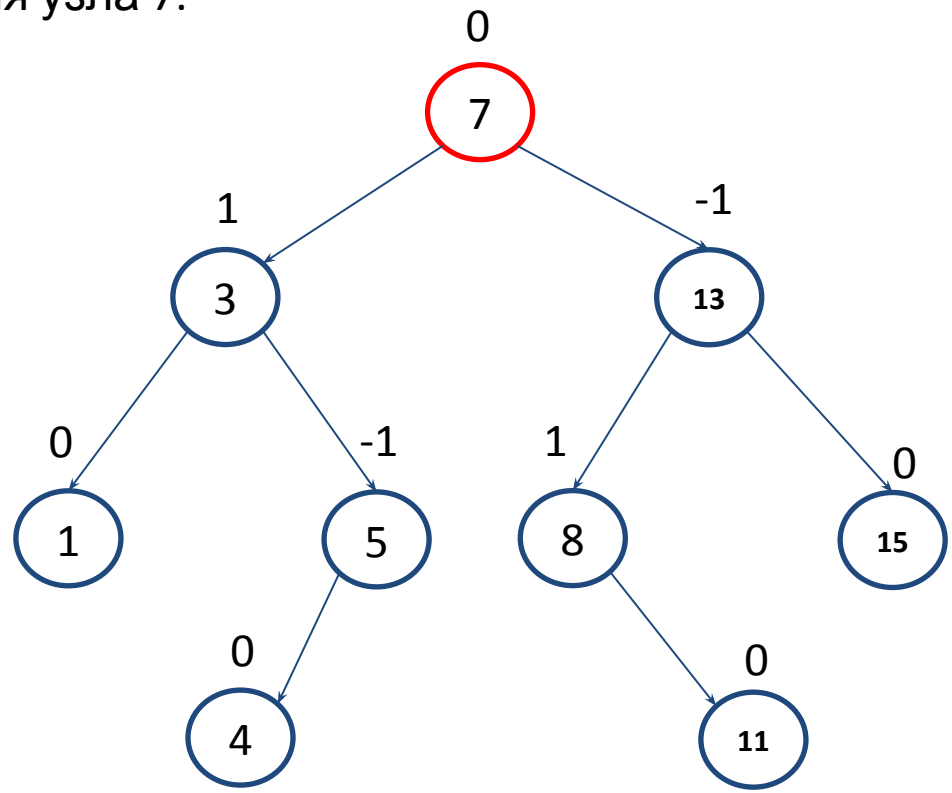
Удаление узла из АВЛ-дерева (случай 3)

Необходимо устранить разбалансировку в узле 8, выполнив RR-поворот.



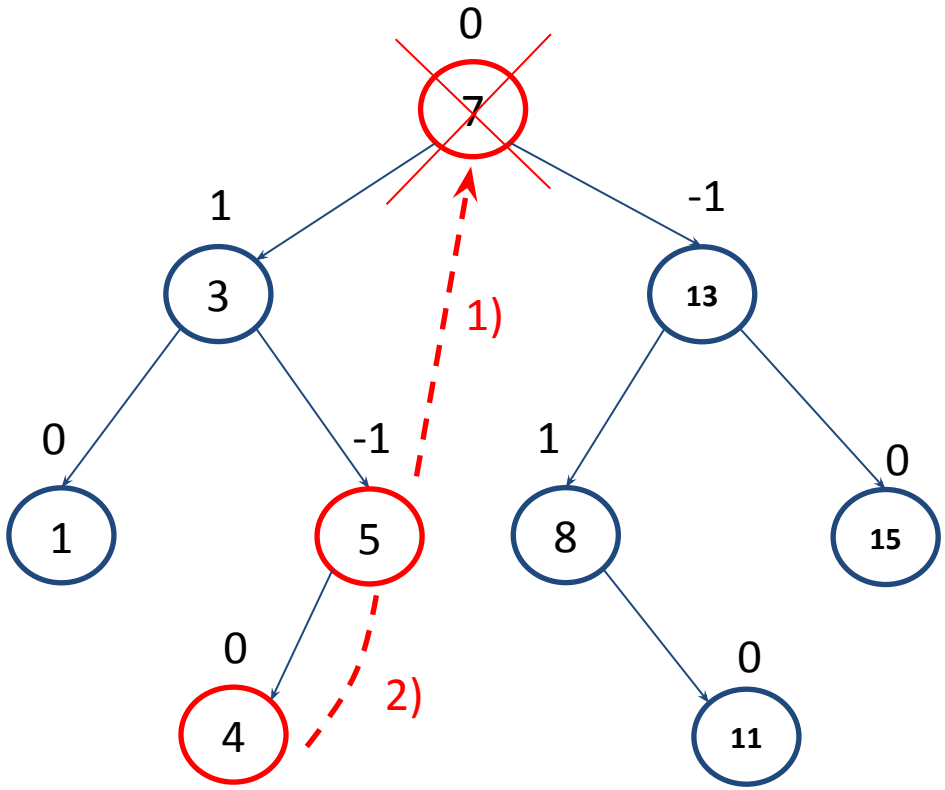
Удаление узла из АВЛ-дерева (случай 4)

Те же условия, что и в случае 3, только узел с наибольшим значением (самый правый узел) имеет левое поддерево (правое поддерево он не может иметь, т.к. он является самым правым). Рассмотрим этот случай на примере удаления узла 7.



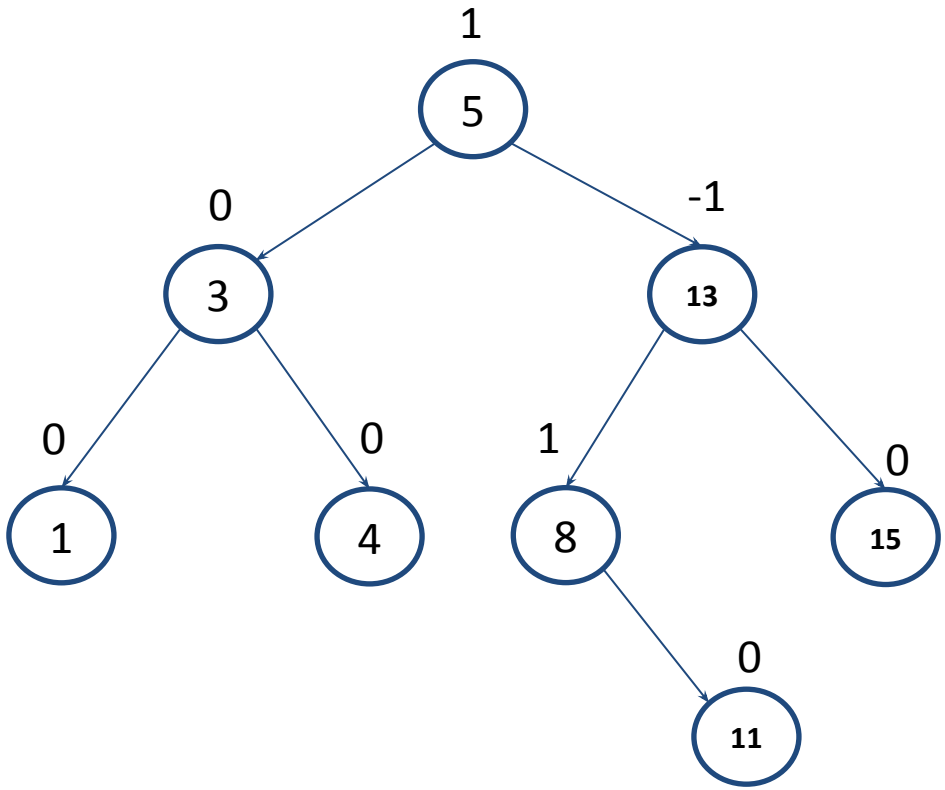
Удаление узла из АВЛ-дерева (случай 4)

1) Переместим найденный узел 5 на место удаляемого узла 7, как мы это делали в случае 3. 2) Переместим (перевесим) узел 4 (левое поддерево узла 5) на место узла 5, как мы это делали в случае 2.

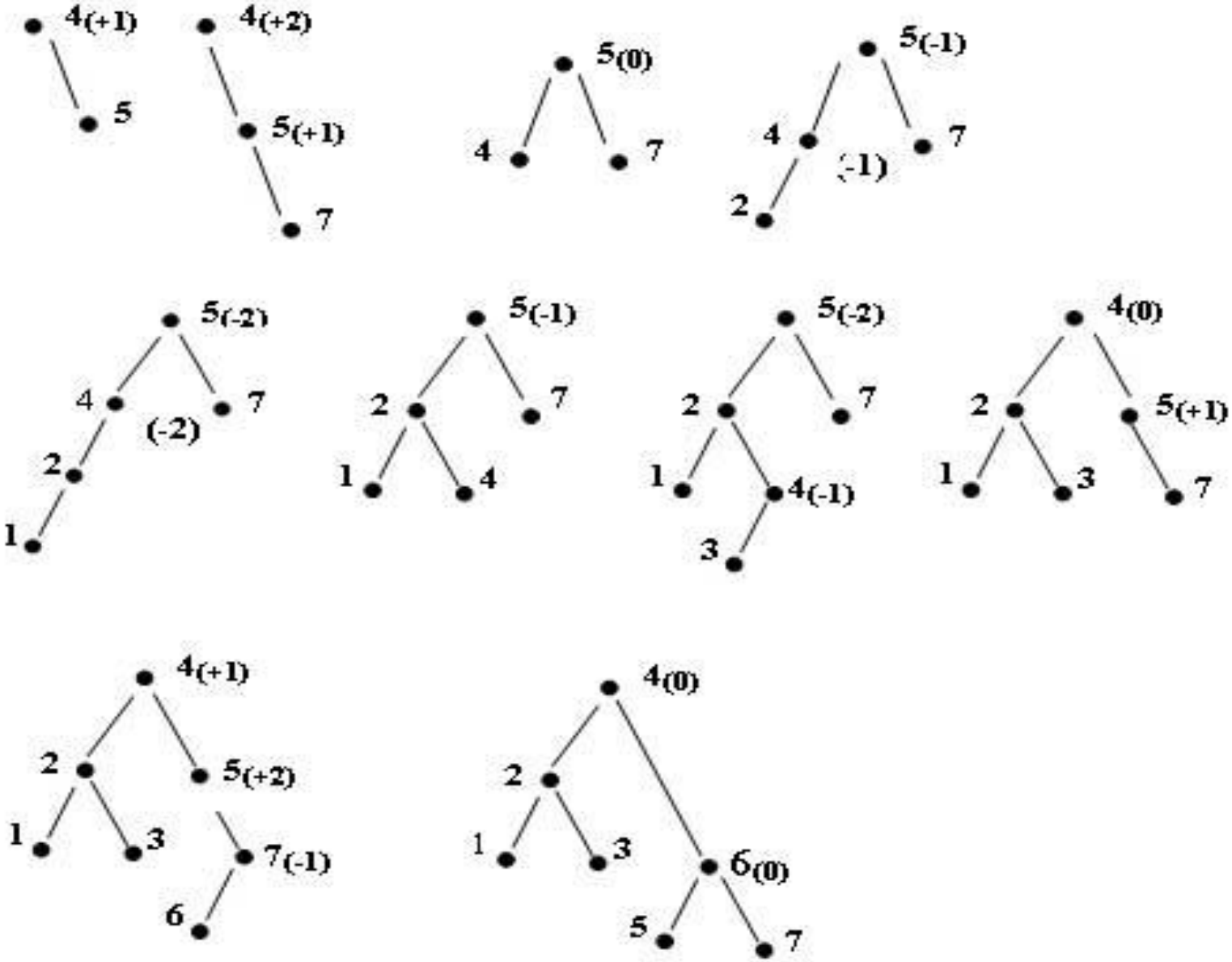


Удаление узла из АВЛ-дерева (случай 4)

Удаление завершено.



Пример построения АВЛ-дерева



Пример построения сбалансированного дерева двоичного поиска

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