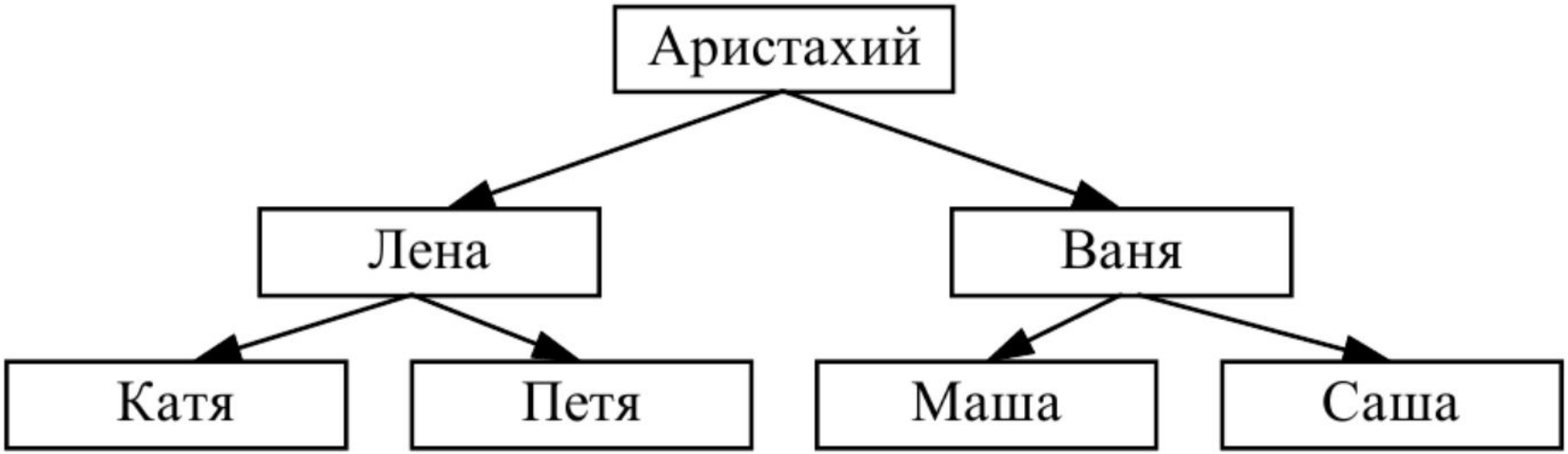
Алгоритмы и структуры данных

Лекция 6. Двоичные (бинарные) деревья поиска

Дерево – это совокупность узлов (вершин) и соединяющих их направленных ребер (дуг), причем в каждый узел (за исключением одного - корня) ведет ровно одна дуга.

Корень – это начальный узел дерева, в который не ведет ни одной дуги.

Примером может служить **генеалогическое дерево** - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.



Двоичные деревья

На практике используются главным образом деревья особого вида, называемые двоичными (бинарными).

Двоичным деревом называется дерево, каждый узел которого имеет не более двух сыновей.

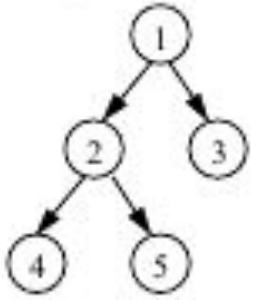
Можно определить двоичное дерево и рекурсивно:

1) пустая структура является двоичным деревом;
2) дерево – это корень и два связанных с ним двоичных дерева, которые называют левым и правым поддеревом .

Двоичные деревья **упорядочены**, то есть различают левое и правое поддеревья.

Строго двоичным деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Полным двоичным деревом называется дерево, у которого все листья находятся на одно уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.



Стандартные операции:
-поиск элемента по значению
-добавление элемента
-поиск максимального/минимального элемента
-поиск предыдущего/следующего по величине элемента
-удаление элемента
-различные варианты обхода дерева – для его вывода, записи в файл или удаления

Высота дерева поиска

Высота дерева определяется как длина самого длинного пути от корня до листа.

Обход дерева

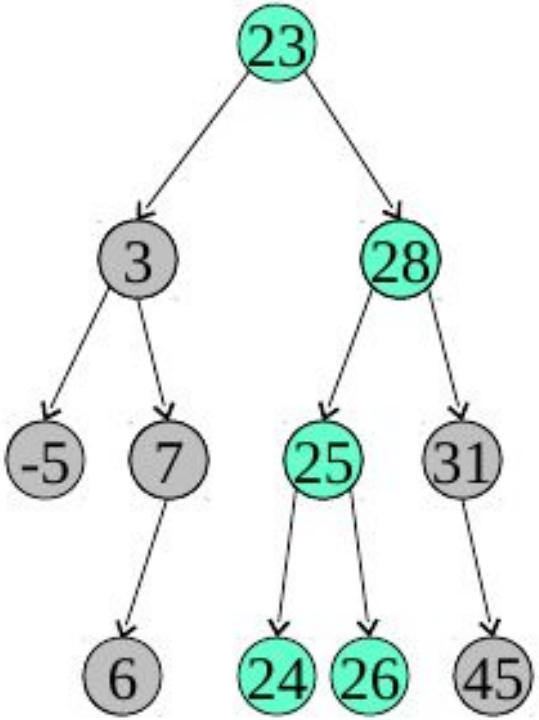
Одной из необходимых операций при работе с деревьями является обход дерева, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют **три варианта обхода**:

1) **КЛП** (корень – левое – правое): сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;
2) **ЛКП** (левое – корень – правое): сначала посещается левое поддерево, затем корень, а затем – правое;
3) **ЛПК** (левое – правое – корень): сначала посещается левое поддерево, затем правое, а затем – корень.

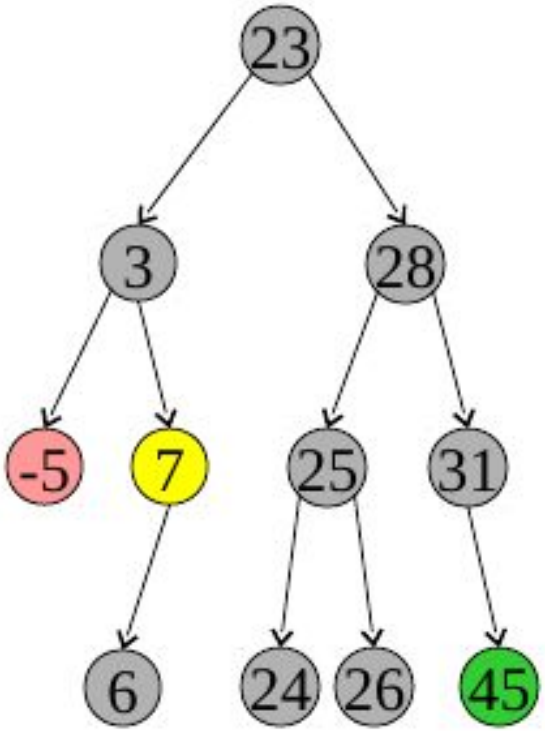
Пример последовательного добавления элементов

23, 28, 25, 3, 7, 31, 45, -5, 6, 24, 26
(один из возможных вариантов)



Поиск максимального или минимального значения

Начинаем с корня дерева. Спускаемся по дереву влево, пока это возможно – то есть, существует левый потомок узла. Последний рассмотренный узел и содержит минимум.

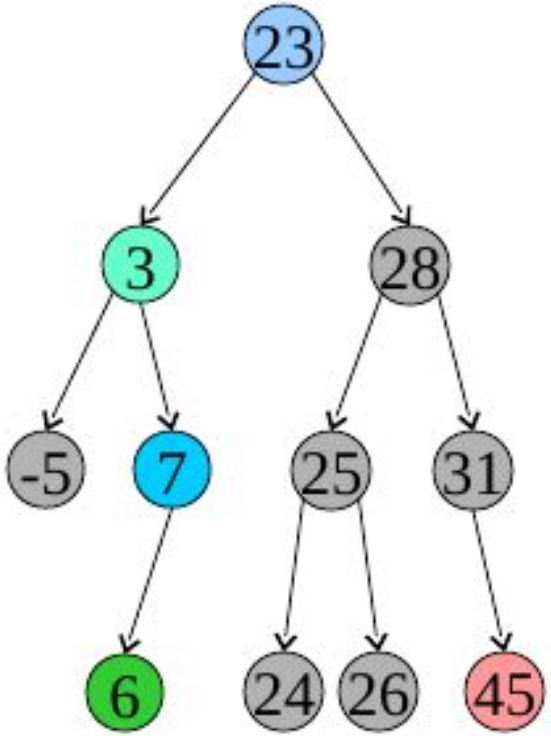


Для максимума тем же способом идём вправо.

Можно искать минимум или максимум не всего дерева, а одного из его поддеревьев.
Жёлтым отмечен максимум левого поддерева.

Поиск следующего элемента

Найдём элемент. Если у него есть **правый** потомок, найдём минимум в правом поддереве элемента (где этот потомок является корнем)



Так, у элемента с числом 3 на схеме есть правый потомок. Минимум в соответствующем поддереве – число 6.

Если правого потомка нет…

При поиске элемента будем запоминать последний элемент, который оказался больше искомого – на нём поиск пошёл в **левое** поддерево. Он будет следующим, если правого потомка *действительно* нет.

На схеме этой ситуации соответствуют элементы 7 и 23.

Поиск предыдущего производится аналогично.

Варианты простого удаления элемента

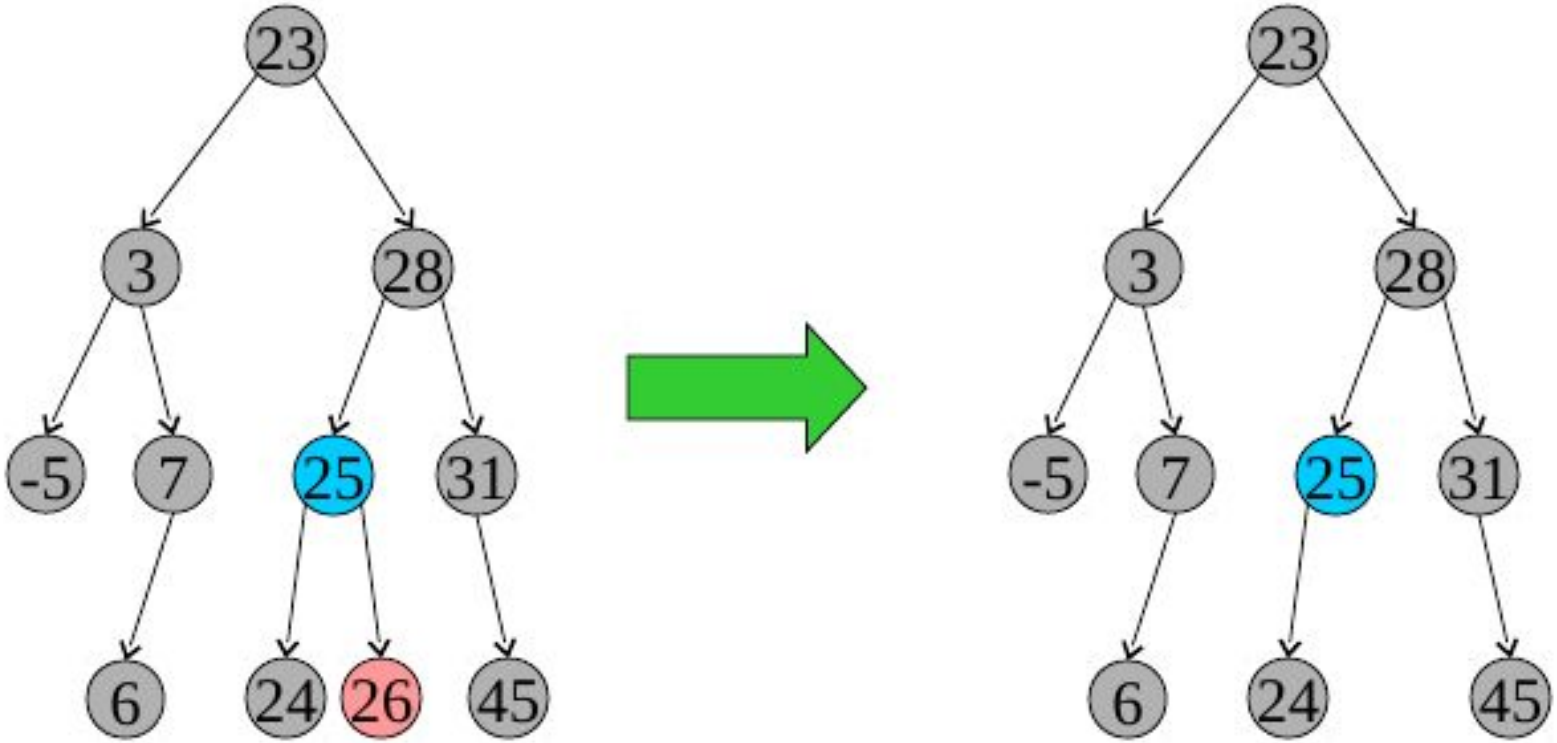
Случай 1. Удаляемый элемент является листом – то есть, не имеет потомков.
Решение: В переменную left или right элемента-отца, где был указатель на удаляемый, записать NULL.

Случай 2. У удаляемого элемента один потомок.
Решение: В переменную left или right элемента-отца, где был указатель на удаляемый, записать адрес потомка удаляемого.

Случай 3. У удаляемого элемента два потомка.
Решение: Найти и «отцепить» любой из элементов: содержащий предыдущее или следующее значение. Поставить его на место удаляемого.

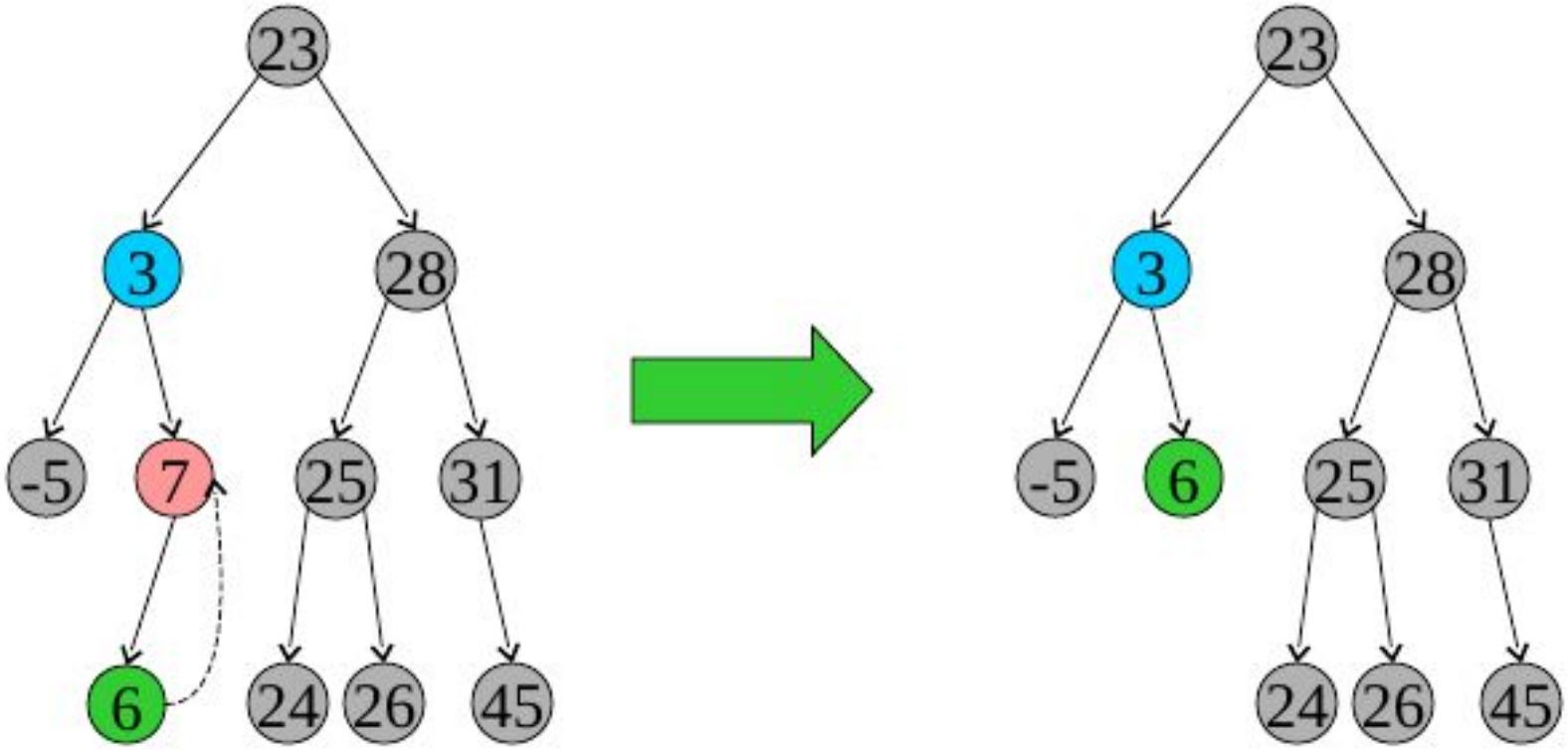
Сам найденный элемент стирается из памяти.

Случай 1. Удаляемый элемент является листом – то есть, не имеет потомков.



Удаление элемента 26.

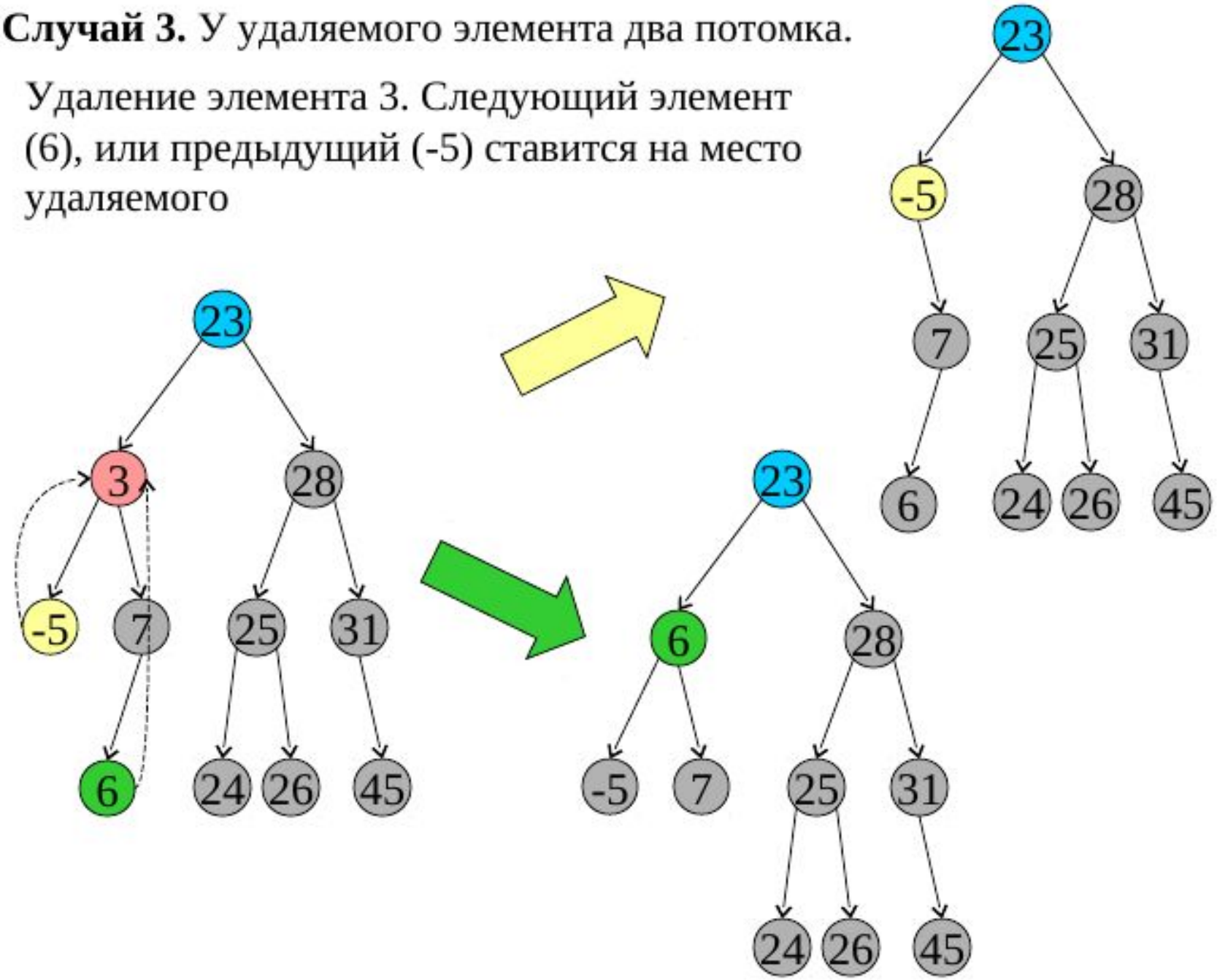
Случай 2. У удаляемого элемента один потомок.



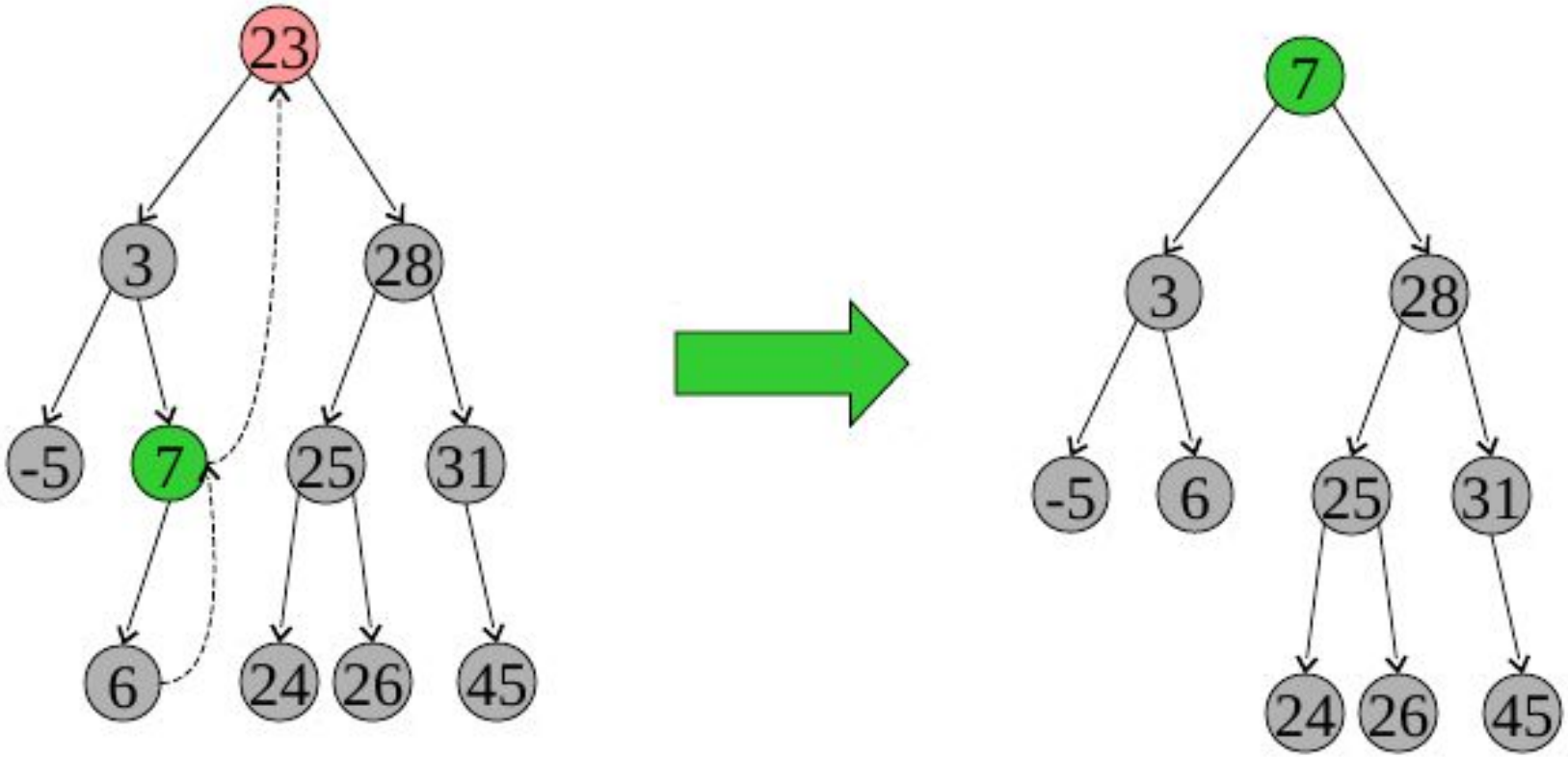
Удаление элемента 7. Потомок (6) ставится на место удаляемого

Случай 3. У удаляемого элемента два потомка.

Удаление элемента 3. Следующий элемент (6), или предыдущий (-5) ставится на место удаляемого

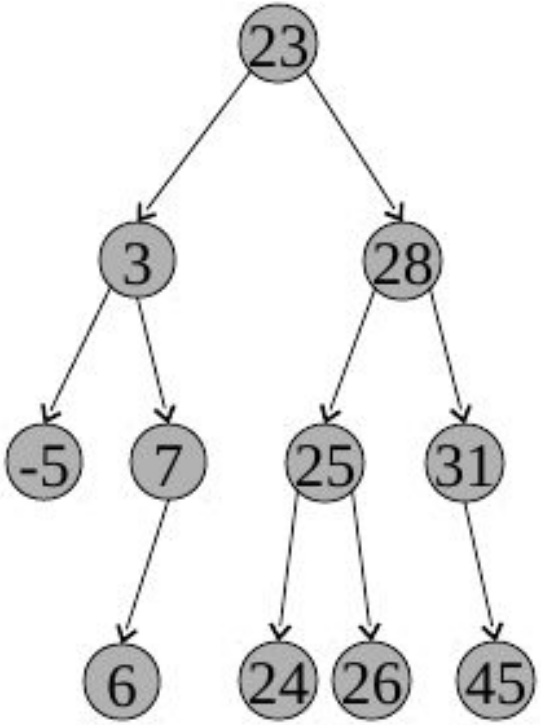


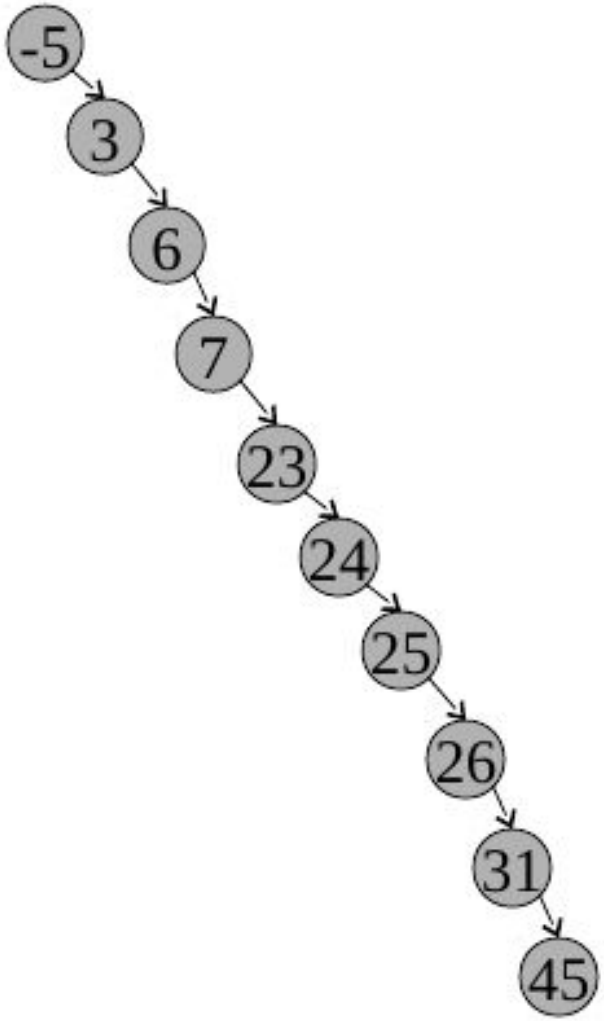
Случай 3. У удаляемого элемента два потомка. Удаляем корень.



Пример последовательного добавления элементов

Дерево с предыдущих слайдов можно получить, если добавлять элементы в таком порядке:
23, 28, 25, 3, 7, 31, 45, -5, 6, 24, 26
(один из возможных вариантов)



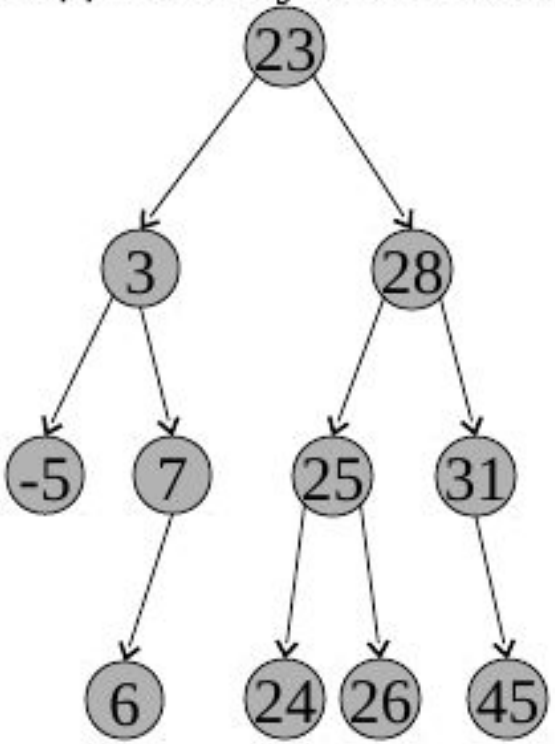


А если добавлять по возрастанию – получим вырожденное дерево

Сбалансированное дерево

Простое добавление-удаление не подходит для стабильной быстрой работы с данными, хранящимися в дереве: иногда дерево оказывается вырожденным.

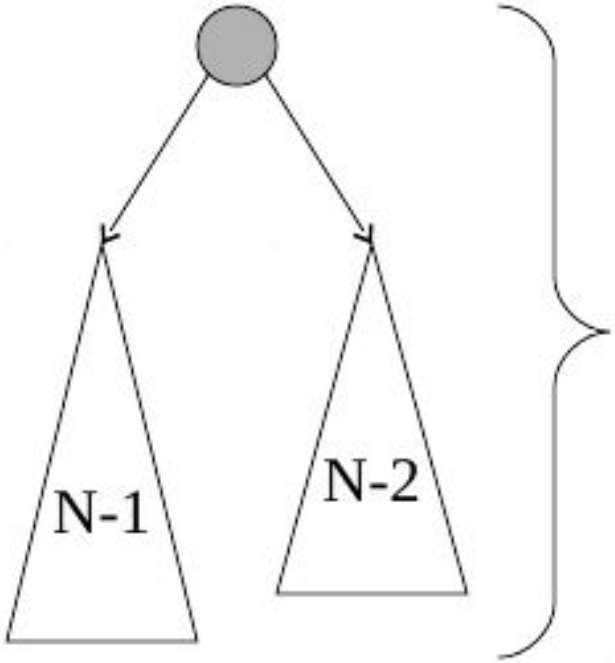
У сбалансированного (*по высоте*) дерева высоты поддеревьев **каждого** из узлов отличаются не более чем на единицу.



Дерево из предыдущих примеров является сбалансированным.

Существует несколько вариантов наборов алгоритмов добавления-удаления, при которых поддерживается сбалансированность.

Свойства: количество элементов



Сбалансированное дерево с N уровнями – это корень и два сбалансированных поддерева; одно с N-1 уровнями, второе – как минимум с N-2.

Минимальное количество элементов такого дерева составит
$k(N)=1+k(N-1)+k(N-2)$

Поскольку $k(1)=1$, $k(2)=2$, получаем следующие минимальные количества:

N	3	4	5	6	7	8	9	10
k(N)	4	7	12	20	33	54	88	143

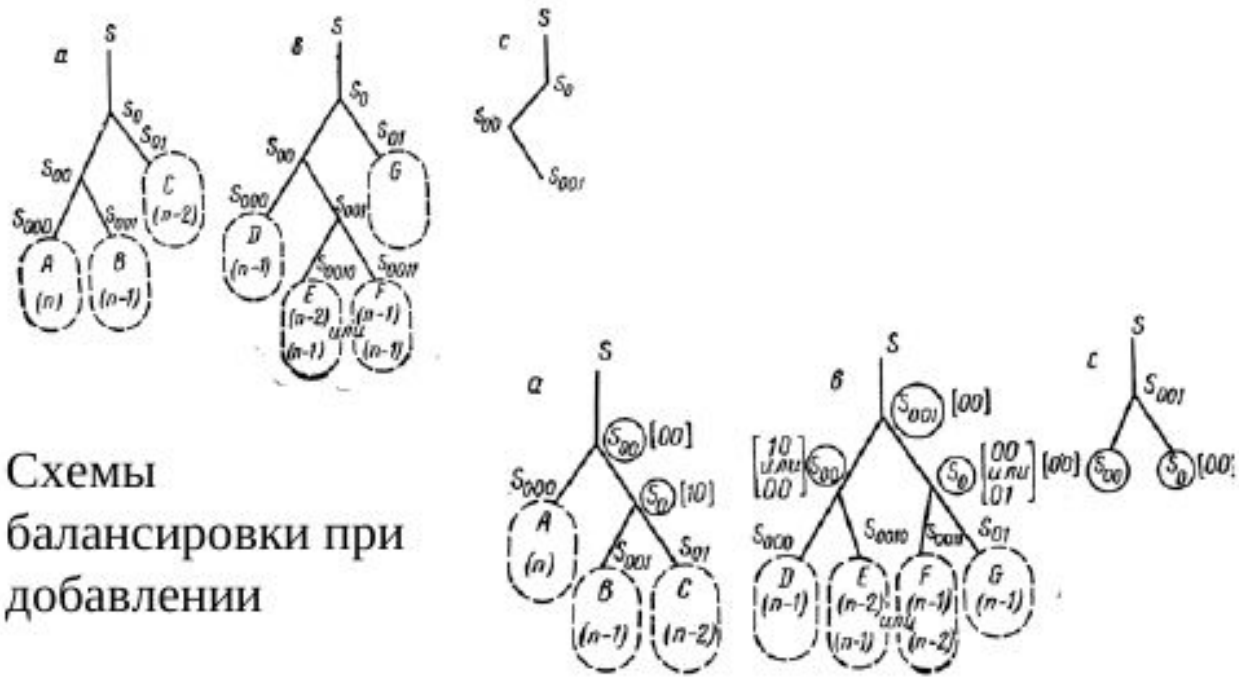
Для сравнения, минимальное число элементов полного дерева равно 2^N

АВЛ-дерево

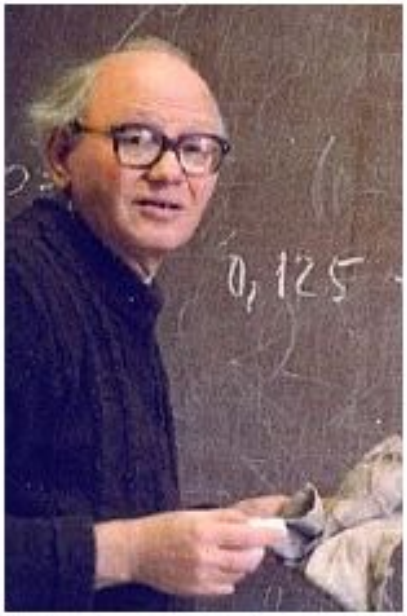
Один из вариантов сбалансированного (*по высоте*) дерева, назван по фамилиям авторов

1962, «Один алгоритм организации информации»

В оригинальной статье были описаны алгоритмы добавления, но не алгоритмы удаления.



Схемы балансировки при добавлении



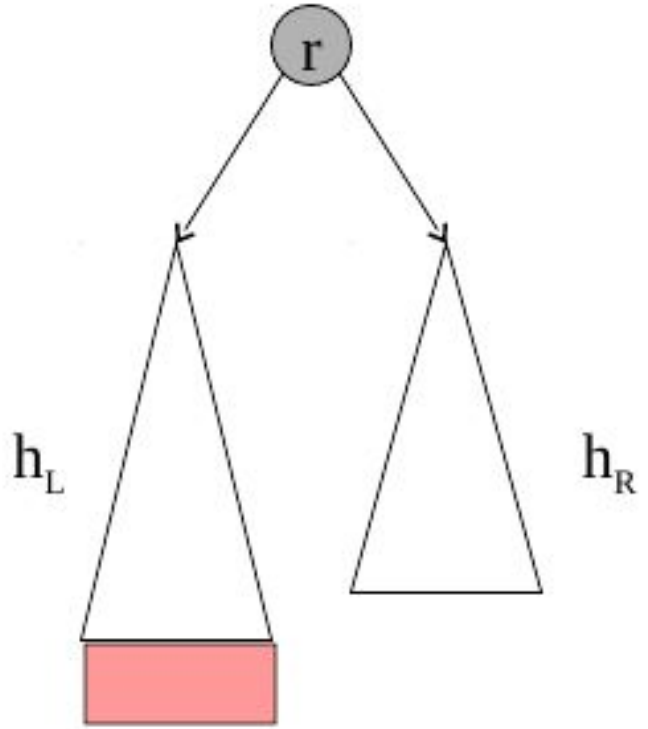
Г. М.Адельсон-Вельский

Е. М. Ландис

Добавление элемента

На схеме, r – корень (root), h_L – высота левого поддерева, h_R – правого.

Пусть, элемент был добавлен в левое поддерево и увеличил его высоту.

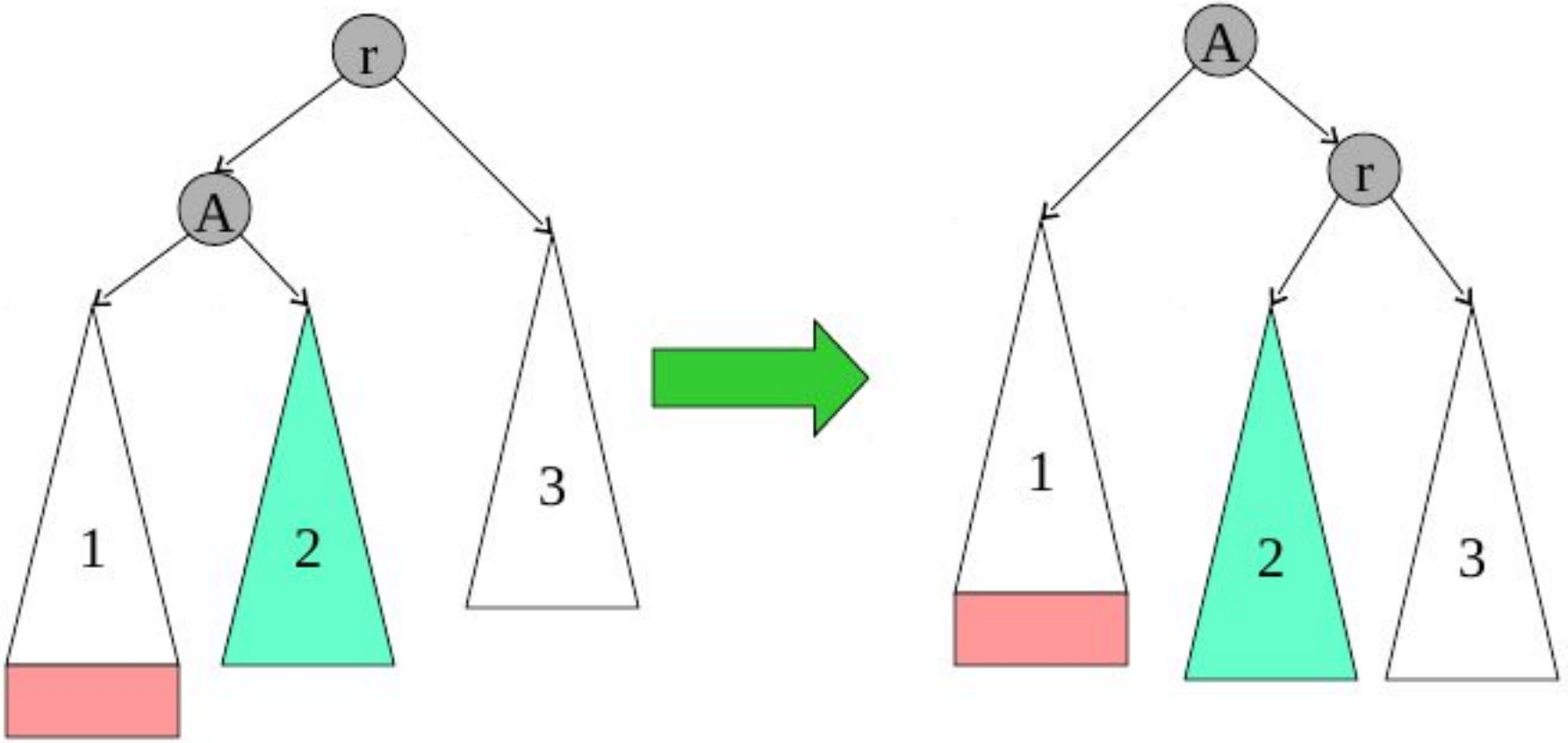


Возможны три случая:

1. $h_L < h_R$ – после вставки поддеревья выровнялись
2. $h_L = h_R$ – дерево всё ещё сбалансированное
3. $h_L > h_R$ – (на схеме) после вставки левое поддерево на 2 выше. Требуется балансировка.

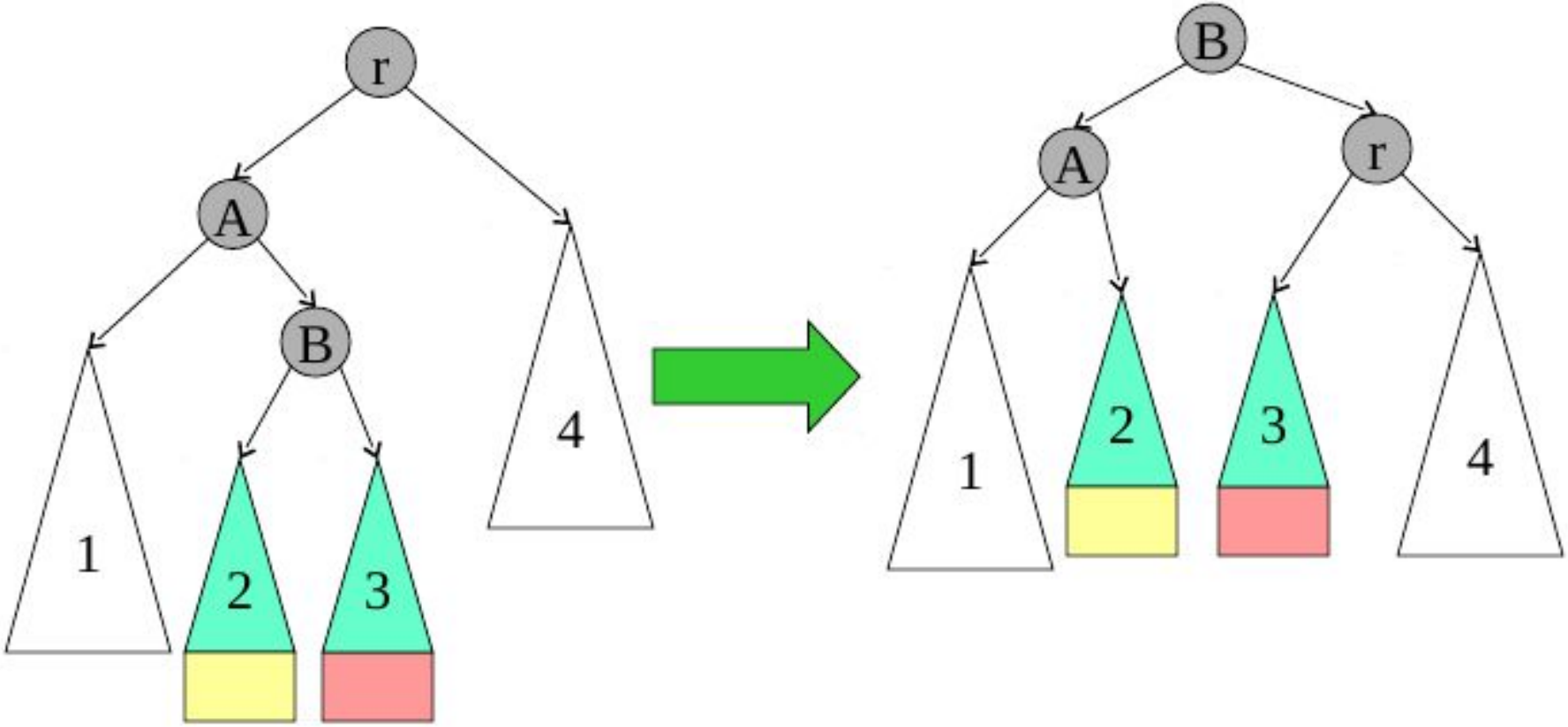
Балансировка 1 – «малое вращение»

Элемент был добавлен в левое поддерево левого поддерева.
Корень левого поддерева (A) становится корнем дерева

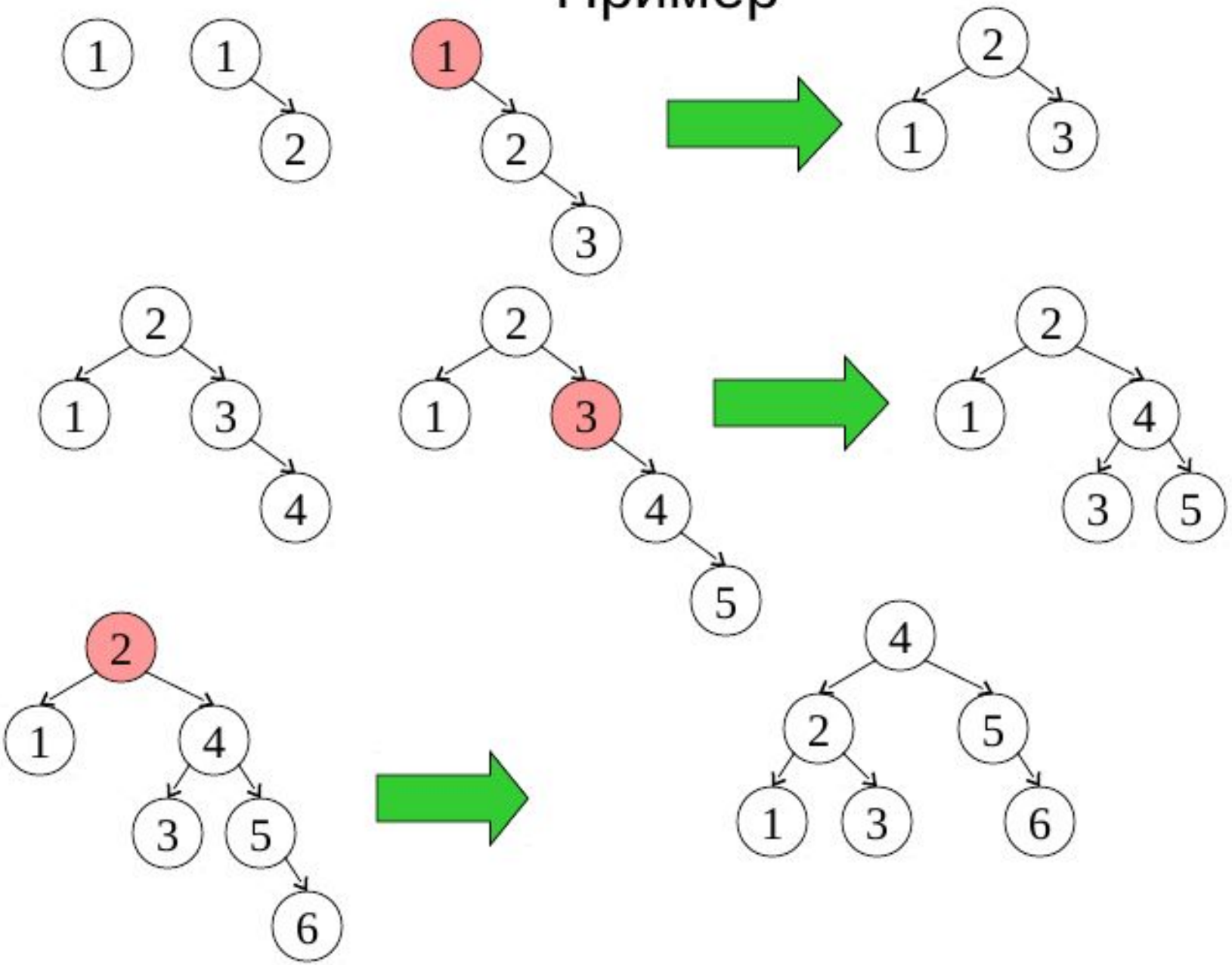


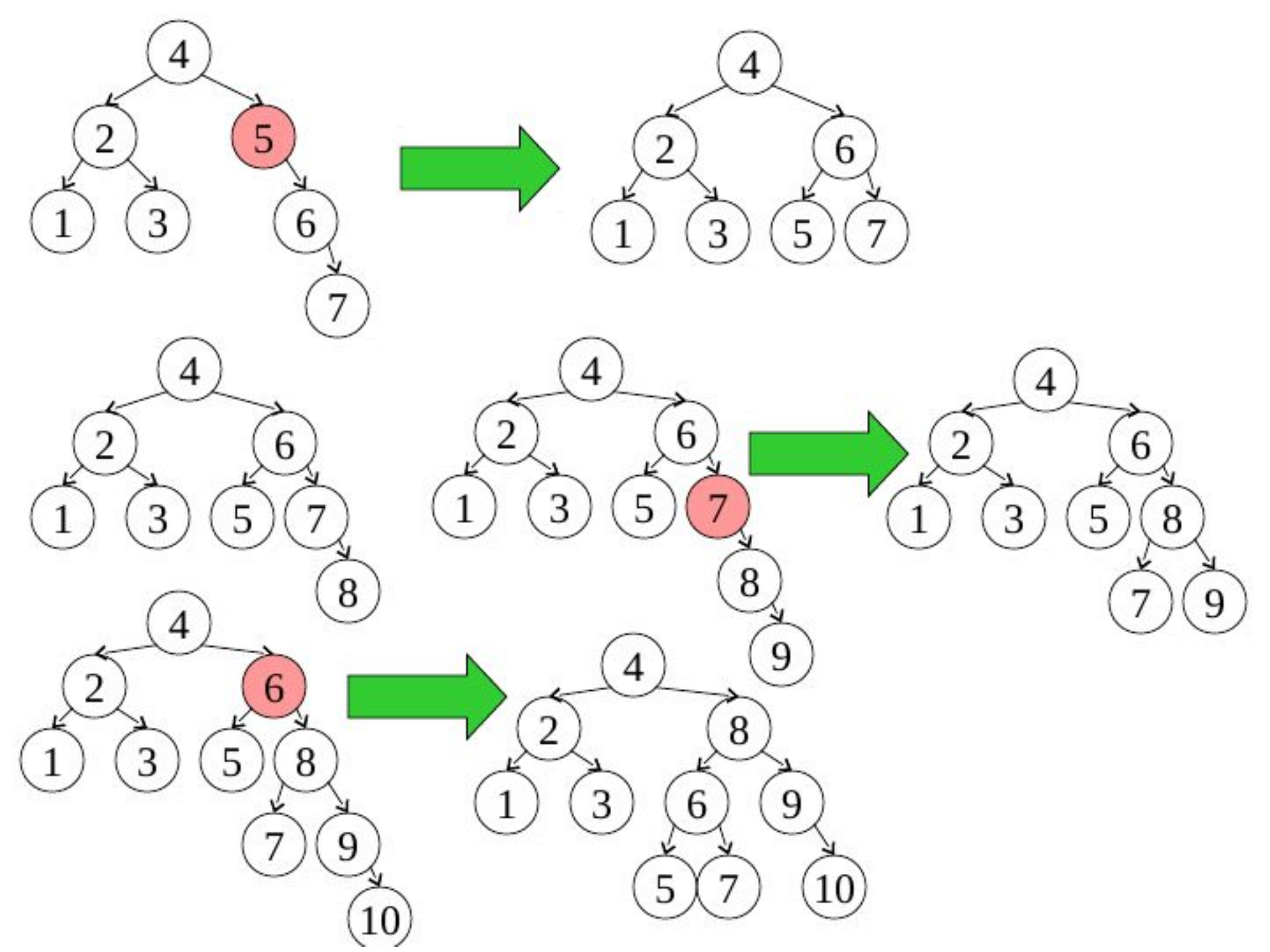
Балансировка 2 – «большое вращение»

Элемент был добавлен в правое поддерево левого поддерева. Корень правого поддерева левого поддерева (B) становится корнем дерева



Пример




4
2
5
1
3
6
7
4
2
6
1
3
5
7
4
2
6
1
3
5
7
8
4
2
6
1
3
5
7
8
9
4
2
6
1
3
5
8
7
9
4
2
6
1
3
5
8
7
9
10
4
2
8
1
3
6
9
5
7
10

АВЛ – дерево, комментарии

Балансировка при добавлении в правое поддерево делается симметрично.

Если расход памяти важен, для хранения состояния узла хватит 2 бит: поддеревья равны, левое выше, правое выше. Функция добавления в этом случае возвращает 1, если высота увеличилась и 0 иначе.
Можно хранить высоты явно. В любом случае нет смысла использовать тип данных больше чем `char` для их хранения – высота всего дерева невелика даже при большом объёме данных.

Для удаления не придумано ничего лучше, чем попробовать «обратить» балансировку при добавлении. Потенциально это может привести к перестроению дерева на каждом уровне.

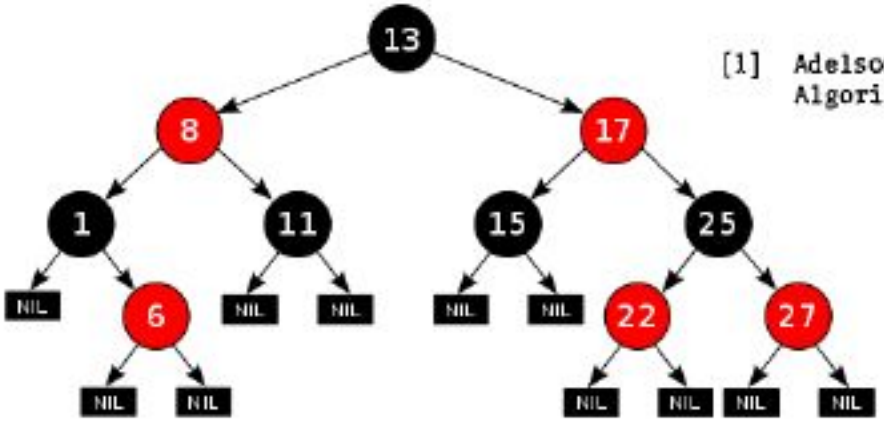
Красно-чёрное дерево

1972, «Symmetric Binary B-Trees. Data Structure and Maintenance Algorithms»
Красно-чёрными деревьями эти сбалансированные деревья назвал Роберт Седжвик (Robert Sedgewick) в 1978 году, стараясь более наглядно представить алгоритмы.

В работе приведён полный набор алгоритмов – как для добавления, так и для удаления. Их много!

Первая ссылка в статье Байера – на работу Адельсона-Вельского и Ландиса

Rudolf Bayer



BIBLIOGRAPHY

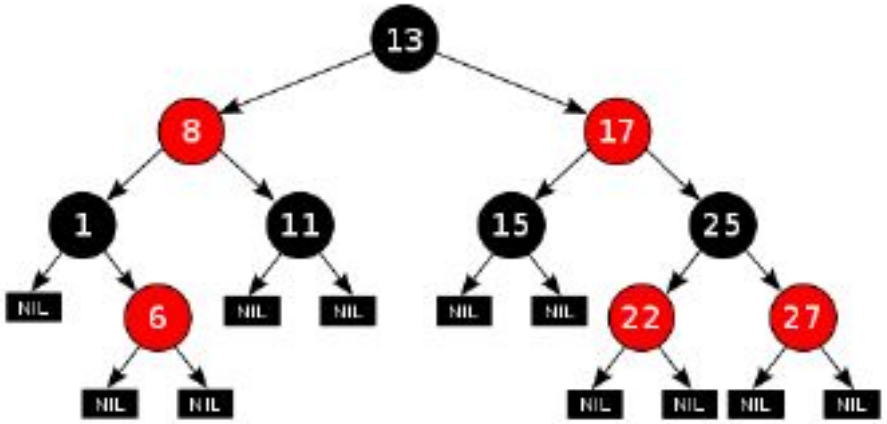
[1] Adelson-Velskii, G.M. and Landis, E.M., An Information Organization Algorithm, DANSSR, No. 2, 1962.

Красно-чёрное дерево

Правила

1. Каждый узел либо красный, либо чёрный
2. Корень – чёрный.
3. Нулевые указатели считаются листьями, причём листья – чёрные.
4. Потомки красного узла – чёрные. (Потомки чёрного узла – не обязательно красные).
5. Любой путь от корня поддерева до листа содержит одинаковое число чёрных узлов. (Глубина по чёрным узлам).

Грубая оценка на основании правил 4 и 5 показывает, что длины двух соседних поддеревьев отличаются не более, чем в 2 раза.



Каждый новый узел изначально считается красным. Если это нарушает одно из правил, обычно 4 или 5, производится балансировка…

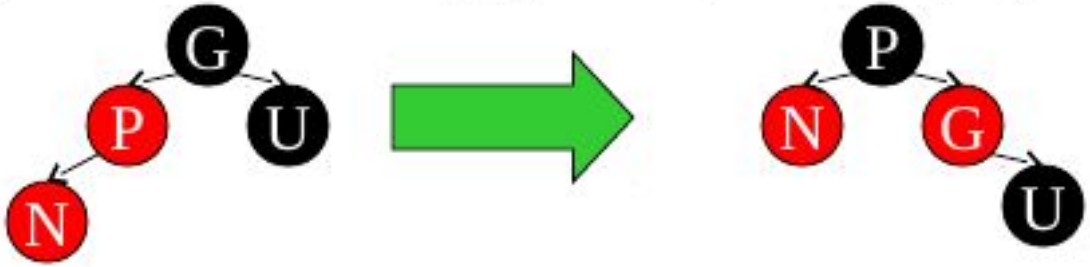
Балансировки при добавлении

Работают рекурсивно. Если текущий узел – красный, он может создать проблемы.

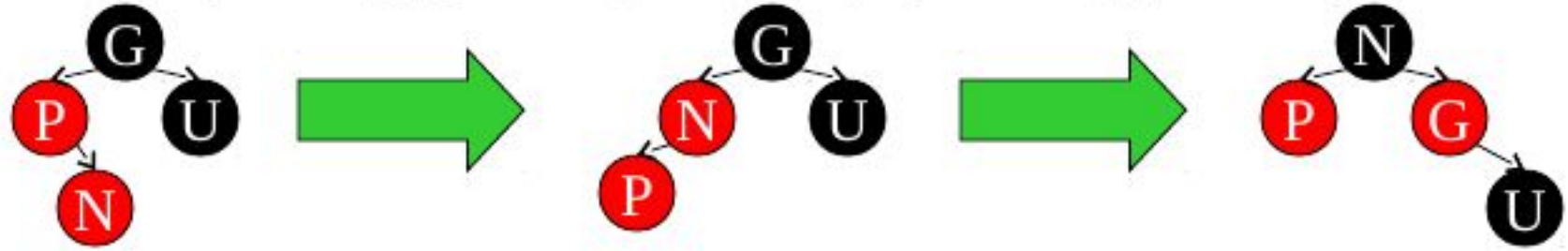
1. Если это корень всего дерева – меняем его цвет на чёрный.
2. Если предок – чёрный, всё сбалансировано.
3. Если «отец» и «дядя» - красные

После проверить, не нарушает ли «дедушка» G одно из правил

4. Добавление в левое поддерево «отца» P. «Отец»-красный, «дядя» U - чёрный



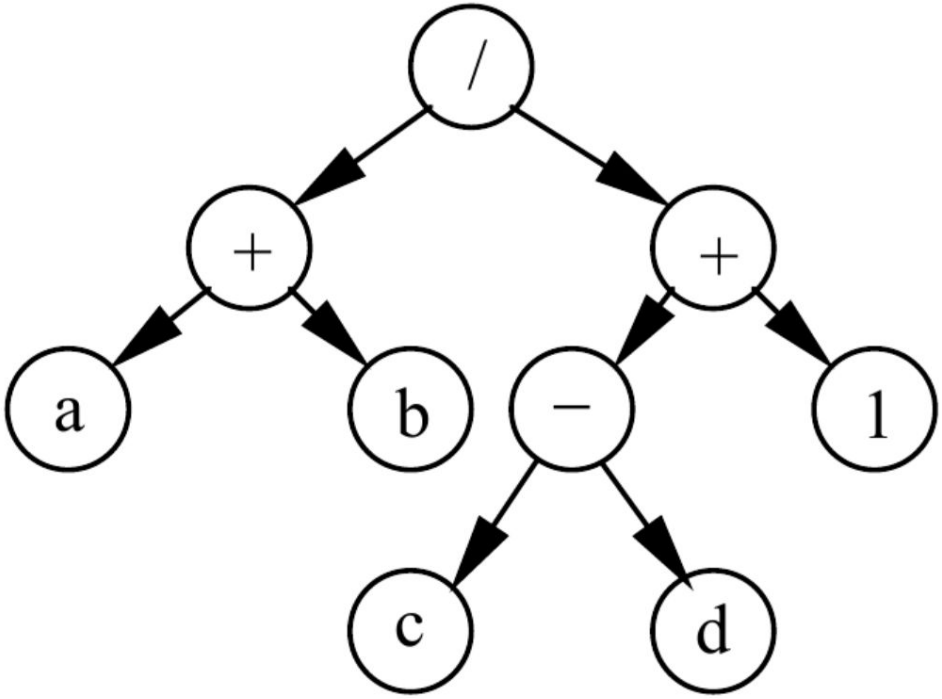
5. Добавление в правое поддерево «отца» P. «Отец»-красный, «дядя» U - чёрный



Дерево для арифметического выражения

(a + b) / (c - d + 1)

Листья содержат числа и имена переменных (операндов), а внутренние вершины и корень – арифметические действия и вызовы функций. Вычисляется такое выражение снизу, начиная с листьев. Как видим, скобки отсутствуют, и дерево полностью определяет порядок выполнения операций.



Операция	Отсортированный массив	Сбалансированное дерево поиска
Отыскать	$O(\log n)$	$O(\log n)$
Минимум	$O(1)$	$O(\log n)$
Максимум	$O(1)$	$O(\log n)$
Предшественник	$O(\log n)$	$O(\log n)$
Преемник	$O(\log n)$	$O(\log n)$
Вывести в отсортированном порядке	$O(n)$	$O(n)$
Выбрать	$O(1)$	$O(\log n)$
Взять ранг	$O(\log n)$	$O(\log n)$
Вставить	$O(n)$	$O(\log n)$
Удалить	$O(n)$	$O(\log n)$

MyQuiz.ru

308478