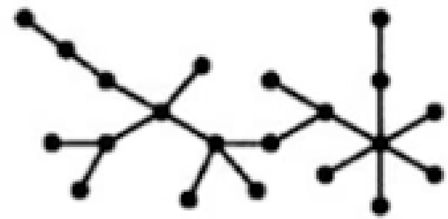
Деревья

__Определение 1.__ **Дерево (свободное)** – непустая коллекция **вершин** и **ребер**, удовлетворяющих **определяющему свойству дерева**.



Вершина (узел) – простой объект, который может содержать некоторую информацию.

Ребро – связь между двумя вершинами.

Путь в дереве – список отдельных вершин, в котором следующие друг за другом вершины соединяются ребрами дерева.

Определяющее свойство дерева – существование только одного пути, соединяющего любые два узла.

__Определение 2__ **(равносильно первому). Дерево (свободное)** – неориентированный связный граф без циклов.

Дерево
(частный вид ациклического графа)

Определение. *(Ориентированным) деревом T* называется (ориентированный) граф $G = (A,R)$ со специальной вершиной $r \in A$, называемый *корнем*, у которого

- степень по входу вершины r равна 0,
- степень по входу всех остальных вершин дерева T равна 1,
- каждая вершина $a \in A$ достижима из r.

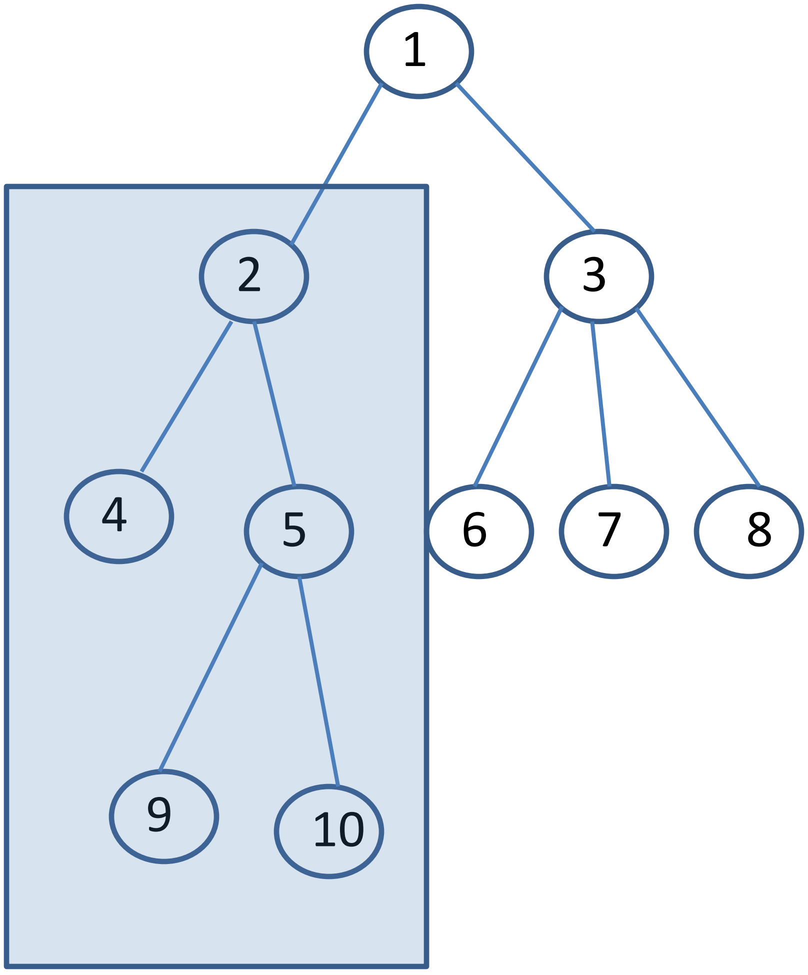
Дерево T обладает следующими свойствами:

- *T—ациклический граф,*
- *для каждой вершины дерева T существует единственный путь, ведущий из корня в эту вершину.*

Поддеревом дерева $T = (A, R)$ называется любое дерево

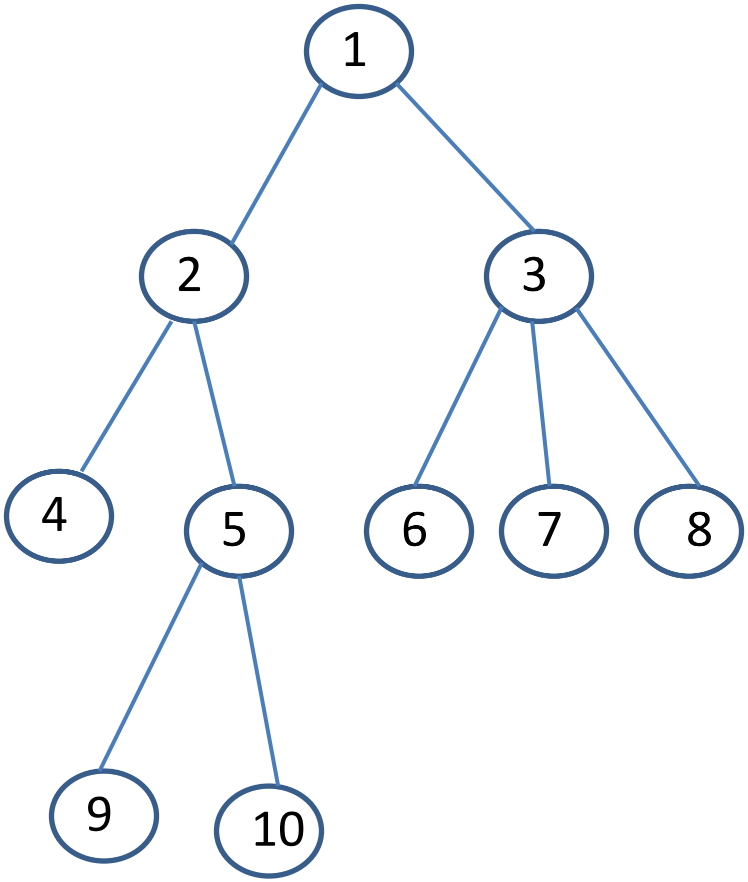
$T' = (A', R')$, у которого

1) A' непусто и содержится в A,
2) $R' = (A' \times A') \cap R$,
3) ни одна вершина из множества $A \setminus A'$ не является потомком вершины из множества A'.



Ориентированный граф, состоящий из нескольких деревьев, называется *лесом*.

Пусть $T=(A, R)$ – дерево, $(a, b) \in R$, тогда a – *предок* b, а b – *потомок* a.



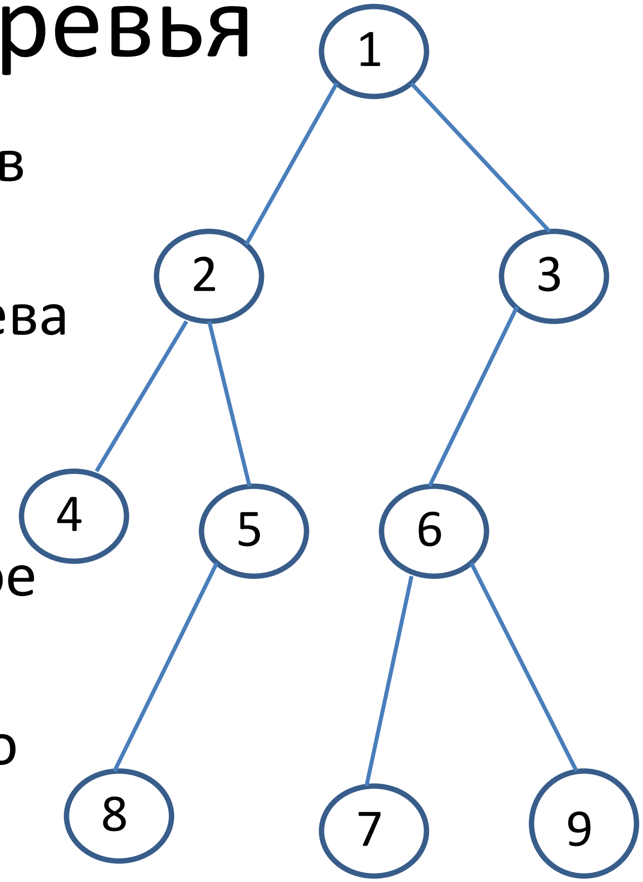
Глубина или уровень вершины – длина пути от корня до этой вершины.

Высота вершины – длина максимального пути от этой вершины до листа.

Высота дерева – длина максимального пути от корня до листа.

Глубина корня = 0.

Бинарные деревья

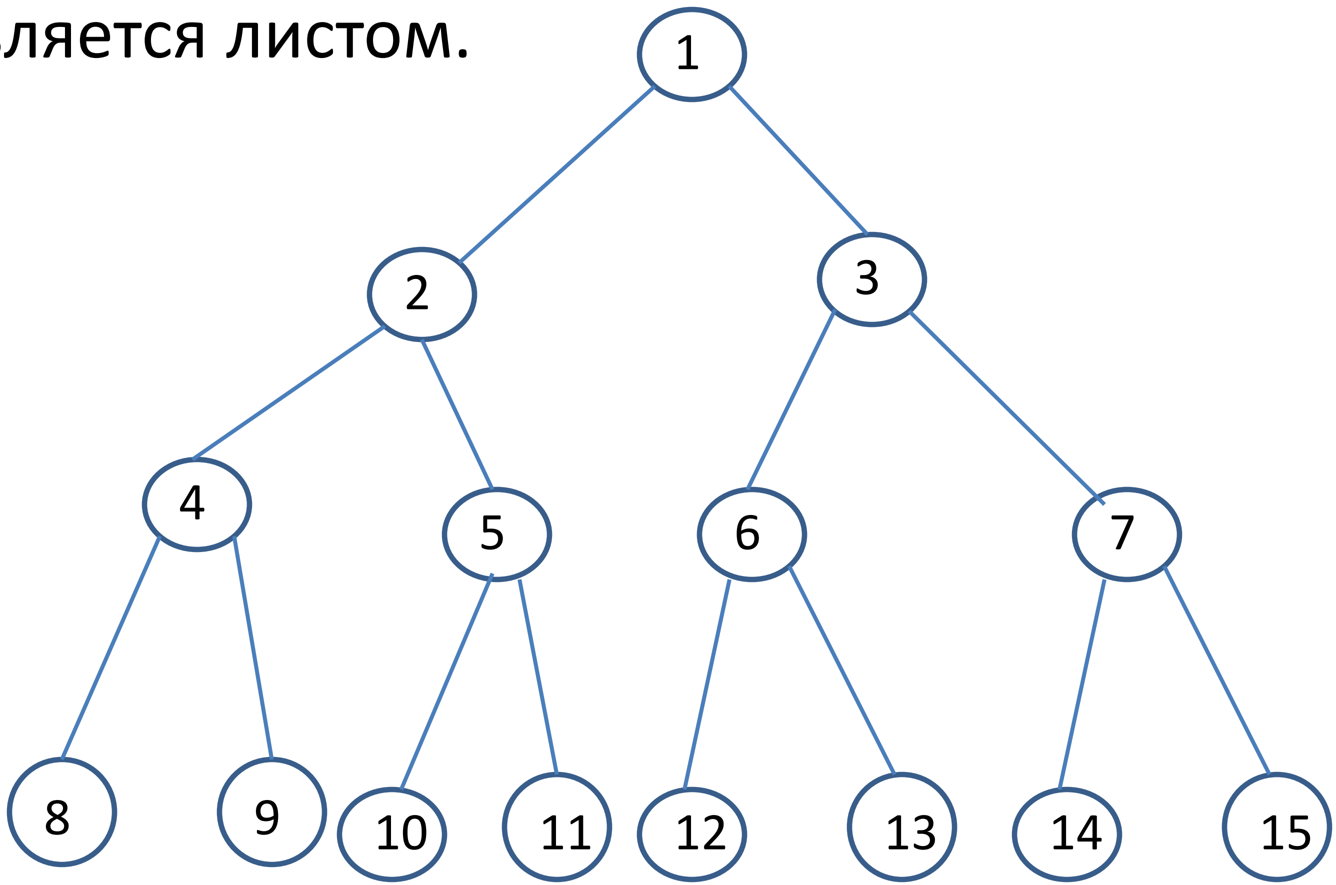


Упорядоченное дерево – это дерево, в котором множество потомков каждой вершины упорядочено слева направо.

Бинарное дерево – это упорядоченное дерево, в котором:

1) Любой потомок – либо левый либо правый,
2) любой узел имеет не более одного левого и не более одного правого потомка.

Бинарное дерево называется *полным*, если существует некоторое целое *k*, такое что любой узел глубины меньше *k* имеет как левого, так и правого потомка, а если узел имеет глубину *k*, то он является листом.



Реализация бинарного дерева

```
typedef struct tree {
        int sz;// размер
        int * key; // ключи
        struct tree *left, * right;
} T_tree;
T_tree * t;
t = (T_tree*)malloc(sizeof(T_tree));
if(t) {   t -> sz = k;
        t -> key =(int*)malloc(sizeof(int)*(t->sz));
        t -> left = NULL;
        t -> right = NULL;
}
```

Представление полных бинарных деревьев с помощью массива

Пусть T[2^k-1] – массив для хранения вершин дерева, k- высота дерева.

В T[1] хранится корень дерева.

Левый сын узла *i* расположен в позиции 2**i*,

правый сын – в позиции 2**i*+1.

Отец узла, находящегося в позиции i>1, расположен в позиции $\lfloor i/2 \rfloor$.

Обходы дерева

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

в глубину

Обходы

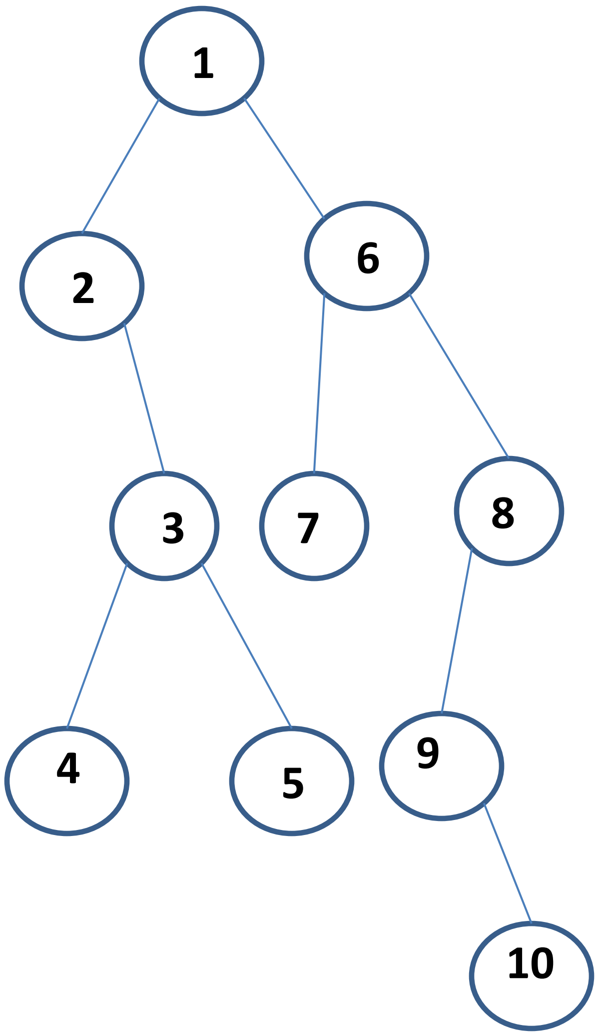
в ширину

Обходы деревьев в глубину

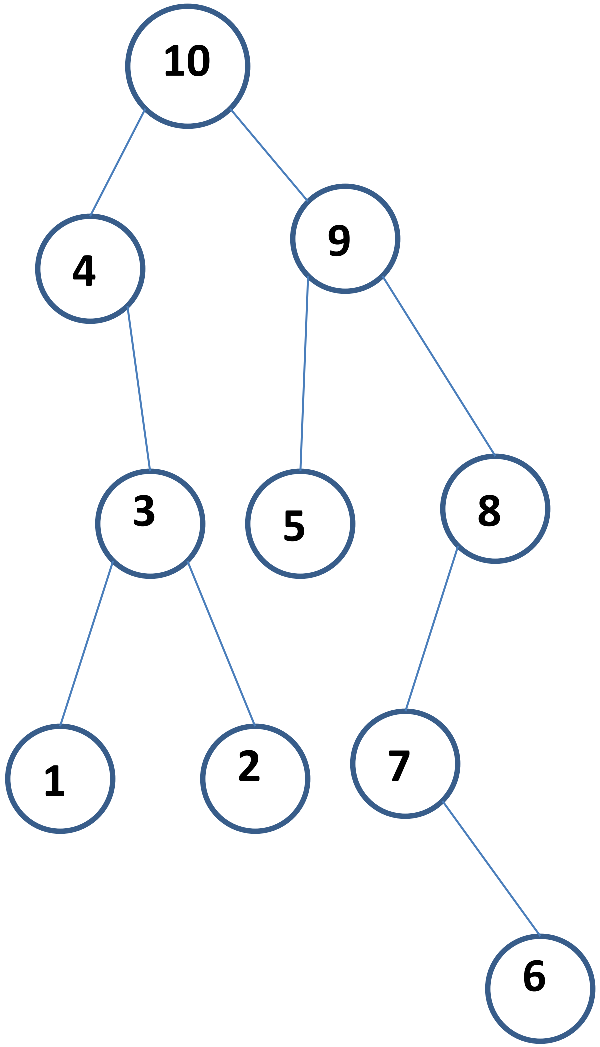
Пусть T – дерево, r- корень, v_1, v_2,…, v_n – потомки вершины r.

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

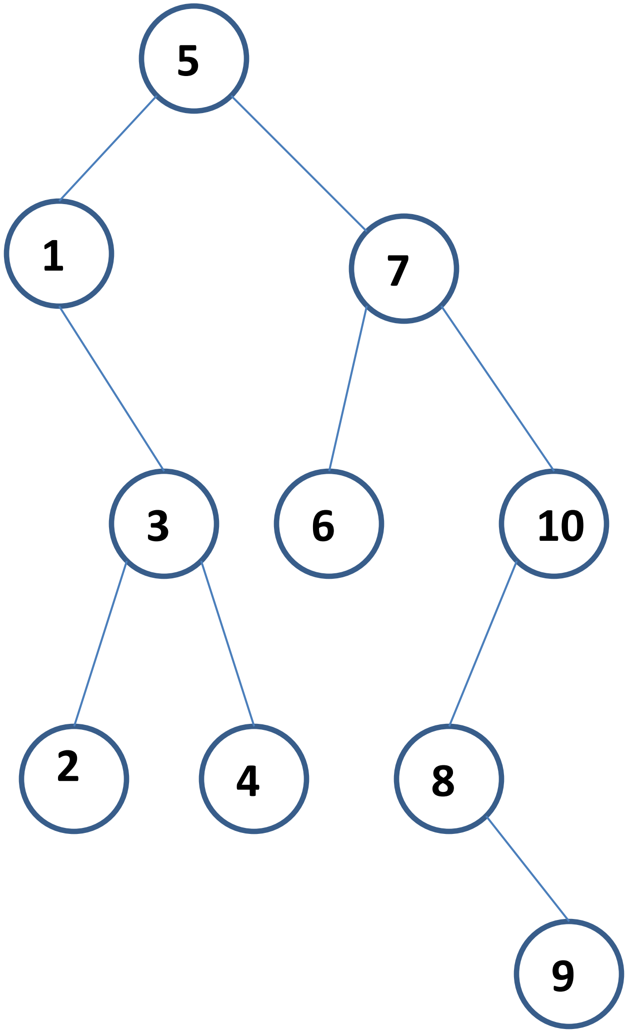
Обходы деревьев в глубину. Пример 1.



Прямой

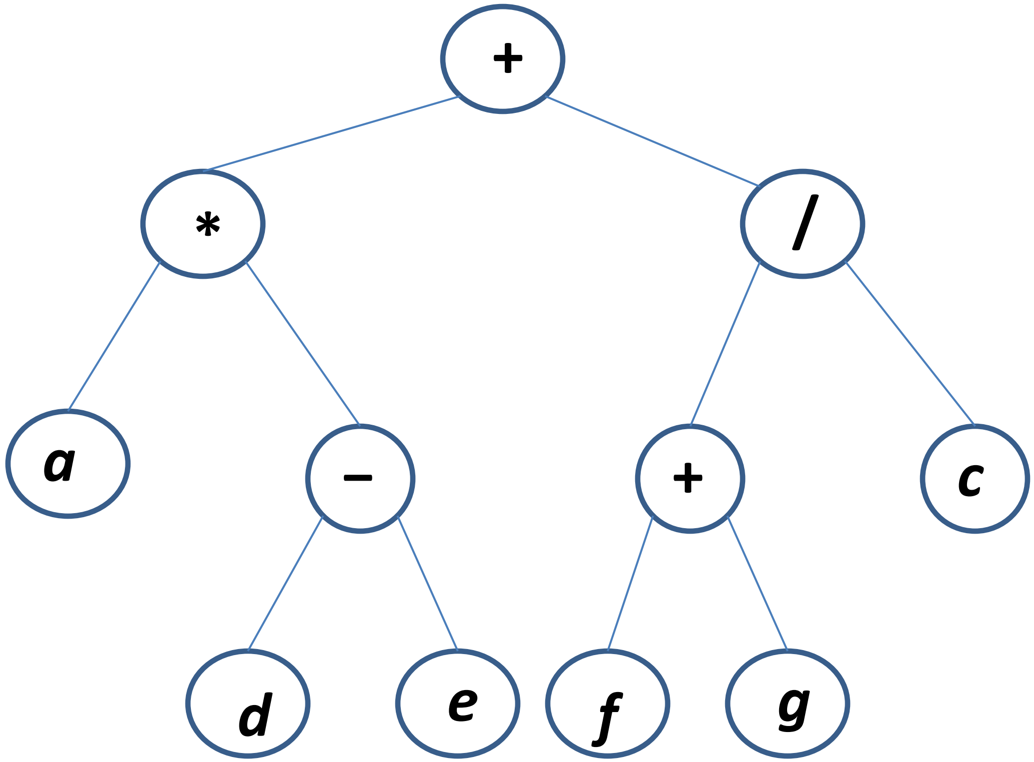


Обратный



Внутренний

Обходы деревьев в глубину. Пример 2



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

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

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

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.

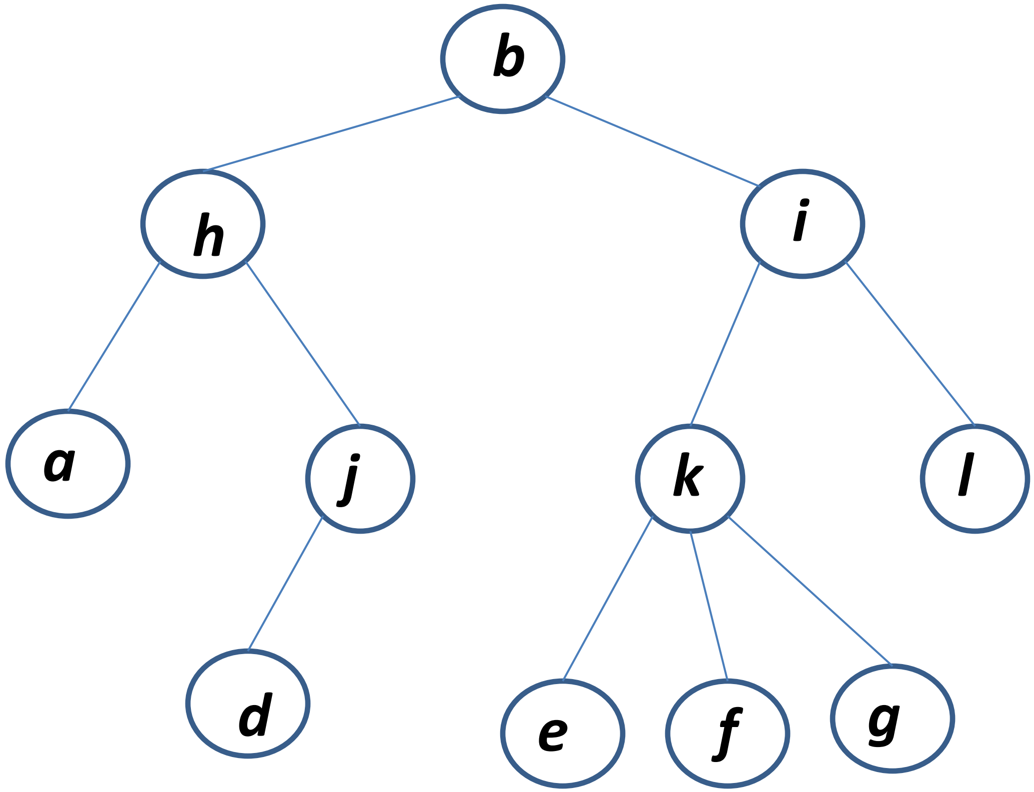
Шаг 1:

Взять из очереди очередную вершину. Поместить в очередь всех ее потомков 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 $Lrep\ (T) = a$.

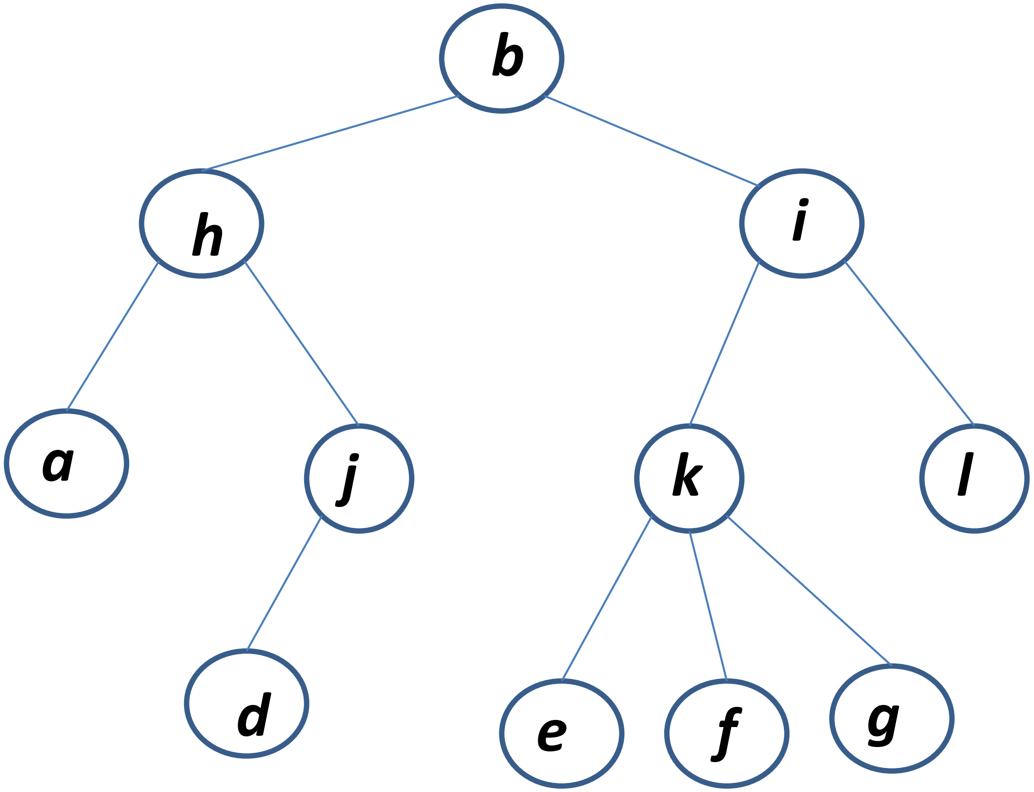
Определение. *Правое скобочное представление* $Rrep(T)$ дерева T:

(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 $Rrep(T) = a$.

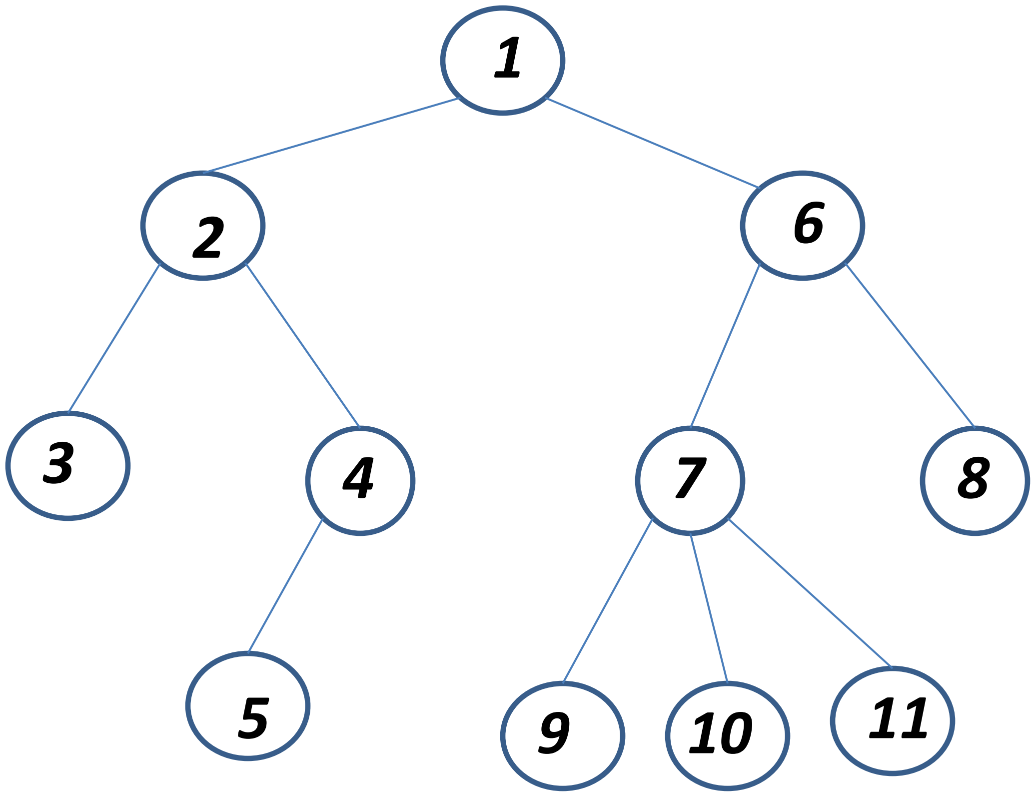
Скобочные представления деревьев



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

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.

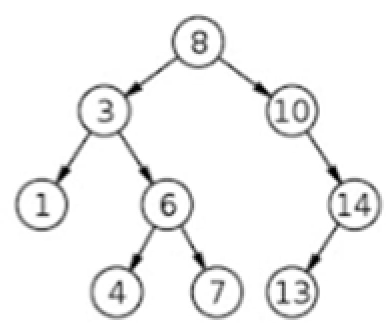


0 1 2 2 4 1 6 6 7 7 7

Дерево двоичного поиска

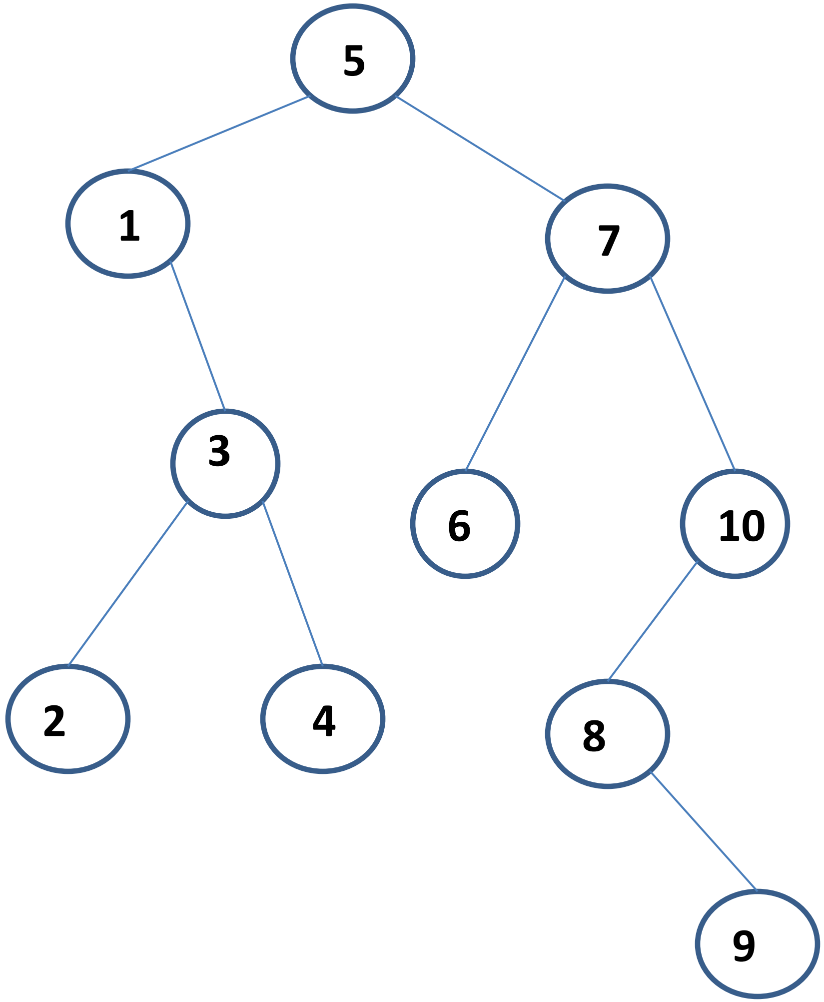
Двоичное дерево поиска – это двоичное дерево, с каждым узлом которого связан ключ, и выполняется следующее дополнительное условие:

Ключ в любом узле больше или равен ключам во всех узлах левого поддерева и меньше или равен ключам во всех узлах правого поддерева.



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;
}
```

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

Сбалансированные деревья (АВЛ)

АВЛ-дерево – сбалансированное двоичное дерево поиска. Для каждой его вершины высоты ее двух поддеревьев различаются не более чем на 1.

АВЛ-деревья

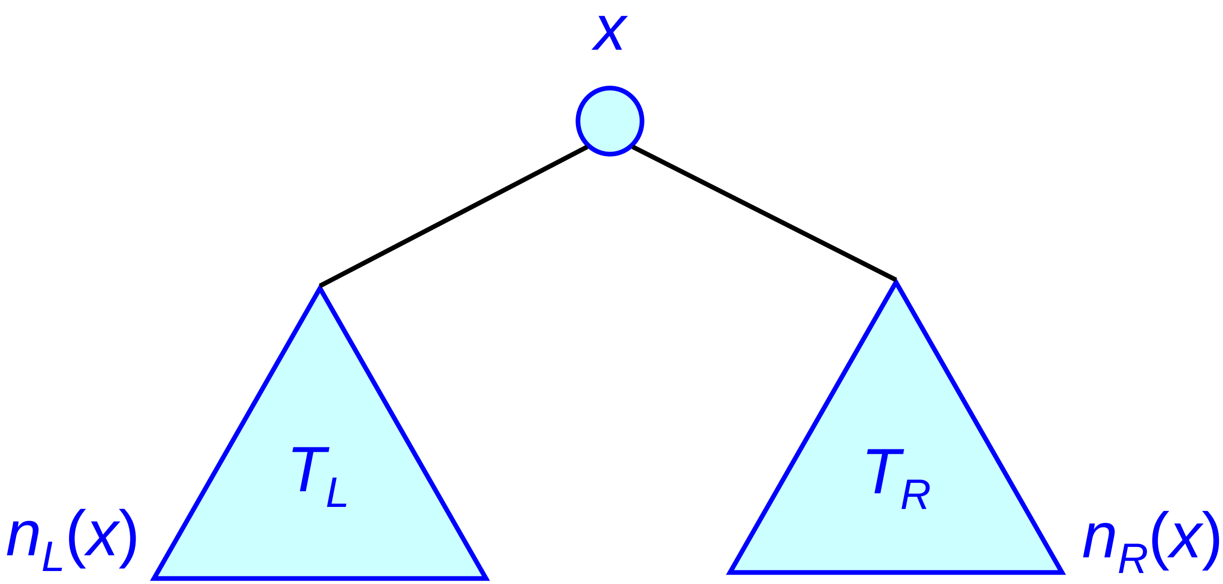
(1964 г. - Г.М.Адельсон-Вельский, Е.М. Ландис)

Теорема.

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

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

Сбалансированные деревья (АВЛ)

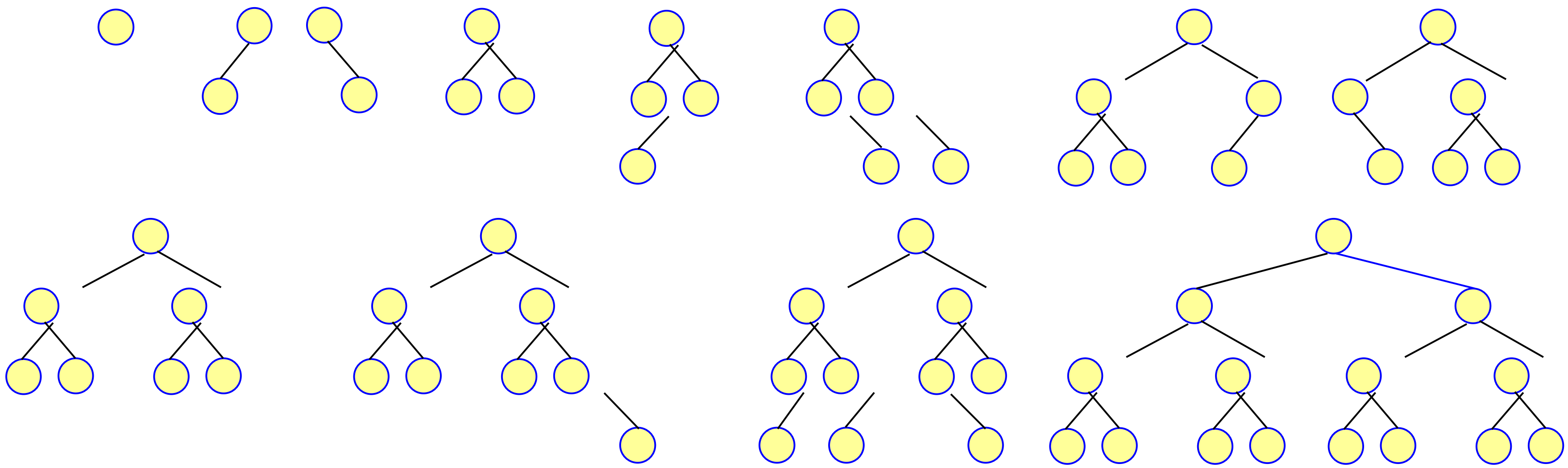


$$\forall\, x \in T$$

$$|n_L(x) - n_R(x)| \le 1,$$

где $n_L(x)$ ($n_R(x)$) – количество узлов в левом (правом) поддереве узла x

Примеры идеально сбалансированных деревьев



В идеально сбалансированном дереве число узлов n и высота дерева h связаны соотношением

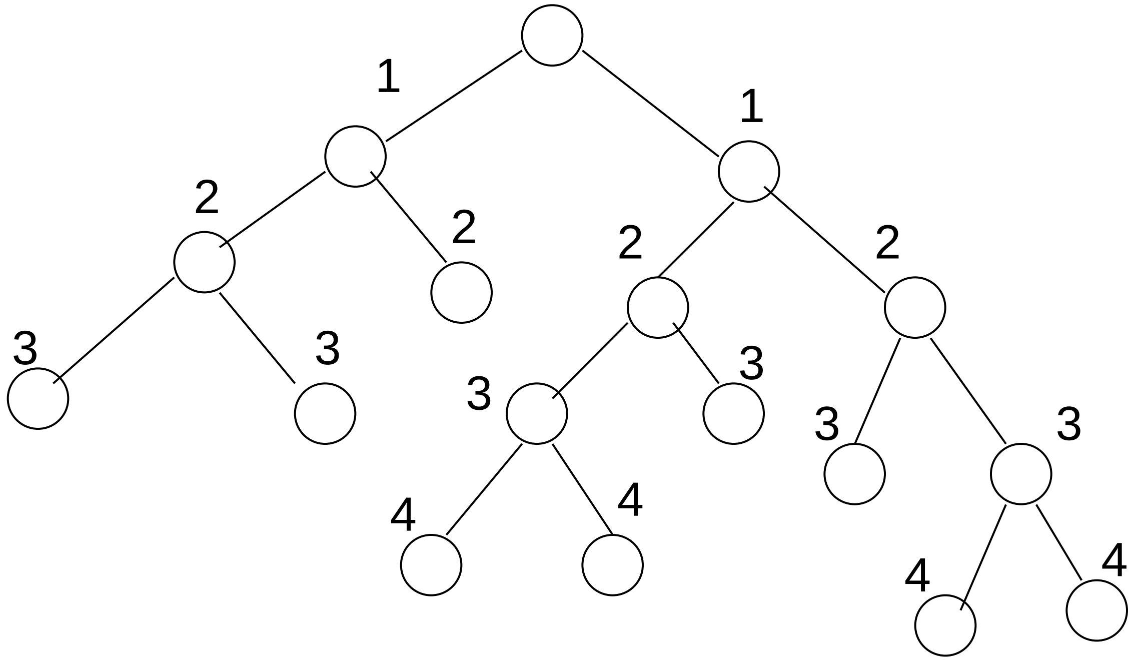
$$\boxed{2^{h-1} < n+1 \le 2^h}$$

или

$$\boxed{h = \left\lceil \log_2(n+1) \right\rceil}^{*}$$

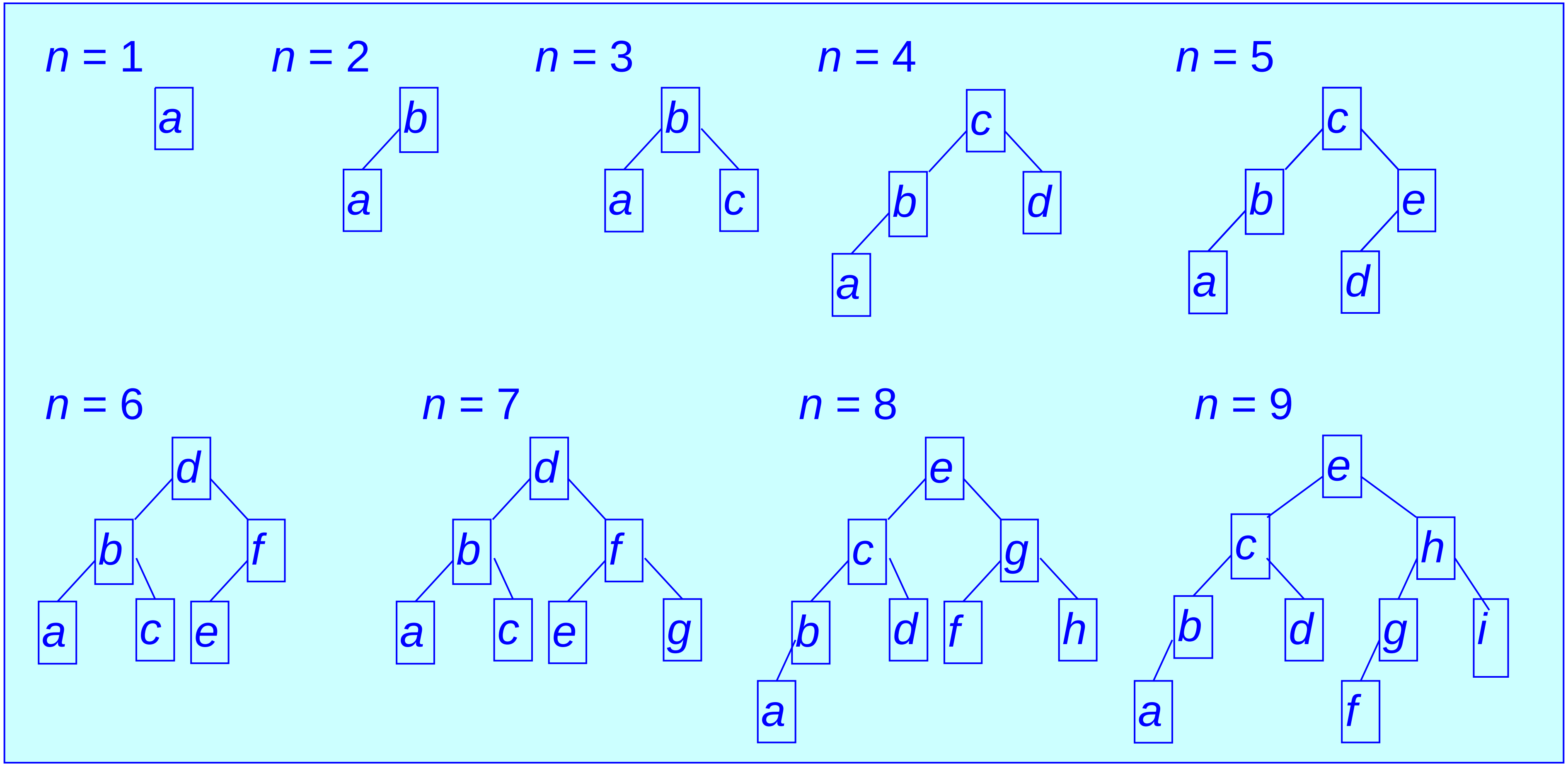
Высота дерева определена так, что при $n = 0$ имеем $h = 0$, а при $n = 1$ имеем $h = 1$

Пример



Алгоритм построения сбалансированного дерева

Примеры работы алгоритма. Пусть во входном файле находится последовательность *a, b, c, d, e, f, g, h, i.*

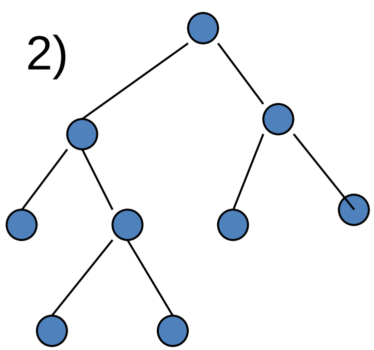


Утверждение

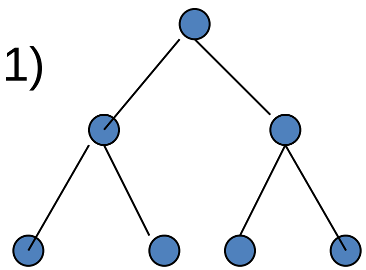
n – количество концевых вершин
m – неотрицательное целое
d – длина пути от корня до вершины

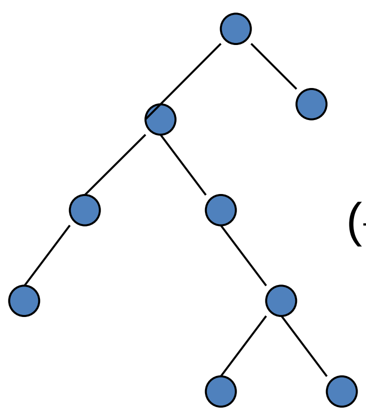
Дерево сблалансировано, если

1) $n = 2^m \Rightarrow d = m$
2) $2^m < n < 2^m \implies d = m \vee d = m + 1$



1)

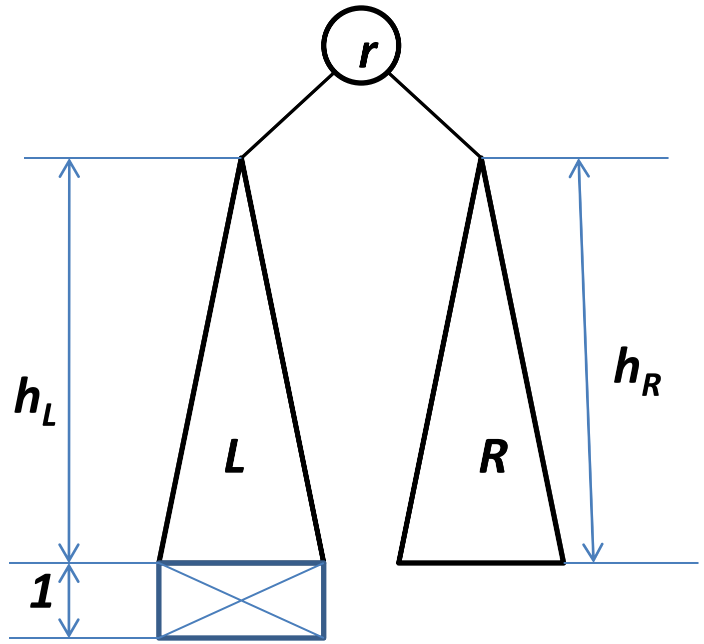




$(\neg 1 \;\&\; \neg\, 2)$

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*

Показатель сбалансированности

Операции вставки и удаления узла должны постоянно отслеживать соотношение высот левого и правого поддеревьев узла.

Для хранения этой информации можно добавить поле balanceFactor, которое содержит разность высот правого и левого поддеревьев.

balanceFactor = height(right subtree) - height(left subtree)

В AVL-дереве показатель сбалансированности должен быть в диапазоне [-1, 1].

- -1: Высота левого поддерева на 1 больше высоты правого поддерева.
- 0: Высоты обоих поддеревьев одинаковы.
- +1: Высота правого поддерева на 1 больше высоты левого поддерева.

Вставка элемента (продолжение)

Процесс вставки почти такой же, что и для бинарного дерева поиска. Осуществляется рекурсивный спуск по левым и правым сыновьям, пока не встретится пустое поддерево, а затем производится пробная вставка нового узла в этом месте.

Поскольку процесс рекурсивный, обработка узлов ведется в обратном порядке. При этом показатель сбалансированности родительского узла можно скорректировать после изучения эффекта от добавления нового элемента в одно из поддеревьев.

Необходимость корректировки определяется для каждого узла, входящего в поисковый маршрут.

Три возможных ситуации

Случай 1

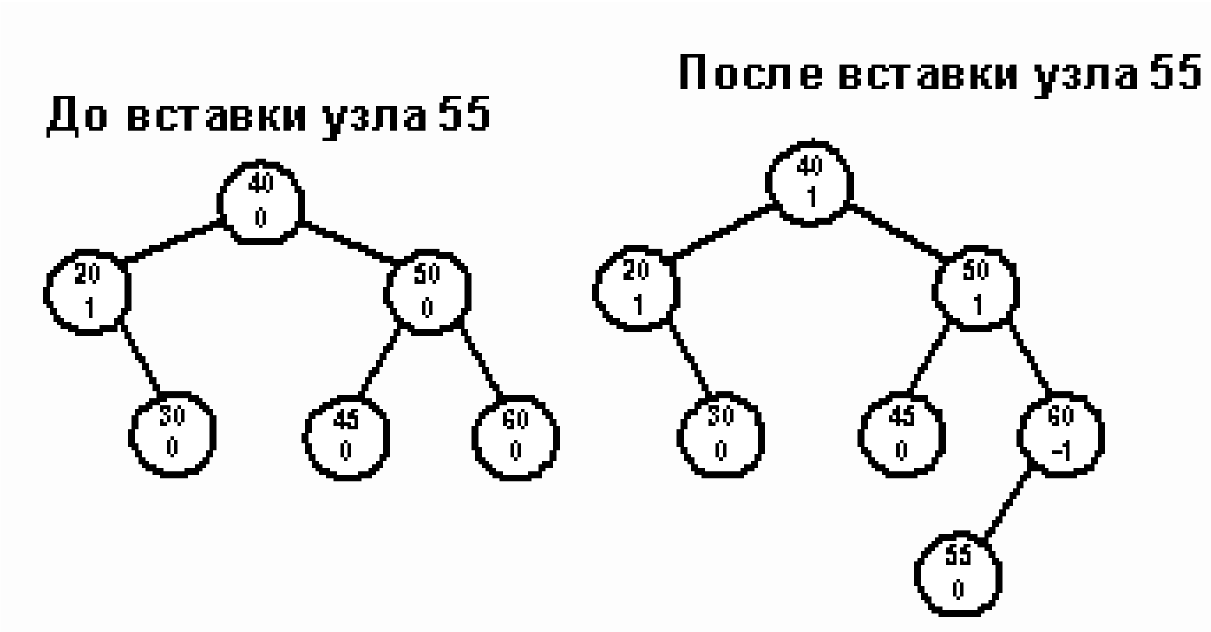
Узел на поисковом маршруте изначально является сбалансированным (balanceFactor = 0).

После вставки в поддерево нового элемента узел стал перевешивать влево или вправо в зависимости от того, в какое поддерево была произведена вставка.

Если элемент вставлен в левое поддерево, показателю сбалансированности присваивается -1, а если в правое, то 1.

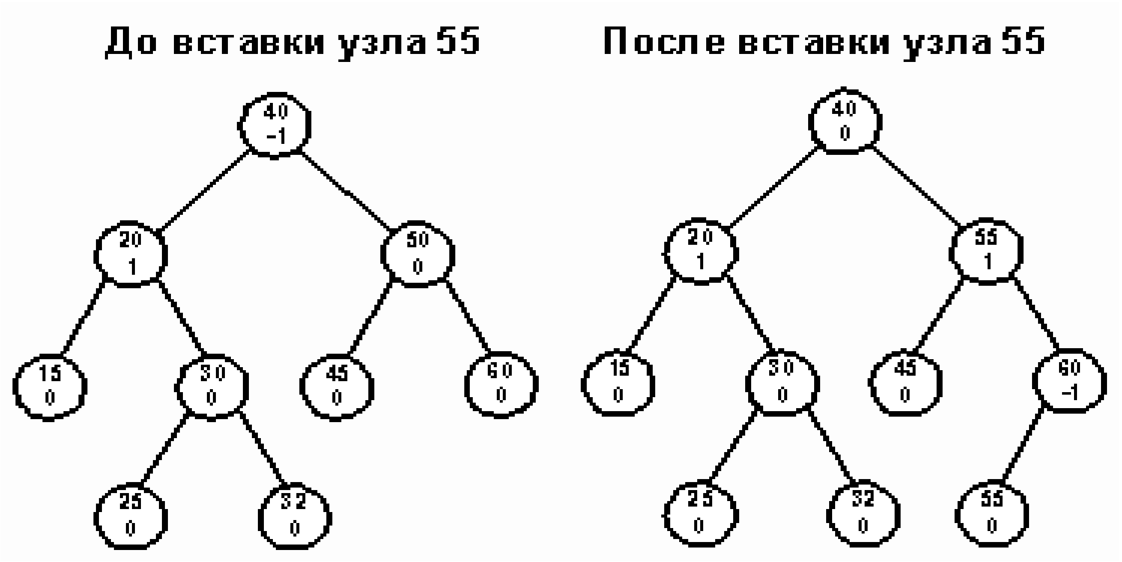
Пример

Например, на пути 40-50-60 каждый узел сбалансирован. После вставки узла 55 показатели сбалансированности изменяются.



Случай 2

Одно из поддеревьев узла перевешивает, и новый узел вставляется в более легкое поддерево. Узел становится сбалансированным.

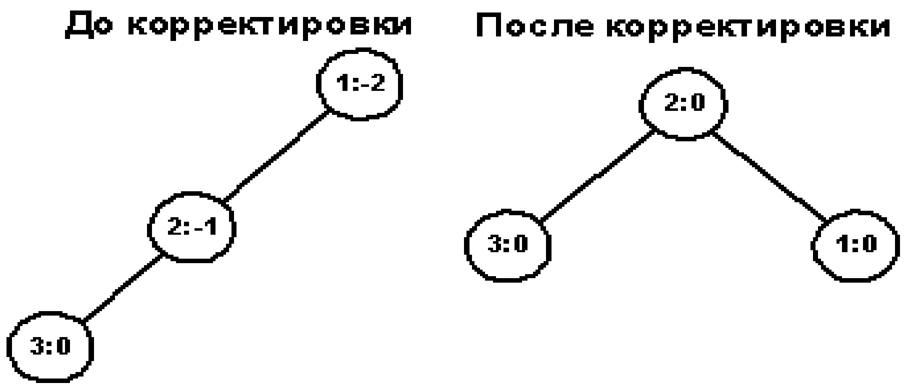


Случай 3

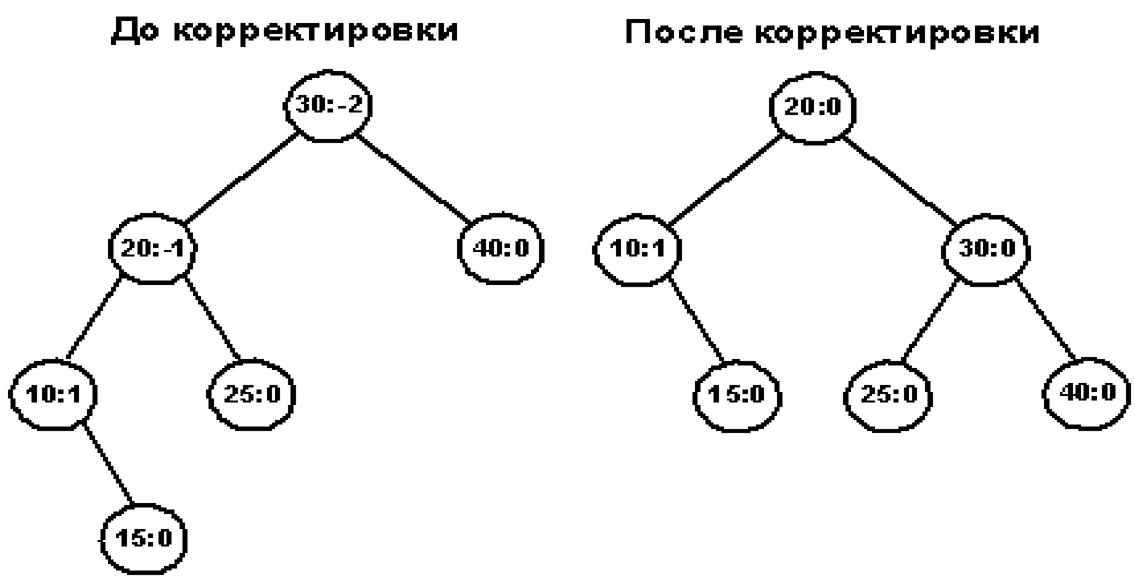
Одно из поддеревьев узла перевешивает, и новый узел помещается в более тяжелое поддерево. Тем самым нарушается условие сбалансированности, так как balanceFactor выходит за пределы -1..1.

Чтобы восстановить равновесие, нужно выполнить поворот.

Повороты

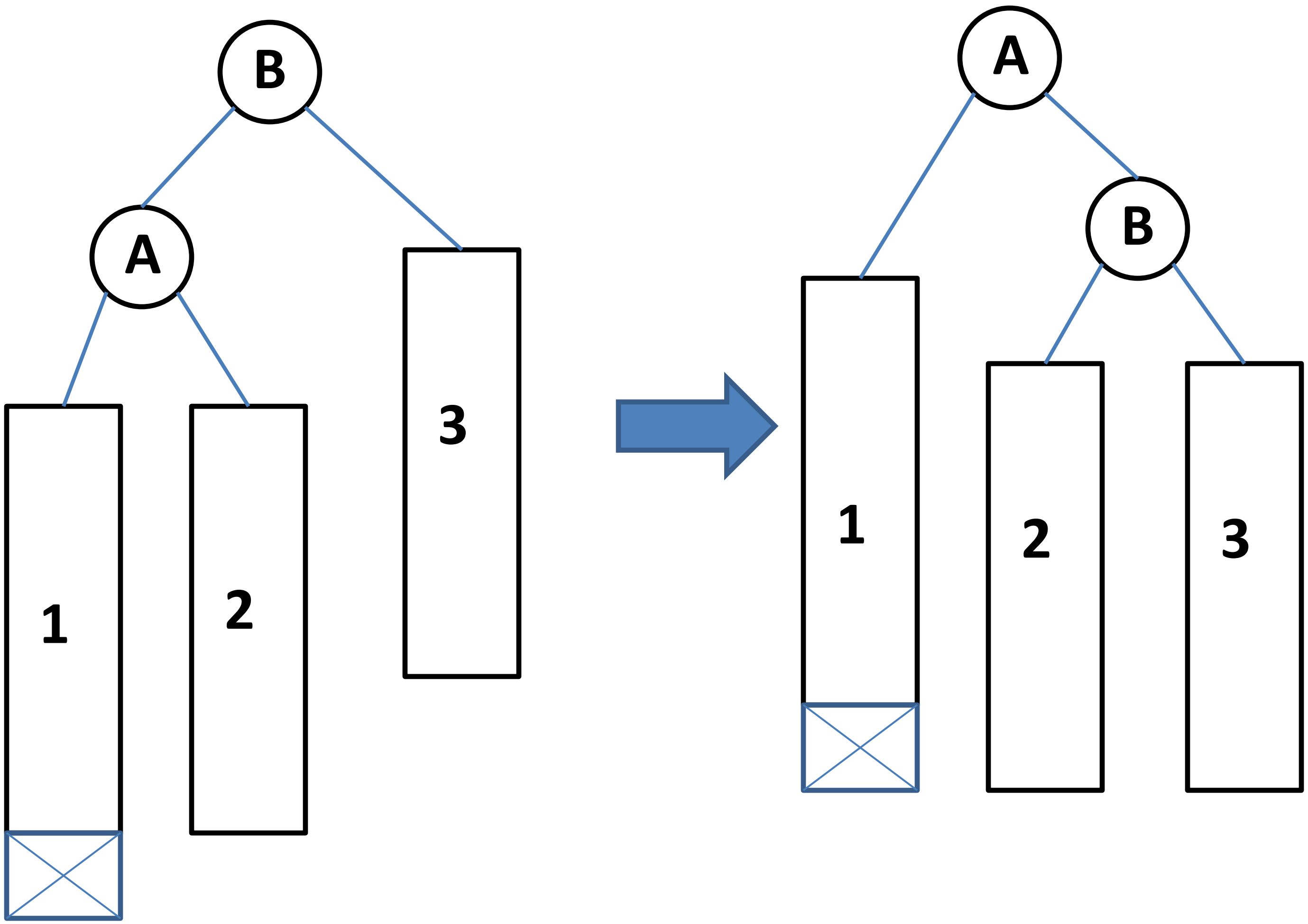


Одинарный поворот

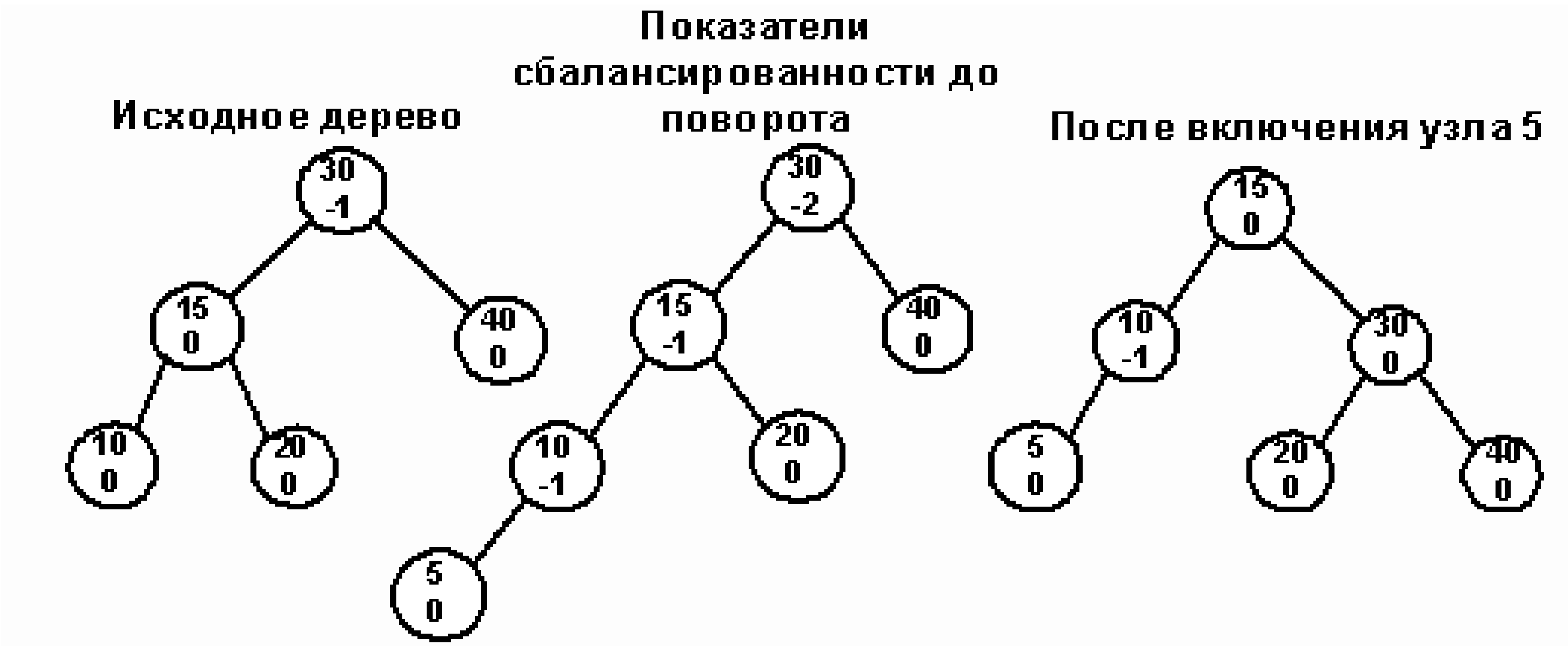


Двойной поворот

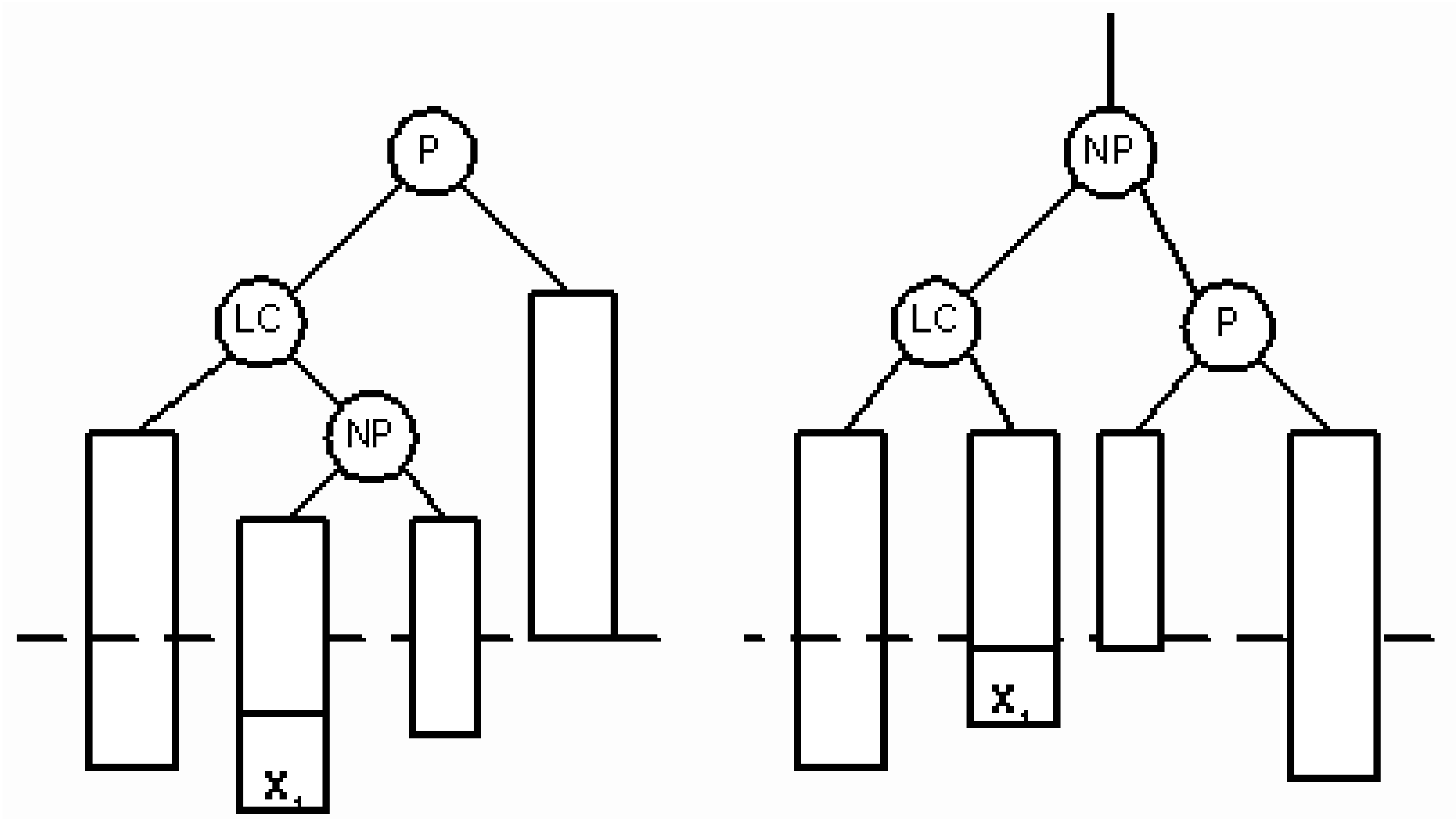
Перебалансировка - левая ветвь перегружена



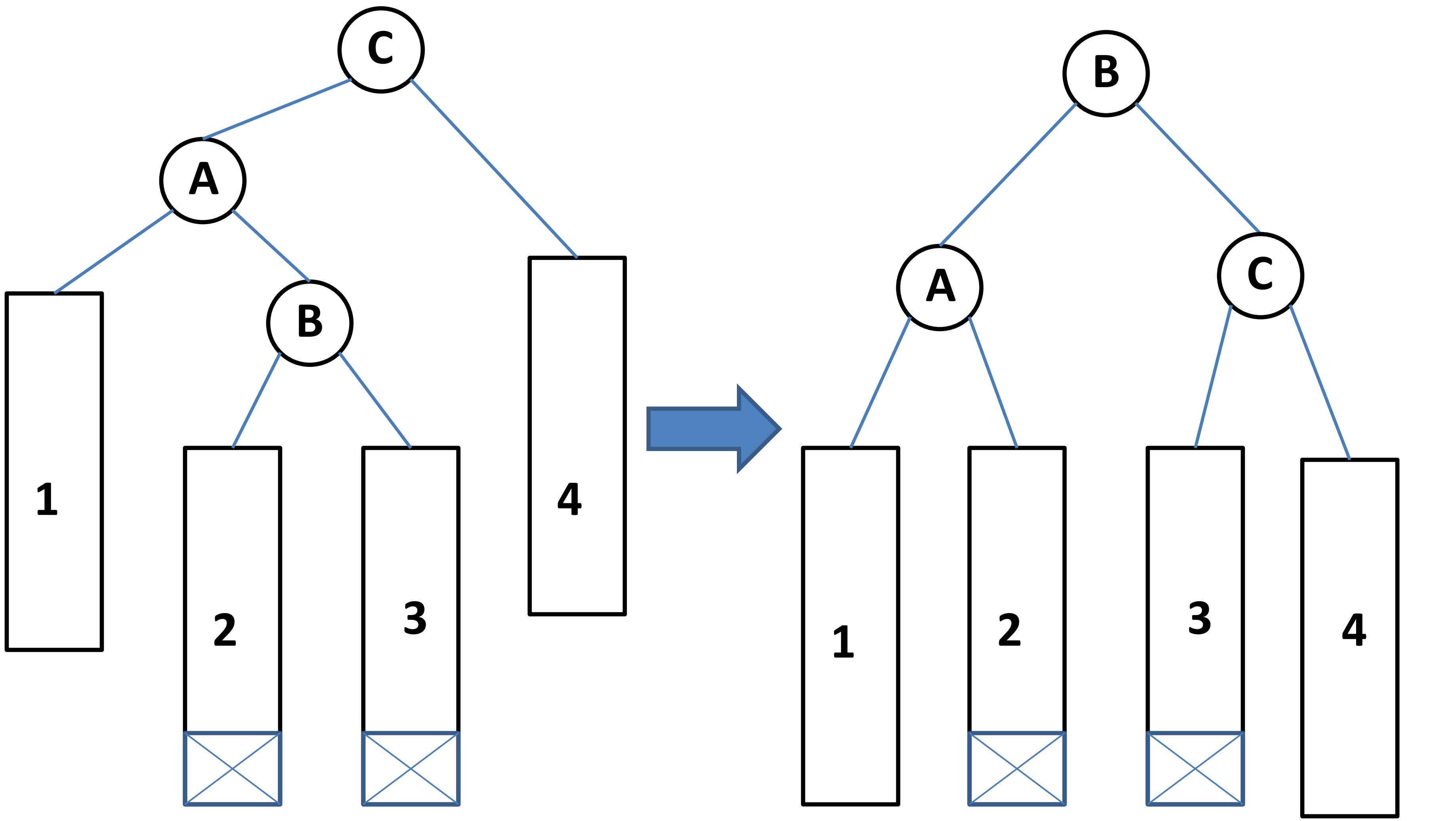
Пример


Исходное дерево
Показатели сбалансированности до поворота
После включения узла 5
30
-1
15
0
40
0
10
0
20
0
30
-2
15
-1
40
0
10
-1
20
0
5
0
15
0
10
-1
30
0
5
0
20
0
40
0

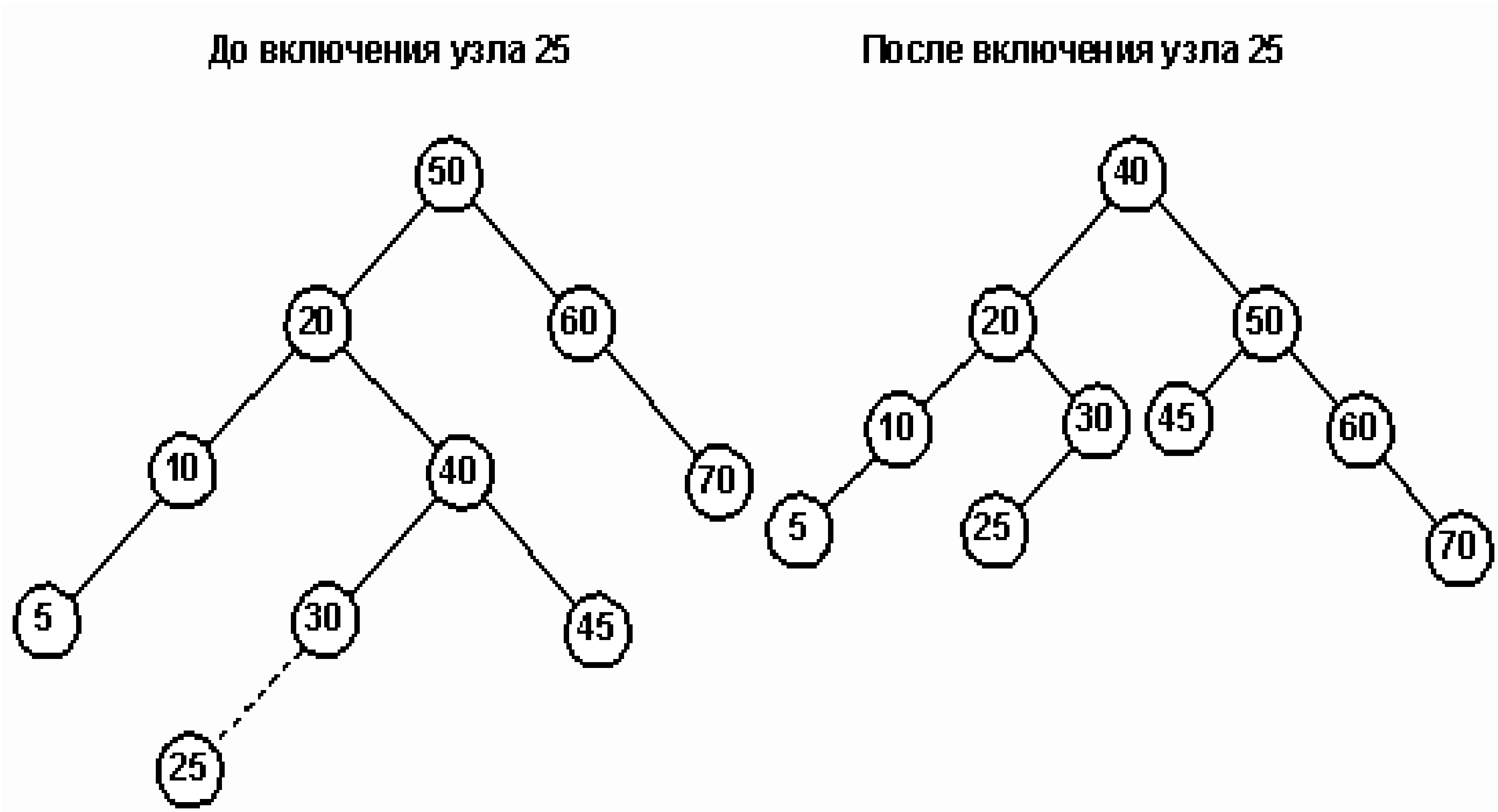
Правая ветвь левого поддерева перегружена



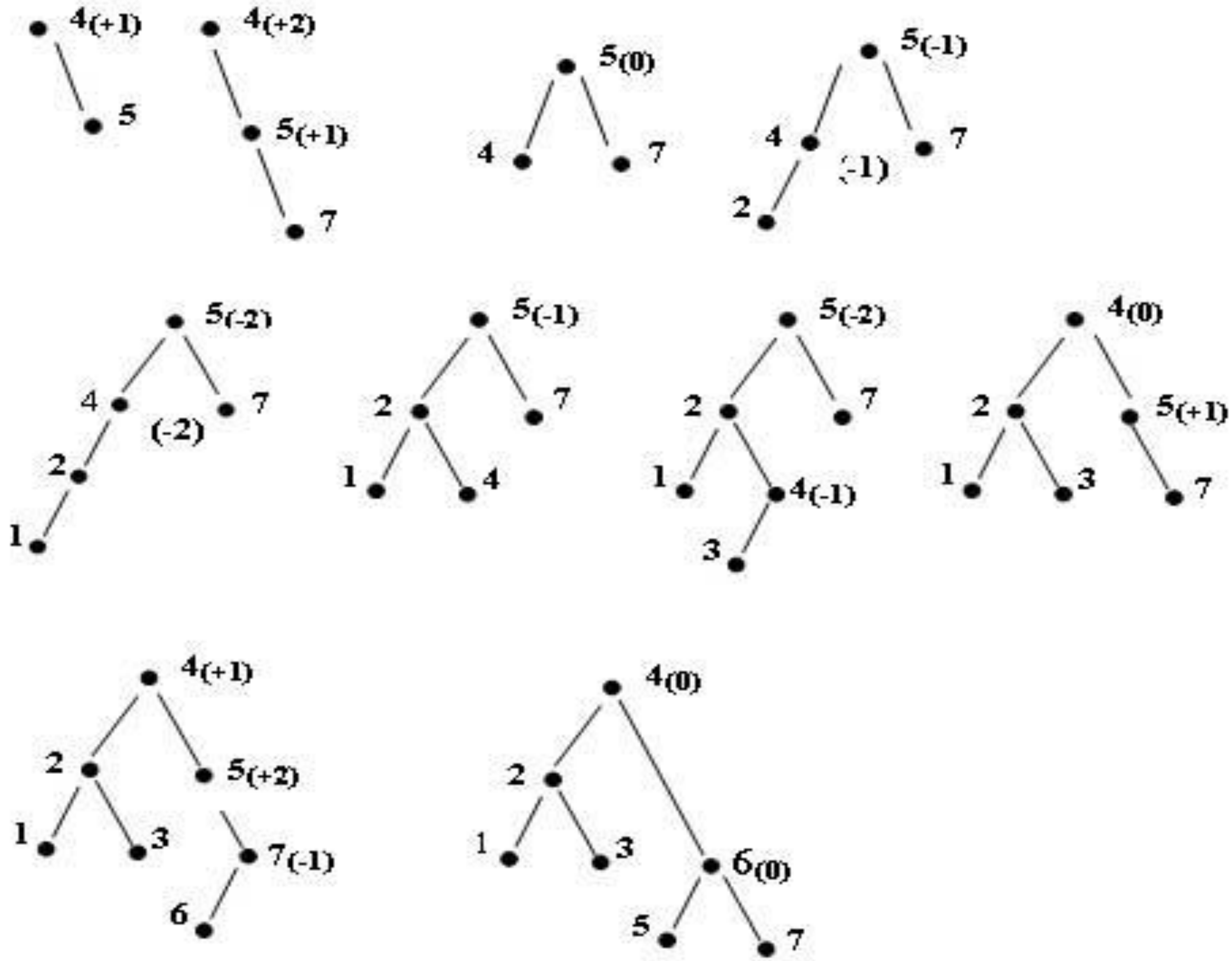
Правая ветвь левого поддерева перегружена



Пример


До включения узла 25
После включения узла 25
50
20
60
10
40
70
5
30
45
25
40
20
50
10
30
45
60
5
25
70

Пример построения АВЛ-дерева



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

Красно-чёрное дерево обладает следующими свойствами.

1) Все листья черны.

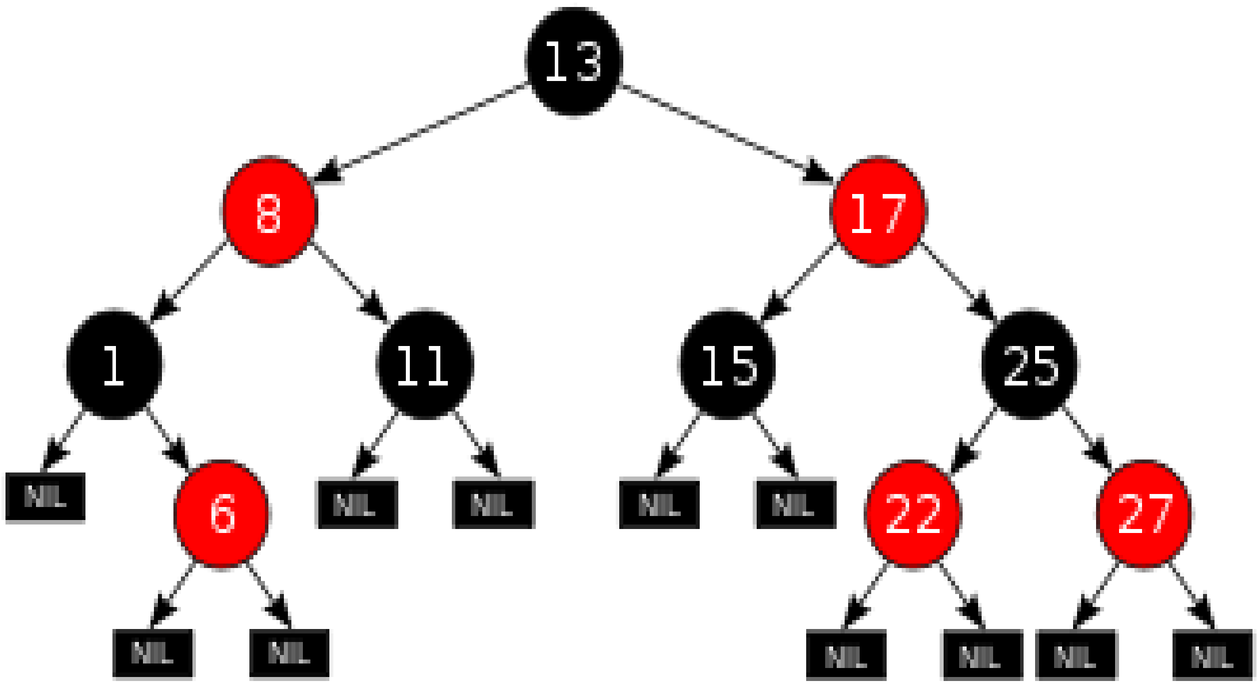
2) Все потомки красных узлов черны (запрещена ситуация с двумя красными узлами подряд).

3) На всех ветвях дерева, ведущих от его корня к листьям, число чёрных узлов одинаково.

Это число называется чёрной высотой дерева.

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

Пример



13
8
17
1
11
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL
NIL
NIL

Свойства красно-чёрного дерева

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

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

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

Сравнение с АВЛ-деревом

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

Тогда:

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

  Поскольку $N(0) = 1$, $N(1) = 1$, N(h) растёт как последовательность Фибоначчи, следовательно, $N(h) = \Theta(\lambda h)$, где

  $$\lambda = (\sqrt{5}+1)/2 \approx 1,62$$

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

Следовательно, при том же количестве листьев красно-чёрное дерево может быть выше АВЛ-дерева, но не более чем

$\log \lambda / \log \sqrt{2} \approx 1,388$ раз.

Поиск, вставка, удаление

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

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

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

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

Операция вставки нового узла

Чтобы вставить узел, мы сначала

- ищем в дереве место, куда его следует добавить.
- Новый узел всегда добавляется как лист, поэтому оба его потомка являются **NULL**-узлами и предполагаются черными.
- После вставки красим узел в красный цвет.
- После этого смотрим на предка и проверяем, не нарушается ли красно-черное свойство. Если необходимо, мы перекрашиваем узел и производим поворот, чтобы сбалансировать дерево.

Вставив красный узел с двумя **NULL**-потомками, мы сохраняем свойство черной высоты. Однако, при этом может оказаться нарушенным свойство 3, согласно которому оба потомка красного узла обязательно черны. В нашем случае оба потомка нового узла черны по определению, поскольку они являются **NULL**-узлами.

Вставка нового узла

Случай 1. Если отец и дядя(другой сын деда вставляемого узла) оба красные, тогда цвет отца и дяди меняется на черный, а цвет деда на красный.

Таким образом проблема перемещается на 2 уровня вверх, и операция повторяется уже для деда узла.

Случай 2. Если отец нового узла красный, а дядя черный, существуют два похожих подслучая. Если вставляемый узел левый сын своего отца, тогда цвет отца меняется на черный, цвет деда меняется на красный и дерево поворачивается направо вокруг отца нового узла.

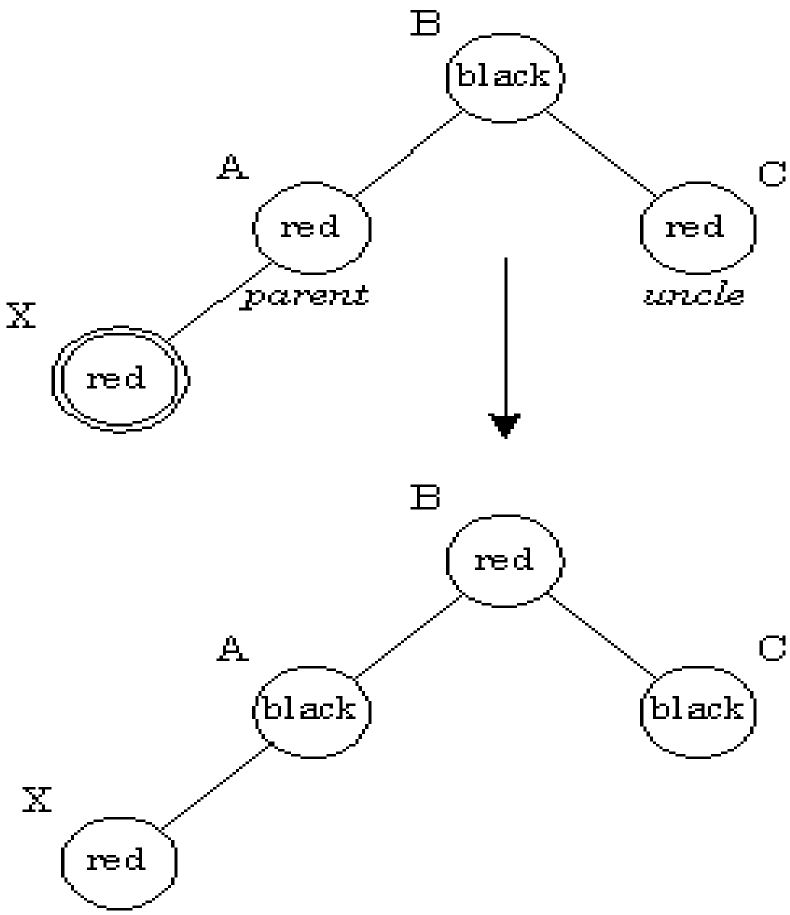
Таким образом нарушение полностью устраняется и алгоритм завершается.

Случай 3. Если новый узел правый сын своего отца, то сначала осуществляется левый поворот вокруг отца и затем выполняется все как в случае 2.

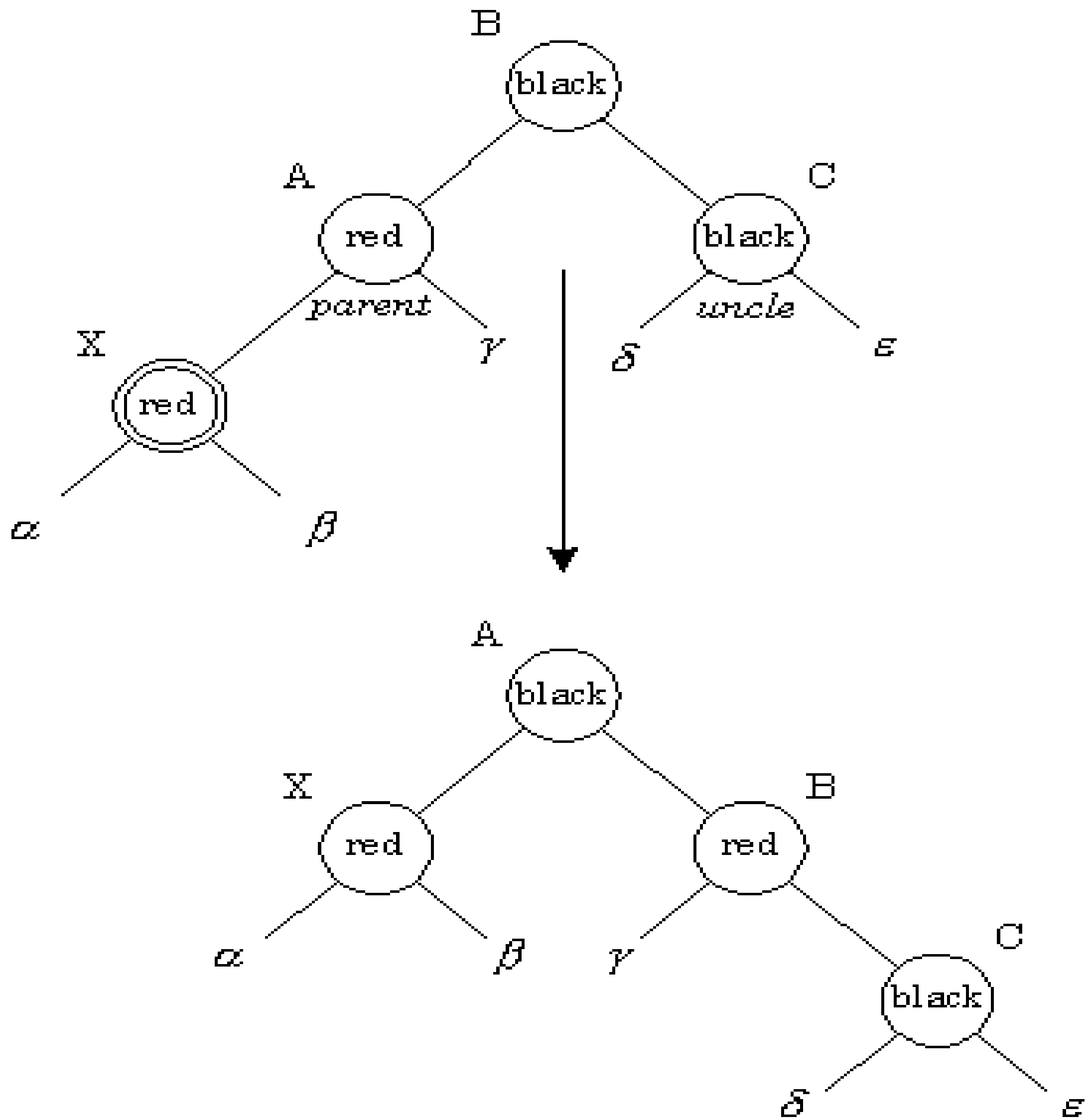
Обратите внимание на распространение влияния красного узла на верхние узлы дерева. В самом конце корень мы красим в черный цвет корень дерева.
Если он был красным, то при этом увеличивается черная высота дерева.

Красный предок, красный "дядя"

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



Красный предок, черный "дядя"



Удаление узла

Если удаляемый узел красный все правила сохраняются и все прекрасно.

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

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

Библиотека C++

Класс AVL-деревьев исторически был первым примером использования сбалансированных деревьев.

В настоящее время, однако, более популярен класс *красно-черных деревьев*.

Именно красно-черные деревья используются, например, в реализации множества и нагруженного множества (классы set и map), которая входит в стандартную библиотеку классов языка C++.

Связь с B-деревьями

RB- дерево можно рассматривать как двоичное дерево, построенное из B-дерева с максимальным количеством потомков у любой вершины = 4, по следующим правилам:

1) Каждый узел окрашен либо в красный, либо в чёрный цвет.
2) Вершина с количеством потомков ≤ 2 переносится в бинарное дерево без изменений и окрашивается в чёрный цвет.
3) В вершине с количеством потомков = 3 первый потомок присоединяется непосредственно, а другие два - через соединительный узел, соединительные узлы окрашиваются в красный цвет, остальные остаются чёрными.
4) К вершине с количеством потомков = 4 потомки присоединяются через два Соединительных узла красного цвета (по два к каждому).

Таким образом получаем бинарное дерево, являющееся моделью B-дерева с максимальным количеством потомков у вершины = 4.

В исходном B-дереве (так как оно сбалансировано) все пути от корня до любого листа имеют одинаковую длину. По построению очевидно, что любой путь в RB-дереве возрастает не более чем в два раза.
Таким образом, можем получить минимальный путь, равным по длине пути в исходном дереве, и максимальный, превышающий по длине путь в исходном дереве в два раза.

Виды записи выражений

- Префиксная (операция перед операндами)
- Инфиксная или скобочная (операция между операндами)
- Постфиксная или обратная польская (операция после операндов)

Примеры:

a + (f – b * c / (z – x) + y) / (a * r – k)   - инфиксная

+a / + – f /*b c – z x y –*a r k   - префиксная

a f b c * z x – / – y + a r * k – / +   - постфиксная

Перевод из инфиксной формы в постфиксную

Вход: строка, содержащая арифметическое выражение, записанное в инфиксной форме

Выход: строка, содержащая то же выражение, записанное в постфиксной форме (обратной польской записи).

Обозначения:

числа, строки (идентификаторы) – операнды;

Знаки операций	Приоритеты операций
(	1
)	2
=	3
+, –	4
***, /**	5

Алгоритм

Шаг 0:

Взять первый элемент из входной строки и поместить его в X.

Выходная строка и стек пусты.

Шаг 1:

Если X – операнд, то дописать его в конец выходной строки.

Если X = ‘(‘, то поместить его в стек.

Если X = ‘)‘, то вытолкнуть из стека и поместить в конец выходной строки все элементы до первой встреченной открывающей скобки. Эту скобку вытолкнуть из стека.

Если X – знак операции, отличный от скобок, то

пока стек не пуст, и верхний элемент стека имеет приоритет, больший либо равный приоритету X, вытолкнуть его из стека и поместить в выходную строку.

Затем поместить X в стек.

Шаг 2:

Если входная строка не исчерпана, то поместить в X очередной элемент входной строки и перейти на Шаг 1, иначе

пока стек не пуст, вытолкнуть из стека содержимое в выходную строку.

Пример

Входная строка:

a + (f – b * c / (z – x) + y) / (a * r – k)

X =

Выходная строка:

Стек:

Вычисления на стеке

Вход: строка, содержащая выражение, записанное в постфиксной форме.

Выход: число - значение заданного выражения.

Алгоритм:

Шаг 0:

Стек пуст.

Взять первый элемент из входной строки и поместить его в X.

Шаг 1:

Если X – операнд, то поместить его в стек.

Если X – знак операции, то вытолкнуть из стека два верхних элемента, применить к ним соответствующую операцию, результат положить в стек.

Шаг 2:

Если входная строка не исчерпана, то поместить в X очередной элемент входной строки и перейти на Шаг 1, иначе вытолкнуть из стека результат вычисления выражения.

Пример

Входная строка:

5 2 3 * 4 2 / – 4 / + 1 –

Стек: