09.12.2024

Балансировка деревьев. Решение конечных уравнений.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

План лекции

Решение конечных уравнений

20 минут

Эффективные деревья

70 минут

B-деревья и приложения (1 часть)

0 минут

План лекции

Решение конечных уравнений

20 минут

Эффективные деревья

70 минут

B-деревья и приложения (1 часть)

0 минут

Введение

Решение большинства физических и математических задач приводит к возникновению конечных уравнений, которые могут быть представлены в форме $f(x) = 0$. Если $f(x)$ является полиномом, такие уравнения называют ***алгебраическими***, в противном случае — ***трансцендентными***. Зачастую точное решение конечных уравнений не может быть выражено через элементарные функции. Примером может служить задача из квантовой механики о поиске уровня энергии E основного состояния в прямоугольной яме конечной глубины:

$$-\frac{\hbar^2}{2m}\psi'' + (U(x) - E)\psi = 0, \qquad U(x) = \begin{cases} -U_0, |x| \le a \\ 0, |x| > a \end{cases}$$

Волновая функция основного состояния чётная и имеет вид

$$\psi(x) = \begin{cases} A\cos kx, |x| \le a \\ Be^{-qx}, |x| > a \end{cases}, \qquad k^2 = \frac{2m}{\hbar^2}(U_0 + E), \qquad q^2 = -\frac{2m}{\hbar^2}E$$

Условие непрерывности решения $\psi(x)$ и его первой производной в точке $x = a$ приводит к трансцендентному уравнению, точное решение которого не может быть представлено в элементарных функциях:

$$ctg\sqrt{\frac{2ma^2U_0}{\hbar^2}(1-\xi)} = \sqrt{\frac{1}{\xi} - 1}, \qquad \xi = -\frac{E}{U_0}$$

Метод деления отрезка пополам

Пожалуй, самым простым в реализации методом поиска корней функции одной вещественной переменной является метод дихотомии, или деления отрезка пополам. Пусть на отрезке $[a, b]$ задана непрерывная функция $f : [a, b]$, причём $f(a) \cdot f(b) \leq 0$. Непрерывная функция принимает на отрезке все промежуточные значения и, следовательно, существует точка $x^* \in [a, b] : f(x^*) = 0$. Наша задача состоит в нахождении такой точки (корня $f(x)$).

Для решения поставленной задачи вычислим значение функции f в средней точке отрезка, f (1/2(a +b)). Если $f(a) \cdot f$(1/2 (a +b)) ≤ 0, положим a_1 = a, b_1 = 1/2 (a +b), в противном случае возьмем a_1 = 1/2 (a +b), b_1 = b. Шаг дихотомии выполнен. Мы свели задачу о поиске корня на отрезке $[a, b]$ к аналогичной задаче для отрезка [a1, b1], ширина которого в два раза меньше. Повторяя процесс многократно, получим последовательность отрезков $[a_n, b_n]$, причём bn - $an \to 0$ при $n \to \infty$.

Максимальная абсолютная погрешность определения корня x^* равна 1/2 (b - a) и на каждой итерации уменьшается вдвое. Следовательно, если искомое значение x^* известно по порядку величины, каждый шаг дихотомии даёт один дополнительный бит мантиссы. Это позволяет оценить необходимое число шагов для достижения требуемой точности:

$$N \approx \log_2 {\delta_0}/{\delta}$$

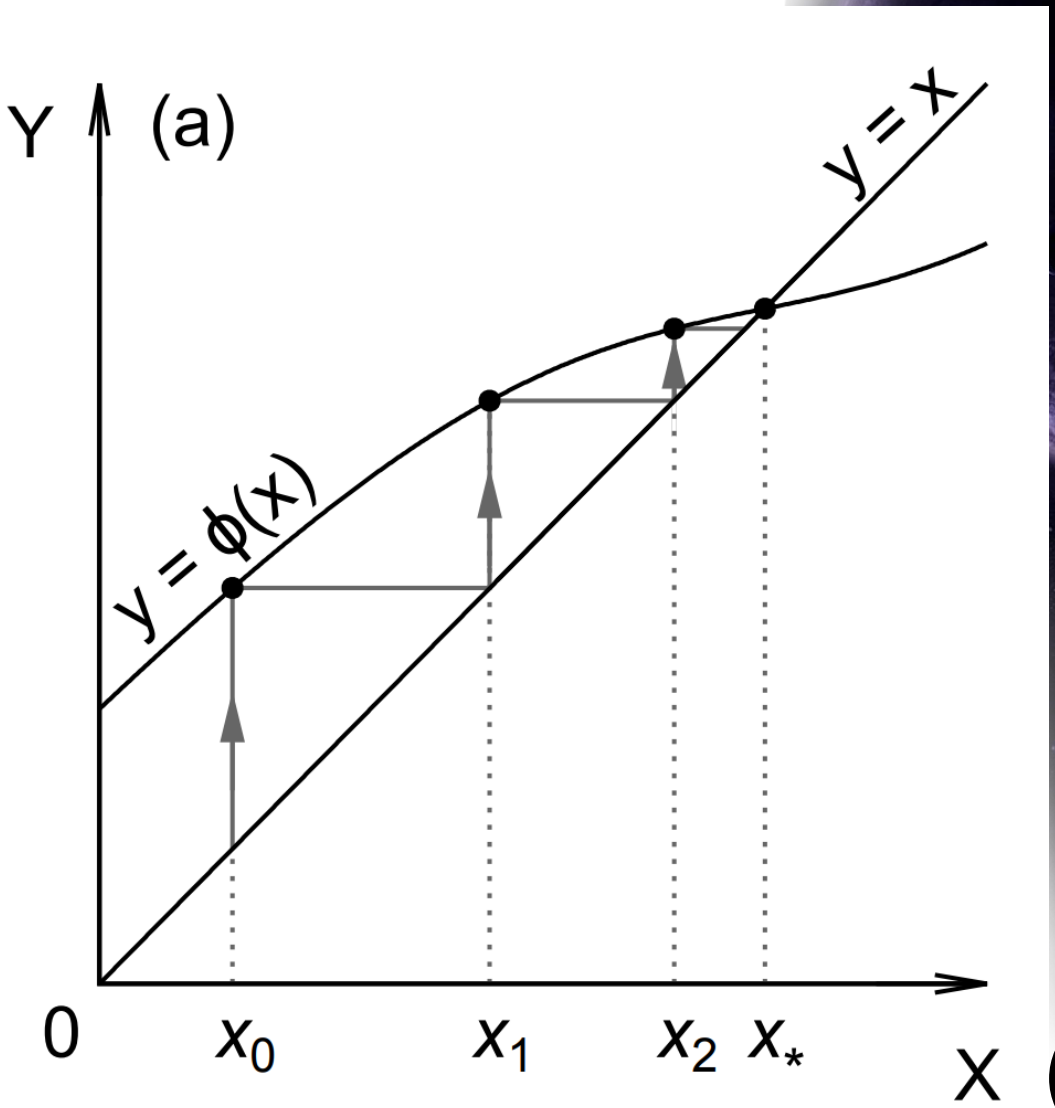
где δ_0 и δ — точность начального приближения и требуемая точность нахождения корня соответственно.

Метод простых итераций

Перепишем уравнение $f(x) = 0$ в виде $\phi(x) = x$, где $\phi(x) := f(x) + x$. Для простоты будем считать вначале, что функция $\phi(x)$ гладкая и удовлетворяет условию $\phi'(x) \le q$, где $q < 1$ — некоторая постоянная. Пусть нам также известно некоторое начальное приближение x_0 к корню x^*. Организуем итерационный процесс вида

$x_{n+1} = \phi(x_n), \qquad n = 0, 1, 2, \ldots$

Данный процесс изображён на рисунке; стрелками обозначены вычисления функции $\phi(x)$. Покажем, что предел последовательности x_n существует и равен искомому корню x^*: $\phi(x^*) = x^*$. Чтобы убедиться в этом, заметим, что функция $\phi(x)$ реализует сжимающее отображение, уменьшая длину произвольного отрезка $[a, b] \to [\phi(a), \phi(b)]$ минимум в $1/q$ раз в силу ограниченности производной $|\phi'| \le q$. При этом корень x^* по определению отображается функцией $\phi(x)$ сам в себя. Следовательно, выбирая $a = x_0$ и $b = x^*$, имеем $|x_n - x^*| \to 0$ при $n \to \infty$, т. е. $x_n \to x^*$, ч. т. д.■

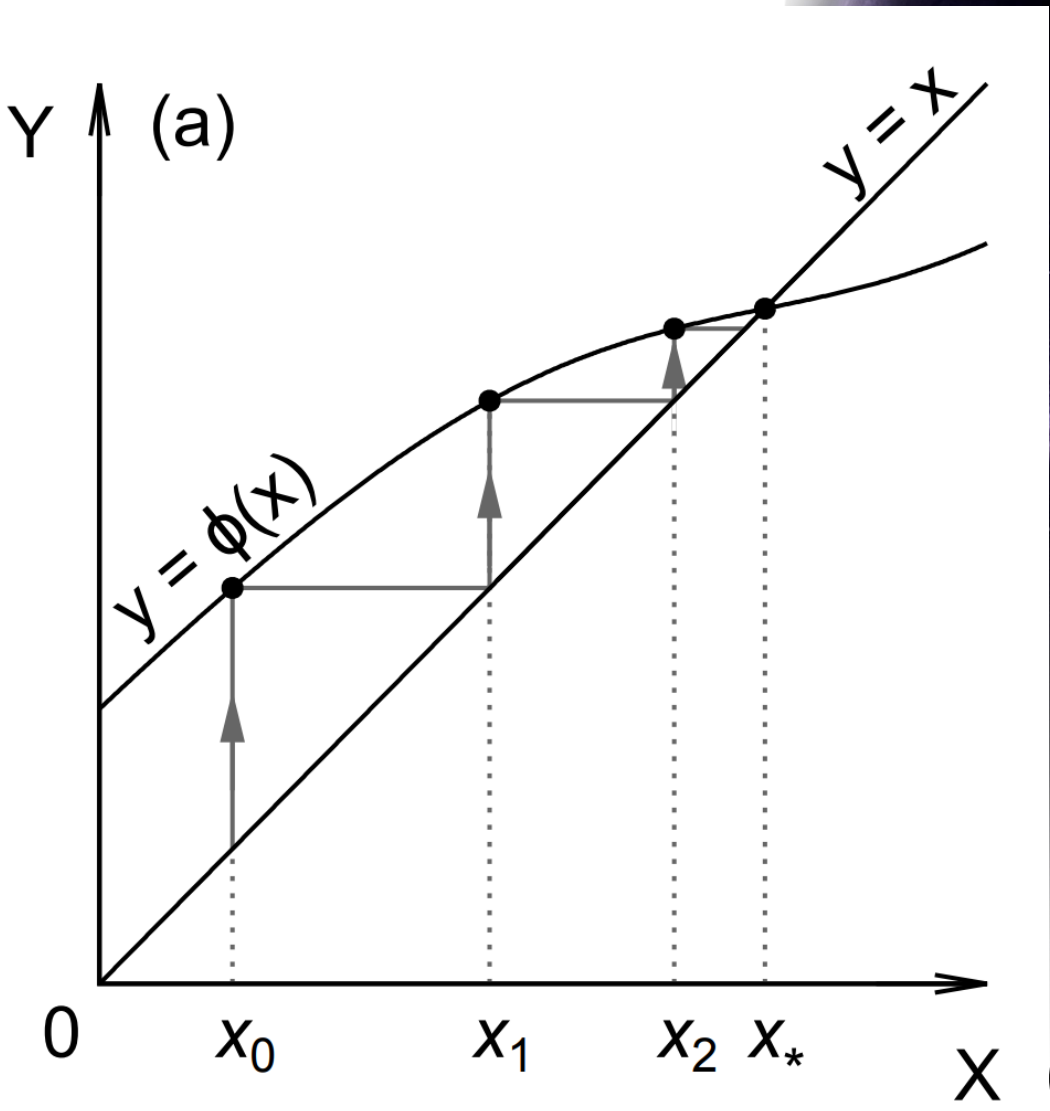


Метод простых итераций

Оценим количество итераций, необходимых для получения приближённого значения корня с требуемой точностью δ. Будем полагать функцию $\phi(x)$ гладкой, а начальное приближение $x0$ достаточно точным, так что $\phi(x_0)$ $\phi(x^*)+\phi'(x^*)$ $(x_0 - x^*)$. Учитывая, что $\phi(x^*) = x^*$, получаем $(x_1 - x^*)$ $(x_0 - x^*)$ $\phi'(x^*)$, т. е. абсолютная погрешность определения искомого корня умножается на каждой итерации на вели- чину $|\phi'| \le q < 1$, откуда получаем оценку для числа итераций:

$$N \approx \frac{\log_2 {\delta_0}/{\delta}}{-\log|\varphi'|}$$

Метод простых итераций сходится быстрее метода деления отрезка пополам при условии $|\phi'(x^*)| < 1/2$. Особенно быстрой будет сходимость при $\phi'(x^*) = 0$. Очевидно, что проведённое выше рассмотрение и оценка количества итераций впоследнем случае будут неприменимы: при малых $|\phi'(x^*)|$ в разложении $\phi(x_0) \approx \phi(x^*)+\phi'(x^*)$ $(x_0 - x)$

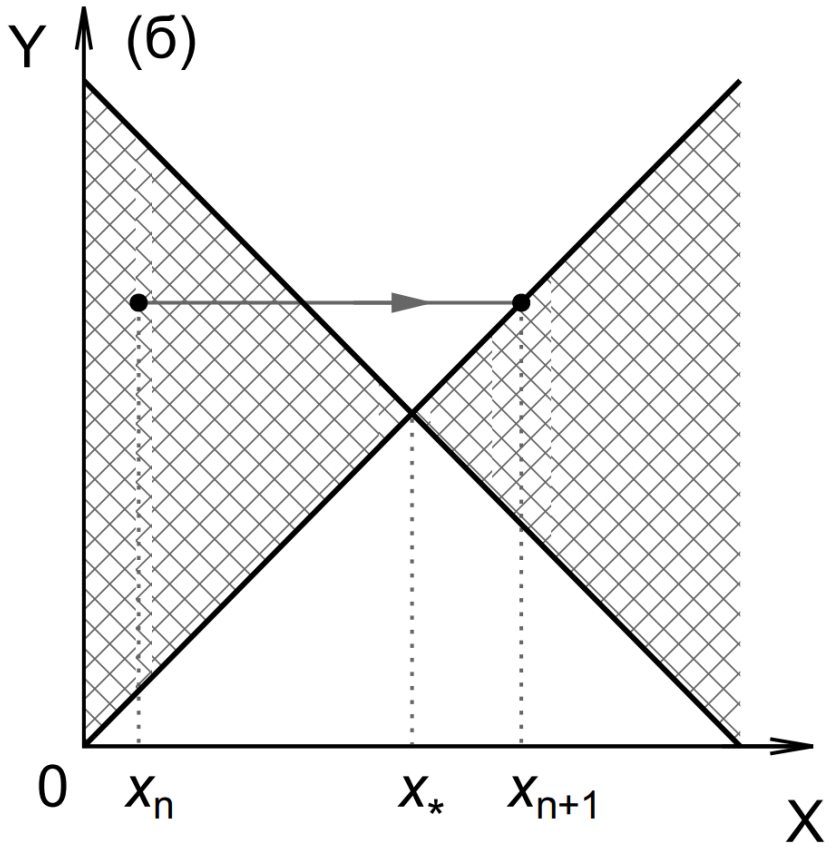


Метод простых итераций

Метод простых итераций сходится быстрее метода деления отрезка пополам при условии $|\phi'(x^*)| < 1/2$. Особенно быстрой будет сходимость при $\phi'(x^*) = 0$. Очевидно, что проведённое выше рассмотрение и оценка количества итераций в последнем случае будут неприменимы: при малых $|\phi'(x^*)|$ в разложении $\phi(x_0) \approx \phi(x^*)+\phi'(x^*)\ (x_0 - x)$

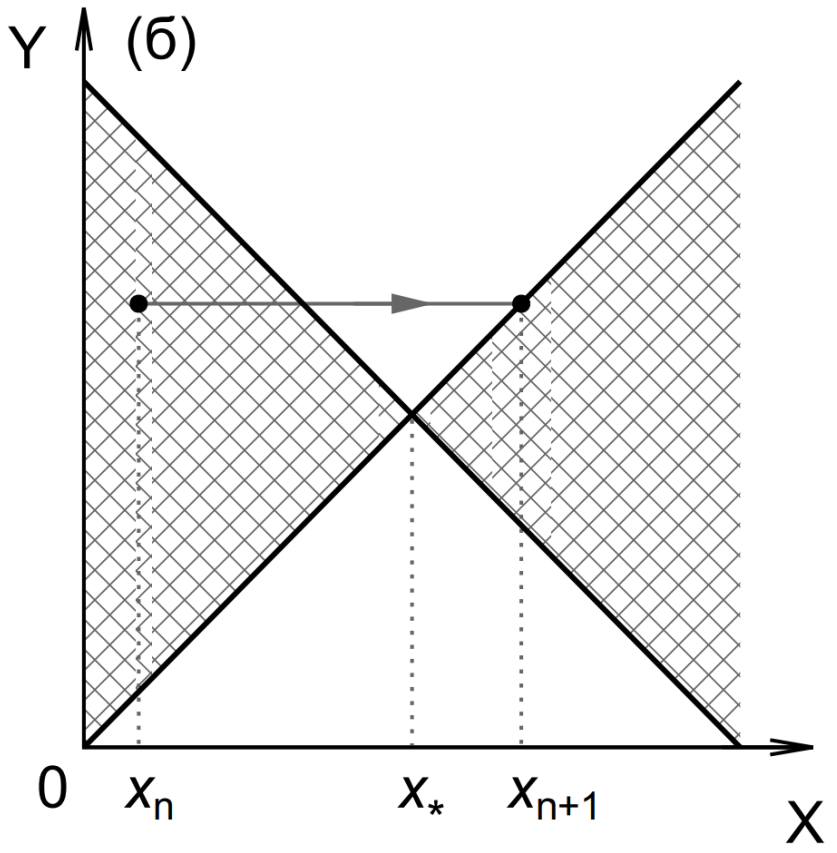
Заметим, что сформулированное вначале ограничение $|\phi'| \le q < 1$ является достаточным, но не необходимым условием сходимости итерационного процесса. Для сходимости итераций достаточно, чтобы отображение ϕ уменьшало длину произвольного отрезка, один из концов которого совпадает с искомым корнем x^*, при этом производная ϕ' может принимать сколь угодно большие значения или даже вовсе не существовать. Итерационный процесс будет сходиться при любом начальном приближении x_0, если график функции $\phi(x)$ лежит между прямыми $y_1 = x^* + q(x - x^*)$ и $y_2 = x^* - q(x - x^*)$, где $0 < q < 1$ — некоторая постоянная (рис. 1.2). На рисунке видно, что итерация $x_{n+1} = \phi(x_n)$ из любой точки $(x_n,\ \phi(x_n))$ внутри заштрихованных секторов улучшит приближение к корню: $|x_{n+1} - x^*| < |x_n - x^*|$.

Очевидно, многие функции не удовлетворяют даже более слабому условию, сформулированному в предыдущем абзаце.



Метод простых итераций

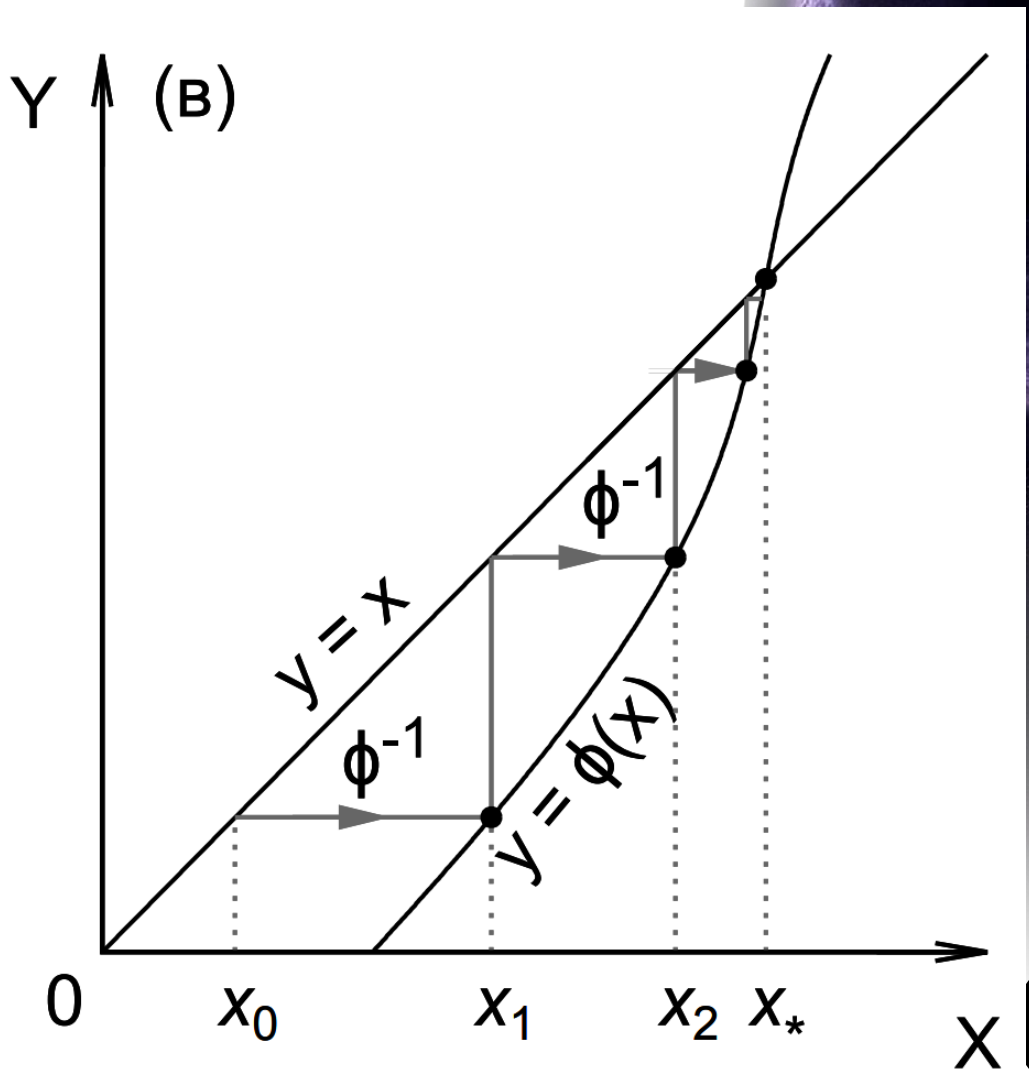
Заметим, что сформулированное вначале ограничение $|\phi'| \le q < 1$ является достаточным, но не необходимым условием сходимости итерационного процесса. Для сходимости итераций достаточно, чтобы отображение ϕ уменьшало длину произвольного отрезка, один из концов которого совпадает с искомым корнем x^*, при этом производная ϕ' может принимать сколь угодно большие значения или даже вовсе не существовать. Итерационный процесс будет сходиться при любом начальном приближении x_0, если график функции $\phi(x)$ лежит между прямыми $y_1 = x^* + q(x - x^*)$ и $y_2 = x^* - q(x - x^*)$, где $0 < q < 1$ — некоторая постоянная (рисунок). На рисунке видно, что итерация $x_{n+1} = \phi(x_n)$ из любой точки $(x_n, \phi(x_n))$ внутри заштрихованных секторов улучшит приближение к корню: $|x_{n+1} - x^*| < |x_n - x^*|$.



Очевидно, многие функции не удовлетворяют даже более слабому условию, сформулированному в предыдущем абзаце. Покажем две модификации итерационного процесса, позволяющие расширить область его применения.

Метод простых итераций

Первая возможность связана с переходом от функции ϕ к обратной функции ϕ^{-1}. В случае, если $|\phi'(x)| \geq q > 1$, можно перейти к обратным функциям в уравнении $\phi(x) = x$, в результате получим итерационный процесс $x_{n+1} = \phi^{-1}(x_n)$, который будет сходиться в силу соотношения $(\phi^{-1})' = 1/\phi'$. Геометрический смысл итераций для обратной функции показан на рисунке, стрелками обозначены вычисления функции $\phi^{-1}(x)$. Производная $\phi' > 1$, поэтому итерационный процесс $x_{n+1} = \phi(x_n)$ будет расходящимся. Для сходимости необходимо, по сути, двигаться в обратную сторону, что и достигается переходом к вычислению обратной функции $\phi^{-1}(x_n)$ вместо $\phi(x_n)$. Итерационный процесс для обратных функций целесообразно использовать в случае, когда обратная функция ϕ^{-1} может быть относительно легко вычислена, а $|\phi'(x^*)| >> 1$.

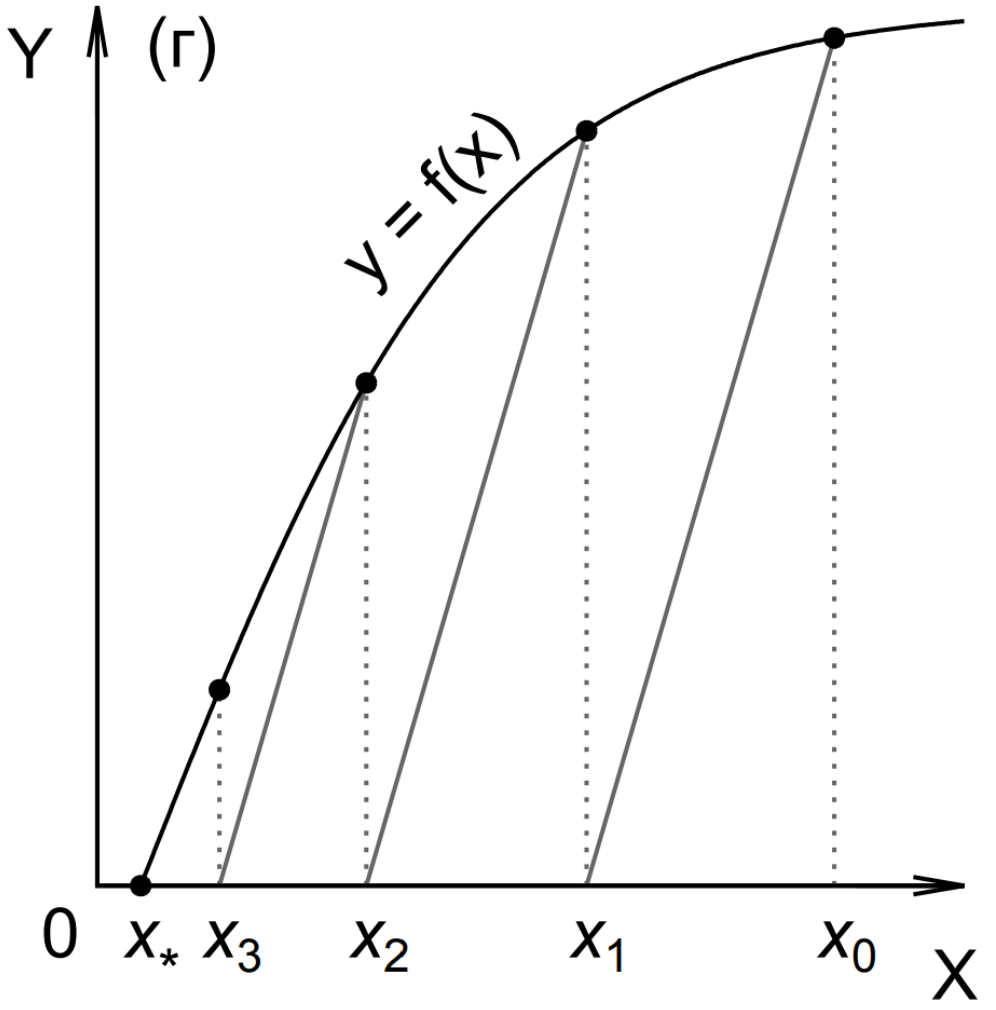


Метод простых итераций

Другая модификация метода итераций может быть получена введением корректирующего множителя. Вместо уравнения $x = x + f(x) = \phi(x)$ будем решать уравнение $x = x - \lambda f(x)$, множество решений которого, очевидно, совпадает с множеством корней уравнения $f(x) = 0$ при условии $\lambda = 0$. Пусть функция $f(x)$ непрерывно дифференцируема, так что $|f'(x)| < \infty$. Производная $\tilde{\phi}'(x) = 1 - \lambda f'(x)$ может быть сделана сколь угодно малой в точке $x = x^*$ соответствующим выбором λ, что обеспечит сходимость итерационного процесса

$$x_{n+1} = x_n - \lambda f(x_n)$$

для любой гладкой функции $f(x)$. Знак λ нужно выбирать равным знаку производной, а значение — по возможности ближе к обратному значению производной $1/f'(x^*)$. Геометрический смысл итерационного процесса показан на русунке.



Выход из цикла итераций можно осуществлять, проверяя условие малости изменения x на последней итерации: $|x_n - x_{n-1}| < \delta$, где δ — требуемая точность. Однако данная оценка может быть слишком грубой, особенно в случае $|\phi'(x^*)| \approx 1$.

Метод простых итераций

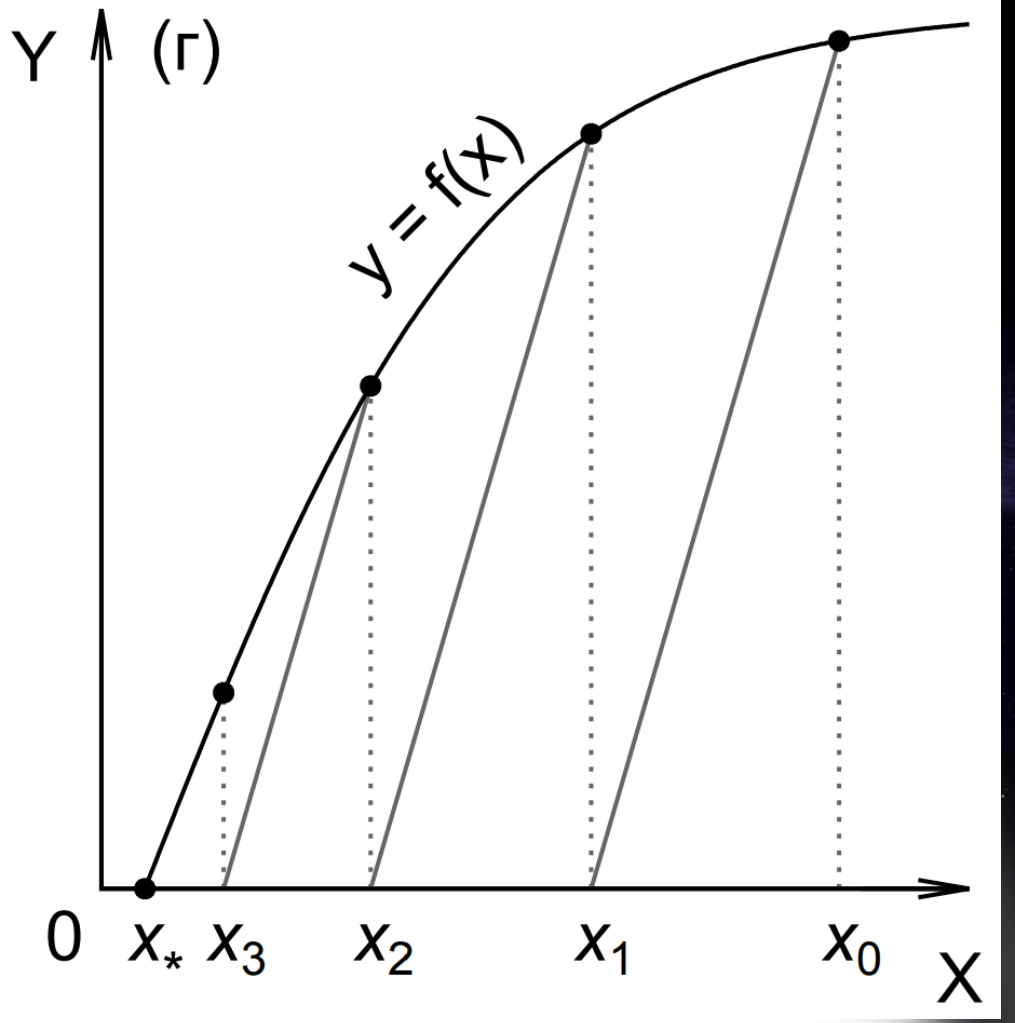
Для получения более точного критерия завершения итерационного процесса рассмотрим последовательность $\{x_k\}$, сходящуюся к корню x^*, полагая x_0 достаточно близким к x^*. Обозначая $R_n := x^* - x_n$, из $x_{n+1} = x_n - \lambda f(x_n)$ получаем

$$R_{n+1} \approx 1 - \lambda f'(x*) R_n \equiv qR_n,$$

т. е. итерации сходятся приблизительно как сумма геометрической прогрессии со знаменателем $q = (x_n - x_{n-1})/(x_{n-1} - x_{n-2})$. Итерационный процесс может быть остановлен, когда погрешность R_n текущего приближения к корню не превосходит требуемой точности вычислений δ. Поскольку $x_n - x_{n-1} = R_{n-1} - R_n \approx R_n/q - R_n$, приходим к условию

$$|R_n| \approx \left|\frac{x_n - x_{n-1}}{\frac{1}{q} - 1}\right| = \frac{(x_n - x_{n-1})^2}{|2x_{n-1} - x_n - x_{n-2}|} < \delta$$

Оценка для R_n в левой части неравенства может быть использована для повышения точности расчётов (сокращения числа итераций, необходимых для получения заданной точности).



Метод Ньютона — Рафсона

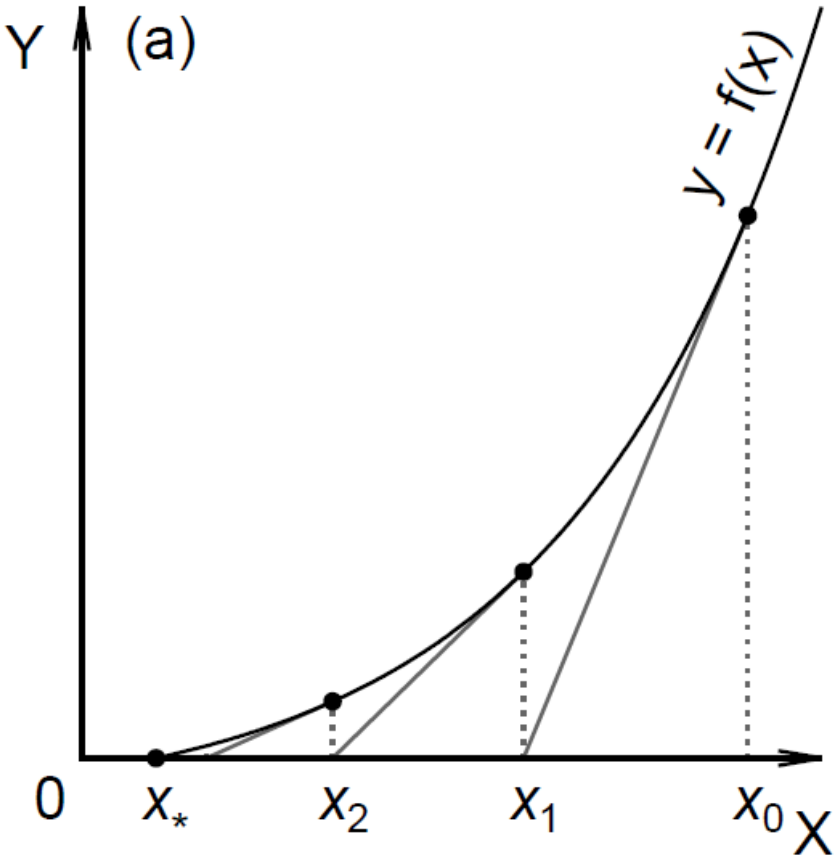
Геометрический смысл рассмотренного выше метода итераций состоит в приближении к искомому корню x^* по прямым с постоянным наклоном (рис. 1.4). Скорость сходимости можно существенно повысить, положив $\lambda(x) = 1/f'(x)$ с тем, чтобы наклон был равен производной функции $f(x)$ (рис. 1.5). При этом мы получим итерационный процесс

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

известный как метод Ньютона, или Ньютона — Рафсона. В выражении полезно увидеть также точное решение линеаризованной задачи $f'(x) = 0$ — это позволит нам впоследствии обобщить данный метод на матричные и операторные уравнения.

Исследуем скорость сходимости процесса к корню x^*. Для этого предположим, как и раньше, что нам задано достаточно хорошее приближение x_n, так что $f(x_n) \approx f'(x^*)\,(x_n - x^*) + 1\,f''(x^*)\,(x_n - x^*)^2$, а $f'(x_n) \approx f'(x^*) + f''(x^*)\,(x_n - x^*)$. Вычитая x^* из обеих частей уравнения, имеем:

$$R_{n+1} \approx R_n + \frac{f'(x_*)(-R_n) + \frac{1}{2}f''(x_*)(-R_n)^2}{f'(x_*) - R_n f''(x_*)}$$

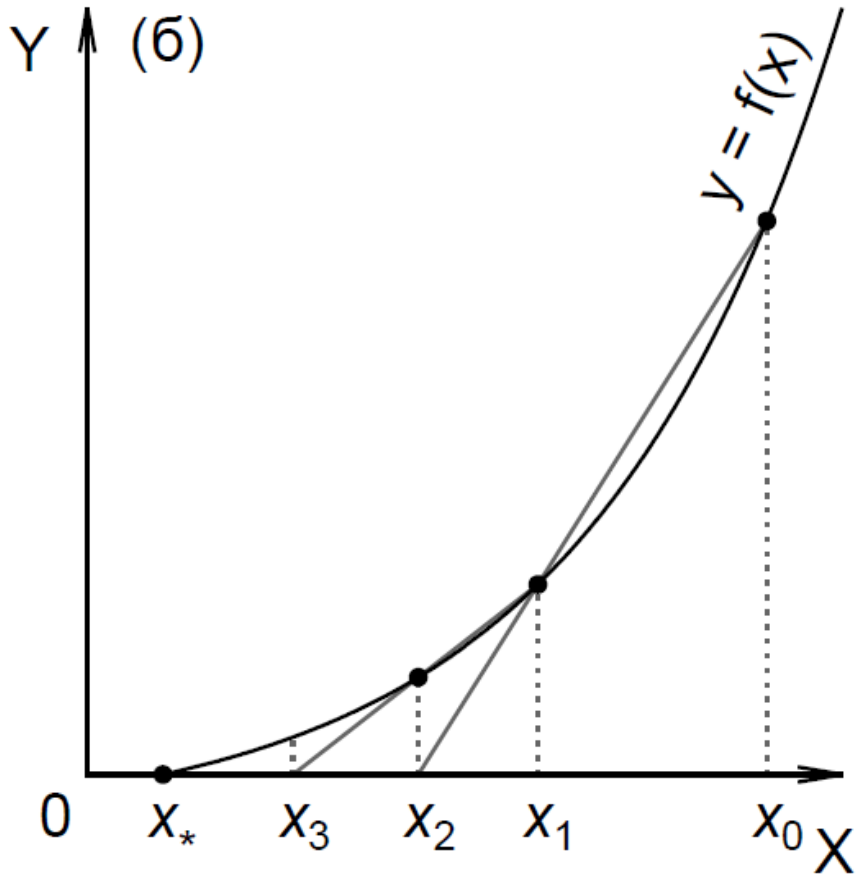


Метод Ньютона — Рафсона

$$R_{n+1} \approx R_n + \frac{f'(x_*)(-R_n) + \frac{1}{2}f''(x_*)(-R_n)^2}{f'(x_*) - R_n f''(x_*)}$$

Легко заметить, что метод Ньютона обладает более высоким порядком сходимости по сравнению с методом итераций. Так, если в методе итераций погрешность убывала приблизительно в геометрической прогрессии ($R_{n+1} \approx qR^1{}_n$), то в методе Ньютона погрешность на каждом следующем шаге оказывается квадратично мала по сравнению с погрешностью на предыдущем шаге, $R_{n+1} = \alpha R^2{}_n$.

Заметим, что выражение (12) допускает возможность расходящего- ся процесса $R_{n+1} > R_n$ при достаточно больших R_n, тогда как при достаточно малых Rn процесс монотонно сходится. Другими словами, в методе Ньютона может существовать область притяжения к корню: в случае, если начальное приближение x_0 лежит в области $a < x_0 < b$, так что x^* (a, b), процесс сходится к корню x^*, в противном случае итерации расходятся (либо сходятся к другому корню, отличному от x^* и лежащему вне отрезка $[a, b]$).



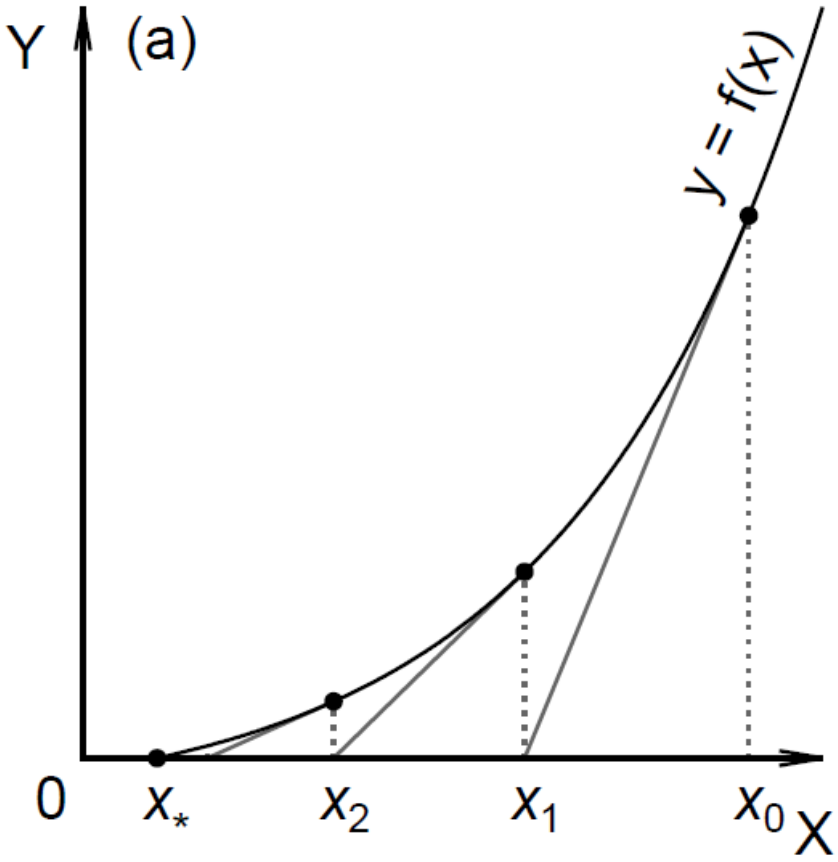
Метод секущих

Метод Ньютона обладает вторым порядком сходимости, однако для его реализации необходимо вычисление на каждом шаге значения функции $f(x_n)$ и её первой производной $f'(x_n)$. Зачастую, однако, производная функции $f\ (x)$ неизвестна (например, $f(x)$ измеряется в эксперименте или является результатом численного решения некоторой сложной задачи). Кроме того, даже если имеется выражение, позволяющее вычислить $f'(x_n)$, такое вычисление требует дополнительного времени на каждом шаге, что может сделать более выгодным использование других методов.

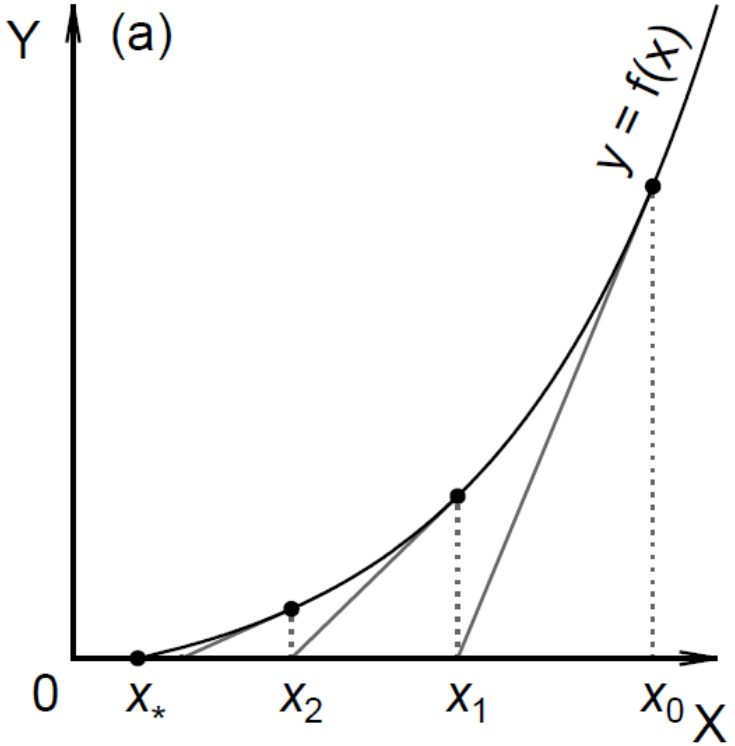
Например, вместо $f'(x)$ при вычислении корректирующего множителя $\lambda = 1/f'(x)$ можно использовать оценки значения производной, полученные с использованием асимптотических разложений $f(x)$ или вычисленные с помощью разделённой разности по результатам двух последних вычислений $f\ (x)$:

$$x_{x+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, n = 1,2, \ldots$$

Итерационный процесс является двухшаговым: для нахождения следующего приближения x_{n+1} необходимо знать два предыдущих.



Метод секущих



Исследуем скорость сходимости процесса. Полагая x_n и x_{n-1} достаточно близкими к корню x^* и заменяя $f(x) \approx f'(x^*)(x - x^*) + 1/2\, f''(x^*)(x - x^*)^2$, имеем:

$$R_{n+1} \approx R_n - R_n \frac{1 - aR_n}{1 - a(R_n + R_{n-1})}, R_n = x_* - x_n, a = \frac{f''(x_*)}{f'(x_*)}$$

Удерживая члены не выше второго порядка по R_n и R_{n-1}, получаем:

$$R_{n+1} \approx aR_nR_{n-1}$$

Попробуем найти степенной закон убывания погрешности на каждой итерации. Подставляя $|R_{n+1}| = a^{\alpha}|R_n|^{\beta}$, получаем два условия на показатели α и β: $\alpha\beta = 1$, $\beta^2 - \beta - 1 = 0$. Для сходимости итерационного процесса необходимо $\beta > 0$, откуда $\beta = \frac{1}{2}(1 + \sqrt{5}) \approx 1.618$

Порядок процесса получился дробным — метод секущих сходится быстрее (по числу шагов) метода простых итераций, но медленнее метода Ньютона. Заметим, однако, что в методе Ньютона на каждом шаге необходимо вычислять значение функции $f(x_n)$ и её первой производной $f'(x_n)$. Полагая, что для вычисления f и f' требуется приблизительно одинаковое количество арифметических операций, получим, что на каждые два вычисления функции приходится один шаг метода Ньютона, дающий (приблизительно) удвоение числа значащих цифр после запятой для корня, $|R_{n+1}| \propto |R_n|^2$. Для сравнения, двукратное вычисление значения функции f позволяет сделать два шага методом секущих, что даёт улучшение точности $|R_{n+2}| \propto |R_n|^{\beta^2}$. Поскольку $\beta^2 \approx 2{,}618$, метод секущих может сходиться быстрее метода Ньютона, если сравнивать не по числу итераций, но по затратам машинного времени, необходимого для нахождения корня с заданной точностью.

Многомерное обобщение

Метод Ньютона очевидным образом может быть обобщён на многомерный случай. Пусть задана система уравнений f (x) = 0, где f = $\{f_1, \ldots, f_n\}$, x = $\{x_1, \ldots, x_n\}$. Выбрав некоторое начальное приближение $x^{(0)} = \{x_1^{(0)}, \ldots, x_n^{(0)}\}$ к корню $x^{(*)} = \{x_1^{(*)}, \ldots, x_n^{(*)}\}$, построим последовательность $x^{(n)}$, n = 1, 2, . . ., сходящуюся к искомому корню. Для этого, как и в одномерном случае, линеаризуем уравнения f(x) = 0. Используя тензорные обозначения и предполагая суммирование по повторяющимся индексам, имеем:

$$0 = f\left(x^{(n)} + \delta x^{(n)}\right) \approx f_i(x^{(n)}) + J_{ij}^{(n)} \delta x_j^{(n)}, \text{где} J_{ij}^{(n)} = \left.\frac{\partial f_i}{\partial x_j}\right|_{x=x^{(n)}}$$

откуда получаем для $x^{(n+1)} = x^{(n)} + \delta x^{(n)}$:

$$x_i^{(n+1)} = x_i^{(n)} - \left[\left(J^{(n)}\right)^{-1}\right]_{ij} f_i(x^{(n)})$$

Поиск комплексных корней

В некоторых задачах может возникнуть необходимость поиска нулей функции на комплексной плоскости. Пусть $f\ (z)$ — аналитическая функция в области $\mathcal{D}$, и $|f(z)| < \infty\ \forall z \in \mathcal{D}$. Разложим $f(z)$ и её производную $f'(z)$ в ряд в точке $z_0 \in \mathcal{D}$:

$$f\ (z) = \sum_{k=n}^{\infty} \frac{c_k}{k!}\ (z\ -\ z_0)^k\ , f'(x) = \sum_{k=1}^{\infty} \frac{c_k}{(k-1)!}\ (z\ -\ z_0)^{k-1}\ , c_k = f^{(k)}(z_0)$$

Если z_0 — нуль функции f, то $c_0 = 0$ и $f'/f = (z - z_0)^{-1} + O(|z - z_0|^0)$. Если z_0 — нуль кратности n, то

$$f\ (z) = \sum_{k=n}^{\infty} \frac{c_k}{k!}\ (z\ -\ z_0)^k\ , \frac{f'(\mathrm{z})}{f(\mathrm{z})}\ =\ \ \frac{\mathrm{n}}{z - z_0} + \mathcal{O}\ (|\mathrm{z}\ -\ \mathrm{z}_0|^0).$$

Используя интегральную формулу Коши, имеем $\oint_\gamma \frac{f'(\mathrm{z})}{f(\mathrm{z})} dz = 2\pi i \sum_i n_j = 2\pi i N$

С точностью до множителя $2\pi i$ получили количество нулей функции $f(z)$ внутри контура γ с учетом их кратности. Стратегия поиска комплексных корней $f\ (z)$ с использованием может быть следующей. Выбираем контур интегрирования γ так, чтобы внутрь него попадало 1-2 нуля функции $f\ (z)$. Допустим, для примера, $\sum n_j = 2$. Далее вычисляем интегралы вида

$$I_1 = \frac{1}{2\pi i} \oint_\gamma \frac{f'(\mathrm{z})}{f(\mathrm{z})} z dz = \sum_i z_j n_j = z_1 + z_2; \quad I_2 = \frac{1}{2\pi i} \oint_\gamma \frac{f'(\mathrm{z})}{f(\mathrm{z})} z^2 dz = \sum_i z_j^2 n_j = z_1^2 + z_2^2$$

Решение (z_1, z_2) полученной системы уравнений даёт нам решение исходной задачи о нахождении комплексных корней $f(z)$.

Заметим, что задача нахождения комплексных корней может быть решена с использованием рассмотренных выше методов Ньютона, секущих или метода итераций. Эти методы могут быть гораздо более эффективны в случае, если известно достаточно хорошее начальное приближение к искомому корню.

Сравнение методов

Какой из рассмотренных выше методов лучше? Однозначного ответа на этот вопрос нет — оптимальный метод зависит от условий поставленной задачи. Метод дихотомии обеспечивает сравнительно невысокую, но при этом гарантированную скорость сходимости для корней нечётной кратности, тогда как сходимость других рассмотренных выше методов может быть условной. В этой связи метод дихотомии может быть использован для получения грубого приближения к корню с последующим его уточнением каким-либо другим методом, имеющим более высокую скорость сходимости. Метод Ньютона эффективен, подходит в том числе для корней с чётной кратностью, но требует вычисления производной. Метод секущих также очень эффективен, однако может приводить к «разболтке» и обеспечивать более низкую точность, особенно в случае кратных корней. Метод простых итераций исключительно прост в реализации и при этом может обеспечивать быструю сходимость к корню, особенно в случае $|\phi'(x^*)| << 1$.

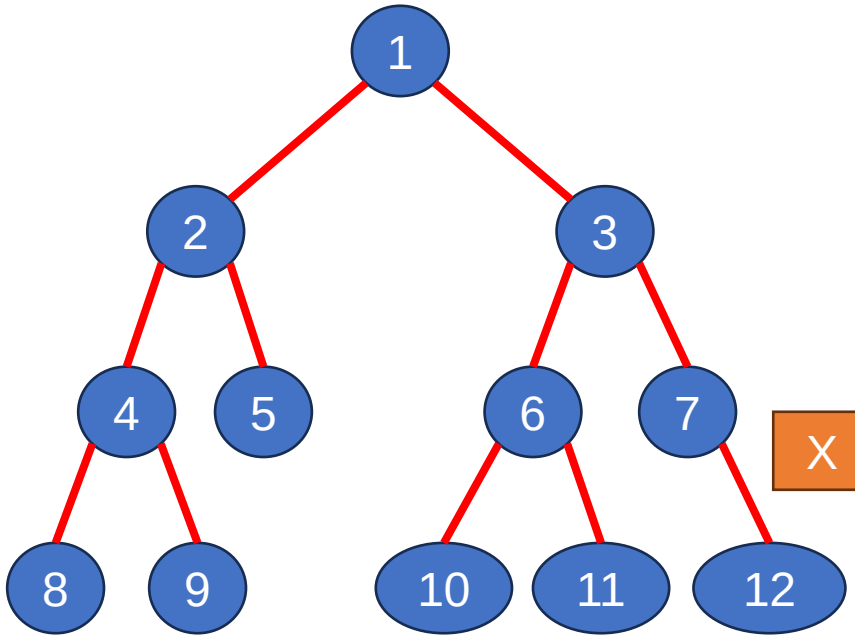
Пожалуй, одним из оптимальных решений в случае, когда неизвестны производные функции, является метод Брента, позволяющий отчасти объединить достоинства дихотомии и метода Ньютона. Следующее приближение строится по трём начальным точкам с использованием квадратичной интерполяции $x(y)$, а в тех случаях, когда найденное с помощью интерполяции приближение не обеспечивает сходимости, применяется бисекция. Такой подход позволяет одновременно достичь гарантированной сходимости при относительно высокой скорости, не уступающей методу дихотомии на самых «плохих» функциях, и значительно ускорить сходимость в «хороших» случаях.

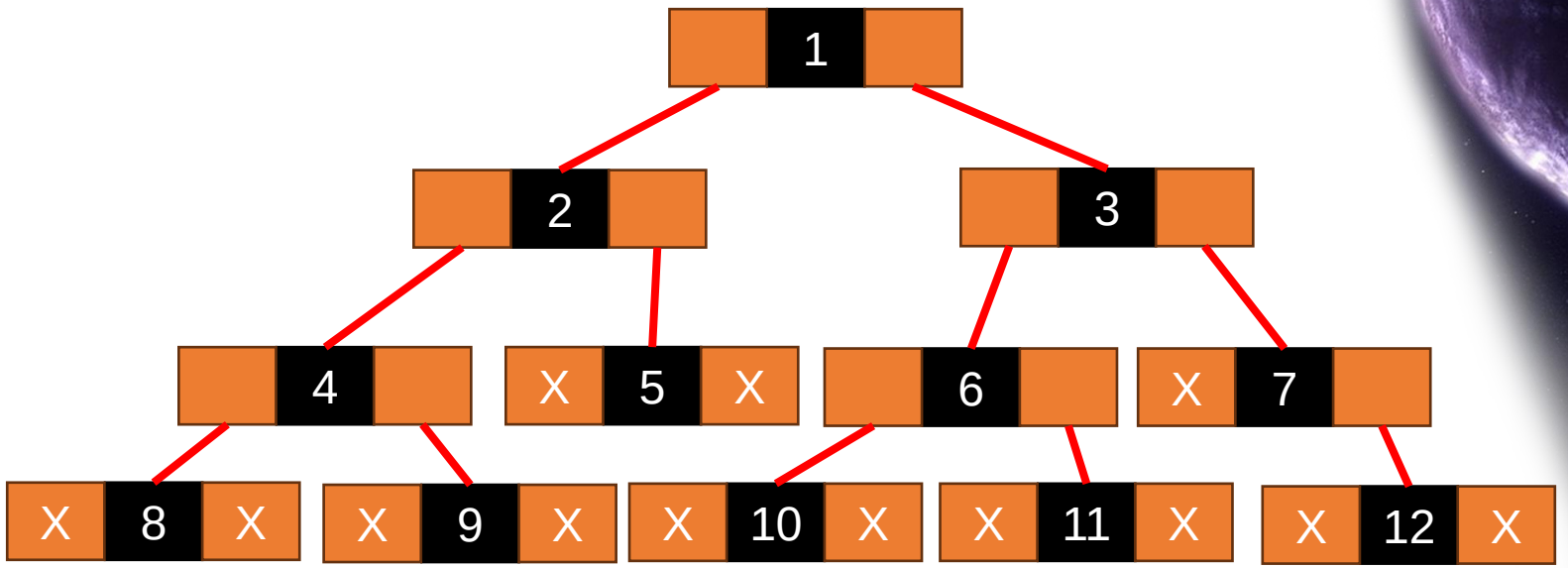
План лекции

Решение конечных уравнений	**Эффективные деревья**	B-деревья и приложения (1 часть)
20 минут	70 минут	0 минут

Связные двоичные деревья

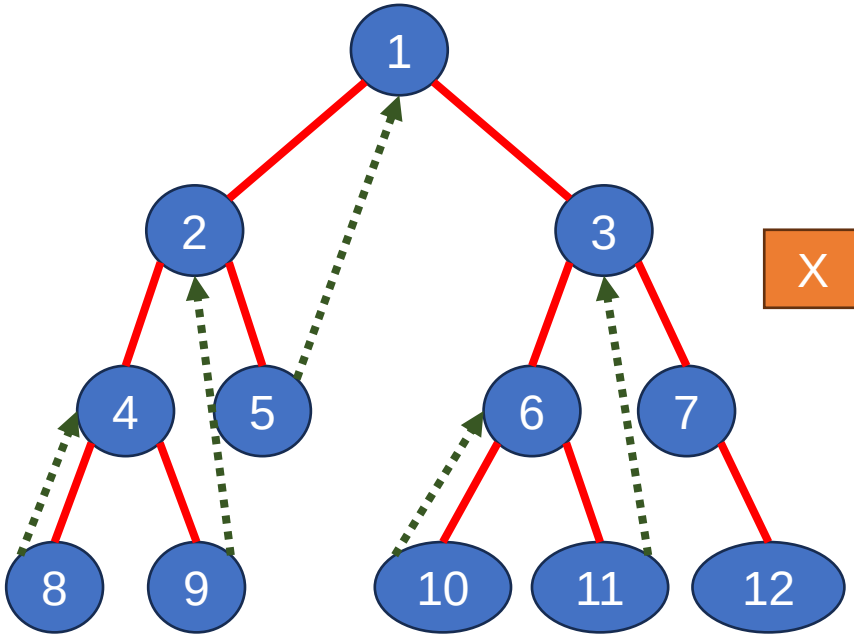
Связное двоичное дерево такое же, как и двоичное дерево, но с разницей в хранении указателей NULL.

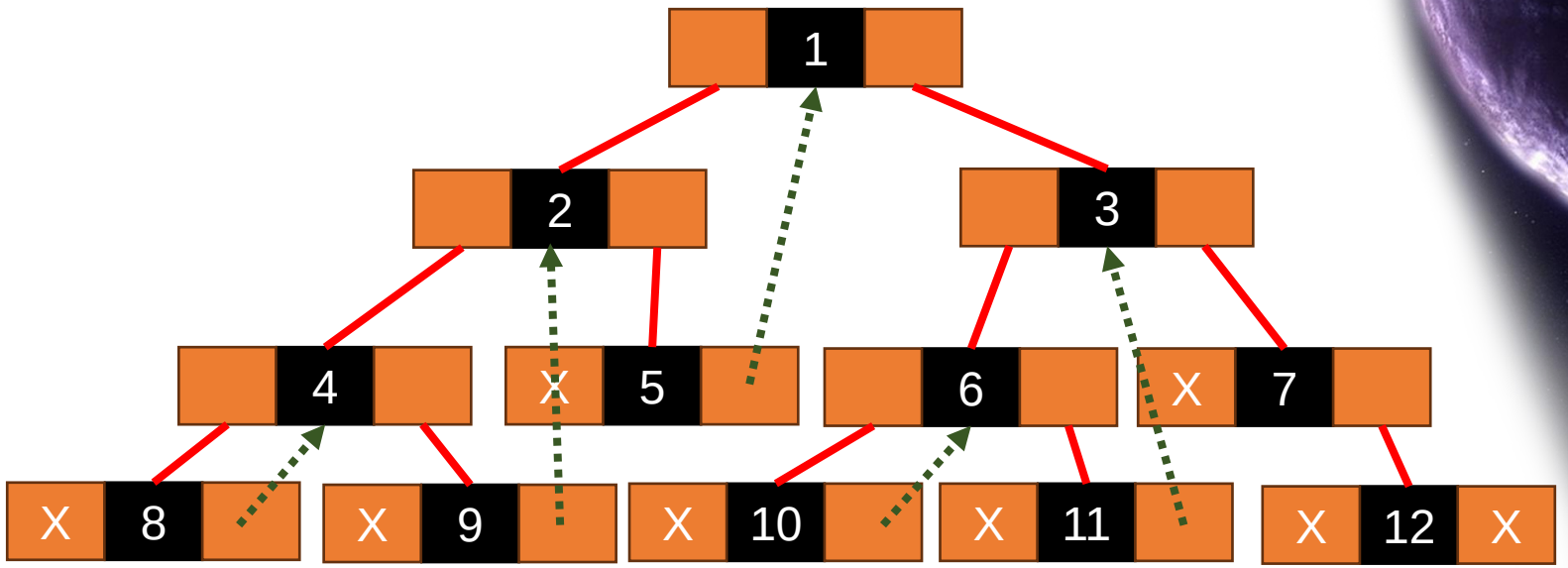




Одностороннее связывание

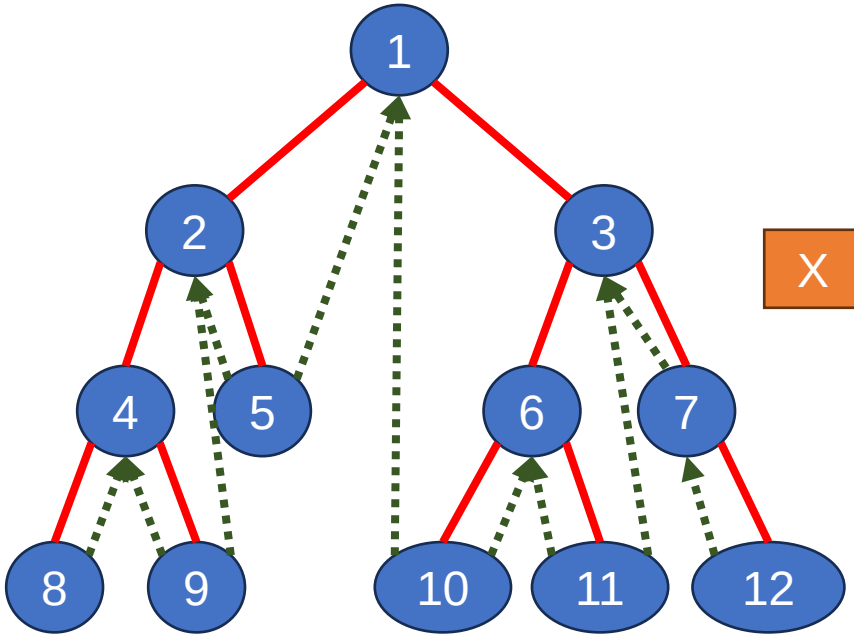
Связное двоичное дерево такое же, как и двоичное дерево, но с разницей в хранении указателей NULL.

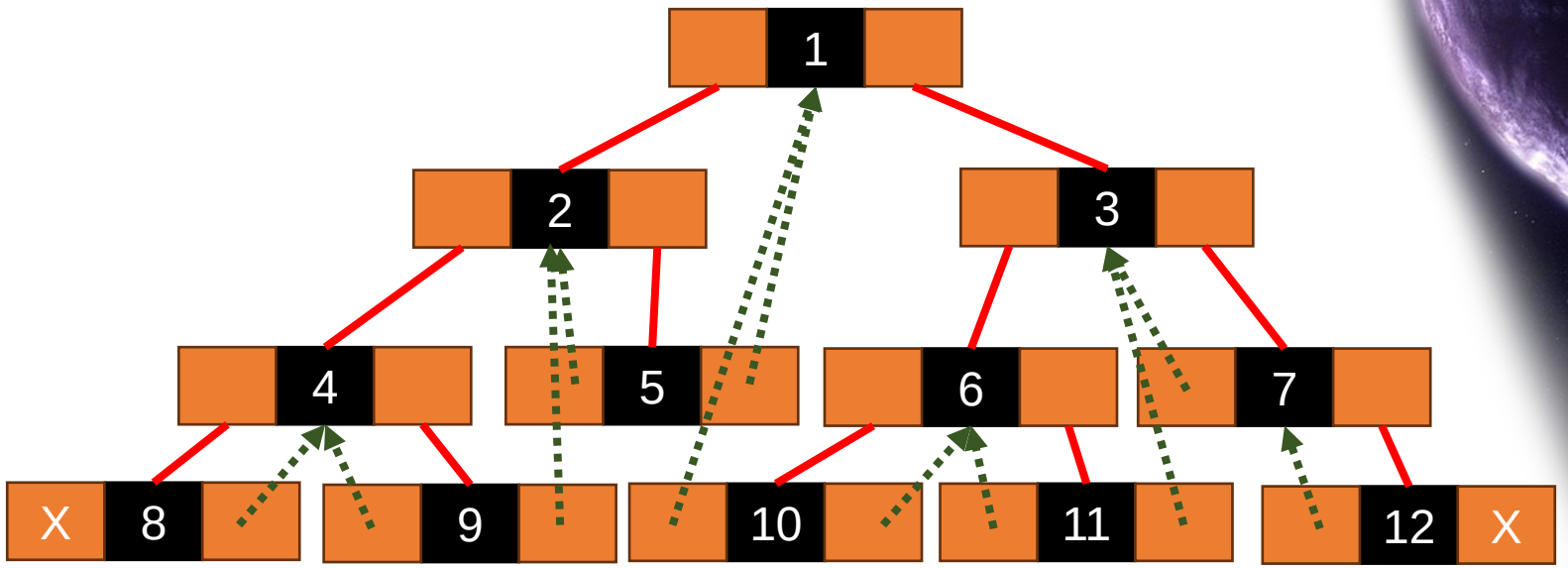




Двухстороннее связывание

Связное двоичное дерево такое же, как и двоичное дерево, но с разницей в хранении указателей NULL.



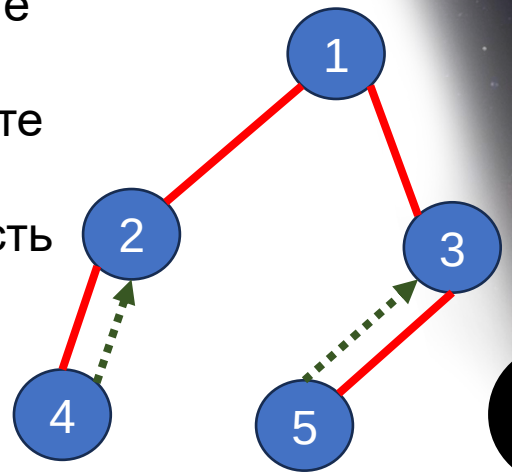


Обход связного бинарного дерева

Для каждого узла сначала посетите левое поддерево, если оно существует и не было посещено ранее. Затем за самим узлом (корнем) следует посещение его правого поддерева (если оно существует). В случае отсутствия правого поддерева проверьте наличие нитевидной ссылки и сделайте нитевидный узел текущим рассматриваемым узлом.
Шаг 1: проверьте, есть ли у текущего узла левый дочерний узел, который не был посещен. Если левый дочерний узел существует, который не был посещен, перейдите к шагу 2, в противном случае перейдите к шагу 3.
Шаг 2: добавьте левого дочернего узла в список посещенных узлов. Сделайте его текущим узлом и перейдите к шагу 6.
Шаг 3: Если текущий узел имеет правого потомка, перейдите к шагу 4, иначе перейдите к шагу 5.
Шаг 4: Сделайте этого правого потомка текущим узлом и перейдите к шагу 6. Шаг 5: выведите узел и, если есть узел с нитью, сделайте его текущим узлом.
Шаг 6: Если все узлы посещены, то END, иначе перейдите к шагу 1.

Обход связного бинарного дерева

1. Узел 1 имеет левого потомка, т. е. 2, который не был посещен. Поэтому добавьте 2 в список посещенных узлов, сделайте его текущим узлом.
2. Узел 2 имеет левого потомка, т. е. 4, который не был посещен. Итак, добавьте 4 в список посещенных узлов, сделайте его текущим узлом.
3. Узел 4 не имеет левого или правого потомка, поэтому выведите 4 и проверьте его связанную ссылку. У него есть связанная ссылка на узел 2, поэтому сделайте узел 2 текущим узлом.
4. Узел 2 имеет левого потомка, который уже был посещен. Однако у него нет правого потомка. Теперь выведите 2 и перейдите по его связанной ссылке на узел 1. Сделайте узел 1 текущим узлом.
5. Узел 1 имеет левого потомка, который уже был посещен. Поэтому выведите 1. Узел 1 имеет правого потомка 3, который еще не был посещен, поэтому сделайте его текущим узлом.
6. Узел 3 имеет левого потомка (узел 5), который не был посещен, поэтому сделайте его текущим узлом.
7. Узел 5 не имеет левого или правого потомка. Итак, выведите 5. Однако у него есть нить ссылки, указывающая на узел 3. Сделайте узел 3 текущим узлом.
8. Узел 3 имеет левого потомка, который уже был посещен. Итак, выведите 3.
9. Теперь не осталось ни одного узла, поэтому мы заканчиваем здесь. Последовательность выведенных узлов — 4 2 1 5 3.



Преимущества связного двоичного дерева

- ✓ Он обеспечивает линейный обход элементов в дереве.
- ✓ Линейный обход исключает использование стеков, которые, в свою очередь, потребляют много памяти и машинного времени.
- ✓ Он позволяет находить родителя заданного элемента без явного использования родительских указателей.
- ✓ Поскольку узлы содержат указатели на упорядоченных предшественника и преемника, нитевидное дерево обеспечивает прямой и обратный обход узлов, как задано упорядоченным способом.

Таким образом, мы видим, что основное различие между двоичным деревом и потоковым двоичным деревом заключается в том, что в двоичных деревьях узел хранит указатель NULL, если у него нет потомка, и поэтому нет возможности вернуться назад.

Дерева AVL

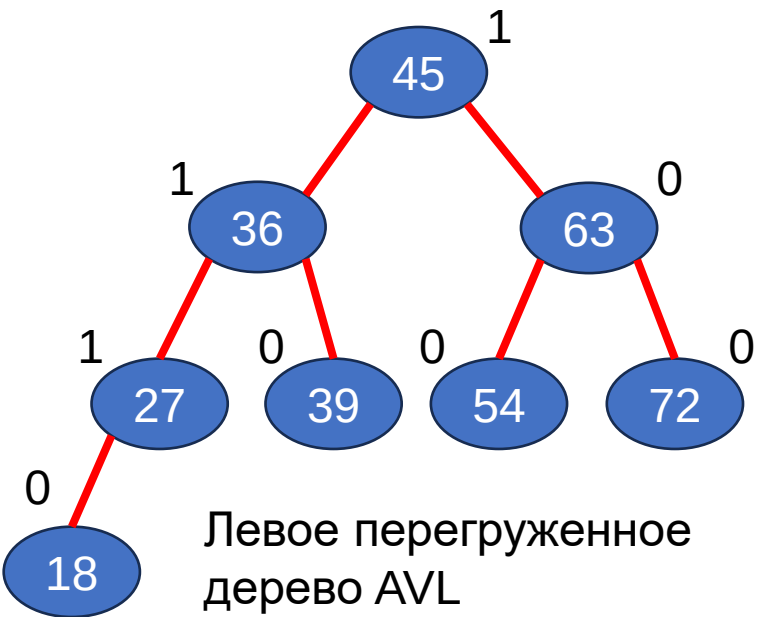
Дерево AVL — это самобалансирующееся бинарное дерево поиска, изобретенное Г. М. Адельсоном-Вельски и Э. М. Ландисом в 1962 году. Дерево названо AVL в честь своих изобретателей. В дереве AVL высоты двух поддеревьев узла могут отличаться не более чем на единицу. Благодаря этому свойству дерево AVL также известно как дерево со сбалансированной высотой. Главное преимущество использования дерева AVL заключается в том, что для выполнения операций поиска, вставки и удаления требуется время O(log n) как в среднем, так и в худшем случае, поскольку высота дерева ограничена O(log n). Структура дерева AVL такая же, как и у бинарного дерева поиска, но с небольшим отличием. В своей структуре оно хранит дополнительную переменную, называемую BalanceFactor. Таким образом, с каждым узлом связан фактор сбалансированности. Коэффициент балансировки узла вычисляется путем вычитания высоты его правого поддерева из высоты его левого поддерева. Двоичное дерево поиска, в котором каждый узел имеет коэффициент балансировки –1, 0 или 1, называется сбалансированным по высоте. Узел с любым другим коэффициентом балансировки считается несбалансированным и требует повторной балансировки дерева.

Коэффициент балансировки = Высота (левого поддерева) – Высота (правого поддерева)

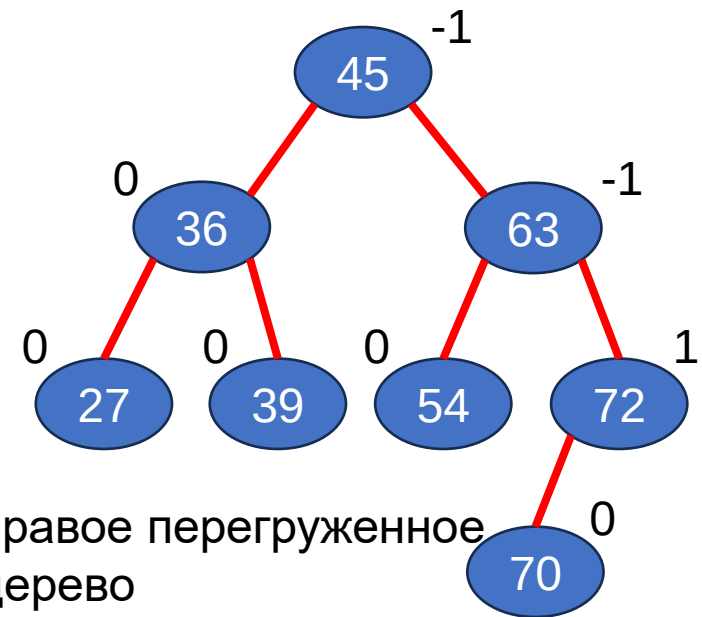
- Если коэффициент балансировки узла равен 1, то это означает, что левое поддерево дерева находится на один уровень выше, чем правое поддерево. Поэтому такое дерево называется деревом с перегрузкой влево.
- Если коэффициент балансировки узла равен 0, то это означает, что высота левого поддерева (самый длинный путь в левом поддереве) равна высоте правого поддерева.
- Если фактор баланса узла равен –1, то это означает, что левое поддерево дерева находится на один уровень ниже, чем правое поддерево. Такое дерево поэтому называется деревом с перегрузкой справа.

Дерева AVL

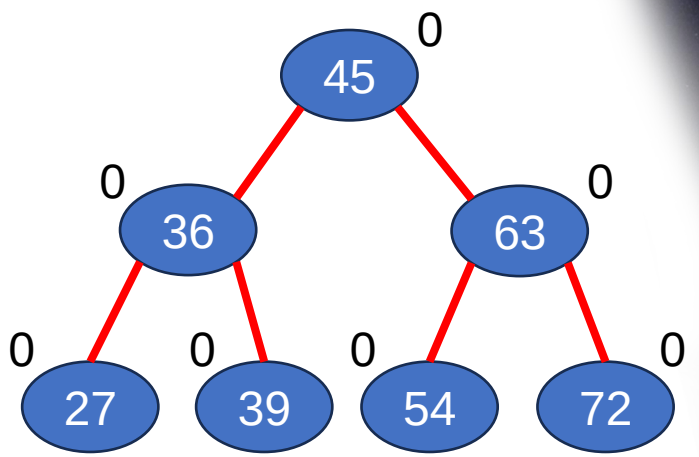
Деревья, представленные на рисунках, являются типичными кандидатами на деревья AVL, поскольку коэффициент балансировки каждого узла равен 1, 0 или –1. Однако вставки и удаления из дерева AVL могут нарушить коэффициент балансировки узлов, и, таким образом, может потребоваться повторная балансировка дерева. Дерево перебалансируется путем выполнения поворота в критическом узле. Существует четыре типа поворотов: поворот LL, поворот RR, поворот LR и поворот RL. Тип поворота, который необходимо выполнить, будет зависеть от конкретной ситуации.



Левое перегруженное дерево AVL



правое перегруженное дерево



сбалансированное дерево

Поиск узла в дереве AVL

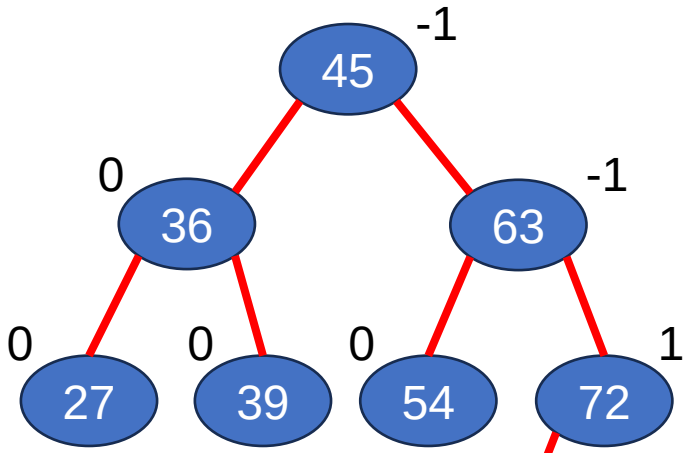
Поиск в дереве AVL выполняется точно так же, как и в бинарном дереве поиска. Из-за балансировки высоты дерева операция поиска занимает время O(log n). Поскольку операция не изменяет структуру дерева, никаких специальных положений не требуется.

Вставка нового узла в дерево AVL

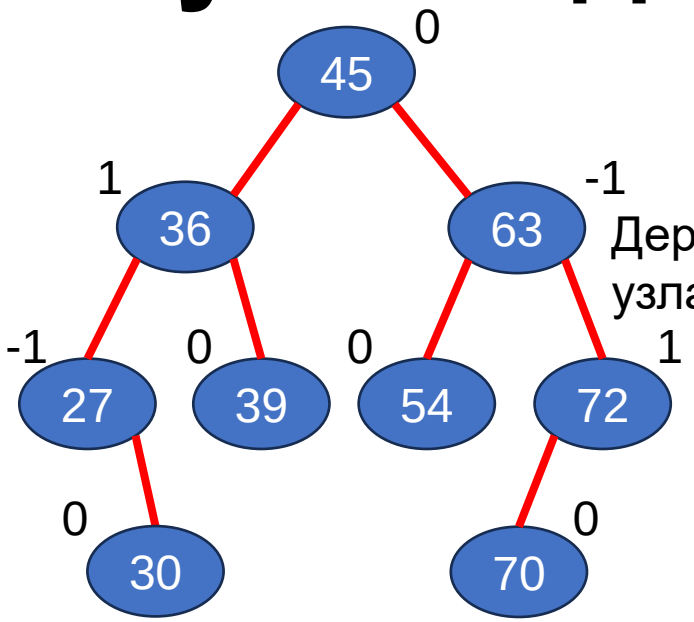
Вставка в дерево AVL также выполняется так же, как и в бинарное дерево поиска. В дереве AVL новый узел всегда вставляется как конечный узел. Но за шагом вставки обычно следует дополнительный шаг поворота. Поворот выполняется для восстановления баланса дерева. Однако, если вставка нового узла не нарушает фактор баланса, то есть если фактор баланса каждого узла по-прежнему равен –1, 0 или 1, то повороты не требуются. Во время вставки новый узел вставляется как конечный узел, поэтому он всегда будет иметь фактор баланса, равный нулю. Единственными узлами, коэффициенты балансировки которых изменятся, являются те, которые лежат на пути между корнем дерева и вновь вставленным узлом. Возможные изменения, которые могут произойти в любом узле на пути, следующие:

- Изначально узел был либо лево-, либо право-тяжелым, а после вставки он становится сбалансированным.
- Изначально узел был сбалансированным, а после вставки он становится либо лево-, либо право-тяжелым.
- Изначально узел был тяжелым (либо левым, либо правым), а новый узел был вставлен в тяжелое поддерево, тем самым создавая несбалансированное поддерево. Такой узел называется ***критическим узлом***.

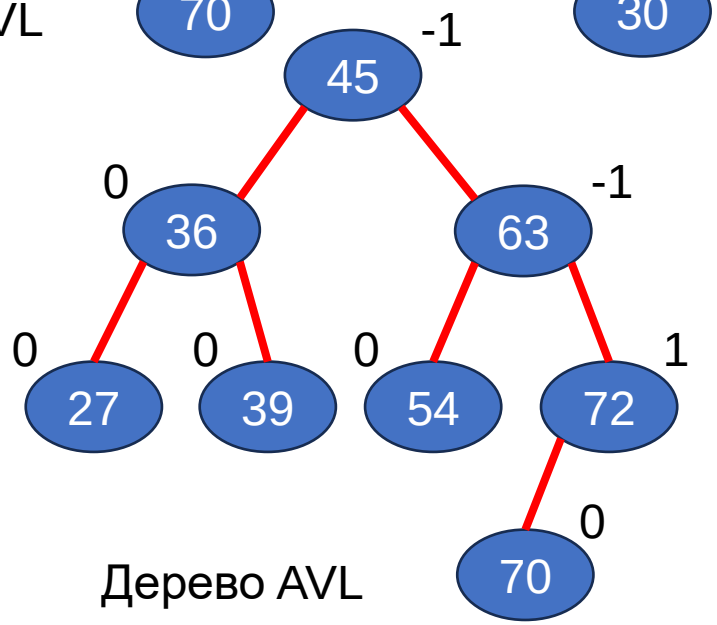
Вставка нового узла в дерево AVL



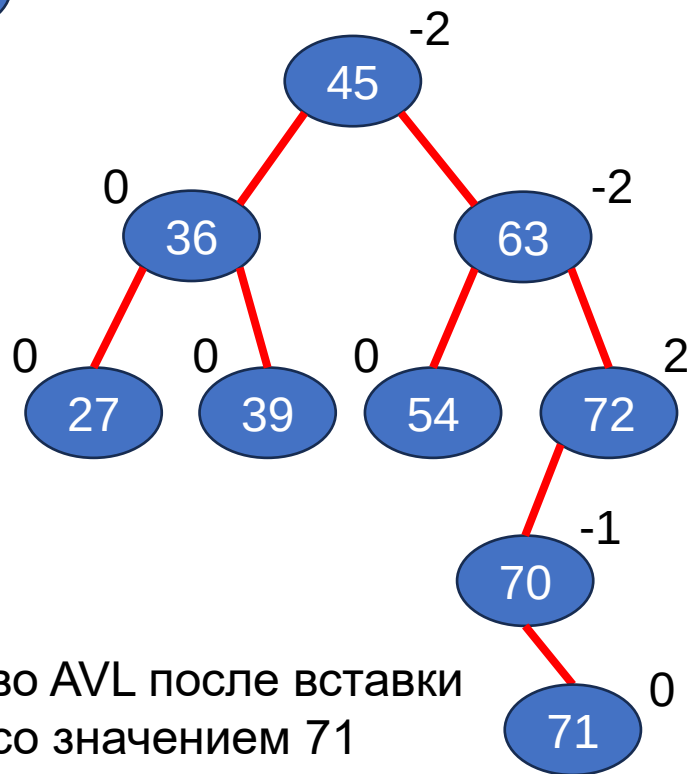
Дерево AVL



Дерево AVL после вставки узла со значением 30



Дерево AVL



Дерево AVL после вставки узла со значением 71

Вставка нового узла в дерево AVL

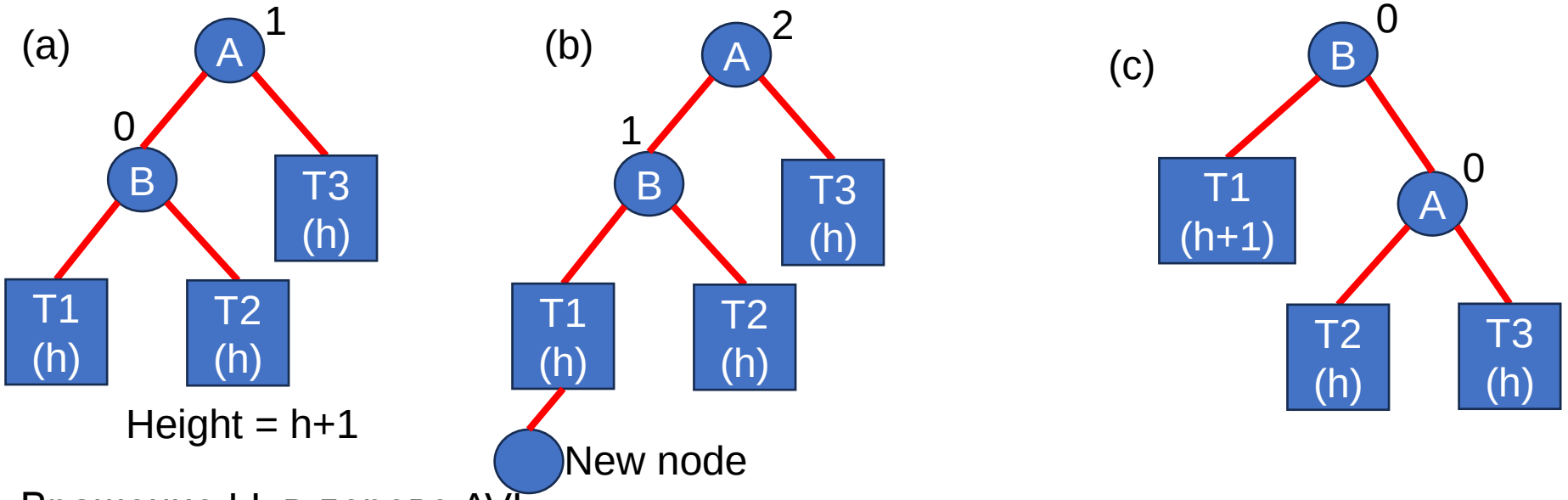
Итак, возникает необходимость выполнить поворот. Для выполнения поворота наша первая задача — найти критический узел. Критический узел — это ближайший узел-предок на пути от вставленного узла к корню, коэффициент балансировки которого не равен ни –1, ни 0, ни 1.

Вторая задача при перебалансировке дерева — определить, какой тип поворота необходимо выполнить. Существует четыре типа поворотов перебалансировки, и применение этих поворотов зависит от положения вставленного узла по отношению к критическому узлу. Четыре категории поворотов:

- ***поворот LL*** Новый узел вставляется в левое поддерево левого поддерева критического узла.
- ***поворот RR*** Новый узел вставляется в правое поддерево правого поддерева критического узла.
- ***поворот LR*** Новый узел вставляется в правое поддерево левого поддерева критического узла.
- ***вращение RL*** Новый узел вставляется в левое поддерево правого поддерева критического узла.

Вращение LL

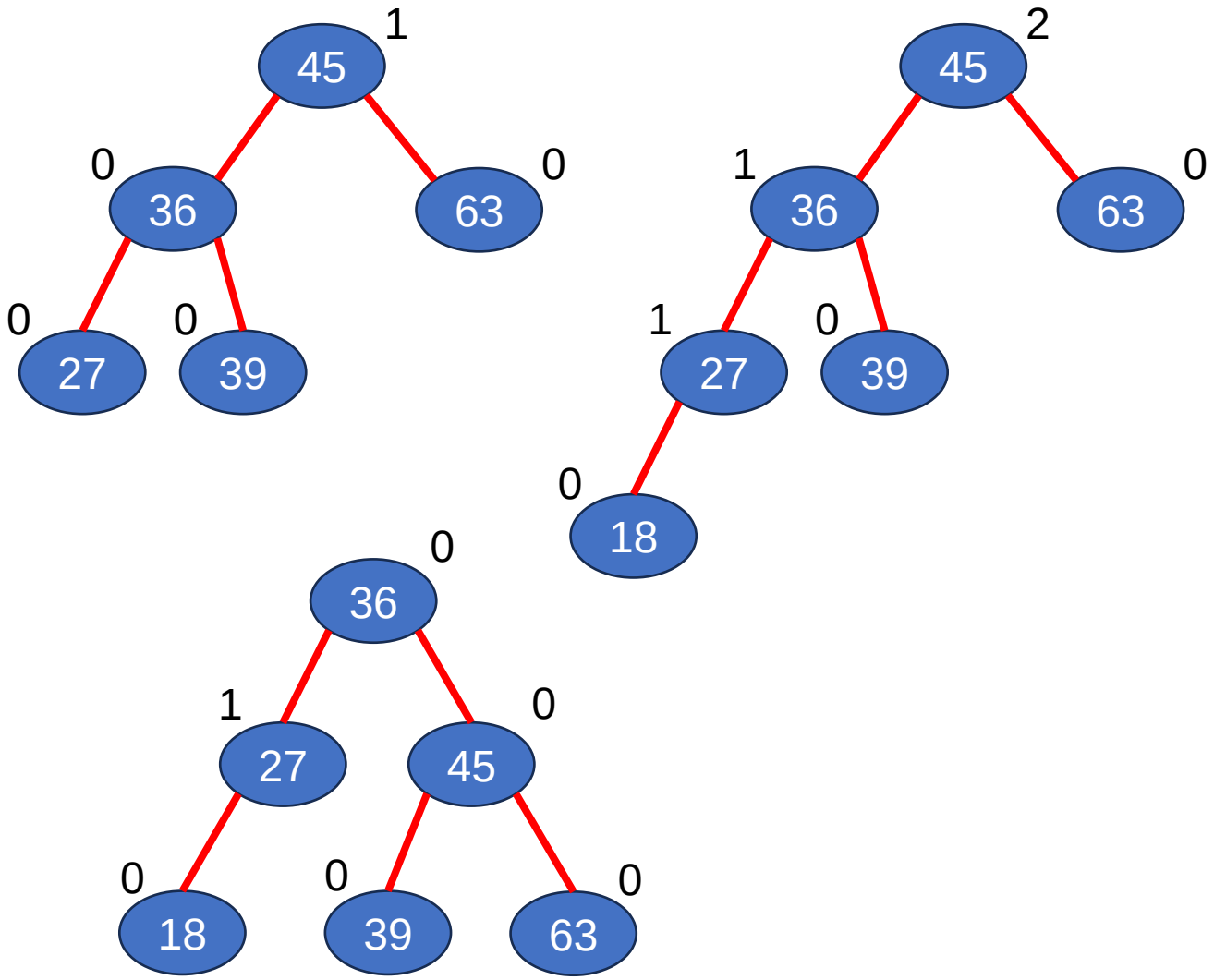
Давайте подробно изучим каждое из этих вращений. Сначала мы увидим, где и как применяется вращение LL. Рассмотрим дерево, приведенное на рисунке, которое показывает дерево AVL. Дерево (a) является деревом AVL. В дереве (b) новый узел вставляется в левое поддерево левого поддерева критического узла A (узел A является критическим узлом, поскольку он является ближайшим предком, коэффициент баланса которого не равен –1, 0 или 1), поэтому мы применяем вращение LL, как показано на дереве (c).



Вращение LL в дереве AVL

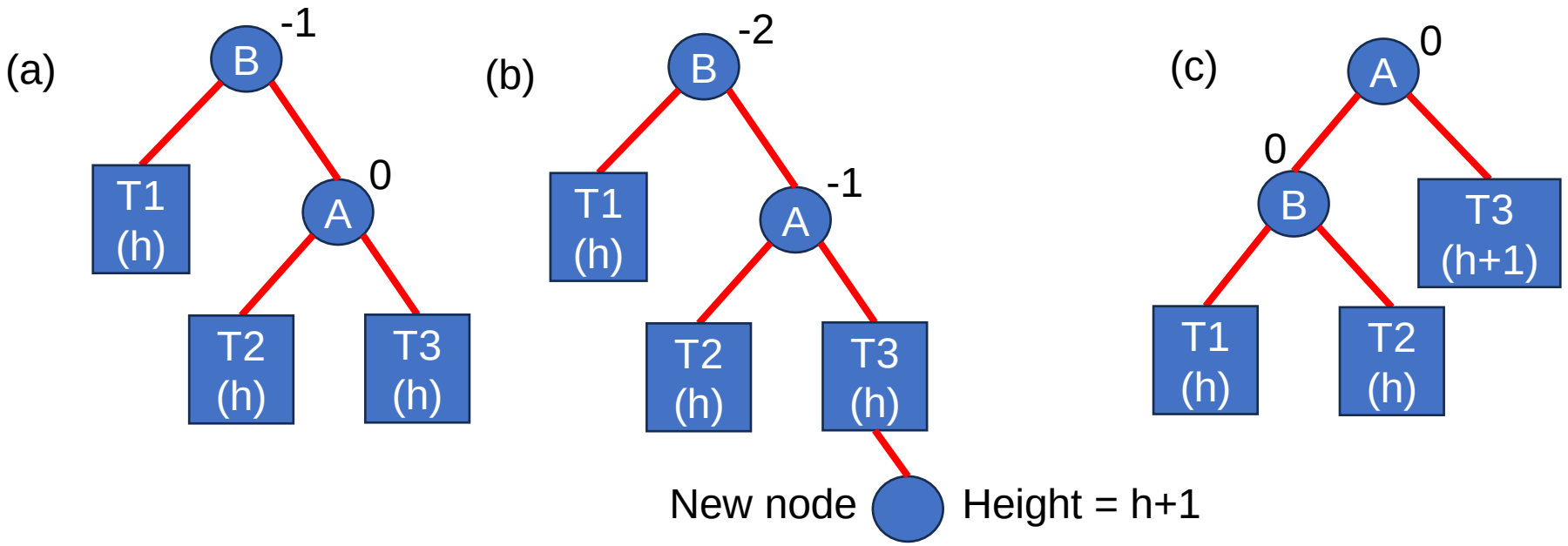
Во время вращения узел B становится корнем, а T1 и A являются его левым и правым потомками. T2 и T3 становятся левым и правым поддеревьями A.

Вращение LL - Пример

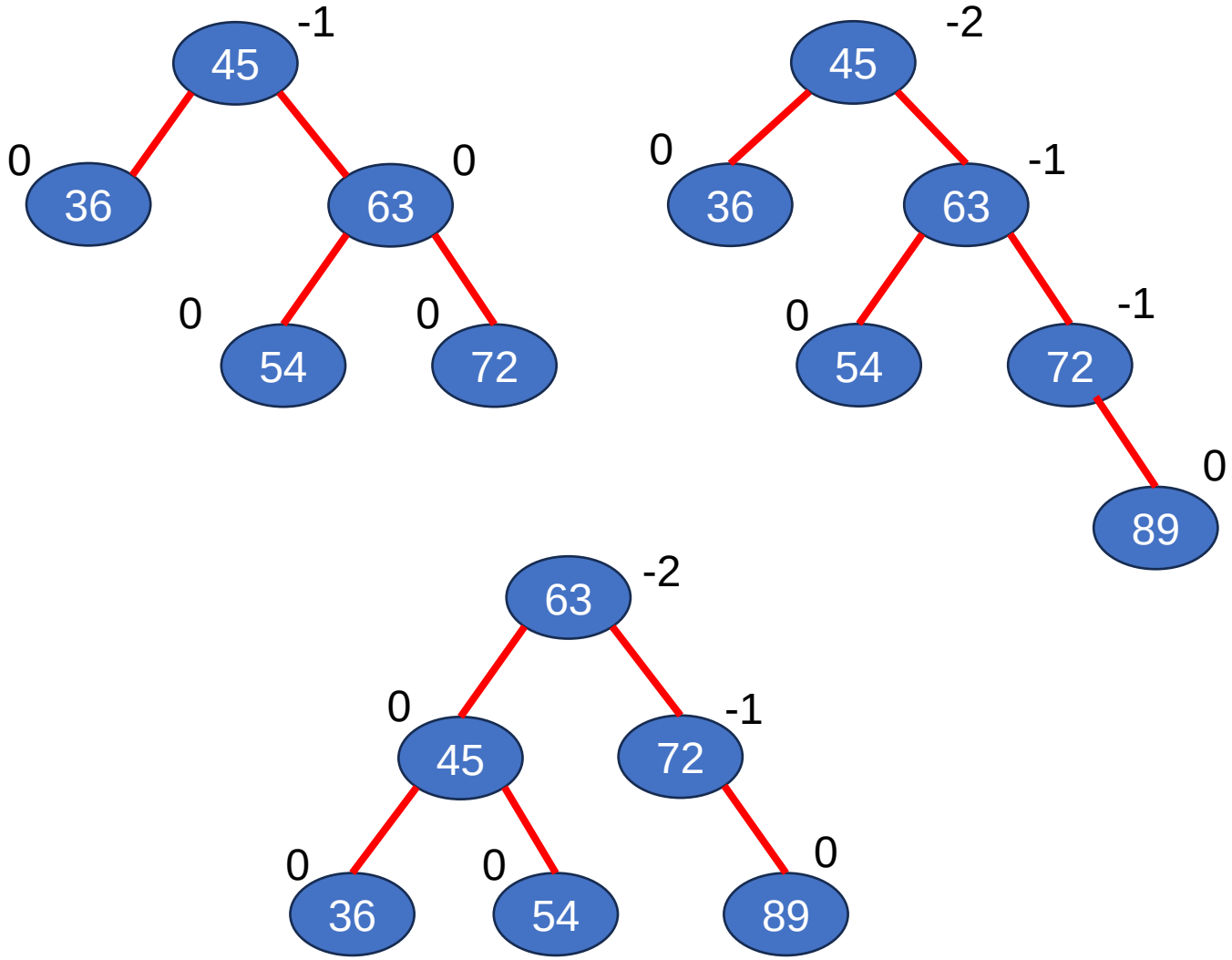


Вращение RR

Дерево (a) является деревом AVL. В дереве (b) новый узел вставляется в правое поддерево правого поддерева критического узла A (узел A является критическим узлом, поскольку он является ближайшим предком, коэффициент баланса которого не равен –1, 0 или 1), поэтому мы применяем вращение RR, как показано в дереве (c). Обратите внимание, что новый узел теперь стал частью дерева T3. Во время вращения узел B становится корнем, а A и T3 — его левым и правым потомками. T1 и T2 становятся левым и правым поддеревьями A.

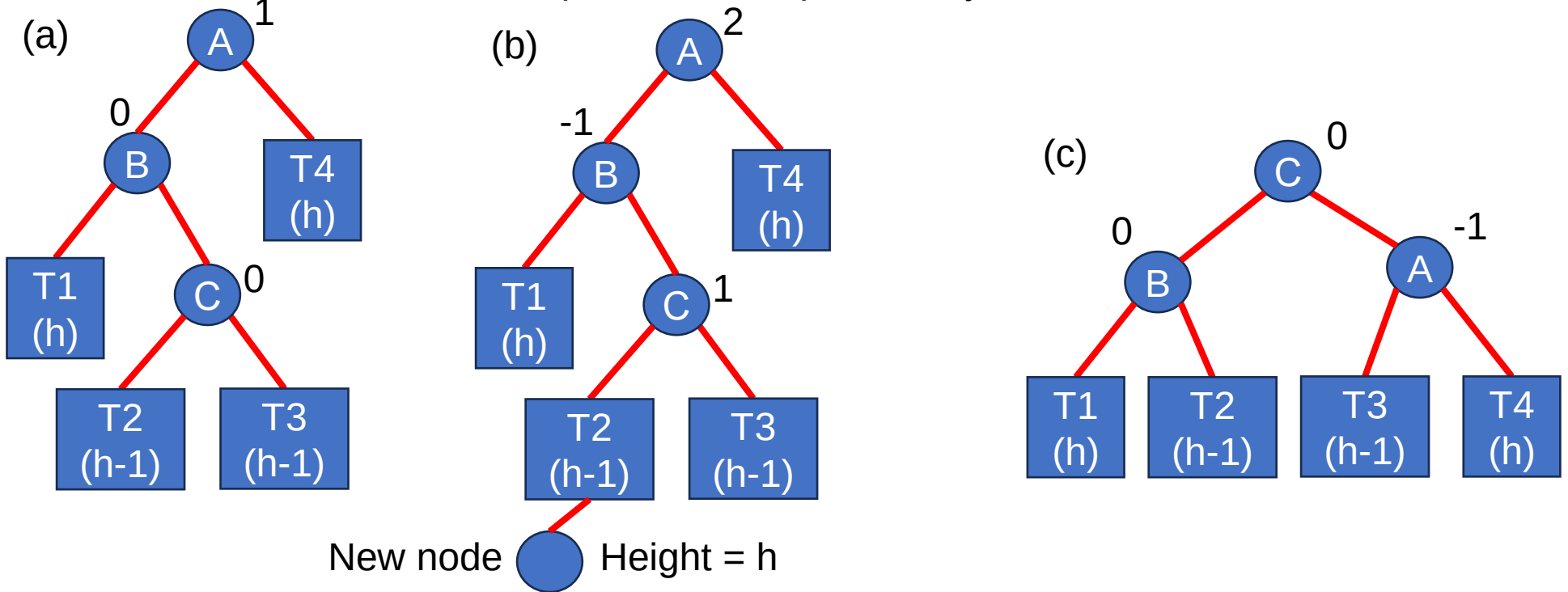


Вращение RR - Пример

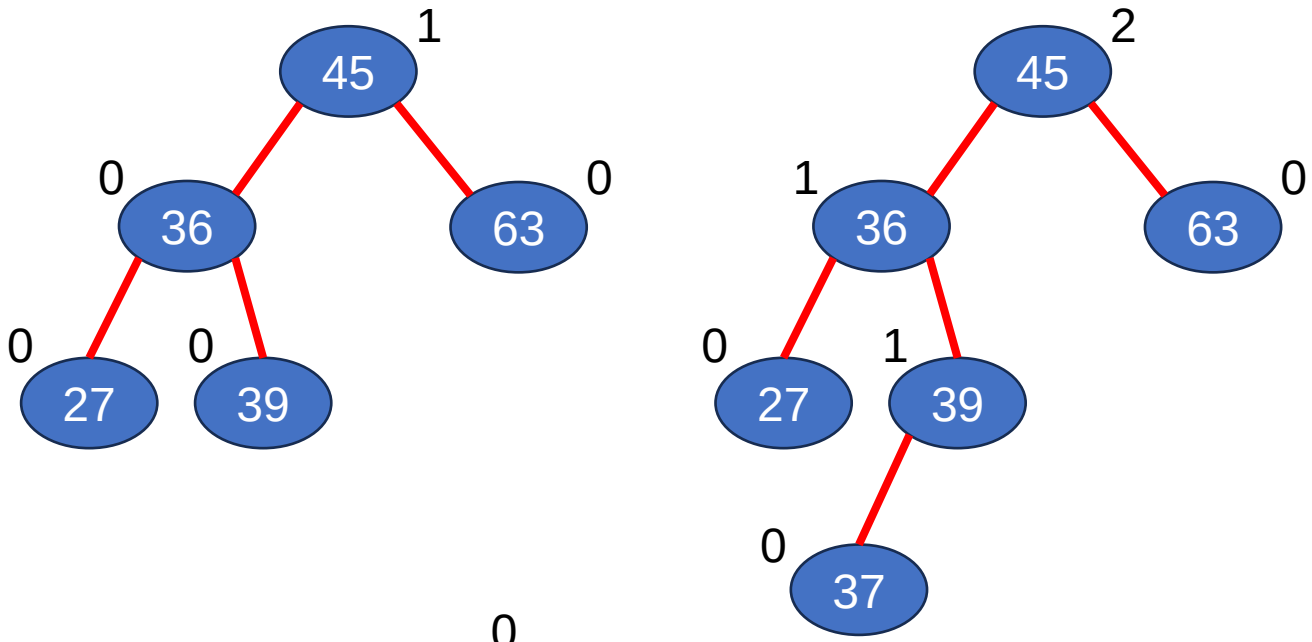


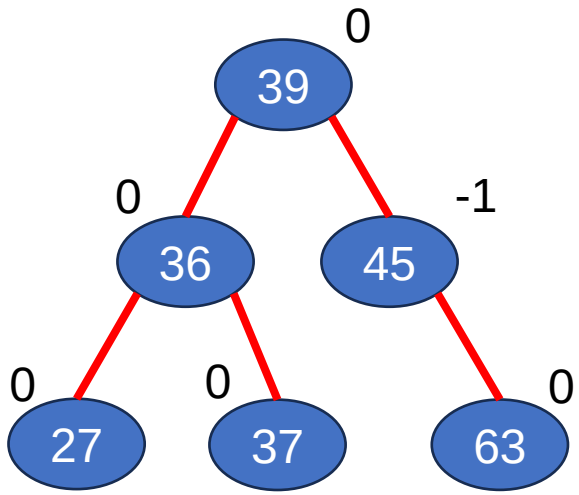
Повороты LR

Дерево (a) является деревом AVL. В дереве (b) новый узел вставляется в правое поддерево левого поддерева критического узла A (узел A является критическим узлом, поскольку он является ближайшим предком, коэффициент балансировки которого не равен –1, 0 или 1), поэтому мы применяем поворот LR, как показано в дереве (c). Обратите внимание, что новый узел теперь стал частью дерева T2. Во время поворота узел C становится корнем, а B и A — его левым и правым потомками. Узел B имеет T1 и T2 в качестве своих левых и правых поддеревьев, а T3 и T4 становятся левым и правым поддеревьями узла A.



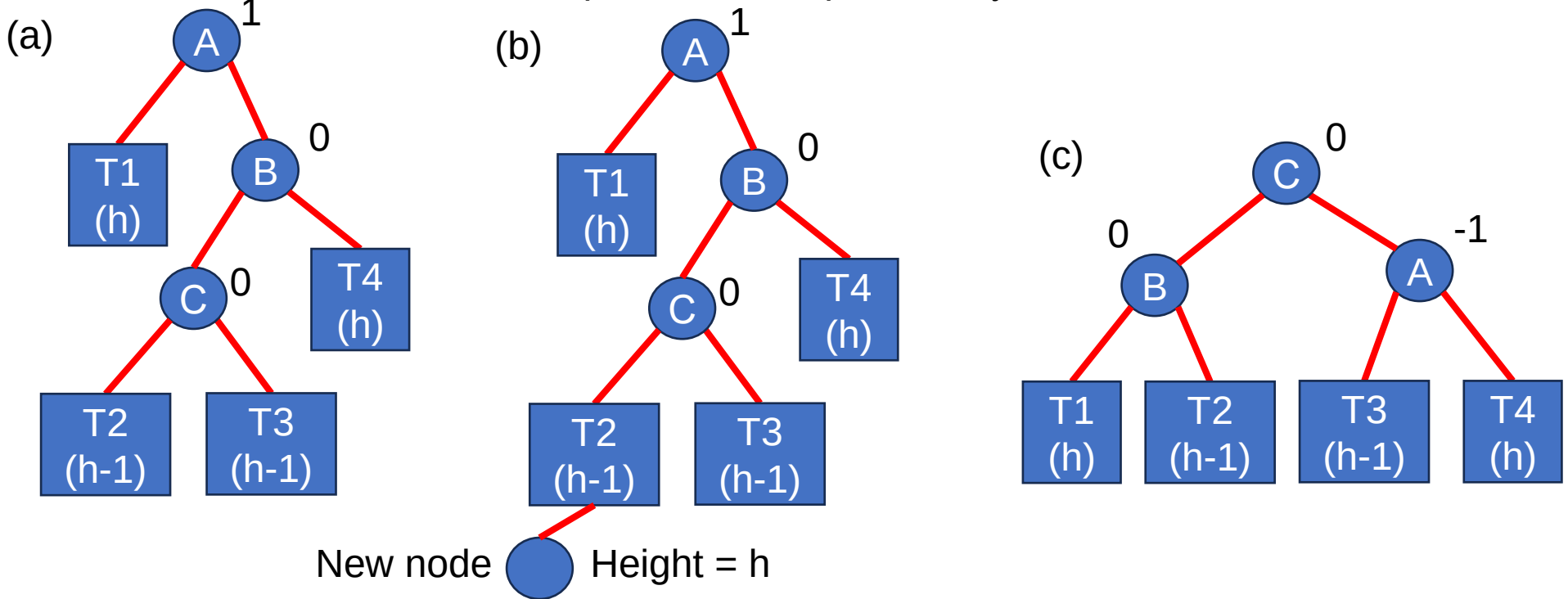
Повороты LR - Пример





Повороты RL

Дерево (a) является деревом AVL. В дереве (b) новый узел вставляется в левое поддерево правого поддерева критического узла A (узел A является критическим узлом, поскольку он является ближайшим предком, коэффициент балансировки которого не равен –1, 0 или 1), поэтому мы применяем вращение RL, как показано в дереве (c). Обратите внимание, что новый узел теперь стал частью дерева T2. Во время вращения узел C становится корнем, а A и B — его левым и правым потомками. Узел A имеет T1 и T2 в качестве своих левого и правого поддеревьев, а T3 и T4 становятся левым и правым поддеревьями узла B.

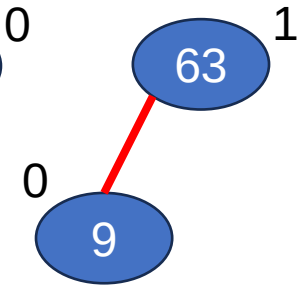


Повороты – общий пример

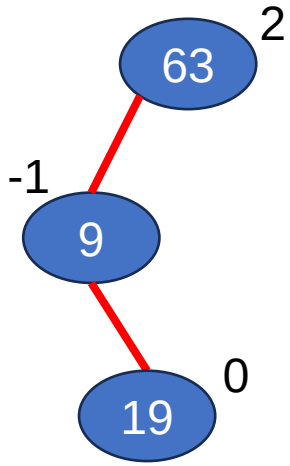
Построить дерево AVL, вставив следующие элементы в указанном порядке.
63, 9, 19, 27, 18, 108, 99, 81.

Шаг 1

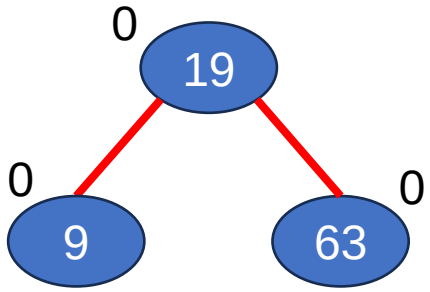
Шаг 2



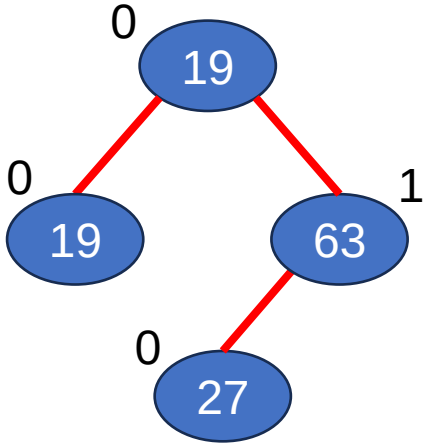
Шаг 3



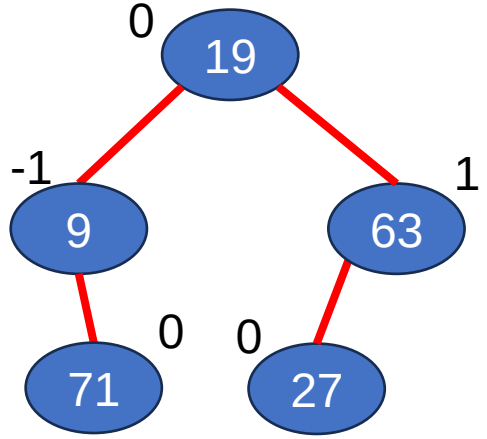
Шаг 4 – После LR поворота



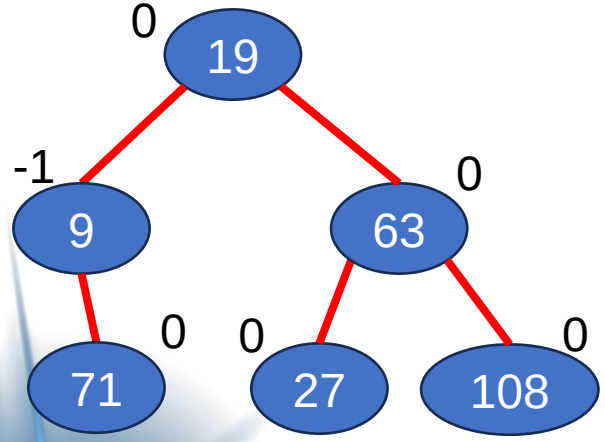
Шаг 5



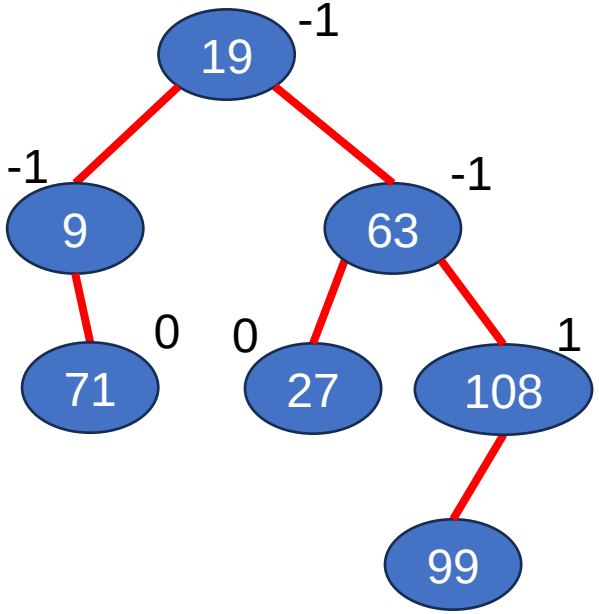
Шаг 6



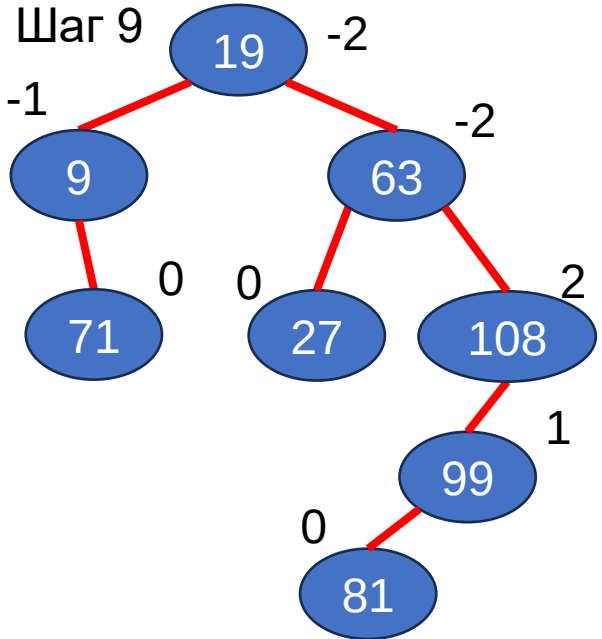
Шаг 7



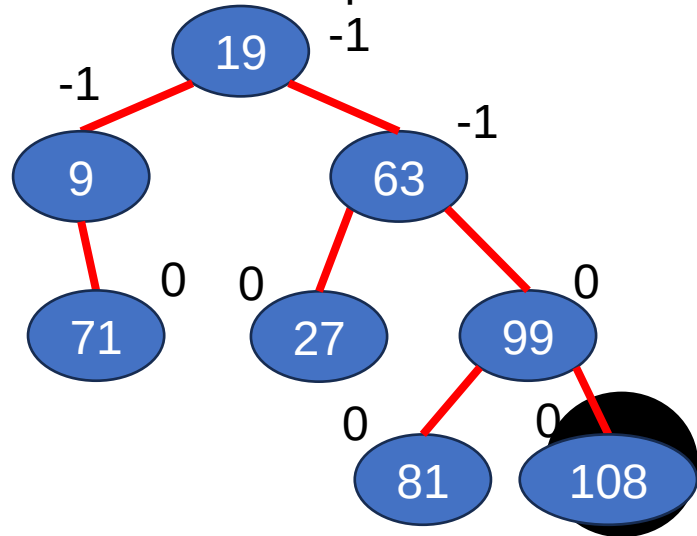
Шаг 8



Шаг 9



Шаг 10 – После RR поворота

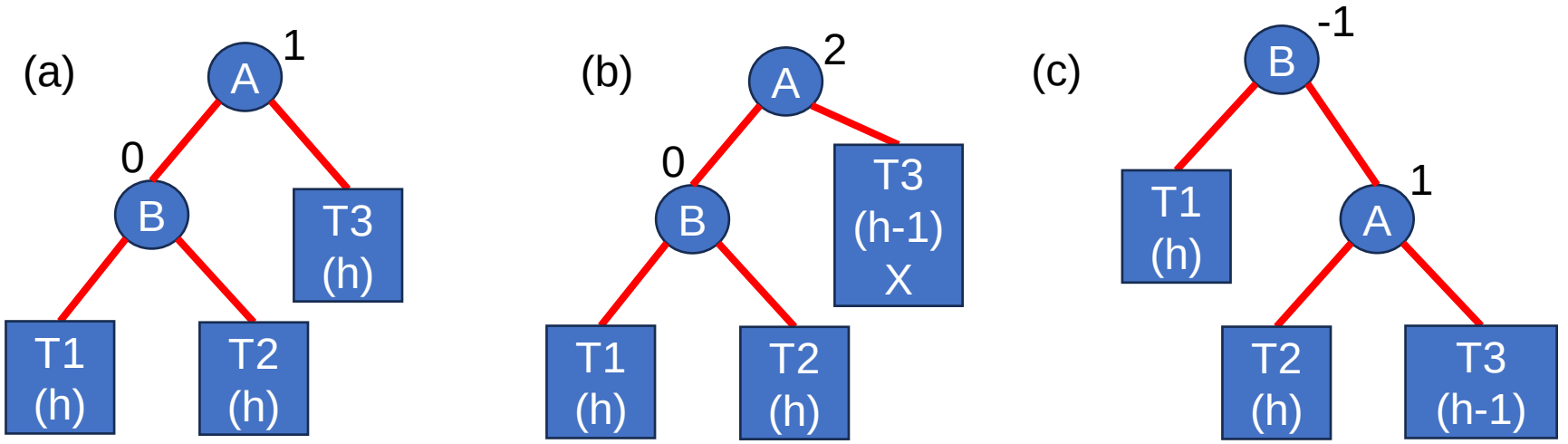


Удаление узла из дерева AVL

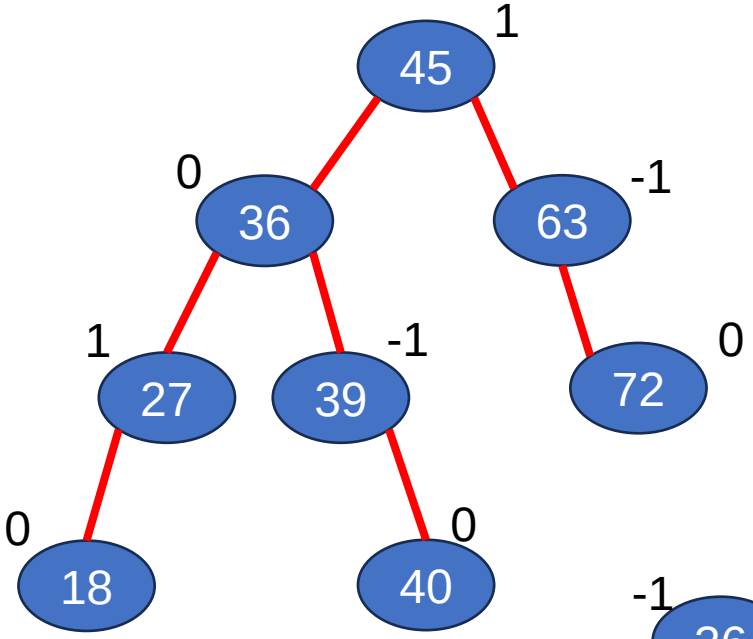
Удаление узла в дереве AVL похоже на удаление узла в деревьях бинарного поиска. Но оно идет на шаг впереди. Удаление может нарушить AVL-свойство дерева, поэтому для повторной балансировки дерева AVL нам нужно выполнить вращения. Существует два класса вращений, которые можно выполнить на дереве AVL после удаления заданного узла. Эти вращения — R-вращение и L-вращение. При удалении узла X из дерева AVL, если узел A становится критическим узлом (ближайший узел-предок на пути от X к корневому узлу, который не имеет своего фактора баланса 1, 0 или –1), то тип поворота зависит от того, находится ли X в левом поддереве A или в его правом поддереве. Если удаляемый узел присутствует в левом поддереве A, то применяется поворот L, в противном случае, если X находится в правом поддереве, выполняется поворот R. Кроме того, существует три категории поворотов L и R. Разновидностями поворота L являются повороты L–1, L0 и L1. Соответственно для поворота R существуют повороты R0, R–1 и R1. В этом разделе мы обсудим только поворот R. Повороты L являются зеркальными отражениями поворотов R.

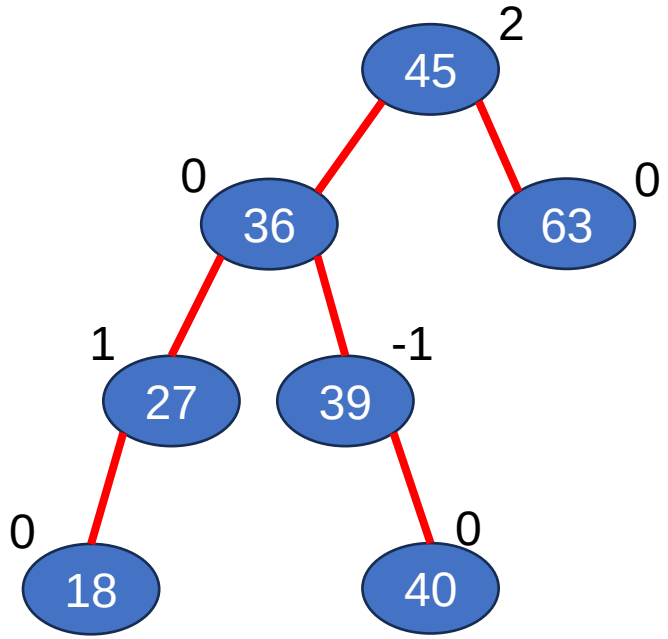
Поворот R0

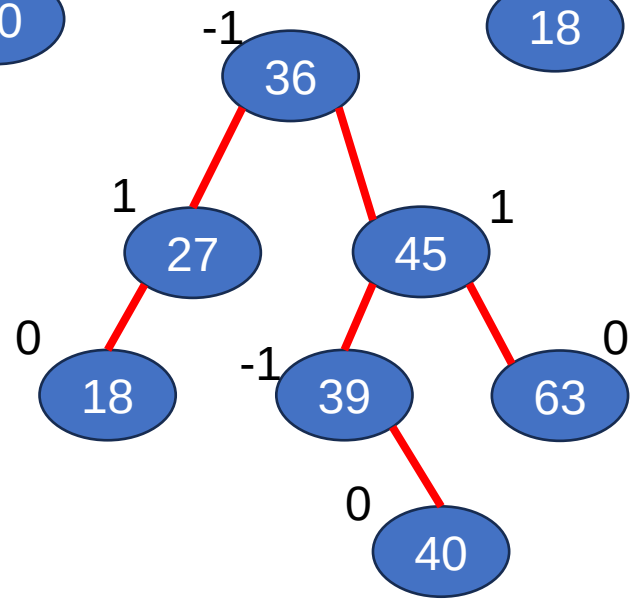
Дерево (a) является деревом AVL. В дереве (b) узел X должен быть удален из правого поддерева критического узла A (узел A является критическим узлом, поскольку он является ближайшим предком, коэффициент баланса которого не равен –1, 0 или 1). Поскольку коэффициент баланса узла B равен 0, мы применяем вращение R0, как показано на дереве (c). В процессе вращения узел B становится корнем, а T1 и A — его левым и правым потомками. T2 и T3 становятся левым и правым поддеревьями A.



Повороты R0 - Пример

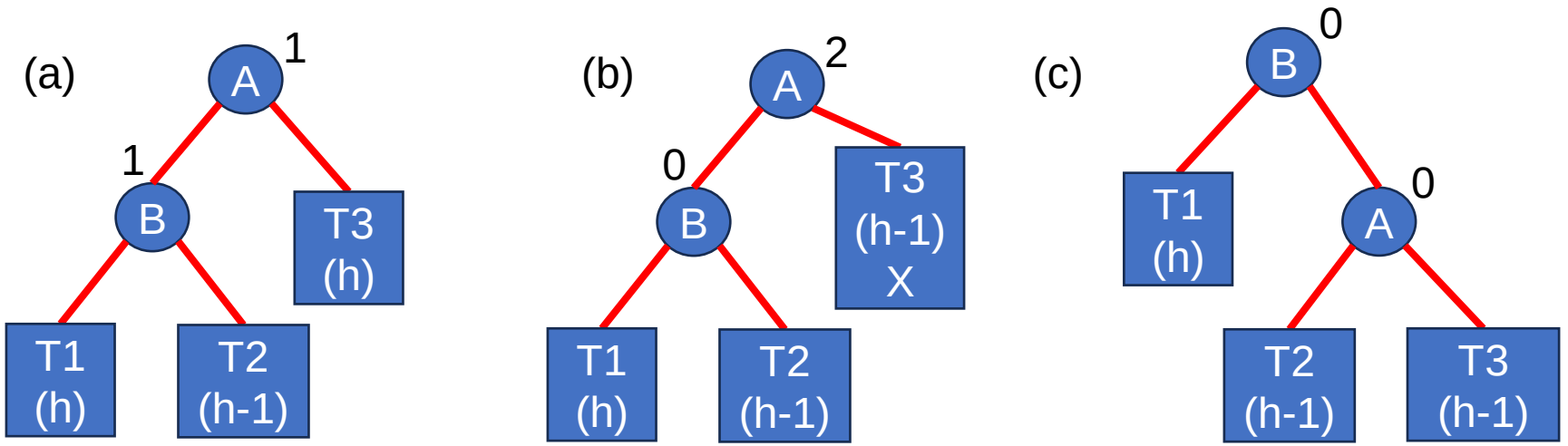




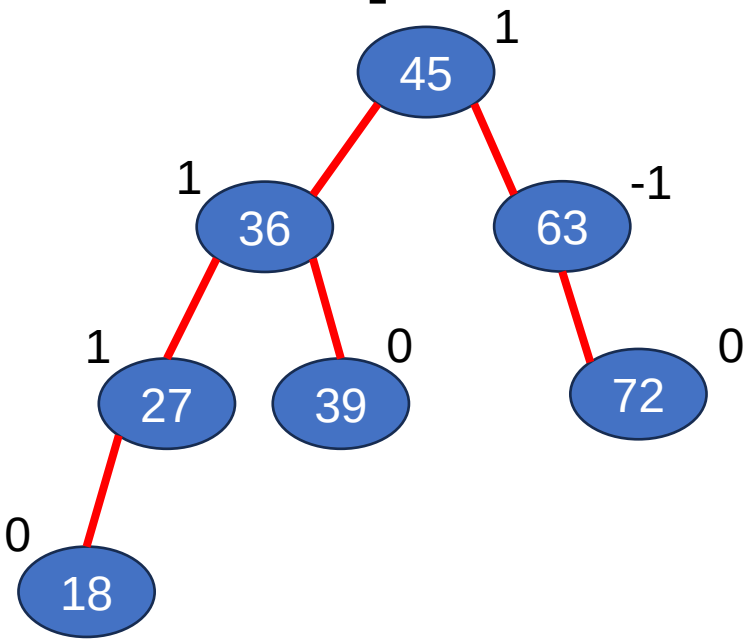


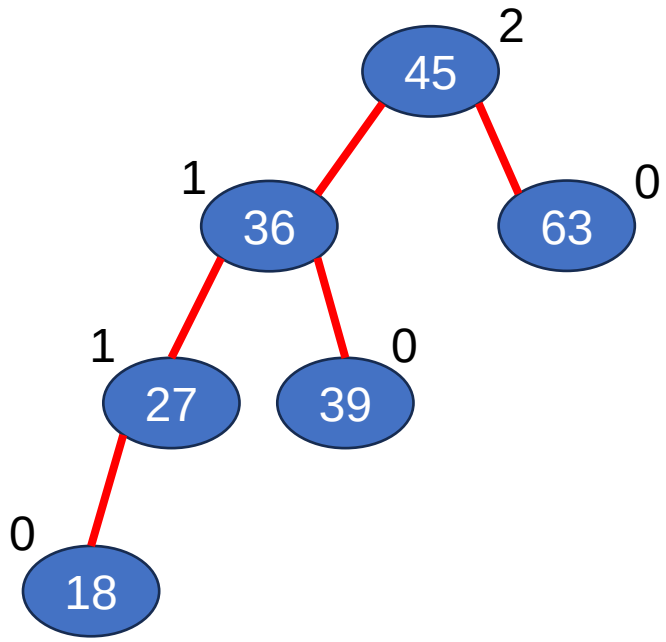
Поворот R1

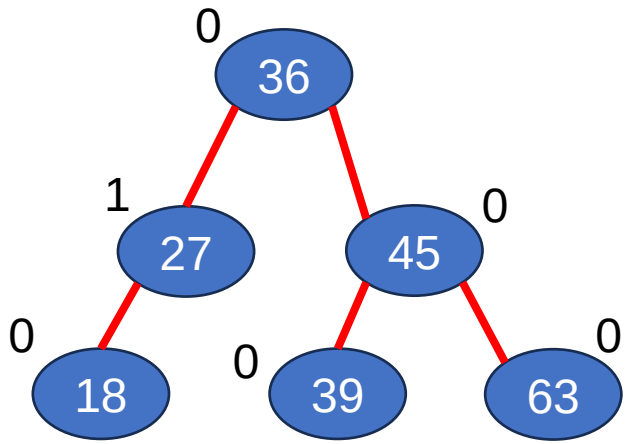
Пусть B будет корнем левого или правого поддерева A (критический узел). Вращение R1 применяется, если коэффициент баланса B равен 1. Обратите внимание, что вращения R0 и R1 аналогичны вращениям LL; единственное отличие состоит в том, что вращения R0 и R1 дают разные коэффициенты баланса. Дерево (a) является деревом AVL. В дереве (b) узел X должен быть удален из правого поддерева критического узла A (узел A является критическим узлом, поскольку он является ближайшим предком, коэффициент баланса которого не равен –1, 0 или 1). Поскольку коэффициент баланса узла B равен 1, мы применяем вращение R1, как показано на дереве (c). В процессе вращения узел B становится корнем, а T1 и A — его левым и правым потомками. T2 и T3 становятся левым и правым поддеревьями A.



Повороты R1 - Пример



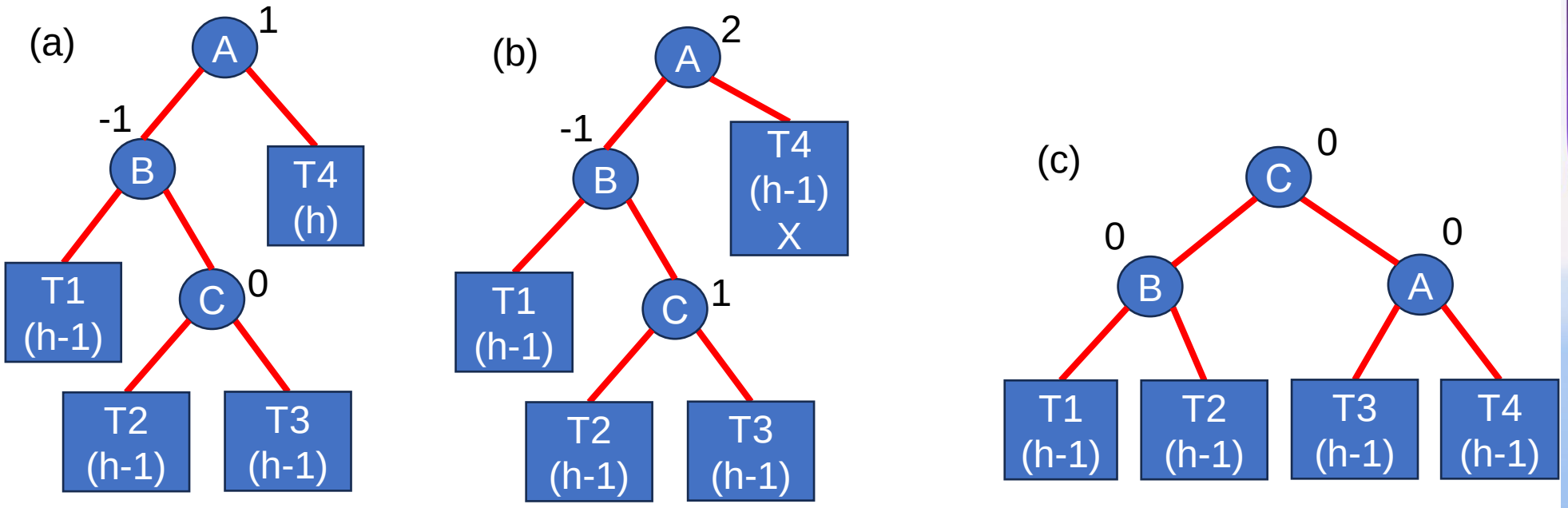




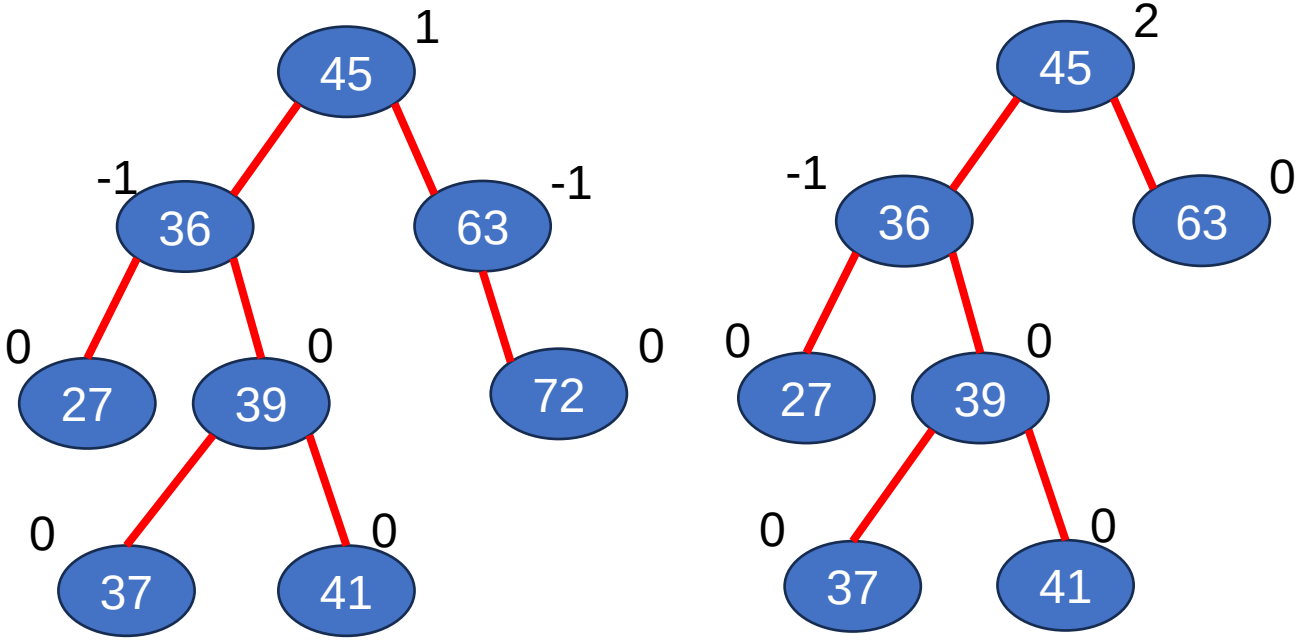
Повороты R-1

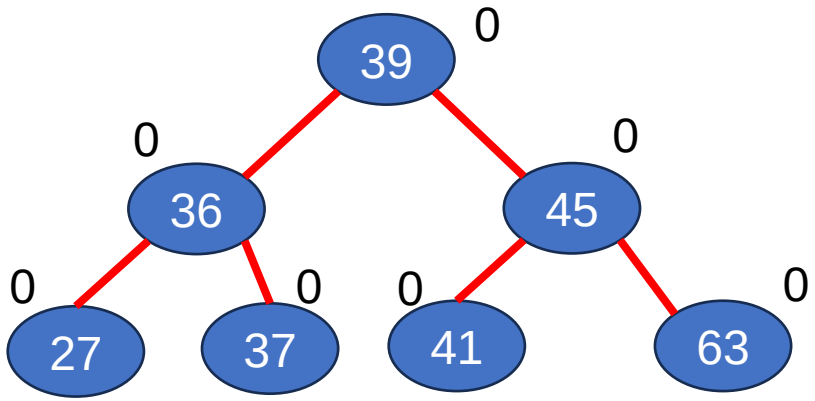
Дерево (a) является деревом AVL. В дереве (b) узел X должен быть удален из правого поддерева критического узла A (узел A является критическим узлом, поскольку он является ближайшим предком, коэффициент баланса которого не равен –1, 0 или 1). Поскольку коэффициент баланса узла B равен –1, мы применяем вращение R–1, как показано в дереве (c).

Во время вращения узел C становится корнем, а T1 и A являются его левым и правым потомками. T2 и T3 становятся левым и правым поддеревьями A.



Повороты R0 - Пример





Красно-черные деревья

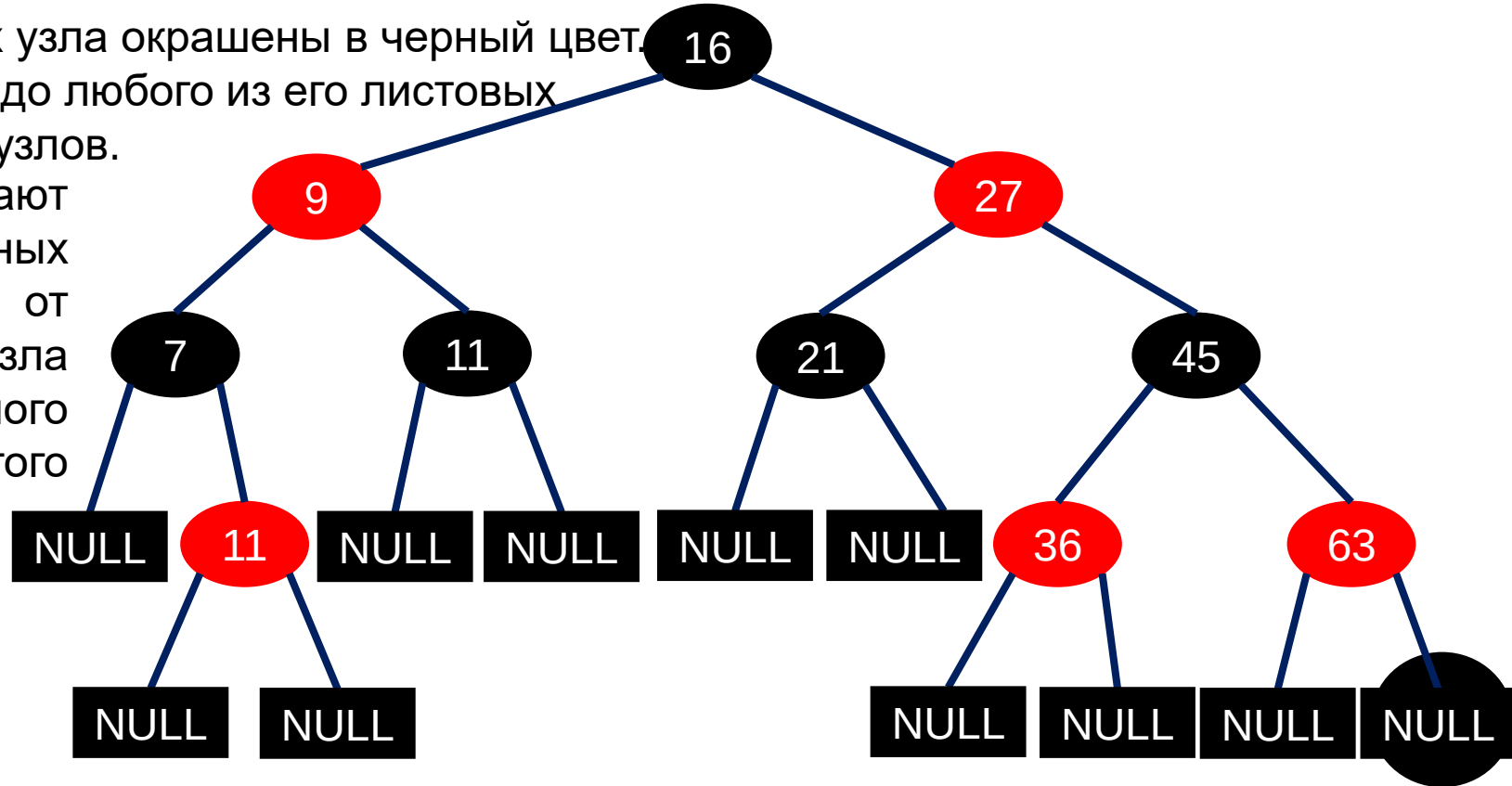
Красно-черное дерево — это самобалансирующееся бинарное дерево поиска, которое было изобретено в 1972 году Рудольфом Байером, который назвал его «симметричным бинарным B-деревом». Хотя красно-черное дерево сложное, оно имеет хорошее время выполнения в худшем случае для своих операций и эффективно в использовании, поскольку поиск, вставка и удаление могут быть выполнены за время O(log n), где n — количество узлов в дереве. Практически, красно-черное дерево — это бинарное дерево поиска, которое вставляет и удаляет разумно, чтобы поддерживать дерево разумно сбалансированным. Особо следует отметить, что в этом дереве в конечных узлах не хранятся никакие данные.

Свойства красно-черных деревьев

Красно-черное дерево — это бинарное дерево поиска, в котором каждый узел имеет цвет, который является либо красным, либо черным. Помимо других ограничений бинарного дерева поиска, красно-черное дерево имеет следующие дополнительные требования:

1. Цвет узла может быть либо красным, либо черным.
2. Цвет корневого узла всегда черный.
3. Все листовые узлы черные.
4. У каждого красного узла оба дочерних узла окрашены в черный цвет.
5. Каждый простой путь от данного узла до любого из его листовых узлов имеет равное количество черных узлов.

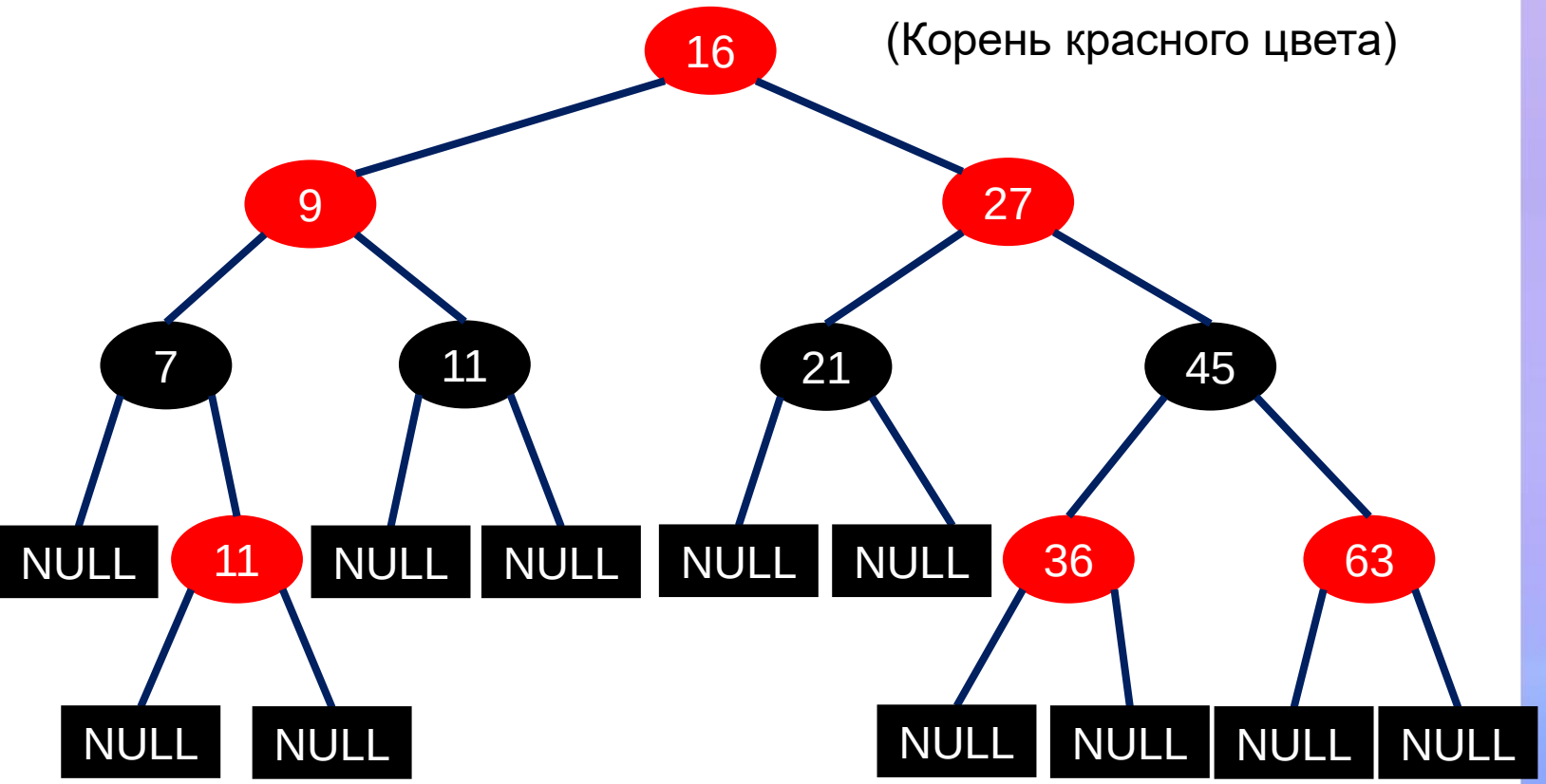
Эти ограничения обеспечивают критическое свойство красно-черных деревьев. Самый длинный путь от корневого узла до любого листового узла не более чем в два раза длиннее самого короткого пути от корня до любого другого листа в этом дереве.



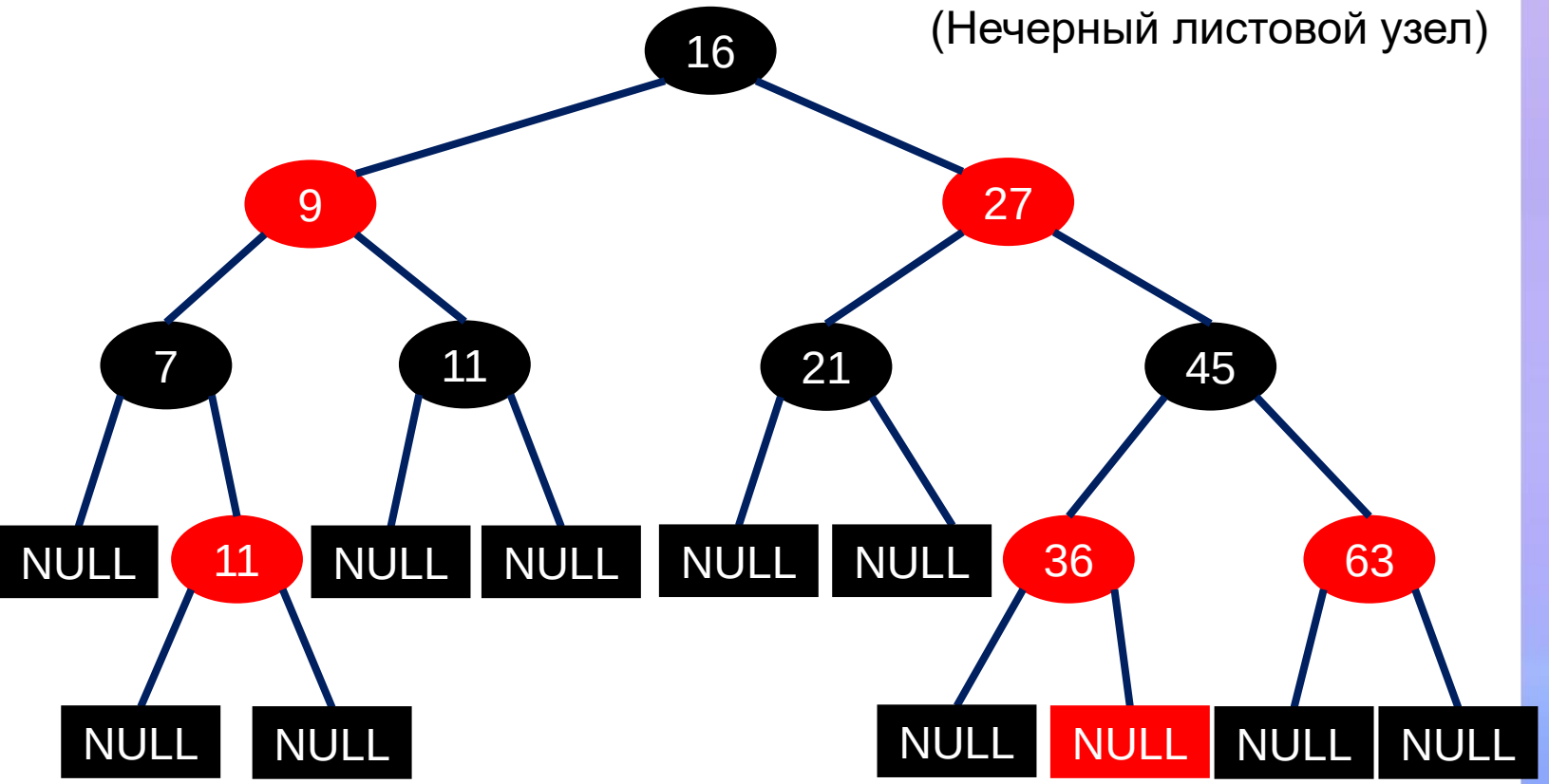
Свойства красно-черных деревьев

Это приводит к примерно сбалансированному дереву. Поскольку такие операции, как вставка, удаление и поиск, требуют наихудшего времени, пропорционального высоте дерева, эта теоретическая верхняя граница высоты позволяет красно-черным деревьям быть эффективными в наихудшем случае, в отличие от обычных бинарных деревьев поиска. Чтобы понять важность этих свойств, достаточно отметить, что согласно свойству 4 ни один путь не может иметь два красных узла подряд. Самый короткий возможный путь будет иметь все черные узлы, а самый длинный возможный путь будет попеременно иметь красный и черный узел. Поскольку все максимальные пути имеют одинаковое количество черных узлов (свойство 5), ни один путь не будет более чем в два раза длиннее любого другого пути.

Не являются красно-черными деревьями

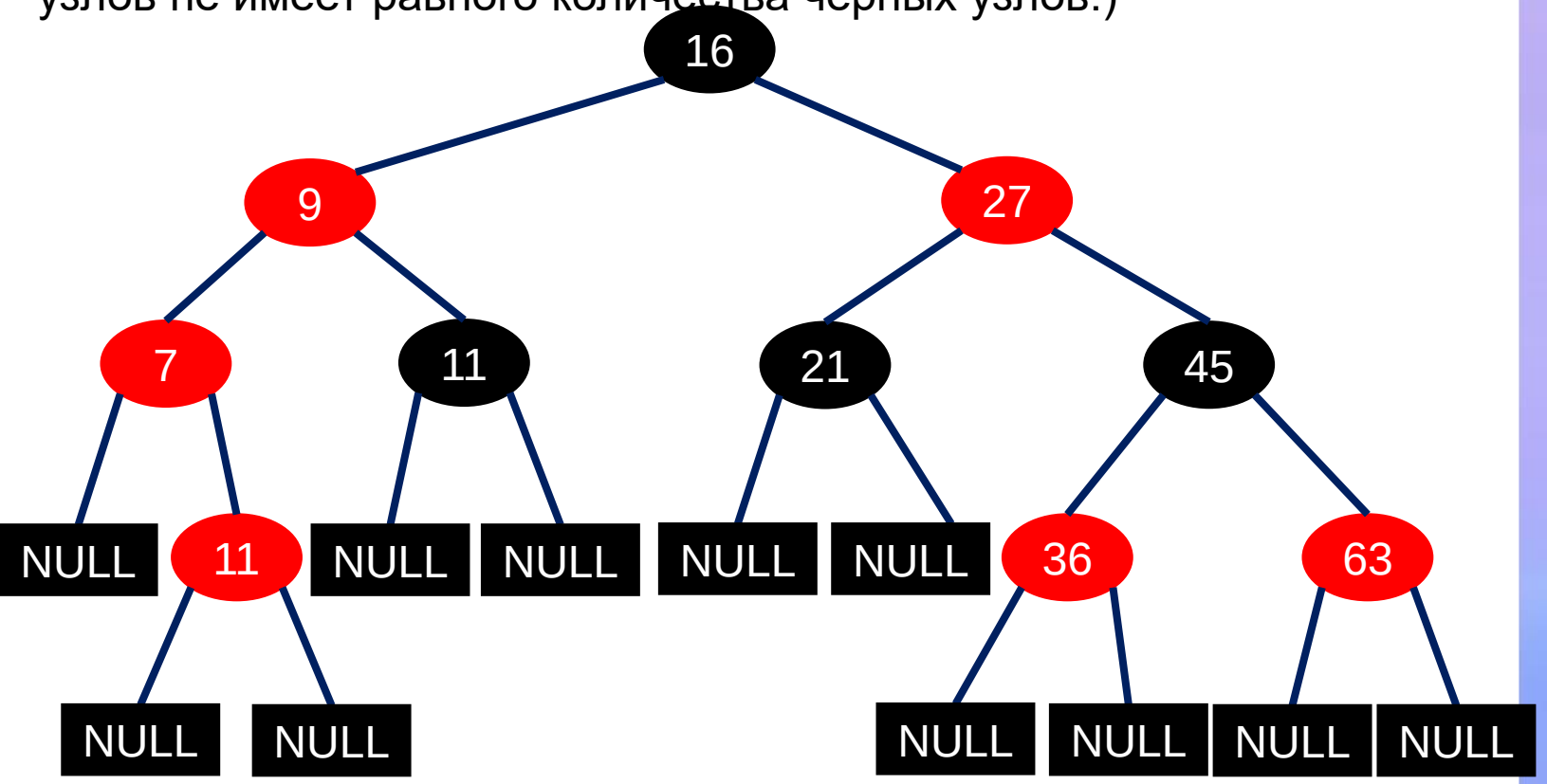


Не являются красно-черными деревьями



Не являются красно-черными деревьями

(Во-первых, каждый красный узел не имеет обоих дочерних узлов, окрашенных в черный цвет. Во-вторых, каждый простой путь от данного узла до любого из его листовых узлов не имеет равного количества черных узлов.)



Операции над красно-черными деревьями

Выполнение операции только для чтения (например, обход узлов в дереве) над красно-черным деревом не требует никаких изменений по сравнению с теми, которые используются для бинарных деревьев поиска. Помните, что каждое красно-черное дерево является частным случаем двоичного дерева поиска. Однако операции вставки и удаления могут нарушать свойства красно-черного дерева. Поэтому эти операции могут создать необходимость восстановления красно-черных свойств, что может потребовать небольшого количества (O(log n) или амортизированных O(1)) изменений цвета.

```
typedef enum
{
    BLACK,
    RED
} bool;
struct node
{
    int val;
    int colour;
    struct node *left;
    struct node *right;
    struct node *parent;
};
```

Вставка узла в красно-черное дерево

Операция вставки начинается так же, как мы добавляем новый узел в двоичное дерево поиска. Однако в двоичном дереве поиска мы всегда добавляем новый узел как лист, в то время как в красно-черном дереве листовые узлы не содержат данных. Поэтому вместо добавления нового узла как листового узла мы добавляем красный внутренний узел, который имеет два черных листовых узла. Обратите внимание, что цвет нового узла красный, а его листовые узлы окрашены в черный цвет.

После добавления нового узла он может нарушить некоторые свойства красно-черного дерева. Поэтому для восстановления их свойства мы проверяем определенные случаи и восстанавливаем свойство в зависимости от случая, который появляется после вставки. Но прежде чем подробно изучать эти случаи, давайте сначала обсудим некоторые важные термины, которые будут использоваться.

Вставка узла в красно-черное дерево

Узел прародителя (G) узла (N) относится к родителю родителя N (P), как в человеческих генеалогических древах. Код C для поиска прародителя узла может быть задан следующим образом:

```
struct node *grand_parent(struct node *n)
{
    // No parent means no grandparent
    if ((n != NULL) && (n->parent != NULL))
        return n->parent->parent;
    else
        return NULL;
}
```

Вставка узла в красно-черное дерево

Узел дяди (U) узла (N) относится к брату родителя N (P), как в человеческих генеалогических древах. Код C для поиска дяди узла может быть задан следующим образом:

```c
struct node *uncle(struct node *n)
{
    struct node *g;
    g = grand_parent(n);
    // With no grandparent, there cannot be any uncle
    if (g == NULL)
        return NULL;
    if (n->parent == g->left)
        return g->right;
    else
        return g->left;
}
```

Вставка узла в красно-черное дерево

Когда мы вставляем новый узел в красно-черное дерево, обратите внимание на следующее:

- Все листовые узлы всегда черные. Поэтому свойство 3 всегда выполняется.
- Свойство 4 (оба потомка каждого красного узла черные) находится под угрозой только при добавлении красного узла, перекрашивании черного узла в красный цвет или повороте.
- Свойство 5 (все пути от любого заданного узла до его листовых узлов имеют одинаковое количество черных узлов) находится под угрозой только при добавлении черного узла, перекрашивании красного узла в черный цвет или повороте.

Вставка узла в красно-черное дерево

Случай 1: *новый узел N добавляется как корень дерева*
В этом случае N перекрашивается в черный цвет, так как корень дерева всегда черный. Поскольку N добавляет один черный узел к каждому пути одновременно, свойство 5 не нарушается. Код C для случая 1 можно задать следующим образом:

```
void case1(struct node *n)
{
    if (n->parent == NULL) // Root node
        n->colour = BLACK;
    else
        case2(n);
}
```

Вставка узла в красно-черное дерево

Случай 2: ***Родительский узел P нового узла черный***

В этом случае оба потомка каждого красного узла черные, поэтому свойство 4 не становится недействительным. Свойство 5 также не находится под угрозой. Это связано с тем, что новый узел N имеет два черных листовых потомка, но поскольку N красный, пути через каждого из его потомков имеют одинаковое количество черных узлов. Код C для проверки для случая 2 можно задать следующим образом:

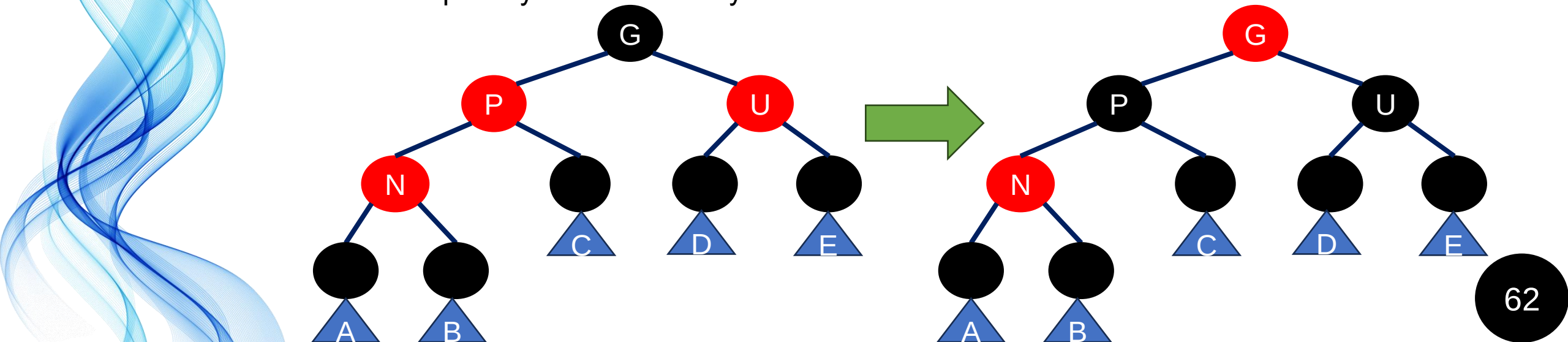
```
void case2(struct node *n)
{
    if (n->parent->colour == BLACK)
        return;
/* Свойство красного черного дерева не нарушается */
    else
        case3(n);
}
```

Вставка узла в красно-черное дерево

В следующих случаях предполагается, что N имеет узел-прародителя G, потому что его родитель P красный, а если бы он был корнем, то был бы черным. Таким образом, у N также есть узел-дядя U (независимо от того, является ли U листовым узлом или внутренним узлом).

Случай 3: *Если и родитель (P), и дядя (U) красные*

В этом случае нарушается Свойство 5, которое гласит, что все пути от любого данного узла до его листовых узлов имеют одинаковое количество черных узлов. Чтобы восстановить Свойство 5, оба узла (P и U) перекрашиваются в черный цвет, а прародитель G перекрашивается в красный цвет. Теперь новый красный узел N имеет черного родителя. Поскольку любой путь через родителя или дядю должен проходить через прародителя, количество черных узлов на этих путях не изменилось.



Вставка узла в красно-черное дерево

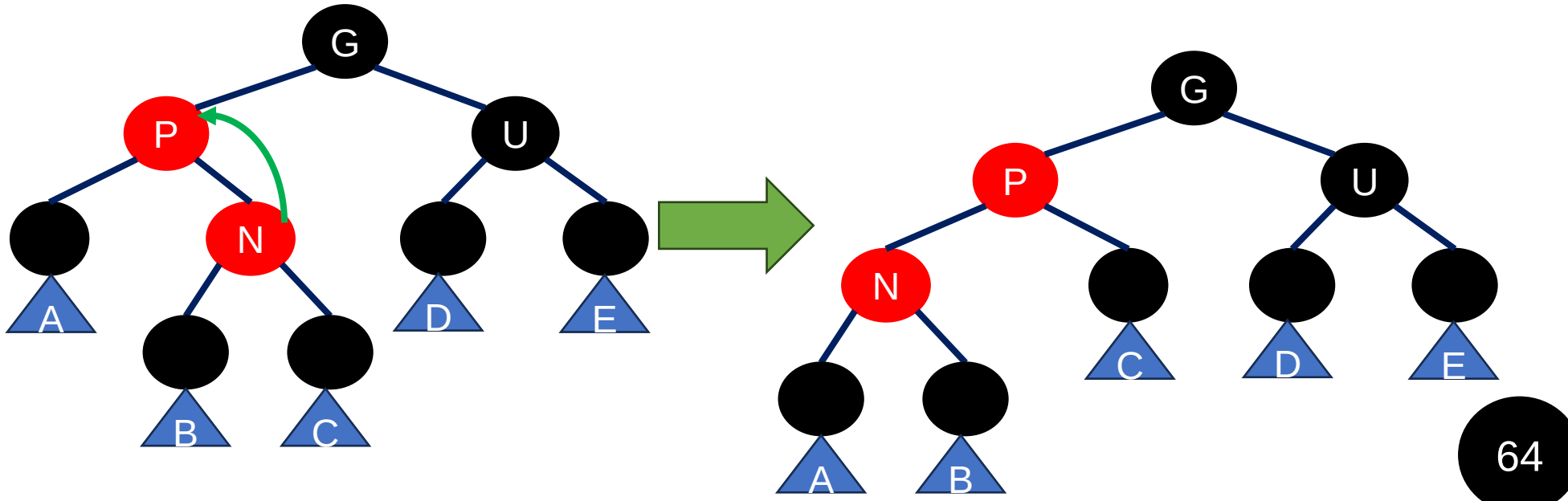
Однако прародитель G теперь может нарушить Свойство 2, которое гласит, что корневой узел всегда черный, или Свойство 4, которое гласит, что оба потомка каждого красного узла черные. Свойство 4 будет нарушено, когда у G есть красный родитель. Чтобы исправить эту проблему, вся эта процедура рекурсивно выполняется на G из случая 1. Код C для вставки случая 3 выглядит следующим образом:

```
void case3(struct node *n)
{
    struct node *u, *g;
    u = uncle(n);
    g = grand_parent(n);
    if ((u != NULL) && (u->colour == RED))
    {
        n->parent->colour = BLACK;
        u->colour = BLACK;
        g->colour = RED;
        case1(g);
    }
    else
        insert_case4(n);
}
```

Вставка узла в красно-черное дерево

Случай 4: ***Родитель P — красный, но дядя U — черный, а N — правый потомок P, а P — левый потомок G***

Чтобы исправить эту проблему, выполняется поворот влево, чтобы поменять роли нового узла N и его родителя P. После поворота обратите внимание, что в коде C мы переименовали N и P, а затем вызывается случай 5 для работы с родителем нового узла. Это делается потому, что свойство 4, которое гласит, что оба потомка каждого красного узла должны быть черными, все еще нарушается. На рисунке показана вставка случая 4. Обратите внимание, что в случае, если N — левый потомок P, а P — правый потомок G, нам нужно выполнить поворот вправо. В коде C, который обрабатывает случай 4, мы проверяем P и N, а затем выполняем поворот влево или вправо.



Вставка узла в красно-черное дерево

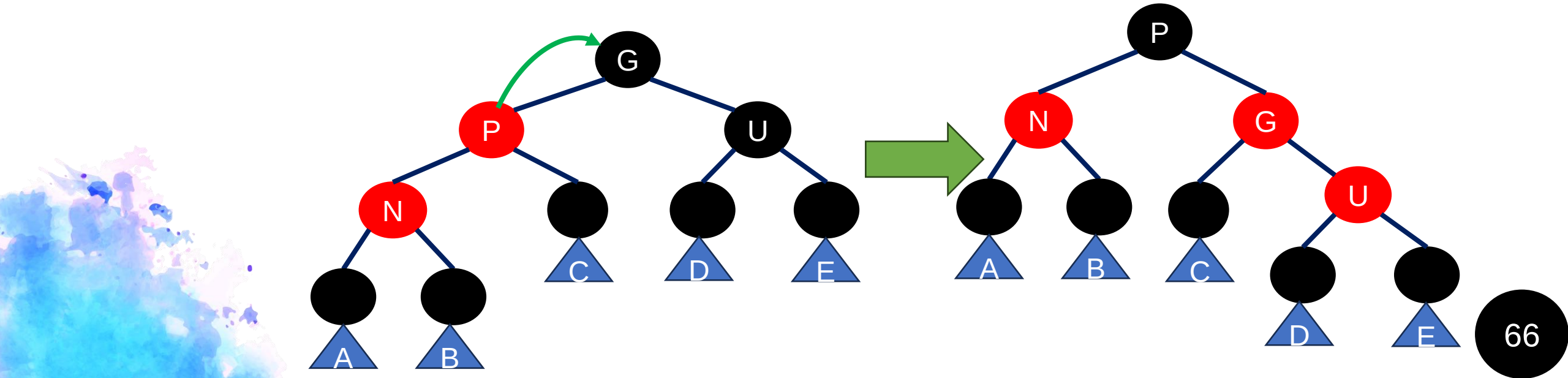
Однако прародитель G теперь может нарушить Свойство 2, которое гласит, что корневой узел всегда черный, или Свойство 4, которое гласит, что оба потомка каждого красного узла черные. Свойство 4 будет нарушено, когда у G есть красный родитель. Чтобы исправить эту проблему, вся эта процедура рекурсивно выполняется на G из случая 1. Код C для вставки случая 3 выглядит следующим образом:

```
void case4(struct node *n)
{
    struct node *g = grand_parent(n);
    if ((n == n->parent->right) && (n->parent == g->left))
    {
        rotate_left(n->parent);
        n = n->left;
    }
    else if ((n == n->parent->left) && (n->parent == g->right))
    {
        rotate_right(n->parent);
        n = n->right;
    }
    case5(n);
}
```

Вставка узла в красно-черное дерево

Случай 5: ***Родительский узел P — красный, но дядя U — черный, а новый узел N — левый потомок P, а P — левый потомок своего родителя G.***

Чтобы исправить эту проблему, выполняется поворот вправо на G (прародительском узле N). После этого поворота бывший родитель P теперь является родителем как нового узла N, так и бывшего прародителя G. Мы знаем, что цвет G черный (потому что в противном случае его бывший потомок P не мог бы быть красным), поэтому теперь поменяем цвета P и G так, чтобы полученное дерево удовлетворяло Свойству 4, которое гласит, что оба потомка красного узла являются черными. Случай 5 вставки проиллюстрирован на рисунке.



Вставка узла в красно-черное дерево

Случай 5: ***Родительский узел P — красный, но дядя U — черный, а новый узел N — левый потомок P, а P — левый потомок своего родителя G.***

```
void case5(struct node *n)
{
    struct node *g;
    g = grandparent(n);
    if ((n == n->parent->left) && (n->parent == g->left))
        rotate_right(g);
    else if ((n == n->parent->right) && (n->parent == g->right))
        rotate_left(g);
    n->parent->colour = BLACK;
    g->colour = RED;
}
```

Удаление узла из красно-черного дерева

Мы начинаем удалять узел из красно-черного дерева так же, как и в случае с бинарным деревом поиска. В бинарном дереве поиска, когда мы удаляем узел с двумя нелистовыми дочерними элементами, мы находим либо максимальный элемент в его левом поддереве узла, либо минимальный элемент в его правом поддереве и перемещаем его значение в удаляемый узел. После этого мы удаляем узел, из которого мы скопировали значение. Обратите внимание, что этот узел должен иметь менее двух нелистовых дочерних элементов. Поэтому простое копирование значения не нарушает никаких красно-черных свойств, а просто сводит проблему удаления к проблеме удаления узла с не более чем одним нелистовым дочерним элементом.

В этом разделе мы предположим, что удаляем узел с не более чем одним нелистовым дочерним элементом, который мы назовем его дочерним элементом. Если у этого узла есть оба листовых дочерних элемента, то пусть один из них будет его дочерним элементом.

При удалении узла, если его цвет красный, то мы можем просто заменить его его дочерним элементом, который должен быть черным. Все пути через удаленный узел будут просто проходить через один красный узел меньше, и как родительский, так и дочерний узел удаленного узла должны быть черными, поэтому ни одно из свойств не будет нарушено.

Удаление узла из красно-черного дерева

Другой простой случай — когда мы удаляем черный узел, у которого есть красный дочерний узел. В этом случае могут быть нарушены свойства 4 и 5, поэтому для их восстановления просто перекрасьте дочерний узел удаленного узла в черный цвет.
Однако возникает сложная ситуация, когда и удаляемый узел, и его дочерний узел черные. В этом случае мы начинаем с замены удаляемого узла его дочерним узлом. В коде C мы обозначаем дочерний узел как (в его новой позиции) N, а его родственный узел (другой дочерний узел его нового родителя) как S. Код C для поиска родственного узла узла можно задать следующим образом:

```
struct node *sibling(struct node *n)
{
    if (n == n->parent->left)
        return n->parent->right;
    else
        return n->parent->left;
}
```

Удаление узла из красно-черного дерева

Мы можем начать процесс удаления, используя следующий код, где функция replace_node подставляет дочерний узел на место N в дереве. Для удобства мы предполагаем, что пустые листья представлены реальными объектами узлов, а не NULL.
Когда и N, и его родительский узел P черные, то удаление P приведет к тому, что пути, которые предшествуют N, будут иметь на один черный узел меньше, чем другие пути. Это нарушит Свойство 5. Поэтому дерево необходимо перебалансировать. Необходимо рассмотреть несколько случаев

```
void delete_child(struct node *n)
{
/* If N has at most one non-null child */
    struct node *child;
    if (is_leaf(n->right))
        child = n->left;
    else
        child = n->right;
    replace_node(n, child);
    if (n->colour == BLACK)
    {
        if (child->colour == RED)
            child->colour = BLACK;
        else
            del_case1(child);
    }
    free(n);
}
```

Удаление узла из красно-черного дерева

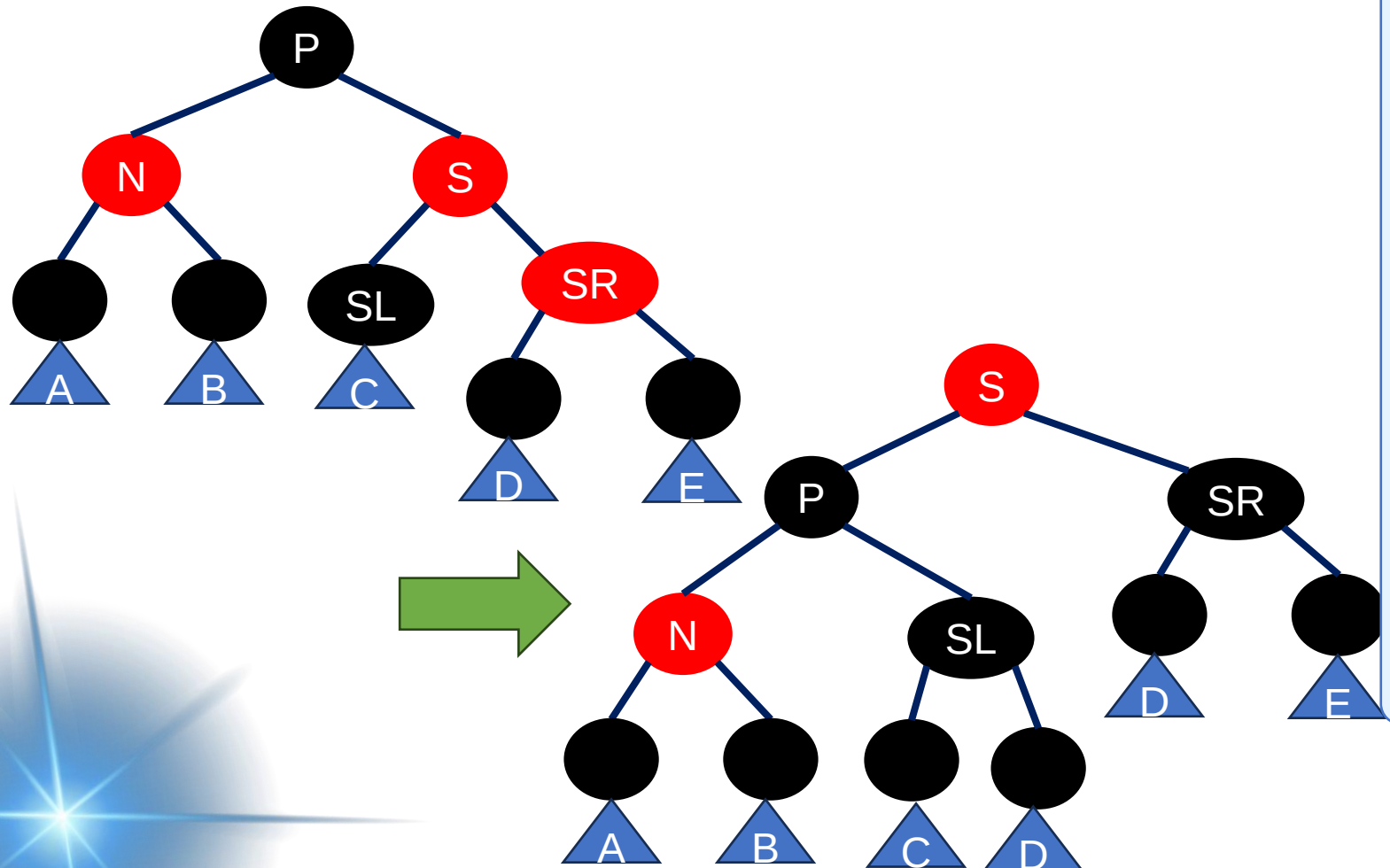
Случай 1: *N — новый корень*
В этом случае мы удалили один черный узел из каждого пути, и новый корень черный, поэтому ни одно из свойств не нарушено.

```
void del_case1(struct node *n)
{
    if (n->parent != NULL)
        del_case2(n);
}
```

Удаление узла из красно-черного дерева

Случай 2: родственный элемент S — красный

В этом случае поменяйте цвета P и S, а затем поверните влево в P. В полученном дереве S станет прародителем N.

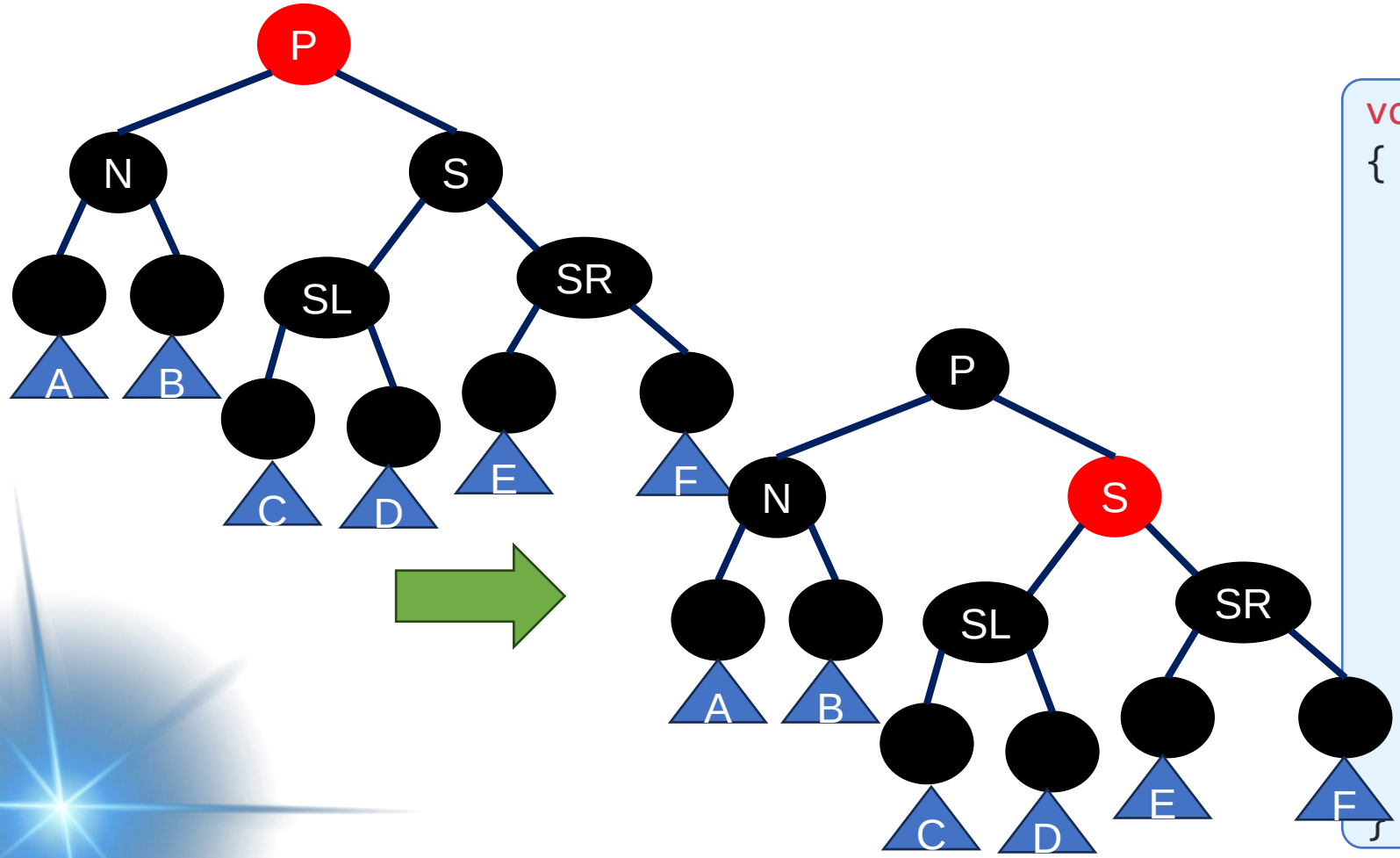



```
void del_case2(struct node *n)
{
    struct node *s;
    s = sibling(n);
    if (s->colour == RED)
    {
        if (n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent);
        n->parent->colour = RED;
        s->colour = BLACK;
    }
    del_case3(n);
}
```

Удаление узла из красно-черного дерева

Случай 3: *P, S и потомки S черные*

В этом случае просто перекрасьте S в красный цвет. В полученном дереве все пути, проходящие через S, будут иметь на один черный узел меньше.

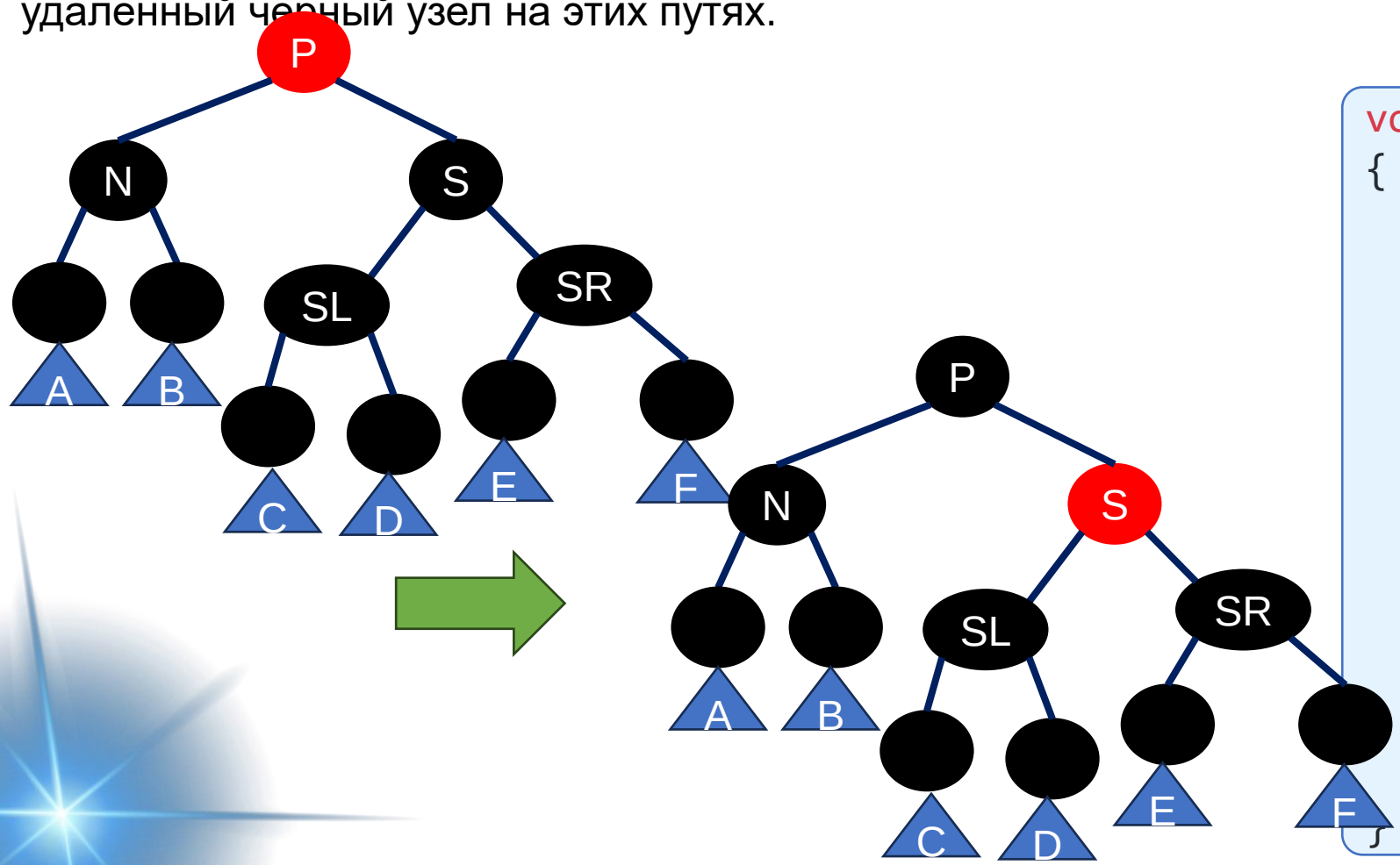



```
void del_case3(struct node *n)
{
    struct node *s;
    s = sibling(n);
    if ((n->parent->colour == BLACK)
        && (s->colour == BLACK) &&
        (s->left->colour == BLACK) &&
        (s->right->colour == BLACK))
    {
        s->colour = RED;
        del_case1(n->parent);
    }
    else
        del_case4(n);
}
```

Удаление узла из красно-черного дерева

Случай 4: Дочерние элементы S и S черные, а P красный

В этом случае мы меняем цвета S и P. Хотя это не повлияет на количество черных узлов на путях, проходящих через S, это добавит один черный узел к путям, проходящим через N, компенсируя удаленный черный узел на этих путях.

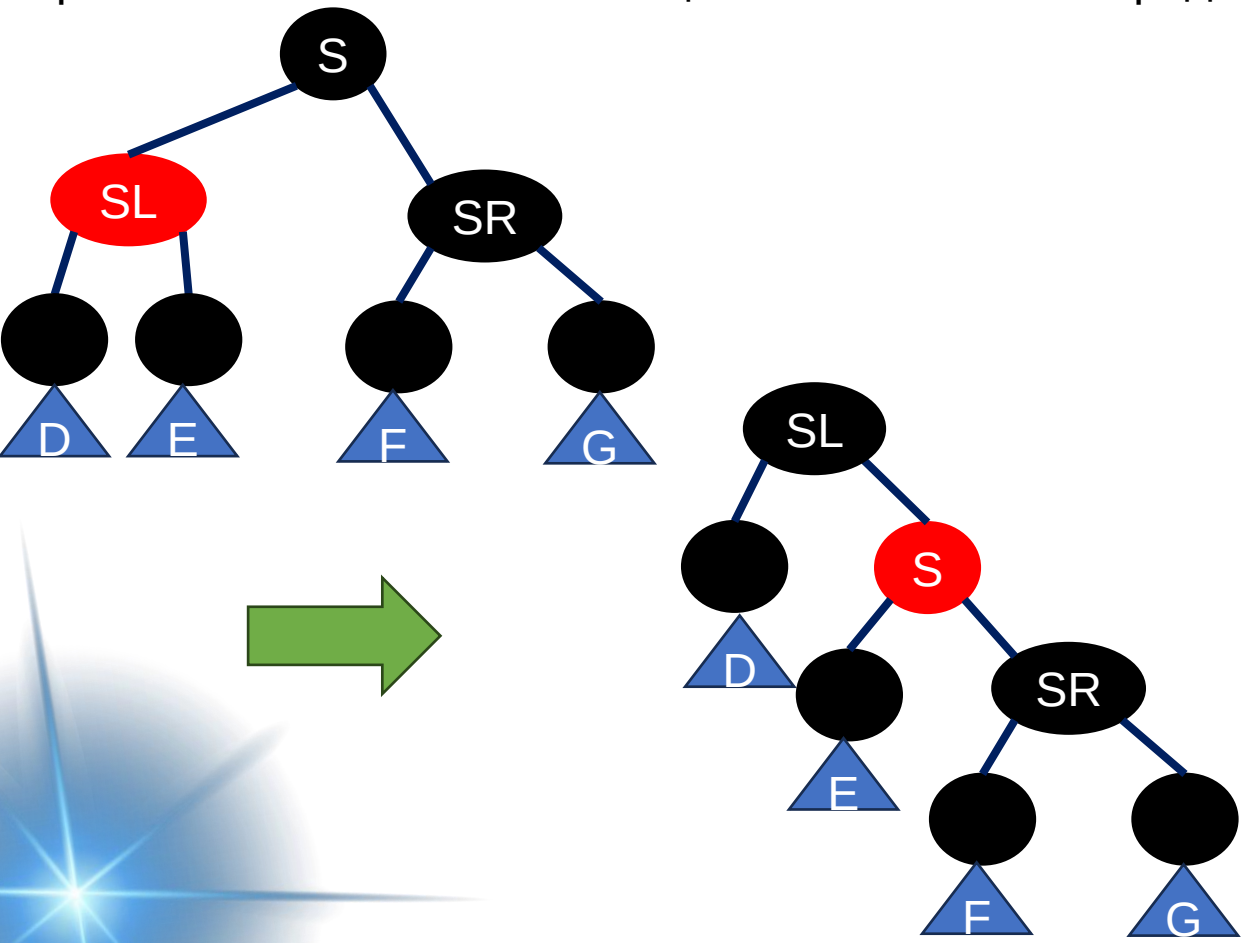



```
void del_case4(struct node *n)
{
    struct node *s;
    s = sibling(n);
    if ((n->parent->colour == RED)
        && (s->colour == BLACK) &&
        (s->left->colour == BLACK) &&
        (s->right->colour == BLACK))
    {
        s->colour = RED;
        n->parent->colour = BLACK;
    }
    else
        del_case5(n);
}
```

Удаление узла из красно-черного дерева

Случай 5: ***N — левый потомок P, а S — черный, левый потомок S — красный, правый потомок S — черный.***

В этом случае выполните поворот вправо в S. После поворота левый потомок S станет родителем S и новым братом N. Также поменяйте цвета S и его нового родителя.

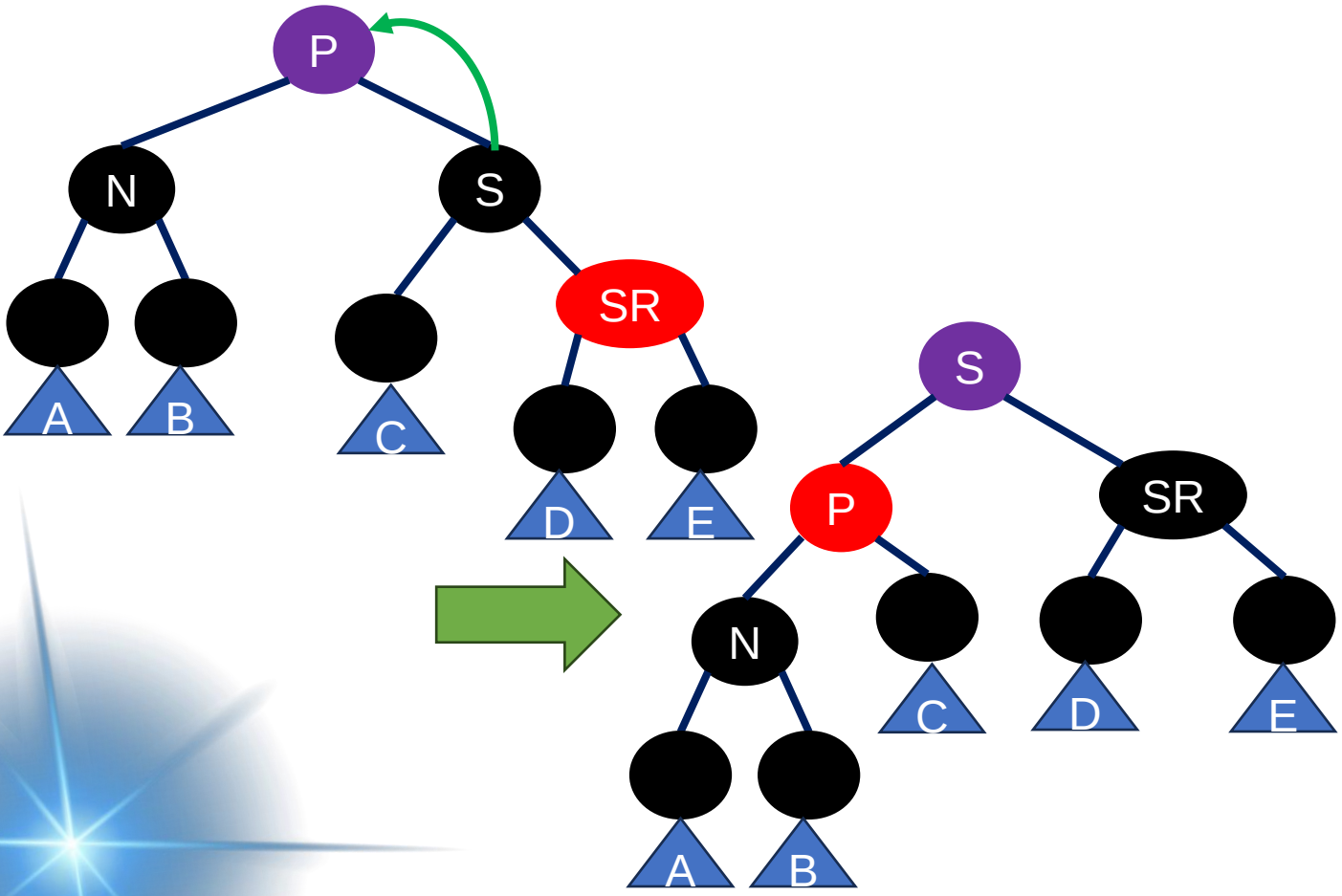



```
void del_case5(struct node *n)
{
    struct node *s;
    s = sibling(n);
    if (s->colour == BLACK)
    {
        if ((n == n->parent->left) &&
                (s->right->colour == BLACK) &&
                (s->left->colour == RED))
            rotate_right(s);
        else if ((n == n->parent->right) &&
                (s->left->colour == BLACK) &&
                (s->right->colour == RED))
            rotate_left(s);
        s->colour = RED;
        s->right->colour = BLACK;
    }
    del_case6(n);
}
```

Удаление узла из красно-черного дерева

Случай 4: Дочерние элементы S и S черные, а P красный

В этом случае мы меняем цвета S и P. Хотя это не повлияет на количество черных узлов на путях, проходящих через S, это добавит один черный узел к путям, проходящим через N, компенсируя удаленный черный узел на этих путях.




```
void del_case6(struct node *n)
{
    struct node *s;
    s = sibling(n);
    s->colour = n->parent->colour;
    n->parent->colour = BLACK;
    if (n == n->parent->left)
    {
        s->right->colour = BLACK;
        rotate_left(n->parent);
    }
    else
    {
        s->left->colour = BLACK;
        rotate_right(n->parent);
    }
}
```

Применение красно-черных деревьев

Красно-черные деревья являются эффективными бинарными деревьями поиска, поскольку они предлагают наихудшую гарантию времени для операций вставки, удаления и поиска. Красно-черные деревья не только ценны в чувствительных ко времени приложениях, таких как приложения реального времени, но также предпочтительны для использования в качестве строительного блока в других структурах данных, которые предоставляют наихудшую гарантию. Деревья AVL также поддерживают операции поиска, вставки и удаления O(log n), но они более жестко сбалансированы, чем красно-черные деревья, что приводит к более медленной вставке и удалению, но более быстрому извлечению данных.

Расширенные (Splay) деревья

Деревья Splay были изобретены Дэниелом Слейтором и Робертом Тарьяном. Splay дерево — это самобалансирующееся бинарное дерево поиска с дополнительным свойством, которое заключается в том, что к недавно использованным элементам можно быстро получить повторный доступ. Оно считается эффективным бинарным деревом, поскольку выполняет основные операции, такие как вставка, поиск и удаление, за амортизированное время O(log(n)). Для многих неравномерных последовательностей операций расширяющиеся деревья работают лучше, чем другие деревья поиска, даже если конкретный шаблон последовательности неизвестен.

Splay дерево состоит из бинарного дерева без дополнительных полей. Когда осуществляется доступ к узлу в расширяющемся дереве, он поворачивается или «расширяется» к корню, тем самым изменяя структуру дерева. Поскольку наиболее часто используемый узел всегда перемещается ближе к начальной точке поиска (или корневому узлу), эти узлы, следовательно, находятся быстрее. Простая идея заключается в том, что если осуществляется доступ к элементу, то, скорее всего, к нему будет осуществлен доступ снова.

В Splay дереве такие операции, как вставка, поиск и удаление, объединены с одной базовой операцией, называемой развертыванием. Развертывание дерева для определенного узла перестраивает дерево так, чтобы поместить этот узел в корень. Методика для этого заключается в том, чтобы сначала выполнить стандартный поиск двоичного дерева для этого узла, а затем использовать вращения в определенном порядке, чтобы переместить узел наверх.

Операции над развернутыми деревьями

четыре основные операции, которые выполняются над splay деревом. К ним относятся

- развертывание
- вставка
- поиск
- удаление.

Развертывание

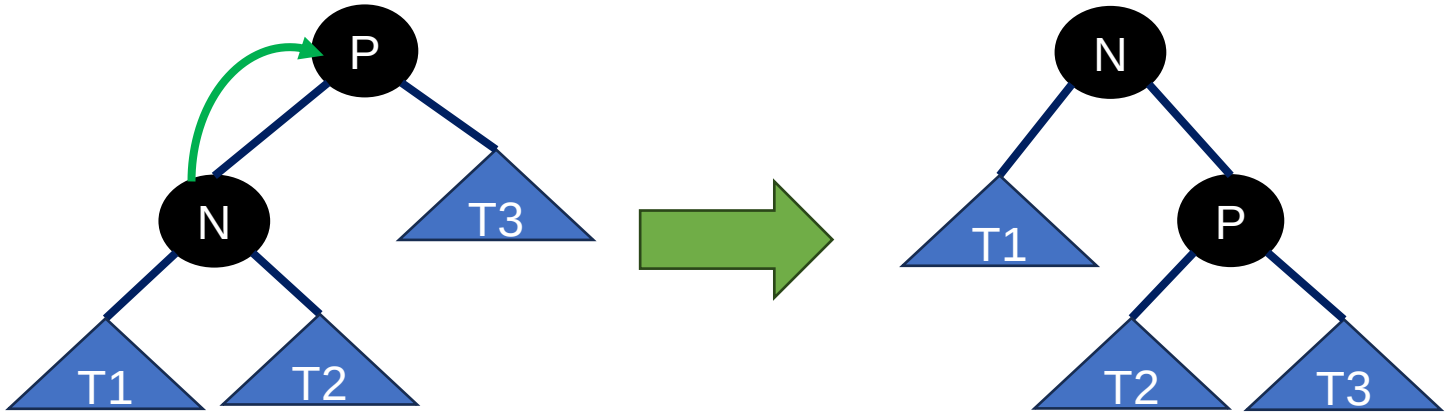
Когда мы получаем доступ к узлу N, развертывание выполняется над N, чтобы переместить его в корень. Для выполнения операции развертывания выполняются определенные шаги развертывания, где каждый шаг перемещает N ближе к корню. Развертывание определенного интересующего узла после каждого доступа гарантирует, что недавно использованные узлы остаются ближе к корню, а дерево остается примерно сбалансированным, так что могут быть достигнуты желаемые амортизированные временные границы. Каждый шаг расширения зависит от трех факторов:

- Является ли N левым или правым потомком своего родителя P,
- Является ли P корнем или нет, и если нет,
- Является ли P левым или правым потомком своего родителя, G (дед N). В зависимости от этих трех факторов у нас есть один шаг расширения на основе каждого фактора.

Развертывание

Шаг зиг

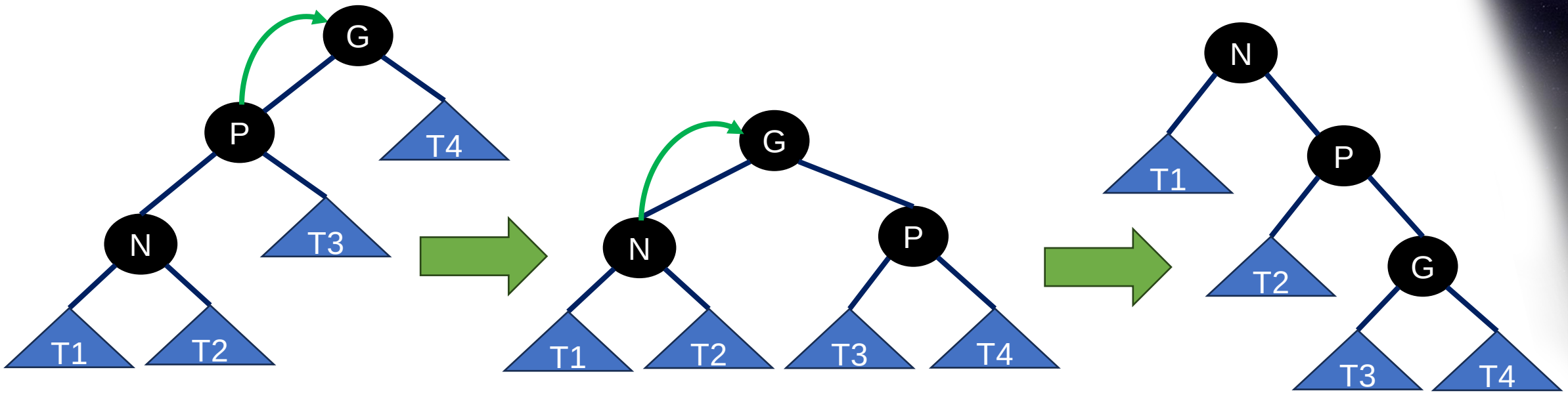
Операция зигзага выполняется, когда P (родитель N) является корнем дерева расширения. На шаге зигзага дерево поворачивается на ребре между N и P. Шаг зигзага обычно выполняется как последний шаг в операции расширения и только когда N имеет нечетную глубину в начале операции.



Развертывание

Шаг зигзиг

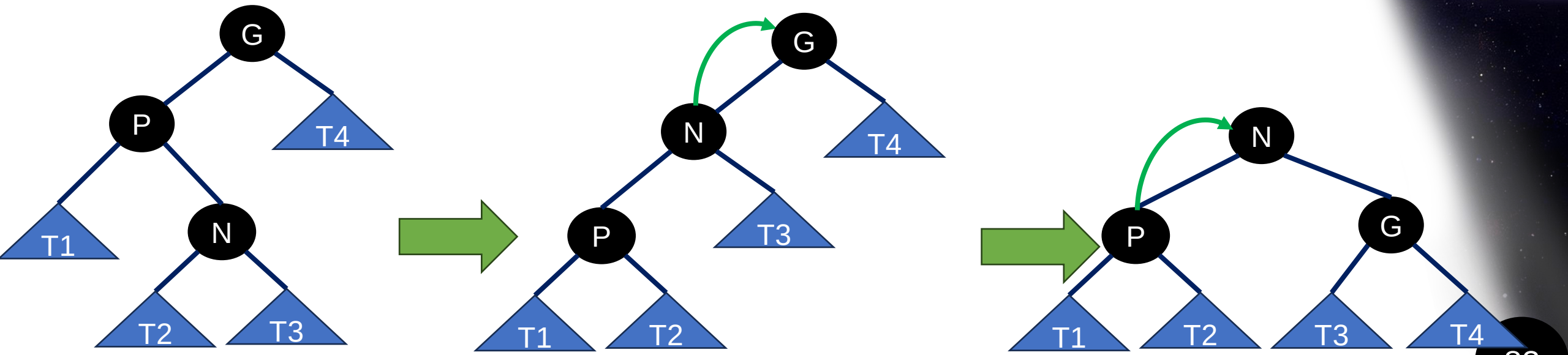
Операция зигзага выполняется, когда P не является корнем. В дополнение к этому, N и P являются либо правыми, либо левыми потомками своих родителей. На рисунке показан случай, когда N и P являются левыми потомками. Во время шага зигзаг сначала дерево поворачивается на ребре, соединяющем P и его родителя G, а затем снова поворачивается на ребре, соединяющем N и P.



Развертывание

Шаг зигзаг

Операция зигзаг выполняется, когда P не является корнем. В дополнение к этому, N является правым потомком P, а P является левым потомком G или наоборот. При шаге зигзаг дерево сначала поворачивается на ребре между N и P, а затем поворачивается на ребре между N и G.

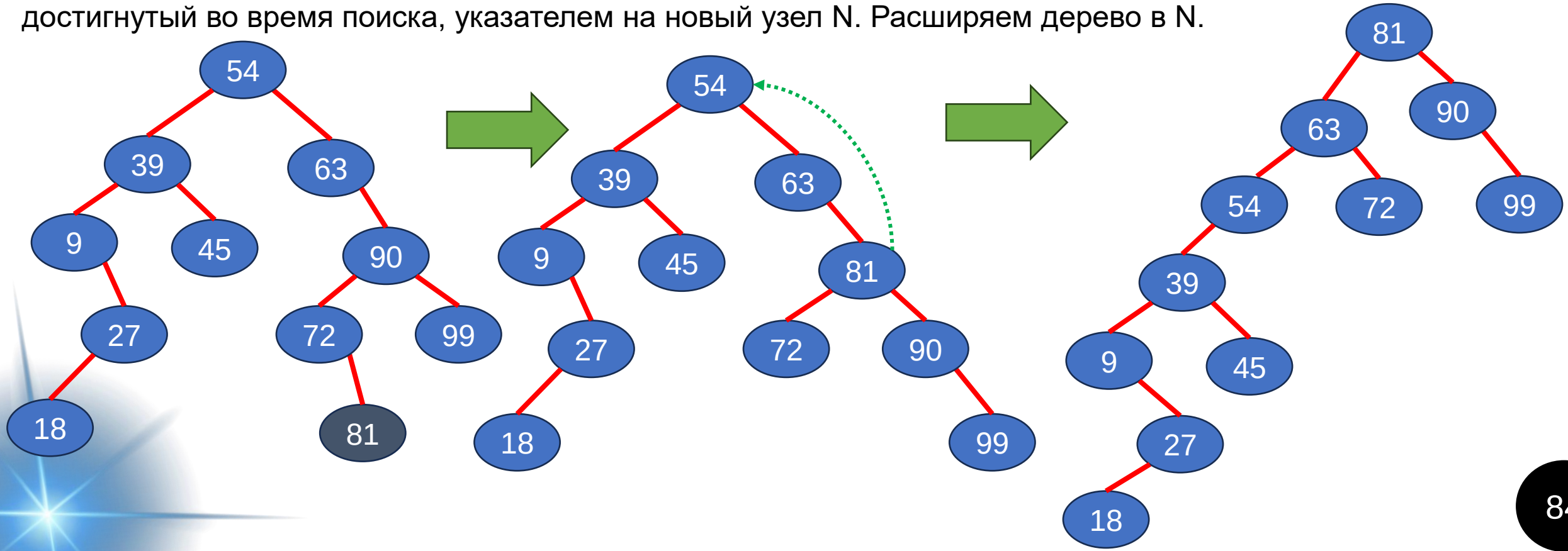


Вставка узла в splay дерево

Хотя процесс вставки нового узла N в расширяющееся дерево начинается так же, как мы вставляем узел в бинарное дерево поиска, но после вставки N становится новым корнем расширяющегося дерева. Шаги, выполняемые для вставки нового узла N в расширяющееся дерево, можно представить следующим образом:

Шаг 1 Поиск N в расширяющемся дереве. Если поиск успешен, расширяем в узле N.

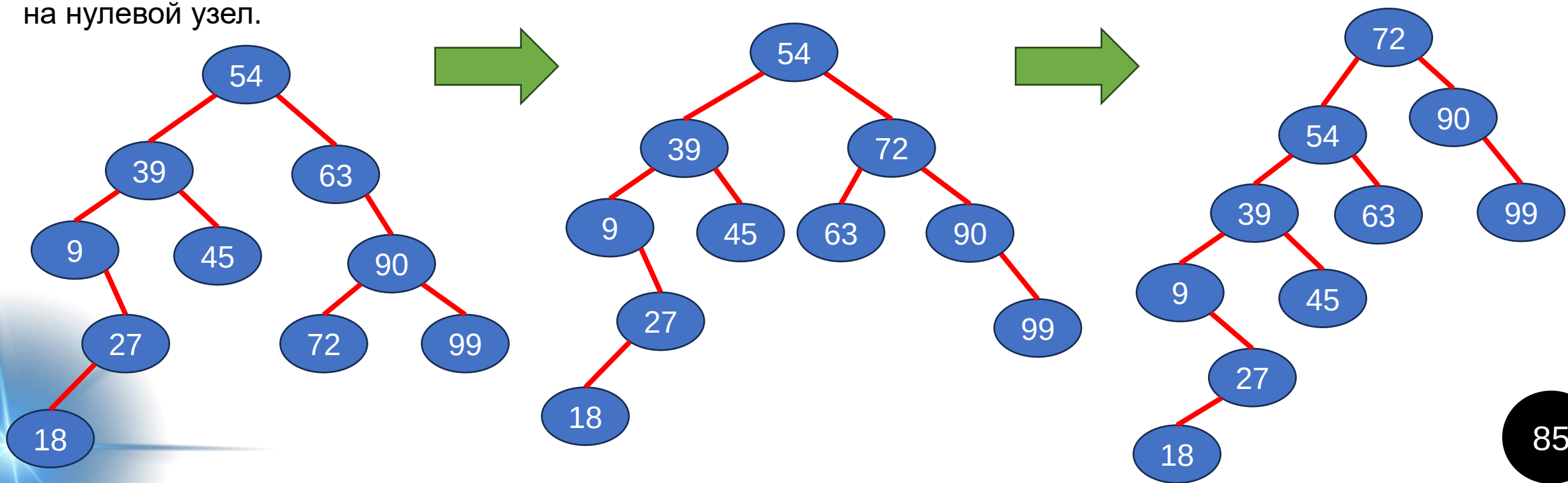
Шаг 2 Если поиск не удался, добавляем новый узел N таким образом, чтобы он заменил указатель NULL, достигнутый во время поиска, указателем на новый узел N. Расширяем дерево в N.



Поиск узла в splay дереве

Если в раскрывающемся дереве присутствует определенный узел N, то возвращается указатель на N; в противном случае возвращается указатель на нулевой узел. Шаги, выполняемые для поиска узла N в раскрывающемся дереве, включают:

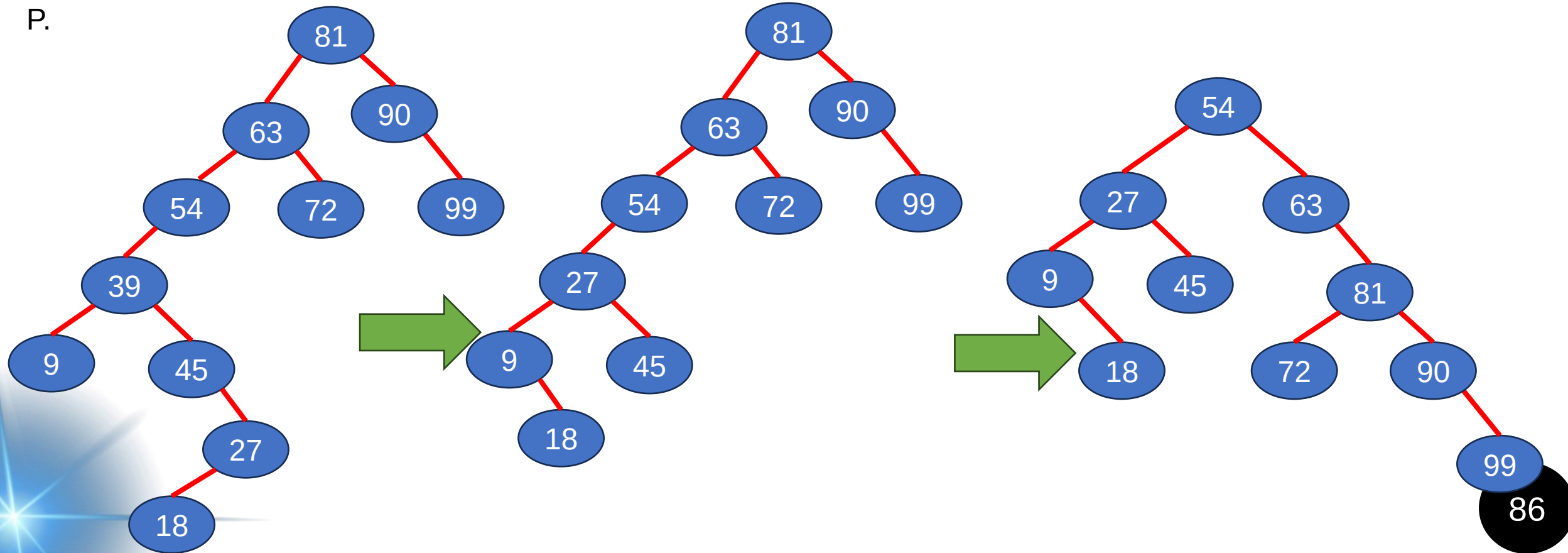
- Просмотрите корень разветвленного дерева, ища N.
- Если поиск успешен и мы достигли N, то растягиваем дерево в N и возвращаем указатель на N.
- Если поиск не удался, т. е. растягиваемое дерево не содержит N, то мы достигли нулевого узла. Растягиваем дерево в последнем ненулевом узле, достигнутом во время поиска, и возвращаем указатель на нулевой узел.



Поиск узла в splay дереве

Чтобы удалить узел N из растягиваемого дерева, мы выполняем следующие шаги:

- Поиск N, который необходимо удалить. Если поиск не удался, растягиваем дерево в последнем ненулевом узле, обнаруженном во время поиска.
- Если поиск успешен и N не является корневым узлом, то пусть P будет родителем N. Замените N соответствующим потомком P (как мы делаем в бинарном дереве поиска). Наконец, растягиваем дерево в P.



Преимущества использования расширяющегося дерева

- Расширяющееся дерево обеспечивает хорошую производительность для операций поиска, вставки и удаления. Это преимущество основано на том факте, что расширяющееся дерево является самобалансирующейся и самооптимизирующейся структурой данных, в которой часто используемые узлы перемещаются ближе к корню, чтобы к ним можно было быстро получить доступ. Это преимущество особенно полезно для реализации кэшей и алгоритмов сборки мусора.
- Расширяющиеся деревья значительно проще в реализации, чем другие самобалансирующиеся бинарные деревья поиска, такие как красно-черные деревья или деревья AVL, в то время как их средняя производительность так же эффективна.
- Расширяющиеся деревья минимизируют требования к памяти, поскольку они не хранят никаких бухгалтерских данных.
- В отличие от других типов самобалансирующихся деревьев, расширяющиеся деревья обеспечивают хорошую производительность (амортизированная O(log n)) с узлами, содержащими идентичные ключи.

Недостатки использования расширяющегося дерева

- При последовательном доступе ко всем узлам дерева в отсортированном порядке результирующее дерево становится полностью несбалансированным. Это требует n доступов к дереву, в которых каждый доступ занимает O(log n) времени. Например, повторный доступ к первому узлу запускает операцию, которая, в свою очередь, занимает O(n) операций для повторной балансировки дерева перед возвратом первого узла. Хотя это создает значительную задержку для конечной операции, амортизированная производительность по всей последовательности по-прежнему составляет O(log n).
- Для равномерного доступа производительность расширяющегося дерева будет значительно хуже, чем несколько сбалансированного простого двоичного дерева поиска. Для равномерного доступа, в отличие от расширяющихся деревьев, эти другие структуры данных обеспечивают гарантии времени в худшем случае и могут быть более эффективными в использовании.

План лекции

Метод Ньютона-Рафсона	Эффективные деревья	B-деревья и приложения (1 часть)
20 минут	70 минут	0 минут