**КФУ, Институт ИТИС, 1 курс, 2 семестр, 2021**

**Семестровая работа №1:**

**"Red-black tree"**

**Выполнила:**

Рыфтина Полина Алексеевна

группа 11-103

**Преподаватель:**

Андреичев М.Д.

*Red-Black tree*

**Красно-чёрное дерево** (англ. red-black tree, RB tree) — один из видов самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

*NIL Nodes in Red-Black Trees*

В литературе красно-черные деревья изображаются с так называемыми нулевыми узлами и без них. Нулевой узел - это лист, который не содержит значения. Нулевые узлы становятся релевантными для алгоритмов позже, например, для определения цветов узлов «дяди» или «брата».

В Java нулевые узлы могут быть представлены просто нулевыми ссылками.

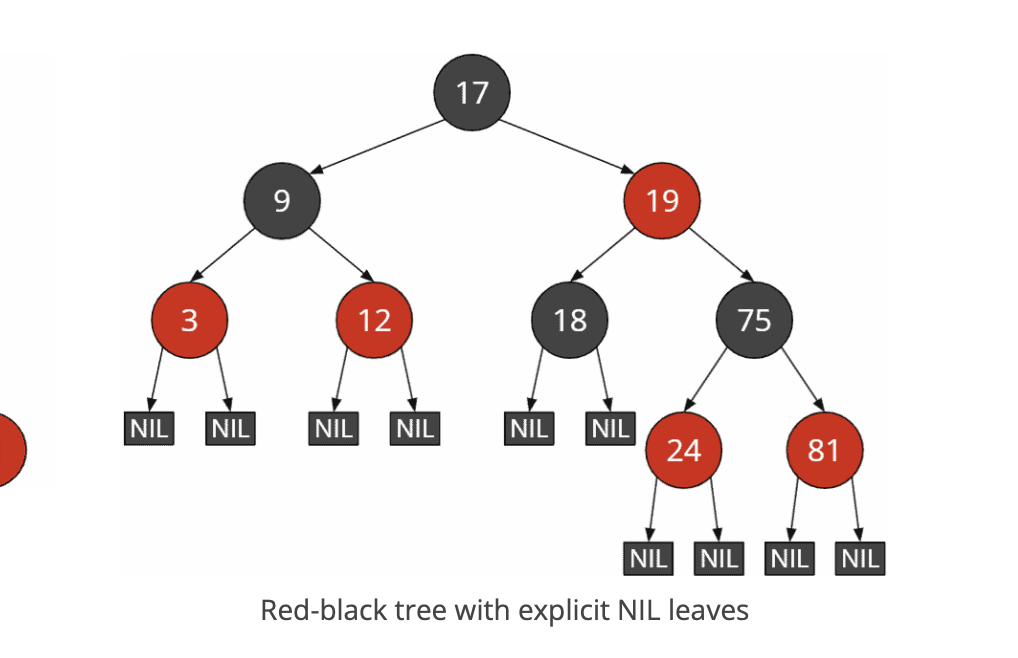
*Историческая справка*

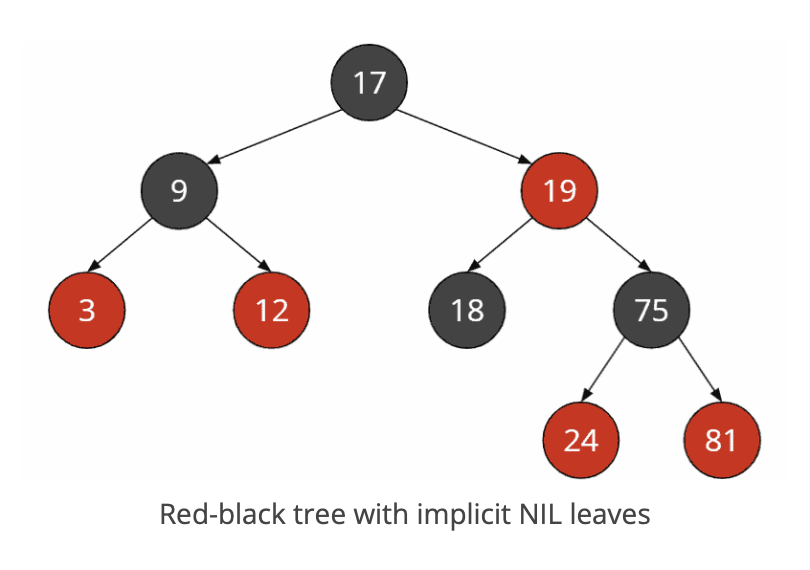
Изобретателем красно-чёрного дерева считают немца Рудольфа Байера. Название «красно-чёрное дерево» структура данных получила в статье Л. Гимбаса и Р. Седжвика (1978). По словам Гимбаса, они использовали ручки двух цветов. По словам Седжвика, красный цвет лучше всех смотрелся на лазерном принтере.

Красно-чёрное дерево используется для организации сравнимых данных, таких как фрагменты текста или числа. Листовые узлы красно-чёрных деревьев не содержат данных, благодаря чему не требуют выделения памяти — достаточно записать в узле-предке в качестве указателя на потомка нулевой указатель. Однако в некоторых реализациях для упрощения алгоритма могут использоваться явные листовые узлы.

*Основные принципы и устройство красно-черного дерева*

1. Каждый узел либо черный, либо красный и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все NIL листы – черные;
4. Красный узел не может иметь красных потомков;
5. Все пути от узла к нижеприведенным листьям содержат одинаковое количество черных узлов.

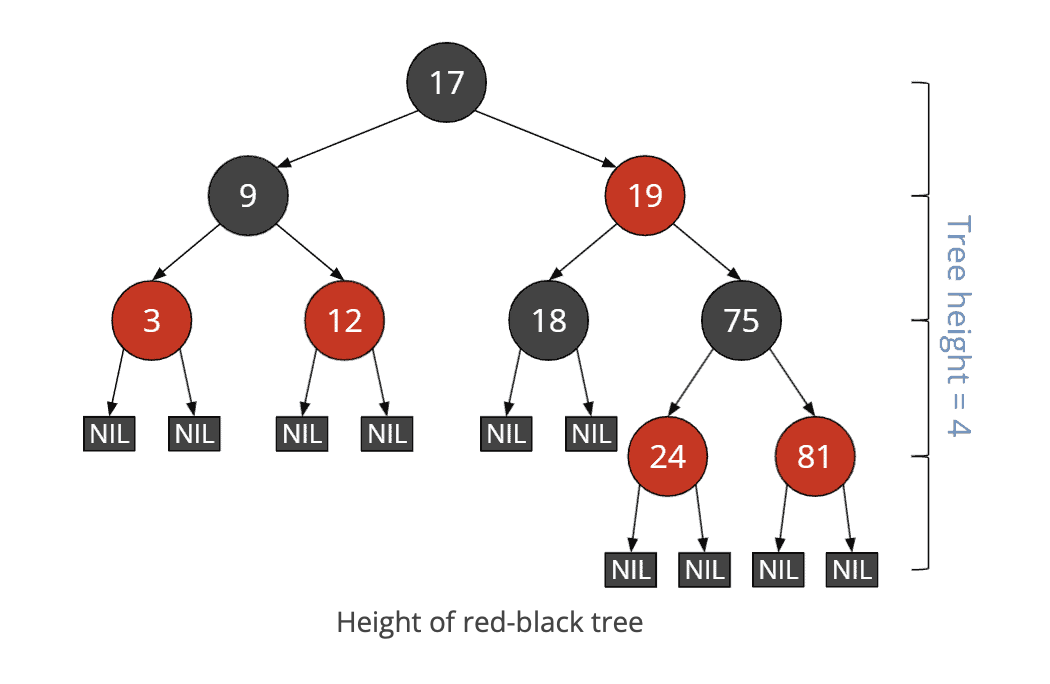
В следующем примере показаны два возможных представления красно-черного дерева. На первом изображении показано дерево без (т.е. с неявными) нулевыми листьями; на втором изображении показано дерево с явными нулевыми листьями.



Благодаря этим ограничениям путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего, и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

*Высота красно-черного дерева*

Мы ссылаемся на высоту красно-черного дерева как на максимальное количество узлов от корня до нулевого листа, не включая корень. Высота красно-черного дерева в приведенном выше примере равна 4:

Самый длинный путь от корня до листа (не считая корня) не более чем в два раза длиннее кратчайшего пути от корня до листа.

Это легко объяснить:

Давайте предположим, что кратчайший путь имеет (в дополнение к корню) n черных узлов и ни одного красного узла. Затем мы могли бы добавить еще n красных узлов перед каждым черным узлом, не нарушая правила 3 (которое мы могли бы переформулировать так: никакие два красных узла не могут следовать друг за другом).

*Черная высота красно-черного дерева*

Черная высота – это количество черных узлов от данного узла до его листьев. Черные нулевые листья подсчитываются, начальный узел - нет.

Черная высота всего дерева – это количество черных узлов от корня (это не учитывается) до листьев NILE

*Реализация красно-черного дерева (с отсылками на исходный код)*

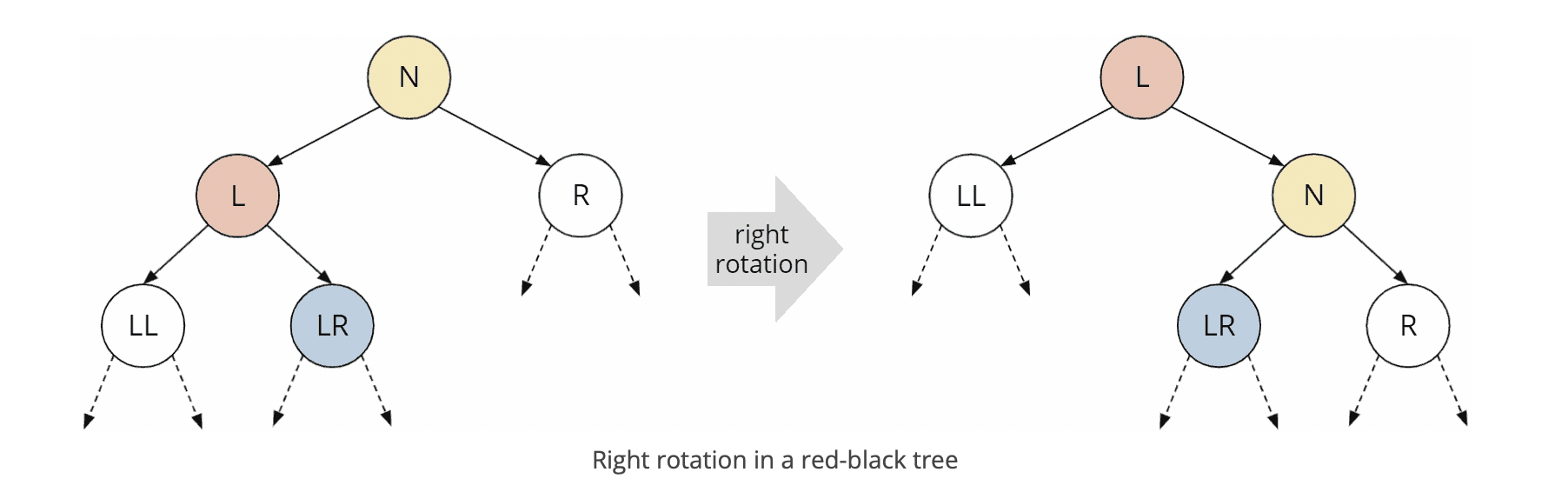
В качестве отправной точки для реализации красно-черного дерева в Java я использую исходный код Java для двоичного дерева поиска.

Узлы представлены классом Node. Для простоты я использую примитивы int в качестве значения узла. Чтобы реализовать красно-черное дерево, помимо дочерних узлов слева и справа, нужны ссылка на родительский узел и цвет узла. Сохраняется цвет в логическом значении, определяемый красным как false, а черным как true.

Реализуется красно-черное дерево в классе RedBlackTree. Этот класс расширяет базовый класс BinaryTree (который предоставляет функцию getRoot()). Добавляются операции: вставка, поиск, удаление.

*Вращение красного-черного дерева*

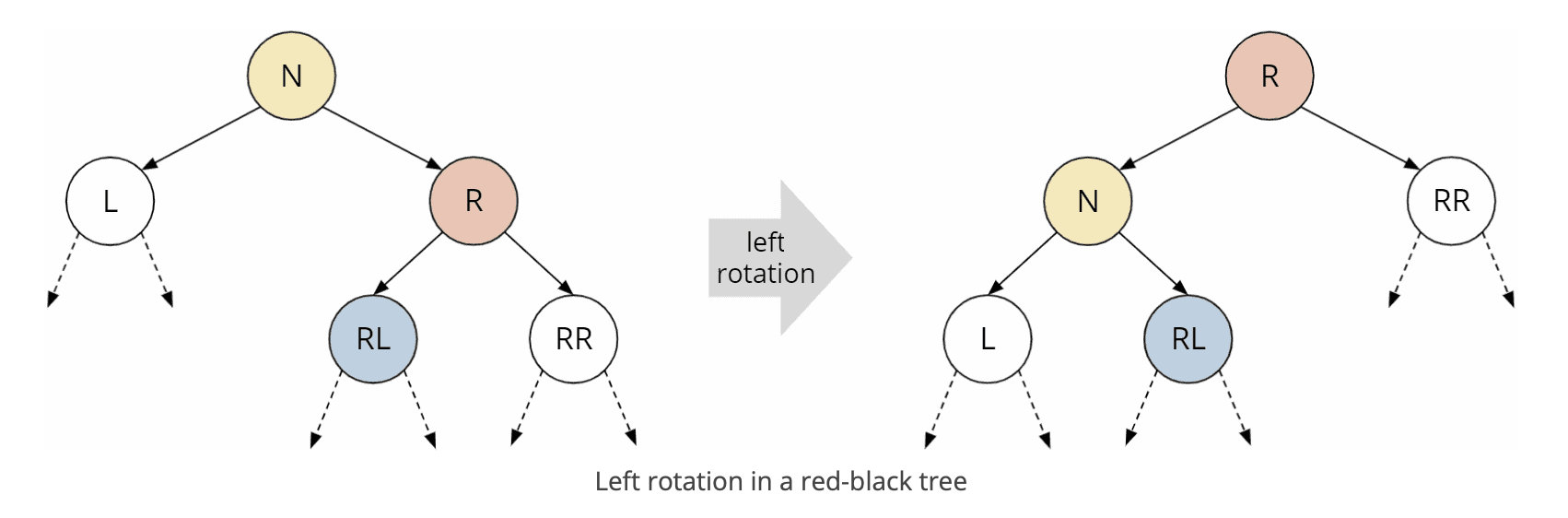
* *Правый поворот*

**

На следующем рисунке показан поворот вправо. Цвета не имеют никакого отношения к цветам красно-черного дерева. Они используются только для лучшего отслеживания перемещений узлов.

Левый узел L становится новым корнем; корень N становится его правым дочерним узлом. Правый дочерний элемент LR левого узла L после поворота становится левым дочерним элементом правого узла N. Два белых узла LL и R не меняют своего взаимного положения. Также необходимо обновить родительские ссылки узлов. Метод replaceParentsChild(), вызываемый в конце, устанавливает связь родитель-потомок между родительским узлом бывшего корневого узла N повернутого поддерева и его новым корневым узлом L.

* *Левый поворот*



Поворот влево работает аналогично: правый узел R перемещается наверх. Корень N становится левым дочерним элементом R. Левый дочерний элемент RL ранее правого узла R становится правым дочерним элементом левого узла N. L и RR после поворота не меняют своего взаимного положения.

*Операции с Красно-Черным деревом*

Как и любое двоичное дерево, красно-черное дерево предоставляет операции по поиску, вставке и удалению узлов.

1. *Поиск*

Поиск работает так же, как в любом двоичном дереве поиска: сначала сравнивается ключ поиска с корнем. Если ключ поиска меньше, продолжается поиск в левом поддереве; если ключ поиска больше, продолжается поиск в правом поддереве.

Повторяем это до тех пор, пока либо не найдем искомый узел, либо пока не достигнем нулевого листа. Достижение нулевого листа означало бы, что ключ, который ищем, не существует в дереве*.*

В исходном коде реализован итеративный вариант поиска.

1. *Вставка*

Чтобы вставить новый узел, находится позиция вставки от корня вниз и прикрепляется новый узел к листу или полулисту. Сначала новый узел окрашивается в красный цвет, чтобы соблюдалось правило 5, т.е. все пути имеют одинаковое количество черных узлов после вставки.

Однако, если родительский узел вставленного узла также красный, нарушается правило 4. Затем необходимо восстановить дерево, перекрасив и / или повернув его так, чтобы все правила снова были выполнены. Это делается функцией fixRedBlackPropertiesAfterInsert(), которая вызывается в последней строке метода insertNode().

Во время восстановления основных принципов деревана возникают пять различных случаев:

Случай 1: Новый узел является корневым;

Случай 2: Родительский узел - это корень, окрашенный красным цветом;

Случай 3: Родительский и дочерний узлы окрашены в красный цвет;

Случай 4: Родительский узел красный, дядя узел черный, вставленный узел - "внутренний внук".

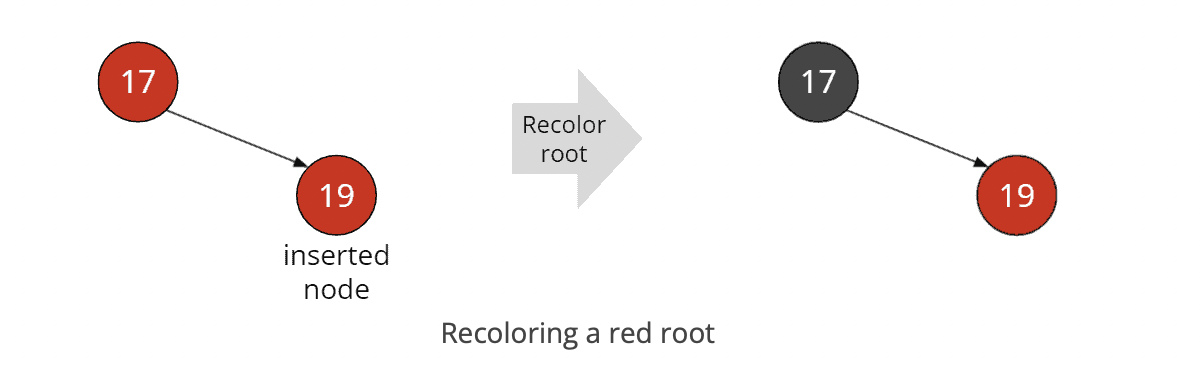
Случай 5: Родительский узел красный, дядя узел черный, вставленный узел - "внешний внук".

Эти пять случаев описаны ниже.

* **Новый Узел Является Корневым**

Если новый узел является корневым, больше ничего не нужно делать. Только учитывая правило 2, нужно перекрасить его в черный.

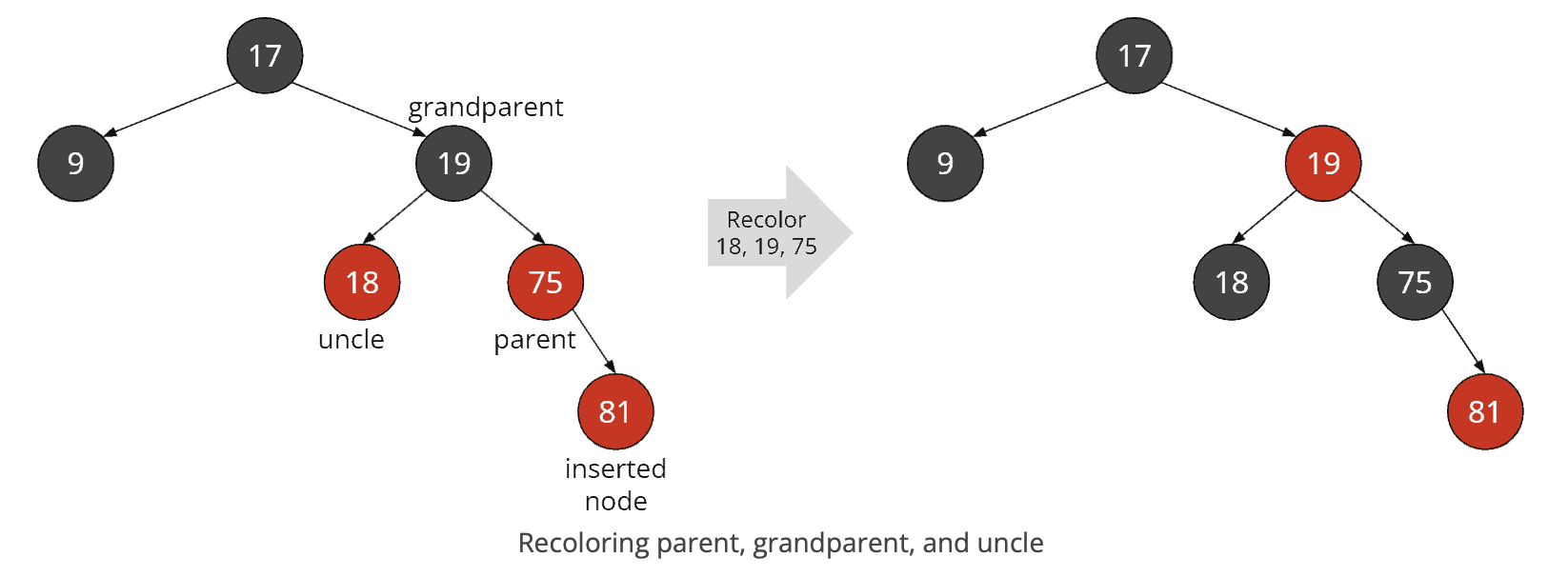
* **Родительский узел – это корень, окрашенный красным цветом;**



В этом случае нарушается правило 4 (красный узел не может иметь красных потомков). Необходимо перекрасить корень в черный цвет.

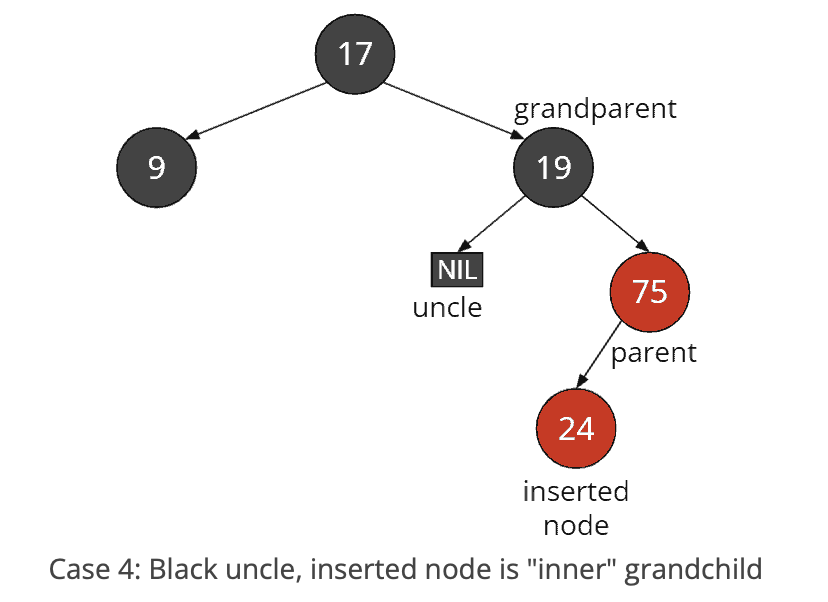
* **Родительский и дочерний узлы окрашены в красный цвет;**

И родитель, и дядя красные. В этом случае:



Перекрашиваются родительский и узел дяди (18 и 75 в примере) в черный цвет, а дедушка (19) - в красный. Таким образом, правило 4 (красный узел не может иметь красных потомков) снова выполняется на вставленном узле. Количество черных узлов на пути не меняется (в примере оно остается равным 2 с учетом корня).

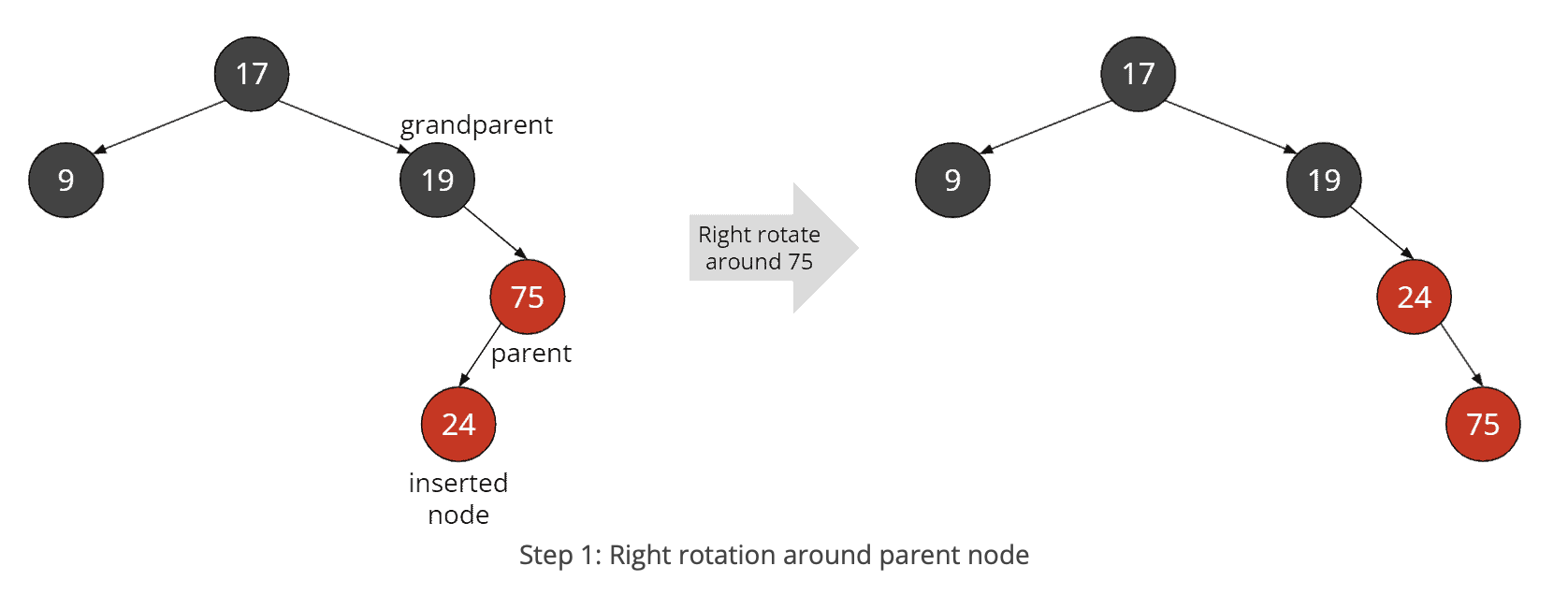
* **Родительский узел красный, дядя узел черный, вставленный узел - "внутренний внук".**

**

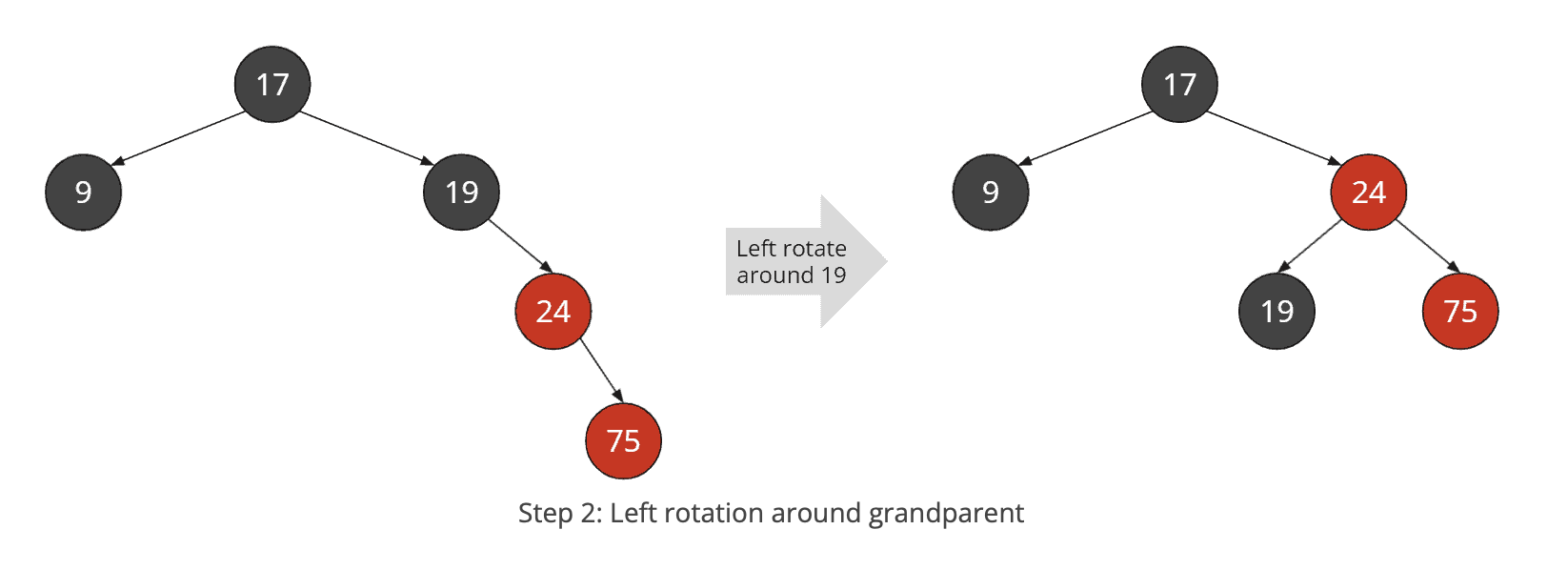
Объяснение: "внутренний внук" означает, что путь от узла дедушки к вставленному узлу образует треугольник, как показано на следующем рисунке с использованием 19, 75 и 24. В этом примере можно увидеть, что NIL лист также считается черным дядей (согласно правилу 3). Для ясности не показаны оба нулевых листа у 9 и 24, а также правый NIL лист у 75.

В этом случае мы сначала поворачиваем родительский узел в направлении, противоположном направлению вставленного узла. (Если вставленный узел является левым дочерним элементом своего родительского узла, мы поворачиваем вправо на родительском узле. Если вставленный узел является правым дочерним узлом, мы поворачиваем влево).

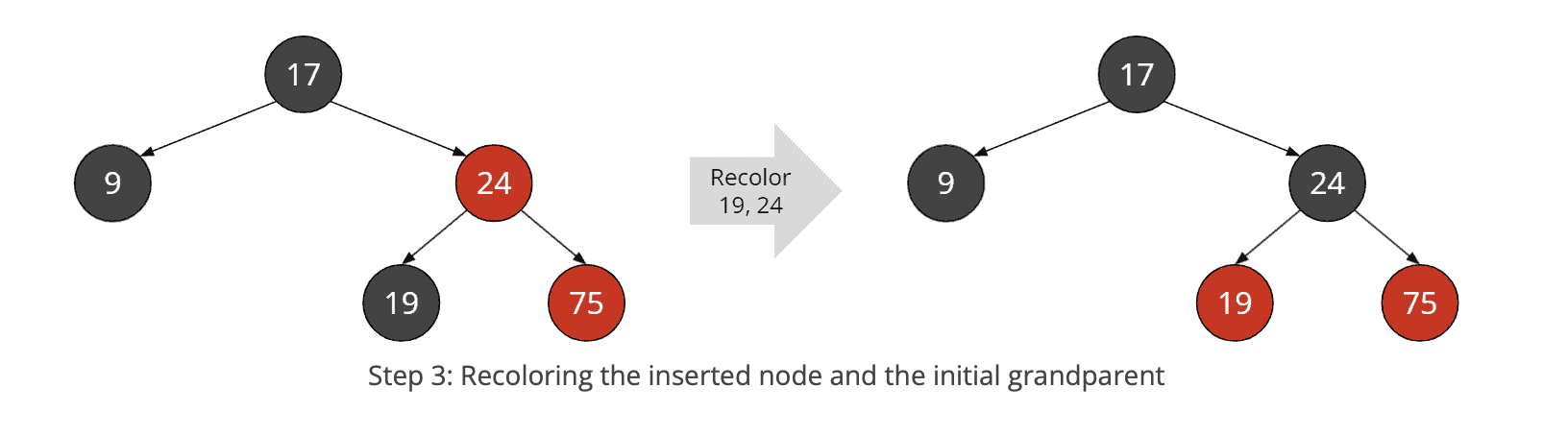
В примере вставленный узел (24) является левым дочерним узлом, поэтому поворачиваем вправо родительский узел (75):



Вторым шагом вращаем в узле дедушки в направлении, противоположном предыдущему вращению. В приведенном примере поворачиваем влево вокруг 19:

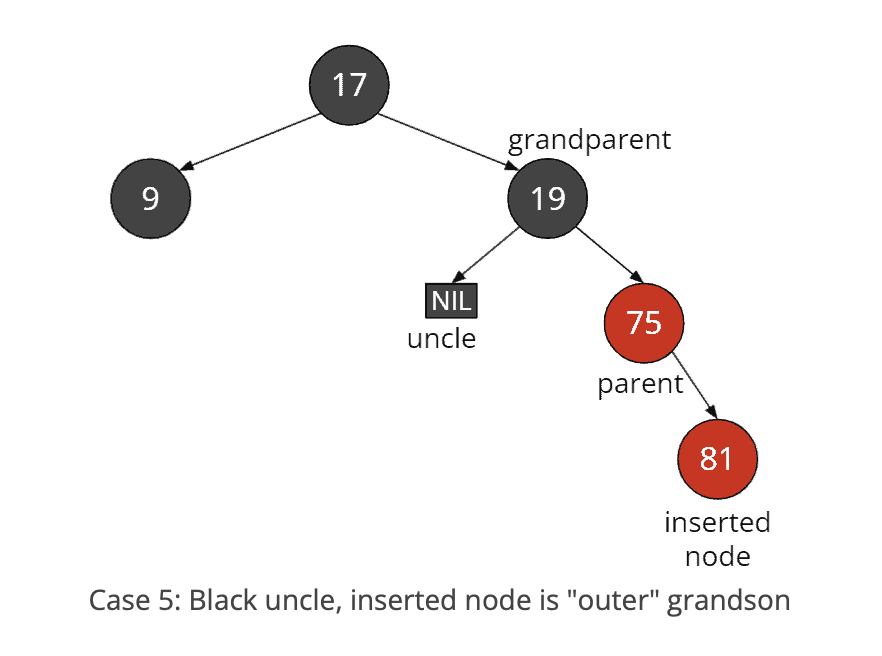
**

Третьим шагом окрашиваем узел, который только что вставили (24), в черный цвет, а исходного дедушку (19) - в красный:

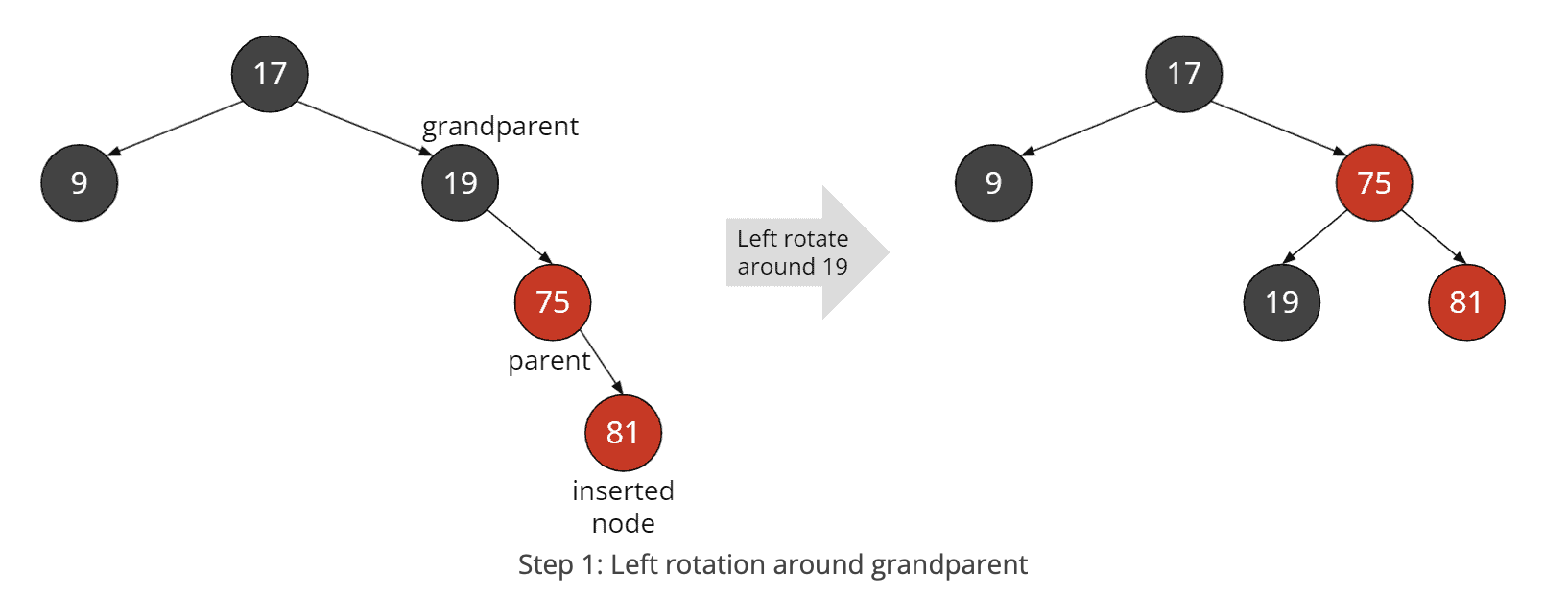


Пример соблюдает все принципы красно-черного дерева.

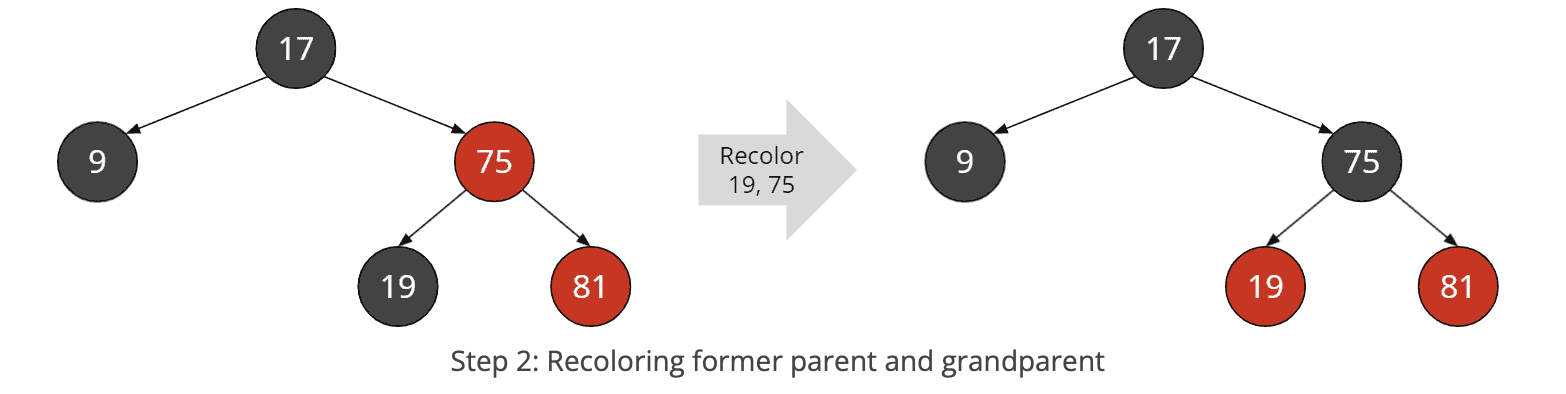
* **Родительский узел красный, дядя узел черный, вставленный узел - "внешний внук".**

"Внешний внук" означает, что путь от дедушки к вставленному узлу образует линию, такую как 19, 75 и 81 в следующем примере**.

В этом случае мы поворачиваем узел дедушки (19 )в направлении, противоположном родительскому и вставленному узлу. В примере родительский и вставленный узлы являются правыми дочерними узлами, поэтому мы поворачиваем дедушку влево:

**

Затем мы перекрашиваем исходного родителя (75 ) в черный цвет, а исходного дедушку (19) в красный:

**

1. *Удаление*

Основные случаи:

* Если узел, подлежащий удалению, не имеет дочерних элементов, мы просто удаляем его.
* Если узел, подлежащий удалению, имеет одного дочернего элемента, мы удаляем узел и позволяем его единственному дочернему элементу переместиться на его место.
* Если узел, подлежащий удалению, имеет двух дочерних элементов, мы копируем содержимое (не цвет!) преемника по порядку элементов в узел, подлежащий удалению, а затем удаляем этого преемника в соответствии с правилом 1 или 2 (этот преемник имеет не более одного сына по определению).

После этого нужно проверить правила дерева и при необходимости исправить его. Чтобы сделать это, нужно запомнить цвет удаленного узла и какой узел был перемещен вверх.

Если удаленный узел красный, то ни одно из правил не нарушится;

Если удаленный узел черный, гарантированно нарушается правило 5 (если только дерево не содержало ничего, кроме черного корня), и, возможно, также было нарушено правило 4, а именно, если оба родительских узла и перемещенный дочерний узел удаленного узла были красными.

При удалении мы должны учитывать на один случай больше, чем при вставке. В отличие от вставки, здесь важен не цвет дяди, а цвет брата удаленного узла.

Случай 1: Удаленный узел является корневым;

Случай 2: Брат красного цвета;

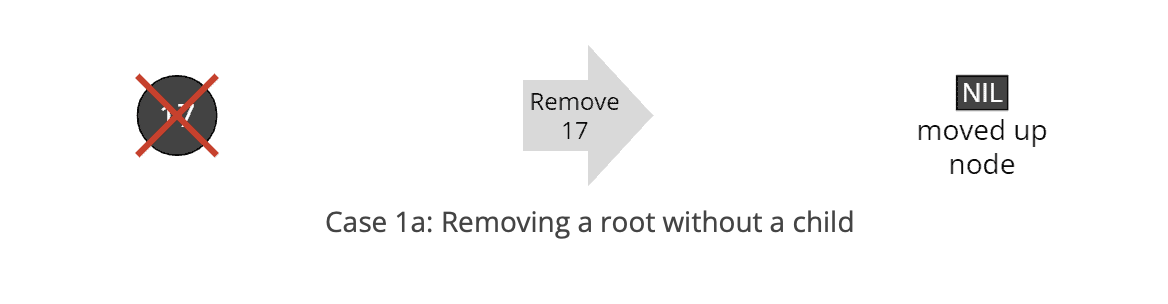
Случай 3: Брат черного цвета и имеет двух черных потомков, родитель красный;

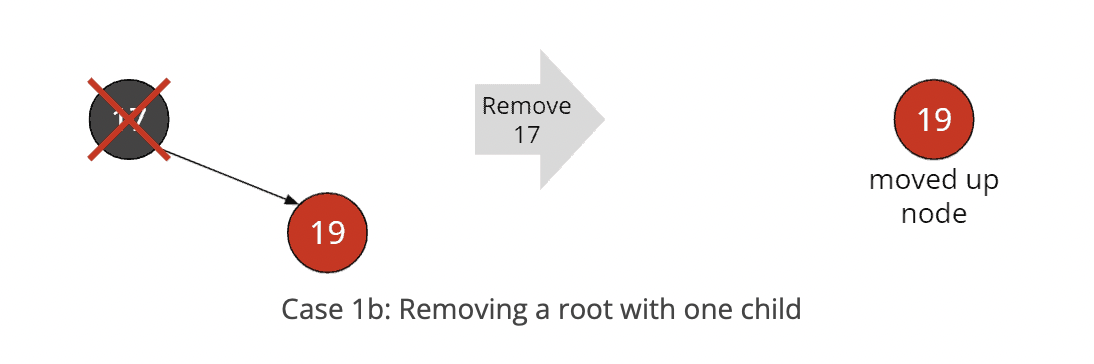
Случай 4: Брат черного цвета и имеет двух черных потомков, родитель черный;

Случай 5: Брат черного цвета и имеет по крайней мере одного красного ребенка, "внешний племянник" черный;

Случай 6: Брат черного цвета и имеет по крайней мере одного красного ребенка, "внешний племянник" красный.

* **Удаленный узел является корневым;**

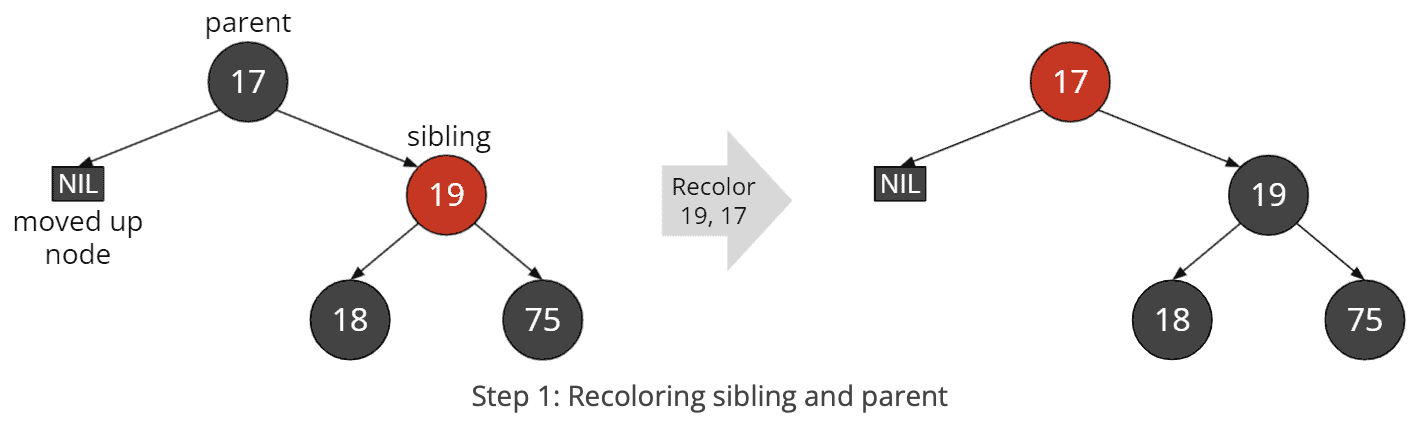




* **Брат красного цвета;**

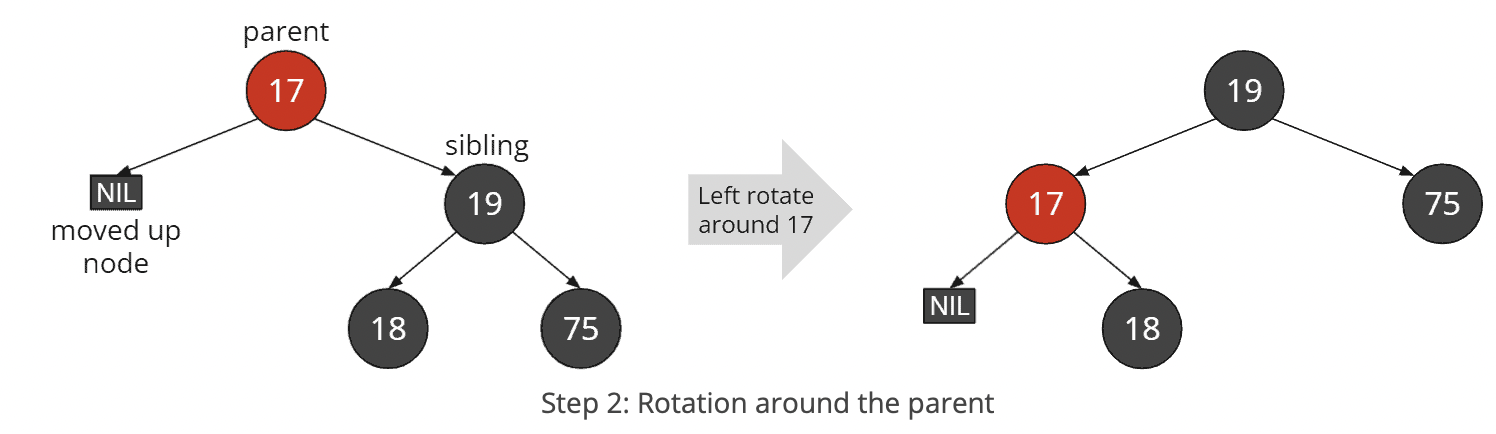


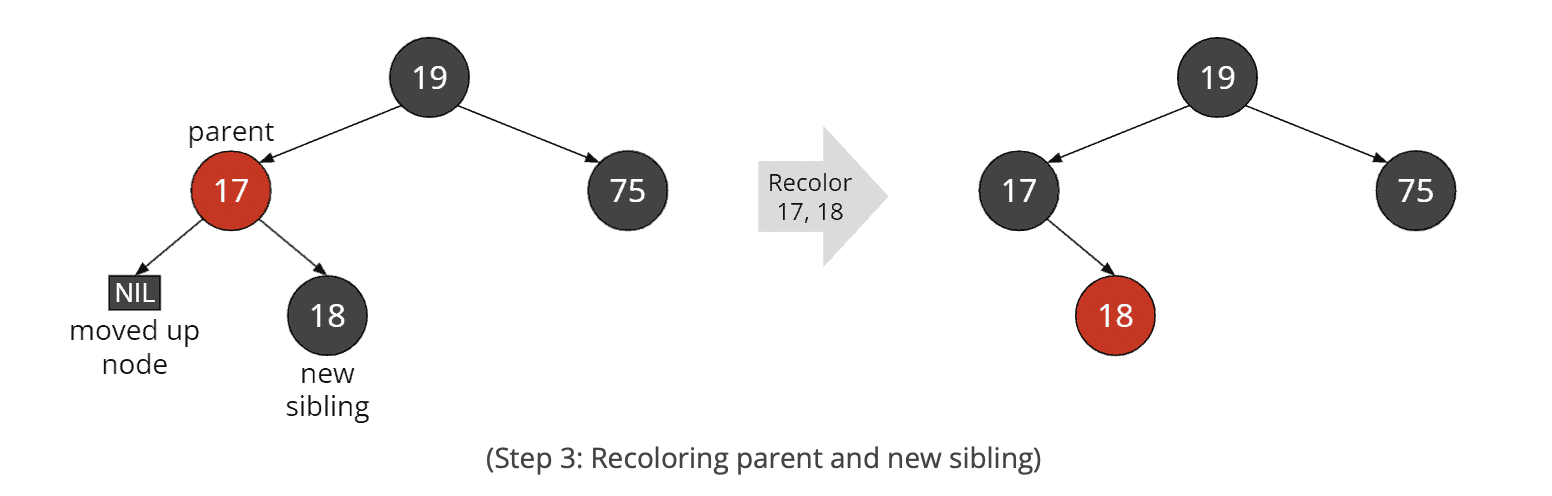
В этом случае мы сначала окрашиваем брата в черный цвет, а родителя - в красный:



Это явно нарушало правило 5: пути в правом поддереве родительского элемента имеют на два черных узла больше, чем пути в левом поддереве. Мы исправляем это, вращаясь вокруг родительского узла в направлении удаленного узла.

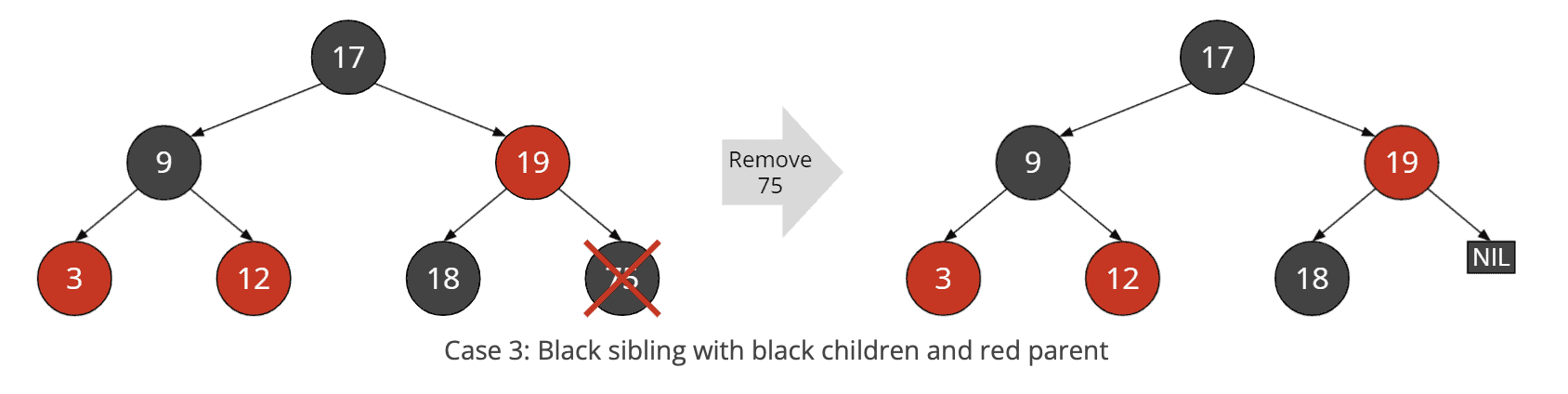
В примере мы удалили левый узел родительского узла – поэтому мы выполняем поворот влево:





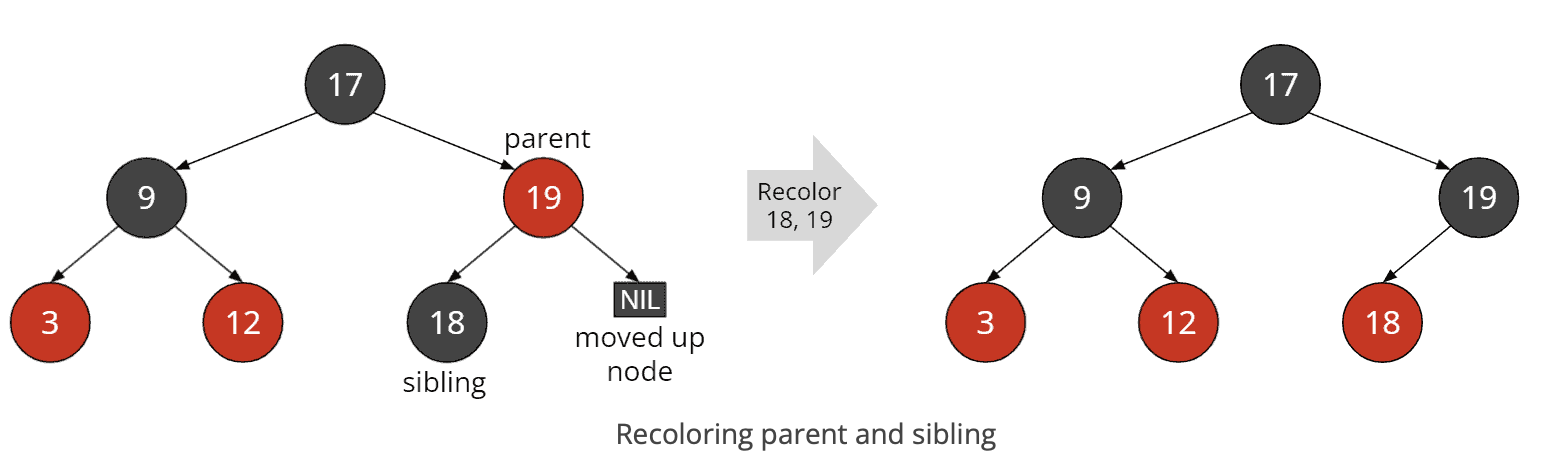
Теперь все пути имеют два черных узла; учтены все принципы красно-черного дерева.

* **Брат черного цвета и имеет двух черных потомков, родитель красный;**



В этом случае восстанавливаем дерево следующим образом:

Перекрашиваем родного брата (18) в красный цвет, а родителя (19) - в черный:



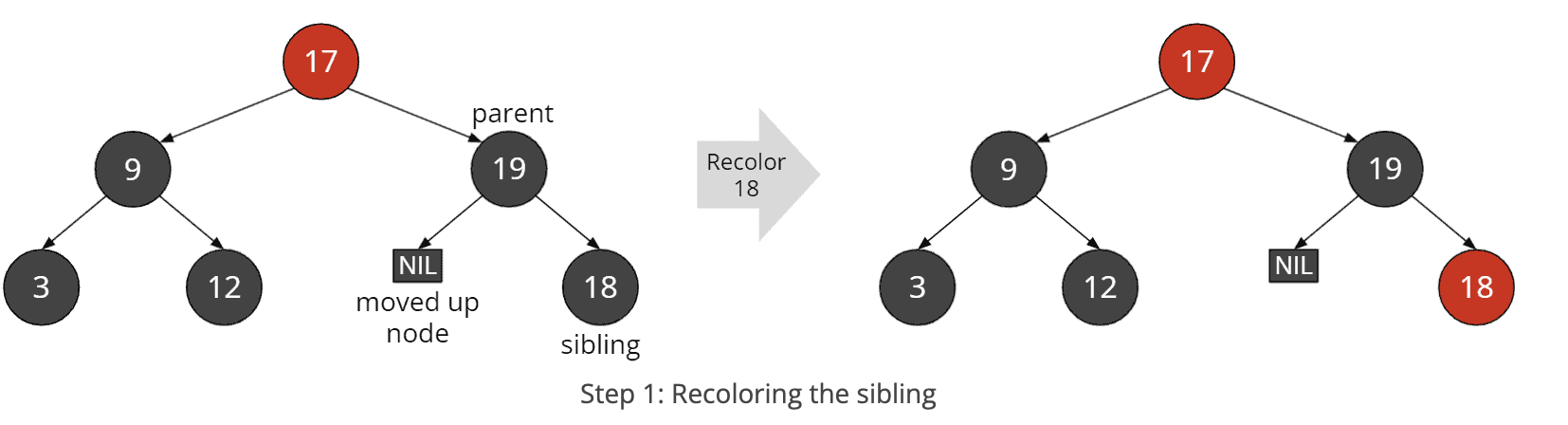
Таким образом, у нас снова есть действительное красно-черное дерево. Количество черных узлов одинаково на всех путях (как того требует правило 5). А поскольку у брата есть только черные дочерние узлы, окрашивание их в красный цвет не может нарушать правило 4.

* **Брат** **черного цвета и имеет двух черных потомков, родитель черный;**

****

Это приводит (как и в случае 3) к нарушению правила 5: на пути к удаленному узлу теперь на один черный узел меньше, чем на всех остальных путях.

В отличие от случая 3, в этом случае родительский узел удаленного узла черный. Сначала окрашиваем родного брата в красный цвет:

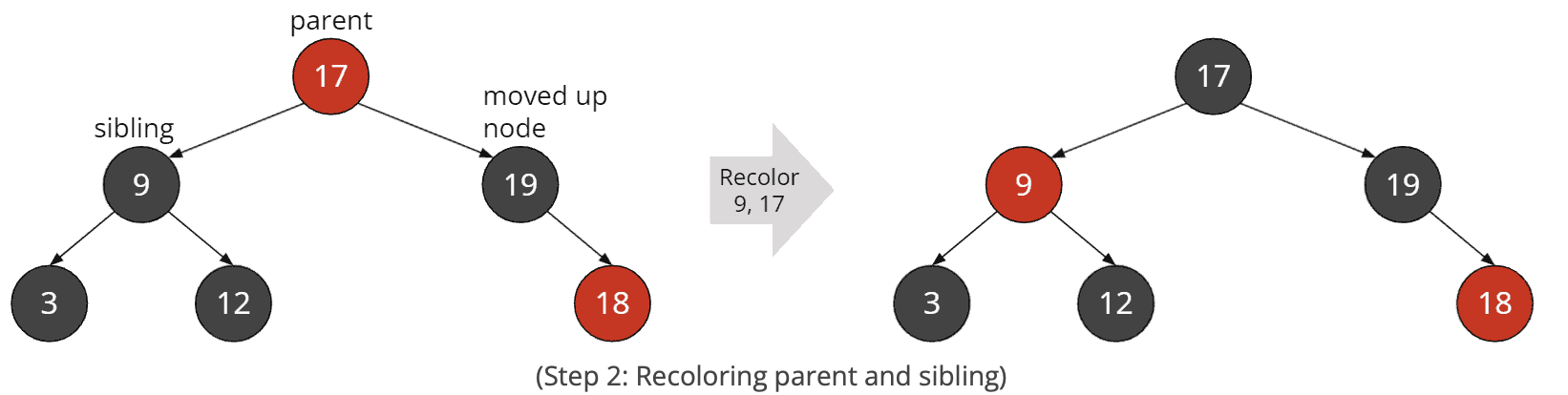


Это означает, что черная высота в поддереве, начинающемся с родительского узла, снова одинакова. Однако в левом поддереве она на единицу выше. Таким образом, правило 5 по-прежнему нарушается.

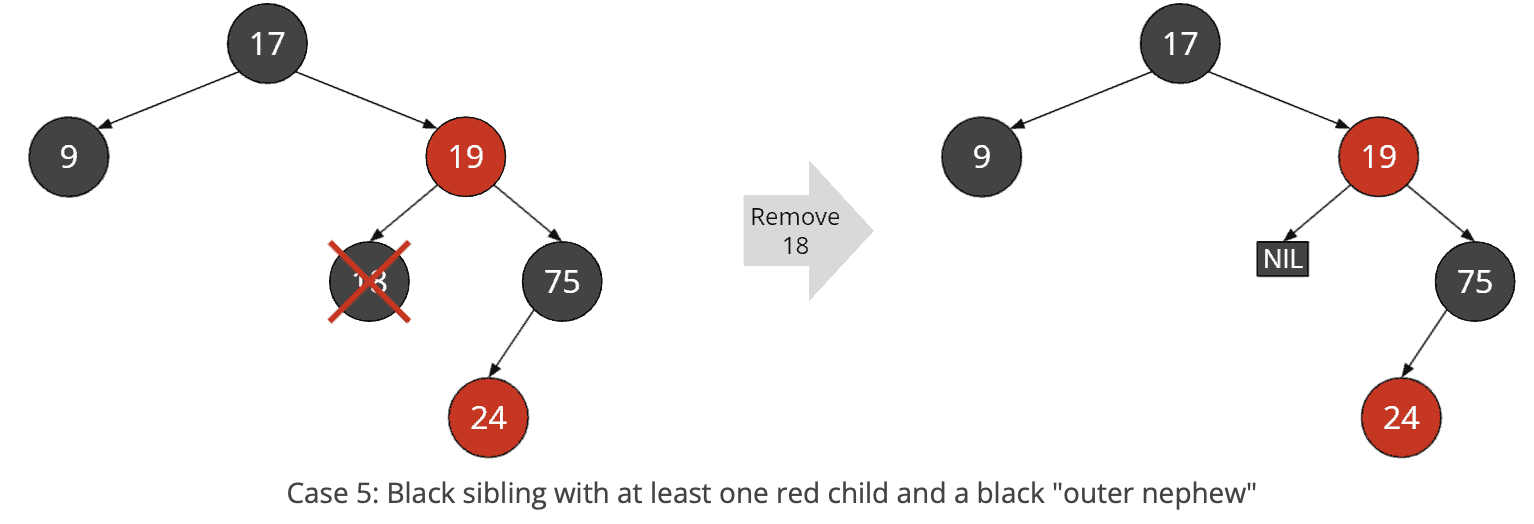
Решается эта проблема «удалением» черного узла между узлами 17 и 19 (что на самом деле не так, но имело бы тот же эффект). Соответственно, рекурсивно вызывается функция восстановления на родительском узле, то есть на 19 (который в данном случае был бы перемещенным узлом).

У 19 есть черный брат (9) с двумя черными потомками (3 и 12) и красным родителем (17). Соответственно, теперь мы возвращаемся к случаю 3.

Мы решаем случай 3, раскрашивая родительский узел в черный цвет, а брата - в красный:



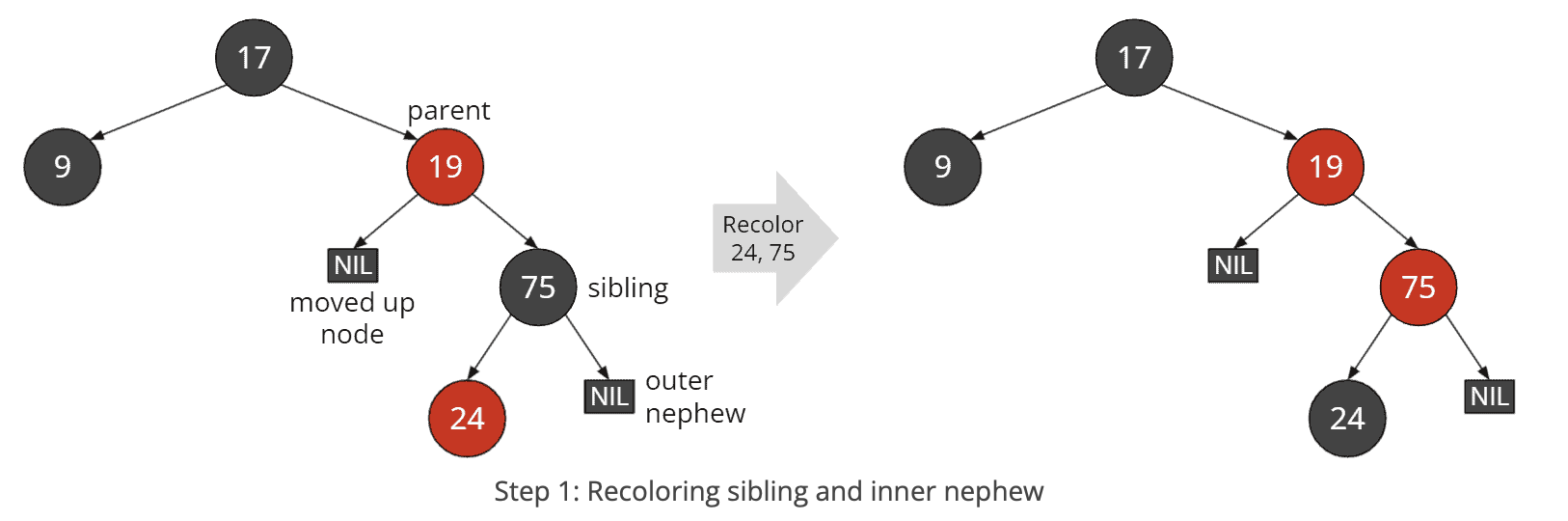
* **Брат черного цвета и имеет по крайней мере одного красного ребенка, "внешний племянник" черный;**

****

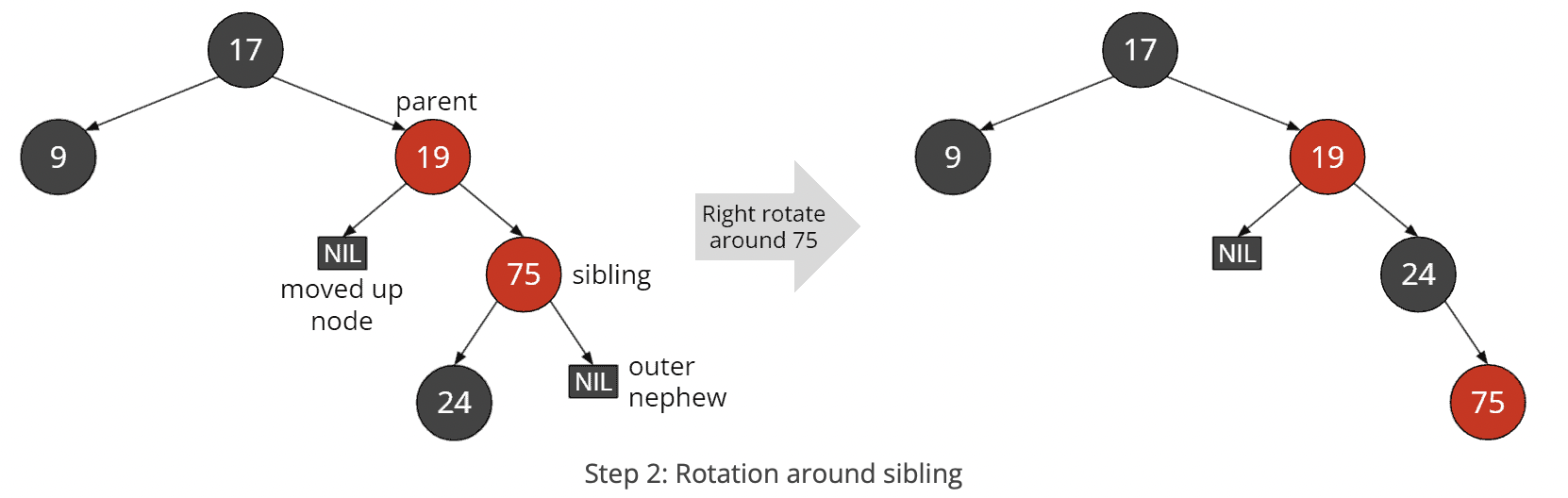
В результате мы снова нарушили правило 5, поскольку у поддерева черная высота больше на единицу. Рассмотрим "внешнего племянника" удаленного узла. "Внешний племянник" – потомок брата, который находится напротив удаленного узла. В примере это правый (и по определению черный) нулевой лист под номером 75.

На следующем рисунке вы можете видеть, что родитель, брат и племянник вместе образуют линию (в примере: 19, 75 и его правый нулевой дочерний элемент).

Восстанавливаем, начиная с окрашивания внутреннего племянника (24 в примере) в черный цвет, а брата (75) - в красный:

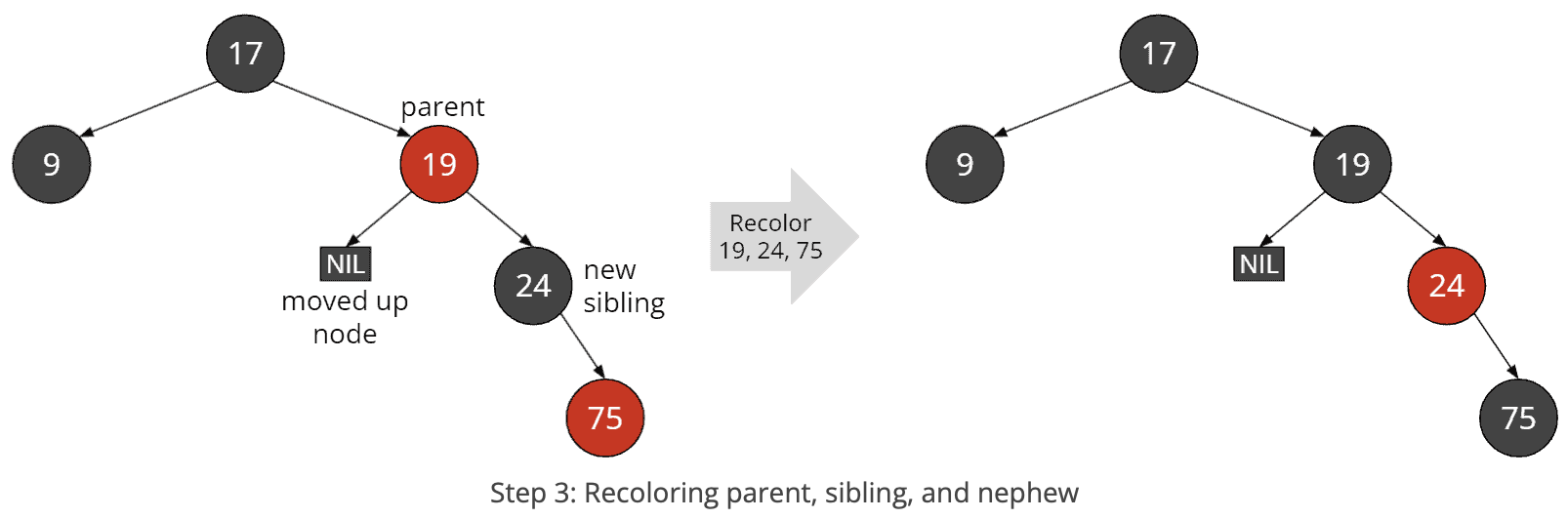


Затем мы выполняем поворот на узле брата удаленного узла в противоположном направлении. В приведенном примере удалили левый дочерний элемент родительского элемента, поэтому выполняем поворот вправо у родственного элемента (75):

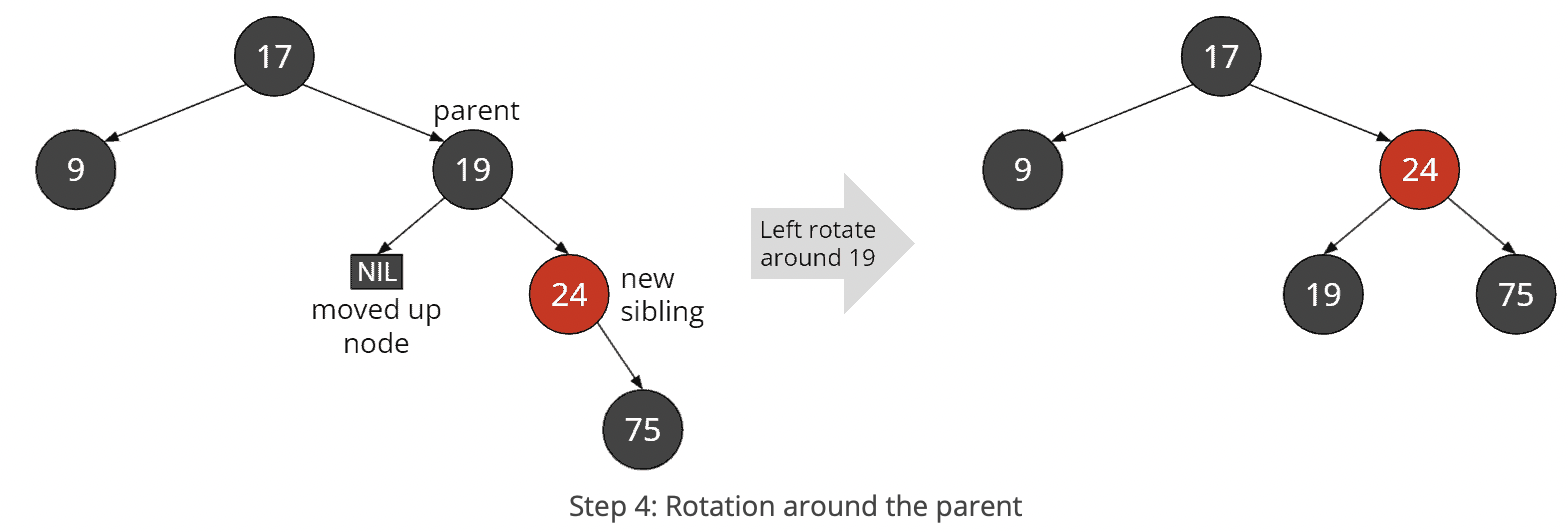
**

Перекрашиваем:

* родного брата в цвет его родителя (в примере: 24 красных).
* Затем родительский (19) и внешний дочерний узел удаленного узла, т.е. правый дочерний элемент нового родственного узла (75 в примере) в черный цвет:

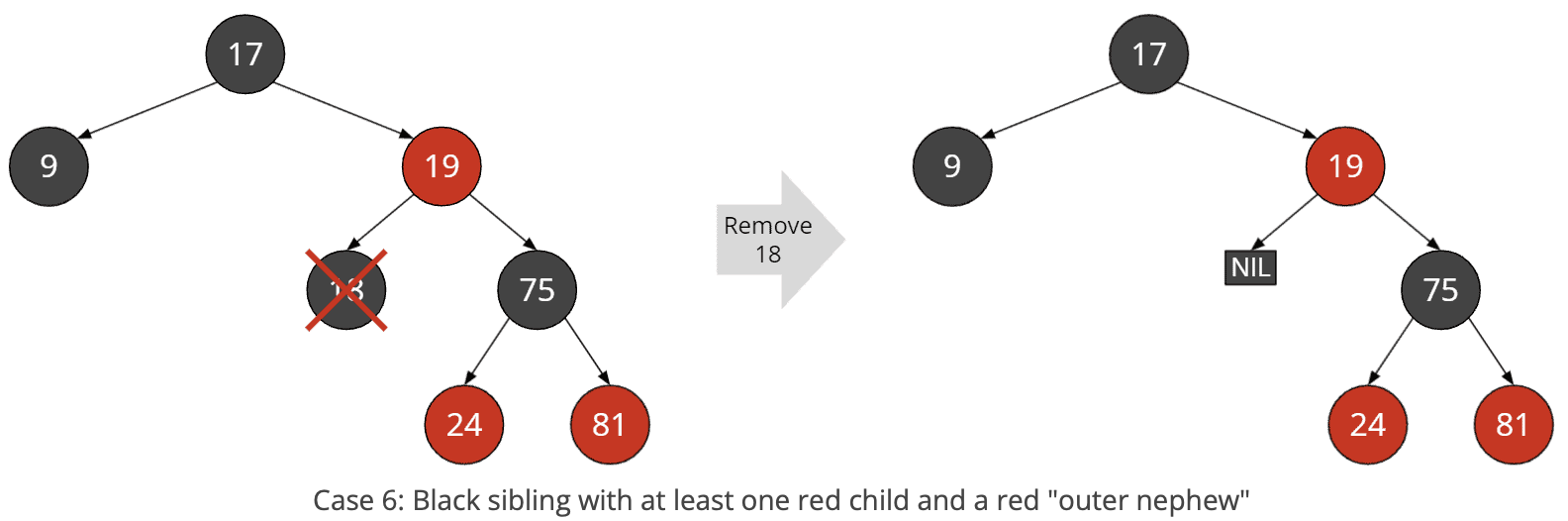


Наконец, выполняется поворот родительского узла в направлении удаленного узла. В примере удаленный узел был левым дочерним узлом, поэтому выполняется поворот влево (в примере на 19):



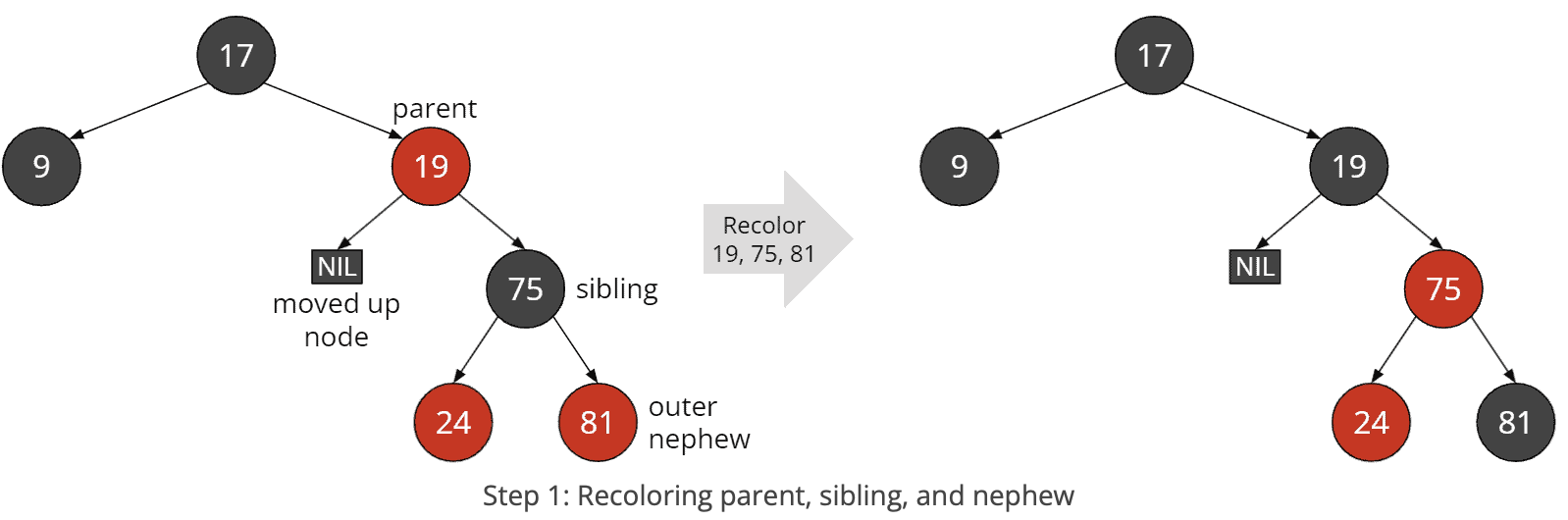
Этот последний шаг восстанавливает соблюдение всех красно-черных правил.

* **Брат черного цвета и имеет по крайней мере одного красного ребенка, "внешний племянник" красный.**

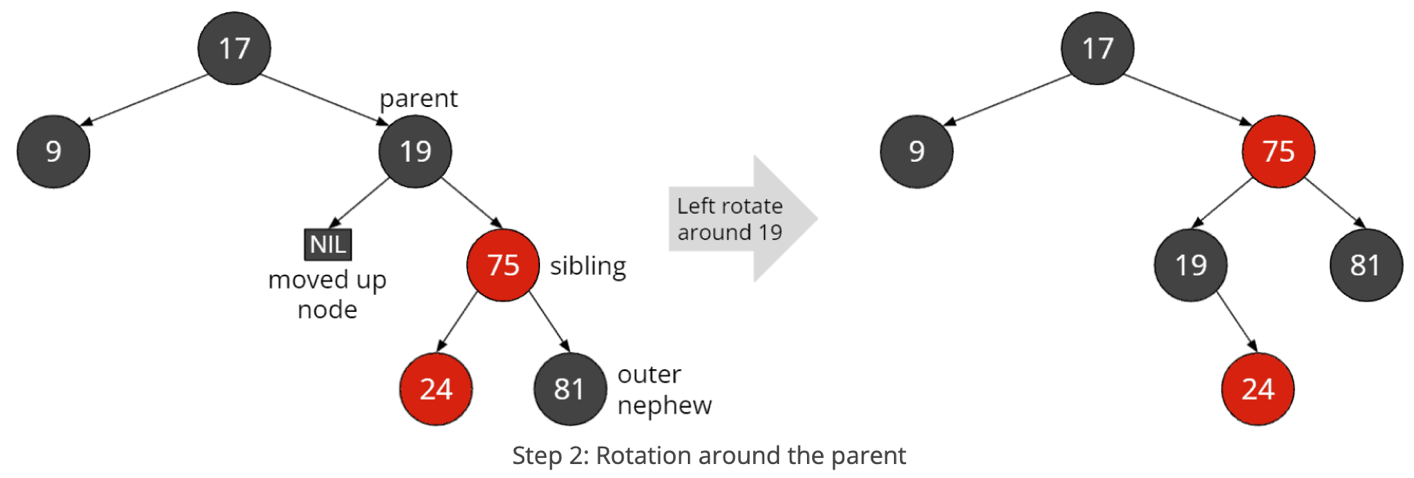
****

В результате, как и в случае 5, мы нарушили правило 5, потому что путь к удаленному узлу теперь содержит на один черный узел меньше.

В случае 6, в отличие от случая 5, внешний цвет (81 в примере) красный, а не черный.

**

Вторым шагом выполняется поворот на родительском узле в направлении удаленного узла. В примере удален левый дочерний элемент; соответственно, выполняется поворот влево вокруг 19:

**

Это вращение восстанавливает красно-черные правила. Никакие два красных узла не следуют друг за другом, и количество черных узлов одинаково на всех путях.

*Оценка времени сложности*

1. *Время поиска*

Операция поиска следует по пути от корня к искомому узлу (или к новому листу). На каждом уровне проводится сравнение. Операция для сравнения постоянна. Таким образом, стоимость поиска пропорциональна высоте дерева. Обозначим через n количество узлов дерева. Самый длинный путь от корня до листа может быть в два раза длиннее самого короткого пути. Отсюда следует, что высота дерева ограничена O(log n). Таким образом, временная сложность поиска узла в красно-черном дереве равна: O(log n).

1. *Время вставки*

При вставке мы сначала выполняем поиск (O(log n)). Далее вставляется узел. Стоимость этого постоянна независимо от размера дерева, поэтому О(1).

Затем проверяются «красно-черные правила», основные принципы устройства дерева, и при необходимости они восстанавливаются. Делается это, начиная со вставленного узла и восходя к корню. На каждом уровне выполняется одна или несколько из следующих операций:

* Проверка цвета родительского узла;
* Определение дяди узла и проверка его цвета;
* Перекрашивание от одного до трех узлов;
* Выполнение одного или двух вращений;

Каждая из этих операций сама по себе имеет постоянное время O(1). Таким образом, общее время проверки и ремонта дерева также пропорционально его высоте. Таким образом, временная сложность вставки в красно-черное дерево также равна: O(log n)

1. *Время удаления*

Так же, как и при вставке, мы сначала ищем узел, который будет удален за время O(log n).

Кроме того, стоимость удаления не зависит от размера дерева, поэтому она постоянна O(1).

Для проверки правил и восстановления дерева выполняется одна или несколько из следующих операций – не более одного раза на уровень:

* Проверка цвета удаленного узла;
* Определение родного брата и изучение его цвета;
* Проверка цвета детей родного брата или сестры;
* Изменение цвета родительского узла;
* Изменение цвета дочернего узла и одного из его дочерних элементов;
* Выполнение одного или двух вращений;

Все эти операции также имеют постоянную сложность сами по себе. Таким образом, общие усилия по проверке и восстановлению правил после удаления узла также пропорциональны высоте дерева.

Таким образом, временная сложность удаления из красно-черного дерева также равна: O(log n).

*Доказательство асимптотических границ\**

Красно-чёрное дерево, которое содержит *n* внутренних узлов, имеет высоту : O(log n)O(log(n)).

Обозначения:

* h(v)*h(v)* — высота поддерева с корнем в v*v*
* *bh(v)* — число чёрных узлов (не считая *v*, если он черный) от v до любого листа в поддереве (называемое черной высотой);

**Лемма:** Поддерево с корнем в узле *v* имеет не менее внутренний узел.

Доказательство леммы (индукцией по высоте):

Основание индукции: *h(v)* = 0h(v)=0.

Если поддерево имеет нулевую высоту, то *v* v должен быть *null*, поэтому *bh(“v”)* = 0bh(″v″)=0.

Итак:

Индукционный шаг: пусть узел *v* v такой, что *h(v)* = kh(v)=k и поддерево имеет не менее  2bh(v)−1 внутренних узлов.

Покажем, что тогда *v’* v′, для которого *h(v’)* = kh(v)=k + 1 h(v′)=k+1, имеет не менее  2bh(v)−1 2bh(v′)−1 внутренних узлов.

Так как *v’*v′ имеет *h(v’)* > 0h(v)=k  h(v′)>0, это внутренний узел. Как таковой он имеет два потомка, оба из которых имеют чёрную высоту *bh(v’)* bh(v′), либо *bh(v’) - 1*bh(v′)−1 (зависит от того, является *v’*v′ красным, или чёрным).

По индукционному предположению каждый потомок имеет не менее 2bh(v′)−1−1 внутренних узлов, поэтому *v’*v′ имеет не менее

внутренних узлов.

Используя эту лемму, мы можем показать, что дерево имеет логарифмическую высоту. Так как по крайней мере половина узлов в любом пути от корня до листа — чёрные (свойство 4 красно-чёрного дерева), чёрная высота корня не менее *h(root)/2h(root)/2*. По лемме имеем:

Поэтому высота корня O(log n)O(log(n))O(log(n)).

*\*Доказательство из статьи Википедии*

*https://ru.wikipedia.org/wiki/Красно-чёрное\_дерево#Доказательство\_асимптотических\_границ*

2bh(v′)−1−1+2bh(v′)−1−1+1=2bh(v′)−1

**Графики (применялось 50 наборов от 10 до 9900 элементов, данные сгенерированы случайно):**

Неровности в графиках вызваны случайной генерацией данных (длина строки от 1 до 1000) и нестабильной работой процессора из-за нерегулирующихся сторонних процессов на моем компьютере.

*Зачем использовать красно-черные деревья?*

1. Во-первых, красно-черное дерево не удовлетворяет условию баланса AVL-дерева, то есть двоичного дерева поиска с разницей в высоте между левым поддеревом и правым поддеревом каждого узла не более 1. Однако предлагается добавить цвета к узлам. Красно-черное дерево использует нестрогий баланс в обмен на уменьшение количества поворотов при добавлении и удалении узлов. Любой дисбаланс будет устранен в течение трех вращений, в то время как AVL является строго сбалансированным деревом, поэтому он добавляется или удаляется Во время узла, в зависимости от различных ситуаций, количество поворотов больше, чем у красно-черного дерева. Таким образом, эффективность вставки красно-черного дерева выше

2. Красно-черное дерево может выполнять операции поиска, вставки и удаления с временной сложностью O (log2 (n)).

3. Проще говоря, красно-черное дерево предназначено для устранения дефектов бинарного дерева поиска, потому что в некоторых случаях бинарное дерево поиска вырождается в линейную структуру.

*Сравнение и выбор красно-черного дерева и сбалансированного дерева*1. Структура дерева баланса более интуитивно понятна, а скорость чтения выше, чем у красно-черного дерева; добавление и удаление узлов не так хорошо, как красно-черное дерево.  
2. Красно-черное дерево, скорость чтения хуже, чем у сбалансированного дерева; добавление и удаление узлов восстанавливают производительность баланса лучше, чем у сбалансированного дерева

*Использование красно-черных деревьев:*TreeMap, TreeSet и нижний слой HashMap после JDK1.8 используют красные и черные деревья.