Дерево бинарного поиска используется для ускорения поиска данных. Но обычное построение не гарантирует минимально возможно высоты дерева, в худшем случае высота дерева может быть $N-1$, что совпадает со временем поиска в последовательных структурах.

Поэтому рассмотрим один из вариантов сбалансированного дерева, имеющего фиксированную высоту.

1 Определения

Красно-черное дерево — дерево бинарного поиска, то есть, слева от корня находятся элементы, меньшие корня, справа — элементы, большие корня и обладающее следующими дополнительными свойствами:

1. Все узлы окрашены двумя цветами — либо красным, либо черным.
2. Корень — черный.
3. Все листья — черные.
4. Оба потомка красного узла — черные.
5. Длина простого пути от данного узла до любого листа содержит одинаковое количество черных узлов. Будет называть эту высоту — *черной высотой* и обозначать h_b.

Для упрощения алгоритма под листьями понимают пустые узлы (то есть, все узлы дерева являются внутренними). Таким образом, все пустые листья являются черными, а узлы могут иметь любой цвет. В данном случае используются *фиктивные листья*.

Иногда снимается требование черного корня, так как корень является потомком всех узлов и перекраска с красного на черный цвет просто увеличивает черную высоту на единицу, что сохраняет свойство 5 красно-черного дерева.

На рисунке 1 приведен пример красно-черного дерева. Буквами `N` обозначены фиктивные листья.

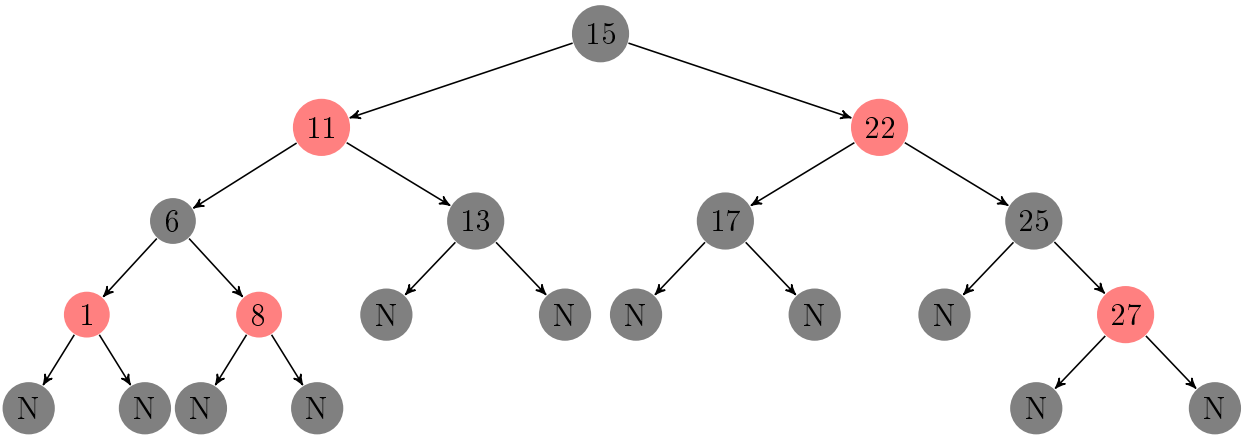


Рис. 1: Пример красно-черного дерева

Сначала убедимся, что дерево с такими свойствами действительно имеет фиксированную высоту.
При высоте h_b количество внутренних узлов не менее $2^{h_b-1}-1$.
Докажем по индукции, используя обычную высоту узла x — $h(x)$:

1. При $h(x)=1$ — получаем, что x — это узел, имеющий в качестве детей только фиктивные листья. Следовательно, черная высота $h_b=1$ и количество внутренних узлов в поддереве x равно $2^{h_b-1}-1=2^0-1=0$.

2. Так как любая внутренняя вершина содержит двух потомков (как узлы, так и фиктивные листы), их высоты на единицу меньше высоты узла x. Черные высоты этих вершин могут быть h_b или $h_b - 1$, если потомок красный или черный, соответственно. Тогда в каждом поддереве $2^{(h_b-1)-1} - 1$ вершин. Всего в поддереве $2 * (2^{h_b-2} - 1) + 1 = 2^{h_b-1} - 1$ вершин($+1$ — это сама вершина x).

   Теперь представим, что x — это корень, тогда в дереве содержится не менее $2^{h_b-1} - 1$ внутренних узлов.

Красно-черное дерево с N узлами имеет высоту $h = O(\log N)$.

Докажем это. Так как по свойству 3 у красного узла всегда черные дети, то количество красных узлов $\frac{h}{2}$, а черных — $\frac{h}{2}+1$. По доказанной выше лемме $N \geqslant 2^{\frac{h}{2}} - 1$. Следовательно $h \leqslant 2\log(N+1) = O(\log N)$.

Сложность: для вставки, удаления, поиска элемента — $O(\log N)$, расход памяти — $O(N)$.

2 Повороты

2.1 Левый поворот

На рисунке 2 представлен левый поворот узла X. После поворота правый ребенок узла X становится его родителем, а сам узел X становится левым ребенком узла Y.

Рис. 2: *а)* — до поворота, *б)* — после поворота

Рассмотрим алгоритм левого поворота:

Исходные параметры: узел `X`, ссылка на указатель `tr` (корень дерева)
Результат: измененное дерево
начало алгоритма
- `Y` — это правый ребенок `X`;
- Правым ребенком для `X` становится левый ребенок `Y`;
- **если** *левый ребенок `Y` не является листом* **тогда**
  - родителем для левого ребенка `Y` становится `X`
- Родителем для `Y` становится родитель `X`;
- **если** *`X` является левым ребенком для родителя `X`* **тогда**
  - `Y` становится левым ребенком для родителя `X`
- **иначе**
  - `Y` становится правым ребенком для родителя `X`;
- `X` становится левым ребенком `Y`;
- Родителем для `X` становится `Y`;
- **если** *у `Y` нет родителя* **тогда**
  - `Y` становится корнем и окрашивается в черный цвет;

Алгоритм 1: Левый поворот

2.2 Правый поворот

На рисунке 3 представлен правый поворот узла X. После поворота левый ребенок узла X становится его родителем, а сам узел X становится правым ребенком узла Y.

Рис. 3: *а)* — до поворота, *б)* — после поворота

Рассмотрим алгоритм правого поворота:

Исходные параметры: узел `X`, ссылка на указатель `tr` (корень дерева)
Результат: измененное дерево
начало алгоритма
- `Y` — это левый ребенок `X`;
- Левым ребенком для `X` становится правый ребенок `Y`;
- **если** *правый ребенок `Y` не является листом* **тогда**
  - родителем для правого ребенка `Y` становится `X`
- Родителем для `Y` становится родитель `X`;
- **если** *`X` является левым ребенком для родителя `X`* **тогда**
  - `Y` становится левым ребенком для родителя `X`
- **иначе**
  - `Y` становится правым ребенком для родителя `X`;
- `X` становится правым ребенком `Y`;
- Родителем для `X` становится `Y`;
- **если** *у `Y` нет родителя* **тогда**
  - `Y` становится корнем и окрашивается в черный цвет;

Алгоритм 2: Правый поворот

2.3 Создание узла

Немного исправим функцию `tree *node(tree *p, int X)` для первоначального создания узла.

Будем создавать узел, имеющий родителя, то есть, левый и правый ребенок узла — `NULL`, цвет узла — красный, значение узла — `X`, родитель для узла — `p`.

Теперь желательно создать отдельную функцию, создающую корень: `tree *root(int X)`, где правый, левый ребенок и родитель узла — `NULL`, цвет узла — черный, значение узла — `X`.

Построение красно-черного дерева начинается также как и для обычного дерева: определяется место для нового элемента (слева или справа от нужного узла), закрашивается красным цветом. А вот потом возникает проверка на удовлетворение свойствам красно-черного дерева и повороты, если вставленный элемент этим свойствам не удовлетворяет.

Есть два поворота, правый и левый. Вообще, вставка и удаление в красно-черное дерево сложны, но содержат не более трех поворотов.

2.4 Дополнительные функции

Для упрощения алгоритма введем несколько дополнительных функций, причем назовем их по аналогии с семейным деревом дед, дядя и брат.

Итак, дед — это родитель родителя текущего узла. Деда не будет у фиктивного листа (у `NULL` элементов не бывает никаких связей) и у корня (так как у него только родитель, равный `NULL`).

Исходные параметры: узел `X`
Результат: указатель на деда текущего узла
начало алгоритма
- **если** *существует узел* `X` *и существует родитель узла* `X` **тогда**
  - возвращаем родителя родителя текущего узла `X`
- **иначе**
  - возвращаем `NULL`

Алгоритм 3: Grandparent

Дядя — брат родителя. Сначала находим деда и другой его ребенок (не родитель текущего узла) будет дядей текущего узла. Если деда нет, значит, мы рассматриваем либо лист, либо корень. А у корня нет родителя, следовательно, нет и дяди.

Исходные параметры: узел `X`
Результат: указатель на дядю текущего узла
начало алгоритма
- `g` — дед узла `X`;
- **если** *не существует узел* `g` **тогда**
  - возвращаем `NULL`
- **если** *родитель* `X` *является левым ребенком узла* `g` **тогда**
  - возвращаем правого ребенка узла `g`
- **иначе**
  - возвращаем левого ребенка узла `g`

Алгоритм 4: Uncle

Брат — это другой ребенок родителя текущего узла. Брата нет у листа и корня.

Исходные параметры: узел `X`
Результат: указатель на брата текущего узла
начало алгоритма
- **если** *существует узел* `X` *и существует родитель узла* `X` **тогда**
  - **если** *узел* `X` *является левым ребенком своего родителя* **тогда**
    - возвращаем правого ребенка родителя узла `X`
  - **иначе**
    - возвращаем левого ребенка родителя узла `X`
- **иначе**
  - возвращаем `NULL`

Алгоритм 5: Sibling

В дальнейшем, во всех описаниях текущий узел будет обозначаться `N`, его родитель — `P`, его дядя — `U`, брат — `S`, племянники — `Sl` и `Sr`.

3 Вставка нового элемента в дерево

Всего есть пять случаев расположения узлов в дереве. Поэтому в программе необходимо сразу создать пять прототипов функций вставки, которые потом описать отдельно (так как они будут вызывать друг друга, без прототипов не обойтись).

Прототипы функций имеют вид:

```
void insert_case1(tree *&tr, tree *X);
```

где `tr` — это указатель на корень дерева, `X` — текущий узел.

Сначала рассмотрим все случаи, а потом опишем функцию вставки нового элемента. Напоминаю, что вставляемый узел всегда окрашен в красный цвет.

3.1 insert_case1

Первый случай самый простой — если `N` является корнем дерева. Тогда его необходимо перекрасить в черный цвет, чтобы сохранилось свойство 2. Так как корень добавляет черный цвет во все пути, то черная высота просто увеличится на единицу и свойство 5 не изменится.

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- **если** *не существует родителя узла* `X` **тогда**
  - перекрашиваем узел `X` в черный цвет.
- **иначе**
  - вызываем функцию `insert_case2 (tr,X)`

Алгоритм 6: insert_case1

Если узел не является корнем, переходим ко второму случаю.

3.2 insert_case2

Если родитель текущего узла имеет черный цвет, все нормально и ничего делать не надо, так добавление красного узла не меняет черную высоту и свойство 5 красно-черного дерева не меняется. Если родитель текущего узла — красный, то нарушается свойство 4 (у красного узла дети — черные), переходим к третьему случаю.

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- **если** *узел* `X` *красный* **тогда**
  - вызываем функцию `insert_case3 (tr,X)`
- **иначе**
  - выходим из функции (`return;`)

Алгоритм 7: insert_case2

3.3 insert_case3

Рассмотрим случай, когда родитель и дядя текущего узла красные. Тогда их дед будет черным, иначе было бы нарушено свойство 4 (у красного узла дети черные). На рисунке 4а представлен такой случай. Цифрами обозначены дети соответствующих узлов. Они черные, так как `P, U, N` красные, следовательно, их дети — черные.

Перекрашиваем дядю и родителя в черный цвет. Чтобы свойство 5 (одинаковая черная высота для всех веток) не нарушилось, надо деда перекрасить в красный цвет. Тогда черная высота для всех веток,

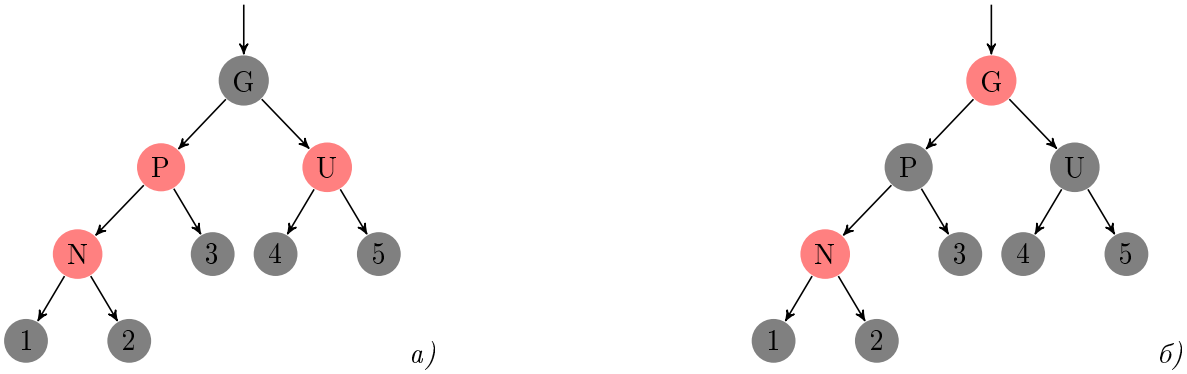


Рис. 4: *а)* — до применения `insert_case3`, *б)* — после применения `insert_case3`

выходящих из деда, не изменится. Но красный дед может нарушить свойство 2 (корень всегда черный) или свойство 4 (у красного родителя дети всегда черные), если родитель деда будет красным. Поэтому рекурсивно вызываем функцию `insert_case1` для деда.

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- `U` — дядя узла `X`
- `G` — дед узла `X`
- **если** *дядя существует И дядя красный И родитель красный* **тогда**
  - Родитель становится черным
  - Дядя становится черным
  - Дед становится красным
  - Вызываем функцию `insert_case1(tr, G)` для деда
- **иначе**
  - Вызываем функцию `insert_case4(tr, X)`

Алгоритм 8: insert_case3

Если родитель красный, а дядя — черный, переходим к следующему случаю.

3.4 insert_case4

Данный случай вспомогательный для пятого случая. Обратите внимание, что в остальных случаях следующая функция вызывается в ветке `else`, то функция `insert_case5(tr, X)` вызывается в любом случае.

Пусть родитель — красный, а дядя — черный. Следовательно, дед — черный (красные дети могут быть только у черного родителя по свойству 4). Брат и дети текущего узла `N` тоже черные по тому же свойству 4. Цвет детей дяди не имеет в данном случае значения. При этом существует 2 варианта (ветки `then` и `else`) в алгоритме:

1. `N` (текущий узел) является правым ребенком для `P` (своего родителя), а `P` — левым ребенком для `G` (деда текущего узла). Тогда делаем левый поворот, в итоге `P` становится левым ребенком текущего узла `N`, который становится левым ребенком своего деда (см. рисунки 5а–б).

2. `N` (текущий узел) является левым ребенком для `P` (своего родителя), а `P` — правым ребенком для `G` (деда текущего узла). Тогда делаем правый поворот, в итоге `P` становится правым ребенком текущего узла `N`, который становится правым ребенком своего деда (см. рисунки 5в–г).

После поворота свойство 4 (у красного родителя черные дети) остаются нарушенным,поэтому переходим к случаю 5 относительно бывшего родителя (`insert_case5(tr, P)`).

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- `G` — дед узла `X`
- **если** *узел `X` — правый ребенок своего родителя И родитель узла `X` — левый ребенок своего деда* **тогда**
  - Выполняем левый поворот относительно узла `X`.
  - Переобозначаем `X = X->left`
- **иначе**
  - **если** *узел `X`— левый ребенок своего родителя И родитель узла `X` — правый ребенок своего деда* **тогда**
    - Выполняем правый поворот относительно узла `X`.
    - Переобозначаем `X = X->right`
- Вызываем функцию `insert_case5(tr, X)`

Алгоритм 9: insert_case4

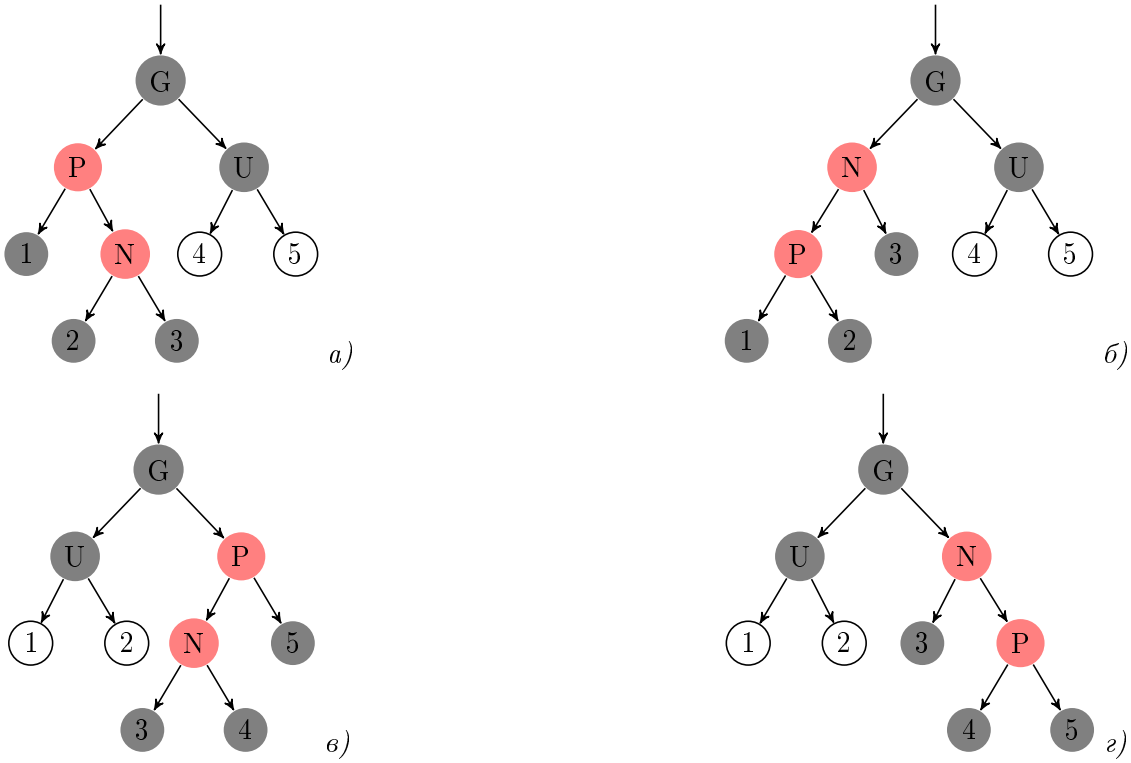


Рис. 5: *а)* — до применения `insert_case4` случай 1, *б)* — после применения `insert_case4` случай 1, *в)* — до применения `insert_case4` случай 2, *г)* — после применения `insert_case4` случай 2

3.5 insert_case5

В данном случае родитель — красный, дядя — черный. Дед, брат и дети текущего узла — черные (по свойству 4), цвет детей дяди не имеет значения. При этом текущий узел — левый ребенок своего родителя, а родитель — левый ребенок деда (рисунок 6а), или, наоборот, текущий узел — правый ребенок своего родителя, а родитель — правый ребенок деда (рисунок 6в).

Выполняем правый (или левый) поворот относительно деда. После этого родитель текущего узла становится родителем также бывшего деда. После этого меняем цвета деда и родителя (рисунки 6б, г). В итоге свойство 5 о равенстве черной высоты не меняется, так как теперь все пути идут через черного родителя. Свойство 4 теперь выполняется.

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- `G` — дед узла `X`
- Цвет родителя узла `X` — теперь черный
- Цвет деда — красный
- **если** *узел `X` — левый ребенок своего родителя И родитель узла `X` — левый ребенок своего деда*
  - **тогда**
    - Выполняем правый поворот относительно узла `G`.
  - **иначе**
    - Выполняем левый поворот относительно узла `G`.

Алгоритм 10: insert_case5

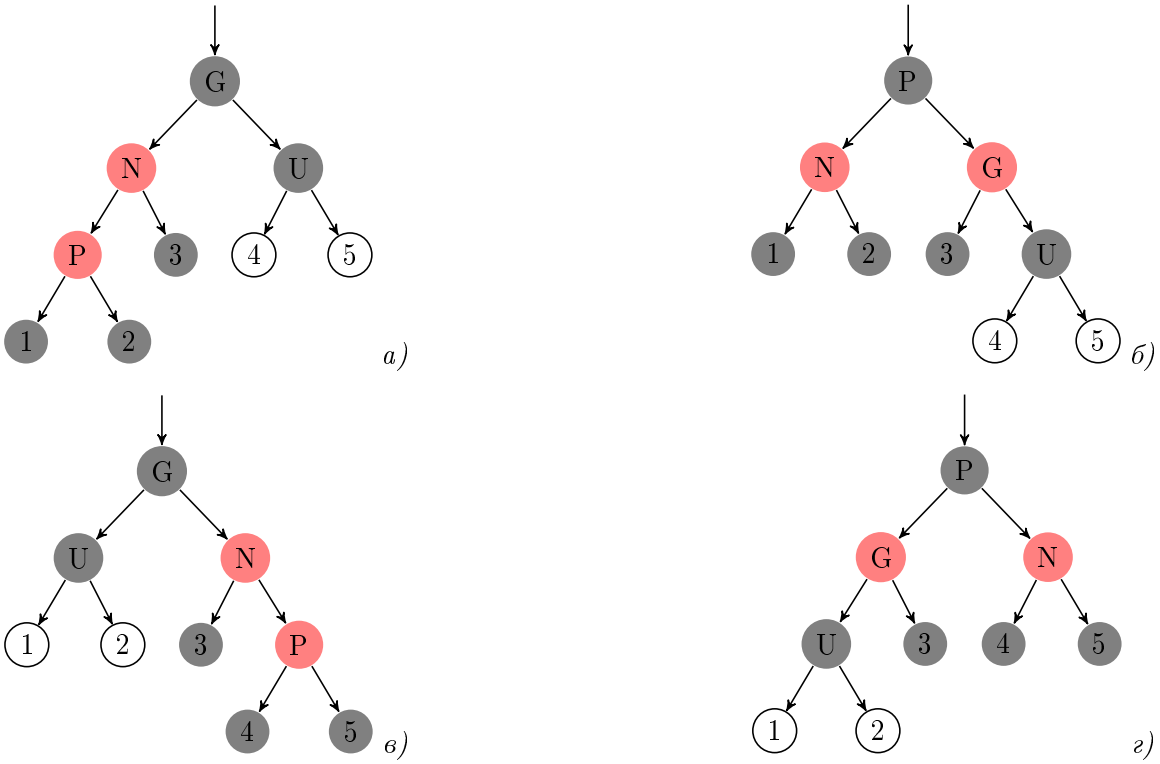


Рис. 6: *а)* — до применения `insert_case5` случай 1, *б)* — после применения `insert_case5` случай 1, *в)* — до применения `insert_case5` случай 2, *г)* — после применения `insert_case5` случай 2

3.6 Функция вставки нового узла в дерево

Вставка элемента похожа на создание обычного дерева бинарного поиска. Ищем свободное место для вставки нового элемента, после этого начинаем балансировку, вызывая функцию `insert_case1`.

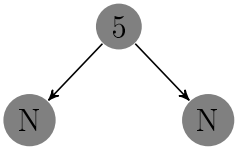
Исходные параметры: указатель на корень дерева (`*&tr`), указатель на предыдущий узел (`*Prev`), число `X`
начало алгоритма
- **если** *`X` меньше значения узла `Prev` И у узла `Prev` нет левого ребенка* **тогда**
  - `Prev->left = node(Prev, X)`
  - Вызываем функцию `insert_case1` относительно `Prev->left`
- **иначе**
  - **если** *`X` больше значения узла `Prev` И у узла `Prev` нет правого ребенка* **тогда**
    - `Prev->right = node(Prev, X)`
    - Вызываем функцию `insert_case1` относительно `Prev->right`
  - **иначе**
    - **если** *`X` меньше значения узла `Prev` И у узла `Prev` есть левый ребенок* **тогда**
      - Вызываем эту функцию `insert` относительно `Prev->left`
    - **иначе**
      - **если** *`X` больше значения узла `Prev` И у узла `Prev` есть правый ребенок* **тогда**
        - Вызываем эту функцию `insert` относительно `Prev->right`

Алгоритм 11: insert

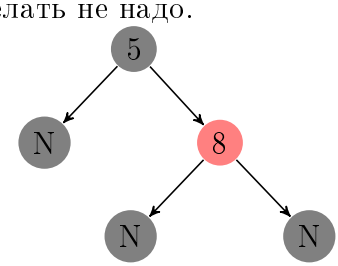
Рассмотрим пример. Для первого элемента воспользуемся функцией создания корня `tr = root(X)`. Как корень, этот элемент будет черным. Для остальных элементов вызывается функция `insert(tr, tr, X)`.

Набор чисел: $5, 8, 11, 7, 6$.

Узел 5 — это корень, он черный и его дети (листья) тоже черные.

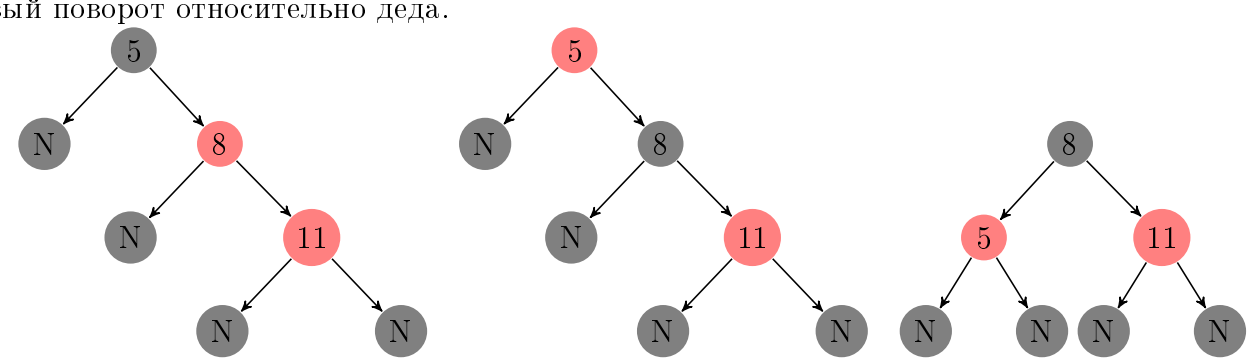


Узел 8 — красный, его родитель — узел 5. Родитель черный, следовательно, это случай 2, ничего делать не надо.



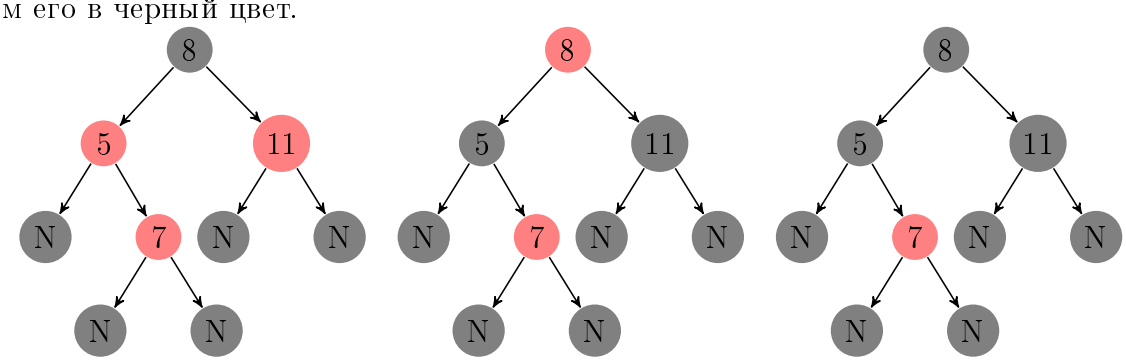
Узел 11 — красный, правый ребенок узла 8. Родитель красный, дядя — черный. Родитель правый ребенок для деда, текущий узел — правый ребенок для родителя. Это случай 5.

Сначала перекрашиваем деда (узел 5) в красный цвет, родителя (узел 8) в черный, а потом совершаем левый поворот относительно деда.



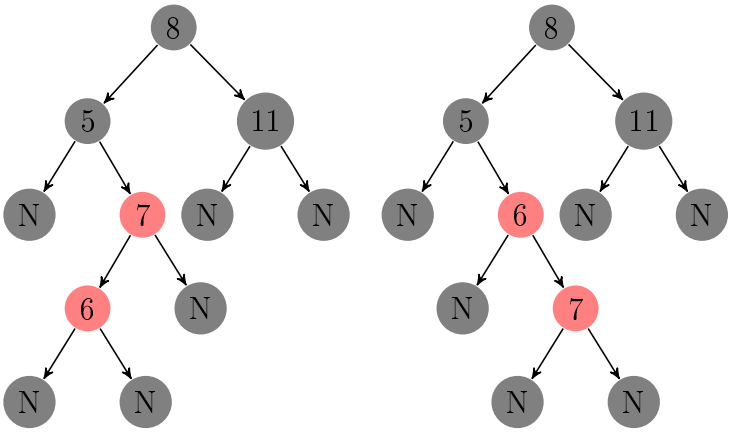
Узел 7 — красный, является правым ребенком узла 5. Родитель и дядя — красные. Это случай 3.

Сначала перекрашиваем родителя (узел 5) и дядю (узел 11) в черный цвет, деда (узел 8) — в красный. Вызываем функцию `insert_case1` относительно деда (узел 8). Так как это корень, просто перекрашиваем его в черный цвет.



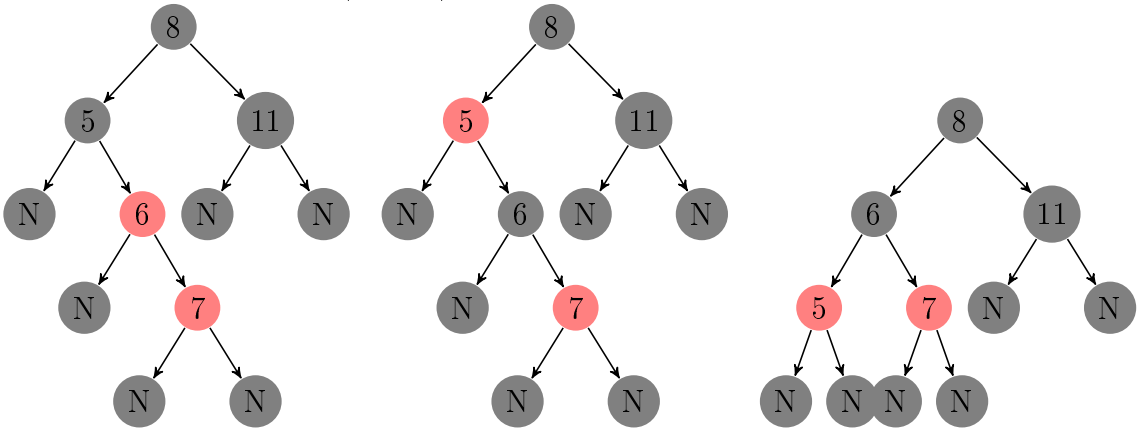
Узел 6 — красный, является левым ребенком узла 7. Родитель (узел 7) является правым ребенком деда (узел 5). Родитель красный, дядя — черный. Это случай 4.

Выполняем правый поворот относительно родителя (узел 7). И вызываем относительно узла 7 функцию `insert_case5`.



Теперь рассматриваем случай 5 относительно узла 7. Он красный, его родитель (узел 6) красный, дядя (лист) — черный. Узел 7 — правый ребенок для родителя (узел 6), родитель (узел 6) — правый ребенок для деда (узел 5).

Перекрашиваем родителя (узел 6) в черный цвет, деда (узел 5) — в красный. Потом делаем левый поворот относительно деда (узел 5).



4 Удаление элемента из красно-черного дерева

В случае удаления узла также существует несколько случаев перебалансировки. Эти функции тоже необходимо описать как прототипы вида `delete_case1(tree*&tr, tree *n)`. Таких функций должно быть шесть.

Аналогично с алгоритмом вставки сначала рассмотрим варианты перебалансировки, а уже потом удаления элемента.

Удаление работает по той же схеме, что и в случае обычного дерева бинарного поиска:

1. Удаляемый узел имеет детьми только фиктивные листья. Важен цвет: если красный, то просто удаляем. Если черный — вызываем перебалансировку.

2. Удаляемый узел имеет одного реального ребенка. Меняем местами удаляемый узел и его ребенка. Если узел красный, то просто удаляемый. Если узел черный, а цвет его ребенка красный, перекрашиваем ребенка в черный. Если и узел и его ребенок черные, вызываем перебалансировку.

3. Удаляемый узел имеет двух реальных детей. Находим либо следующий, либо предыдущий элемент (по симметричному обходу). Это в любом случае будет элемент либо с одним ребенком, либо без реальных детей. Меняем эти значения этих узлов. И удалять теперь будем найденный узел — это уже рассмотренные выше случаи.

4.1 delete_case1

Первый случай — Удаляемый элемент является корнем. Этот вариант возможен только если дерево имеет только одну ветку. В таком случае ребенок становится новым корнем дерева.

Так как корень имеет черный цвет и входит во все длины путей, его удаление приведет к изменению черной высоты на единицу. Свойство 5 (одинаковая длина черного пути для всех узлов) не изменится. Перебалансировка вызывается только для случая черного узла и черного ребенка, то свойство 2 (корень всегда черный тоже не изменится).

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- **если** *не существует родителя узла* `X` **тогда**
  - **если** *узел* `X` *имеет левого ребенка* **тогда**
    - корнем становится левый ребенок узла `X`
  - **иначе**
    - корнем становится правый ребенок узла `X`
- **иначе**
  - вызываем функцию `delete_case2 (tr,X)`

Алгоритм 12: delete_case1

4.2 delete_case2

Удаляемый узел черный, родитель — черный, брат — красный. Так как брат красный, его дети — черные. Цвет остальных узлов значения не имеет.

Перекрашиваем родителя в красный цвет, брата — в черный. И делаем поворот относительно родителя (если узел `X` левый ребенок, используем левый поворот (см. рисунок 7), иначе — правый).

Удаляемый узел по-прежнему остался черным, поэтому вызываем следующую функцию.

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- `S = sibling(X)`
- **если** *цвет брата — красный* **тогда**
  - Цвет родителя узла `X` — теперь красный
  - Цвет брата (узел `S`) — черный
  - **если** *узел* `X` *— левый ребенок своего родителя* **тогда**
    - Выполняем левый поворот относительно родителя узла `X`.
  - **иначе**
    - Выполняем правый поворот относительно родителя узла `X`.
- Вызываем функцию `delete_case3(tr, X)`

Алгоритм 13: delete_case2

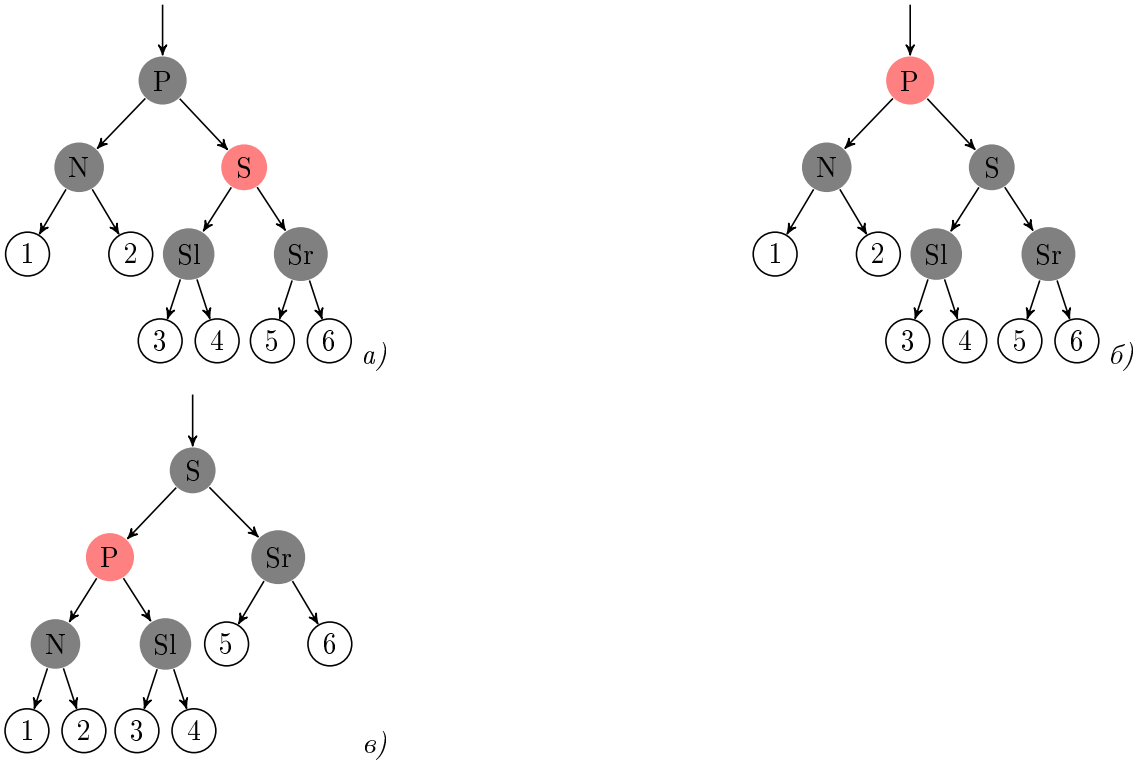


Рис. 7: *а)* — до применения `delete_case2`, *б)* — перекрашивание, *в)* — после поворота

4.3 delete_case3

Удаляемый узел, его родитель, брат и племянники — черные. Тогда перекрашиваем брата в красный цвет (рисунок 8). Если удалим черный узел, нарушится черная высота, поэтому вызываем заново перебалансировку (функцию `delete_case1(tr, X)`).

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
  `S = sibling(X)`
  если *родитель узла* `X` *— черный И брат узла* `X` *— черный И (левый племянник — лист ИЛИ черный) И (правый племянник — лист ИЛИ черный)* **тогда**
    Перекрашиваем брата (узел `S`) в красный цвет .
    Вызываем функцию `delete_case1(tr, X)`
  иначе
    Вызываем функцию `delete_case4(tr, X)`

Алгоритм 14: delete_case3

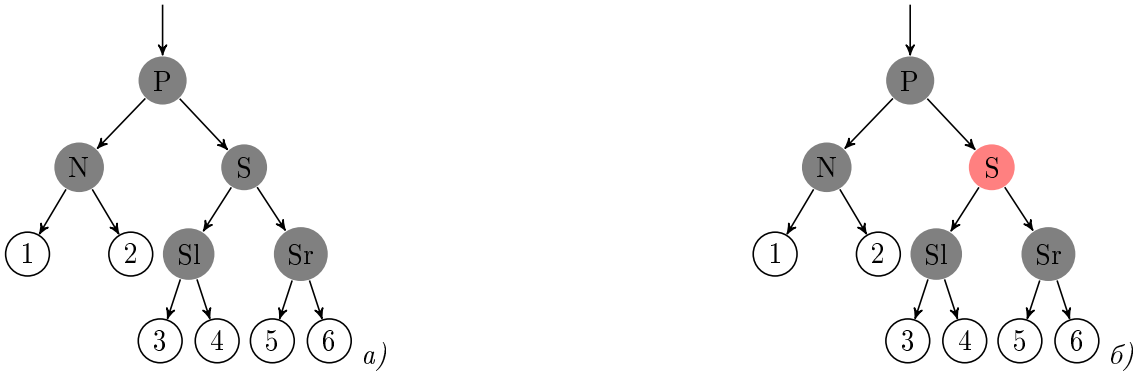


Рис. 8: *а)* — до применения `delete_case3`, *б)* — перекрашивание

4.4 delete_case4

Удаляемый узел, брат и племянники — черные, родитель — красный. В данном случае просто меняем цвета у родителя и брата. Родитель становится черным, брат — красный (рисунок 9). Балансировка сохраняется.

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- `S = sibling(X)`
- **если** *родитель узла* `X` *— красный И брат узла* `X` *— черный И (левый племянник — лист ИЛИ черный) И (правый племянник — лист ИЛИ черный)* **тогда**
  - Перекрашиваем брата (узел `S`) в красный цвет .
  - Перекрашиваем родителя узла `X` в черный цвет.
- **иначе**
  - Вызываем функцию `delete_case5(tr, X)`

Алгоритм 15: delete_case4

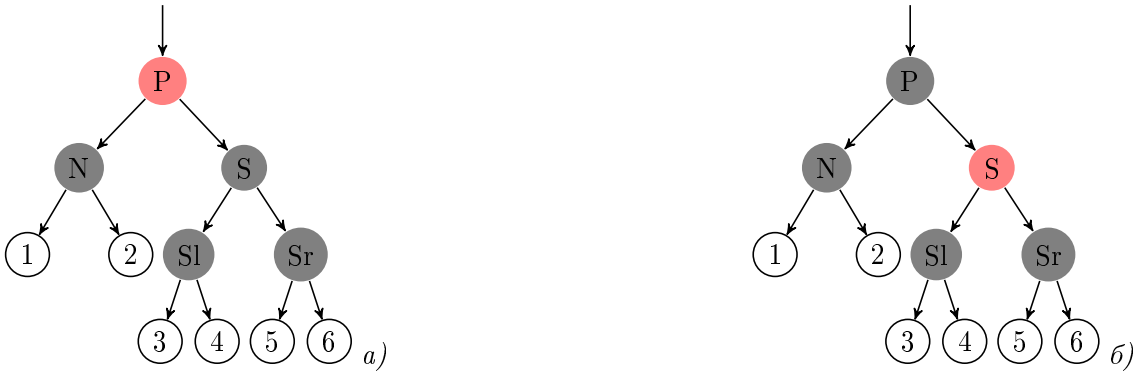


Рис. 9: *а)* — до применения `delete_case4`, *б)* — перекрашивание

4.5 delete_case5

В данном случае цвет удаляемого узла и его родителя не важен. Делаем перебалансировку между братом и его детьми. Брат черный, «прилежащий» племянник — красный, «противоположный» племянник — черный. Дети красного племянника — черные, цвет остальных узлов не важен.

Если удаляемый узел — левый ребенок своего родителя. Тогда его левый племянник будет красным. Тогда перекрашиваем брата в красный цвет, а левого племянника — в черный. После этого выполняем левый поворот относительно брата (рисунок 10).

Если удаляемый узел — правый ребенок своего родителя. Тогда его правый племянник будет красным. Тогда перекрашиваем брата в красный цвет, а правого племянника — в черный. После этого выполняем правый поворот относительно брата (рисунок 11).

Перебалансировка не завершена, вызываем функцию `delete_case6(tr, X)`.

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- `S = sibling(X)`
- **если** *цвет брата — черный* **тогда**
  - **если** *узел `X` является левым ребенком своего родителя И (левый племянник — не лист И красный) И (правый племянник — лист ИЛИ черный)* **тогда**
    - Перекрашиваем брата в красный цвет
    - Перекрашиваем левого племянника в черный цвет
    - Выполняем правый поворот относительно брата
  - **иначе**
    - **если** *узел `X` является правым ребенком своего родителя И (правый племянник — не лист И красный) И (левый племянник — лист ИЛИ черный)* **тогда**
      - Перекрашиваем брата в красный цвет
      - Перекрашиваем правого племянника в черный цвет
      - Выполняем левый поворот относительно брата
- Вызываем функцию `delete_case6(tr, X)`

Алгоритм 16: delete_case5

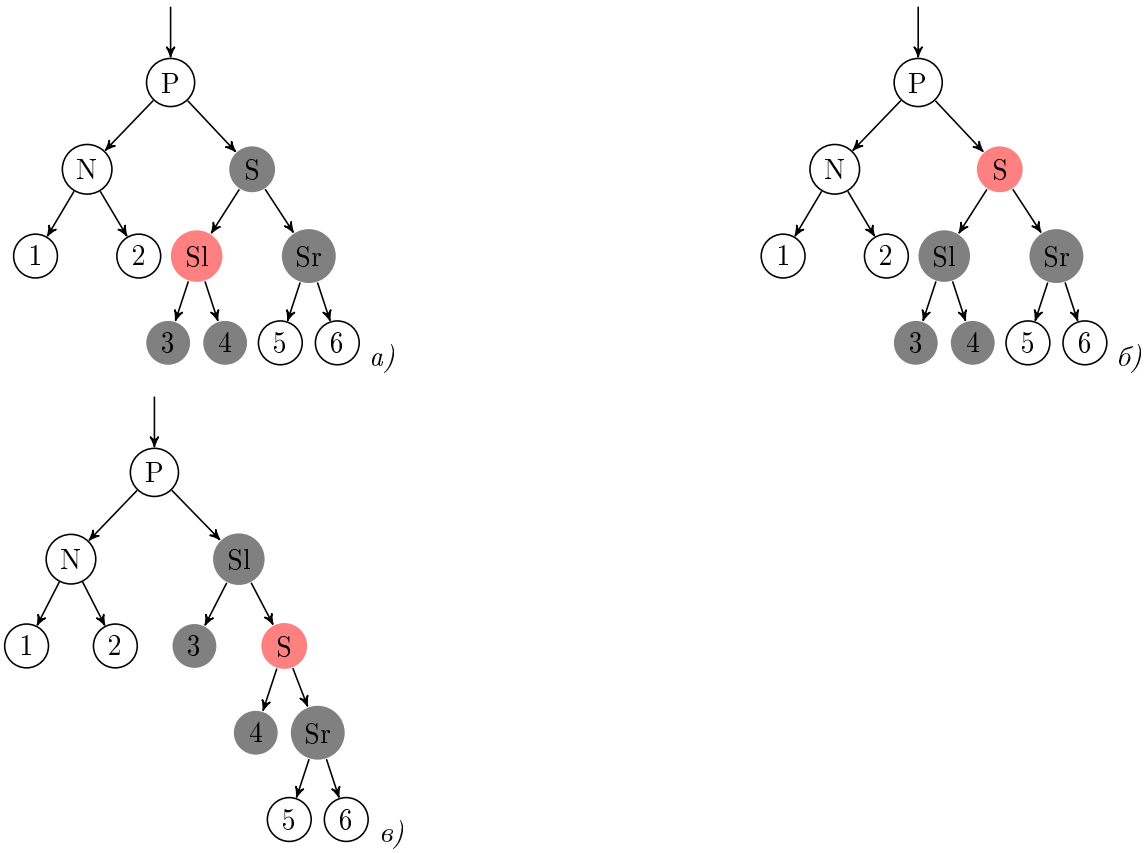


Рис. 10: *а)* — случай, когда удаляемый узел — левый ребенок, до применения `delete_case5`, *б)* — перекрашивание, *в)*— после поворота

4.6 delete_case6

В данном случае цвет удаляемого узла и его родителя не важен. Брат черный, «прилежащий» племянник — черный, «противоположный» племянник — красный. Дети красного племянника — черные, цвет остальных узлов не важен.

Сначала перекрашиваем брата в цвет родителя (если родитель был красным, значит, брат будет красным и наоборот). Родителя перекрашиваем в черный цвет.

Если удаляемый узел — левый ребенок своего родителя. Тогда его правый племянник будет красным. Тогда перекрашиваем правого племянника в черный цвет. После этого выполняем левый поворот относительно родителя (рисунок 12).

Если удаляемый узел — правый ребенок своего родителя. Тогда его левый племянник будет красным. Тогда перекрашиваем левого племянника в черный цвет. После этого выполняем правый поворот относительно родителя (рисунок 13).

После такой перебалансировки у удаляемого узла (`N`) появился один дополнительный черный предок (если родитель (`P`) был красным, он стал черным, если родитель (`P`) был черным, появился черный дед (`S`, а его цвет совпадает с цветом родителя).

Другие пути, не проходящие через `N`:

1. проходят через нового брата узла `N` (`Sl` на рисунке 12в или `Sr` на рисунке 13в). Но эти пути проходят также через узлы `S` и `P`, которые поменялись цветами. Следовательно, черная высота после перебалансировки не изменится.

2. проходят через нового дядю узла `N` (`Sr` на рисунке 12в или `Sl` на рисунке 13в). Раньше эти пути шли через черный узел `S`, его родителя любого цвета и красного ребенка (племянника удаляемого узла). После перебалансировки узел `S` теперь имеет цвет своего родителя, красный племянник стал черным, следовательно, черная высота не изменилась.

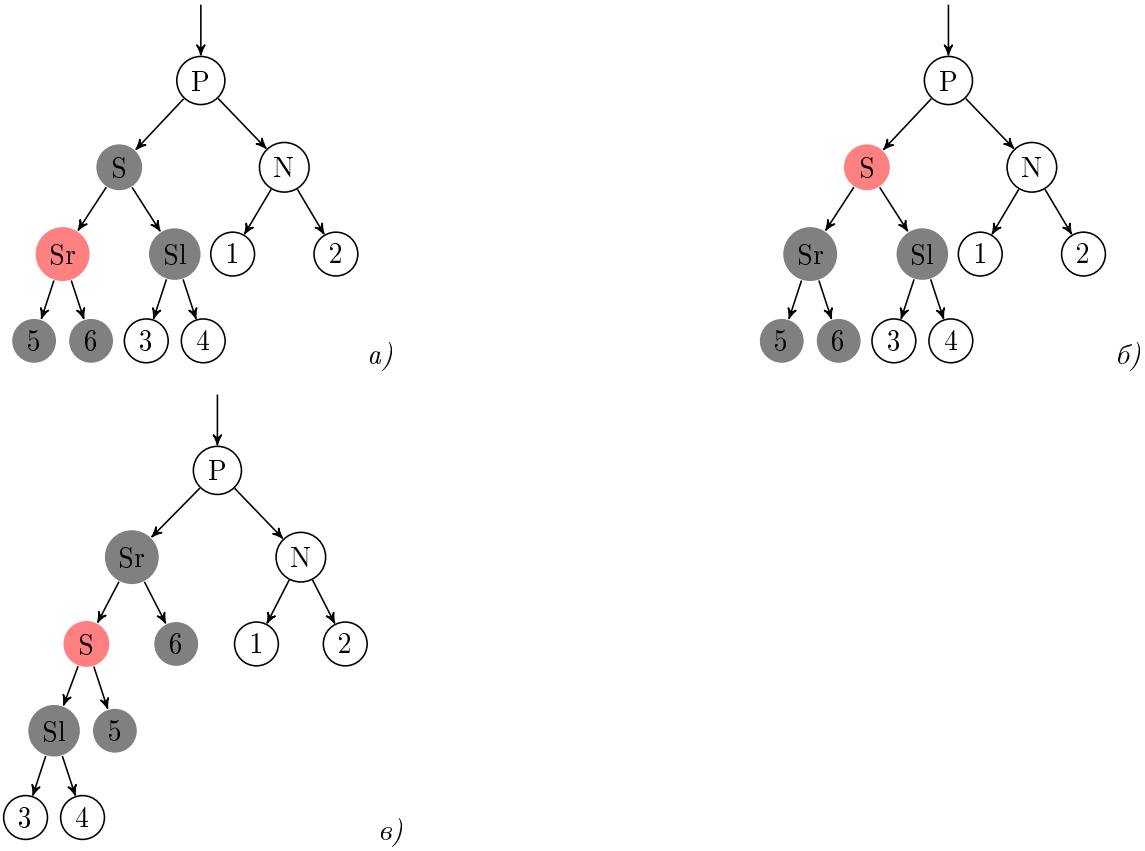


Рис. 11: *а)* — случай, когда удаляемый узел — правый ребенок до применения `delete_case5`, *б)* — перекрашивание, *в)*— после поворота

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
`S = sibling(X)`
Брата перекрашиваем в текущий цвет родителя.
Родителя перекрашиваем в черный цвет.
если *узел* `X` *является левым ребенком* **тогда**
Перекрашиваем правого племянника в черный цвет.
Выполняем левый поворот относительно родителя.
иначе
Перекрашиваем левого племянника в черный цвет.
Выполняем правый поворот относительно родителя.

Алгоритм 17: delete_case6

Таким образом, после перебалансировки пути, проходящие через черный узел `N`, имеют черную высоту на единицу большую, чем все остальные. Тогда после удаления узла `N` длины черных путей будут одинаковы и все свойства красно-черного дерева сохраняться.

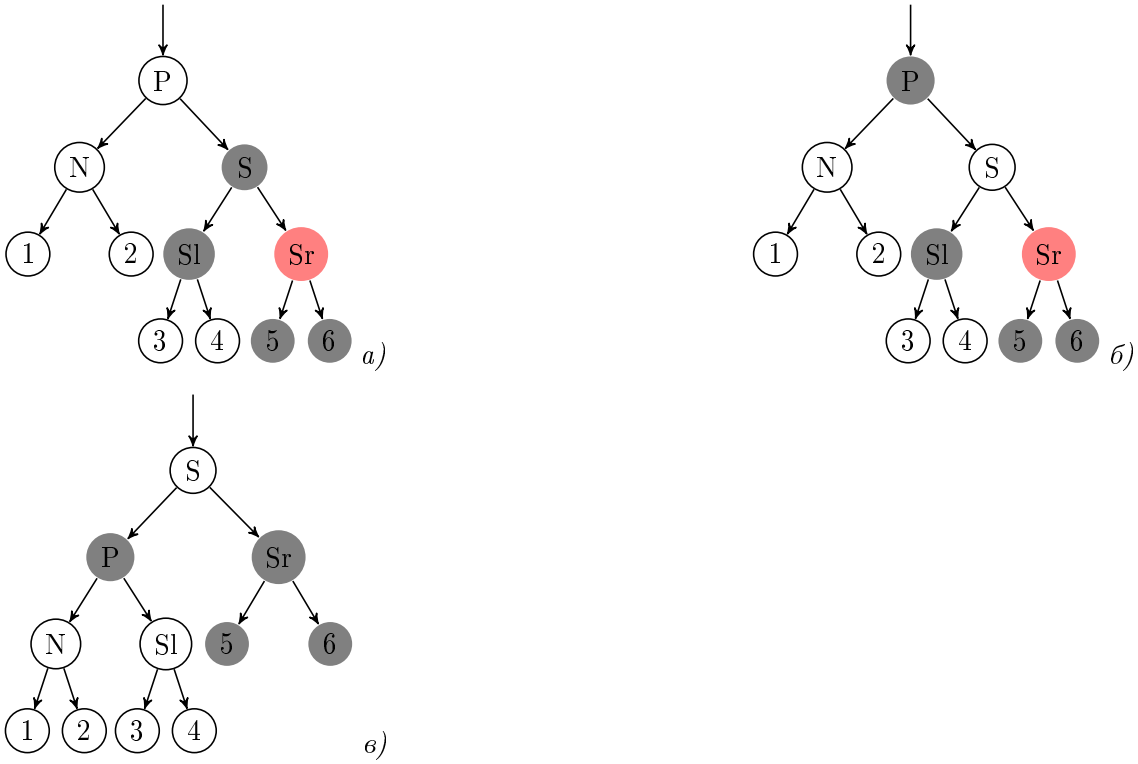


Рис. 12: *а)* — случай, когда удаляемый узел — левый ребенок, до применения `delete_case6`, *б)* — перекрашивание, *в)*— после поворота

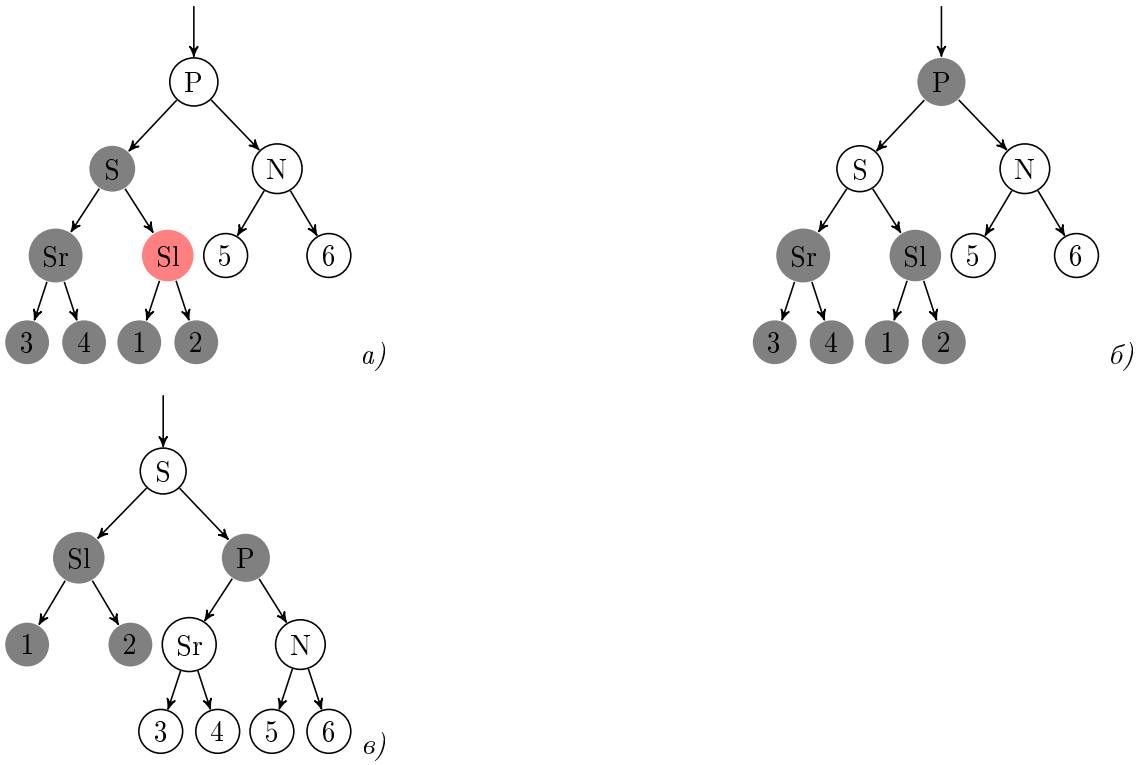


Рис. 13: *а)* — случай, когда удаляемый узел — правый ребенок до применения `delete_case6`, *б)* — перекрашивание, *в)*— после поворота

4.7 Замена

В случае, когда удаляемый узел имеет одного реального ребенка, необходимо сделать так, чтобы этот ребенок стал ребенком своего деда. Для этого воспользуемся функцией `void replace(tree *&tr, tree`

`*n).`

Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- **если** *существует левый ребенок* **тогда**
  - `ch = X->left`
  - Родителем узла `ch` становится родитель узла `X`.
  - **если** *родитель узла `X` существует* **тогда**
    - **если** *узел `X` является левым ребенком своего родителя* **тогда**
      - Левым ребенком родителя узла `X` становится узел `ch`
    - **иначе**
      - Правым ребенком родителя узла `X` становится узел `ch`
- **иначе**
  - `ch = X->right`
  - Родителем узла `ch` становится родитель узла `X`.
  - **если** *родитель узла `X` существует* **тогда**
    - **если** *узел `X` является левым ребенком своего родителя* **тогда**
      - Левым ребенком родителя узла `X` становится узел `ch`
    - **иначе**
      - Правым ребенком родителя узла `X` становится узел `ch`

Алгоритм 18: replace

4.8 Функция удаления узла

Операция удаления имеет следующий алгоритм:

1. Удаляемый узел имеет двух реальных детей. Ищем элемент, на который можно заменить значение удаляемого узла. Если значение удаляемого узла больше значения корня, выбираем самый правый узел (максимальное значение), если меньше — самый левый (минимальное значение). Меняем значения найденного элемента и удаляемого узла. Удаляем теперь найденный элемент. Теперь у удаляемого узла либо один реальный ребенок, либо ни одного.

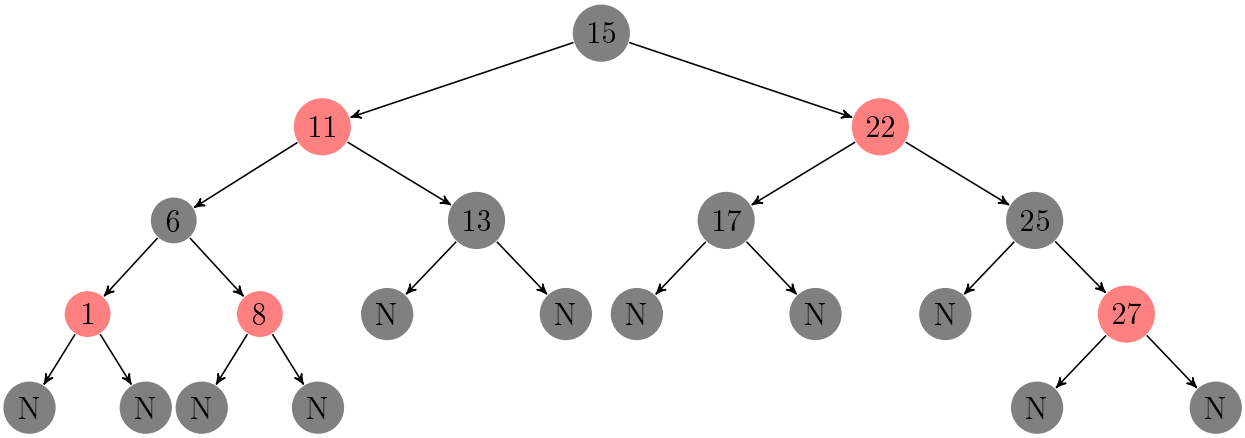
2. Удаляемый узел имеет одного реального ребенка. С помощью функции `replace(tr, N)` меняем местами удаляемый узел и его ребенка. Если удаляемый узел — красный, просто удаляем его. Если удаляемый узел — черный, а его ребенок — красный, то перекрашиваем ребенка и удаляем узел. Если и удаляемый узел и его ребенок — черные, вызываем перебалансировку, а только потом удаляем.

3. Удаляемый узел не имеет реальных детей (только фиктивные черные листья). Если удаляемый узел — красный, удаляем его. Если черный, вызываем перебалансировку.

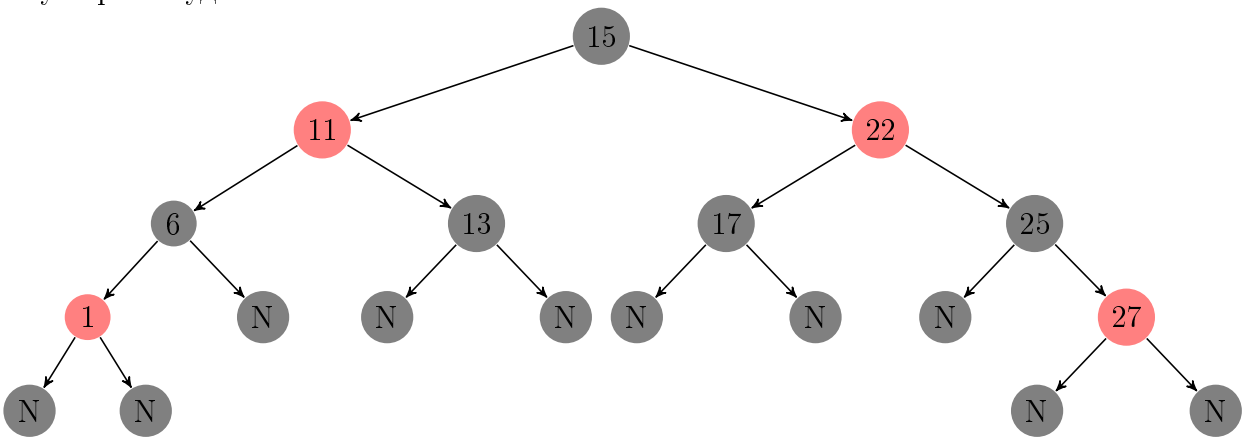
Исходные параметры: указатель на корень дерева, узел `X`
начало алгоритма
- **если** *существует правый И левый ребенок узла* `X` **тогда**
  - **если** *значение узла* `X` *меньше или равно значения корня* **тогда**
    - `buf` равен самому правому узлу левого поддерева узла `X`(не листу)
  - **иначе**
    - `buf` равен самому левому узлу правого поддерева узла `X` (не листу)
  - Меняем значения узла `X` и узла `buf`
  - `X` = `buf`
- **если** *существует левый ИЛИ правый ребенок узла* `X` **тогда**
  - **если** *есть левый ребенок И нет правого ребенка* **тогда**
    - `ch` равен левому ребенку
  - **если** *есть правый ребенок И нет левого ребенка* **тогда**
    - `ch` равен правому ребенку
  - Вызываем функцию `replace(tr,X)`
  - **если** *узел* `X` *черный* **тогда**
    - **если** *цвет ребенка красный* **тогда**
      - Перекрашиваем узел `ch` в черный цвет
    - **иначе**
      - Вызываем функцию `delete_case1(tr, X)`
- **иначе**
  - //нет детей
  - **если** *узел* `X` *черный* **тогда**
    - Вызываем функцию `delete_case1(tr, X)`
  - **иначе**
    - **если** *узел* `X` *является левым ребенком своего родителя* **тогда**
      - Обнуляем левого ребенка родителя узла `X`
    - **иначе**
      - Обнуляем правого ребенка родителя узла `X`
- Удаляем узел `X`

Алгоритм 19: delete_one

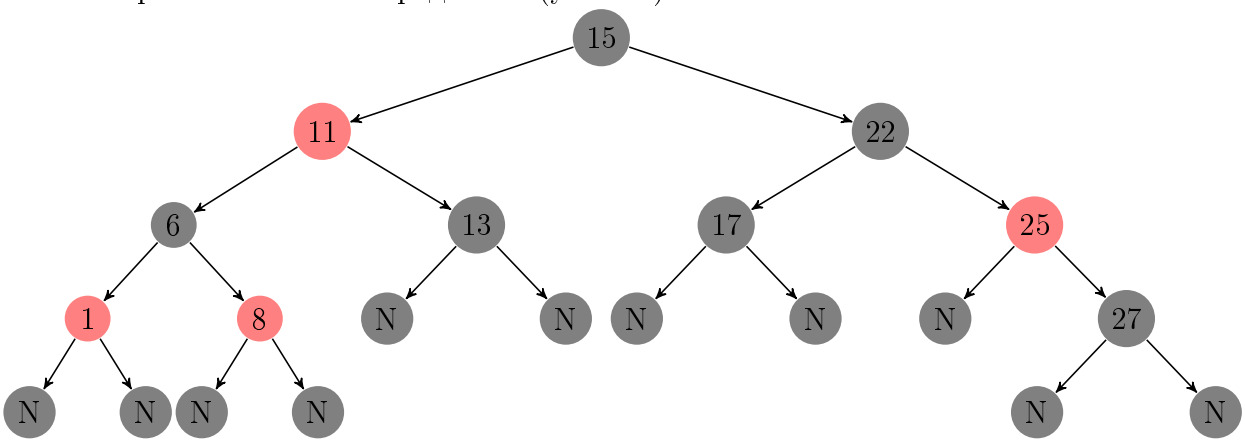
Рассмотрим примеры удаления узлов из красно-черного дерева.

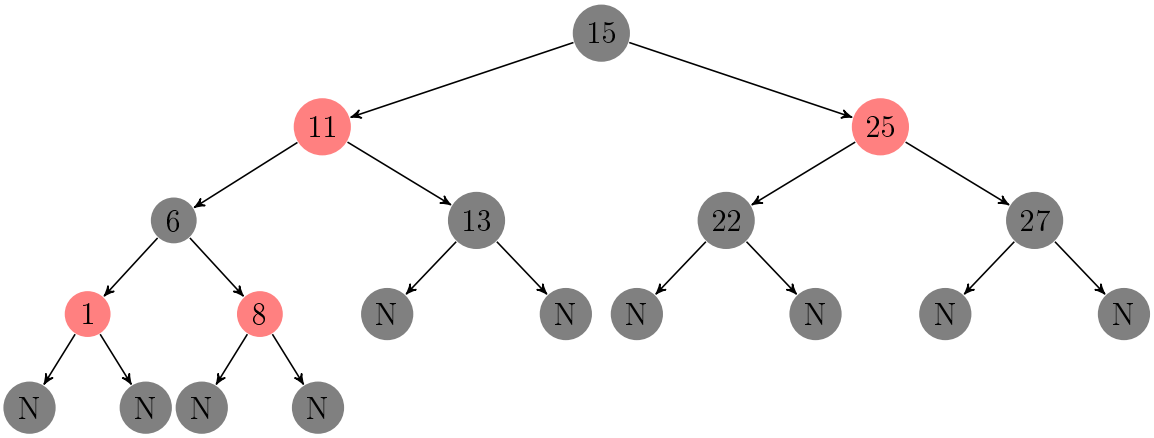


Удаляем узел 8. Он не имеет детей и красный, следовательно, его удаление не изменит черную высоту. Просто удаляем.

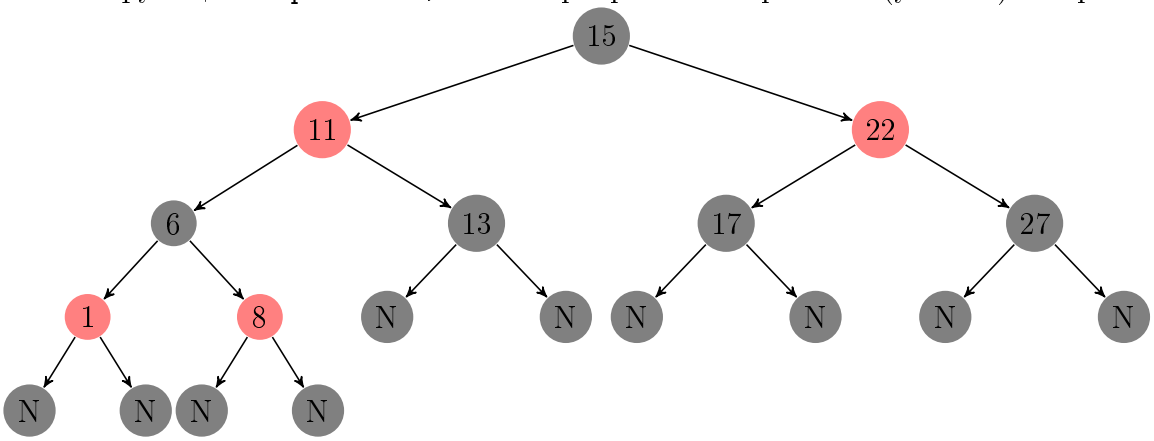


Удаляем узел 17. Он не имеет детей, но черный. Вызываем перебалансировку. Родитель (узел 22) — красный, узел 17 его левый ребенок. Брат (узел 25) — черный, правый племянник (узел 27)— красный. Вызывается функция `delete_case6(tr, 17)`. Сначала перекрашиваем. Брат (узел 25) становится красным (цвет родителя), родитель (узел 22) и племянник (узел 27) становятся черными. После этого делаем правый поворот относительно родителя (узел 22).



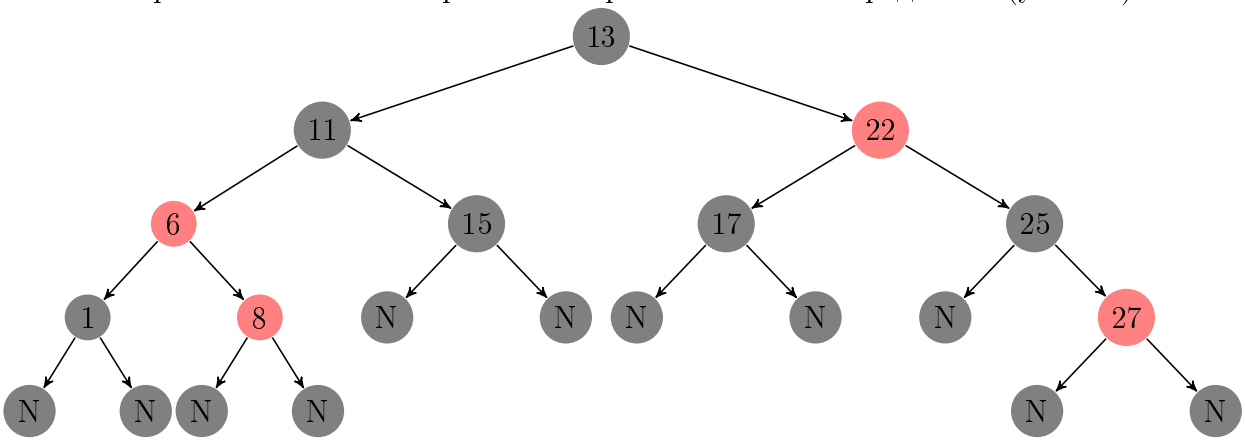


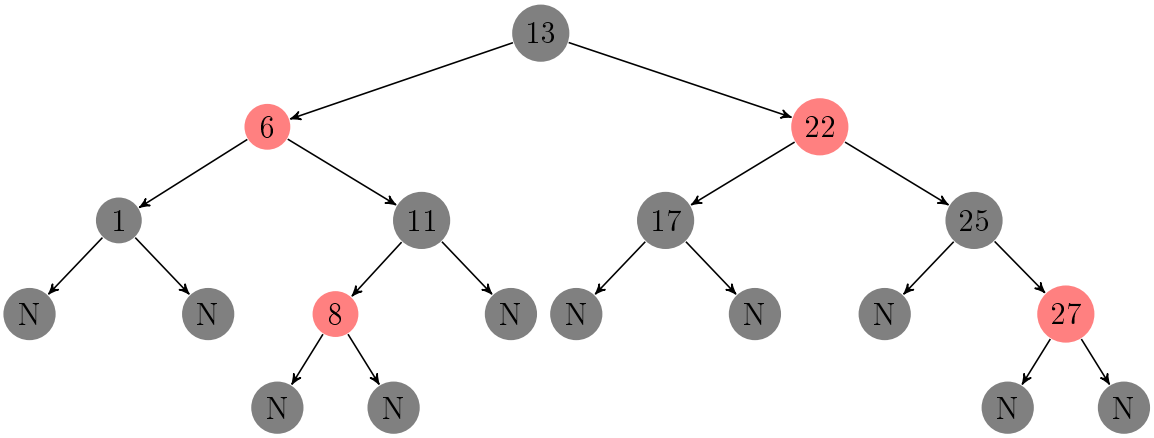
Удаляем узел 25. Имеет одного ребенка (узел 27). Ребенок красный, сам удаляемый узел — черный. Вызываем функцию `replace(tr, 25)`. Перекрашиваем ребенка (узел 27) в черный цвет.



Удаляем узел 15. Он имеет двоих детей, является корнем. Ищем максимум в левой ветке. Это узел 13. Меняем местами их значения. Теперь узел 15 не имеет детей, является черным. Вызываем перебалансировку.

Родитель узла 15 теперь красный узел 11. Узел 15 — правый ребенок. Брат (узел 6) — черный. Левый племянник (узел 1) — красный. Вызываем функцию `delete_case6(tr, 15)`. Сначала перекрашиваем. Брат (узел 6) становится красным (цвет родителя), родитель (узел 11) и левый племянник (узел 1) становятся черными. Вызываем правый поворот относительно родителя (узел 11).




13
6
22
1
11
17
25
N
N
8
N
N
N
N
N
27
N
N
N
N