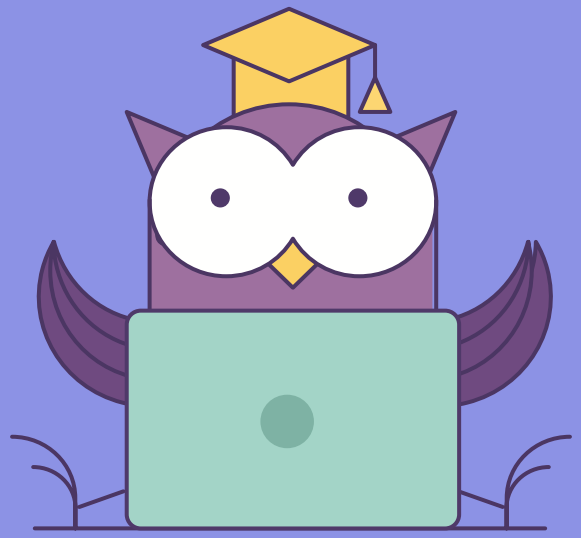
ОНЛАЙН-ОБРАЗОВАНИЕ

# Не забыть включить запись!

# Меня хорошо слышно && видно?



Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

# Поехали!

OTUS

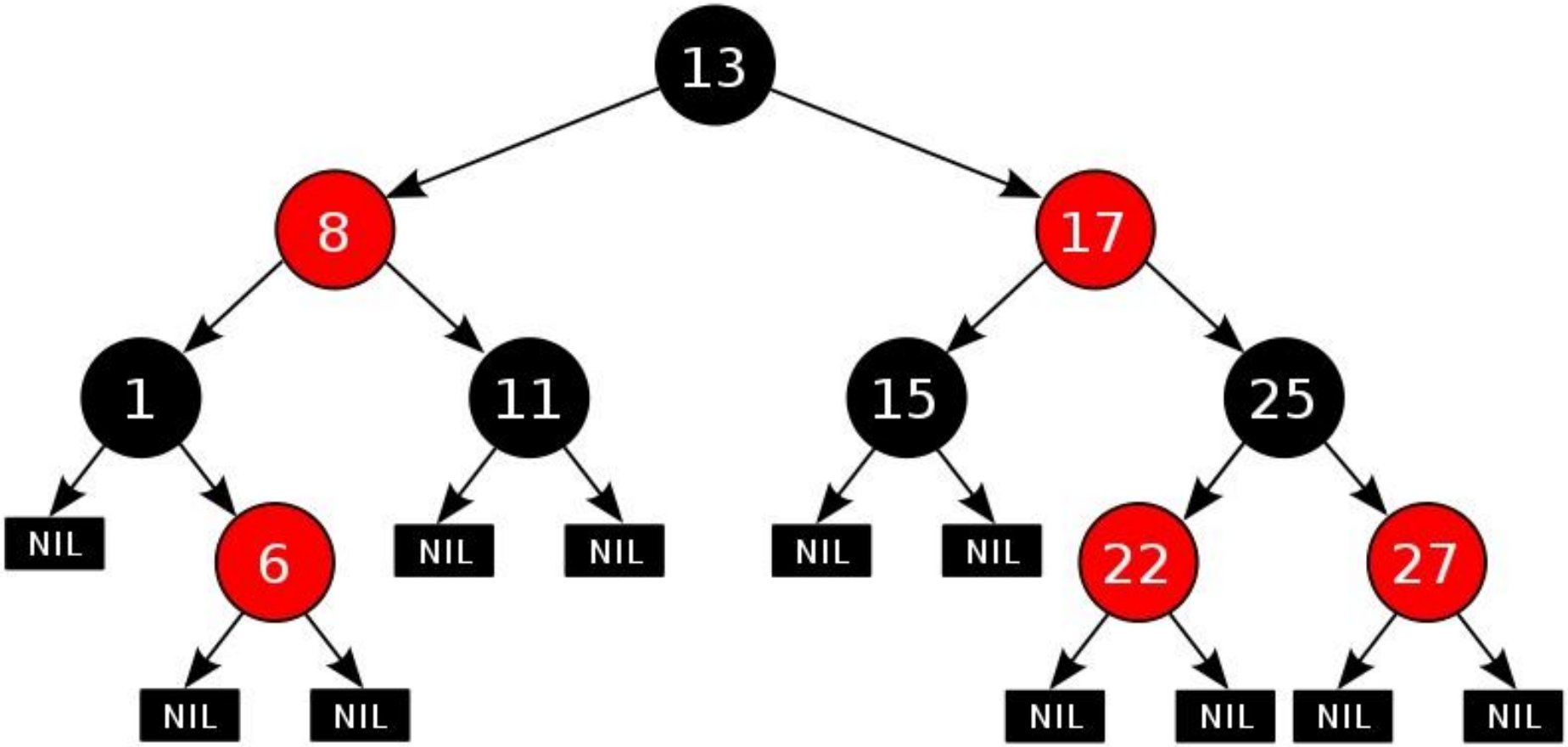
# Красно-черные и расширяющиеся деревья

Мазнев Валерий

# Содержание

- Красно-черные деревья
- Расширяющиеся деревья
- Рандомизированные деревья

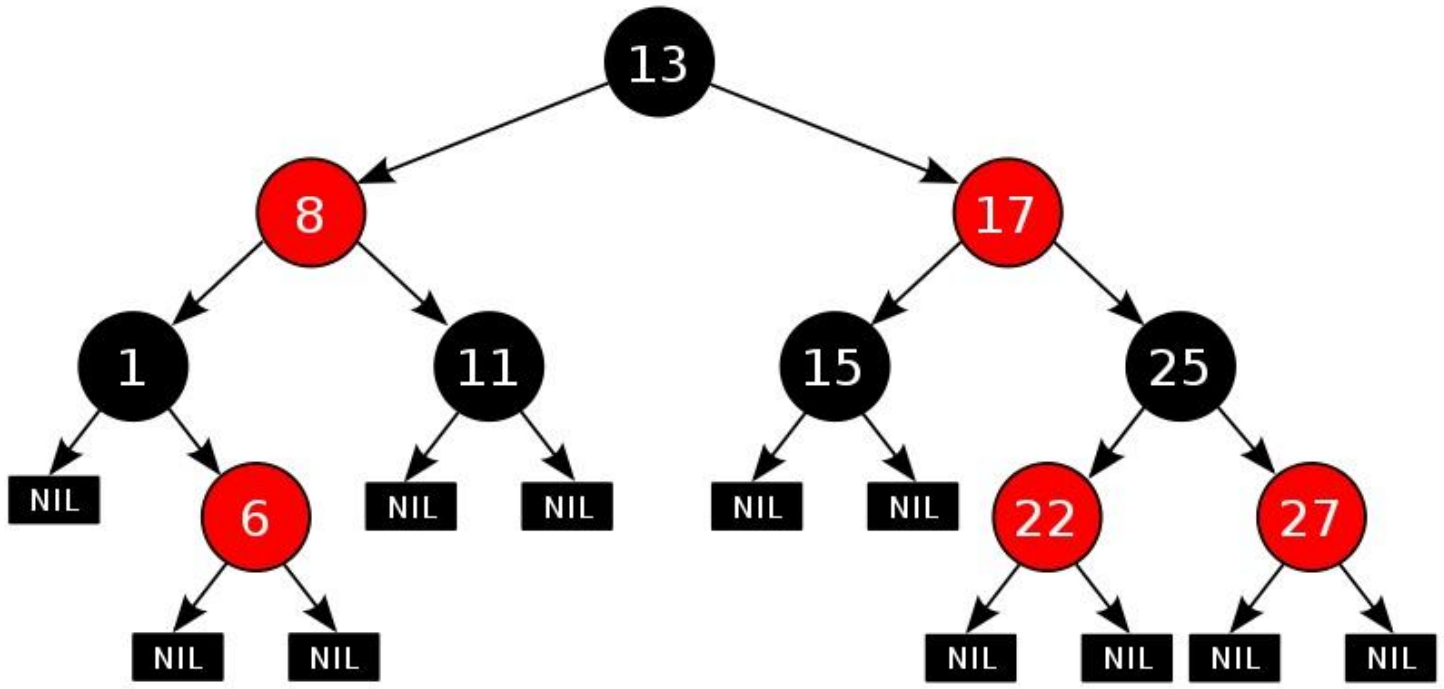
Это вид двоичного дерева поиска



# Свойства красно-черного дерева

- Узел либо красный, либо черный
- Корень всегда черный
- Все листья NIL - черные
- Оба потомка каждого красного узла — черные
- Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число черных узлов

- Примерно сбалансированное дерево
- Рудольф Байер, 1978 г.



# Вставка

- Как и в обычном дереве поиска
- Вставляемый узел всегда красный
- Проверяем, нарушились ли свойства дерева, и если да - то проводим балансировку

# Вспомогательная функция

- Получить дядю

```
Дядя:
    если родитель == родитель.родитель.левый то
        вернуть родитель.родитель.правый
    иначе
        вернуть родитель.родитель.левый
```

# Вспомогательная функция

- Получить деда

```
Дед:
        вернуть родитель.родитель
```

# Темнинология

- N - новый узел
- P - родитель
- G - дед
- U - дядя

# Родитель пусто

- Если новый узел- корень, то перекрашиваем

```
ПроверкаВставки()
    если родитель пусто
        цвет = черный
    иначе
        проверкаВставки2()
```

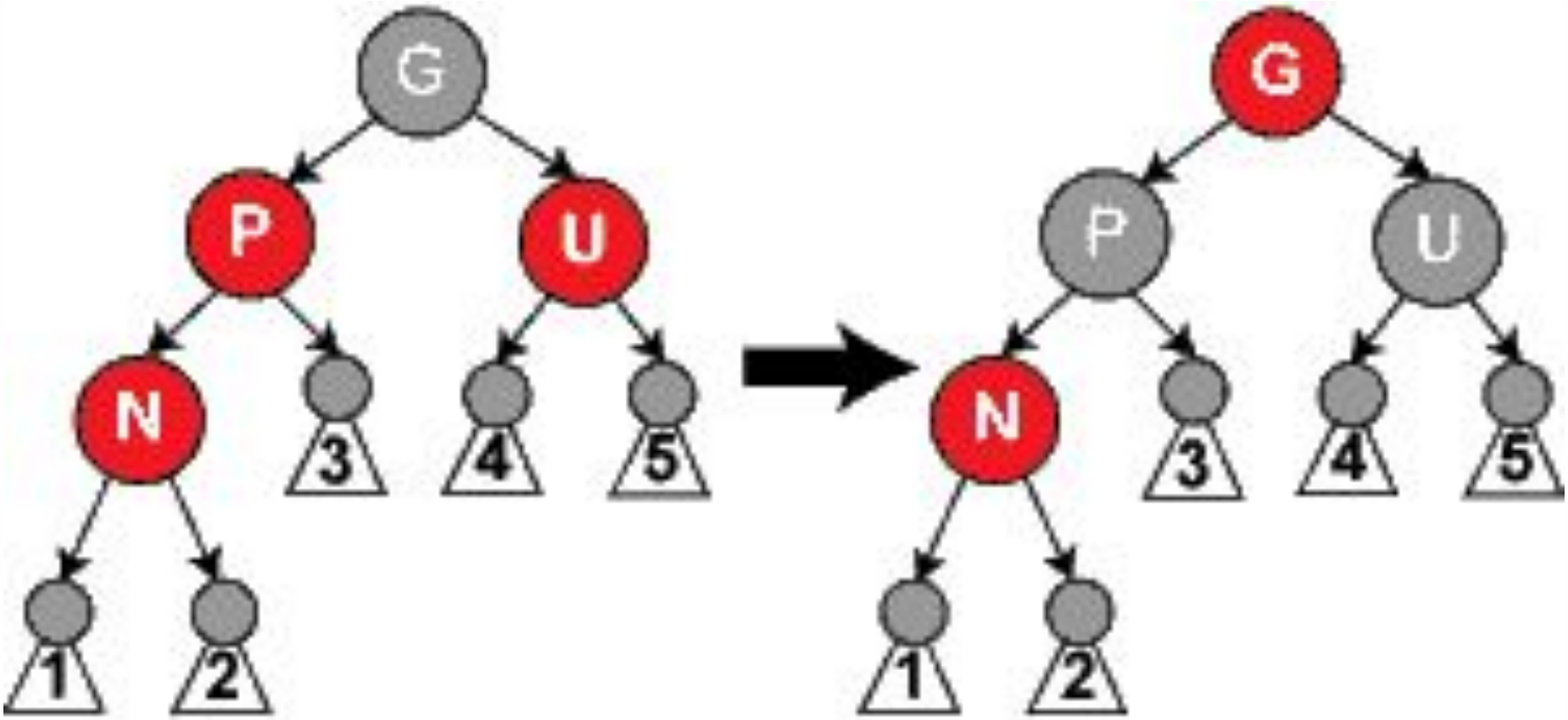
# Отец черный

- Если отец нового узла черный - то все ок

```
ПроверкаВставки2()
    если родитель.цвет == черный
        выход
    иначе
        проверкаВстаки3()
```

# Проверка/балансировка

- Если отец и дядя красные



## Отец и дядя красные

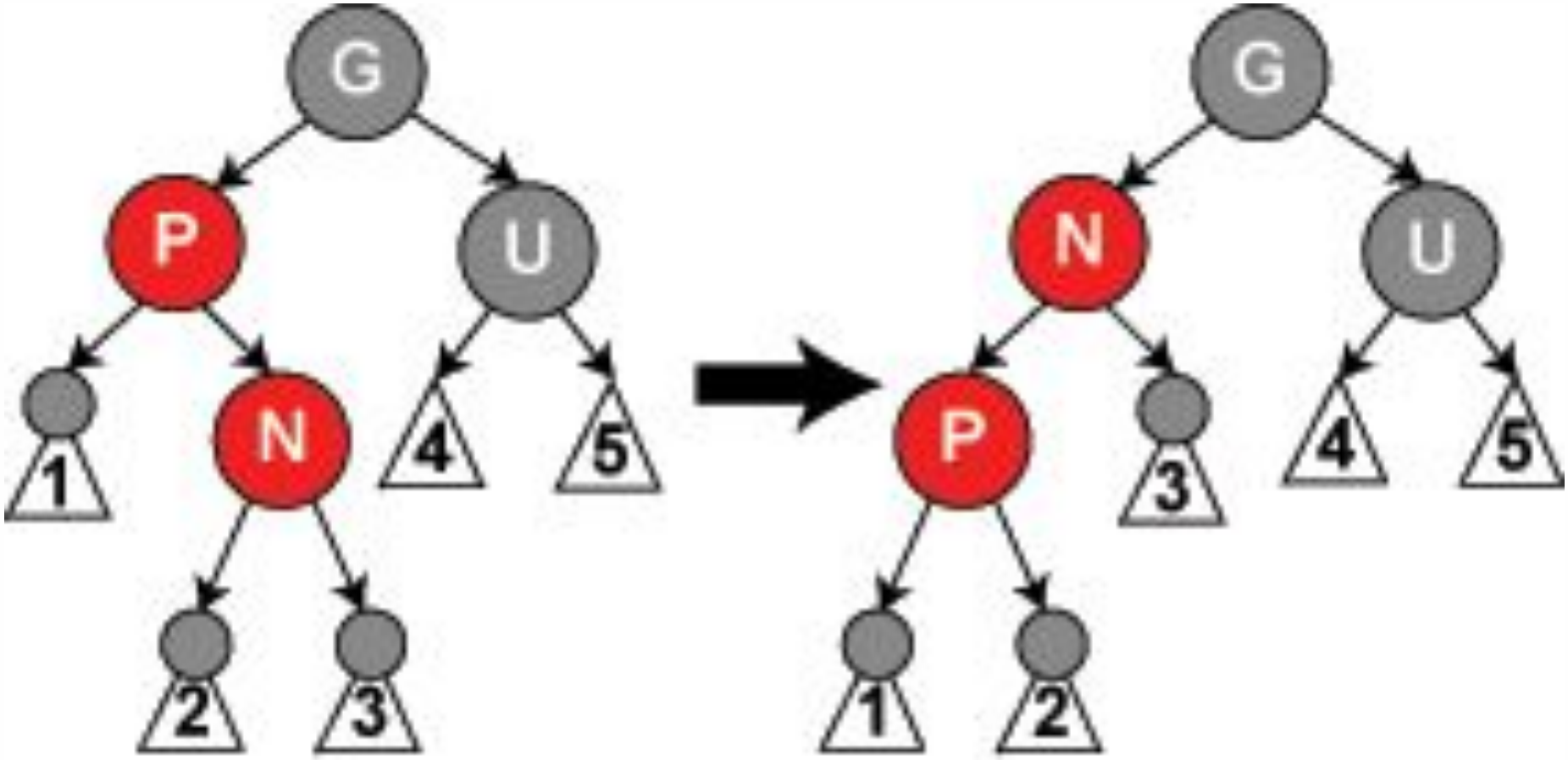
- Перекрасить отца и дядю в черный
- Перекрасить деда в красный
- Восстановить свойства дерева начиная с деда

- Перекрашиваем

```
ПроверкаВставки3()
    если дядя не пусто && дядя.цвет == черный
        родитель.цвет = черный
        дядя.цвет = черный
        дед.цвет = красный
        дед.ПроверкаВставки()
    иначе
        проверкаВстаки4()
```

## Родитель - красный, дядя — чёрный.

- Если мы левый, а родитель правый или наоборот

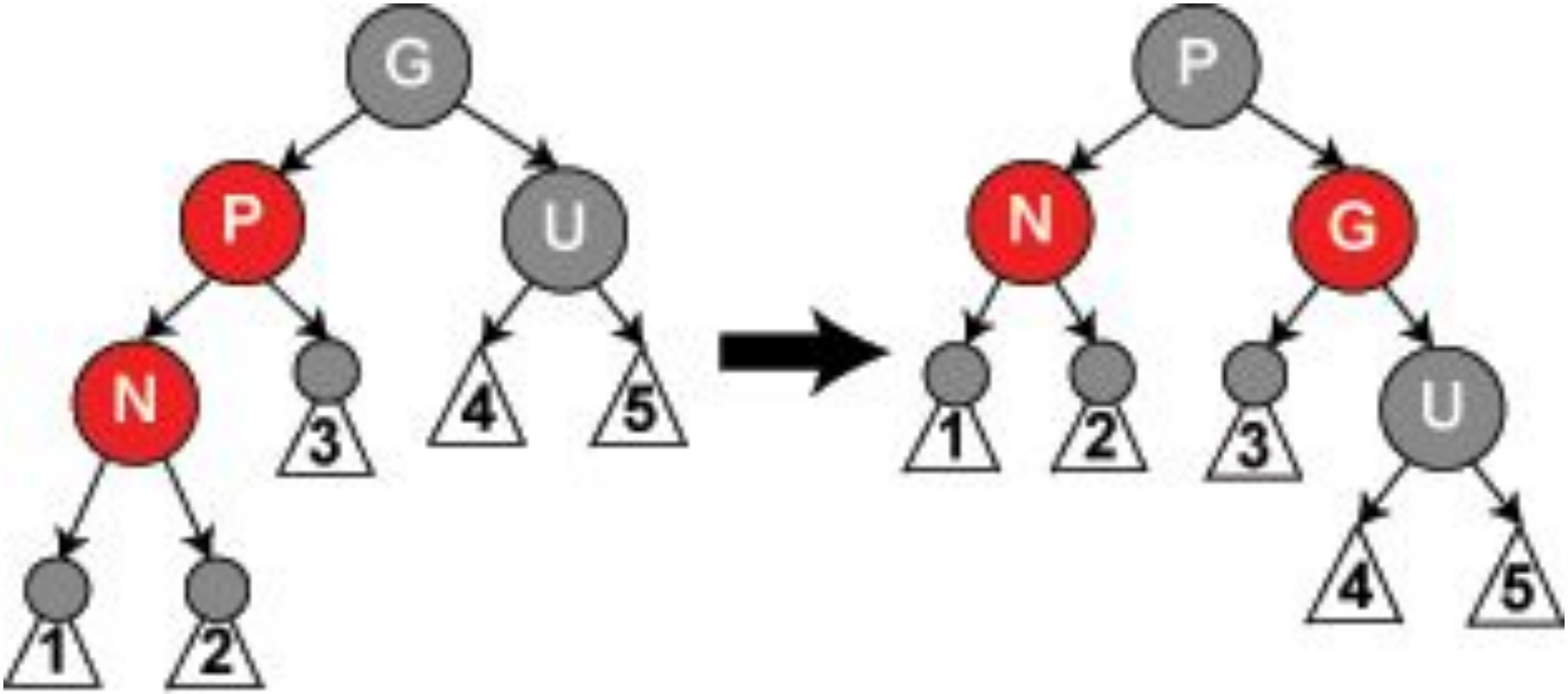


- Поворот

```
ПроверкаВставки4()
    если мы правый ребенок && отец левый ребенок
        отец.ВращатьВлево()
    иначе
        если мы левый ребенок && отец правый ребенок
            отец.ВращатьВправо()
… // продолжение следует
```

## Родитель - красный, дядя — чёрный.

- Если мы левый, и родитель левый или наоборот.



# Родитель красный, дядя черный (или дяди нет)

- Перекрасить
- Поворот относительно деда

```
родитель.цвет = черный
дед.цвет = красный
если мы левый потомок и отец левый потомок
    дед.ВращатьНаправо()
иначе
    дед.ВращатьНалево()
```

# Темнинология

- M - удаляемый узел
- C - потомок, которого нашли на замену
- N - новое положение этого потомка
- S - его брат
- Sl - левый потомок S
- Sr - правый потомок S
- P - родитель
- G - дед
- U - дядя

# Удаление

```
Удалить:
 если левый не пусто и правый не пусто то
    узел = найтиМинимальныйВПравом();
    узел.удалитьСОднимПотомком();
    узел.родитель = родитель
    узел.левый = левый
    узел.правый = правый
    узел.цвет = цвет
иначе
    удалитьСОднимПотомком();
```

# Вспомогательная функция

- Заменить узел

```
ЗаменитьУзел:
    узел.родитель = родитель
    узел.левый = левый
    узел.правый = правый
    узел.цвет = цвет
```

# Вспомогательная функция

- Получить брата

```
Брат:
    если текущий == родитель.левый то
        вернуть родитель.правый
    иначе
        вернуть родитель.левый
```

# Удаление одним потомком

OTUS

```
УдалитьСОднимПотомком:
    потомок = левый пусто ? правый : левый;

    ЗаменитьУзел(потомка);
    если цвет черный то
        если потомок.цвет == красный то
            потомок.цвет = черный
        иначе
            СбалансироватьПослеУдаления()
```

# СбалансироватьПослеУдаления

```
СбалансироватьПослеУдаления:
    если родитель не пусто то
        ...
```

# Удаление вершины

## Удаление вершины

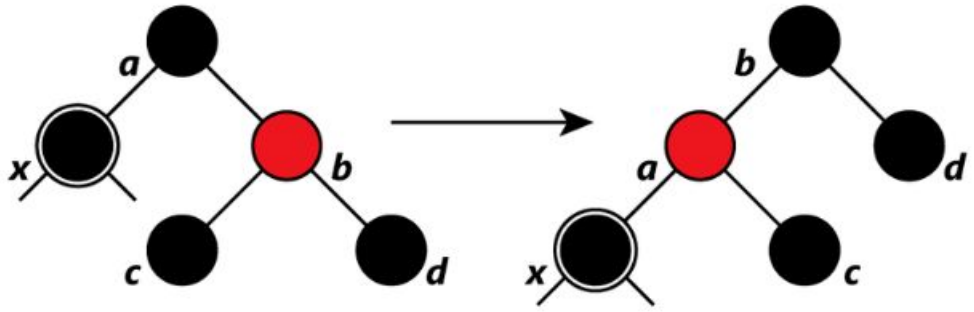
При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

- Если у вершины нет детей, то изменяем указатель на неё у родителя на $nil$.
- Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
- Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

# Балансировка после удаления вершины

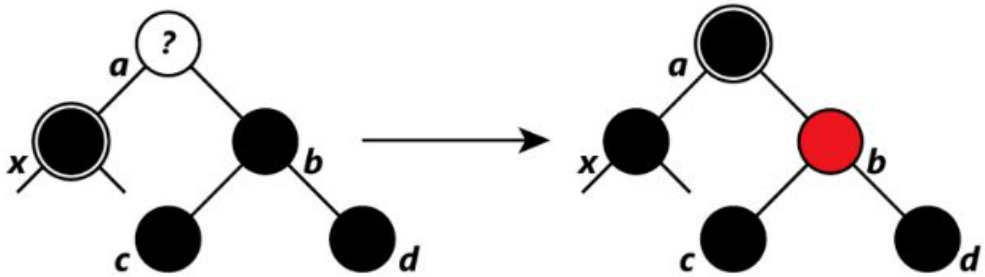
Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

- Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас $x$ имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.



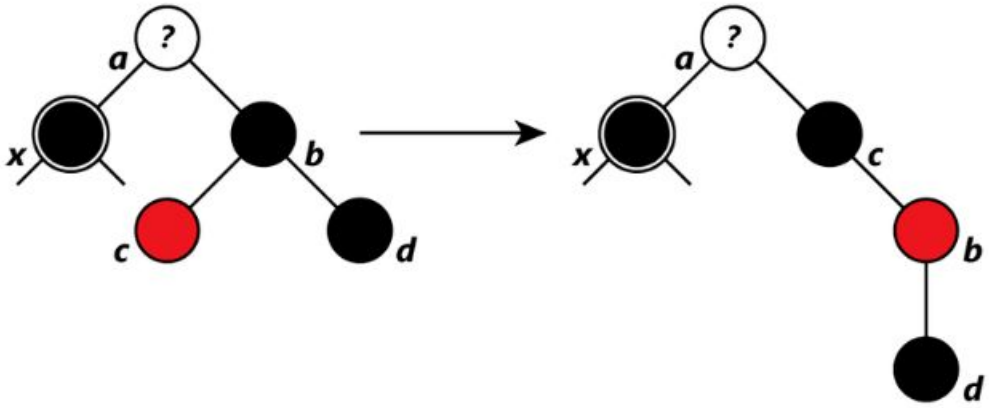
# Балансировка после удаления вершины

- Если брат текущей вершины был чёрным, то получаем три случая:
  - Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через $b$, но добавит один к числу чёрных узлов на путях, проходящих через $x$, восстанавливая тем самым влиянние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.



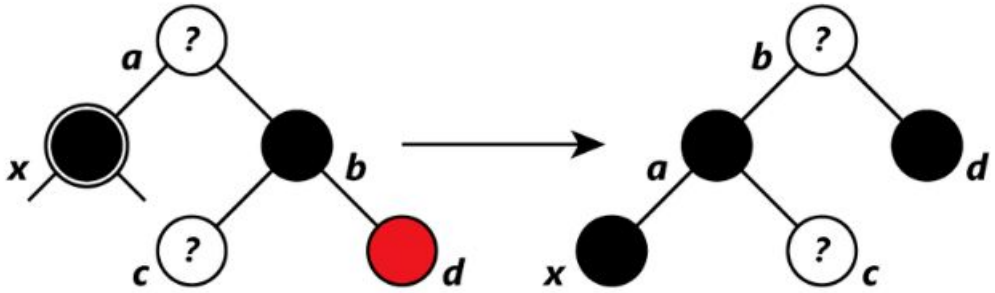
# Балансировка после удаления вершины

- Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у $x$ есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни $x$, ни его отец не влияют на эту трансформацию.

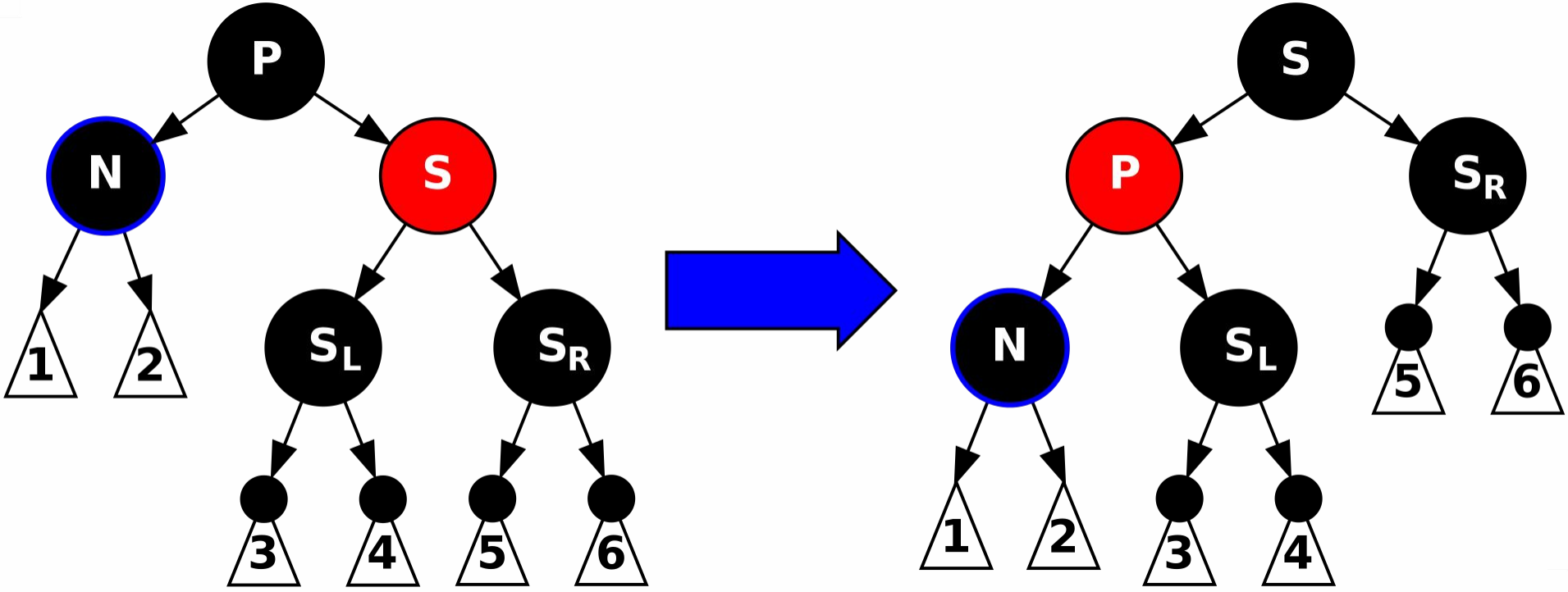


# Балансировка после удаления вершины

- Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство $3$ и $4$ не нарушаются. Но у $x$ теперь появился дополнительный чёрный предок: либо $a$ стал чёрным, или он и был чёрным и $b$ был добавлен в качестве чёрного дедушки. Таким образом, проходящие через $x$ пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.



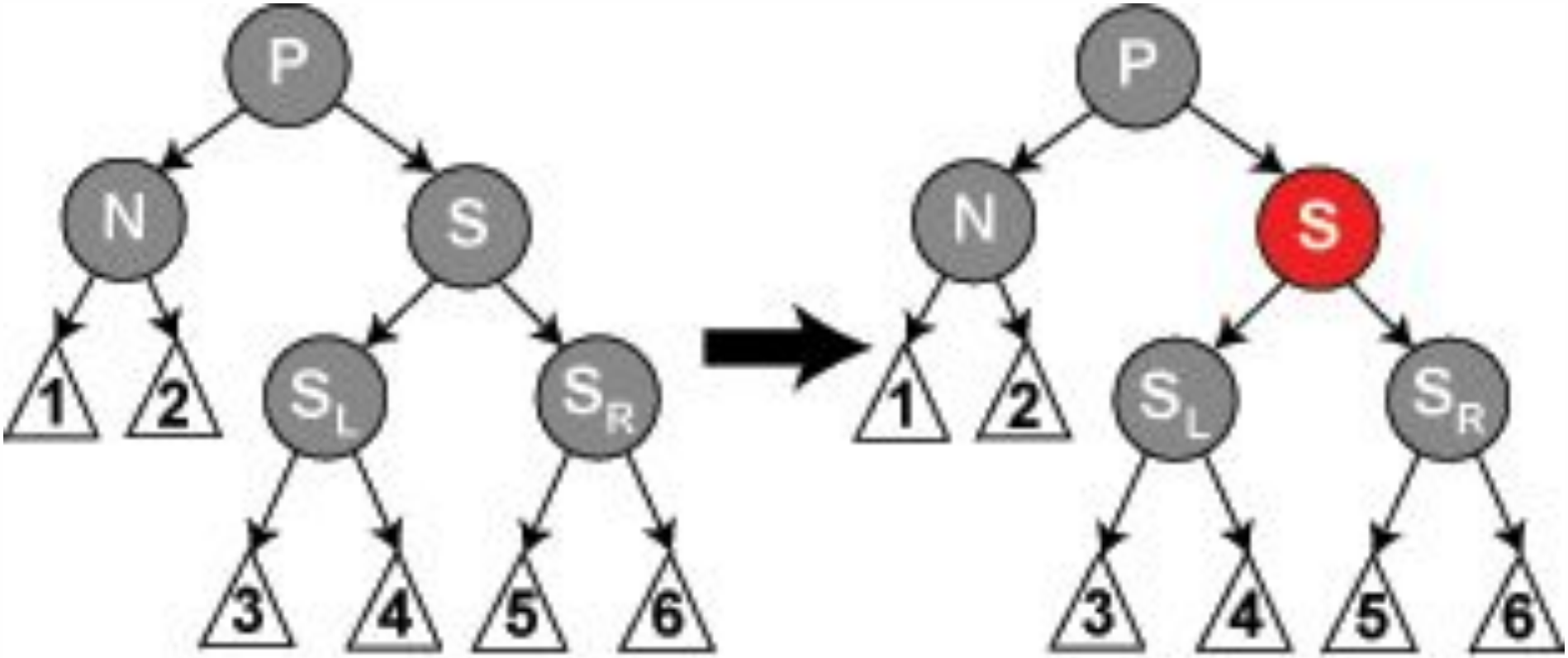
- S - красный



- S - красный

```
...
 брат = НайтиБрата();
 если брат.цвет == красный то
     родитель.цвет = красный
     брат.цвет = черный
     если текущий == родитель.левый то
         родитель.вращатьНалево()
     иначе
         родитель.вращатьНаправо()
...
```

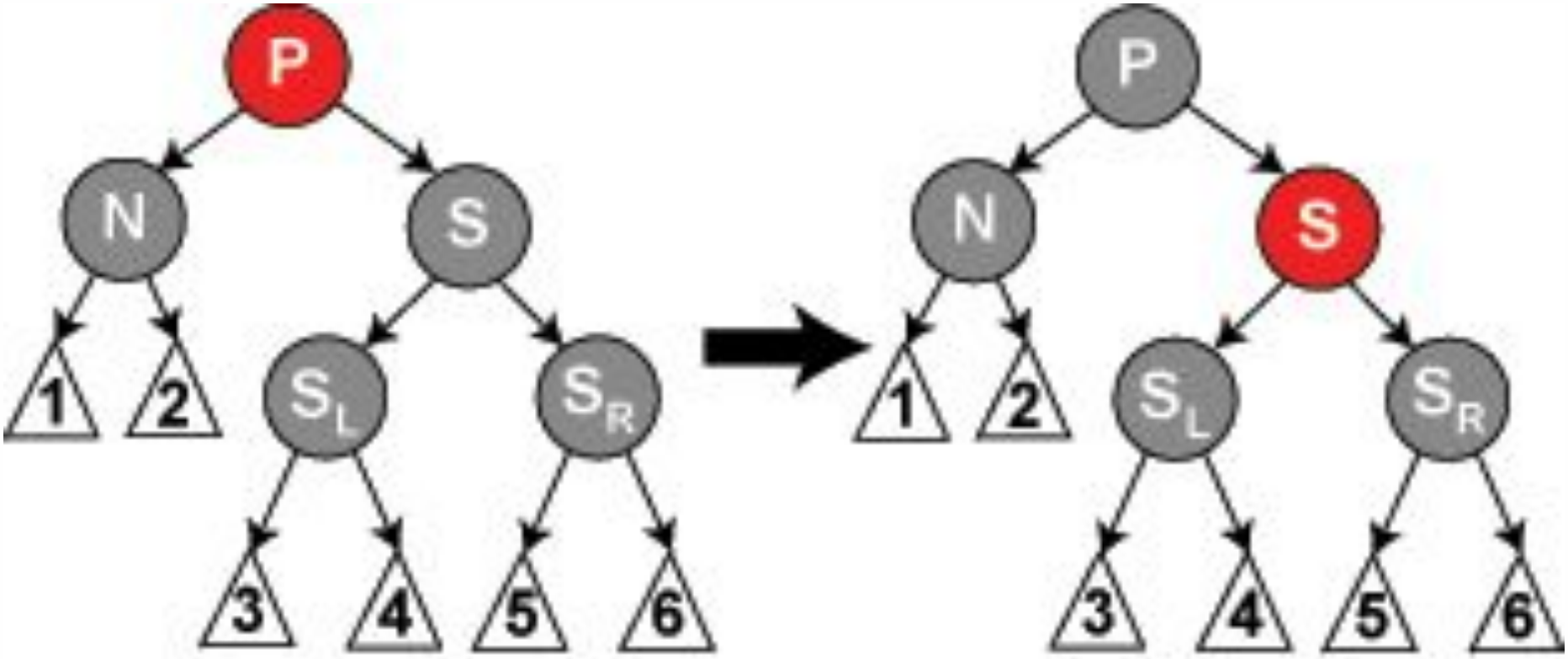
- S, P и Sl, Sr - черные



- S, P и Sl, Sr - черные

```
...
 если родитель.цвет == черный и
       брат.цвет == черный и
       брат.левый.цвет == черный и
       брат.правый.цвет == черный то
     брат.цвет = красный;
     родитель.СбалансироватьПослеУдаления()
 иначе
     ...
```

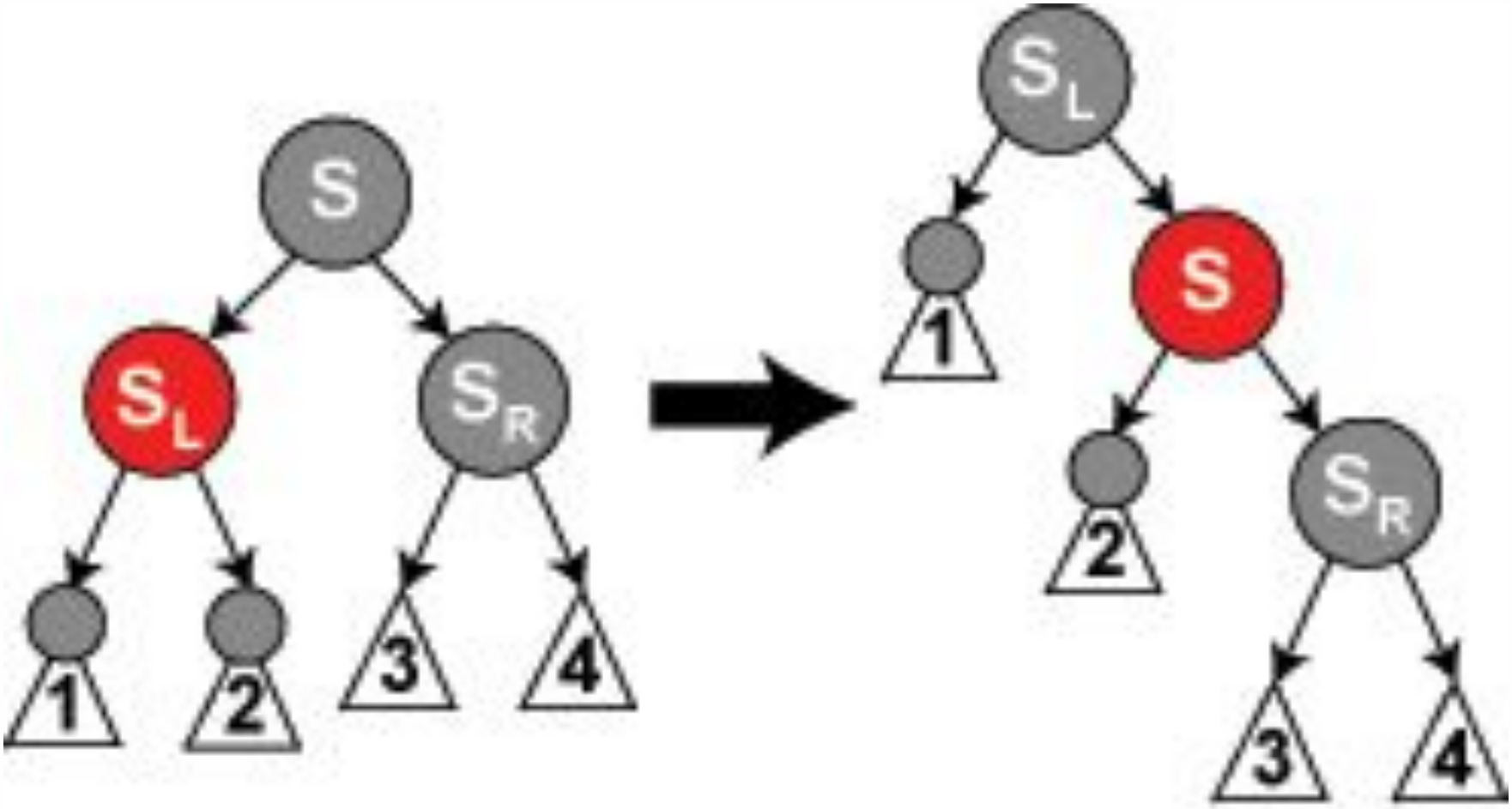
- S, Sl, Sr - черные, P - красный



- S, Sl, Sr - черные, P - красный

```
...
 если родитель.цвет == красный и
       брат.цвет == черный и
       брат.левый.цвет == черный и
       брат.правый.цвет == черный то
     брат.цвет = красный
     родитель.цвет = черный
 иначе
     ...
```

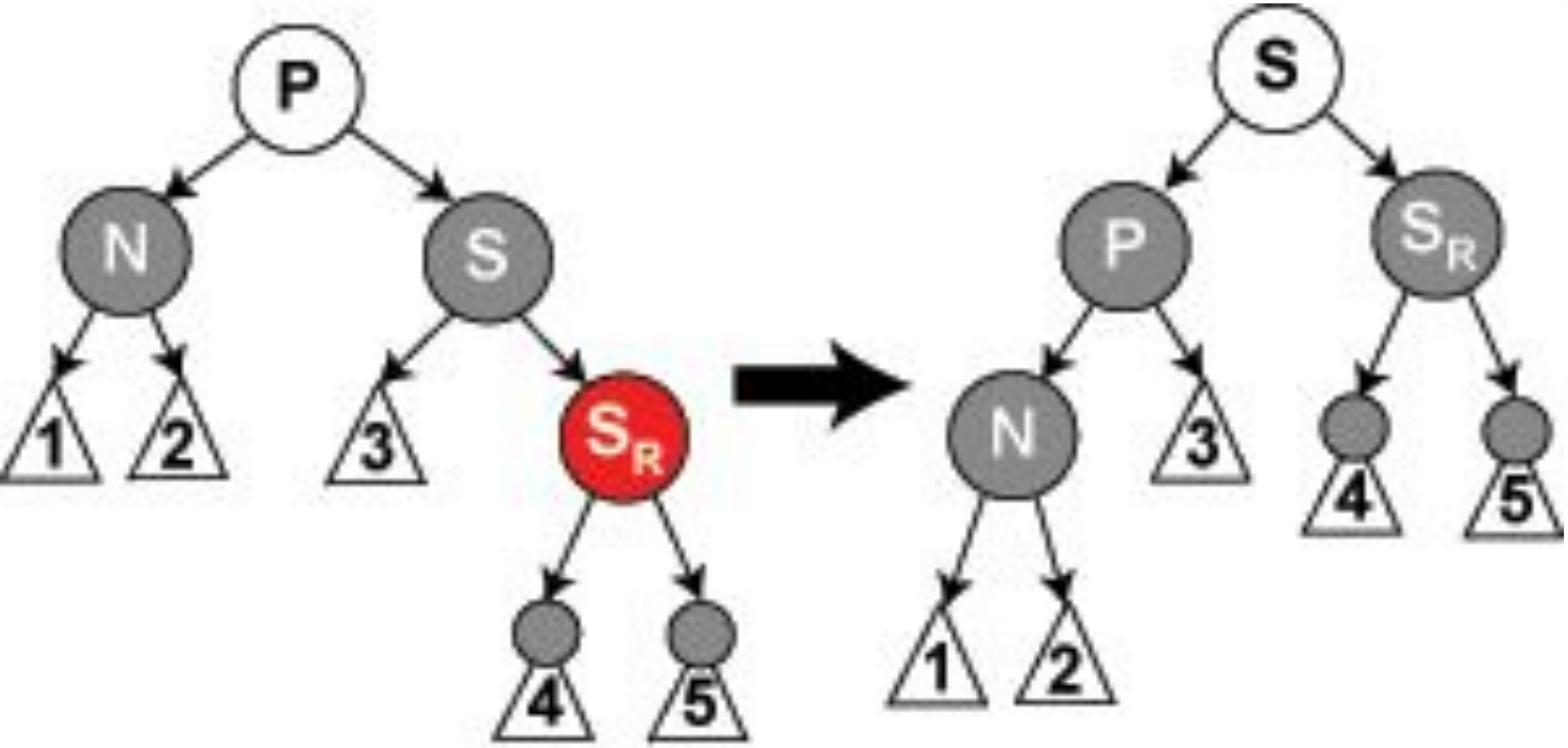
- S, Sr - черные, Sl - красный



- S, Sr - черные, Sl - красный

```
...
 если брат.цвет == черный то
      если брат.правый.цвет == черный и
           брат.левый.цвет == красный и
           текущий == родитель.левый то
        брат.цвет = красный
        брат.левый.цвет = черный
        брат.ВращатьВправо()
      иначе
        // то же самое зеркально
...
```

- S - черный, Sr - красный



# СбалансироватьПослеУдаления

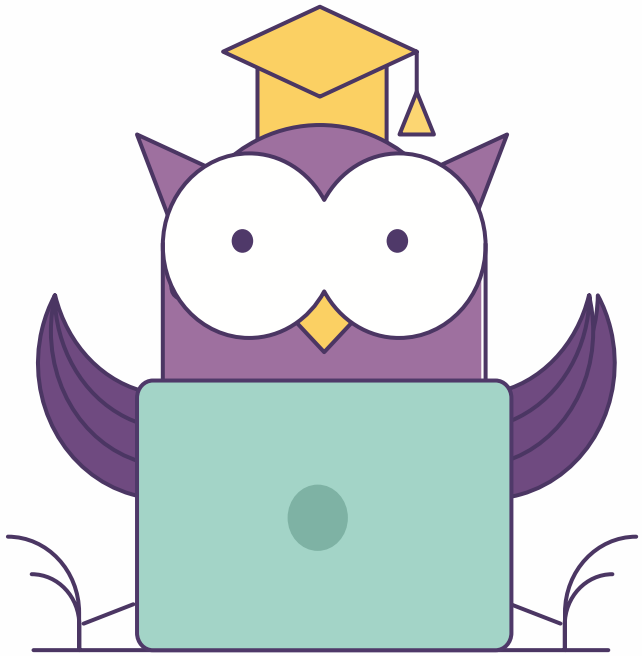
- S - черный, Sr - красный

```
...
 брат.цвет = родитель.цвет;
 родитель.цвет = черный;

 если текущий == родитель.левый то
        брат.правый.цвет = черный;
        родитель.ВращатьВлево();
 иначе
        брат.левый.цвет = черный;
        родитель.ВращатьВправо()
```

# Вопросы?

- Вопросы по красно-черным деревьям?



N - количество вершин,

h - высота дерева

Для АВЛ дерева

$$N(h) = \Theta(\lambda^h), \text{ где } \lambda = (\sqrt{5}+1)/2 \approx 1,62$$

Для красно-черного

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

Итого

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

# Тест производительности

| test set | representation | BST | AVL | RB |
|---|---|---|---|---|
| normal | plain | 5.22 | 4.62 | 4.49 |
| | parents | 4.97 | 4.47 | 4.33 |
| | threads | 5.00 | 4.63 | 4.51 |
| | right threads | 5.12 | 4.66 | 4.57 |
| | linked list | 5.05 | 4.79 | 4.59 |
| sorted | plain | * | 4.21 | 4.90 |
| | parents | * | 4.04 | 4.70 |
| | threads | * | 4.25 | 5.02 |
| | right threads | * | 4.24 | 5.02 |
| | linked list | * | 4.31 | 4.98 |
| shuffled | plain | 5.98 | 5.80 | 5.69 |
| | parents | 5.80 | 5.68 | 5.51 |
| | threads | 5.80 | 5.77 | 5.65 |
| | right threads | 5.89 | 5.80 | 5.71 |
| | linked list | 5.88 | 6.06 | 5.84 |

Table 6: Times, in seconds, for 5 runs of the unsorted and sorted versions of the cross-reference collator for each kind of tree. *Pathological case not measured.

| test set | representation | BST | AVL | RB |
|---|---|---|---|---|
| normal | plain | 3.24 | 3.09 | 3.01 |
| | parents | 3.10 | 2.94 | 2.86 |
| | threads | 3.12 | 3.04 | 2.98 |
| | right threads | 3.21 | 3.06 | 3.02 |
| | linked list | 3.19 | 3.21 | 3.08 |
| shuffled | plain | 3.23 | 3.28 | 3.24 |
| | parents | 3.13 | 3.12 | 3.05 |
| | threads | 3.15 | 3.20 | 3.15 |
| | right threads | 3.22 | 3.22 | 3.19 |
| | linked list | 3.24 | 3.43 | 3.32 |

Table 7: Times, in seconds, for 50 runs of a reduced version of the cross-reference collator for each kind of tree designed to fit within the processor cache.

P6 performance counters [24] directly confirms that reducing the test set size reduces additional cache misses in the shuffled case from 248% to 33%.

# Домашнее задание

- Реализовать красно-чёрное дерево
- Сравнить производительность

# Расширяющиеся деревья

- Это вид двоичного дерева поиска
- Splay - скошенный
- Не хранит дополнительной информации для поддержания структуры
- Роберт Тарьян и Даниель Слейтор в 1983 году

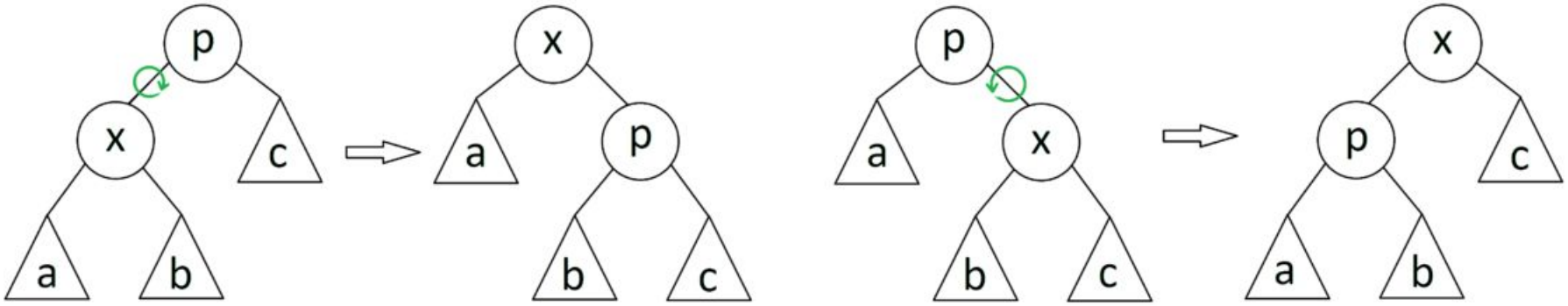
# Расширяющиеся деревья

- Подъем текущей вершины в корень при любой операции с деревом

- Вставка
- Удаление
- Поиск

# Зачем?

# Повороты налево и направо



# Splay

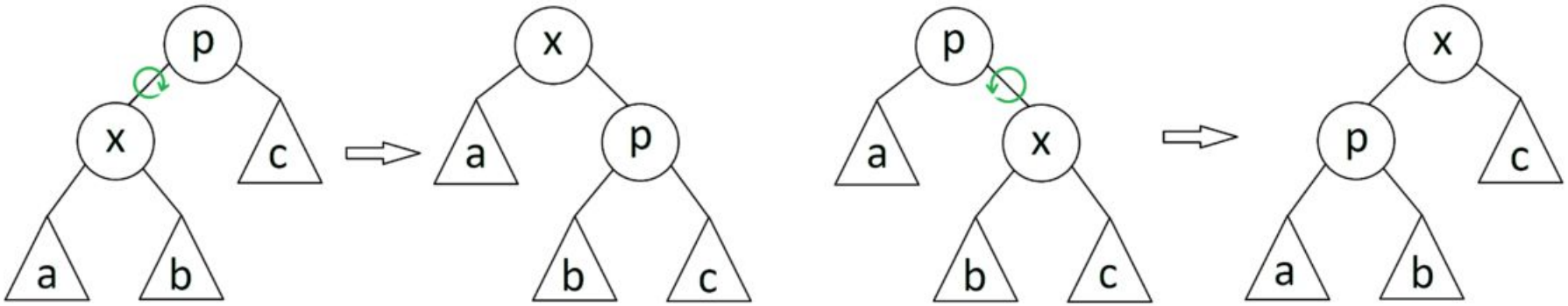
```
Перекосить():
  пока родитель не пусто то
      если родитель.левый == текущий то
          ВращатьНаправо()
      иначе
          ВращатьНалево()
```

# Splay - деревья

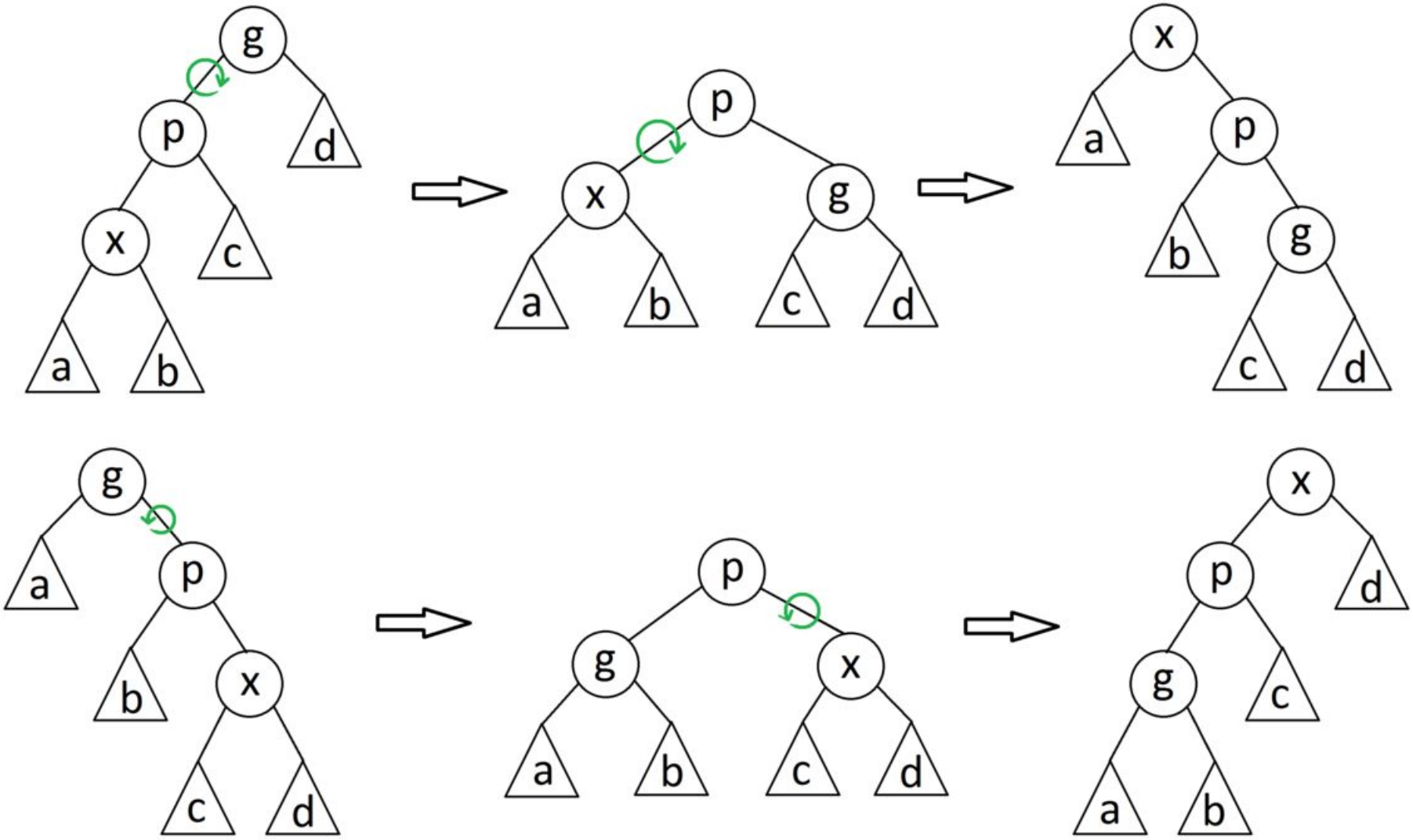
- Zig поворот
- Zig-Zig поворот
- Zig-Zag поворот

# Zig-поворот

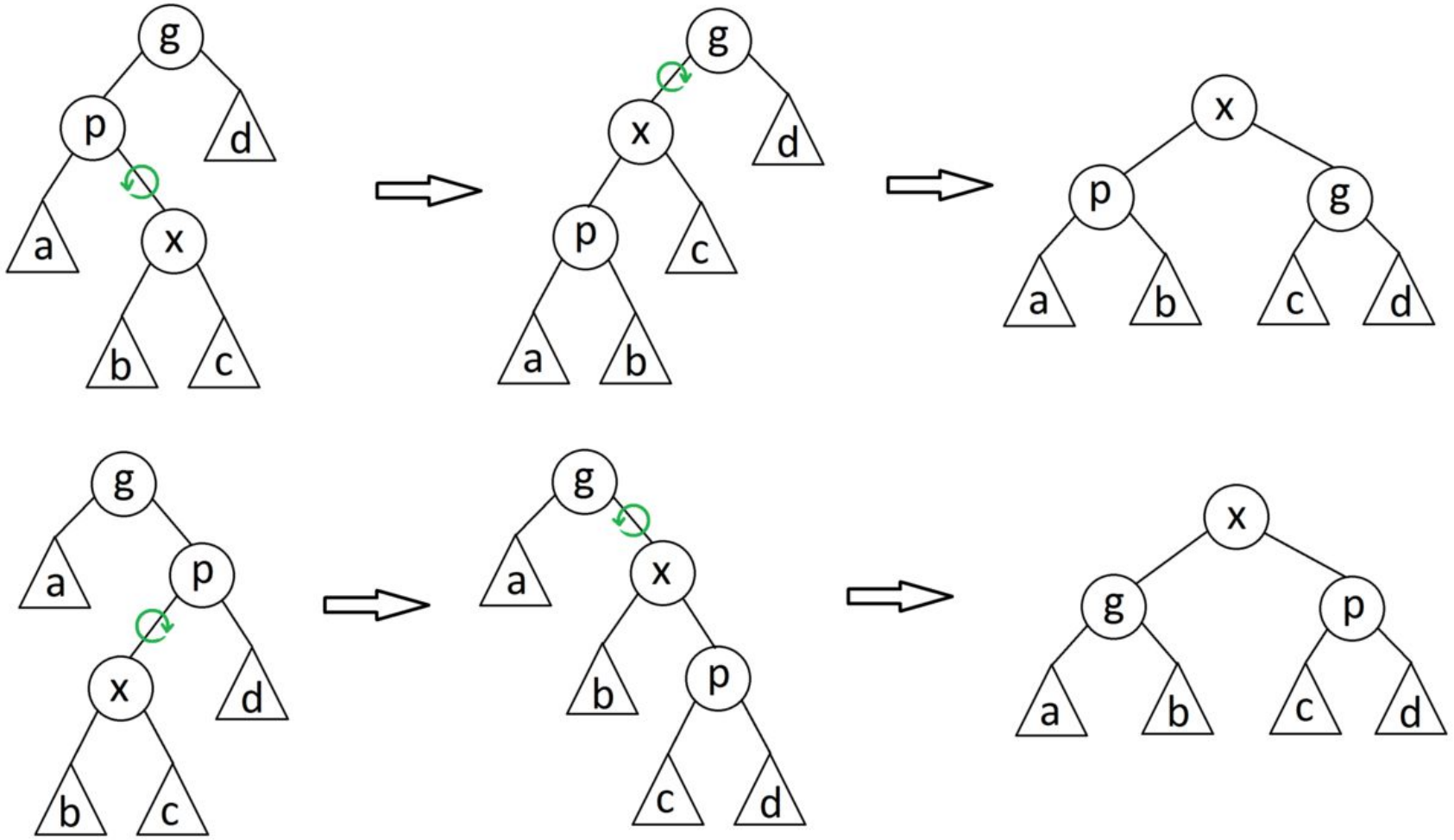
Выполняется только один раз, если высота дерева нечетная



# Zig-Zig поворот



# Zig-Zag поворот



# Splay

```
Перекосить():
  если родитель пусто то
    выход
  дед = родитель.родитель
  если дед пусто то
    Zig()
    выход
  иначе
    если (дед.левый==родитель) == (родитель.левый==узел) и
         (дед.правый==родитель) == (родитель.правый==узел) то
      ZigZig()
    иначе
      ZigZag()

    Перекосить()
```

# Поиск

- Как обычном дереве поиска
- В конце вызывается функция Splay

# Удаление

- Сначала вызывается функция Splay
- Удаляем корень
- Слияние поддеревьев

Пусть $q_i$ - число раз, которое запрошен элемент i

Тогда m запросов поиска выполняется

$$O\left(m + \sum_i q_i \log \frac{m}{q_i}\right)$$

Splay-дерево будет амортизационно работать не хуже, чем самое оптимальное фиксированное дерево

Пусть $t_j$ - число запросов, которое совершили к элементу j с момента x

Тогда m запросов поиска выполняется

$$O\left(m + n\log n + \sum_j \log(t_j + 1)\right)$$

В среднем недавно запрошенный элемент не уплывает далеко от корня

# Тест производительности

OTUS

| test set | representation | BST | AVL | RB | splay |
|---|---|---|---|---|---|
| normal | plain | 5.22 | 4.62 | 4.49 | 4.03 |
| | parents | 4.97 | 4.47 | 4.33 | 4.00 |
| | threads | 5.00 | 4.63 | 4.51 | 4.03 |
| | right threads | 5.12 | 4.66 | 4.57 | 4.06 |
| | linked list | 5.05 | 4.79 | 4.59 | 4.03 |
| sorted | plain | * | 4.21 | 4.90 | 2.91 |
| | parents | * | 4.04 | 4.70 | 2.90 |
| | threads | * | 4.25 | 5.02 | 2.89 |
| | right threads | * | 4.24 | 5.02 | 2.87 |
| | linked list | * | 4.31 | 4.98 | 2.83 |
| shuffled | plain | 5.98 | 5.80 | 5.69 | 6.54 |
| | parents | 5.80 | 5.68 | 5.51 | 6.61 |
| | threads | 5.80 | 5.77 | 5.65 | 6.56 |
| | right threads | 5.89 | 5.80 | 5.71 | 6.63 |
| | linked list | 5.88 | 6.06 | 5.84 | 6.64 |

Table 6: Times, in seconds, for 5 runs of the unsorted and sorted versions of the cross-reference collator for each kind of tree. *Pathological case not measured.

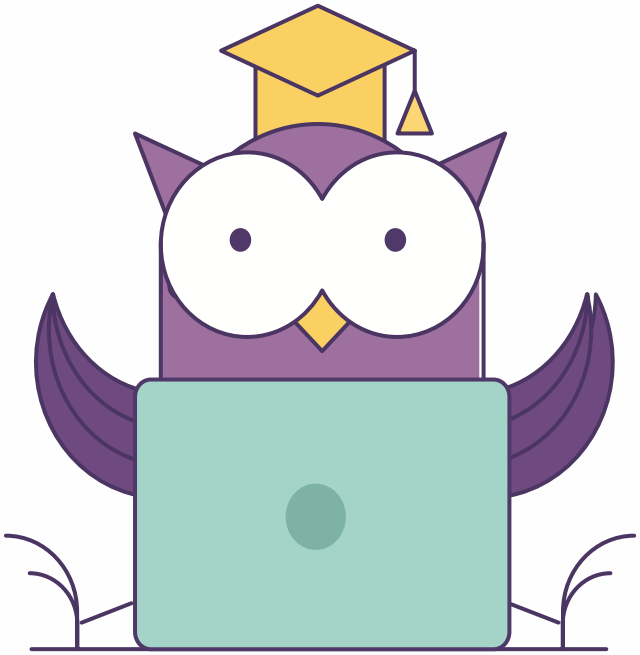
| test set | representation | BST | AVL | RB | splay |
|---|---|---|---|---|---|
| normal | plain | 3.24 | 3.09 | 3.01 | 2.80 |
| | parents | 3.10 | 2.94 | 2.86 | 2.81 |
| | threads | 3.12 | 3.04 | 2.98 | 2.82 |
| | right threads | 3.21 | 3.06 | 3.02 | 2.85 |
| | linked list | 3.19 | 3.21 | 3.08 | 2.87 |
| shuffled | plain | 3.23 | 3.28 | 3.24 | 3.51 |
| | parents | 3.13 | 3.12 | 3.05 | 3.55 |
| | threads | 3.15 | 3.20 | 3.15 | 3.55 |
| | right threads | 3.22 | 3.22 | 3.19 | 3.60 |
| | linked list | 3.24 | 3.43 | 3.32 | 3.64 |

Table 7: Times, in seconds, for 50 runs of a reduced version of the cross-reference collator for each kind of tree designed to fit within the processor cache.

P6 performance counters [24] directly confirms that reducing the test set size reduces additional cache misses in the shuffled case from 248% to 33%.

# Вопросы?

- Вопросы по splay деревьям?



# Рандомизированные деревья

- Это вид двоичного дерева поиска
- Хранит уровень, как АВЛ

# Insert

```
Вставить(ключ)
  если random() % (узел.высота+1) == 0 то
      ВставитьВКорень(узел);
  если текущий.ключ > ключ то
      если левый пусто то
          текущий.левый = новый Лист(ключ);
      иначе
          текущий.левый = левый.Вставка(ключ);
  иначе
      если правый пусто то
          текущий.правый = новый Лист(ключ);
      иначе
          текущий.правый = правый.Вставка(ключ);

  ПересчитатьВысоту();
```

# Вставить в корень

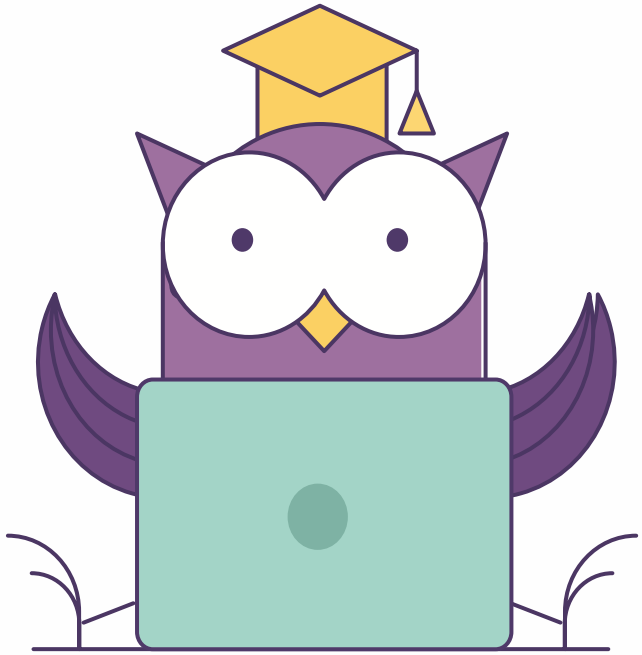
OTUS

```
ВставитьВКорень(ключ):
 если текущий.ключ == ключ то
     выход // либо заменить либо сохранить в список
 если текущий.ключ > ключ то
     если левый пусто то
         текущий.левый = новый Лист(ключ)
     иначе
         текущий.левый = левый.ВставитьВКорень(ключ)
     ВращатьНаправо()
 иначе
     если правый пусто то
         текущий.правый = новый Лист(ключ)
     иначе
         текущий.правый = правый.ВставитьВКорень(ключ)
     ВращатьНалево()
```

# Вставить в корень - а-ля splay

```
ВставитьВКорень(ключ):
 узел = обычнаяВставка(ключ)
 нашРодитель = родитель
 пока узел.родитель != нашРодитель то
    если узел.родитель.левый == узел то
        ВращатьНаправо()
    иначе
        ВращатьНалево()
```

- Вопросы по рандомизированным деревьям?



# Домашнее задание

- Реализовать красно-чёрное дерево
- Сравнить производительность

# Тест на производительность

- Время вставки
- Время поиска
- Время удаления
- Максимальная высота дерева

# Тест на вставку

- Вставка 5 млн случайных чисел
- Вставка 5 млн упорядоченных чисел
- Вставка данных из dataset
- Замер высоты дерева

# Тест на поиск

- Поиск 5 млн случайных чисел
- Поиск 5 тыс случайных чисел в цикле 1000 раз
- Поиск 5 млн случайных данных из dataset

# Тест на удаление

- Удаление 1 млн случайных чисел
- Замер высоты дерева

# Что мы сравниваем

- *Обязательная часть*
  АВЛ и красно черное

- Опциональная часть
  BST ||
  Splay ||
  Random

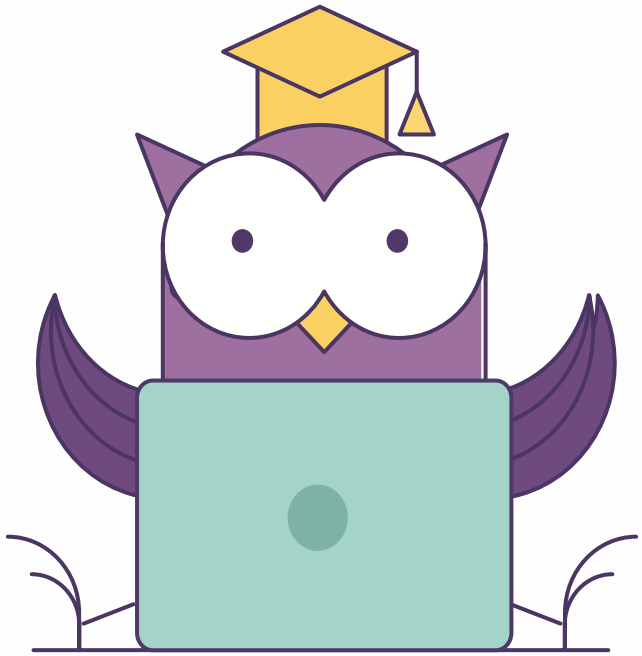
# Глобальный тест

- Выложить BST, Splay, Random в Slack
- Для Java - произвольный выбор
- Для Python и C++
  - договориться

# Глобальный тест

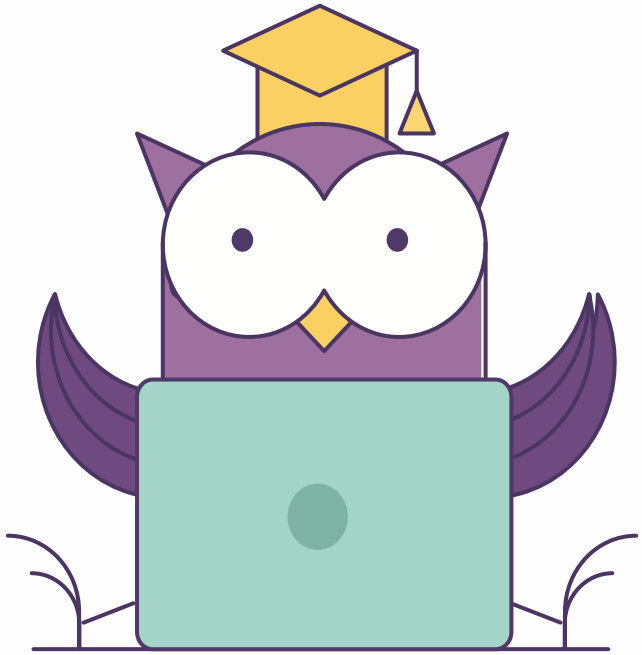
- Добавить Hash
- Добавить стандартную реализацию
- Дедлайн опциональной части - лекции по алгоритмам на графах
- Задам попроще ДЗ на следующее занятие

- Вопросы по ДЗ?



# Total recall

- Красно-черные деревья
- Расширяющиеся деревья
- Рандомизированные деревья



# Заполните, пожалуйста, опрос о занятии

# Спасибо
# за внимание!