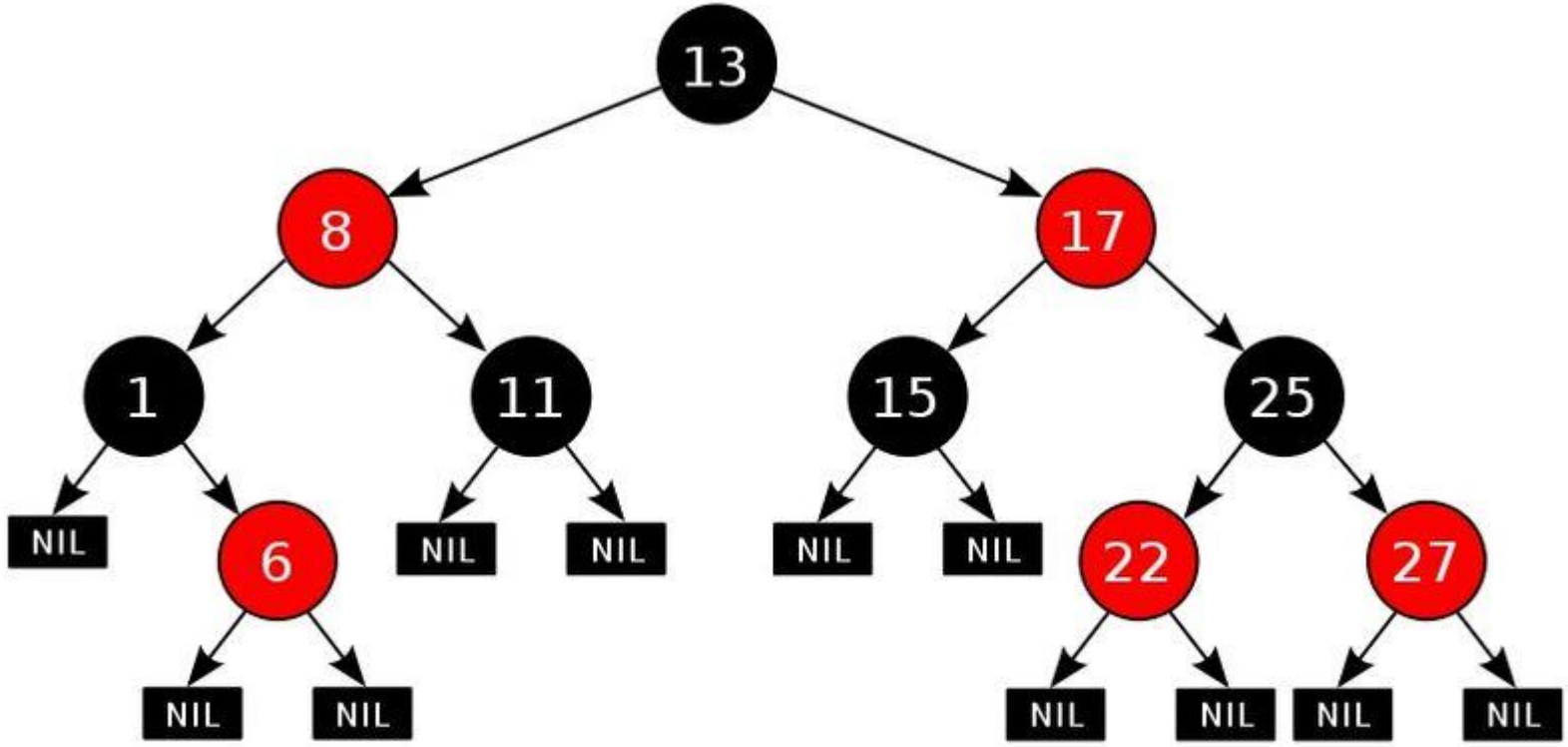
КРАСНО-ЧЕРНОЕ ДЕРЕВО

ОТ:

КЫЗЫЛЬСКОГО МЯСНИКА, ПЯТИ БУТЫЛОК ВОДКИ, БЕЗДОМНОЙ МОРТИРЫ, ФАНА ГУСАРОВОЙ И ПАТРИКА БЕЙТМЕНА

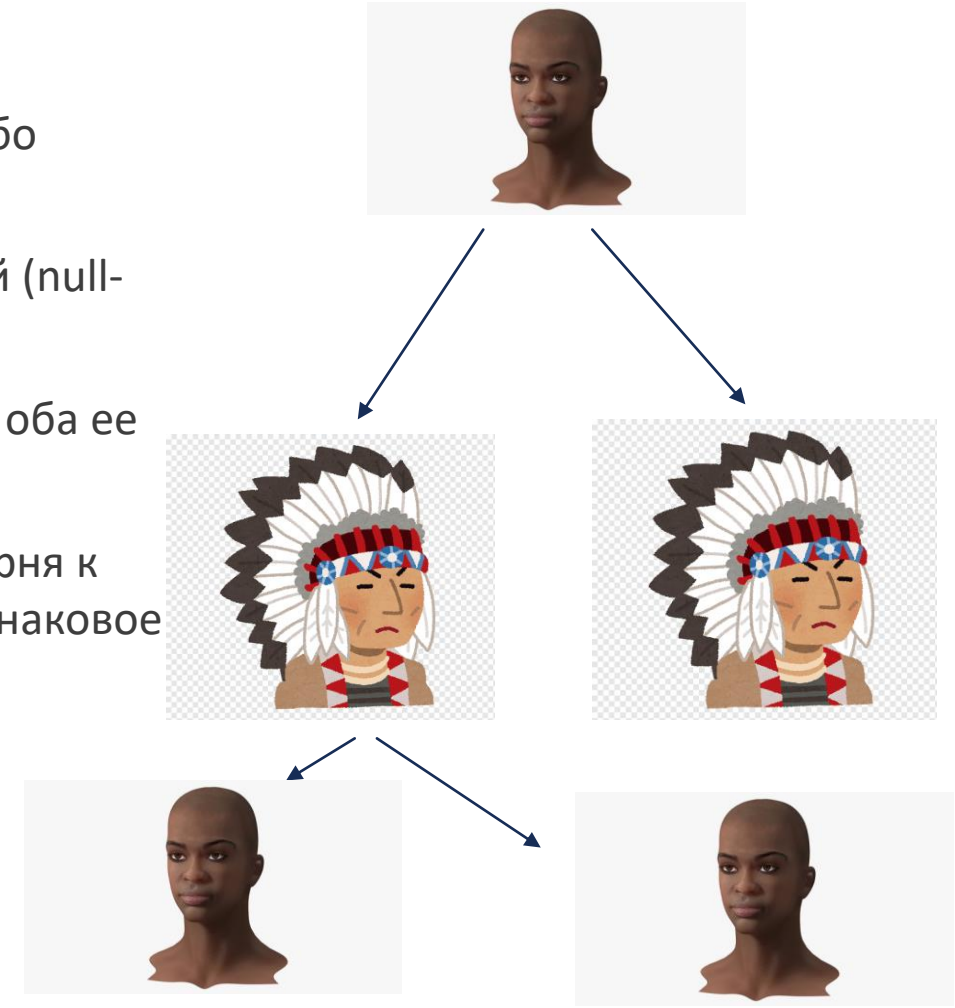
СВОЙСТВА


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

СВОЙСТВА

По Кормену:

1. Каждая вершина — либо красная, либо черная
2. Каждый лист — черный (null-узлы)
3. Если вершина красная, оба ее ребенка черные
4. Все пути, идущие от корня к листьям, содержат одинаковое количество черных вершин(черная высота



```
typedef struct node{
    int data;
    clr color;
    struct node *parent;
    struct node *left;
    struct node *right;
    node(int value,clr setColor) {
        data = value;
        left = right = nullptr;
        color = setColor;
    }
}binTree;
```

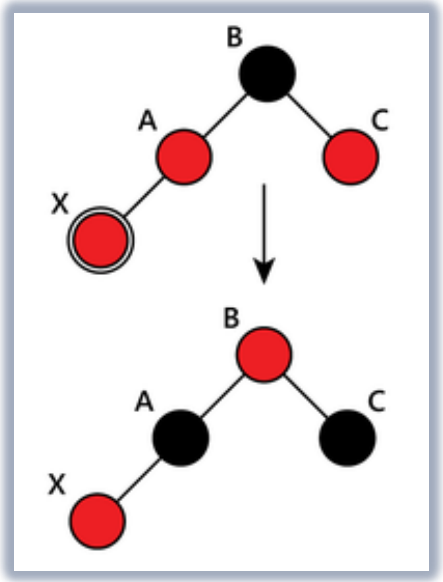
ВСТАВКА ЭЛЕМЕНТА (ПОИСК ПОДХОДЯЩЕГО МЕСТА)

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет *nullptr* (то есть этот сын — лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Теперь проверяем балансировку.

```
binTree *insert(binTree *root, int value) {
    binTree *newNode = new node(value, setColor: red); //конструктор создания новой ноды
    if(root == nullptr) {//создание корня дерева
        root = newNode;
        newNode->parent = nullptr;
    } else {
        binTree *tmp = root;
        binTree *parent = nullptr;
        while (tmp != nullptr) {//проход по дереву с поиском нужной позиции
            parent = tmp;
            if (tmp->data < newNode->data)
                tmp = tmp->right;
            else
                tmp = tmp->left;
        }
        newNode->parent = parent;//найдя нужную позицию прикрепляем новую ноду
        if(parent->data < newNode->data)
            parent->right = newNode;
        else
            parent->left = newNode;
    }
    fixInsert( node: newNode,root);//запускается функция балансировки дерева
    return root;
}
```

ВСТАВКА ЭЛЕМЕНТА (СЛУЧАЙ ПЕРВЫЙ)

```
binTree *fixInsert(binTree *node, binTree *root) {
    if(root->left == nullptr && root->right == nullptr) {
        node->color = black;
        return root; //выставление цвета для корня дерева
    }
    while(node->parent->color == red) {
        if(node->parent->parent->left == node->parent) {
            if(node->parent->parent->right && node->parent->parent->right->color == red) {
                //если дядя есть и он красный
                node->parent->color = black;//меняем цвет отца
                node->parent->parent->right->color = black;//так же меняем цвет дяди
                node->parent->parent->color = red;//меняем цвет деда
            }else {
```

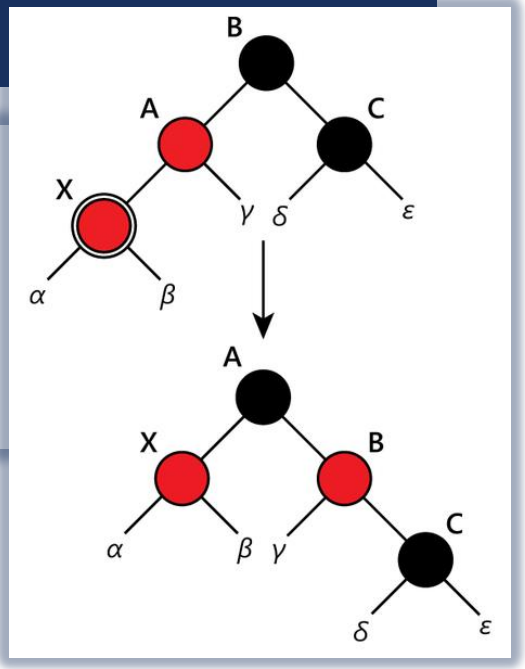


"Дядя" этого узла тоже красный. Тогда, чтобы сохранить свойства, просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы" черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству

ВСТАВКА ЭЛЕМЕНТА (СЛУЧАЙ ВТОРОЙ)

```
}else {
    if(node->parent->right == node) {
        node = node->parent;
        //меняем местами красные и черные элементы(отца и ребенка)
        // для сохранения черной высоты
        rotateLeft( root: node);
    }
    node->parent->color = black;
    node->parent->parent->color = red;
    rotateRight( root: node->parent->parent);
```

```
    else
        if t – правый сын
            t = parent
            leftRotate(t)
        parent = black
        grandfather = red
        rightRotate(grandfather)
```



"Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство и постоянство черной высоты сохраняются.

```
binTree *fixInsert(binTree *node, binTree *root) {
    if(root->left == nullptr && root->right == nullptr) {
        node->color = black;
        return root; //выставление цвета для корня дерева
    }
    while(node->parent->color == red) {
        if(node->parent->parent->left == node->parent) {
            if(node->parent->parent->right && node->parent->parent->right->color == red) {
                //если дядя есть и он красный
                node->parent->color = black;//меняем цвет отца
                node->parent->parent->right->color = black;//так же меняем цвет дяди
                node->parent->parent->color = red;//меняем цвет деда
            }else {
                if(node->parent->right == node) {
                    node = node->parent;
                    //меняем местами красные и черные элементы(отца и ребенка)
                    // для сохранения черной высоты
                    rotateLeft( root: node);
                }
                node->parent->color = black;
                node->parent->parent->color = red;
                rotateRight( root: node->parent->parent);

            }
        }else {
            if(node->parent->parent->left && node->parent->parent->left->color == red) {
                node->parent->color = black;
                node->parent->parent->left->color = black;
                node->parent->parent->color = red;
            }else {
                if(node->parent->left == node) {
                    node = node->parent;
                    rotateRight( root: node);
                }
                node->parent->color = black;
                node->parent->parent->color = red;
                rotateLeft( root: node->parent->parent);

            }
        }
    }
    root->color = black;
    return root;
```

```
func fixInsertion(t: Node)
    if t – корень
        t = black
        return
    // далее все предки упоминаются относительно t
    while "отец" красный // нарушается свойство 3
        if "отец" – левый ребенок
            if есть красный "дядя"
                 parent = black
                 uncle = black
                 grandfather = red
                 t = grandfather
            else
                if t – правый сын
                    t = parent
                    leftRotate(t)
                parent = black
                grandfather = red
                rightRotate(grandfather)
        else // "отец" – правый ребенок
            if есть красный "дядя"
                parent = black
                uncle = black
                grandfather = red
                t = grandfather
            else // нет "дяди"
                if t – левый ребенок
                    t = t.parent
                    rightRotate(t)
                parent = black
                grandfather = red
                leftRotate(grandfather)
    root = black // восстанавливаем свойство корня
```

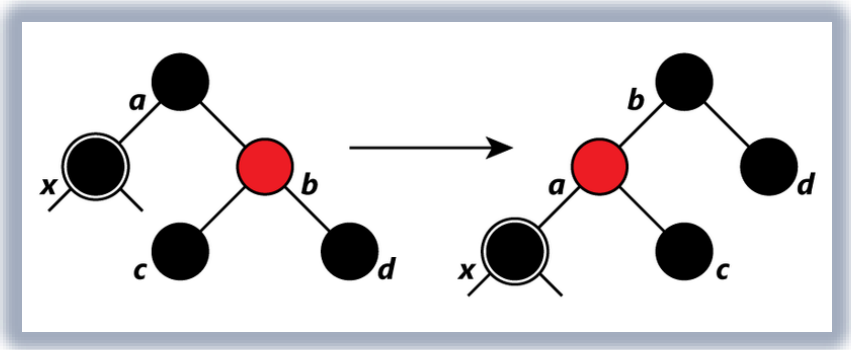
УДАЛЕНИЕ ЭЛЕМЕНТА

Удаление происходит по тому же принципу что и в наивной реализации бинарного дерева поиска, однако если мы удаляем черный элемент, то изменяется черная высота, следовательно следует отбалансировать дерево

```
func delete(key)
    Node p = root
    // находим узел с ключом key
    while p.key != key
        if p.key < key
            p = p.right
        else
            p = p.left
    if у p нет детей
        if p – корень
            root = nil
        else
            ссылку на p у "отца" меняем на nil
        return
    Node y = nil
    Node q = nil
    if один ребенок
        ссылку на у от "отца" меняем на ребенка у
    else
        // два ребенка
        у = вершина, со следующим значением ключа // у нее нет левого ребенка
        if у имеет правого ребенка
            y.right.parent = y.parent
        if у – корень
            root = y.right
        else
            у родителя ссылку на у меняем на ссылку на первого ребенка у
    if y != p
        p.colour = y.colour
        p.key = y.key
    if y.colour == black
        // при удалении черной вершины могла быть нарушена балансировка
        fixDeleting(q)
```

УДАЛЕНИЕ(ЕСЛИ БРАТ КРАСНЫЙ)

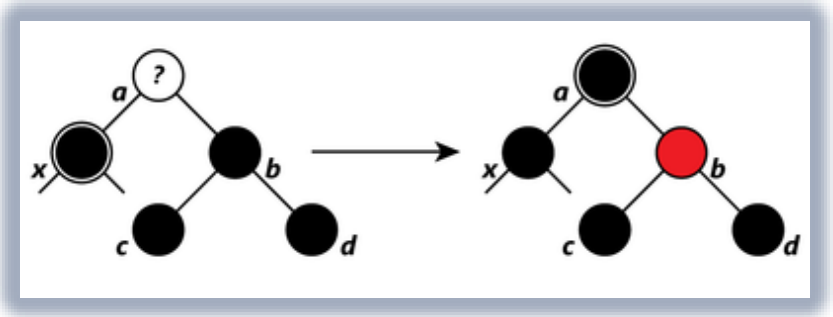
```
func fixDeleting(p: Node)
    // далее родственные связи относительно p
    while p – черный узел и не корень
        if p – левый ребенок
            if "брат" красный
                brother = black
                parent = red
                leftRotate(parent)
```



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.

УДАЛЕНИЕ С ЧЕРНЫМ БРАТОМ(ПЕРВЫЙ СЛУЧАЙ)

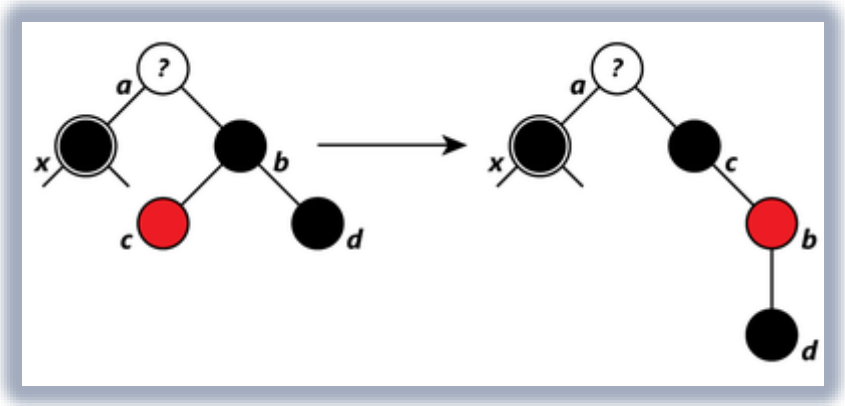
```
    leftRotate(parent)
if у "брата" черные дети            // случай 1: "брат" красный с черными детьми
    brother = red
else
```



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.

УДАЛЕНИЕ С ЧЕРНЫМ БРАТОМ(ВТОРОЙ СЛУЧАЙ)

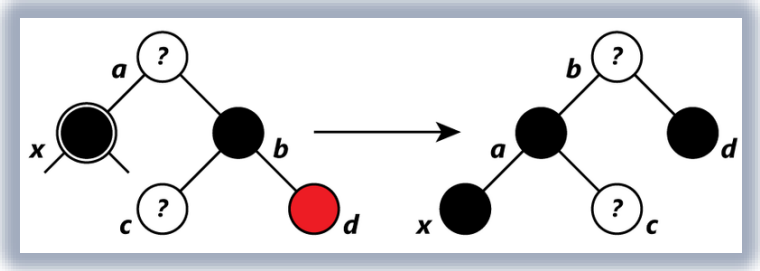
```
else
    if правый ребенок "брата" черный // случай, рассматриваемый во втором подпункте:
        brother.left = black         // "брат" красный с черными правым ребенком
        brother = red
        rightRotate(brother)
```



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.

УДАЛЕНИЕ С ЧЕРНЫМ БРАТОМ(ТРЕТИЙ СЛУЧАЙ)

```
        brother.colour = parent.colour   // случай, рассматриваемый в последнем подпункте
        parent = black
        brother.right = black
        leftRotate(parent)
        p = root
```



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 свойства 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.

СХЕМА УДАЛЕНИЯ

Удалили черный?

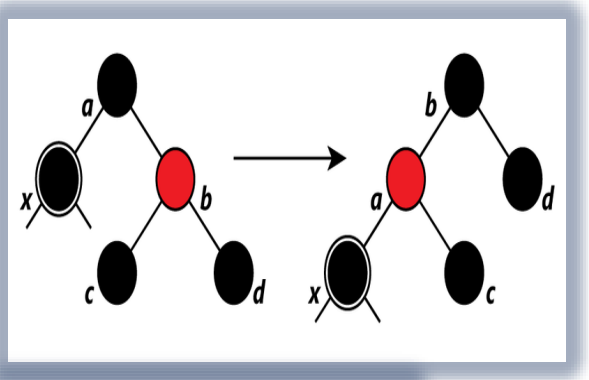
Да

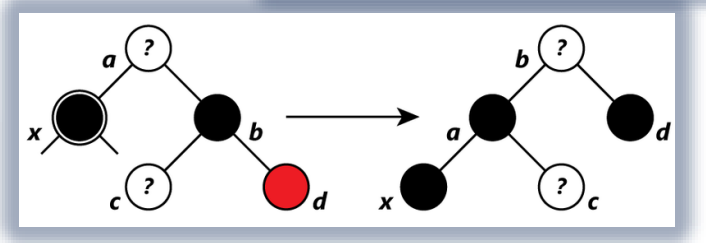
Нет

Брат черный?

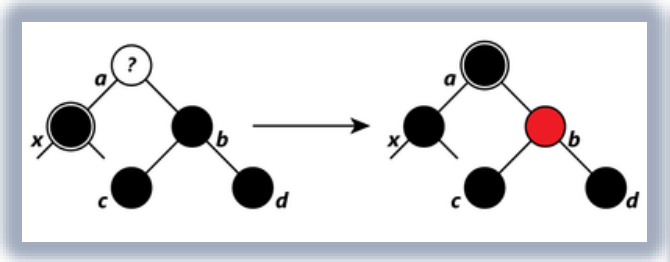
Нет

Да

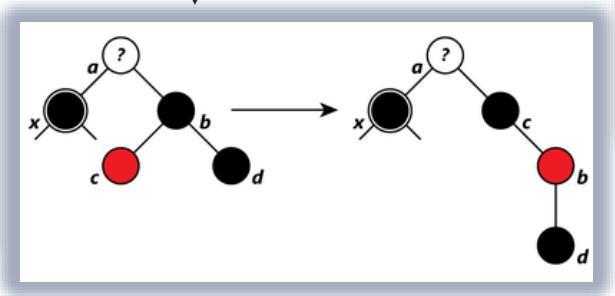




Правый ребенок красный



Оба ребенка у брата черные



Правый ребенок черный

```
func delete(key)
    Node p = root
    // находим узел с ключом key
    while p.key != key
        if p.key < key
            p = p.right
        else
            p = p.left
    if у p нет детей
        if p – корень
            root = nil
        else
            ссылку на p у "отца" меняем на nil
        return
    Node y = nil
    Node q = nil
    if один ребенок
        ссылку на у от "отца" меняем на ребенка у
    else
        // два ребенка
        y = вершина, со следующим значением ключа // у нее нет левого ребенка
        if у имеет правого ребенка
            y.right.parent = y.parent
        if y – корень
            root = y.right
        else
            у родителя ссылку на у меняем на ссылку на первого ребенка у
    if y != p
        p.colour = y.colour
        p.key = y.key
    if y.colour == black
        // при удалении черной вершины могла быть нарушена балансировка
        fixDeleting(q)
```

```
func fixDeleting(p: Node)
    // далее родственные связи относительно p
    while p – черный узел и не корень
        if p – левый ребенок
            if "брат" красный
                brother = black
                parent = red
                leftRotate(parent)
            if у "брата" черные дети              // случай 1: "брат" красный с черными детьми
                brother = red
            else
                if правый ребенок "брата" черный // случай, рассматриваемый во втором подпункте:
                    brother.left = black          // "брат" красный с черными правым ребенком
                    brother = red
                    rightRotate(brother)
                brother.colour = parent.colour    // случай, рассматриваемый в последнем подпункте
                parent = black
                brother.right = black
                leftRotate(parent)
                p = root
        else // p – правый ребенок
            // все случаи аналогичны тому, что рассмотрено выше
            if "брат" красный
                brother = black
                parent = red
                rightRotate(p.parent)
            if у "брата" черные дети
                brother = red
            else
                if левый ребенок "брата" черный
                    brother.right = black
                    brother = red
                    leftRotate(brother);
                brother = parent
                parent = black
                brother.left = black
                rightRotate(p.parent)
                p = root
      p = black
      root = black
```

СПАСИБО ЗА ВНИМАНИЕ!!!