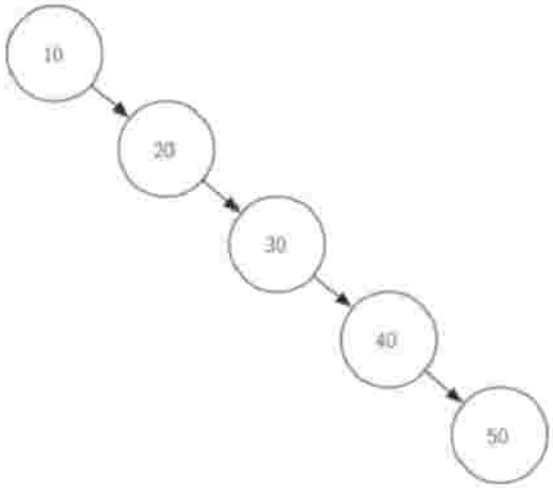
Binary tree algorithms: AVL, RB-tree, Heavy-light

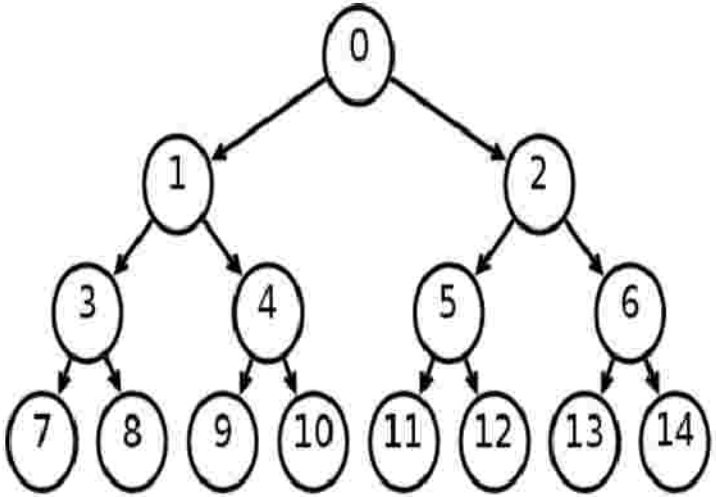
Morozov Ilya
Kameka Valery
Paramonov Anton

AVL tree

Problem

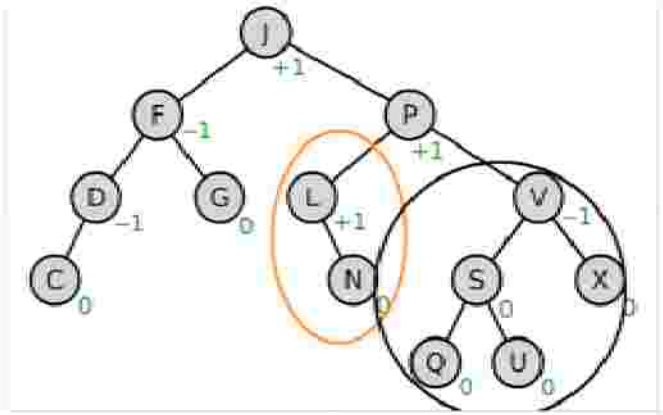


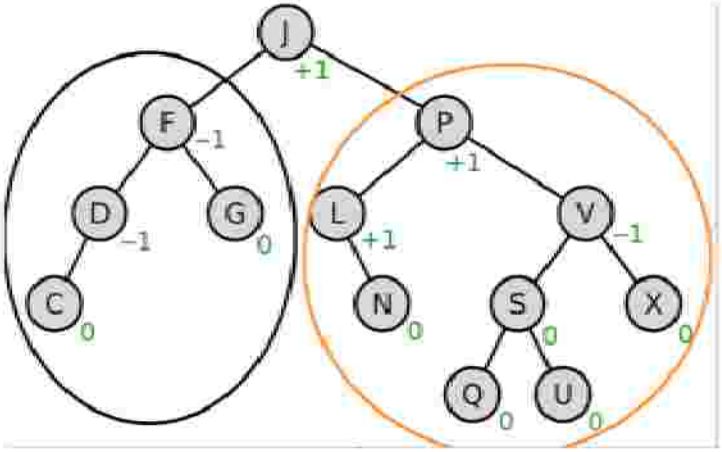
- Bad tree



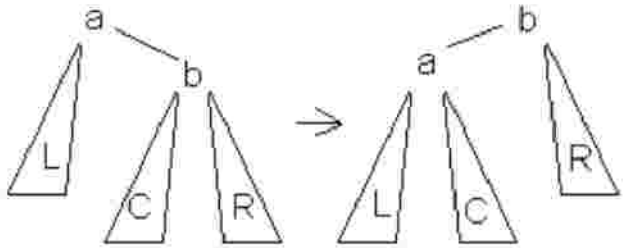
- Good tree

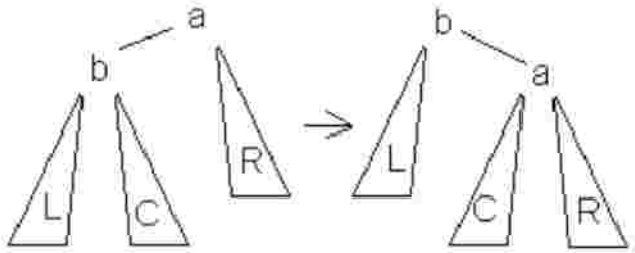
$$h \leq \lfloor 1.45 \log_2(n+2) \rfloor$$

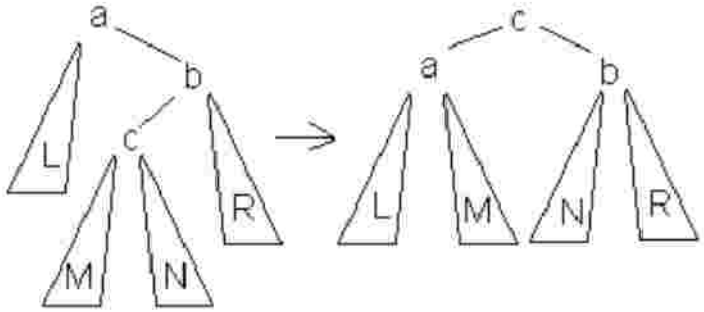


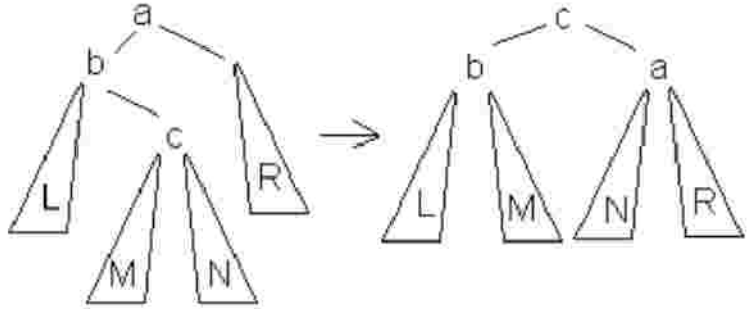


- AVL tree — only 4 operations:
  - Small left rotation
  - Small right rotation
  - Big left rotation
  - Big right rotation


a
b
L
C
R
b
a
L
C
R


a
b
L
C
R
b
a
L
C
R


a
b
c
L
M
N
R
c
a
b
L
M
N
R


a
b
c
L
M
N
R
c
b
a
L
M
N
R

```
typedef struct Node
{
    int32_t key;
    uint32_t height;
    Node* left;
    Node* right;
}Node;
```

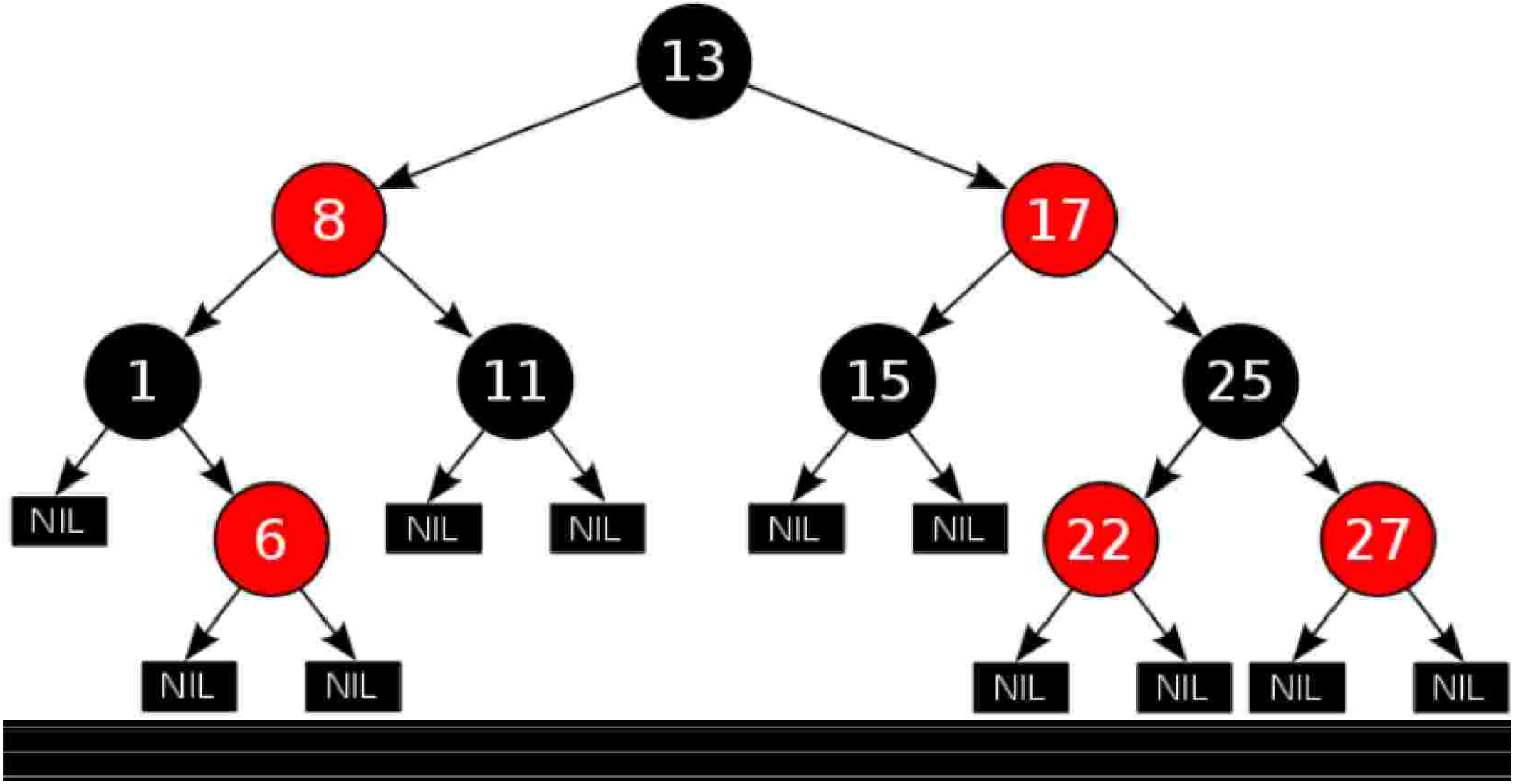
```
Node* rotate_right(Node* p) //
{
    Node* q = p->left;

    p->left = q->right;
    q->right = p;

    recount_height_of_tree(p);
    recount_height_of_tree(q);

    return q;
}
```

```
Node* rotate_left(Node* q) //
{
    Node* p = q->right;

    q->right = p->left;
    p->left = q;

    recount_height_of_tree(q);
    recount_height_of_tree(p);

    return p;
}
```

```
Node* rebalance_tree(Node* p) // балансировка узла p
{
    recount_height_of_tree(p);
    if(balance_of_the_tree(p) == 2)
    {
        if(balance_of_the_tree(p->right) < 0)
            p->right = rotate_right(p->right);
        return rotate_left(p);
    }
    if(balance_of_the_tree(p) == -2)
    {
        if(balance_of_the_tree(p->left) > 0)
            p->left = rotate_left(p->left);
        return rotate_right(p);
    }
    return p; // балансировка не нужна
}
```

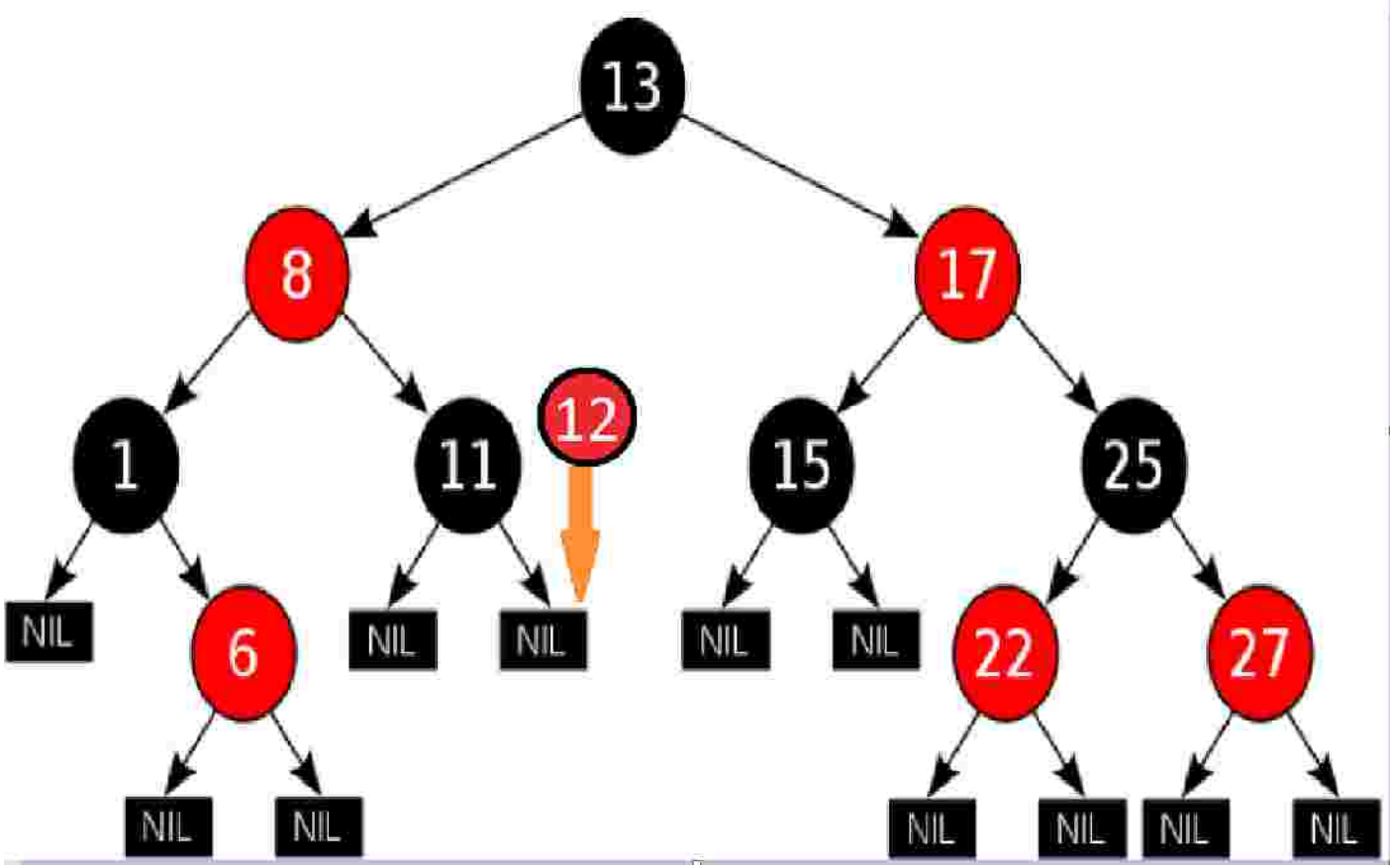
```
Node* insert_node(Node* p, int32_t k)
{
    if(!p) {
        p = malloc(sizeof(Node));
        create_node(p, k);
    }
    if(k < p->key)
        p->left = insert_node(p->left, k);
    else
        p->right = insert_node(p->right, k);
    return rebalance_tree(p);
}
```

Red-black tree


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

- Each node is either red or black.
- The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
- All leaves (NULL) are black.
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant NULL nodes goes through the same number of black nodes.

- Insert


13
8
17
1
11
12
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL
NIL
NIL

- Case 1

```
void insert_case1(struct Node *n)
{
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(n);
}
```

- Case 2

```c
void insert_case2(struct Node *n)
{
    if (n->parent->color == BLACK)
        return; /* Tree is still valid */
    else
        insert_case3(n);
}
```

? Case 3

```
void insert_case3(struct Node *n)
{
    struct node *u = uncle(n), *g;

    if ((u != NULL) && (u->color == RED)) {
        n->parent->color = BLACK;
        u->color = BLACK;
        g = grandparent(n);
        g->color = RED;
        insert_case1(g);
    }
    else
    {
        insert_case4(n);
    }
}
```

? Case 4

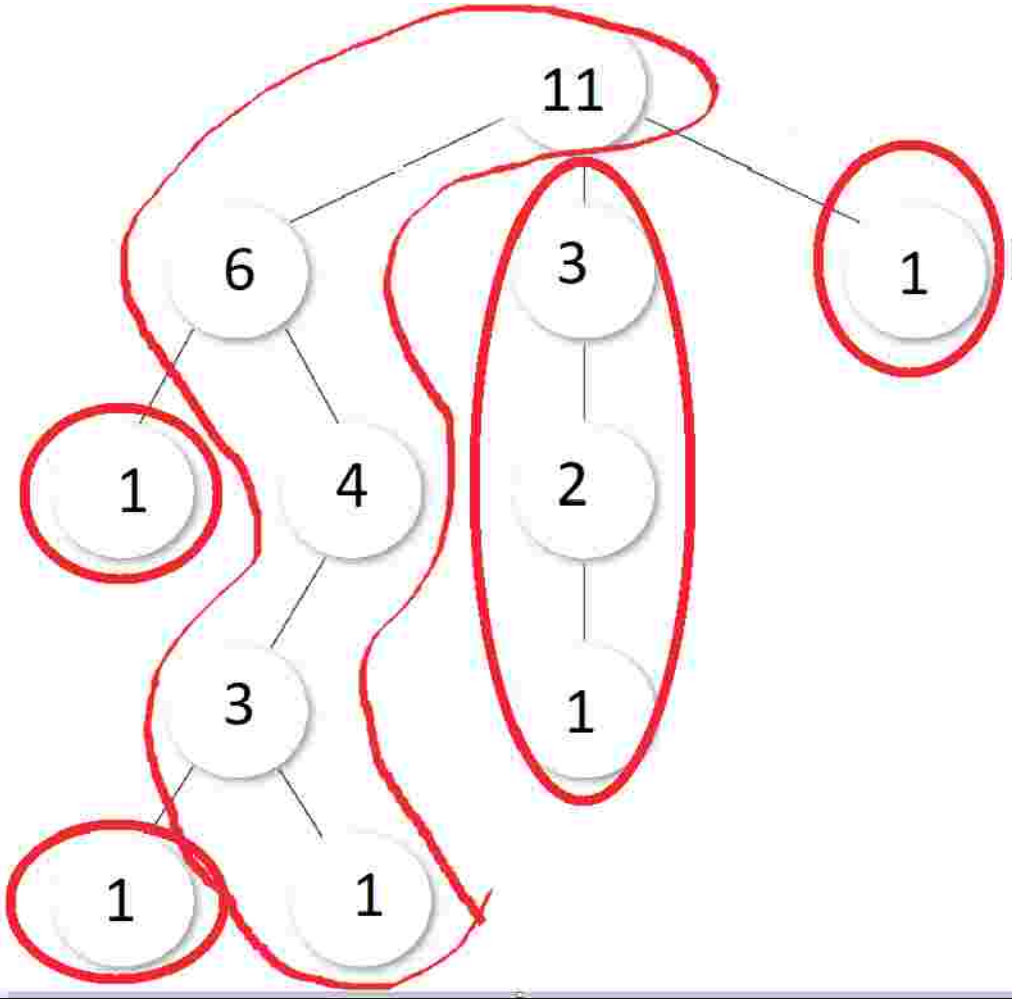
```
void insert_case4(struct Node *n)
{
    struct Node *g = grandparent(n);

    if ((n == n->parent->right) && (n->parent == g->left)) {
        rotate_left(n->parent);
        n = n->left;
    }
    else
    {
        if ((n == n->parent->left) && (n->parent == g->right)) {
            rotate_right(n->parent);


            n = n->right;
        }
    }
    insert_case5(n);
}
```

? Case 5

```
void insert_case5(struct Node *n)
{
    struct Node *g = grandparent(n);

    n->parent->color = BLACK;
    g->color = RED;

    if ((n == n->parent->left) && (n->parent == g->left))
        rotate_right(g);
    else
        rotate_left(g);

}
```

Heavy-light decomposition

$$W(v) / 2 <= W(c)$$


11
6
3
1
1
4
2
3
1
1
1