1 Red-Black-Tree

1.1 Introduction and general properties

A red-black-tree is a self-balancing binary search tree. In addition to the invariants of the latter, the former garantees that the tree remains roughly balanced even when it receives input sequences that constitute pathological cases for normal binary search trees (sorted input).

Every node in a red-black-tree is assigned one of two colors (traditionally red and black). The following invariants hold after each insertion into, or deletion from a red-black-tree.

(1) The root node of the tree is black. Every other node is either red or black.

(2) If a node is red, it doesn't have a red child.

(3) All paths from the root to a leaf go through the same number of black nodes.

If b is the number of black nodes on every path from the root to a leaf, then the shortest such path (which contains no red nodes at all) has a length of b, and the longest such path (where black and red nodes alternate) has a length of $2b$. Thus the tree guarantees that the lengths of any two paths from the root to a leaf (i.e. the depths of any two leaf nodes) differ by at most a factor of 2.

$$\forall \text{ leaf nodes } l_i, l_j \in T : \text{depth}(l_i) \leq 2 \cdot \text{depth}(l_j)$$

1.2 Groups of Nodes

Let a *group* of nodes in a red-black-tree denote either *any black node together with all its direct red child nodes,* or the *null-group* which contains no nodes at all[1]. Any node in a red-black-tree belongs to exactly one group: If we combine invariants 1 and 2 with this definition, we can infer that the nearest descendant nodes below a group must be black, and so they constitute roots of groups themselves. We can rephrase invariant 3 as follows:

(3) All paths from the root to a leaf go through the same number of groups.

The *weight* of a group g is the number of nodes it contains, obviously $0 \leq \text{weight}(g) \leq 3$. The null-group has a weight of 0. If $\text{weight}(g) = 1$, g is said to be *empty:* it doesn't contain any red child nodes. If $\text{weight}(g) = 3$, g is said to be *full:* it contains two red child nodes and it's not possible to add another node to the group. If $\text{weight}(g) = 2$, it's possible to rotate g in such a way that the former red child becomes the black root of the group, and the former black root becomes a red child on the opposite site; this rotation does not affect any other group than g.

[1]Calling nothing a group simplifies a few arguments.

In order to preserve this invariant, inserting a node into a red-black-tree must not add a new group to the tree (except at the very root where the addition affects all paths equally). Likewise, deleting a node from the tree must not remove a group entirely (again, except at the root). The fact that groups can have variable numbers of red nodes provides us with enough flexibility though: We can shift nodes between groups, rotate sub-trees, and recolor nodes to make sure we have some leeway at the position where we need to insert or delete a node.

1.3 Insertion

Insertion into a normal binary search tree is straightforward. You start at the root, and then you always go left if the new value is less than the current value, or you go right if it is greater. When you've nowhere left to go (i.e. you'd need to go left but there's no left child, or the other way around) you simply add the value at that position.

Insertion into a red-black-tree works similarly but it must not add a new group somewhere down the tree because that would violate invariant 3. Therefore insertion is not possible if the direct parent of the to-be-added node is part of a full group. The solution is to reduce the weight of that group: If $1 \leq \mathrm{weight}(g) \leq 2$, it is always possible to add another node to g.

```
procedure decrease-weight(n):
    p := parent of n
    pp := grandparent of n
    l, r := children of n

    // assume that n is black and l, r are red

    case 1: n is the root of the tree
        recolor:
            l, r: black

    case 2: p is black
        recolor:
            n: red
            l, r: black

    case 3: p is red
        case 3.1: pp has 1 red child (p)
            case 3.1a: pp->p->n is right-right or left-left
                right-right: rotate-left(pp)
                left-left: rotate-right(pp)
                recolor:
                    pp: red
                    p: black
                    n: red
                    l, r: black
```

```
            case 3.1b: pp->p->n is right-left or left-right
                right-left:
                    rotate-right(p)
                    rotate-left(pp)
                left-right:
                    rotate-left(p)
                    rotate-right(pp)
                recolor:
                    pp: red
                    l, r: black

        case 3.2: pp has 2 red children
            decrease-weight(pp)
            decrease-weight(n) // try again

procedure insert(n, v):
    p := parent of n

    if v = n.value:
        return
    else if v < n.value and n has a left child:
        insert(n.left, v)
        return
    else if v > n.value and n has a right child:
        insert(n.right, v)
        return

    case 1: n is black
        if v < n.value:
            n.left = new node(v, red)
            n.left.parent = n
        else:
            n.right = new node(v, red)
            n.right.parent = n

    case 2: n is red
        case 2.1: weight(p) = 2
            decrease-weight(p)
            // now n can be anywhere, go again:
            insert(n, v)

        case 2.2: weight(p) = 1
            case 2.2.1: v < n.value and n = p.left
                rotate-right(p)
                n.left = new node(v, red)
            case 2.2.2: v > n.value and n = p.right
                rotate-left(p)
                n.right = new node(v, red)
            case 2.2.3: v < n.value and n = p.right
                p.left = new node(v, red)
```

```
    swap(p, p.left)
case 2.2.4: v > n.value and n = p.left
    p.right = new node(v, red)
    swap(p, p.right)
```

1.4 Deletion

Deletion is only possible if the group we delete from is not empty.