Red-Black Trees

CS21B017 Chetan Moturi
CS21B033 Mallela Vidhyabhushan

April 2023

1 Introduction

A binary tree search tree of height can support basic operations like search, insert, and delete in logarithmic time, $O(\lg n)$, on average. However, the worst-case time complexity of these operations is $O(n)$. This is because the tree can degenerate into a linked list. To avoid this, we can use a self-balancing binary search tree. A self-balancing binary search tree is a binary search tree in which the height of the tree is $O(\lg n)$. The height of a tree is the maximum number of edges from the root to a leaf.
One such self-balancing tree is called the Red-Black Tree. A red-black tree is a binary search tree in which each node has an extra bit, and that bit is often interpreted as the color (red or black) of the node. By constraining how the colors are used in any particular tree and how the tree is modified, we can ensure that no path from the root to leaf is more than twice as long as any other path so that the tree is approximately balanced.
Fun Fact! The color ”red” was chosen because it was the best-looking color produced by the color laser printer available to the authors while working at Xerox PARC.

2 Some Important Terminologies and Properties of Red-Black Trees

The red-black tree is balanced by assigning a color to each node and ensuring that no path from the root to the leaf is twice as long as the other.
This ensures the tree is approximately balanced. The following code snippet shows the node structure of an RB Tree.

```
enum Color {RED, BLACK, DOUBLE_BLACK};

struct Node
{
    int data;
    int color;
    Node *left, *right, *parent;

    explicit Node(int);
};
```

1. **Color**: A red-black tree consists of one extra bit of storage compared to regular Binary search trees. A node can only be RED or BLACK.
2. **Root Property**: The root is always BLACK.
3. Every node is either BLACK or RED.
4. **Sentinel Nodes**: If a child or parent of a node doesn't exist, we will assign them NIL. These are the leaves of the RB Tree, and other key-bearing nodes are called internal nodes.
5. Every leaf is BLACK.
6. If a node is RED, then both its children are BLACK
7. **Black Height**: The number of black nodes on Every simple path from, but not including, node x to a leaf. It is denoted by bh(x).
8. All nodes have an equal black height, i.e., an equal number of black nodes from the node to descendant leaves.
9. The black height of a Red-Black Tree is the same as bh(root).
10. The height of the red-black tree is at most $2 \cdot \lg(N + 1)$
11. A Red-Black Tree of height h and number of nodes N, has $bh > \frac{N}{2}$.

3 The RBTree class

The Rb tree class with the required functions is given below,

```
    class RBTree
{
    private:
        Node *root;
    protected:
        void rotateLeft(Node *&);
        void rotateRight(Node *&);
        void fixInsertRBTree(Node *&);
        void fixDeleteRBTree(Node *&);
        int getColor(Node *&);
        void setColor(Node *&, int);
        Node *minValueNode(Node *&);
        Node *maxValueNode(Node *&);
        Node* insertBST(Node *&, Node *&);
        Node* deleteBST(Node *&, int);
        int getBlackHeight(Node *);
    public:
        RBTree();
        void insertValue(int);
        void deleteValue(int);
        void merge(RBTree);
};
```

We will discuss these functions in detail now.

4 Search in RB Tree

Search in RB tree is as same as the search in a regular Binary search tree, but the search is a lot faster as the tree height is of $O(\lg n)$.
The search algorithm Given a pointer to the root of the tree and a key k, Search returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

Algorithm 1 Red-Black Tree Search

```
1: procedure RBTreeSearch($T, x$)
2:     if $T$ = null or $x = T$.key then
3:         return $T$
4:     else if $x < T$.key then
5:         return RBTreeSearch($T$.left, $x$)
6:     else
7:         return RBTreeSearch($T$.right, $x$)
8:     end if
9: end procedure
```

5 Insertion

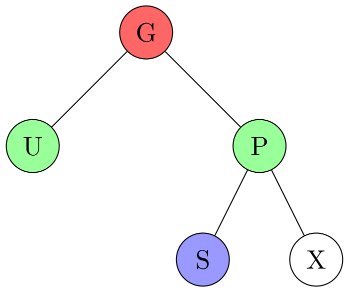
Inserting a node involves first searching through the tree to find its correct spot. Then the node is inserted there as a red node. Since it does not change the number of black nodes from any node to any of it's leaves i.e the black height, the only issue we need to deal with is the violation of no red-red adjacent property.
When inserting a new node into a Red-Black Tree, there are two possible cases that can arise depending on the color of the node's parent and uncle (the sibling of the node's parent).

1. **CASE 1**: The uncle is RED.
2. **CASE 2**: The uncle is BLACK.
    (a) **CASE 2.1**: Left-Left case.
    (b) **CASE 2.2**: Left-Right case.
    (c) **CASE 2.3**: Right-Left case.
    (d) **CASE 2.4**: Right-Right case.
3. Trivial cases: Parent is black.

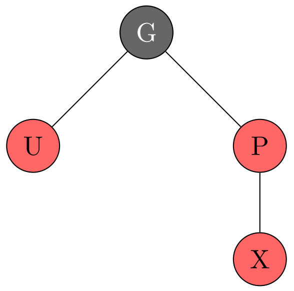
Let's discuss the cases in detail below,

5.1 CASE 1: Uncle is Red

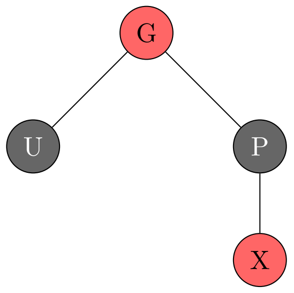
In this case, we can simply recolor the parent, uncle, and grandparent nodes as black and red, respectively. We then repeat the same process for the grandparent node, which may violate the Red-Black Tree properties further up the tree. If the grandparent is the root, we simply recolor it as black to satisfy the root's black color property.
In the below tree, G is grandparent, U is uncle, P is parent and S is sibling of X.



Initially the tree is as follows, with X being the node that is inserted,



We know that the inserted node's grandparent will be black, so all we need to do is switch the coloring of the inserted node's grandparent with the coloring of the inserted node's parent and its parent's sibling. This case might need to continue to be fixed up through the root of the tree, though, because the inserted node's grandparent may have a parent who is red.
Hence the final tree is,

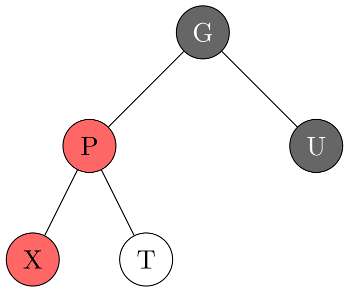


5.2 CASE 2: Uncle is Black

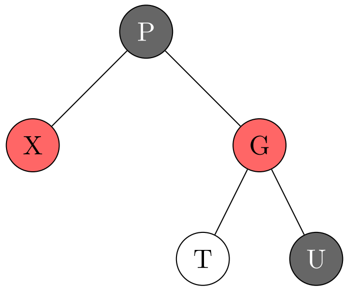
As discussed before there are four sub-cases in this scenario. Let's discuss their respective solutions.

5.2.1 Left-Left Case

If the node's parent is the left child of the grandparent and the inserted node is the left child of the parent.
Solution : Right rotate Grandfather(parent's parent) and swap the colors of grandparent and parent of the node.
Let us look at an example, once again we are inserting node X,

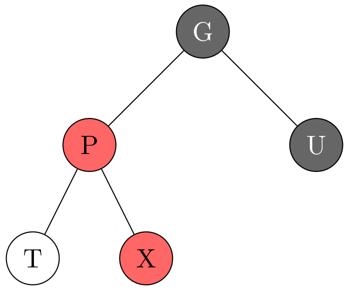


After Right rotation of Grandfather, swap colors of grandfather and parent. The final tree will look like,

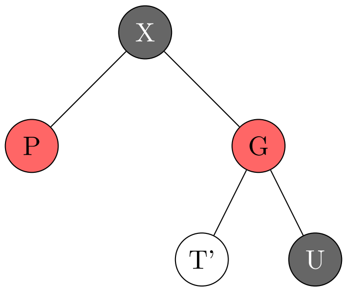


5.2.2 Left-Right Case

the node's parent is the left child of the grandparent and the inserted node is the right child of the parent.
Solution: Left rotate the parent and then apply Left-left rotate on the parent. Swap the colors of the inserted node and the grandparent. Let's look at an example,

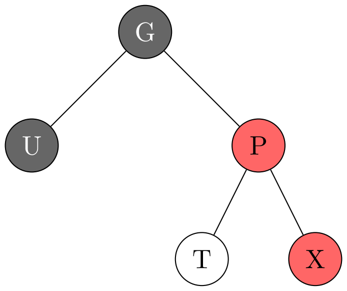


After Left rotate parent, then apply case 5.2.1, we have the final tree,

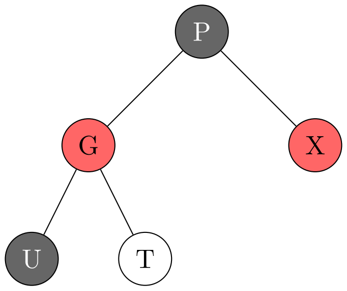


5.2.3 Right-Right Case

The node's parent is the right child of the grandparent and the inserted node is the right child of the parent.
Solution: Left rotate Grandfather(parent's parent) and swap the colors of grandparent and parent of the node. Let's look at an example,

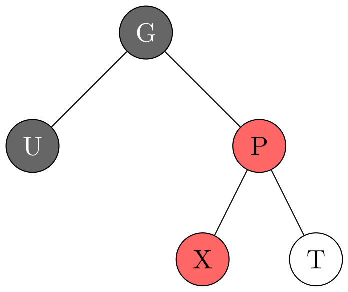


After fixing up the tree by the above solution we have the final tree as,

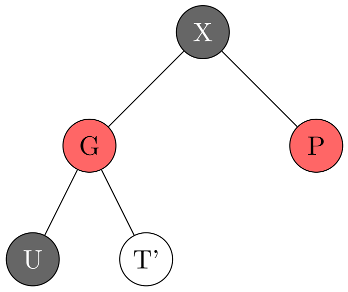


5.2.4 Right-Left Case

the node's parent is the right child of the grandparent and the inserted node is the left child of the parent. **Solution**: Right rotate the parent of the inserted node and then apply the right-right rotate on the parent node. Swap the colors of the inserted node and the grandparent. Let's look at an example,



After fixing the tree up we have the resulting tree,



5.3 Pseudo code for Insertion and Insertion Fixup

Algorithm 2 RB-Tree Insertion

```
procedure RB-INSERT($T, z$)
    $y \leftarrow$ NIL
    $x \leftarrow T.root$
    while $x \neq$ NIL do
        $y \leftarrow x$
        if $z.key < x.key$ then
            $x \leftarrow x.left$
        else
            $x \leftarrow x.right$
        end if
    end while
    $z.p \leftarrow y$
    if $y =$ NIL then
        $T.root \leftarrow z$
    else if $z.key < y.key$ then
        $y.left \leftarrow z$
    else
        $y.right \leftarrow z$
    end if
    $z.left \leftarrow$ NIL
    $z.right \leftarrow$ NIL
    $z.color \leftarrow$ RED
    RB-INSERT-FIXUP($T, z$)
end procedure
```

Algorithm 3 RB-Tree Insertion Fixup

```
procedure RB-INSERT-FIXUP($T, z$)
    while $z.p.color$ = RED do
        if $z.p = z.p.p.left$ then
            $y \leftarrow z.p.p.right$
            if $y.color$ = RED then
                $z.p.color \leftarrow$ BLACK
                $y.color \leftarrow$ BLACK
                $z.p.p.color \leftarrow$ RED
                $z \leftarrow z.p.p$
            else
                if $z = z.p.right$ then
                    $z \leftarrow z.p$
                    LEFT-ROTATE($T, z$)
                end if
                $z.p.color \leftarrow$ BLACK
                $z.p.p.color \leftarrow$ RED
                RIGHT-ROTATE($T, z.p.p$)
            end if
        else
            $y \leftarrow z.p.p.left$
            if $y.color$ = RED then
                $z.p.color \leftarrow$ BLACK
                $y.color \leftarrow$ BLACK
                $z.p.p.color \leftarrow$ RED
                $z \leftarrow z.p.p$
            else
                if $z = z.p.left$ then
                    $z \leftarrow z.p$
                    RIGHT-ROTATE($T, z$)
                end if
                $z.p.color \leftarrow$ BLACK
                $z.p.p.color \leftarrow$ RED
                LEFT-ROTATE($T, z.p.p$)
            end if
        end if
    end while
    $T.root.color \leftarrow$ BLACK
end procedure
```

6 Deletion

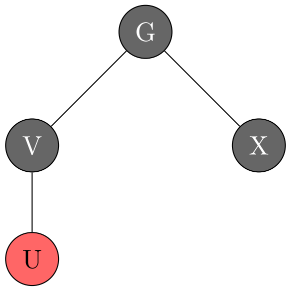
Like other basic operations on the RB Tree deletion takes time $O(\lg n)$. Deleting a node from a RB Tree is more complicated than insertion. When a node is deleted it can either have no children, one child or two children.In delete, the main violated property is, change of black height in sub trees as deletion of a black node may cause reduced black height in one root to leaf path.
To understand deletion, the notion of **double black** is used. When a black node is deleted and replaced by a black child, the child is marked as **double black**. The main task now becomes to convert this **double black** to single black.
To delete a node from a Red Black Tree follow the steps as given below,

1. **Perform standard BST delete**: When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).
2. **Simple Case: If either u or v is red**
3. **Complex Case: If Both u and v are Black**.
   (a) colour u as double black.
   (b) **NOTE!** If v is a leaf, then u is NULL, and the colour of NULL is considered black.
   (c) For a sibling node s of current node u that is double black but not the root,
      i. **case 1**: If sibling node s is black and at least one of its children is red. This will again contain the sub cases of RR, RL, LR and LL.
      ii. **case 2**: If the sibling s is black and both of its children are black.
      iii. **case 3**: If sibling s is red, perform a rotation to move the old sibling node up,recolor the old sibling and parent. The new sibling is always black. This converts the tree to the black sibling case (by rotation) and leads to previous cases. This case is divided into two sub cases; left and right cases.
4. If u is the root, make it single black.

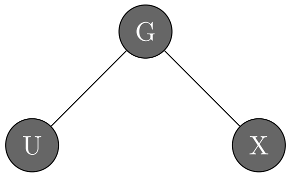
Let's discuss the cases possible in detail along with their respective examples,

6.1 Simple Case: Either u or v is red

Here as mentioned before, v is the node to be deleted and u is the child which replaced v. We mark the replaced child as black (deletion causes no change in black height). Note that as v is the parent of u, and two consecutive reds are not allowed in the red-black tree, both u and v cannot be red. Let's look at the example below,

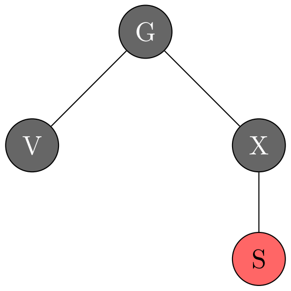


Since we are deleting V, we replace it with U and make U black to preserve the black height. So the final tree looks like,

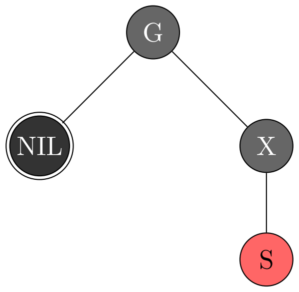


6.2 Complex Case: Both u and v are black

color the node u as double black. Now, the task is to convert the double black into single black. If v is leaf then u is NIL so it is a black node. Hence deletion of black leaf causes double black.



After deleting V we get the tree,



In the above tree we see u becomes a double black NIL node.
Now we'll see the cases for a sibling node s for a double black node u which is not the root,

6.2.1 If sibling node s is black and at least one of its children is red

We need to perform rotations to re-balance the tree, as such this consists of four different sub cases depending on the relative position of s and r, the red child of s.

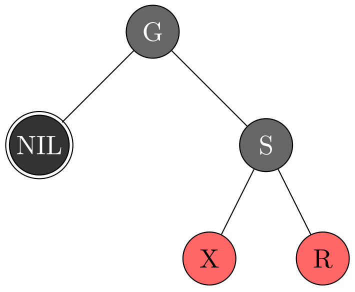
Right-Right case: In this case, S is right child of it's parent and R is right child of S.
Conditions:

1. Double Black's sibling is black.
2. Double Black's sibling's far child is red

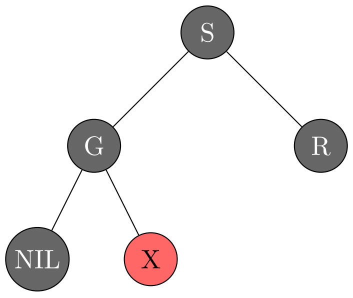
Solution:

1. Swap color of DB's parent with DB's sibling's color.
2. Perform rotation at DB's parent in direction of DB.
3. Remove DB sign and make the node normal black node.
4. Change colour of DB's sibling's far red child to black.

Let us look at an example for better understanding, The tree at its double black stage is given below,



After performing the above steps we get the tree,



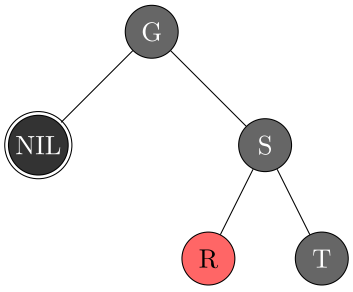
Right-Left Case: In this case S is the right child of its parent and R is the red left child of S.
Conditions:

1. Double Black's sibling is black.
2. Double Black's sibling's far child is black.
3. DB's sibling's child which is near is red.

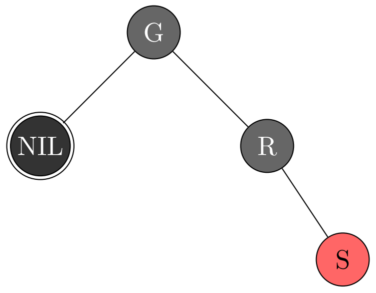
Solution:

1. Swap color of sibling with sibling's red child.
2. Perform rotation at sibling node in direction opposite of DB.
3. apply the previous case.

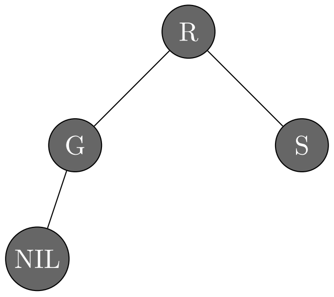
Let us look at the example below, V has been deleted and U is double black, T is any sub-tree/node.



After performing the first two steps we get the tree,



Here, T is a sentinel node (or any sub-tree).
Now, we can apply the right-right case and re balance the tree.



Left-Left Case: where s is left child of its parent and r is left child of s; it is mirror case of the right-right case.

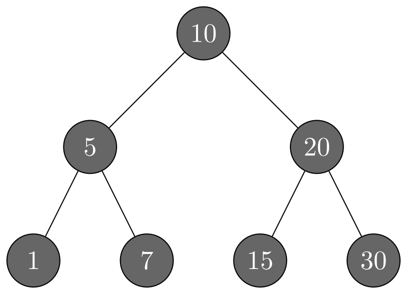
Left-Right Case: where s is left child of its parent and r is right child of s; it is mirror case of the left-right case.

6.2.2 If the sibling S is black and both of its children are black

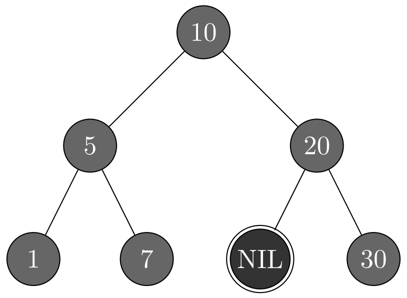
The solution to the above case is as follows,

1. Remove the DB(if NIL-DB remove the node, for other nodes, remove the DB sign.
2. Make DB's sibling red.
3. If DB's parent is black make it DB, else make it black.

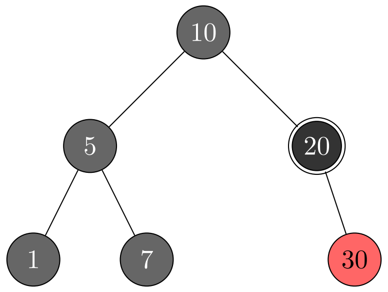
This will become clear with the following example, the initial tree is given below and we will remove,



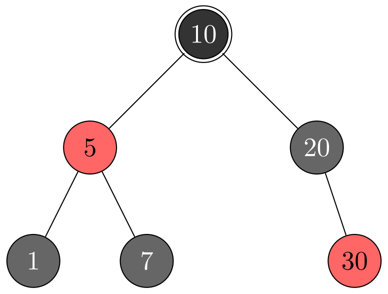
After deleting 15 we get,



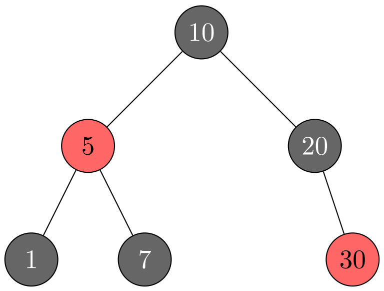
After performing steps 2 & 3 we get,



Repeat this again for the double black-20, we get,



Since DB is the root we can demote it to black, giving us the final tree as,

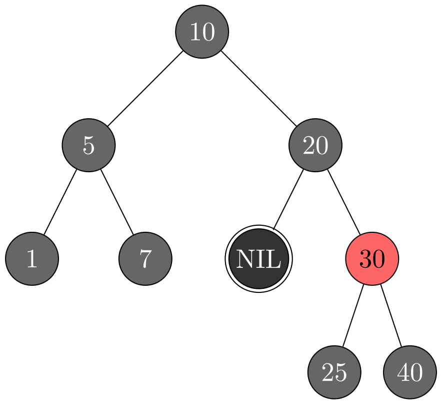


6.2.3 If sibling S is red

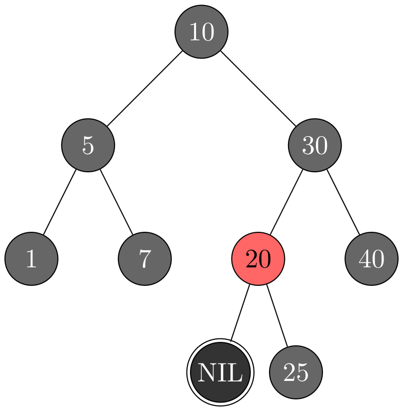
The solution to the above case is as follows,

1. swap DB'S parent's and S's colors.
2. perform rotation at parent node in direction of DB.
3. check which case can be applied to the new tree and re-balance the tree.

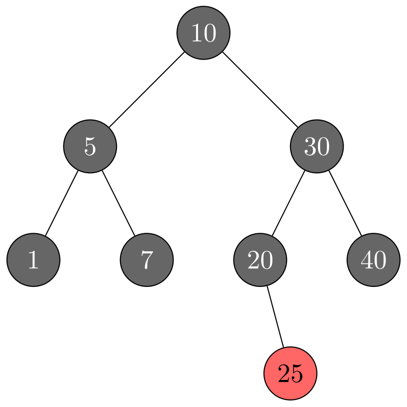
Let's see the example below,



After fixing the tree by the above rules we get,



We see that this is the case in which sibling and both children are black, applying the rules we get,



6.3 Trivial Cases

6.3.1 DB is the root:

Simply convert it into a single black node.

6.3.2 If node to be deleted is red:

Remove the node as it doesn't change the black height.

7 Pseudo code for Deletion and Deletion Fixup

Algorithm 4 RB-Tree Deletion

```
procedure RB-DELETE($T, z$)
    $y \leftarrow z$
    $y - original - color \leftarrow y.color$
    if $z.left = nil$ then
        $x \leftarrow z.right$
        RB-TRANSPLANT($T, z, z.right$)
    else if $z.right = nil$ then
        $x \leftarrow z.left$
        RB-TRANSPLANT($T, z, z.left$)
    else
        $y \leftarrow$ TREE-MINIMUM($z.right$)
        $y - original - color \leftarrow y.color$
        $x \leftarrow y.right$
        if $y.parent = z$ then
            $x.parent \leftarrow y$
        else
            RB-TRANSPLANT($T, y, y.right$)
            $y.right \leftarrow z.right$
            $y.right.parent \leftarrow y$
        end if
        RB-TRANSPLANT($T, z, y$)
        $y.left \leftarrow z.left$
        $y.left.parent \leftarrow y$
        $y.color \leftarrow z.color$
    end if
    if $y - original - color = Black$ then
        RB-DELETE-FIXUP($T, x$)
    end if
end procedure
```

Algorithm 5 RB-Tree-Deletion-FixUp

```
procedure RB-DELETE-FIXUP($T, z$)
    while $x \neq T.root$ and $x.color = Black$ do
        if $x = x.parent.left$ then
            $w \leftarrow x.parent.right$
            if $w.color = Red$ then
                $w.color \leftarrow Black$
                $x.parent.color \leftarrow Red$
                Left-Rotate($T, x.parent$)
                $w \leftarrow x.parent.right$
            end if
            if $w.left.color = Black$ and $w.right.color = Black$ then
                $w.color \leftarrow Red$
                $x \leftarrow x.parent$
            else
                if $w.right.color = Black$ then
                    $w.left.color \leftarrow Black$
                    $w.color \leftarrow Red$
                    Right-Rotate($T, w$)
                    $w \leftarrow x.parent.right$
                end if
                $w.color \leftarrow x.parent.color$
                $x.parent.color \leftarrow Black$
                $w.right.color \leftarrow Black$
                Left-Rotate($T, x.parent$)
                $x \leftarrow T.root$
            end if
        else
            $w \leftarrow x.parent.left$
            if $w.color = Red$ then
                $w.color \leftarrow Black$
                $x.parent.color \leftarrow Red$
                Right-Rotate($T, x.parent$)
                $w \leftarrow x.parent.left$
            end if
            if $w.right.color = Black$ and $w.left.color = Black$ then
                $w.color \leftarrow Red$
                $x \leftarrow x.parent$
            else
                if $w.left.color = Black$ then
                    $w.right.color \leftarrow Black$
                    $w.color \leftarrow Red$
                    Right-Rotate($T, w$)
                    $w \leftarrow x.parent.left$
```

Algorithm 6 Contd.

```
                end if
                $w.color \leftarrow x.parent.color$
                $x.parent.color \leftarrow Black$
                $w.left.color \leftarrow Black$
                RIGHT-ROTATE($T, x.parent$)
                $x \leftarrow T.root$
            end if
        end if
    end while
    $x.color \leftarrow Black$
end procedure
```

8 RB Tree height proof

Proof. We will prove by induction on the number of nodes n in the tree that the maximum height of a red-black tree with n nodes is at most $2\log_2(n+1)$.

Base Case: When $n = 1$, the tree consists of a single black root node, and its maximum height is 0. The inequality $0 \leq 2\log_2(1+1)$ holds, so the base case is true.

Inductive Hypothesis: Assume that for any red-black tree with k nodes, where $1 \leq k \leq n$, the maximum height is at most $2\log_2(k+1)$.

Inductive Step: Consider a red-black tree with $n+1$ nodes. Let T denote the tree's root and T_l and T_r denote its left and right subtrees, respectively. Let h_l and h_r denote the maximum heights of T_l and T_r, respectively.

Since T is black and both of its children are red, the maximum height of T is $1 + \max h_l, h_r$. By the inductive hypothesis, we have $h_l \leq 2\log_2(k_l+1)$ and $h_r \leq 2\log_2(k_r+1)$, where k_l and k_r denote the number of nodes in T_l and T_r, respectively. Therefore,

$$\begin{aligned}
\text{max\_height}(T) &= 1 + \max(h_l, h_r) \\
&\leq 1 + \max(2\log_2(k_l+1), 2\log_2(k_r+1)) \\
&= 1 + 2\lg[\max((k_l+1), (k_r+1))] \\
&\leq 1 + 2\log_2(k_l + k_r + 2) \\
&\leq 2\log_2(n+2) \\
&= 2\log_2[(n+1)+1]
\end{aligned}$$

Thus, the maximum height of a red-black tree with $n+1$ nodes is at most $2\log_2(n+2)$. By induction, the statement is true for all $n \geq 1$. □