Red-Black Trees

Red-Black Trees (RBT) are Binary Search Trees (BST) in which every node contains one additional information: its color, which can either be red or black. Compared to BSTs, RBTs guarantee that basic dynamic-set operations take at most $O(log_2\ n)$. In a RBT, keys are stored in internal nodes, and leaves as well as the parent of the root are replaced by a sentinel node **NIL**. RBTs are BSTs that satisfy the following Red-Black properties:

1. Each node is either *red* or *black*.
2. The root is *black*.
3. All leaves (NIL) are *black*.
4. If a node is *red*, then both its children are *black*.
5. All simple paths from a node to a leaf contain the same number of *black* nodes.

The number of nodes on any simple path from a node x (excluded) down to a leaf is called **black-height** of x, or **b-height**, denoted by $bh(x)$.
The black-height of the tree is the black-height of the root.

LEMMA: A RBT with n internal nodes has a height at most $2\ log_2(n+1)$

Proof:
We first prove by induction that any subtree with root x has at least $2^{bh(x)} - 1$ internal nodes. If the height is 0, then x is a leaf, therefore it has $2^0 - 1 = 0$ internal nodes.
Any internal node x has 2 children, both of which have a black-height of either $bh(x)$ or $bh(x) - 1$ depending if its red or black, respectively.
Therefore, each child has at least $2^{bh(x)-1} - 1$ internal nodes, thus the subtree rooted at x as at least:
$((2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1)$ *(both children + the root itself)* =

$2^1\ 2^{bh(x)-1} - 2 + 1 = 2^{bh(x)-1} - 1.$

According to property *(4)*, at least half of the nodes from the root down to a leaf

must be black, therefore, the b-height of the root must be at least $2^{\frac{h}{2}} - 1$:
$n \geq 2^{\frac{h}{2}} - 1$

$n + 1 \geq 2^{\frac{h}{2}}$

$log_2(n + 1) \geq \frac{h}{2}$

$h \leq 2log_2(n + 1)$

Insertion and Deletion in a RBT may lead to violations of the Red-Black properties.
In order to restore such properties, we need to

1. Change the pointer structure of the tree
2. Change the color of some nodes

We change the pointer structure through *rotations*, which may either be *left rotations* or *right rotations*.
In the left rotation of a Node x, we assume its right child y to be $\neq NIL$, x then becomes y's left child, and y's left child becomes x's right child.
The *right rotation* is completely symmetric.
Rotations run in $O(1)$ time, because only pointers are changed by them while other attributes in a node remain the same.

Insertions

We can insert a node in a RBT in $O(\log(n))$ time. To do so, we first insert the new node Z in the RBT as if it were a normal BST, we then color Z red. Since this insertion may violate the *Red-Black properties* we call an auxiliary procedure *insertFixup* to recolor nodes and performs rotations.
There are four cases to consider: since we added a *red* node, the only properties that the insertion may violate are *(2)* and *(4)*.

1. If the tree violates property *(2)*, then Z is the root, which is the only internal node of the tree. Since its parent and children are sentinels, property *(4)* cannot be violated, therefore, we simply recolor Z as black.
2. If the Z's uncle is red, then the tree violates property *(4)*, therefore, we recolor Z's uncle, parent and grandparent.

3. If Z's uncle is black, and either Z's parent is a right child and Z is a left child OR Z's parent is a left child and Z is a right child, we rotate Z's parent in the opposite direction of Z.

4. If Z's uncle is black, and both Z's parent and Z are right children OR both Z's parent and Z are left children, we rotate Z's grandparent in the opposite direction of Z, we then recolor the original parent and grandparent of Z.

Insertion in a RBT has a time complexity of $O(\log(n))$.
The basic inserting operation takes $O(\log(n))$, we then perform $\log(n)$ recolors and rotations, which take $O(1)$ time.

Deletions

Just like insertions, deletions in a RBT may lead to a violation of the Red-Black properties.
To delete a node Z, we first delete it like we would in a normal BST, with a slightly different *transplant* procedure and by keeping track of a node Y that may violate the properties, then call an additional procedure *deleteFixup* which recolors nodes and performs rotations.
If Z's original color was red, then red-black properties still hold:

1. No black-heights have changed.

2. No red nodes have been made adjacent, because Y takes Z's place and its color.

3. Since Y could not have been the root, the root remains black.

If Z's original color was black, then removing or moving Y may violate the Red-Black properties, as any path containing Y may have a *"missing black"*.
Therefore, we add an *"extra black"* to X and try to move it up the tree until either X is both red and black - at which point we simply color it black - or X is the root, at which point we can remove the extra black.
We have to handle 4 cases (and their symmetrical variants), depending on the node X and its sibling W:

1. If W is red, both of W's children are black, we swap the colors of W and and X's parent, and perform a rotation on X's parent on the same side of X.
   X's sibling W is now black, therefore, we have moved to one of the other 3 cases.

2. If W is black and both of its children are black, we change W's color to red and repeat the loop on X's parent.

3. If W is black and its right child is black (and W's left child is red), we recolor W and W's left child, and then perform a right rotation on W. X's new sibling is black with a red child, therefore, we are now in the *4th* case.

4. If W is black and its right child is red, we transfer to W the color of its parent and then change the color of W's parent and W's right child to black.
   We then perform a left rotation on X's parent and set X as the root (causing the loop to terminate).

The time complexity of the delete operation is $O(\log(n))$ without the *deleteFixup* procedure, because the height of a tree of n nodes is at most $\log(n)$.
Within the *deleteFixup* procedure, *case 1, 3 and 4* only perform a constant number of color changes and rotations.
Case 2 is the only case in which the while loop can be repeated, and then the pointer X moves up the three at most $\log(n)$ times.
Therefore, the overall time complexity of the delete operation is $O(\log(n))$.