Analysis of Algorithms I: Red-Black Trees

Xi Chen

Columbia University

Goal: Maintain a dynamic subset S of a universe U and handle the following operations: Insert, Delete, Search, Min, Max, Successor, Predecessor (and more) in worst-case $O(\lg n)$ time, where $n = |S|$.

1. AVL trees
2. 2-3 trees
3. 2-3-4 trees
4. B-trees

We discuss red-black trees in this class. A red-black tree is a very "balanced" binary search tree. Difficulty: insertion and deletion.

In a usual binary search tree, an internal node may have either 1 or 2 child nodes and every node has a key, including the leaves. Given such a BST, we transform it as follows for convenience when describing red-black trees. If a child of a node u does not exist, (instead of setting the corresponding pointer of u to be nil) we create a new leaf with key $=$ nil and set the corresponding pointer of u to be this nil leaf. After this, every internal node of the tree has both left and right child nodes (though none, or one, or both of them might be nil); every internal node has a key from the universe; and every leaf has its key $=$ nil. Check page 310 for an example (ignore the colors for now).

From now on, by a BST we always mean a BST after this transformation. Use induction to prove the following:

Lemma

A binary search tree with n internal nodes has $n + 1$ nil leaves.

Thus, a binary search tree with n internal nodes (or n keys) has $2n + 1$ nodes in total.

A red-black tree is a binary search tree (and thus, satisfies the BST property in the first place!) in which each node v has one extra bit of info: its color $v.\text{color} \in \{\,\text{Red}, \text{Black}\,\}$, such that the following three properties are satisfied:

1. The root of the tree is black and every nil leaf is black; (so all paths from the root to a leaf start and end with a black node.)
2. If a node is red then both its children are black;
3. For every node v, all paths from v to its leaves have the same number of black nodes. Given a node v, we refer to the number of black nodes from (but not including) v to a leaf the black-height of v.

An example of a red-black tree can be found on page 310. Some discussions: First, Property 2 also implies that if a node is red then its parent must be black (why?). As a result, in any path from the root to a leaf, there cannot be two consecutive red nodes. For every red node in such a path, both its successor and predecessor must be black. (Or equivalently, no two red nodes are directly connected in the tree.) So for any path P from the root to a leaf:

$$\#\text{ of red nodes in } P \leq \#\text{ of black nodes in } P$$

(where $\#$ stands for "the number") and thus,

$$2 \cdot \#\text{ of black nodes in } P \geq \#\text{ of nodes in } P$$

For Property 3, it is equivalent to the following weaker statement:

Lemma

A red-black tree satisfies Property 3 if and only if all paths from the root to a leaf have the same number of black nodes.

So whenever we need to check that a given tree satisfies Property 3, it suffices to show that all paths from the root to a leaf have the same number of black nodes. But whenever we need to use Property 3, we can use its original and stronger form (for any node v, all paths from v to a leaf have the same number of black nodes). Also think about the shape of a red-black tree when it has no red node at all to get some intuition as why it is "balanced".

The first thing we show about red-black trees is that they are very "balanced" and have $O(\lg n)$ depth when it has n internal nodes (and thus, contains n keys). Because a red-black tree is a binary search tree, all the query operations described for binary search trees apply, e.g., Search, Min, Max, Predecessor and Successor. All these operations have worst-case $O(\lg n)$ running time because red-black trees always have $O(\lg n)$ depth. (What about Insert and Delete? Do they still work?)

We give two proofs for the following theorem:

Theorem

A red-black tree with n internal nodes has $O(\lg n)$ depth.

By the lemma on slide 4, the total number of nodes is $2n + 1$. Let bh denote the black-height of the root. By Property 3, any path from the root to a leaf has bh many black nodes (excluding the root itself) and thus, has length $\geq$ bh. As a result, if we only consider levels $0, 1, \ldots,$ bh of the tree, it must be a complete binary tree of depth bh (why?). This implies that

$$2n + 1 \geq 1 + 2^1 + \cdots + 2^{\text{bh}} = 2^{\text{bh}+1} - 1$$

and thus, $\text{bh} = O(\lg n)$. Moreover, by Property 2 we have

$$\text{depth of the tree} = \text{length of the longest path} \leq 2 \cdot \text{bh}$$

and thus, the depth of a red-black tree is $O(\text{bh}) = O(\lg n)$.

An alternative proof: Given a red-black tree T, we keep all the black nodes (including the nil leaves) but merge every red node into its parent. Denote the new tree by T'. Clearly a leaf of T remains to be a leaf in T'. The new tree T' has the following properties: Every internal node of T' has 2, 3 or 4 children (also called a 2-3-4 tree); and all paths from the root of T' to a leaf have the same length. It is equal to the black-height of the root in T which we denote by bh. (Prove both properties.) They imply

$$n + 1 = \text{number of leaves in } T = \text{number of leaves in } T' \geq 2^{\text{bh}}$$

and thus, $\text{bh} = O(\lg n)$. The rest of the proof is the same.

So we know that any red-black tree has low depth. The most difficult part is then how to properly insert or delete an internal node, while maintaining all the red-black properties (and the BST property as well, of course) so that the new tree (with one more or one less node) remains a red-black tree. Clearly we cannot just use the insert and delete operations for general BSTs (which we will refer to as BST-Insert and BST-Delete) since in general they result in a BST that violates the red-black properties. We discuss RB-Insert in details below, while details of RB-Delete can be found in the textbook.

High-level idea of RB-Insert:

1. Call BST-Insert to insert the new node first; denote the new node by z and color it red. (Note that z has two nil children.)
2. If the parent $z.p$ of z turns out to be black (lucky!), we are done (show that this new tree is indeed a red-black tree).
3. If the parent $z.p$ is red, we are somewhat close (show that this new tree satisfies all red-black properties, except one violation: both z and $z.p$ are red).
4. fix this violation using recoloring and "rotation".

We start by describing the the "rotation" operation.

Rotation is a basic operation used in RB-Insert. It only uses $O(1)$ running time and reconstructs the tree locally. See Figure 13.2 on page 313 for the definition of Left-Rotate and Right-Rotate. The key property here is that rotation preserves the BST property (ignore the colors for now). To see this, consider the subtree on the left of Figure 13.2. It becomes the subtree on the right after a left-rotation at x. If the original tree satisfies the BST property:

$$a.k \leq x.k \leq b.k \leq y.k \leq c.k, \quad \forall\, a \in \alpha,\ b \in \beta \text{ and } c \in \gamma$$

this implies that the new tree also satisfies the BST property.

The best example to demonstrate the operation of RB-Insert can be found in Figure 13.4 on page 317. In the next few slides, we follow the operation of RB-Insert on this tree closely and try to summarize the three cases to be discussed formally later.

First, we call BST-Insert to insert the new node with key $= 4$, denoted by z, and color it red. This gives us Figure 13.4(a) in which z has two black nil children (all nil leaves are not drawn in the picture). This tree clearly satisfies the BST property because it is the result of BST-Insert. Check each of the red-black properties. The only violation at this moment is that the parent $z.p$ (with key 5) of z is red as well (if $z.p$ happens to be black then we are already done). This violates Property 2, but all other red-black properties hold, e.g., all paths from the root to a leaf still have 3 black nodes (including the black nils).

To deal with this violation, first note that the grandparent $z.p.p$ of z (the node with key $= 7$) must be black (why? because $z.p$ is red and because it is a red-black tree before inserting z). We also note that the uncle node of z, y in Figure 13.4(a) with key 8, is red as well. (The uncle of z being red, however, is not necessarily the case. If the uncle of z is black we will apply a different operation.) So if we recolor 5 and 8 to be black and 7 to be red, we resolve the violation between z and $z.p$. Actually all red-black properties get preserved (in particular, check Property 3), except that a new violation of Property 2 occurs: $z.p.p$ and $z.p.p.p$ are both red (if $z.p.p.p$ is black we are done). At the end, we set z to be $z.p.p$.

The operation described in the last slide (recoloring only) will be referred to as Case 1 in RB-Insert. To summarize, if

1. the only violation comes from z and $z.p$ being both red; and
2. z's uncle is red as well,

we can recolor z's parent, uncle and grandparent (push the black from z's grandparent down to its parent and uncle) to resolve this violation. A new violation may occur, however. But it appears at a higher level if $z.p.p$ and $z.p.p.p$ are both red.

Next in Figure 13.4(b), z is now the node with key $= 7$. We cannot use the same method to push up the violation because the uncle of z, y in the picture with key $= 14$, is black. Also notice that the path from z's grandparent to z is a zigzag (one left edge and one right edge). In this case, we perform a left rotation at $z.p$ to reconstruct the subtree locally. After the rotation, $z.p$ becomes the left child of z and we set it to be z (Figure 13.4(c)). Check that all red-black properties are preserved, though the violation caused by z and $z.p$ remains since they are still both red.

The operation (a rotation only) described in the last slide will be referred to as Case 2 in RB-Insert. To summarize, if

1. the only violation comes from z and $z.p$ being both red;
2. the uncle of z is black; and
3. the path from z's grandparent to z is a zigzag,

then we perform a rotation so that all conditions stay the same (the tree still satisfies all red-black properties except the violation caused by z and $z.p$; and the uncle of z is still black) except that the path from z to its grandparent now is a zigzig. Case 2 basically is a preparation for Case 3 in which we finally resolve the violation once and for all and obtain a red-black tree.

Finally in Figure 13.4(c), the only violation is that both z and $z.p$ are red. Note that the uncle of z is black and the path from z's grandparent to z is a zigzig (after Case 2). For this situation, we just do a right-rotation at z's grandparent (the node with key 11) and get the tree depicted in Figure 13.4(d). We also recolor $z.p$ (with key 7) to be black and the sibling of z (with key 11) to be red. Check the tree we get is a red-black tree. (Note that even if the tree depicted in Figure 13.4 is a subtree of a bigger one, after these three steps we get a red-black tree.) Thus, RB-Insert terminates.

The operation (rotation and recoloring) described in the last slide will be referred to as Case 3 in RB-Insert. To summarize, if

1. the only violation comes from z and $z.p$ being both red;
2. the uncle of z is black; and
3. the path from z's grandparent to z is a zigzig,

then we perform a rotation and recolor two nodes. By the end, there is no violation anymore and we get a red-black tree. Finally we summarize RB-Insert in the next slide.

1. Call BST-Insert to insert the new node; denote it by z and color z red (note that z has two nil child nodes.)
2. While $z \neq$ root and $z.\text{color} = z.p.\text{color} = \text{red}$
3. do if $z.p = z.p.p.\text{left}$
4. if the uncle of z is red: Case 1 (recoloring only)
5. else (the uncle of z is black)
6. if $z = z.p.\text{right}$ (zigzag)
7. Case 2 (prepare for Case 3; rotation only)
8. Case 3 (resolve the violation) and exit
9. else ($z.p = z.p.p.\text{right}$)
10. same with "right" and "left" exchanged
11. Set the root to be black if it is red

Note that there are two "big" cases in the while loop: z's parent is either a left child or a right child. We will only discuss the case when z's parent is a left child. There will be three subcases (these are the Case 1, 2 and 3 we mentioned earlier). There will also be three subcases when z's parent is a right child but they are symmetric: just switch "right" and "left".

We next describe the operations of Case 1, 2 and 3 more formally. Case 1 can be found in Figure 13.5. Assume that the whole tree satisfies all red-black properties and the only violation is caused by z and $z.p$: both of them are red. Because it is Case 1, z's uncle, y in the picture, is red as well. (As we mentioned earlier, the grandparent of z must be black.)

Because the tree satisfies Property 2 (except for z and $z.p$) and 3, we know that the roots of α, β, γ, δ, ϵ are all black. Moreover, all of them have the same black-height (otherwise show that there is a violation of Property 3). Let bh denote their black-height, then any path from $z.p.p$ to a leaf has $\text{bh} + 2$ black nodes.

In Case 1, we simply recolor $z.p$ and y to be black (push the black down from $z.p.p$) and recolor $z.p.p$ to be red. It is clear that there is no violation of Property 2 in the subtree depicted. Also Property 3 remains to hold in the whole tree. This is because any path from $z.p.p$ to a leaf still has $\text{bh} + 2$ black nodes! We end Case 1 by setting z to be $z.p.p$. By the end, we get a tree that satisfies all red-black properties, except that there is a possible violation caused by z and $z.p$ if $z.p$ is red as well (otherwise we are done). So in Case 1, we recolor three nodes and push the violation up for two levels.

Case 2 is very simple so check Figure 13.6. Again, assume the whole tree satisfies all red-black properties and the only violation is caused by z and $z.p$: both are red. In Case 2, the uncle of z is black (so Case 1 does not apply) and the path from z to z's grandparent is a zigzag. Then in Case 2, we just do a left rotation at $z.p$ and set z to be $z.p$ (the node with key A in the picture).

Because the whole tree satisfies Property 2 (except for$z$ and $z.p$) and Property 3 before the rotation, we know that all the roots of α, β, γ are black and have the same black-height, denoted by bh. Thus, before rotation every path from the node with key C to a leaf has $\text{bh} + 2$ black nodes. After the rotation, it is easy to check that the only violation to Property 2 is caused by z and $z.p$. And Property 3 is preserved: all paths from the node with key C to a leaf still have $\text{bh} + 2$ black nodes. Thus, the tree we get after Case 2 satisfies all red-black properties except the one caused by z and $z.p$, but now the path from z's grandparent to z is a zigzig (so we are now ready to apply Case 3 and finally resolve the violation).

In Case 3, we have a tree that satisfies all red-black properties except a violation caused by z and $z.p$. Moreover, the uncle of z is black and the path from z to z's grandparent (which must be black) is a zigzig. Because the tree satisfies Property 3, we know that all the roots of α, β, γ, δ are black (note that here the root of δ is black because it is the uncle of z) and have the same black-height, denoted by bh. So all paths from $z.p.p$ (the node with key C in the picture) to a leaf have bh $+ 2$ black nodes.

In Case 3, we perform a right rotation at $z.p.p$ (the node with key C in the picture). After the rotation, we recolor $z.p$ (B in the picture) to be black and the sibling of z (C in the picture, the grandparent of z before the rotation) to be red. Now it can be shown that this tree satisfies all the red-black properties. In particular, all paths from $z.p$ to a leaf still have $\text{bh} + 2$ black nodes so Property 3 is preserved (why?). It also satisfies Property 2 because there is no violation in the subtree depicted and the root of this subtree (B in the picture) is now black. So we are done and RB-Insert terminates.

To summarize, RB-Insert starts with at most one violation caused by z (the new node) and $z.p$: both are red. Then depending on the color of z's uncle, we either apply Case 1 or Case 2/3. In Case 1, we recolor three nodes and push the violation up for two levels (and make sure that no other violations occur). In Case 2/3, we perform one (Case 3) or two (Case 2) rotations followed by recoloring two nodes, and finally obtain a red-black tree. (Think about what happens if we never run into Case 2 or 3. Will we still get a Red-Black tree? Also when do we get to use line 11 of RB-Insert?)

A formal proof of the correctness of RB-Insert is given in the textbook, where you can find a loop invariant and how it is maintained with roughly the same ideas described in the previous slides. About the running time of RB-Insert:

1. How many times do we run into Case 2/3? At most once! By the end of Case 3 we get a RB tree and RB-Insert terminates.
2. How many times do we run into Case 1? $O(\lg n)$ because every time we run into Case 1, we push the violation up for two levels but the tree only has $O(\lg n)$ levels.

As a result, the worst-case running time of RB-Insert is $O(\lg n)$. Check RB-Delete in the textbook (similar idea but more involved).