Red-Black and 2-3-4 Trees

July 16, 2020

Description

Red-black trees are loosely self-balanced BSTs. They have a 1-to-1 correspondence to 2-3-4 trees, and are easier and more efficient to implement; however, the theory behind the rebalancing is clearer to illustrate with 2-3-4 trees.

Compared to AVL trees, lookup times in a red-black tree are generally slower because they are less strictly balanced. However, insertion and deletion are generally faster because they require fewer rotations to rebalance the tree. In general red-black trees are used for language libraries (e.g. C++'s set and map) and AVL trees are used in databases, where fast lookups are preferred.

Properties

2-3-4 trees have the following properties:

1. Each node has either 2, 3, or 4 children (and 1 fewer key).
2. The depth of every leaf node is the same.

Red-black trees have the following properties.

1. Nodes are colored red or black.
2. The root is always black.
3. No red node has a red child.
4. Suppose that the null children of all the nodes are each represented by an imaginary "nil" node. Then, every path from a given node to each of its descendant nil nodes has the same number of black nodes.

Note that by Property 4, every root-to-nil path has the same number of black nodes. We refer to this value as the *black height* of the tree. The black height of a red-black tree is equal to the height of its corresponding 2-3-4 tree.

Insertion

To insert into a red-black tree, we start with a normal BST insert and color the inserted node N red. There are three cases for the position of N.

1. N is the root. In this case, we just color the insertednode the black and we have increase the black height of the tree by 1.

2. N has a black parent. In this case, the black height has not changed and no double red was created, so we are done.

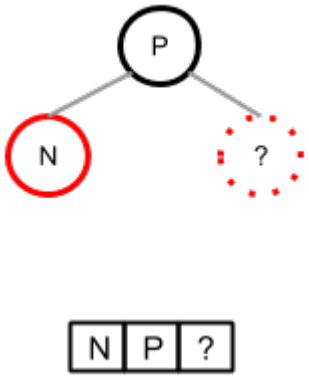

Figure 1: Insertion where N has a black parent

3. N has a red parent. Since the root is always black, we know a grandparent exists. This case is further divided into two cases:

   (a) N's uncle is black or null. Note that S must be a leaf, otherwise black height will not be constant. To rebalance, we simply perform an AVL rotation about the grandparent and recolor.

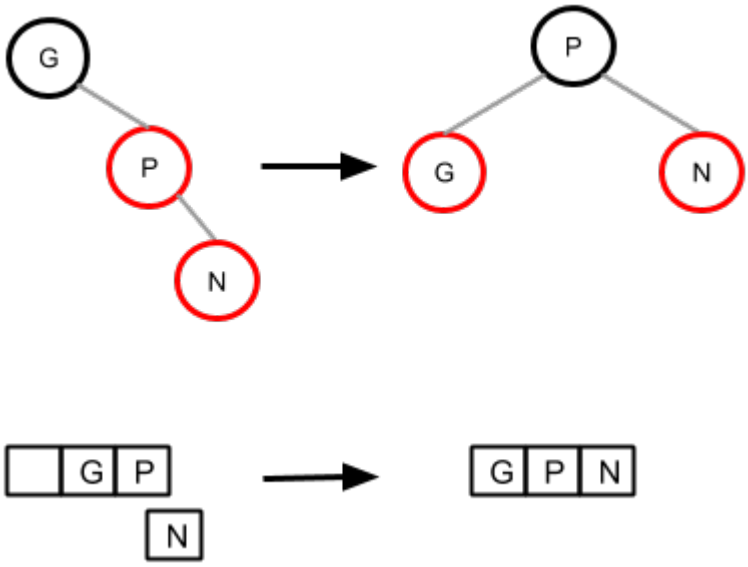

Figure 2: Insertion where N has a red parent and black/null uncle: RR rotation

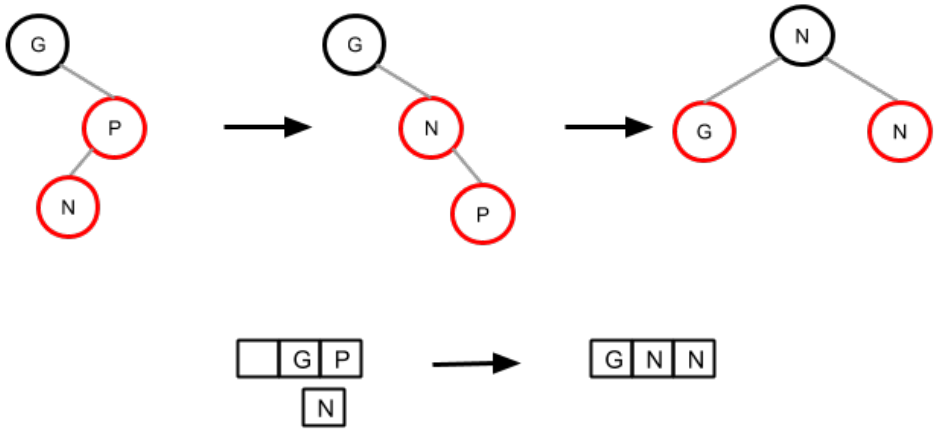

Figure 3: Insertion where N has a red parent and black uncle: RL rotation

(b) N's uncle is red. To rebalance, we color the parent and uncle black, color the grandparent red (if it is not the root), and then recurse on the grandparent if we created a double red. Note how this corresponds to "pushing up" a node in the 2-3-4 tree.

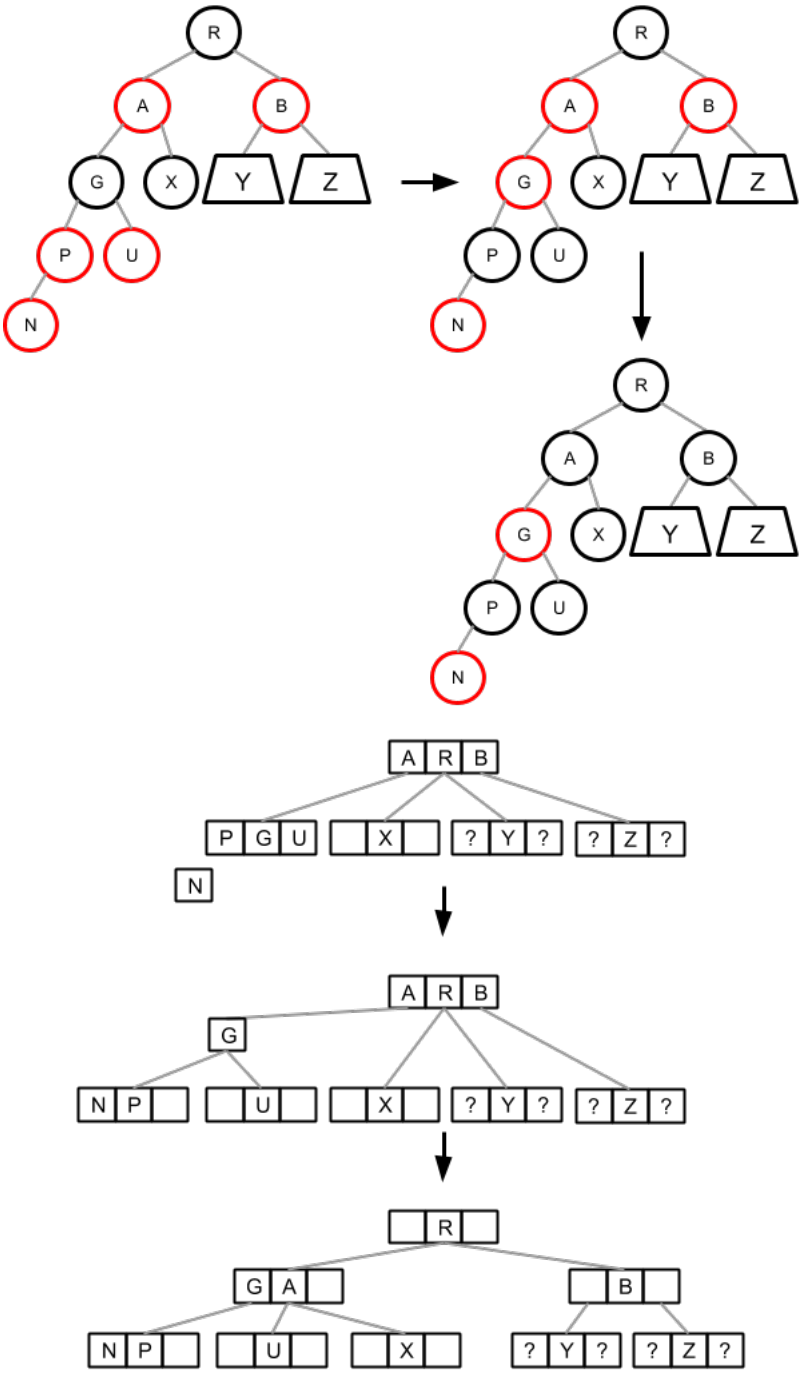

Figure 4: Insertion where N has a red parent and red uncle

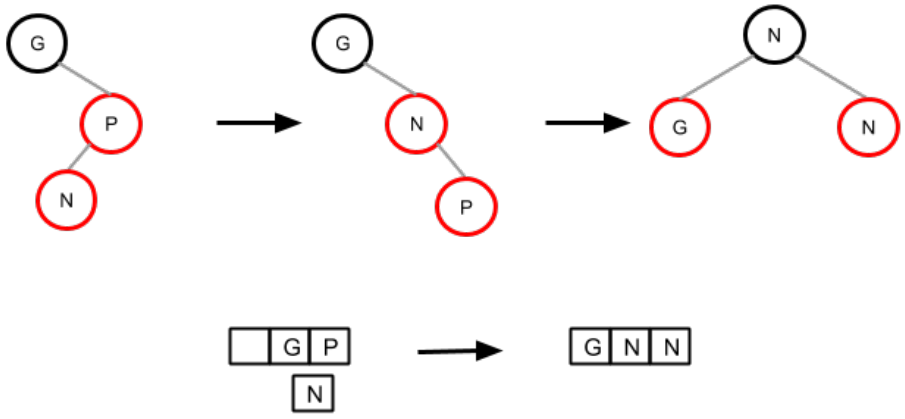

Figure 5: Insertion where N has a red parent and black/null uncle: RL rotation

Deletion

For deletion, we once again start with a normal BST delete. Recall that in a BST deletion, the node we ultimately remove from the tree will either be a leaf or have only one child. However, if we are deleting a red node, it cannot have a single black child because the black height would be different between the left and right subtrees, and by rule, it cannot have a red child. That leaves us with three cases:

1. Red leaf. The node can simply be removed without changing the black height.

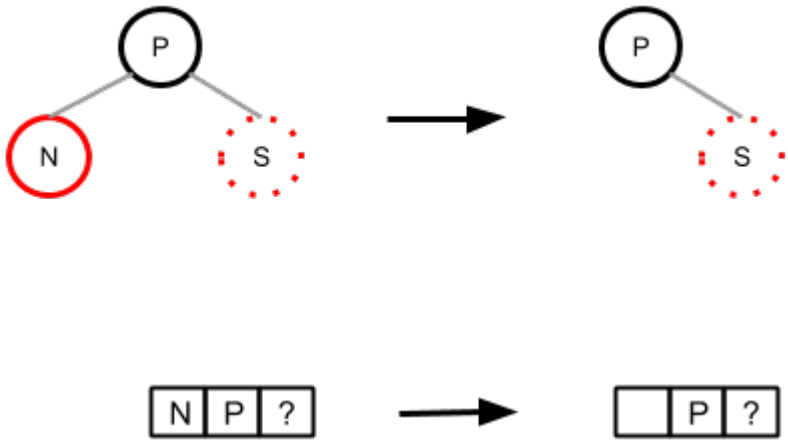

Figure 6: Deletion where N is a red leaf

2. Black node with a single child. Note that its child must be red to maintain a constant black height. We replace N with the child and color the child black. The black height of the tree does not change.

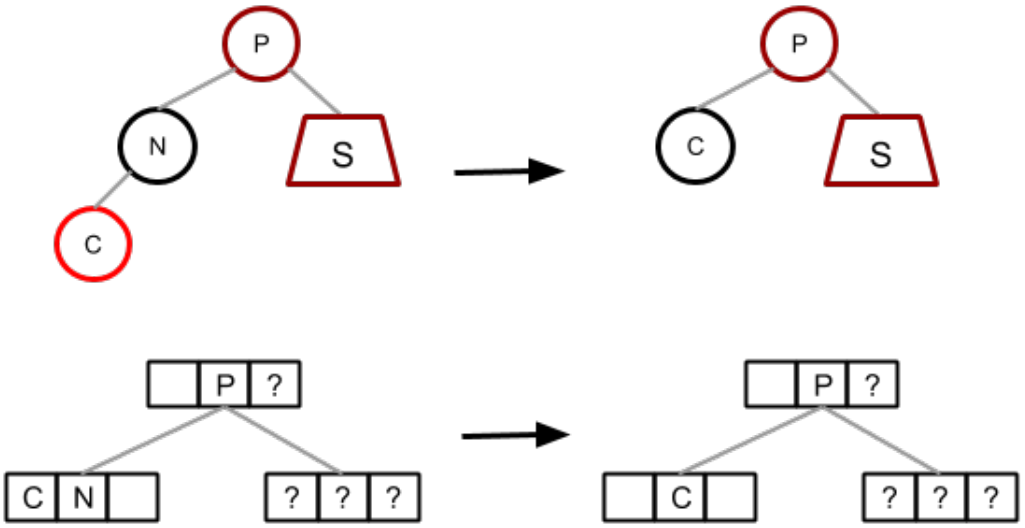

Figure 7: Deletion where N is a black node with a single child

3. Black leaf. This case is further divided into three cases.

   (a) N has a black sibling and a red nephew. N is a leaf, so ? must be red or null to maintain constant black height. To rebalance, we perform an AVL rotation about the parent and recolor.

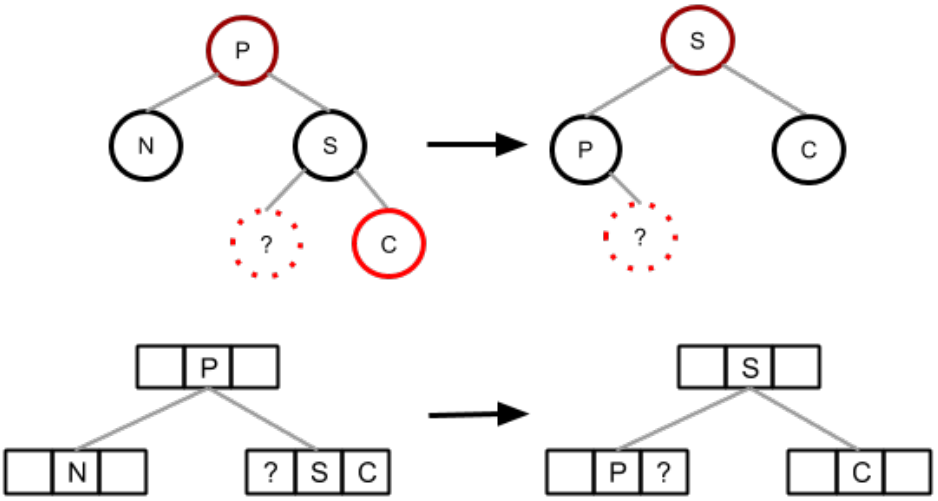

Figure 8: Deletion where N is a black leaf and has a black sibling and a red nephew: RR rotation

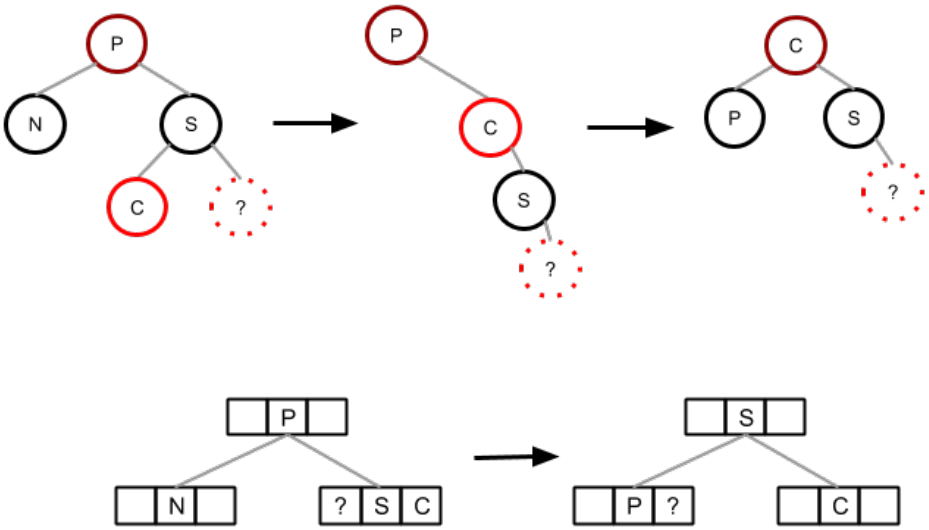

Figure 9: Deletion where N is a black leaf and has a black sibling and a red nephew: RL rotation

(b) N has a black sibling and no red nephews. Here, we color the sibling red and the parent black. Then, we recurse on the parent (case 3) on if it was already black and is not the root. Note that there is no risk of double red

because both children are black or null.

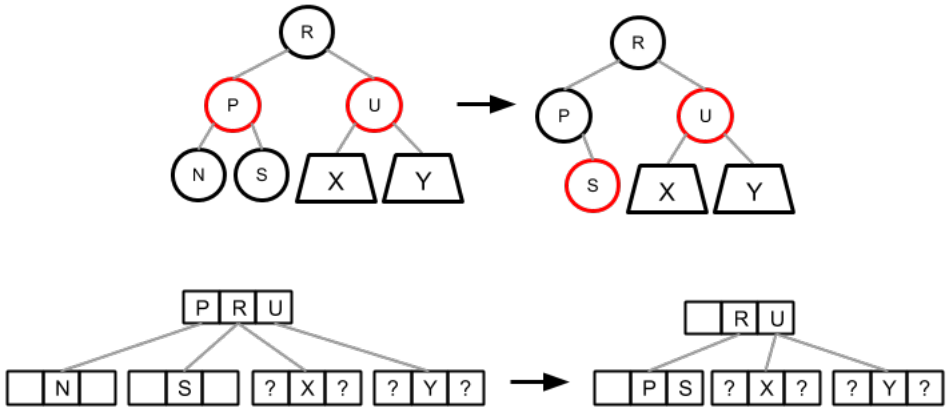

Figure 10: Deletion where N has a black sibling, no red nephews, and a red parent

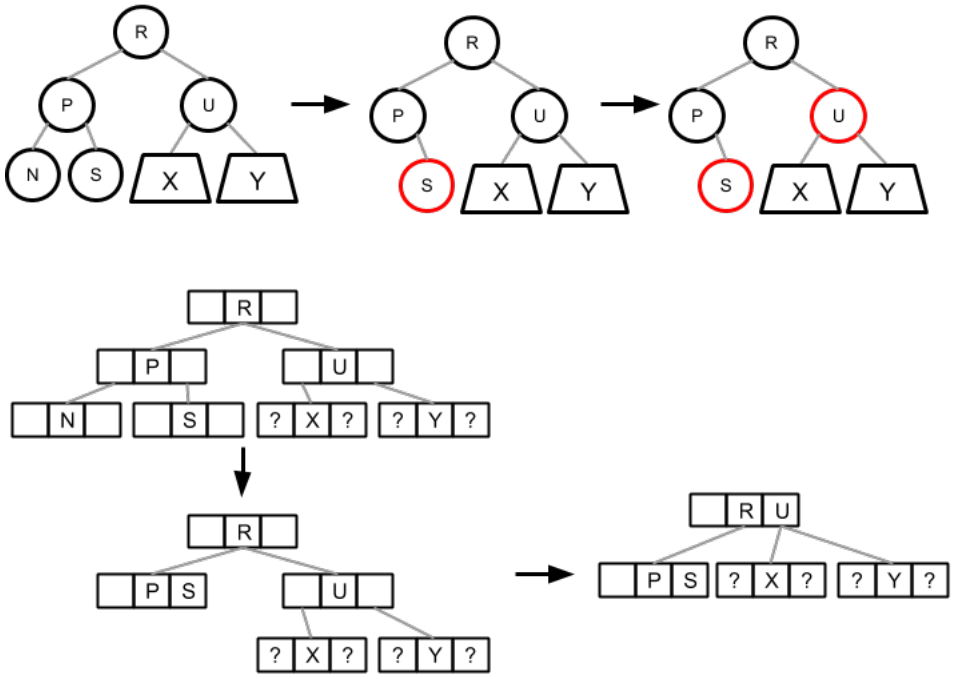

Figure 11: Deletion where N is a black leaf and has a black sibling, no red nephews, and a black parent

(c) N has a red sibling. Note that P has to be black to prevent a double

red. Also, X and Y must be non-null, black and have no black children because the black height is 2. To rebalance, we perform a normal rotation about P, color P red, and color S black. Now we have a valid tree and we are deleting a black leaf with a black sibling, so we can proceed to case 3a or 3b.

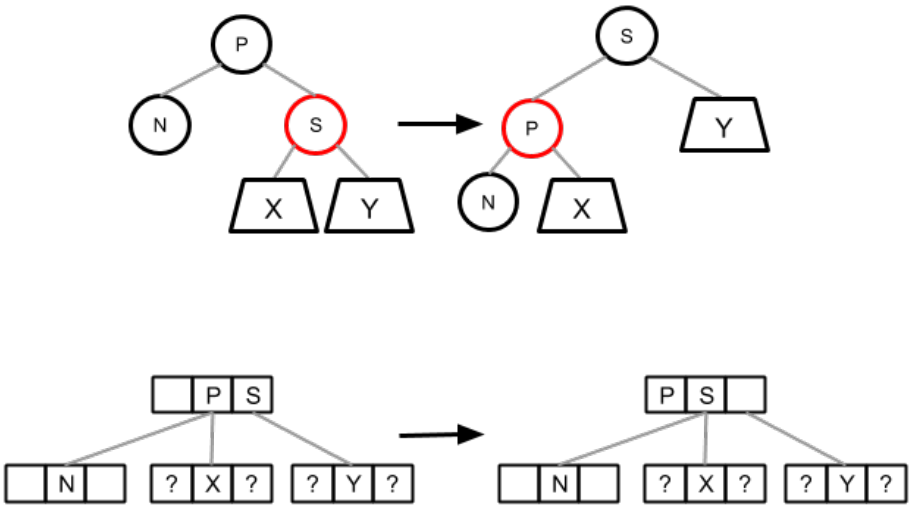

Figure 12: Deletion where N is a black leaf and has a red sibling

Time Complexity

Suppose the black height of a red-black tree is h. Then the smallest possible number of nodes n occurs when all the nodes are black (in the case of a 2-3-4 tree of height h, when all the nodes are 2-nodes), so $n = 2^h - 1$. The largest possible number of nodes n is when each black node has 2 red children (in the case of 2-3-4 tree of height h, when every node is a 4-node). In this case, the tree has height $2h$, and therefore $n = 2^{2h} - 1 = 4^h - 1$. From the perspective of a 2-3-4 tree, $n = 3\sum_{i=1}^{h} 4^{i-1} = 4^h - 1$. Thus, $\log_4 n < \log_4(n+1) \leq h \leq \log_2(n+1) < 1 + \log_2 n$. Since all operations are worst case $O(h)$, we have an asymptotic runtime of $O(\log n)$.