CS241 -- Lecture Notes: Self-Balancing Binary Search Tree

Daisy Tang

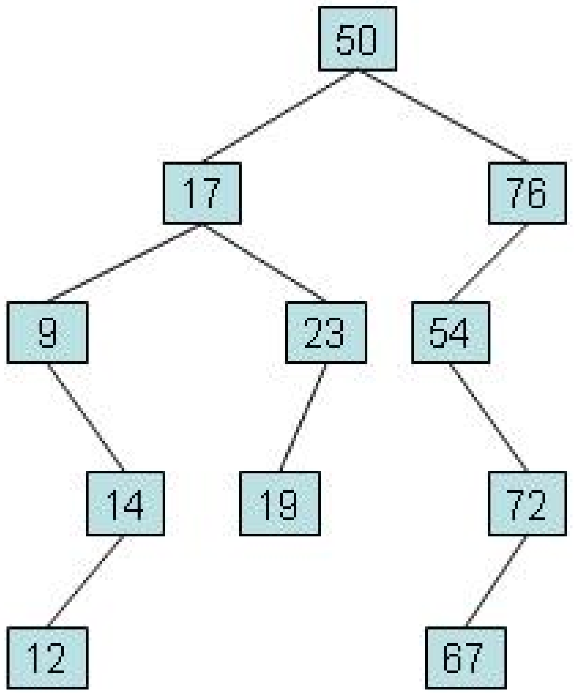
Back To Lectures Notes

Self-balancing binary search tree. Click here to see the animation.

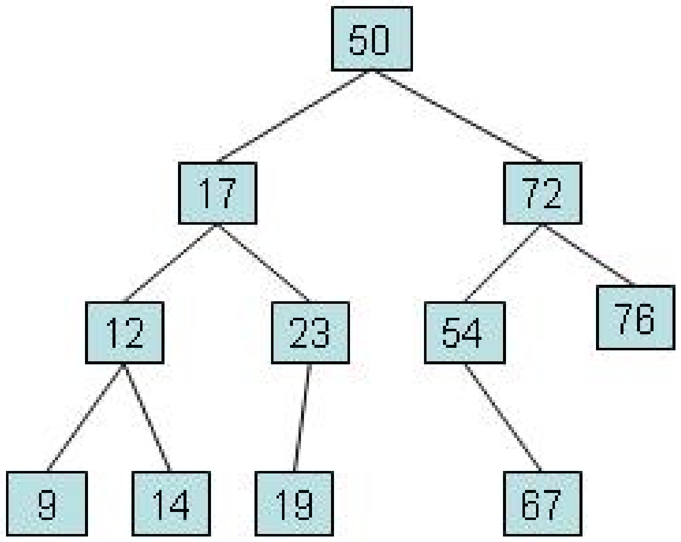
Introduction

Definitions:

- A self-balancing binary search tree or height-balanced binary search tree is a binary search tree (BST) that attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible at all times, automatically.



An unbalanced tree



The same tree after being height-balanced

Overview

Question: What is the major disadvantage of an ordinary BST?

Most operations on a BST take time proportional to the height of the tree, so it is desirable to keep the height small.

Self-balancing binary trees solve this problem by performing transformations on the tree at key times, in order to reduce the height. Although a certain overhead is involved, it is justified in the long run by ensuring fast execution of later operations.

The height must always be at most the ceiling of $log_2 n$.

Balanced BSTs are not always so precisely balanced, since it can be expensive to keep a tree at minimum height at all times; instead, most algorithms keep the height within a constant factor of this lower bound.

Operation	Big-O time
Lookup	$O(\log n)$
Insertion	$O(\log n)$
Removal	$O(\log n)$
In-order traversal	$O(n)$

Implementations:

- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

Self-Balancing BST Applications:

- They can be used to construct and maintain ordered lists, such as priority queues.
- They can also be used for associative arrays; key-value pairs are inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables.
- They can be easily extended to record additional information or perform new operations. These extensions can be used, for example, to optimize database queries or other list-processing algorithms.

AVL Trees

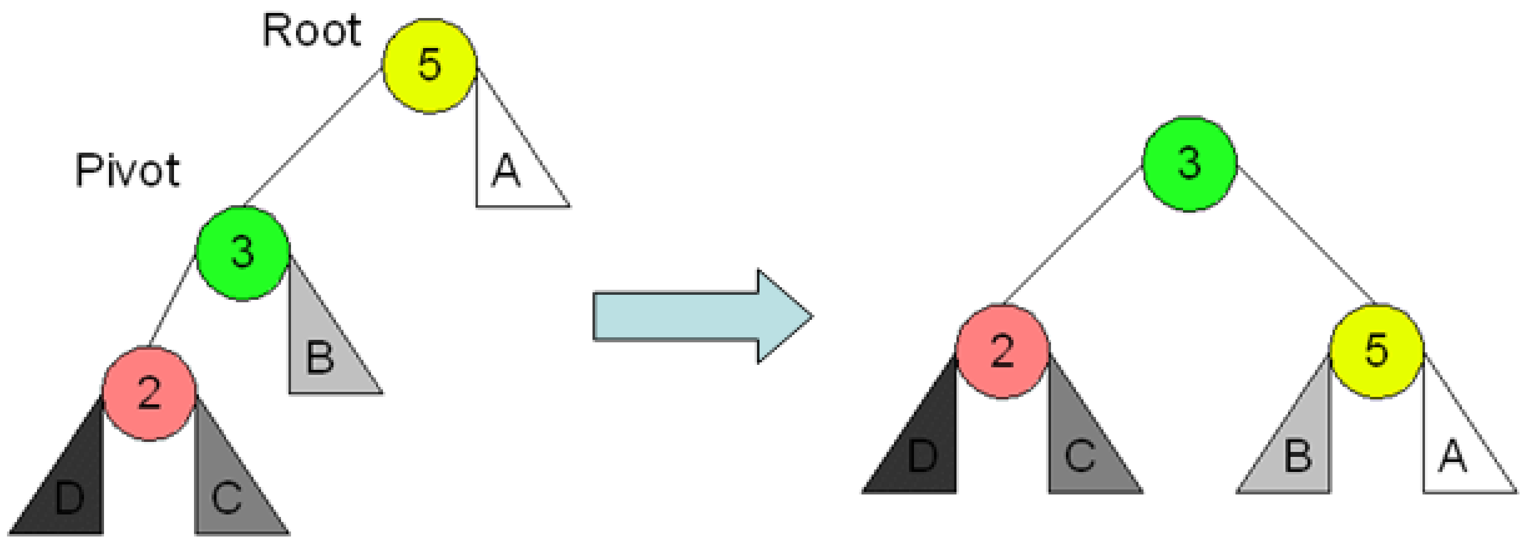
Properties of an AVL tree:

- In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced.
- Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where *n* is the number of nodes in the tree.
- Insertions and deletions may require the tree to be *rebalanced* by one or more tree rotations.
- The **balance factor** of a node is *the height of its right subtree minus the height of its left subtree* and a node with a balance factor 1, 0, or -1 is considered balanced.

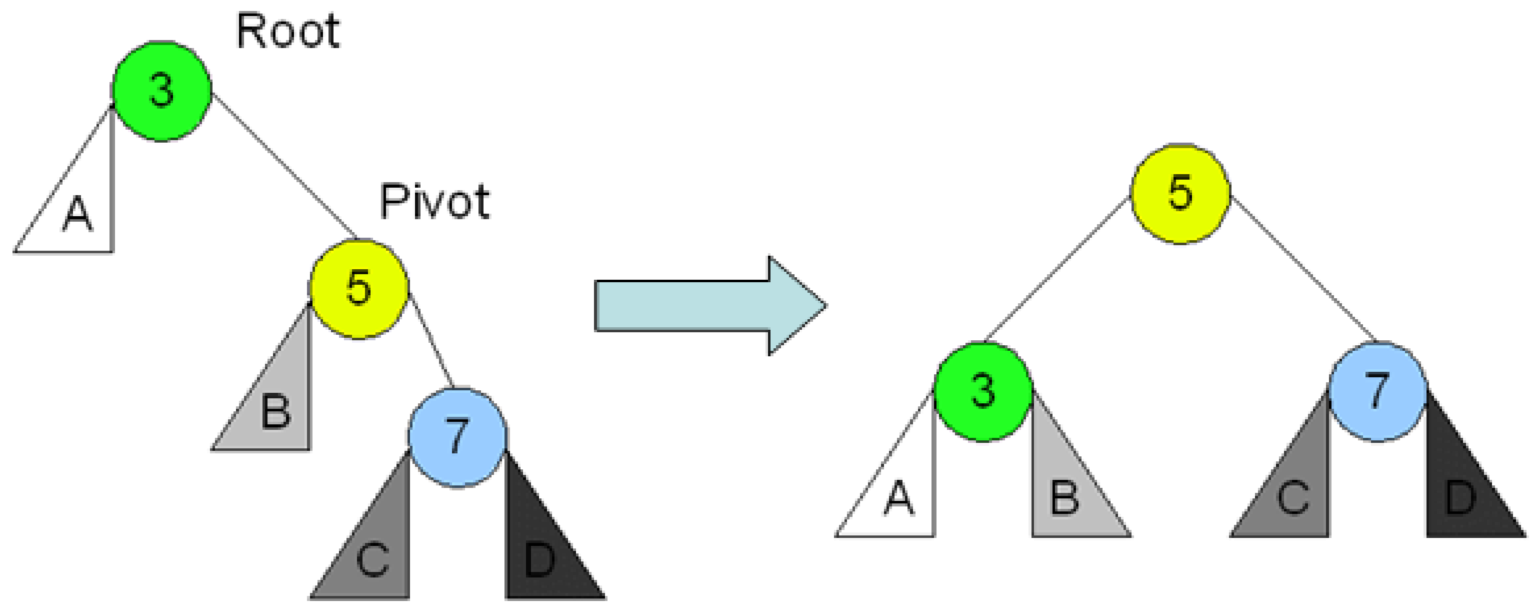
Insertion:

- After inserting a node, it is necessary to check *each of the node's ancestors* for consistency with the AVL rules.
- For each node checked, if the balance factor remains 1, 0, or -1 then no rotations are necessary. Otherwise, it's unbalanced.
- After each insertion, at most two tree rotations are needed to restore the entire tree.

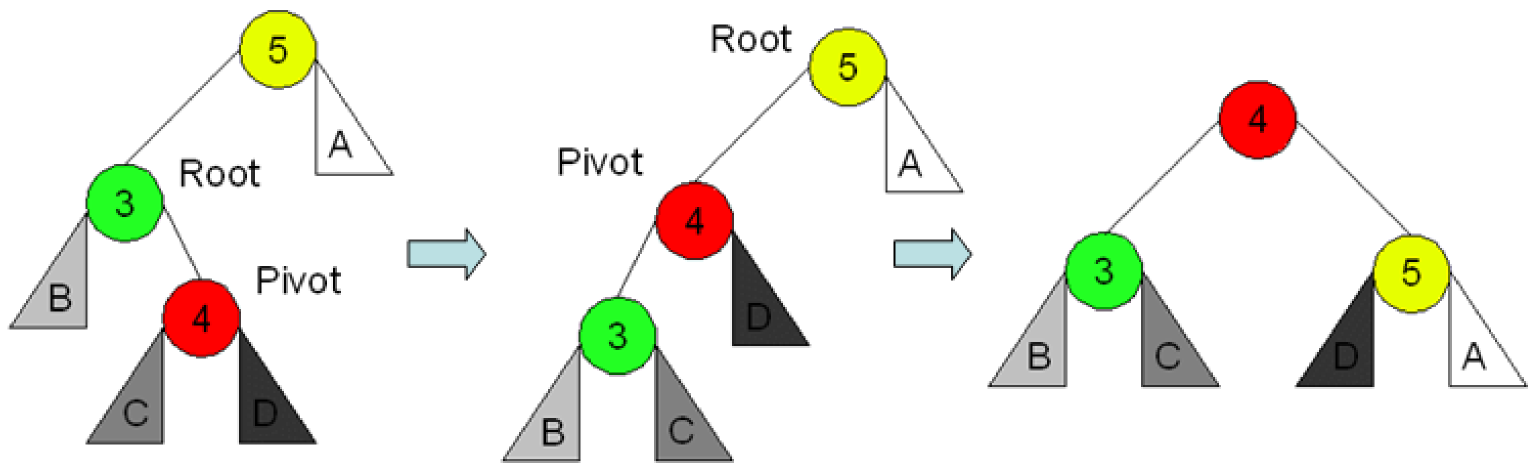
There are four cases, choosing which one depends on different types of unbalanced relations. In the following cases, assume Root is the initial parent before a rotation and Pivot is the child to take the root's place.



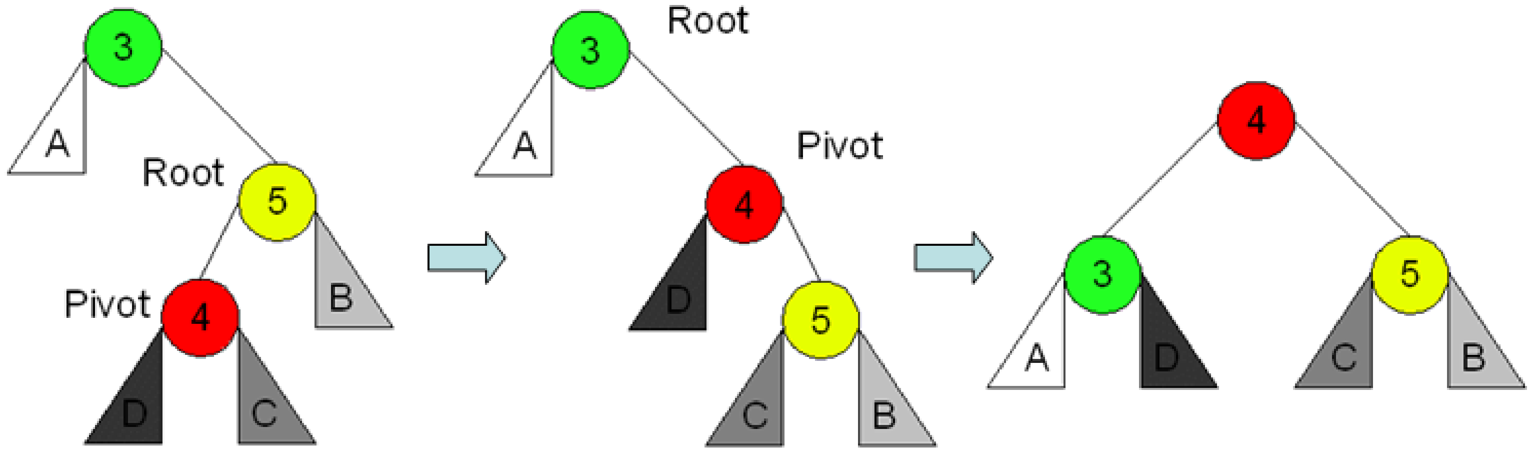
Left-Left Case: Right Rotation



Right-Right Case: Left Rotation



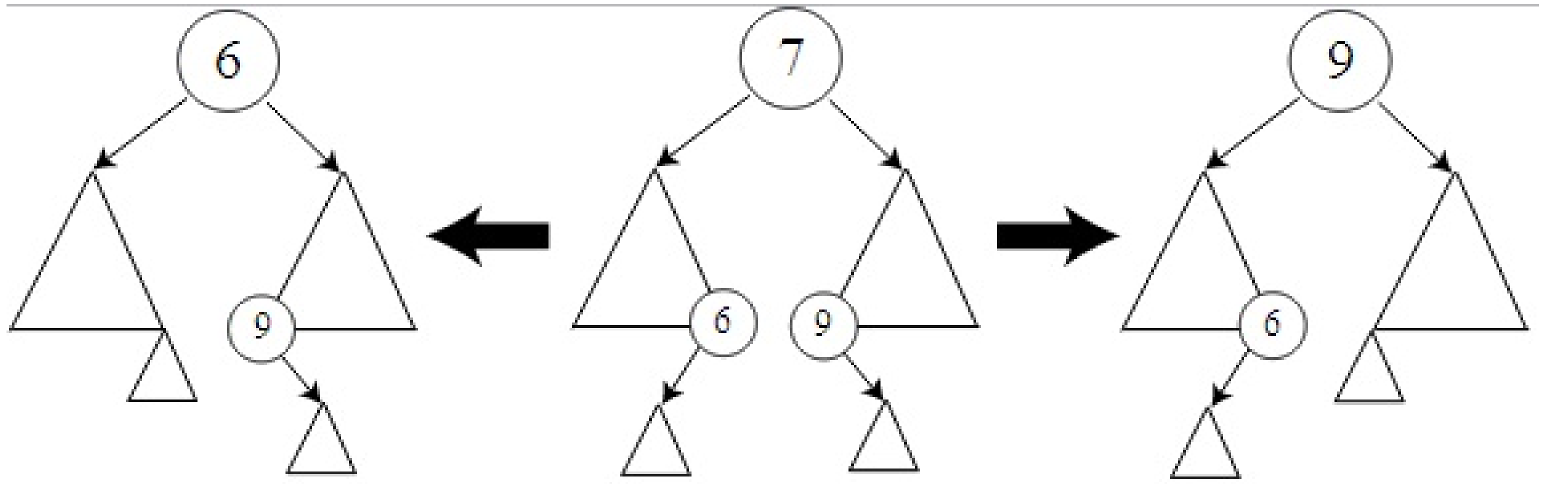
Left-Right Case: Left-Rotation and then Right-Rotation



Right-Left Case: Right-Rotation and then Left-Rotation

Deletion:

- If a node is a leaf, remove it.
- If the node is not a leaf, replace it with either the largest in its left subtree (rightmost) or the smallest in its right subtree (leftmost), and remove that node. The node that was found as replacement has at most one subtree.
- After deletion, retrace the path from parent of the replacement to the root, adjusting the balance factors as needed.
- More complicated rules for stopping. The retracing can stop if the balance factor becomes -1 or +1 indicating that the height of the subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.



Overall, the time required is *O*(log *n*) for lookup, plus a maximum of *O*(log *n*) rotations on the way back to the root, so the deletion can be completed in *O*(log *n*) time.

Lookup (Search):

Lookup in an AVL tree is exactly the same as in an unbalanced BST. Because of the height-balancing of the tree, a lookup takes *O*(log *n*) time.

Example. Insert 14, 17, 11, 7, 53, 4, 13, 12, 8 into an empty AVL tree and then remove 53, 11, 8 from the AVL tree. Please take a look at the following slides for AVL tree insertion and deletion animation (use the slide show mode).

Your exercise:

- Build an AVL tree with the following values: 15, 20, 24, 10, 13, 7, 30, 36, and 25.
- Remove 24 and 20 from the above tree.

Discussion:

- Compare binary search trees with hash tables. Find pros and cons of each data structure.
    - Time complexity of operations
    - Space complexity of data structure
    - Handling varying input sizes
    - Traversal
    - Other supported operations?

Red-Black Trees

A red-black tree (RB-tree) is a type of self-balancing BST. It is complex, but has a good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in *O*(log *n*) time, where *n* is the total number of elements in the tree.

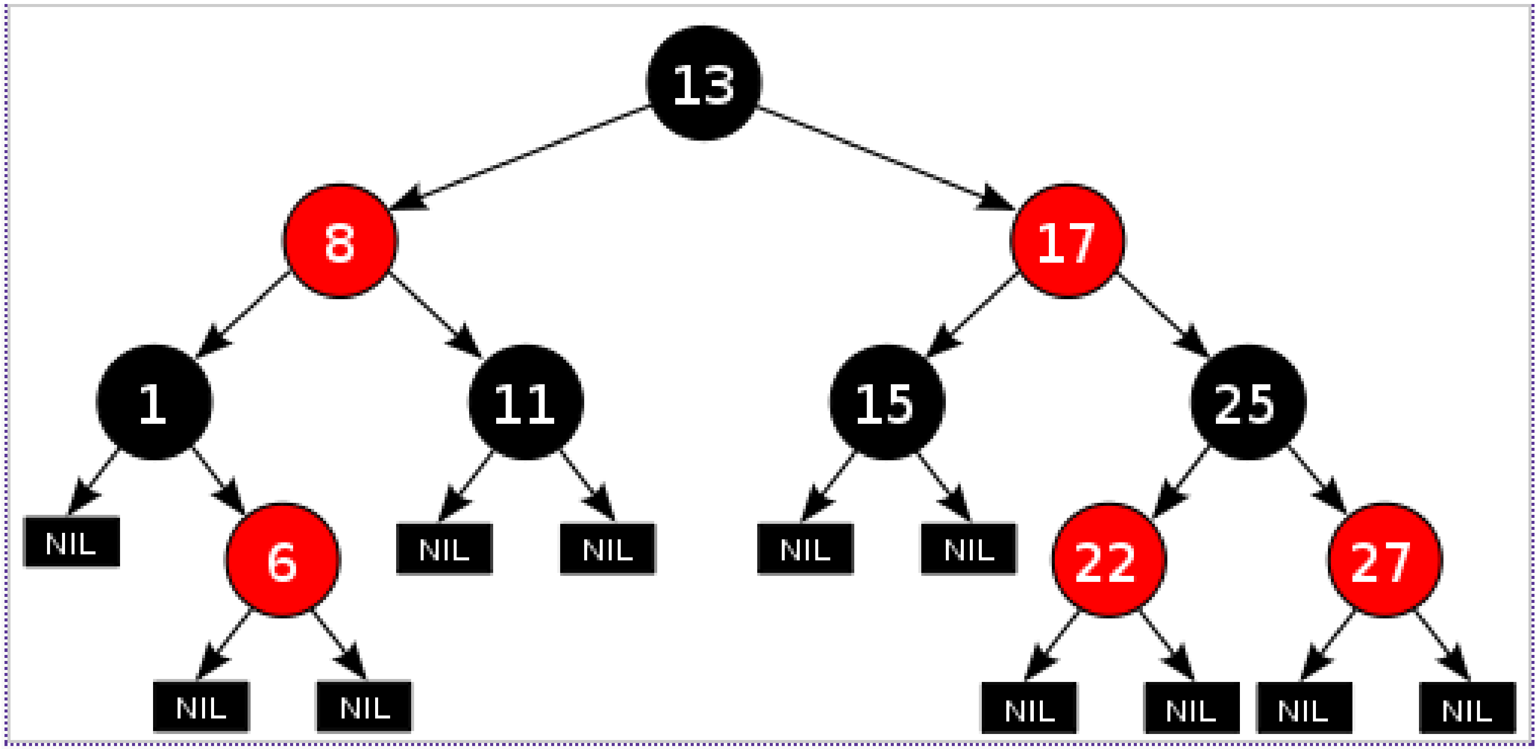
- In RB-trees, the leaf nodes are not relevant and do not contain data. A null child pointer can encode the fact that this child is a leaf.
- Like BSTs, RB-trees allow efficient in-order traversals of elements.
- The search time on a RB-tree results in *O*(log *n*) time.

Properties:

A RB-tree is a BST where each node has a color attribute, the value of which is either *red* or *black*. In addition to the ordinary requirements imposed on BSTs, the following additional requirements apply to RB-trees:

1. A node is either red or black.
2. The root is black.

3. All leaves are black.
4. Both children of every red node are black.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.



The above constraints enforce a critical property of RB-trees:

- The longest path from the root to any leaf is no more than twice as long as the shortest path from the root to any other leaf in that tree.
- The result is that the tree is roughly balanced.
- Insertion, deletion, and search require worst-case time proportional to the height of the tree, the theoretical upper bound on the height allows RB-trees to be efficient in the worst case.

To see why these properties guarantee this, it suffices to note that no path can have two red nodes in a row, due to property 4. The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black nodes, by property 5, this shows that no path is more than twice as long as any other path.

Insertions and removals are quite complex in a RB-tree in order to keep the properties.

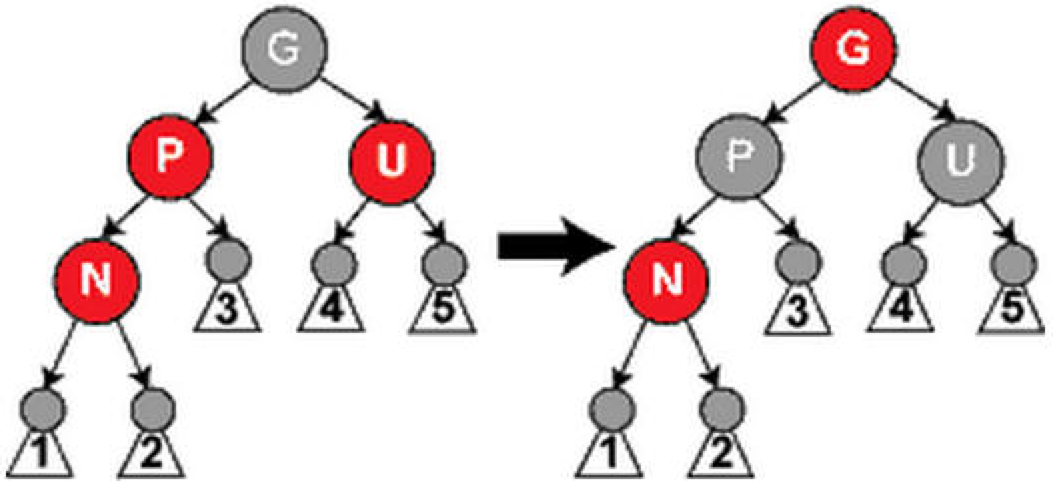
Insertion:

Insertion begins by adding the node as any BST insertion does and by coloring it red. It's a red inner node with two black leaves.

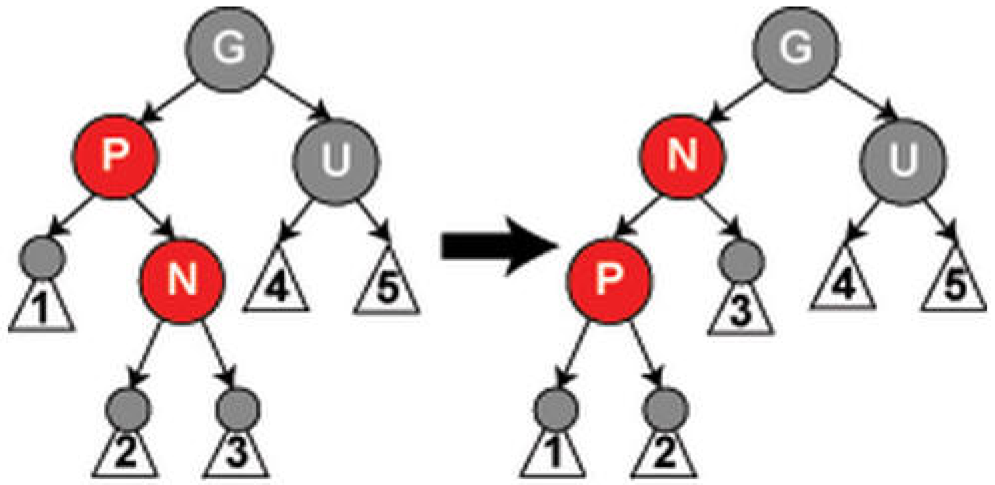
- Property 3 (all leaves are black) always holds.
- Property 4 (both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.
- Property 5 (all paths have same number of black nodes) is threatened only by adding a black node, repainting a red node black (or vice versa), or a rotation.

In the following description, we have labels N (current node), P (N's parent), G (N's grandparent), and U (N's uncle).

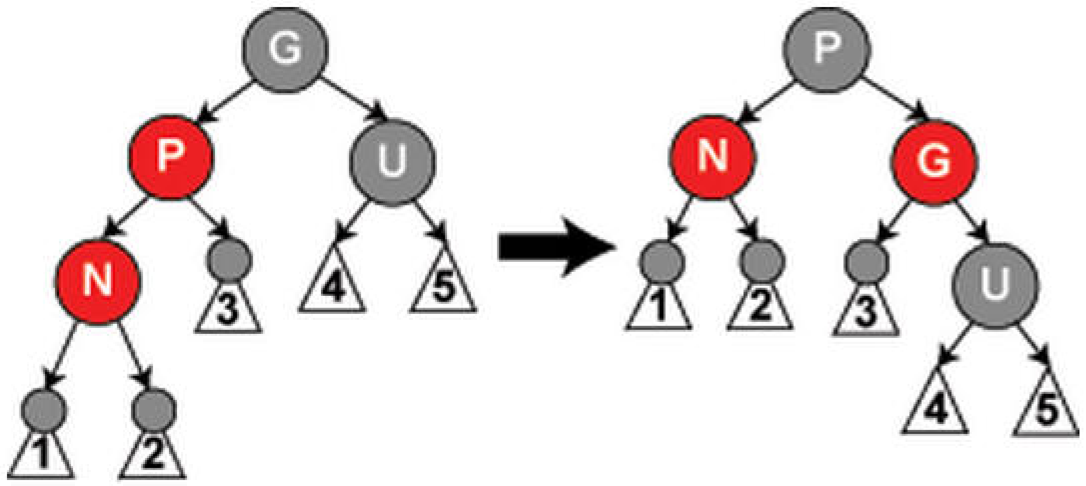
- Case 1: N is root. It's repainted black to satisfy property 2.
- Case 2: P is black. Property 4 (children of red are black) is not violated. Property 5 holds since N has two black leaf children, but N is red.
- Case 3: if both P and U are red, repaint them black and repaint G red. Recursively insert G.



- Case 4: P is red, but U is black; N is the right child of P, and P is the left child of G. Perform left rotation on P. Then go to Case 5.



- Case 5: right rotation. Repaint G and P.



For deletion on a RB-Tree, please see the Wikipedia link for details.

Let's try it out with the following sequence of values: 14, 17, 11, 7, 53, 4, 13, 12, and 8.

Last updated: Oct. 2013