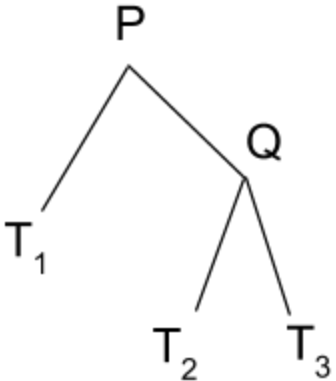
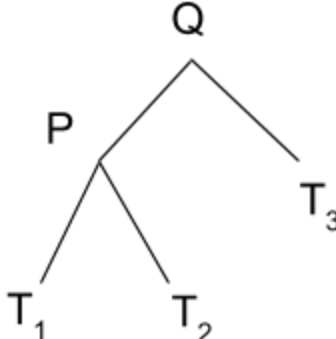
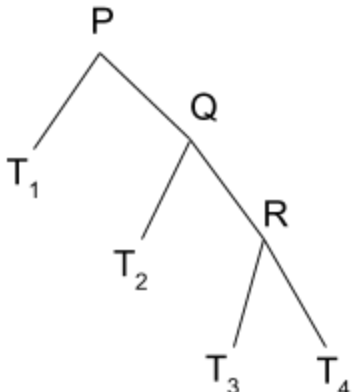
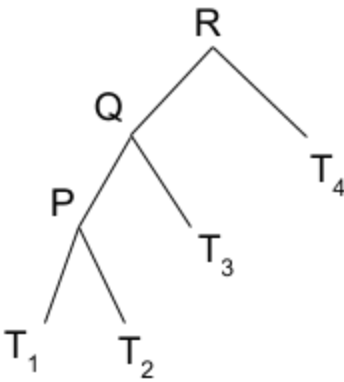
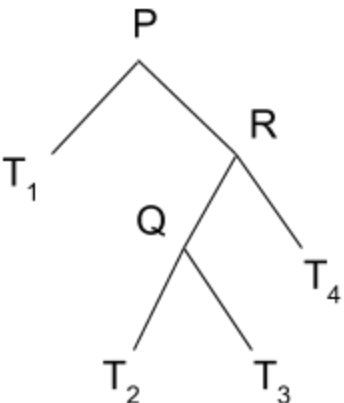
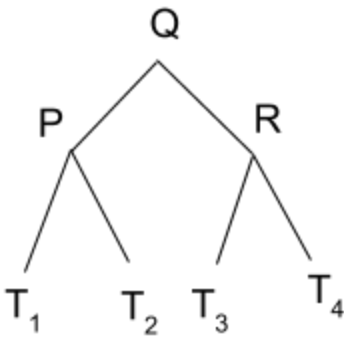


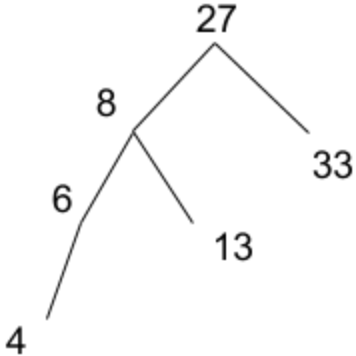
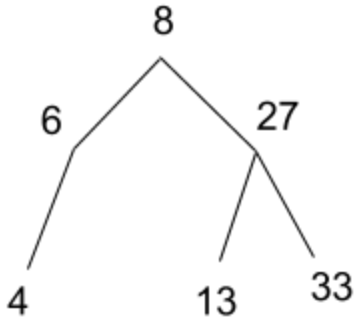
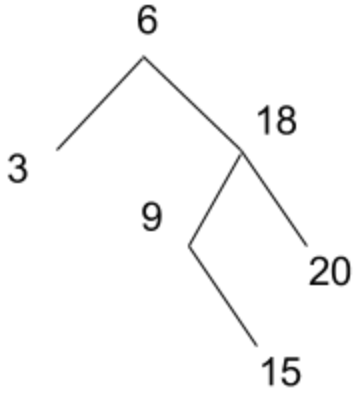
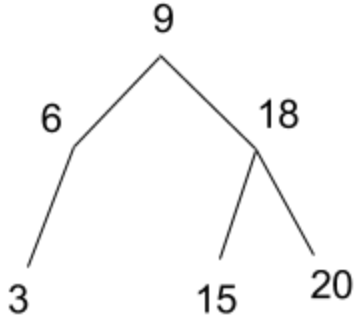
Balanced BST: a BST in which every node satisfies some balancing condition between its left and right subtrees. The goal is to keep the height of trees to be $O(\log n)$. When nodes go out of balance, because of add or remove operations, the tree is rotated to restore balance at all nodes. In the following examples, the balance property is violated at node P because of an operation in the subtree of a child or a grandchild.

<p>[R] Single rotation (Zig): Left rotation at P →</p> 	<p>← Right rotation at Q [L]</p> 
<p>[RR] Double left rotation(Zig-Zig) at P due to R →</p> 	<p>← Double right rotation at R due to P [LL]</p> 
<p>[RL] Double rotation (Zig-Zag) at P due to Q →</p> 	<p>Tree after rotation:</p> 
<p>Symmetric case [LR] is not shown: double rotation</p>	

Rotation operations used to restore balance in search trees

AVL Tree: A binary search tree that satisfies the following balance condition at every node: the difference between the heights of the left subtree and the right subtree is at most one. A new field is added to Entry class of tree node to keep track of the height of the subtree rooted at that node.

AVL trees inherit all operations of the BST class. When an add operation is performed, the height of ancestors of the new node may increase by 1. Some of these nodes may violate the balance condition. A single or double rotation is performed at the lowest node that goes out of balance, to restore balance to the tree. Similarly, when a remove operation is performed, the heights of ancestors of the removed node may decrease by 1. As a consequence, some nodes may go out of balance. Just one single or double rotation is needed at the lowest node that goes out of balance, to restore balance to the tree.

<p>C1: Left subtree of left child of node u has excess height: rotate right at u. Single right rotation [R] at 27 in example below:</p> 	<p>C1: Tree after rotation is shown below. Heights of nodes involved in the rotation needs to be updated after the rotation (8, 27 in this example):</p> 
<p>C2: Left subtree of right child of u has excess height. Perform a double rotation [RL] at u. Double rotation [RL] is needed at 6 in this example:</p> 	<p>C2: Tree after rotation is shown below. Heights of nodes 6, 18, and 9 need to be updated.</p> 
<p>C3: Right subtree of right child of u has excess height. Symmetric to C1: Single left rotation [L] at u.</p>	<p>C4: Right subtree of left child of u has excess height. Symmetric to C2. Double rotation [LR] at u.</p>

Update operations on AVL trees have a downward pass to a new node that is added or removed, and an upward pass updating the height of nodes of affected ancestors. In all cases, at most one (single or double) rotation is performed to restore balance. Some implementations store $\text{left.height} - \text{right.height}$ (difference in heights of subtrees) instead of height.

Red-Black Tree: A binary search tree, where every node has a color: Red or Black. The balance condition is more complex. The number of edge cases in the code is considerably reduced if null pointers at the leaf nodes are replaced by special NIL nodes that are colored Black. A Red-Black tree satisfies the following conditions: (1) root and all NIL nodes are colored black, (2) a red node does not have a red parent, (3) every path from a node u to a descendant leaf has the same number of black nodes ($bh(u)$, the black height of u). It can be shown that the height of a red-black tree with n leaves is at most $2\log(n)$.

The $\text{add}(x)$ operation on Red-Black trees uses the add operation of BST. If x is already in the tree, the element is replaced and no new Entry node is added to the tree. Otherwise, a new Entry node t is added to the tree with x in it. Nodes are created with Red color. If its parent p_t is Black, then no further change is required. If t and p_t are both Red nodes, the tree needs to be repaired, and $\text{repair}(t)$ is called. After repair returns, the root node is colored Black.

Terms:

t current node (red)
 p_t parent of node t
 g_t parent of p_t , grandparent of t
 u_t sibling of p_t , uncle of t

// t and its parent are colored Red

$\text{repair}(t)$:

if p_t is root of tree then return
 // root is colored Black by add after repair

while t is Red do

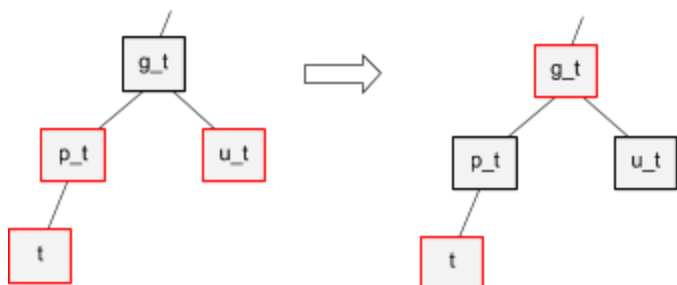
if t is root or p_t is root or p_t is Black then return

if u_t is colored Red // Case 1

set color of p_t and u_t to Black

set color of g_t to Red

$t \leftarrow g_t$; continue



else if u_t is Black, and // Case 2

(2a) $g_t \rightarrow p_t \rightarrow t$ is LL then

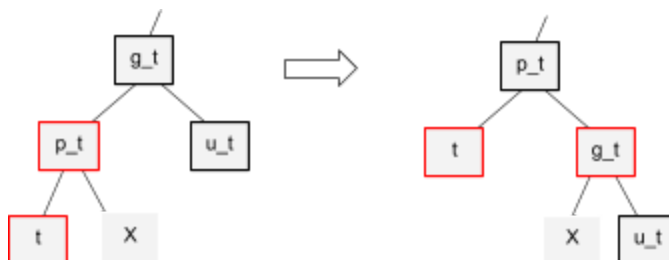
Rotate [R] at g_t

Recolor p_t to Black and g_t to Red
 return

(2b) $g_t \rightarrow p_t \rightarrow t$ is RR then

Rotate [L] at g_t

Recolor p_t to Black and g_t to Red
 return



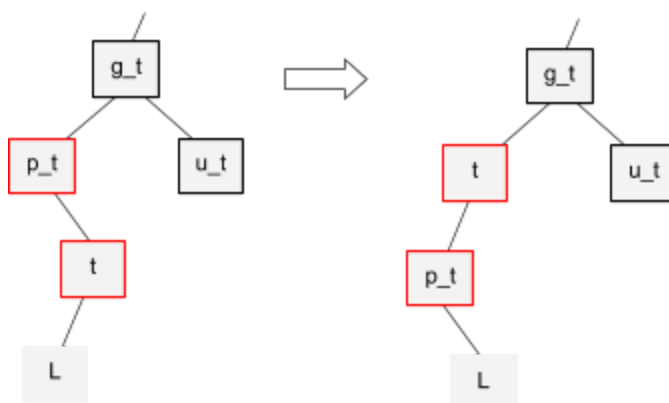
else if u_t is Black, and // Case 3

(3a) $g_t \rightarrow p_t \rightarrow t$ is LR then

Rotate [L] at p_t and apply Case 2a

(3b) $g_t \rightarrow p_t \rightarrow t$ is RL then

Rotate [R] at p_t and apply Case 2b



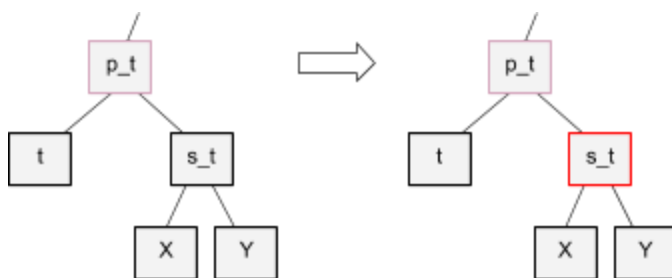
The remove(x) operation starts by applying the remove operation of BST. If x is not in the tree or if the bypassed node is Red, then nothing further needs to be done. Otherwise, a Black node was removed by the bypass operation. Let c be child node of the bypassed node that took its place in the tree. The subtree of c has a black height deficit of 1 with respect to the other parts of the tree. In this case, fix(c) is called. The purpose of fix is to fix the tree by boosting the black height of that node. When fix returns, the root is colored Black.

// t has a deficit of 1 in its black height
fix(t): // In this code, s_t is sibling of t
 while t is not root do

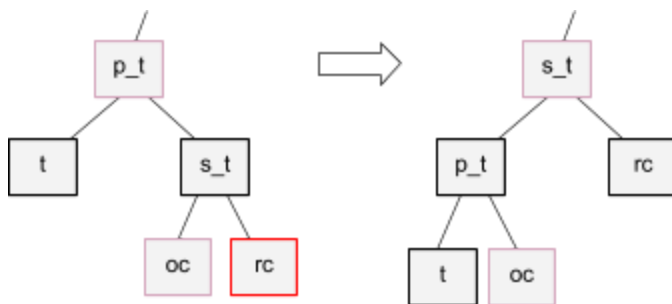
Case 1: t is Red:
 Recolor t Black and return



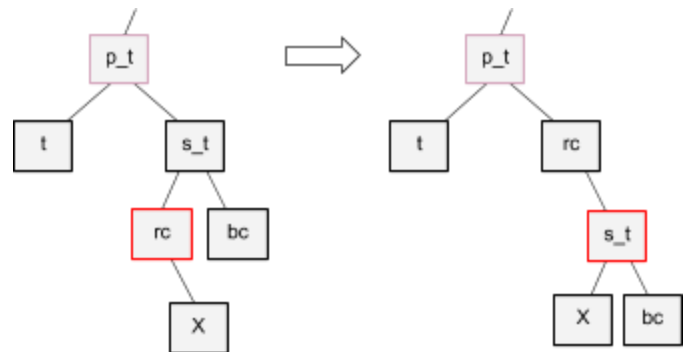
Case 2: s_t and its children are Black:
 Recolor s_t to Red
 $t \leftarrow p_t$



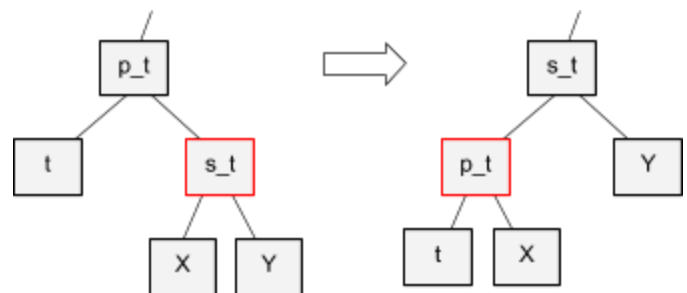
Case 3: s_t is Black, with Red child rc, and
 $p_t \rightarrow s_t \rightarrow rc$ is RR (or LL):
 Rotate [L] (or [R]) at p_t
 Exchange colors of p_t and s_t
 Recolor rc to Black and return



Case 4: s_t is Black, with Red child rc, and
 $p_t \rightarrow s_t \rightarrow rc$ is RL (or LR):
 Rotate [R] (or [L]) about s_t
 Exchange colors of rc and s_t
 Apply Case 3 and return



Case 5: s_t is Red
 Rotate about p_t (same direction as t)
 Exchange colors of p_t and s_t
 Apply one of cases 2-4 and return



Single-pass algorithms for Red-Black Trees: nodes are recolored and rotated on the way down (in find) such that Case 1 of repair, and Case 2 of fix never appear. Then algorithm can return after 1 or 2 rotations.