**[Computer science course notes](http://www.microveggies.com/csci)**

**Home**

[**Lecture20 - Tree balancing - Red black trees**](http://www.microveggies.com/csci/index.php/csci-2270-lecture-notes/8-lecture20-tree-balancing-red-black-trees)

**Details**

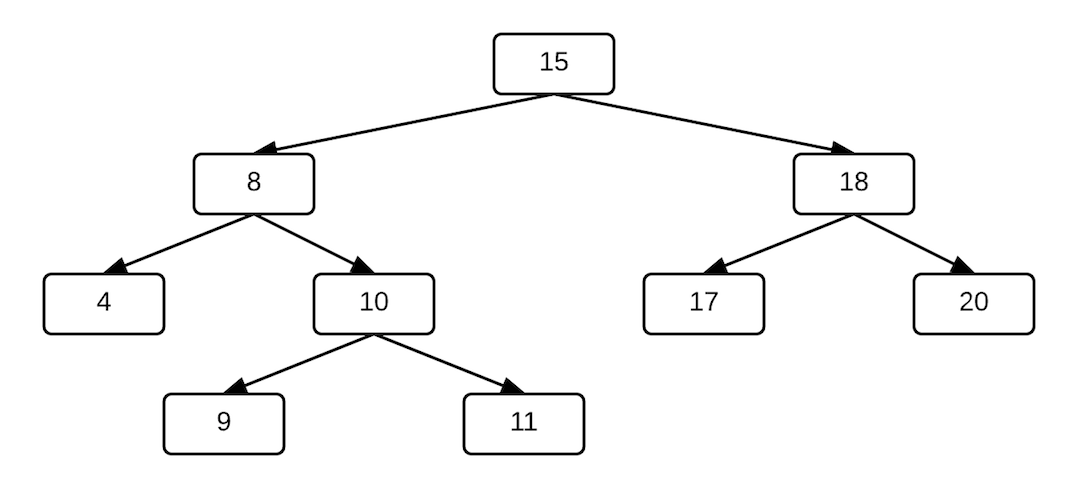
Written by Rhonda Hoenigman

 Published: 02 March 2015

 Hits: 2481

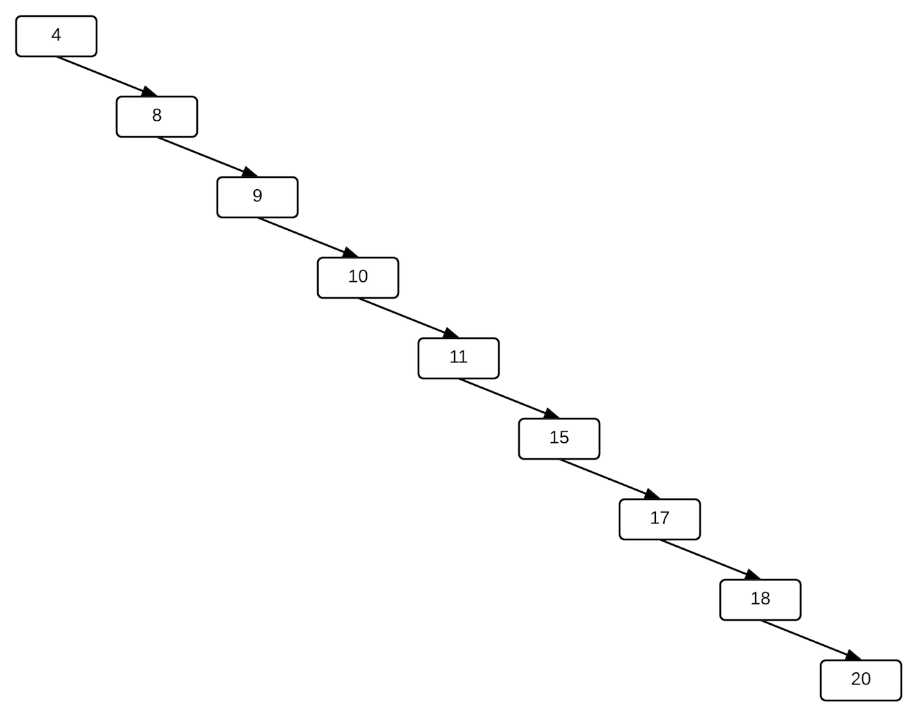
**BST Complexity**

When we search for a specified node in a binary search tree, the time it takes to find the node often depends on how the tree is built. For example, consider a tree built from the following sequence of integers: <15, 8, 4, 18, 10, 17, 20, 9, 11>. Reading the integers in the order they are presented and adding them to the tree generates the following binary search tree:



The nodes are filled at all levels except the last and there are roughly the same number of nodes to the left an right of the root of the tree.

However, the data could be presented in a different order and a different tree would be generated. Sorting the data first, for example, would produce the sequence: <4, 8, 9, 10, 11, 15, 17, 18, 20> and the tree generated from that data ordering would look like:



**Definition of balanced tree**

A balanced binary search tree has the minimum possible maximum height. For each node x, the heights of the left and right subtrees of x differ by at most 1.

When we search for a specified node in a binary search tree, the worst-case runtime occurs when the node is at the bottom of the tree. When the tree is balanced, like the tree in the first image, the distance from the root to any leaf node at the bottom of the tree, is log2(n), where n is the number of nodes in the tree. In this example, there are 9 nodes in the tree and 3-4 levels. In the worst case, we would have to do 4 comparisons to find a node in the tree. Calculating log2(9) ≈ 3.16 shows us that log2(n) is a good approximation. In contrast to the height balanced tree in the first image, the unbalanced tree in the second image could require n comparisons when searching for a node. If were were searching for the 20, for example, we would have to evaluate all 9 nodes in the tree to find the 20. On smaller trees, it may not matter that much if the tree is structured like an n-node linked list, but with thousands or millions of nodes, having a balanced tree can significantly improve the runtime. For example, consider a tree with a million nodes. If the tree is balanced, then the height of the tree is log2(1000000)≈19, meaning we should be able to find anything in the tree in approximately 19 comparisons. However, if the tree is unbalanced, we would have to do up to 1,000,000 comparisons.

In a BST, basic operations, such as search, add, delete, minimum, and maximum run in O(h) time, where h is the height of the tree. When n = h, where n is the number of nodes, then these operations are O(n) and the BST has the same runtime properties as a linked list.

**Tree Balancing**

Unfortunately, it's not always possible to control data ordering to ensure a balanced tree. Instead, we turn to tree-balancing algorithms to ensure that as nodes are added to the tree, they are added to the left and right side of the root and the height of the tree is O(log2(n)).

**Red-black trees**

A red-black tree is a BST, where each node in the tree is also assigned a color, either red or black. We can use this node coloring to build a tree where no path in the tree is more than twice as long as any other path. This is a variation of a binary search tree, has a height O(lg n), a height on the order of lg n, which guaranteed a worst-case runtime of O(lg n) on basic operations such as search, add, and delete.

Each node in the tree has at least the following properties:

* color
* key
* left
* right
* parent

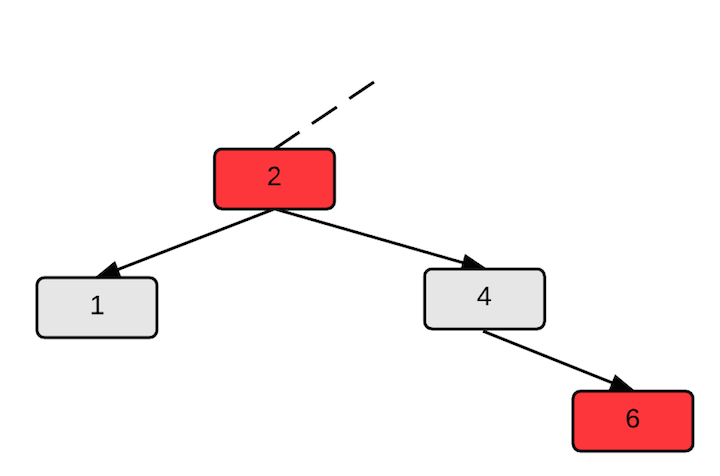
A red-black tree is a binary search tree with the following additional properties:

1. Every node has a color, either red or black.
2. The root is black.
3. Every leaf (NULL) is black. (Leaves are nodes with no children.)
4. If a node is red, the both of its children must be black.
5. For each node, all paths from the node to the leaves at the bottom of the tree contain the same number of black nodes.

Red-black trees also use a single sentinel node for all leaves of the tree. For a red-black tree T, T.nil represents nil throughout the tree, including the root's parent and the leaf nodes. The sentinel is always black. The sentinel T.nil has the same attributes of a regular node.

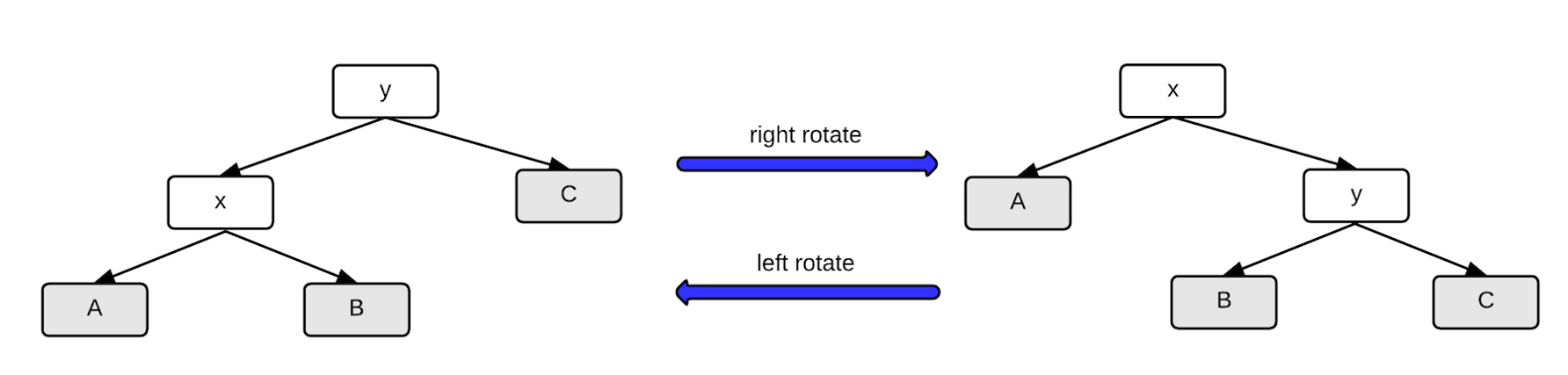
**Rotations: Left and Right**

When we add or delete a node from a red-black tree, the red-black properties can be destroyed. For example, consider the following red-black subtree.



if we delete the 4 from this tree, a red node will now have a red child, which violates property 4. We can restore the red-black properties using rotations and re-coloring the nodes. Rotations are local operations on nodes in the tree that preserve the BST ordering in the tree. They are used to restructure the tree and are used on almost all balanced search trees.

There are two types of rotations: a left rotation and a right rotation.



In this image, the nodes labeled A, B, and C represent subtrees, not individual nodes.

As shown in this image, a subtree rooted at x that undergoes a left rotation can be returned to its original state with a right rotation rooted at y, and vice versa. The left and right rotations are inverses of each other.

The pseudocode for a left rotation (Cormen, page 313):

left\_rotate(T, x) //T is the tree and x is the node about which to rotate

1. y = x.right

2. x.right = y.left

3. if y.left != T.nil

4.   y.left.p = x

5. y.p = x.p

6. if x.p == T.nil

7.   T.root = y

8. elseif x == x.p.left

9.   x.p.left = y

10. else   
11. x.p.right = y

11. y.left = x

12. x.p = y

Pseudocode for a right rotation. The left and right pointers are reversed and we set x with respect to y.

right\_rotate(T,y)

1.  x = y.left

2.  y.left = x.right

3.  if x.right != T.nil

4.    x.right.p = y

5.  x.p = y.p

6.  if y.p == T.nil

7.    T.root = x

8.  else if y == y.p.left

9.    y.p.left = x

10. else

11.   y.p.right = x

12. x.right = y

13. y.p = x

In C/C++, a left rotate implementation could look something like:

left\_rotate( Tree T, node \*x ) {

1. node \*y;

2. y = x->right;

3. /\* y's left subtree becomes x's right subtree \*/

4. x->right = y->left;

5. if ( y->left != NULL )

6. y->left->parent = x;

7. /\* y's new parent was x's parent \*/

8. y->parent = x->parent;

9. /\* Set the parent to point to y instead of x \*/

10. /\* First see whether we're at the root \*/

11. if ( x->parent == NULL )

12. T->root = y;

13. else{

14. if ( x == (x->parent)->left ){

15. /\* x was on the left of its parent \*/

16. x->parent->left = y;

17. }else{

18. /\* x must have been on the right \*/

19. x->parent->right = y;

20. }

21. }

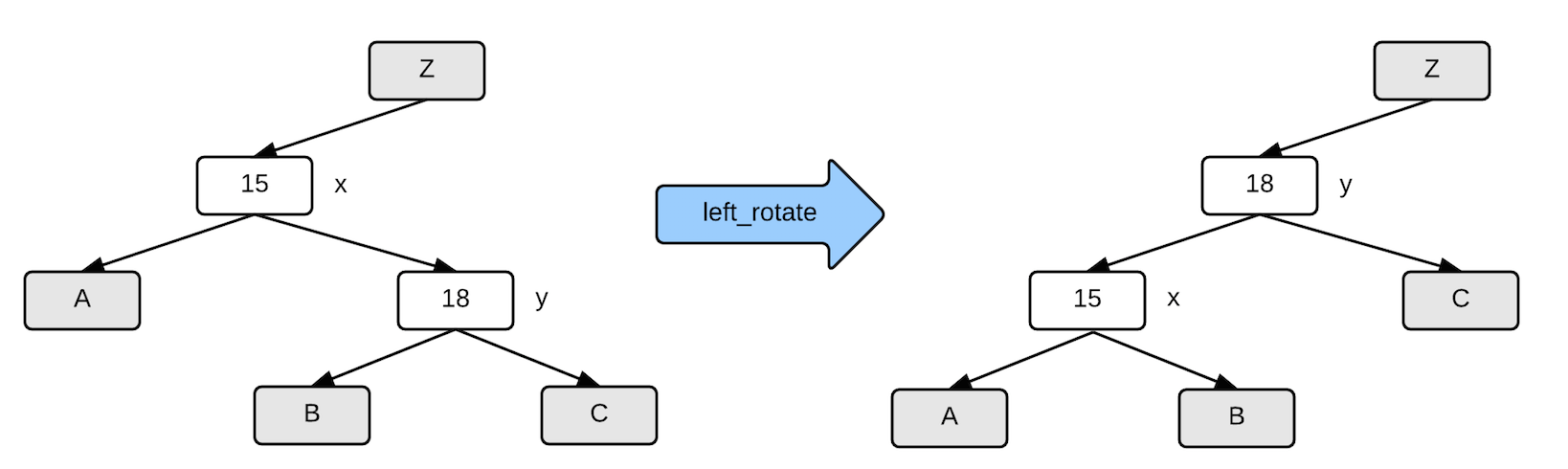
22. /\* Finally, put x on y's left \*/

23. y->left = x;

24. x->parent = y;

}

Assume we're starting with this subtree, and we want to do a left rotation to produce a new subtree.



In this tree, node x is the 15, and we set node y to be the 18 in Line 2 of the left\_rotate function.

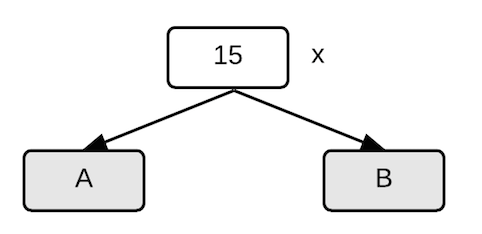
Nodes that change in the rotation:

* x.parent
* y.parent
* x.right
* y.left

Nodes that don't change in the rotation:

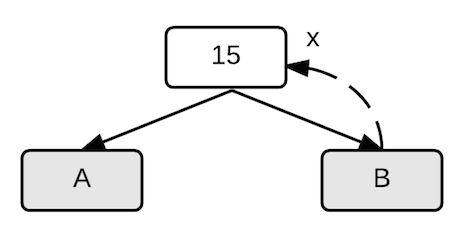
* y.right
* x.left

In Line 4, x->right = y->left yields



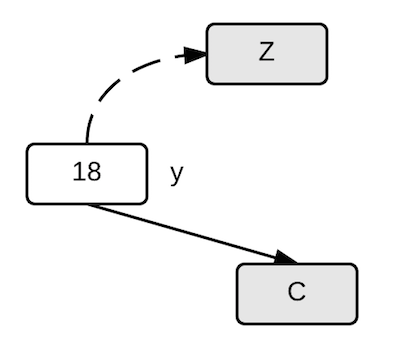
In Lines 5 and 6, we check if y has a left child, and if so, update the parent to be x.

y->left->parent = x yields



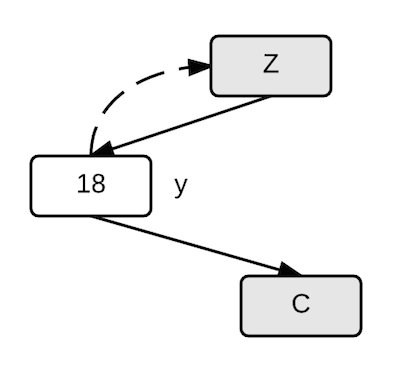
In Line 8, we update y's parent, effectively moving y into x's position in the tree.

y->parent = x->parent yields



In Lines 11-21, we're resetting the pointers for x->parent to point to y instead of x. In this example, x is it's parent's left child. We could also set up an example where x is its parent's right child.

x->parent->left = y yields

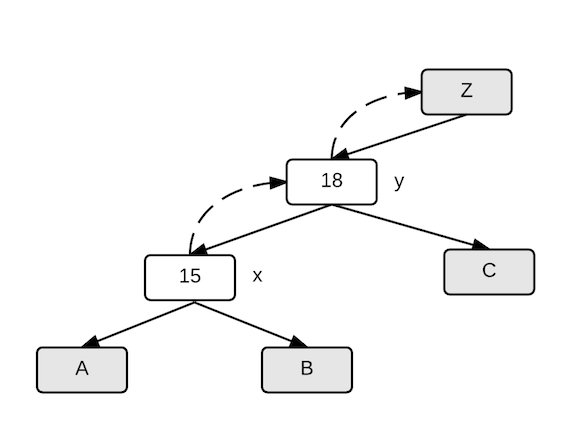


In Lines 23 and 24, we move x into position as y's left child

y->left = x

x->parent = y

yields

****

**Inserting a node into a red-black tree**

Nodes are added to red-black trees in the same way they are added to regular binary search trees. However, when a node is added, it can destroy the red-black tree properties, and there are additional steps needed to restore these properties. We start at the bottom of the tree and work our way to the top.

rb\_insert( Tree T, node x ) {

1. /\* Insert in the tree in the usual way \*/

2. tree\_insert( T, x );  
3. /\*Assume node exists in tree T called nil. It is empty node maintained for red-black structure.\*/  
4. x->left = T->nil;   
5. x->right = T->nil;

6. /\* Now restore the red-black property \*/

7. x->color = red;

8. while ( (x != T->root) && (x->parent->color == red) ) {

9. if ( x->parent == x->parent->parent->left ) {

10. /\* If x's parent is a left, y is x's right 'uncle' \*/

11. y = x->parent->parent->right;

12. if ( y->color == red ) {

13. /\* case 1 - change the colors \*/

14. x->parent->color = black;

15. y->color = black;

16. x->parent->parent->color = red;

17. /\* Move x up the tree \*/

18. x = x->parent->parent;

19. }

20. else {

21. /\* y is a black node \*/

22. if ( x == x->parent->right ) {

23. /\* and x is to the right \*/

24. /\* case 2 - move x up and rotate \*/

25. x = x->parent;

26. left\_rotate( T, x );

27. }

28. /\* case 3 \*/

29. x->parent->color = black;

30. x->parent->parent->color = red;

31. right\_rotate( T, x->parent->parent );

32. }

33. }

34. else {

35. /\* repeat the "if" part with right and left exchanged \*/

36. }

37. }

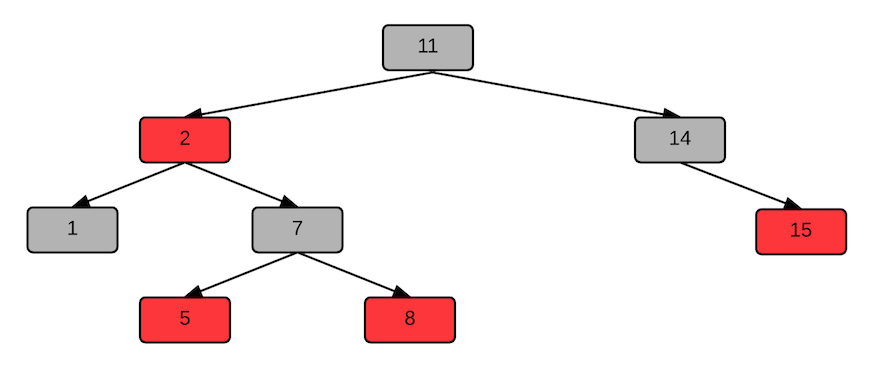
38. /\* Color the root black \*/

39. T->root->color = black;

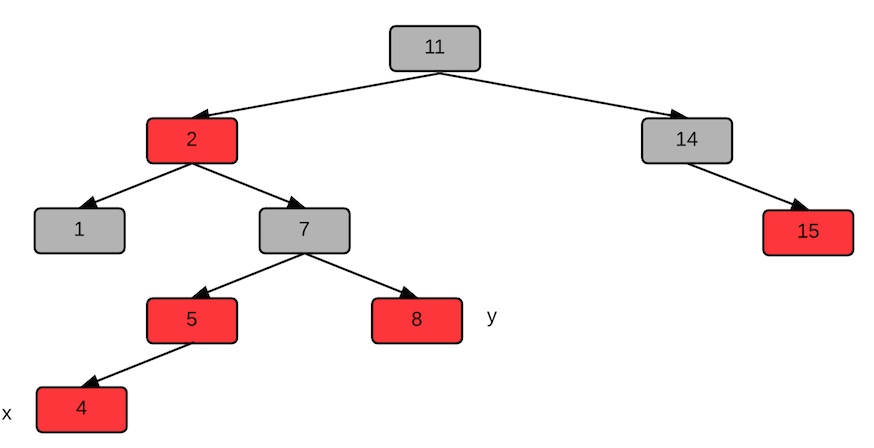
}

**Example:**

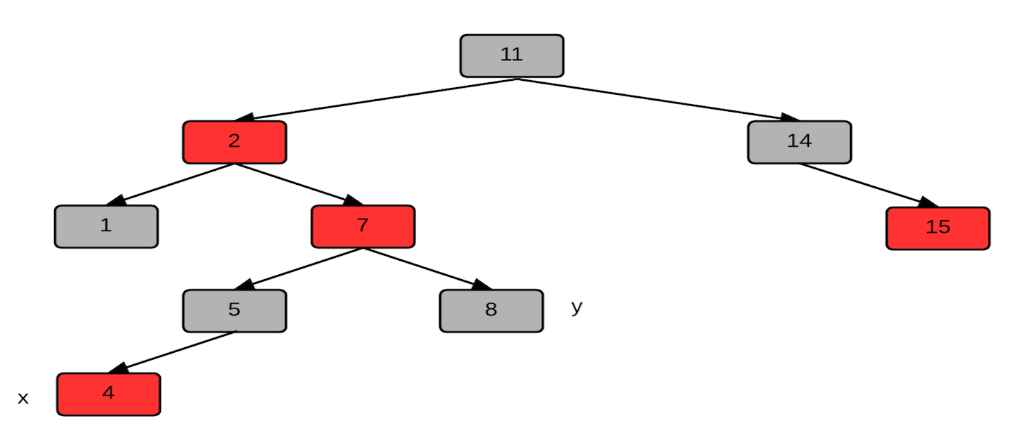
The original tree.



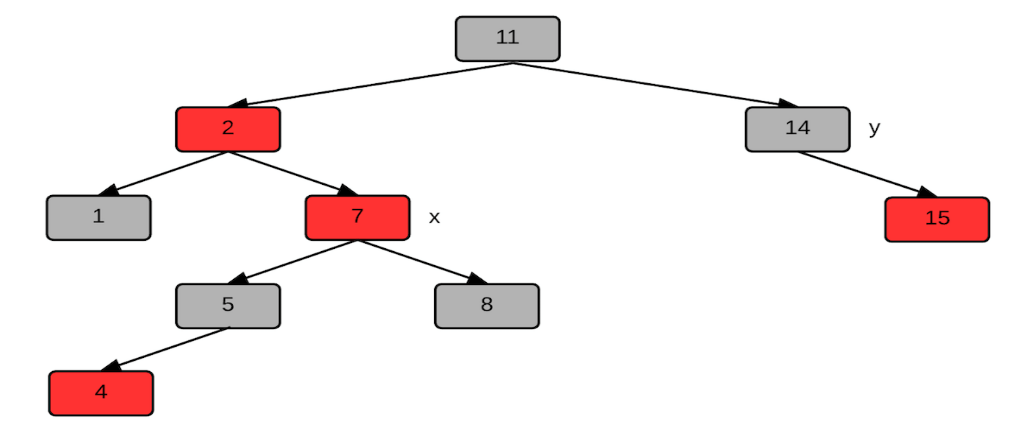
Add a 4 node. This creates a situation where the 5 node has a red child, which violates Property 4. We set this 4 node to be x, and the nodes "uncle" as y. We care about the "uncle" node because the 5 and the 8 nodes are both child nodes of the 7. Any coloring changes we make the the subtree rooted at 7 will affect both the 5 and the 8.



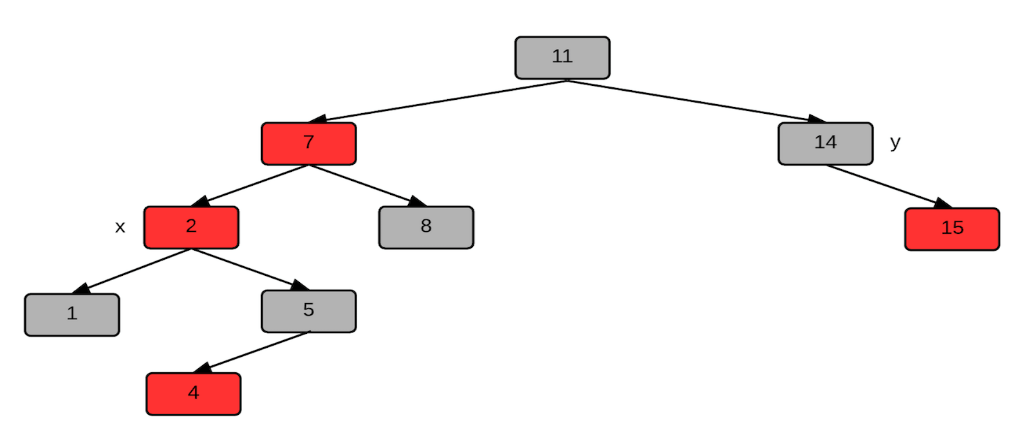
To fix the coloring, the first step is to change the coloring to remove the violation. We re-color the 5, 7, 8, which fixes the issues of the 5 being red and having a red child. We change the 7 to be black because we need the path from root to leaf to have the same number of black nodes for all paths. But, now Property 4 is violated between the 2 and the 7.



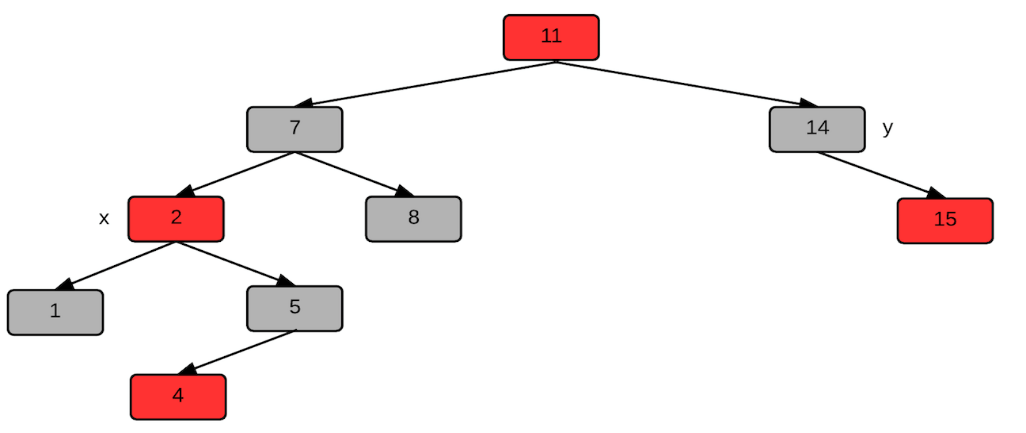
We move up in the tree to examine x->parent->parent. The nodes below this level should have the correct coloring, but we may have created a problem above the 7 by changing its color. We change x to point to the 7 node, and since we're concerned with 7's parent we also need to consider the other child at the same level as the 2. We set y to be the 14, which is 7's "uncle".



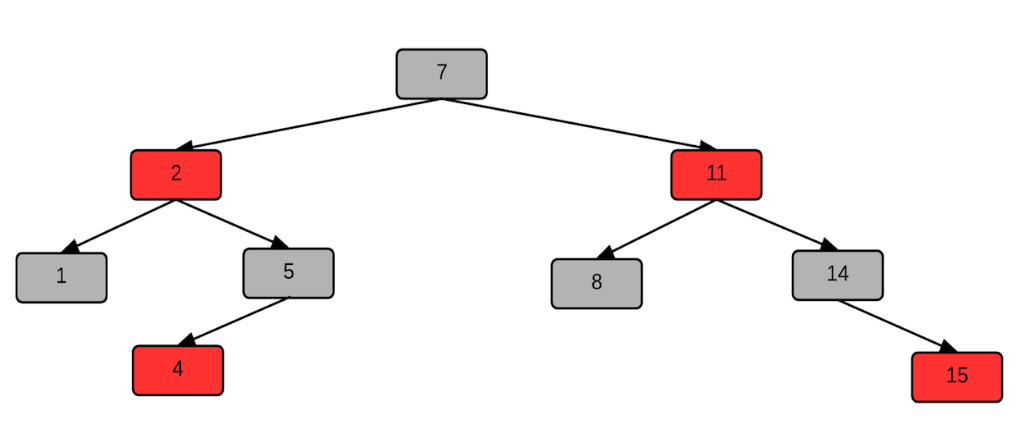
We ultimately need to do a right rotation to balance the tree. First, we need to setup the left side of the tree with a left rotation. We move x up in the tree to point to the 2 node, and apply the left rotate algorithm. The 7 node is the y in the left rotate. The resulting tree looks like:



Re-color the 7 node to be black and the root to be red. A red root violates Property 2. We do a right rotate with y = 11 and x = 7.



Perform a right rotation. The heights of both subtrees are equal and all conditions on red-black trees are satisfied.



[**Computer science course notes**](http://www.microveggies.com/csci)

# Home

## [Lecture21 - Red black trees II](http://www.microveggies.com/csci/index.php/csci-2270-lecture-notes/9-lecture21-red-black-trees-ii)

**Details**

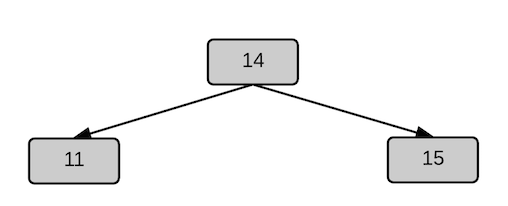
Written by Rhonda Hoenigman

 Published: 04 March 2015

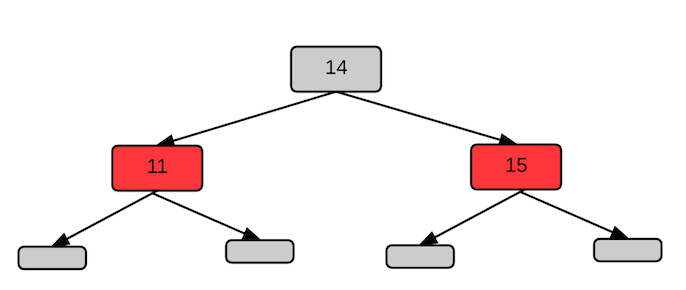
 Hits: 1974

### Red-black tree structure

In a red-black tree, the NULL pointers for child nodes are replaced by pointers to external leaf nodes. For example, with a regular binary tree, we represented the nodes like this:



The same collection of nodes on a red-black tree would be represented as:



The leaf nodes are black to satisfy Property 3 of a red-black tree.

When building a red-black tree, we use T.nil as an empty tree node for the red-black tree T. T.nil has all the same properties of a regular node and is used for maintaining the structure needed in the red-black tree.

### Insert a node

Inserting a node into a red-black tree is similar to inserting a node into a regular binary search tree. There are four differences.

**In a red-black tree:**

1. Replace all instances of NIL in the tree insert algorithm with T.nil.
2. Set the left and right children of the new node to T.nil instead of nil.
3. Set the color of the new node to red.
4. Assigning red to the new node may violate one of the red-black properties, restore the properties through tree balancing.

The C++ code to balance a red-black tree is given in the Lecture20 notes. The code to insert a node is in the binary search tree nodes. In these notes, we'll walk through building a red-black tree, starting with adding the root node. When a node is added to the tree, there are two properties that could be violated:

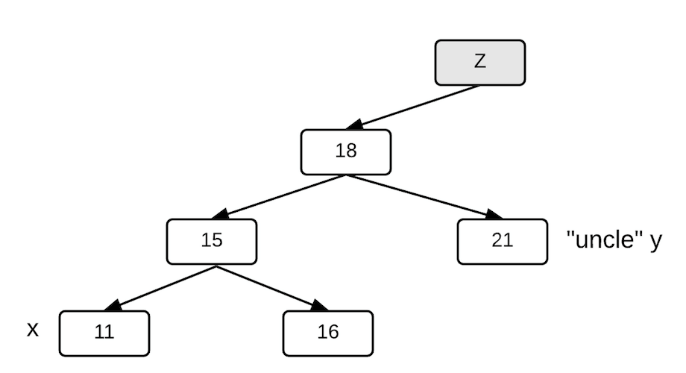
* The root must be black
* The children of a red node must be black.

Both violations are possible because a new node is initially colored red.

There are six cases to consider when inserting a node into a red-black tree, three of them are symmetric to the other three depending on whether the parent of the new node is the left or right child of its parent. We'll discuss three of the cases.

Let x be the new node added to the red-black tree T. We know that x.color = red, x.left =T.nil, and x.right = T.nil.

Let y be x's "uncle", which is the same as x.parent.parent.right.



**Case 1: x's uncle y is red.**

If x.parent.parent.right is red, then x.parent is also red. Recolor both of these nodes to be black, and recolor x.parent.parent to be black. Then, move x up the tree two levels to point to x.parent.parent the next time through the while loop.

**Case 2: x's uncle y is black and x is a right child.**

**Case 3: x's uncle y is black and x is a left child.**

In cases 2 and 3, the difference is whether x is a left or right child of its parent. If it's a right child, we can do a left rotation on x's parent, which makes x a left child. We then apply the algorithm for case 3, which recolors the parent and parent.parent nodes of x and does a right rotation about x.parent.parent.

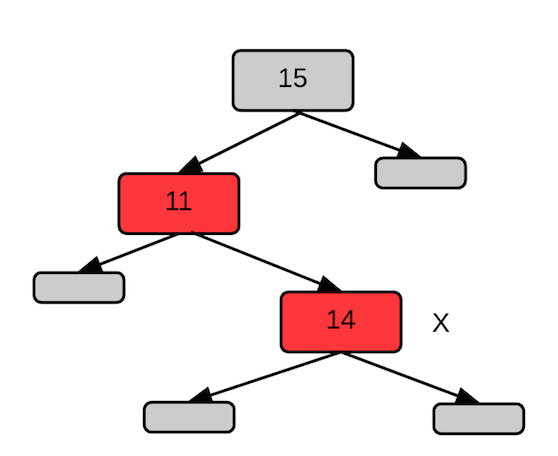
**Example: Build a red-black tree from the following sequence.**

<15, 11, 14, 2, 1>

1: Add 15, as the root node. Color it red. The while loop conditions are both false, we skip the code for Cases 1, 2, and 3. Change the color to black on the last line of the rb\_insert algorithm.

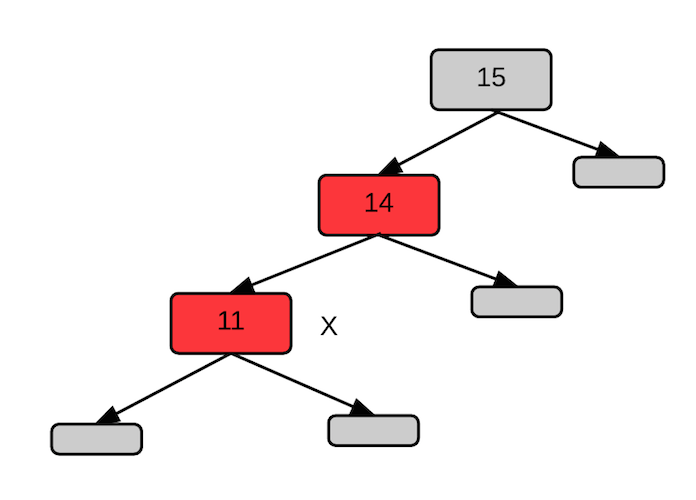
2: Add 11 as left child of 15 and color it red. The while loop conditions are both false.

3. Add 14 as right child of the 11, which violates the property that a red node can't have a red child. The new node is labeled x.

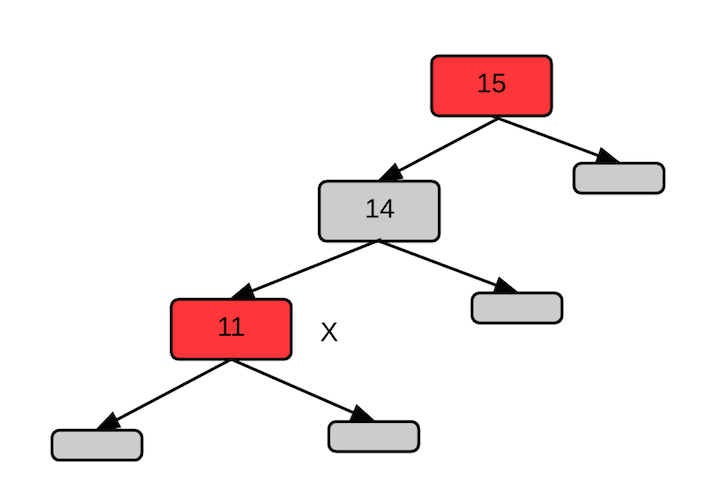


The T.nil child nodes are shown explicitly.

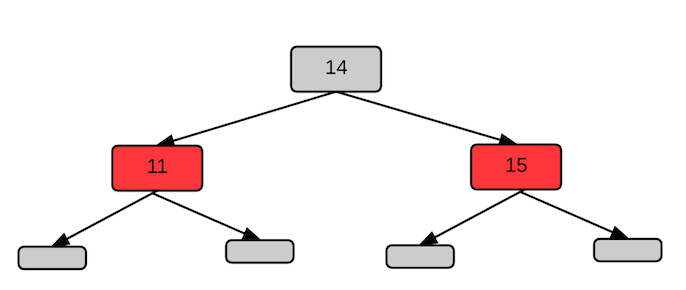
In the while loop, x.parent.color is red, and x's parent, the 11, is the left child of its parent. We check the color of the "uncle" node, which is black because it's T.nil. This is Case 2 because x is a right child. Do a left rotation about x's parent to get:



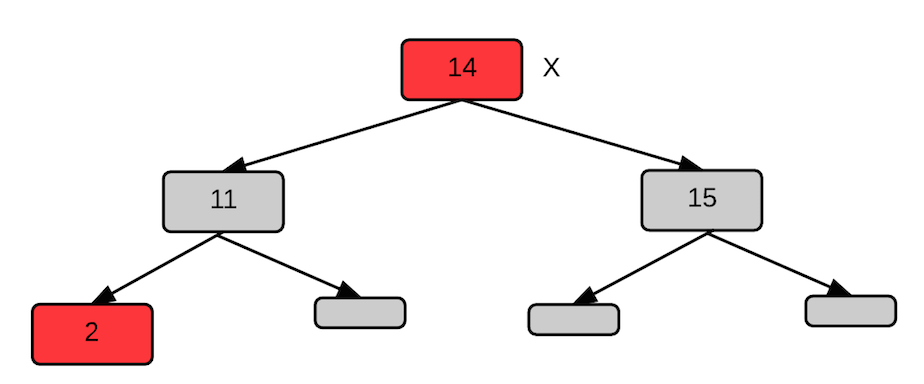
Next, apply Case 3 code and recolor the parent and parent.parent nodes.



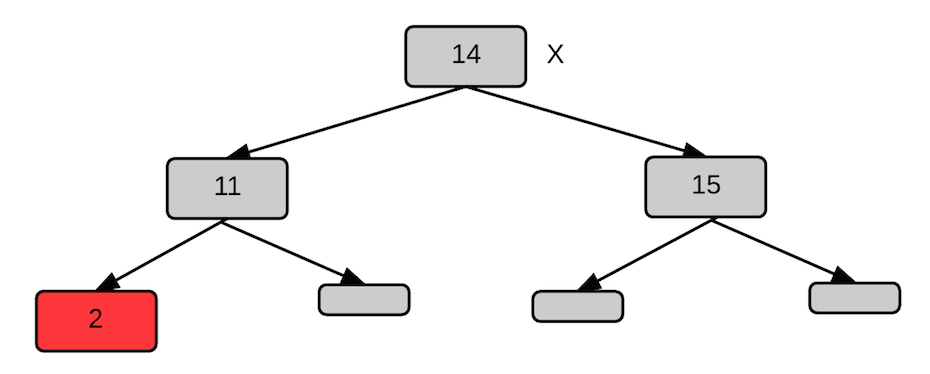
Do a right rotation to get:



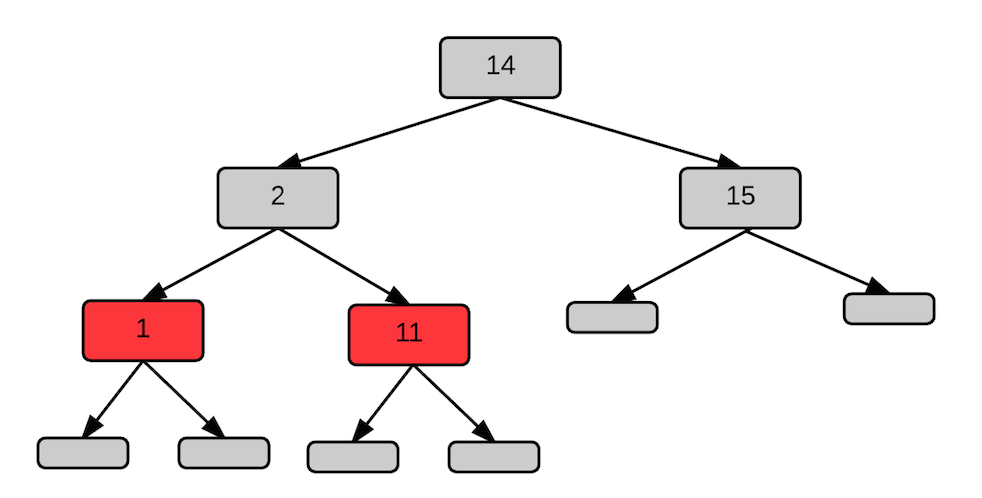
4: Add 2 as the left child of the 11. This is Case 1. Recolor the nodes and set x = x.parent.parent. At the next iteration of the while loop, the tree looks like:



Both conditions on the while loop are false, so the next line to execute changes the color of the root node to black and the tree looks like:



5: Add 1 as the left child of the 2, which gives us a red node with a red parent. The uncle node is the right child of the 11, which is black. This is Case 3. Change the colors for the x.parent and x.parent.parent nodes and right rotate on x.parent.parent to get:



### Deleting a node in a red-black tree

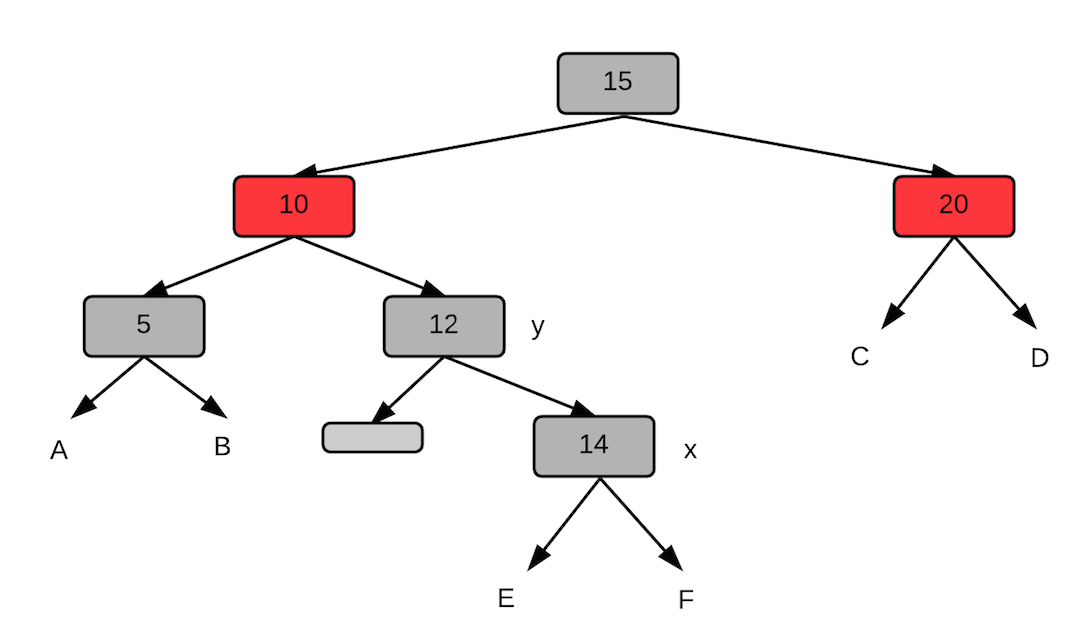
We use the same algorithm for deleting a node from a red-black tree as we use for a regular binary search tree with the additional requirement of managing the color of the node deleted and restoring the red-black properties of the tree after the node is deleted.

**For a binary search tree delete:**

* If the node has no children - delete the node.
* If the node has one child - replace the node with its remaining child.
* If the node has two children - replace the node with the minimum node in its right branch.

When a node is deleted from a red-black tree, violations of the red-black properties can be introduced if the node's replacement y was black.

**Example: delete the 10**



In this example, the deleted 10 will be replaced by the node y, which is black. The node x will be y's replacement. In doing so, we've reduced the number of black nodes on the path, and we will need to call a rebalancing algorithm on the node x.

**Red-black tree delete:**

* Maintain an additional pointer y as the node removed or moved in delete process.
  + When z has fewer than 2 children, y = z.
  + When z has two children, y is z's successor.
* Maintain an additional variable that stores y's original color.
  + When y is z's successor, and y.color ≠ z.color, the replacement may introduce violations of the red-black tree properties if y.color = black.
  + If y is red, the red-black properties hold because:
    - The number of black nodes on any path hasn't changed.
    - No red nodes are adjacent.
    - The root is still black.
* Maintain an additional pointer x as the node that replaces y.

When node y was black, there are three problems that can cause violations in the red-black properties.

* If y had been the root and a red child of y becomes the root, then Property 2 is violated. (The root of the tree must be black.)
* If x and x.parent are red, then Property 4 is violated. (Children of red nodes must be black.)
* Moving y within the tree causes any path that previously contained y to have one fewer black nodes.

[**Computer science course notes**](http://www.microveggies.com/csci)

# Home

## [Lecture22 - Red black trees III](http://www.microveggies.com/csci/index.php/csci-2270-lecture-notes/10-lecture22-red-black-trees-iii)

**Details**

Written by Rhonda Hoenigman

 Published: 06 March 2015

 Hits: 1616

### Deleting a node

Deleting a node from a red-black tree is similar to deleting a node from a regular binary search tree. The following pseudocode from pages 323 - 326 from your textbook shows the conditions that need to be considered.

The red-black transplant algorithm is used to replace a node with another specified node. There are three cases to consider: the node being replaced is the root, a left child, or a right child.

rb\_transplant(T, u, v) //u is the original node, v is the replacement

  if u.parent == T.nil

     T.root = v

  elseif u == u.parent.left

     u.parent.left = v

  else

     u.parent.right = v

  v.parent = u.parent

The delete algorithm is similar to the binary search tree delete algorithm, with a few additions to track the color properties of the replacement node, and the replacement-replacement node.

rb\_delete(T, z) //z is the node to delete

1.  y = z

2.  y-original-color = y.color

3.  if z.left == T.nil

4.       x = z.right

5.       rb\_transplant(T, z, z.right)

6.  elseif z.right == T.nil

7.       x = z.left

8.       rb\_transplant(T, z, z.left)

9.  else

10.     y = tree\_minimum(z.right)

11.     y-original-color = y.color

12.     x = y.right

13.     if y.p == z

14.        x.parent = y

15.     else

16.        rb\_transplant(T, y, y.right)

17.        y.right = z.right

18.        y.right.parent = y

19.     rb\_transplant(T, z, y)

20.     y.left = z.left

21.     y.left.parent = y

22.     y.color = z.color

23. if y-original-color == BLACK

24.    rb\_delete\_fixup(T, x)

The fixup routine to restore the red-black properties is called on the x node, which is y's replacement. This routine is called when the color of y, which is the minimum value in z's right branch, is black.

rb\_delete\_fixup(T, x)

1.   while x != T.root and x.color == BLACK

2.      if x == x.parent.left

3.             w = x.parent.right

4.             if w.color == RED

5.                w.color = BLACK

6.                x.parent.color = RED

7.                left\_rotate(T, x.parent)

8.                w = x.parent.right

9.             if w.left.color == BLACK and w.right.color == BLACK

10.               w.color = RED

11.               x = x.parent

12.           else  
13. if w.right.color == BLACK

14.               w.left.color = BLACK

15.               w.color = RED

16.               right\_rotate(T, w)

17.               w = x.parent.right

18.           w.color = x.parent.color

19.           x.parent.color = BLACK

20.           w.right.color = BLACK

21.           left\_rotate(T, x.p)

22.           x = T.root

23.     else (same as the then clause with right and left exchanged)

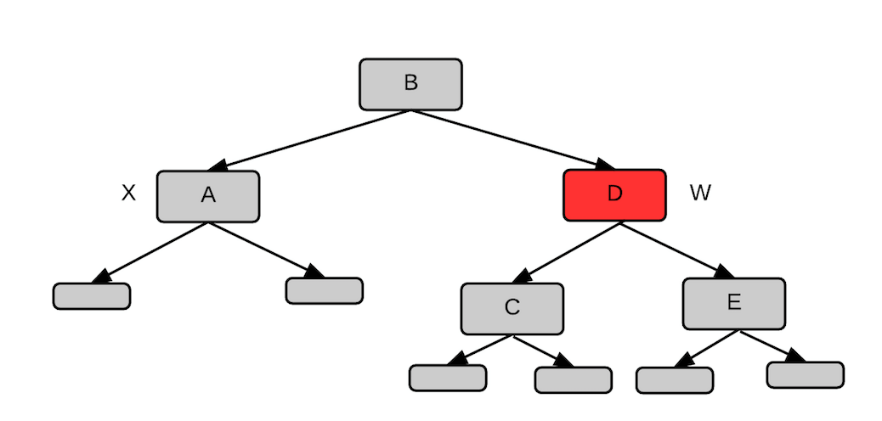
24.  x.color = BLACK

In the rb-delete-fixup routine there are four cases to consider when restoring the red-black properties, and they about the color of x's sibling.

(Examples from page 329 in Cormen.)

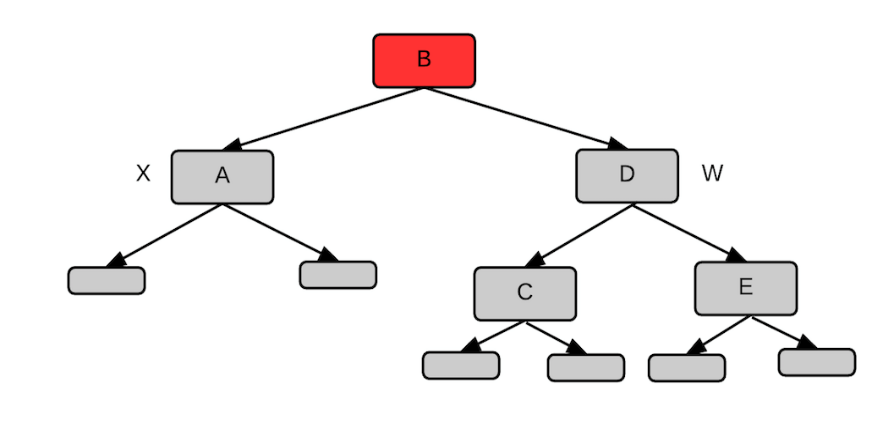
**Case 1: x's sibling w is red.**

**Example:**

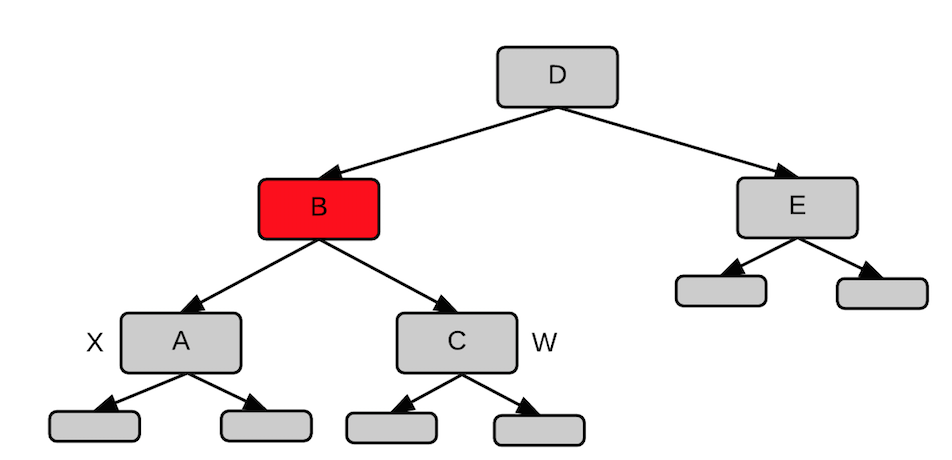


In this example, the child nodes of A, C, and E are all T.nil. However, they could also be subtrees.

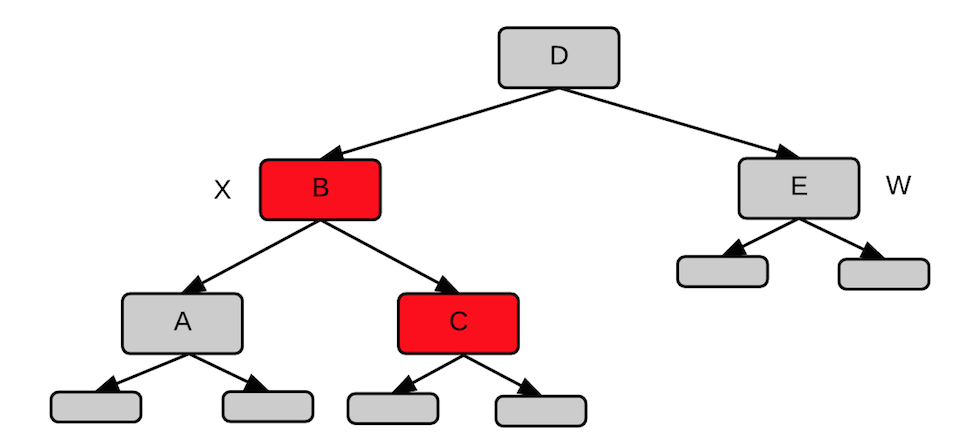
On line 5-6 of rb\_delete\_fixup, the color of W and its parent are switched to get:



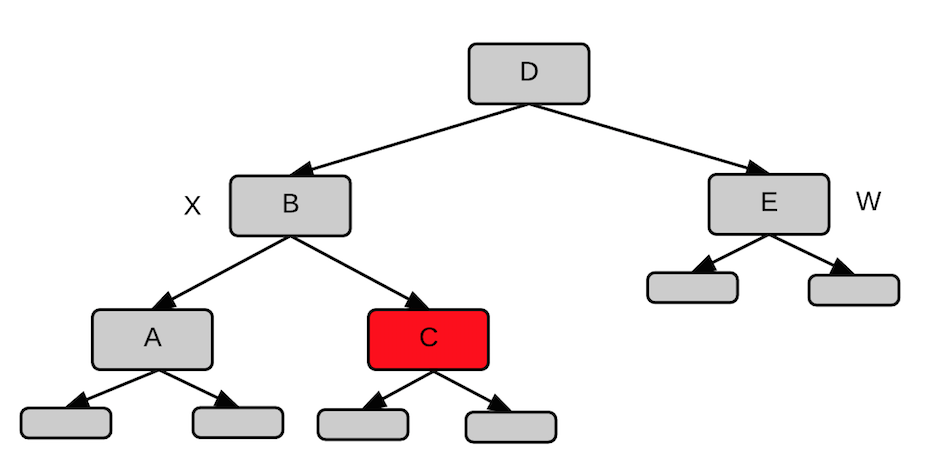
Next, a left rotation and reassigning of W yields:



The previous step is the end of the Case 1 code. Next, x is set to x.parent and w is recolored to yield:

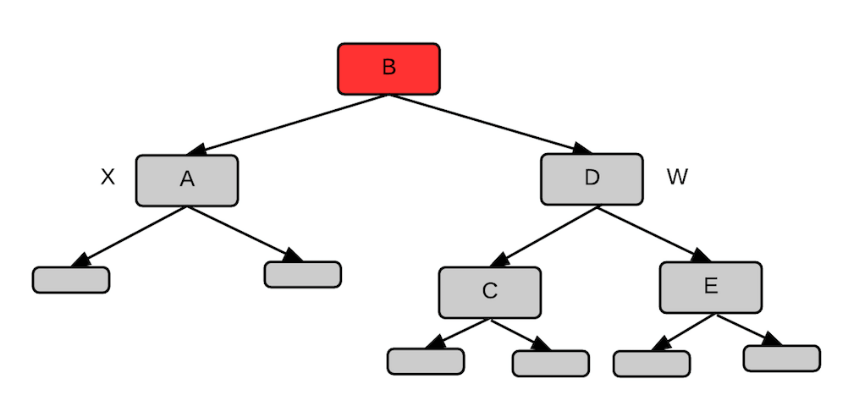


We start another iteration of the while loop, and since x is not black, we skip the while and recolor x to yield:

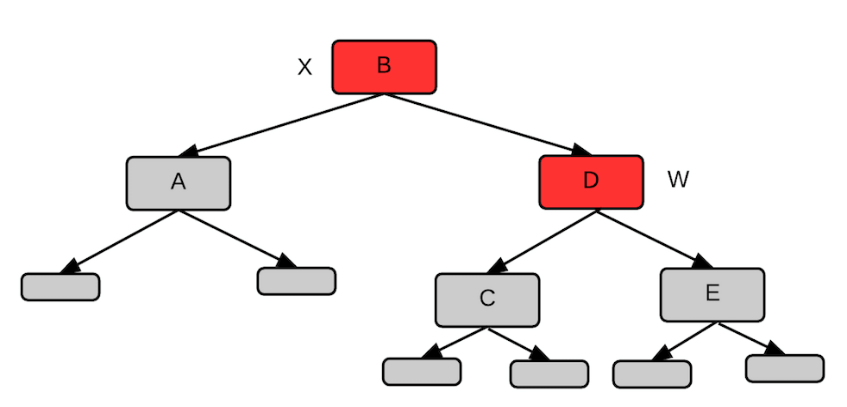


**Case 2: x's sibling w is black, and both of w's children are black.**

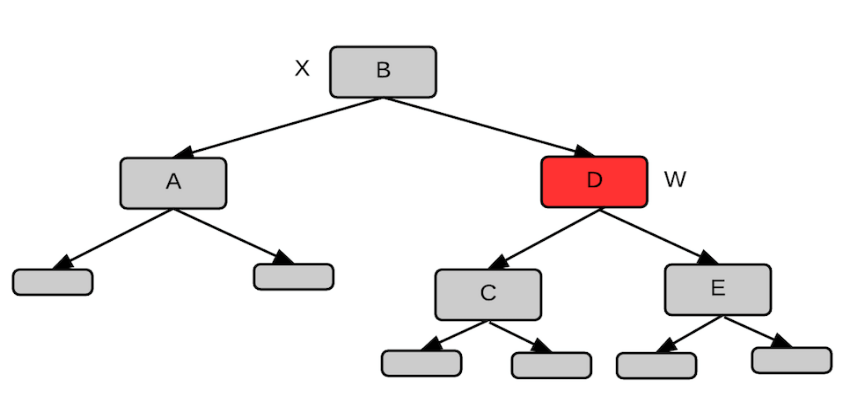
**Example:**



We recolor w and w's parent and then set x to x.parent to get:

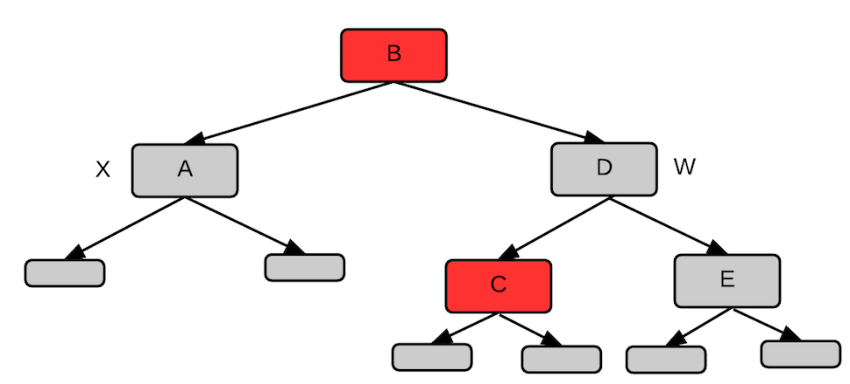


We exit the while loop since x is red, and then recolor x to get:

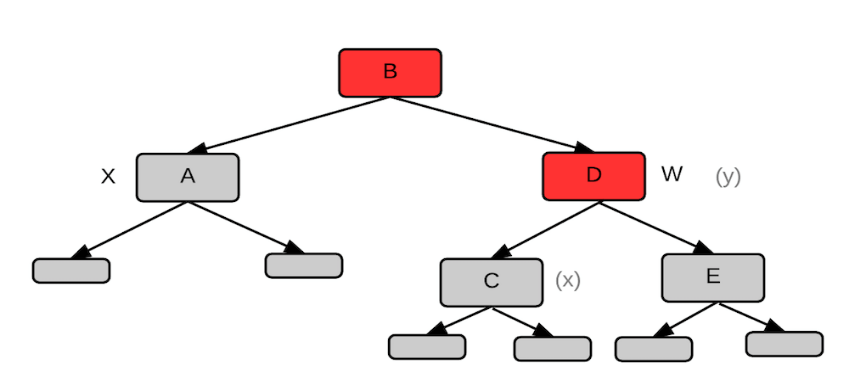


**Case 3: x's sibling w is black, w's left child is red, and w's right child is black.**

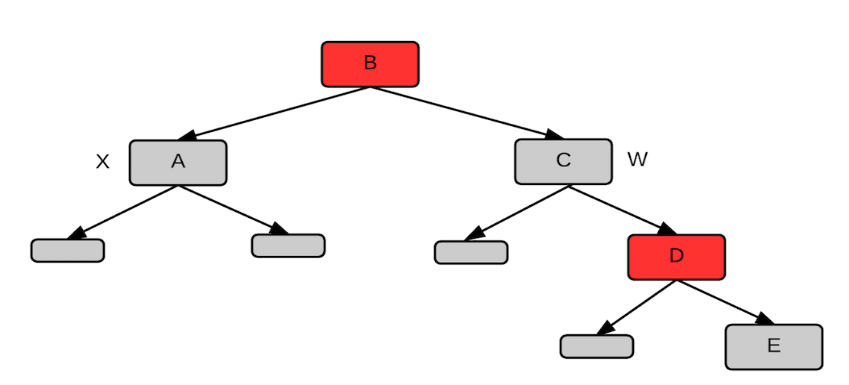
**Example:**



Recolor w and its left child to get:



Next, do a right rotation on w:

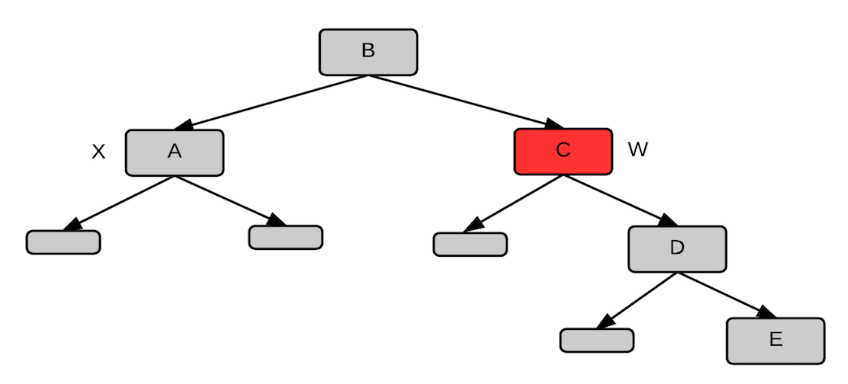


This is the end of the step for Case 3.

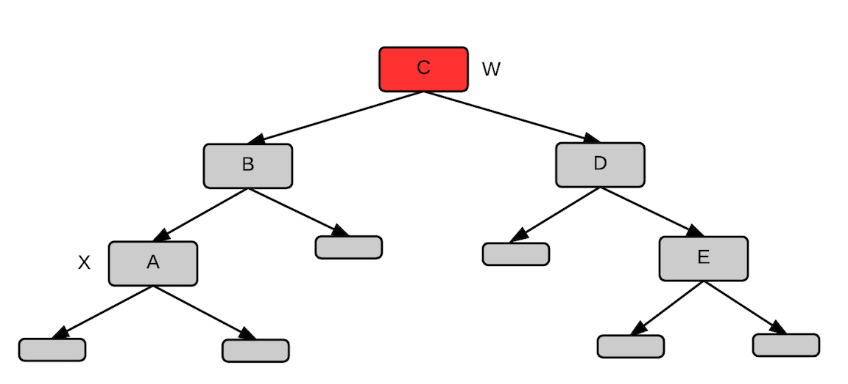
**The tree is now an example of Case 4: x's sibling w is black, w's left child is black and w's right child is red.**

We came to Case 4 through Case 3. We wouldn't see this configuration with the difference in the left and right subtrees otherwise.

Recolor w, its parent, and its right child to get:



Do a left rotation about x.parent to get:



Next, x is set to T.root to cause the loop to exit. We can leave w red because it's not the root of the tree, just the root of the subtree shown.

© 2015 Computer science course notes