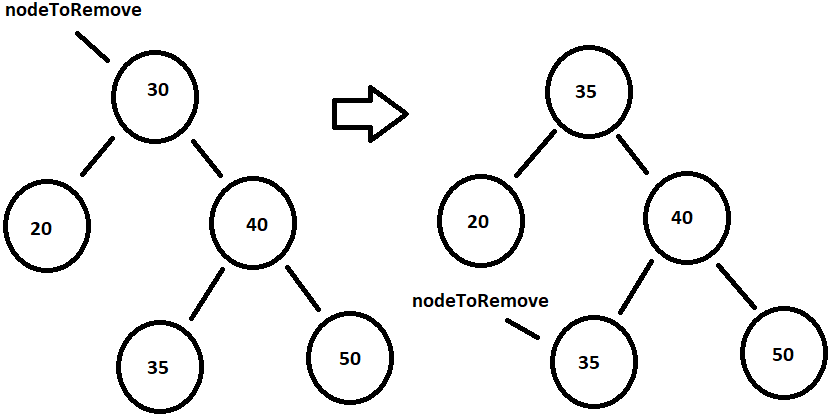
## Red Black Trees Deletion

The deletion of nodes in the red-black tree starts with identifying if the node is of degree 0, degree 1, or degree 2. No matter what the case maybe, the programmer should always consider the edge case of the root being deleted and adjust their code accordingly.

## Deleting nodes of degree 2

If the node is of degree 2, then the programmer should swap the node’s value with that of its inorder successor, like it would with a naïve binary search tree. Then the programmer would consider removing the new node from its location of the tree. Here is an example of a situation when a node is of degree 2 and the switch that will occur.

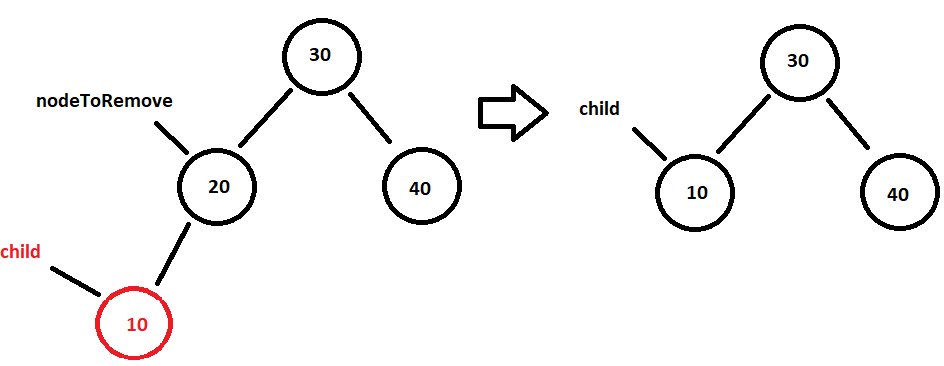


When the node is swapped with its inorder successor, the programmer knows that the new node is either a node of degree 0 or a node of degree 1 and can then apply the rules of those nodes to its deletion.

When deleting a node, the node to be deleted is often indicated as the letter **v** and the node that will be replacing it is often indicated by the letter **u**. However, in these nodes, the deleting node will be called nodeToRemove and the replacing node will be called child.

## Deleting when u or v is red

When the nodeToRemove, v, or the child of the nodeToRemove, u, is red then the deletion is straight forward. The child of the nodeToRemove will take the place of the nodeToRemove and will become black.



The programmer could write a method in the RedBlackTree. The method would receive the node to remove and make sure that its child, regardless of whether it is the left or right child, is removed from the node. The child, which is also sent up through the method, would then make sure it becomes a black node then would be returned through the method so that it can be attached to the appropriate node.

private RedBlackNode simpleCaseDeletion(RedBlackNode nodeToRemove,

RedBlackNode child)

{

nodeToRemove.setLeft(null);

nodeToRemove.setRight(null);

if(child != null)

child.setColor(1);

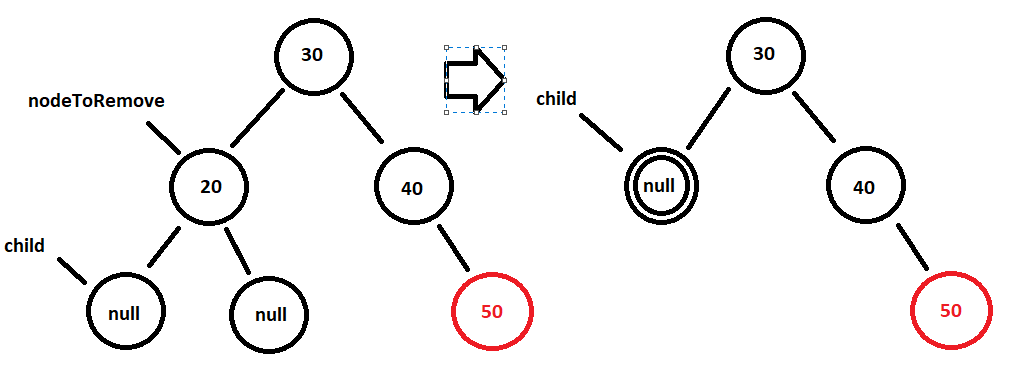
return child;

}

In the case of removing a red leaf, all the programmer has to do is to make sure the parent that is pointing to the red leaf instead points to a null.

## Deleting when u and v are black

Deleting a nodeToRemove and its child which are both black involves the introduction of the double black node. The double black node occurs when one branch is heavier than the other branch. The best way to consider a double black node in relationship to the other two nodes is that of a closed set or {red, black, double black} or {0, 1, 2}. A null is considered a black node, although it is not counted as part of the black height. When the nodeToRemove, v, is black and has no children, then it is considered replaced by the null children which are both black nodes creating a double black node as seen below:

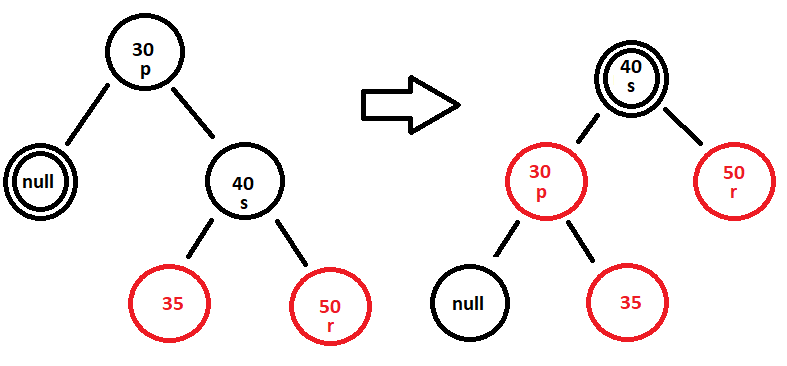


The red black tree height is no longer distributed evenly. The left branch is of black height 0 and the right branch is of black height one. A restructuring needs to take place so that it is evenly distributed. To evenly distribute the remaining values, the programmer must look at the relationship of the double node to its sibling and sibling’s children.

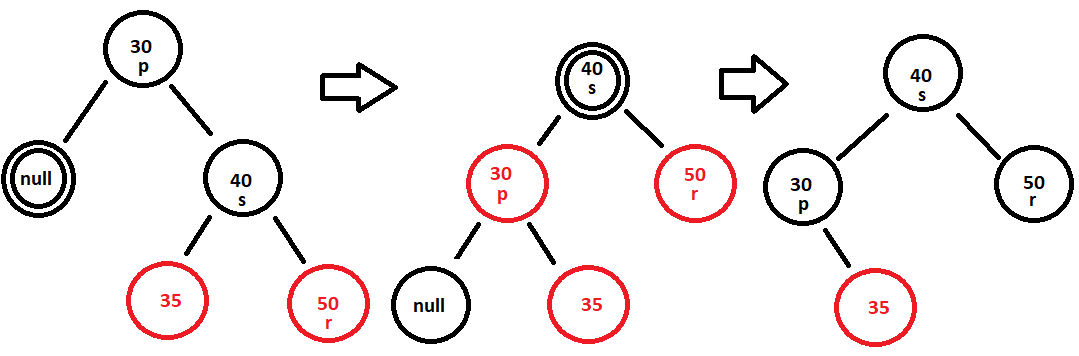
## Restructuring with a black sibling that has at least one red child

If the sibling of the double black node is black and one of the sibling’s children are red, then the programmer would want to do a restructuring of the subtree. The goal is to rotate the subtree such that the sibling becomes the new parent and the parent replaces double node. Just like there are two types of rotations, there are two types of restructuring: outside-outside restructuring and outside-inside restructuring.

The example above shows the need for an outside-outside restructuring, specifically a right-right restructuring. The programmer identifies the parent node, the sibling node, and red node. The parent node will set its right child to the sibling’s left child. The sibling will make its left child the parent. The parent would become red because it is moving down a level, subtracting 1 from its color value (black – 1 = red). For that same reason, the sibling would become a double black because moving up a level would add 1 to its color value (black + 1 = double black). The double black null node would become a normal null. The red node would maintain its color.



After this rotation is complete, the sibling will distribute its double black node to its children, effectively moving one of its black node counts onto both of the children. The sibling will become black (double black – 1 = black) while the children will also become black (red + 1 = black). The subtree is now balanced with a black height of 2.



A method written for this right-right restructure would use the right-right rotation method which was covered with insertion into a red black tree. The restructure method would receive a RedBlackNode reference to the parent, the sibling, and the sibling’s red child node. The programmer would then call the right-right rotation method. Once the call to the rotation is completed, the programmer would recolor the parent, sibling, and red node. This can be done by calling the colorSwap method and sending up the new sub-tree root, s. It is possible that the double black node that caused the need for the restructuring was not a null. If that was the case, the double black node should be reduced to a black node. Because that double black node was not sent to the method as a RedBlackNode, the RedBlackTree would see the node as a BinaryNode since the RedBlackTree is using inheritance to create itself. The programmer would have to cast the double black node as a RedBlackNode before attempting to use the RedBlackNode color methods. When the programmer is finished with the restructuring, the programmer would return the root of the subtree that was restructured, which is now the former sibling node still pointed to by the reference s. Here is what the code could look like:

private RedBlackNode rightRightRestructure(RedBlackNode p,

RedBlackNode s, RedBlackNode r)

{

rightRightRotation(p, s);

colorSwap(s);

if(p.left()!=null)

{

RedBlackNode temp = (RedBlackNode)(p.left());

temp.setColor(temp.getColor()-1);

}

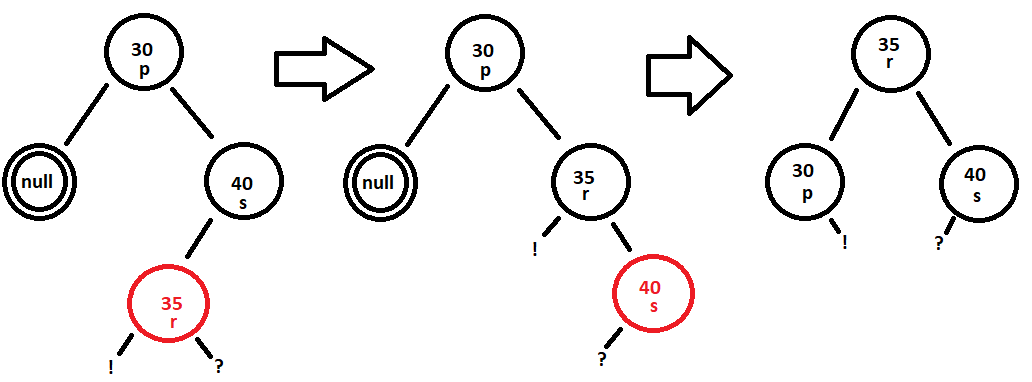
return s;

}

The left-left restructuring would be similar to the right-right restructuring.

The programmer now considers the outside-inside restructuring. The programmer would want to make the rotate the sibling and red node such that it would become an outside-outside restructuring and then applies that to the restructuring. During an insertion, when a rotation was required in the outside-inside rotation, the parent and x node did not change colors because they were both red nodes and the rearrangement was made so that the outside-outside rotation would fix the red rule. However, in this restructuring, the red rule is not broken, so the programmer could treat this rotation as an outside-outside rotation with the sibling being the grandparent and the red node being the parent. This will be fine because an outside-outside rotation only changes the color of the parent and the grandparent.

Once the rotation of the sibling and the red node is complete, the programmer can apply a call to the outside – outside restructure where the red node is treated as the sibling and the sibling is treated as the red node. The figure below shows a right-left restructuring. The first subtree goes from the original subtree to the rotated sibling and red node. The resulting subtree then uses an outside-outside restructuring to complete the full restructuring.



The method written for the outside-inside restructure is deceivingly simple as it calls on two previously created methods. The programmer would make sure the parent node’s right child would connect to the new root of the subtree that is created when a leftLeftRotation is called using the sibling node and the red child node. When this method is done, the subtree that has the parent as its root is now an outside-outside restructuring subtree, specifically right-right. The programmer would return the results of a call to the right-right restructuring method. The method is seen below

private RedBlackNode rightLeftRestructure(RedBlackNode p,

RedBlackNode s, RedBlackNode r)

{

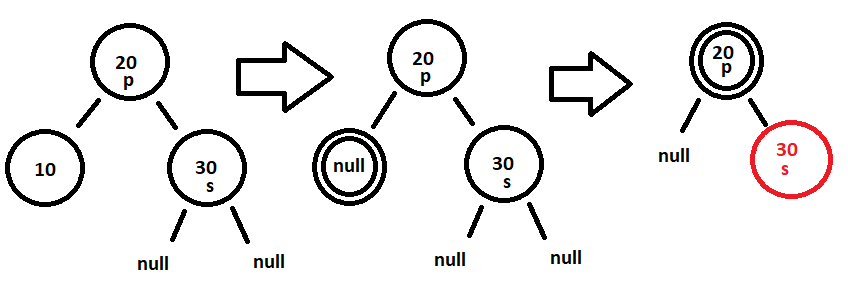
p.setRight(leftLeftRotation(s, r));

return rightRightRestructure(p, r, s);

}

## Recoloring when the sibling and its children are black OR the parent is red.

If the sibling of the double black node does not have any red children, then the parent and the children must be recolored. This will push the double black up to the parent and take the coloring of the children down by one, effectively propagating the problem up the tree. The parent would be taking on an addition node count, causing the two children to each lose a node count. This means that the parent would become a double black (black + 1 = double black), the double black null node would become a black null node (double black – 1 = black) and the sibling will become a red node (black - 1 = red). When this recoloring is done, the programmer will have to go up the tree to correct the double black problem. This is known as propagating.

The method of the recoloring would receive the parent node. The parent node’s color would increase by a level. The method would then grab the left and right child of the parent. The children would be cast into a RedBlackNode and then, as long as the node exist, the node’s color would decrease by a level. Here is a sample of what the recoloring method could look like

private void recolor(RedBlackNode p)

{

p.setColor(p.getColor()+1);

RedBlackNode temp = (RedBlackNode)(p.right());

if(temp!=null)

temp.setColor(temp.getColor()-1);

temp = (RedBlackNode)(p.left());

if(temp!=null)

temp.setColor(temp.getColor()-1);

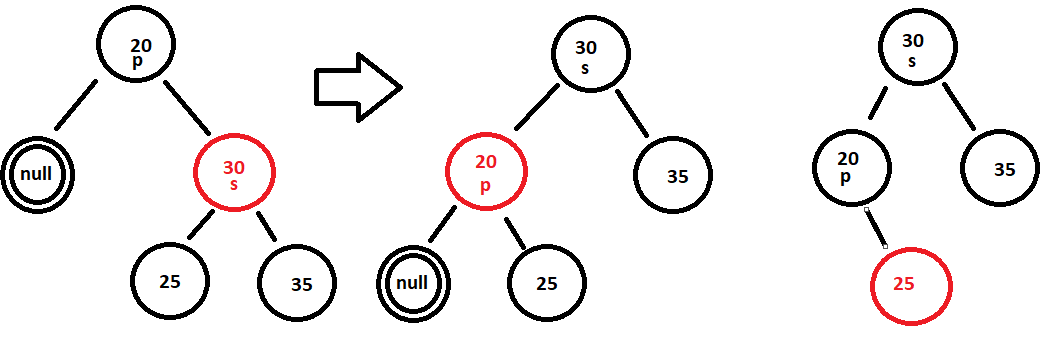
}

When this recoloring is done, the program would have to restructure/recolor subtree that has the parent as its left or right branch.

If the recoloring occurred because the parent was red (this is a special case), then check for a rotation for the parent’s children.

## Adjusting when sibling is a red node

The final consideration concerning a double black node occurs if the sibling is red. When that occurs, the programmer would do an adjustment which is an outside-outside rotation followed by a recoloring on the parent node. It is important to note that the outside-outside is a rotation and not a restructuring, this will ensure that the double black node continues to be a double black node.



The method would be called an adjustment left method because it is adjusting towards the left of the tree. When the method is completed, the sibling would be returned because it is the new root of the subtree. Here is what the method could look like.

private RedBlackNode adjustmentLeft(RedBlackNode p, RedBlackNode s)

{

rightRightRotation(p, s);

recolor(p);

return s;

}

## The rare case when the parent and the sibling children are red but the sibling is red

Occasionally, the programmer will come across a binary tree in which the parent and the siblings children are red, but the sibling is a black node. This can lead to confusion, so we must write a special case method. The special case would receive the parent and the sibling. The sibling would to a color swap, then the parent and sibling would rotate, much like it did in an outside-inside rotation, causing the sibling to become the new subroot. Finally, the parent would do a recoloring. The method then returns the sibling.

Shape

Description automatically generated

The above image shows this special case. Here is what the program would look like, the following is written if the left node is being deleted and everything is shifting to the left. The would need to be another method for the right node being deleted:

**private** RedBlackNode specialCaseLeft(RedBlackNode p, RedBlackNode s)

{

colorSwap(s);

p.setRight(s.left());

s.setLeft(p);

recolor(p);

**return** s;

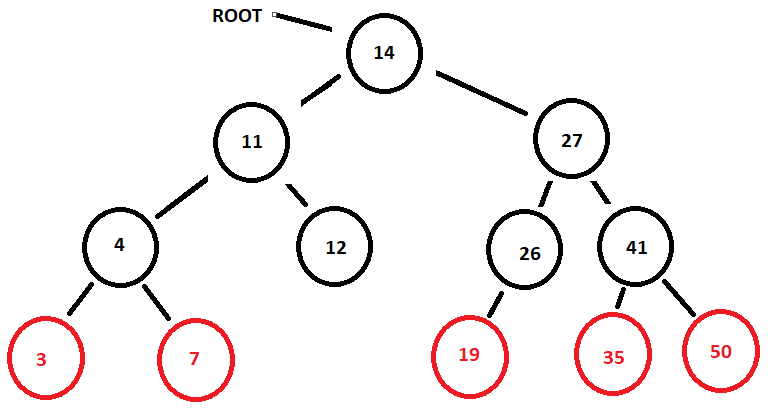
}

## The root as a double black node

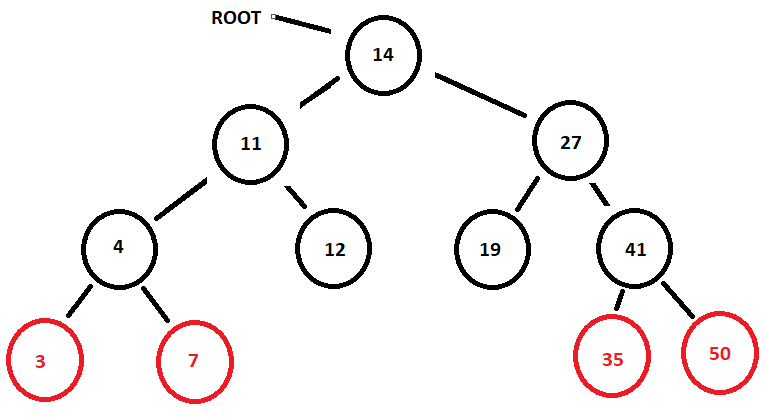
There will come a time when the root is a double black node, which is not allowed. The root is always a black node. When the root becomes a double black node, it means that the black height of the tree has just reduced by one because of the deleted node. The programmer would change the double black root node into a black root node.

## Example of deletion from a red-black tree

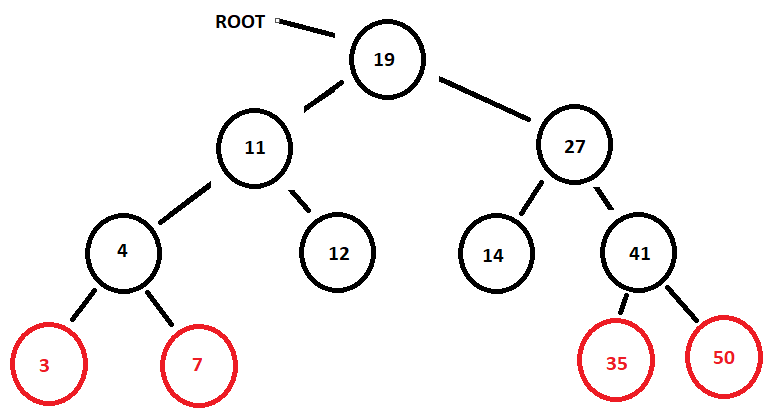
Consider the previous notes red black tree of 7,27,11,3,14,12,26,41,19,35,4,50.



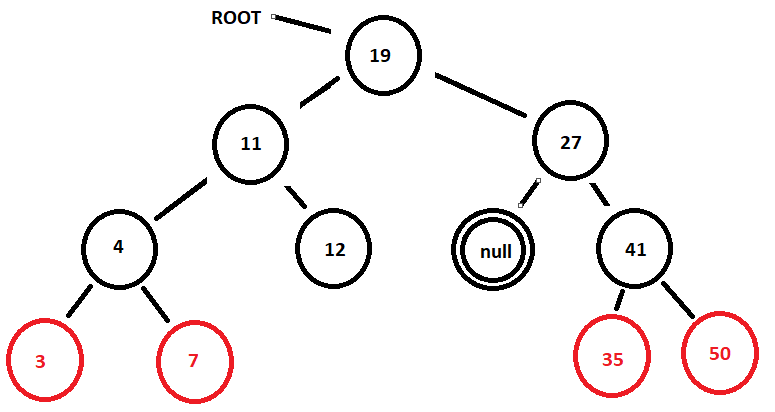
Deleting the value 26 will be a simple case deletion. The 26 node will be removed and the 19 node would move up and, in so doing, become a black node.



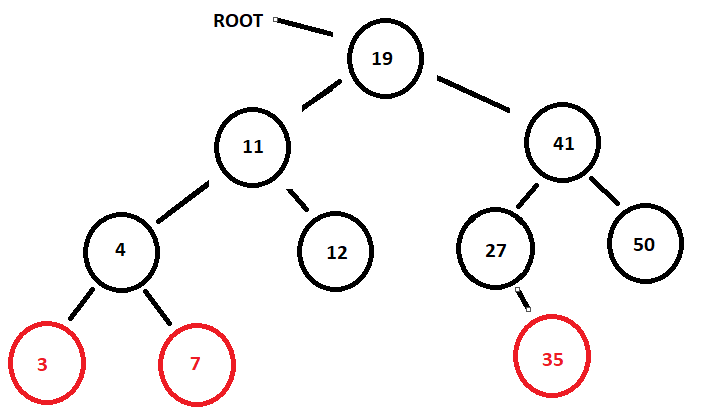
When deleting the 14 node, the programmer would swap with its inorder successor node 19.



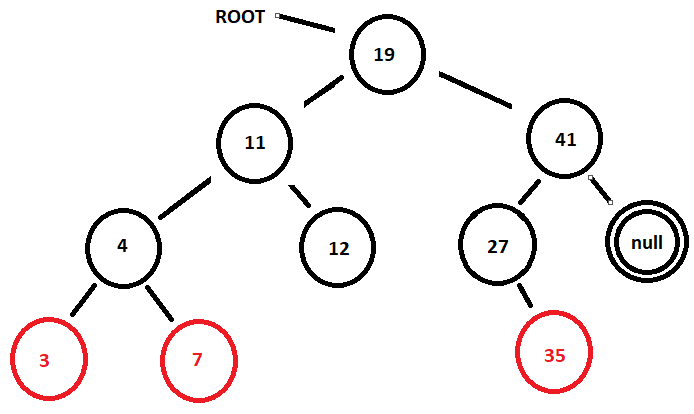
The programmer would then delete the 14 node causing the two children null nodes to become a double black null node



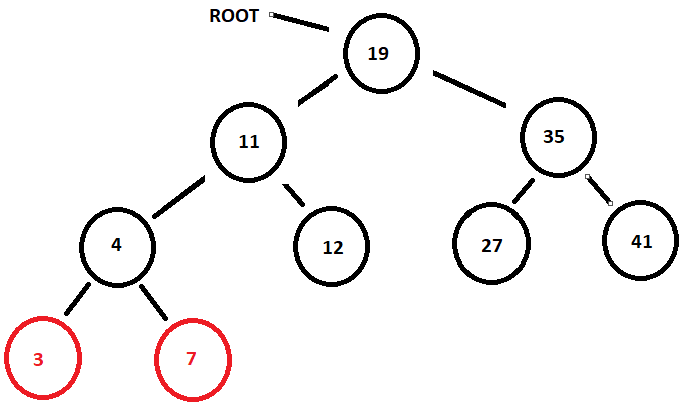
The sibling of the double black node null is black with two red children so a right – right restructuring is performed.



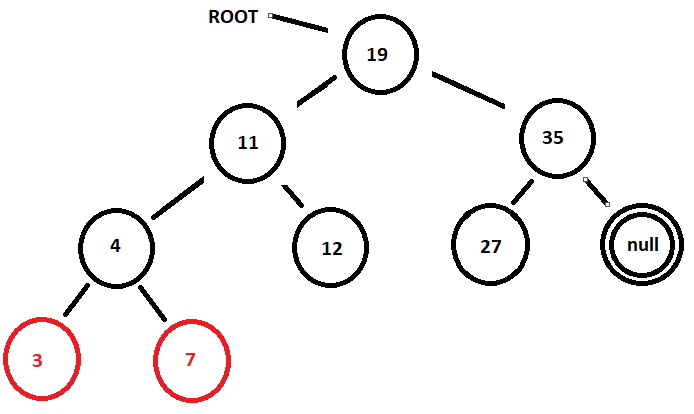
Deleting the 50 black leaf node would lead to a double black node as the right child of the 41 node.



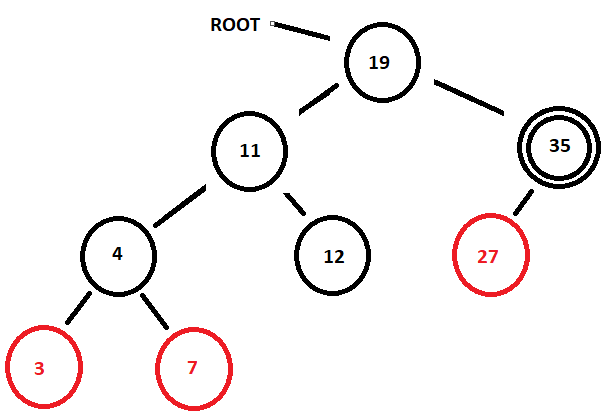
The sibling of the double black node is black with an interior red child. This will require an outside-inside restructuring and will create the following result.



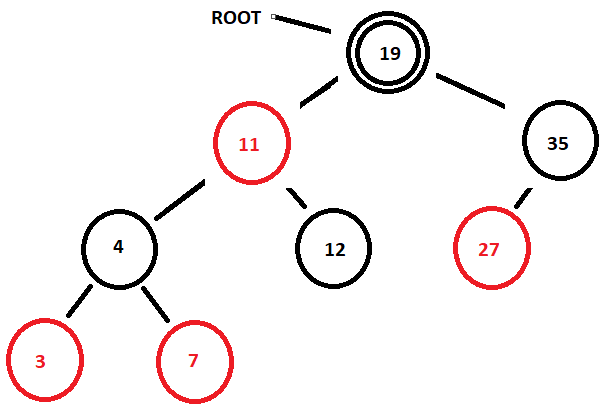
Deleting the 41 node will create a double black node as the right child of the 35 node.



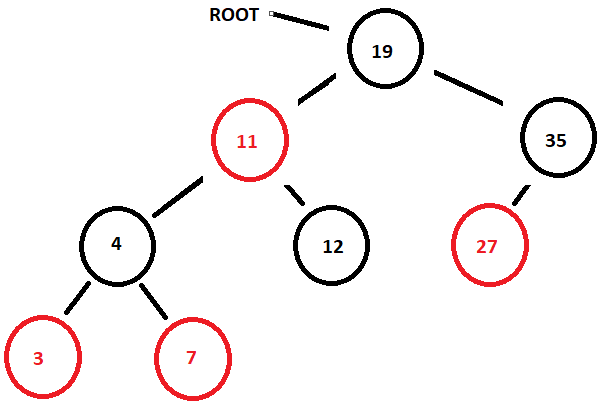
The sibling of the double black node is a leaf and is considered to have two black node children. A recoloring needs to be performed on the 35 node which will cause the 35 node to become a double black node.



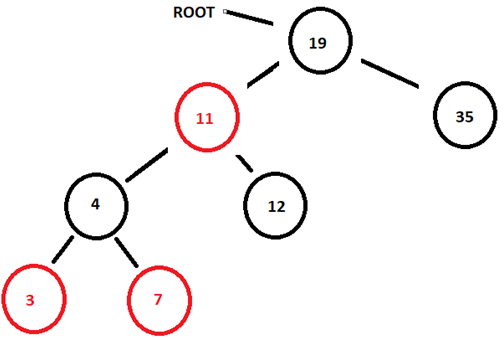
This double black node’s sibling also has two black children, so another recoloring is performed on the root.



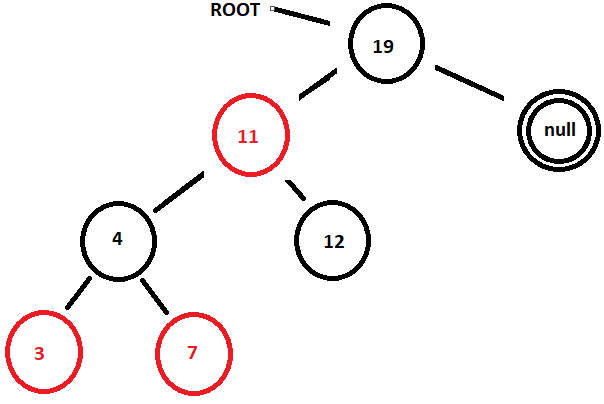
The root cannot be a double black node and is converted back to a black node. The black height has been reduced by one because of the deletion of the 41 node.



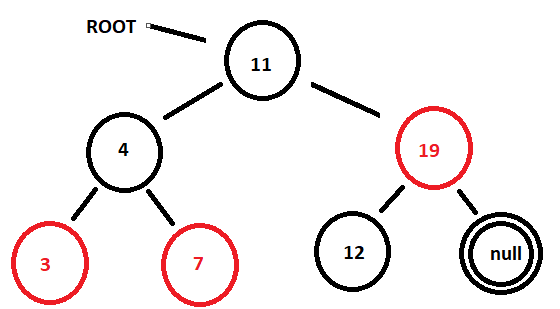
The 27 node is a red node, so deleting the value 27 is done by just setting the 35 node’s left child to null.



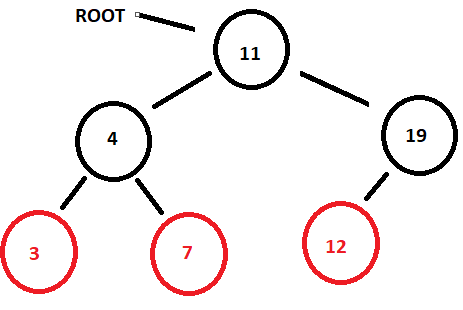
Deleting the 35 node will cause the root’s right child to be pointing to a double black node



The sibling of the double black node is a red sibling which requires an adjustment to the right. The programmer would first do a left-left rotation.



Then a recoloring of the 19 node subtree will occur.



Deleting from a red-black tree can be followed if the user remembers when to use a restructuring, a recoloration, and an adjustment.