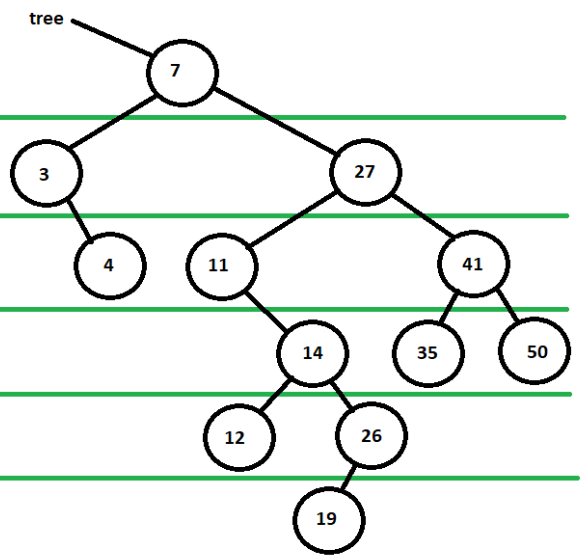
# Red-Black Binary Search Trees

A binary search tree has a tendency to become unbalanced, especially if the root of the tree is an extreme value of the set of values being set in the tree. Consider the following



The right branch is far heavier than the left branch. This is because the 7 node is an extreme value of the set. The 27 node sub-tree also shows an unbalanced tree because the left branch is deeper than the right branch. This will lead to a less than ideal time complexity. Therefore, a balanced search tree was created. A balanced search tree rebalances the tree every time a node is added to a tree, ensuring that the time complexity stays as close to O(log N) as it can. One of these balanced search trees is called a **red-black tree**.

## Properties of a Red-Black Tree

There are four properties that guide the creation and upkeep of a red-black tree:

1. Every node is either a red or a black node.
2. The root, and consequently all null nodes, are considered a black node.
3. The **red rule** states that if a node is red then neither its parent nor its children are red.
4. the **path rule** states that all paths from the root to a termination leaf must have the same number of black nodes in it. This is known as the **black height** of the red-black tree, not to be confused with the height of the tree.

## The color Attribute

A red-black binary node class is similar to the binary node class, except that it also holds an integer value which indicates if it is red or black. In fact, the best way to create a red-black binary node is to inherit the binary node and create the color attribute along with all the necessary changes. If the color is zero, then it is considered red, otherwise it is black. The programmer chooses the color to be an integer rather than a Boolean because it gives the programmer more flexibility when it comes to deleting nodes from red-black trees. Here is a partial declaration of a red-black node:

public class RedBlackNode extends BinaryNode{

private int myColor;

public RedBlackNode(Comparable x)

{

super(x);

myColor = 0;

}

public String toString()

{

String temp = super.toString();

temp+=", Color:"+(myColor==0?"Red":"Black");

return temp;

}

//other methods are not shown

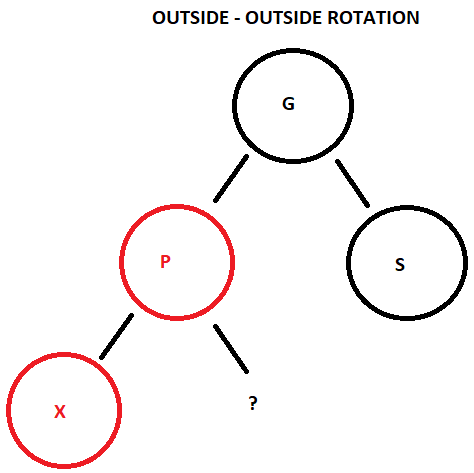
}

## Insertion into a Red-Black Tree

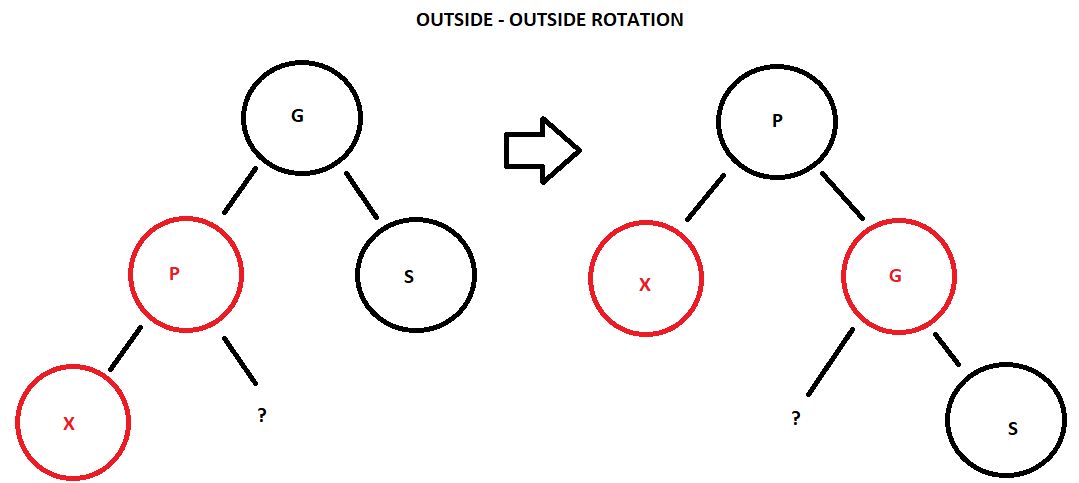
Insertion into a red-black tree must adhere to the rules of the red-black tree. A new node is always added in as a leaf. This means it can’t be black because it will violate the path rule. The new node must come in as a red leaf. This new node will follow the path down to its location and attach to the appropriate child of the parent. If that parent is black, then the insertion is done and the tree is balanced. If there is no parent, then the new node is the root and is immediately switched to black because the root is always black. If the parent is red, then there will be some rotations to make in order to balance the tree.

## Case 1: Outside-Outside Rotation

We will consider four nodes inside of a bigger tree: The node labeled G will be the grandparent, the node labeled P will be the parent, the node labeled S will be the sibling, and the node labeled X will be the new leaf node. Consider the following image



This example is not the complete red-black tree, but rather a small portion of it. The programmer is just interested in making a rotation to these nodes. A **rotation** is the swapping of roles between a parent and a child node. The first case is considered when the sibling of the parent is black and the new leaf is on the outside edge of the subtree. This is called outside-outside. During an **outside-outside rotation**, specifically the left-left rotation, the parent node will swap with the grandparent node, and make the grandparent node into its right child. The grandparent would assign its left child pointer to the parent’s right child. Finally, the parent and grandparent would recolor.



A left-left rotation has the parent move up a level and to the right, whereas, a right-right rotation would have the parent move up a level and to the left. Here is the code that could be used for a left-left rotation

public class RedBlackTree extends BinarySearchTree{

private RedBlackNode leftLeftRotation(RedBlackNode g,

RedBlackNode p)

{

g.setLeft(p.right());

p.setRight(g);

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

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

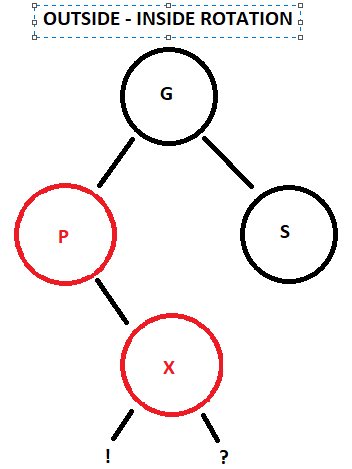
return p;

}

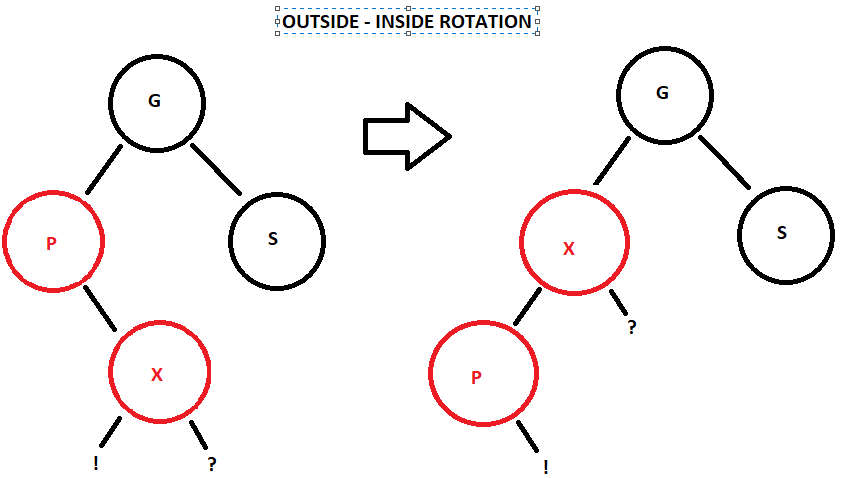
}

## Case 2: Outside-Inside Rotation

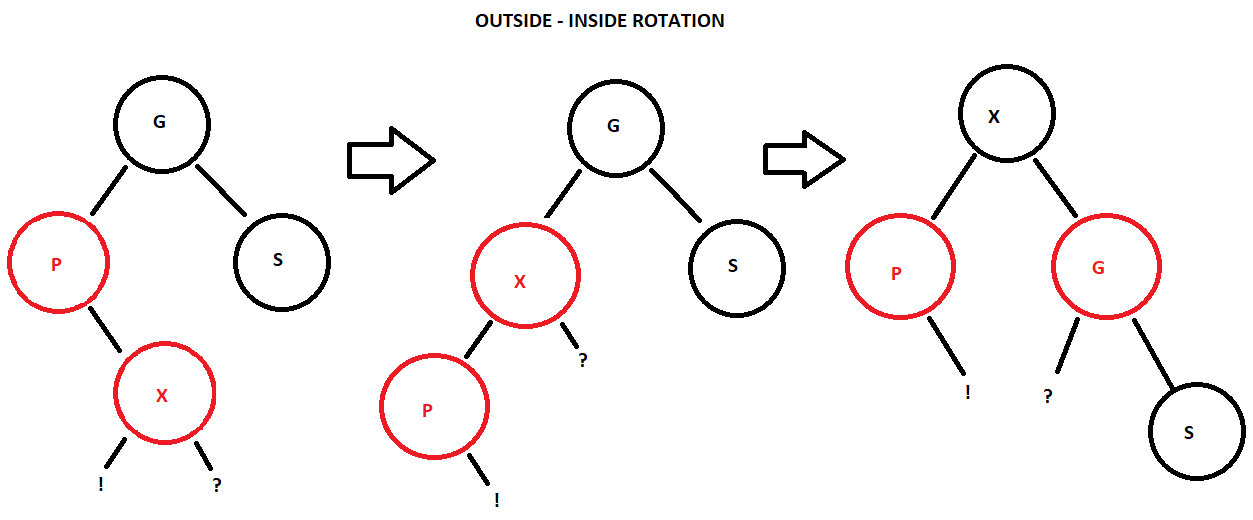
The next type of rotation occurs when the sibling of the parent is black and the new leaf is on the interior edge of the subtree. This is called outside-inside. The image below is an example of an outside-inside, specifically left-right.



Two rotations are called for in this case. First, the inside leaf will rotate with the parent without any color change. The leaf’s left pointer will point at the parent. When the programmer is dealing with a rotation where X is not a leaf, then the programmer needs to consider the left child of the X node, as seen by the exclamation point. The parent’s right pointer will point to the left child of the X node.



After this, the rotation becomes an outside-outside rotation. However, instead of the parent swapping with the grandparent, it is the leaf, or X node, that swaps with the grandparent.



This was a left-right rotation. The right-left rotation would behave the exact same way except going it the reverse direction. Here is the code that could be used for a left-right rotation

private RedBlackNode leftRightRotation(RedBlackNode g, RedBlackNode p,

RedBlackNode x)

{

p.setRight(x.left());

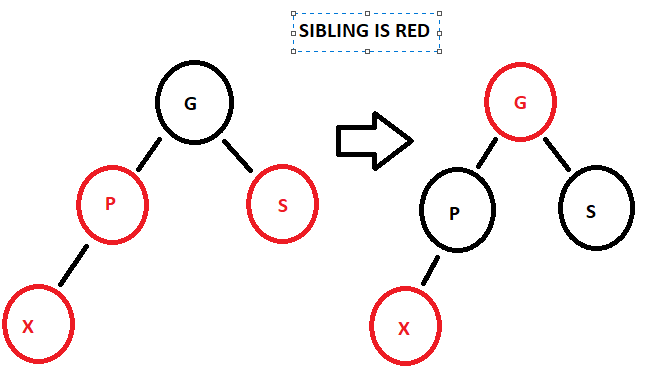
x.setLeft(p);

return leftLeftRotation(g, x);

}

## Case 3: Red Sibling

The third type of rotation is when the sibling of the parent is red. This means that the parent is red, the sibling is red, and the leaf is red. Best solution is to change the grandparent to red and change both the parent and the sibling to black.



However, if the grandparent is also red, then the programmer would have to look at the grandparent’s parent. This can lead to some convoluted rotations. As a solution, on the path down to the leaf, if programmer encounter’s a black parent with two red children, then recolor all three nodes. The programmer needs to check if a rotation needs to be applied to the grandparent and it’s own parent after this color swap occurs. Because of this swap, the programmer should always check to see if the root has become red, and if it has, then change the root back to black. Here is the code that could be use for a parent/children color swap:

private void colorSwap(RedBlackNode x)

{

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

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

temp.setColor(temp.setColor()+1);

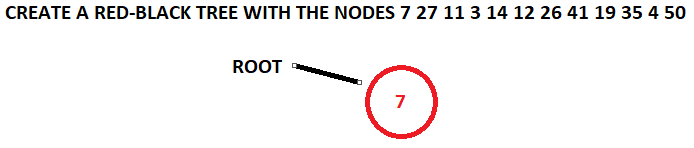
temp = (RedBlackNode)(x.right());

temp.setColor(temp.setColor()+1);

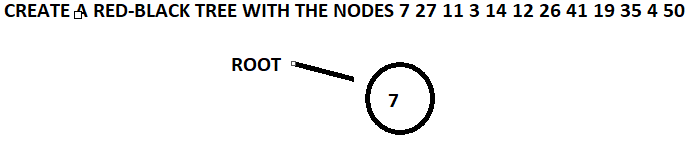
}

## Example of Inserting a List of Numbers into a Red Black Tree

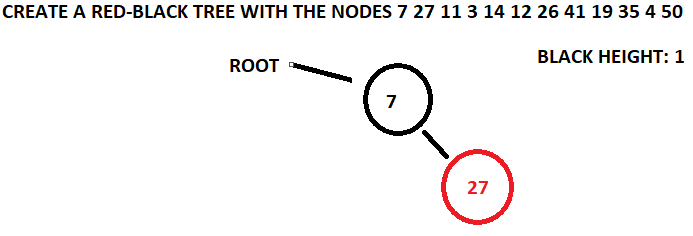
Consider the following list of numbers to be placed into a red black tree: 7, 27, 11, 3, 14, 12, 26, 41, 19, 35, 4, 50. The first value to go into the tree as a red node is the value 7. It is placed into the tree as the root.



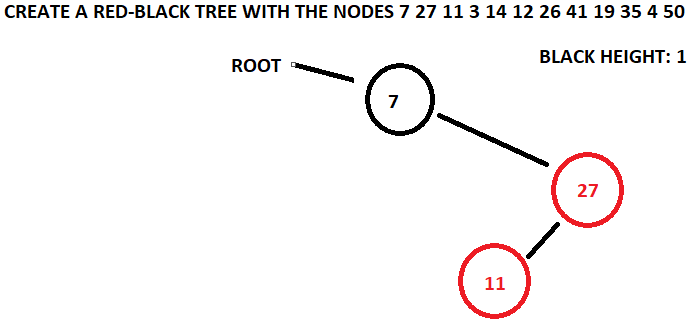
The root should always be black, so the 7 node is changed to black and the black height of the tree becomes 1.



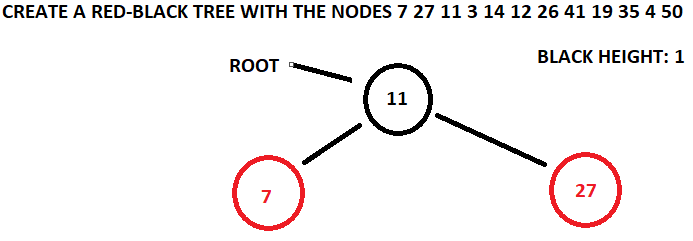
The next value to be placed into the tree is the value 27, which is brought in as a red leaf and placed as the right child of the root.



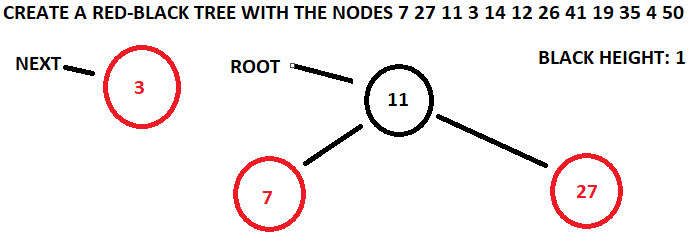
The value 11 is created as a red node and is placed into the tree. It goes to the right child of the 7 node and then is placed as the left child of the 27 node.



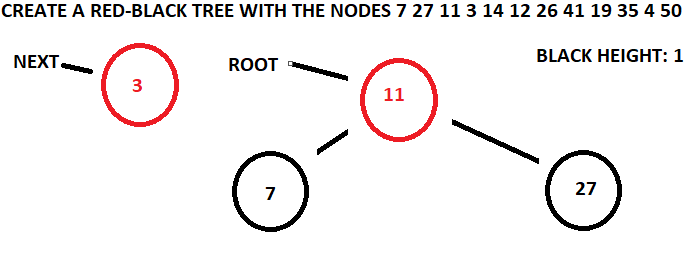
The insertion of the 11 node creates an outside-inside rotation. The 11 node is rotated with the 27 node such that the 11 node becomes the parent of its new right child, the 27 node. The 11 node is made right child of the 7 node. This intermediary step creates an outside-outside rotation with the 7 node being the grandparent and the 11 node being the parent. Performing an outside-outside rotation will change the grandparent’s and the parent’s color, make the grandparent the left child of the parent, and make the parent the new root. The black height is maintained at one, and the tree is balanced.



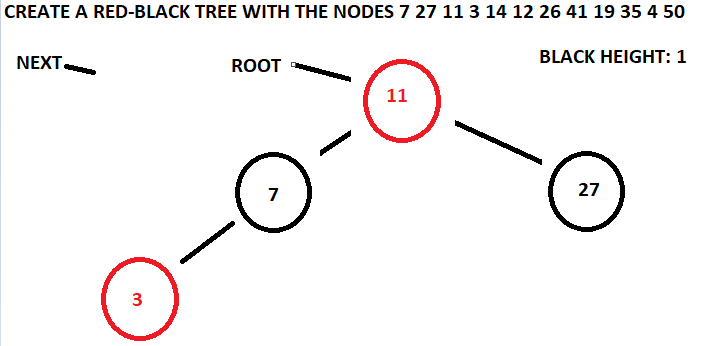
A red node with the value 3 is created to be placed into the tree. When the 3 node starts the path down the tree, the first node it is compared to is a black node with two red children.



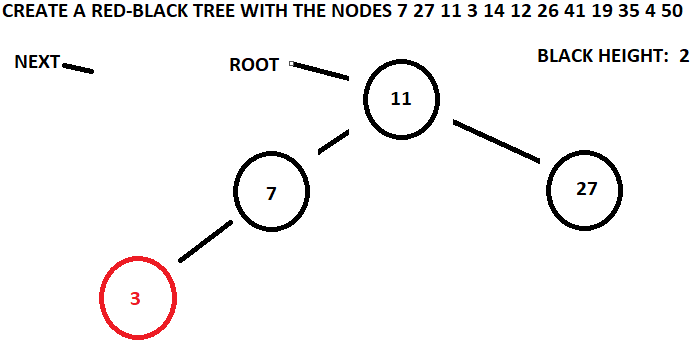
This is a good time to do a color swap, because it will mitigate any problematic rotations that could occur further down the tree. The black parent node and both red children change color.



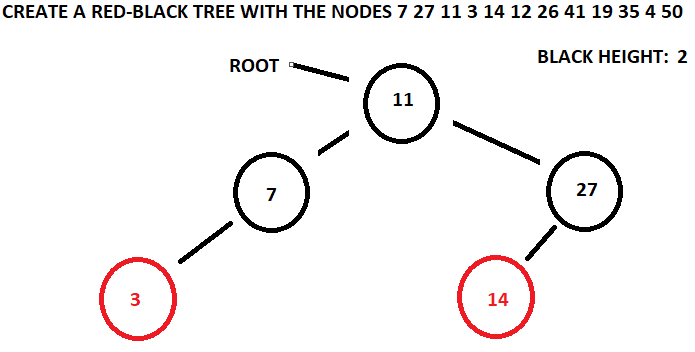
The red 3 node goes to the left child of the 11 node. Then the 3 node is placed as the left child of the 7 node.



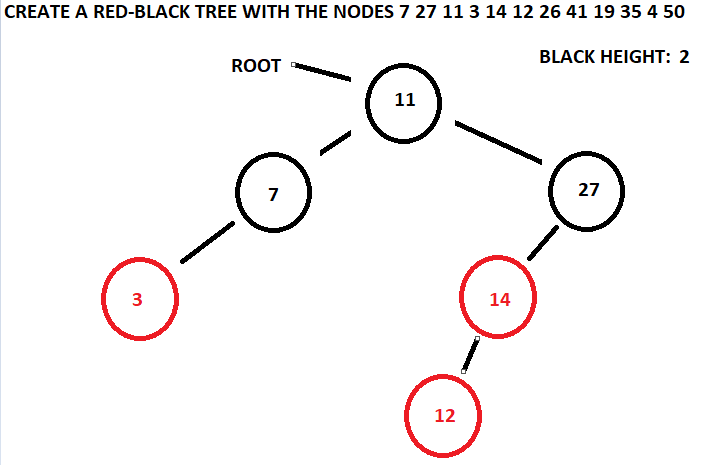
Once the 3 node is placed, the root is checked. The root cannot be red, so it is immediately changed to black and the black height becomes 2.



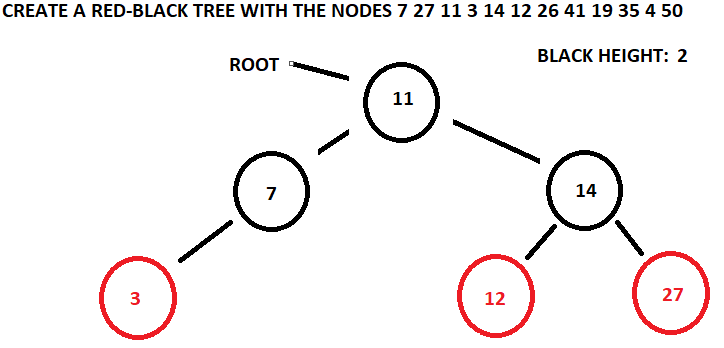
The value 14 is placed into the tree next. It goes to the right child of the root, then places itself as the red left child of the 27 node .



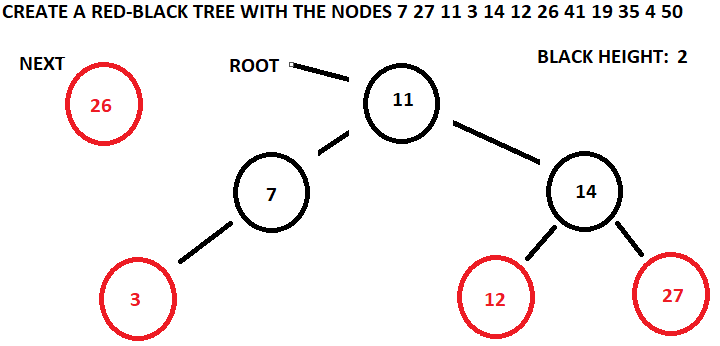
The red 12 node is the next value to be placed into the tree. It goes to the right child of the root, then to the left child of the 27 node, where it is placed as the left child of the 14 node.



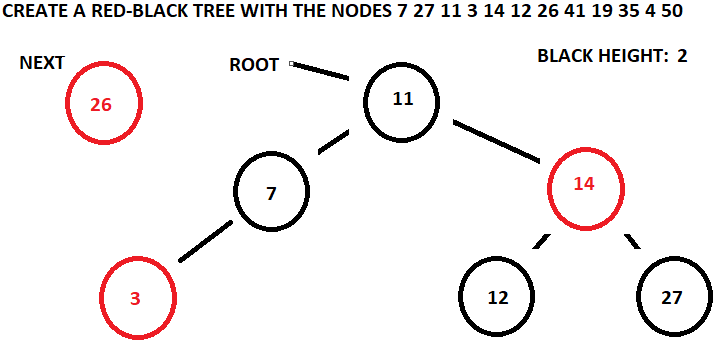
The placement of the 12 node creates the need for an outside-outside rotation where the grandparent is the 27 node and the parent is the 14 node. Both the grandparent and the parent change colors and the parent makes the grandparent its right child, while the parent moves up and becomes the right child of the root. The tree is now balanced and the black height did not need to change.



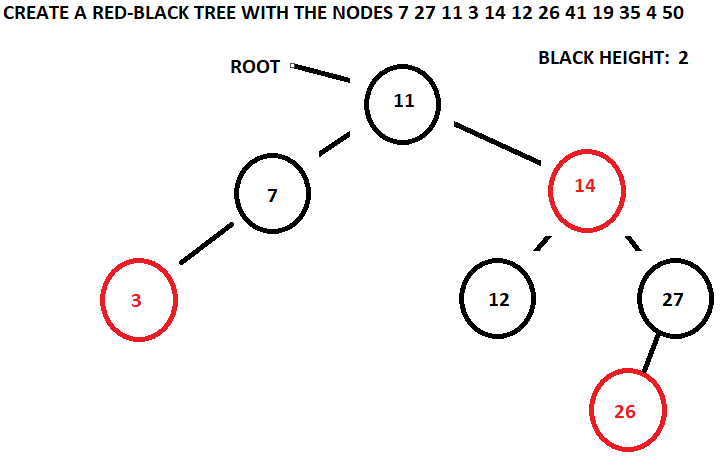
The value 26 is created as a red node and starts to follow the path down the tree. It goes to the right child of the root and then notices that the black 14 node has two red children. The path is paused so that a recoloring can occur.



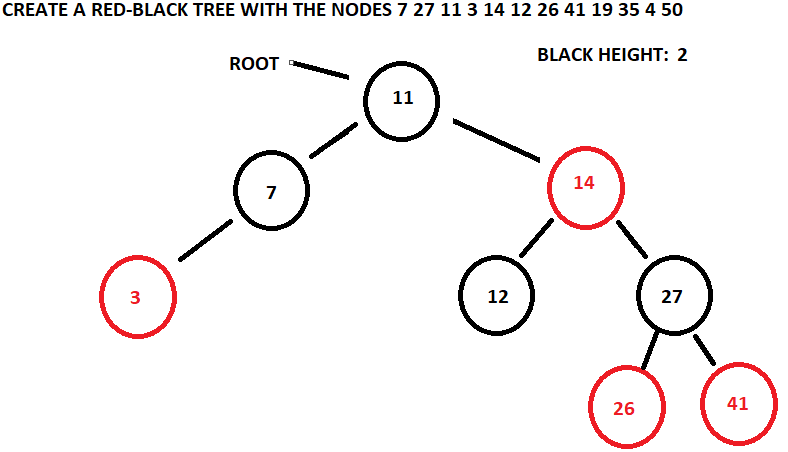
The black 14 node becomes red and the two red children nodes become black. A check is made to make sure that a rotation is not needed with this recoloring. It is not, so the insertion of the 26 node can continue.



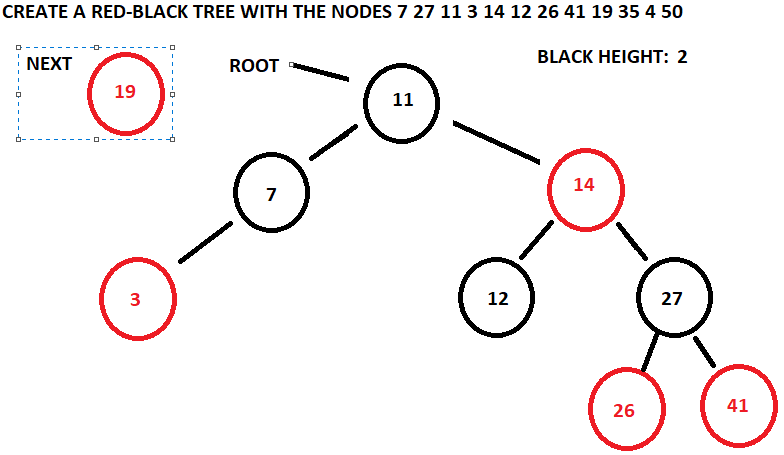
The 26 node goes to the right child of the 14 node and installs itself as the left child of the 27 node. The black height is still 2.



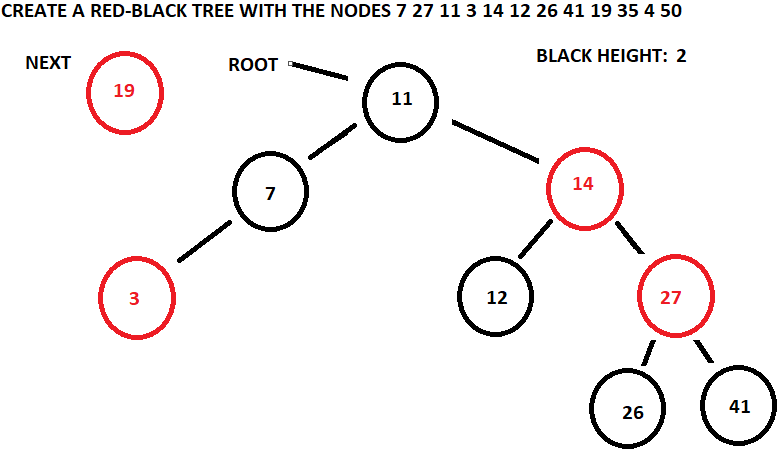
The value 41 is creates a red node. It goes to the right of the root, to the right of the 14 node, and becomes the right child of the 27 node.



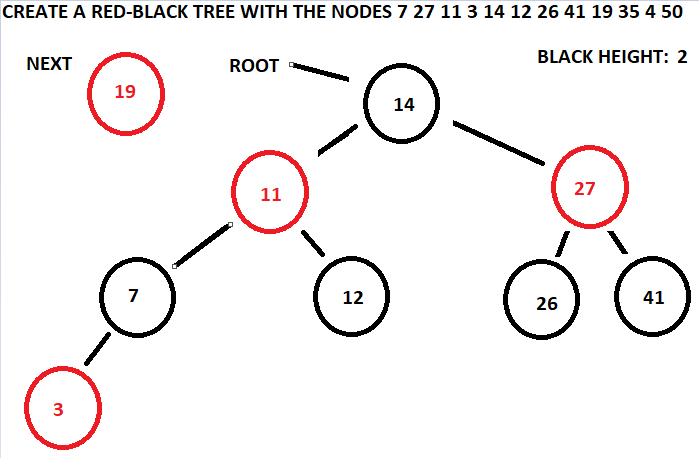
The value 19 is created as a red node and starts its path down to its placement. It goes to the right of the root and then goes to the right of the 14 node. When it comes to the 27 node, the need for a recoloring arises because the black 27 node has two red children.



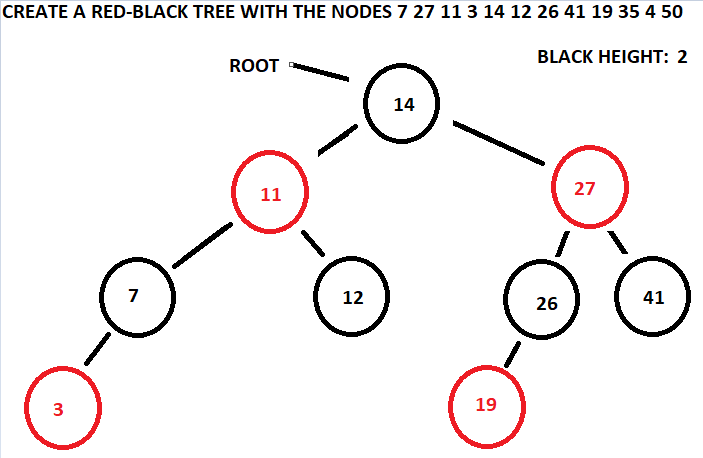
The 27 node is made red and the two children are made black. Because of the recoloring, an outside-outside rotation has become necessary to correct the tree.



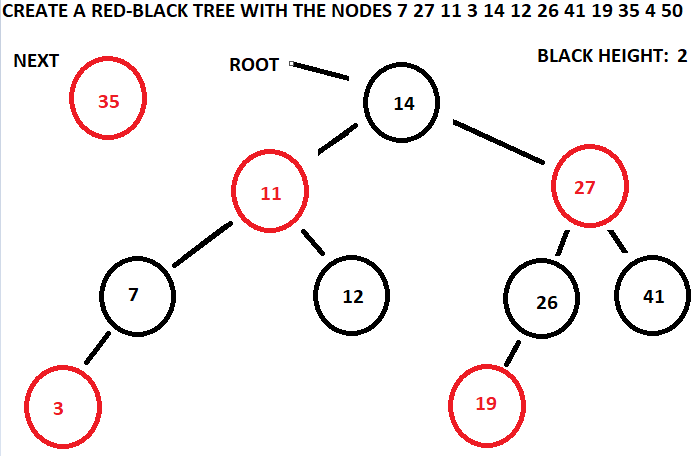
The outside-outside rotation uses the root as the grandparent and the red 14 node as the parent. Both the grandparent and parent change colors. The grandparent’s right child becomes the parents left child. The parent’s left child is now the grandparent. The new root is the parent and the black height is maintained.



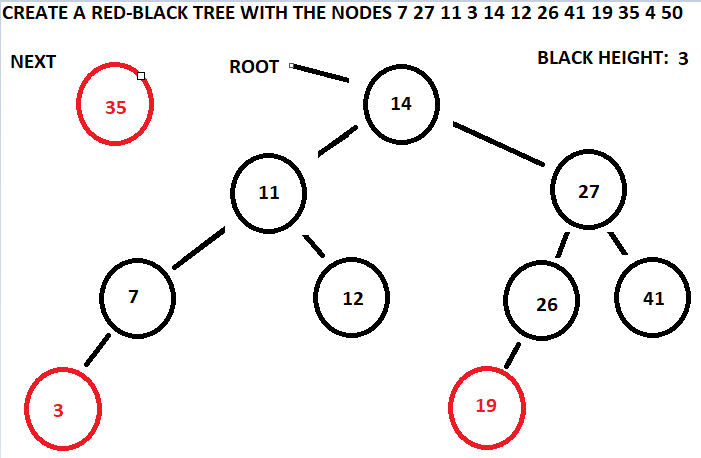
The 19 node, which had paused at the 27 node, now continues down its path. The 19 node goes to the left child of the 27 node and becomes the left child of the 26 node.



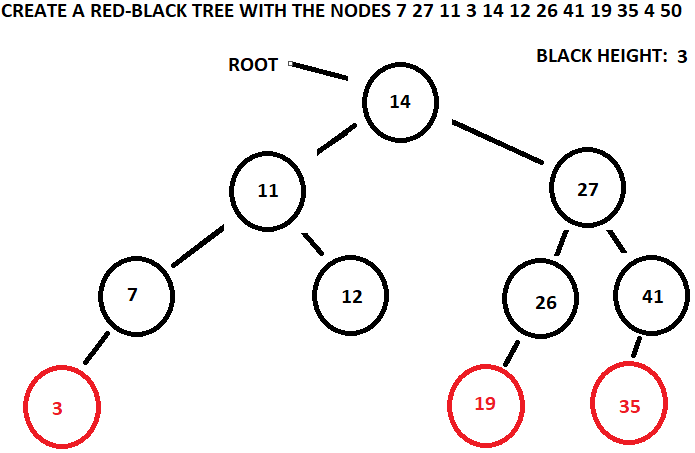
The red 35 node starts going down its path of the tree but immediate notices that the root is a black node with 2 red childred



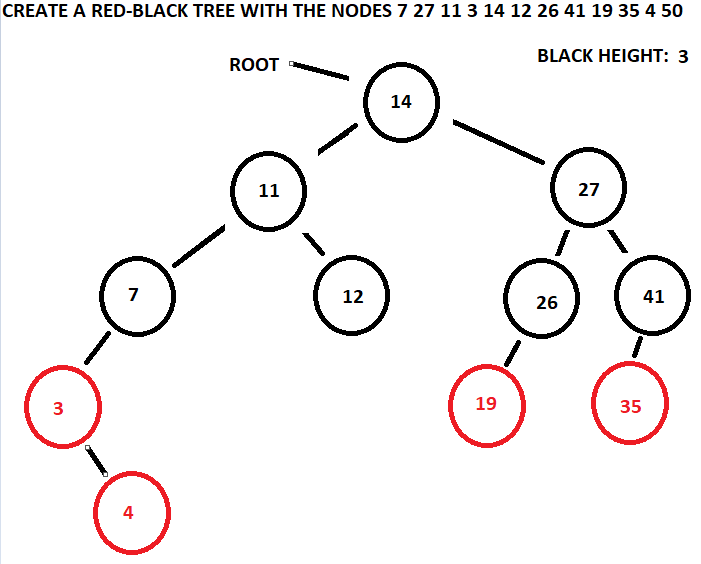
The root is changed to red and the children are changed to black. However, since the root cannot be red it is changed back to black and the black height is increased to three.



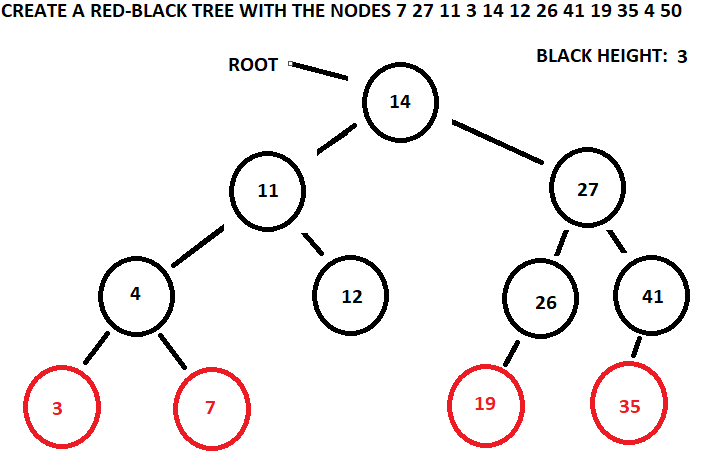
The red 35 node then continues its path through the tree. It goes to right of the root, then it goes to right child of the 27 node, then it becomes the left child of the 41 node.



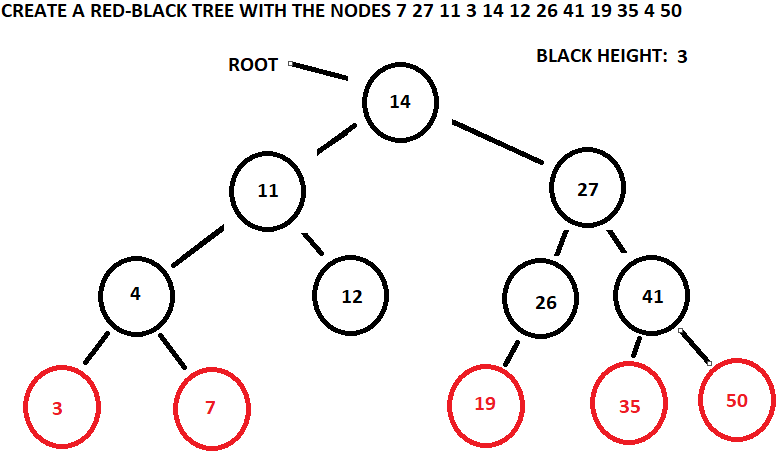
The 4 value is now placed into the tree. It goes to the left of the root, to the left of the 11 node, to the left of the 7 node, and is placed as the red right child of the 3 node.



The placement of the 4 node creates the need for an outside-inside rotation. The grandparent is the 7 node, the parent is the 3 node, and the x node is the 4 node. The x node makes the parent node its left child and the grandparent node makes the x node is left child. This sets of the need for the outside-outside rotation. The grandparent remains the 7 node and the parent is now considered the 4 node. Both the parent and grandparent node change colors. The parent makes the grandparent its right child and the grandparent makes its left child a null node. The parent is now the new right child of the 11 node.



Finally, the value 50 goes into the tree. The 50 node, which is red, goes to right of the root. From there, the node goes to the right of the 27 node. The 50 node is then placed as the right child of the 41 node.



This concludes the insertion of the list into a red black tree. The next set of notes will cover the deletion of values from the red black tree.