Red-Black Tree Demonstration

Problem:

For some reason which I do not understand, sometimes in Firefox in Linux typed numbers do not register in the applet. I have found that a few reloads gets it to work eventually.

New:

As of 24 June 2006 every important rule which the applet uses to execute an insertion or deletion step is displayed in a textfield prior to its graphical rendering. We believe this will help understand the complex interaction of rules and current tree configuration, especially in the case of deletion. Rules have numbers which correspond with detailed explanations at the bottom of this page (see the sections below on insertion rules and deletion rules).

As of 25 June 2006 nodes that are selected for deletion remain visible in the tree until the last click on **Next Step** for that run.

Java Version:

If this applet does not work with your browser it is probably because it has been compiled with Sun's Java version 1.6 and your java plugin is version 1.5 or earlier. In that case you can download the source code (see below) and compile yourself or try an older version such as this one or its maintenance version.

Usage:

To insert a node: type an integer. It shows up in the textfield labeled **value:** (you may need to click somewhere into the applet first). Click on the **Add Node** button to begin insertion of a node with the specified integer value. Repeatedly click on the **Next Step** button to advance step-by-step through the insertion algorithm and see what changes are made at each iteration. As long as the Adding sign is visible between the **Restart** and **Undo** buttons you need to keep clicking (but you can cancel by hitting the **Undo** button). To insert and avoid the step-by-step feature: enter a number in the textfield and hit the return key. Click on the **Restart** button to restart from an empty tree. To delete a node: click on the **Delete Node** button then click on the node you wish to delete. The node should turn green. Repeatedly click on the **Next Step** button to see the steps involved in deleting the node as long as the Deleting sign is visible (or cancel with **Undo**). The delete feature has been updated as of 8 December 2002 to eliminate all previously reported problems. If you find a problem please send email to franco@gauss.ececs.uc.edu. Click on the **Undo** button to restore the tree to its state before the last node was inserted or deleted. Choose "Fast" for normal speed. If this is too much for your computer, select "Slow" or "Crawl".

Quick add:

Type an integer into the textfield and hit return. The new object will be inserted into the red-black tree without having to hit the **Next** button at all.

Hot Keys:

If the dot panel, **Add Node** button, **Delete Node** button, **Next Step** button, **Undo** button, or **Restart** button are in focus you can use key presses to enter numbers and initiate actions. To insert a node: key in a number (which should appear in the value text field) then hit the "A" key. To advance to the next step, either while inserting or deleting, repeatedly hit the "N" key. To insert and avoid the step-by-step feature: key in a number then hit the "Return" key. To restart with an empty tree, hit the "R" key. To delete a node: hit the "D" key, then use the

mouse to select a node to delete, then use the "N" key to advance. To undo the previous move, hit the "U" key. Speed and the choice of duplicates are not controlled by hot keys. To put the dot panel in focus, just click on it.

Deletion colors:

A node that is selected for deletion turns green when it is selected. The deletion algorithm uses the notion of successor node (also predecessor node - see below). If a successor or predecessor is needed, on the next step the selected node is returned to its original color and remains part of the red-black tree until it is replaced at the end of the deletion run, otherwise it stays green until the run is completed whereupon it is deleted. If a successor or predecessor node is needed, it becomes yellow and stays yellow for the entire run of the deletion algorithm. The green and yellow nodes are not considered part of the red-black tree but are needed for reference so they retain positions as though they are part of the red-black tree until the end of the run. At that point, either there is a yellow node and it replaces the node originally selected for deletion with its color changed to red or black as necessary, or there is a green node and it is just deleted.

Sentinels:

It is possible to implement red-black tree algorithms by using *Null* to indicate the absence of a child. However, we take the more customary approach of attaching invisible black nodes called *sentinels* to red-black tree nodes instead. This means there really are no leaves among the visible nodes. Using sentinels makes the algorithms easier to implement but causes peculiarities in some descriptions during deletion. A description might say we need to worry about the "near" nephew of a node but that "near" nephew is nowhere to be found! That is OK, there really is a near nephew: it is a black sentinel.

Documentation:

As of 8 December 2002 this applet uses insertion and deletion procedures as described in:

> Berman and Paul. *Sequential and Parallel Algorithms*.
> Brooks/Cole PWS Publishing Co, 1997 (ISBN:0-534-94674-7).

These rules and a description of red-black trees are repeated in the sections below after the Source Code section. How a successor or predecessor is chosen is discussed in the last section below.

Source Code:

The source code is now in reasonable shape and includes comments which point to the cases described in the text cited above. It may be found here. It uses a Stream class found here to facilitate "next" step pauses. The necessary applet tag looks like this:

```
<applet code="RedBlack.class" height=560 width=900>
```

The source code implements two interesting features for helping to speed the understanding of red-black trees. A parameter tag may be added to the applet tag in order to start the applet with a series of quick insertions. This is done, for example, as follows:

```
<applet code="RedBlack.class" height=560 width=900>
<param name="args" value="12 6 25 10 3 18 55 11 7 4 2 15 21 33 98 12 9 13 16 20 22">
</applet>
```

To see how this looks click here (the applet you will be directed to only works on Java 1.2.2 and higher interpreters so the buttons may not work on your browser). Secondly, there is code for

including a button called **Color** so that clicking on a node and then the **Color** button will reverse the node's color. You can try it in the applet above if your buttons are working. Just uncomment the lines

```
//saved_pick = pick;
```

in the mousePressed method of class RedBlack,

```
//p1.add(colorit = new Button("Color"));
//colorit.addActionListener(this);
```

in the init method of class RedBlack and

```
//else if (evt.getSource() == colorit) colorIt();
```

in the actionPerformed method of class RedBlack.

The code is designed for SUN Java 1.6. If your browser cannot comprehend this applet try this version or compile the source code above.

Tree Relatives:

- A *node* is an item of data stored in a red black tree. A node has a unique number to identify it.
- A red-black tree has numerous *levels* on which nodes reside. The top level is called level 0, the next level under that is level 1, then level 2 and so on. The maximum number of nodes on level i is 2^i.
- Every node except one has a single *parent*. If a node resides on level i, its parent is on level $i-1$. A line between two nodes is used to indicate parenthood. It is understood that the node at a higher level is the parent.
- The node that resides on level 0 is called the *root*. Every non-empty red-black tree has a single root. The root has no parent.
- Every node has at most two *children*. If a node resides on level i, then its children are on level $i+1$. A node that has no children is called a *leaf*. A node is the parent of its children. Therefore, the line connecting two nodes expresses parenthood and childhood at the same time.
- A child's position is important: we distinguish *left child* from *right child*. There can be at most one of each with respect to a single parent.
- If two nodes have the same parent, they are called *siblings*.
- A child of a sibling of a node is called a *nephew* of that node. If the sibling X of node Y is a left child, then the right child of X is the *"near" nephew* of Y. If the sibling X of Y is a right child then the left child of X is the "near" nephew of Y. Other nephews are called *"far" nephews*.
- If node X is a left child of node Y and node Y is a left child of node Z, then we say that X and Z are on *opposite sides* of Y. If node X is a right child of node Y and node Y is a right child of node Z, then we also say that X and Z are on opposite sides of Y. Otherwise, X and Z are on the *same side* of Y.

Red-Black Trees:

These are binary trees with the following properties.

1. Every node has a value.
2. The value of any node is greater than the value of its left child and less than the value of its right child.
3. Every node is colored either red or black.

4. Every red node that is not a leaf has only black children.
5. Every path from the root to a leaf contains the same number of black nodes.
6. The root node is black.

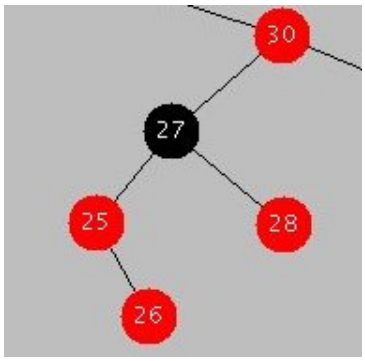
An *n* node red-black tree has the property that its height is *O(lg(n))*. Another important property is that a node can be added to a red-black tree and, in *O(lg(n))* time, the tree can be readjusted to become a larger red-black tree. Similarly, a node can be deleted from a red-black tree and, in *O(lg(n))* time, the tree can be readjusted to become smaller a red-black tree. Due to these properties, red-black trees are useful for data storage.

Insertion Rules:

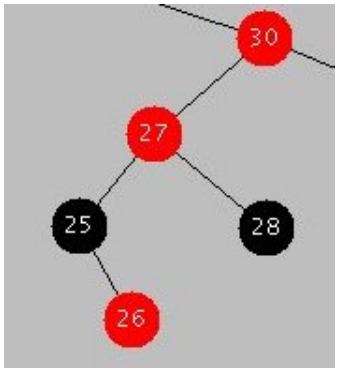
When a node is inserted in the tree it is given the color red. This does not affect the black node count on any path to a leaf. But it could lead to a single pair of consecutive red nodes in the tree. If the new node becomes a child of a black node there is no problem. But it may become a child of a red node. The double red violation will begin at a leaf. The rules below are designed to move the double violation up toward the root without affecting any path's black node count until it can be eliminated by bringing down a black from above or it reaches the root where it can be eliminated since the root can be colored black without consequence.

Let *current* refer to the red node that has a red child thereby identifying the location of the violation. The parent of current will always be black. The list below shows all possible states for current. The insertion algorithm performs the action associated with the correct state and either terminates or repeats.

Current's sibling is red



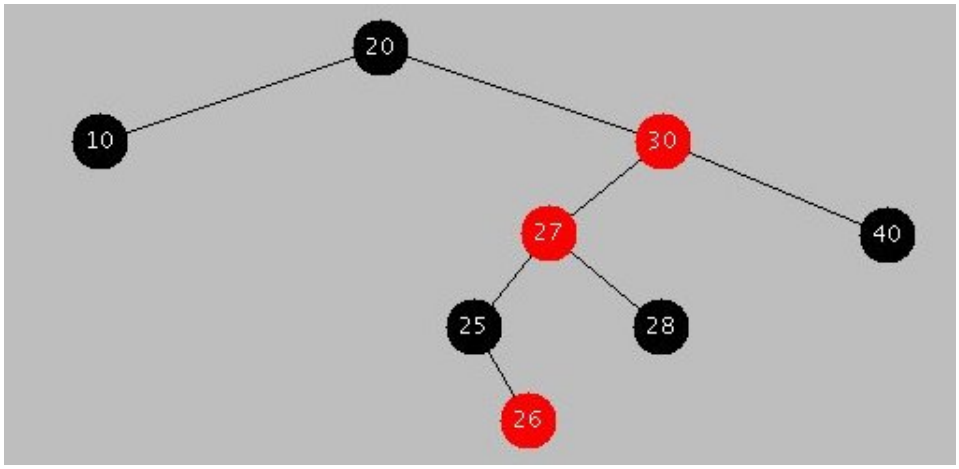
Current is node number 25 in the picture on the left. The result is shown at the right. Current is changed to node number 30.



<1>: **action**: Since the parent is black and current is red, current and its sibling may be colored black and the parent may colored red without affecting the number of blacks on any path to a leaf. In addition, this eliminates the double red below the parent but may cause a double red above the parent. Hence, set current to the parent of the parent and continue to **<4>**.

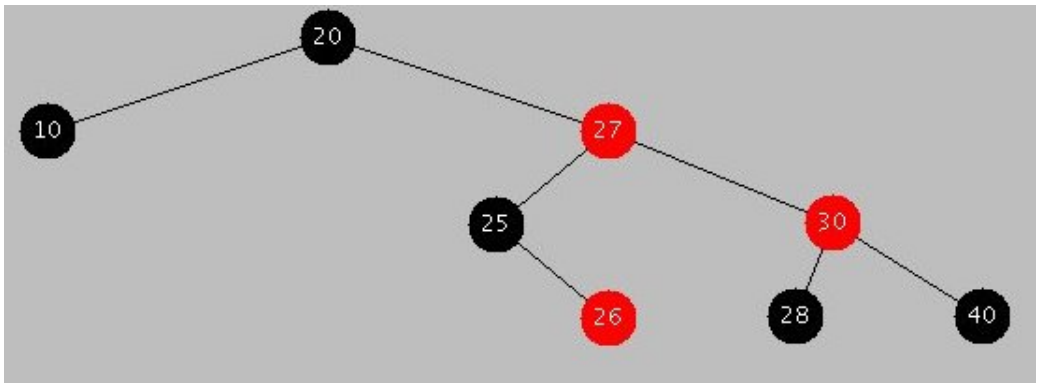
Current's sibling is black, current's red child is on same side as parent

Current is node number 30 in this picture. The result is shown in the next picture below where current is changed to node number 27.



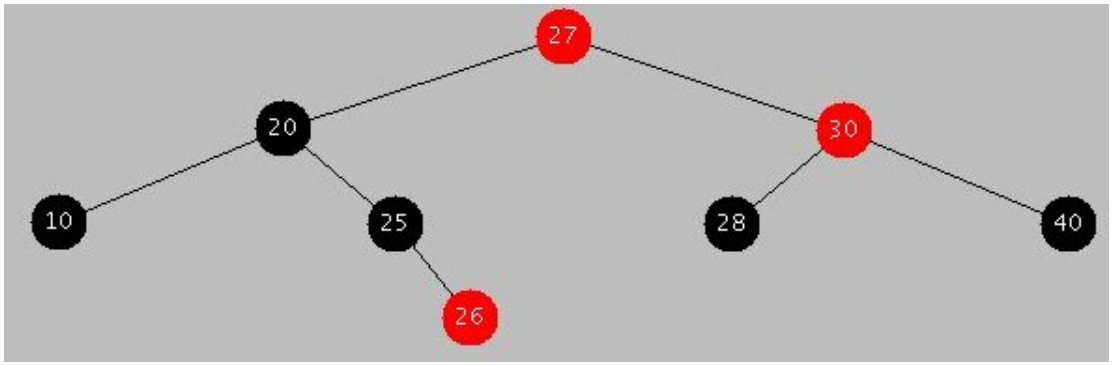
<2>: **action**: If current is the right child of its parent, rotate clockwise around current. Otherwise rotate counter-clockwise around current. Set current to the new parent of current (the pre-rotation red child of current). This satisfies the next rule and does not change any red-black tree violations. Continue to **<3.1>**.

Current's sibling is black, current's red child is on opposite side as parent



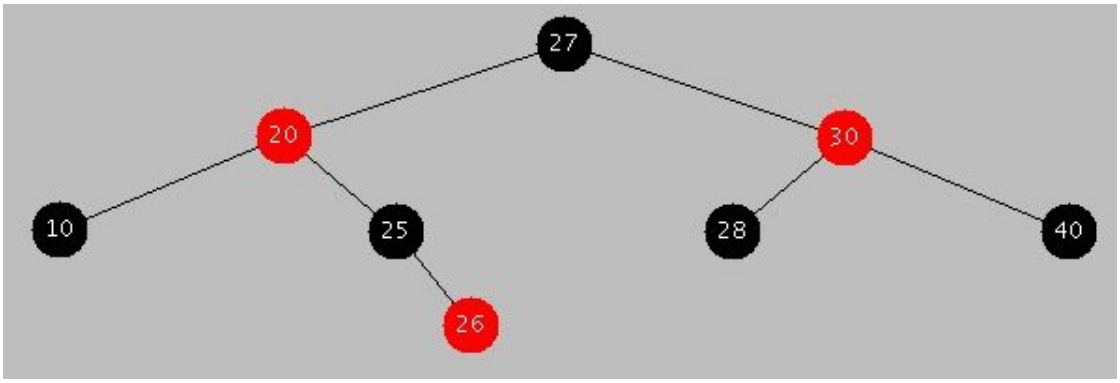
Current is node number 27 in this picture. The result of this step is shown in the picture below.

<3.1>: **action**: Rotate in the direction that causes current to become the parent of its pre-rotation parent. All paths through current's pre-rotation red child now are short one black because a black node has been rotated out. There are still two reds in a row. But the black node count on paths through the pre-rotation parent are unchanged. Hence...



Current is node number 27 in this picture. The result of this step is shown in the picture below.

<3.2>: **action**: Exchange colors between current and its pre-rotation parent. The black count through the pre-rotation parent remains unchanged. However, since current is made black, the double red is eliminated and all paths through the red child have increased their black node count by 1. Hence there are no red-black tree violations and the algorithm terminates here.



Red-black tree upon termination of Step <3.2>.

Check to see if there is still a problem

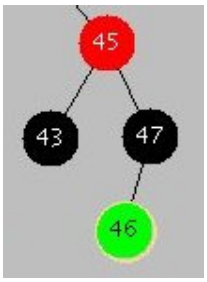
<4>: **action**: If current is the root or the root sentinel, the only thing left to do is set the root to black and exit. Otherwise, repeat.

Deletion Rules:

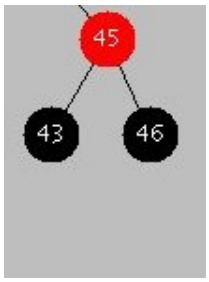
Let *current* refer to the node whose parent and siblings are queried in a deletion step. Initially *current* is the node selected for deletion. Suppose it has two children. Then another node is selected for deletion instead: namely, either a successor of current (a leaf or a node with only a right child of next greater value), or a predecessor of current (a leaf or node with only a left child of next smaller value). Either is easy to find and rules for doing so are in the next section. The deleted successor or predecessor is saved until the run terminates and then it replaces the node originally selected to be deleted. If a successor or predecessor is necessary, it becomes *current* when it is found. Due to the use of a successor, only nodes with at most one child need to be considered for deletion.

Assume current initially has at most one child. Consider five cases (with labels - the red ones are impossible cases):

Current is a red leaf.



Current (to be deleted) is node number 46 on the left. The result is shown on the right



[1]: **action**: If a successor or predecessor is used, replace the node selected for deletion with the leaf. Otherwise just delete the leaf. The definition of red-black trees is not violated.

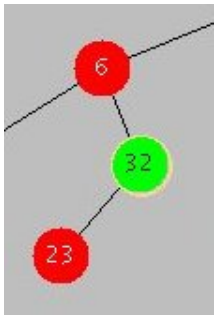
Current is red with one child.

[2]: **action**: This is impossible since the child must be black to avoid two reds in a row on a path but then the number of blacks on paths through the left side of the node is different from the number of blacks on paths through the right side.

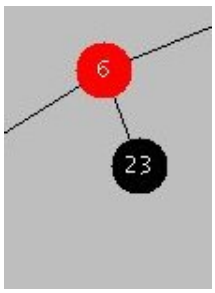
Current is black with one black child.

[3]: **action**: This is impossible since the number of blacks on paths through the left side of the node is different from the number of blacks on paths through the right side.

Current is black with one red child.



Current (to be deleted) is node number 32 on the left. Result is shown on the right.



[4]: **action**: The child must be a leaf, otherwise the definition of red-black trees is violated. Unhook current from the tree and make current's child a child of current's former parent (there is one non-violating way to do it). If current is a successor or predecessor, it replaces the node selected for deletion. Otherwise current is deleted. There is no violation of the definition of red-black trees since the only change is that a red leaf is deleted.

Current is black and has no children.

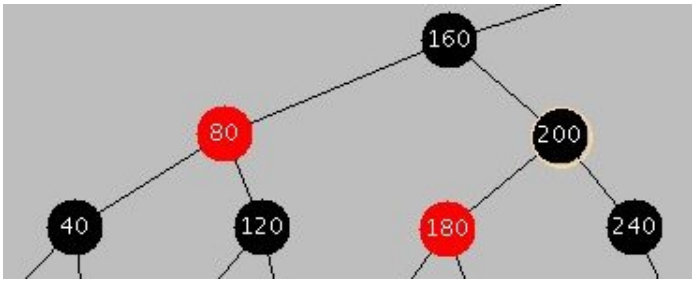
[5]: This step may be reached immediately or through Step **[5.1.2]** or **[5.2.1]**. If immediately, current is either the node selected for deletion, or the successor node, or the predecessor node. In all three cases unhook it from the tree (that is, replace it in the tree with a sentinel), but retain the notion of parent, sibling and such as though it is still in the tree because it may be needed in one or more steps below. If current is the root, just delete it and terminate.

The important invariant that holds at this point is the following: *the black node count on all paths through (the possibly "phantom") current, and only those paths through current, is short by one and there are no double red violations anywhere in the tree*.

The rules below either maintain the invariant as current rises in the tree or find a way to increase the black node count on paths through current. Specifically, either a black must be added to the path through current or the black node count on all other paths must be reduced. There are three cases to consider:

Current's sibling is red.

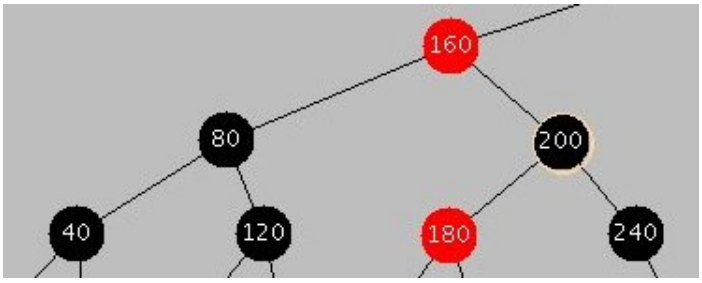
Since the sibling is red, the black node count in the sibling's tree cannot be reduced, and the black node count in current's tree cannot be increased. Therefore, a rotation is performed from the sibling's tree to current's tree to move a red node there. There are two action steps:



Current is node number 200, the parent is 160. The result of this step is shown below.

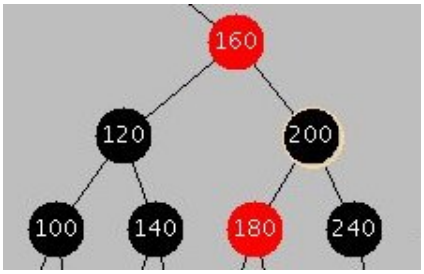
[5.1.1]: **action**: The colors of parent and sibling are exchanged. The parent color is then red and the sibling color is black. We want to rotate the red of current's sibling into current's path to give current's tree a chance to pick up a black. But that would reduce the black count in the sibling's tree

because a black would be rotated out. So, we exchange colors between current and sibling first.



Current is node number 200, the sibling is 80. The result is shown below.

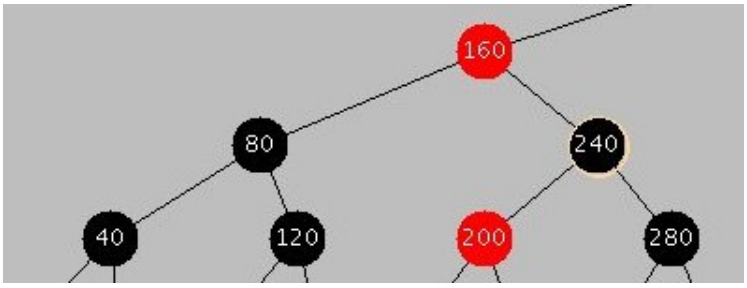
[5.1.2]: **action**: If the sibling is to the left of the node, rotate clockwise around the parent, otherwise rotate counter-clockwise around the parent. The black node count has not changed on any path but all paths through current have an extra red node which is the parent of current. Proceed to Step **[5]**.



Current is node number 200, the sibling is now 120. The rotated parent is now a right child of its new parent.

Current's sibling is black with two black children.

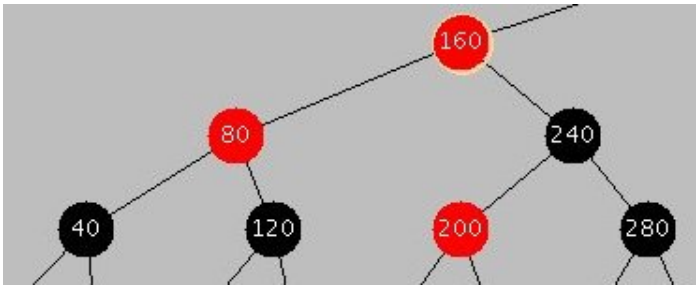
The black node count in the sibling's tree is reduced to match the count in current's subtree. In the lucky event that a double red is caused, set the uppermost of the pair, which is parent to current and its sibling, to black. In that case all violations are eliminated; replace the node selected for deletion with successor or predecessor, if there was one, and terminate. There are two action steps:



Current is node number 240, the sibling is 80. The sibling's black children are 40 and 120. The parent is 160. The result of this step is shown below.

[5.2.1]: **action**: Make the sibling red. This satisfies the goal but may introduce a double red violation at the parent and sibling. Let the parent of current be the new current. If current is black, there is no double red violation and all paths through it are short 1 black. Then the situation described in Step **[5]** applies so continue to Step **[5]**.

Current is node number 160.



[5.2.2]: **action**: Otherwise, current is red. Make it black. All paths now have the proper black node count and there are no double red violations. In this case, replace the node selected for deletion with successor or predecessor, if there was one, and terminate.

Current's sibling is black with one or two red children.

The opportunity to *rotate a black into current's path* now exists. We refer to this rotation below as the *final rotation* (which always happens) to distinguish it from a preliminary rotation (which may or may not happen) that is described later. We have to be careful about the after-final-rotation sibling of current, though: if it is red and the parent is red there will be a double red violation. This can be solved by *making the red pre-final-rotation parent black* before the final rotation. Unfortunately, if the pre-final-rotation sibling is black, the black node count through current will then be raised by 2 after the final rotation. This can be solved by *making the black pre-final-rotation sibling red if its pre-final-rotation parent is red* (the parent will then be turned black). This coloring policy results in the correct black node count through current as well as its after-final-rotation sibling, provided the pre-final-rotation sibling is black.

But it is possible that the pre-final-rotation sibling is red (therefore the pre-final-rotation parent is black). Since, in this case, it is required for the sibling to be black initially, the only way this could happen is by means of the preliminary rotation, yet to be described. But that rotation would have pulled a black out of the "near" nephew's path. If we extend the above mentioned sibling coloring policy to *require the sibling to take the color of the parent before the final rotation*, that black is restored. Hence when the near nephew's tree is moved to be a sibling of current on the final rotation, its black node count is unchanged, even if the pre-final-rotation sibling had been red.

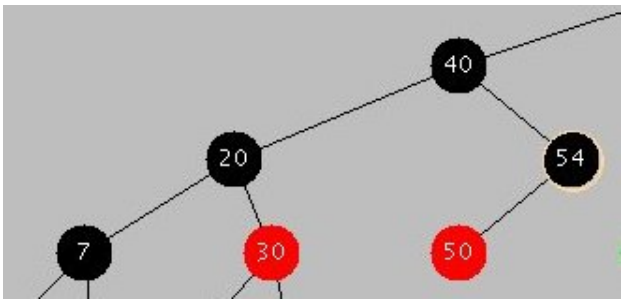
Moreover, the above operations introduce no double red violations.

So, the only problem is that the black node count on current's "far" nephew before final rotation (which is current's parent's sibling after rotation) will be reduced by 1. This is easily taken care of by *making the "far" nephew black* if it had been red. Doing so also eliminates the possibility of a double red violation due to the former sibling becoming red.

But, if the "far" nephew had been black, *a red must be moved into the "far" nephew side* from the "near" nephew (this is possible since there must be at least one red nephew) by a *preliminary rotation* around the sibling. That red could be set to black to increase the black node count on the sibling side. But if the parent were also set to black from red, as described above, the count will be increased too much. Luckily, the afore mentioned policy of setting the

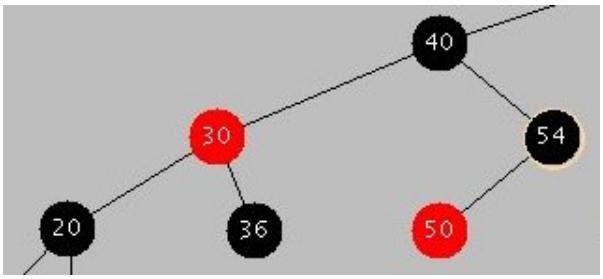
sibling's color to that of its parent before the final rotation ensures this does not happen.

Thus, there are two or three action steps depending on whether current's "far" nephew is black before any rotations take place:



Current is node number 54, the "far" nephew is node number 7, the sibling is 20. After rotation (shown below), the new sibling and old "near" nephew is 30.

[5.3.1]: **action**: If the "far" nephew is black, rotate in that direction around the sibling. This is the preliminary rotation referred to above. Current's new (pre-final-rotation) sibling is its old (pre-preliminary-rotation) "near" nephew which must be red. The purpose of this step is to fix the potential problem that the black node count of paths through current's (pre-preliminary-rotation) "far" nephew become 1 short once the black node is rotated into current's path (after the final rotation). By rotating the "near" nephew red in here, we have the opportunity to pick up a black in the "far" nephew's path in the next step.

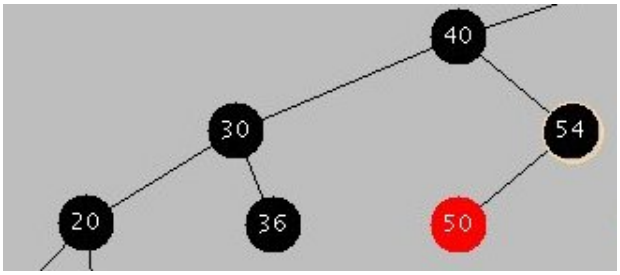


After the preliminary rotation above. The "far" nephew is 20, the sibling is 30 and the parent is 40. The result of this step is shown below.

[5.3.2]: **action**: The "far" nephew and sibling of this step may differ from the original "far" nephew and sibling due to the rotation of Step **[5.3.1]**. Set the color of current's "far" nephew to black, set the color of current's sibling to the color of its parent, set the parent's color to black. One or more of these nodes may already have been the color it or they was or were set to. It is simply easier to set the colors without checking first to see what they are.

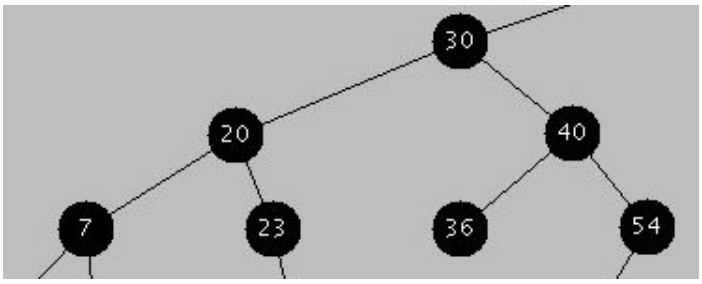
If current's pre-final-rotation parent is red, we risk the possibility of a post-final-rotation double red violation between the parent and current's post-final-rotation sibling. To prevent this, the parent is made black. But that will increase the black node count through current by one too many if current's pre-final-rotation sibling is black. So, in that case the pre-final-rotation sibling is made red. If the pre-final-rotation sibling has been rotated in from **[5.3.1]** and the parent is black, it needs to be changed to black to keep the "far" nephew's black count the same after final rotation. But, if the parent had been red before the final rotation, then the rotated-in pre-final-sibling must remain red otherwise the black node count through the "far" nephew will be too high by 1 after final rotation. The simple rule *color the*

"far" nephew black, make the sibling color the same as the color of its parent, color the parent black satisifes the above.



After setting colors as above. The result of this step is shown below.

[5.3.3]: **action**: Rotate around the parent in the direction of current. This is the final rotation. If there is a successor or predecessor, it replaces the node selected for deletion. The algorithm terminates.



After the final rotation.

Successors and Predecessors:

The algorithm is made as simple as possible by reducing all non-trivial cases to either deleting a red leaf or black node that has at most one leaf child via the notions of *successor* and *predecessor*. The successor of a node of value *X* is the node of the tree whose value is the least that is greater than *X*. The predecessor of a node of value *X* is a node of the tree whose value is the greatest that is less than *X*. Each is easy to find: just do a depth first search on leftmost child, in the case of successor, and rightmost child, in the case of predecessor. The search ends when a leftmost or rightmost child does not exist. Therefore, the search ends at a node with at most one child. Moreover, if the node is red, it must be a leaf and if it is black its only possible child must be a red leaf. Thus, after rebalancing, the successor or predecessor can assume the position of the deleted node without violating the red-black property number 2. In the case of a red leaf below a black successor or predecessor, the red leaf can simply be moved up to be a child of the successor's or predecessor's former parent and its color changed to black. Actually, the only hard cases involve a black successor or predecessor with no children.

This deletion algorithm may use either a successor or a predecessor. The decision is made as follows: the successor is used if it is red or it is not a black leaf (that is, it could be black with a red leaf); otherwise the predecessor is used. Whether a predecessor or successor is used, either is referred to as the successor in the above description.