Red-Black Trees

Binary Search Tree where

part of definition and can never be made false

1) every node is colored red or black (color of node needs only <u>1 bit</u> of data; 0 ⇒ black and 1 ⇒ red or vice-versa)

2) we will consider a NULL as black; ptrs to black nodes and ptrs to NULL are in one category, and ptrs to red nodes are in another category

Insert & Remove <u>Could</u> change tree so that these properties are not true, and if so, we'll need to fix that

3) no red node can have a red child; children of red nodes are always NULL or else black nodes

4) # of black nodes along any path from root to a NULL is the same as in any other such path (sometimes we call this the "black height")

ALSO: Root of tree will always be black

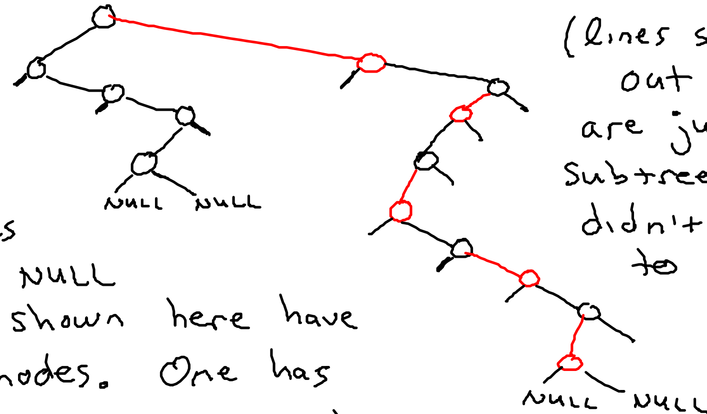
Property #3 forces every red node in a path from root to NULL to be followed by a black node or NULL (if it were followed by a red node, that would be a red node with a red child, which is not allowed)

Combine that with property #4, which says all root-to-NULL paths have the same number of black nodes.

Property #3 says that each such path can only have a single red node after a black node → you can't have two or three red nodes in a row after a black node. So, if all root-to-NULL paths in a tree have B black nodes:

- shortest possible path has B nodes, all black
- longest possible path has a red after every black

EX:
B = 5



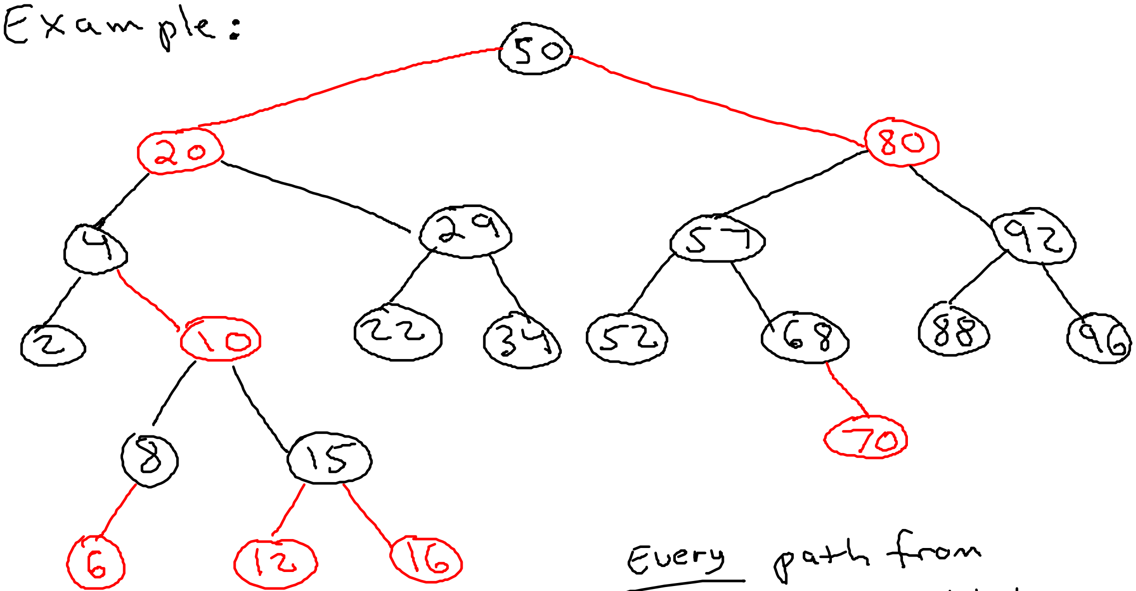
(lines sticking out of nodes are just subtrees we didn't bother to draw)

All 4 paths from root to NULL that we've shown here have 5 black nodes. One has 0 red nodes (the minimum). One has 5 red nodes — the maximum for a 5-black-node path, since there is no way to add another red node to that path without having two red nodes in a row (unless we were willing to have a new, red root but that would add a red node ~~to~~ all paths, even the one we said currently has no red nodes).

The point? The point is that the longest path is no worse than twice the length of the shortest path.

This is another way to ensure a **balanced** ($O(\log n)$ height) tree

Example:



Every path from root to NULL has 3 black nodes (plus the NULL itself)

So, what we need to do now is study the insertion and removal algorithms.

Red - Black Insertion

Being a BST, we do BST insertion first → hence insert as a new leaf. We have two choices

1) we could insert as a black leaf; that will mess up the black height property and we'll have to fix that

2) we could insert as a red leaf; that might result in a red child of a red parent, and if so, we'll have to fix that.

Situation 2 is easier to deal with and also less likely to be a problem, so we go with that → i.e. we insert the new leaf as a red leaf, not a black leaf.

With that insertion done, we only have a problem now if the new leaf's parent was also red → if it's black we can stop since no rules are violated.

Attempt #1

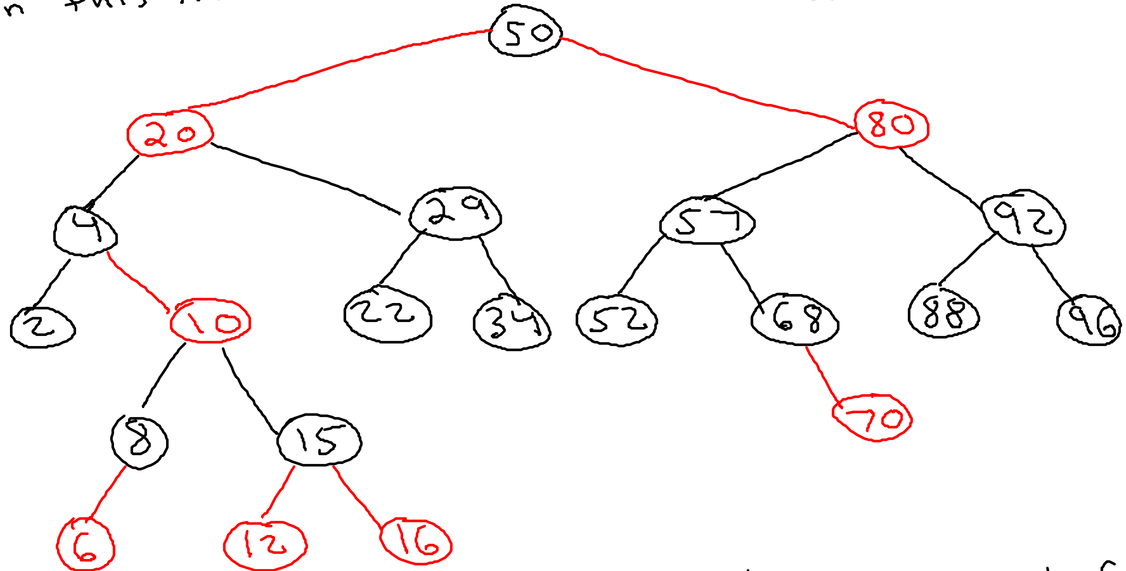
```
RBInsert(value)
{
    BST insert of value as new leaf
    color new leaf red, and let's label
        it "x"
    if (x's parent is black)
        DONE
    else   // x's parent is red
        DO STUFF (what stuff?)
}
```

However, some of the stuff we might do if x's parent is red will involve labelling a <u>different</u> red node as x and seeing if <u>its</u> parent is red. So we really want a loop, not a conditional. (which means reaching root should <u>also</u> be a stopping case for the loop)

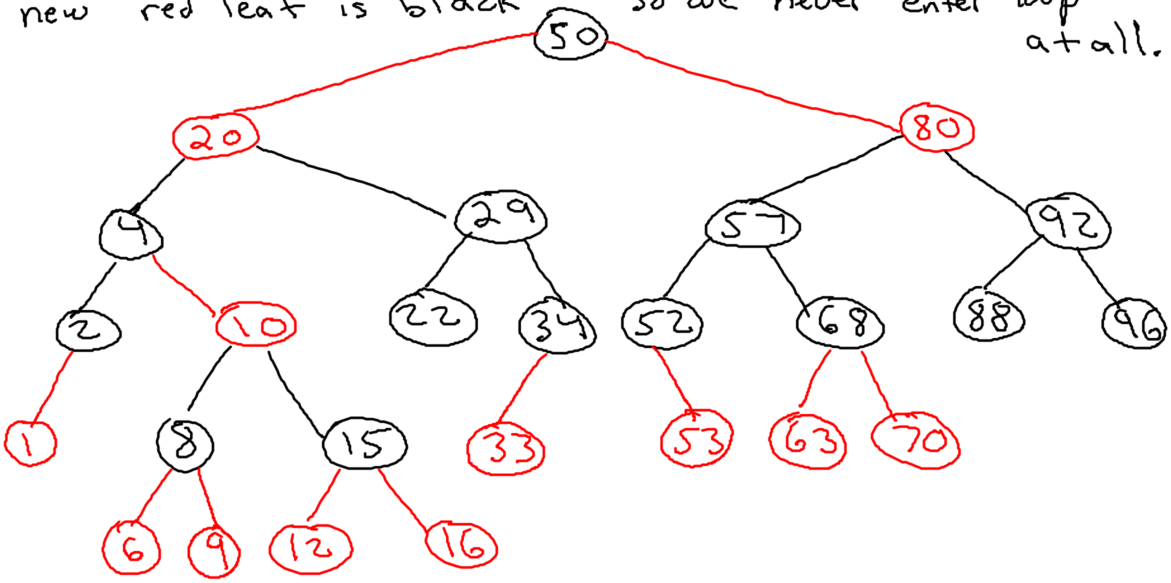
Attempt # 2

```
RBInsert( value )
{
    BST insert of value as new leaf
    color new leaf red, and let's label
        it "x"

    while (x isn't the root and x's parent
                                  is red)
    {
        // Insert cases we must still
        // talk about
    }
    color root black; // in case we turn it red
                                  in loop
}
```

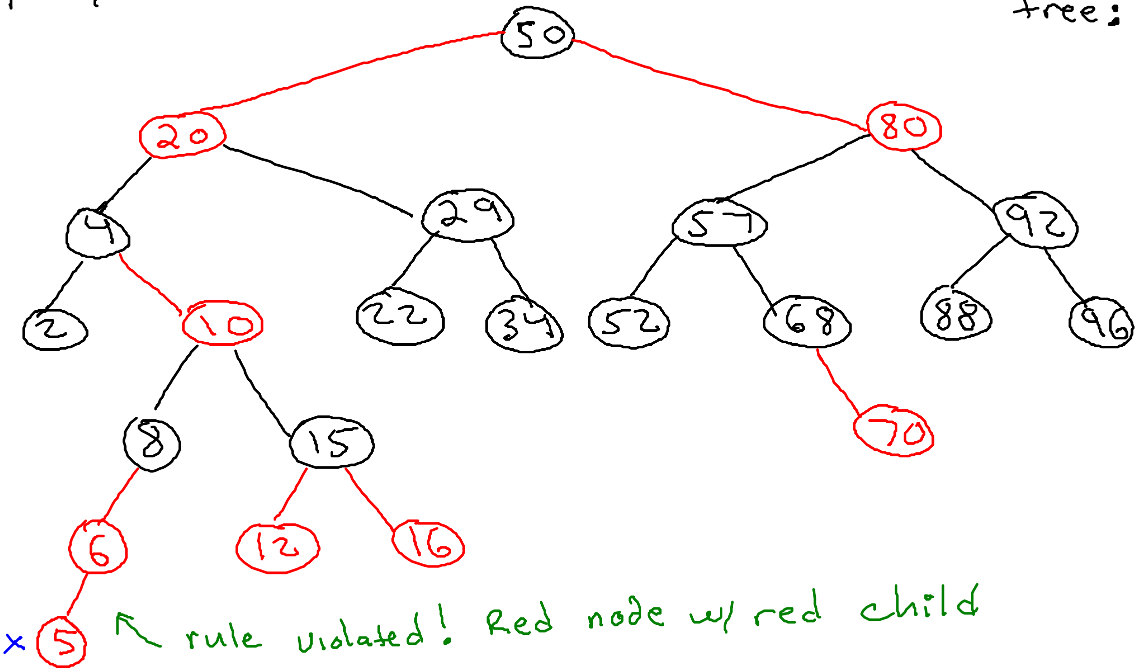
Example tree from earlier (we'll begin all our examples on this tree unless otherwise stated):



Insert 1, 33, 53, 63, and 9. In each case, parent of new red leaf is black so we never enter loop at all.

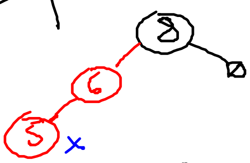


Now, consider the insertion of 5 into our example tree:

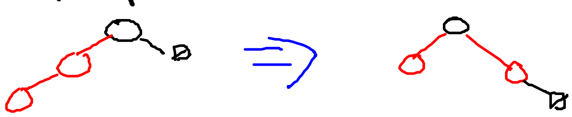


↖ rule violated! Red node w/ red child

x's parent's sibling (x's "uncle") is black:



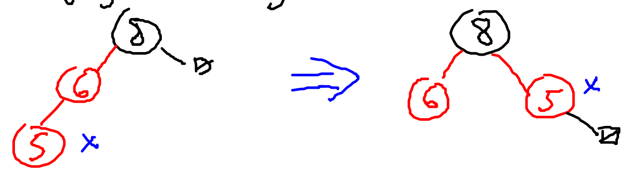
So we could take one of our red nodes from the left of 8 and insert it between the black 8 and its black child. This would fix the red-black properties:



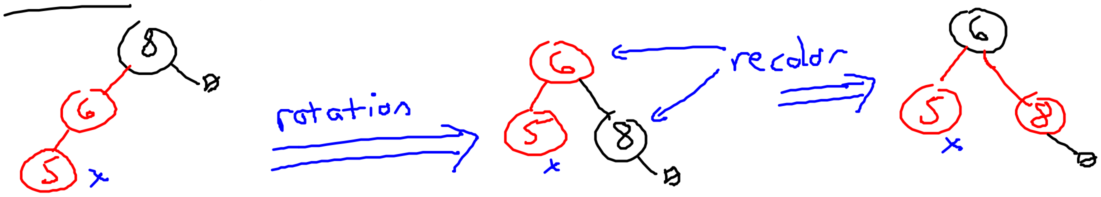
position-wise, this is the change we want to make

But we must keep it a BST as well!

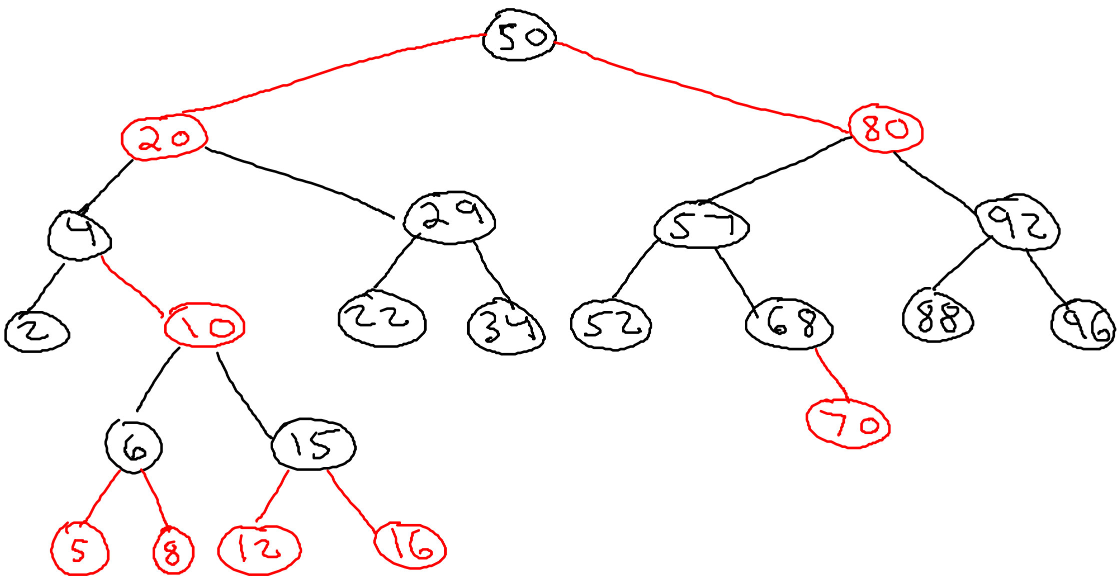
Simply moving a node to make that red-black position change:



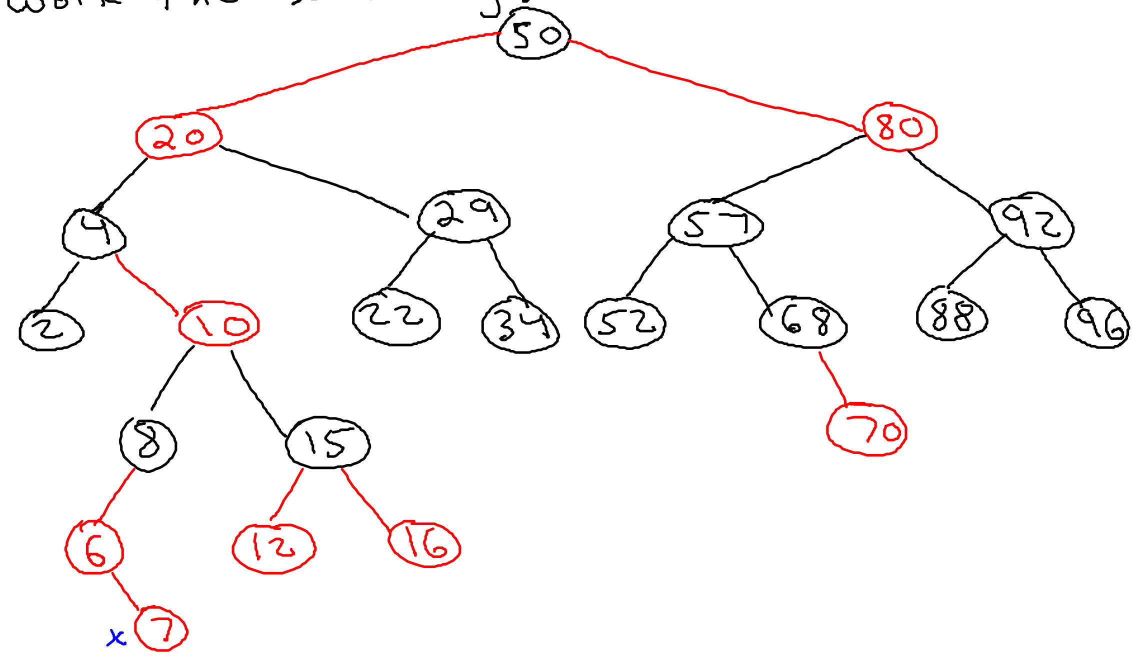
is not a legal BST, but a rotation and re coloring to get that red-black position change would keep it a legal BST:



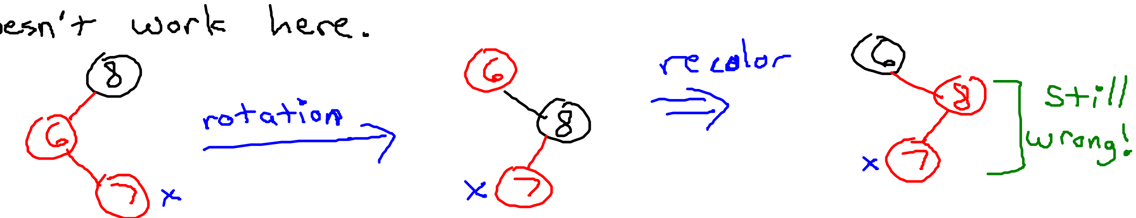
Completed insertion:

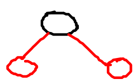


On the other hand, inserting 7 doesn't quite work the same way:



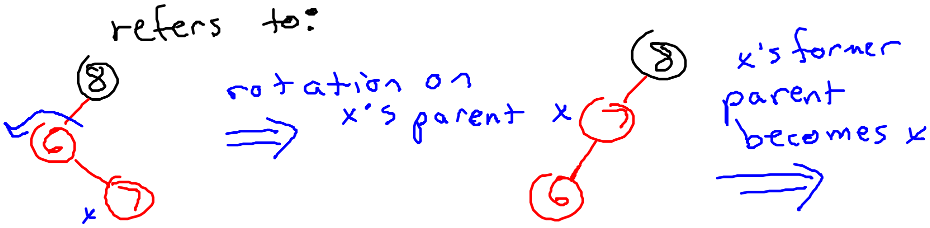
The rotation / recolor from the previous example doesn't work here.



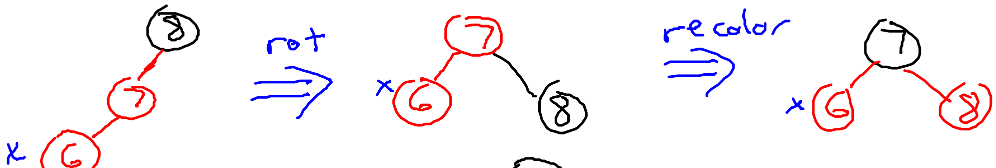
We <u>can</u> get the  setup we are looking for (like in the previous example) but this time its the new node that should become the black node, not the new node's parent as in the previous example.

Solution → <u>two</u> single rotations

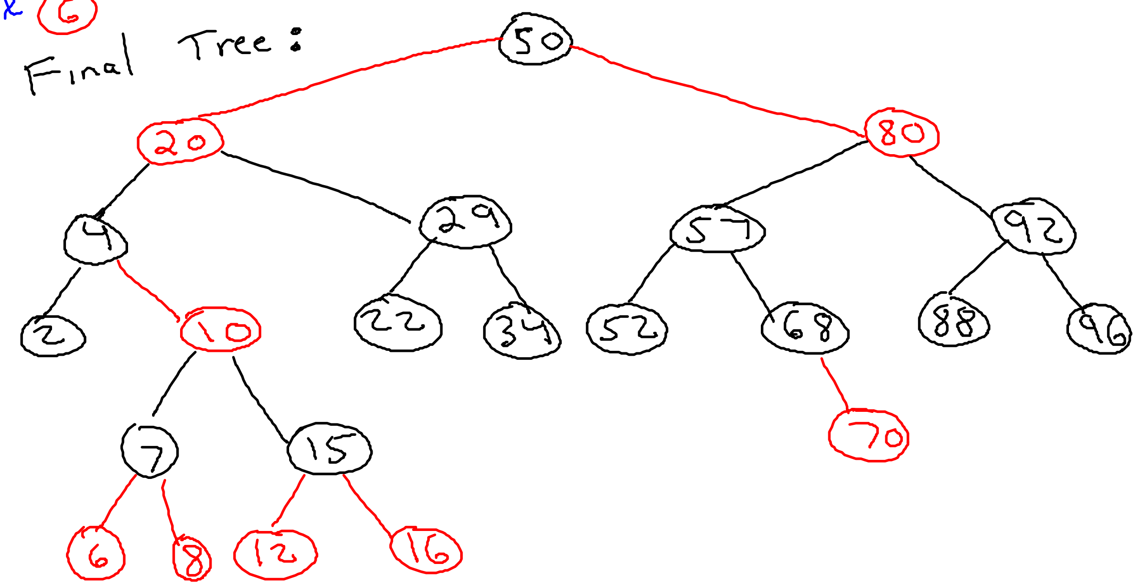
STEP 1) convert this situation to one our earlier solution will work for by doing a rotation and changing what node "x" refers to:



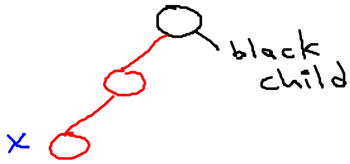
STEP 2) now, do the rotation and recolor from the "insert 5" case



Final Tree:



It's not too different from the AVL tree

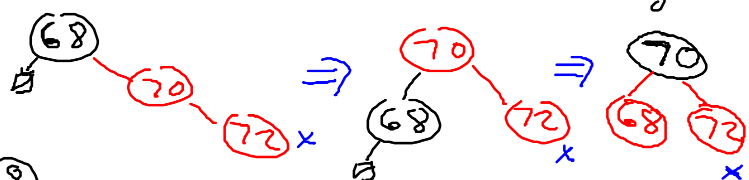


x is "outer" node (further from uncle than sibling is), so one single rotation and some recolorings will fix things

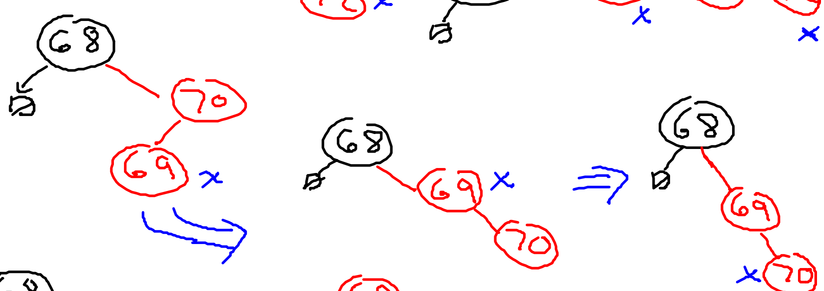
x is "inner" node (closer to uncle than sibling is), so in a sense we have a double rotation, since we need to perform two single rotations (and some recolorings) to fix this case.

Mirror image would work the same way:

Insert 72:

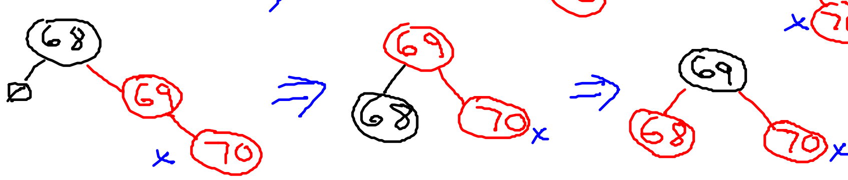


Insert 69:

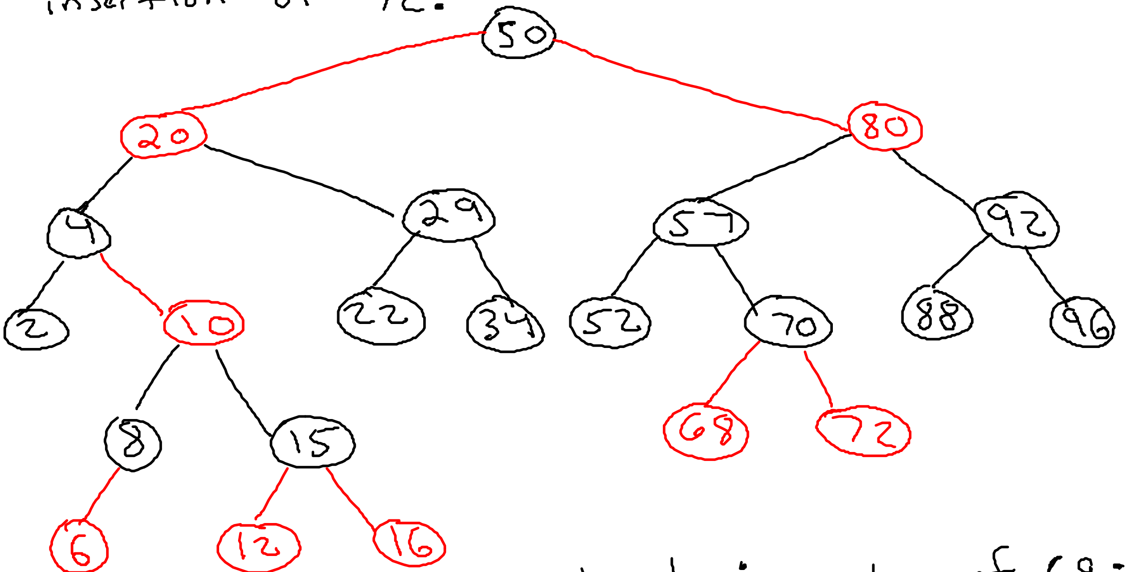


STEP 1)

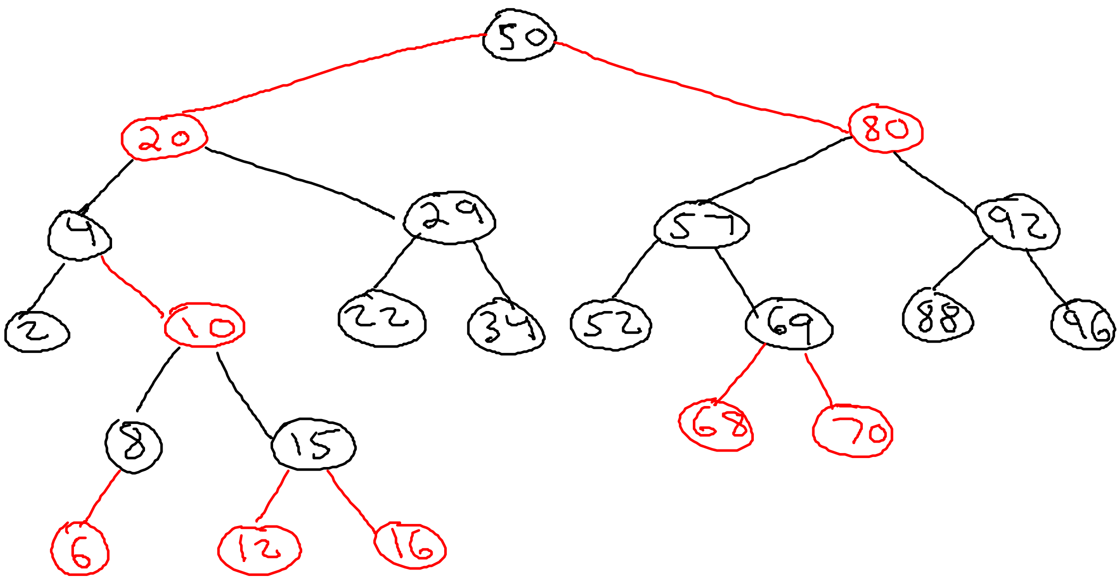
STEP 2)



Here is our example tree after a completed insertion of 72:



Example tree after completed insertion of 69:

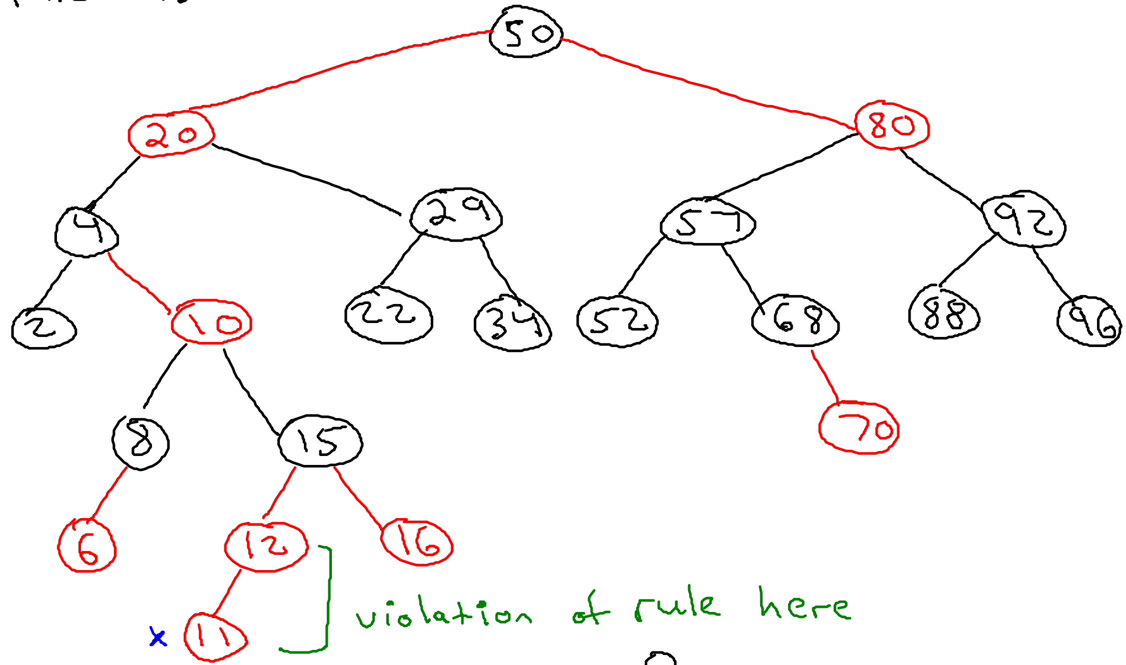


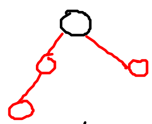
```
RBInsert(value)
{
    BST insert of value as new leaf
    color new leaf red, and let's label it "x"
    while (x isn't the root and x's parent
                                      is red)
    {
        // CASE 1:
        if (x's uncle is black)
        {    if (x is closer to uncle than x's
                      sibling is)
             {
                 • single rotation on x's
                   parent, away from uncle
                 • x's former parent relabelled
                   as x
             }
             // now x is definitely further from
             // uncle than x's sibling is
             • single rotation on x's grandparent
               toward uncle
             • x's former grand parent colored red
             • x's parent colored black
        } // END CASE 1
        else
           // CASE 2
    } // end while
    color root black; // in case we turn it red
                                      in loop
}
```

newly added

step 1 of "double rotation", if needed

All our recent insertion examples (5, 7, 72, 69) relied on x's uncle being black. Consider now the insertion of 11:

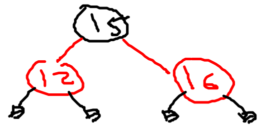


But this node layout :  doesn't allow for the "just move a red to the other side via rotations" approach since there is already a red node there and so we'd still end up with a red node with red child.

we need to try something new!

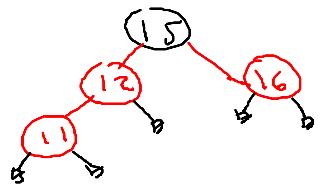
Let's consider black heights

Before insertion:



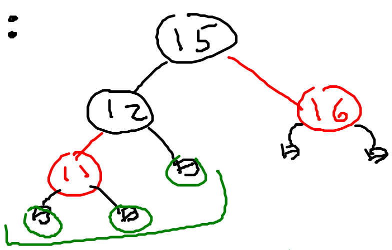
However many black nodes there were prior to this subtree, this subtree adds one more black node (15) en route to any of the four NULL pointers. Whatever we do to this subtree, we should still be adding only one black node to each of the paths to the NULL pointers.

Insert 11:



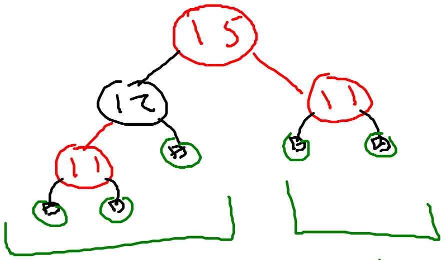
Still one black node added by this subtree to each path...

→ If we make 12 black so that 11 is not a child of a red node, then there are three paths to NULL with an extra black node:



too many black nodes for these 3 paths

We could fix that by making 15 red...

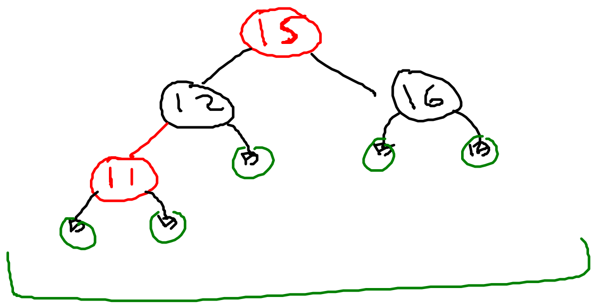


these three paths okay again ...

... but now these two paths have too few black nodes

... but making 15 red results in two paths missing a black node.

Last fix: make 11 black as well:



all paths have proper number of black nodes once again

Only problem → turning 15 red could make it the red child of a red parent.
If so, we can handle it by calling 15 "x" like we did for the new red leaf, and checking our cases again.

Final Insert algorithm.

```
RB Insert (value)
{    BST Insert → insert value as new red
         leaf and label it "x"
     while ( x not root and x's parent is red)
     {    if (x's uncle is black) // CASE 1
          {    if (x is closer to uncle than x's
                         sibling is)
               {
                    • single rotation on x's
                         parent, away from uncle
                    • x's former parent relabelled
                         as x
               }
               // now x is definitely further from
               // uncle than x's sibling is
               • single rotation on x's grandparent
                    toward uncle
               • x's former grandparent colored red
               • x's parent colored black
          } // END CASE 1
```

```
else // CASE 2: x's uncle is red
{
    • color x's parent black
    • color x's uncle black
    • color x's grandparent red
    • relabel x's grandparent as x
}
} // end while loop
color root black;
} // end RBInsert
```

newly added

Removing from a red-black tree

We'll start by doing ordinary BST removal. (And remember, that means that if the value you want to delete is in a 2-child node, you'll really end up deleting its in-order successor node instead.)

Once you remove a node, consider its color:

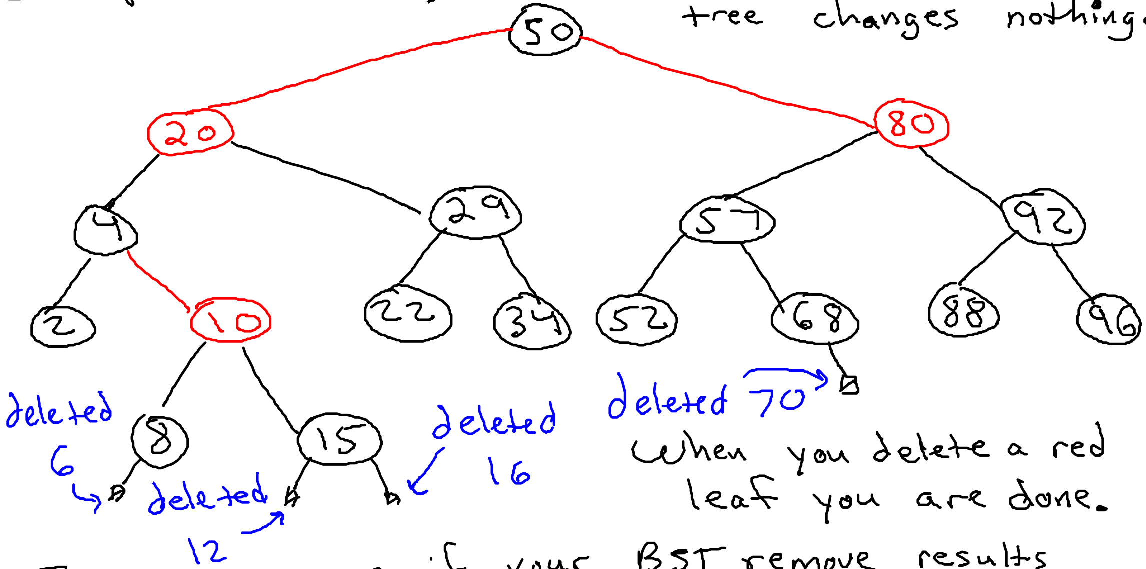
→ if you removed a red node, neither the "no two reds in a row" nor the "all black heights are the same" rule could have been violated, so you can stop.

→ if you removed a black node, both rules could be broken! We need to fix that.

If both rules are broken, that's actually easier to fix than if just the black height rule is broken. The reason for that is that if we have two red nodes in a row and we're short a black node, coloring a red node black can fix both problems.

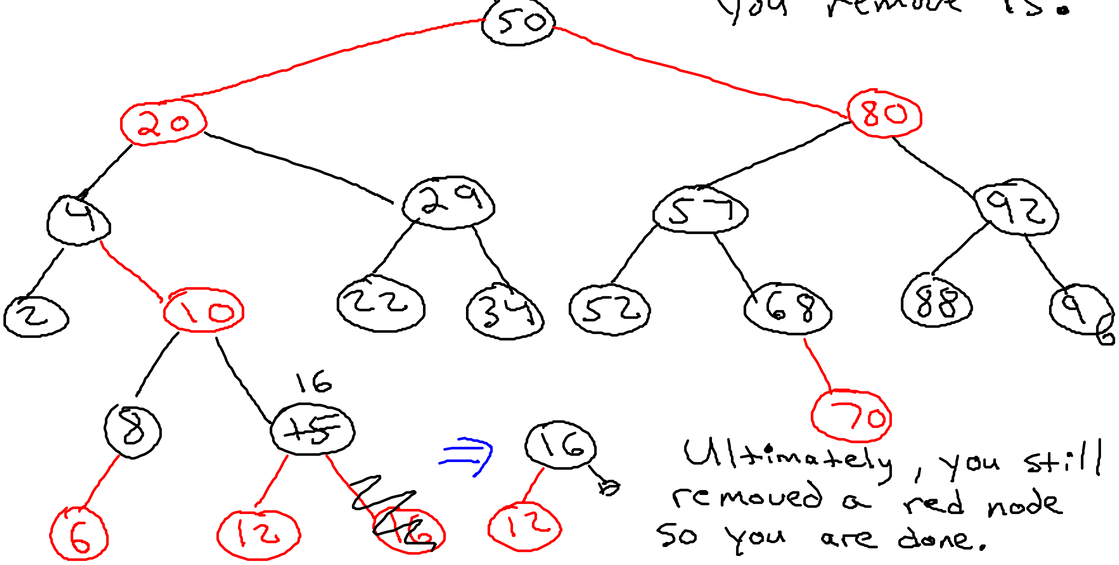
```
RBRemove (value)
{   BST removal
    if deleted node was red
        return;
    else  // deleted node was black
    {
      What the pointer that used
        to point to our deleted node
        now points to, label that "x".
      while (x not root and x is black)
      {   // Remove cases, to be discussed

      }
      color x black;
    } // end else
} // end RBRemove
```

Examples: deleting any red leaf from our example tree changes nothing.



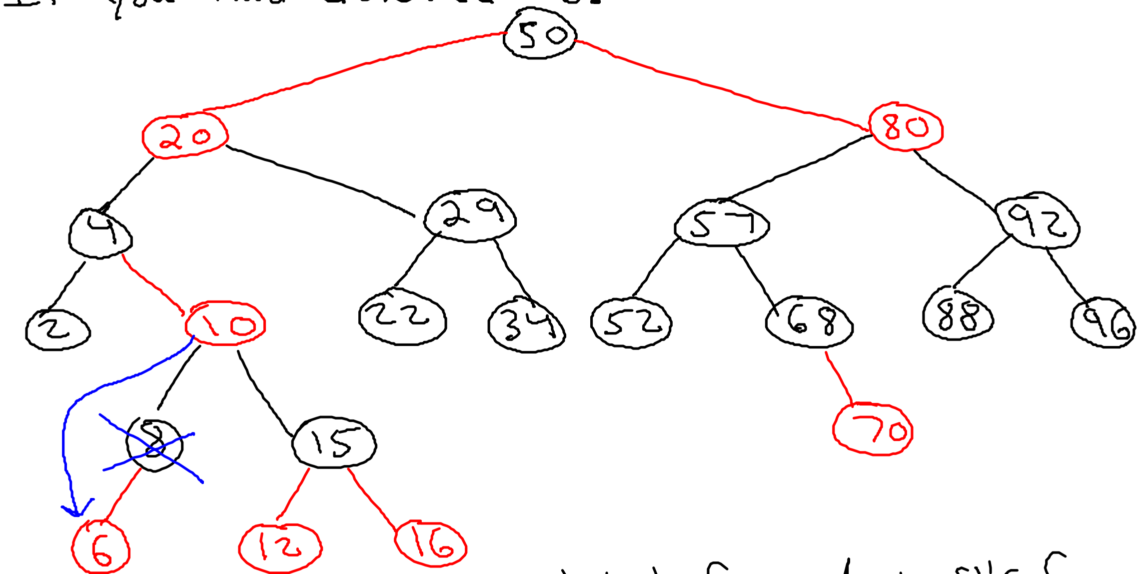
When you delete a red leaf you are done.

It's the same if your BST remove results in an IOS red leaf being deleted, such as when you remove 15:

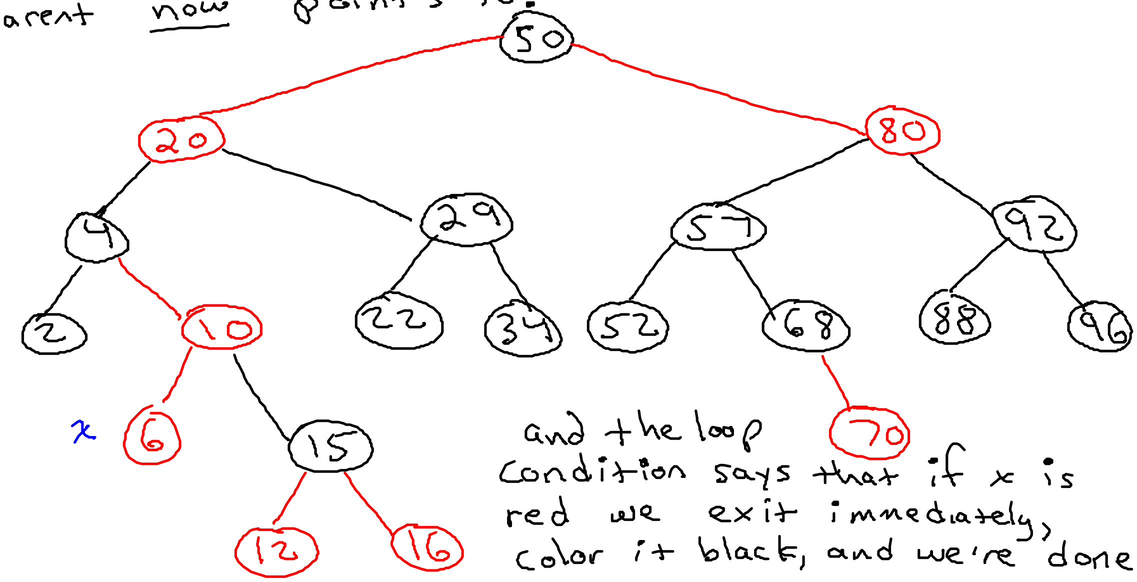


Ultimately, you still removed a red node so you are done.

If you had deleted 8:

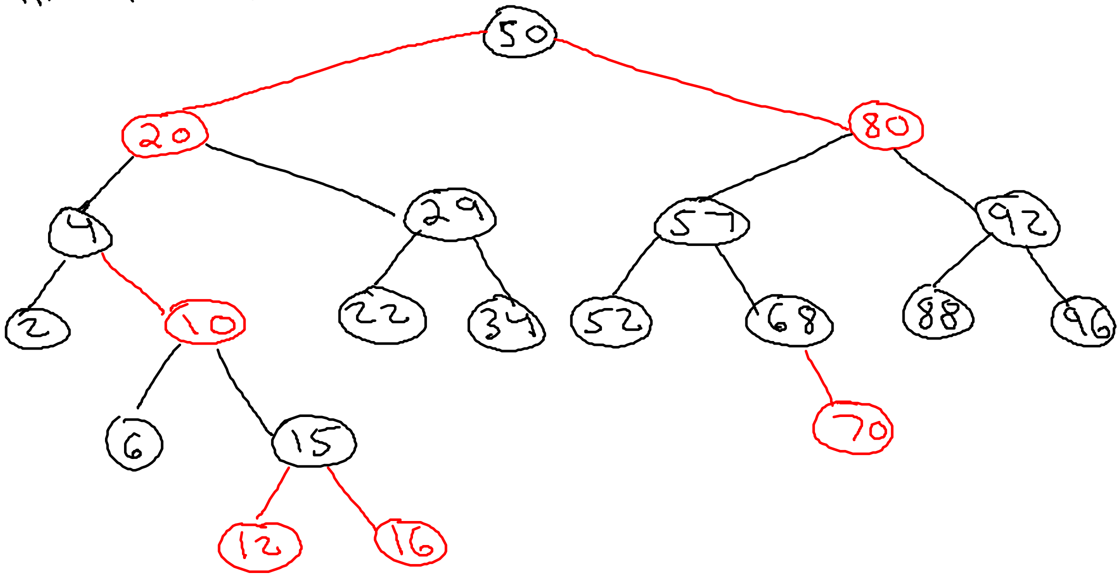


Then "x" becomes the label for what 8's former parent <u>now</u> points to:



and the loop condition says that if x is red we exit immediately, color it black, and we're done.

Final tree after removal of 8:



So those are some "trivial case" situations. The way we actually enter the loop is if:

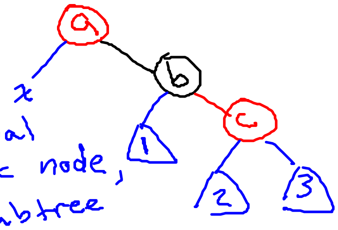
1) we delete a black node, and
2) it was a leaf

(If the deleted black node had 1 red child, well, that's the case we just discussed, where that red child would turn black. And there <u>can't</u> be a black node with just <u>1</u> child and that 1 child was black, that would be a black-height violation.)

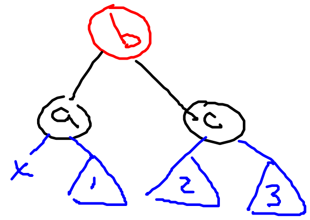
So, remember that the child of what we deleted (perhaps that child is NULL), we'll label "x". Consider these two scenarios:

SCENARIO #1

due to removal of black node, this subtree is short one black node



rotate and color change $\Longrightarrow$



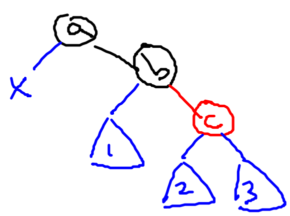
The path to 3 does not gain or lose any black nodes

The path to 2 does not gain or lose any black nodes

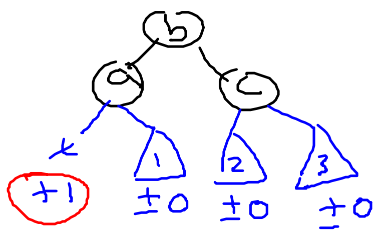
The path to 1 does not gain or lose any black nodes

$\Longrightarrow$ But the path to x <u>does</u> gain a black node, thus making up for our shortage of black nodes in that part of the tree.

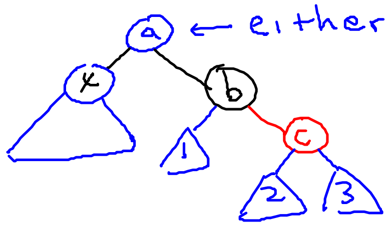
Likewise if the root of that subtree were black:



rotate and re color $\Longrightarrow$



What we are seeing is that, given x, if we match this pattern:



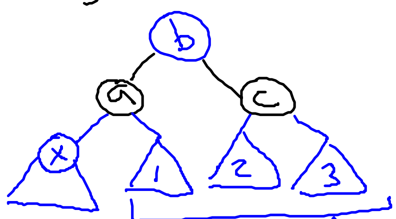
← either color

i.e.: 1) x's sibling is black
2) the child of x's sibling farther from x is red

Then you can perform this operation sequence:

1) the child of x's sibling farther from x — which is currently red — color it black
2) color x's sibling whatever color x's parent is
3) color x's parent black
4) single rotation on x's parent toward x

to fix your tree — i.e. to add a black node to the paths that need it without taking black nodes <u>away</u> from other paths.

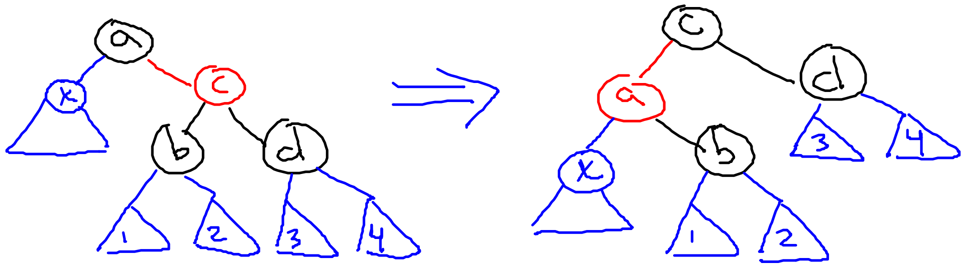


extra black on route to x

black height has <u>not</u> changed for these

So, we want to get to a point where we can apply that operation. First order of business is to make sure our sibling is black – if not, we need to change that.

CASE 1: x's sibling is red:



(Verify for yourself that we haven't changed the number of black nodes on <u>any</u> root-to-NULL path.)

Specifically, what we have done here is:

1) color x's sibling black
2) color x's parent red
3) single rotation on x's parent <u>toward</u> x

Now x's sibling (a different sibling than before) is black.

x's sibling might be black, but we still need a red node in the "sibling's child furthest from x" position in order to apply our ultimate repair pattern discussed earlier. Whether we had to apply case 1 first, or instead just arrived directly at this position, our remaining cases break down as follows:

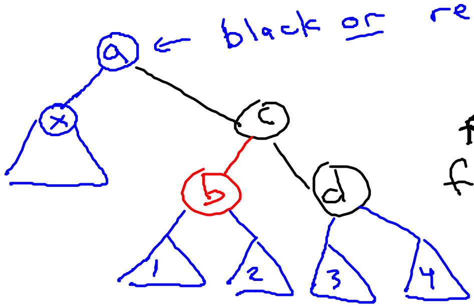
CASE 2: x's sibling has at least one red child

$\boxed{\text{Case 2a}}$: the red child of x's sibling is <u>not</u> where we need it to be; fix this and move on to case 2b.

$\boxed{\text{Case 2b}}$: the red child of x's sibling <u>is</u> where we need it to be to apply our repair pattern → apply repair pattern and we are done!

CASE 3: x's sibling has no red children → we will have to shift problem area further up tree and try again

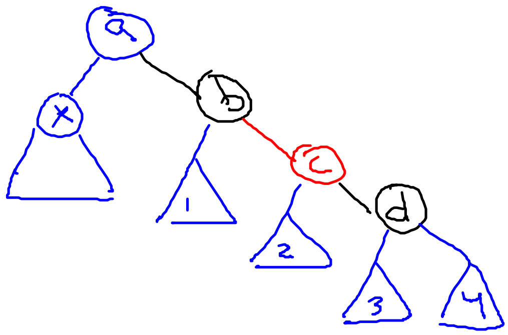
Case 2a:



← black or red, it doesn't matter

x's sibling is black and it has at least one red child; furthest of sibling's children from x is black, not red. (so other child of sibling of x must be red)

⇓



① color x's sibling red

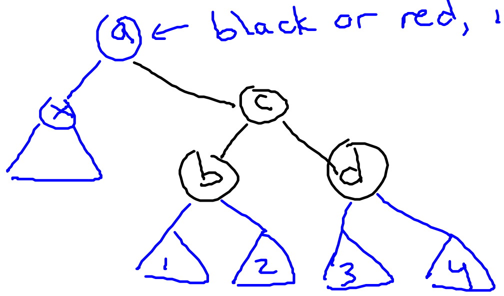
② color child of x's sibling closest to x black

③ single rotation on x's sibling away from x.

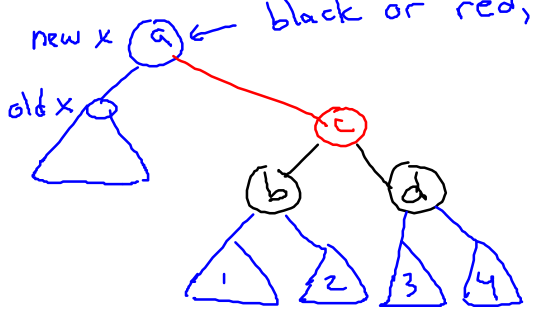
(Verify for yourself that no black heights are changed by this operation.)

Now we can move on to case 2b, i.e. the repair operation we discussed earlier.

case 3) if x's sibling is black but has no red children, we won't be able to apply the repair pattern here.



So, since x's subtree is short one black node, we'll color x's sibling red so that the other subtree of x's parent is short a black node as well. Then, since every path through x's parent is short a black node, we can relabel x's parent as x and run through the loop (including checking loop condition first) again.



(1) color x's sibling red

(2) label x's parent as "x"

All the paths through the sibling of the former x have lost a black node.

```
RB Remove (value)
{   BST removal
    if deleted node was red
        return;
    else  // deleted node was black
    {
      What the pointer that used
        to point to our deleted node
        now points to, label that "x".
      while (x not root and x is black)
      {  if (x's sibling is red)  // CASE 1
         {
            1) color x's sibling black
            2) color x's parent red
            3) single rotation on x's
                  parent toward x
         }
         // Now, x's sibling is definately
         // black.
```

if (at least one of x's sibling's children is red) // CASE 2

{

  if (x's sibling's child furthest from x is black) // CASE 2a

  {

    (1) color x's sibling red

    (2) color child of x's sibling closest to x black

    (3) single rotation on x's sibling away from x.

  }

  // Now, x's sibling's child furthest
  // from x is red. Run repair
  // operation to fix tree.

  (1) color child of x's sibling furthest from x black

  (2) color x's sibling whatever color x's parent is

  (3) color x's parent black

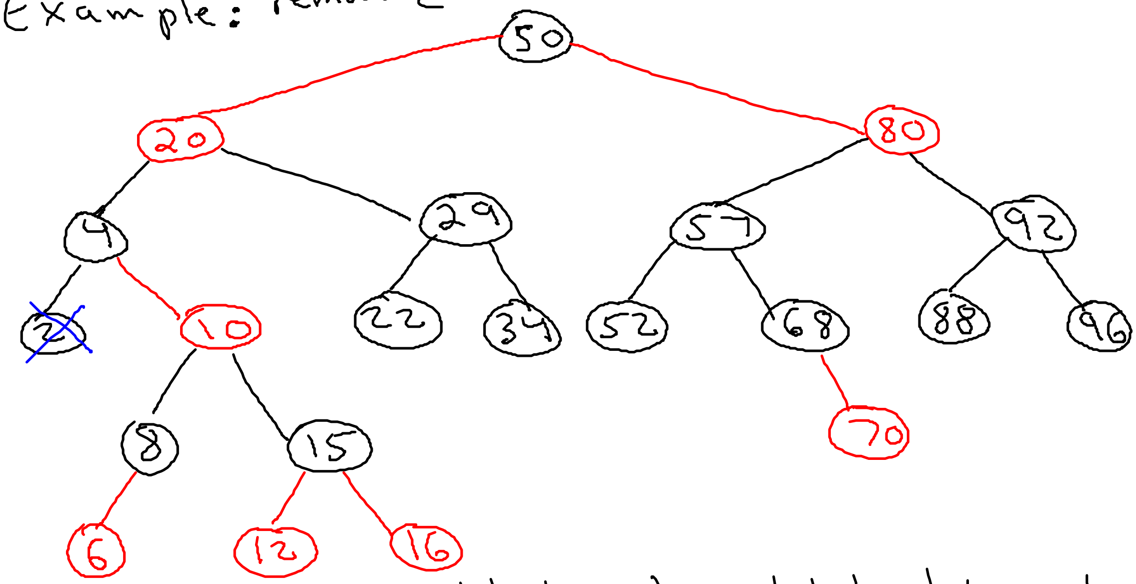
  (4) single rotation on x's parent toward x

after repair operation runs, we can break from loop, since tree is fixed

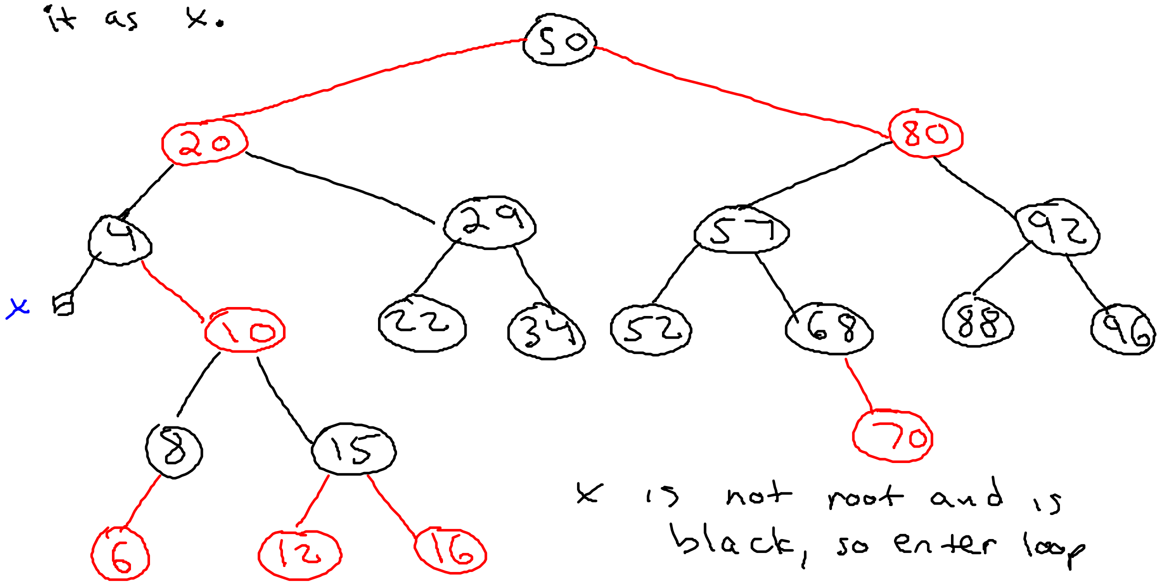
} // end if

```
else // CASE 3; x's black sibling has
{     // no red children
        1) color x's sibling red
        2) relabel x's parent as "x"
}
} // end while-loop
color x black;
} // end else
} // end RBRemove
```

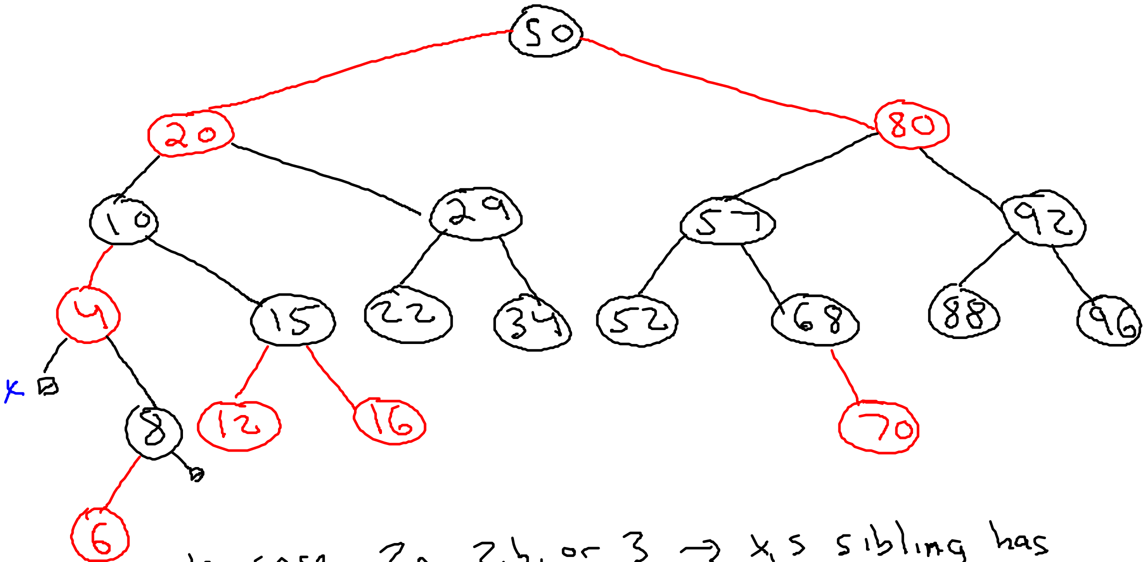
Example: remove 2



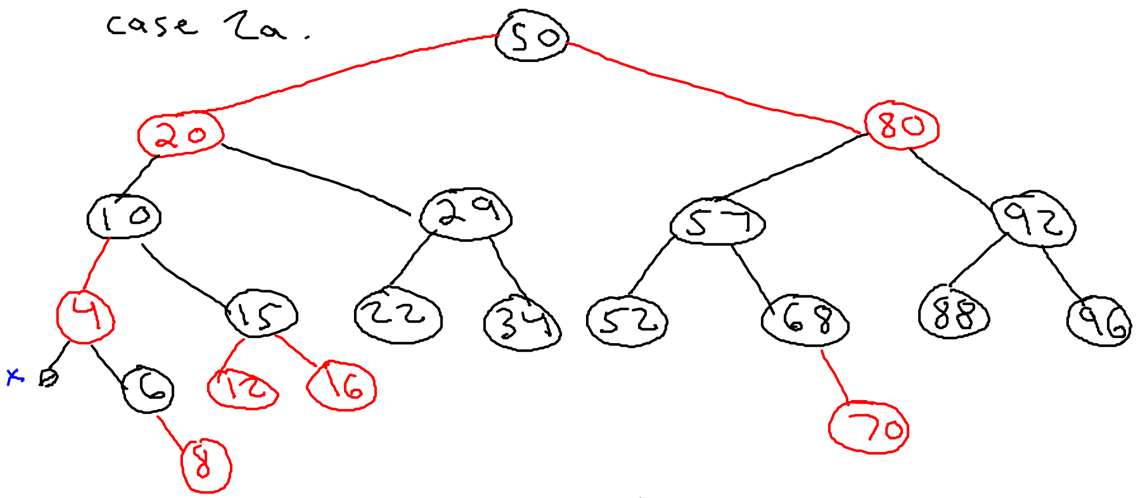
We are removing black node; label what replaces it as x.



x is not root and is black, so enter loop x's sibling is red → so run case 1

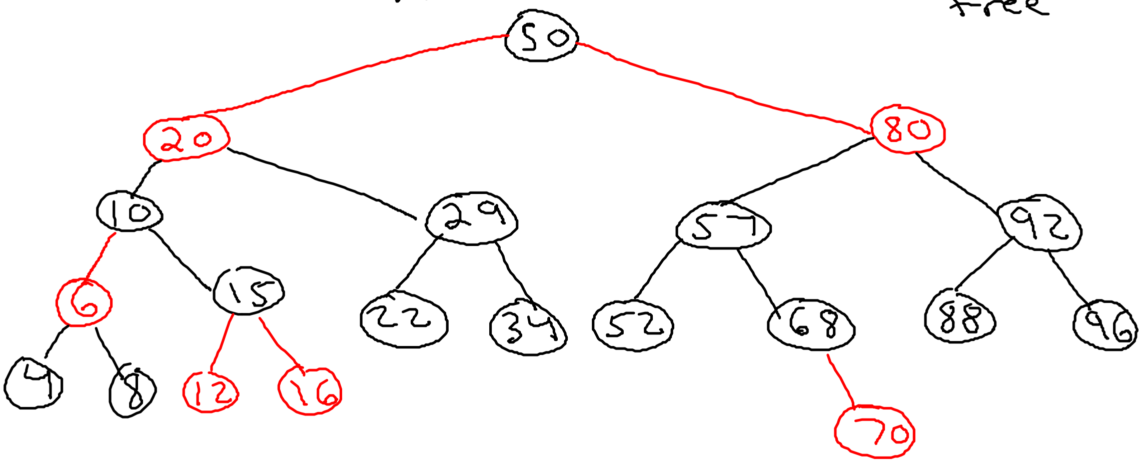


Now onto case 2a, 2b, or 3 → x's sibling has a red child but not in the right place, so case 2a.



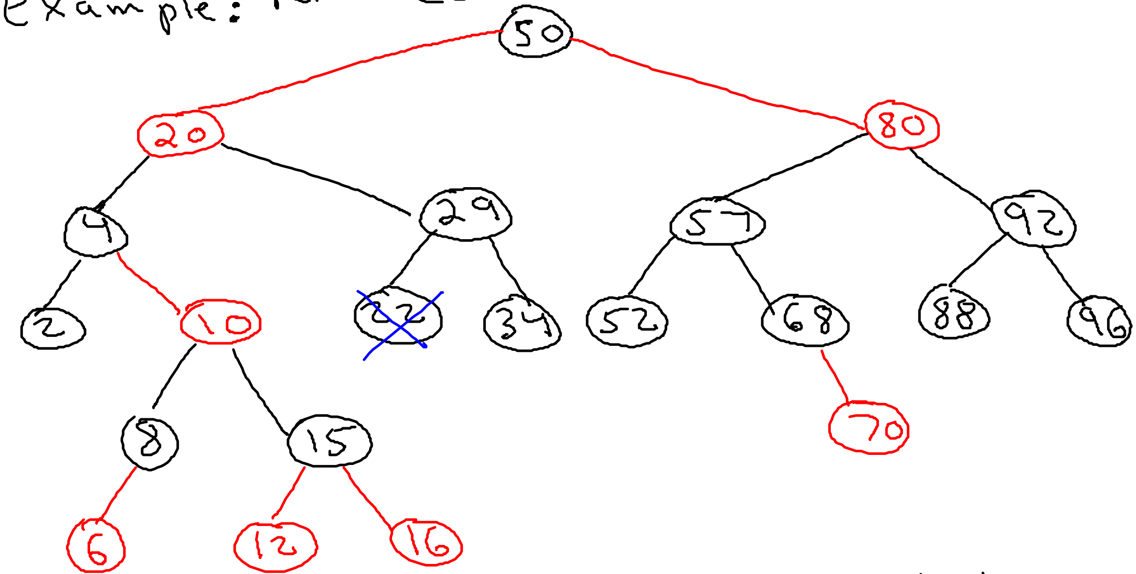
2a is always followed by 2b ...

result after applying case 2b to previous tree

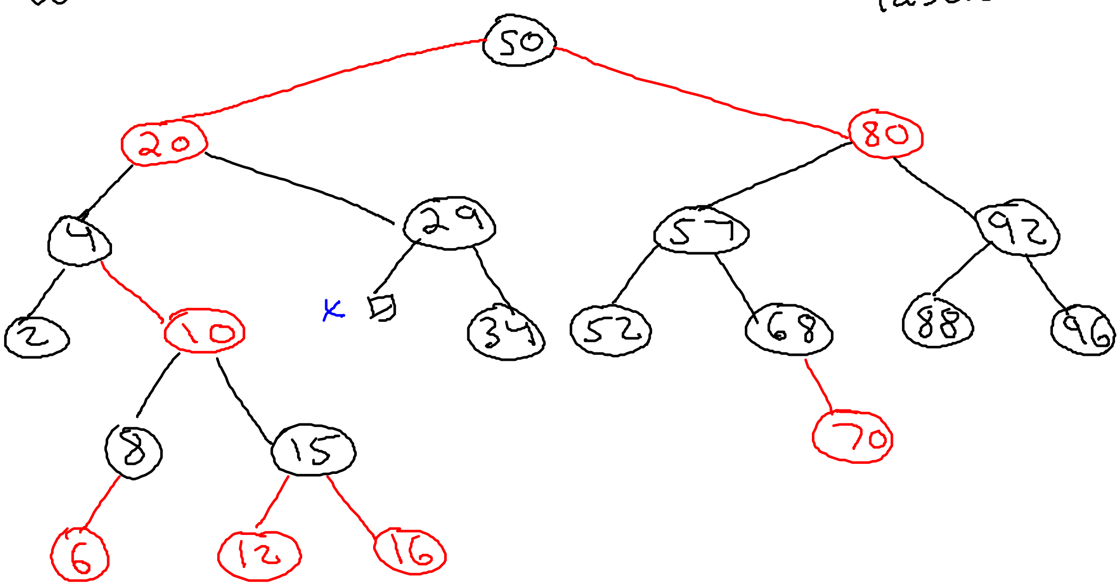


Tree is fixed!

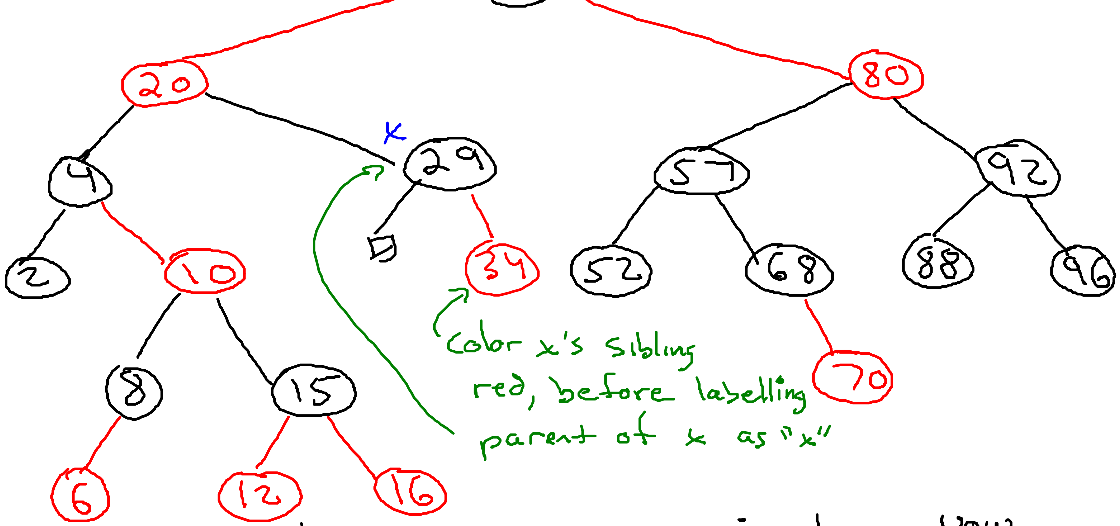
Example: remove 22



what ptr to former 22 <u>now</u> points to, is labeled "x"

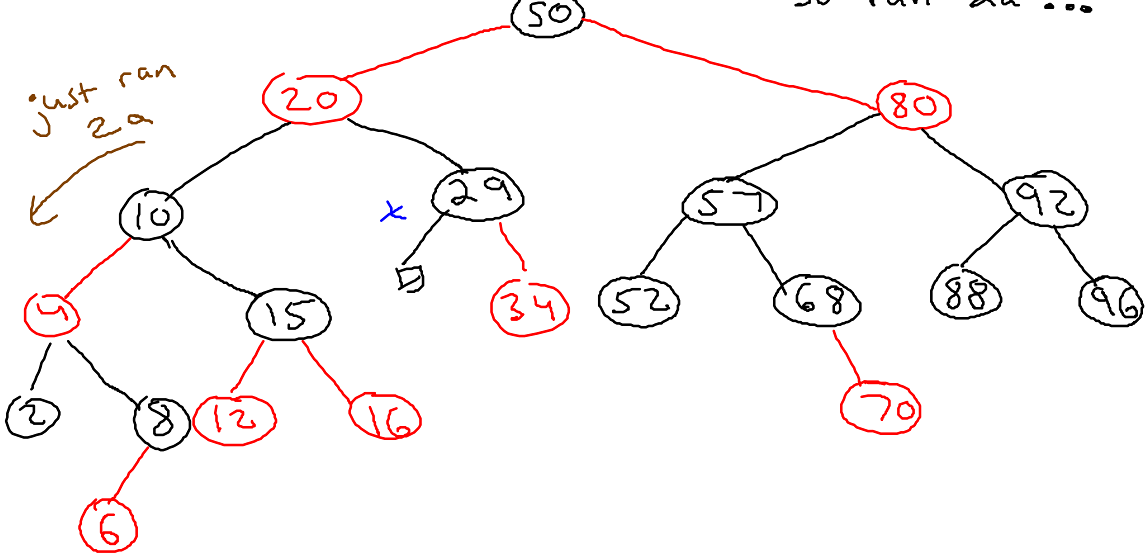


x's sibling is black and has 2 black children (both NULL) ; case 3

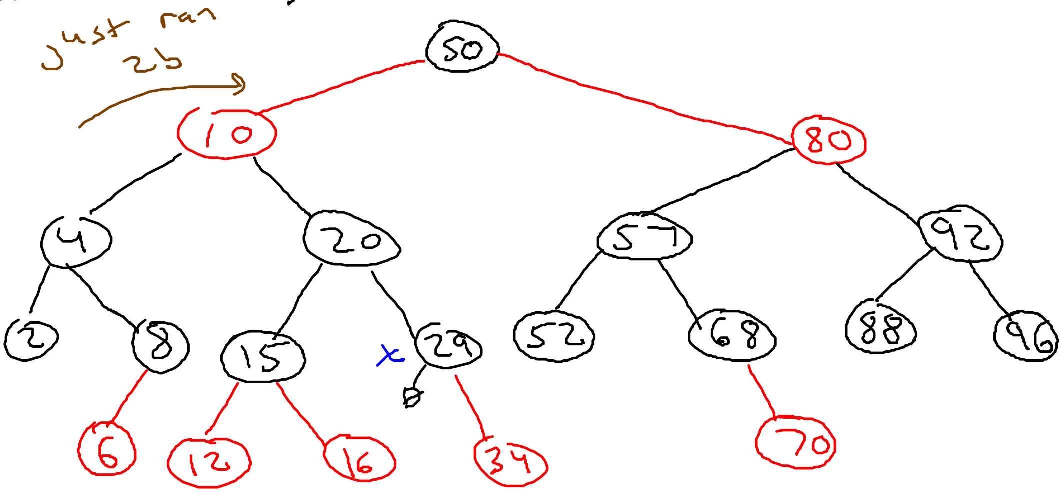


color x's sibling red, before labelling parent of x as "x"

x is still black, so can't exit loop. Now x's sibling is black with red child in wrong position. So run 2a ...



... followed by 2b.



Now we can break from loop; tree is fixed!