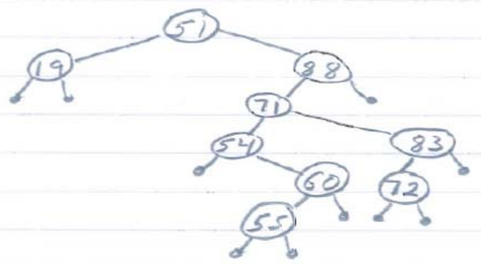
Red-Black Trees
Delete

Deleting an entry from a red-black tree works exactly the same as deleting from a binary search tree (BST) up to the point of fixing the tree once the node containing the entry is removed. Fixing the tree depends on the color of the node deleted as well as the colors of the surrounding nodes.

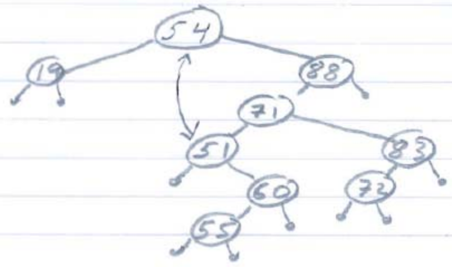
Here's a quick review of BST delete:

1. find the entry to delete
2. swap the entry with its left-most-right
3. delete the node containing the entry we want removed
4. reattach the right child of the node we removed as the left child of the parent.

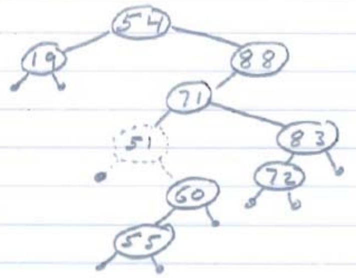
Example: Delete 51



swap 51 with 54

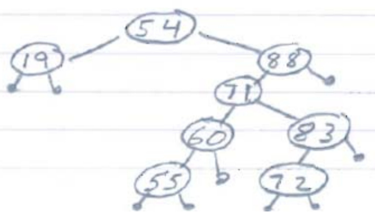


delete node containing 51



reattach 60 as the left child of 71

Note: This will always be a "right child" since 51 was a "left most" node



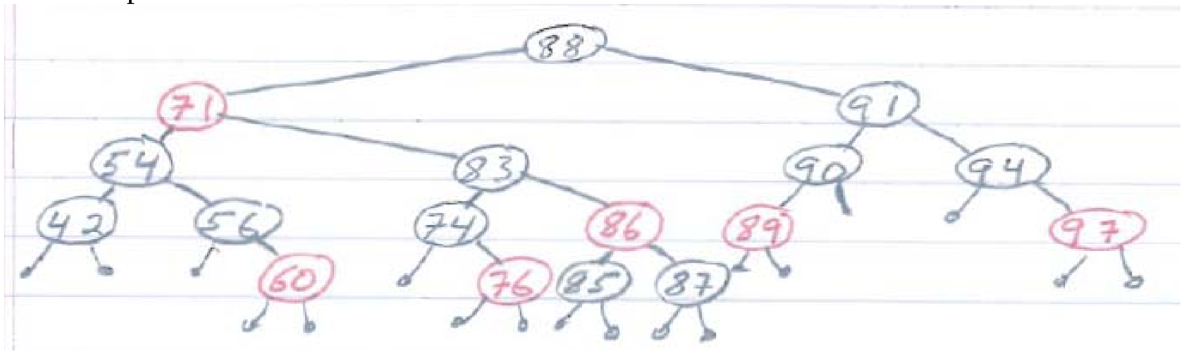
Deleting from a red-black tree

There are three general cases to consider:

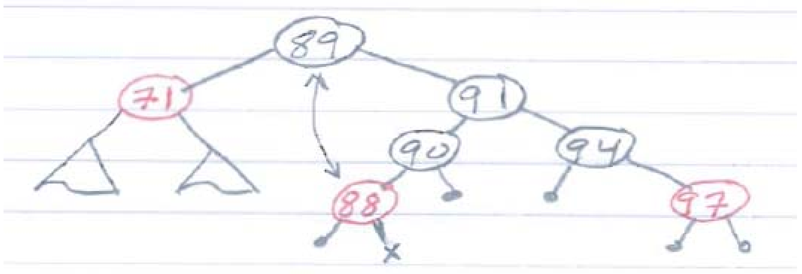
I. deleted node was red
II. deleted node was black with a red right child
III. deleted node was black with a black right child.

Case I: Delete a red node

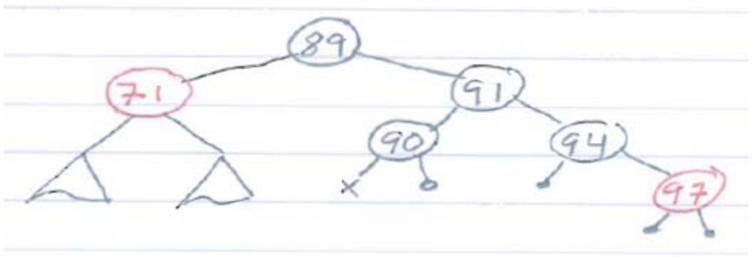
- This is the simplest case. Since the node was red, we don't need to do anything to fix the tree.
- Example: Delete 88



Swap 88 with `leftMostRight` …89



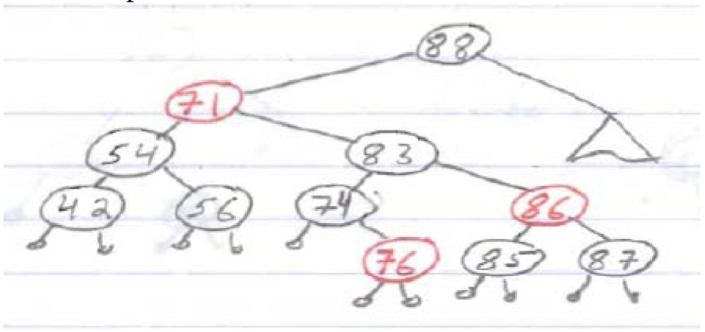
Remove the node containing 88 and its left child. The left child will always be a placeholder since we used `leftMostRight`. Reattach 88's left child.



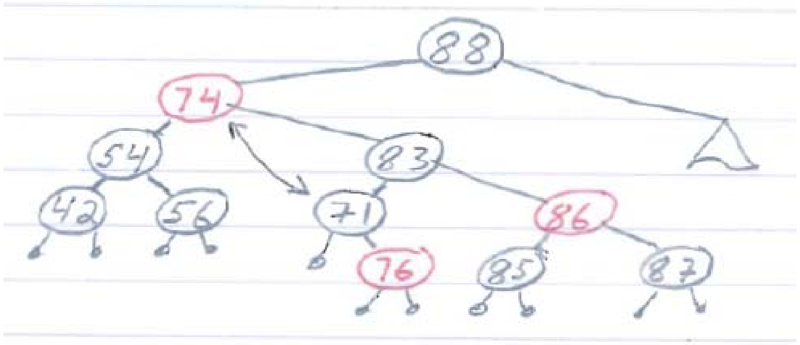
Since the node we removed was red, there is nothing else we need to do.

Case II: Delete a black colored node that has a red right child

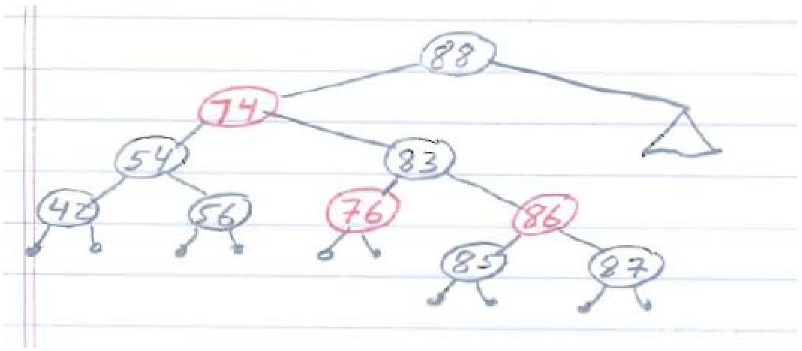
- Removing a black node will upset the black depth of the tree. However, since we are reattaching a red node in place of the black node we removed, we can simply recolor the red node as black to fix the black depth
- Example: Delete 71



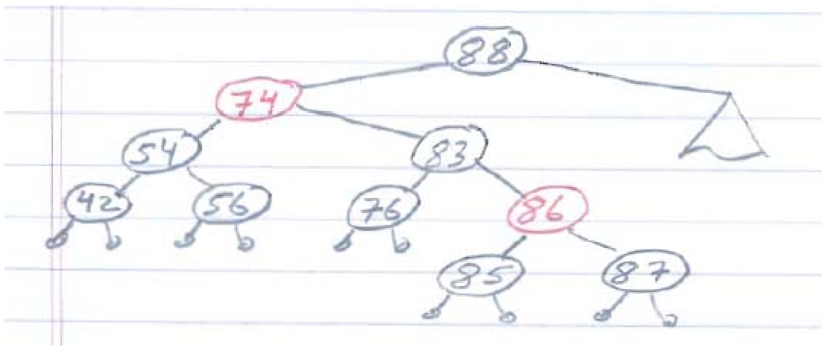
Swap 71 with `leftMostRight` …74



Remove black node containing 71 and reattach its right child (76) to 83.

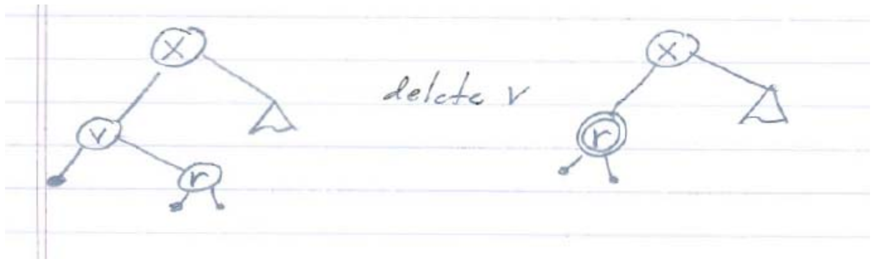


The black depth property has been violated by removing a black node. In this case, we can recolor 76 black to fix it.



Case III: Delete a black node with a black right child resulting in a "double black"

Now let's look at a situation where we delete a black node (v) with a black right child (r).

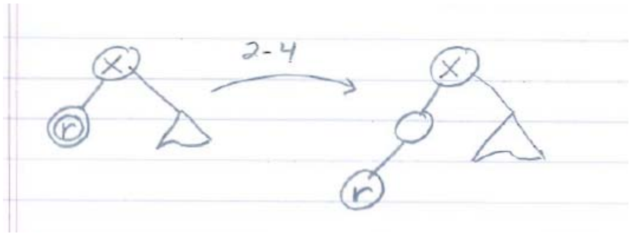


Since v is black and r is black, when we remove the node containing v we can't fix the black depth by recoloring r like we did previously.

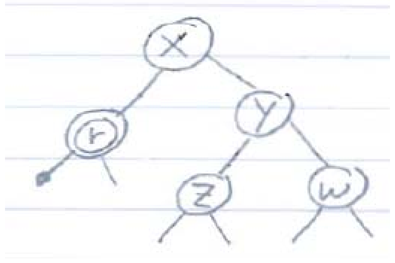
What we'll do is temporarily color r a fictitious "double black." This is simply a way to keep track of where we violated the black depth property.

What we do about fixing the double black depends on the neighboring nodes.

First, let's look at a double black in terms of a 2-4 tree. A double black is analogous to an underflow.



To fix an underflow with either transfer or fusion depending on the sibling node. The same applies here. The way we'll fix the tree will depend on the color of the sibling node (y) and also y's children.

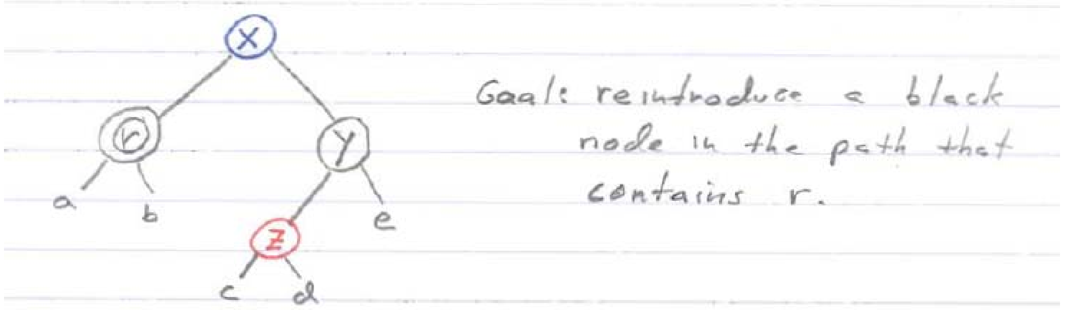


There are three situations we need to account for:

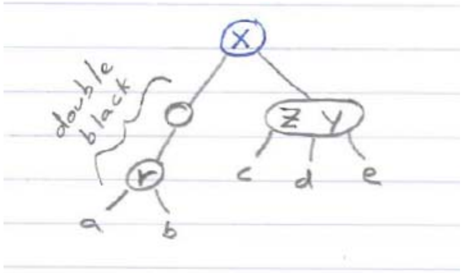
a. r's sibling (y) is black with a red child
b. r's sibling (y) is black with no red children
c. r's sibling (y) is red

Case IIIa: r's sibling (y) is black with a red child.

- in this case r is double black, y is black and y has a red child
- x is colored blue to indicate that x's color doesn't matter.

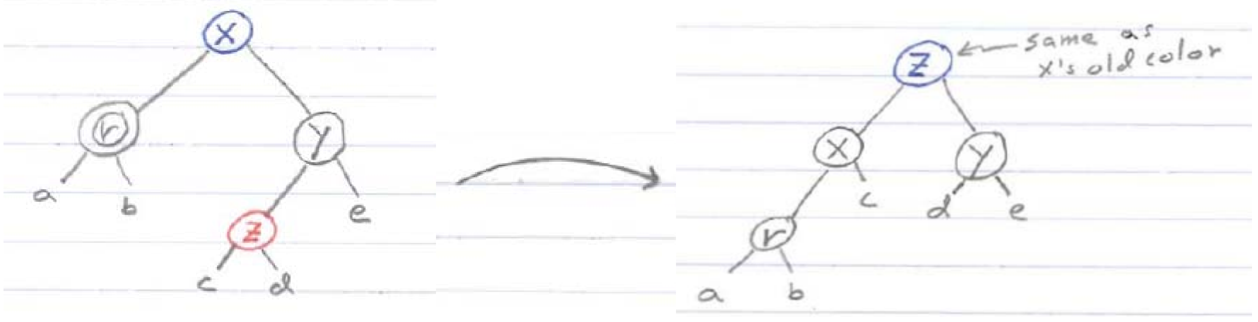


In terms of a 2-4 tree, the double black denotes an underflow.

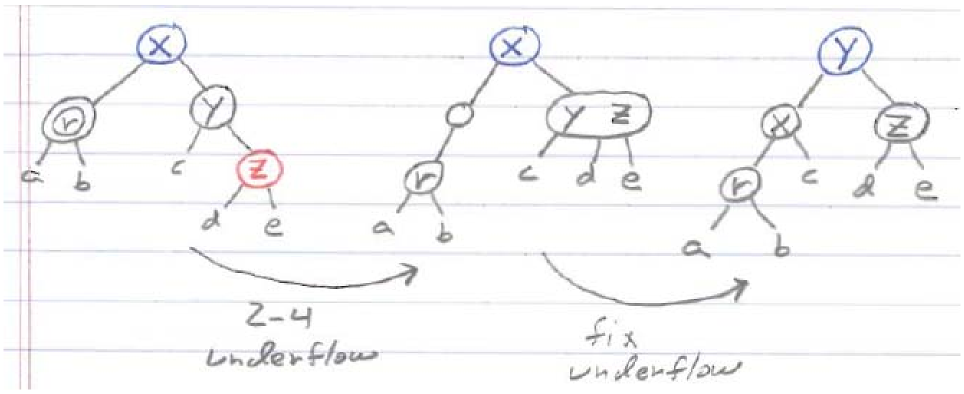


The underflow is fixed by dropping x to the empty node, promoting z to the node vacated by x and transferring z's left child (c) to become the right child of x.

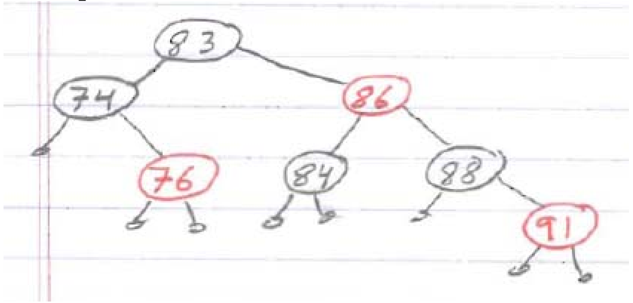
In essences, it is fixed by rotating and recoloring.



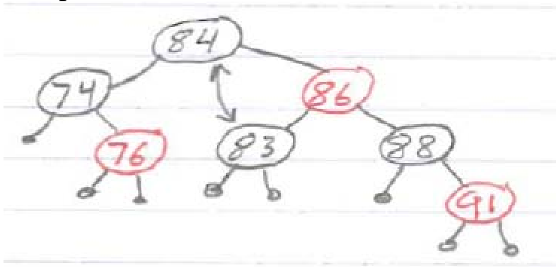
Likewise, if y's right child had been red instead of its left child, we would have:



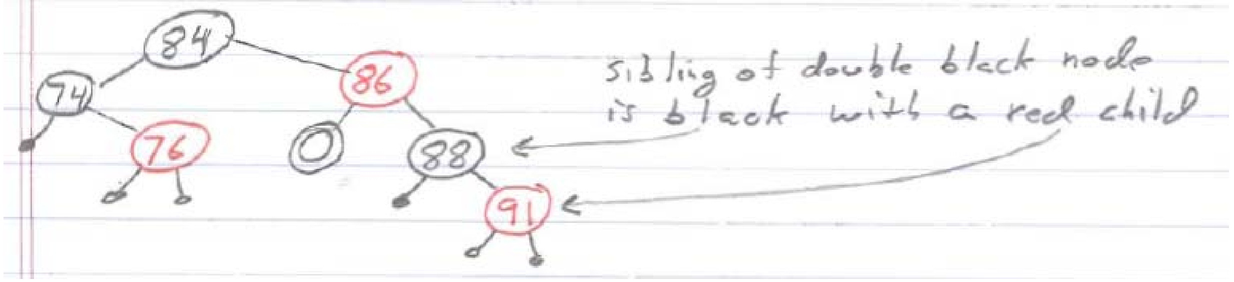
Example: Delete 83



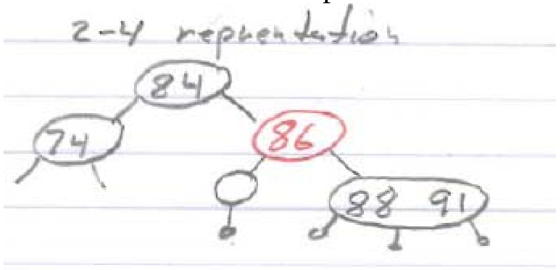
Swap 83 with 84



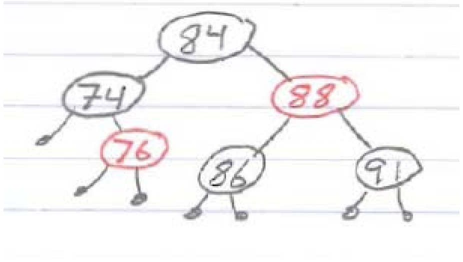
Remove node containing 83. The result is a double black because 83 was black and it's right child was black (remember, all external nodes are black).



We'll revert to the 2-4 representation to fix the tree.

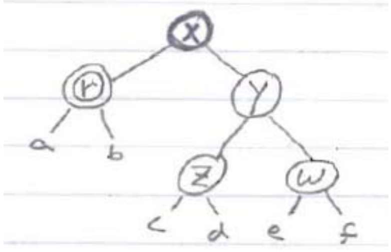


Using 2-4 tree operations, we fix the underflow using the *transfer* operation. Drop 86 to the empty node, promote 88 to the empty node vacated by 86 and transfer 88's left child to become the right child of 86. Note: the node vacated by 86 maintains its original color.

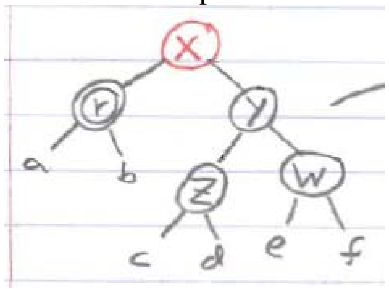


Case IIIb: r's sibling is black with no red children

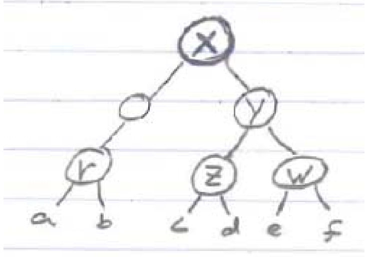
- r is double black, y is black and both children are colored black
- x's color <u>will</u> play a role in the final outcome
- we will fix the double black by simply recoloring the nodes with no restructuring involved.
- this recoloring <u>may</u> propagate the double black up the tree



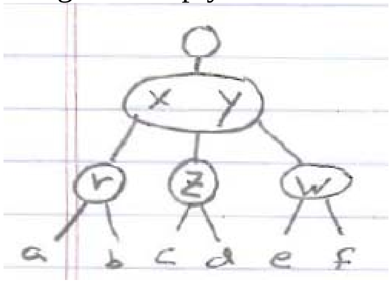
or



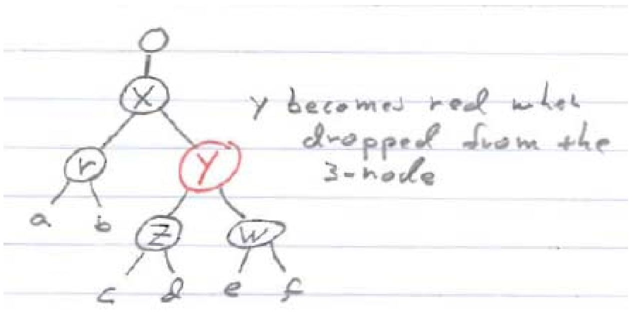
Converting this to a 2-4 tree we see that we have an underflow where the sibling of the underflow node doesn't have extra entries that we can steal.



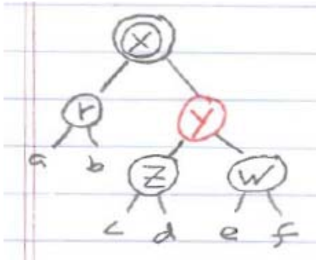
If you remember back to your 2-4 tree operations, we fix this with *fusion*.
Merge the empty node with its sibling (y) and drop x to fill the empty entry in the merged node.



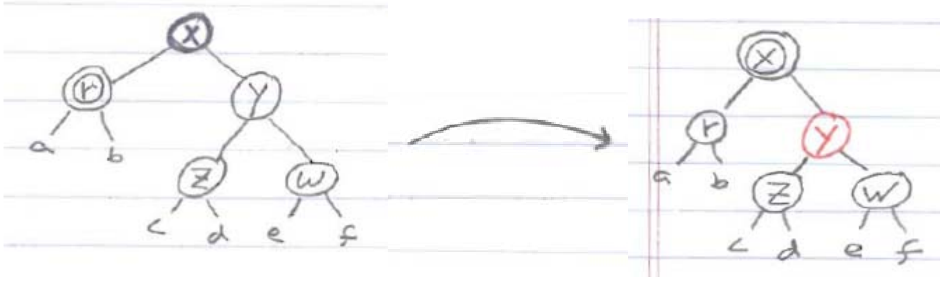
When we convert this back to a red-black tree, y becomes red when it is dropped from the 3-node it was sharing with x.



Notice that this leaves an underflow in the node vacated by x, which makes x double black.

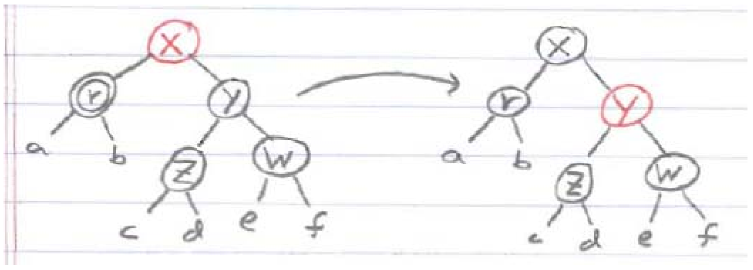


The end result was a recoloring of nodes with no structural changes; y became red and the double black was propagated up the tree from r to x.



What if the double black reaches the root?
This recoloring propagated the double black up the tree just like passing an underflow up the tree with 2-4 trees. If the double black reaches the root, it is fixed by recoloring the root black instead of double black. Since it is the root node, the black depth of every external node in the tree is decreased by one, so the depth property of the tree is maintained.

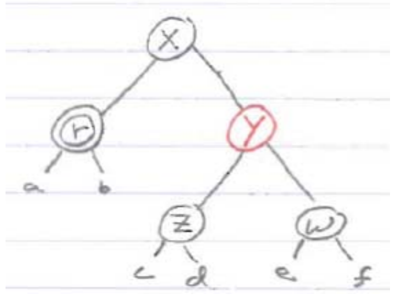
What if x were red instead of black?
In the above example x started out black and ended up double black. If x starts out red then everything stays the same, except that when we propagate the double black from r to x, x becomes black instead of double black.



Case IIIc: r's sibling is red

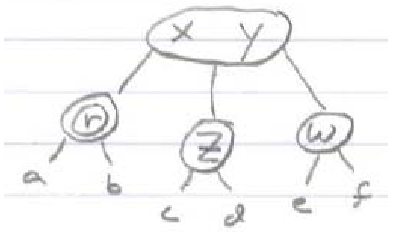
- r is double black and y is red

- x will always be black. If it wasn't then we would have had a red-red situation with both y and x being red
- The goal is to restructure this tree so that r ends up with a black sibling instead of red

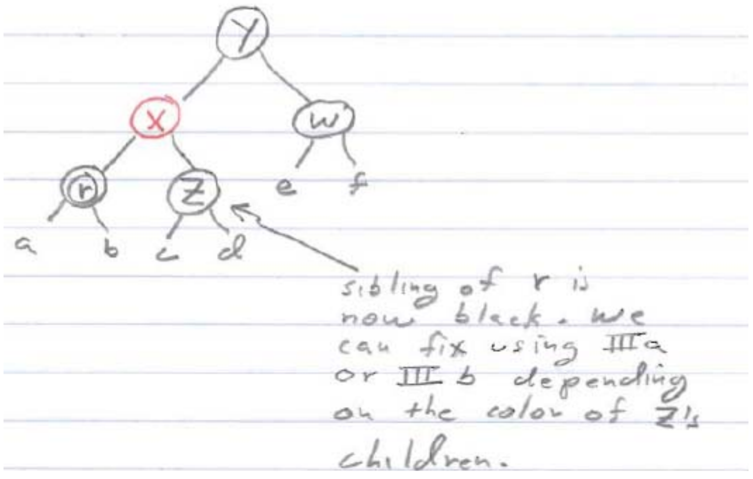


We're not going to fix this directly. Instead, we are going to restructure the tree by rotating around y and recoloring so that we end up with r's sibling being black. Once r's sibling is black, we can fix the tree using case IIIa or IIIb above.

If we convert this to a 2-4 tree, x and y come together to make a 3-node.

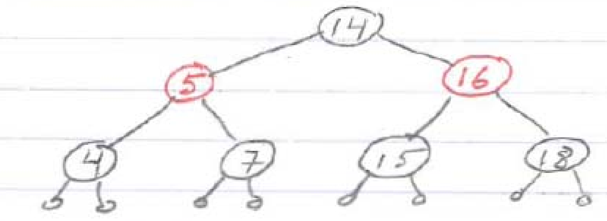


We now convert this back to a red-black tree by dropping x from the 3-node. The key to the restructuring is to bring y up and merging it with x, then dropping x (instead of y). By dropping x, r's sibling now becomes z which is a black node.

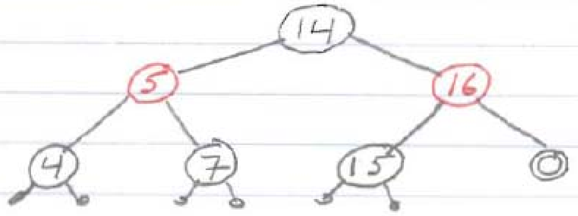

y
x
w
r
z
e
f
a
b
c
d
sibling of r is now black. we can fix using IIIa or III b depending on the color of Z's children.

Depending on the color of z's children, we fix the tree using the algorithm described for case IIIa or IIIb as appropriate.

Example:

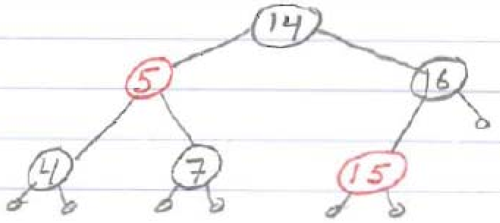


delete 18

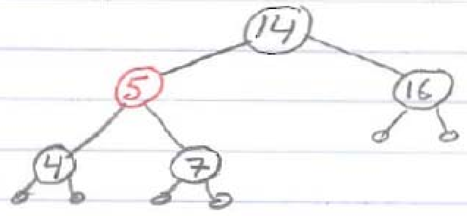


sibling is black
with black children
Case III b: recolor

recolor
16 → black
15 → red

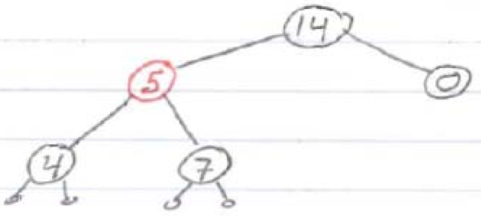


delete 15



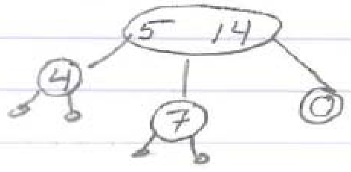
Deleted red node.
No fix necessary

delete 16

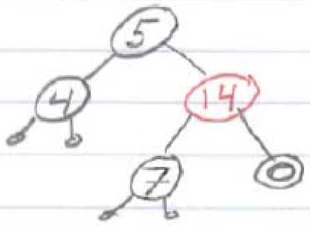


- deleted black node with a black right child → double black
- sibling of double black is red
Case III c: restructure to get a black sibling

Convert to 2-4 Tree

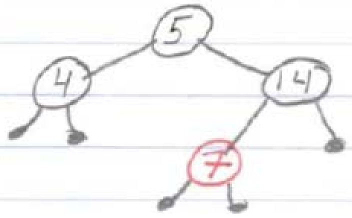


Convert back to Red-Black Tree



Sibling of double black is black
with black children
Case IIIb: recolor

recolor
14 → black
7 → red



Red-Black Tree Performance

find: O(log n). Worst case, find must search from the root to an external node. Since the height of the tree is O(log n), find is O(log n).

insert: O(log n). All new entries are added at leaf nodes so it takes O(log n) to get to the location where the entry will be added. If adding the node introduce a red-red situation it must be fixed. Typically the fix will occur locally in O(1) time. However, it is possible that the fix will cause the red-red to propagate up the tree. Worst case, the red-red could propagate all the way to the root which would require log n fixes. Since each fix is O(1) the total time to fix the tree is worst case O(log n).

delete: O(log n). Finding the entry to remove and swapping it with it's leftMostRight could take O(log n) time. Removing the node is a constant time operation. The work required to fix the tree once the node is removed depends on the specific situation. In most cases, it takes constant time O(1). There is the possibility for a double black to propagate up the tree tree to the root, in which case fixing the tree takes O(log n).