2/6/2018 (1)

Deletion in a Red-Black Tree

Initially, we will delete a node just like we delete a node in a normal binary search tree. Here are the cases we will look at:

1) ~~Red leaf node~~
2) Black node, with one red child node
3) Black leaf node

In the first case, the normal binary search tree delete is sufficient. Removing a red node does not change the black depths of any node, nor create a red child for any red node.

In the second case, after we complete the binary search tree delete, we must simply recolor the child node of the deleted node to black. Changing this color adds one to the black depths of each node in the subtree of the deleted node, restoring the equality of the black depths of all external nodes. Also, changing a node to black does not violate any of the other Red-Black Tree specifications.

Neither of the strategies mentioned above are adequate for dealing with the third case. Instead, when we patch in the child of the deleted node in this case, in order to temporarily preserve the black depth property, we will color this child node a fictitious "double black" color.

Before we discuss how to deal with "double black" nodes, let's real quickly justify why the cases above are the only cases we will deal with. First off, we will only delete nodes with 0 or 1 child. Neither colored node can have one black child. If it did, the black height of the node's null child would not be proper. Further more, a red node can NOT have a red child. These observations narrow the cases to the situations listed above.

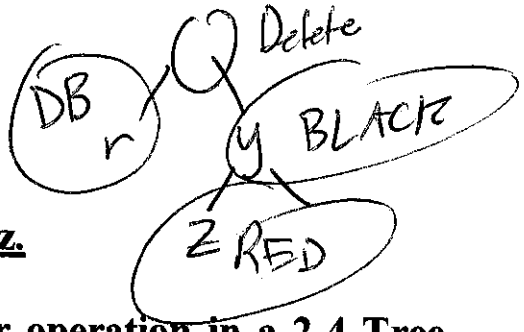
The remaining cases with this "double black" node can be categorized as follows:

1) The sibling of the "double black" node is black and has a red child.
2) The sibling of the "double black" node is black and both children are black.
3) The sibling of the "double black" node is red.

(Note that initially, even though the double black node is a null node, after starting the recoloring/restructuring process, we may create a double black node that is NOT null.)

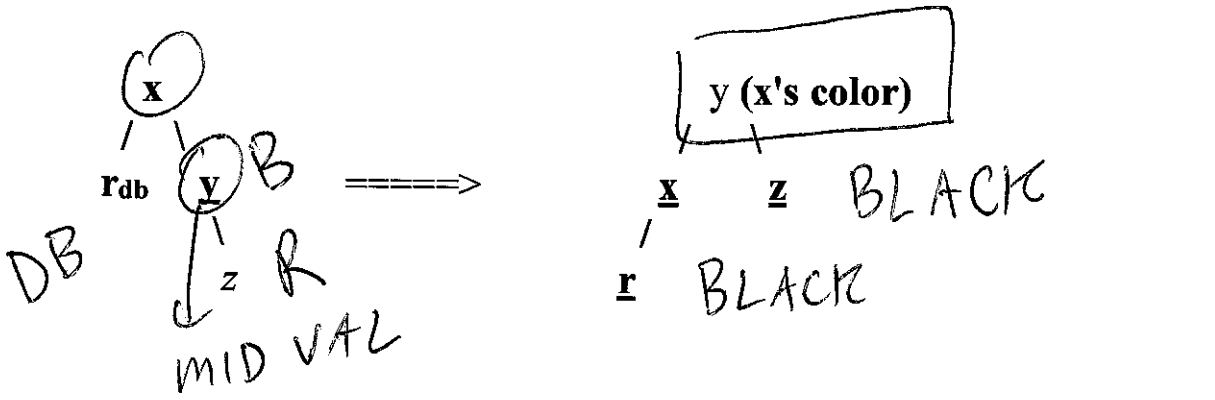
To deal with each of these situations, let's first set up names for all of the important nodes:

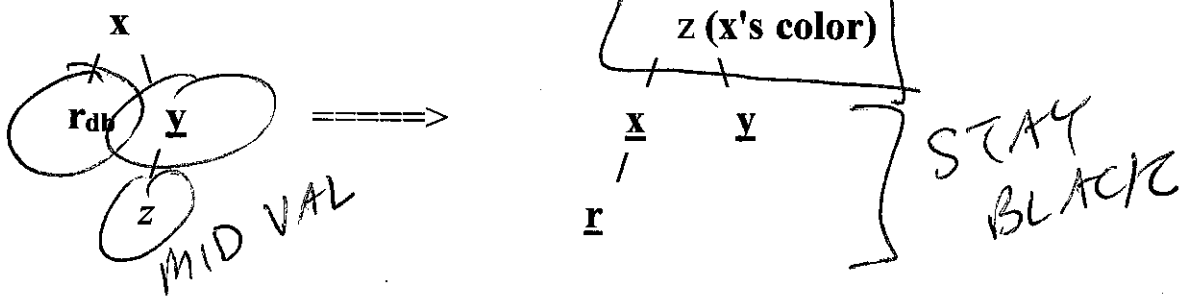
1) Let the child of the deleted node, which is colored "double black" be r.
2) Let y be the sibling of r.
3) Let z be a child of y, in each case, the specific child will be designated.
4) Let x be the parent of y.

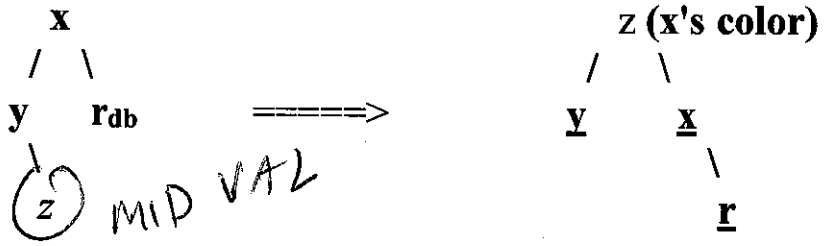


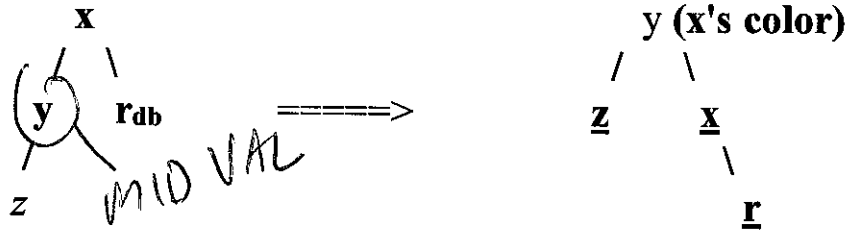
<u>Case 1: y is black and has a red child z.</u>

This case corresponds to the transfer operation in a 2-4 Tree delete. Take the nodes, x, y, and z and relabel them a, b, and c, in their inorder ordering. Place b where x used to be, and then have a and c be the left and right children of x, respectively. Color a and c black, and color b whatever color x USED to be. This eliminates the "double black" problem, so we can stop here.





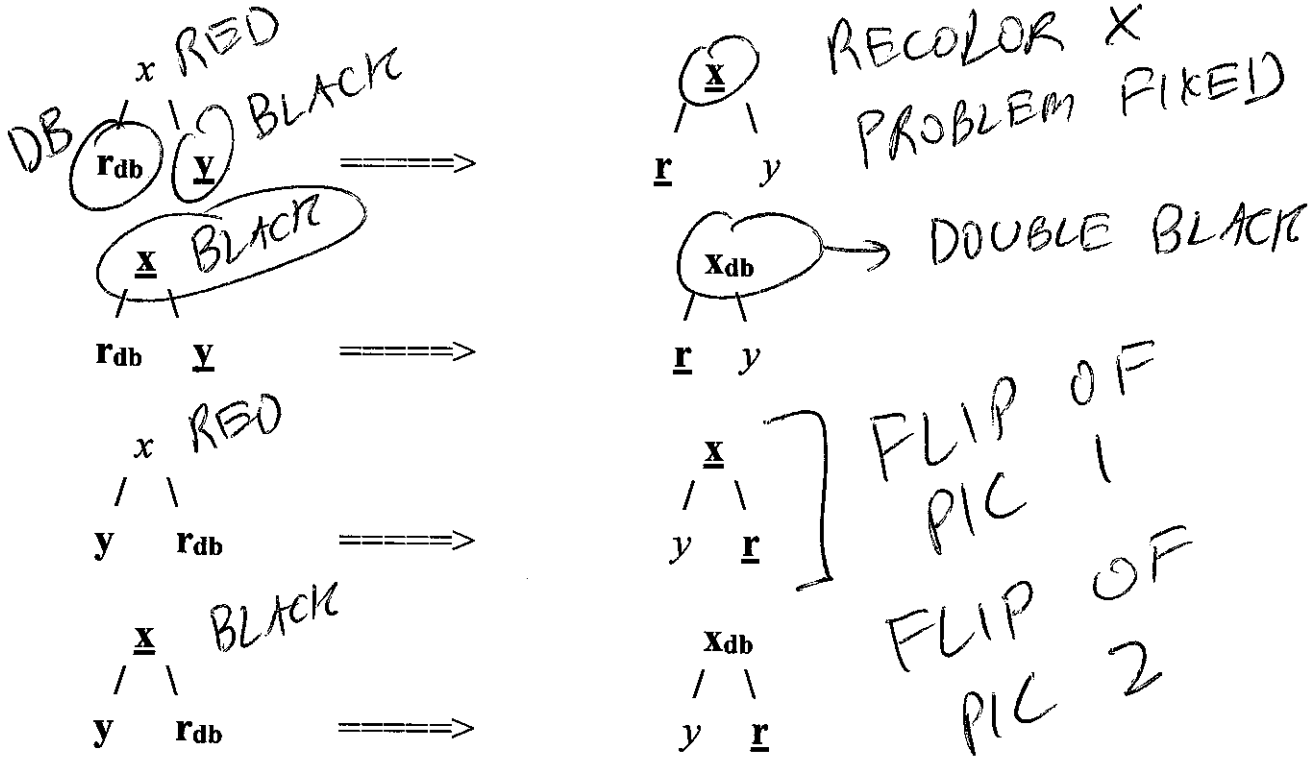




Notice how in each of these situations, we have been able to eliminate the double black node, but maintain the "black depth" of each external node in the tree.

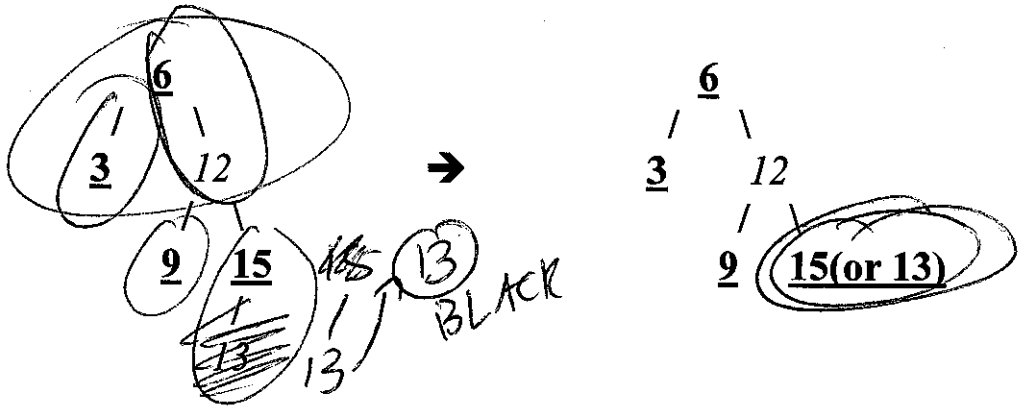
Case 2: y is black and both of its children are as well.

This case corresponds to a fusion operation in a 2-4 Tree. We deal with this case by just recoloring, instead of making any structural changes to the tree. In particular, we will color r black, (changing it from "double black") and then color y red. What this does is subtract one from the black depth of every external node in the subtree of x. To compensate for this, we must change x from red to black. BUT, this only works if x was red to begin with!!! If it's not, to maintain the "black depth" at the external nodes in the subtree rooted at x, we must color x "double black." In essence, if this occurs, we have pushed the "double black" node up the tree, much like a fusion operation can propogate another fusion operation.



Examples of Red-Black Tree Deletions

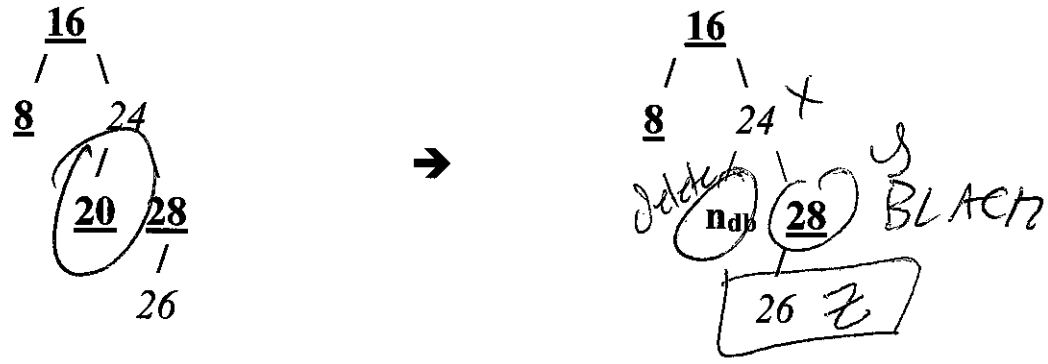
The first case is an example of cases 1 and 2 w/o any double black nodes. It corresponds to deleting from a 3 or 4 node in a 2-4 Tree. Delete either 13 or 15 from this Red-Black Tree:



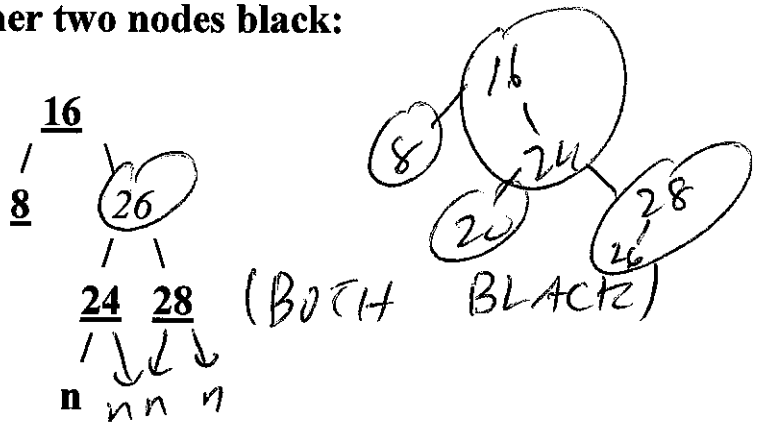
As mentioned in the beginning of the lecture, we simply do the normal binary search tree delete and color the child of the deleted node black. (If you delete 13, you don't need to do this.)

The corresponding 2-4 Tree delete is as follows:

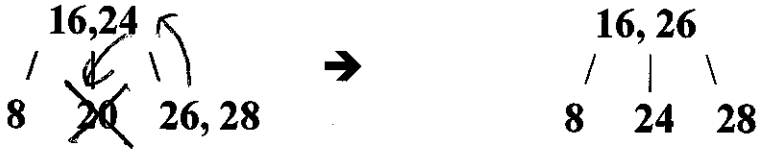
<u>Case 1: Double black node's sibling has a red child.</u>
Delete 20 from the tree below:



In this case, label x as 24, y as 28, and z as 26. We want to rearrange these nodes so that 26 is at the root of this subtree and color it red, the old color of 24. Also, we will make the other two nodes black:

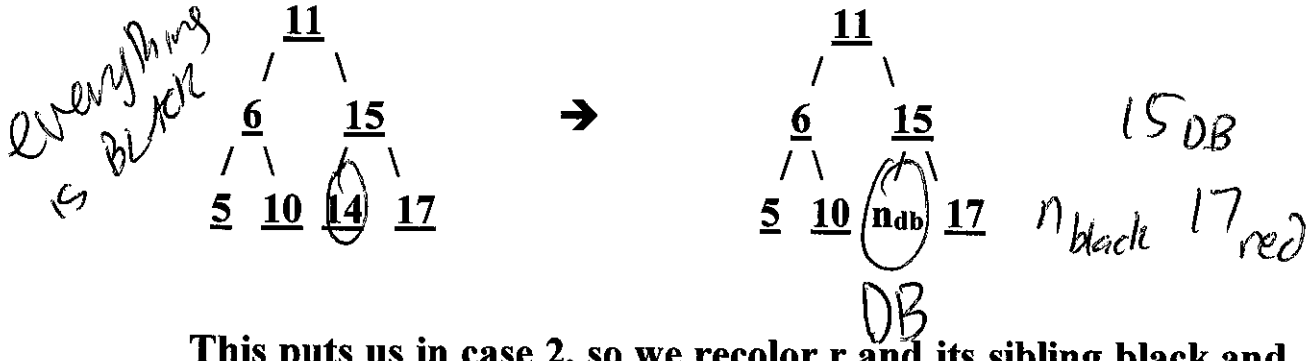


The corresponding 2-4 Tree Delete (a transfer from an adjacent sibling) is as follows:

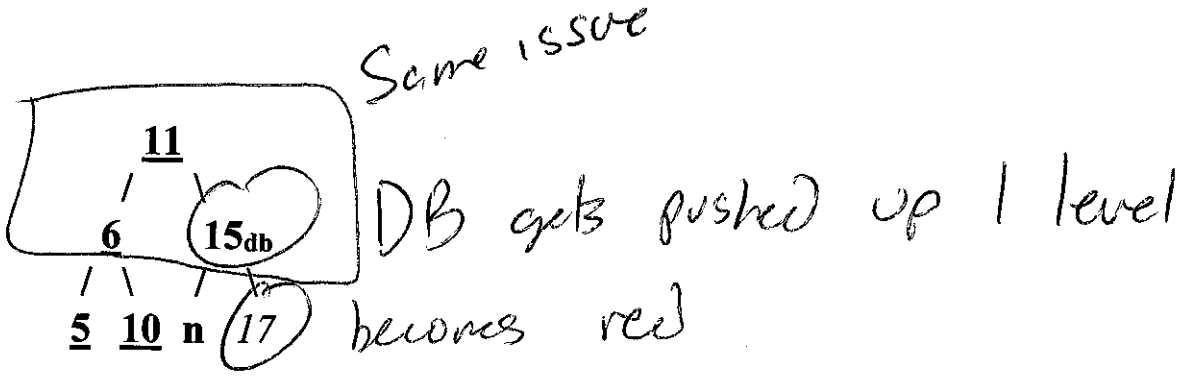


Case 2: Double black node's sibling has 2 black children

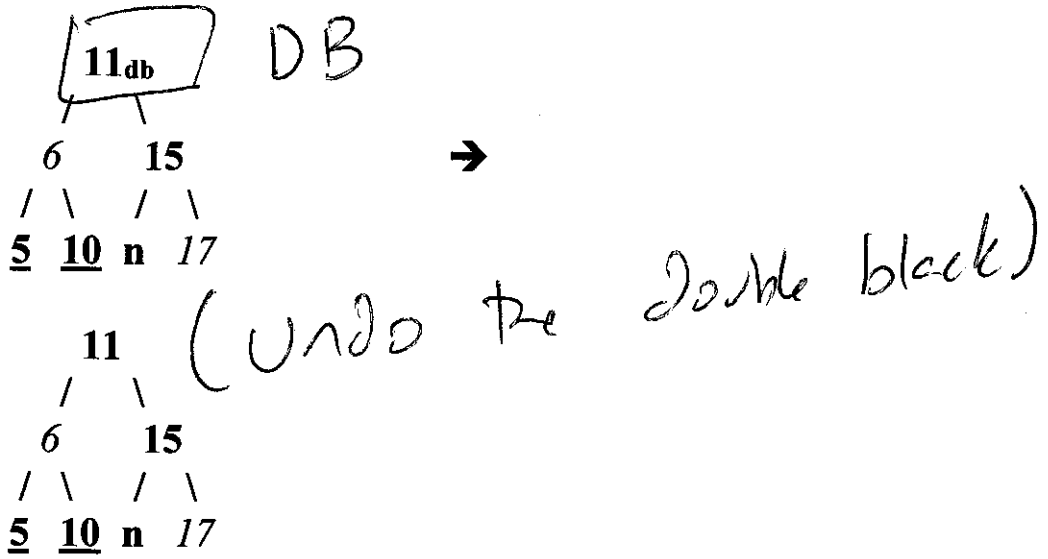
Finally, let's look at an example from case 2, which includes a propagation of the double black. (You should be able to tell from this example in what situation the double black would NOT be propagated.) Delete 14 from the tree below.



This puts us in case 2, so we recolor r and its sibling black and red respectively, pushing a red up to the parent node with 15:

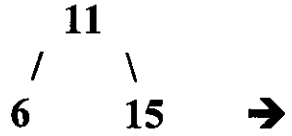


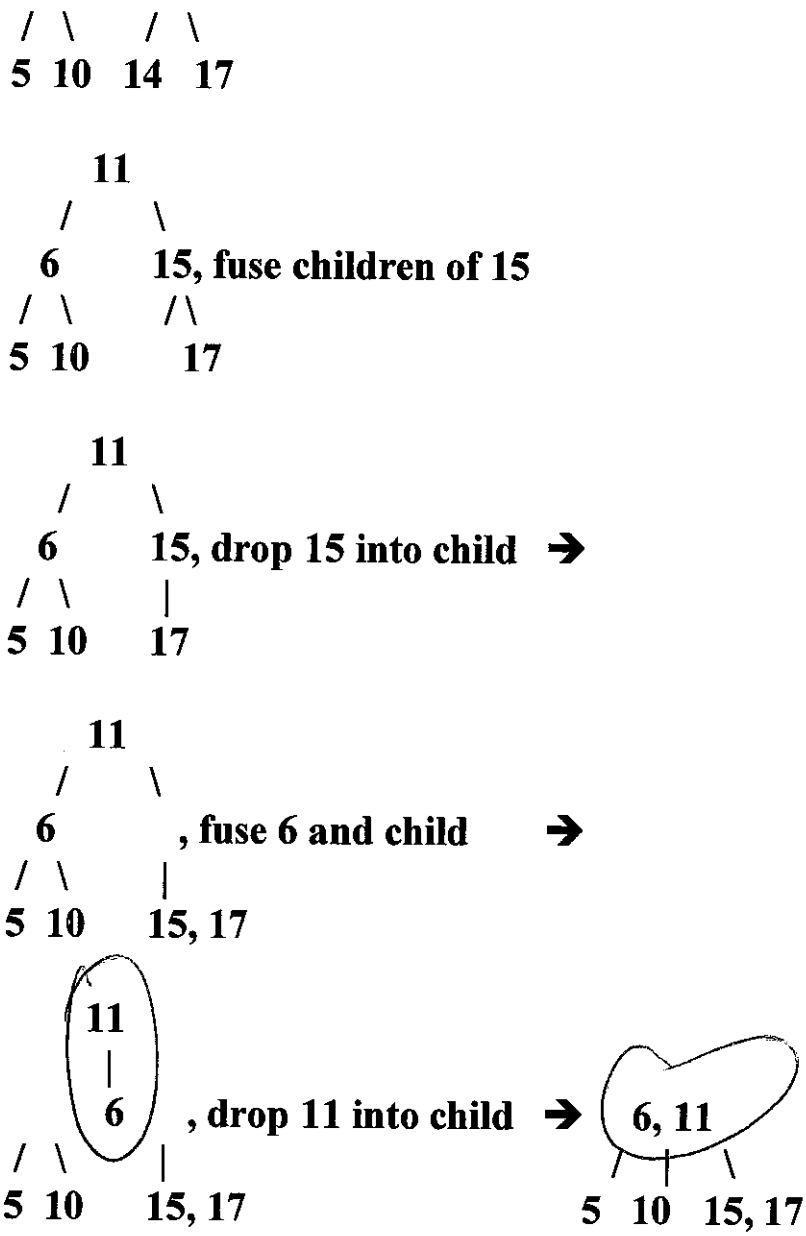
Now, we have pushed the double black node up a level in the tree. We see this puts us in case 2 again:



There's no need to have a double black root. Just color it back to black since this will subtract one from the black height of each path. As you might imagine, this case corresponds to the one in the 2-4 Tree that fuses nodes and drops parent nodes into the fused nodes. If we leave an empty root, our 2-4 Tree height drops by one, as our black height in this case decrements by one as well.

Now, let's look at the corresponding 2-4 Tree Delete:

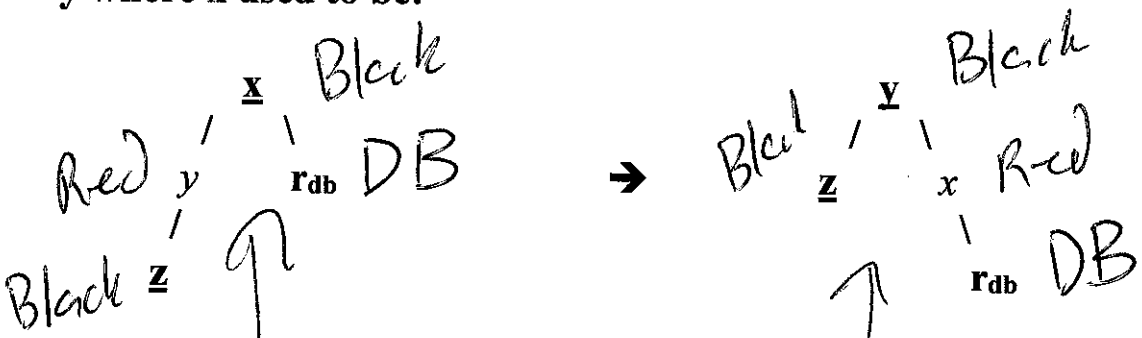




and drop empty root node (~changing db root to black.)

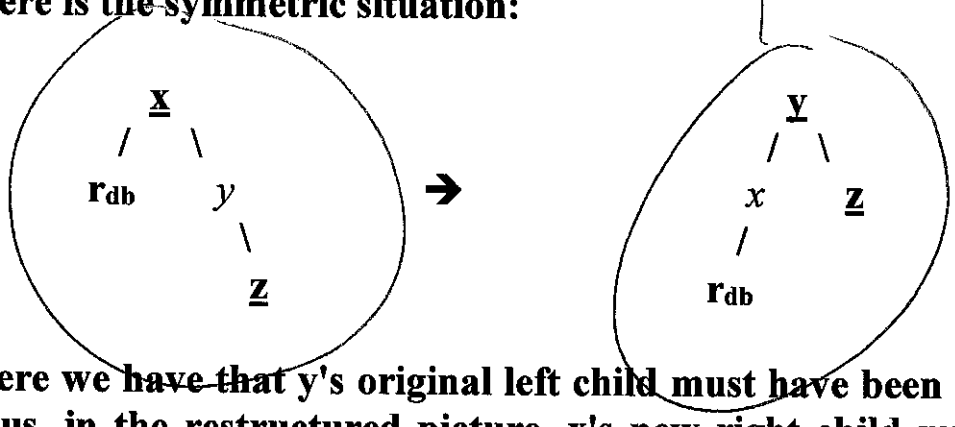
Red-Black Tree Deletion (Case 3)

This is the last case where we have to deal with a double black node r. In this situation, y, the sibling of r, is red. If y is a right child of x (where x is r's parent), let z be the right child of y. Otherwise, let z be the left child of y. (Note that both x and z must be black.) Perform a restructuring on the node z, placing y where x used to be:



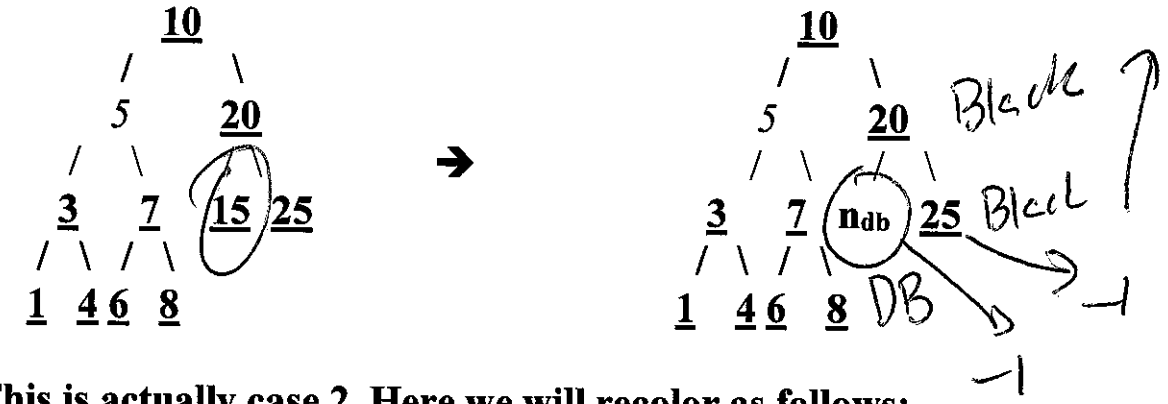
Since y's right child in the original picture MUST BE black, x's new left child MUST BE black as well. This puts us in either case 1 or case 2 to deal with the "double black" node.

Here is the symmetric situation:

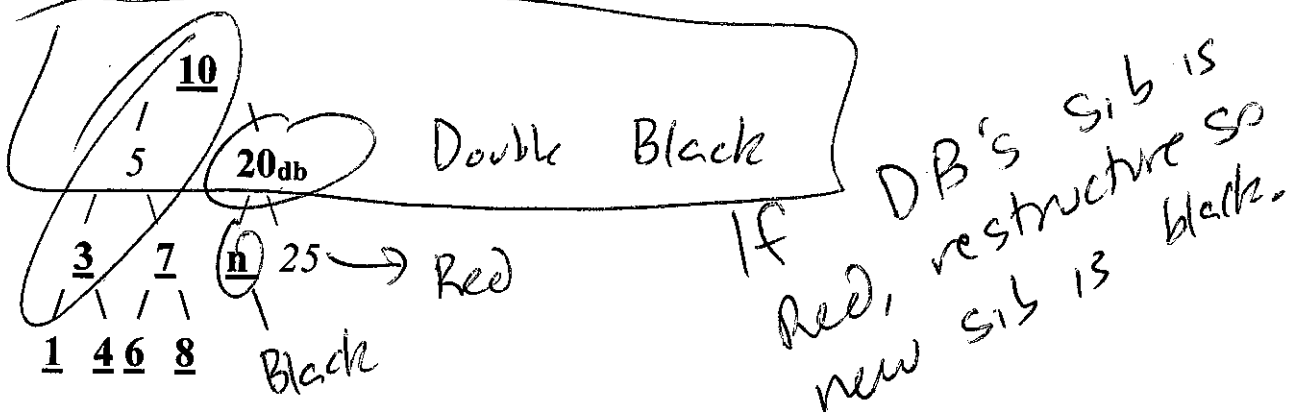


Here we have that y's original left child must have been black, thus, in the restructured picture, x's new right child must be black as well, putting us in either case 1 or case 2 as desired.

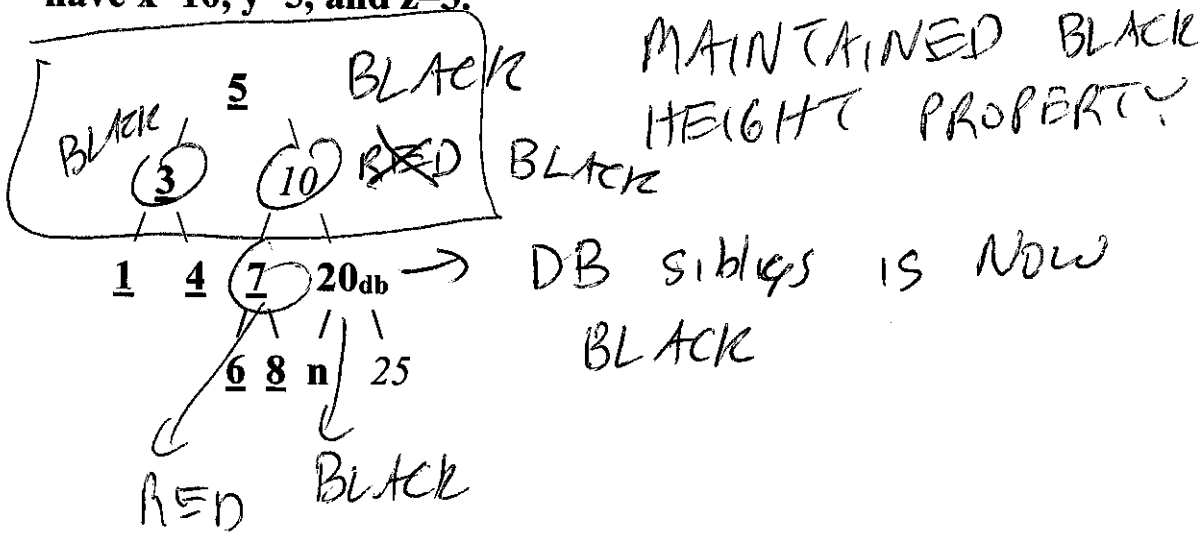
In both of these situations, since x is red, we will NOT propagate the "double black" node. Instead it will be handled in one operation of either case 1 or case 3.
Consider deleting 15 from the Red-Black Tree below:



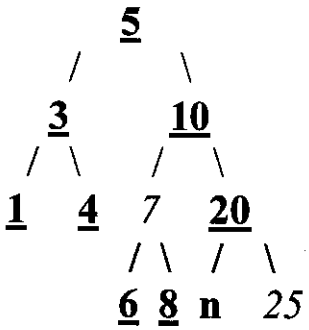
This is actually case 2. Here we will recolor as follows:



Now we have a situation where we are in case 3. Here we will have x=10, y=5, and z=3.

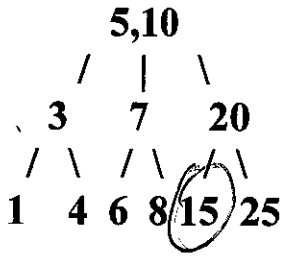


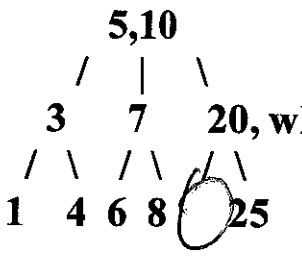
Now, we are in case 2, since both of 7's children are black. Deal with this case with a recoloring:



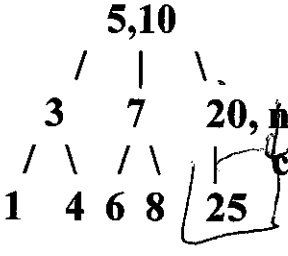
NEW PIC

Consider the corresponding delete in a 2-4 Tree:



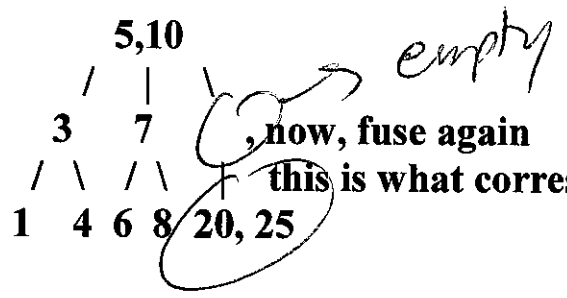


20, which causes a fusion operation

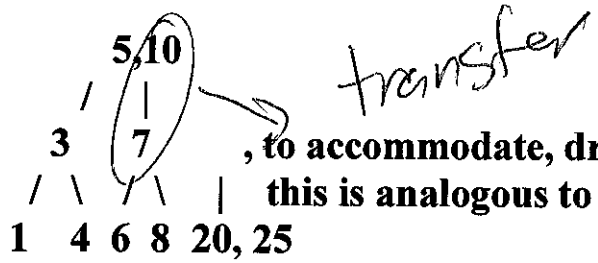


20, now, to remedy this, drop 20 into the child. corresponding to the first case 2 work.

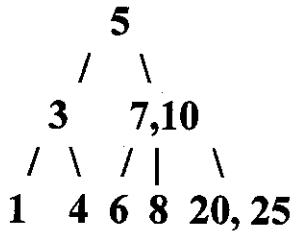
fuse



, now, fuse again
this is what corresponds to case 3.



, to accommodate, drop 10 into fused node
this is analogous to the last case 2 we see.



Notice that case three corresponds to the case when we have to drop a value from a parent node into a child after a fusion, where the parent node DOES have a value to spare. This is why after this occurs, only a single application of case 1 or case 2 is necessary. The difference in case corresponds to a difference in which value from the parent node will end up dropping into the fused node.

2/6/2018 (13)

Del red leaf node

Summarizing Red-Black Tree Delete

We have a couple simple cases to deal with, which we can do without any extra work. These correspond to removing a value from a 2-4 tree where there are other values at the node the removal is made form.

The rest of the cases result in a "double black" colored node. These cases all correspond to restructuring operations in the 2-4 Tree, such as fusions, and dropping elements into a node. The goal is to deal with the double black(DB) node to get rid of this property. Here is the outline of these cases:

Case 1: DB node has black sibling with at least one red child.

This fixes a the problem structurally. No extra work is required after this case completes. This corresponds to a transfer operation in a 2-4 Tree.

Push up black color

Case 2: DB node has black sibling with two black children.

This uses a recoloring and no structural change. It may solve the problem, but may ALSO propagate the DB node to the parent of the current DB node. This corresponds to a fusion and drop operation in a 2-4 Tree.

Propagate problem up

Case 3: DB Node has red sibling

STRUCTURAL CHANGE

A structural change here puts you in case 1 or case 2. At this point, a single application of either case is sufficient. This corresponds to a fusion where you have enough values in the parent node to drop one into the fused child.

NEW SITUATION SO DB has a Black Sibling

Avg Case Analysis QuickSort

array[low..high]

```
QS(array, low, high) {
    if (low >= high) return;
    int mid = partition(array, low, high);
    QS(array, low, mid-1);
    QS(array, mid+1, high);
}
```

high-low+1 → O(n)

Prob $\frac{1}{n}$ Left = 0, Right = n-1

Prob $\frac{1}{n}$ Left = 1, Right = (n-2)

Prob $\frac{1}{n}$ Left = 2, Right = n-3

$T(n) = T(0) + T(n-1) + O(n)$

$T(n) = T(1) + T(n-2) + O(n)$

$T(n) = T(2) + T(n-3) + O(n)$

Expectation = $\sum P_x \cdot X$ for each possible X.

If $T(n)$ represents AVG CASE run time of Quick Sort, then:

$$T(n) = \frac{1}{n}\left(T(0) + T(n-1) + O(n)\right) + \frac{1}{n}\left(T(1) + T(n-2) + O(n)\right) + \frac{1}{n}\left(T(2) + T(n-3) + O(n)\right) + \cdots + \frac{1}{n}\left(T(n-1) + T(0) + O(n)\right)$$

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + cn$$

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2 \quad \text{(mult } n\text{)}$$

$$(n+1)T(n+1) = 2\sum_{i=0}^{n} T(i) + c(n+1)^2 \quad \text{(new eq plug in } n+1\text{)}$$

$$(n+1)T(n+1) - nT(n) = 2T(n) + c(2n+1)$$

$$\frac{(n+1)T(n+1)}{(n+1)(n+2)} = \frac{(n+2)T(n)}{(n+1)(n+2)} + \frac{c(2n+1)}{(n+1)(n+2)} \quad \text{(trick: divide by both products)}$$

$$\frac{T(n+1)}{n+2} = \frac{T(n)}{n+1} + \frac{c(2n+1)}{(n+1)(n+2)}$$

Let $S(n) = \frac{T(n)}{n+1}$

$$S(n+1) = S(n) + \frac{C(2n+1)}{(n+1)(n+2)}$$

$2n+1$ is "close to" $2\times(n+1)$

replace $2n+1$ with $2n+2$ and get a slightly higher answer.

2 $$S(n+1) = S(n) + \frac{2C}{n+2}$$