Week 9: Balanced Trees

1

In the worst case, all BST operations runs in O(N) time. This case happens when the tree has a linear structure. We want to maintain additional properties on a BST so that it is balanced. Perfect balance is hard to achieve, so we usually relax the balance properties. As long as we can make sure that the height of the BST is always O(log N), we are happy!

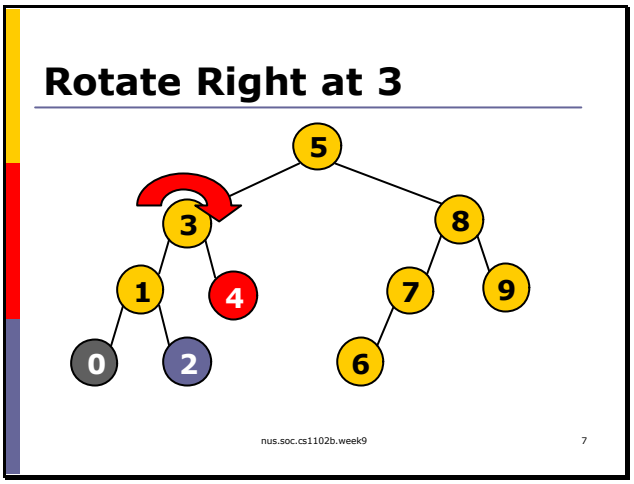
Readings

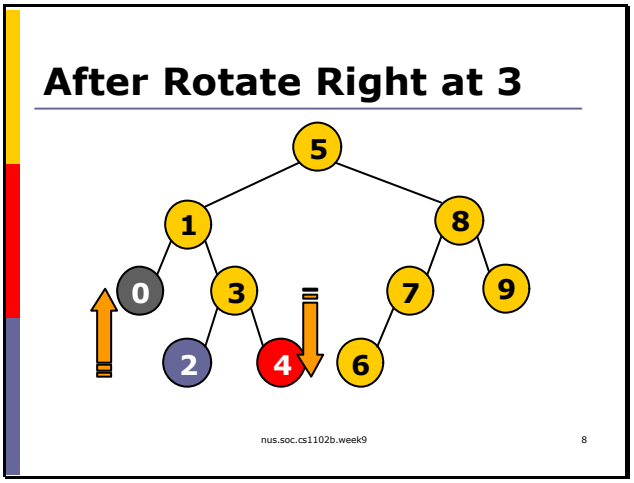
- Required
  - [Weiss] ch19.4 and ch19.5 – 19.5.1
  - http://www.ddj.com/print/documentID=14585
- Fun
  - http://www.seanet.com/users/arsen/avltree.html
  - http://www.cs.washington.edu/homes/sds/rb.html
    (don't take this too seriously)
- Exercise
  - [Weiss] 19.5

Rotate Operation

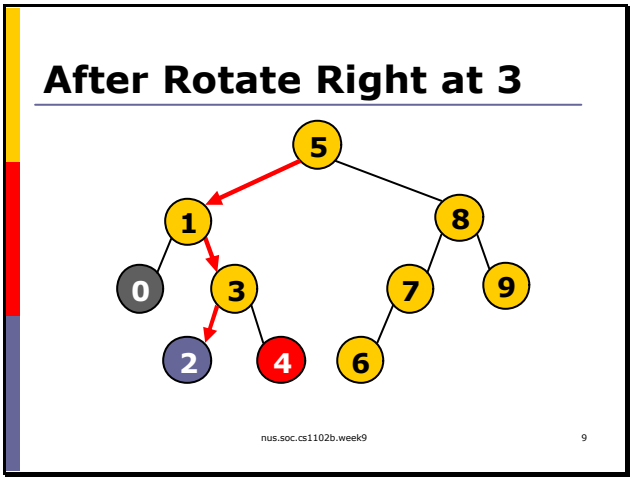
6

The rotate operation is an important operation for maintaining the balance of a BST.



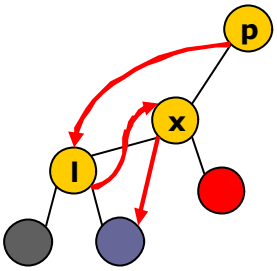


Rotation changes the heights of nodes. In this example, the depth of node 4 increases by 1. The depth of node 0 decreases by 1. The depth of node 2 remains unchanged.



Right rotation modifies the following pointers.

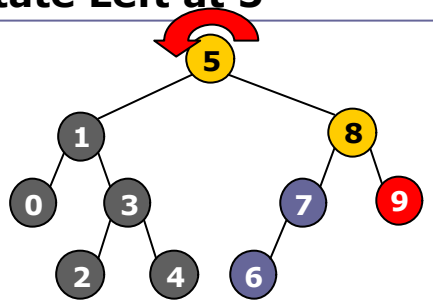
Rotate Right at x



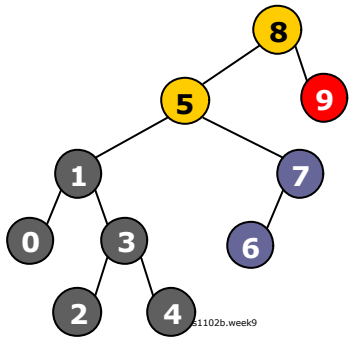

```
rotateRight(x)
  l = x.left
  if l is empty
      return
  x.left  = l.right
  l.right = x
  p = x.parent
  if x is a left child
      p.left = l
  else
      p.right  = l
```


The pseudocode on the right shows how we rotate right at x. The red arrows are the pointers after modification.

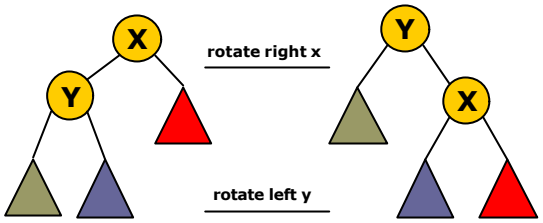
Rotate Left at 5



After Rotate Left at 5



Rotation Summary



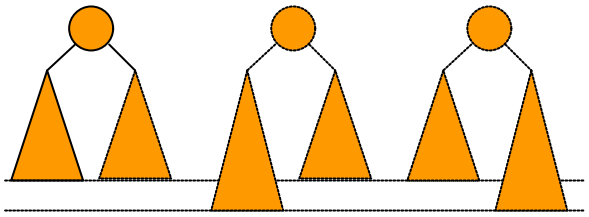
AVL Tree

AVL Tree Properties

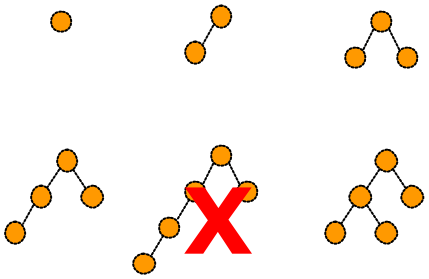
- Difference in height between left and right subtree is at most one.

$$|H_l - H_r| \leq 1$$

AVL Tree Properties



AVL Tree Example



Height of an AVL Tree

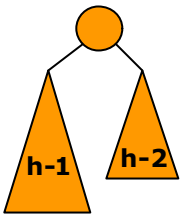
- Minimal AVL trees: AVL Tree with fewest possible number of nodes
- Minimal AVL trees with height 0

- Minimal AVL trees with height 1

Height of an AVL Tree

- n(h): number of nodes in a minimal AVL tree with height h.

- n(0) = 1
- n(1) = 2
- n(h) = 1 + n(h-1) + n(h-2)
  > 2n(h-2)



The last line comes from the fact that n(h-1) > n(h-2).

Height of an AVL Tree

n(h) > 2n(h-2)
> 2 * 2n(h-4)
> 4n(h-4)
> 4 * 2n(h-6)
> 8n(h-6)
:
> 2^i n(h-2i)

Height of an AVL Tree

n(h) > 2^i n(h-2i)

when h – 2i = 0, i = h/2

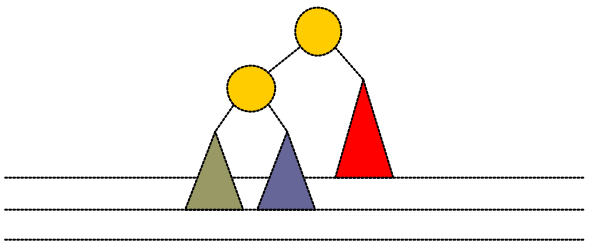
n(h) > $2^{h/2}$
h < 2 log n(h)
h < 2 log N
h = O(log N)

Verify that when h is odd, the height is also O(log N)

AVL Tree Insertion

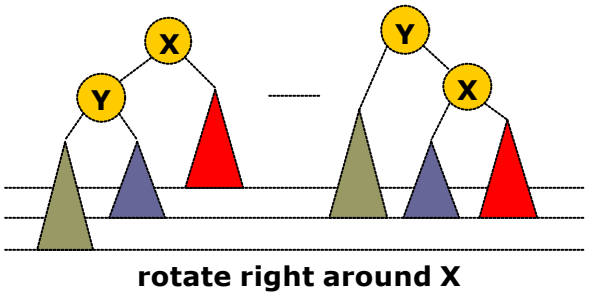
23

Idea



nus.soc.cs1102b.week9 24

Case 1: Outside

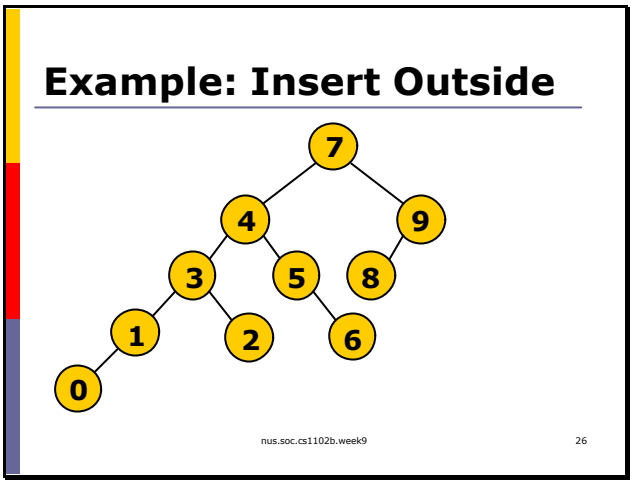


rotate right around X

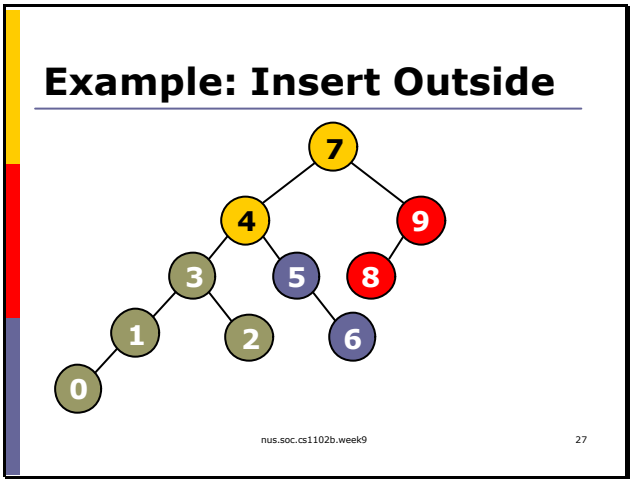
nus.soc.cs1102b.week9 25

We can insert into the red subtree, blue subtree, or gray subtree. Insertion in red subtree never violates the AVL tree property. But insertion into blue and gray subtree _may_ cause a violation.

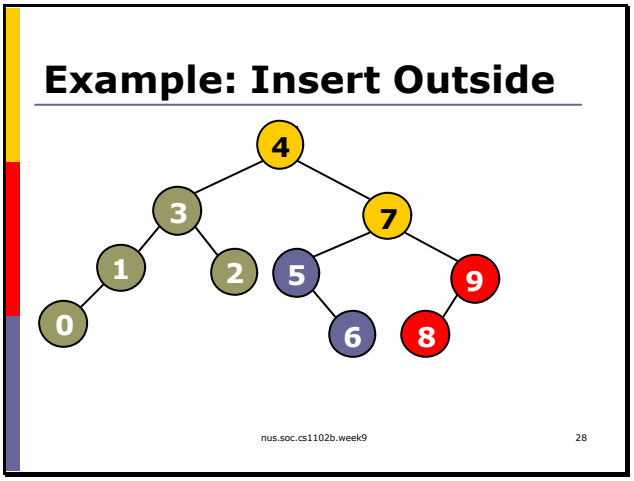
To insert into an AVL tree, we insert the node as usual. After insertion, travel from new node back to the root. At each node, checks if $| H_l - H_r | = 1$. If violation occurs, rotate the tree based on the following cases.



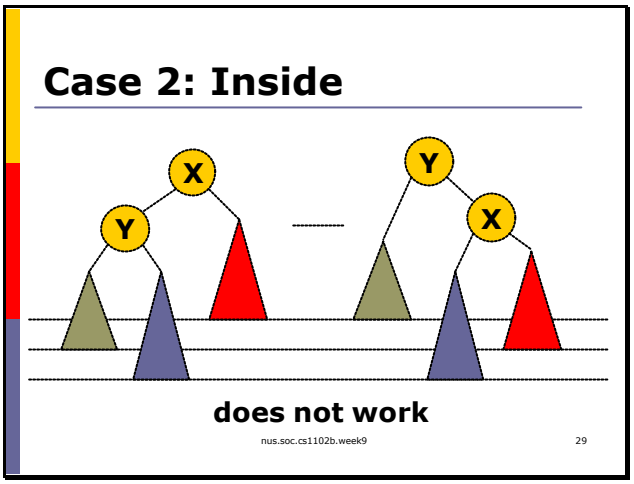
0 is the new inserted node. In the first pass, we move down the tree just like insertion into a BST.



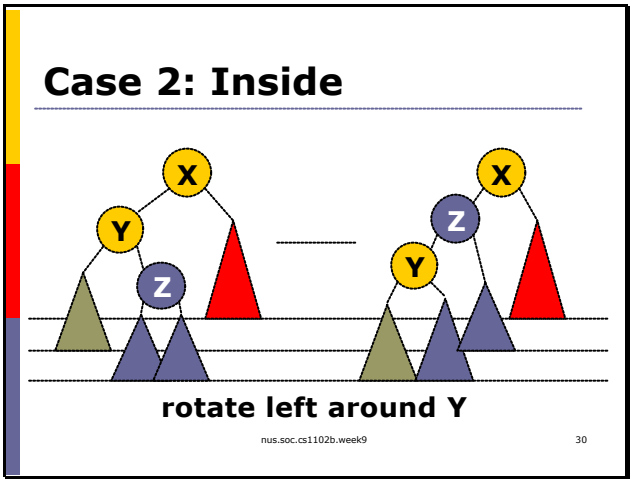
Then, we climb our way back up. On our way towards the root, we check if the current subtree violates the AVL Tree properties. In this example, the AVL Tree property is violated at the root 7.



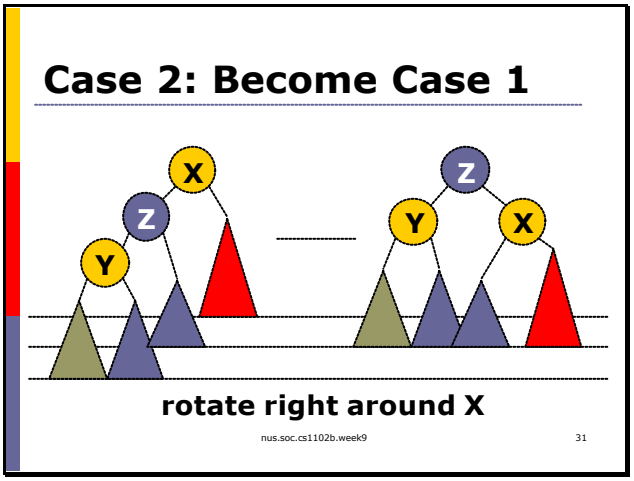
We therefore perform a single right rotation at 7 to fix the tree.



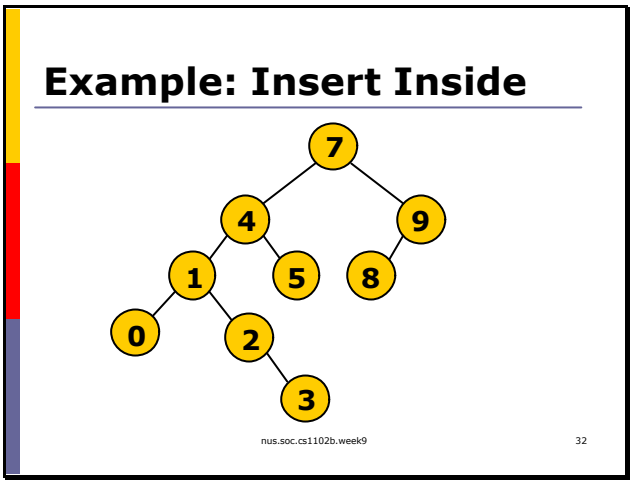
Single rotation does not work if the new node that causes violation belongs in the blue sub-tree. We need double rotations.



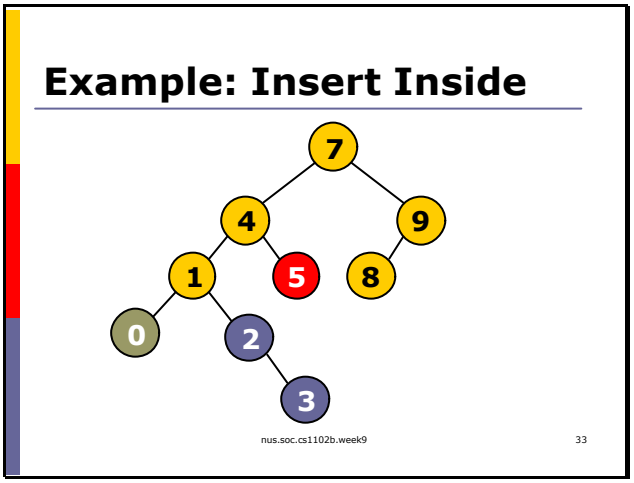
We first rotate left around Y. We have reduced our problem to the previous one: Case 1.



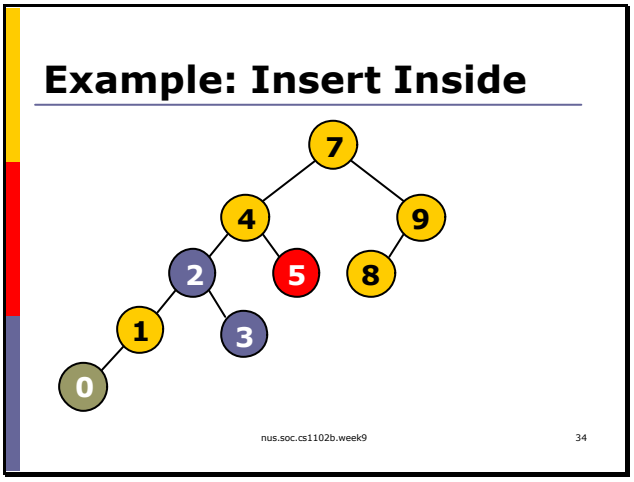
Now, we do a rotation around X to fix the problem.



In this example, we insert 3.



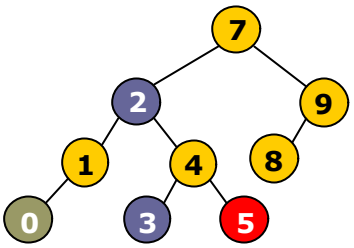
The AVL Tree property is violated at 4. We do a double rotation.



First, we rotate left at 1. We get the tree shown in the left.

Then, we rotate right around 4.

Example: Insert Inside

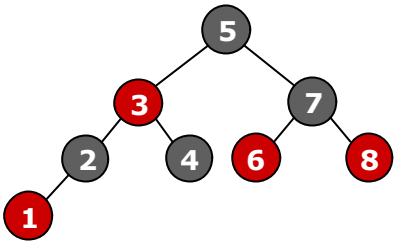


Summary

- Insert outside: Single Rotation
- Insert inside: Double Rotation
- Two passes needed: first pass down to insert, second pass up to fix.

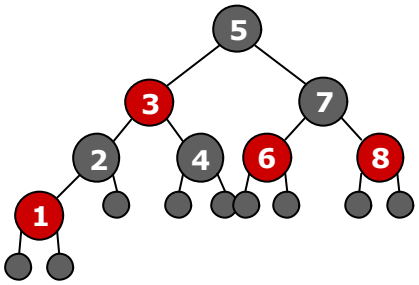
Red-Black Tree

A Red-Black Tree



A Red-Black Trees has 4 properties.
(i) All nodes must be colored red or black.
(ii) The root must be black
(iii) If a parent is red, its children must be black
(iv) All paths from the root to the null pointer must contain the same number of black nodes.

Dummy Leaves



We add dummy black leaves to simplify implementation of red-black trees.

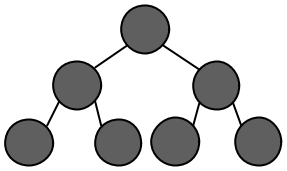
Properties of RB Tree

- B: Number of black nodes along a path
- N: Number of nodes

$$N >= 2^B - 1$$
$$B <= \log_2(N+1)$$

To get the first inequality, we consider the smallest possible red-black tree with B black nodes along the path from root to null pointer. Such a red-black tree must contain only black nodes, and is a complete binary tree.

Complete Binary Tree



$= 2^B-1$

Recall how we compute the number of nodes in a complete binary tree (See Week 8: Trees). The number of nodes is 2^B-1. Note that we do not consider the dummy leaves here.

Height of an RB Tree

- h: Height of an RB Tree

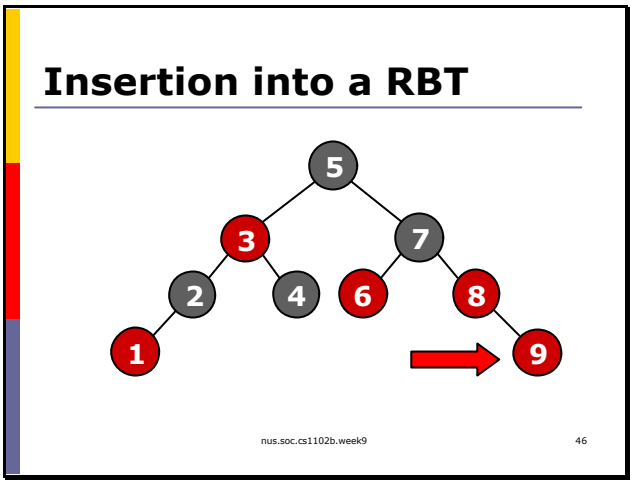
$h <= 2B$

$h <= 2 \log_2 (N+1)$

$h = O(\log N)$

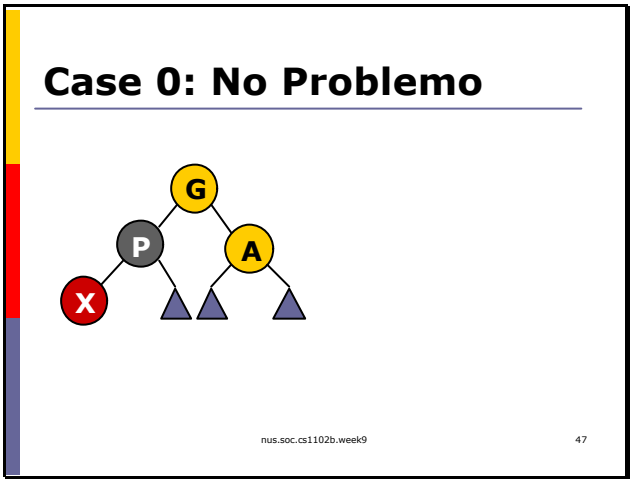
For a general red-black tree, since we cannot have two consecutive red nodes, the maximum height must be 2B. From $B = \log (N+1)$, we get $h = 2\log (N+1)$.

Red-Black Tree Insertion

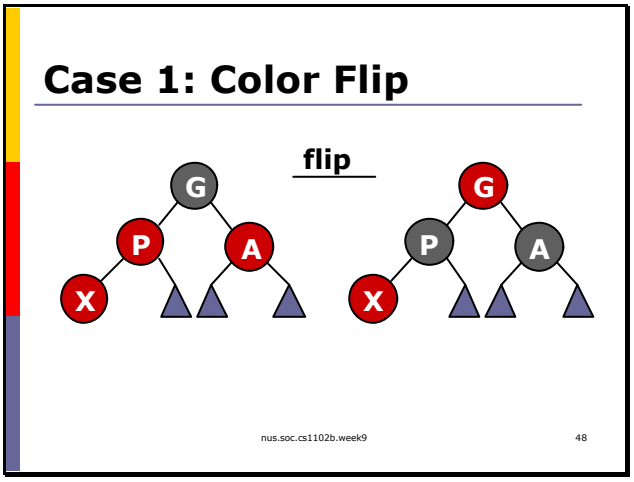


We always color the new node red, because if the new node is black, we violate rule number 4.

We insert as usual, then travel up the tree, trying to fix the tree if violation of red-black tree rules occurs.

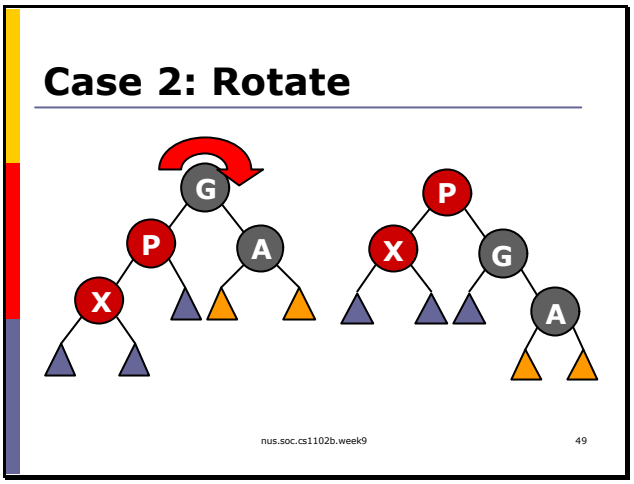


If the parent of X is black, we are done. Note that in this case, we do not care if X is left or right child of its parent. Nor do we care about the colors of the grandparent or uncle.

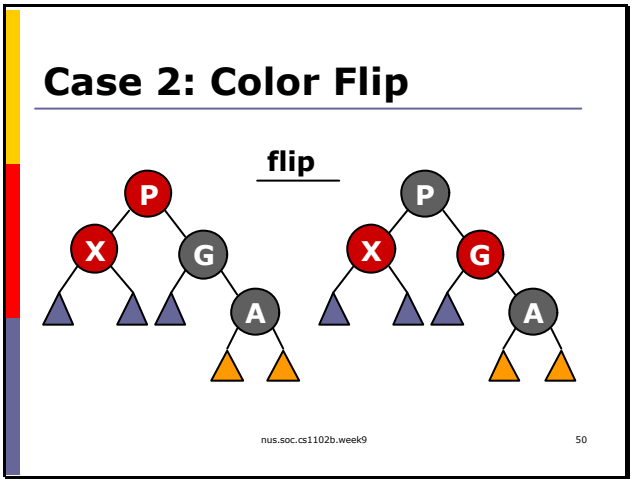


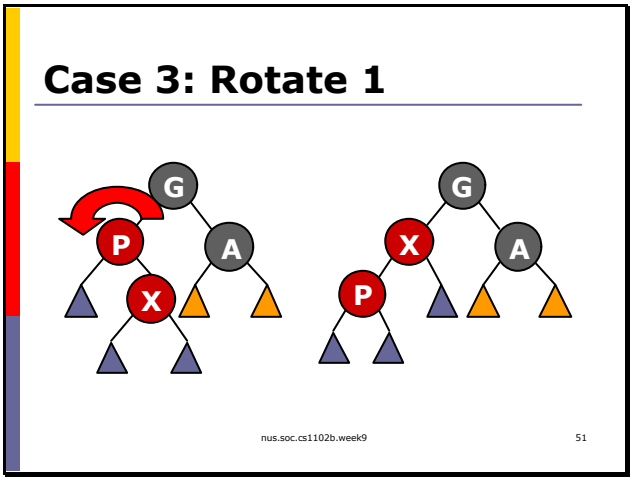
If the parent is red, we look at the color of its uncle. If the uncle is red, we just do a color flip. Note that in this case, we do not care if X is left or right child of its parent either. If the parent of G is black, then we are done. Otherwise, parent of G is also red, and we have violated rule 3. We must continue to fix the tree.

Note that even though I draw X as a leaf, this case may happen in general, when X is an internal node.



If X's uncle is black, and X is a left child, we do a single right rotation around its grandparent, and then do a color flip. After this, node that the parent is black, so we do not need to fix the tree any further (unlike case 1).

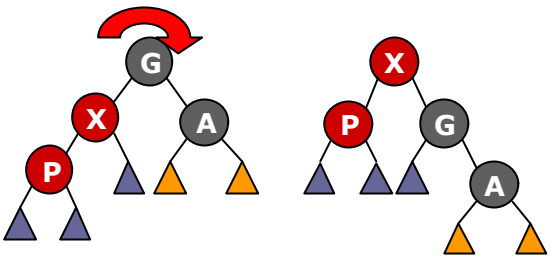




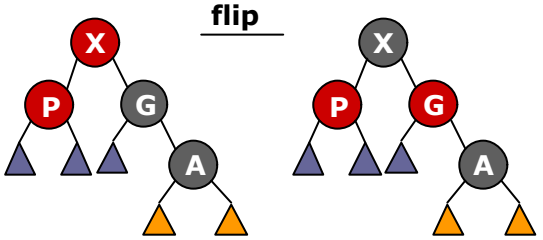
If X's uncle is black, and X is a right child, we need two rotations and a color flip.
Just like case 2, after two rotations and one color flip, we do not need to fix the tree any further.

Note that after the first rotation, we have reduce the problem to the previous one (case 2).

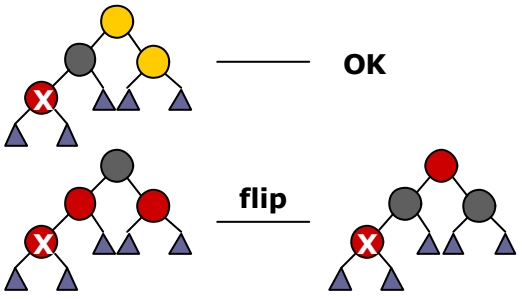
Case 3: Rotate 2



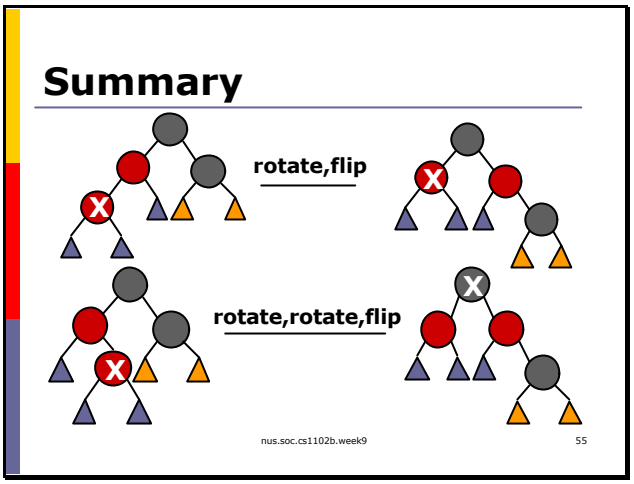
Case 3: Color Flip

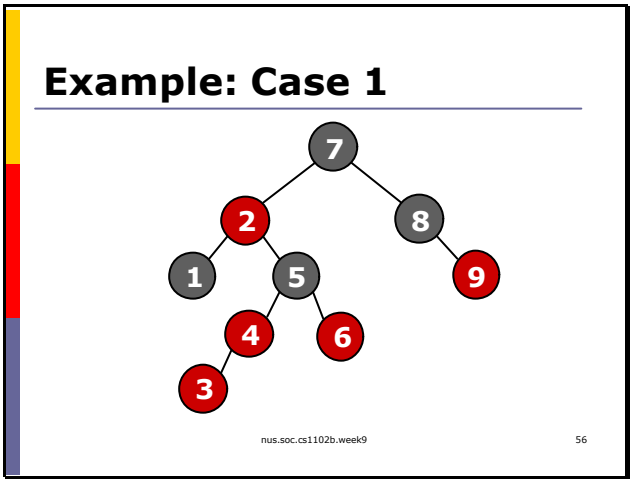


Summary

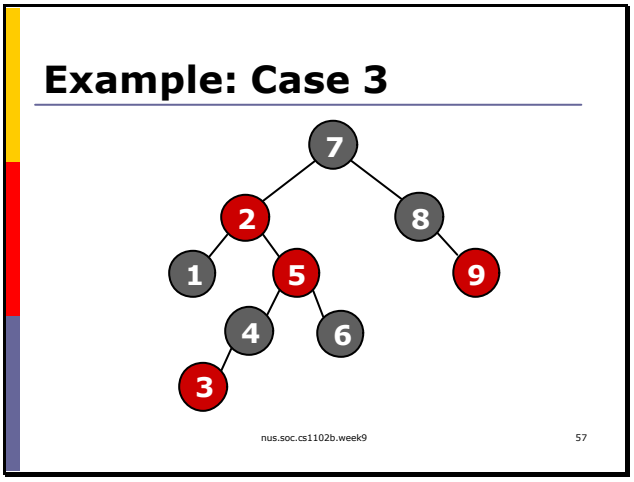


In summary, we insert, mess up the tree, and then use a bunch of rotations and color flips to correct the tree.





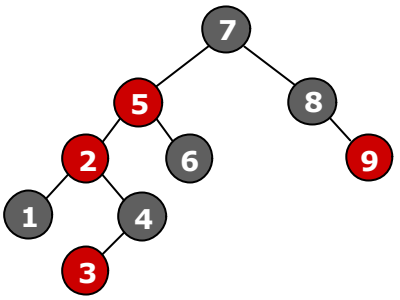
In this example, we insert 3. We color 3 as red.



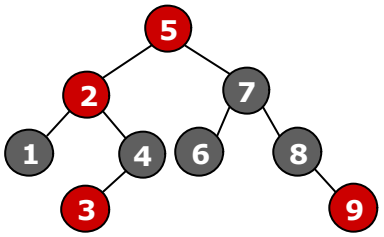
We flip the color. But we still have two consecutive red nodes in 2 and 5.

We move up the tree to fix it again. Note the position of 2,5,7 and 8. This is case 3. Do a double rotation and then a color flip.

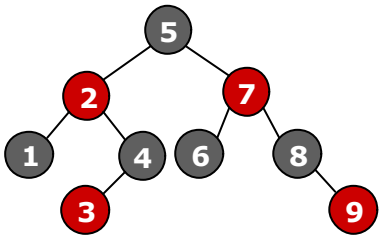
Example: Rotate



Example: Rotate



Example: Flip

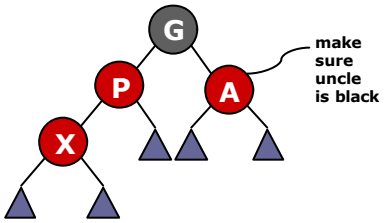


Top-Down Insertion

62

The rest of these materials are not covered in the lecture, and are optional.

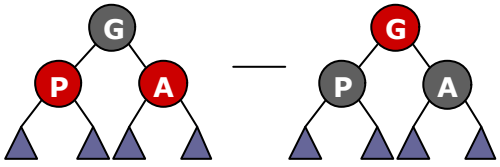
Idea



nus.soc.cs1102b.week9 63

Using the insertion routine described previously, we need two passes to insert into a red-black tree: first pass down to insert, second pass up to rotate. But we only need the second pass because in case 1, we change the color of the grandparent to red. If we make sure that case 1 never happens (i.e., the uncle is always black), then we do not need to insert in two passes.

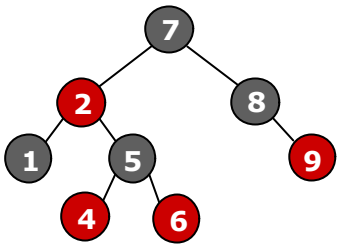
Idea



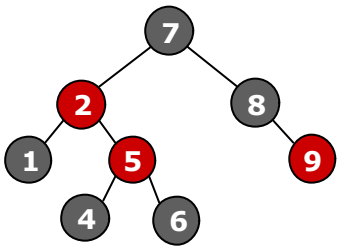
nus.soc.cs1102b.week9 64

During the first pass of insert, when we traverse down the tree, everytime there is a potential that parent and uncle are both red, (that is, we see a black node with two red children), we do a color flip. Now, the grandparent is red, and this may violate rule 3 if the great-grandparent is red as well. But we can be sure that granduncle is black (WHY?)

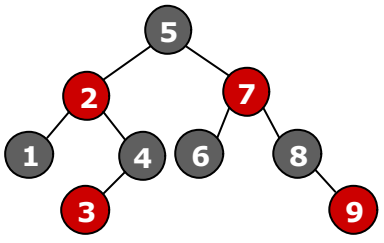
Insert 3



Insert 3



Insert 3



Top-Down Deletion

68

delete key

```
X = root
while X is not empty
  if X.data == key
      delete X
  else if X.data < key
      X = X.left
  else
      X = X.right
```

nus.soc.cs1102b.week9 69

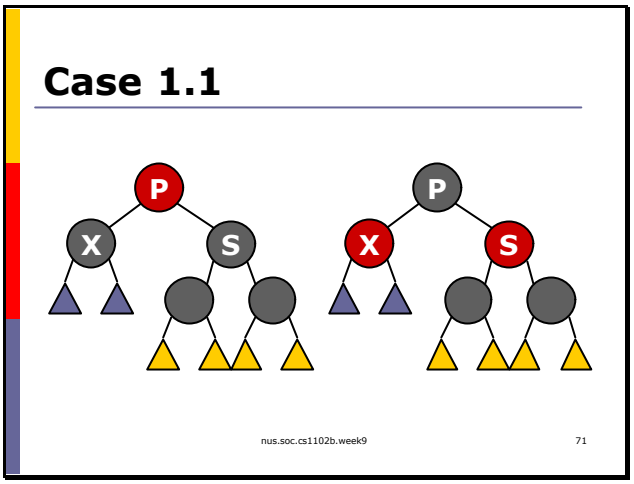
Idea: Make X Red

```
X = root
while X is not empty
  if X.data == key
      delete X
  else if X.data < key
      X = X.left
  else
      X = X.right
```

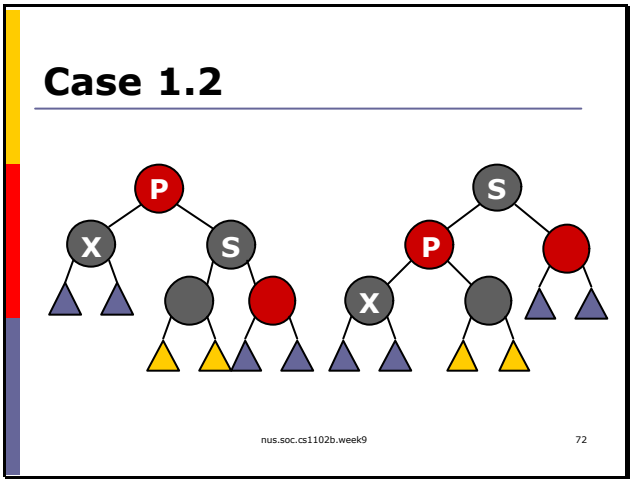
make X red
fix the tree

nus.soc.cs1102b.week9 70

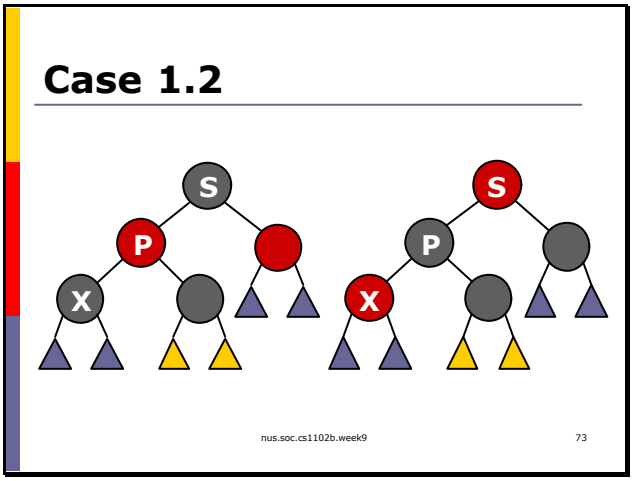
Just like top-down insertion, we can do top-down deletion. The idea here is that we want to make sure the node to be deleted is red. (WHY?). As we traverse down the tree to find the node to be deleted, we color the current node as red. This of course will mess up the tree. Therefore, we have to fix it.

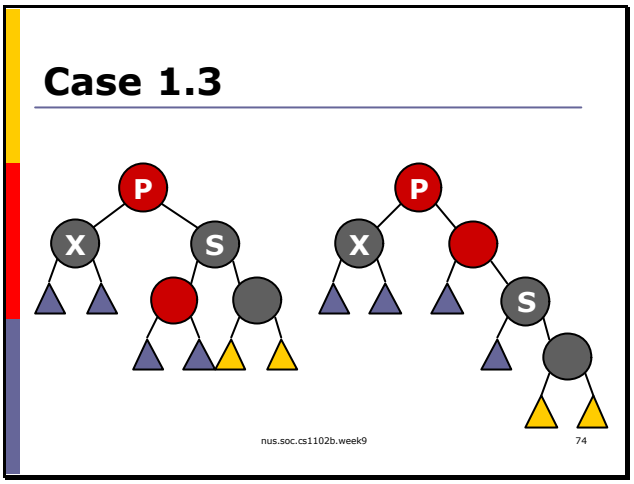


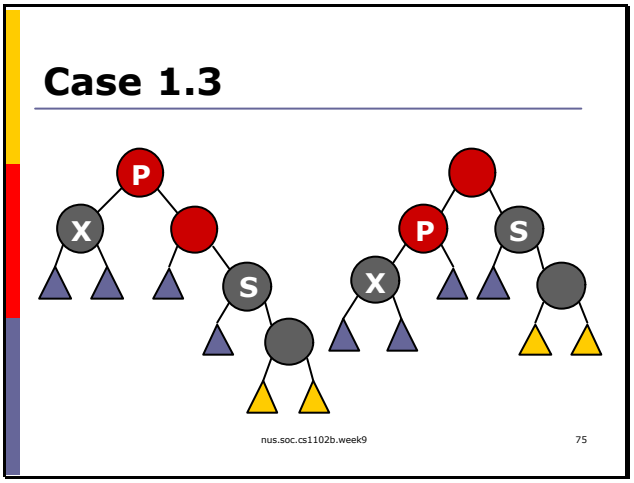
If both children of X are black, and both children of X's sibling are black as well, we just do a color flip to color X red and keep the RB-Tree properties.

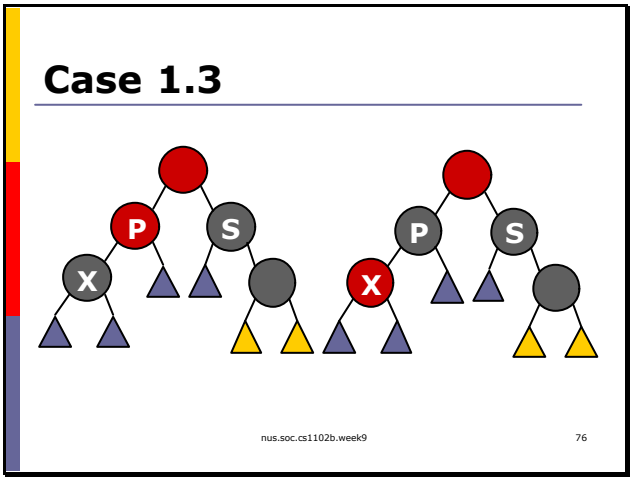


If both children of X are black, and X's right niece (children of X's sibling) is red, we need to perform a single rotation, follow by a color flip to color X as red and maintain the RB Tree properties.

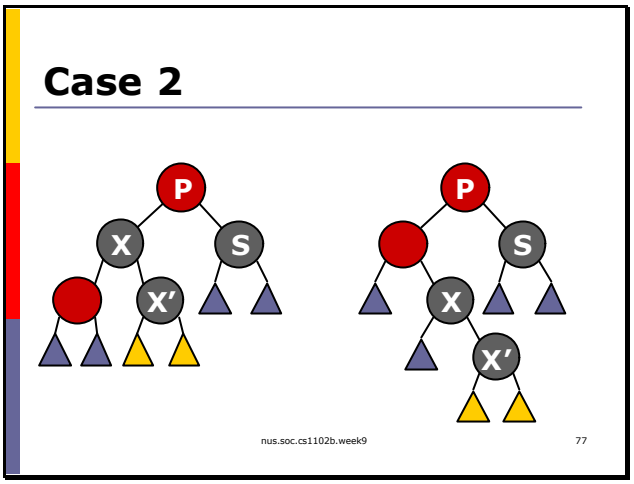


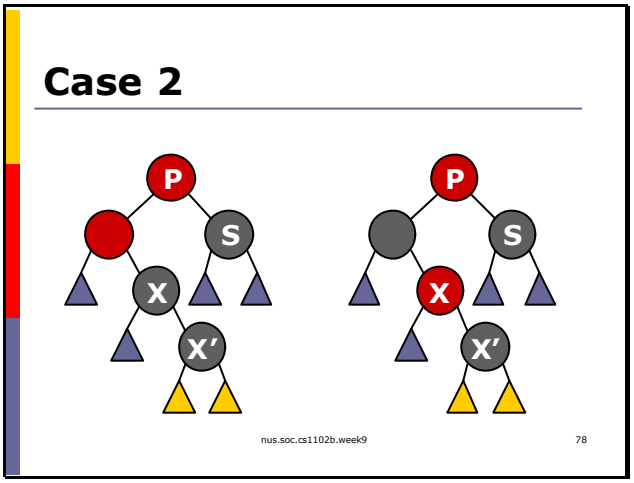






If both children of X are black, and X's left niece (children of X's sibling) is red, we need to perform a double rotation, follow by a color flip to color X as red and maintain the RB Tree properties.





Now consider the case if X has a red child. We do nothing, and hope that the next step of traversal brings up to the red node. If so, we are OK. Otherwise, we have step onto a black node X'. We do a rotation and a color flip. But X' is still black. Now we check which case (1.1,1.2,1.3 or 2) applies again to make X' red.