19.5 – Red-Black Trees

1. Red-black tree demo – http://gauss.ececs.uc.edu/RedBlack/redblack.html

2. In an AVL tree, we recurse down the tree to find the insertion (removal) location, then work back up the tree to update the heights and rebalance the tree. In a *red-black tree*, a single pass down the tree will take care of business. An iterative solution is used.

3. A *red-black tree* is a binary search tree with the following properties:

    1. Every node is either red or black
    2. Root node is black
    3. The children of any red node are black
    4. Every path from a node to a null link must contain the same number of black nodes

4. Implications of definition

    a. Height of any red-black tree is at most $2\log(n+1)$
    b. Cannot have two successive red nodes

5. Naturally, insert and remove change the tree and can change and possibly destroy the coloring property. Next, we consider some algorithms that prevent this

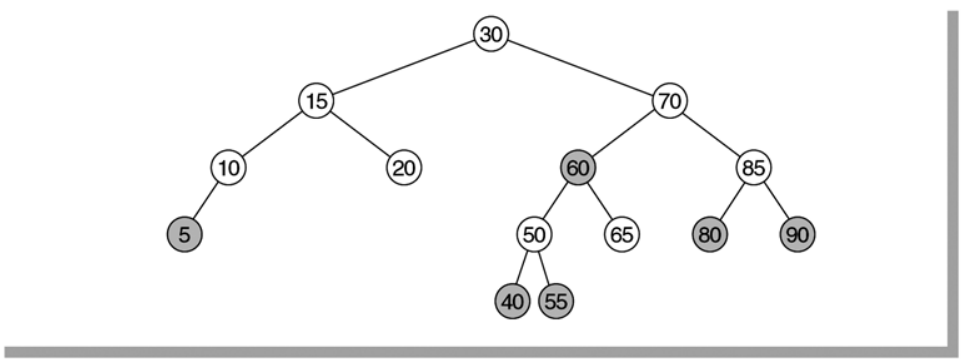


figure 19.34

A red–black tree: The insertion sequence is 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, and 55 (shaded nodes are red).

19.5.1 – Bottom-up insertion

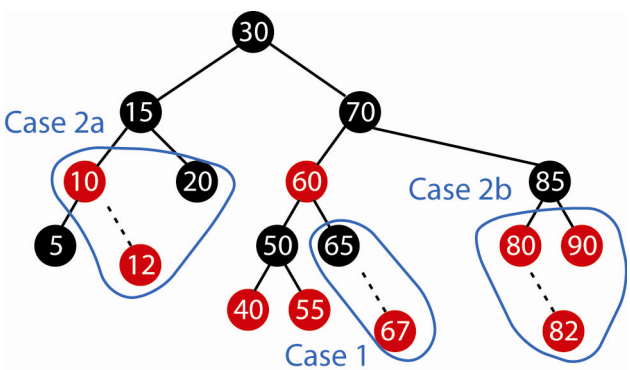
1. The AVL insertion algorithm inserts a new item as a leaf. If we color this node black, we violate rule 4 because we would be adding an additional black node along the path. Thus, new items must be colored red.

2. When we color a new node red, there are two possibilities:

   Case 1: Parent is black. In this case, all 4 rules are satisfied and the insertion is complete.

   Case 2: Parent is red. In this case, rule 3 is violated. To fix this, we will do a rotation (either single or double) and a color change.

   Case 2a: Parent's sibling is black (assume a null link is defined to be black).
   Case 2b: Parent's sibling is red.



3. Consider resolving Case 2a. Let X be the node that is inserted. There are four sub-sub-cases:

   <u>X is an outside grandchild:</u>
   i. X is a Left Child and Parent is a Left Child (AVL Case 1)
   ii. X is a Right Child and Parent is a Right Child (AVL Case 4)

   <u>X is an inside grandchild:</u>
   iii. X is a Right Child and Parent is a Left Child (AVL Case 2)
   iv. X is a Left Child and Parent is a Right Child (AVL Case 3)

   a. Case 2a i – In the figure below, we think more generally. We say that X is the node where we have a violation, which may result from an insert or a remove. What can we say about the Grandparent of X? It must be black, otherwise, we would have had an additional violation. A single rotation and color swap will resolve the situation as shown below:

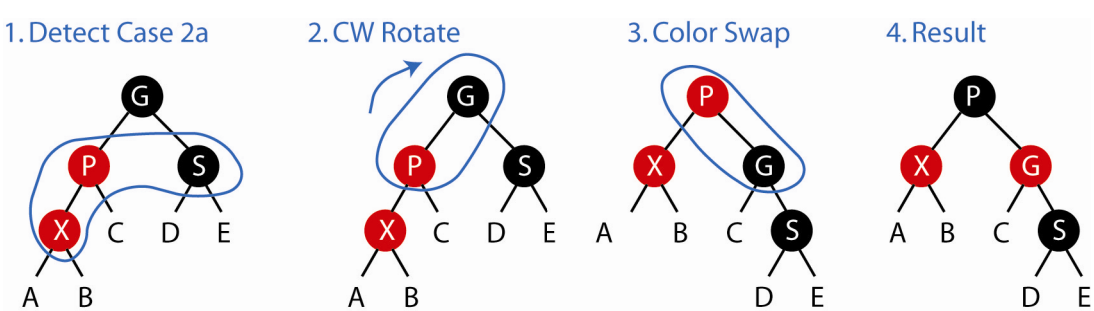
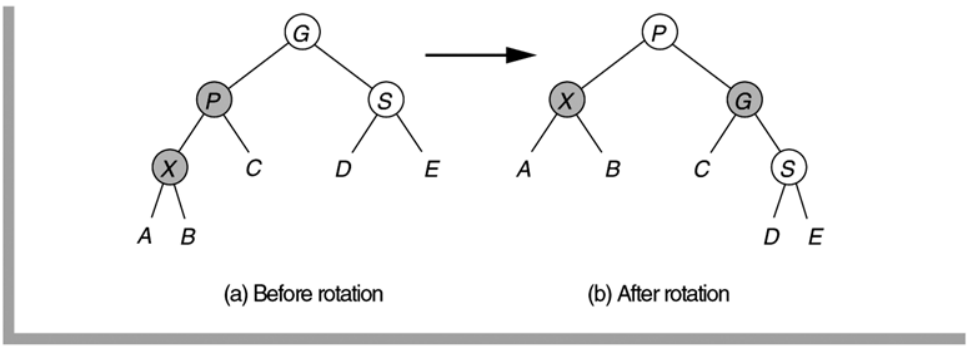


figure 19.35

If S is black, a single rotation between parent and grandparent, with appropriate color changes, restores property 3 if X is an outside grandchild.



b. Case 2a i – Algorithm

If we define:

```
colorSwap( node1, node 2) = swap the colors of nodes 1 and 2
```

Then, we can see that this case resolves to (AVL Case 1):

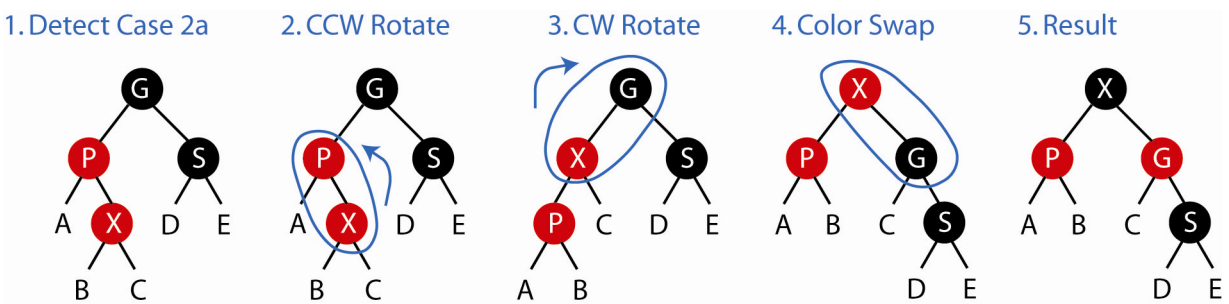
```
rotate( parent, grandparent, CW )
colorSwap( parent, grandparent )
```

c. Case 2a ii – Algorithm

And, the reflexive case (AVL Case 4) is:

```
rotate( parent, grandparent, CCW )
colorSwap( parent, grandparent )
```

d. Case 2a iii – The case where X is an inside grandchild is resolved with a double rotation and a color swap as shown below:



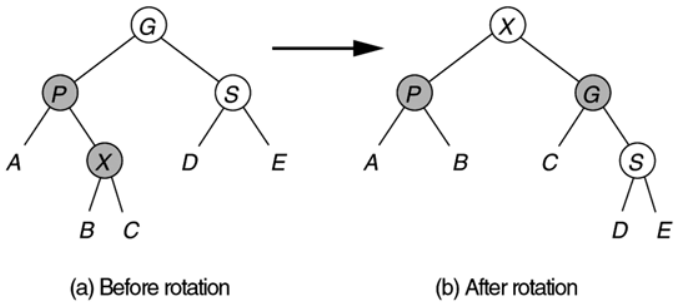


figure 19.36

If *S* is black, a double rotation involving *X*, the parent, and the grandparent, with appropriate color changes, restores property 3 if *X* is an inside grandchild.

e. Case 2a iii - Algorithm

We can see that this case resolves to (AVL Case 2):

```
rotate( node, parent, CCW )
rotate( node, grandparent, CW )
colorSwap( node, grandparent )
```

f. Case 2a iv - Algorithm

And the reflexive case (AVL Case 3) is:

```
rotate( node, parent, CW )
rotate( parent, grandparent, CCW )
colorSwap( parent, grandparent )
```

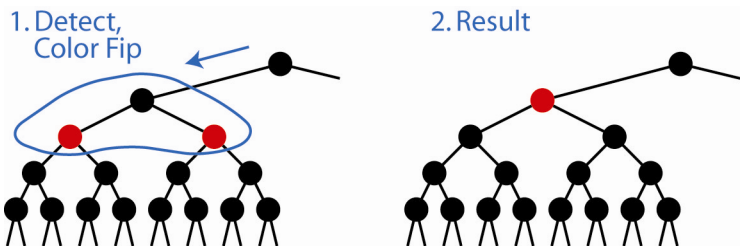
4. Case2b – (Red parent and red sibling) Rotation scheme can work sometimes, sometimes requires an additional color swap. Sometimes, though it doesn't work, and you have to keep percolating up the tree resolving problems. Thus, we have lost the top-down pass to resolve everything. However, the techniques (rotations) described above will be useful. Thus, a bit of a dead end. But, don't worry, we'll use these ideas presented so far in a solution!

19.5.2 – Top-down red-black trees

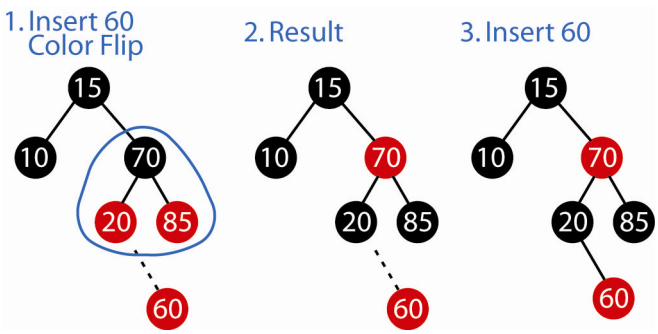
1. Come up with an approach that guarantees that when we arrive at a leaf and insert a node, S (the sibling of parent) is not red. Thus we eliminate Case 2b and can deal with Cases 1 and 2a as above.

2. The idea is that on the way down, whenever we see a (black) node X with two red children, we swap colors. X becomes red and the children become black. Thus, we will introduce the algorithm:

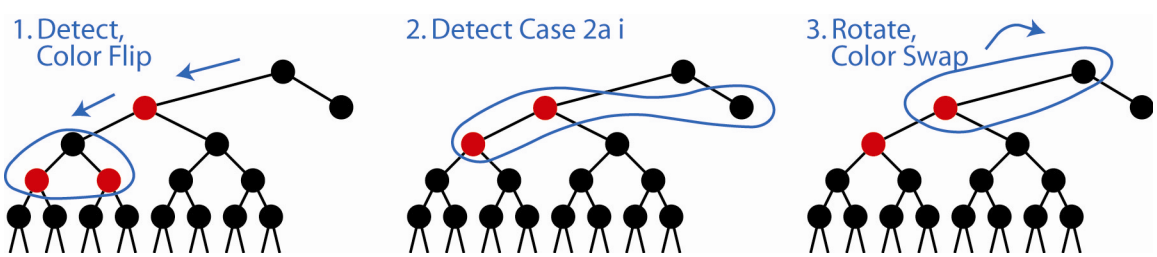
```
colorFlip( node, children ) – Make node red and children black
```

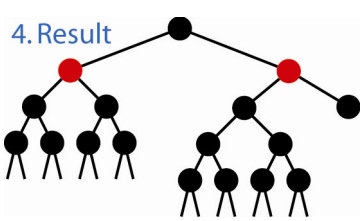


3. Example:

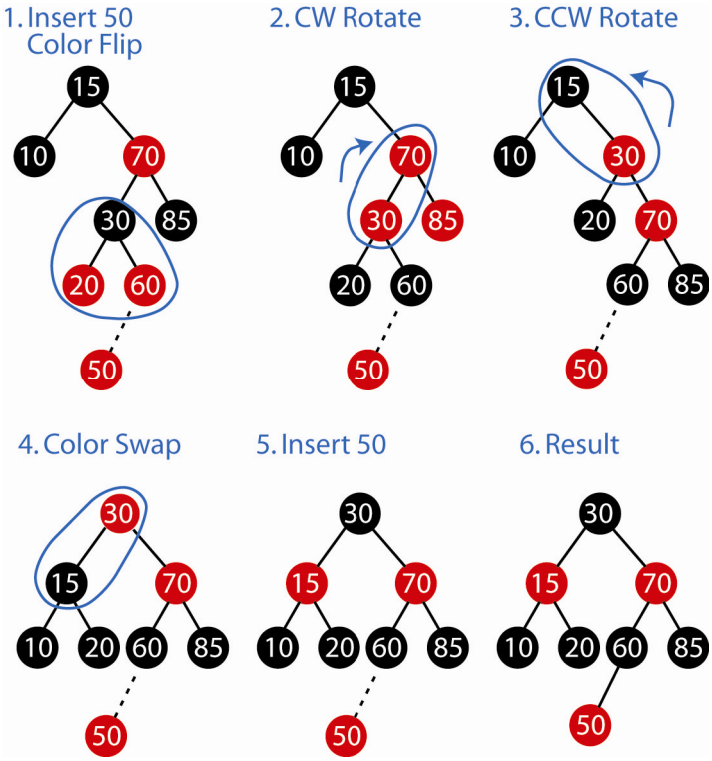


4. Note that a colorFlip may cause a Red-Red violation between the node, X and its parent. Thus, after a colorFlip, we must check the parent to see if it is red. If it is, then we just follow the solution for case 2a in section 19.5.1 (above). Note that indeed we do have Case 2a (a black sibling), because if the sibling were red, it would have been color flipped previously.

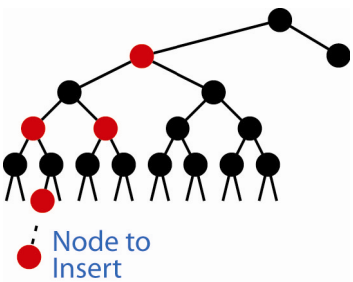




5. Example:



6. Once all the colorFlips are done, as we progress down the tree, we finally insert the new node. Let's call that node, *node*. If node's parent is red, then we again come back to the solution for case 2a from section 19.5.1.



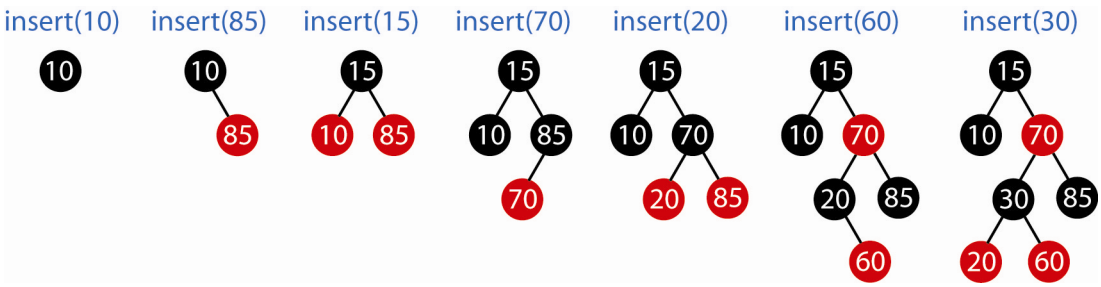
7. Consider this rotate algorithm to help us consolidate the insert algorithm. Note that it doesn't actually rotate unless there are 2 reds in a row.

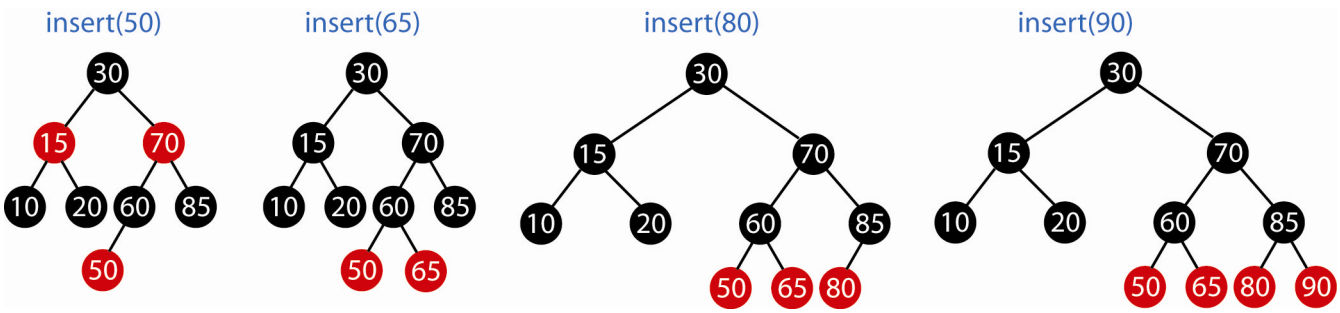
```
rotate( node, parent, grandparent )

   if node's parent is red
      if node is left child and parent is left child // case 1
         rotate( parent, grandparent, CW )
         colorSwap( parent, grandparent )
      else if node is right child and parent is left child // case 2
         rotate( node, parent, CCW )
         rotate( node, grandparent, CW )
         colorSwap( node, grandparent )
      else if node is left child and parent is right child // case 3
         rotate( node, parent, CW )
         rotate( node, grandparent, CCW )
         colorSwap( node, grandparent )
      else if node is right child and parent is right child // case 4
         rotate( parent, grandparent, CCW )
         colorSwap( parent, grandparent )
   if node is the root and red
      change node's color to black
```

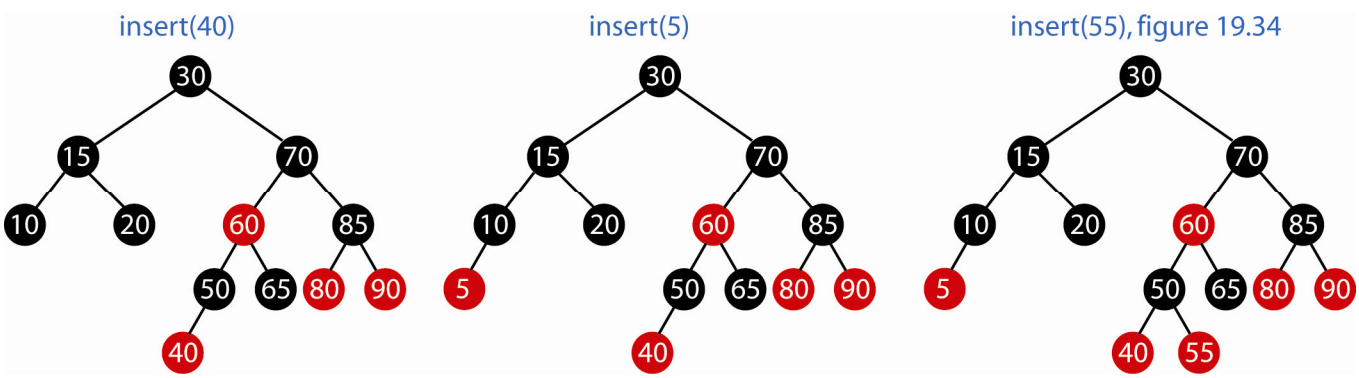
8. Thus, an algorithm for insert for a red-black tree:

```
insert( key ) : newNode
{
   return insert( root, key )
}

insert( node, key ) : newNode
{
   while node is not null
      if node is black and both children are red
         colorFlip( node, children )
         rotate( node, parent, grandparent )

      if key is less than node
         if node has left child
            Advance to left child: node = node.left
         else
            X = new Node(key)
            Connect node to new node: node.left = X
            Advance to new node: node = X
            rotate( node, parent, grandparent )
            return X

      if key is greater than node
         if node has right child
            Advance to right child: node = node.right
         else
            X = new Node(key)
            Connect node to new node: node.right = X
            Advance to new node: node = X
            rotate( node, parent, grandparent )
            return X
```

9. Example: Build a red-black tree by inserting these nodes in the following order: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55. The results of each step are shown below. The final tree is figure 19.34 from the text.







Homework 19.11

1. Build a Red-Black tree by inserting nodes in this order: 1,2,3,4,5,6,7,8,9

2. Show the tree that results from inserting 16 into the Red-Black tree shown below.