CMSC 341
Lecture 15

Bottom Up Implementation

Perform insertion; new node is red

Check for balance violations

- Are all nodes red or black?
- Is every NULL pointer black?
- If a node is red, are both of its children black?
- Does every path from a node to a descendent leaf contain the same number of black nodes?

If necessary, make adjustments starting at altered node

Bottom Up Insertion (cont)

Insert node; X is pointer to it

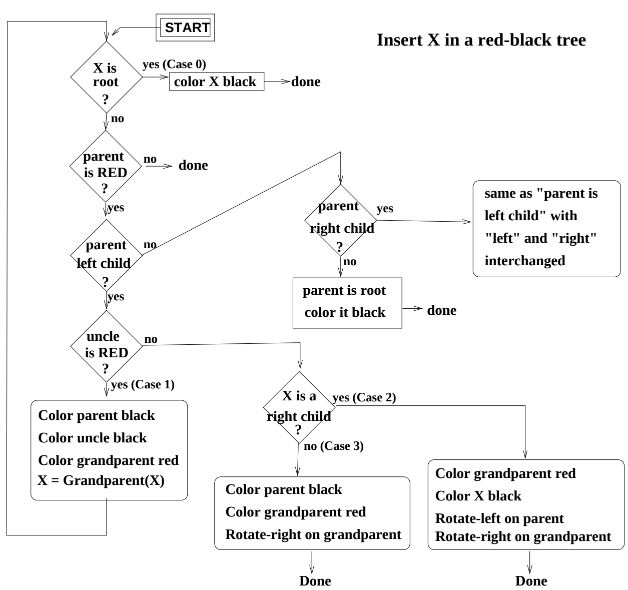
Cases

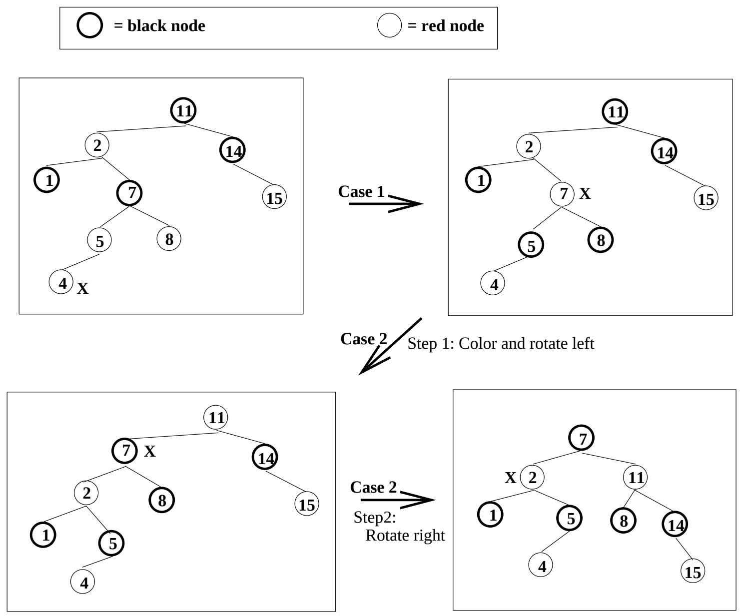
0: X is the root -- color it black

1: Both parent and uncle are red -- color parent and uncle black, color grandparent red, point X to grandparent, check new situation

2: Parent is red, but uncle is black. X and its parent are both left or both right children -- color parent black, color grandparent red, rotate right on grandparent

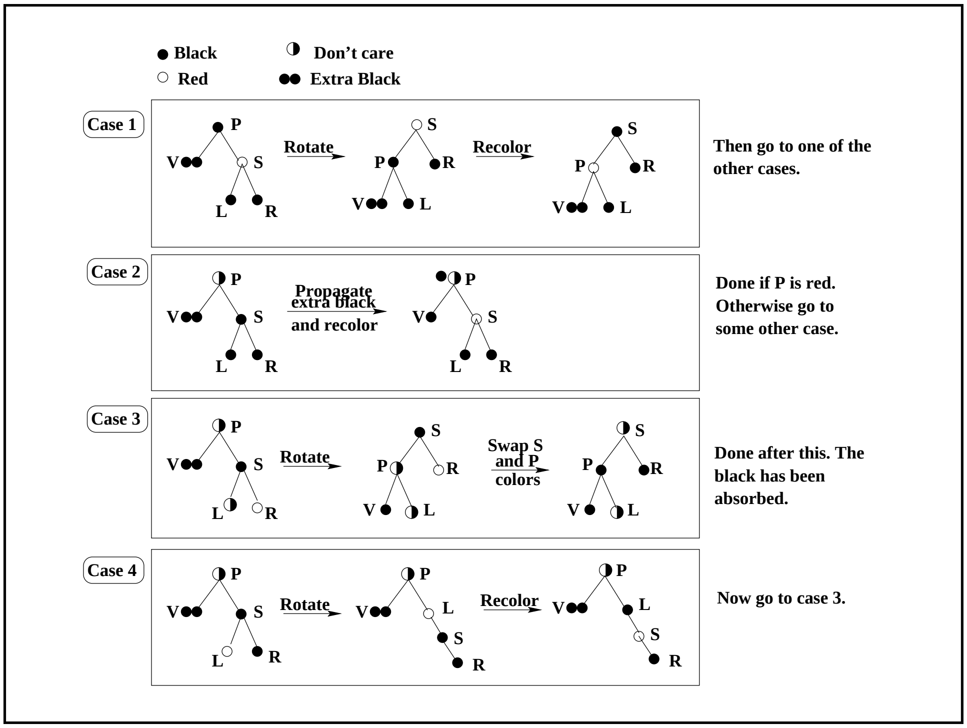
3: Parent is red, but uncle is black. X and its parent are opposite type children -- color grandparent red, color X black, rotate left on parent, rotate right on grandparent

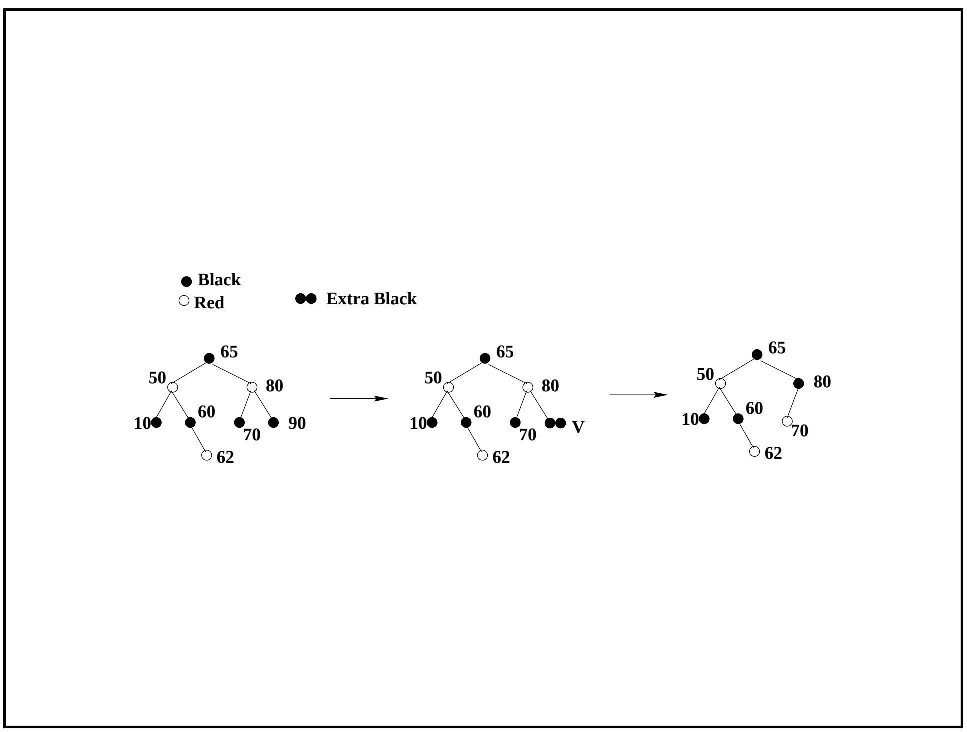




Reorient

```
template <class Comparable>
void RedBlackTree<Comparable>::handleReorient (const
  Comparable & item) {
  current->color = RED;
  current->left->color = current->right->color = BLACK;
  if (parent->color == RED) {
      grand->color = RED;
      if ((item < grand->element) != (item < parent-
  >element)
              parent = rotate(item, grand); // start dbl
  rotate
      current = rotate(item, great);
      current->color = BLACK;
      }
  header->right->color = BLACK; // make root black
  }
```


Black
Don't care
Red
Extra Black
Case 1
Rotate
Recolor
Then go to one of the other cases.
Case 2
Propagate extra black and recolor
Done if P is red. Otherwise go to some other case.
Case 3
Rotate
Swap S and P colors
Done after this. The black has been absorbed.
Case 4
Rotate
Recolor
Now go to case 3.


Black
Red
Extra Black

Rotate

```
template <class Comparable>
RedBlackNode<Comparable>
  *RedBlackTree<Comparable>::rotate(const Comparable
  &item, RedBlackNode<Comparable> *theParent) const; {
  if (item < theParent->element) {
      if (item < theParent->left->element)
             rotateWithLeftChild(theParent->left)  // LL
      else
             rotateWithRightChild(theParent->left) // LR
      return theParent->left;}
  else {
      if (item < theParent->right->element)
             rotateWithLeftChild(theParent->right) // RL
      else
             rotateWithRightChild(theParent->right) // RR
      return theParent->right;}
  }
```