CMSC 341
Lecture 16

Announcements

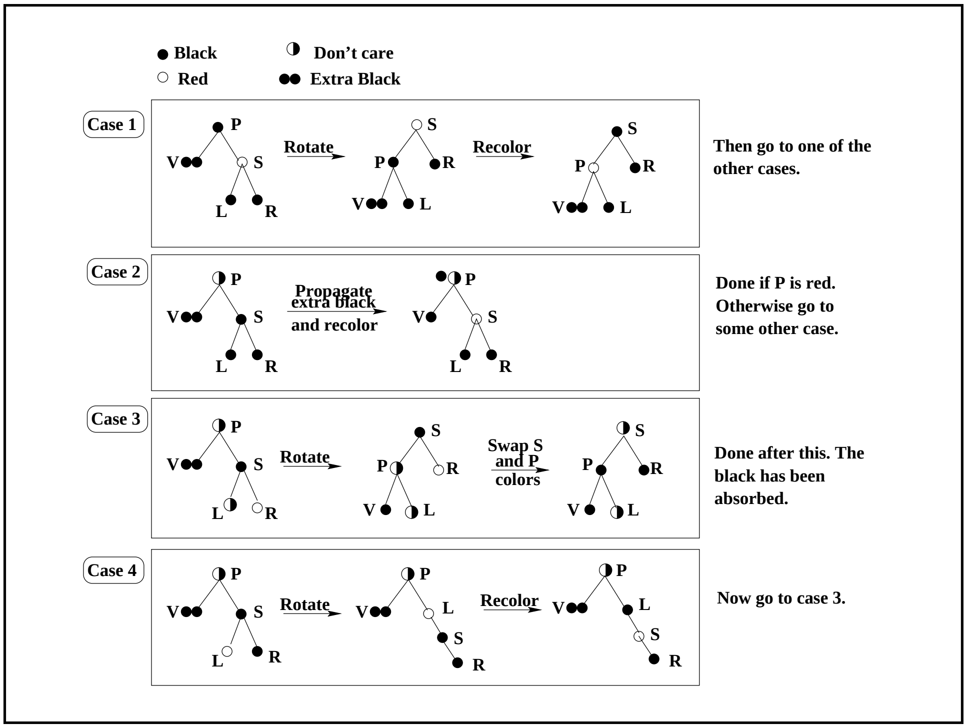
Second midterm on Monday
Exam review Wed and Thurs eves
Drop deadline Thurs

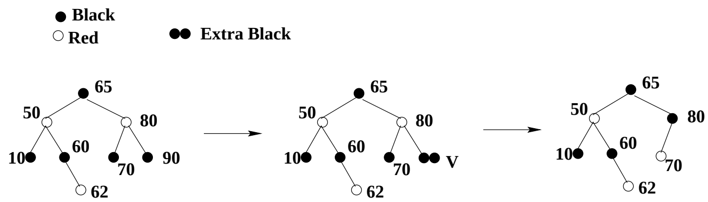
Bottom Up Insertion

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 (zig-zag): 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
- 3 (zig-zig): 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





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;}
   }
```

Rotate With Some Child

```
template <class Comparable>
void RedBlackTree<Comparable>::rotateWithLeftChild(
  RedBlackNode<Comparable> * & k2) const{
  RedBlackNode<Comparable> *k1 = k2->left;
  k2->left = k1->right;
  k1->right = k2;
  k2 = k1;
  }
template <class Comparable>
void RedBlackTree<Comparable>::rotateWithRightChild(
  RedBlackNode<Comparable> * & k1) const{
  RedBlackNode<Comparable> *k2 = k1->right;
  k1->right = k2->left;
  k2->left = k1;
  k1 = k2;
  }
```

Asymptotic Cost

O(lg n) to descend to insertion point

O(1) to do insertion

O(lg n) to ascend and readjust == worst case only for case 1

Total: O(log n)

Bottom Up Deletion

Normal BST Deletion

0: if node is leaf, just delete

1: if node has one child, replace it with child

2: if node has two children, replace value at node by value of its inorder predecessor, then recursively delete inorder predecessor

Eventually case 0 or 1 will be reached, where node is replaced by another node V. If node to be deleted then is red, no adjustments are needed. If black, adjustments must be made to preserve the number of black nodes.

Bottom Up Deletion (cont)

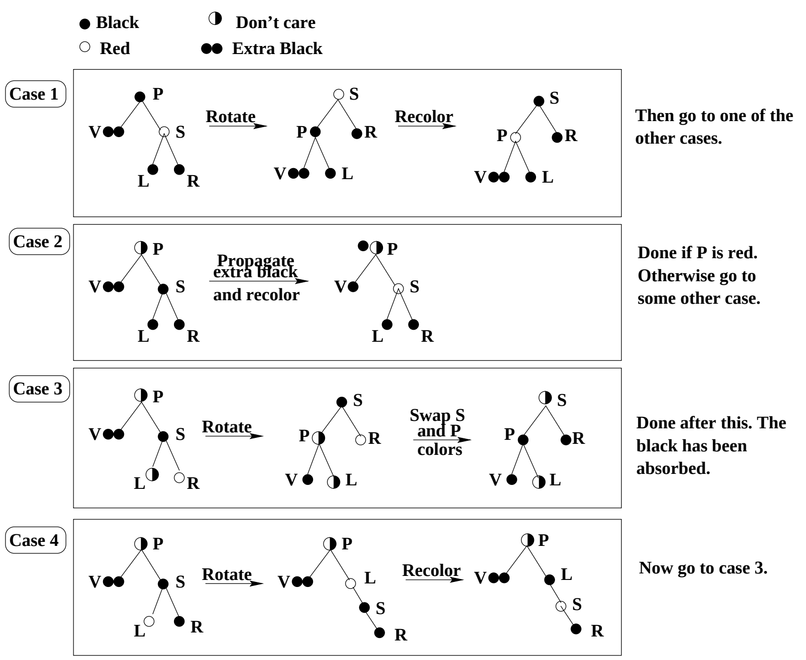
Cases:

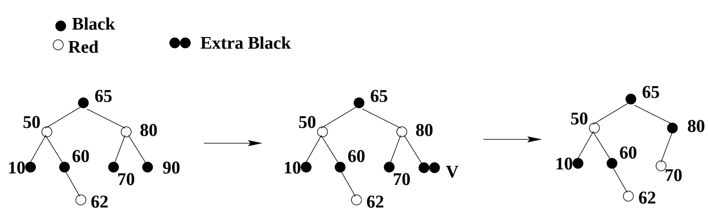
0: If S, sibling of V, is red. Do rotation and recoloring to produce a situtation that can be handled as another.

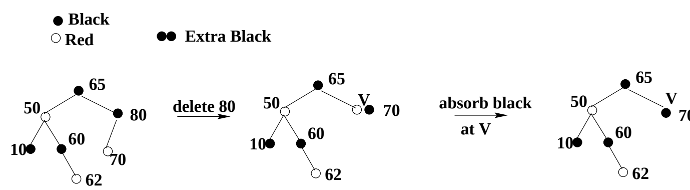
1: Node S is black and both its children are black. Color S red. If P is red, make P black and terminate. Otherwise, check again and continue.

2: Node S is black and its right child is red. Do a rotation followed by a color swap between S and P. Now done.

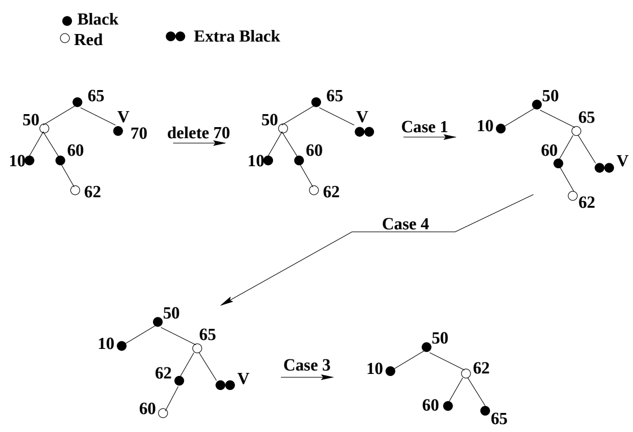
3: Node S is black, its left child is red, and its right child is black. Do a rotation around S and a recoloring. The situation can now be handled as 2.


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


Black
Red
Extra Black
65
50
80
10
60
70
90
62
65
50
80
10
60
70
V
62
65
50
80
10
60
70
62



Black
Red
Extra Black
65
50
80
10
60
70
62
delete 80
65
V
50
70
10
60
62
absorb black
at V
65
V
50
70
10
60
62



Black
Red
Extra Black
65
V
50
70
10
60
62
delete 70
65
V
50
10
60
62
Case 1
50
10
65
60
V
62
Case 4
50
10
65
62
V
60
Case 3
50
10
62
60
65

Top Down Implementation

Make adjustments on way down

Perform operation

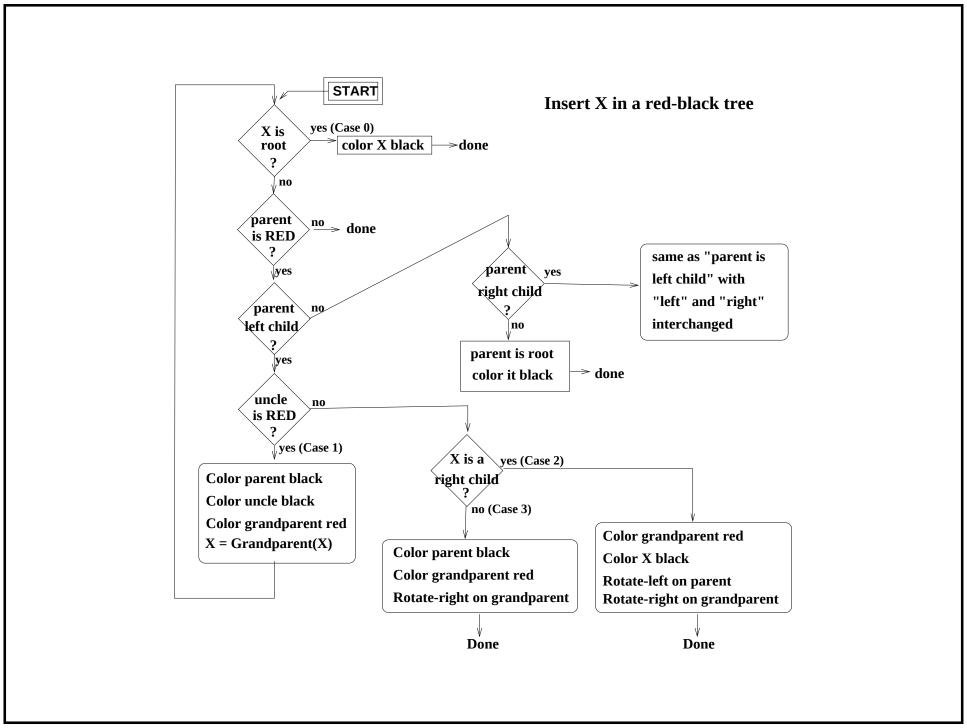
Top Down Insertion

Make adjustments on way down

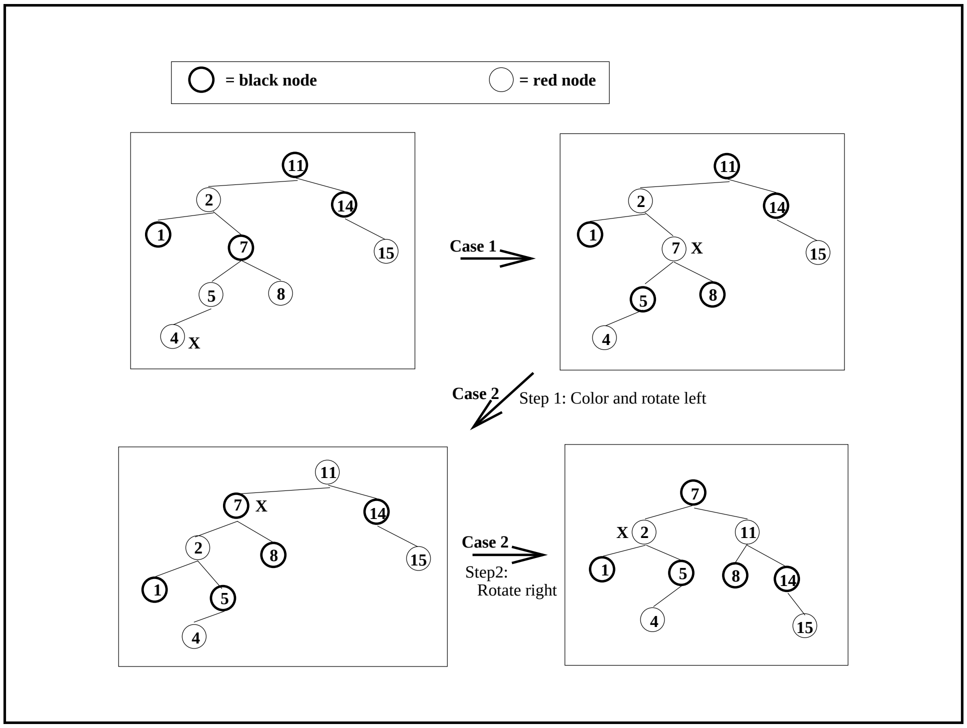
- if X has two red children, make X red and children black.
  - If P red, perform rotation to fix (P’s sibling can’t be red)

Insert as red node at leaf

- since parent was black, no further adjustments necessary



Insert X in a red-black tree
START
X is root ?
yes (Case 0)
color X black
done
no
parent is RED ?
no
done
yes
parent left child ?
no
yes
parent right child ?
yes
same as "parent is left child" with "left" and "right" interchanged
no
parent is root color it black
done
uncle is RED ?
no
yes (Case 1)
Color parent black
Color uncle black
Color grandparent red
X = Grandparent(X)
X is a right child ?
yes (Case 2)
no (Case 3)
Color parent black
Color grandparent red
Rotate-right on grandparent
Done
Color grandparent red
Color X black
Rotate-left on parent
Rotate-right on grandparent
Done



= black node
= red node
Case 1
Case 2
Step 1: Color and rotate left
Case 2
Step2:
Rotate right
X

Insert

```
template <class Comparable>
void RedBlackTree<Comparable>::insert(const Comparable &x)
  {
  current = parent = grand = header;
  nullNode->element = x;
  while (current->element != x) {
      great = grand; grand = parent; parent = current;
      current = (x < current->element) ?
            current->left : current->right;
      // if two red children, recolor
      if ((current->left->color == RED)  &&
                  (current->right->color) == RED))
            handleReorient(x);
      }
```

Insert (cont)

```
  // insertion fails if already present
  if (current != nullNode) return;
  current = new RedBlackNode<Comparable>(x,
                              nullNode, nullNode);

  // attach to parent
  if (x < parent->element)
      parent->left = current;
  else
      parent->right = current;
  handleReorient();
  }
```

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); // dbl rotate
      current = rotate(item, great);
      current->color = BLACK;
      }
  header->right->color = BLACK; //make root black
  }
```