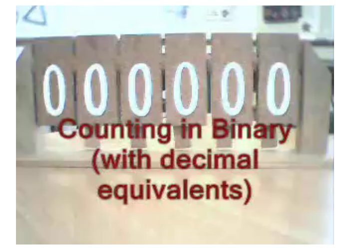
Binary Search Trees

Feedback summary

- Board work…
  - More / Less
  - Intuitive narrative
- Notes:
  - Find right balance of how much to include rely on your note-taking
- Highlight important ideas
- And when all else fails….



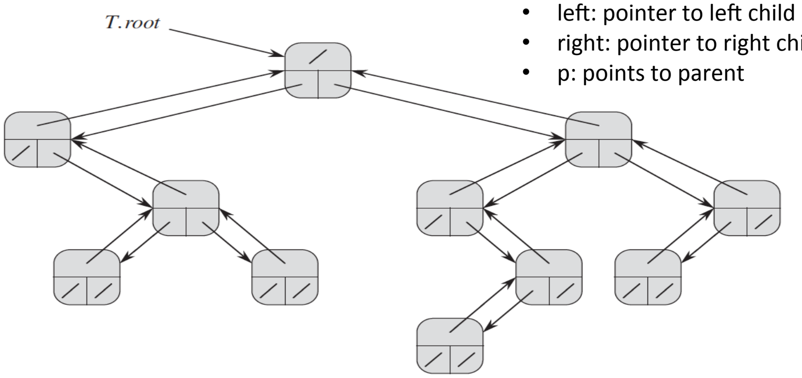
Upcoming

- Midterm review
  - Oct 3: Midterm review session
  - Edmunds 101, right before colloquium
  - Bonus Question
- Midterm
  - Take home
  - Due Tuesday, Oct 8

Overview

- Data structures that support many dynamic-set operations
- Can be used as a both a dictionary and as a priority queue
- Basic operations take time proportional to the height of the tree
  - Complete binary tree with *n* nodes: worst case $\Theta(\log n)$
  - For linear chain of *n* nodes: worst case $\Theta(n)$
- Different types of search trees include:
  - binary search trees (chapter 12),
  - red-black trees (chapter 13), and
  - B-trees (chapter 18)

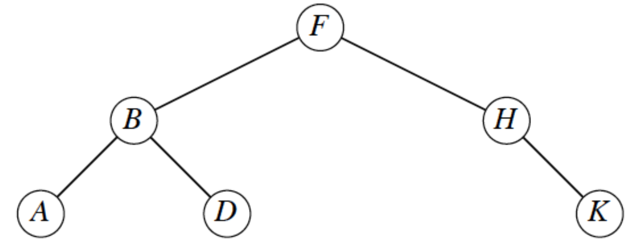
Quick Review: Binary Tree Representations



Each node contains:

- key (or other data)
- left: pointer to left child
- right: pointer to right child
- p: points to parent

Binary Tree Property



- Binary-search-tree property:
  – If *y* is in the left subtree of *x*, then *y*.key ≤ *x*.key
  – If *y* is in the right subtree of *x*, then *y*.key ≥ *x*.key

Inorder-Tree-Walk

```
INORDER-TREE-WALK(x)
 if x ≠ NIL
     INORDER-TREE-WALK(x.left)
     print key[x]
     INORDER-TREE-WALK(x.right)
```

- Idea:
  – Check that x is not null
  – Recursively, print the keys of nodes in *x*'s left subtree
  – Print *x*'s key
  – Recursively, print the keys of nodes in *x*'s right subtree

Inorder-Tree-Walk


```
INORDER-TREE-WALK(x)
 if x ≠ NIL
     INORDER-TREE-WALK(x.left)
     print key[x]
     INORDER-TREE-WALK(x.right)
```

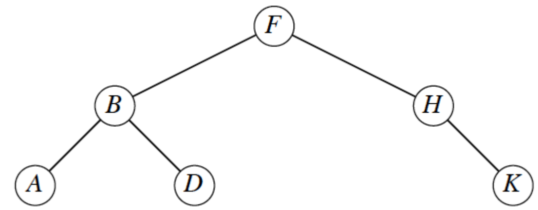

- Correct?
- Time?

Binary Search Tree Operations

- Queries
  - Search
  - Min / Max
  - Successor / Predecessor
- Insertion / deletion
- Cost?

Tree-Search

- Example
  - Find D
  - Find C
- Idea:
  - Start at root
  - If key == *k* or Null, return x
  - If key < *k*, recurse on left subtree
  - If key > *k*, recurse on right subtree

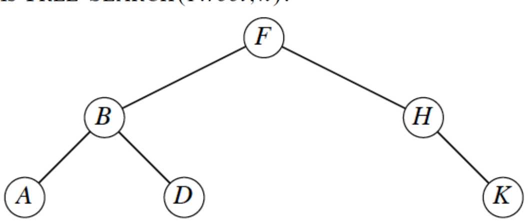


Tree-Search

```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

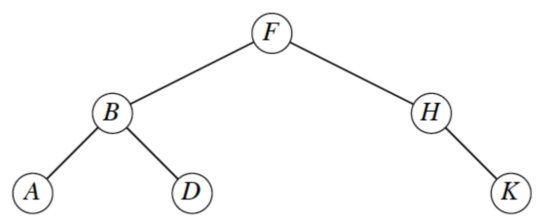
Initial call is TREE-SEARCH($T.root$, k).

- Time



Minimum and Maximum

- The binary-search-tree property guarantees that
  - The minimum key of a binary search tree is located:
  - The maximum key of a binary search tree is located:

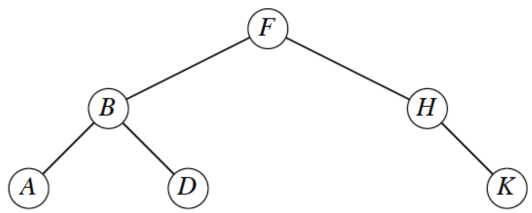


-

Minimum and Maximum

- Minimum / Maximum operations a special case of search
- Advantage of iterative formulation:
- Time?

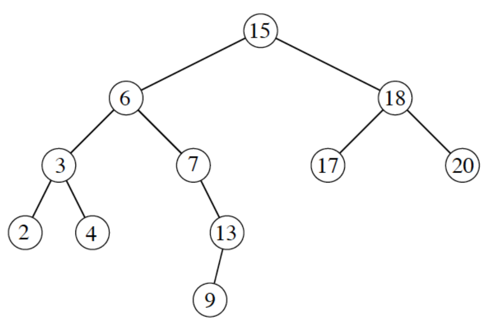
```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x

TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```



Successor / predecessor

- Assuming distinct keys, the successor of node *x* is the node *y* such that *y.key* is the smallest key > *x.key*
- No comparisons necessary! Why?

Successor / predecessor



- Successor of the node with key value:

Worksheet:

- Write pseudo-code for Successor
- Feel free to use previously defined operations…

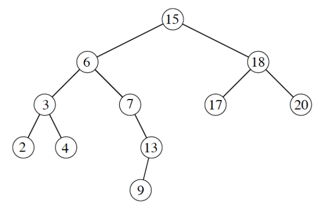
Successor

2 Cases:

- x has non-empty subtree
  - Return minimum of x's right subtree
- x has empty right subtree
  - Recursively visit parent node until we travel right

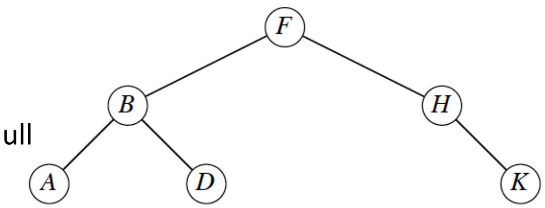
- Predecessor is symmetric
- Time?

TREE-SUCCESSOR(x)



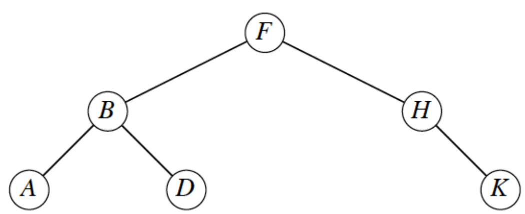
Insertion

- Where to insert?
  - Binary search
    - Stop when x = Null
    - Insertion point
  - Need a "trailing pointer"
    - Keep track of which parent to connect to



Insertion

```
TREE-INSERT(T, z)
 y = NIL
 x = T.root
 while x ≠ NIL
     y = x
     if z.key < x.key
         x = x.left
     else x = x.right
 z.p = y
 if y == NIL
     T.root = z       // tree T was empty
 elseif z.key < y.key
     y.left = z
 else y.right = z
```



Example: Insert(T,C)

Time:

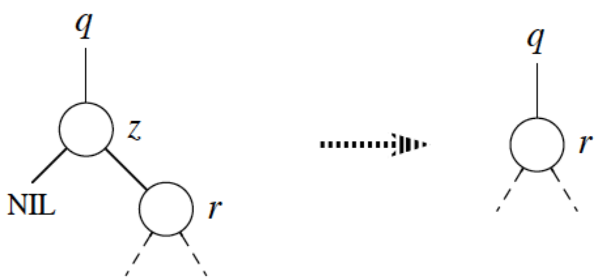
Delete

- Requires moving subtrees around within the binary search tree
- Transplant replaces *u* with *v*

```
TRANSPLANT(T, u, v)
 if u.p == NIL
     T.root = v
 elseif u == u.p.left
     u.p.left = v
 else u.p.right = v
 if v ≠ NIL
     v.p = u.p
```

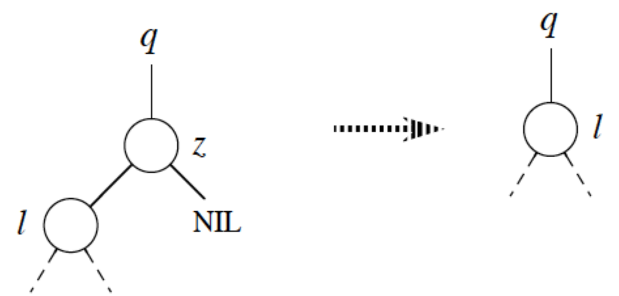
Deletion: Case 1 – no left child

- If *z* has no left child, replace z by right child
- Covers the case where x has no children



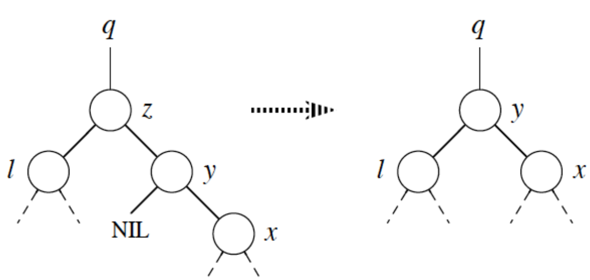
Deletion: Case 2 – just left child

- If z has one child to the left, replace z by left child



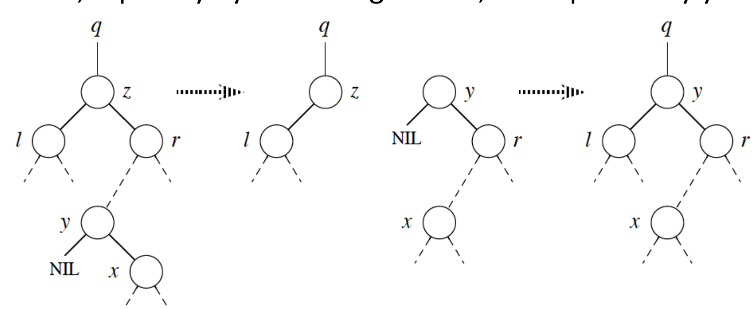
Deletion: Case 3 – two children

- Replace z with its successor y
- Y must lie in z's right subchild and have no left child (why?)
- If *y* is *z*'s right child, replace x by z, leaving *y* right child alone



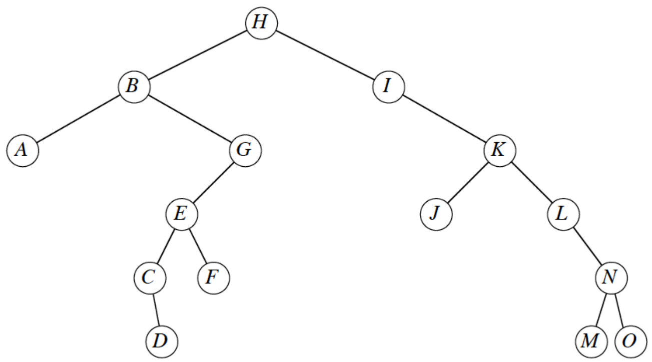
Deletion: Case 3 – two children

- Replace z with its successor y
- Y must lie in z's right subchild and have no left child (why?)
- Else, replace y by its own right child, the replace z by y



Tree-Delete Example

- Delete: I, G, K, B



Tree-Delete

```
TREE-DELETE(T, z)
 if z.left == NIL
     TRANSPLANT(T, z, z.right)          // z has no left child
 elseif z.right == NIL
     TRANSPLANT(T, z, z.left)           // z has just a left child
 else // z has two children.
     y = TREE-MINIMUM(z.right)          // y is z's successor
     if y.p ≠ z
         // y lies within z's right subtree but is not the root of this subtree.
         TRANSPLANT(T, y, y.right)
         y.right = z.right
         y.right.p = y
     // Replace z by y.
     TRANSPLANT(T, z, y)
     y.left = z.left
     y.left.p = y
```

Time:

- All lines constant time besides Tree-Minimum

Minimizing running time

- We've been analyzing running times in terms of *h* rather than *n*
- Problem: worst-case height for binary search tree is: $\Theta(n)$
  - Depends on order of insertion / deletion
  - E.g., {A,B,C,D,E,F}, {B,O,E,R,K,O,E,L}
  - Who's the most unbalanced?
- Solution(s):
  - Analyze expected case? (Section 12.4)
  - Restructure tree to guarantee small height (balanced tree): $O(\log n)$

Red-Black Tree

- A variation of binary search trees
- ***Balanced***: height is $O(\log n)$
- Operations will take $O(\log n)$ in the worst case

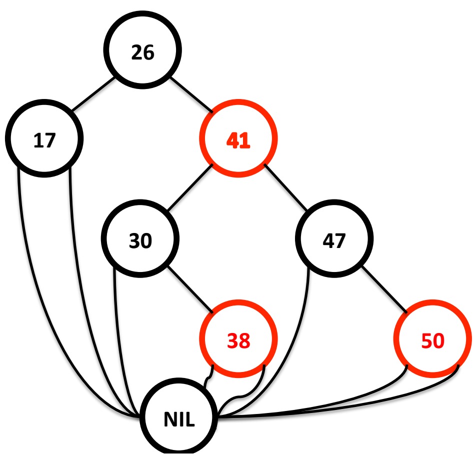
Red-black tree

- A **red-black** tree is a binary search tree +1 bit per node: a *color* attribute, which is either red or black
- All leaves are empty and colored black
  - Use single black node, *T.nil*, for root and all leaves of tree
- Inherits all other properties of a binary search tree

Red-Black Tree Properties

1. Every node is either red or black.
2. The root is black.
3. Every (null) leaf is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

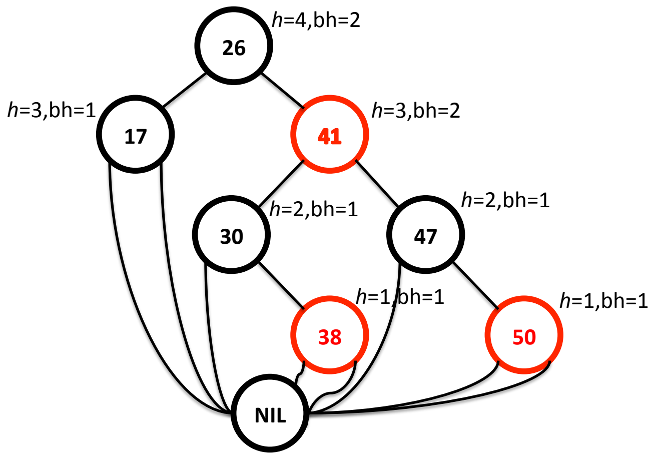
Red-Black Tree Example



Height of a Red-Black Tree

- **Height of a node:** is the number of edges in a longest path to a leaf
- **Black-height of a node *x***: bh(*x*) is the number of black nodes (including NIL) on the path from *x* to a leaf, not including *x*.
  - Well-defined due to property 5

Red-Black Tree Example



Worksheet: Prove

- **Claim 1**: Any node with height h has black-height ≥ $h/2$.

Worksheet: Prove

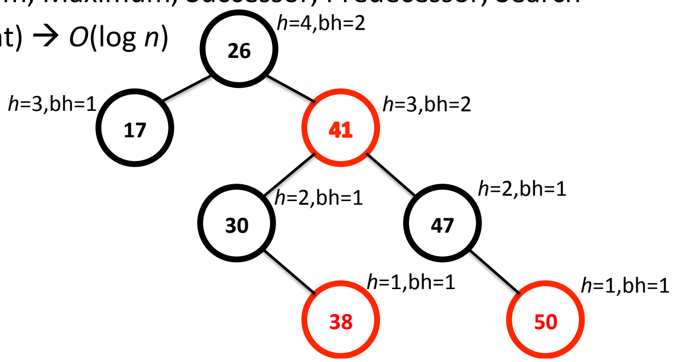
- **Claim 2**: Any node with height h has black-height ≥ $h/2$.

The key to success

- **Lemma:** A red black tree with n internal nodes has height ≤ $2\log(n+1)$.
- **Proof:** Let h and b be the height and black-height of the root. Then,

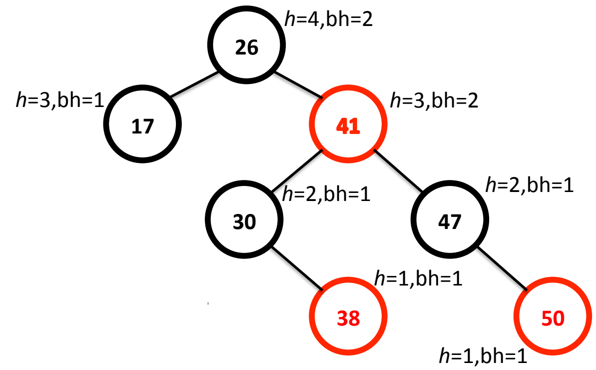
Operations on Red-black trees

- Non-modifying operations run as before
  - Minimum, Maximum, Successor, Predecessor, Search
  - *O*(height) → *O*(log *n*)



Inserting

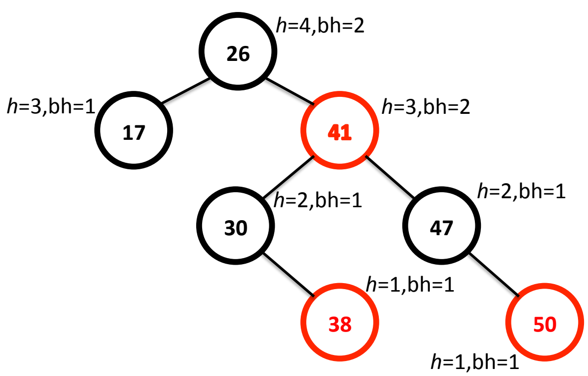
- When inserting a new key, e.g. 40, should it be:



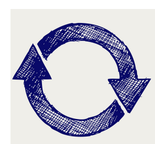
  - Red?
  - **Black?**

Deleting

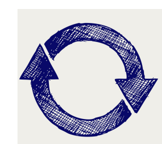
- When deleting an existing key, what happens if it's:



  - Red?
  - **Black?**

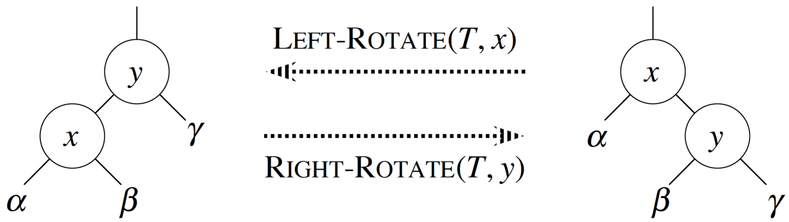


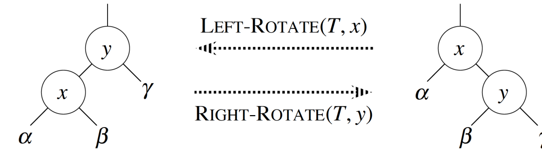
Rotations



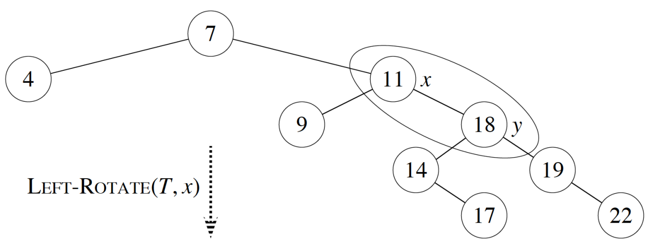
- The basic tree-restructuring operation
- Needed to maintain red-black trees as balanced binary search trees
- Changes only the local pointer structure.
- Does not impact binary tree property
- Both left and right rotations (inverse of each other)
- Takes as input: red-black tree and a node within the tree

Rotates





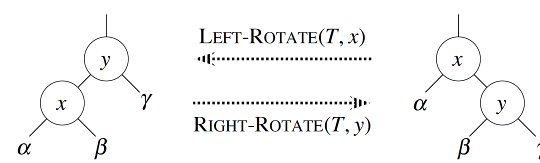

```
LEFT-ROTATE(T, x)
  y = x.right
  x.right = y.left
  if y.left ≠ T.nil
      y.left.p = x
  y.p = x.p
  if x.p == T.nil
      T.root = y
  elseif x == x.p.left
      x.p.left = y
  else x.p.right = y
  y.left = x
  x.p = y
```



Rotate Analysis

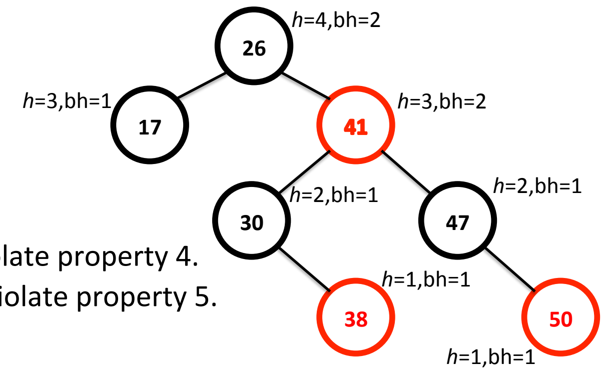
```
LEFT-ROTATE(T, x)
  y = x.right
  x.right = y.left
  if y.left ≠ T.nil
      y.left.p = x
  y.p = x.p
  if x.p == T.nil
      T.root = y
  elseif x == x.p.left
      x.p.left = y
  else x.p.right = y
  y.left = x
  x.p = y
```

- Time:
- Used in AVL and splay trees
- Can also talk about rotating on edge, rather than node



Inserting

- When inserting a new key, e.g. 40, should it be:



  - **Red?** Could violate property 4.
  - **Black?** Could violate property 5.

```
RB-INSERT(T, z)
 y = T.nil
 x = T.root
 while x ≠ T.nil
     y = x
     if z.key < x.key
         x = x.left
     else x = x.right
 z.p = y
 if y == T.nil
     T.root = z
 elseif z.key < y.key
     y.left = z
 else y.right = z
 z.left = T.nil
 z.right = T.nil
 z.color = RED
 RB-INSERT-FIXUP(T, z)
```

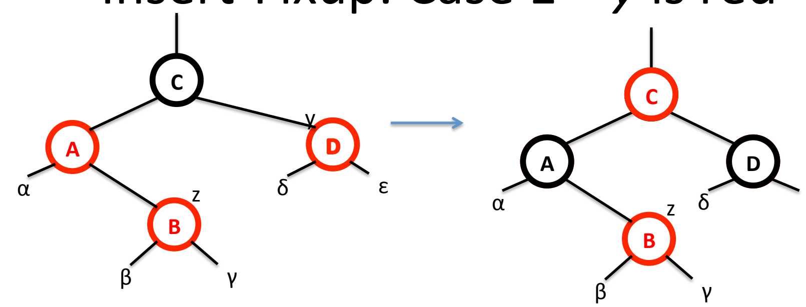
Insertion

- Idea:
  - Run insertion as before
  - Set z to red
  - Fix possible violations:
  1.
  2.
  3.
  4.
  5.

Insert-Fixup

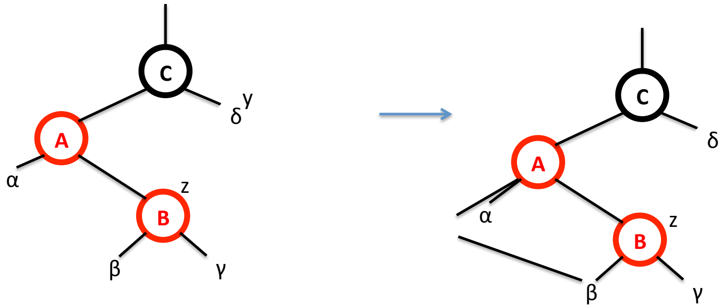
```
RB-INSERT-FIXUP(T, z)
 while z.p.color == RED
     if z.p == z.p.p.left
         y = z.p.p.right
         if y.color == RED
             z.p.color = BLACK                          // case 1
             y.color = BLACK                            // case 1
             z.p.p.color = RED                          // case 1
             z = z.p.p                                  // case 1
         else if z == z.p.right
                 z = z.p                                // case 2
                 LEFT-ROTATE(T, z)                      // case 2
             z.p.color = BLACK                          // case 3
             z.p.p.color = RED                          // case 3
             RIGHT-ROTATE(T, z.p.p)                     // case 3
     else (same as then clause with "right" and "left" exchanged)
 T.root.color = BLACK
```

Insert-Fixup: Case 1 – *y* is red



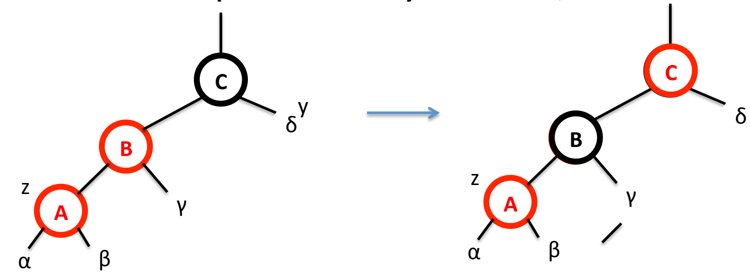
- *z.p.p* must be black
- Make *z.p* and *y* black
- Make *z.p.p* red
- Set *z.p.p* as the new *z*

Insert-Fixup: Case 2 – *y* is black, z is right child



- Rotate *z* and *z.p*

Insert-Fixup: Case 3 – *y* is black, z is left child



- Make *z.p* black and *z.p.p* red
- Rotate on *z.p.p*

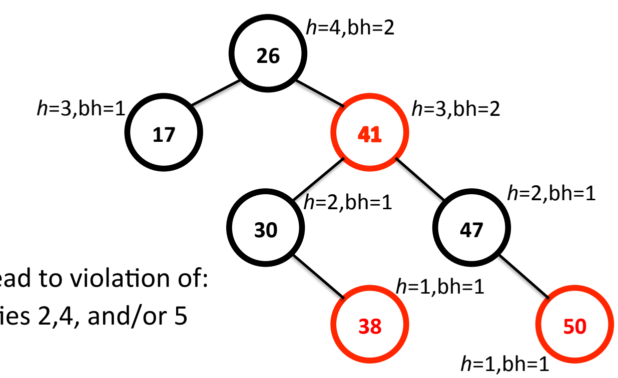
- No longer have 2 reds in a row
- *z.p* is now black → no more iterations

RB-Insert Analysis

- *O*(log *n*) time for original insert
- Within RB-Insert-Fixup
  - Each iteration takes *O*(1) time
  - Each iteration either fixes the tree or moves *z* up two levels
  - There are at most 2 rotations overall
  - *O*(log *n*) levels → *O*(log *n*) time

Deleting

- When deleting an existing key, what happens if it's:
  - **Red?** OK
  - **Black?** Could lead to violation of: properties 2,4, and/or 5



Deletion

```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if y.p == z
          x.p = y
      else RB-TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      RB-TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
      y.color = z.color
  if y-original-color == BLACK
      RB-DELETE-FIXUP(T, x)
```

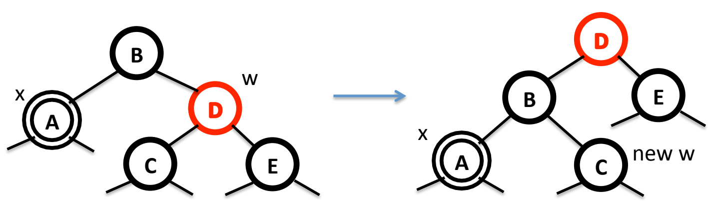
- Idea:
  - Run deletion (mostly) as before
  - If y's original color black, fix possible violations:
  1.
  2.
  3.
  4.
  5.

RB Delete Fixup

```
RB-DELETE-FIXUP(T, x)
  while x ≠ T.root and x.color == BLACK
      if x == x.p.left
          w = x.p.right
          if w.color == RED
              w.color = BLACK                                   // case 1
              x.p.color = RED                                   // case 1
              LEFT-ROTATE(T, x.p)                               // case 1
              w = x.p.right                                     // case 1
          if w.left.color == BLACK and w.right.color == BLACK
              w.color = RED                                     // case 2
              x = x.p                                           // case 2
          else if w.right.color == BLACK
                  w.left.color = BLACK                          // case 3
                  w.color = RED                                 // case 3
                  RIGHT-ROTATE(T, w)                            // case 3
                  w = x.p.right                                 // case 3
              w.color = x.p.color                               // case 4
              x.p.color = BLACK                                 // case 4
              w.right.color = BLACK                             // case 4
              LEFT-ROTATE(T, x.p)                               // case 4
              x = T.root                                        // case 4
      else (same as then clause with "right" and "left" exchanged)
  x.color = BLACK
```

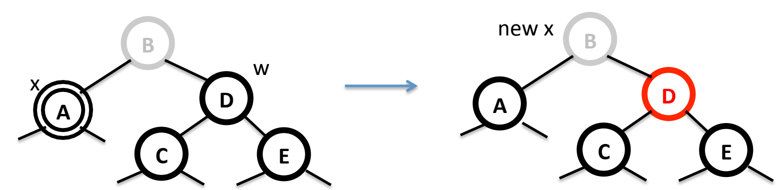
- Idea: Move the extra black up the tree until
  - x points to a red & black node → turn it into a black node
  - x points to the root → just remove the extra black, or
  - We can otherwise fix it
- Within the while loop:
  - x always points to a nonroot, doubly black node
  - w is x's sibling

Delete-Fixup: Case 1 – *w* is red



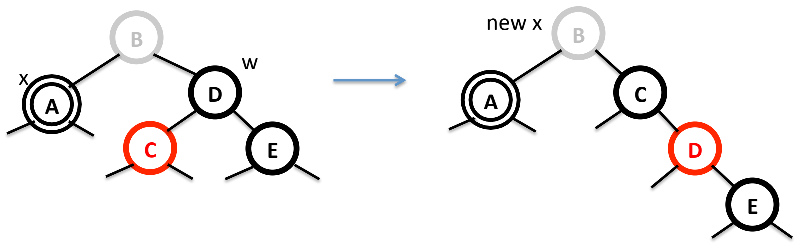
- W must have black children
- Make w black and x.p red
- Left rotate on x.p
- New sibiling of x must now be black
- Goto case 2,3, or 4

Delete-Fixup: Case 2 – *w* is black & both children are black



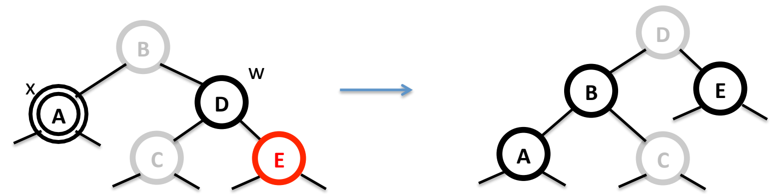
- Take 1 black off *x* and *w*
- Move black to *x.p*
- Do next iteration with *x.p* as new *x*
- If *new x* had been red, break and *new x* to black

Delete-Fixup: Case 3 – *w* is black
& *w*'s left child is red, right child is black



- Make *w* red and *w*'s left child black
- Right rotate on *w*
- → case 4

Delete-Fixup: Case 4 – *w* is black
& *w*'s right child is red



- Make *w* be *x.p*'s color
- Make *x.p* black and *w*'s right child black
- Left rotate on *w*
- Remove extra black on x
- Set *x* to root

RB-Delete Analysis

- $O(\log n)$ time to delete
- RB-Delete-Fixup
  - Only Case 2 iterates
    - (all others progress to next case or halt)
    - *X* moves up one level
  - Each of cases 1, 3, and 4 has 1 rotation → ≤ 3 rotations in all
  - Hence $O(\log n)$ overall

Wrap-up

- Data structures that support many dynamic-set operations
- Can be used as a both a dictionary and as a priority queue
- Basic operations take time proportional to the height of the tree
  - Complete binary tree with *n* nodes: worst case $\Theta(\log n)$
  - For linear chain of *n* nodes: worst case $\Theta(n)$
- Red-black trees
  - Variation of binary search tree
  - Balanced: height is $O(\log n)$
  - Operations take $O(\log n)$ in the worst case