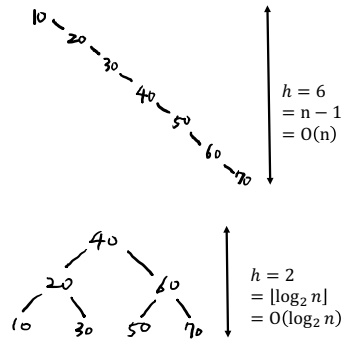


Balanced Binary Search Trees

An Idea To Avoid Worst Case Binary Search Tree (Degeneration Into List)

Limitations of Ordinary BST

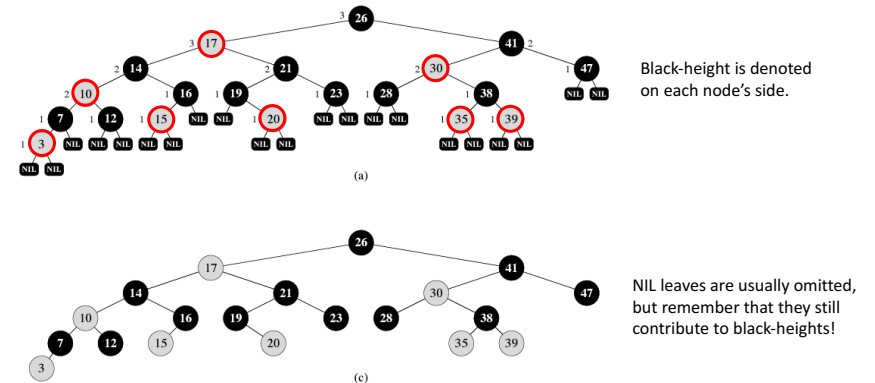
- $O(h)$ time complexities for all core operations (search/insert/delete/...)
- What is the resulting *ordinary* BST when values are inserted in the following order?
10, 20, 30, 40, 50, 60, 70
- What is the *optimal* BST for the above input sequence?
 - Optimal in the sense of smallest height h .



Red-Black Tree

- Special kind of *BST* with additional node property (color=red/black only) and following additional requirements so that the worst case tree height is still guaranteed to be $O(\lg n)$, not $O(n)$:
 1. Every node is either red or black.
 2. The root is black.
 3. Every leaf (NIL) is black. A value-bearing node is NOT considered a leaf.
 4. If a node is red, then both its children are black.
 - No two consecutive reds along any simple downward path.
 5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
 - $bh(x)$: Black-height of node x , denoting # black nodes on any simple path from x , not including x itself

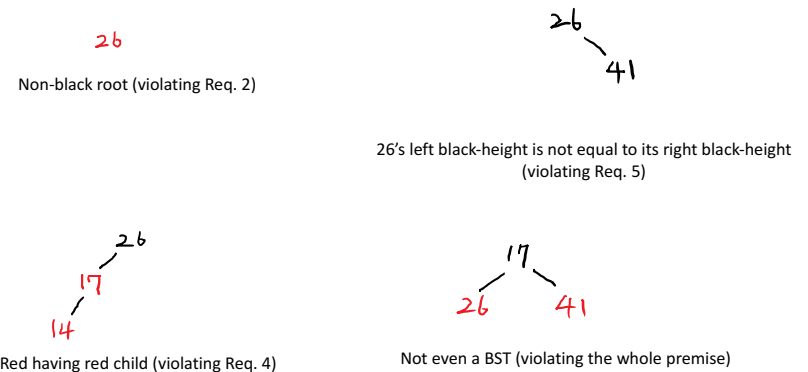
Red-Black Tree Example (CLRS pp.310)



How To Distinguish Non-Red-Black Tree

- Requirements 1, 2, 3 are trivial:
 - Every node is either red or black.
 - Root is black.
 - Every leaf (NIL) is black.
- Requirement 4 is not too hard:
 - If a node is red, then both its children are black.
- Requirement 5 is probably the hardest to check:
 - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes
 - Technique: Calculate/write black-height from bottom-up!

Non-Red-Black Tree Examples

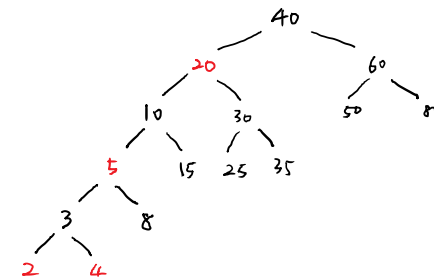


Balanced Nature of Red-Black Trees

- Lemma (CLRS pp.309): A red-black tree with n internal nodes has height at most $2 \lg(n + 1)$.
 - Sub-lemma: The subtree rooted at any node x has at least $2^{bh(x)} - 1$ internal nodes.
 - Proof by induction: Read/understand carefully.
 - There's an intuitive understanding to this lemma, though.
 - Another observation: The black-height of the root must be at least $h/2$.
 - Because of requirement 4, at least half the nodes on any simple path from the root to a leaf must be black.
- Thus, applying the above sub-lemma on the root, we get:
 - $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$
 - Moving 1 and taking logarithms, we get $h \leq 2 \lg(n + 1)$.

Intuition On Red-Black Tree's Height Bound

- Worst case left/right height difference (skewness) could be at most twice!



- You can't add any child to 2 or 4, without first adding other nodes on other parts of the red-black tree, thus limiting any more deviation!

Inserting in a Red-Black Tree

Insert As Done On An Ordinary BST, Then Fix To Recover Red-Black Properties

Lesson Objectives

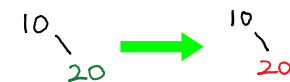
- Identify the resulting red-black tree after inserting an arbitrary new value into a given red-black tree.
- Given an incomplete Red-Black tree insertion code, fill in the blanks for correct Red-Black tree insertion operations.

Inserting New Value, Retaining R-B Properties

- Recall inserting 10, 20, 30, 40, 50, 60, 70 in sequence to an ordinary BST
- What if the BST needs to be R-B tree all the time?
 - Find the insertion spot, treating the given R-B tree as an ordinary BST
 - Determine the color of the inserted node
 - So that the chosen color doesn't violate the R-B properties
 - If neither choice is possible,
 - Fix the tree structure!

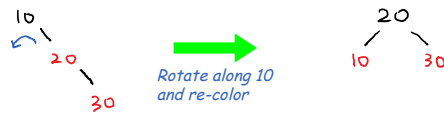
Trivial Insertions

- Inserting into an empty R-B tree:
 - The inserted value occupies the root node, which must be colored black.
- If insertion location is a black node's child:
 - The inserted node can be simply colored red, without violating any R-B properties (no consecutive reds, same black heights everywhere).

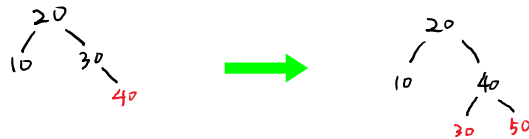


Nontrivial Insertions

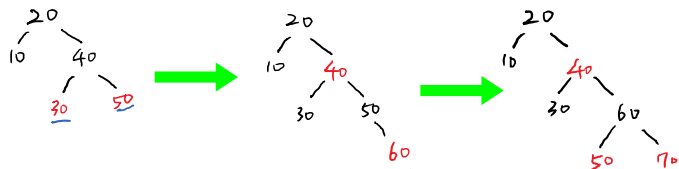
- If insertion location is a red node's child:
 - The newly inserted node can't be black (matching-left-and-right-black-heights rule broken).
 - The newly inserted node can't be red either (no-consecutive-reds rule broken).
 - Then how?
 - Keep one rule that's harder to enforce (same black heights), and fix the other rule that's broken (no consecutive reds).
 - That is, insert as a red node, breaking the no-consecutive-reds rule, and fix it along the path to root.



- Insert 50: Just rotate-and-recolor (50's uncle is not red, but black).

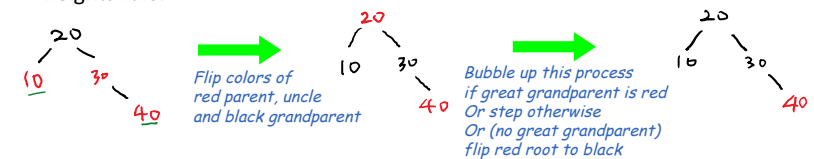


- Inserting 60, 70, 80, ... : Same pattern, but may bubble up.

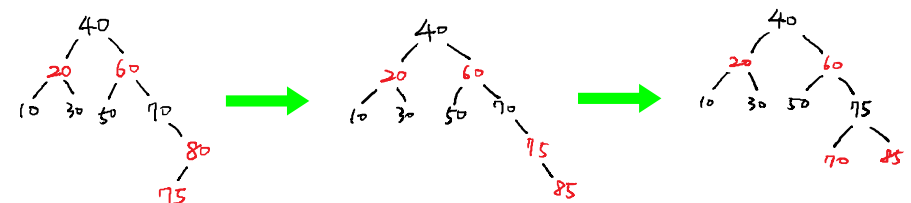


More Nontrivial Insertions

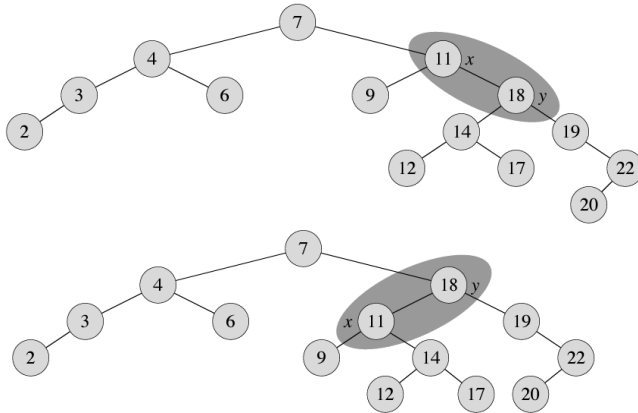
- Insert 40:
 - Rotate along 20 & recolor doesn't work, because 40's uncle is red.
 - Instead of rotating, we can simply flip colors of 40's parent, uncle & grandparent, and yet still meeting the same-black-heights rule!
 - Then the grandparent and the great grandparent need to be checked for consecutive reds, and this process repeats (bubbling up).
 - If there's no great grandparent, then the grandparent is root, which is now red, but can be simply repainted black, without violating the same-black-heights rule.



- Inserting 75: Just to show a different situation (Case 2 in CLRS)—We can easily transform it to a well-known case (Case 3 in CLRS).

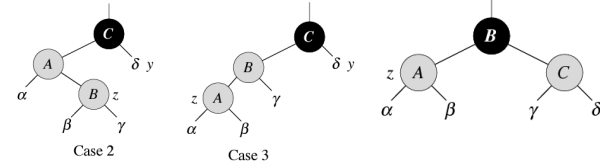


BST Rotation Example (CLRS Fig. 13.3)



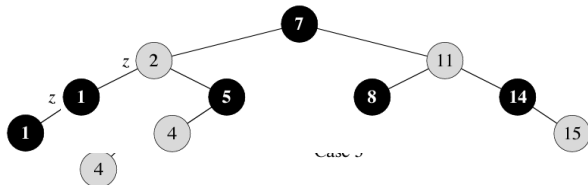
Rotating Red-Black Tree

- When to rotate:
 - For the red violating node (z), its parent is red, its uncle is black.
 - Its grandparent must be always black.
- CLRS Fig. 13.6 (δ is a subtree whose root is black)

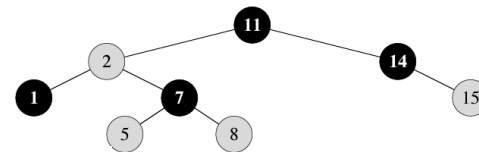


- After this process, there's no more consecutive-reds-violation!

Red-Black Tree Rotation (Fix-Up) Example (CLRS Fig. 13.4)



Actual Code: Finding Spot

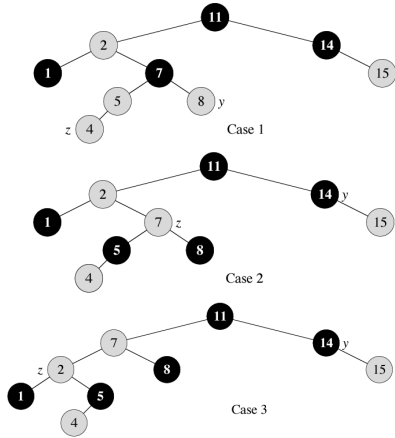


RB-INSERT(T, z)

```

1   $y = T.nil$ 
2   $x = T.root$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11  elseif  $z.key < y.key$ 
12      $y.left = z$ 
13  else  $y.right = z$ 
14   $z.left = T.nil$ 
15   $z.right = T.nil$ 
16   $z.color = RED$ 
17  RB-INSERT-FIXUP( $T, z$ )
    
```

Actual Code: Fix-Up



```

RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK
6              y.color = BLACK
7              z.p.p.color = RED
8              z = z.p.p
9          else if z == z.p.p.right
10             z = z.p
11             LEFT-ROTATE(T, z)
12             z.p.color = BLACK
13             z.p.p.color = RED
14             RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
           with "right" and "left" exchanged)
16  T.root.color = BLACK
    
```

Time Complexity Analysis

- Insertion (as a red node): $O(h)$, obviously
 - Traversing downward along the path to a leaf.
- Fix-up (resolving consecutive reds): $O(h)$ too!
 - Traversing upward along the path to root at most once.
 - In each iteration of RB-INSERT-FIXUP(T, z)'s while loop,
 - There are fixed number of operations
 - Each iteration pushes up z one level up
 - The loop can iterate at most all the way up to root, which is h times.
- Therefore, $O(h) = O(\lg n)$ all the time (incl. worst case)!

Deleting Existing Value from a Red-Black Tree

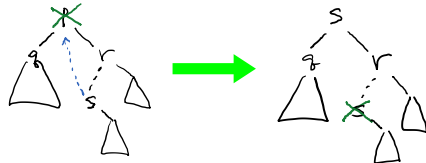
Allow Extra Black On Any Node, Bubble It Up Or Pass It Over
Still Very Complicated!

Lesson Objectives

- Identify the resulting red-black tree after deleting an arbitrary existing value from a given red-black tree.
- Given an incomplete Red-Black tree deletion code, fill in the blanks for correct Red-Black tree deletion operations.

Problem Reduction Of RB-DELETE

- Only consider cases of deleting a node with at most one non-NIL child
 - If the value to be deleted is found at a node with two children,
 - Find its successor node (which can't have a left child)
 - Copy the successor value to the original node to be deleted
 - Then delete the successor node (move up its right child to its position)

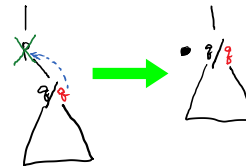


DELETE(p) → Copy p's successor's value to p's node, then DELETE(p's successor node)

Note: In CLRS, it's not copying & deleting, but TRANSPLANTING twice! (s.right to s, s to p)

More Problem Reduction

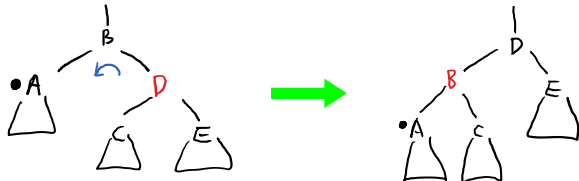
- Deleting a red node is straightforward
 - In fact, if the node to be deleted is red (with at most one child), then it must be a value-bearing leaf (with no value-bearing child). Can be easily removed without violating any red-black properties
- Deleting a black node is complicated
 - Move up its right child (from previous slide)
 - However, the deleted black node's "black" color has to stay.
 - Giving an **extra black** to the node that's moved up to the deleted black node
 - This extra black needs to be fixed up.



- Note that q might be NIL!
- If q is NIL, then of course q is black, and the NIL now has extra black.
- If q is not NIL and red, then it can be simply recolored to black and we are done.
- Once this step is done, there's no more transplanting and the extra black will need to be fixed.

How To Fix Extra Black

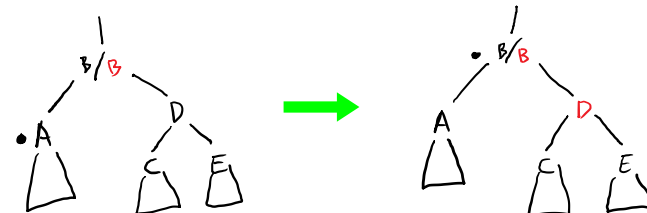
- Case 1: Extra black node's sibling is red
 - The red sibling must have two black children (same-black-heights rule) and its parent must be black too (no-consecutive-reds rule).
 - Rotate the tree along the parent and make the extra black node's sibling black (Transform to Case 2)



Recolor B & D so that the black-height property is still met.

Note: Any of A, C, and E may be NIL!

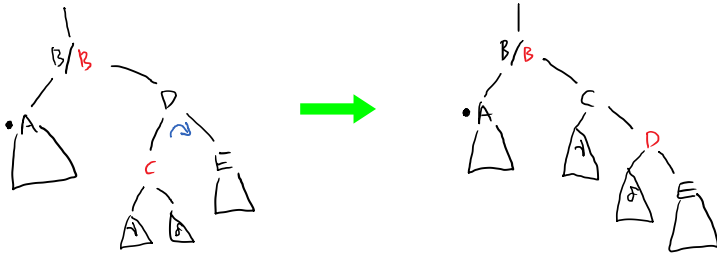
- Case 2: Extra black node's sibling and its two children are all black
 - We can recolor the sibling to red, and bubble up the extra black to the parent.
 - If the parent was originally red, it can be simply recolored to black, and we are done. Otherwise, continue fixing up the extra black.



- Black-height requirement is still satisfied
- And the consecutive reds can be easily fixed by recoloring with the extra black
- Or the extra black can be bubbled up/passed along again (iteration)

Note: Any of A, C, and E may be NIL!

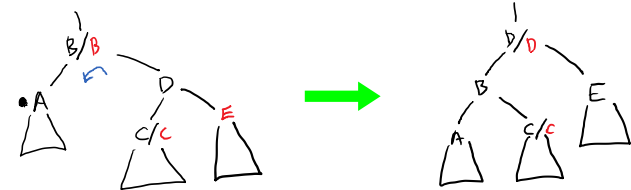
- Case 3: Extra black node's sibling and only its right child is black
 - Rotate along the sibling, and recolor so that it's transformed to Case 4.



- Recolor C & D so that the black heights are still the same along any path to a leaf.
- This is Case 4.

Note: Any of A, γ , δ and E may be NIL!

- Case 4: Extra black node's sibling is black and its right child is red.
 - Rotate & recolor (maintaining the same black heights)
 - The extra black is gone, and we are done.



Recolor B, D, E as necessary to retain the same black heights and to remove the extra black.

Note: Any of A and C may be NIL!

RB-DELETE(T, z)

```

1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4    x = z.right
5    RB-TRANSPLANT(T, z, z.right)
6  elseif z.right == T.nil
7    x = z.left
8    RB-TRANSPLANT(T, z, z.left)
9  else y = TREE-MINIMUM(z.right)
10   y-original-color = y.color
11   x = y.right
12   if y.p == z
13     x.p = y
14   else RB-TRANSPLANT(T, y, y.right)
15     y.right = z.right
16     y.right.p = y
17   RB-TRANSPLANT(T, z, y)
18   y.left = z.left
19   y.left.p = y
20   y.color = z.color
21  if y-original-color == BLACK
22    RB-DELETE-FIXUP(T, x)

```

Actual Code:
RB-TRANSPLANT(), RB-DELETE()

RB-TRANSPLANT(T, u, v)

```

1  if u.p == T.nil
2    T.root = v
3  elseif u == u.p.left
4    u.p.left = v
5  else u.p.right = v
6  v.p = u.p

```

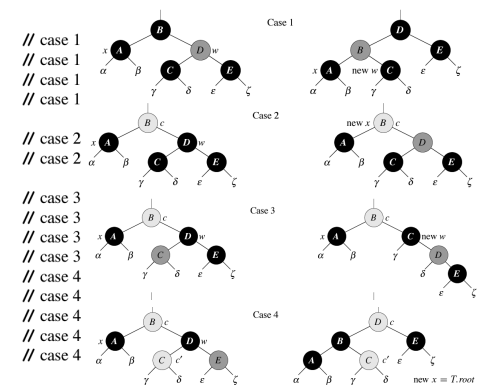
RB-DELETE-FIXUP(T, x)

```

1  while x ≠ T.root and x.color == BLACK
2    if x == x.p.left
3      w = x.p.right
4      if w.color == RED
5        w.color = BLACK
6        x.p.color = RED
7        LEFT-ROTATE(T, x.p)
8        w = x.p.right
9      if w.left.color == BLACK and w.right.color == BLACK
10       w.color = RED
11       x = x.p
12     else if w.right.color == BLACK
13       w.left.color = BLACK
14       w.color = RED
15       RIGHT-ROTATE(T, w)
16       w = x.p.right
17     w.color = x.p.color
18     x.p.color = BLACK
19     w.right.color = BLACK
20     LEFT-ROTATE(T, x.p)
21     x = T.root
22   else (same as then clause with "right" and "left" exchanged)
23   x.color = BLACK

```

RB-DELETE-FIXUP()



Time Complexity Analysis

- Still $O(h) = O(\lg n)$.
 - Case 3 & 4: Fixed # operations & terminates
 - Case 1: Fixed # operations, transforms to Case 2.
 - Case 2: Fixed # operations,
 - Then terminates if the extra black node's parent is red
 - Or else repeat, but one-level up, meaning it can repeat only up to h many times.
 - Thus $O(h)$!
- We achieved $O(\lg n)$ for all operations in all cases (incl. worst)

There are Other Balanced Trees

- AVL Trees: BSTs balanced by height
 - For any node, subtrees should have height difference of at most 1
 - This implies that the worst tree has size $n(h) = n(h-1) + n(h-2) + 1$
 - Then $h \leq 1.44 \log_2(n)$
- Weight-balanced BSTs
 - For any node, subtrees should have a bounded size ratio:
$$1 - \beta \leq \text{size}(\text{left})/\text{size}(\text{right}) \leq 1 + \beta$$
 - Then $h \leq c(\beta) \log_2(n)$