Balanced Binary Search Trees

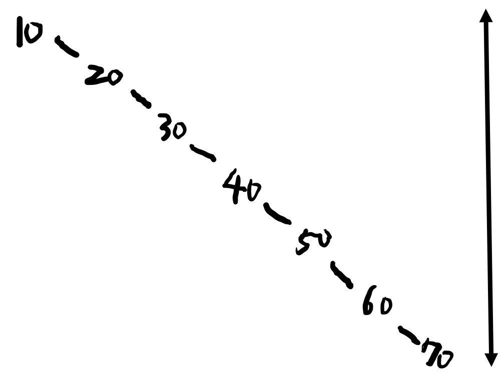
An Idea To Avoid Worst Case Binary Search Tree (Degeneration Into List)

Limitations of Ordinary BST

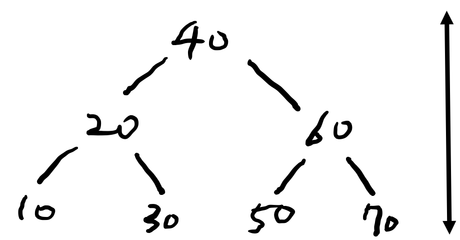
- $O(h)$ time complexities for all core operations (search/insert/delete/…)
- What is the resulting *ordinary* BST when values are inserted in the following order?

$$10, 20, 30, 40, 50, 60, 70$$

- What is the *optimal* BST for the above input sequence?
    - Optimal in the sense of smallest height h.



$h = 6$
$= \mathrm{n} - 1$
$= \mathrm{O}(\mathrm{n})$

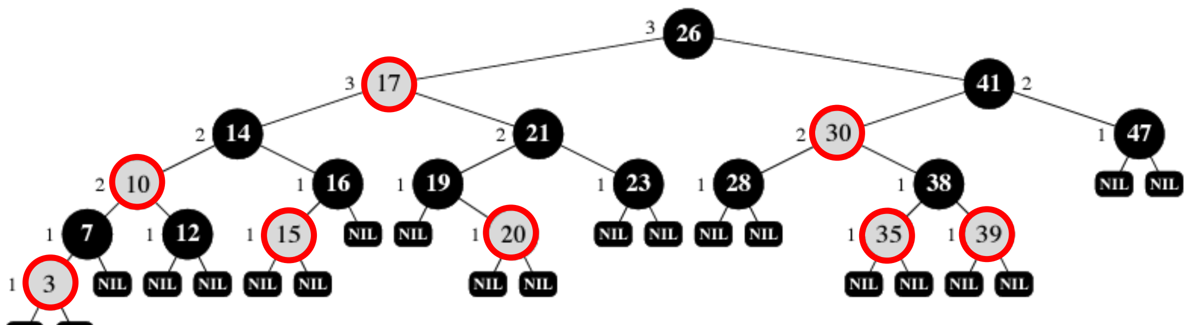


$h = 2$
$= \lfloor \log_2 n \rfloor$
$= \mathrm{O}(\log_2 n)$

Red-Black Tree

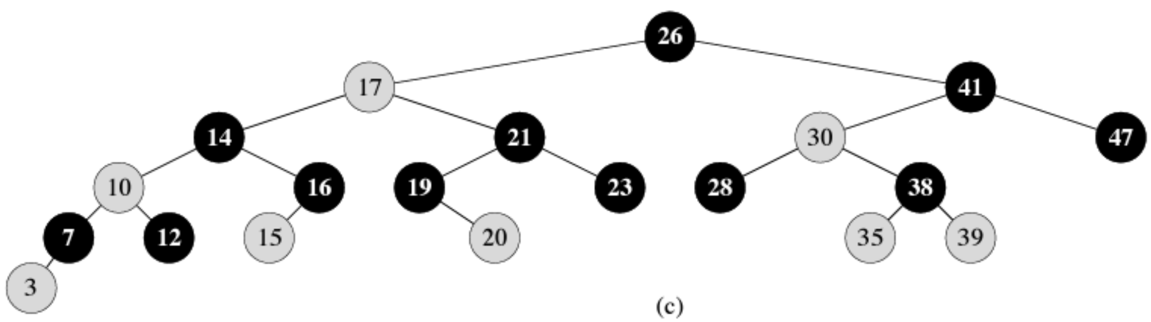
- Special kind of ***BST*** with additional node property (color=red/black only) and following additional requirements so that the worst case tree height is still guaranteed to be $O(\lg n)$, not $O(n)$:
  1. Every node is either red or black.
  2. The root is black.
  3. Every leaf (NIL) is black. A 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)



(a)

Black-height is denoted on each node's side.



(c)

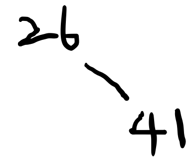
NIL leaves are usually omitted, but remember that they still contribute to black-heights!

How To Distinguish Non-Red-Black Tree

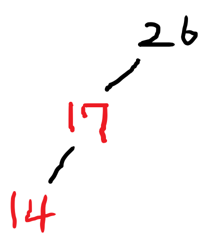
- Requirements 1, 2, 3 are trivial:
  1. Every node is either red or black.
  2. Root is black.
  3. Every leaf (NIL) is black.
- Requirement 4 is not too hard:
  4. If a node is red, then both its children are black.
- Requirement 5 is probably the hardest to check:
  5. For each node, all simple paths 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

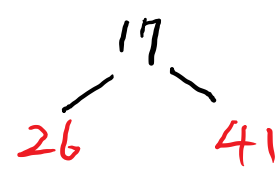
Non-black root (violating Req. 2)



26’s left black-height is not equal to its right black-height (violating Req. 5)



Red having red child (violating Req. 4)



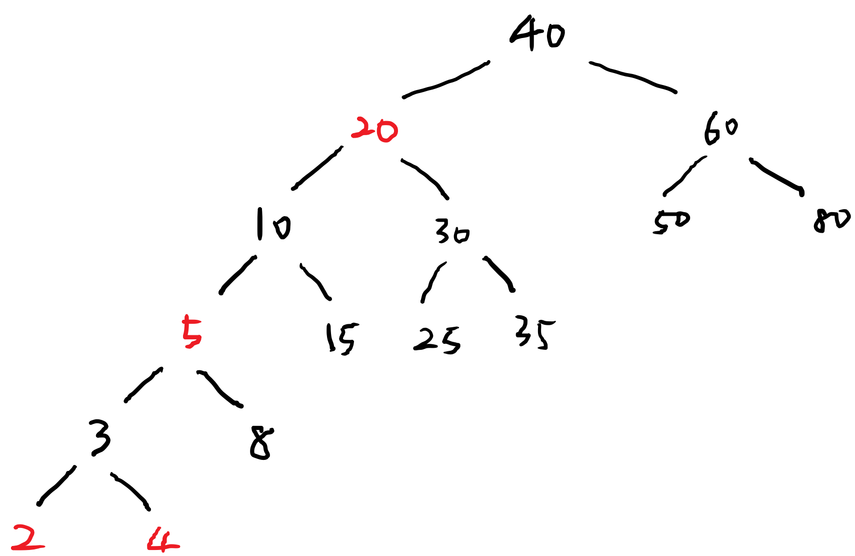
Not even a BST (violating the whole premise)

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

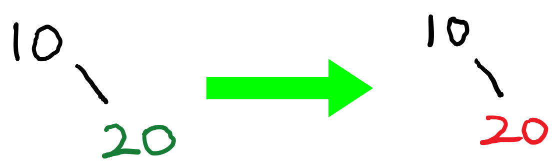
- Identify the resulting red-black tree after inserting an arbitrary new value into a given red-black tree.
- Given an incomplete Red-Black tree insertion code, fill in the blanks for correct Red-Black tree insertion operations.

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!

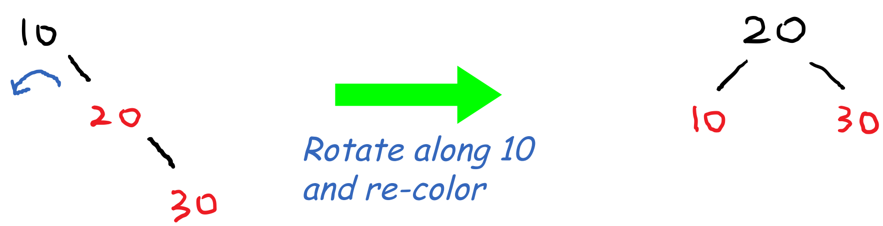
Trivial Insertions

- 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).



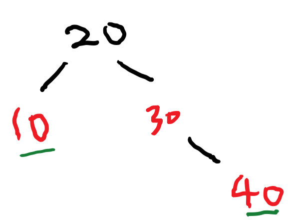
Nontrivial Insertions

- 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.

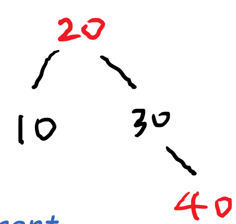


More Nontrivial Insertions

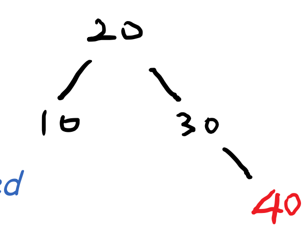
- Insert 40:
  - Rotate along 20 & recolor doesn’t work, because ***<u>40’s uncle is red</u>***.
  - Instead of rotating, we can simply flip colors of 40’s parent, uncle & grandparent, and yet still meeting the same-black-heights rule!
  - Then the grandparent and the great grandparent need to be checked 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.



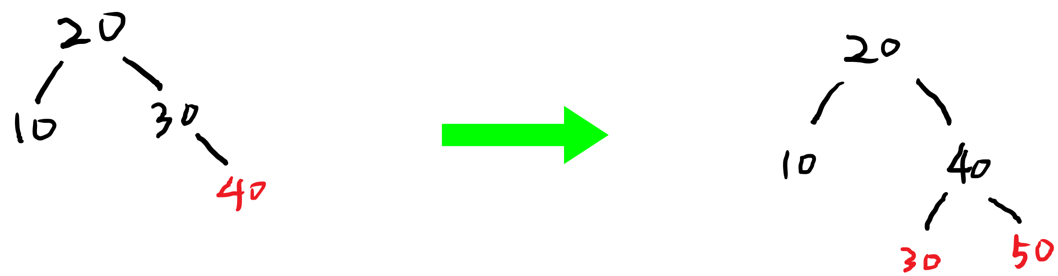
Flip colors of
red parent, uncle
and black grandparent



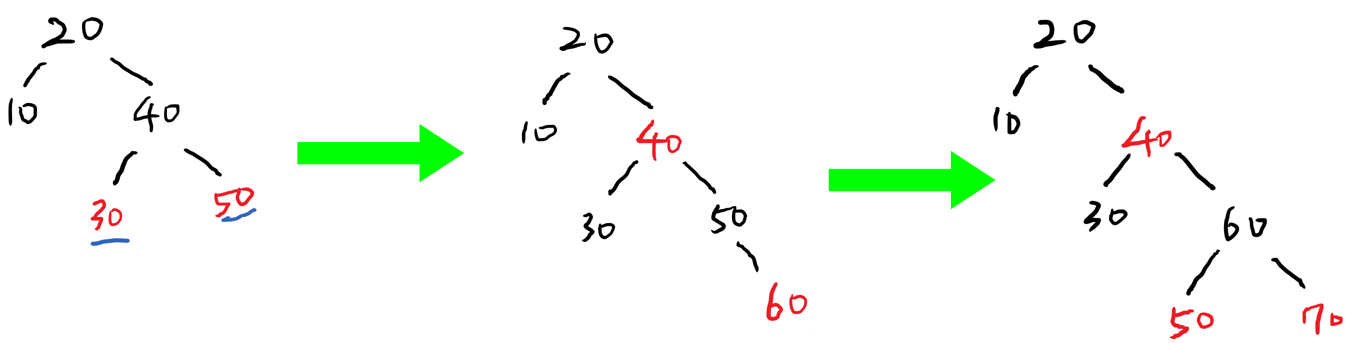
Bubble up this process
if great grandparent is red
Or step otherwise
Or (no great grandparent)
flip red root to black



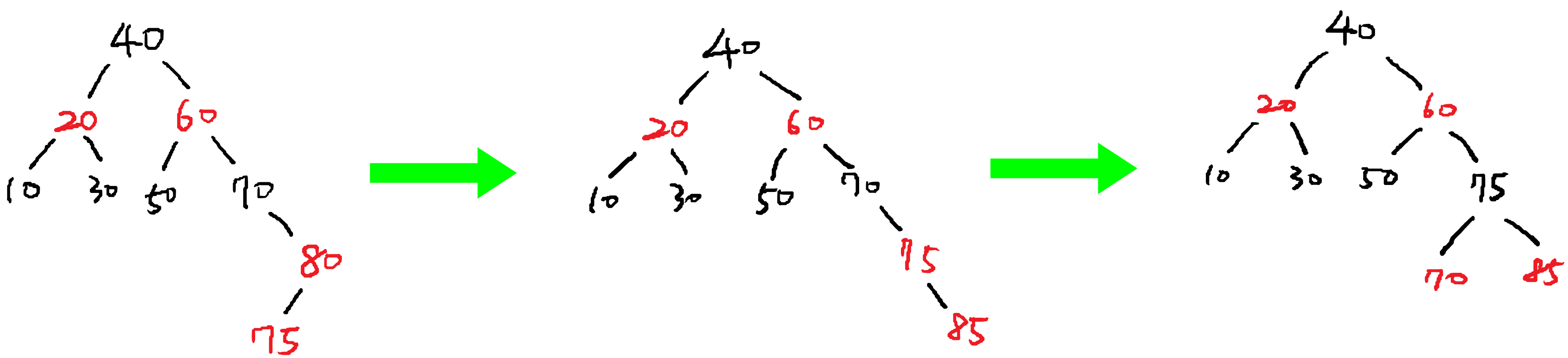
- Insert 50: Just rotate-and-recolor (50's uncle is not red, but black).



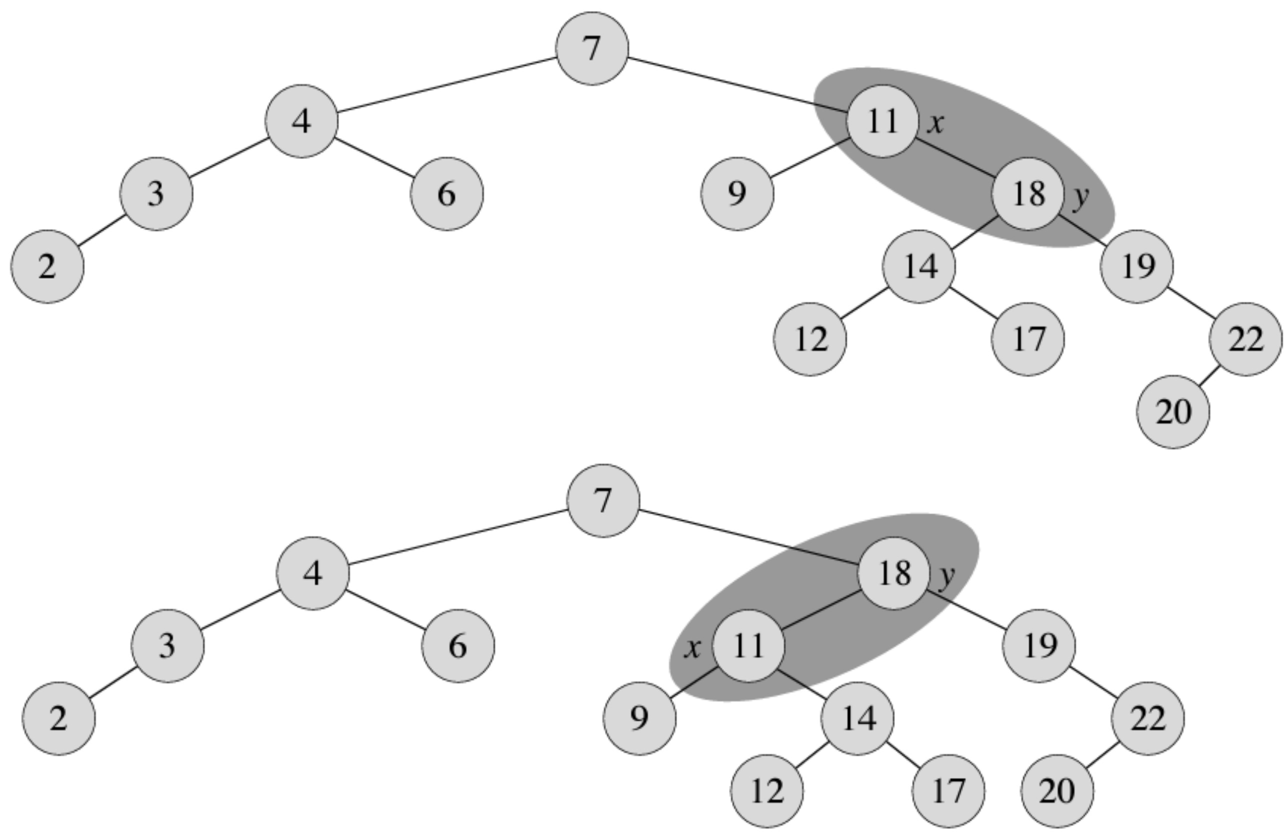
- Inserting 60, 70, 80, … : Same pattern, but may bubble up.



- 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).

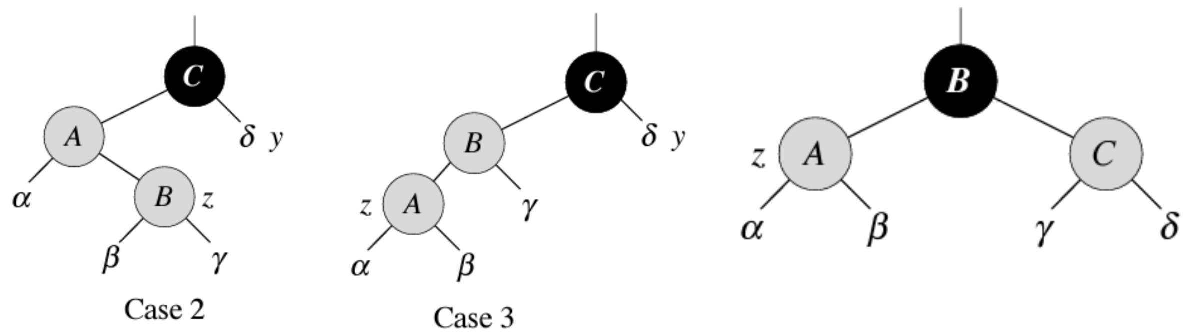


BST Rotation Example (CLRS Fig. 13.3)



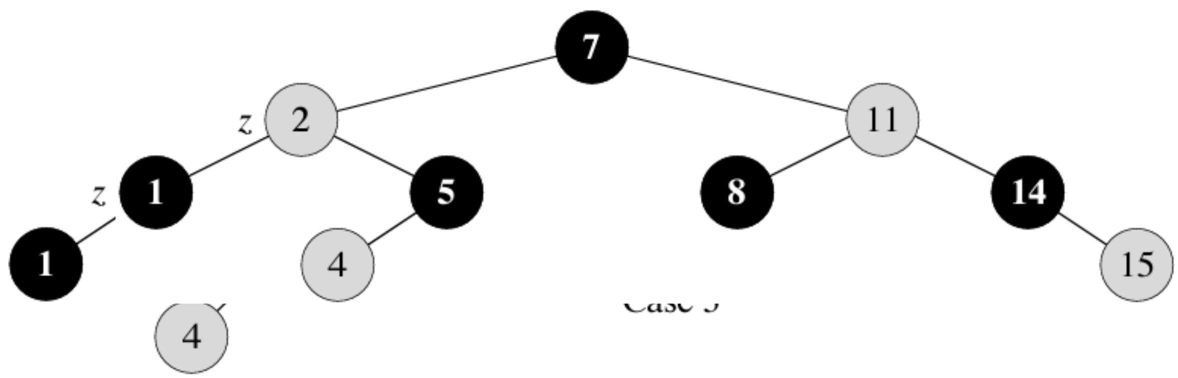
Rotating Red-Black Tree

- 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)

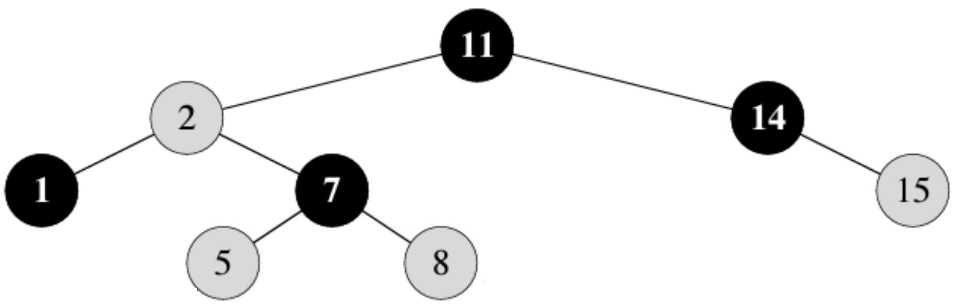


- After this process, there's no more consecutive-reds-violation!

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



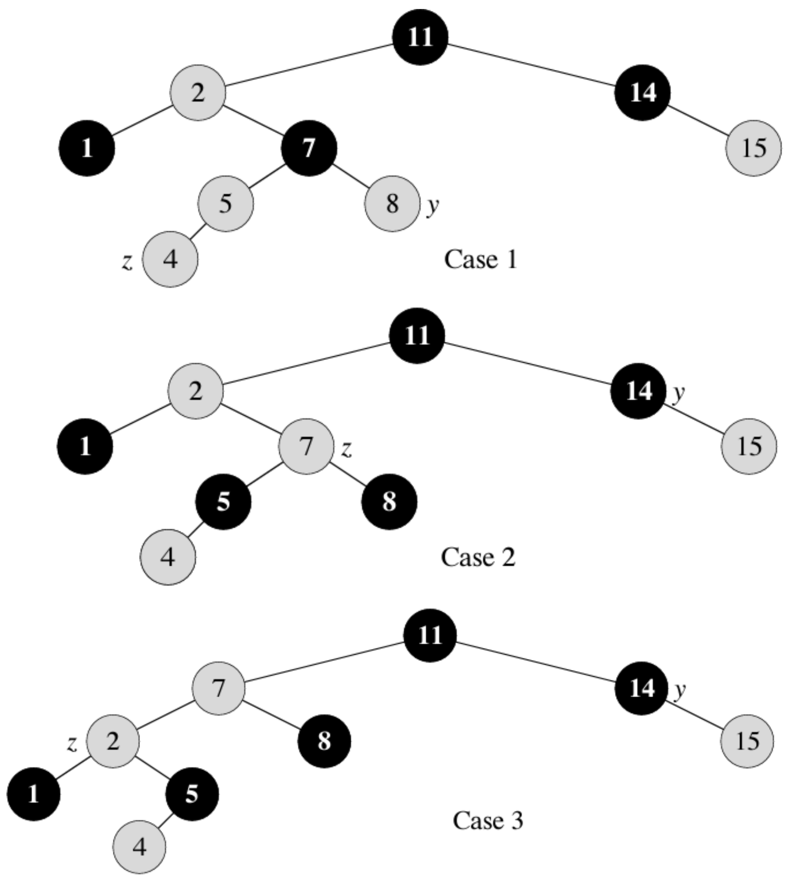
Actual Code: Finding Spot



RB-INSERT(T, z)

```
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)
```

Actual Code: Fix-Up




```
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
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
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

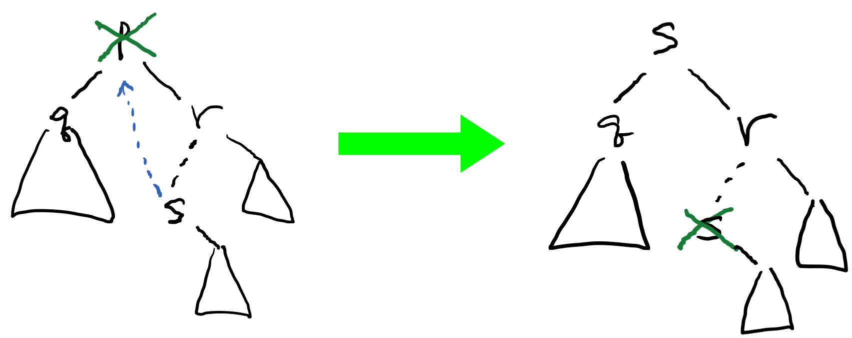
Still Very Complicated!

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)

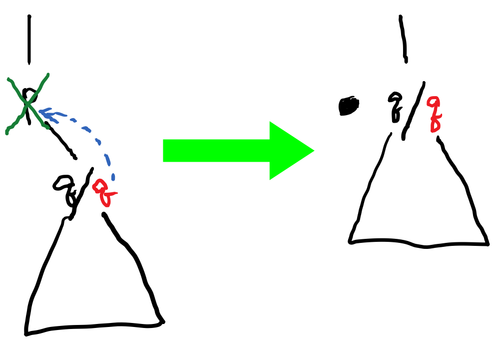


DELETE(p) ➔ Copy p's successor's value to p's node, then DELETE(p's successor node)

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

More Problem Reduction

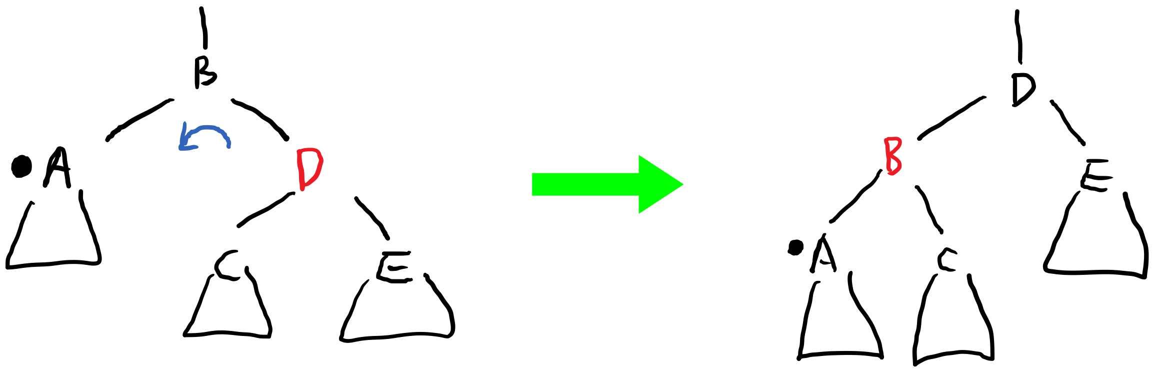
- Deleting a red node is straightforward
  - In fact, if the node to be deleted is red (with at most one child), then it must be a value-bearing leaf (with no value-bearing child). Can be easily removed without violating any red-black properties
- Deleting a black node is complicated
  - Move up its 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

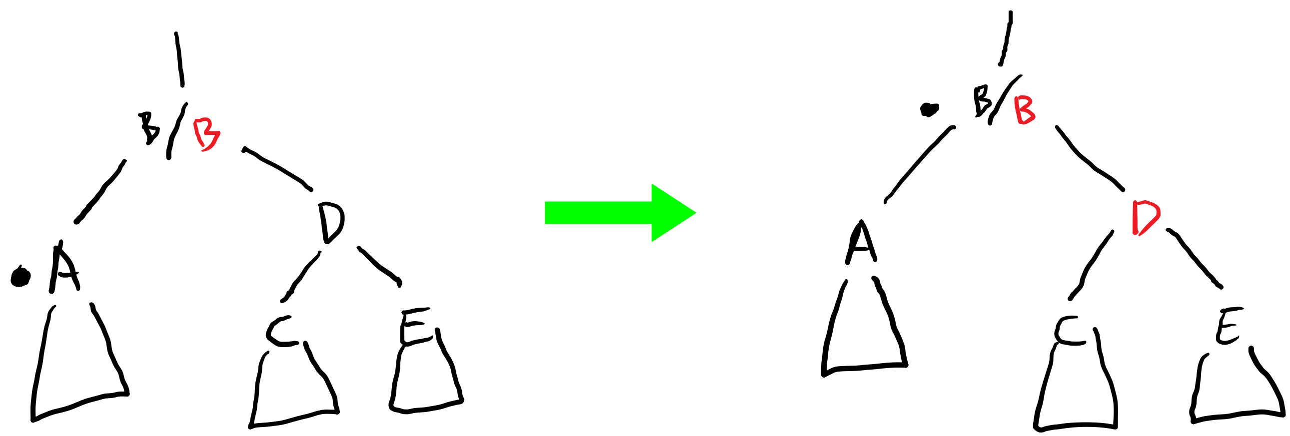
- Case 1: Extra black node's sibling is red
  - The red sibling must have two black children (same-black-heights rule) and its parent must be black too (no-consecutive-reds rule).
  - Rotate the tree along the parent and make the extra black node's sibling black (Transform to Case 2)



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

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

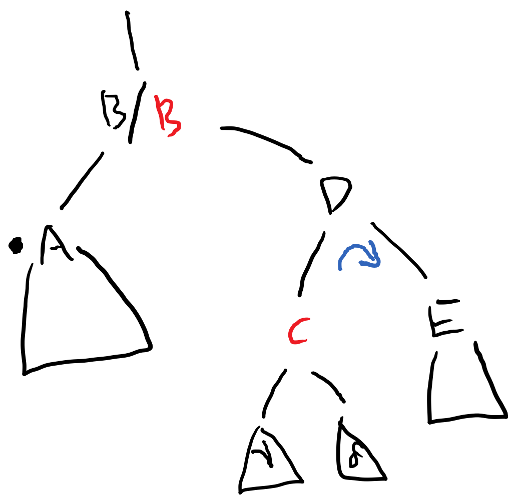
- Case 2: Extra black node’s sibling and its two children are all black
  - We can recolor the sibling to red, and bubble up the extra black to the parent.
  - If the parent was originally red, it can be simply recolored to black, and we are done. Otherwise, continue fixing up the extra black.

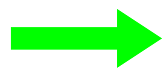


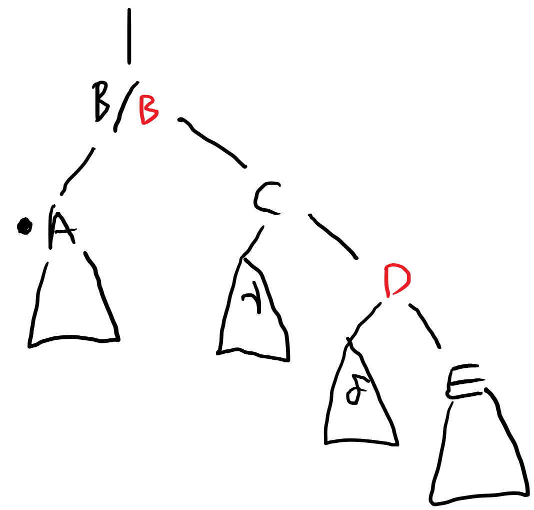
- Black-height requirement is still satisfied
- And the consecutive reds can be easily fixed by recoloring with the extra black
- Or the extra black can be bubbled up/passed along again (iteration)

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.



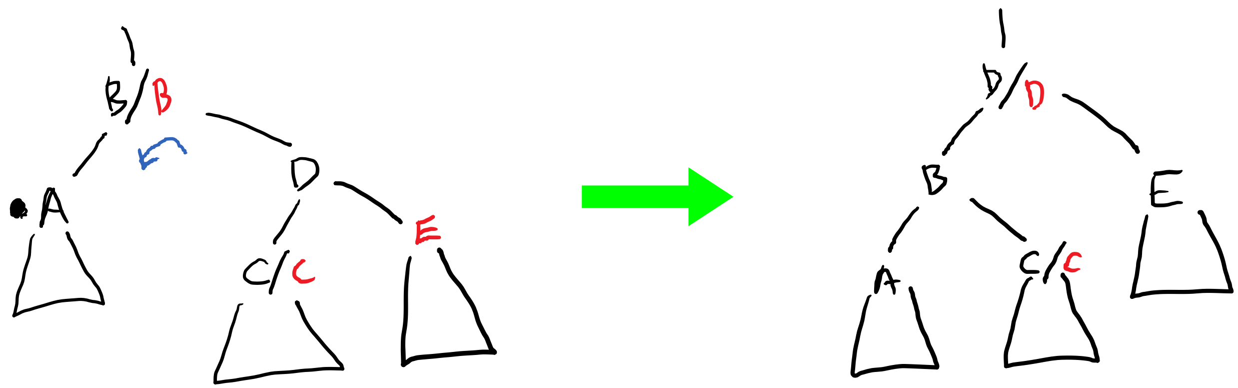




- Recolor C & D so that the black heights are still the same along any path to a leaf.
- This is Case 4.

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!

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

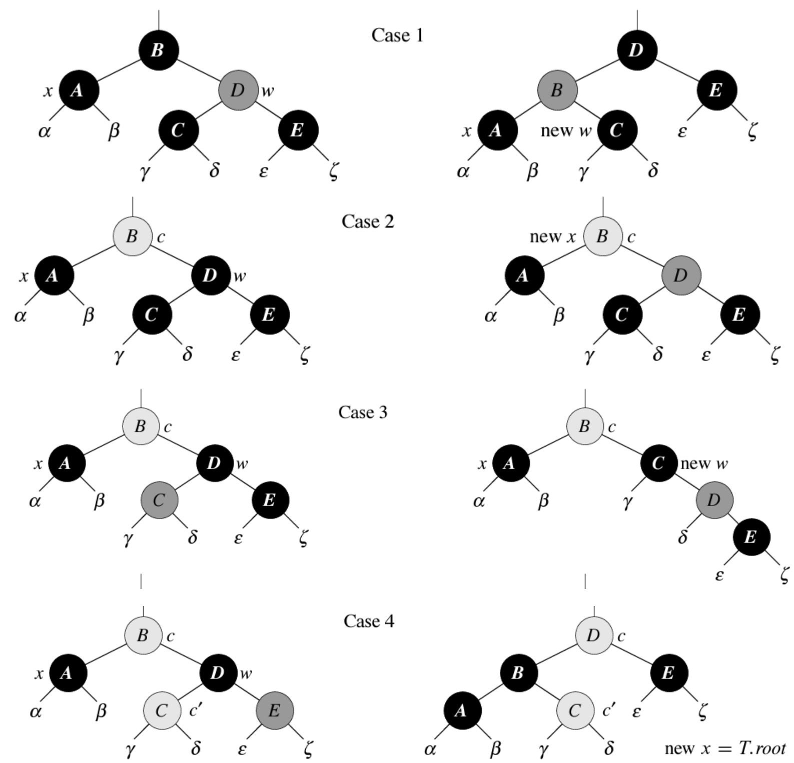
```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y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

```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

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
```



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 $n(h) = n(h\text{-}1) + n(h\text{-}2) + 1$
  - Then $h \leq 1.44\ log_2(n)$
- Weight-balanced BSTs
  - For any node, subtrees should have a bounded size ratio:

$$1 - \beta \leq size(left)/size(right) \leq 1 + \beta$$

  - Then $h \leq c(\beta)\ log_2(n)$