Red-Black Trees-I

ESO207

Indian Institute of Technology, Kanpur

Motivation

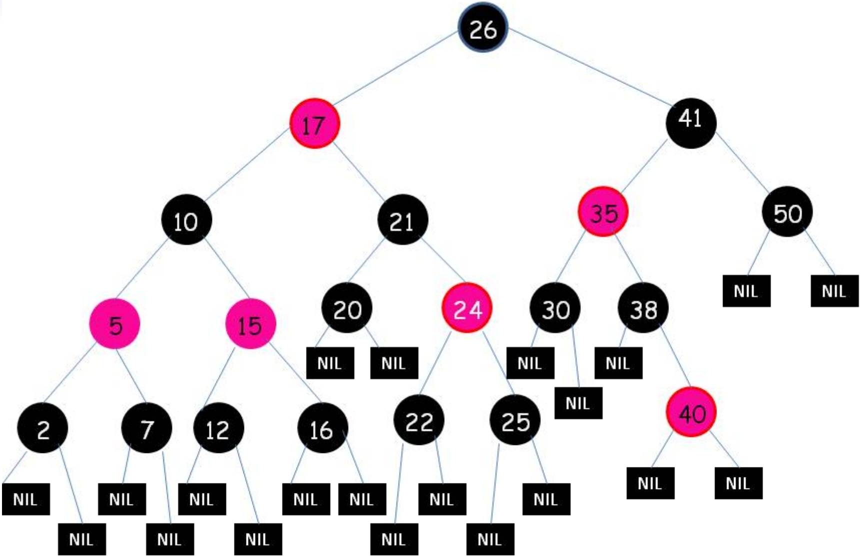
- Binary search trees support SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE.
- Each operation takes time $O(h)$ where h is the height of the tree.
- *Notable deficit*: Binary search trees are not height balanced. So, h is not necessarily $O(\log n)$.

Summary

- *Red-Black* trees *are binary search trees* that satisfy additional properties.
- These properties imply that $h = O(\log n)$.
- Simple enough that INSERT and DELETE operations can still be done in time $O(h) = O(\log n)$.

Definition

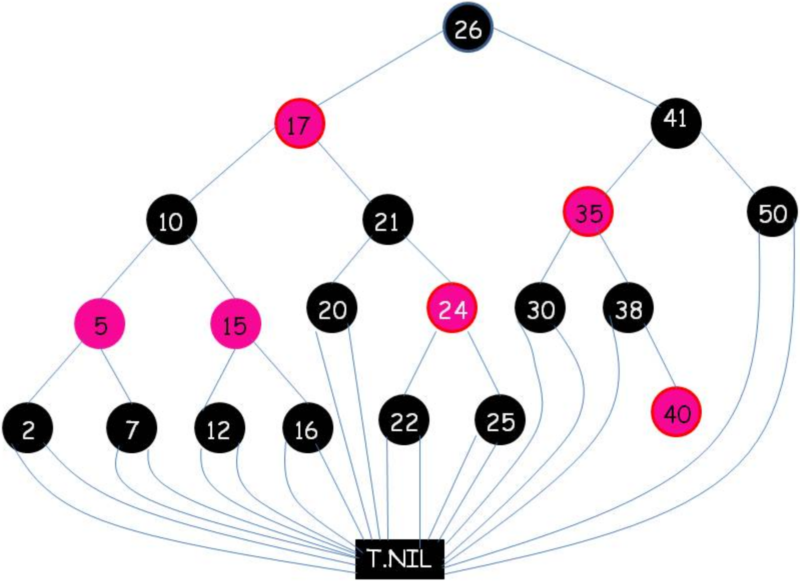
1. A red-black tree *is a **binary search tree*** with
   - one extra bit of storage per node: its color.
   - Every node is colored either RED or BLACK.
2. The root is black.
3. Every leaf (NIL) node is black.
4. If a node is red, then both its children are black.
5. **Black-height property**: for each node, **all** simple paths from the node to descendant leaves (NIL) contain the same number of black nodes.



An example red-black tree

Convention

- Leaf nodes are NIL nodes that are colored black (as shown earlier).
- *A slight modification*: For succcintness, a single node called *T*.NIL is created.
    - Its color is **black**.
    - It has arbitrary values of *left*, *right* and *key*.
    - All pointers to NIL are replaced by pointers to *T*.NIL.
- This reduces storage space for the NIL nodes.



An example red-black tree with sentinel node T.NIL

Black-height of a node

- Recall Definition property 5.
  - **Black-height property**: for each node, **all** simple paths from the node to descendant leaves (NIL) contain the same number of black nodes.
- **Black-height** of a node is the number of black nodes in any simple path from, but not including, a node to a leaf node (NIL or *T*.NIL).
- The notion of *black-height* of a node is well defined by property 5, since,
  - all descending simple paths from the node have the same number of black nodes.

Black-height of node and tree

- Denote black-height of a node x as $bh(x)$.
- The black height of a red-black tree is the black height of its root node.

Near-balanced property of red-black trees-I

Property 1: The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes

.

- Recall: Only NIL(or T.NIL) nodes are leaf nodes, all other nodes are internal nodes.
- Property 1 shows a certain minimum population in the sub-tree rooted at x.
- We will prove by induction on height of a node (not black-height of a node).
- Base case: $bh(x) = 0$. This means x is a NIL(or T.NIL) node. Then,

  $$2^{bh(x)} - 1 = 2^0 - 1 = 0$$

  which is correct since x is a NIL node (no internal nodes in subtree rooted at x.)

Property 1 (proof: induction case)

- Induction case. Suppose x has positive height and is an internal node.
- Without loss of generality, let x have two children.
- The black height of a child is either $bh(x)$ (if the child is red) or $bh(x) - 1$ (if child is black).
- Height of each child of $x \leq$ height of x (trivial!).

Property 1 (proof: induction case)

- So by induction hypothesis applied to each child node of x
  the sub-tree rooted at each child has at least $2^{bh(x)-1} - 1$ internal nodes.
- The sub-tree rooted at x contains at least

$$\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1$$

  internal nodes.
- This proves property 1: Number of internal nodes in the subtree rooted at x is at least $2^{bh(x)} - 1$.

Property 2

- Definition property 4: Red nodes have both children black.
- Half the nodes on any simple path from root to leaf (NIL) is black.
- Let h be the height of the tree (i.e., of root node).
- Hence, $h \leq 2bh(x)$ or, $bh(x) \geq h/2$.
- Hence, number of nodes

$$n \geq 2^{bh(\text{root})} - 1 \geq 2^{h/2} - 1$$

or,

$$h \leq 2\log_2(n+1)$$

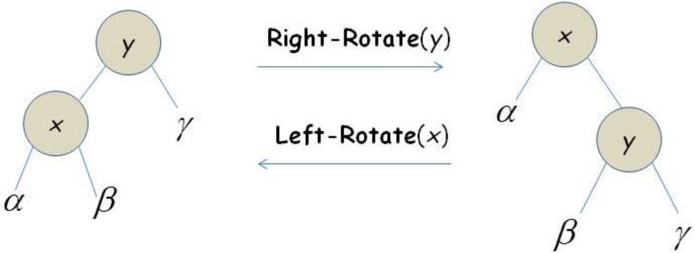
Property 2

Property 2: A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

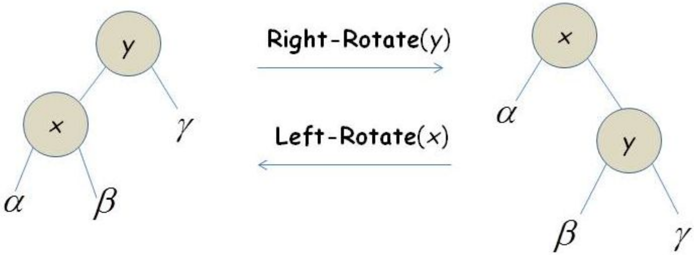
- We have just proved Property 2.
- This gives a “near-balanced” property of red-black trees.
- Since, red-black trees are binary search trees, hence, finding MAXIMUM, MINIMUM, PREDECESSOR and SUCCESSOR can be done in time $O(h) = O(\log n)$.
- We will have to design **new algorithms** for INSERT and DELETE.

Rotations: Towards Insertions and Deletions

- We will move towards designing algorithms for INSERT and DELETE operations.
- Central to this is the concept of **rotations**.
- Two kinds of rotations: Left rotation $\rightleftarrows$ Right rotation.

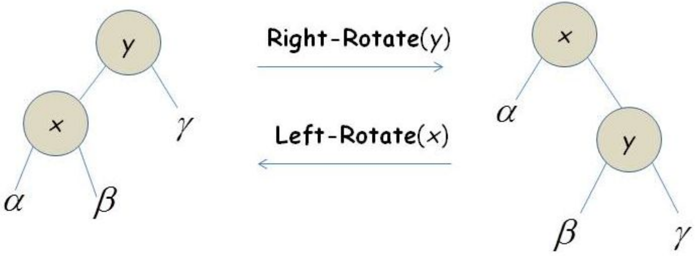


Rotations



- Figure shows right rotation of the subtree on the left about the node *y* and left rotation of the subtree on the right about the node *x*.
- α, β and γ are sub-trees (that could possibly be NIL).
- *Rotations preserve the binary search tree property.*

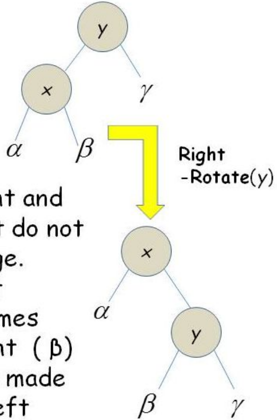
Rotations



- Rotations, by themselves, do not alter the color of any node.
- After rotation, a red node may have a red child (violation!).
- Rotations may or may not preserve black height property (another possible violation!)
- But, binary search tree property is preserved!

Pseudo-code

```
RIGHT-ROTATE(T, y)
// Right rotate subtree rooted at y.
x = y.left
y.left = x.right
if x.right ≠ NIL
        x.right.p = y
x.right = y
pp = y.p
if pp ≠ NIL
        if y == pp.left
                x = pp.left
        else
                x = pp.right
else
        T.root = x
x.p = pp
y.p = x
```



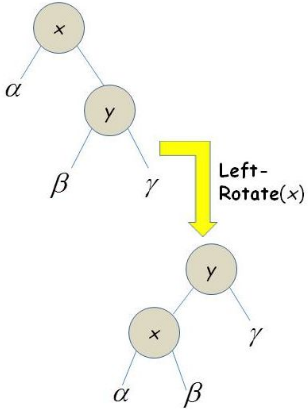
1. y.right and x.left do not change.
2. y.left becomes x.right (β)
3. x.p is made the left (resp. right) child of y.p as y.

Rotation

- Code for *Left-Rotate*(T, x) is symmetric.
- Both procedures run in time $O(1)$.

Left rotation: Psuedo-code

```
LEFT-ROTATE(T, x)
// Left rotate subtree rooted at x.
    y = x.right
    x.right = y.left
    if y.left ≠ NIL
        x.right.p = x
    y.left = x
    pp = x.p
    if pp ≠ NIL
        if x == pp.left
            y = pp.left
        else
            y = pp.right
    else
        T.root = y
    y.p = pp
    x.p = y
```



Red-black tree: Insertion

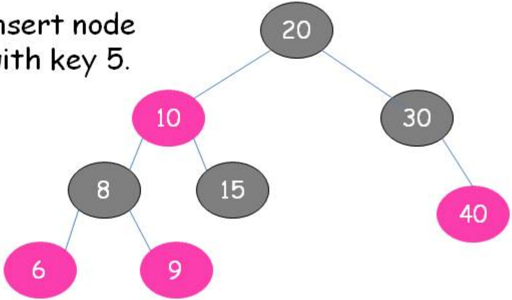
- INSERT(T, z).
- Two steps: one pass down the tree and another pass (possibly) back up the tree.
- In the first step, the new node, say z, is inserted as if the tree was simply a binary search tree and disregarding the coloring properties.
- The newly inserted node is colored RED.
- May be possible violations of the coloring properties.
- A new routine RB-INSERT-FIXUP is called to fix the problem and remove the violations.

Outline of insertion

RB-INSERT(T, z)
1. BSTINSERT(T, z)
2. $z.color =$ RED
3. RB-INSERT-FIXUP(T, z)

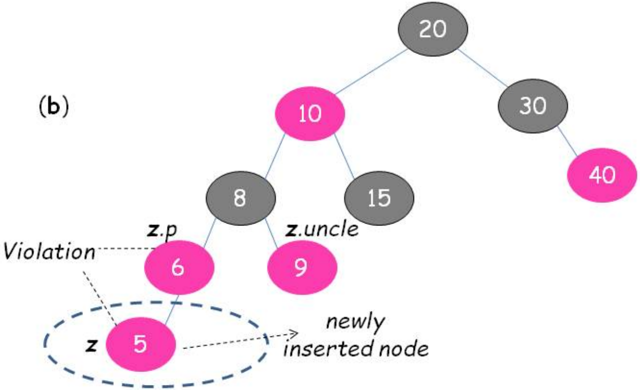
Example

Insert node with key 5.



Insert as if the tree was a binary search tree, and insert the node as a leaf with red color.

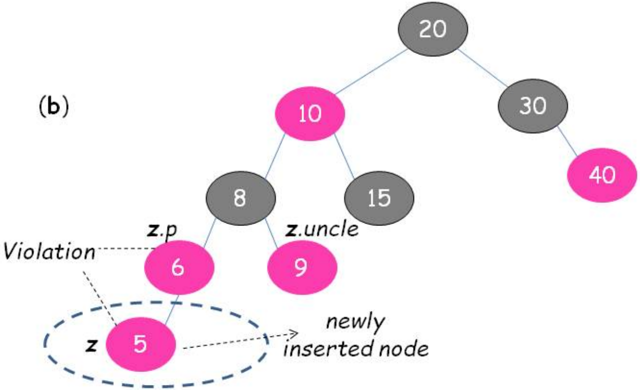
Example contd.



After insertion as in a binary search tree

- The first step is to insert the node z as if the red-black tree is a binary search tree.
- That is, insert into the position just "as z falls off the tree".
- Color z as red.
- *The black-height of all nodes remain unchanged.*
- If $z.p$ (z's parent) is black, insertion is completed.
- If $z.p$ is red, then both z and its parent are red: this is a violation. So RB-INSERT-FIXUP(T, z).

Example contd.



- *Case 1 applies: **z** is red, **z**'s parent is red and **z**'s uncle is red.*
- *Make **z**'s parent and **z**'s uncle black and make **z**'s grandparent red. Set **z** to **z**'s grandparent.*

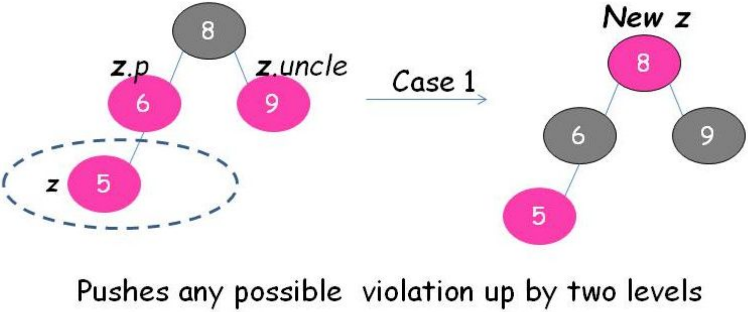
Case 1

RB-Insertion: Case 1 out of three cases.

- z is colored red, $z.p$ is colored red — Basic violation, and,
- $z.uncle$ is colored red.
- Therefore, $z.p.p$ (grandparent of z) exists.
- Why? Because, $z.p$ is colored red and cannot be the root of T. Root is black.
- $z.p.p$ is colored black.
- Why? It has two red children $z.p$ and $z.uncle$ and so $z.p.p$ cannot be red (red nodes must have only black children). So it is black.

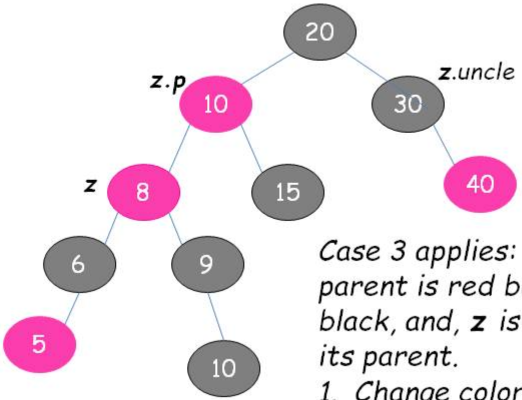
Case 1

- z is colored red, $z.p$ is colored red and $z.uncle$ is colored red. $z.p.p$ is black.
- Change color of $z.p$ and $z.uncle$ to black. Change color of $z.p.p$ to red.



Case 3

(c)



*Case 3 applies: **z** is red, **z**'s parent is red but **z**'s uncle is black, and, **z** is the left child of its parent.*

1. *Change color of **z**.p to black.*
2. *Change color of **z**'s grandparent to red.*
3. *Rotate **right** about **z**.p.p.*

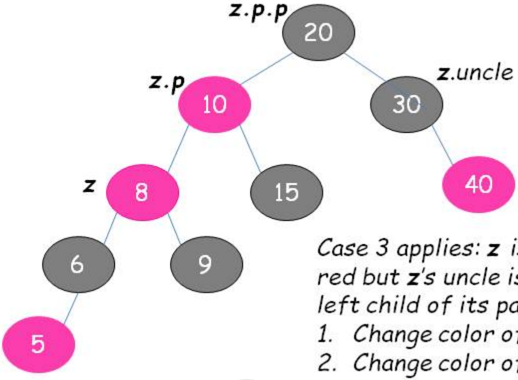
Case 3: definition

- z is colored red, and,
- $z.p$ is colored red, and,
- $z.uncle$ is colored black, and,
- z is left child of its parent.

Case 3: Handling

- Change color of $z.p$ to black.
- Change color of z's grandparent to red.
- Rotate right about $z.p.p$.

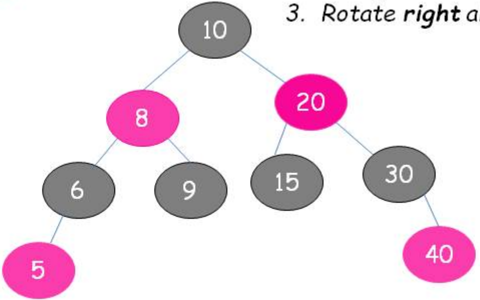
(c)



Case 3 applies: ***z*** *is red,* ***z****'s parent is red but* ***z****'s uncle is black, and,* ***z*** *is the left child of its parent.*

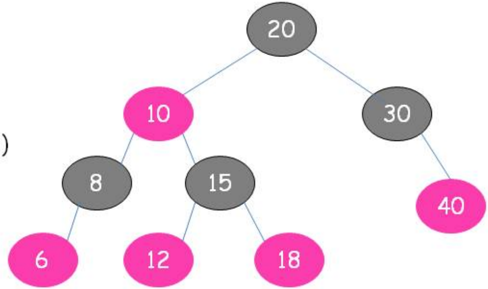
1. *Change color of* ***z****.p to black.*
2. *Change color of* ***z****.p.p to red.*
3. *Rotate* ***right*** *about* ***z****.p.p.*

(d)



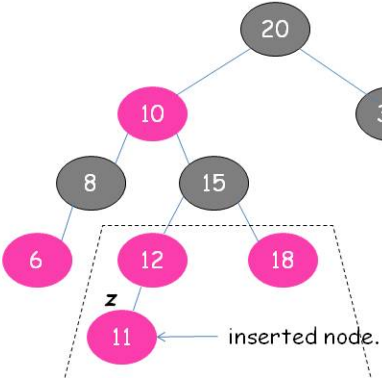
This is a proper red-black tree with no violations.

(a)



Insert node with key 11.

(b)

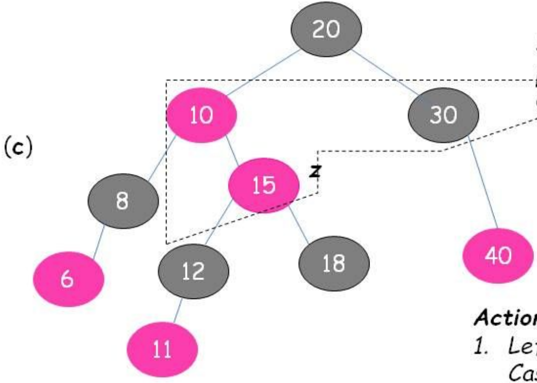


Case 1 applies: *z's uncle is red.*

1. ***Make z****'s parent and* ***z****'s uncle black.*
2. ***Make z****'s grandparent red.*
3. ***Set z*** *to be* ***z****'s grandparent.*

Violation: ***z*** *and* ***z****'s parent are both colored red.*

Case 2



State of tree after Case 1 is processed.
Current focus shown in dots.

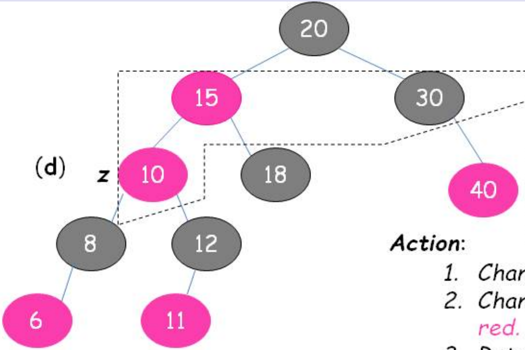
***Case 2 applies** because:*

1. ***z** and **z**'s parent are red.*
2. ***z**'s uncle is black.*
3. ***z** is the right child of its parent.*

Action:

1. *Left-rotate about **z** to get to Case 3.*
2. ***Set z** to its current parent.*

(d)



State of tree after Case 2 is processed.

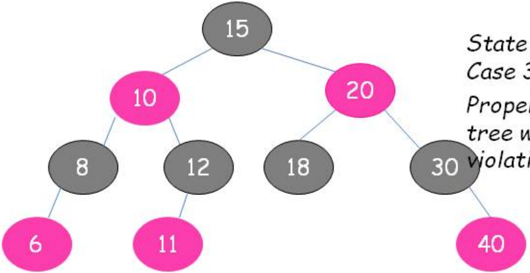
Case 3 applies *because:*

1. ***z*** *and* ***z****'s parent are red.*
2. ***z****'s uncle is black.*
3. ***z*** *is the left child of its pa*

Action:

1. *Change color of* ***z****.p to black.*
2. *Change color of* ***z****'s grandparent to* *red.*
3. *Rotate* ***right*** *about* ***z****.p.*

(e)



State of tree after Case 3 is processed.

Proper red-black tree with no violations.

Insert Fixup routine

- Repeatedly, check which case the violation falls into, and apply the corresponding fixup.

```
RB-INSERT-FIXUP(T, z)
caseno = RB-INSERT-FIXUP-CHECK-CASE(T, z)
while caseno > 0
    if caseno == 1
        RB-INSERT-FIXUP-APPLY-CASE-1(T, z)
    elseif caseno == 2
        RB-INSERT-FIXUP-APPLY-CASE-2(T, z)
    elseif caseno == 3
        RB-INSERT-FIXUP-APPLY-CASE-3(T, z)
    caseno = RB-INSERT-FIXUP-CHECK-CASE(T, z)
T.root.color = BLACK
```

Some observations

- Fixup of Case 2 leads to Case 3.
- Fixup of Case 3 gives a red-black tree with no violations: hence insertion terminates.
- Fixup of Case 1 pushes the violation up the tree by two levels. This can be iterative.
- Fixups of all cases are done in $O(1)$ time.
- Number of times Case 1 repeatedly occurs is $O(h) = O(\log n)$. Hence, time taken is $O(\log n)$.

Auxiliary procedure

```
UNCLE(z)
    if z == NIL  or  z.p == NIL  or  z.p.p == NIL
        return NIL
    elseif z.p == z.p.p.left
        return z.p.p.right
    else
        return z.p.p.left
```

Find which case is violated

```
RB-INSERT-FIXUP-CHECK-CASE(T, z)
    if z.color == BLACK or z.p == NIL or z.p.color == BLACK
        return 0
    elseif UNCLE(z).color == RED
        return 1
    elseif z == z.p.right
        return 2
    else
        return 3
```

Pseudo-code: RB-INSERT-FIXUP

```
RB-INSERT-FIXUP(T, z)
    caseno = RB-INSERT-FIXUP-CHECK-CASE(T, z)
    while caseno > 0
        if caseno == 1
            z.p.color = BLACK                 // Case 1
            UNCLE(z).color = BLACK            // Case 1
            z.p.p.color = RED                 // Case 1
            z = z.p.p                         // Case 1
        elseif caseno == 2
            LEFT-ROTATE(T, z)                 // Case 2
            z = z.left                        // Case 2
        elseif caseno == 3
            z.p = BLACK                       // Case 3
            z.p.p = RED                       // Case 3
            RIGHT-ROTATE(T, z.p)              // Case 3
T.root.color = BLACK
```

Invariant

Let z denote the node that is inserted as per insertion into binary search tree. z is colored red. Invariant:

a. z is colored red.

b If $z.p$ is the root, then $z.p$ is black.

c If the tree violates any of the red-black properties then it violates at most one, namely, either

  c.1 Property 2: The root is black, or,

  c.2 Property 4: If a node is red, then both its children are black.

Initialization of the RB-INSERT-FIXUP-PROCEDURE

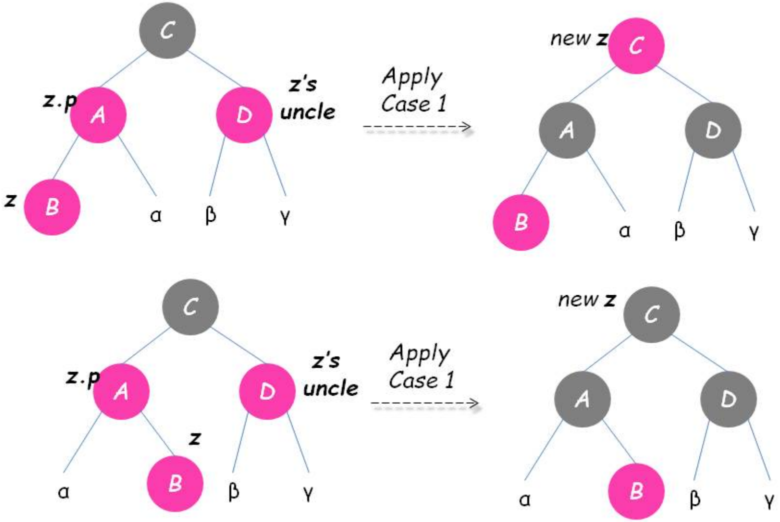
- When this procedure is initialized, z is the inserted node and is colored red.
- If $z.p$ is the root, then $z.p$ started out black and did not change prior to call.
- Important: black-heights of all old nodes remain unchanged, because z is inserted as a red node.
- If root is red, then, the root must be z. (This case is easy: change z's color to black and terminate.)
- Otherwise, the only possible violation is that $z.p$ and z are both red. No other red-black properties are violated anywhere else in the tree.

Invariant: Maintenance

- Consider the more complicated case when z and $z.p$ are colored red, and therefore, $z.p.p$ exists and is colored black.
- There are two symmetric cases:
  1. $z.p$ is the left child of $z.p.p$ (considered here), and,
  2. $z.p$ is the right child of $z.p.p$ (symmetric, not considered).

Invariant: Maintenance

- ***Case 1*:** z's uncle y is red.
- So z , $z.p$ and $z.uncle$ are all red.
- Action:
  1. Change color of $z.p$ and $z.uncle$ to black.
  2. Change color of $z.p.p$ to red.
  3. Set z to $z.p.p$. Make the grandparent as the new z.



Case 1: z is red, z's parent is red and z's uncle is red. The action taken is to make the parent and the uncle of z to be black and make the grandfather of z to be red. This may ``**push the violation up the tree**." There is a symmetric case when z.p is the right child of its parent.

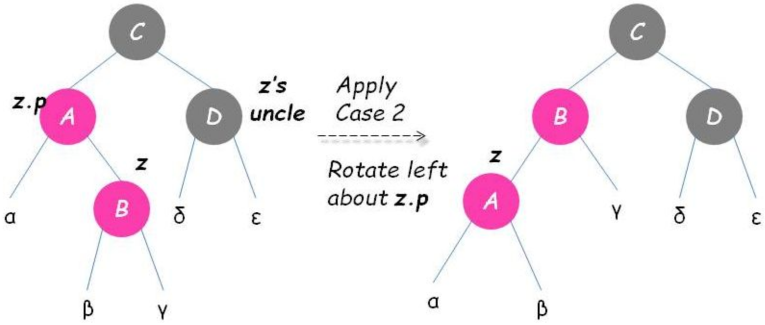
Invariant: Maintenance

- **Note:** Black height of every node remains the same before and after the transformation corresponding to Case 1.
- There can be at most one violation, namely that the new z (which is the old grandparent) is red and *its* parent may also be colored red.
- In this case, the violation moves up the tree by two levels.
- There are no other possible violations in the red-black tree.
- If the new z's parent is colored black, then the process terminates.

Invariant: Maintenance

- **Case 2**: z' uncle y is black and z is the right child of its parent.
- **Case 3**: z's uncle y is black and z is the left child of its parent.

Case 2



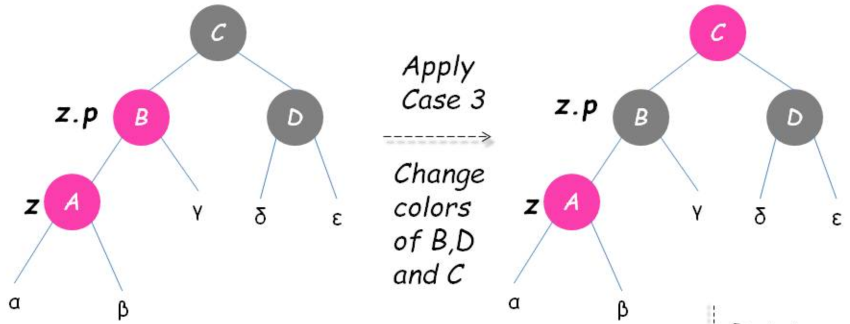
Case 2: There is a symmetric case for Case 2 when z's parent is the right child of its parent.

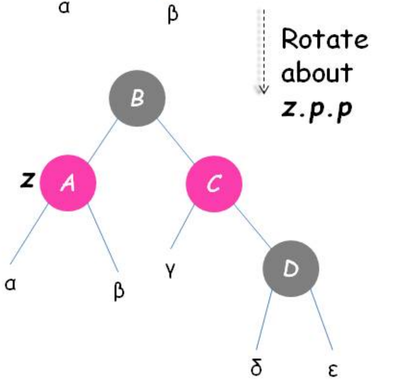
Case 2

- Action: Rotate left around $z.p$. z becomes the parent and $z.p$ becomes the left child of z.
- Rename $z.p$ as z.
- We are now in Case 3. (The purpose of the action for case 2 is to transform into an instance of Case 3).

Case 3

- z and $z.p$ are red, z's uncle is black and z is the left child of $z.p$.
- Note that $z.p.p$ is black.
- Actions:
  1. Change the color of $z.p$ and $z.uncle$ to black.
  2. Change color of $z.p.p$ to red.
  3. Rotate right about $z.p.p$.





1. The black heights of A,α,β, γ and D are same.
2. So black height property of A, C, B are preserved.
3. Also, the black height of B is the same as the black height of C in the original tree .

Invariant: Case 3

- Case 3 preserves black height property (needs some verification).
- There are no violations of red-black properties.
- So, the insertion procedure terminates.
- Case 3 is a termination case.

Complexity Analysis

- Each of the cases takes $O(1)$ time to complete.
- Case 1 may be called at most $O(\log n)$ times.
- If Case 2 or Case 3 is called, then the procedure terminates in $O(1)$ time.
- Hence time required is bounded by $O(\log n)$.