Binary Search Tree

Sayantan Maity Indronil Bhattacharjee Abdur Razzak

New Mexico State University

1. Insertion of a node into a binary search tree.

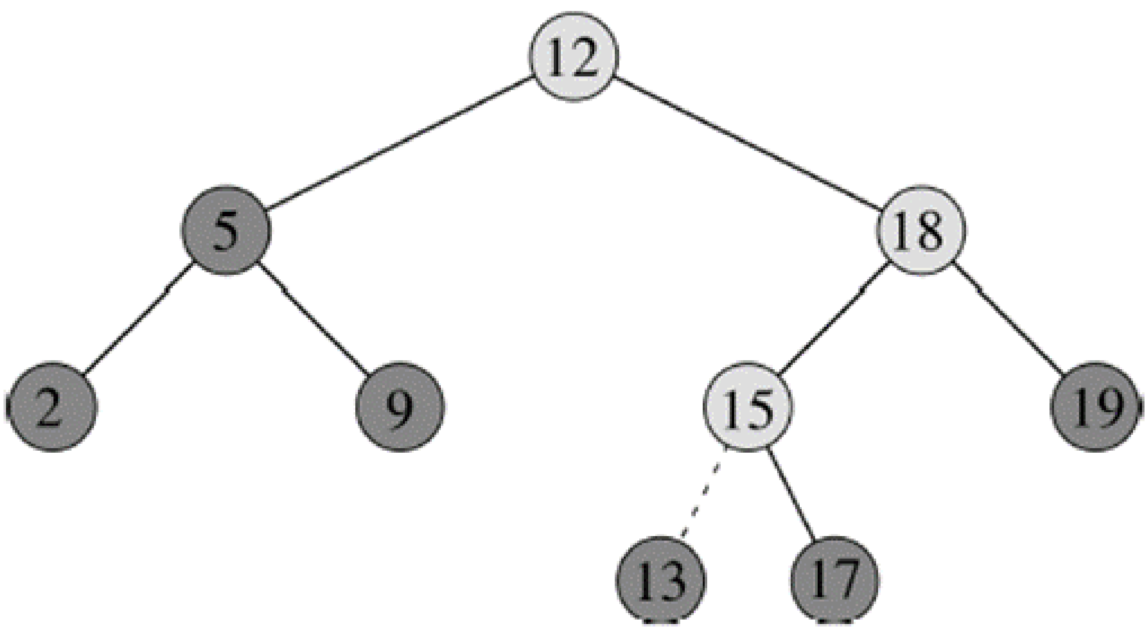


Figure 1: Insertion Model

To insert a new value v into a binary search tree T, we use the procedure `TREEINSERT`. The procedure takes a node z for which z.key $= v$, z.left $=$ NIL, and z.right $=$ NIL.

It modifies T and some of the attributes of z in such a way that it inserts z into an appropriate position in the tree.

Algorithm 1 TREE-INSERT(T, z)

[b]

```
x = T.root
par = NIL
while x ≠ NIL do
    par = x
    if z.key ≤ x.key then
        x = x.left
    else
        x = x.right
    end if
end while
z.p = par
if par == NIL then
    T.root = z // T was empty
else if z.key ≤ par.key then
    par.left = z
else
    par.right = z
end if
```

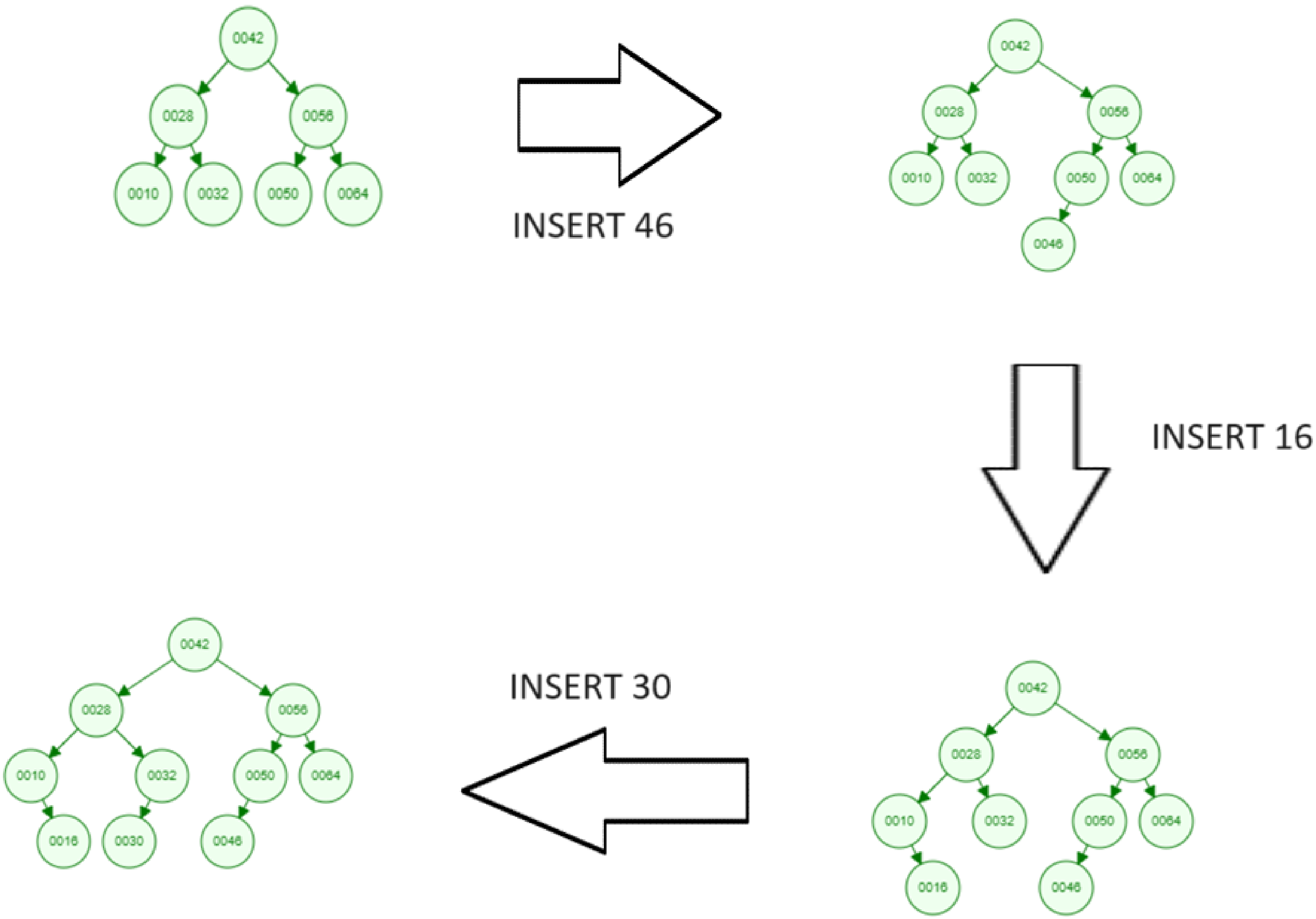


Figure 2: Insertion Live Example

2. Deletion of a node into a binary search tree.

1. If x has no children, then we simply remove it by modifying its parent to replace x with NIL as its child.
2. If x has just one child, then we elevate that child to take x's position in the tree by modifying x's parent to replace x by x's child.
3. If x has two children, then we find x's successor y—which must be in x's right subtree—and have y take x's position in the tree. The rest of x's original right subtree becomes y's new right subtree, and x's left subtree becomes y's new left subtree.

TRANSPLANT(T, u, v)
//replace node u by node v in binary search tree T
//u cannot be NIL; v can be NIL

Algorithm 2 Transplant

```
if u.p == NIL // u is the root
    T.root = v
elseif u == u.p.left // u is the left child of its parent
    u.p.left = v
else // u is the right child of its parent
    u.p.right = v
if v ≠ NIL
    v.p = u.p
```

Algorithm 3 TREE-DELETE(T, z)

```
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else
    y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        z.right.p = y
        TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.p = y
```

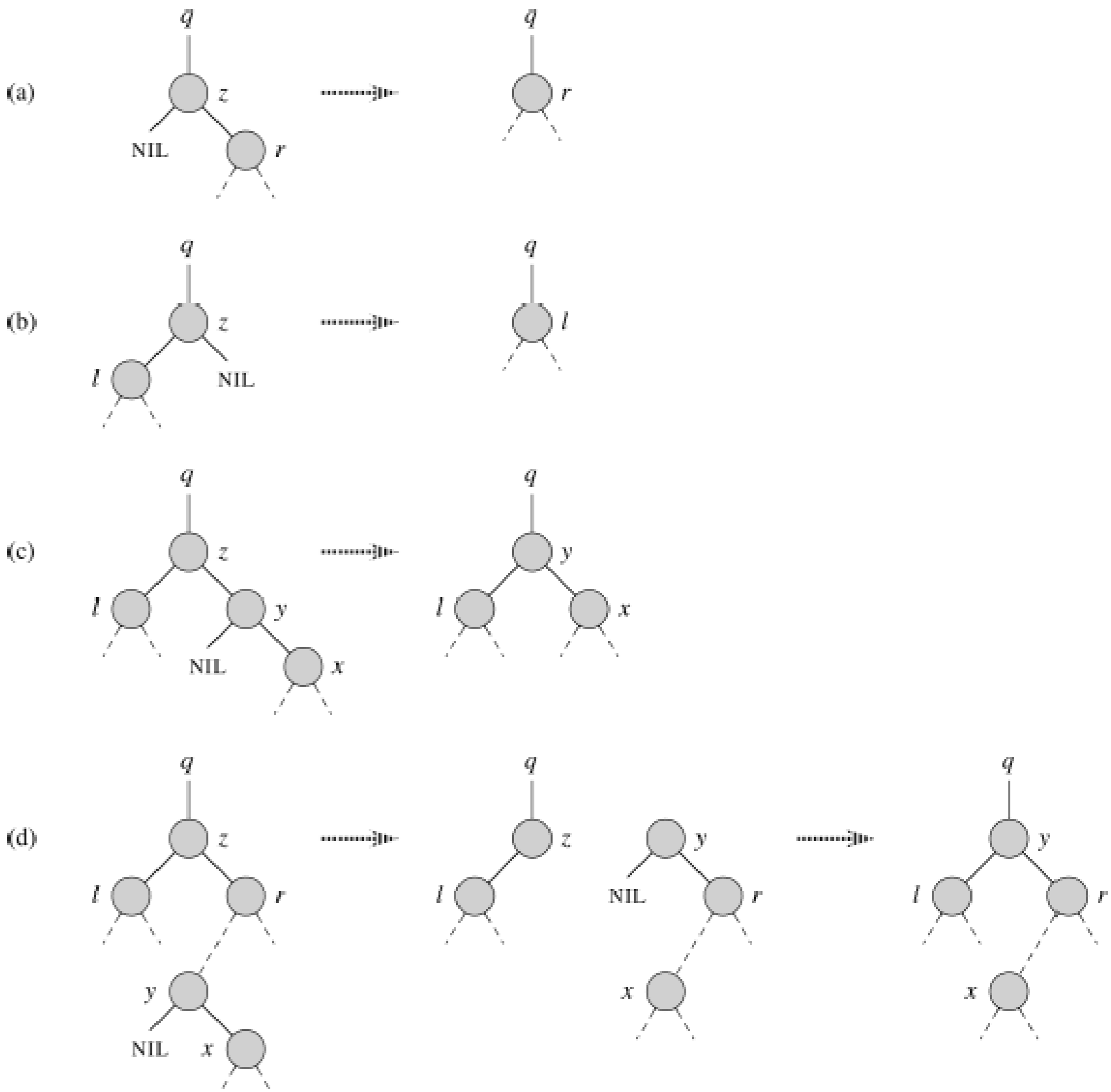


Figure 3: Deletion Model

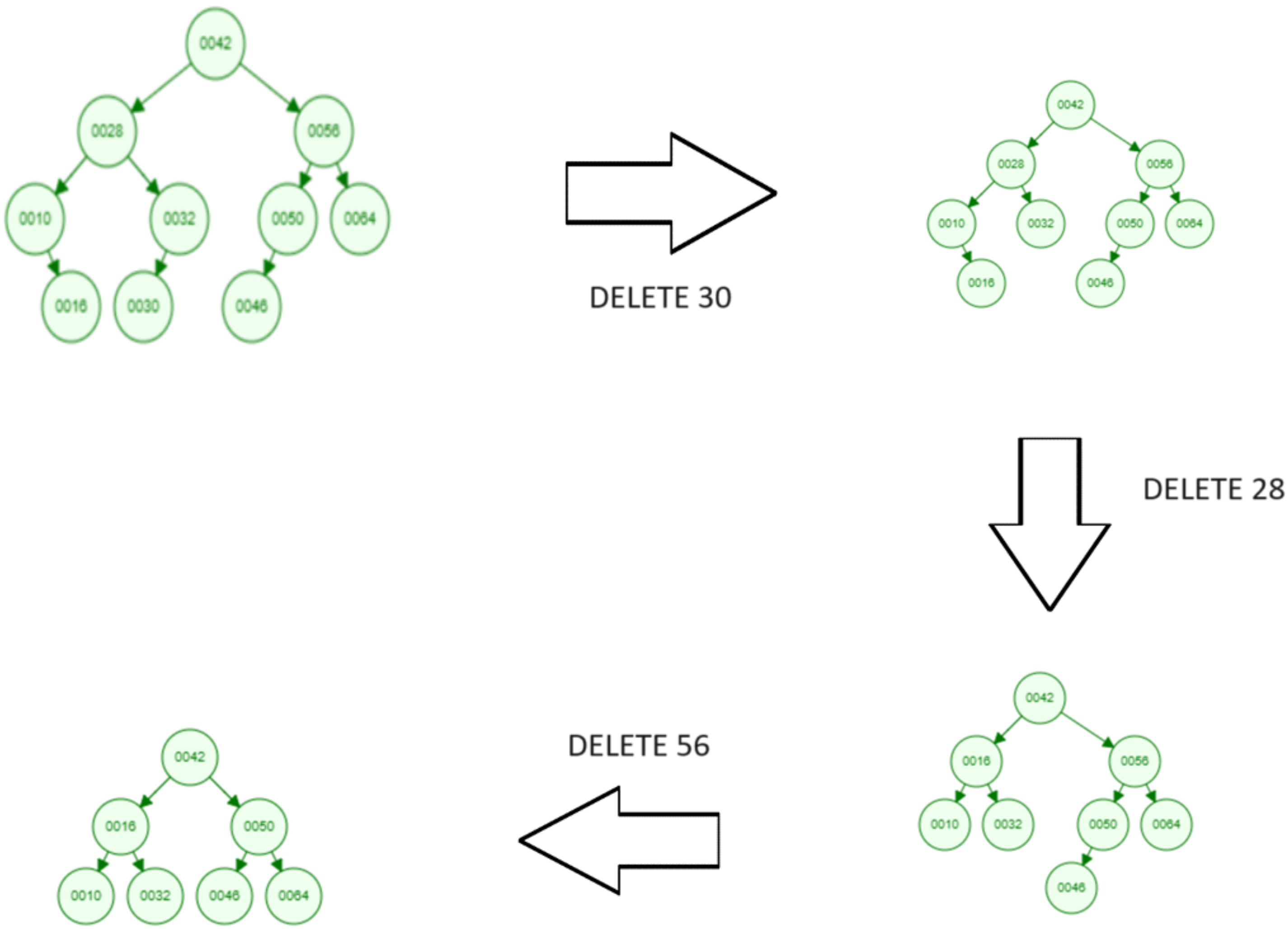


Figure 4: Deletion Live Example

Red-Black Trees

The lecture notes are mostly based on Chapter 14 of Cormen, Leiserson, Rivest, and Stein. Introduction to Algorithms. 3rd Ed. 2009. MIT Press. Cambridge, Massachusetts.

Contents

Motivation:

Red-black trees are one of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take O(lg n) time in the worst case.

1 What is red-black tree

A red-black tree is a binary search tree

- With one extra bit of storage per node: Color
- Color can be either red or black
- Ensures that there is no such path, which is more than twice as long as any other
- Balanced

A node in a red-black tree contains:

- key
- left: pointer to left child.

  left = NIL if no left child.
- right: pointer to right child.

  right = NIL if no right child.
- p: pointer to parent.

  p=NIL for the root node.
- **color: color of the node**

Lemma 1

A red-black tree with n internal nodes has height at most 2 lg(n + 1)

2 Properties of a red-black tree

- Every node is either **RED** or **BLACK**.
- The root is black.
- Every leaf (NIL) is black.
- If a node is red, both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

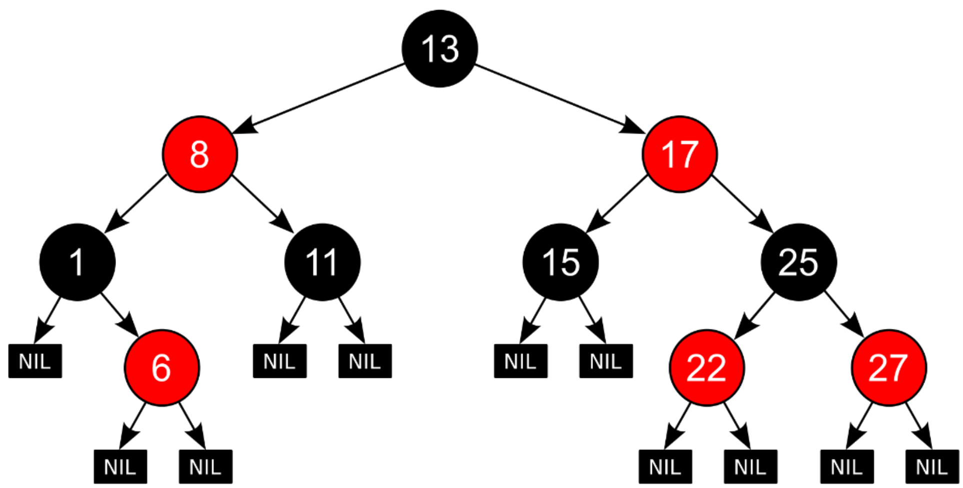


Figure 1: Example of a red-black tree.

3 Rotation of nodes in a red-black tree

```
LEFT-ROTATE(T, x)
y = x.right                  // set y
x.right = y.left             // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                    // link x's parent to y
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                   // put x on y's left
x.p = y
```

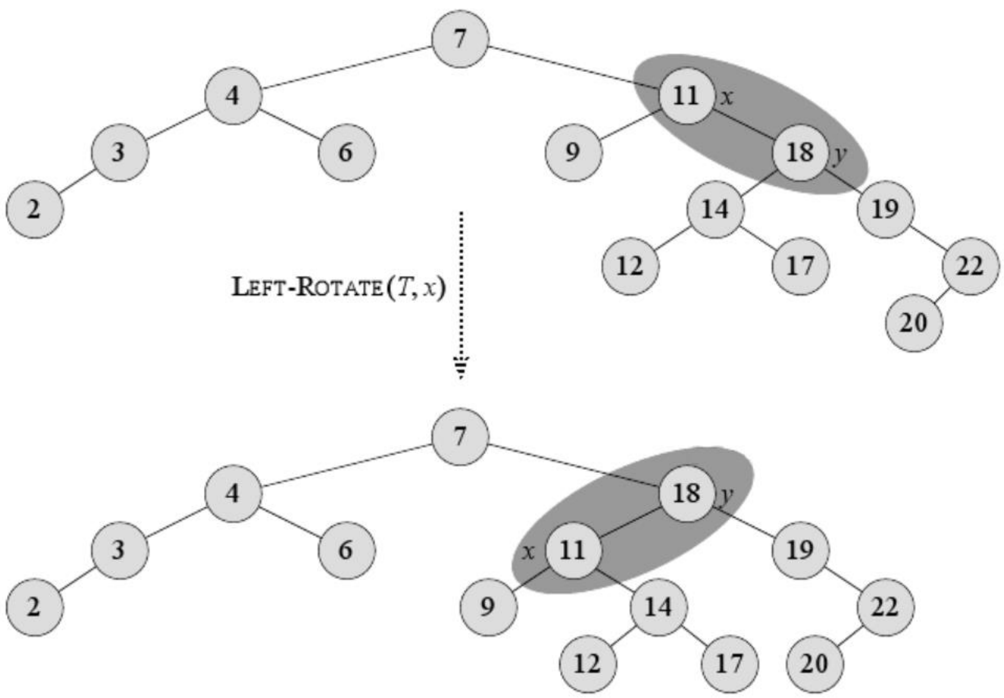


Figure 2: Left rotation in a red-black tree

```
RIGHT-ROTATE(T,x)
y = x.left                      // set y
x.left = y.right                // turn y's right subtree into x's left subtree
if y.right ≠ T.nil
    t.right.p = x
y.p = x.p                       // link x's parent to y
if x.p == T.nil
    T.root = y
else if x == x.p.left
    x.p.left = y
else x.p.right = y
y.right =x                      // put x on y's right
x.p =y
```

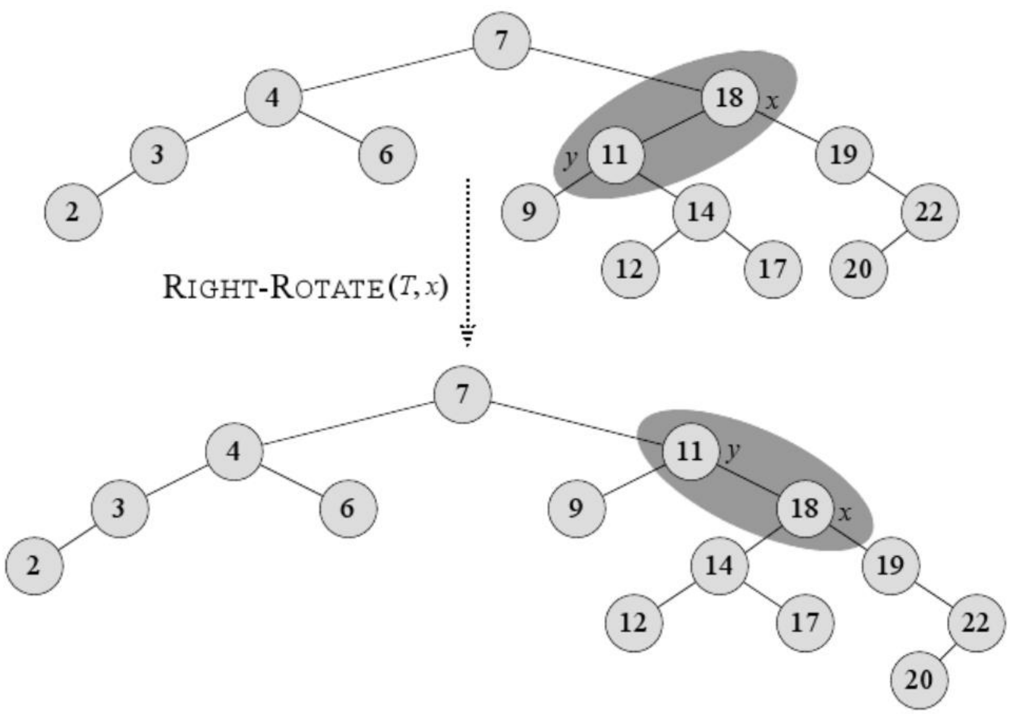


Figure 3: Right rotation in a red-black tree.

4 Insertion of a node into a red-black tree

```
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)
```

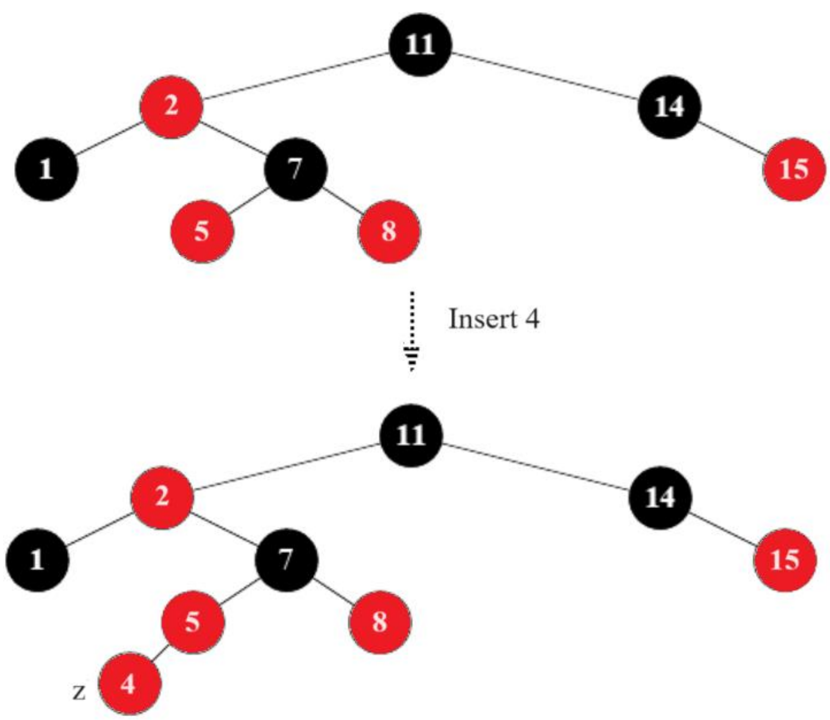


Figure 4: Insertion in a red-black tree before fixup.

- RB-INSERT-FIXUP restores the red-black properties to the tree

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left                     // if z's parent is a left child
        y = z.p.p.right                      // y is z's uncle
        if y.color == RED                    // are z's parent and uncle both red?
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p                        Case 1
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)            Case 2
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)           Case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

- Uncle node: Parent node's sibling (Child from same parent)

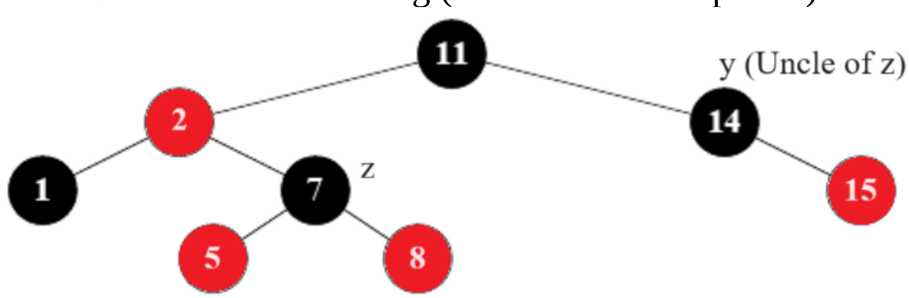


Figure 5: Uncle node.

- The cases to check to initiate insertion-fixup:

Case 1. z's uncle y is red
- Color y and z's parent black,
- Color z's grandparent red.
- Update z = z's grandparent.

Case 2. z’s uncle y is black and z is a right child

- Update z= z’s parent.
- Left rotate z.

Case 3. z’s uncle y is black and z is a left child

- Color z’s parent black
- Color z’s grandparent red.
- Right rotate z’s grandparent.

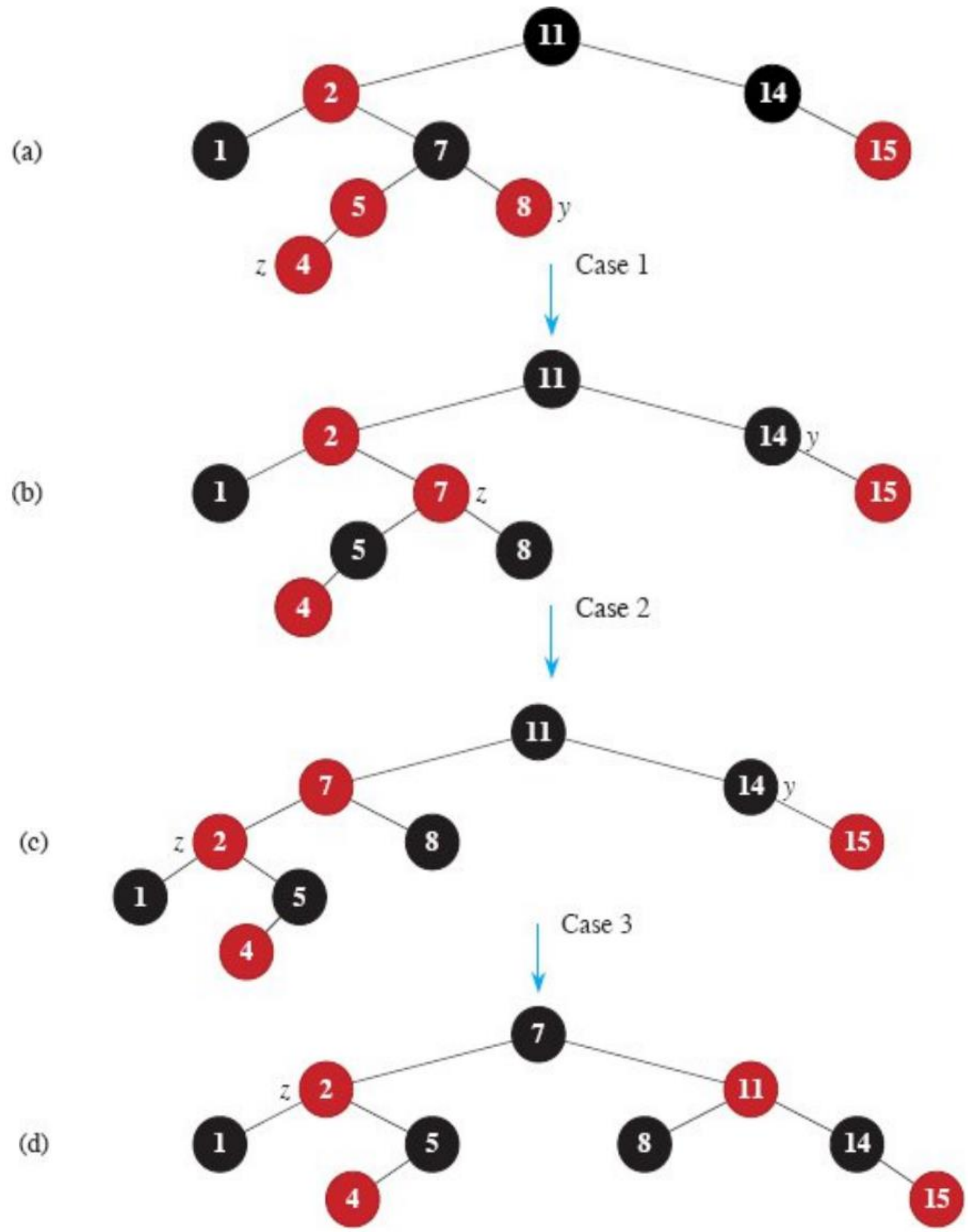


Figure 6: RB-Insertion-Fixup cases.

5 Deletion of a node from a red-black tree

- RB-DELETE deletes a node from the tree
- RB-TRANSPLANT helps to move subtrees within the tree

```
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
```

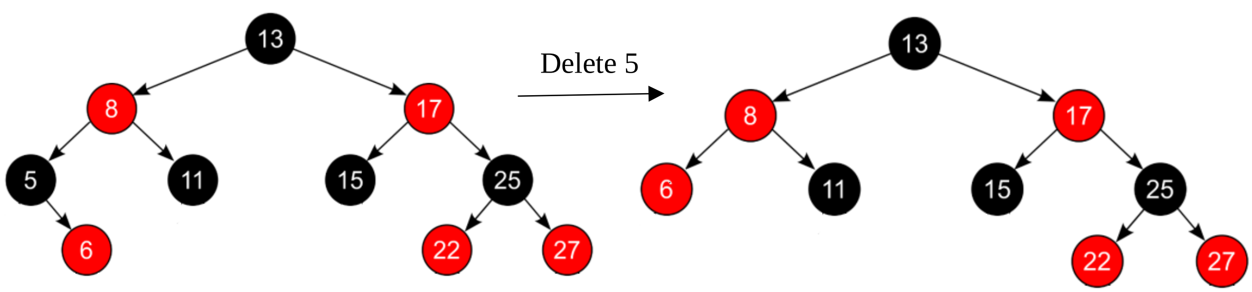


Figure 7: Deletion in a red-black tree.

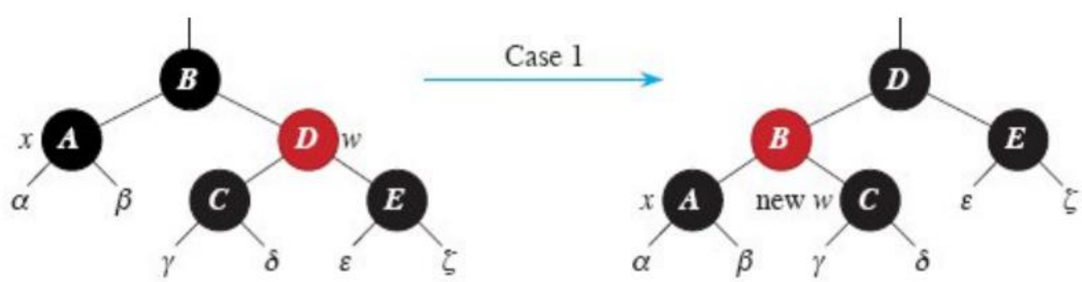
- RB-DELETE-FIXUP restores the red-black properties to the tree

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

- The cases to check to initiate delete-fixup:

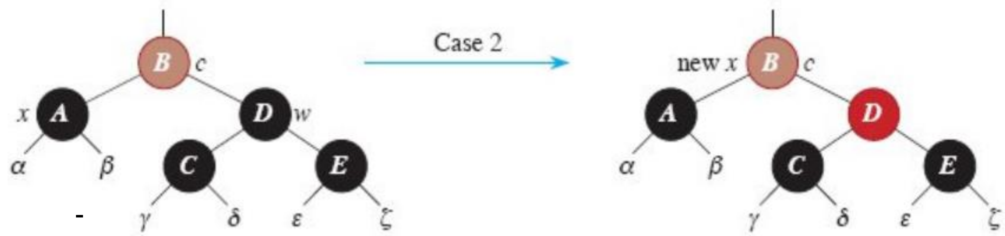
Case 1. x's sibling w is **red**

- Color w black
- Color x's parent red
- Left rotate x's parent
- Update w = right child of x's parent



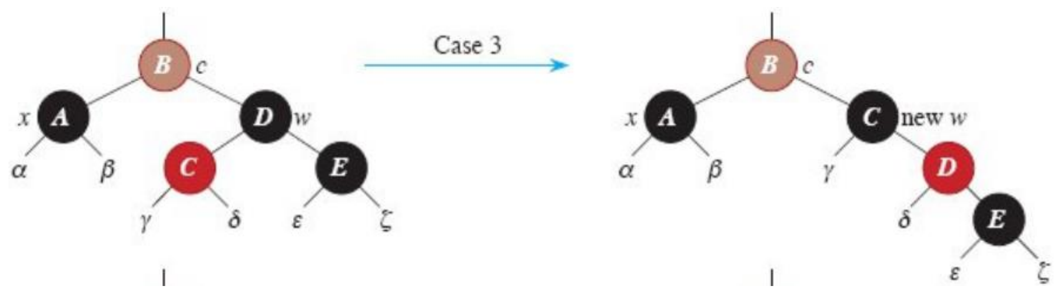
Case 2. x's sibling w is **black**, and both of w's children are **black**

- Color w red
- Update x = parent of x



Case 3. x's sibling w is **black**, w's left child is **red**, and w's right child is **black**

- Color w's left child black
- Color w red
- Right rotate w
- Update w = right child of x's parent



Case 4. x's sibling w is black, and w's right child is **red**

- Color w as x's parent
- Color w's right child black
- Left rotate parent of x
- Update x = root of the tree

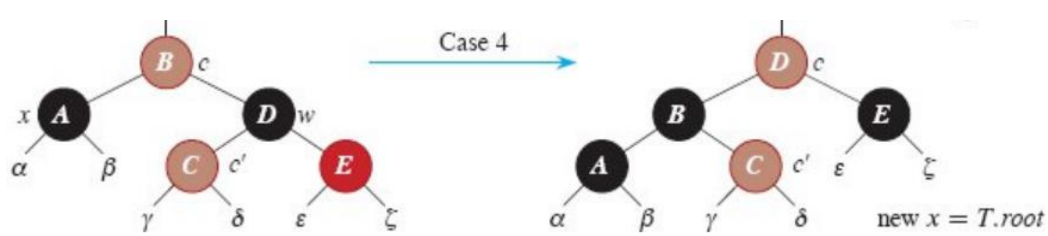


Figure 8: Deletion-fixup cases.

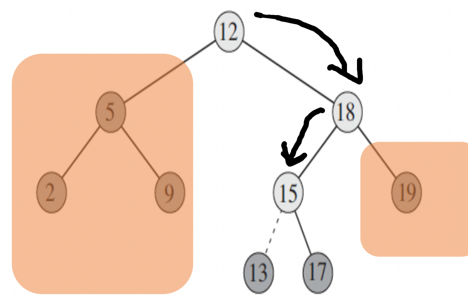
- Finally, color x to black.

Runtime Analysis

Insertion:

```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.root
3   while x ≠ 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 == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

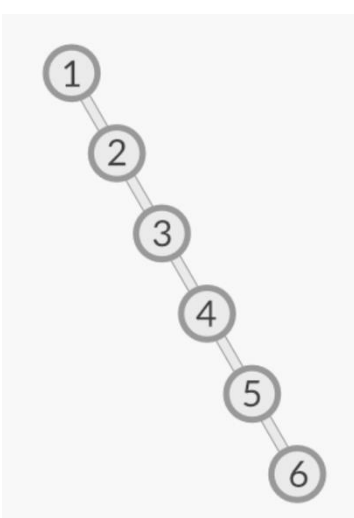
1	1
2	1
3	Log n
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	1
13	1



Complexity: 1+1+ Log n + 1 + 1+ … + 1 ~ O(log n)

Height of the tree is CEIL (Log n)

Problem: UNBALANCED – height of the tree is not always log n.



Inserting 7?

Height of the TREE is N

Insertion complexity: Worse Case O (n)

Height of the TREE is important while inserting.

Height Balanced: Log n

Red-Black Insertion:

```
RB-INSERT(T, z)
1   y = T.nil
2   x = T.root
3   while x ≠ 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)
```

1	1
2	1
3	Log n
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	O(RB-Inser-Fixup)

Complexity: 1+1+ Log n + 1 + 1+ … + 1 + O(RB-INSERT-FIXUP)

~Log n [calculated in next page]

Overall Complexity: O(log n)

Note:

RB-INSERT-FIXUP fixes the height balanced.

Height is remains Log n

Height is important:

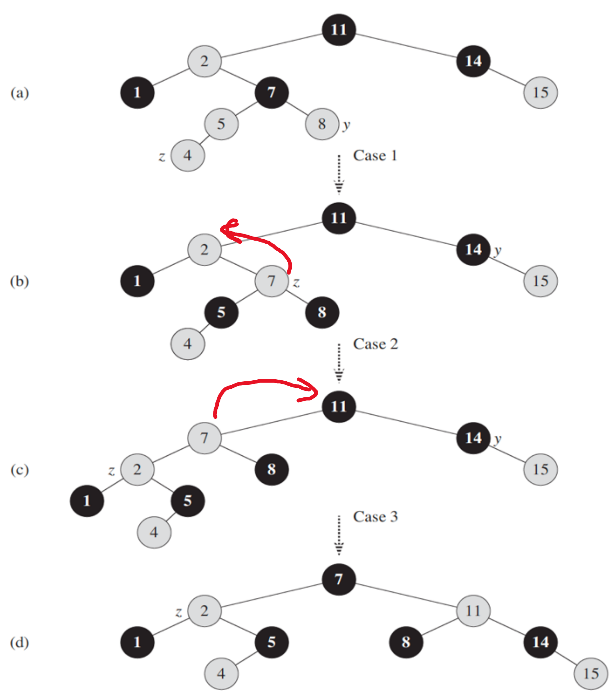
height is n→inserting next element is O(n)

height is log n → Inserting next element O(log n)

```
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.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
```

1	Log(n) at worse
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	* O (LEFT-ROTATE)
12	1
13	1
14	* O (RIGHT-ROTATE)
15	1
16	1

O(1)




```
LEFT-ROTATE(T, x)
1  y = x.right
2  x.right = y.left
3  if y.left ≠ T.nil
4      y.left.p = x
5  y.p = x.p
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right = y
11 y.left = x
12 x.p = y
```

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	1

Complexity of RB-Insert-Fixup: Log n

Insertion Scenarios:

1. BST:
    a. During insertion, each number may face height: (log n) or n.

    b. Tree remains unbalanced after each insertion.

    c. Should be fine for small number of insertions.

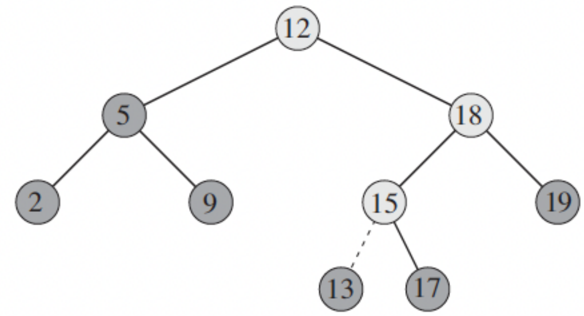
2. RB-BST:
    a. During Insertion, Each number faces height (log n)

    b. Tree always balanced at any point of time.

    c. Good for big number of insertions.

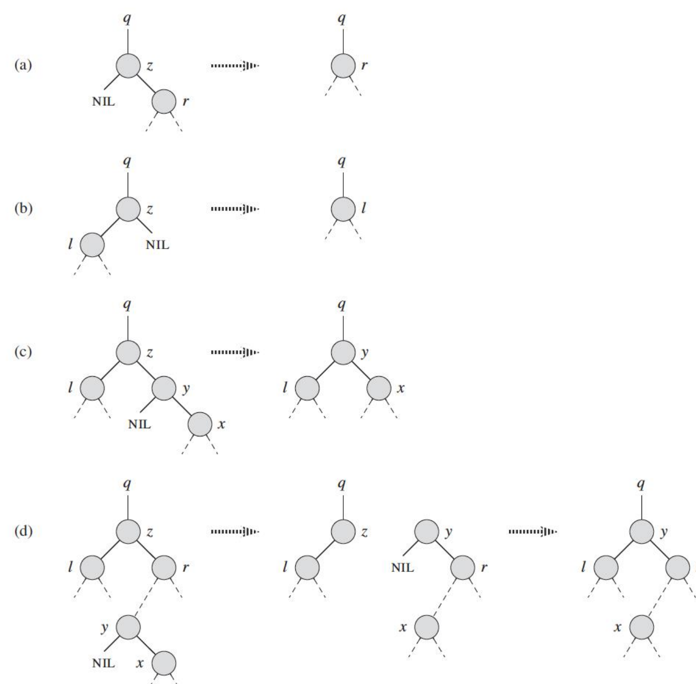
```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

O(h)



O(1)

```
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
```



Complexity: O (h)

```
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)
```

1	1
2	1
3	1
4	1
5	O (RB-Transplant)
6	1
7	1
8	O (RB-Transplant)
9	O(h)
10	1
11	1
12	1
13	1
14	O (RB-Transplant)
15	1
16	1
17	O (RB-Transplant)
18	1
19	1
20	1
21	1
22	1
23	O(RB-Delete-Fixup)

O(1)

```
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
```

Complexity: O(h)

1	Log(n) at worse
2	1
3	1
4	1
5	1
6	1
7	* O (LEFT-ROTATE)
8	1
9	1
10	1
11	1
12	1
13	1
14	1
15	*O(RIGHT-ROTATE)
16	1
17	1
18	1
19	1
20	*O(LEFT-ROTATE)
21	1
22	1
23	1

Overall complexity of Delete:

O(h) + O(h) at worse from fixup

O(h)

Advantage of RB-DETETE:

Tree remains balanced: Height O (log n)

Many operations would be efficient in this process.

Summary:

1. Discussed Binary Search Tree Insertion and Deletion.
2. Learned about Red Black BST Insertion and Deletion process.
3. Complexity analysis and efficiency.