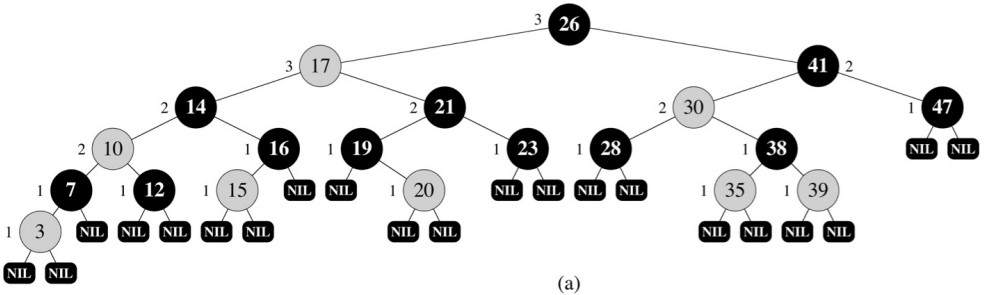
Red-Black Trees

- Properties
  - The normal binary search tree properties still hold
  - Every node is either red or black
  - Every leaf is black (the NIL child pointers)
  - If a node is red, then both its children are black
  - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

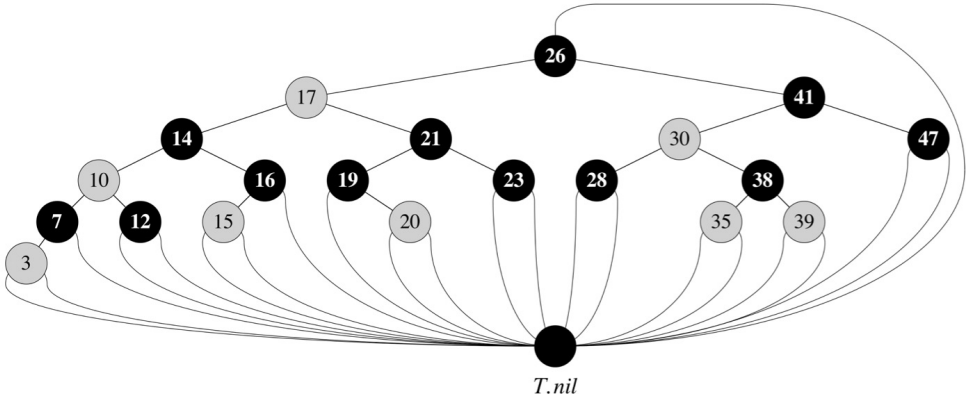
- Since NIL leaves are considered black, a special sentinel value T.nil is used in place of NIL in the algorithms, giving leaves a black color attribute

- No changes need to be made to BST query operations, except we look for T.nil instead of NIL
- Red-black trees keep the tree approximately balanced when inserting and deleting

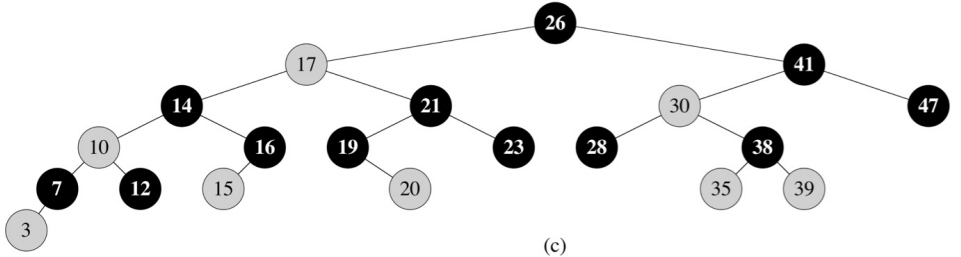
h is at most $2 \lg(n+1)$ by lemma 13.1


3 26
3 17
41 2
2 14
2 21
2 30
1 47
2 10
1 16
1 19
1 23
1 28
1 38
NIL NIL
1 7
1 12
1 15
NIL
NIL
1 20
NIL NIL
NIL NIL
1 35
1 39
1 3
NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL

(a)


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
T.nil

(b)

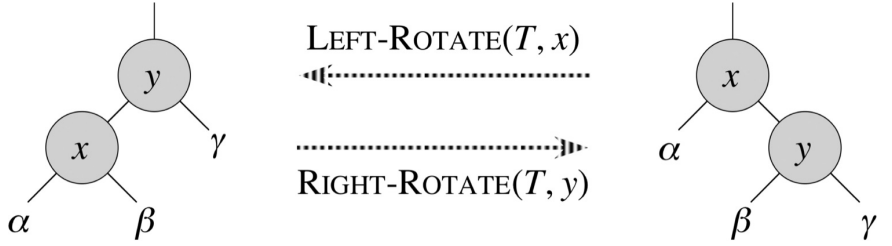

26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3

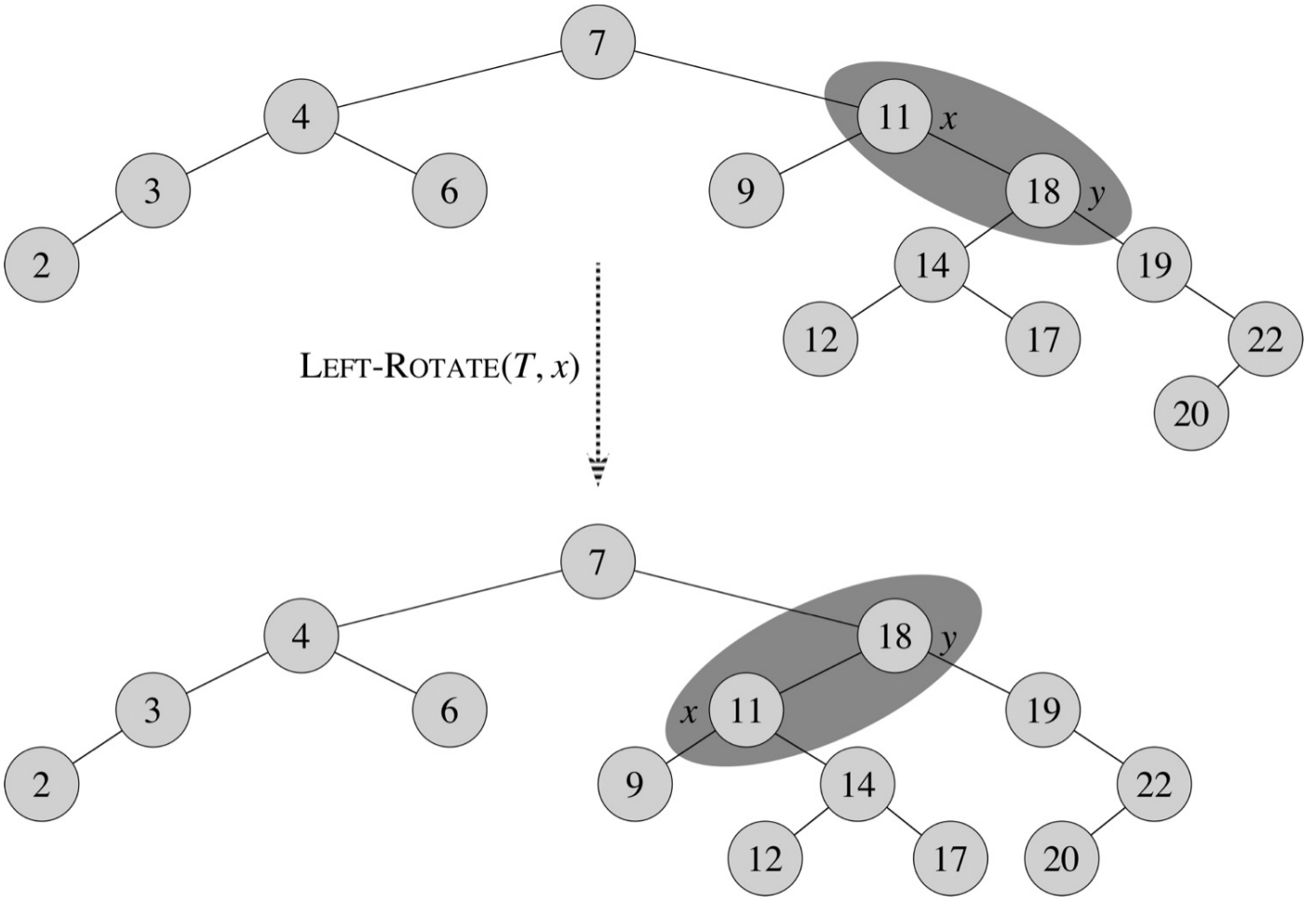
(c)

- Adjustments to the tree structure happen through rotations

```
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
```

O(1)




7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

- Inserting

  - The new node is colored red and put in place using BST insertion

  - A fixup function is called which adjusts things as necessary

RB-Insert-Fixup (T, z)

- Restors the RBT properties after inserting z into T
- Possible violations
  - The root is red
  - z's parent is red

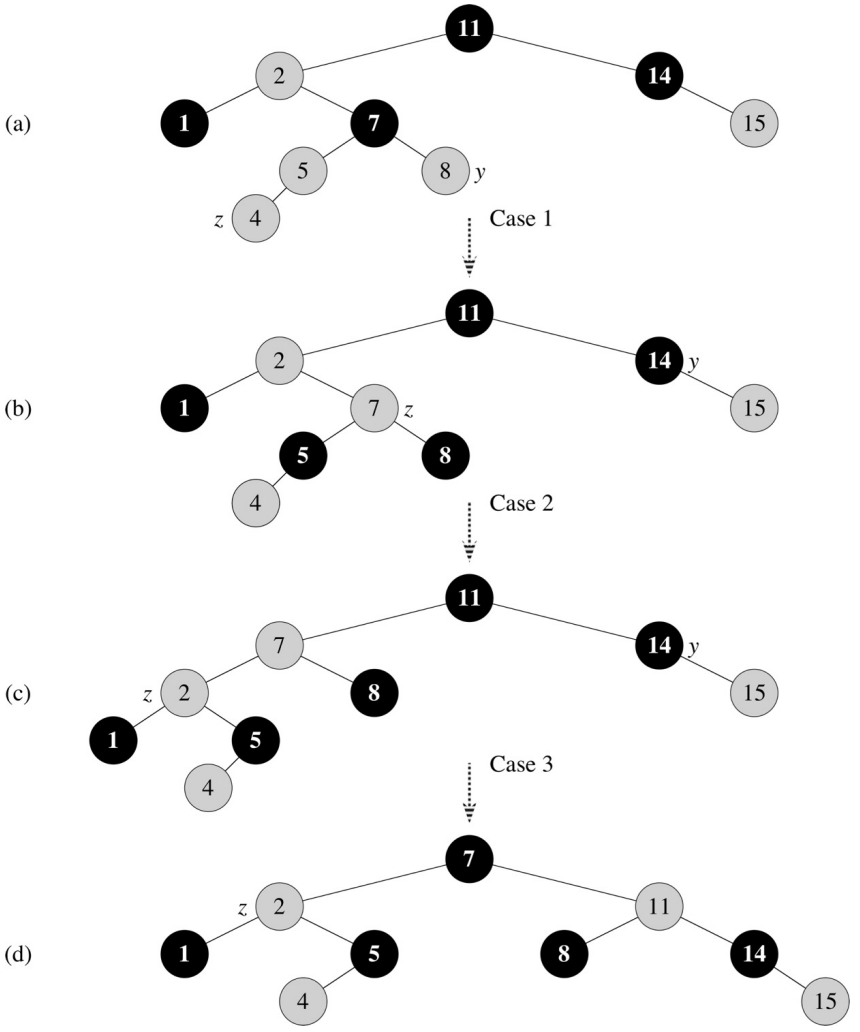
- If z's parent is red
  - If z's uncle is red
    - Color z's parent and z's uncle black
    - Move the z pointer to z's grandparent and repeat the fixup process w/ the new z

- If z's uncle is black
  - If z's parent is a left child and z is a right child, perform a left rotation to make z a left child. If z's parent is a right child and z is a left child, rotate left
  - Color z's parents black, color z's grandparents red, and perform a right rotation on z's grandparent

```
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
```

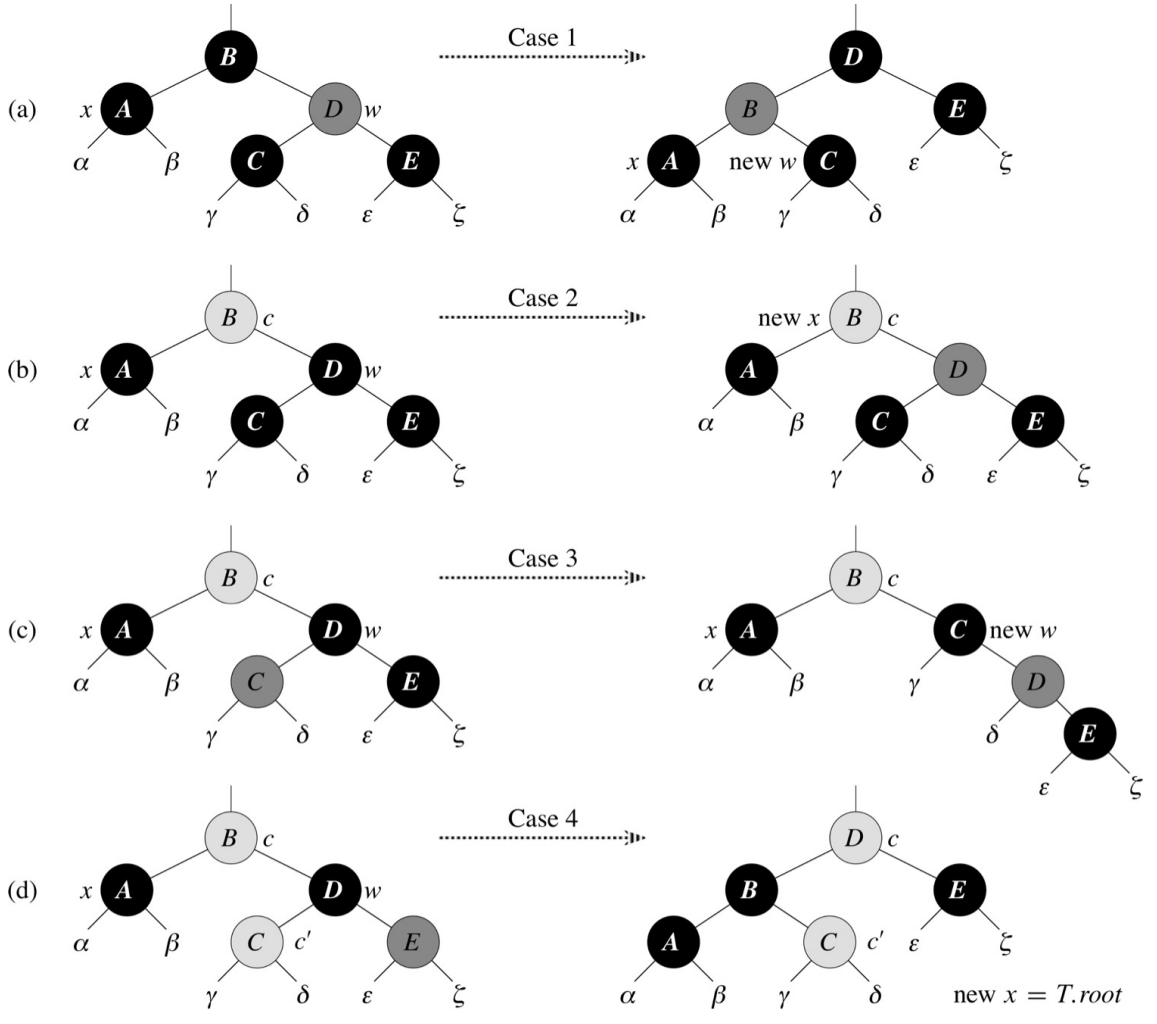
O(h)

O(lg n)


(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15

```
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-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
```


(a)
Case 1
B
x A
D w
α β
C
E
γ δ ε ζ
D
B E
x A
new w C
ε ζ
α β γ δ
(b)
Case 2
B c
x A
D w
α β
C
E
γ δ ε ζ
new x B c
A
D
α β
C
E
γ δ ε ζ
(c)
Case 3
B c
x A
D w
α β
C
E
γ δ ε ζ
B c
x A
C new w
α β γ
D
δ
E
ε ζ
(d)
Case 4
B c
x A
D w
α β
C c′
E
γ δ ε ζ
D c
B
E
A
C c′
ε ζ
α β γ δ
new x = T.root