Data Structures and Algorithms [1]

BITS-Pilani K. K. Birla Goa Campus

[1] Material for the presentation taken from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Third Edition*;

- Which of the following operations will affect the height of a BST?

  SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE

Ch. 13: Red-black Trees

- Which of the following operations will affect the height of a BST?

  SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE
- Can we have a Binary Search Tree data structure that can automatically adjust its height after each insertion (and deletion) so that $h = O(\lg n)$?

Ch. 13: Red-black Trees

- Which of the following operations will affect the height of a BST?

  SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE
- Can we have a Binary Search Tree data structure that can automatically adjust its height after each insertion (and deletion) so that $h = O(\lg n)$?
- Red-black trees are one of many search-tree schemes that are "balanced."

Ch. 13: Red-black Trees

- Which of the following operations will affect the height of a BST?

  SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE
- Can we have a Binary Search Tree data structure that can automatically adjust its height after each insertion (and deletion) so that $h = O(\lg n)$?
- Red-black trees are one of many search-tree schemes that are "balanced."
- Applications of red-black trees :
  - Process scheduler in Linux OS

- Which of the following operations will affect the height of a BST?

  SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE
- Can we have a Binary Search Tree data structure that can automatically adjust its height after each insertion (and deletion) so that $h = O(\lg n)$?
- Red-black trees are one of many search-tree schemes that are "balanced."
- Applications of red-black trees :
  - Process scheduler in Linux OS
  - Implementation of C++ STL : set, map, multiset, multimap.

- Each node in a red-black tree has one extra bit of storage to store the color of the node.

Red-black tree : Properties

- Each node in a red-black tree has one extra bit of storage to store the color of the node.
- Color can be either RED or BLACK.

Red-black tree : Properties

- Each node in a red-black tree has one extra bit of storage to store the color of the node.
- Color can be either RED or BLACK.
- So, each node of the Red-black tree contains the following attributes:

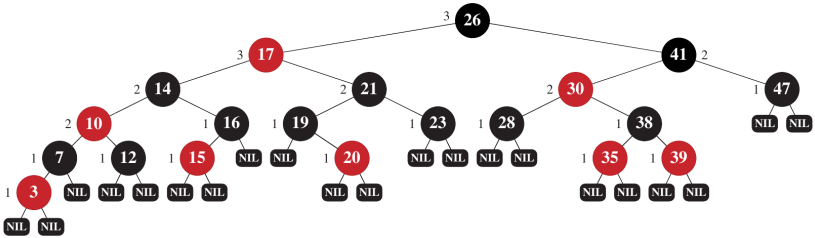
  color, *key*, *left*, *right* and *p*

Red-black tree : Properties

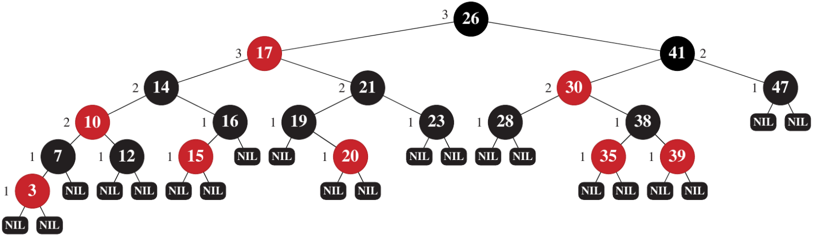
- Each node in a red-black tree has one extra bit of storage to store the color of the node.
- Color can be either RED or BLACK.
- So, each node of the Red-black tree contains the following attributes:

  color, *key*, *left*, *right* and *p*
- In an RB Tree, we will use a Sentinel node NIL instead of the NIL value.

Sentinel NIL leaves in Red-black trees

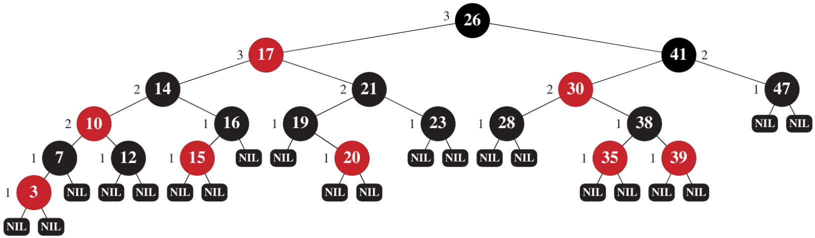


Sentinel NIL leaves in Red-black trees



- When we use the NIL node as a leaf node, the RB tree becomes a full (or a proper) binary tree.

Sentinel NIL leaves in Red-black trees



- When we use the NIL node as a leaf node, the RB tree becomes a full (or a proper) binary tree.
- All the nodes that contain the actual key values become the internal nodes of the RB tree.

Red-black tree : Properties

- A red-black tree is a binary tree that satisfies the following **red-black properties**:

Red-black tree : Properties

- A red-black tree is a binary tree that satisfies the following **red-black properties**:

1. Every node is either red or black.

Red-black tree : Properties

- A red-black tree is a binary tree that satisfies the following **red-black properties**:

1. Every node is either red or black.
2. The root is black.

Red-black tree : Properties

- A red-black tree is a binary tree that satisfies the following **red-black properties**:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL node) is black.

Red-black tree : Properties

- A red-black tree is a binary tree that satisfies the following **red-black properties**:

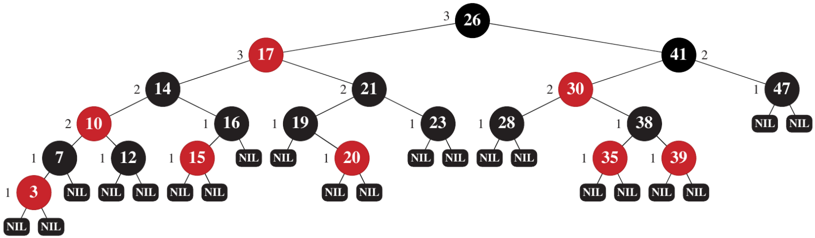
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL node) is black.
4. If a node is red, then both its child nodes are black.

Red-black tree : Properties

- A red-black tree is a binary tree that satisfies the following **red-black properties**:

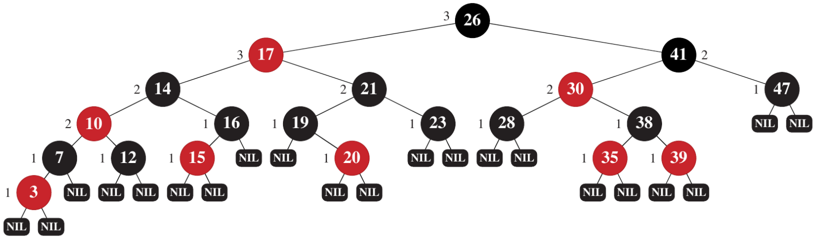
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL node) is black.
4. If a node is red, then both its child nodes are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Are all the RB tree properties satisfied?

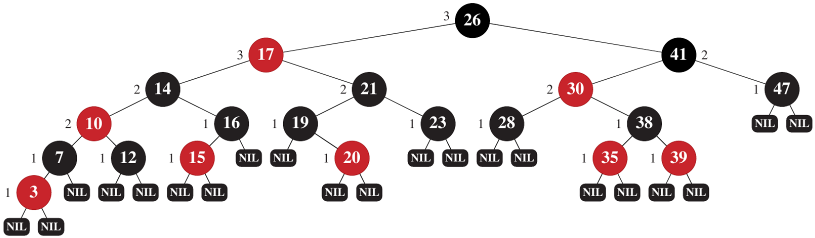


1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL node) is black.
4. If a node is red, then both its child nodes are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Sentinel NIL leaf nodes

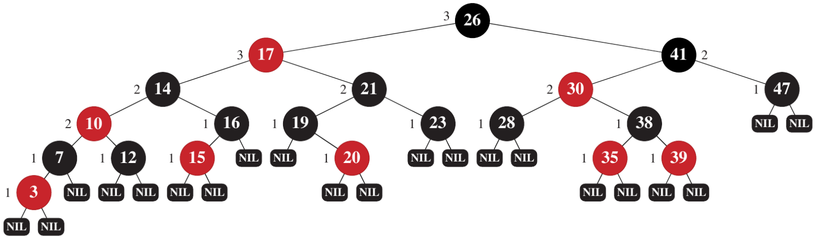


Sentinel NIL leaf nodes



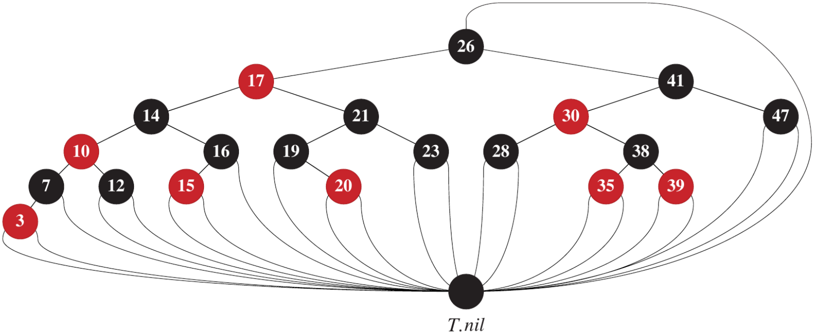
- $T.nil.color = BLACK$;

Sentinel NIL leaf nodes



- $T.nil.color = BLACK$; All the other attributes of $T.nil$, can take arbitrary values — because the other values don't matter.

Red-black tree with one sentinel node



Red-black trees

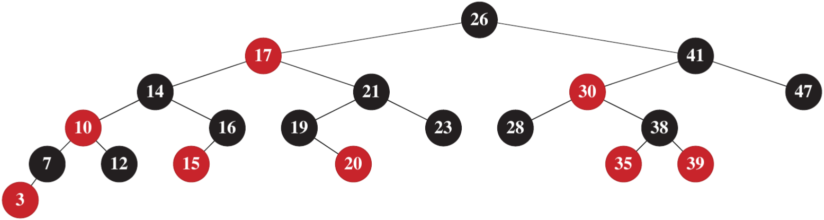
- We are interested only in the internal nodes of the Red-black tree, since they hold the key values.

Red-black trees

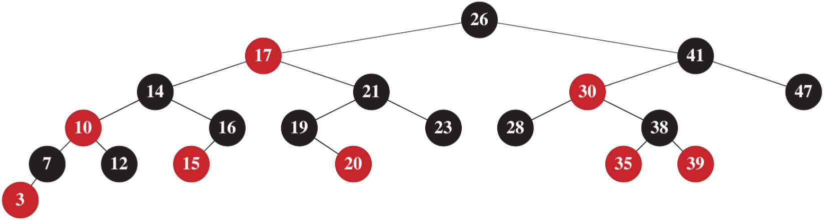
- We are interested only in the internal nodes of the Red-black tree, since they hold the key values.
- So, while drawing we will omit the leaf (i.e. $T.nil$ sentinel) node.

Red-black trees

- We are interested only in the internal nodes of the Red-black tree, since they hold the key values.
- So, while drawing we will omit the leaf (i.e. $T.nil$ sentinel) node.

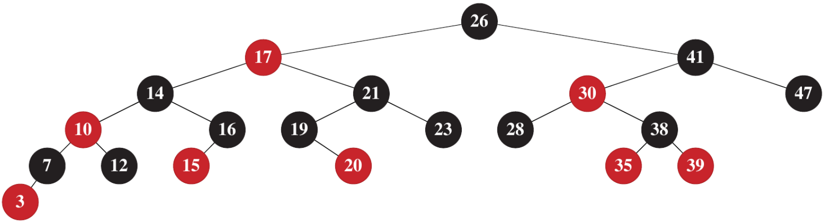


Height and Black height of a node



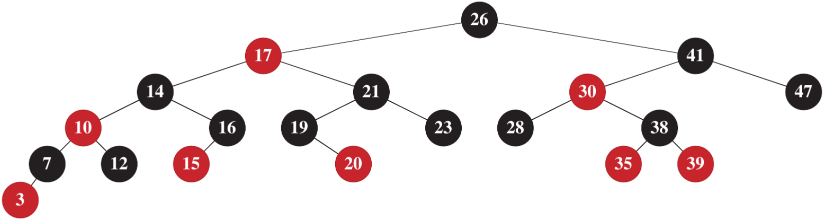
- Height of a node x : Number of edges on the *longest* simple downward path from x to a leaf node.

Height and Black height of a node



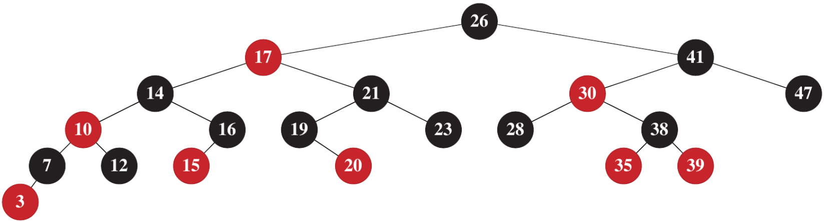
- Height of a node x : Number of edges on the *longest* simple downward path from x to a leaf node.
- What is the height of node **14**?

Height and Black height of a node



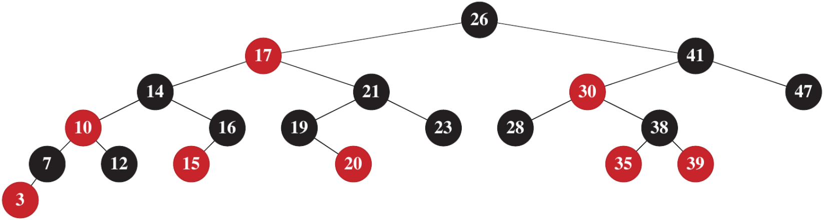
- Height of a node x : Number of edges on the *longest* simple downward path from x to a leaf node.
- What is the height of node **14**? 4

Height and Black height of a node



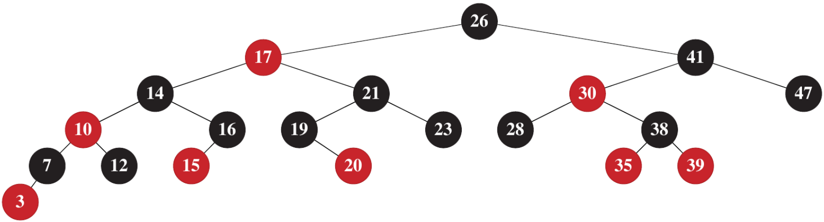
- Height of a node x : Number of edges on the *longest* simple downward path from x to a leaf node.
- What is the height of node **14**? 4
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.

Height and Black height of a node



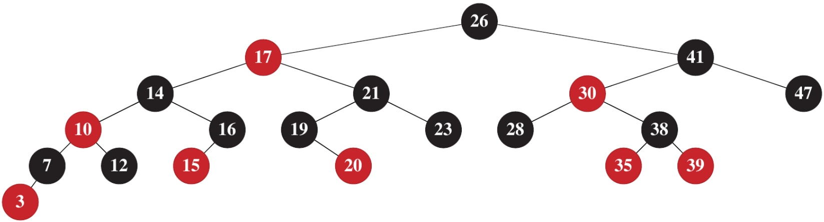
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.

Height and Black height of a node



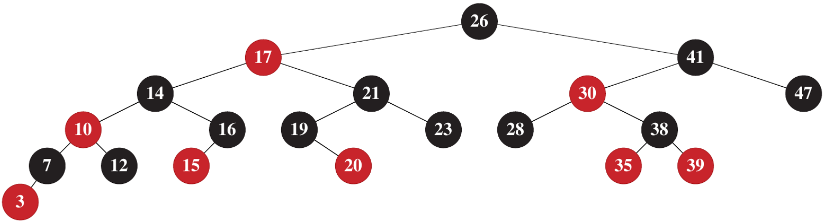
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) =$

Height and Black height of a node



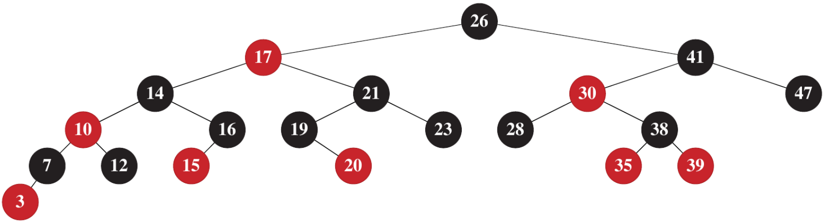
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3$

Height and Black height of a node



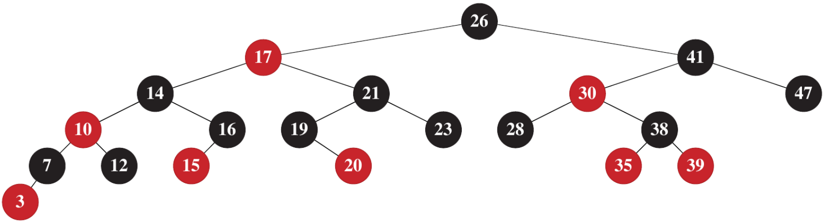
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3$ $\qquad$ $bh(\mathbf{14}) =$

Height and Black height of a node



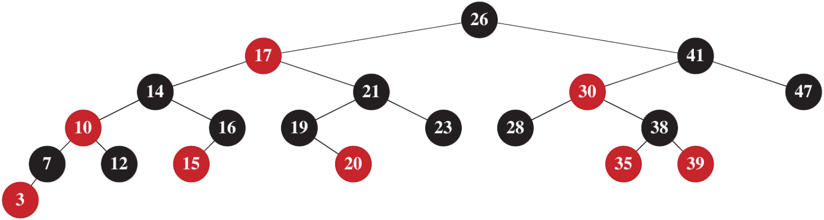
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3 \qquad bh(\mathbf{14}) = 2$

Height and Black height of a node



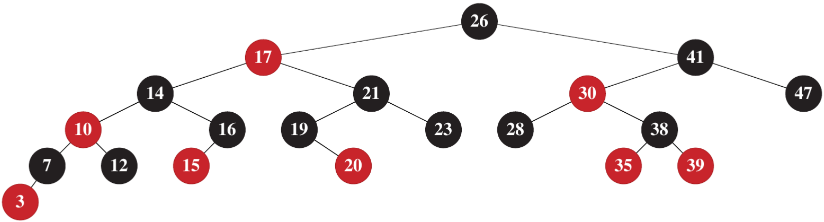
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3 \qquad bh(\mathbf{14}) = 2$
- $bh(\mathbf{41}) =$

Height and Black height of a node



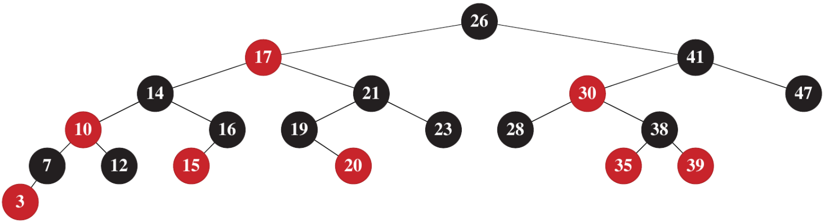
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3 \qquad bh(\mathbf{14}) = 2$
- $bh(\mathbf{41}) = 2$

Height and Black height of a node



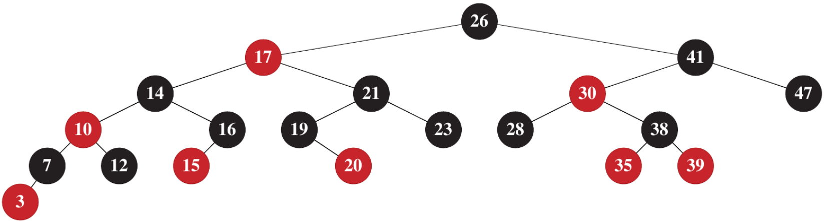
- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3$ $\quad$ $bh(\mathbf{14}) = 2$
- $bh(\mathbf{41}) = 2$ $\quad$ $bh(\mathbf{30}) =$

Height and Black height of a node



- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3$ $\quad bh(\mathbf{14}) = 2$
- $bh(\mathbf{41}) = 2$ $\quad bh(\mathbf{30}) = 2$

Height and Black height of a node



- **Black-height** of node x : Number of black nodes on *any* downward path from node x to a leaf node, not including node x.
- $bh(\mathbf{17}) = 3$ $\quad$ $bh(\mathbf{14}) = 2$
- $bh(\mathbf{41}) = 2$ $\quad$ $bh(\mathbf{30}) = 2$
- Is the following statement true?

  Let x be an internal node. Then black height of a child node of x will be at least $bh(x) - 1$.

Height of Red-black tree

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

Height of Red-black tree

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

- To prove the above lemma we will first prove that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1$

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes $(bh(x) \geq 1)$.

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes ($bh(x) \geq 1$).
  Black height of a child node of node x will be at least $bh(x) - 1$.

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes ($bh(x) \geq 1$).
  Black height of a child node of node x will be at least $bh(x) - 1$.
- Induction step assumption : There are at least $2^{bh(x)-1} - 1$ internal nodes in the subtree rooted at a child node of x.

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes ($bh(x) \geq 1$).
  Black height of a child node of node x will be at least $bh(x) - 1$.
- Induction step assumption : There are at least $2^{bh(x)-1} - 1$ internal nodes in the subtree rooted at a child node of x.
- Number of internal nodes in a subtree rooted at x

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes $(bh(x) \geq 1)$.
  Black height of a child node of node x will be at least $bh(x) - 1$.
- Induction step assumption : There are at least $2^{bh(x)-1} - 1$ internal nodes in the subtree rooted at a child node of x.
- Number of internal nodes in a subtree rooted at x

$$\geq 2^{bh(x)-1} - 1 +$$

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes ($bh(x) \geq 1$).
  Black height of a child node of node x will be at least $bh(x) - 1$.
- Induction step assumption : There are at least $2^{bh(x)-1} - 1$ internal nodes in the subtree rooted at a child node of x.
- Number of internal nodes in a subtree rooted at x

$$\geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 +$$

Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes ($bh(x) \geq 1$).
  Black height of a child node of node x will be at least $bh(x) - 1$.
- Induction step assumption : There are at least $2^{bh(x)-1} - 1$ internal nodes in the subtree rooted at a child node of x.
- Number of internal nodes in a subtree rooted at x

$$\geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1$$

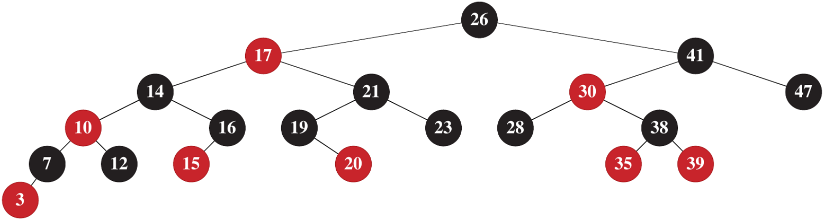
Minimum nodes in a subtree rooted at node x

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

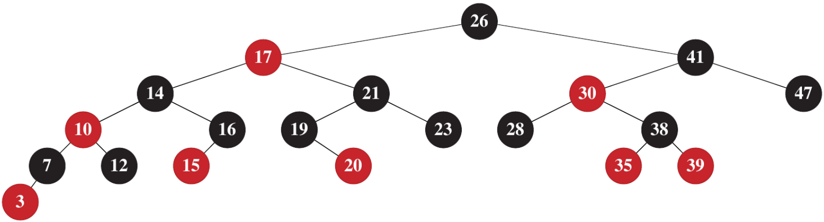
- We will do proof by induction
- Base case: $bh(x) = 0$ (means x is a leaf node)
  Number of internal nodes $\geq 2^{bh(x)} - 1 = 0$
- Induction step:
  Let x be an internal node having two child nodes ($bh(x) \geq 1$).
  Black height of a child node of node x will be at least $bh(x) - 1$.
- Induction step assumption : There are at least $2^{bh(x)-1} - 1$ internal nodes in the subtree rooted at a child node of x.
- Number of internal nodes in a subtree rooted at x

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

Red-black tree properties



Red-black tree properties



1. Every node is 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 child nodes are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Height of Red-black tree

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

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node).

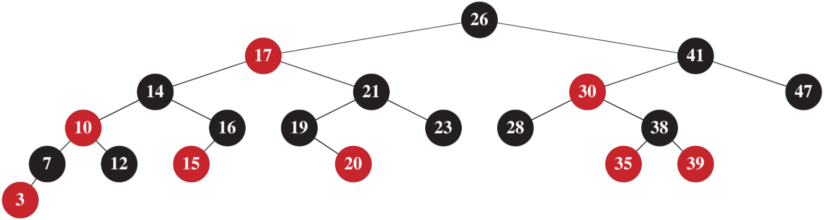
Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node). $bh(T.root)$ must be at least $h/2$.

$bh(T.root)$



- $bh(T.root)$ must be at least $h/2$.

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node).
  $bh(T.root)$ must be at least $h/2$. (Due to property 4)

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node). $bh(T.root)$ must be at least $h/2$. (Due to property 4) So, the number of internal nodes in the RB tree must be at least:

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node). $bh(T.root)$ must be at least $h/2$. (Due to property 4) So, the number of internal nodes in the RB tree must be at least:

$$n \geq 2^{bh(T.root)} - 1$$

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node). $bh(T.root)$ must be at least $h/2$. (Due to property 4)
  So, the number of internal nodes in the RB tree must be at least:

$$
\begin{aligned}
n &\geq 2^{bh(T.root)} - 1 \\
&\geq 2^{h/2} - 1
\end{aligned}
$$

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node). $bh(T.root)$ must be at least $h/2$. (Due to property 4) So, the number of internal nodes in the RB tree must be at least:

$$
\begin{aligned}
n &\geq 2^{bh(T.root)} - 1 \\
&\geq 2^{h/2} - 1 \\
(n+1) &\geq 2^{h/2}
\end{aligned}
$$

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node). $bh(T.root)$ must be at least $h/2$. (Due to property 4) So, the number of internal nodes in the RB tree must be at least:

$$
\begin{aligned}
n &\geq 2^{bh(T.root)} - 1 \\
&\geq 2^{h/2} - 1 \\
(n+1) &\geq 2^{h/2} \\
\lg(n+1) &\geq h/2
\end{aligned}
$$

Height of Red-black tree

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

Lemma 2: A subtree rooted at any node x in a RB tree contains at least $2^{bh(x)} - 1$ internal nodes.

- Let h be the height of the RB tree (i.e the root node). $bh(T.root)$ must be at least $h/2$. (Due to property 4) So, the number of internal nodes in the RB tree must be at least:

$$\begin{aligned} n &\geq 2^{bh(T.root)} - 1 \\ &\geq 2^{h/2} - 1 \\ (n+1) &\geq 2^{h/2} \\ \lg(n+1) &\geq h/2 \\ h &\leq 2\lg(n+1) \end{aligned}$$

Implication of **Lemma 1**

- $2\lg(n+1) = O(\lg n)$

Implication of **Lemma 1**

- $2\lg(n+1) = O(\lg n)$

$$2\lg(n+1) \leq c \lg n$$

Implication of **Lemma 1**

- $2\lg(n+1) = O(\lg n)$

$$2\lg(n+1) \leq c \lg n$$
$$\lg(n+1)^2 \leq \lg n^c$$

Implication of **Lemma 1**

- $2\lg(n+1) = O(\lg n)$

$$\begin{aligned} 2\lg(n+1) &\leq c\lg n \\ \lg(n+1)^2 &\leq \lg n^c \\ (n+1)^2 &\leq n^c \end{aligned}$$

Implication of **Lemma 1**

- $2\lg(n+1) = O(\lg n)$

$$
\begin{aligned}
2\lg(n+1) &\le c \lg n \\
\lg(n+1)^2 &\le \lg n^c \\
(n+1)^2 &\le n^c \\
c = 3, &\qquad n_0 = 3
\end{aligned}
$$

Implication of **Lemma 1**

- $2\lg(n+1) = O(\lg n)$

$$\begin{aligned} 2\lg(n+1) &\leq c\lg n \\ \lg(n+1)^2 &\leq \lg n^c \\ (n+1)^2 &\leq n^c \\ c = 3, &\qquad n_0 = 3 \end{aligned}$$

- If all the red-black tree properties are satisfied, then $h = O(\lg n)$.

Implication of **Lemma 1**

- $2\lg(n+1) = O(\lg n)$

$$
\begin{aligned}
2\lg(n+1) &\leq c\lg n \\
\lg(n+1)^2 &\leq \lg n^c \\
(n+1)^2 &\leq n^c \\
c = 3, &\qquad n_0 = 3
\end{aligned}
$$

- If all the red-black tree properties are satisfied, then $h = O(\lg n)$.
- If we can ensure that the red-black tree properties are always satisfied, then SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE operations can be performed in $O(\lg n)$ time.

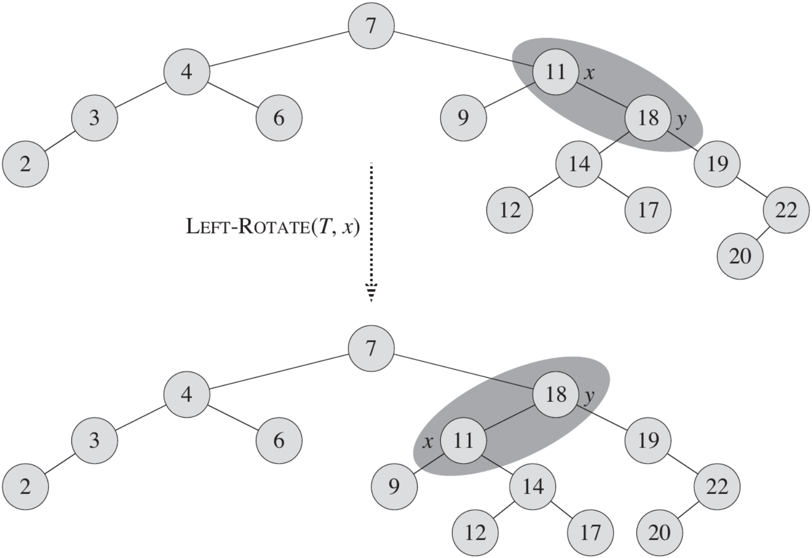
Rotations

- Rotation will be one of the operations that will help us ensure that the red-black tree properties hold after insertions and deletions.

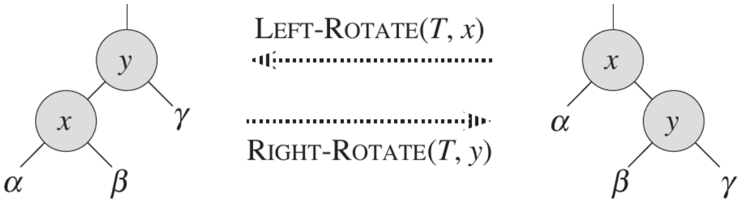
Rotations

- Rotation will be one of the operations that will help us ensure that the red-black tree properties hold after insertions and deletions.
- Binary search tree property continues to be satisfied after rotation

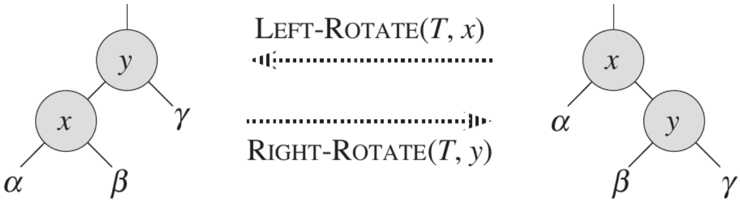
Rotation: BST property continues to be satisfied



Left and Right rotation



Left and Right rotation



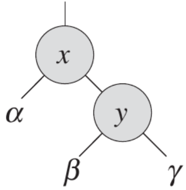
- BST property is maintained by both left and right rotations.

Pseudocode for Left rotate

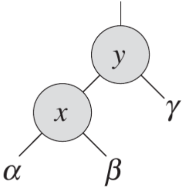
LEFT-ROTATE(T, x)

```
y = x.right
x.right = y.left
```

Before



After

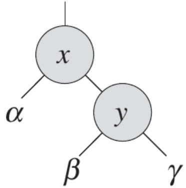


Pseudocode for Left rotate

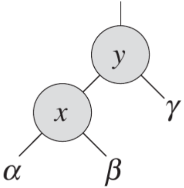
LEFT-ROTATE(T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
        y.left.p = x
```

Before



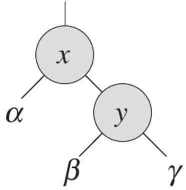
After



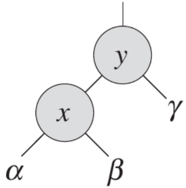
Pseudocode for Left rotate

```
LEFT-ROTATE(T, x)
y = x.right
x.right = y.left
if y.left ≠ T.nil
        y.left.p = x
y.p = x.p
```

Before



After

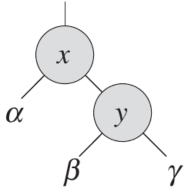


Pseudocode for Left rotate

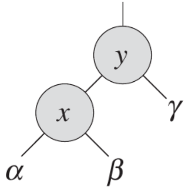
LEFT-ROTATE(T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
        y.left.p = x
y.p = x.p
if x.p == T.nil
        T.root = y
elseif x == x.p.left
        x.p.left = y
else x.p.right = y
```

Before



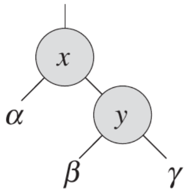
After



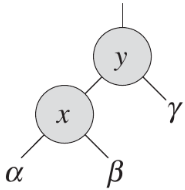
Pseudocode for Left rotate

```
LEFT-ROTATE(T, x)
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
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
x.p = y
```

Before



After



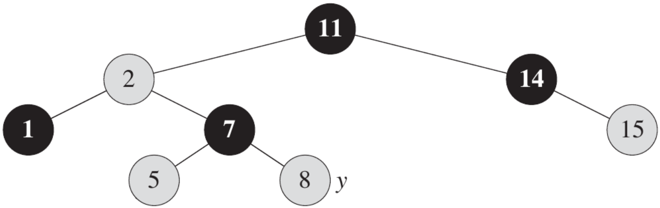
ROTATE procedure

- Running time of left and right rotate operations :

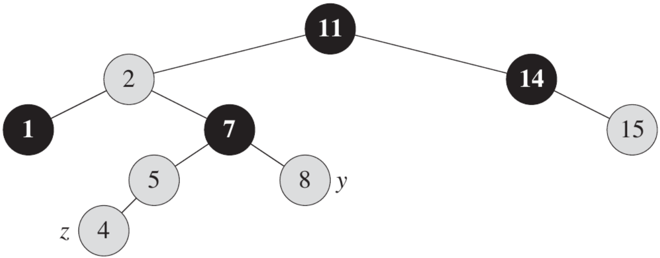
ROTATE procedure

- Running time of left and right rotate operations : $O(1)$

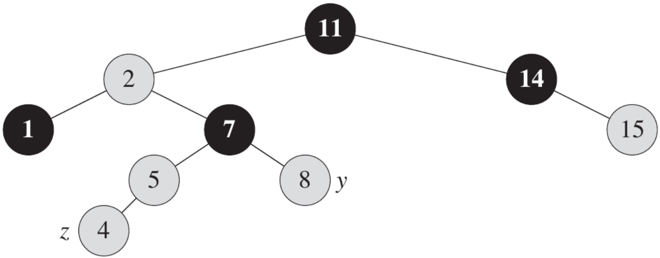
Red-black tree insert



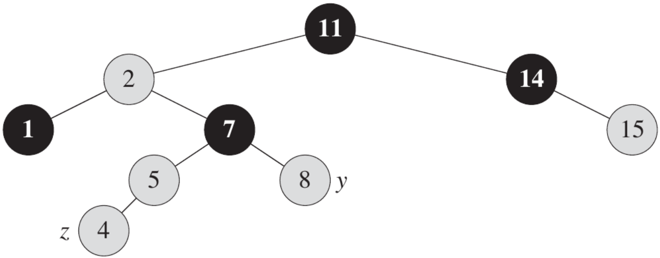
Red-black tree insert



Red-black tree insert



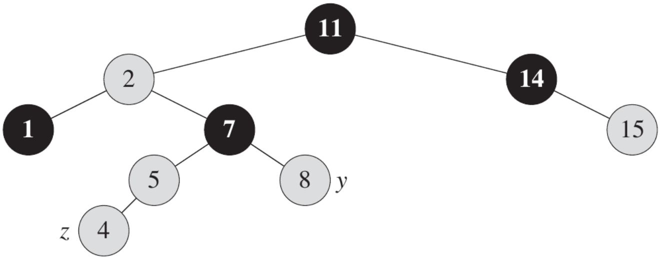
Differences with insert operation in BST:



Differences with insert operation in BST:

1. The new node is inserted as an internal node of the RB tree.

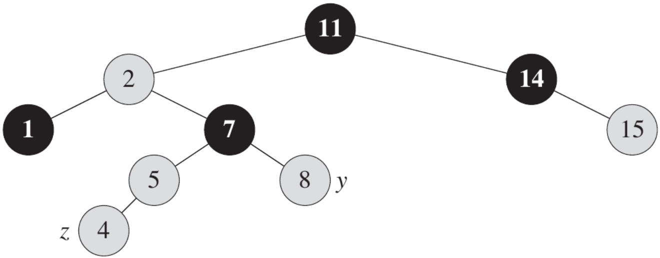
Red-black tree insert



Differences with insert operation in BST:

1. The new node is inserted as an internal node of the RB tree.
2. New node is always colored red.

Red-black tree insert

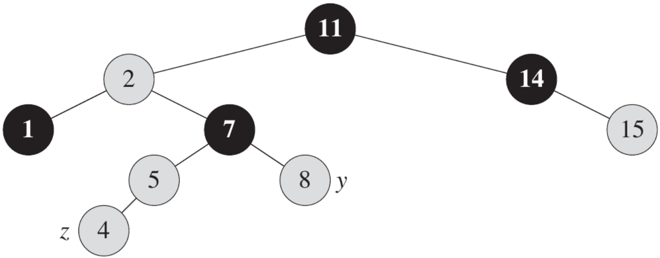


Differences with insert operation in BST:

1. The new node is inserted as an internal node of the RB tree.
2. New node is always colored red.

- What RB tree property can get violated as a result of a node insertion?

Red-black tree insert



Differences with insert operation in BST:

1. The new node is inserted as an internal node of the RB tree.
2. New node is always colored red.

- What RB tree property can get violated as a result of a node insertion?
- Either property 2 or property 4 (not both) can get violated.

RB-INSERT : Red-black tree insert

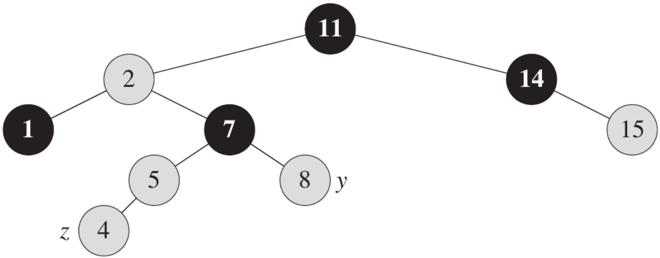
```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

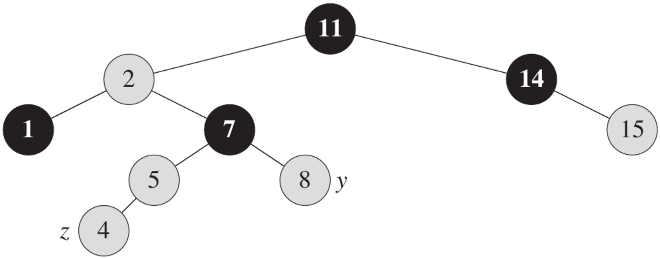
```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

RB-INSERT-FIXUP : Fix the violated property

```
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
```

RB-INSERT-FIXUP : Case 1



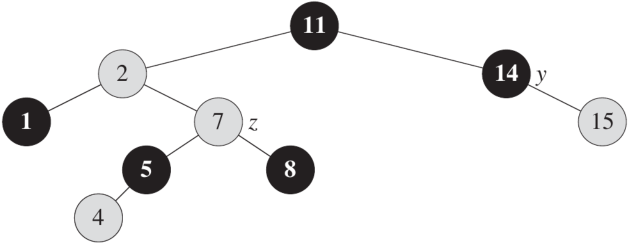


- In case 1, we are assuming that z.p.p is a black node. Why?

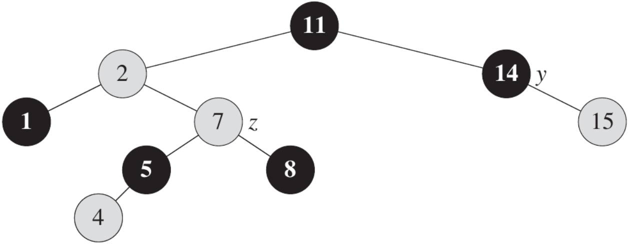
RB-INSERT-FIXUP : Fix the violated property

```
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
```

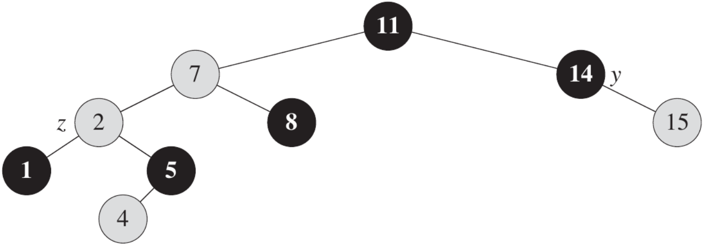
Case 2:



Case 2:



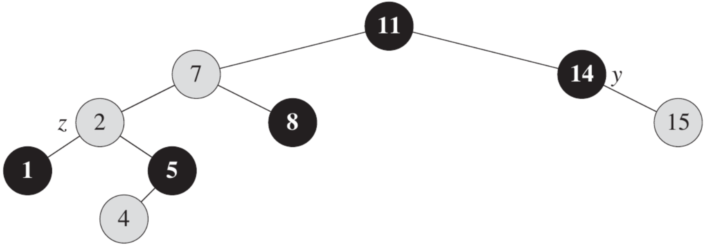
Case 3:



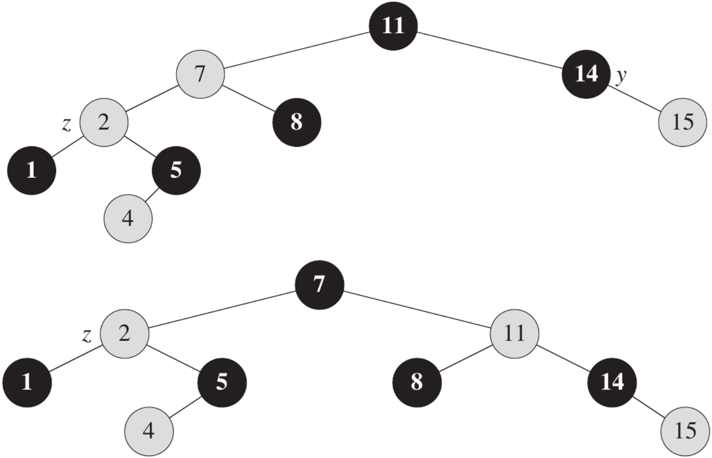
RB-INSERT-FIXUP : Fix the violated property

```
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
```

RB-INSERT-FIXUP : Case 3



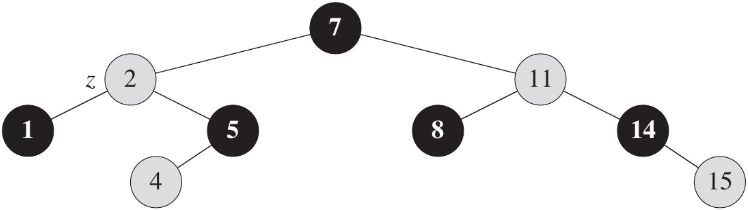
RB-INSERT-FIXUP : Case 3



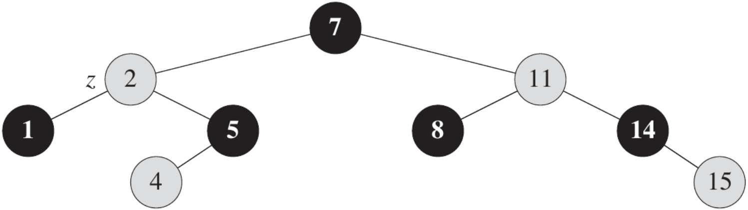
RB-INSERT-FIXUP : Fix the violated property

```
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
```

RB-INSERT-FIXUP : All RB tree properties satisfied



RB-INSERT-FIXUP : All RB tree properties satisfied



- Note: Color of the y node helps in differentiating Case 1 from Case 2 and Case 3.

RB-INSERT-FIXUP : Last three cases

```
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
```

RB-INSERT-FIXUP

- Each iteration ensures that either only Property 2 or only Property 4 is violated.

RB-Insert-Fixup

- Each iteration ensures that either only Property 2 or only Property 4 is violated.
- 6 cases are sufficient to cover all scenarios.

RB-Insert-Fixup

- Each iteration ensures that either only Property 2 or only Property 4 is violated.
- 6 cases are sufficient to cover all scenarios.
- In each iteration, the z node moves up the RB tree; and each iteration takes constant time.

RB-INSERT-FIXUP

- Each iteration ensures that either only Property 2 or only Property 4 is violated.
- 6 cases are sufficient to cover all scenarios.
- In each iteration, the z node moves up the RB tree; and each iteration takes constant time.
- RB-INSERT-FIXUP would take $O(\lg n)$ time.

RB-INSERT-FIXUP

- Each iteration ensures that either only Property 2 or only Property 4 is violated.
- 6 cases are sufficient to cover all scenarios.
- In each iteration, the z node moves up the RB tree; and each iteration takes constant time.
- RB-INSERT-FIXUP would take $O(\lg n)$ time.
- Overall running time of RB-INSERT will be $O(\lg n)$.

RB-INSERT-FIXUP

- Each iteration ensures that either only Property 2 or only Property 4 is violated.
- 6 cases are sufficient to cover all scenarios.
- In each iteration, the z node moves up the RB tree; and each iteration takes constant time.
- RB-INSERT-FIXUP would take $O(\lg n)$ time.
- Overall running time of RB-INSERT will be $O(\lg n)$.
- What Binary search tree will we get if we insert keys in the following order : 4, 3, 2, 1 ?

RB-INSERT-FIXUP

- Each iteration ensures that either only Property 2 or only Property 4 is violated.
- 6 cases are sufficient to cover all scenarios.
- In each iteration, the z node moves up the RB tree; and each iteration takes constant time.
- RB-INSERT-FIXUP would take $O(\lg n)$ time.
- Overall running time of RB-INSERT will be $O(\lg n)$.
- What Binary search tree will we get if we insert keys in the following order : 4, 3, 2, 1 ?
- What RB tree will we get for the above example?

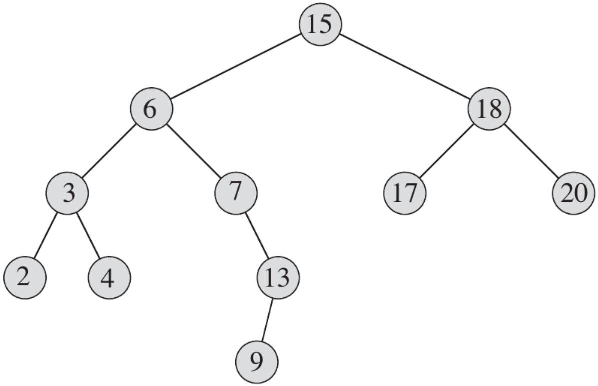
RB-DELETE procedure

- RB-DELETE procedure is similar to the TREE-DELETE procedure.

RB-DELETE procedure

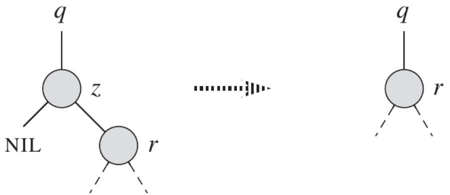
- RB-DELETE procedure is similar to the TREE-DELETE procedure.
- For deleting a node, we need to considered multiple cases.

Tree-Delete

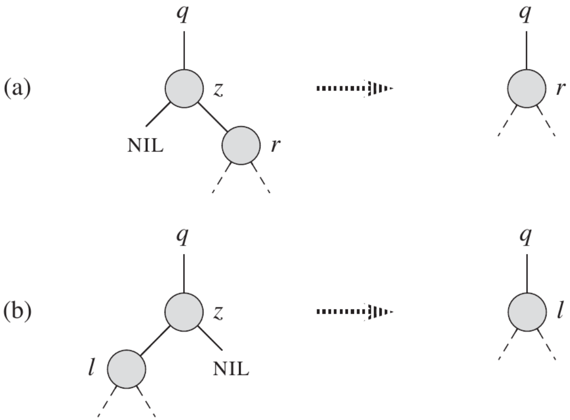


RB-DELETE similar to TREE-DELETE

(a)



RB-DELETE similar to TREE-DELETE



- When node z has two child nodes, we find the successor of node z.

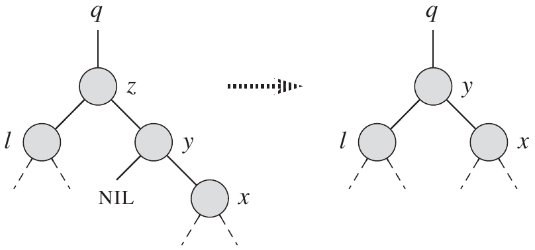
TREE-DELETE : Node z has two child nodes

- When node z has two child nodes, we find the successor of node z.
- We replace node z with its successor.

TREE-DELETE : Node z has two child nodes

(c) Successor of node z is its right child.

(c)

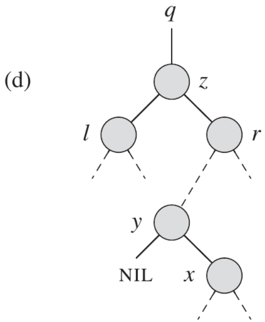


TREE-DELETE : Node z has two child nodes

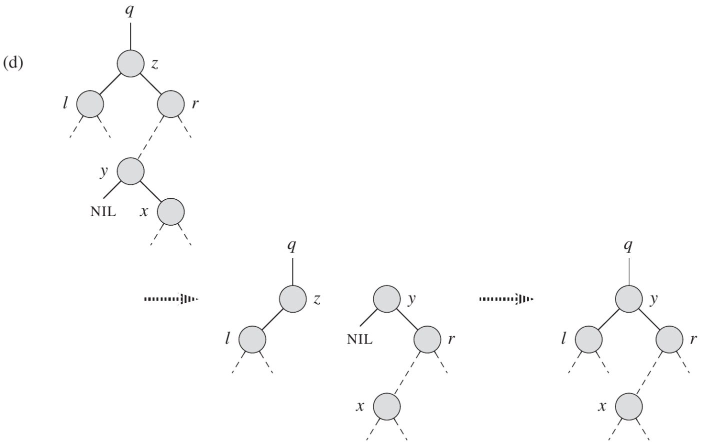
(d) Successor of node z is not its right child

TREE-DELETE : Node z has two child nodes

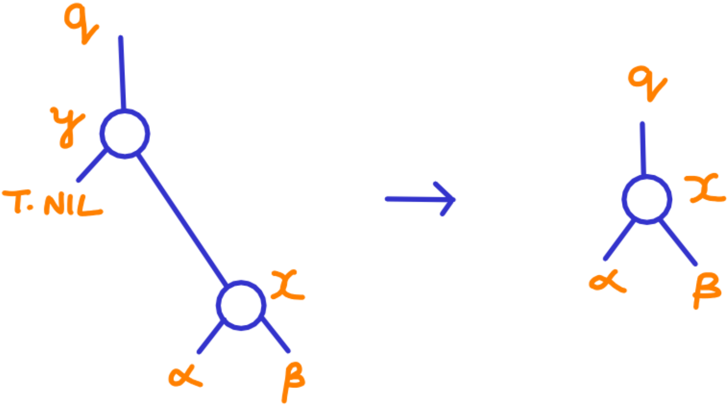
(d) Successor of node z is not its right child



(d) Successor of node z is not its right child



RB-DELETE procedure



RB-DELETE procedure

- If the node x is "red-and-black", we can make it black. (All the RB tree properties will be satisfied.)

RB-DELETE procedure

- If the node x is "red-and-black", we can make it black. (All the RB tree properties will be satisfied.)
- If the node x is "doubly black", then we will try to move the "doubly black" node up the tree.

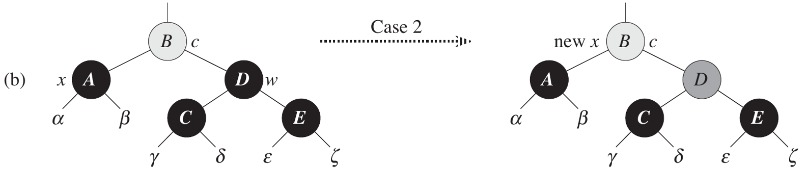
RB-DELETE procedure

RB-DELETE procedure : Case 2

RB-DELETE procedure : Case 2

- How are we sure that the sibling of node x is not a leaf node?

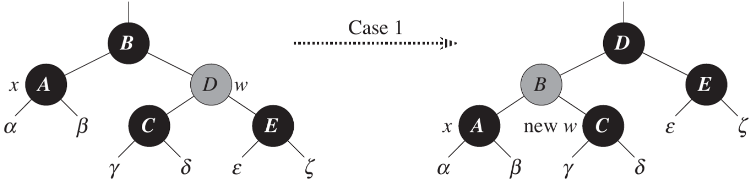
RB-DELETE procedure : Case 2



RB-DELETE procedure : Case 1

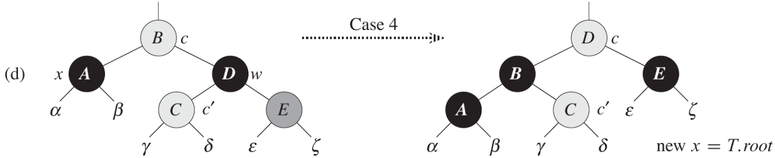
RB-Delete procedure : Case 1

(a)



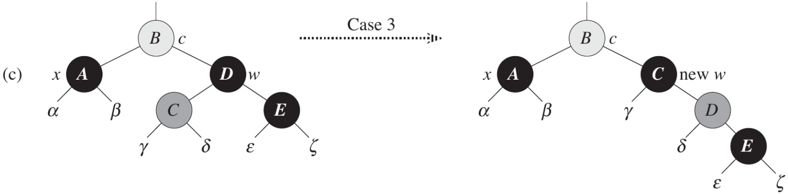
RB-DELETE procedure : Case 4

RB-DELETE procedure : Case 4

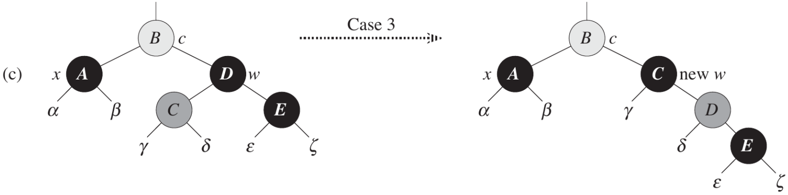


RB-DELETE procedure : Case 3

RB-DELETE procedure : Case 3



RB-DELETE procedure : Case 3



RB Tree Visualization tool:
`https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`

RB-Delete

- In case 2, we either fix the problem or move a doubly black node up the tree.

RB-Delete

- In case 2, we either fix the problem or move a doubly black node up the tree.
- In case 3 and 4, we fix property 5 in at most two rotations.

RB-Delete

- In case 2, we either fix the problem or move a doubly black node up the tree.
- In case 3 and 4, we fix property 5 in at most two rotations.
- In case 1, we obtain case 2, 3 or 4 in one rotation.

RB-Delete

- In case 2, we either fix the problem or move a doubly black node up the tree.
- In case 3 and 4, we fix property 5 in at most two rotations.
- In case 1, we obtain case 2, 3 or 4 in one rotation.
- So, RB-DELETE-FIXUP procedure would take $O(\lg n)$ time in the worst case.

RB-Delete

- In case 2, we either fix the problem or move a doubly black node up the tree.
- In case 3 and 4, we fix property 5 in at most two rotations.
- In case 1, we obtain case 2, 3 or 4 in one rotation.
- So, RB-DELETE-FIXUP procedure would take $O(\lg n)$ time in the worst case.
- The overall running time of RB-DELETE procedure will be $O(\lg n)$ time.

TRANSPLANT procedure in RB-DELETE

```
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
```

```
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 procedure

Red-black trees: Conclusions

- RB-INSERT and RB-DELETE procedures take $O(\lg n)$ time and maintain all the red-black tree properties.

Red-black trees: Conclusions

- RB-INSERT and RB-DELETE procedures take $O(\lg n)$ time and maintain all the red-black tree properties.
- So, Red-black trees can perform all the common dynamic set operations in $O(\lg n)$ time.