Red-Black Trees

Ali Alilooee

Ohio State University

Presentation Outline

Red-black tree

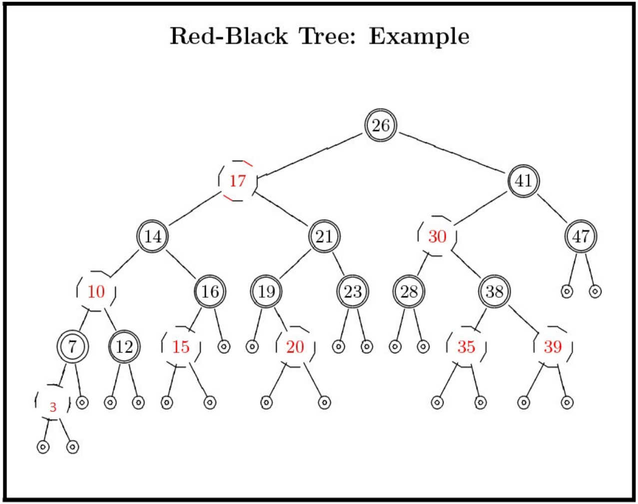
Definition

A **red-black tree** is a binary search tree with the following properties:

- Every node is either red or black;
- The root is black;
- Every leaf is NIL and is black;
- If a node is red, then both its children are black;
- All simple paths from the root to any leaf contain the same number of black nodes.
- (Note: Every node in a binary tree is either a leaf or has BOTH a left AND right child.)

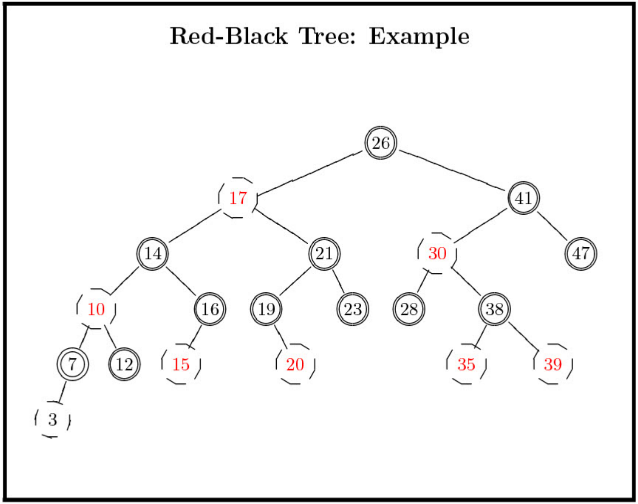
Example

CSE 2331



Example

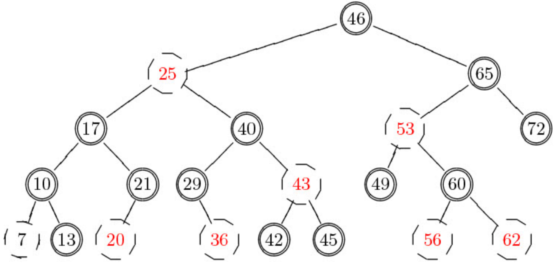
CSE 2331



Example

NOT a Red-Black Tree

This tree is NOT a red-black tree. Why not?



Some theorem

Theorem

In a red-black tree, at least half of the nodes on any paths from the root to a leaf must be black. In other words, if x is the root of the red-black tree T, then we have

$$\# \text{ black of } x \geq \frac{h}{2}.$$

Black-height

Definition

The black height of a node x on a red-black tree T is the number of black nodes on any path to a NIL not containing x. We denote the black-height x with $bh(x)$.

Black-height

Definition

The black height of a node x on a red-black tree T is the number of black nodes on any path to a NIL not containing x. We denote the black-height x with $bh(x)$.

Theorem

Let T be a red-black tree with n nodes and root x. Then we have

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

Proof.

Extra Extra credit. Use induction on h. $\square$

Another theorem

Theorem

A red black-tree with n nodes has height h for which we have

$$h \leq 2\log(n+1).$$

Proof.

Since we have $n \geq 2^{bh(x)} - 1$ then we can conclude

$$bh(x) \leq \log(n+1).$$

Another theorem

Theorem

A red black-tree with n nodes has height h for which we have

$$h \leq 2\log(n+1).$$

Proof.

Since we have $n \geq 2^{bh(x)} - 1$ then we can conclude

$$bh(x) \leq \log(n+1).$$

On the other hand, we know $\frac{h}{2} \leq bh(x)$

Another theorem

Theorem

A red black-tree with n nodes has height h for which we have

$$h \leq 2\log(n+1).$$

Proof.

Since we have $n \geq 2^{bh(x)} - 1$ then we can conclude

$$bh(x) \leq \log(n+1).$$

On the other hand, we know $\frac{h}{2} \leq bh(x)$and then we can conclude

$$h \leq 2\log(n+1).$$

$\square$

Presentation Outline

Insert

function RBLocateParent(T, z)

$y \leftarrow NIL$;
$x \leftarrow T.root$;
while (x is not a leaf) **do**
    $y \leftarrow x$;
    if ($z.key < x.key$) **then**
        $x \leftarrow x.left$;
    else
        $x \leftarrow x.right$;
    end if
end while
return (y);

Insert

function RBTreeInsert(T, z)

```
y ← RBLocateParent(T, z);
z.parent ← y;
if (y = NIL) then
    T.root ← z;
else if (z.key < y.key) then
    y.left ← z;
else
    y.right ← z;
end if
z.left ← leaf;
z.right ← leaf;
z.color ← Red;
RBInsertFixup(T,z);
```

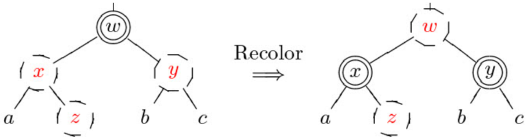
Insert

function Sibling(x) * Return sibling of x */

```
if (x.parent = NIL) then
    error "Root has no siblings.";
end if
p ← x.parent;
if (p.left = x) then
    return (p.right);
else
    return (p.left);
end if
```

Insert Fixup: Case I

Red-Black Tree Insert Fixup: Case I



function RBInsertFixupA(T, **alters** z)

while ($z \neq T$.root) **and** (z.parent.color $\neq$ Black) **do**

- $y \leftarrow$ Sibling(z.parent);
- **if** (y.color $=$ Black) **then return**;
- z.parent.color $\leftarrow$ Black;
- y.color $\leftarrow$ Black;
- $z \leftarrow z$.parent.parent;
- z.color $\leftarrow$ Red;

end

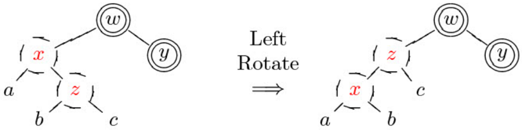
Insert

function RBInsertFixupA $(T, \text{alters } z)$ * Return sibling of x */

while $(z \neq T.root)$ and $(z.parent.color \neq Black)$ **do**
  $y \leftarrow Sibling(z.parent)$;
  if $(y.color = Black)$ **then return** ;
  end if
  $z.parent.color \leftarrow Black$;
  $y.color \leftarrow Black$;
  $z \leftarrow z.parent.parent$;
  $z.color \leftarrow Red$;
end while

CSE 2331

Insert Fixup: Case II



If the parent x of z is red and its "uncle" is black:
If z is a right child and its parent x is a left child:

- $z \leftarrow x$
- Left Rotate on x;
- Apply algorithm for Case III.

Insert

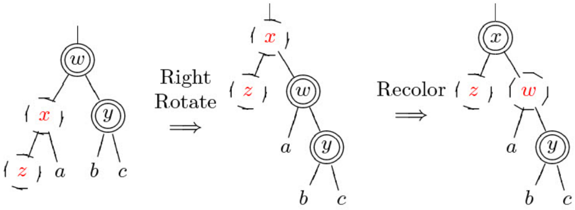
function RBInsertFixupB $(T, \text{alters } z)$ * Return sibling of x */

if $(z = T.root)$ and $(z.parent.color = Black)$ **then return** ;
end if
$x \leftarrow z.parent$;
$w \leftarrow x.parent$;
if $(z = x.right)$ and $(x = w.left)$ **then**
    $z \leftarrow x$;
    LeftRotate(T, w);
else if $(z = x.left)$ and $(x = w.right)$ **then**
    Handle same as above with “right” and “left” exchanged
    ...
end if

Insert Fixup: Case III

CSE 2331

Insert Fixup: Case III



If the parent of z is red and its "uncle" is black:
If z is a left child and its parent is a left child:

- Right Rotate on the grandparent of z;
- Color the parent of z Black;
- Color the sibling of z red.

Insert

function RBInsertFixupC (T, alters z) * Return sibling of x */

if $(z = T.root)$ and $(z.parent.color = Black)$ **then return** ;
end if
$x \leftarrow z.parent$;
$w \leftarrow x.parent$;
if $(z = x.left)$ and $(x = w.left)$ **then**
    RightRotate(T, w);
    $x.color \leftarrow Black$;
    $w.color \leftarrow Red$;
else if $(z = x.right)$ and $(x = w.right)$ **then**
    Handle same as above with "right" and "left" exchanged
    ...
end if