Analysis of Algorithms

Red-Black Trees

Andres Mendez-Vazquez

October 1, 2018

Outline

Outline

Well Balanced Trees

Do you remember AVL (Adelson-Velskii and Landis) Trees?

Quite nice recursive methods for balancing the tree!!!

It is based on an height Invariant

At any node in the tree, the heights of the left and right subtrees differs by at most 1.

Well Balanced Trees

Do you remember AVL (Adelson-Velskii and Landis) Trees?

Quite nice recursive methods for balancing the tree!!!

It is based on an height Invariant

At any node in the tree, the heights of the left and right subtrees differs by at most 1.

Thus, it is necessary to add an extra field to the Node Structure

Structure of a Node

Structure 1: STRUCT NODE

```
Key key
int height
Value val
Node Left, Right
```

Outline

Observations

Due to the balancing methods

- AVL Trees requiere to have a $O(\log_2 n)$ rotations.
- AVL Trees operations cost $O(\log_2 n)$.

Observations

Due to the balancing methods

- AVL Trees requiere to have a $O\left(\log_2 n\right)$ rotations.
- AVL Trees operations cost $O\left(\log_2 n\right)$.

There is a problem about the height

How much memory it is necessary to allocate for the height field in massive binary search trees?

Observations

Due to the balancing methods

- AVL Trees requiere to have a $O(\log_2 n)$ rotations.
- AVL Trees operations cost $O(\log_2 n)$.

There is a problem about the height

How much memory it is necessary to allocate for the height field in massive binary search trees?

Furthermore

Arguments in favor of AVL Trees

- Search is $O(\log N)$ since AVL trees are always balanced.
- Insertion and deletions are also $O(\log N)$.

Furthermore

Arguments in favor of AVL Trees

- Search is $O(\log N)$ since AVL trees are always balanced.
- Insertion and deletions are also $O(\log N)$.

Arguments against using AVL trees

- Difficult to program and debug.
- The dynamic space nature of the balancing (Height) factor.
- Asymptotically faster but re-balancing costs time.
- Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

Furthermore

Arguments in favor of AVL Trees

- Search is $O(\log N)$ since AVL trees are always balanced.
- Insertion and deletions are also $O(\log N)$.

Arguments against using AVL trees

- Difficult to program and debug.
- The dynamic space nature of the balancing (Height) factor.
- Asymptotically faster but re-balancing costs time.
- Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

Furthermore

Arguments in favor of AVL Trees

- Search is $O(\log N)$ since AVL trees are always balanced.
- Insertion and deletions are also $O(\log N)$.

Arguments against using AVL trees

- Difficult to program and debug.
- The dynamic space nature of the balancing (Height) factor.
- Asymptotically faster but re-balancing costs time.
- Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

Furthermore

Arguments in favor of AVL Trees

- Search is $O(\log N)$ since AVL trees are always balanced.
- Insertion and deletions are also $O(\log N)$.

Arguments against using AVL trees

- Difficult to program and debug.
- The dynamic space nature of the balancing (Height) factor.
- Asymptotically faster but re-balancing costs time.
- Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

Furthermore

Arguments in favor of AVL Trees

- Search is $O(\log N)$ since AVL trees are always balanced.
- Insertion and deletions are also $O(\log N)$.

Arguments against using AVL trees

- Difficult to program and debug.
- The dynamic space nature of the balancing (Height) factor.
- Asymptotically faster but re-balancing costs time.
- Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

Outline

Definitions

Definition

A Red Black Tree is a Binary Search Tree where each node has an extra field, its color.

Definitions

Definition

A Red Black Tree is a Binary Search Tree where each node has an extra field, its color.

Properties

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

Definitions

Definition

A Red Black Tree is a Binary Search Tree where each node has an extra field, its color.

Properties

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

Definitions

Definition

A Red Black Tree is a Binary Search Tree where each node has an extra field, its color.

Properties

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

Definitions

Definition

A Red Black Tree is a Binary Search Tree where each node has an extra field, its color.

Properties

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

Definitions

Definition

A Red Black Tree is a Binary Search Tree where each node has an extra field, its color.

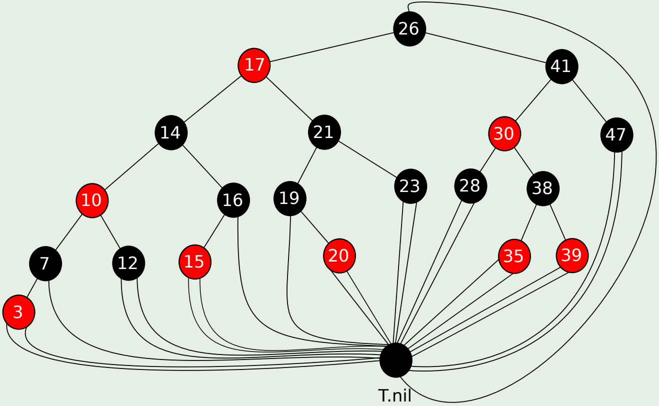
Properties

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

Outline

Red-Black Trees

Example



Height on a Red Black Tree

Black Height $bh(x)$

We call the number of black nodes on any path from, but not including, a node x down to a leaf the black height of the node.

Outline

Lemma for the height of Red-Black Trees

Theorem

A Red-Black Trees with n internal nodes has height at most $2\log(n+1)$.

Lemma for the height of Red-Black Trees

Theorem

A Red-Black Trees with n internal nodes has height at most $2\log(n+1)$.

Proof: Step 1

- Prove that any subtree rooted at x **contains at least** $2^{bh(x)} - 1$ internal nodes.
- If $bh(x) = 0$, then

Lemma for the height of Red-Black Trees

Theorem

A Red-Black Trees with n internal nodes has height at most $2\log(n+1)$.

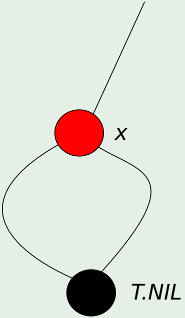
Proof: Step 1

- Prove that any subtree rooted at x **contains at least** $2^{bh(x)}-1$ internal nodes.
- If $bh(x)=0$, then

Outline

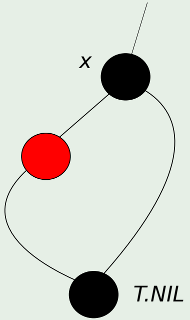
Examples for $bh(x) = 0$

Case I



Examples for $bh(x) = 0$

Case II - There are other, but they are similar



Then

Then, we have

- Thus $2^{bh(x)} - 1 = 2^0 - 1 = 0$.
- Now with $bh(x) > 0$, we have that child[x] has height $bh(x)$ or $bh(x) - 1$.

Then

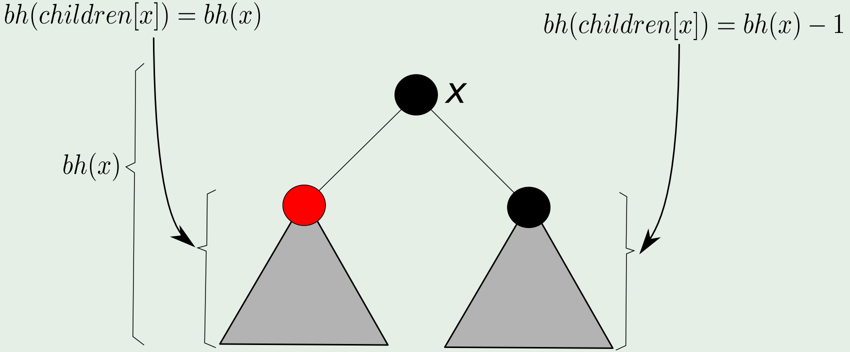
Then, we have

- Thus $2^{bh(x)} - 1 = 2^0 - 1 = 0$.
- Now with $bh(x) > 0$, we have that child[x] has height $bh(x)$ or $bh(x) - 1$.

Outline

Thus, we have the following

Example



Proof

Case I

- Height of child is $bh(x)$.
- Then, the child is red, if not subtree rooted at x will have height $bh(x)+1$!
- Now, we have two subtrees from the child with red root...(The children are black)

Proof

Case I

- Height of child is $bh(x)$.
- Then, the child is red, if not subtree rooted at x will have height $bh(x) + 1$!
- Now, we have two subtrees from the child with red root...(The children are black)

Thus, we have

- Thus, each of this subtrees has height $bh(x) - 1 \Rightarrow$ each tree contains at least $2^{bh(x)-1} - 1$ nodes by inductive hypothesis.
- Then the child contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1$ internal nodes.
- Finally, the tree rooted contains $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1$ (One for the node rooted at the child node)

Proof

Case I

- Height of child is $bh(x)$.
- Then, the child is red, if not subtree rooted at x will have height $bh(x) + 1$!
- Now, we have two subtrees from the child with red root...(The children are black)

Thus, we have

- Thus, each of this subtrees has height $bh(x) - 1 \Rightarrow$ each tree contains at least $2^{bh(x)-1} - 1$ nodes by inductive hypothesis.
- Then the child contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1$ internal nodes.
- Finally, the tree rooted contains $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1$ (One for the node rooted at the child node)

Proof

Case I

- Height of child is $bh(x)$.
- Then, the child is red, if not subtree rooted at x will have height $bh(x)+1$!
- Now, we have two subtrees from the child with red root...(The children are black)

Thus, we have

- Thus, each of this subtrees has height $bh(x)-1 \Rightarrow$ each tree contains at least $2^{bh(x)-1}-1$ nodes by inductive hypothesis.
- Then the child contains at least $2^{bh(x)-1}-1+2^{bh(x)-1}-1$ internal nodes.
- Finally, the tree rooted contains $2^{bh(x)-1}-1+2^{bh(x)-1}-1+1$ (One for the node rooted at the child node)

Proof

Case I

- Height of child is $bh(x)$.
- Then, the child is red, if not subtree rooted at x will have height $bh(x)+1$!
- Now, we have two subtrees from the child with red root...(The children are black)

Thus, we have

- Thus, each of this subtrees has height $bh(x)-1 \Rightarrow$ each tree contains at least $2^{bh(x)-1}-1$ nodes by inductive hypothesis.
- Then the child contains at least $2^{bh(x)-1}-1+2^{bh(x)-1}-1$ internal nodes.
- Finally, the tree rooted contains $2^{bh(x)-1}-1+2^{bh(x)-1}-1+1$ (One for the node rooted at the child node)

Proof

Case I

- Height of child is $bh(x)$.
- Then, the child is red, if not subtree rooted at x will have height $bh(x) + 1$!
- Now, we have two subtrees from the child with red root...(The children are black)

Thus, we have

- Thus, each of this subtrees has height $bh(x) - 1 \Rightarrow$ each tree contains at least $2^{bh(x)-1} - 1$ nodes by inductive hypothesis.
- Then the child contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1$ internal nodes.
- Finally, the tree rooted contains $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1$ (One for the node rooted at the child node)

Proof

Finally

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Proof

Finally

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Case II

- Height of the child is $bh(x) - 1$.
- Now, we have two subtrees with at least $2^{bh(x)-1} - 1$ nodes by inductive hypothesis.
- Thus, we have that the tree with root x contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1$ internal nodes.

Proof

Finally

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Case II

- Height of the child is $bh(x) - 1$.
- Now, we have two subtrees with at least $2^{bh(x)-1} - 1$ nodes by inductive hypothesis.
- Thus, we have that the tree with root x contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1$ internal nodes.

Proof

Finally

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Case II

- Height of the child is $bh(x) - 1$.
- Now, we have two subtrees with at least $2^{bh(x)-1} - 1$ nodes by inductive hypothesis.
- Thus, we have that the tree with root x contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1$ internal nodes.

Proof

Again

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Proof

Again

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Finally

- Now given that h is the height of the tree, we have by property 4 that $bh(T) \geq \frac{h}{2}$.
- Then $n \geq 2^{bh(root)} - 1 \geq 2^{\frac{h}{2}} - 1$.
- Then, $h \leq 2\log(n+1)$ $\square$.

Proof

Again

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Finally

- Now given that h is the height of the tree, we have by property 4 that $bh(T) \geq \frac{h}{2}$.
- Then $n \geq 2^{bh(root)} - 1 \geq 2^{\frac{h}{2}} - 1$.
- Then, $h \leq 2\log(n+1)$ $\square$.

Proof

Again

- Then, the tree with root x contains at least $2 \times 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$

Finally

- Now given that h is the height of the tree, we have by property 4 that $bh(T) \geq \frac{h}{2}$.
- Then $n \geq 2^{bh(root)} - 1 \geq 2^{\frac{h}{2}} - 1$.
- Then, $h \leq 2\log(n+1)$ $\square$.

cinvestav

Lemma for the height of Red-Black Trees

Corollary

From the previous theorem we conclude that SEARCH, MINIMUM, etcetera, can be implemented in $O(\log n)$.

Outline

Rotations in Red-Black Trees

Purpose

Rotations are used to maintain the structure of the Red-Black Trees.

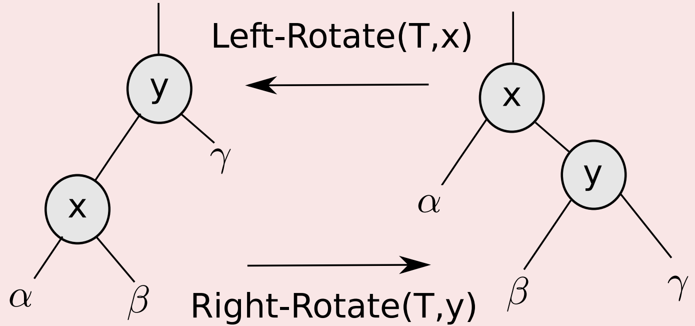
Rotations in Red-Black Trees

Purpose

Rotations are used to maintain the structure of the Red-Black Trees.

Types of rotations

There are left and right rotations, they are inverse to each other.



Example of Rotations in Red-Black Trees

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
```

Example of Rotations in Red-Black Trees

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

Example of Rotations in Red-Black Trees

LEFT-ROTATE(T,x)

1. y = x.right ▷ Set y
2. x.right = y.left ▷ Turn y's left subtree into x's right subtree
3. if y.left ≠ T.nil
4.   y.left.p = x
5. y.p = x.p ▷ Link x's parent to y
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 ▷ Put x on y's left
12. x.p = y

Example of Rotations in Red-Black Trees

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
```

Example of Rotations in Red-Black Trees

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
```

Example of Rotations in Red-Black Trees

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
```

Example of Rotations in Red-Black Trees

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
```

Example of Rotations in Red-Black Trees

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
```

Example of Rotations in Red-Black Trees

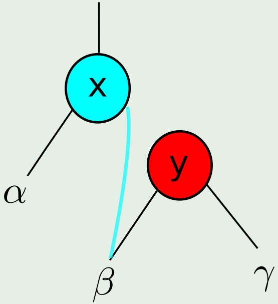
LEFT-ROTATE(T,x)

```
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
```

Example Left-Rotate

Step 1 - the correct right child is changed

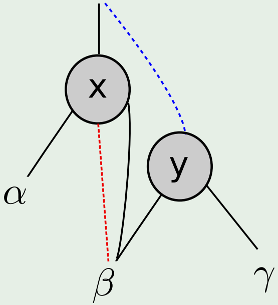
1. y = x.right ▷ Set y
2. x.right = y.left ▷ Turn y's left subtree into x's right subtree



Example Left-Rotate

Step 2 - the parents are set correctly

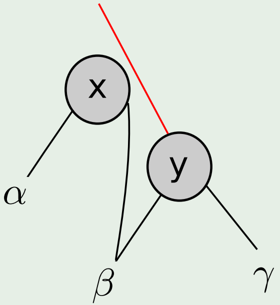
3. if y.left $\neq$ T.nil
4.   y.left.p $=$ x
5. y.p $=$ x.p



Example Left-Rotate

Step 3

```
if x.p == T.nil          ▷Set y to be the root if x was it
      T.root = y
elseif x == x.p.left
      x.p.left = y
else x.p.right = y
```

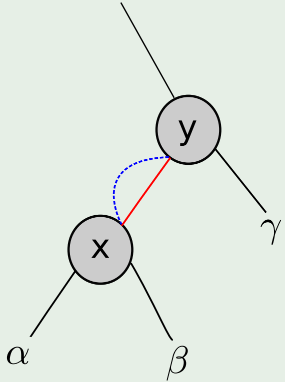


Example Left-Rotate

Step 4

11. y.left = x ▷ Put x on y's left

12. x.p = y



Outline

First than anything

Something Notable

You still have a Binary Search Tree!!!

There

How do you do insertion in a Binary Search Tree?

First than anything

Something Notable

You still have a Binary Search Tree!!!

There

How do you do insertion in a Binary Search Tree?

Outline

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

First

Search Variables being Initialized.

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

Second

Binary search for insertion.

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

Third

Change parent of the node to be inserted.

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

Fourth

Test to see if the Tree is empty!!!

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

Fifth

Insert node z in the correct left or right child:

- if z.key $<$ y.key $\Rightarrow$ y.left $=$ z
- if z.key $\geq$ y.key $\Rightarrow$ y.right $=$ z

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

Sixth

Make z's leafs equal to T.nil.

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

Seventh

Make z's color equal to **RED**.

Insertion Code in Red-Black Trees

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 = T.RED
RB-Insert-Fixup(T.z)
```

Eight

Call RB-Insert-Fixup!!!

Outline

RB-Insert-Fixup

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
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else (“right” and “left” exchanged)
T.root.color = BLACK
```

Case 1

- z’s uncle is RED
  - Change of parent and uncle’s color to BLACK
  - Move problem to z’s grandfather

Outline

Outline

Prior to the first iteration of the loop

We start with a red-black tree with no violations

- Then, the algorithm insert the red node z at the bottom of the Red-Black Trees.
  - This Tree does not violate properties 1,3 and 5.

Prior to the first iteration of the loop

We start with a red-black tree with no violations

- Then, the algorithm insert the red node z at the bottom of the Red-Black Trees.
  - This Tree does not violate properties 1,3 and 5.

Properties

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

Important

If $z.p$ is the root

- Then, $z.p$ began as a black node no change happened when Fix-Up is Called

Then

Fix-Up is Called

- The RB-Insert-Fixup is called because the following possible violations.

Case I

- If z is the first node to be inserted, you violate the property 2.
  - It is the only violation on the entire Red-Black Trees.
  - Because the parent and both children of z are the sentinel.

Then

Fix-Up is Called

- The RB-Insert-Fixup is called because the following possible violations.

Case I

- If z is the first node to be inserted, you violate the property 2.
  - It is the only violation on the entire Red-Black Trees.
  - Because the parent and both children of´ are the sentinel.

Now

Case II

- If the tree violates property 4

z was inserted after a red node

- Thus, z and $z.p$ are red
  - The Tree does not violate any other property

Now

Case II

- If the tree violates property 4

z was inserted after a red node

- Thus, z and $z.p$ are red
  - The Tree does not violate any other property

Outline

Some Notes

Something Notable

- We need to consider six cases

However

- but three of them are symmetric to the other three
  - Depending if $z.p$ to be a left or right child of $z.p.p$ given
    1. **if z.p == z.p.p.left**

Some Notes

Something Notable

- We need to consider six cases

However

- but three of them are symmetric to the other three
  - Depending if $z.p$ to be a left or right child of $z.p.p$ given
    1. **if z.p == z.p.p.left**

Therefore

If $z.p$ is red

- We enter the loop of Fix-Up code...

Which tells us that

- If $z.p$ is red, $z.p$ cannot be the root $\Rightarrow$ $z.p.p$ exists

Therefore

If $z.p$ is red

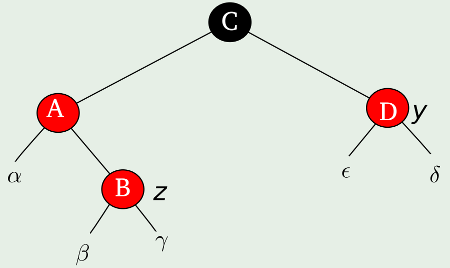
- We enter the loop of Fix-Up code...

Which tells us that

- If $z.p$ is red, $z.p$ cannot be the root $\Rightarrow$ $z.p.p$ exists

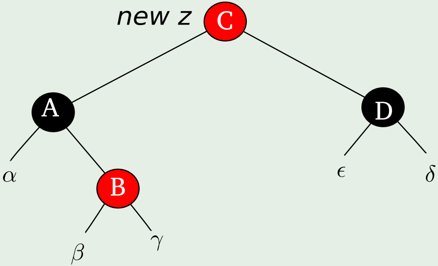
Maintenance in Insertion in Red-Black Trees

Case I - z's uncle is red



Maintenance in Insertion in Red-Black Trees

Recolor parent and uncle to black



Thus

Observations

1. Recoloring will fix the z.parent.color $==$ red and keeps the bh property
    1. It will move the problem upwards.
2. Nevertheless, the tree rooted at A, B and D are Red-Black Trees.
3. So the new z will move the problem upward for the next iteration.

Thus

Observations

1. Recoloring will fix the z.parent.color == red and keeps the bh property
   1. It will move the problem upwards.
2. Nevertheless, the tree rooted at A, B and D are Red-Black Trees.
3. So the new z will move the problem upward for the next iteration.

Thus

Observations

1. Recoloring will fix the z.parent.color == red and keeps the bh property
   1. It will move the problem upwards.
2. Nevertheless, the tree rooted at A, B and D are Red-Black Trees.
3. So the new z will move the problem upward for the next iteration.

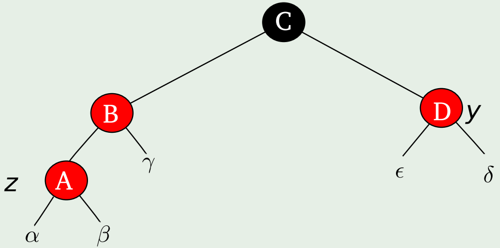
Thus

Observations

1. Recoloring will fix the z.parent.color $==$ red and keeps the bh property
   1. It will move the problem upwards.
2. Nevertheless, the tree rooted at A, B and D are Red-Black Trees.
3. So the new z will move the problem upward for the next iteration.

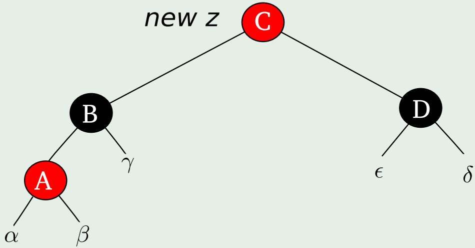
The Symmetric Case

We have



The Symmetric Case

We have



RB-Insert-Fixup

RB-Insert-Fixup(T,z)

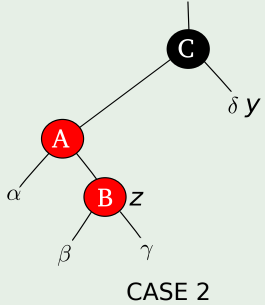
```
while z.p.color == RED
    if z.p == z.p.p.left
        y=z.p.p.right
        if y.color ==RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else ("right" and "left" exchanged)
T.root.color = BLACK
```

Case 2

- if z is in the right child then
  - Move the problem to the parent by making z = z.p
  - Rotate left using z as the rotation node

Insertion in Red-Black Trees

Case 2 - z's uncle y is black and z is a right child



Thus

First

Here simple recoloring will not work.

Rotate

We rotate first from to left using z.p.

This moves case 2 toward case 3

To get ready for the final fix-up.

Thus

First

Here simple recoloring will not work.

Rotate

We rotate first from to left using z.p.

This moves case 2 toward case 3

To get ready for the final fix-up.

Thus

First

Here simple recoloring will not work.

Rotate

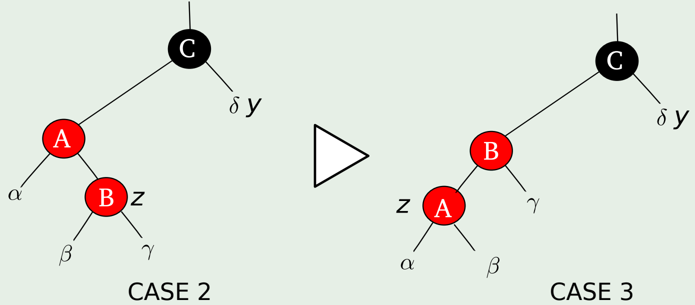
We rotate first from to left using z.p.

This moves case 2 toward case 3

To get ready for the final fix-up.

From Case 2 to Case 3

Example



RB-Insert-Fixup

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
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else ("right" and "left" exchanged)
T.root.color = BLACK
```

Case 3

- if z is in the left child then
    - Recolor z's parent to BLACK
    - recolor z's grandparent to RED
    - Rotate right using the grandparent

Then

We do the following

Then, we recolor B.color = **BLACK**, C.color = **RED** (No problem γ and δ are black nodes)

Finally

Then, you rotate right using $z.p.p$ to fix the black height property!!

Then

We do the following

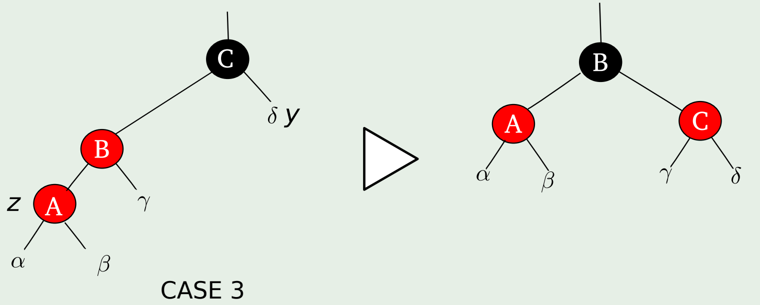
Then, we recolor B.color $=$ **BLACK**, C.color $=$ **RED** (No problem γ and δ are black nodes)

Finally

Then, you rotate right using $z.p.p$ to fix the black height property!!

From Case 3 to final recoloring

Case 3 - It allows to fix the bh locally



Outline

When the loop terminates

it does so because $z.p$ is black (Sentinel or not)

- The Tree does not violate property 4 at loop termination.
  - If a node is red, then both its children are black.

By the loop invariant

- The only property that might fail to hold is property 2
  - **The root is black.**

When the loop terminates

it does so because $z.p$ is black (Sentinel or not)

- The Tree does not violate property 4 at loop termination.
  - If a node is red, then both its children are black.

By the loop invariant

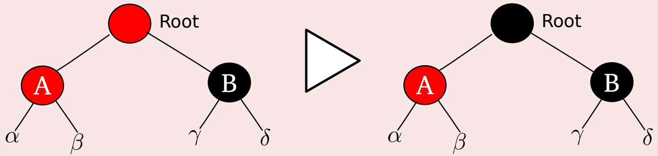
- The only property that might fail to hold is property 2
  - **The root is black.**

Then

Root Recoloring

After pushing the problem up to the root by the while loop color the root to **BLACK**!!!

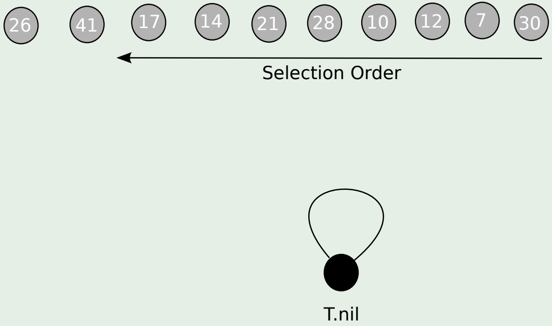
We have then



Outline

Example: Insertion in Red-Black Trees

Example Tree is empty

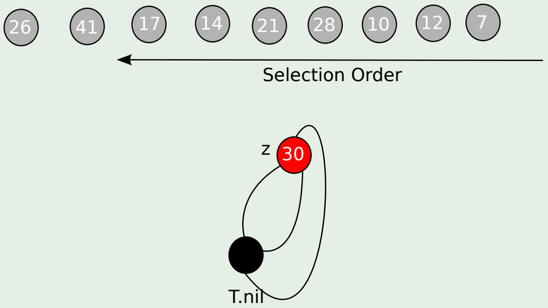


Example: Insertion in Red-Black Trees

The root becomes red!!!

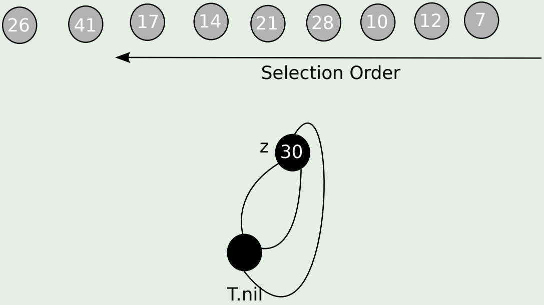
STEPS

1. $y = T.nil$
2. $x = T.root$
3. We never go into the search loop
4. $z.p = y$
5. if $y == T.nil$
6. $\quad T.root = z$
7. $z.left = T.nil$
8. $z.right = T.nil$
9. $z.color = T.RED$



Example: Insertion in Red-Black Trees

Fix the problem



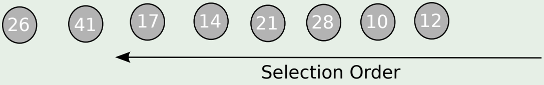
ONLY STEP

1. T.root.color=BLACK

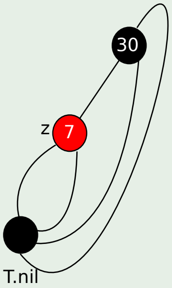
It happens at the end of the Fix-up

Example: Insertion in Red-Black Trees

Insert 7

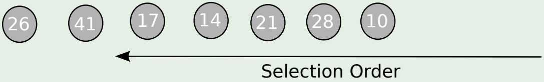


1. Do a binary search to find a place to insert
2. Parent of z is not RED $\Rightarrow$ no fix-up

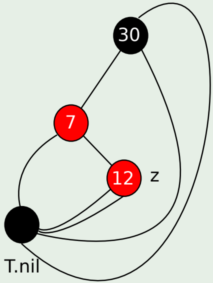


Example: Insertion in Red-Black Trees

Insert 12

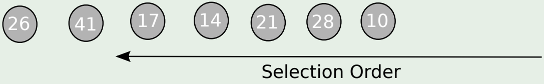


1. Do a binary search to find a place to insert
2. Parent of z is RED!!!



Example: Insertion in Red-Black Trees

Case 2 into Case 3

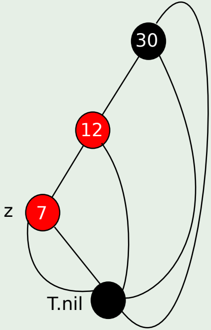


1. Re-balance first to the left

   if z==z.p.right

   z=z.p

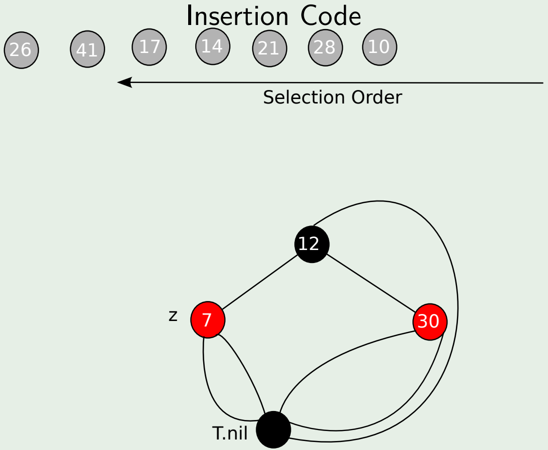
   Left-Rotate(T, z)



Example: Insertion in Red-Black Trees

Case 3

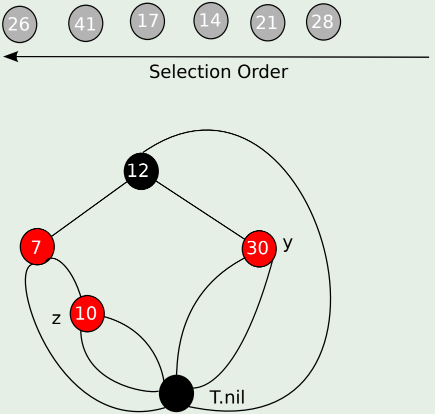
1. Re-color and re-balance to the right
   z.p.color = BLACK
   z.p.p.color=RED
   Right-Rotate(T, z.p.p)
2. No problem to fix
3. The root is already BLACK so nothing happens



Example: InsertionInsertion Code in Red-Black Trees

Insert 10

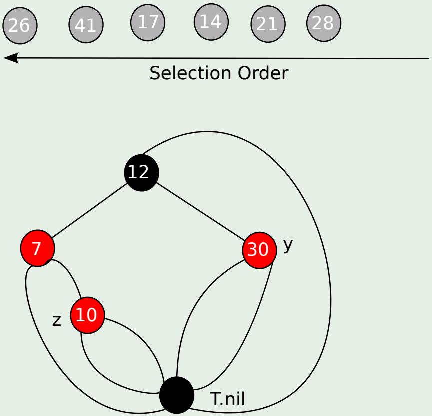
1. We insert 10
2. We color it to RED
3. We need to fix the problem



Example: Insertion in Red-Black Trees

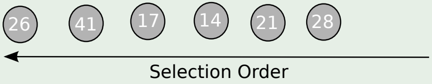
Case 1

1. if z.p == z.p.p.left
2. y=z.p.p.right
3. if y.color ==RED

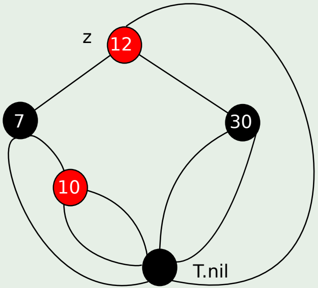


Example: Insertion in Red-Black Trees

Case 1



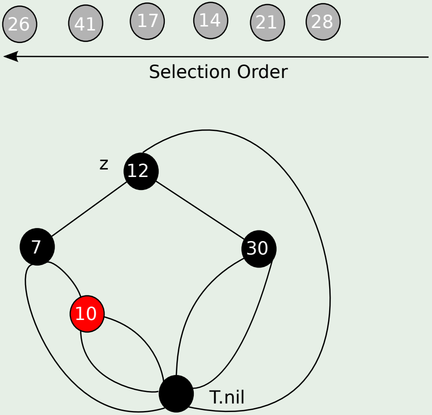
1. if y.color ==RED
2. z.p.color = BLACK
3. y.color = BLACK
4. z.p.p.color = RED
5. z = z.p.p



Example: Insertion in Red-Black Trees

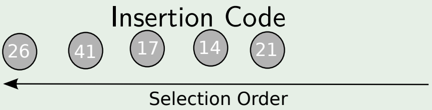
Case 1 You get out of the loop and...

1. T.root.color = BLACK

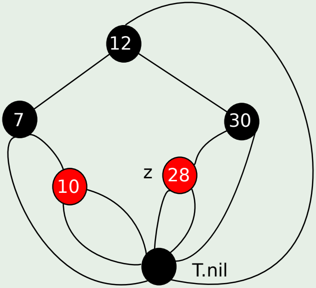


Example: Insertion in Red-Black Trees

Insert 28

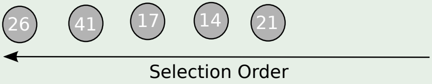


1. We insert 28
2. We color it to RED
3. No problem to fix...

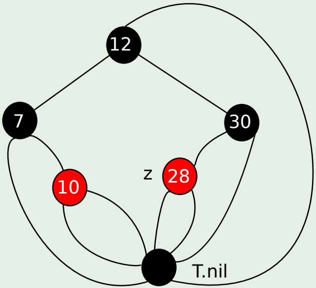


Example: Insertion in Red-Black Trees

Insert 28



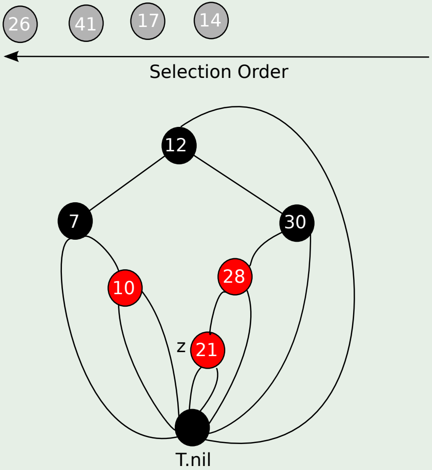
1. We insert 28
2. We color it to RED
3. No problem to fix...



Example: Insertion in Red-Black Trees

Insert 21

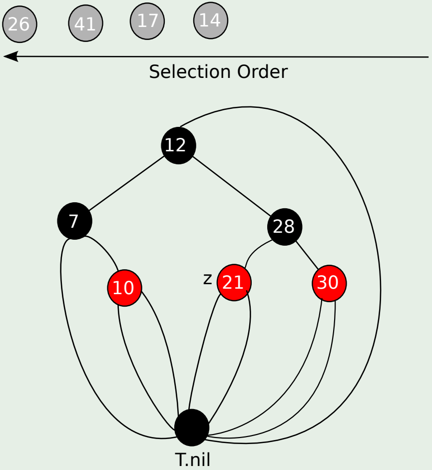
1. We insert 21
2. We color it to RED
3. We need to fix...



Example: Insertion in Red-Black Trees

Case 3

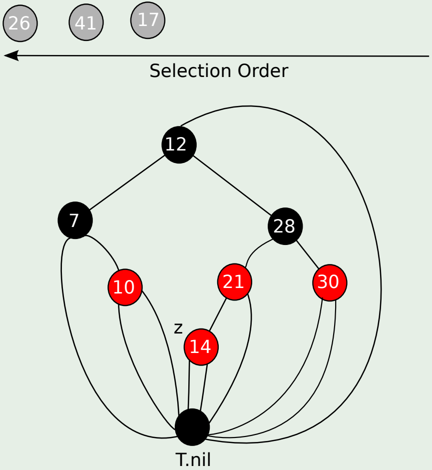
1. Re-color and
   re-balance to the right
   z.p.color = BLACK
   z.p.p.color=RED
   Right-Rotate(T, z.p.p)
2. No problem to fix
3. The root is already BLACK so nothing happens



Example: Insertion in Red-Black Trees

Insert 14

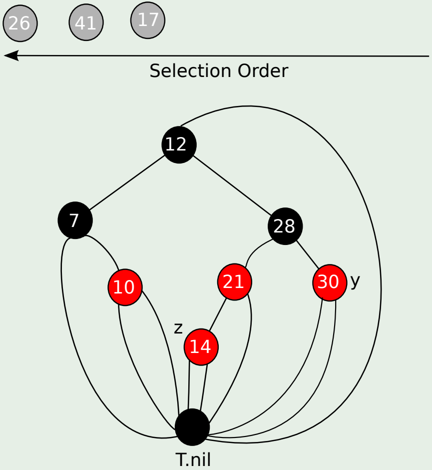
1. We insert 14
2. We color it to RED
3. We need to fix...



Example: Insertion in Red-Black Trees

Case 1

1. if z.p == z.p.p.left
2. y=z.p.p.right
3. if y.color ==RED

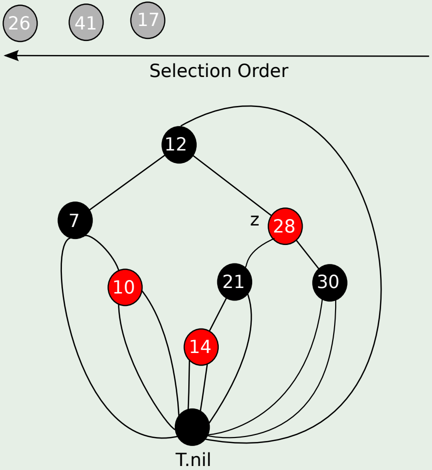


Example: Insertion in Red-Black Trees

Case 1

1. if y.color ==RED
2. z.p.color = BLACK
3. y.color = BLACK
4. z.p.p.color = RED
5. z = z.p.p

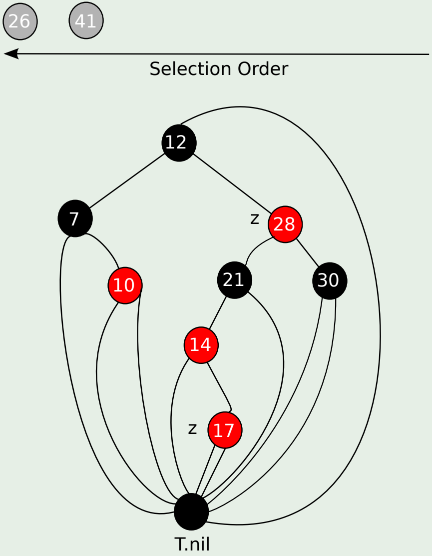
Nothing to do after!!!

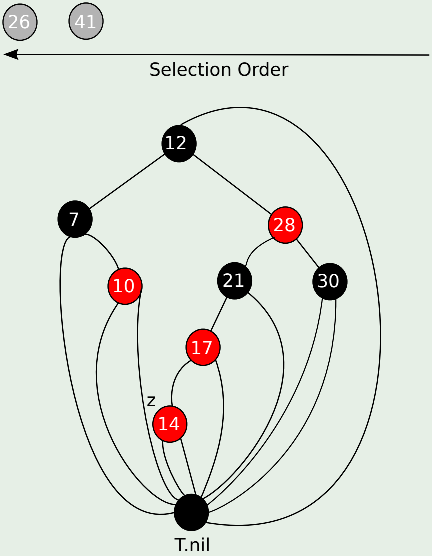


Example: Insertion in Red-Black Trees

Insert 17

1. Do a binary search to find a place to insert
2. Parent of z is RED!!!





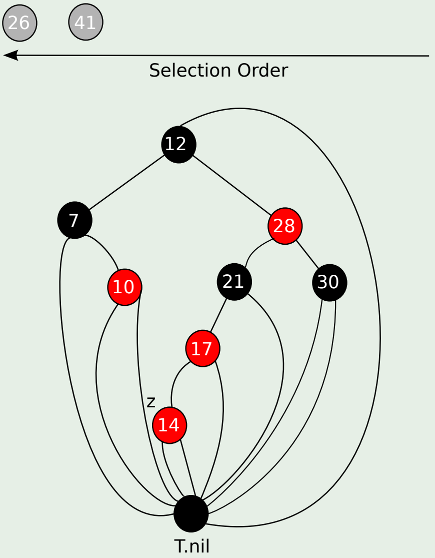
1. Re-balance first to the left
   if z==z.p.right

       z=z.p

       Left-Rotate(T, z)

Example: Insertion in Red-Black Trees

Case 3

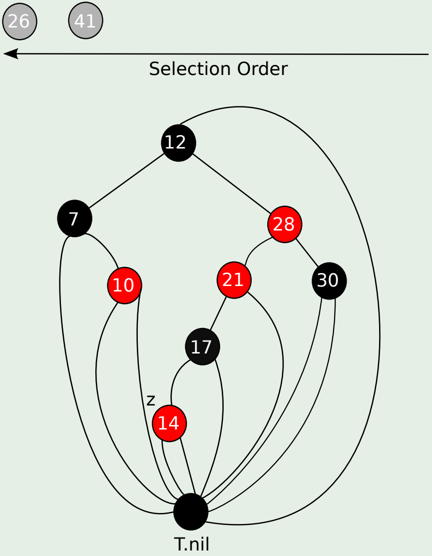


1. Re-color and re-balance to the right
   z.p.color = BLACK
   z.p.p.color=RED
   Right-Rotate(T, z.p.p)
2. No problem to fix
3. The root is already BLACK so nothing happens

Example: Insertion in Red-Black Trees

Case 3

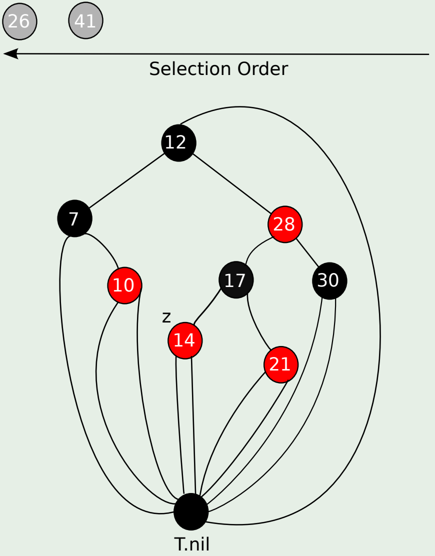
1. Re-color and re-balance to the right
   z.p.color = BLACK
   z.p.p.color=RED
   Right-Rotate(T, z.p.p)
2. No problem to fix
3. The root is already BLACK so nothing happens



Example: Insertion in Red-Black Trees

Case 3

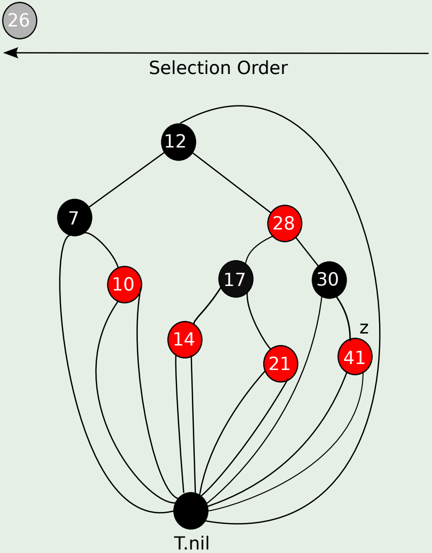
1. Re-color and
   re-balance to the right
   z.p.color = BLACK
   z.p.p.color=RED
   Right-Rotate(T, z.p.p)
2. No problem to fix
3. The root is already BLACK so nothing happens



Example: Insertion in Red-Black Trees

Insert 41

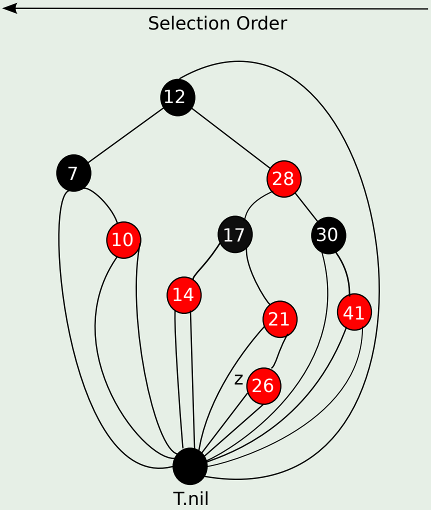
1. Put in the correct node by binary search
2. Because 41's parent is BLACK nothing to fix



Example: Insertion in Red-Black Trees

Case 1 Symmetrical

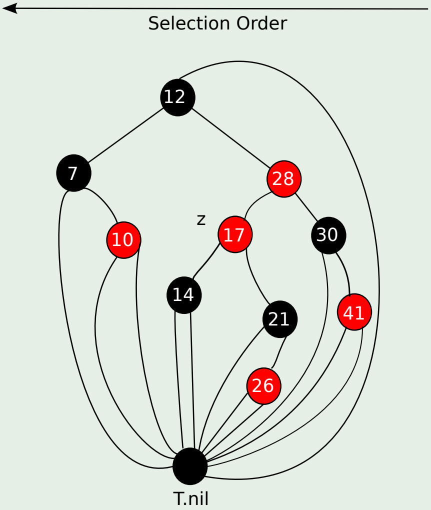
1. if z.p == z.p.p.right
2. y=z.p.p.left
3. if y.color ==RED
4. z.p.color = BLACK
5. y.color = BLACK
6. z.p.p.color = RED
7. z = z.p.p



Example: Insertion in Red-Black Trees

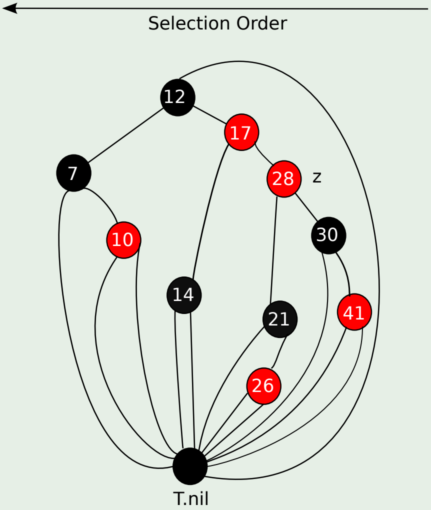
Case 1 Symmetrical

1. if z.p == z.p.p.right
2. y=z.p.p.left
3. if y.color ==RED
4. z.p.color = BLACK
5. y.color = BLACK
6. z.p.p.color = RED
7. z = z.p.p



Example: Insertion in Red-Black Trees

Case 2 to case 3 - Symmetrical

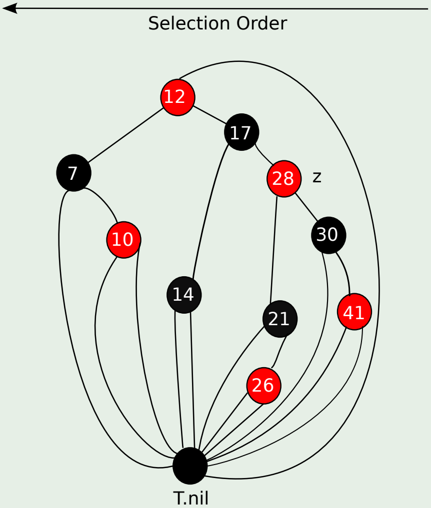


1. Re-balance first to the right
   if z==z.p.left
       z=z.p
       Right-Rotate(T, z)

Example: Insertion in Red-Black Trees

Case 3 - Symmetrical

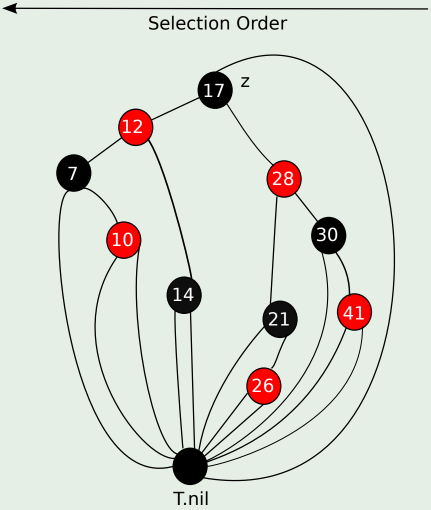
1. Re-color and re-balance to the right
   z.p.color = BLACK
   z.p.p.color=RED
   Left-Rotate(T, z.p.p)
2. No problem to fix
3. The root is already BLACK so nothing happens



Example: Insertion in Red-Black Trees

Case 3 - Symmetrical

1. Re-color and re-balance to the right
   z.p.color = BLACK
   z.p.p.color=RED
   Left-Rotate(T, z.p.p)
2. No problem to fix
3. The root is already BLACK so nothing happens



Complexity of RB-Insert

It is easy to see that we have

$$O(\log n) \tag{1}$$

Outline

The Main Idea

Here, we use the idea of removing

By pushing in its place its successor.

The Main Idea

Here, we use the idea of removing

By pushing in its place its successor.

Therefore we have a problem

- If the successor of a node is the node y.
  - And y is black.
- We have removed a black node from a path.

The Main Idea

Here, we use the idea of removing

By pushing in its place its successor.

Therefore we have a problem

- If the successor of a node is the node y.
  - And y is black.
- We have removed a black node from a path.

The Main Idea

Here, we use the idea of removing

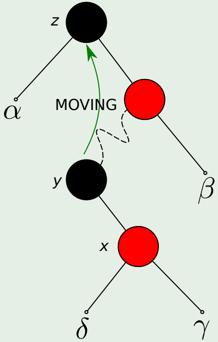
By pushing in its place its successor.

Therefore we have a problem

- If the successor of a node is the node y.
  - And y is black.
- We have removed a black node from a path.

Thus

Example



Outline

Deletion in Red-Black Trees

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
```

Case 1

- Store the info of the node to be deleted

Deletion in Red-Black Trees

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
```

Case 2

- If the left child is empty
- Store the info of the right child
- Move z.right into the position of z

Deletion in Red-Black Trees

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
```

Case 3

- If the right child is empty
- Store the info of the left child
- Move z.left into the position of z

Deletion in Red-Black Trees

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
```

Case 4

- Find the successor of z
- Store the info of it: Color and right child

Deletion in Red-Black Trees

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
```

Case 5

- If parent of succesor is z then set parent of x to y

Deletion in Red-Black Trees

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
```

ø

Case 6

- Substitute y with y.right
- set y.right with z.right
- set the parent of y.right to y

Deletion in Red-Black Trees

RB-DELETE(T,z)

```
        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)
```

Case 7

- Substitute z with y
- Make y.left to z.left
- Make the parent of y.left to y
- Make the color of y to the color of z

Deletion in Red-Black Trees

RB-DELETE(T,z)

```
        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)
```

Case 8

- If y-original-color == BLACK then call RB-Delete-Fixup(T,x)
- After all x points to the node that:
  - It is moved into the position of y.
  - Where y was moved into the position of z.

Where RB-Transplant

RB-Transplant(T, u, v)

1. **if** $u.p == T.nil$
2. $\quad T.root = v$
3. **elseif** $u == u.p.left$
4. $\quad u.p.left = v$
5. **else** $u.p.right = v$
6. $v.p = u.p$

Outline

A virtual node y to be removed or moved around

Case 1

- In line 1, y is removed when it points to z and has less than two children.

A virtual node y to be removed or moved around

Case 1

- In line 1, y is removed when it points to z and has less than two children.

Case 2

- In line 9, y is moved around when z has two children
  - Because y=Tree-Minimum(z.right).

A virtual node y to be removed or moved around

Case 1

- In line 1, y is removed when it points to z and has less than two children.

Case 2

- In line 9, y is moved around when z has two children
  - Because y=Tree-Minimum(z.right).

Then

y will move to z's position.

A virtual node y to be removed or moved around

Case 1

- In line 1, y is removed when it points to z and has less than two children.

Case 2

- In line 9, y is moved around when z has two children
  - Because y=Tree-Minimum(z.right).

Then

y will move to z's position.

A virtual node y to be removed or moved around

Now, the color of y's can change

- Therefore, it gets stored in y-original-color (Lines 2, 10).

- This can produce a violation

A virtual node y to be removed or moved around

Now, the color of y's can change

- Therefore, it gets stored in y-original-color (Lines 2, 10).

Then, when z has two children

- Then y moves to z's position and y gets the same color than z.
- This can produce a violation.

A virtual node y to be removed or moved around

Now, the color of y's can change

- Therefore, it gets stored in y-original-color (Lines 2, 10).

Then, when z has two children

- Then y moves to z's position and y gets the same color than z.
- This can produce a violation.

The node x gets position y

In lines 4, 7, and 11, x is set to point to

- to y's only child or
- T.nil

The node x gets position y

In lines 4, 7, and 11, x is set to point to

- to y's only child or
- T.nil

Since x is going to move to y's original position

The x.p is pointed to y's parent.

- Observe that when RB-Transplant is called in lines 5, 8, or 14, the second parameter passed is the same as x.

The node x gets position y

In lines 4, 7, and 11, x is set to point to

- to y's only child or
- T.nil

Since x is going to move to y's original position

The x.p is pointed to y's parent.

Unless z is y's original parent

The assignment of x.p takes place in line 6 of RB-Transplant.

- Observe that when RB-Transplant is called in lines 5, 8, or 14, the second parameter passed is the same as x.

The node x gets position y

In lines 4, 7, and 11, x is set to point to

- to y's only child or
- T.nil

Since x is going to move to y's original position

The x.p is pointed to y's parent.

Unless z is y's original parent

The assignment of x.p takes place in line 6 of RB-Transplant.

- Observe that when RB-Transplant is called in lines 5, 8, or 14, the second parameter passed is the same as x.

The node x gets position y

In lines 4, 7, and 11, x is set to point to

- to y's only child or
- T.nil

Since x is going to move to y's original position

The x.p is pointed to y's parent.

Unless z is y's original parent

The assignment of x.p takes place in line 6 of RB-Transplant.

- Observe that when RB-Transplant is called in lines 5, 8, or 14, the second parameter passed is the same as x.

The node x gets position y

if z is not the original y's parent

We do not want x.p to point to it since we are going to remove it.

Then

In line 13 of RB-Delete, x.p is set to point to y.

Finally

y will take the position of z in line 17.

The node x gets position y

if z is not the original y's parent

We do not want x.p to point to it since we are going to remove it.

Then

In line 13 of RB-Delete, x.p is set to point to y.

Finally

y will take the position of z in line 17.

The node x gets position y

if z is not the original y's parent

We do not want x.p to point to it since we are going to remove it.

Then

In line 13 of RB-Delete, x.p is set to point to y.

Finally

y will take the position of z in line 17.

The node x gets position y

Then

If y was originally black after taking the $z.color$ can produce a violation, then RB-Delete-Fixup is called.

This can happen

If y was originally red the Red-Black Trees properties still hold.

The node x gets position y

Then

If y was originally black after taking the $z.color$ can produce a violation, then RB-Delete-Fixup is called.

This can happen

If y was originally red the Red-Black Trees properties still hold.

Deletion in Red-Black Trees

Question

What if we removed a black node?

Explanation

We have three problems

1. If y was a root and a RED child becomes the new root, we have violated property 2.
2. If both x and x.p are RED, then we have violated property 4.
3. Moving y around decreases the black-height on a section of the Red Black Tree.
4. Thus, Property 5 is violated.

Explanation

We have three problems

1. If y was a root and a RED child becomes the new root, we have violated property 2.
2. If both x and x.p are RED, then we have violated property 4.
3. Moving y around decreases the black-height on a section of the Red Black Tree.
4. Thus, Property 5 is violated.

Explanation

We have three problems

1. If y was a root and a RED child becomes the new root, we have violated property 2.
2. If both x and x.p are RED, then we have violated property 4.
3. Moving y around decreases the black-height on a section of the Red Black Tree.
4. Thus, Property 5 is violated.

Explanation

We have three problems

1. If y was a root and a RED child becomes the new root, we have violated property 2.
2. If both x and x.p are RED, then we have violated property 4.
3. Moving y around decreases the black-height on a section of the Red Black Tree.
4. Thus, Property 5 is violated.

Outline

Fix-up

How?

- **This could be fixed assuming that x has an "extra black."**

Fix-up

How?

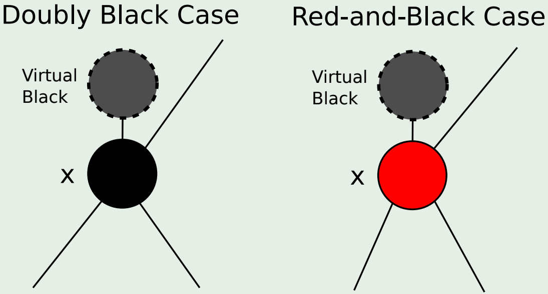
- **This could be fixed assuming that x has an “extra black.”**

Meaning

- **This means that the node is “doubly black” or “red-and-black.”**

Example

We have then



How is this done?

Thus

The procedure RB-DELETE -FIXUP restores properties 2, 4, and 5 by using the while loop to push the extra BLACK node up the tree.

How is this done?

Thus

The procedure RB-DELETE -FIXUP restores properties 2, 4, and 5 by using the while loop to push the extra BLACK node up the tree.

Until

- If we have that x is a **red-and-black**, we simply need to change the color of the node to BLACK (Line 23).

- If we have that x is pointing to the root, remove "extra node"

How is this done?

Thus

The procedure RB-DELETE -FIXUP restores properties 2, 4, and 5 by using the while loop to push the extra BLACK node up the tree.

Until

- If we have that x is a **red-and-black**, we simply need to change the color of the node to BLACK (Line 23).

Then, use Rotations

- Use suitable rotations and re-colorings until x stops to be a doubly black node.
- If we have that x is pointing to the root, remove "extra node."

How is this done?

Thus

The procedure RB-DELETE -FIXUP restores properties 2, 4, and 5 by using the while loop to push the extra BLACK node up the tree.

Until

- If we have that x is a **red-and-black**, we simply need to change the color of the node to BLACK (Line 23).

Then, use Rotations

- Use suitable rotations and re-colorings until x stops to be a doubly black node.
- If we have that x is pointing to the root, remove "extra node."

Outline

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
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                Right-Rotate(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            Left-Rotate(T, x.p)
            x = T.root
    else (same with "right" and "left" exchanged)
x.color = BLACK
```

While loop

- Because a violation on the bh, you need to move x up until the problem is fixed up.

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
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                Right-Rotate(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            Left-Rotate(T, x.p)
            x = T.root
    else (same with "right" and "left" exchanged)
x.color = BLACK
```

Finding who you are

- Find which child are you
- Make w the other child

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
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                Right-Rotate(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            Left-Rotate(T, x.p)
            x = T.root
    else (same with "right" and "left" exchanged)
x.color = BLACK
```

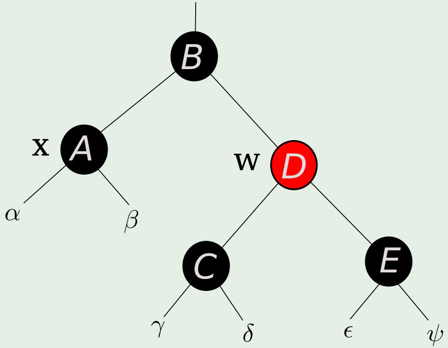
Case 1

- if x is BLACK and w is RED
- Fix the bh problem by making w BLACK, x's parent to RED then rotate left using x's parent
- Make w = x.p.right moving the problem down.

Outline

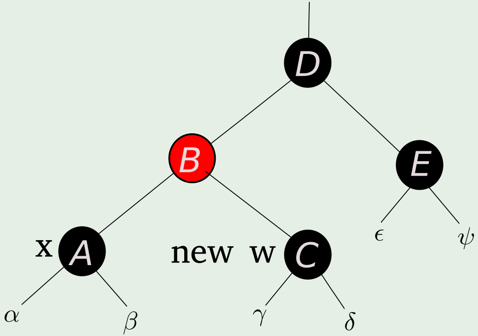
Suitable Rotations and Recoloring

Case 1 - x's sibling w is red. You keep the bh property of the other subtrees



Suitable Rotations and Recoloring

Case 1 - x's sibling w is red. You keep the bh property of the other subtrees



RB-DELETE-FIXUP(T,x)

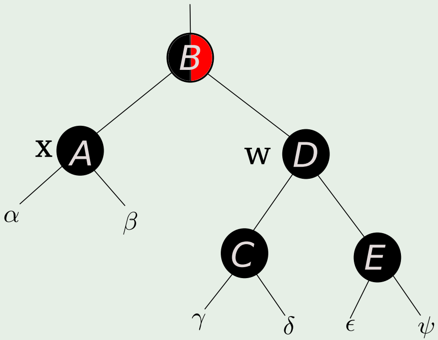
```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK
            x.p.color = RED
            Left-Rotate(T, x.p)
            w = x.p.right
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                Right-Rotate(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            Left-Rotate(T, x.p)
            x = T.root
    else (same with "right" and "left" exchanged)
x.color = BLACK
```

Case 2

- Now if w.left's color and w.right's color is BLACK
- We do something smart decrease the bh height at w by making w's color to RED
- Move the problem fropm to x to x.p (After all the subtree have the same height at x.p)

Suitable Rotations and Recoloring

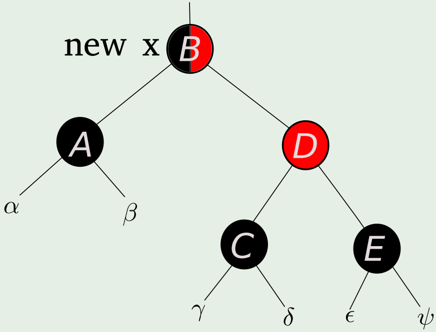
Case 2 - x's sibling w is black, and both of w's children are black.



Note: The Node with half and half colors has the meaning that it can be red or black.

Suitable Rotations and Recoloring

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



RB-DELETE-FIXUP(T,x)

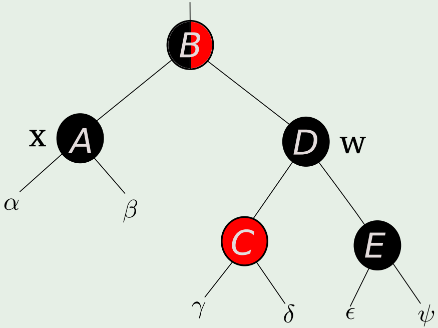
```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK
            x.p.color = RED
            Left-Rotate(T, x.p)
            w = x.p.right
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                Right-Rotate(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            Left-Rotate(T, x.p)
            x = T.root
    else (same with "right" and "left" exchanged)
x.color = BLACK
```

Case 3

- If w.right's color is BLACK
- We do something smart, we re-color and do a right rotation at w
- This does not change the Red-Black Trees properties of w
- but prepare the situation for fixing the x height problem in case 4.

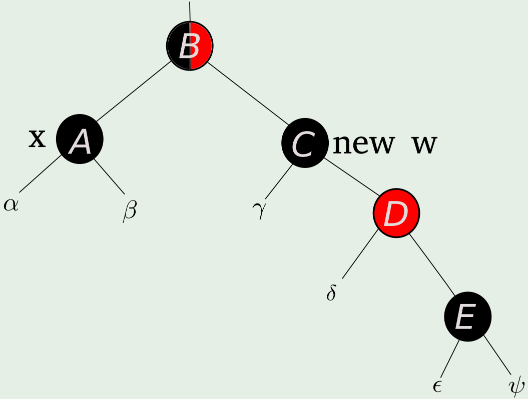
Suitable Rotations and Recoloring

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



Suitable Rotations and Recoloring

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



RB-DELETE-FIXUP(T,x)

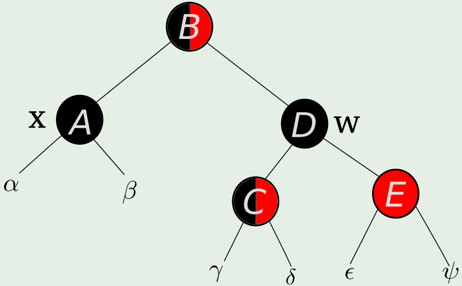
```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK
            x.p.color = RED
            Left-Rotate(T, x.p)
            w = x.p.right
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                Right-Rotate(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            Left-Rotate(T, x.p)
            x = T.root
    else (same with "right" and "left" exchanged)
x.color = BLACK
```

Case 4

- We are ready to fix our problem!!! with respect to x (Case 2 and 3 where a preparation to fix the problem)
- We increase the height of the bh with the problem, x.

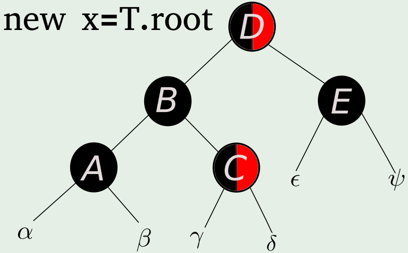
Suitable Rotations and Recoloring

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



Suitable Rotations and Recoloring

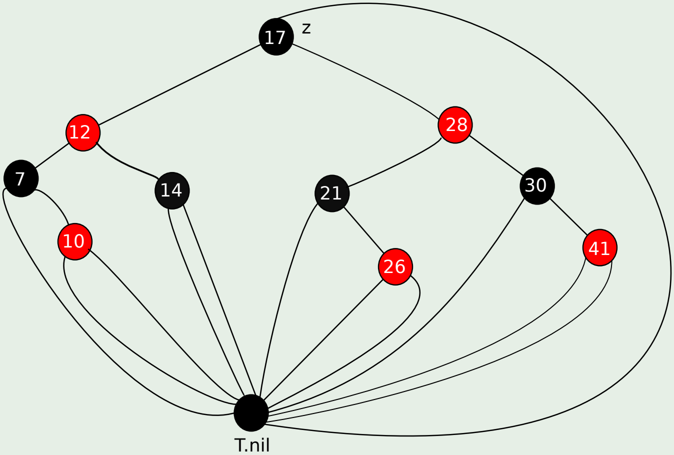
Case 4: x's sibling w is black, and w's right child is red.



Outline

Deletion in Red-Black Trees

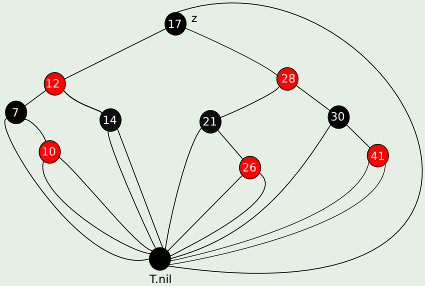
Delete 17



Deletion in Red-Black Trees

Store the info about z

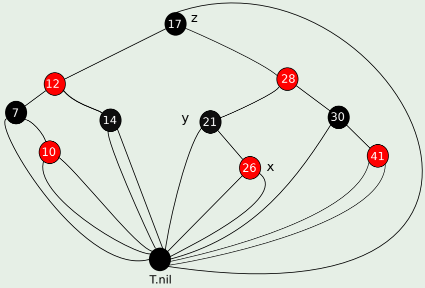
- $y = z$
- y-original-color $=$ y.color



Deletion in Red-Black Trees

None of the children of z are T.nil, thus

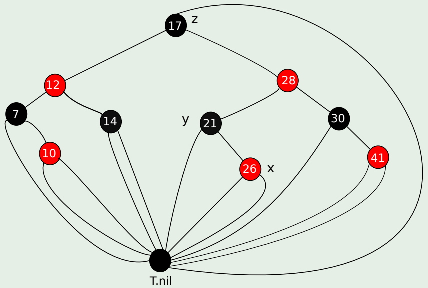
- else y = Tree-Minimum(z.right)
- y-original-color = y.color
- x = y.right



Deletion in Red-Black Trees

We have that y.p$\neq z$

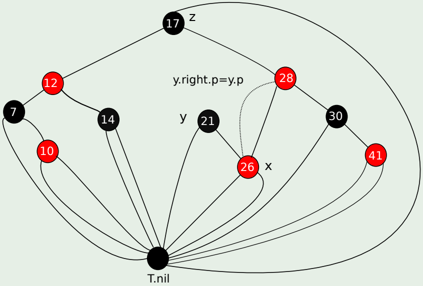
- else RB-Transplant(T,y,y.right)
- y.right $=$ z.right
- y.right.p $=$ y



Deletion in Red-Black Trees

Transplant(T, y, y.right)

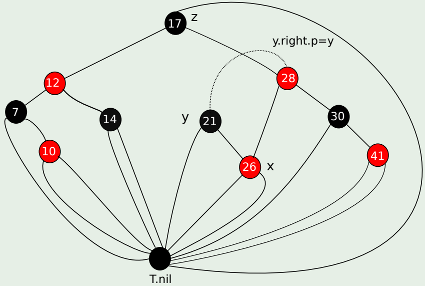
- elseif y == y.p.left
-     y.p.left = y.right
- y.right.p = y.p



Deletion in Red-Black Trees

Now, we move pointers

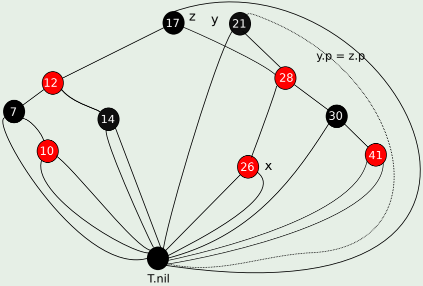
- $y.right = z.right$
- $y.right.p = y$



Deletion in Red-Black Trees

Transplant(z,y)

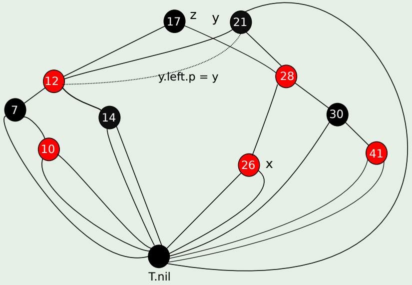
- if z.p == T.nil
- T.root = y
- y.p = z.p



Deletion in Red-Black Trees

Move pointers

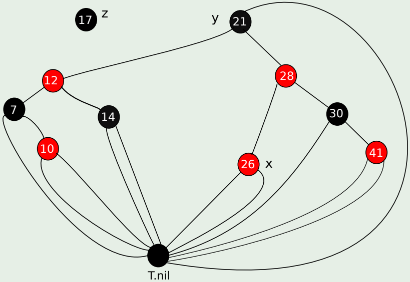
- y.left = z.left
- y.left.p = y
- y.color = z.color



Deletion in Red-Black Trees

We remove z safely and we go into Delete-Fixup(T,x)

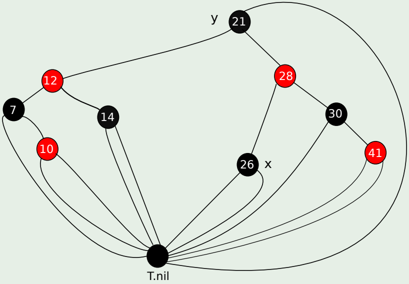
- z.right = NULL
- z.left = NULL
- z.parent = NULL



Deletion in Red-Black Trees

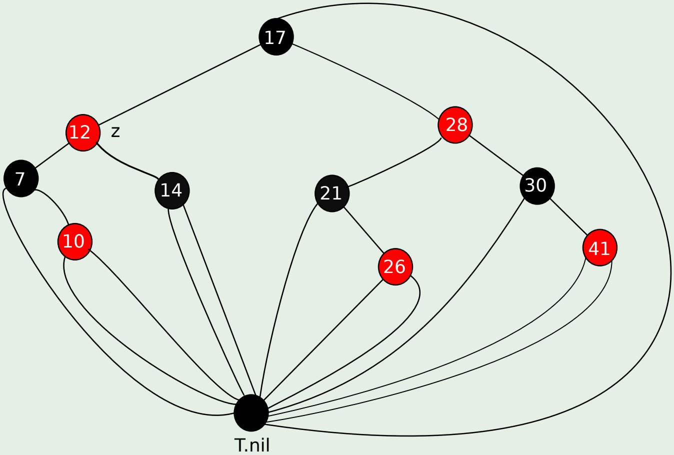
We remove z safely and we go into Delete-Fixup(T,x)

- Never enter into the loop
- Simply do x.color = BLACK



Deletion in Red-Black Trees

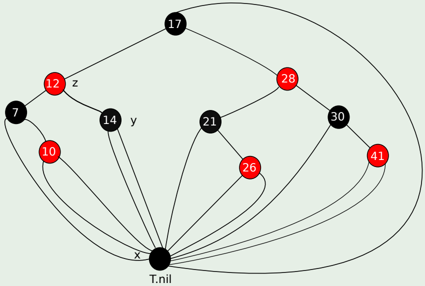
Delete 12



Deletion in Red-Black Trees

None of the children of z are T.nil, thus

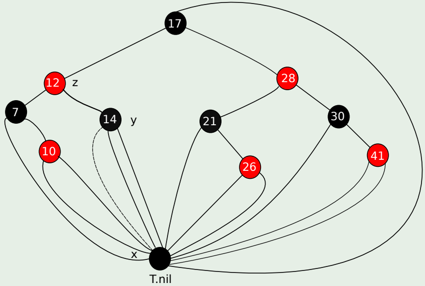
- else y = Tree-Minimum(z.right)
- y-original-color = y.color (BLACK)
- x = y.right (T.NIL)



Deletion in Red-Black Trees

We have that y.p$==z$

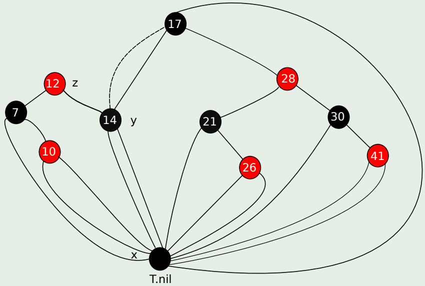
- if y.p == z
- x.p = y



Deletion in Red-Black Trees

Next

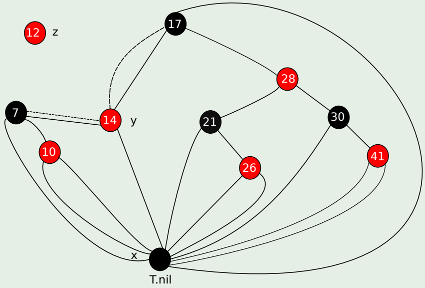
- RB-Transplant(T, z, y)
- y.left = z.left
- y.left.p = y
- y.color = z.color



Deletion in Red-Black Trees

Next

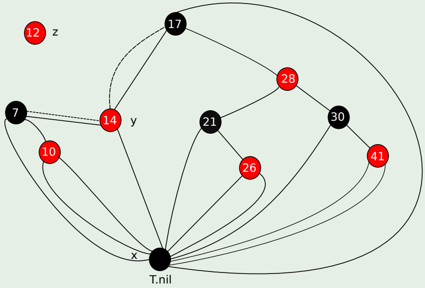
- RB-Transplant(T, z, y)
- y.left = z.left
- y.left.p = y
- y.color = z.color



Deletion in Red-Black Trees

Next

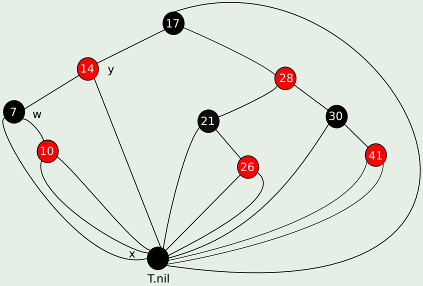
- RB-Transplant(T, z, y)
- y.left = z.left
- y.left.p = y
- y.color = z.color



Deletion in Red-Black Trees

We remove z safely and we go into Delete-Fixup(T,x)

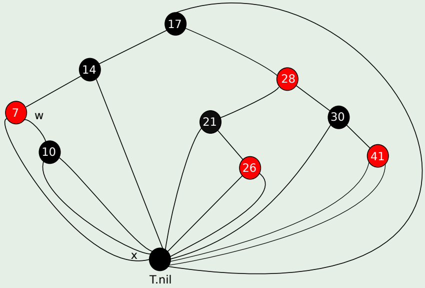
- enter into the loop
- if $x == x.p.right$
- $\quad w = x.p.left$



Deletion in Red-Black Trees

We enter into case 4

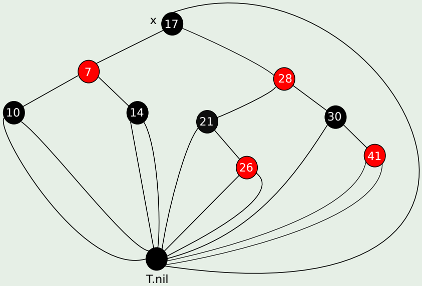
1. $w.color = x.p.color$
2. $x.p.color = BLACK$
3. $w.right.color = BLACK$



Deletion in Red-Black Trees

Rotate and move the x

1. Left-Rotate(T, x.p)
2. $x = T.root$



Applications of Red-Black Trees

Completely Fair Scheduler (CFS)

- It is a task scheduler which was merged into 2.6.23 release of the Linux Kernel.
- It is a replacement of earlier O(1) scheduler.
- CFS algorithm was designed to maintain balance (fairness) in providing processor time to tasks.

Sorting using Parallel Implementations

- Running in $O(\log \log n)$ time

Applications of Red-Black Trees

Completely Fair Scheduler (CFS)

- It is a task scheduler which was merged into 2.6.23 release of the Linux Kernel.
- It is a replacement of earlier O(1) scheduler.
- CFS algorithm was designed to maintain balance (fairness) in providing processor time to tasks.

Sorting using Parallel Implementations

- Running in $O(\log\log n)$ time

Outline

Exercises

From Cormen's book solve

- 13.1-1
- 13.1-3
- 13.1-5
- 13.1-7
- 13.2-2
- 13.2-3
- 13.2-4
- 13.2-5
- 13.3-2
- 13.3-4
- 13.4-2
- 13.4-4