Lecture 5

Instructor: Subrahmanyam Kalyanasundaram

22nd August 2019

Plan

- Last class, we saw INSERT in Red-Black Trees

Plan

- Last class, we saw INSERT in Red-Black Trees
- Today, we see DELETE

Red-Black Trees

RBTs have the following properties:

1. All nodes are colored either Red or Black.
2. The root node and the leaf nodes (NIL) are black.
3. Both children of a red node are black.
   No double red.
4. For any node x, all paths from x to the descendant leaves have the same number of black nodes. = Black height(x)

Red-Black Trees

RBTs have the following properties:

1. All nodes are colored either Red or Black.
2. The root node and the leaf nodes (NIL) are black.
3. Both children of a red node are black.
   No double red.
4. For any node x, all paths from x to the descendant leaves have the same number of black nodes. = Black height(x)

Black height of a red black tree is the black height of its root.

A Red-Black Tree supports all procedures of a BST:

- INSERT(*val*) – Inserts *val* into the RBT rooted at *node*.
- SEARCH(*val*) – Returns True of *val* exists in the BST rooted at *node*. False otherwise.
- SUCC(*val*) – Returns the smallest element greater than *val* in the RBT.
- PRED(*val*) – Returns the largest element lesser than *val* in the RBT.
- DELETE(*val*) – Deletes *val* from the RBT.

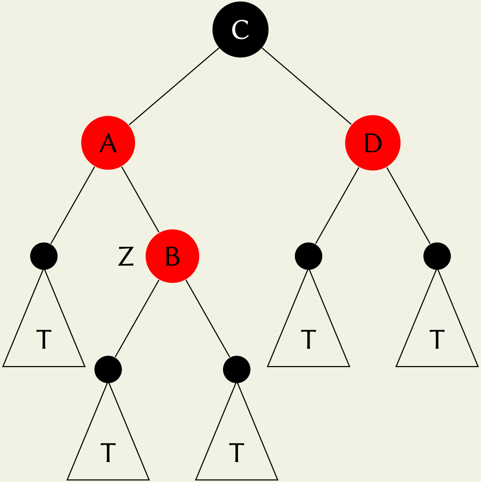
The procedures in green are implemented exactly like in a BST.

FixInsert pseudocode

Algorithm 1 FixInsert called on node Z

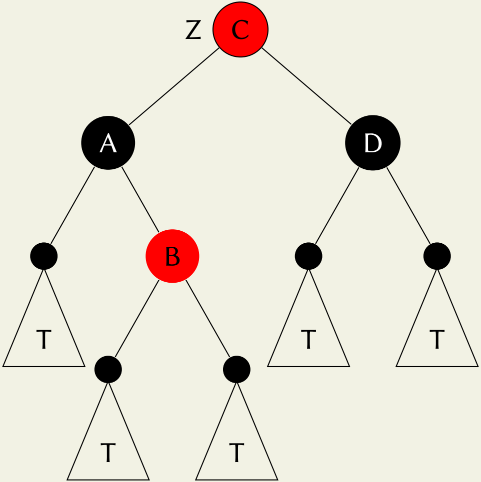
```
while color(parent(Z)) = red do
    U ← Uncle(Z)
    if parent(Z) is the left child of the grandparent then
        if color(U) = red then
            Recolor parent, uncle and grandparent.
            Z ← grandparent(Z).
        else
            if Z is the right child then
                Z ← parent(Z); Left rotate at (Z)
            end if
            Recolor parent and grandparent.
            Right rotate at grandparent(Z).
        end if
    end if
end while
color(root)← black.
```

Case 1



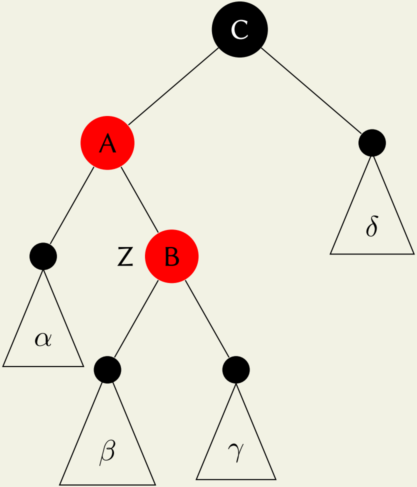
T = subtree of black height k
B could be on either side

Case 1



T = subtree of black height *k*
B could be on either side

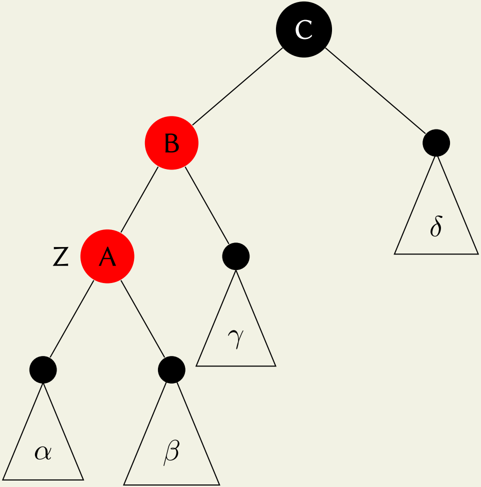
Case 2



$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

We do: Left Rotate at A to get to Case 3

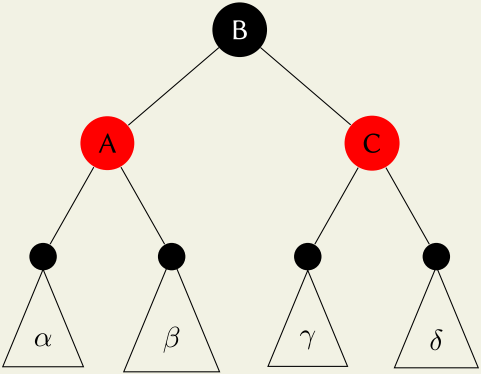
Case 2 –> Case 3



$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

We do right rotate at C

Case 3



$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

Solved!

Summary of Insert

- Case 1: Only recoloring. Pushes up the violation.
- Case 2: Only one rotation. Leads to Case 3.
- Case 3: Only one rotation. No more violations!

Summary of Insert

- Case 1: Only recoloring. Pushes up the violation.
- Case 2: Only one rotation. Leads to Case 3.
- Case 3: Only one rotation. No more violations!

- We have $\leq O(\log n)$ recolorings and ≤ 2 rotations
- Total time: $O(\log n)$

How did anyone come up with such an idea?

- 1962: First self-balancing tree invented by Adelson-Velsky and Landis: AVL Trees
- 1972: Bayer invents "symmetric binary B-trees" which became known as 2-3-4 Trees
- 1978: Guibas and Sedgewick studied 2-3-4 trees and introduced the analogous red-black notion
- Improved over the years

Delete procedure

The procedure to delete a node M at a high level:

- Case 1: M has two non-leaf children.
  - Replace (data of) M with the successor.
  - Splice out (delete) successor. This makes it case 2.

DELETE procedure

The procedure to delete a node M at a high level:

- Case 1: M has two non-leaf children.
  - Replace (data of) M with the successor.
  - Splice out (delete) successor. This makes it case 2.
- Case 2: M has at most one non-leaf child. Call this C.
  - Trivial case: M is red.
  - Minor case: M is black but C is red.
  - Major case: M and C are both black.

Note: If M has both leaf children (NIL), then C is any one of the NIL nodes.

M has at most one non-leaf child C

Trivial Case

M is red:

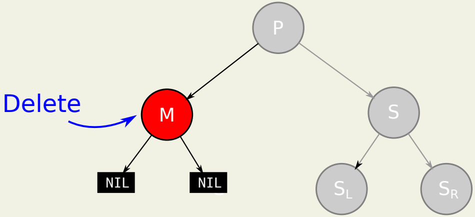
Resolution:

- Then simply replace M with C.

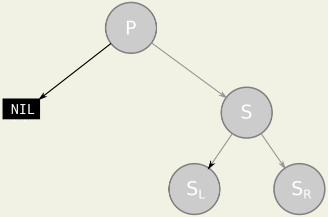
Note:

- M could not have been root.
- For all paths, the black height is not affected.

Trivial Case



Trivial Case



M has at most one non-leaf child C

Minor Case

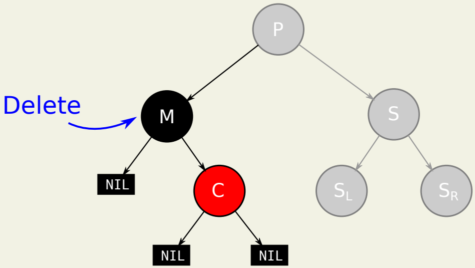
M is black and C is red.

Resolution:

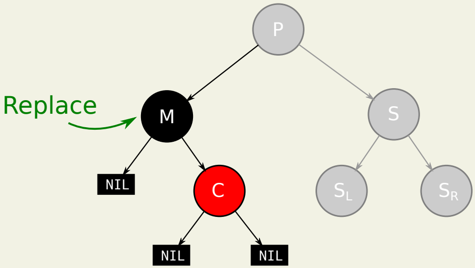
- Replace (splice) M with C.
- Place a "black token" on C.
- Safely discard black token by coloring C black.

The token indicates that the node contributes an extra black to the black count.

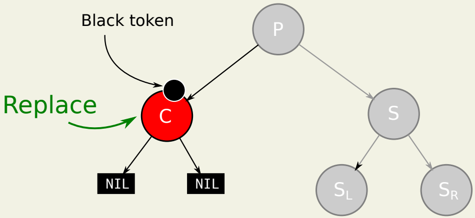
Minor Case



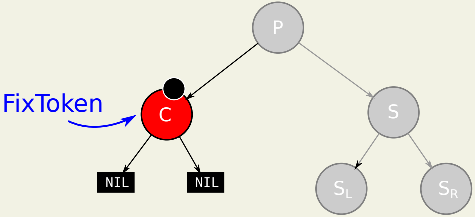
Minor Case



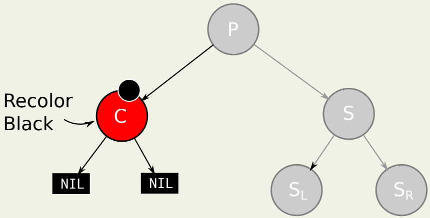
Minor Case



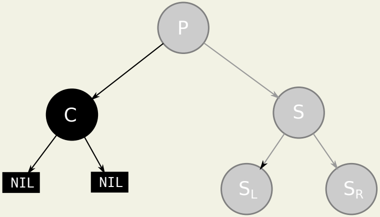
Minor Case



Minor Case



Minor Case



M has at most one non-leaf child C

Major Case

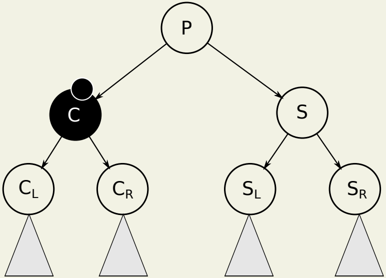
M is black and C is black.

Resolution:

- Replace (splice) M with C.
- Place a black token on C.
- Four cases to safely remove the token placed on C.

Major case - notation

Before replacement of M by C



Removing the token from C

Four cases based on the sibling S of C:

1. S is red.
2. S is black and has both children colored black.
3. S is black and has left child red and right child black.
4. S is black and has right child red.

Removing the token from C

Case 1

Sibling S of C is red.

Since S is red:

- It must have both black children.
- The parent P of S must be black.

Removing the token from C

Case 1

Sibling S of C is red.

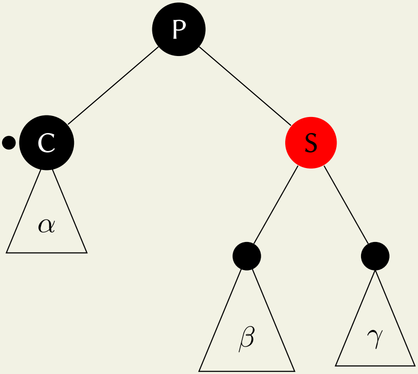
Since S is red:

- It must have both black children.
- The parent P of S must be black.

Resolution:

- Recolor S black and its parent red.
- Rotate at parent. (left rotate if C was left child)
- This is now one of cases 2, 3 or 4.

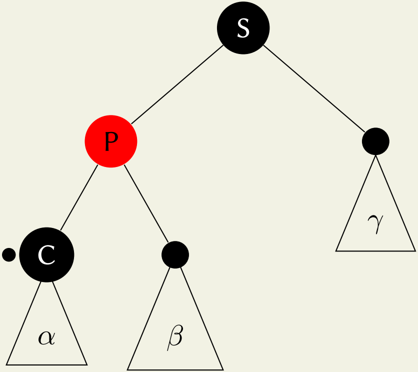
Case 1



$\alpha, \beta, \gamma =$ subtrees of black height *k* (including the black token)

Left Rotate about P and recolor

Case 1



$\alpha, \beta, \gamma, \delta =$ subtrees of black height *k* (including the black token)

Now in Case 2, 3 or 4

Case 2

Sibling S is black and has both black children.

Removing the token from C

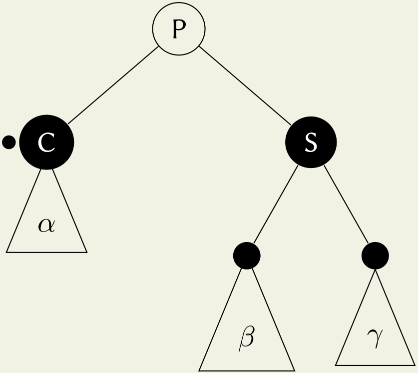
Case 2

Sibling S is black and has both black children.

Resolution:

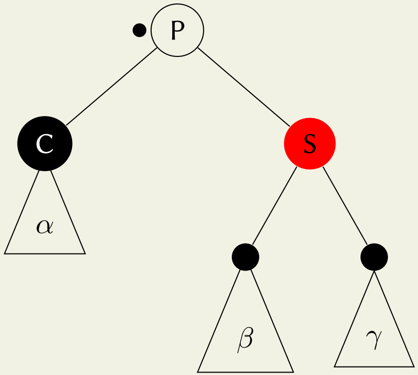
- Remove a black from both C and S.
- Paste token on the parent P.

Case 2



α = subtree of black height $k + 1$ (including the black token)
β, γ = subtrees of black height k
Shift the token to P

Case 2



P can absorb the token if red, else fixup continues

If we came from Case 1, then P is red.

Case 3

Sibling S is black.

Left child S_L is red.

S_R is black.

Case 3

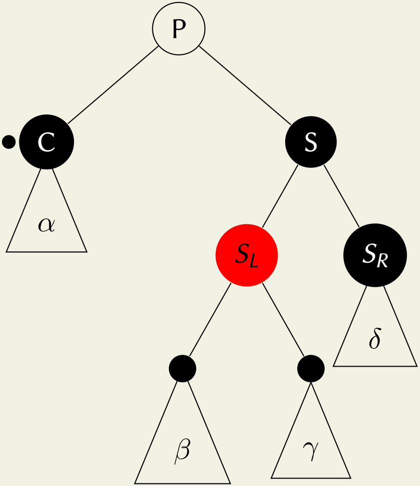
Sibling S is black.
Left child S_L is red.
S_R is black.

Resolution:

- Swap colors of S_L and S.
- Rotate right at S.

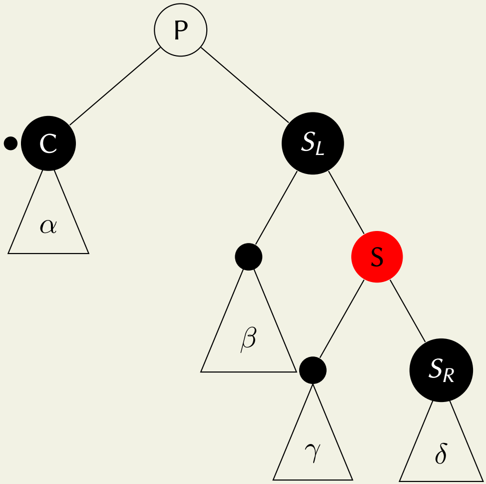
This gives us case 4.

Case 3



$\alpha =$ subtree of black height $k + 1$ (including the black token)
$\beta, \gamma, \delta =$ subtrees of black height k
Rotate and Recolor

Case 3



Now we are in Case 4

Case 4

Sibling S is black.

Right child S_R is red.

Removing the token from C

Case 4

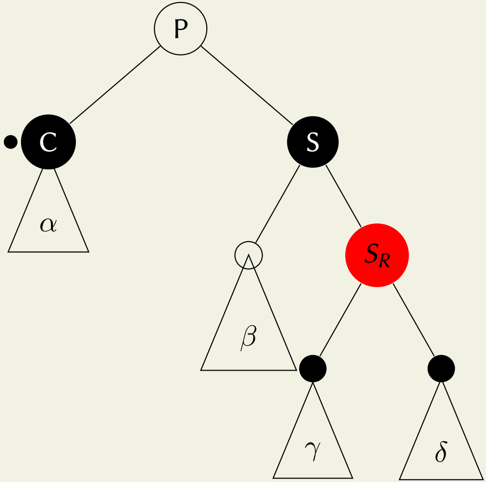
Sibling S is black.

Right child S_R is red.

Resolution:

- Color S_R black.
- S inherits the color of parent P.
- Color P black.
- Rotate left at P.
- Set token on root.

Case 4

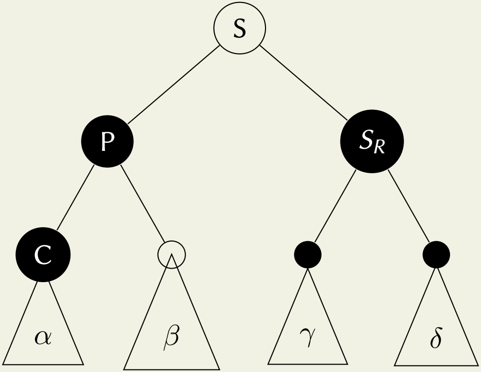


$\alpha =$ subtree of black height $k + 1$ (including the black token)

$\beta, \gamma, \delta =$ subtrees of black height k

Rotate and Recolor

Case 4



$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

Solved!

Case Progression

- Case 1 $\rightarrow$ Case 2 $\rightarrow$ end
- Case 1 $\rightarrow$ Case 3 $\rightarrow$ Case 4 $\rightarrow$ end
- Case 1 $\rightarrow$ Case 4 $\rightarrow$ end
- Case 2 $\rightarrow$ Loop in Case 2 $\rightarrow$ Case 1, 3 or 4
- Case 3 $\rightarrow$ Case 4 $\rightarrow$ end
- Case 4 $\rightarrow$ end