CIS 330
C/C++ and Unix

Red-Black Tree

Binary Search Tree (BST)

Search Tree

- Data structure that support many dynamic-set operations (i.e., change over time), including
- **Search, minimum, maximum, predecessor, successor, insert, and delete**

Binary Search Tree - Search tree where each node has two children (except for the leaves)

- Items can be looked up, as they are stored in sorted form (by key)

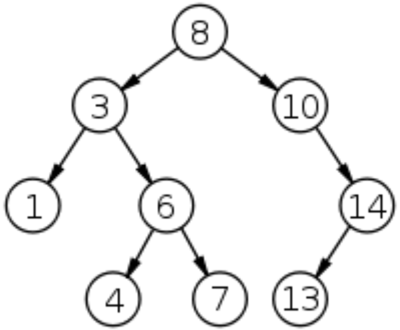
Complexity

- Log N - if complete
- N - if a linear chain with n nodes

Property of a BST

Binary Search Tree with

- Size 9
- Height 3
- Root at 8



Each node contains

- Key
- Left, right, parent node pointers (nil/null if missing)

Property

- Let x be a node in a BST -
  - if y is a node in the **left** subtree, then **key[y] <= key[x]**
  - if y is a node in the **right** subtree, then **key[x] <= key[y]**

Red-Black Tree

BST with extra bit of information in each node - its color - RED or BLACK

By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other (i.e., it is approximately balanced)

Property

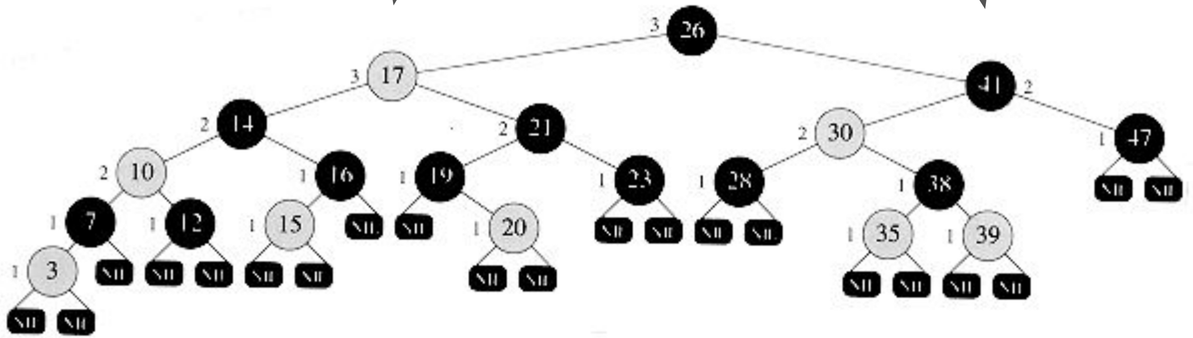
1. Every node is either RED or BLACK
2. The root is BLACK
3. Every leaf (a NIL) is BLACK (we'll see what this means in a bit)
4. If a node is RED, then both its children are BLACK
5. For each node, all paths from the node to descendent leaves contain the same number of BLACK nodes

Red-Black Tree

Red

Black

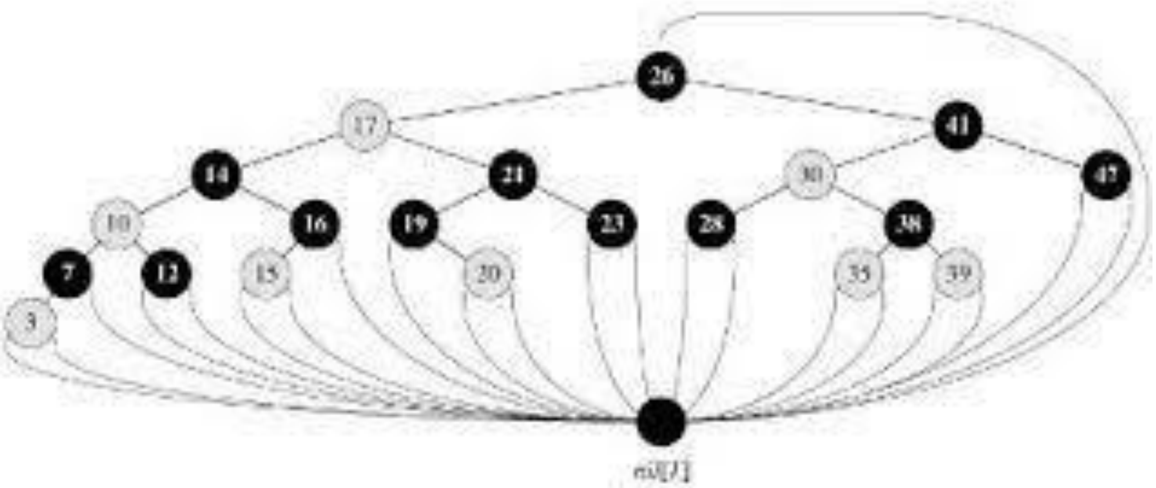
Every leaf is NIL



Every NIL (or nullptr in C++) is replaced with an actual Node*

We call this the sentinel

- Sentinel is of type `Node*` (or `RBTNode*`)
- Sentinel color is BLACK
- Other fields (i.e., parent, left, right, and key) are arbitrary (and usually not important)
- Represented as ***nil*[T]** in CLRS



There is a single sentinel and all pointers that used to point to NIL/nullptr in BST now points to this sentinel

For example

To accommodate RBT in our BST implementation, modifications have been made by adding a defaulted argument ***nilptr***

```
Node* BST::get_min(Node* in)
{
    Node* cur = in;
    while(cur->get_left() != nullptr) {
        cur = cur->get_left();
    }
    return cur;
}

get_min(in)
```

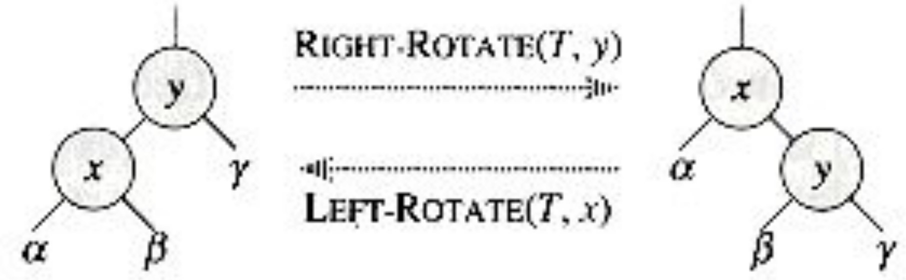
```
Node* get_min(Node* in, Node* nilptr = nullptr);

Node* BST::get_min(Node* in, Node nilptr)
{
    Node* cur = in;
    while(cur->get_left() != nilptr) {
        cur = cur->get_left();
    }
    return cur;
}

get_min(in, sentinel)
```

Red-Black Tree

What is required for the homework

- **rb-insert**
  - **rb-insert-fixup** (required to maintain RBT property)
    - **left-rotate, right-rotate**
- **rb-delete**
  - **rb-delete-fixup** (required to maintain RBT property)
    - left-rotate, right-rotate

Rotation




```
LEFT-ROTATE(T, x)
y ← right[x]                    ▷ Set y.
right[x] ← left[y]              ▷ Turn y's left subtree into x's right subtree.
if left[y] ≠ NIL
    then p[left[y]] ← x
p[y] ← p[x]                     ▷ Link x's parent to y.
if p[x] = NIL
    then root[T] ← y
    else if x = left[p[x]]
            then left[p[x]] ← y
            else right[p[x]] ← y
left[y] ← x                     ▷ Put x on y's left.
p[x] ← y
```

Rotation

Right rotate is similar to left-rotate (somewhat symmetrical)

Design the algorithm first, and then implement

Insert

Reuse existing code!
However, be careful here, you may need to do some additional things to account for **sentinels** in the skeleton code

```
RB-INSERT(T, x)
TREE-INSERT(T, x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED
    do if p[x] = left[p[p[x]]]
          then y ← right[p[p[x]]]
               if color[y] = RED
                  then color[p[x]] ← BLACK                    ▷ Case 1
                       color[y] ← BLACK                       ▷ Case 1
                       color[p[p[x]]] ← RED                   ▷ Case 1
                       x ← p[p[x]]                            ▷ Case 1
                  else if x = right[p[x]]
                          then x ← p[x]                       ▷ Case 2
                               LEFT-ROTATE(T, x)              ▷ Case 2
                       color[p[x]] ← BLACK                    ▷ Case 3
                       color[p[p[x]]] ← RED                   ▷ Case 3
                       RIGHT-ROTATE(T, p[p[x]])               ▷ Case 3
          else (same as then clause
                  with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

rb-insert-fixup

Used to restore the red-black properties of the tree

Recall that

Property

1. Every node is either RED or BLACK
2. The root is BLACK
3. Every leaf is a sentinel and is BLACK (sentinel is BLACK by definition)
4. If a node is RED, then both its children are BLACK
5. For each node, all paths from the node to descendent leaves contain the same number of BLACK nodes

After regular BST insertion, which properties are violated?

1 -> every node is still RED or BLACK

3 -> Every leaf (or sentinel) is BLACK

5-> What about this one?

rb-insert-fixup

For each node, all paths from the node to descendent leaves contain the same number of BLACK nodes

- Node x replaces the sentinel (BLACK) on the left or right
  - 1 fewer BLACK
  - 1 more RED (because x is colored RED during rb-insert)
- x's children are pointing to the sentinel
  - 1 more BLACK to left
  - 1 more BLACK to right
- Total number of BLACK remains the same
  - BLACK: -1 + 1 = **0** (either right or left)

rb-insert-fixup

Property

1. ~~Every node is either RED or BLACK~~
2. **The root is BLACK**
3. ~~Every leaf (a NIL) is BLACK (we'll see what this means in a bit)~~
4. **If a node is RED, then both its children are BLACK**
5. ~~For each node, all paths from the node to descendent leaves contain the same number of BLACK nodes~~

2 -> If the new node becomes the root, this is violated

- Fixed by setting the root to BLACK (since BLACK node can have either RED or BLACK children, no property is violated)

4-> Newly added node is RED, and if it's added to a parent who happens to be RED, this is violated

Insert

The while loop maintains some invariant at the beginning of each iteration

```
RB-INSERT(T, x)
  TREE-INSERT(T, x)
  color[x] ← RED
  while x ≠ root[T] and color[p[x]] = RED
      do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]
                 if color[y] = RED
                    then color[p[x]] ← BLACK              ▷ Case 1
                         color[y] ← BLACK                 ▷ Case 1
                         color[p[p[x]]] ← RED             ▷ Case 1
                         x ← p[p[x]]                      ▷ Case 1
                    else if x = right[p[x]]
                            then x ← p[x]                 ▷ Case 2
                                 LEFT-ROTATE(T, x)        ▷ Case 2
                         color[p[x]] ← BLACK              ▷ Case 3
                         color[p[p[x]]] ← RED             ▷ Case 3
                         RIGHT-ROTATE(T, p[p[x]])         ▷ Case 3
            else (same as then clause
                    with "right" and "left" exchanged)
  color[root[T]] ← BLACK
```

rb-insert-fixup

Invariant

1. Node x (input) is RED
2. If p[x] is the root, p[x] is BLACK
3. **If there is a violation of the red-black properties, there is at most one violation, and it's either property 2 or property 4**
   a. **If it's violation of 2, it occurs because x is the root (and is therefore, RED)**
   b. **If it's violation of 4, it occurs because both x and p[x] are RED.**

3 needs to be resolved so that the algorithm can terminate

During the algorithm iteration, either

1. x moves up the tree, or
2. some rotation is performed and the loop terminates

Insert

```
RB-INSERT(T, x)
  TREE-INSERT(T, x)
  color[x] ← RED
  while x ≠ root[T] and color[p[x]] = RED
      do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]
                 if color[y] = RED
                   then color[p[x]] ← BLACK              ▷ Case 1
                        color[y] ← BLACK                 ▷ Case 1
                        color[p[p[x]]] ← RED             ▷ Case 1
                        x ← p[p[x]]                      ▷ Case 1
                   else if x = right[p[x]]
                          then x ← p[x]                  ▷ Case 2
                               LEFT-ROTATE(T, x)         ▷ Case 2
                        color[p[x]] ← BLACK              ▷ Case 3
                        color[p[p[x]]] ← RED             ▷ Case 3
                        RIGHT-ROTATE(T, p[p[x]])         ▷ Case 3
            else (same as then clause
                   with "right" and "left" exchanged)
  color[root[T]] ← BLACK
```

First conditional takes care of when p[x] is a left child (of p[p[x]]

Second part takes care of when p[x] is a right child (of p[p[x]])

Insert

y is the sibling of p[x]
(or x's uncle)

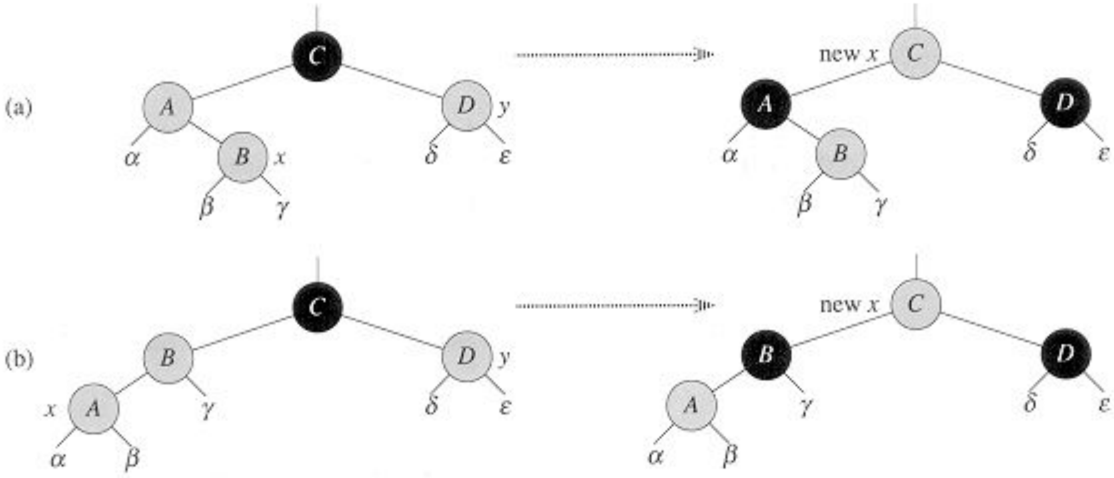
Case 1 - if both parent and uncle are RED

```
RB-INSERT(T, x)
TREE-INSERT(T, x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED
      do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]
                 if color[y] = RED
                    then color[p[x]] ← BLACK                  ▷ Case 1
                         color[y] ← BLACK                     ▷ Case 1
                         color[p[p[x]]] ← RED                 ▷ Case 1
                         x ← p[p[x]]                          ▷ Case 1
                    else if x = right[p[x]]
                            then x ← p[x]                     ▷ Case 2
                                 LEFT-ROTATE(T, x)            ▷ Case 2
                         color[p[x]] ← BLACK                  ▷ Case 3
                         color[p[p[x]]] ← RED                 ▷ Case 3
                         RIGHT-ROTATE(T, p[p[x]])             ▷ Case 3
            else (same as then clause
                    with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

Insert

Case 1

x's uncle y is RED



p[p[x]] has to be BLACK (otherwise it wouldn't have been a legal RBT in the first place)

Color p[x] and y BLACK, and p[p[x]] as RED, fixing the problem ***locally***

Move x up the tree by two levels to p[p[x]], and repeat the while loop with p[p[x]] as the new input x

Insert

Is invariance at the beginning of the while loop maintained?

1. new x is RED (since that's what we did)
2. if p[x] is the root, p[x] is BLACK (since we haven't touched this, node, it should be BLACK if it is the root)
3. If new x is the root at the start of the next iteration, there was only 1 violation of property 4 and it has been fixed and the only **violation to consider is property 2**.

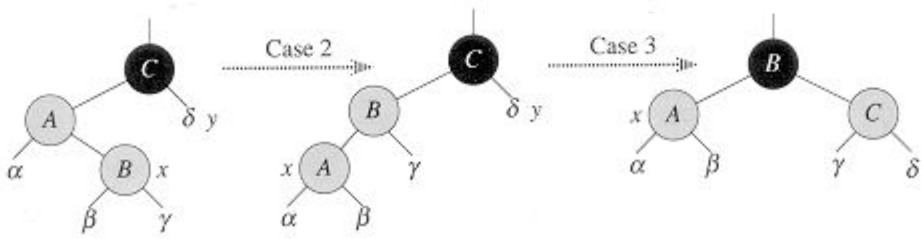
   If new x is NOT the root at the start of the next iteration, then we have not created a violation of property 2. If p[new x] is RED, we now created **a violation to property 4**, otherwise there is **no violation** to property 4

   → Therefore, there is at most 1 violation, either violation of property 2 or property 4

Insert

Case 2 - x's uncle y is BLACK and x is a right child

Case 3 - x's uncle y is BLACK and x is a left child



Left rotation transforms case 2 to case 3

Color change + right rotate fixes the problem

Insert

```
RB-INSERT(T, x)
 TREE-INSERT(T, x)
 color[x] ← RED
 while x ≠ root[T] and color[p[x]] = RED
     do if p[x] = left[p[p[x]]]
           then y ← right[p[p[x]]]
                if color[y] = RED
                  then color[p[x]] ← BLACK                    ▷ Case 1
                       color[y] ← BLACK                       ▷ Case 1
                       color[p[p[x]]] ← RED                   ▷ Case 1
                       x ← p[p[x]]                            ▷ Case 1
                  else if x = right[p[x]]
                         then x ← p[x]                        ▷ Case 2
                              LEFT-ROTATE(T, x)               ▷ Case 2
                       color[p[x]] ← BLACK                    ▷ Case 3
                       color[p[p[x]]] ← RED                   ▷ Case 3
                       RIGHT-ROTATE(T, p[p[x]])               ▷ Case 3
           else (same as then clause
                  with "right" and "left" exchanged)
 color[root[T]] ← BLACK
```

Case 1 - if both parent and uncle are RED

Case 3 - you are the left child

Case 2 - if uncle is BLACK and you are a right child

That was the easy part

Let’s move on to delete

Delete

Similar to insert

Requires a small modification to deletes (BST) and a fixup code to re-"balance" the tree

Delete

RB-DELETE (T, z)

```
if left[z] = nil[T] or right[z] = nil[T]
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
   then x ← left[y]
   else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
  then root[T] ← x
    else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        If y has other fields, copy them, too.
if color[y] = BLACK
   then RB-DELETE-FIXUP (T,x)
return y
```

NIL/nullptr replaced by nil[T]

BST delete:
if(x != NIL)
  p[x] <- p[y]
If x is the sentinel, it is used to temporarily store p[y]

The BST assignment required "splicing" the nodes instead of copying the data - the "key" data for Node class was made private so that you couldn't replace existing data in Nodes

Red-black tree fixup on x (not z)

If y is RED, fixup is not necessary, as red-black properties still hold when y is spliced out

Delete

If the spliced out node, y, is BLACK, three problems may arise

1. if y had been the root, and a RED child of y becomes the new root, we have violated **property 2** (i.e., root is BLACK)
2. if both x and p[y] (which is now p[x]) were RED, then we have violated **property 4** (if a node is RED, both its children are BLACK)
3. y's removal causes any path that had previously contained y to have one fewer BLACK node and violates **property 5** (all paths from a node to descendant leaves contains the same number of BLACK nodes)

What can we do?

- Say that node x has an "extra" BLACK - i.e., increase the count of BLACK nodes by 1 whenever it encounters x
- This fixes the violation to property 5.
- Additionally, assume that x can be either BLACK-BLACK, or RED-BLACK (both options add an extra BLACK to x)

Delete

RB-DELETE-FIXUP

Property 2 and 4 are easy to handle (similar in idea to what we did in insert)

By trying to fix property 5, we made a node RED-BLACK or BLACK-BLACK, we broke property 1

fixup moves the extra BLACK up the tree until

- x points to a RED-BLACK node, in which case we make it BLACK, or
- x points to the root, in which case the extra BLACK is simply removed, or
- suitable rotation and recoloring can be performed

Remember that the node x passed to fixup is one of two nodes

- The node that was y's sole child before y was spliced out (if y had a child that was not the sentinel/NIL)
- Sentinel nil[T] (if y had no children)

Delete

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w ← right[p[x]]
               if color[w] = RED
                  then color[w] ← BLACK                           ▷ Case 1
                       color[p[x]] ← RED                          ▷ Case 1
                       LEFT-ROTATE(T, p[x])                       ▷ Case 1
                       w ← right[p[x]]                            ▷ Case 1
               if color[left[w]] = BLACK and color[right[w]] = BLACK
                  then color[w] ← RED                             ▷ Case 2
                       x ← p[x]                                   ▷ Case 2
                  else if color[right[w]] = BLACK
                          then color[left[w]] ← BLACK             ▷ Case 3
                               color[w] ← RED                     ▷ Case 3
                               RIGHT-ROTATE(T, w)                 ▷ Case 3
                               w ← right[p[x]]                    ▷ Case 3
                       color[w] ← color[p[x]]                     ▷ Case 4
                       color[p[x]] ← BLACK                        ▷ Case 4
                       color[right[w]] ← BLACK                    ▷ Case 4
                       LEFT-ROTATE(T, p[x])                       ▷ Case 4
                       x ← root[T]                                ▷ Case 4
          else (same as then clause
                with "right" and "left" exchanged)
color[x] ← BLACK
```

Within the while loop, x is always a non-root and a double-BLACK node

Delete

RB-DELETE-FIXUP(T, x)

```
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
        then w ← right[p[x]]
            if color[w] = RED
                then color[w] ← BLACK                    ▷ Case 1
                    color[p[x]] ← RED                    ▷ Case 1
                    LEFT-ROTATE(T, p[x])                 ▷ Case 1
                    w ← right[p[x]]                      ▷ Case 1
            if color[left[w]] = BLACK and color[right[w]] = BLACK
                then color[w] ← RED                      ▷ Case 2
                    x ← p[x]                             ▷ Case 2
                else if color[right[w]] = BLACK
                        then color[left[w]] ← BLACK      ▷ Case 3
                            color[w] ← RED               ▷ Case 3
                            RIGHT-ROTATE(T, w)           ▷ Case 3
                            w ← right[p[x]]              ▷ Case 3
                    color[w] ← color[p[x]]               ▷ Case 4
                    color[p[x]] ← BLACK                  ▷ Case 4
                    color[right[w]] ← BLACK              ▷ Case 4
                    LEFT-ROTATE(T, p[x])                 ▷ Case 4
                    x ← root[T]                          ▷ Case 4
        else (same as then clause
                with "right" and "left" exchanged)
color[x] ← BLACK
```

Determine if x is a left child

Same algorithm applies if ix is right child

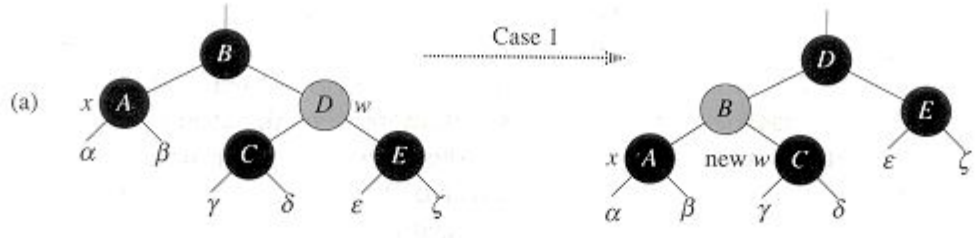
Delete

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
        then w ← right[p[x]]
            if color[w] = RED
                then color[w] ← BLACK                          ▷ Case 1
                    color[p[x]] ← RED                          ▷ Case 1
                    LEFT-ROTATE(T, p[x])                       ▷ Case 1
                    w ← right[p[x]]                            ▷ Case 1
            if color[left[w]] = BLACK and color[right[w]] = BLACK
                then color[w] ← RED                            ▷ Case 2
                    x ← p[x]                                   ▷ Case 2
                else if color[right[w]] = BLACK
                        then color[left[w]] ← BLACK            ▷ Case 3
                            color[w] ← RED                     ▷ Case 3
                            RIGHT-ROTATE(T, w)                 ▷ Case 3
                            w ← right[p[x]]                    ▷ Case 3
                    color[w] ← color[p[x]]                     ▷ Case 4
                    color[p[x]] ← BLACK                        ▷ Case 4
                    color[right[w]] ← BLACK                    ▷ Case 4
                    LEFT-ROTATE(T, p[x])                       ▷ Case 4
                    x ← root[T]                                ▷ Case 4
        else (same as then clause
                with "right" and "left" exchanged)
color[x] ← BLACK
```

Maintain a pointer (w) to sibling of x

Delete

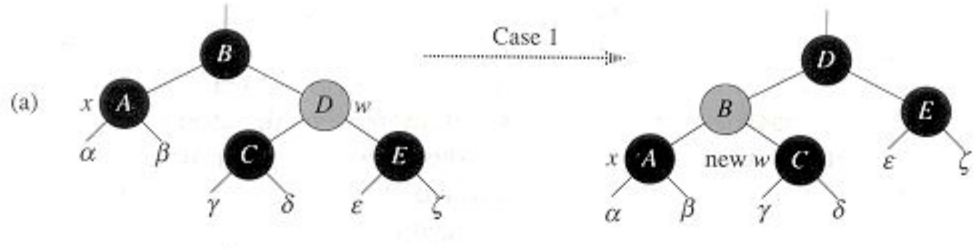
In each of the four cases, # of BLACK nodes (including x's extra BLACK) is preserved (to maintain property 5)



# of BLACK nodes from root (B or D) to alpha or beta is 3 (A is x and has two BLACKs) before and after the transformation

Delete

Case 1: x's sibling, w, is RED



We know both of w's children must be black (due to red-black tree's property 4)

p[x] must also be black (if p[x] was RED, w could not be RED)

Therefore, we can switch the colors of w and p[x] and then perform a left-rotation on p[x] without violating red-black property

- p[x] becomes RED, w becomes BLACK
- left-roate: w replaces p[x], and p[x] becomes w's child
- Since w is now BLACK, it can have either RED or BLACK children
- New sibling of x (which is one of w's children prior to rotation) is also BLACK

Delete

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w ← right[p[x]]
               if color[w] = RED
                  then color[w] ← BLACK                        ▷ Case 1
                       color[p[x]] ← RED                       ▷ Case 1
                       LEFT-ROTATE(T, p[x])                    ▷ Case 1
                       w ← right[p[x]]                         ▷ Case 1
               if color[left[w]] = BLACK and color[right[w]] = BLACK
                  then color[w] ← RED                          ▷ Case 2
                       x ← p[x]                                ▷ Case 2
                  else if color[right[w]] = BLACK
                          then color[left[w]] ← BLACK          ▷ Case 3
                               color[w] ← RED                  ▷ Case 3
                               RIGHT-ROTATE(T, w)              ▷ Case 3
                               w ← right[p[x]]                 ▷ Case 3
                       color[w] ← color[p[x]]                  ▷ Case 4
                       color[p[x]] ← BLACK                     ▷ Case 4
                       color[right[w]] ← BLACK                 ▷ Case 4
                       LEFT-ROTATE(T, p[x])                    ▷ Case 4
                       x ← root[T]                             ▷ Case 4
          else (same as then clause
                  with "right" and "left" exchanged)
color[x] ← BLACK
```

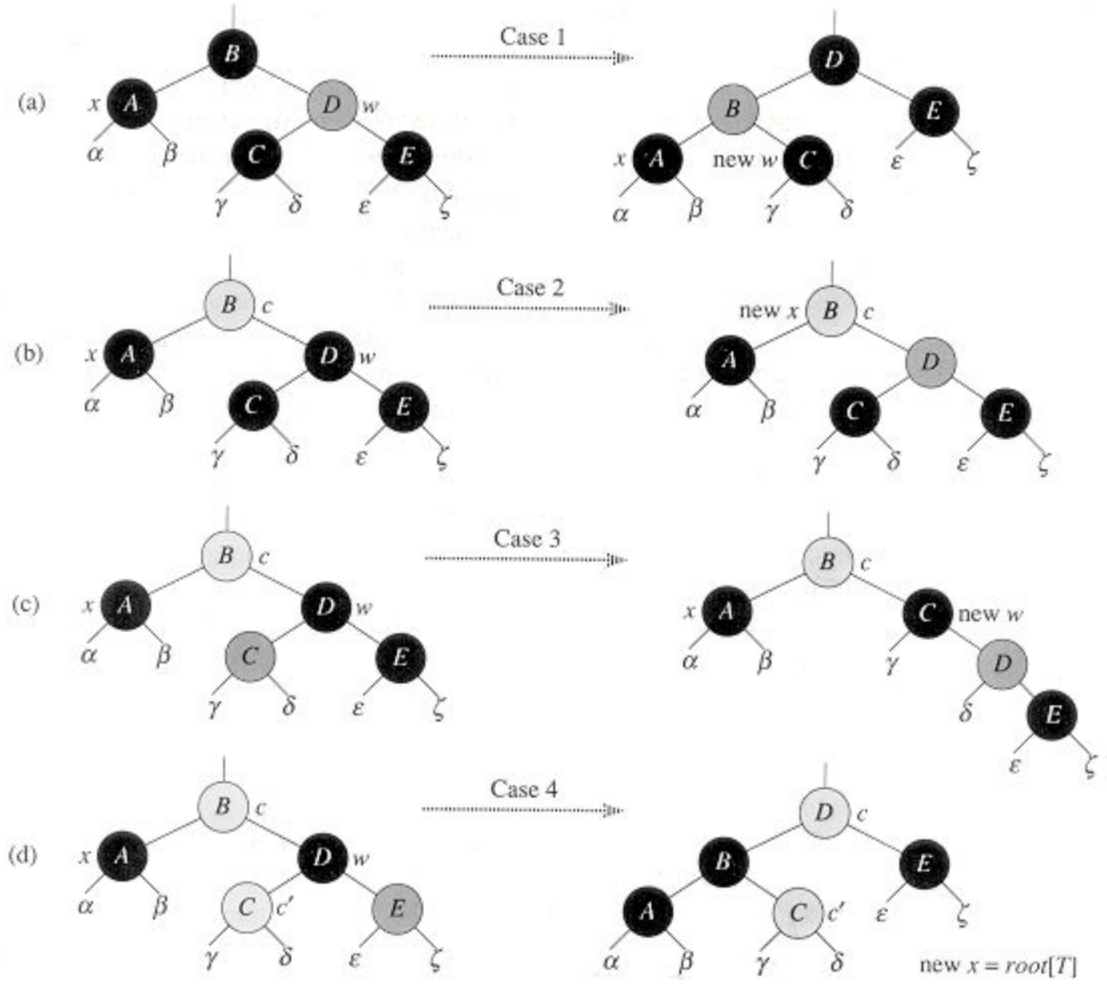
Delete

Case 2: x's sibling, w, is BLACK, and both of w's children are BLACK

Make w RED and set x to be p[x] - this has the effect of

- We take one BLACK off both x and w
- x becomes "regular" BLACK , and w becomes RED
- To compensate for removing BLACK from x and w, add a BLACK to p[x] (p[x] it could original be either RED or BLACK)
    - Since x and w are on different paths, adding one BLACK to the parent maintains the same # of BLACKS down either path
    - p[x] is now either RED-BLACK or BLACK-BLACK
- If p[x] is BLACK-BLACK, we repeat this in the while loop by setting new x to be p[x], which keeps pushing the extra BLACK up the tree. If it's RED-BLACK, while loop terminates
- Also, note that if Case 2 is entered through Case 1, our p[x] becomes RED-BLACK (because in case 1, parent of x and w was made RED).

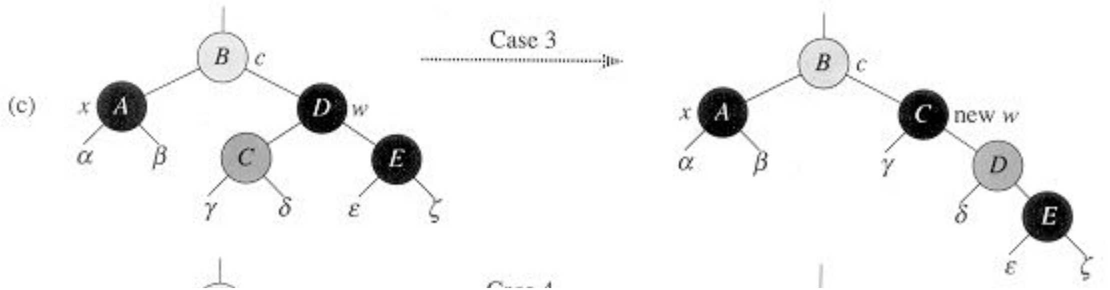
Delete



Delete

Case 3: x’s sibling, w, is BLACK, left[w] is RED, right[w] is BLACK

We can switch the colors of w and its left child and then perform a right-rotation on w

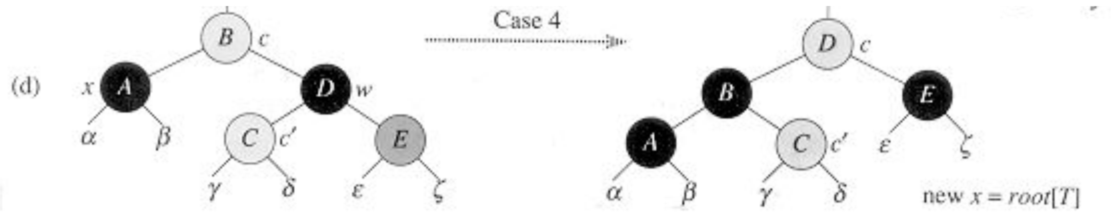


Maintains the properties of red-black tree

This transforms Case 3 to Case 4

Delete

Case 4: x's sibling, w, is BLACK, and w's right child is RED



Change colors

- w gets p[x]'s color
- p[x] is made BLACK
- right[w] is made BLACK

Left-rotate on p[x]

This removes the extra BLACK on x, without violating any properties

- # of BLACK on path from root (B or D) to alpha/beta remains 2
- Same for all the others (i.e., gamma, delta, epsilon, zeta)

Forcibly terminate by setting x to root

More details

http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap14.htm#:~:text=A%20red%2Dblack%20tree%20is,be%20either%20RED%20or%20BLACK.