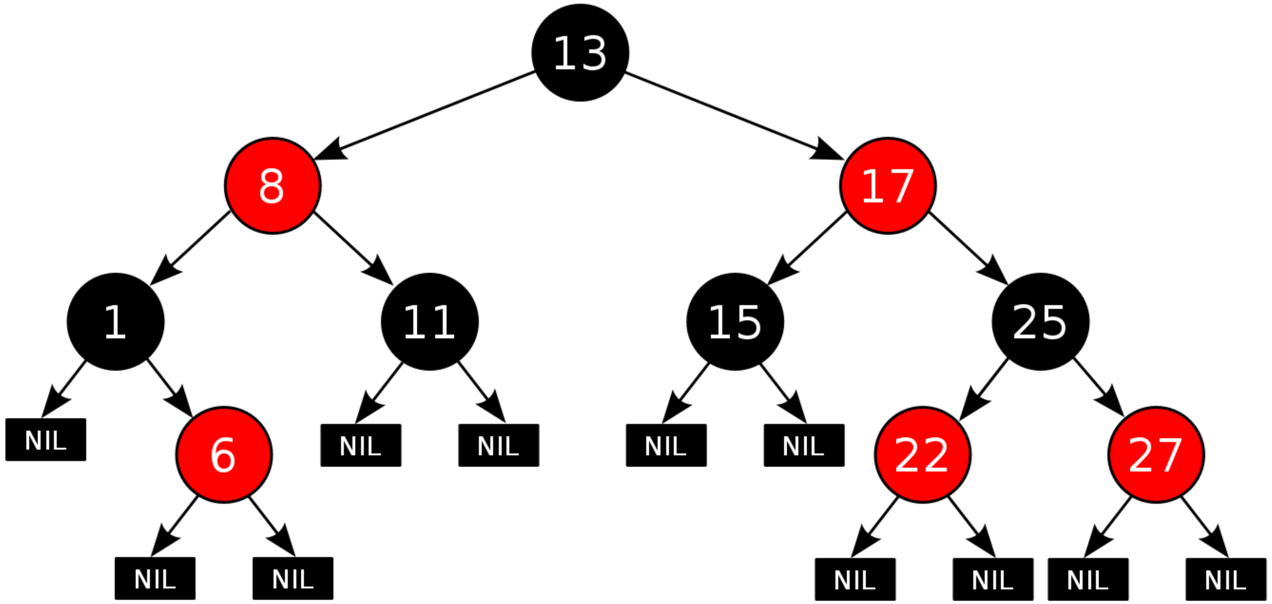
Algorithms and Data Structures (CSci 115)

California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- **Red-Black trees**
  - Definitions
  - How to search, insert, remove elements



Definitions

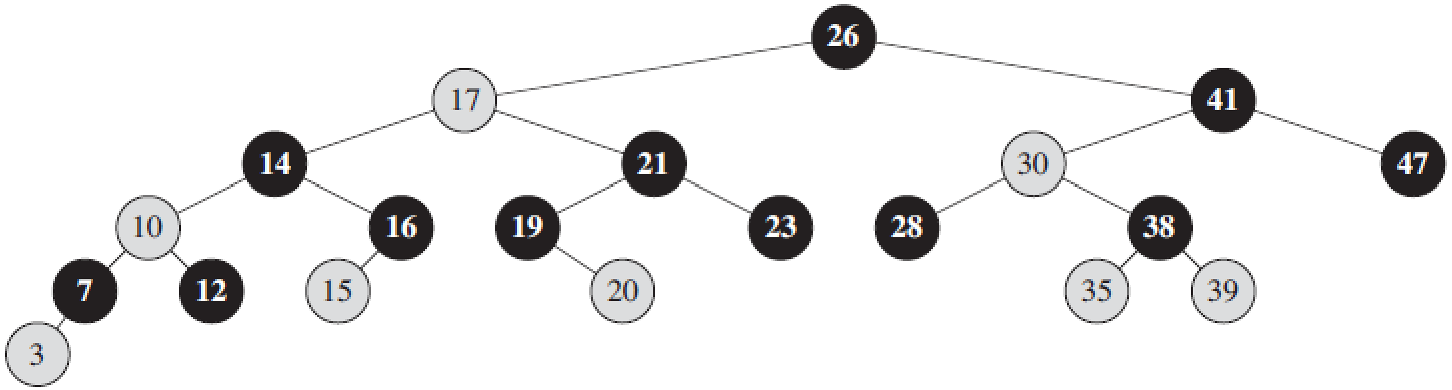
- Red-Black Tree (RBT) (1972 – Rudolph Bayer)
  - Binary Search Tree (BST)
    - with one extra **bit** of storage per node:
      - its color: **RED** or **BLACK**.
  - Constraint of the node colors on any simple path from the root to a leaf,
    - RBT ensure that:
      - no such path is more than 2 times as long as any other,
      - → the tree is approximately balanced.
  - Each node of the tree contains the attributes
    1. Color
    2. Key
    3. Left
    4. Right
    5. Parent
  - If a child or the parent of a node does not exist
    - the corresponding pointer attribute of the node contains the value NIL. (NIL == NULL == null pointer)
  - NULL == pointers to leaves (external nodes) of the BST
  - the key-bearing nodes == internal nodes of the tree.

Properties

- 5 key 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 simple paths from the node to descendant leaves contain the **same** number of **black** nodes.

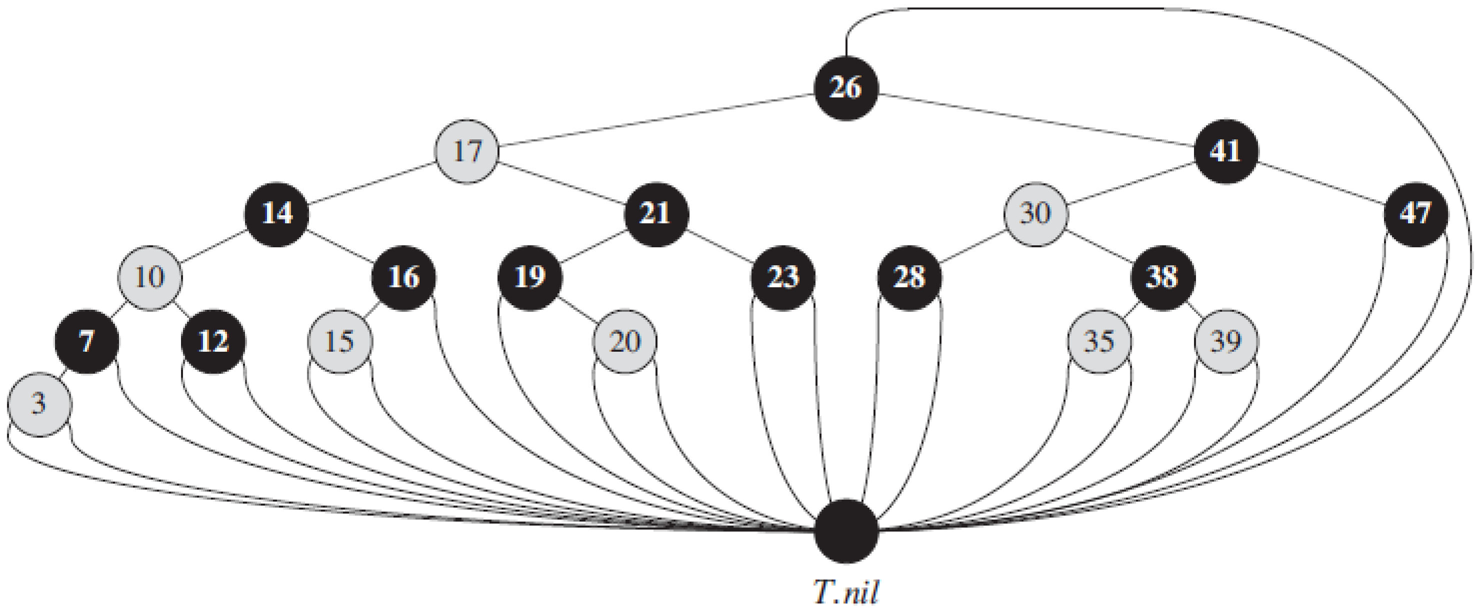
Example

- RBT
  - with leaves and the root's parent omitted entirely.



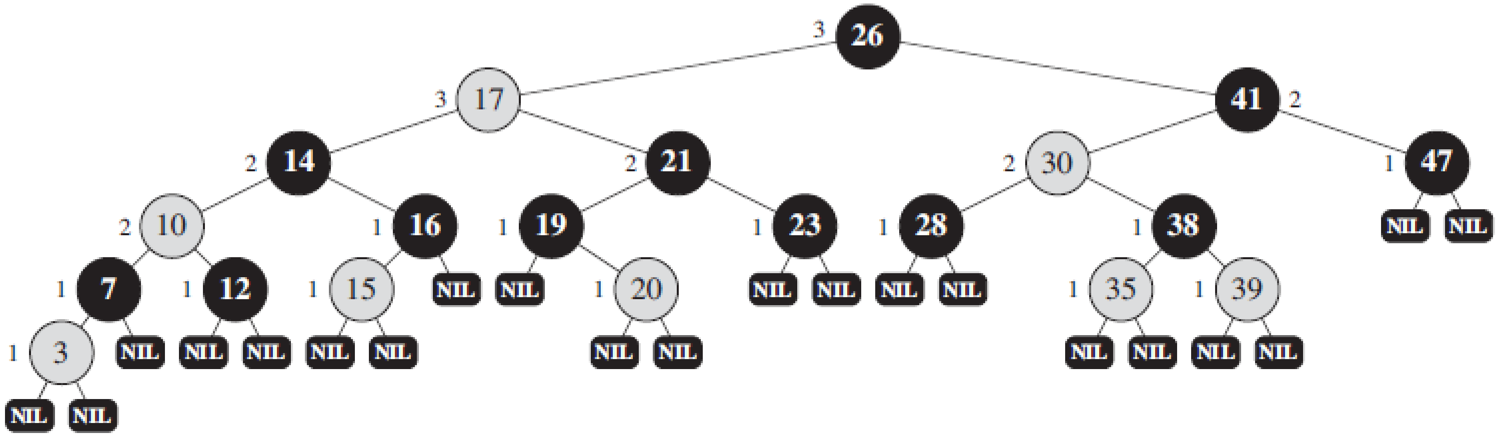
Example

- RBT
  - with each NIL replaced by the single sentinel T:*nil*,
    - which is always **black**, and with black-heights omitted.



Example

- RBT
  - ➢Every leaf, shown as a NIL = **black**.
  - ➢Each non-NIL node is marked with its **black**-height
  - ➢NILs have black-height 0.



Definitions

- The **black-height** of the node: bh(x)
  - Number of **black** nodes on any simple path from, but not including, a node x down to a leaf
- Property 5 (red → both children are black) →
  - well defined notion of **black**-height because
    - all descending simple paths from the node have the **same number of black nodes**.
  - We define the **black-height** of a RBT == the **black-height** of its **root**.
- RBT with n internal nodes → height at most 2*log(n+1)

Proof (1)

- Let h be the height of the tree.
- Show that
  - ➢ Subtree rooted at any node x contains at least $2^{bh(x)}-1$ internal nodes.
- By induction on the height of x.
  - ➢ If (h(x)==0) then
    - x must be a leaf (T.*nil*)
    - the subtree rooted at x contains at least $2^{bh(x)}-1 = 0$ internal nodes.
  - ➢ Inductive step
    - consider a node x with h>0 and x is an internal node with 2 children.
    - Each child has bh(x) (**red**) or bh(x)-1 (**black**)
      - Because the h(x.child) < h(x)
  - ➢ Apply the inductive hypothesis
  - ➢ → each child has at least $2^{bh(x)-1}-1$ internal nodes
  - ➢ → the subtree rooted at x contains **at least**
    - $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1) + 1 = 2^{bh(x)}-1$ internal nodes

Proof (2)

- Property 4
  - At least half the nodes on any simple path from the root to a leaf, not including the root, must be **black**
- → bh(root) must be at least h/2
- → $n \geq 2^{h/2}-1$
  - → $\log(n+1) \geq h/2$
  - $2 \log(n+1) \geq h$

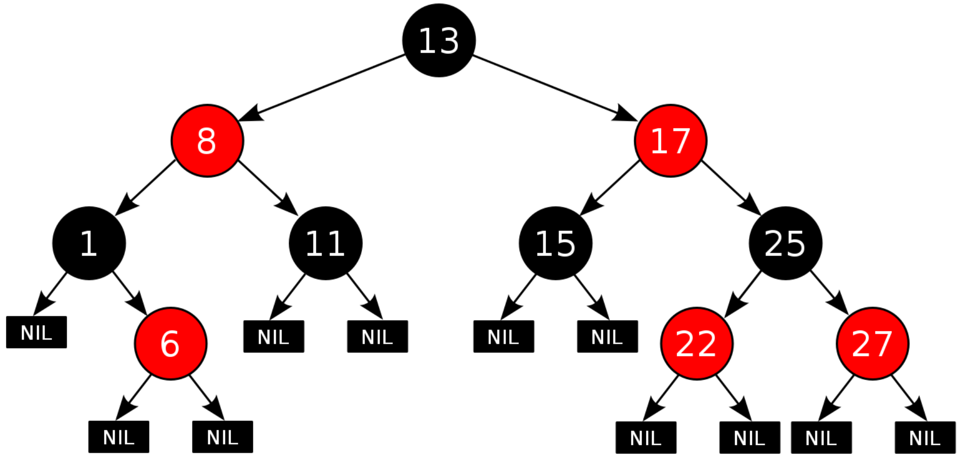
- We can implement Search, Minimum, Maximum, Successor, and Predecessor in O(log n)

Relationships with B-trees

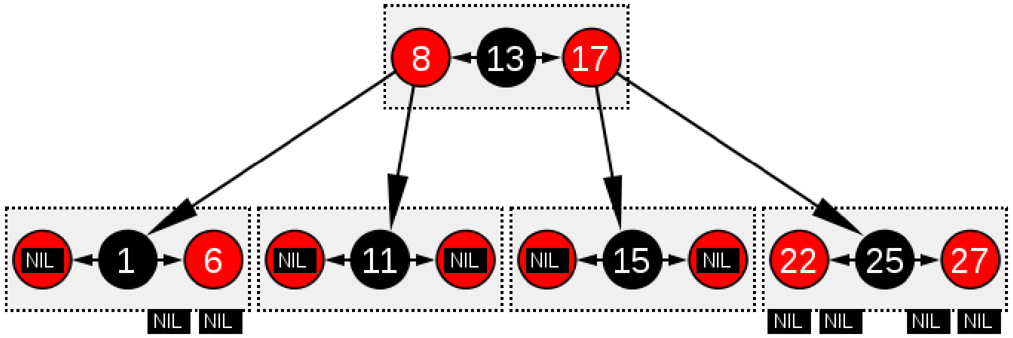
- RBT
  - Same structure as a B-tree of order **4**
    - Each node can contain
      - between 1 and 3 values
      - between 2 and 4 child pointers
  - In this B-tree, each node will contain only 1 value matching the value in a **black** node of the RBT
    - with an optional value before and/or after it in the same node
    - matching an equivalent **red** node of the RBT
- Move up the **red** nodes
  - → they align horizontally with their parent **black** node
    - Through the creation of an horizontal cluster.
    - B-tree, / modified graphical representation of the RBT → all leaf nodes have the same depth.
- A minimum fill factor of 33% of values per cluster
  - with a maximum capacity of 3 values.

Relationships with B-trees

- RBT versus B-Tree
  - Property 4



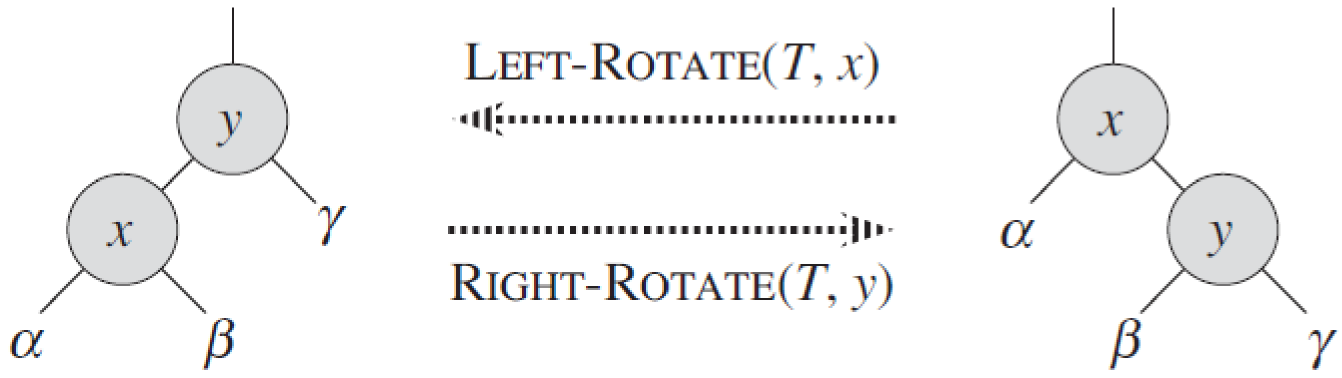
RBT



B-tree

Red Black Tree

- Insert & Delete
  - We must keep the constraints of the tree
  - → change the colors of some of the modes in the RBT
  - → change the pointer structure of the RBT
    - Through Rotation
      - Like AVL trees



Insertion & Deletion

- Node z
  - Parent: z.p
  - Grand-parent: z.p.p
  - Uncle (y)
    - z.p.p.left
      - if z.p.p.right == z.p
    - z.p.p.right
      - if z.p.p.left == z.p

Insertion

- Insert node **z** in tree T
  - Default insert like in a BST
  - Set color to RED
  - InsertFixup

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

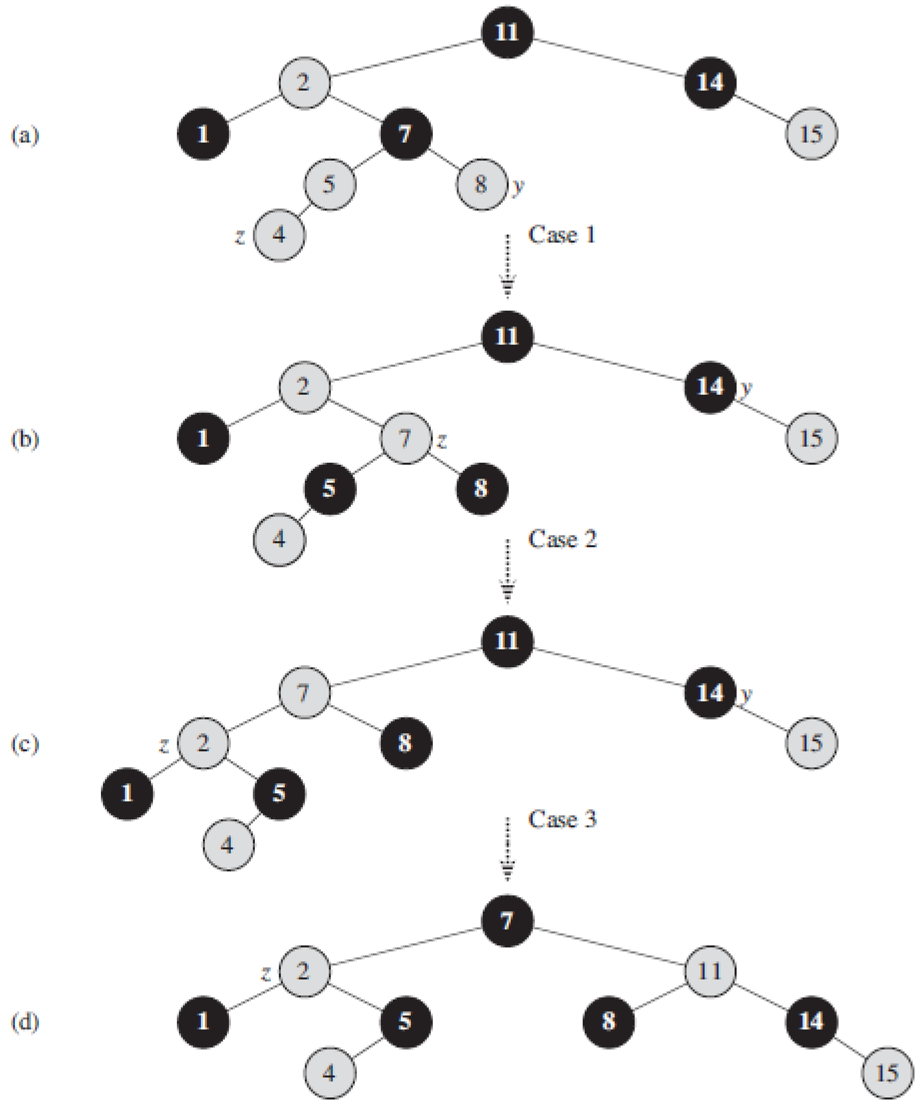
Find where to insert
Same as BST

Special case: root?

Default color: **RED**

Insertion

- Fix the tree, consider **3 main cases**
  - ➢z and its parent z.p are red
    - → violation of property 4
  - ➢z's uncle y is red → **case 1**
    - recolor nodes and move the pointer z up the tree **(b)**
  - ➢z and its parent are both red
    - but z's uncle y is black
    - Because z is the right child of z.*p* → **case 2**
    - perform a **left rotation (c)**
    - z is the left child of its parent → **case 3** recoloring and **right rotation (d)**



Insertion

- Fix tree
  - Pseudo Code

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right                      z.p.p = grand parent
        if y.color == RED                    y is RED
            z.p.color = BLACK                              // case 1
            y.color = BLACK                                // case 1
            z.p.p.color = RED                              // case 1
            z = z.p.p                                      // case 1
        else if z == z.p.right              y is Black
                z = z.p                                    // case 2
                LEFT-ROTATE(T, z)                          // case 2
            z.p.color = BLACK                              // case 3
            z.p.p.color = RED                              // case 3
            RIGHT-ROTATE(T, z.p.p)                         // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

What we want to keep in the loop (**invariant**)

1. Node z is red
2. If (z.p is the root) then z.p is black
3. If the tree violates any of the RBT properties then it violates at most one of them and the violation is property 2 or property 4.
    - If property 2 violation
    - → because z is the root and is red.
    - If property 4 violation
    - → because both z and z.p are red.

Insertion

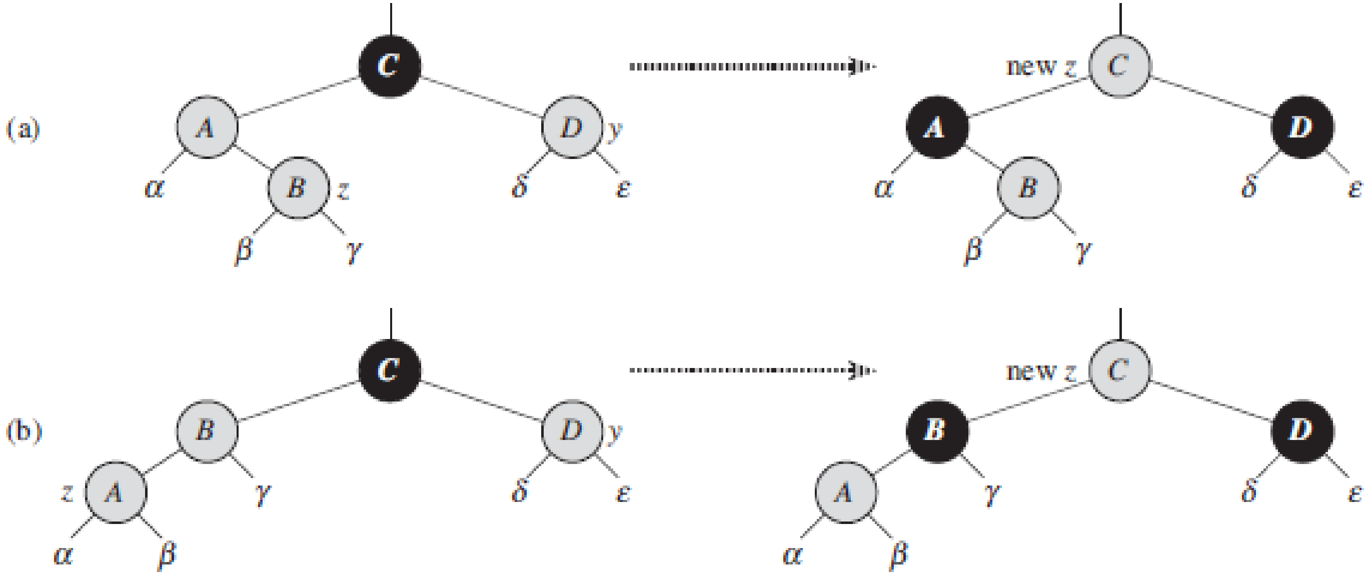
- **Case 1:**
  - z's uncle y is red
- **Case 2:**
  - z's uncle y is black and z is a right child
- **Case 3:**
  - z's uncle y is black and z is a left child

Insert - Case 1

- Violation of Property 4 (if a node is red, both children are black)
    - z and z.p : red

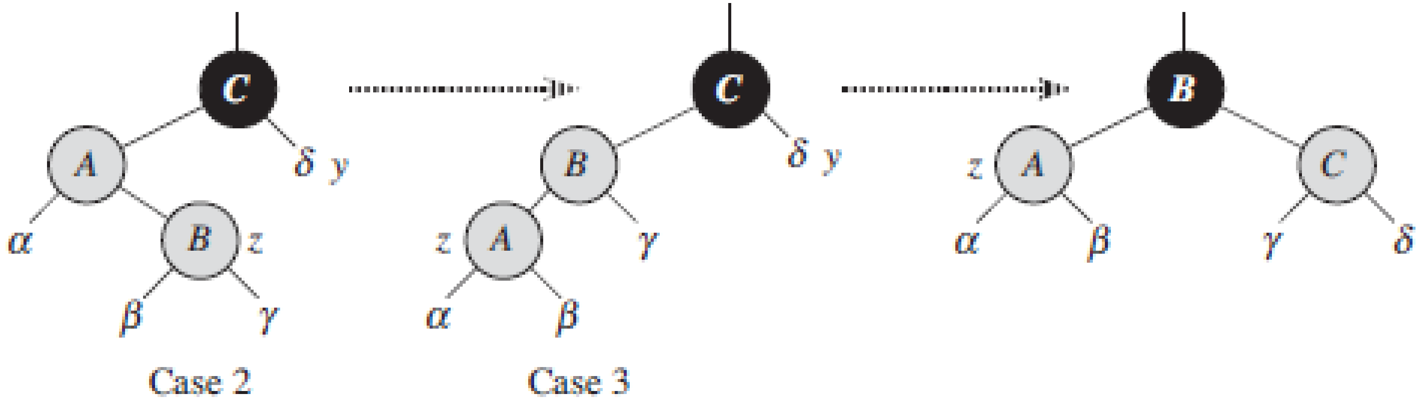
z is **right** child

z is **left** child



Insert - Case 2 & 3

- Violation of Property 4
  - z and z.p: both red ☹



Each of the subtrees α, β, and γ has a black root

- α, β, and γ from property 4
- δ because case 1 otherwise

each has the same bh (black height)

Transform Case 2 → Case 3 by a **left rotation** (preserves property 5)

  - all downward simple paths from a node to a leaf have the same number of blacks.

- Case 3 causes some **color changes** and a **right rotation (**preserve property 5).

The **while** loop then terminates because property 4 is satisfied == no longer 2 red nodes in a row.

Insert

- Complexity
  - Height of a RBT on n nodes = O(log n)
  - → RB-INSERT take O(log n) in time.
  - RB-INSERTFIXUP,
    - **while** loop repeats **only if**
      - case 1 happens and then the pointer z moves 2 levels up the tree.
      - → total number of times the **while** loop can be executed is O(log n)
    - → O(log n)time.
  - It never performs more than 2 rotations
    - because since the **while** loop terminates if case 2 or case 3 is executed!!

Delete

- Transplant function
  - It replaces one subtree as a child of its parent with another subtree
    - replaces the subtree rooted at node **u** with the subtree rooted at node **v**
  - /!\ Reference to T.nil , not nil
    - Use of a sentinel

```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
        T.root = v
elseif u == u.p.left
        u.p.left = v
else u.p.right = v
v.p = u.p
```

- Difference with BST delete
  - Keep track of potential issues for violation of the RBT properties

Delete

- Main idea
  - Delete node z
    - If (z has fewer than 2 children) then
      - z is removed from the tree
    - y = z
    - If (z has 2 children) then
      - y should be z’s successor
      - y moves into z’s position in the tree.
      - Remember y’s color before it is removed from or moved within the tree
      - Keep track of the node x that moves into y’s original position in the tree
        - since node x might also cause violations
    - After deleting node z,
      - RB-DELETE calls RB-DELETE-FIXUP
        - changes colors and performs rotations to restore the RBT properties.

Delete

- Pseudo code
  - O(log(n))

RB-Transplant(T,u,v):
replaces the subtree rooted at node **u** with the subtree rooted at node **v**

```
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
    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)
```

Delete

- If (y.c==red)
  - ➢RBT properties are respected when y is removed because:

1. No bh in the tree have changed.
2. No red nodes have been made adjacent.
   - ➢Because y takes z's place in the tree, along with z's color
   - ➢→ cannot have 2 adjacent red nodes at y's new position in the tree.
   - ➢if y was not z's right child then y's original right child x replaces y in the tree
   - ➢If (y.c==red) then x.c==black
     - o so replacing y by x cannot cause 2 red nodes to become adjacent!
3. As y could not have been the root if it was red
   - ➢→ root remains black.

Delete

- If (y.c==black)
  - ➢ Remove y → change hb
  - ➢ Problems to fix the tree ! ☹

1. If y had been the root and a red child of y becomes the new root
   - ➢ → violation of property 2.
2. If (x.c==red) and (x.*p.c*==red)
   - ➢ → violation of property 4.
3. Moving y within the tree causes any simple path that previously contained y to have 1 fewer black node
   - ➢ → violation of property 5 (by any ancestor of y in the tree!)
   - ➢ Correction
     - o say that node x, now occupying y's original position, has an "extra" black
     - o if (add 1 to the count of black nodes on any simple path that contains x)
     - o Then under this interpretation, property 5 holds. 
     - o When we remove or move the black node y → we "push" its blackness onto node x.
   - ➢ Problem:
   - ➢ now node x is neither red nor black → violation of property 1.

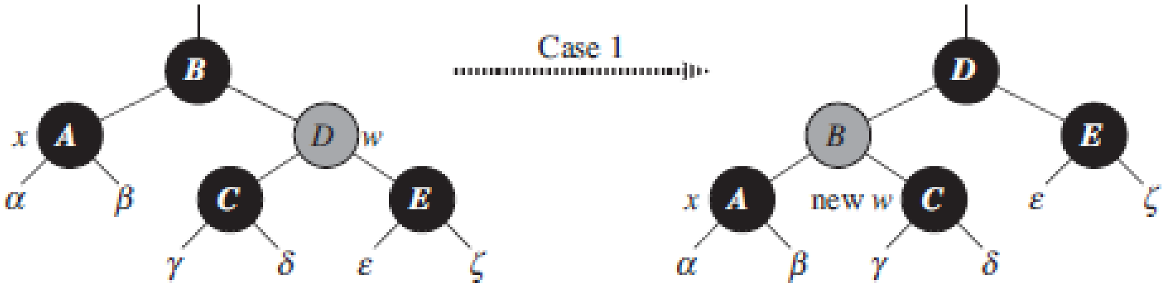
Delete

- Delete Fixup
  - Pseudo-code

```
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                                   // case 1
            x.p.color = RED                                   // case 1
            LEFT-ROTATE(T, x.p)                               // case 1
            w = x.p.right                                     // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                     // case 2
            x = x.p                                           // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                          // case 3
                w.color = RED                                 // case 3
                RIGHT-ROTATE(T, w)                            // case 3
                w = x.p.right                                 // case 3
            w.color = x.p.color                               // case 4
            x.p.color = BLACK                                 // case 4
            w.right.color = BLACK                             // case 4
            LEFT-ROTATE(T, x.p)                               // case 4
            x = T.root                                        // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

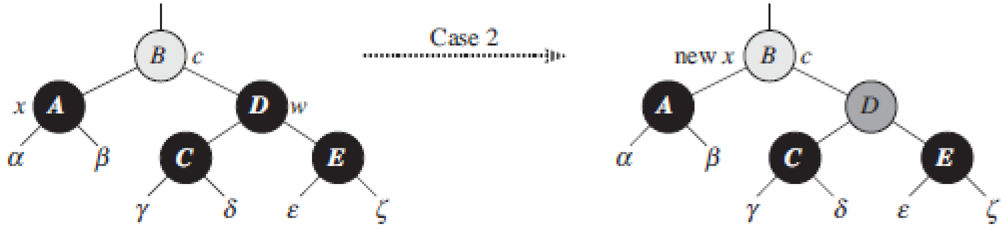
Delete (Fixup)

- **Case 1**:
  - It happens when
    - node w (sibling of x is red) w.c==red
  - w must have black children
    - we can **switch** the colors of w and x.*p*
    - Perform a **left-rotation** on x.*p* without violating any RBT properties
  - New sibling of x (1 of w's children prior to the rotation): **black**
  - case 1 → case 2, 3, or 4.



Delete (Fixup)

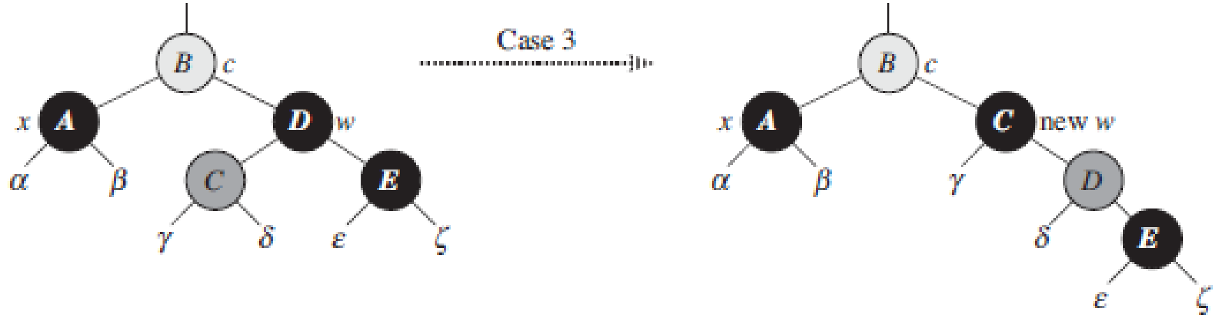
- **Case 2**:
  - It happens when
    - (x’s sibling w.c==black) and (w.right.c==black) and (w.left.c==black)
  - As w is black → take one black off both x and w
    - leaving x with only 1 black and leaving w red.
  - Compensate for removing 1 black from x and w
    - Add an extra black to x.p that was red or black
    - How?
      - by repeating the **while** loop with x.p as the new node x.
      - If we enter **Case 2** through **Case 1**, the new node x is red-and-black, since the original x.p was red.



- → color of the new node x is red
  - and the loop terminates when it tests the loop condition.

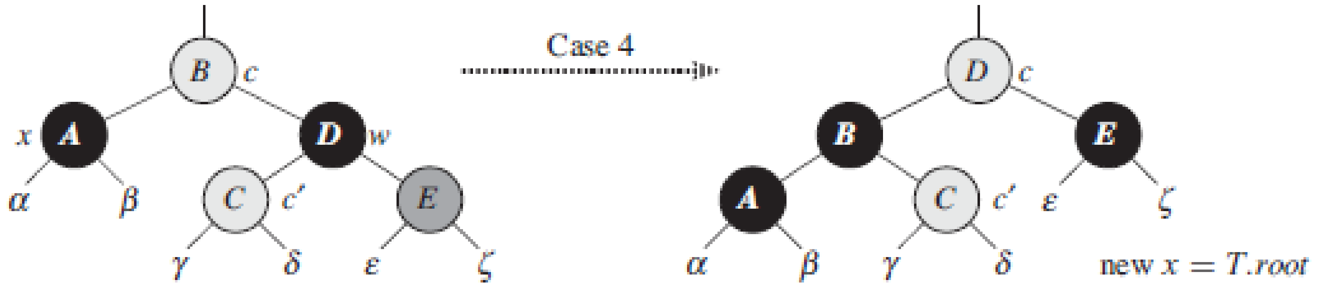
Delete (Fixup)

- **Case 3**:
  - It happens when
    - w.c==black and w.left.c==red and w.right.c==black
  - Switch the colors of w and w.left
  - Then perform a **right rotation** on w without
    - violating any of the RBT properties
- The new sibling w of x is now a **black** node
  - with a red right child
  - Transform Case 3 → case 4.



Delete (Fixup)

- **Case 4**:
  - It happens when
    - (x's sibling w.c==black) and (w.right.c==red)
  - Some color changes + left rotation on x.p
    - Remove the extra black on x, making it singly black
      - without violating any of the RBT properties.
  - Setting x to be the root causes the while loop to terminate when it tests the loop condition.



Delete

- Complexity analysis
  - height of a RBT of n is O(log n),
    - total cost of the procedure without the call to RB-DELETEFIXUP is log n time
  - Within RB-DELETE-FIXUP
    - Cases 1, 3, and 4:
      - lead to termination after performing a **constant number of color changes** and
      - + At most 3 rotations
    - Case 2 is the **only** case where the **while** loop can be repeated
      - the pointer x moves **up** the tree at most O(log n) times,
        - no rotations!
  - → RB-DELETE-FIXUP
    - takes O(log n) time
    - Performs at most 3 rotations
  - Total time for RB-DELETE = O(log n)

Conclusion

- Red-Black trees
  - A form of semi-balanced binary tree
  - Compared to other trees
    - more general purpose: add, remove, and look-up
    - but AVL trees have faster look-ups at the cost of slower add/remove !
- Applications
  - Linux kernel
    - The anticipatory, deadline, and CFQ I/O schedulers: RBT to track requests
    - The packet CD/DVD driver: RBT trees
    - The high-resolution timer uses an RBT to organize outstanding timer requests
    - The ext3 file system tracks directory entries in an RBT
    - Virtual memory areas (VMAs) are tracked with RBTs

- Complexity

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Questions ?

- Acknowledgment + Reading
  - ➢Csci 115 book Section 7.4
  - ➢Chapter 13, Red-Black Trees, Introduction to Algorithms, 3^{rd} Edition.