香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 11: Red-black tree

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Red-Black tree

- A "balanced" binary search tree
  - It guarantees an O(logn) running time for many operations, such as search, insertion, and deletion

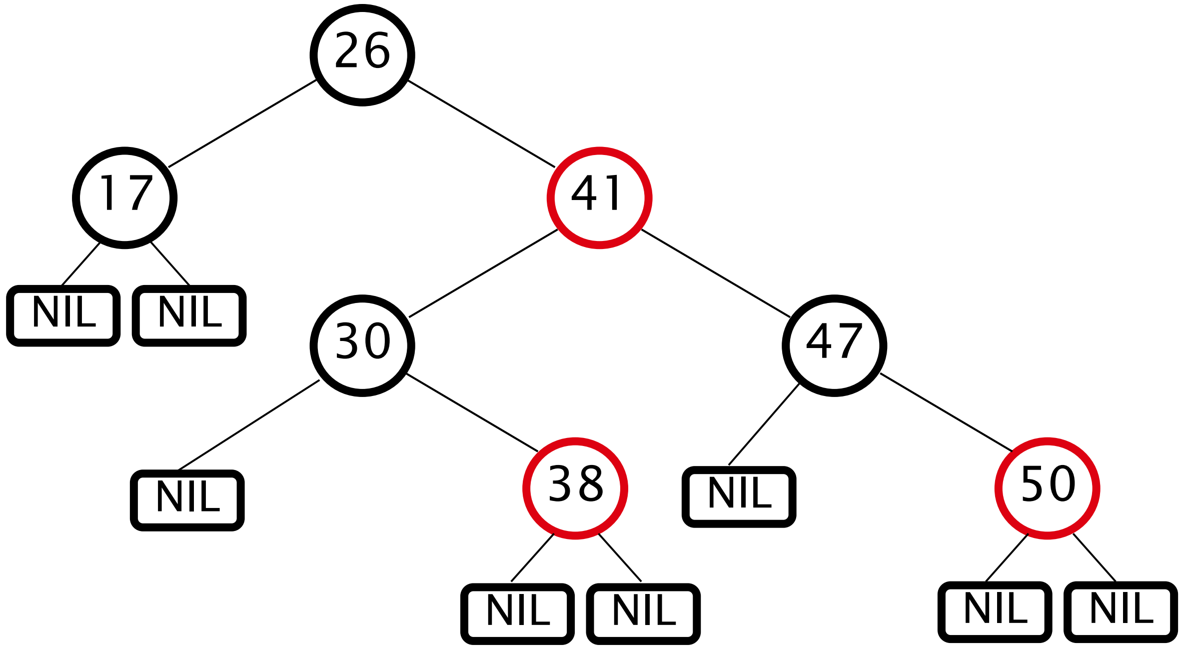
- Red-black tree
  - A binary search tree has an additional attribute for its nodes: color which can be **red** or **black**
  - Restrict the way nodes can be colored on any path from the root to a leaf
  - Ensures that no path is more than twice as long as any other path

Red-Black tree 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**

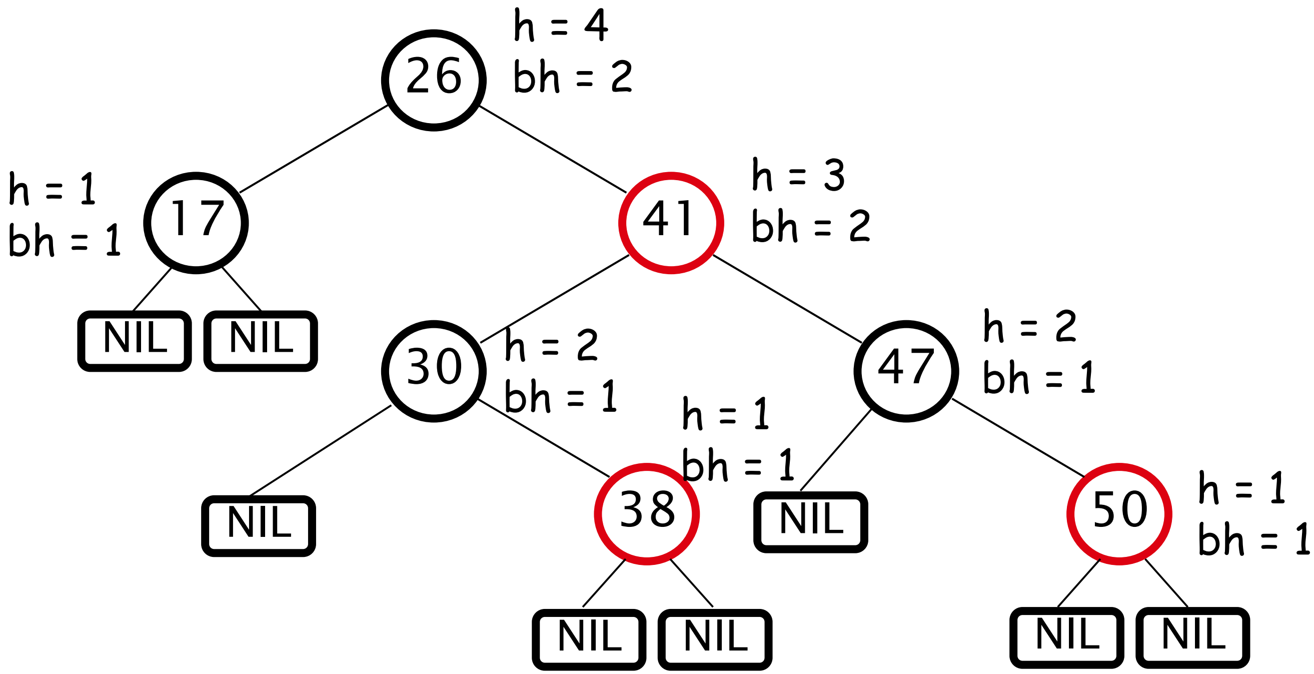
   No two consecutive red nodes on a simple path from the root to a leaf
5. For each node, all paths from that node to descendant leaves contain the same number of **black** nodes

Example



- For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leaves
  - NIL[T] has the same fields as an ordinary node
  - Color[NIL[T]] = BLACK
  - The other fields may be set to arbitrary values

Black height of a node



- **Height of a node:**
  - The number of edges in the **longest** path to a leaf
- **Black-height** of a node x:
  - bh(x) is the number of black nodes (including NIL) on the path from x to a leaf, <u>not counting x</u>

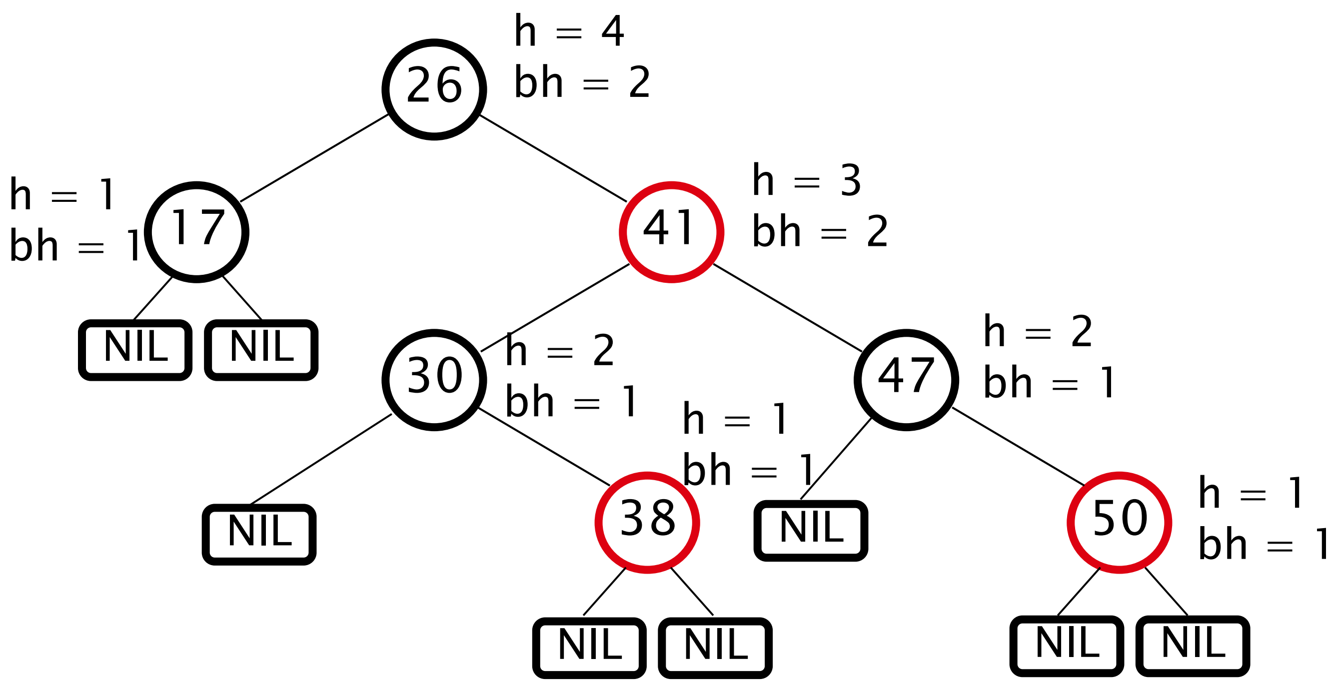
Important property of Red-Black tree

A red-black tree with n internal nodes has height at most $2\log(n + 1)$

- Need to prove two claims first …

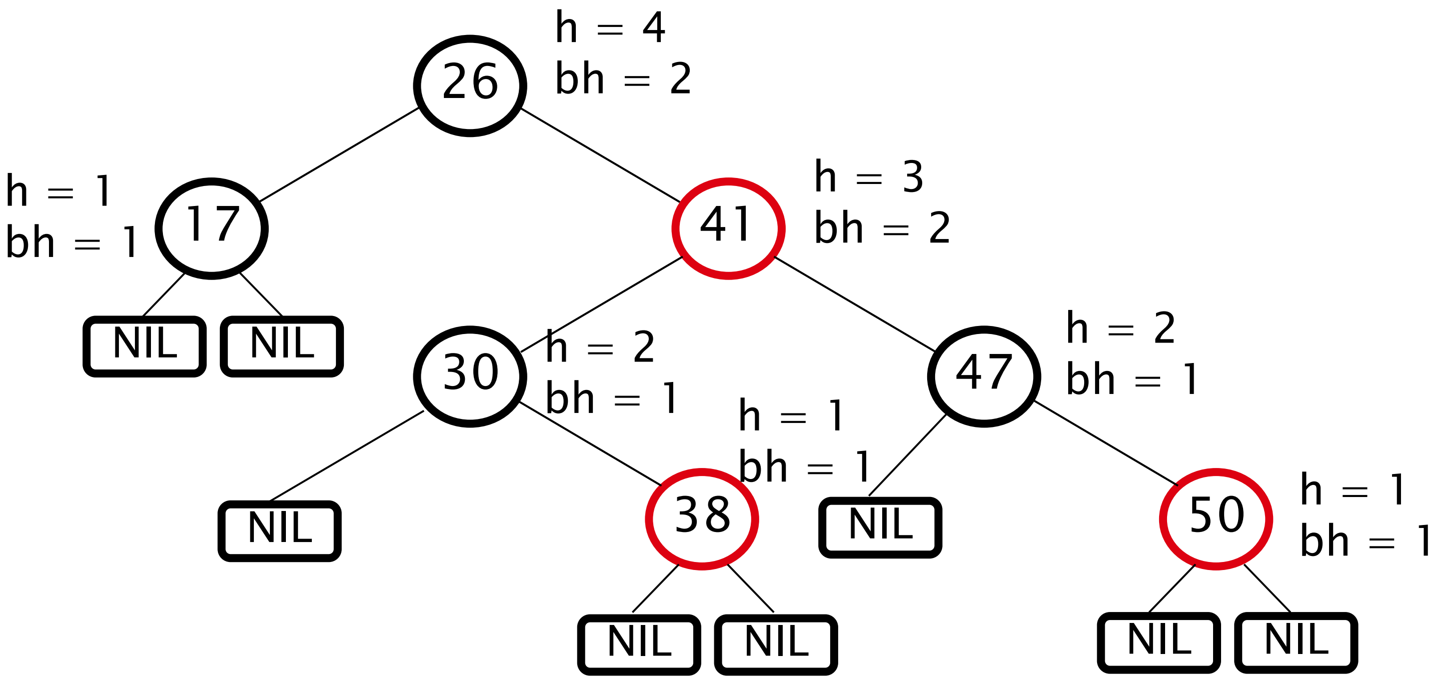
Claim 1

- Any node x with height h(x) has $bh(x) \geq h(x)/2$
- **Proof**
  - By property 4, at most h/2 **red** nodes on the path from the node to a leaf
  - Hence at least h/2 are **black**



Claim 2

- The subtree rooted at any node x contains **at least** $2^{bh(x)} - 1$ internal nodes



Claim 2 (Cont'd)

Proof: By induction on **h[x]**

Basis: h[x] = 0 $\Rightarrow$

x is a leaf (NIL[T]) $\Rightarrow$

bh(x) = 0 $\Rightarrow$

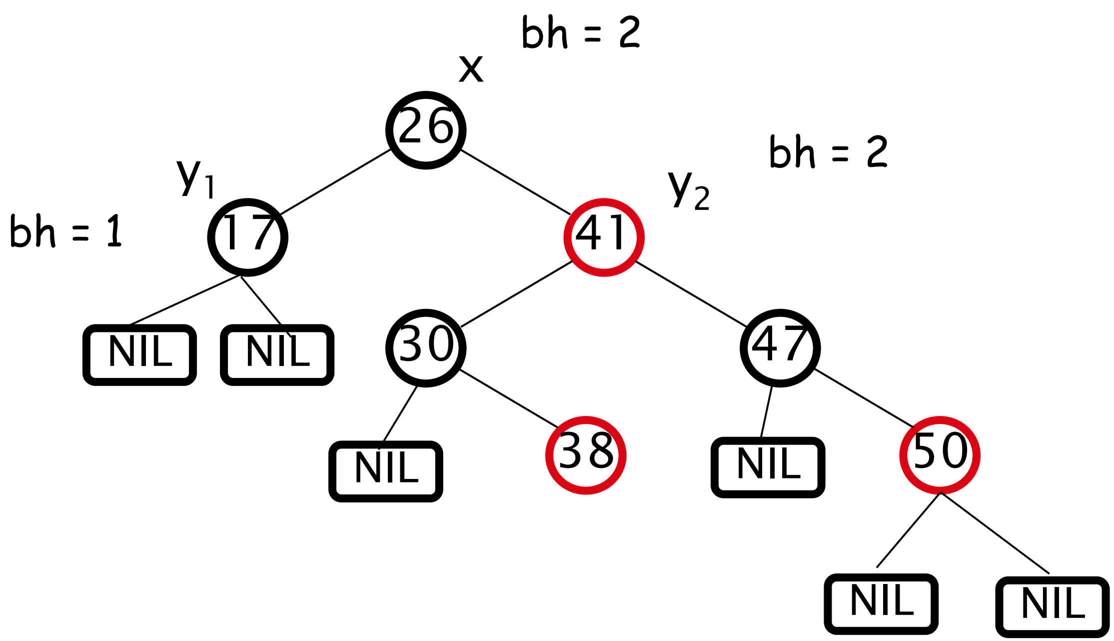
# of internal nodes: $2^0 - 1 = 0$

Inductive Hypothesis: assume it is true for h[x]=h-1

Claim 2 (Cont'd)

Inductive step:

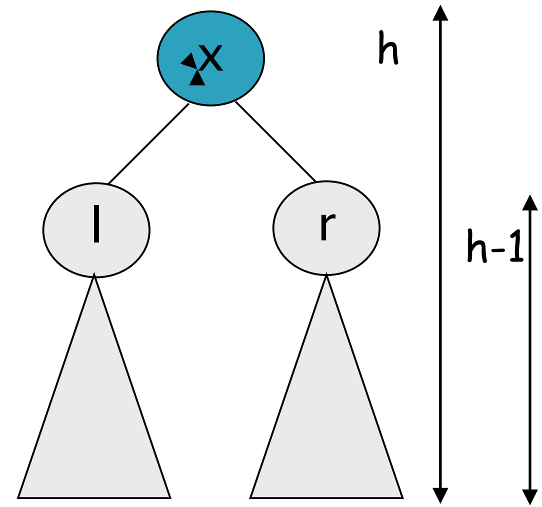
- Prove it for h[x]=h
- Let bh(x) = b, then any child y of x has:
  - bh (y) = b (if the child is **red**), or
  - bh (y) = b - 1 (if the child is **black**)



Claim 2 (Cont'd)

- Using inductive hypothesis, the number of internal nodes for each child of x is at least (if it is black):

$$2^{bh(x)-1} - 1$$



- The subtree rooted at x contains at least:

$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 =$

$2 \cdot (2^{bh(x)-1} - 1) + 1 =$

$2^{bh(x)} - 1$ internal nodes

$bh(l) \geq bh(x)-1$

$bh(r) \geq bh(x)-1$

Important property of Red-Black tree

A red-black tree with n internal nodes has height <u>at most</u> $2\log(n + 1)$
Proof in the next slides.

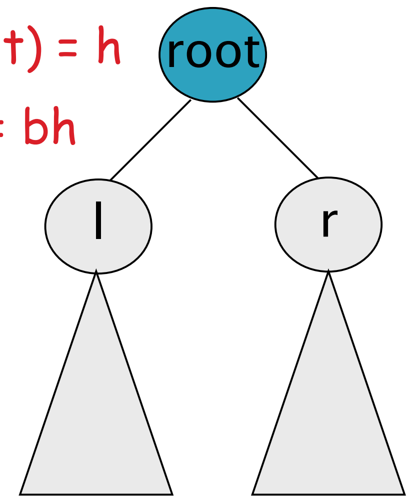
- Claim 1: Any node x with height h(x) has $bh(x) \geq h(x)/2$
- Claim 2: The subtree rooted at any node x contains **at least** $2^{bh(x)} - 1$ internal nodes

Heights of Red-Black tree

Lemma: A red-black tree with n internal nodes has height at most 2log(n + 1).

height(root) = h

bh(root) = bh



Proof:

$$n \geq 2^{bh} - 1 \geq 2^{h/2} - 1$$

number **n** of internal nodes

since bh ≥ h/2

- Add 1 to both sides and then take logs:

$$n + 1 \geq 2^{bh} \geq 2^{h/2}$$

$$\lg(n + 1) \geq h/2 \Rightarrow$$

$$h \leq 2 \lg(n + 1)$$

Exercise 1

- What is the ratio between the longest path and the shortest path in a red-black tree?

  - The shortest path is at least bh(root)
  - The longest path is equal to h(root)
  - We know that h(root)≤2bh(root)

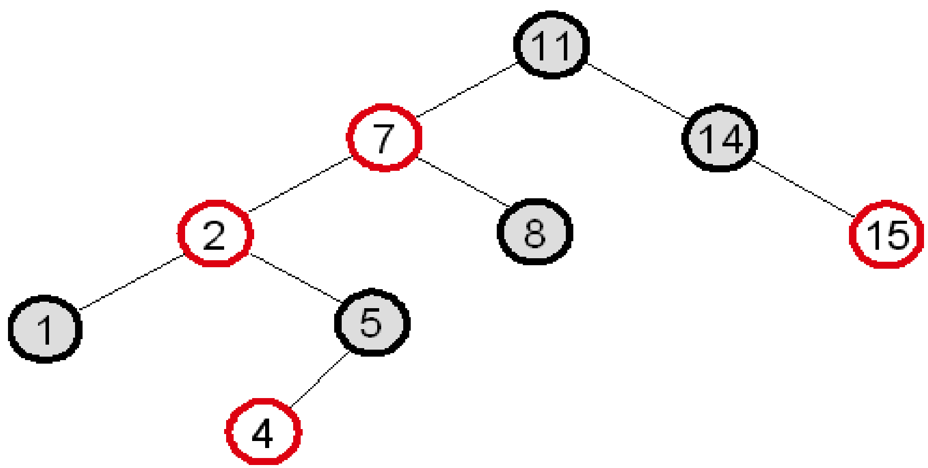
  - Therefore, the ratio is ≤2

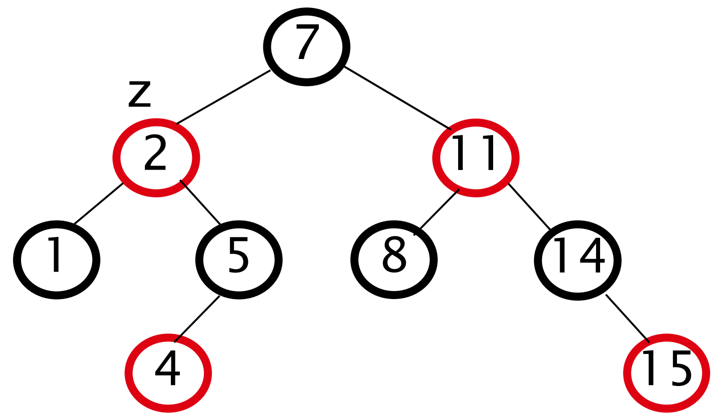
Exercise 2

- What is the largest possible number of internal nodes in a red-black tree with black-height k?
- Can all the nodes be black in a red-black tree?

Exercise 3

- What red-black tree property is violated in the tree below? How would you restore the red-black tree property in this case?
  - Property violated: if a node is red, both its children are black
  - Fixup: color 7 black, 11 red, then right-rotate around 11





Operations of Red-black tree

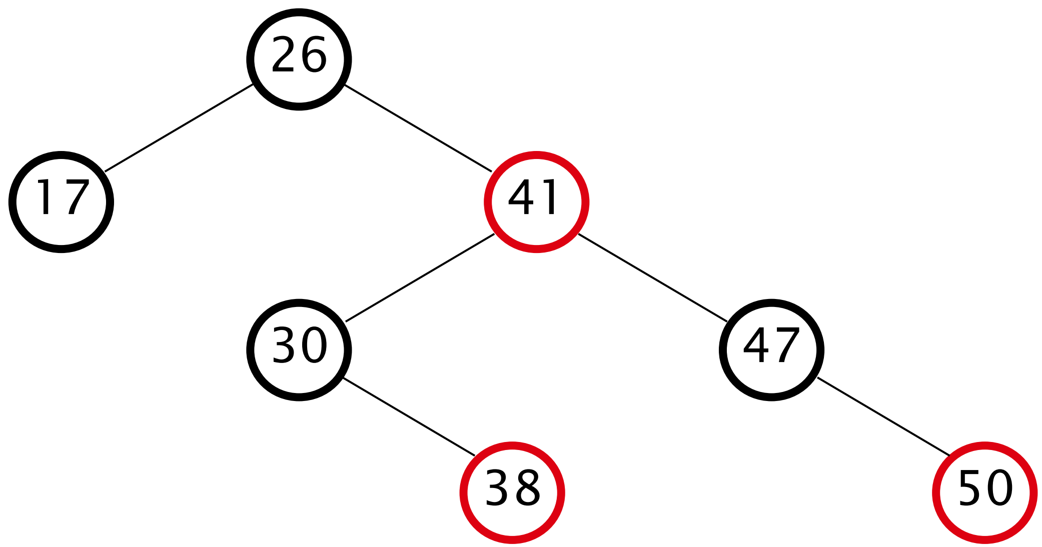
- The non-modifying operations: MINIMUM, MAXIMUM, and SEARCH run in O(h) time
  - They take O(logn) time on red-black trees
  - SEARCH is similar to the search on binary search tree

- What about INSERT and DELETE?
  - They will still run in O(logn) time
  - We have to guarantee that the modified tree will still be a red-black tree

INSERT operation

INSERT: Suppose we want to insert 35. What color to make the new node?

- Red?
  - Property 4 is violated: if a node is red, then both its children are black
- Black?
  - Property 5 is violated: all paths from a node to its leaves contain the same number of black nodes



Rotations

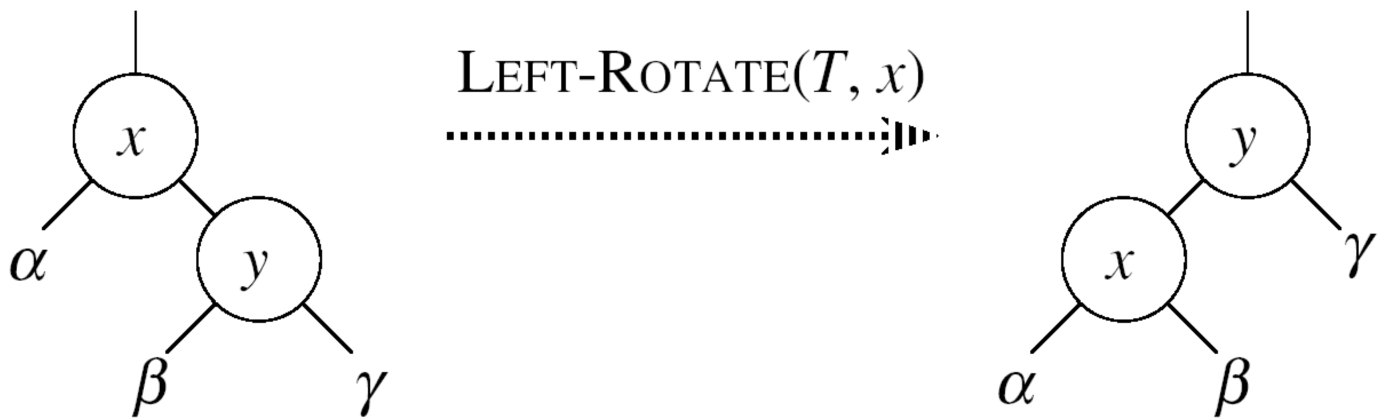
- Operations for re-structuring the tree after insert and delete operations on red-black trees

- Rotations take a red-black tree and a node within the tree and:
  - Together with some node re-coloring they help restore the red-black-tree property
  - Change some of the pointer structure
  - **Do not** change the binary-search tree property

- Two types of rotations:
  - Left & right rotations

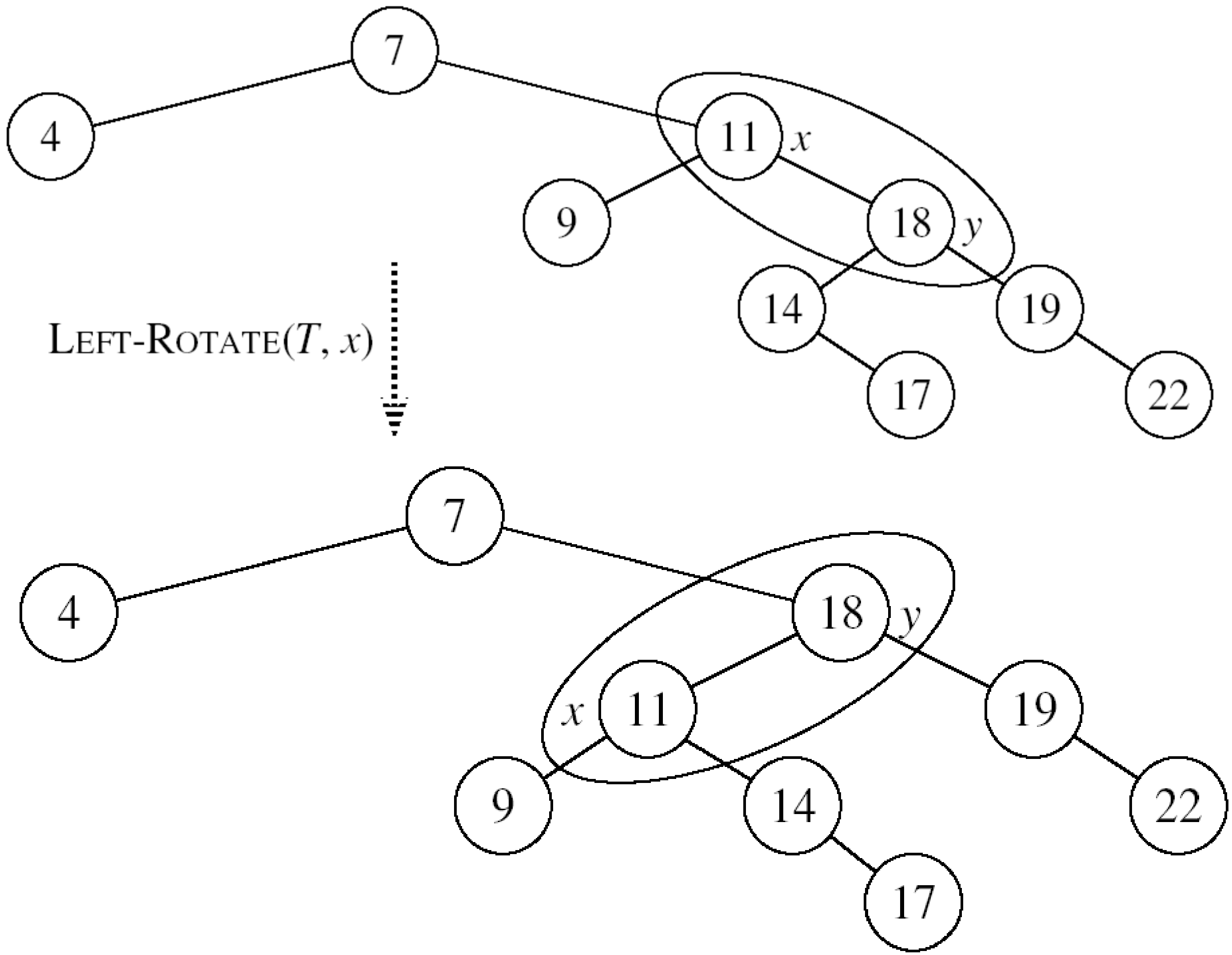
Left rotation

- Assumptions for a left rotation on a node x:
  - The right child of x (y) is not NIL



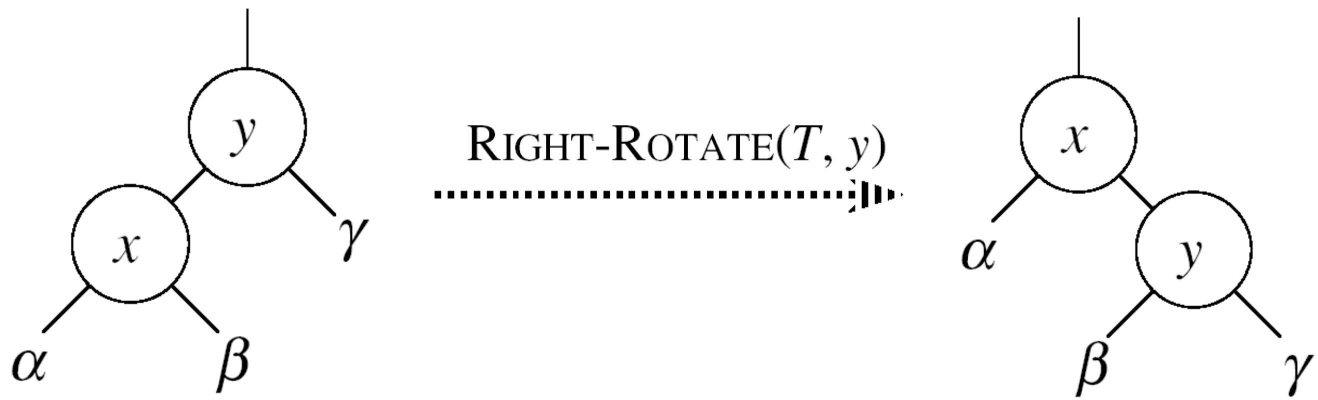
- Idea:
  - Pivots around the link from x to y
  - Makes y the new root of the subtree
  - x becomes y's left child
  - y's left child becomes x's right child

Example



Right rotation

- Assumptions for a right rotation on a node x:
  - The left child of y (x) is not NIL



- Idea:
  - Pivots around the link from y to x
  - Makes x the new root of the subtree
  - y becomes x's right child
  - x's right child becomes y's left child

INSERT

- Goal:
  - Insert a new node z into a red-black-tree

- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black-tree properties
    - Use an auxiliary procedure RB-INSERT-FIXUP

Properties affected by INSERT

1. Every node is either **red** or **black** OK!
2. The root is **black**

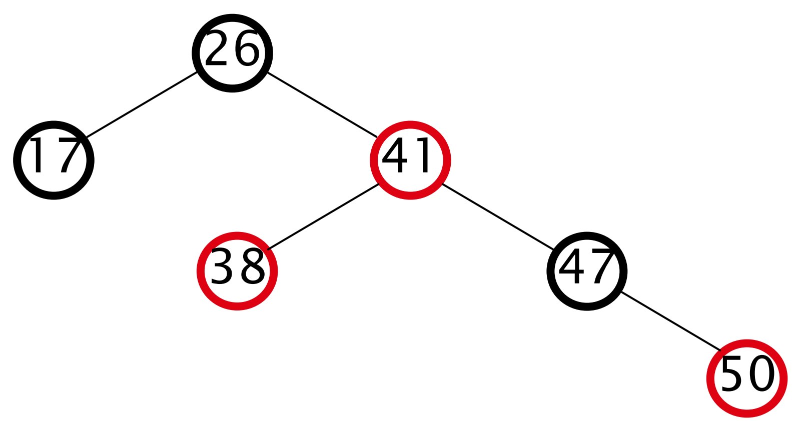
If the root is changed
⇒ May not OK

3. Every leaf (NIL) is **black** OK!
4. If a node is **red**, then both its children are **black**

If p(z) is red ⇒ not OK
z and p(z) are both red

OK!

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes



INSERT(T,z)

```
y ← NIL
x ← root[T]
```

- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

```
while x ≠ NIL
        do y ← x
                if key[z] < key[x]
                    then x ← left[x]
                else x ← right[x]
```

- Go down the tree until reaching a leaf
- At that point y is the parent of the node to be inserted

```
p[z] ← y
```

Sets the parent of z to be y

```
if y = NIL
    then root[T] ← z
```

The tree was empty: set the new node to be the root

```
    else if key[z] < key[y]
            then left[y] ← z
            else right[y] ← z
```

Otherwise, set z to be the left or right child of y, depending on whether the inserted node is smaller or larger than y's key

```
left[z] ← NIL
right[z] ← NIL
color[z] ← RED
```

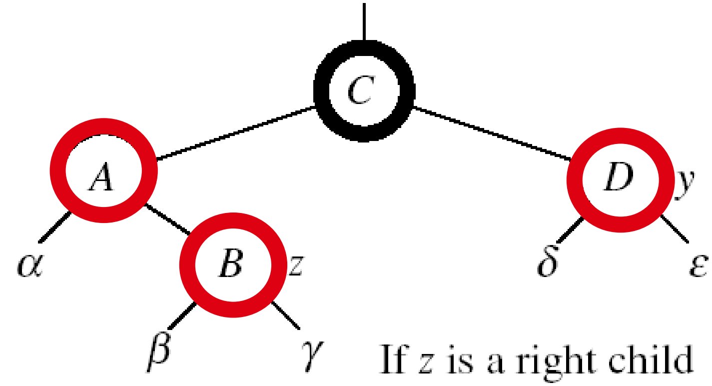
Set the fields of the newly added node

```
RB-INSERT-FIXUP(T, z)
```

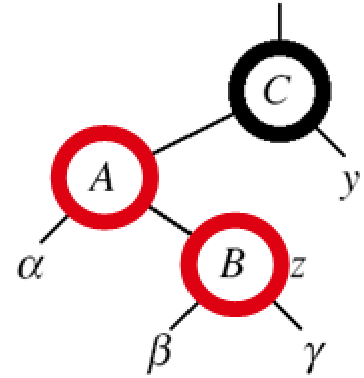
Fix any inconsistencies that could have been introduced by adding this new red node

RB-Insert-Fixup(T, z)

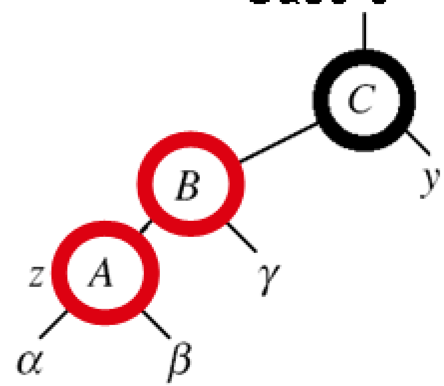
- Case 1: z's uncle y is **red**
  - Solution: recolor
- Case 2: z's uncle y is **black** and z is a right child
  - Solution: double rotation
  - Can be transferred to Case 3
- Case 3: z's uncle y is **black** and z is a left child
  - Solution: single rotation



Case 2



Case 3



RB-Insert-Fixup(T, z)

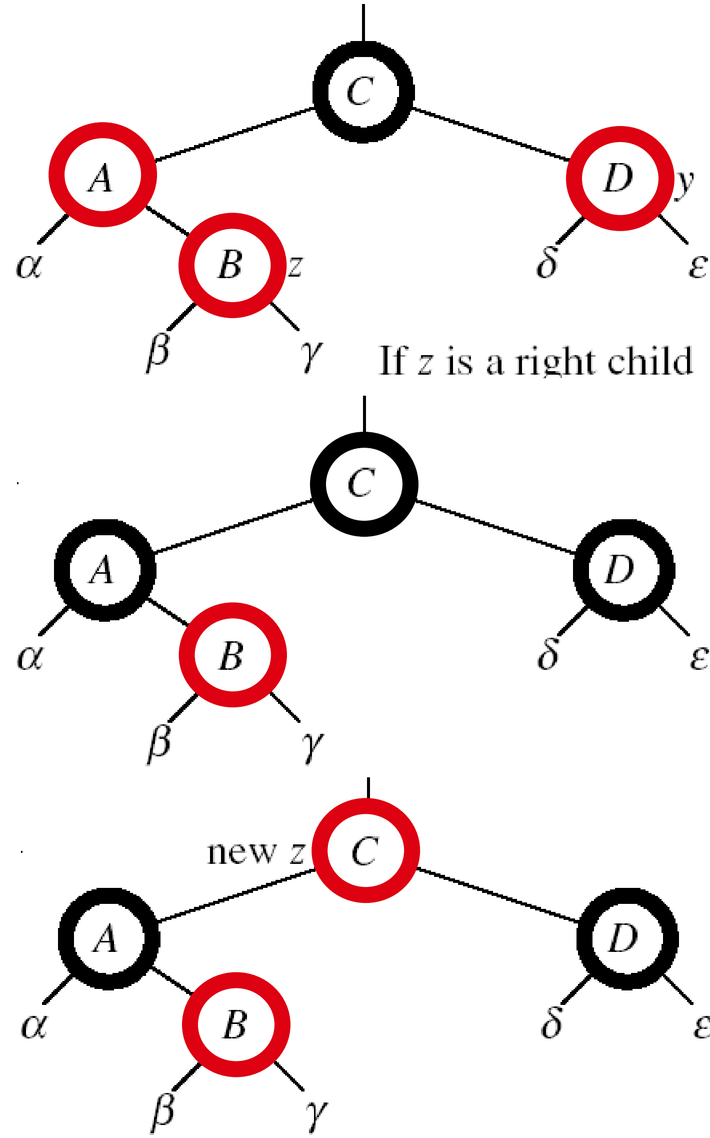
```
while z.p.color == red          The while loop repeats only when
                                Case 1 is executed: O(lgn) times
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == red
            z.p.color = black                   // case 1
            y.color = black                     // case 1
            z.p.p.color = red                   // case 1
            z = z.p.p                           // case 1
        else if z == z.p.right
                z = z.p                         // case 2
                Left-rotation (T, z)            // case 2
            z.p.color = black                   // case 3
            z.p.p.color = red                   // case 3
            Right-rotation (T, z.p.p)           // case 3
    else (same as then clause with "right" and "left" exchanged)
T.root.color = black        may just insert the root or the red violation reach root
```

INSERT - Case 1

z's "uncle" (y) is **red**

Idea: (<u>z is a right</u>)

- p[p[z]] (z's grandparent) must be black: p[z] is red
- Color p[z] **black**
- Color y **black**
- Color p[p[z]] **red**
- z = p[p[z]]
  - Push the **"red"** violation up the tree

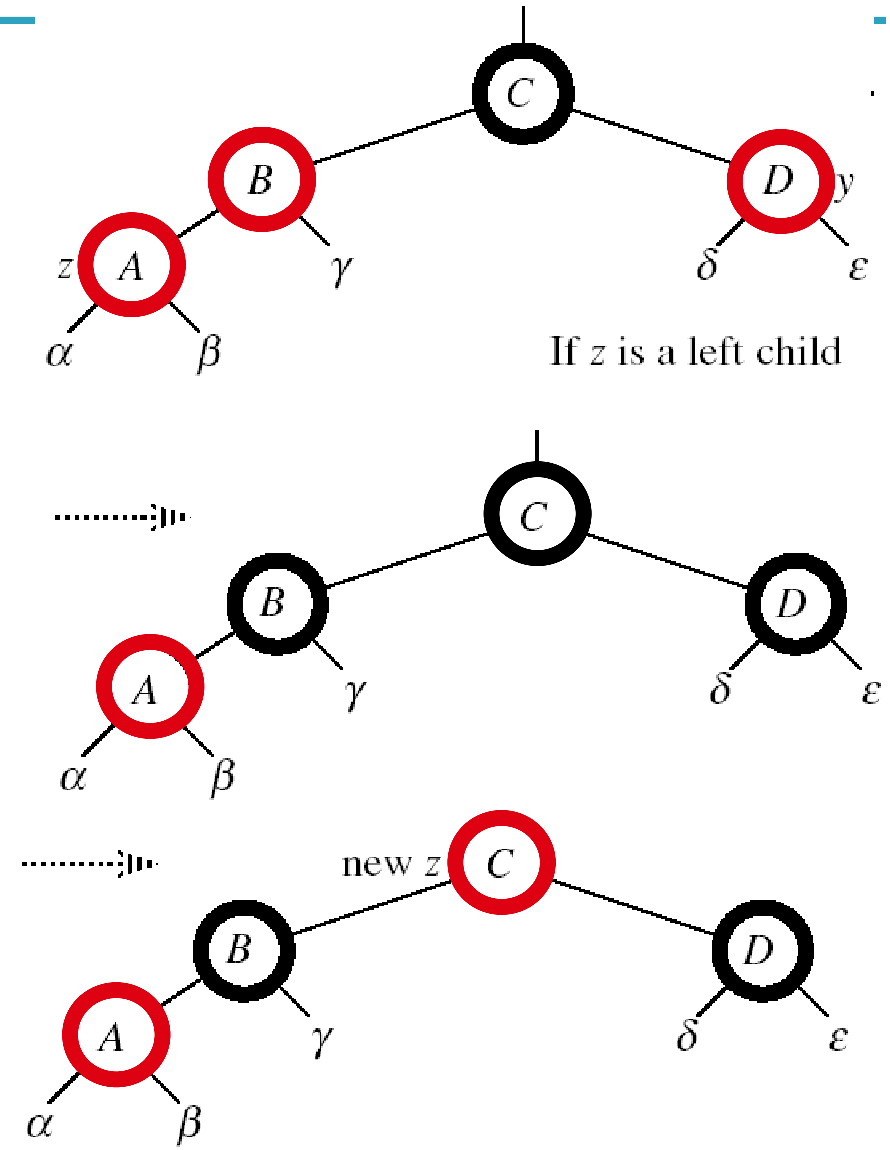


INSERT - Case 1

z's "uncle" (y) is **red**

Idea: (z is a left child)

- p[p[z]] (z's grandparent) must be black: p[z] is red
- Color p[z] ← **black**
- Color y ← **black**
- Color p[p[z]] ← **red**
- z = p[p[z]]
  - Push the **"red"** violation up the tree



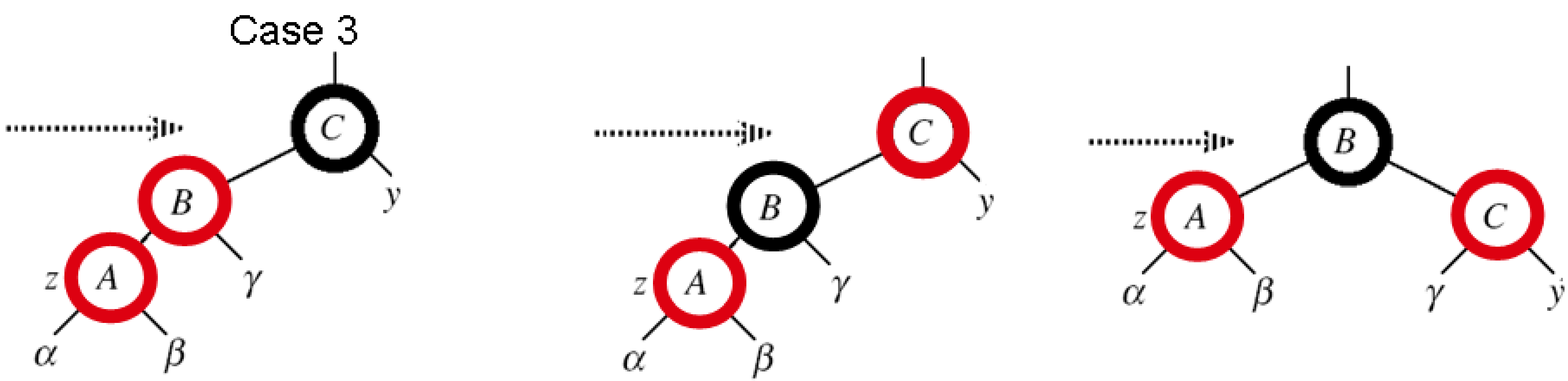
INSERT - Case 3

Case 3:

- z's "uncle" (y) is **black**
- z is a left child

Idea:

- Color p[z] ← **black**
- Color p[p[z]] ← **red**
- RIGHT-ROTATE(T, p[p[z]])
- No longer have 2 reds in a row
- p[z] is now black



INSERT - Case 2

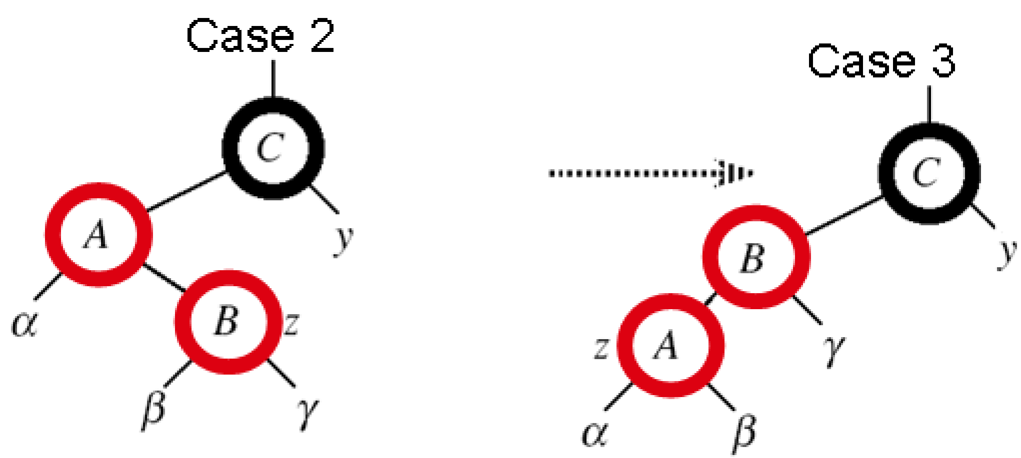
Case 2:

- z's "uncle" (y) is **black**
- z is a right child

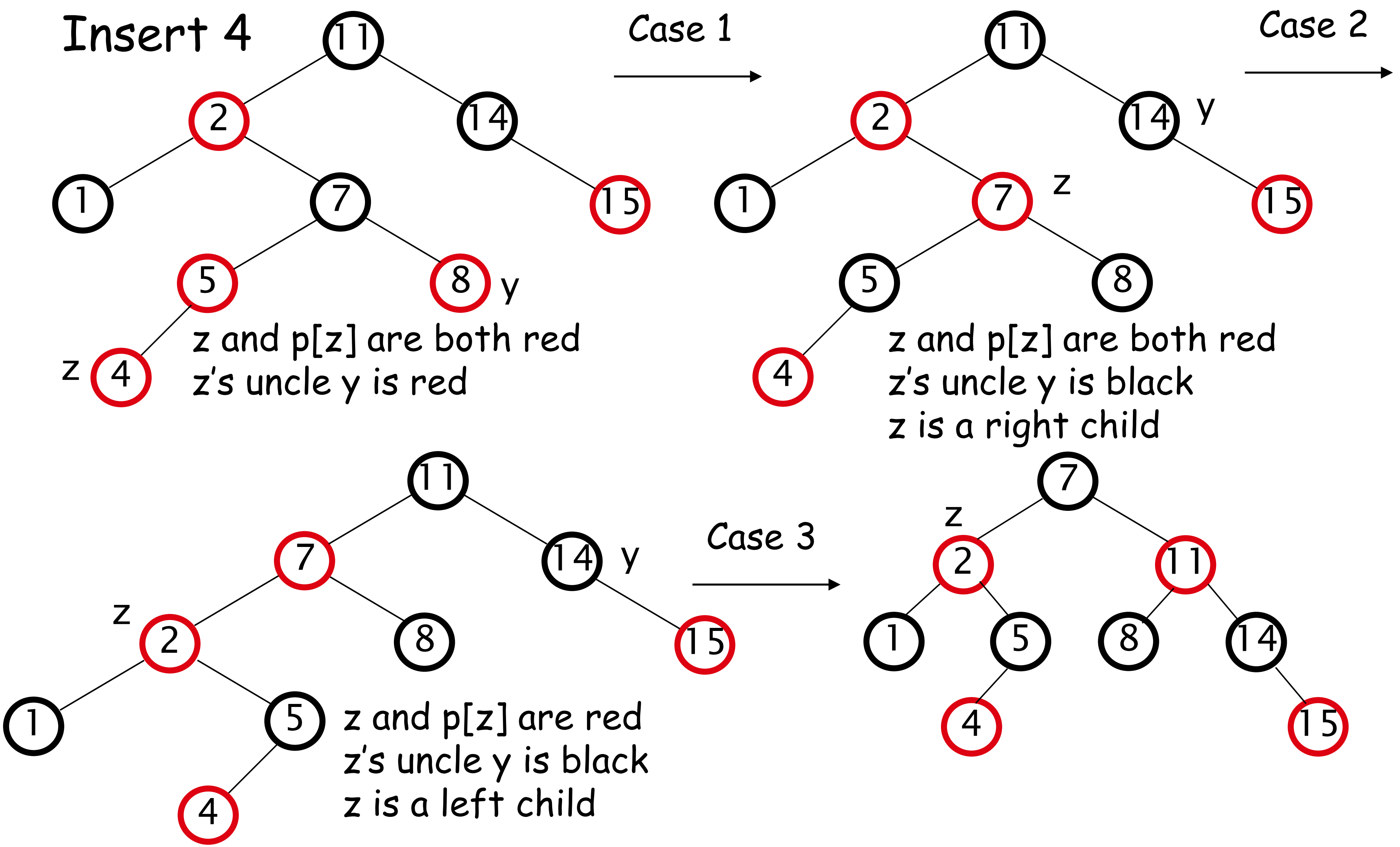
Idea:

- $z \leftarrow p[z]$
- LEFT-ROTATE(T, z)

$\Rightarrow$ now z is a left child, and both z and p[z] are red $\Rightarrow$ case 3



Example



Insert 4
11
2
14
1
7
15
5
8
y
z
4
z and p[z] are both red
z's uncle y is red
Case 1
11
2
14
y
1
7
z
15
5
8
4
z and p[z] are both red
z's uncle y is black
z is a right child
Case 2
11
7
14
y
z
2
8
15
1
5
4
z and p[z] are red
z's uncle y is black
z is a left child
Case 3
7
z
2
11
1
5
8
14
4
15

Complexity analysis

- **Time complexity of detailed steps**
  - A red-black tree has $O(\log n)$ height
  - Search for insertion location takes $O(\log n)$ time
  - Addition to the node takes $O(1)$ time

  - The while loop will be executed at most $O(\log n)$ time
    - Each recoloring and each rotation take $O(1)$ time
    - Never performs more than two rotations, since the loop terminates if case 2 or case 3 is executed

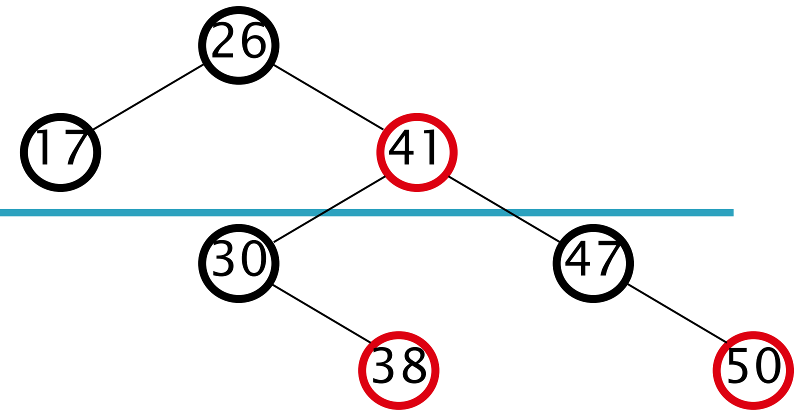
- An insertion in a red-black tree takes $O(\log n)$ time

What are the advantages of red-black tree over balanced BST?

Exercise 4

- When we initially insert a node into a red-black tree, we initially set the color of the new node to red. Why didn't we choose to set the color to black?

DELETE operation

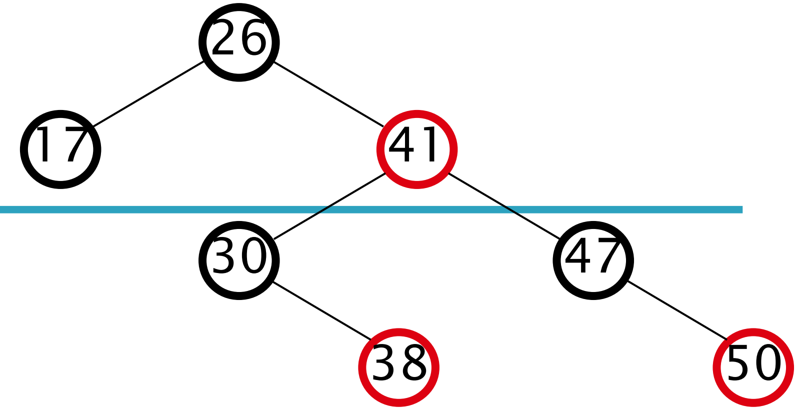


DELETE: the color of the node to be removed -- **red**

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

Note: the deletion of a red node is the same as the deletion of a node in BST

DELETE operation



DELETE: the color of the node to be removed -- **Black**

1. Every node is either **red** or **black** OK!
2. The root is **black**

   Not OK! If removing the root and the child that replaces it is **red**
3. Every leaf (NIL) is **black** OK!
4. If a node is **red**, then both its children are **black**

   Not OK! Could create two red nodes in a row

   Not OK! Could change the black heights of some nodes
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

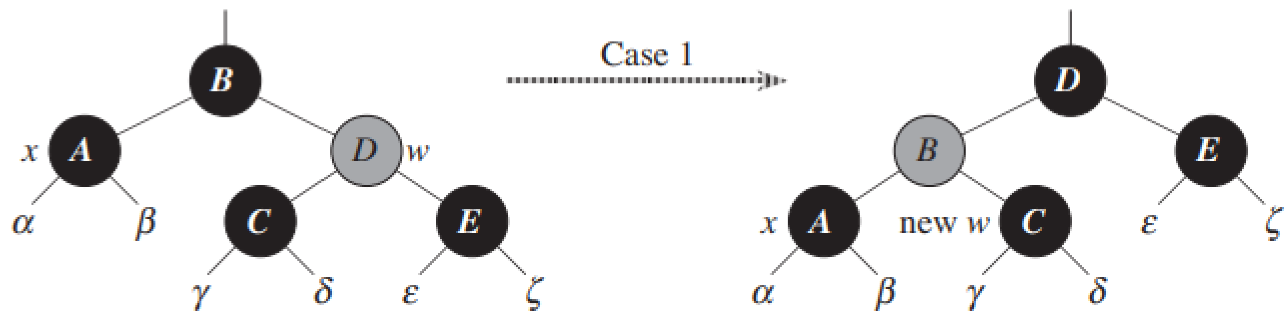
Deletion on red-black tree

- Similar to the deletion on BST, but need to use an auxiliary procedure RB-Delete-Fixup to restore the red-black tree properties

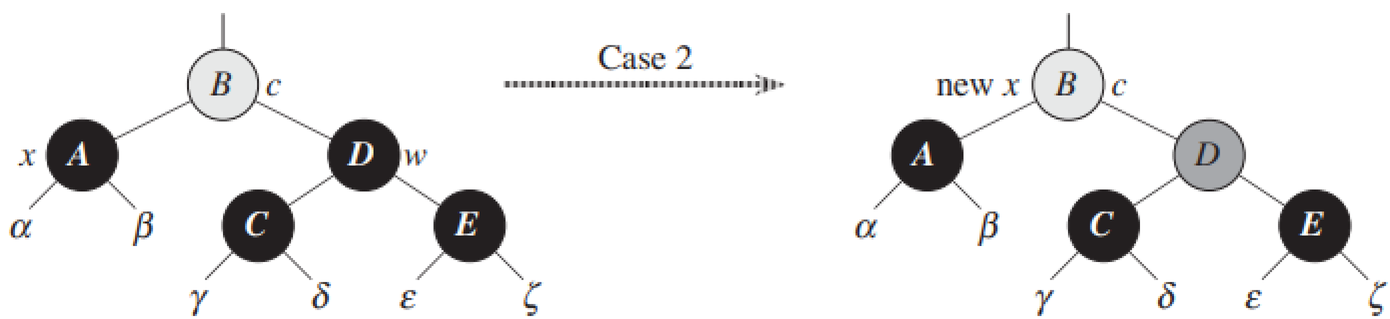
- Four different cases of RB-Delete-Fixup
  - Case 1: x's sibling w is **red**
  - Case 2:x's sibling w is **black**, and both of w's children are **black**
  - Case 3:x's sibling w is **black**, w's left child is **red**, and w's right child is **black**
  - Case 4: x's sibling w is **black**, and w's right child is **red** (left child either color)

Cases 1 and 2

- Case 1: x's sibling w is **red**
  - Solution: rotate and recolor

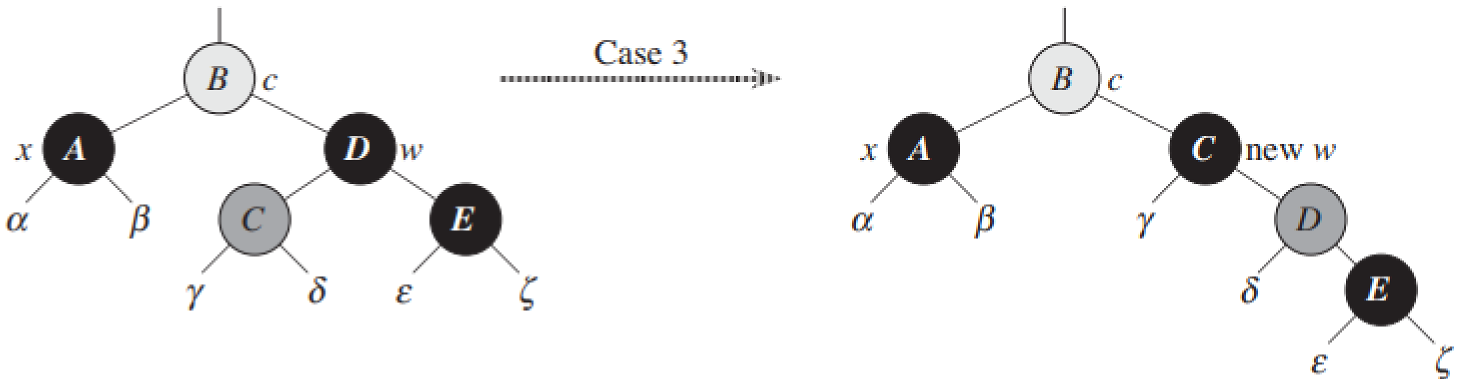


- Case 2:x's sibling w is **black**, and both of w's children are **black**
  - Solution: recolor

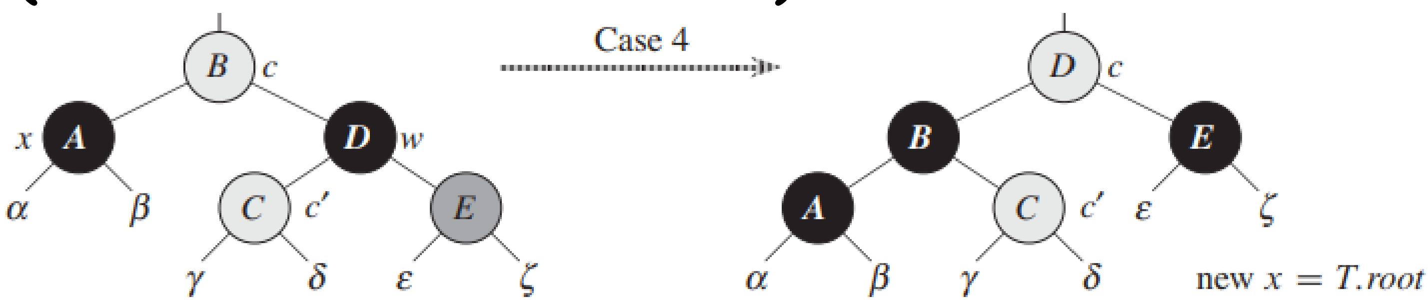


Cases 3 and 4

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



- Case 4: x's sibling w is **black**, and w's right child is **red** (left child either color)



Red-Black Trees - Summary

- Operations on red-black-trees:
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)

- Red-black-trees guarantee that the height of the tree will be O(lgn)

Recommended reading

- Reading
  - Chapter 13, textbook
- Next lectures
  - Sorting
  - Chapter 7&8, textbook