# Lecture 6 – Tree II

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

# Outline

- Binary Search Tree
- Red-Black Tree

# Binary Search Tree

# Dynamic Collection with Tree

Requirement for a dynamic collection, supporting:

- Search
- Find minima
- Find maxima
- Find predecessor
- Find successor

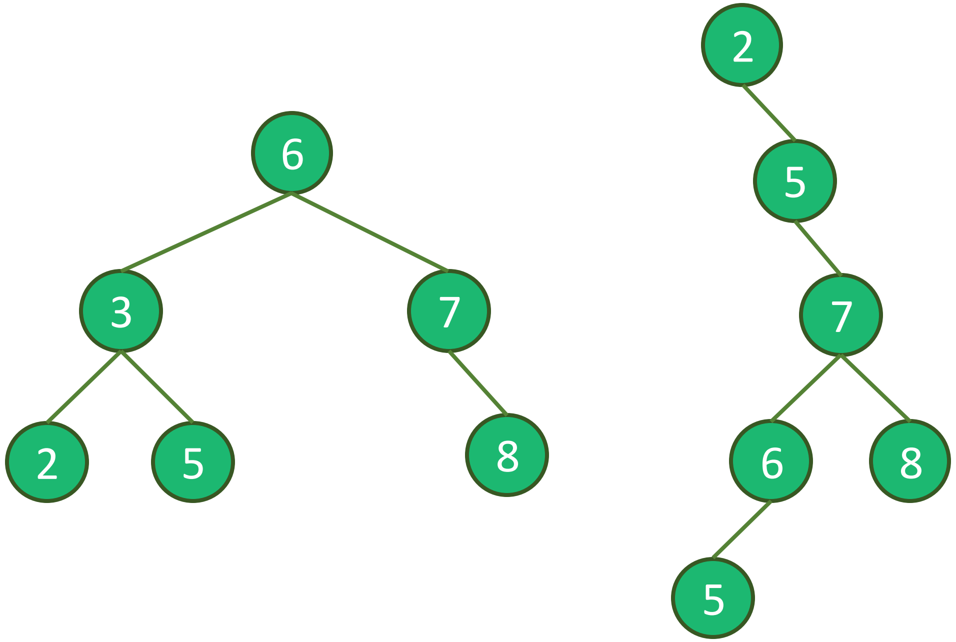
Think: what do we have so far?

- Linear list: slow, $\Theta(n)$
- Hash table: only search by key, no comparison
- Binary search: only static, updating sorted array require $\Theta(n)$ for insertion
- Heap: only minima/maxima

# Binary Search Tree

A binary tree that organizes nodes with keys, the organization of the keys satisfies the **binary-search-tree property**:

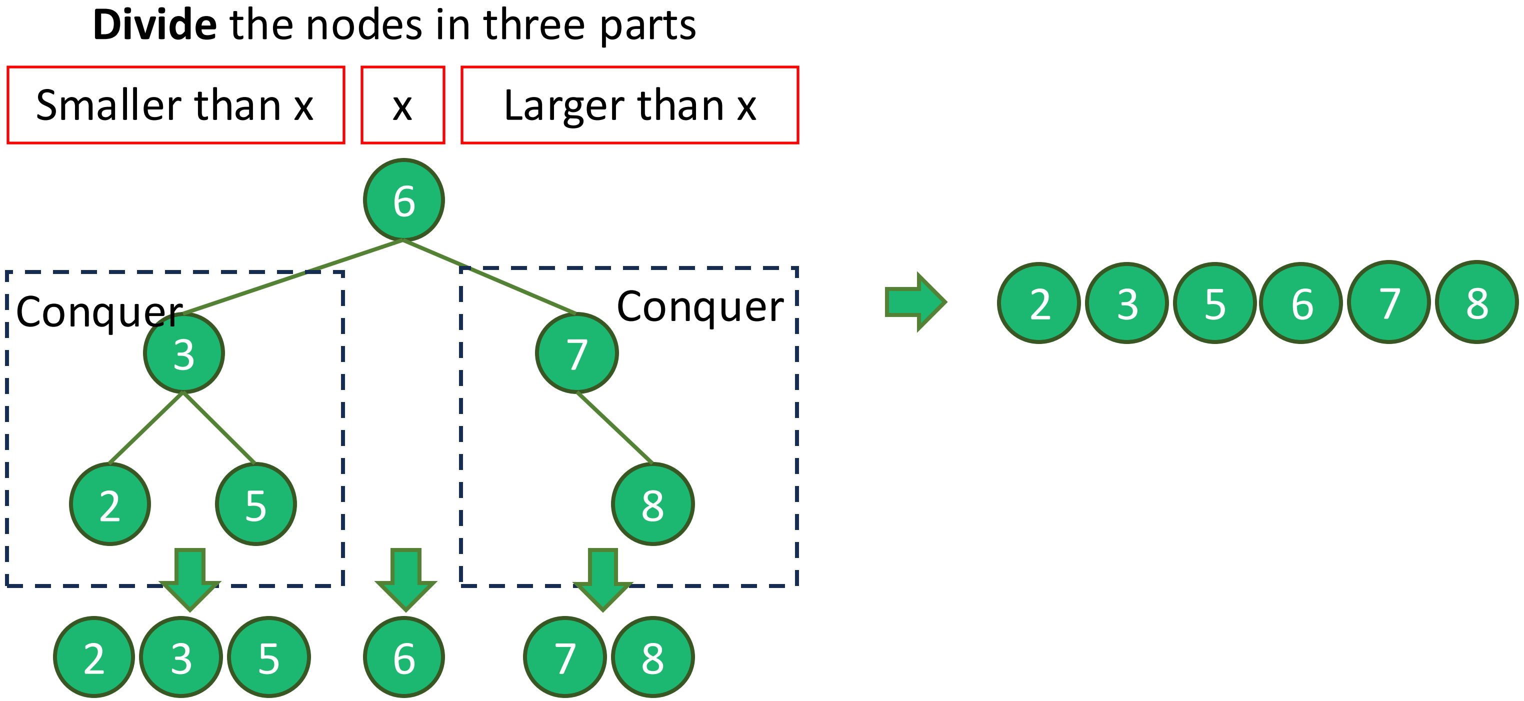
- For any node $x$, there is $y.key \leq x.key$ for any node $y$ in its left subtree
- For any node $x$, there is $y.key \geq x.key$ for any node $y$ in its right subtree



# Query: Sort

Problem: Given a BST, output keys in BST in increasing order

- Hint: divide and conquer (quick sort)
- In-order traversal returns the sorted sequence



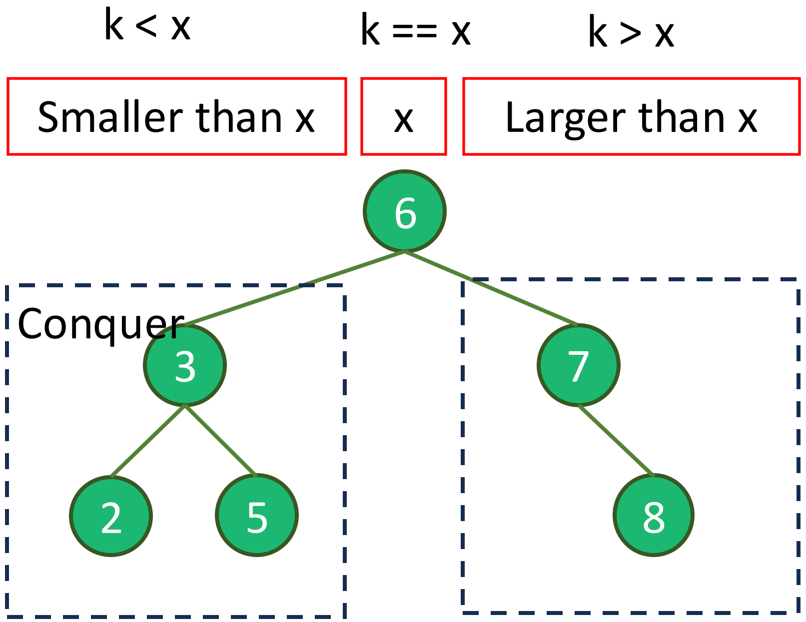

```
SORT(x)
1. if x != NIL
2.      SORT(x.left)
3.      print x.key
4.      SORT(x.right)
```

Complexity: $\Theta(n)$

# Query: Search

Problem: given a BST and a key $k$, find a node x whose x.key == $k$

- Hint: divide-and-conquer (binary search / k-smallest problem)
- Solution: we know which subtree the k belongs to
- Complexity: $\mathcal{O}(h)$ – one way from root to the leaf

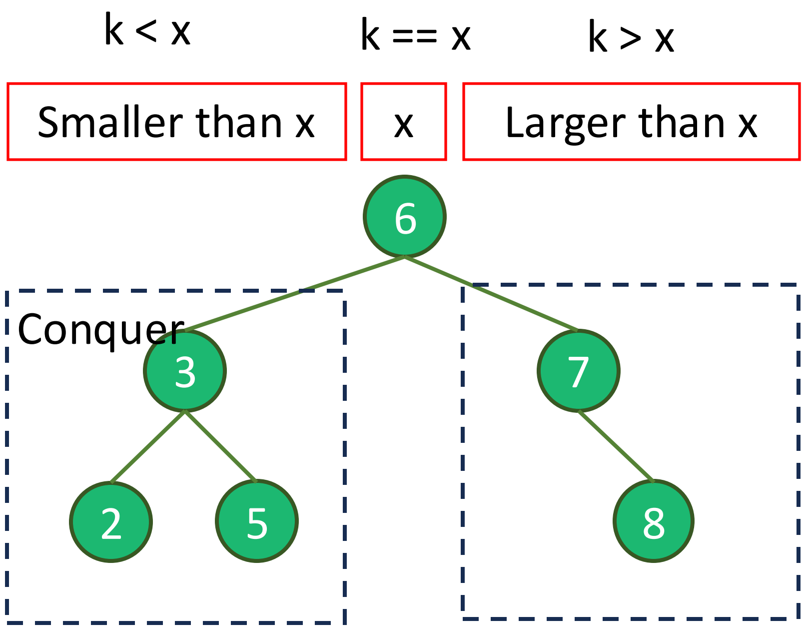



**Recursive**

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key:
    return x
if k < x.key:
    return TREE-SEARCH(x.left, k)
else: return TREE-SEARCH(x.right, k)
```

# Query: Search

- Extension: given a BST and a key $k$, find the node $\arg\max_{\{u \mid u.key \leq k\}} u.key$




```
TREE-SEARCH(x, k)                    Recursive
if x == NIL or x.key == k:
    return x
if x.key > k:
    return TREE-SEARCH(x.left, k)
else:
    u = TREE-SEARCH(x.right, k)
    return u if u != NIL else x
```

# Query: Minimum/Maximum

Find minimum/maximum: go left/right child until encountering a NIL

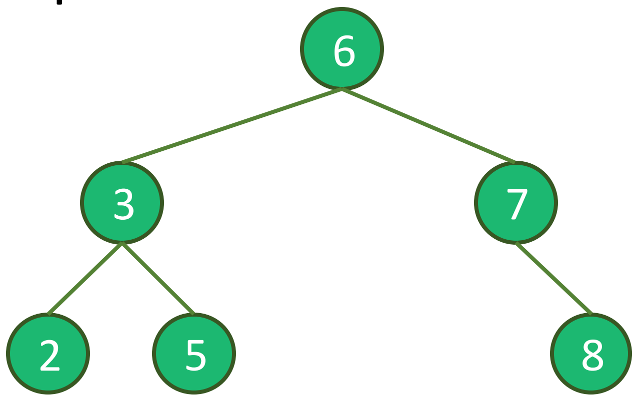
- Complexity: $\mathcal{O}(h)$ distance from root to the corresponding node

```
TREE-MINIMUM(x)
1. while x.left != NIL:
2.      x = x.left
3. return x
```

```
TREE-MAXIMUM(x)
1. while x.right != NIL:
2.      x = x.right
3. return x
```

Extension: Find a given node's predecessor/successor in the sorted sequence

- Example: what is the node with largest key smaller than node(6)?
- Maximum of left subtree / Minimum of right subtree



# Modification

Problem: how to insert/delete nodes while still holding the binary-search-tree property?

# Modification: Insertion

Given a new node $z$, where should we insert it?

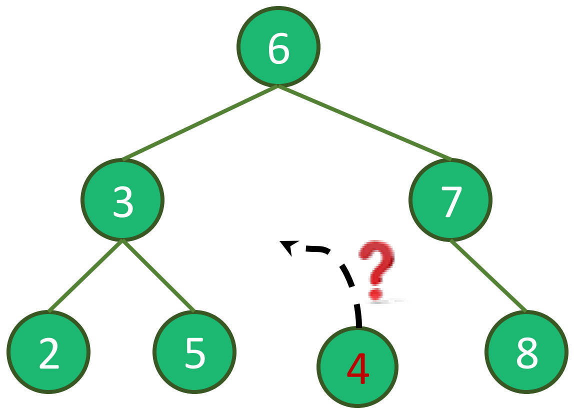
- <=root: the root's left subtree
- >=root: the root's right subtree

For the current root, the BST property still holds

Solution: Go left/right until reaching an empty position

Complexity: O(h) – go from root to NIL, single direction

```
TREE-INSERT(x, z)
if z.key < x.key:
    if x.left == NIL: x.left = z
    else:
        TREE-INSERT(x.left, z)
else:
    if x.right == NIL: x.right = z
    else:
        TREE-INSERT(x.right, z)
```

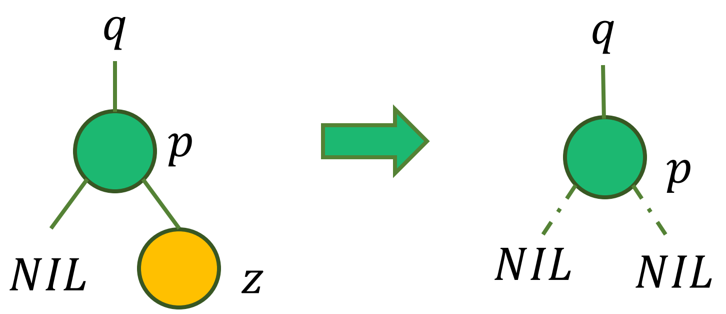


# Modification: Deletion

Given an BST and a node $z$, remove it while still holding BST property

(Case 1) remove leaf node $z$

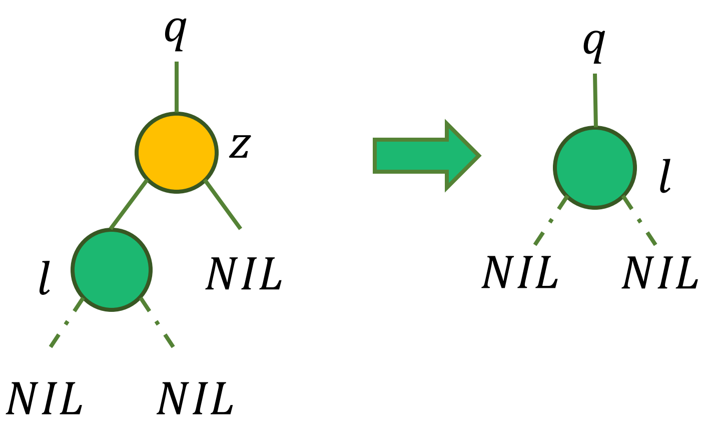
- Directly remove, will not change the BST property
- why: only $z$'s ancestors' subtrees are influenced, the component are unchanged except for deleting z



# Modification: Deletion

(Case 2) $z$ has one child

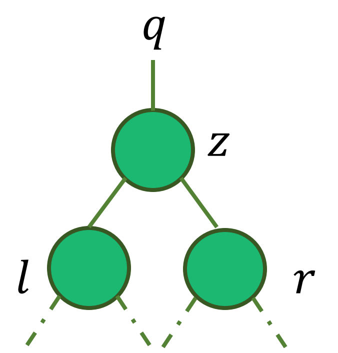
- Replace $z$ with the child
- why: only ancestors of $z$ are influenced, left subtree and right subtree's component does not change except for deleting z



# Modification: Deletion

(Case 3) $z$ has two children

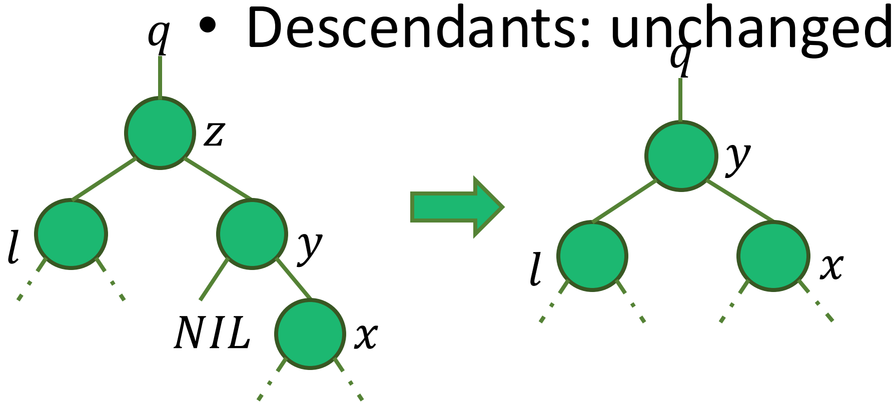
- Think: which node can be used to replace z?
  - Requirement: ≥left_subtree, ≤right_subtree
  - Answer: predecessor/successor of z (i.e., largest in left_subtree or smallest in right_subtree)
- Find predecessor/successor: $\mathcal{O}(h)$
- WLOG, supposing successor, property:
  - In $z$'s right subtree
  - Has no left child



# Modification: Deletion

(Case 3) $z$ has two children: replace $z$ with its successor $y$

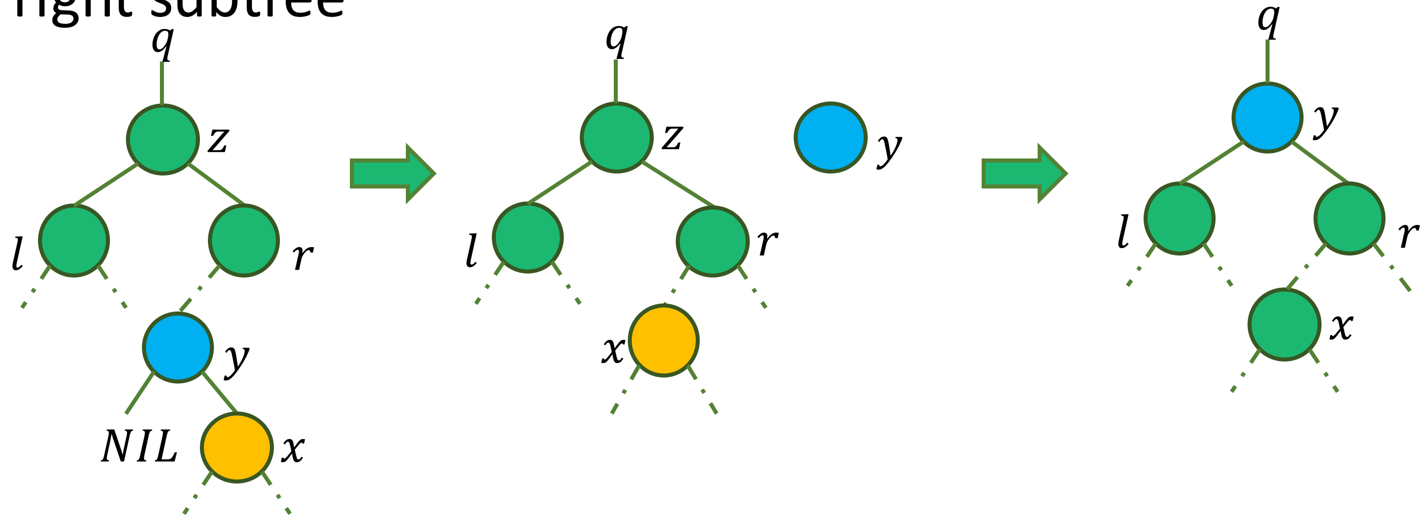
- (Case 3.1) $y$ is $z$'s right child
    - Replace $z$ by $y$
    - BST property:
        - z's ancestor: subtree component unchanged
        - Current node: z'left subtree <= z <=y <= y's right subtree
        - Descendants: unchanged subtrees



# Modification: Deletion

(Case 3) $z$ has two children: replace $z$ with its successor $y$

- (Case 3.2) $y$ is not $z$'s right child
  - Replace $y$ by its child x, then replace $z$ by $y$
  - BST property:
    - x's subtrees unchanged, r's subtree - only removes one element
    - Current: left subtree ≤ y ≤ right subtree



# Modification: Deletion

Complexity: O(h) – need to find successor in case 3

```
TREE-DELETE(T, z)
if z.left == NIL:
    TRANSPLANT(T, z, z.right)
elif z.right == NIL:
    TRANSPLANT(T, z, z.left)
else:
    y = TREE-NIMINUM(z.right)
    if y.p != z:
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Replace one subtree with another subtree

- (change child node of its parent)

```
TRANSPLANT(T, u, v)
if u.p == NIL:
    T.root = v
elif u == u.p.left:
    u.p.left = v
else:
    u.p.right = v
if v != NIL:
    v.p = u.p
```

# Summary of Complexity

Query

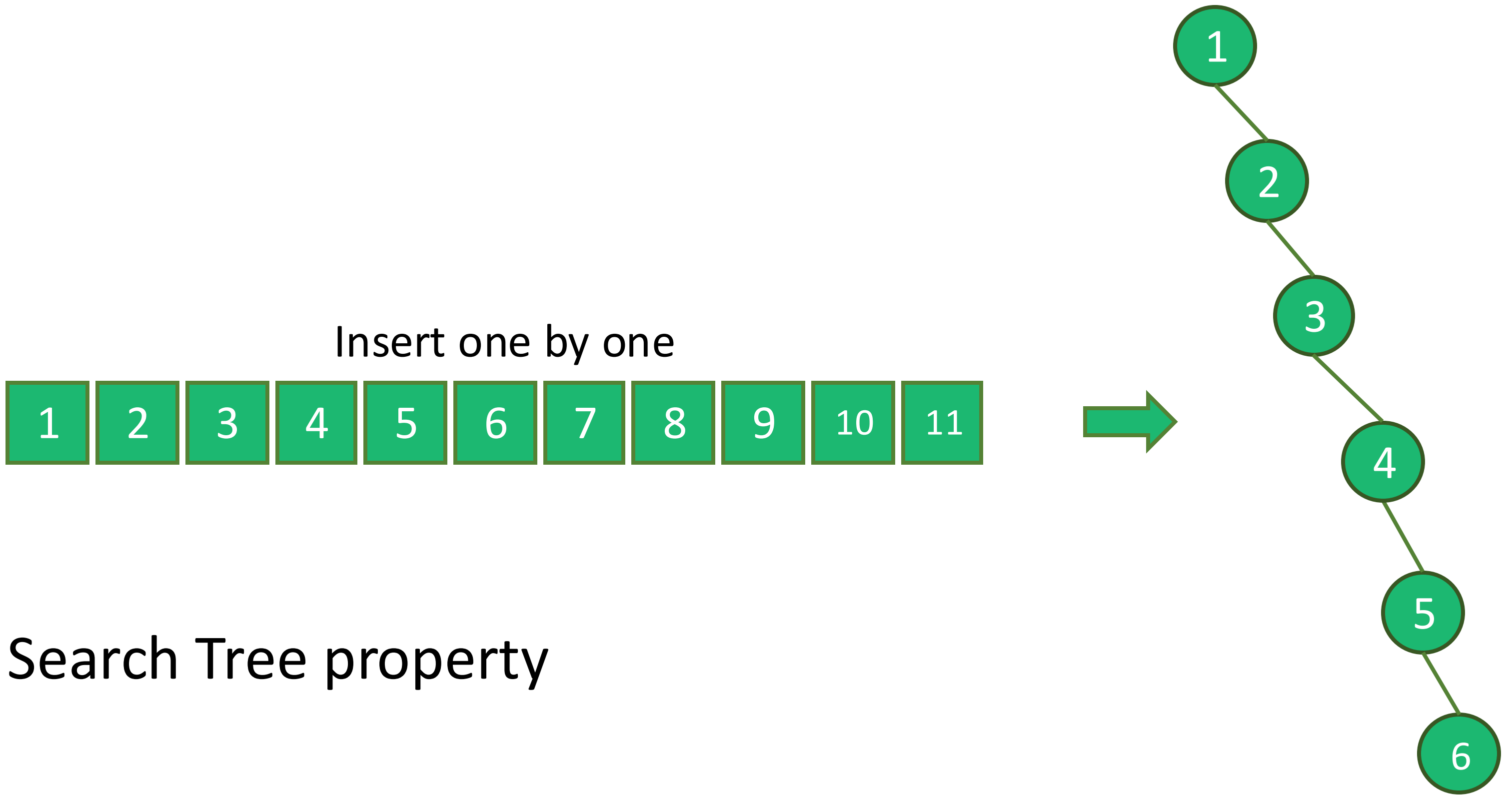
- Search $O(h)$
- Successor $O(h)$
- Predecessor $O(h)$
- Minimum $O(h)$
- Maximum $O(h)$

Modification: restore Binary Search Tree property

- Delete $O(h)$
- Insert $O(h)$

Any problem?

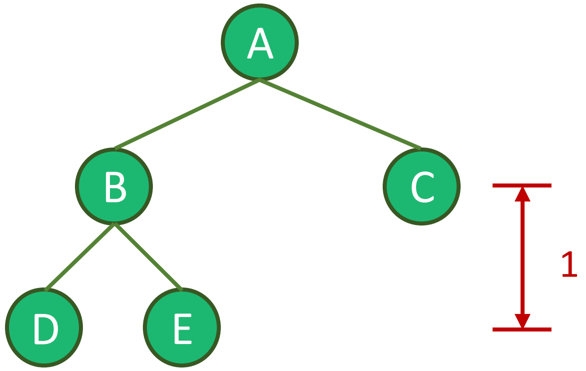
- In the worst case, $n$ nodes generate a tree with height $n - 1$



# Balanced Binary Tree

**Balanced binary tree**: the height of left and right subtrees for every node differ by no more than 1

- **Property:** the height is $O(\log n)$



# Self-Balanced Binary Search Tree

Automatically keeps its height small in the face of arbitrary item insertions and deletions

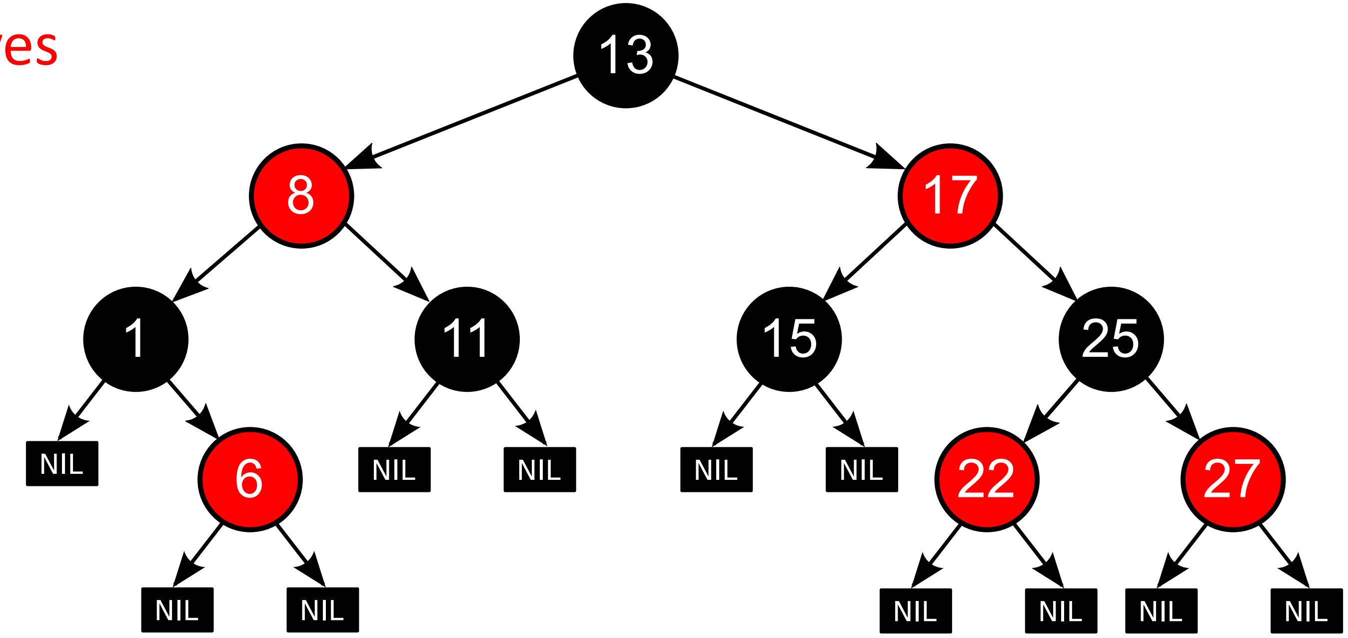
- AVL Tree
- Red Black Tree
- Treap
- …

**Idea**: bound the difference in height between every node’s left and right subtrees

# Red-Black Trees

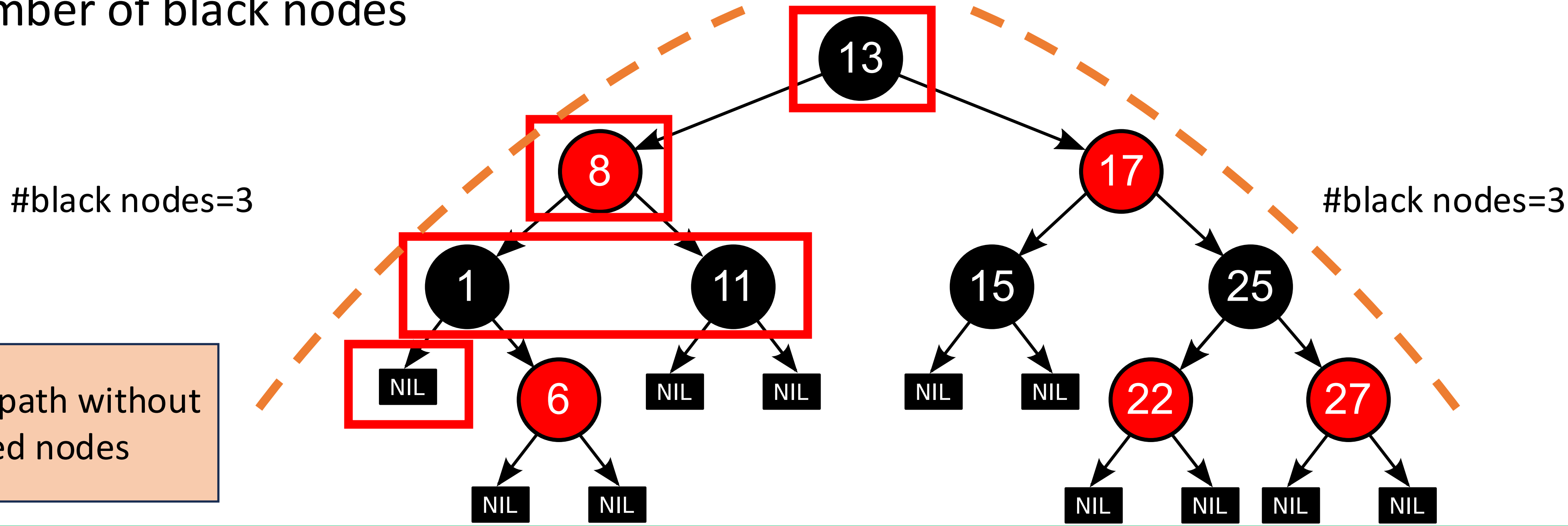
# Red-black Tree

- A kind of "balanced" binary search tree to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case
- A red-black tree is a binary search tree with one extra storage per node: color, which can be either RED or **BLACK**
- We refer to NIL as leaves



# Red-black Tree Properties

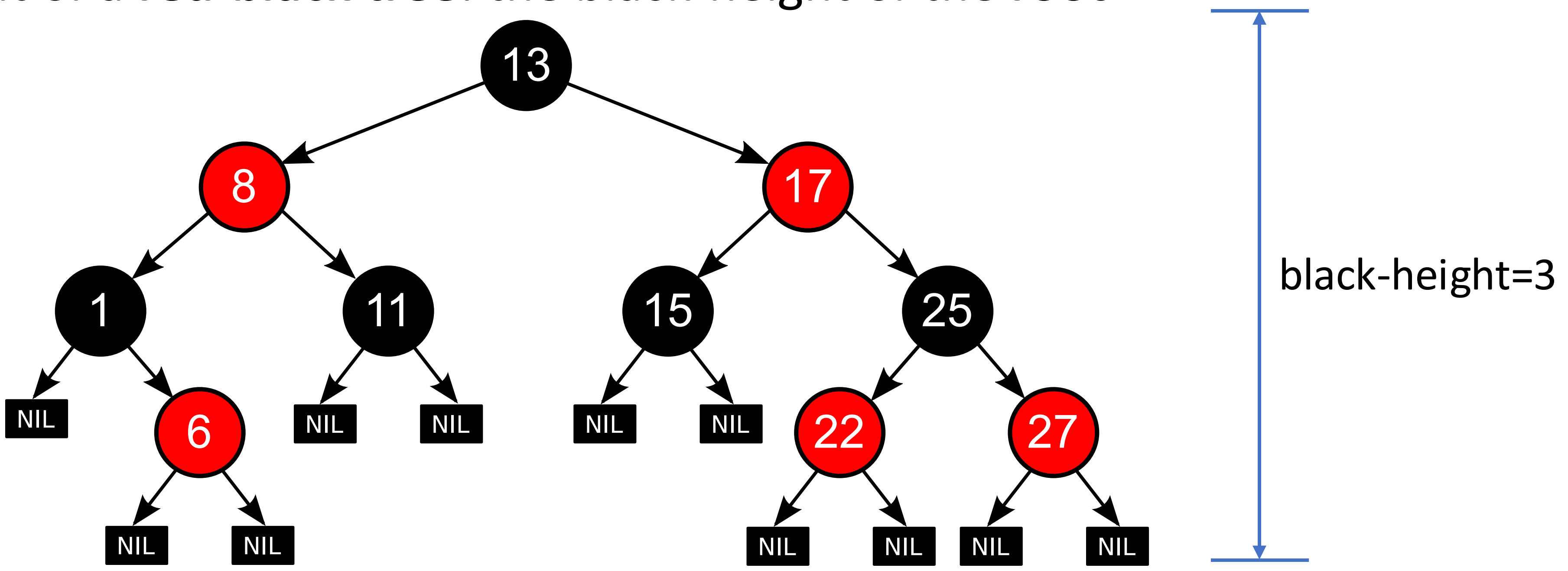
1. Every node is either red or black
2. The root and leaves (NIL) are black
3. If a node is red, both children are black - (obviously, parent is also black)
4. For each node, all simple paths from it to all descendant leaves contain the same number of black nodes




Simple path: path without duplicated nodes

# Black Height

- Black Height $bh(x)$: number of black nodes on any simple path from a node $x$ down to a descendant leaf
- Black-height of a **red-black tree**: the black-height of the **root**



# Height of Red-Black Tree

Lemma: A red-black tree with $n$ nodes has a height $\leq 2\log(n+1)$

1. Subtree (T) with root $x$ has at least $2^{bh(x)} - 1$ nodes

- Base case: if $\mathrm{T}.height = 0$, a leaf $(NIL) \Rightarrow bh(x) = 1 \Rightarrow\ 2^{bh(x)} - 1 = 1$. The claim holds
- Inductive step for height > 0 (Assuming height <k adheres, prove for height k):
  - For T.height = k
  - x has two children. (1) $x$ is red: $bh(child) = bh(x)$; (2) $x$ is black: $bh(child) = bh(x) - 1$
  - Each child's subtree height < $k \Rightarrow$ each has least $2^{bh(x)-1} - 1$ nodes
  - $x$'s subtree has $\geq 1 + (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) = 2^{bh(x)} - 1$ nodes

2. Let $h$ be the height of the tree. With property 3 (red node's children are black), at least half the nodes from root to a leaf are black. Then $bh(\mathrm{root}) \geq \frac{h}{2}$; then $n \geq 2^{\frac{h}{2}} - 1$, i.e., $h \leq 2\log(n+1)$

# INSERT and DELETE operation?

- Requirements
  - Binary search tree property
  - Red-Black Tree Property
- BST property: use the basic insert and delete algorithm in BST?
- How to hold the red-black tree property?

# INSERT

Consider what is the color?

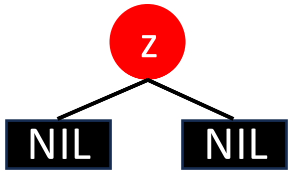
- Black?
  - One node must has one extra black node on the path from it to a descendant leaf, violate property 4
- Red: no influence on the black height of any node
  - May violate property 3

```
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
elif 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)
```

# Property Violation

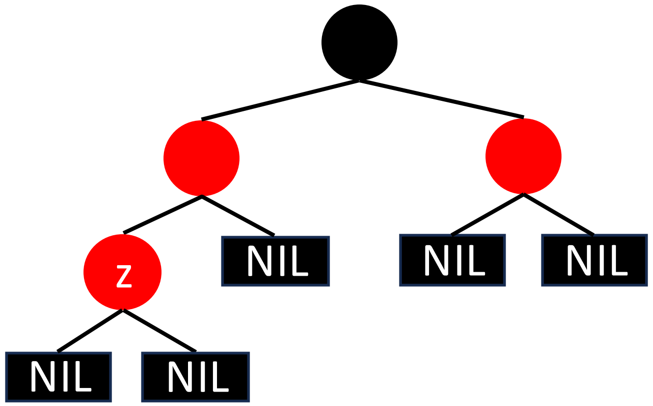
What properties are violated after inserting a red node z

1. Every node is either red or black ✅

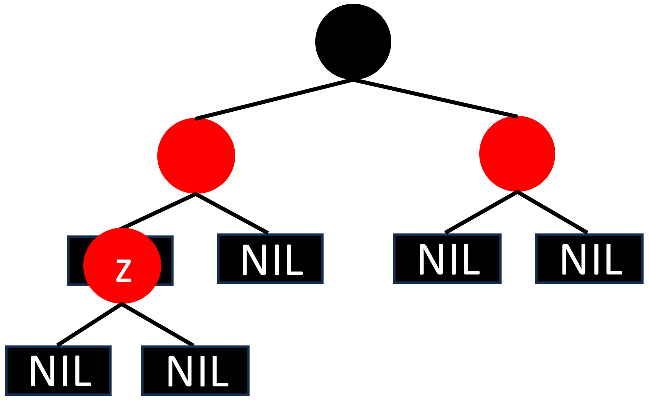



2. The root and leaves are black

   ❌: z may be the root




3. If a node is red, both children are black ❌: z.p can be red
4. For each node, all simple paths from it to all descendant leaves contain the same number of black nodes ✅
   - Red node z replaces a (black) NIL, but it has NIL children



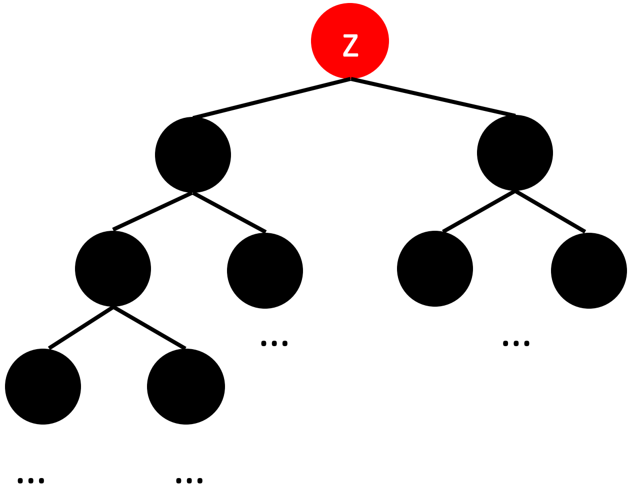

Summary:

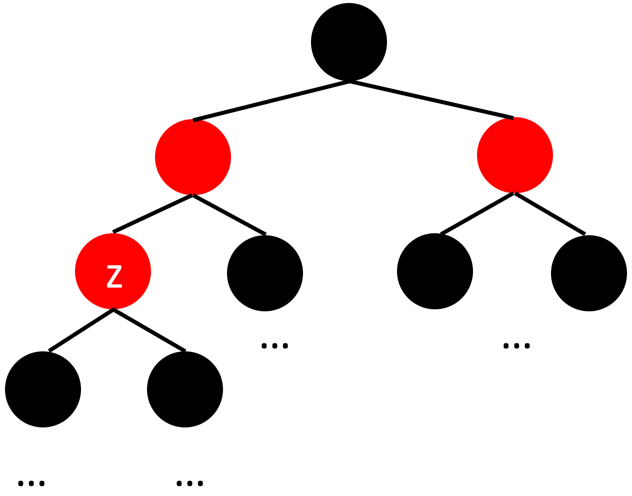
- z violates property 2, or
- z-z.p link violates property 3

# Property Violation

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

How to fix it?



# RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

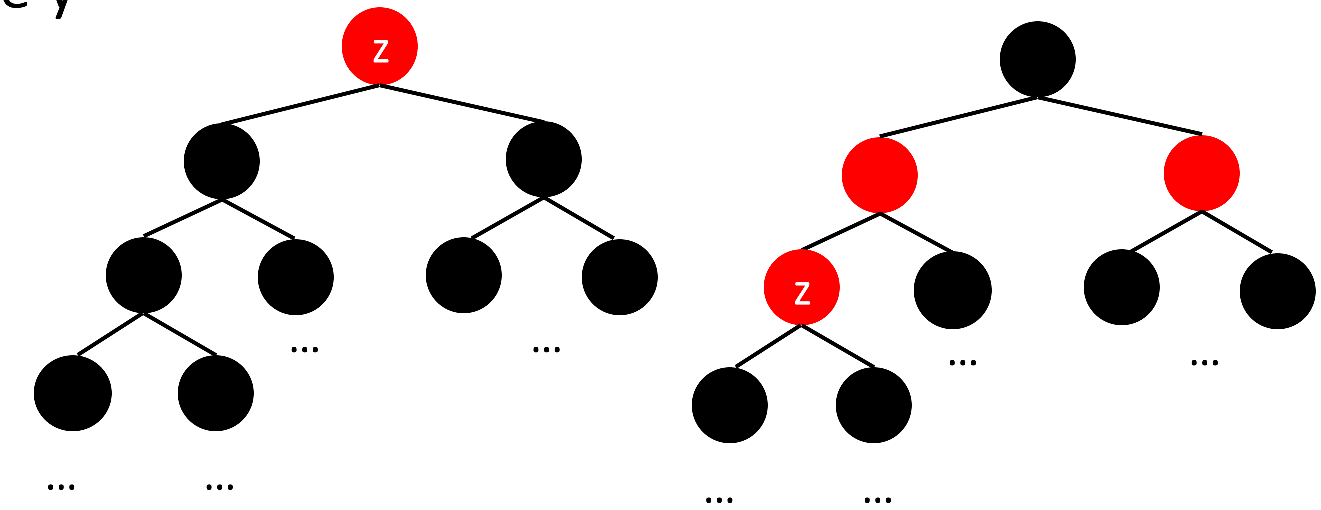
Key idea: different cases based on color of parent and uncle

**Case 1**: z is the root (no parent)

**Case 2**: The parent is black

**Case 3**: The parent is red, consider the uncle y

- **Case 3.1**: y is red
- **Case 3.2**: y is black
  - **Case 3.2.1**: y is black (triangle)
  - **Case 3.2.2**: y is black (line)

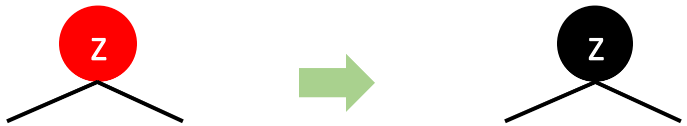


# RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 1**: z is the root (no parent)

- Color it black




Properties:

1. Every node is either red or black ✅
2. The root and leaves are black ✅
3. If a node is red, both children are black ✅: z becomes black, not applicable
4. For each node, all simple paths from it to all descendant leaves contain the same number of black nodes ✅: bh(x) for all the nodes except for z are unchanged

## RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 2**: The parent is black

Properties:

1. Every node is either red or black ✅
2. The root and leaves are black ✅: has parent, not root
3. If a node is red, both children are black ✅: z.p is black, not applicable
4. For each node, all simple paths from it to all descendant leaves contain the same number of black nodes ✅

No modification

# RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 3**: The parent is red, consider the uncle y

- **Case 3.1**: y is red
- **Case 3.2**: y is black
  - **Case 3.2.1**: y is black (triangle)
  - **Case 3.2.2**: y is black (line)

Requirement: fix property 3 (red node have black children)

# RB-INSERT-FIXUP

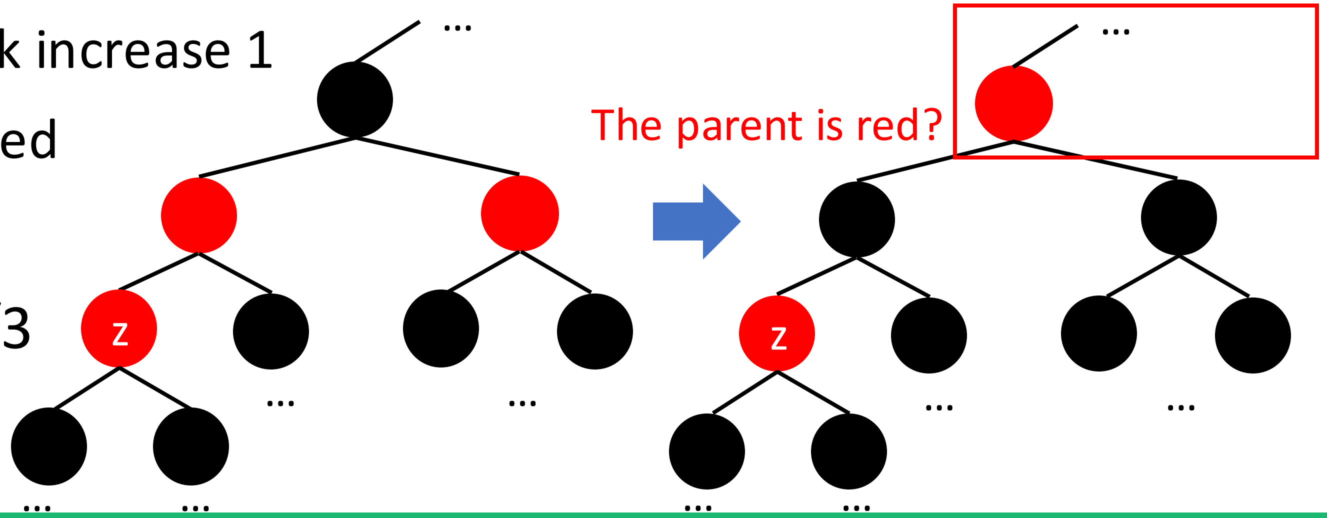
**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 3.1**: The parent is red (then grandparent is black), uncle y is red

- Solution: color both parent and uncle black, color grandparent red
  - Property 3: z.p and z.uncle become black, not applicable
  - Property 4:

1. z.p and z.uncle: all simple-paths' #black increase 1
2. z.p.p: all simple-paths' #black unchanged
3. z.p.p.ancestor: unchanged

- Grandparent may violate the property 2/3

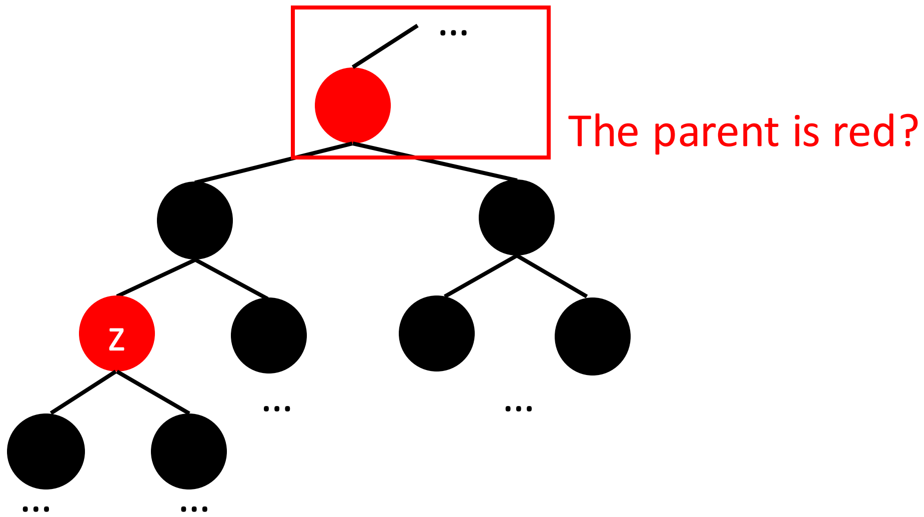


# RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 3.1**: The parent is red (then grandparent is black), uncle y is red

- Solution: color both parent and uncle black, color grandparent red
- **Observation**: regarding the grandparent as z, the problem persists, but z is level up




Properties:

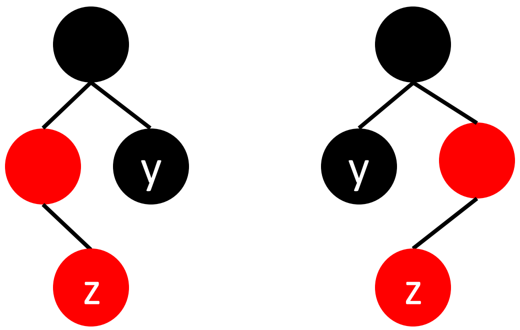
1. Every node is either red or black ✅
2. The root and leaves are black ❌: break when z become root
3. If a node is red, both children are black ❌: break with z-z.p link
4. For each node, all simple paths from it to all descendant leaves contain the same number of black nodes ✅

# RB-INSERT-FIXUP

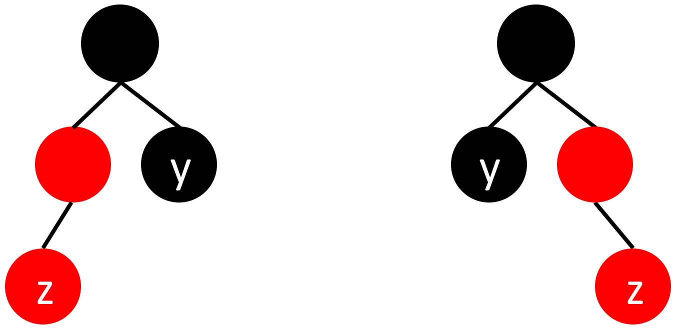
**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

- **Case 3.2.1**: y is black (triangle) - z is left & z.p is right child or z is right & z.p is left child
- **Case 3.2.2**: y is black (line) - z & z.p are both left child or are both right child

triangle




line

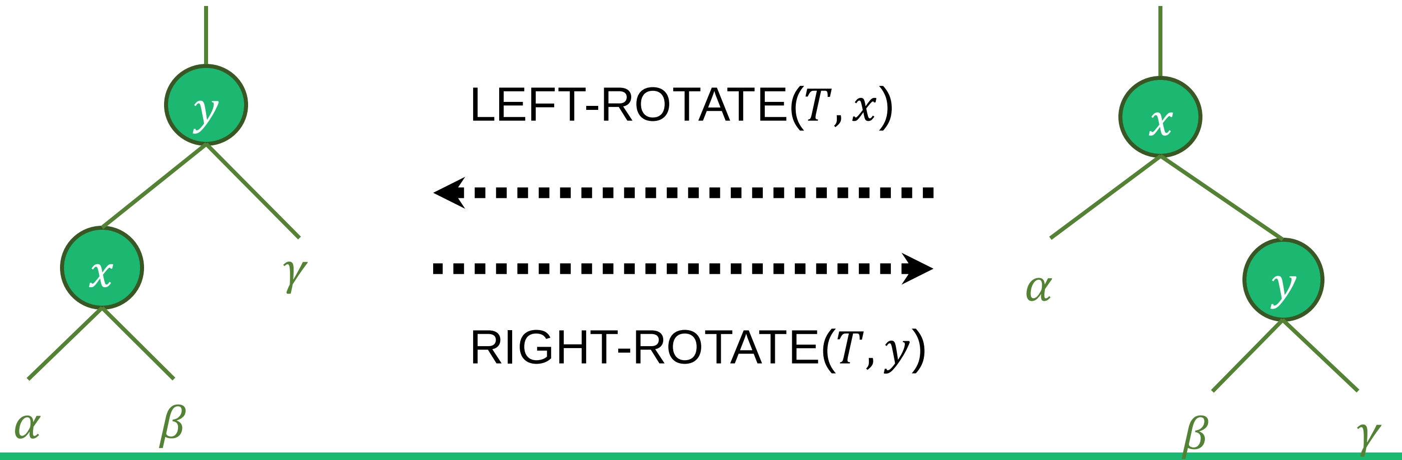



Can be transformed to each other with “rotation”

# Rotation

A local operation that changes positions of the nodes while preserving BST property

- Symmetric operations: left and right rotation
- Left rotation on a node x (assume right child y is not NIL)
  - "pivots" around the link x-y: y become the new root, x is y's left child, y's left child is x's right child
- Easy to prove BST property



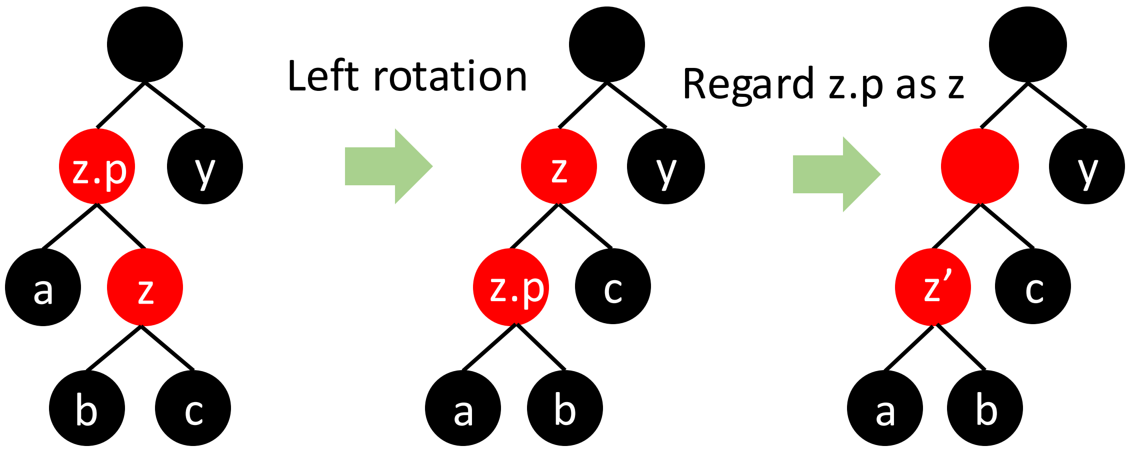

```
LEFT-ROTATE(T,x)
y = x.right
x.right = y.left
if y.left !=T.nil
      y.left.p = x
y.p = x.p
if x.p == T.nil
      T.root = y
elif x = x.p.left
      x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

# RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 3.2.1**: y is black (triangle) - z is left & z.p is right child or z is right & z.p is left child

- Solution: can transform to **Case 3.2.2** with rotation on z.p, then solve **Case 3.2.2**




Notice: The other is symmetric

Properties:

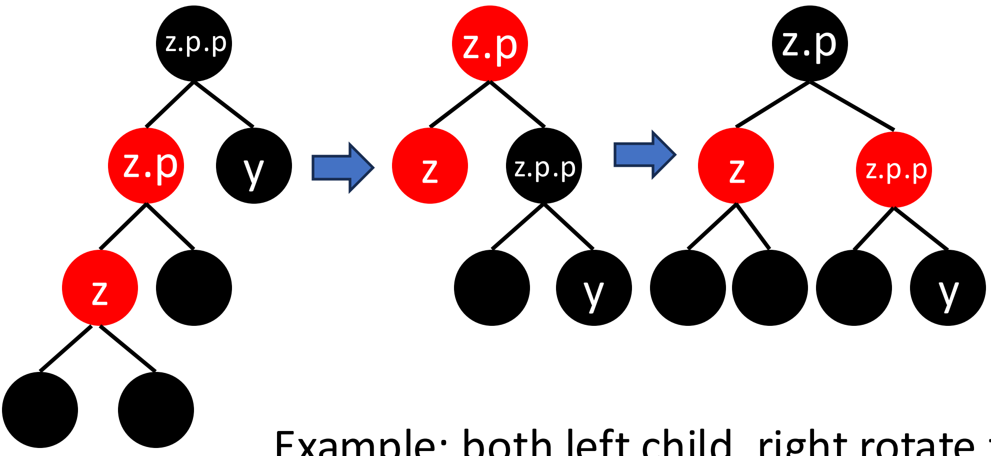
1. Every node is either red or black ✅
2. The root and leaves are black ✅
3. If a node is red, both children are black
   - Only z'-z'.p breaks it, others not involved
4. For each node, all simple paths from it to all descendant leaves contain the same number of black nodes ✅ originally bh(a) = bh(b) = bh(c), still holds

# RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 3.2.2**: y is black (line) - z & z.p are both left child or are both right child

- Solution: rotate z.p.p & recolor parent black, grandparent red




Properties:

1. Every node is either red or black ✅
2. The root and leaves are black ✅ : current root or original root, black
3. If a node is red, both children are black
   ✅ : z unchanged, z.p.p has z.p's other child (black) and uncle (black)
4. For each node, all simple paths from it to all descendant leaves contain the same number of black nodes

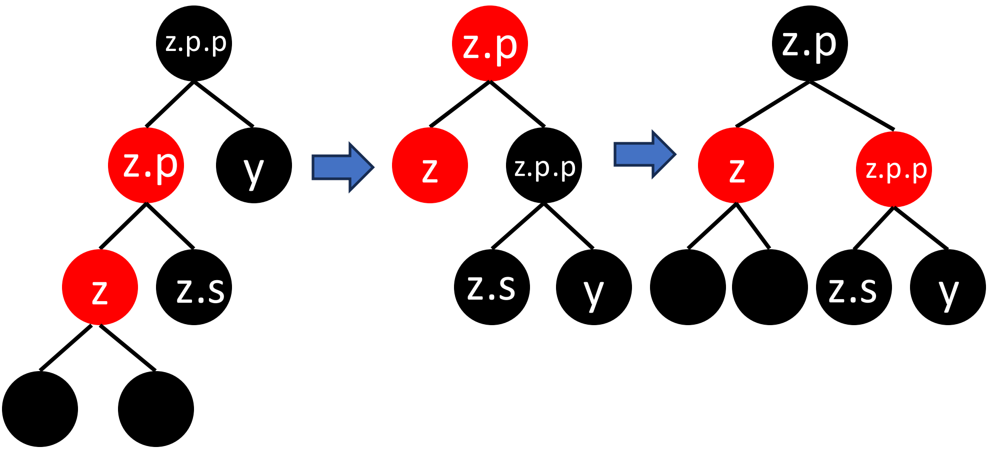
Example: both left child, right rotate the grandparent

# RB-INSERT-FIXUP

**A generalized situation (z may have non-NIL child):** given a tree satisfying property 1 and 4, only a red node $z$ may break property 2 or its parent z-$z.p$ link may break property 3

**Case 3.2.2**: y is black (line) - z & z.p are both left child or are both right child

- Solution: rotate z.p.p & recolor parent black, grandparent red




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

- z: unchanged
- z.p.p: y & z.s unchanged, so all right path bh(y), all left path bh(z.s), since originally holds property 4, bh(y) + 1 = bh(z.p) + 1 = bh(z.s) + 1, so satisfy
- z.p: z unchanged, so all left path bh(z) + 1, right path bh(y)+1=bh(z.p) + 1=bh(z) +1
- (the same as original, so ancestors unchanged)

# RB-INSERT-FIXUP

**Summary**

**Case 1**: z is the root (no parent)

**Case 2**: The parent is black

**Case 3**: The parent is red, consider the uncle y

- **Case 3.1**: y is red
- **Case 3.2**: y is black
  - **Case 3.2.1**: y is black (triangle)
  - **Case 3.2.2**: y is black (line)

```
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 # case 3.1
            y.color = BLACK # case 3.1
            z.p.p.color = RED # case 3.1
            z = z.p.p # case 3.1
        elif z == z.p.right
            z = z.p # case 3.2.1
            LEFT-ROTATE(T,z) # case 3.2.1
        z.p.color = BLACK # case 3.2.2
        z.p.p.color = RED # case 3.2.2
        RIGHT-ROTATE(T,z.p.p) # case 3.2.2
     else
        (same but exchange “right” and “left”)
T.root.color = BLACK # case 1 & Case 2
```

# Complexity for Insertion

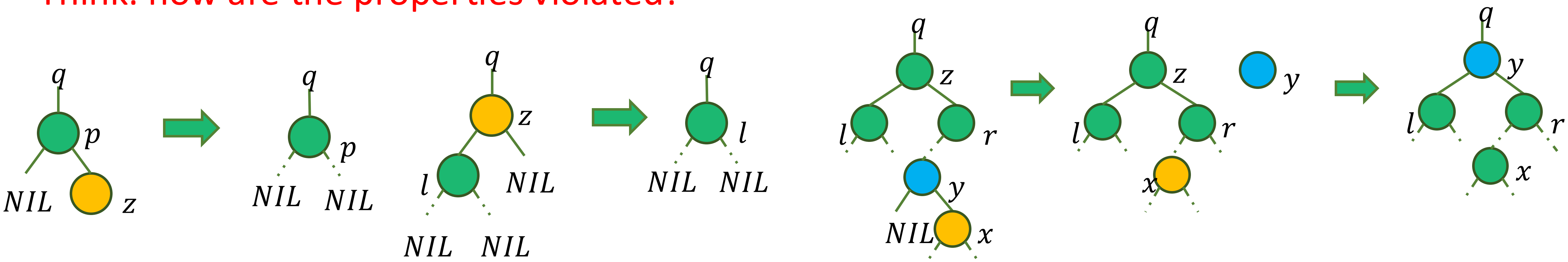
- Insertion: $\mathcal{O}(\log n)$
- Recover red-black tree property
  - Case 1 & 2: red-black tree after 1 step
  - Case 3.2: red-black tree after 1 step
  - Case 3.1 occurs, $z$ moves two levels up the tree: at most go $\mathcal{O}(\log n)$ levels up
- In total: $\mathcal{O}(\log n)$

# Delete

Recall delete in BST:

- z has 0 non-NIL child: directly remove
- z has 1 non-NIL child: replace with non-NIL child
- z has 2 non-NIL children: replace with z' s successor y, replace y with its only child
  - In red-black tree: replace z's value (not color) with successor y, then delete y
  - Goal - Preserve color structure as much as we can

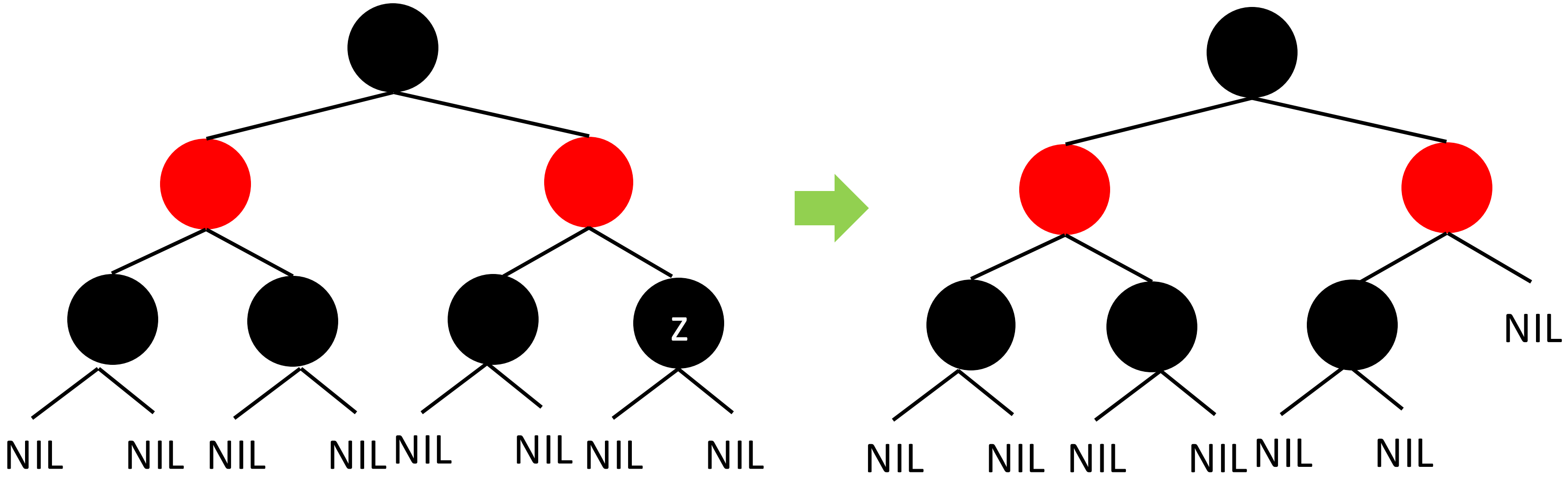
Think: how are the properties violated?



# Property Violation

*z has 0 non-NIL child: remove*

- z is red: no property violation
- z is black:
  - z is root: no property violation
  - z is not root: must violate property 4 (z.ancestor decrease 1 black node on paths involving z)

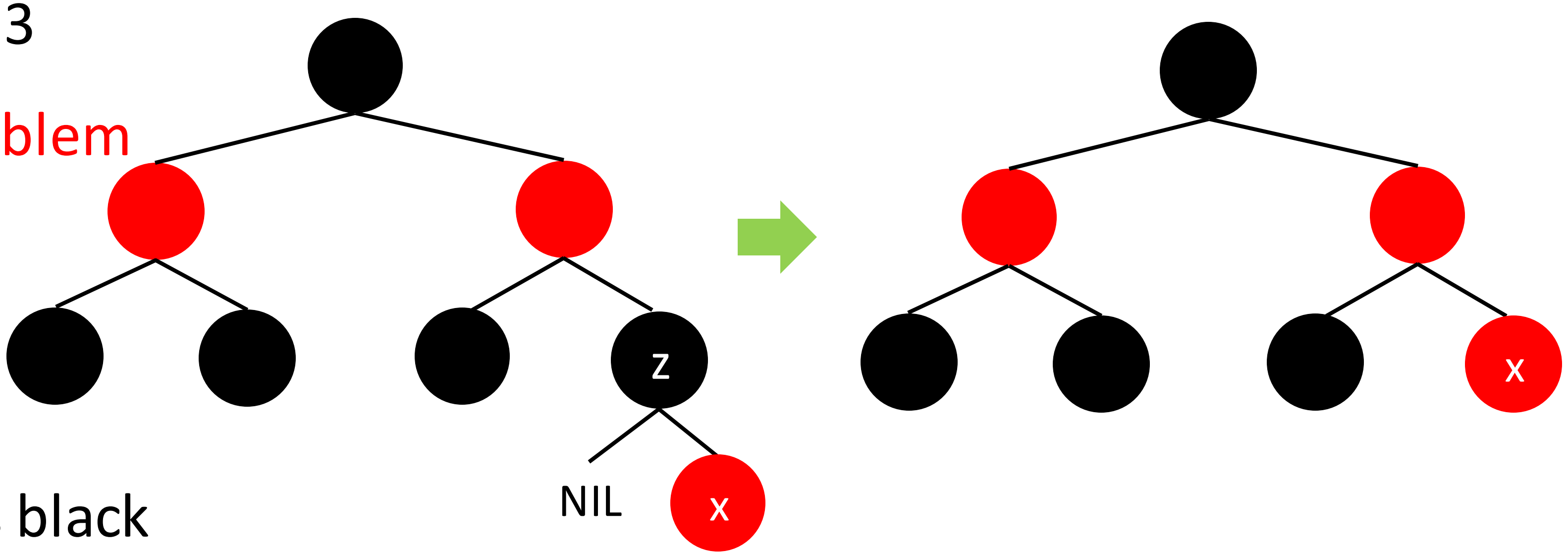


# Property Violation

*z has 1 non-NIL child: delete, replace with non-NIL child y*

Observation: x must be red (the other branch is only a NIL), then z must be black

- z is root: violate property 2
- z is not root:
  - Must violate property 4 (z.ancestor decrease 1 black node on one side)
  - z.p is red: violate property 3
- Color x black solves all the problem




Summarize the two cases:

violation happens only when z is black

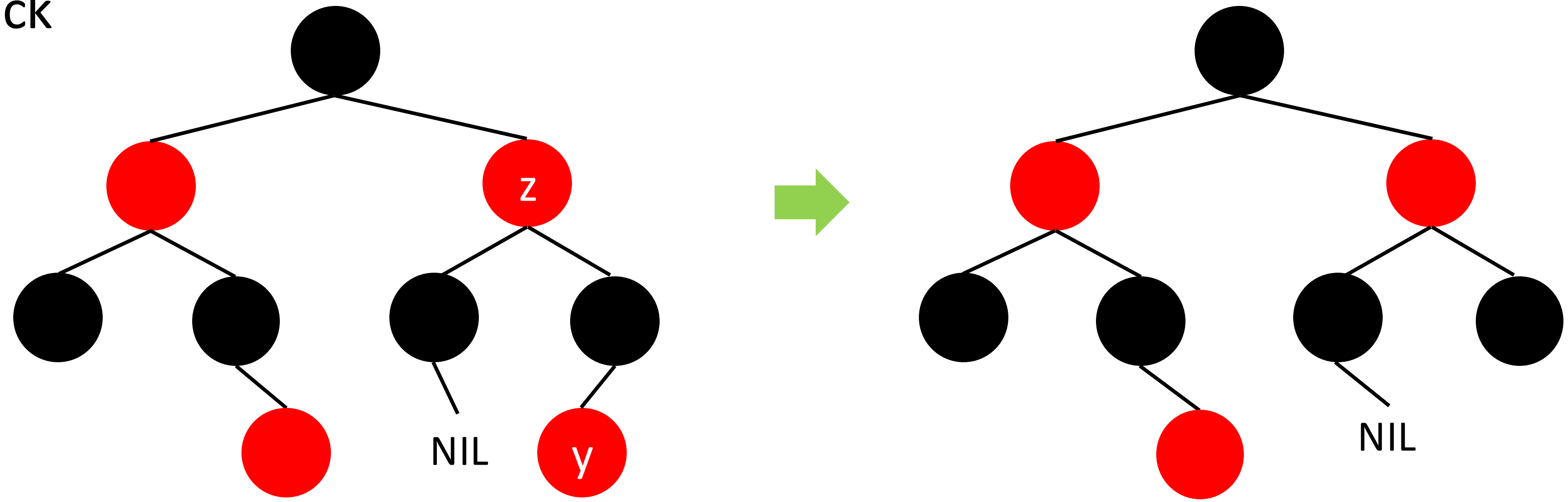
## Property Violation

*z has 2 non-NIL children: replace z's value (not color) with successor y, then delete y*

Red-black tree's property only cares about color. So the only factor violating the properties is the deleted node (instead of changing value)

- y has at most 1 non-NIL child, deleting y belongs to previous two cases

Summarize all the cases: (possibly after changing value), violation happens iff *the deleted node* is black



# Delete

1. Track the color of the deleted node
2. Restore red-black tree property after the deleting a black node

```
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)
elif 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)
```

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

# Fix the Violation

Given that deleted node is black, consider the 2 cases in terms of the node taking its place (x):

- Case 0: x must be black (NIL)
  - x is root: red-black tree
  - x is not root:  violate property 4
- Case 1: x must be red
  - Color it black makes a red-black tree
- We can use the color to distinguish the 2 cases

```
RB-DELETE-FIXUP(T, x)
if x != T.root and x.color == BLACK
    # fix violation
else
    x.color = BLACK
```

# x is black and not root?

Generalize the post-deletion situation: a ***quasi*** red-black tree where, from each of x's ancestor, u, simple paths to a leaf containing x has a black node deficit than the others

Key idea: different cases based on color of x's sibling w and w's children

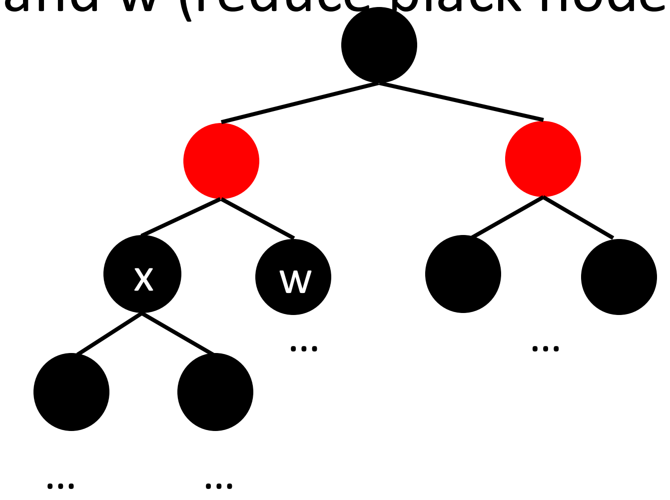
Intuition: we are comparing simple paths going through x and w (reduce black nodes from w?)

**Case 1**: w is red

**Case 2**: w is black

- **Case 2.1**: w.left & w.right are black
- **Case 2.2**: w.left is red and w.right is black
- **Case 2.3**: w.right is red

Note: assume x is left child, right is symmetric




Generalize: x can be non-NIL

# RB-DELETE-FIXUP

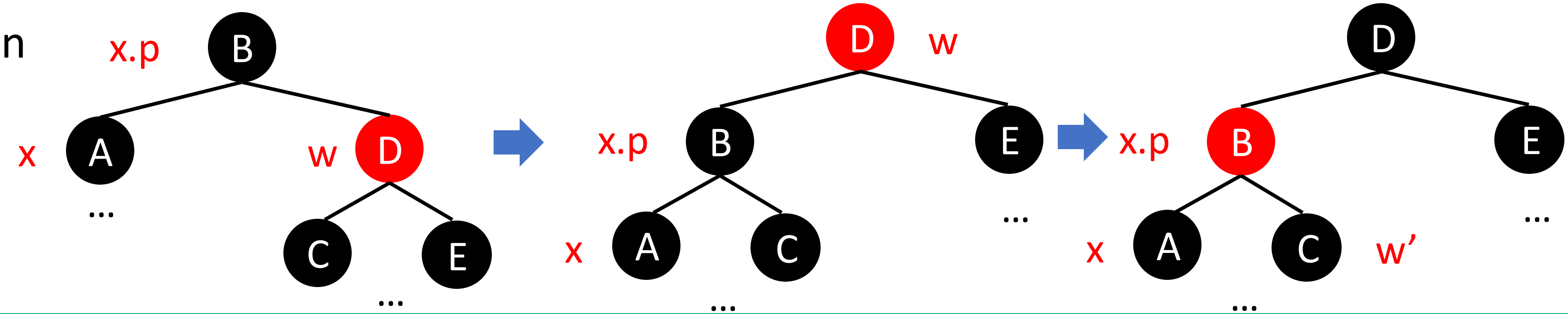
**Case 1**: x's sibling w is red

- (since we suppose x is left child) Left rotate x.p
- Switch the color of w and x.p

Regarding the new sibling as w, case 1 is transformed to case 2

*Still a quasi red-black tree (consider x's ancestors):*

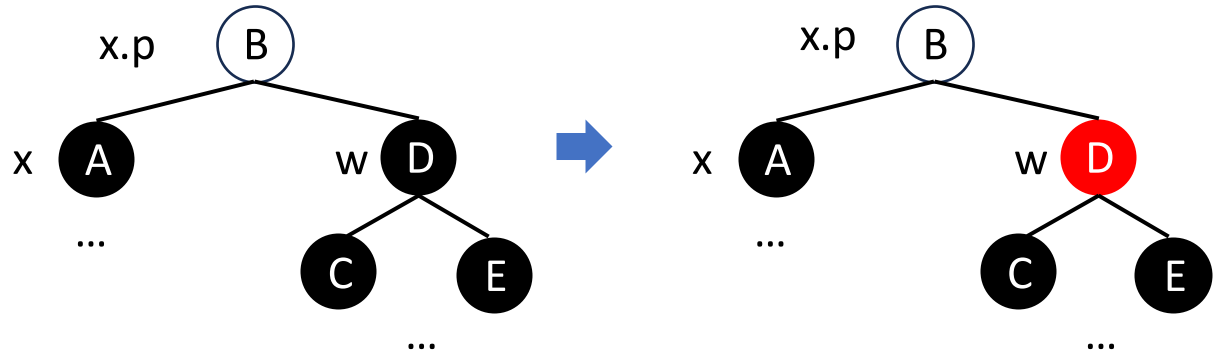
- A: red-black tree, unchanged, suppose bh(A) = n
- B: right path n + 1, left paths containing A has a black node deficit
- D: right path n + 2, left paths containing A has a black node deficit
- More ancestors: same situation
  - (Except for paths containing A)
  - The tree's bh is originally n+2
  - Now is still n+2



# RB-DELETE-FIXUP

**Case 2.1**: x’s sibling w is black, and both of w’s children are black

- Color w red
- If x.p is red: color it black
- If x.p is black: consider x.p as the new x



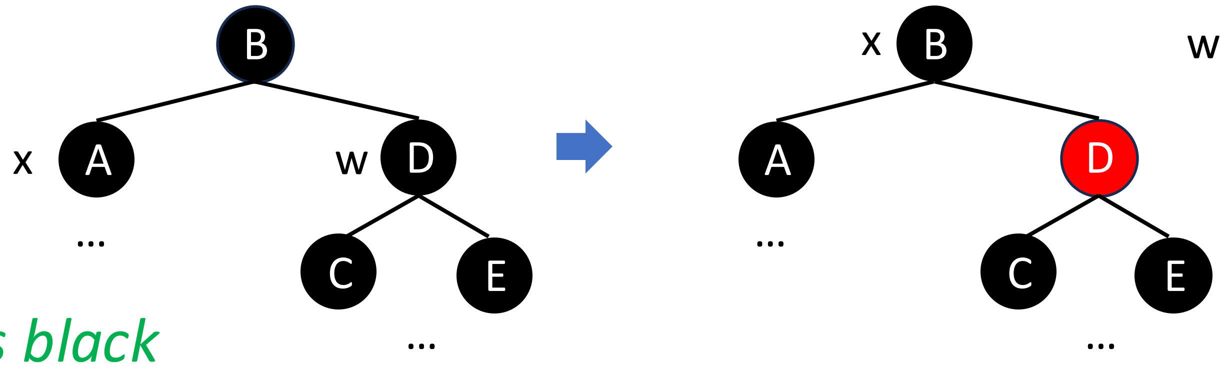

*The problem is solved when x.p is red:*

- A: Red-black properties stay unchanged. Assume $bh(A) = n$.
- D: Both left and right paths have black height n. Hold red-black tree property.
- B: Both left and right paths have black height n + 1. Hold red-black tree property.
- Ancestors of B: Initially, paths containing A (of course also contains B) have a black node deficit. Changing B to black adds one black node for them, making all paths meet red-black tree properties.

# RB-DELETE-FIXUP

**Case 2.1**: x's sibling w is black, and both of w's children are black

- Color w red
- If x.p is red: color it black
- If x.p is black: consider x.p as the new x




*Prove it still a quasi red-black tree when x.p is black*

- A: Red-black properties stay unchanged. Assume $bh(A) = n$.
- D: Both left and right paths have black height n + 1. Hold red-black tree property.
- B: Both left and right paths have black height n + 1. Hold red-black tree property.
- Ancestors of B: Initially, paths containing A have a black node deficit. Now paths containing D also have a black node deficit. Thus paths containing B have a black node deficit

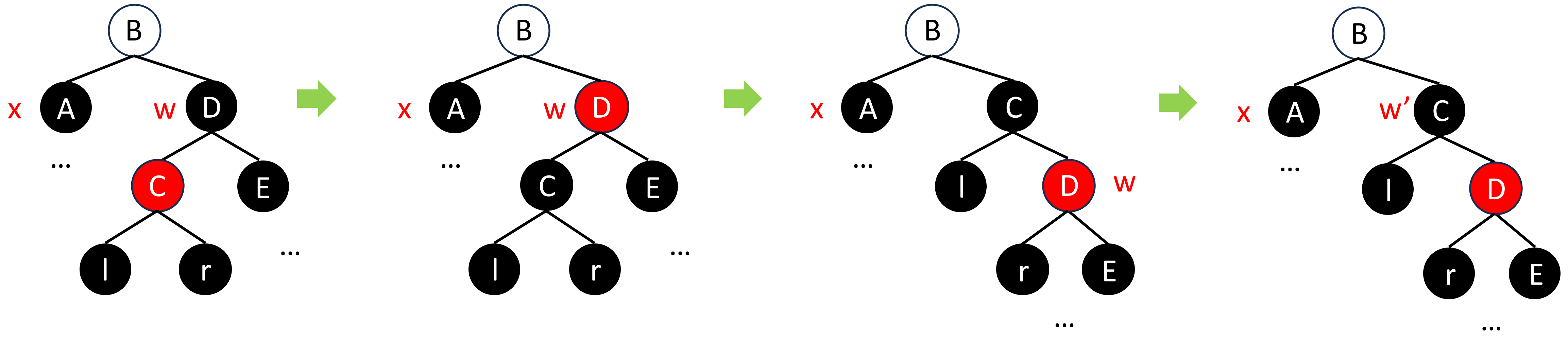
*Therefore, move the problem one level up*

# RB-DELETE-FIXUP

**Case 2.2**: x's sibling w is black, w's left child is red, and w's right child is black

- Color w.left black, color w red
- Right rotation around w

Taking new sibling as w, case 2.2 is transformed to case 2.3

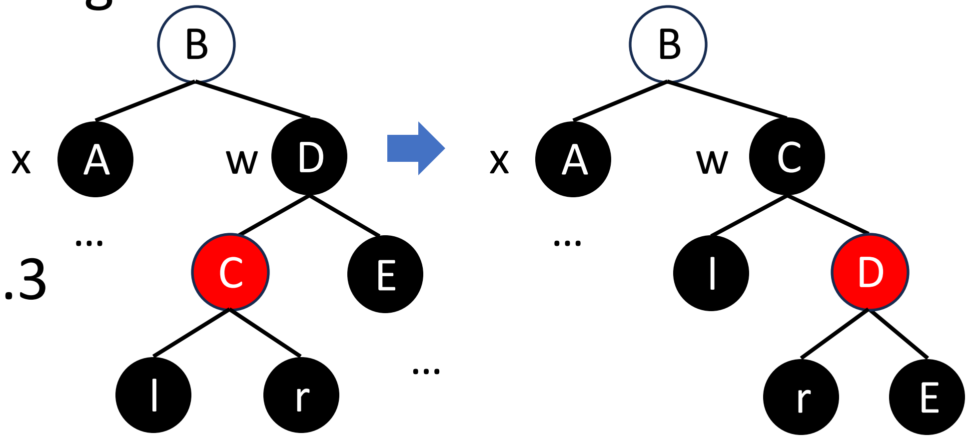


# RB-DELETE-FIXUP

**Case 2.2**: x's sibling w is black, w's left child is red, and w's right child is black

- Color w.left black, color w red
- Right rotation around w

Taking new sibling as w, case 2.2 is transformed to case 2.3




*Still a quasi red-black tree (consider x's ancestors):*

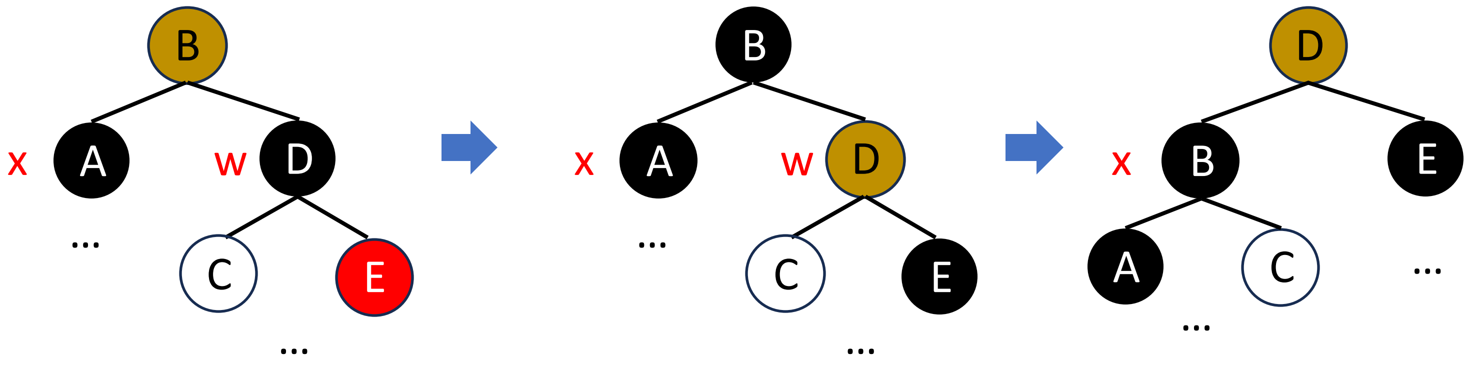
- A: Red-black properties stay unchanged. Assume $bh(A) = n$.
- D: Both left and right paths have black height n. Hold red-black tree property.
- C: Both left and right paths n + 1. Hold red-black tree property.
- B: Right paths n + 1 + ? (unchanged), left paths containing A n + ?, has a black node deficit
- B.ancestors: same situation, unchanged

…

# RB-DELETE-FIXUP

**Case 2.3**: x's sibling w is black, and w's right child is red

- Switch w and x.p's color
- Color x.p and w.right black
- Left rotate x.p




*The problem is solved:*

- A: Red-black properties stay unchanged. Assume $bh(A) = n$.
- E: Originally all paths n, now n + 1. Red-black tree properties stay unchanged.
- C: Originally all paths n, now n. Red-black tree properties stay unchanged.
- B: left paths become n + 1, right paths become n + 1. Hold Red-black tree properties.
- D: Both left and right paths n + 1 + ?. Hold Red-black tree properties.
- Ancestors: originally *B-A-leaves* paths' #blacknodes= n + ?, *B-D-leaves* paths' #blacknodes= n + 1 + ?, now all n + 1 + ?. Therefore, paths going through A has 1 extra black node filled.

Think: is it possible that D-D.p are both red?

## Summary

**Case 1**: w is red

**Case 2**: w is black

- **Case 2.1**: w.left & w.right are black
- **Case 2.2**: w.left is red and w.right is black
- **Case 2.3**: w.right is red

Complexity:

- Case 2.2-Case 2.3: directly solve the problem
- Case 1-Case 2.1: 1 level up or directly solve the problem

```
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 2.1
            x = x.p # case 2.1
        else
            if w.right.color == BLACK
                w.left.color = BLACK # case 2.2
                w.color = RED # case 2.2
                RIGHT-ROTATE(T,w) # case 2.2
                w = x.p.right # case 2.2
            w.color = x.p.color # case 2.3
            x.p.color = BLACK # case 2.3
            w.right.color = BLACK # case 2.3
            LEFT-ROTATE(T, x.p) # case 2.3
            x = T.root # case 2.3
    else (same but exchange “right” and “left”)
x.color = BLACK
```

## Complexity

Complexity:

- Case 2.2-Case 2.3: directly solve the problem
- Case 1-Case 2.1: 1 level up or directly solve the problem

For each level up, the height of subtree rooted on x increase by 1

At most increase O(log n) times since it is a red-black tree when it stops.

```
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 2.1
            x = x.p # case 2.1
        else
            if w.right.color == BLACK
                w.left.color = BLACK # case 2.2
                w.color = RED # case 2.2
                RIGHT-ROTATE(T,w) # case 2.2
                w = x.p.right # case 2.2
            w.color = x.p.color # case 2.3
            x.p.color = BLACK # case 2.3
            w.right.color = BLACK # case 2.3
            LEFT-ROTATE(T, x.p) # case 2.3
            x = T.root # case 2.3
    else (same but exchange “right” and “left”)
x.color = BLACK
```

# Summary

- Binary Search Tree
- Red-Black Tree

# Thank you!

AIAA 5037  Advanced Algorithms and Data Structures