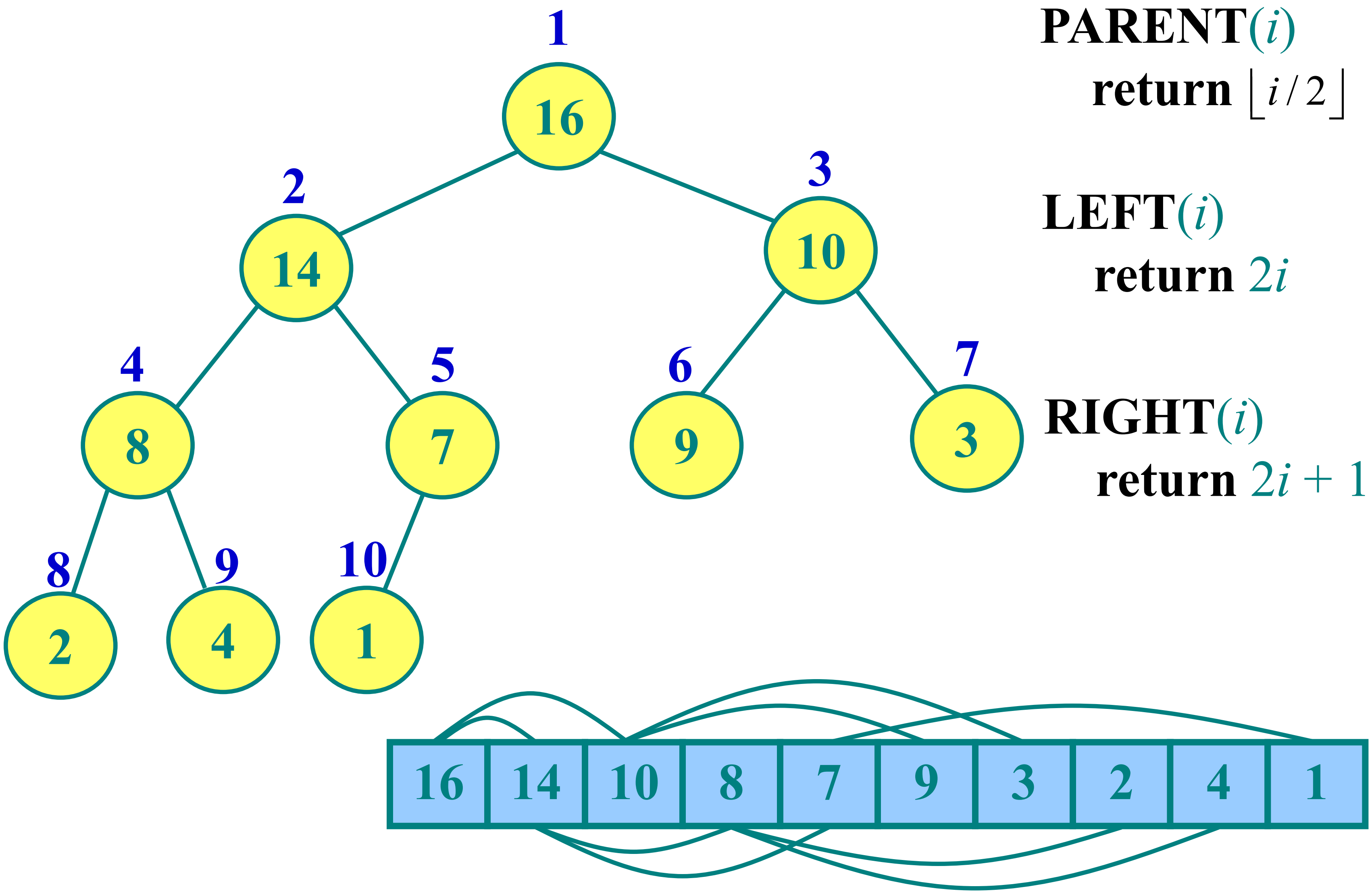
Data Structures and Algorithm

Xiaoqing Zheng
zhengxq@fudan.edu.cn

Trees (max heap)

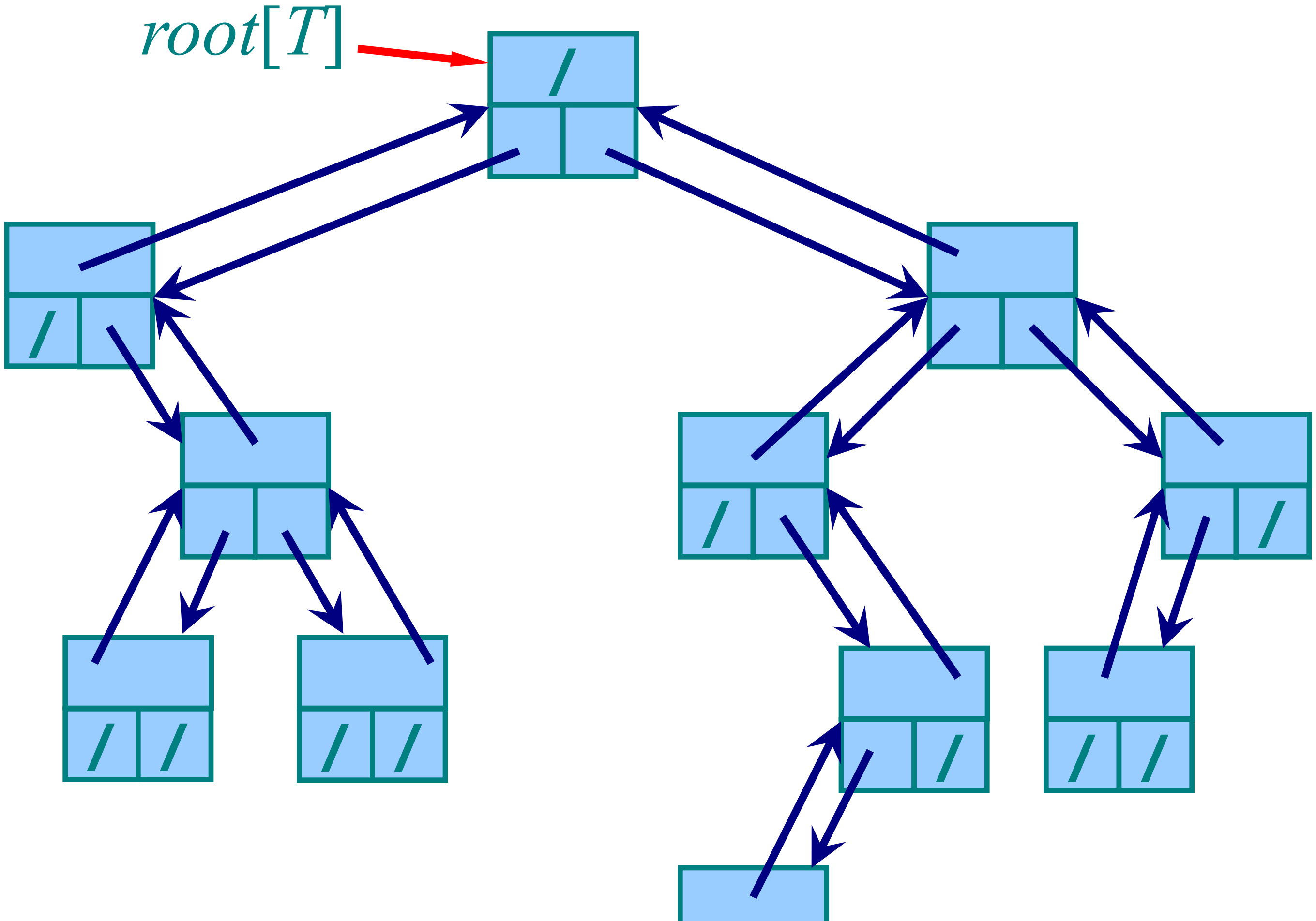


PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

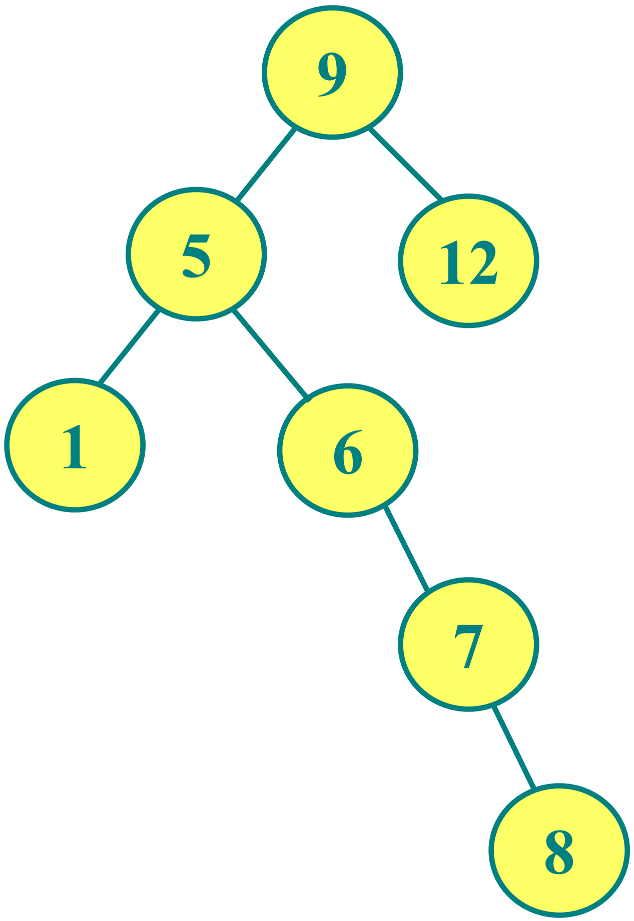
Binary trees



Not array!

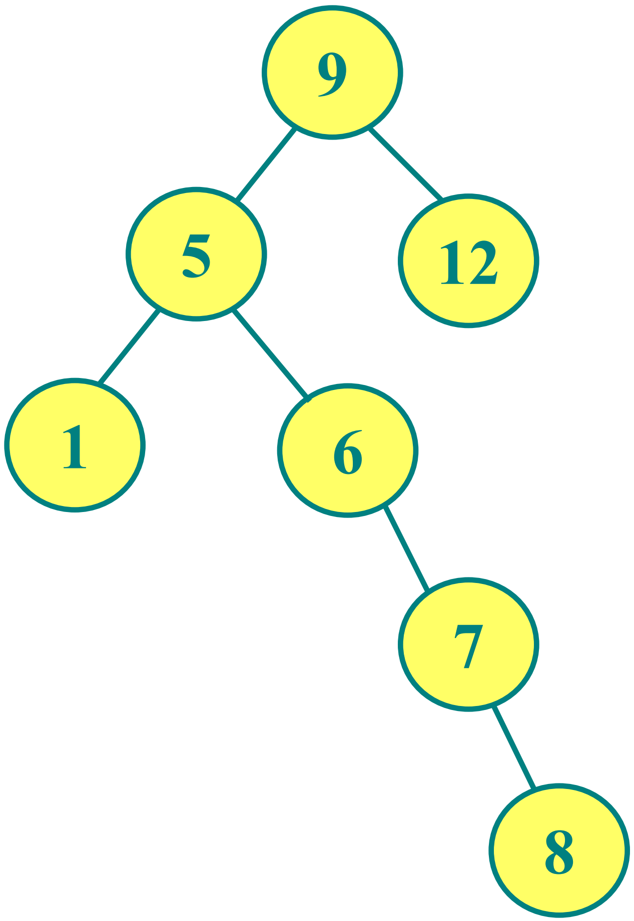
Binary Search Tree

- **Each node** *x* has:
  - *key*[*x*]
  - **Pointers**:
    - *left*[*x*]
    - *right*[*x*]
    - *p*[*x*]

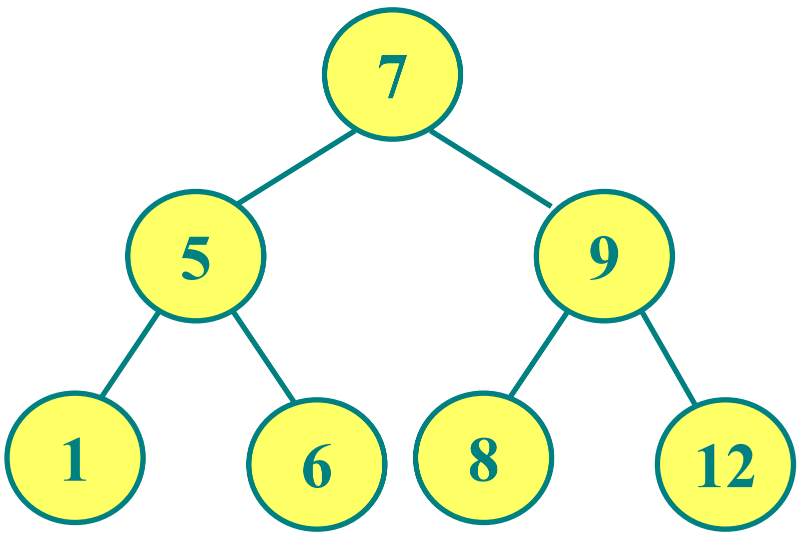


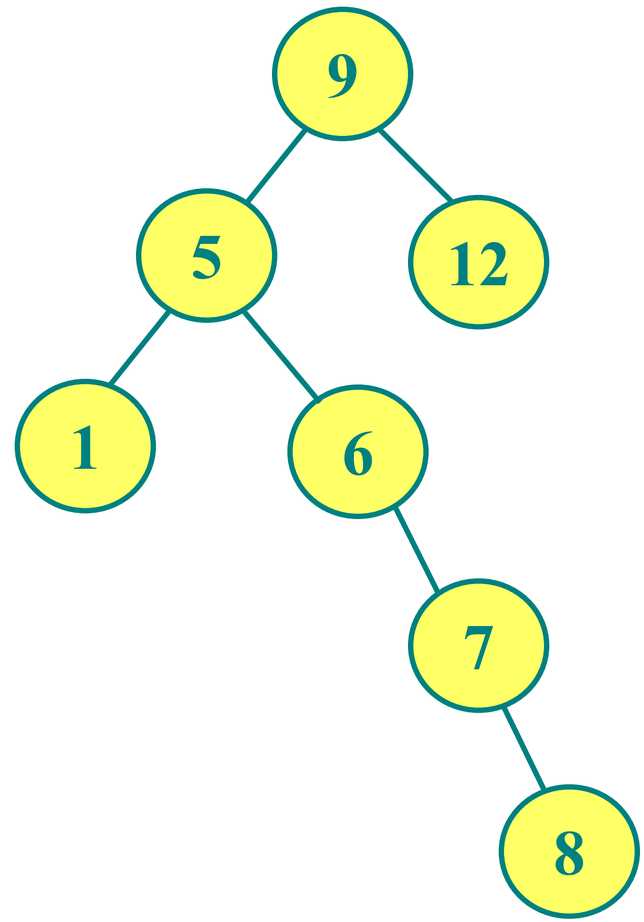
Binary Search Tree

- **Property**: for any node x:
  - For all nodes y in the left subtree of x:
    $key[y] \leq key[x]$
  - For all nodes y in the right subtree of x:
    $key[y] \geq key[x]$
- Given a set of keys, is BST for those keys ***unique***?



No uniqueness





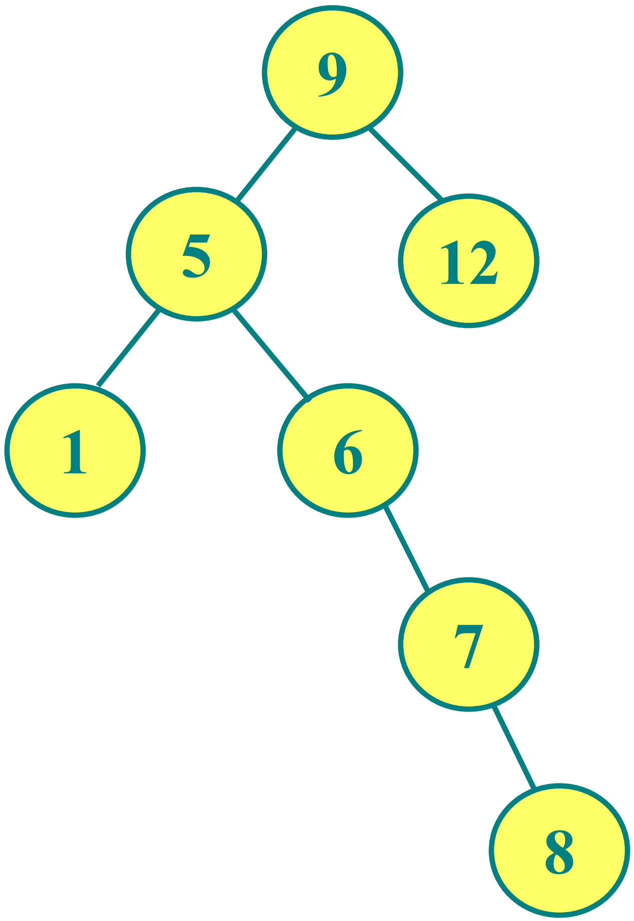
What can we do given BST ?

Sort !

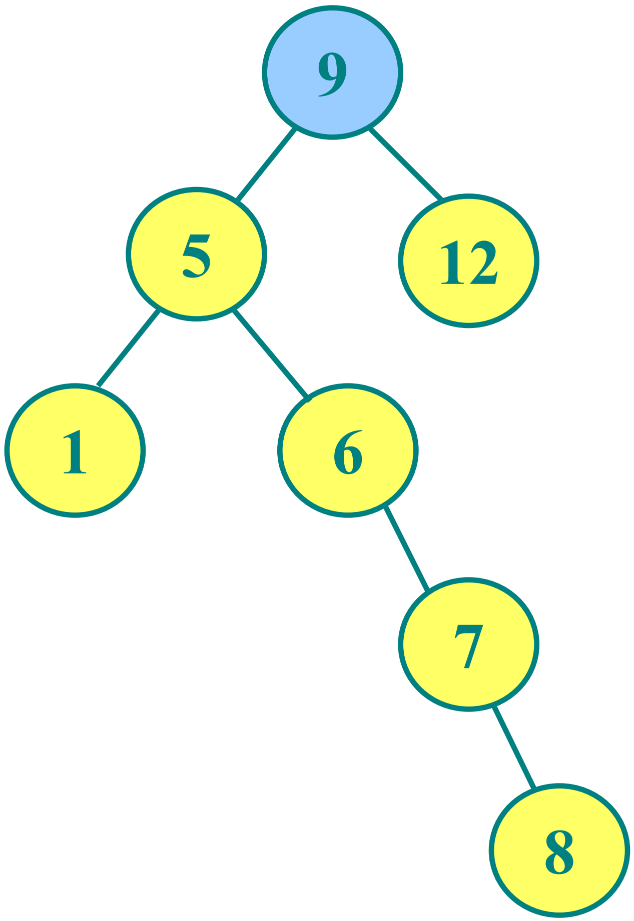
INORDER-TREE-WALK(x)
1. **if** $x \neq$ NIL
2. **then** INORDER-TREE-WALK(*left*[x])
3. print *key*[x]
4. INORDER-TREE-WALK(*right*[x])

A ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.

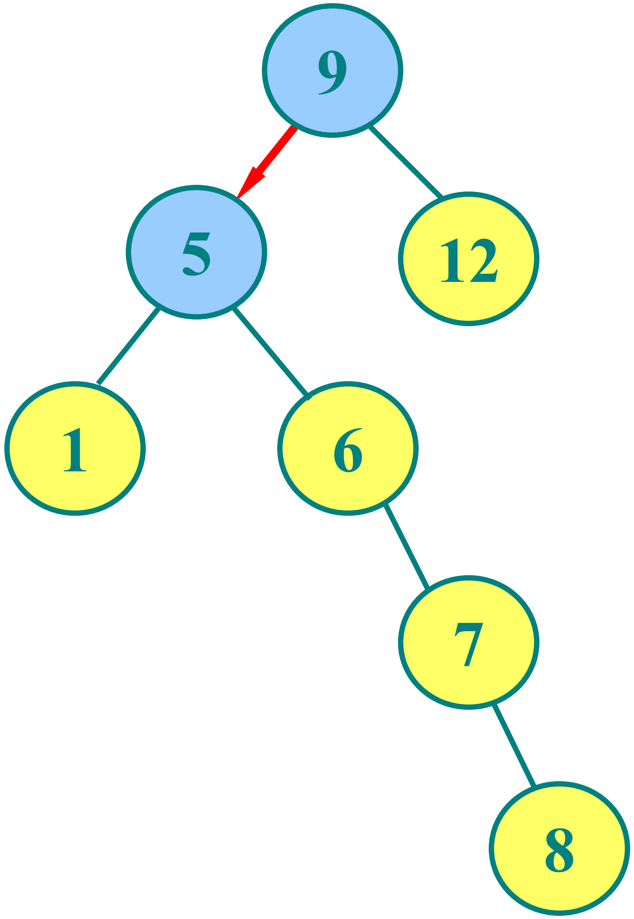
Sort ?



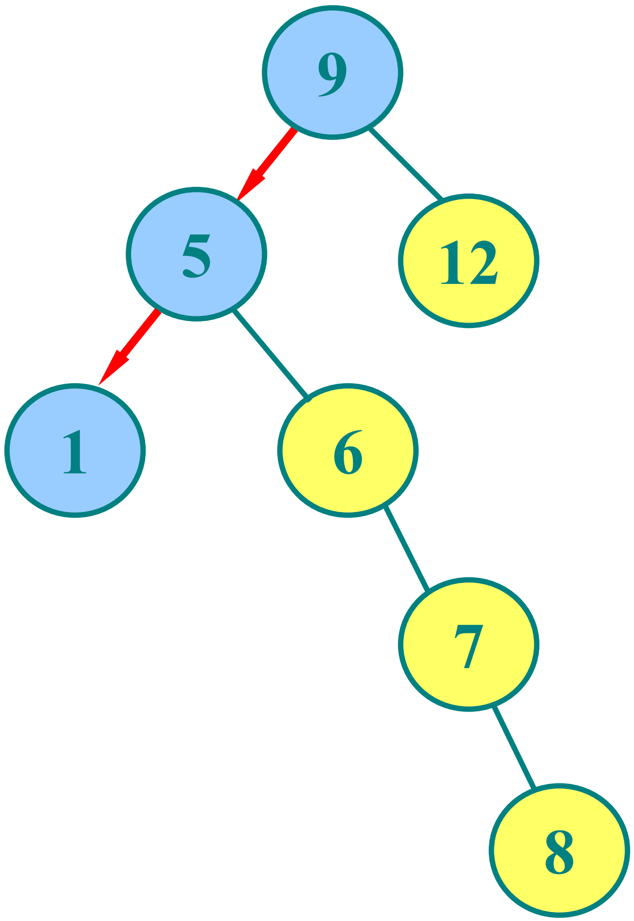
Sort ?



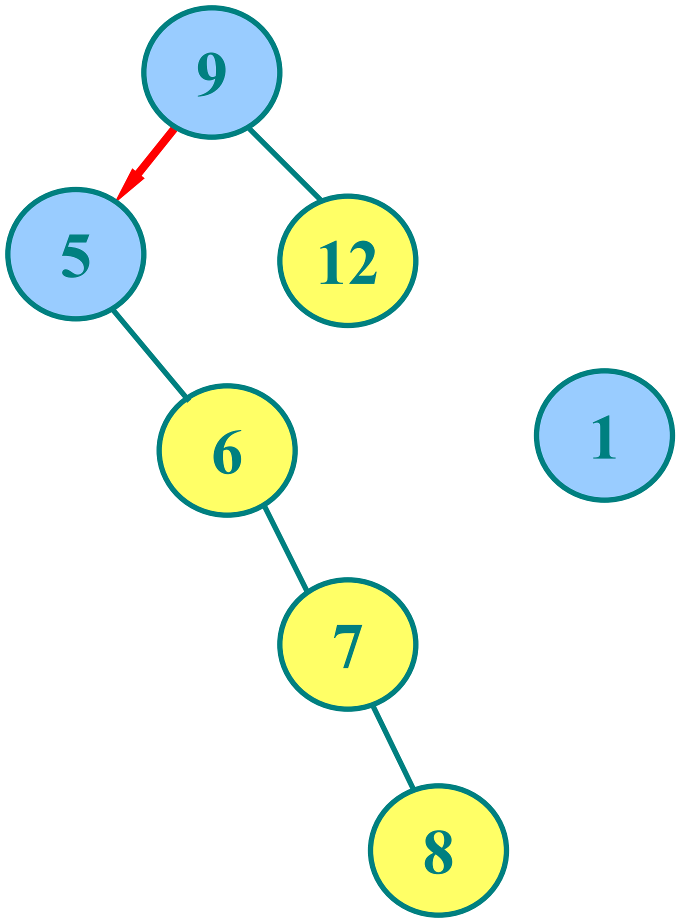
Sort ?



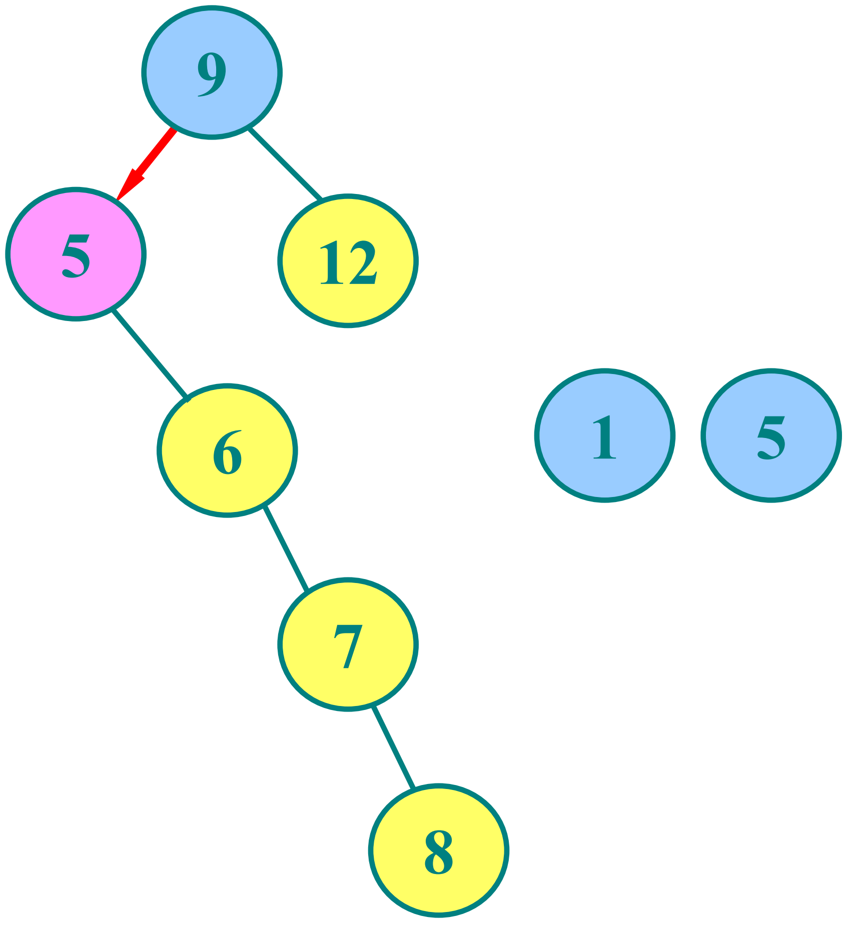
Sort ?



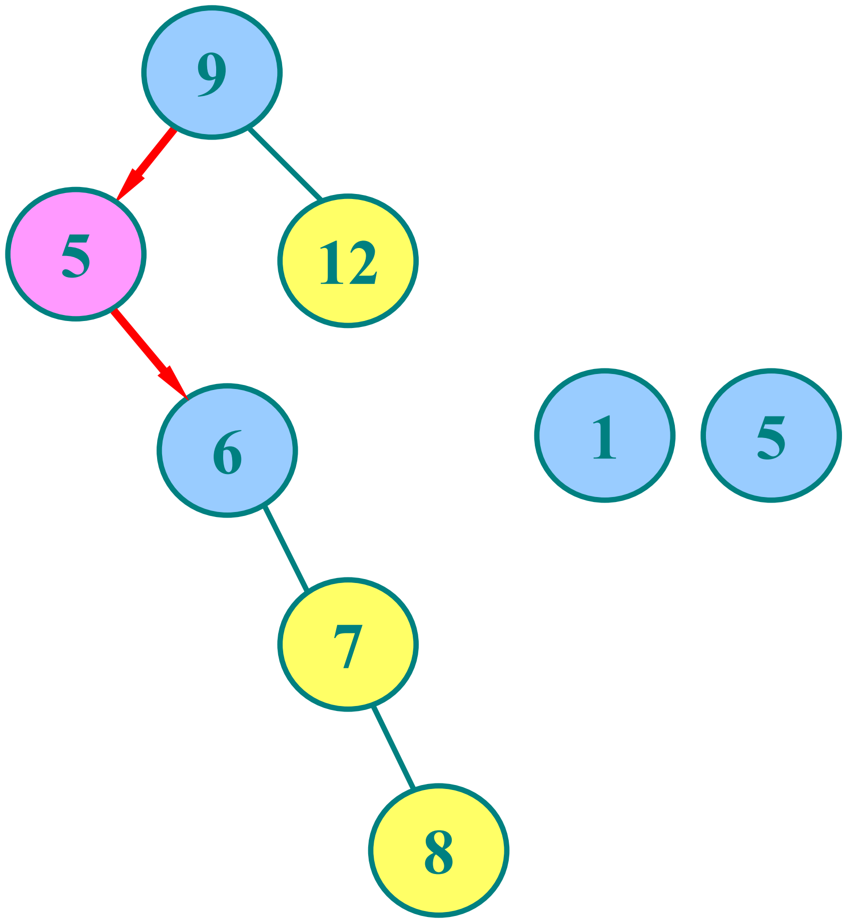
Sort ?



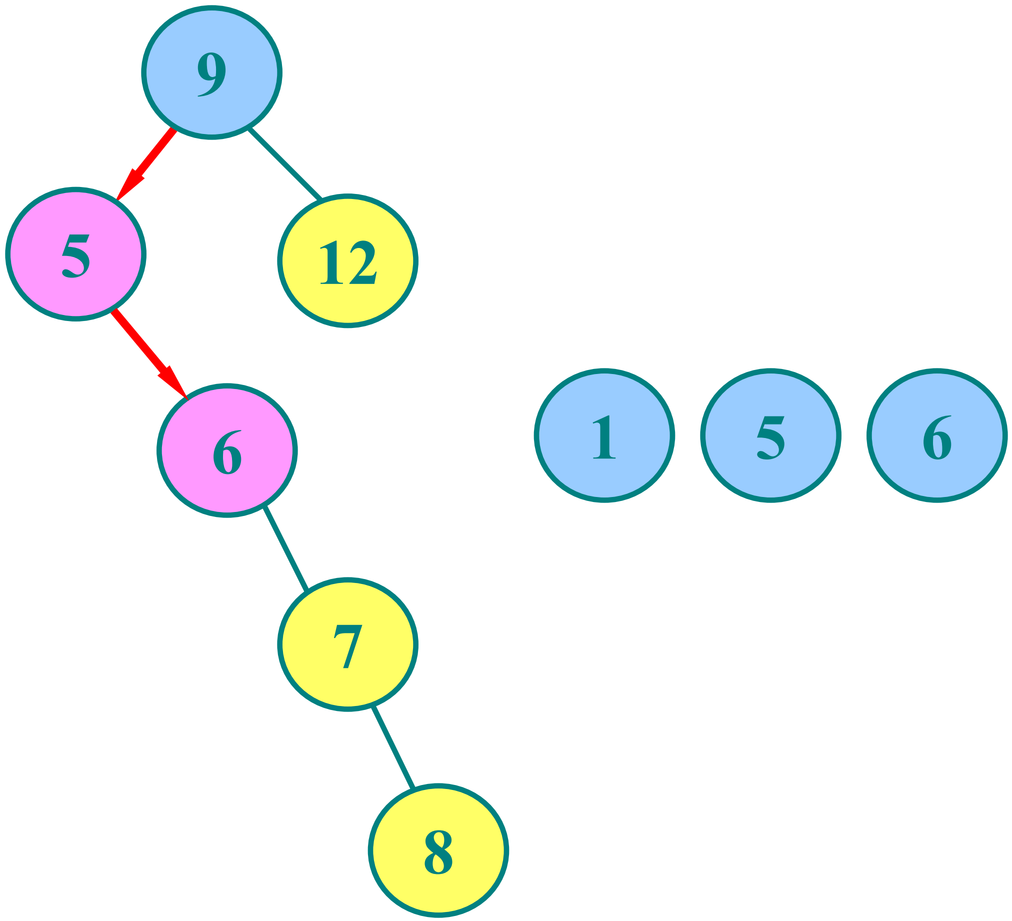
Sort ?



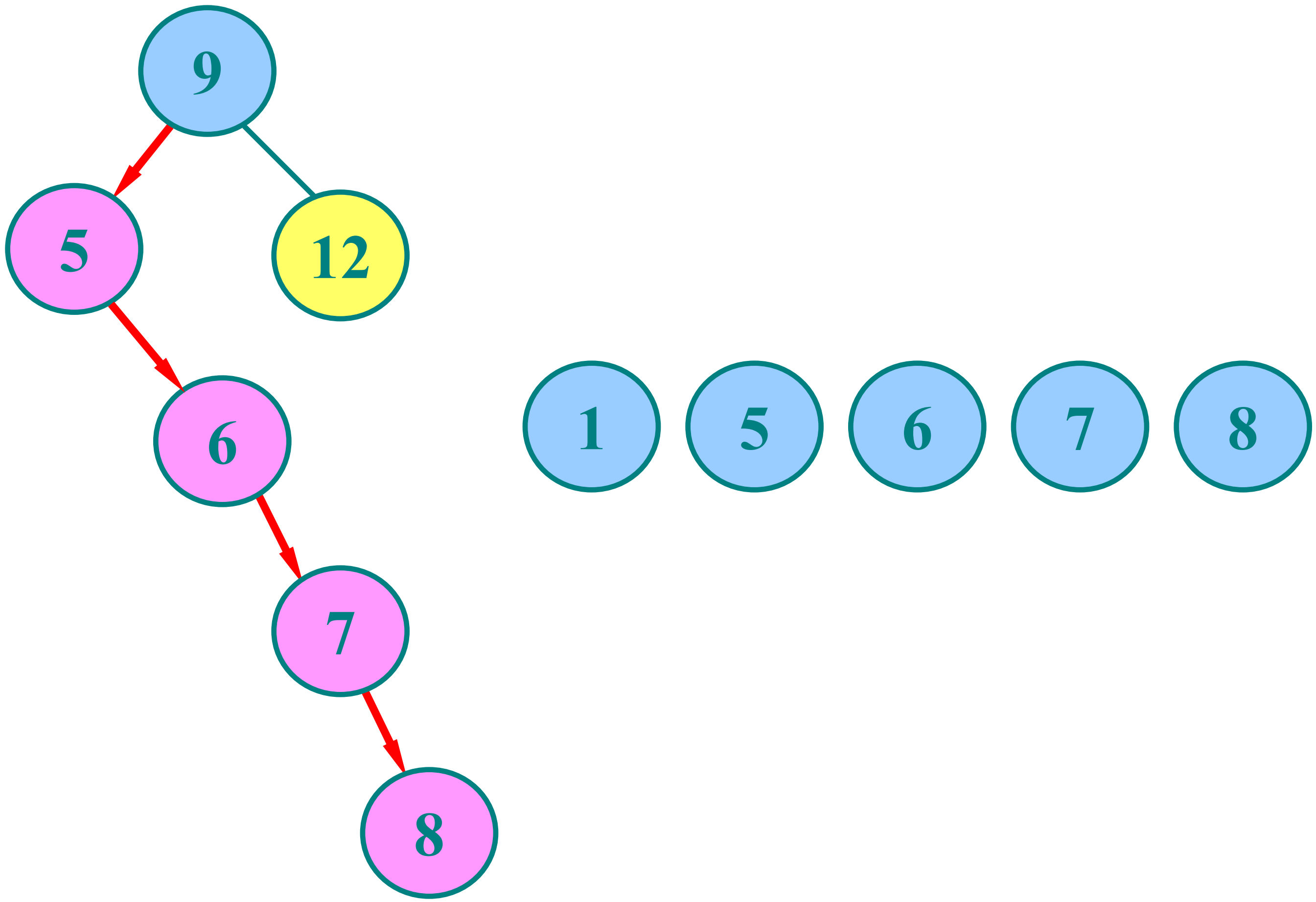
Sort ?



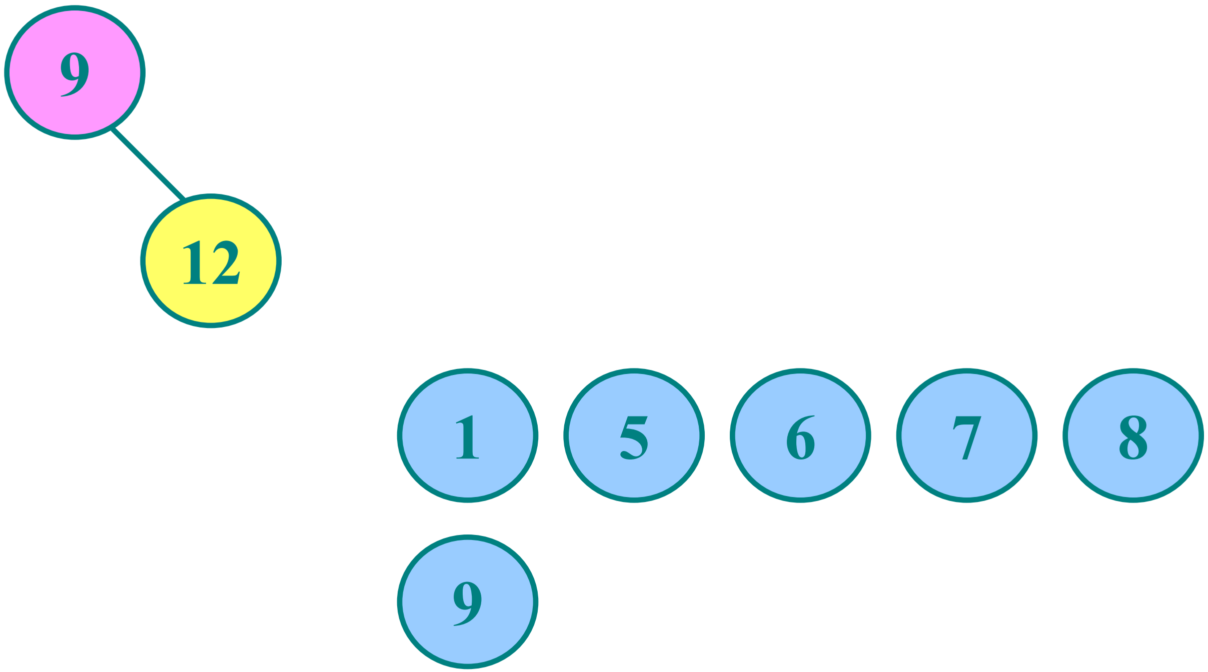
Sort ?



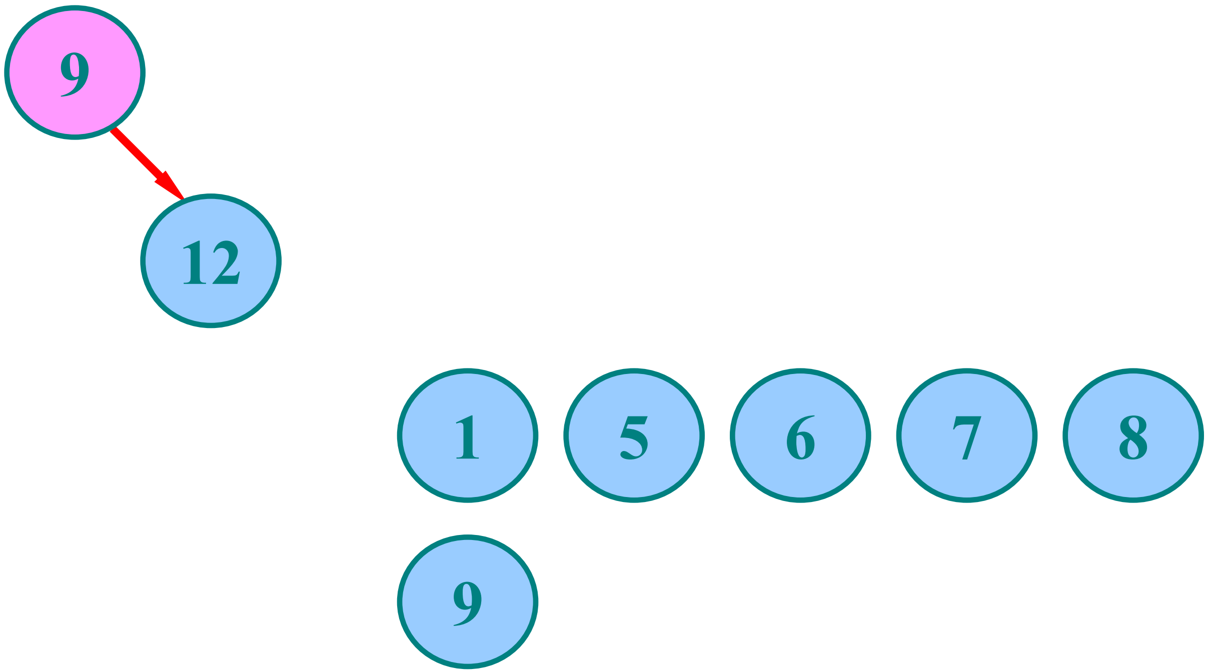
Sort ?



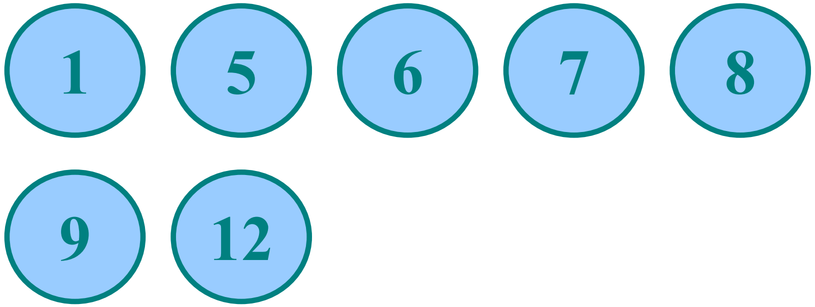
Sort ?



Sort ?



Sort ?



Analysis of inorder-walk

Theorem. If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ times.

Substitution method

$$T(n) = (c + d)n + c$$

Base case: $n = 0$, $T(0) = (c + d) \cdot 0 + c = c$

For $n > 0$,

$$\begin{aligned} T(n) &= T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d) \cdot (n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c \end{aligned}$$

Sorting

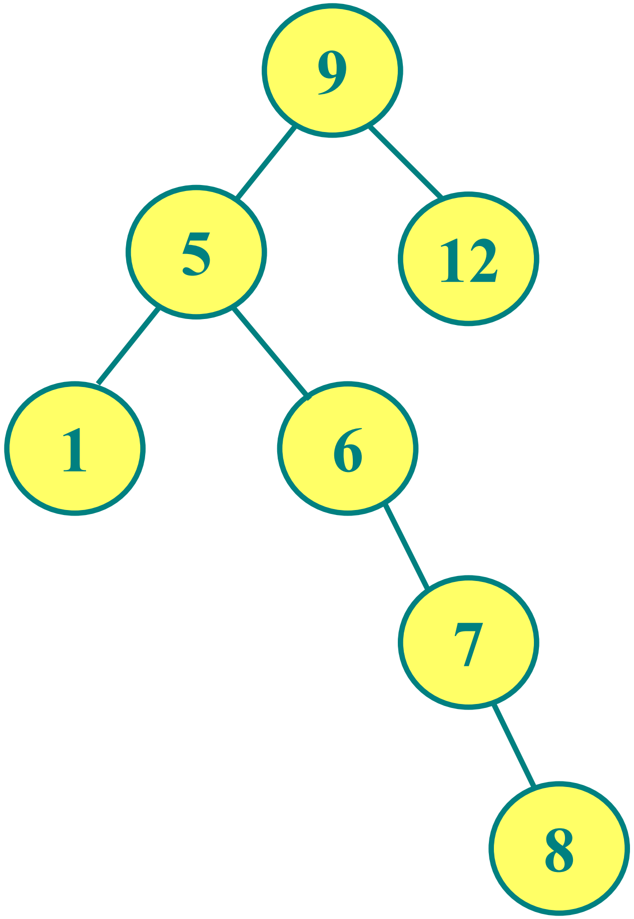
Does it mean that we can sort n keys in $O(n)$ time?

No.

It just means that building a binary search tree takes $\Omega(nlgn)$ time (in the comparison model)

BST as a data structure

- **Operations**:
  - *Insert*(*x*)
  - *Delete*(*x*)
  - *Search*(*k*)



Search

TREE-SEARCH(x, k)

1. **if** x = NIL or $k = key[x]$
2. **then return** x
3. **if** $k < key[x]$
4. **then return** TREE-SEARCH($left[x]$, k)
5. **else return** TREE-SEARCH($right[x]$, k)

Search

ITERATIVE-TREE-SEARCH(x, k)

1. **while** $x \neq \text{NIL}$ **and** $k \neq key[x]$
2. **do if** $k < key[x]$
3. **then** $x \leftarrow left[x]$
4. **else** $x \leftarrow right[x]$
5. **return** x

On most computers, this version is more efficient.

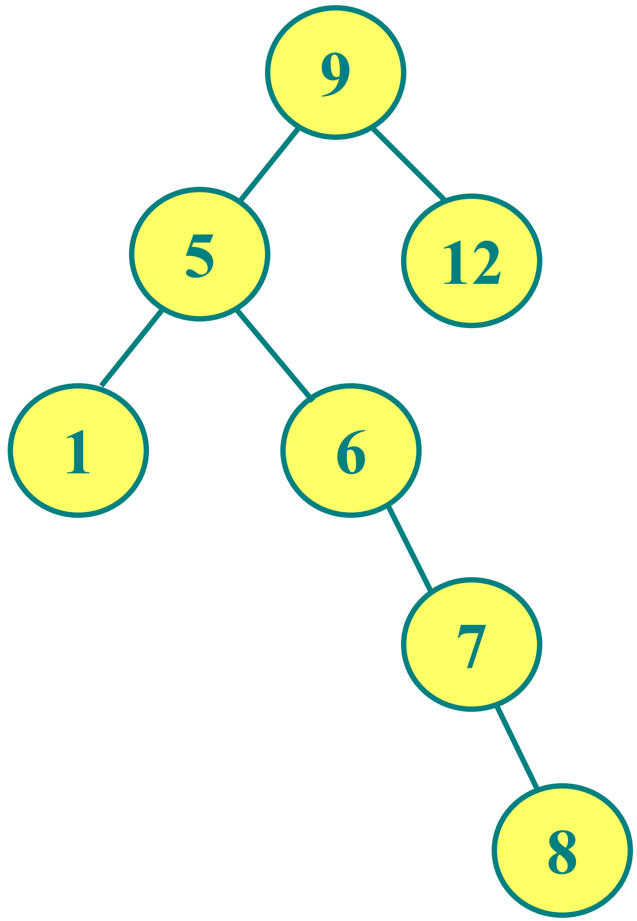
Minimum and maximum

TREE-MINIMUM(x)

1. **while** $left[x] \neq$ NIL
2.   **do** $x \leftarrow left[x]$
3. **return** x

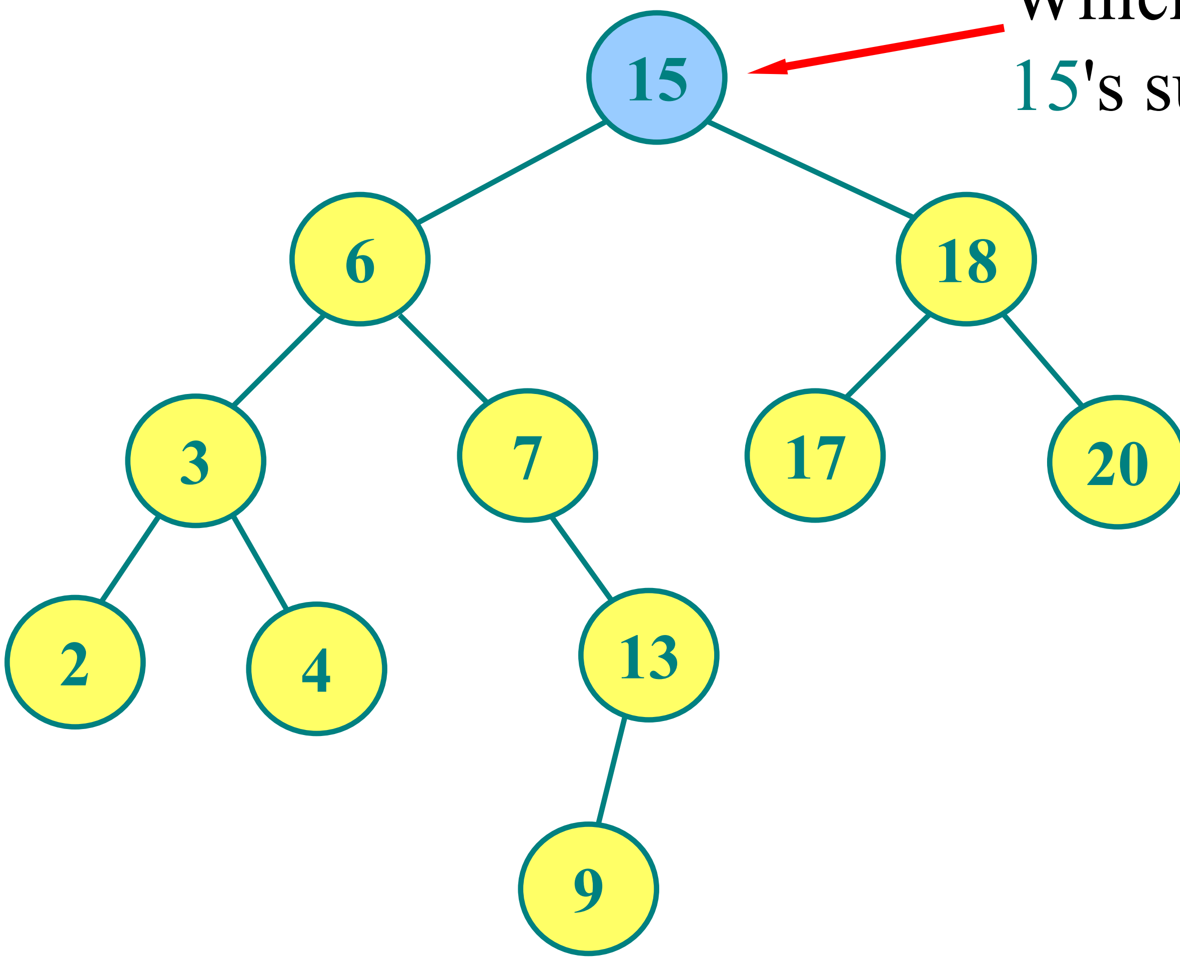
TREE-MAXIMUM(x)

1. **while** $right[x] \neq$ NIL
2.   **do** $x \leftarrow right[x]$
3. **return** x

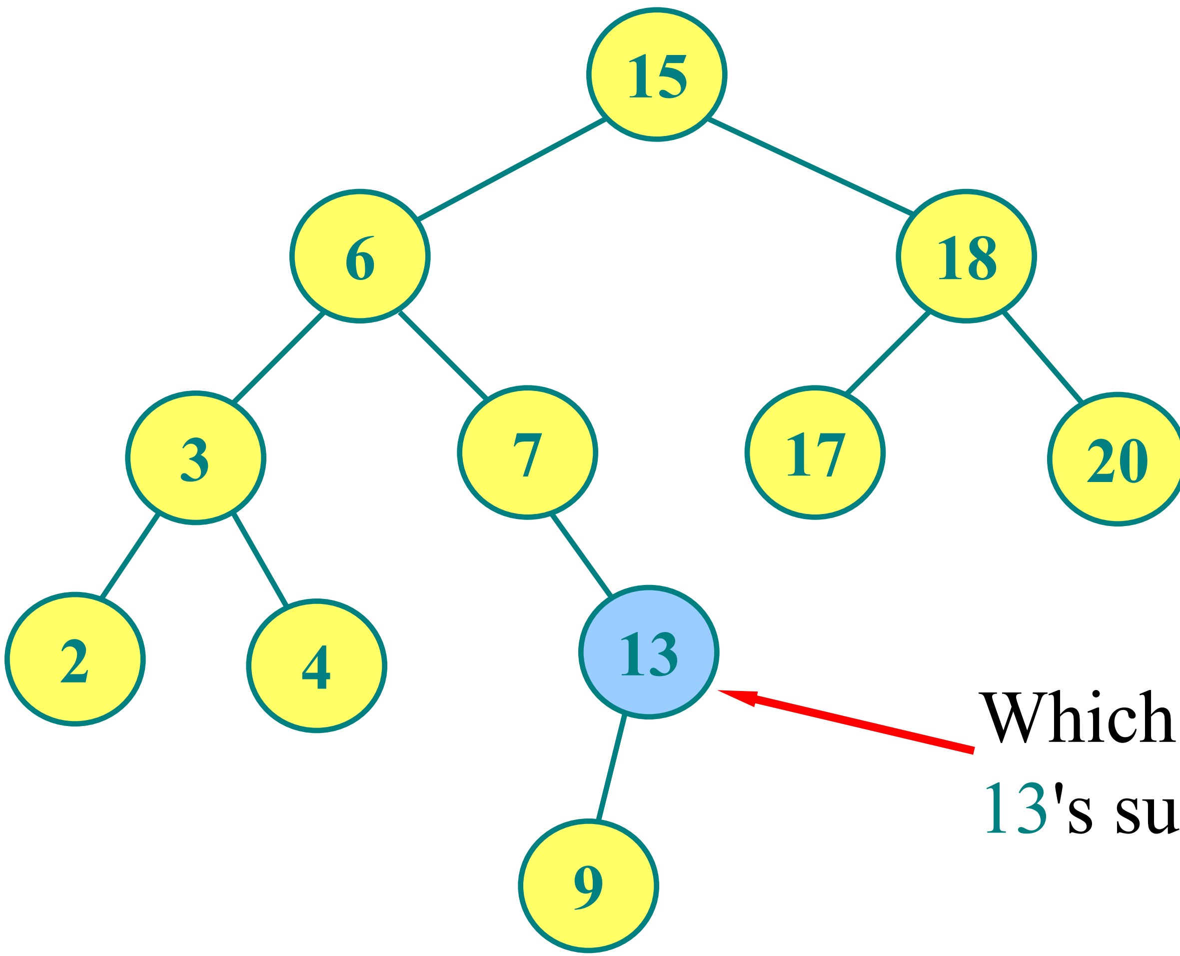


Successor and predecessor

Which is the node 15's successor



Successor and predecessor



Which is the node 13's successor

Successor and predecessor

TREE-SUCCESSOR(x)

1. **if** $right[x] \neq \text{NIL}$
2. **then return** TREE-MINIMUN$(right[x])$
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{NIL}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[x]$
7. **return** y

Running time

$O(h)$

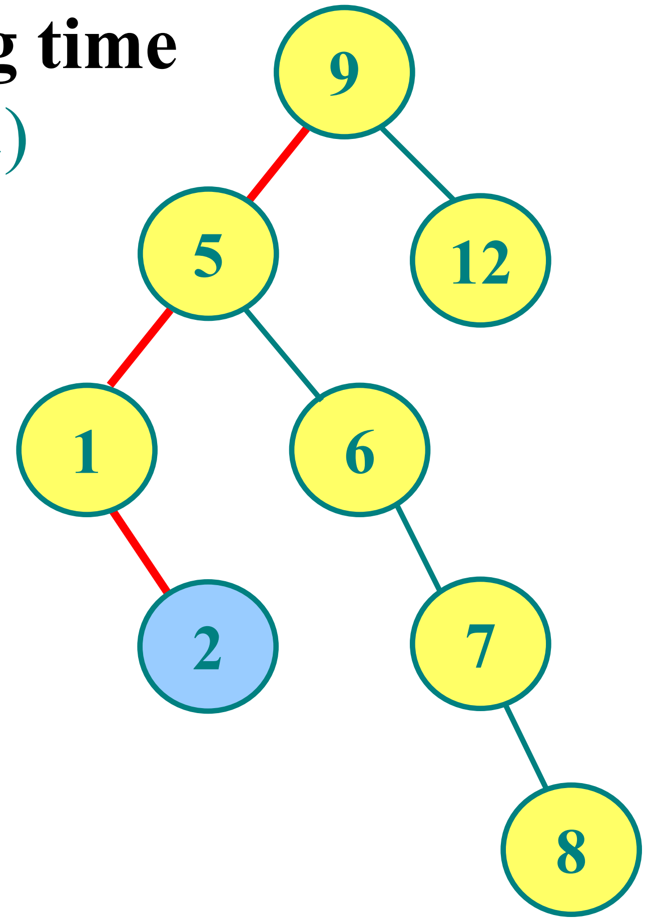
Constructing BST

TREE-INSERT(T, z)

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

Running time

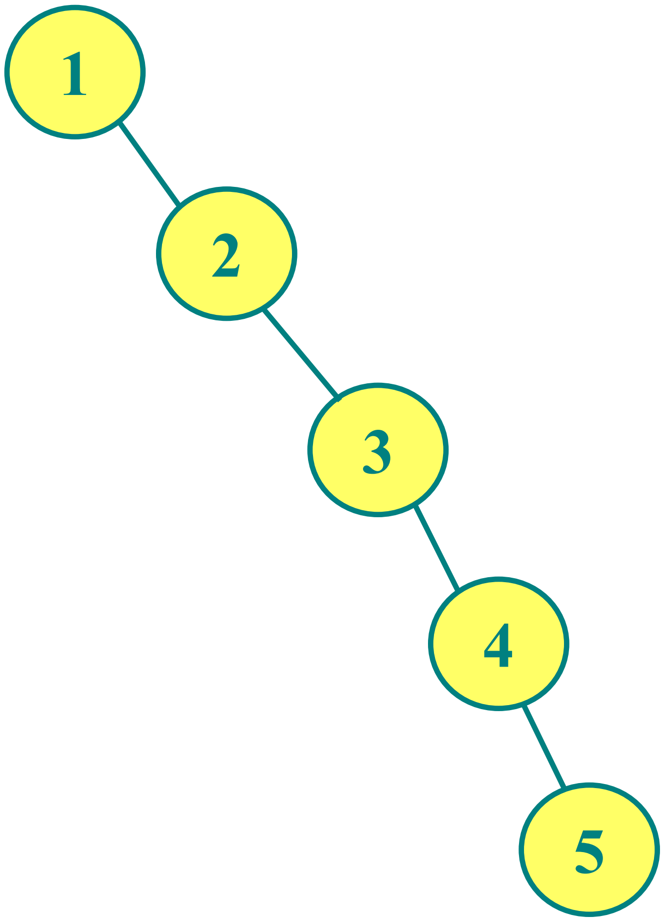
$O(h)$



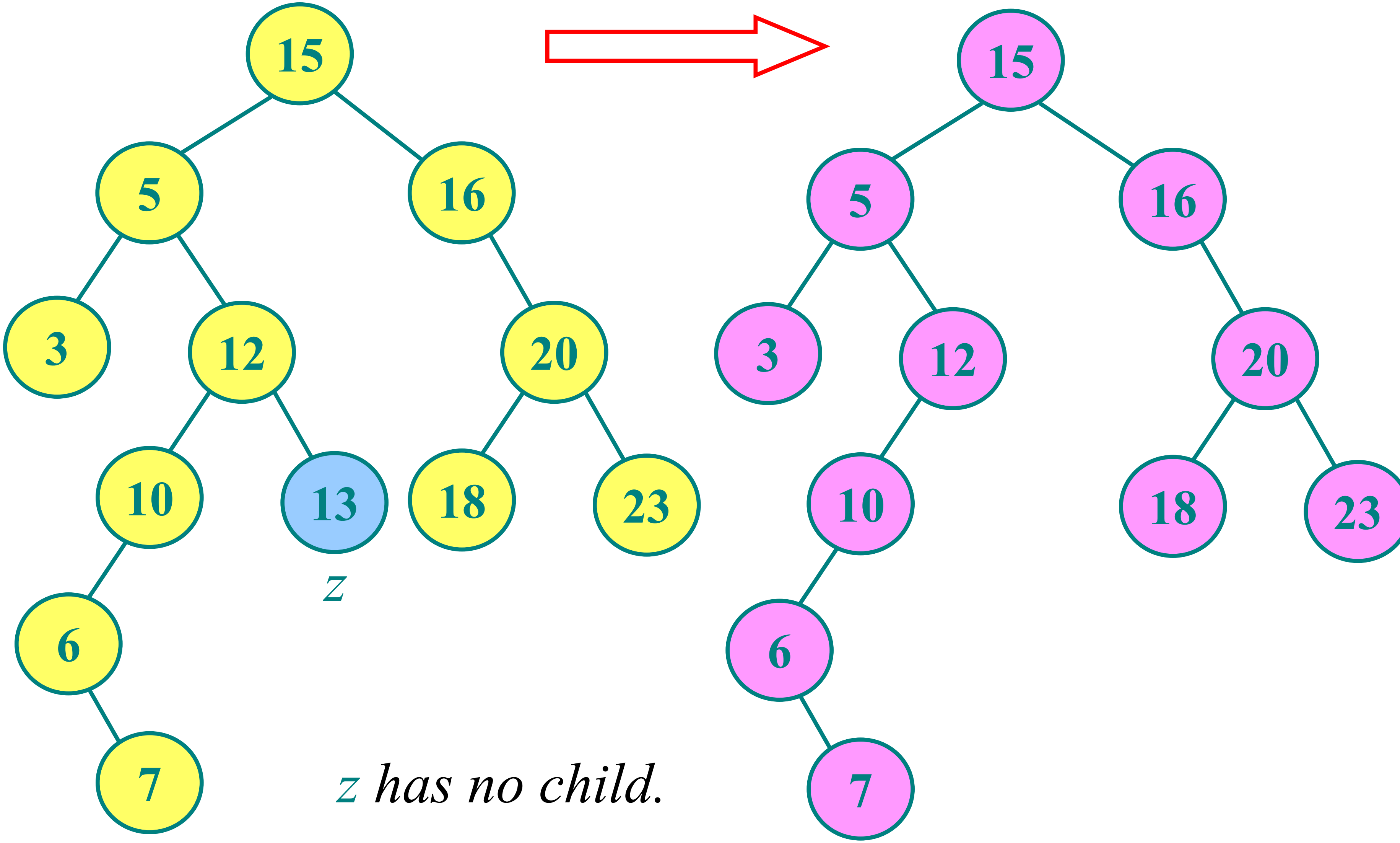
TREE-INSERT$(T, 2)$

Analysis

- After we insert n elements, what is the worst possible BST height?
- Pretty bad: $n - 1$
- **Average:** $O(nlgn)$

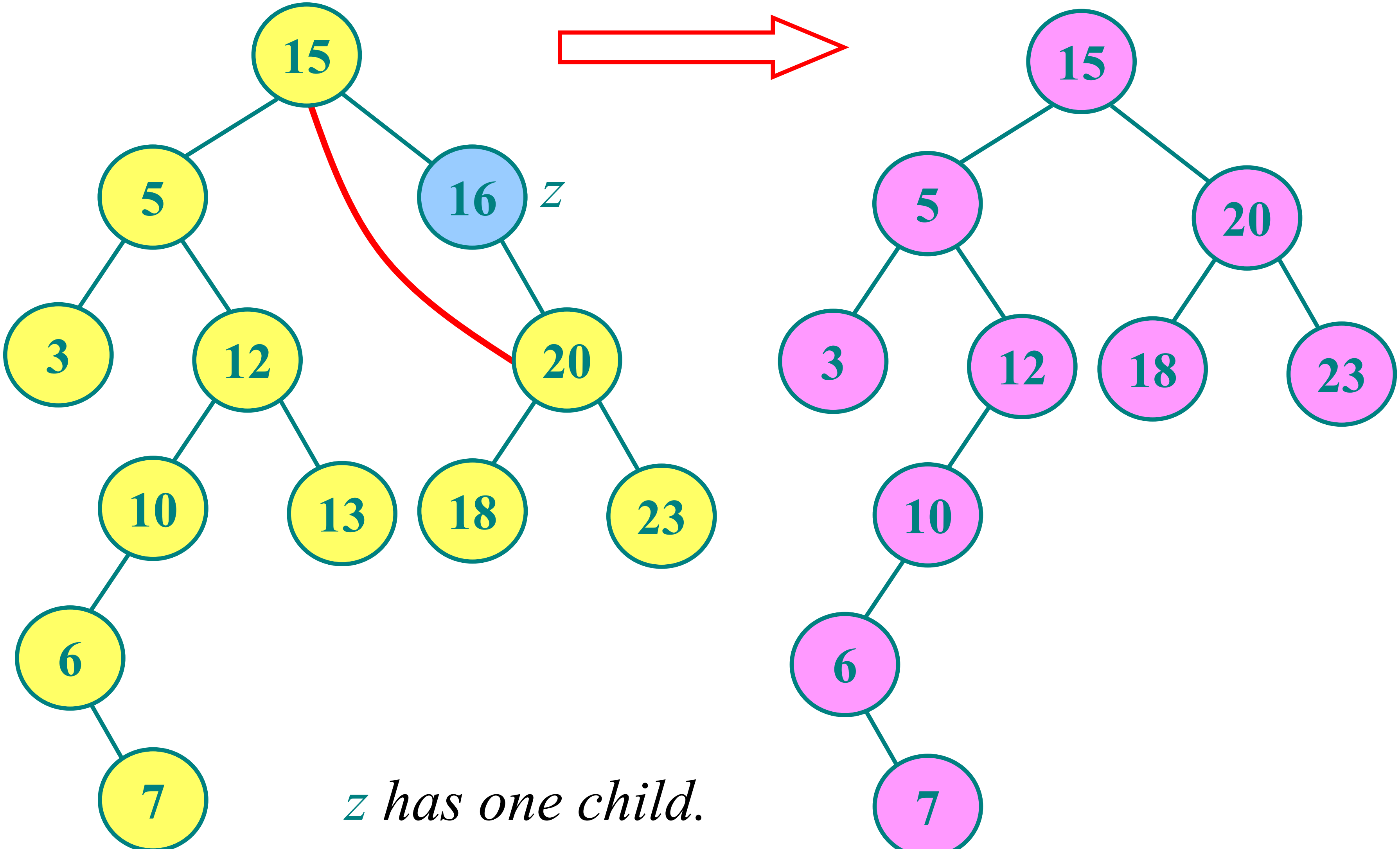


Deletion (case 1)



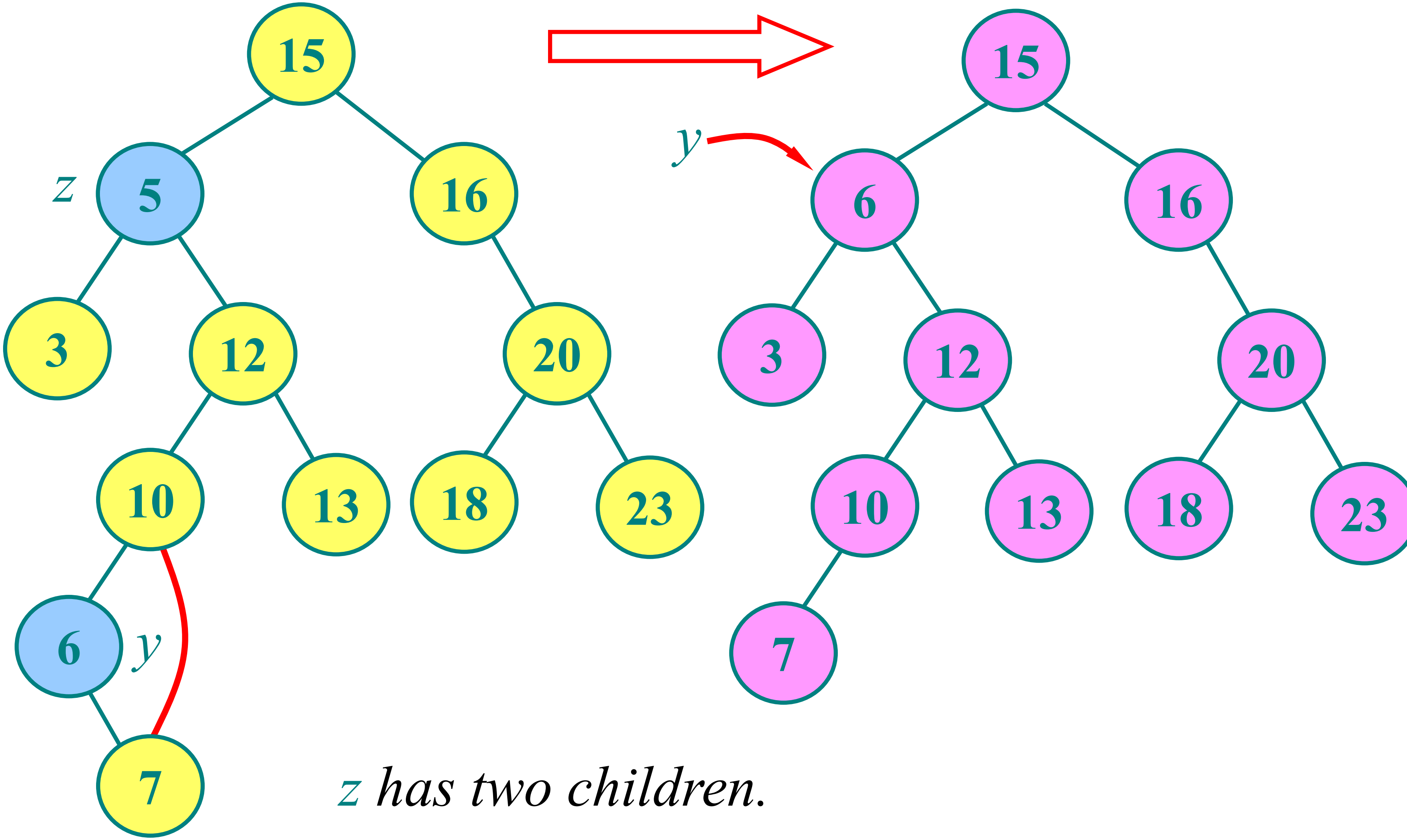
z has no child.

Deletion (case 2)



z has one child.

Deletion (case 3)



z has two children.

Deletion

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    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]
return y
```

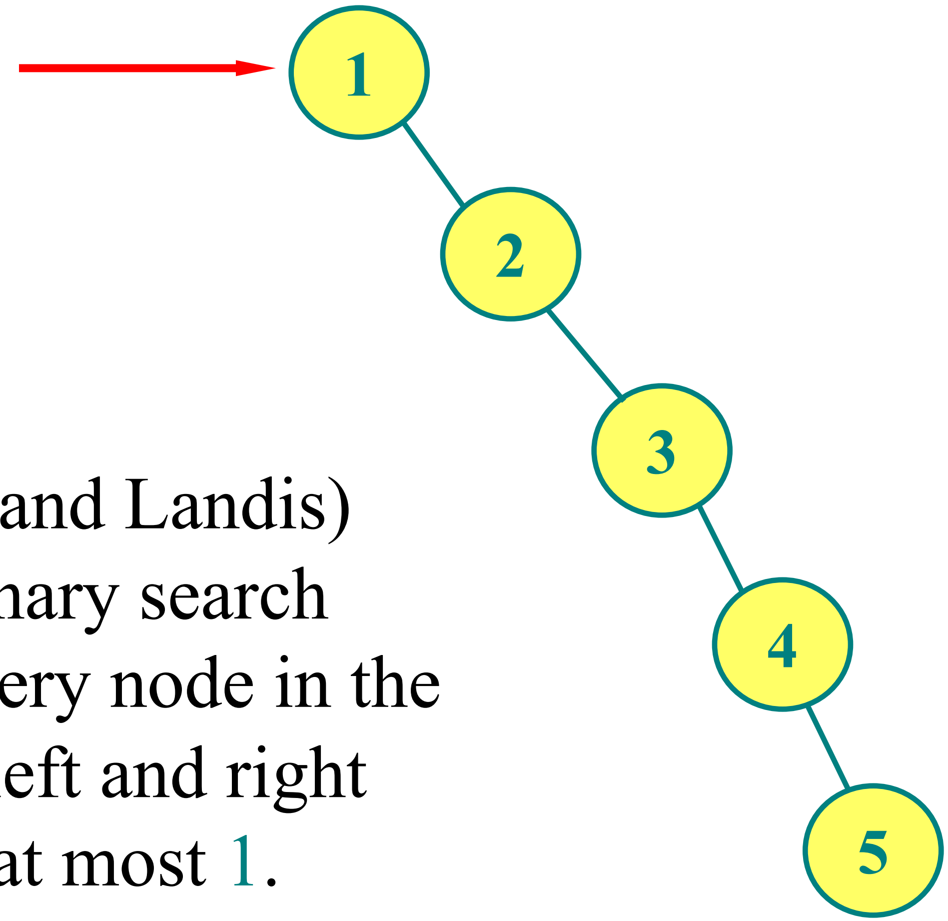
Running time:

$O(h)$

Note: *z's successor just has one child or z has one child.*

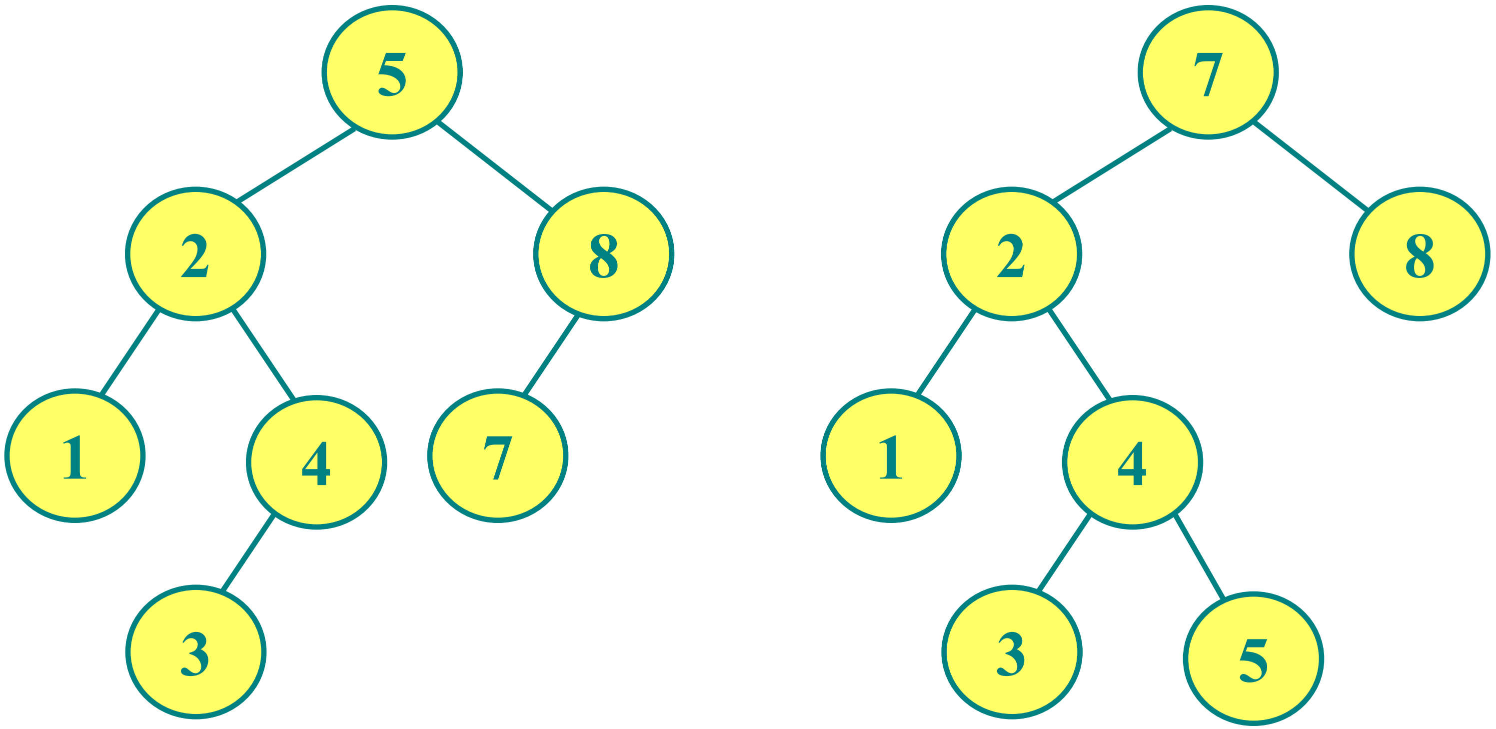
Balanced search trees

Balanced search trees,
or how to avoid this
even in the worst case



AVL (Adelson-Veskii and Landis) tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1.

AVL trees



Which one is AVL tree?

AVL trees

A ***violation*** might occur in ***four case*** when we insert new node to the AVL tree.

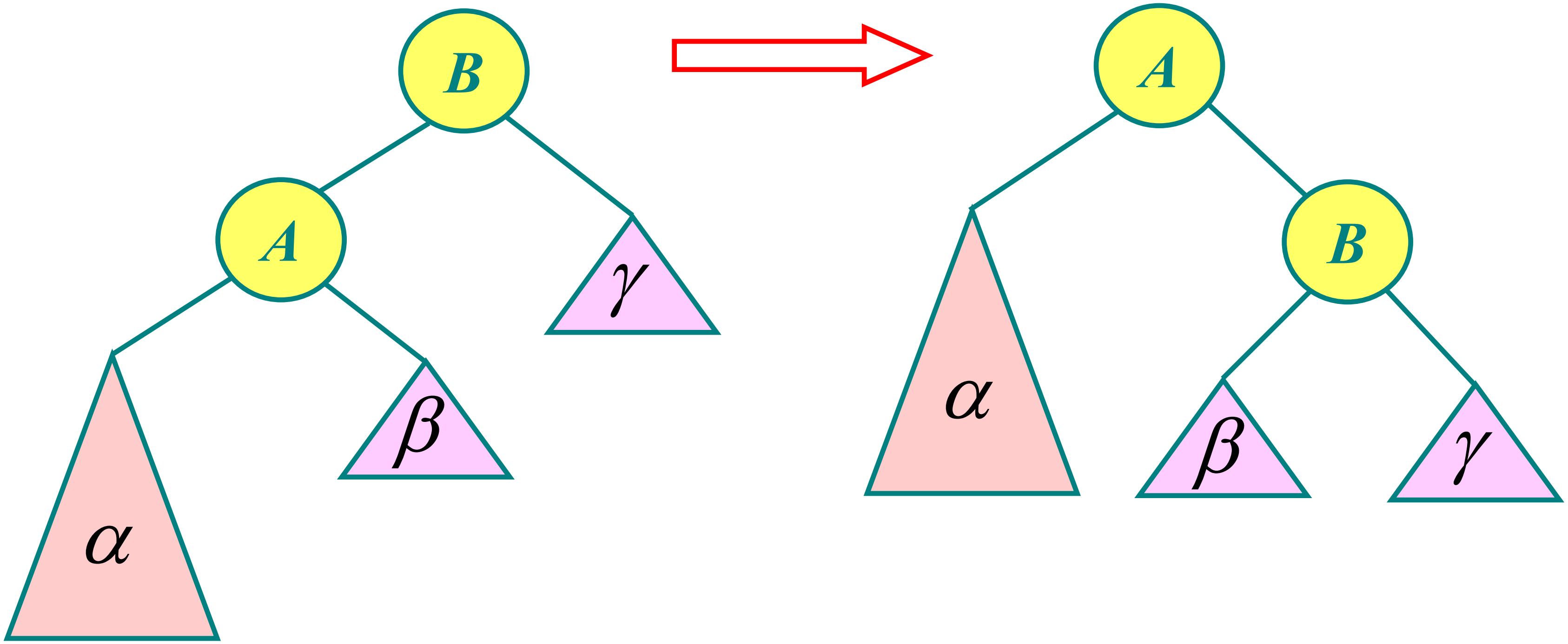
Case 1: an insertion into the left subtree of the left child of *R*.

Case2 : an insertion into the right subtree of the left child of *R*.

Case 3: an insertion into the left subtree of the right child of *R*.

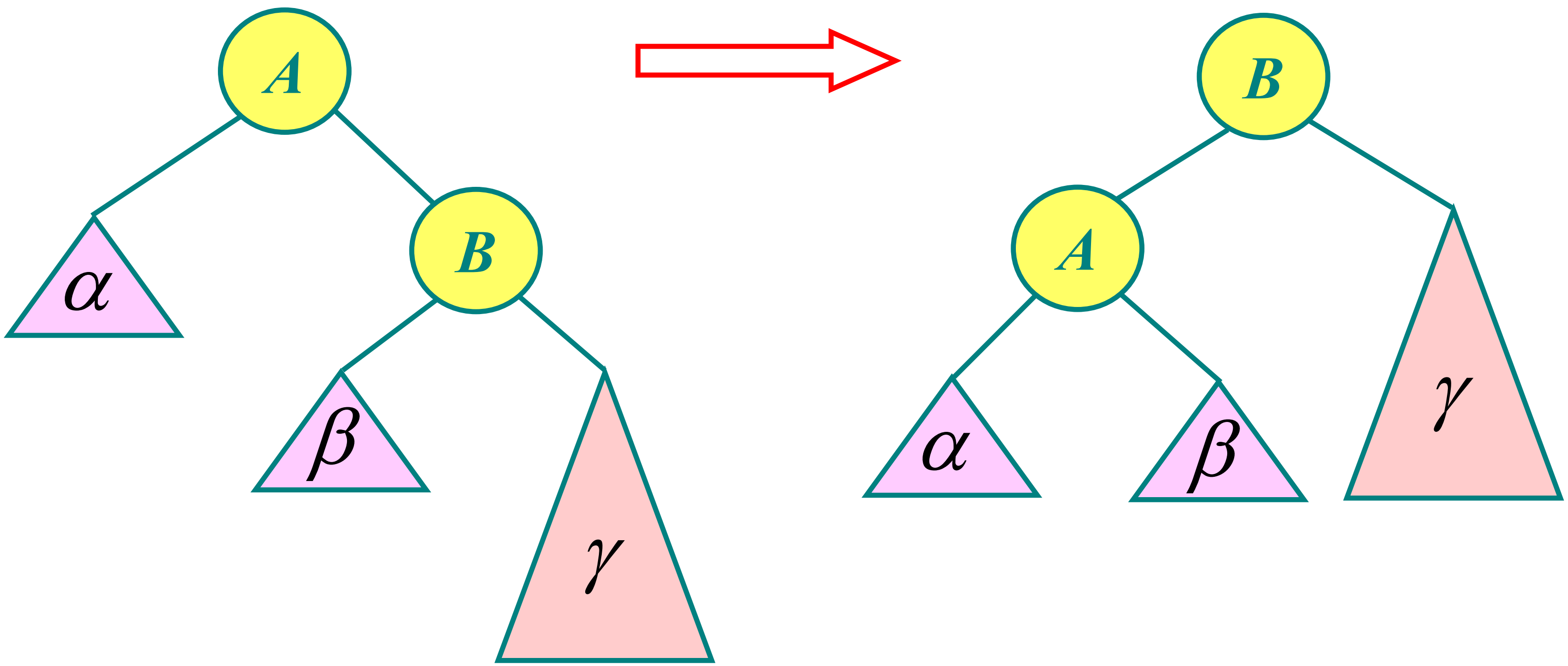
Case 4: an insertion into the right subtree of the right child of *R*.

Single rotation



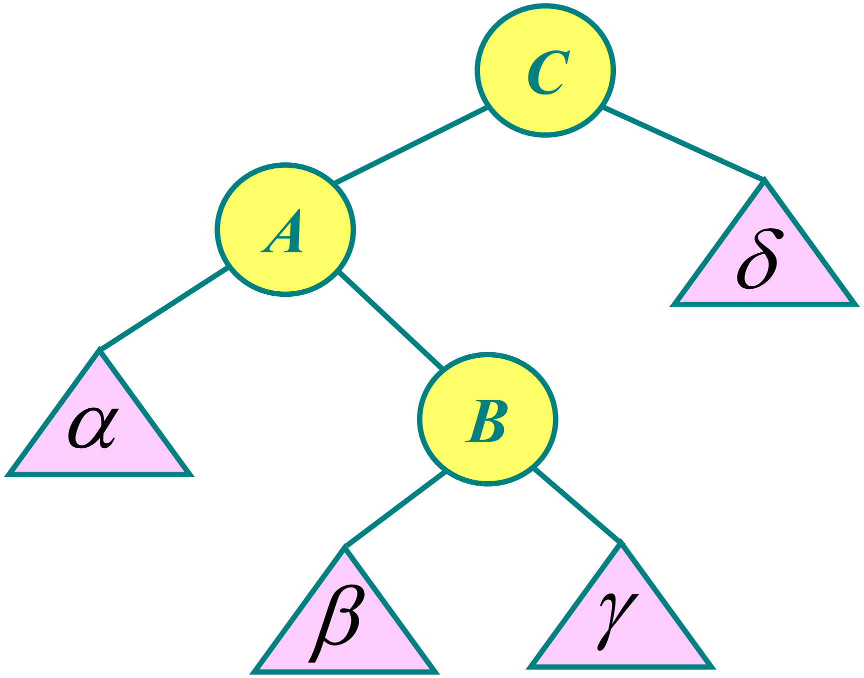
Right rotation to fix case 1

Single rotation



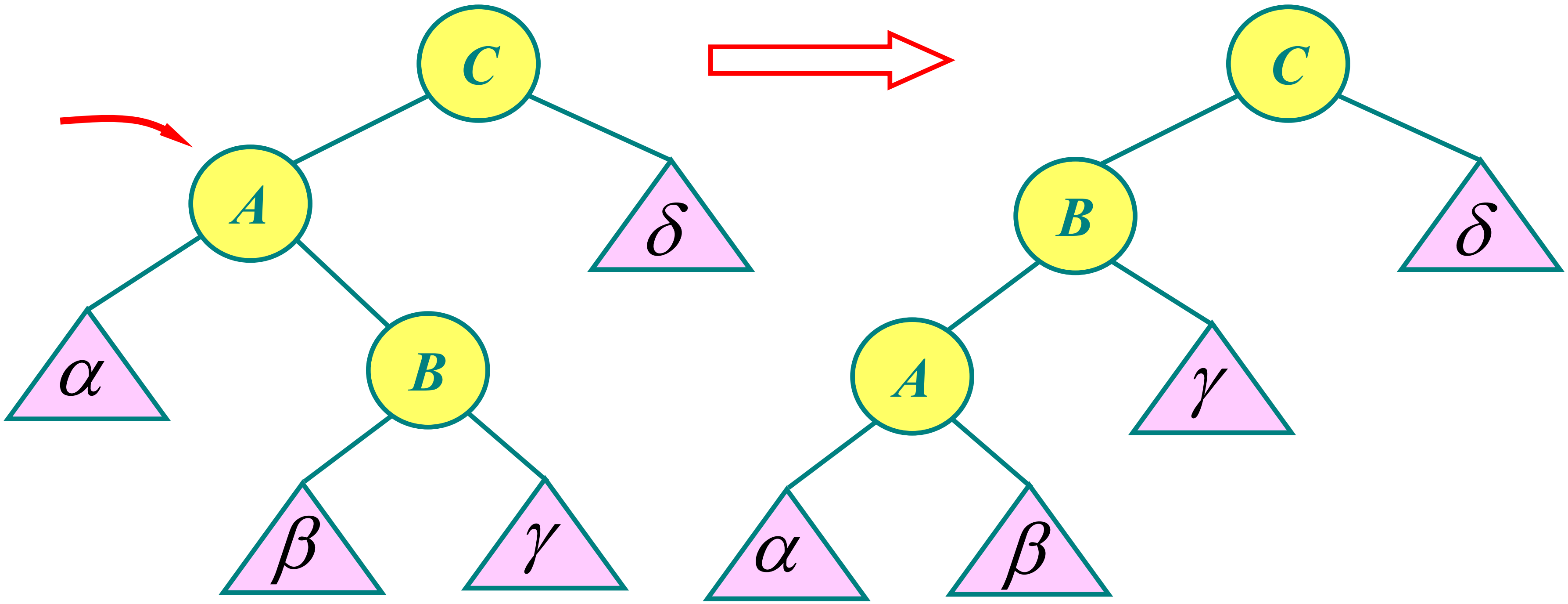
Left rotation to fix case 4

Double rotation



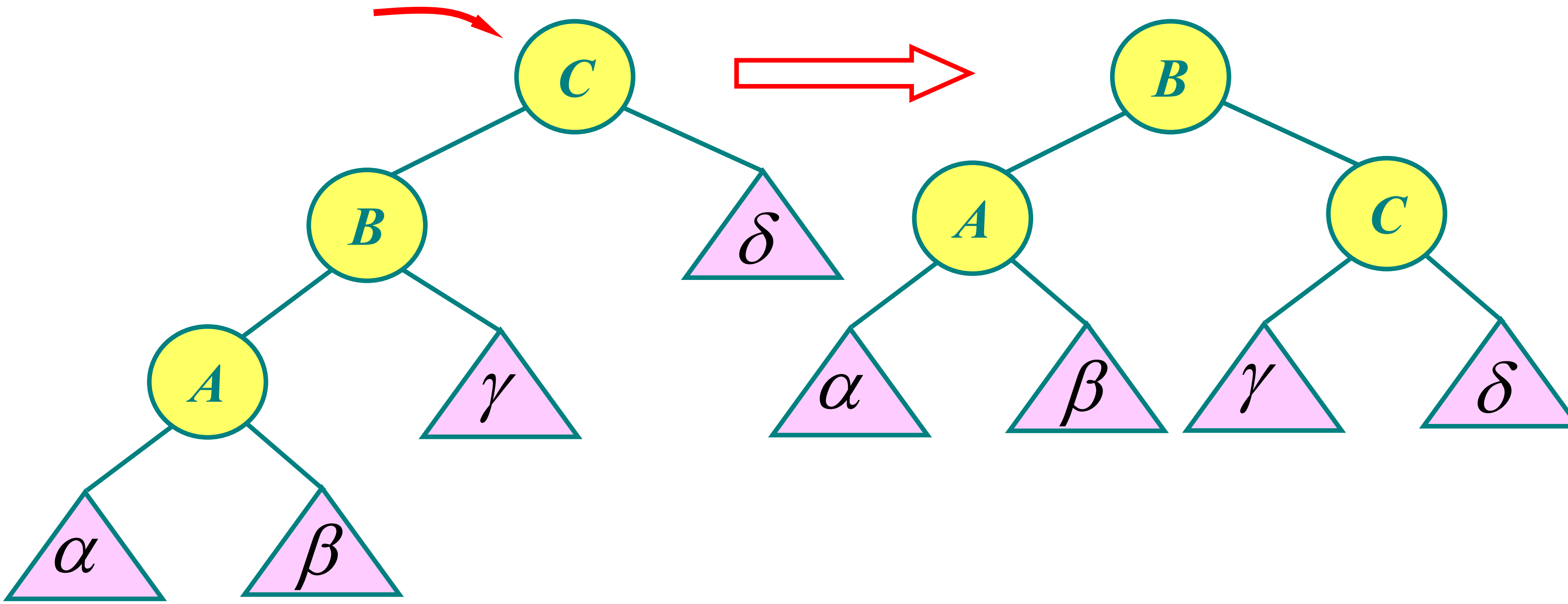
Single rotation fails to fix case 2

Double rotation (first step)



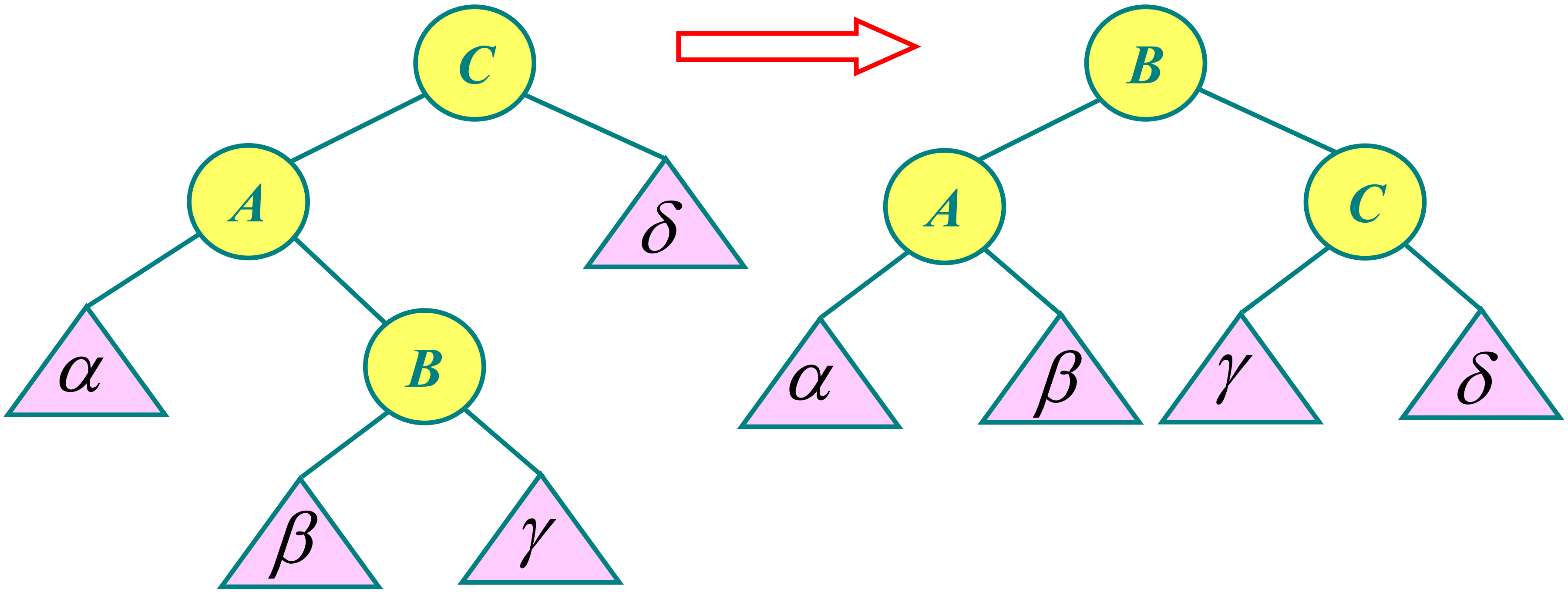
Left rotation

Double rotation (second step)



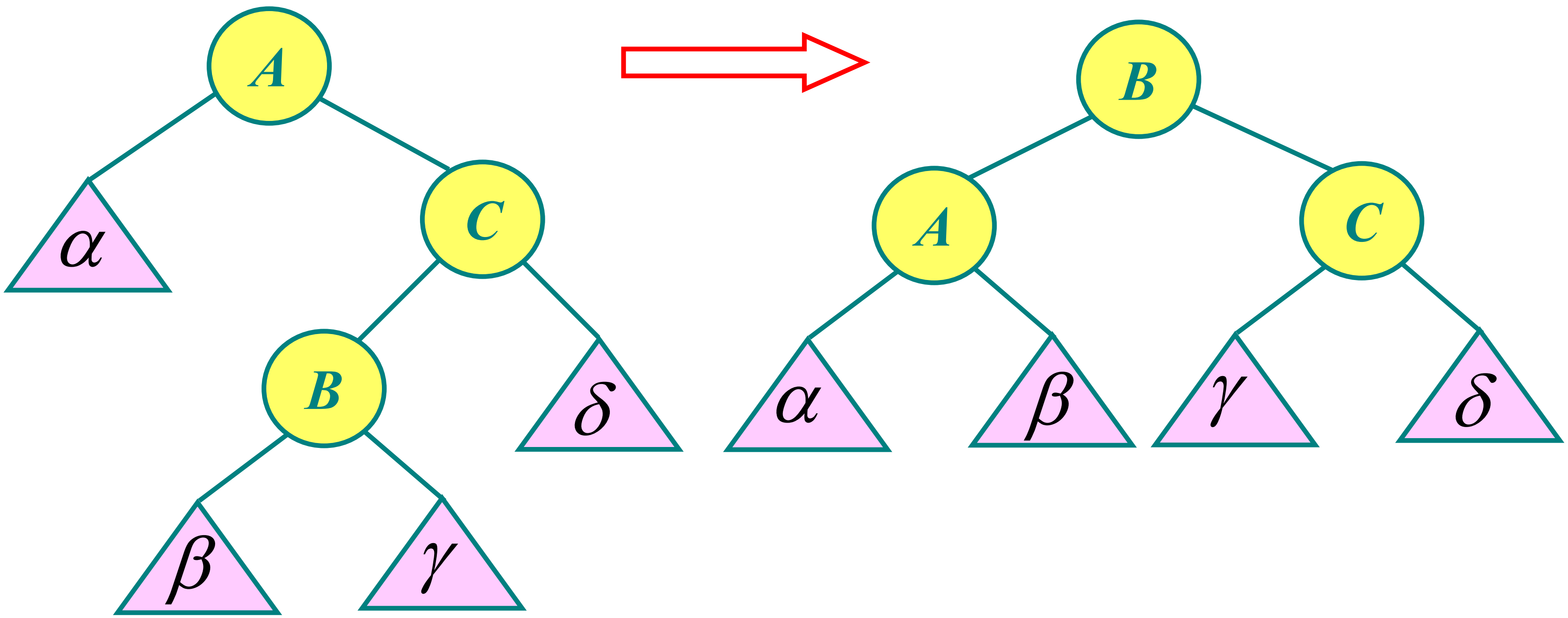
Right rotation

Double rotation



Left-right double rotation to fix case 2

Double rotation

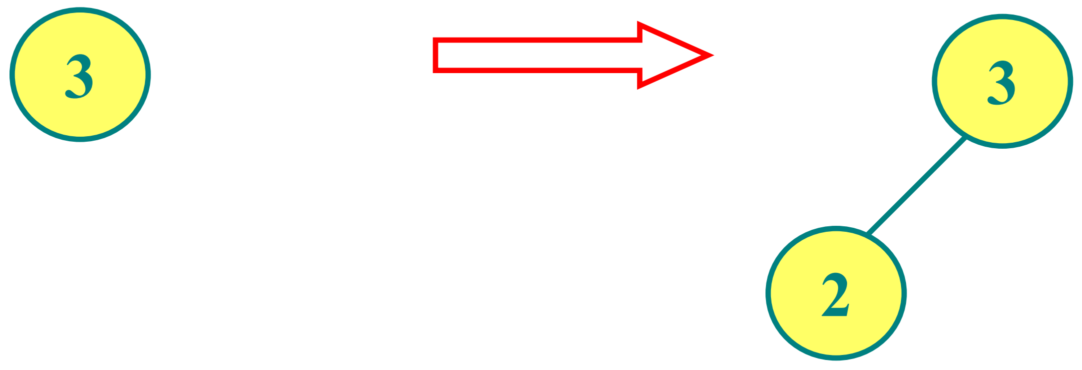


Right-left double rotation to fix case 3

AVL tree rotation

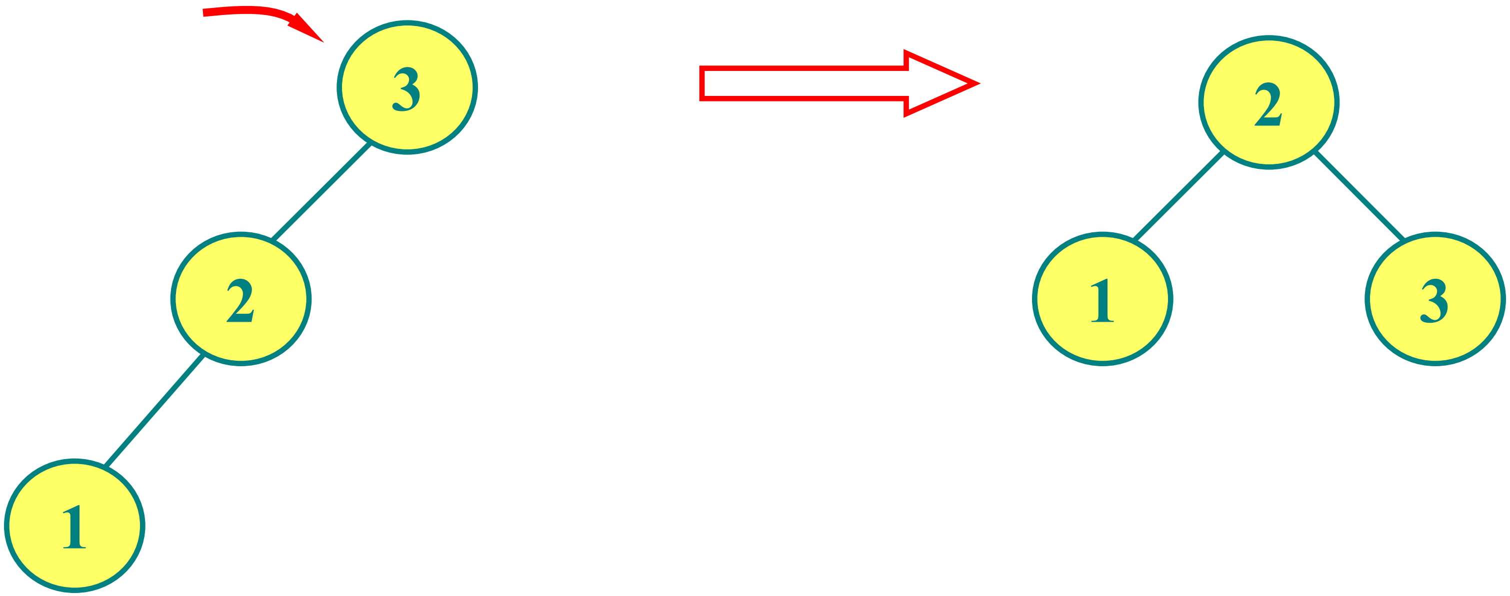
Four types	**Rotation**
***Case* 1:** *Left-left*	*Right rotation*
***Case* 4:** *Right-right*	*Left rotation*
***Case* 2:** *Left-right*	*Left-right double rotation*
***Case* 3:** *Right-left*	*Right-left double rotation*

AVL tree example



Insert 2

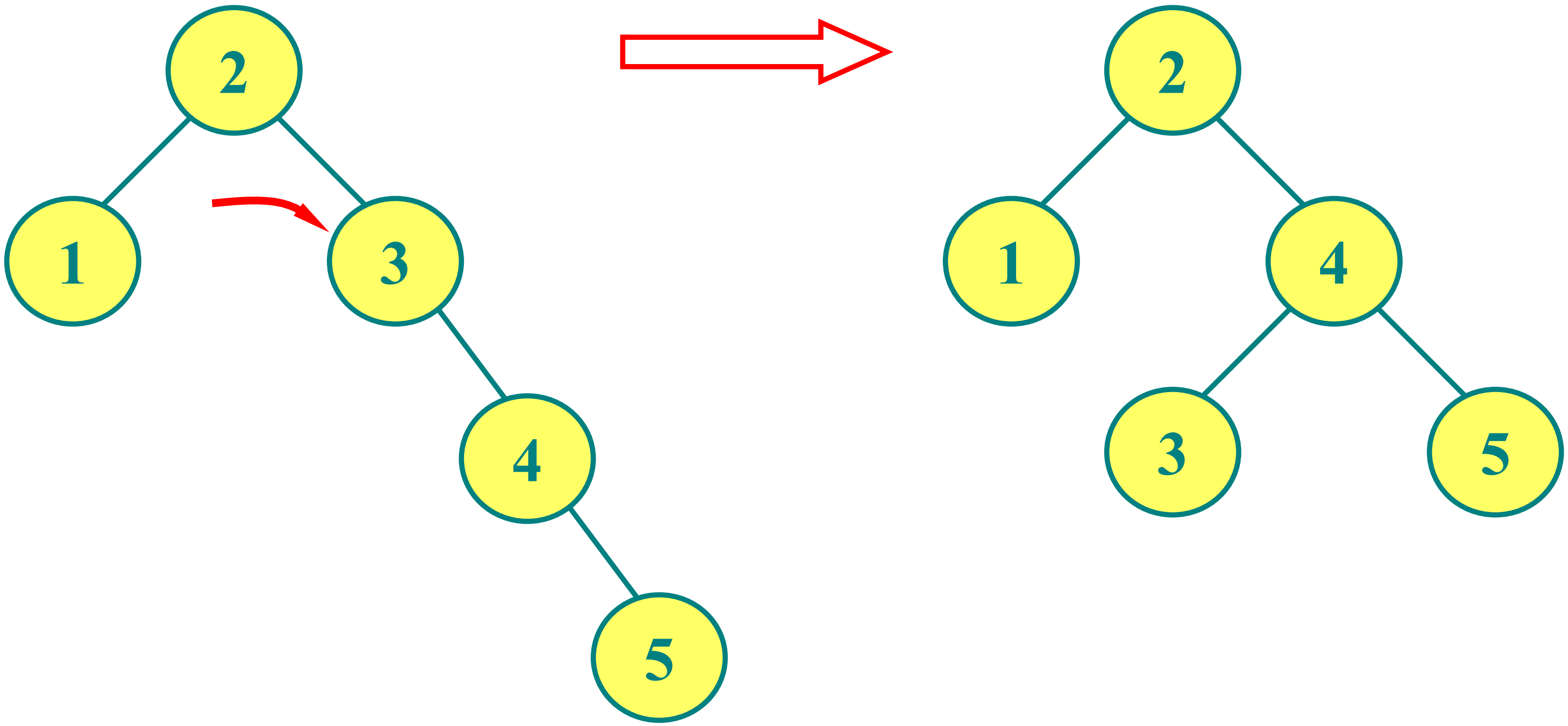
AVL tree example (cont.)



Insert 1

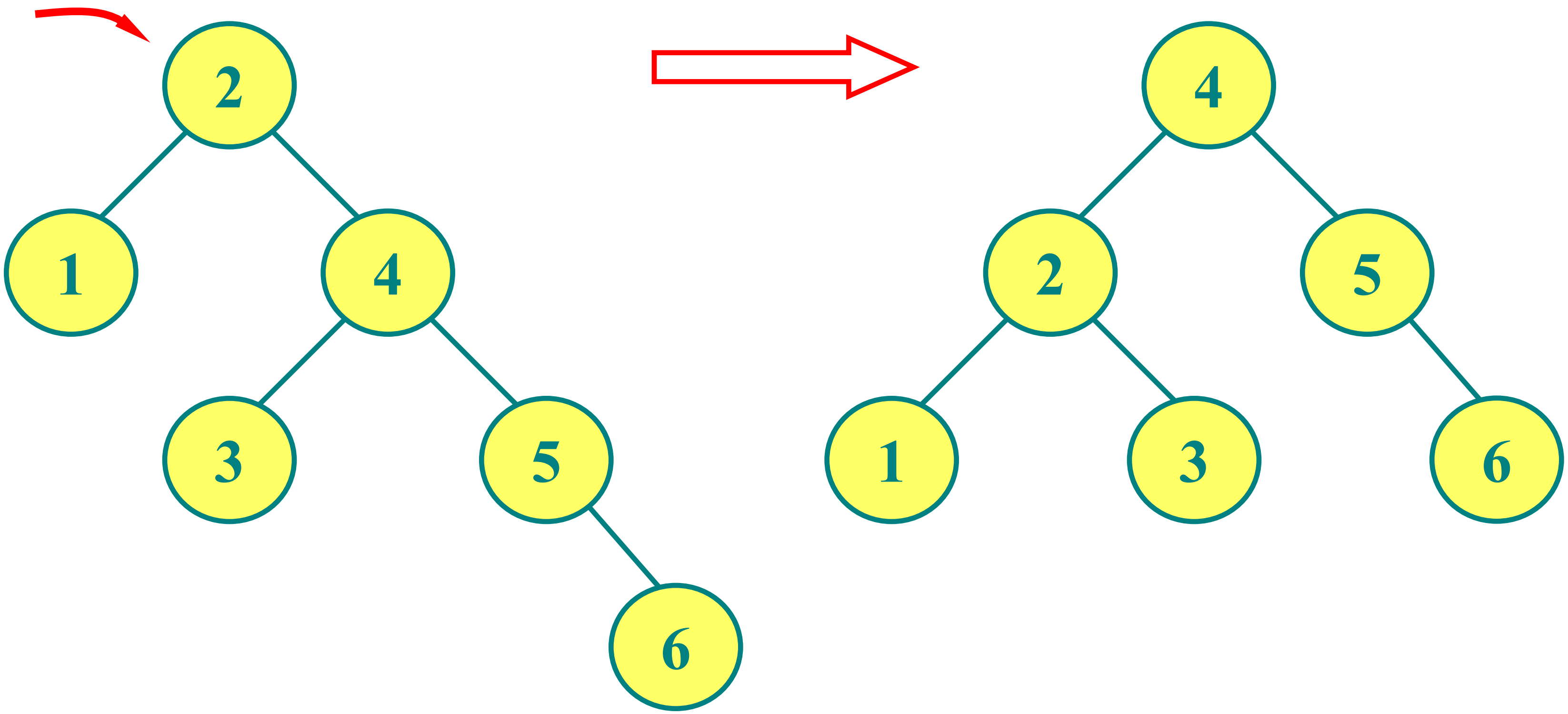
Right rotation

AVL tree example (cont.)



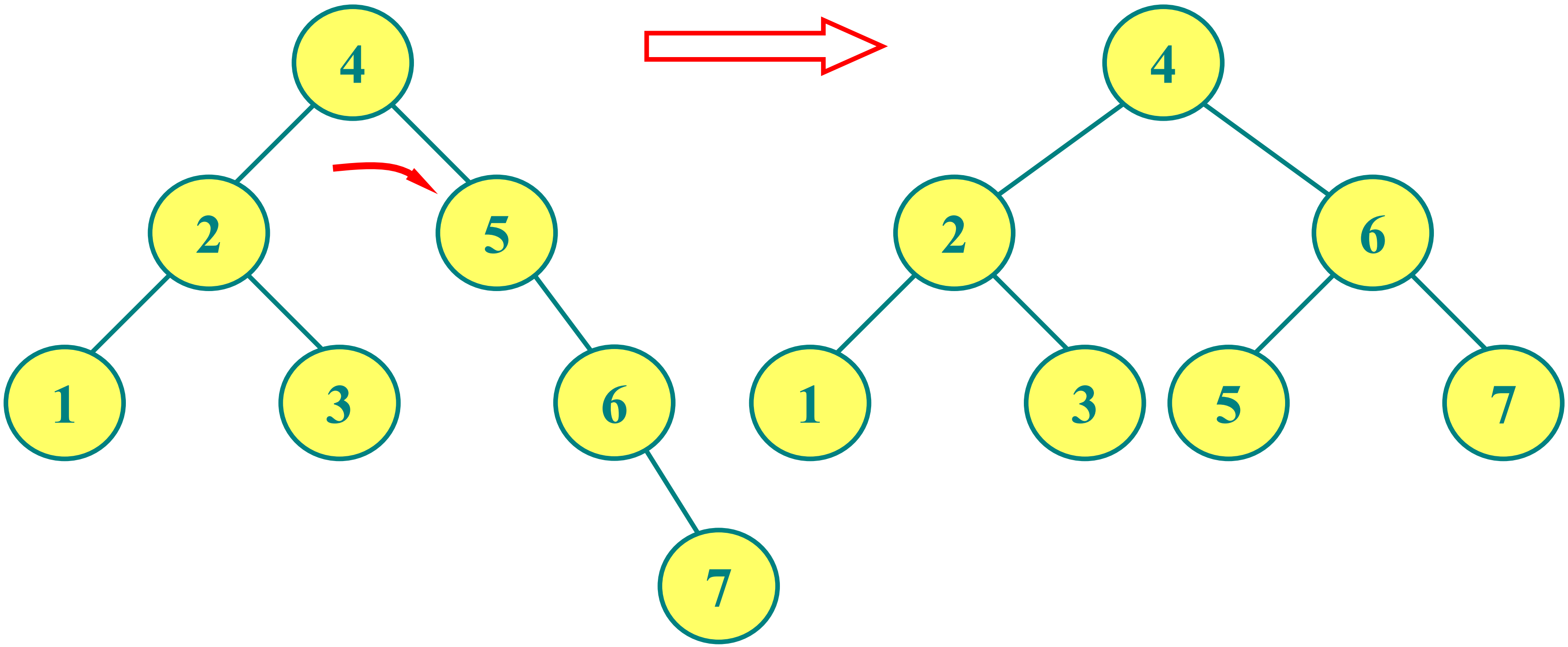
Insert 4 and 5 ***Left rotation***

AVL tree example (cont.)



Insert 6 ***Left rotation***

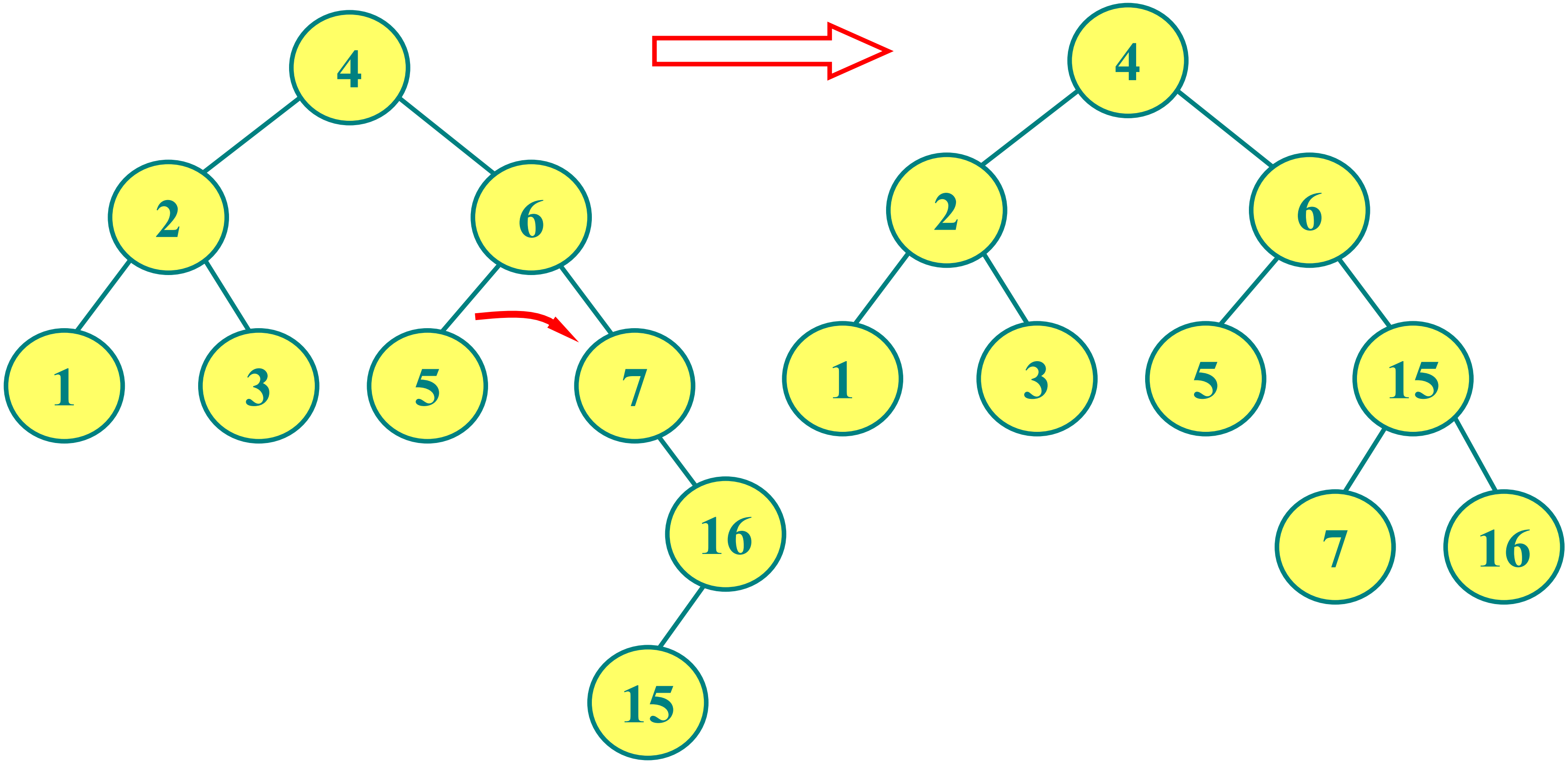
AVL tree example (cont.)



Insert 7

Left rotation

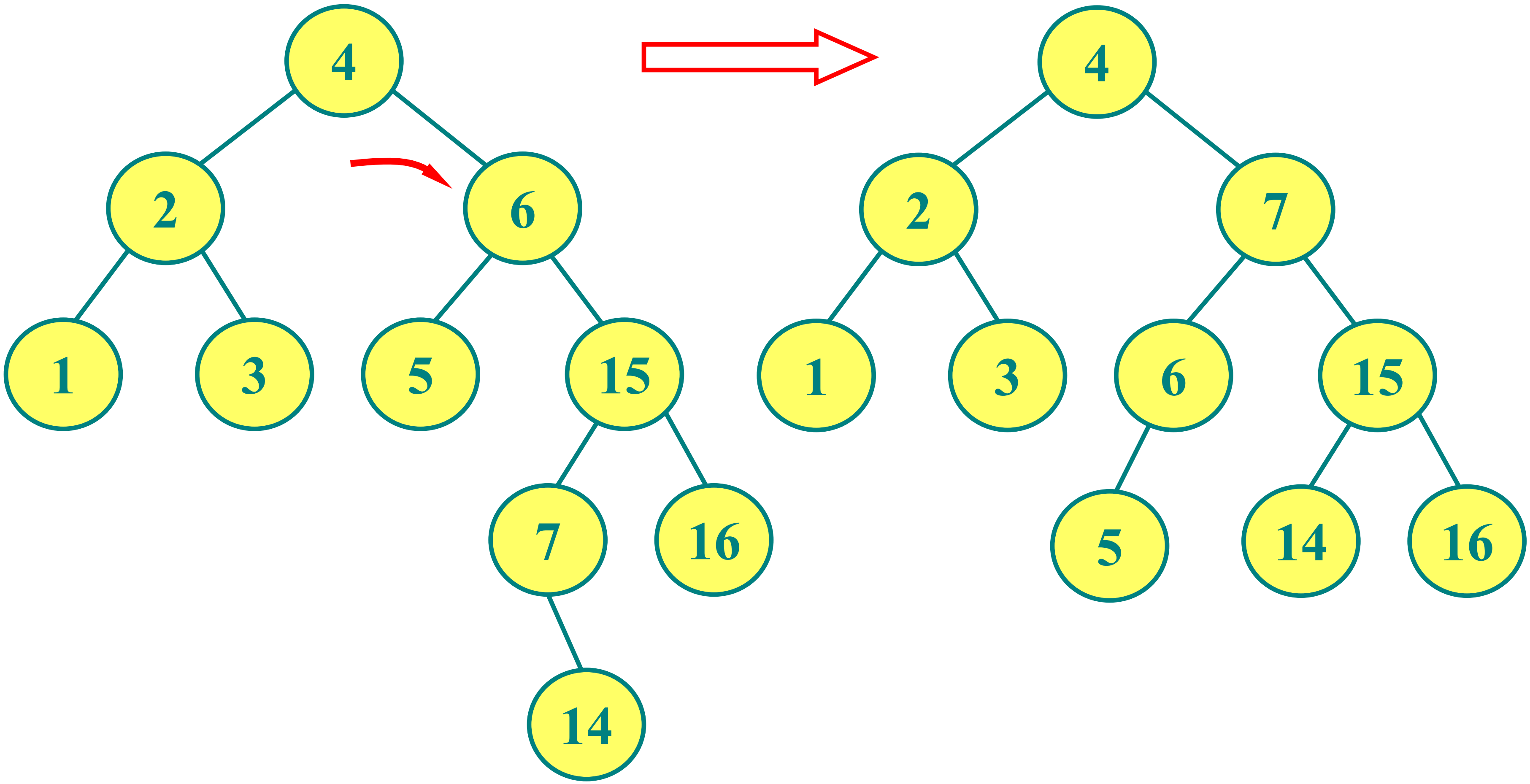
AVL tree example (cont.)



Insert 16 and 15

Right-left rotation

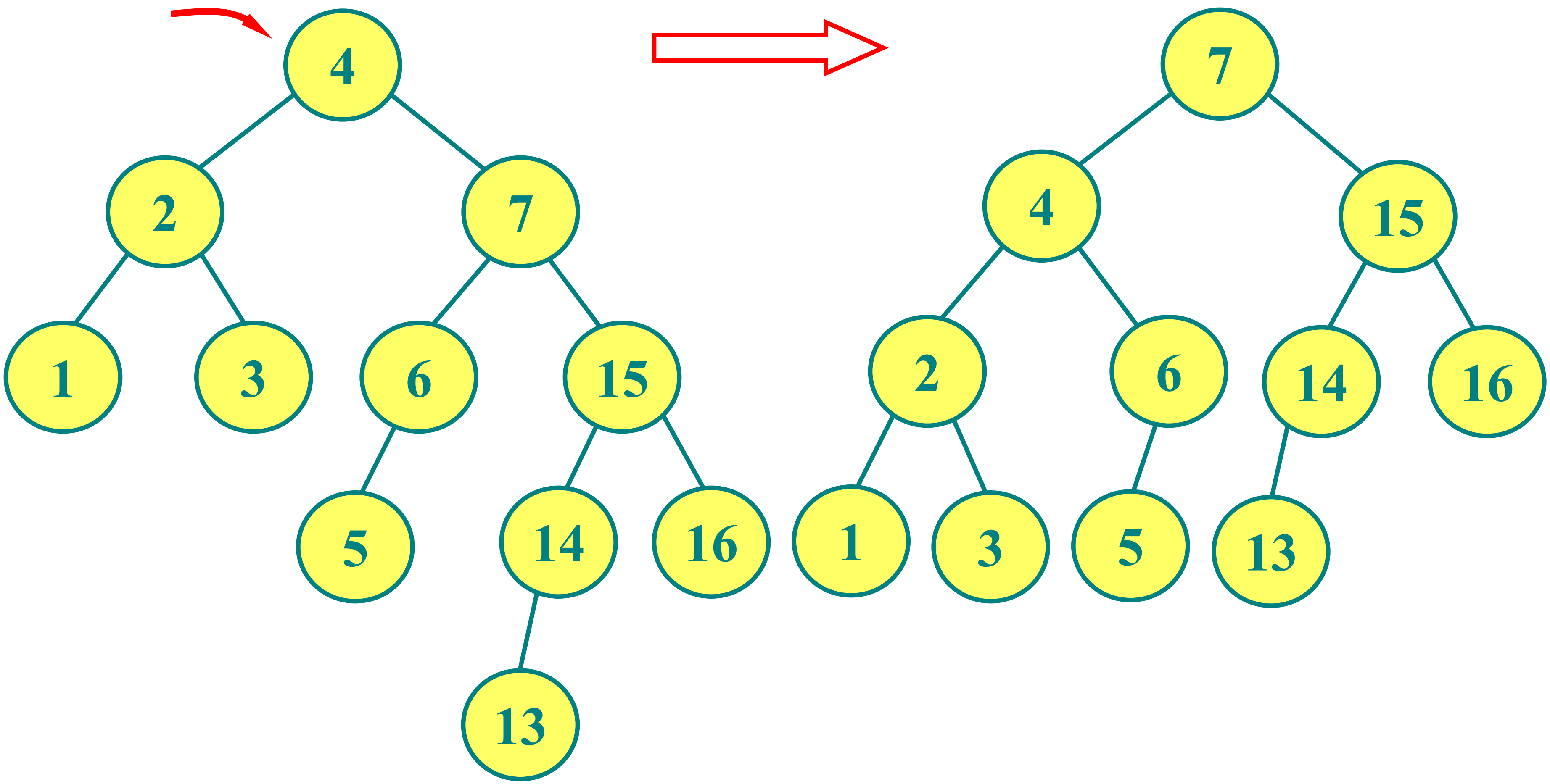
AVL tree example (cont.)



Insert 14

Right-left rotation

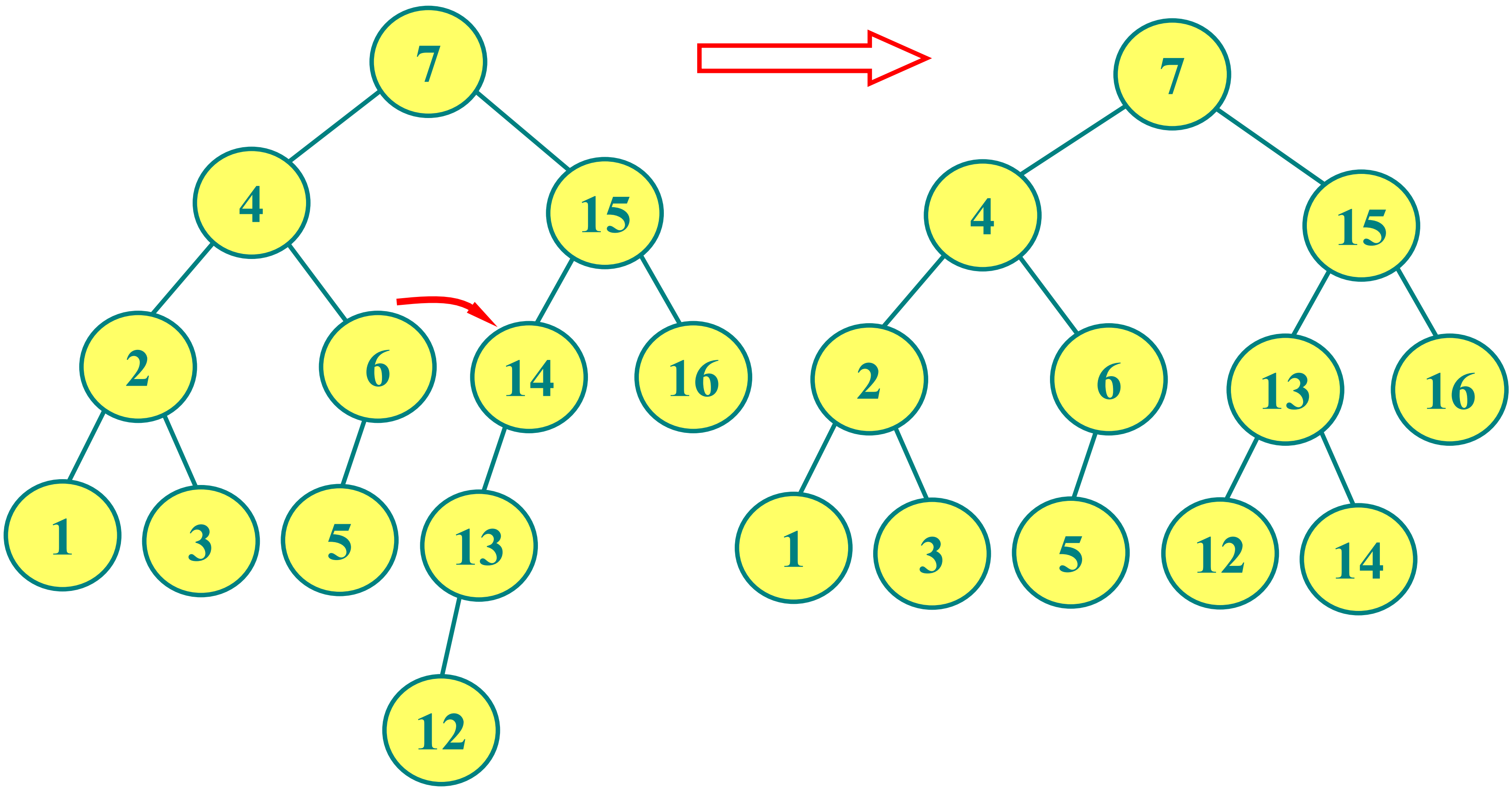
AVL tree example (cont.)



Insert 13

Left rotation

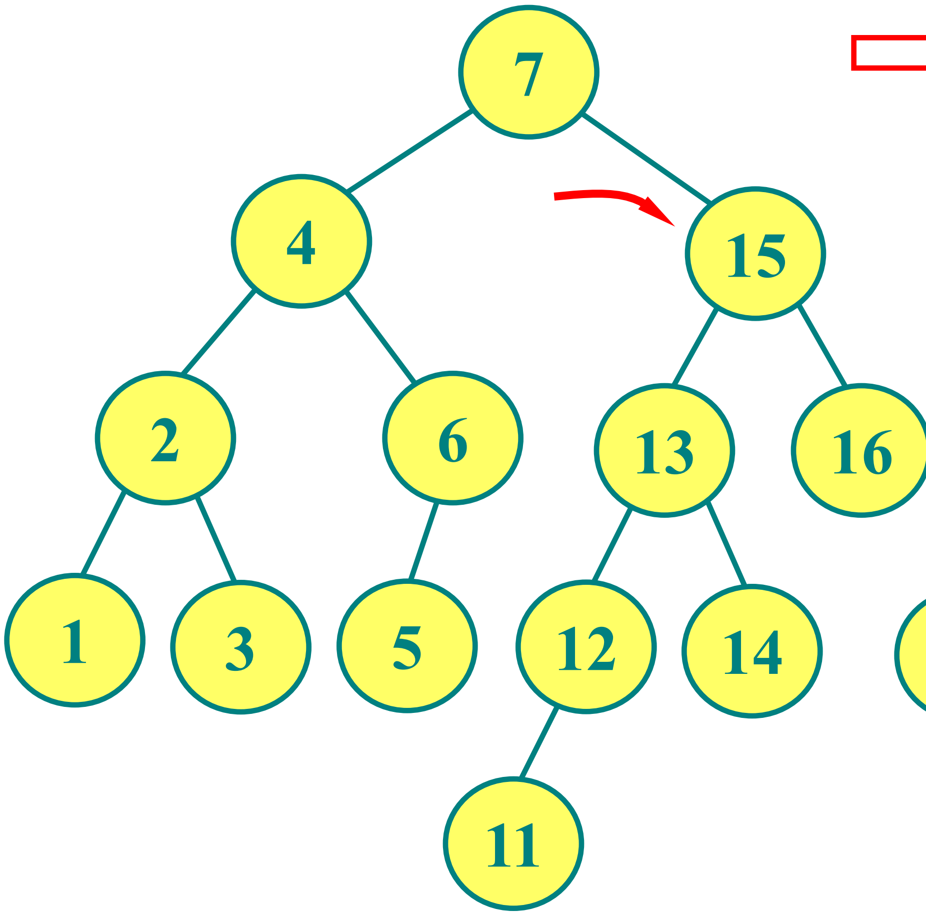
AVL tree example (cont.)



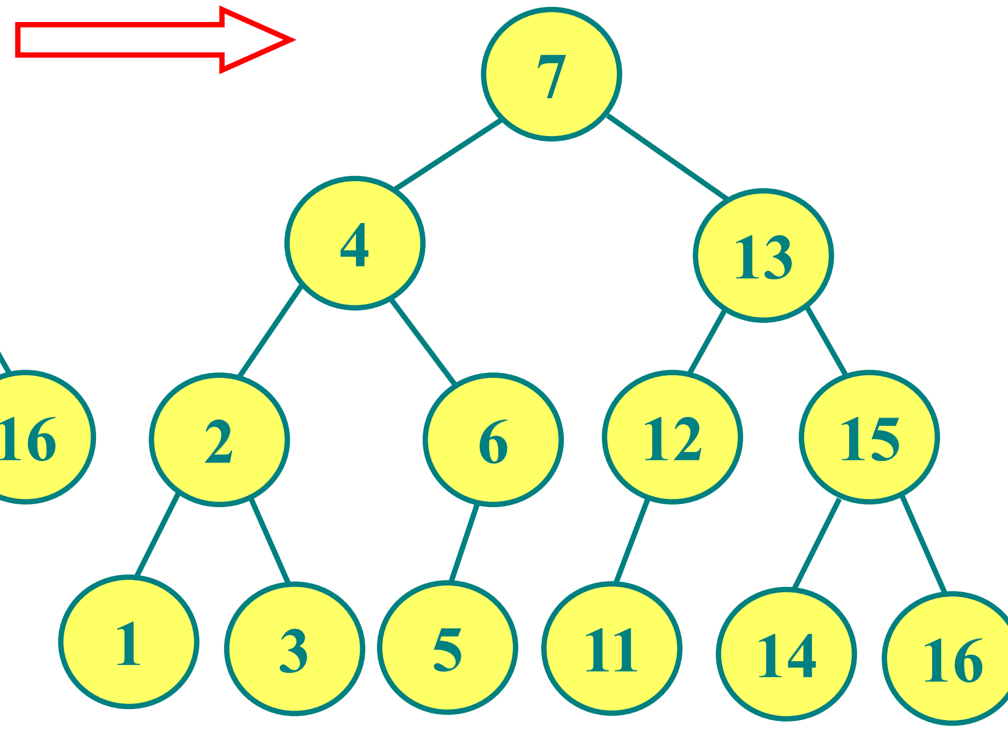
Insert 12

Right rotation

AVL tree example (cont.)

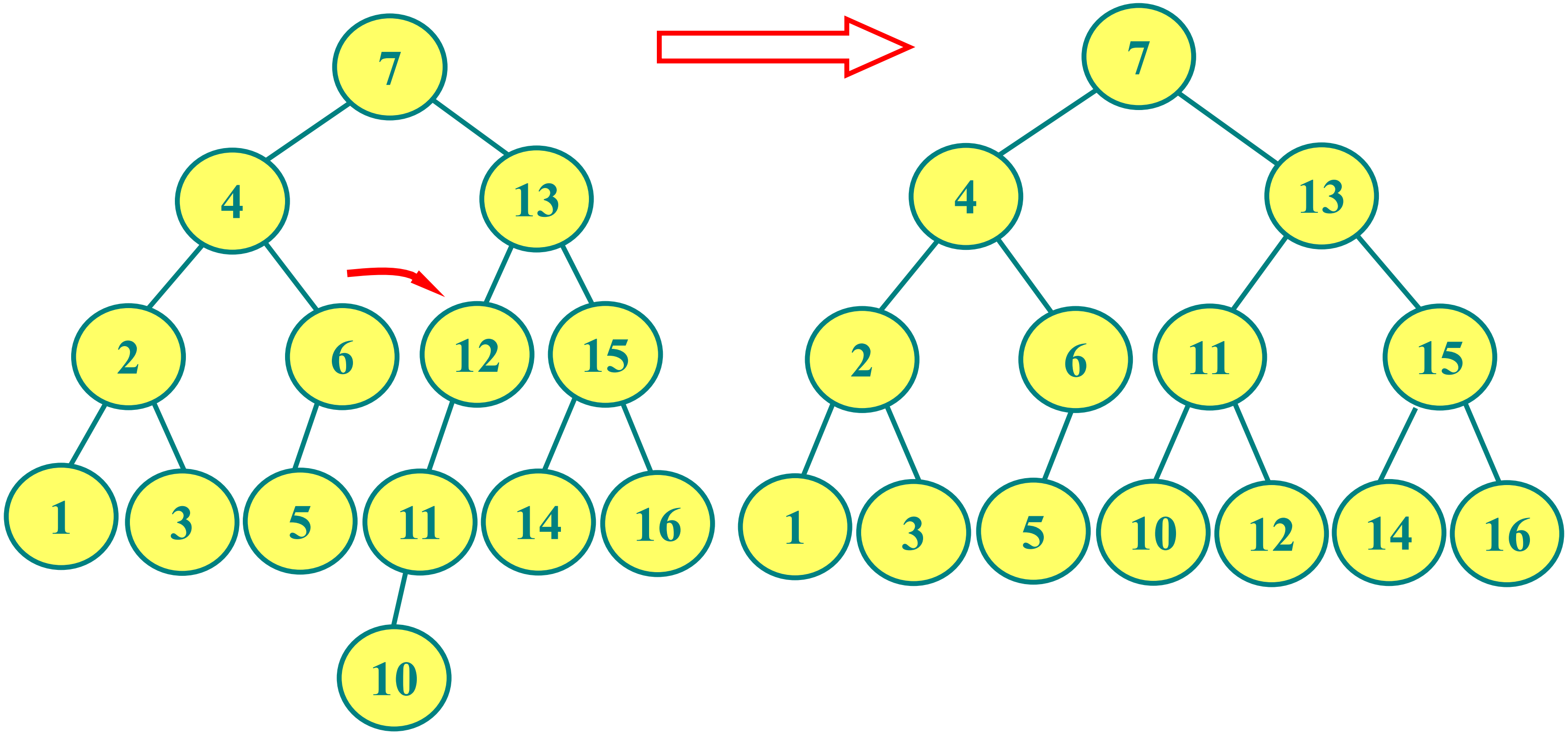


Insert 11



Right rotation

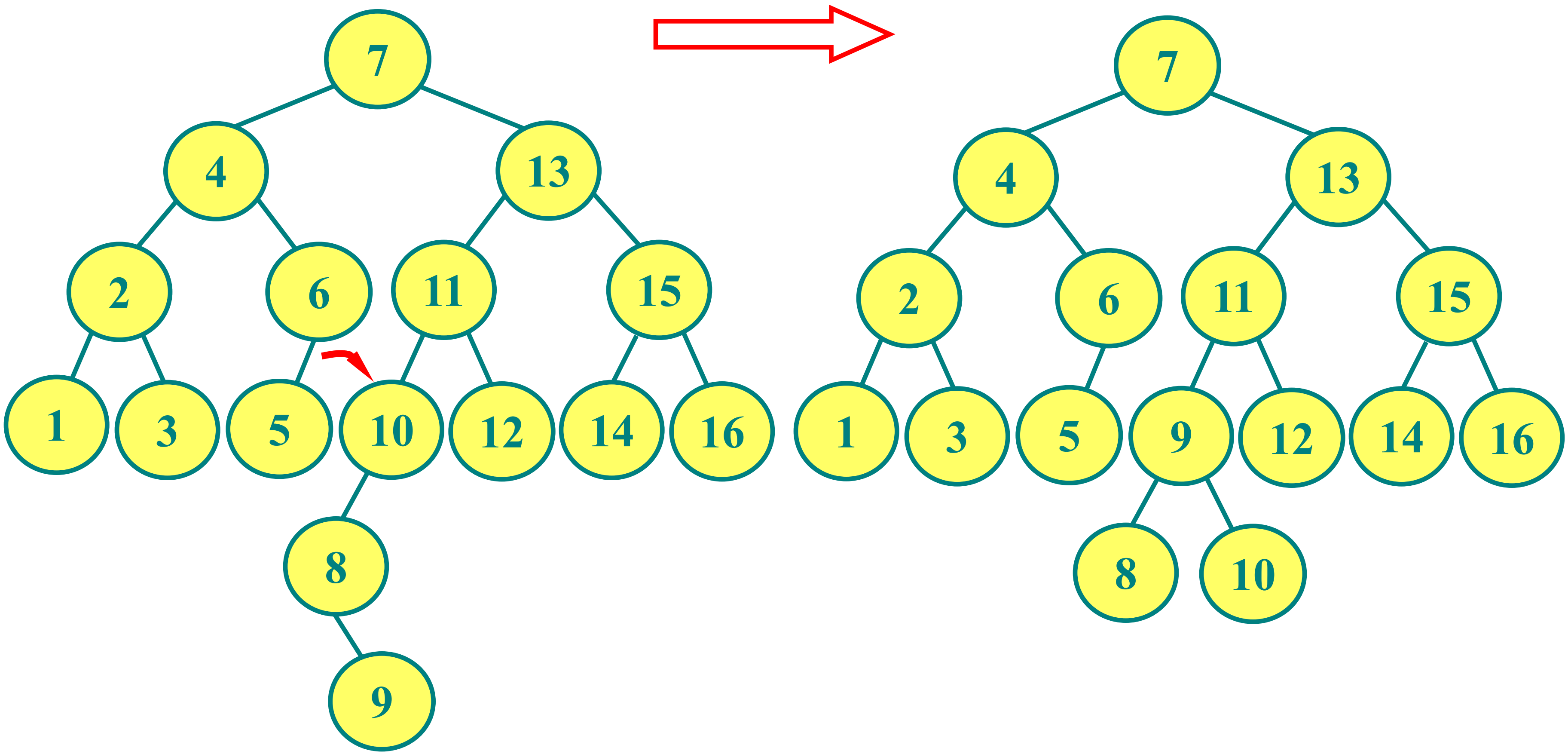
AVL tree example (cont.)



Insert 10

Right rotation

AVL tree example (cont.)



Insert 8 and 9

Left-right rotation

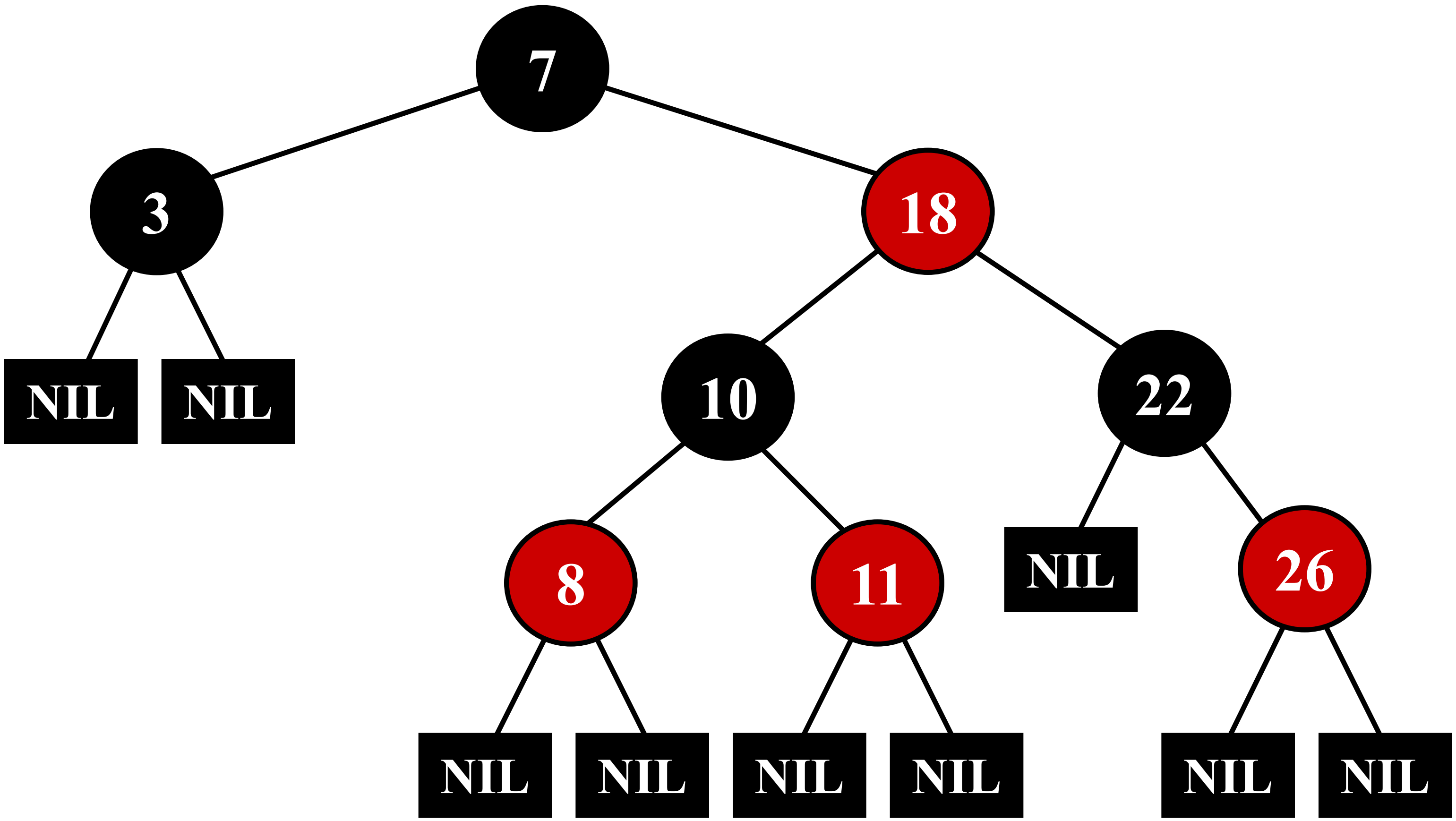
Red-black trees

BSTs with an extra one-bit color field in each node.

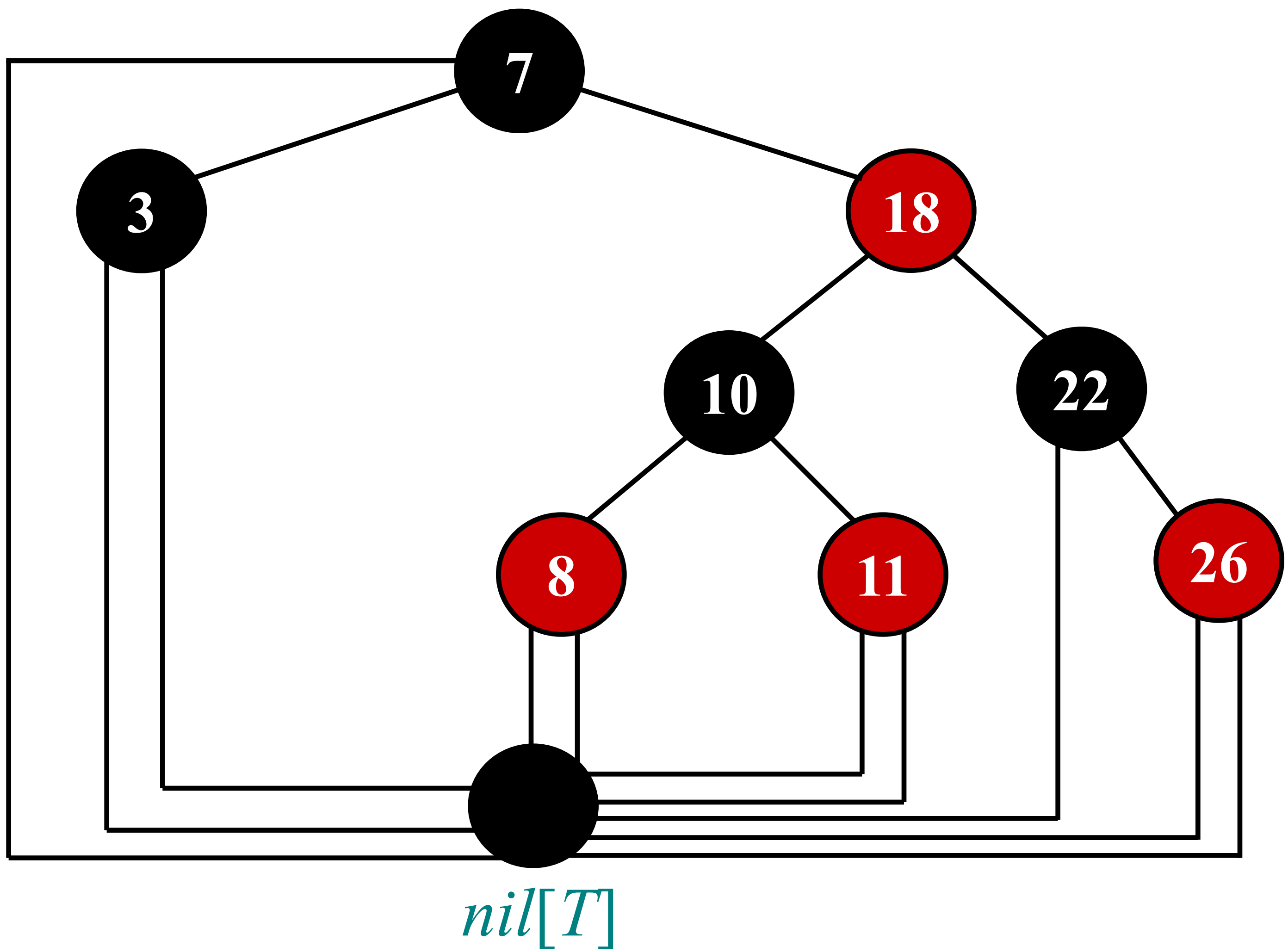
Red-black properties:

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. All simple paths from any node x to a descendant leaf have the same number of **black** nodes.

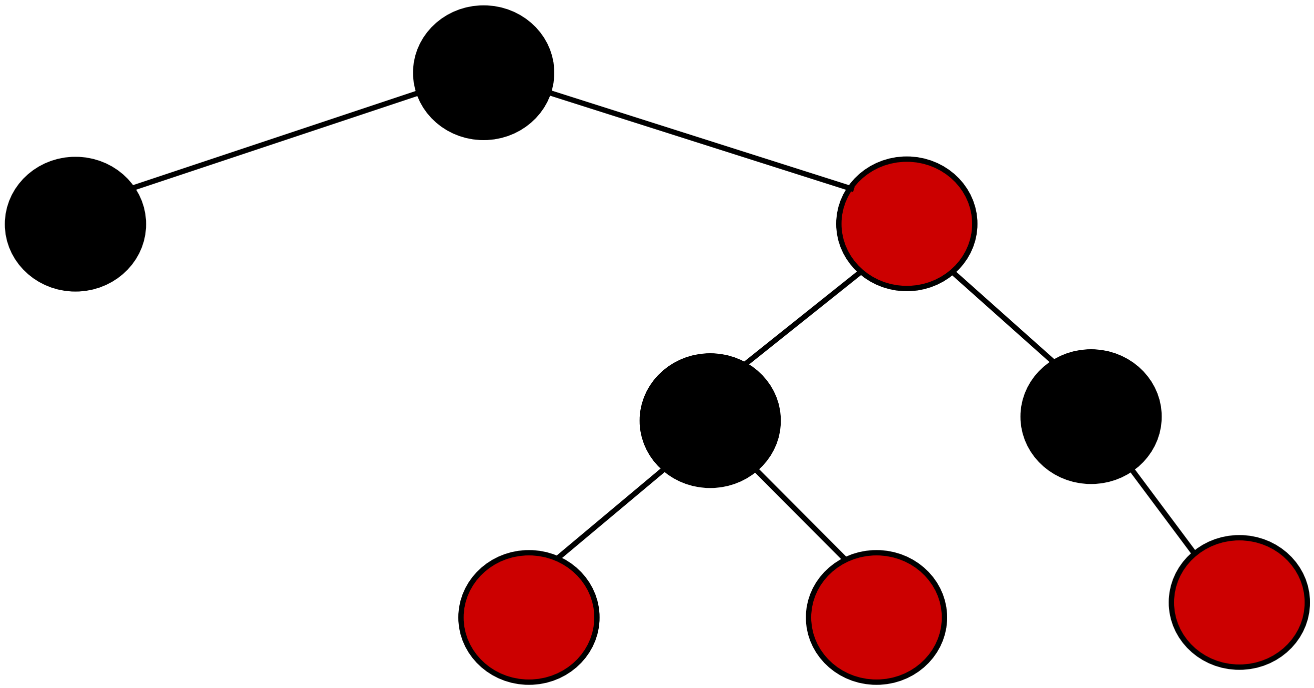
Example of a red-black tree



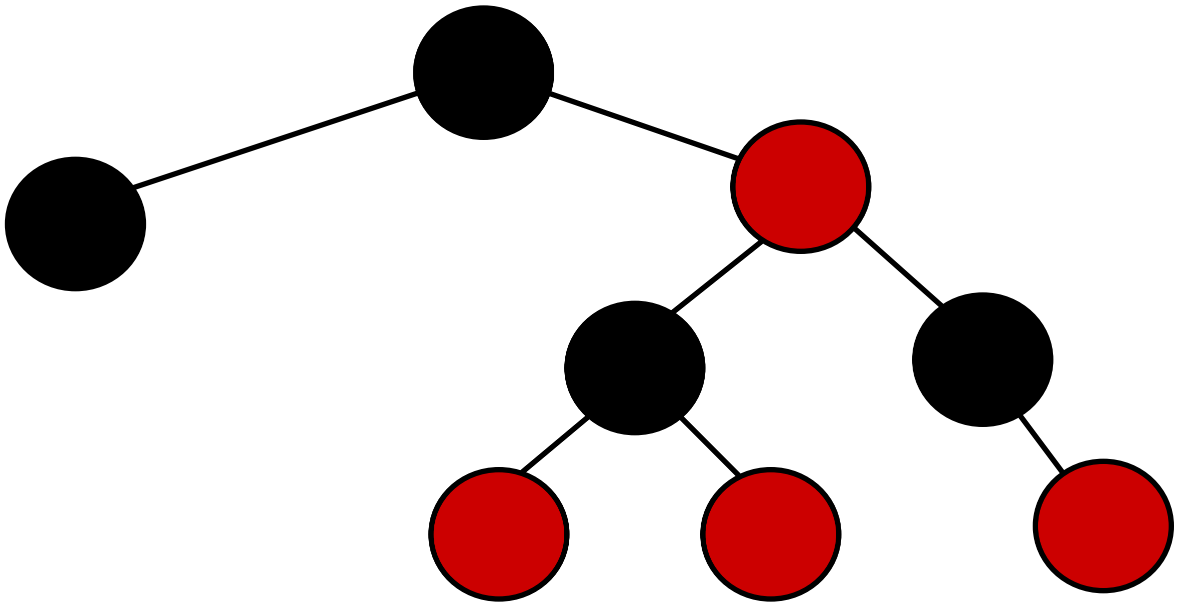
Example of a red-black tree



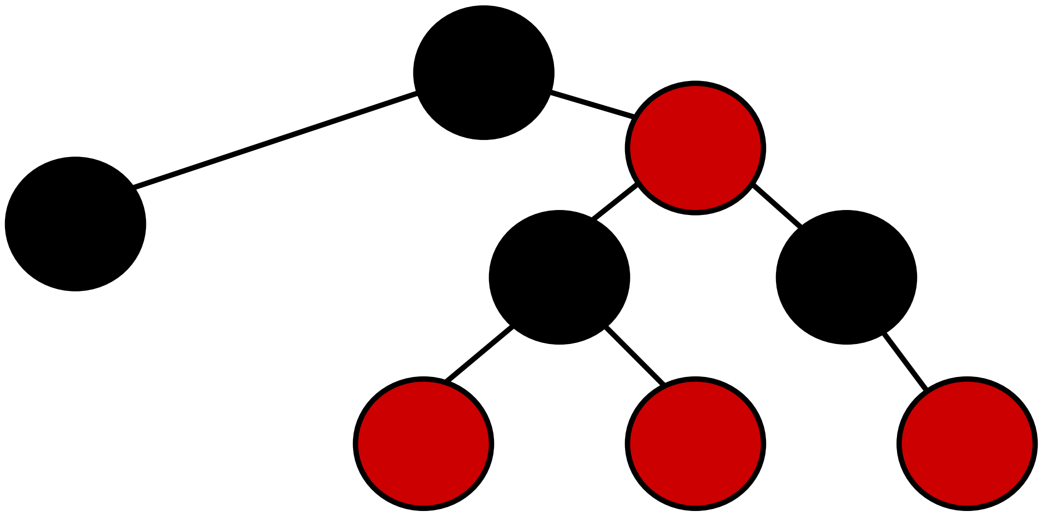
Height of a red-black tree



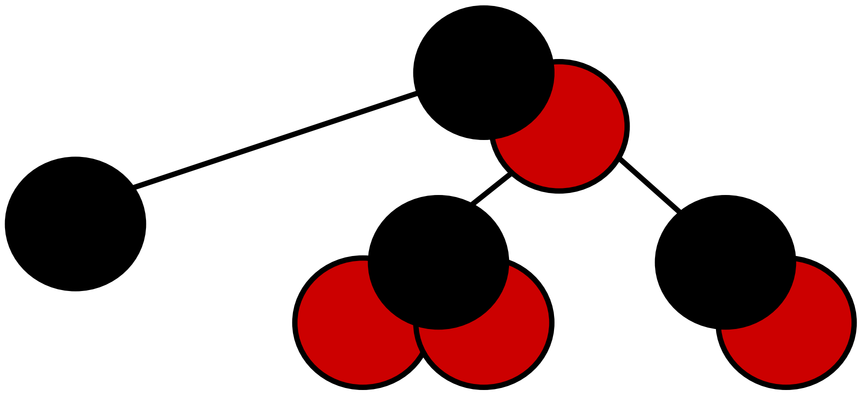
Height of a red-black tree



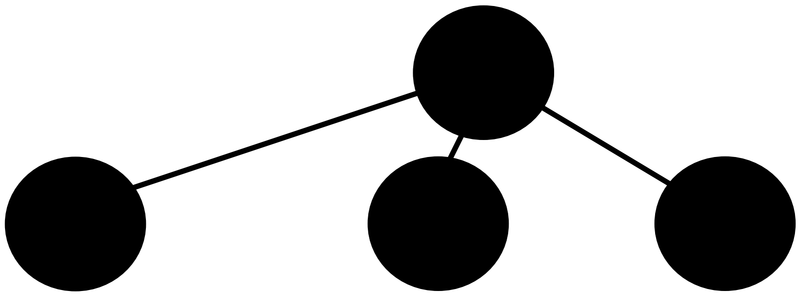
Height of a red-black tree



Height of a red-black tree



Height of a red-black tree



Lemma of red-black tree

We call the number of black nodes on any path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted $bh(x)$.

Lemma.

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

Dynamic-set operations **search**, **minimum**, **maximum**, **successor**, and **predecessor** can be implemented in $O(lgn)$ time on red-black trees.

Proof

Subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

- **Base case:**
  Height of x is 0, then x must be a leaf ($nil[T]$), subtree rooted at x contains at least
  $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

- **Inductive:**
  Height of a child of x is less than the height of x itself, 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 (cont.)

According to ***property* 4**, at least the half nodes on any simple path from the root to a leaf, not including the root, must be black.

Consequently, the black-height of the root must be at least $h/2$; thus,

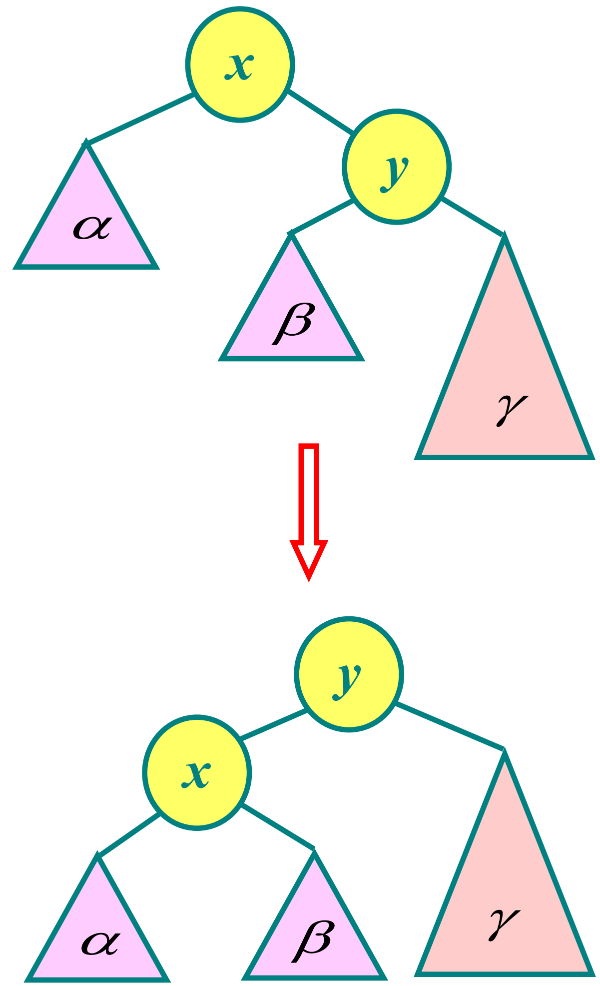
$$n \geq 2^{h/2-1} - 1. \quad \Rightarrow$$

$$h \leq 2lg(n+1). \quad \square$$

Left rotation

LEFT-ROTATE(T, x)

```
y ← right[x]
right[x] ← left[y]
p[left[y]] ← x
p[y] ← p[x]
if p[x] = nil[T]
   then root[T] ← y
   else if x = left[p[x]]
           then left[p[x]] ← y
           else  right[p[x]] ← y
left[y] ← x
p[x] ← y
```



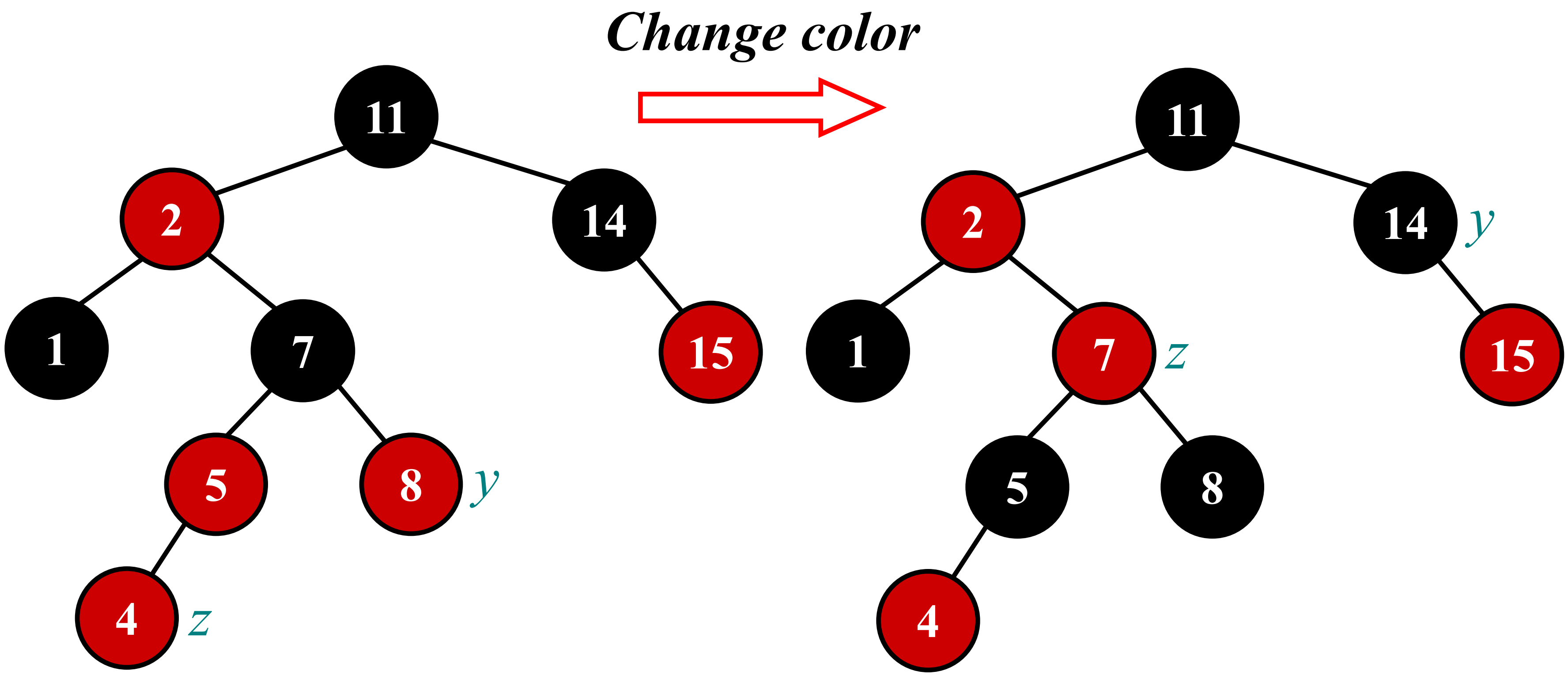
RB-Insertion

```
RB-INSERT(T, z)
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = nil[T]
    then root[T] ← z
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```

RB-Insertion

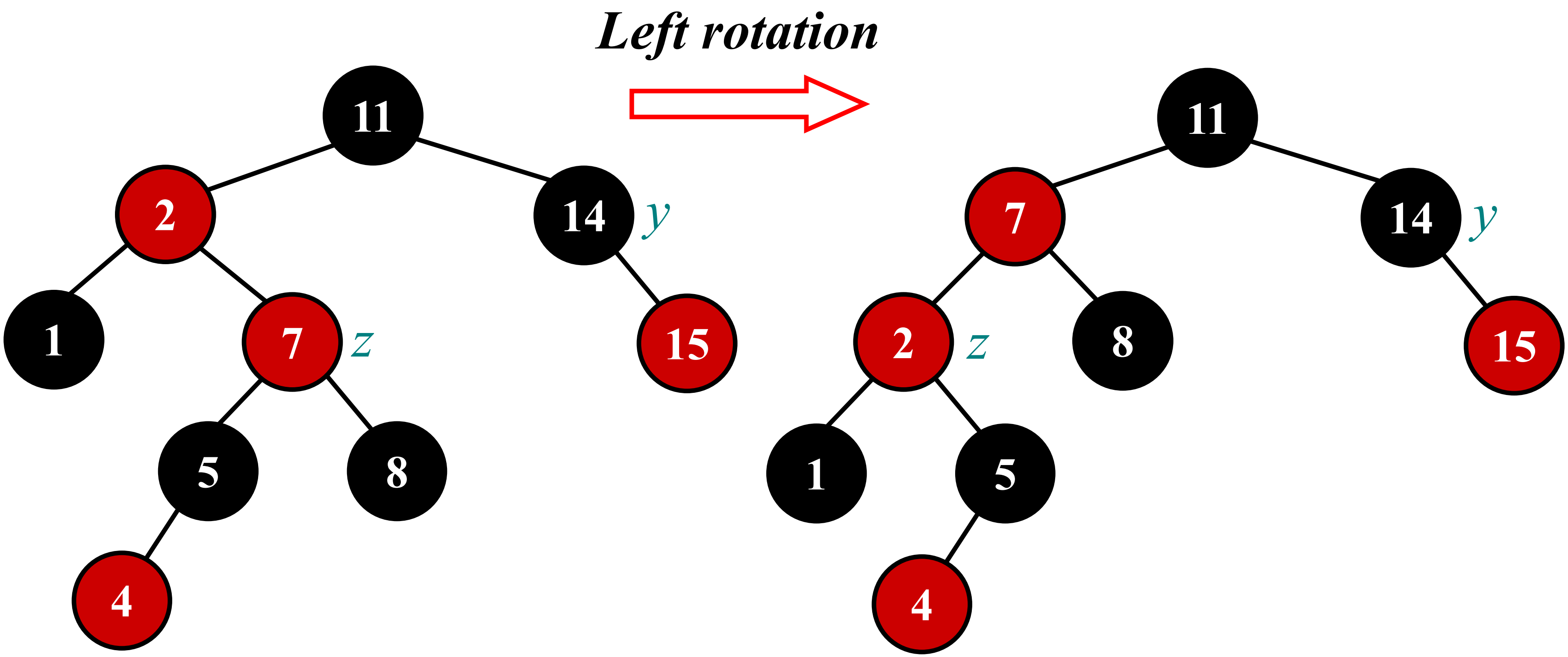
Which of the red-black properties can be violated upon the call to RB-INSERT-FIXUP?

RB-Insertion (case 1)



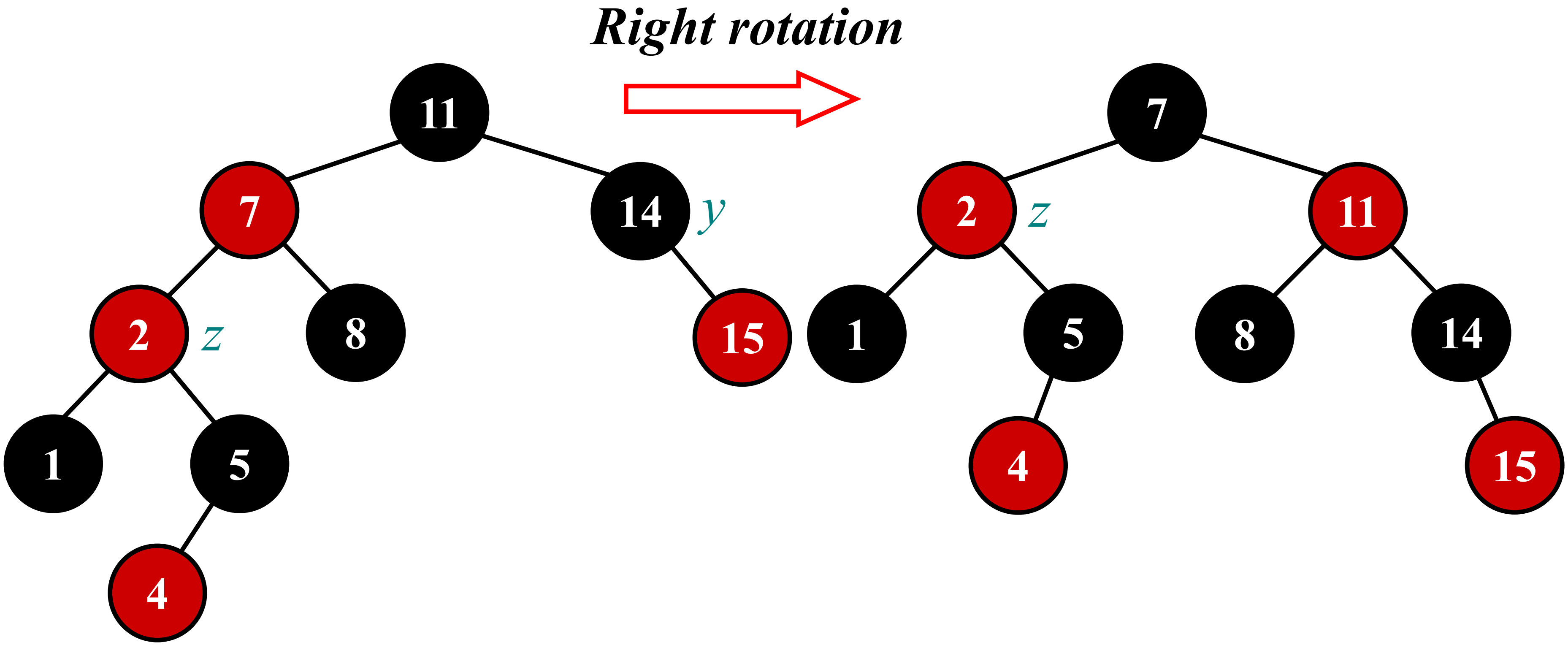
Case 1: *z's uncle y is red*

RB-Insertion (case 2)



Case 2: ***z*'s uncle *y* is black and *z* is a right child.**
***Convert case 2 to* 3.**

RB-Insertion (case 3)



Case 3: ***z*'s uncle *y* is black and *z* is a left child**

RB-tree insertion

Types		Operation
z's father is ***left*** *child*	***Case 1L:*** *z's uncle is red.*	*Change color.*
	Case 2L: *z's uncle is black and z is* ***right*** *child.*	***Left*** *rotation, $p(z)$.*
	Case 3L: *z's uncle is black and z is* ***left*** *child.*	***Right*** *rotation, $p(p(z))$.*
z's father is ***right*** *child*	***Case 1R:*** *z's uncle is red.*	*Change color.*
	Case 2R: *z's uncle is black and z is* ***left*** *child.*	***Right*** *rotation, $p(z)$.*
	Case 3R: *z's uncle is black and z is* ***right*** *child.*	***Left*** *rotation, $p(p(z))$.*

RB-Insertion

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
          then y ← right[p[p[z]]]
               if color[y] = RED
                  then color[p[z]] ← BLACK             Case 1
                       color[y] ← BLACK                Case 1
                       color[p[p[z]]] ← RED            Case 1
                       z ← p[p[z]]                     Case 1
```

RB-Insertion

else if $z = right[p[z]]$

  then $z \leftarrow p[z]$ — **Case 2**

   LEFT-ROTATION(T, z) — **Case 2**

  $color[p[z]] \leftarrow BLACK$ — **Case 3**

  $color[p[p[z]]] \leftarrow RED$ — **Case 3**

  RIGHT-ROTATION$(T, p[p[z]])$ — **Case 3**

else (same as **then** clause
  with "right" and "left" exchanged)

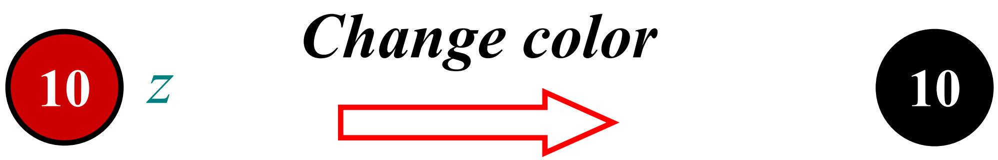
$color[root[T]] \leftarrow BLACK$

Running time:

$O(lgn)$

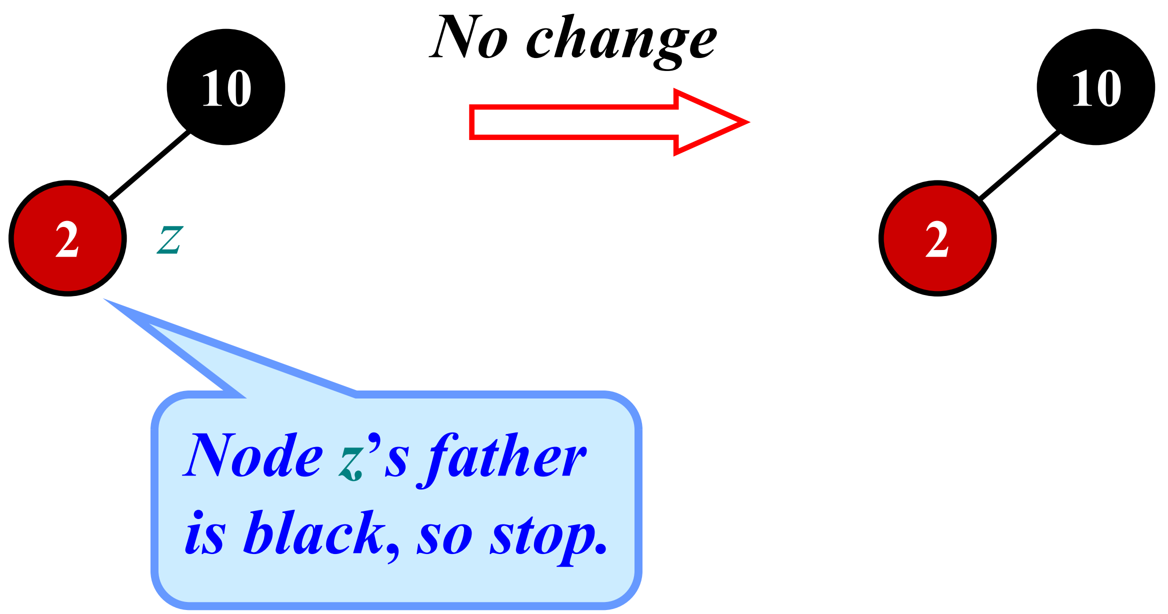
RB-Example

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



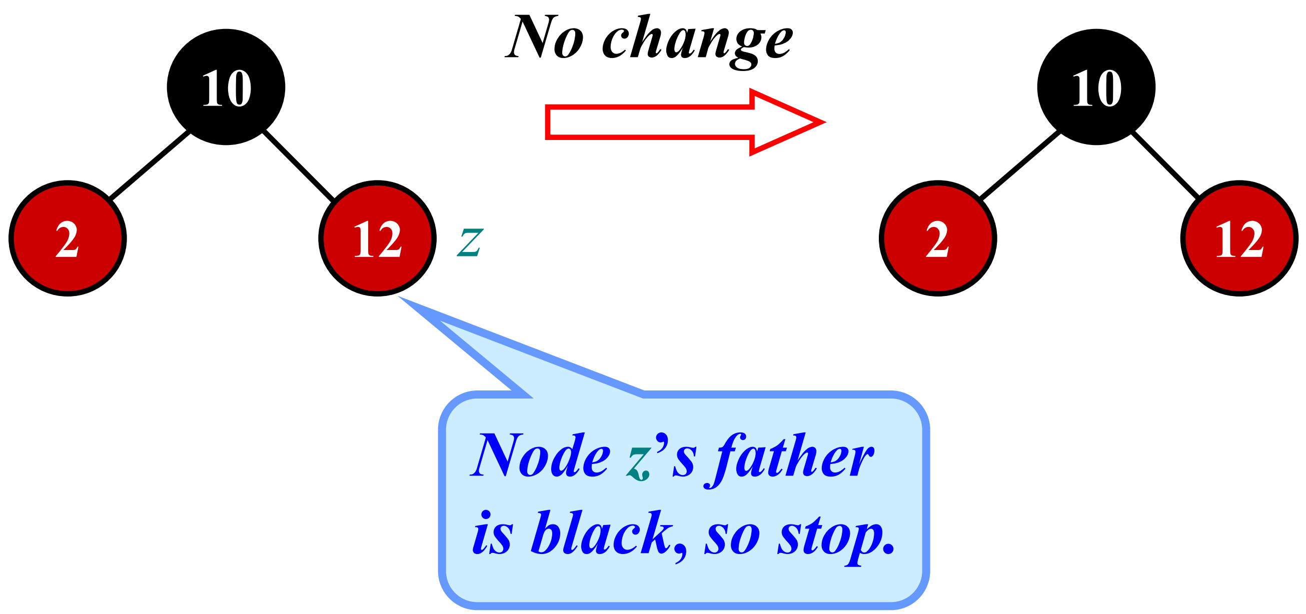
RB-Example (cont.)

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



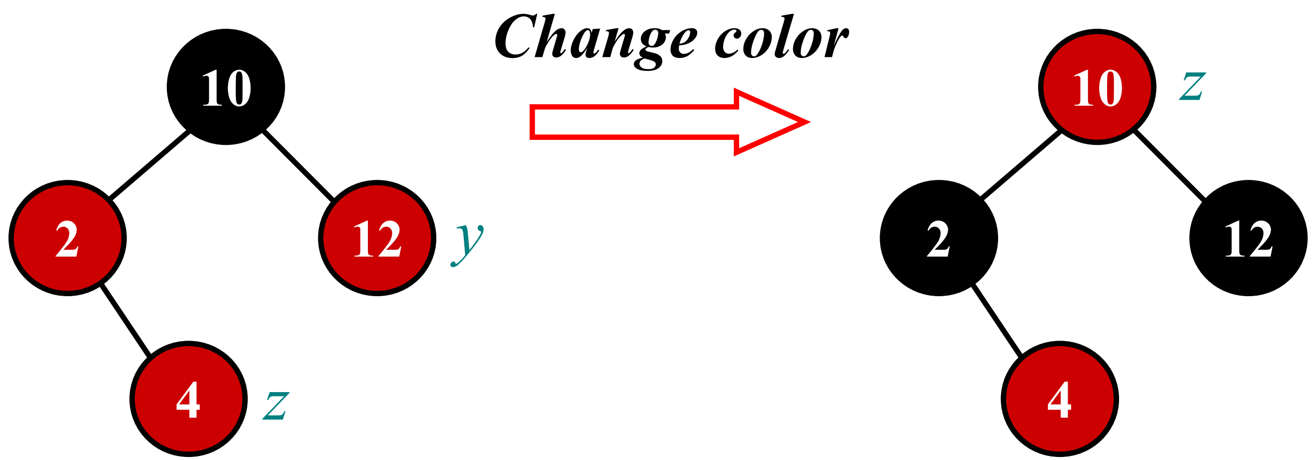
RB-Example (cont.)

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



RB-Example (cont.)

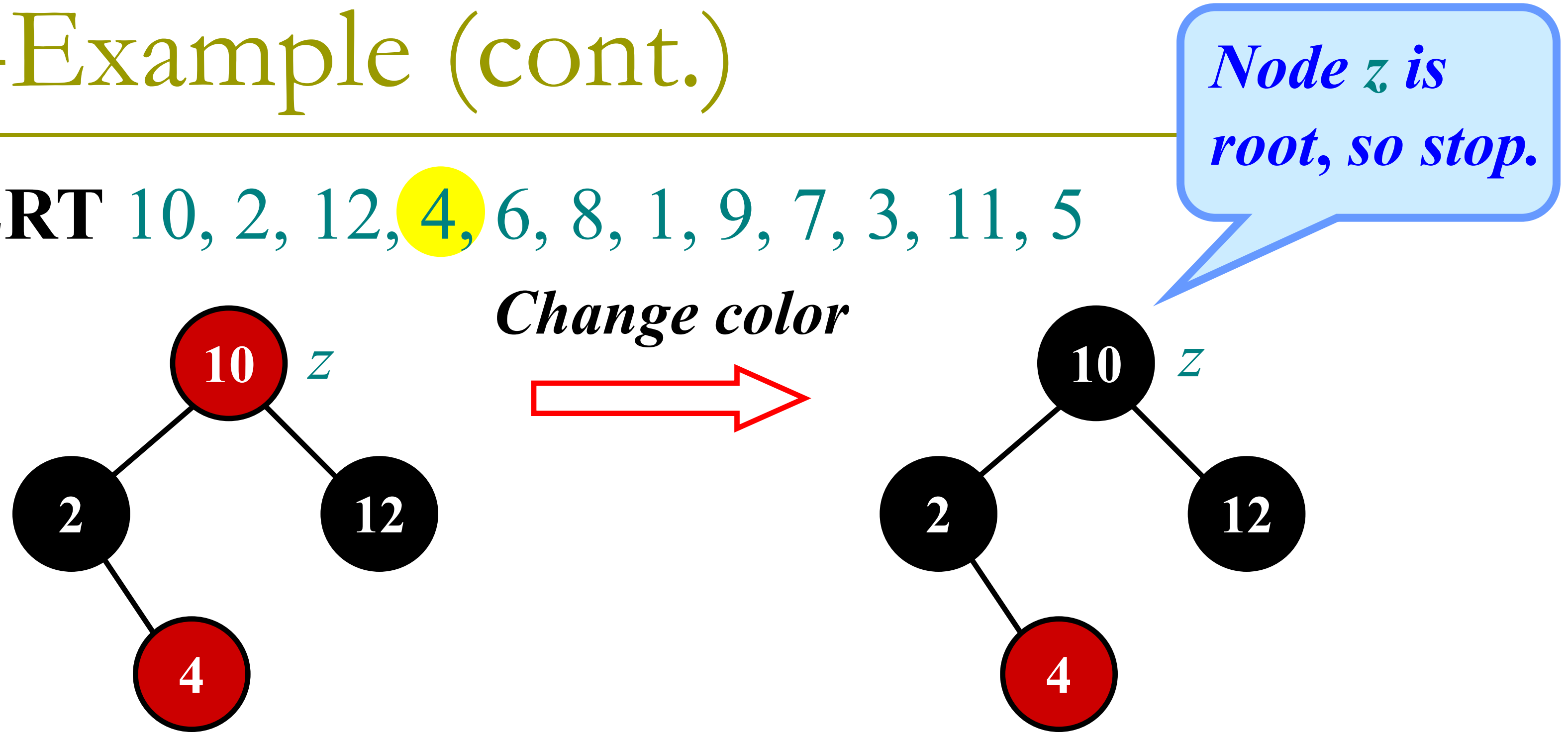
INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



Case 1L: ***z*'s uncle *y* is red and we get new *z*.**

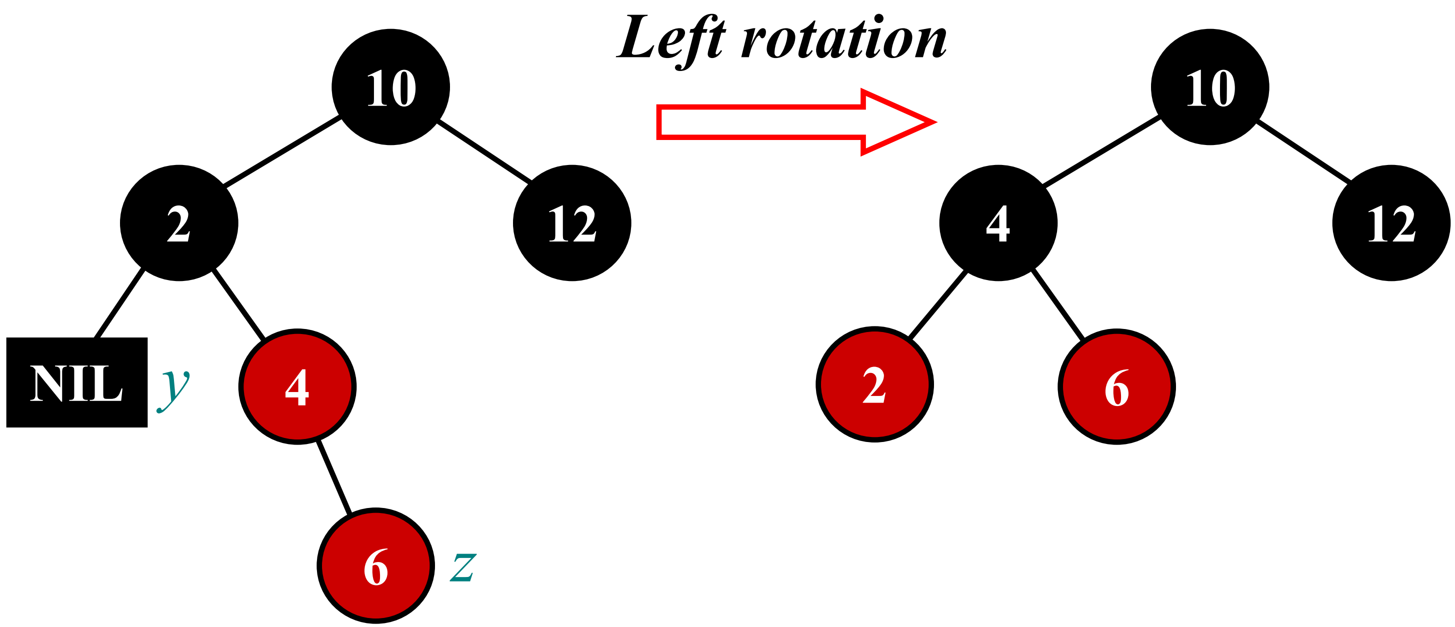
RB-Example (cont.)

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



RB-Example (cont.)

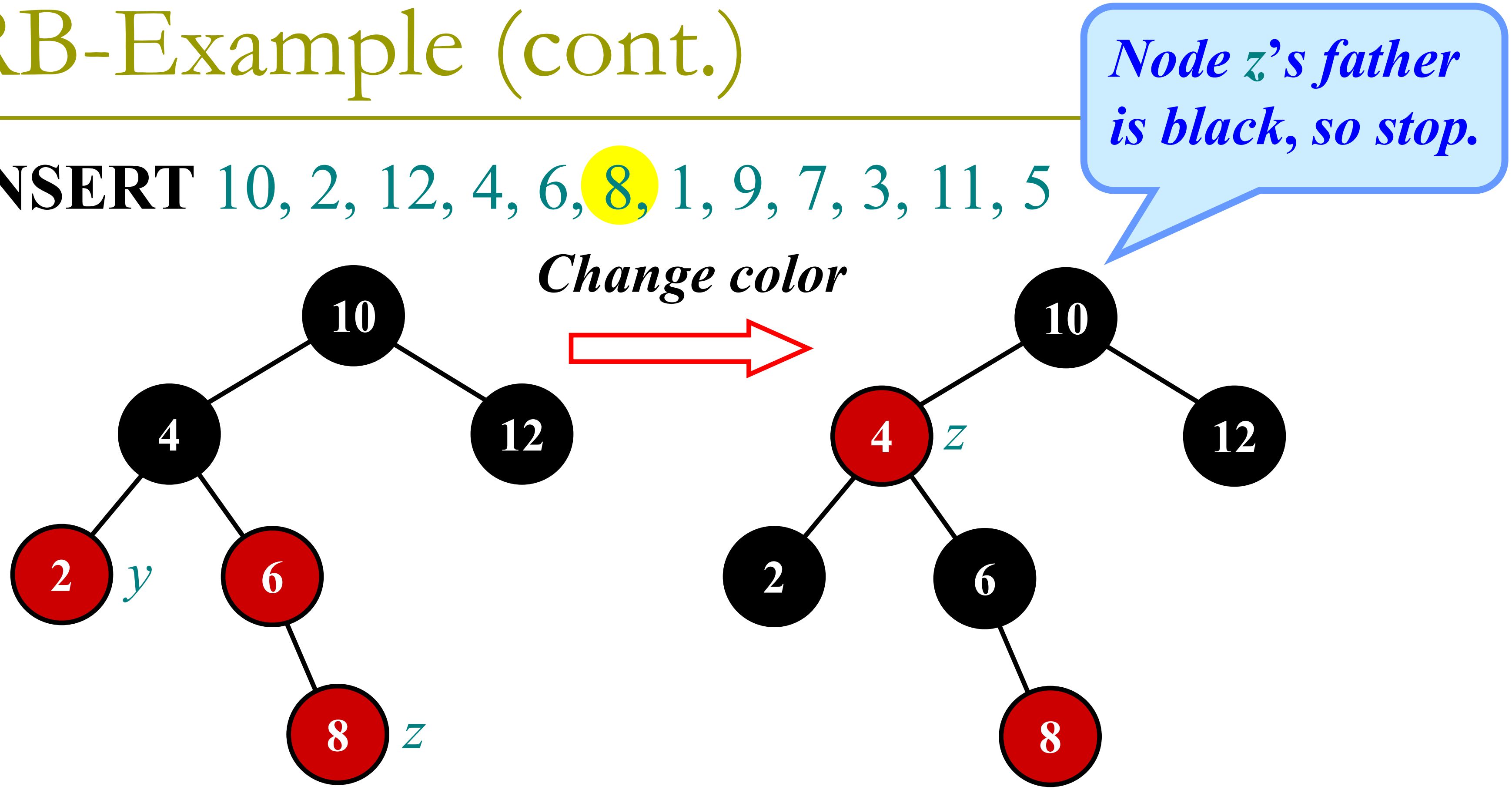
INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



Case 3R: ***z's uncle y is black and z is a right child.***

RB-Example (cont.)

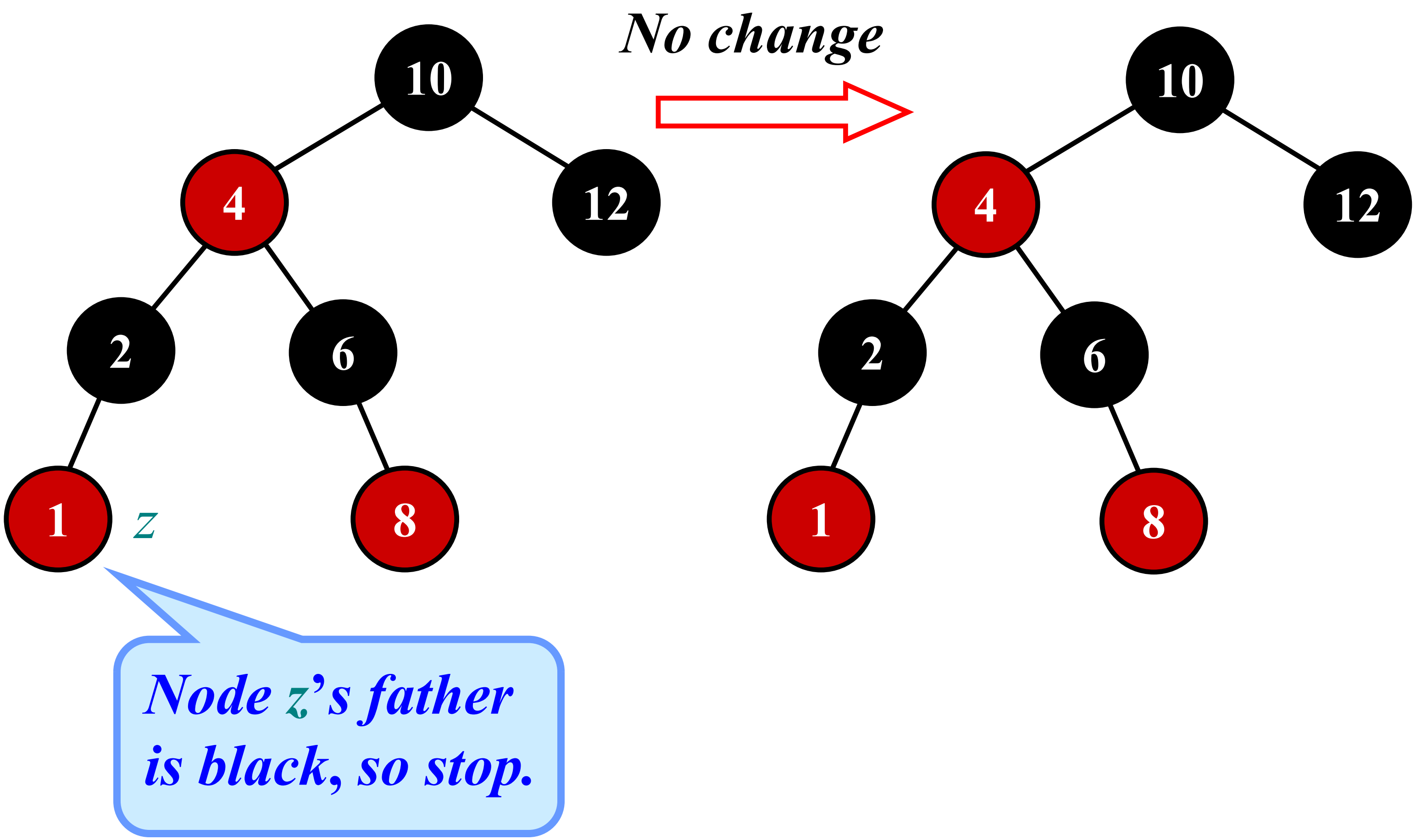
INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



Case 1R: ***z's uncle y is red and we get new z.***

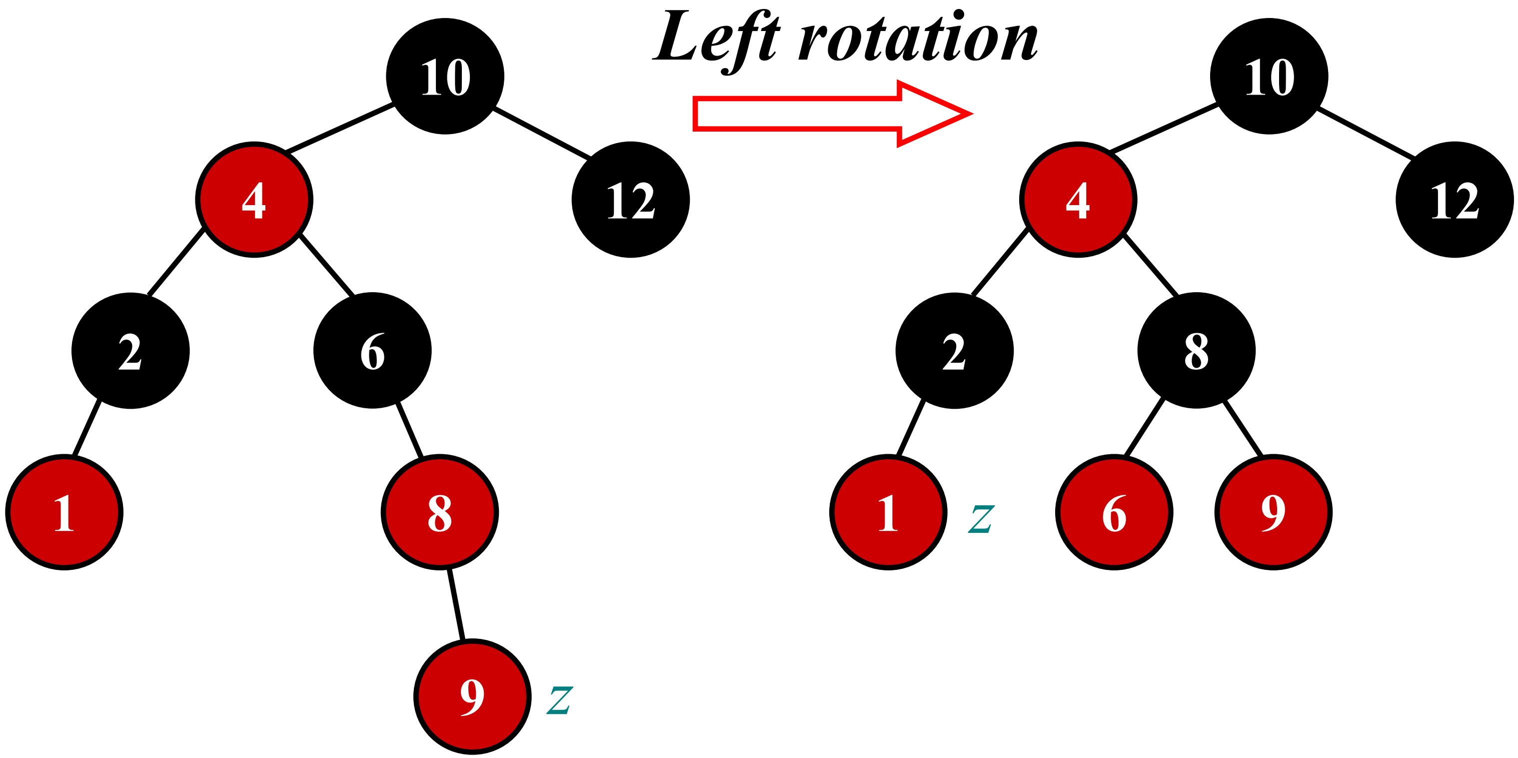
RB-Example (cont.)

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



RB-Example (cont.)

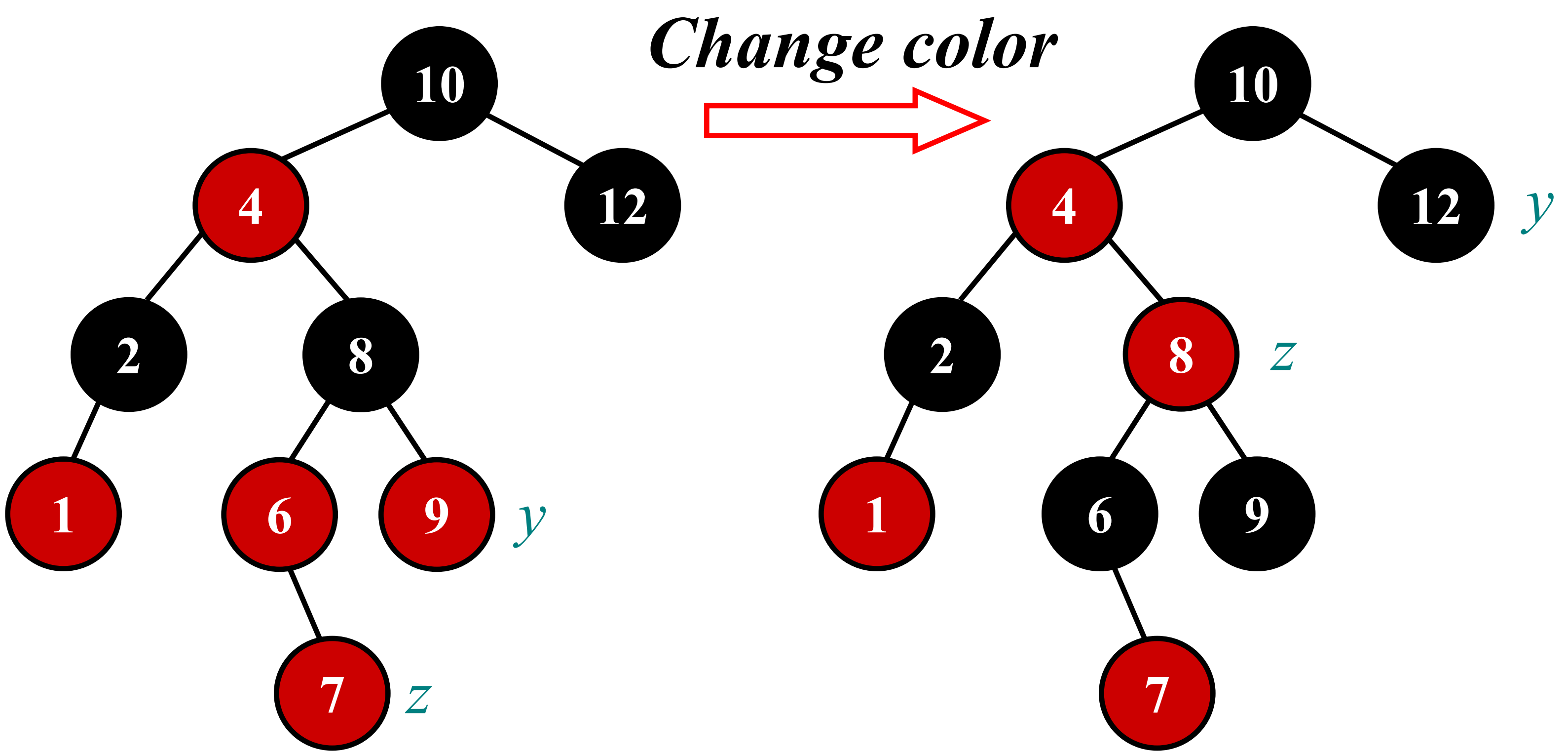
INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



Case 3R: ***z*'s uncle *y* is black and *z* is a right child.**

RB-Example (cont.)

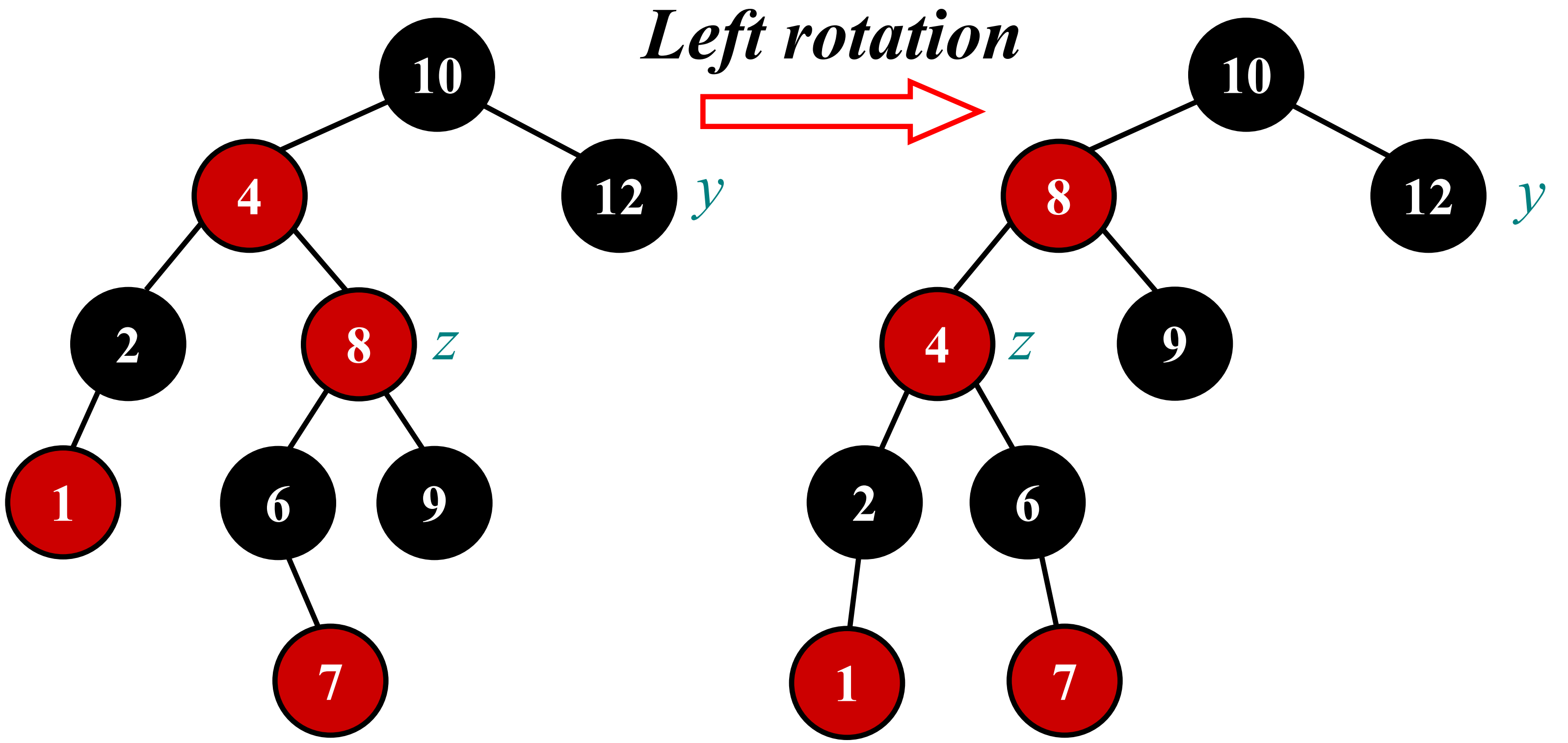
INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



Case 1L: *z's uncle y is red and we get new z.*

RB-Example (cont.)

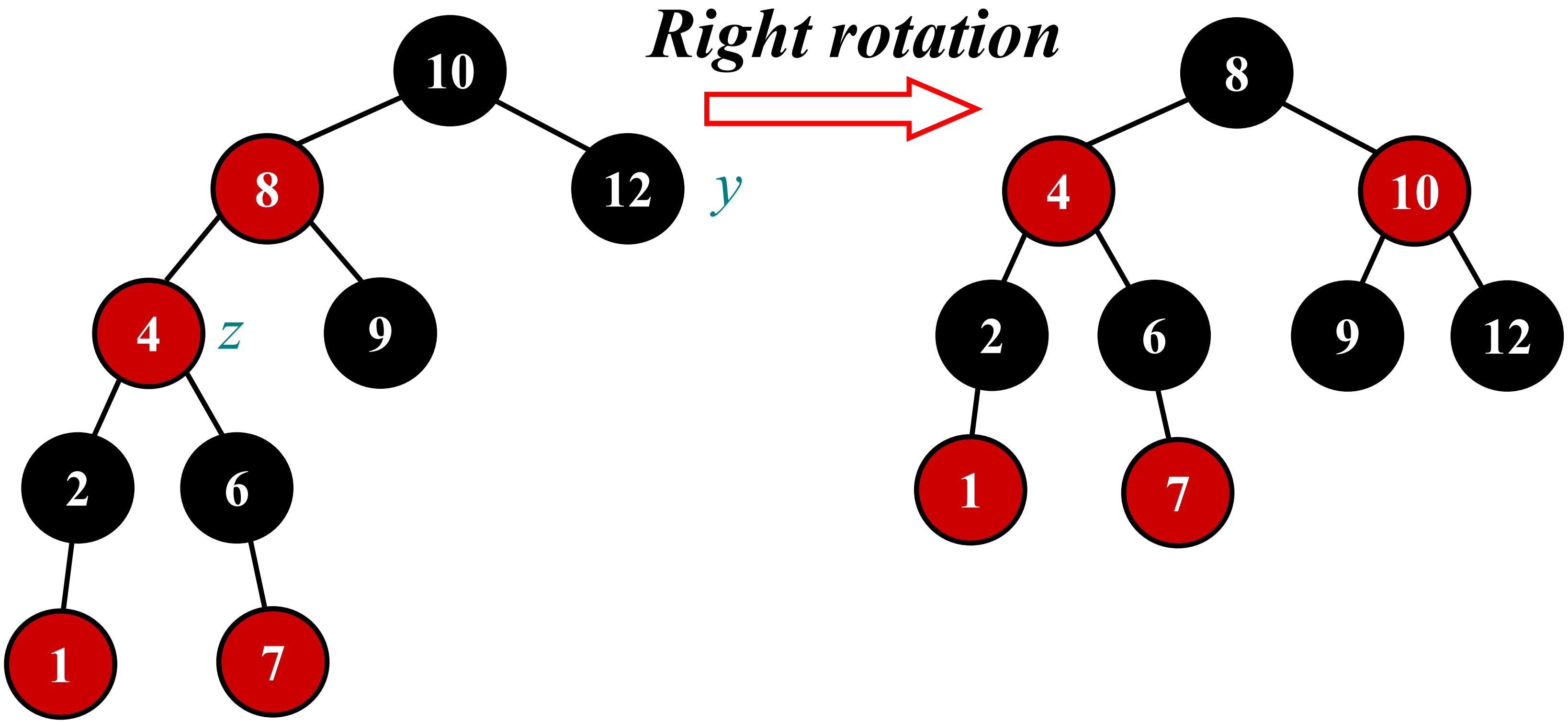
INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



Case 2L: *z's uncle y is black and z is a right child.*

RB-Example (cont.)

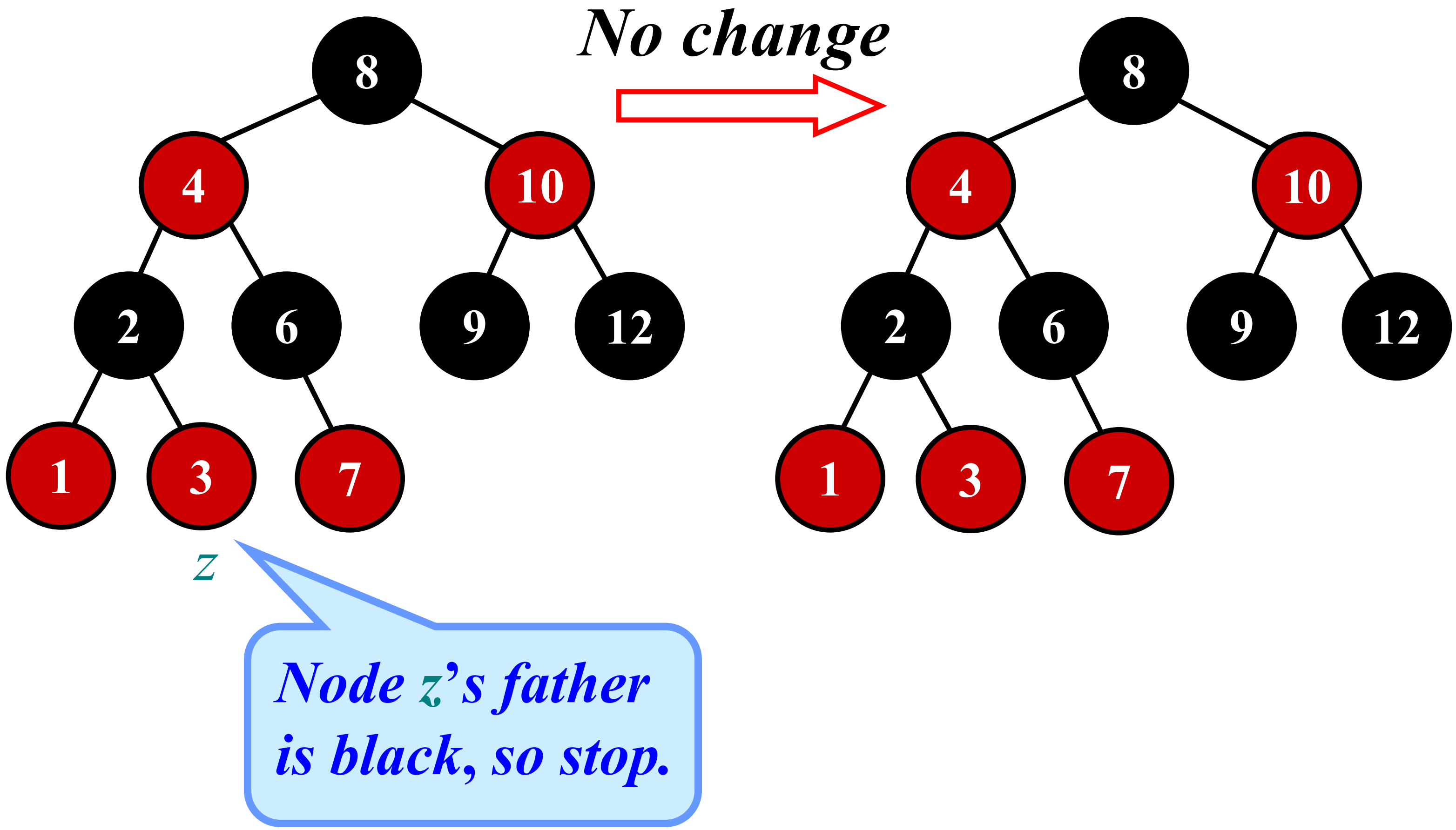
INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



Case 3L: *z's uncle y is black and z is a left child.*

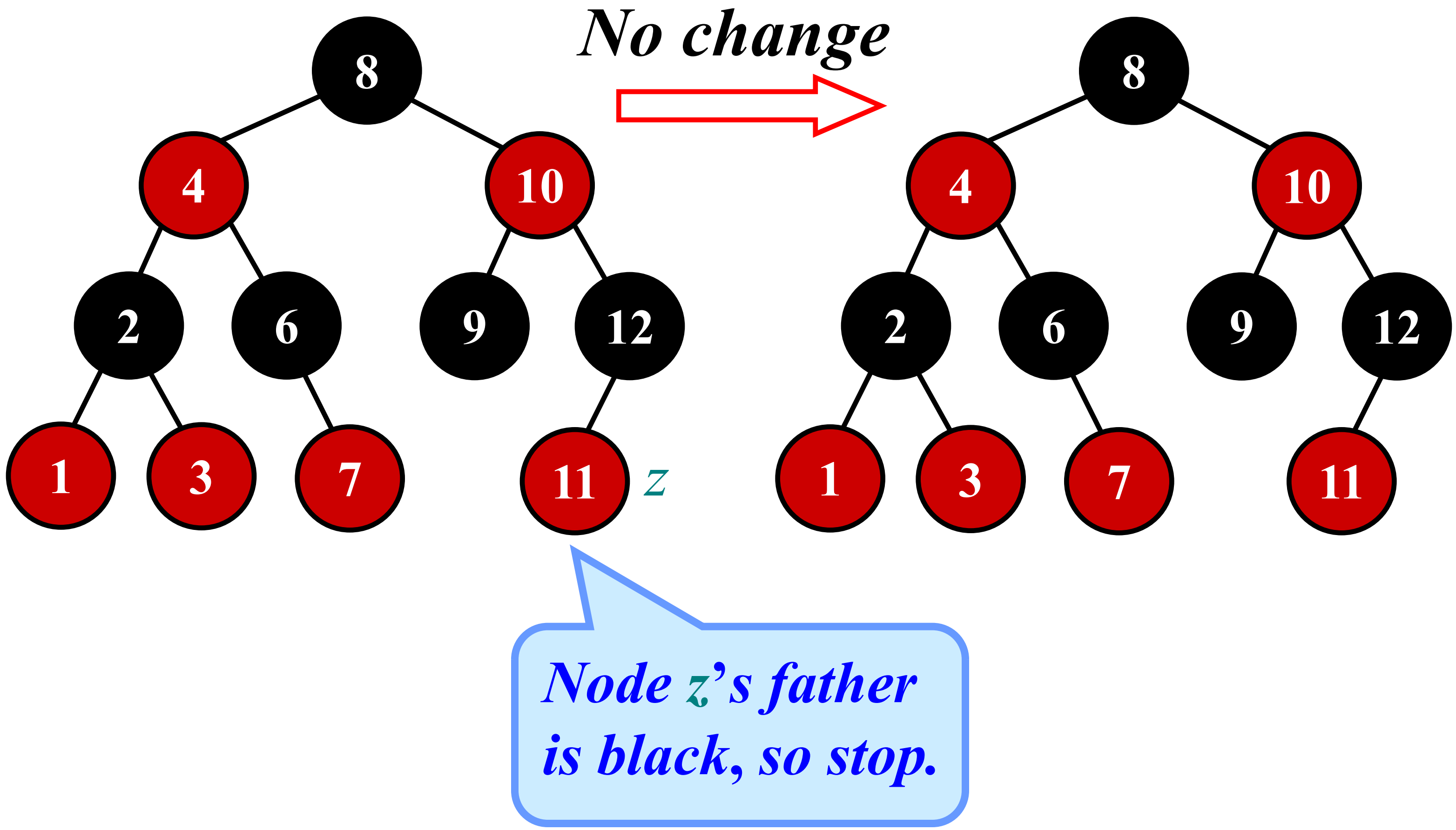
RB-Example (cont.)

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



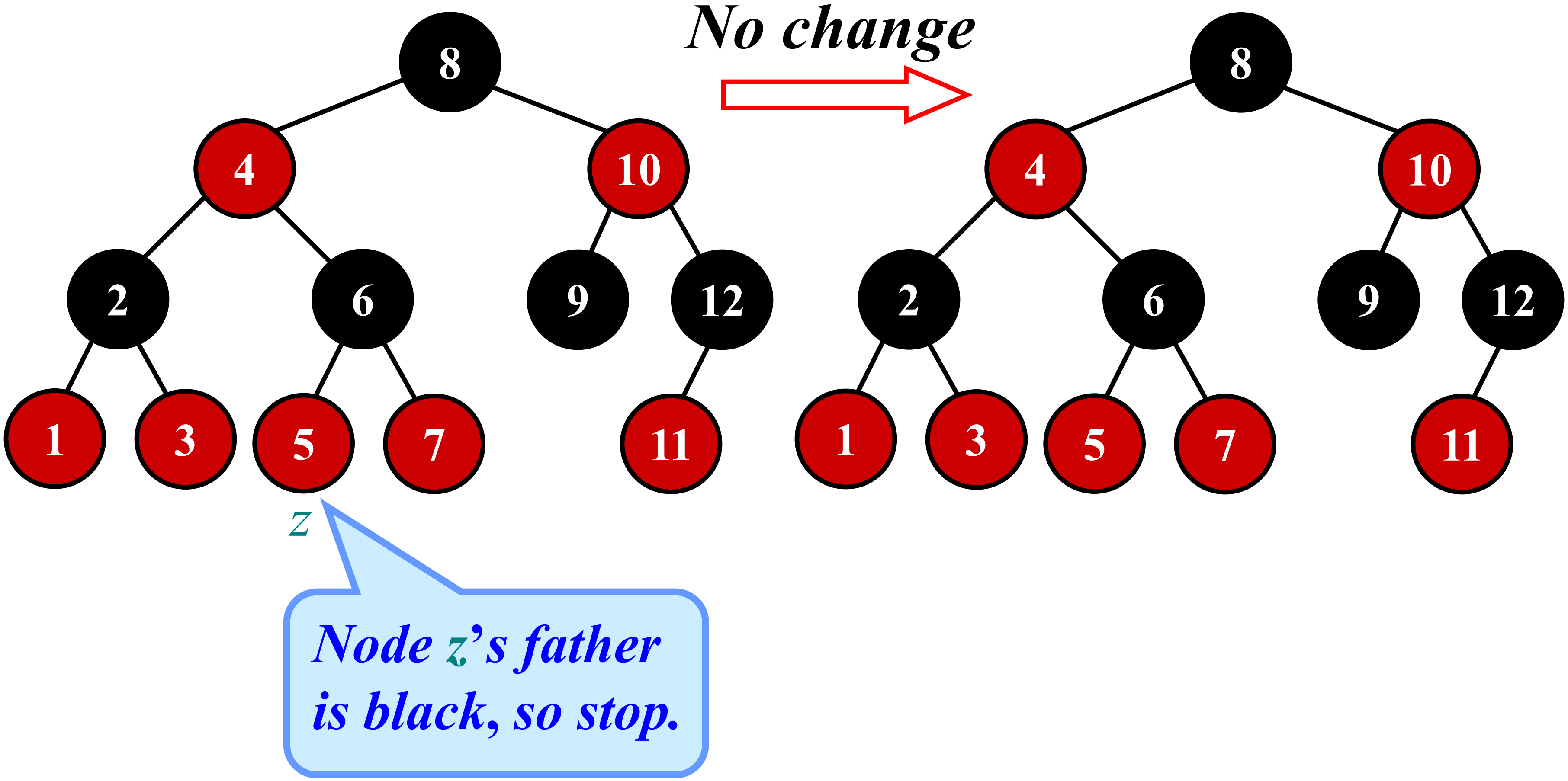
RB-Example (cont.)

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



RB-Example (cont.)

INSERT 10, 2, 12, 4, 6, 8, 1, 9, 7, 3, 11, 5



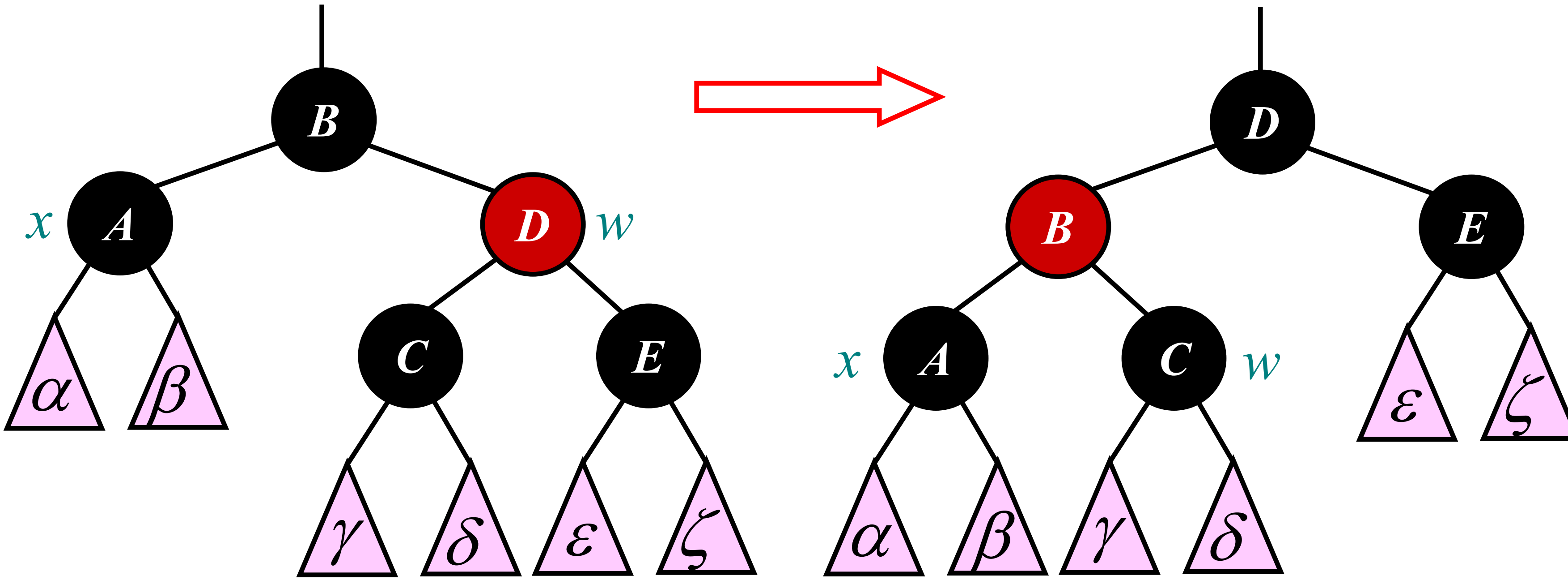
RB-Deletion

```
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 color[y] = BLACK
    then RB-DELETE-FIXUP(T, x)
return y
```

RB-Deletion

Which of the red-black properties can be violated upon the call to RB-DELETE-FIXUP?

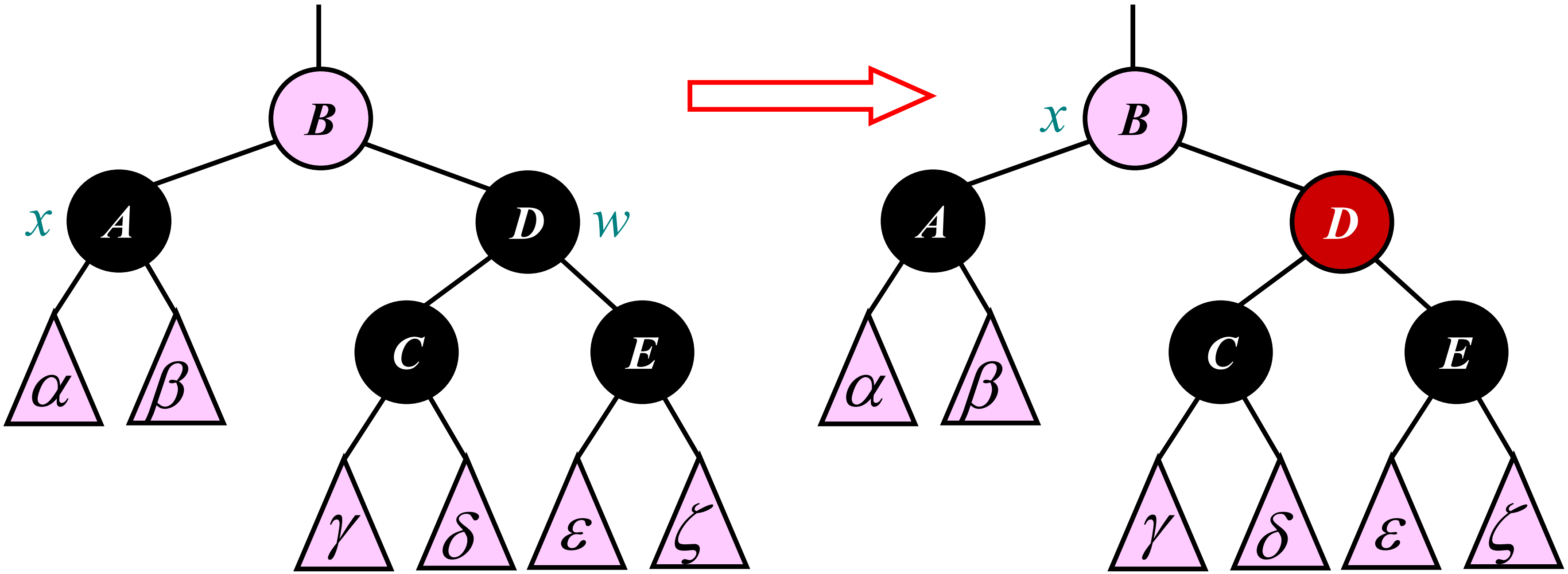
RB-Deletion (case 1)



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

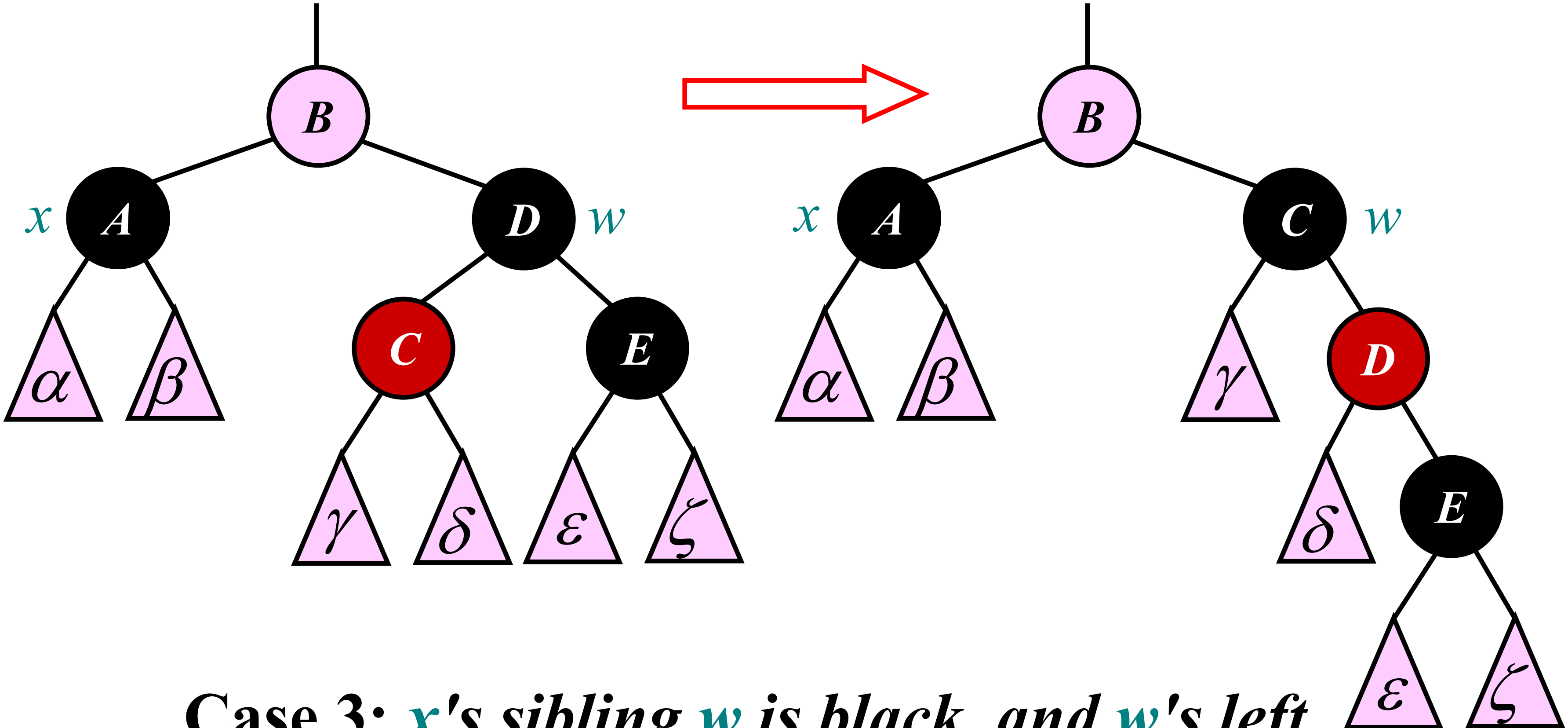
***Convert case* 1 *to* 2, 3, *or* 4.**

RB-Deletion (case 2)



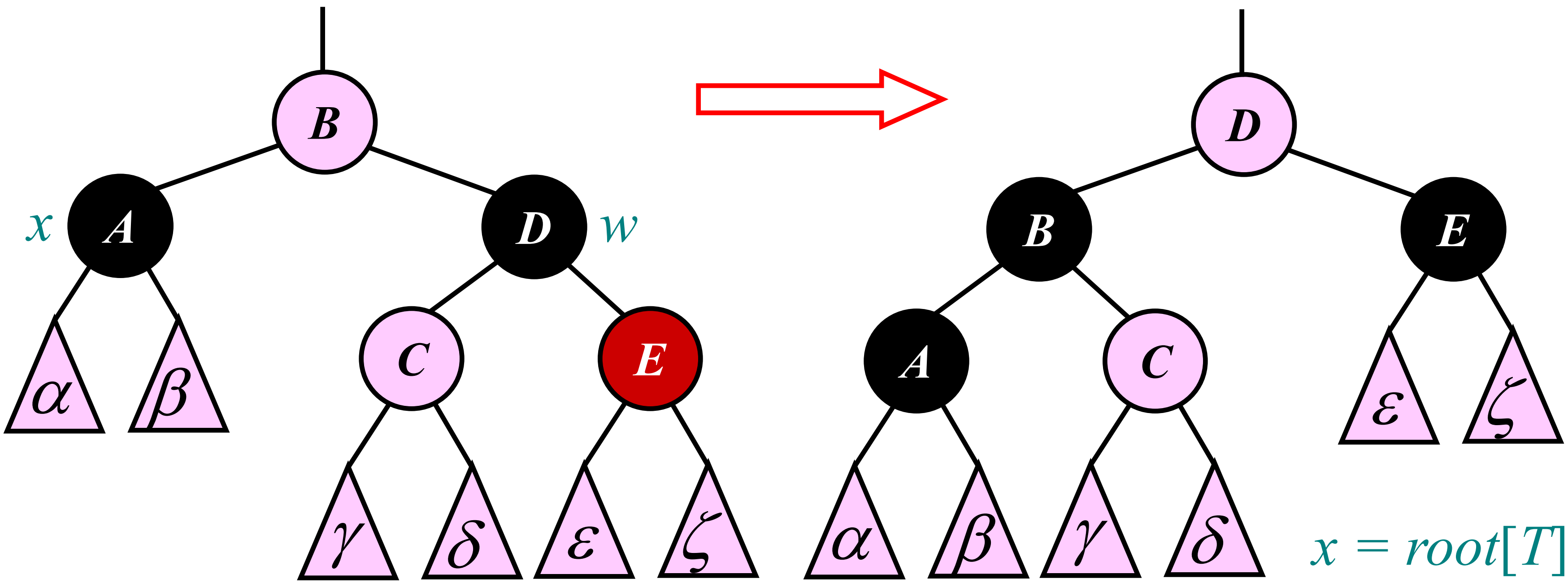
Case 2: *x's sibling w is black, and both of w's children are black. Get the new node x.*

RB-Deletion (case 3)



Case 3: *x's sibling w is black, and w's left children is red, and w's right child is black. Convert case* 3 *to* 4.

RB-Deletion (case 4)



Case 4: *x's sibling w is black, and w's right child is red.*
Terminate the while loop.

RB-tree deletion

Types		Operation
*z is **left** child*	***Case* 1L:** *x's sibling w is red.*	***Left** rotation, $p(x)$.*
	***Case* 2L:** *x's sibling w is black and both of w's children are black.*	*Change color.*
	***Case* 3L:** *x's sibling w is black, and w's **left** children is red, and w's **right** child is black.*	***Right** rotation, w.*
	***Case* 4L:** *x's sibling w is black, and w's **right** child is red.*	***Left** rotation, $p(x)$.*
*z is **right** child*	***Case* 1R:** *x's sibling w is red.*	***Right** rotation, $p(x)$.*
	***Case* 2R:** *x's sibling w is black and both of w's children are black.*	*Change color.*
	***Case* 3R:** *x's sibling w is black, and w's **right** children is red, and w's **left** child is black.*	***Left** rotation, w.*
	***Case* 4R:** *x's sibling w is black, and w's **left** child is red.*	***Right** rotation, $p(x)$.*

RB-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-ROTATION(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
```

RB-Deletion

```
          else if color[right[w]] = BLACK
                  then color[left[w]] ← BLACK          Case 3
                       color[w] ← RED                  Case 3
                       RIGHT-ROTATION(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-ROTATION(T, p[x])               Case 4
                  x ← root[T]                          Case 4
      else (same as then clause
               with "right" and "left" exchanged)
color[x] ← BLACK
```

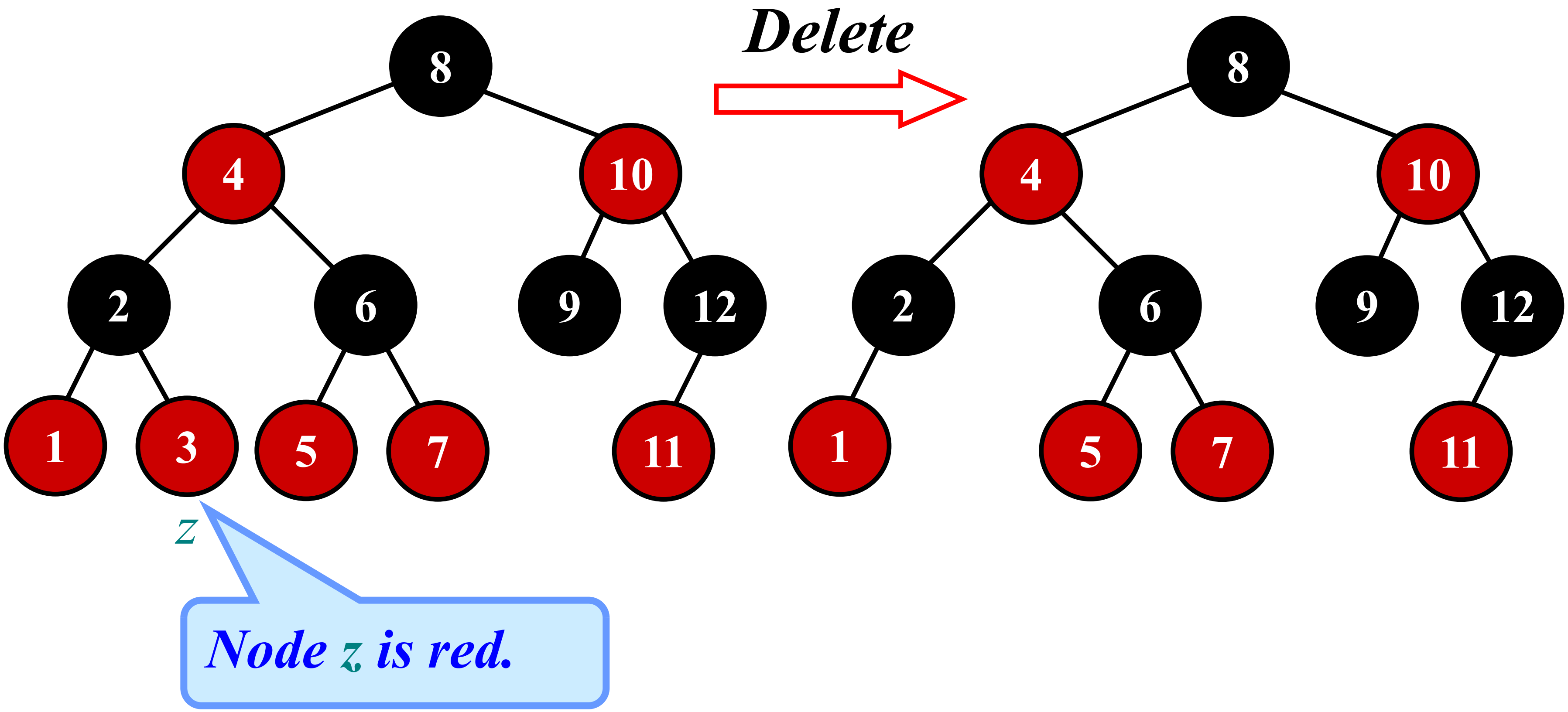
Analysis of RB-Deletion

What is the running time of RB-DELETE?

Running time:
$O(lgn)$

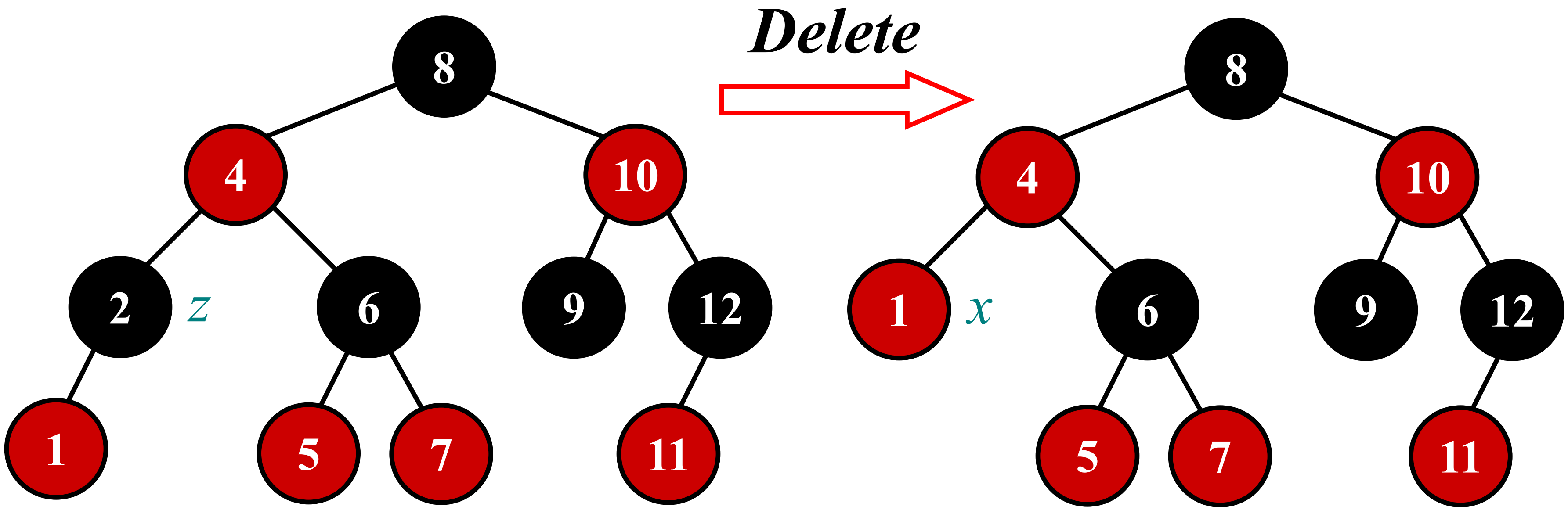
RB-Example

Delete 3



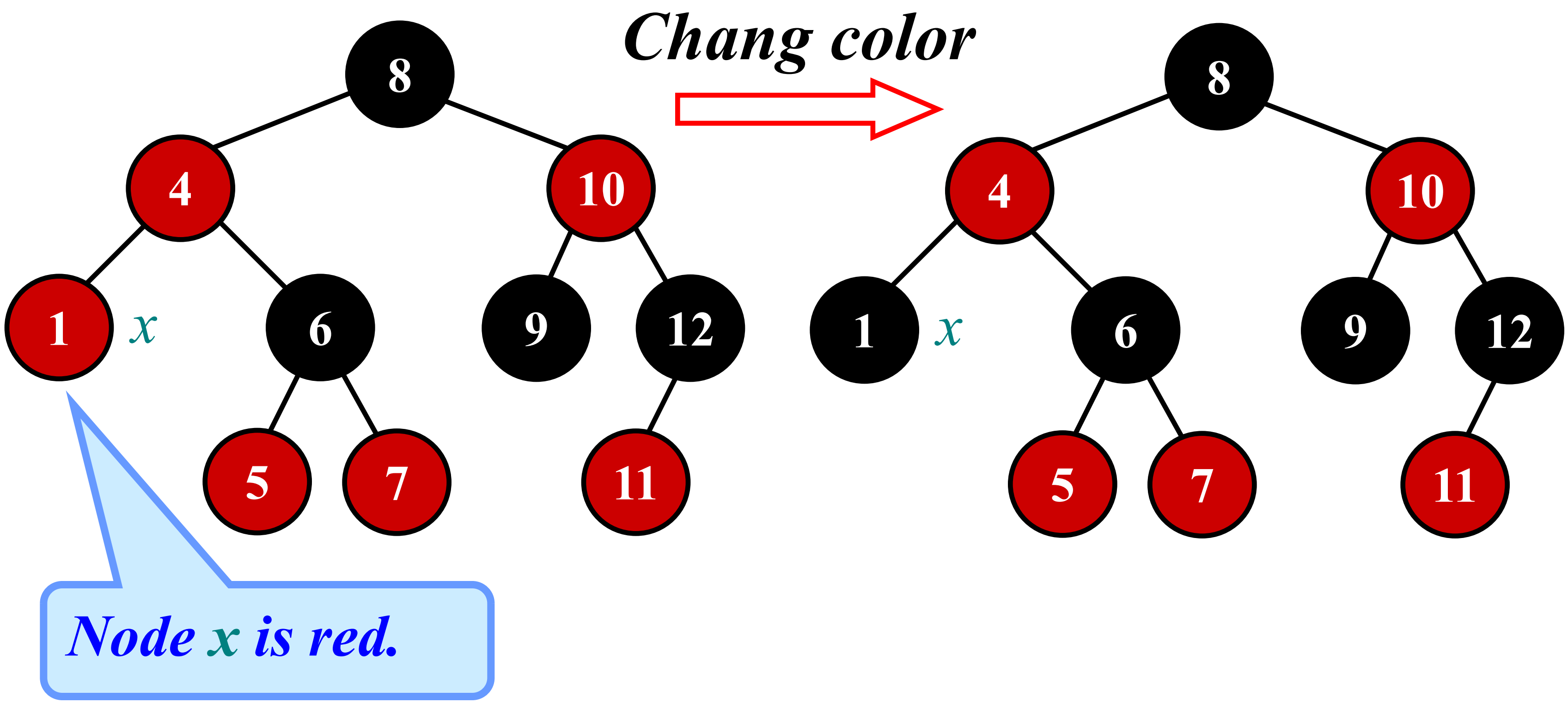
RB-Example

Delete 2



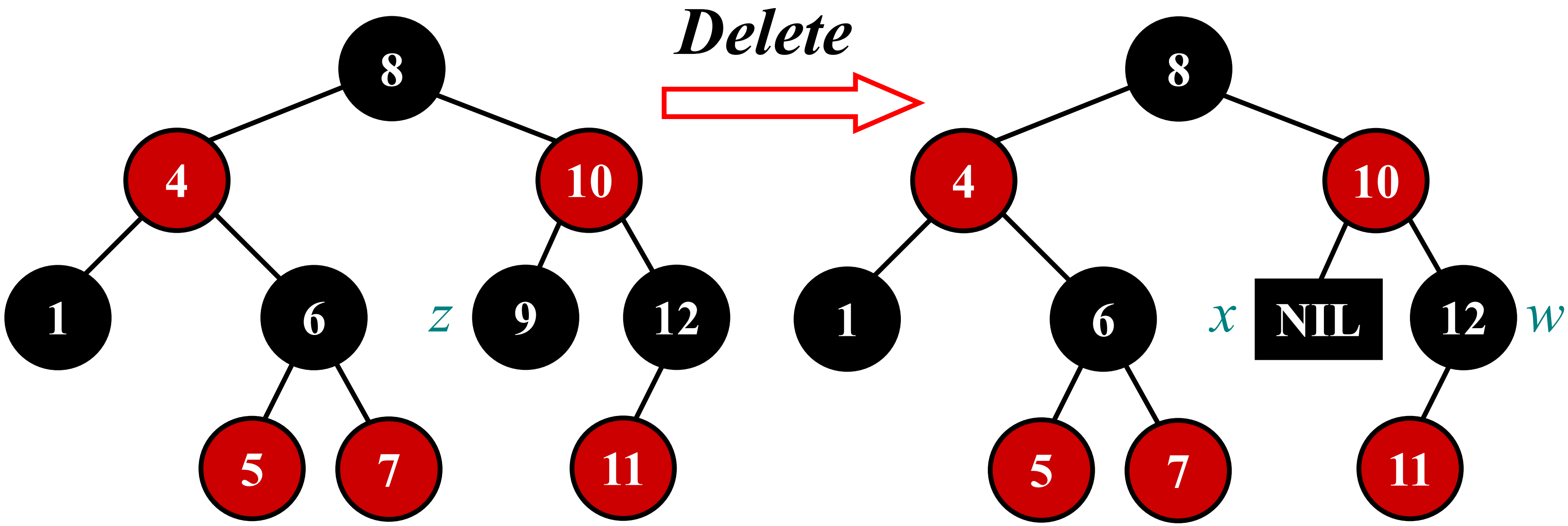
RB-Example

Delete 2



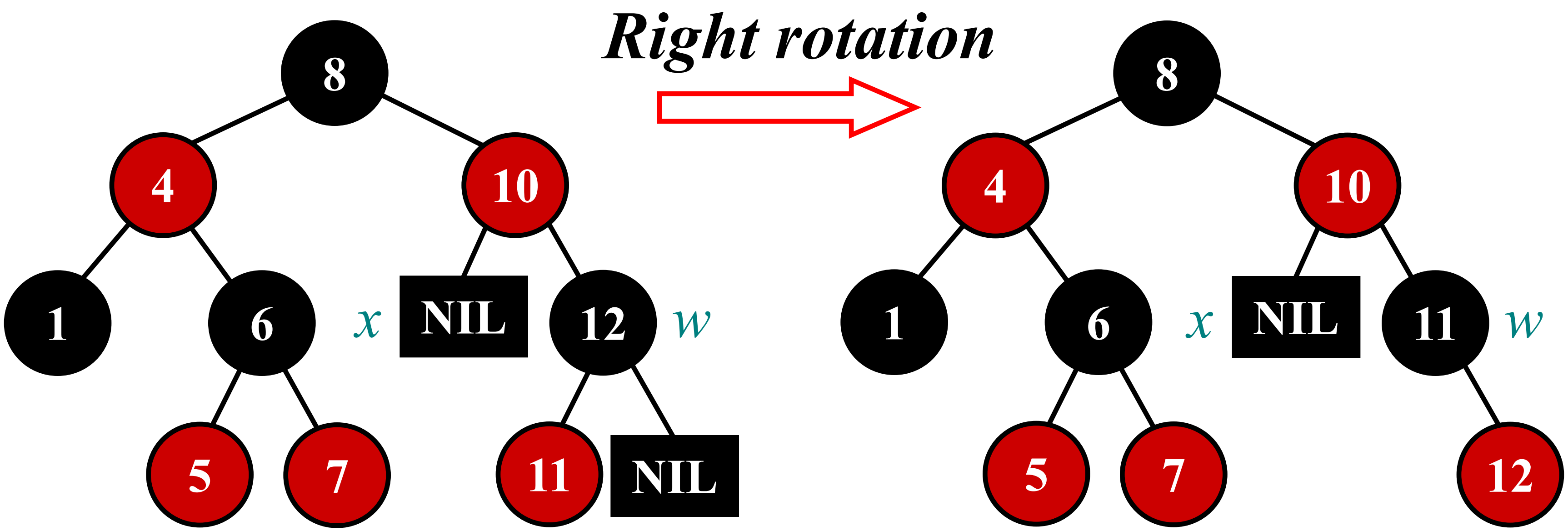
RB-Example

Delete 9



RB-Example

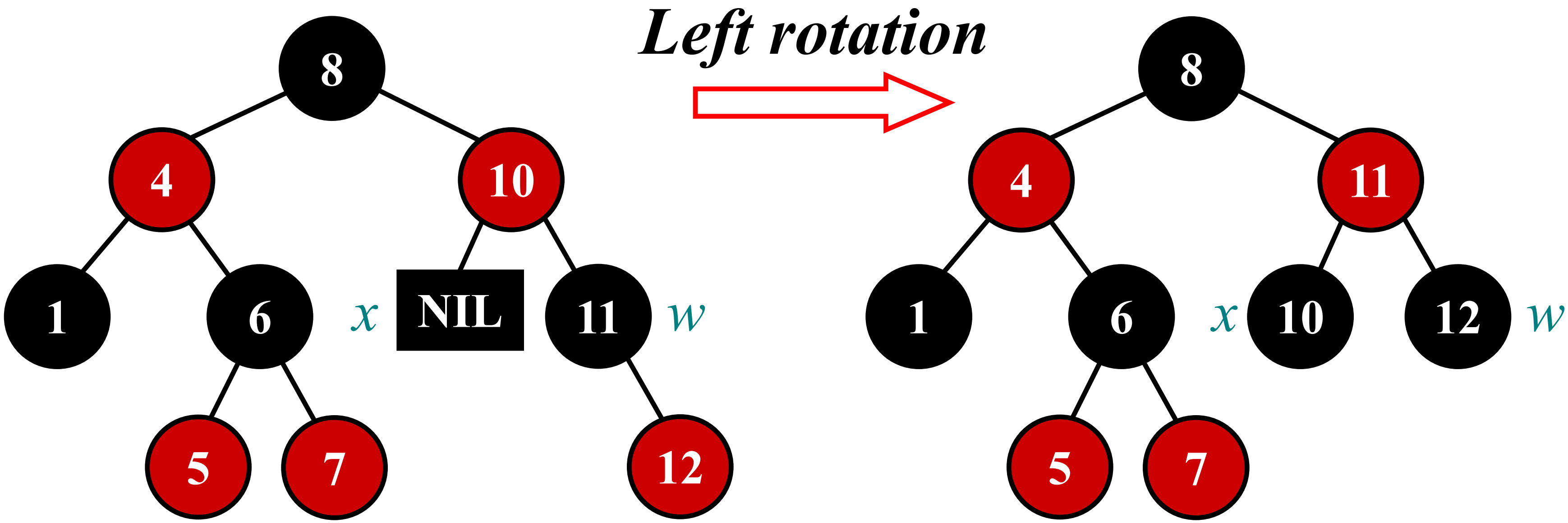
Delete 9



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

RB-Example

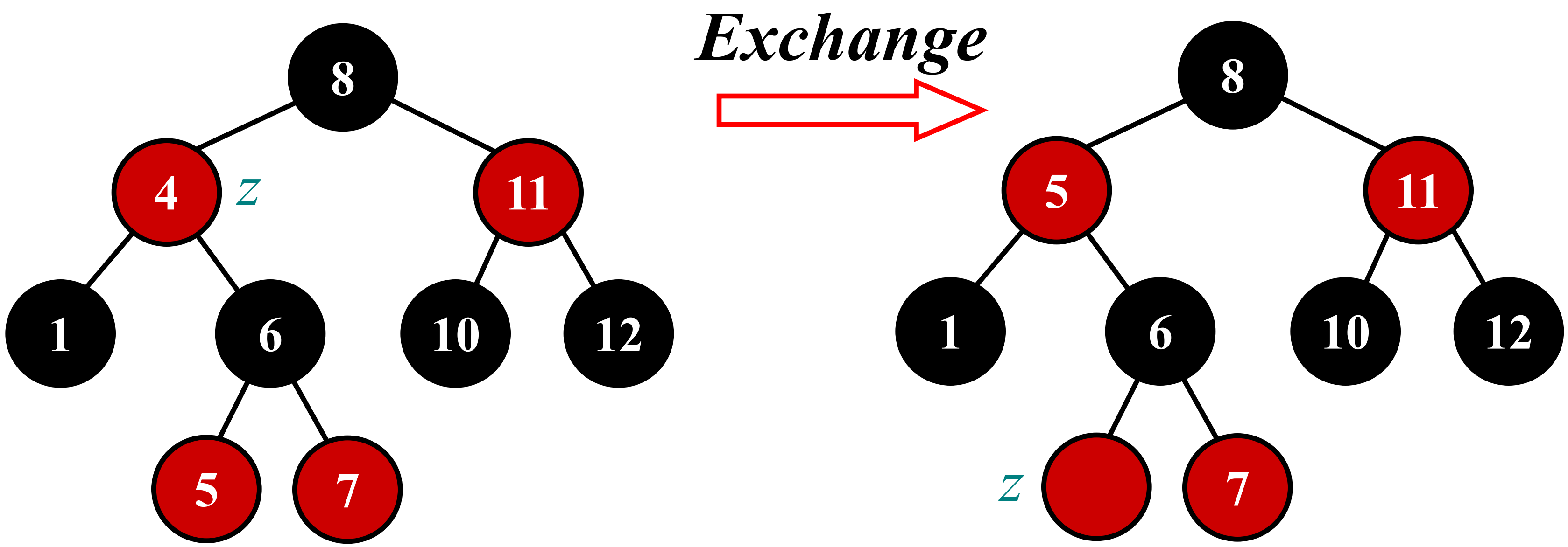
Delete 9



Case 4L: ***x's sibling w is black, and w's right child is red.***

RB-Example

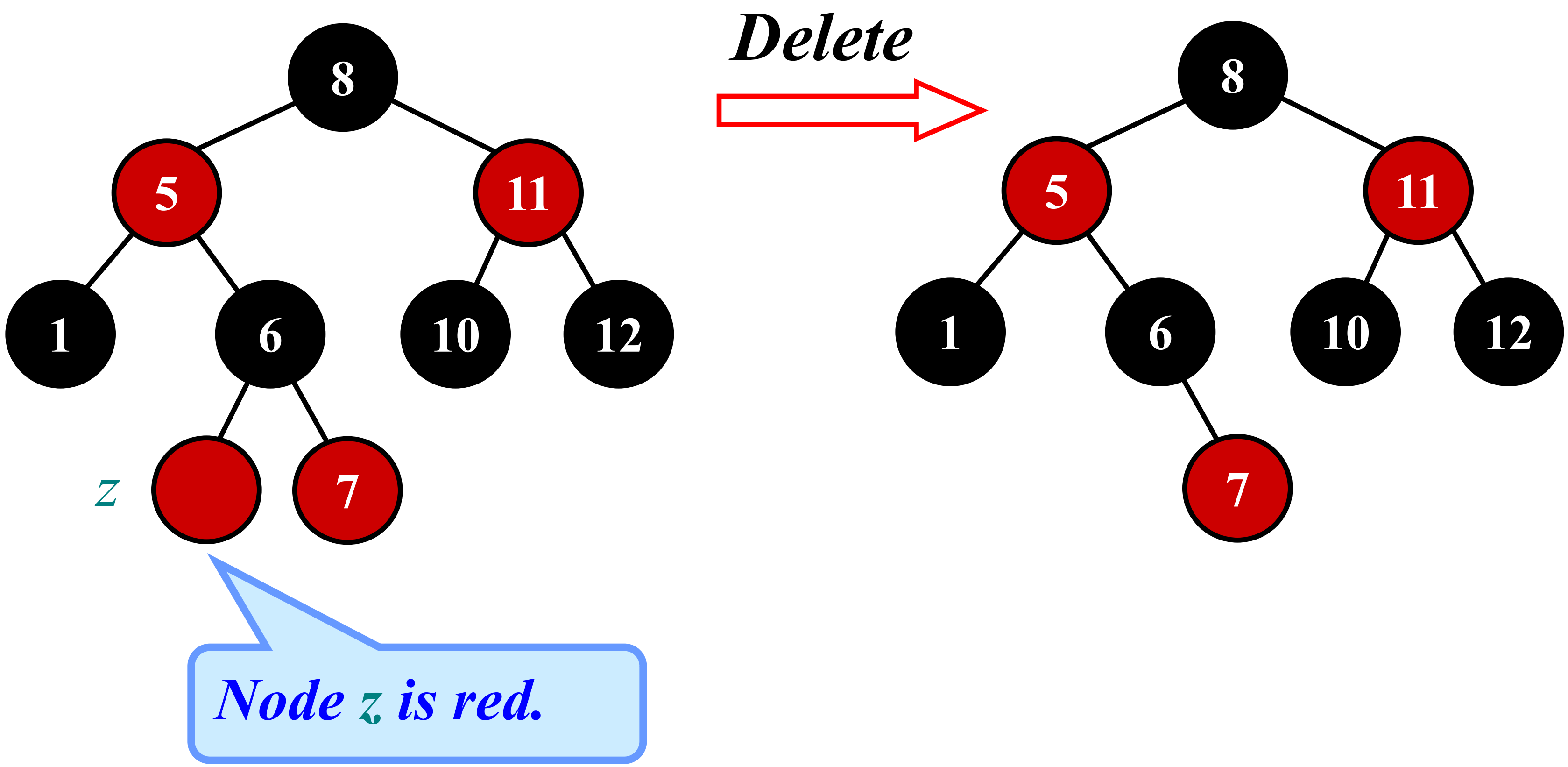
Delete 4



Which is 4's successor?

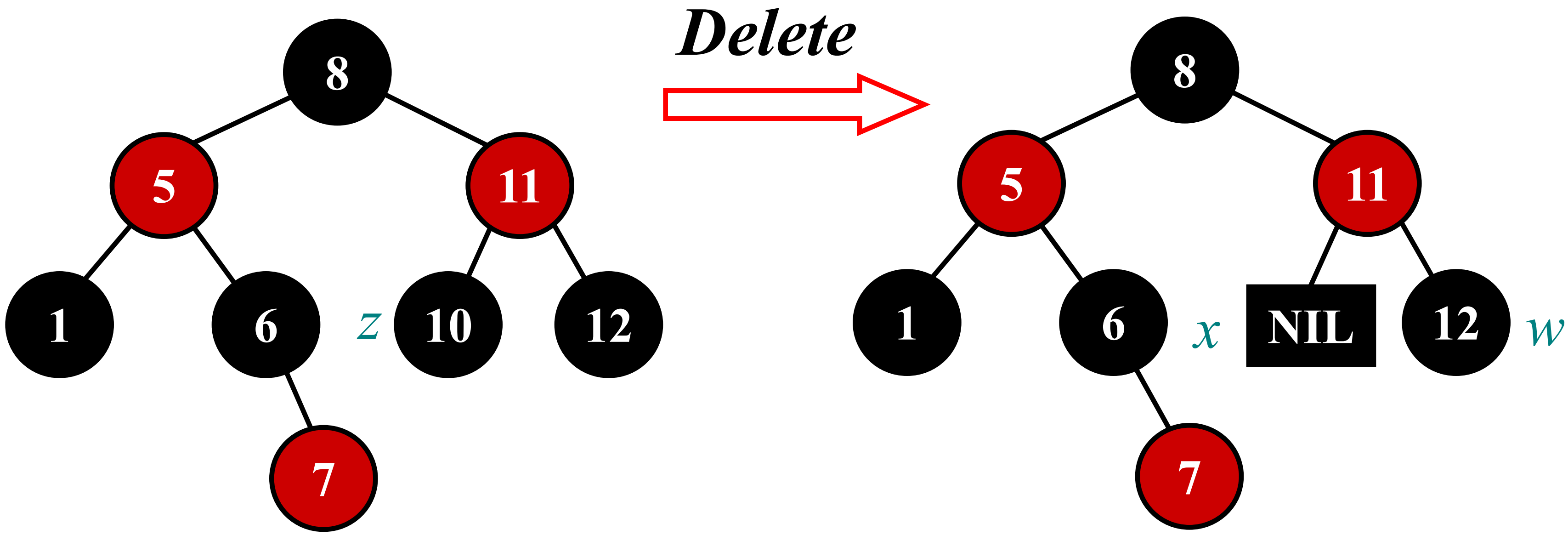
RB-Example

Delete 4



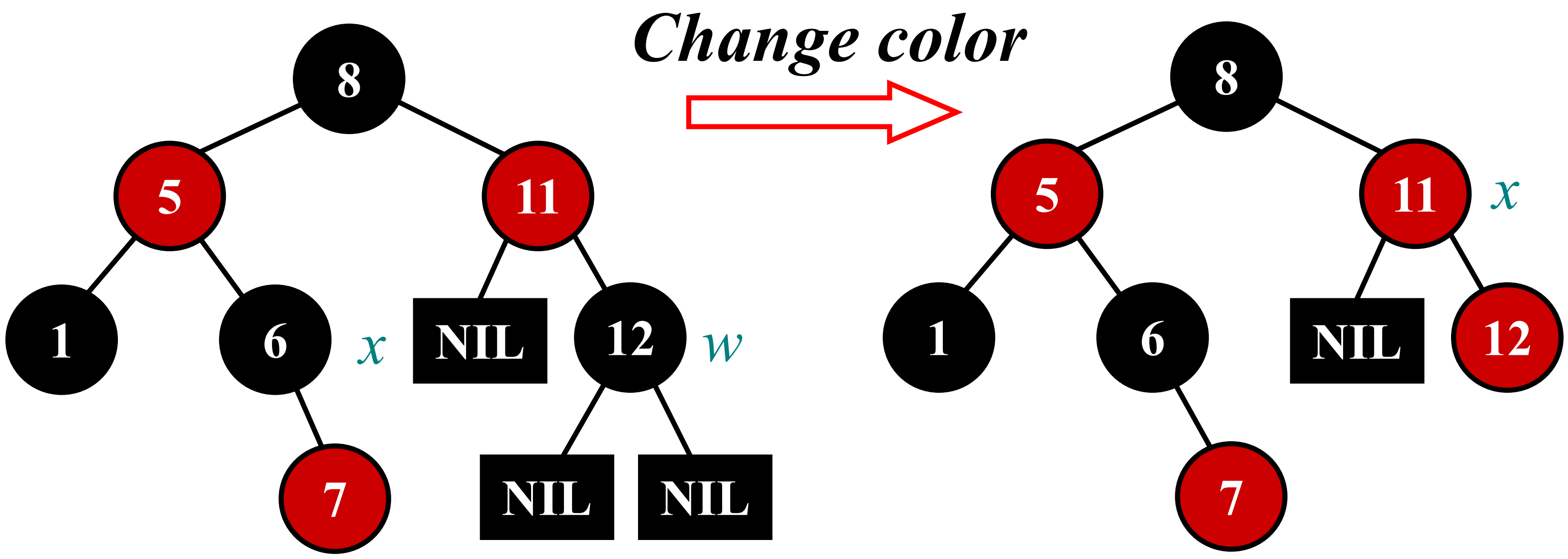
RB-Example

Delete 10



RB-Example

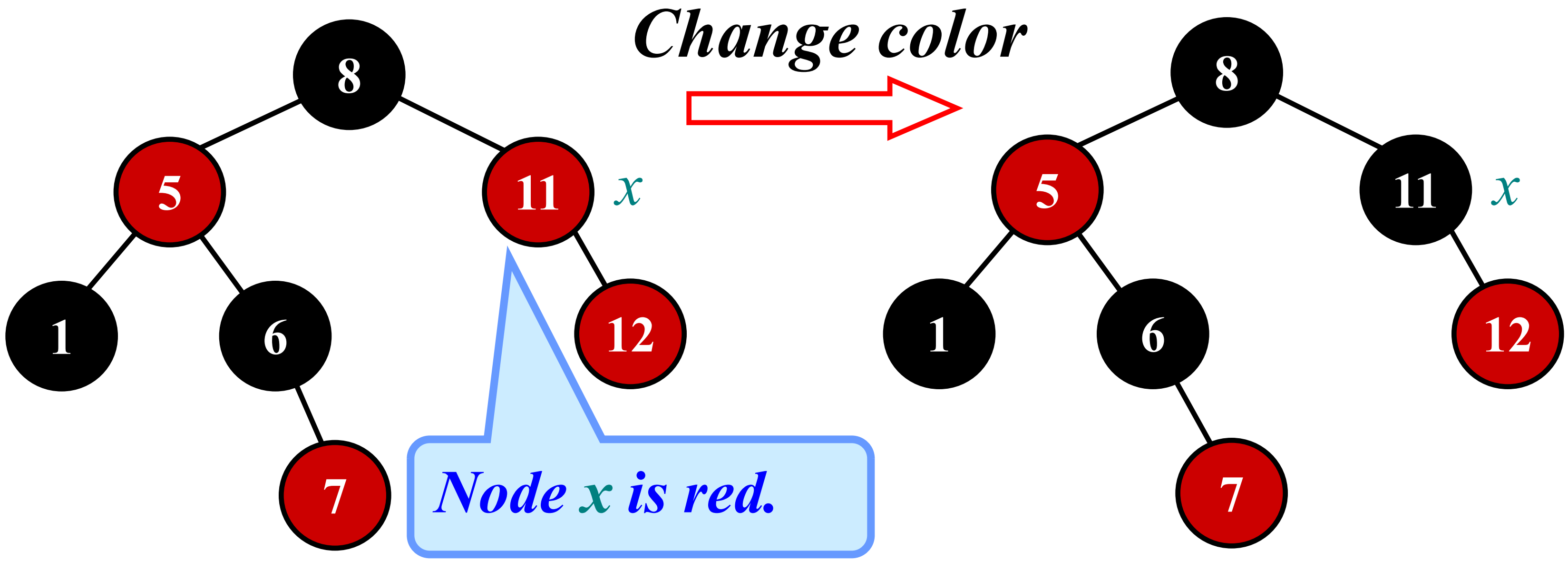
Delete 10



Case 2L: *x's sibling w is black, and both of w's children are black. Then, we get new x.*

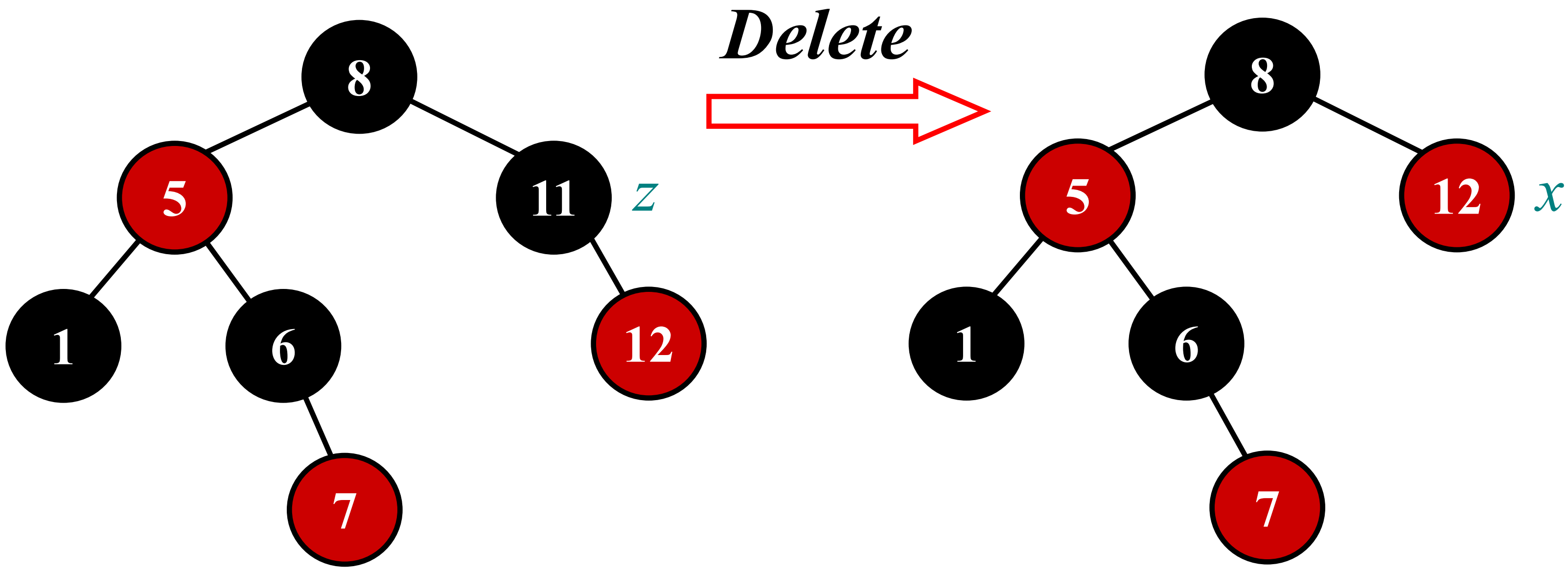
RB-Example

Delete 10



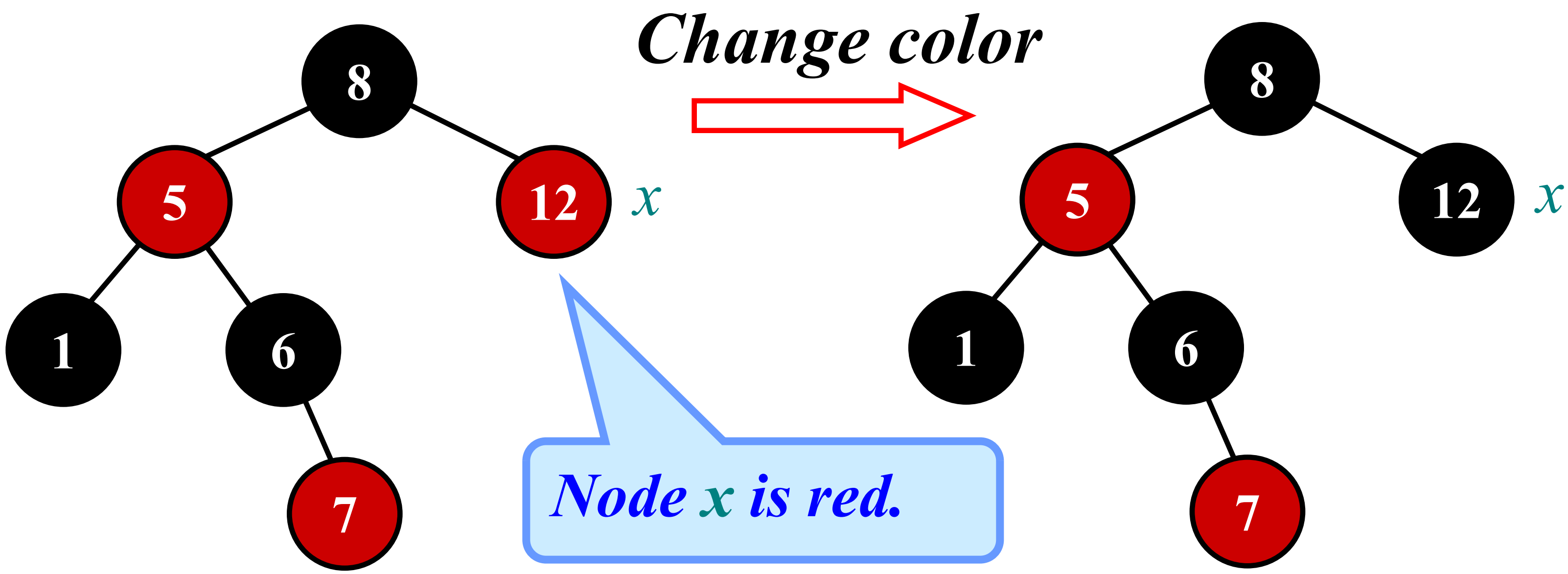
RB-Example

Delete 11



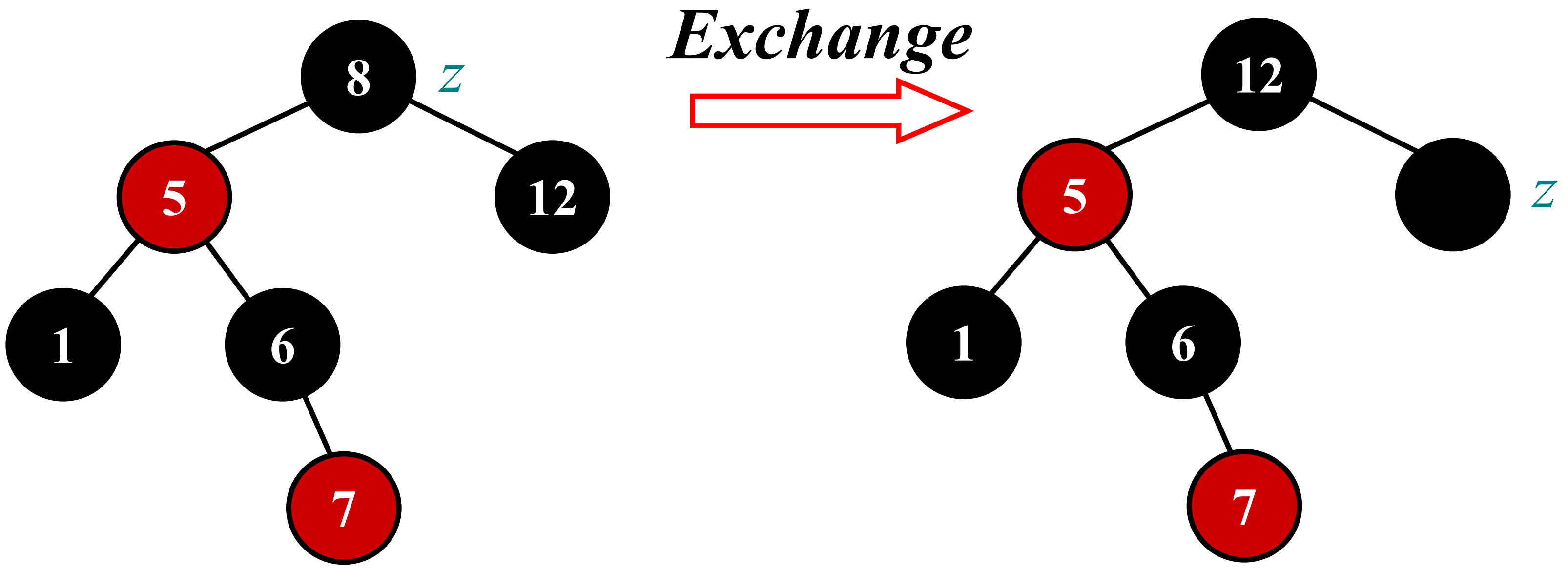
RB-Example

Delete 11



RB-Example

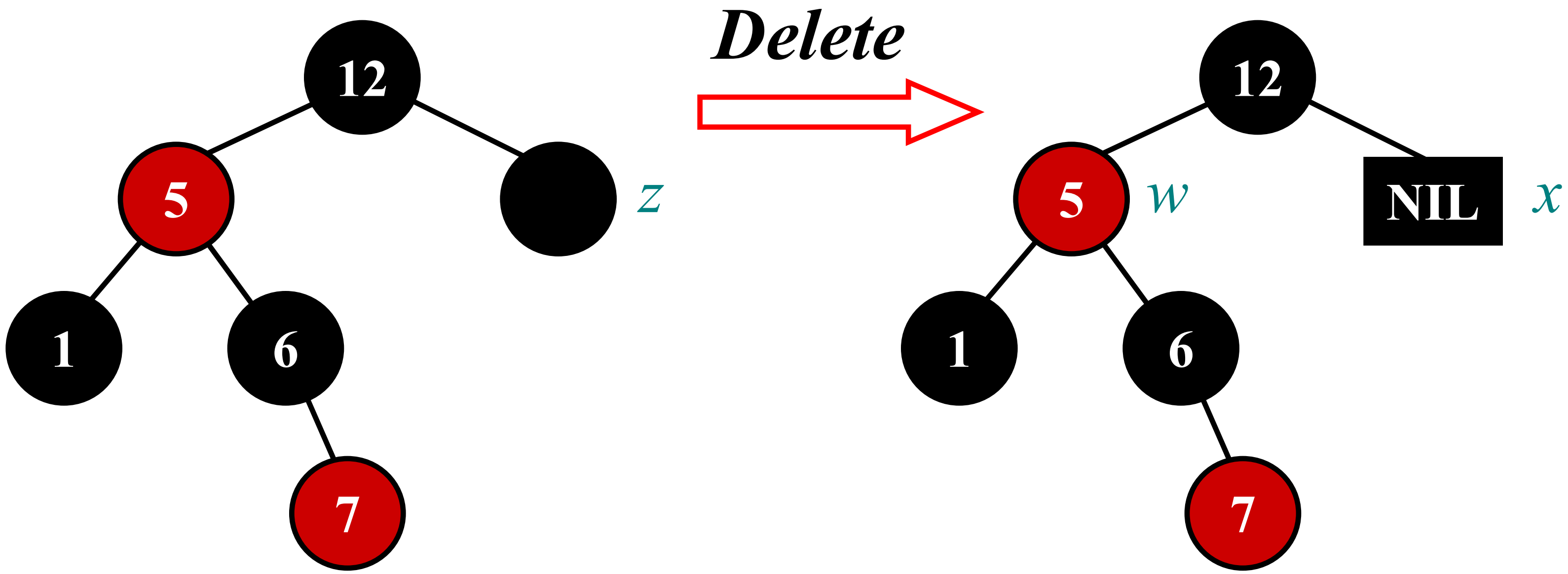
Delete 8



Which is 8's successor?

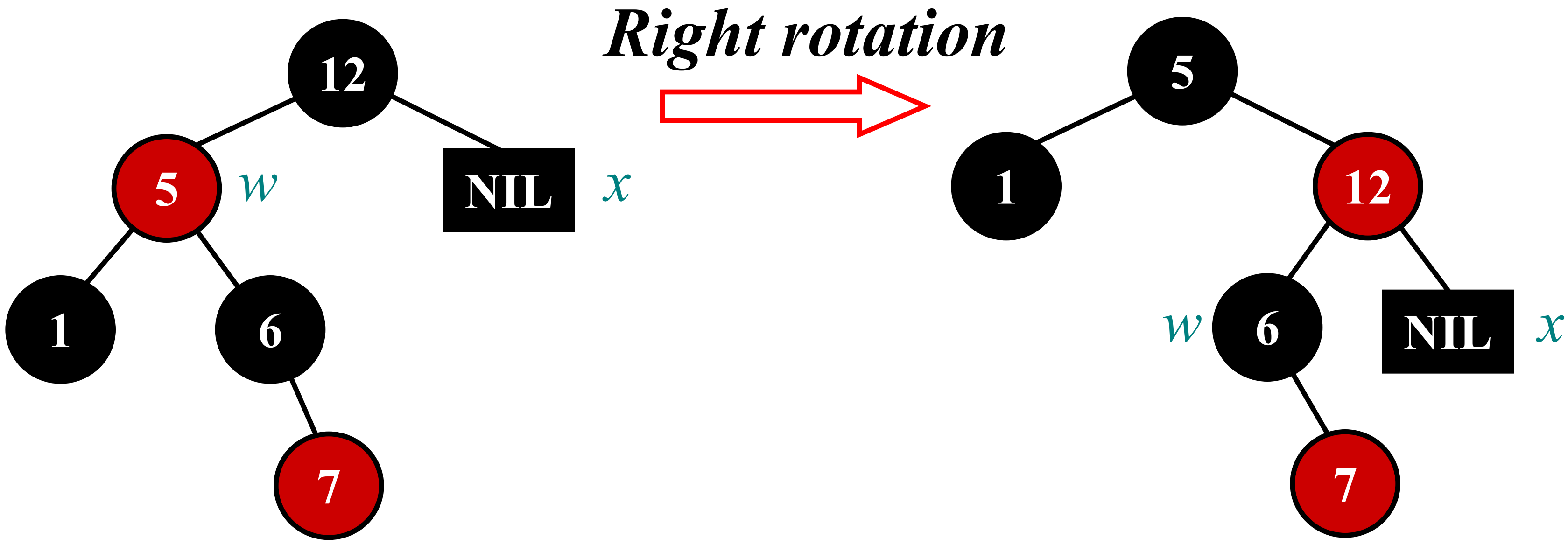
RB-Example

Delete 8



RB-Example

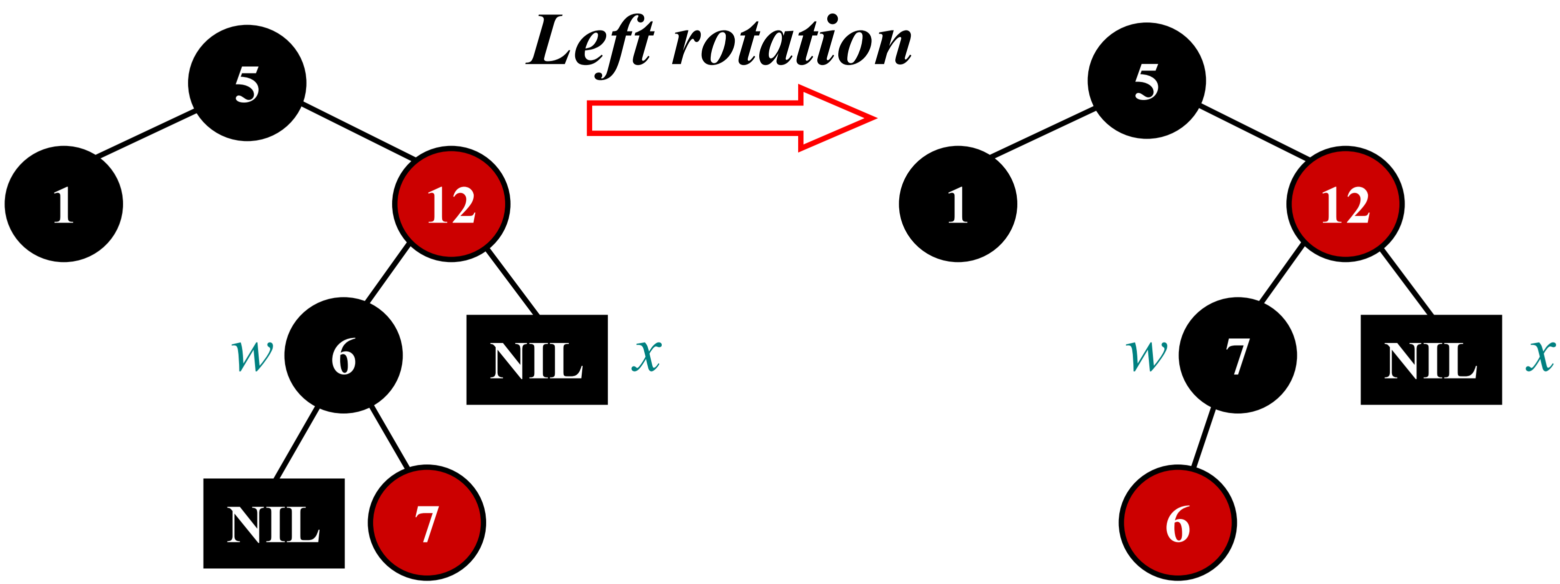
Delete 8



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

RB-Example

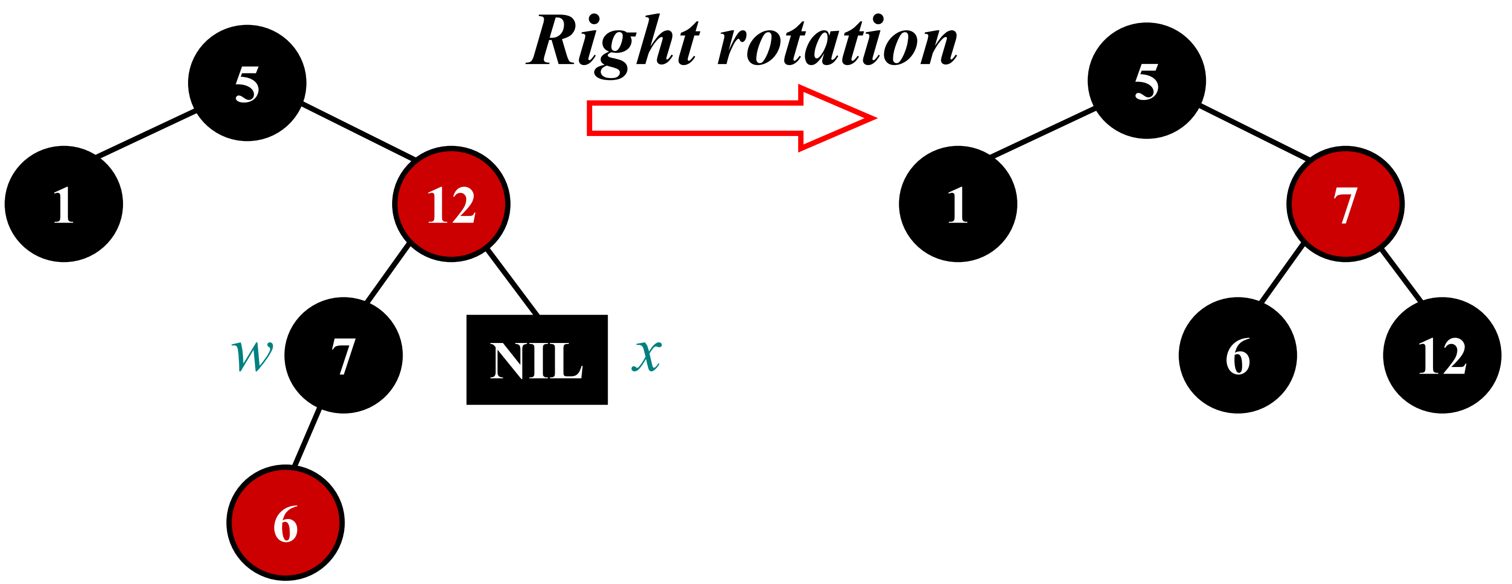
Delete 8



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

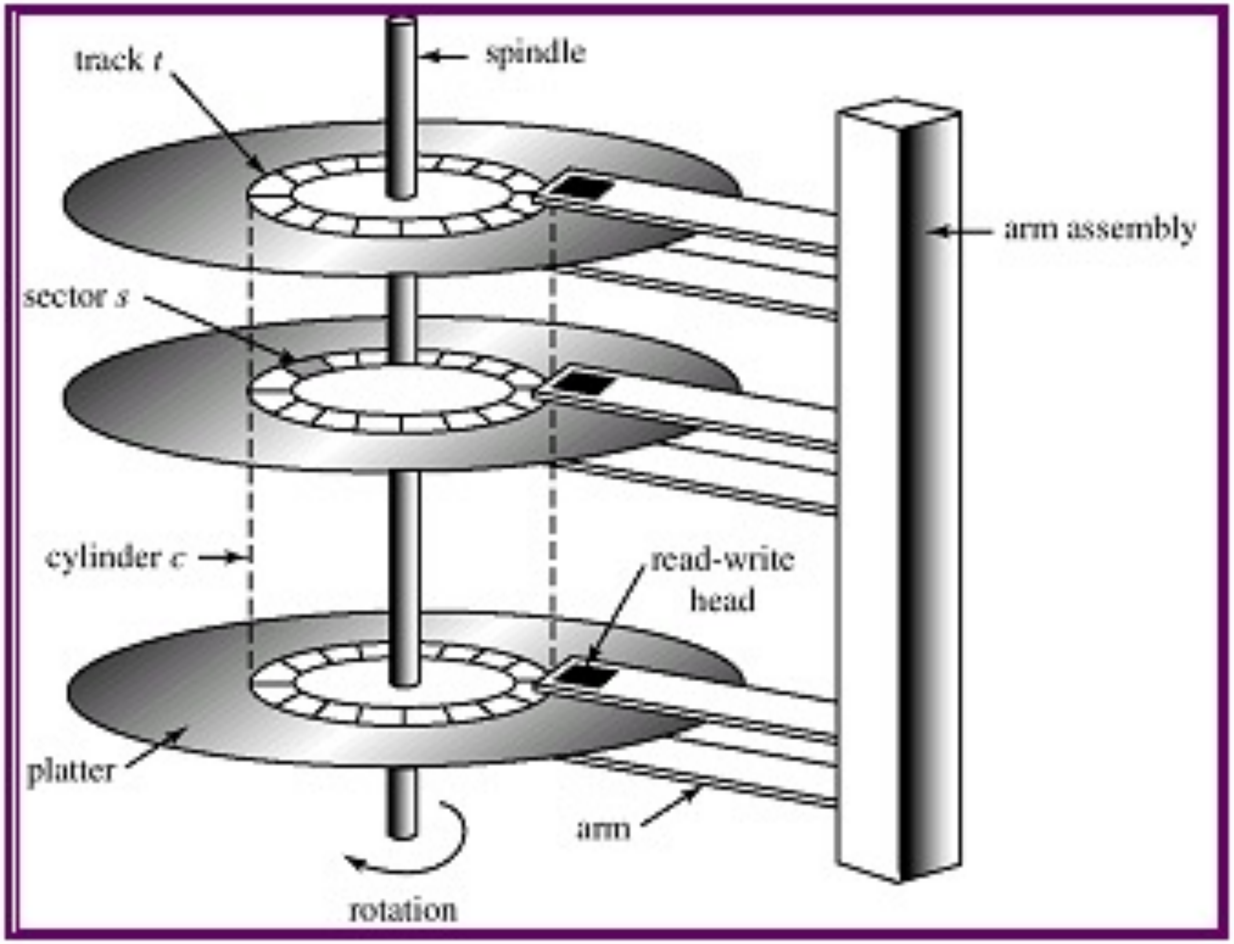
RB-Example

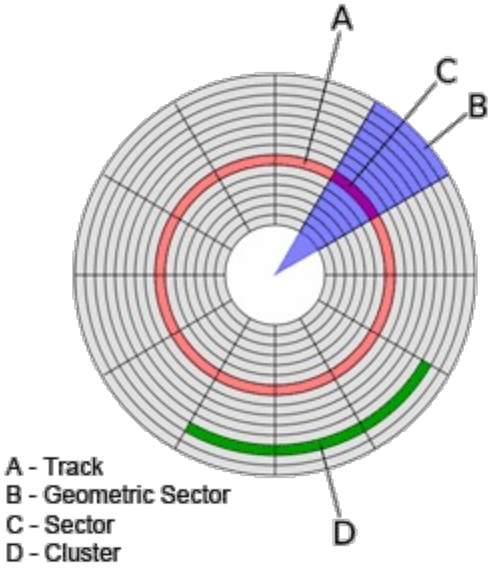
Delete 8



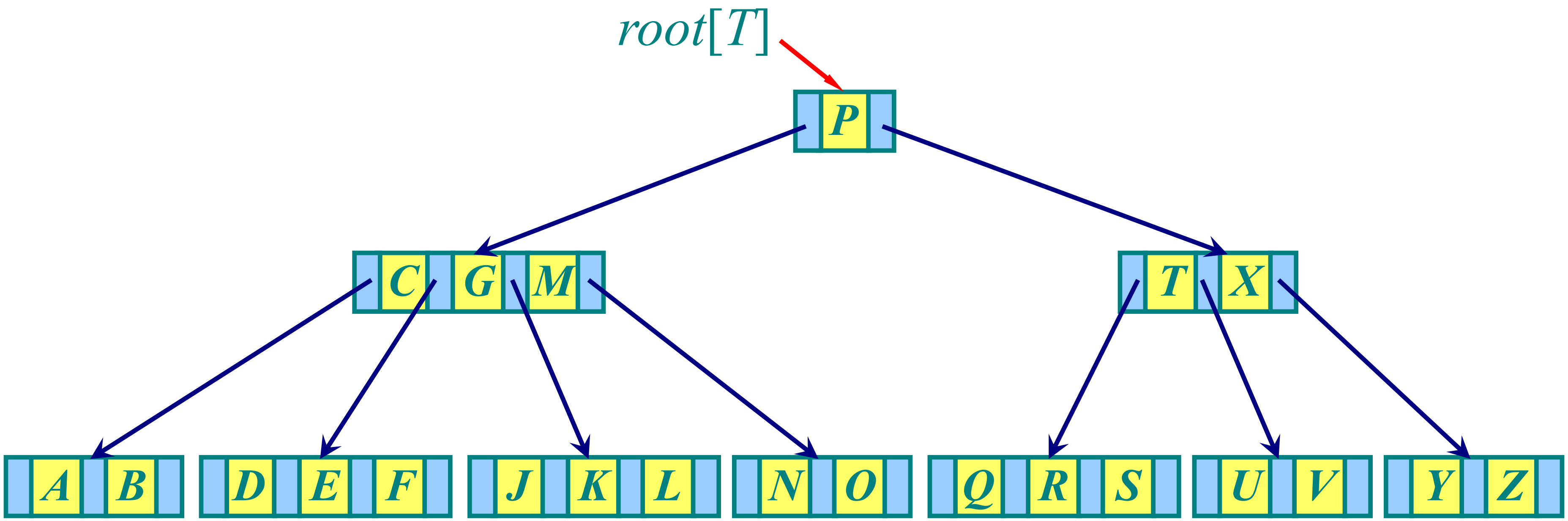
Case 4R: *x's sibling w is black, and w's left child is red.*

Typical disk drive



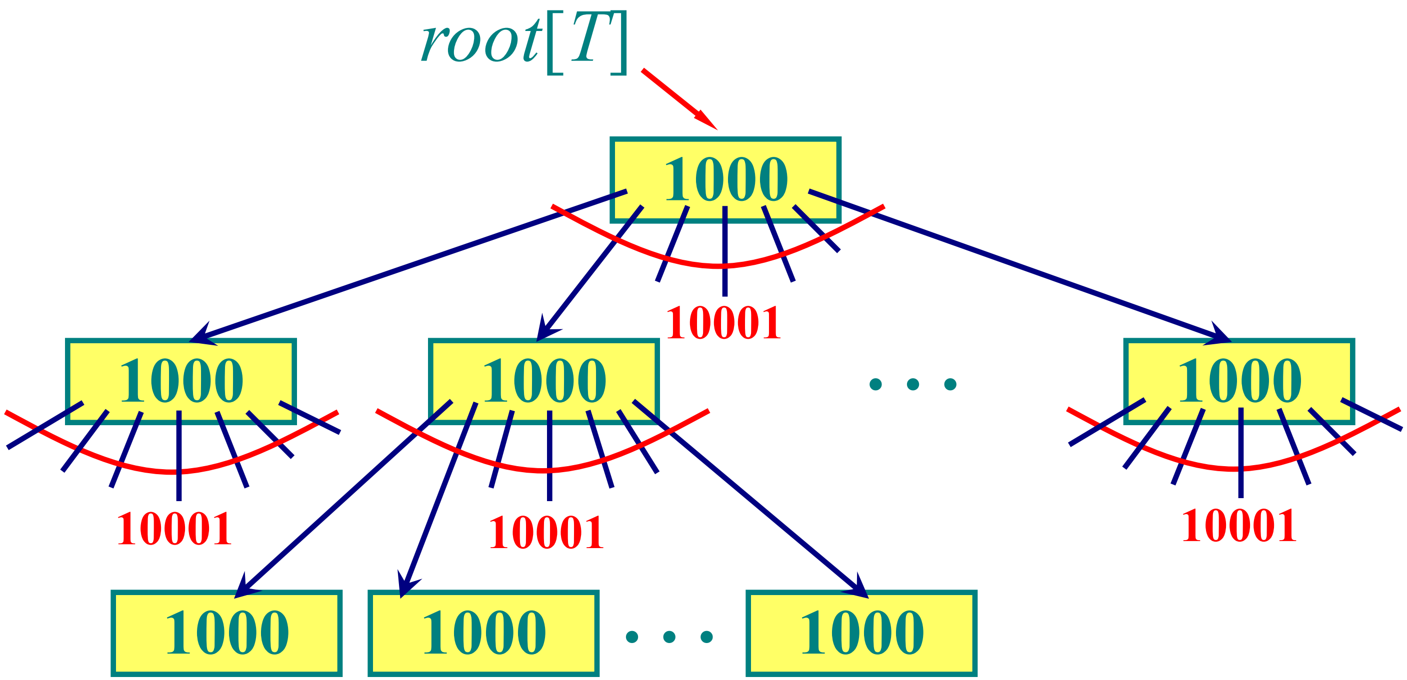


B-tree



The ***minimum degree*** for this **B-tree** is $t = 2$, every node other than the root must have at least 1 keys and every node can contain at most 3 keys (**2-3-4 tree**)

B-tree (1000 keys)



Each internal node and leaf contains 1000 keys.

Definition of B-trees

A ***B-tree*** T haves the following properties:

1. Every node x has the following fields:

- $n[x]$, the number of keys currently stored in node x;
- the $n[x]$ keys themselves, stored in nondecreasing order, so that $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$;
- $leaf[x]$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains $n[x] + 1$ has the pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

Definition of B-trees

3. The keys $key_i[x]$ separate ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}$$

4. All leaves have the same depth, which is the tree's height h.

Definition of B-trees

5. There are lower and upper bounds on the number of key a node can contain ($t \geq 2$, ***minimum degree***)

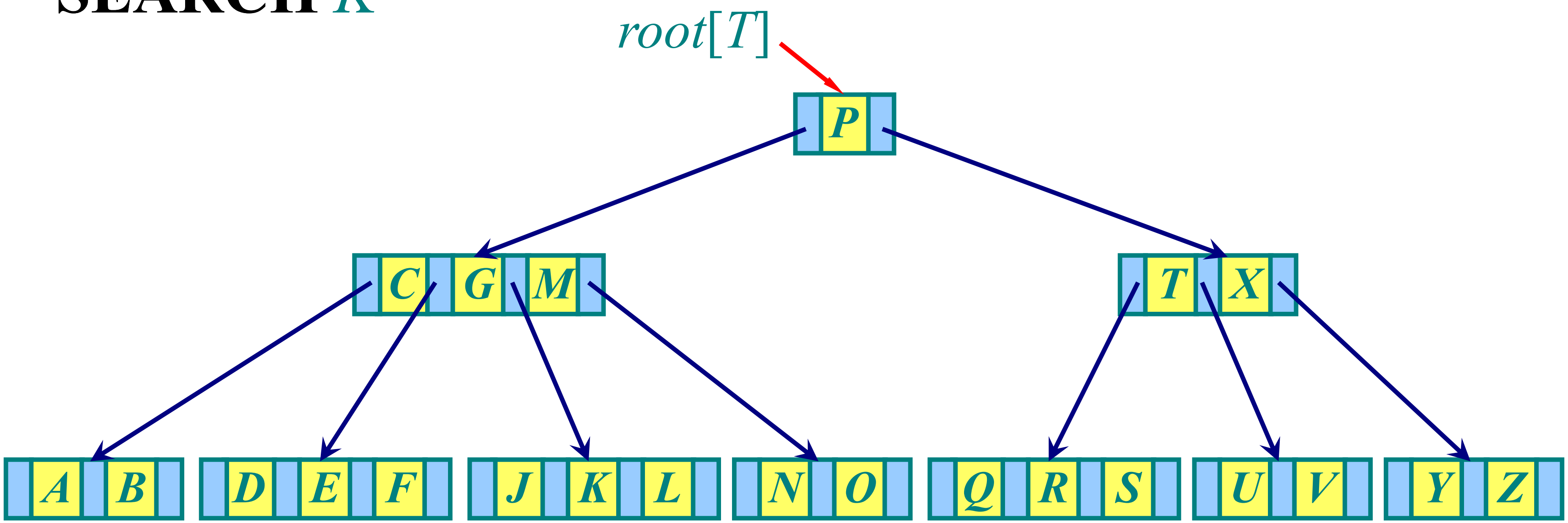
- Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children;
- Every node can contain at most $2t - 1$ keys. An internal node can have at most $2t$ children.

Theorem. If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq log_t n.$$

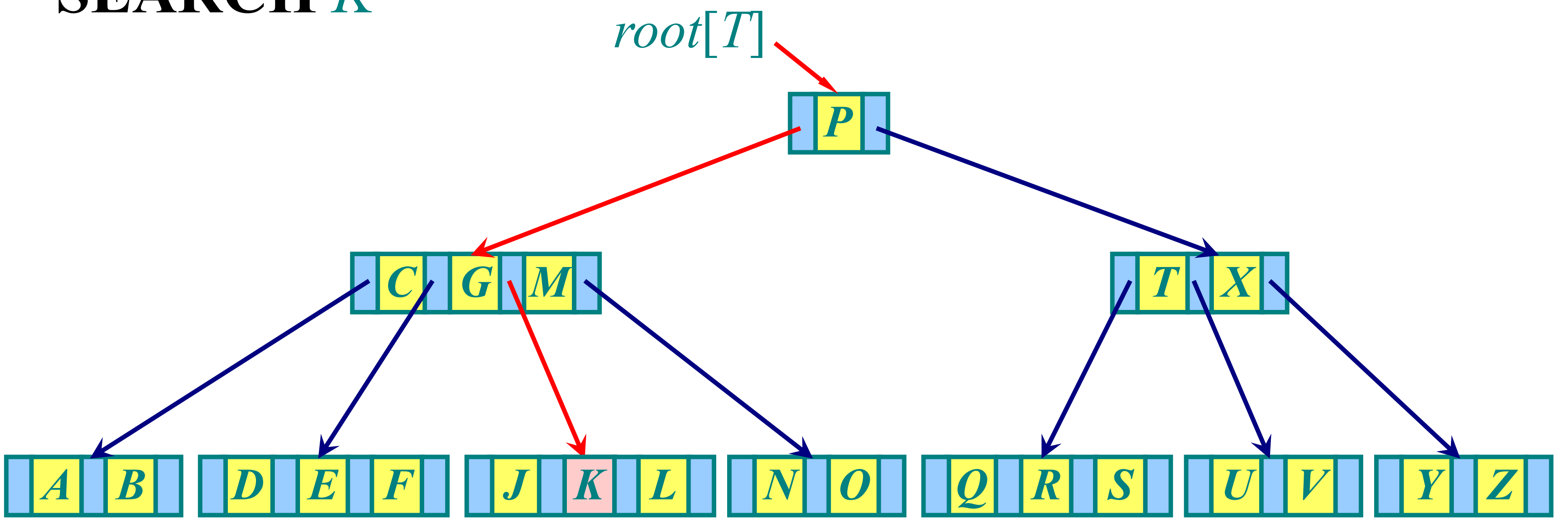
Searching a B-tree

SEARCH *K*



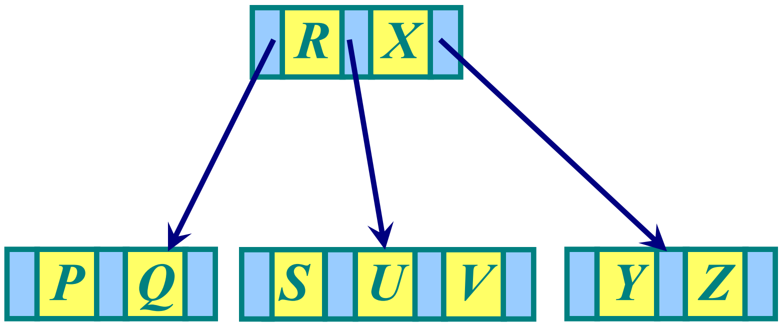
Searching a B-tree

SEARCH *K*



Splitting (B-tree)

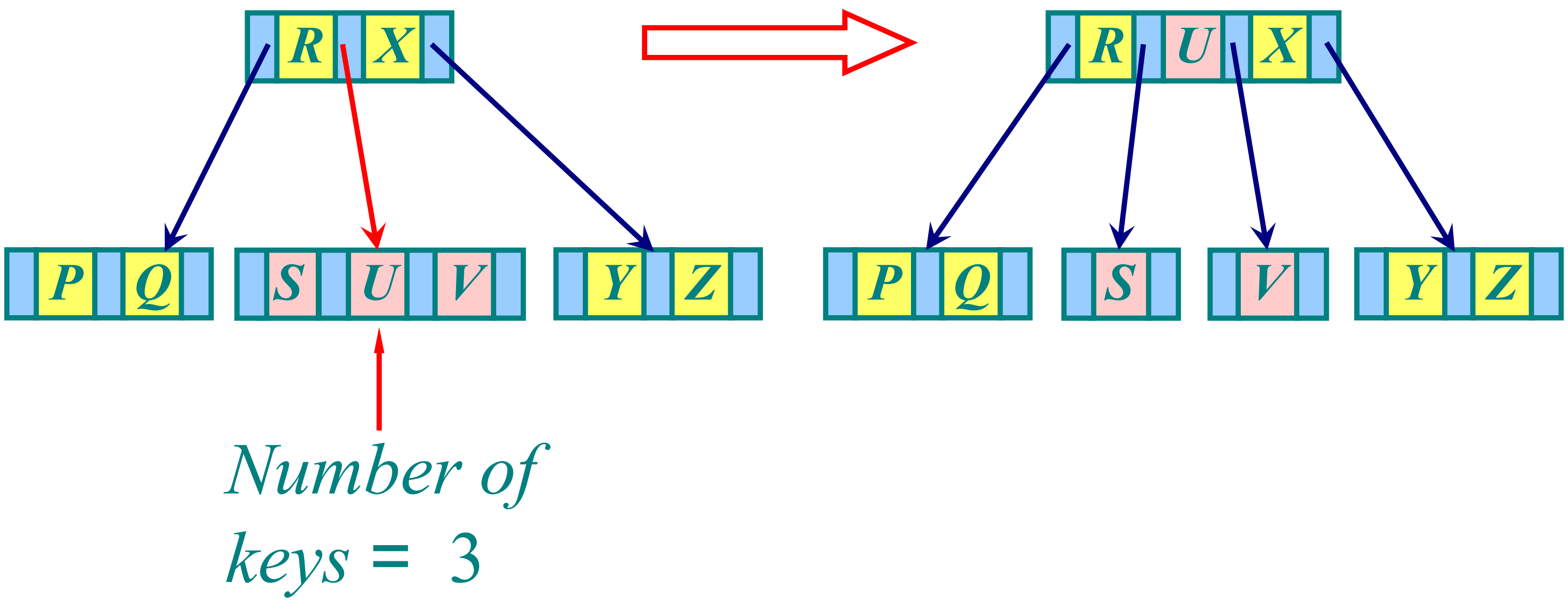
Try to INSERT *T*



Minimum degree $t = 2$

Splitting (B-tree)

Try to INSERT *T*



Minimum degree $t = 2$

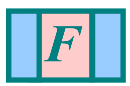
Insertion (B-tree)

INSERT *F*, *S*, *Q*, *K*, *C*, *L*, *H*, *T*, *V*, *W*, *M*, *R*, *N*, *P*, *A*, *B*, *X*, *Y*, *D*, *Z*, *E*.

Minimum degree $t = 2$

Insertion (B-tree)

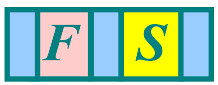
INSERT *F*, *S*, *Q*, *K*, *C*, *L*, *H*, *T*, *V*, *W*, *M*, *R*, *N*, *P*, *A*, *B*, *X*, *Y*, *D*, *Z*, *E*.



Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F*, *S*, *Q*, *K*, *C*, *L*, *H*, *T*, *V*, *W*, *M*, *R*, *N*, *P*, *A*, *B*, *X*, *Y*, *D*, *Z*, *E*.



Minimum degree $t = 2$

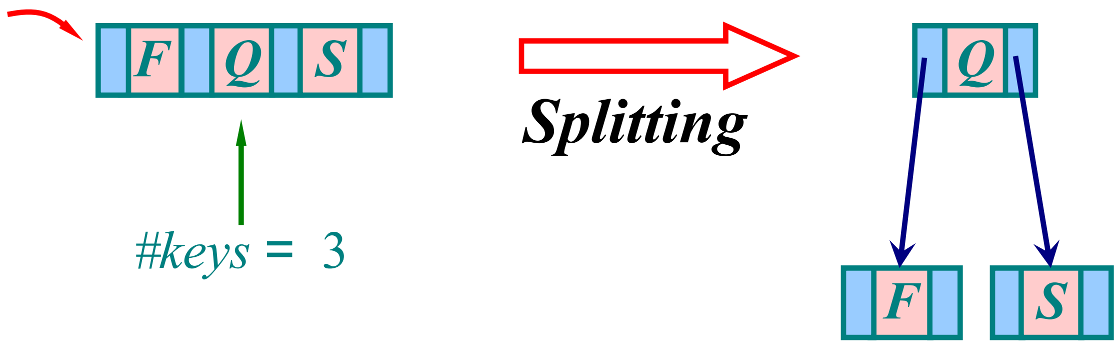
Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

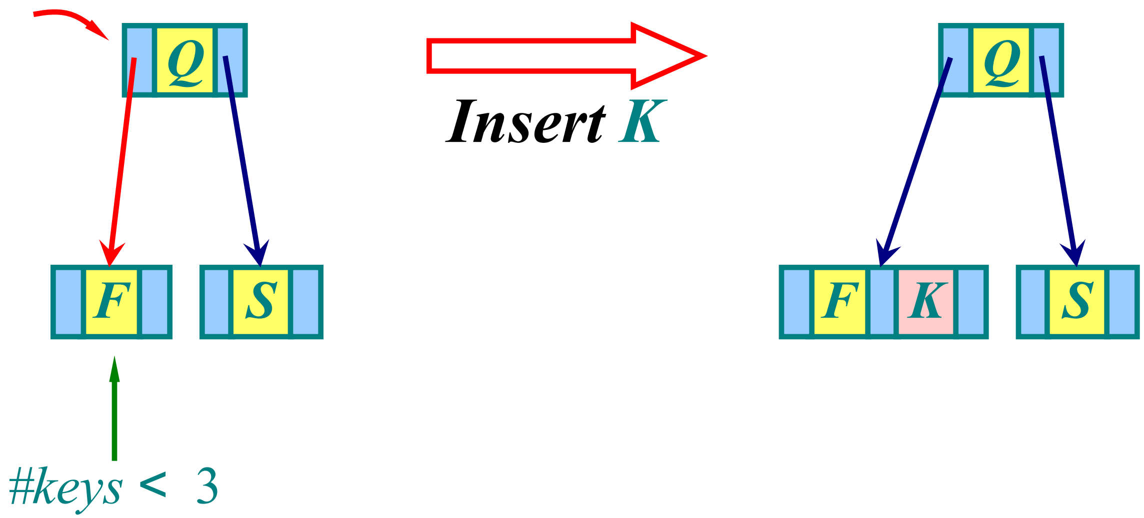


Case 1: *current node is root and has* 3 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F*, *S*, *Q*, *K*, *C*, *L*, *H*, *T*, *V*, *W*, *M*, *R*, *N*, *P*, *A*, *B*, *X*, *Y*, *D*, *Z*, *E*.

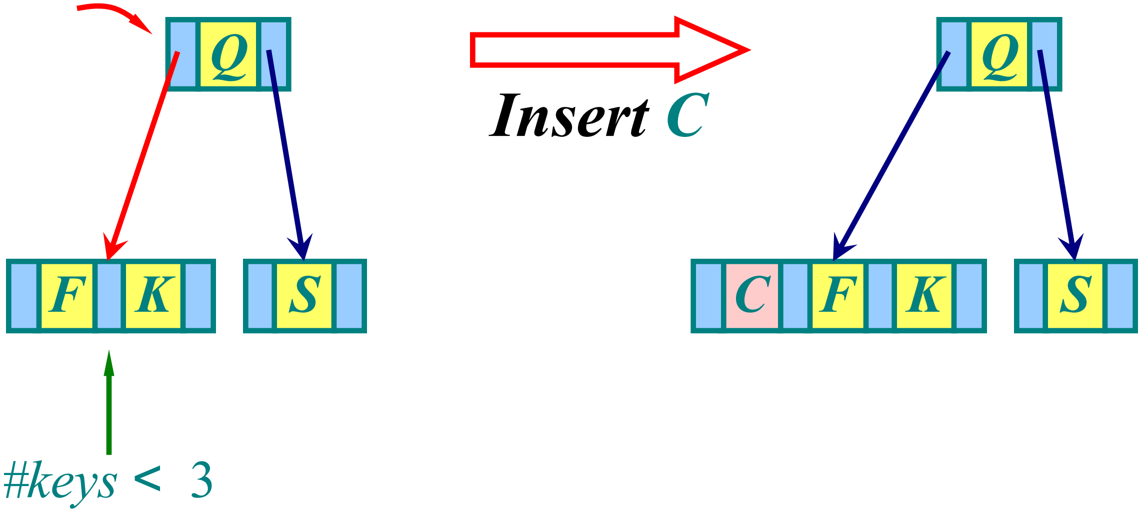


Case 2: *current node has at most* 2 *keys and the appropriate subtree has at most* 2 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

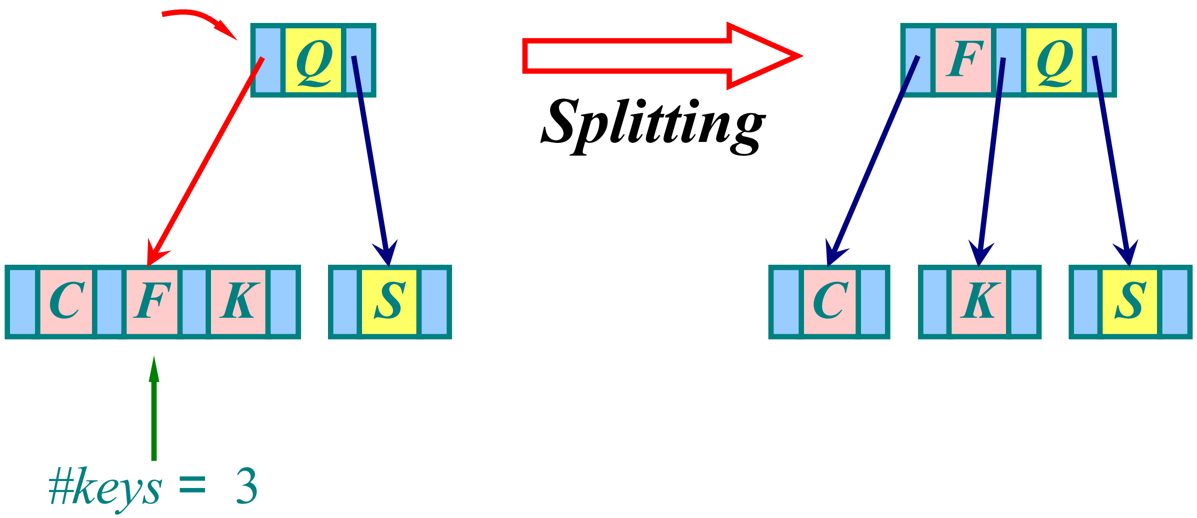


Case 2: *current node has at most* 2 *keys and the appropriate subtree has at most* 2 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

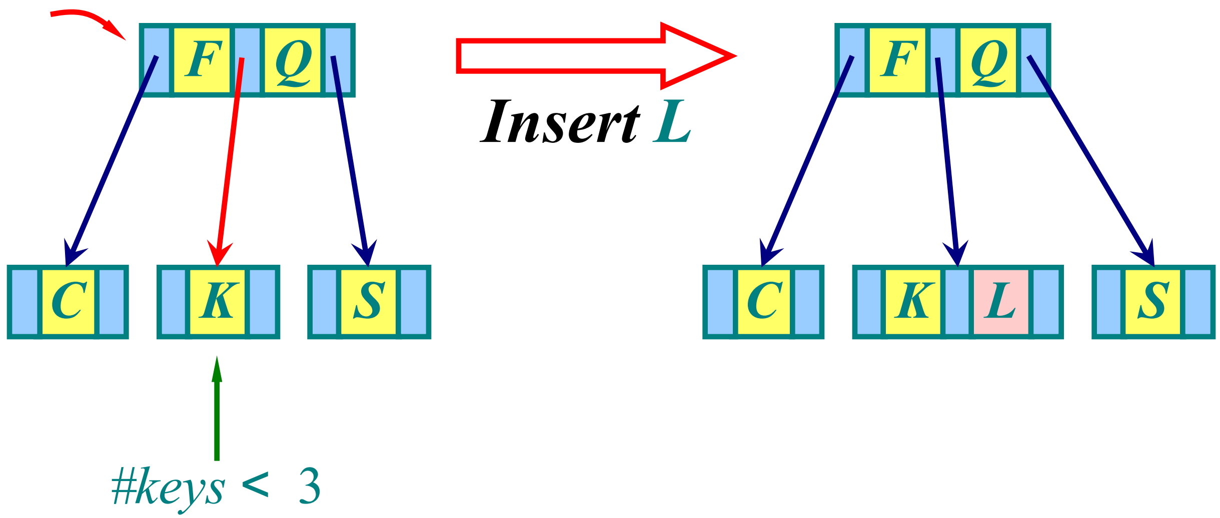


Case 3: *current node has at most* 2 *keys and the appropriate subtree has* 3 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

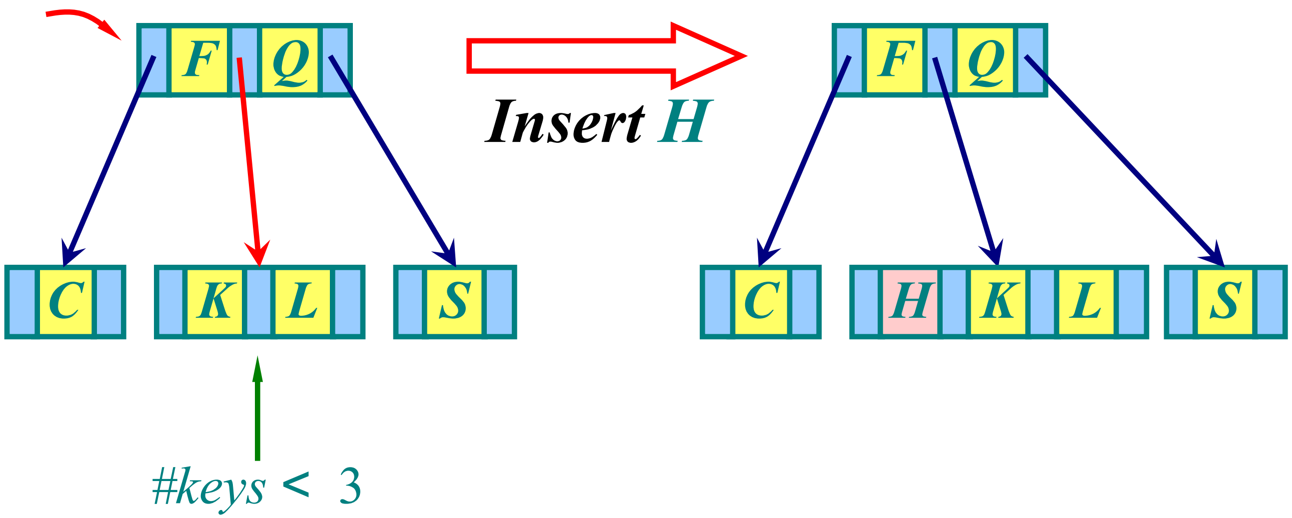


Case 4: *the appropriate subtree has at most* 2 *keys* (*after case* 3).

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

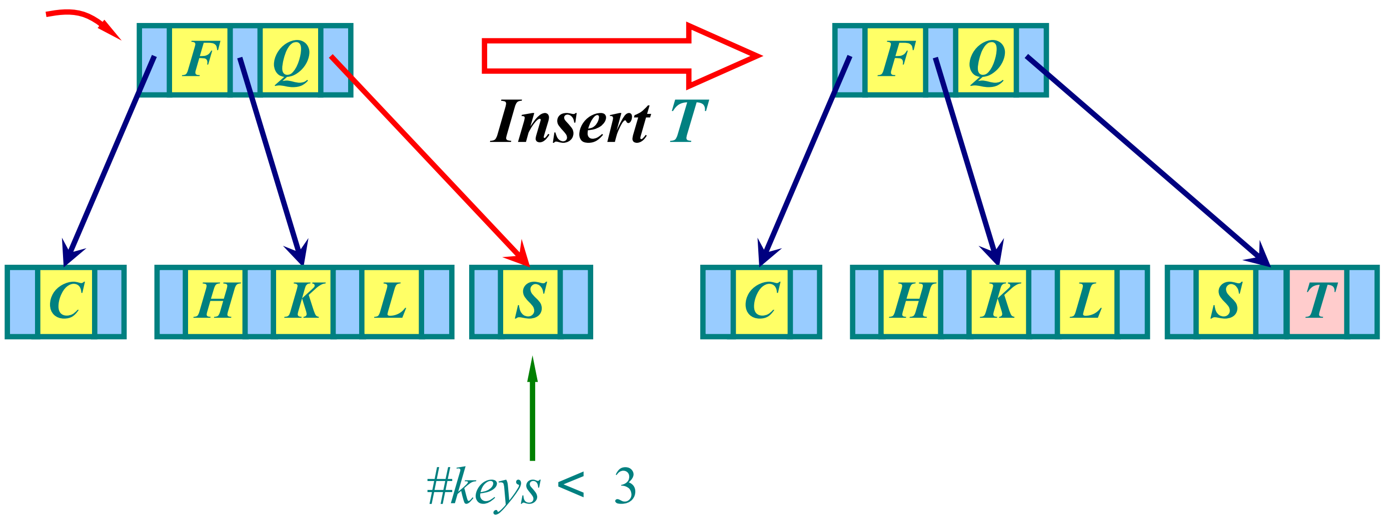


Case 2: *current node has at most* 2 *keys and the appropriate subtree has at most* 2 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

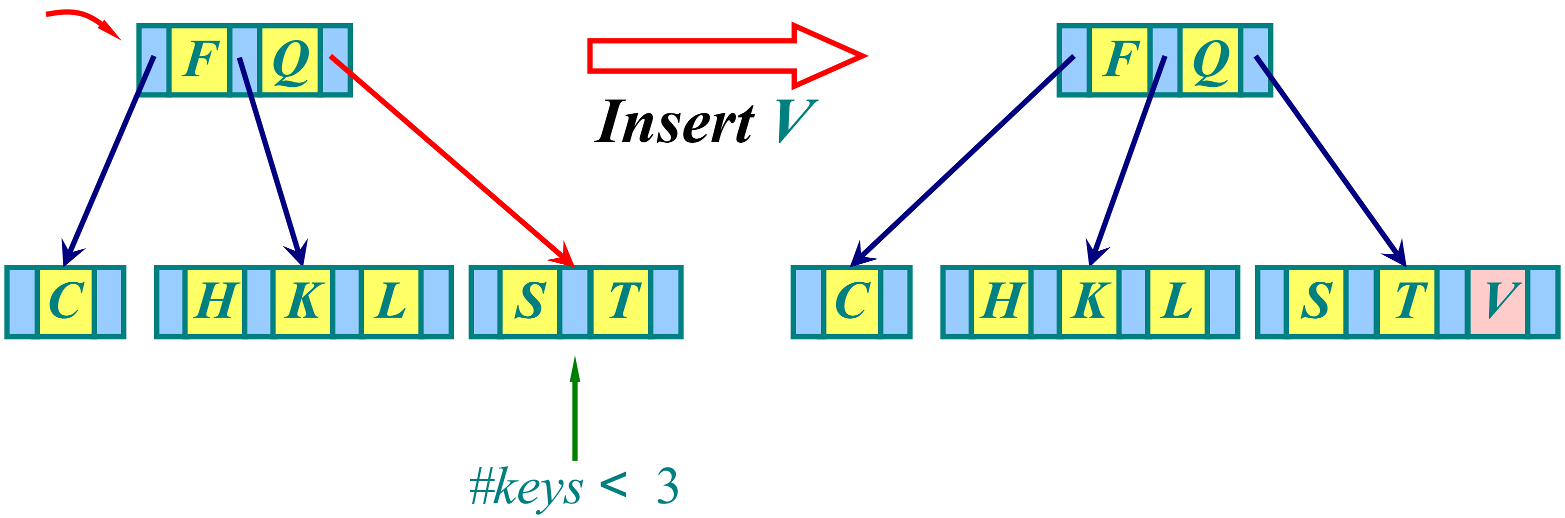


Case 2: *current node has at most* 2 *keys and the appropriate subtree has at most* 2 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

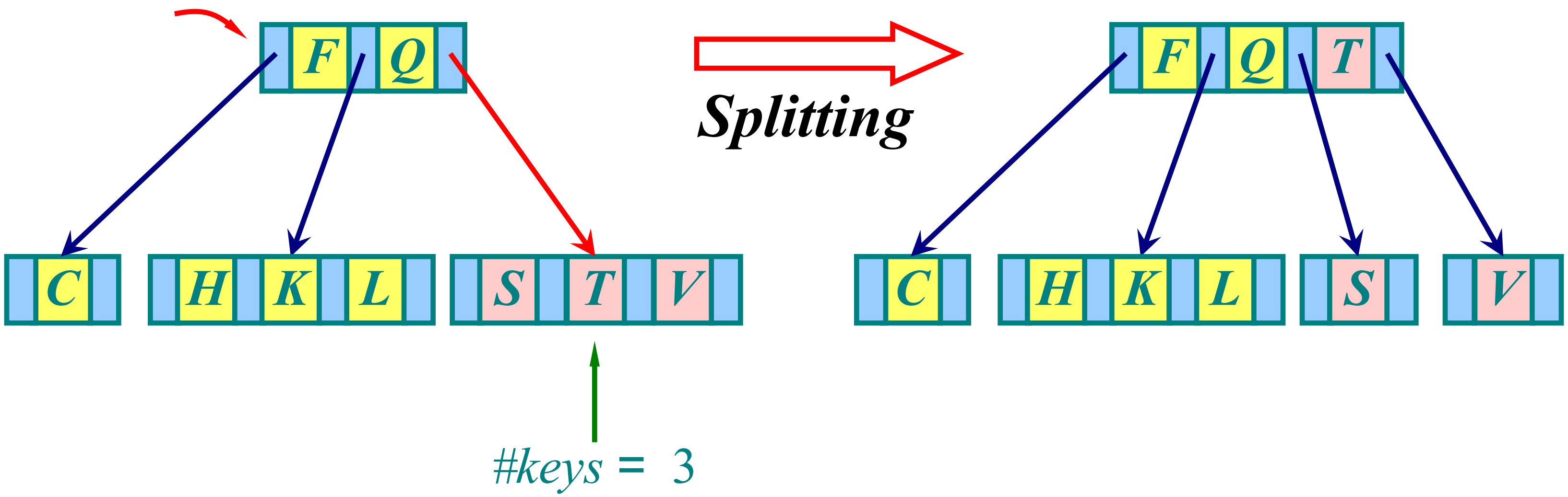


Case 2: *current node has at most* 2 *keys and the appropriate subtree has at most* 2 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

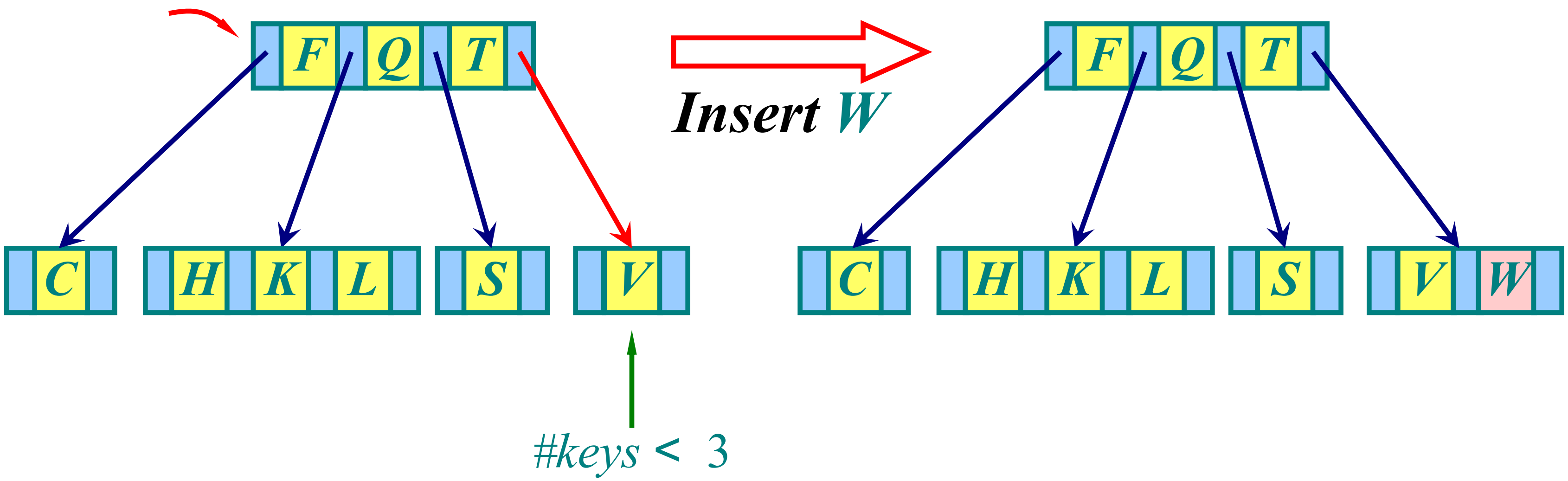


Case 3: *current node has at most* 2 *keys and the appropriate subtree has* 3 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

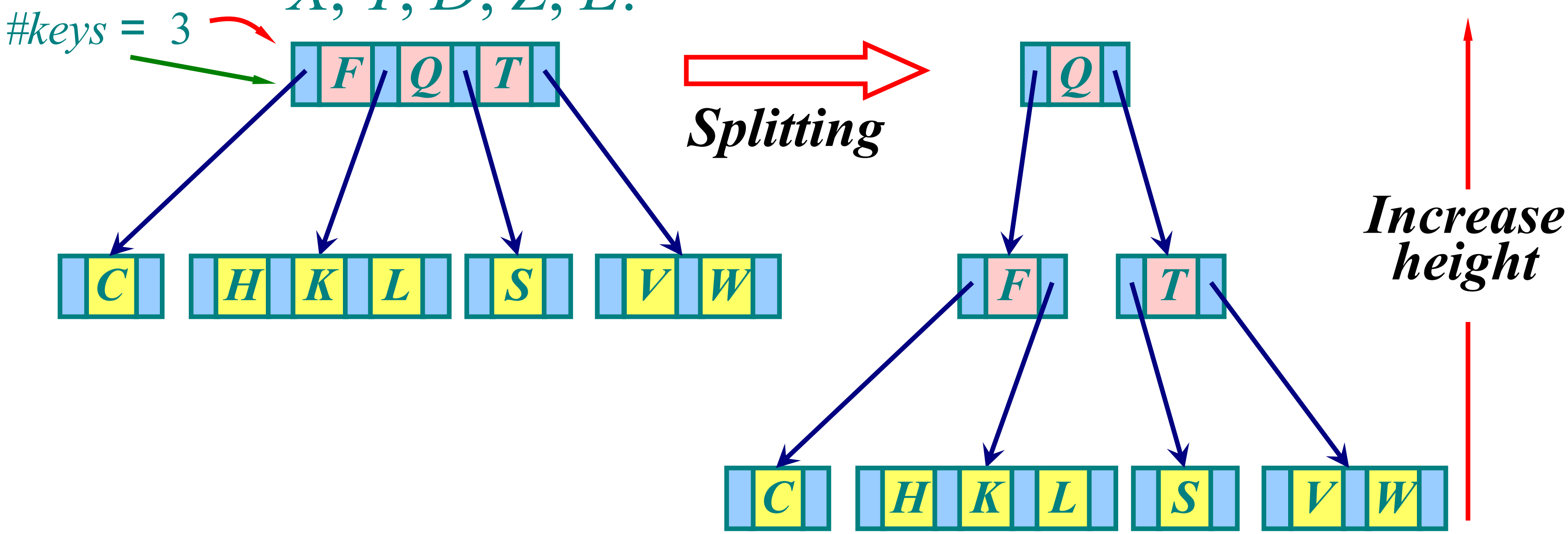


Case 4: *the appropriate subtree has at most* 2 *keys* (*after case* 3).

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F*, *S*, *Q*, *K*, *C*, *L*, *H*, *T*, *V*, *W*, *M*, *R*, *N*, *P*, *A*, *B*, *X*, *Y*, *D*, *Z*, *E*.

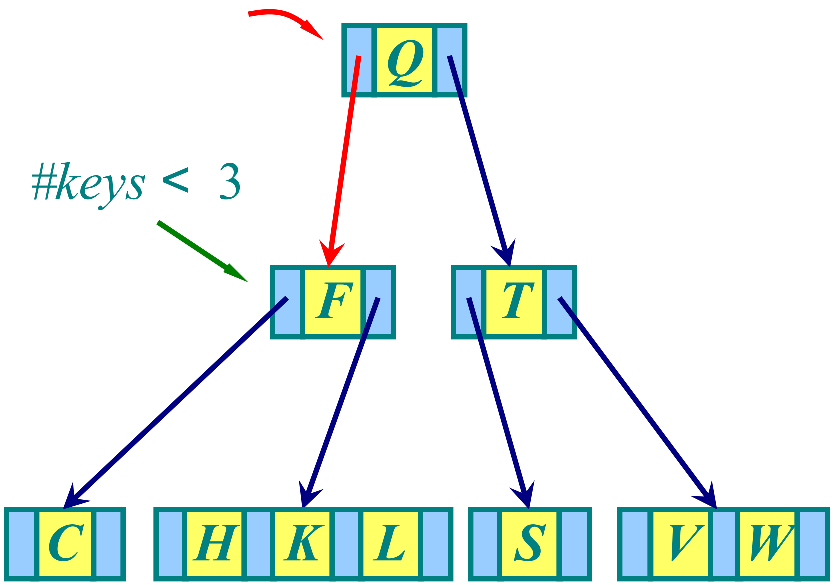


Case 1: *current node is root and has* 3 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

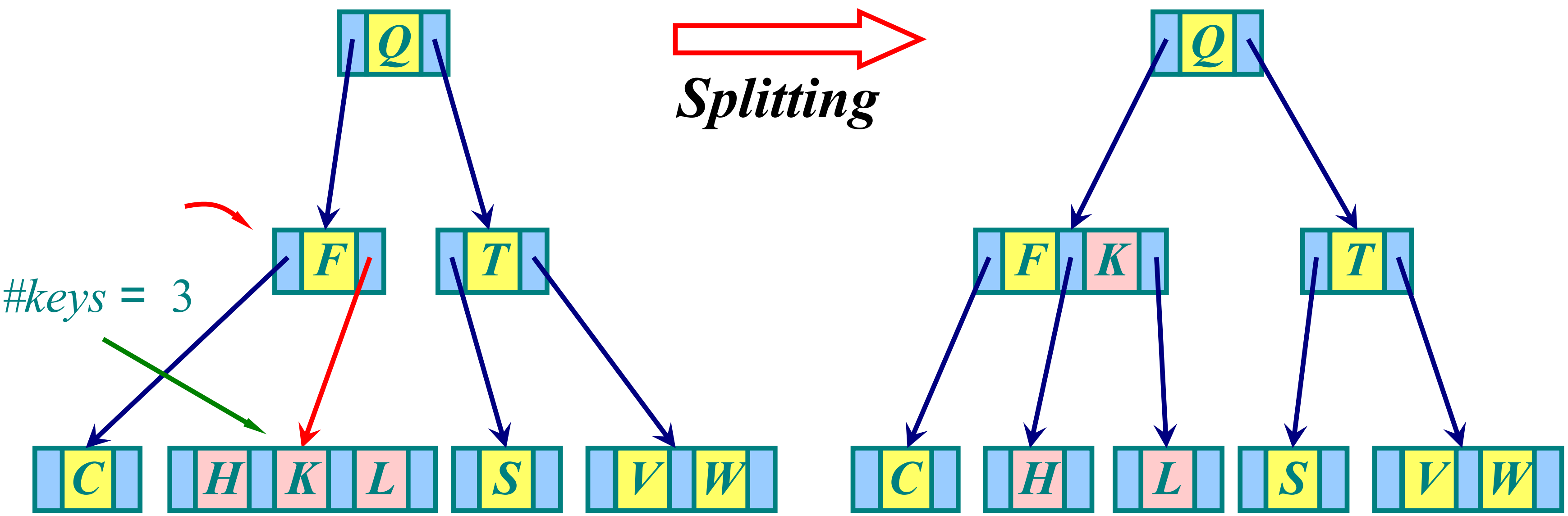
INSERT *F*, *S*, *Q*, *K*, *C*, *L*, *H*, *T*, *V*, *W*, *M*, *R*, *N*, *P*, *A*, *B*, *X*, *Y*, *D*, *Z*, *E*.



Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

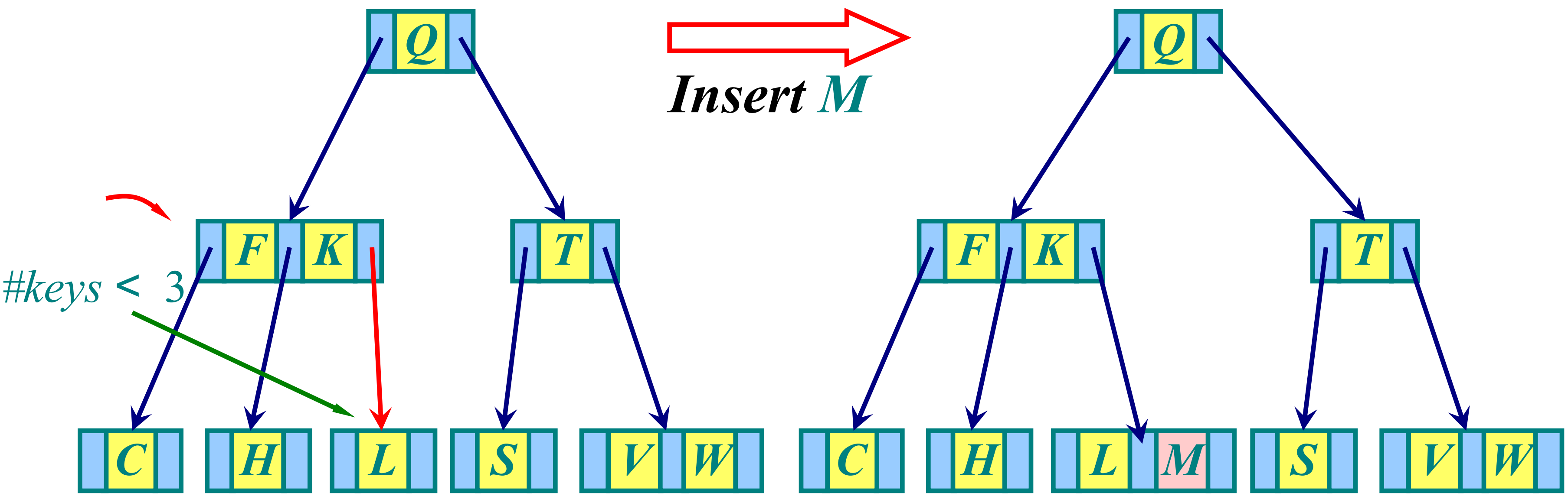


Case 3: *current node has at most* 2 *keys and the appropriate subtree has* 3 *keys.*

Minimum degree $t = 2$

Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*

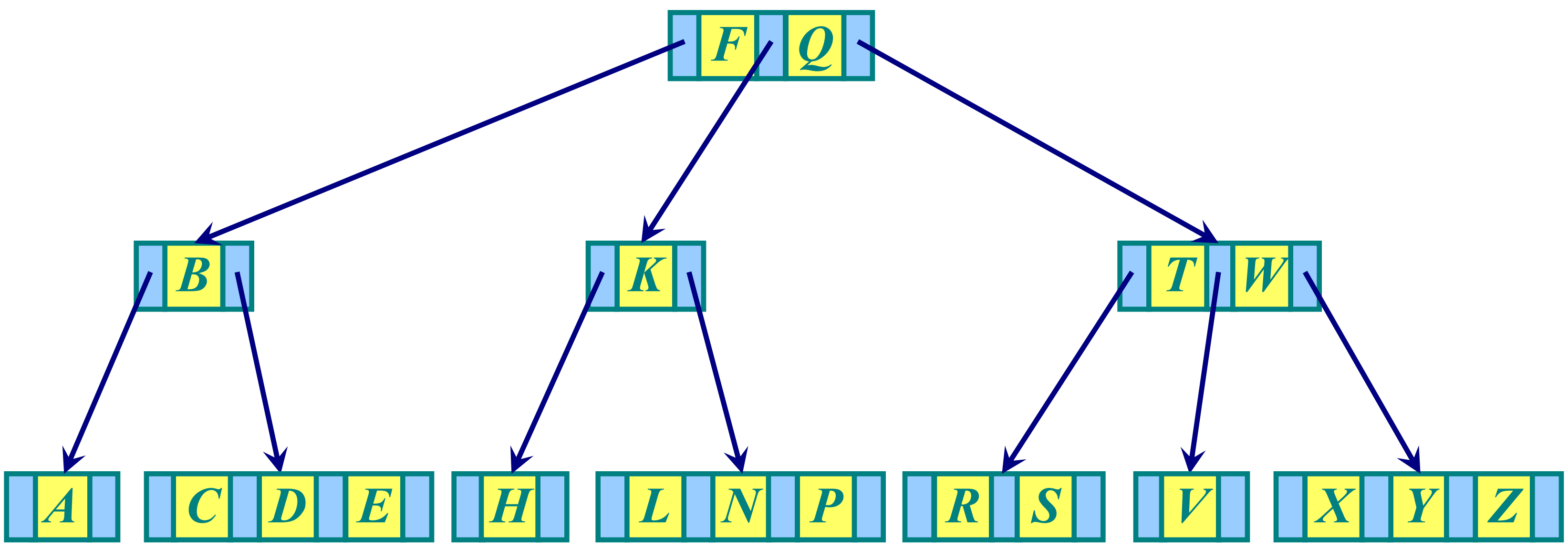


Case 4: *the appropriate subtree has at most* 2 *keys* (*after case* 3).

Minimum degree $t = 2$

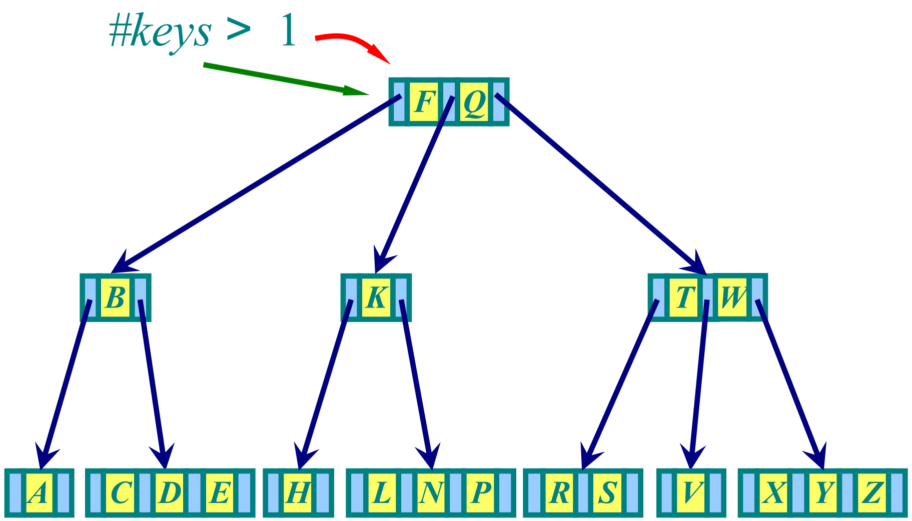
Insertion (B-tree)

INSERT *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.*



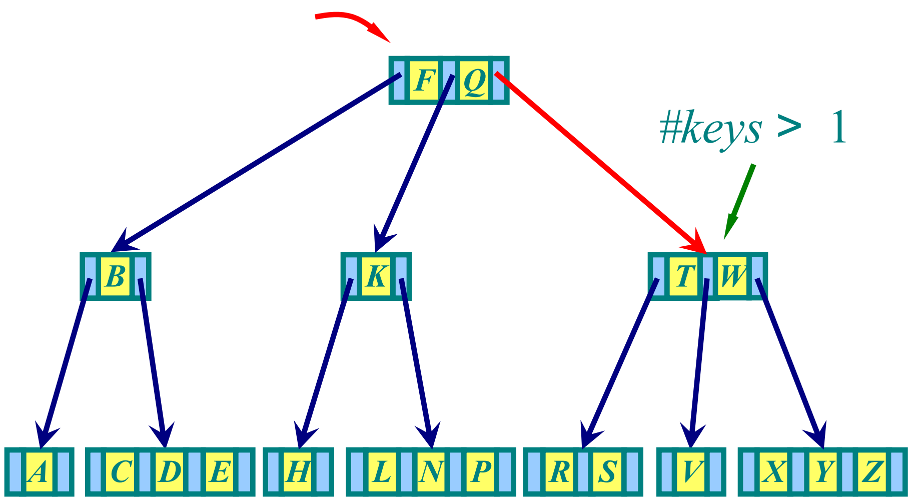
Deletion (B-tree)

DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



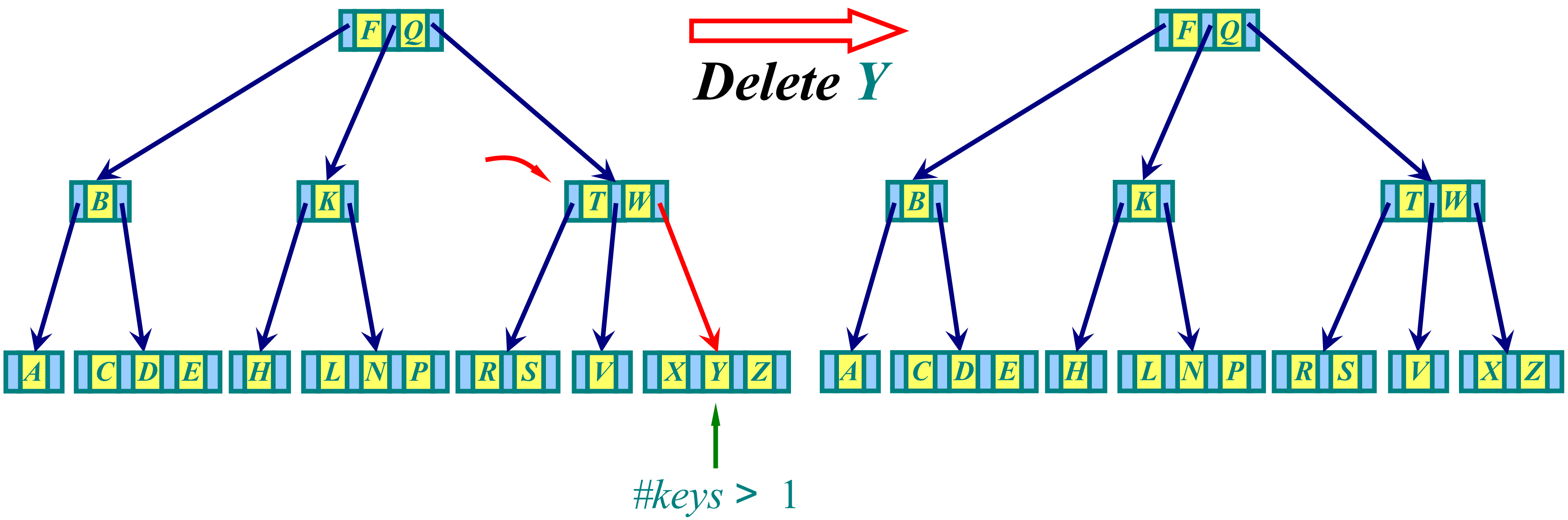
Deletion (B-tree)

DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Deletion (B-tree)

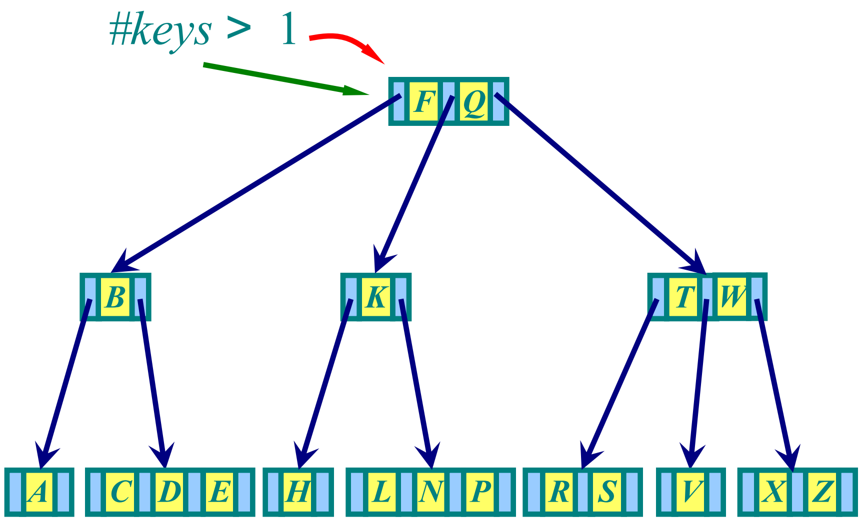
DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Case 1: *key* $k = Y$ *is in a leaf.*

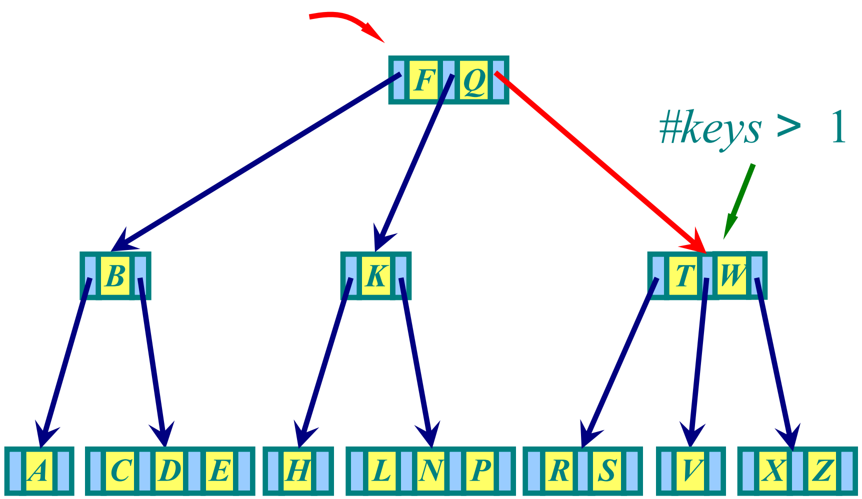
Deletion (B-tree)

DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



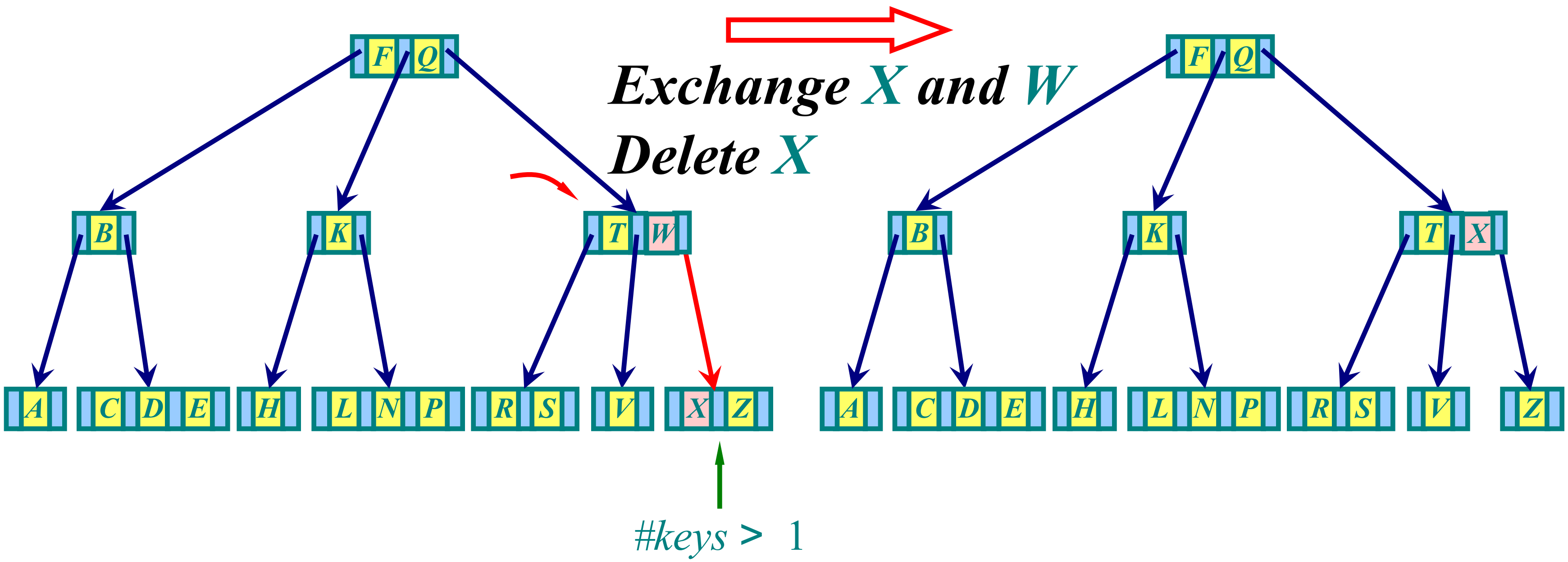
Deletion (B-tree)

DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Deletion (B-tree)

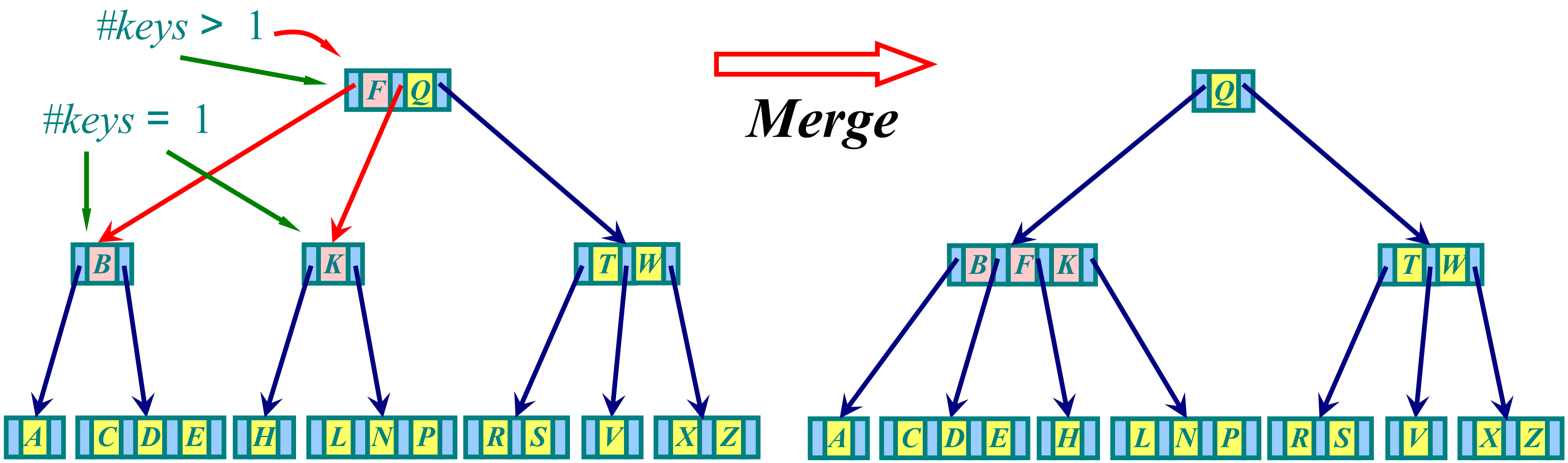
DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Case 2-*a*: *key* $k = W$ *is in a internal node and one of its children that precedes or* ***follows*** k *has at least* 2 *keys.*

Deletion (B-tree)

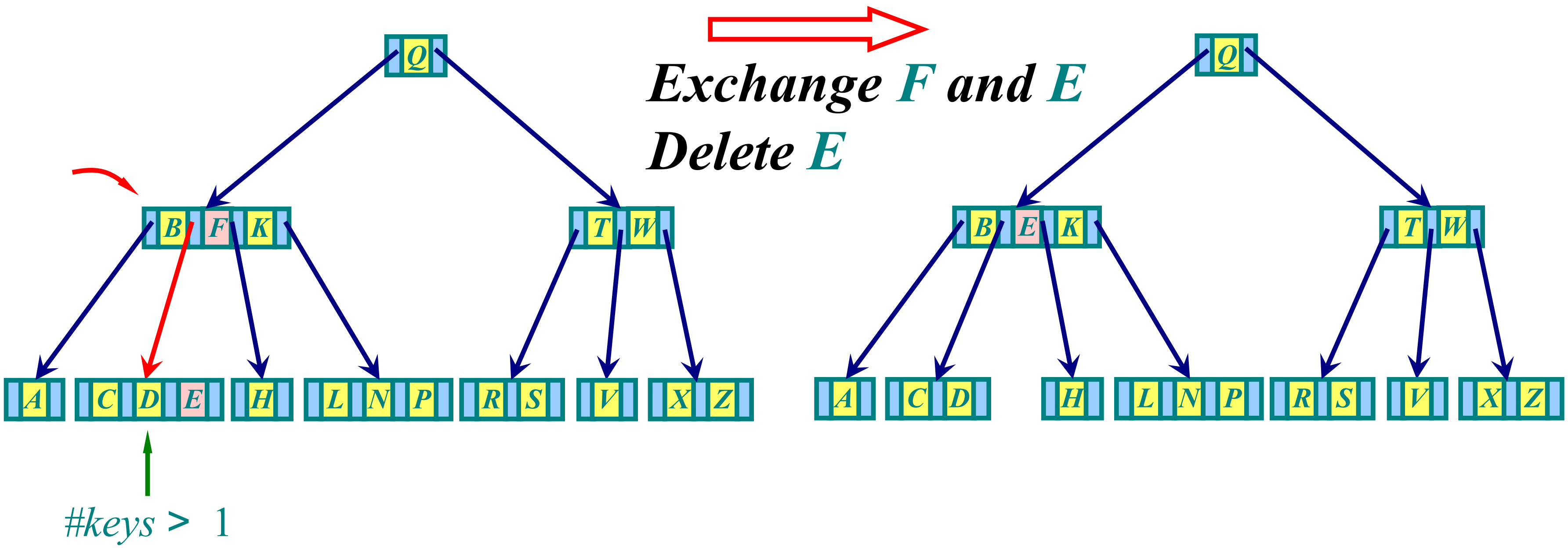
DELETE *F* other than *W* **Minimum degree** $t = 2$



Case 2-*b*: *key k = F is in a internal node and the both of its children that* ***precedes*** *or* ***follows*** *k only has 1 key.*

Deletion (B-tree)

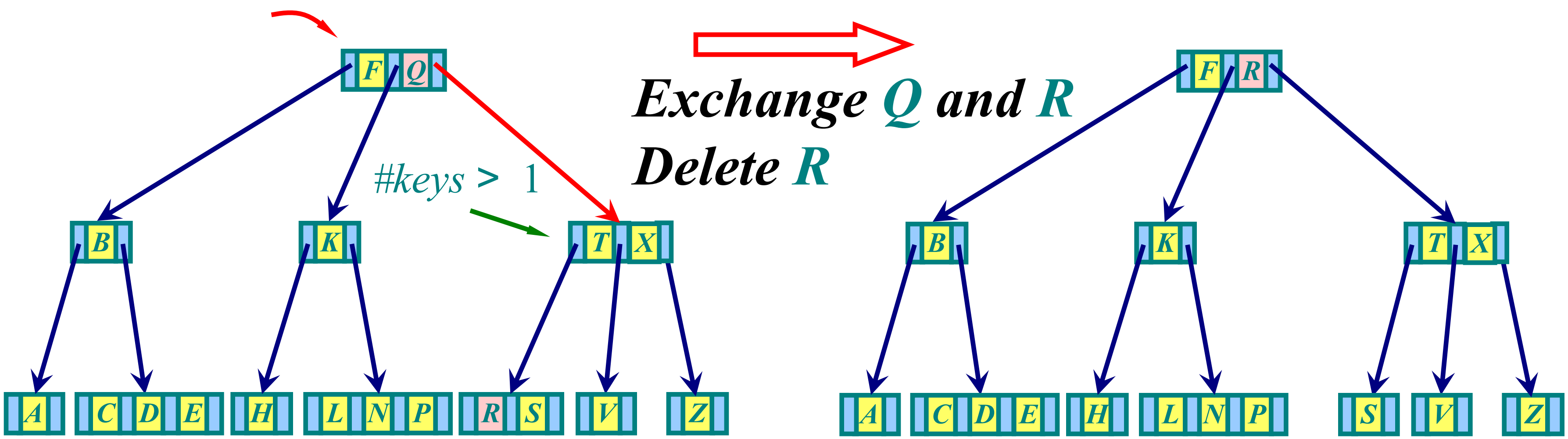
DELETE *F* other than *W* **Minimum degree** $t = 2$



Case 2-*a*: *key k = F is in a internal node and one of its children that* ***precedes*** *or follows k has at least 2 keys.*

Deletion (B-tree)

DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$

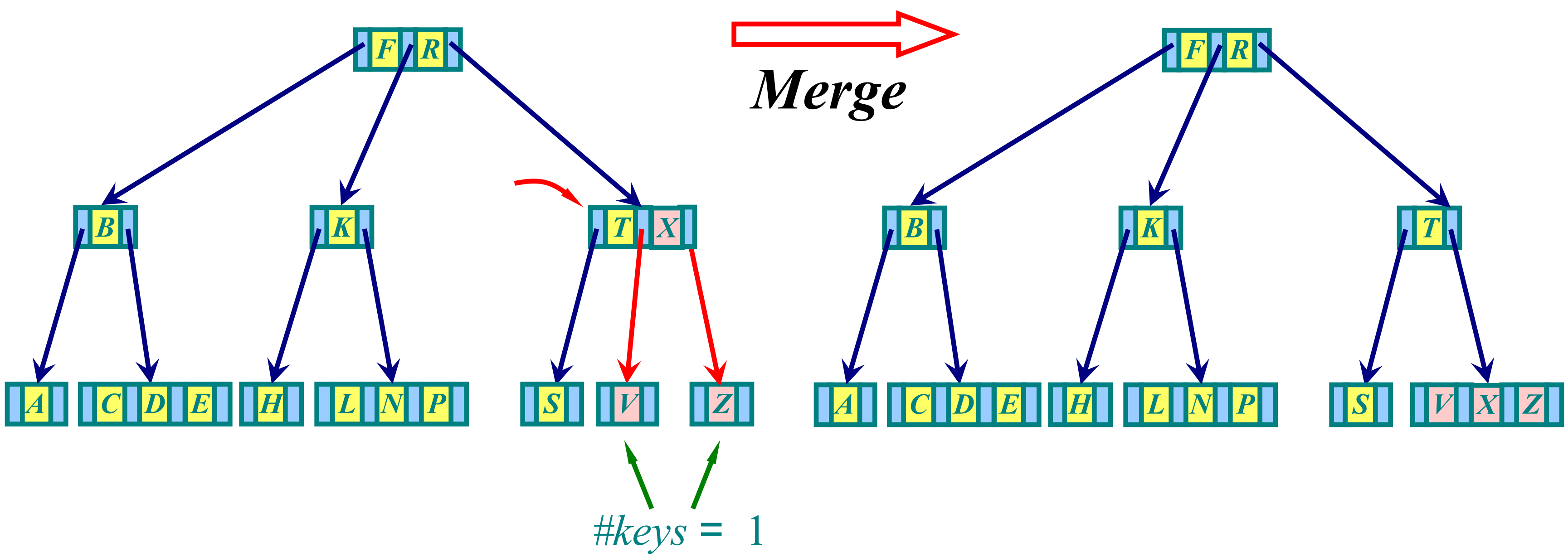


Which is Q's successor? It is R.

Case 2-*a*: *key $k = Q$ is in a internal node and one of its children that precedes or **follows** k has at least 2 keys.*

Deletion (B-tree)

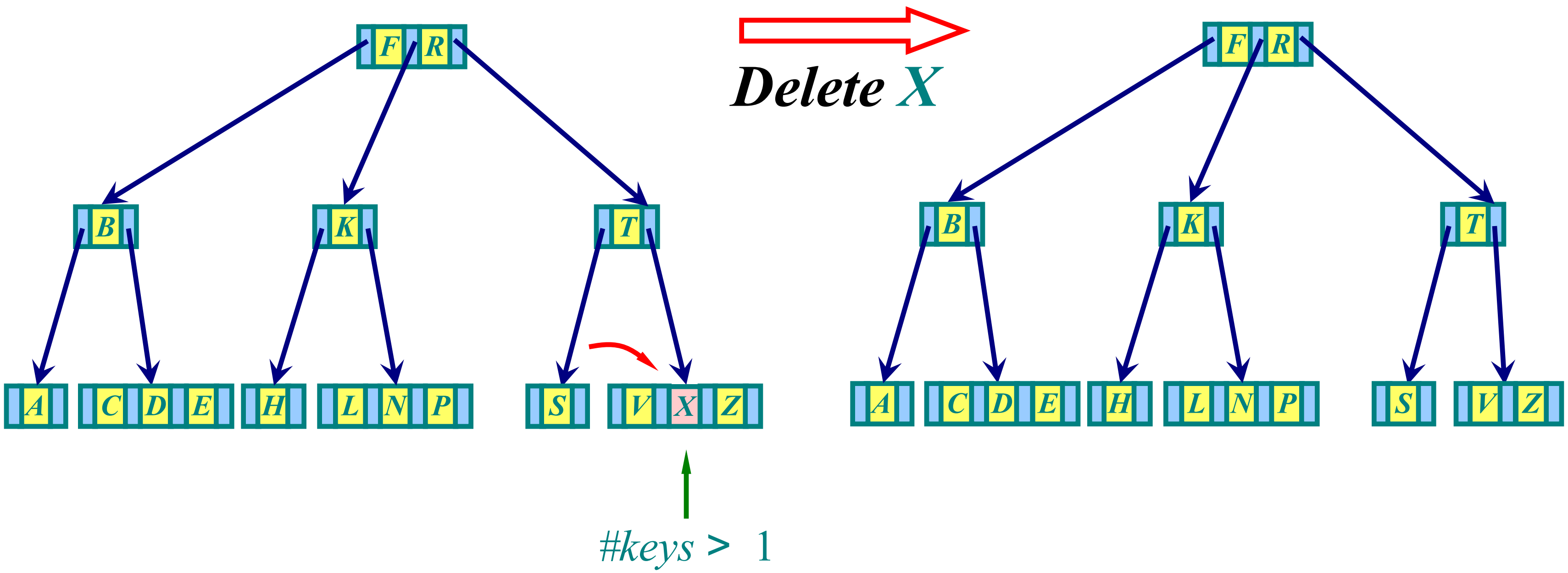
DELETE Y, W, Q, X, K, B, H, P **Minimum degree** $t = 2$



Case 2-*b*: *key* $k = X$ *is in a internal node and the both of its children that* ***precedes*** *or* ***follows*** k *only has* 1 *key.*

Deletion (B-tree)

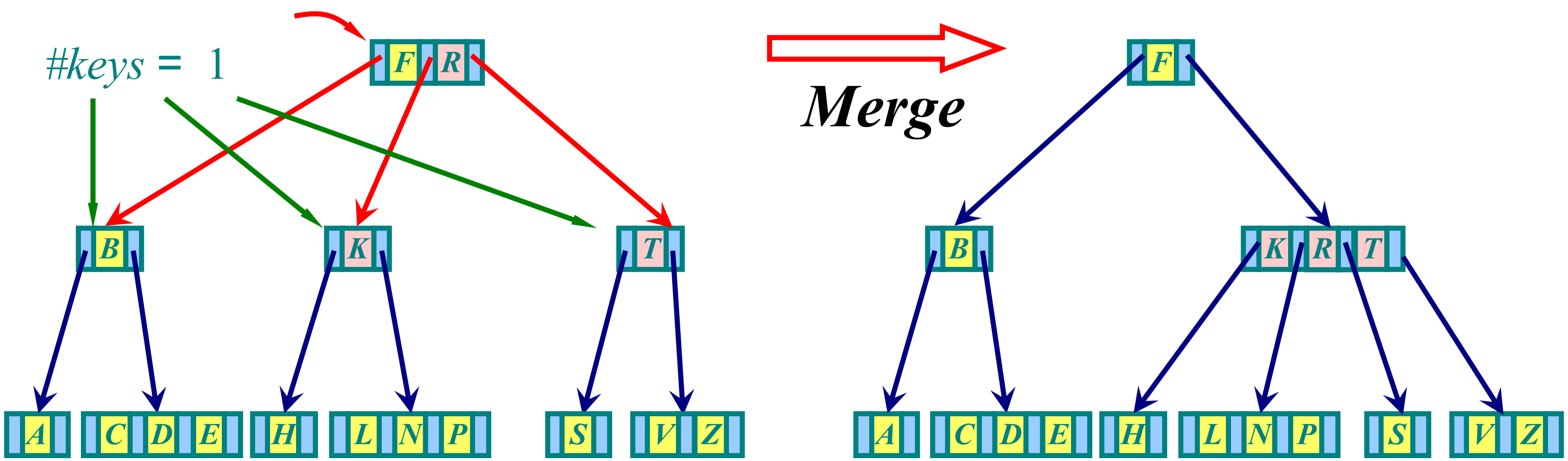
DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Current node *is* { *V*, *X*, *Z* }

Deletion (B-tree)

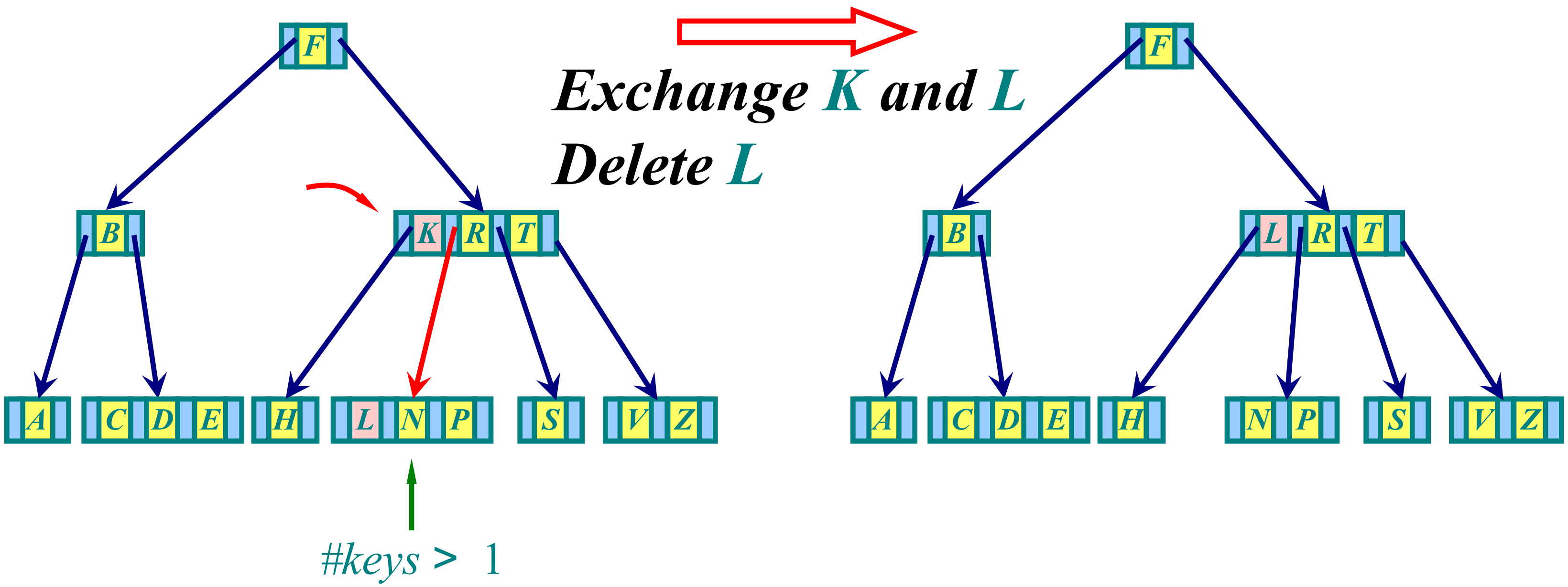
DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Case 3-*b*: *key* $k = K$ *is not present in a internal node and the appropriate subtree that must contain* k *has only* 1 *key and the subtree's immediate siblings have only* 1 *key .*

Deletion (B-tree)

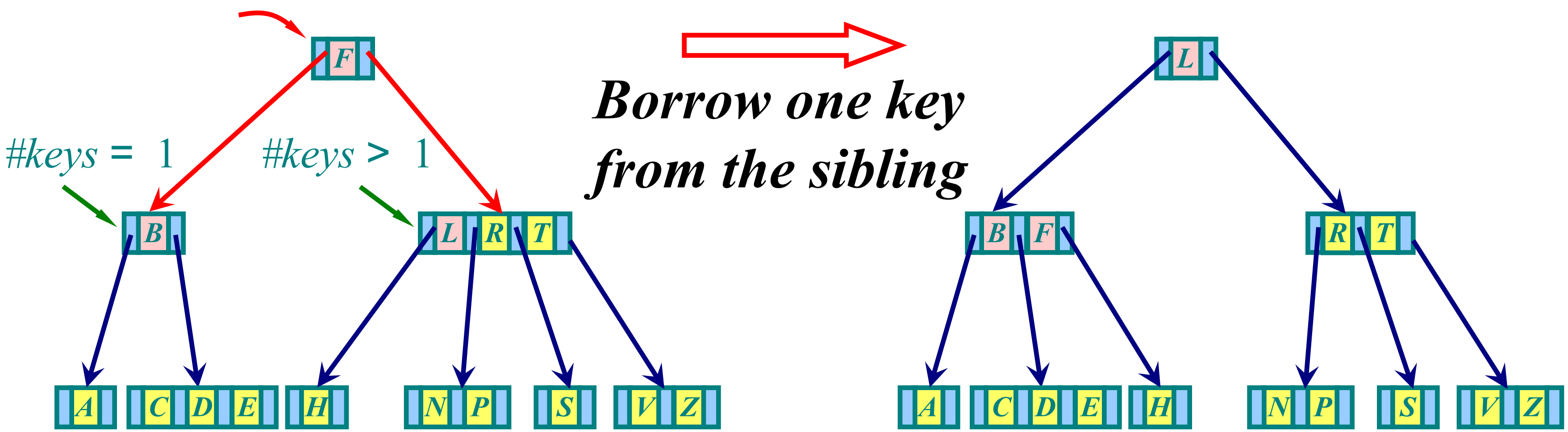
DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Case 2-*a*: *key k = K is in a internal node and one of its children that precedes or* ***follows*** *k has at least* 2 *keys.*

Deletion (B-tree)

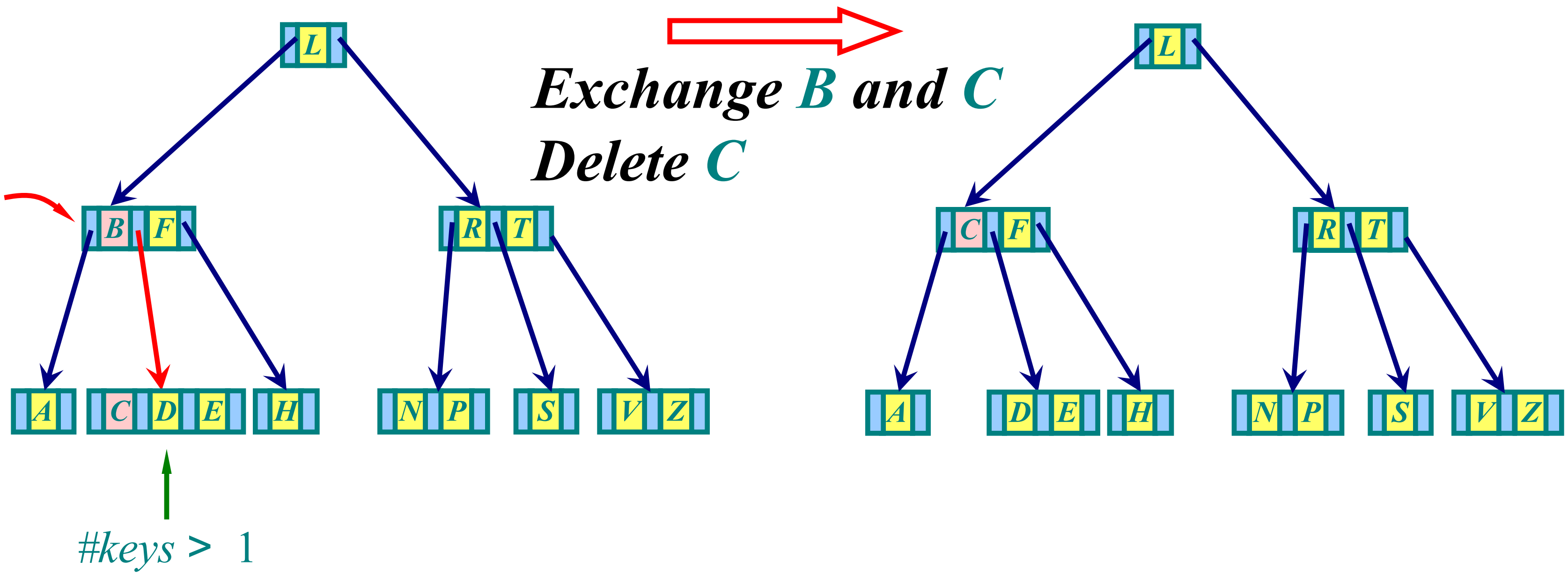
DELETE *Y, W, Q, X, K, B, H, P*   **Minimum degree** $t = 2$



Case 3-*a*: *key* $k = B$ *is not present in a internal node and the appropriate subtree that must contain* k *has only* 1 *key and one of the subtree's immediate siblings has at least* 2 *keys .*

Deletion (B-tree)

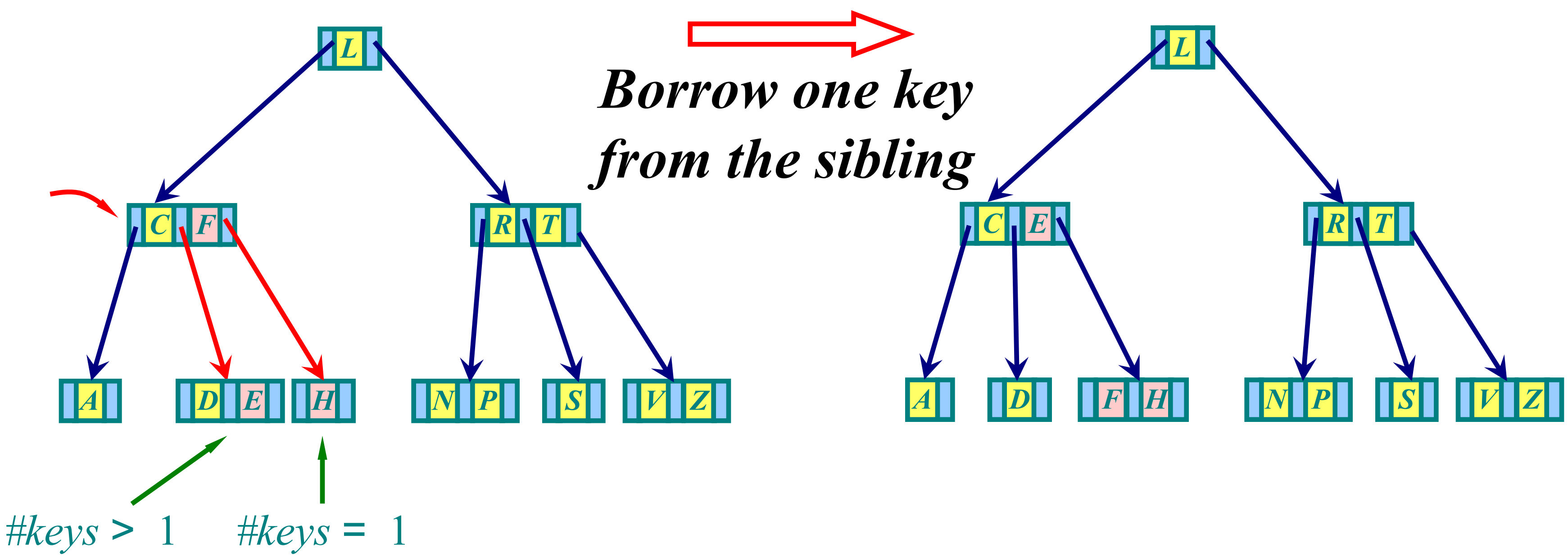
DELETE *Y, W, Q, X, K, B, H, P* **Minimum degree** $t = 2$



Case 2-*a*: *key $k = B$ is in a internal node and one of its children that precedes or **follows** k has at least 2 keys.*

Deletion (B-tree)

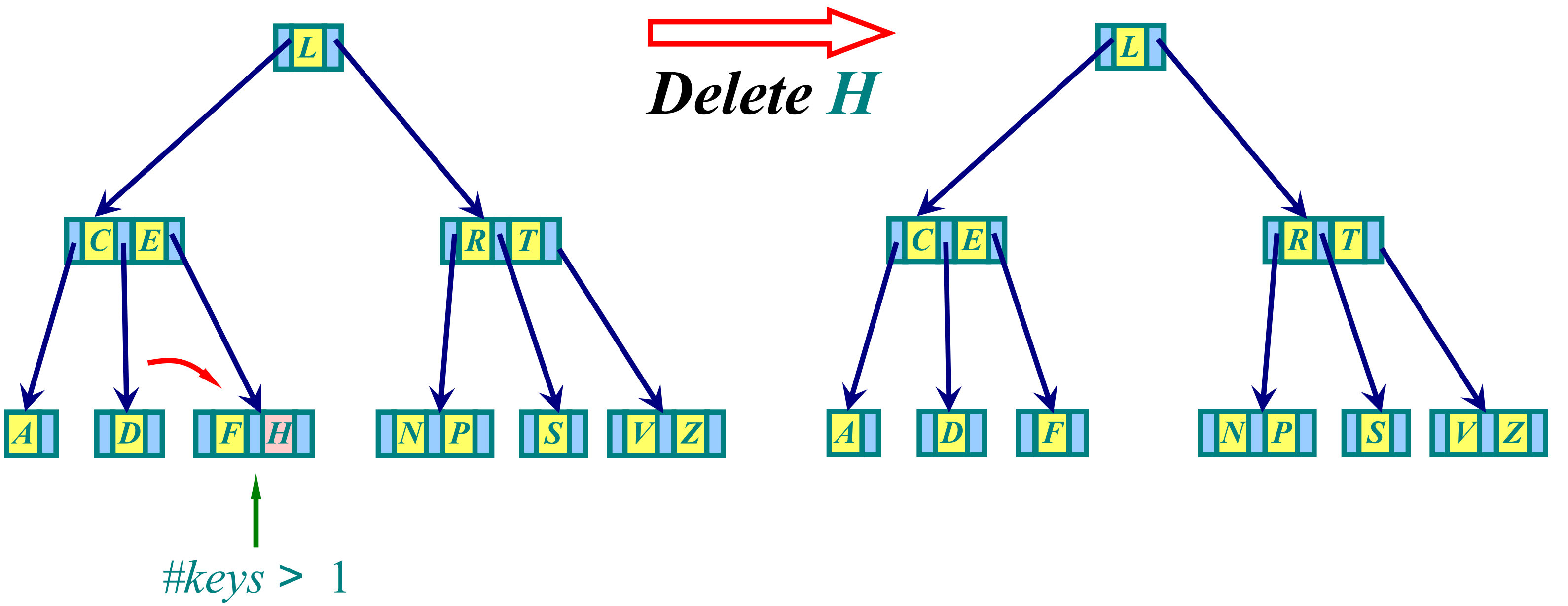
DELETE *Y, W, Q, X, K, B, H, P* **Minimum degree** $t = 2$



Case 3-*a*: *key k = H is not present in a internal node* and *the appropriate subtree that must contain k has only 1 key and one of the subtree's immediate siblings has at least 2 keys .*

Deletion (B-tree)

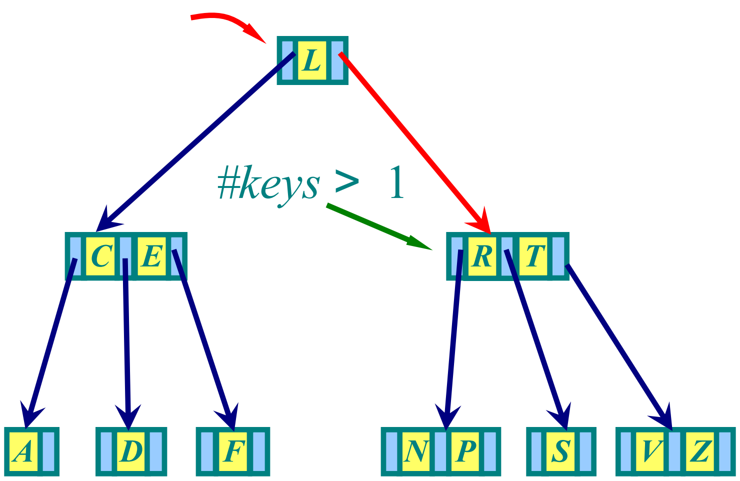
DELETE *Y, W, Q, X, K, B, H, P* **Minimum degree** $t = 2$



Case 1: *key* $k = H$ *is in a leaf.*

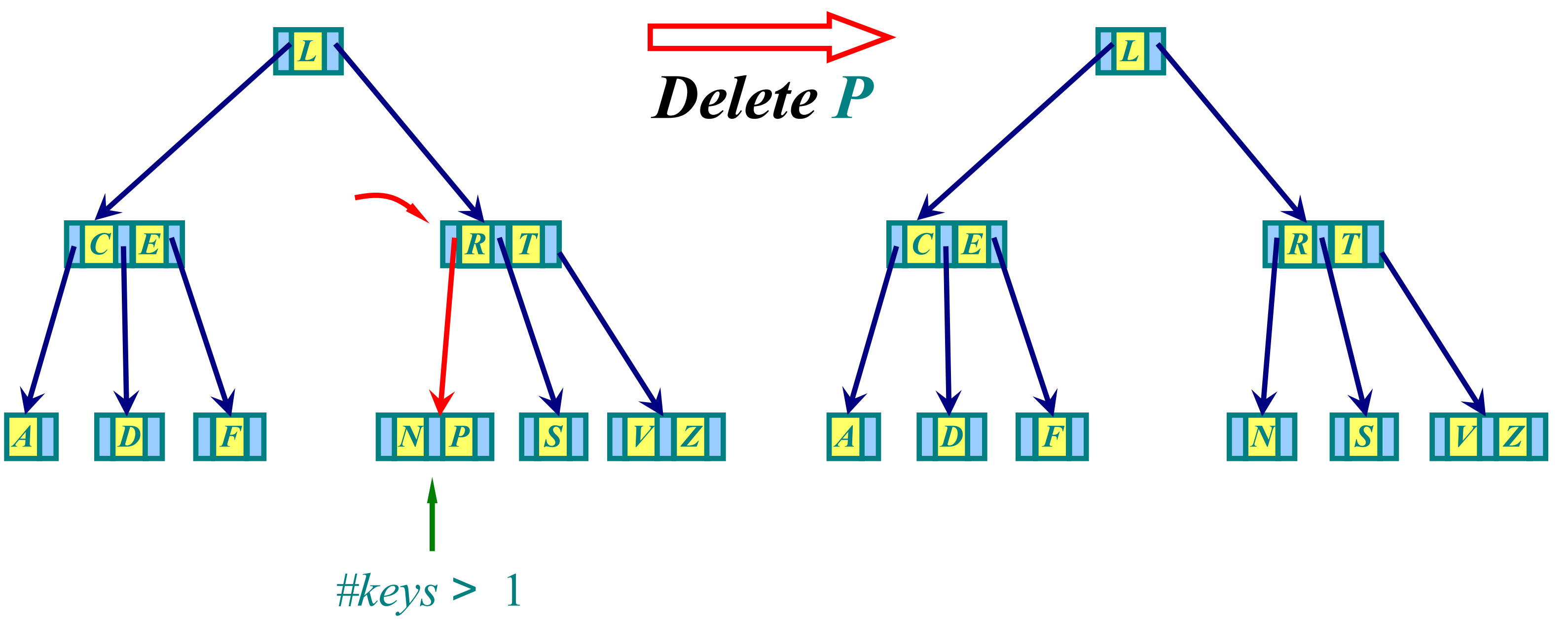
Deletion (B-tree)

DELETE *Y*, *W*, *Q*, *X*, *K*, *B*, *H*, *P* **Minimum degree** $t = 2$



Deletion (B-tree)

DELETE Y, W, Q, X, K, B, H, P **Minimum degree** $t = 2$



Case 1: *key* $k = P$ *is in a leaf.*

B-tree

Thinking and practice.

- *Write code for* **B-TREE-SEARCH**(x, k)
- *Write code for* **B-TREE-SPLIT-CHILD**(x, i, y)
- *Write code for* **B-TREE-INSERT**(T, k)
- *Write code for* **B-TREE-DELETE**(T, k)

How about B+ tree?

Any questions?

Xiaoqing Zheng
Fudan University