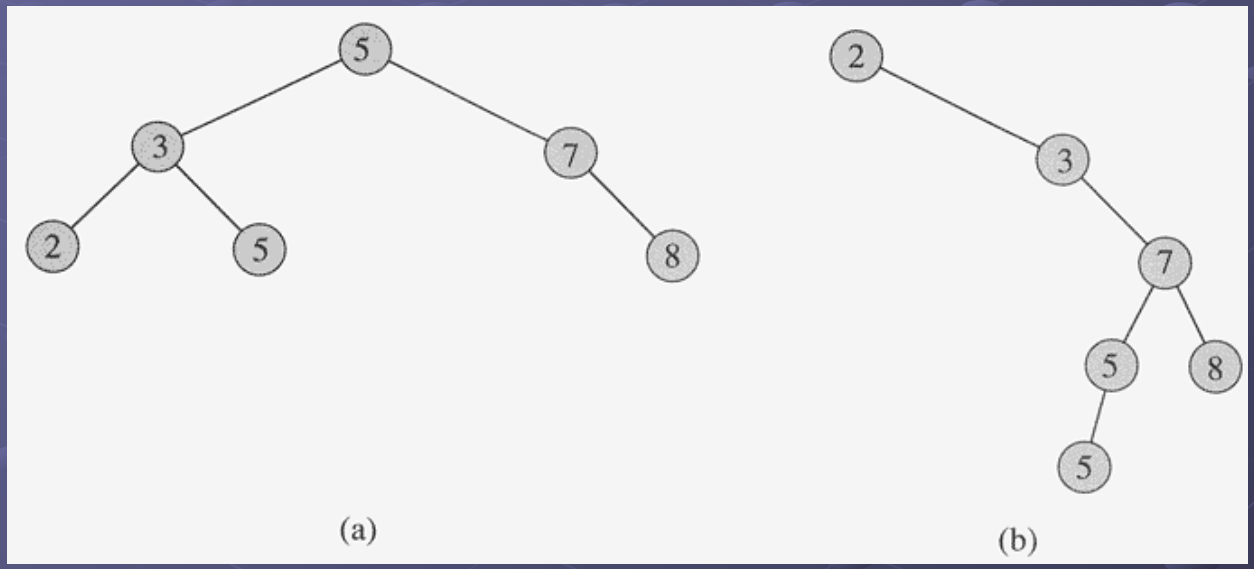
Binary Search Trees

交通大學 資訊工程系

What Is a Binary Search Tree?

☞ **Binary search tree**



- **The keys in a binary search tree satisfy the *binary-search-tree property*.**

What Is a Binary Search Tree?

- ***Binary-search-tree property***
  - Let x be a node in a binary search tree.
    - If y is a node in the left subtree of x, then $y.key \le x.key$.
    - If y is a node in the right subtree of x, then $y.key \ge x.key$.
- **Inorder tree walk**

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

What Is a Binary Search Tree?

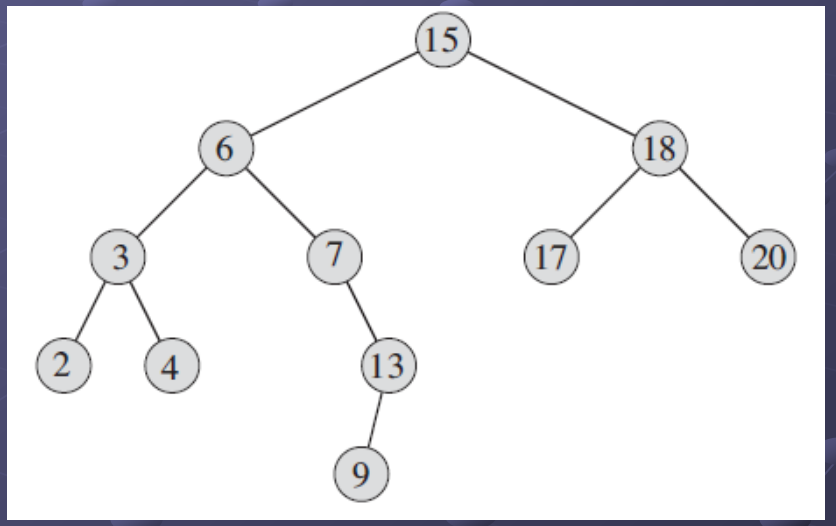
- Preorder tree walk
- Postorder tree walk
- *Theorem 12.1*
  - If x is the root of an n-node subtree, then the call INORDER-WALK(x) takes $\Theta(n)$ time.
  - Proof:

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

Querying a Binary Search Tree

Searching

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



- The running time of TREE-Search is O(*h*).

Querying a Binary Search Tree

- **Iterative tree search**

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

- **Minimum and maximum**

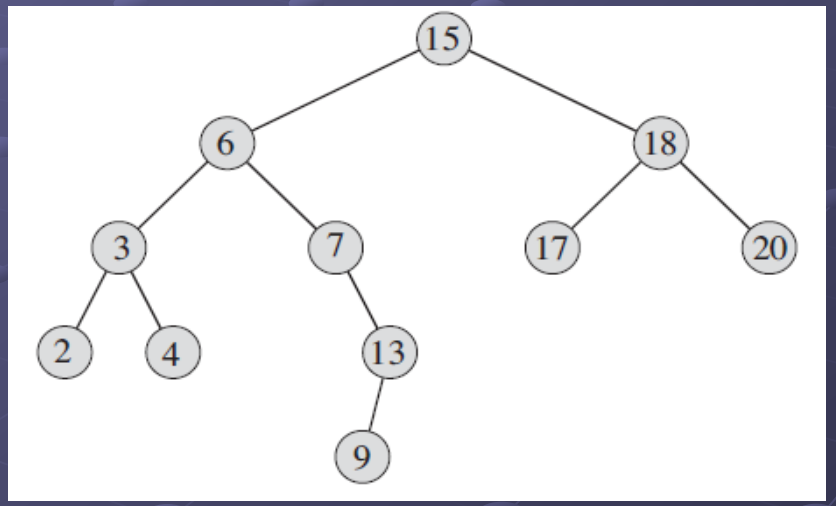
```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

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

Querying a Binary Search Tree

- Successor and predecessor

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

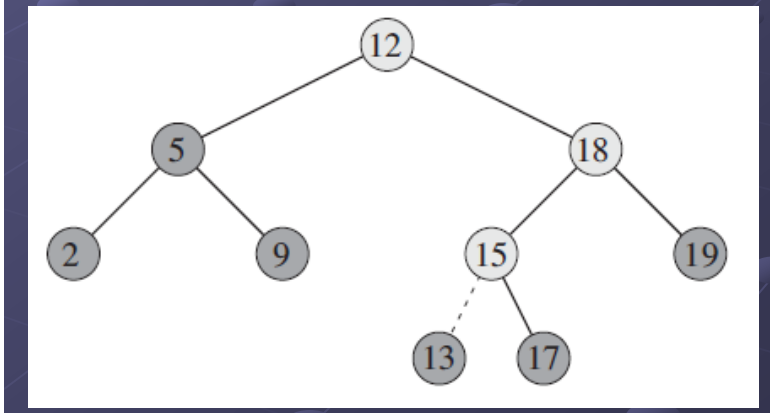


- *Theorem 12.2*
  - The dynamic-set operations, SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be made to run in $O(h)$ time on a binary search tree of height h.

Insertion and Deletion

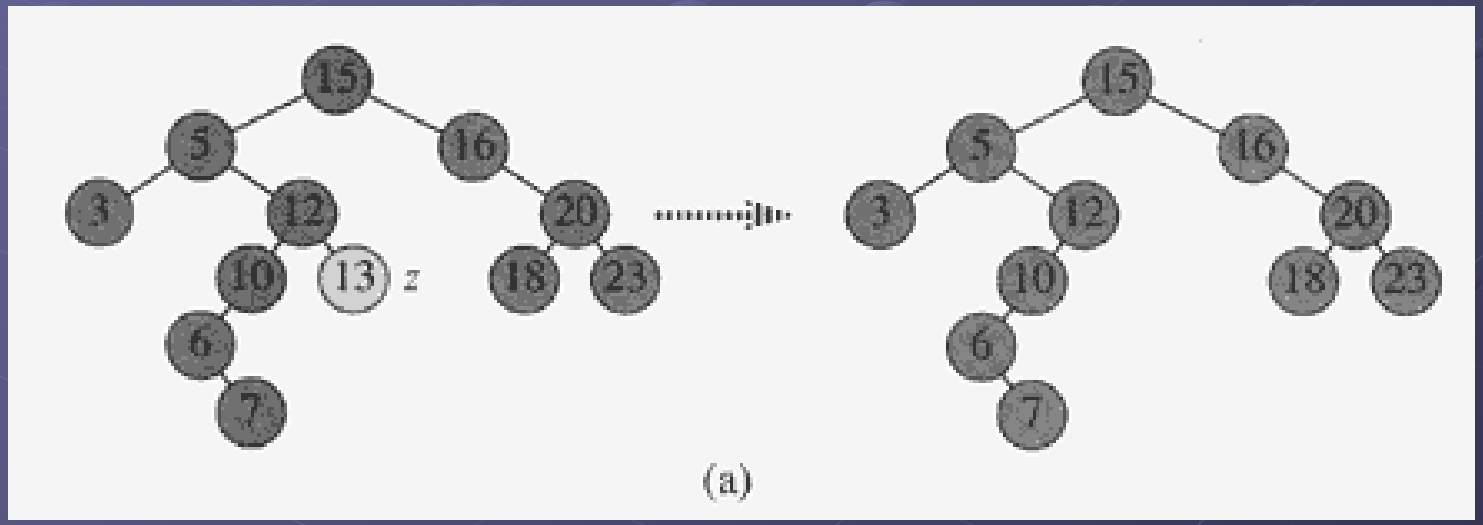
Insertion

```
TREE-INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z          // tree T was empty
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```



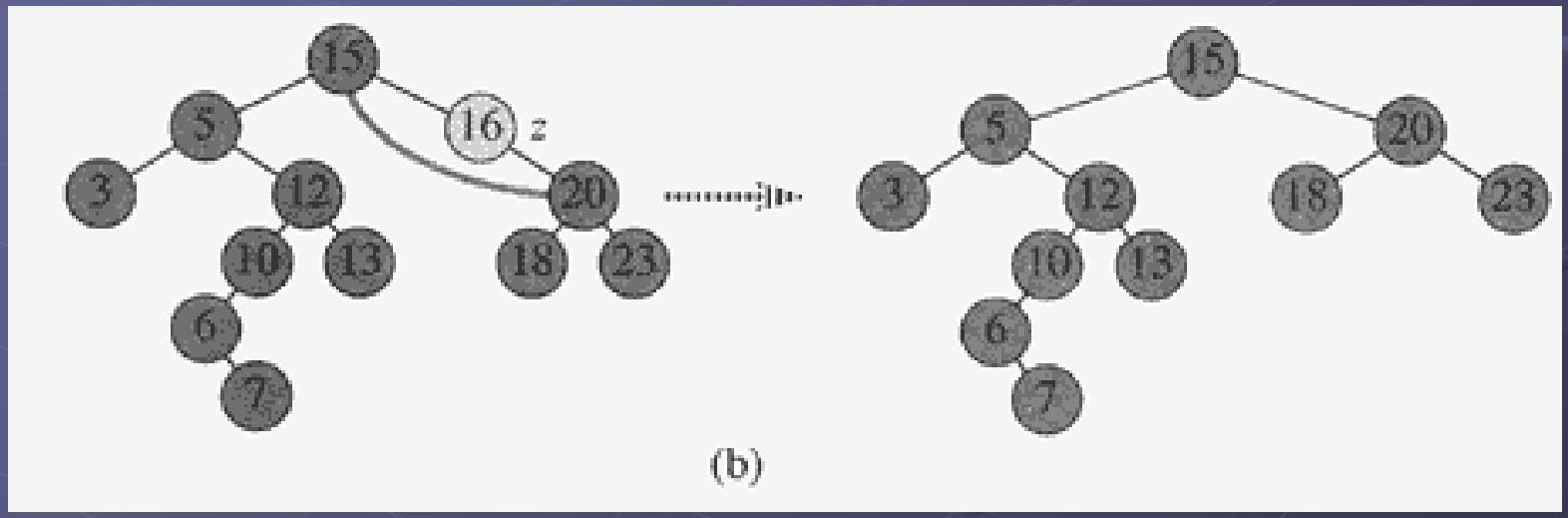
Insertion and Deletion

- **Deletion**
  - **Case 1: If *z* has no children.**



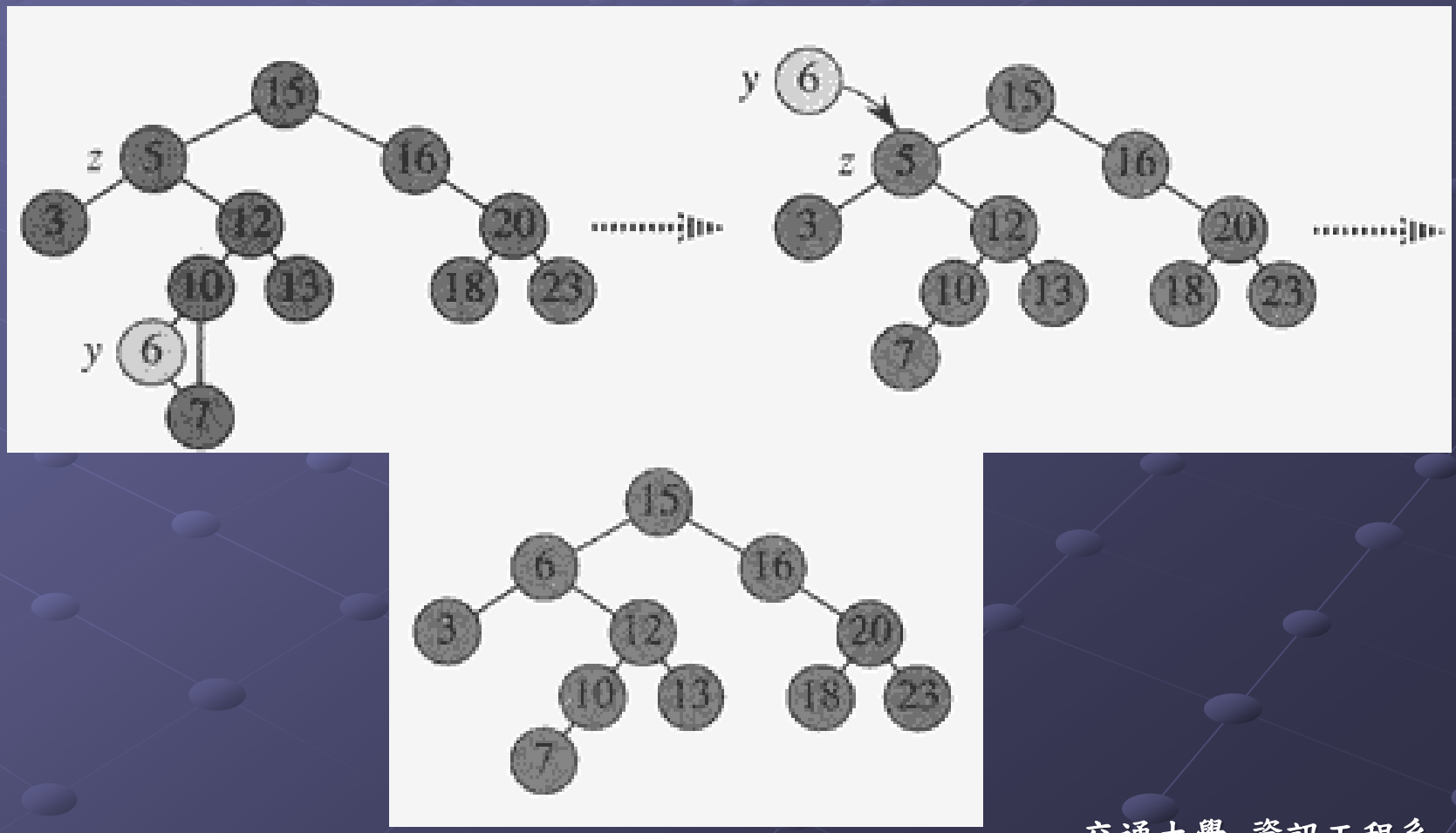
Insertion and Deletion

- **Case 2: If the node has only a single child.**



Insertion and Deletion

- **Case 3: If the node has two children.**



Insertion and Deletion

- **TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v.**
  - **Node $u's$ parent becomes node $v's$ parent.**
  - **$u's$ parent ends up having v as its appropriate child.**

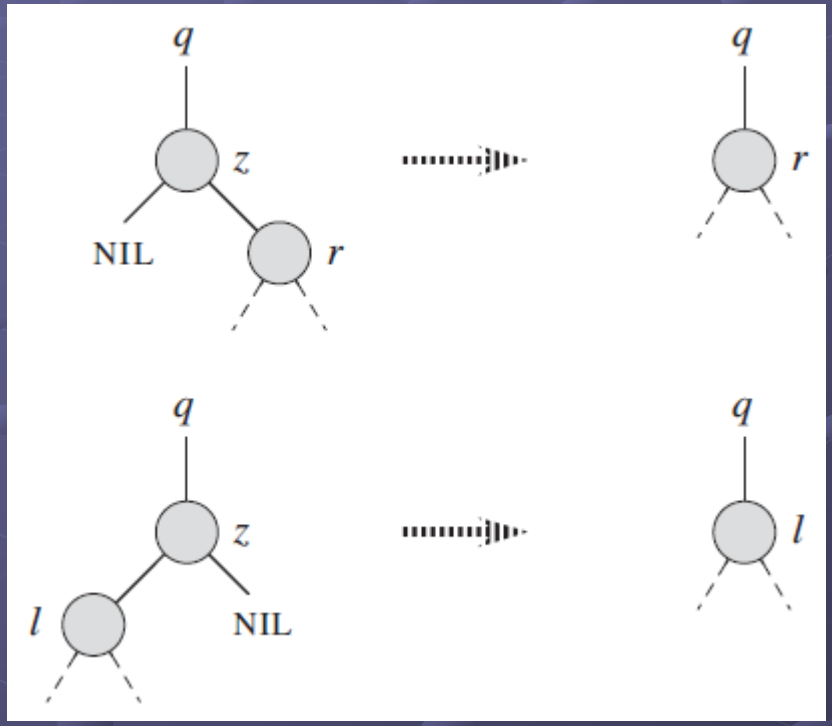
```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7       v.p = u.p
```

Insertion and Deletion

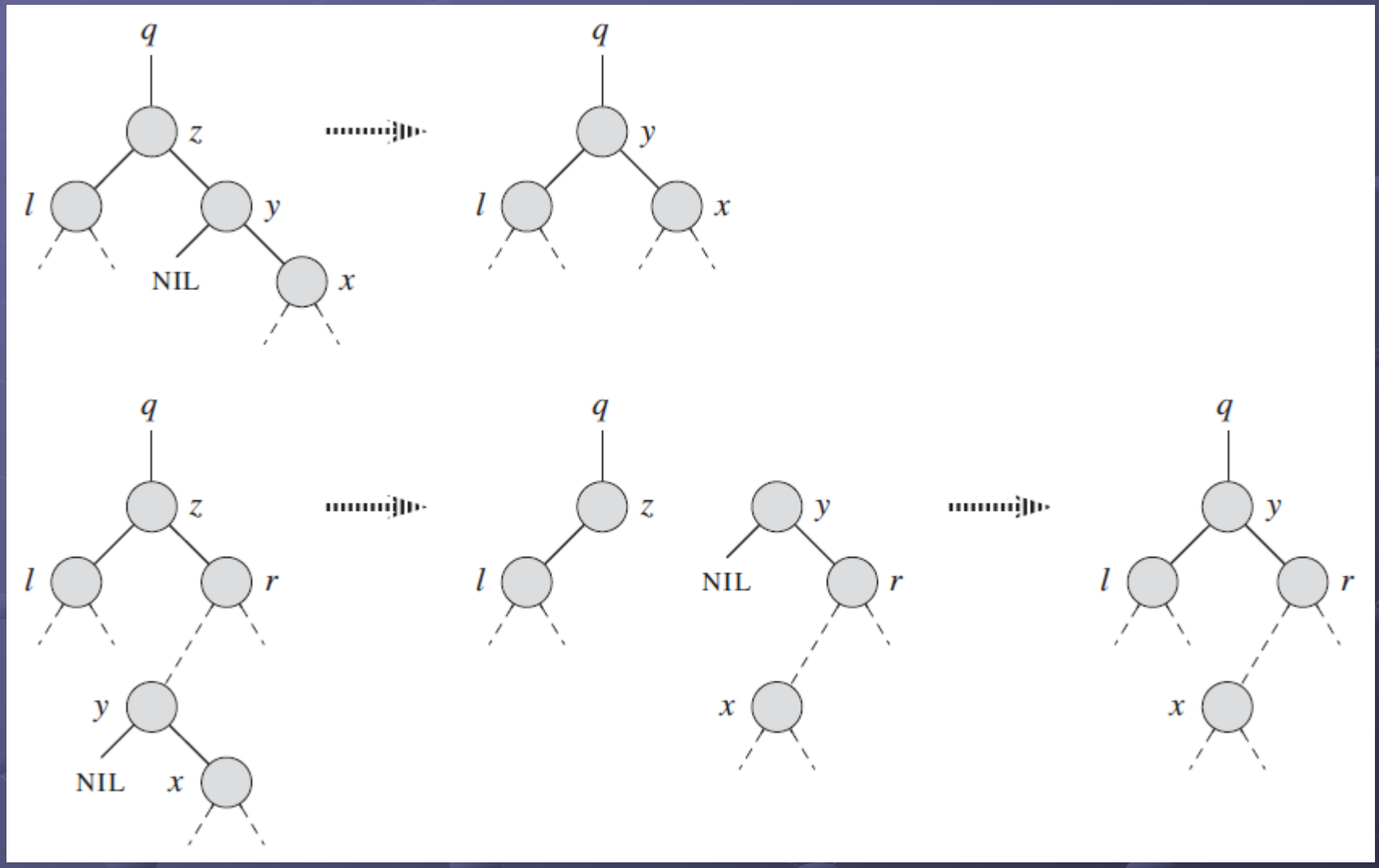
- **Tree delete with TRANSPLANT**

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Insertion and Deletion



Insertion and Deletion



Insertion and Deletion

☞ ***Theorem 12.3***

- **The dynamic-set operations, INSERT and DELETE can be made to run in O(*h*) time on a binary search tree of height *h*.**