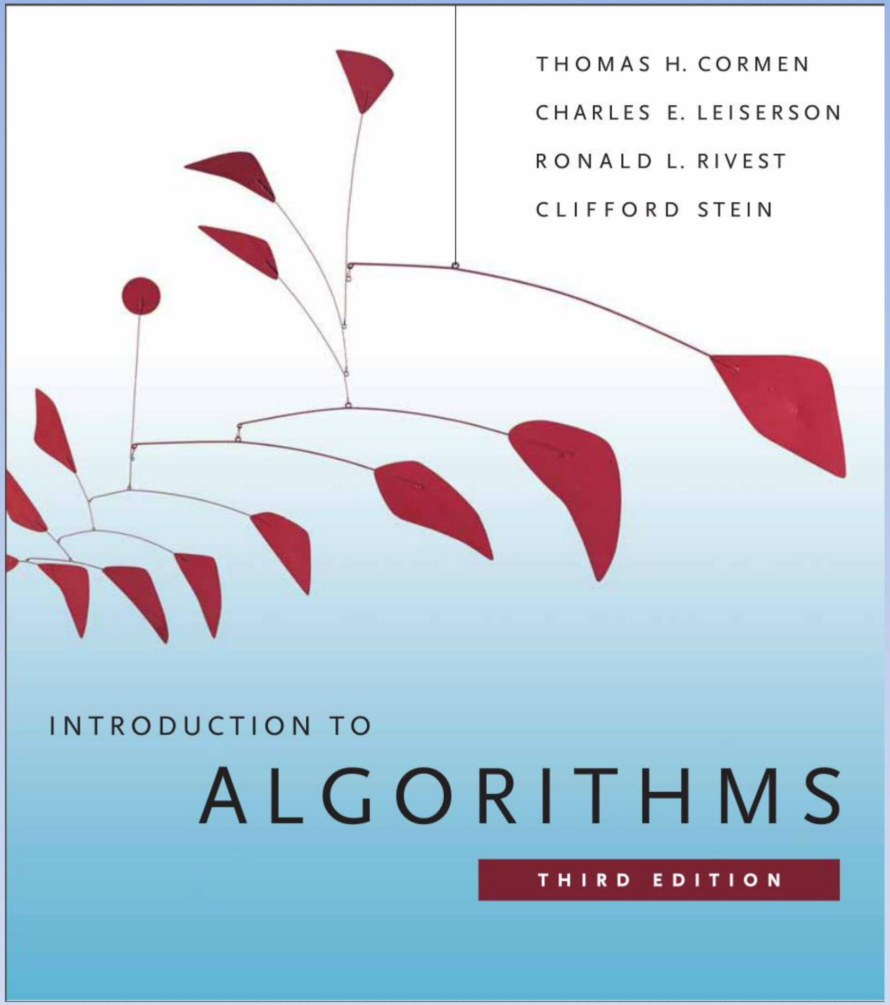
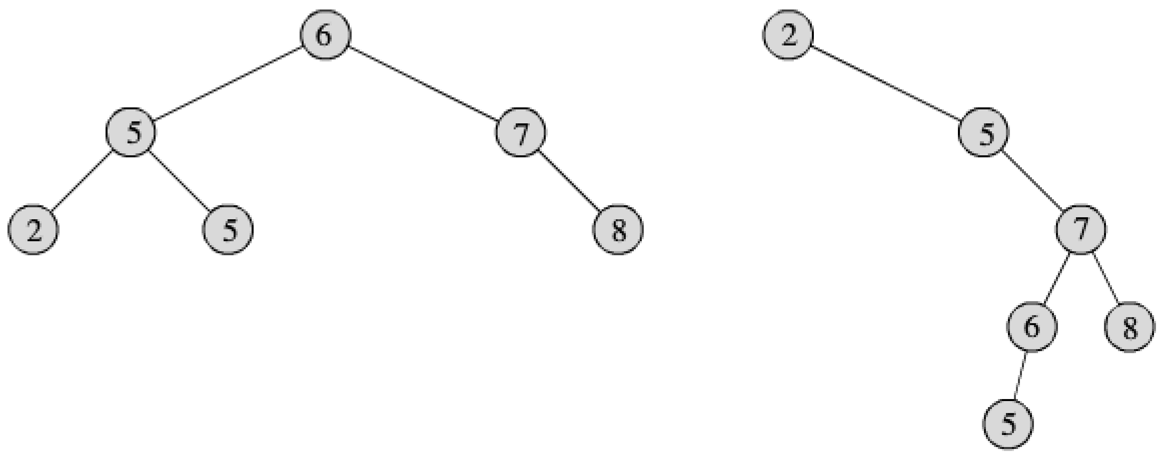


174 : Chapter 12

Binary Search Trees

Binary Search Trees



(a) (b)

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

Binary Search Trees

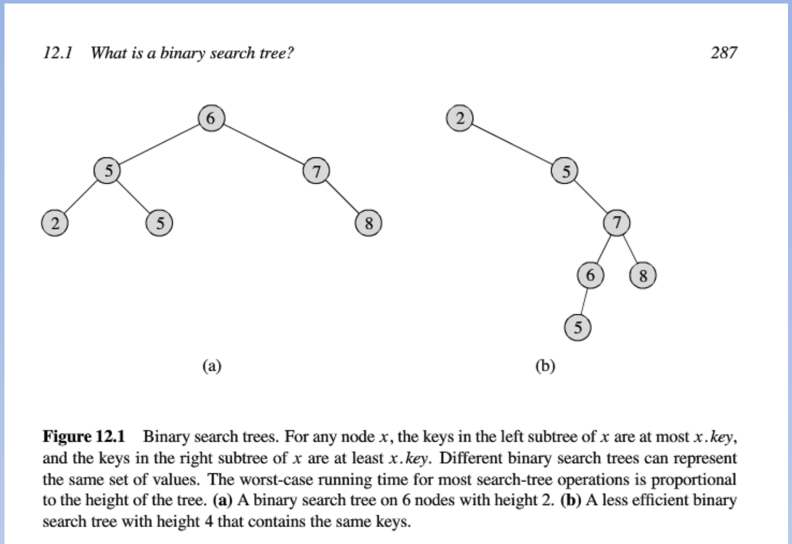


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

- Linked Data Structure
- Each node has pointers (along with key value and satellite data):
  - p: Parent
  - left: Left Subtree
  - right: Right Subtree

Binary-Search Tree Property

12.1 What is a binary search tree? 287

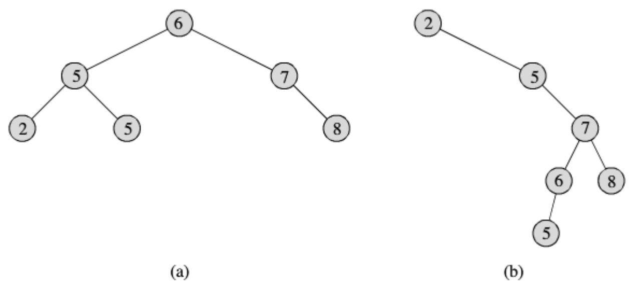
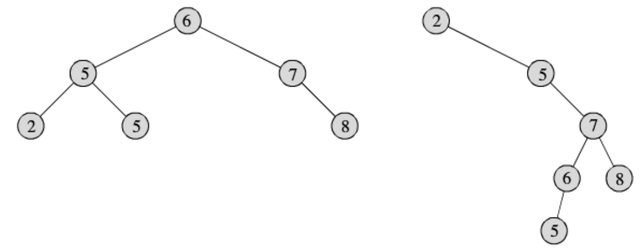


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

- Let x be a node in a binary search tree.
- IF y is a node in the left subtree of x,
  - THEN y.key $\leq$ x.key.
- IF y is a node in the right subtree of x,
  - THEN y.key $\geq$ x.key.

Binary-SearchTree Property



(a) (b)

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

- Inorder Walk allows printing out key value in sorted order:


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

Theorem 12.1
If x is the root of an n-node subtree, then the cal
takes $\Theta(n)$ time.

- In the ***substitution method***, we guess a bound and then use mathematical induction to prove our guess correct.
- The ***recursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- The ***master method*** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n) , \tag{4.2}$$

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree. Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$. It remains to show that $T(n) = O(n)$.

Since INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq \text{NIL}$), we have $T(0) = c$ for some constant $c > 0$.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is bounded by $T(n) \leq T(k) + T(n-k-1) + d$ for some constant $d > 0$ that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

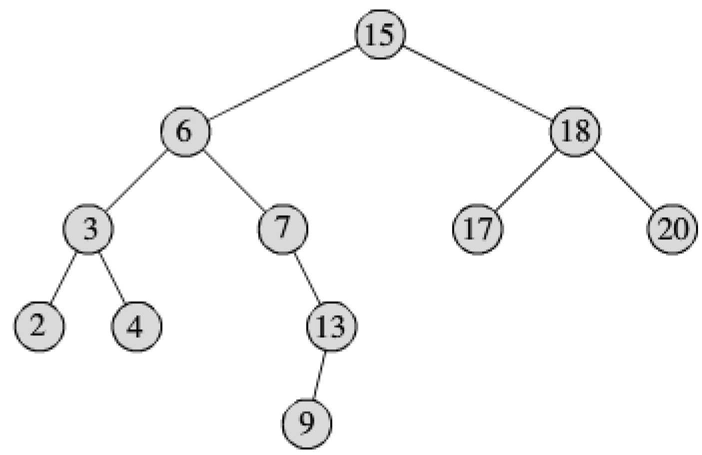
We use the substitution method to show that $T(n) = O(n)$ by proving that $T(n) \leq (c + d)n + c$. For $n = 0$, we have $(c + d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

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

which completes the proof. ■

Searching

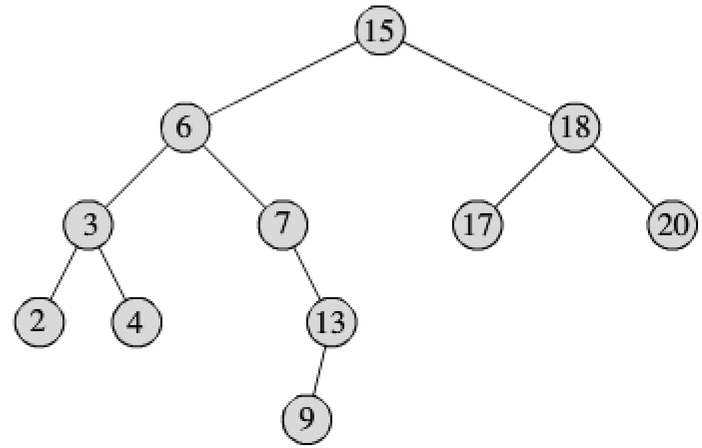
290 *Chapter 12 Binary Search Trees*




```
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)
```

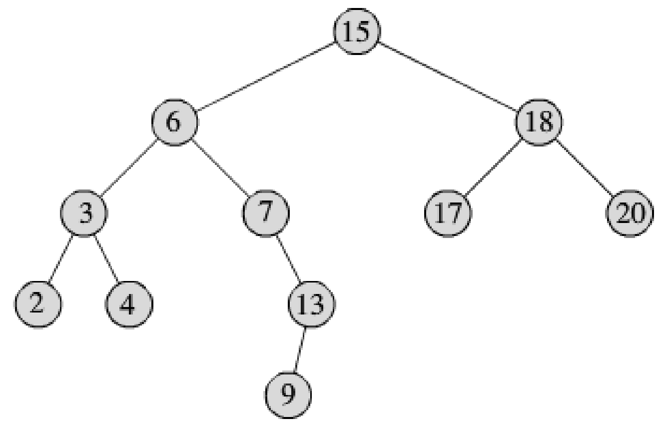
Iterative Tree Searching

290 *Chapter 12 Binary Search Trees*



12.2 Querying a binary search tree 291

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

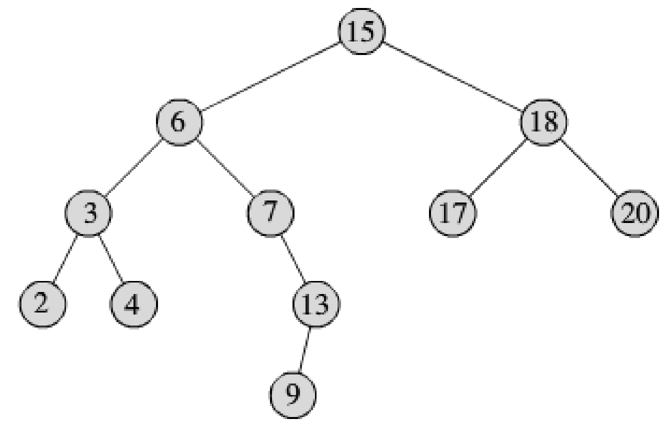



```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

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

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

IF Node has a Right
Succ is Min of Right

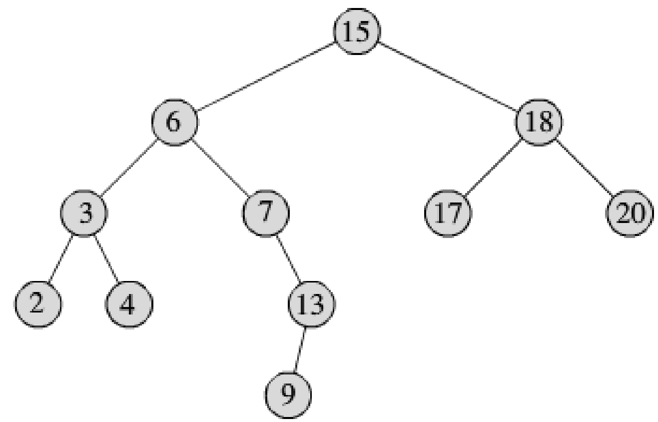



```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

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

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-M ... ight)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

ELSE: Look To Parent



TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    x = x.right
return x
```

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    return TREE-MINI .right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

IF: x is right child
Look To Parent

Return
First Left Child's Parent
or Nil

Insertion

- Example:
  - Inserting node z with key=13

12.3 *Insertion and deletion*

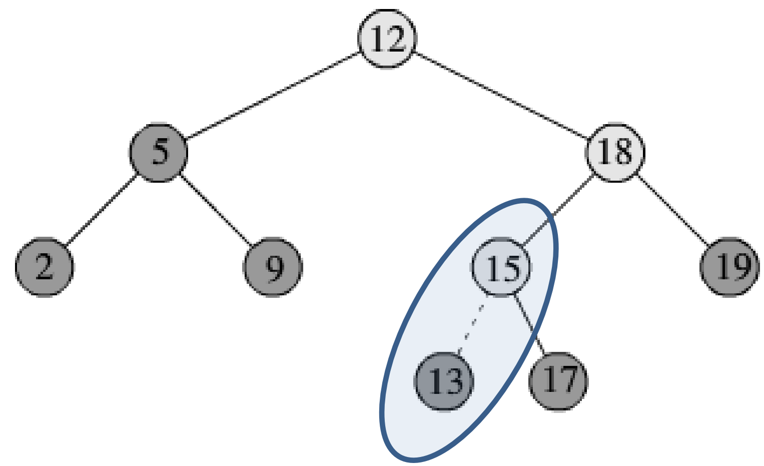


Figure 12.3 Inserting an item with key 13 into a binary search tree.

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

- y is trailing pointer
  - parent of x

Insertion

- Example:
  - Inserting node z with key=13

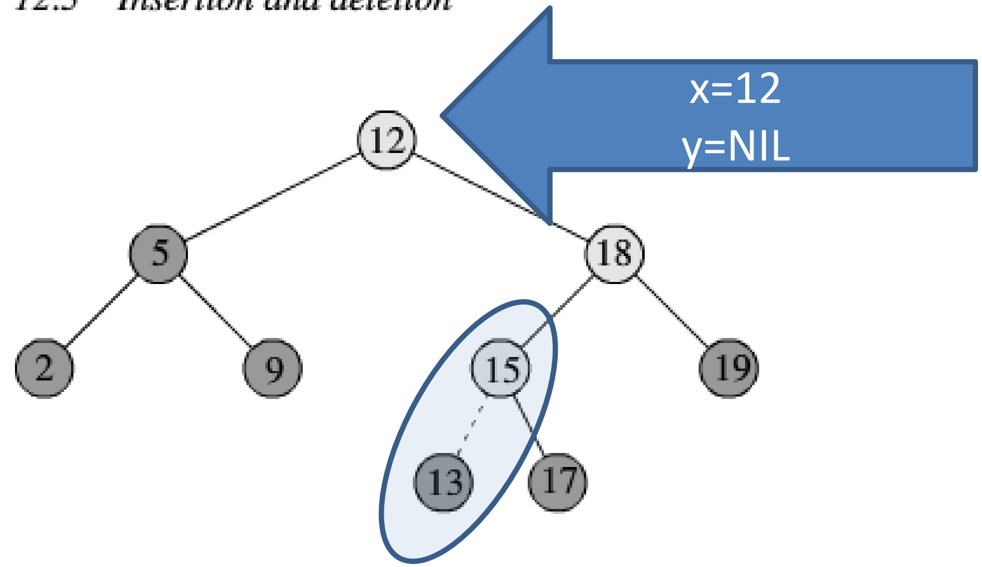


Figure 12.3 Inserting an item with key 13 into a binary search tree.

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

- 13 > 12

Insertion

- Example:
  - Inserting node z with key=13

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z        // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

12.3 Insertion and deletion

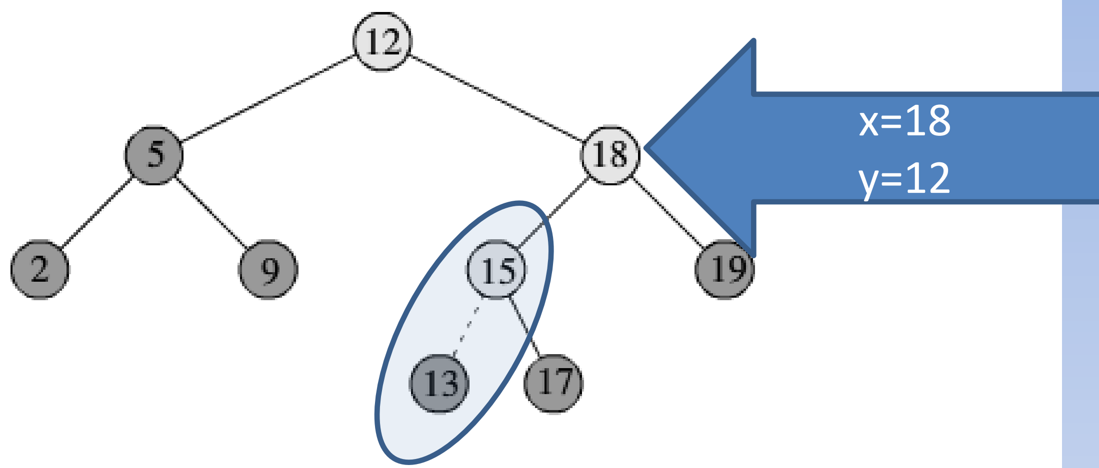


Figure 12.3 Inserting an item with key 13 into a binary search tree.

- 13 < 18

Insertion

- Example:
  – Inserting node z with key=13

12.3 *Insertion and deletion*

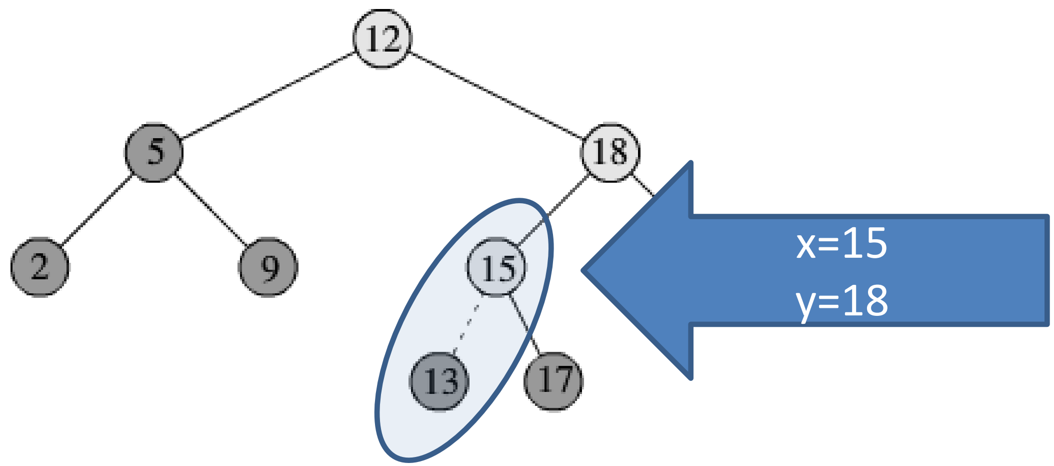


Figure 12.3 Inserting an item with key 13 into a binary search tree.

- 13 < 15

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Insertion

- Example:
  - Inserting node z with key=13

12.3 Insertion and deletion

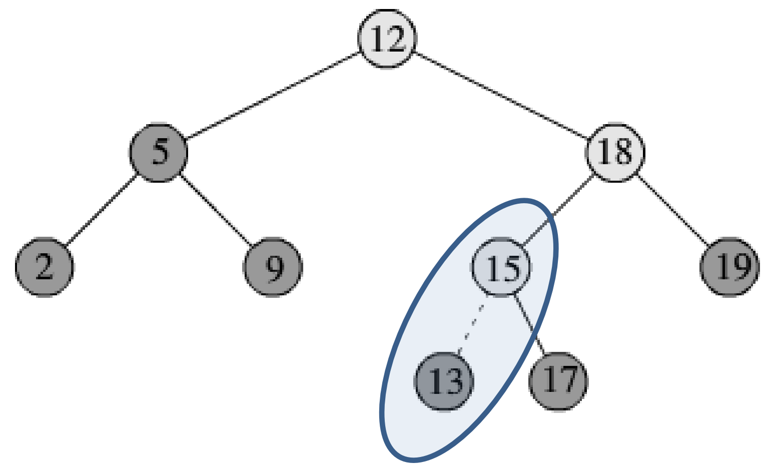


Figure 12.3 Inserting an item with key 13 into a binary search tree.

- 13 < 15

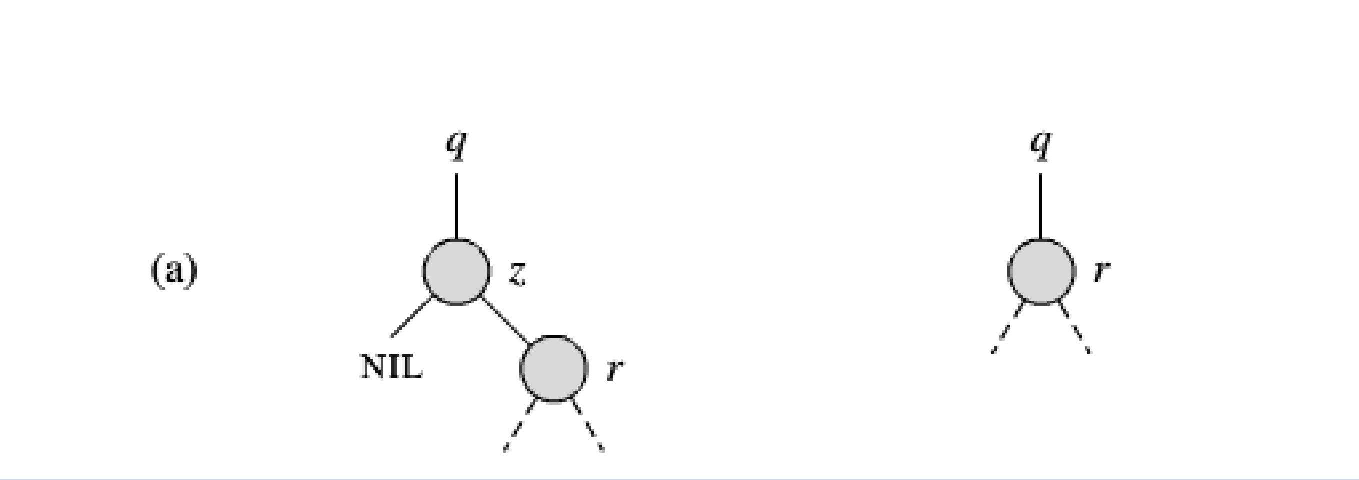
```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

z's parent is y

z becomes y's left or right child

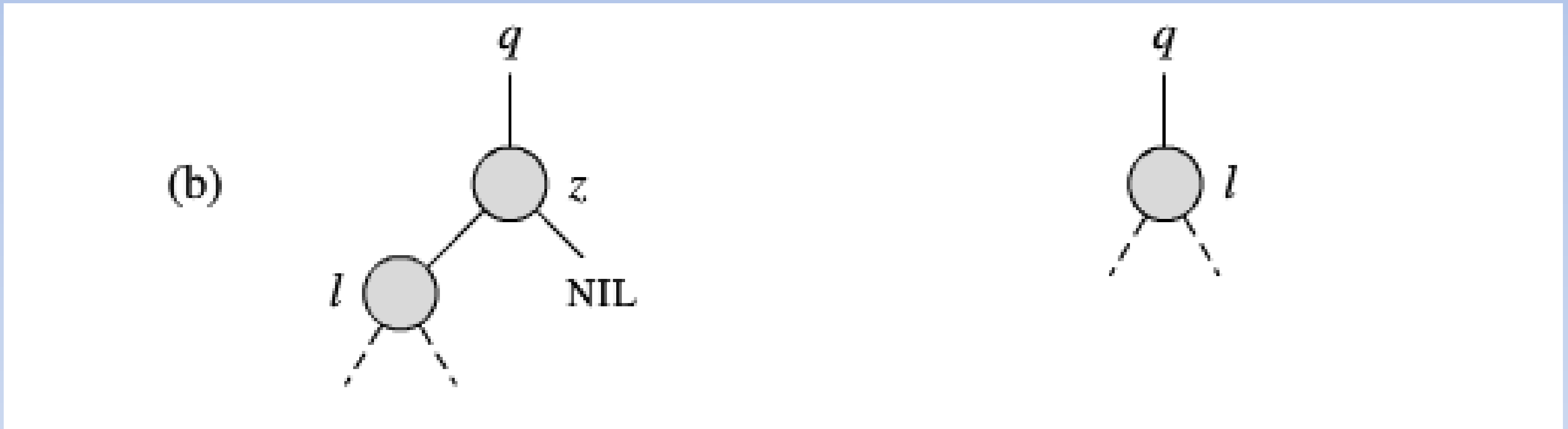
Deletion
Several Different Cases

- No Left Subchild



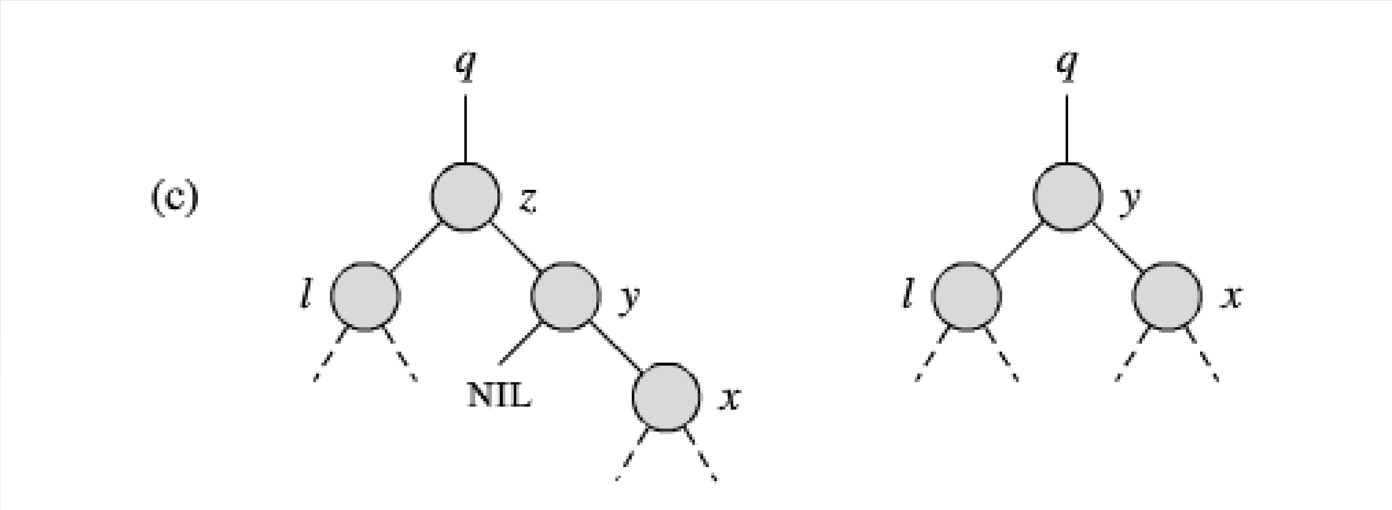
Deletion

- No Right Subchild



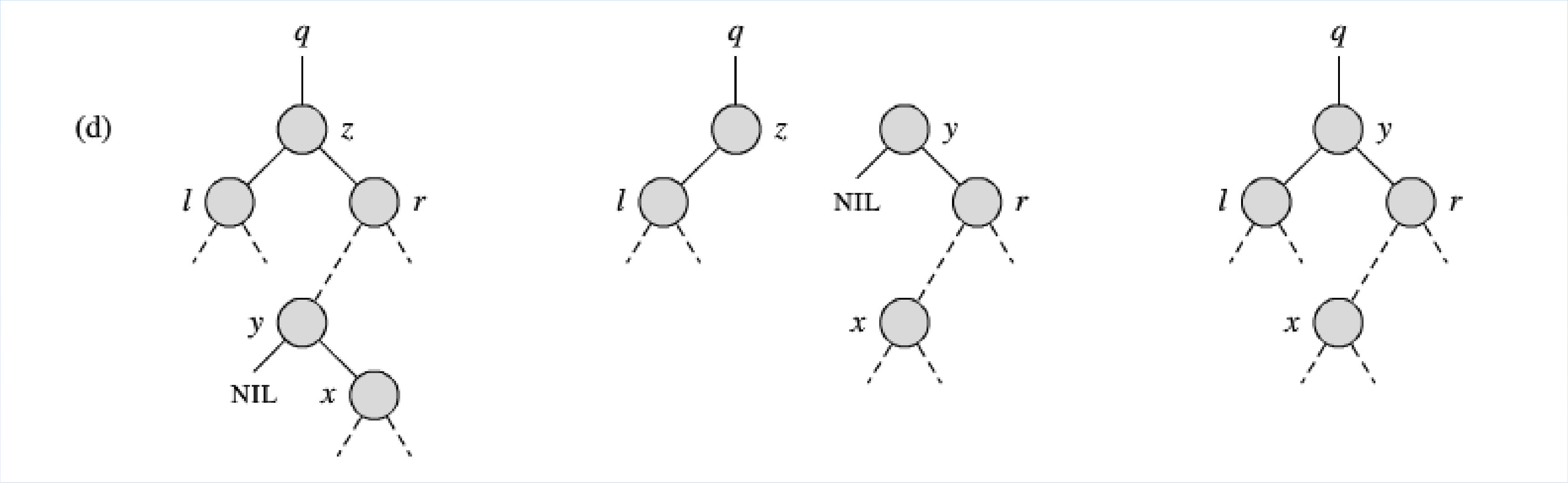
Deletion

- Node Z has two children;
  - its left child is node l,
  - its right child is its successor y,
  - and y's right child is node x.
- We replace Z by y,
  - updating y's left child to become l,
  - but leaving x as y's right child.

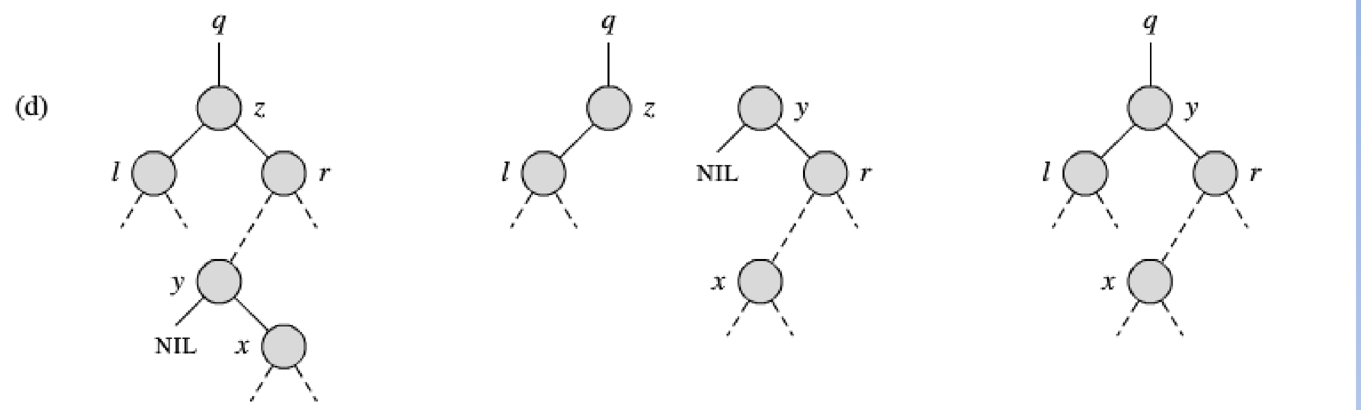


Deletion

- Node Z has two children :
  - left child l and right child r
  - its successor y $\neq$ r lies within the subtree rooted at r.
- We replace y by its own right child x,
- We set y to be r's parent.
- Then, we set y to be q's child and the parent of l.



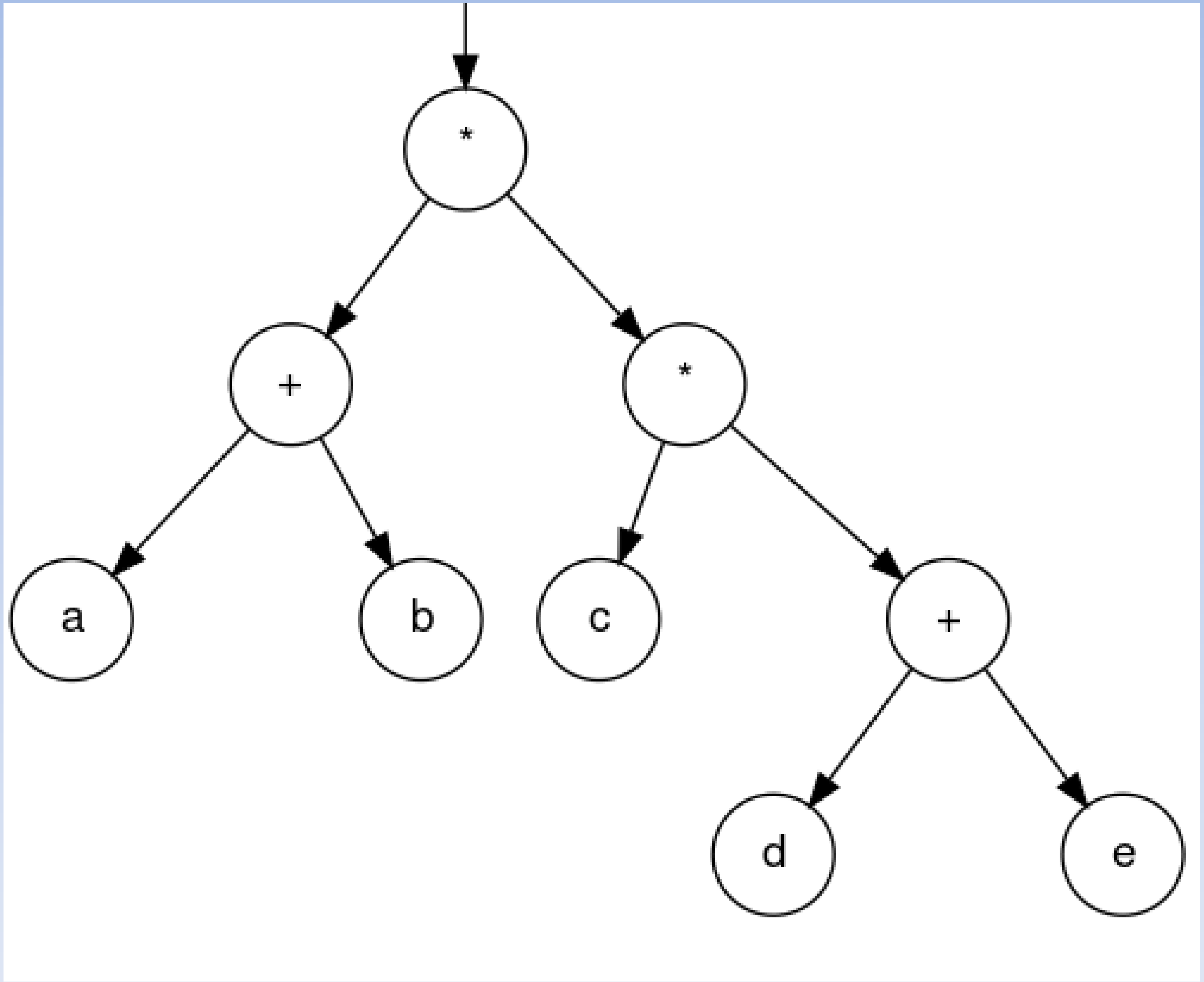
Deletion



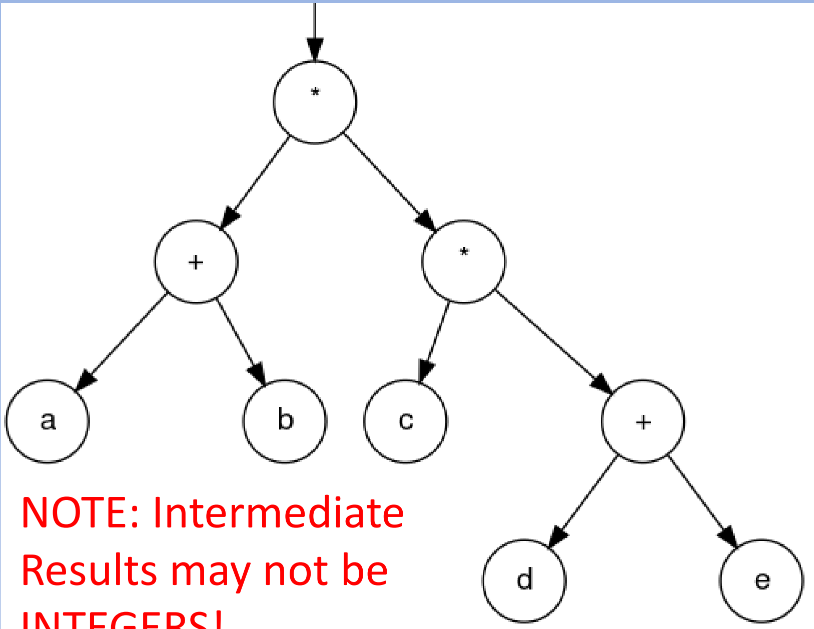

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(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
```

Expression Tree



Expression Tree



NOTE: Intermediate Results may not be INTEGERS!

- Eval(Node):
- if (node.key==OPERATOR)
  - leftVal = Eval(node.left)
  - rightVal = Eval(node.right)
  - return ExecuteOperator(node.key, leftVal, rightVal)
- Else return node.key