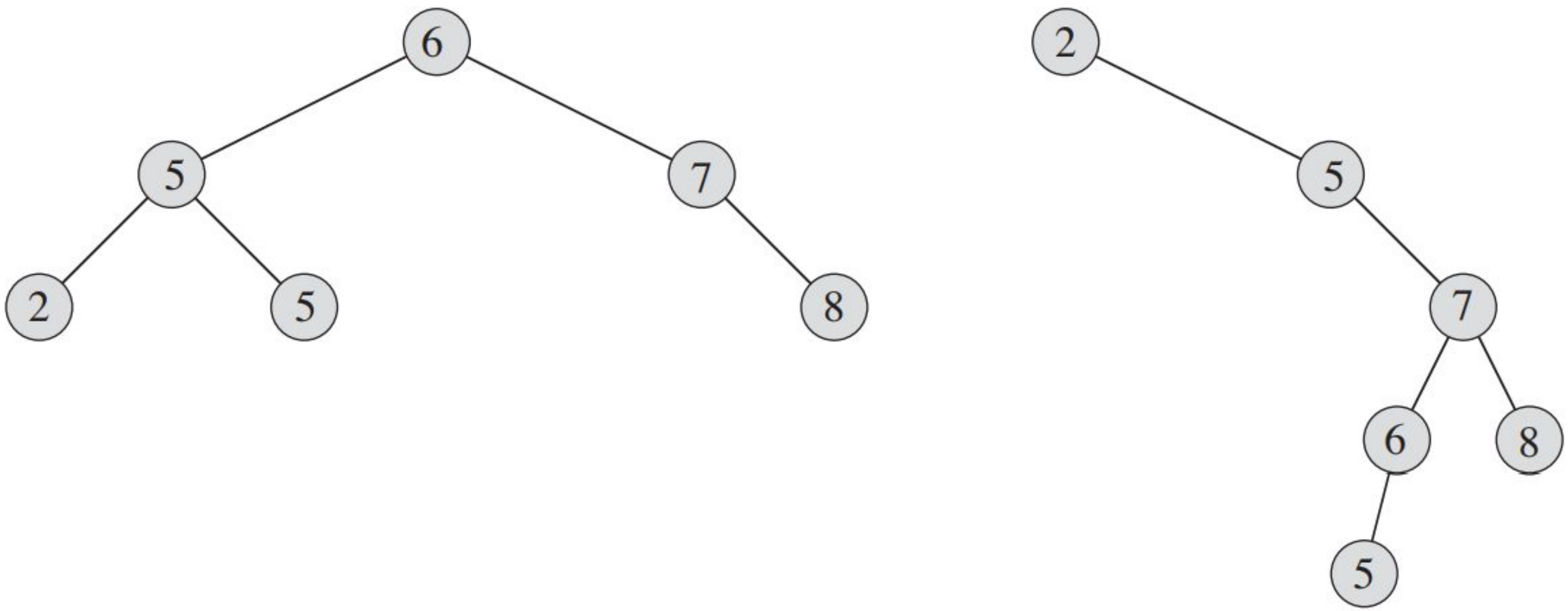
Binary Search Tree

DATASTRUCTURES & ALGORITHMS

Binary search tree(BST)

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 $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.



(a) (b)

Inorder tree walk

- The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk**.
- This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree.
- Similarly, 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.

Inorder tree walk

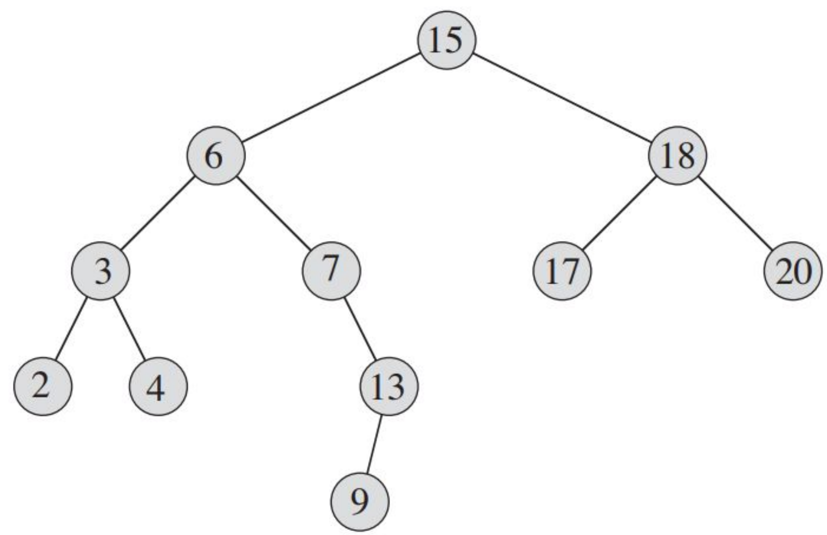
```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

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

Querying a binary search tree

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



Querying a binary search tree

TREE-MINIMUM(x)

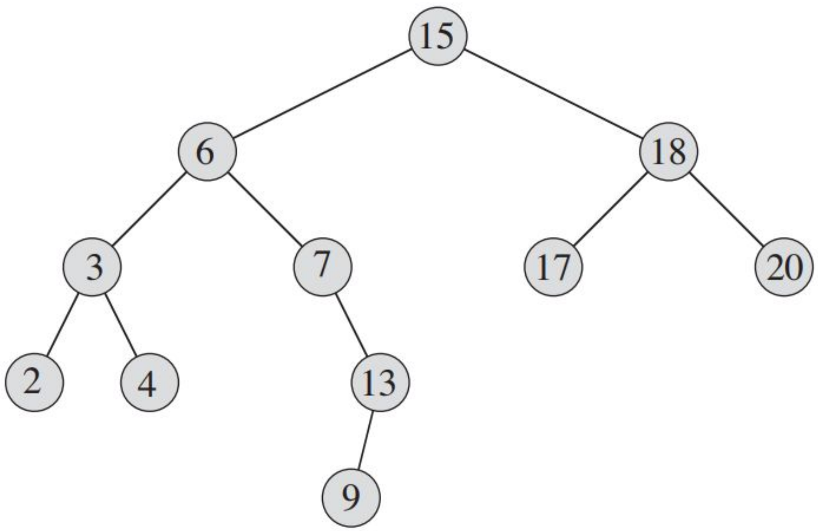
```
while x.left ≠ NIL
    x = x.left
return x
```

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
```



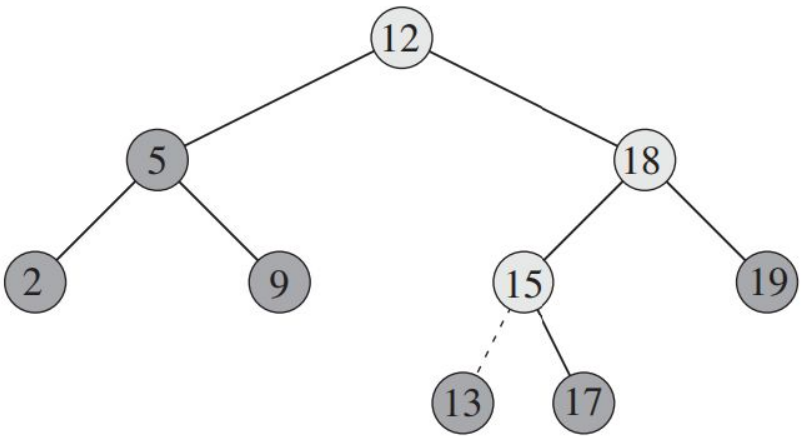
Running time

We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in $O(h)$ time on a binary search tree of height h. ■

Insertion

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
```



12.3-1

Give a recursive version of the TREE-INSERT procedure.

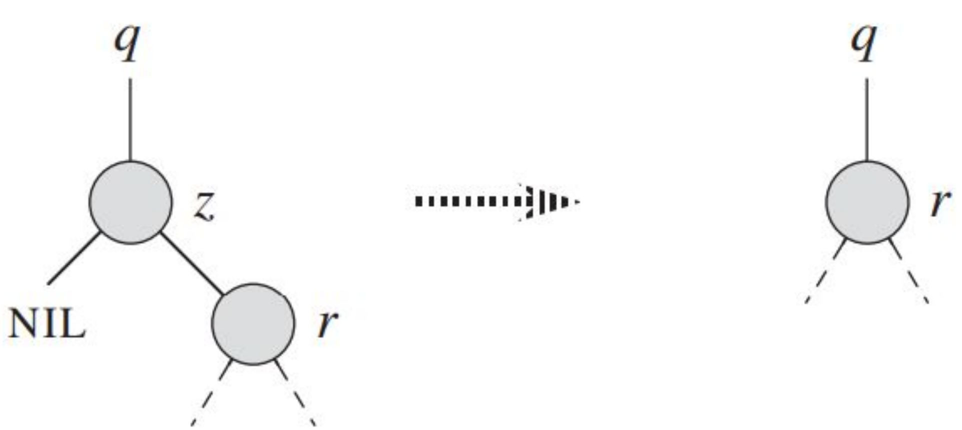
Deletion

The overall strategy for deleting a node **z** from a binary search tree T has three basic cases:

- If **z** has no children, then we simply remove it by modifying its parent to replace **z** with NIL as its child.
- If **z** has just one child, then we elevate that child to take **z**'s position in the tree by modifying **z**'s parent to replace **z** by **z**'s child.
- If **z** has two children, then we find **z**'s successor y—which must be in **z**'s right subtree—and have y take **z**'s position in the tree. The rest of **z**'s original right subtree becomes y's new right subtree, and **z**'s left subtree becomes y's new left subtree. This case is the tricky one because, as we shall see, it matters whether y is **z**'s right child.

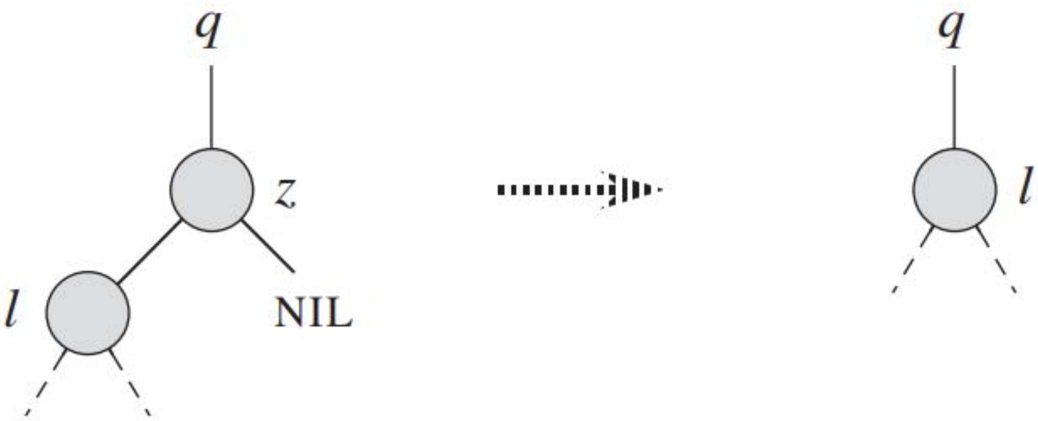
Deletion

If **z** has no left child , then we replace **z** by its **right child**, which may or may not be NIL. When **z**'s right child is NIL, this case deals with the situation in which **z** has no children. When **z**'s right child is non-NIL, this case handles the situation in which **z** has just one child, which is its right child.



Deletion

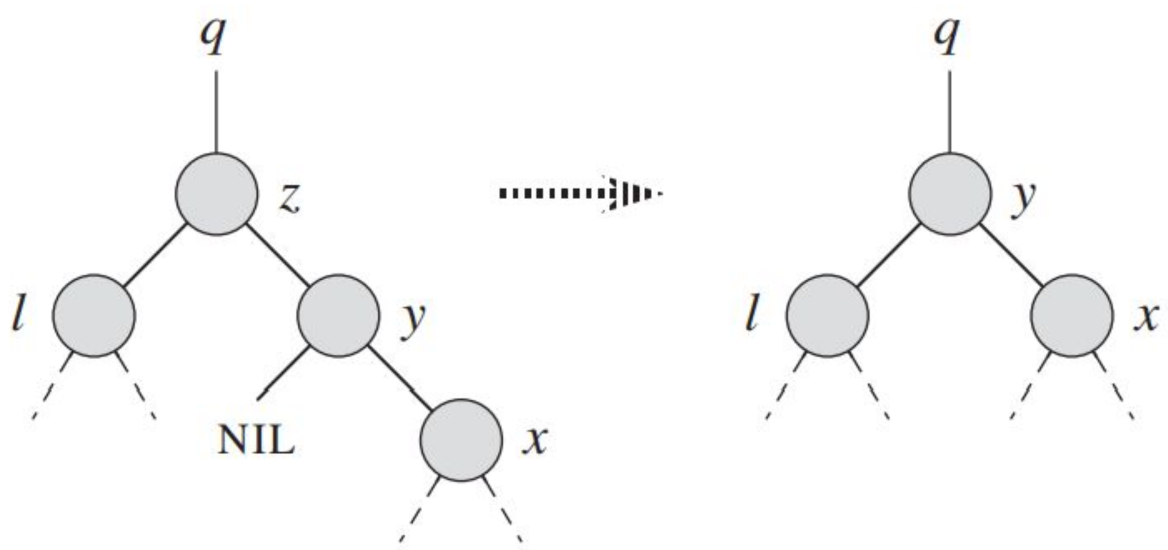
If **z** has just one child, which is its left child , then we replace **z** by its left child.



Deletion

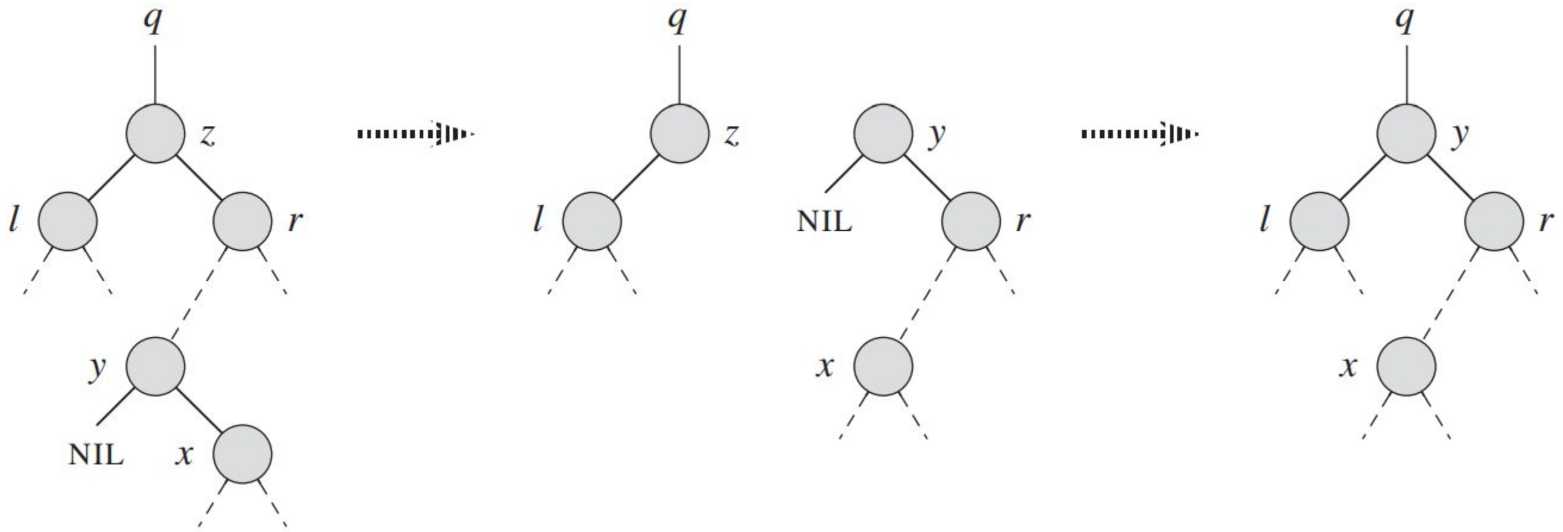
Otherwise, **z** has both a left and a right child. We find **z**'s successor y, which lies in **z**'s right subtree and has no left child.

If y is **z**'s right child , then we replace **z** by y.



Deletion

Otherwise, replace y by its own right child, and then we replace **z** by y.



Transplant

TRANSPLANT replaces the subtree rooted at node **u** with the subtree rooted at node **v** , node **u**’s parent becomes node **v**’s parent, and **u**’s parent ends up having **v** as its appropriate child.

```
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
```

Deletion

```
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
```

We can implement the dynamic-set operations INSERT and DELETE so that each one runs in $O(h)$ time on a binary search tree of height h. ■

Randomly built binary search trees

Theorem 12.4
The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$.