Next Computational Problem: Searching

AUBURN UNIVERSITY

Hugh Kwon

Slides adapted from Dr. Debswapna Bhattacharya's class

Input: A sorted array A of n distinct numbers, and a number k

Output: The index i of the array cell in which k appears, or -1 if k is not in A

Binary Search Algorithm

```
BinarySearch(A: array [p…r] of number sorted in the ascending order, k: number)
if p==r then
        if A[p]==k then return p else return −1
mid=⌊(p + r)/2⌋
if A[mid]==k then return mid
else if A[mid]>k then
        return  BinarySearch(A[p...mid−1], k)
else return BinarySearch(A[mid+1...r], k)
```

Thinking Assignments

How/why does this algorithm work?

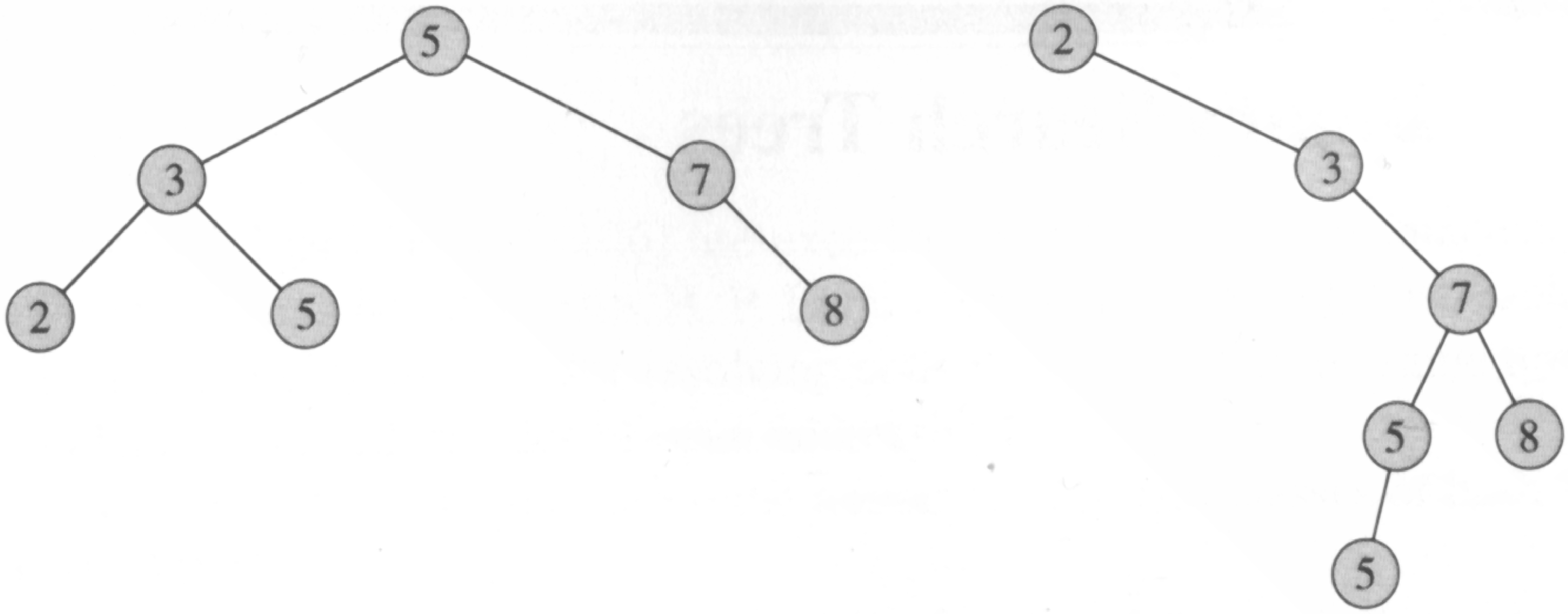
- Draw its recursion tree for a specific input Is it correct?

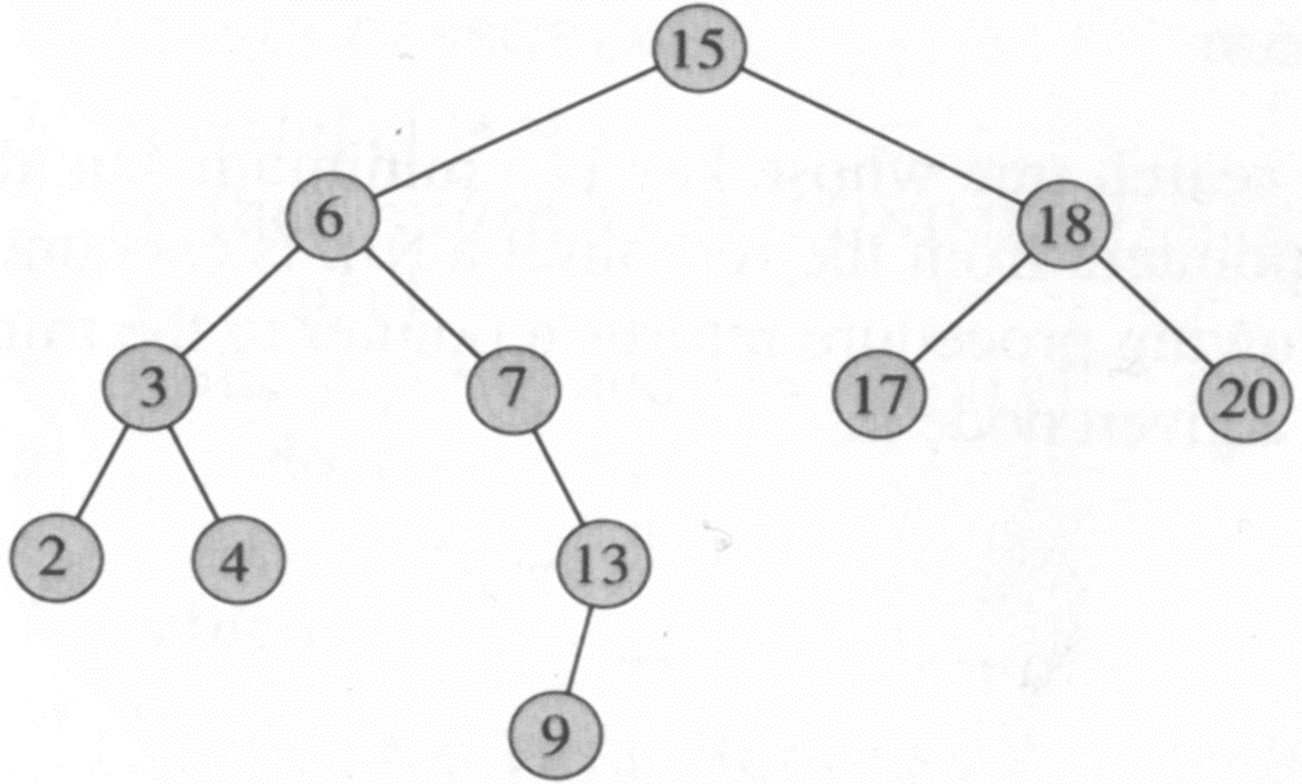
How efficient is it?

- Estimate its complexity approximately
- Calculate its detailed complexity
  - Develop its two recurrence relations
  - Solve them

- 12.1 What is a binary search tree?
- ***Binary-search property*:**
  - Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key[y] $\leq$ key[x]. If y is a node in the right subtree of x, then key[x] $\leq$ key[y].

Binary Search Tree




15
6
18
3
7
17
20
2
4
13
9

TREE-SEARCH(x,k)

```
if x==NIL or k==x.key
        return x
if k < x.key
then 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
```

Thinking Assignment: Understand how this algorithm works, and how it is similar to and different from the recursive algorithm.

The primary use of the BST data structure is to enable efficient search. But there are other useful algorithmic operations available on this data structure.

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 tree, then the call INORDER-TREE-WALK(x) takes Θ(n) time...why?

1. Each recursive execution takes (ignoring the recursive calls) takes Θ(1) time.
2. Exactly one recursive execution per node, and there are n nodes, so total # of recursive executions is Θ(n)

Write the Preorder tree walk algorithm

Write the Postorder tree walk algorithm

Maximum and Minimum

Where in the tree is max element?

Where in the tree is min element?

Maximum and Minimum

- TREE-MINIMUM(x)

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

- TREE-MAXIMUM(x)

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

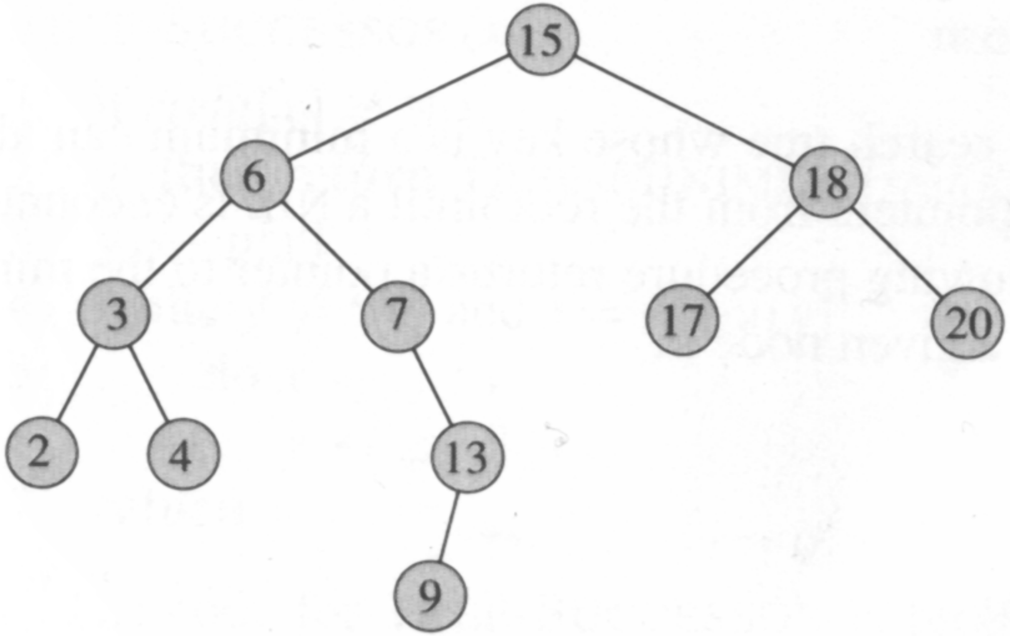
Successor of a tree node x is the node with the smallest key greater than x.key.

Where in the tree will you find it?

1. If x is the largest node in the BST, its successor is NIL
2. Else it is the smallest node in the right subtree of x
3. But if x has no right subtree, it is the first ancestor of x along a left edge

TREE-SUCCESSOR(x:node)

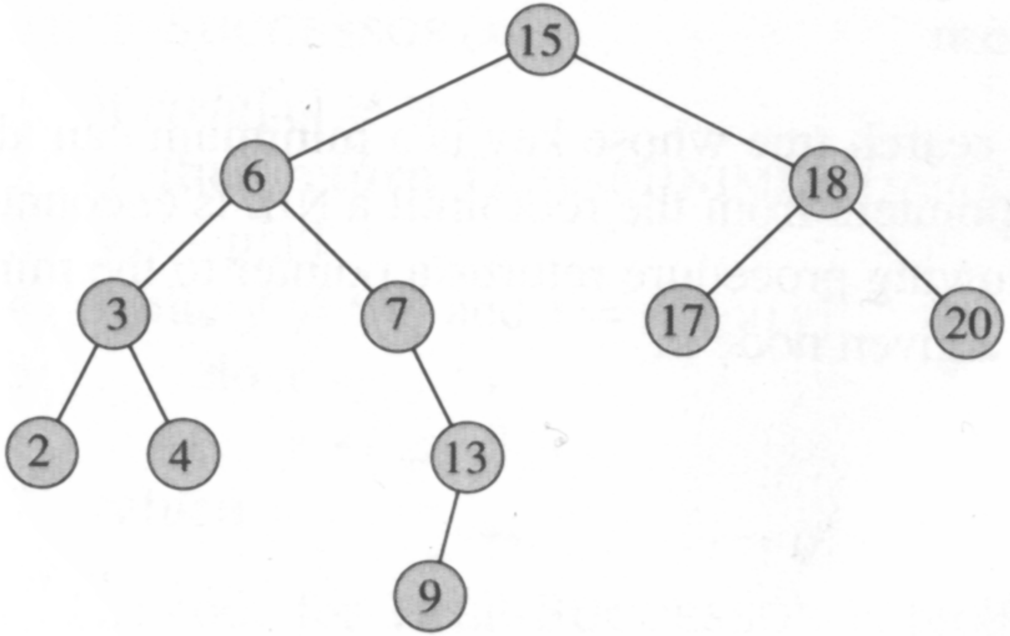
```
1   if x.right≠NIL
2       return TREE-MINIMUM(x.right)
3   y=x.parent
4   while y≠NIL and x==y.right
5       x=y
6       y=y.parent
7   return y
```



CASE I: If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree, which we find in line 2 by calling TREE-MINIMUM(x.right). For example, the successor of the node with key 15 in is the node with key 17.

TREE-SUCCESSOR(x:node)

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



CASE II: If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. The successor of the node with key 13 is the node with key 15. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3–7 of TREE-SUCCESSOR handle this case

Predecessor of tree node x is the node with the smallest key greater than x.key... think about where in the tree you can find it, based on the discussion about where to find successor

Write the algorithm TREE-PREDECESSOR by modifying the TREE-SUCCESSOR algorithm appropriately

Operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR run in O(h) time on a binary search tree of height h.

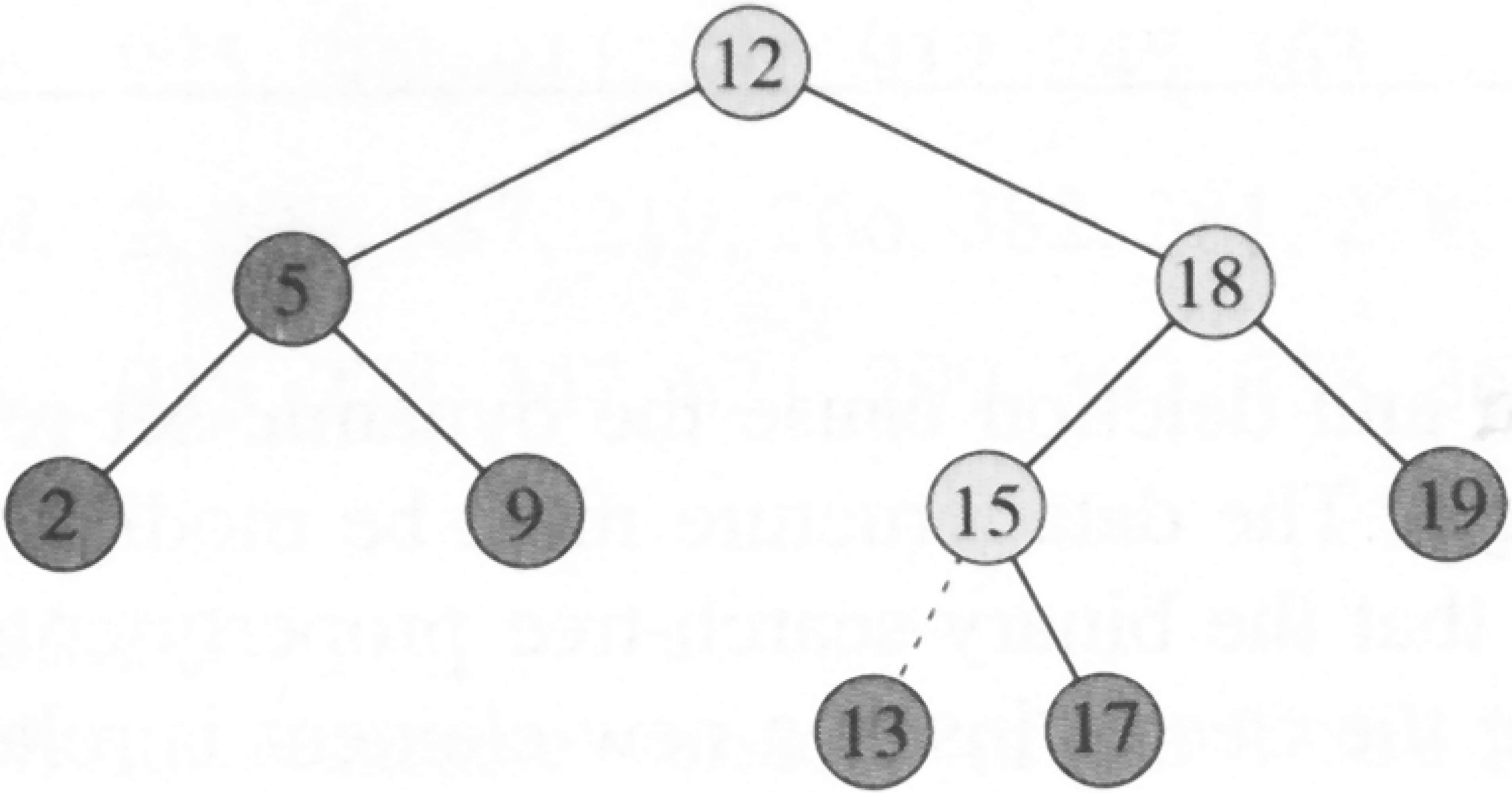
12.3 Insertion and deletion

Insertion

Tree-Insert(T,z)

```
y = NIL
x = T.root
while x ≠ NIL
      do y = x
           if z.key < x.key
                then x = x.left
                else x = x.right
z.parent = y
if y == NIL
      T.root = z
else if z.key < y.key
           y.left = z
      else y.right = z
```

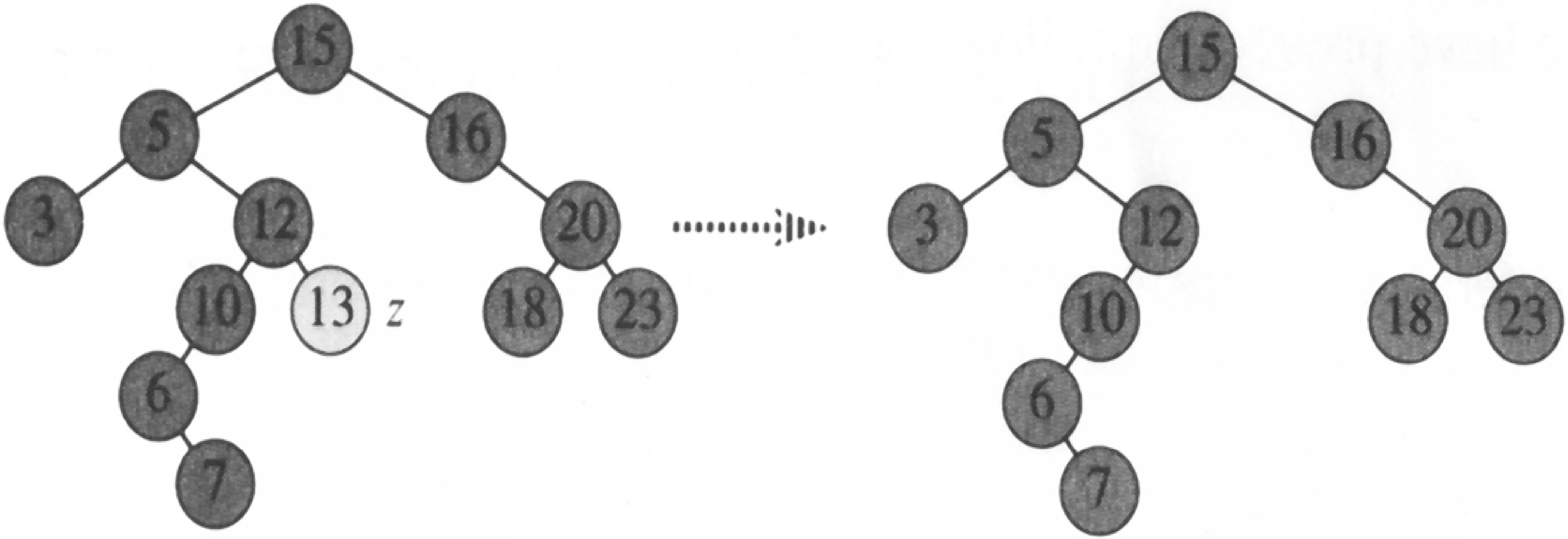
Inserting an item with key 13 into a binary search tree



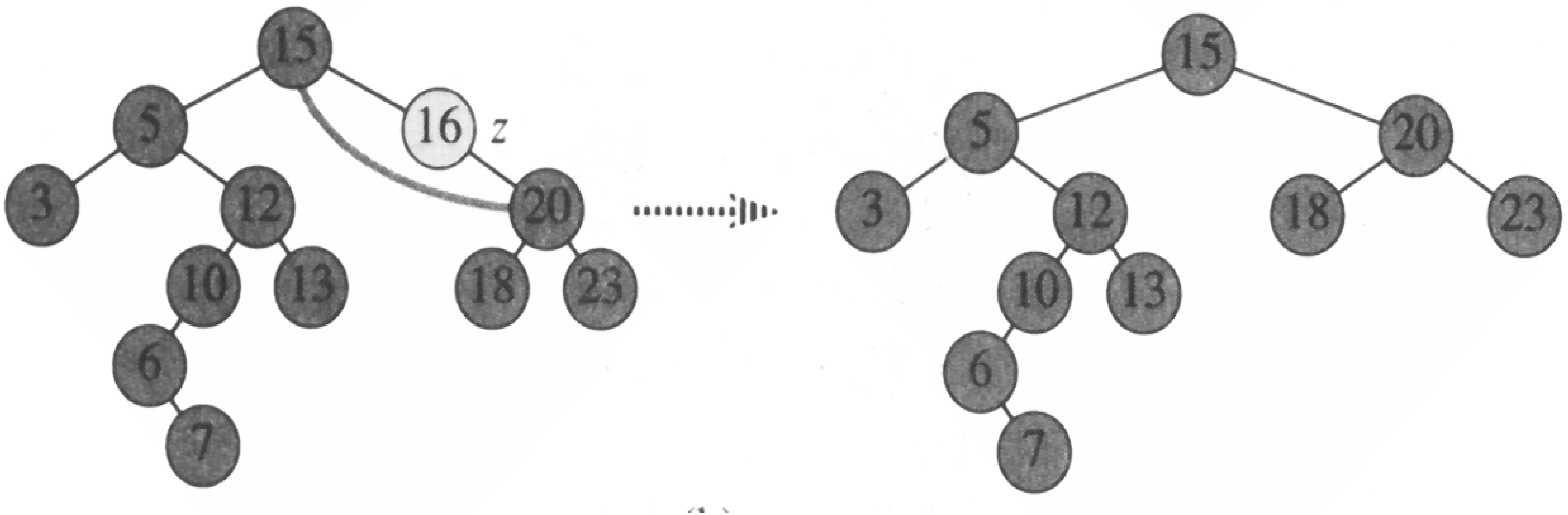
Deletion

```
Tree-Delete(T,z)
if z.left == NIL or z.right == NIL
    y = z
else y = Tree-Successor(z)
if y.left ≠ NIL
    x = y.left
else x = y.right
if x ≠ NIL
    x.parent = y.parent
if y.parent == NIL
    T.root = x
else if y == y.parent.left
    y.parent.left = x
else y.parent.right = x
if y ≠ z
    z.key = y.key
    copy y's satellite data into z
return y
```

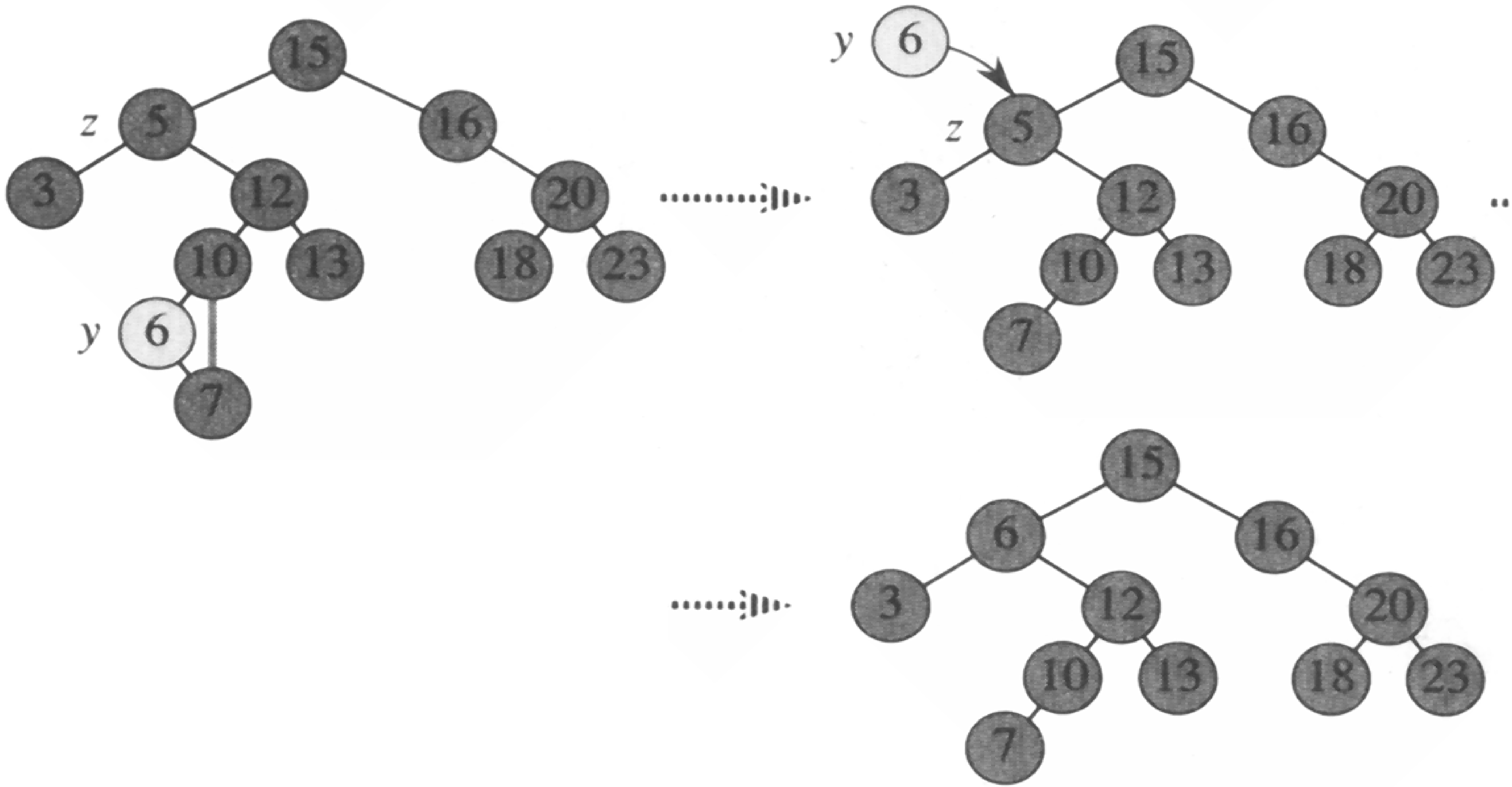
Deletion: node to be deleted, z, has no children



Deletion: z has only one child



Deletion: z has two children



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

Reading Assignments

You should read and understand the insertion and deletion operations and algorithms from Section 12.3. The Tree-Delete algorithm in the text is different from the one discussed in class in its mechanics, not strategy, so figuring out how TRANSPLANT(T,u,v) (p. 296) and TREE-DELETE(T,z) (p. 298) work will give you practice in understanding algorithm mechanics and enable you to see how the same strategy can be implemented by two different algorithms – the one in these slides and the one in the text.

Ch. 12 Reading Assignments

Sections 12.1 – 12.3

Ch. 12 Thinking Assignments

Problems 12.1-1 through 12.1-4

Problems 12.2-1 through 12.2-4

Problems 12.3-1 through 12.3-4

AUBURN UNIVERSITY