Data Structure and Algorithm

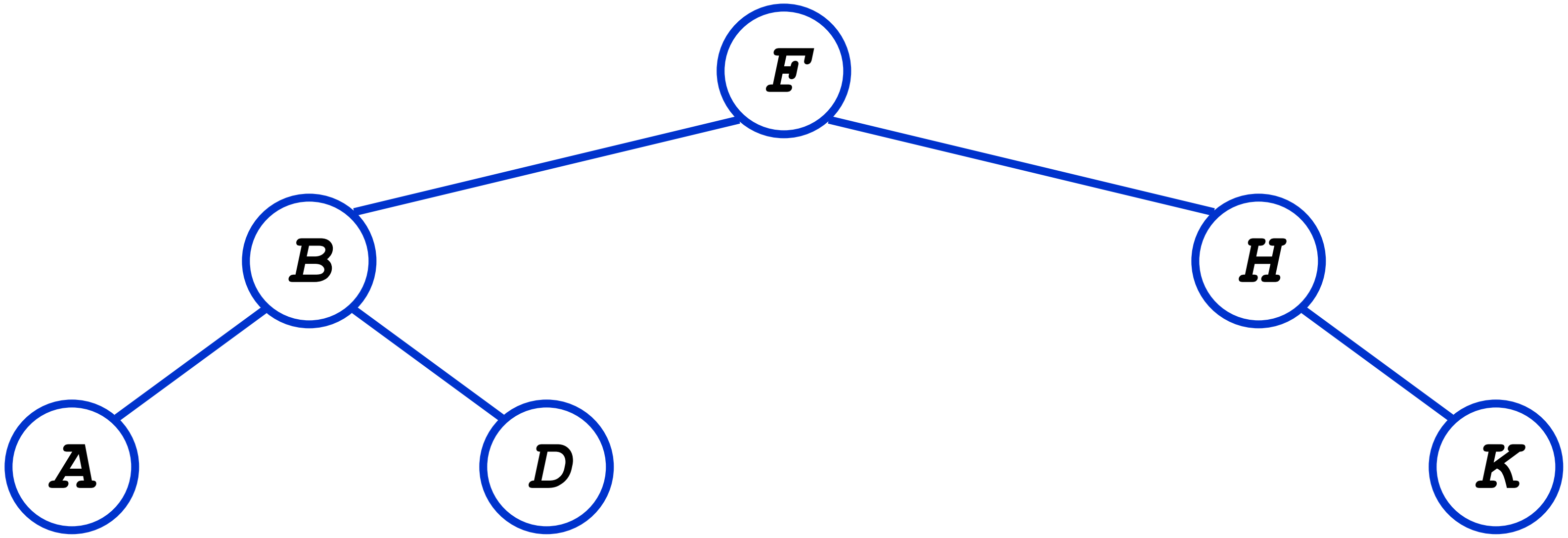
Binary Search Trees

Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- Each element has:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

Binary Search Trees…

- BST property:
  key[leftSubtree(x)] ≤ key[x] ≤ key[rightSubtree(x)]
- Example:



Binary Search Trees :Example

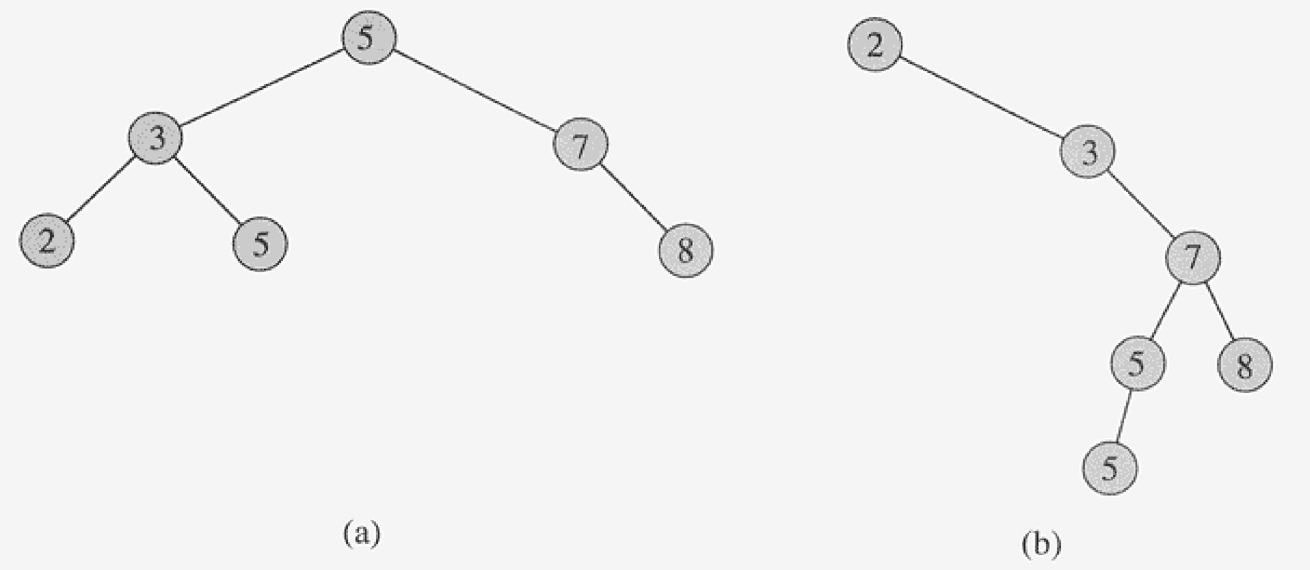
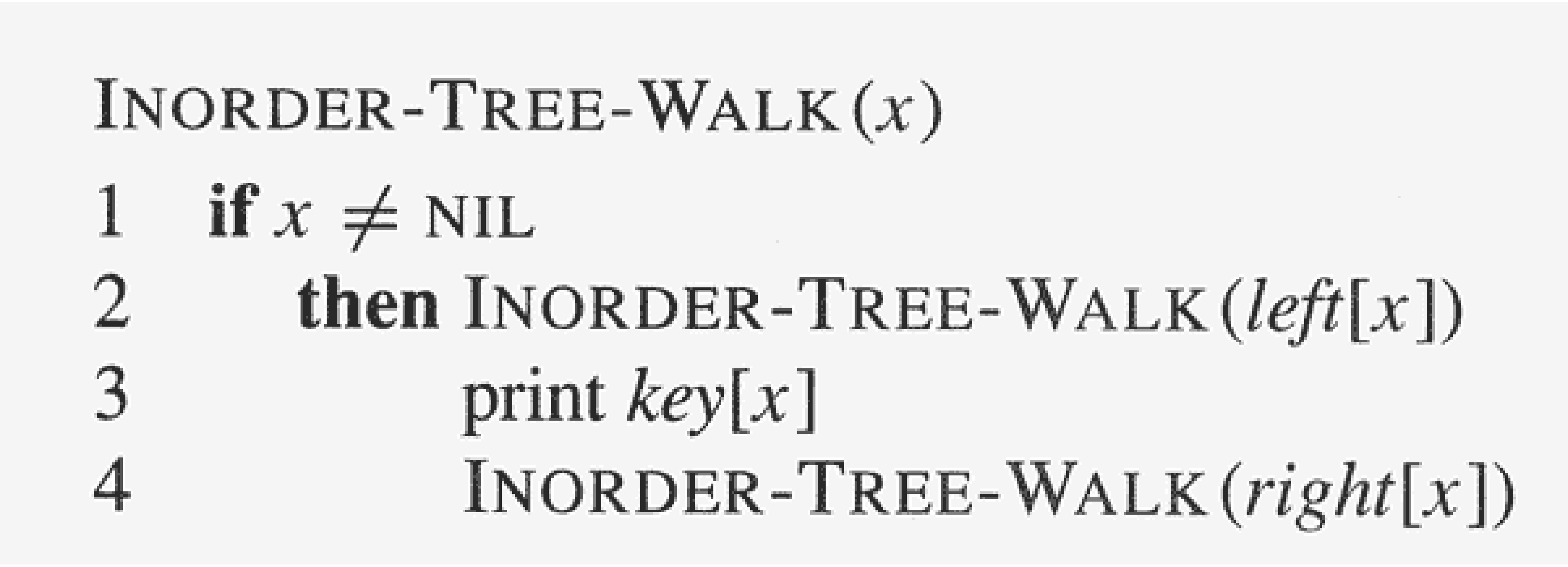


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional 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 Tree Walk

- *What does the following code do?*

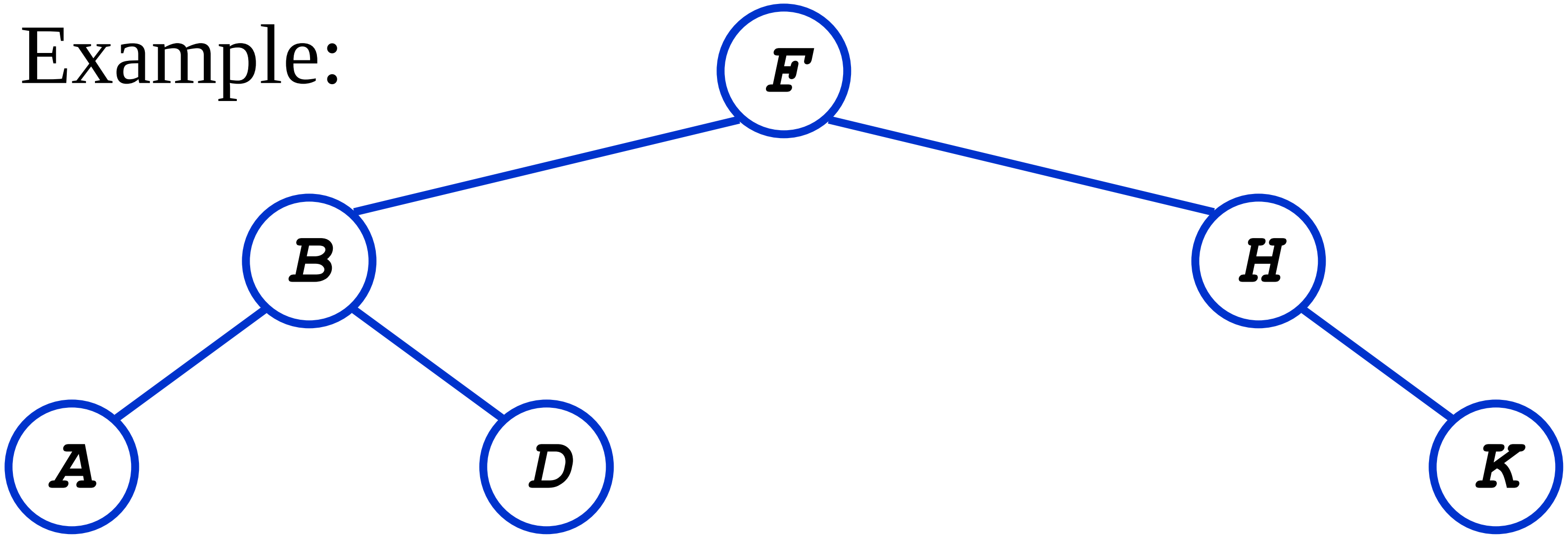


```
INORDER-TREE-WALK($x$)
if $x \neq$ NIL
    then INORDER-TREE-WALK($left[x]$)
        print $key[x]$
        INORDER-TREE-WALK($right[x]$)
```

- A: prints elements in sorted (increasing) order

Inorder Tree Walk…

- Example:



- *How long will a tree walk take?*
- *Prove that inorder walk prints in monotonically increasing order*

Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
TREE-SEARCH($x$, $k$)
if $x$ = NIL or $k$ = $key[x]$
    then return $x$
if $k < key[x]$
    then return TREE-SEARCH($left[x]$, $k$)
    else return TREE-SEARCH($right[x]$, $k$)
```

BST Search: Example

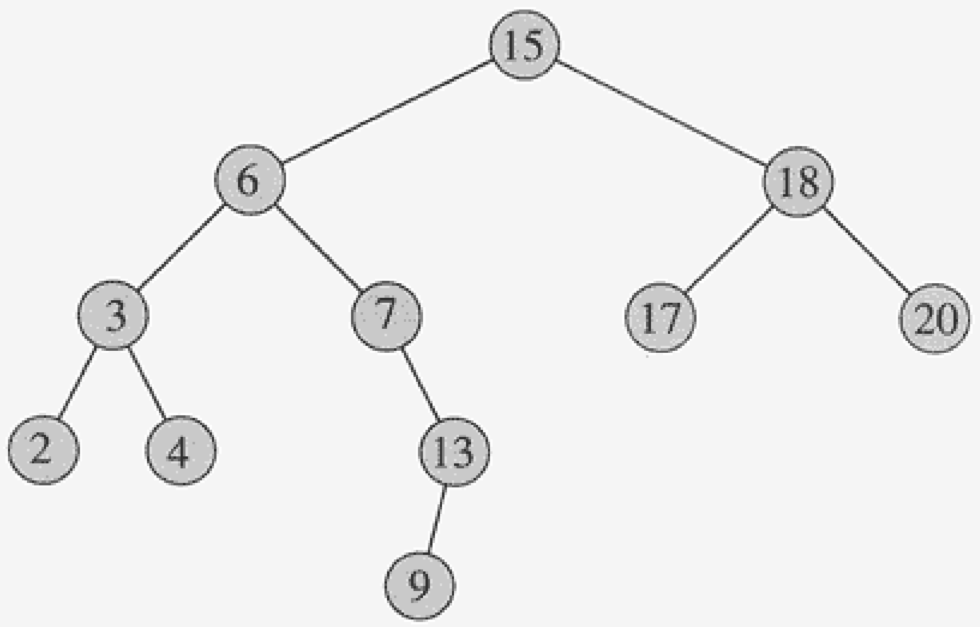
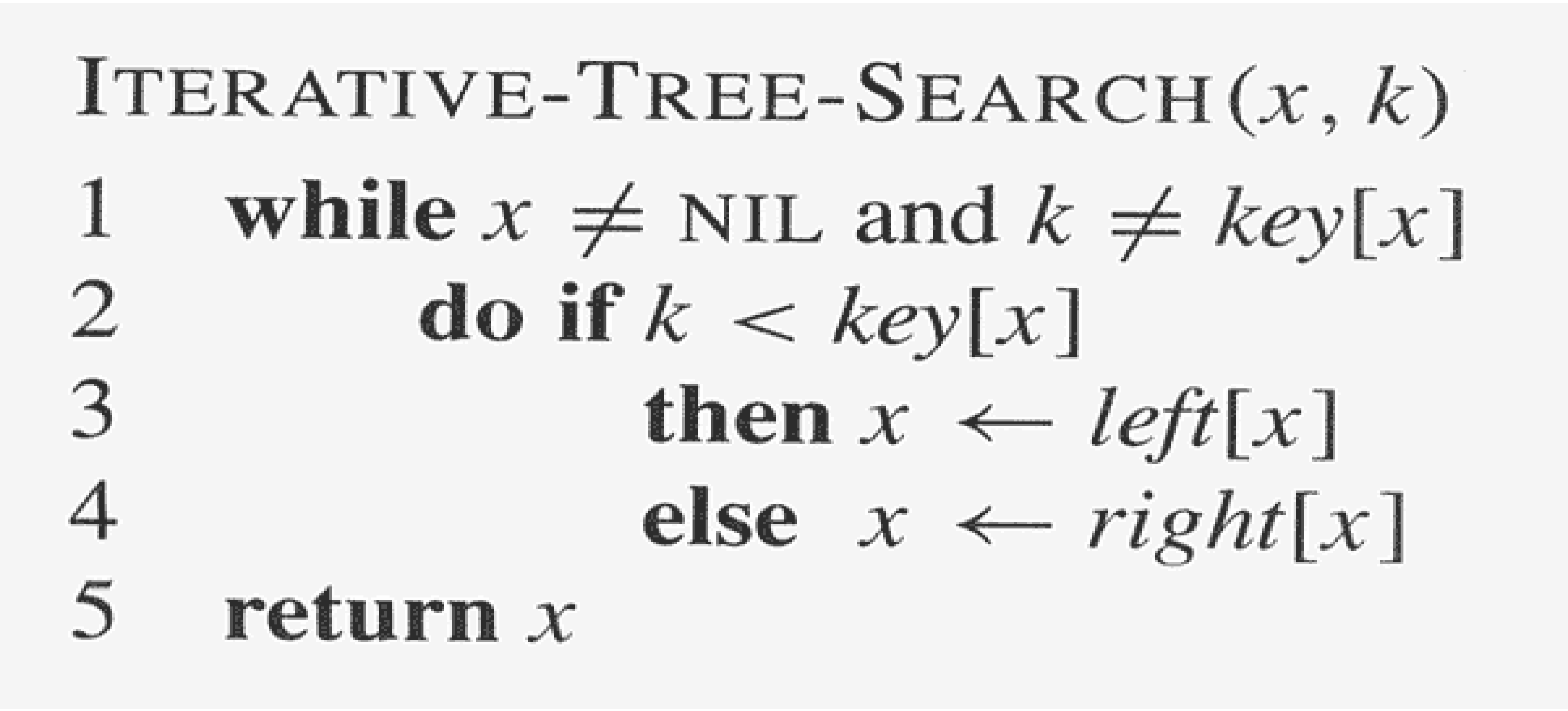


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path 15 → 6 → 7 → 13 from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Operations on BSTs: Search…

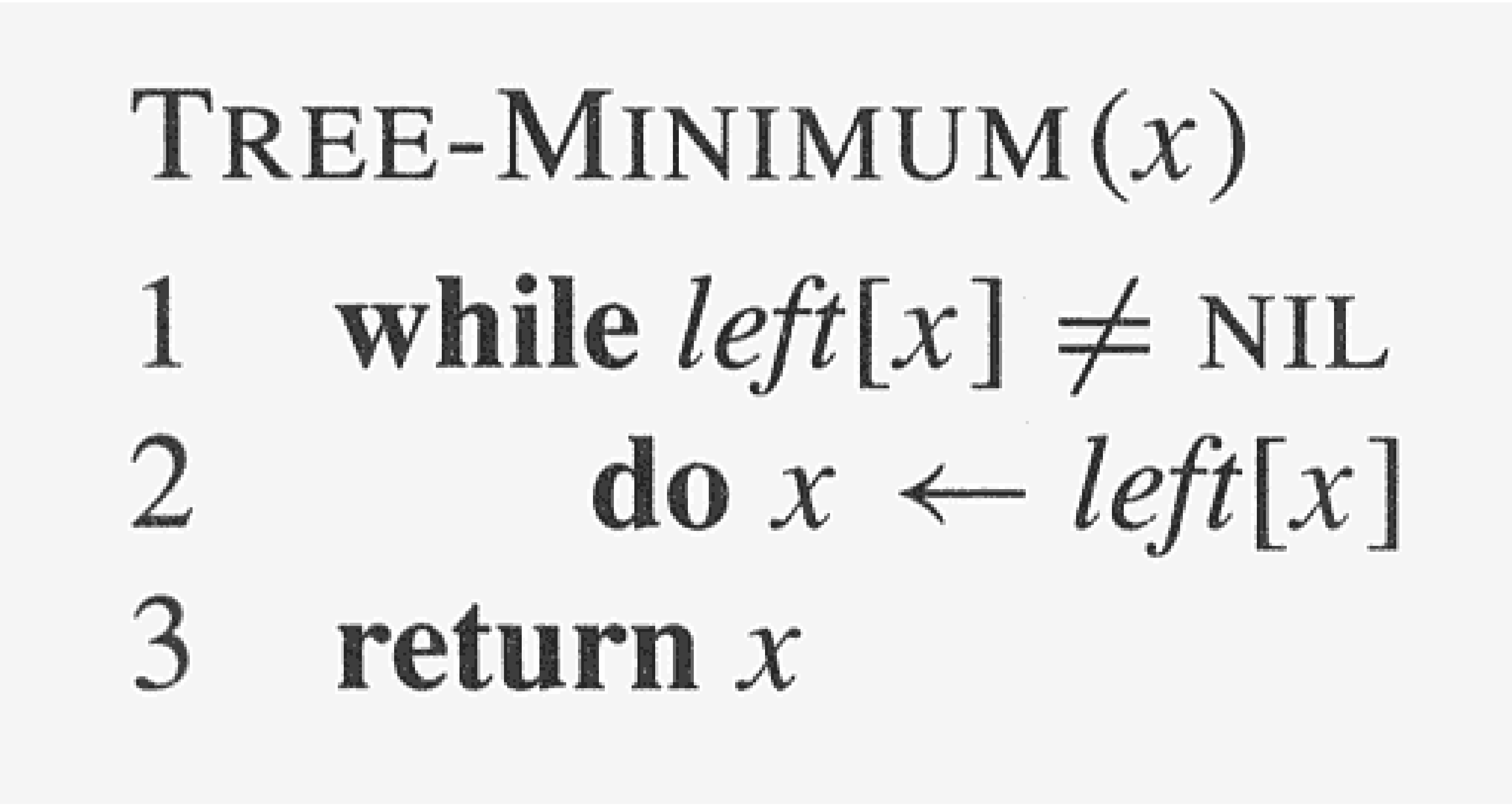
- Here’s another function that does the same:



```
ITERATIVE-TREE-SEARCH($x, k$)
    while $x \neq$ NIL and $k \neq key[x]$
        do if $k < key[x]$
              then $x \leftarrow left[x]$
              else $x \leftarrow right[x]$
    return $x$
```

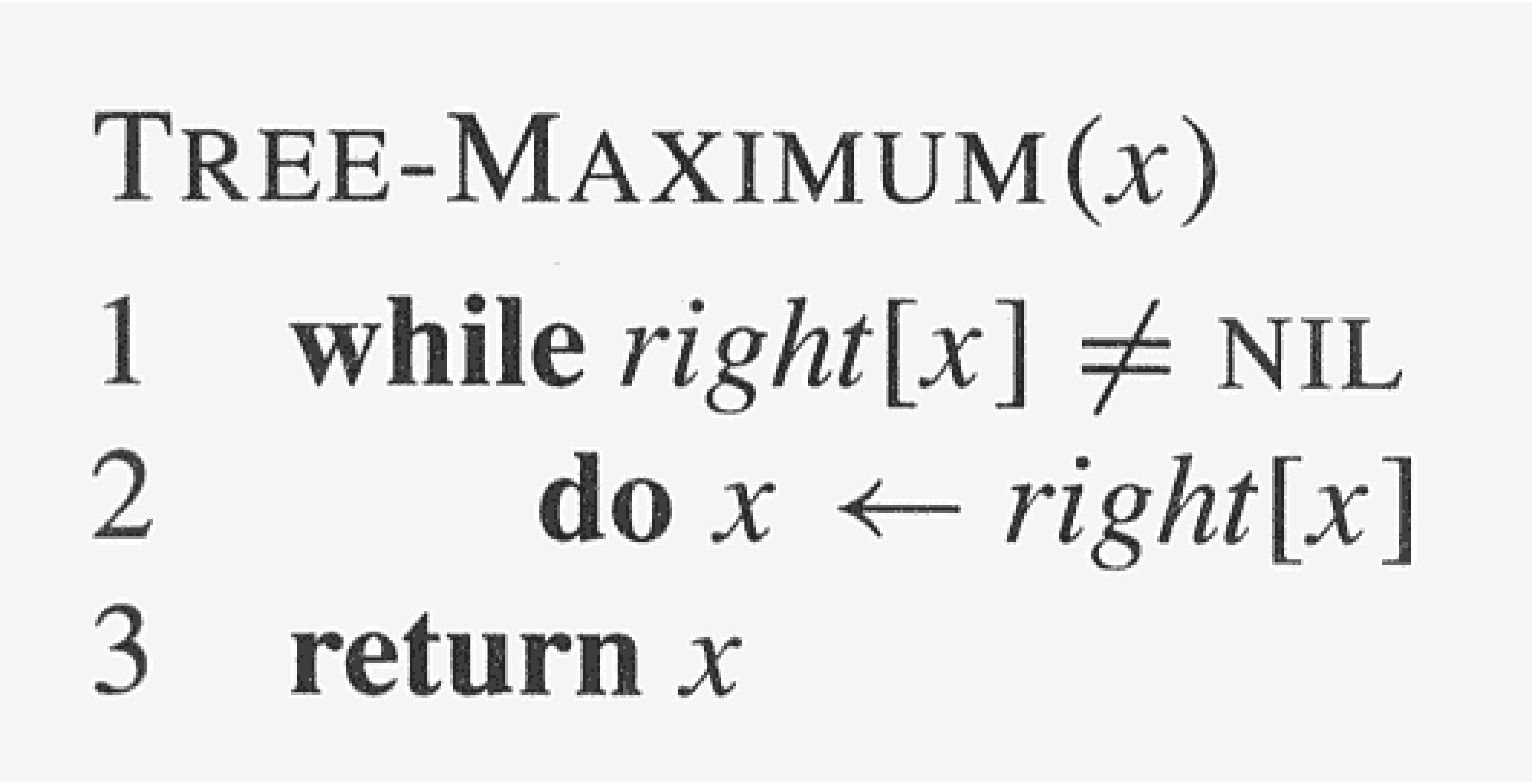
- *Which of these two functions is more efficient?*

Operations on BSTs: Minimum



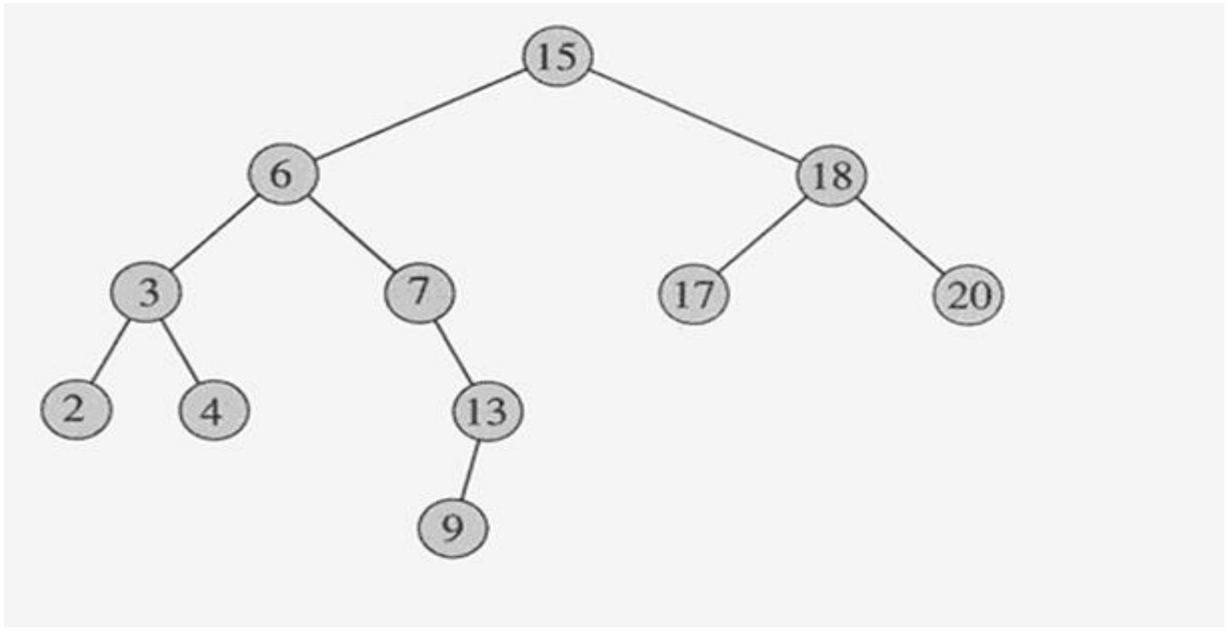
```
TREE-MINIMUM($x$)
1  while $left[x] \neq$ NIL
2      do $x \leftarrow left[x]$
3  return $x$
```

Operations on BSTs: Maximum



```
TREE-MAXIMUM($x$)
1   while $right[x] \neq$ NIL
2        do $x \leftarrow right[x]$
3   return $x$
```

BST Operations: Successor

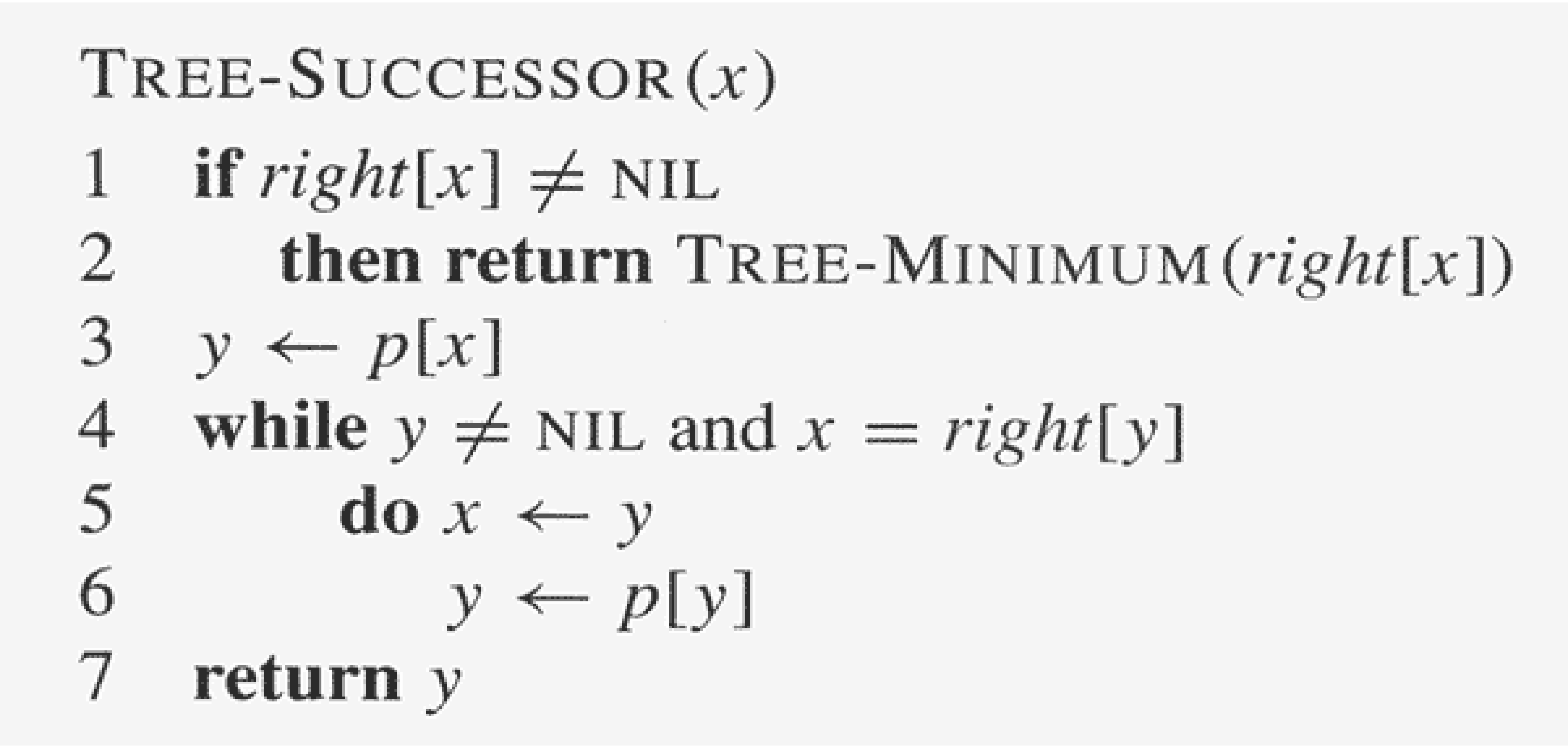


- *What is the successor of node 13? Node 15? Node 20?*
- *What are the general rules for finding the successor of node x? (hint: two cases)*

BST Operations: Successor…

- Two cases:
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.

BST Operations: Successor…



```
TREE-SUCCESSOR($x$)
if $right[x] \neq$ NIL
    then return TREE-MINIMUM($right[x]$)
$y \leftarrow p[x]$
while $y \neq$ NIL and $x = right[y]$
    do $x \leftarrow y$
       $y \leftarrow p[y]$
return $y$
```

Predecessor: similar algorithm

Operations of BSTs: Insert

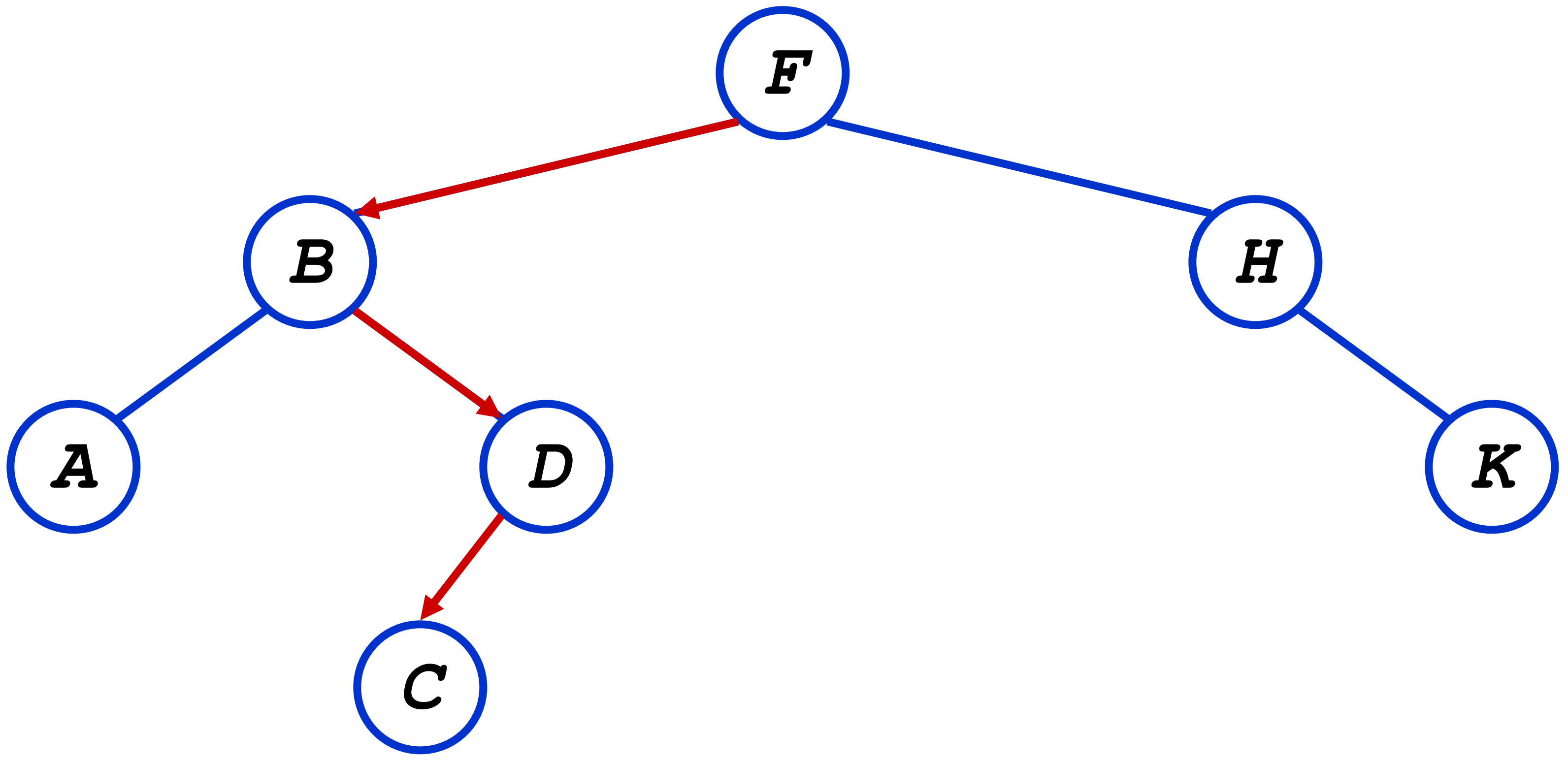
- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)

Operations of BSTs: Insert…

```
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                          ▷ Tree T was empty
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
```

BST Insert: Example

- Example: Insert *C*



BST Search/Insert: Running Time

- *What is the running time of TreeSearch() or TreeInsert()?*
- A: O(h), where h = height of tree
- *What is the height of a binary search tree?*
- A: worst case: h = O(n) when tree is just a linear string of left or right children
  - We'll keep all analysis in terms of h for now
  - Later we'll see how to maintain h = O(lg n)

Sorting With Binary Search Trees

- Informal code for sorting array A of length *n*:

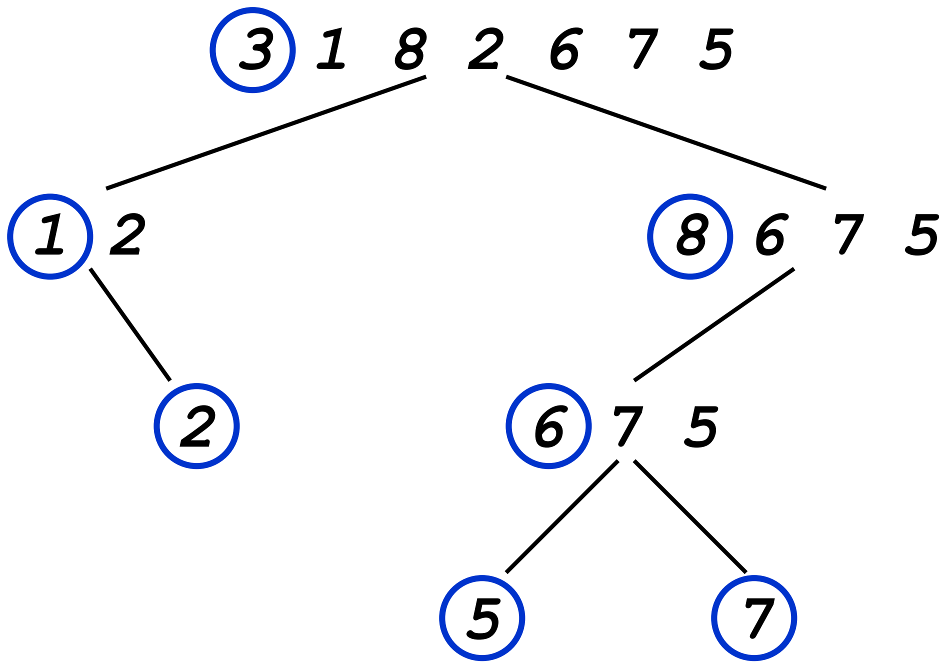
```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    Inorder-Tree-Walk(root);
```

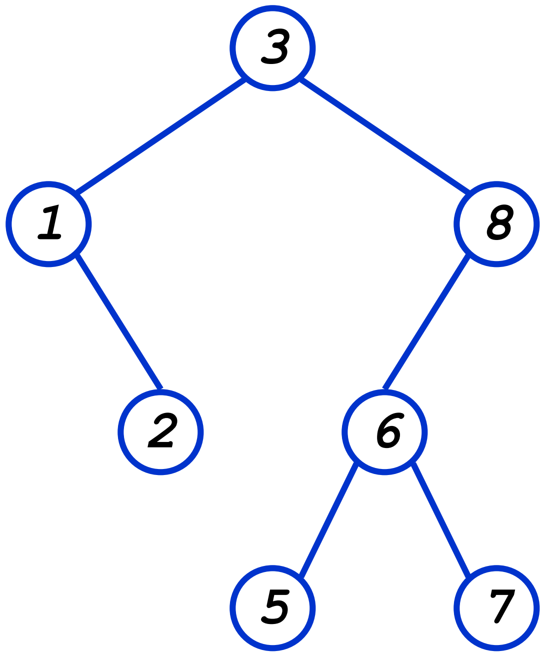
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs…

- Average case analysis
  - It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```





Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as quicksort, different order!
    - Example: consider inserting 5

Sorting with BSTs

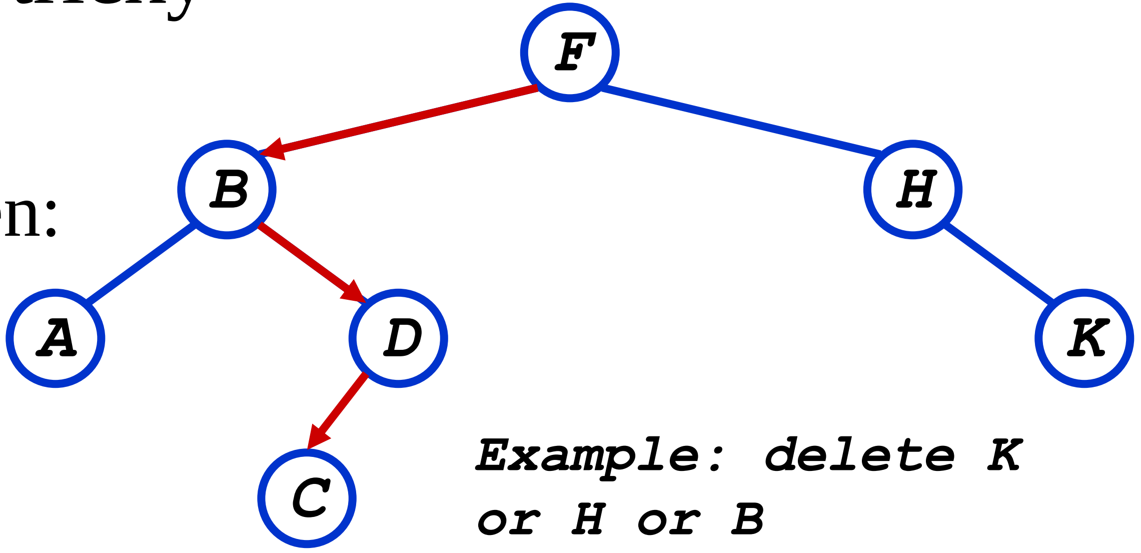
- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTsort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTSort? Why?*
- A: quicksort
  - Better constants
  - Sorts in place
  - Doesn’t need to build data structure

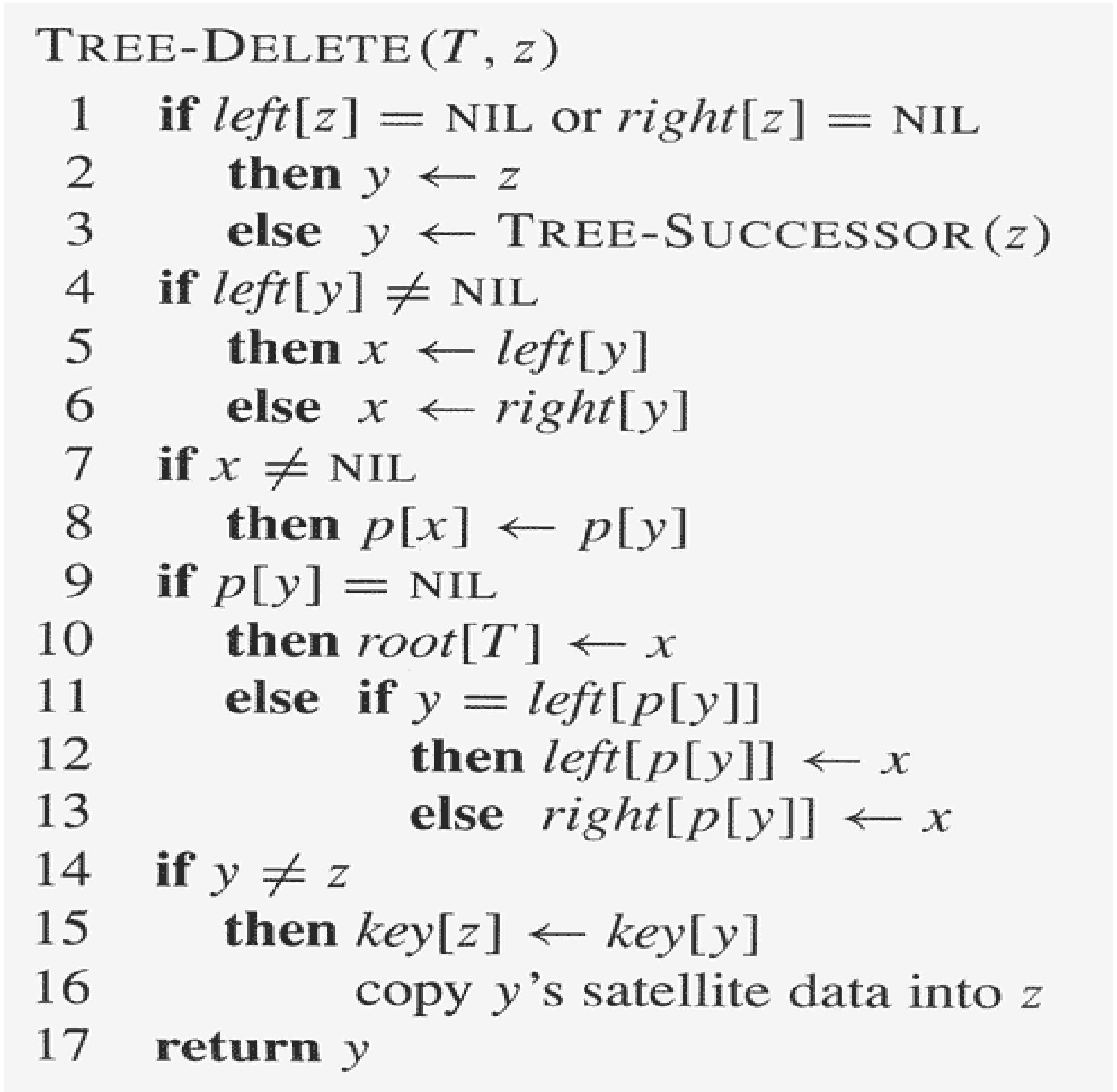
BST Operations: Delete

- Deletion is a bit tricky
- 3 cases:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it



Example: delete K or H or B

BST Operations: Delete…



```
TREE-DELETE$(T, z)$
if $left[z] = $ NIL or $right[z] = $ NIL
    then $y \leftarrow z$
    else $y \leftarrow$ TREE-SUCCESSOR$(z)$
if $left[y] \neq$ NIL
    then $x \leftarrow left[y]$
    else $x \leftarrow right[y]$
if $x \neq$ NIL
    then $p[x] \leftarrow p[y]$
if $p[y] = $ NIL
    then $root[T] \leftarrow x$
    else if $y = left[p[y]]$
            then $left[p[y]] \leftarrow x$
            else $right[p[y]] \leftarrow x$
if $y \neq z$
    then $key[z] \leftarrow key[y]$
         copy $y$'s satellite data into $z$
return $y$
```

BST Delete: Example

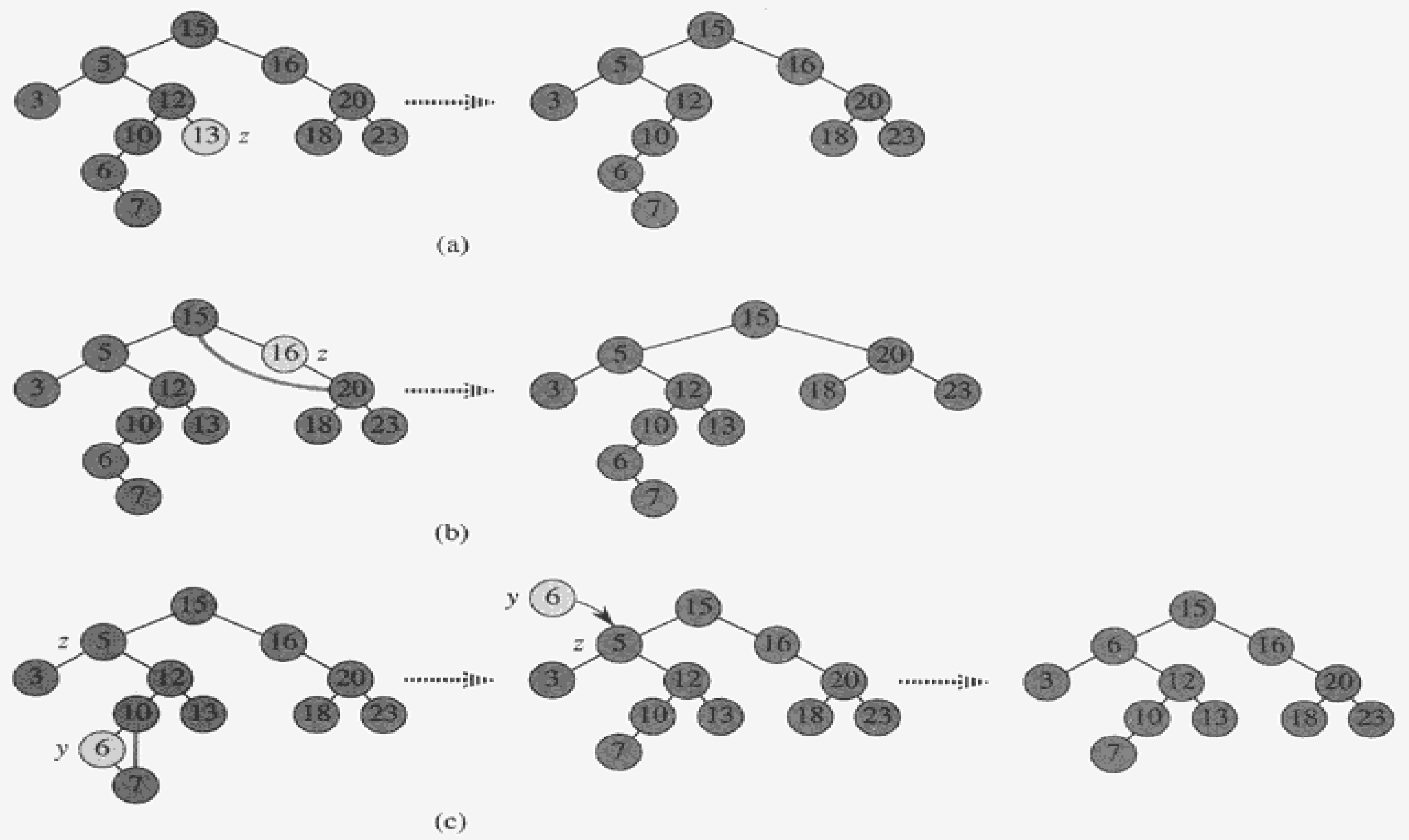


Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

BST Operations: Delete…

- *Why will case 2 always go to case 0 or case 1?*
- A: because when x has 2 children, its successor is the minimum in its right subtree
- *Could we swap x with predecessor instead of successor?*
- A: yes. *Would it be a good idea?*
- A: might be good to alternate

More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*
  - Insert
  - Minimum
  - Extract-Min