Announcements

- **Weekly reading assignment:**

  Chapter 12 & 13, CLRS

- **Homework #4 Due on October 27**

- **Extra Help Session after class on Thursday, 10/27/05**

Binary Search Tree

- Can be represented by a linked data structure in which each node is an object. Each node contains fields: *key, left, right and parent*.
- The keys are stored in such a way that they satisfy the *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 $key[y] \le key[x]$. If y is a node in the right subtree of x, then $key[x] \le key[y]$.

- Example: Input is D, A, B, H, K, F

What are BSTs good for?

- Building priority queues
  - Extraction of min/max
- Sorting
  - Searching
  - Insertion
  - Deletion
- Querying
  - Database
  - Geometric operations

Inorder-Tree-Walk(x)

1. if $x \neq NIL$
2. then Inorder-Tree-Walk(*left*[p])
3. print *key*[x]
4. Inorder-Tree-Walk(*right*[p])

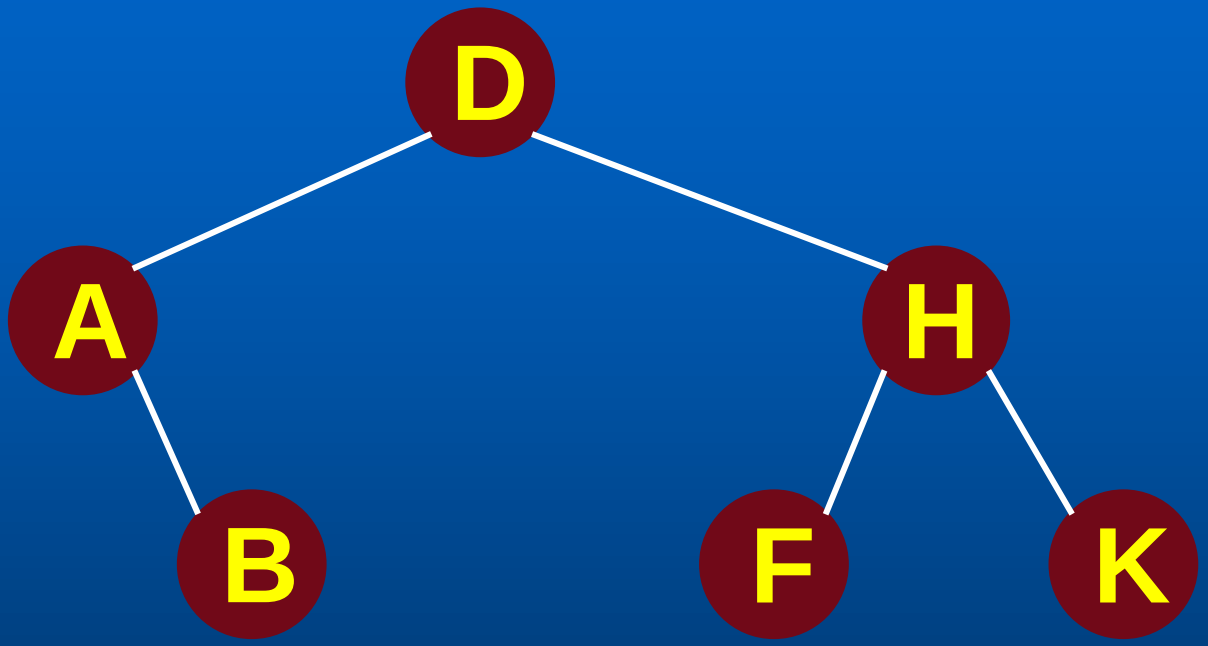
Q: How long does the walk take?

Q: Can you prove its correctness?

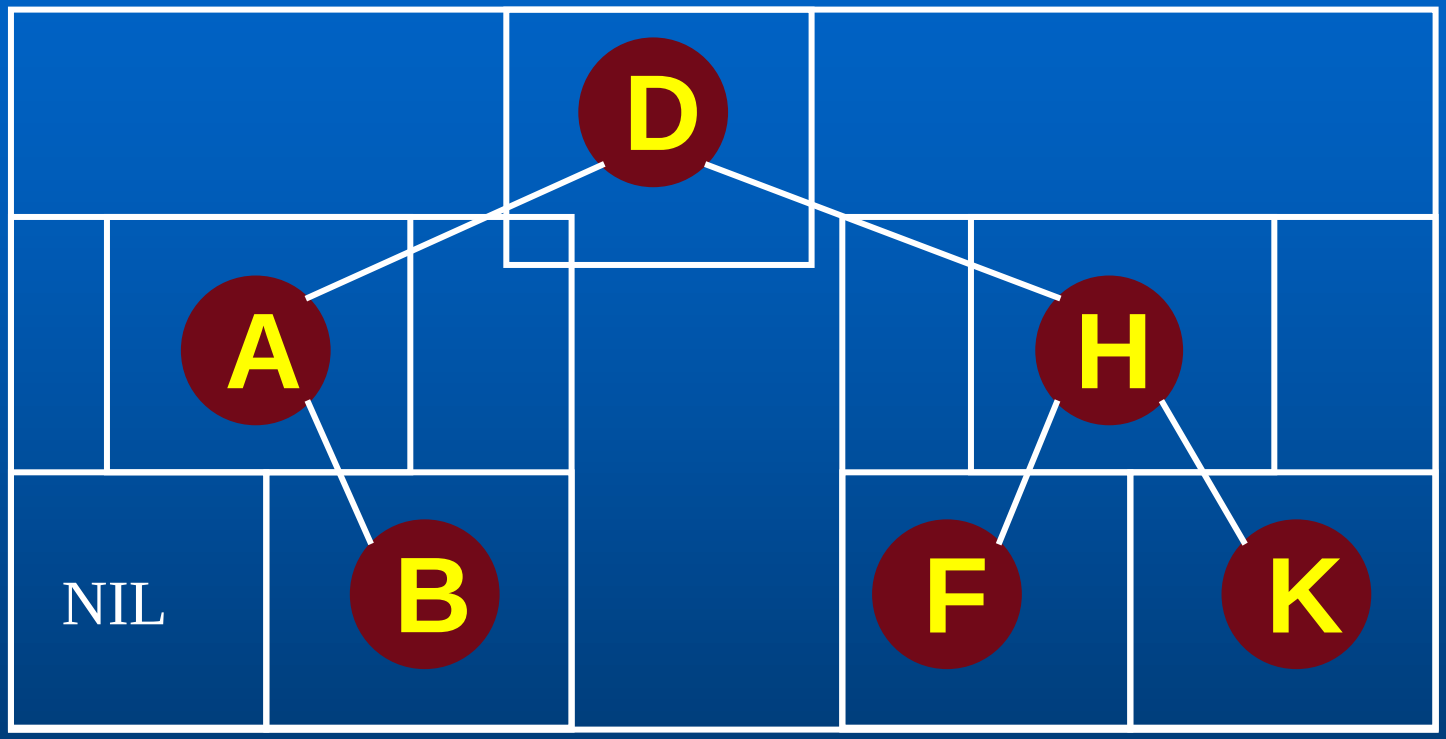
Proving Correctness

- Must prove that it prints all elements, in order, and that it terminates
- Induction on size of tree. Size=1: Easy
- Size >1:
  - Prints left subtree in order by induction
  - Prints root, which comes after all elements in left subtree (still in order)
  - Prints right subtree in order (all elements come after root, so still in order)

Example



Example



A B D F H K

Querying a Binary Search Tree

- All dynamic-set operations such as “search”, “minimum”, “maximum”, “successor” and “predecessor” can be supported in $O(h)$ time.
- For a balanced binary tree, $h = \Theta(\log n)$; for an unbalanced tree that resembles a linear chain of n nodes, then $h = \Theta(n)$ in the worst case.

Querying a Binary Search Tree

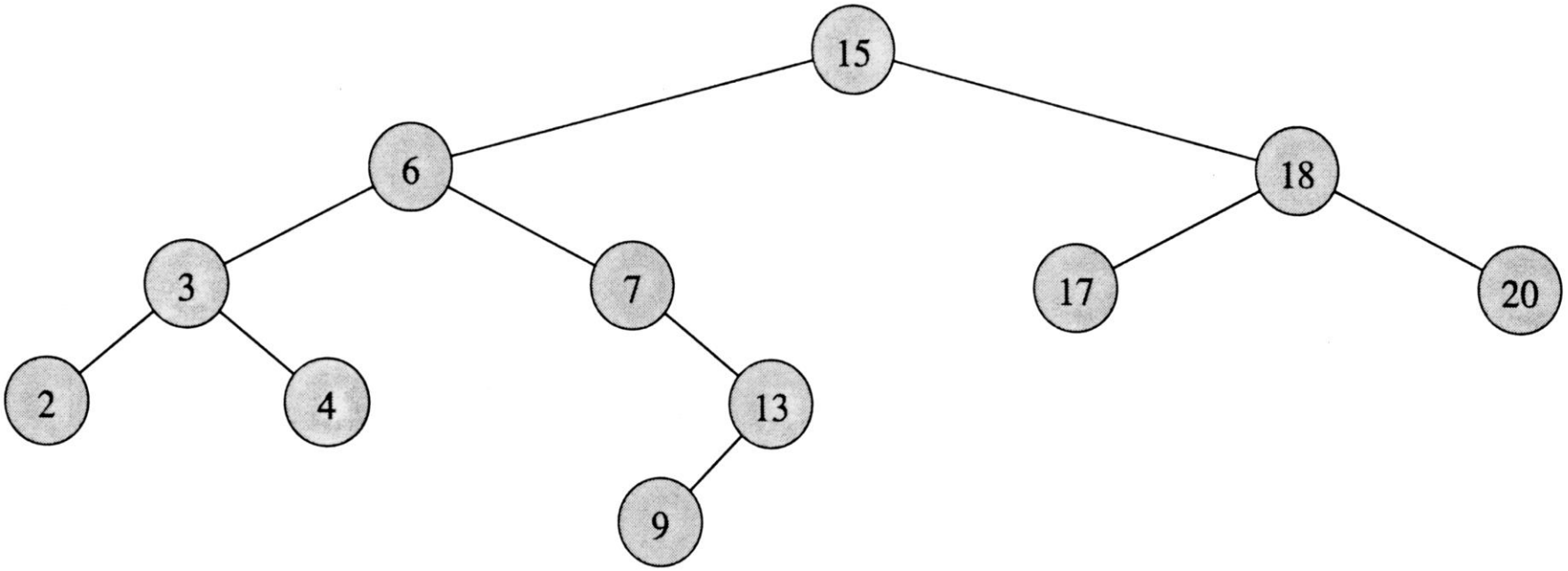


Figure 13.2 Queries on a binary search tree. To search for the key 13 in the tree, the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ is followed from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *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.

Tree-Search(x, k)

1. if $x = NIL$ *or* $k = key[x]$
2. then return x
3. if $k < key[x]$
4. then return Tree-Search($left[x]$, k)
5. else return Tree-Search($right[x]$, k)

Iterative-Tree-Search(x, k)

1. while $x \neq NIL$ and $k \neq key[x]$
2. do if $k < key[x]$
3. then $x \leftarrow left[x]$
4. else $x \leftarrow right[x]$
5. return x

Finding Min & Max

Minimum(x)

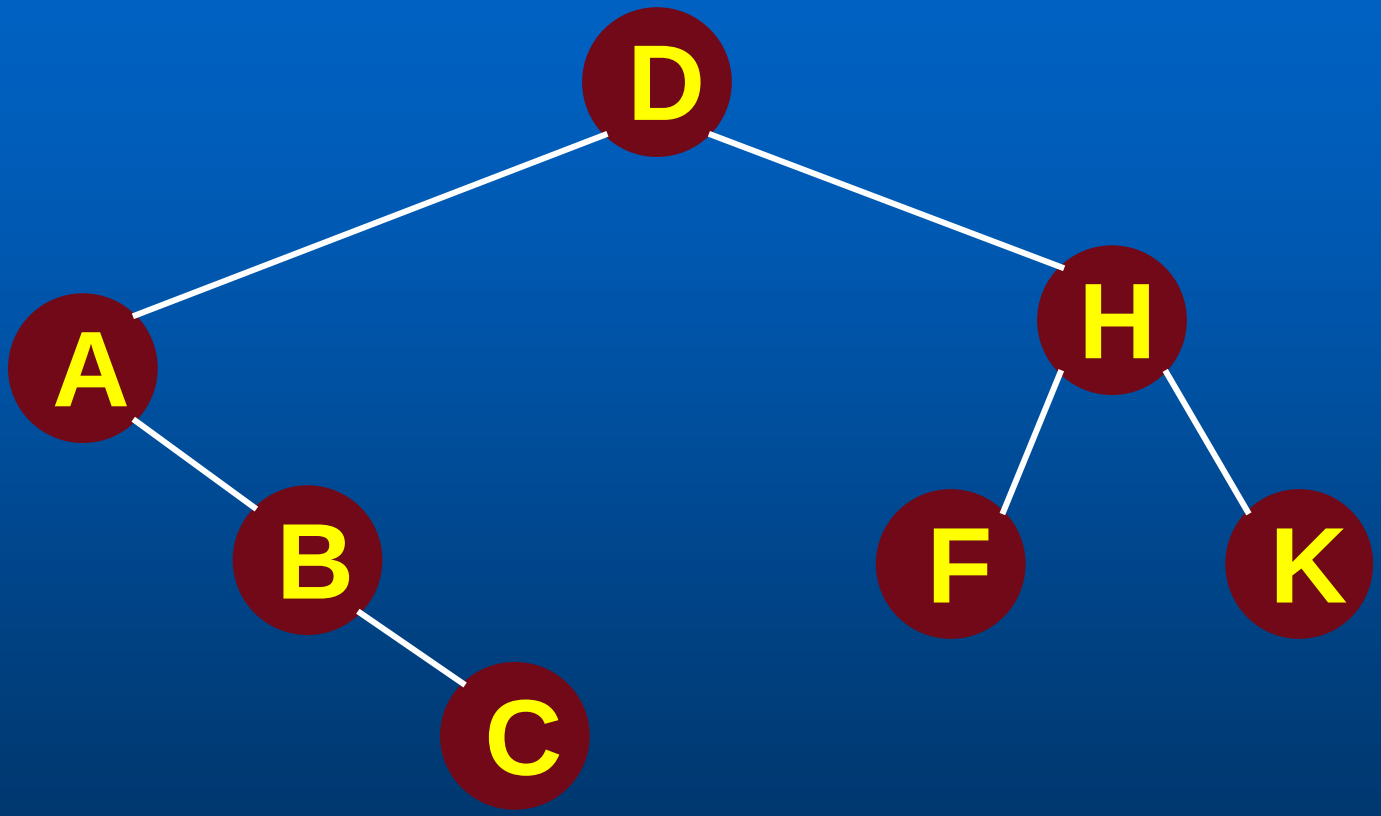
1. while $left[x] \neq NIL$
2. do $x \leftarrow left[x]$
3. return x

Maximum(x)

1. while $right[x] \neq NIL$
2. do $x \leftarrow right[x]$
3. return x

Q: How long do they take?

Tree Successor



Successor(x) in Words

if *right(x)* **is not empty**

then return minimum in right subtree

else keep going up and leftward on the tree to the node where it makes the first right turn (i.e., find the lowest ancestor whose left child is also an ancestor)

Tree-Successor (x)

1. if $right[x] \neq NIL$
2. then return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. while $y \neq NIL$ and $x = right[y]$
5. do $x \leftarrow y$
6. $y \leftarrow p[y]$
7. return y

Think about predecessor ……

Node Insertion

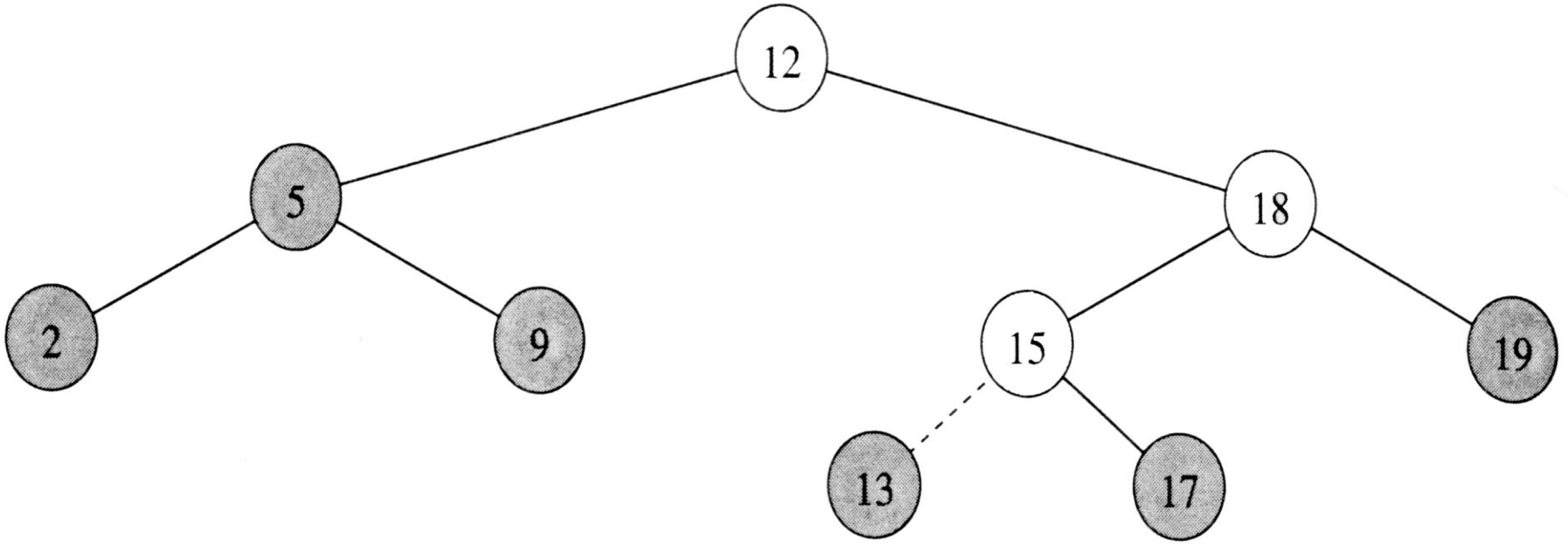


Figure 13.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

```
TREE-INSERT(T, z)
 1   y ← NIL
 2   x ← root[T]
 3   while x ≠ NIL
 4         do y ← x
 5            if key[z] < key[x]
 6               then x ← left[x]
 7               else x ← right[x]
 8   p[z] ← y
 9   if y = NIL
10      then root[T] ← z
11      else if key[z] < key[y]
12              then left[y] ← z
13              else right[y] ← z
```

Analysis on Insertion

- Initialization: $O(1)$
- While loop in lines 3-7 causes 2 pointers (x is the candidate location to insert z, and y is the parent of x) to move down the tree till we reach a leaf ($x = \text{NIL}$). This runs in $O(h)$ time on a tree of height h.
- Lines 8-13 insert the value: $O(1)$

$\Rightarrow$ TOTAL: $O(h)$ time to insert a node

Exercise: Sorting Using BSTs

Sort (A)

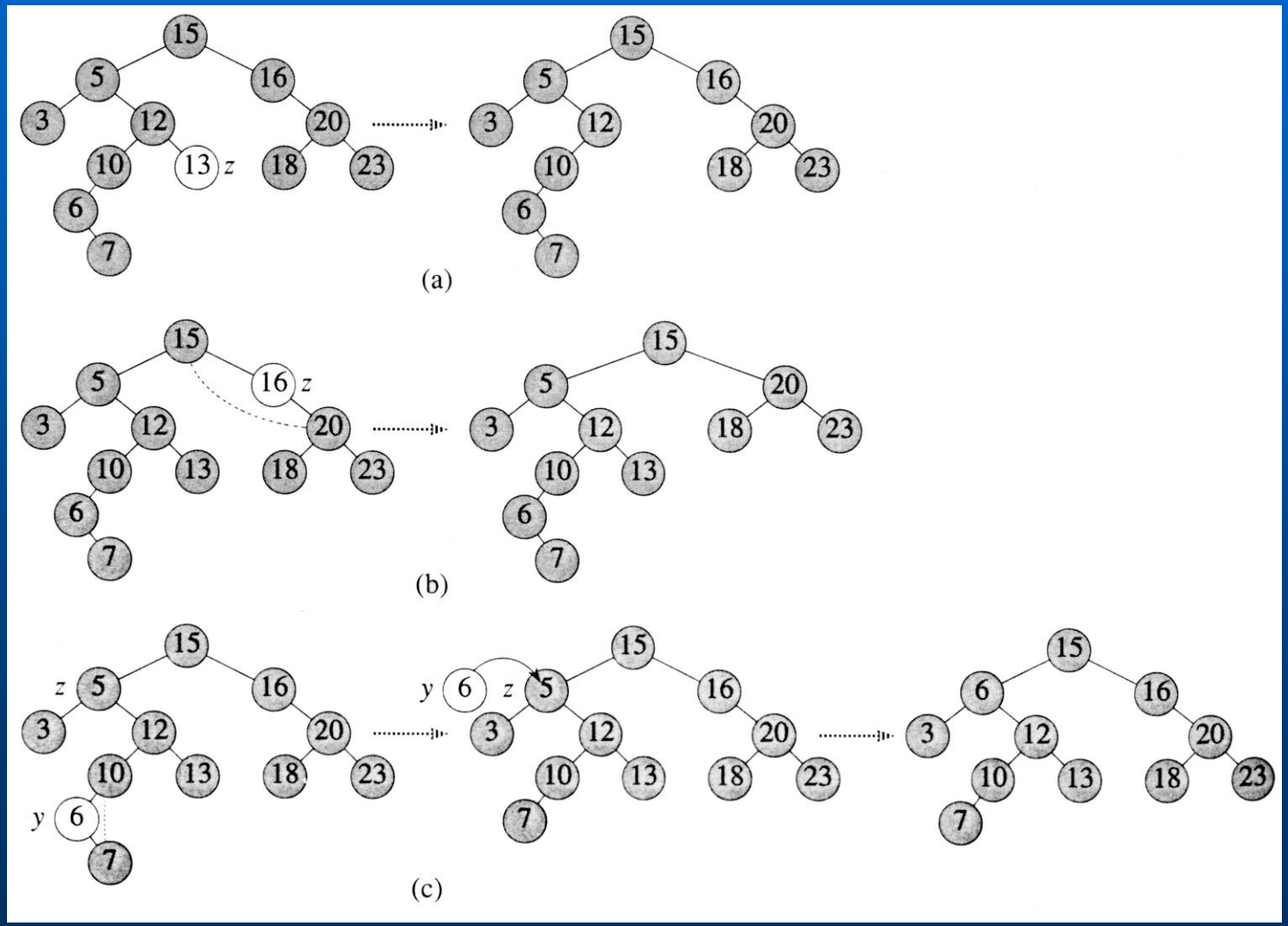
for *i* = *1* **to** *n*

do tree-insert(A[*i*])

inorder-tree-walk(*root*)

- What are the worst case and best case running times?
- In practice, how would this compare to other sorting algorithms?

Node Deletion



Tree-Delete (T, x)

if x has no children ♦ case 0
  then remove x
if x has one child ♦ case 1
  then make $p[x]$ link to child
if x has two children ♦ case 2
  then swap x with its successor
    perform case 0 or case 1 to delete it

⇒ TOTAL: $O(h)$ time to delete a node

Alternate Form of Case 2

- Delete successor of x (but save the data)
- Overwrite x with data from successor
- Equivalent; steps in different order

Correctness of Tree-Delete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  - Because when x has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child).
- Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree.