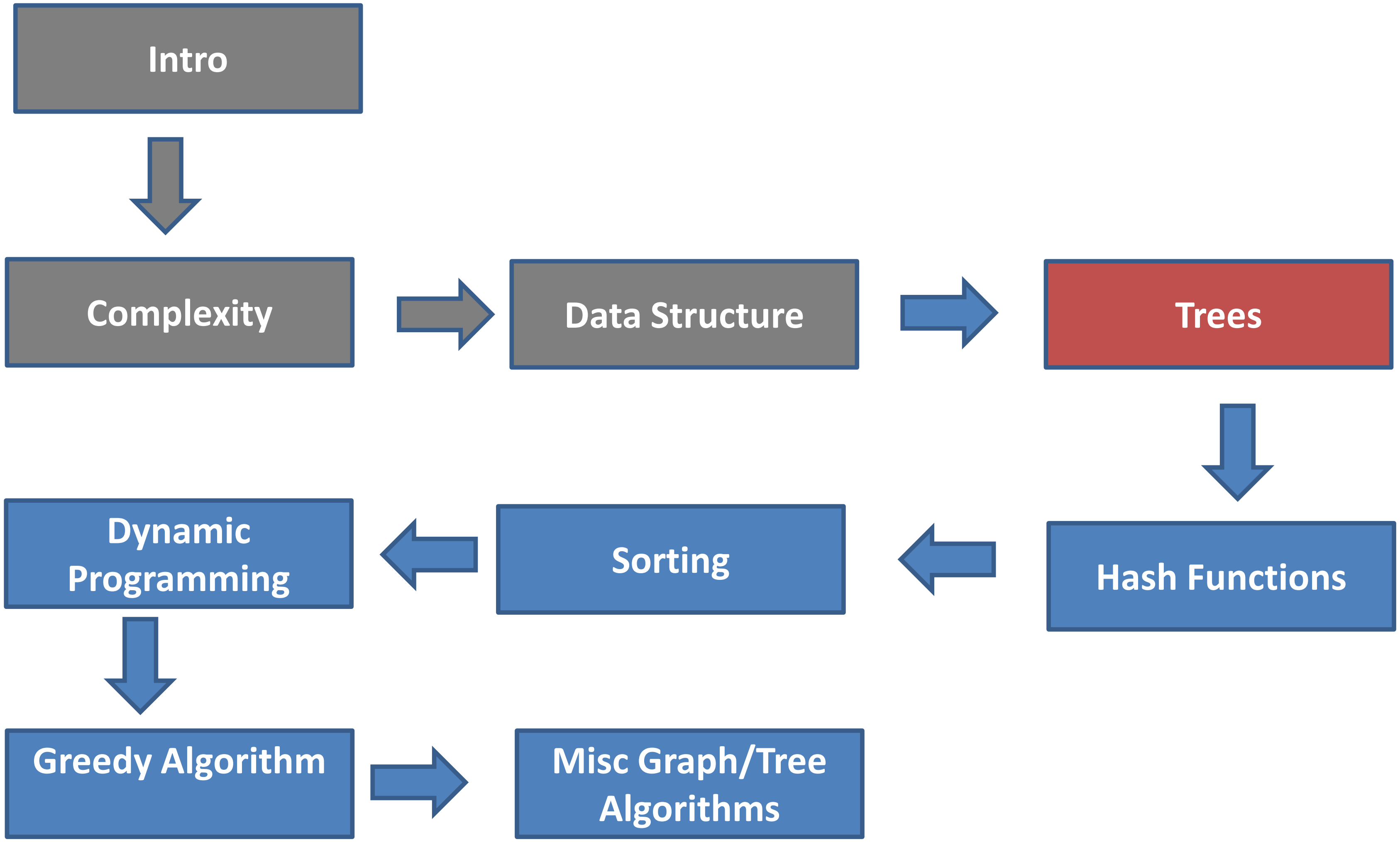
An Introduction to Algorithms

By

Hossein Rahmani

h_rahmani@iust.ac.ir
http://webpages.iust.ac.ir/h_rahmani/


Intro
Complexity
Data Structure
Trees
Hash Functions
Sorting
Dynamic Programming
Greedy Algorithm
Misc Graph/Tree Algorithms

Binary Trees

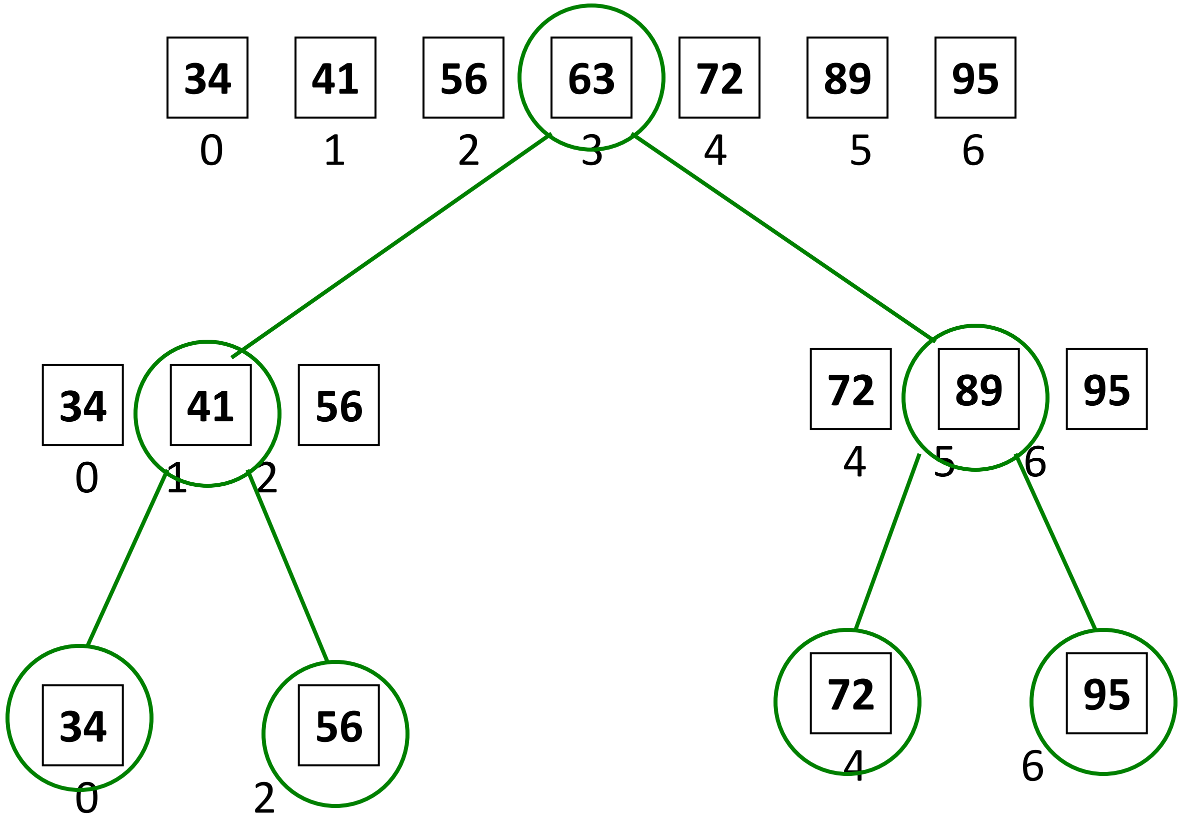
- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Binary Search Trees

- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
- This data organization leads to O(log n) complexity for searches, insertions and deletions in certain types of the BST (balanced trees).
  - O(h) in general

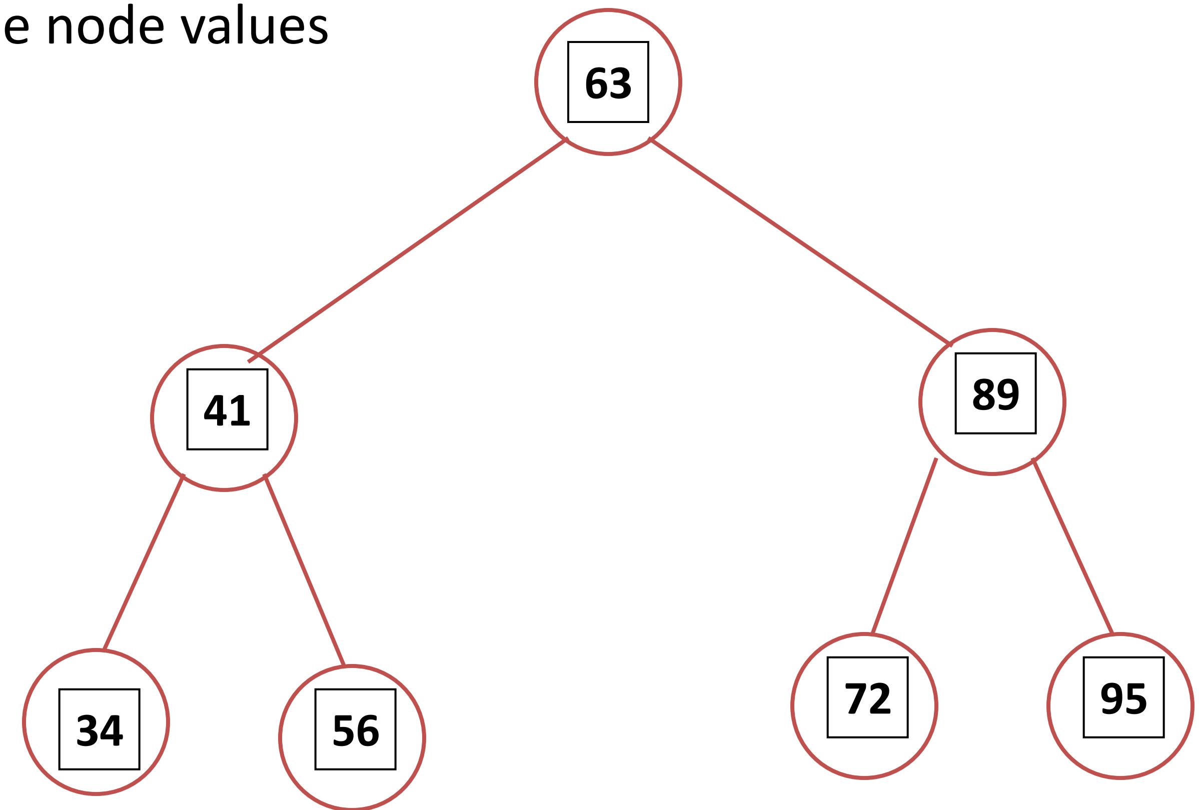
Binary Search Algorithm

Binary Search algorithm of an array of *sorted* items reduces the search space by one half after each comparison



Organization Rule for BST

- The values in all nodes in the left subtree of a node are less than the node value
- The values in all nodes in the right subtree of a node are greater than the node values

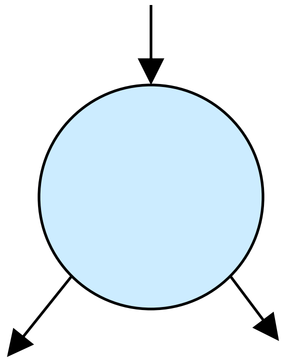


Binary Search Trees

- Data structure that can support Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - Dictionaries.
  - Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

BST – Representation

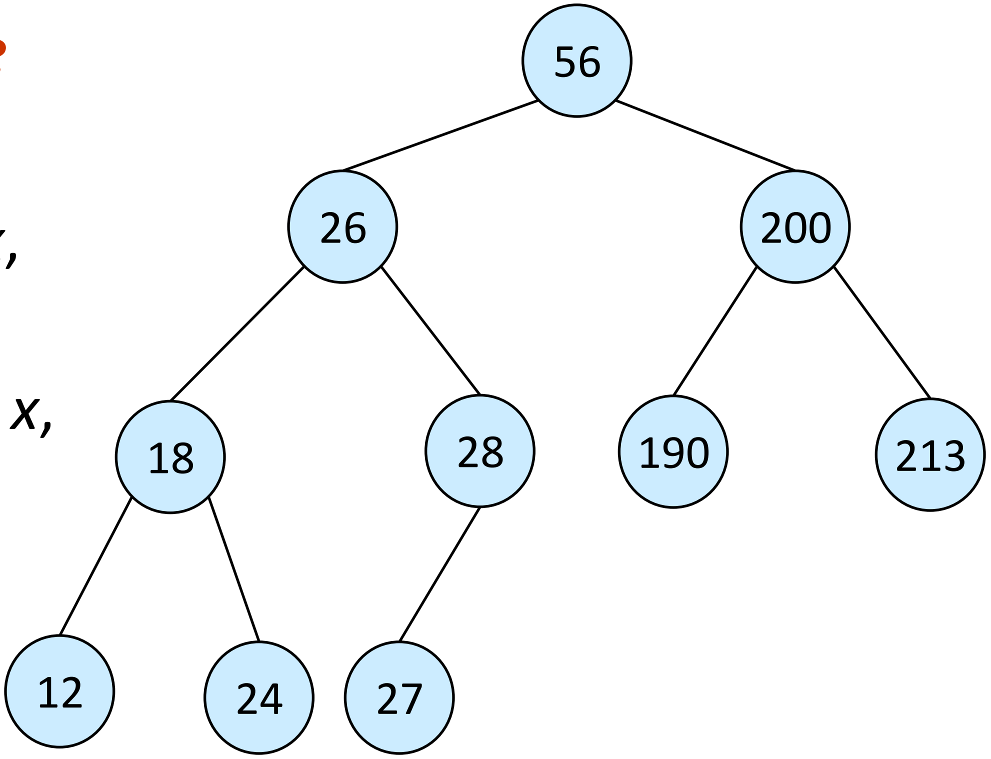
- Represented by a linked data structure of nodes.
- *root*(*T*) points to the root of tree *T*.
- Each node contains fields:
  - *key*
  - *left* – pointer to left child: root of left subtree.
  - *right* – pointer to right child : root of right subtree.
  - *p* – pointer to parent. *p*[*root*[T]] = NIL (optional).



Note: If ***balanced***, an array representation may be better.

Binary Search Tree Property

- Stored keys must satisfy the *binary search tree* property.
  - $\forall$ *y* in left subtree of *x*, then *key*[*y*] $\leq$ *key*[*x*].
  - $\forall$ *y* in right subtree of *x*, then *key*[*y*] $\geq$ *key*[*x*].



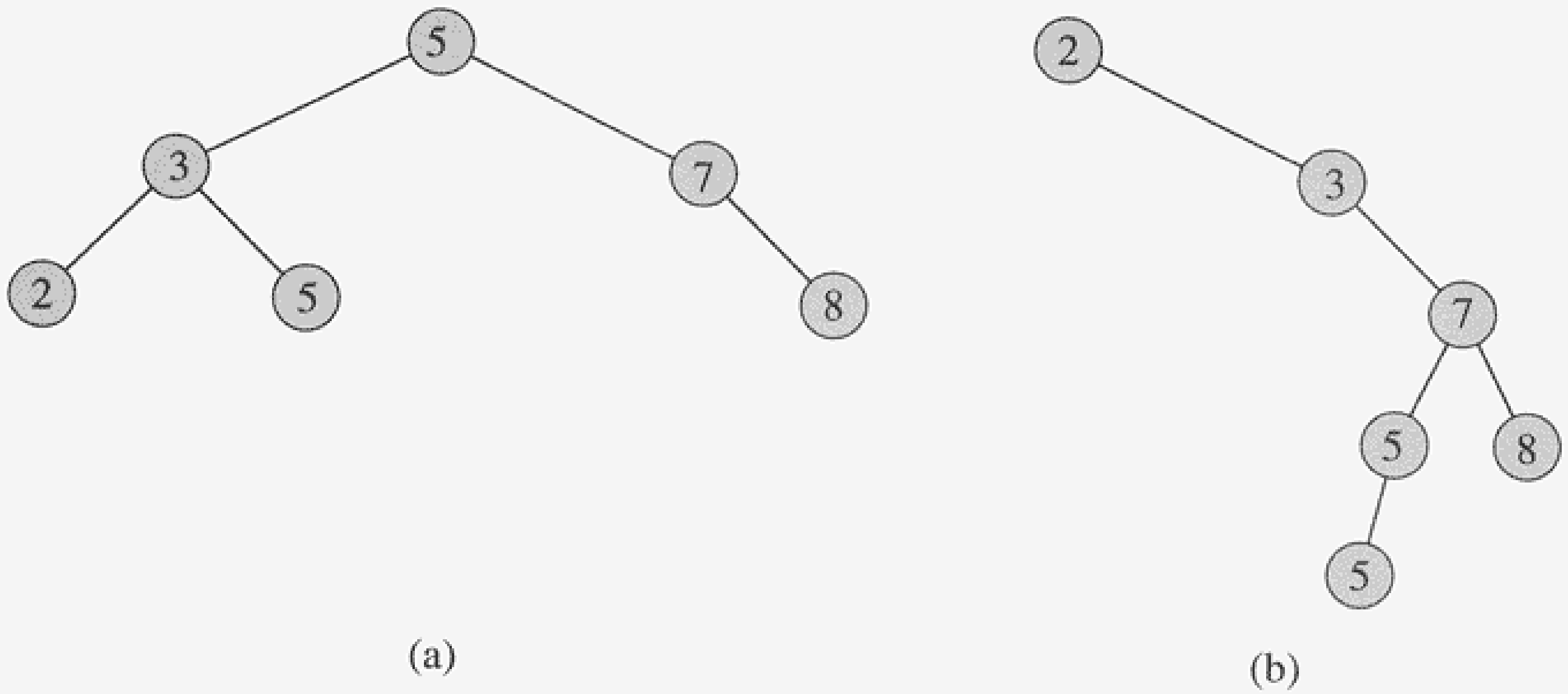
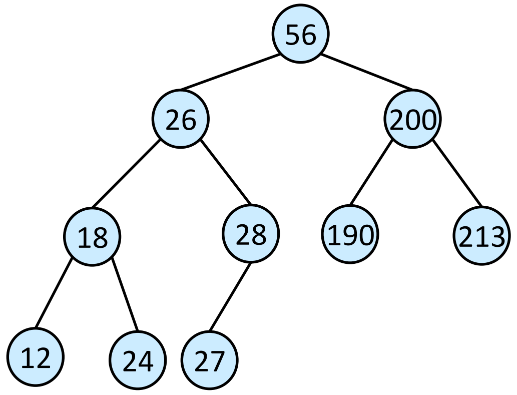


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional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

```
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])
```



Querying a Binary Search Tree

- All operations can be supported in $O(h)$ time.
- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of *n* nodes in the worst case.

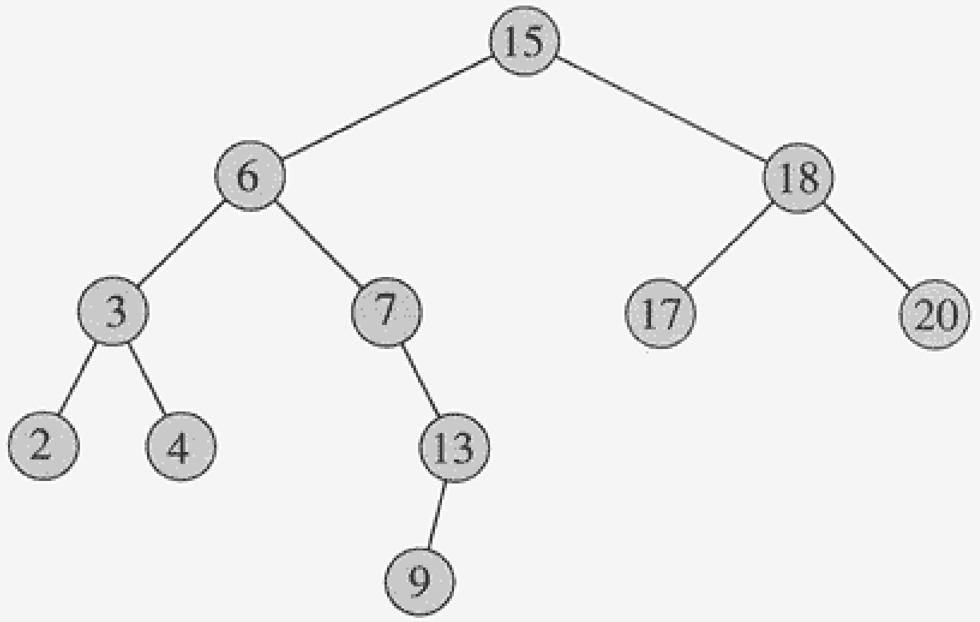
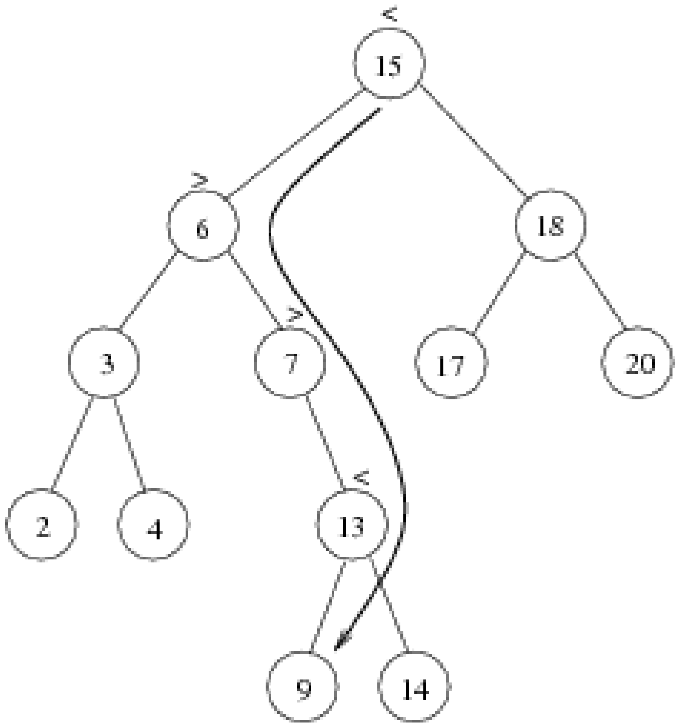


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path 15 → 6 → 7 → 13 from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. 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.

Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

```
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 $x$
```

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

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

Finding Min & Max

- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.
  - » The maximum is located at the right-most node.

Tree-Minimum(x)

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

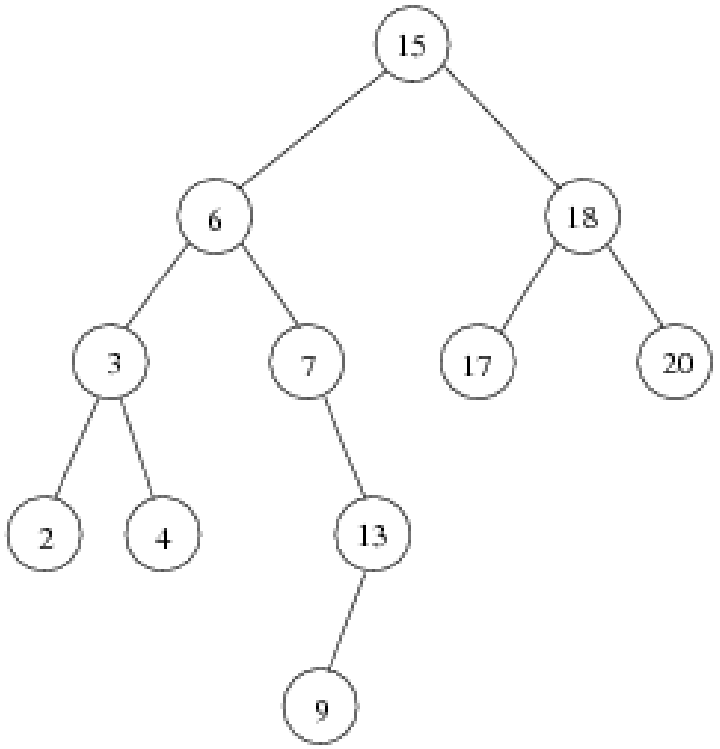
Tree-Maximum(x)

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

Q: How long do they take?

Inorder traversal of BST

- Print out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Predecessor and Successor

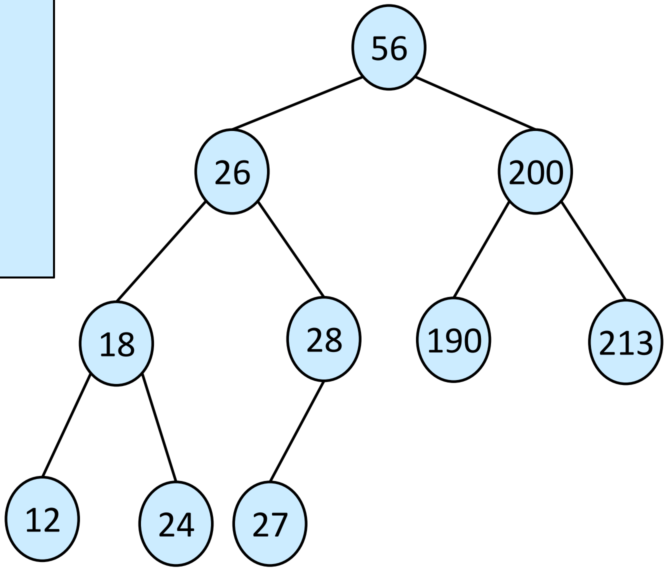
- Successor of node *x* is the node *y* such that *key*[*y*] is the smallest key greater than *key*[*x*].
- The successor of the largest key is NIL.
- Search consists of two cases.
  - If node *x* has a non-empty right subtree, then *x*'s successor is the minimum in the right subtree of *x*.
  - If node *x* has an empty right subtree, then:
    - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
    - *x*'s successor *y* is the node that *x* is the predecessor of (*x* is the maximum in *y*'s left subtree).
    - In other words, *x*'s successor *y*, is the lowest ancestor of *x* whose left child is also an ancestor of *x*.

Pseudo-code for Successor

```
Tree-Successor(x)
•       if right[x] ≠ NIL
2.        then return Tree-Minimum(right[x])
3.    y ← p[x]
4.    while y ≠ NIL and x = right[y]
5.    do x ← y
6.        y ← p[y]
7.    return y
```

Code for ***predecessor*** is symmetric.

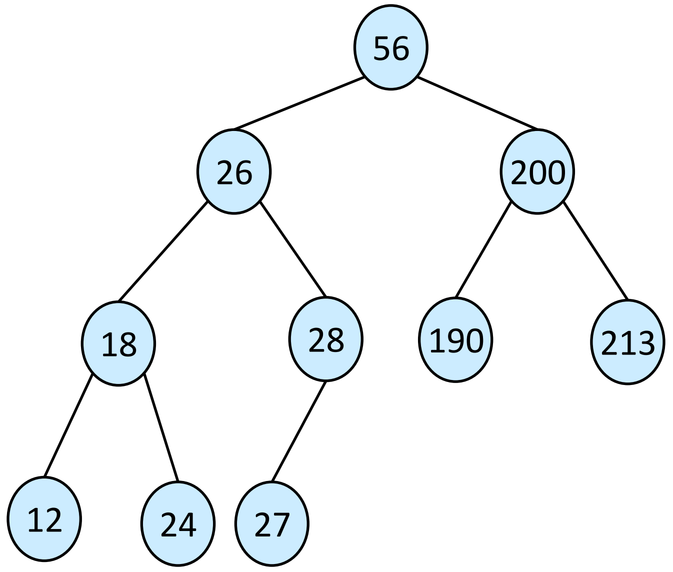
Running time: $O(h)$



BST Insertion

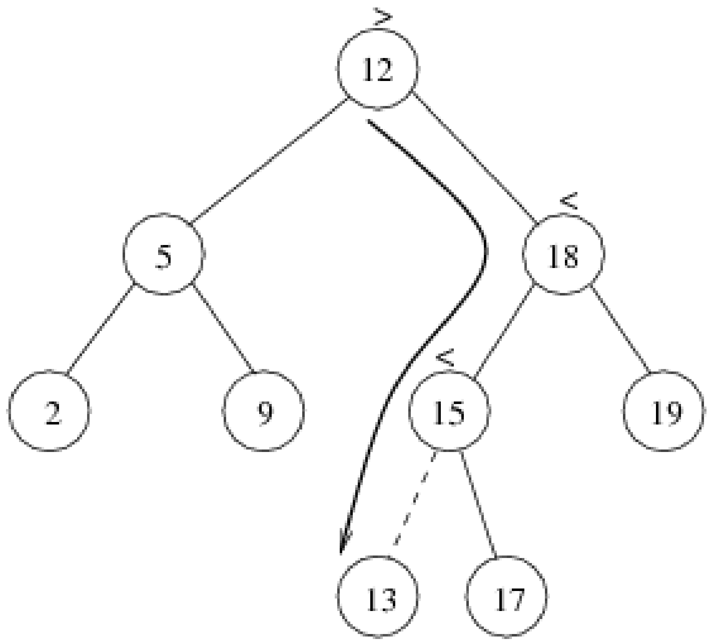
```
Tree-Insert(T, z)
1.   $y \leftarrow$ NIL
2.   $x \leftarrow root[T]$
3.   while $x \neq$ NIL
4.       do $y \leftarrow x$
5.           if $key[z] < key[x]$
6.               then $x \leftarrow left[x]$
7.               else $x \leftarrow right[x]$
8.   $p[z] \leftarrow y$
9.   if $y$ = NIL
10.      then $root[t] \leftarrow z$
11.      else if $key[z] < key[y]$
12.          then $left[y] \leftarrow z$
13.          else $right[y] \leftarrow z$
```

- Ensure the binary-search-tree property holds after change.
- Insertion is easier than deletion.



insert

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



- Time complexity = O(height of the tree)

Analysis of Insertion

- Initialization: $O(1)$
- While loop in lines 3-7 searches for place to insert z, maintaining parent y.
  This takes $O(h)$ time.
- Lines 8-13 insert the value: $O(1)$

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

```
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
```

Tree-Delete (T, x)

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

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

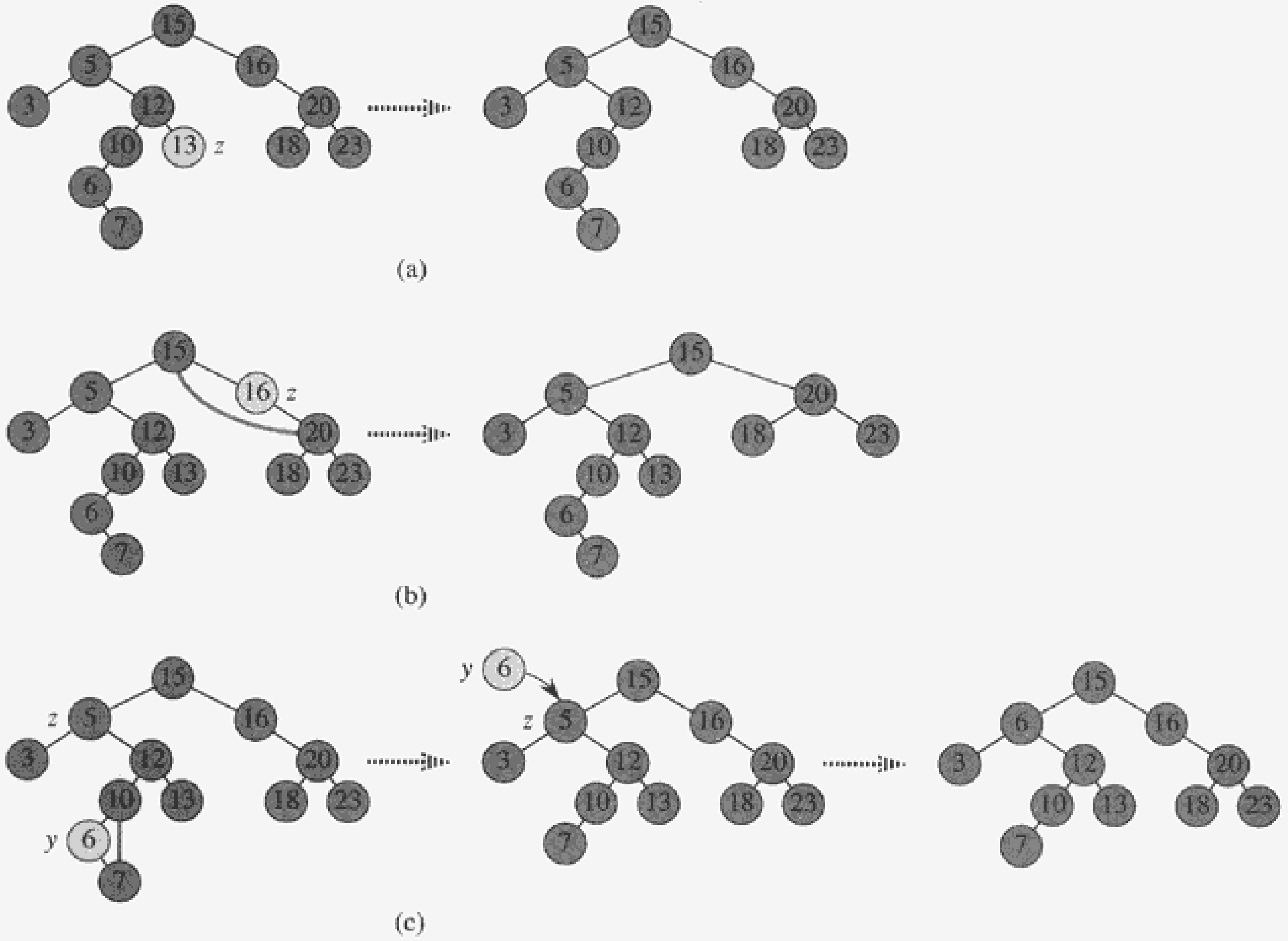


Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out 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.

```
TREE-DELETE($T$, $z$)
if $left[z] = \text{NIL}$ or $right[z] = \text{NIL}$
    then $y \leftarrow z$
    else $y \leftarrow$ TREE-SUCCESSOR($z$)
if $left[y] \neq \text{NIL}$
    then $x \leftarrow left[y]$
    else $x \leftarrow right[y]$
if $x \neq \text{NIL}$
    then $p[x] \leftarrow p[y]$
if $p[y] = \text{NIL}$
    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 $y$
```

Quiz 1

A binary search tree (BST) is built by inserting tree following values in the given order: 4,25,15,12,20,70,40.

The Post Order Traversal will be

A. 12, 15, 20, 40,70,25, 4

B. 12,20, 15,40, 70,25, 4

C. 4,25, 70, 40,15, 12,20

D. 4,12, 15, 20, 25,40,70

Quiz 2

Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree?

Quiz 3

The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?

A	2
B	3
C	4
D	6

Quiz 4

How many distinct BSTs can be constructed with 3 distinct keys?

How many distinct BSTs can be created out of 4 distinct keys?