Where We Are

Studying the absolutely essential ADTs of computer science and classic data structures for implementing them

ADTs so far:

- List — add, remove, isEmpty, ….
- Stack: — push, pop, isEmpty, …
- Queue: — enqueue, dequeue, isEmpty, …

Next:

- Binary Search Tree
- Priority queue: — insert, deleteMin, …
- Dictionary/Map: — key-value pairs
- Set: — just keys

Objectives

To become familiar with binary search trees.

To represent binary trees using linked data structures.

To search an element in binary search tree.

To insert an element into a binary search tree.

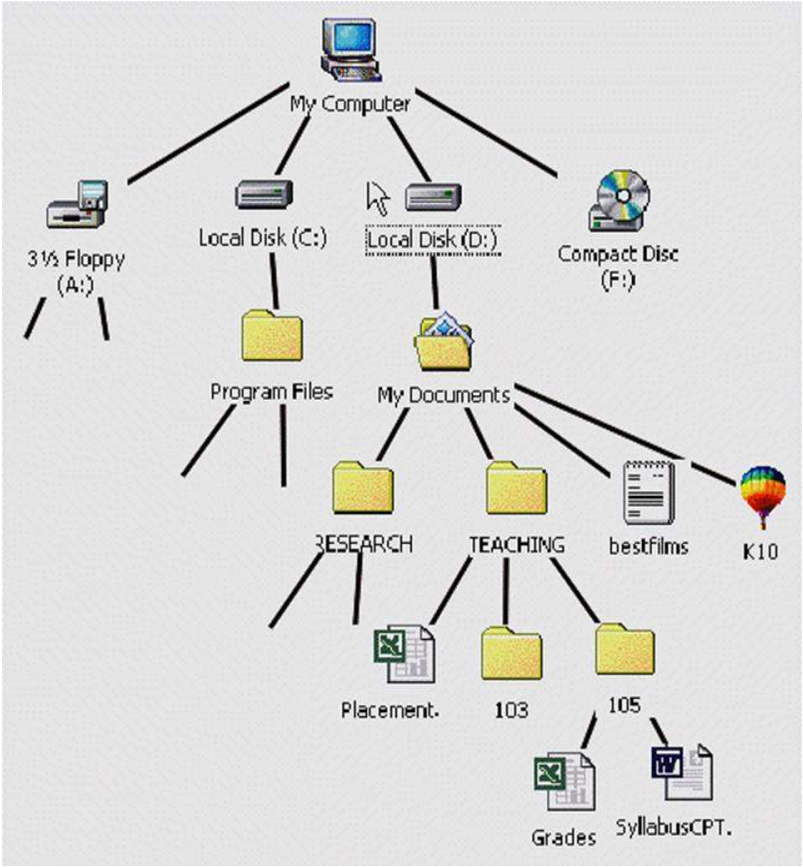
To traverse elements in a binary tree.

To design and implement the Tree interface and the BST class.

To delete elements from a binary search tree.

What is a Tree

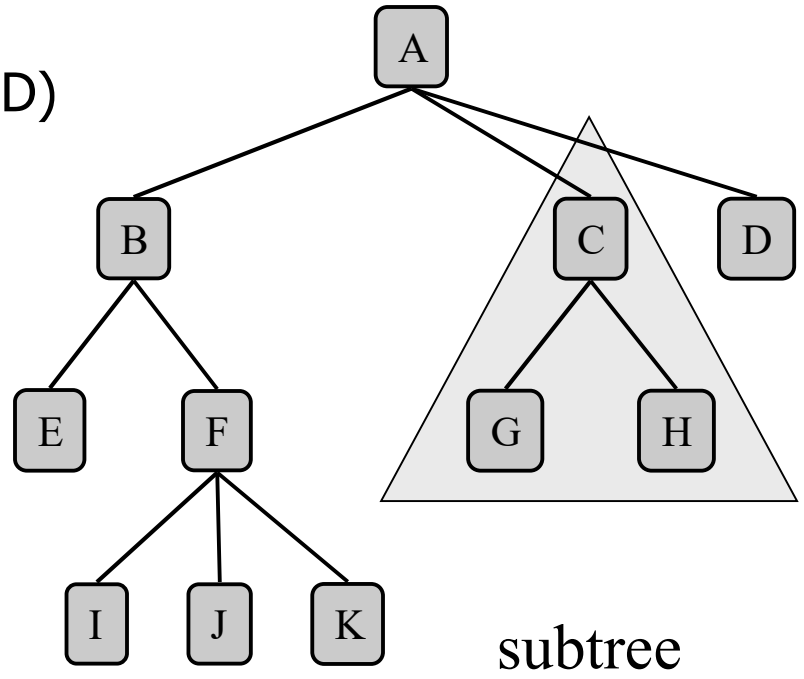
- A list, stack, or queue is a linear structure that consists of a sequence of elements.
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
- Organization charts
- File systems
- Programming environments



Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.

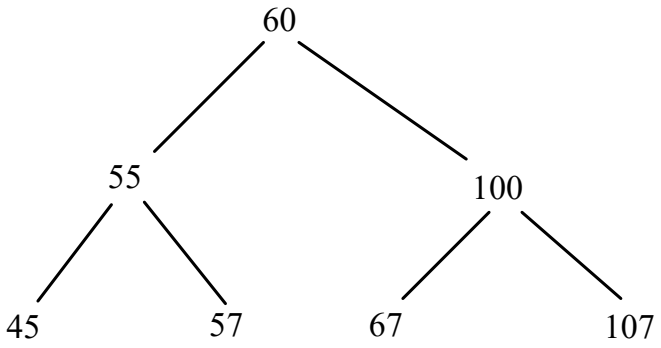
- Subtree: tree consisting of a node and its descendants



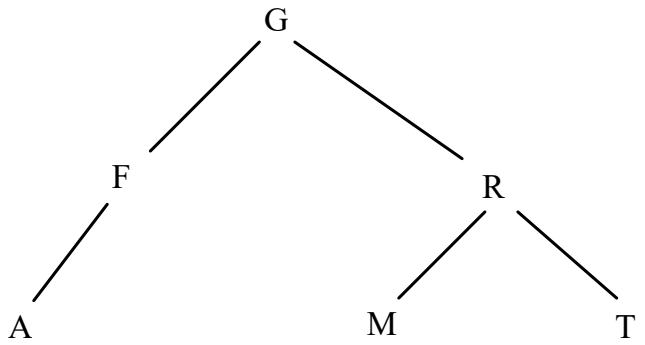
Why Trees?

Trees offer speed ups because of their branching factors

- Binary Search Trees are structured forms of *binary search*



(A)



(B)

Why Trees?

Trees offer speed ups because of their branching factors

- Binary Search Trees are structured forms of *binary search*

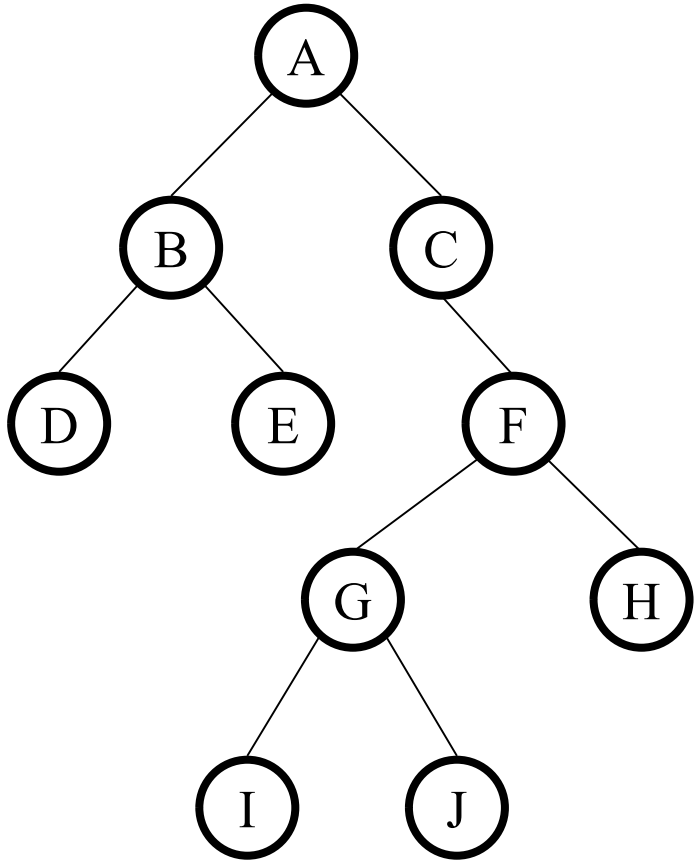
	Insert	Find	Delete
Worse-Case	O(n)	O(n)	O(n)
Average-Case	O(log n)	O(log n)	O(log n)

Binary Search Trees: A Review

Binary Trees

A non-empty binary tree consists of a

- a root (with data)
- a left subtree (may be empty)
- a right subtree (may be empty)

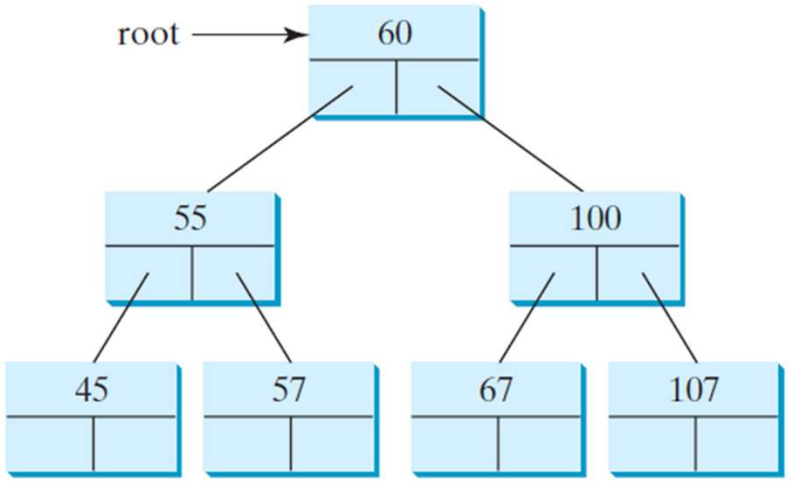


Representation:

Data	
left pointer	right pointer

Representing Binary Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named **left** and **right** that reference the left child and right child, respectively, as shown in Figure 25.2.



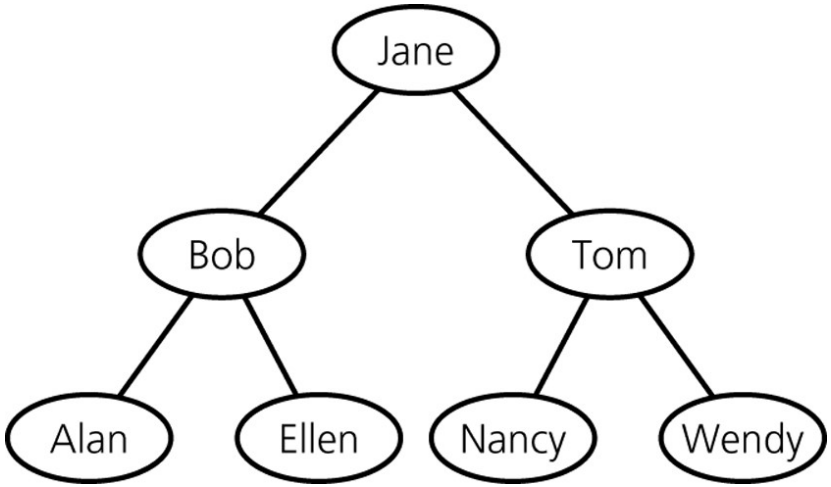

```
class TreeNode<E> {
 E element;
 TreeNode<E> left;
 TreeNode<E> right;
 public TreeNode(E o) {
   element = o;
 }
}
```

Terminology

BSTs are binary trees with the following added criteria:

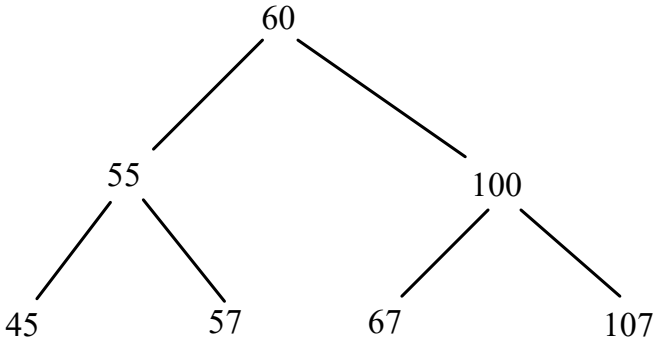
- Each node has a key for comparing nodes
- Keys in left subtree are smaller than node's key
- Keys in right subtree are larger than node's key
- Both T_L and T_R are binary search trees

A binary search tree of names

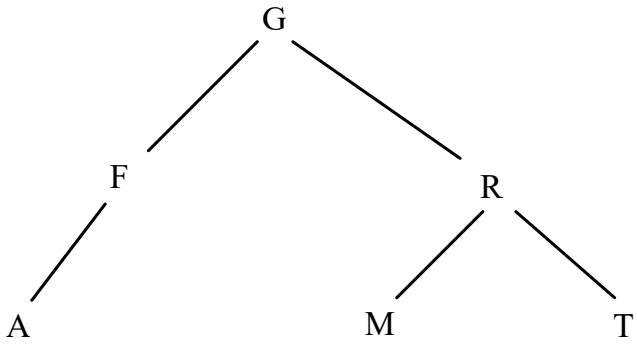


Binary Search Trees

- Why Use Binary Search Tree
  - It combines the advantages of two other structures.
    - Ordered array
    - Linked list



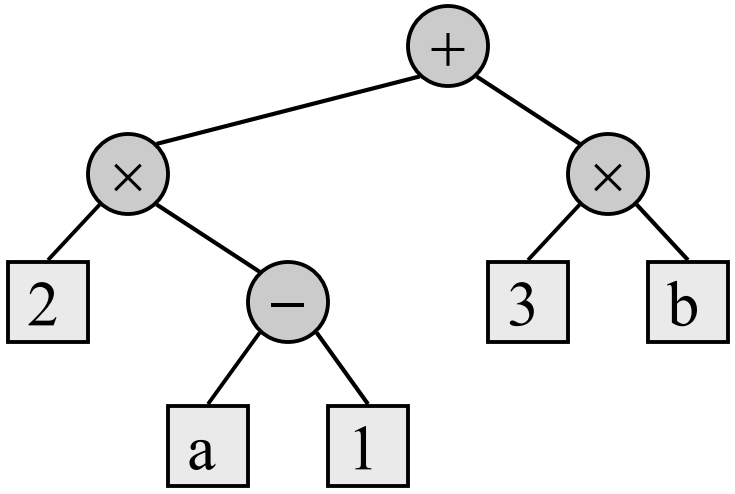
(A)



(B)

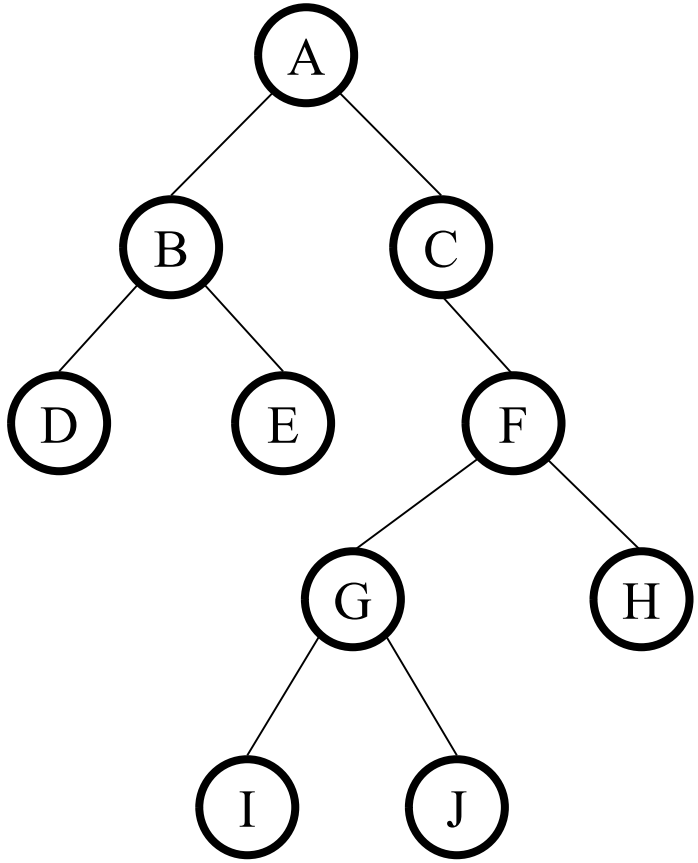
Arithmetic Expression Tree

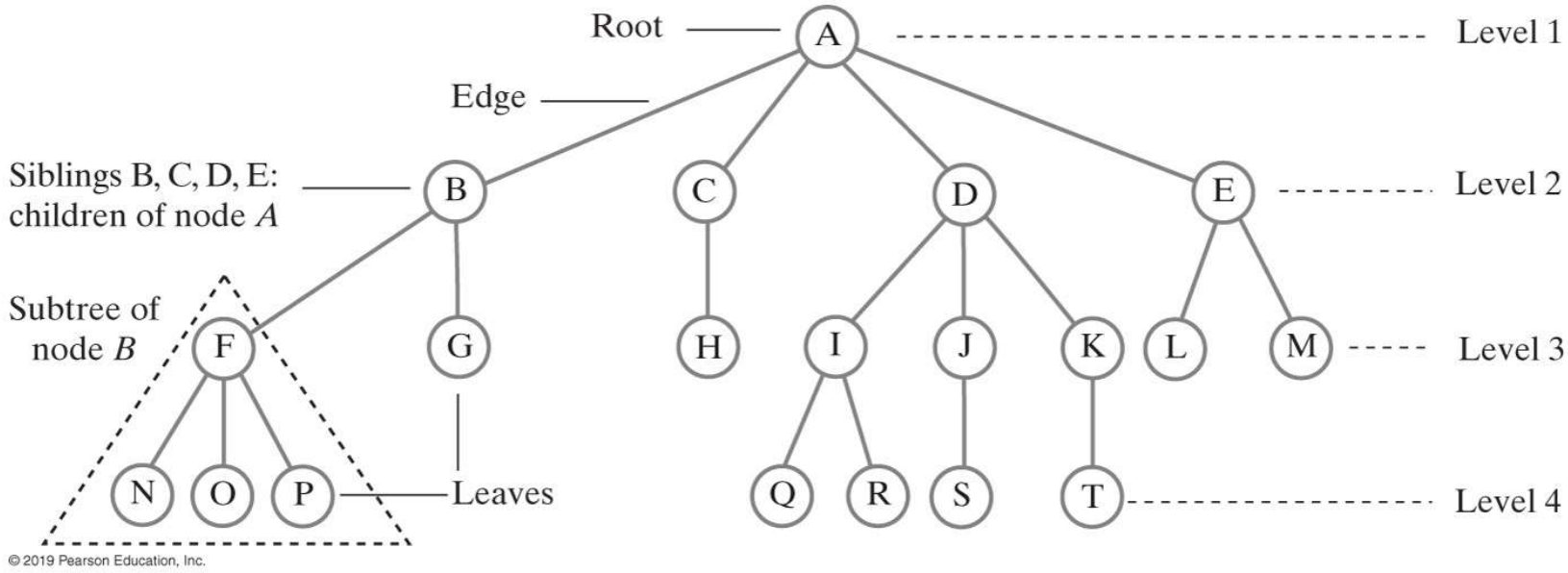
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Terminology

- The length of a path
  - Is the number of the nodes in the path.
- The depth of a node
  - Is the length of the path from the root to the node.
- Level of a node n in a tree T
    - Is the depth of n

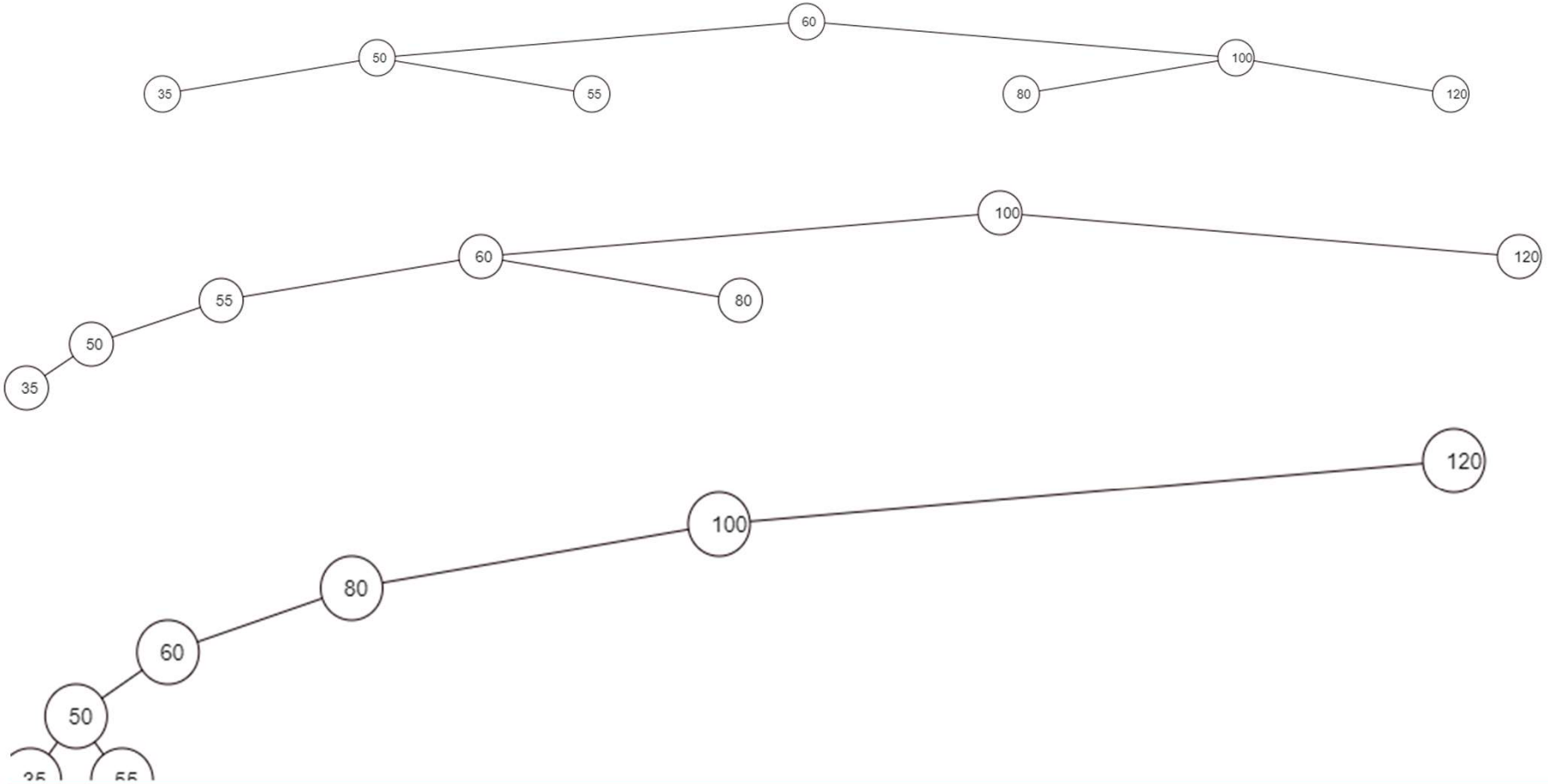



Root
A
Level 1
Edge
Siblings B, C, D, E:
children of node A
B
C
D
E
Level 2
Subtree of
node B
F
G
H
I
J
K
L
M
Level 3
N
O
P
Leaves
Q
R
S
T
Level 4
© 2019 Pearson Education, Inc.

Terminology

- The height of trees
  - Level of a node n in a tree T
    - If n is the root of T, it is at level 1
    - If n is not the root of T, its level is 1 greater than the level of its parent
  - Height of a tree T defined in terms of the levels of its nodes
    - If T is empty, its height is 0
    - If T is not empty, its height is equal to the maximum level of its nodes

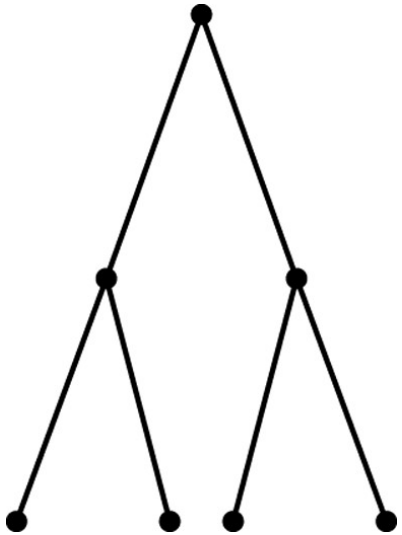
Terminology



Binary trees with the same nodes but different heights

Terminology

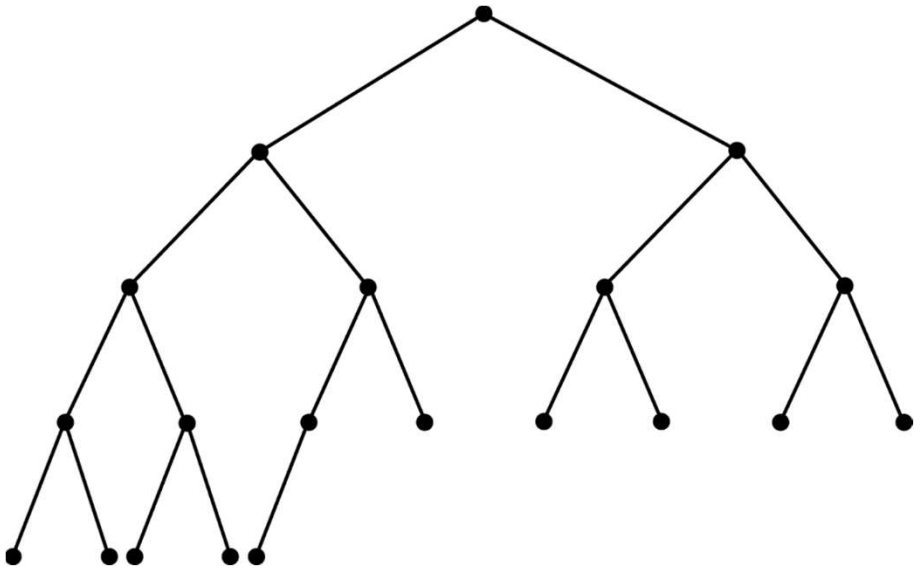
- Full, complete, and balanced binary trees
  - Recursive definition of a full binary tree
    - If T is empty, T is a full binary tree of height -1
    - If T is not empty and has height h > -1, T is a full binary tree if its root's subtrees are both full binary trees of height h – 1



A full binary tree of height 3

Terminology

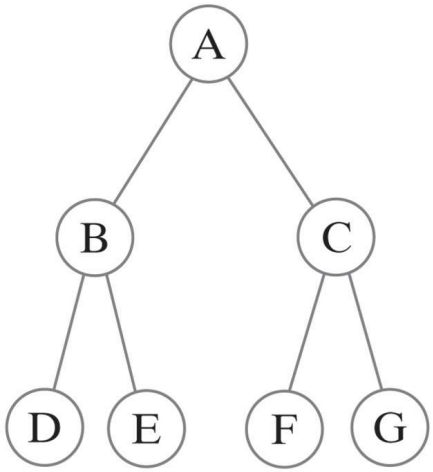
- Complete binary trees
  - A binary tree T of height h is complete if
    - All nodes at level h – 2 and above have two children each, and
    - When a node at level h – 1 has children, all nodes to its left at the same level have two children each, and
    - When a node at level h – 1 has one child, it is a left child



A complete binary tree

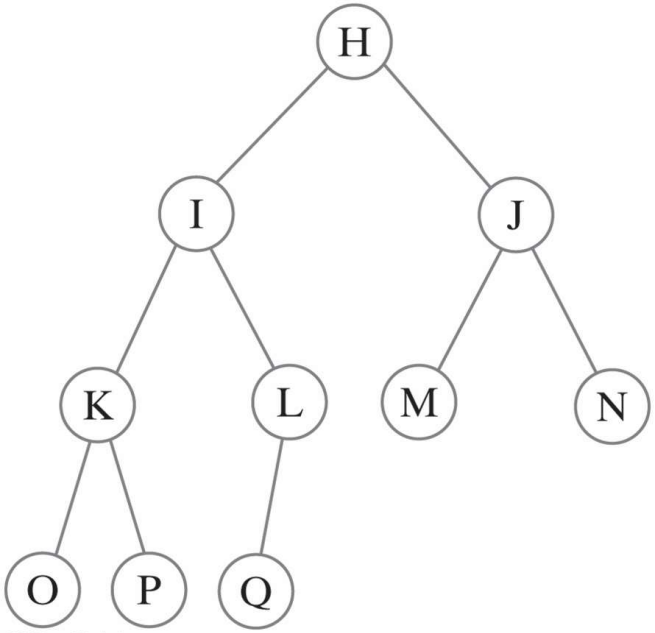
Binary Trees

(a) Full tree

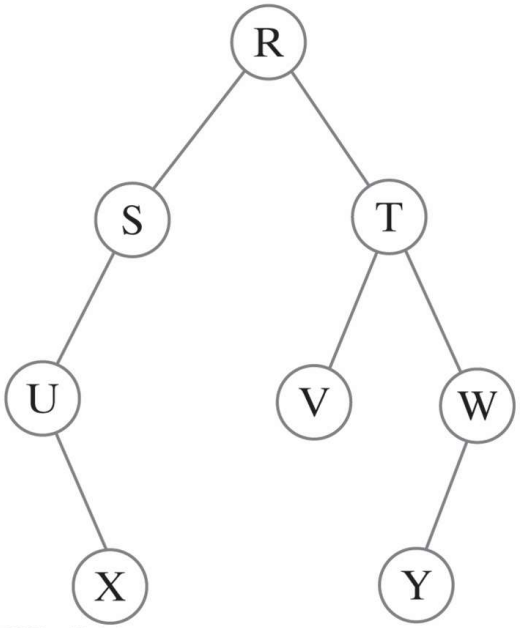


Left children: B, D, F
Right children: C, E, G

(b) Complete tree



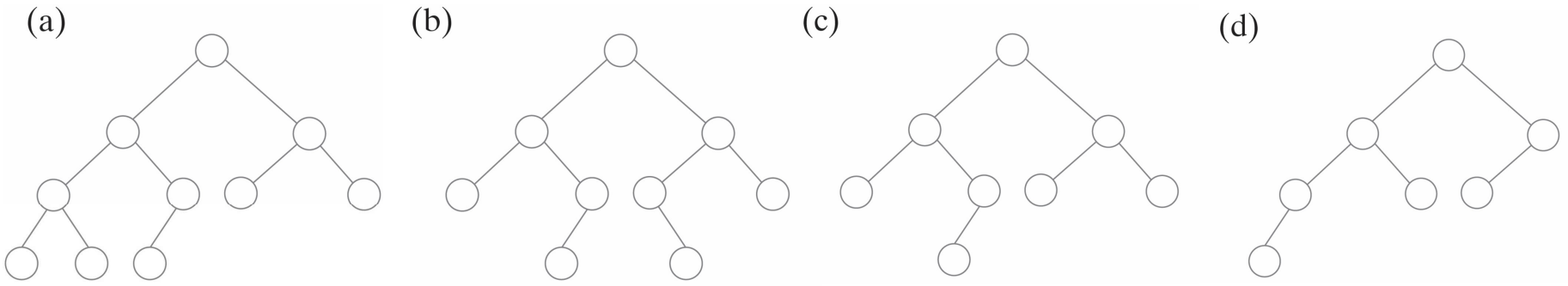
(c) Tree that is not full and not complete



Terminology

- Balanced binary trees
  - A binary tree is balanced if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1
- Full binary trees are complete
- Complete binary trees are balanced

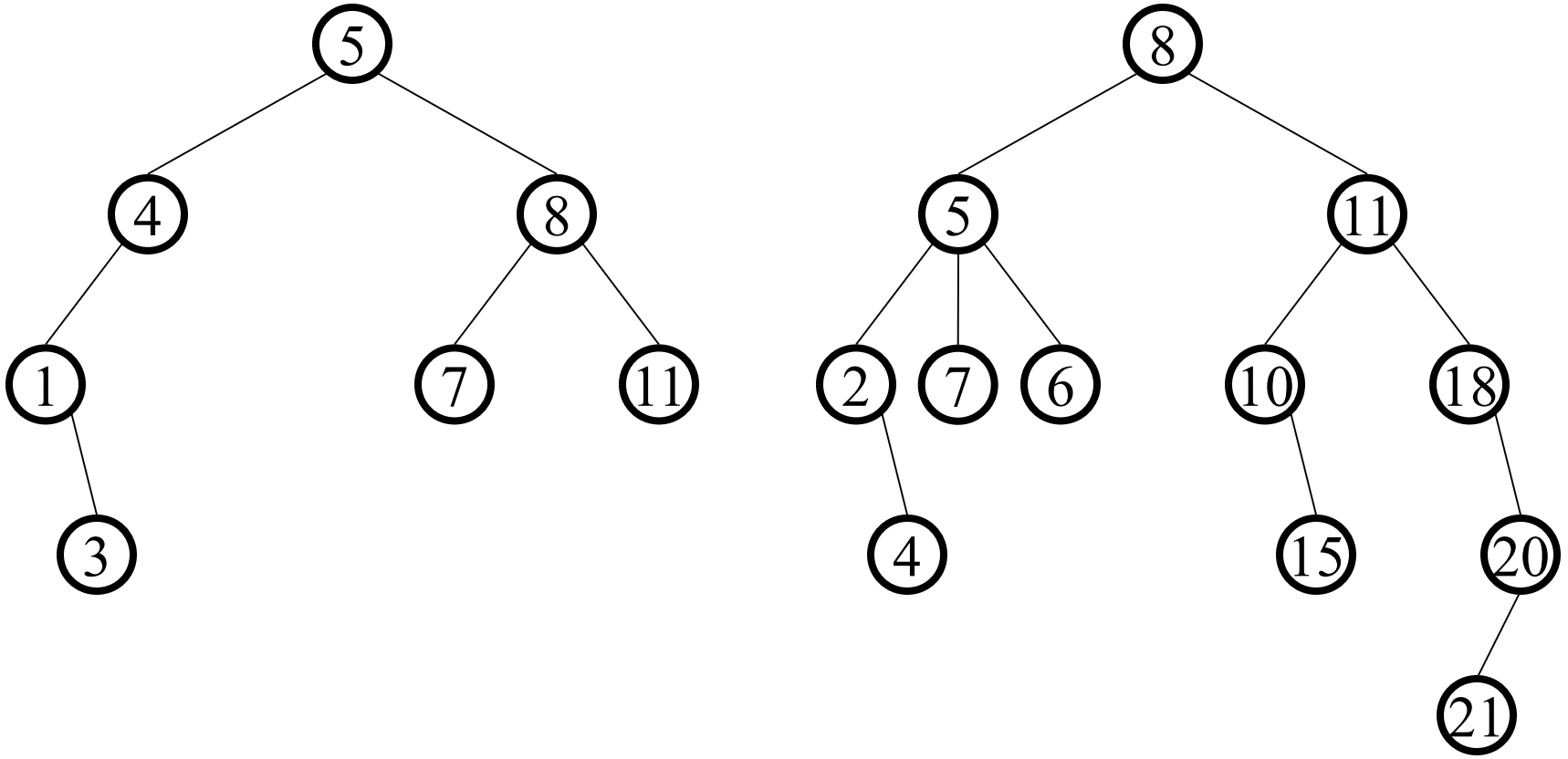
Binary Trees



Balanced and complete

Balanced, but not complete

Are these BSTs?



Tree Traversal (1 of 5)

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents DFS & BFS:

- Depth-first search (DFS)
  - The algorithm begins at the root node and then it explores each branch before backtracking. It is implemented using stacks.
  - Recursion is used for backtracking.
- For Binary trees, there are three types of DFS traversals.

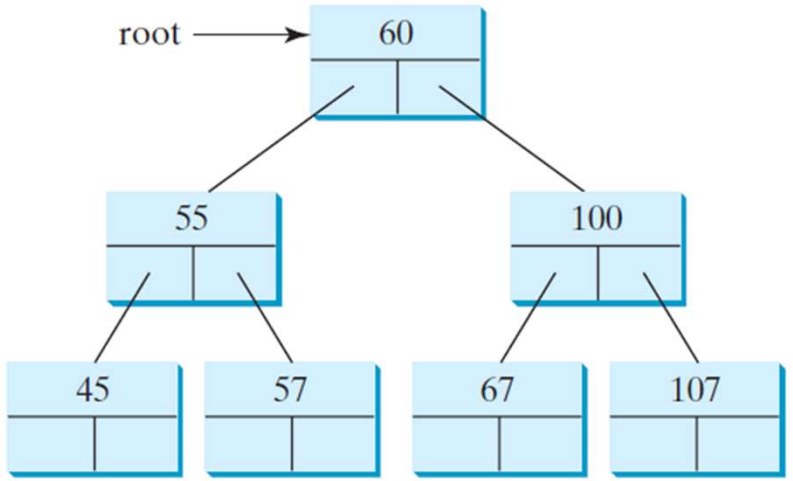
1. **In-Order**
2. **Pre-Order**
3. **Post-Order**

Tree Traversal (2 of 5)

The inorder traversal is to visit the left subtree of the current node first recursively, then the current node itself,

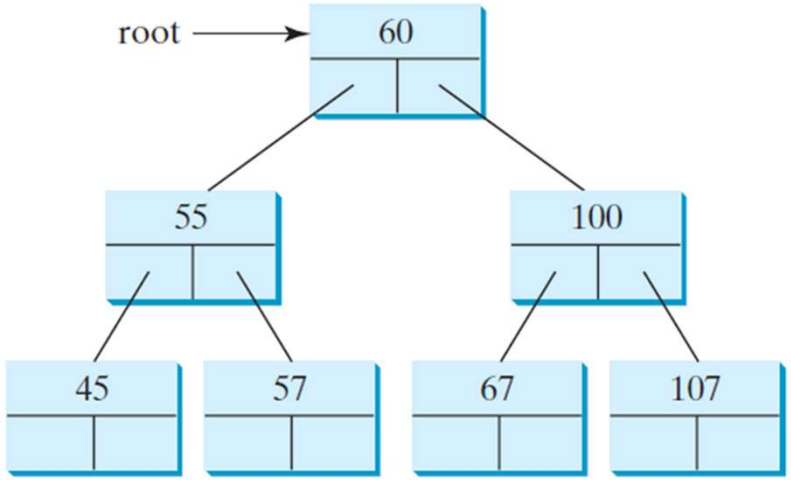
and finally the right subtree of the

current node recursively.



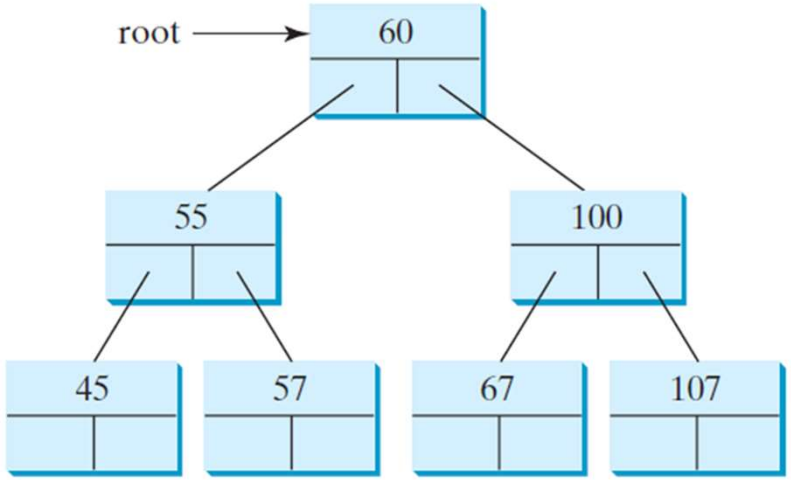
Tree Traversal (3 of 5)

The preorder traversal is to visit the current node first, then the left subtree of the current node recursively, and finally the right subtree of the current node recursively.



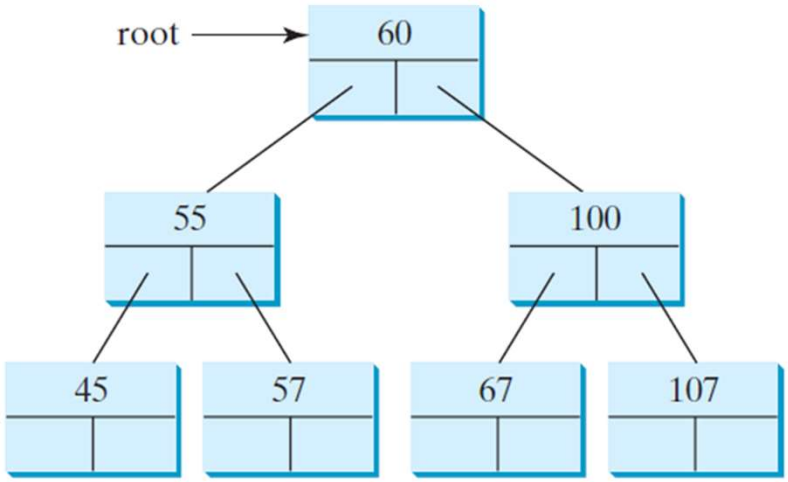
Tree Traversal (4 of 5)

The postorder traversal is to visit the left subtree of the current node first recursively, then the right subtree of the current node recursively, and finally the current node itself.

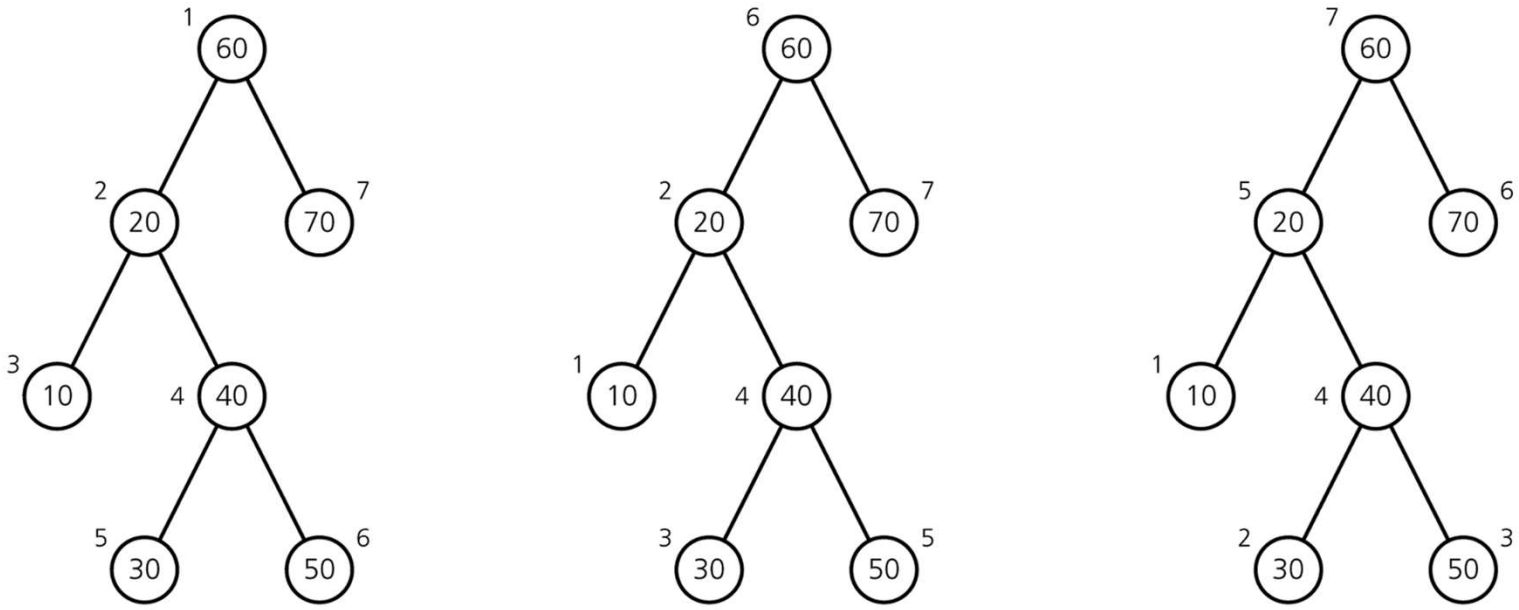


Tree Traversal (5 of 5)

- **Breadth-First Search (BFS)**
  - The breadth-first traversal is to visit the nodes level by level. First, visit the root, then all children of the root from left to right, then grandchildren of the root from left to right, and so on.



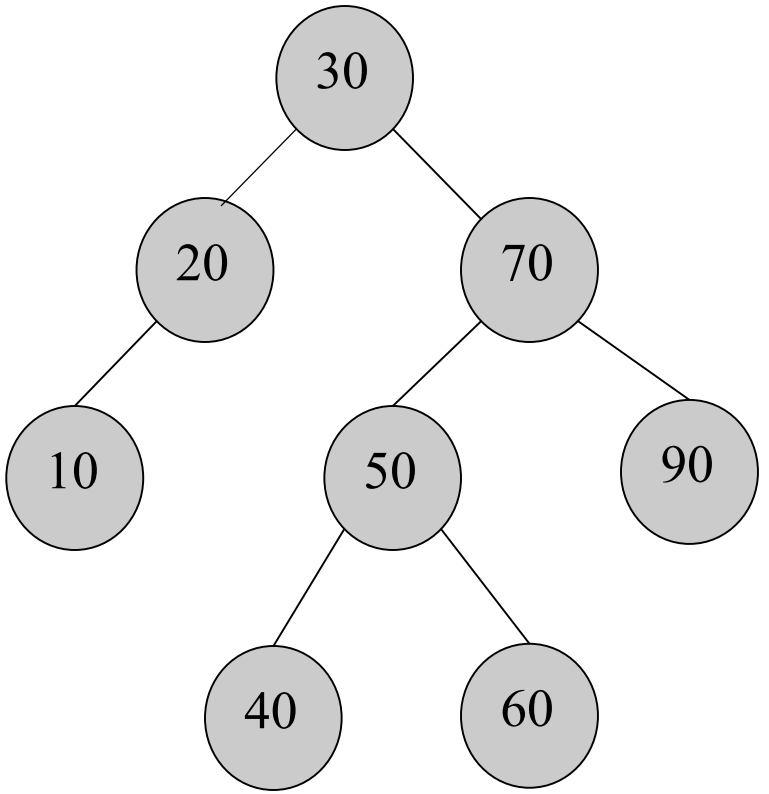
Traversal of a Binary Tree



Traversals of a binary tree: preorder; inorder; postorder

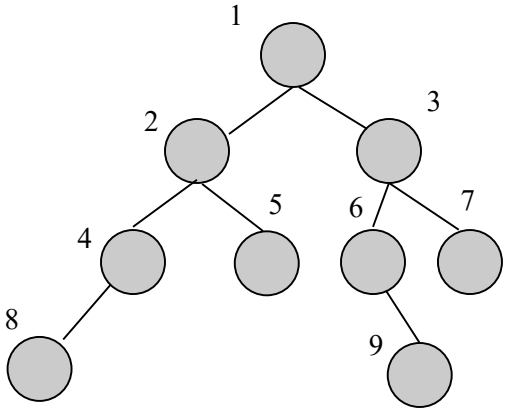
Check Point

- Traverse
  - Inorder
  - Preorder
  - Postorder



Check Point

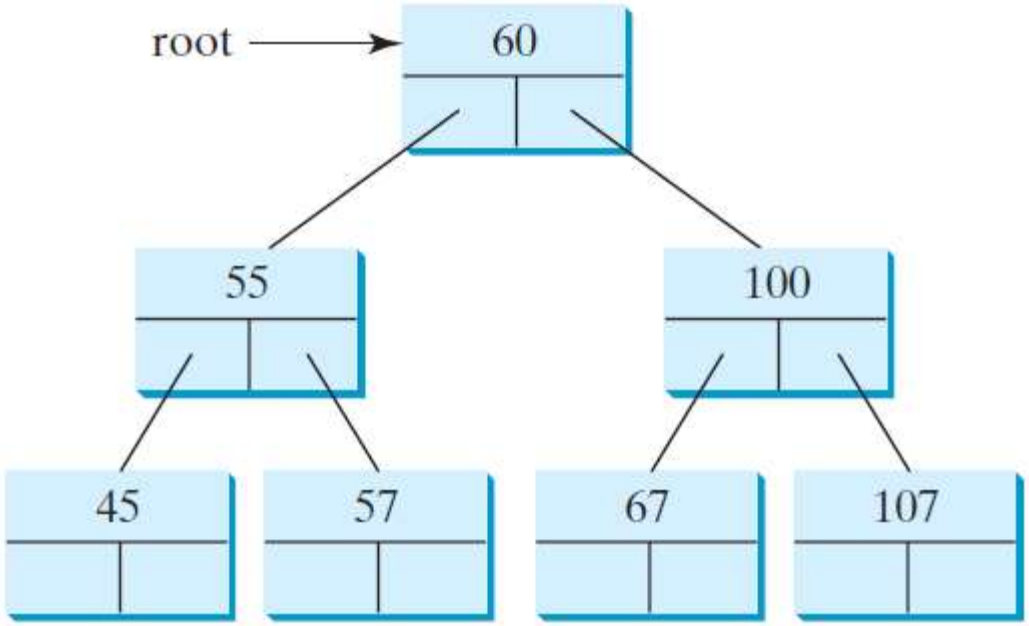
- Consider the following BST. The numbers simply label the nodes so that you can reference them; they do not indicate the contents of the nodes.



1. Which node must contain the inorder successor of the value in the root
2. In what order will an inorder traversal visit the nodes of this tree?

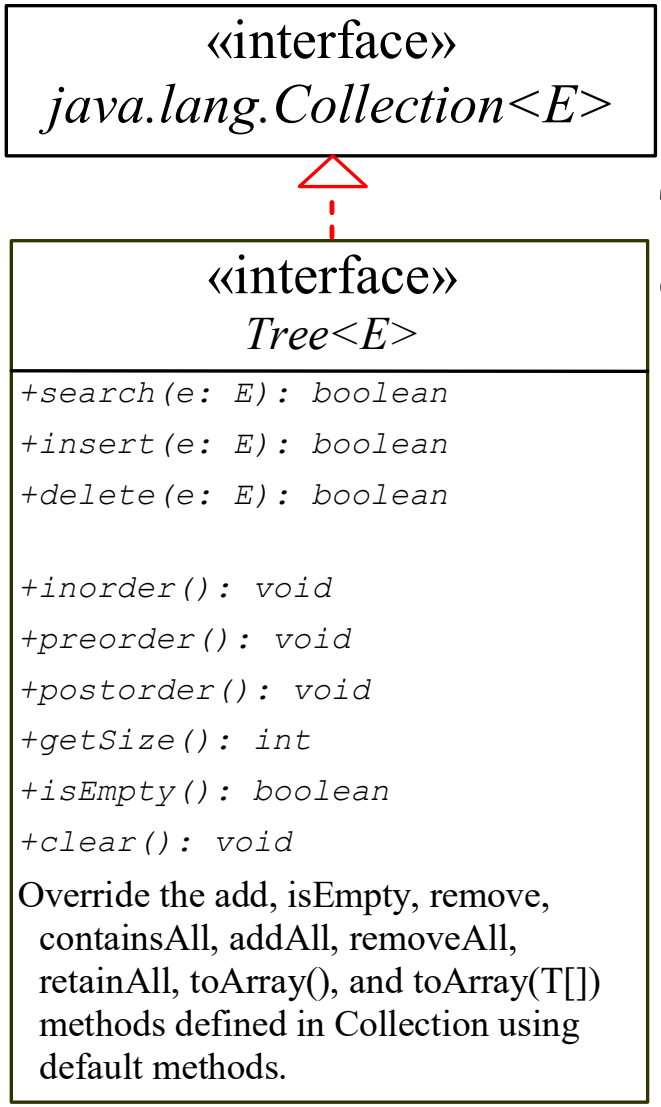
Representing Binary Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively.




```
class TreeNode<E> {
  E element;
  TreeNode<E> left;
  TreeNode<E> right;

  public TreeNode(E o) {
    element = o;
  }
}
```

The Tree Interface



The Tree interface defines common operations for trees.

Returns true if the specified element is in the tree.

Returns true if the element is added successfully.

Returns true if the element is removed from the tree successfully.

Prints the nodes in inorder traversal.

Prints the nodes in preorder traversal.

Prints the nodes in postorder traversal.

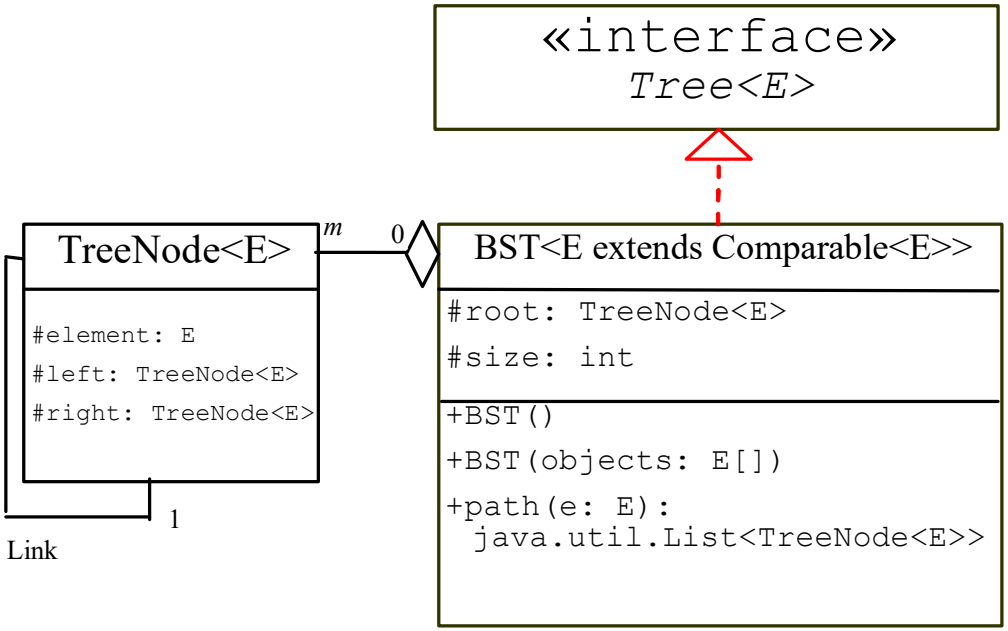
Returns the number of elements in the tree.

Returns true if the tree is empty.

Removes all elements from the tree.

The BST Class

Let's define the binary tree class, named BST with A concrete BST class can be defined to extend AbstractTree.



The root of the tree.

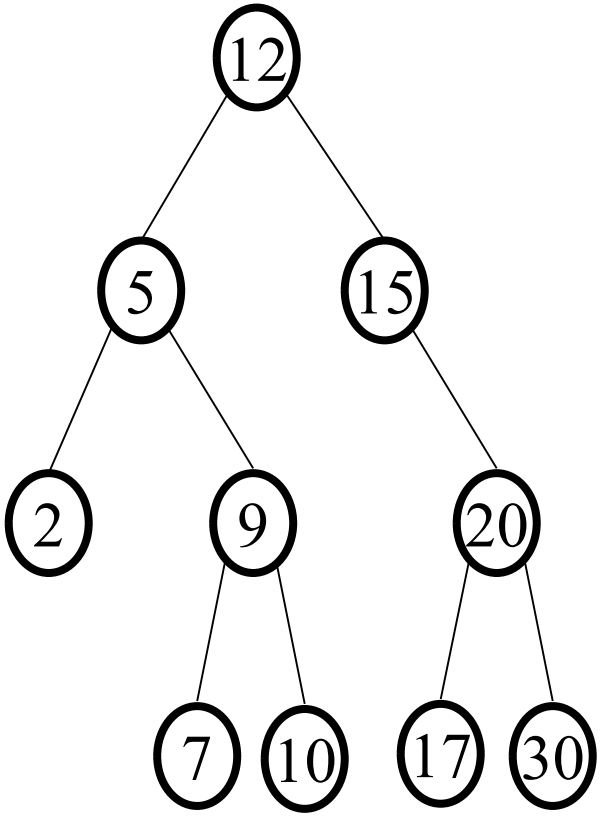
The number of nodes in the tree.

Creates a default BST.

Creates a BST from an array of elements.

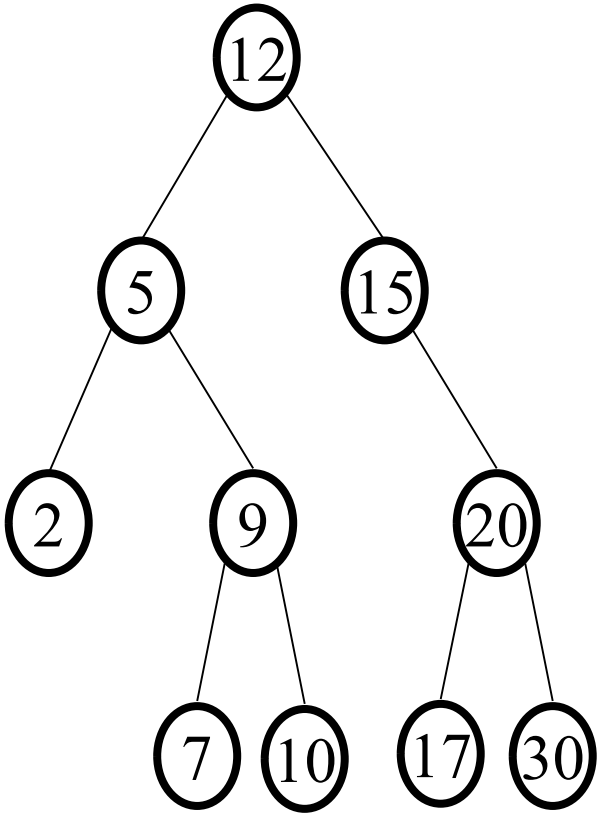
Returns the path of nodes from the root leading to the node for the specified element. The element may not be in the tree.

Search in BST, Iterative

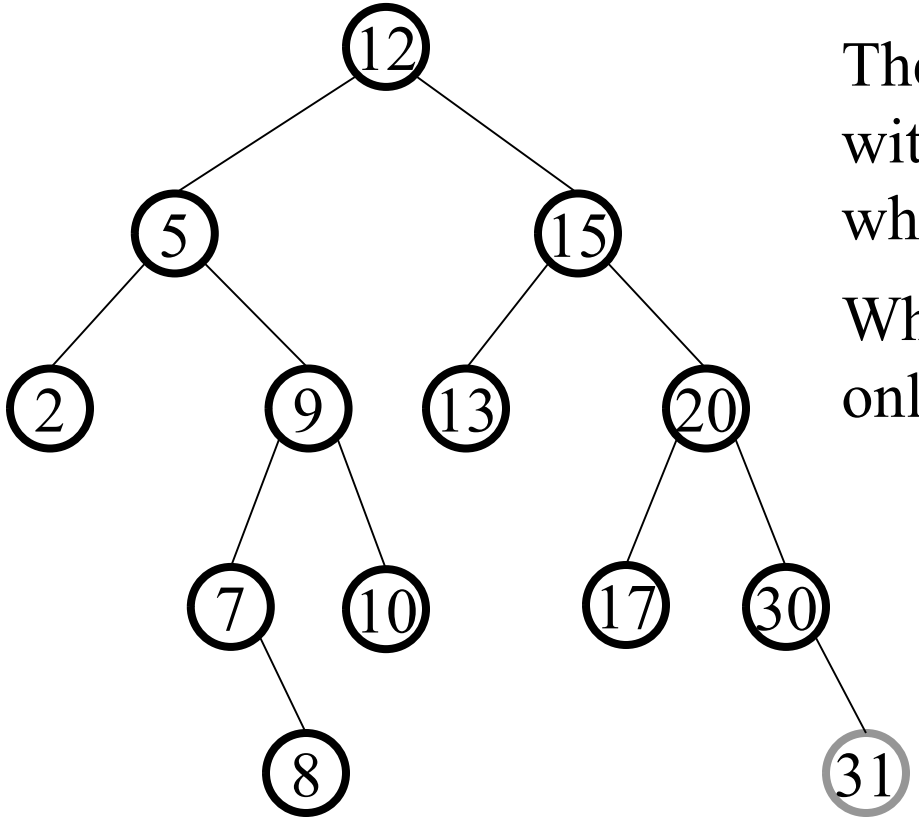



```
public boolean search(E element) {
  TreeNode<E> current = root; // Start
from the root
  while (current != null)
    if    (element < current.element)
{
      current = current.left; // Go
left
    }
    else if (element >
current.element) {
      current = current.right; // Go
right
    }
    else // Element matches
current.element
      return true; // Element is found
  return false; // Element is not in
the tree
}
```

Search in BST, Recursive



Insert in BST



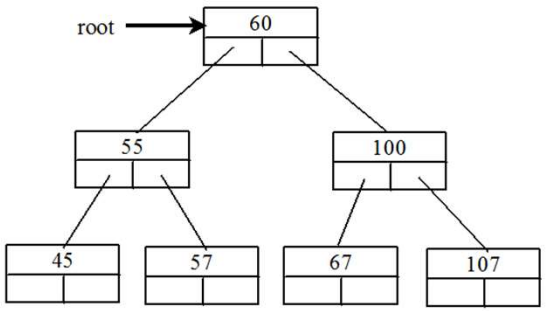
The code for insert is the same as with search except you add a node when you fail to find it.

What makes it easy is that inserts only happen at the leaves.

Inserting an Element to a Binary Tree

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

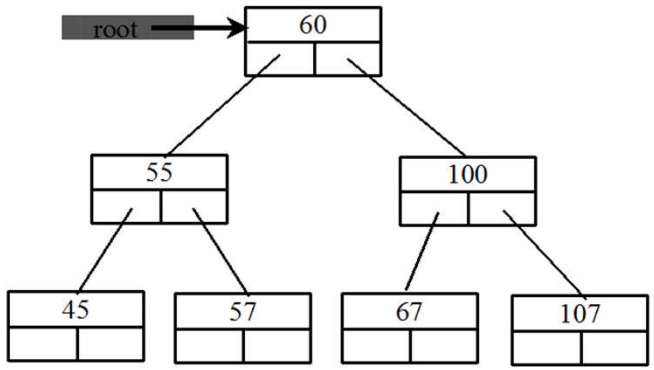


Trace Inserting 101 into the following tree

(1 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

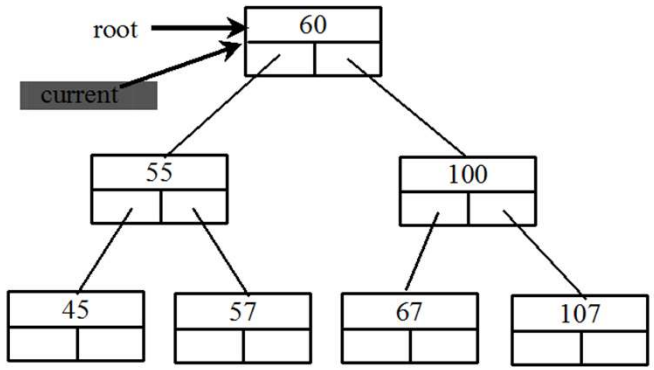
Insert 101 into the following tree.



Trace Inserting 101 into the following tree
(2 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

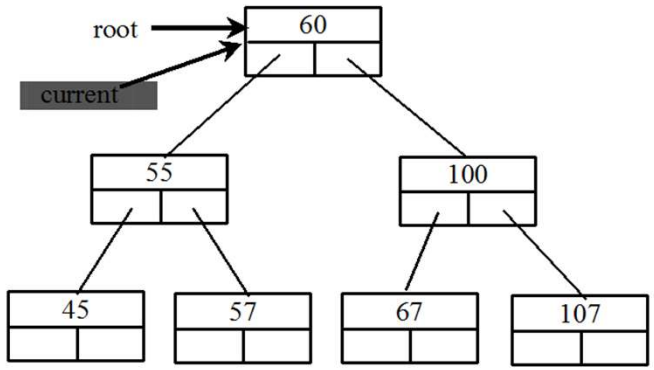
Insert 101 into the following tree.



Trace Inserting 101 into the following tree
(3 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.



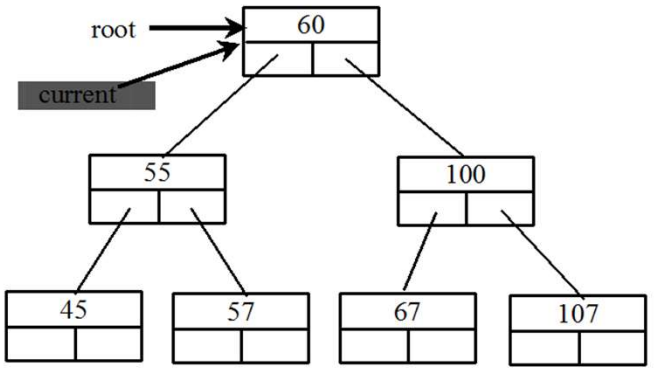
Trace Inserting 101 into the following tree

(4 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 60?

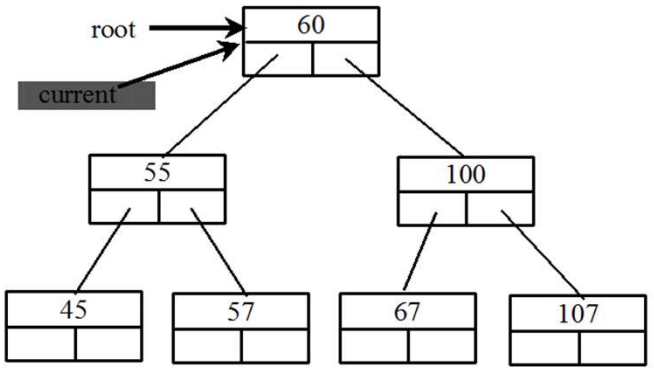


Trace Inserting 101 into the following tree
(5 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60?

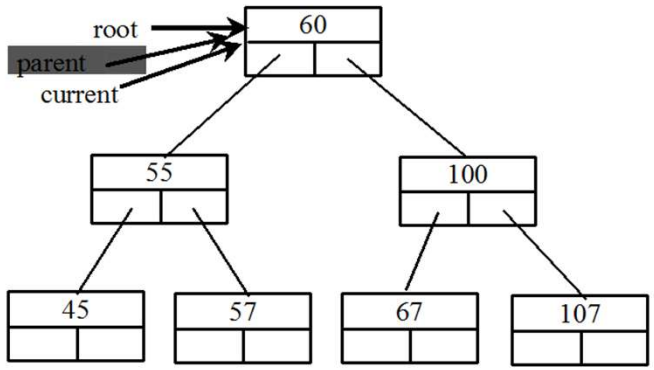


Trace Inserting 101 into the following tree
(6 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60 true

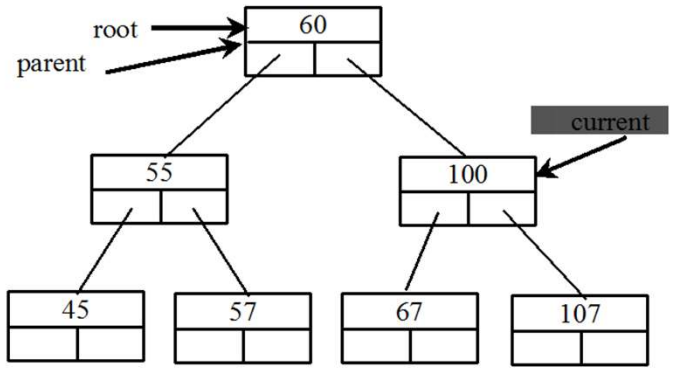


Trace Inserting 101 into the following tree
(7 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60 true

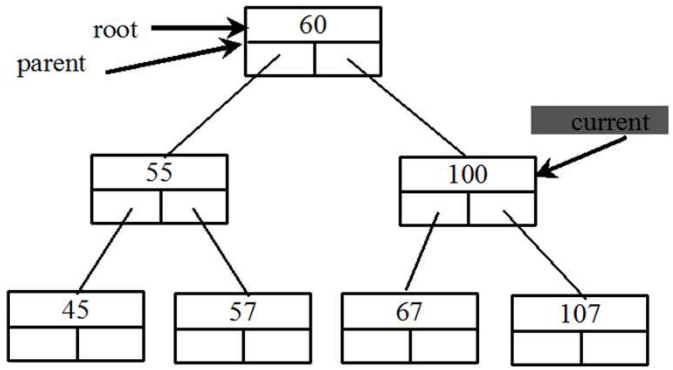


Trace Inserting 101 into the following tree
(8 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60 true

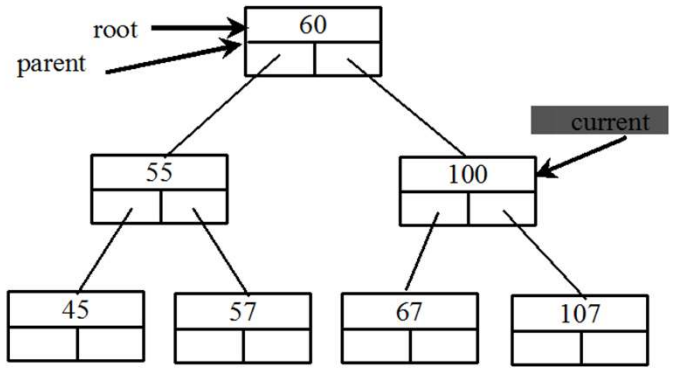


Trace Inserting 101 into the following tree
(9 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 false

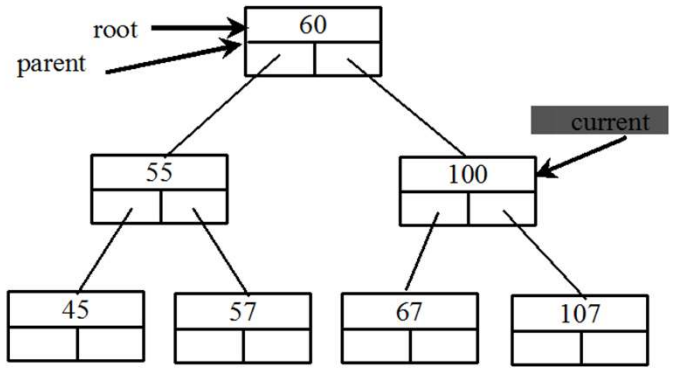


Trace Inserting 101 into the following tree
(10 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

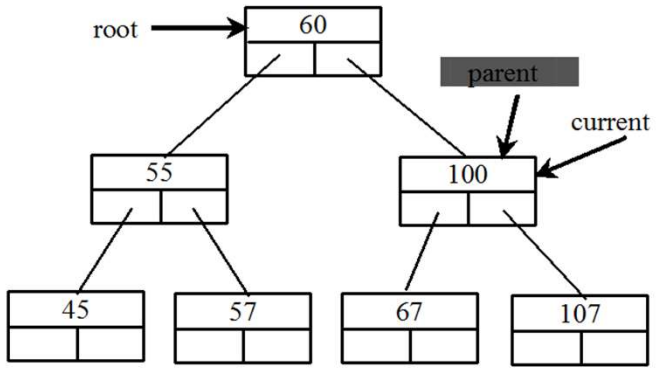


Trace Inserting 101 into the following tree
(11 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

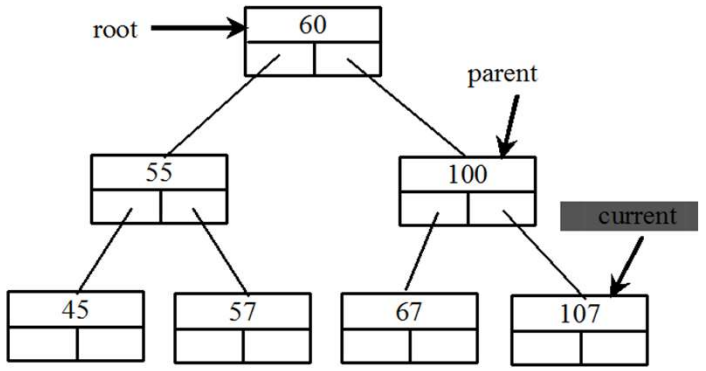


Trace Inserting 101 into the following tree
(12 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

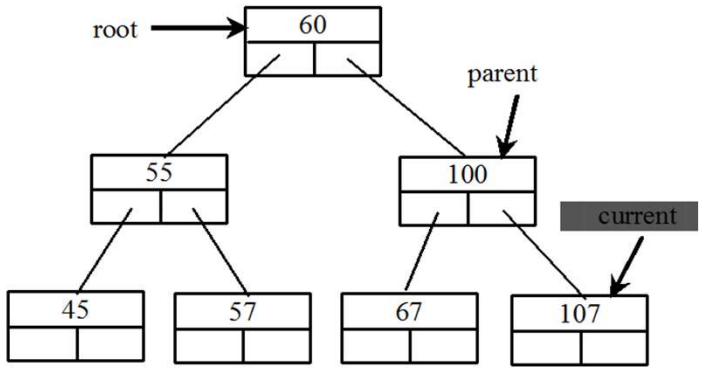


Trace Inserting 101 into the following tree
(13 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

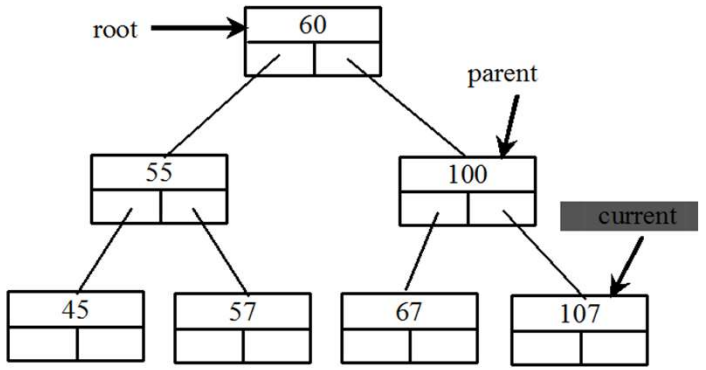


Trace Inserting 101 into the following tree
(14 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true

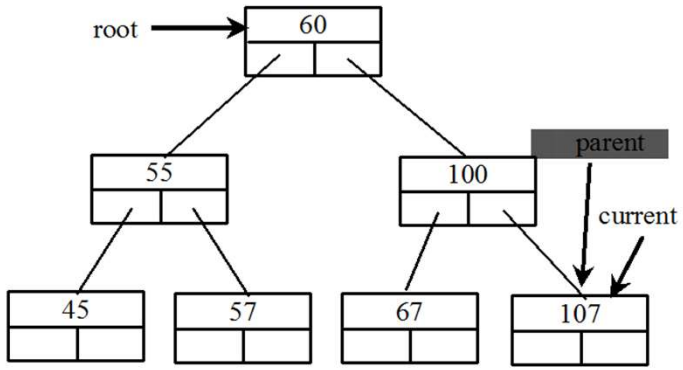


Trace Inserting 101 into the following tree
(15 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true

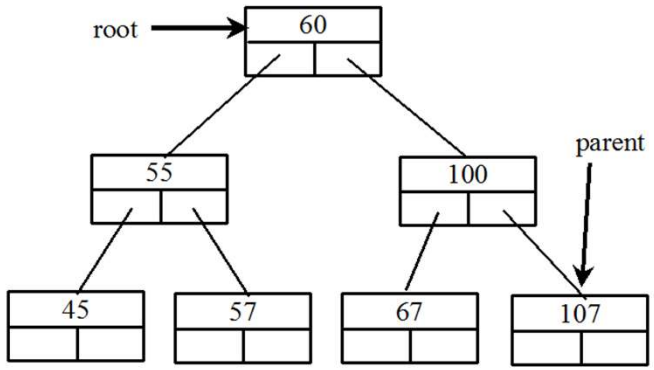


Trace Inserting 101 into the following tree
(16 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true



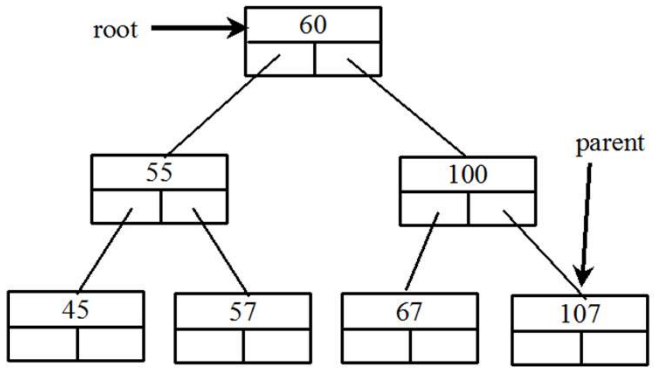
Since current.left is null,current becomes null

Trace Inserting 101 into the following tree
(17 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

current is null now

Insert 101 into the following tree.



Since current.left is null,current becomes null

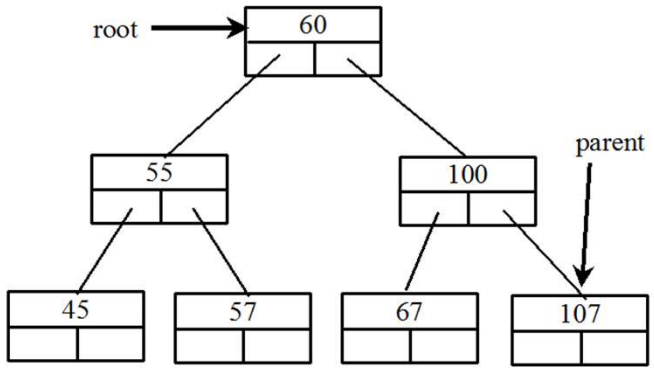
Trace Inserting 101 into the following tree

(18 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

101 < 107 true

Insert 101 into the following tree.



Since current.left is null,current becomes null

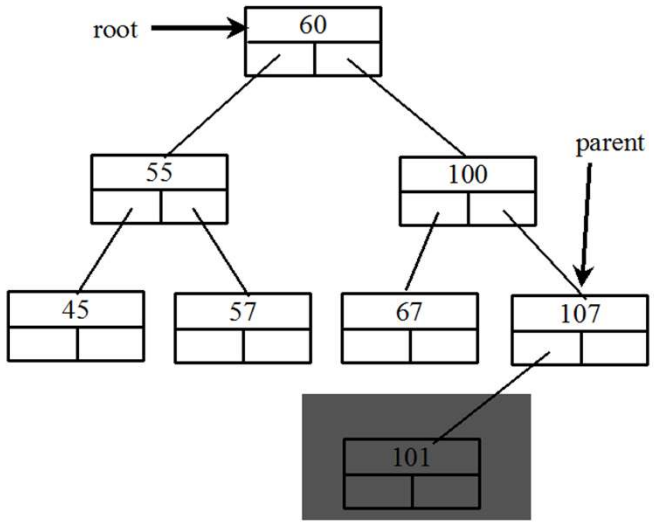
Trace Inserting 101 into the following tree

(19 of 20)

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true

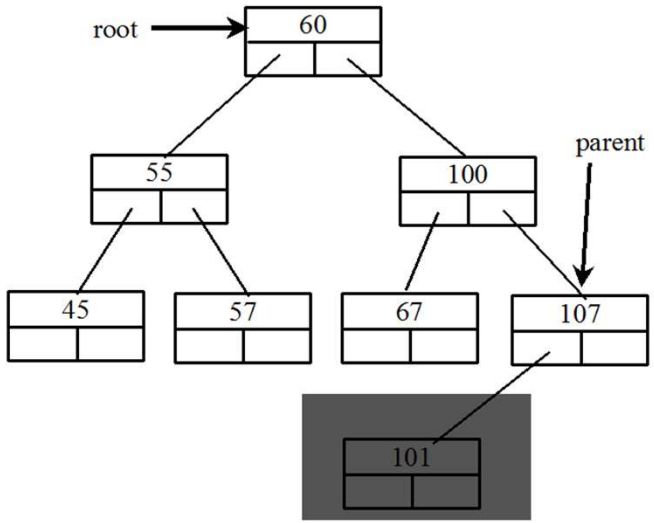


Trace Inserting 101 into the following tree
(20 of 20)

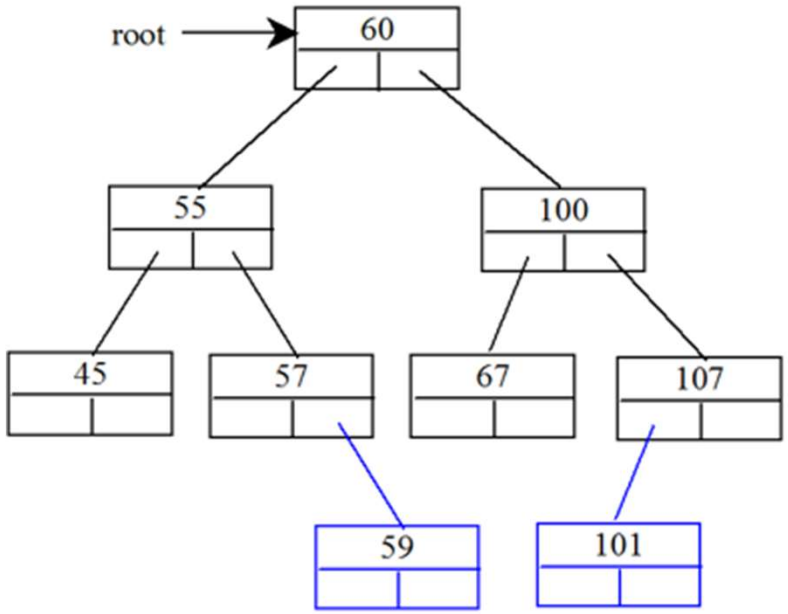
```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted

  // Create the new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

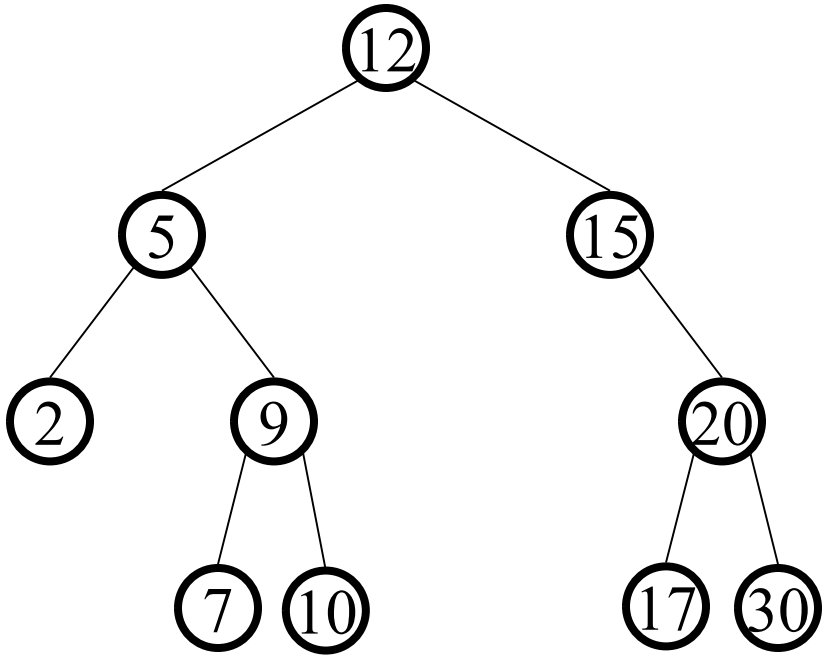
101 < 107 true



Inserting 59 into the Tree (Recursion)



Deletion in BST



Why might deletion be harder than insertion?

Deletion

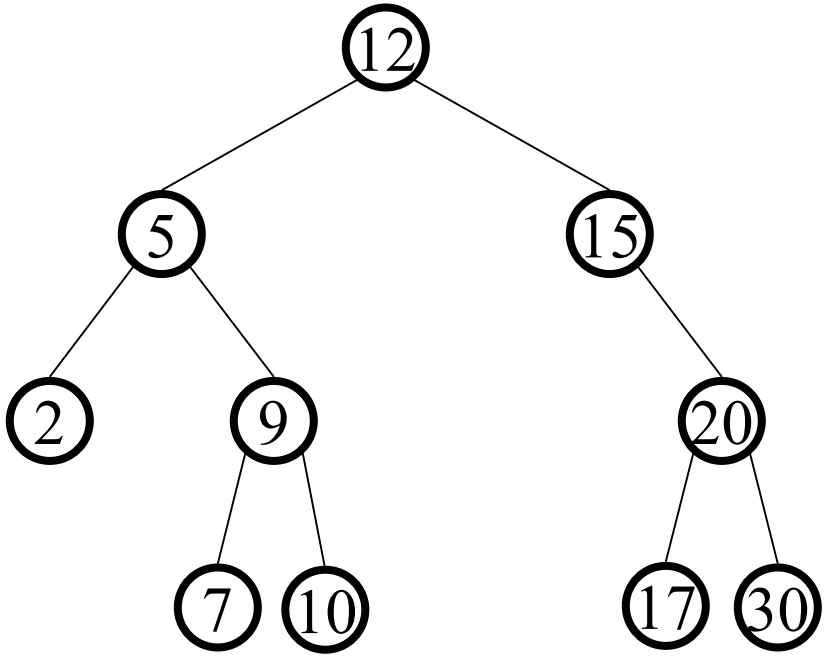
Removing an item disrupts the tree structure

Basic idea:

- find the node to be removed (current) and also its parent node (parent)
- Current node may be a left child or a right child of the parent node
- Remove it
- Fix the tree so that it is still a BST

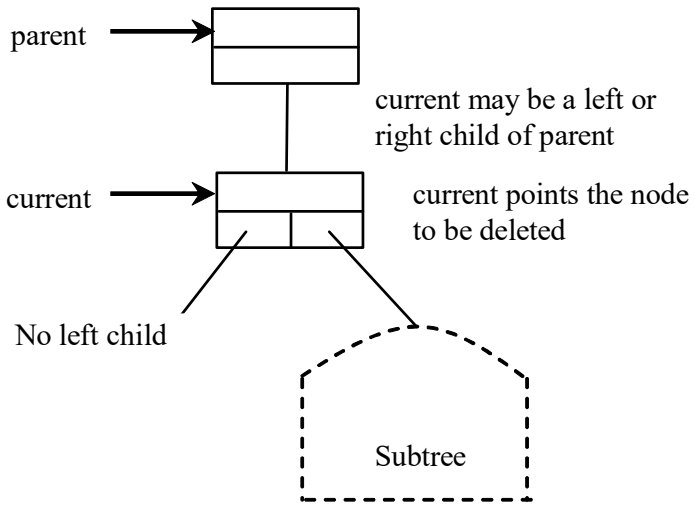
Two cases:

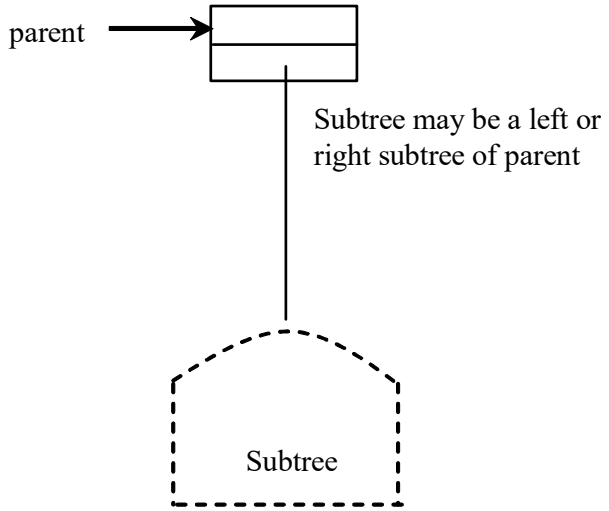
- node has no left child
  - Leaf (no right child)
- node has left child
  - Internal node



Deleting Elements in a Binary Search Tree

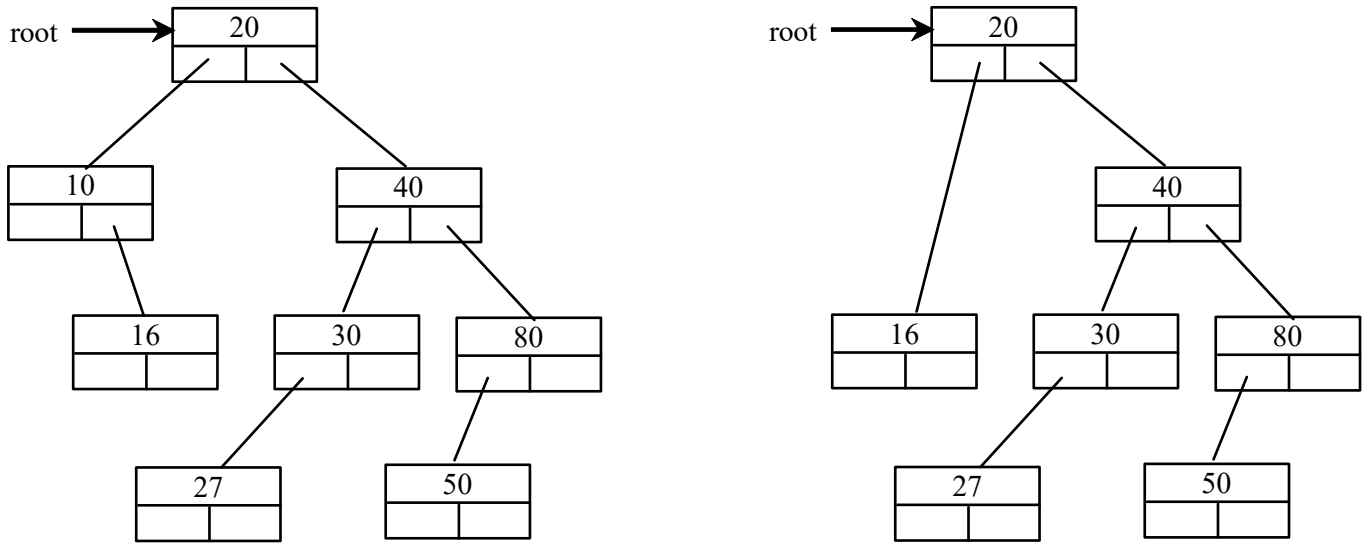
Case 1: The current node does not have a left child, as shown in this figure (a). Simply connect the parent with the right child of the current node, as shown in this figure (b).



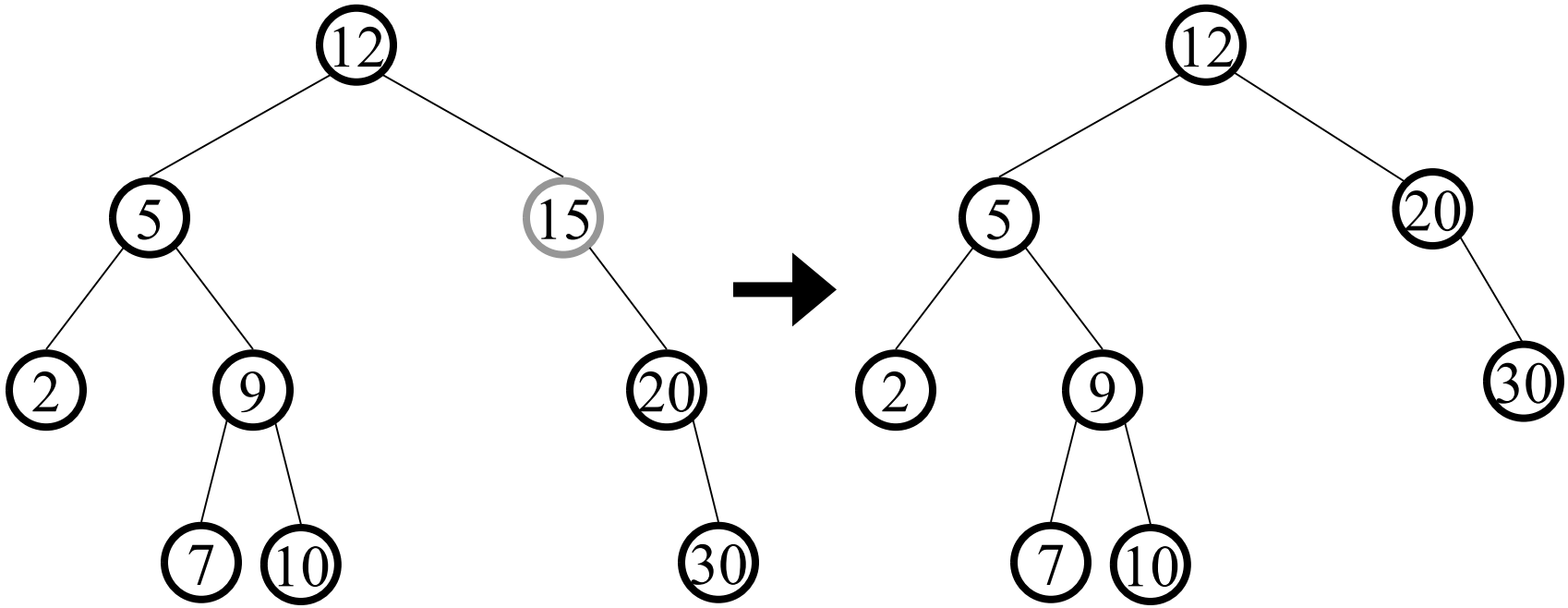


Deleting Elements in a Binary Search Tree

For example, to delete node 10. Connect the parent of node 10 with the right child of node 10.



Case 1: Delete 15



`delete(15)`

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the node's two child subtrees

Options are

- successor from right subtree
- predecessor from left subtree
- These are the easy cases of predecessor/successor

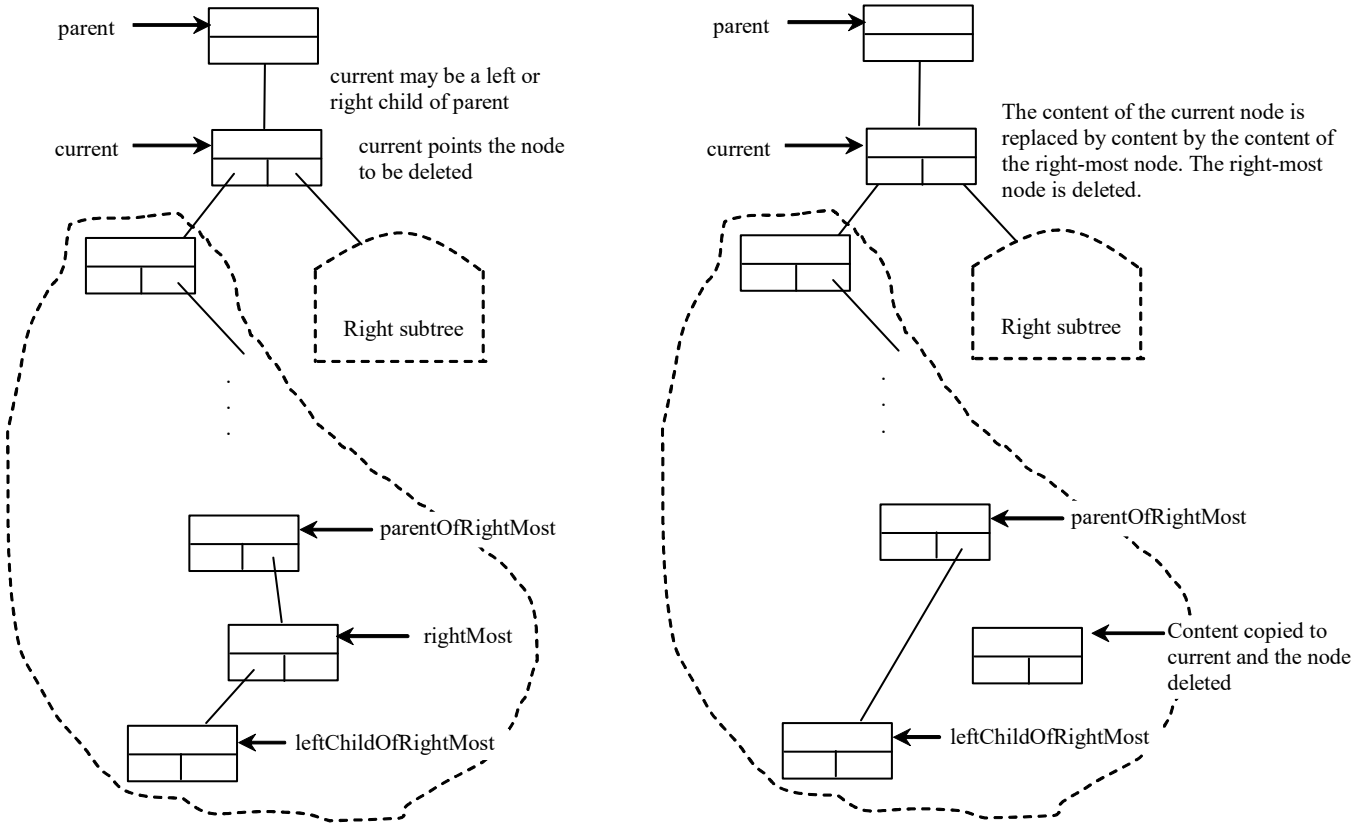
Either option is fine as both are guaranteed to exist in this case

Deleting Elements in a Binary Search Tree

Case 2: The current node has a left child. Let rightMost point to the node that contains the largest element in the left subtree of the current node and parentOfRightMost point to the parent node of the rightMost node. Note that the rightMost node cannot have a right child, but may have a left child. Replace the element value in the current node with the one in the rightMost node, connect the parentOfRightMost node with the left child of the rightMost node, and delete the rightMost node.

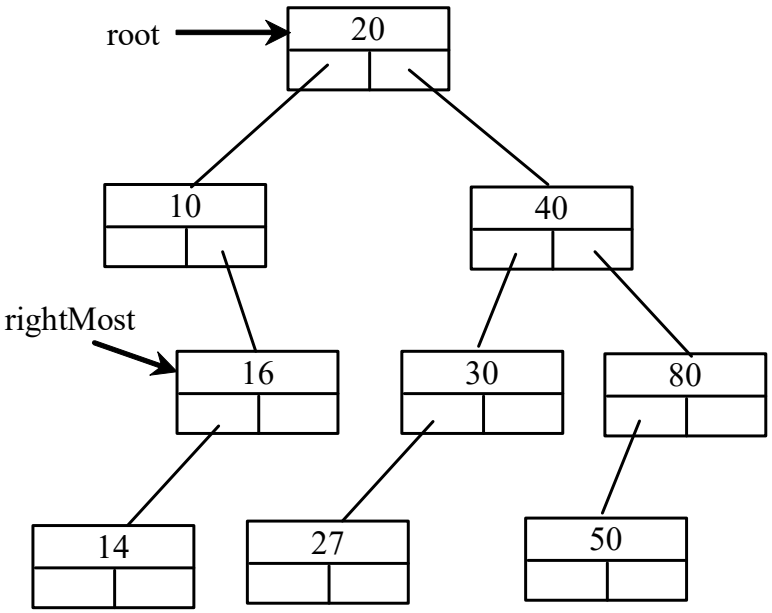
Deleting Elements in a Binary Search Tree

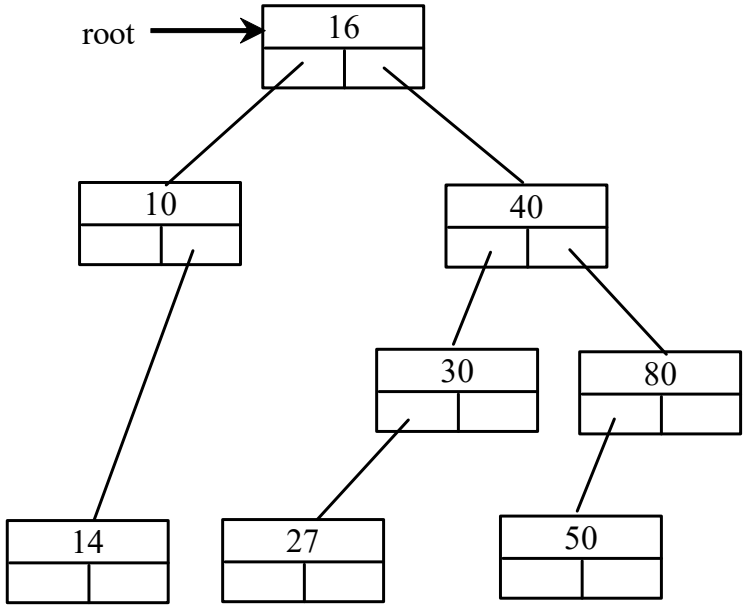
Case 2 diagram



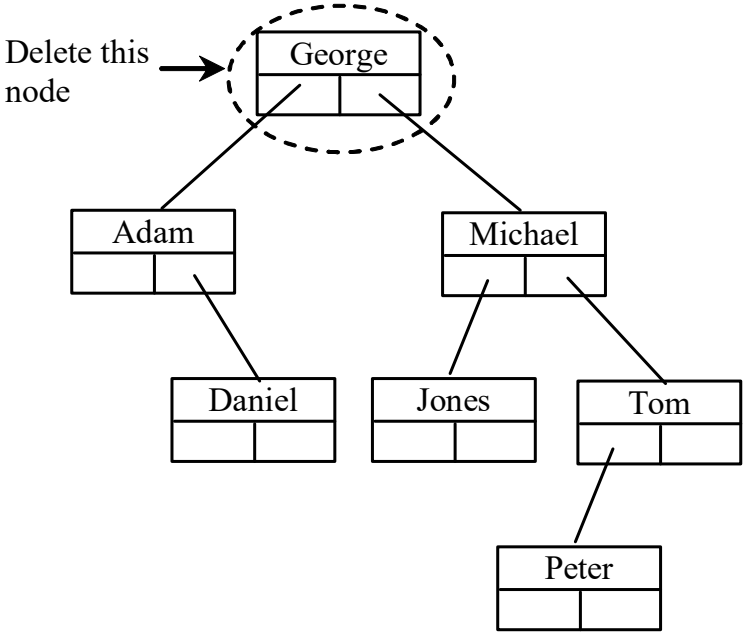
Deleting Elements in a Binary Search Tree

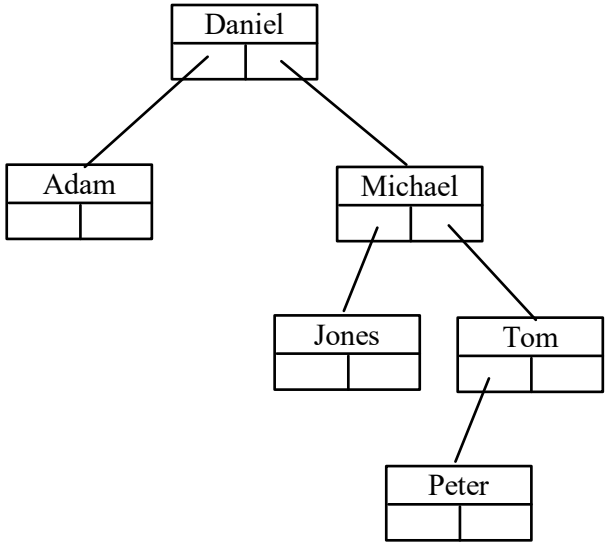
Case 2 example, delete 20



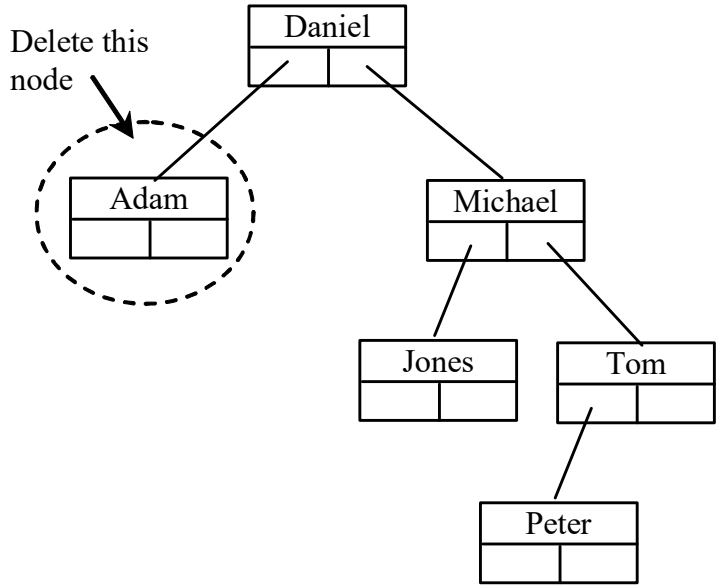


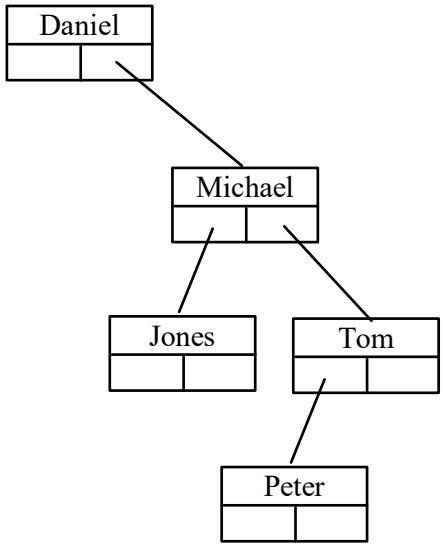
Examples (1 of 3)



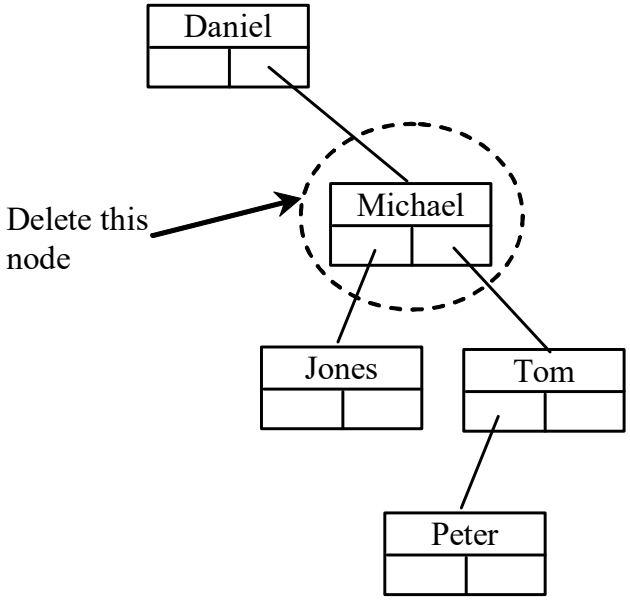


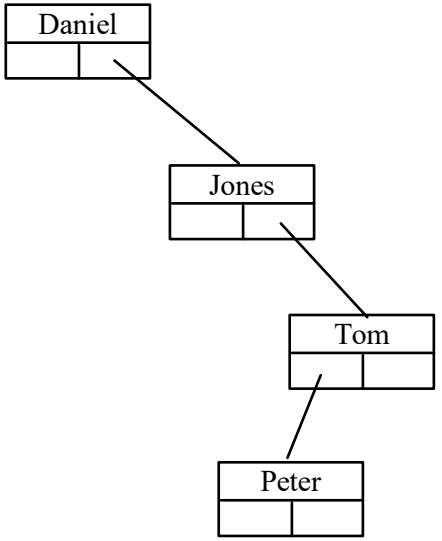
Examples (2 of 3)





Examples (3 of 3)



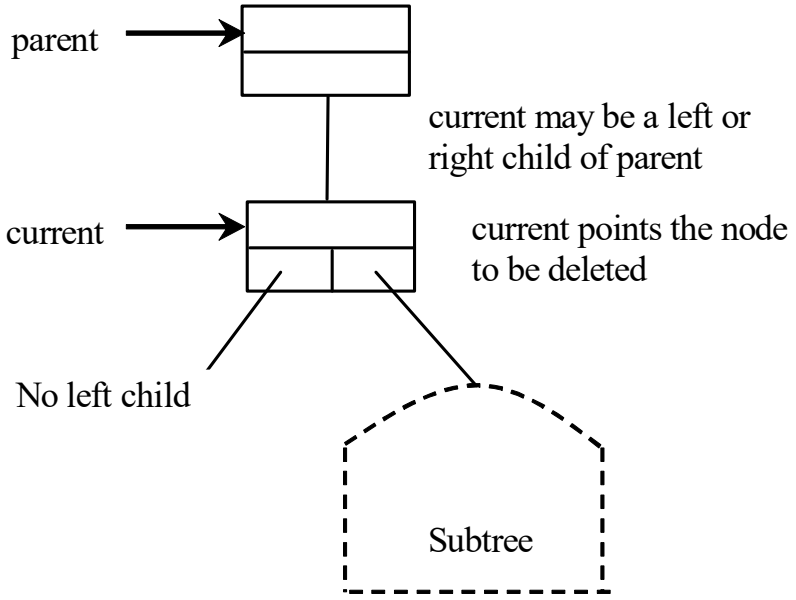


Deleting Elements in a Binary Search Tree

To delete an element from a binary tree:

1. First locate the node that contains the element and also its parent node.
   - Let current point to the node that contains the element in the binary tree
   - parent point to the parent of the current node.
   - The current node may be a left child or a right child of the parent node.
   - There are three cases to consider:
     - The node to be deleted is a leaf (has no children)
     - The node to be deleted has one child
     - The node to be deleted has two children

```
 * Return false if the element is not in the tree */
  public boolean delete(E e) {
    // Locate the node to be deleted and also locate its parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null) {
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        break; // Element is in the tree pointed by current
    }
    if (current == null)
        return false; // Element is not in the tree
```

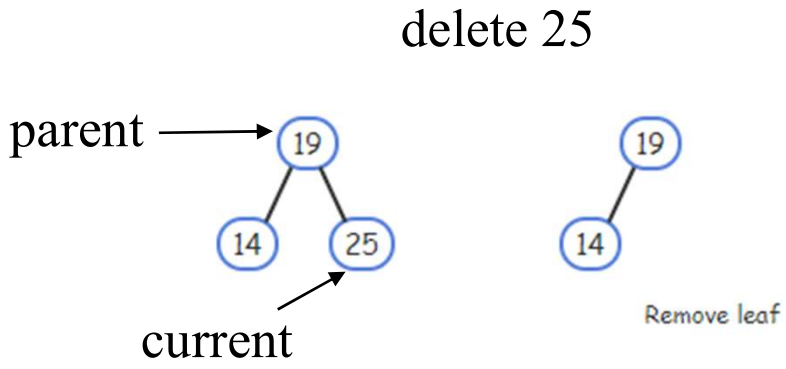


Deleting Elements in a Binary Search Tree

Case 1: The node to be deleted has no left child

- Change the appropriate child field of the node's parent to point to null

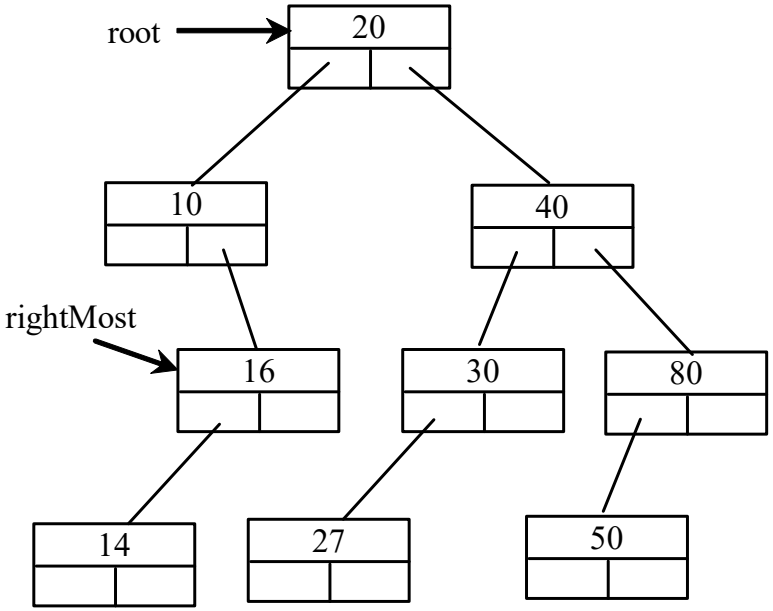
```
// Case 1: current has no left child
    if (current.left == null ) {
// Connect the parent with the right child of current
if (parent == null) {
        root = current.right;
    }
    else if (e < parent.element)
            parent.left = current.right;
        else
            parent.right = current.right;
}
 else
 {  // case 2
```

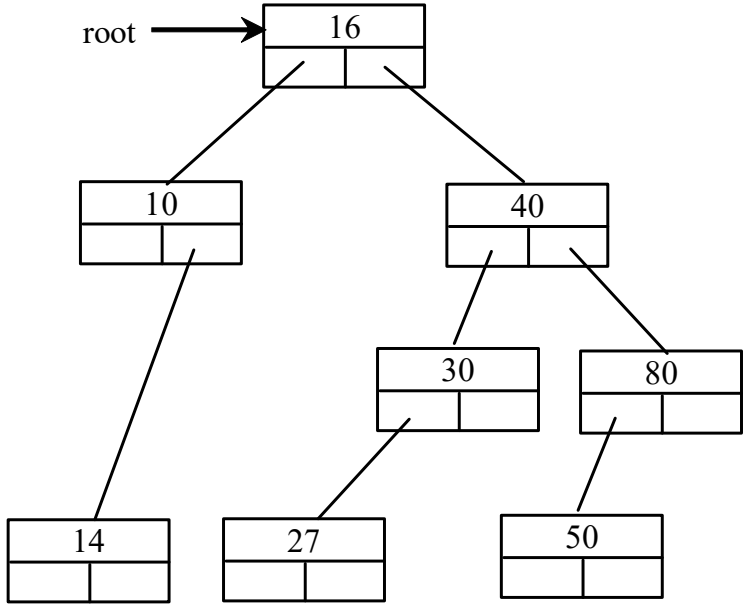


Deleting Elements in a Binary Search Tree

Case 2: The node to be deleted has left child

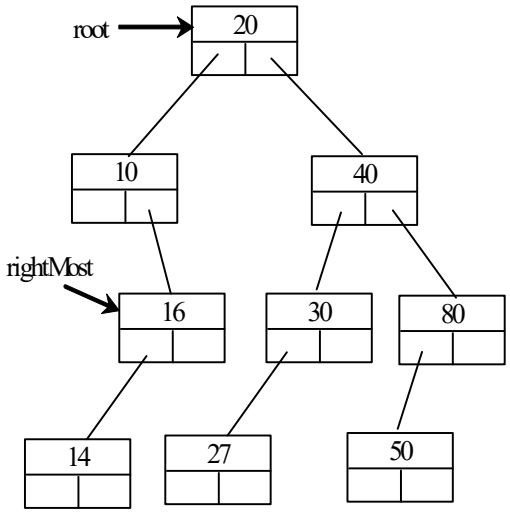
- Find predecessor: RightMost child in the left subtree of current node
- Copy predecessor to current nod
- Remove predecessor from right subtree





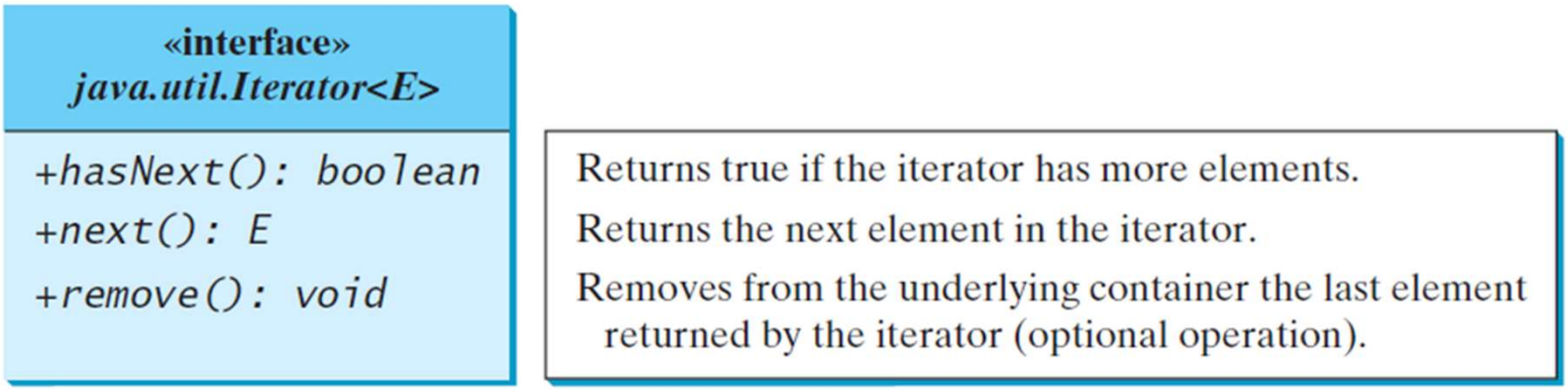
Case 2

```
  TreeNode<E> parentOfRightMost = current;
  TreeNode<E> rightMost = current.left;

   // Keep going to the right
   while (rightMost.right != null) {
      parentOfRightMost = rightMost;
      rightMost = rightMost.right;
    }

// Replace the element in current by the element in
// rightMost
    current.element = rightMost.element;

    // Eliminate rightmost node
    if (parentOfRightMost.right == rightMost)
      parentOfRightMost.right = rightMost.left;
     else
      // Special case: parentOfRightMost == current
      parentOfRightMost.left = rightMost.left;
   }

     size--;
     return true; // Element deleted
}
```



Iterators

An **iterator** is an object that provides a uniform way for traversing the elements in a container such as a set, list, binary tree, etc.



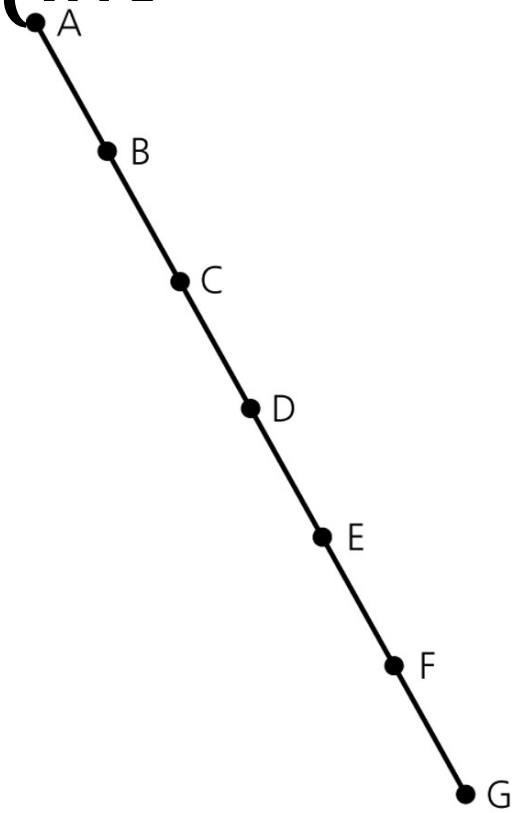
TestBSTWithIterator

binary tree time complexity

It is obvious that the time complexity for the inorder, preorder, and postorder is O(n), since each node is traversed only once. The time complexity for search, insertion and deletion is the height of the tree. In the worst case, the height of the tree is O(n).

The Efficiency of Binary Search Tree Operations

- The maximum number of comparisons for a retrieval, insertion, or deletion is the height of the tree
- The maximum and minimum heights of a binary search tree
  - n is the maximum height of a binary tree with n nodes



A maximum-height binary tree with seven nodes

The Efficiency of Binary Search Tree Operations

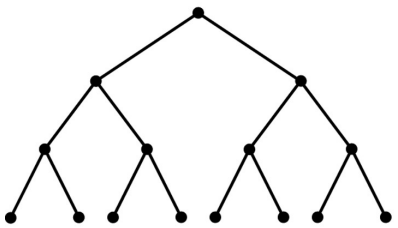
- Theorem

  A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes

- Theorem

  The maximum number of nodes that a binary tree of height h can have is $2^h - 1$

Counting the nodes in a full binary tree of height *h*



Level	Number of nodes at this level	Number of nodes at this and previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
.	.	.
.	.	.
.	.	.
h	2^{h-1}	$2^h - 1$

Summary

- Binary search trees come in many shapes
  - The height of a binary search tree with n nodes can range from a minimum of $\lceil \log_2(n + 1) \rceil$ to a maximum of n
  - The shape of a binary search tree determines the efficiency of its operations
- An inorder traversal of a binary search tree visits the tree's nodes in sorted search-key order
- The treesort algorithm efficiently sorts an array by using the binary search tree's insertion and traversal operations

Review

- What is the maximum number of nodes in a binary tree with 8 levels?
- What is the maximum and minimum number of levels of a tree with 2011 nodes?
- What is the max and min number of leaves of a tree with 10 levels?

Treesort

- Treesort
  - Uses the ADT binary search tree to sort an array of records into search-key order
  - Efficiency
    - Average case: O(n * log n)
    - Worst case: $O(n^2)$

The Efficiency of Binary Search Tree Operations

- Theorem 11-4

  The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1) \rceil$

- The height of a particular binary search tree depends on the order in which insertion and deletion operations are performed

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

Figure 11-34
The order of the retrieval, insertion, deletion, and traversal operations for the reference-based implementation of the ADT binary search tree