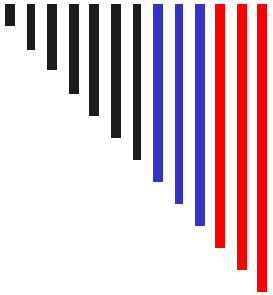


Unit 4

Trees

Objectives

Upon completion you will be able to

- **Explain Basic Tree concepts.**
- **Implement Binary search tree (BST)**
- **Understand AVL trees**

In a **non-linear list**, each element can have more than one successor.

In a **tree**, an element can have only one predecessor.

In a **graph**, an element can have one or more predecessors.

A **Tree** consists of a finite set of elements, called nodes, and a finite set of directed lines, called branches, that connect the nodes.

The number of branches associated with a node is the **degree** of the node.

When the branch is directed toward the node, it is an **indegree** branch

When the branch is directed away from the node, it is an **outdegree** branch.

The sum of the indegree and outdegree branches is the **degree** of the node.

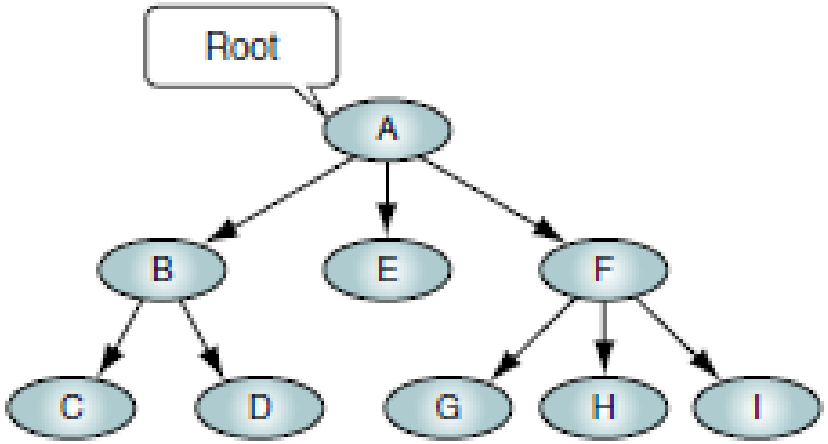


FIGURE 6-1 Tree

If the tree is not empty, the first node is called the **root.** The indegree of the root is zero.

A **leaf** is any node with an outdegree of zero, that is, a node with no successors.

A node that is not a root or a leaf is known as an **internal** node.

A node is a **parent** if it has successor nodes—that is, if it has an outdegree greater than zero.

A node with a predecessor is a **child**. A child node has an indegree of one.

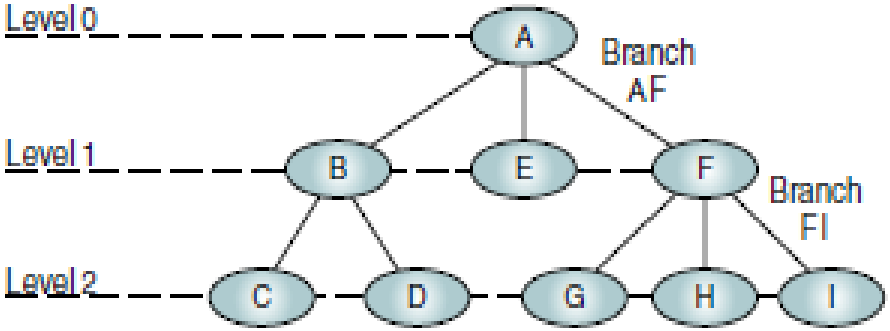
Two or more nodes with the same parent are **siblings**.

An **ancestor** is any node in the path from the root to the node.

A **descendant** is any node in the path below the parent node; that is, all nodes in the paths from a given node to a leaf are descendants of that node.

A **path** is a sequence of nodes in which each node is adjacent to the next one. Every node in the tree can be reached by following a unique path starting from the root.

The **level** of a node is its distance from the root. Because the root has a zero distance from itself, the root is at level 0.



Root: A	Siblings: {B,E,F}, {C,D}, {G,H,I}
Parents: A, B, F	Leaves: C,D,E,G,H,I
Children: B, E, F, C, D, G, H, I	Internal nodes: B,F

FIGURE 6-2 Tree Nomenclature

The **height** of the tree is the level of the leaf in the longest path from the root plus 1.

By definition the height of an empty tree is -1.

A tree may be divided into **subtrees**. A subtree is any connected structure below the root. The first node in a subtree is known as the root of the subtree and is used to name the subtree.

Subtrees can also be further subdivided into subtrees.

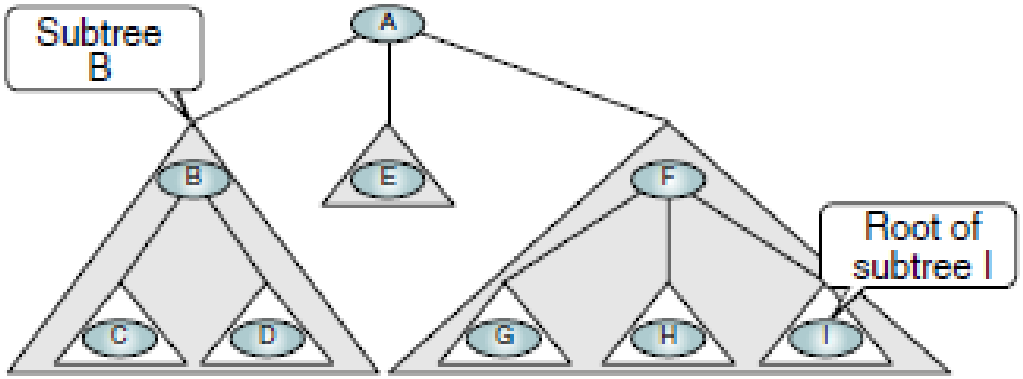


FIGURE 6-3 Subtrees

A tree is a set of nodes that either:
1. Is empty, or
2. Has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees

A **Binary Tree** is a tree in which no node can have more than two subtrees; the maximum outdegree for a node is two. In other words, a node can have zero, one, or two subtrees. These subtrees are designated as the left subtree and the right subtree.

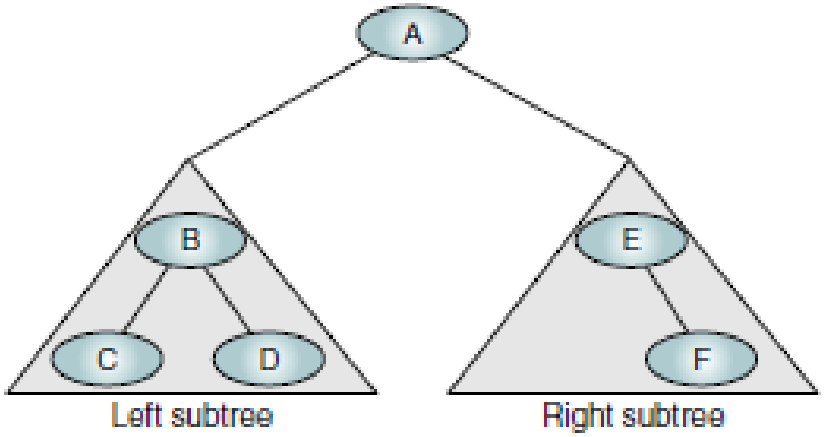


FIGURE 6-5 Binary Tree

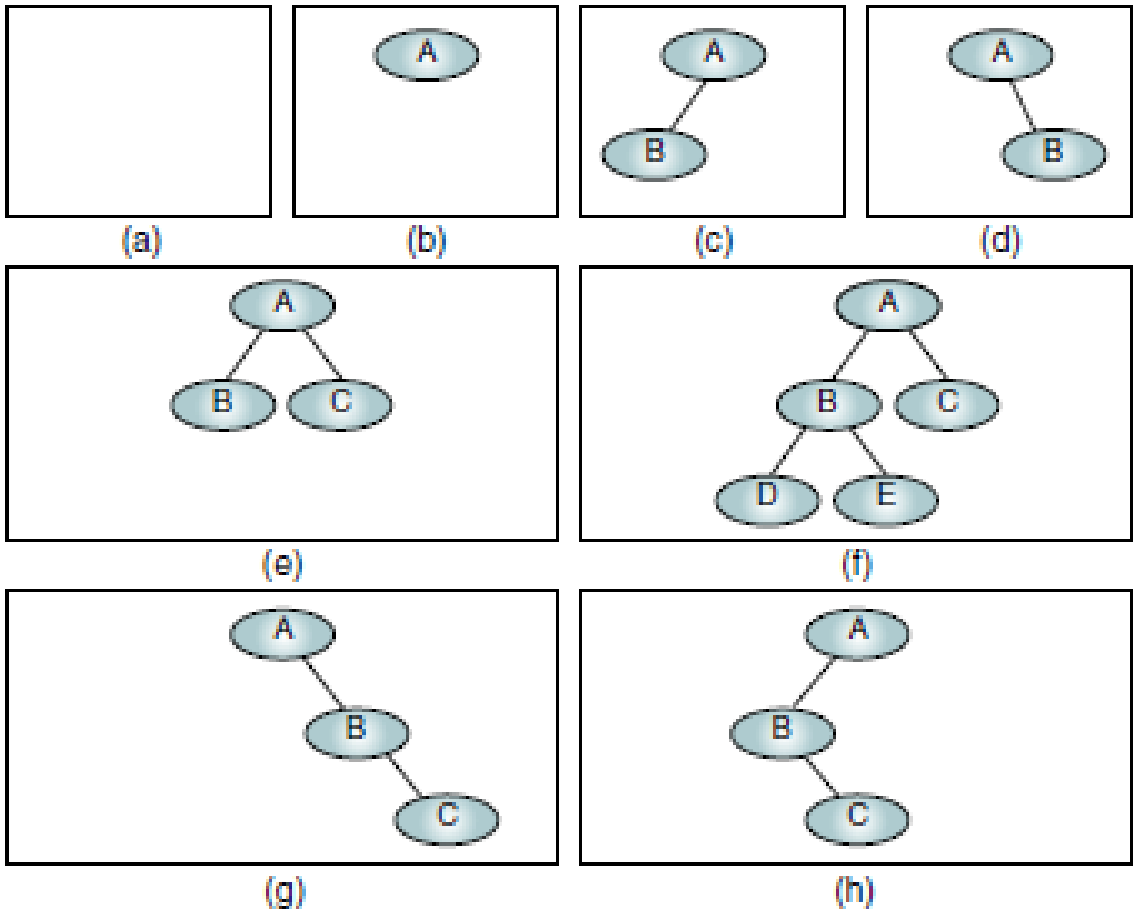


FIGURE 6-6 Collection of Binary Trees

Null Tree is a tree with no nodes

Properties of Binary Trees

Height of Binary Trees

Maximum Height

Given that we need to store *N* nodes in a binary tree, the maximum height, *H*max, is

$$H_{max} = N$$

Minimum Height

$$H_{min} = \lfloor \log_2 N \rfloor + 1$$

Minimum Number of Nodes

$$N_{min} = H$$

Maximum Number of Nodes

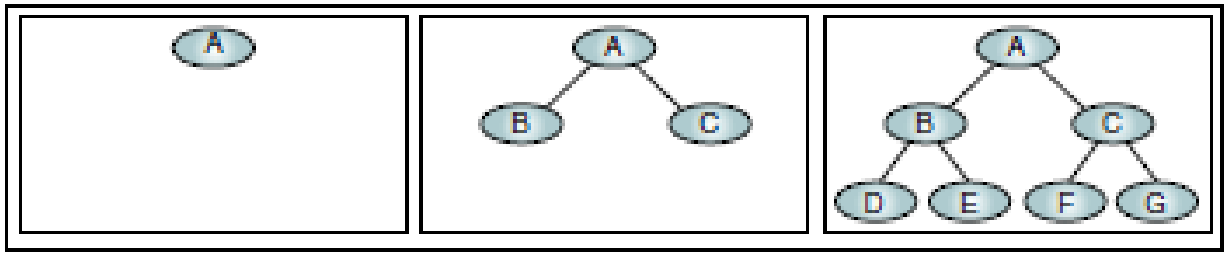
$$N_{max} = 2^H - 1$$

The balance factor of a binary tree is the difference in height between its left and right subtrees.

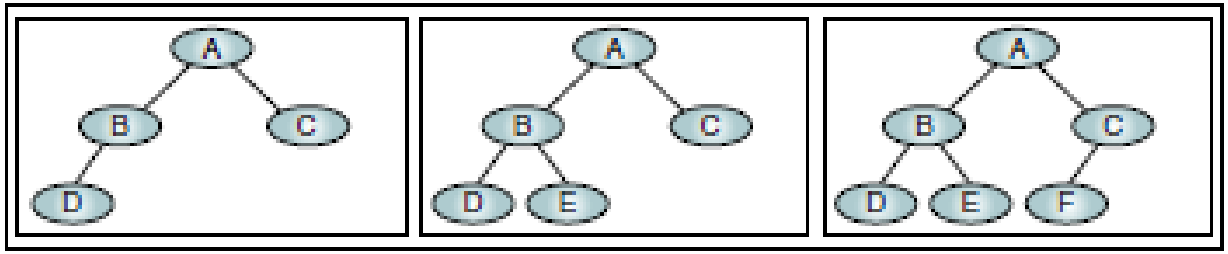
$$B = H_L - H_R$$

In a balanced binary tree, the height of its subtrees differs by no more than one (its balance factor is –1, 0, or +1), and its subtrees are also balanced.

A **Complete Tree** has the maximum number of entries for its height (see formula *N*max in "Height of Binary Trees"). The maximum number is reached when the last level is full (see Figure 6-7). A tree is considered **Nearly Complete** if it has the minimum height for its nodes (see formula *H*min) and all nodes in the last level are found on the left.



(a) Complete trees (at levels 0, 1, and 2)



(b) Nearly complete trees (at level 2)

FIGURE 6-7 Complete and Nearly Complete Trees

Binary Tree Traversals

A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence. The two general approaches to the traversal sequence are depth first and breadth first.

In the **depth-first traversal**, the processing proceeds along a path from the root through one child to the most distant descendent of that first child before processing a second child. In other words, in the depth-first traversal, we process all of the descendants of a child before going on to the next child.

In a **breadth-first traversal**, the processing proceeds horizontally from the root to all of its children, then to its children's children, and so forth until all nodes have been processed. In other words, in the breadth-first traversal, each level is completely processed before the next level is started.

Depth-first Traversals

Given that a binary tree consists of a root, a left subtree, and a right subtree, we can define six different depth-first traversal sequences. Computer scientists have assigned three of these sequences standard names in the literature.

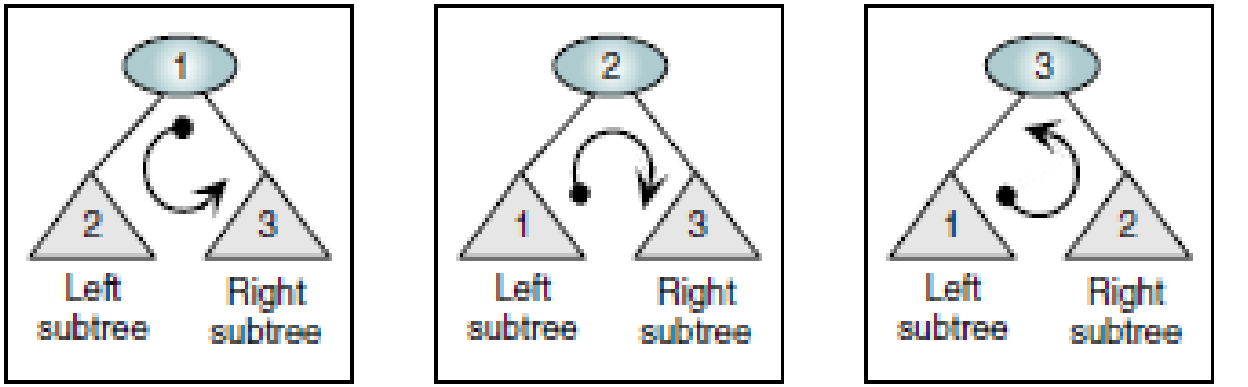


FIGURE 6-8 Binary Tree Traversals

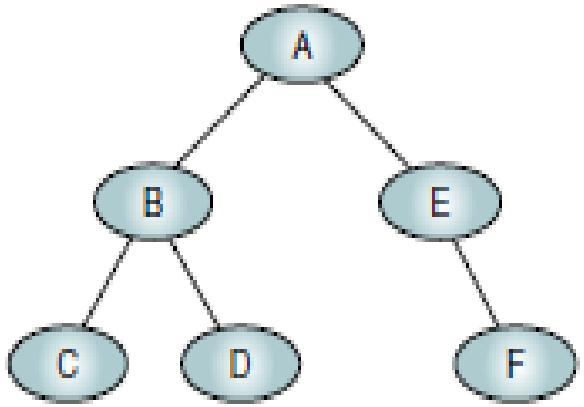


FIGURE 6-9 Binary Tree for Traversals

Preorder Traversal (NLR)

In the preorder traversal, the root node is processed first, followed by the left subtree and then the right subtree. It draws its name from the Latin prefix *pre*, which means to go before. Thus, the root goes before the subtrees.

ALGORITHM 6-2 Preorder Traversal of a Binary Tree

```
Algorithm preOrder (root)
Traverse a binary tree in node-left-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
1  if (root is not null)
   1   process (root)
   2   preOrder (leftSubtree)
   3   preOrder (rightSubtree)
2  end if
end preOrder
```

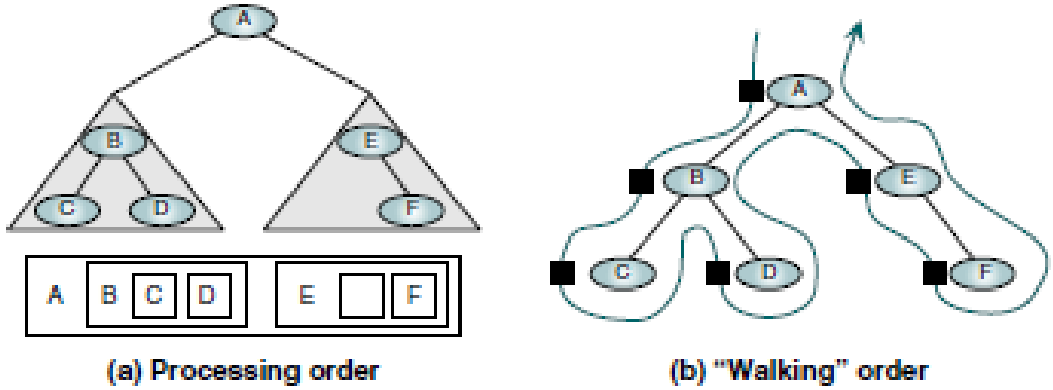


FIGURE 6-10 Preorder Traversal—A B C D E F

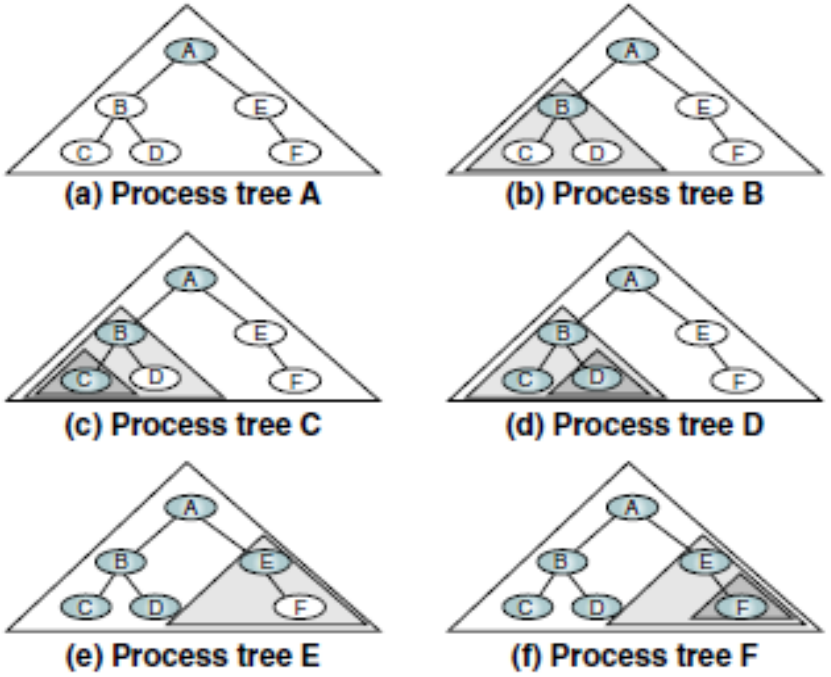


FIGURE 6-11 Algorithmic Traversal of Binary Tree

Inorder Traversal (LNR)

The inorder traversal processes the left subtree first, then the root, and finally the right subtree. The meaning of the prefix *in* is that the root is processed *in between* the subtrees.

ALGORITHM 6-3 Inorder Traversal of a Binary Tree

```
Algorithm inOrder (root)
Traverse a binary tree in left-node-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
1  if (root is not null)
   1   inOrder (leftSubTree)
   2   process (root)
   3   inOrder (rightSubTree)
2  end if
end inOrder
```

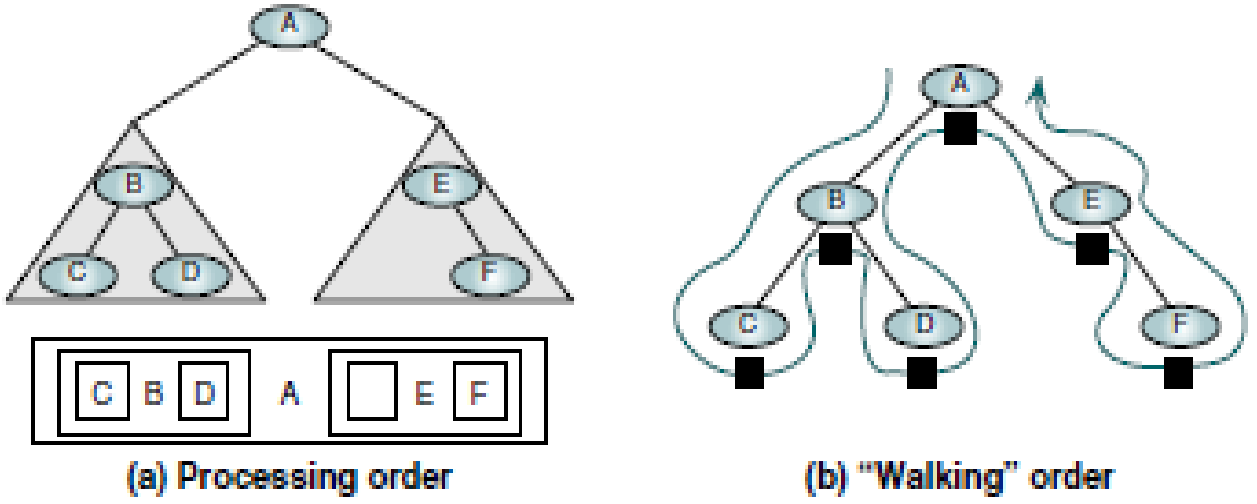


FIGURE 6-12 Inorder Traversal—C B D A E F

Postorder Traversal (LRN)

The last of the standard traversals is the postorder traversal. It processes the root node after (*post*) the left and right subtrees have been processed. It starts by locating the far-left leaf and processing it. It then processes its right sibling, including its subtrees (if any). Finally, it processes the root node.

ALGORITHM 6-4 Postorder Traversal of a Binary Tree

```
Algorithm postOrder (root)
Traverse a binary tree in left-right-node sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
 1 if (root is not null)
   1  postOrder (left subtree)
   2  postOrder (right subtree)
   3  process (root)
 2 end if
end postOrder
```

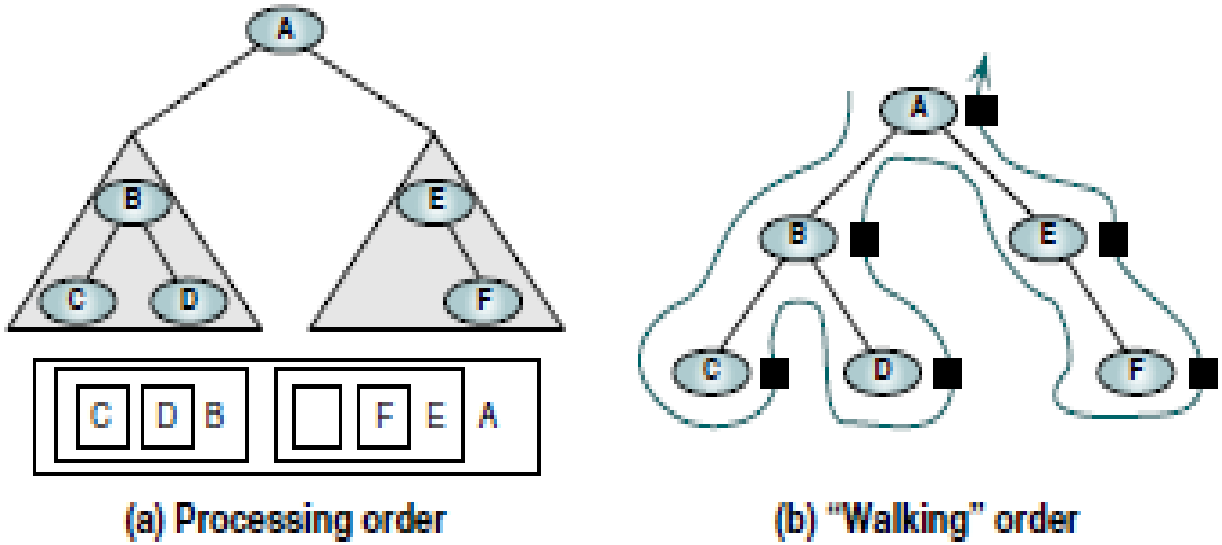


FIGURE 6-13 Postorder Traversal—C D B F E A

Breadth-first Traversals

In the breadth-first traversal of a binary tree, we process all of the children of a node before proceeding with the next level. In other words, given a root at level *n*, we process all nodes at level *n* before proceeding with the nodes at level *n* + 1. To traverse a tree in depth-first order, we used a stack. (Remember that recursion uses a stack.) To traverse a tree in breadth-first order, we use a queue.

ALGORITHM 6-5 Breadth-first Tree Traversal

```
Algorithm breadthFirst (root)
Process tree using breadth-first traversal.
   Pre    root is node to be processed
   Post   tree has been processed
1  set currentNode to root
2  createQueue (bfQueue)
3  loop (currentNode not null)
   1  process (currentNode)
   2  if (left subtree not null)
      1  enqueue (bfQueue, left subtree)
   3  end if
   4  if (right subtree not null)
      1  enqueue (bfQueue, right subtree)
   5  end if
   6  if (not emptyQueue(bfQueue))
      1  set currentNode to dequeue (bfQueue)
   7  else
      1  set currentNode to null
   8  end if
4  end loop
5  destroyQueue (bfQueue)
end breadthFirst
```

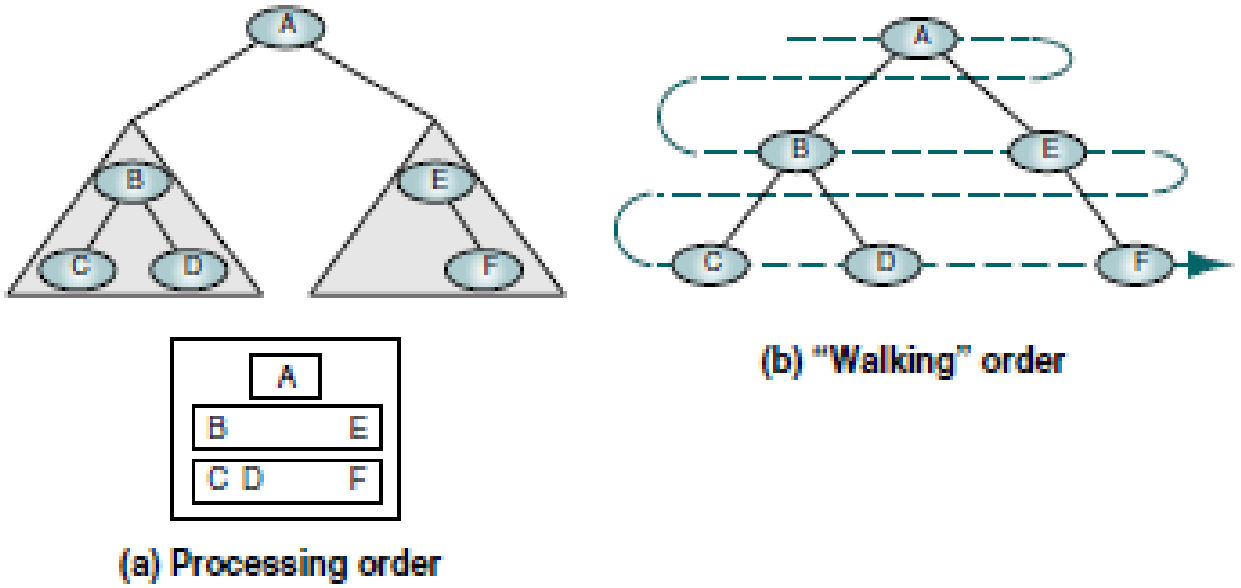


FIGURE 6-14 Breadth-first Traversal

Expression Trees

One interesting application of binary trees is expression trees. An expression is a sequence of tokens that follow prescribed rules. A token may be either an operand or an operator. In this discussion we consider only binary arithmetic operators in the form operand–operator–operand. The standard arithmetic operators are +, -, *, and /.

An expression tree is a binary tree with the following properties:

- Each leaf is an operand.
- The root and internal nodes are operators.
- Subtrees are subexpressions, with the root being an operator.

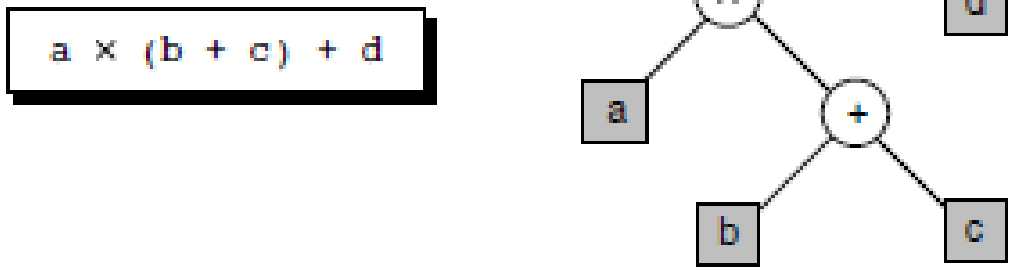
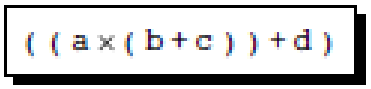


FIGURE 6-15 Infix Expression and Its Expression Tree



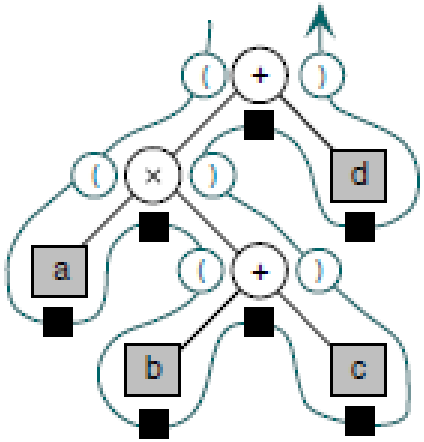


FIGURE 6-16 Infix Traversal of an Expression Tree

ALGORITHM 6-6 Infix Expression Tree Traversal

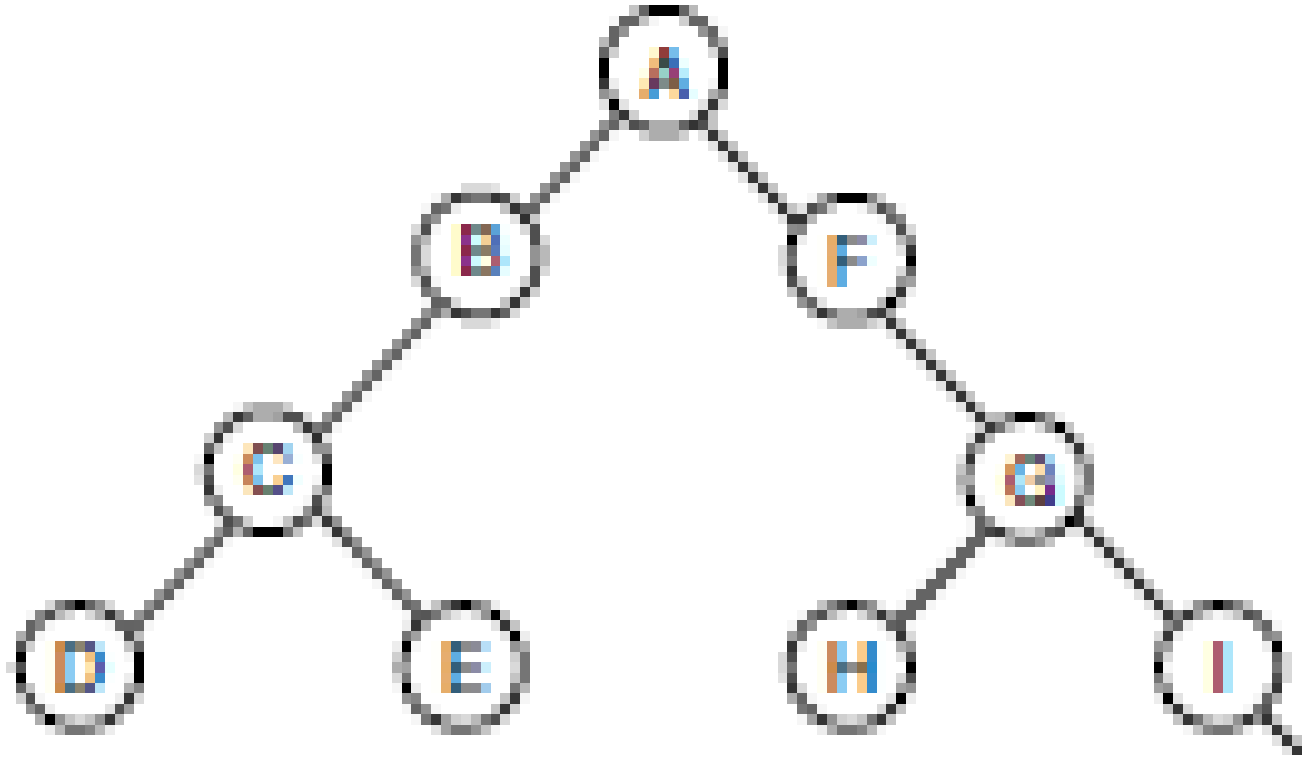
```
Algorithm infix (tree)
Print the infix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the infix expression has been printed
 1 if (tree not empty)
   1   if (tree token is an operand)
       1   print (tree-token)
   2   else
       1   print (open parenthesis)
       2   infix (tree left subtree)
       3   print (tree token)
       4   infix (tree right subtree)
       5   print (close parenthesis)
   3   end if
 2 end if
end infix
```

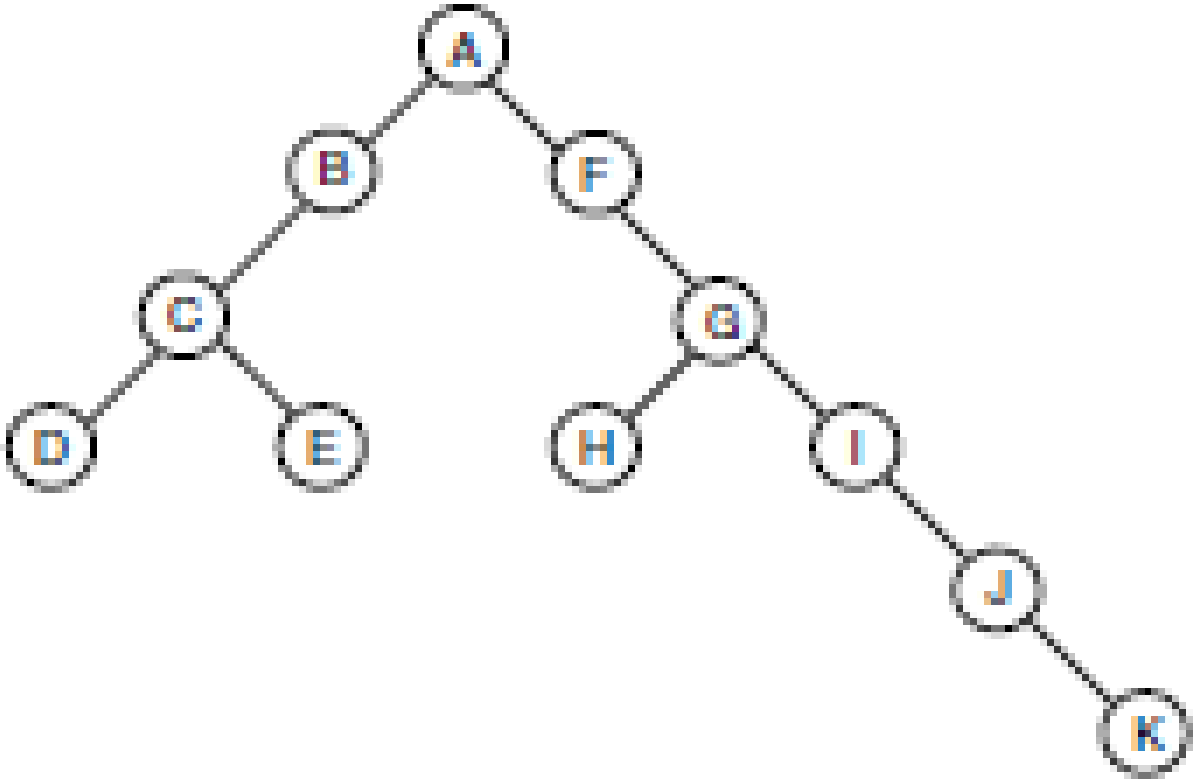
ALGORITHM 6-7 Postfix Traversal of an Expression Tree

```
Algorithm postfix (tree)
Print the postfix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the postfix expression has been printed
 1 if (tree not empty)
   1  postfix (tree left subtree)
   2  postfix (tree right subtree)
   3  print   (tree token)
 2 end if
end postfix
```

ALGORITHM 6-8 Prefix Traversal of an Expression Tree

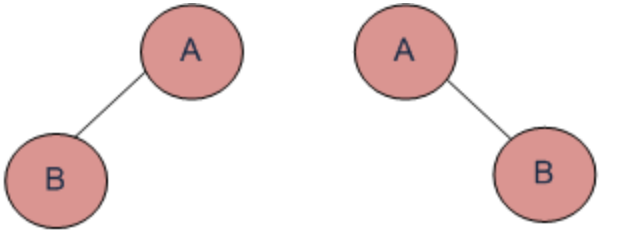
```
Algorithm prefix (tree)
Print the prefix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the prefix expression has been printed
1  if (tree not empty)
   1  print  (tree token)
   2  prefix (tree left subtree)
   3  prefix (tree right subtree)
2  end if
end prefix
```


A
B
F
C
G
D
E
H
I


A
B
F
C
G
D
E
H
I
J
K

If you are given two traversal sequences, can you construct the binary tree?

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.



Trees having Preorder, Postorder and Level-Order and traversals

Therefore, following combination can uniquely identify a tree.
Inorder and Preorder.
Inorder and Postorder.
Inorder and Level-order.

And following do not.
Postorder and Preorder.
Preorder and Level-order.
Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.
Preorder Traversal = AB
Postorder Traversal = BA
Level-Order Traversal = AB
So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

Construct Tree from given Inorder and Preorder traversals

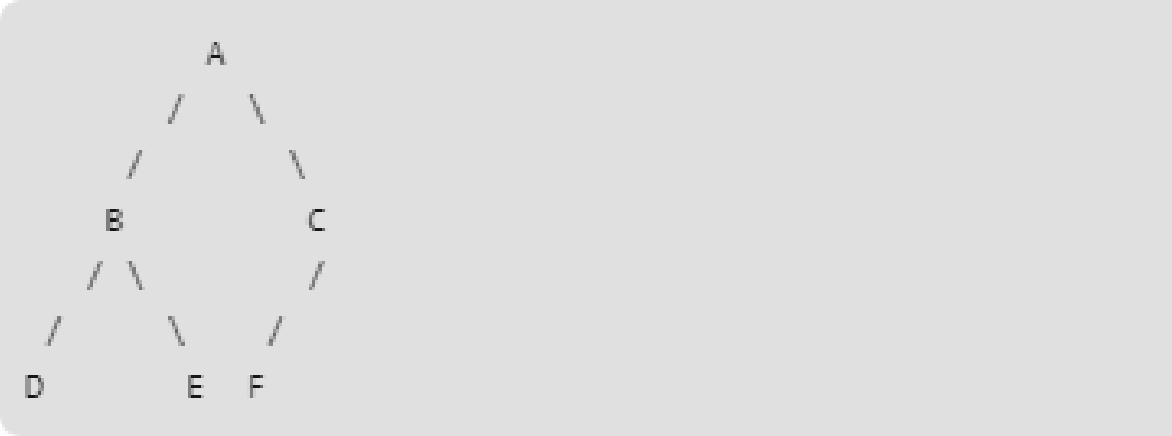
Let us consider the below traversals:

Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element is the root of the tree. So we know 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So we know below structure now.

We recursively follow above steps and get the following tree.



BINARY SEARCH TREE (BST)

In the design of the linear list structure, we had two choices: an array or a linked list.

The array structure provides a very efficient search algorithm but its insertion and deletion algorithms are very inefficient.

On the other hand, the linked list structure provides efficient insertion and deletion, but its search algorithm is very inefficient.

What we need is a structure that provides an efficient search algorithm and at the same time efficient insert and delete algorithms.

The binary search tree and the AVL tree provide that structure

BINARY SEARCH TREE (BST)

A binary search tree (BST) is a binary tree with the following properties:

- All items in the left subtree are less than the root.
- All items in the right subtree are greater than or equal to the root.
- Each subtree is itself a binary search tree.

In a binary search tree, the left subtree contains key values less than the root and the right subtree contains key values greater than or equal to the root.

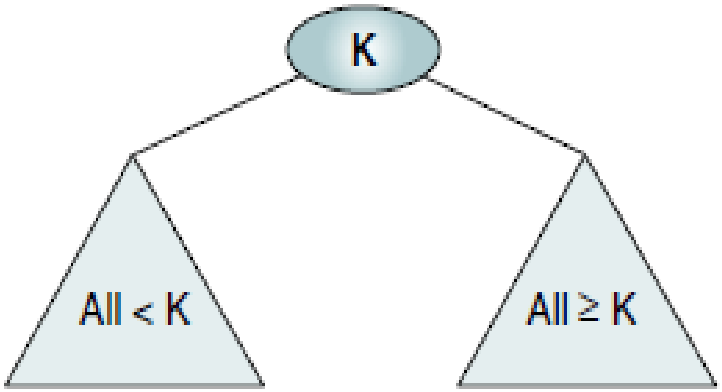


FIGURE 7-1 Binary Search Tree

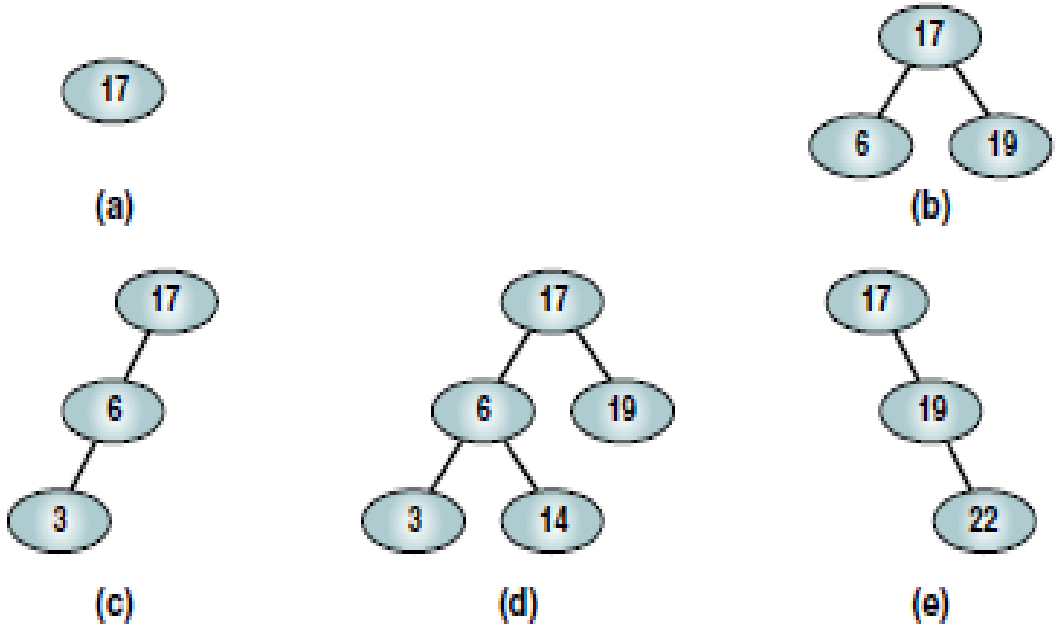


FIGURE 7-2 Valid Binary Search Trees

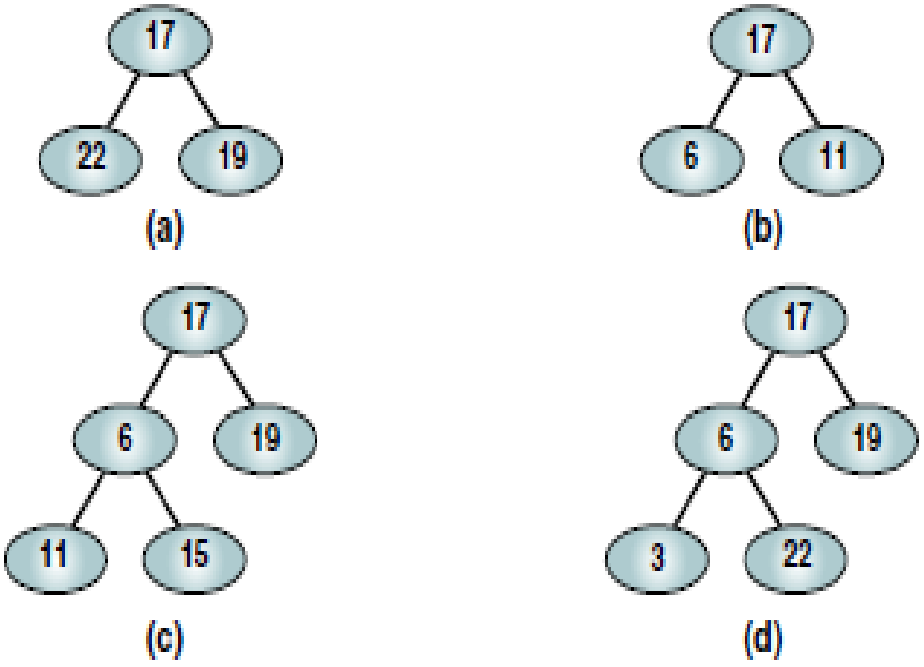


FIGURE 7-3 Invalid Binary Search Trees

BST Operations

Traversals

Searches

- Find the Smallest Node
- Find the Largest Node
- BST Search

Insertion

Deletion

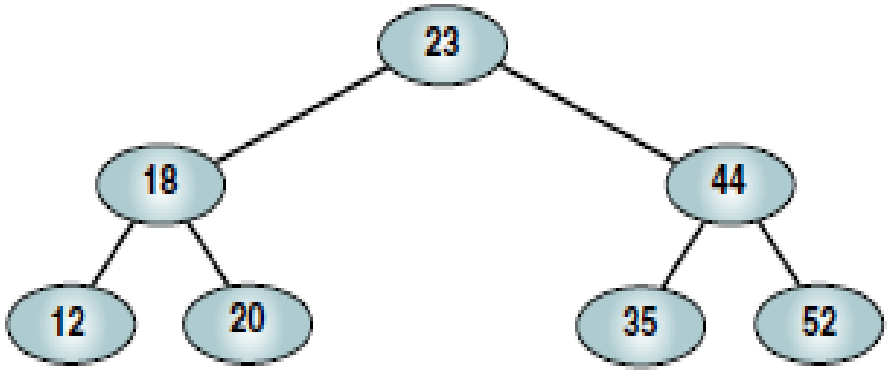


FIGURE 7-4 Example of a Binary Search Tree

Preorder BST Traversal

```
23 18 12 20 44 35 52
```

Postorder BST Traversal

```
12 20 18 35 52 44 23
```

Inorder BST Traversal

```
12 18 20 23 35 44 52
```

ALGORITHM 7-1 Find Smallest Node in a BST

```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
1  if (left subtree empty)
   1   return (root)
2  end if
3  return findSmallestBST (left subtree)
end findSmallestBST
```

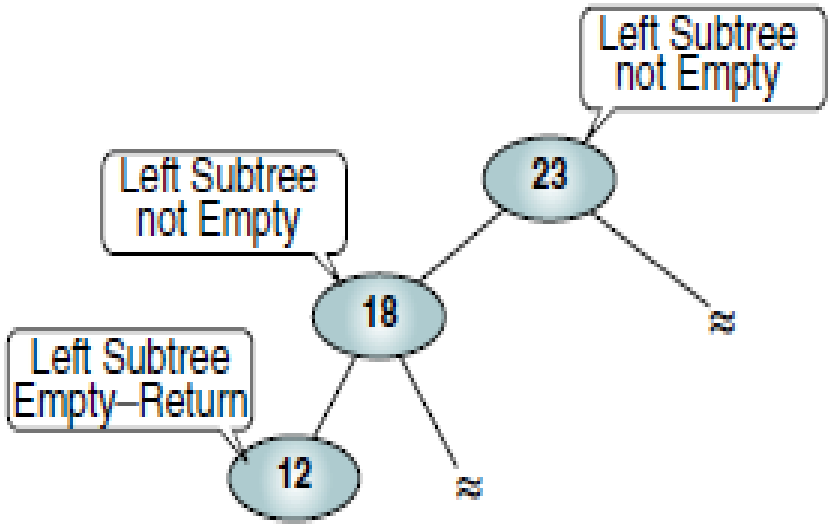


FIGURE 7-5 Find Smallest Node in a BST

ALGORITHM 7-2 Find Largest Node in a BST

```
Algorithm findLargestBST (root)
This algorithm finds the largest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of largest node returned
 1 if (right subtree empty)
   1  return (root)
 2 end if
 3 return findLargestBST (right subtree)
end findLargestBST
```

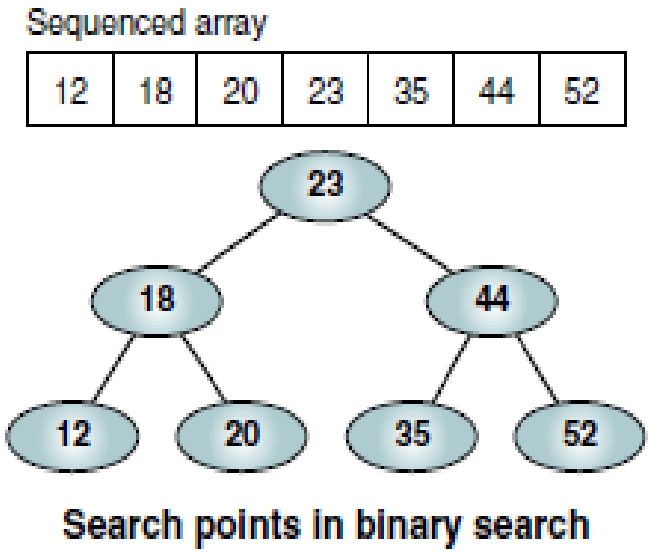


FIGURE 7-6 BST and the Binary Search

ALGORITHM 7-3 Search BST

```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
   Pre    root is the root to a binary tree or subtree
          targetKey is the key value requested
   Return the node address if the value is found
          null if the node is not in the tree
1  if (empty tree)
      Not found
   1  return null
2  end if
3  if (targetKey < root)
   1  return searchBST (left subtree, targetKey)
4  else if (targetKey > root)
   1  return searchBST (right subtree, targetKey)
5  else
      Found target key
   1  return root
6  end if
end searchBST
```

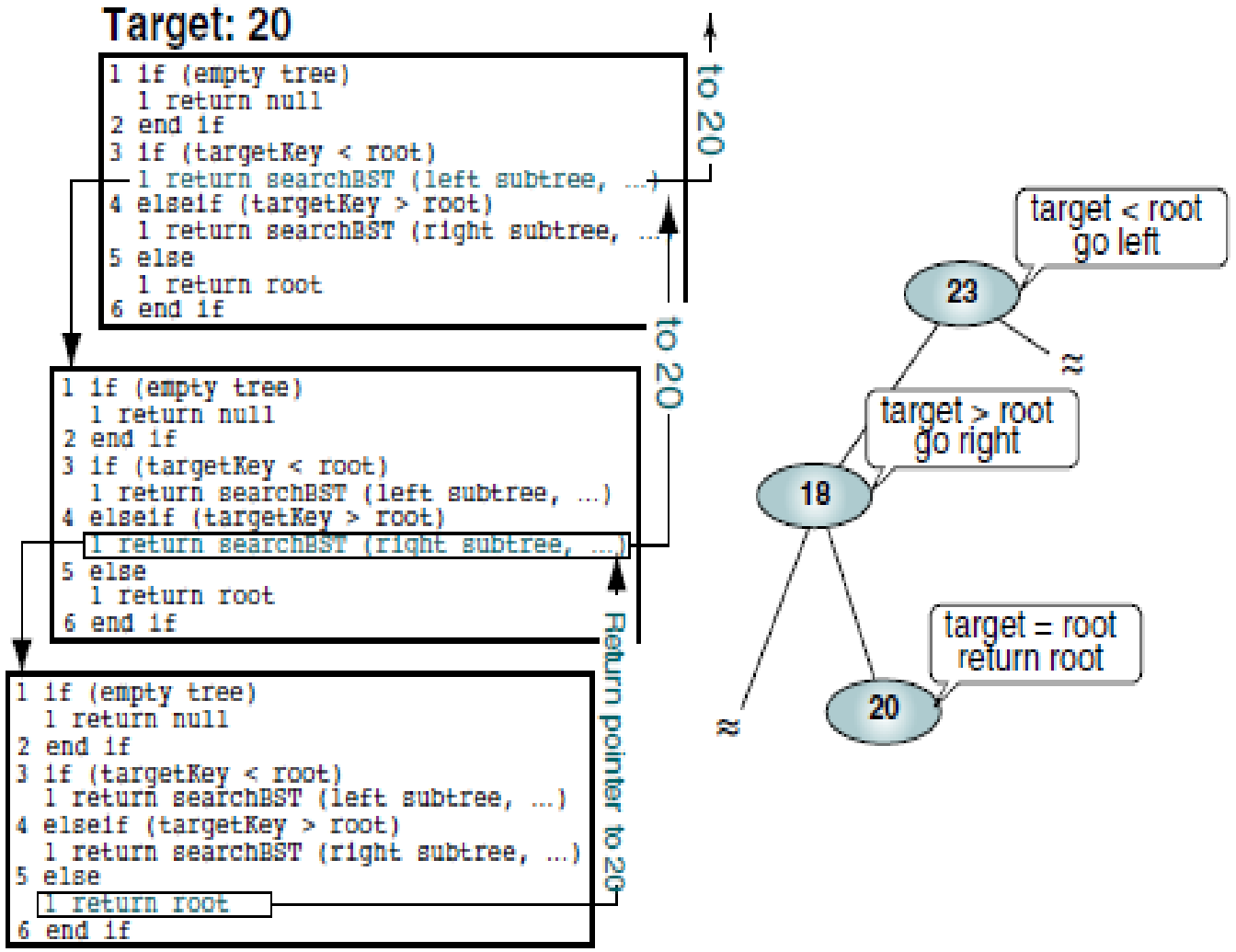


FIGURE 7-7 Searching a BST

Insertion

The **insert node** function adds data to a BST. To insert data all we need to do is follow the branches to an empty subtree and then insert the new node. In other words, all inserts take place at a leaf or at a leaflike node—a node that has only one null subtree.

All BST insertions take place at a leaf or a leaflike node.

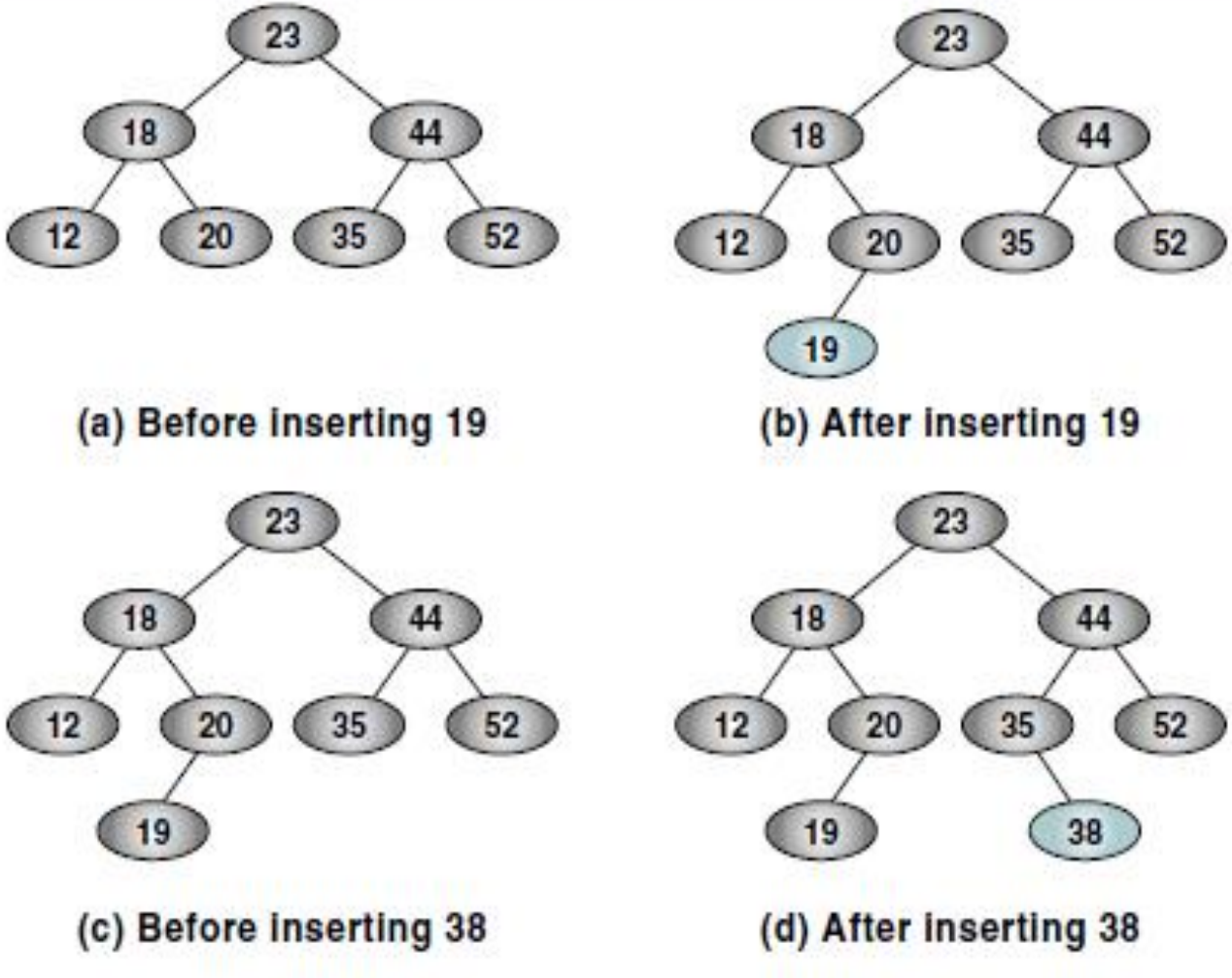


FIGURE 7-8 BST Insertion

ALGORITHM 7-4 Add Node to BST

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
   Pre    root is address of current node in a BST
          newNode is address of node containing data
   Post   newNode inserted into the tree
   Return address of potential new tree root
1  if (empty tree)
   1   set root to newNode
   2   return newNode
2  end if
   Locate null subtree for insertion
3  if (newNode < root)
   1   return addBST (left subtree, newNode)
4  else
   1   return addBST (right subtree, newNode)
5  end if
end addBST
```

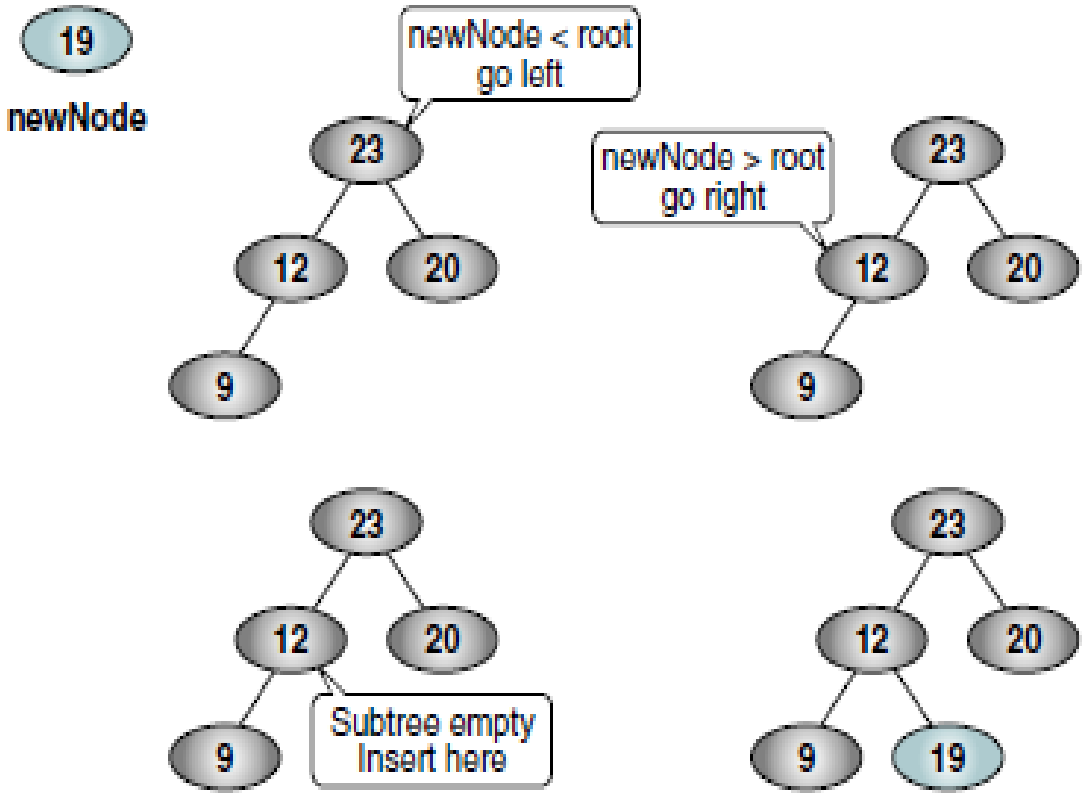


FIGURE 7-9 Trace of Recursive BST Insert

Deletion

To **delete a node** from a binary search tree, we must first locate it. There are four possible cases when we delete a node:

1. The node to be deleted has no children. In this case, all we need to do is delete the node.
2. The node to be deleted has only a right subtree. We delete the node and attach the right subtree to the deleted node's parent.
3. The node to be deleted has only a left subtree. We delete the node and attach the left subtree to the deleted node's parent.
4. The node to be deleted has two subtrees. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced trees. Rather than simply delete the node, therefore, we try to maintain the existing structure as much as possible by finding data to take the place of the deleted data. This can be done in one of two ways: (1) we can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data or (2) we can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data. Regardless of which logic we use, we will be moving data from a leaf or a leaflike node that can then be deleted.

ALGORITHM 7-5 Delete Node from BST

```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
   Pre    root is reference to node to be deleted
          dltKey is key of node to be deleted
   Post   node deleted
          if dltKey not found, root unchanged
   Return true if node deleted, false if not found
1  if (empty tree)
   1  return false
2  end if
3  if (dltKey < root)
   1  return deleteBST (left subtree, dltKey)
4  else if (dltKey > root)
   1  return deleteBST (right subtree, dltKey)
5  else
      Delete node found--test for leaf node
   1  If (no left subtree)
      1  make right subtree the root
      2  return true
   2  else if (no right subtree)
      1  make left subtree the root
      2  return true
   3  else
         Node to be deleted not a leaf. Find largest node on
         left subtree.
      1  save root in deleteNode
      2  set largest to largestBST (left subtree)
      3  move data in largest to deleteNode
      4  return deleteBST (left subtree of deleteNode,
                           key of largest
   4  end if
6  end if
end deleteBST
```

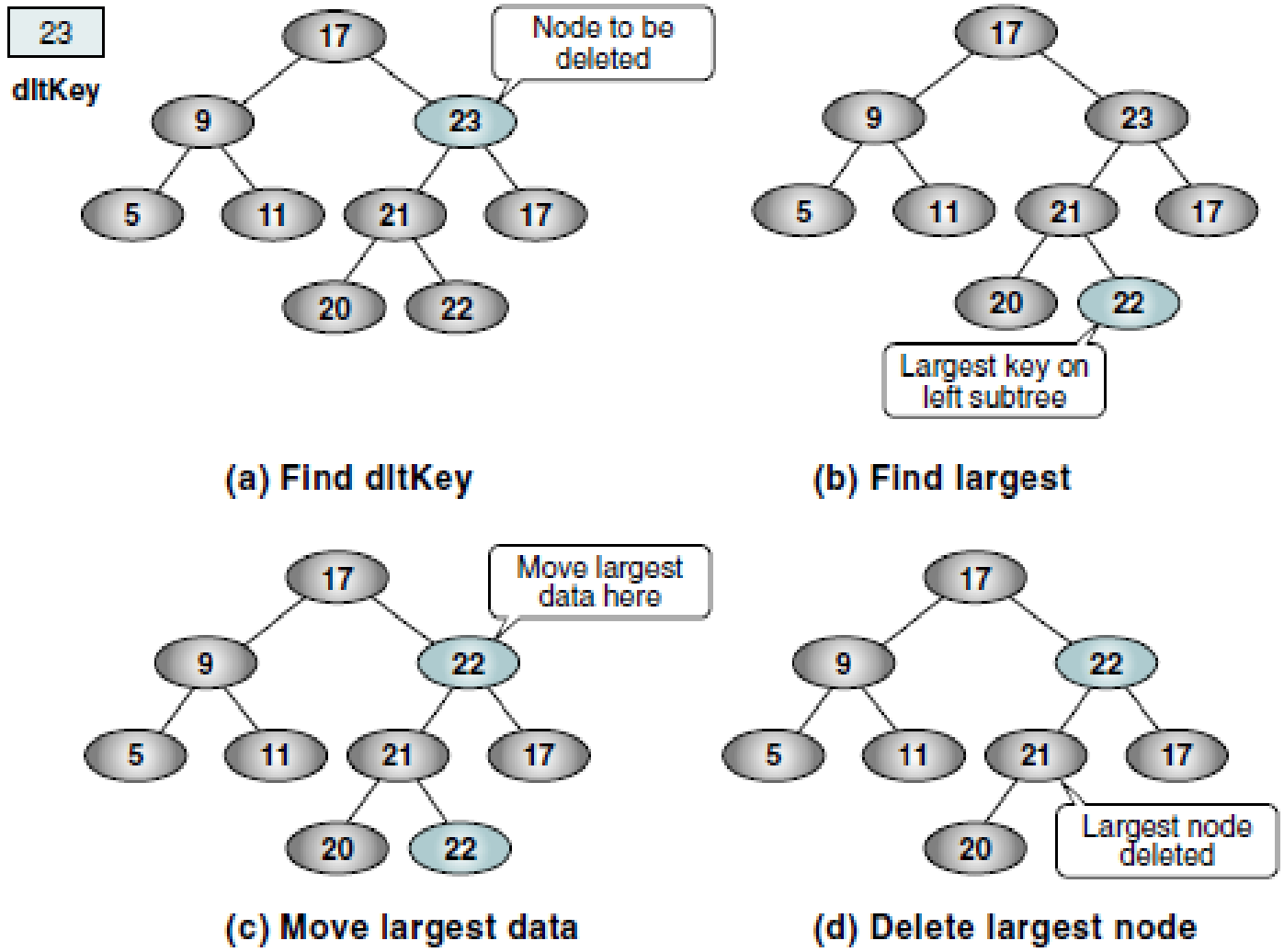


FIGURE 7-10 Delete BST Test Cases

Integer Application

The BST tree integer application reads integers from the keyboard and inserts them into the BST.

AVL Tree

In 1962, two Russian mathematicians, G. M. Adelson-Velskii and E. M. Landis, created the balanced binary tree structure that is named after them—the AVL tree. An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1. It is thus a balanced binary tree.

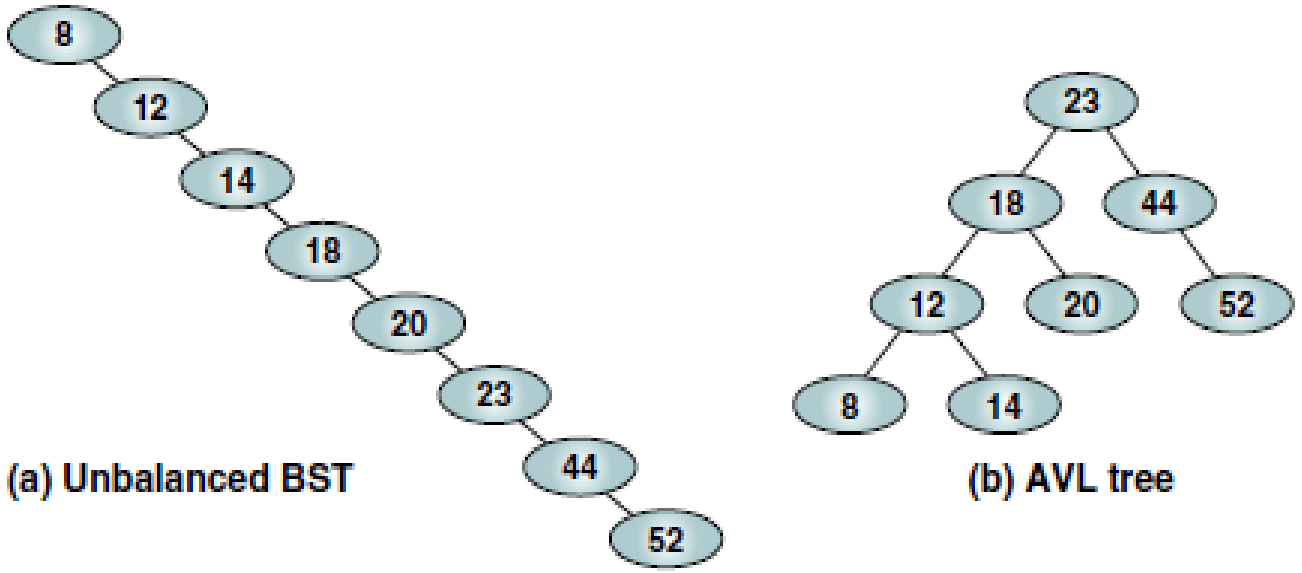


FIGURE 8-1 Two Binary Trees

Consider the tree in Figure 8-1(a), it takes two tests to locate 12. It takes three tests to locate 14. It takes eight tests to locate 52. In other words, the search effort for this particular binary search tree is $O(n)$.

Now consider the tree in Figure 8-1(b), locating nodes 8 and 14 requires four tests. Locating nodes 20 and 52 requires three tests. In other words, the maximum search effort for this tree is either three or four. Its search effort is $O(\log n)$.

We are now ready to define an AVL tree. An **AVL tree** is a binary tree that either is empty or consists of two AVL subtrees, T_L, and T_R, whose heights differ by no more than 1, as shown below.

$$|H_L - H_R| <= 1$$

In this formula, H_L is the height of the left subtree and H_R is the height of the right subtree (the bar symbols indicate absolute value). Because AVL trees are balanced by working with their height, they are also known as **height-balanced trees.**

An AVL tree is a height-balanced binary search tree.

AVL Tree Balance Factor

The balance factor for any node in an AVL tree must be +1, 0, or –1 LH for left high (+1) to indicate that the left subtree is higher than the right subtree, EH for even high (0) to indicate that the subtrees are the same height, and RH for right high (–1) to indicate that the left subtree is shorter than the right subtree.

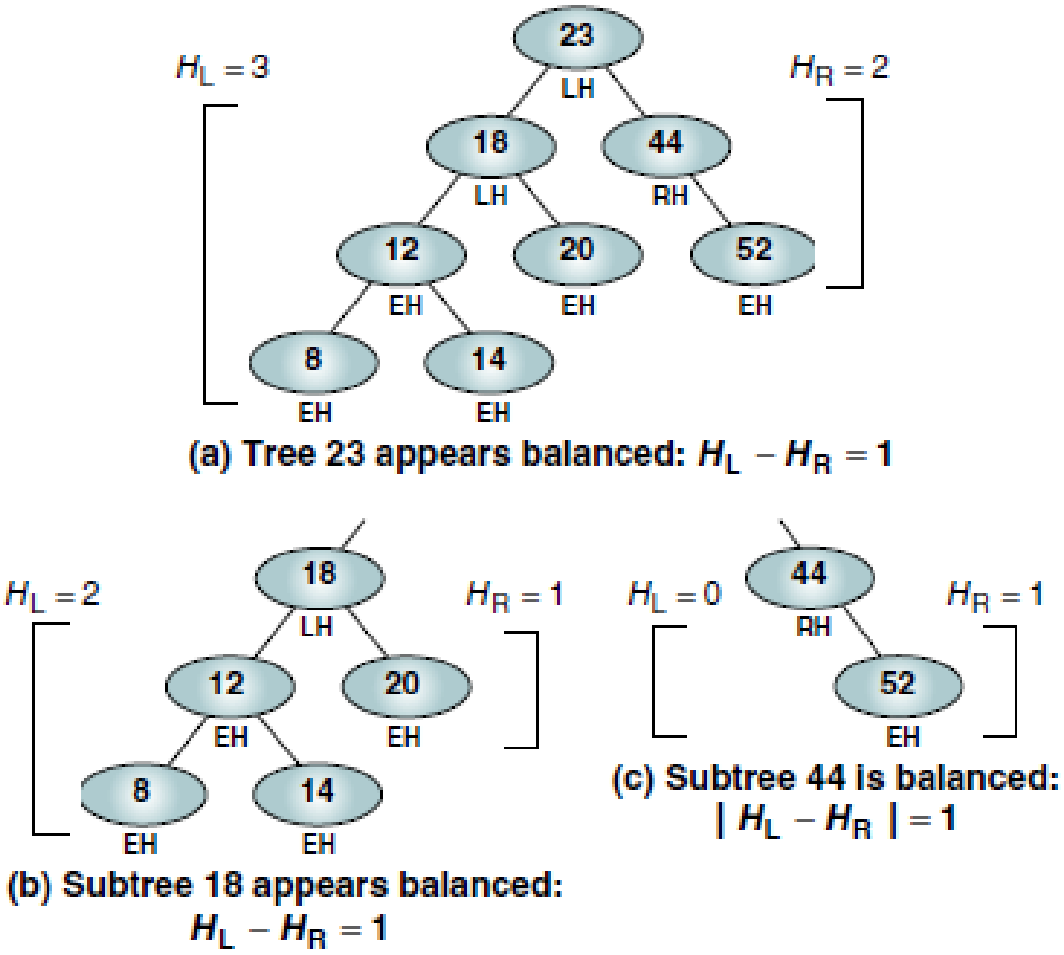


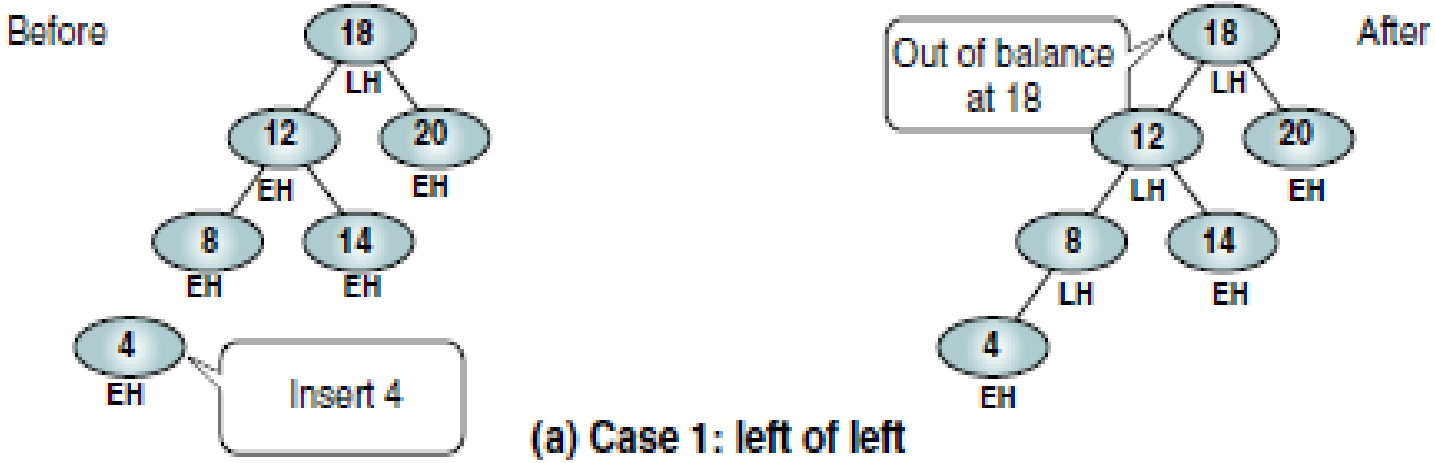
FIGURE 8-2 AVL Tree

Balancing Trees

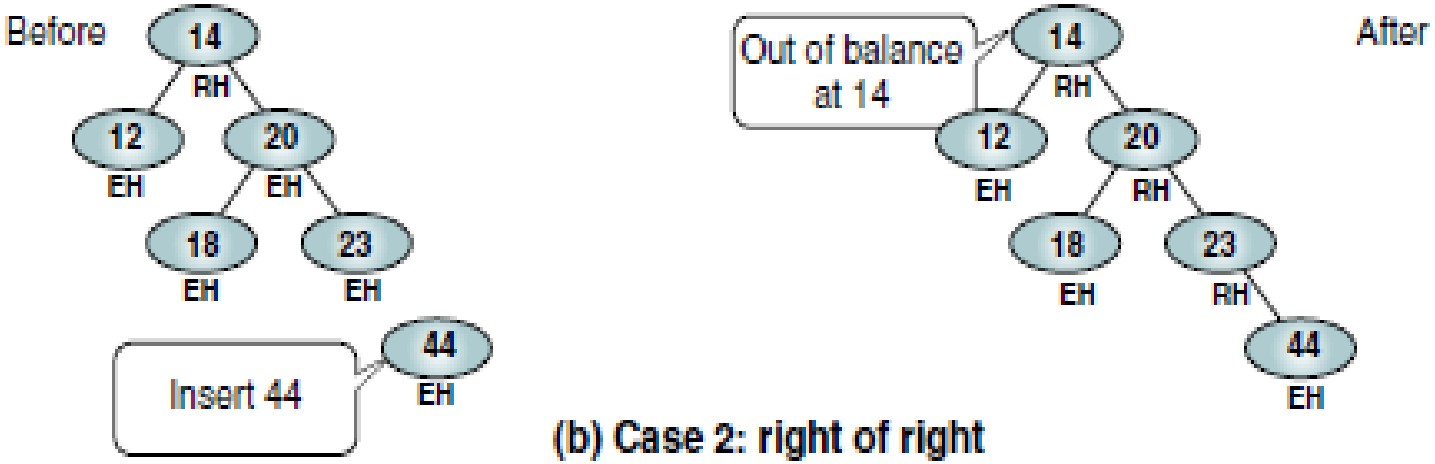
Whenever we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced. When we detect that a tree has become unbalanced, we must rebalance it. AVL trees are balanced by rotating nodes either to the left or to the right. In this section we discuss the basic balancing algorithms.

We consider four cases that require rebalancing. All unbalanced trees fall into one of these four cases:

1. Left of left—A subtree of a tree that is left high has also become left high.
2. Right of right—A subtree of a tree that is right high has also become right high.
3. Right of left—A subtree of a tree that is left high has become right high.
4. Left of right—A subtree of a tree that is right high has become left high.



(a) Case 1: left of left



(b) Case 2: right of right

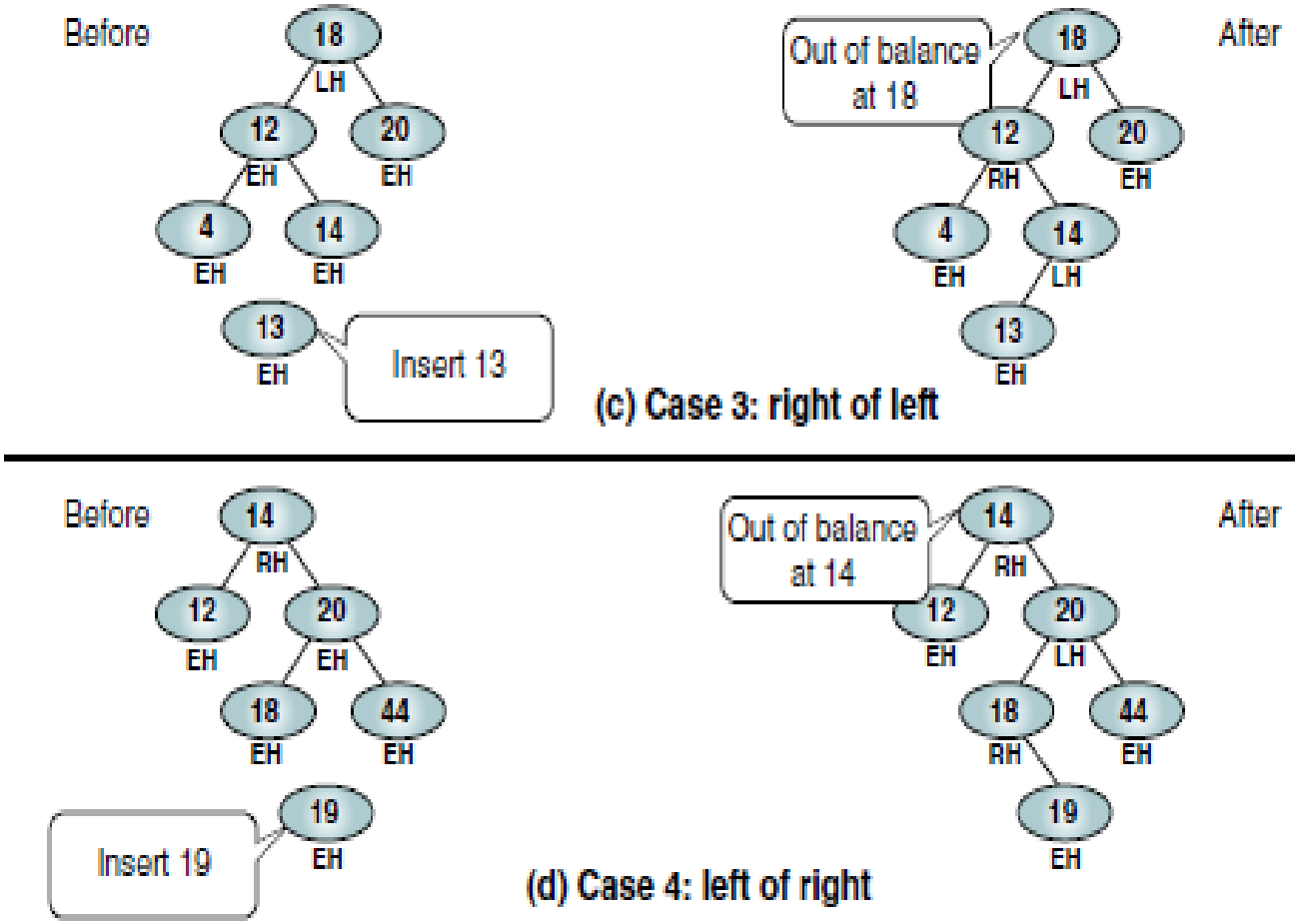


FIGURE 8-3 Out-of-balance AVL Trees

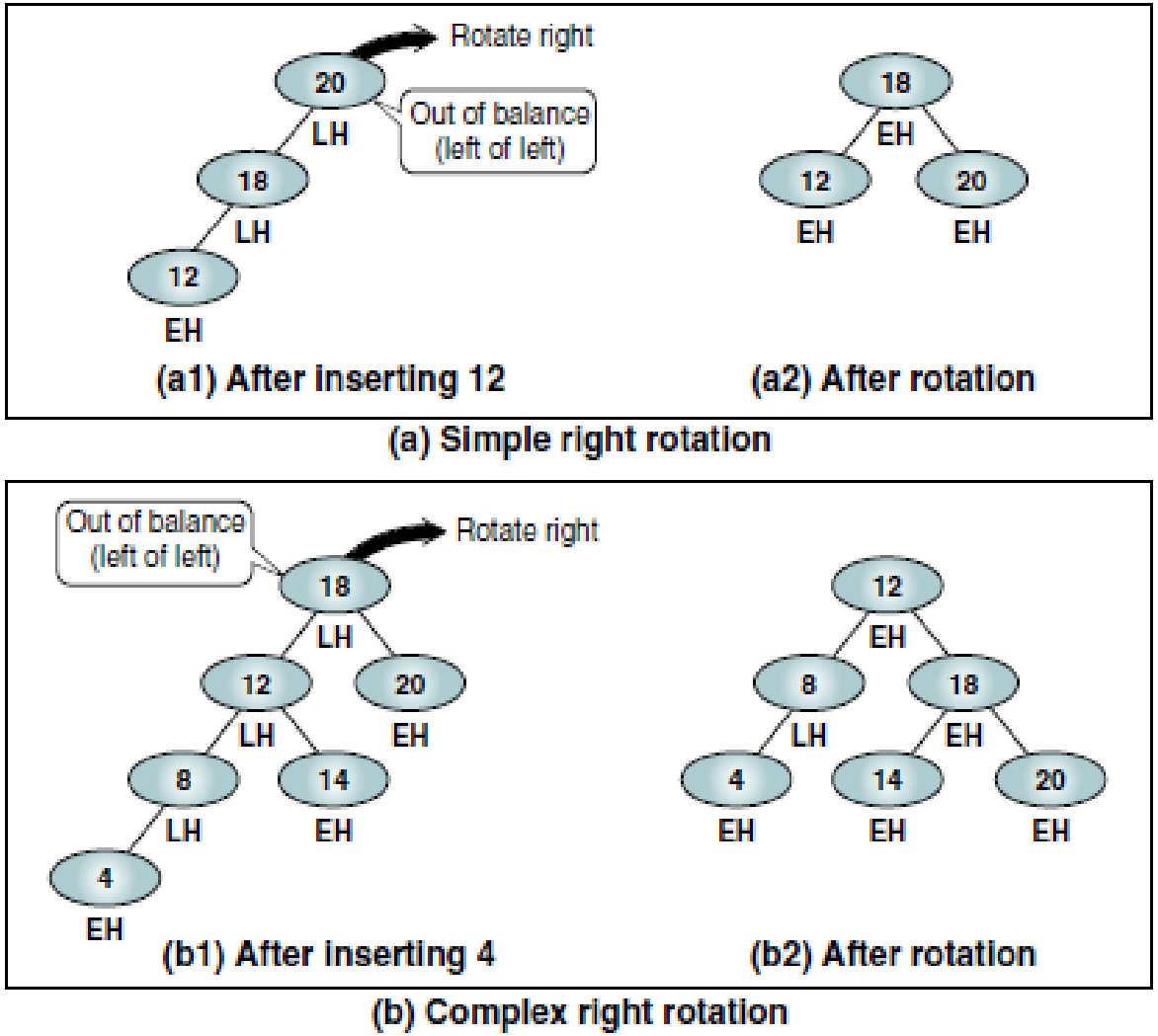
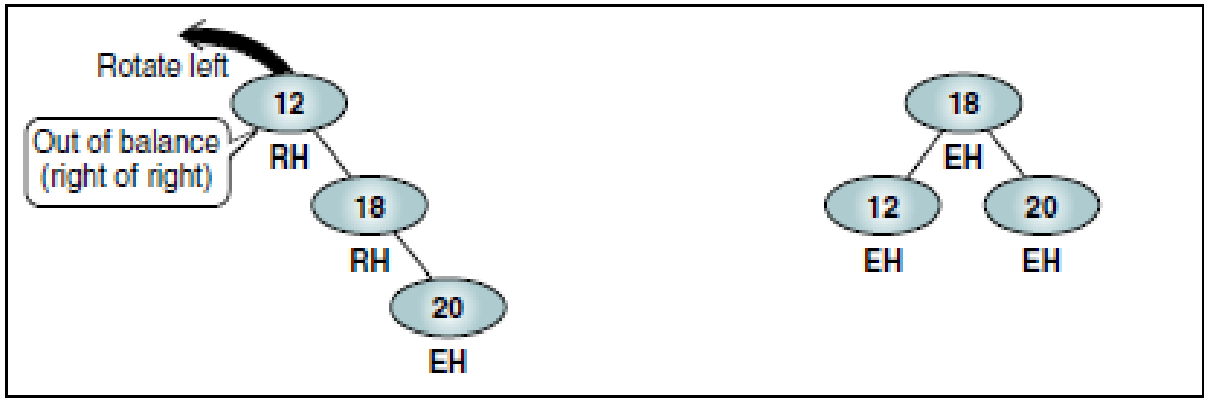
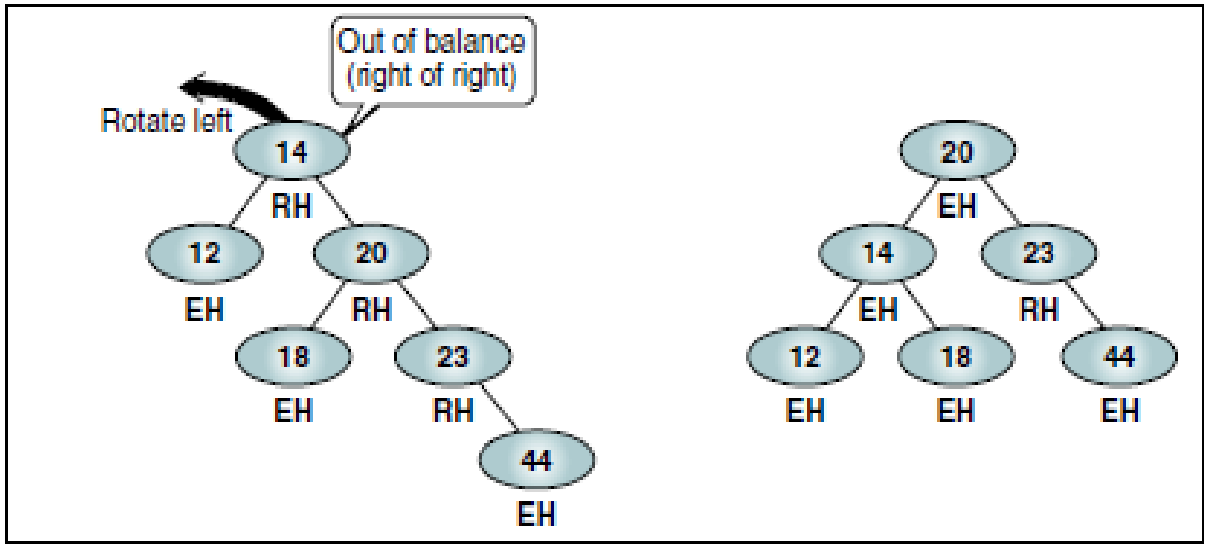


FIGURE 8-4 Left of Left—Single Rotation Right

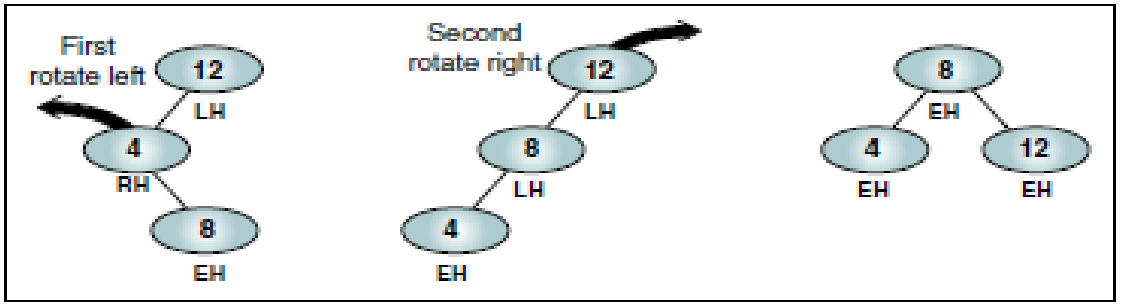


(a) Simple left rotation

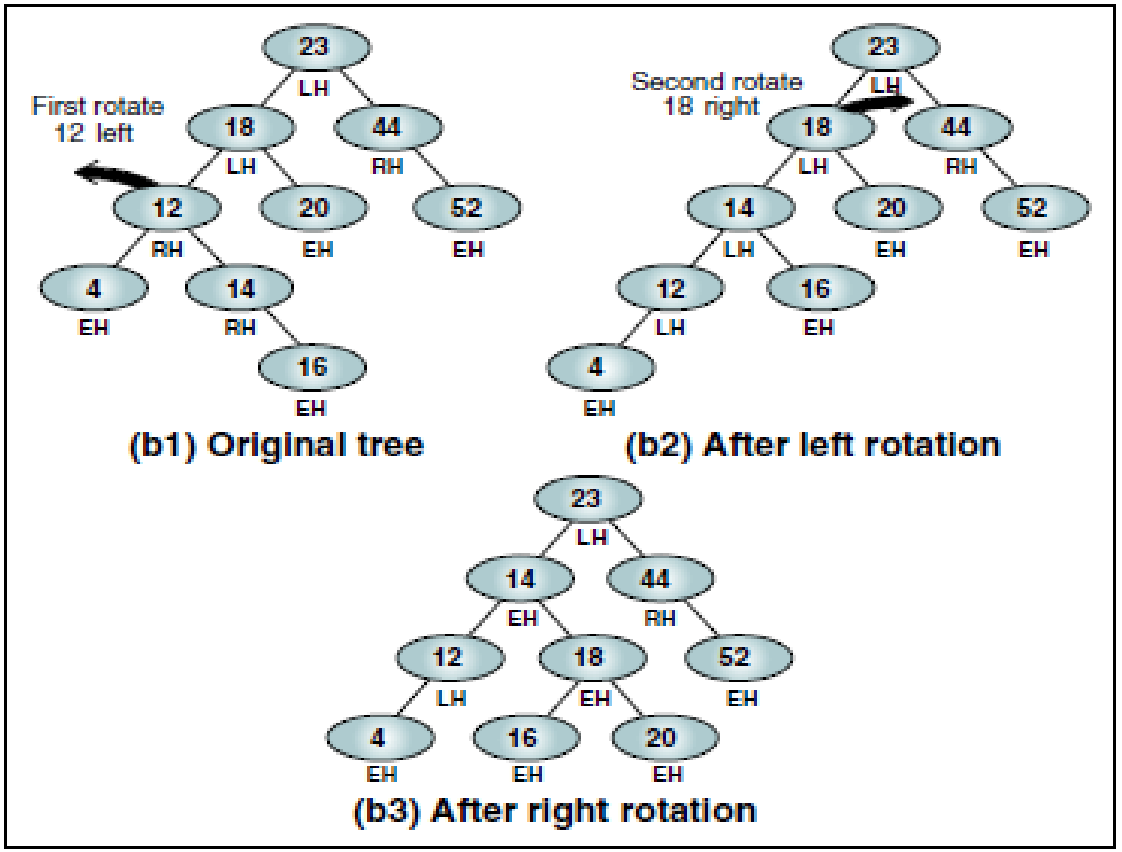


(b) Complex left rotation

FIGURE 8-5 Right of Right—Single Rotation Left



(a) Simple double rotation right



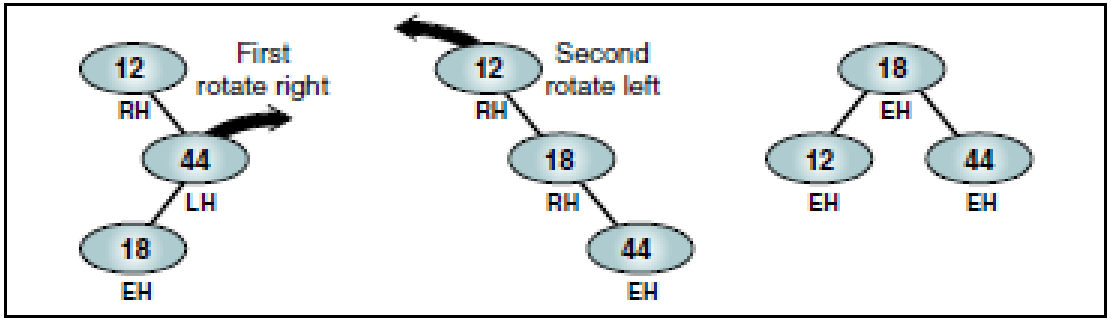
(b1) Original tree

(b2) After left rotation

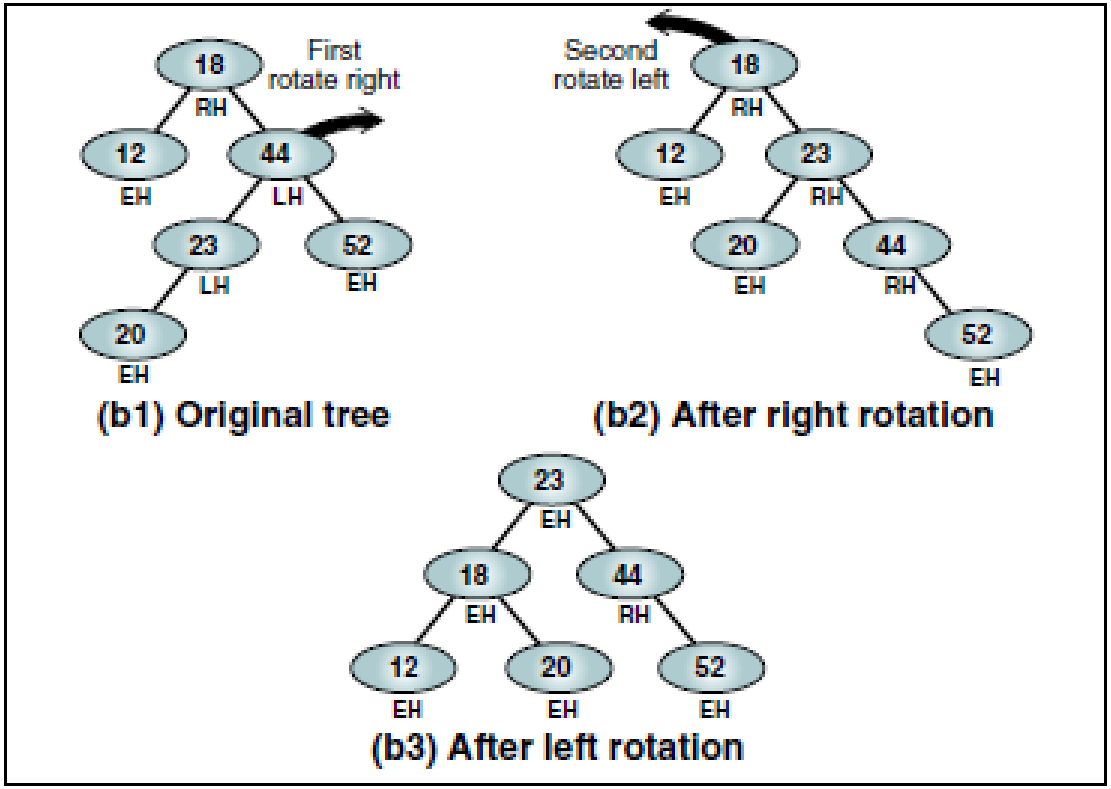
(b3) After right rotation

(b) Complex double rotation right

FIGURE 8-6 Right of Left—Double Rotation Right



(a) Simple double rotation right



(b) Complex double rotation right

FIGURE 8-7 Left of Right—Double Rotation Right