Trees

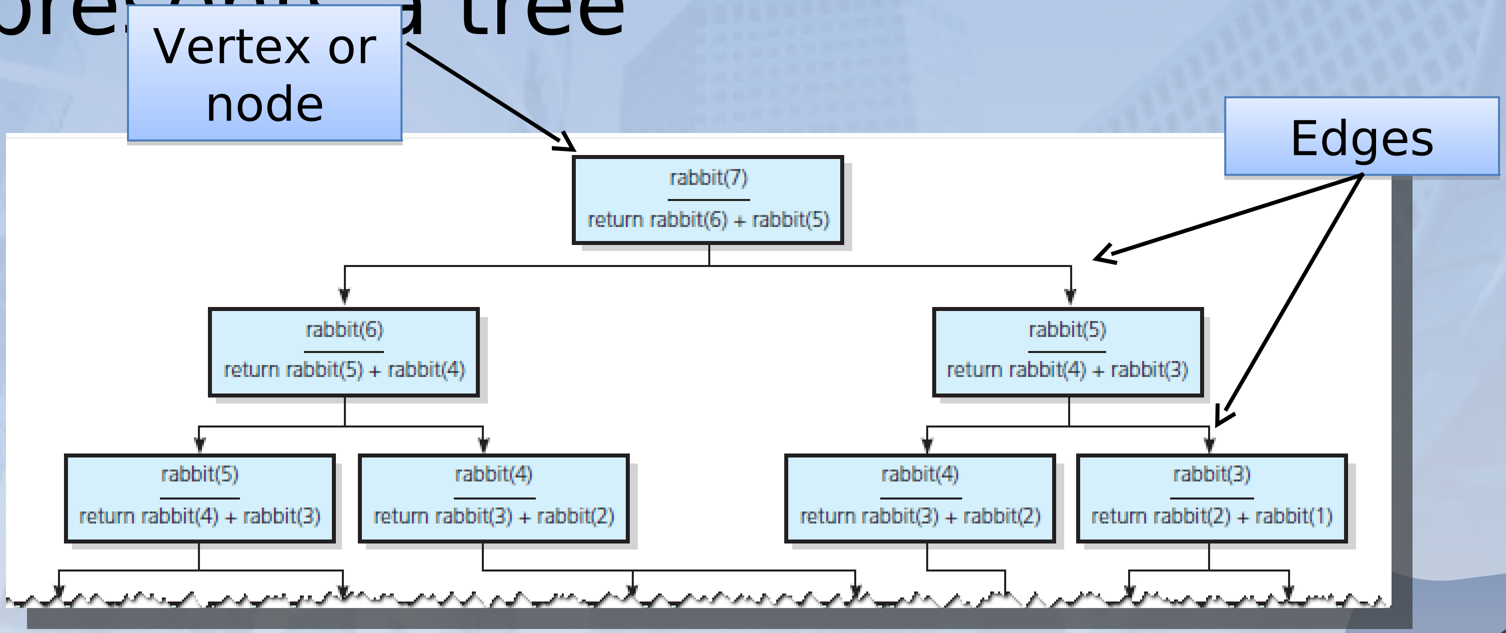
Chapter 15

Trees

- Lists, stacks, and queues are linear in their organization of data
  - Items are one after another.
- In this chapter, we organize data in a nonlinear, hierarchical form
  - Item can have more than one immediate successor

Terminology

- Use trees to represent relationships
- Recall Figure 2-19 … diagram represents a tree



Terminology

- Trees are hierarchical in nature
  - Means a parent-child relationship between nodes

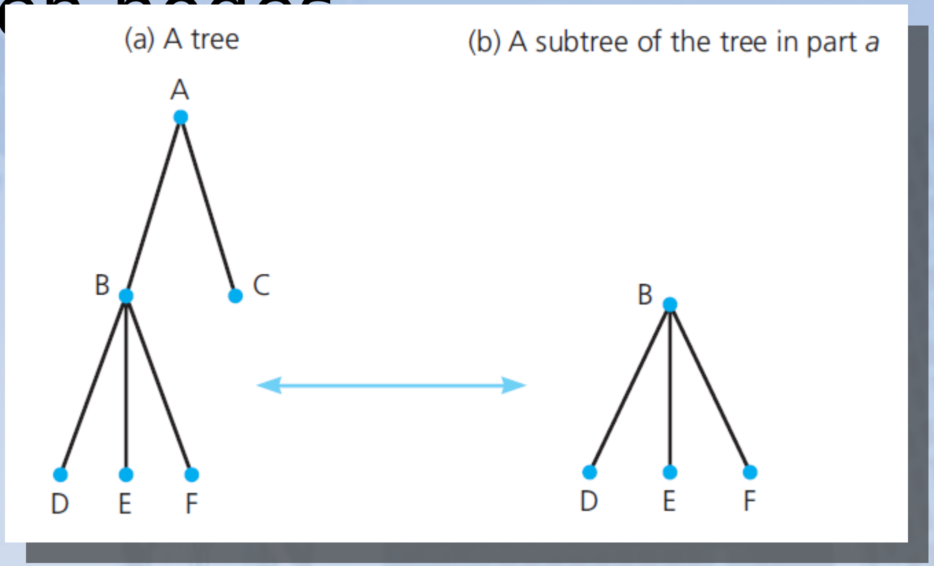


FIGURE 15-1 A tree and one of its subtrees

Terminology

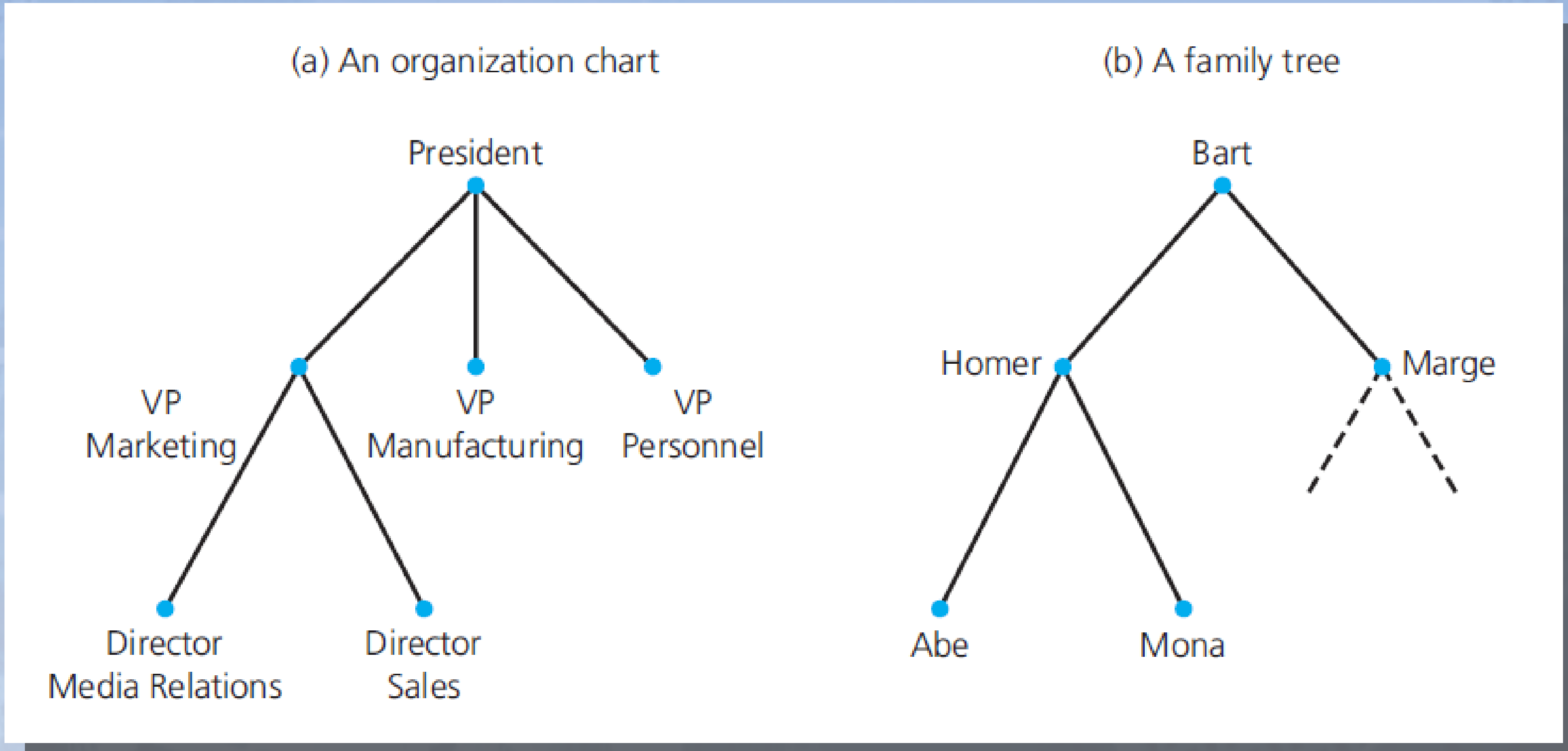


FIGURE 15-2

Kinds of Trees

- General tree
  - Set *T* of one or more nodes
  - *T* is partitioned into disjoint subsets
- Binary tree
  - Set of *T* nodes – either empty or partitioned into disjoint subsets
  - Single node *r*, the root
  - Two (possibly empty) sets – left and right subtrees

Kinds of Trees

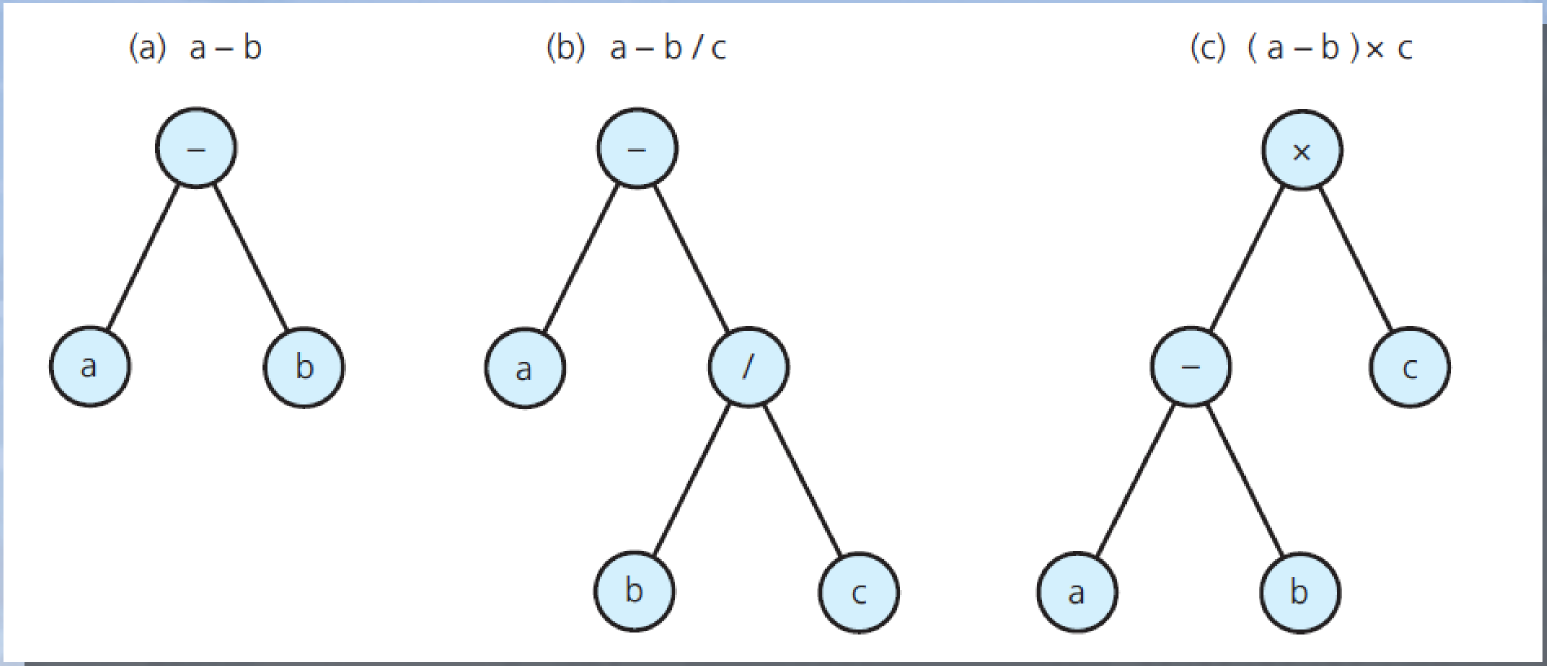


FIGURE 15-3 Binary trees that represent algebraic expressions

Kinds of Trees

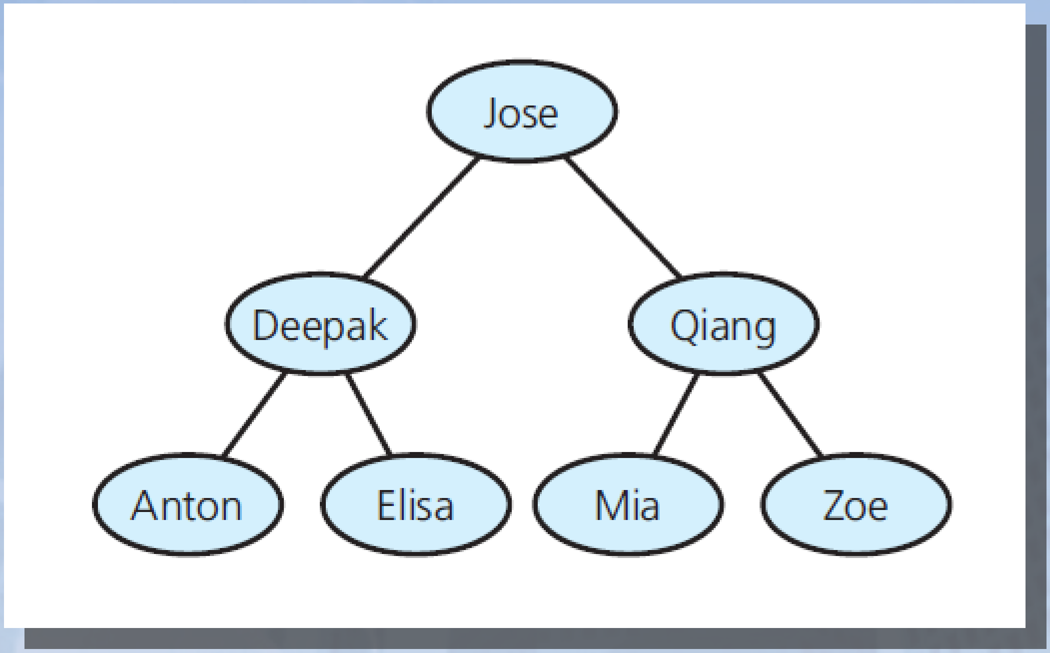


FIGURE 15-4 A binary search tree of names

The Height of Trees

- Level of a node, *n*
  - If *n* is root, level 1
  - If *n* not the root, level is 1 greater than level of its parent
- Height of a tree
  - Number of nodes on longest path from root to a leaf
  - *T* empty, height 0
  - *T* not empty, height equal to max level of nodes

The Height of Trees

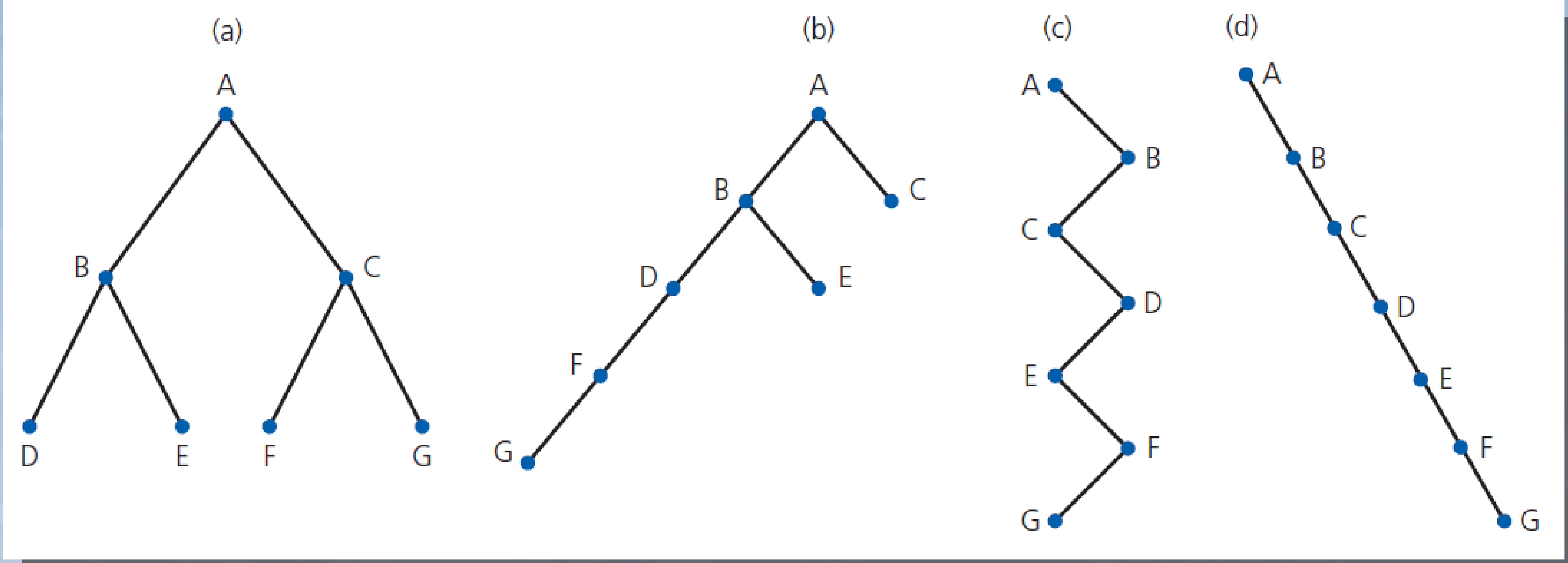


FIGURE 15-5 Binary trees with the same nodes but different heights

Full, Complete, and Balanced Binary Trees

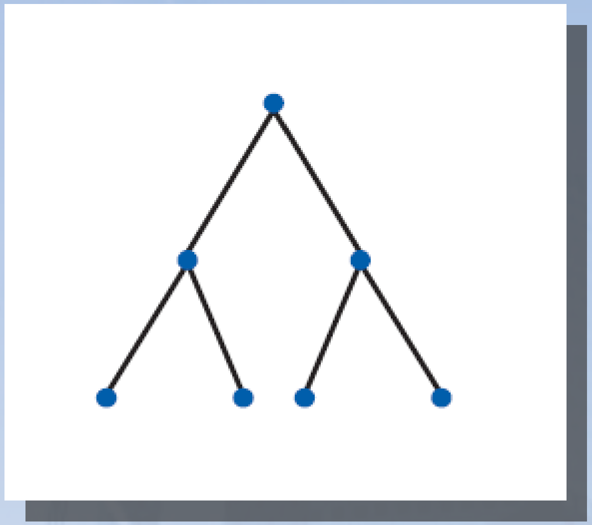


FIGURE 15-6 A full binary tree of height 3

Full, Complete, and Balanced Binary Trees

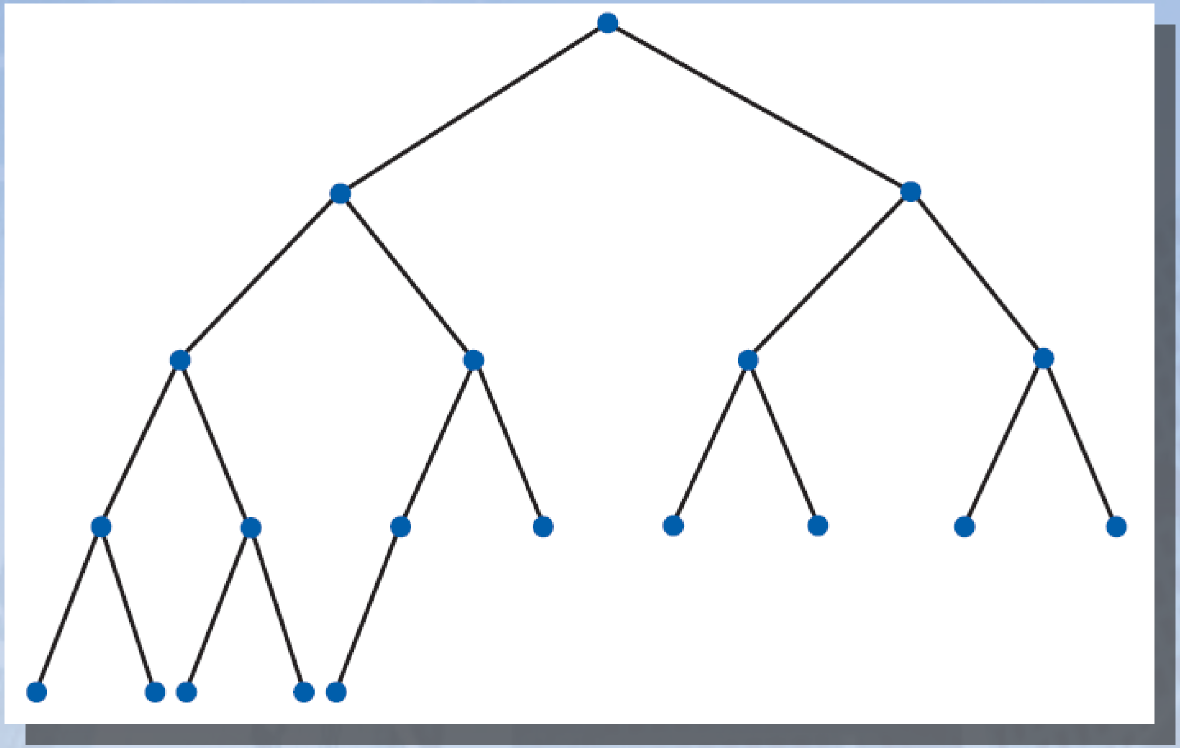


FIGURE 15-7 A complete binary tree

The Maximum and Minimum Heights of a Binary Tree

- Binary tree with *n* nodes
  - Max height is *n*
- To minimize height of binary tree of *n* nodes
  - Fill each level of tree as completely as possible
  - A complete tree meets this requirement

The Maximum and Minimum Heights of a Binary Tree

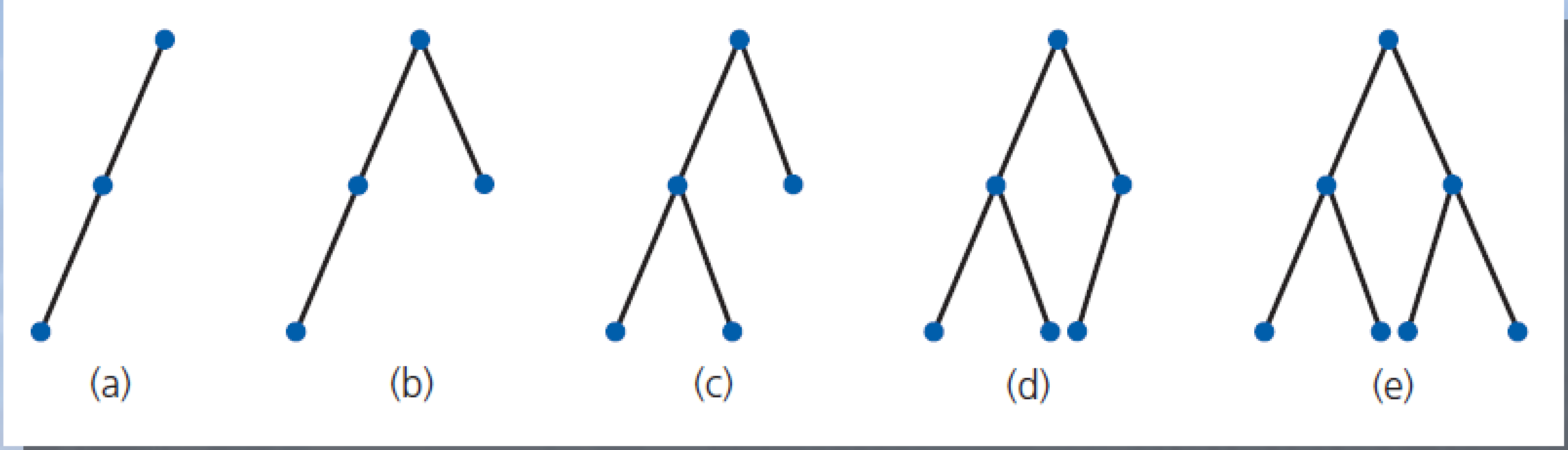
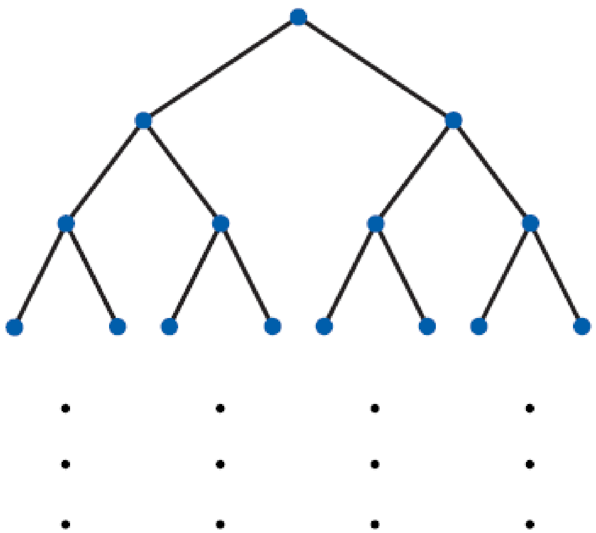


FIGURE 15-8 Binary trees of height 3

The Maximum and Minimum Heights of a Binary Tree



Level	Number of nodes at this level	Total number of nodes at this level and all 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}	$2^h - 1$

FIGURE 15-9 Counting the nodes in a full binary tree of height *h*

The Maximum and Minimum Heights of a Binary Tree

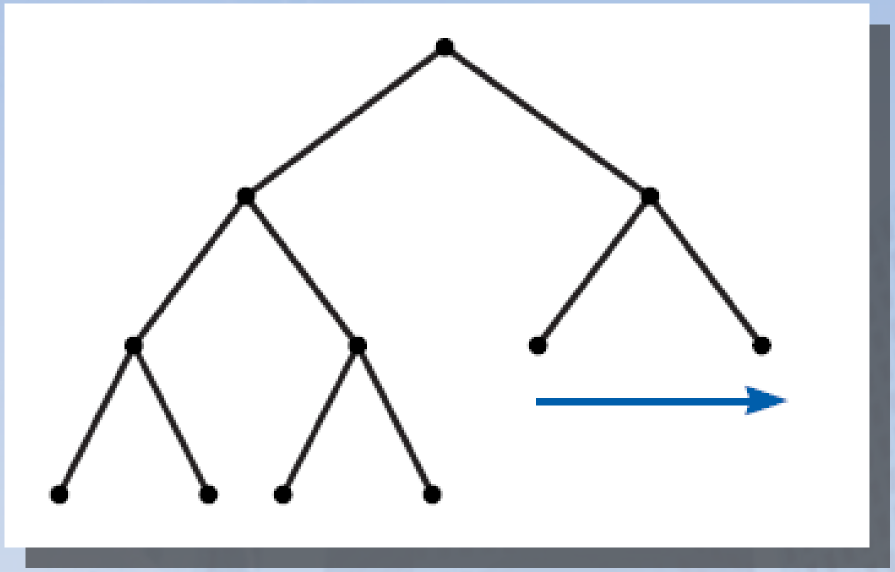


FIGURE 15-10 Filling in the last level of a tree

The ADT Binary Tree

- Operations of ADT binary tree
  - Add, remove
  - Set, retrieve data
  - Test for empty
  - Traversal operations that visit every node
- Traversal can visit nodes in several different orders

Traversals of a Binary Tree

- Pseudocode for general form of a recursive traversal algorithm

```
if (T is not empty)
{
    Display the data in T's root
    Traverse T's left subtree
    Traverse T's right subtree
}
```

Traversals of a Binary Tree

- Options for when to visit the root
  - Preorder: before it traverses both subtrees
  - Inorder: after it traverses left subtree, before it traverses right subtree
  - Postorder: after it traverses both subtrees
- Note traversal is O(n)

Traversals of a Binary Tree

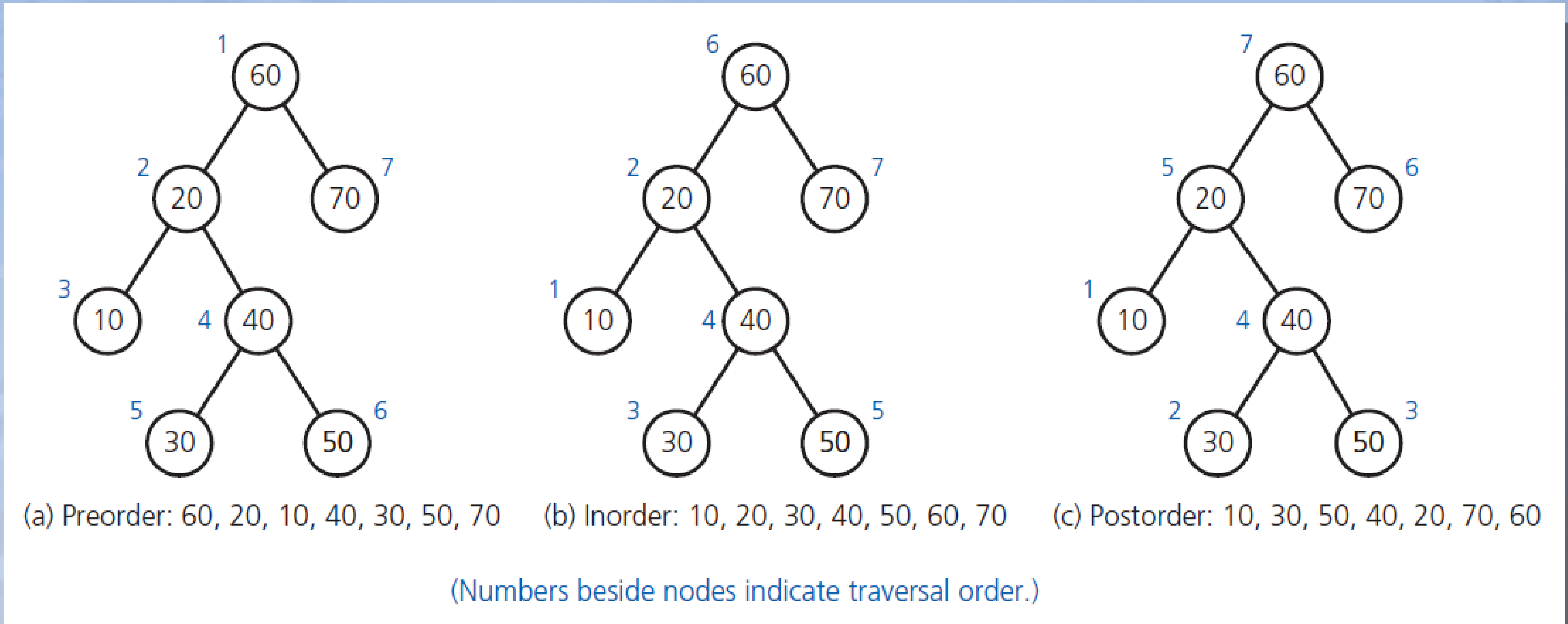


FIGURE 15-11 Three traversals of a binary tree

Traversals of a Binary Tree

```
// Traverses the given binary tree in preorder.
// Assumes that "visit a node" means to process the node's data item.
preorder(binTree: BinaryTree): void
{
   if (binTree is not empty)
   {
      Visit the root of binTree
      preorder(Left subtree of binTree's root)
      preorder(Right subtree of binTree's root)
   }
}
```

Preorder traversal algorithm

Traversals of a Binary Tree

```
// Traverses the given binary tree in inorder.
// Assumes that "visit a node" means to process the node's data item.
inorder(binTree: BinaryTree): void
{
   if (binTree is not empty)
   {
      inorder(Left subtree of binTree's root)
      Visit the root of binTree
      inorder(Right subtree of binTree's root)
   }

}
```

Inorder traversal algorithm

Traversals of a Binary Tree

```
// Traverses the given binary tree in postorder.
// Assumes that "visit a node" means to process the node's data item.
postorder(binTree: BinaryTree): void

{   if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

Postorder traversal algorithm

Binary Tree Operations

```
BinaryTree
-----------------------------------------------------------

-----------------------------------------------------------
+isEmpty(): boolean
+getHeight(): integer
+getNumberOfNodes(): integer
+getRootData(): ItemType
+setRootData(newData: ItemType): void
+add(newData: ItemType): boolean
+remove(target: ItemType): boolean
+clear(): void
+getEntry(target: ItemType): ItemType
+contains(target: ItemType): boolean
+preorderTraverse(visit(item: ItemType): void): void
+inorderTraverse(visit(item: ItemType): void): void
+postorderTraverse(visit(item: ItemType): void): void
```

FIGURE 15-12 UML diagram for the class BinaryTree

Interface Template for the ADT Binary Tree

```
/** Interface for the ADT binary tree.
 @file BinaryTreeInterface.h */

#ifndef BINARY_TREE_INTERFACE_
#define BINARY_TREE_INTERFACE_
#include "NotFoundException.h"

template<class ItemType>
class BinaryTreeInterface
{
public:
   /** Tests whether this binary tree is empty.
    @return  True if the binary tree is empty, or false if not. */
   virtual bool isEmpty() const = 0;

   /** Gets the height of this binary tree.
    @return  The height of the binary tree. */
   virtual int getHeight() const = 0;

   /** Gets the number of nodes in this binary tree.
    @return  The number of nodes in the binary tree. */
   virtual int getNumberOfNodes() const = 0;
```

LISTING 15-1 An interface template for the ADT binary tree

Interface Template for the ADT Binary Tree

```
   /** Gets the data that is in the root of this binary tree.
    @pre  The binary tree is not empty.
    @post  The root's data has been returned, and the binary tree is unchanged.
    @return  The data in the root of the binary tree. */
   virtual ItemType getRootData() const = 0;

   /** Replaces the data in the root of this binary tree with the given data,
       if the tree is not empty. However, if the tree is empty, inserts a new
       root node containing the given data into the tree.
    @pre  None.
    @post  The data in the root of the binary tree is as given.
    @param newData  The data for the root. */
   virtual void setRootData(const ItemType& newData) = 0;

   /** Adds the given data to this binary tree.
    @param newData  The data to add to the binary tree.
    @post  The binary tree contains the new data.
    @return  True if the addition is successful, or false if not. */
   virtual bool add(const ItemType& newData) = 0;
```

LISTING 15-1 An interface template for the ADT binary tree

Interface Template for the ADT Binary Tree

```
/** Removes the specified data from this binary tree.
 @param target  The data to remove from the binary tree.
 @return  True if the removal is successful, or false if not. */
virtual bool remove(const ItemType& target) = 0;

/** Removes all data from this binary tree. */
virtual void clear() = 0;

/** Retrieves the specified data from this binary tree.
 @post  The desired data has been returned, and the binary tree
    is unchanged. If no such data was found, an exception is thrown.
 @param target  The data to locate.
 @return  The data in the binary tree that matches the given data.*/
virtual ItemType getEntry(const ItemType& target) const = 0;
```

LISTING 15-1 An interface template for the ADT binary tree

Interface Template for the ADT Binary Tree

```
   /** Tests whether the specified data occurs in this binary tree.
    @post  The binary tree is unchanged.
    @param target  The data to find.
    @return  True if data matching the target occurs in the tree, or false if not. */
   virtual bool contains(const ItemType& target) const = 0;

   /** Traverses this binary tree in preorder (inorder, postorder) and
       calls the function visit once for each node.
   @param visit  A client-defined function that performs an operation on
      either each visited node or its data. */
   virtual void preorderTraverse(void visit(ItemType&)) const = 0;
   virtual void inorderTraverse(void visit(ItemType&)) const = 0;
   virtual void postorderTraverse(void visit(ItemType&)) const = 0;

   /** Destroys this tree and frees its assigned memory. */
   virtual ~BinaryTreeInterface() {  }
}; // end BinaryTreeInterface
#endif
```

LISTING 15-1 An interface template for the ADT binary tree

The ADT Binary Search Tree

- Recursive definition of a binary search tree
  - *n*'s value is greater than all values in its left subtree T_L.
  - *n*'s value is less than all values in its right subtree T_R.
  - Both T_L and T_R are binary search trees.

The ADT Binary Search Tree

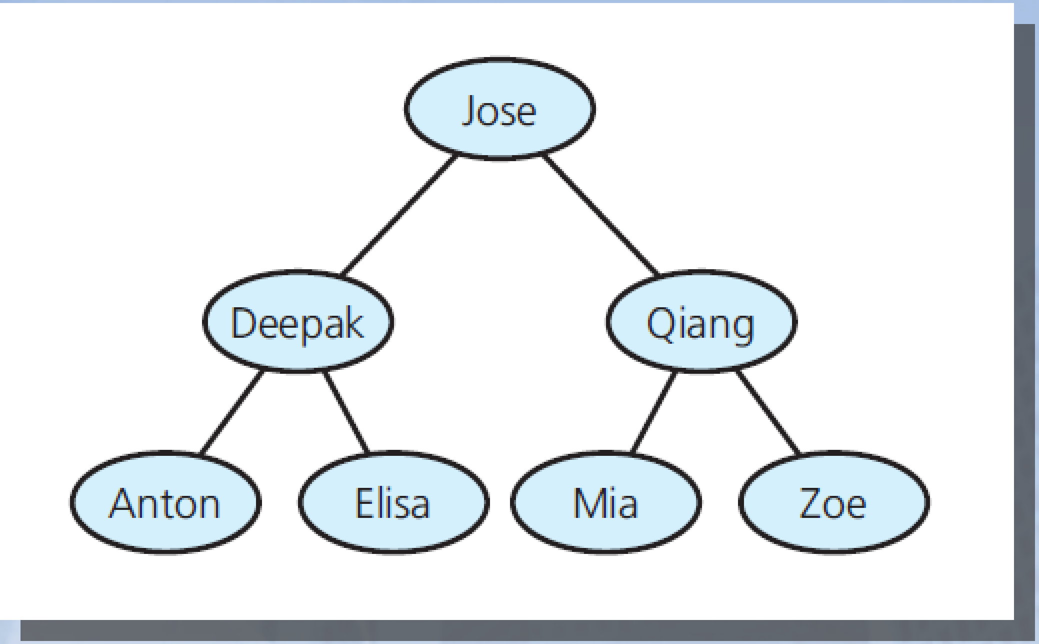


FIGURE 15-13 A binary search tree of names

Binary Search Tree Operations

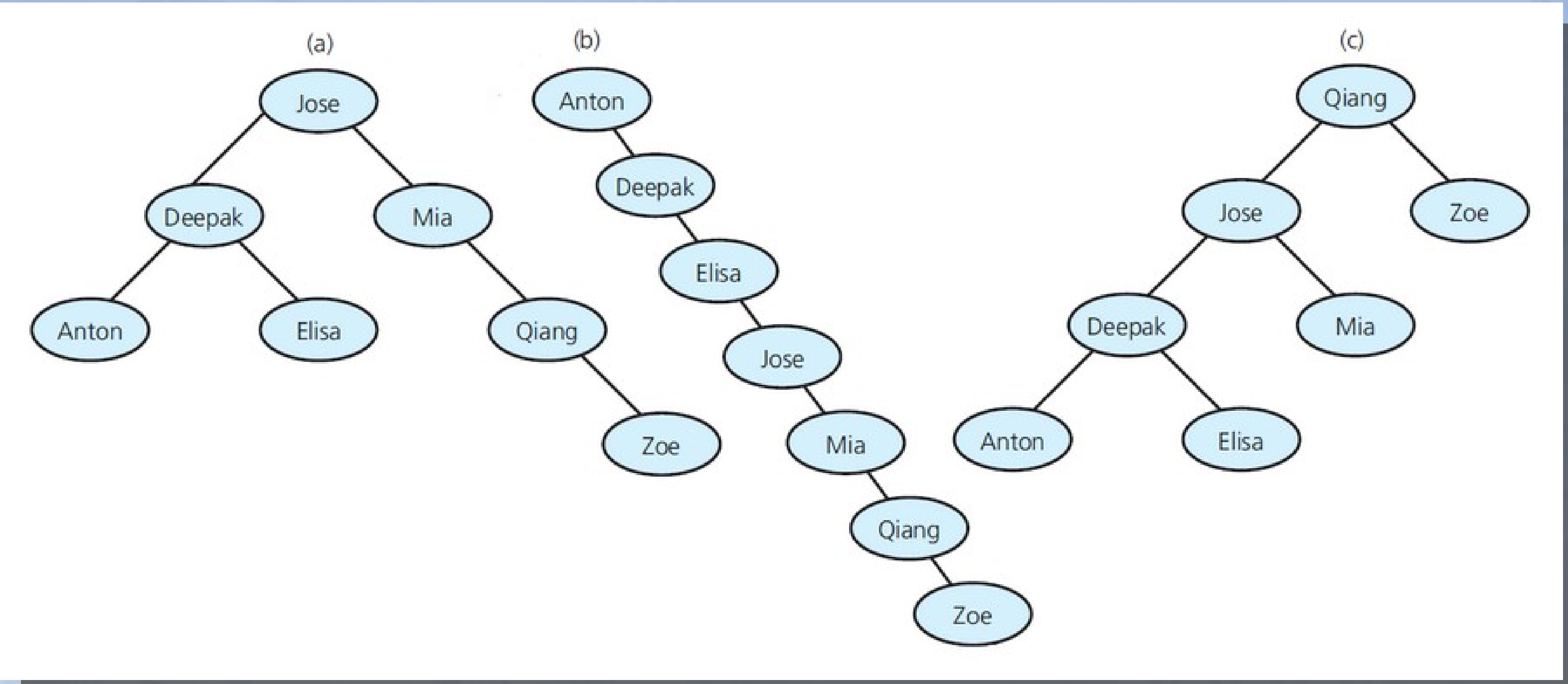


FIGURE 15-14 Binary search trees with the same data as in Figure 15-13

Binary Search Tree Operations

- Test whether a binary search tree is empty.
- Get the height of a binary search tree.
- Get the number of nodes in a binary search tree.
- Get the data in a binary search tree's root.
- Add the given data item to a binary search tree.
- Remove the specified data item from a binary search tree.
- Remove all data items from a binary search tree.
- Retrieve the specified data item in a binary search tree.
- Test whether a binary search tree contains specific data.
- Traverse the nodes in a binary search tree in preorder, inorder, or postorder.

Operations that define the ADT binary search tree

Searching a Binary Search Tree

```
// Searches the binary search tree for a given target value.
search(bstTree: BinarySearchTree, target: ItemType)
{
   if (bstTree is empty)
      The desired item is not found
   else if (target == data item in the root of bstTree)
      The desired item is found
   else if (target < data item in the root of bstTree)
      search(Left subtree of bstTree, target)
   else
      search(Right subtree of bstTree, target)
}
```

Search algorithm for a binary search tree

Searching a Binary Search Tree

Anton	Deepak	Elisa	Jose	Mia	Qiang	Zoe
0	1	2	3	4	5	6

FIGURE 15-15 An array of names in sorted order

Creating a Binary Search Tree

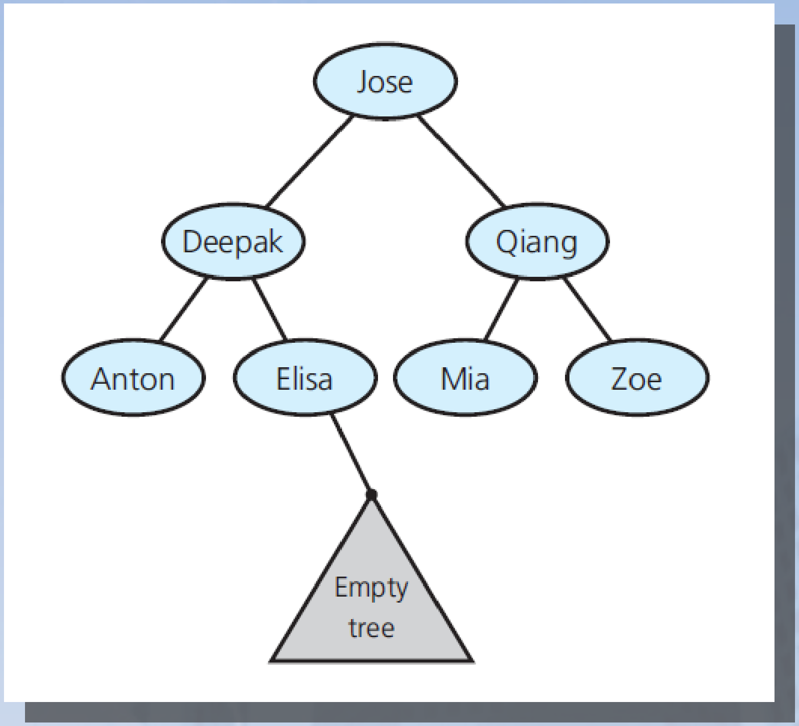


FIGURE 15-16 Empty subtree where the search algorithm terminates when looking for Finn

Traversals of a Binary Search Tree

```
// Traverses the given binary tree in inorder.
// Assumes that "visit a node" means to process the node's data item.
inorder(binTree: BinaryTree): void
{
   if (binTree is not empty)
   {
      inorder(Left subtree of binTree's root)
      Visit the root of binTree
      inorder(Right subtree of binTree's root)
   }

}
```

Inorder traversal of a binary search tree visits tree's nodes in sorted search-key order

Efficiency of Binary Search Tree Operations

- Max number of comparisons for retrieval, addition, or removal
  - The height of the tree
- Adding entries in sorted order
  - Produces maximum-height binary search tree
- Adding entries in random order
  - Produces near-minimum-height binary search tree

Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	O(log *n*)	O(*n*)
Addition	O(log *n*)	O(*n*)
Removal	O(log *n*)	O(*n*)
Traversal	O(*n*)	O(*n*)

FIGURE 15-17 The Big O for the retrieval, addition, removal, and traversal operations of the ADT binary search tree

End

Chapter 15