A Binary Search Tree Implementation

Chapter 25

Contents

- Getting Started
  - An Interface for the Binary Search Tree
  - Duplicate Entries
  - Beginning the Class Definition
- Searching and Retrieving
- Traversing
- Adding an Entry
  - A Recursive Implementation
  - An Iterative Implementation

Contents

- Removing an Entry
  - Removing an Entry Whose Node Is a Leaf
  - Removing an Entry Whose Node Has One Child
  - Removing an Entry Whose Node Has Two Children
  - Removing an Entry in the Root
  - A Recursive Implementation
  - An Iterative Implementation

Contents

- The Efficiency of Operations
  - The Importance of Balance
  - The Order in Which Nodes Are Added
- An Implementation of the ADT Dictionary

Objectives

- Decide whether a binary tree is a binary search tree
- Locate a given entry in a binary search tree using fewest comparisons
- Traverse entries in a binary search tree in sorted order
- Add a new entry to a binary search tree

Objectives

- Remove entry from a binary search tree
- Describe efficiency of operations on a binary search tree
- Use a binary search tree to implement ADT dictionary

Getting Started

- Characteristics of a binary search tree
  - A binary tree
  - Nodes contain **`Comparable`** objects
- For each node
  - Data in a node is greater than data in node's left subtree
  - Data in a node is less than data in node's right subtree

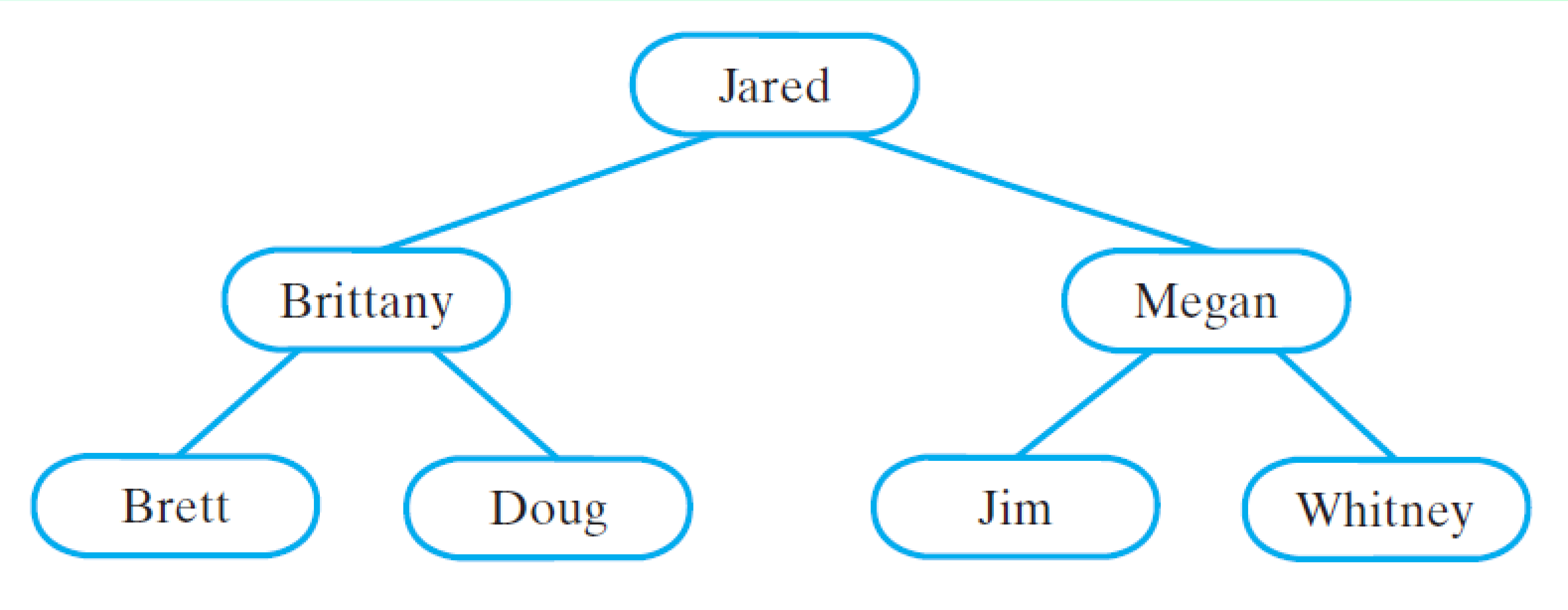


Figure 25-1 A binary search tree of names

Interface for the Binary Search Tree

- Additional operations needed beyond basic tree operations
- Database operations
  - Search
  - Retrieve
  - Remove
  - Traverse
- Note interface, Listing 25-1

Note: Code listing files must be in same folder as PowerPoint files for links to work

Understanding Specifications

- Interface specifications allow use of binary search tree for an ADT dictionary
- Methods use return values instead of exceptions
  - Indicates success or failure of operation

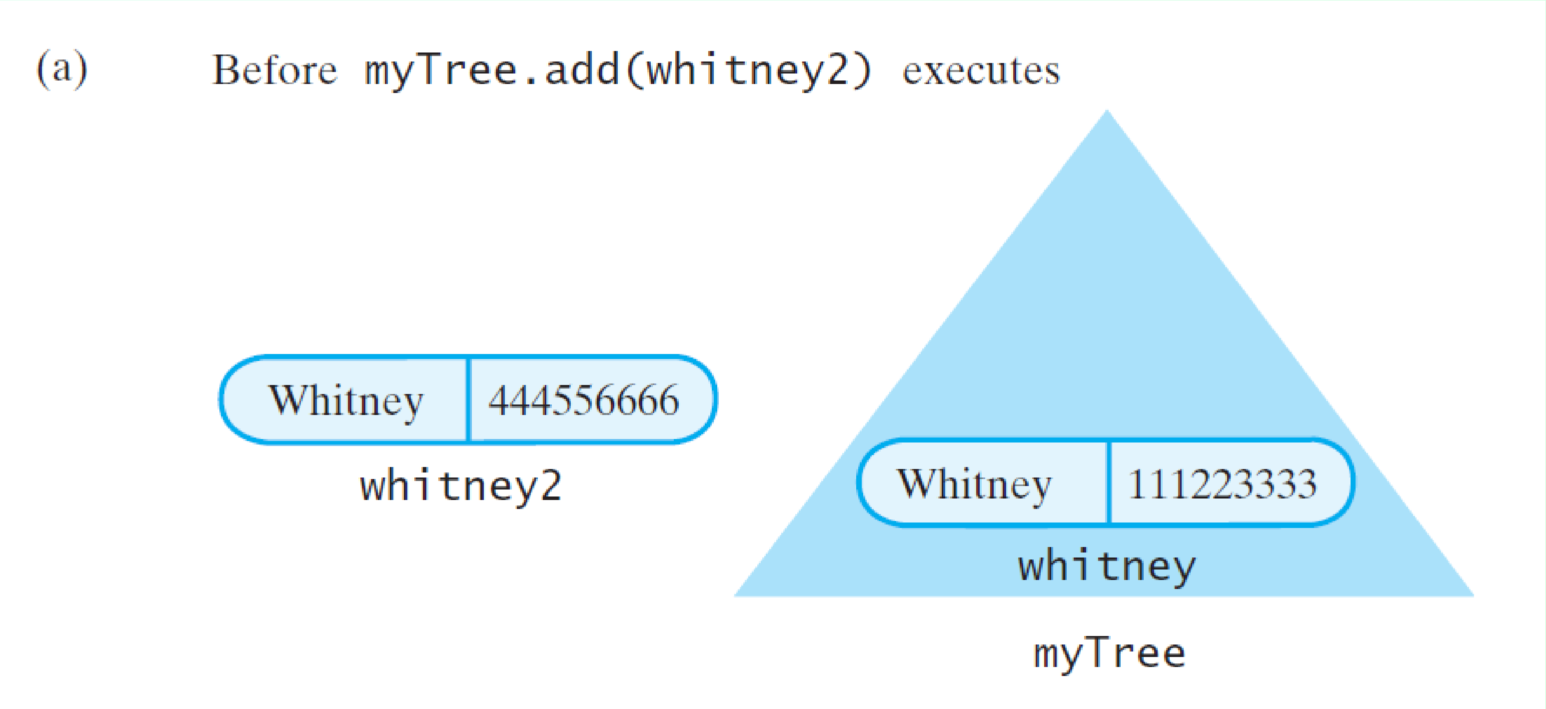


Figure 25-2 Adding an entry that matches an entry already in a binary search tree

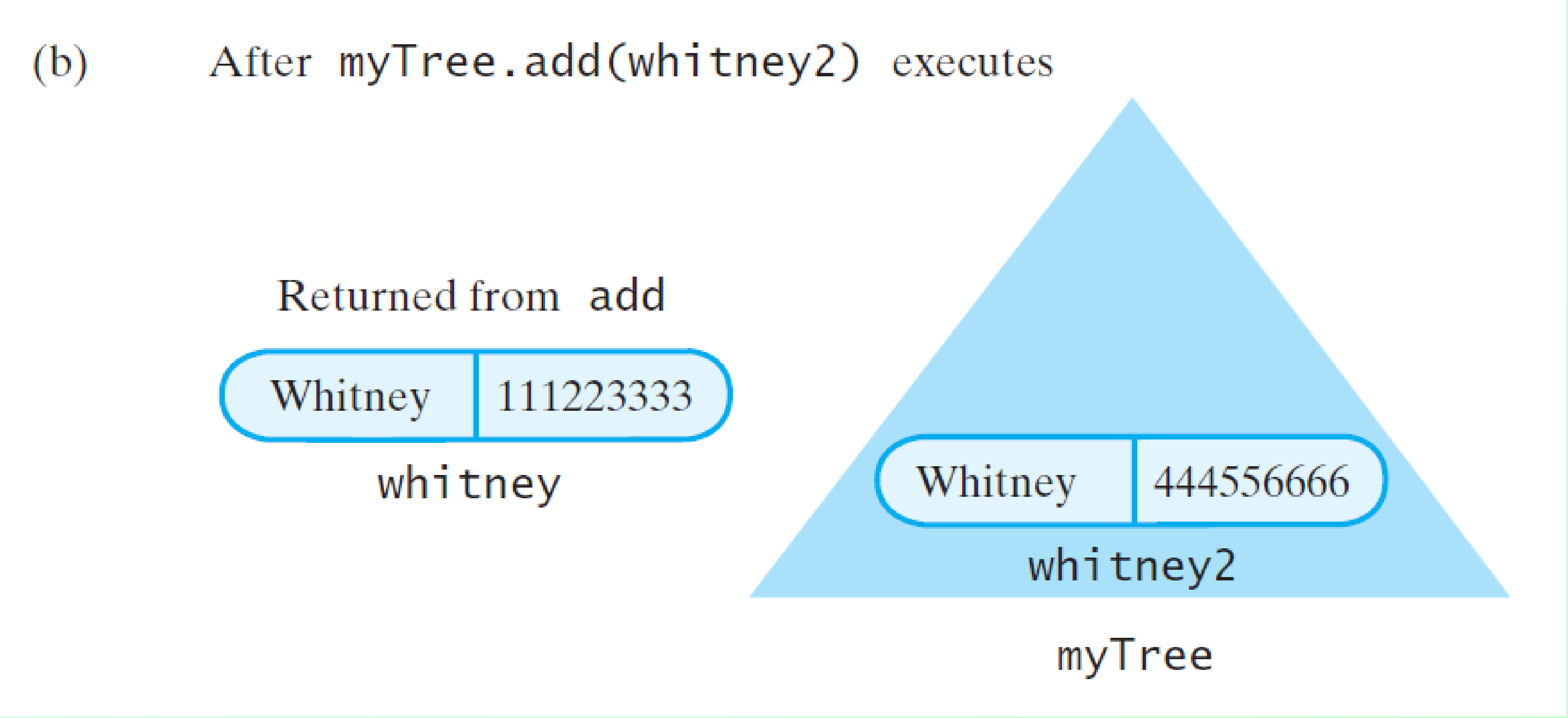


Figure 25-2 Adding an entry that matches an entry already in a binary search tree

Duplicate Entries

- For simplicity we do not allow duplicate entries
  - **`Add`** method prevents this
- Thus, modify definition
  - Data in node is greater than data in node’s left subtree
  - Data in node is less than *or equal* to data in node’s right subtree

Inorder traversal of the tree visits duplicate entry *Jared* immediately after visiting the original *Jared*.

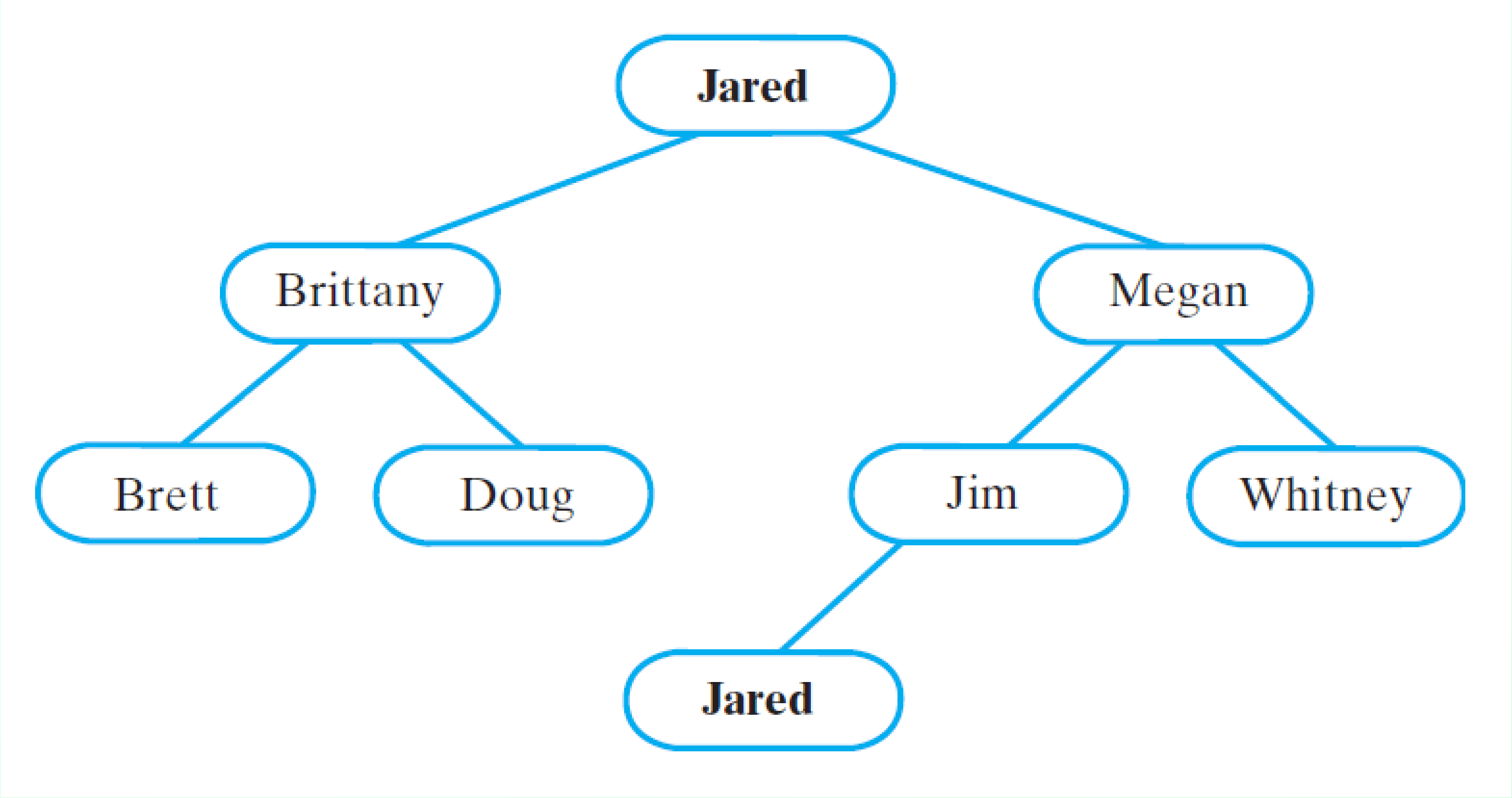
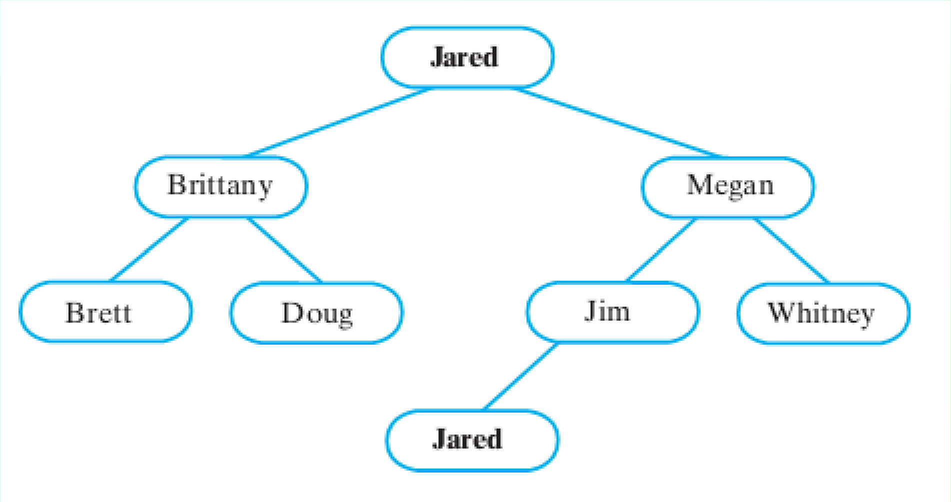
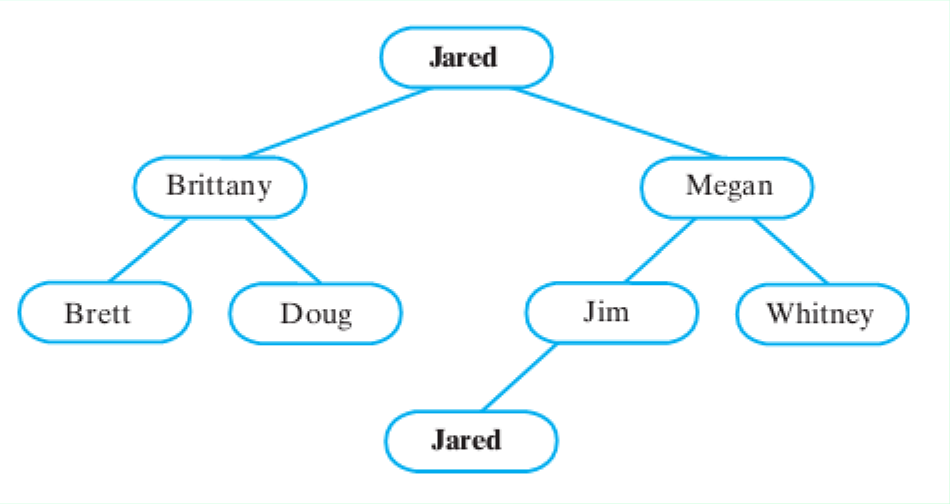


Figure 25-3 A binary search tree with duplicate entries

Question 1 If you add a duplicate entry Megan to the binary search tree in Figure 25-3 as a leaf, where should you place the new node?



Question 1 If you add a duplicate entry Megan to the binary search tree in Figure 25-3 as a leaf, where should you place the new node?



As the left child of the node that contains Whitney.

Beginning the Class Definition

- Source code of class **`BinarySearchTree`**, Listing 25-2
  - Note methods which disable **`setTree`** from **`BinaryTree`**
- de

Question 2 The second constructor in the class BinarySearchTree calls the method setRootNode . Is it possible to replace this call with the call setRootData(rootEntry)? Explain.

Question 3 Is it necessary to define the methods isEmpty and clear within the class BinarySearchTree ? Explain.

Question 2 The second constructor in the class BinarySearchTree calls the method setRootNode . Is it possible to replace this call with the call setRootData(rootEntry)? Explain.

No. The constructor first calls the default constructor of BinaryTree, which sets root to null. The method setRootData contains the call root.setData(rootData), which would cause an exception.

Question 3 Is it necessary to define the methods isEmpty and clear within the class BinarySearchTree ? Explain.

No; BinarySearchTree inherits these methods from BinaryTree.

Q 4 When getEntry calls findEntry, it passes getRootNode() as the first argument. This argument's data type is BinaryNodeInterface<T>, which corresponds to the type of the parameter rootNode . If you change rootNode 's type to BinaryNode<T> , what other changes, if any, must you make?

Question 5 Under what circumstance will a client of BinarySearchTree be able to call the other methods in TreeIteratorInterface ? Under what circumstance will such a client be unable to call these methods?

Q 4 When getEntry calls findEntry, it passes getRootNode() as the first argument. This argument's data type is BinaryNodeInterface<T>, which corresponds to the type of the parameter rootNode . If you change rootNode 's type to BinaryNode<T> , what other changes, if any, must you make?

In getEntry 's call to findEntry, you would cast getRootNode() to BinaryNode<T> , as follows:

```
findEntry((BinaryNode<T>)getRootNode(), entry)
```

Within findEntry, the first recursive call to findEntry must be

```
findEntry((BinaryNode<T>)rootNode.getLeftChild(), entry)
```

since the return type of getLeftChild is BinaryNodeInterface<T>. Analogous comments apply to the second recursive call and getRightChild.

Question 5 Under what circumstance will a client of BinarySearchTree be able to call the other methods in TreeIteratorInterface ? Under what circumstance will such a client be unable to call these methods?

The situation is like that described for setTree in Segment 25.6. BinarySearchTree inherits the methods declared in TreeIteratorInterface from BinaryTree. An object whose static type is BinarySearchTree can invoke these methods, but an object whose static type is SearchTreeInterface cannot.

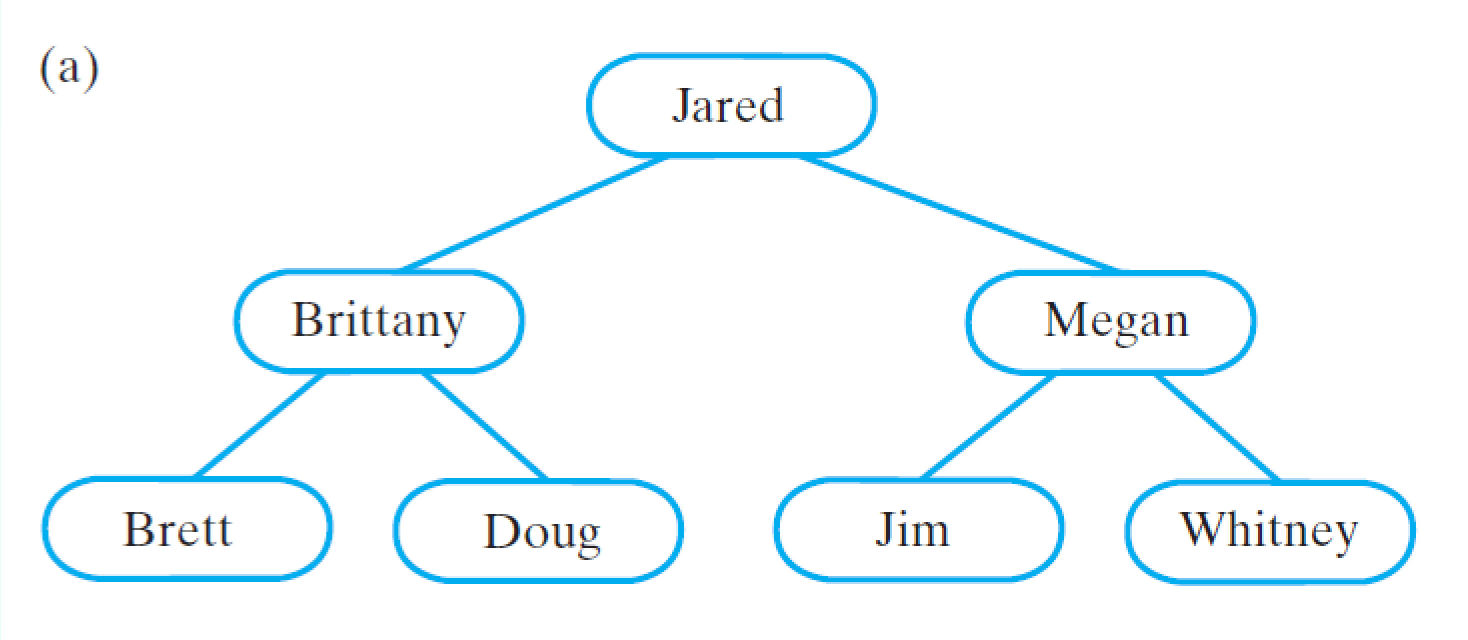


FIGURE 25-4 (a) A binary search tree;

Note recursive implementation

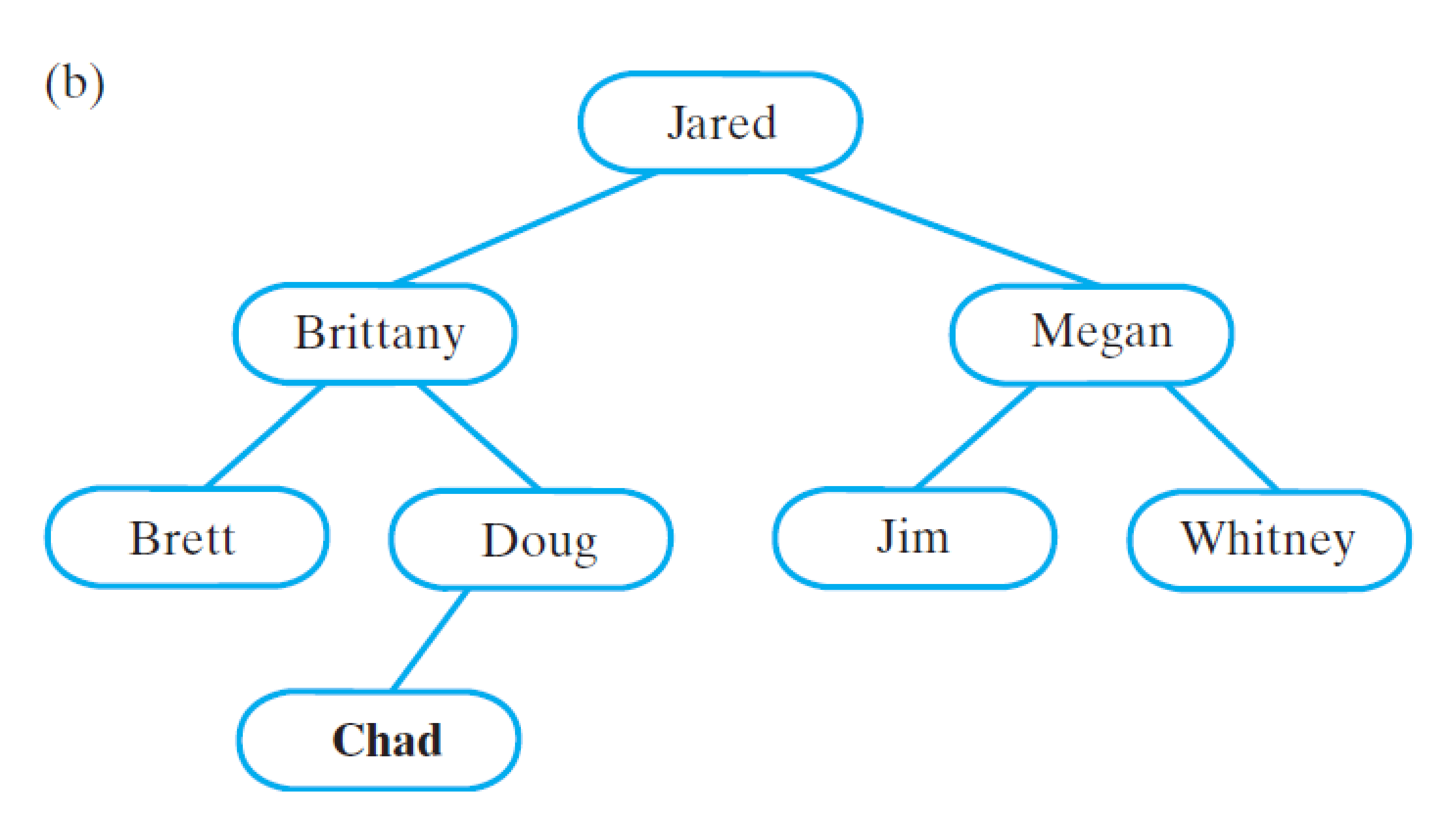
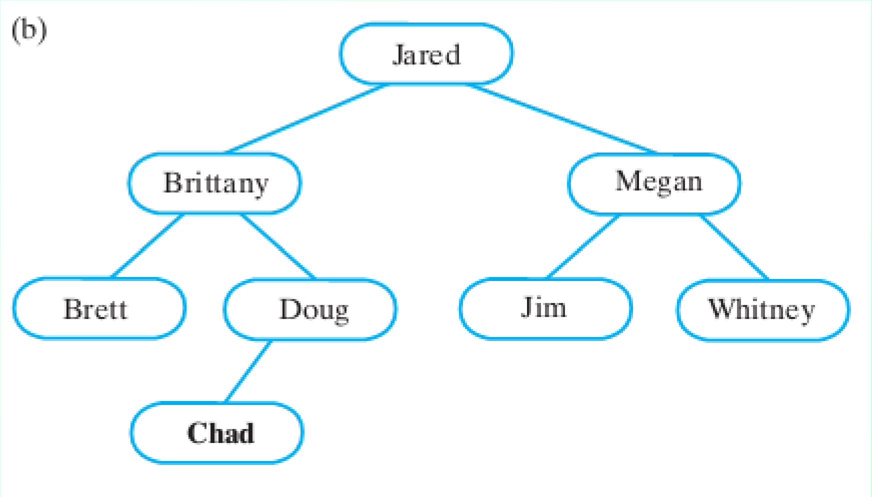
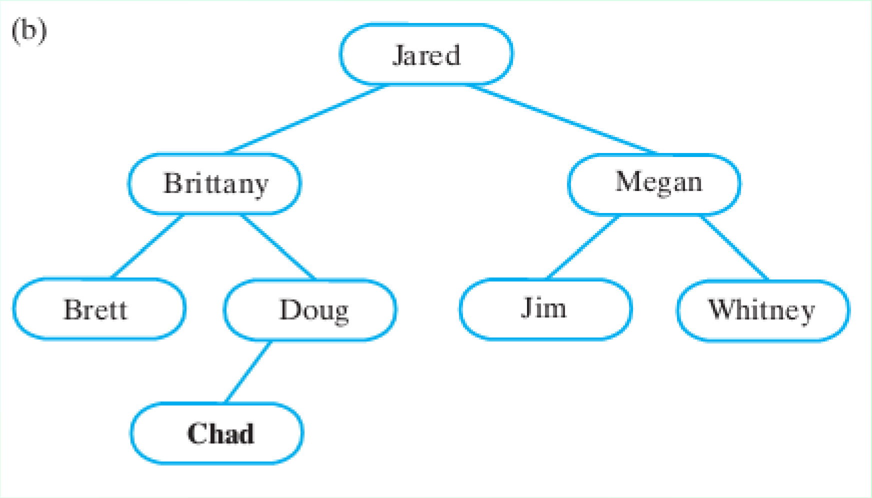


FIGURE 25-4 (b) the same tree after adding *Chad*

Q 6 Add the names Chris, Jason, and Kelley to the binary search tree in Figure 25-4b.

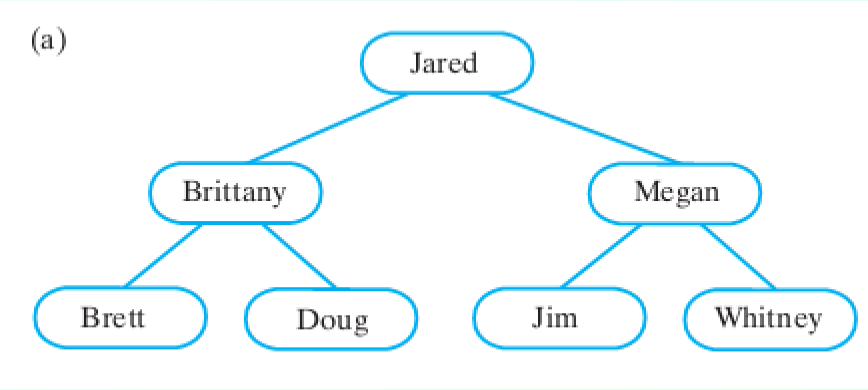


Q 6 Add the names Chris, Jason, and Kelley to the binary search tree in Figure 25-4b.

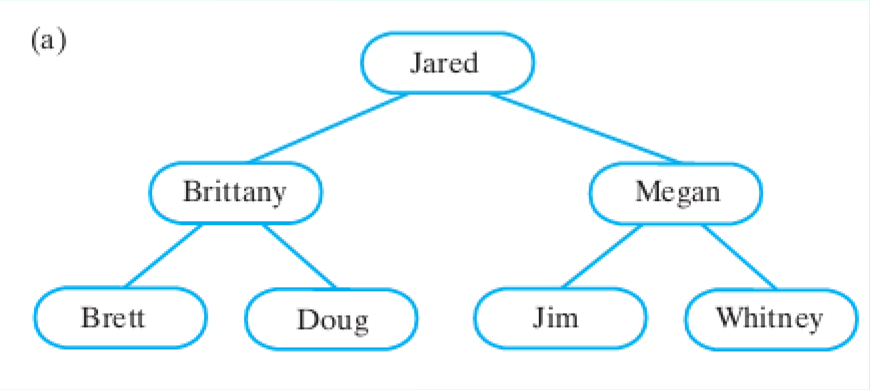


Chris is the right child of Chad. Jason is the left child of Jim . Kelley is the right child of Jim.

Q 7 Add the name Miguel to the binary search tree in Figure 25-4a, and then add Nancy. Now go back to the original tree and add Nancy and then add Miguel . Does the order in which you add the two names affect the tree that results?



Q 7 Add the name Miguel to the binary search tree in Figure 25-4a, and then add Nancy. Now go back to the original tree and add Nancy and then add Miguel . Does the order in which you add the two names affect the tree that results?



When you add Miguel first, Miguel is the left child of Whitney, and Nancy is the right child of Miguel. When you add Nancy first, Nancy is the left child of Whitney, and Miguel is the left child of Nancy. Thus, the order of the additions does affect the tree that results.

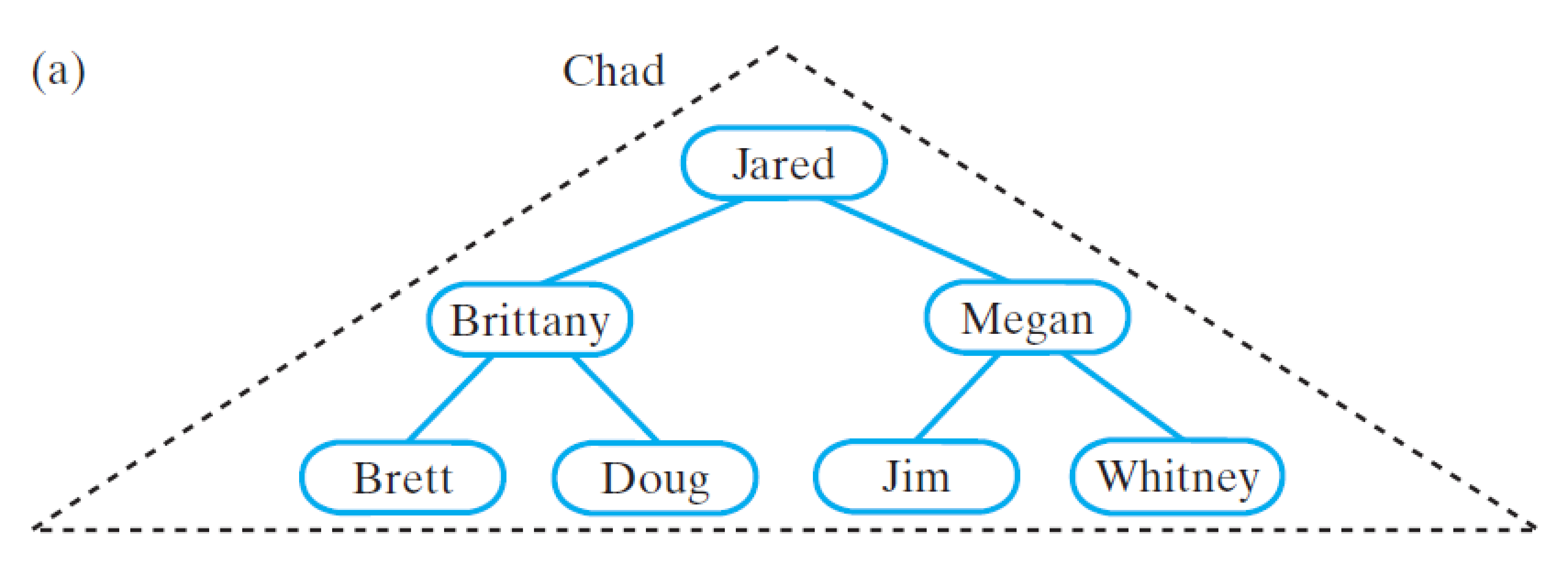


Figure 25-5 Recursively adding Chad to smaller subtrees of a binary search tree

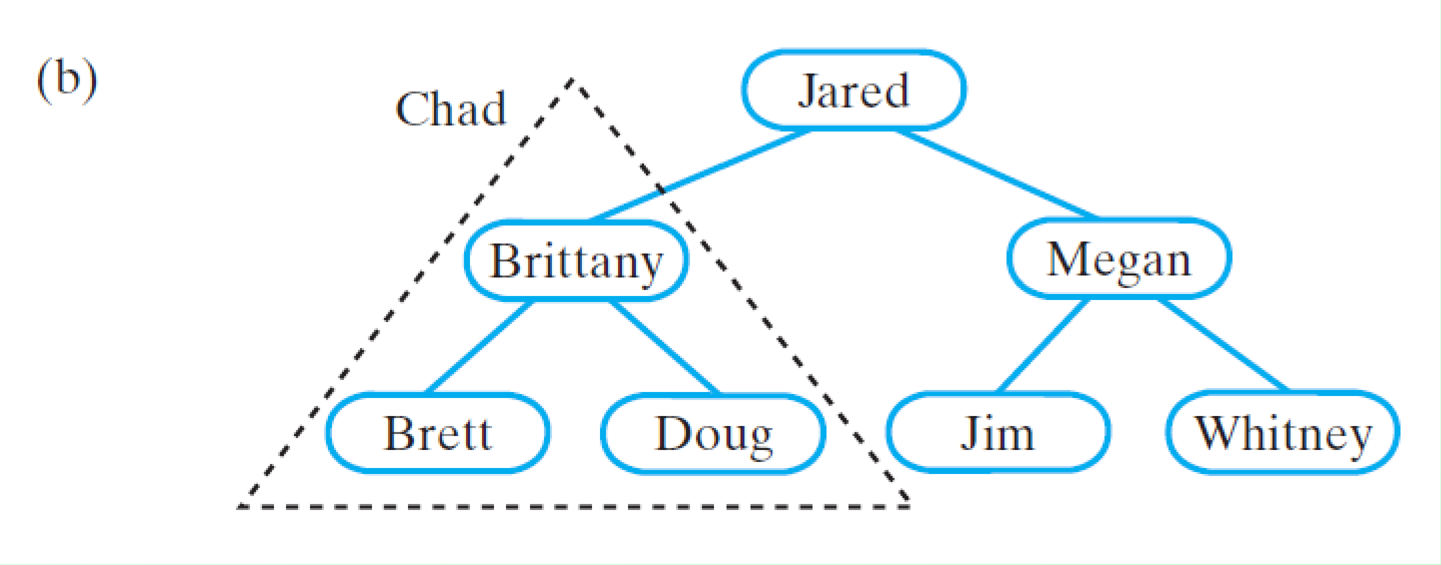


Figure 25-5 Recursively adding Chad to smaller subtrees of a binary search tree

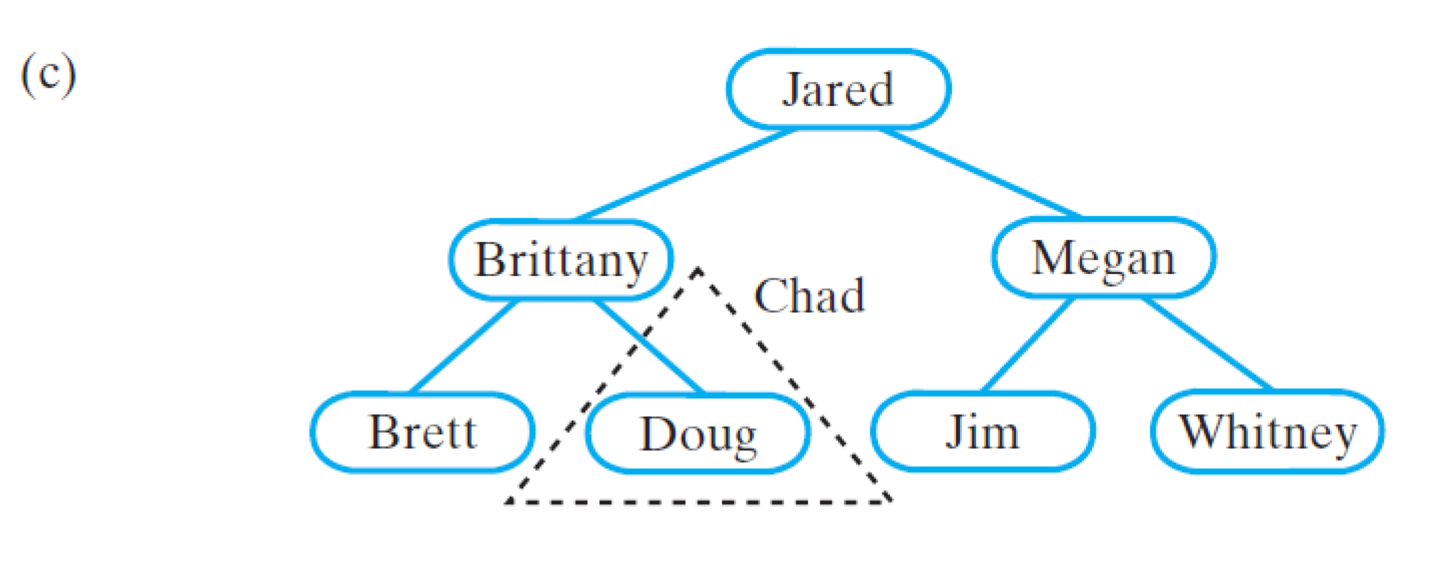


Figure 25-5 Recursively adding Chad to smaller subtrees of a binary search tree

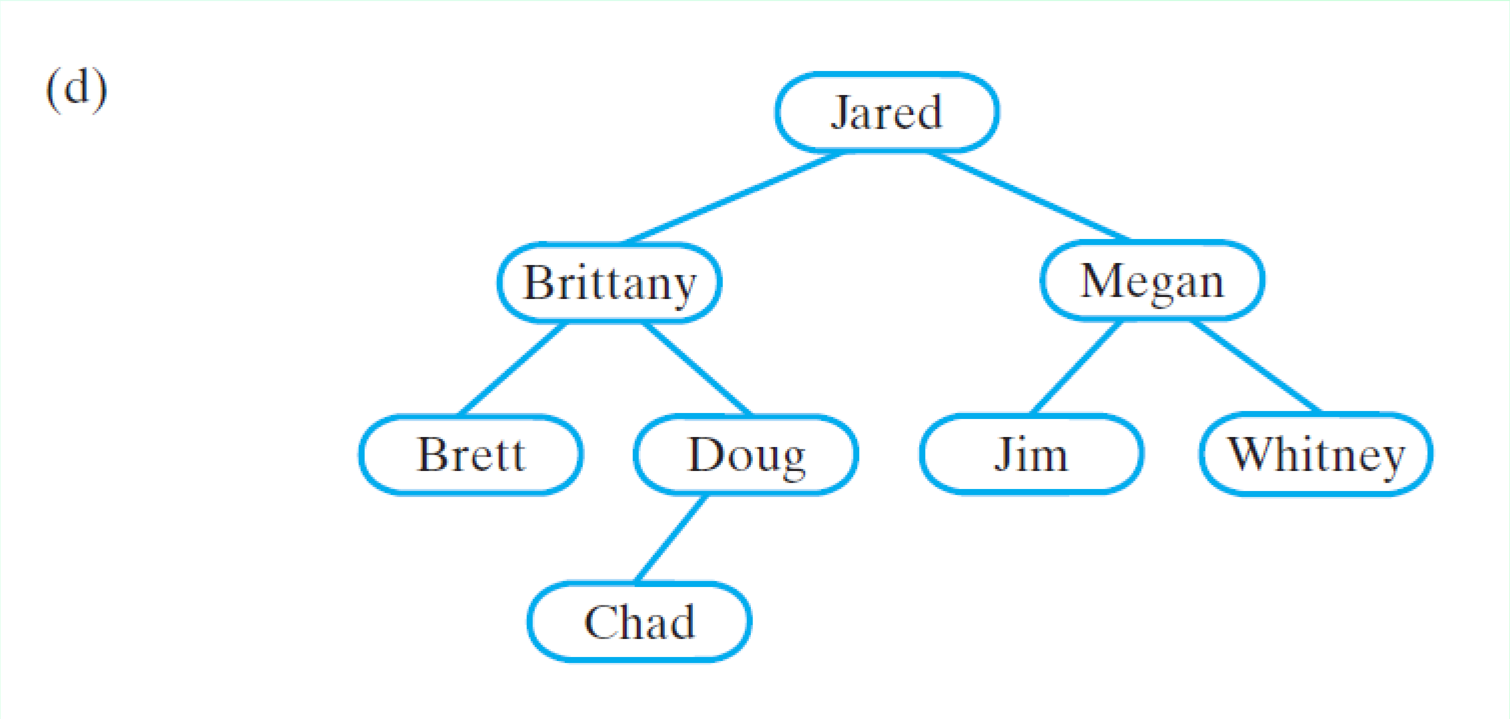


Figure 25-5 Recursively adding Chad to smaller subtrees of a binary search tree

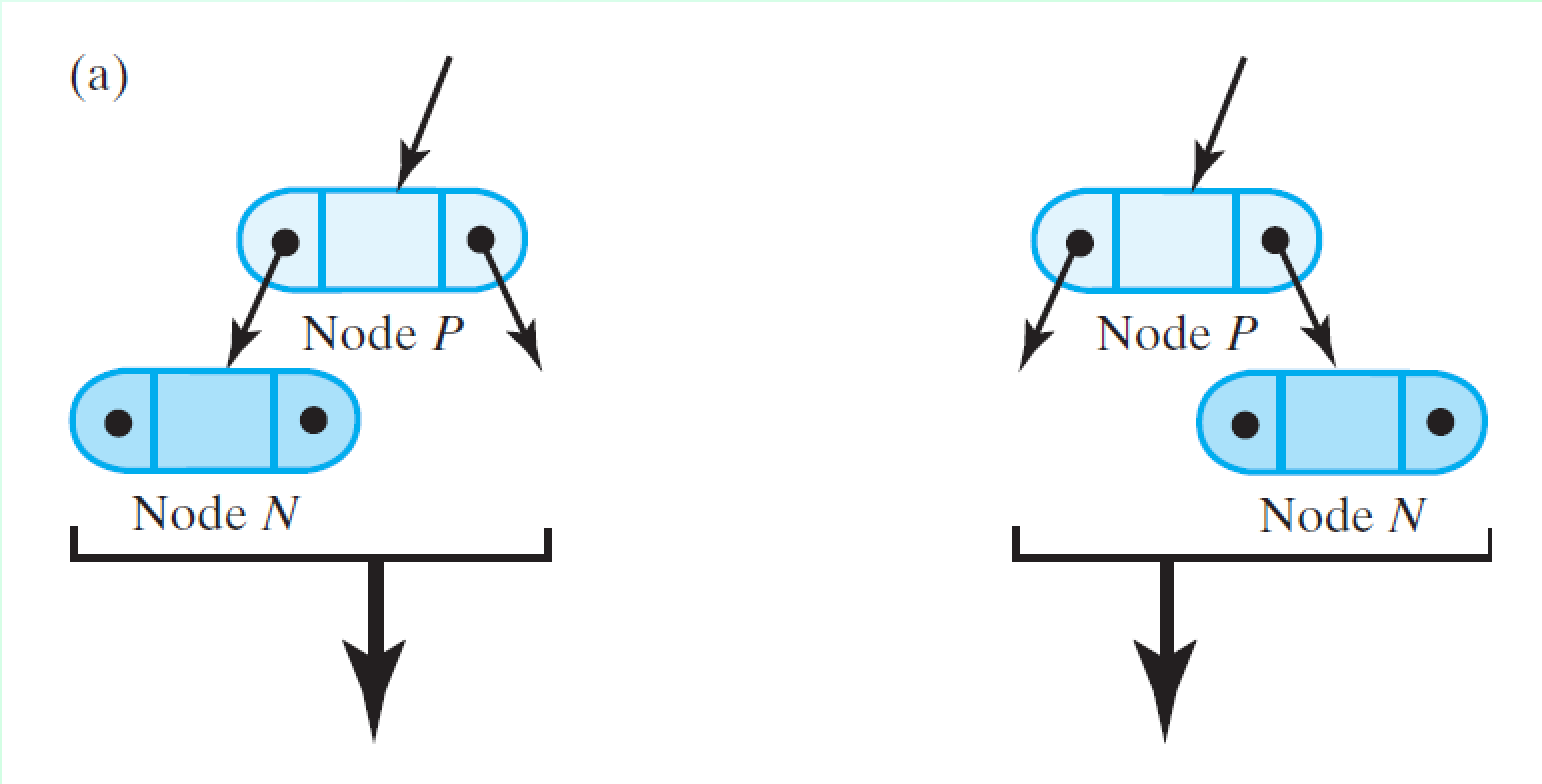


Figure 25-6 (a) Two possible configurations of a leaf node N;

Figure 25-6 (b) the resulting two possible configurations after removing node *N*

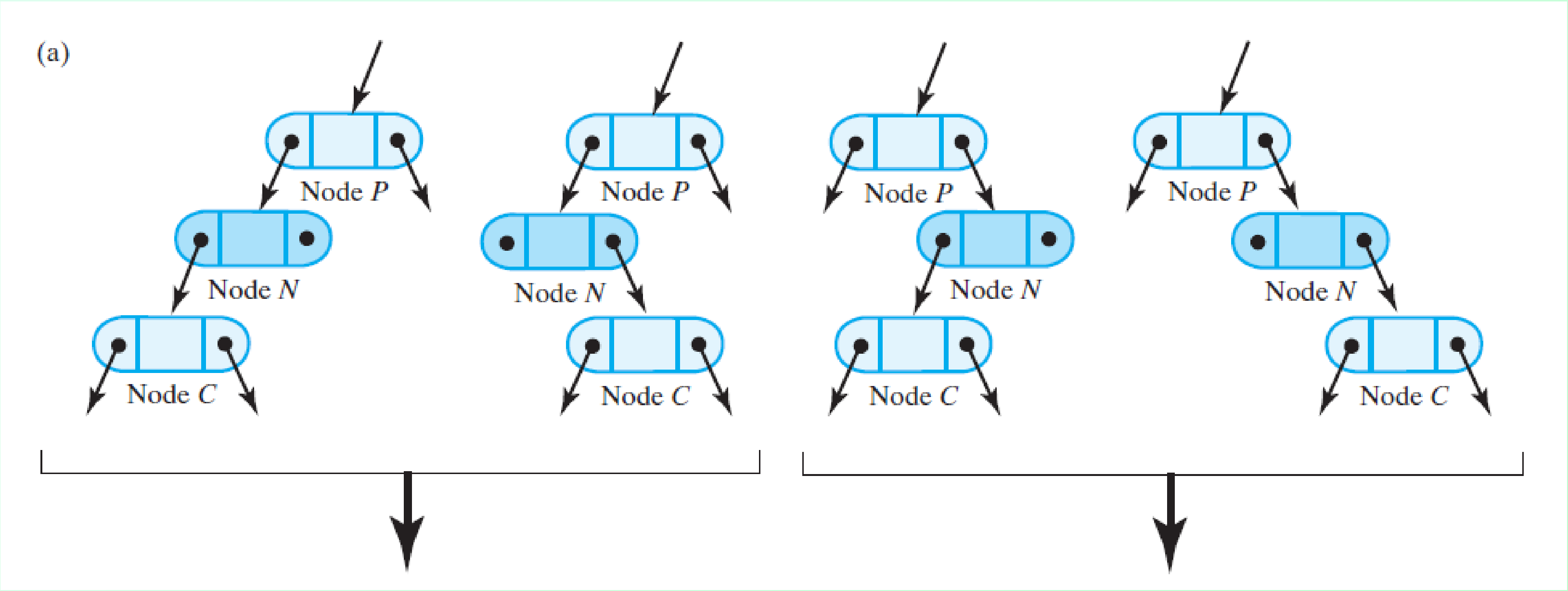


Figure 25-7 (a) Four possible configurations of a node N that has one child;

Figure 25-7 (b) the resulting two possible configurations after removing node *N*

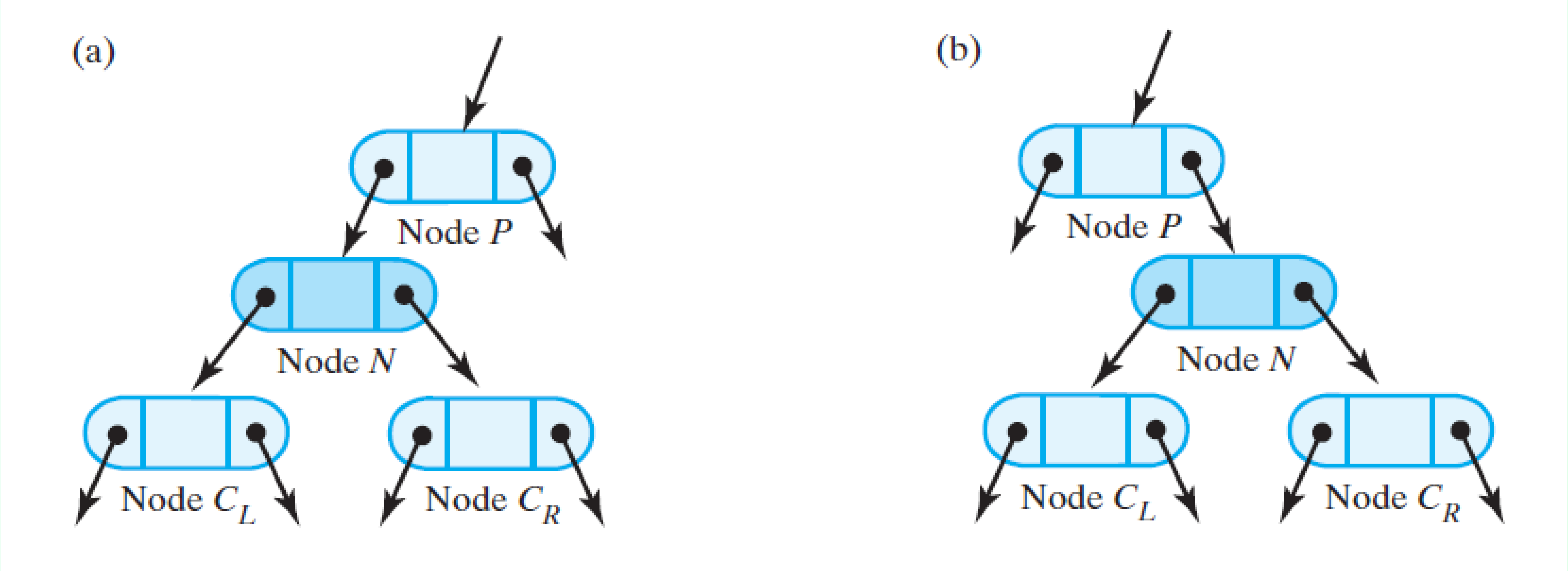


Figure 25-8 Two possible configurations of a node N that has two children

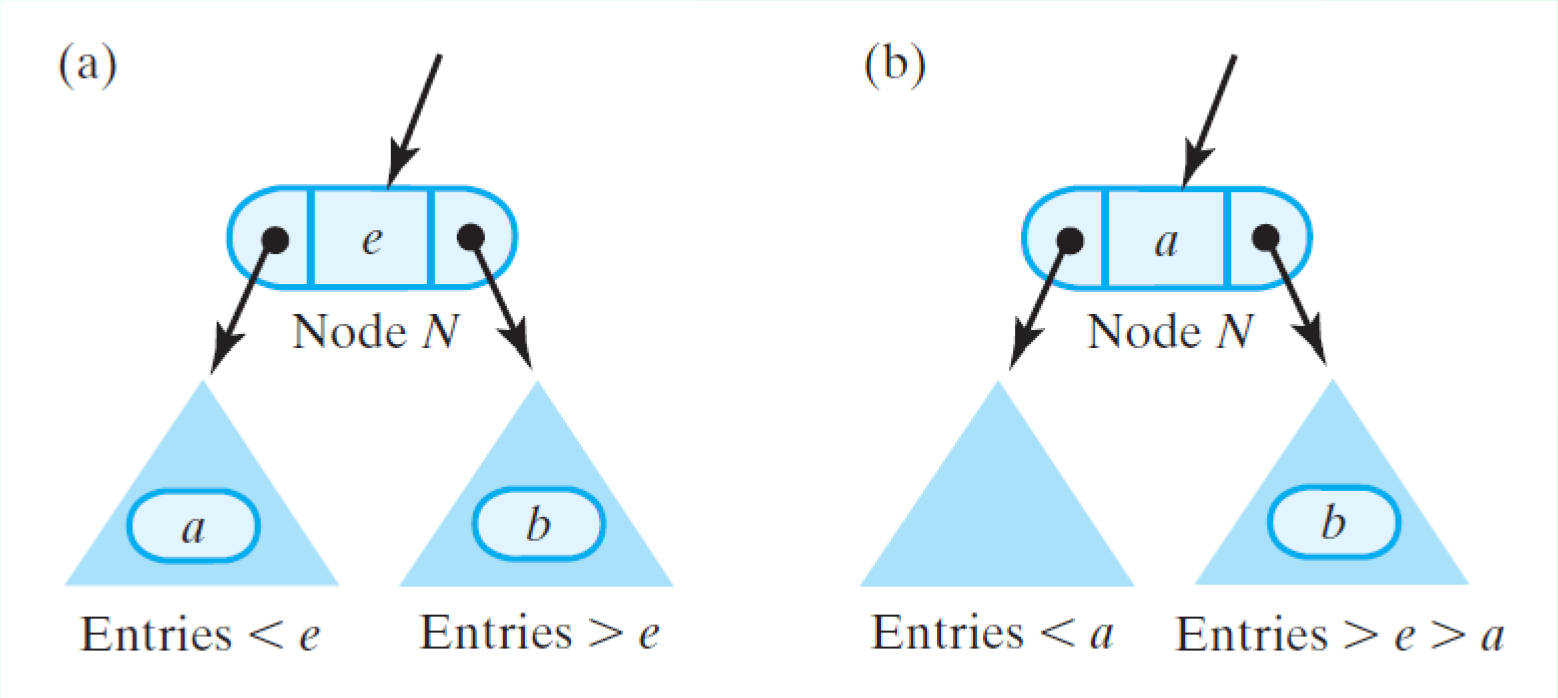


Figure 25-9 Node *N* and its subtrees: (a) the entry *a* is immediately before the entry *e*, and *b* is immediately after *e*; (b) after deleting the node that contained *a* and replacing *e* with *a*

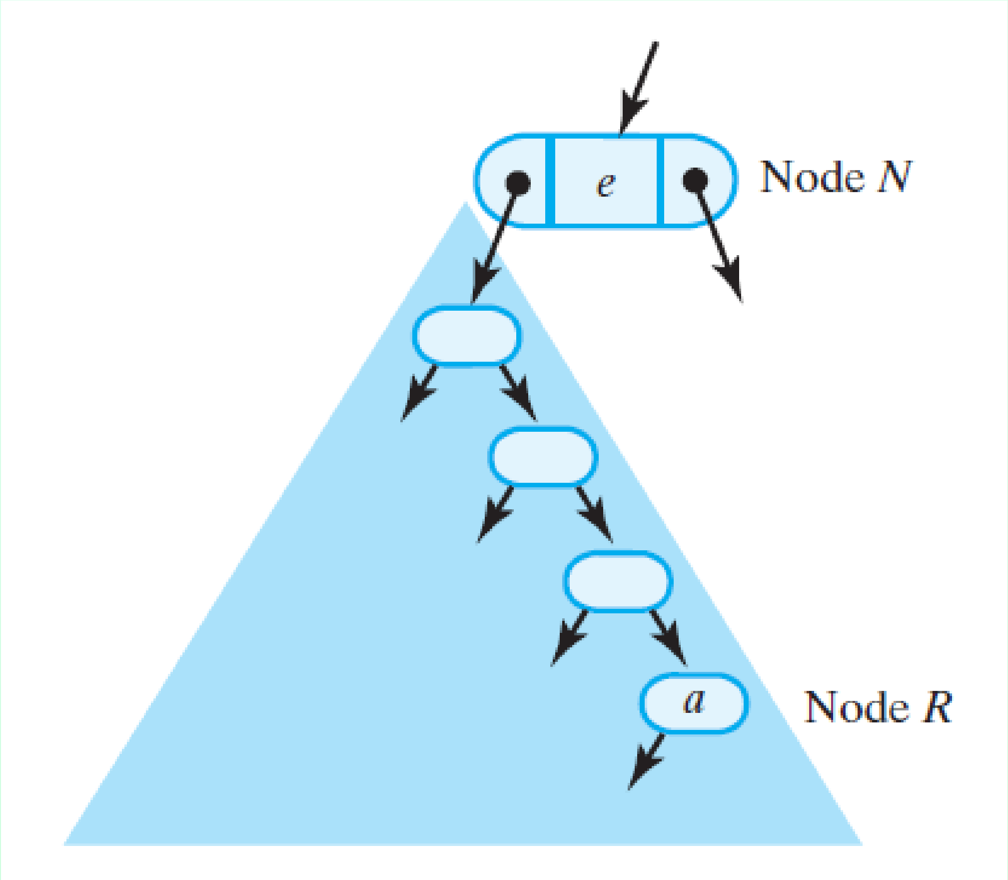


Figure 25-10 The largest entry *a* in node *N*'s left subtree occurs in the subtree's rightmost node *R*

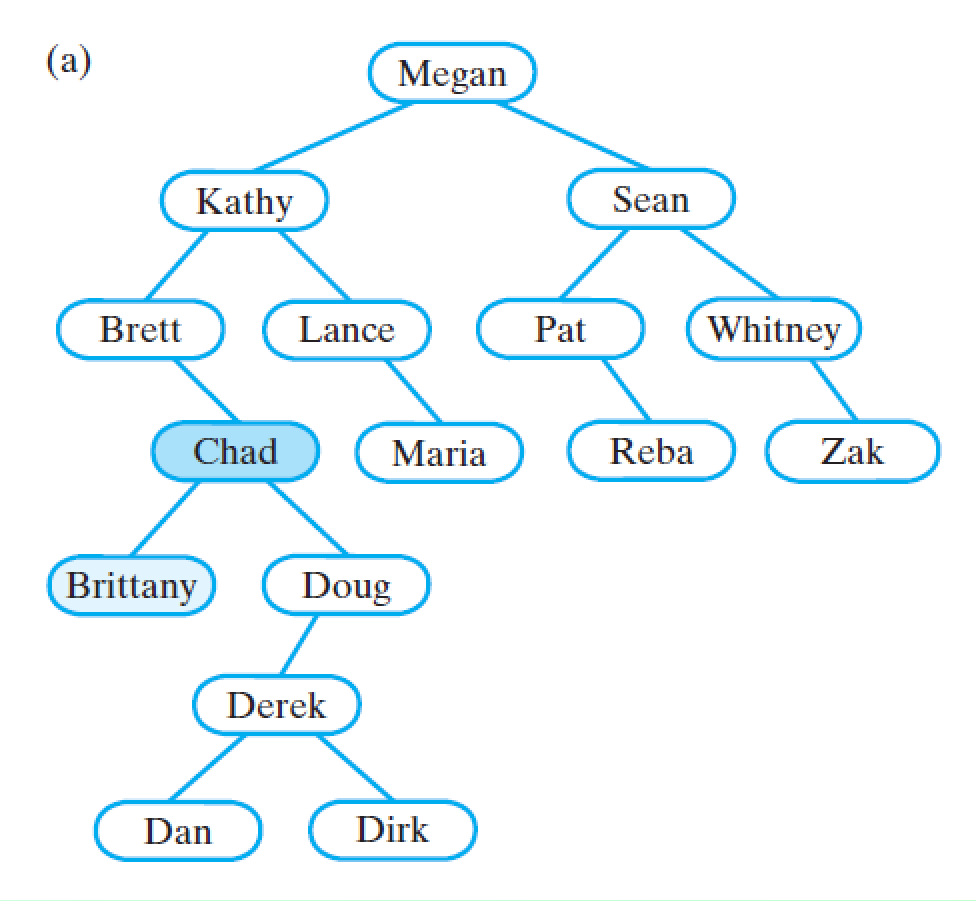


Figure 25-11 (a) A binary search tree

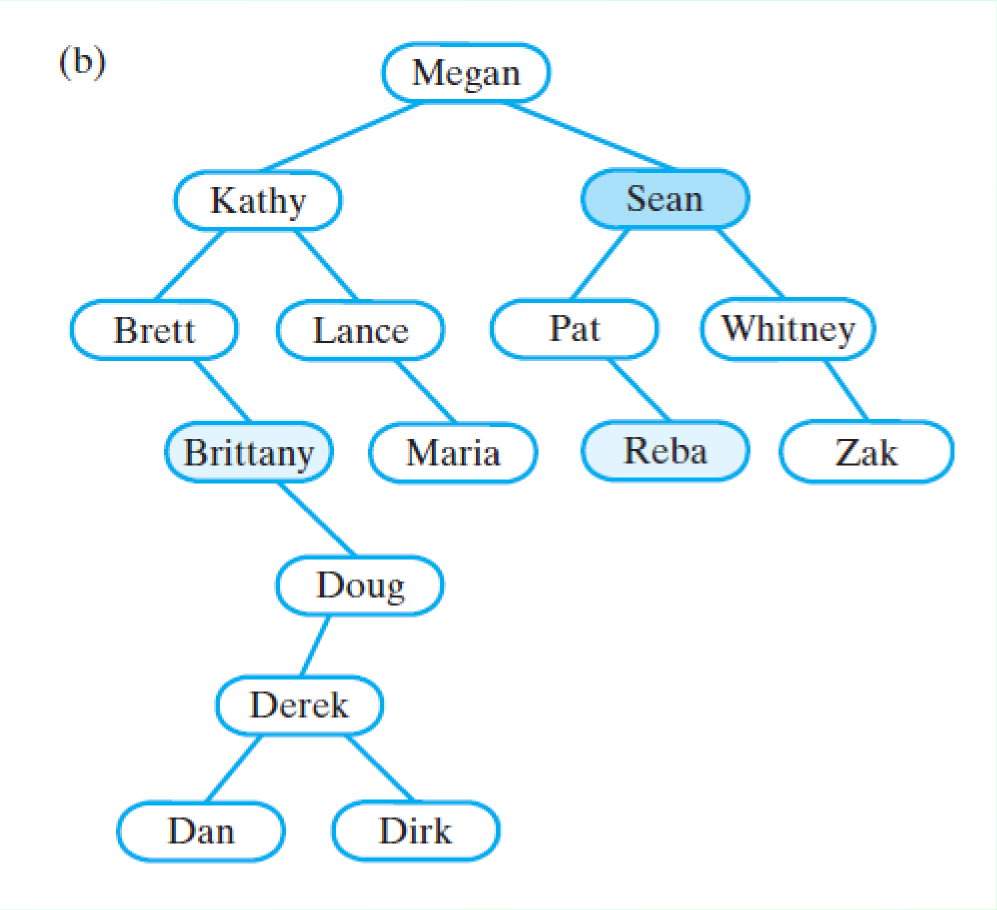


Figure 25-11 (b) after removing *Chad*

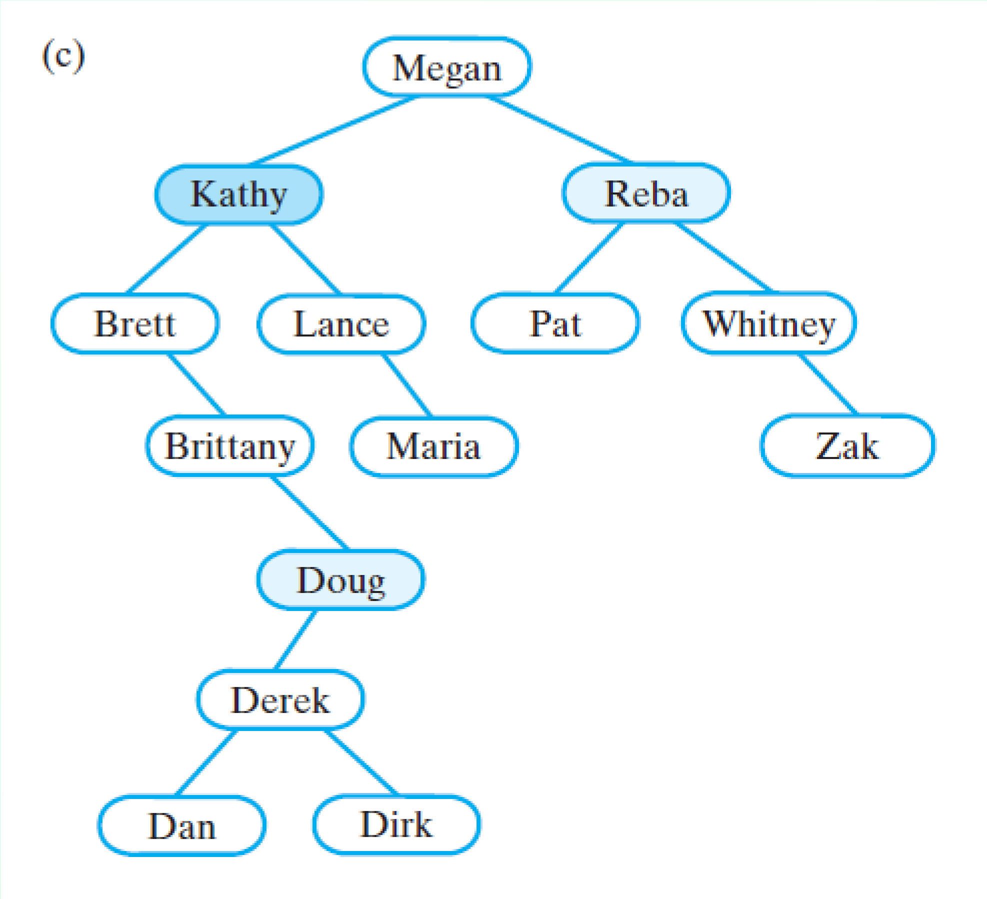


Figure 25-11 (c) after removing *Sean*

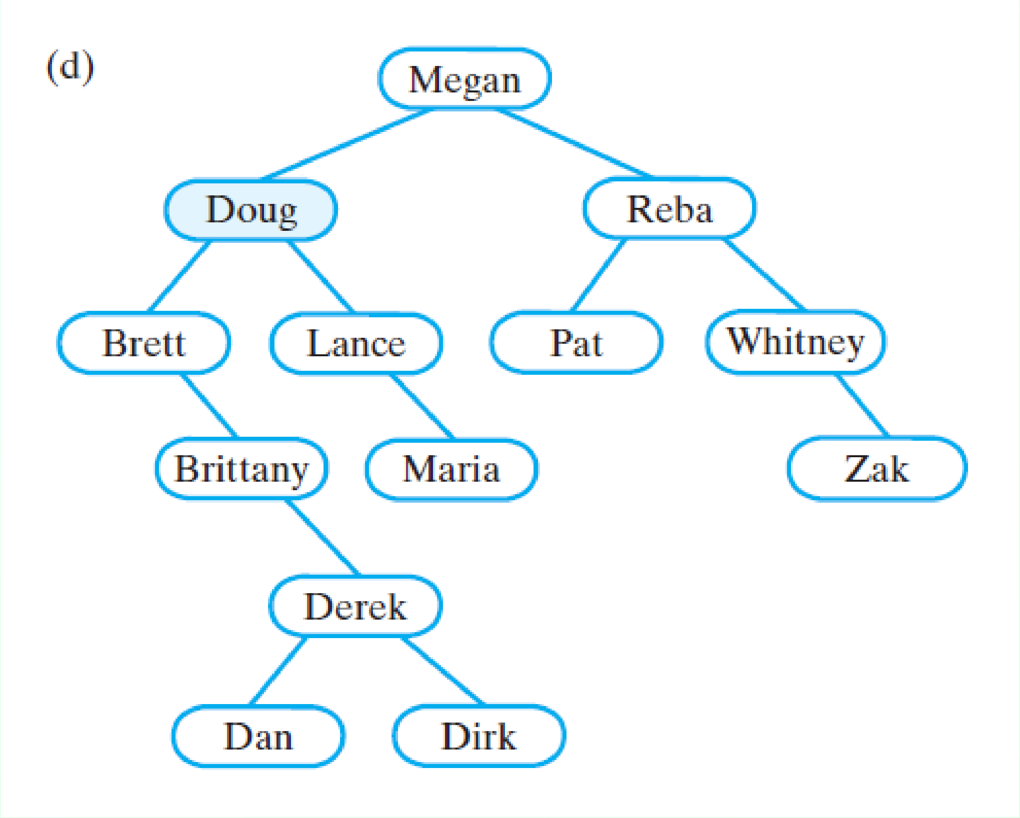
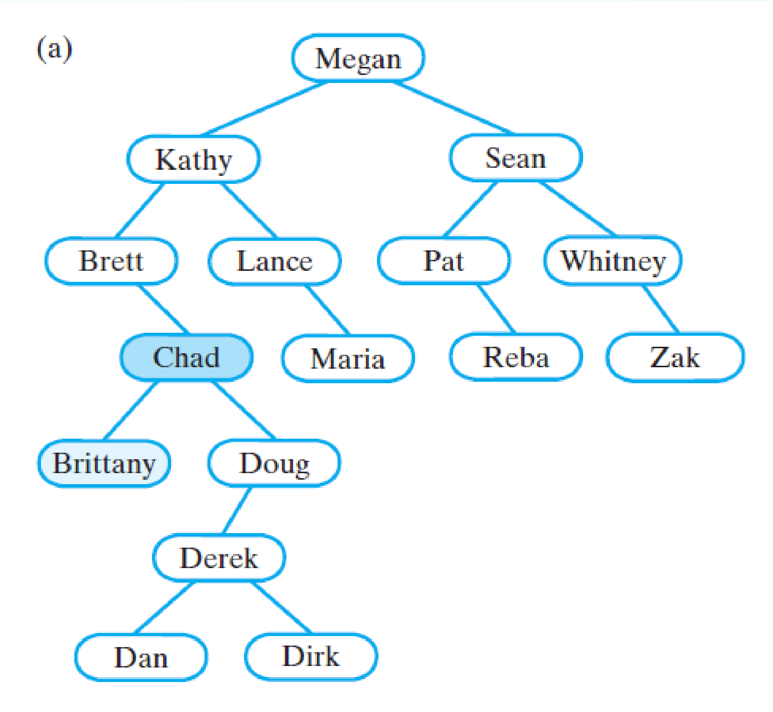
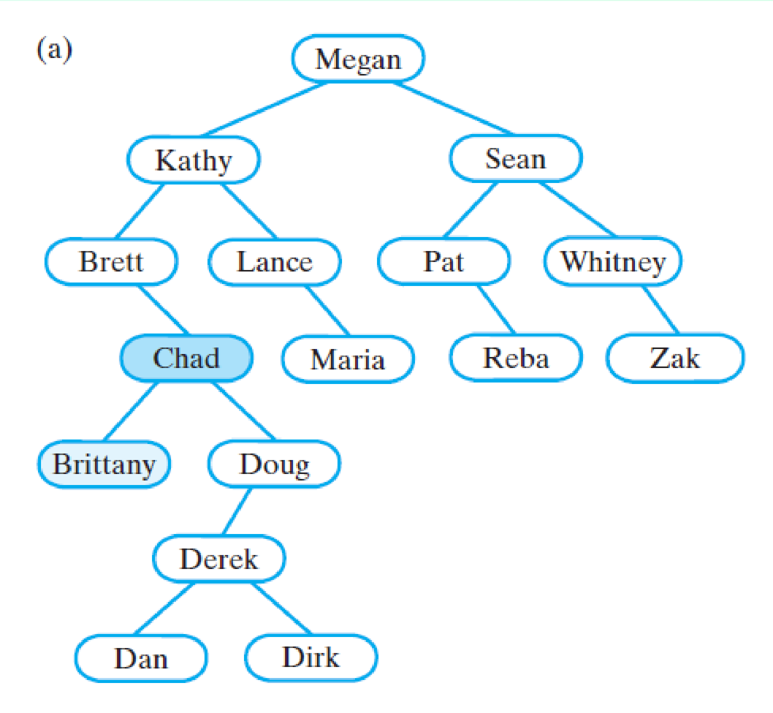


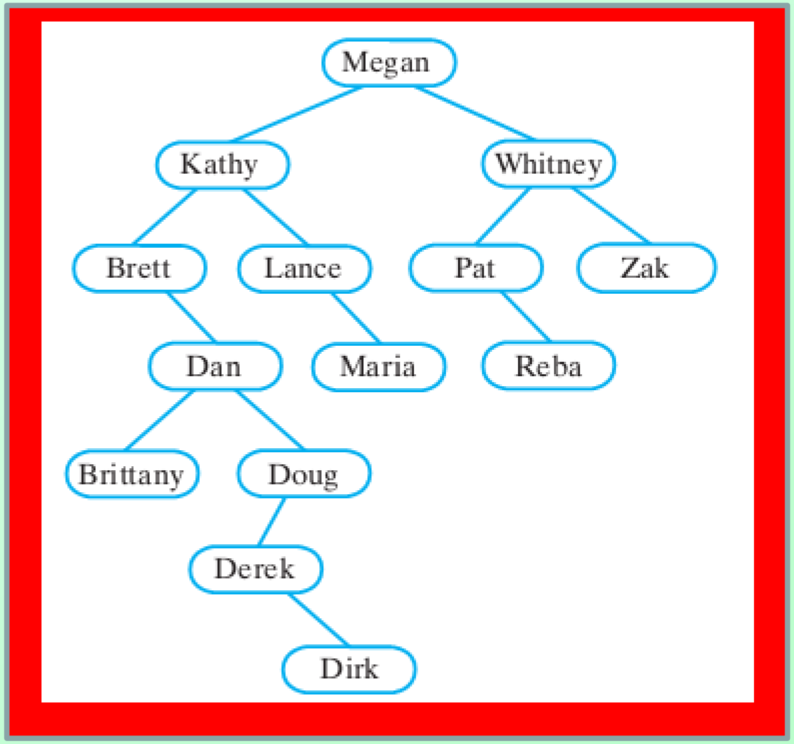
Figure 25-11 (d) after removing *Kathy*

Q 8 The second algorithm described in Segment 25.25 involves the inorder successor. Using this algorithm, remove Sean and Chad from the tree in Figure 25-11a.

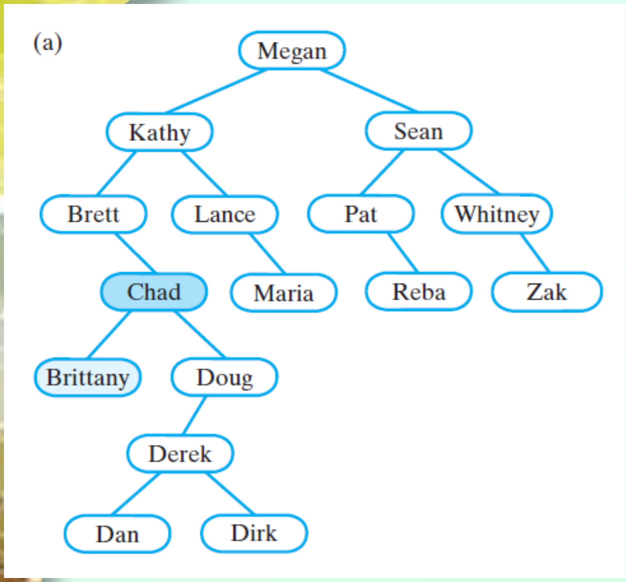


Q 8 The second algorithm described in Segment 25.25 involves the inorder successor. Using this algorithm, remove Sean and Chad from the tree in Figure 25-11a.

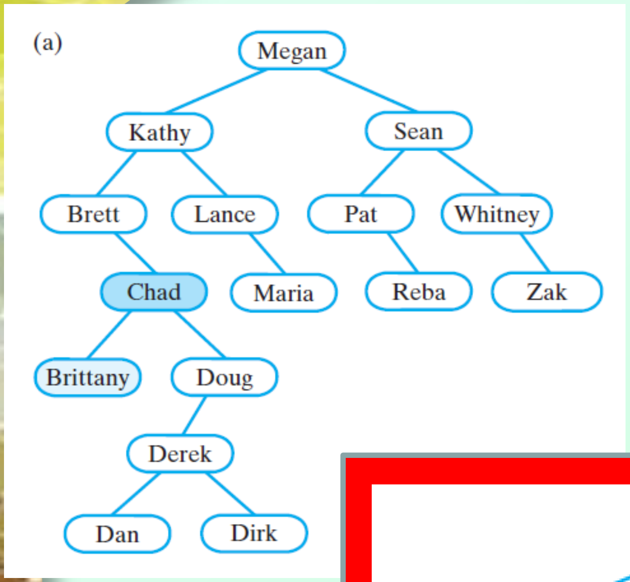


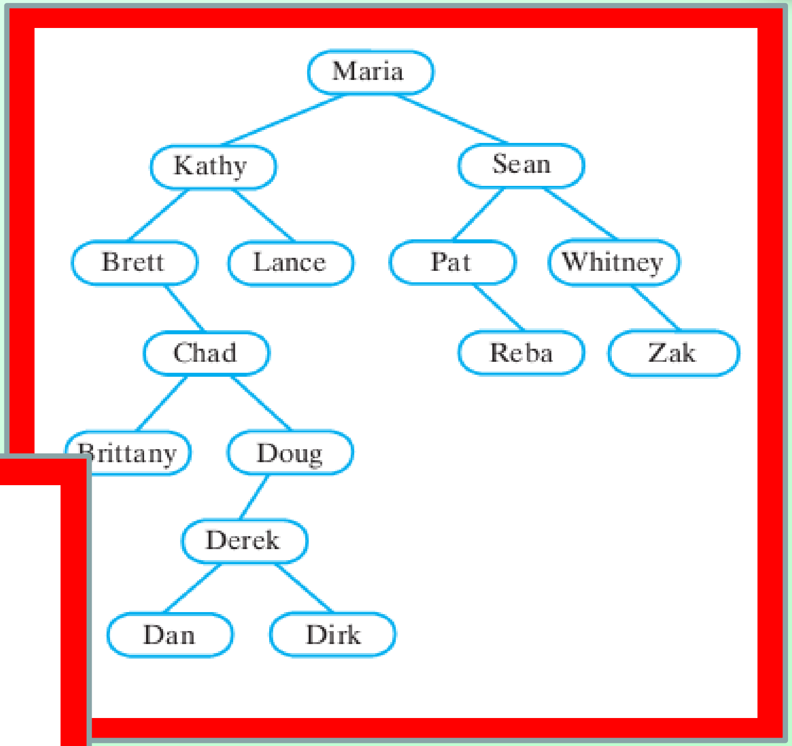


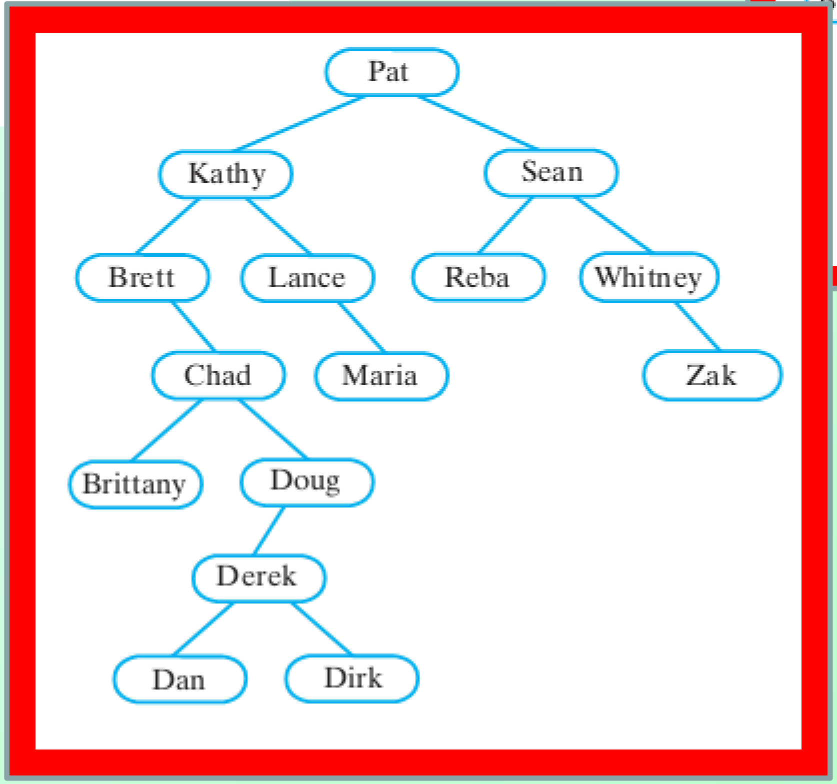
Q 9 Remove Megan from the tree in Figure 25-11a in two different ways.



Q 9 Remove Megan from the tree in Figure 25-11a in two different ways.







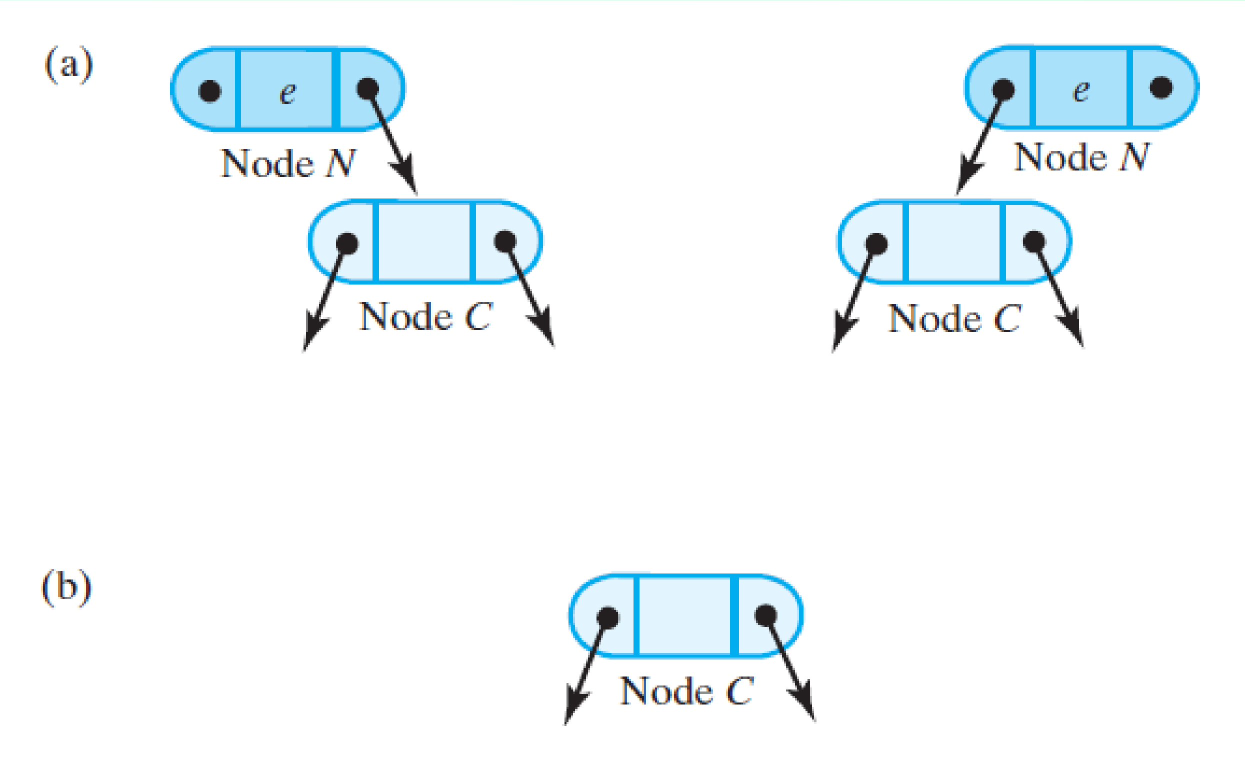


Figure 25-12 (a) two possible configurations of a root that has one child; (b) after removing the root

Efficiency of Operations

- Operations **`add`**, **`remove`**, and **`getEntry`** require search that begins at root
- Worst case:
  - Searches begin at root and examine each node on path that ends at leaf
  - Number of comparisons each operation requires, directly proportional to height *h* of tree

Efficiency of Operations

- Tallest tree has height *n* if it contains *n* nodes
- Operations **`add`**, **`remove`**, and **`getEntry`** are O(*h*)
- Note different binary search trees can contain same data

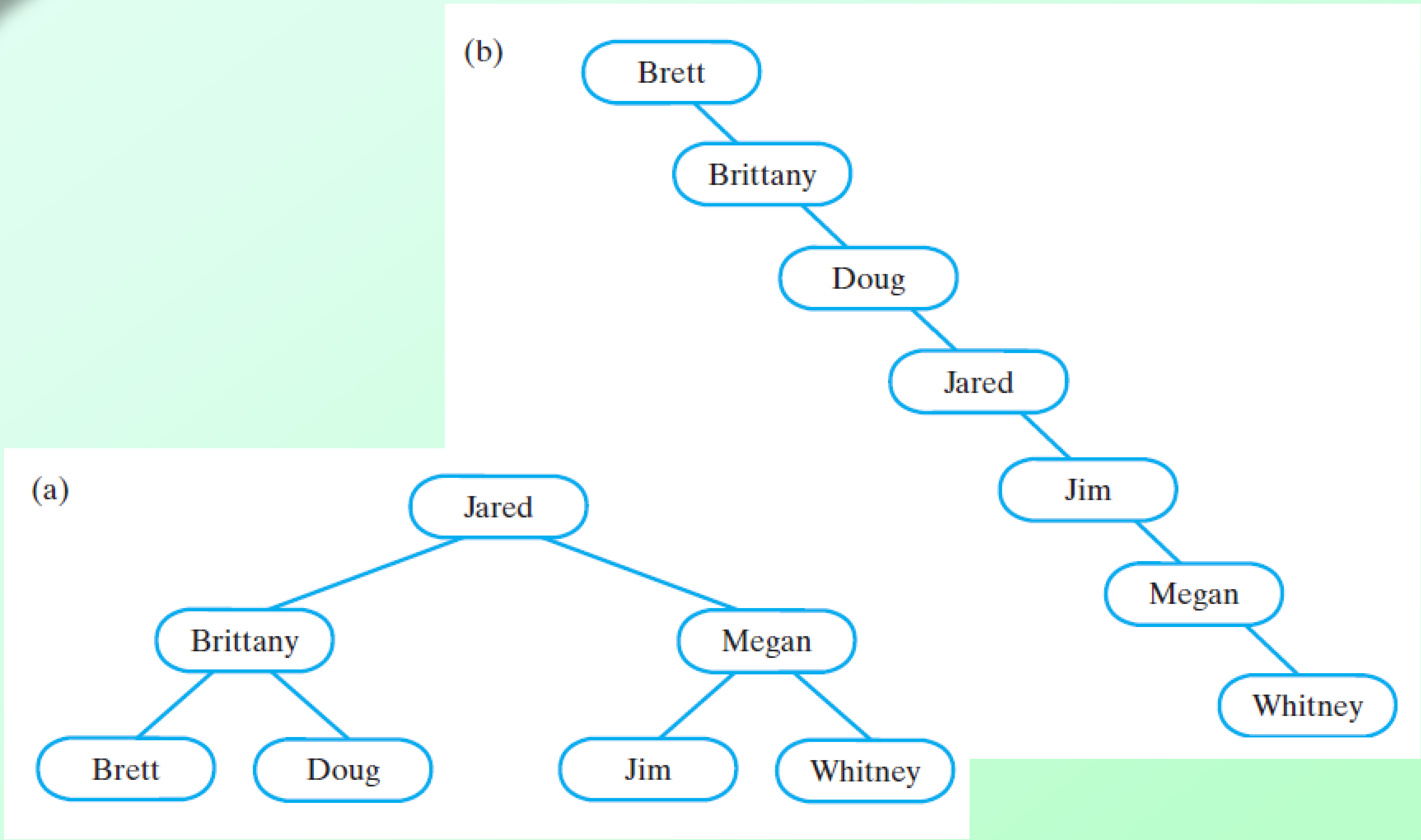


Figure 25-13 Two binary search trees that contain the same data

Efficiency of Operations

- Tallest tree has height *n* if it contains *n* nodes
  - Search is an O(*n*) operation
- Shortest tree is full
  - Searching full binary search tree is O(log *n*) operation

Q 11 Using Big Oh notation, what is the time complexity of the method contains?

Q 12 Using Big Oh notation, what is the time complexity of the method isEmpty?

Q 11 Using Big Oh notation, what is the time complexity of the method contains?

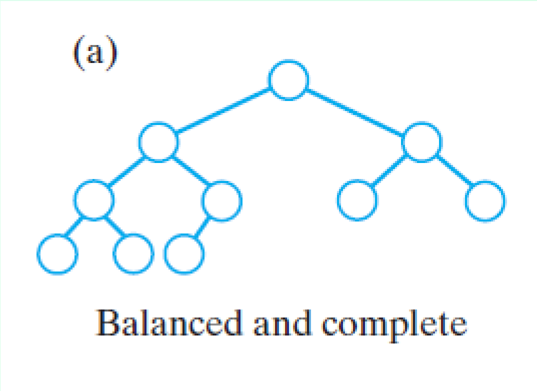
Since the method contains invokes getEntry , the efficiency of these methods is the same. So if the tree's height is as small as possible, the efficiency is O(log n). If the tree's height is as large as possible, the efficiency is O(n).

Q 12 Using Big Oh notation, what is the time complexity of the method isEmpty?

O(1).

Importance of Balance

- Full binary search tree not necessary to get O(log *n*) performance
  - Complete tree will also give O(log *n*) performance
- Completely balanced tree
  - Subtrees of each node have exactly same height
- Height balanced tree
  - Subtrees of each node in tree differ in height by no more than 1



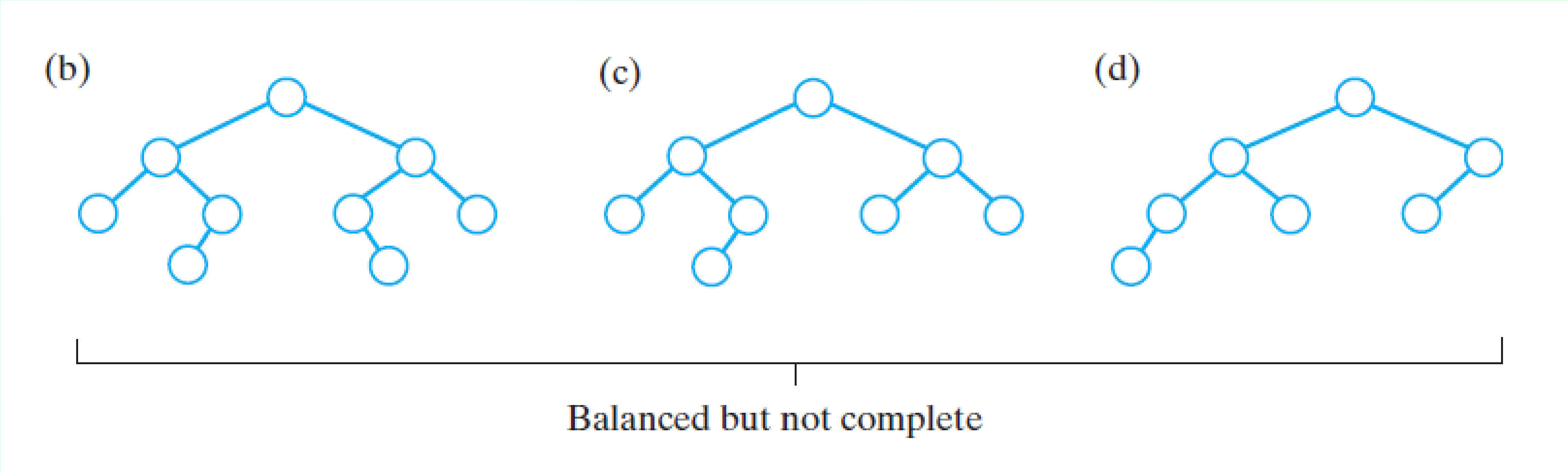
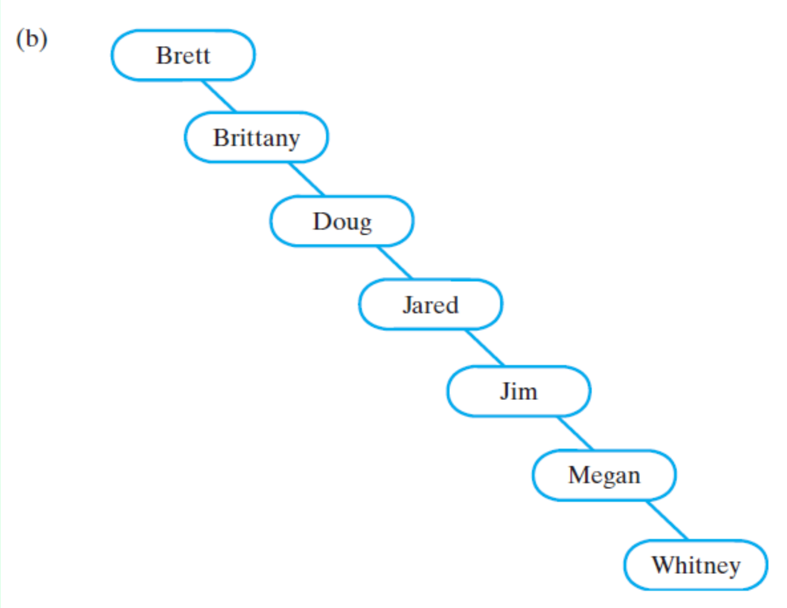


Figure 25-14 Some binary trees that are height balanced

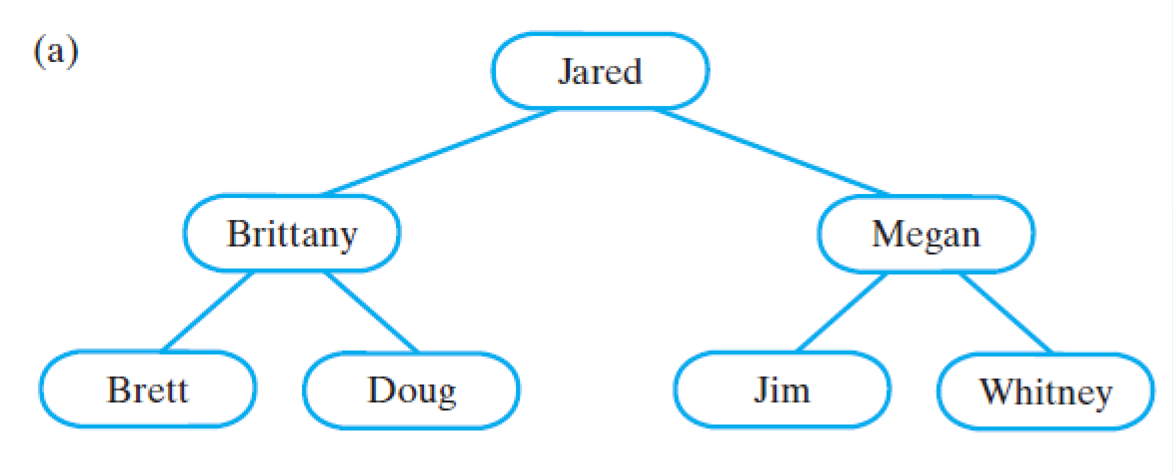
Order in Which Nodes Added

- Adding nodes in sorted order results in tall tree, low efficiency operations



Order in Which Nodes Added

- Add data to binary search tree in random order
  - Expect tree whose operations are O(log *n*).



Implementation of the ADT Dictionary

- Recall interface for a dictionary from Chapter 19, section 4

```java
import java.util.Iterator;
public interface DictionaryInterface<K, V>
{
   public V add(K key, V value);
   public V remove(K key);
   public V getValue(K key);
   public boolean contains(K key);
   public Iterator<K> getKeyIterator();
   public Iterator<V> getValueIterator();
   public boolean isEmpty();
   public int getSize();
   public void clear();
} // end DictionaryInterface
```

Implementation of the ADT Dictionary

- Consider a dictionary implementation that uses balanced search tree to store its entries
  - A class of data objects that will contain both search key and associated value
- Note listing of such a class, Listing 25-3

End

Chapter 25

THIRD EDITION
Data Structures and Abstractions with Java™
FRANK M. CARRANO