A Binary Search Tree Implementation

Chapter 25

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

Binary Search Tree (BST)

- For each node in a binary search tree
  - Node’s data is greater than all data in node’s left subtree
  - Node’s data is less than all data in node’s right subtree
- Every node in a binary search tree is the root of a binary search tree

Binary Search Tree

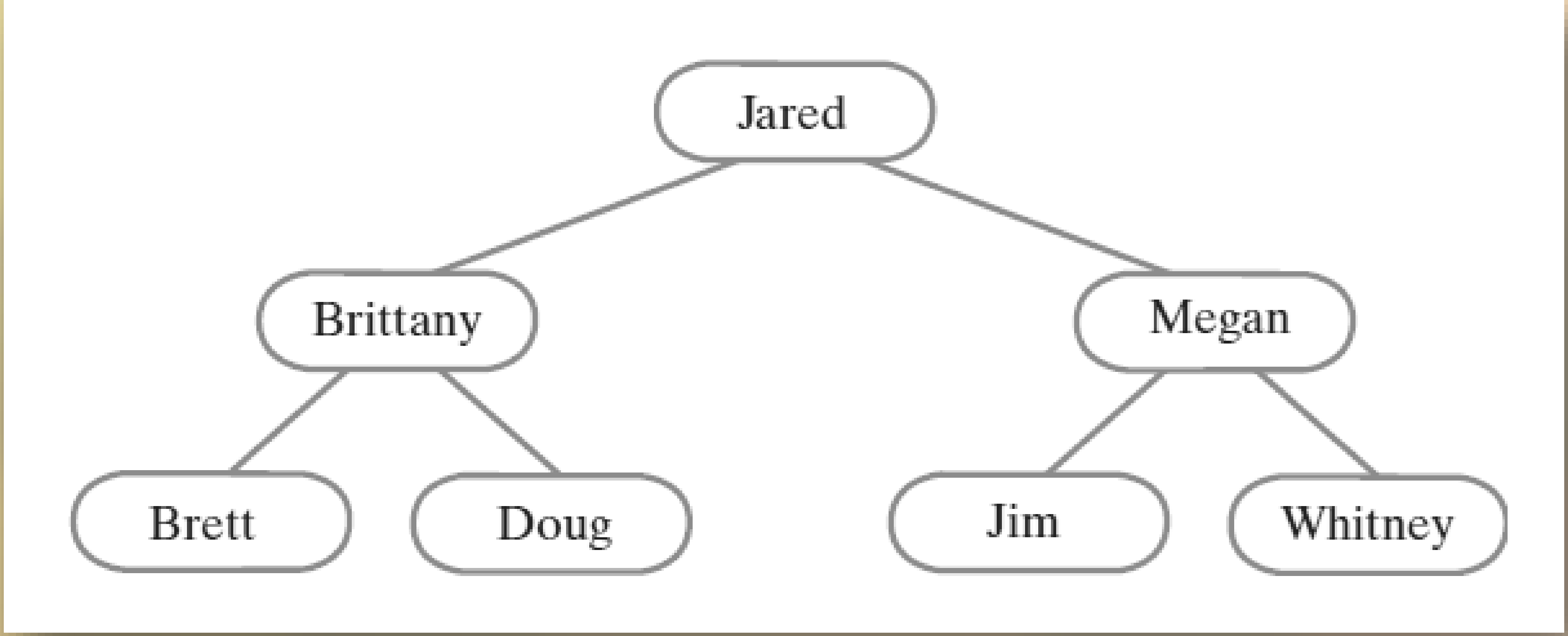

FIGURE 25-1 A binary search tree of names

Interface for the BST

```
package TreePackage;
import java.util.Iterator;
public interface SearchTreeInterface<T extends Comparable<? super T>>

{
   /** Searches for a specific entry in this tree.
       @param entry  An object to be found.
       @return  True if the object was found in the tree. */
   public boolean contains(T entry);

   /** Retrieves a specific entry in this tree.
       @param entry  An object to be found.
       @return  Either the object that was found in the tree or
                null if no such object exists. */
   public T getEntry(T entry);

   /** Adds a new entry to this tree, if it does not match an existing
       object in the tree. Otherwise, replaces the existing object with
       the new entry.
```


LISTING 25-1 An interface for a search tree

Interface for the BST

```
/** Adds a new entry to this tree, if it does not match an existing
    object in the tree. Otherwise, replaces the existing object with
    the new entry.
    @param newEntry  An object to be added to the tree.
    @return  Either null if newEntry was not in the tree already, or
             the existing entry that matched the parameter newEntry
             and has been replaced in the tree. */
public T add(T newEntry);

/** Removes a specific entry from this tree.
    @param entry  An object to be removed.
    @return  Either the object that was removed from the tree or
             null if no such object exists. */
public T remove(T entry);

/** Creates an iterator that traverses all entries in this tree.
    @return  An iterator that provides sequential and ordered access
             to the entries in the tree. */
public Iterator<T> getInorderIterator();
```


LISTING 25-1 An interface for a search tree

Understanding the Specifications

- Methods will use return values instead of exceptions to indicate whether an operation has failed.
- Pay attention to
  - the entry's `equals` method
  - the entry's `compareTo` method

Understanding the Specifications

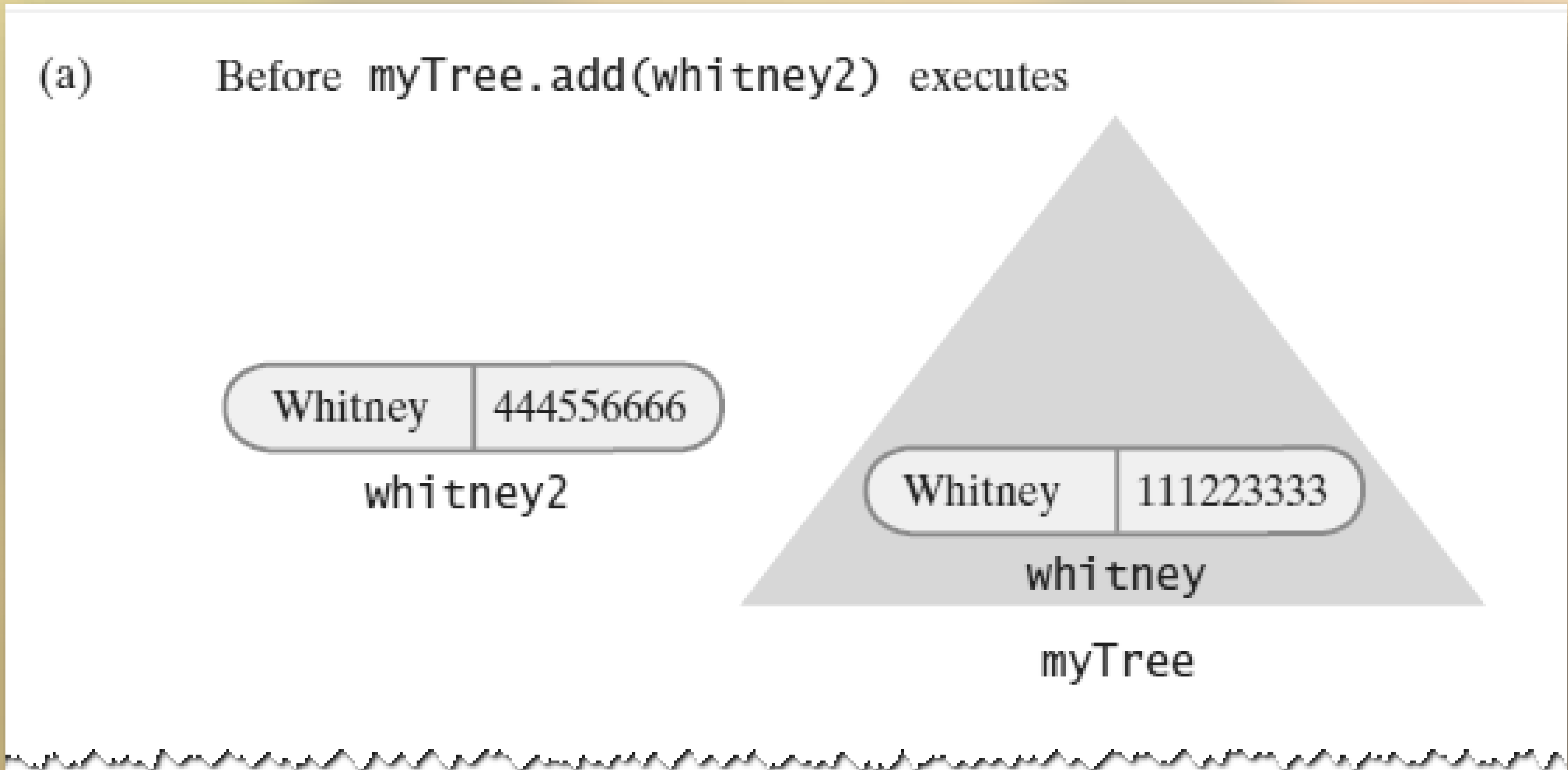

FIGURE 25-2 Adding an entry that matches an entry already in a binary search tree

Understanding the Specifications

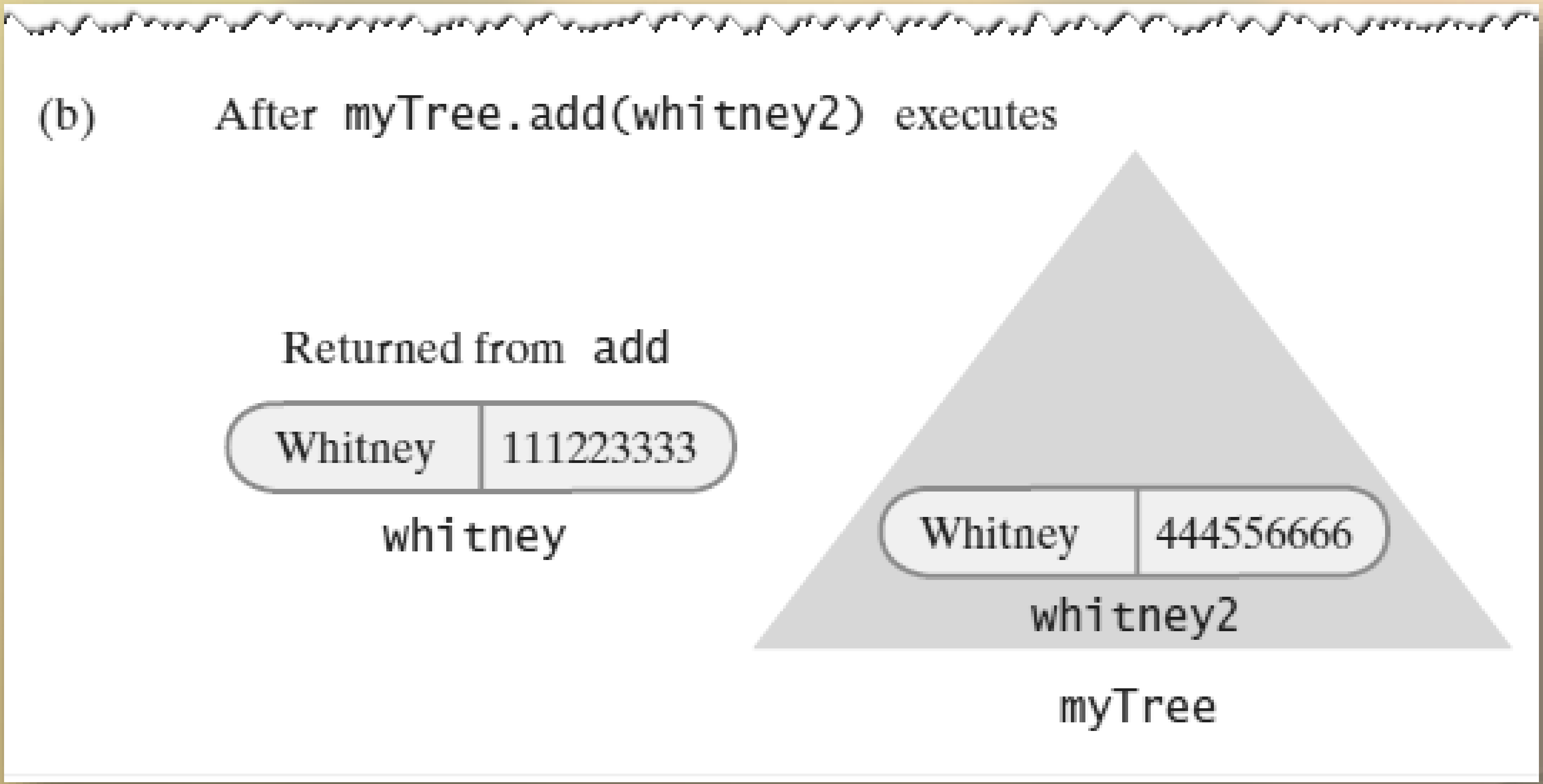

FIGURE 25-2 Adding an entry that matches an entry already in a binary search tree

If Duplicate Entries Are Allowed

- If any entry *e* has a duplicate entry *d*, we require that d occur in the right subtree of *e*'s node
- For each node in a binary search tree:
  - Data in a node is greater than data in node's left subtree
  - Data in a node is less than *or equal to* data in node's right subtree

If Duplicate Entries Are Allowed

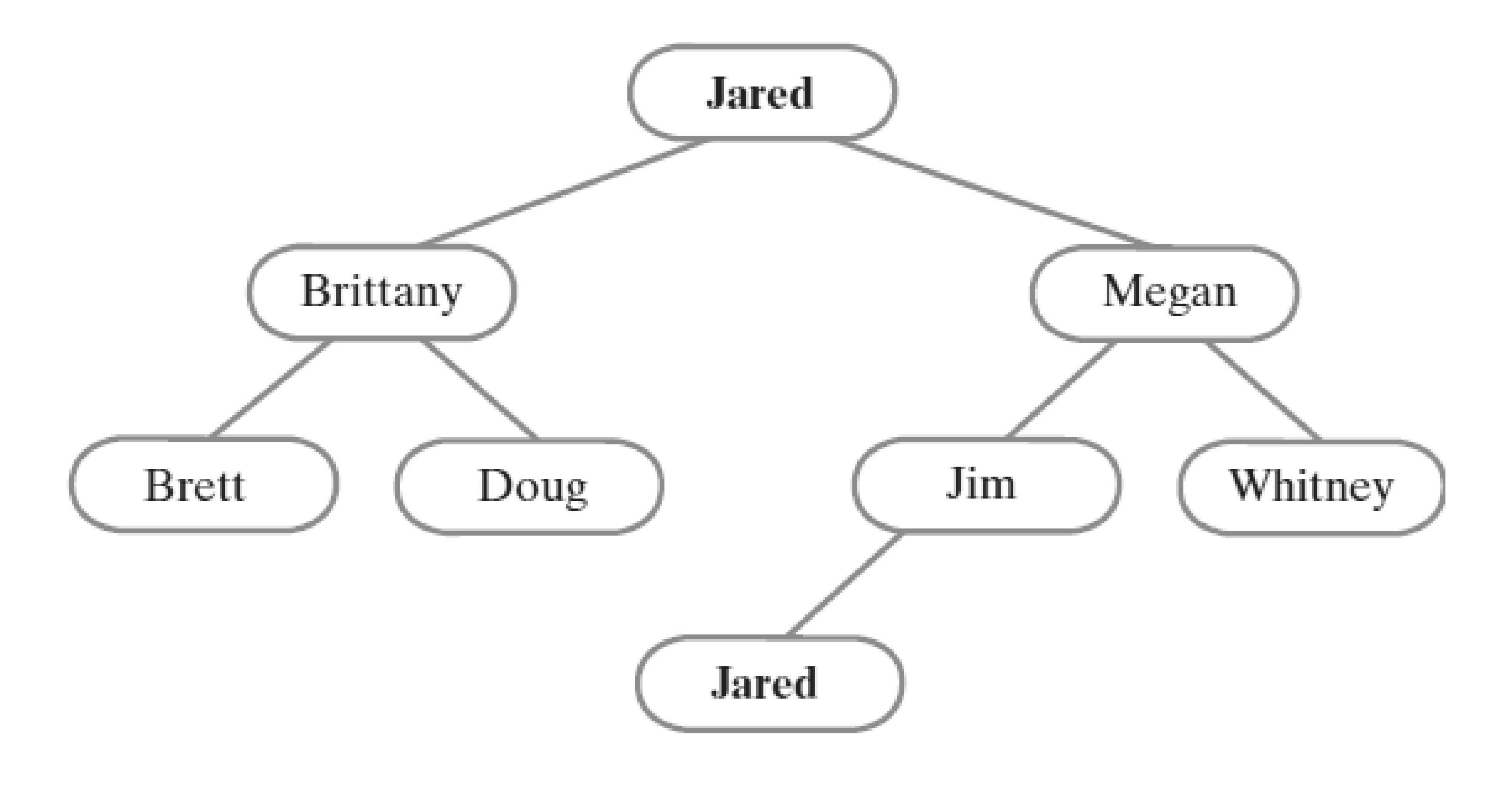

FIGURE 25-3 A binary search tree with duplicate entries

Searching and Retrieving

```
Algorithm bstSearch(binarySearchTreeRoot, desiredObject)
// Searches a binary search tree for a given object.
// Returns true if the object is found.

if (binarySearchTreeRoot is null)
   return false
else if (desiredObject == object in binarySearchTreeRoot)
   return true
else if (desiredObject < object in binarySearchTreeRoot)
   return bstSearch(left child of binarySearchTreeRoot, desiredObject)
else
   return bstSearch(right child of binarySearchTreeRoot, desiredObject)
```

Algorithm that describes actual implementation more closely

```
public T getEntry(T entry)
{
   return findEntry(getRootNode(), entry);
} // end getEntry

private T findEntry(BinaryNode<T> rootNode, T entry)
{
   T result = null;

   if (rootNode != null)
   {
      T rootEntry = rootNode.getData();

      if (entry.equals(rootEntry))
         result = rootEntry;
      else if (entry.compareTo(rootEntry) < 0)
         result = findEntry(rootNode.getLeftChild(), entry);
      else
         result = findEntry(rootNode.getRightChild(), entry);
   } // end if

   return result;
} // end findEntry
```


The method `getEntry` uses `findEntry`

Searching and Retrieving

```
public boolean contains(T entry)
{
   return getEntry(entry) != null;
} // end contains
```

Method `contains` can simply call `getEntry` to see whether a given entry is in the tree:

Adding an Entry

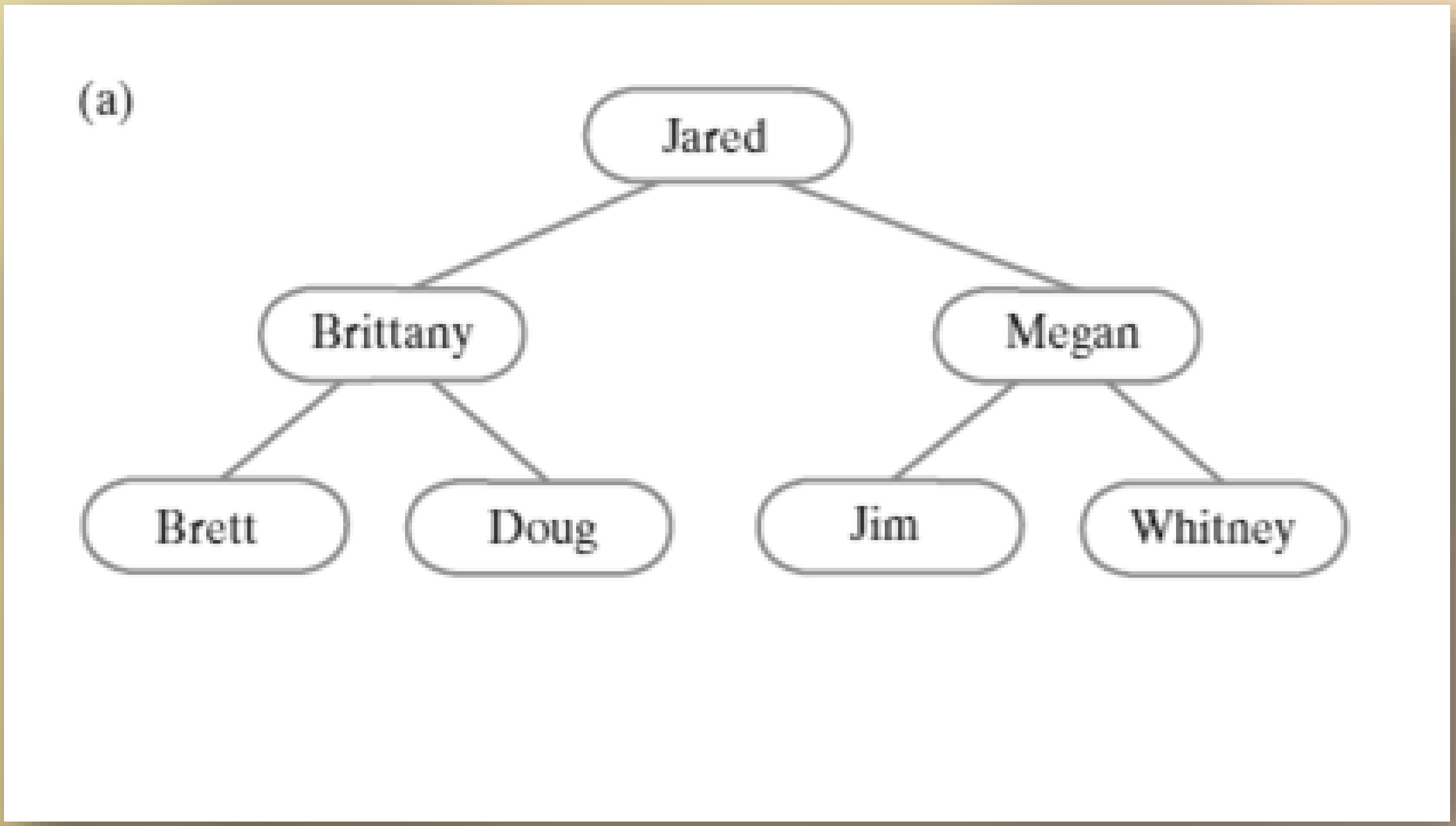

FIGURE 25-4 (a) A binary search tree;

Adding an Entry

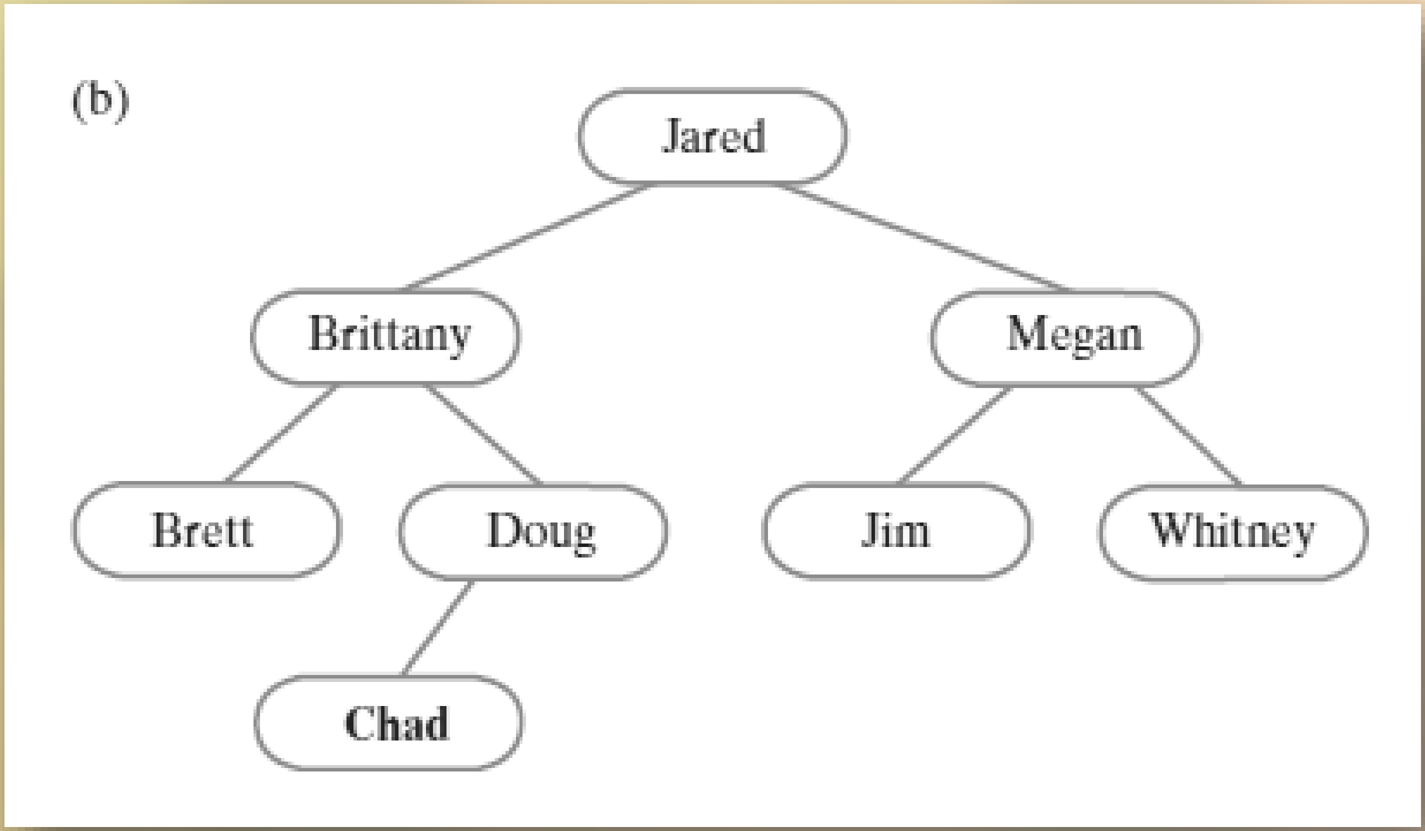

FIGURE 25-4 (b) the same tree after adding *Chad*

Recursive Implementation

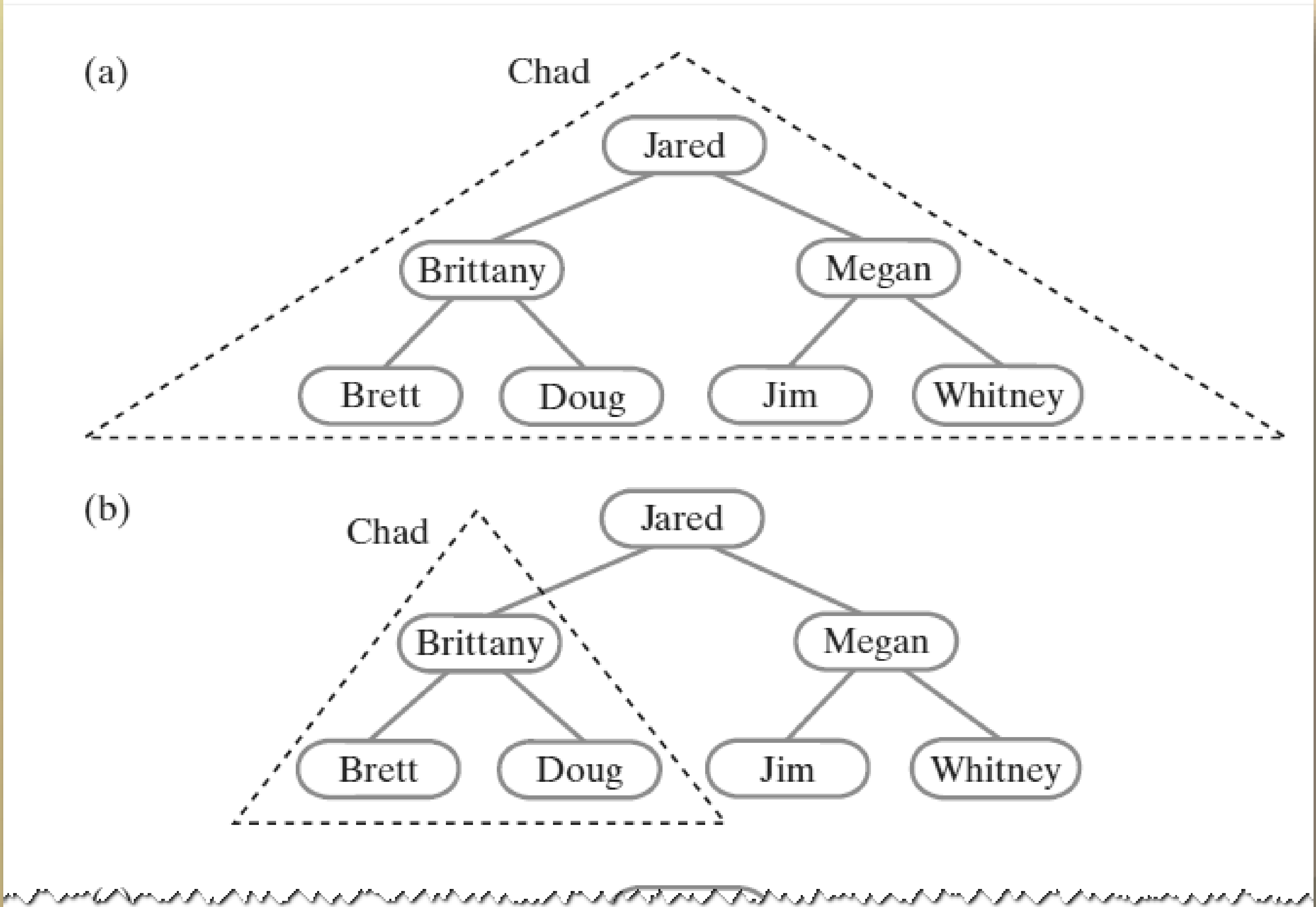

FIGURE 25-5 Recursively adding *Chad* to smaller subtrees of a binary search tree

Recursive Implementation

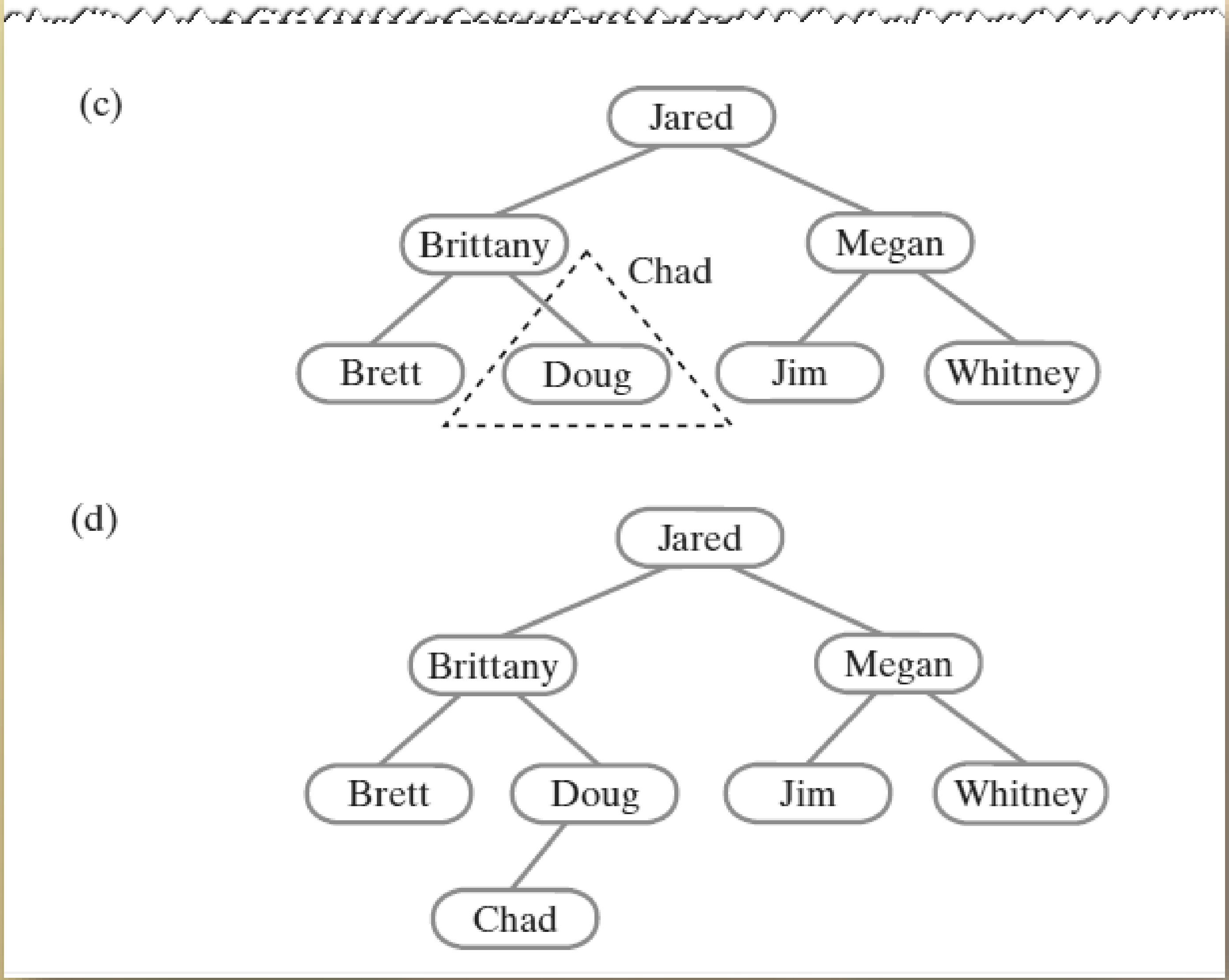

FIGURE 25-5 Recursively adding *Chad* to smaller subtrees of a binary search tree

Recursive Implementation

```
Algorithm add(binarySearchTree, newEntry)
// Adds a new entry to a binary search tree.
// Returns null if newEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by newEntry.

result = null
if (binarySearchTree is empty)
   Create a node containing newEntry and make it the root of binarySearchTree
else
   result = addEntry(binarySearchTree, newEntry)

return result;
```

Handle the addition to an empty binary search tree as a special case

Recursive Implementation

```
Algorithm addEntry(binarySearchTree, newEntry)
// Adds a new entry to a binary search tree that is not empty.
// Returns null if newEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by newEntry.

result = null
if (newEntry matches the entry in the root of binarySearchTree)
{
   result = entry in the root
   Replace entry in the root with newEntry
}
else if (newEntry < entry in the root of binarySearchTree)
{
   if (the root of binarySearchTree has a left child)
      result = addEntry(left subtree of binarySearchTree, newEntry)
   else
      Give the root a left child containing newEntry
}
else // newEntry > entry in the root of binarySearchTree
```

Recursive algorithm for adding a new entry

Recursive Implementation

```
else // newEntry > entry in the root of binarySearchTree
{
    if (the root of binarySearchTree has a right child)
        result = addEntry(right subtree of binarySearchTree, newEntry)
    else
        Give the root a right child containing newEntry
}

return result
```

Recursive algorithm for adding a new entry

Recursive Implementation

```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<>(newEntry));
   else
      result = addEntry(getRootNode(), newEntry);

   return result;
} // end add
```


The public method **add**

Recursive Implementation

```
// Adds newEntry to the nonempty subtree rooted at rootNode.
private T addEntry(BinaryNode<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());

   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())
         result = addEntry(rootNode.getLeftChild(), newEntry);
      else
         rootNode.setLeftChild(new BinaryNode<>(newEntry));
   }
```

Private recursive method **`addEntry`**

Recursive Implementation

```
    else
    {
       assert comparison > 0;

       if (rootNode.hasRightChild())
          result = addEntry(rootNode.getRightChild(), newEntry);
       else
          rootNode.setRightChild(new BinaryNode<>(newEntry));
    } // end if

    return result;
} // end addEntry
```

Private recursive method `addEntry`

Iterative Implementation

```
Algorithm addEntry(binarySearchTree, newEntry)
// Adds a new entry to a binary search tree that is not empty.
// Returns null if newEntry did not exist already in the tree. Otherwise, returns
// tree entry that matched and was replaced by newEntry.

result = null
currentNode = root node of binarySearchTree
found = false

while (found is false)
{
    if (newEntry matches the entry in currentNode)
    {
        found = true
        result = entry in currentNode
        Replace entry in currentNode with newEntry
    }
```


Iterative algorithm for adding a new entry

Iterative Implementation

```
}
else if (newEntry < entry in currentNode)
{
   if (currentNode has a left child)
      currentNode = left child of currentNode
   else
   {
      found = true
      Give currentNode a left child containing newEntry
   }
}
else // newEntry > entry in currentNode
{
```

Iterative algorithm for adding a new entry

Iterative Implementation

```
    else // newEntry > entry in currentNode
    {
        if (currentNode has a right child)
            currentNode = right child of currentNode
        else
        {
            found = true
            Give currentNode a right child containing newEntry
        }
    }
}

return result
```


Iterative algorithm for adding a new entry

Iterative Implementation

```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
```


Iterative implementation of the method `addEntry`

Iterative Implementation

```
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```


Iterative implementation of the method **addEntry**

Removing an Entry Whose Node Is a Leaf

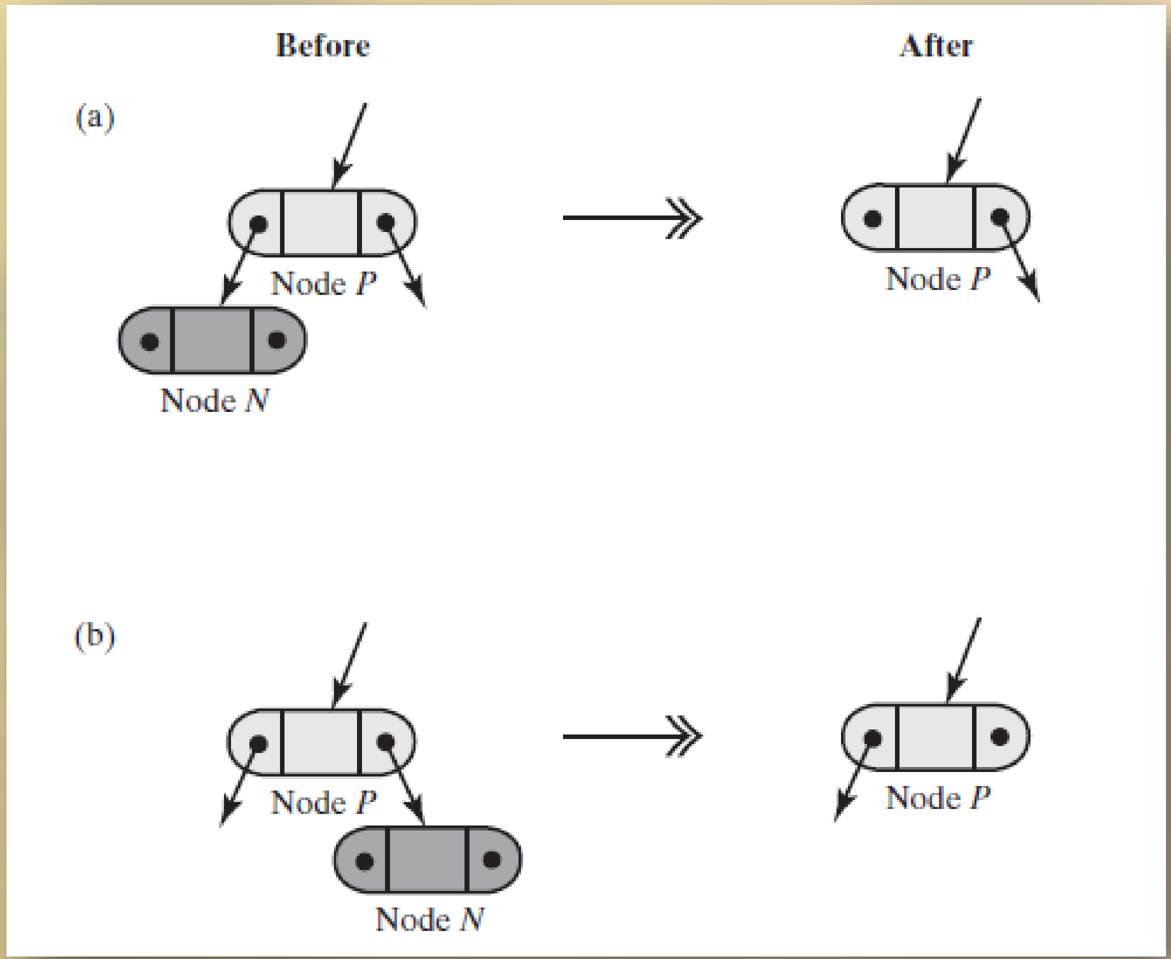

FIGURE 25-6 (a) Two possible configurations of a leaf node N; (b) the resulting two possible configurations after removing node

Removing an Entry Whose Node Has One Child

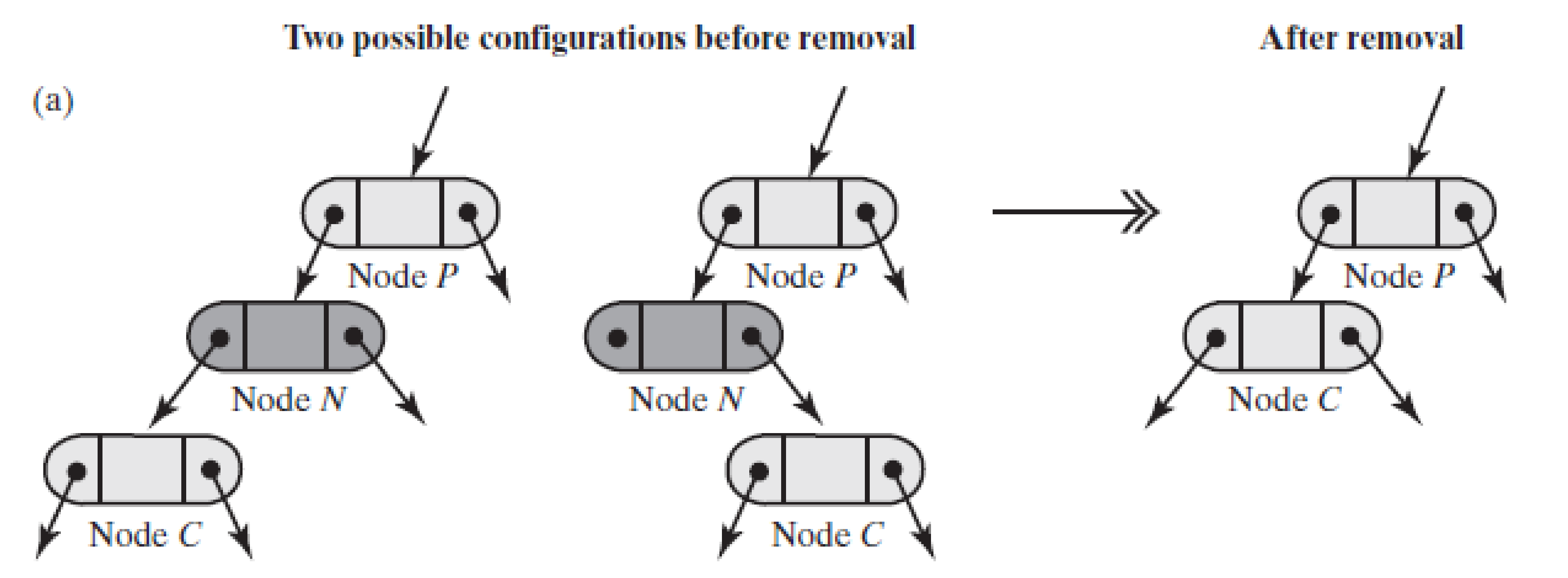

FIGURE 25-7 Removing a node N from its parent node P when N has one child and is itself (a) a left child

Removing an Entry Whose Node Has One Child

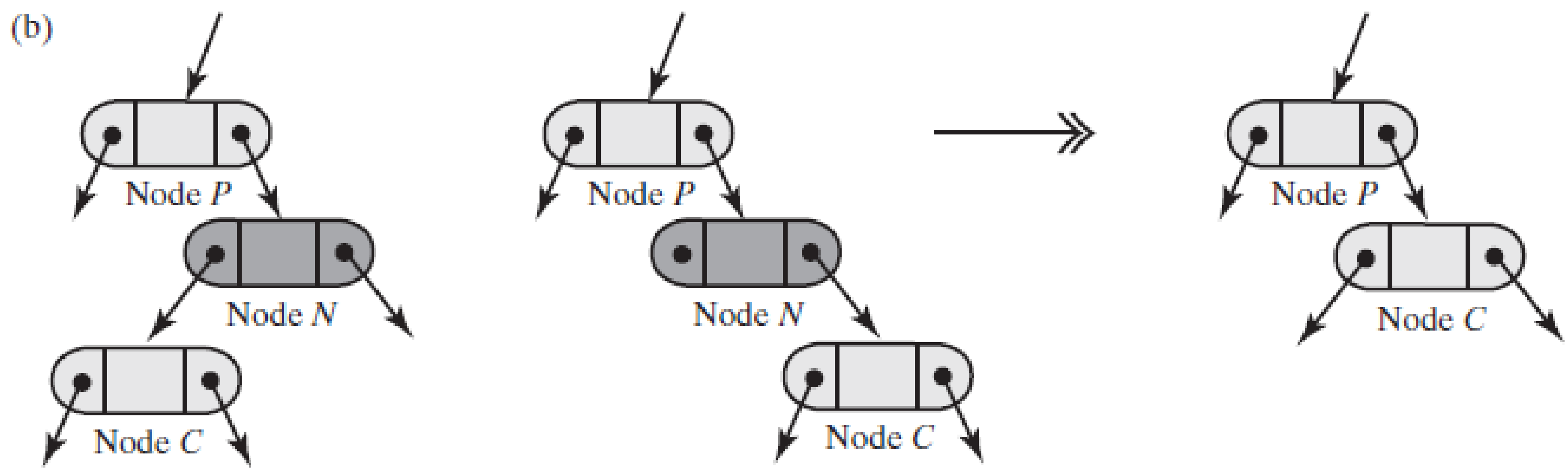

FIGURE 25-7 Removing a node N from its parent node P when N has one child and is itself (b) a right child

Removing an Entry Whose Node Has Two Children

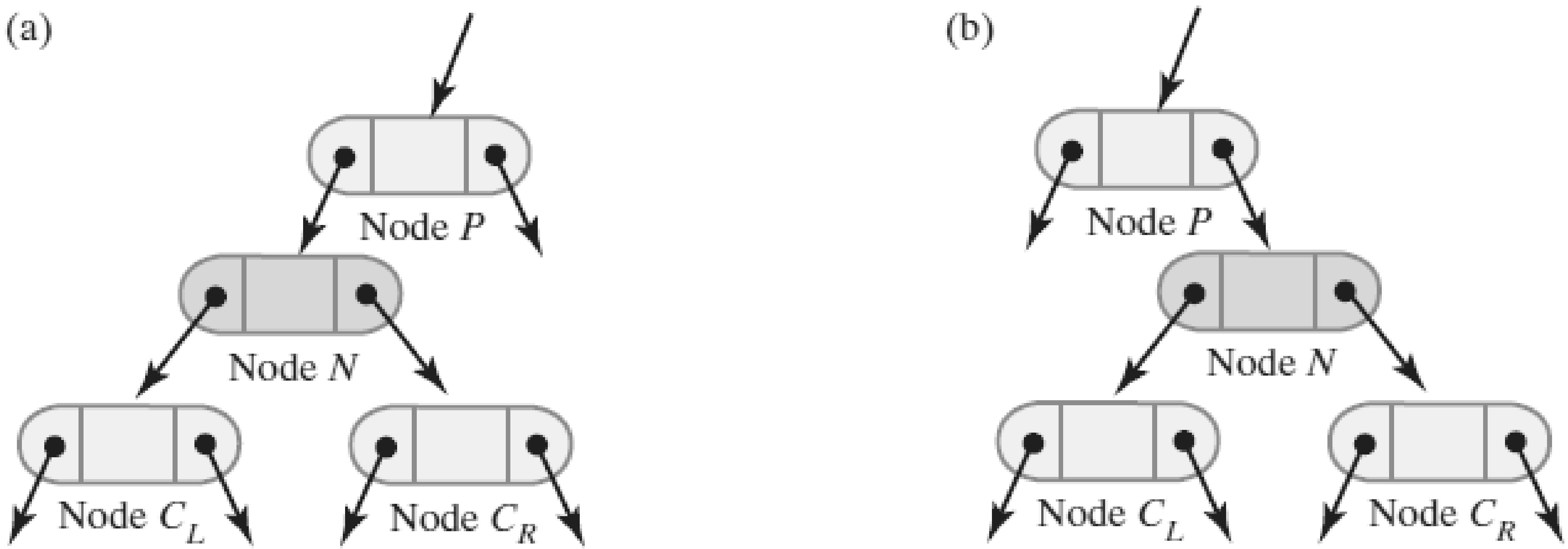

FIGURE 25-8 Two possible configurations of a node N that has two children

Removing an Entry Whose Node Has Two Children

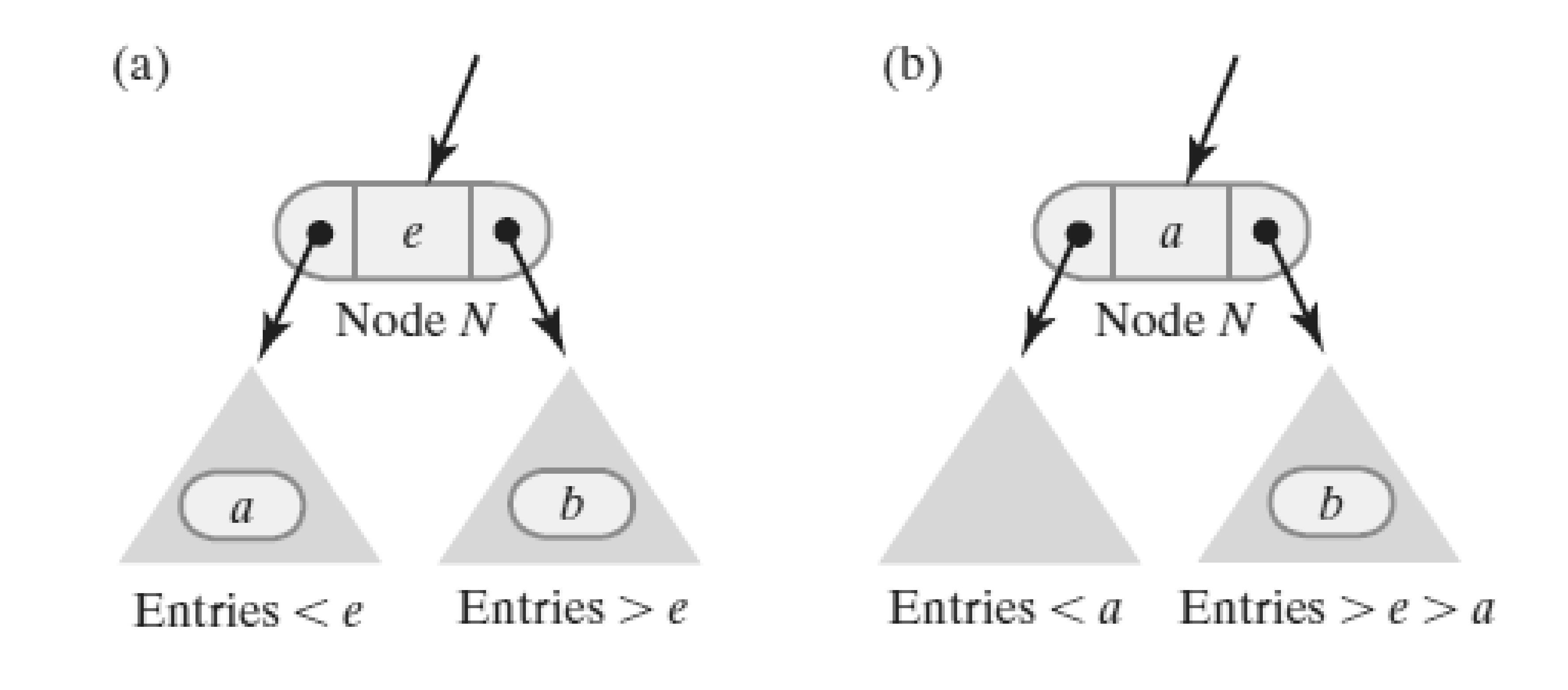

FIGURE 25-9 Node N and its subtrees: (a) the entry *a* is immediately before the entry *e*, and *b* is immediately after *e*; (b) after deleting the node that contained *a* and replacing *e* with *a*

Removing an Entry Whose Node Has Two Children

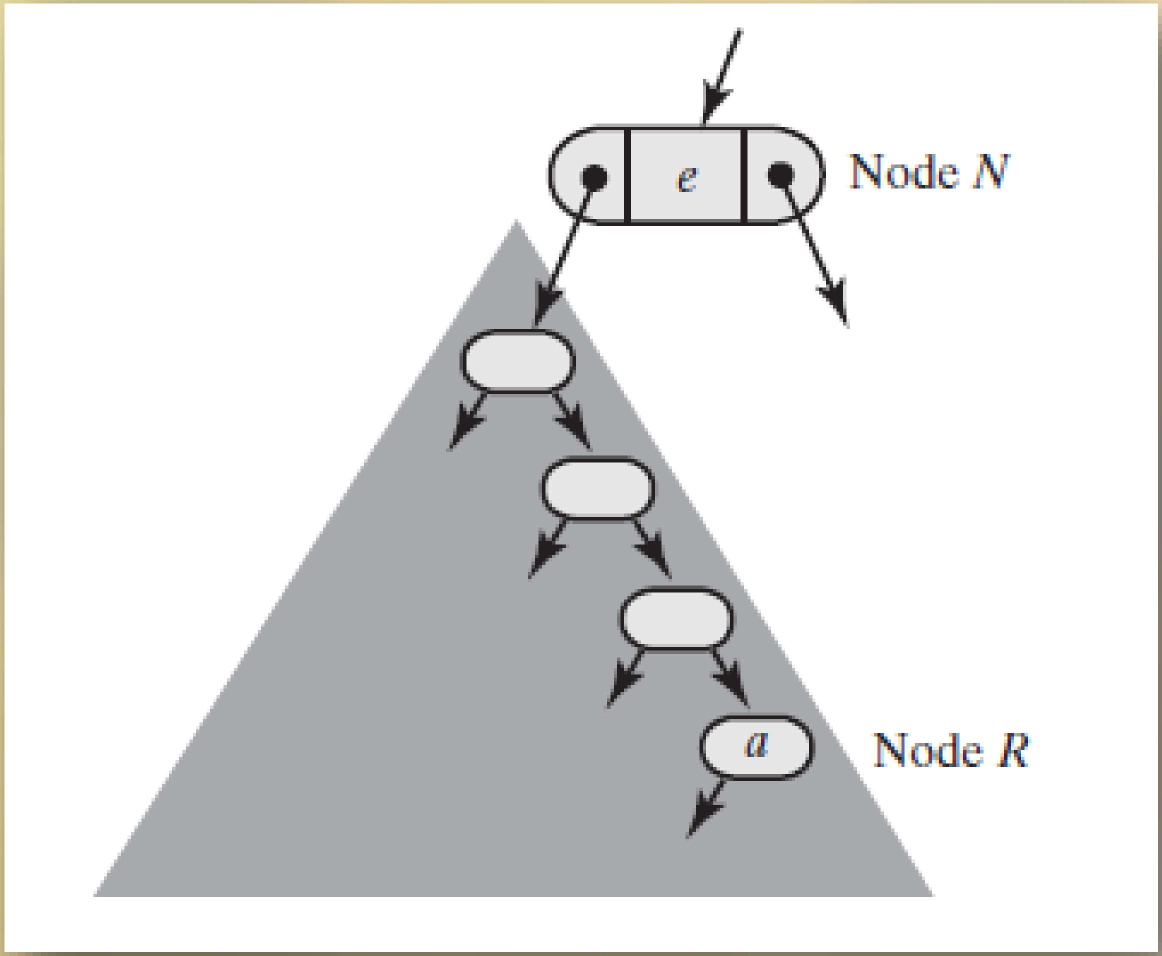

FIGURE 25-10 The largest entry *a* in node *N*'s left subtree occurs in the subtree's rightmost node *R*

Removing an Entry Whose Node Has Two Children

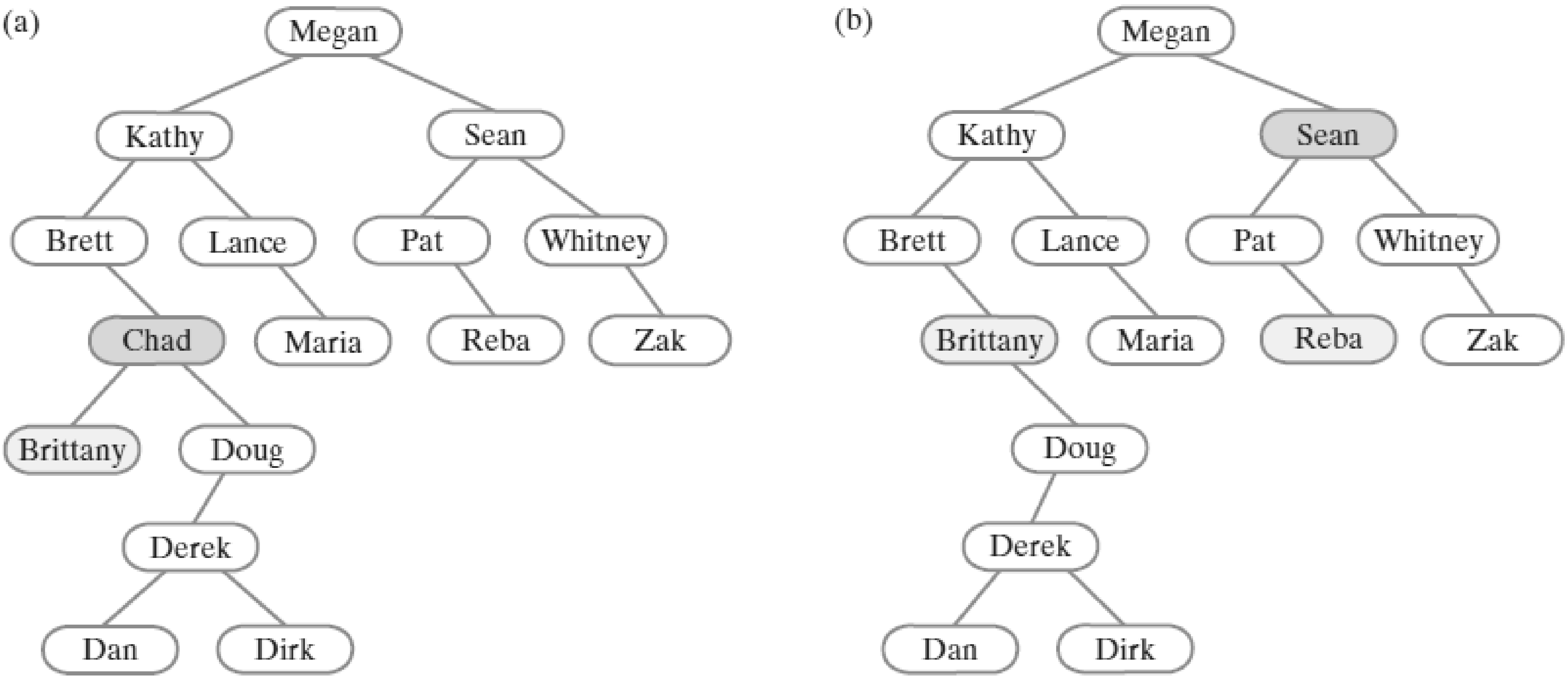

FIGURE 25-11 (a) A binary search tree; (b) after removing *Chad*

Removing an Entry Whose Node Has Two Children

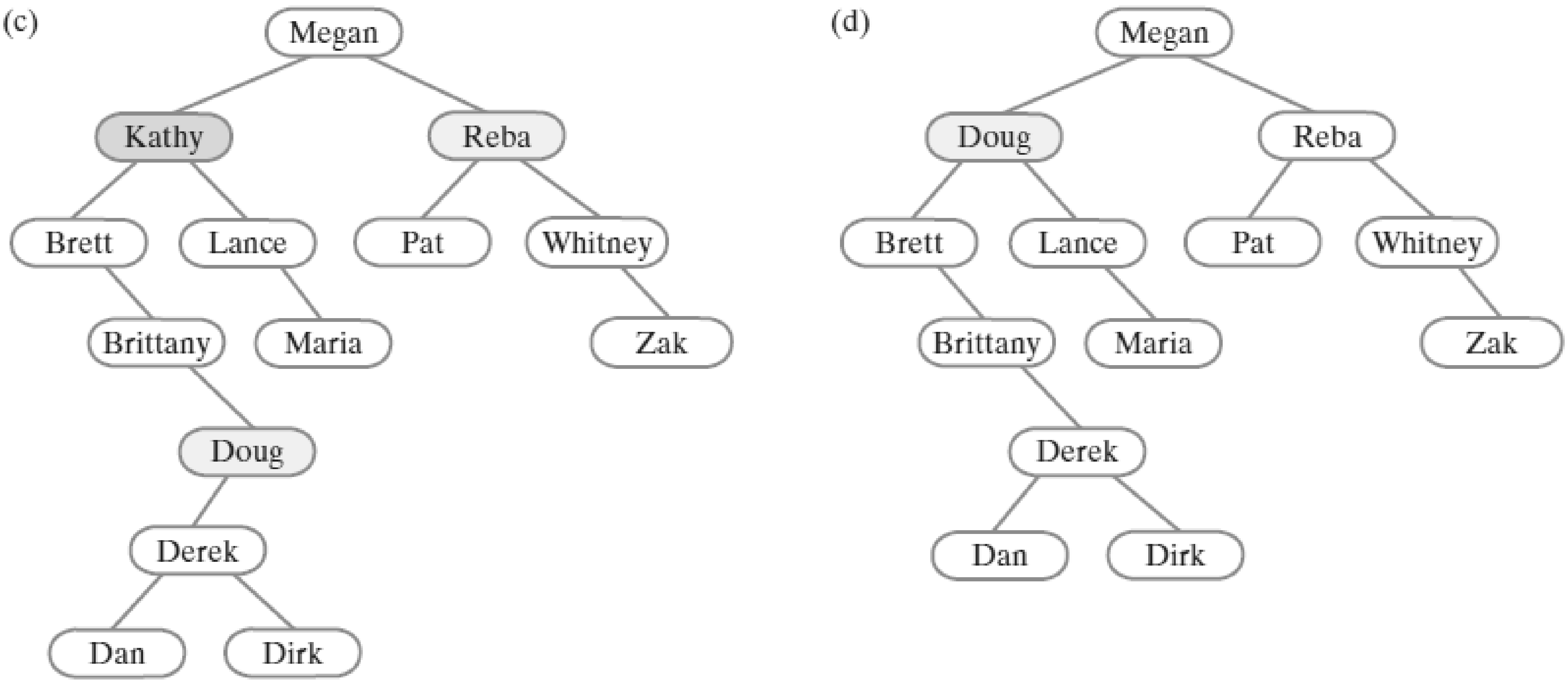

FIGURE 25-11 (c) after removing *Sean*; (d) after removing *Kathy*

Removing an Entry in the Root

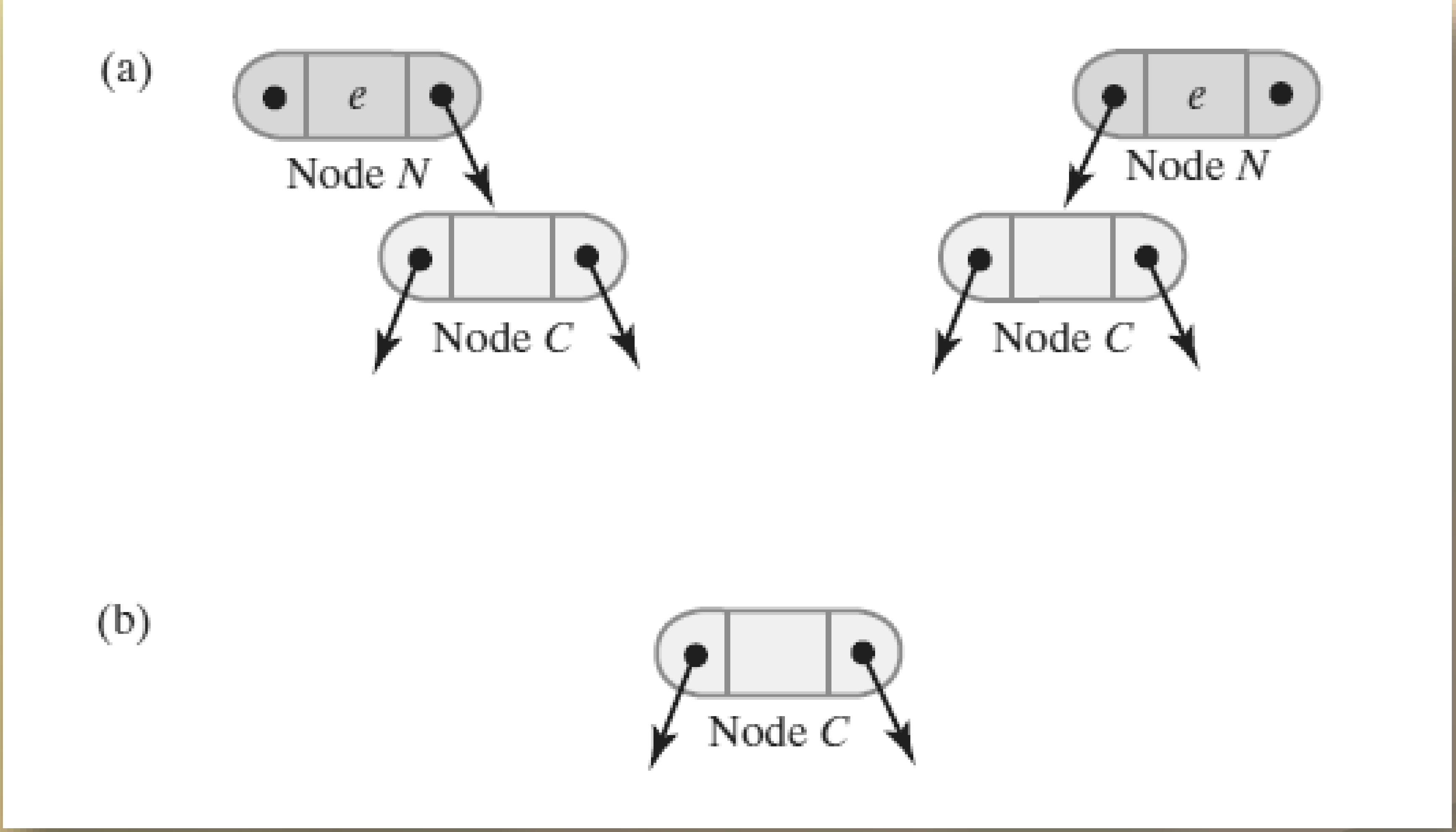

FIGURE 25-12 (a) Two possible configurations of a root that has one child; (b) after removing the root

Recursive Implementation

```
Algorithm remove(binarySearchTree, entry)
oldEntry = null
if (binarySearchTree is not empty)
{
   if (entry matches the entry in the root of binarySearchTree)
   {
      oldEntry = entry in root
      removeFromRoot(root of binarySearchTree)
   }
   else if (entry < entry in root)
      oldEntry = remove(left subtree of binarySearchTree, entry)
   else  // entry > entry in root
      oldEntry = remove(right subtree of binarySearchTree, entry)
}
return oldEntry
```

Recursive algorithm describes the method's logic at a high level

Recursive Implementation

```
// Removes an entry from the tree rooted at a given node.
// rootNode is a reference to the root of a tree.
// entry is the object to be removed.
// oldEntry is an object whose data field is null.
// Returns the root node of the resulting tree; if entry matches
//         an entry in the tree, oldEntry's data field is the entry
//         that was removed from the tree; otherwise it is null.
private BinaryNode<T> removeEntry(BinaryNode<T> rootNode, T entry,
                                  ReturnObject oldEntry)
{
   if (rootNode != null)
   {
      T rootData = rootNode.getData();
      int comparison = entry.compareTo(rootData);

      if (comparison == 0)      // entry == root entry
      {
         oldEntry.set(rootData);
         rootNode = removeFromRoot(rootNode);
      }
      else if (comparison < 0) // entry < root entry
      {
```


The private method **removeEntry**

Recursive Implementation

```
      else if (comparison < 0) // entry < root entry
      {
         BinaryNode<T> leftChild = rootNode.getLeftChild();
         BinaryNode<T> subtreeRoot = removeEntry(leftChild, entry, oldEntry);
         rootNode.setLeftChild(subtreeRoot);
      }
      else                     // entry > root entry
      {
         BinaryNode<T> rightChild = rootNode.getRightChild();
         rootNode.setRightChild(removeEntry(rightChild, entry, oldEntry));
      } // end if
   } // end if
   return rootNode;
} // end removeEntry
```


The private method **removeEntry**

Recursive Implementation

```
Algorithm removeFromRoot(rootNode)
// Removes the entry in a given root node of a subtree.

if (rootNode has two children)
{
    largestNode = node with the largest entry in the left subtree of rootNode
    Replace the entry in rootNode with the entry in largestNode
    Remove largestNode from the tree
}
else if (rootNode has a right child)
    rootNode = rootNode's right child
else
    rootNode = rootNode's left child // Possibly null
// Assertion: If rootNode was a leaf, it is now null

return rootNode
```

The algorithm **removeFromRoot**

Recursive Implementation

```
// Removes the entry in a given root node of a subtree.
// rootNode is the root node of the subtree.
// Returns the root node of the revised subtree.
private BinaryNode<T> removeFromRoot(BinaryNode<T> rootNode)
{
   // Case 1: rootNode has two children
   if (rootNode.hasLeftChild() && rootNode.hasRightChild())
   {
      // Find node with largest entry in left subtree
      BinaryNode<T> leftSubtreeRoot = rootNode.getLeftChild();
      BinaryNode<T> largestNode = findLargest(leftSubtreeRoot);

      // Replace entry in root
      rootNode.setData(largestNode.getData());

      // Remove node with largest entry in left subtree
      rootNode.setLeftChild(removeLargest(leftSubtreeRoot));
```


The private method `removeFromRoot`

Recursive Implementation

```
        rootNode.setData(largestNode.getData());

        // Remove node with largest entry in left subtree
        rootNode.setLeftChild(removeLargest(leftSubtreeRoot));
    } // end if

    // Case 2: rootNode has at most one child
    else if (rootNode.hasRightChild())
        rootNode = rootNode.getRightChild();
    else
        rootNode = rootNode.getLeftChild();

    // Assertion: If rootNode was a leaf, it is now null

    return rootNode;
} // end removeEntry
```

The private method **removeFromRoot**

Recursive Implementation

```
// Finds the node containing the largest entry in a given tree.
// rootNode is the root node of the tree.
// Returns the node containing the largest entry in the tree.
private BinaryNode<T> findLargest(BinaryNode<T> rootNode)
{
   if (rootNode.hasRightChild())
      rootNode = findLargest(rootNode.getRightChild());

   return rootNode;
} // end findLargest
```

The private method `findLargest`

Recursive Implementation

```
// Removes the node containing the largest entry in a given tree.
// rootNode is the root node of the tree.
// Returns the root node of the revised tree.
private BinaryNode<T> removeLargest(BinaryNode<T> rootNode)
{
   if (rootNode.hasRightChild())
   {
      BinaryNode<T> rightChild = rootNode.getRightChild();
      rightChild = removeLargest(rightChild);
      rootNode.setRightChild(rightChild);
   }
   else
      rootNode = rootNode.getLeftChild();

   return rootNode;
} // end removeLargest
```

The private method **removeLargest**

Iterative Implementation

```
Algorithm remove(entry)
result = null
currentNode = node that contains a match for entry
parentNode = currentNode's parent

if (currentNode != null) // That is, if entry is found
{
   result = currentNode's data (the entry to be removed from the tree)

   // Case 1
   if (currentNode has two children)
   {
      // Get node to remove and its parent
      nodeToRemove = node containing entry's inorder predecessor; it has at most one child
      parentNode = nodeToRemove's parent

      Copy entry from nodeToRemove to currentNode
      currentNode = nodeToRemove
      // Assertion: currentNode is the node to be removed; it has at most one child
      // Assertion: Case 1 has been transformed to Case 2
   }

   // Case 2: currentNode has at most one child
   Delete currentNode from the tree
}

return result
```


Pseudocode that describes what `remove` must do:

Iterative Implementation

```java
public T remove(T entry)
{
   T result = null;

   // Locate node (and its parent) that contains a match for entry
   NodePair pair = findNode(entry);
   BinaryNode<T> currentNode = pair.getFirst();
   BinaryNode<T> parentNode = pair.getSecond();

   if (currentNode != null) // Entry is found
   {
      result = currentNode.getData(); // Get entry to be removed

      // Case 1: currentNode has two children
      if (currentNode.hasLeftChild() && currentNode.hasRightChild())
      {

         // Replace entry in currentNode with the entry in another node
         // that has at most one child; that node can be deleted
```


The public method `remove`

Iterative Implementation

```
        // Get node to remove (contains inorder predecessor; has at
        // most one child) and its parent
        pair = getNodeToRemove(currentNode);
        BinaryNode<T> nodeToRemove = pair.getFirst();
        parentNode = pair.getSecond();

        // Copy entry from nodeToRemove to currentNode
        currentNode.setData(nodeToRemove.getData());

        currentNode = nodeToRemove;
        // Assertion: currentNode is the node to be removed; it has at
        //            most one child
        // Assertion: Case 1 has been transformed to Case 2
     } // end if

     // Case 2: currentNode has at most one child; delete it
     removeNode(currentNode, parentNode);
  } // end if

  return result;
} // end remove
```

The public method `remove`

Iterative Implementation

```
private NodePair findNode(T entry)
{
   NodePair result = new NodePair();
   boolean found = false;
   . . .
   if (found)
      result = new NodePair(currentNode, parentNode);
      // Located entry is currentNode.getData()

   return result;
} // end findNode
```

The private method **`findNode`**

Iterative Implementation

```
// Find the inorder predecessor by searching the left subtree; it will be the largest
// entry in the subtree, occurring in the node as far right as possible
leftSubtreeRoot = left child of currentNode
rightChild = leftSubtreeRoot
priorNode = currentNode
while (rightChild has a right child )
{
   priorNode = rightChild
   rightChild = right child of rightChild
}
// Assertion: rightChild is the node to be removed and has no more than one child
```

Pseudocode for the private method **`getNodeToRemove`**

Iterative Implementation

```
private NodePair getNodeToRemove(BinaryNode<T> currentNode)
{
   // Find node with largest entry in left subtree by
   // moving as far right in the subtree as possible
   BinaryNode<T> leftSubtreeRoot = currentNode.getLeftChild();
   BinaryNode<T> rightChild = leftSubtreeRoot;
   BinaryNode<T> priorNode = currentNode;

   while (rightChild.hasRightChild())
   {
      priorNode = rightChild;
      rightChild = rightChild.getRightChild();
   } // end while

   // rightChild contains the inorder predecessor and is the node to
   // remove; priorNode is its parent

   return new NodePair(rightChild, priorNode);
} // end getNodeToRemove
```


Implementation of the private method `getNodeToRemove`

Iterative Implementation

```
private void removeNode(BinaryNode<T> nodeToRemove, BinaryNode<T> parentNode)
{
   BinaryNode<T> childNode;

   if (nodeToRemove.hasLeftChild())
      childNode = nodeToRemove.getLeftChild();
   else
      childNode = nodeToRemove.getRightChild();

   // Assertion: If nodeToRemove is a leaf, childNode is null
   assert (nodeToRemove.isLeaf() && childNode == null) ||
          !nodeToRemove.isLeaf();

   if (nodeToRemove == getRootNode())
      setRootNode(childNode);
   else if (parentNode.getLeftChild() == nodeToRemove)
      parentNode.setLeftChild(childNode);
   else
      parentNode.setRightChild(childNode);
} // end removeNode
```


The private method **removeNode**

Efficiency of Operations

- For tree of height *h*
  - The operations `add`, `remove`, and `getEntry` are O(h)
- If tree of *n* nodes has height *h* = *n*
  - These operations are O(n)
- Shortest tree is full
  - Results in these operations being O(log n)

Efficiency of Operations

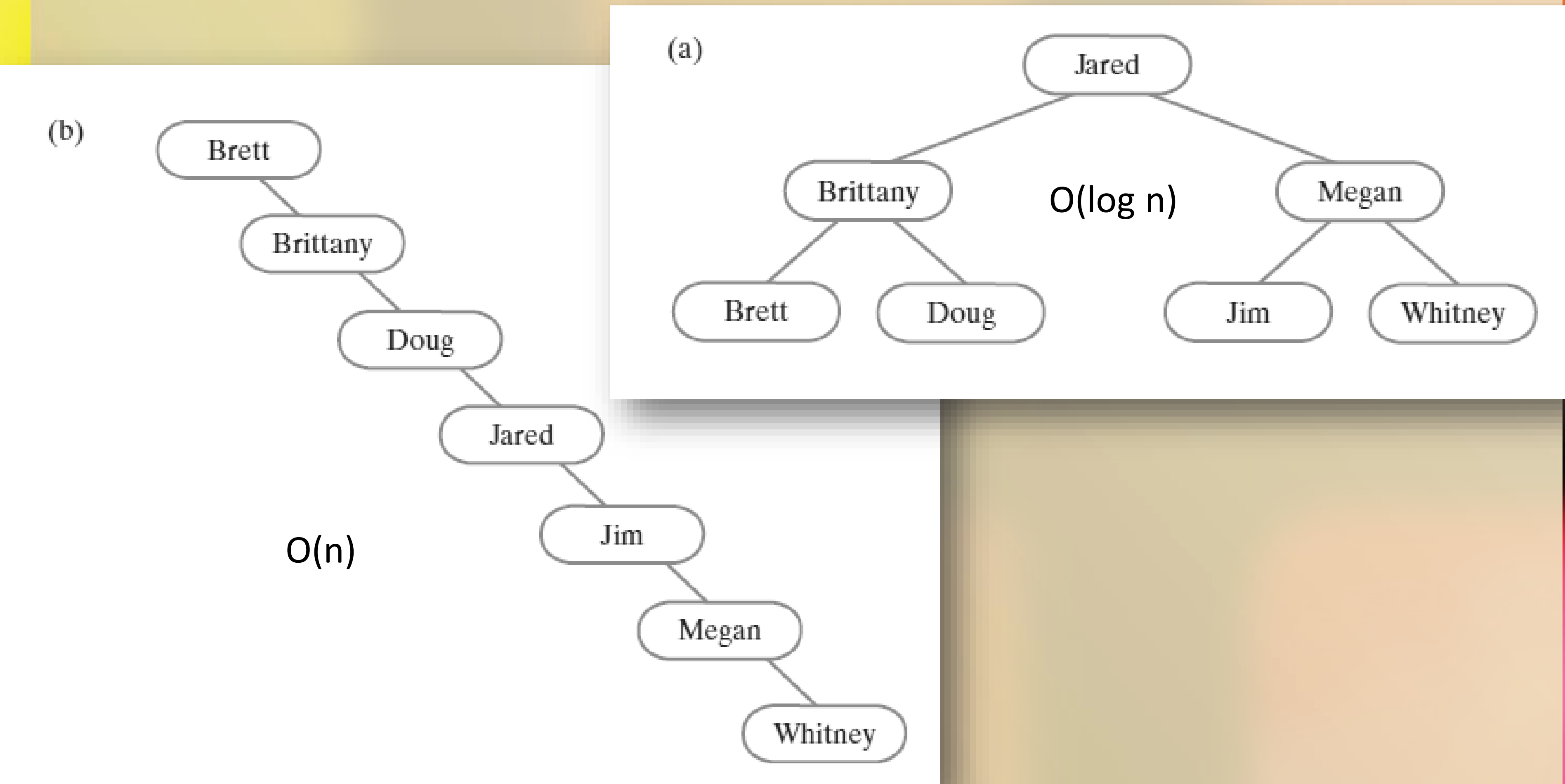

FIGURE 25-13 Two binary search trees that contain the same data

The Importance of Balance

- Do not need a full binary search tree to get O(log n) performance
- Complete tree will also give us O(log n) performance.

Order in Which Nodes Are Added

- Order in which you add entries to a binary search tree affects the shape of the tree
- If you add entries into an initially empty binary search tree, do not add them in sorted order.

End

Chapter 25