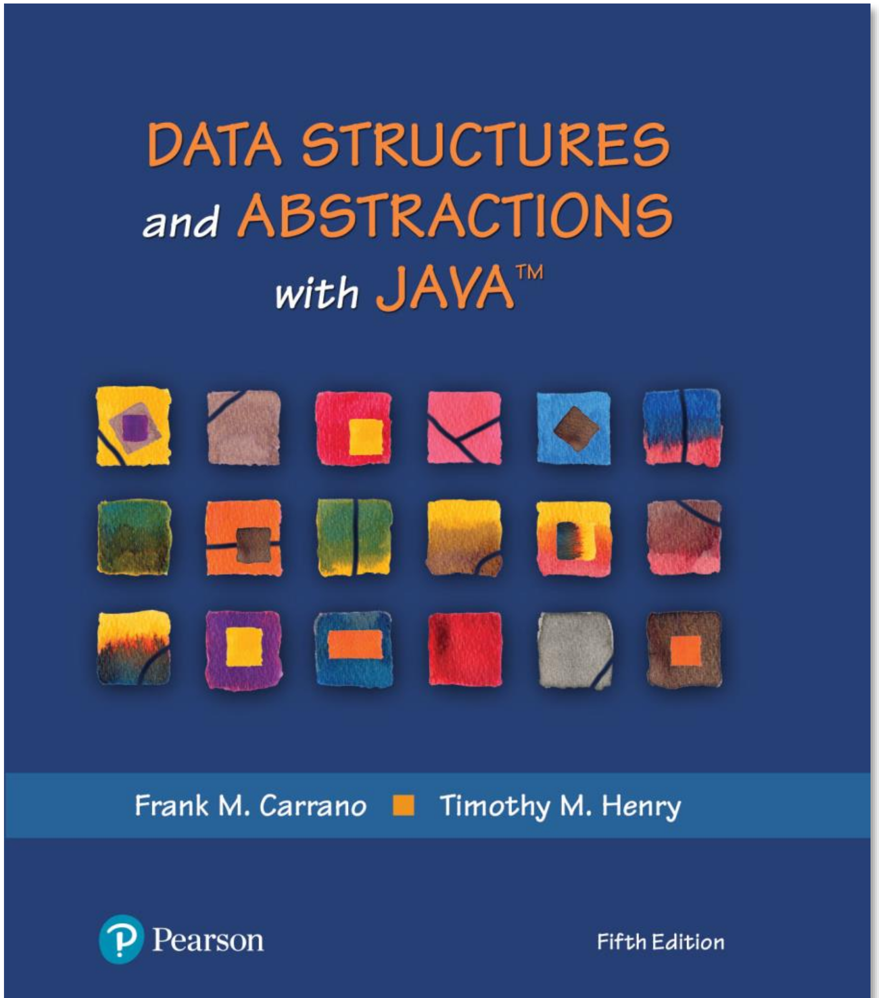
Data Structures and Abstractions with Java™

5th Edition



Chapter 26

A Binary Search Tree Implementation

Binary Search Tree

- 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

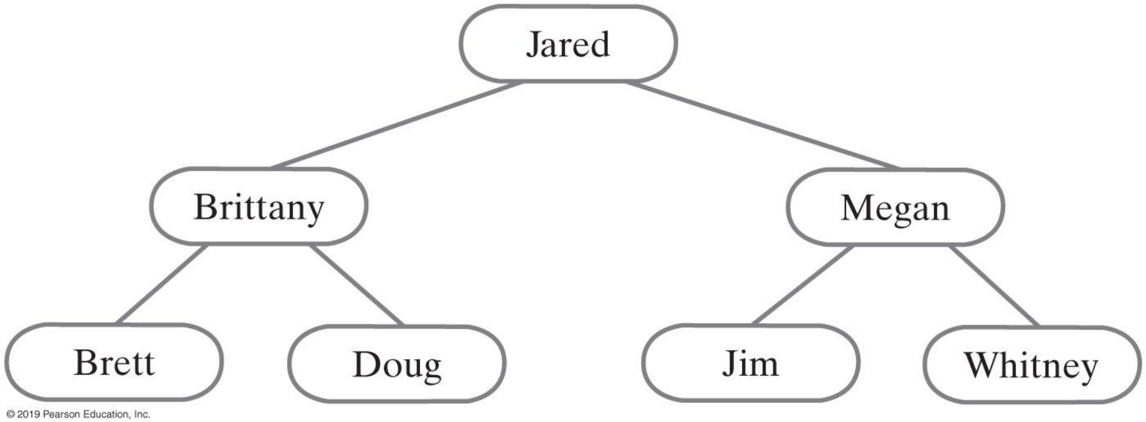


FIGURE 26-1 A binary search tree of names

Interface for the Binary Search Tree (Part 1)

```java
/** An interface for a search tree. */
public interface SearchTreeInterface<T extends Comparable<? super T>>
     extends TreeInterface<T>
{
  /** Searches for a specific entry in this tree.
     @param anEntry  An object to be found.
     @return  True if the object was found in the tree. */
  public boolean contains(T anEntry);

  /** Retrieves a specific entry in this tree.
     @param anEntry  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 anEntry);

  /** 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 anEntry  An object to be added to the tree.
     @return  Either null if anEntry was not in the tree but has been added, or
           the existing entry that matched the parameter anEntry
           and has been replaced in the tree. */
  public T add(T anEntry);
```

LISTING 26-1 An interface for a search tree

Interface for the Binary Search Tree (Part 2)

```
  /** Removes a specific entry from this tree.
      @param anEntry  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 anEntry);

  /** 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();
} // end SearchTreeInterface
```

LISTING 26-1 An interface for a search tree

Understanding the Specifications

- Methods will use return values instead of exceptions to indicate whether an operation has failed
  - `getEntry`, returns same entry it is given to find
  - `getEntry` returns an object in tree and matches given entry according to the entry's `compareTo` method

Adding to a Binary Search Tree

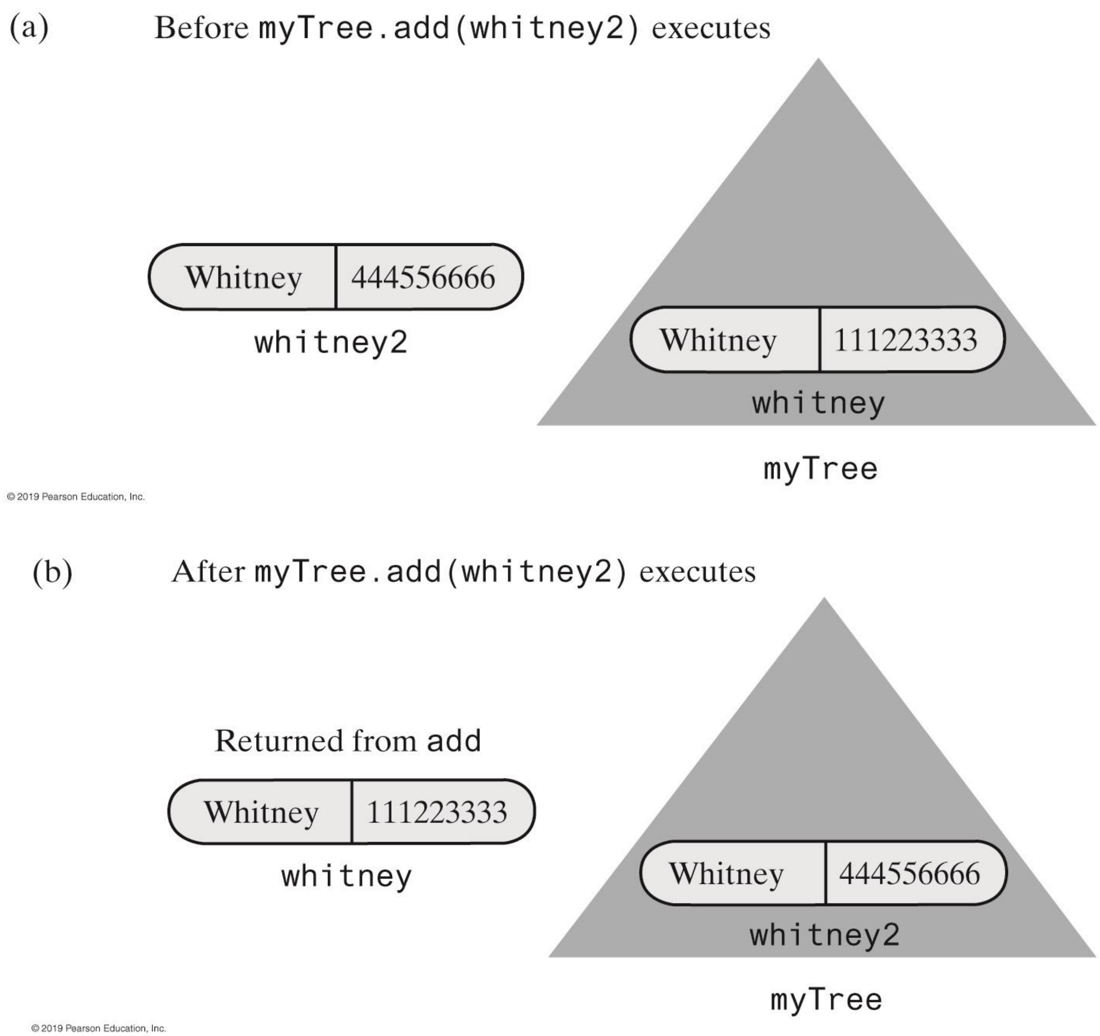
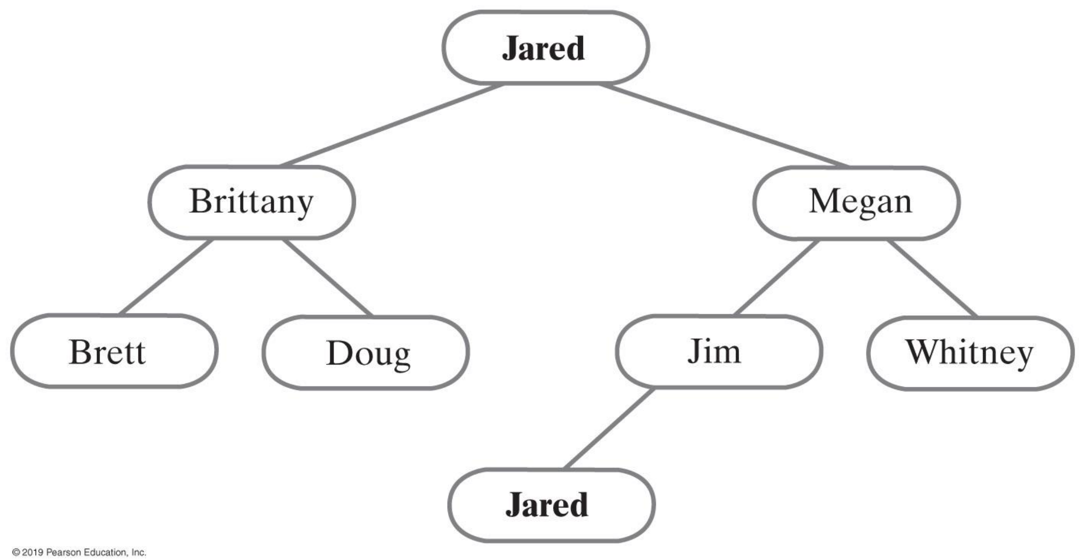


FIGURE 26-2 Adding an entry that matches an entry already in a binary search tree

Duplicate Entries

- If any entry ***e*** has a duplicate entry ***d***, we arbitrarily 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

FIGURE 26-3 A binary search tree with duplicate entries



Beginning the Class Definition

```
package TreePackage;
import java.util.Iterator;
/** A class that implements ADT binary search tree by extending BinaryTree. */
public class BinarySearchTree<T extends Comparable<? super T>>
         extends BinaryTree<T> implements SearchTreeInterface<T>
{
  public BinarySearchTree()
  {
    super();
  } // end default constructor

  public BinarySearchTree(T rootEntry)
  {
    super();
    setRootNode(new BinaryNode<T>(rootEntry));
  } // end constructor

  // Disable setTree (see Segment 26.6)
  public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
                      BinaryTreeInterface<T> rightTree)
  {
    throw new UnsupportedOperationException();
  } // end setTree

    /* Implementations of other methods goes here.  */
} // end BinarySearchTree
```

LISTING 26-2 An outline of the class `BinarySearchTree`

Searching and Retrieving

```
Algorithm bstSearch(binarySearchTree, desiredObject)
// Searches a binary search tree for a given object.
// Returns true if the object is found.
if (binarySearchTree is empty)
   return false
else if (desiredObject == object in the root of binarySearchTree)
   return true
else if (desiredObject < object in the root of binarySearchTree)
   return bstSearch(left subtree of binarySearchTree, desiredObject)
else
   return bstSearch(right subtree of binarySearchTree, desiredObject)
```

Recursive algorithm to search a binary search tree

Pearson

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

Searching and Retrieving

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

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

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

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

  return result;
} // end findEntry
```

The method `getEntry` uses `findEntry`

Searching and Retrieving

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

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

Adding to a Binary Search Tree

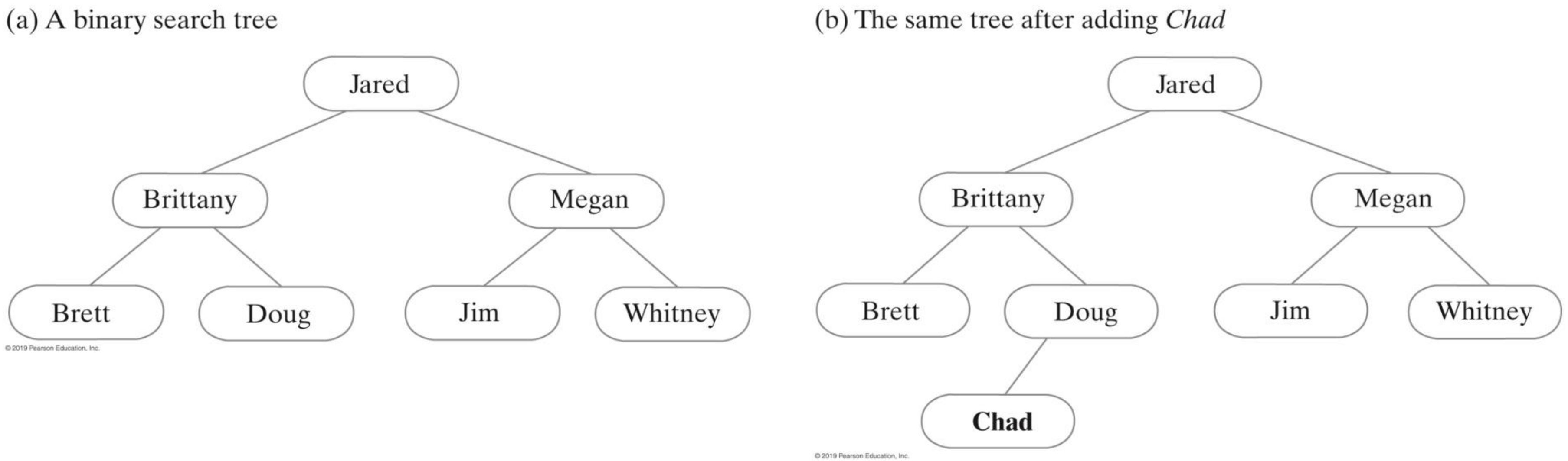


FIGURE 26-4 A binary search tree before and after adding Chad

Adding to a Binary Search Tree

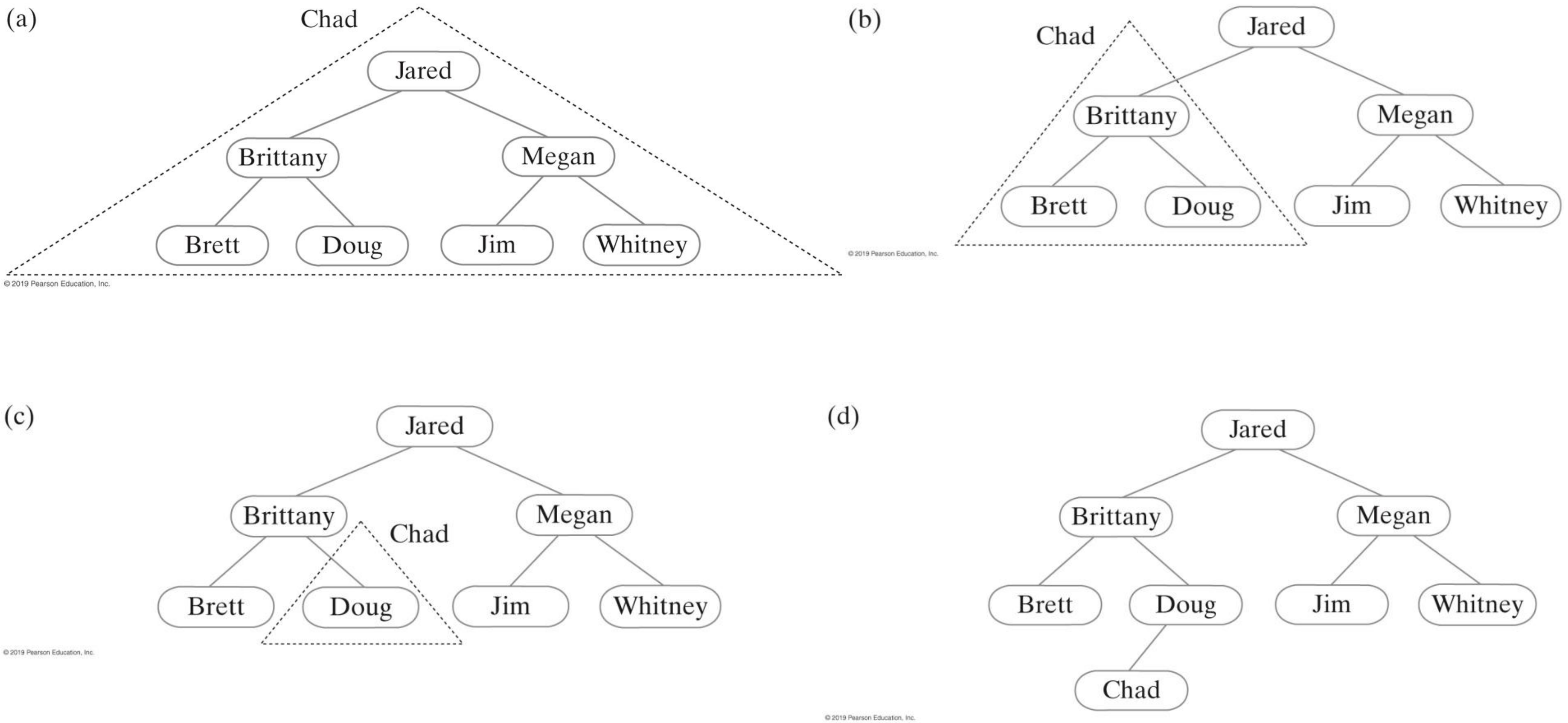


FIGURE 26-5 Recursively adding Chad to smaller subtrees of a binary search tree

Recursive Add Implementation (Part 1)

```
Algorithm addEntry(binarySearchTree, anEntry)
// Adds an entry to a binary search tree that is not empty.
//  Returns null if anEntry did not exist already in the tree. Otherwise, returns the
//tree entry that matched and was replaced by anEntry.
result = null
if (anEntry matches the entry in the root of binarySearchTree)
{
   result = entry in the root
   Replace entry in the root with anEntry
}
else if (anEntry < entry in the root of binarySearchTree)
{
   if (the root of binarySearchTree has a left child)
      result = addEntry(left subtree of binarySearchTree, anEntry)
   else
Give the root a left child containing anEntry
}
```

Recursive algorithm for adding a new entry

Recursive Add Implementation

```
Algorithm addEntry(binarySearchTree, anEntry)
// Adds an entry to a binary search tree that is not empty.
//  Returns null if anEntry did not exist already in the tree. Otherwise, returns the
//tree entry that matched and was replaced by anEntry.
result = null
if (anEntry matches the entry in the root of binarySearchTree)
{
    result = entry in the root
    Replace entry in the root with anEntry
    }
else if (anEntry < entry in the root of binarySearchTree)
{
    if (the root of binarySearchTree has a left child)
        result = addEntry(left subtree of binarySearchTree, anEntry)
    else
    Give the root a left child containing anEntry
}
else // anEntry > entry in the root of binarySearchTree
{
    if (the root of binarySearchTree has a right child)
        result = addEntry(right subtree of binarySearchTree, anEntry)
    else
        Give the root a right child containing anEntry
}
return result
```

Recursive algorithm for adding a new entry

Recursive Implementation

```
Algorithm add(binarySearchTree, anEntry)
// Adds an entry to a binary search tree.
// Returns null if anEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by anEntry.
result = null
if (binarySearchTree is empty)
   Create a node containing anEntry and make it the root of binarySearchTree
else
   result = addEntry(binarySearchTree, anEntry)
return result;
```

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

Recursive Implementation

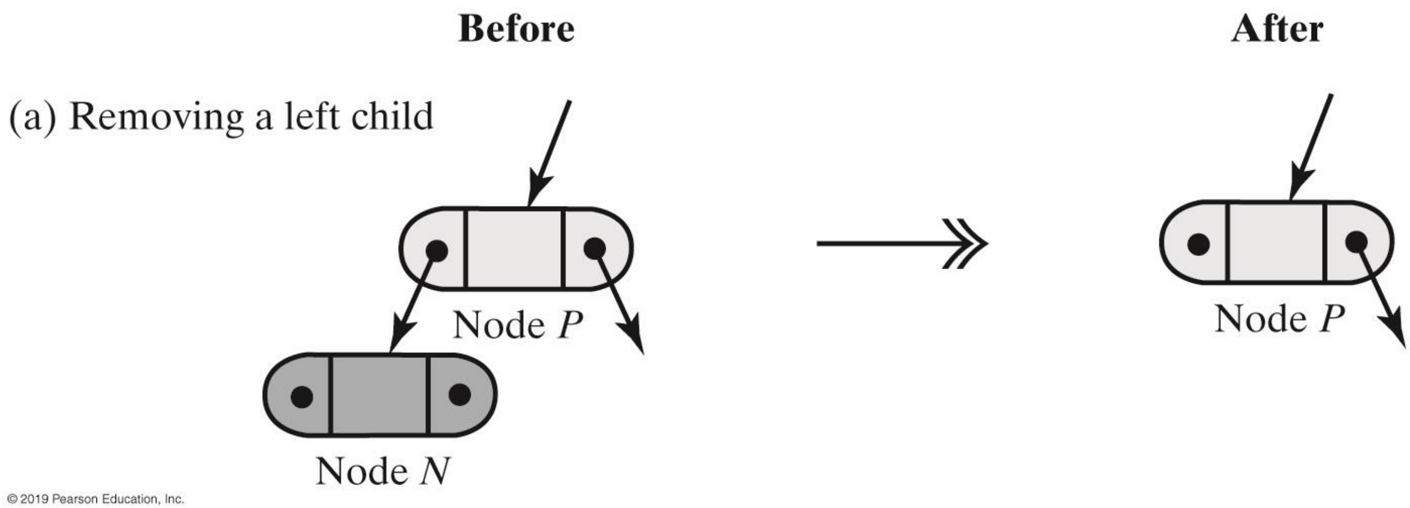
```
public T add(T anEntry)
{
   T result = null;

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

   return result;
} // end add
```

The public method `add`

Recursive Implementation — method `addEntry`

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

  if (comparison == 0)
  {
    result = rootNode.getData();
    rootNode.setData(anEntry);
  }
  else if (comparison < 0)
  {
    if (rootNode.hasLeftChild())
      result = addEntry(rootNode.getLeftChild(), anEntry);
    else
      rootNode.setLeftChild(new BinaryNode<>(anEntry));
  }
  else
  {
   if (rootNode.hasRightChild())
      result = addEntry(rootNode.getRightChild(), anEntry);
    else
      rootNode.setRightChild(new BinaryNode<>(anEntry));
  } // end if

  return result;
} // end addEntry
```

Removing a Value



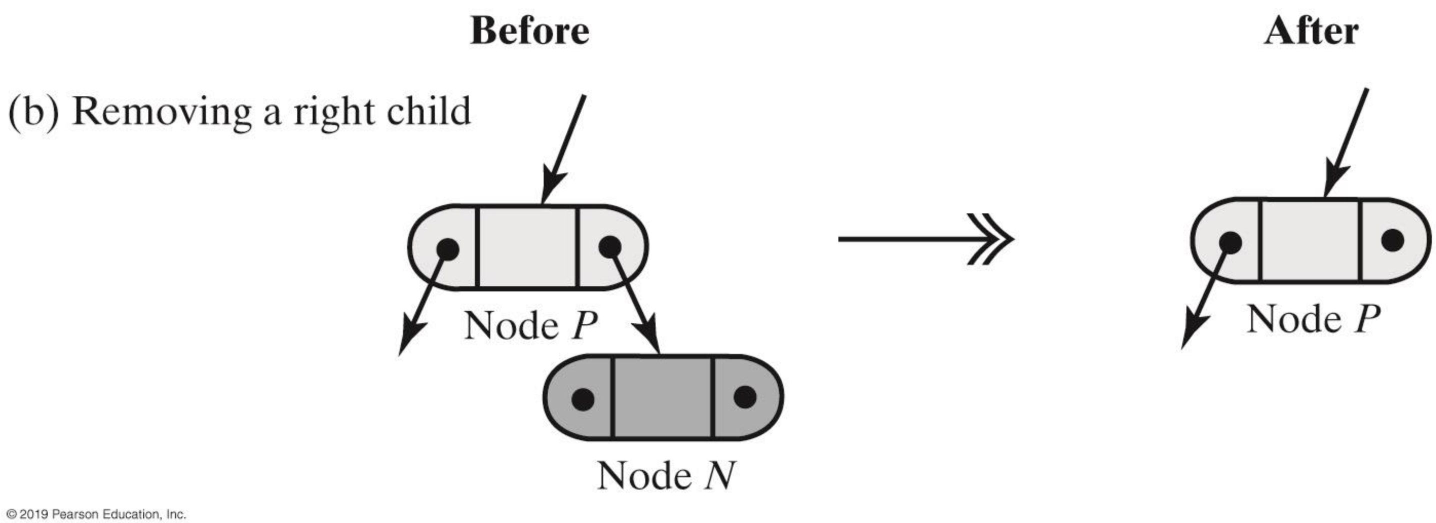


FIGURE 26-6 Removing a leaf node N from its parent node P

Removing a Value

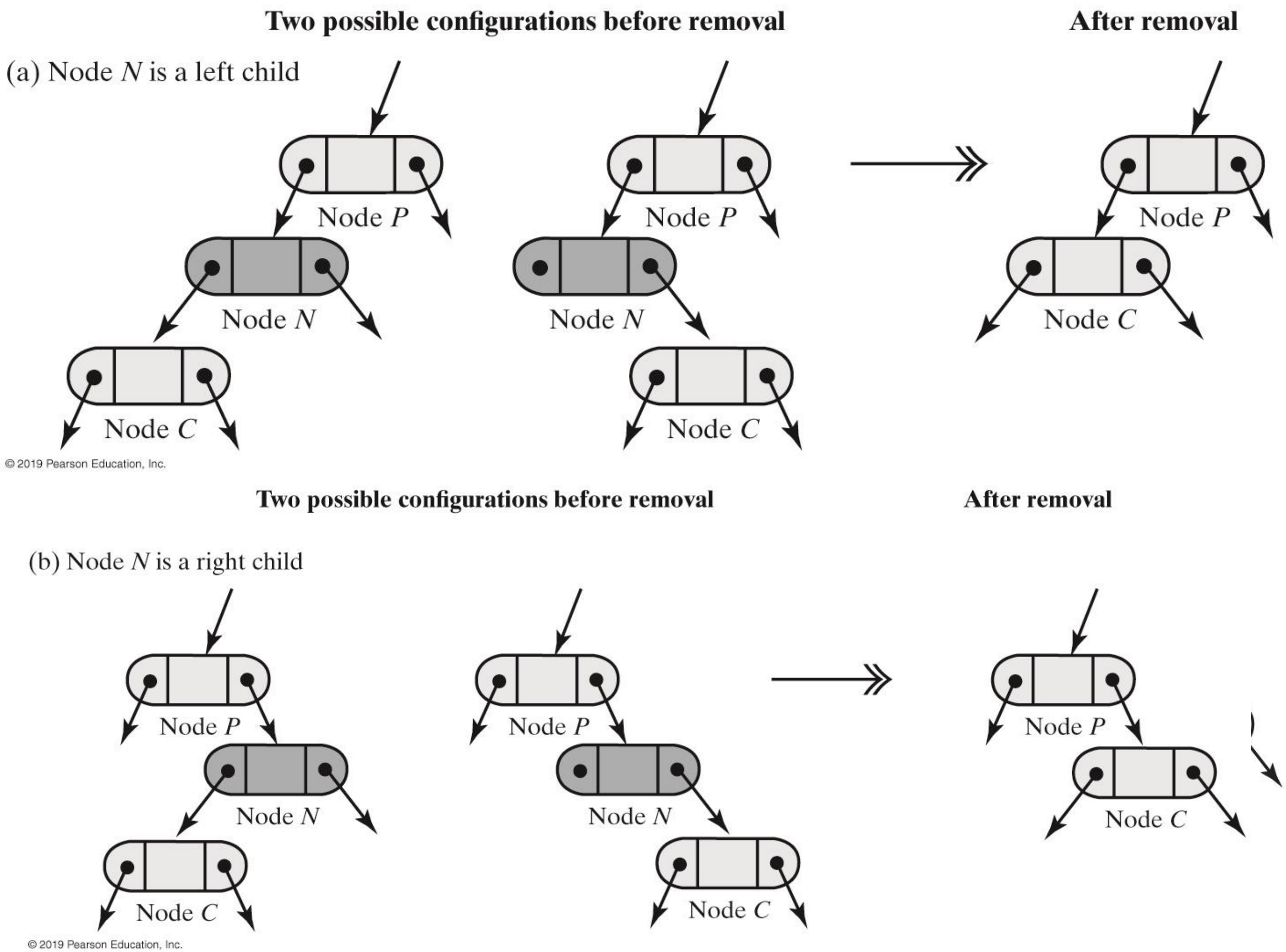


FIGURE 26-7 Removing a node *N* from its parent node *P* when *N* has one child

Removing a Value

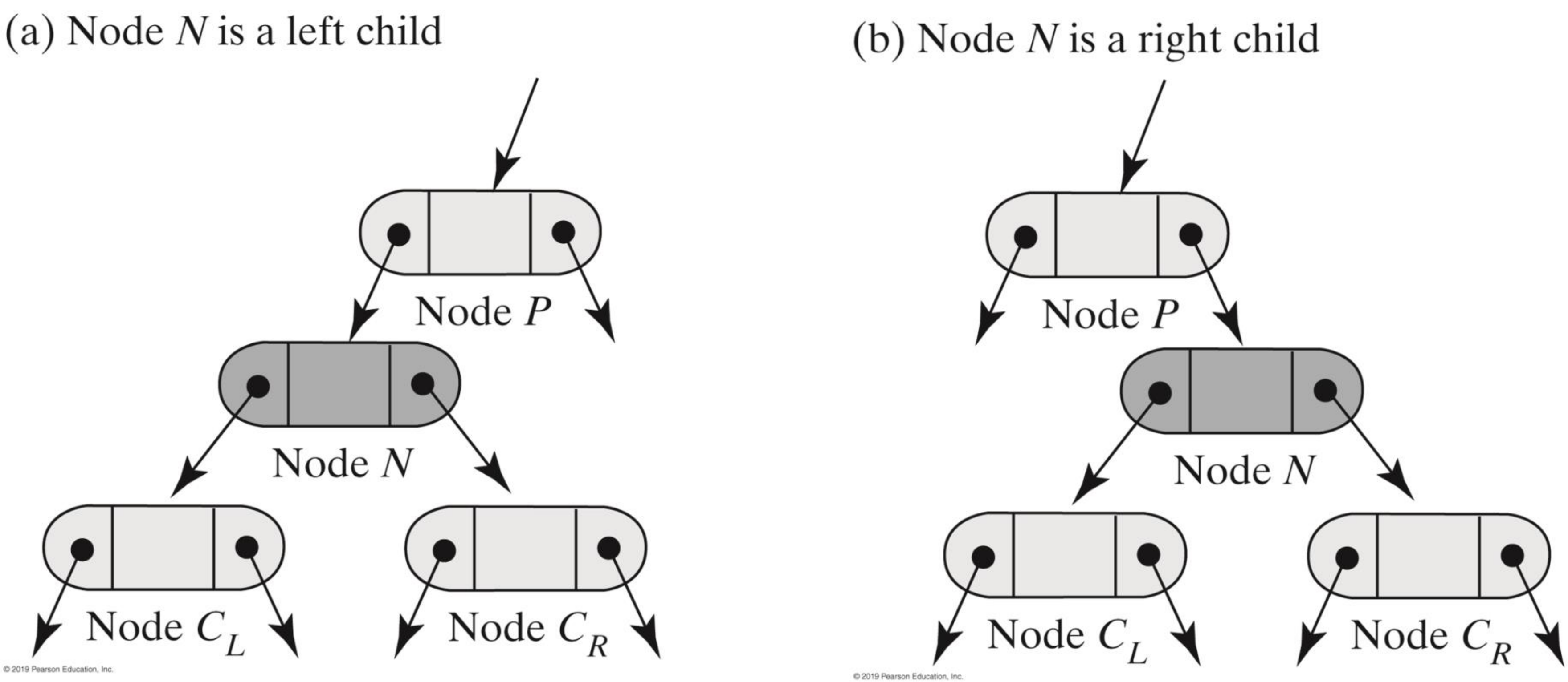


FIGURE 26-8 Two possible configurations of a node N that has two children

Removing a Value

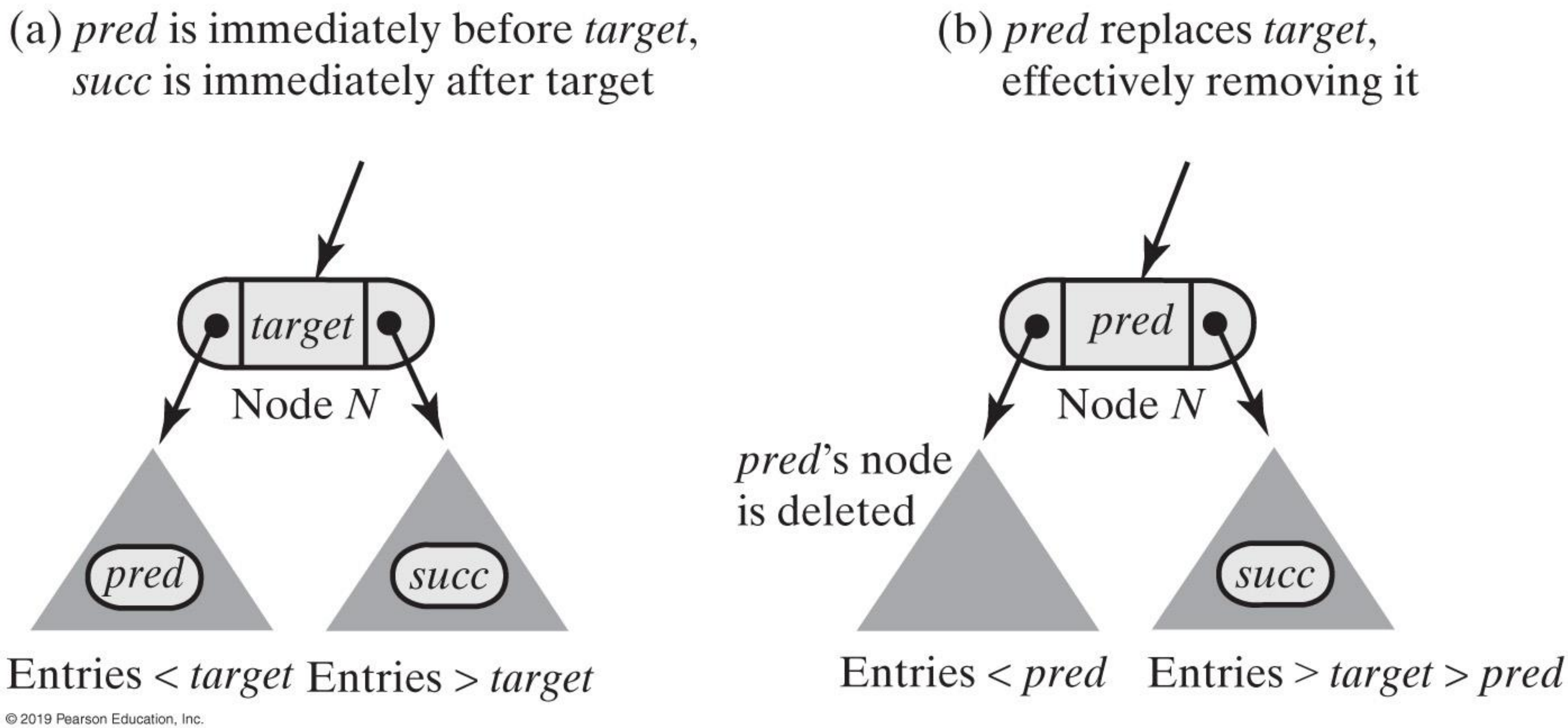


FIGURE 26-9 Node *N* and its subtrees before and after removing target

Removing a Value

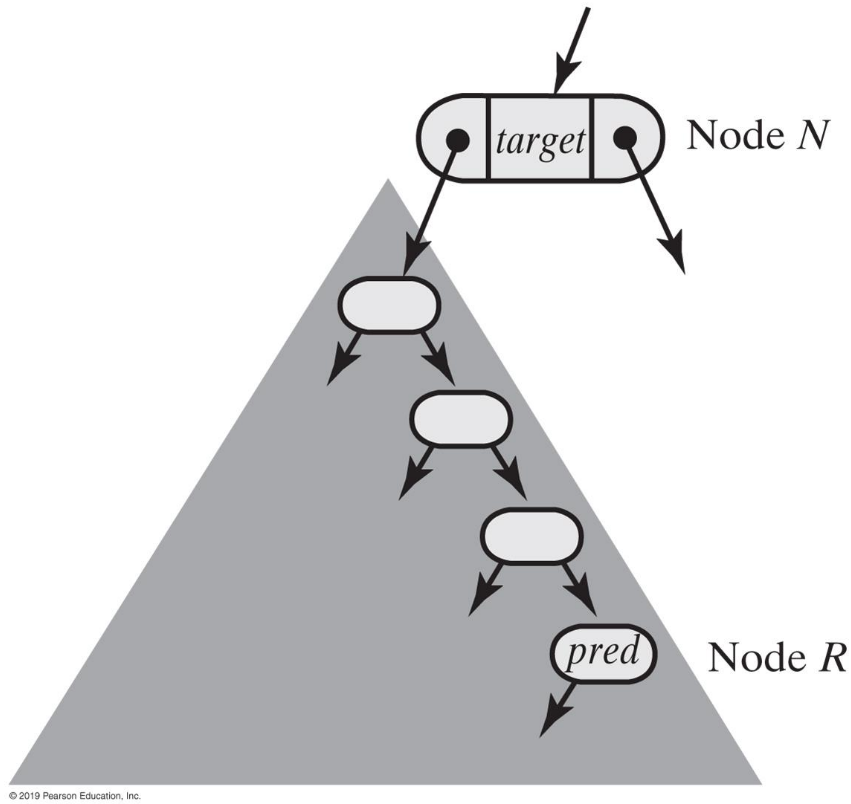
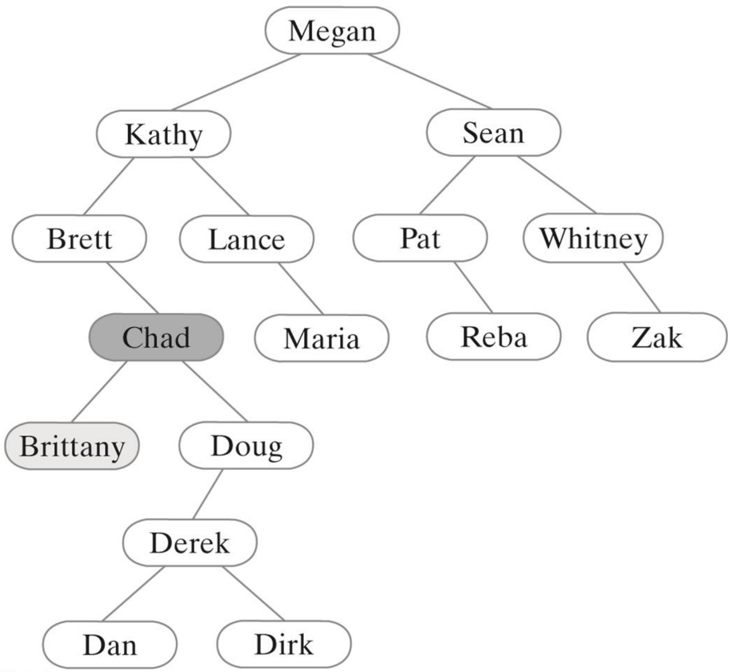


FIGURE 26-10 The largest entry *pred* in node *N's* left subtree occurs in the subtree's rightmost node *R*

FIGURE 26-11 Successive removals from a binary search tree (Part 1)

(a) A binary search tree



(b) The tree after removing *Chad*

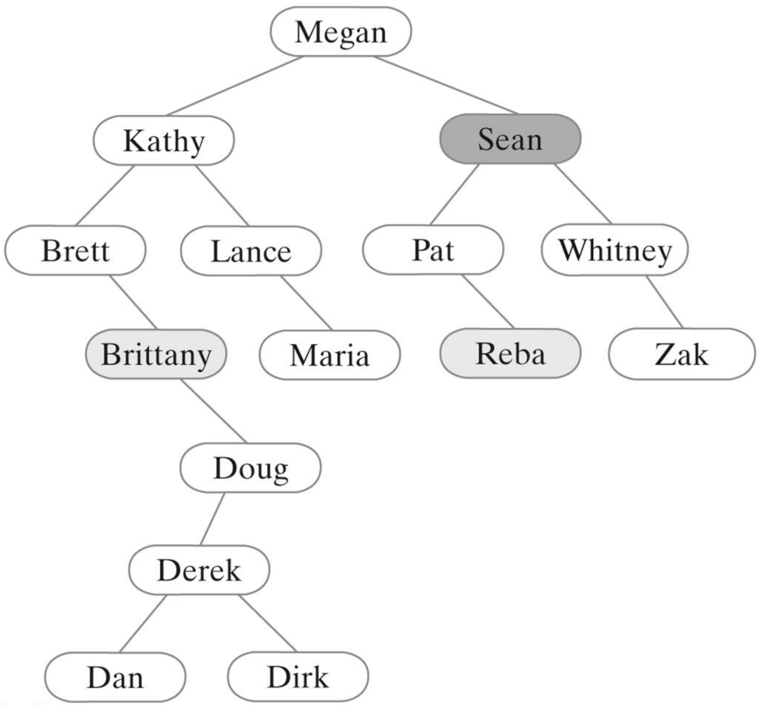
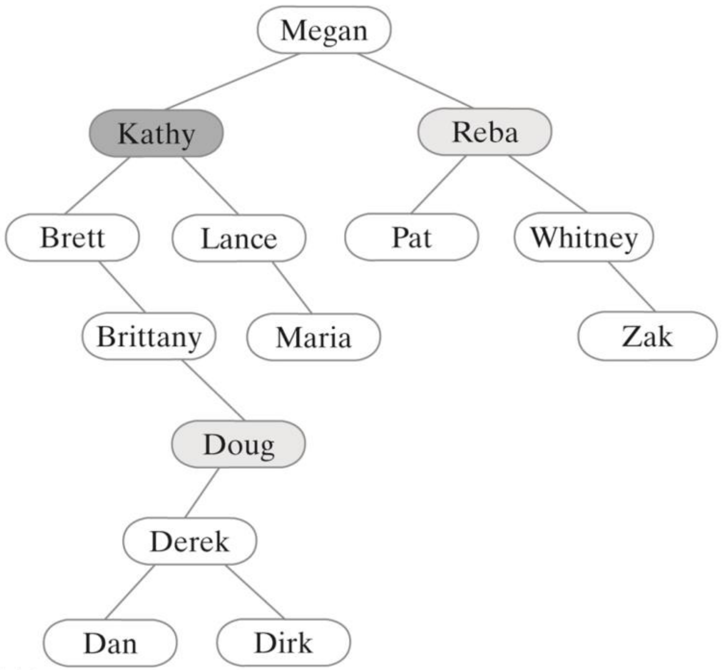
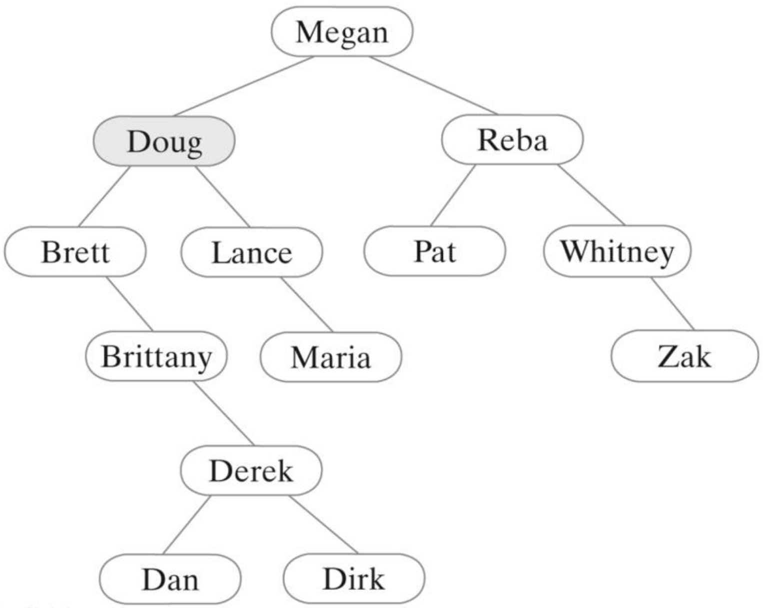


FIGURE 26-11 Successive removals from a binary search tree (Part 2)

(c) The tree after removing *Sean*



(d) The tree after removing *Kathy*



Removing a Value

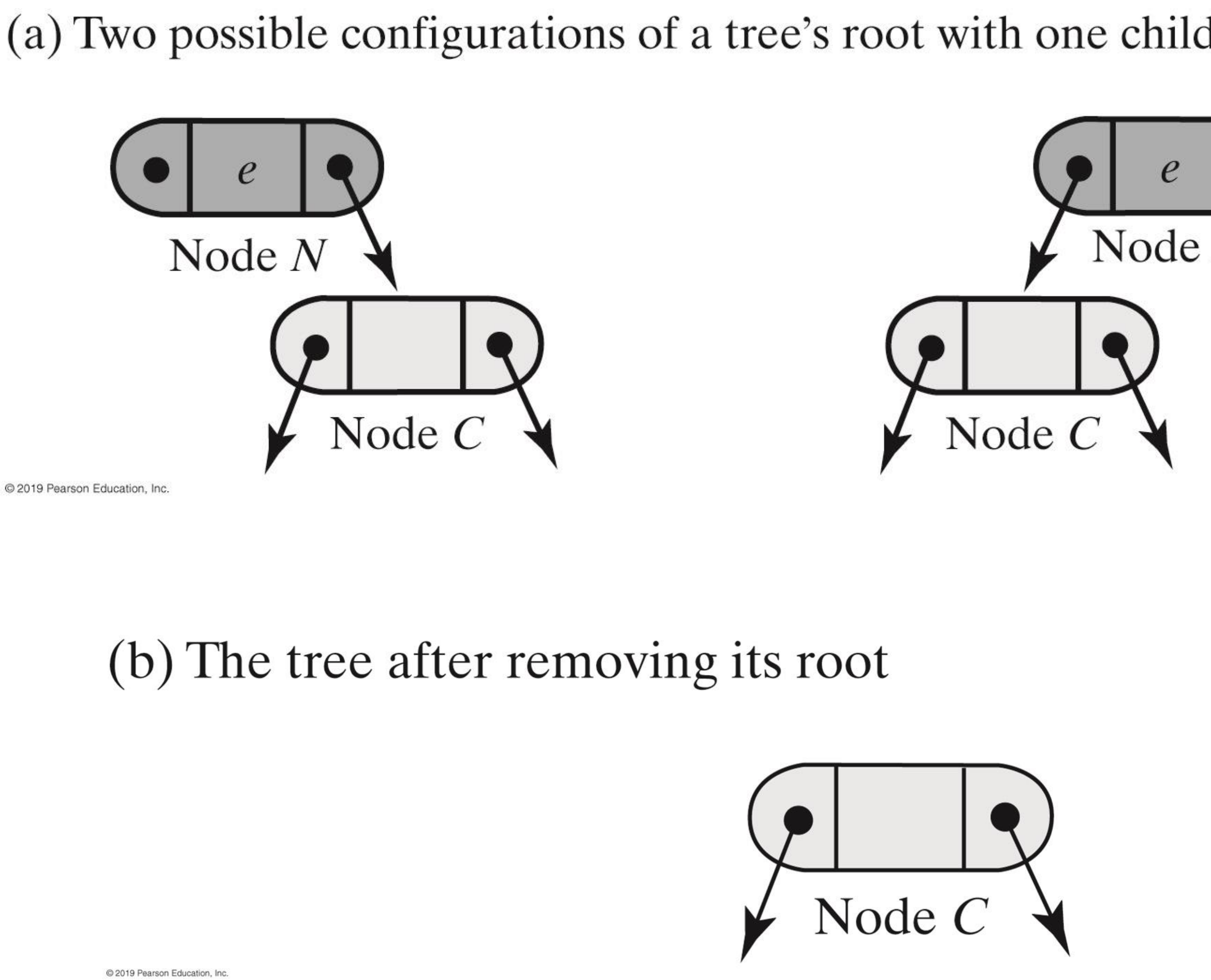


FIGURE 26-12 Removing the root when it has one child

Recursive Implementation

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

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

Recursive Implementation

```java
public T remove(T anEntry)
{
   ReturnObject oldEntry = new ReturnObject(null);
   BinaryNode<T> newRoot = removeEntry(getRootNode(), anEntry, oldEntry);
   setRootNode(newRoot);

   return oldEntry.get();
} // end remove
```

The public method `remove`

Recursive Implementation

```
// Removes an entry from the tree rooted at a given node.
private BinaryNode<T> removeEntry(BinaryNode<T> rootNode, T anEntry,
                                  ReturnObject oldEntry)
{
   if (rootNode != null)
   {
      T rootData = rootNode.getData();
      int comparison = entry.compareTo(rootData);

      if (comparison == 0)       // anEntry == root entry
      {
         oldEntry.set(rootData);
         rootNode = removeFromRoot(rootNode);
      }
      else if (comparison < 0)   // anEntry < root entry
      {
         BinaryNode<T> leftChild = rootNode.getLeftChild();
         BinaryNode<T> subtreeRoot = removeEntry(leftChild, anEntry, oldEntry);
         rootNode.setLeftChild(subtreeRoot);
      }
      else                       // anEntry > root entry
      {
         BinaryNode<T> rightChild = rootNode.getRightChild();
         // A different way of coding than for left child:
         rootNode.setRightChild(removeEntry(rightChild, anEntry, 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

```java
// Removes the entry in a given root node of a 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));
   } // 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

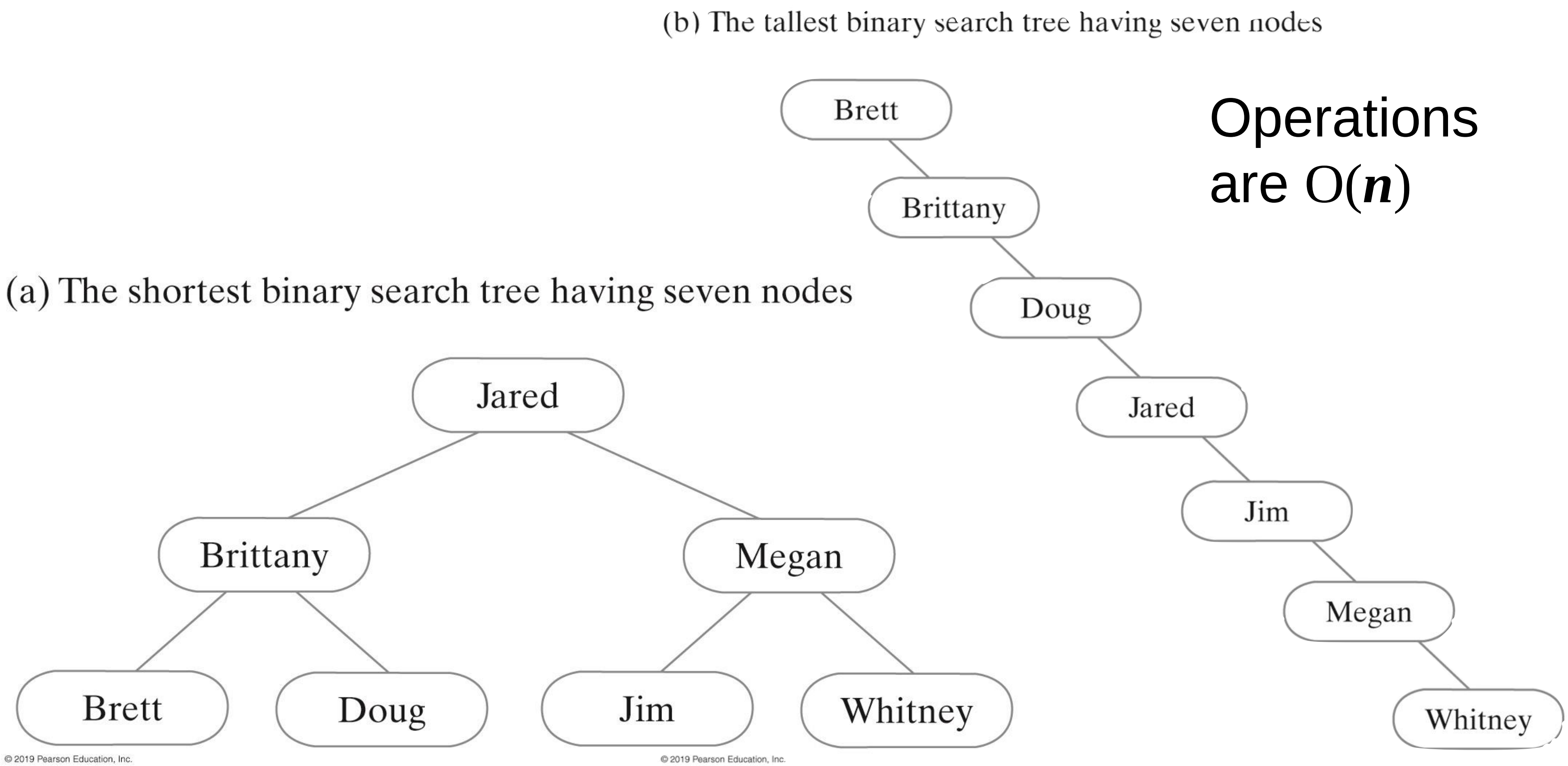
```
// Removes the node containing the largest entry in a given tree.
// rootNode is the root node of the tree.
// Returns the root node of the revised tree.
private BinaryNode<T> removeLargest(BinaryNode<T> rootNode)
{
   if (rootNode.hasRightChild())
   {
      BinaryNode<T> rightChild = rootNode.getRightChild();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

   return rootNode;
} // end removeLargest
```

The private method `removeLargest`

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



Operations are O($\boldsymbol{n}$)

Operations are O(log $\boldsymbol{n}$)

FIGURE 26-13 Two binary search trees that contain the same data

End

Chapter 26

Pearson