Lecture 27: BST Implementation

CS 0445: Data Structures

Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Nov 21, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

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

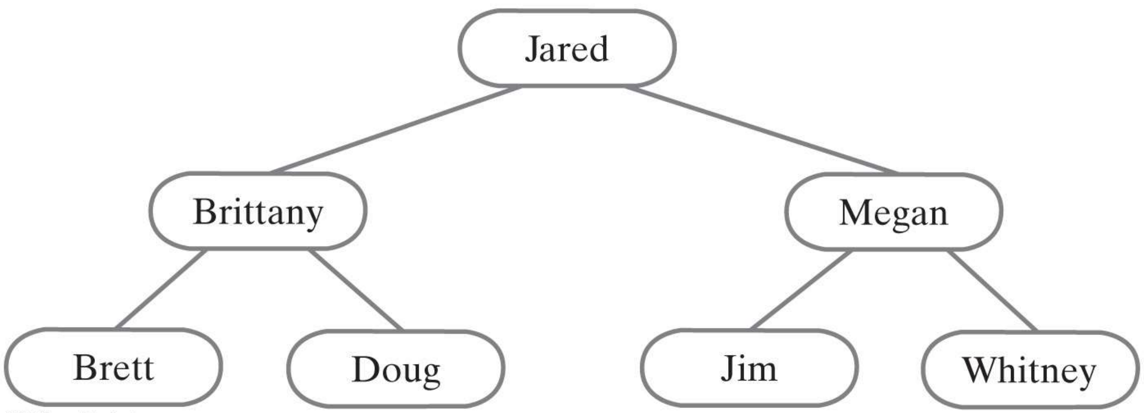


FIGURE 26-1 A binary search tree of names

Interface for the Binary Search Tree (Part 1)

- LISTING 26-1 An interface for a search tree

```
/** 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);
```

Interface for the Binary Search Tree (Part 2)

- LISTING 26-1 An interface for a search tree

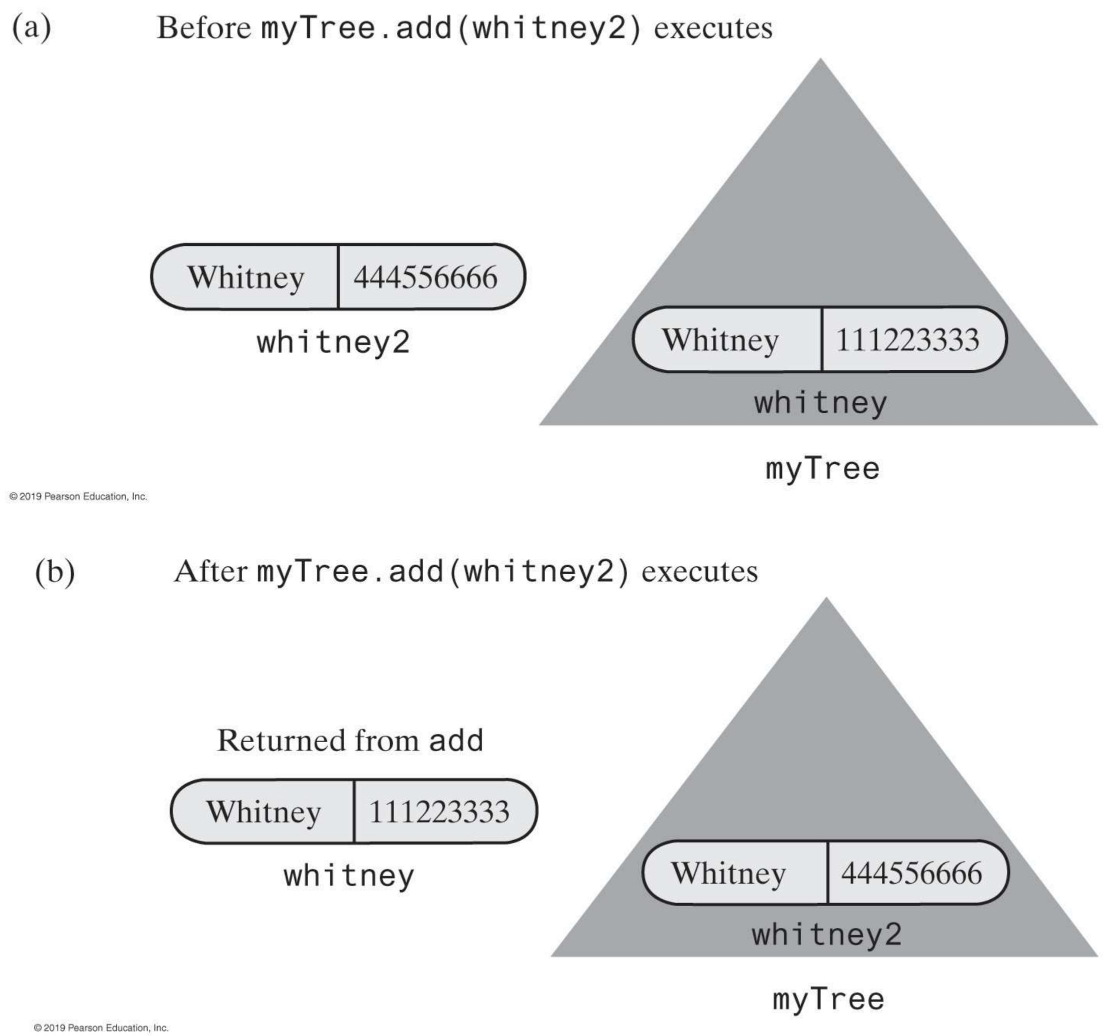
```
/** Removes a specific entry from this tree.
   @param anEntry  An object to be removed.
   @return  Either the object that was removed from the tree or
            null if no such object exists. */
public T remove(T anEntry);

/** 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
```

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

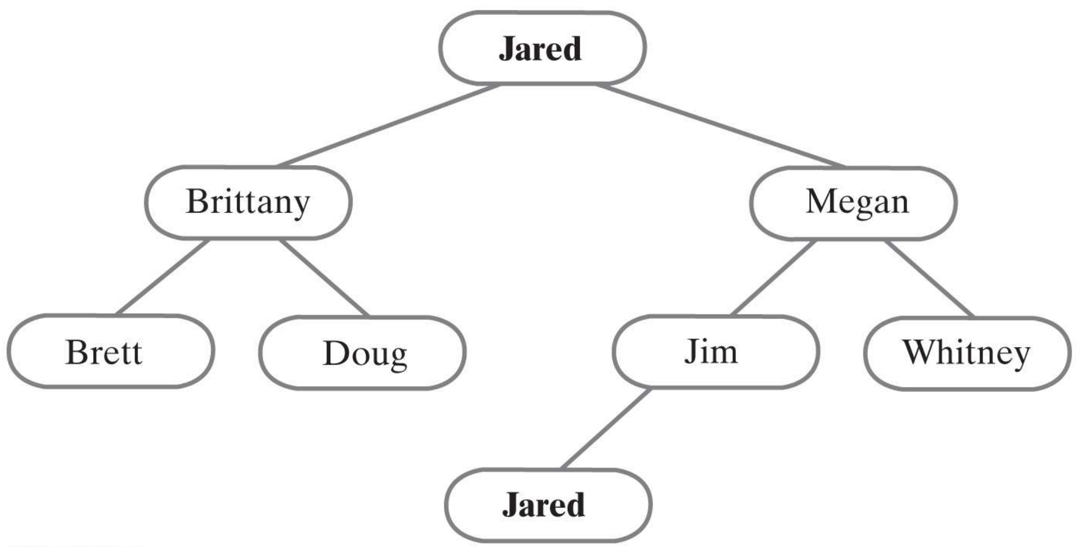


FIGURE 26-3
A binary search tree with duplicate entries

Beginning the Class Definition

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

```
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
```

Searching and Retrieving

- Recursive algorithm to search a binary search tree

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)**

Searching and Retrieving

- Algorithm that describes actual implementation more closely

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)**

Searching and Retrieving

- The method `getEntry` uses `findEntry`

```
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
```

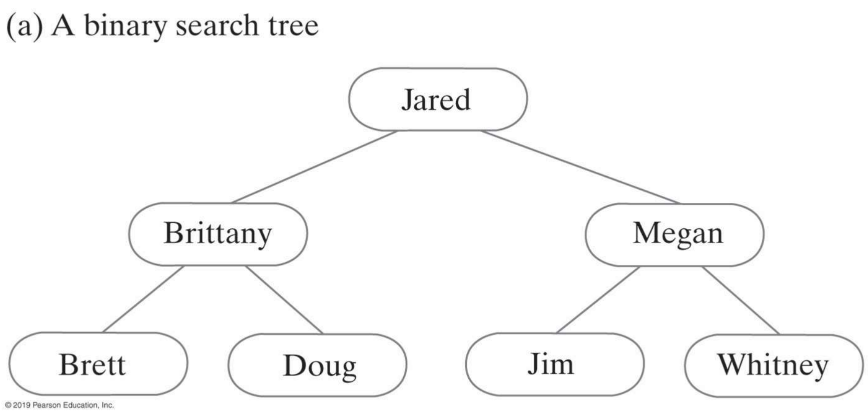
Searching and Retrieving

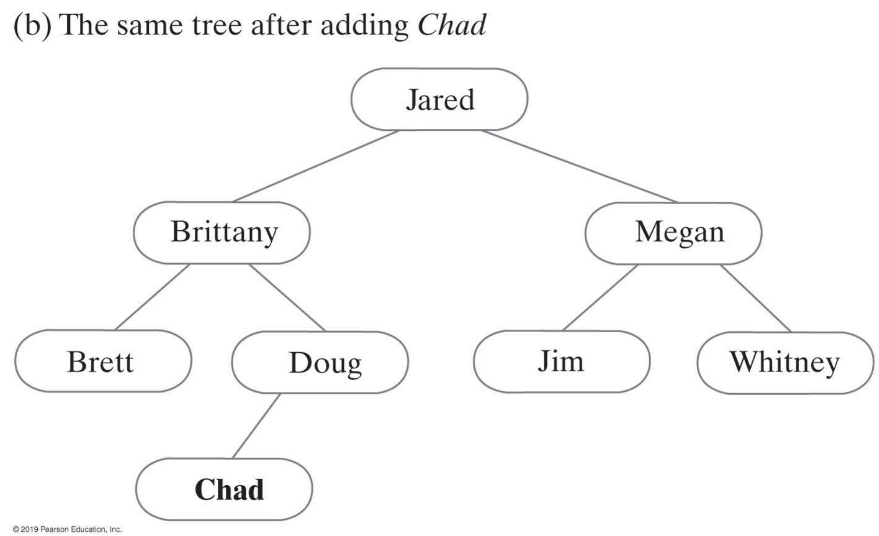
- Method `contains` can simply call `getEntry` to see whether a given entry is in the tree

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

Adding to a Binary Search Tree

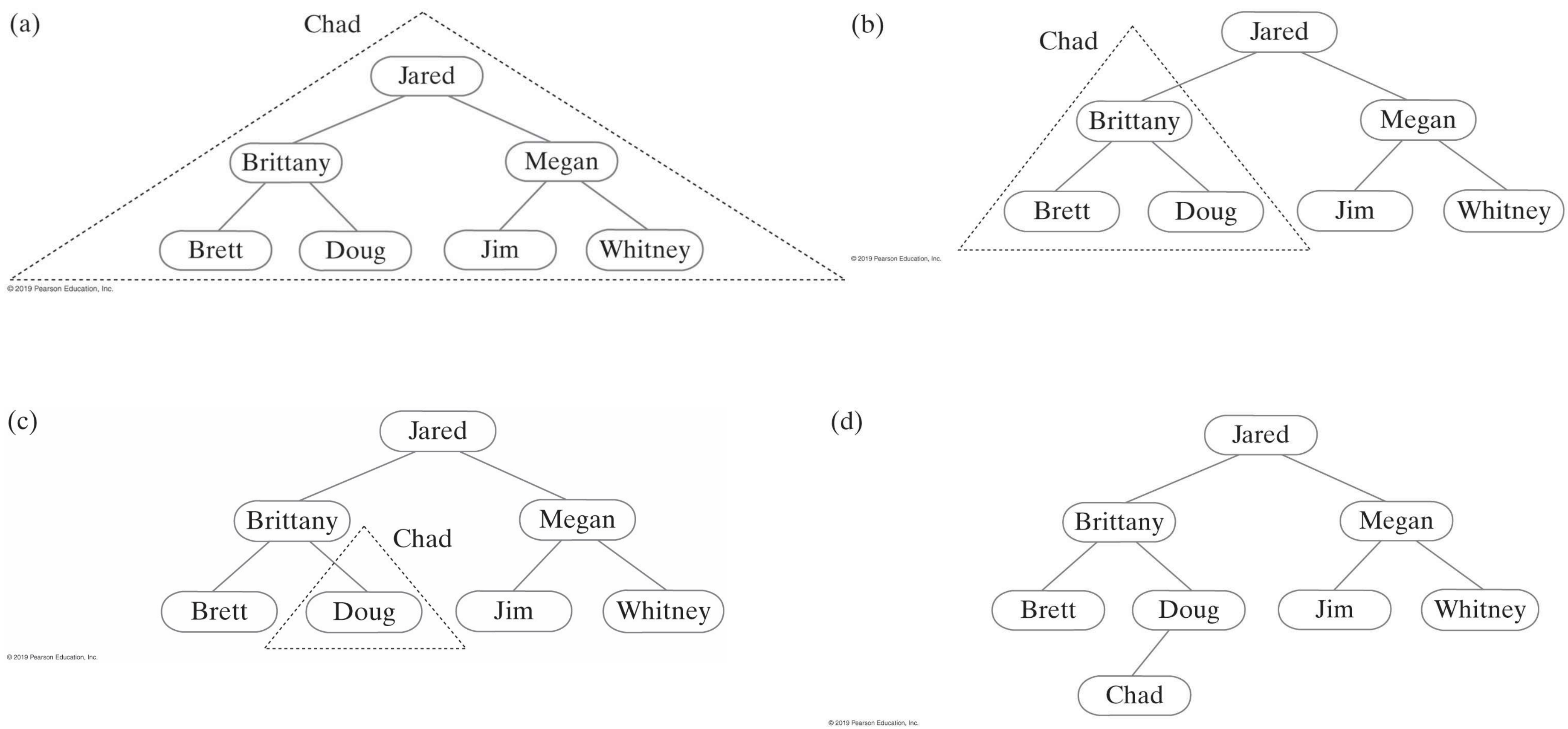
- FIGURE 26-4 A binary search tree before and after adding Chad





Adding to a Binary Search Tree

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



Recursive Add Implementation

- Recursive algorithm for adding a new entry

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 Implementation

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

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;

Recursive Implementation

- The public method `add`

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

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

  return result;
} // end add
```

Recursive Implementation — method `addEntry`

```java
// 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
```

Iterative Implementation (Part 1)

- Iterative algorithm for adding a new entry

Algorithm **addEntry(binarySearchTree, anEntry)**
// *Adds a new 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
currentNode = *root node of* **binarySearchTree found = false**
while (found *is false***)**
{
    if (anEntry *matches the entry in* **currentNode)**
    {
        found = true
        result = *entry in* **currentNode**
        Replace entry in **currentNode** *with* **anEntry**
    }
    else if (newEntry < *entry in* **currentNode)**
    {

Iterative Implementation (Part 2)

- Iterative algorithm for adding a new entry

```
                if (currentNode has a left child)
                        currentNode = left child of currentNode
                else
                {
                        found  = true
                        Give currentNode a left child containing anEntry
                }
        } // end if-else
        else // anEntry > entry in currentNode
        {
                if (currentNode has a right child)
                        currentNode = right child of currentNode
                else
                {
                        found  = true
                        Give currentNode a right child containing anEntry
                }
        }  // end if
} // end while
return result
```

Iterative Implementation— method `addEntry` (Part 1)

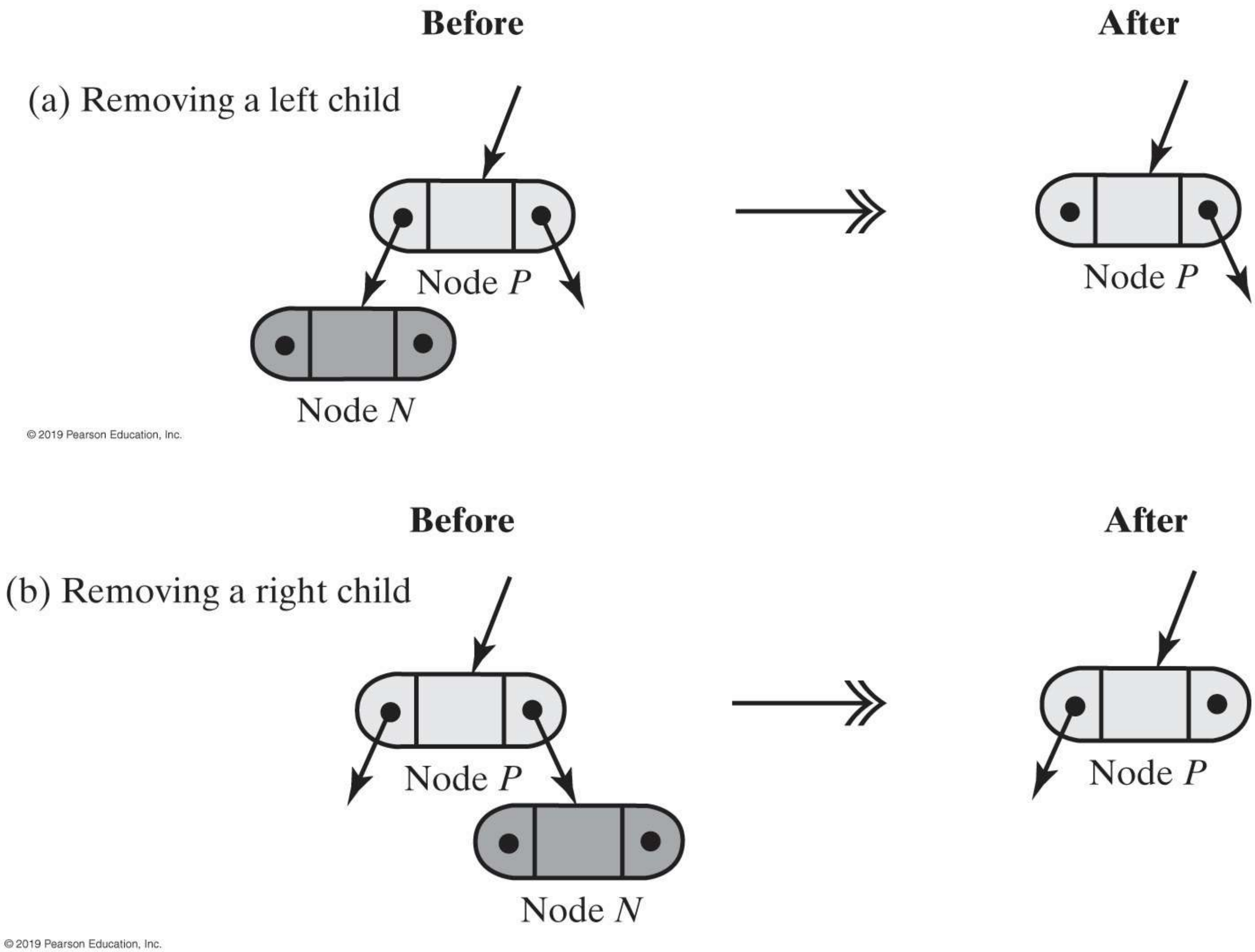
```
private T addEntry(T anEntry) {
  BinaryNode<T> currentNode = getRootNode();
  // Assertion: currentNode != null
  T result = null;
  boolean found = false;

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

    if (comparison == 0)
    {  // anEntry matches currentEntry;
      // return and replace currentEntry
      found = true;
      result = currentEntry;
      currentNode.setData(anEntry);
    }
    else if (comparison < 0)
    {
      if (currentNode.hasLeftChild())
        currentNode = currentNode.getLeftChild();
      else
      {
        found = true;
        currentNode.setLeftChild(new BinaryNode<>(anEntry));
      } // end if
    }
```

Iterative Implementation— method `addEntry` (Part 2)

```
else {
      // Assertion: comparison > 0

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

  return result;
} // end addEntry
```

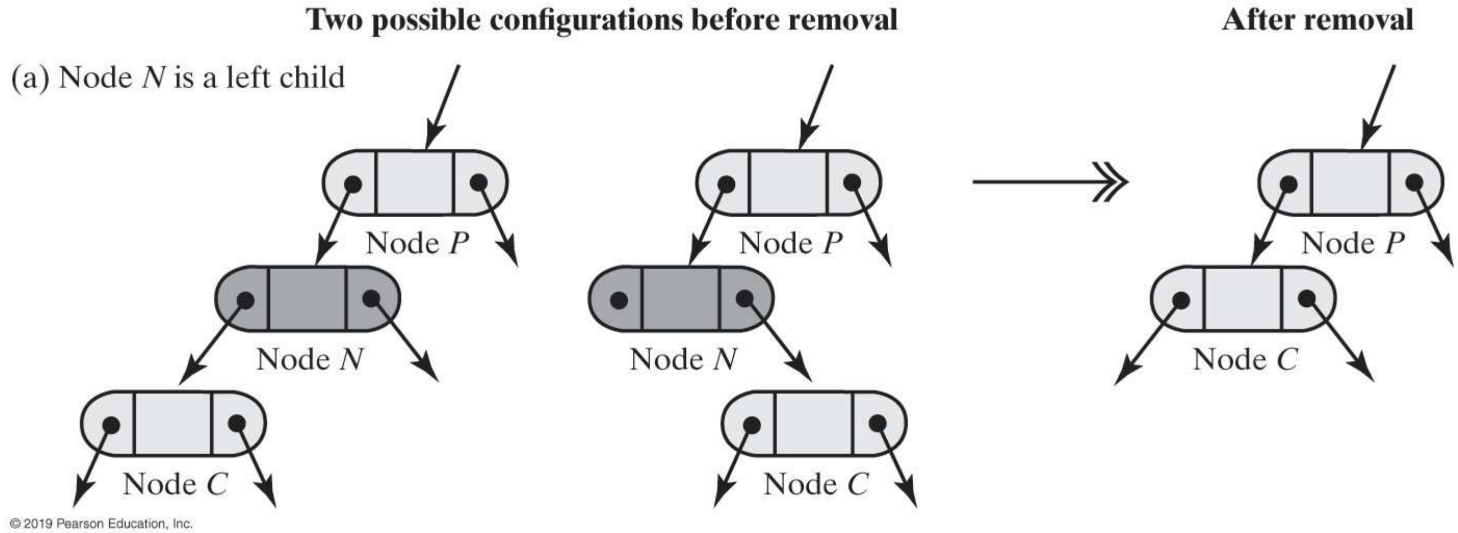
Removing a Value

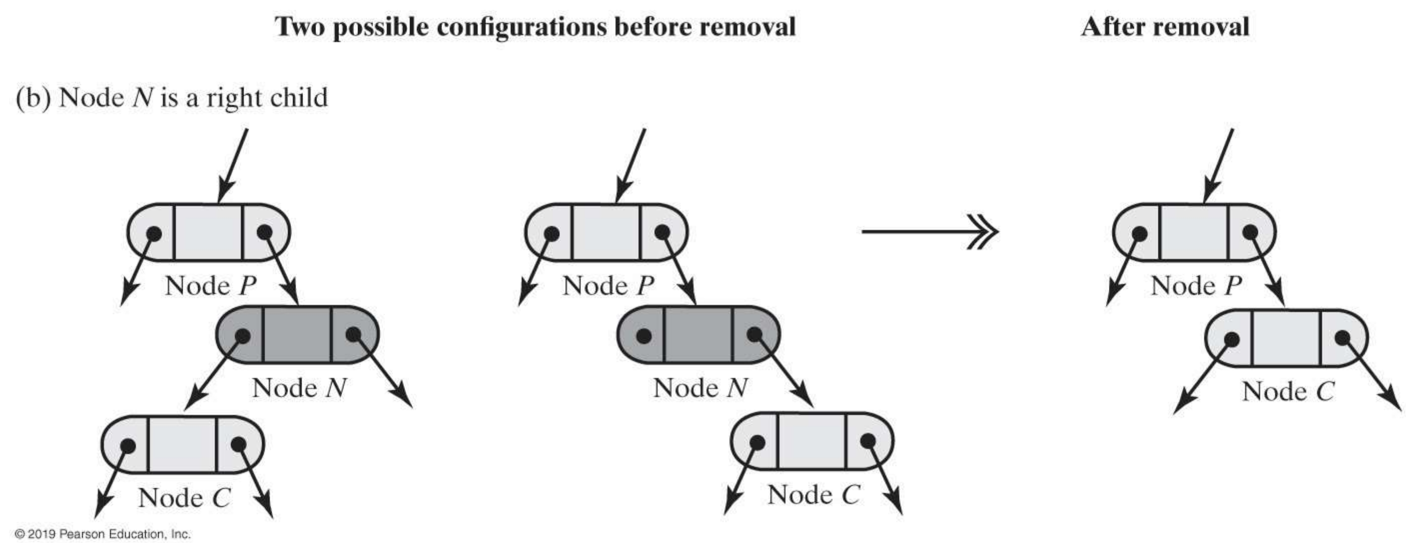
- FIGURE 26-6 Removing a leaf node N from its parent node P



Removing a Value

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

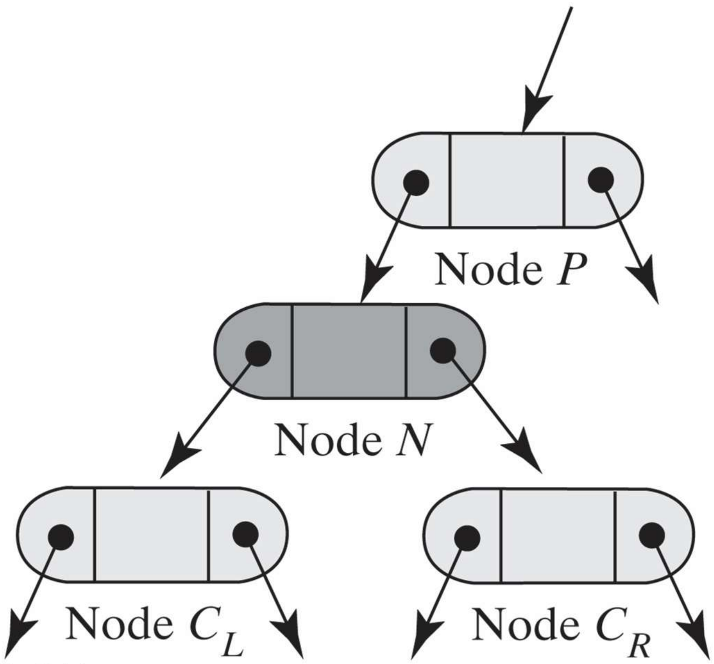




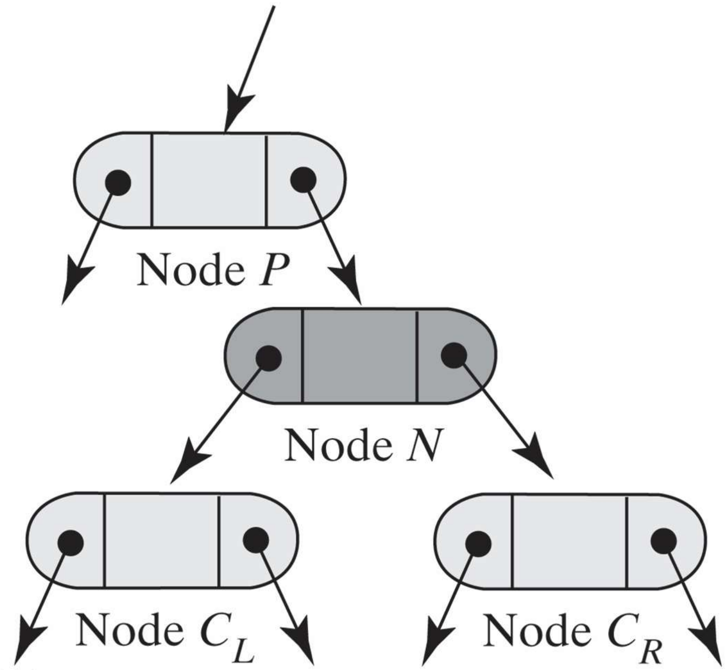
Removing a Value

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

(a) Node *N* is a left child

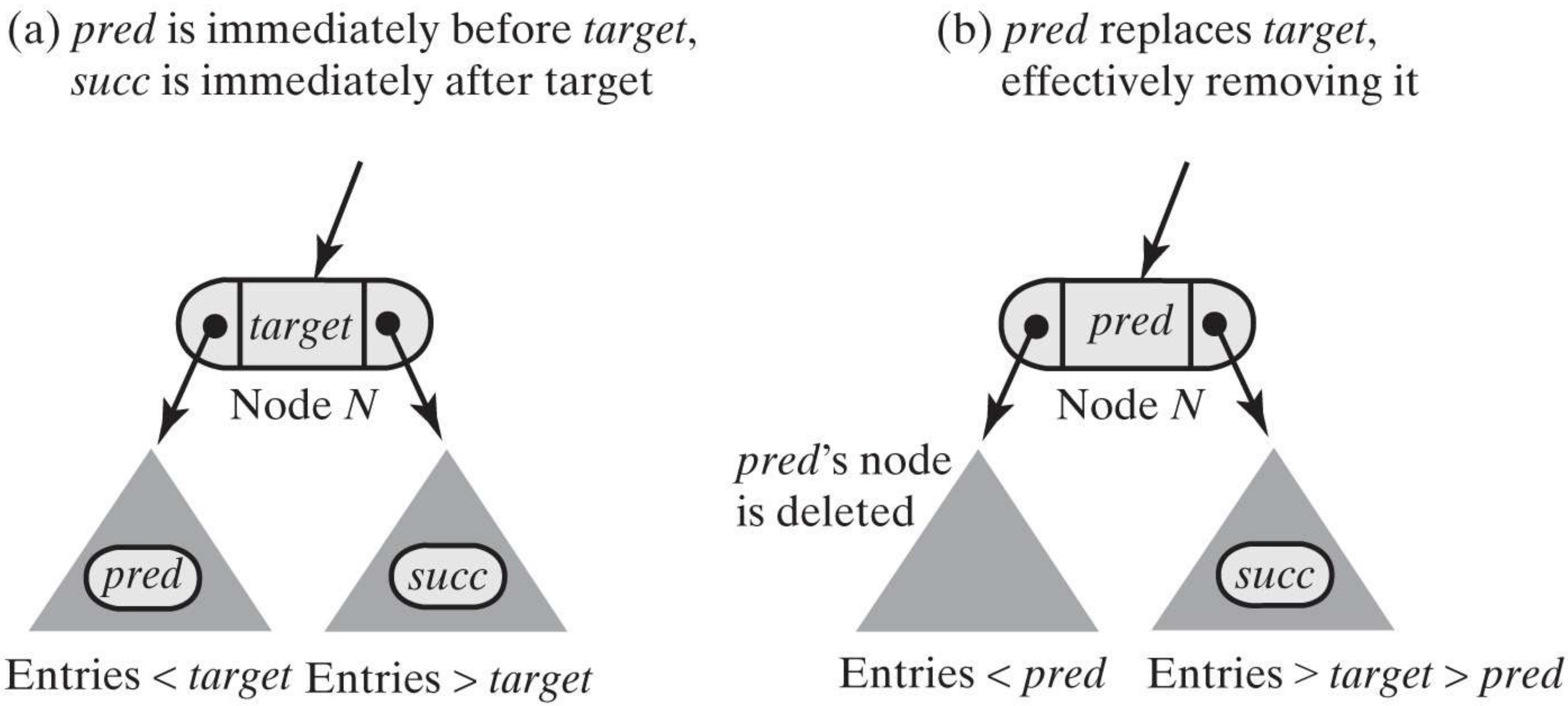


(b) Node *N* is a right child



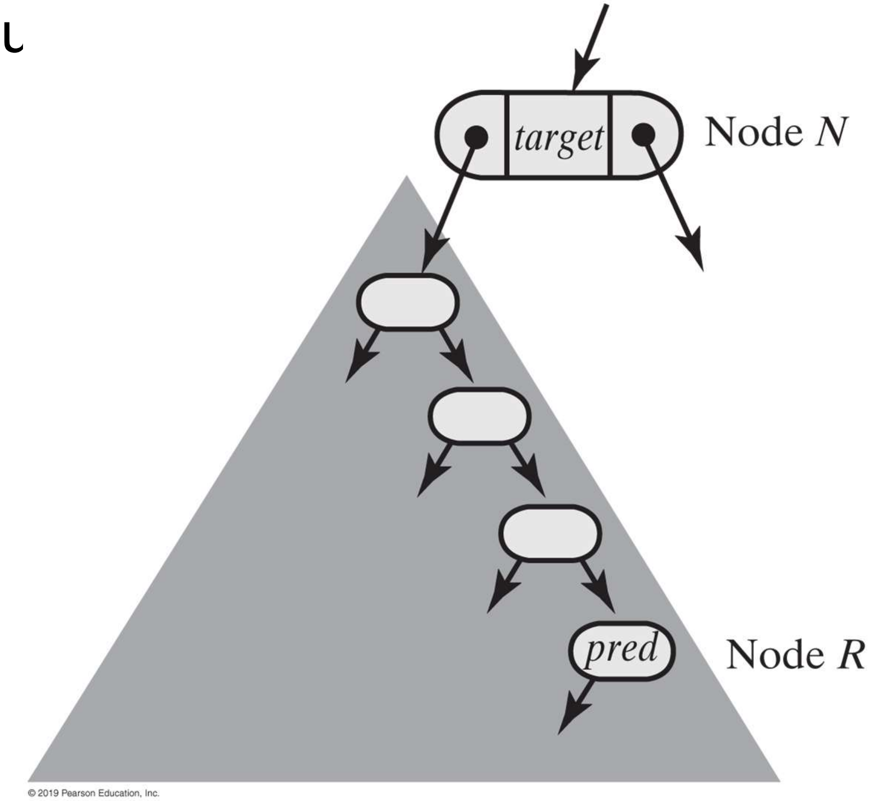
Removing a Value

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



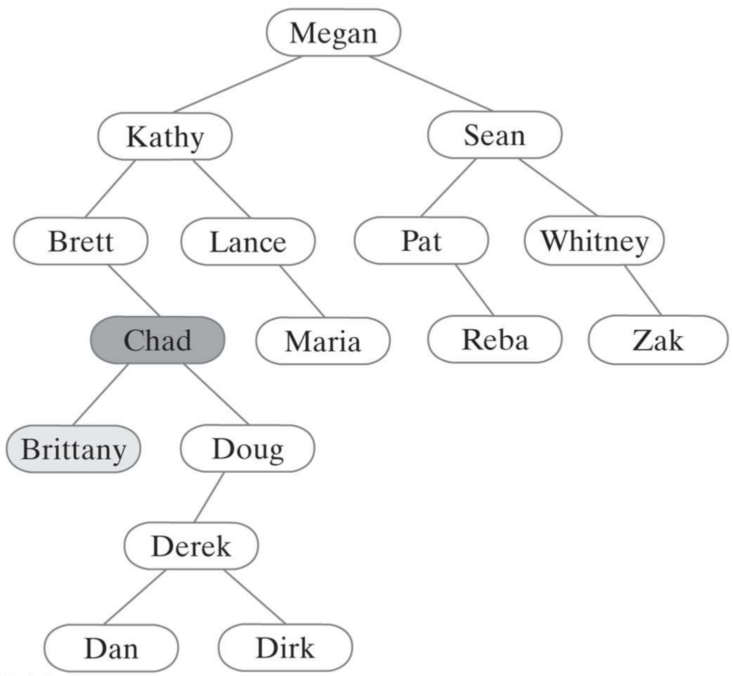
Removing a Value

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

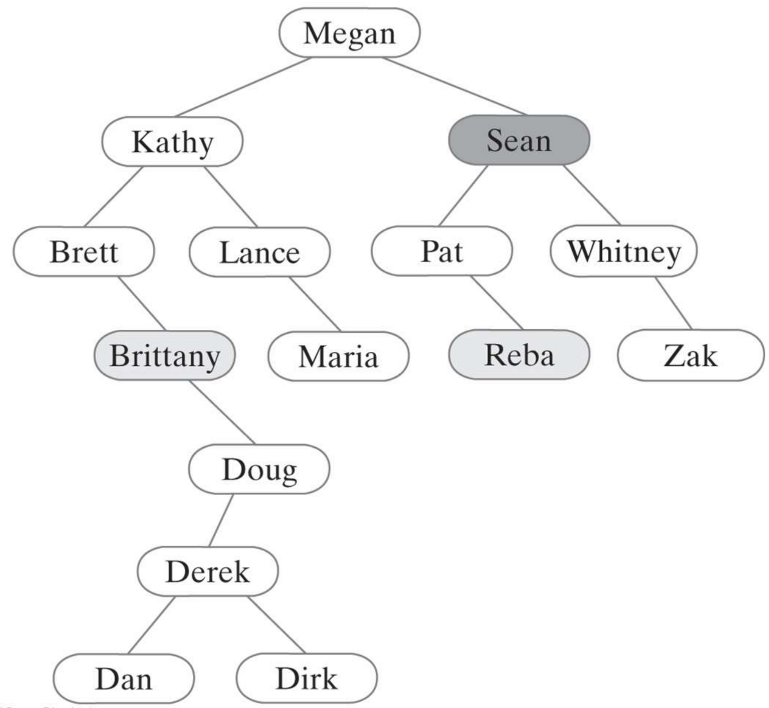


Successive removals from a binary search tree (Part 1)

(a) A binary search tree

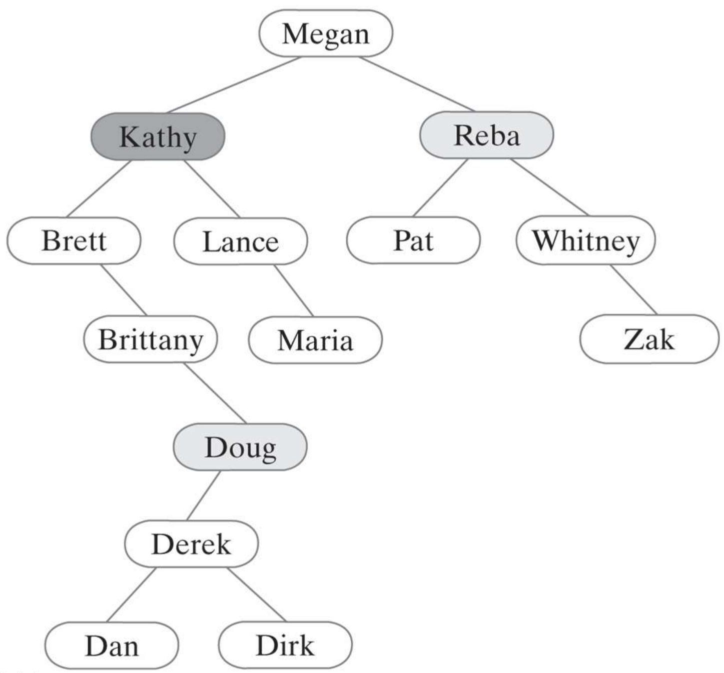


(b) The tree after removing *Chad*

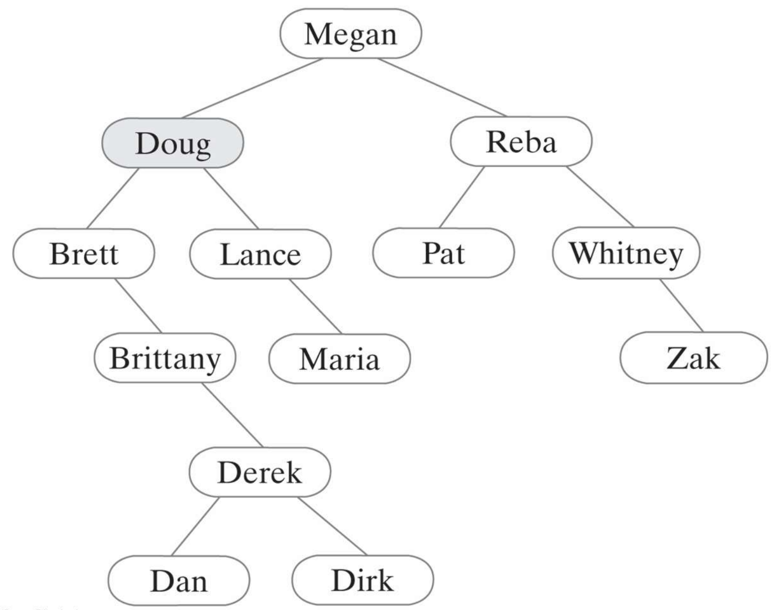


Successive removals from a binary search tree (Part 2)

(c) The tree after removing *Sean*



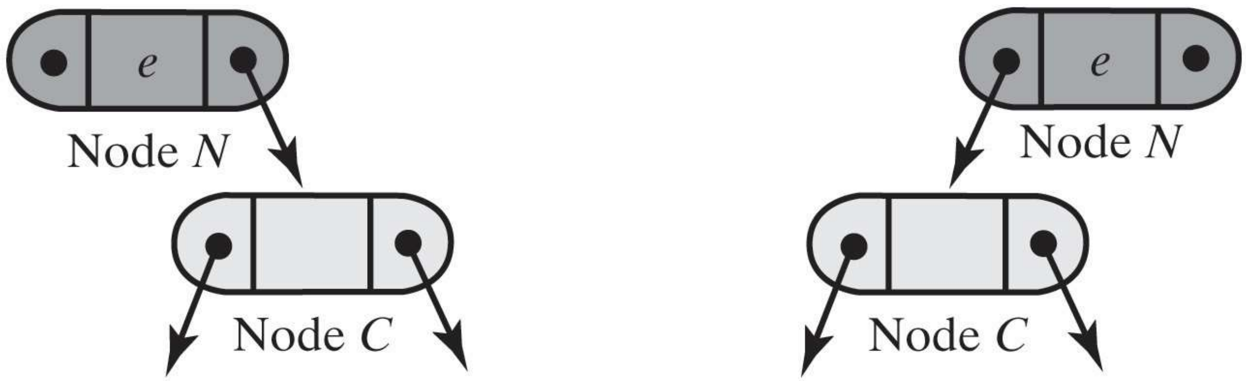
(d) The tree after removing *Kathy*



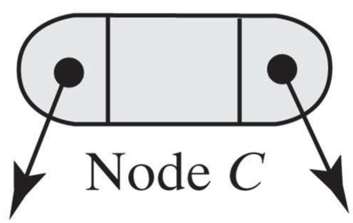
Removing a Value

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

(a) Two possible configurations of a tree's root with one child



(b) The tree after removing its root



Recursive Implementation

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

```
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 Implementation

- The public method `remove`

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

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

Recursive Implementation

- The private method `removeEntry`

```
// 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
```

Recursive Implementation

- The algorithm `removeFromRoot`

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

Recursive Implementation

- The private method `removeFromRoot`

```
// 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
```

Recursive Implementation

- The private method `findLargest`

```
// 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
```

Recursive Implementation

- The private method `removeLargest`

```
// 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
```

Iterative Implementation

- Pseudocode that describes `remove`

Algorithm **remove(anEntry)**
result = null
currentNode = *node that contains a match for* **anEntry**
parentNode = currentNode*'s parent*
if (currentNode != null) *// That is, if entry is found*
{
    result = currentNode*'s data (the anEntry to be removed from the tree)*
    // Case 1
    if (currentNode *has two children***)**
    {
        // Get node to remove and its parent
        nodeToRemove = *node containing* **anEntry** *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

Iterative `remove` Implementation (Part 1)

- The public method `remove`

```
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

      // 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();
```

Iterative `remove` Implementation (Part 2)

- The public method `remove`

```
      // 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
```

Iterative Implementation

- The private method `findNode`

```
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
```

Iterative Implementation

- Pseudocode for the private method `getNodeToRemove`

// Find the in-order 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*

Iterative `getNodeToRemove` Implementation

- Implementation of the private method `getNodeToRemove`

```
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
```

Iterative `removeNode` Implementation

- The private method `removeNode`

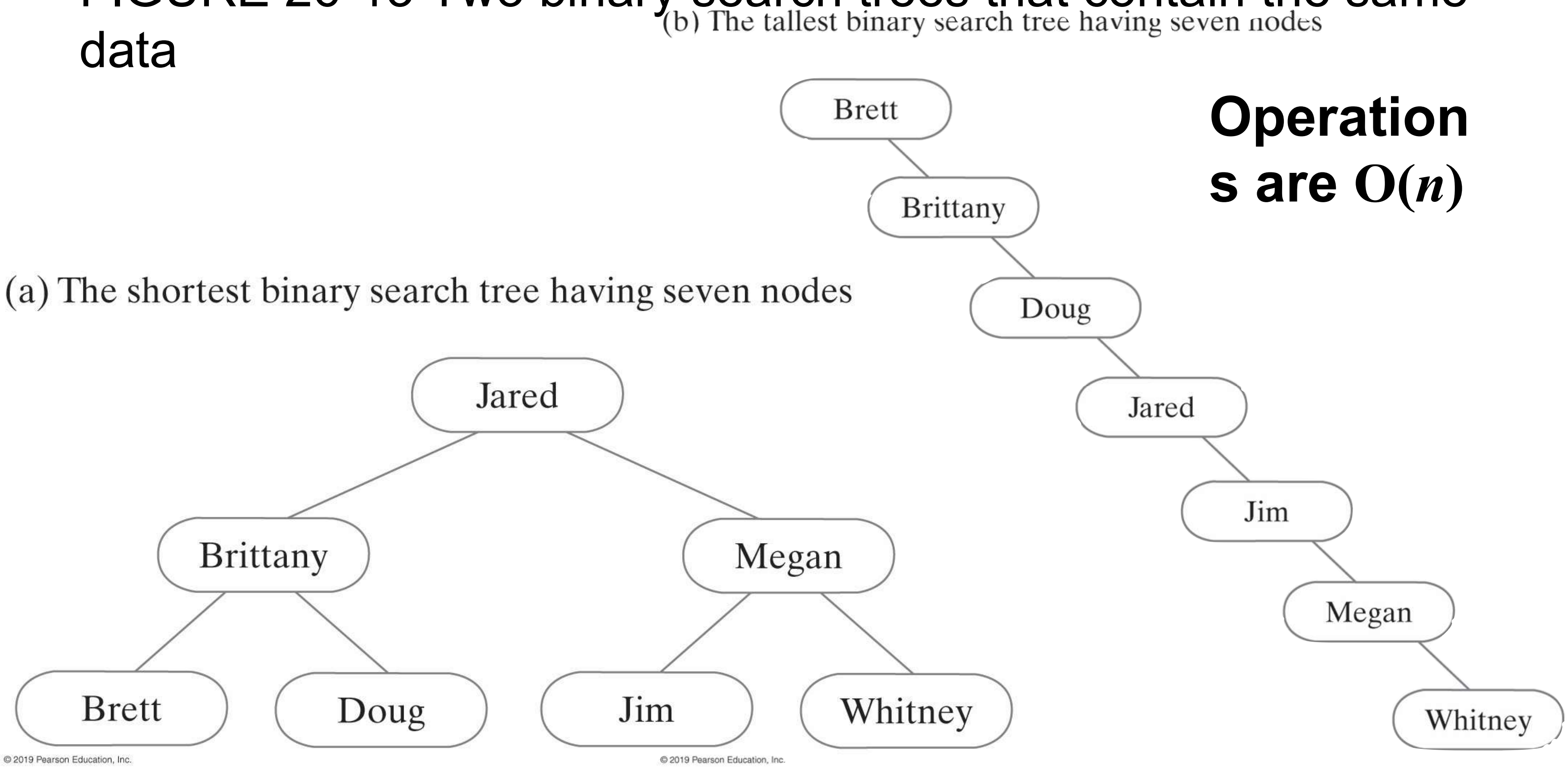
```
private void removeNode(BinaryNode<T> nodeToRemove,
                                        BinaryNode<T> parentNode)
{
  BinaryNode<T> childNode;

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

  // Assertion: if nodeToRemove is a leaf, childNode is null

  if (nodeToRemove == getRootNode())
    setRootNode(childNode);
  else if (parentNode.getLeftChild() == nodeToRemove)
    parentNode.setLeftChild(childNode);
  else
    parentNode.setRightChild(childNode);
} // end 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 26-13 Two binary search trees that contain the same data



Operations are O(n)

Operations are O(log n)