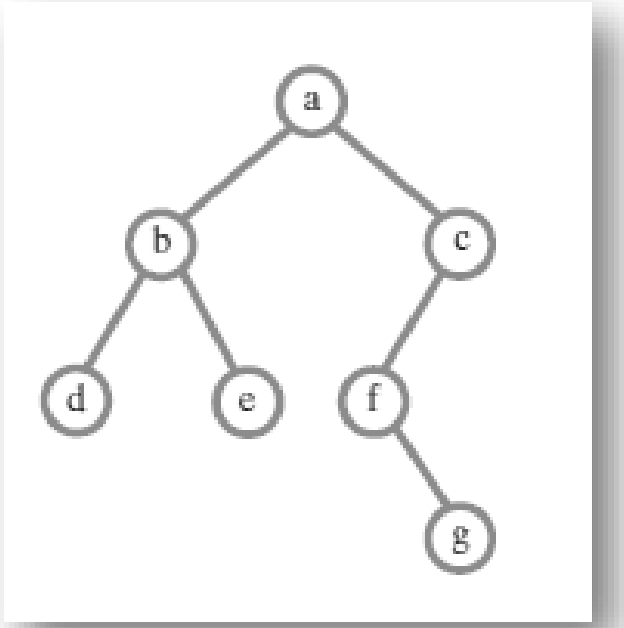
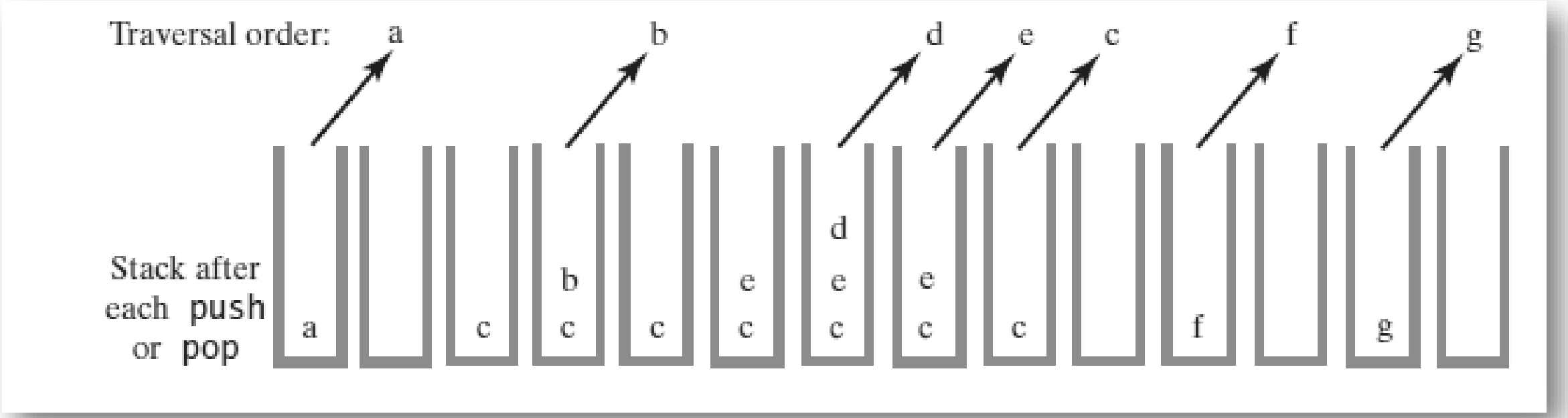
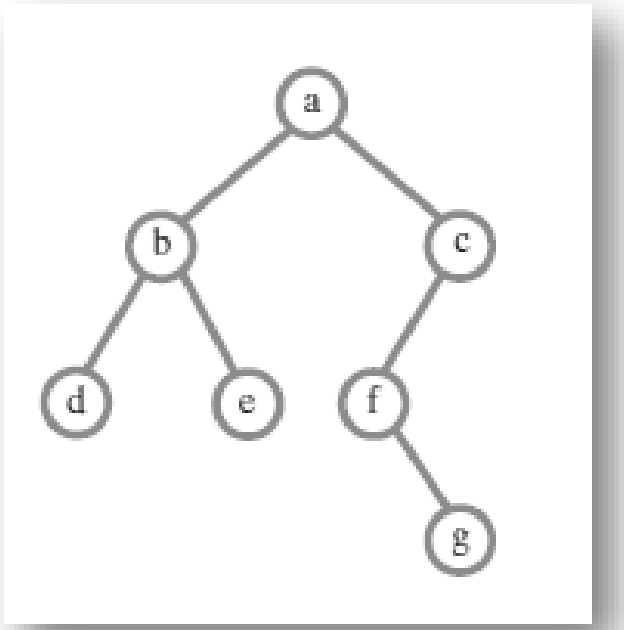
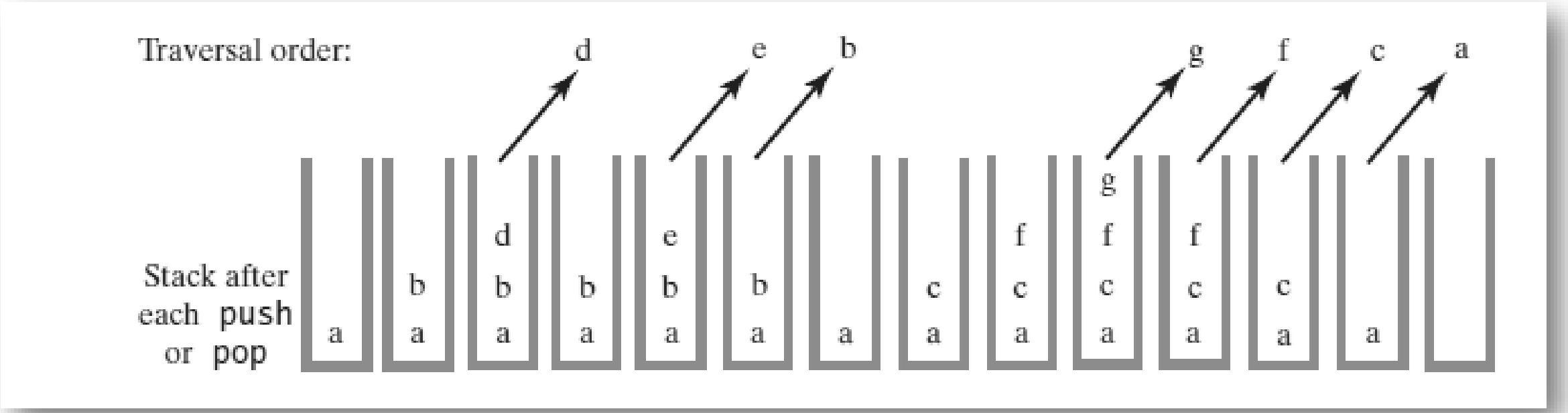
Iterative preorder traversal





Iterative postorder traversal





Revisiting Cloneable

```
public class Object
{
 ...
 protected native Object clone() throws CloneNotSupportedException;
 ...
}

public interface Cloneable
{
}
```

Cloning an Array

```
public interface Copyable extends Cloneable
{
 public Object clone();
}

public class AList<T extends Copyable> implements ListInterface<T>, Cloneable {}
```

Cloning an Array

```
public interface Copyable extends Cloneable
{
 public Object clone();
}

public class AList<T extends Copyable> implements ListInterface<T>, Cloneable {}
```

```
public interface Copyable extends Cloneable
{
 public Object clone();
}

public class AList<T extends Copyable> implements ListInterface<T>, Copyable {}
```

Cloning an Array (cont.)

```
public interface Copyable extends Cloneable
{
 public Object clone();
}

public interface CloneableListInterface<T>
       extends ListInterface<T>, Copyable // or Cloneable
{
}

public class AList<T extends Copyable> implements CloneableListInterface<T>{}
```

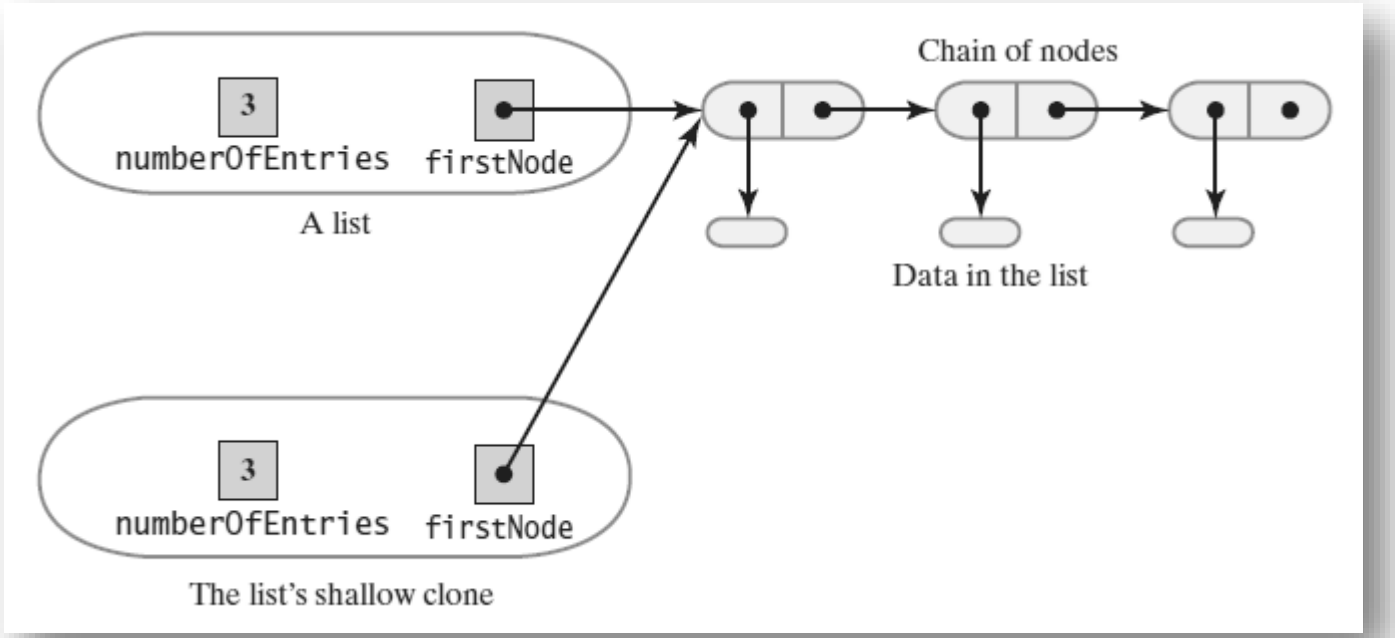
```
private T[] list;
...
T[] tempList = (T[])new Object[capacity];
...
```

```
private T[] list;
...
T[] tempList = (T[])new Object[capacity];
...
```

```
private T[] list;
...
T[] tempList = (T[])new Copyable[capacity];
...
```

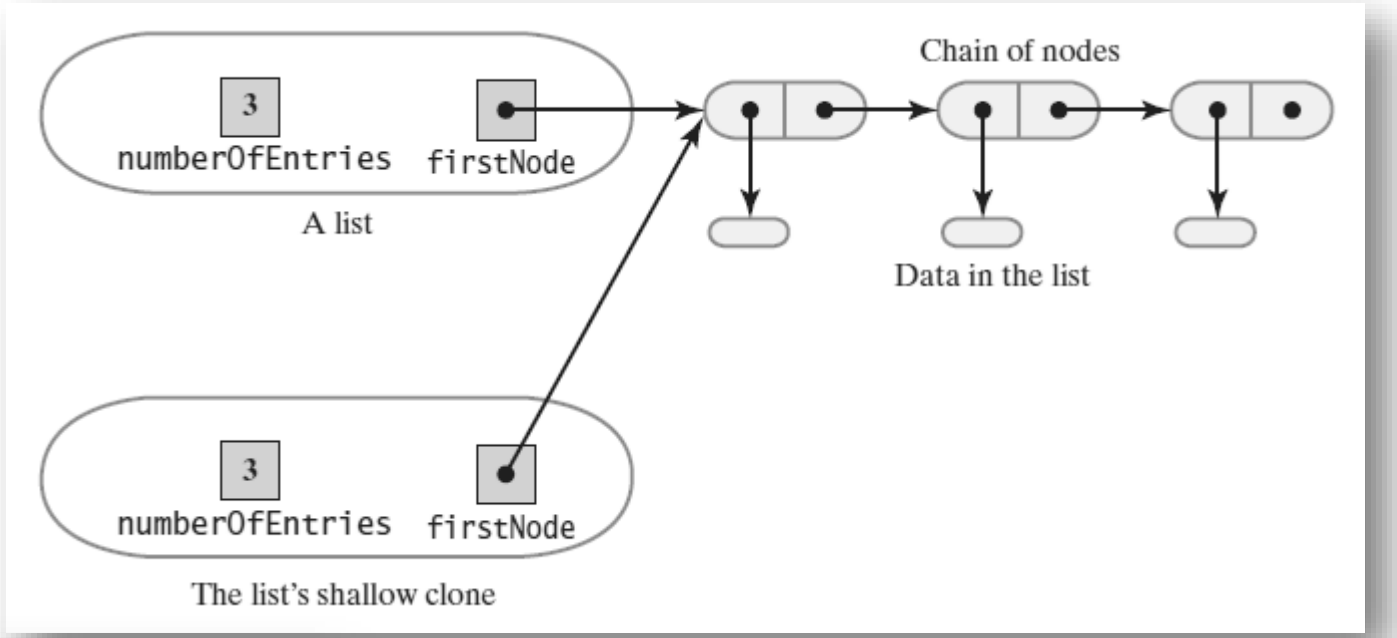
```
public Object clone()
{
 AList<T> theCopy = null;

 try
 {
  @SuppressWarnings("unchecked")
  AList<T> temp = (AList<T>)super.clone();
  theCopy = temp;
 }
 catch(CloneNotSupportedException ex)
 {
  throw new Error(ex.toString());
 }

 theCopy.list = list.clone();

 for(int index=1; index <= numberOfEntries; index++)
 {
  @SuppressWarnings("unchecked")
  T temp = (T)list[index].clone();
  theCopy.list[index] = temp;
 }
 return theCopy;
}
```

Cloning a Chain

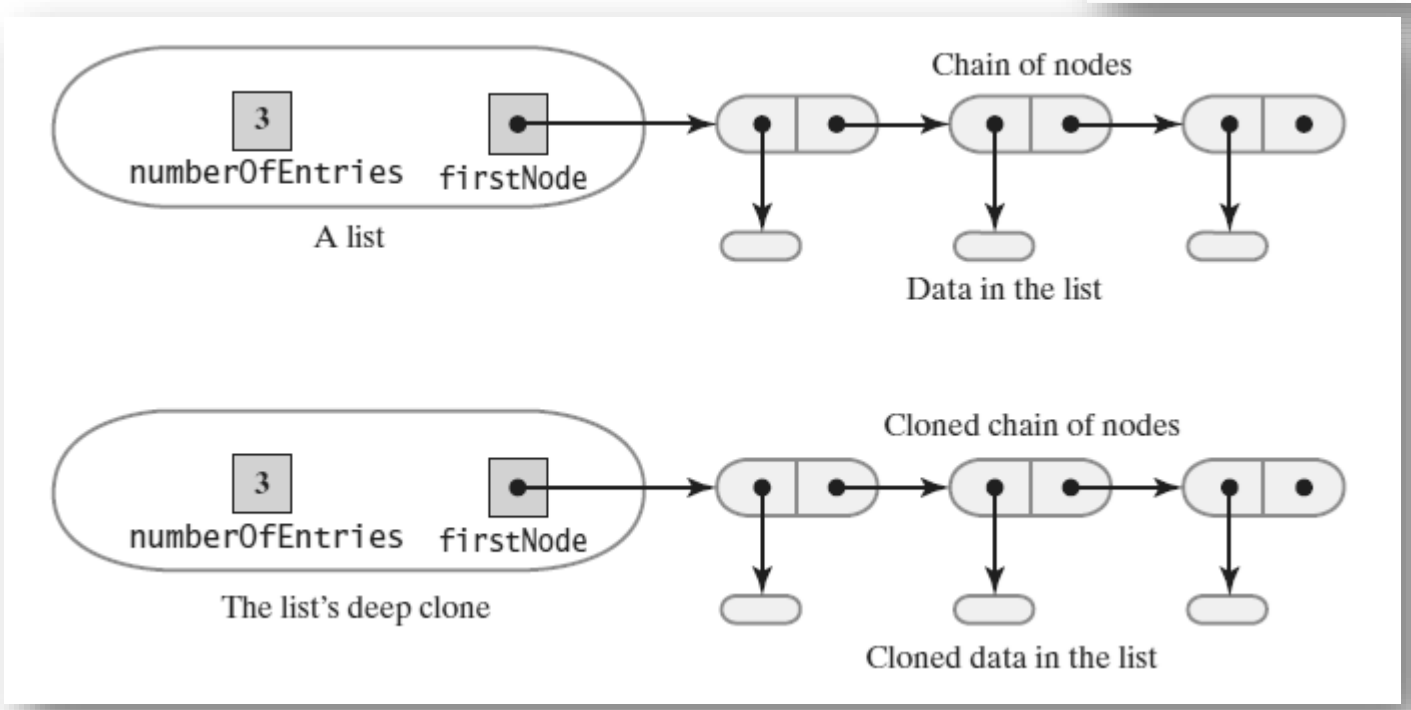


A list and its shallow clone: linked implementation.

Cloning a Chain



A list and its shallow clone: linked implementation.



A list and its deep clone: linked implementation.

Cloning a node

```
private class Node implements Cloneable
{
 private T    data;
 private Node next;
 ...
 protected Object clone()
 {
  Node theCopy = null;
  try
  {
   @SuppressWarnings("unchecked")
   Node temp = (Node)super.clone();
   theCopy = temp;
  }
  catch (CloneNotSupportedException e)
  {
   throw new Error(e.toString());
  }

  @SuppressWarnings("unchecked")
  T temp = (T)data.clone();
  theCopy.data = temp;
  theCopy.next = null;

  return theCopy;
 } // end clone
} // end Node
```

Cloning a chain (cont.)

```
public Object clone()
{
 LList<T> theCopy = null;
 try
 {
  @SuppressWarnings("unchecked")
  LList<T> temp = (LList<T>)super.clone();
  theCopy = temp;
 }
 catch (CloneNotSupportedException e)
 {
  throw new Error(e.toString());
 }
 if (firstNode == null)
 {
  theCopy.firstNode = null;
 }
 else
 {
  @SuppressWarnings("unchecked")
  Node temp = (Node)firstNode.clone();
  theCopy.firstNode = temp;

  Node newRef = theCopy.firstNode;
  Node oldRef = firstNode.getNextNode();
  for (int count = 2; count <= numberOfEntries; count++)
  {
   @SuppressWarnings("unchecked")
   Node temp2 = (Node)oldRef.clone();
   newRef.setNextNode(temp2);
   newRef = newRef.getNextNode();
   oldRef = oldRef.getNextNode();
  } // end for
 } // end if
 return theCopy;
} // end clone
```

Cloning a Binary Node

```
public Object clone()
{
  BinaryNode<T> theCopy = null;
  try
  {
    @SuppressWarnings("unchecked")
    BinaryNode<T> temp = (BinaryNode<T>) super.clone();
    theCopy = temp;
  } catch (CloneNotSupportedException e)
  {
    throw new Error("BinaryNode cannot clone: " + e.toString());
  }

  theCopy.data = (T) data.clone();

  if (left != null)
    theCopy.left = (BinaryNode<T>) left.clone();

  if (right != null)
    theCopy.right = (BinaryNode<T>) right.clone();

  return theCopy;
} // end clone
```

```
package java.util;
...
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
 ...
 /**
  * Returns a shallow copy of this <tt>ArrayList</tt> instance.
  * (The elements themselves are not copied.)
  *
  * @return a clone of this <tt>ArrayList</tt> instance
  */
 public Object clone() {
  try {
   ArrayList<?> v = (ArrayList<?>) super.clone();
   v.elementData = Arrays.copyOf(elementData, size);
   v.modCount = 0;
   return v;
  }
  catch (CloneNotSupportedException e) {
   // this shouldn't happen, since we are Cloneable
   throw new InternalError(e);
  }
 }
 ...
}
```

jdk1.8.0_102 src.zip

```
package java.util;
...
public class LinkedList<E> extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
 ...
 @SuppressWarnings("unchecked")
 private LinkedList<E> superClone() {
  try {
   return (LinkedList<E>) super.clone();
  }
  catch (CloneNotSupportedException e) {
   throw new InternalError(e);
  }
 }
```

```
 /**
  * Returns a shallow copy of this {@code LinkedList}.
  * (The elements themselves are not cloned.)
  *
  * @return a shallow copy of this {@code LinkedList}
  * instance
  */
 public Object clone() {
  LinkedList<E> clone = superClone();

  // Put clone into "virgin" state
  clone.first = clone.last = null;
  clone.size = 0;
  clone.modCount = 0;

  // Initialize clone with our elements
  for (Node<E> x = first; x != null; x = x.next)
   clone.add(x.item);

  return clone;
 }
 ...
}
```

jdk1.8.0_102 src.zip

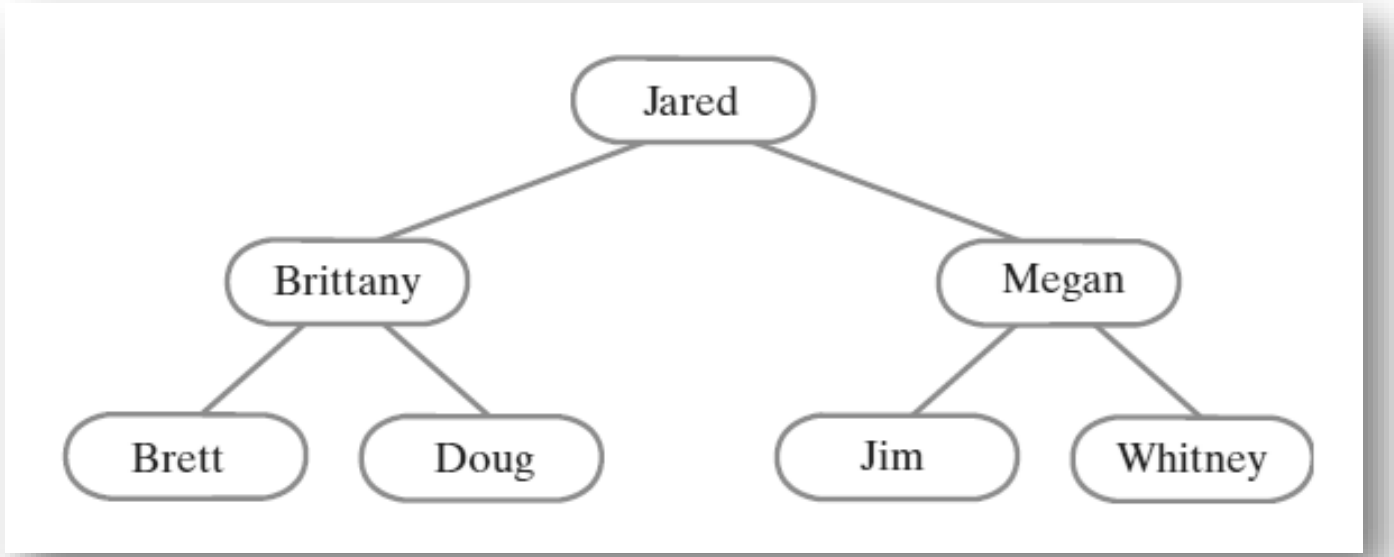
A Binary Search Tree Implementation

Intro

- A **binary search tree** is a binary tree whose nodes contain Comparable objects and are organized as follows. For each node in the three,
  - The data in a node is greater than the data in the node's left subtree
  - The data in a node is less than the data in the node's right subtree

Intro

- A **binary search tree** is a binary tree whose nodes contain Comparable objects and are organized as follows. For each node in the three,
  - The data in a node is greater than the data in the node's left subtree
  - The data in a node is less than the data in the node's right subtree



```java
package TreePackage;
import java.util.Iterator;
/**
 * 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 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);
```

```java
public interface TreeInterface<T>
{
 public T getRootData();
 public int getHeight();
 public int getNumberOfNodes();
 public boolean isEmpty();
 public void clear();
} // end TreeInterface
```

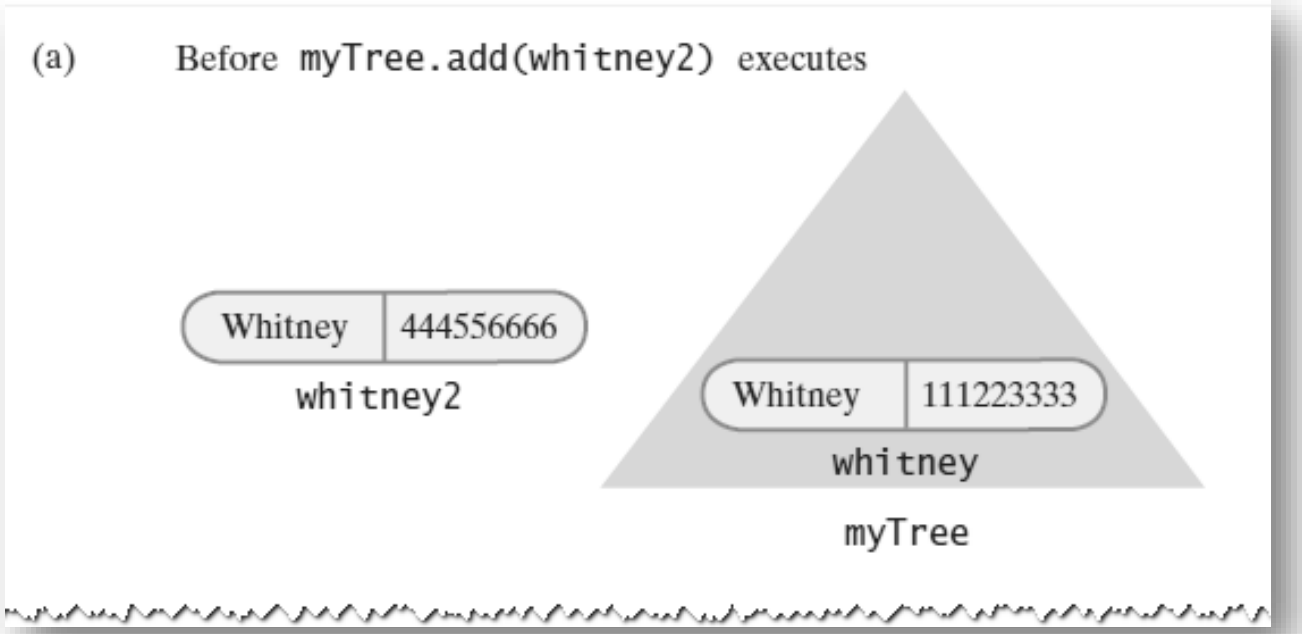
```java
 /**
  * Adds a new entry to this tree, if it does not match an existing object in
  * the tree. Otherwise, replaces the existing object with the new entry.
  *
  * @param newEntry An object to be added to the tree.
  * @return Either null if newEntry was not in the tree already, or the existing
  *         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();
} // end SearchTreeInterface
```

Suppose that compareTo bases its comparison only on the name field.

```
SearchTreeInterface<Person> myTree = new BinarySearchTree<>();
Person whitney = new Person("Whitney", "111223333");
Person returnValue = myTree.add(whitney);

Person whitney2 = new Person("Whitney", "444556666");
returnValue = myTree.add(whitney2);
```

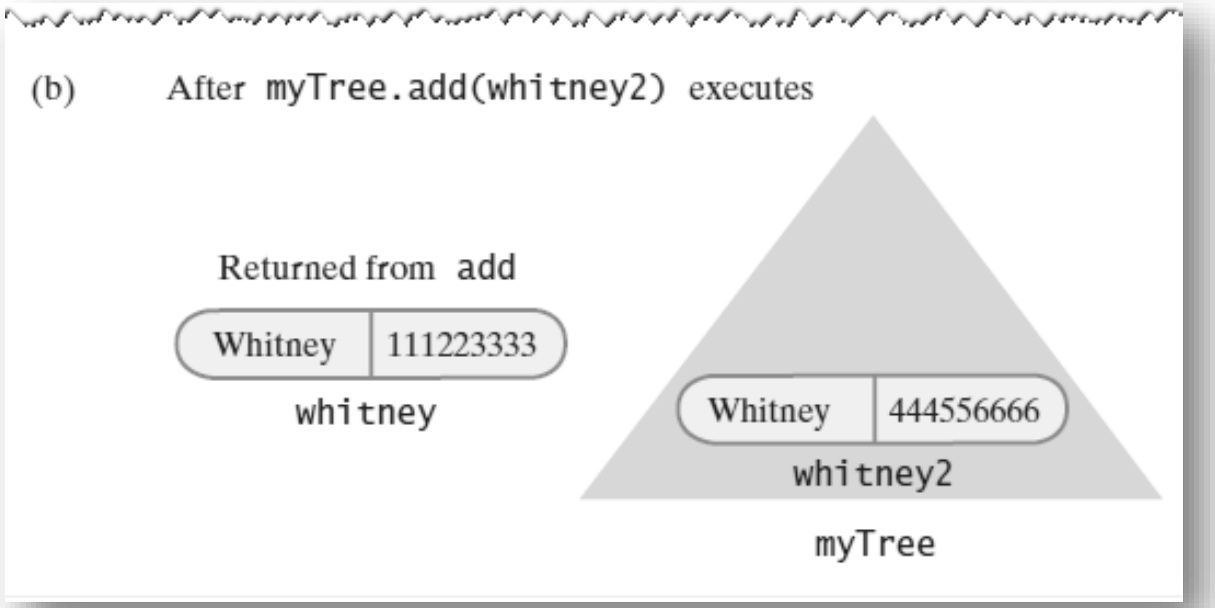
Suppose that compareTo bases its comparison only on the name field.

```
SearchTreeInterface<Person> myTree = new BinarySearchTree<>();
Person whitney = new Person("Whitney", "111223333");
Person returnValue = myTree.add(whitney);

Person whitney2 = new Person("Whitney", "444556666");
returnValue = myTree.add(whitney2);
```



Suppose that compareTo bases its comparison only on the name field.

```
SearchTreeInterface<Person> myTree = new BinarySearchTree<>();
Person whitney = new Person("Whitney", "111223333");
Person returnValue = myTree.add(whitney);

Person whitney2 = new Person("Whitney", "444556666");
returnValue = myTree.add(whitney2);
```



Suppose that compareTo bases its comparison only on the name field.

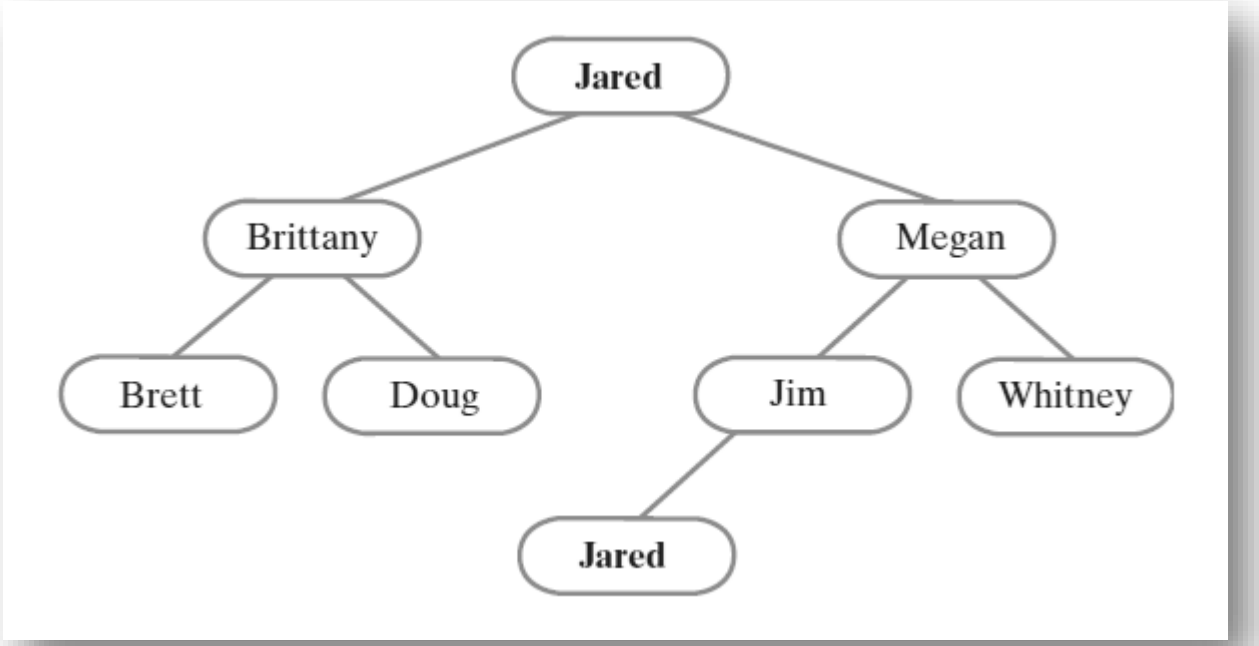
```
SearchTreeInterface<Person> myTree = new BinarySearchTree<>();
Person whitney = new Person("Whitney", "111223333");
Person returnValue = myTree.add(whitney);

Person whitney2 = new Person("Whitney", "444556666");
returnValue = myTree.add(whitney2);

returnValue = myTree.getEntry(whitney);
```

Suppose that compareTo bases its comparison only on the name field.

```
SearchTreeInterface<Person> myTree = new BinarySearchTree<>();
Person whitney = new Person("Whitney", "111223333");
Person returnValue = myTree.add(whitney);

Person whitney2 = new Person("Whitney", "444556666");
returnValue = myTree.add(whitney2);

returnValue = myTree.getEntry(whitney);

returnValue = myTree.remove(whitney);
```

Duplicate entries

- Definition could be modified as follows, for example. For each node in a binary search tree,
  - The data in a node is greater than the data in the node's left subtree
  - The data in a node is less than ***or equal*** *to* the data in the node's right subtree



Duplicate entries

- Definition could be modified as follows, for example. For each node in a binary search tree,
  - The data in a node is greater than the data in the node's left subtree
  - The data in a node is less than ***or equal*** *to* the data in the node's right subtree

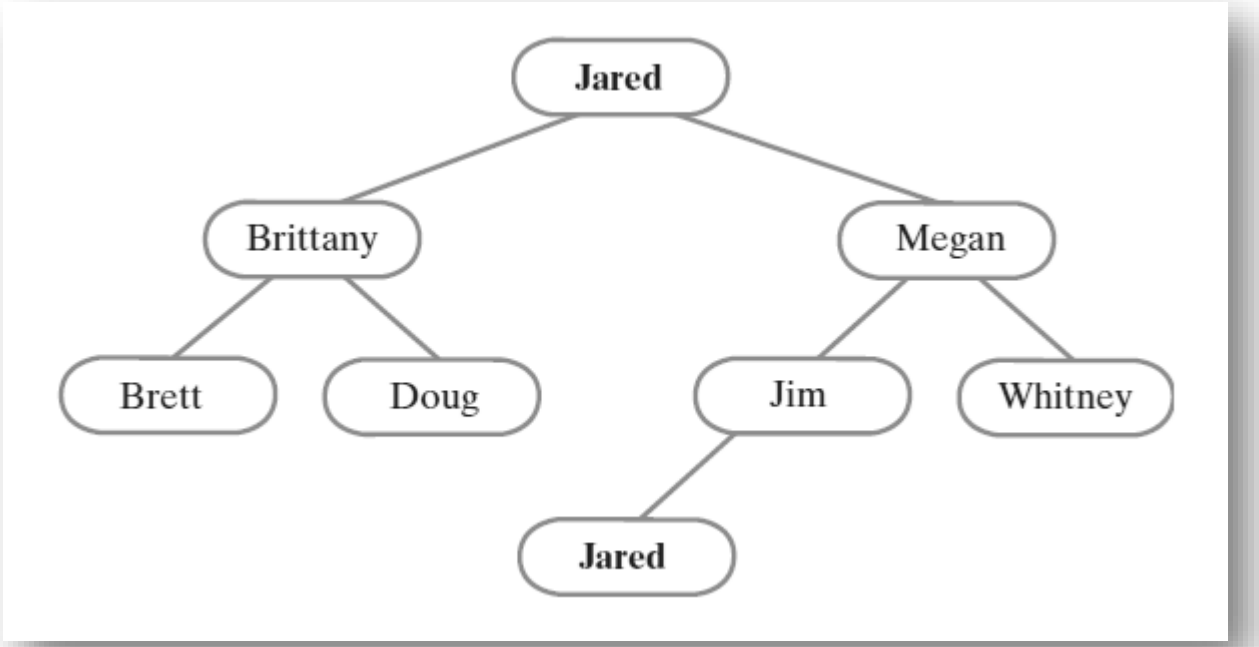
With duplicate entries permitted, the **add** method has less to do.

- But, which entry will **getEntry** retrieve?
- Will the method **remove** delete the first occurrence of an entry or all occurrences?

We will not consider duplicates in following implementation.

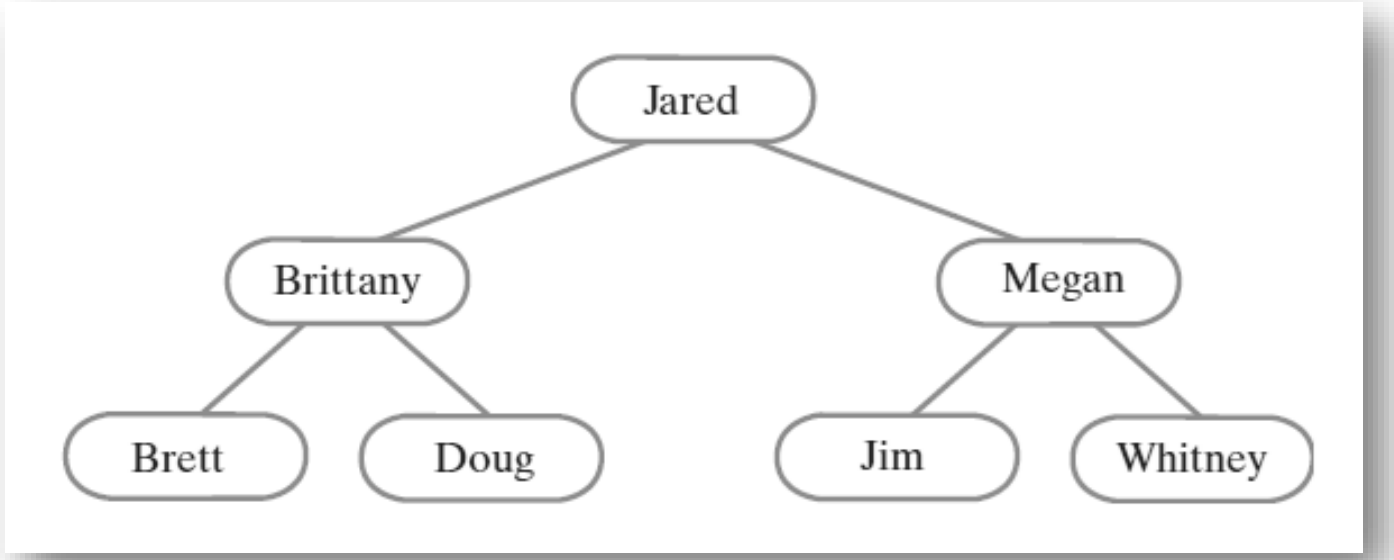
Question

- If you add a duplicate entry *Megan* to the binary search tree below as a leaf, where should you place the new node?



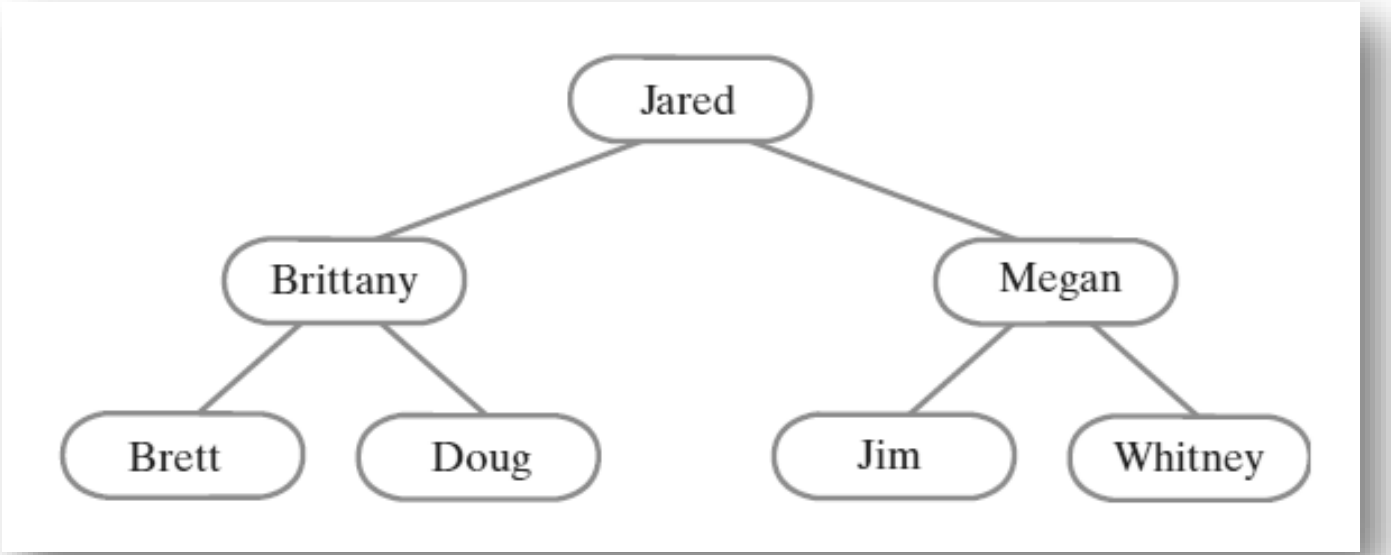
Question

- Add the name *Miguel* to the binary search tree below, and then add *Nancy*.



Question

- Add the name *Miguel* to the binary search tree below, and then add *Nancy*. Now, instead add first *Nancy* and then add *Miguel*. Does the order in which you add the names affect the tree that results?



```java
/**
 * A class that implements the ADT binary search tree by extending BinaryTree.
 * Recursive version.
 */

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<>(rootEntry));
 } // end constructor

 public void setTree(T rootData) { throw new UnsupportedOperationException(); } // end set


 public void setTree(T rootData, BinaryTreeInterface<T> leftTree, BinaryTreeInterface<T> r
 {
  throw new UnsupportedOperationException();
 } // end setTree
```

The search algorithm

```
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)
```

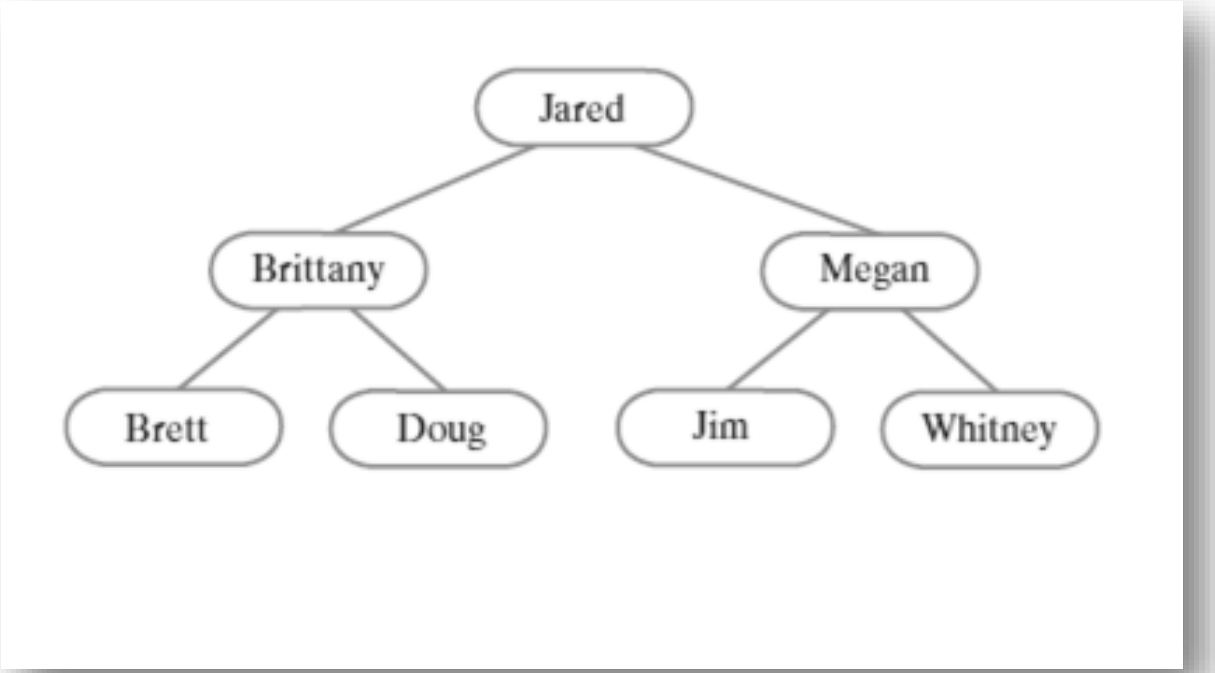
```java
public T getEntry(T entry)
{
 return findEntry(getRootNode(), entry);
} // end getEntry
```

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

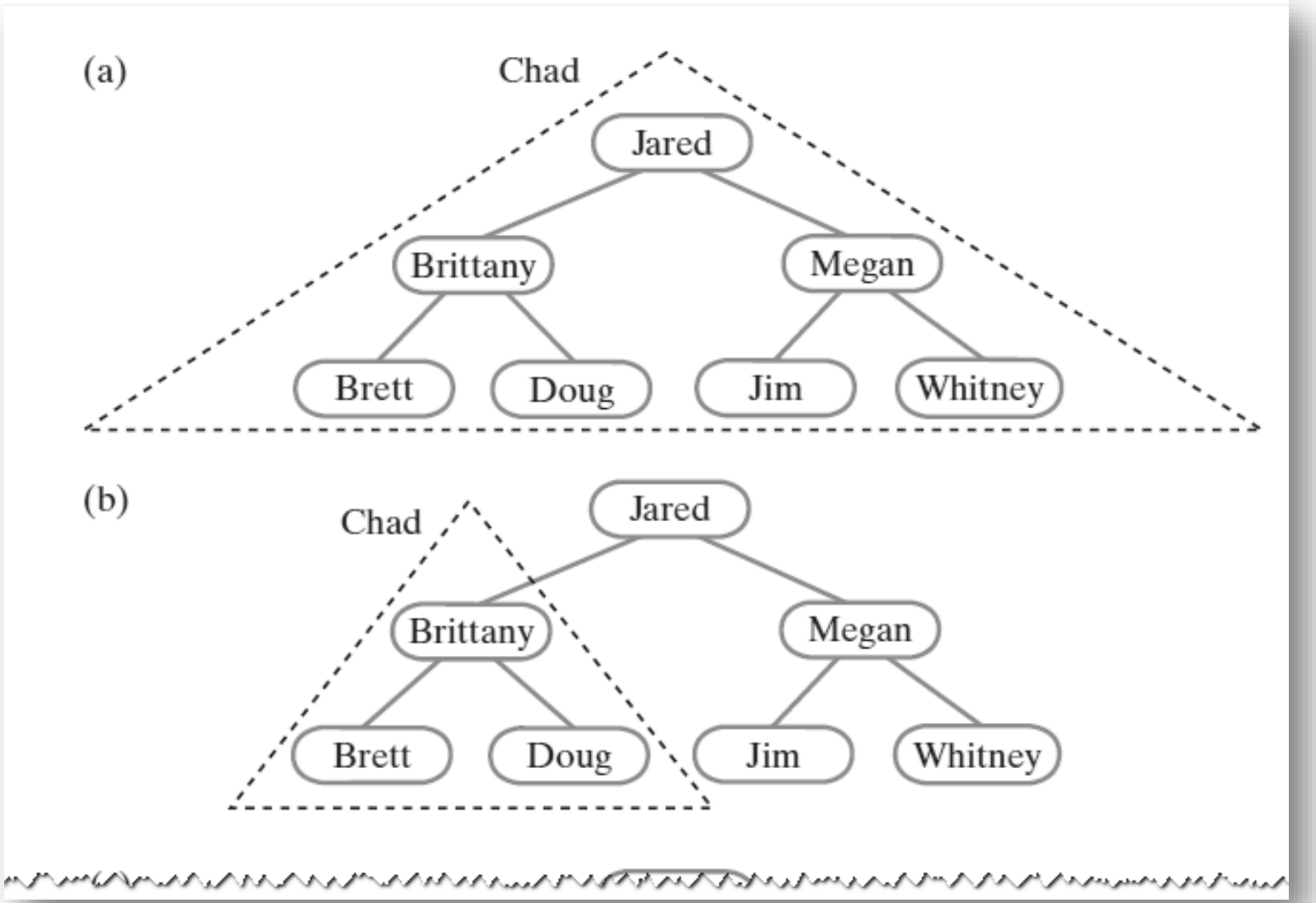
```java
private T findEntry(BinaryNode<T> rootNode, T entry)
{
 T result = null;

 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
```

Adding an Entry

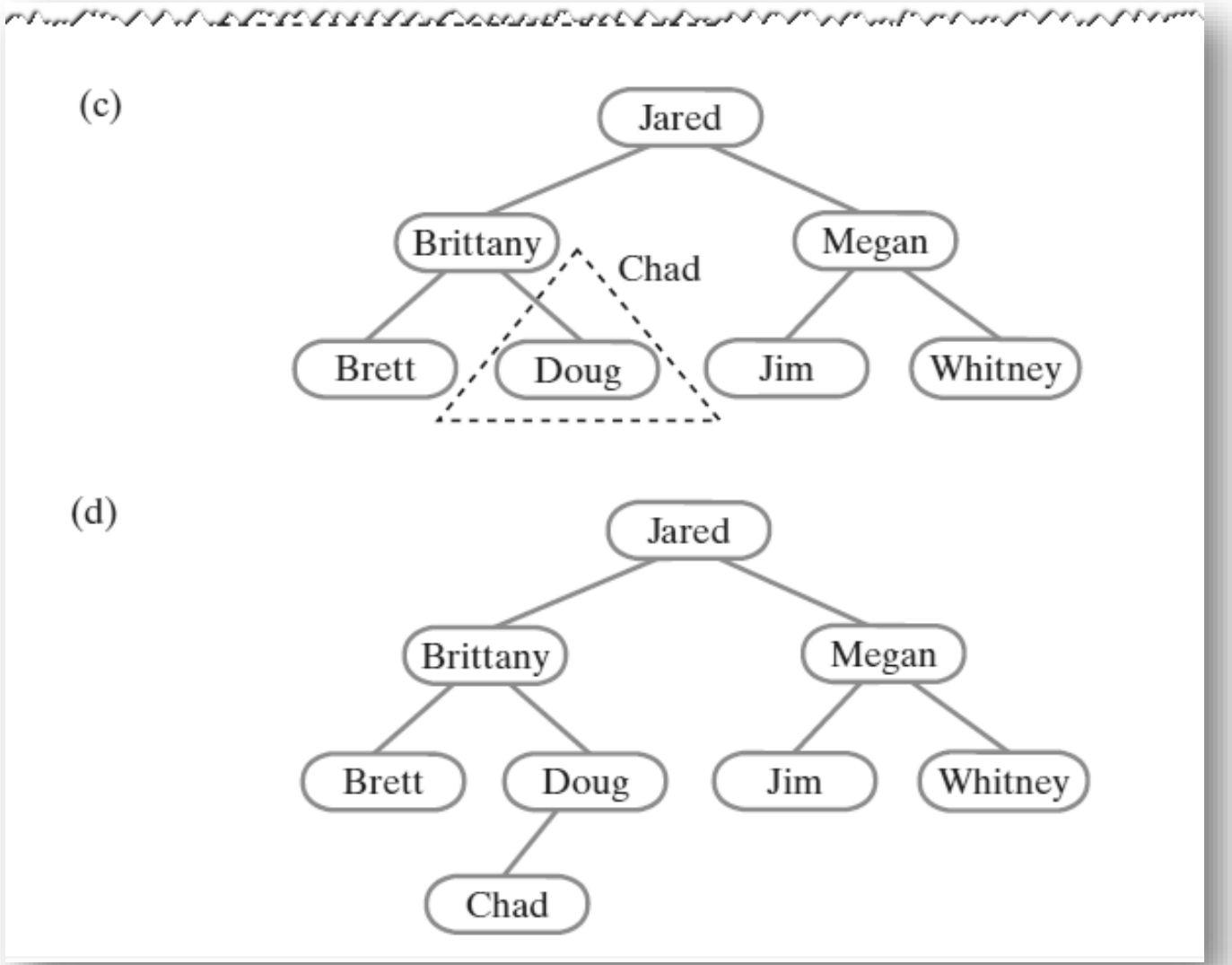
Add a *Chad* to below tree



Adding an Entry:
A recursive implementation

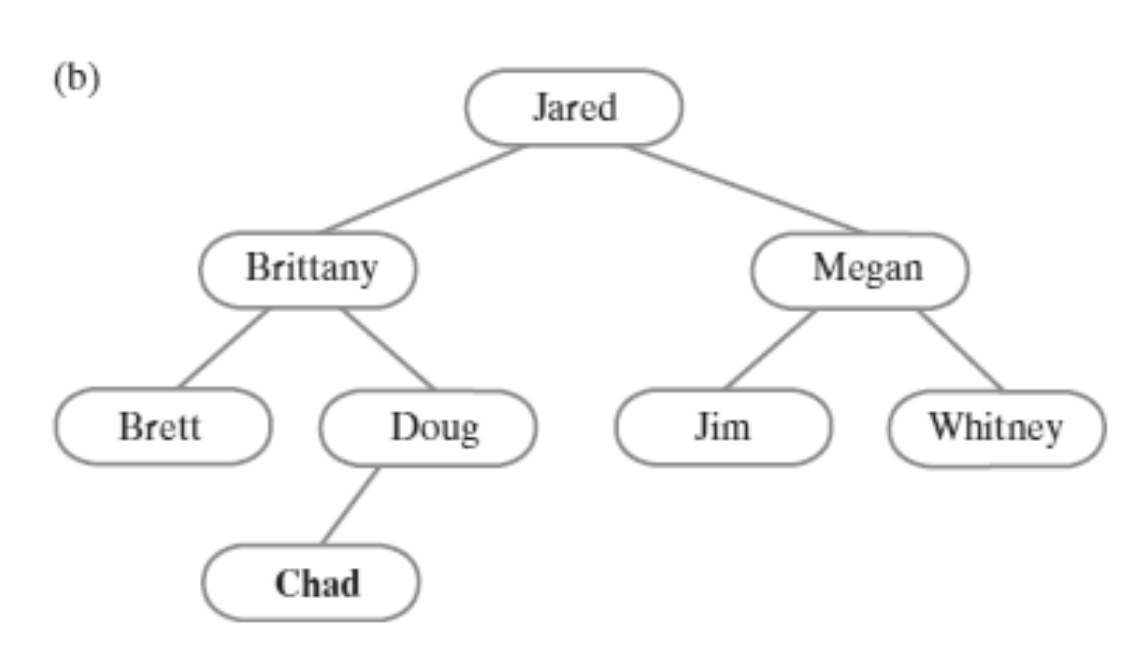


Adding an Entry:
A recursive implementation



Question

- Add the names *Chris*, *Jason*, and *Kelley* to the binary search tree below.



```java
// 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));
 }
 else
 {
  assert comparison > 0;

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

 return result;
} // end addEntry
```

```java
// 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.getDat

 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));
 }
 else
 {
  assert comparison > 0;

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

 return result;
} // end addEntry
```

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

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

 return result;
} // end add
```

An iterative implementation of the method **addEntry**

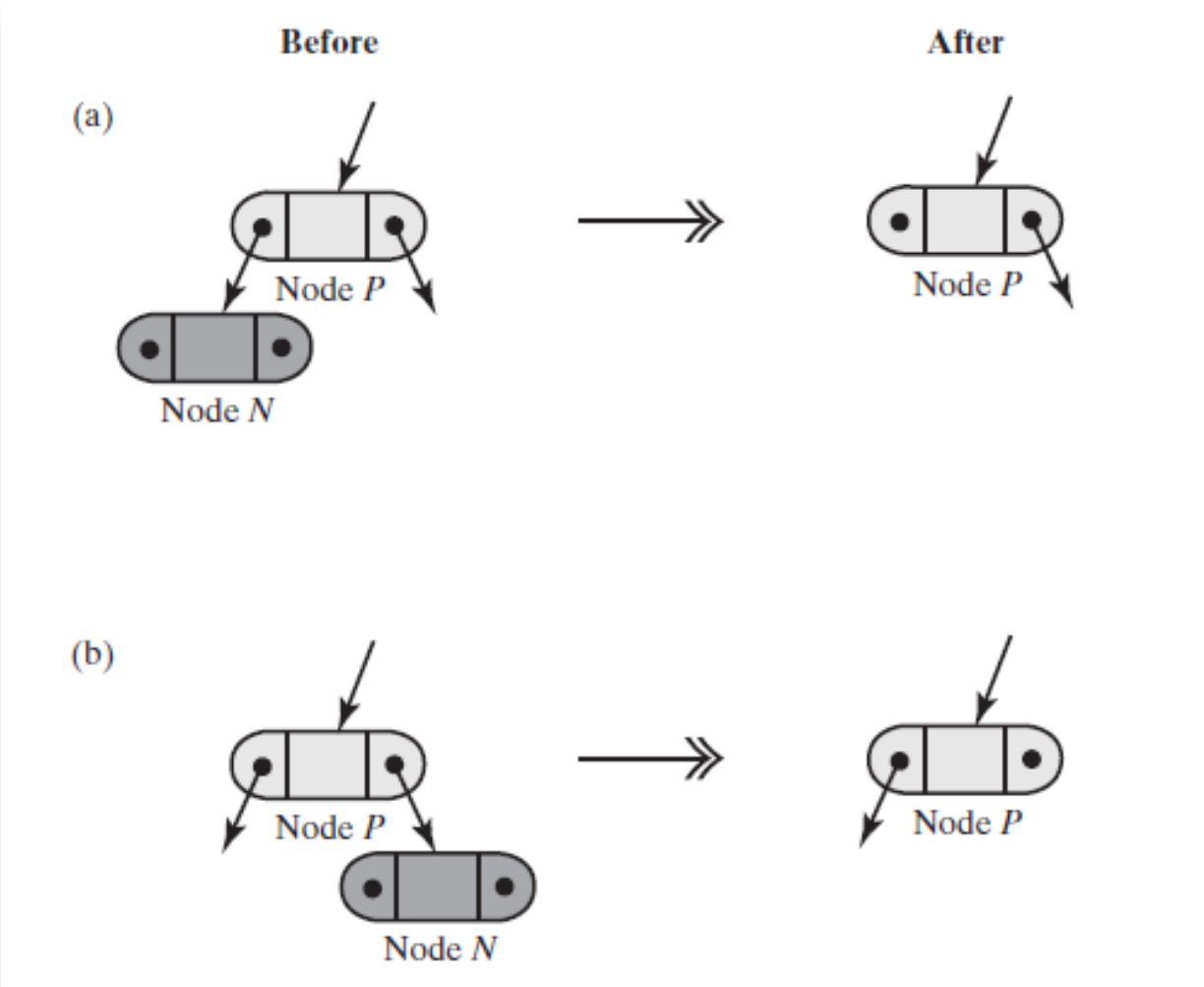
```
private T addEntry(T newEntry)
{
 BinaryNode<T> currentNode = getRootNode();
 assert currentNode != null;
 T result = null;
 boolean found = false;

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

  if (comparison == 0)
  {
   found = true;
   result = currentEntry;
   currentNode.setData(newEntry);
  }
  else if (comparison < 0)
  {
   if (currentNode.hasLeftChild())
    currentNode = currentNode.getLeftChild();
   else
   {
    found = true;
    currentNode.setLeftChild(new BinaryNode<>(newEntry));
   } // end if
  }
  else
  {
   assert comparison > 0;

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

 return result;
} // end addEntry
```

Removing an Entry

- Removing an entry, if found, is somewhat more involved than adding an entry, as the required logic depends upon how many children belong to the node containing the entry.

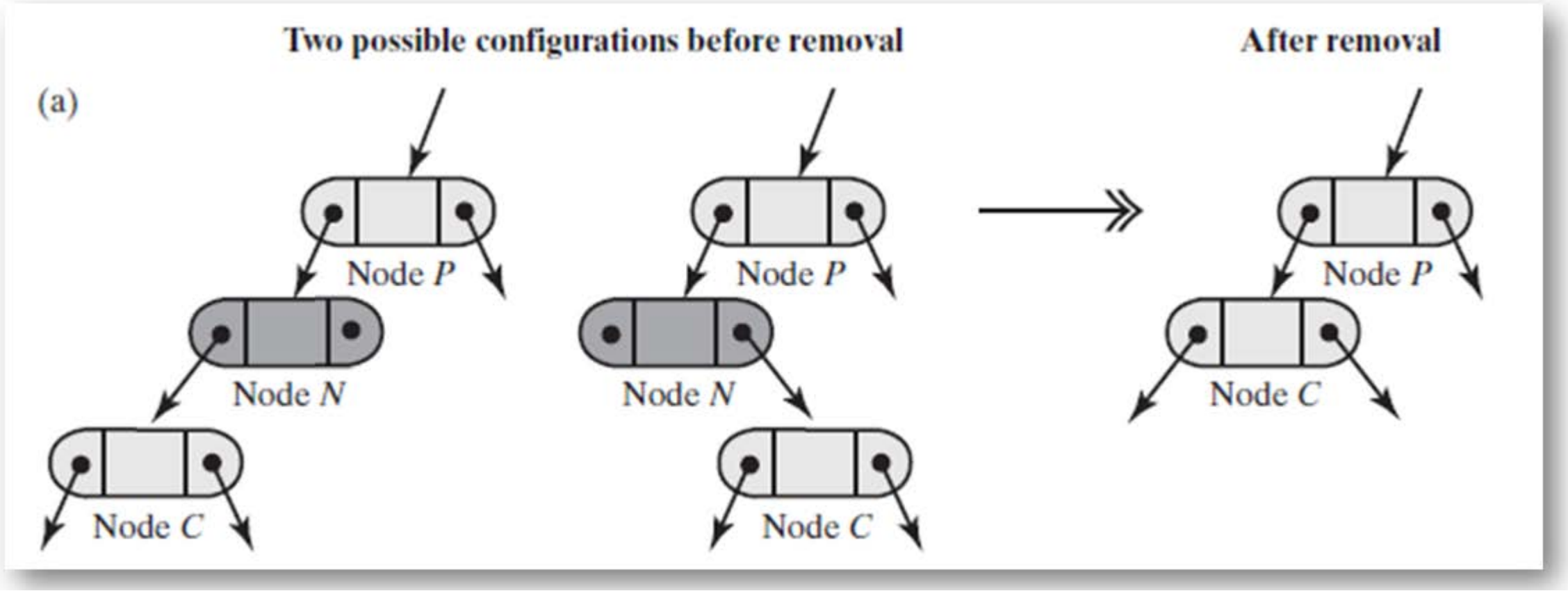
- We have three possibilities:
  1. The node has no children – it is a leaf
  2. The node has one child
  3. The node has two children

1. Removing an **Entry** whose **Node** is a **Leaf**

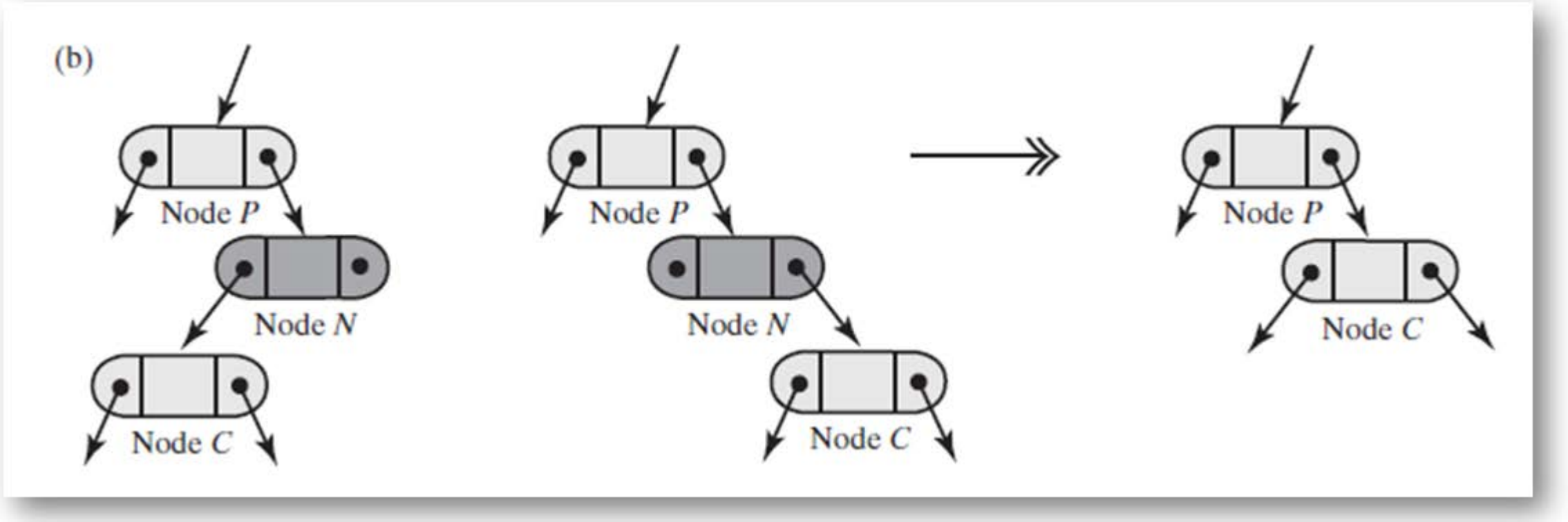


Removing a leaf node *N* from its parent node *P* when *N* is **(a)** a left child; **(b)** a right child.

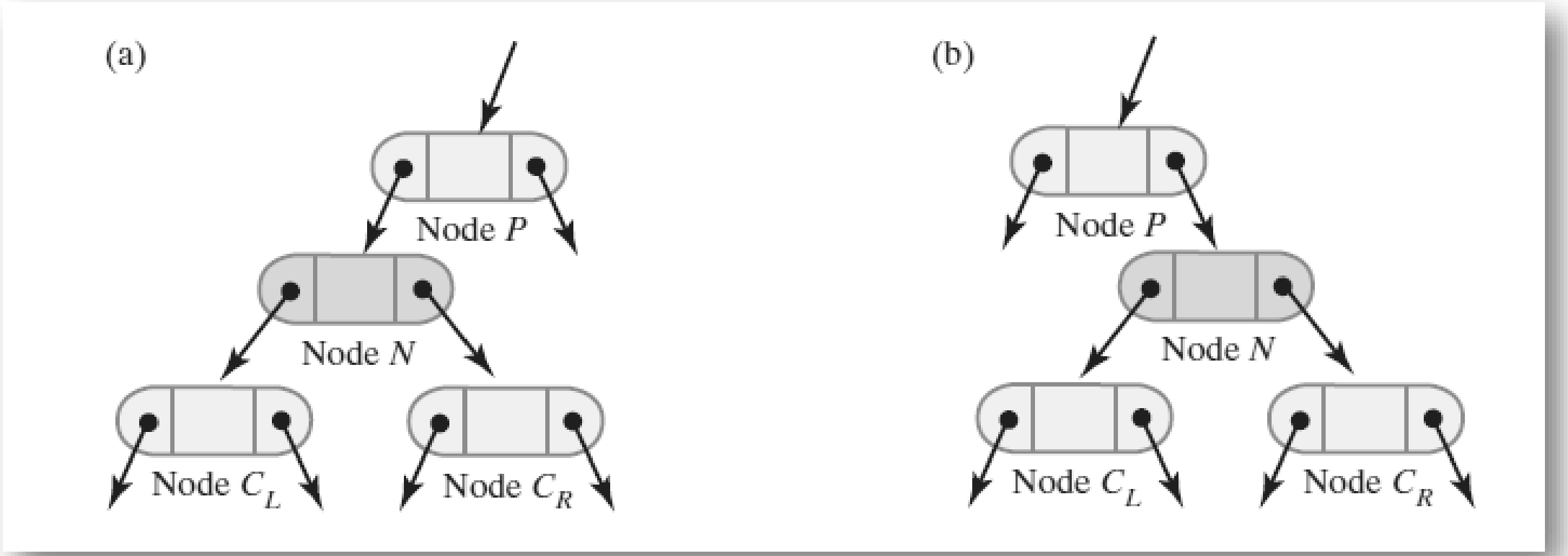
2. Removing an **Entry** whose **Node** has **One Child**



2. Removing an **Entry** whose **Node** has **One Child** (cont.)

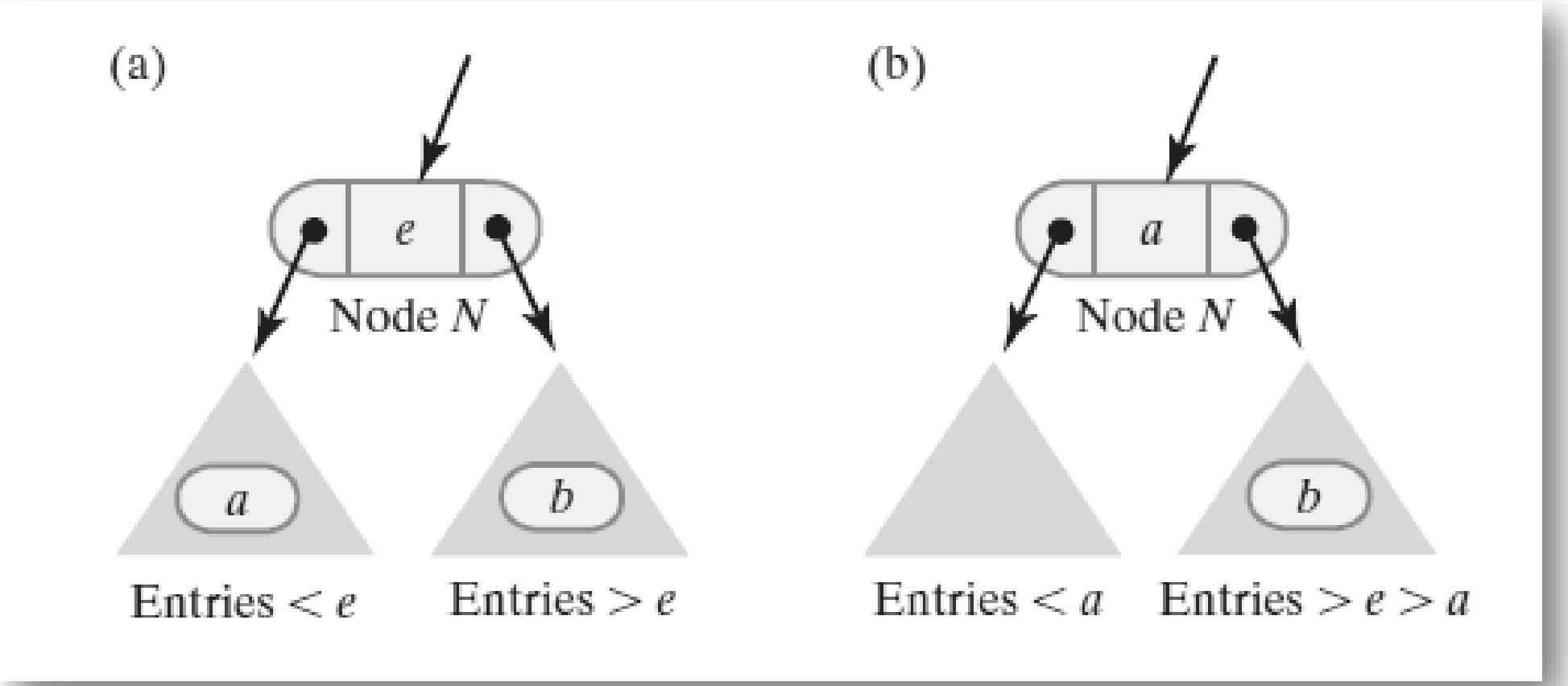


3. Removing an **Entry** whose **Node** has **Two Children**



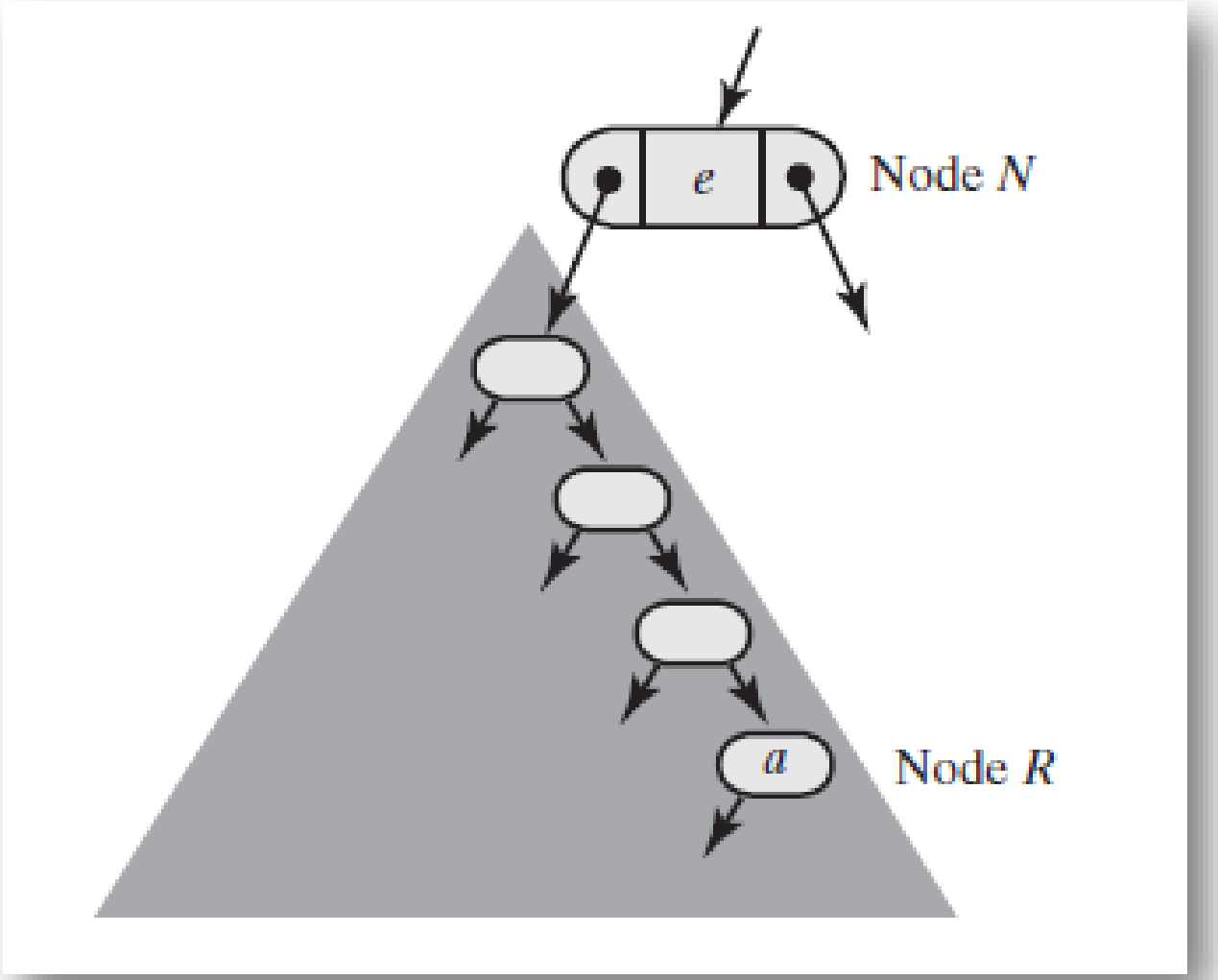
3. Removing an **Entry** whose **Node** has **Two Children** (cont.)

… **a** < **e** < **b** … An inorder traversal of the tree would visit these entries in this same order. Thus, **a** is called the *inorder predecessor* of **e**, and **b** is the *inorder successor* of **e**.



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**.

3. Removing an **Entry** whose **Node** has **Two Children**: Locating the entry **a**



3. Removing an **Entry** whose **Node** has **Two Children** (cont.)

Algorithm Remove the entry e from a node N that has two children

Find the rightmost node R in N's left subtree
Replace the entry in node N with the entry that is in node R
Delete node R

Algorithm Remove the entry e from a node N that has two children

Find the leftmost node L in N's right subtree
Replace the entry in node N with the entry that is in node L
Delete node L

References

- F. M. Carrano & T. M. Henry, “Data Structures and Abstractions with Java”, 4th Ed., 2015. Pearson Education, Inc.