A Binary Search Tree Implementation

Cont.

```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
```

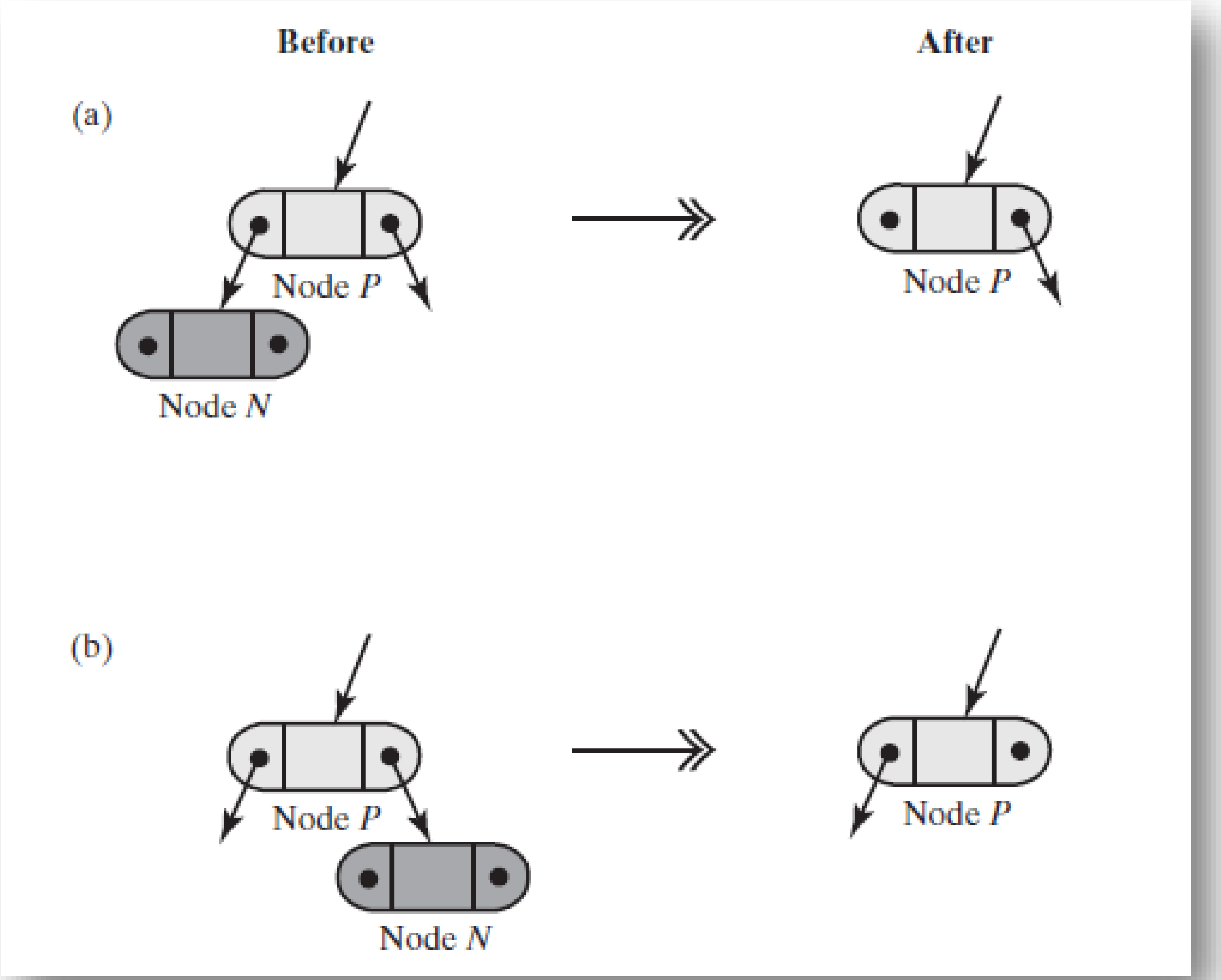
```java
public T add(T newEntry)
{
 T result = null;

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

 return result;
} // end add
```

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.

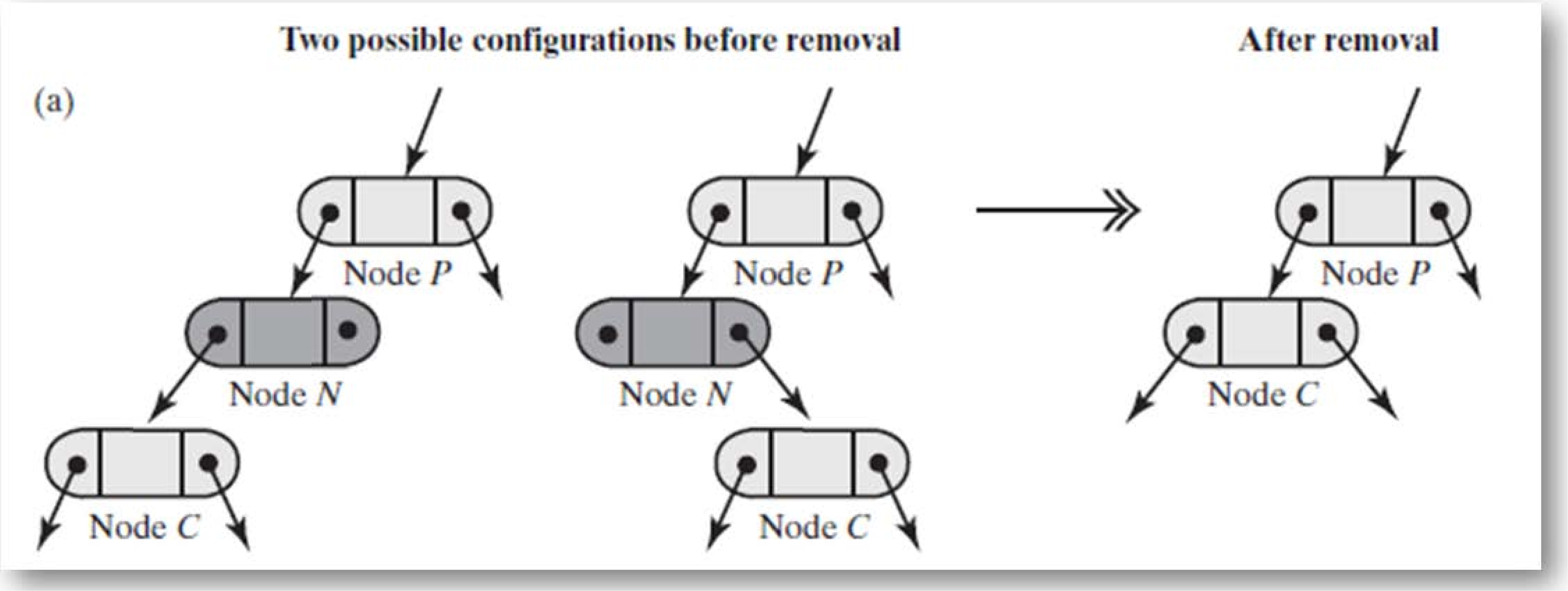
- We have three possibilities:
  1. The node has no children – it is a leaf
  2. The node has one child
  3. The node has two children

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

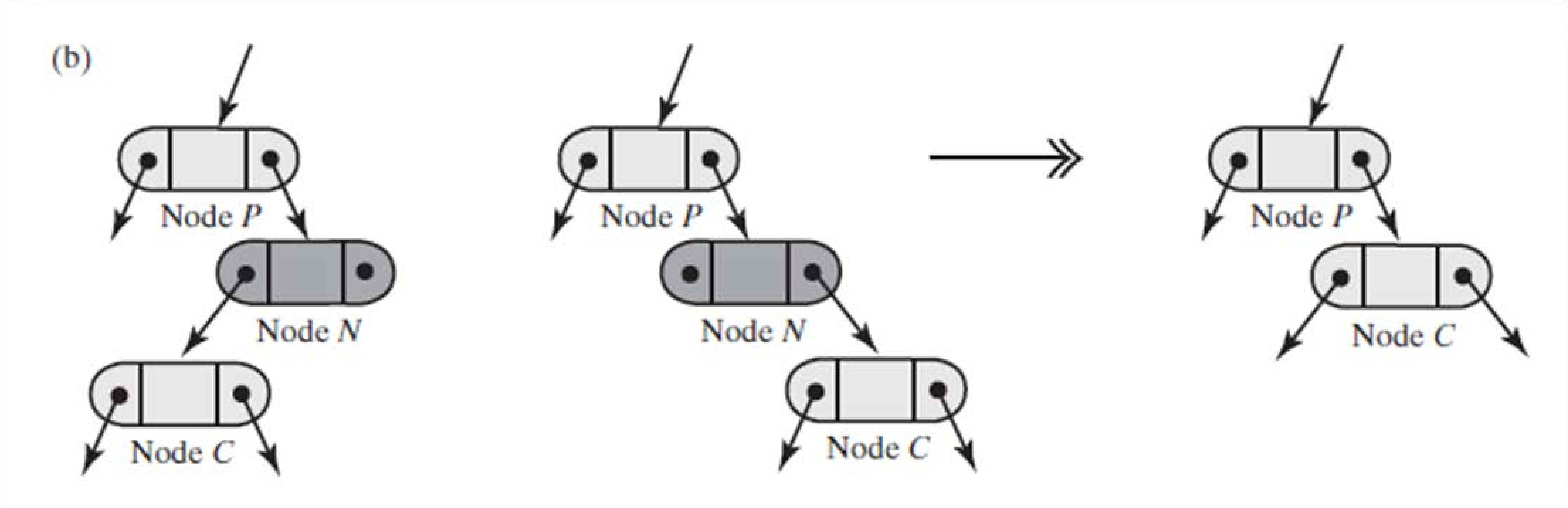


Removing a leaf node *N* from its parent node *P* when *N* is **(a)** a left child; **(b)** a right child.

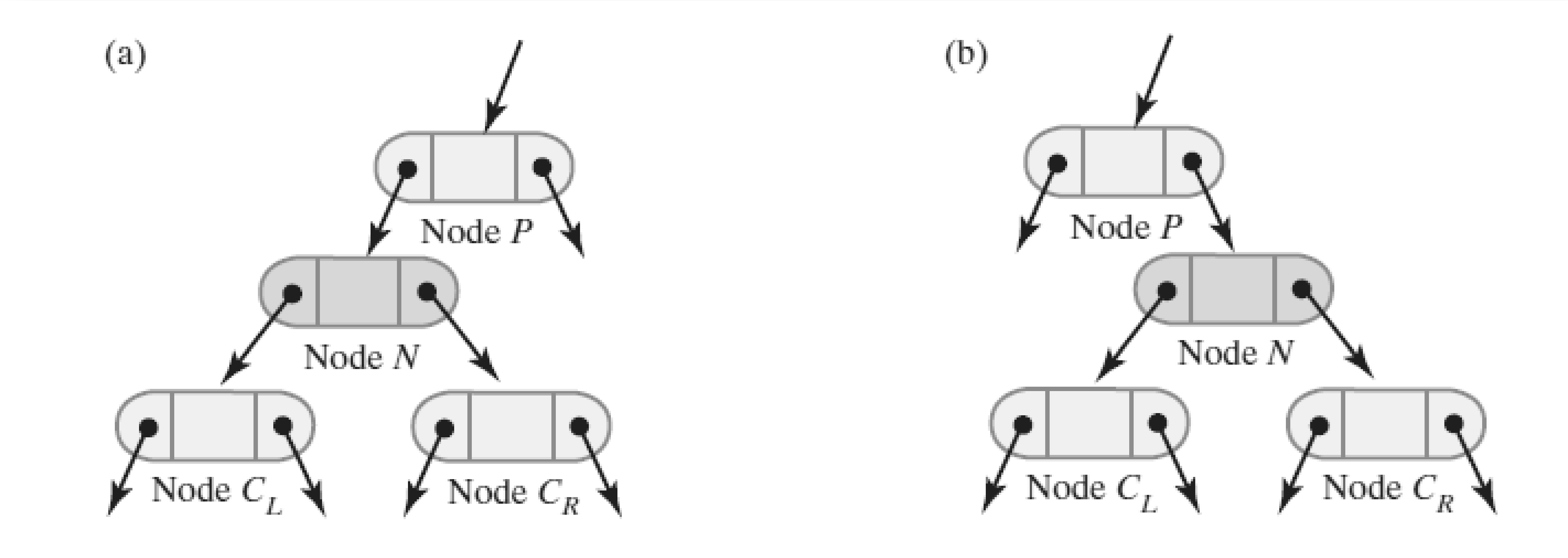
2. Removing an **Entry** whose **Node** has **One Child**



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

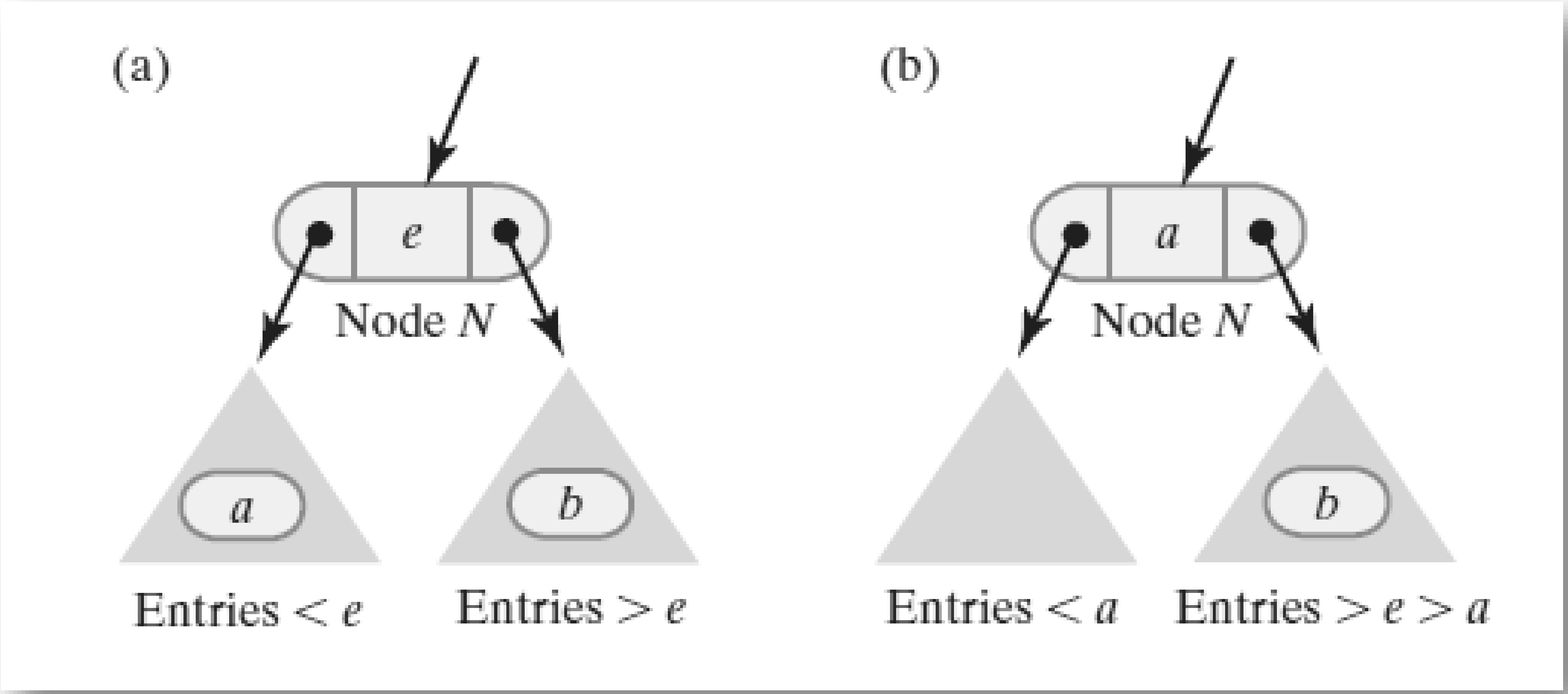


3. Removing an **Entry** whose **Node** has **Two Children**



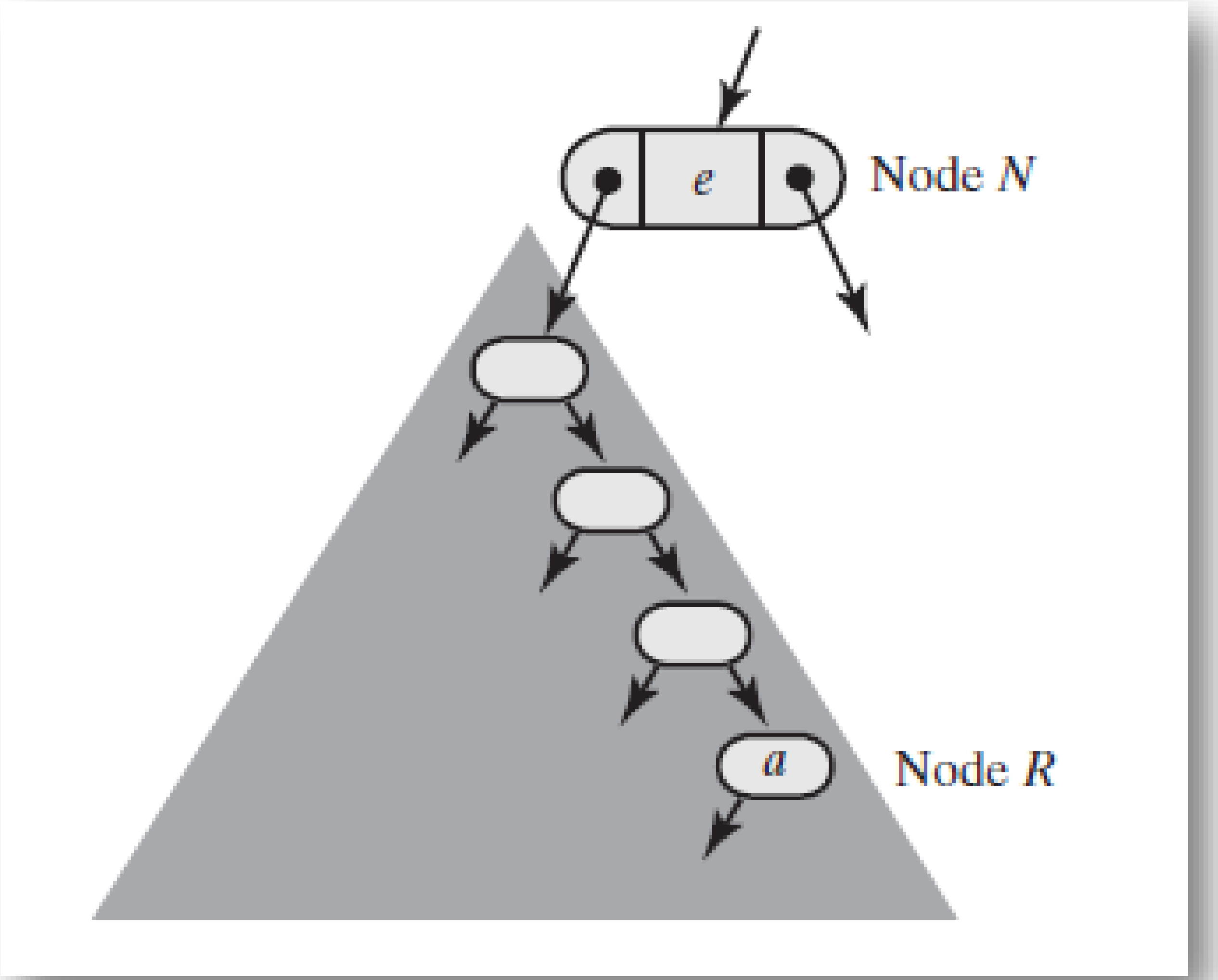
3. Removing an **Entry** whose **Node** has **Two Children** (cont.)

… **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

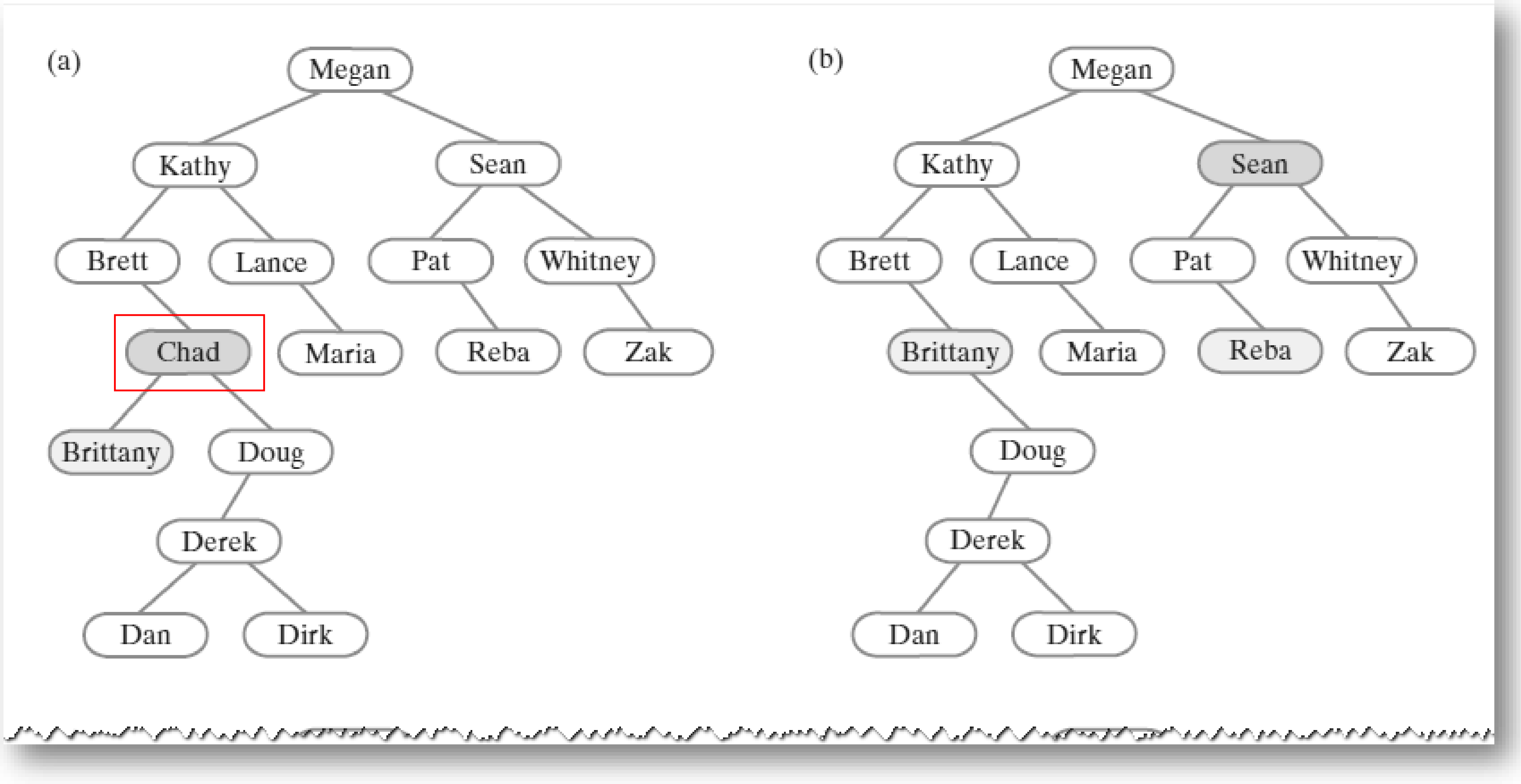
Delete node R

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

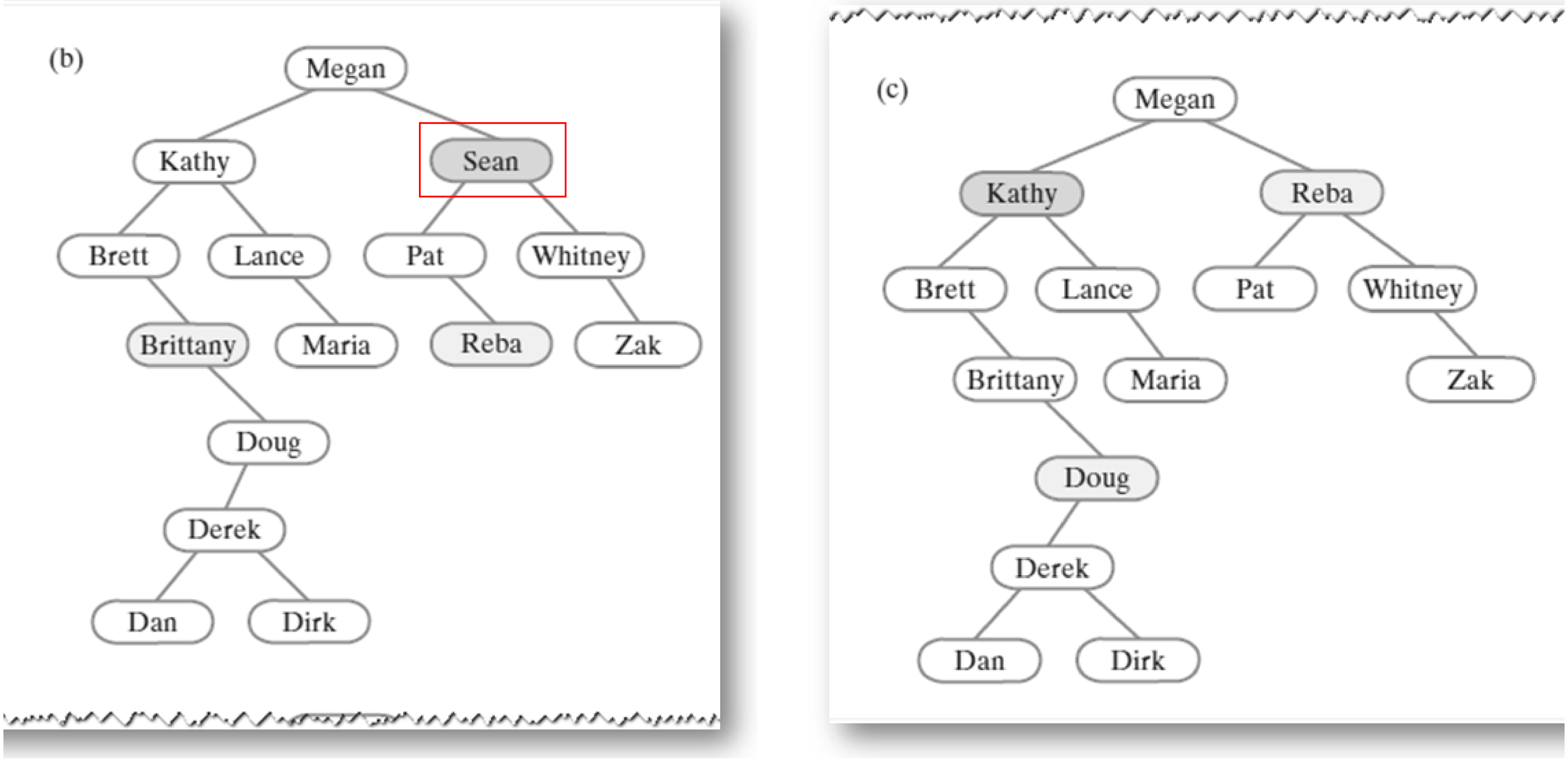
Find the leftmost node L in N's right subtree

Replace the entry in node N with the entry that is in node L

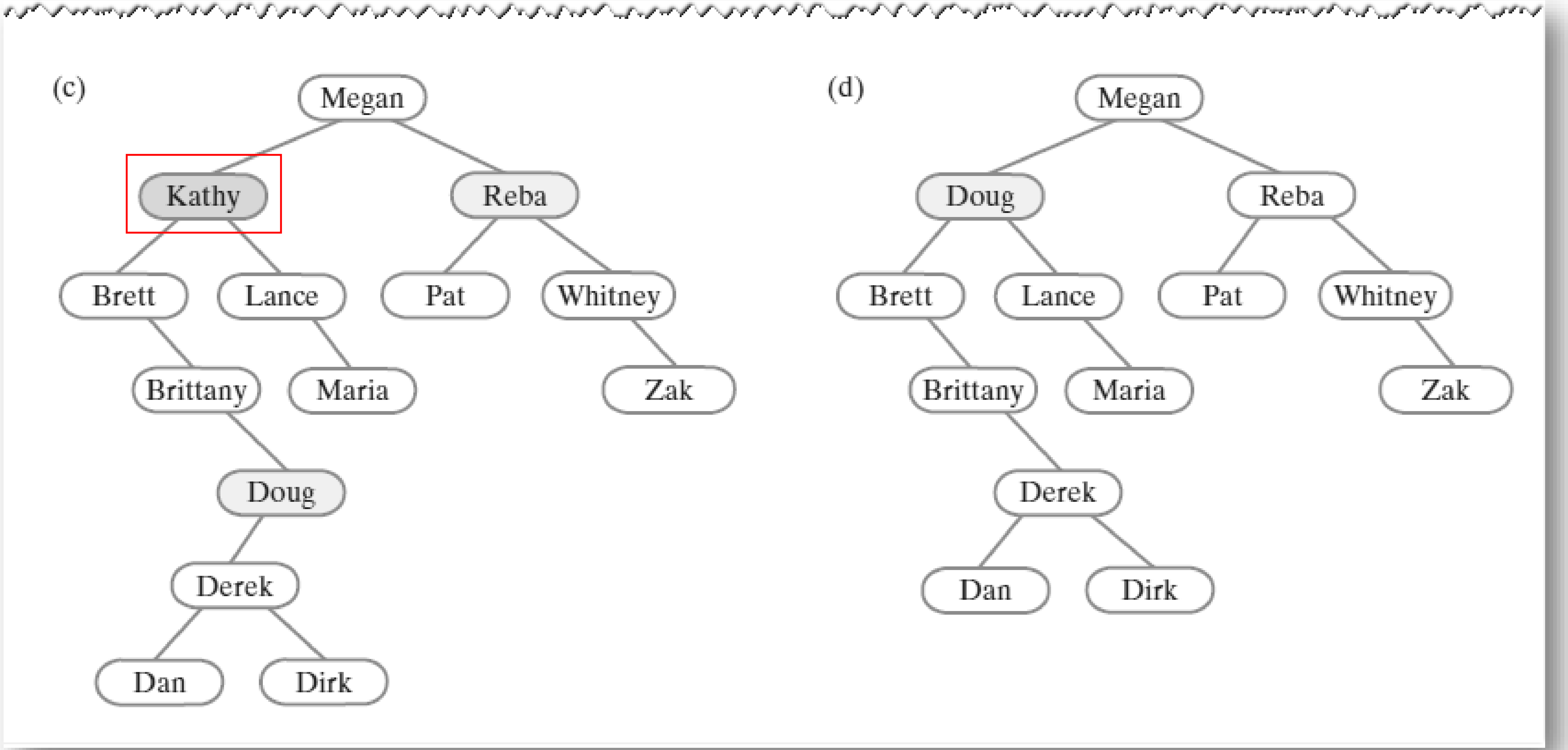
Delete node L



Example: (a) A binary search tree; (b) after removing *Chad*; (c) after removing *Sean*; (d) after removing *Kathy*.

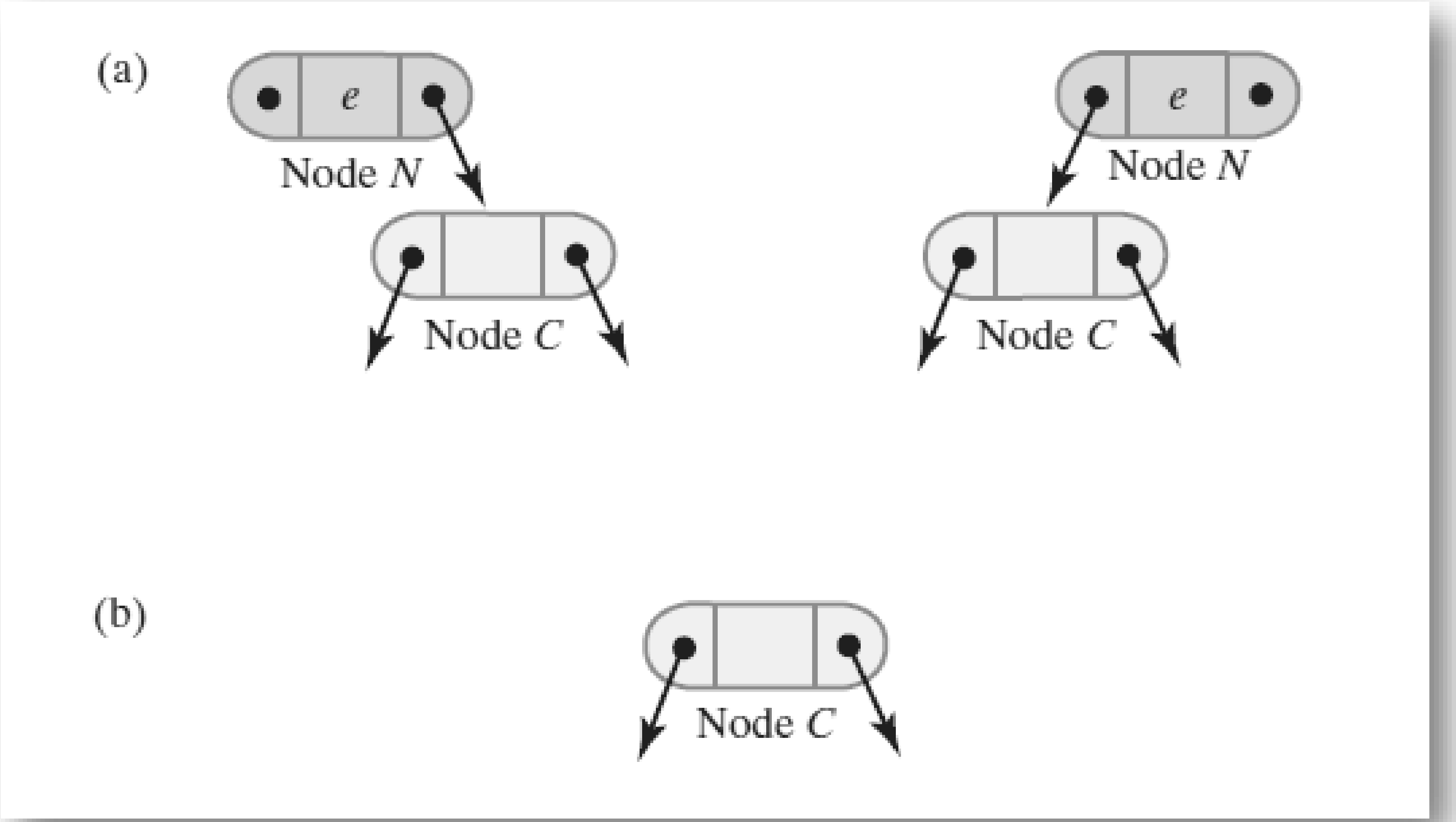


Example: (a) A binary search tree; (b) after removing *Chad*; (c) after removing *Sean*; (d) after removing *Kathy*.



Example: (a) A binary search tree; (b) after removing *Chad*; (c) after removing *Sean*; (d) after removing *Kathy*.

Removing an **Entry** in the **Root**



(a) Two possible configurations of a root that has one child; (b) after removing the root.

The method **remove**

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

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

```
private class ReturnObject
{
 private T item;

 private ReturnObject(T entry)
 {
     item = entry;
 } // end constructor

 public T get()
 {
     return item;
 } // end get

 public void set(T entry)
 {
     item = entry;
 } // end set
} // end ReturnObject
```

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

 return rootNode;
} // end removeEntry
```

The method **removeFromRoot**

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> rootNode)
{
 if (rootNode.hasLeftChild() && rootNode.hasRightChild())
 {
  BinaryNode<T> leftSubtreeRoot = rootNode.getLeftChild();
  BinaryNode<T> largestNode = findLargest(leftSubtreeRoot);

  rootNode.setData(largestNode.getData());

  rootNode.setLeftChild(removeLargest(leftSubtreeRoot));
 }
 else if (rootNode.hasRightChild())
  rootNode = rootNode.getRightChild();
 else
  rootNode = rootNode.getLeftChild();

 return rootNode;
} // end removeEntry
```

The methods **findLargest** and **removeLargest**

```
private BinaryNode<T> findLargest(BinaryNode<T> rootNode)
{
 if (rootNode.hasRightChild())
  rootNode = findLargest(rootNode.getRightChild());

 return rootNode;
} // end findLargest
```

```
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 Efficiency of Operations

- Each of the operations **add**, **remove**, and **getEntry** requires a search that begins at the root of the tree.
  - For a tree of height h, these operations are $O(h)$.
- The tallest tree has height n if it contains n nodes. In fact, this tree looks like a linked chain, and searching it is like search a linked chain. It is an $O(n)$ operation. Thus, **add**, **remove**, and **getEntry** operations for this tree are also $O(n)$.
- The balance is important.
  - Height balanced …
- The order in which nodes are added affects the shape of the tree.
  - If you add entries into an initially empty binary search tree, do not add them in sorted order.
  - For example, assume you are given data as follows: *Brett*, *Brittany*, *Doug*, *Jared*, *Jim*, *Megan*, *Whitney*.

References

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