Data Structures

CS 284

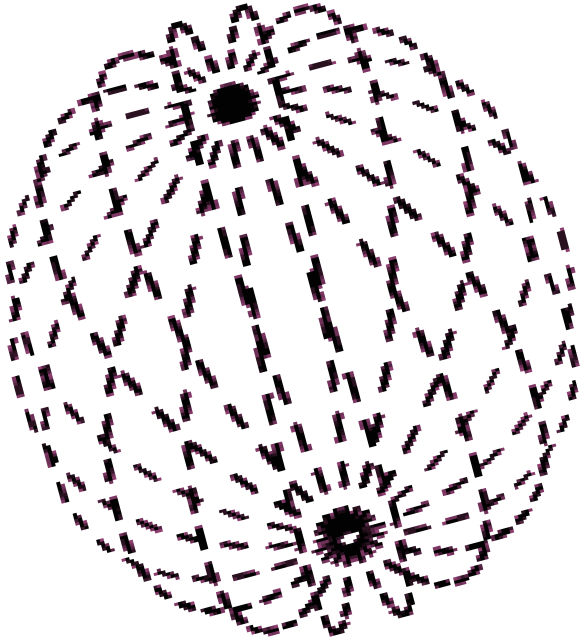
Spring 2022

THE INNOVATION UNIVERSITY
INSTITUTE OF TECHNOLOGY
1870
STEVENS

by:
Abdelnasser Ouda
Email: abdelnasser @ yahoo.com

Unit 4: Trees

- 4.1: Tree Concepts
- 4.2: Java Interfaces & Implementation for Trees
- 4.3: Binary Search Trees
- 4.4: Heap Trees
- 4.5: AVL Trees
- 4.6: 2-3 & 2-4 Trees
- 4.7: Red-Black Trees



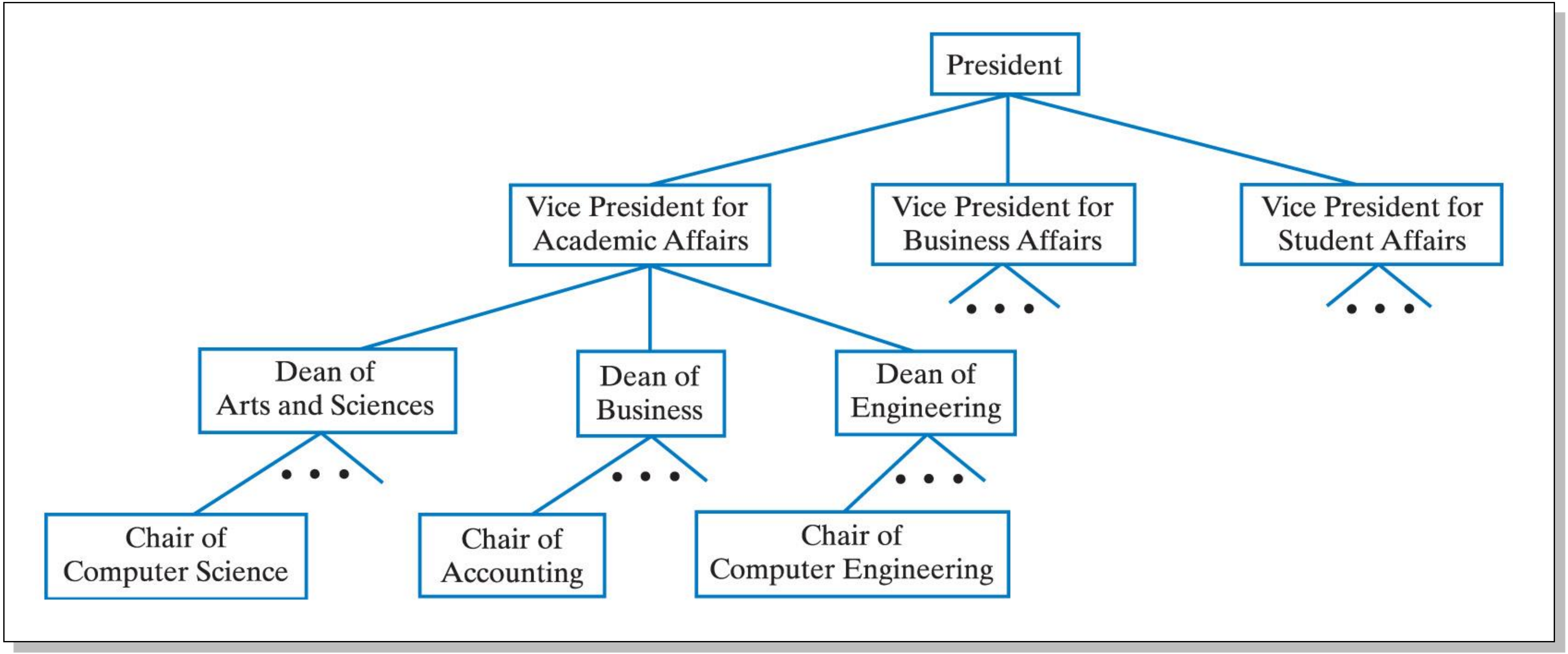
4.1: Tree Concepts

What is a Tree

- A list, stack, or queue is a linear structure that consists of a sequence of elements.
- In Computer Science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments

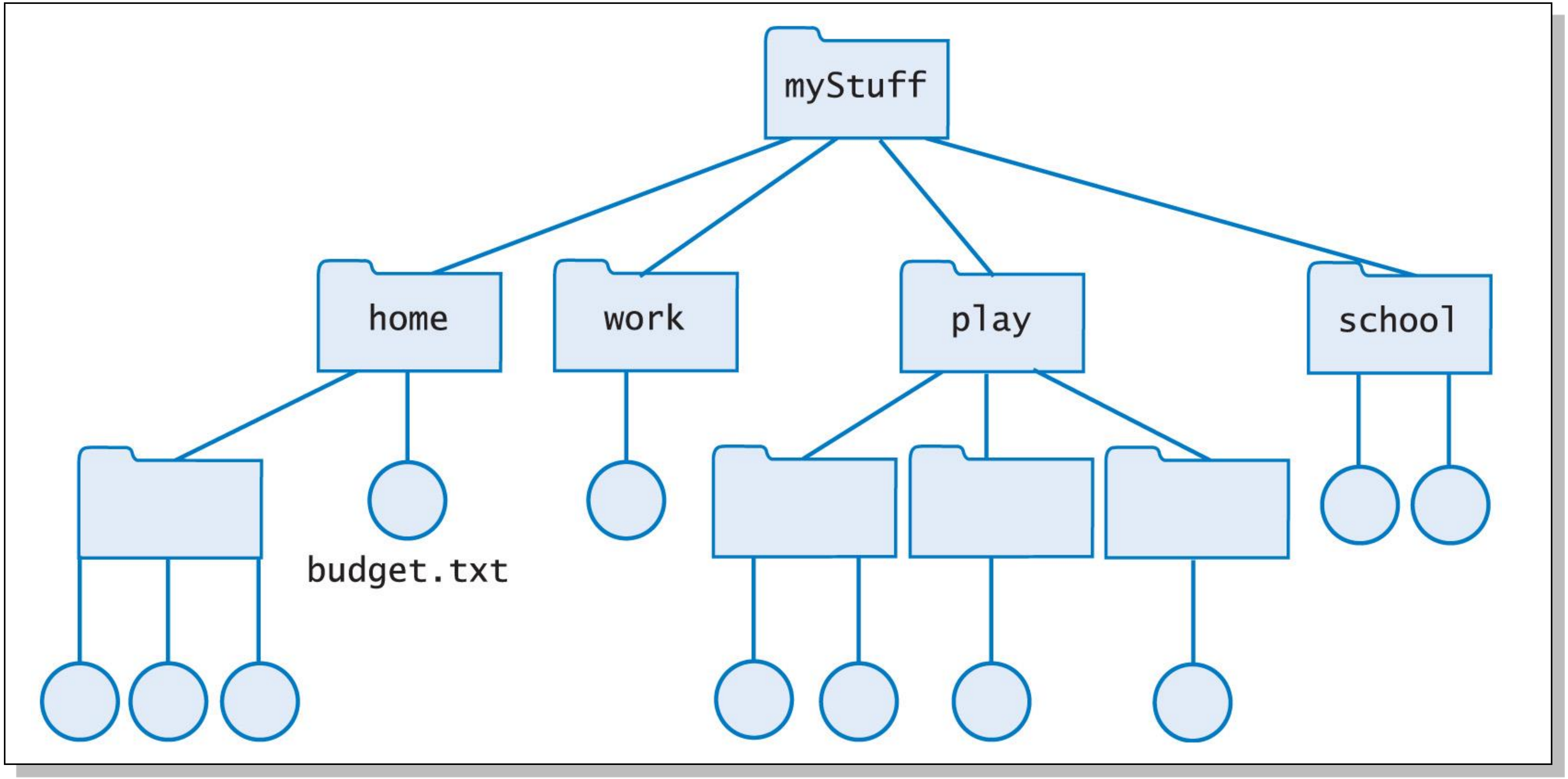
Hierarchical Organization

- Example: A university's organization



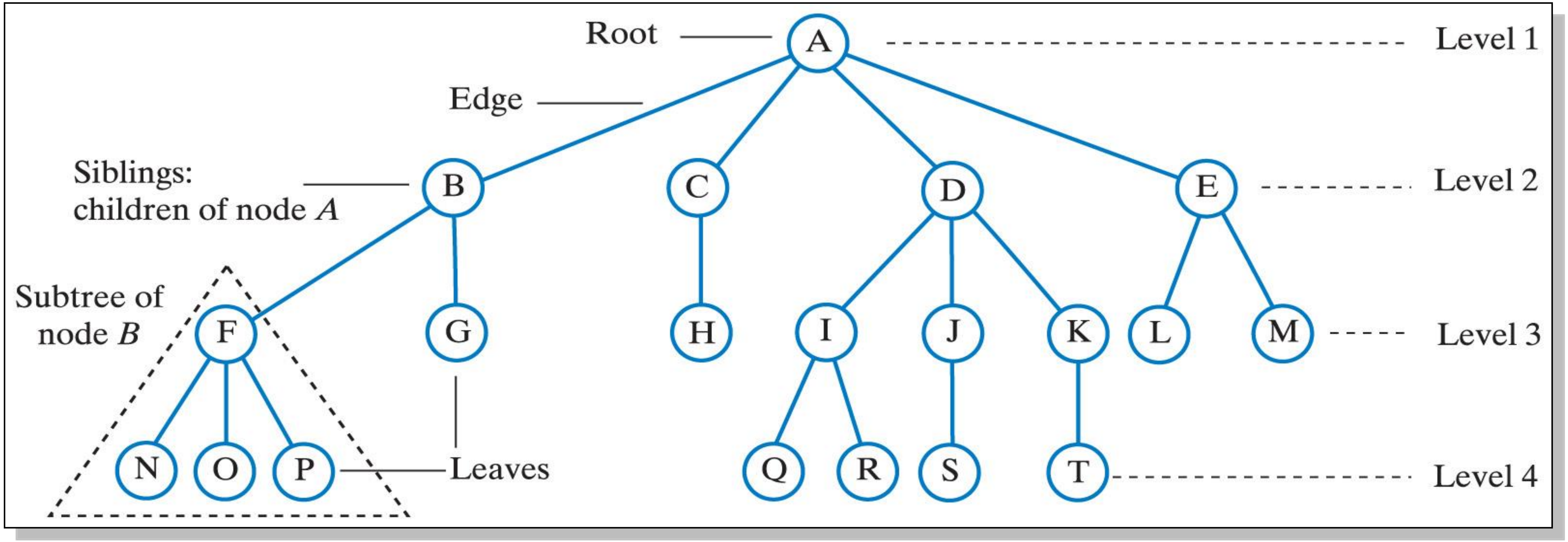
Hierarchical File Systems

- Example: File Directories



Tree Terminology

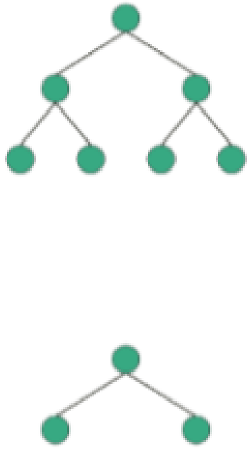
- A tree is a set of **nodes**, connected by **edges** that indicate relationships among nodes
- Nodes arranged in hierarchy levels
  - Top level is a single node called the **root**, node with no parent
  - The **height h** of a nonempty tree is the length of the path from the root node to its furthest leaf
  - The height of a tree that contains a single node is **0**
  - A node is reached from the root by a **path**, **Path-Length** is the number of edges that compose it



Binary Trees - Each node has at most two children

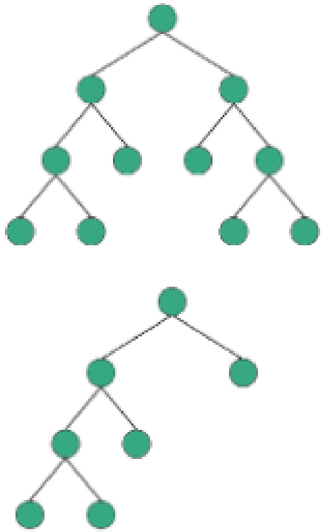
Perfect B-Tree

A Binary Tree in which all internal nodes have 2-children and all the leaf nodes are at the same depth or same level



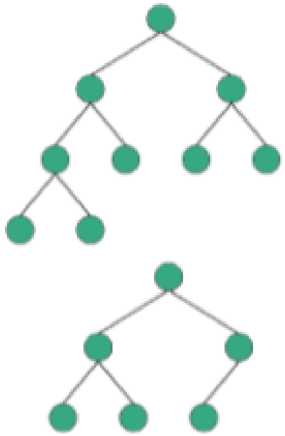
Full B-Tree

A Binary Tree in which every node has 0 or 2 children



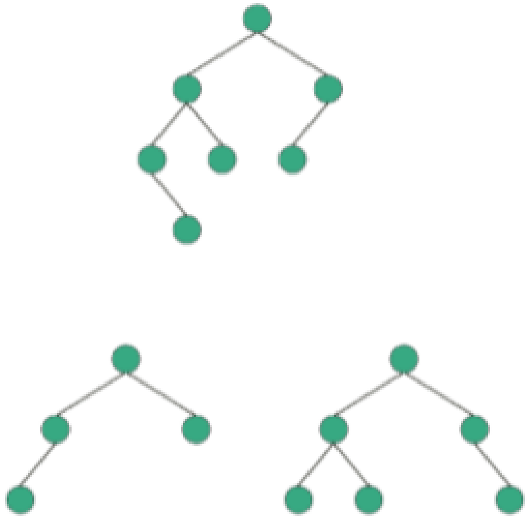
Complete B-Tree

All levels completely filled with nodes except the last level and in the last level, all the nodes are as left side as possible



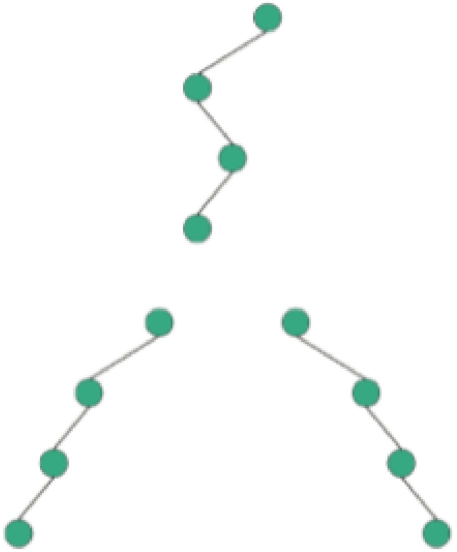
Balanced B-Tree

A Binary Tree in which height of the left and the right sub-trees of every node may differ by at most 1



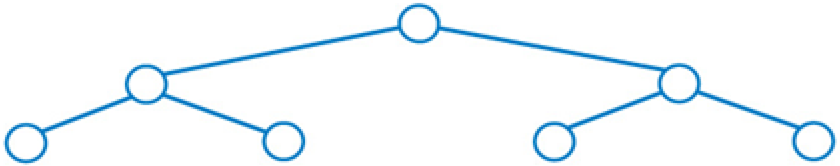
Degenerate(or Pathological) B-Tree

A Binary Tree where every parent node has only one child node.



Binary Trees

- The number of nodes in a prefect binary tree as a function of the tree's height.

Full Tree	**Height** **h**	**Number of Nodes** $2^{h+1} - 1$
	0	$1 = 2^1 - 1$
	1	$3 = 2^2 - 1$
	2	$7 = 2^3 - 1$

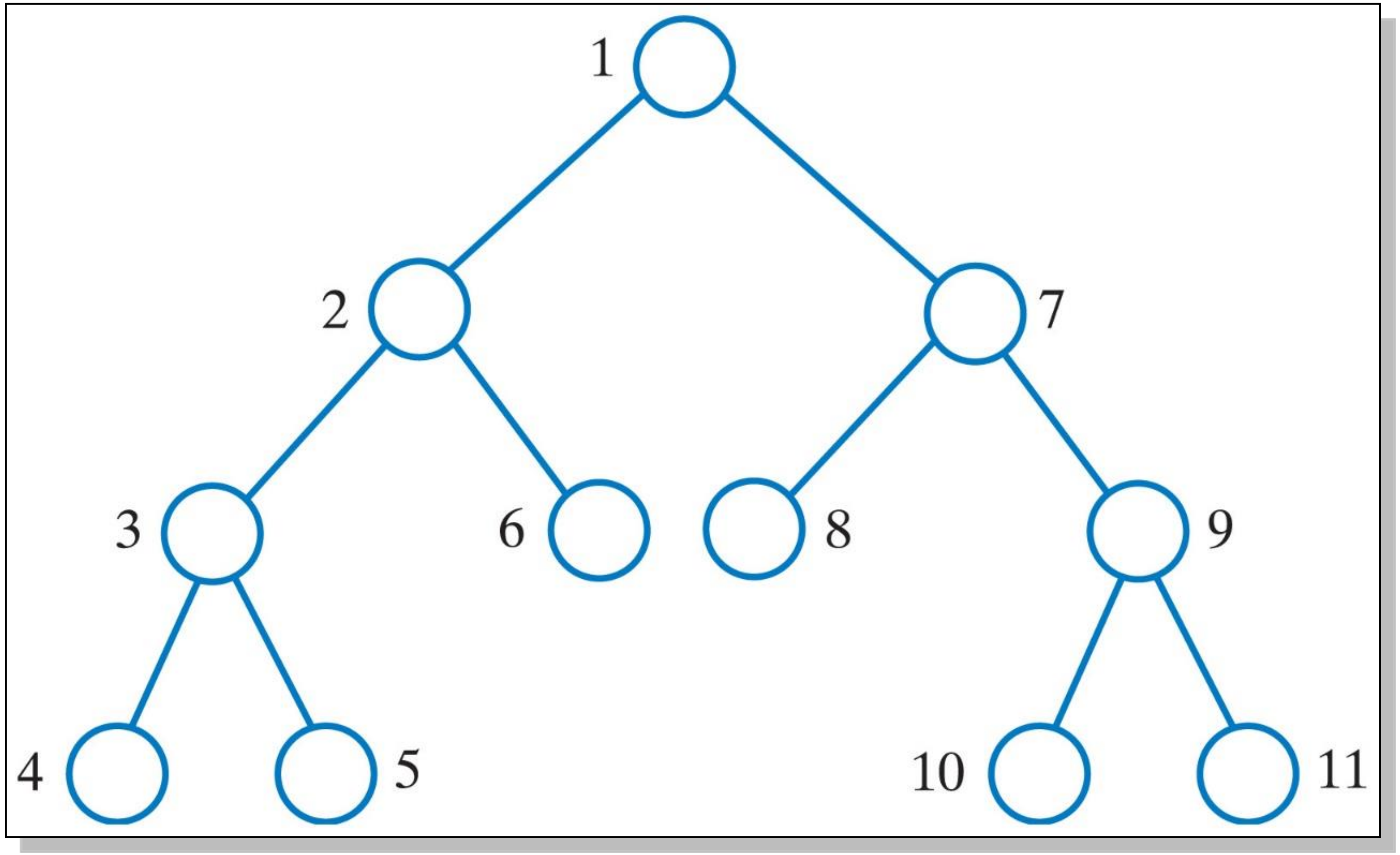
- The height of a binary tree with **n** nodes that is either complete or full is $\log_2(n + 1)$

Traversals of a Tree

- Visiting a node
  - Processing the data within a node
  - This is the action performed on each node during traversal of a tree
- A traversal can pass through a node without visiting it at that moment
  - inorder, preorder, postorder, depth-first, and breadth-first traversals.

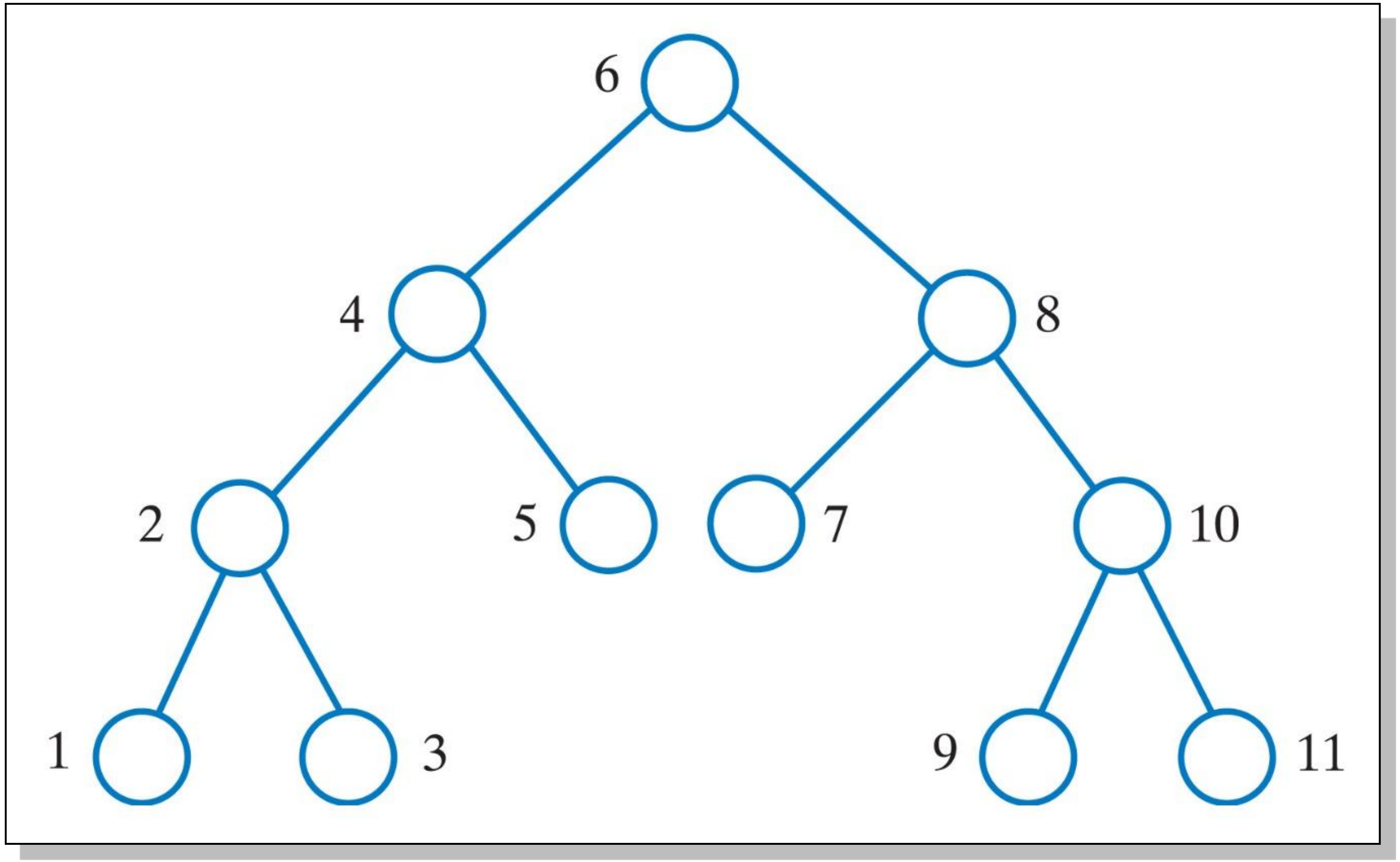
Traversals of a Tree

- Preorder traversal: the current node is visited first, then recursively the left subtree of the current node, and finally the right subtree of the current node recursively



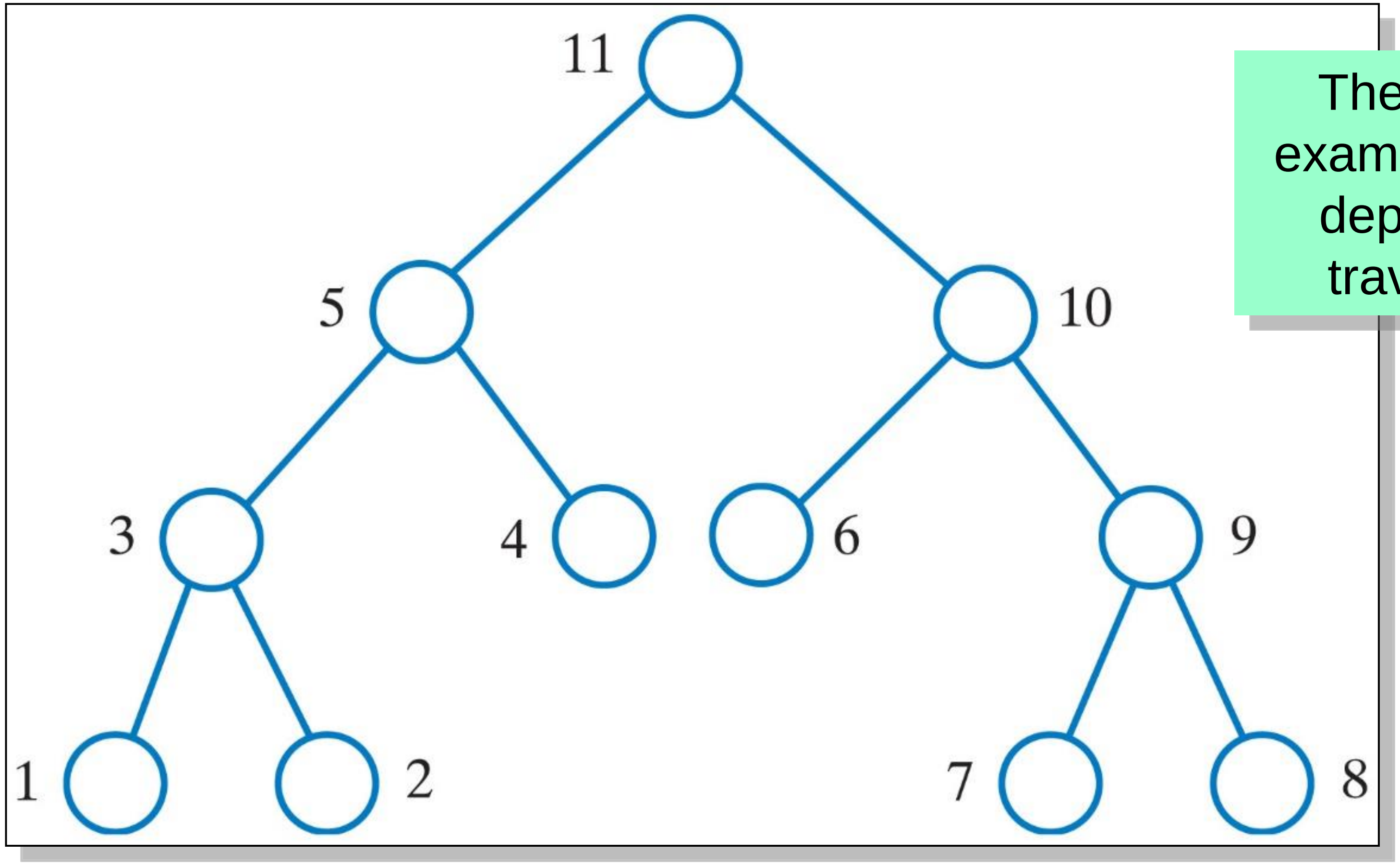
Traversals of a Tree

- Inorder traversal: visit the left subtree of the current node first recursively, then the current node itself, and finally the right subtree of the current node recursively



Traversals of a Tree

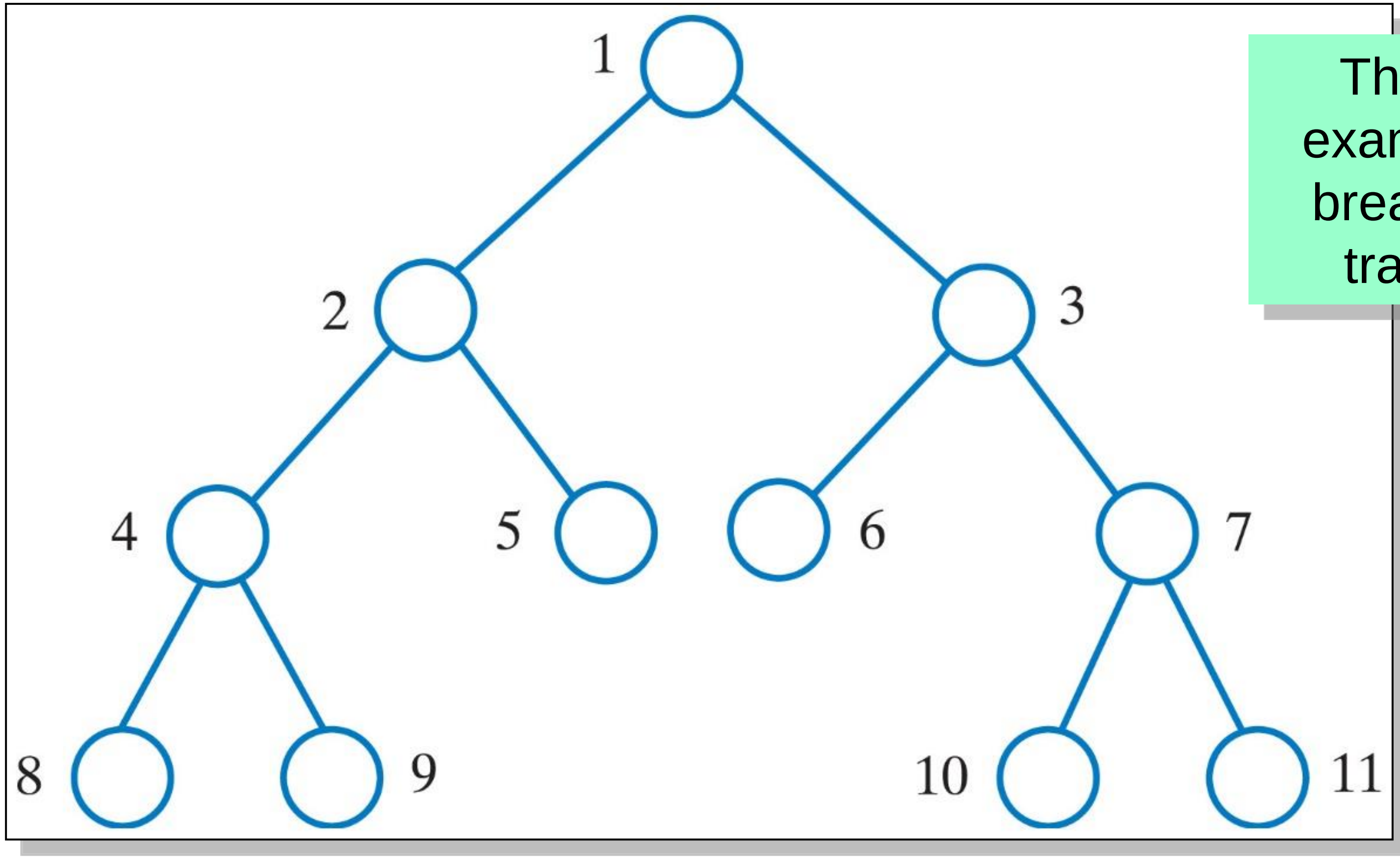
- Postorder traversal: visit the left subtree of the current node first, then the right subtree of the current node, and finally the current node itself



These are examples of a depth-first traversal.

Traversals of a Tree

- Level-order traversal: begin at the root, visit nodes one level at a time

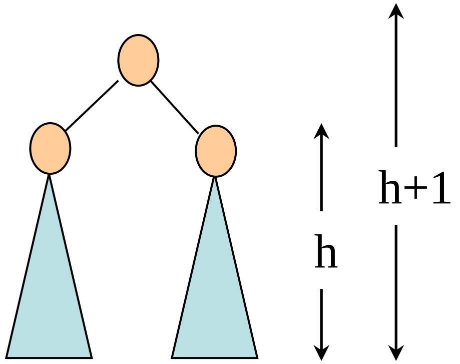


This is an example of a breadth-first traversal.

https://liveexample.pearsoncmg.com/dsanimation/BSTeBook.html

Trees Properties - Practice Questions

- A complete binary tree is defined inductively as follows.
  - A complete binary tree of height **0** consists of **1** node which is the root.
  - A complete binary tree of height **h+1** consists of two complete binary trees of height **h** whose roots are connected to a new root.
- Let **T** be a complete binary tree of height **h**. Prove that the size of the tree (number of nodes in **T**) is $2^{h+1} - 1$.



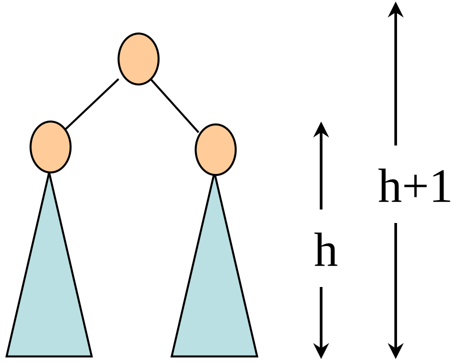
Trees Properties - Practice Questions

- **Proof by Induction**: let **T(h)** be the number of nodes of **T** at height **h**.
- **Base case**: **h=0**, by definition we have only one node which is the root.

  $T(0) = 1$ and $2^{0+1} – 1 = 2\text{-}1 = 1$

  So, **$T(h) = 2^{h+1} – 1$ holds for $h=0$**
- **Induction hypothesis**: Assume for any complete binary tree of height **h** the size of this tree is **$T(h) = 2^{h+1} – 1, h>0$**
- **Induction step**: We know that a complete binary tree of height **h+1** consists of two complete binary tree each of height **h** whose root are connected to a new root. i.e.,

$$
\begin{aligned}
T(h+1) &= 1 + 2\,T(h) \\
&= 1 + 2(2^{h+1} – 1) \quad \textbf{by induction hypothesis} \\
&= 1 + 2^{h+2} – 2 \\
&= 2^{h+2} – 1 = 2^{(h+1)+1} – 1
\end{aligned}
$$



So, **$T(h) = 2^{h+1} – 1$ holds for $h+1$**

Trees Properties - Practice Questions

- Let **T** be a complete binary tree of height **h**. The height of a node in **T** is the node's distance to a leaf (e.g., the root has height **h**, whereas a leaf has height **0**).

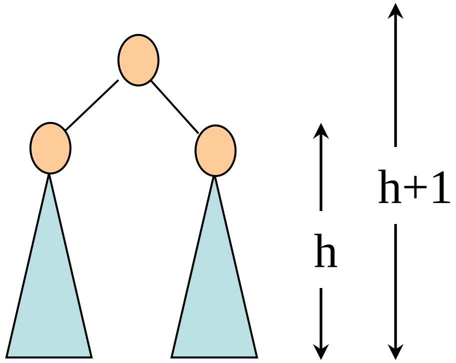
- Prove that the sum of the heights of all the nodes in **T** is $\mathbf{2^{h+1} - h - 2}$.

Trees Properties - Practice Questions

- **Proof by Induction**: let **S(h)** be the sum of the height of all nodes in **T**
- **Base case**: **h=0**, by definition we have only one node which is the root.

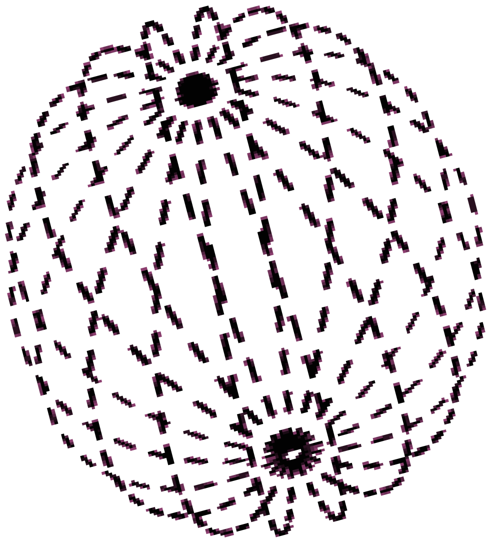
  $S(0) = 0$ and $2^{0+1} – 0 - 2 = 0$
  So, **$S(h) = 2^{h+1} – h - 2$ holds for $h=0$**

- **Induction hypothesis**: Assume for any complete binary tree **T** of height **h**,
  $S(h) = 2^{h+1} – h - 2, h>0$
- **Induction step**: a complete binary tree of height **h+1** consists of two complete binary tree each of height **h** whose root are connected to a new root. i.e.,



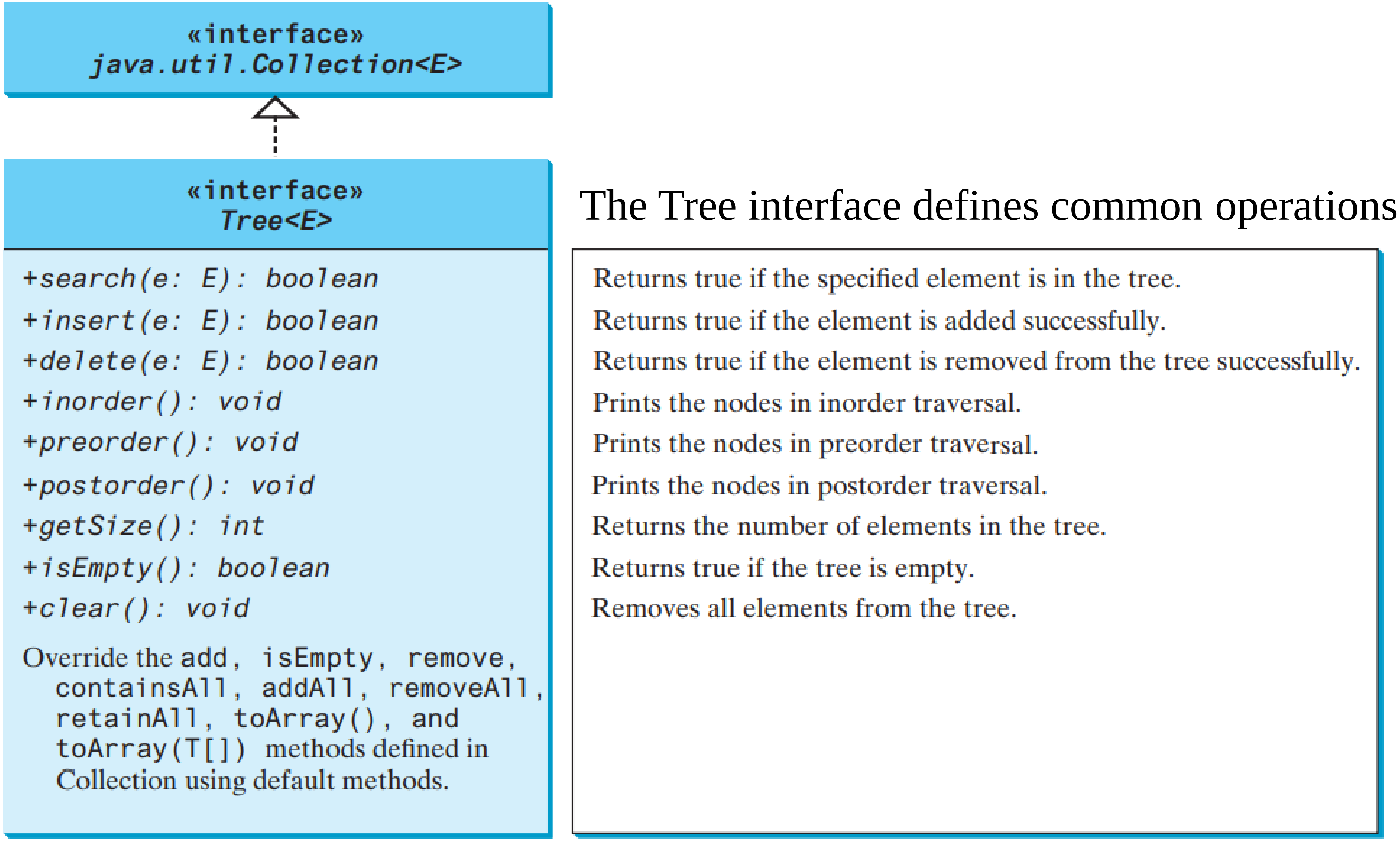
$S(h+1)$ = the height of the root + $2\,S(h)$
$= (h+1) + 2(2^{h+1} – h -2)$ **by induction hypothesis**
$= 2^{h+2} – h – 3 = 2^{(h+1)+1} – (h+1) - 2$

Then, **$S(h) = 2^{h+1} – h - 2$ holds for $h+1$**



4.2: Java Interfaces & Implementation for Trees

Java Interfaces for Trees



«interface»
java.util.Collection<E>

«interface»
Tree<E>

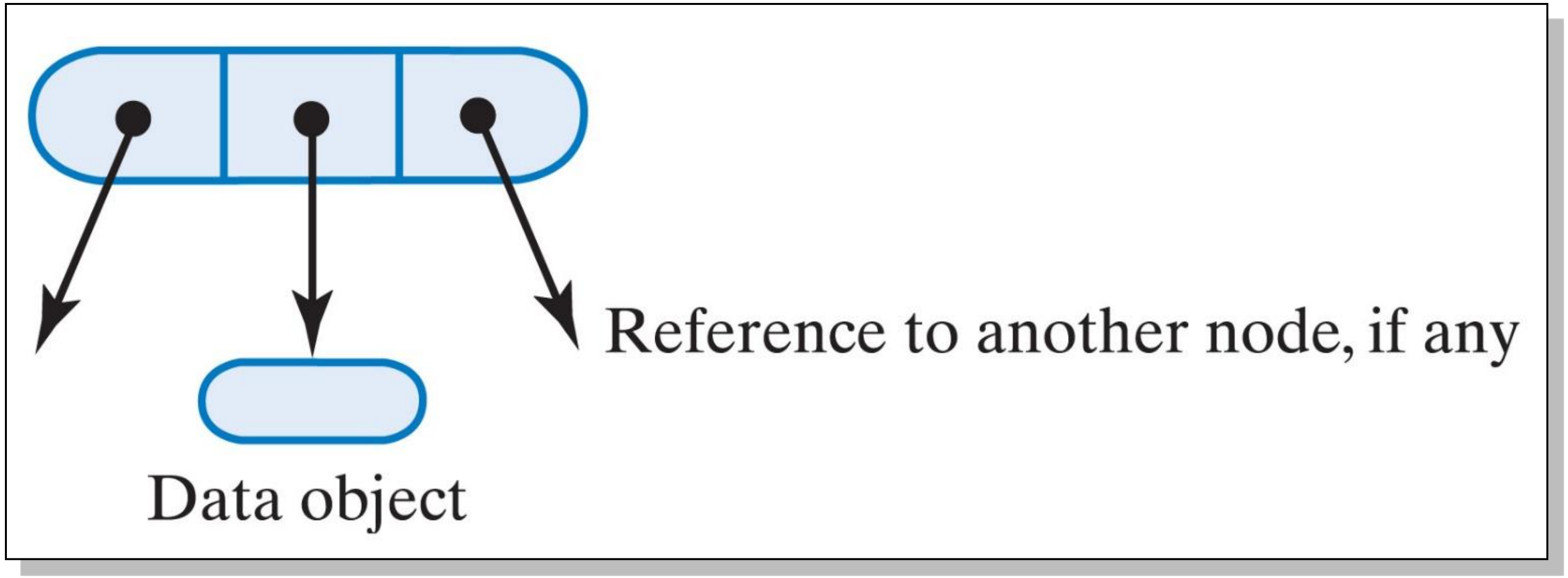
The Tree interface defines common operations for trees.

`+search(e: E): boolean`	Returns true if the specified element is in the tree.
`+insert(e: E): boolean`	Returns true if the element is added successfully.
`+delete(e: E): boolean`	Returns true if the element is removed from the tree successfully.
`+inorder(): void`	Prints the nodes in inorder traversal.
`+preorder(): void`	Prints the nodes in preorder traversal.
`+postorder(): void`	Prints the nodes in postorder traversal.
`+getSize(): int`	Returns the number of elements in the tree.
`+isEmpty(): boolean`	Returns true if the tree is empty.
`+clear(): void`	Removes all elements from the tree.

Override the `add`, `isEmpty`, `remove`, `containsAll`, `addAll`, `removeAll`, `retainAll`, `toArray()`, and `toArray(T[])` methods defined in Collection using default methods.

Tree

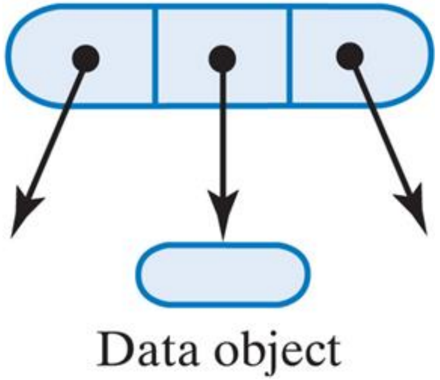
Nodes in a Binary Tree



- Let us examine the **`BinaryNodeInterface`**
- And then the class **`BinaryNode`** that implement it

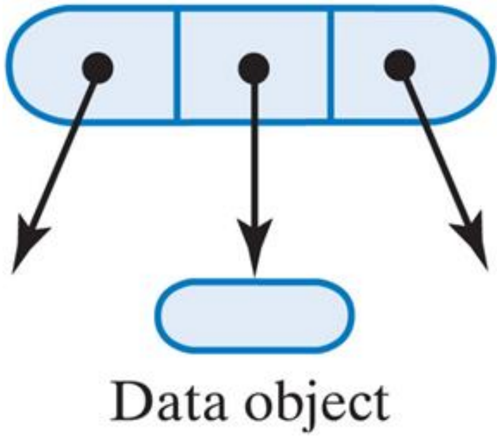
```
package TreePackage;
interface BinaryNodeInterface < T >
{
    /** Task: Retrieves the data portion of the node.
    * @return the object in the data portion of the node */
    public  T getData ();

    /** Task: Sets the data portion of the node.
    * @param newData the data object */
    public void setData (T newData);

    /** Task: Retrieves the left child of the node.
    * @return the node that is this nodes left child */
    public BinaryNodeInterface < T > getLeftChild ();

    /** Task: Retrieves the right child of the node.
    * @return the node that is this nodes right child */
    public BinaryNodeInterface < T > getRightChild ();

    /** Task: Sets the nodes left child to a given node.
    * @param leftChild a node that will be the left child */
    public void setLeftChild (BinaryNodeInterface < T > leftChild);

    /** Task: Sets the nodes right child to a given node.
    * @param rightChild a node that will be the right child */
    public void setRightChild (BinaryNodeInterface < T > rightChild);
```

An interface for the nodes in a binary tree

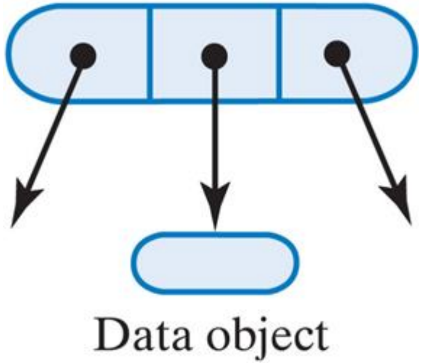


An interface for the nodes in a binary tree

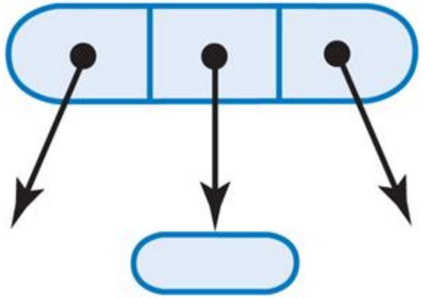
```
    /** Task: Detects whether the node has a left child.
    * @return true if the node has a left child */
    public boolean hasLeftChild ();

    /** Task: Detects whether the node has a right child.
    * @return true if the node has a right child */
    public boolean hasRightChild ();

    /** Task: Detects whether the node is a leaf.
    * @return true if the node is a leaf */
    public boolean isLeaf ();

    /** Task: Counts the nodes in the subtree rooted at this node.
    *@returnthenumberof nodes in the subtree rooted at this node */
    public int getNumberOfNodes ();

    /** Task: Computes the height of the subtree rooted at this node.
    * @return the height of the subtree rooted at this node */
    public int getHeight ();

    /** Task: Copies the subtree rooted at this node.
    * @return the root of a copy of the subtree rooted at this node */
    public BinaryNodeInterface < T > copy ();

} // end BinaryNodeInterface
```



```
package TreePackage;
class  BinaryNode < T > implements BinaryNodeInterface < T > {
    private T data;
    private BinaryNode < T > left;
    private BinaryNode < T > right;

    public BinaryNode () {
        this (null);  // call next constructor
    } // end default constructor
    public BinaryNode (T dataPortion) {
        this (dataPortion, null, null); // call next constructor
    } // end constructor
    public BinaryNode (T dataPortion, BinaryNode < T > leftChild, BinaryNode < T > rightChild) {
        data = dataPortion;
        left = leftChild;
        right = rightChild;
    } // end constructor

   public T getData () {
        return data;
    } // end getData
    public void setData (T newData) {
        data = newData;
    } // end setData
    public BinaryNodeInterface < T > getLeftChild () {
        return left;
    } // end getLeftChild
    public void setLeftChild (BinaryNodeInterface < T > leftChild) {
        left = (BinaryNode < T > ) leftChild;
    } // end setLeftChild
    public boolean hasLeftChild () {
        return left != null;
    } // end hasLeftChild
    public boolean isLeaf () {
        return (left == null) && (right == null);
    } // end isLeaf
```

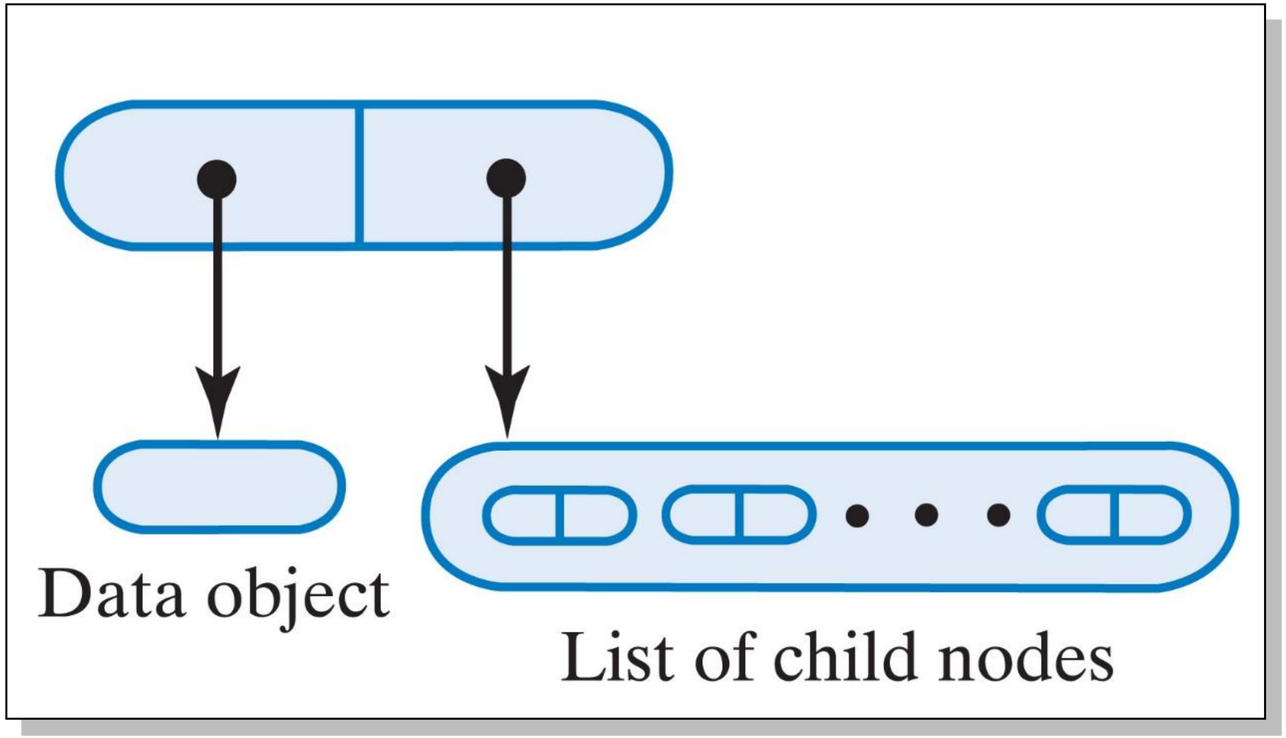


```
    // Implementations of getRightChild, setRightChild, and hasRightChild are analogous to
    // their left-child counterparts.
    public BinaryNodeInterface < T > copy () {
        BinaryNode < T > newRoot = new BinaryNode < T > (data);
        if (left != null)
            newRoot.left = (BinaryNode < T > ) left.copy ();
        if (right != null)
            newRoot.right = (BinaryNode < T > ) right.copy ();
        return newRoot;
    } // end copy

    public int getHeight () {
        return getHeight (this); // call private getHeight
    } // end getHeight

    private int getHeight (BinaryNode < T > node) {
        int height = 0;
        if (node != null)
            height = 1 + Math.max (getHeight (node.left), getHeight (node.right));
        return height;
    } // end getHeight

    public int getNumberOfNodes () {
        int leftNumber = 0;
        int rightNumber = 0;
        if (left != null)
            leftNumber = left.getNumberOfNodes ();
        if (right != null)
            rightNumber = right.getNumberOfNodes ();
        return 1 + leftNumber + rightNumber;
    } // end getNumberOfNodes
} // end BinaryNode
```

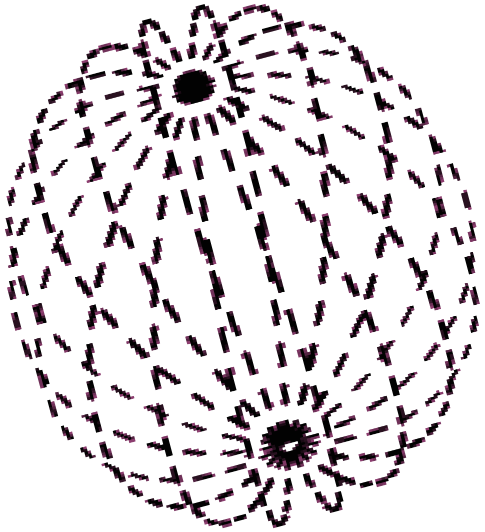


Data object

General Trees



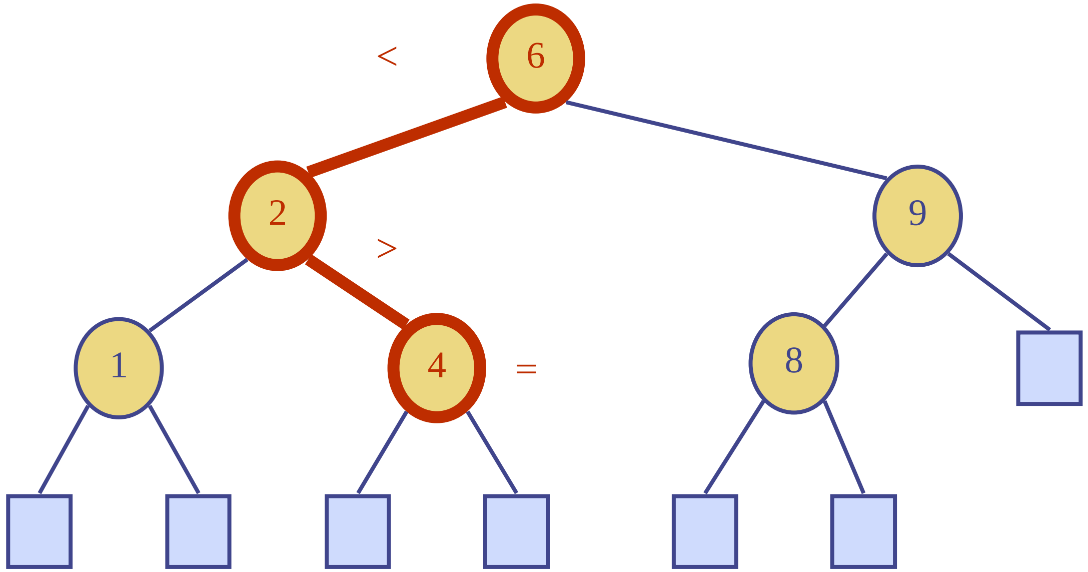
A node for a general tree.



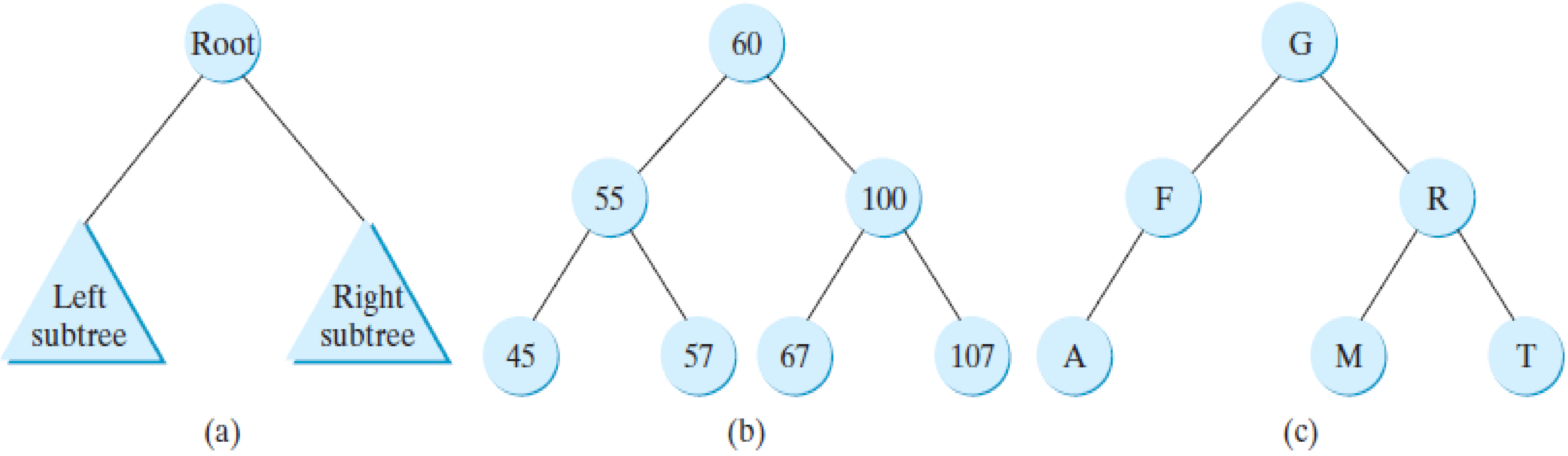
4.3: Binary Search Trees

Binary Search Trees

- A search tree organizes its data so that a search is more efficient
- Binary search tree
  - Nodes contain **Comparable** objects
  - A node's data is greater than the data in the node's left subtree
  - A node's data is less than the data in the node's right subtree
  - Every node in a binary search tree is the root of a binary search tree

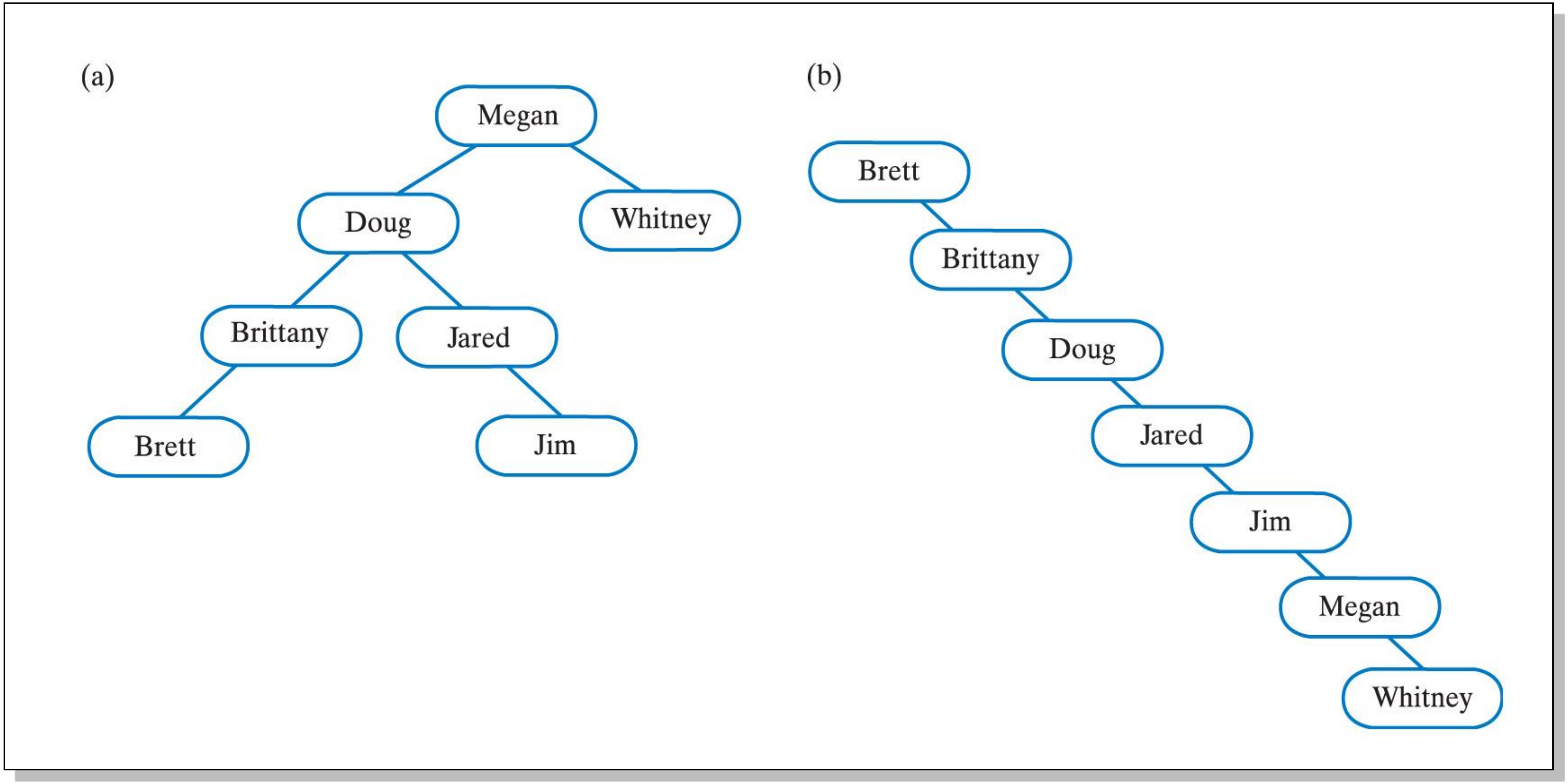


Binary Search Trees

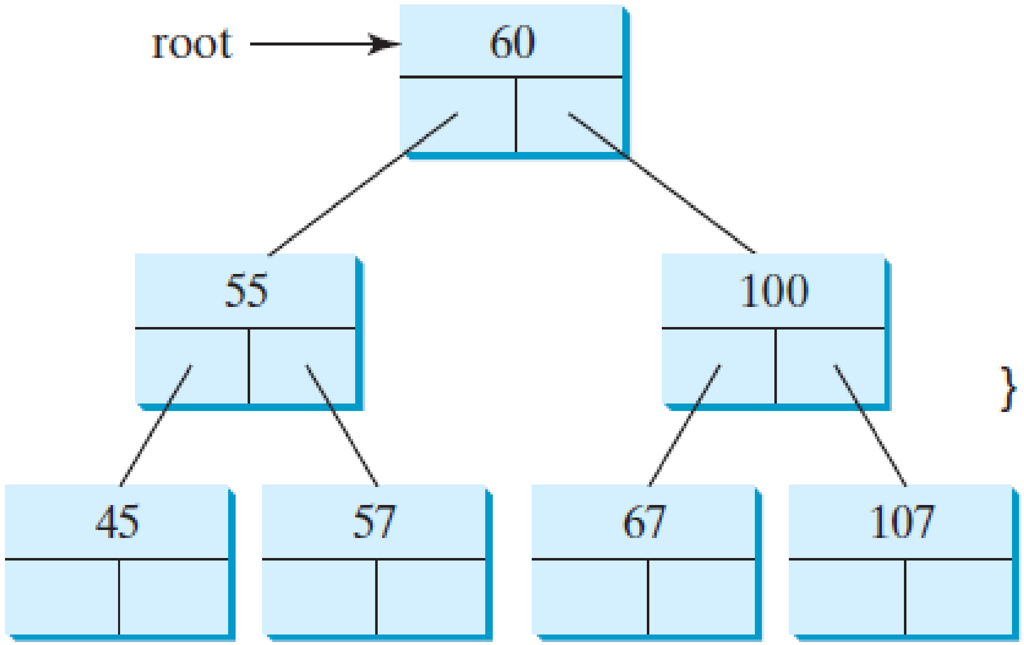


Each node in a binary tree has zero, one, or two subtrees.

Binary Search Trees



Binary Search Trees



A node can be defined as a class, as follows:

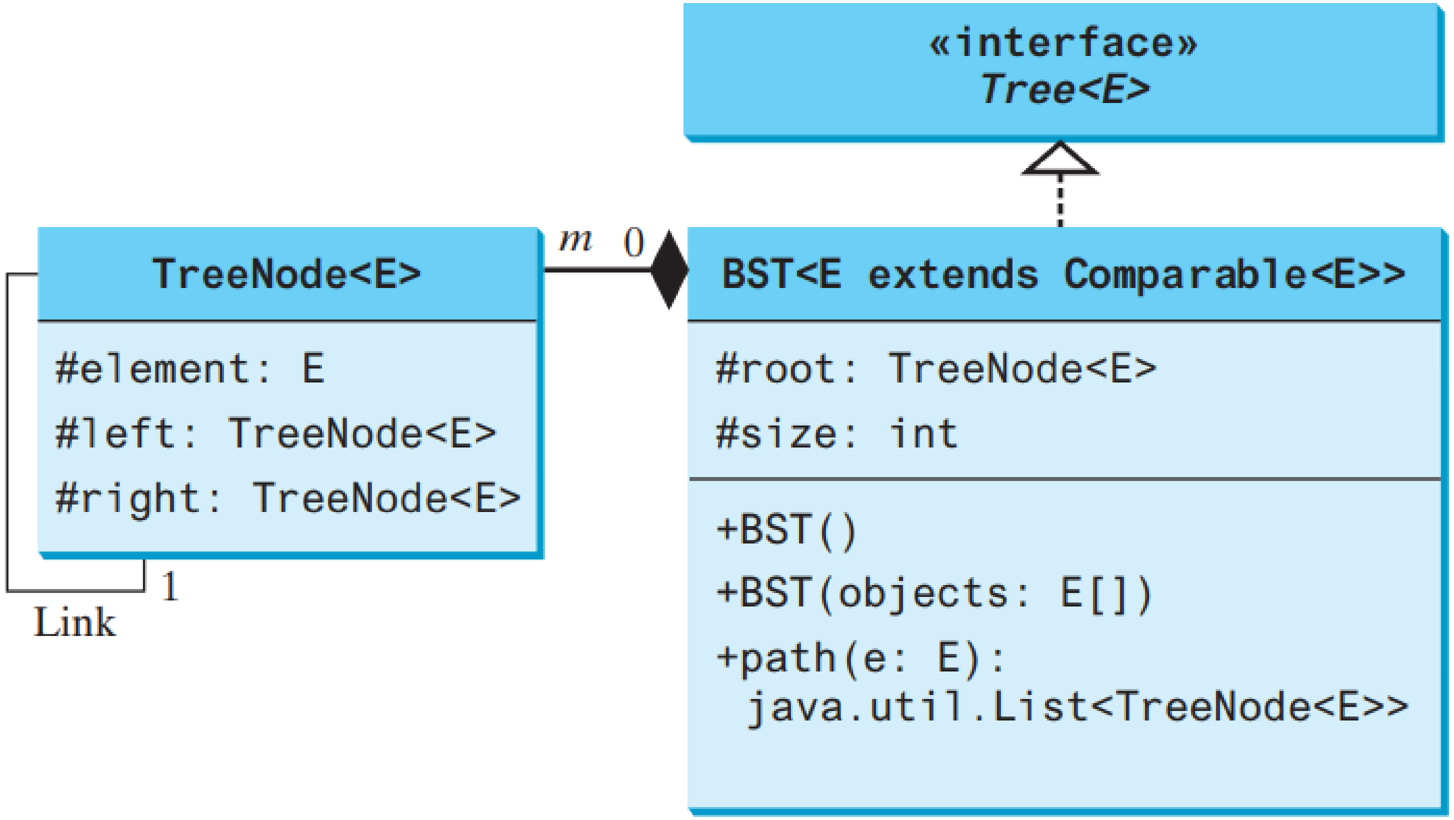
```
class TreeNode<E> {
  protected E element;
  protected TreeNode<E> left;
  protected TreeNode<E> right;

  public TreeNode(E e) {
    element = e;
  }
}
```

A binary tree can be represented using a set of linked nodes

An Interface for the Binary Search Tree

- Includes common operations of a tree
- Also includes basic database operations
    - Search
    - Retrieve
    - Add
    - Remove
    - Traverse

An Implementation for the Binary Search Tree



The BST class defines a concrete BST

The root of the tree.

The number of nodes in the tree.

Creates a default BST.

Creates a BST from an array of elements.

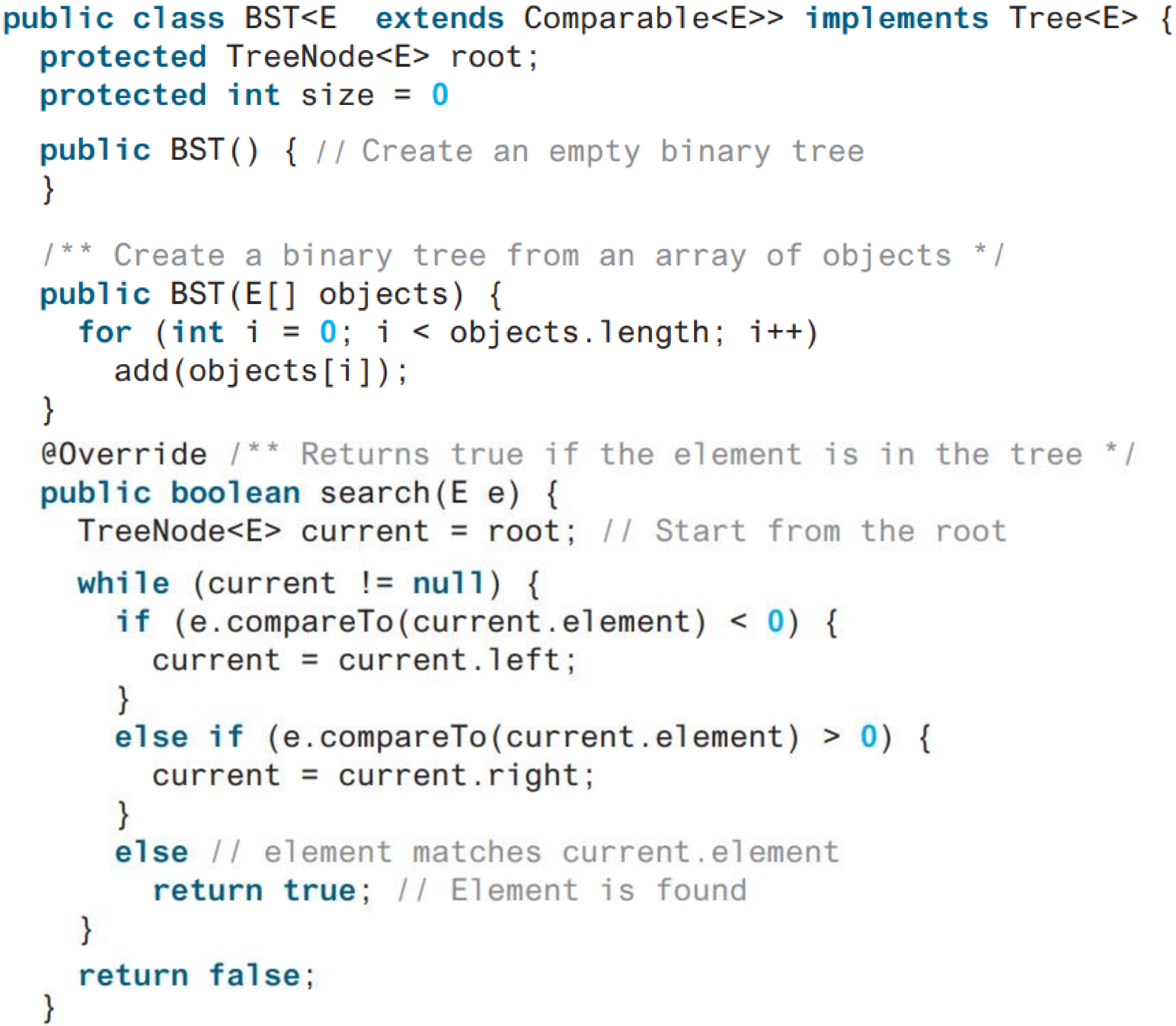
Returns the path of nodes from the root leading to the node for the specified element. The element may not be in the tree.

BST

Recursively Binary 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)
```

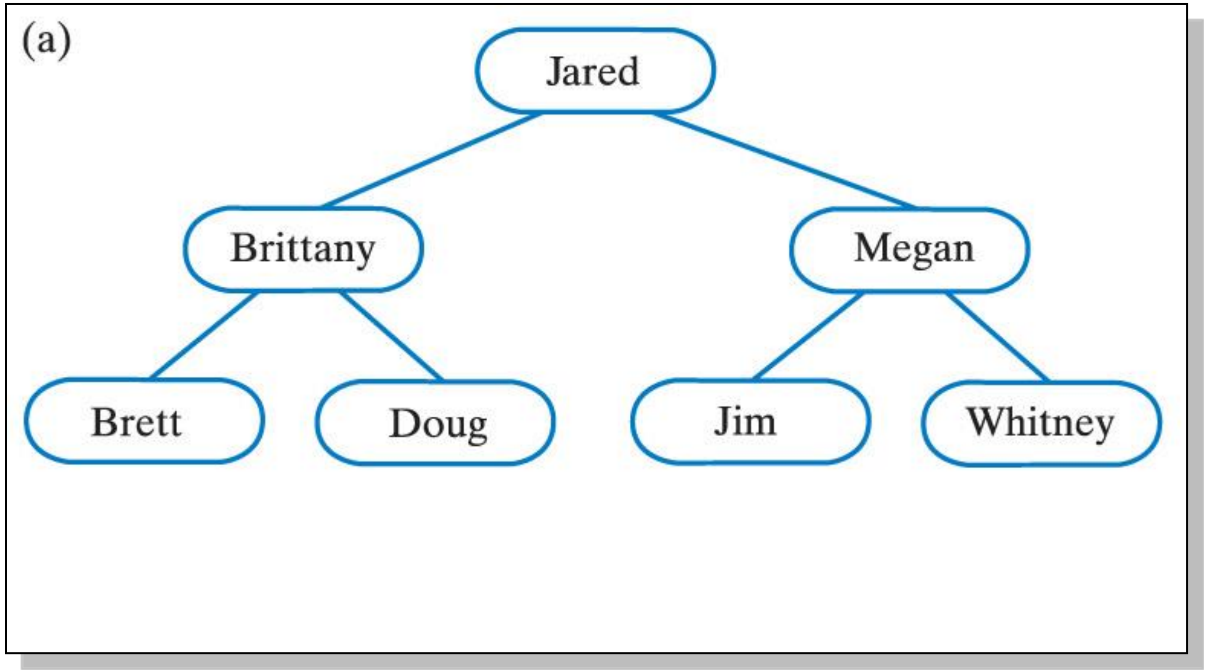
BST Implementation



```
public class BST<E  extends Comparable<E>> implements Tree<E> {
  protected TreeNode<E> root;
  protected int size = 0

  public BST() { // Create an empty binary tree
  }

  /** Create a binary tree from an array of objects */
  public BST(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }
  @Override /** Returns true if the element is in the tree */
  public boolean search(E e) {
    TreeNode<E> current = root; // Start from the root

    while (current != null) {
      if (e.compareTo(current.element) < 0) {
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        current = current.right;
      }
      else // element matches current.element
        return true; // Element is found
    }

    return false;
  }
```

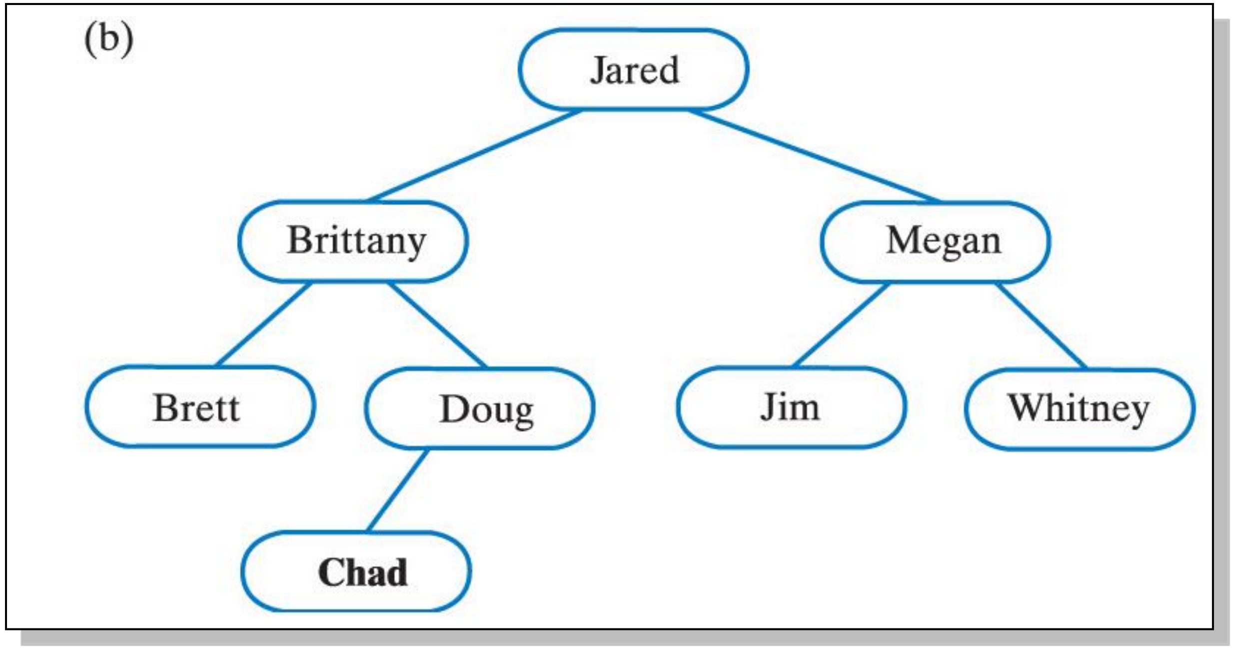
Recursively BS Implementation

Adding an Entry

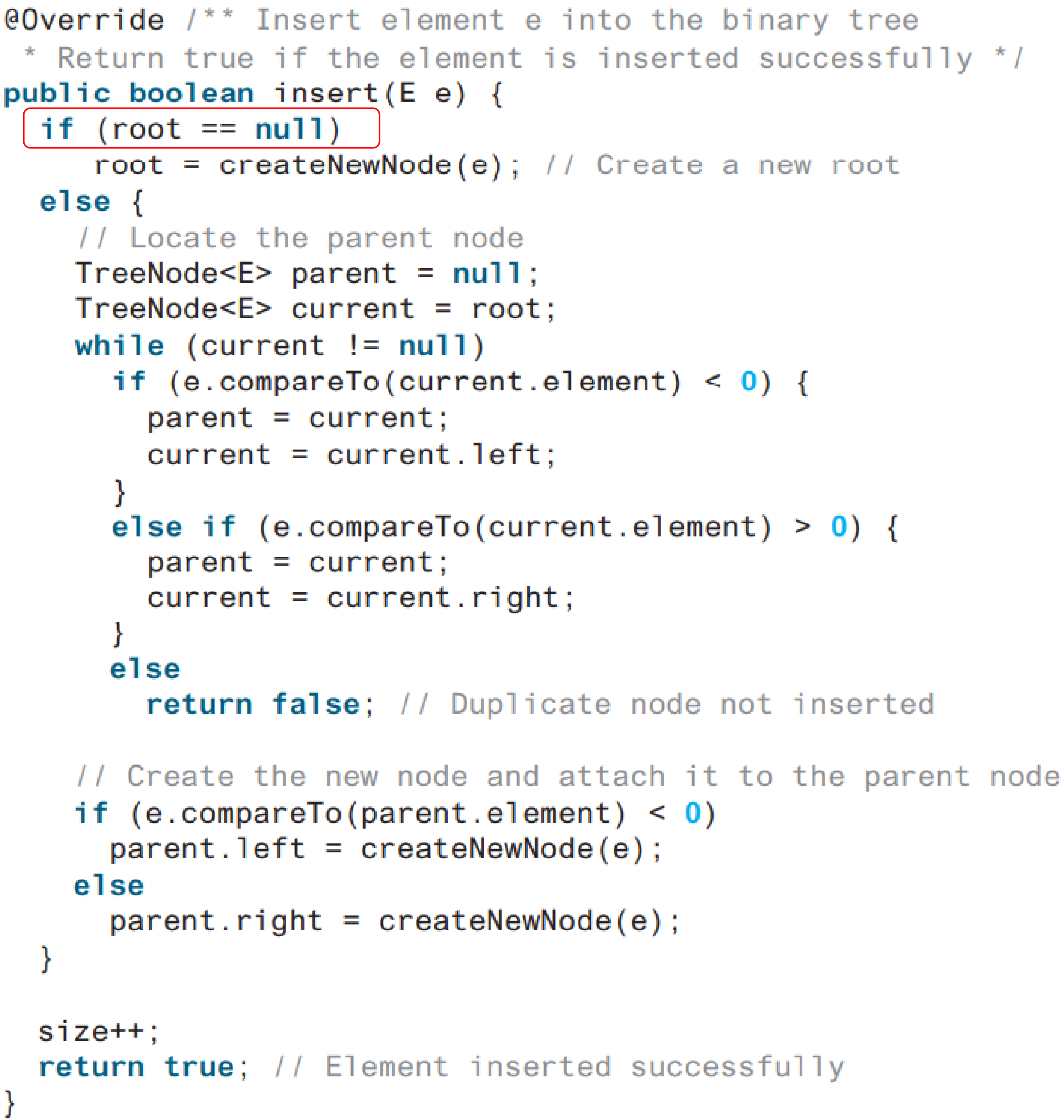


(a) A binary search tree;

(b) The same tree after adding *Chad*.

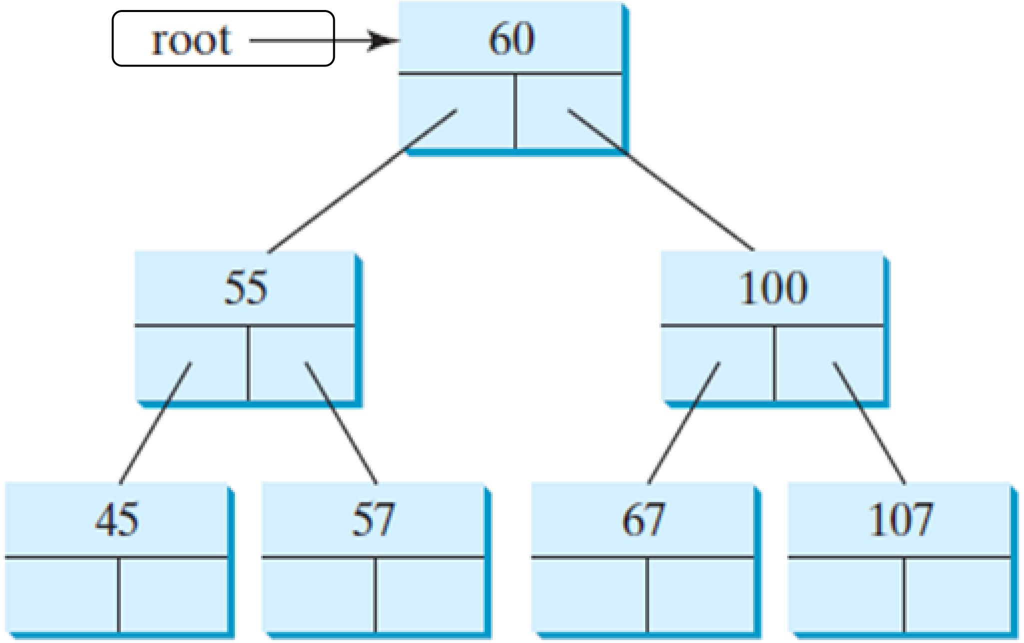


Adding an Entry



```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

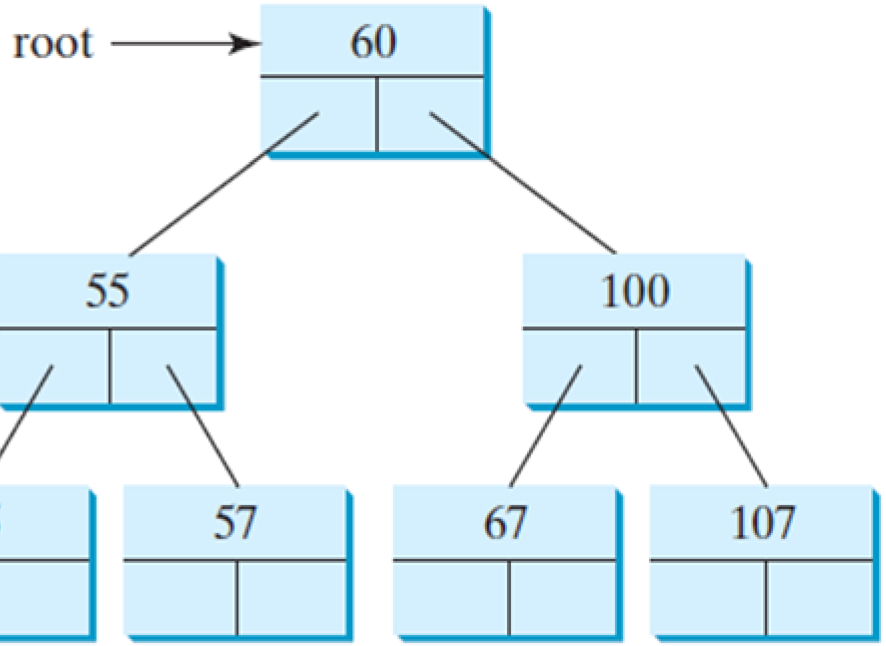
Insert 101 into the following tree.



Adding an Entry

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

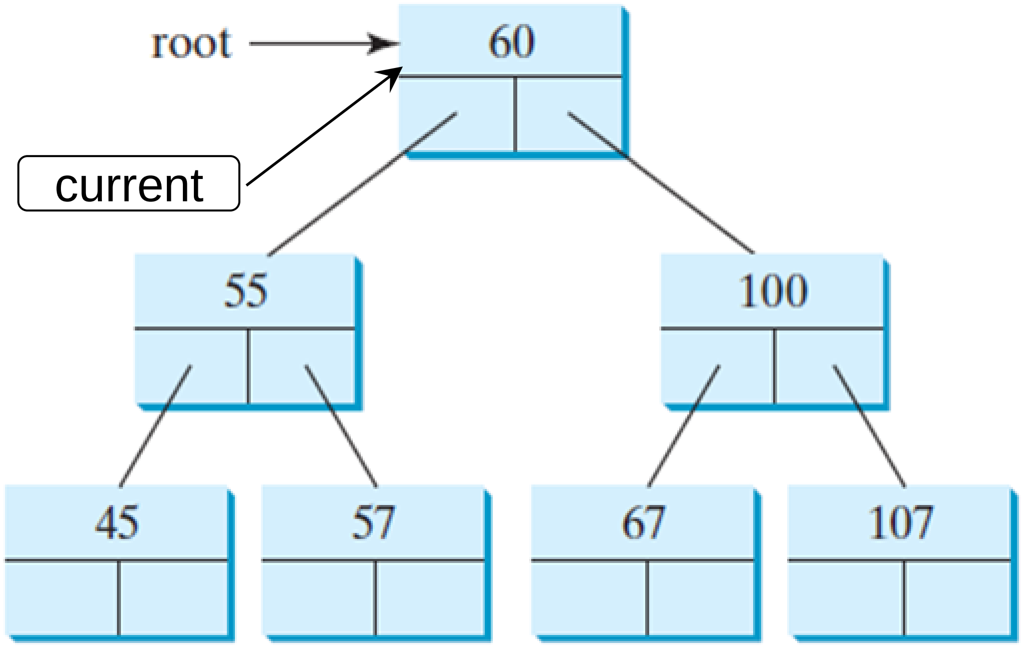
Insert 101 into the following tree.



```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

Adding an Entry

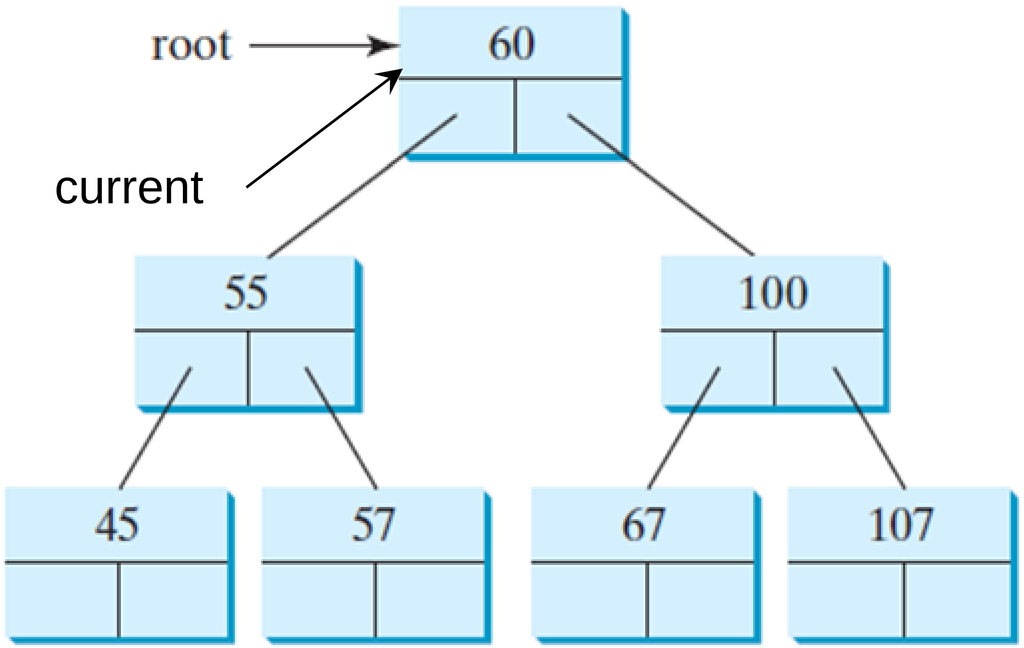
Insert 101 into the following tree.



Adding an Entry

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

Insert 101 into the following tree.

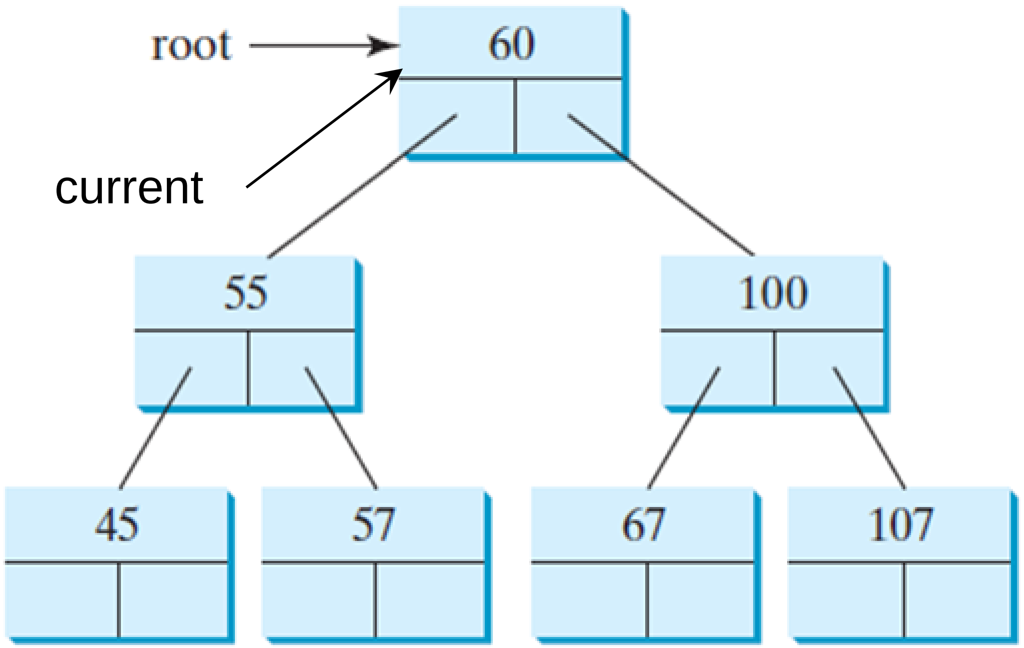


```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 60?

Adding an Entry

Insert 101 into the following tree.

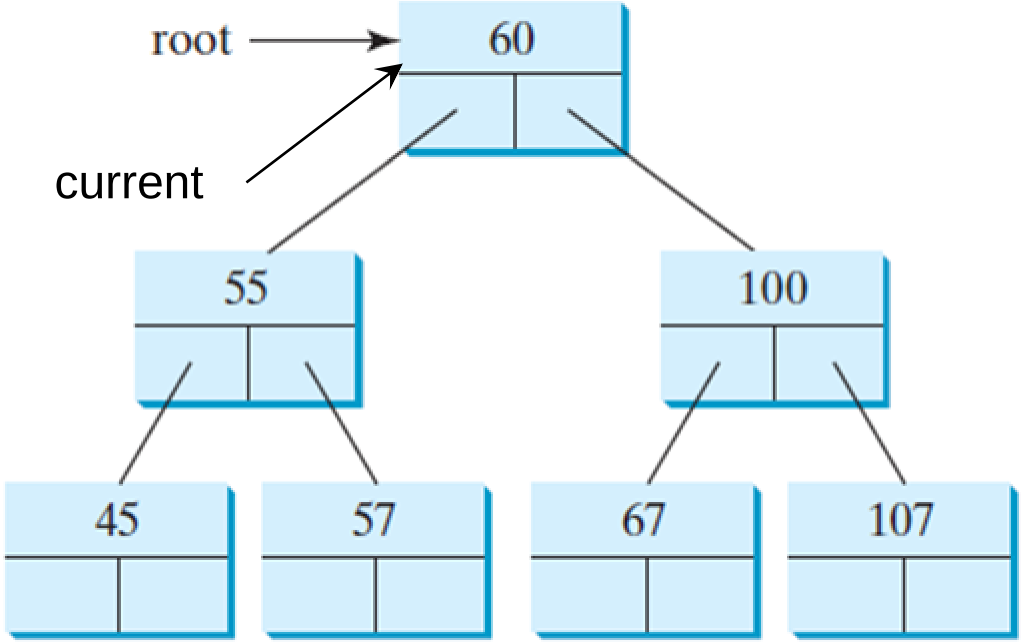


Adding an Entry

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 60?

Insert 101 into the following tree.

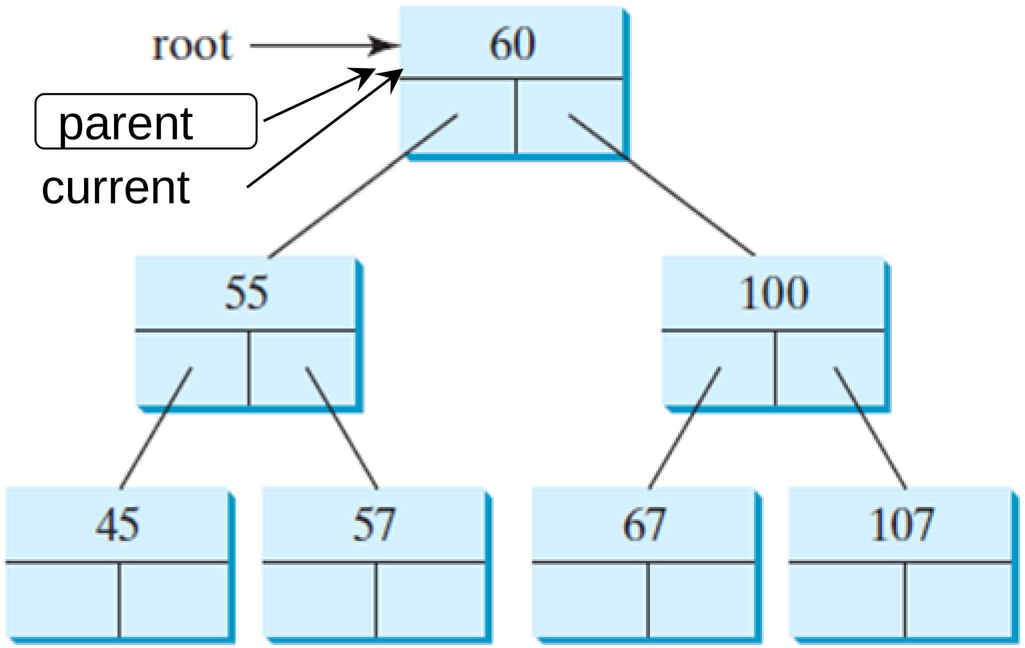


Adding an Entry

Insert 101 into the following tree.

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
    root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 60?

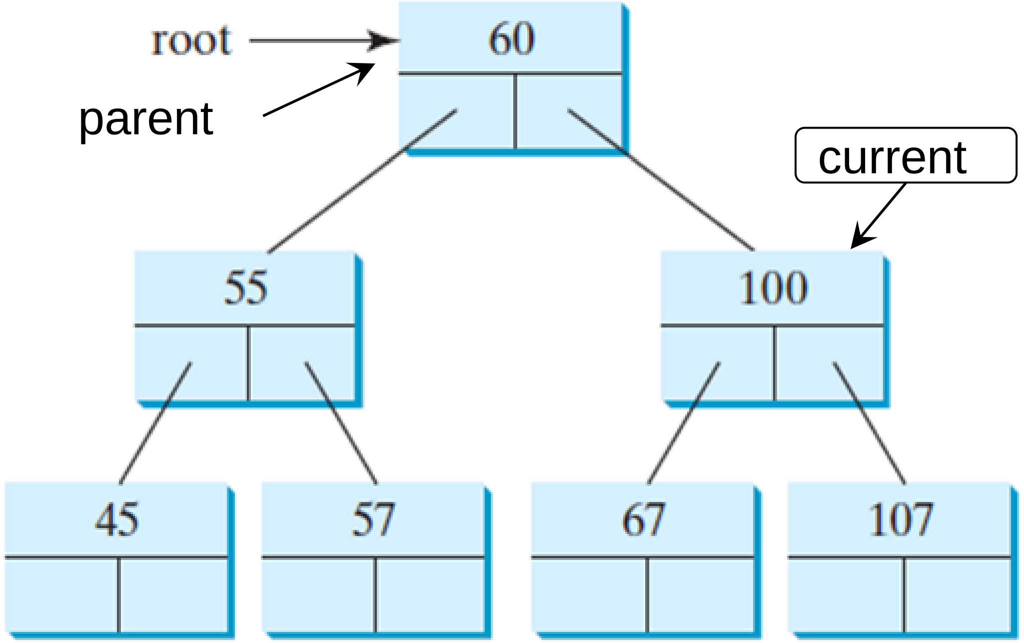


Adding an Entry

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 60?

Insert 101 into the following tree.

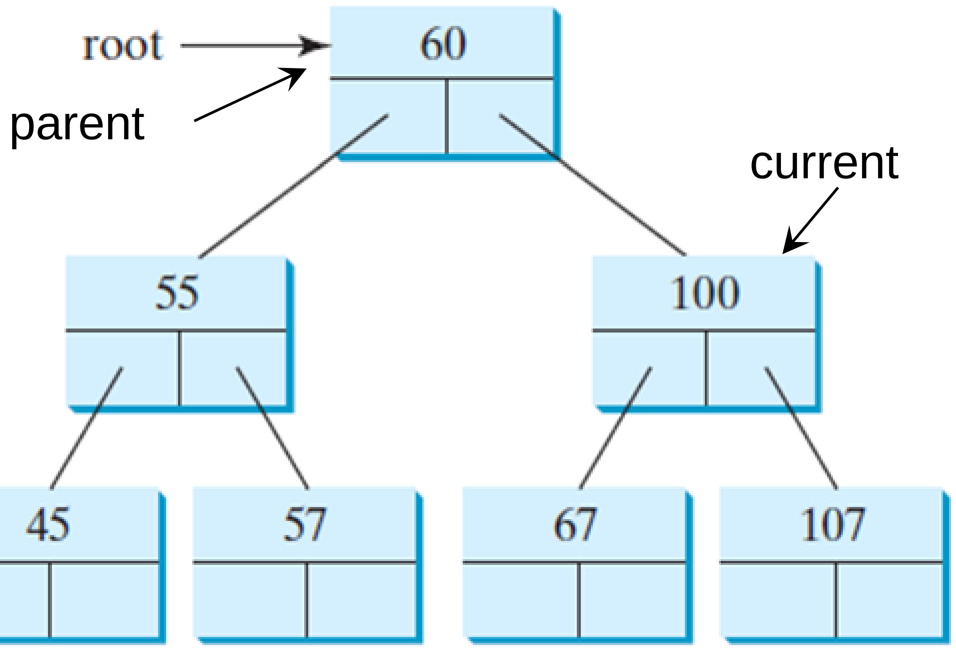


Adding an Entry

Insert 101 into the following tree.

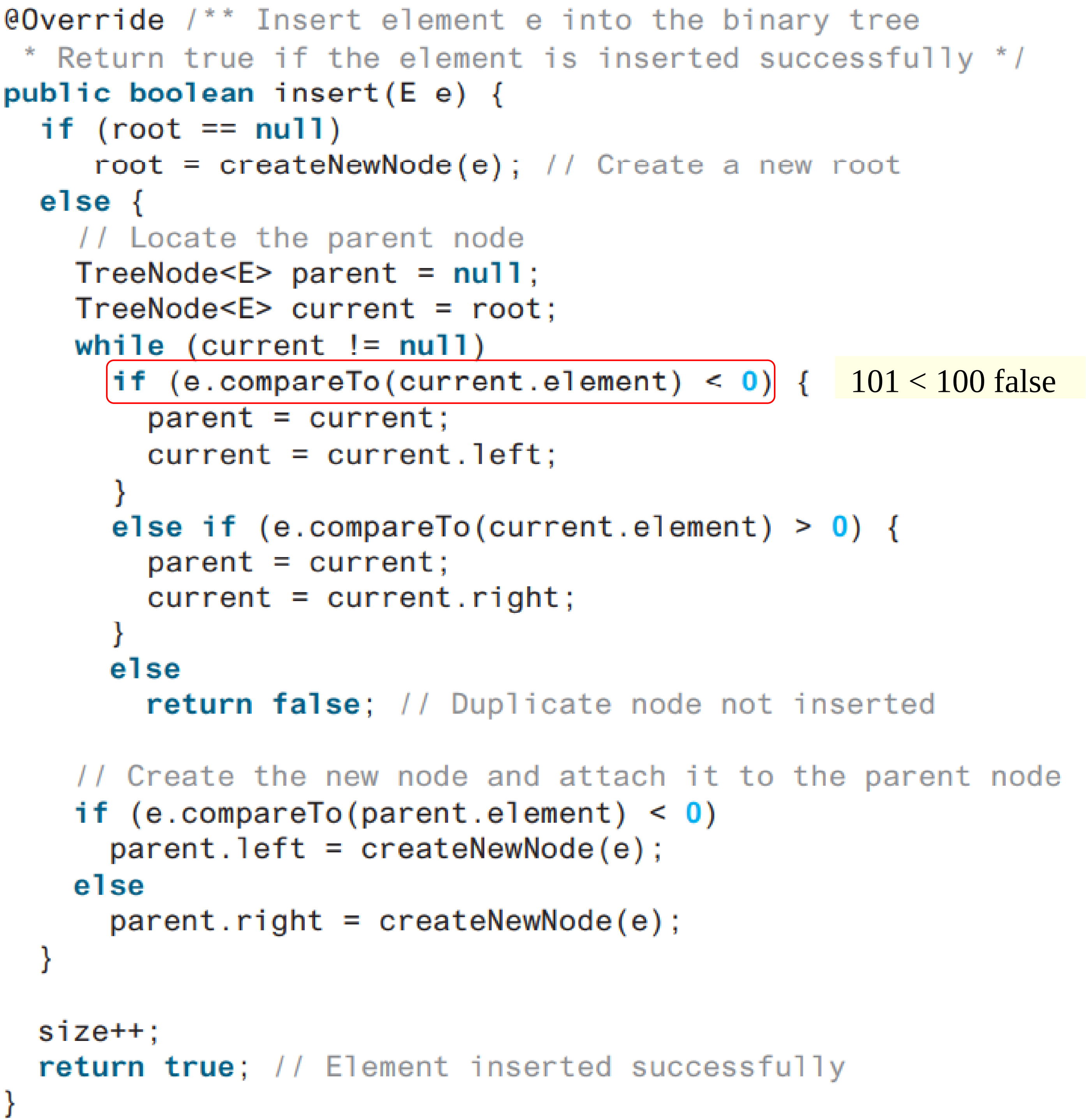
```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 60?



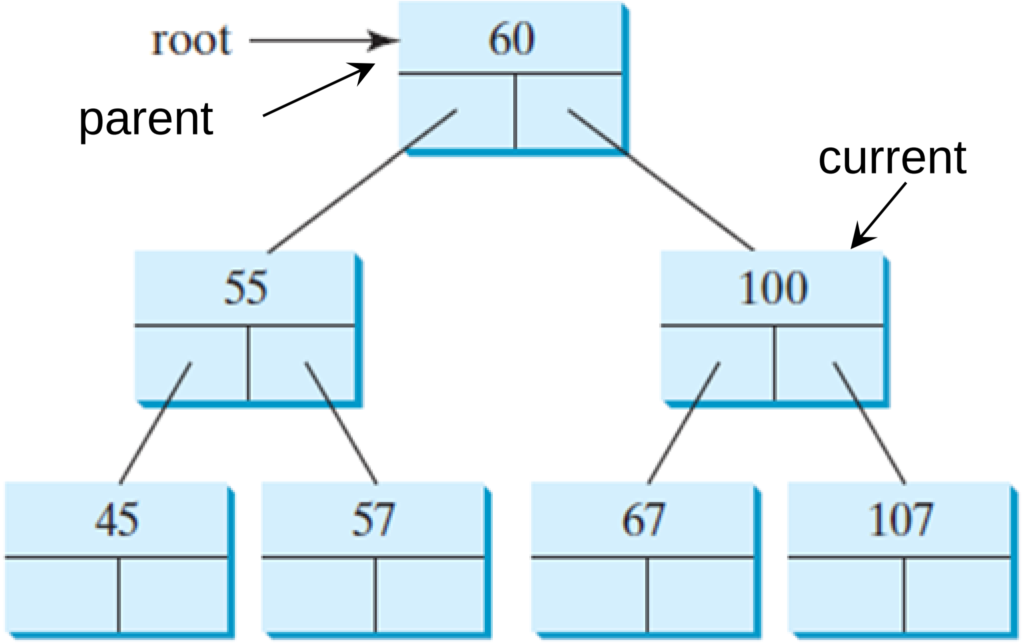
Adding an Entry

Insert 101 into the following tree.



```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 100 false

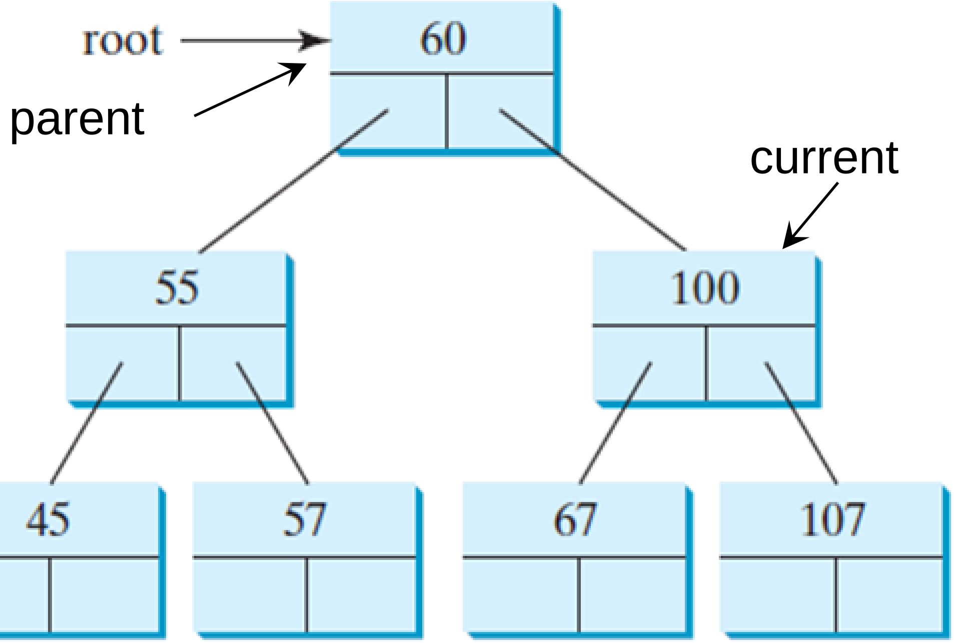


Adding an Entry

Insert 101 into the following tree.

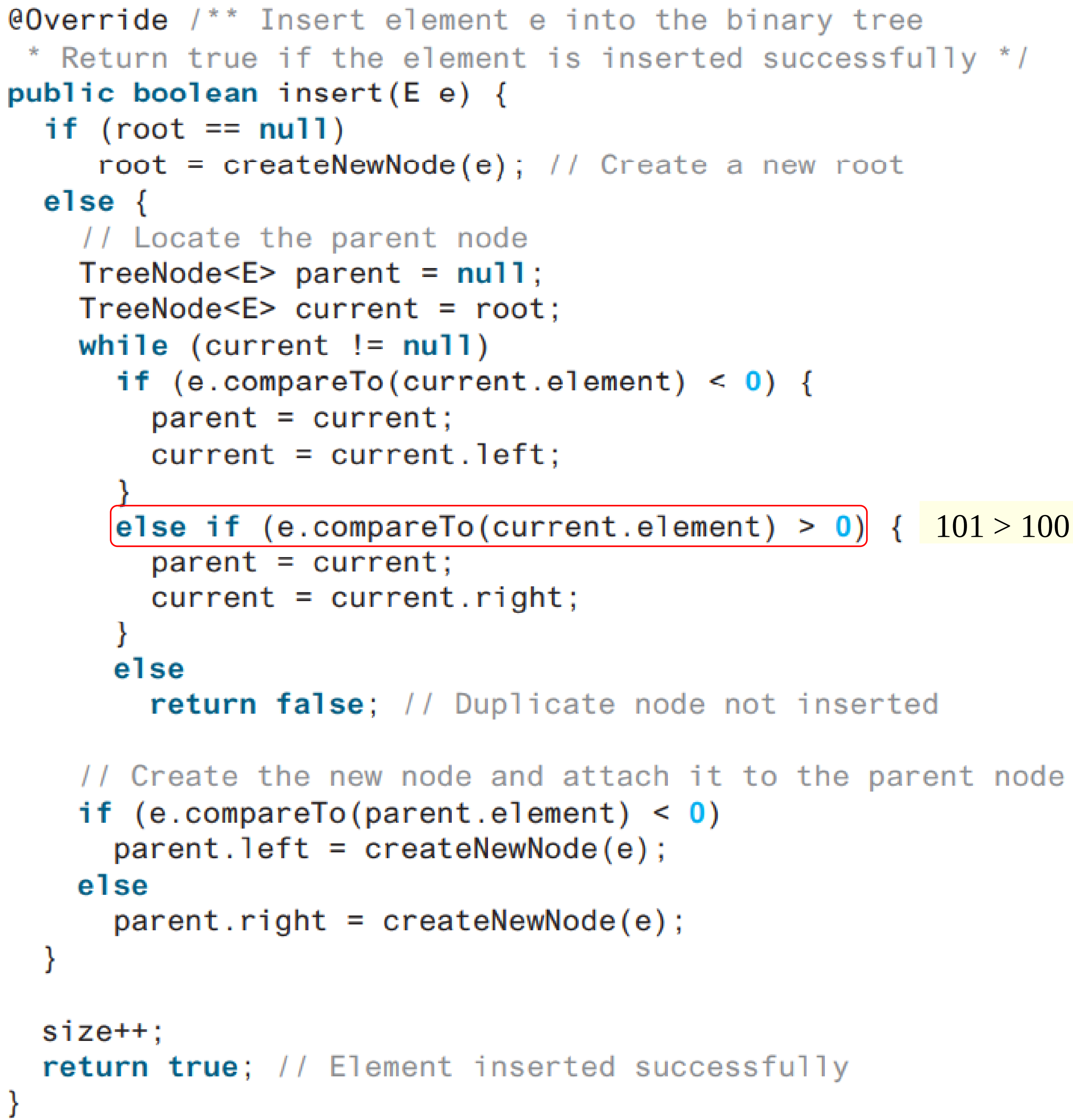
```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 100 true



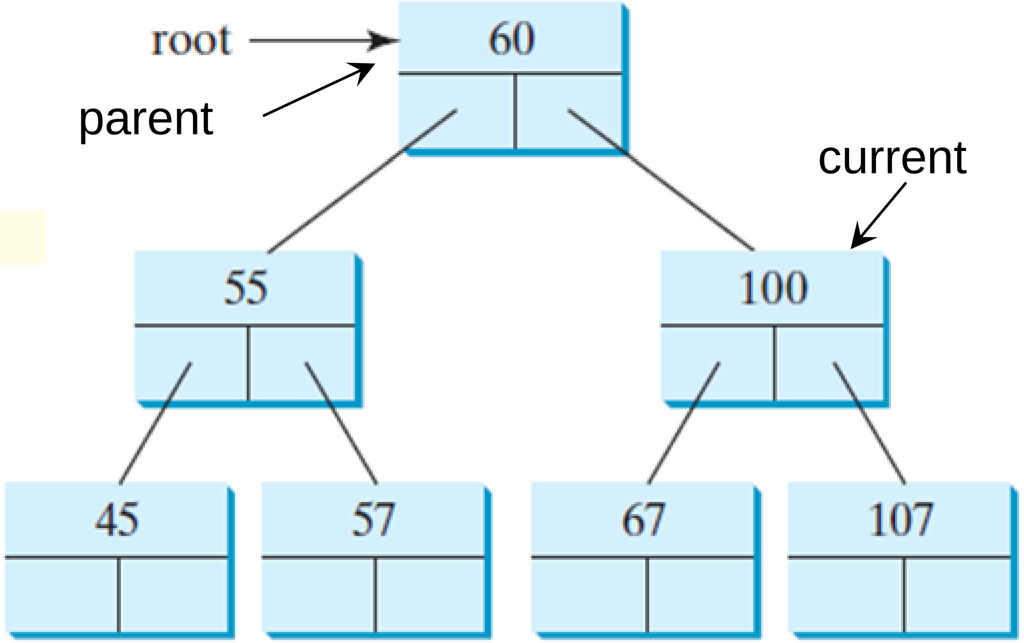
Adding an Entry

Insert 101 into the following tree.



```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 100 true

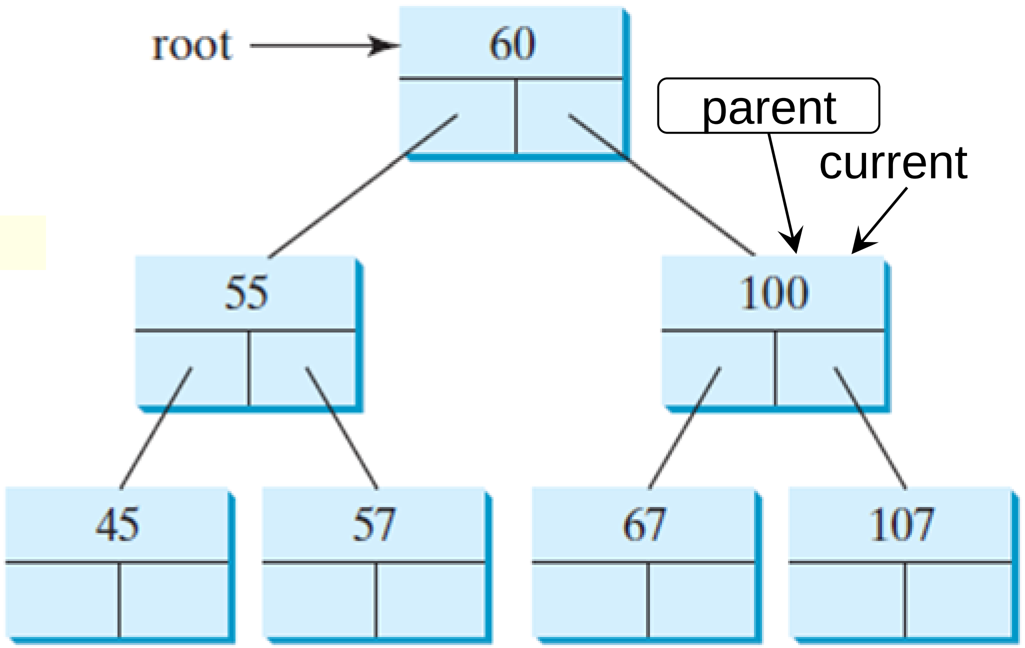


Adding an Entry

Insert 101 into the following tree.

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 100 true

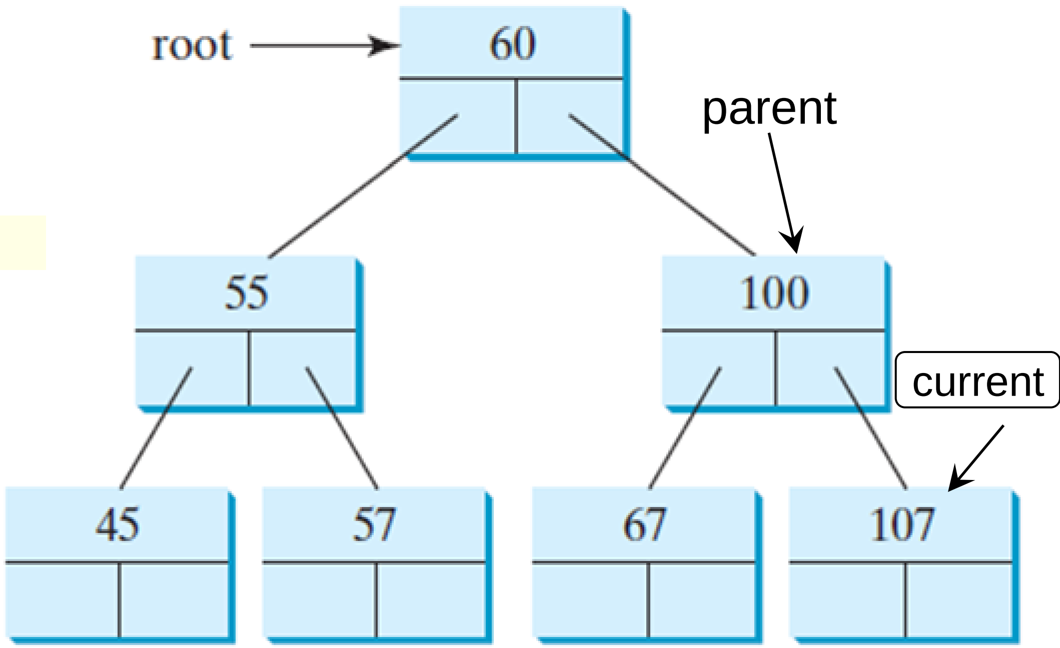


Adding an Entry

Insert 101 into the following tree.

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 100 true

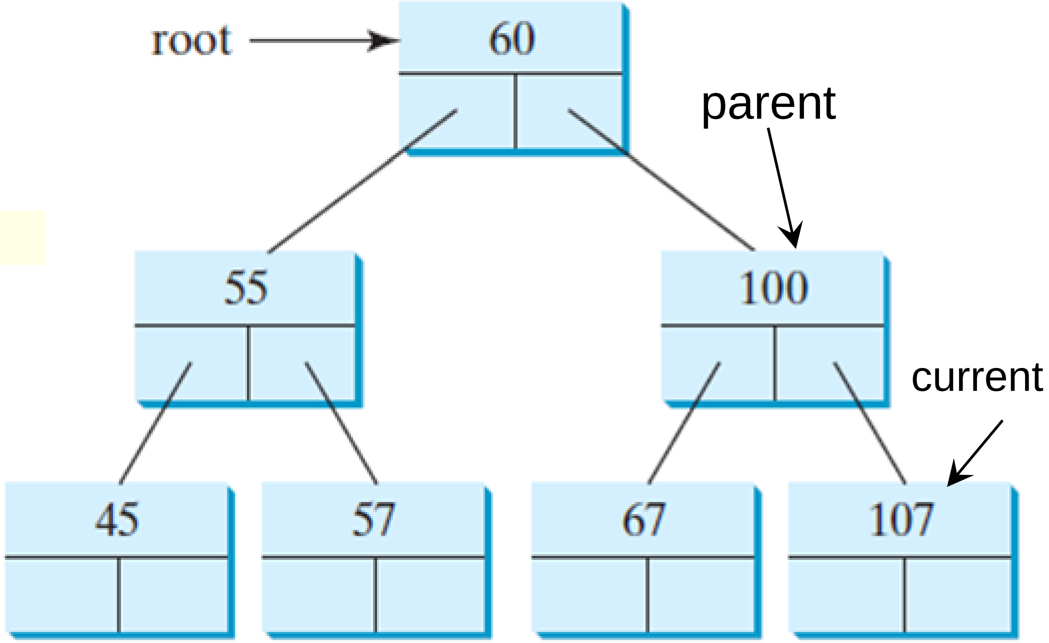


Adding an Entry

Insert 101 into the following tree.

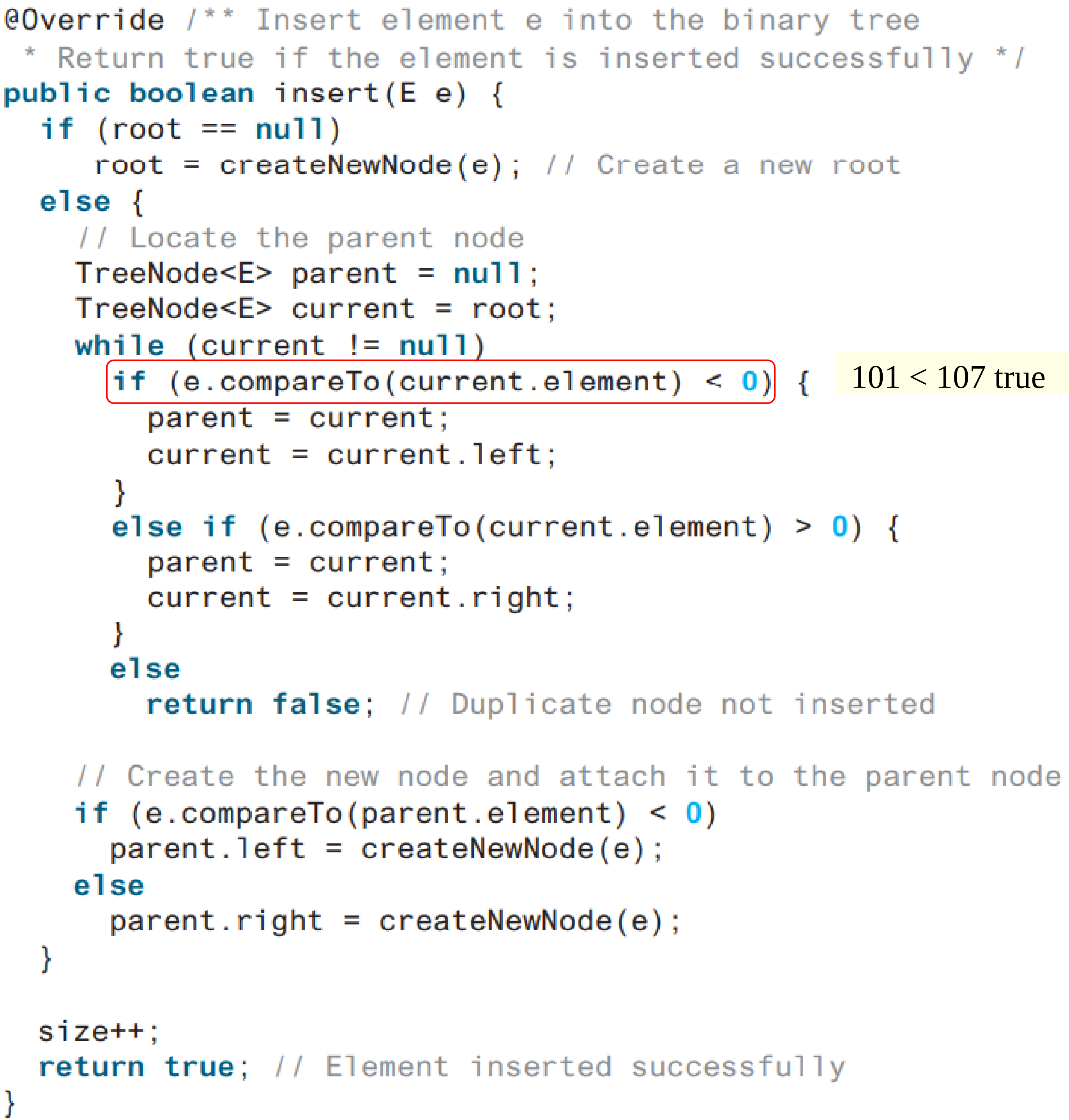
```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 > 100 true



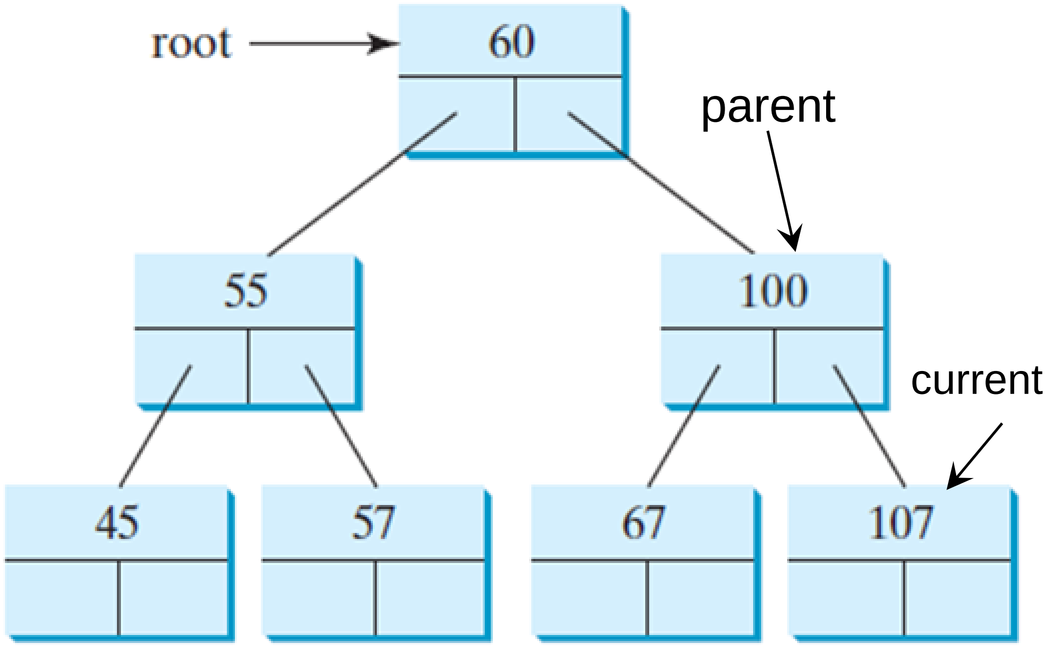
Adding an Entry

Insert 101 into the following tree.



```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 107 true

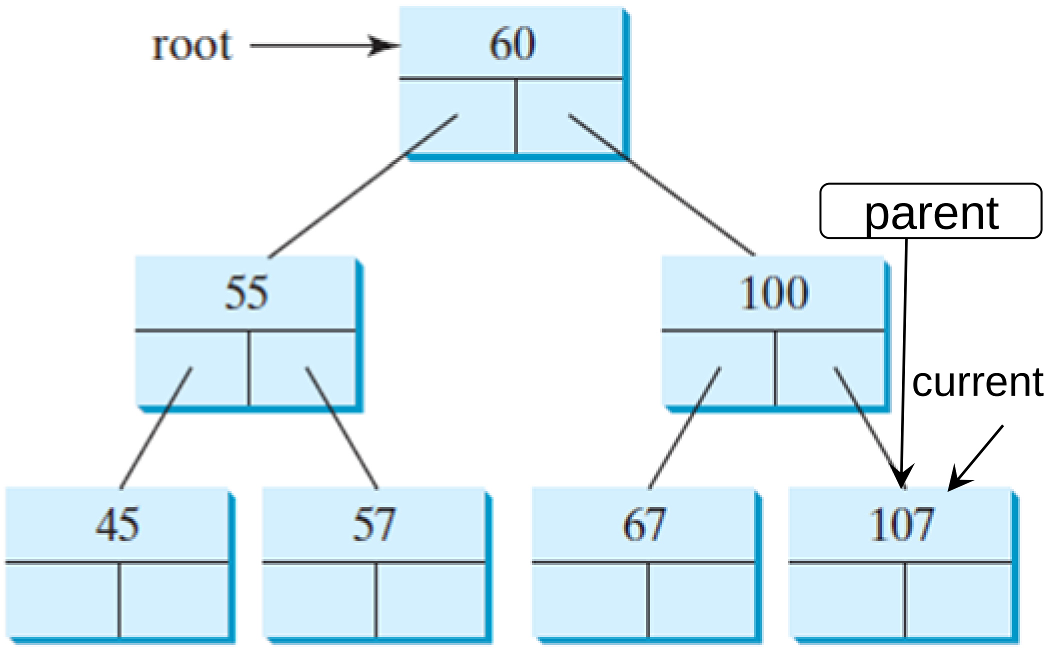


Adding an Entry

Insert 101 into the following tree.

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 107 true

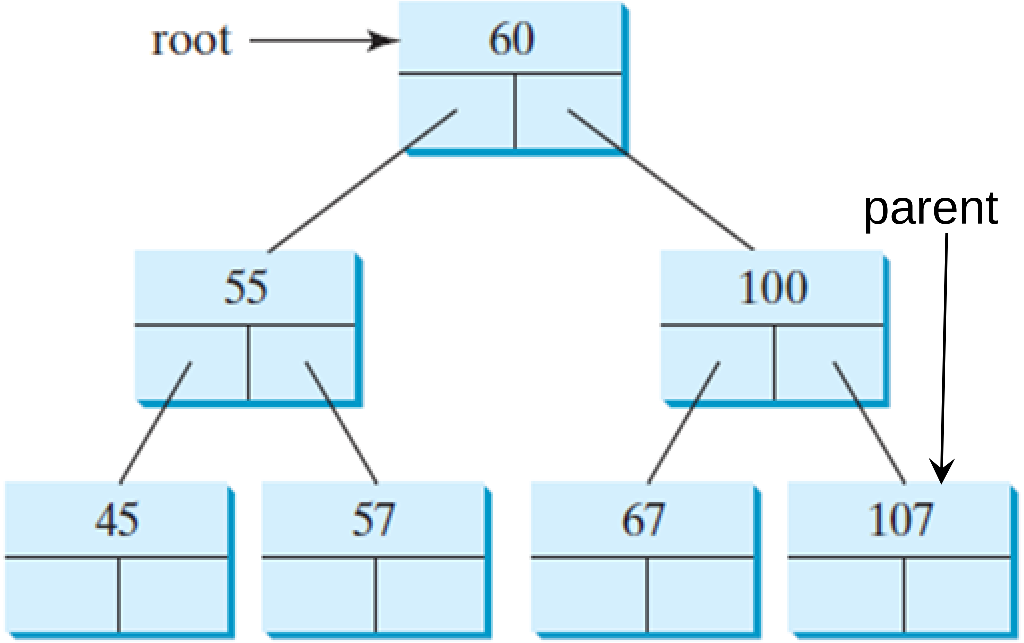


Adding an Entry

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 107 true

Insert 101 into the following tree.



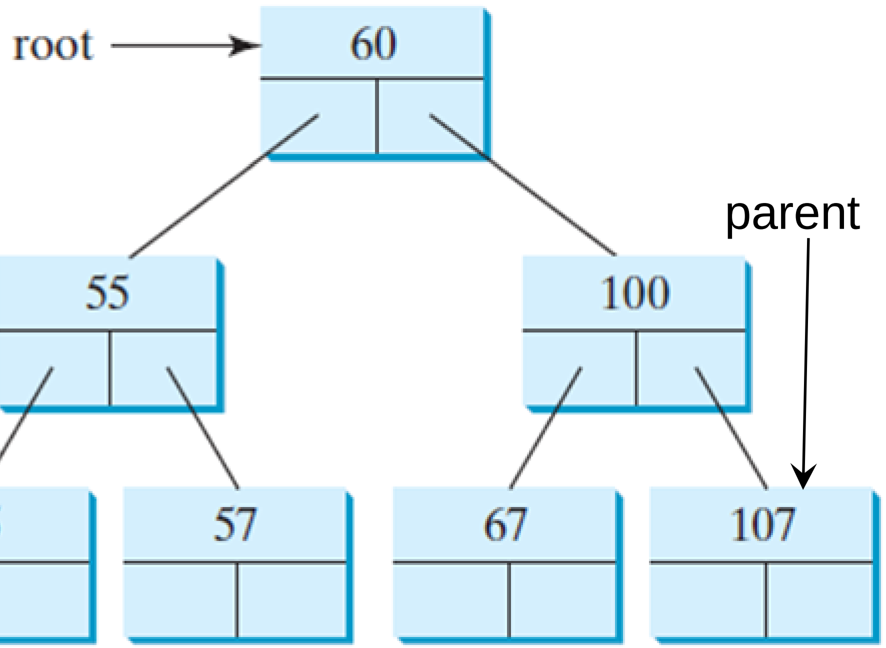
Since current.left is null,current becomes null

Adding an Entry

Insert 101 into the following tree.

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

current is null now



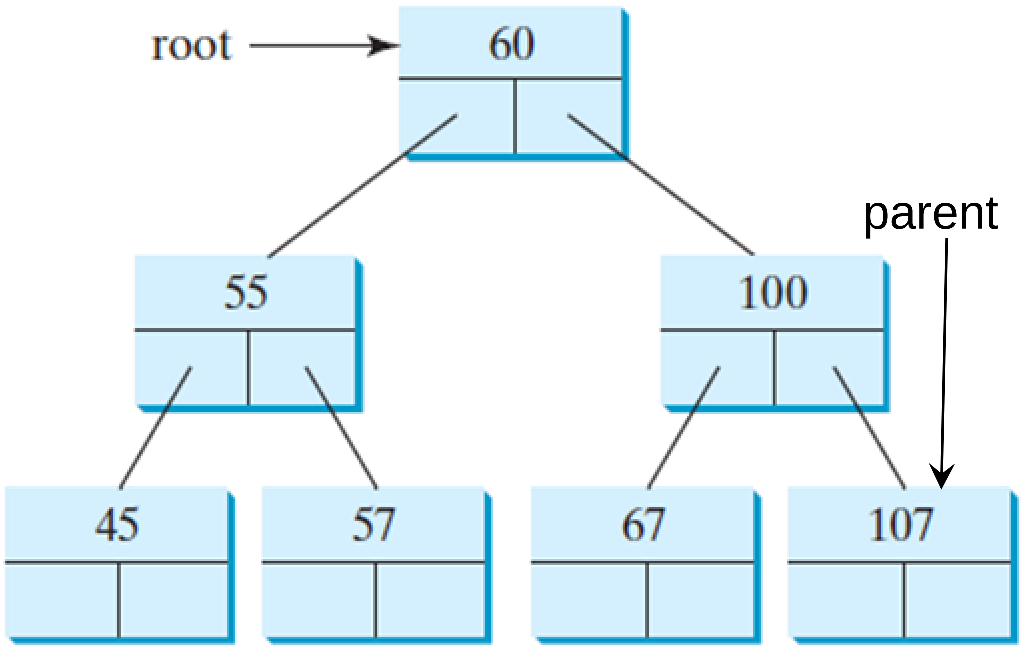
Since current.left is null,current becomes null

Adding an Entry

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 107 true

Insert 101 into the following tree.



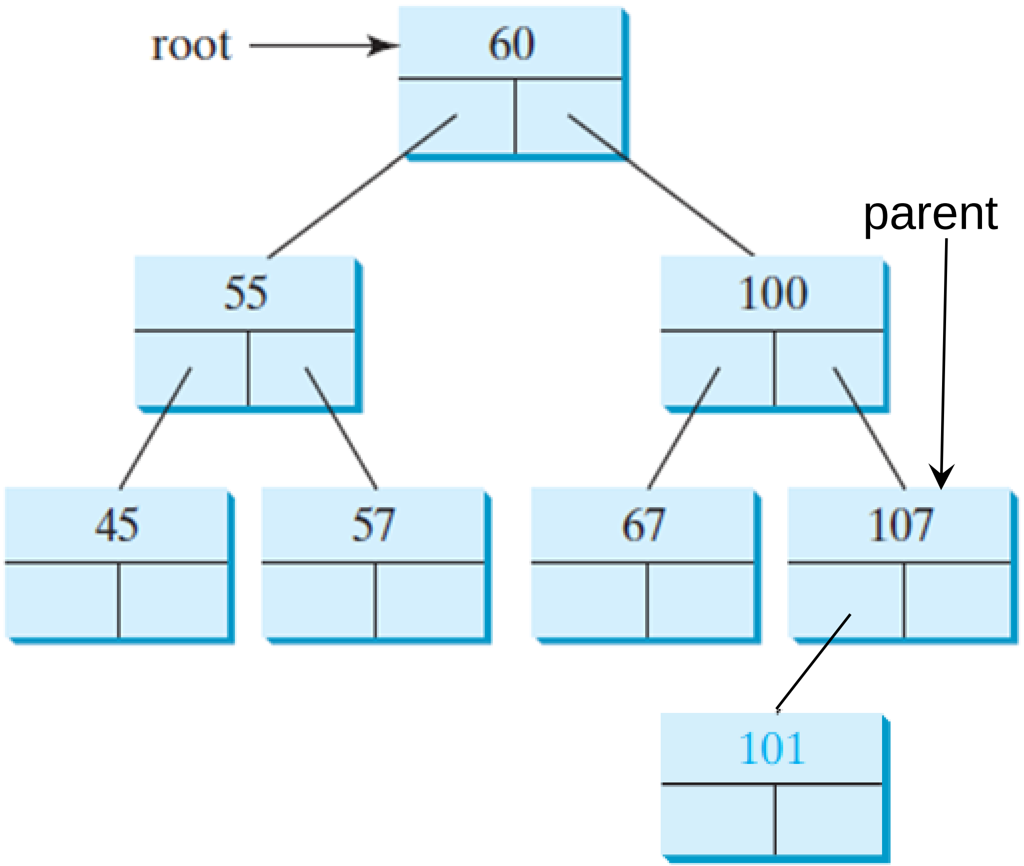
Since current.left is null,current becomes null

```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 107 true

Adding an Entry

Insert 101 into the following tree.

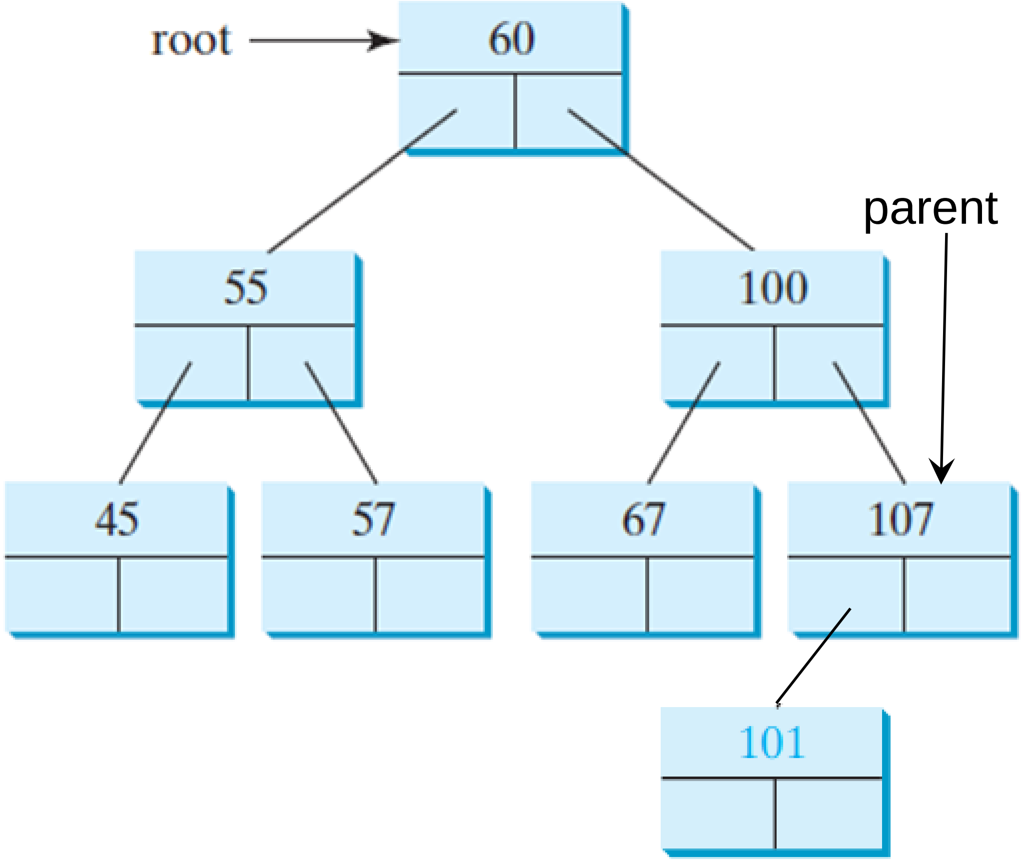


```
@Override /** Insert element e into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
     root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted successfully
}
```

101 < 107 true

Adding an Entry

Insert 101 into the following tree.

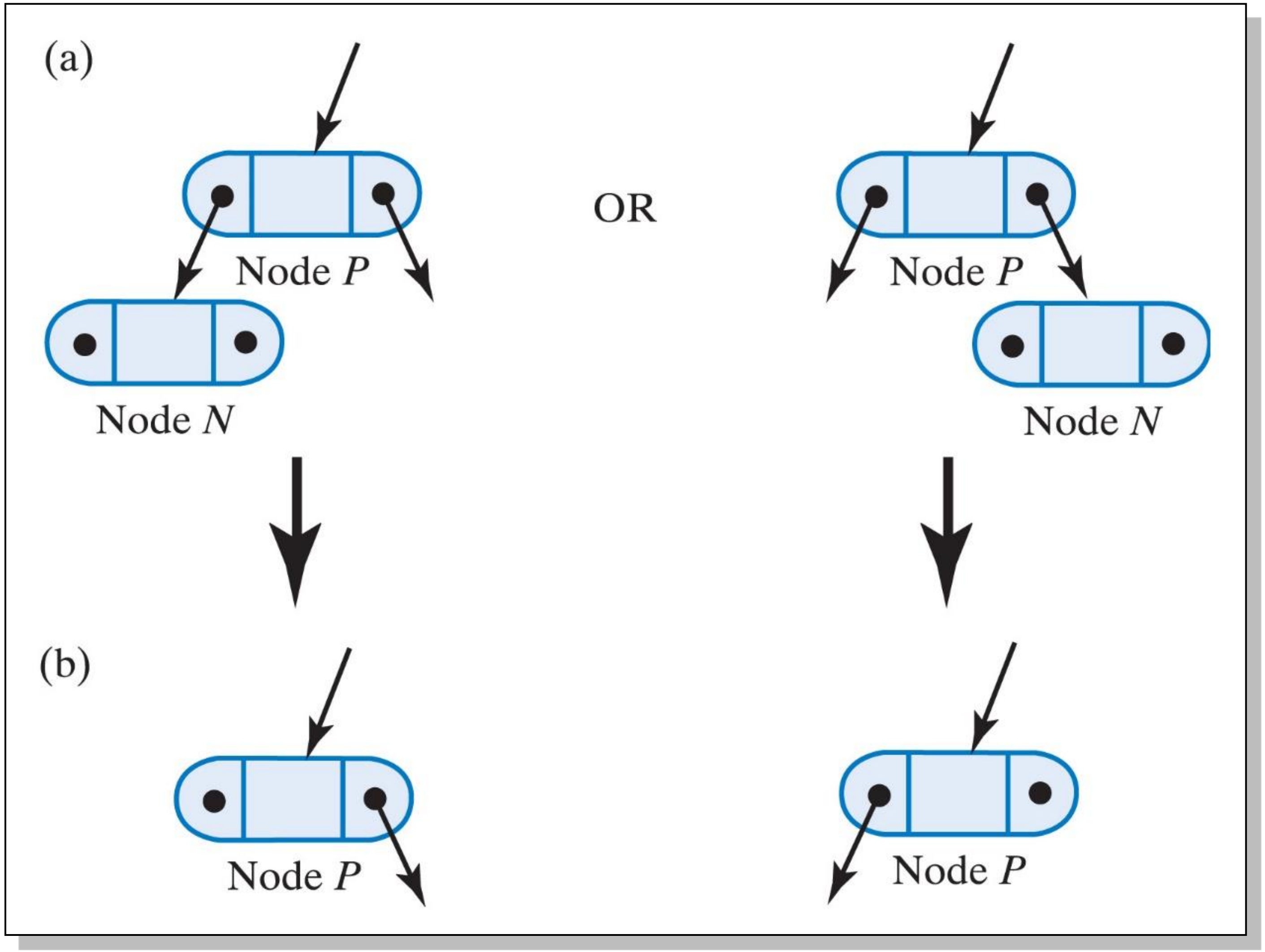


Removing an Entry

- The `remove` method must receive an entry to be matched in the tree
  - If found, it is removed
  - Otherwise the method returns null
- Three cases

Case0:	The node has no children, it is a leaf (simplest case)
Case1:	The node has one child
Case2:	The node has two children

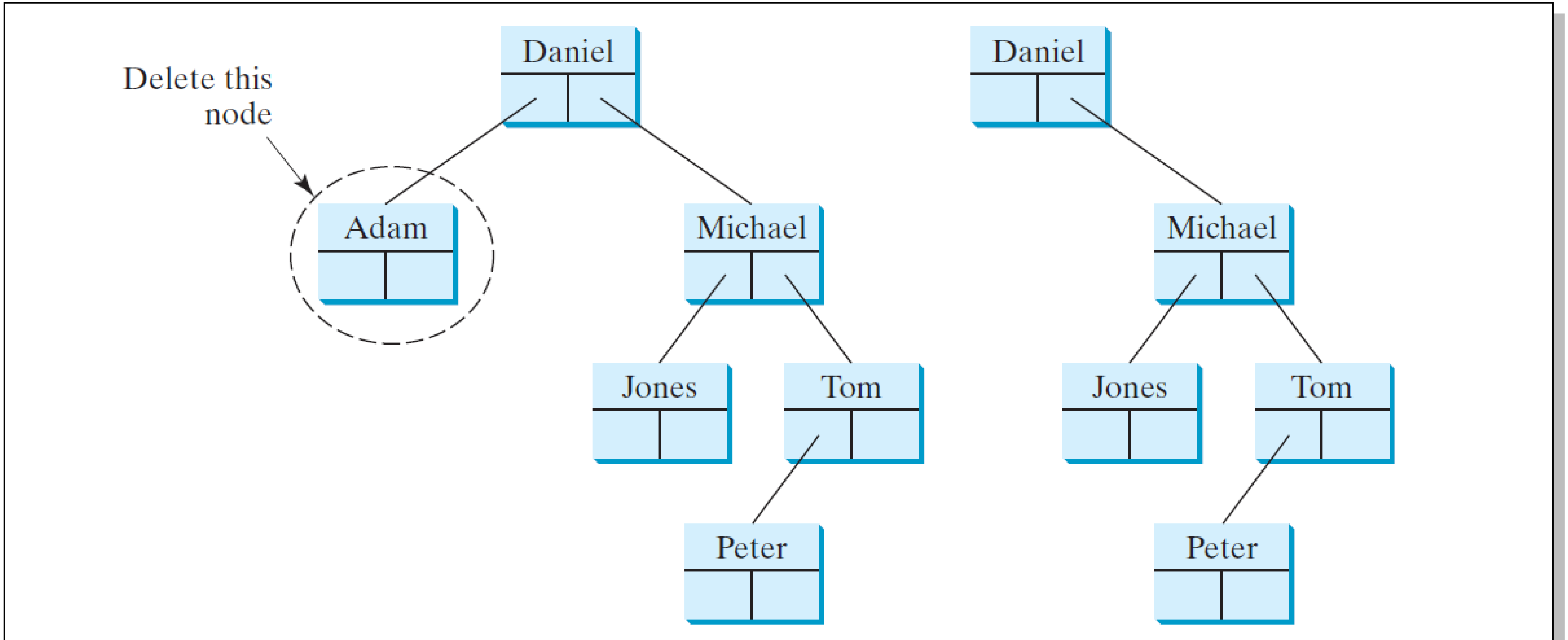
Case0: Removing an Entry, Node a Leaf



(a) Two possible configurations of leaf node *N*;

(b) the resulting two possible configurations after removing node *N*.

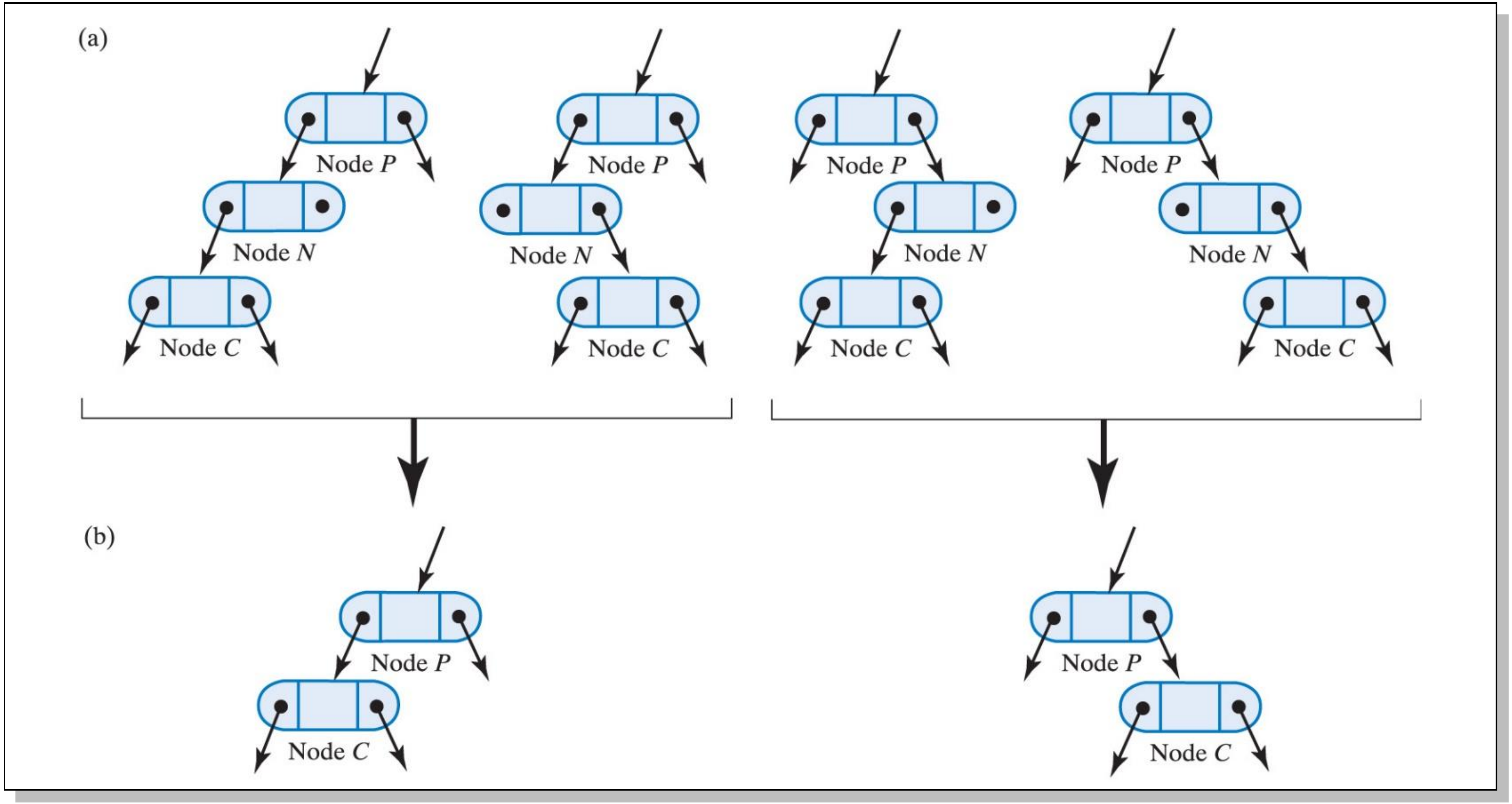
Case0: Removing an Entry, Node a Leaf



(a) Deleting Adam (b) After Adam is deleted

Deleting Adam falls into Case 0

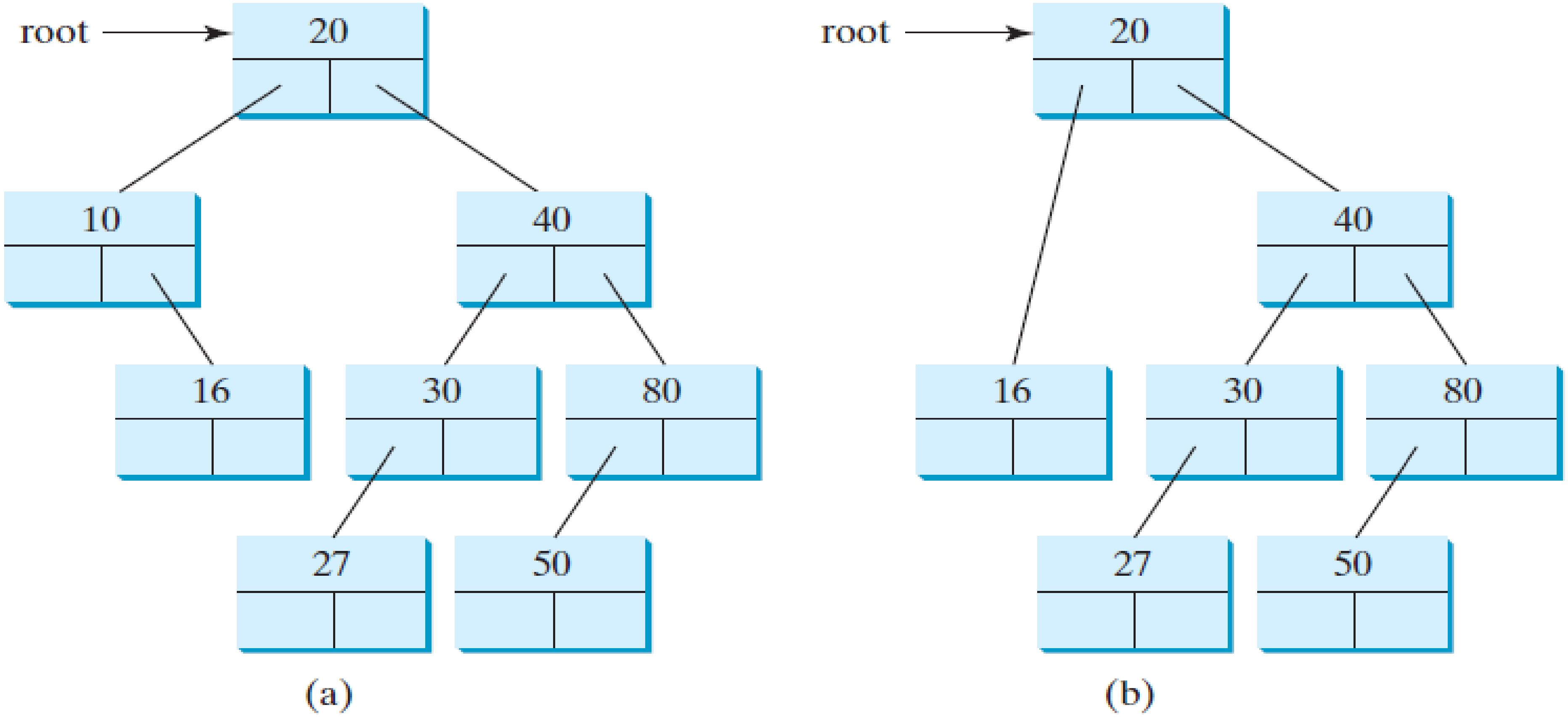
Case1: Removing an Entry, Node Has One Child



(a) Two possible configurations of leaf node *N*;

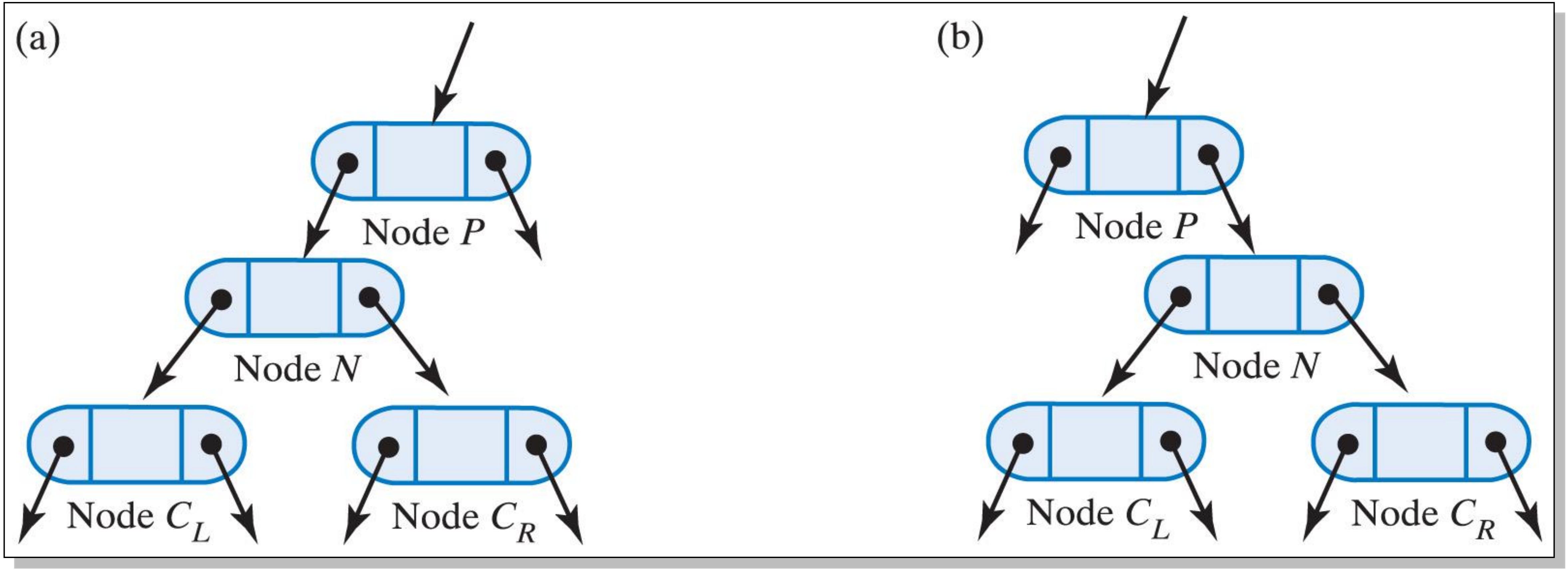
(b) the resulting two possible configurations after removing node *N*.

Case1: Removing an Entry, Node Has One Child



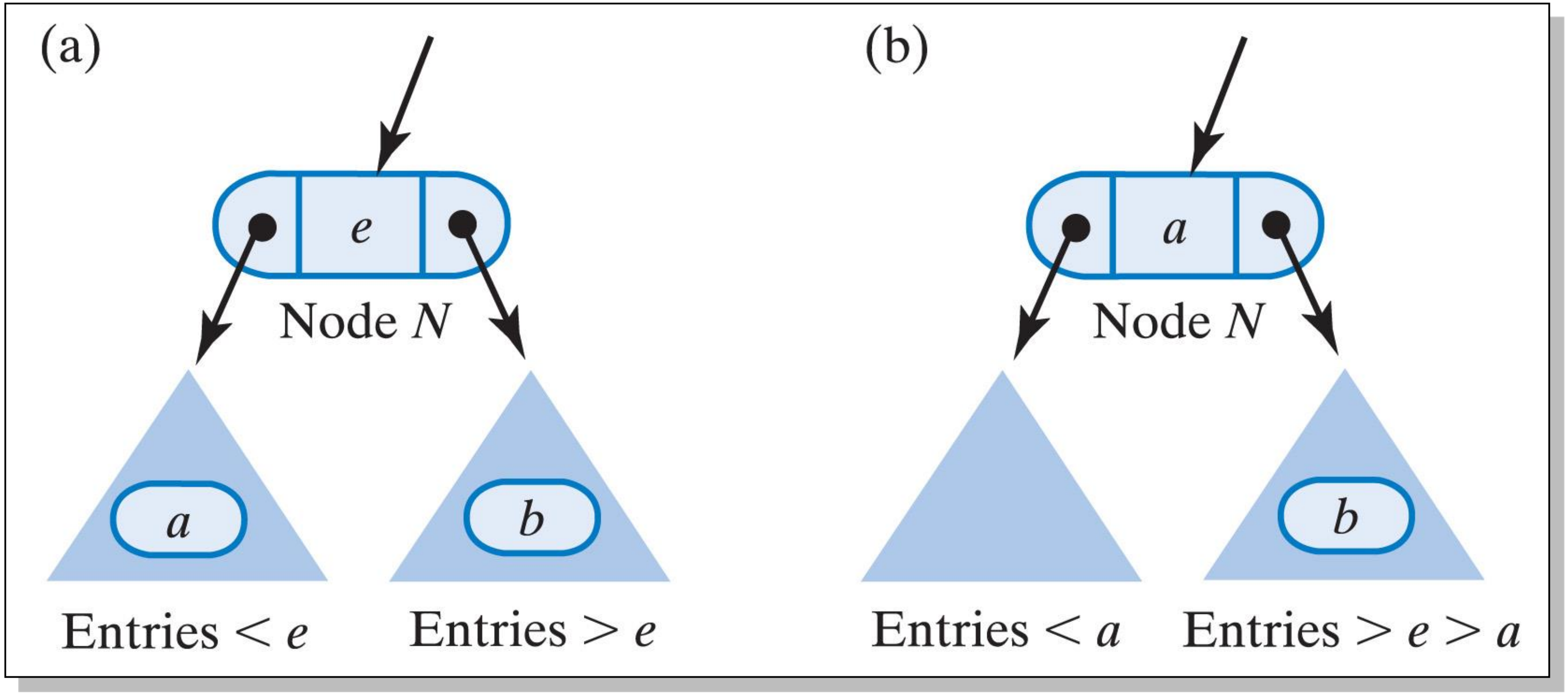
Case 1: Deleting node 10 from (a) results in (b).

Case2: Removing an Entry, Node Has Two Children



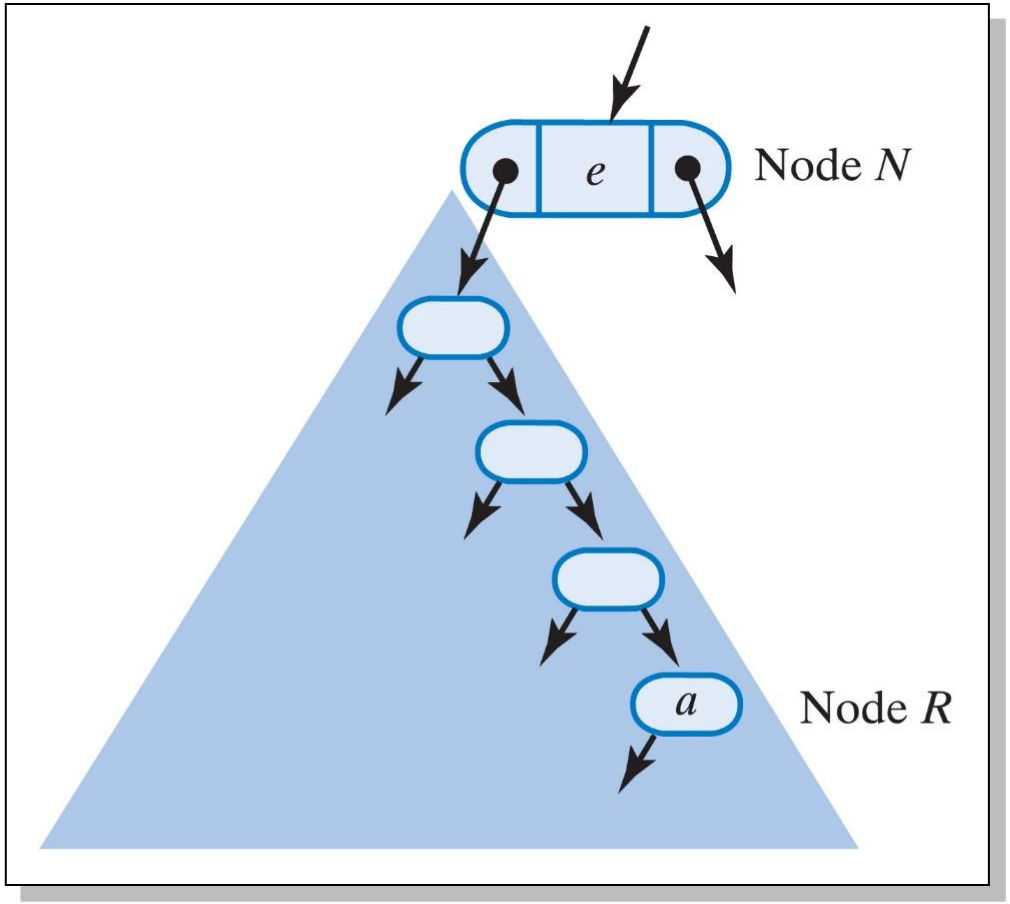
Two possible configurations of node *N* that has two children.

Removing an Entry, Node Has Two Children



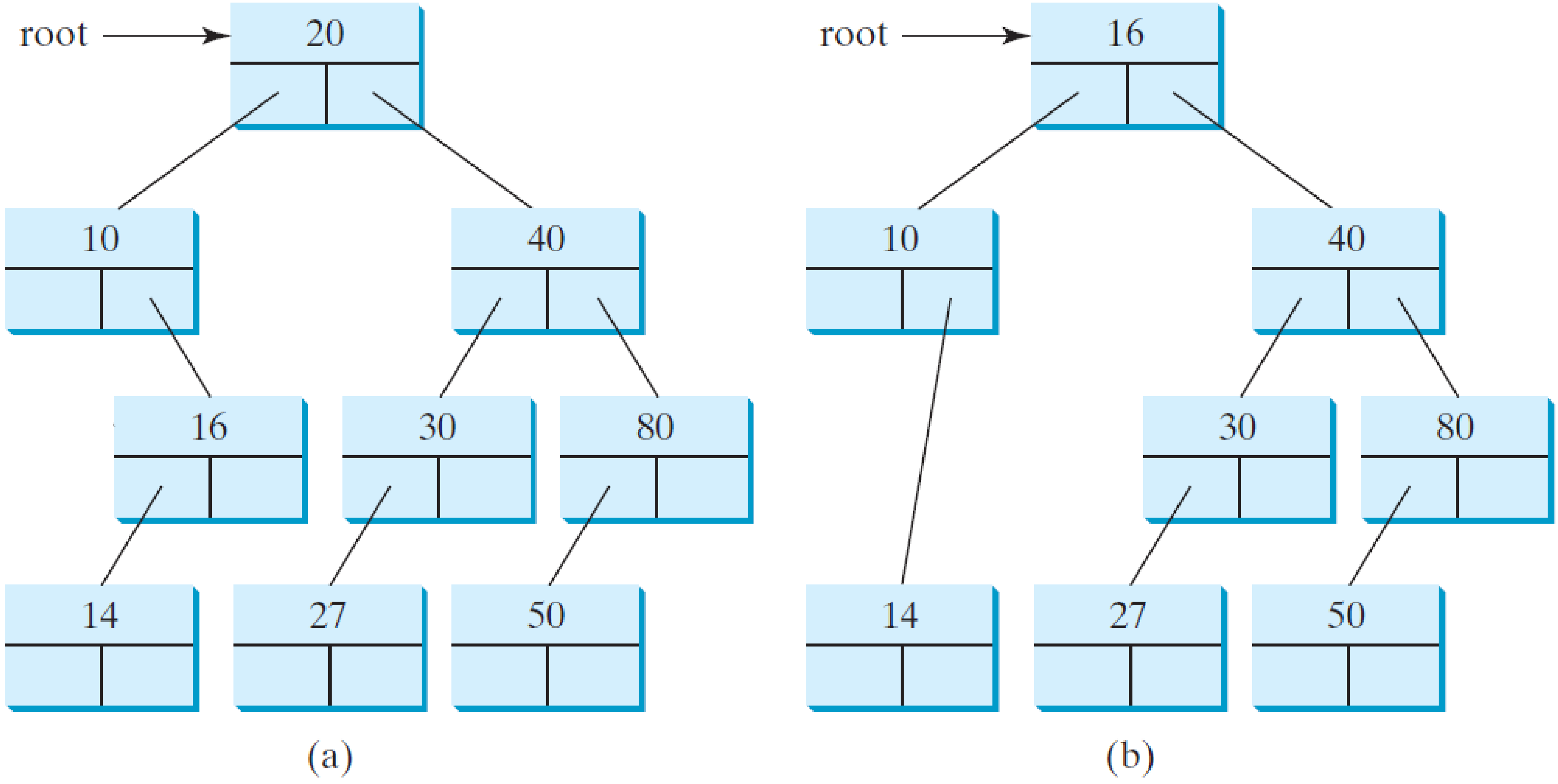
Node *N* and its subtrees; (a) entry *a* is immediately before *e, b* is immediately after *e*; (b) after deleting the node that contained *a* and replacing *e* with *a.*

Case2: Removing an Entry, Node Has Two Children



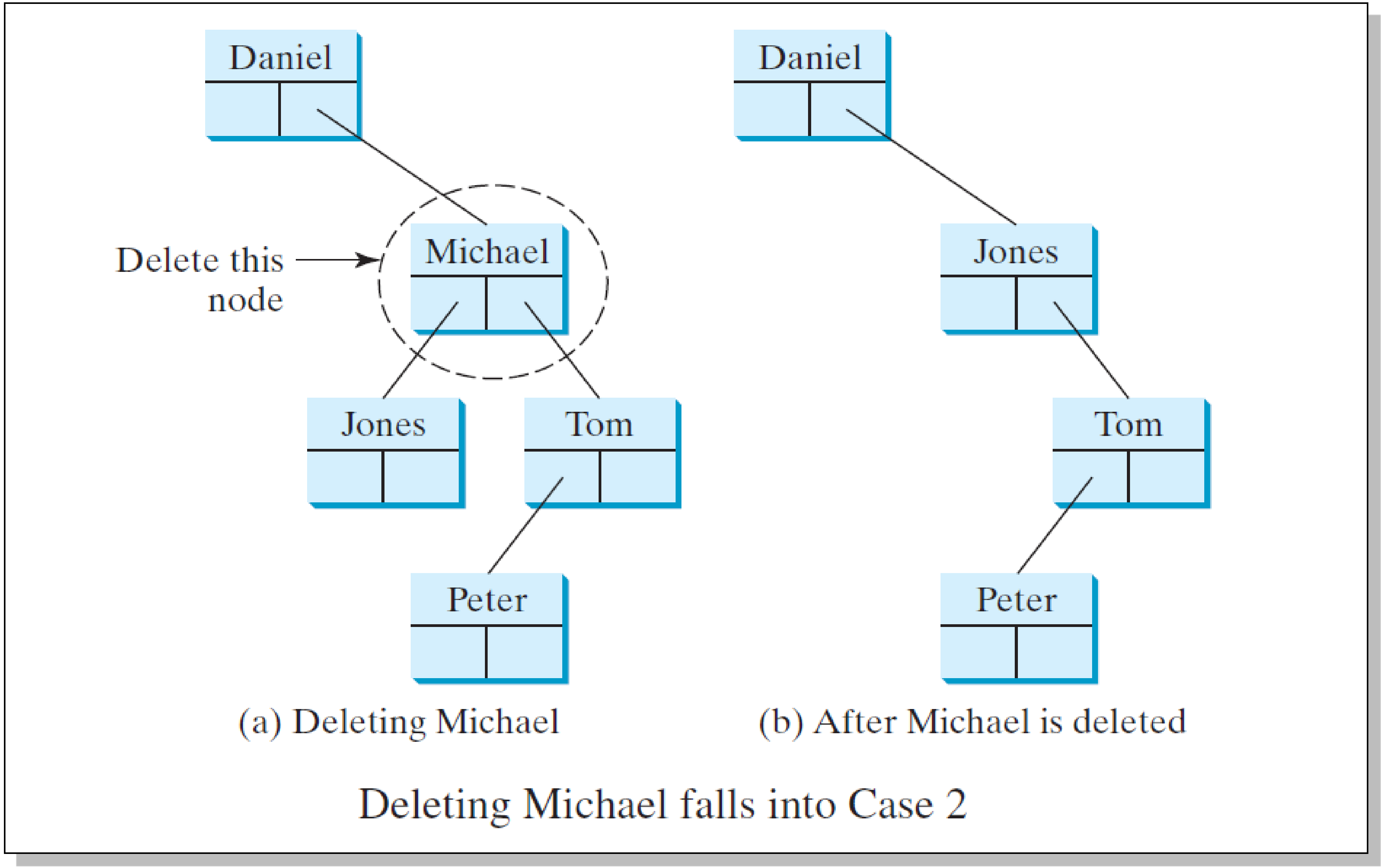
The largest entry *a* in node *N*'s left subtree occurs in the subtree's rightmost node *R*.

Case2: Removing an Entry, Node Has Two Children



Case 2: Deleting node **20** from (a) results in (b).

Case2: Removing an Entry, Node Has Two Children

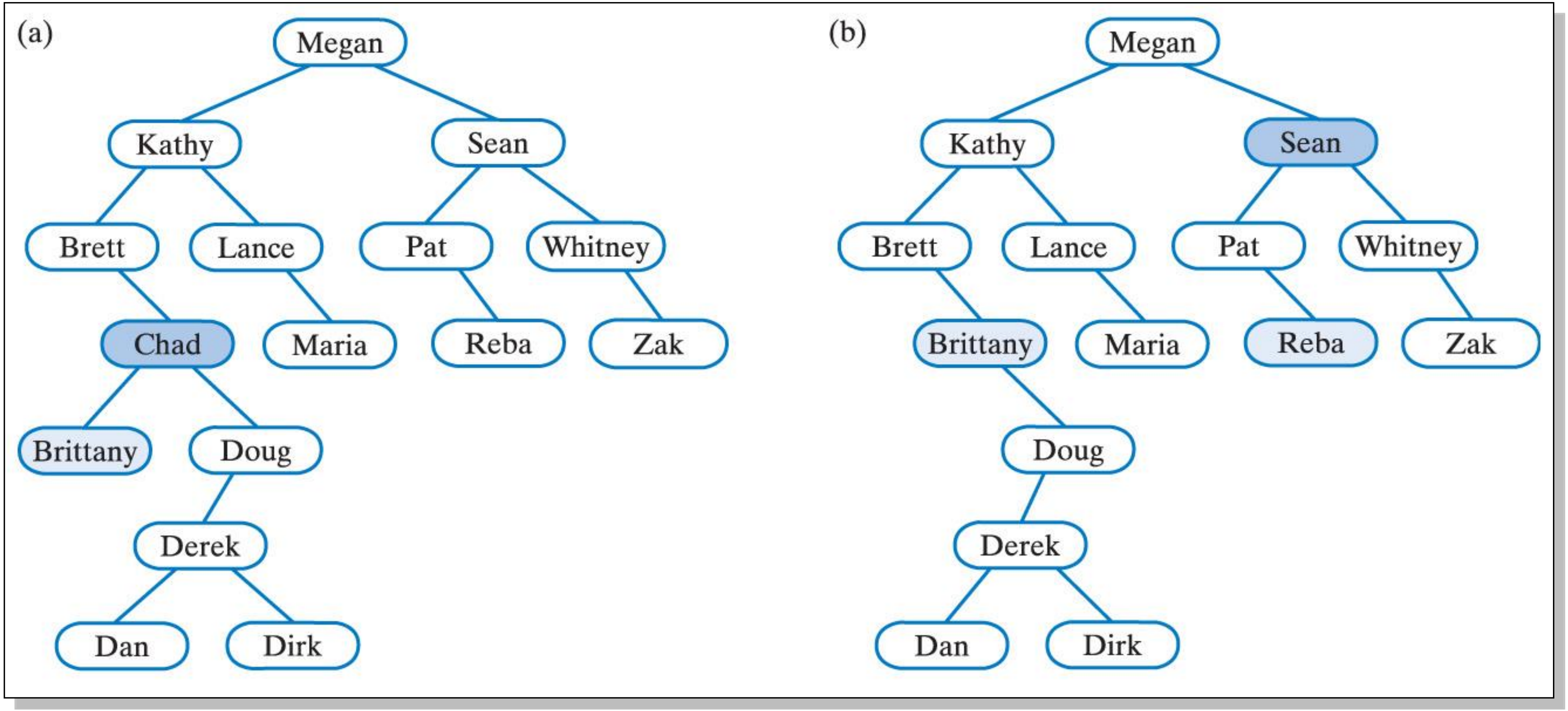


(a) Deleting Michael

(b) After Michael is deleted

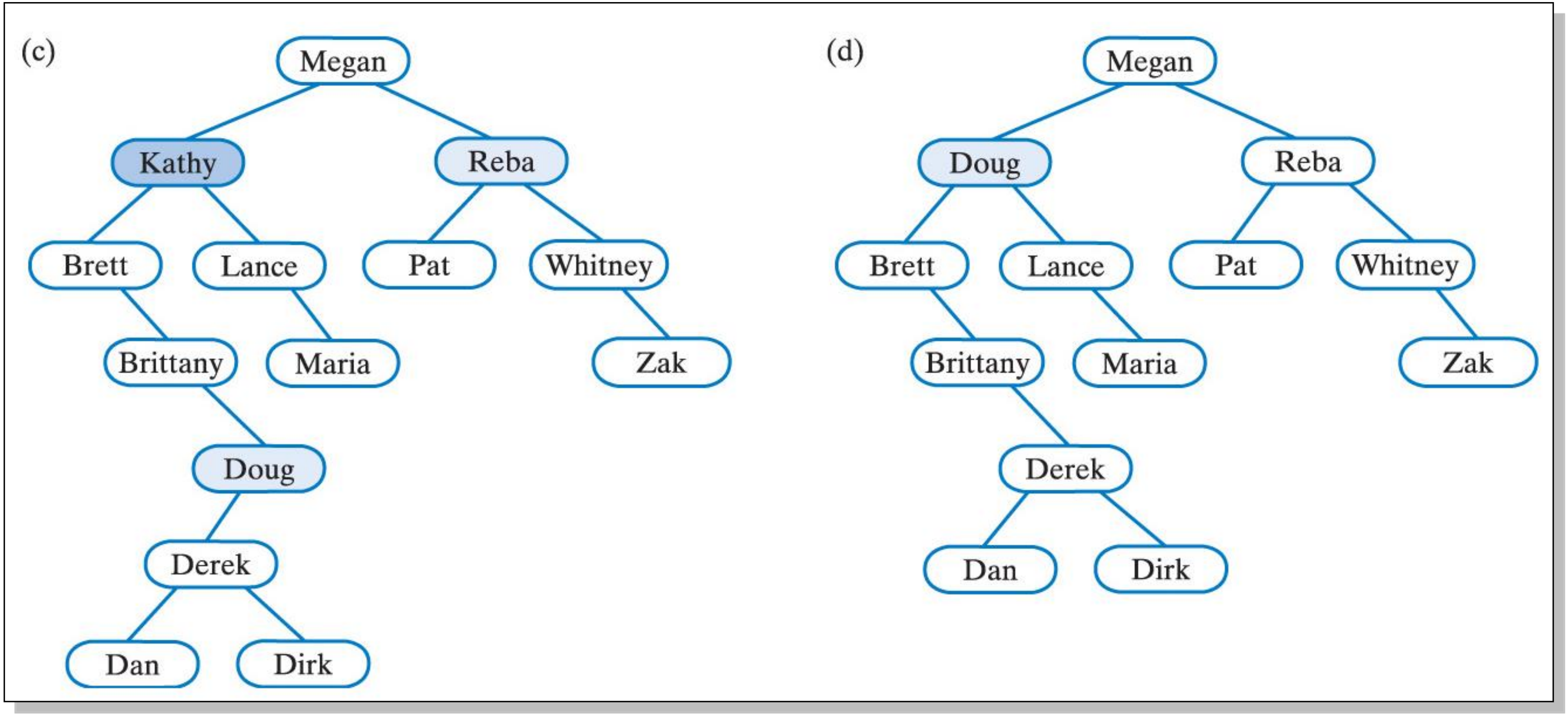
Deleting Michael falls into Case 2

Case2: Removing an Entry, Node Has Two Children



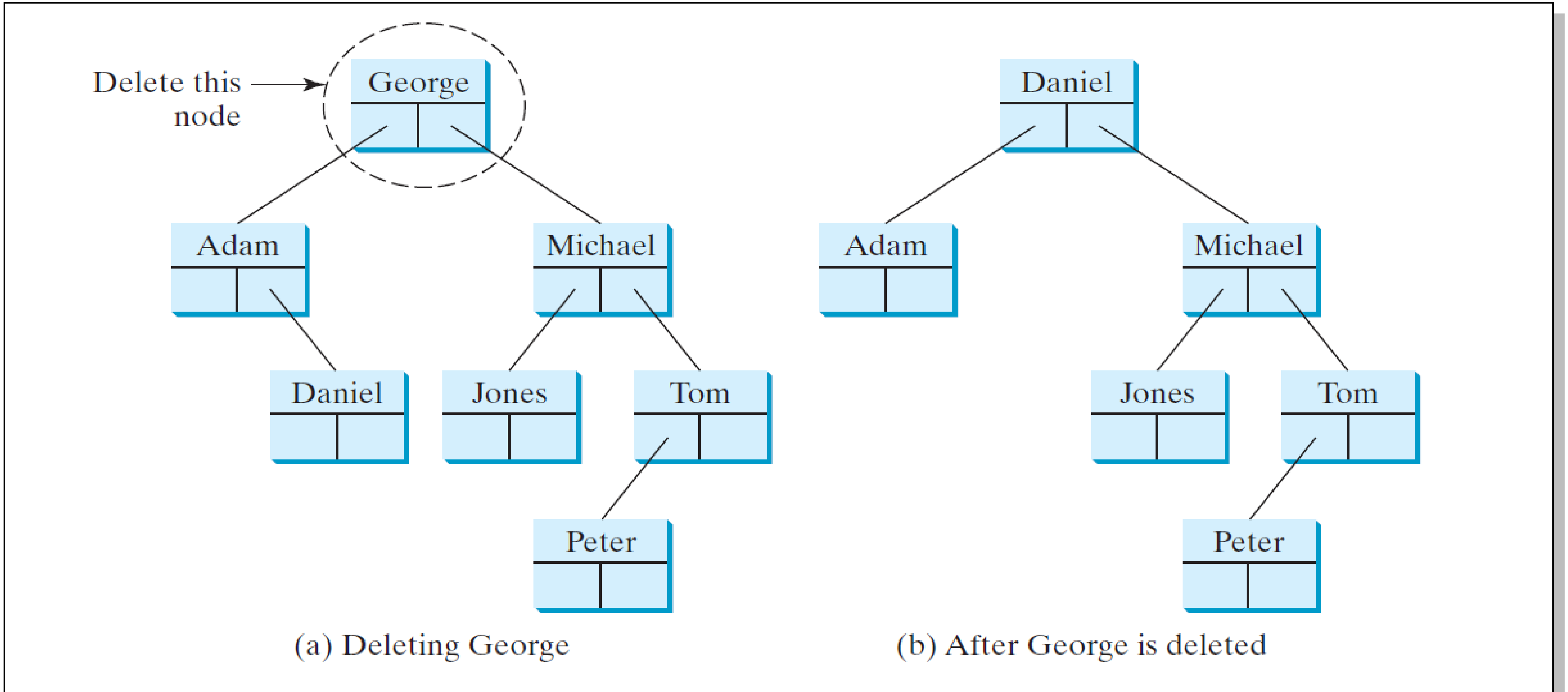
(a) A binary search tree; (b) after removing *Chad*;

Case2: Removing an Entry, Node Has Two Children



(c) after removing *Sean*; (d) after removing *Kathy*.

Case2: Removing an Entry, Node Has Two Children

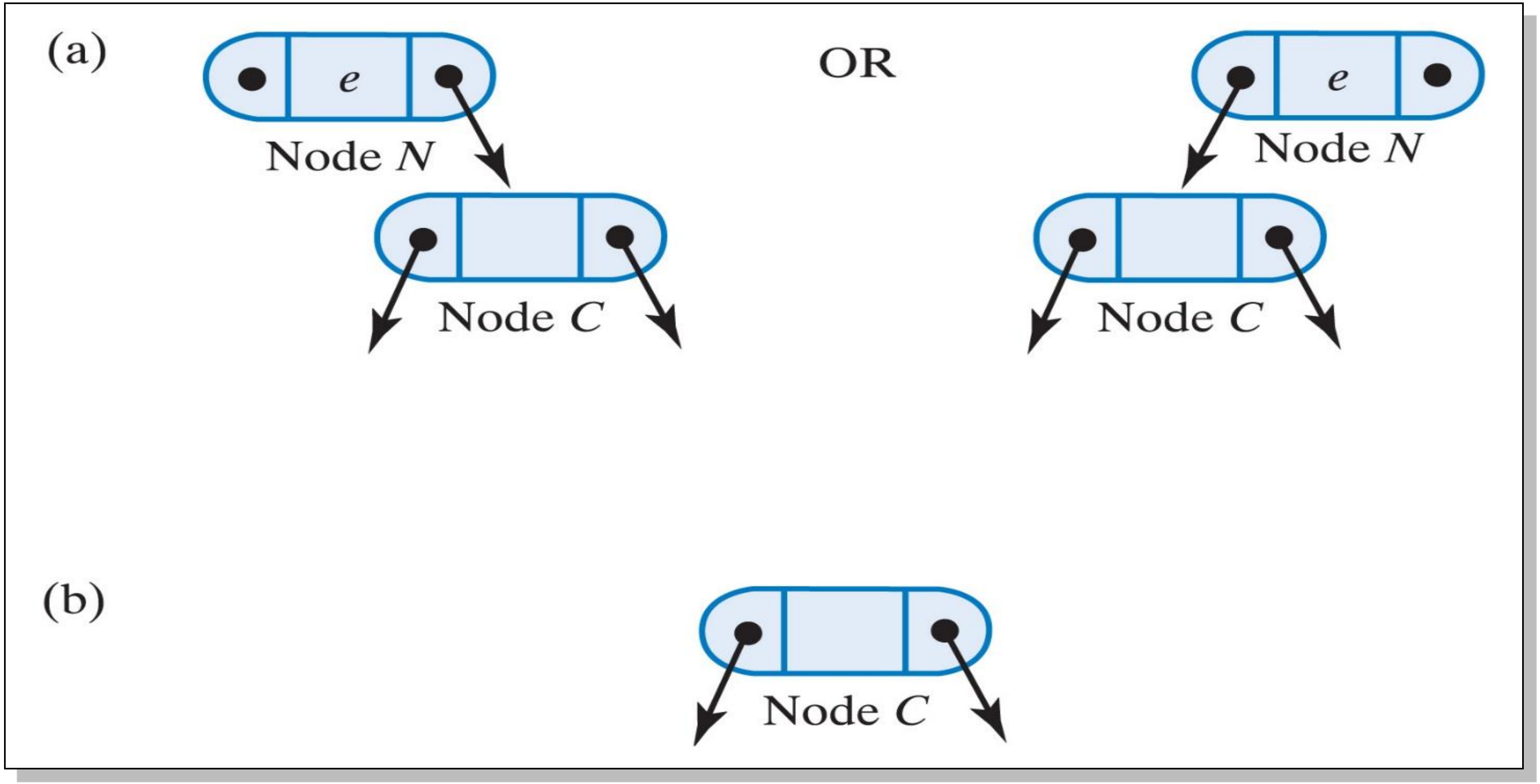


(a) Deleting George

(b) After George is deleted

Deleting George falls into Case 2

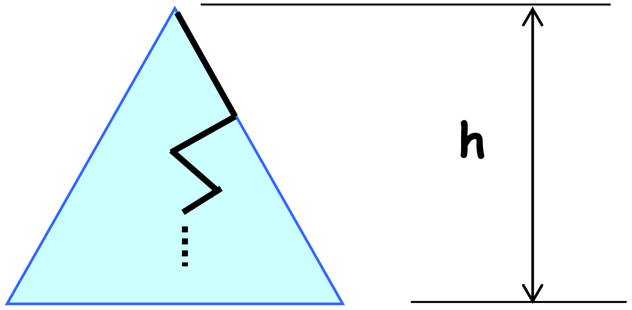
Removing an Entry in the Root



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

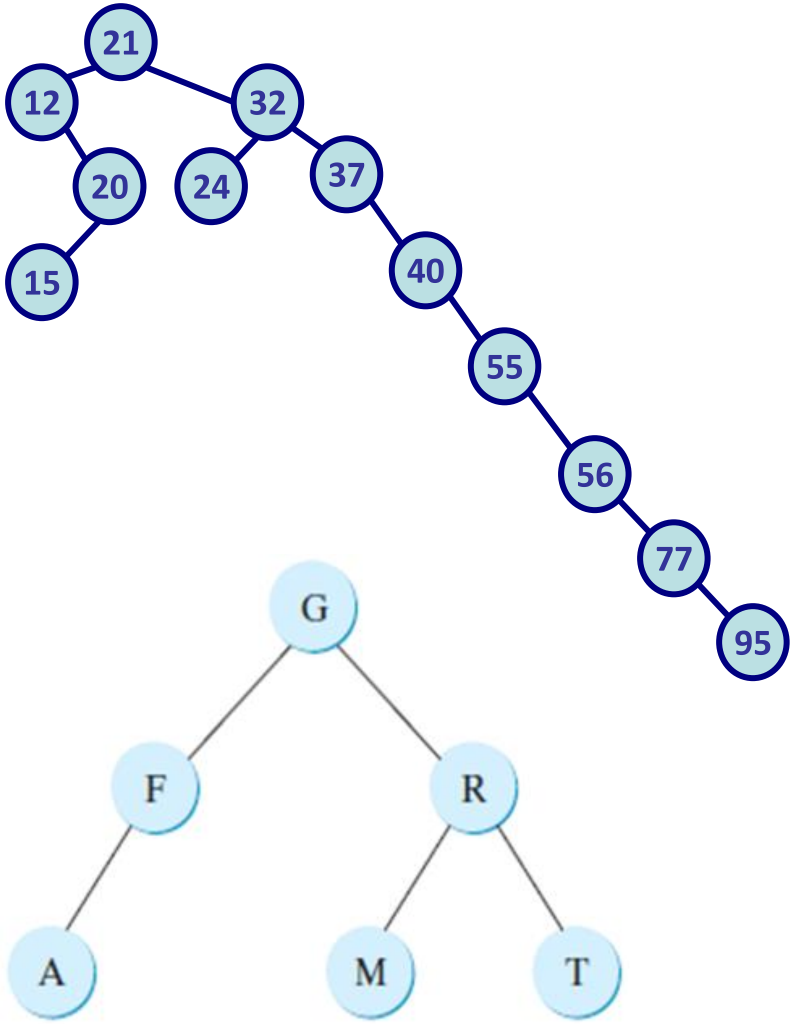
Efficiency of Operations

- It is obvious that the complexity for the inorder, preorder, and postorder is O(n), since each node is traversed only once.
- Operations **add**, **remove**, **getEntry** require a search that begins at the root
- Maximum number of comparisons is directly proportional to the **height,** ***h*** of the tree
- These operations are **O(*h*)**
- Thus we desire the shortest binary search tree we can create from the data



Efficiency of Operations

- Because the shape of a BST is determined by the order that data is inserted, we run the risk of trees that are essentially lists
- So, the worst case for a single BST operation can be **O(n)**, and for **m** operations can be **O(m*n)**
- On average, the height of the tree is **O(logn)**. So, the average time for search, insertion, deletion in a BST is **O(logn)**.
- In **balanced** BST single operation can be done in **O(log n)**, and for **m** operations, **O(m log n)**

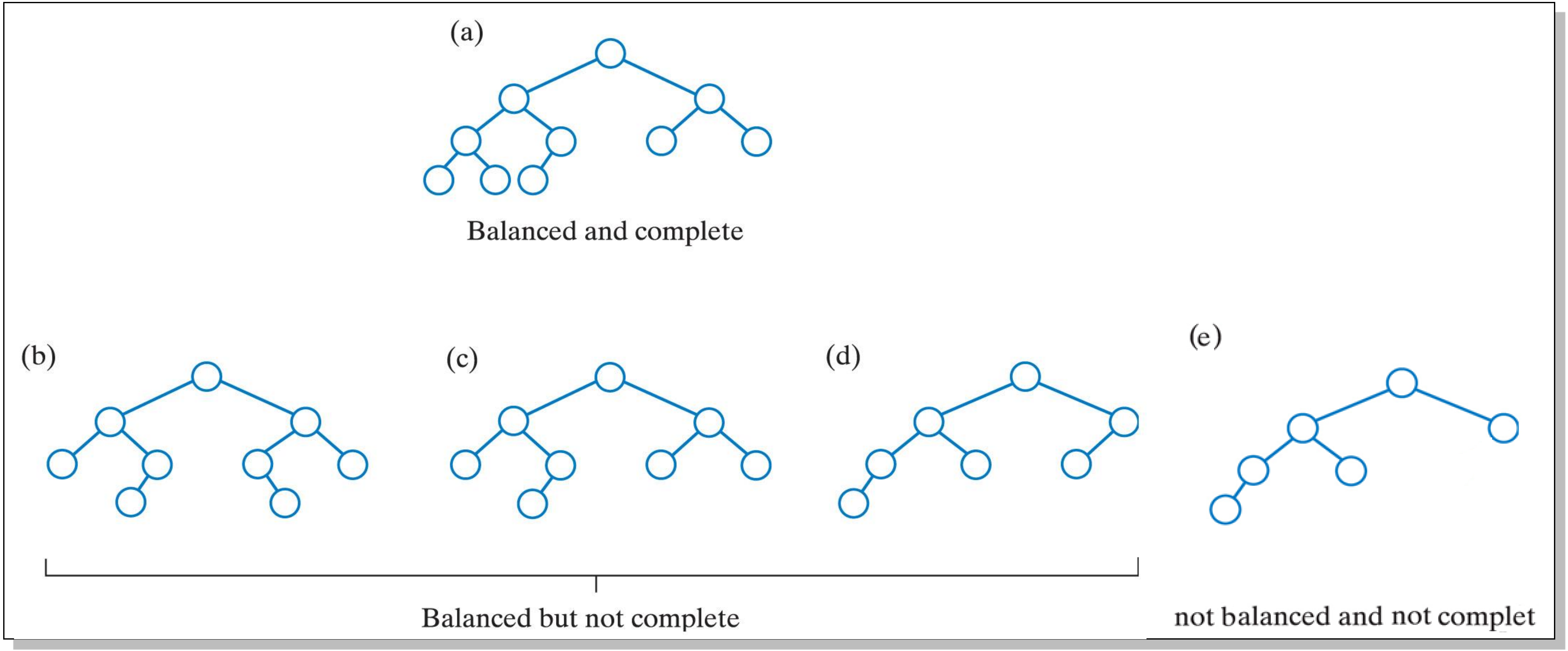


Importance of Balance

- Completely balanced
  - Subtrees of each node have exactly same height
- Height balanced
  - Subtrees of each node in the tree differ in height by no more than 1
- Completely balanced or height balanced trees are balanced

https://liveexample.pearsoncmg.com/dsanimation/BSTeBook.html

Importance of Balance



Some binary trees that are height balanced.

https://liveexample.pearsoncmg.com/dsanimation/BSTeBook.html

Assignment

- Given a Binary Tree, write Java functions to check whether the given Binary Tree is:
  a) a perfect tree ◀
  b) a full tree ◀
  c) a complete tree
  d) a balanced tree
  e) a degenerate tree

Questions?