BACS2063 Data Structures and Algorithms

BINARY TREES

Chapter 9

Learning Outcomes

At the end of this lecture, you should be able to

- Describe applications of binary trees such as expression trees and decision trees.
- Implement binary trees and binary search trees
- Discuss the factors that affect the efficiency of the binary search tree operations

Tree Concepts

- Previous ADTs place data in *linear order*.
- Some data organizations require categorizing data into groups, subgroups
  - This is hierarchical classification
  - Data items appear at various levels within the organization
- A tree provides a hierarchical organization in which data items have *ancestors* and descendants
  - This organization is richer and more varied than any other ADT

Hierarchical Organization

- Example: Family trees

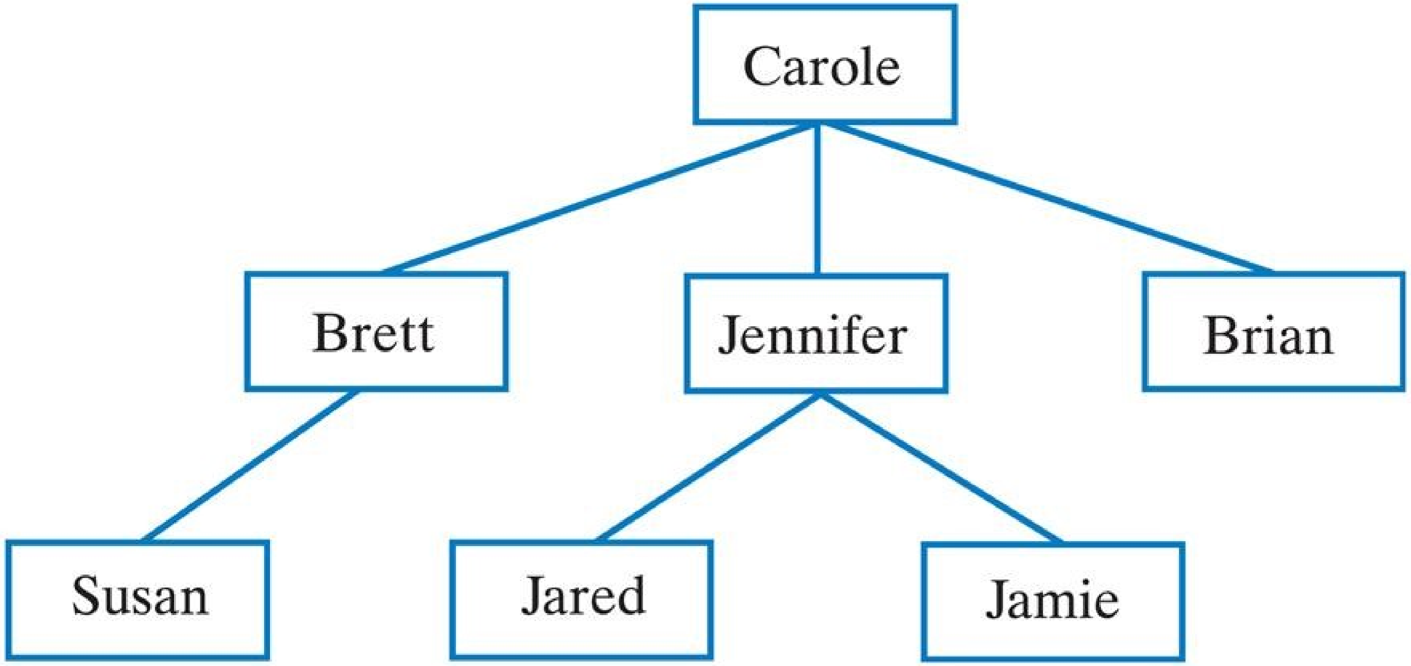


Fig. 25-1 Carole's children and grandchildren.

Hierarchical Organization

- Example: Organization Charts

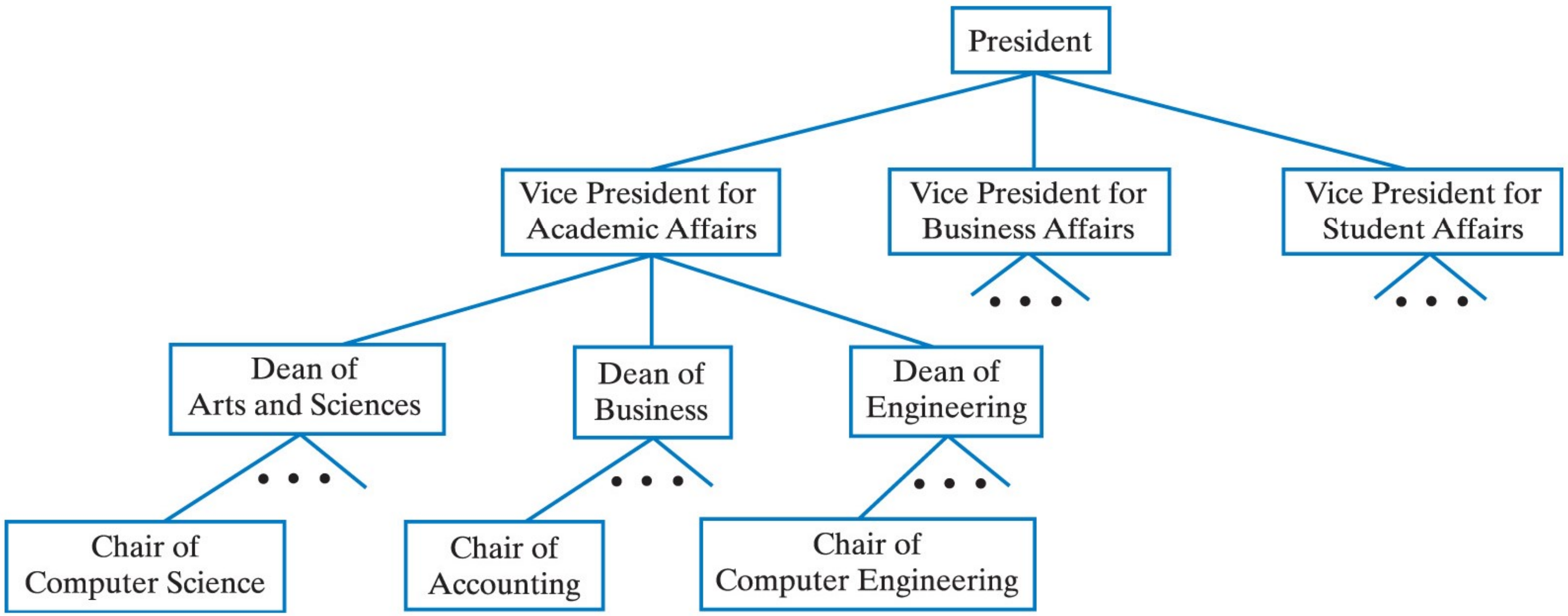


Fig. 25-3 A university's administrative structure.

Hierarchical Organization

- Example: File Directories

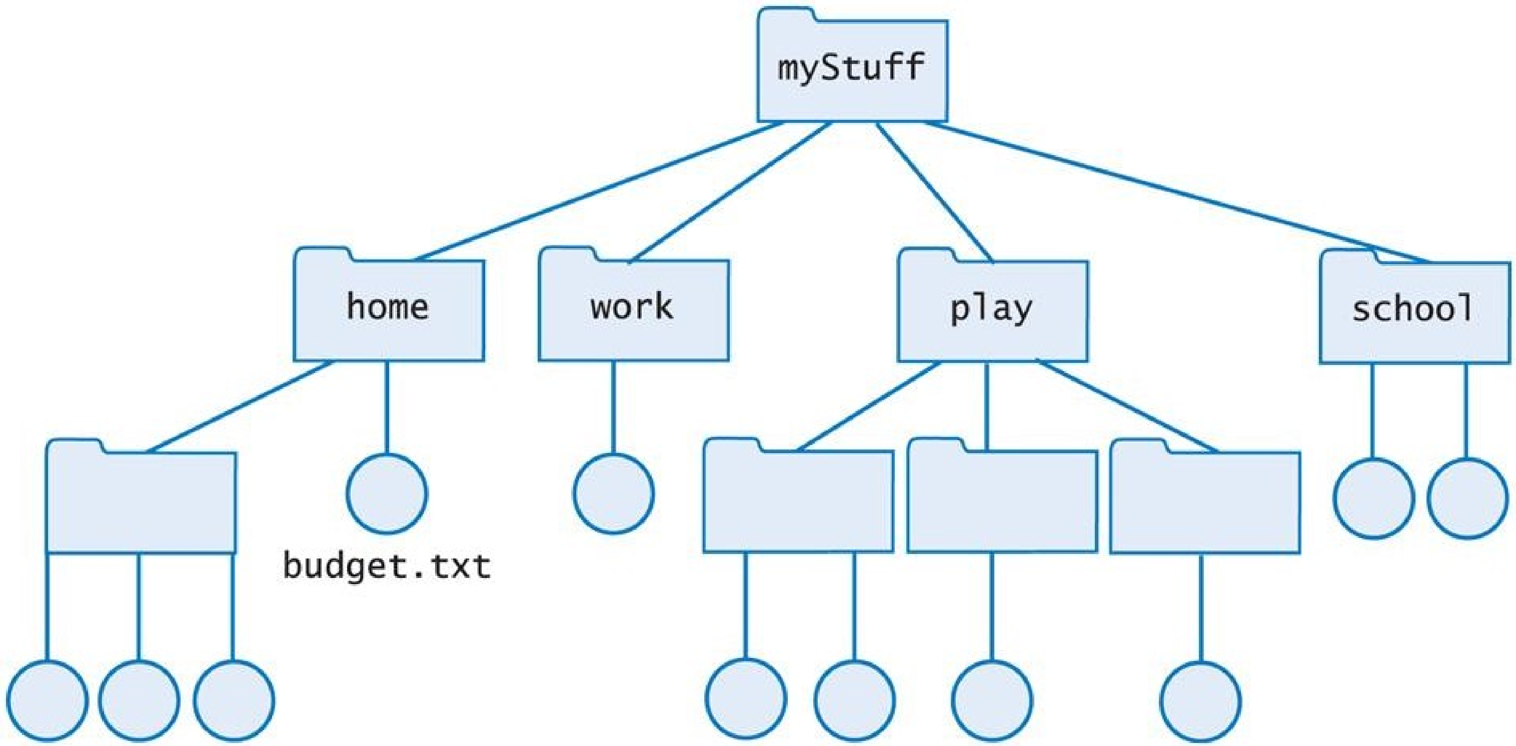


Fig. 25-4 Computer files organized into folders

Tree Terminology

- A tree is a *set of nodes connected by edges*.
- The edges indicate relationships among nodes.
- Nodes are arranged in *levels*.
  - Indicate the nodes' hierarchy
  - Top level is a single node called the *root*

Tree Terminology

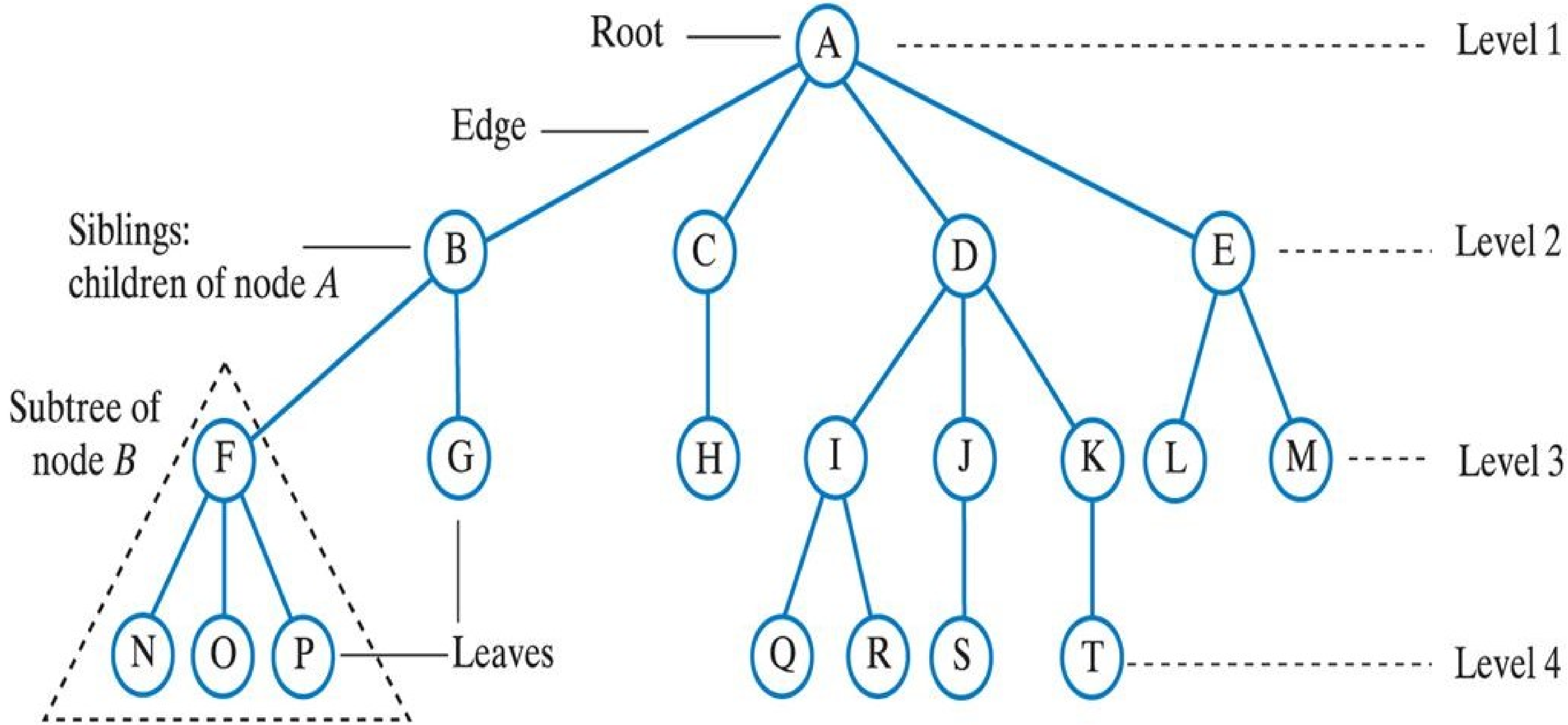


Fig. 25-5 A tree equivalent to the tree in Fig. 25-4

Tree Terminology

- Nodes at a given level are *children* of nodes of the previous level
- A node with children is the *parent* node of those children – node *A* is the parent of nodes *B*,*C*,*D* and *E*.
- Nodes with the same parent are *siblings* - nodes *B*,*C*,*D* and *E* have the same parent *A*.
- A node with no children is a *leaf node* – e.g. nodes *N*, *O* and *P*.
- The only node with no parent is the root node
  – All others have one parent each

Tree Terminology

- A node is reached from the root by a *path*.
  - The length of the path is the number of edges that compose it
- The *height* of a tree is the number of levels in the tree.
  - Height of tree T = 1 + height of the tallest subtree of T.
- The *subtree* of a node is a tree rooted at a child of that node.

Binary Trees

- A tree in which each node has *at most two children*.

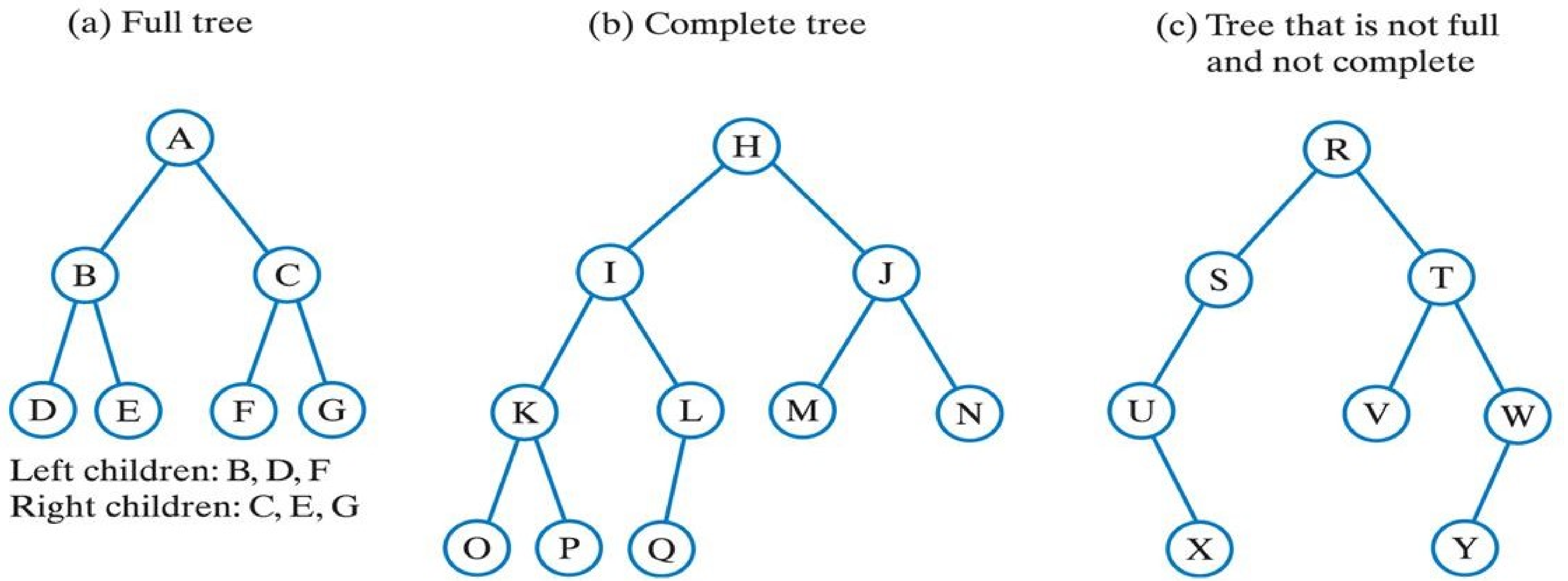
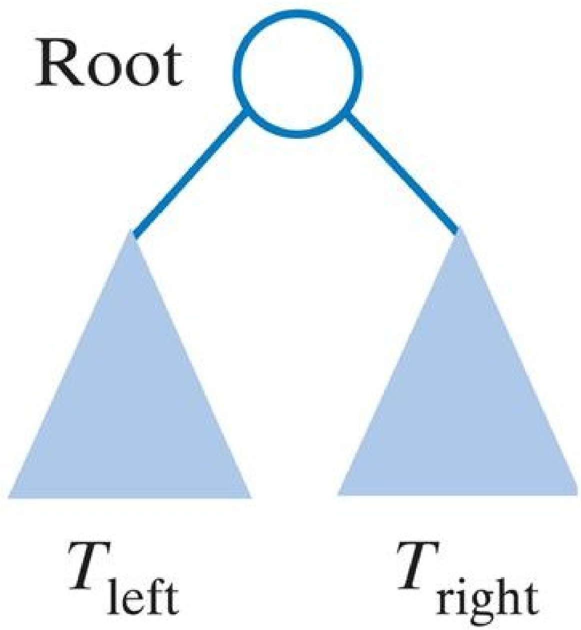


Fig. 25-6 Three binary trees.

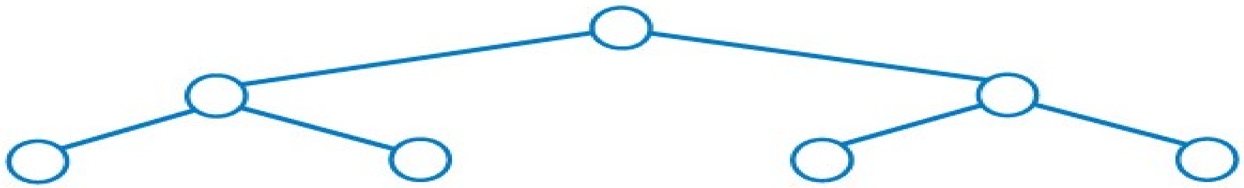
Binary Trees

- A binary tree is either empty or has the following form



  - Where T_{left} and T_{right} are binary trees

Height of Binary Trees

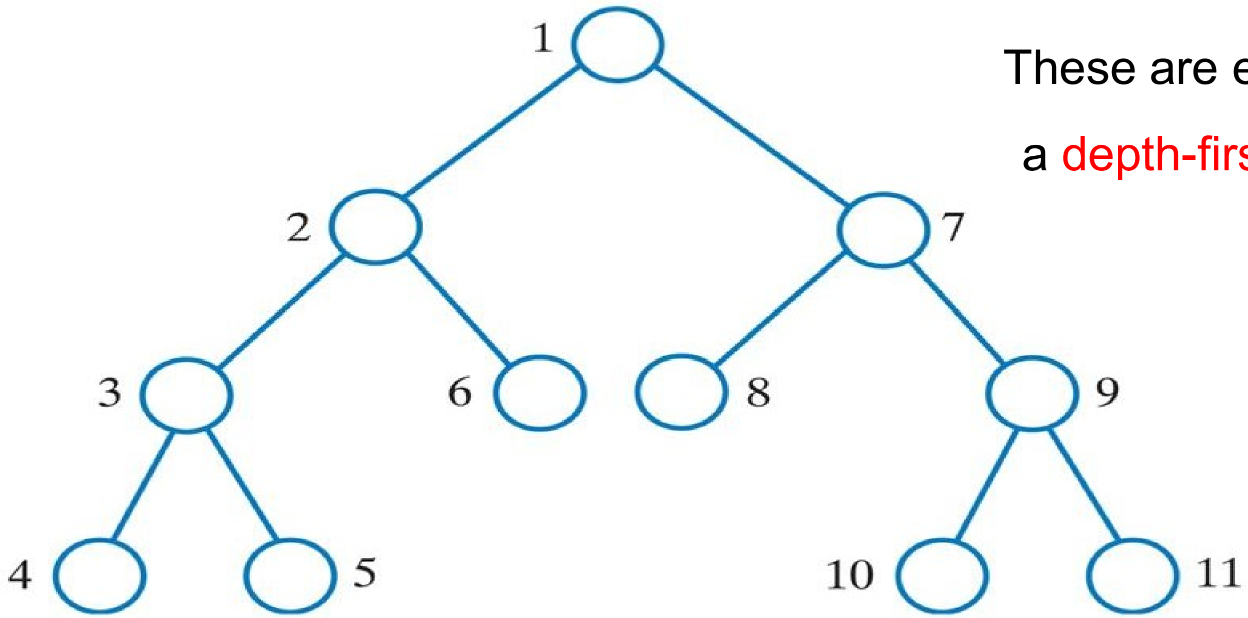
Binary Tree	Height
	1
	2
	3

Traversals of a Tree

- Tree traversal
  - The process of stepping through / visiting all the nodes in the tree.
  - Each node is visited / processed exactly once during a traversal.
- Visiting a node
  - Processing the data within a node
- A traversal can pass through a node without visiting it at that moment

Preorder Traversal

- Visit root before the subtrees.
  - Visit the root
  - Visit all the nodes in the root's left subtree
  - Visit all the nodes in the root's right subtree



These are examples of a depth-first traversal.

Fig. 25-8 The visitation order of a preorder traversal.

Inorder Traversal

- Visit root between visiting the subtrees.
  - Visit all the nodes in the root's left subtree
  - Visit the root
  - Visit all the nodes in the root's right subtree

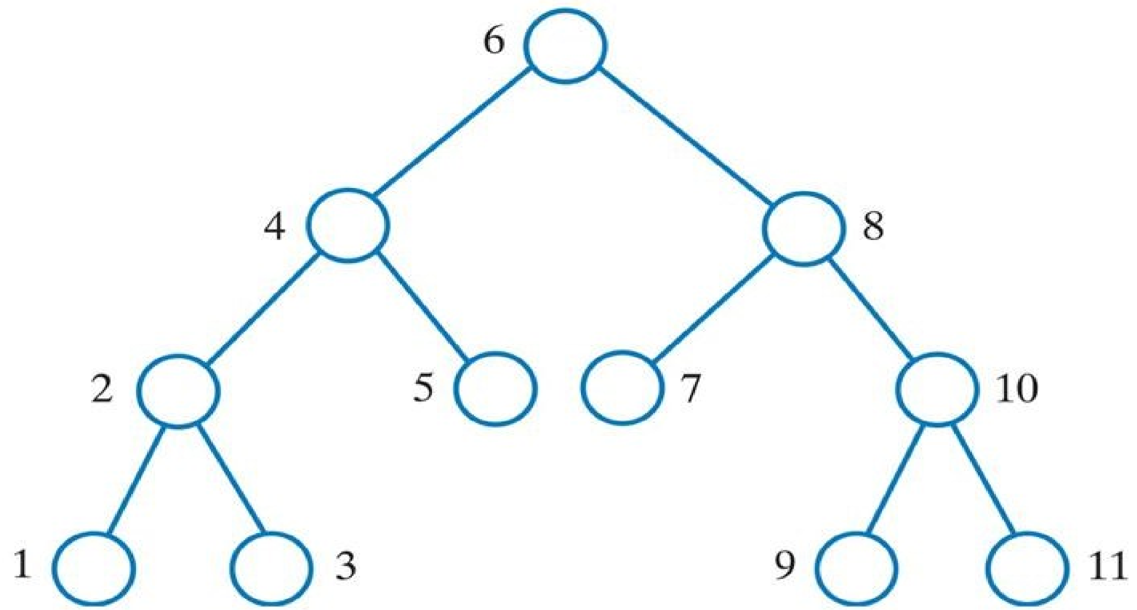


Fig. 25-9 The visitation order of an inorder traversal.

Postorder Traversal

- Visit root after visiting the subtrees.
  - Visit all the nodes in the root's left subtree
  - Visit all the nodes in the root's right subtree.
  - Visit the root

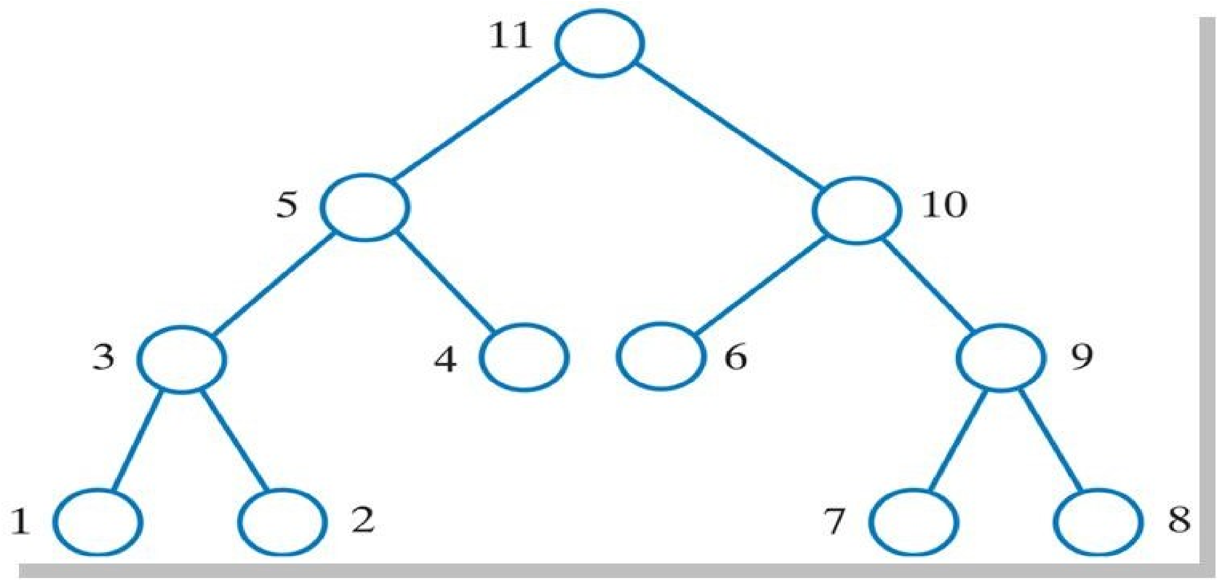


Fig. 25-10 The visitation order of a postorder traversal.

Level-order Traversal *[Optional]*

- Begin at the root, visit nodes one level at a time

This is an example of

a breadth-first traversal.

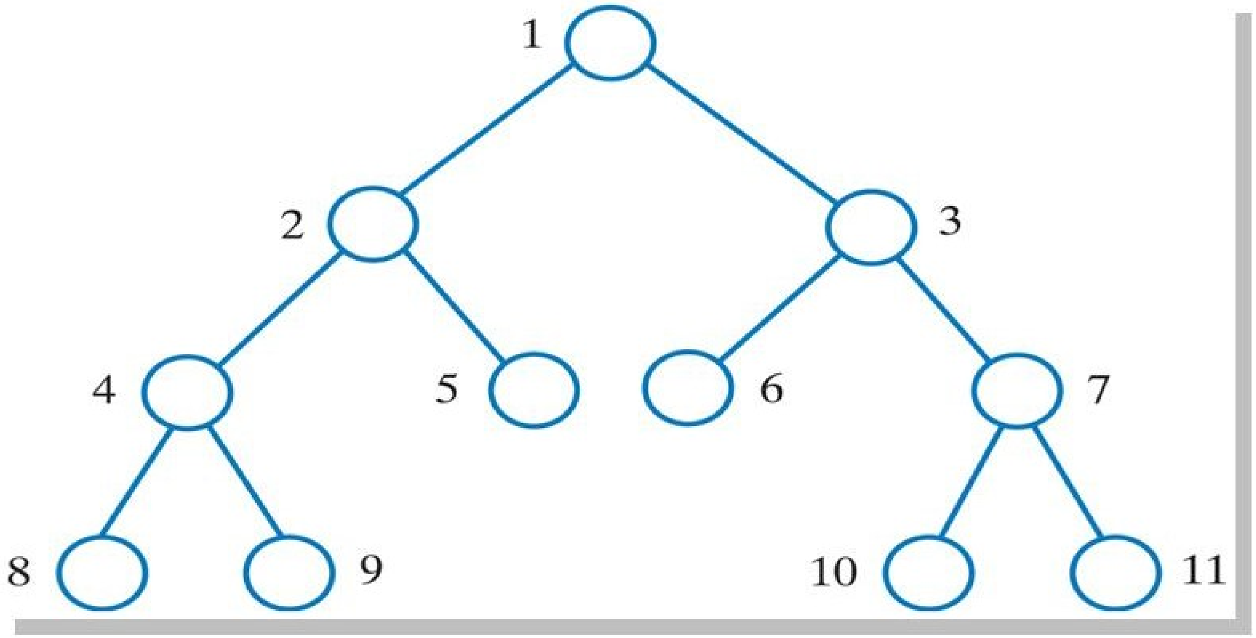


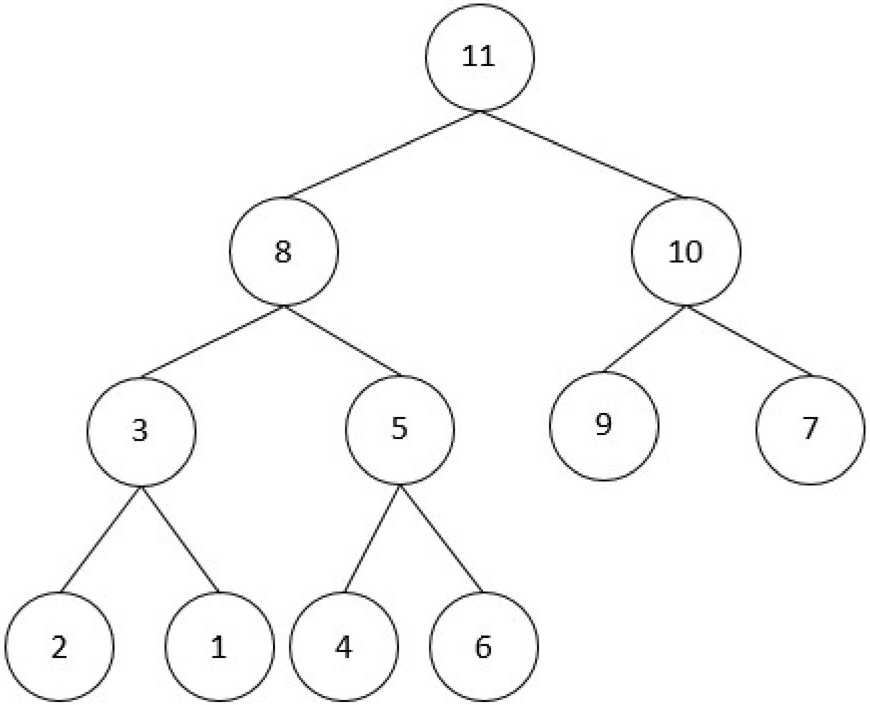
Fig. 25-11 The visitation order of a level-order traversal.

Exercise

- Reconstruct a unique *binary tree* for the following traversal outputs

  (i)Preorder: 11,8,3,2,1,5,4,6,10,9,7

  Inorder: 2,3,1,8,4,5,6,11,9,10,7

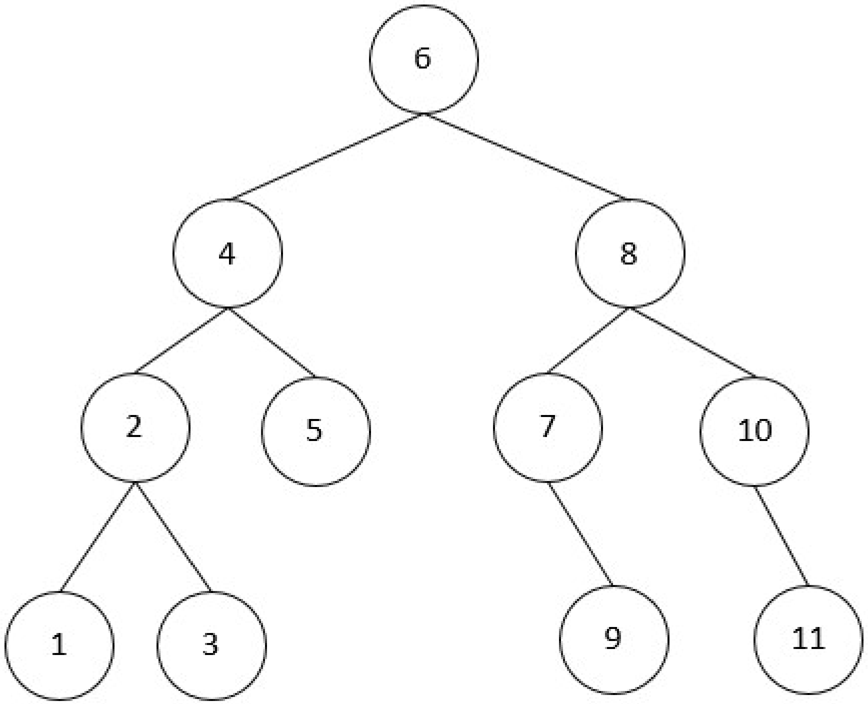


Exercise

- Reconstruct a unique *binary tree* for the following traversal outputs

  (i)Preorder: 6,4,2,1,3,5,8,7,9,10,11

  Inorder: 1,2,3,4,5,6,7,9,8,10,11



An Interface for Binary Trees

- **BinaryTreeInterface.java**

```
public interface BinaryTreeInterface<T>     {
  public T getRootData();
  public boolean isEmpty();
  public void clear();
  public void setTree(T rootData);
  public void setTree(T rootData,
  BinaryTreeInterface<T> leftTree,
  BinaryTreeInterface<T> rightTree);
}
```

Building a binary tree

Consider a client program to build the following binary tree:

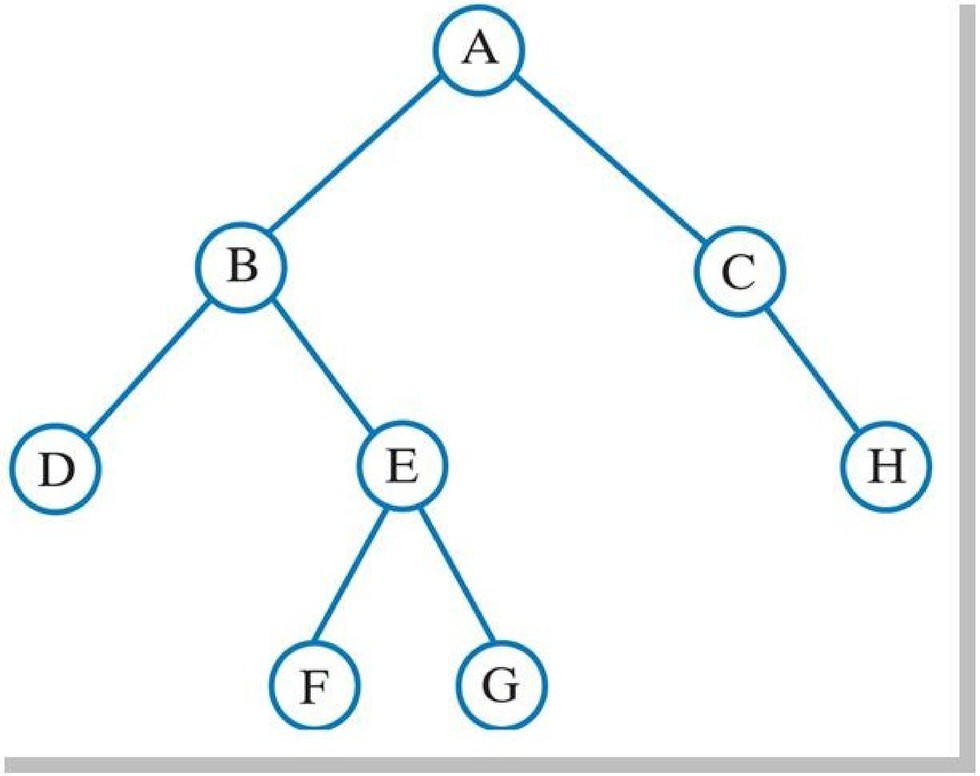


Fig. 25-13 A binary tree whose nodes contain one-letter strings

Building a binary tree

Given that the class **`BinaryTree`** implements the interface **`BinaryTreeInterface`**, we can build the given binary tree in Figure 25-13 (of previous slide) as follows:

- Represent each of its leaves (D, F, G, H) as a one-node tree each.
- Moving up the tree from its leaves,
  - Use method **`setTree()`** to form the next level of subtrees (i.e. subtree E, B, C)
  - Use method **`setTree()`** to form the final tree (A)

Sample Code

In `Chapter9\binarytree` folder:

- **`BinaryTreeInterface.java`**
- **`BinaryTree.java`**
- **`BinaryTreeDriver.java`**

Binary Tree Example: *Expression Trees*

An expression tree represents an algebraic expression whose operators are binary.

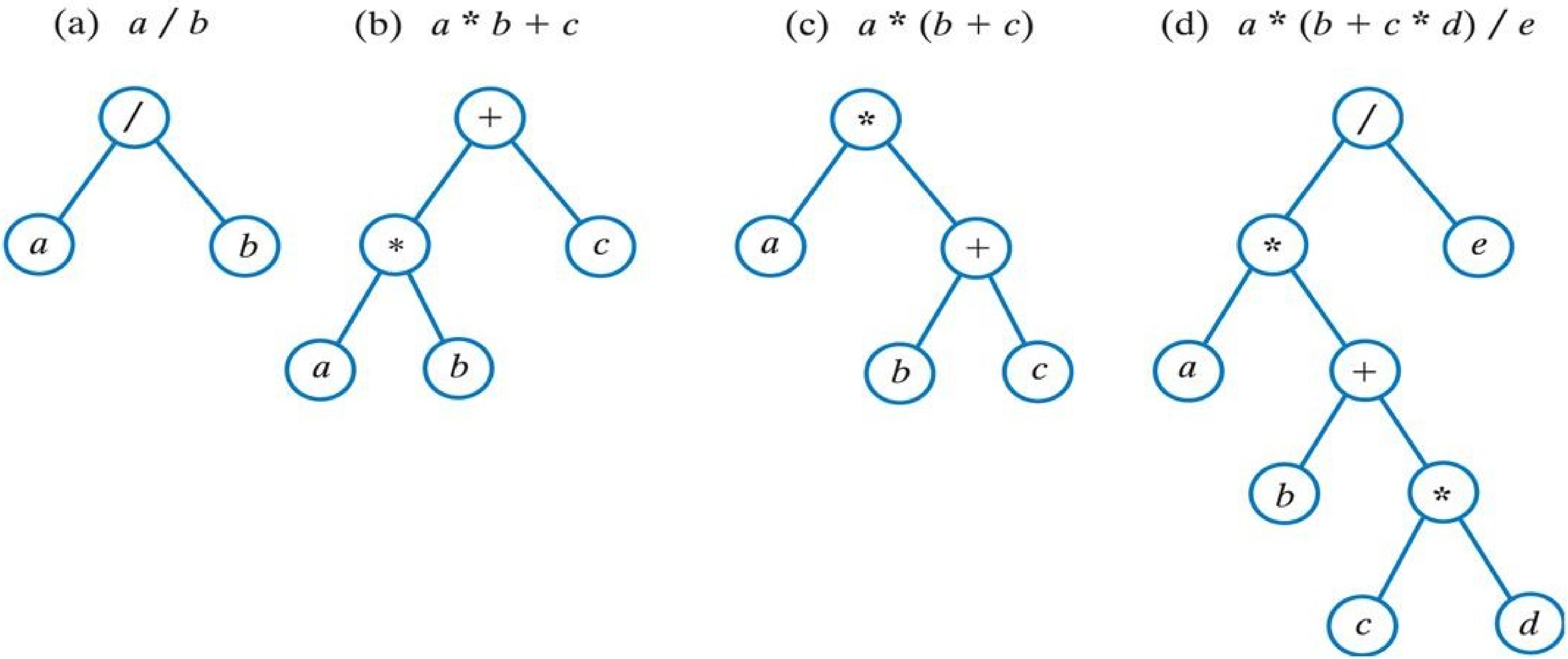


Fig. 25-14 Expression trees for four algebraic expressions.

Traversal of Expression Trees

- An inorder traversal of an expression tree visits variables and operators in the tree in the order in which they appear in the original infix expression.
- A preorder traversal produces the prefix form of the expression.
- A postorder traversal produces the postfix form of the expression.

Algorithm **`evaluate`**

```
Algorithm evaluate(expressionTree)
if (expressionTree is empty)
  return 0
else {
  firstOperand = evaluate(left subtree of
                              expressionTree)
  secondOperand = evaluate(right subtree of
                              expressionTree)
  operator = the root of expressionTree
  return the result of the operation operator
      and its operands firstOperand and
      secondOperand
}
```

Binary Tree Example: *Decision Trees*

A decision tree can be the basis of an expert system

- Helps users solve problems, make decisions

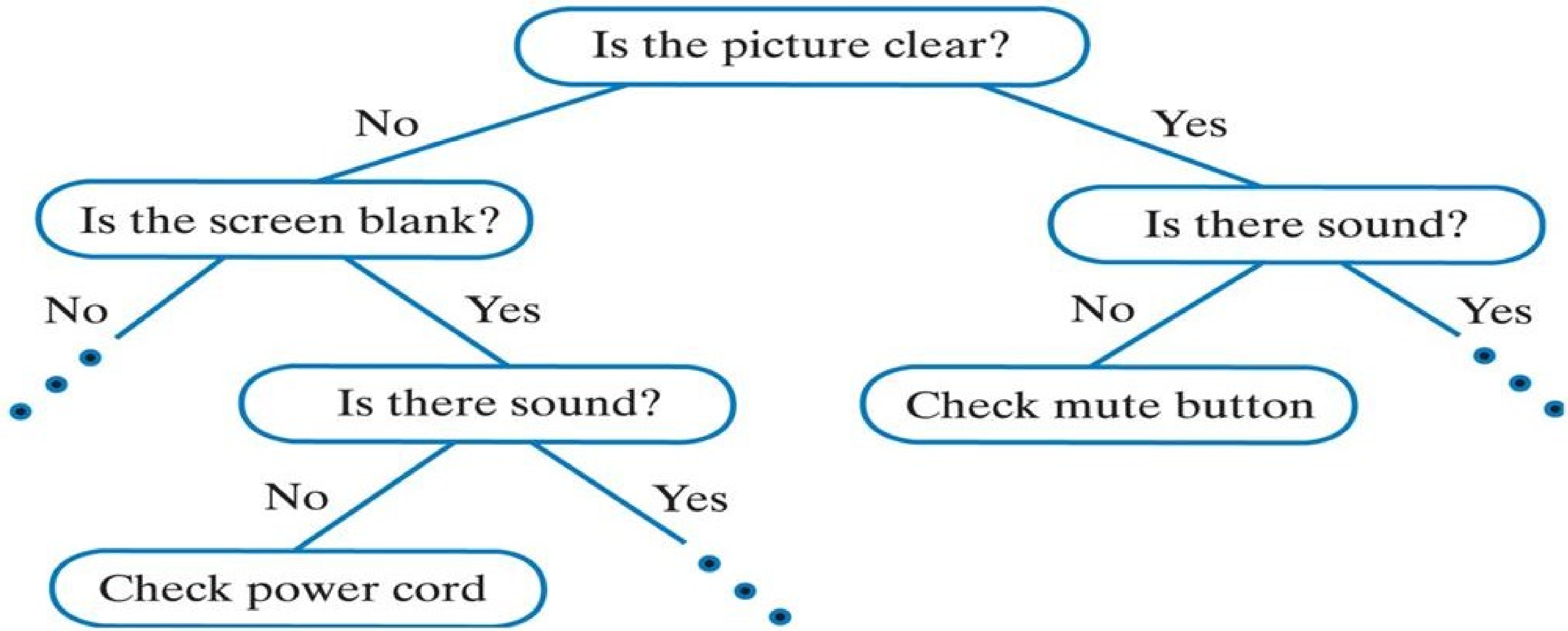


Fig. 25-15 A portion of a binary decision tree.

Decision Tree for a Guessing Game (1)

- Example: Guess-the-Country

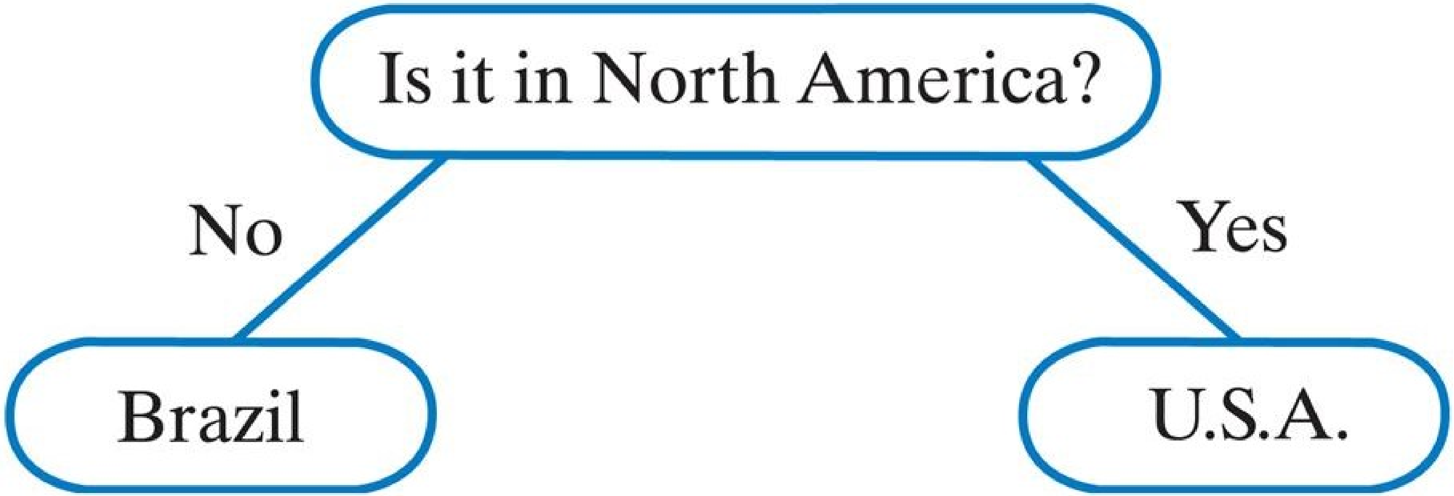


Fig. 25-16 An initial decision tree for a guessing game.

Decision Tree for a Guessing Game (2)

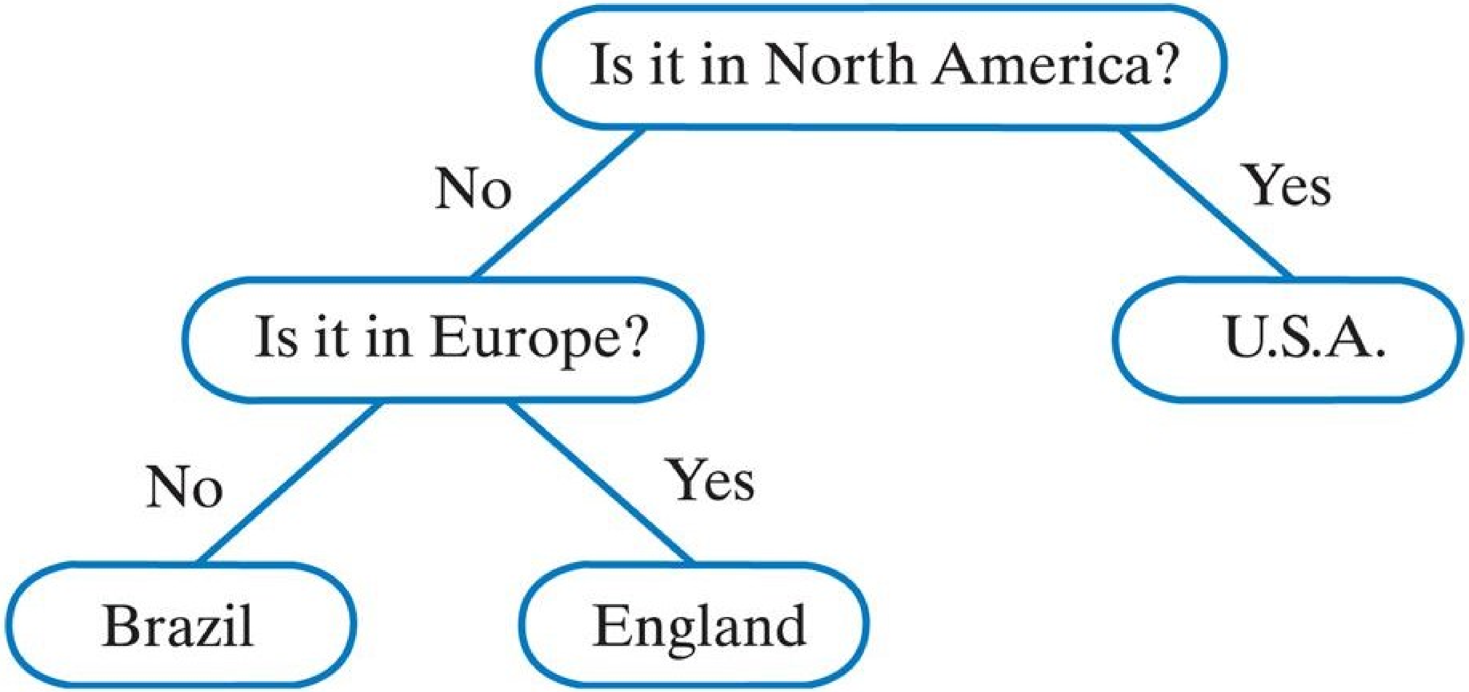


Fig. 25-17 The decision tree for a guessing game after acquiring another fact.

Traversing Recursively

- Postorder traversal
  - Public method for user, calls private method

```
public void postorderTraverse() {
  postorderTraverse(root);
}

private void
    postorderTraverse(BinaryNodeInterface<T> node){
   if (node != null) {
      postorderTraverse(node.getLeft());
      postorderTraverse(node.getRight());
      System.out.print(node.getData() + " ");
   }
}
```

Implementing Tree Traversals

- In the `BinaryTree` class, the method **`postorderTraverse()`** carried out a postorder traversal of the tree.
- However, this method *can only display* the data during traversal.
- To provide the client with more flexibility, we should define the traversals as *iterators*.
  - In this way, the client can do more than simply display data during a visit and we can control when each visit takes place.

Tree traversals using iterators

- Use an *iterator* that has the methods **`hasNext()`** and **`next()`**, as given in the interface **`java.util.Iterator`**.
- As illustrate with lists, we can define a method that returns such an iterator.
- Since we can have several kinds of tree traversals, a tree class could have several methods that each return a different kind of iterator.

Tree Implementation

- The most common implementation of a tree uses a linked structure.
- Nodes similar to those used in linked lists represent each element in the tree.

Nodes in a Binary Tree

- A node object that represents a node in a tree references both the data and the node's children.
- It contains a *reference to a data object* and *references to its left child and right child*, which are other nodes in the tree.
  - Either reference to a child could be **`null`**.
  - If both of them are **`null`**, the node is a leaf node.

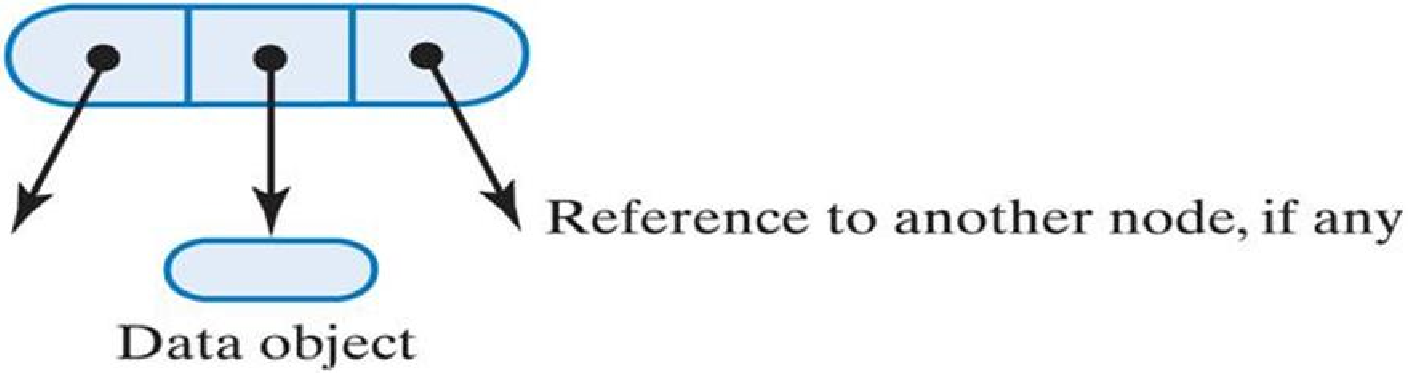


Fig. 26-1 A node in a binary tree.

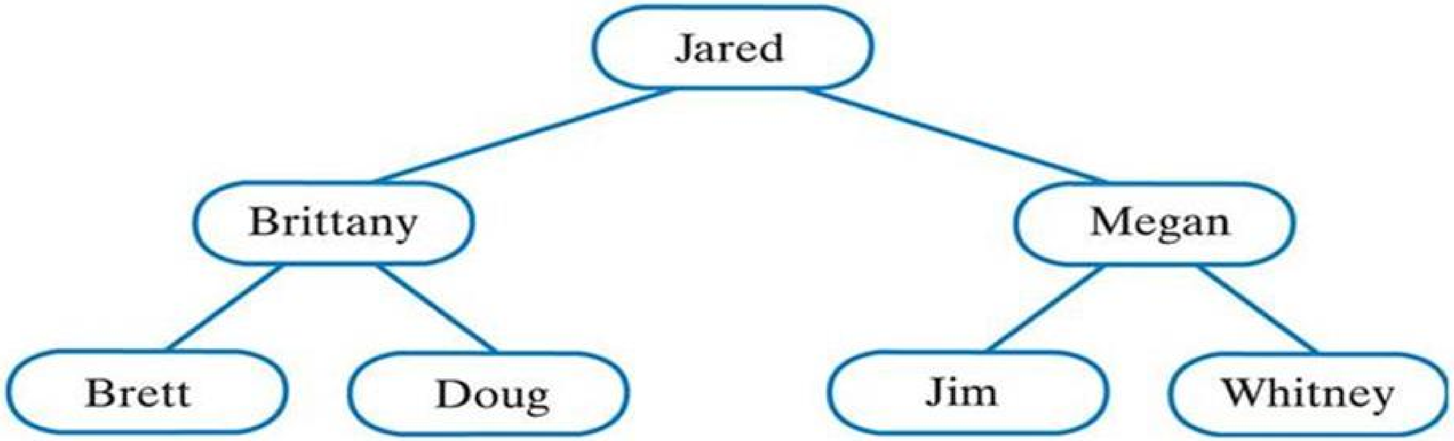
Node for Binary Trees

```
private class Node {
  private T data;
  private Node left;
  private Node right;
  . . .
}
```

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*

Binary Search Trees



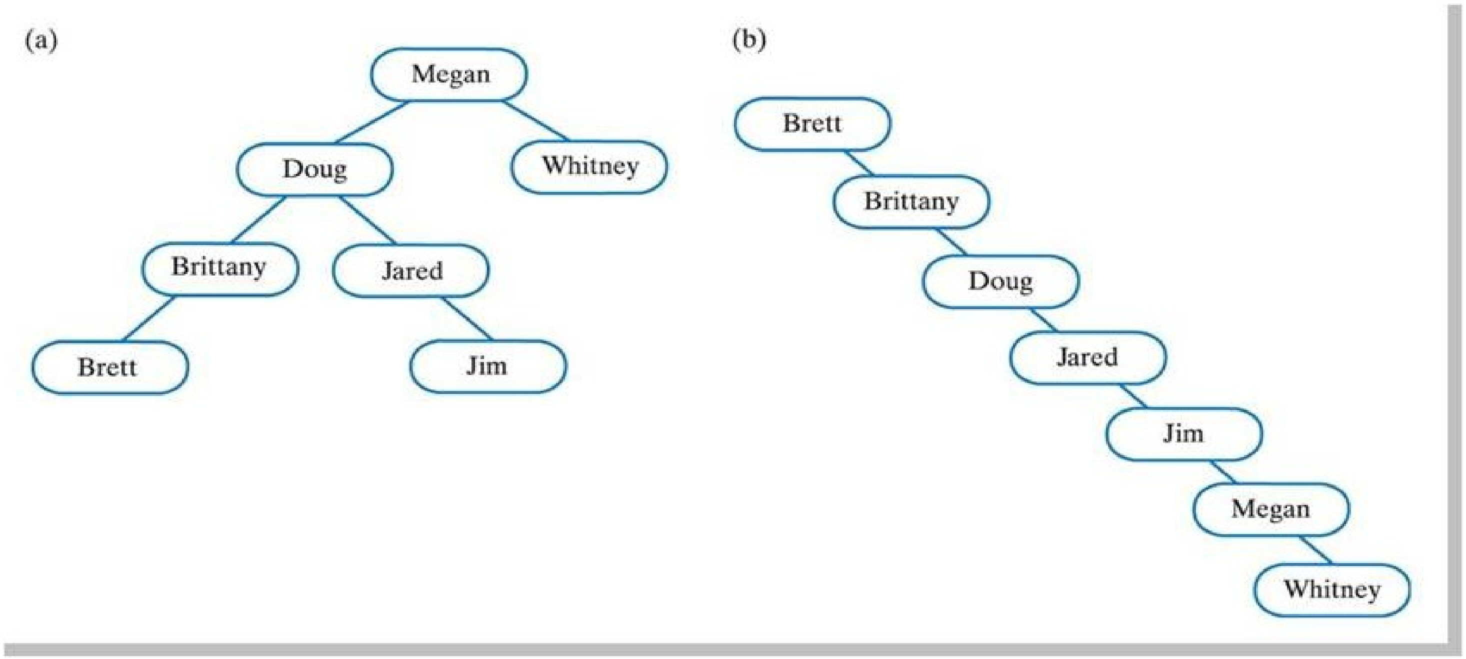


Fig. 25-18 A binary search tree of names.

Binary Search Trees

- The *efficiency of a search* depends on the *number of comparisons that a successful search requires*, i.e. the number of nodes along the path from the root to the node that contains the desired item.
- The height of a tree directly affects the length of the longest path from the root to a leaf. Hence, the height affects the efficiency of a worst-case search.
- Searching a binary search tree of height h is O(*h*).

Search Tree Characteristics

- A search tree stores data in a way that facilitates efficient searching.
- The nature of a binary search tree enables us to search it using a simple recursive algorithm

Searching in Search Trees

Like performing a binary search on a (sorted) array:

- For a sorted array
  - Search one of *two halves* of the array
- For the binary search tree
  - Search one of two subtrees of the binary search tree

Implementation of Tree Search

- Due to the recursive nature of the binary search tree structure, it is easier to implement its operations using *recursion*.
- *Note:*
  - *The* **`add`**, **`contains`**, **`getEntry`** *and* **`remove`** *operations need to search through the binary search tree in their implementation.*

Recursive 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)

Sample Code

In `Chapter9\binarytree` folder:

- `BinarySearchTreeInterface.java`
- `BinarySearchTree.java`
- `BinarySearchTreeDriver.java`
- `QueueInterface.java`
- `ArrayQueue.java`

Binary Search Tree Implementation

Source code:

- **`BinarySearchTreeInterface.java`**
  - Operations: **`contains`**, **`getEntry`**, **`add`**, **`remove`**, **`isEmpty`**, **`clear`**, **`getPreorderIterator`**, **`getInorderIterator`**, **`getPostorderIterator`**
- **`BinarySearchTree.java`**
  - Implements the interface **`BinarySearchTreeInterface`**
  - Contains private inner classes:
    - **`Node`** - to represent a binary search tree node
    - **`ReturnObject`** – required for the private **`removeEntry`** method to return the removed node's data object

Iterators for Tree Traversals

- Provides the creation of iterator objects to perform tree traversals.

```
import java.util.Iterator;
  …
  public Iterator<T> getPreorderIterator();
  public Iterator<T> getPostorderIterator();
  public Iterator<T> getInorderIterator();
```

getInorderIterator() method

```
public Iterator<T> getInorderIterator() {
  return new InorderIterator();
}
```

Inner class **InorderIterator**

- Define the class **InorderIterator** as a *private inner class* of **BinarySearchTree**.
- Has a queue object as a data field.

```
private class InorderIterator implements Iterator<T> {
   private QueueInterface<T> queue = new ArrayQueue<T>();

   public InorderIterator() {
     inorder(root);
   }

   private void inorder(BinaryNodeInterface<T> treeNode) {
       . . .
   }
   . . .
}
```

Inner class InorderIterator

- The constructor invokes method **inorder()**.
- In method **inorder()**, the “visit” results in the current node’s data being enqueued into the queue.

```
public InorderIterator() {
   inorder(root);
}

private void inorder(BinaryNodeInterface<T> treeNode) {
   if (treeNode != null) {
     inorder(treeNode.getLeft());
     queue.enqueue(treeNode.getData());
     inorder(treeNode.getRight());
   }
}
```

Method `next()`

- Removes and returns the next entry in the queue.

```
public T next() {
  if (!queue.isEmpty())
      return queue.dequeue();
  else
      throw new NoSuchElementException();
}
```

Methods `getEntry` and `findEntry`

- Method `getEntry` returns the located data object.
- Usual strategy for methods with recursive algorithms is applied throughout the class **`BinarySearchTree`**, i.e.:
  - Implement the actual recursive search as a private method `findEntry` that the public method `getEntry` invokes.

The method **getEntry**

```
public T getEntry(T entry) {
  return findEntry(root, entry);
}

private T findEntry(BinaryNode rootNode, T entry) {
  T result = null;
  if (rootNode != null) {
    T rootEntry = rootNode.data;

    if (entry.equals(rootEntry))
      result = rootEntry;
    else if (entry.compareTo(rootEntry) < 0)
      result = findEntry(rootNode.left, entry);
    else
      result = findEntry(rootNode.right, entry);
  }
  return result;
}
```

The method `contains`

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

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

Adding an Entry

- Every addition to a binary search tree adds a new leaf to the tree.

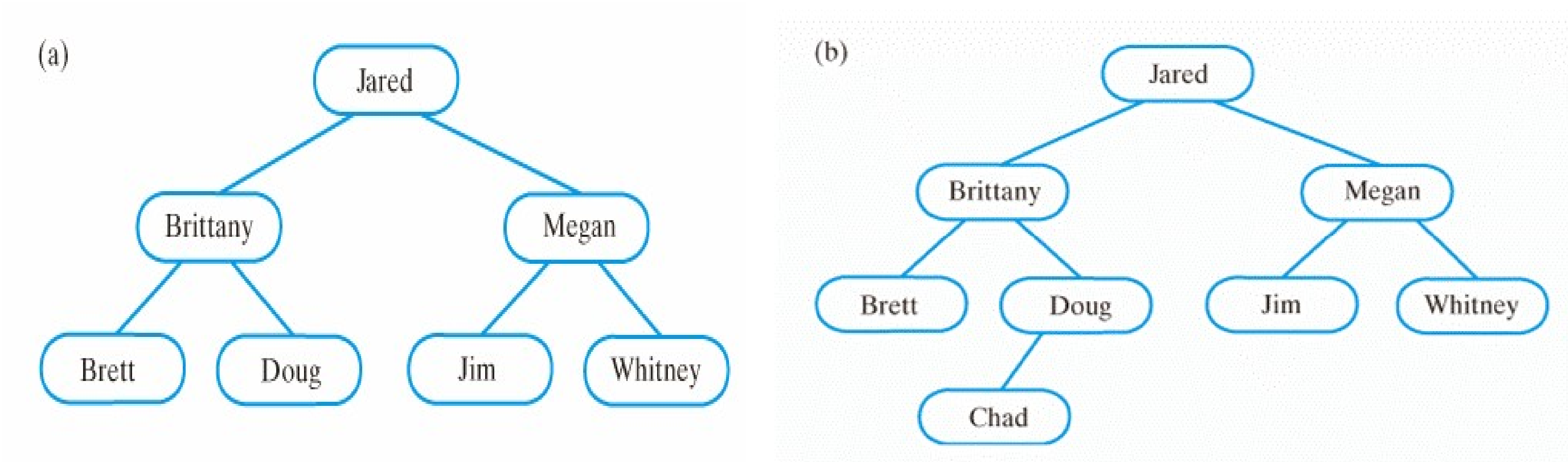


Fig. 27-4 (a) A binary search tree; (b) the same tree after adding *Chad*.

Adding an Entry Recursively

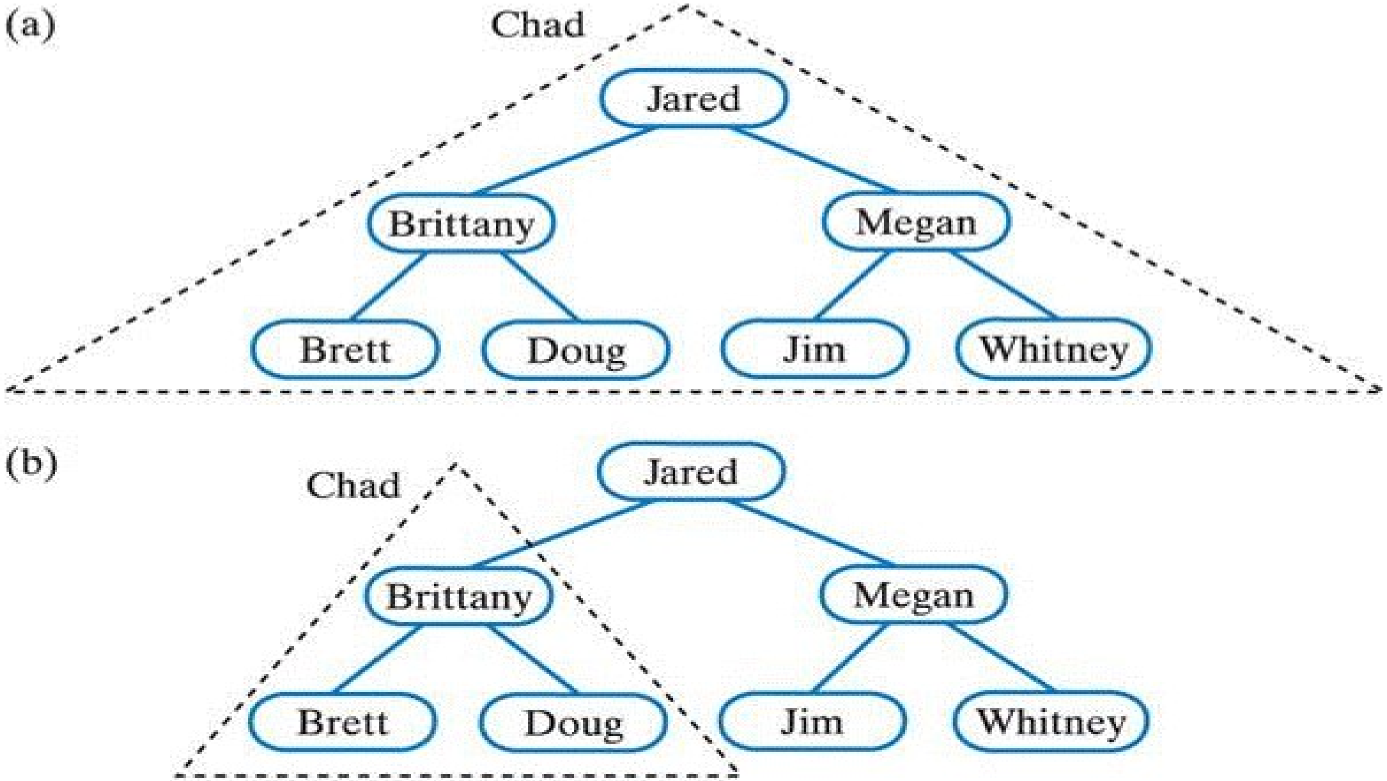


Fig. 27-5 Recursively adding *Chad* to smaller subtrees of a binary search tree … continued →

Adding an Entry Recursively

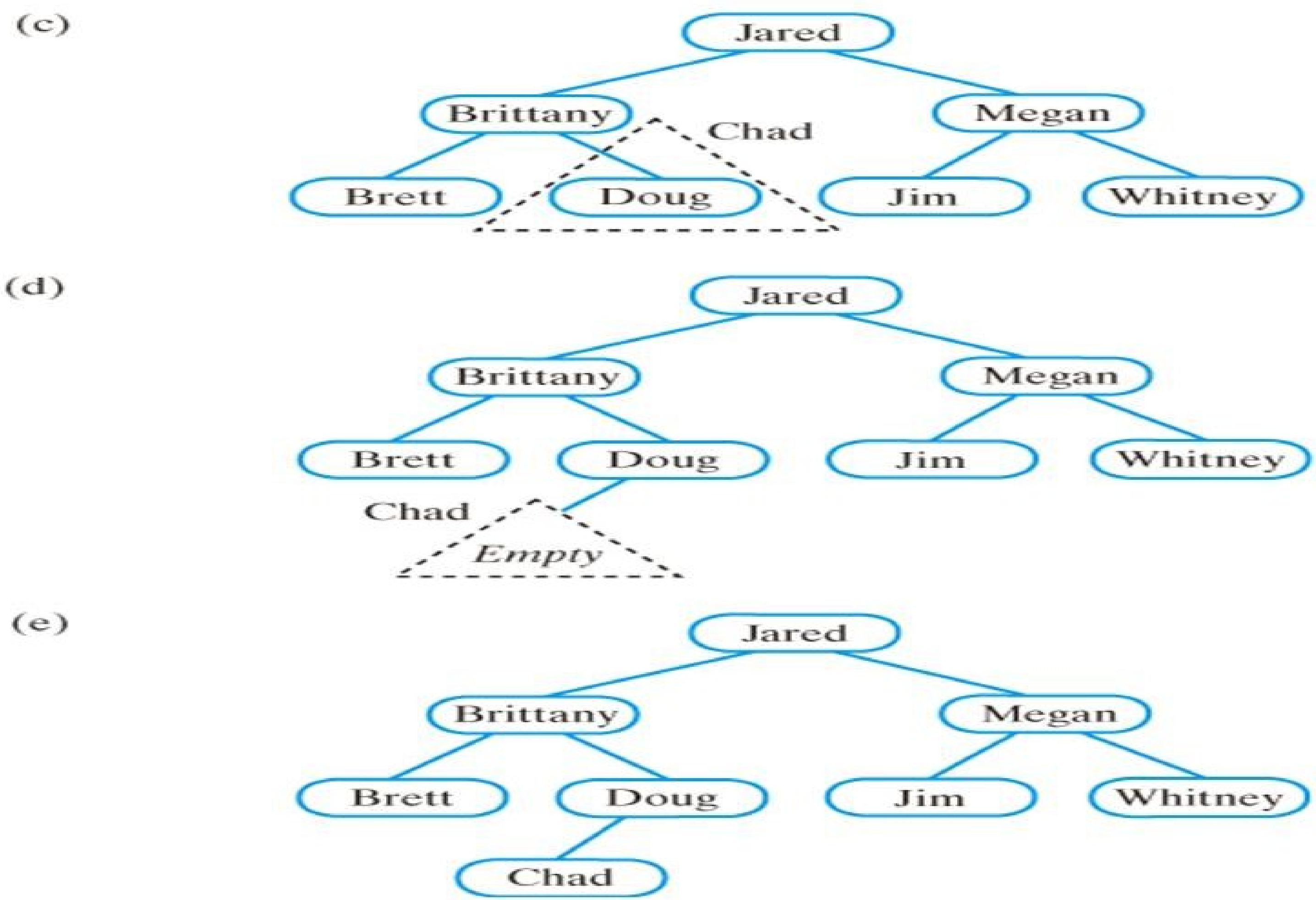


Fig. 27-5 (ctd.) Recursively adding *Chad* to smaller subtrees of a binary search tree.

Methods `add` and `addEntry`

- The public method **`add`** calls the private recursive method **`addEntry`**, if the tree is not empty.
- Like **`findEntry`**, **`addEntry`** has a node as a parameter that is initially the root node of the tree.
- When **`addEntry`** is called recursively, this parameter is either the left child or the right child of the current root.

Methods add and addEntry

```
public T add(T newEntry) {
 T result = null;
  if (isEmpty())
     root =new BinaryNode(newEntry);
  else
     result = addEntry(root, newEntry);
  return result;
 }
```

```
private T addEntry(BinaryNode rootNode, T newEntry) {
 T result = null;
  int comparison = newEntry.compareTo(rootNode.data);
  if (comparison == 0) {
    result = rootNode.data;
    rootNode.data = newEntry;
   }
  else if (comparison < 0) {
      if (rootNode.left != null)
        result = addEntry(rootNode.left, newEntry);
      else
        rootNode.left = new BinaryNode(newEntry);
   }
   else {
      if (rootNode.right != null)
        result = addEntry(rootNode.right, newEntry);
      else
        rootNode.right = new BinaryNode(newEntry);
   }
      return result;
 }
```

Duplicate Entries

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.
  - If duplicates are allowed, place the duplicate in the entry's right subtree.

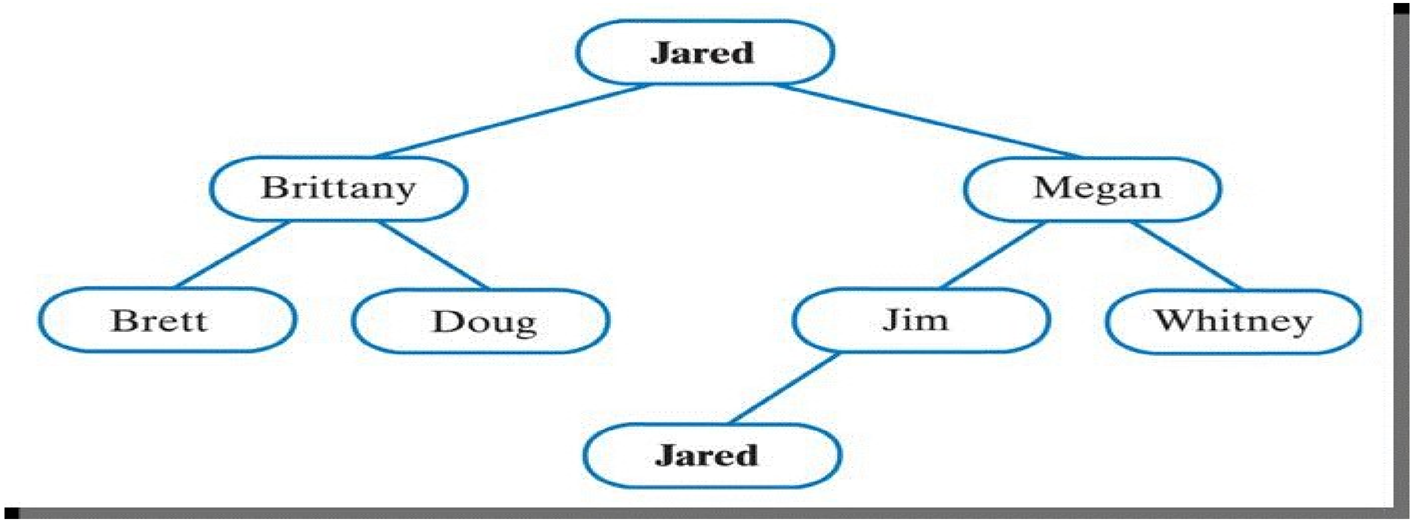
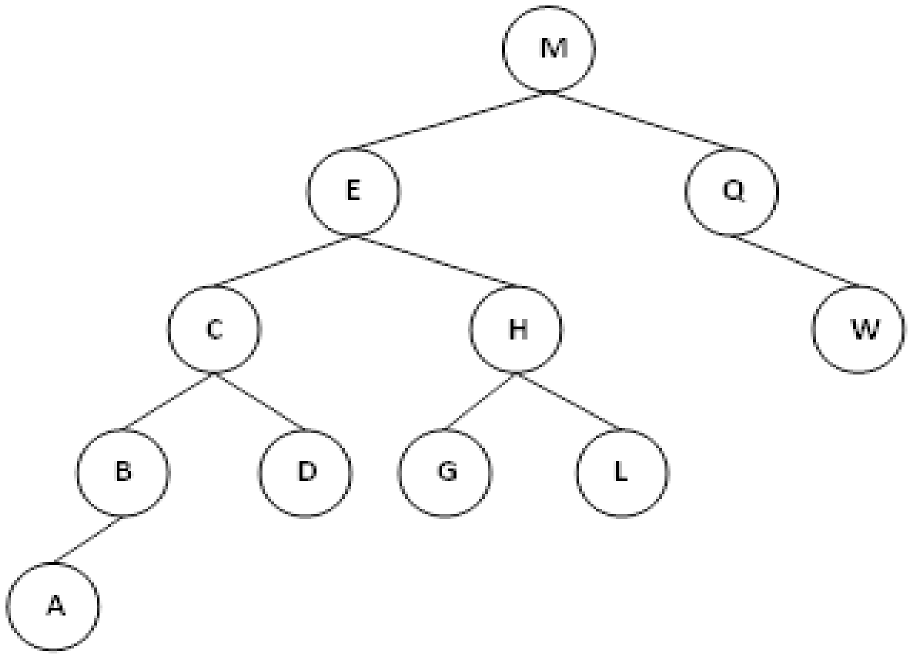


Fig. 27-3 A binary search tree with duplicate entries.

Exercise

- Construct a binary search tree using the following sequence of values: **M, E, Q, W, C, H, B, D, G, L, A**
- Identify the values for **preorder traversal, inorder & postorder traversal**



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`**
- 3 possible cases:
  - The node has no children, it is a leaf (simplest case)
  - The node has one child
  - The node has two children

1. Removing a Leaf Node

If the node is a left child of its parent, then the left reference of its parent is set to **`null`**. Otherwise, if the node is a right child of its parent, then the right reference of its parent is set to **`null`**.

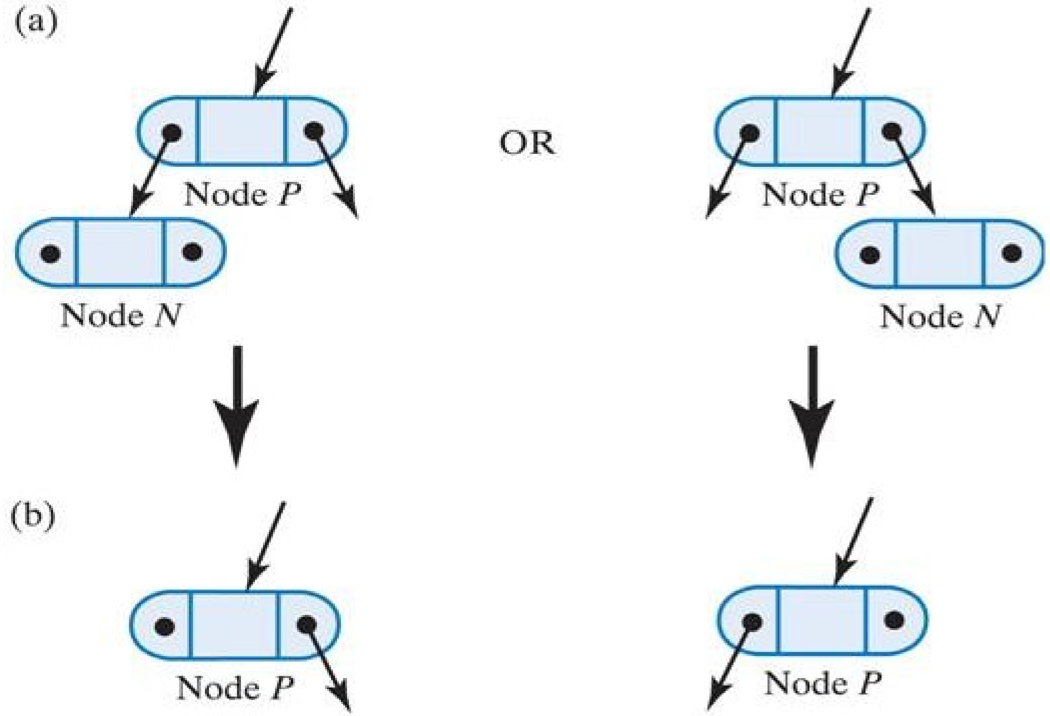
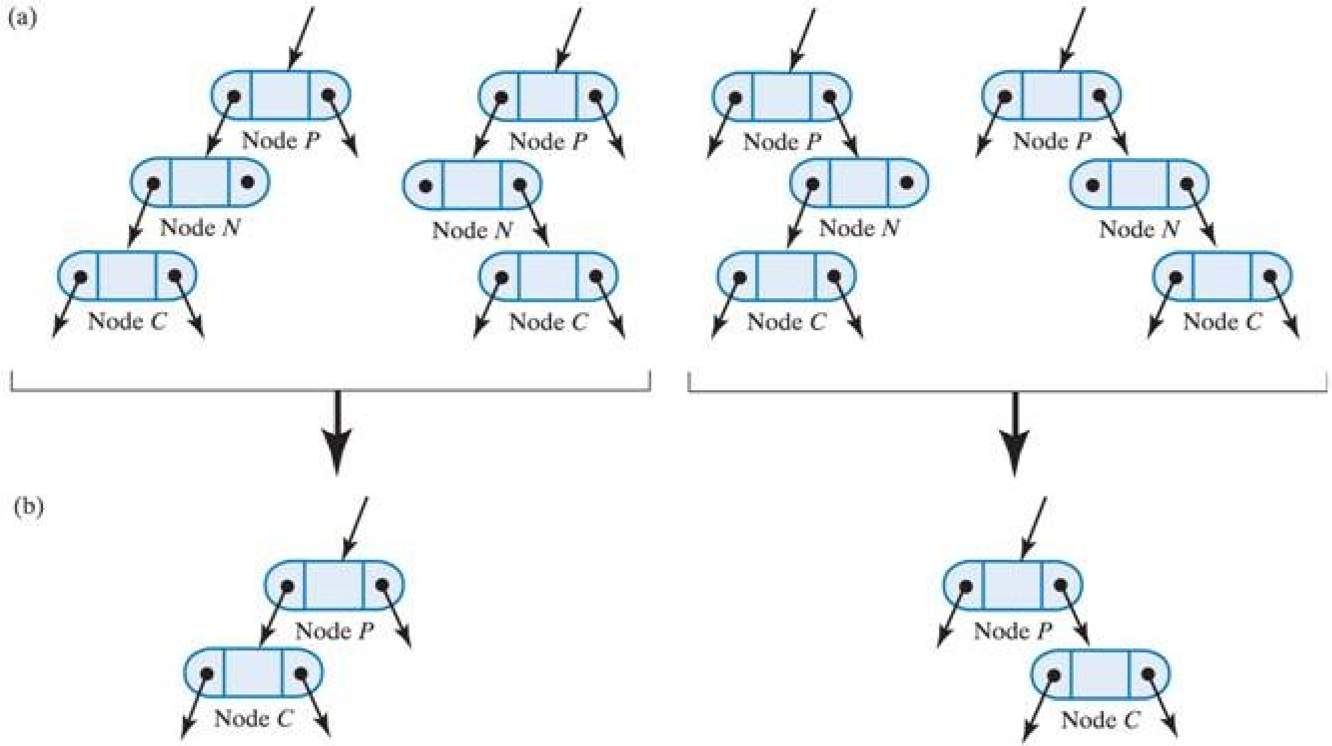


Fig. 27-6 (a) Two possible configurations of leaf node *N*;
(b) the resulting two possible configurations after removing node *N by* setting the child reference of P to null.

2. Removing a Node with 1 Child



make C a child of P instead of N

Fig. 27-6 (a) Two possible configurations of leaf node *N*; (b) the resulting two possible configurations after removing node N.

3. Removing a Node with 2 Children

If we remove N, left with two orphans, P can only accept one of them, no room for both. Thus, **removing N** is not a option. We do not want to remove node N to remove its entry. Find a node A that is easy to remove and replace N's entry, then remove node A.

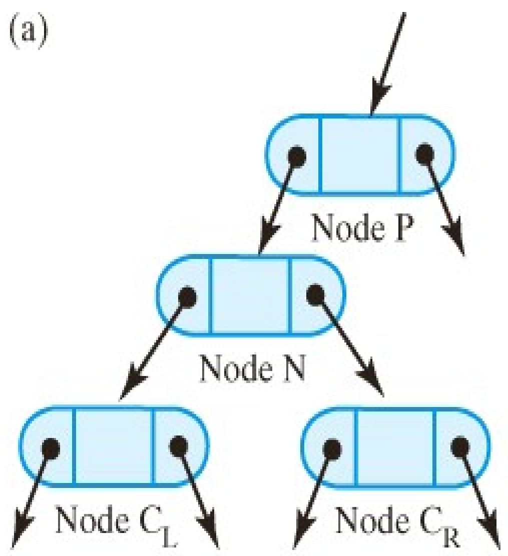


Fig. 27-8 Two possible configurations of node *N* that has two children.

3. Removing a Node with 2 Children

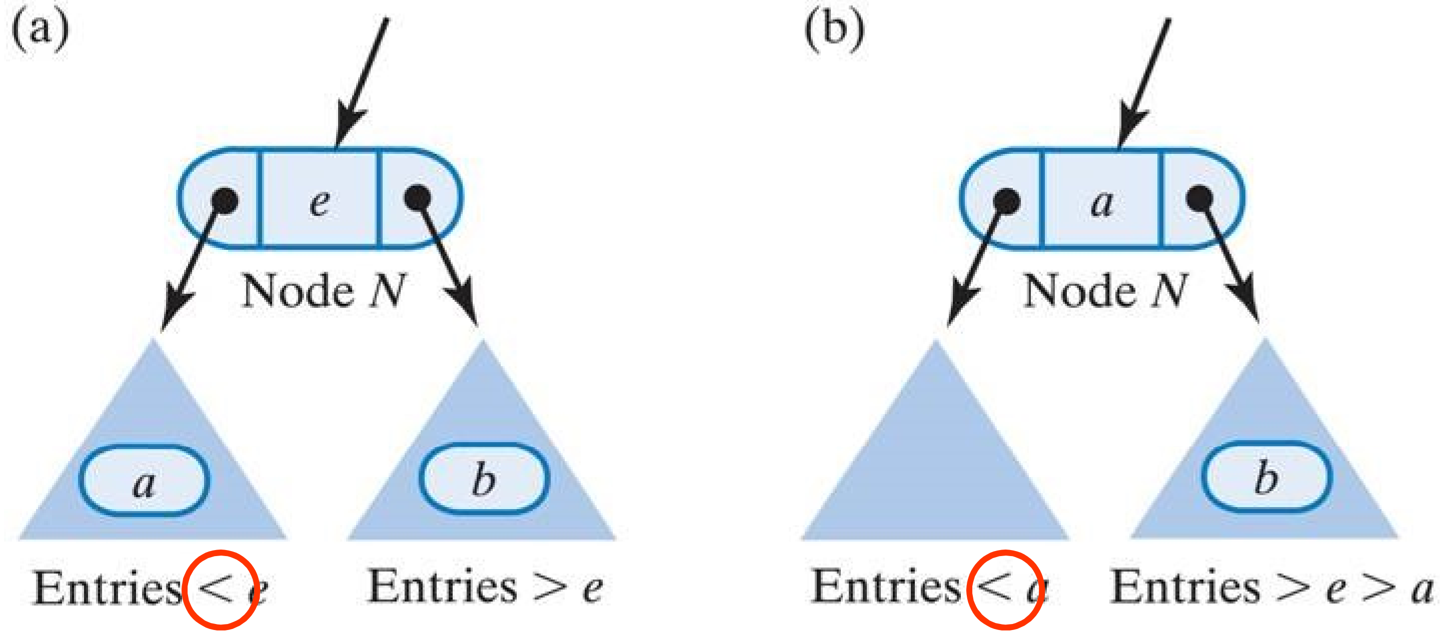


Fig. 27-9 Node *N* and its subtrees;

(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.*

3. Removing a Node with 2 Children

- Delete 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
    - *Note: This will be either deleting a leaf node or a node with only 1 child.*

3. Removing a Node with 2 Children

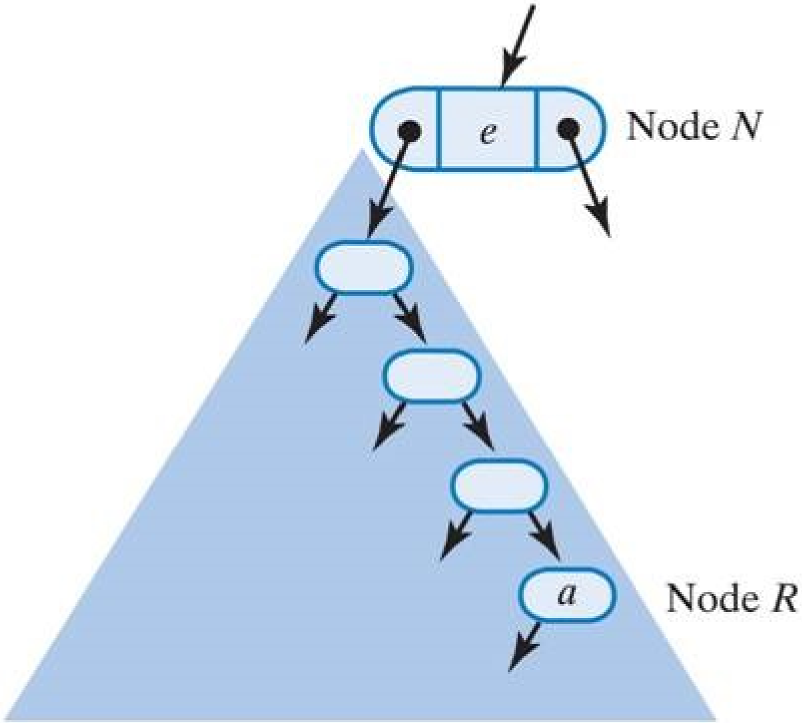


Fig. 27-10 The largest entry *a* in node *N*'s left subtree occurs in the subtree's rightmost node *R*.

3. Removing a Node with 2 Children

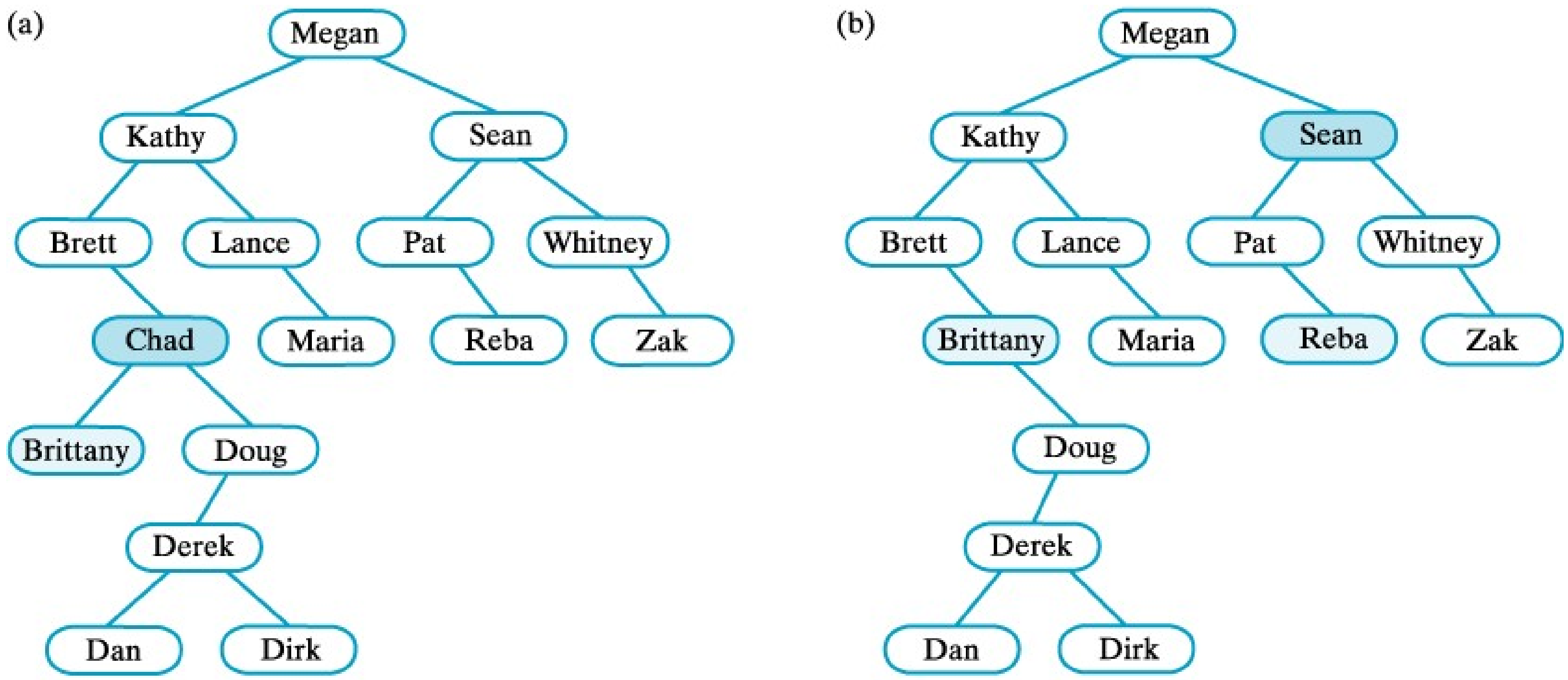


Fig. 27-11 (a) A binary search tree; (b) after removing *Chad*;

3. Removing a Node with 2 Children

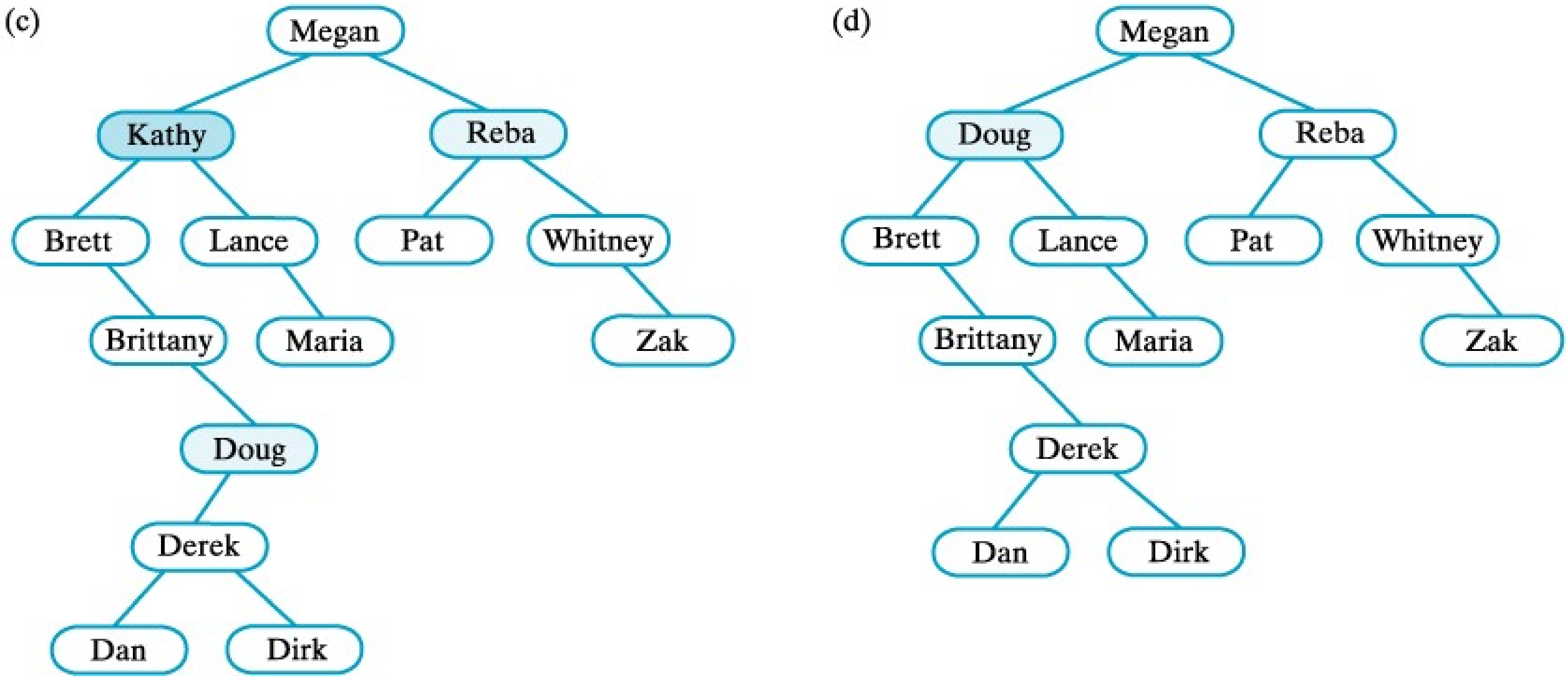
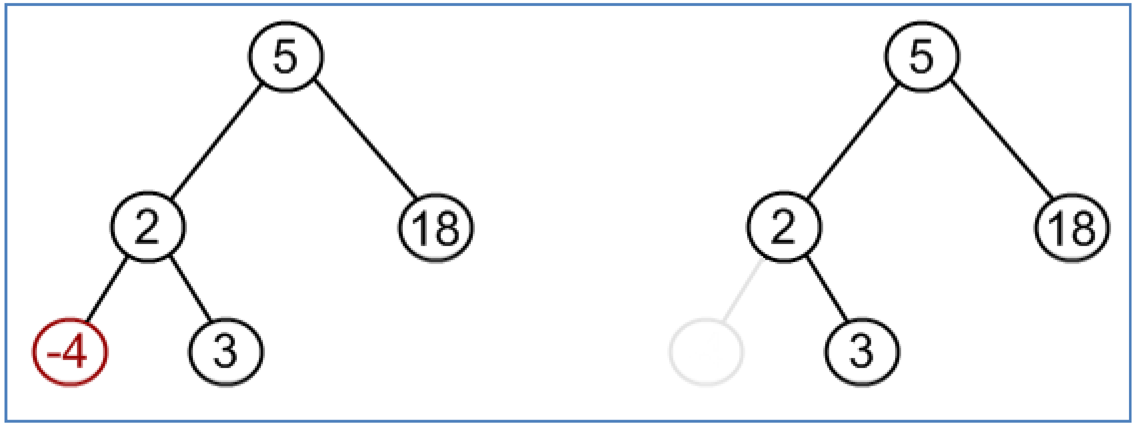


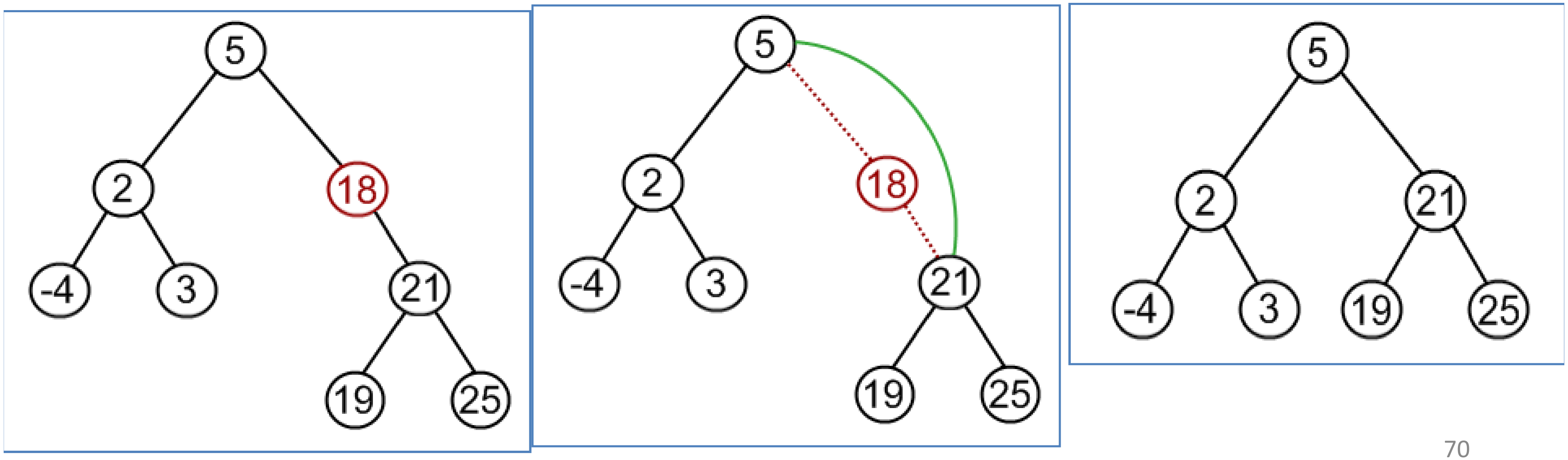
Fig. 27-11 (c) after removing *Sean*; (d) after removing *Kathy.*

Example

- Case 1:remove -4

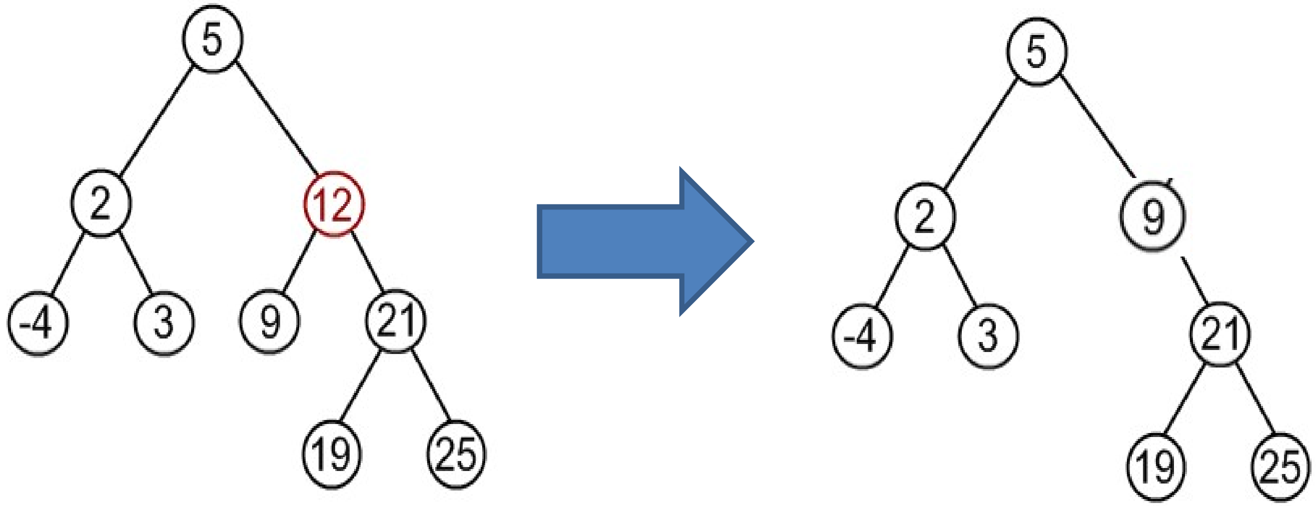


- Case 2:remove 18



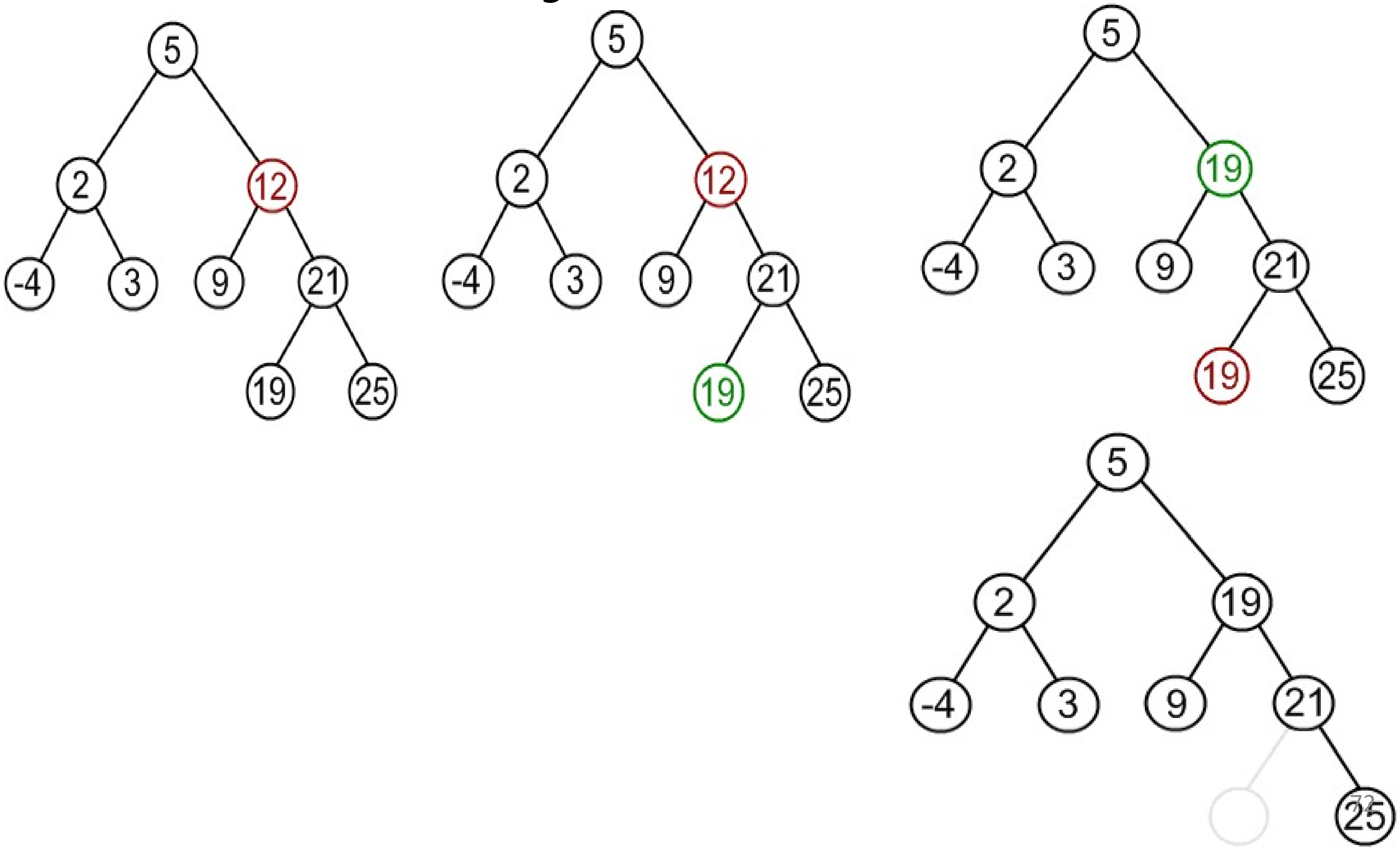
Example

- Case 3: remove 12
- Go to the left subtree of the node 12 with largest value is 9.



Example

- Case 3: remove 12
- (Alternative way) Go to the right subtree of the leftmost node 12 with smallest value is 19.



Removing an Entry in the Root

- The public method **`remove`** calls a private recursive method **`removeEntry`**
  - Method **`remove`** passes the root of the tree to method **`removeEntry`**
  - Since method **`removeEntry`** might remove the root node from the tree, we must ensure a reference to the tree's root is always retained.
    - ➔ **`removeEntry`** returns a reference to the root of the revised tree, which **`remove`** can use to update root
  - However, **`removeEntry`** must also give to **`remove`** the entry it removes
    - ➔ An additional parameter - **`oldEntry`** – is passed to **`removeEntry`** for the method to change its value to the removed entry.

Removing an Entry in the Root

- To enable the additional parameter `oldEntry`'s value to be changed to the removed entry's value, the inner class `ReturnObject` is defined.
- `ReturnObject`
  - An inner class that has a single data field and simple `set` and `get` methods
  - Used as the type for `oldEntry`

Methods remove and removeEntry

```
public T remove(T entry) {
  ReturnObject oldEntry =
        new ReturnObject(null);

  BinaryNode newRoot =
     removeEntry(root, entry,
       oldEntry);

  root = newRoot;

  return oldEntry.get();
}
```

```
private BinaryNode removeEntry(BinaryNode rootNode, T entry,
      ReturnObject oldEntry) {
  if (rootNode != null) {
    T rootData = rootNode.data;
    int comparison = entry.compareTo(rootData);
    if (comparison == 0) {
      oldEntry.set(rootData);
      rootNode = removeFromRoot(rootNode);
    }
    else if (comparison < 0) {
      BinaryNode leftChild = rootNode.left;
      BinaryNode subtreeRoot = removeEntry(leftChild, entry,
       oldEntry);
      rootNode.left = subtreeRoot;
    }
    else {
      BinaryNode rightChild = rootNode.right;
      rootNode.right = removeEntry(rightChild, entry, oldEntry);
    } // end if
  } // end if
  return rootNode;
}
```

Removing an Entry in the Root

It will be a special case only if we actually remove the root node. It occurs when the root has at most one child

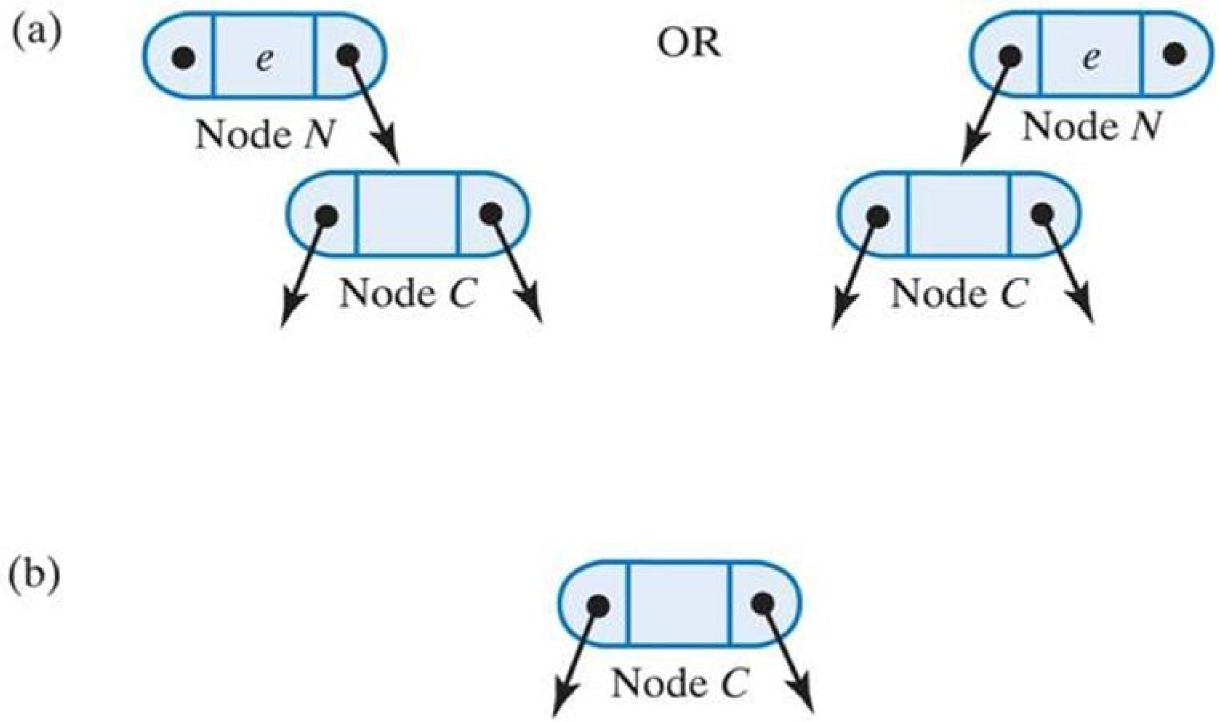
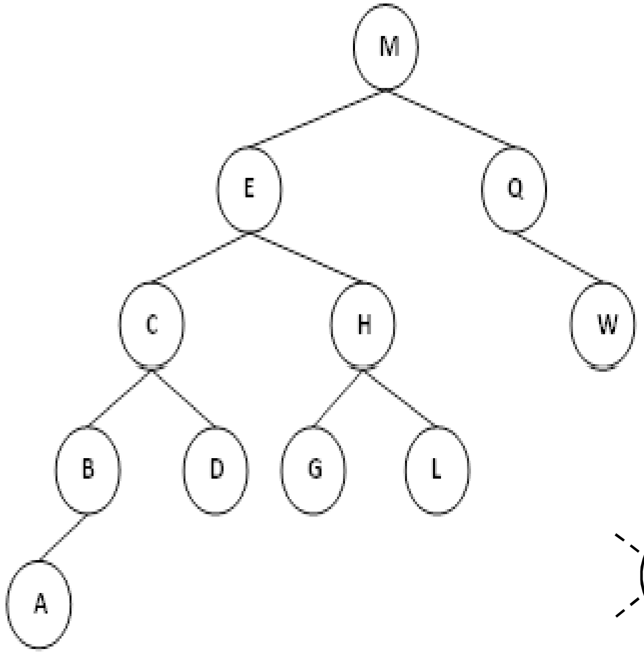
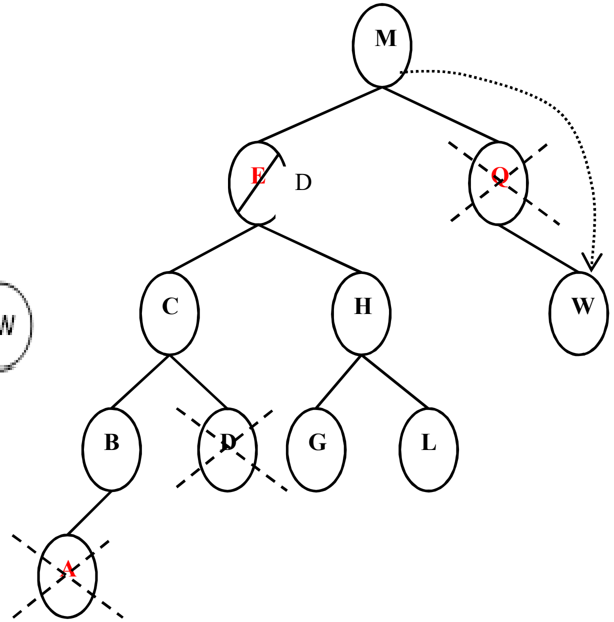


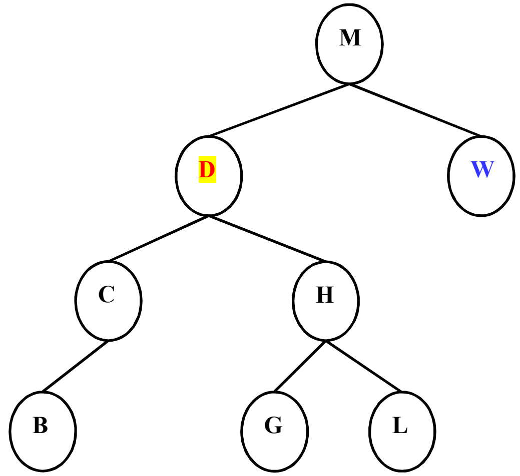
Fig. 27-12 (a) Two possible configurations of a root that has one child; (b) after removing the root.

Exercise

- Explain the steps to remove the values **A, Q** and **E** (*in the given sequence*) from the tree. Show the resulting binary search tree.







Resulting binary search tree

Efficiency of Operations (1)

- Operations **`add`**, **`remove`** and **`getEntry`** require a search that begins at the root
- In the worse case, searches begins at root and examine each node on a path and ends at a leaf.
- Maximum number of comparisons is directly proportional to the height, *h* of the tree. These operations are O(*h*).

Efficiency of Operations (2)

- The tallest tree has height n if it contains n nodes. The tree looks like a linked chain, and the searching is like searching a linked chain, $O(n)$.
- The shortest tree is full. The height of a full tree containing n nodes is $\log_2(n+1)$. Thus, in the worst case, searching a full binary search tree is an $O(\log_2 n)$.
- Both full and complete binary search tree can give us $O(\log_2 n)$ performance.

Efficiency of Operations

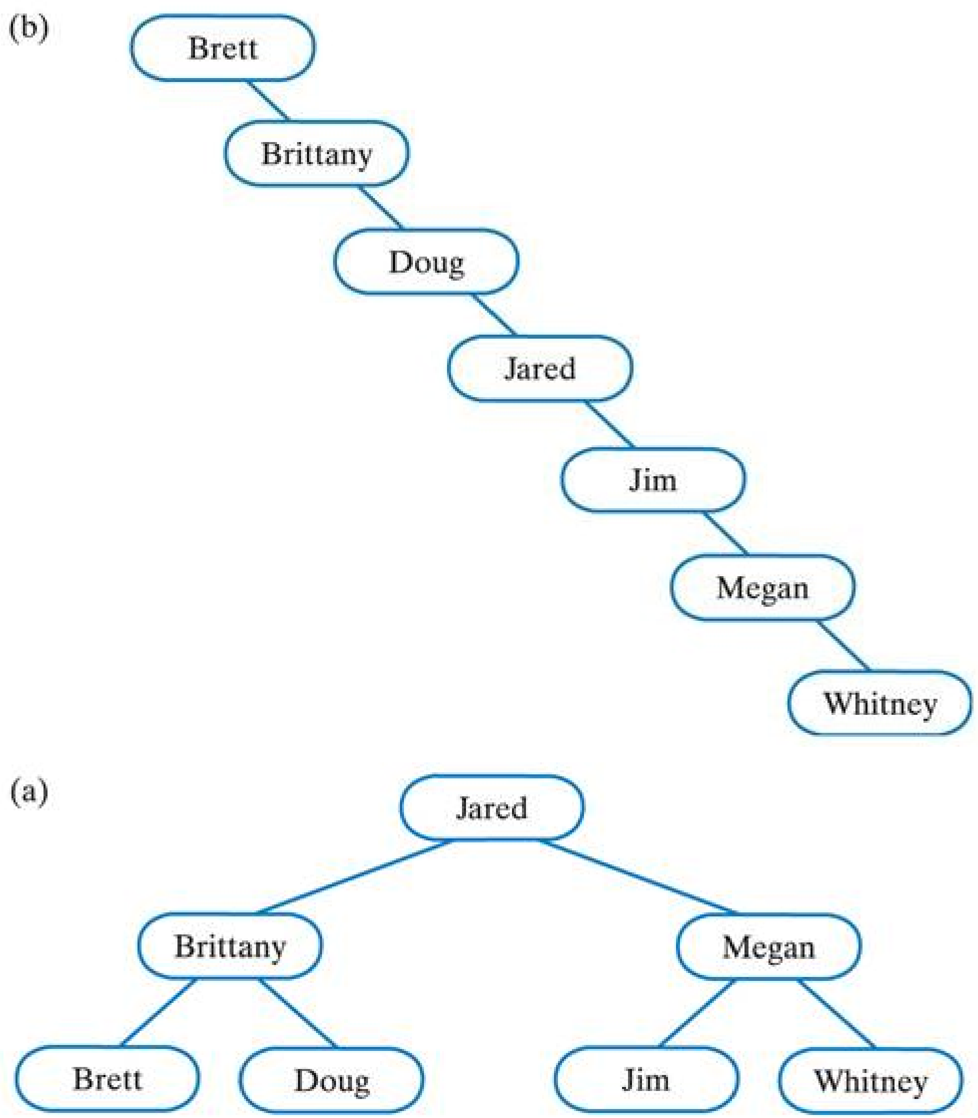


Fig. 27-13 Two binary search trees that contain the same data.

Shorter tree has efficiency O(log n)

Importance of Balance

- Balance of the tree affects the performance of the search process
- *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

Importance of Balance

- If entries are added to an empty binary tree
  - Best <u>not</u> to have them sorted first
  - Tree is more balanced if entries are in random order

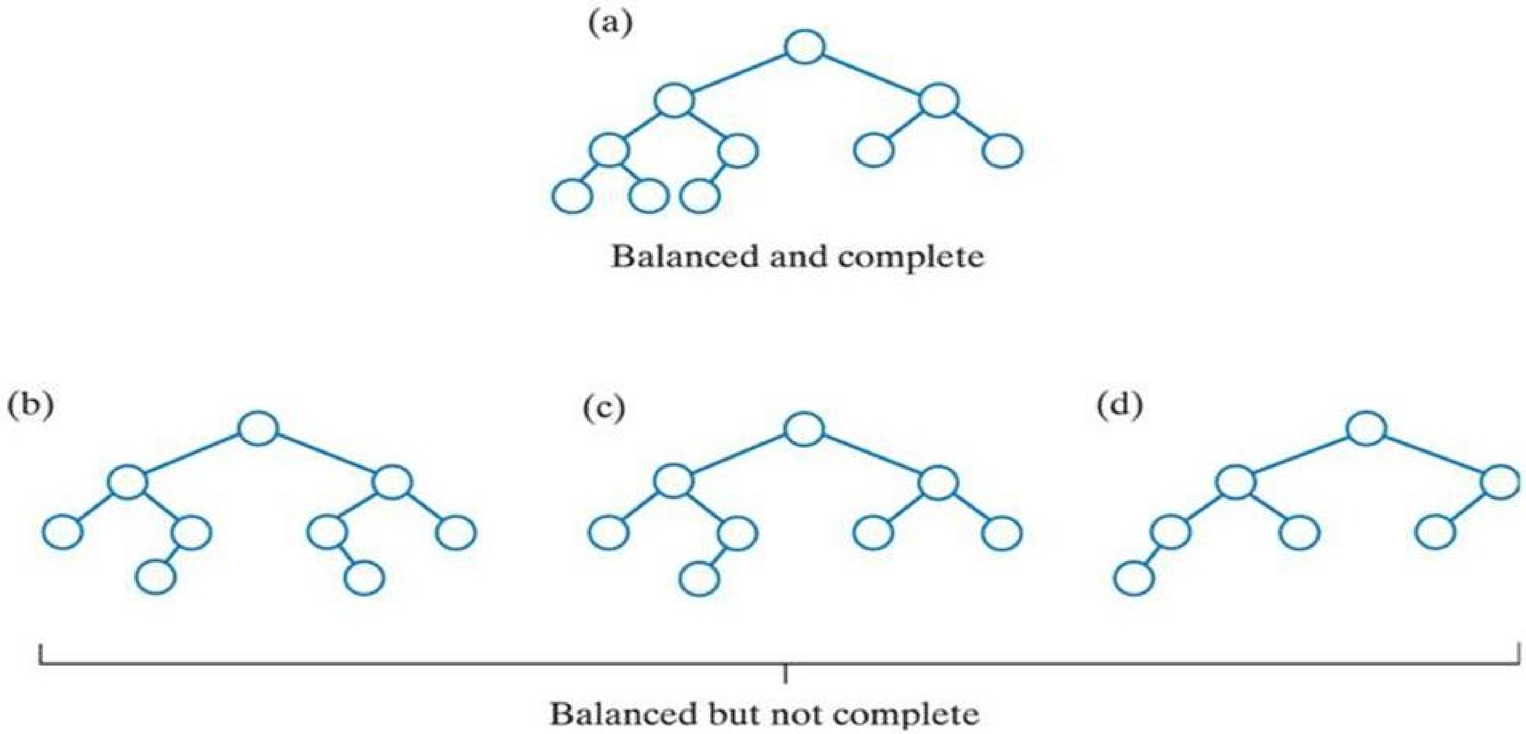


Fig. 27-14 Some binary trees that are height balanced.

Balanced Search Trees (Optional)

- The operations on a binary search tree are O(log *n*) if the tree is balanced. However, the add and remove operations do not ensure that a binary search tree remains balanced.
- *Balanced search trees* carries out rearrangement of nodes whenever it becomes unbalanced. Types of balanced search trees include:
  - AVL Trees
  - 2-3 Trees
  - B-Trees

Review of learning outcomes

You should now be able to

- Describe applications of binary trees such as expression trees and decision trees.
- Implement binary trees and binary search trees
- Discuss the factors that affect the efficiency of the binary search tree operations

References

- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson