King Fahd University of Petroleum & Minerals
College of Computer Science & Engineering

Information & Computer Science Department

Unit 7

Trees, Tree Traversals and Binary Search Trees

Information and Computer Science ICS

Reading Assignment

- "Data Structures and Algorithms in Java", 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
  - Chapter 6 Sections 1-6.
  - Section 6.4.3 regarding "Stackless Depth-First Traversal" and "Threaded Trees" is omitted.

Objectives

Discuss the following topics:

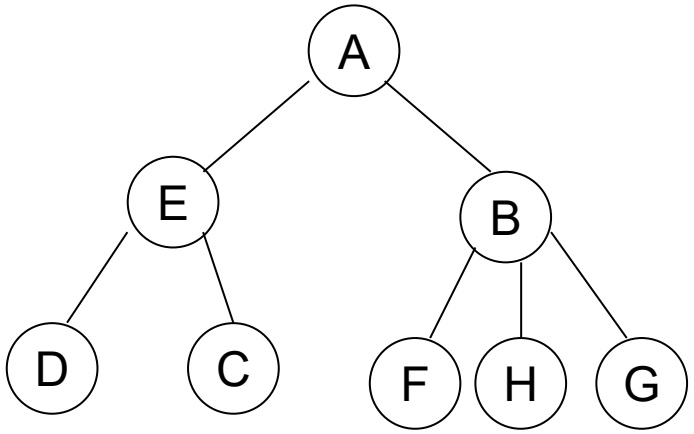
- Trees, Binary Trees, and Binary Search Trees
  - Implementing Binary Trees
  - Searching a Binary Search Tree
  - Tree Traversal
  - Binary Search Tree Insertion
  - Binary Search Tree Deletion

Definition of a Tree

- A tree, is a finite set of nodes together with a finite set of edges (arcs) that define parent-child relationships. Each edge connects a parent to its child. Example:

Nodes={A,B,C,D,E,f,G,H}

Edges={(A,B),(A,E),(B,F),(B,G),(B,H),
(E,C),(E,D)}

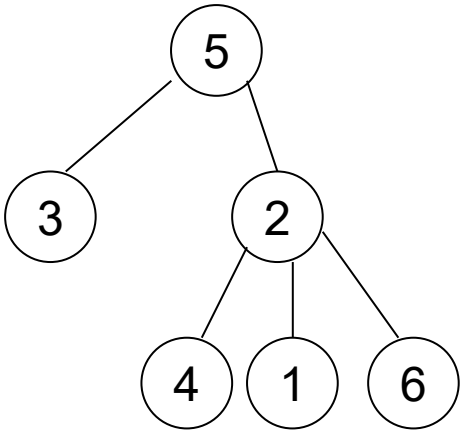


- A **path** from node m_1 to node m_k is a list of nodes $m_1, m_2, \ldots, m_k$ such that each is the parent of the next node in the list.
  - The length of such a path is k - 1.
- Example: A, E, C is a path of length 2.

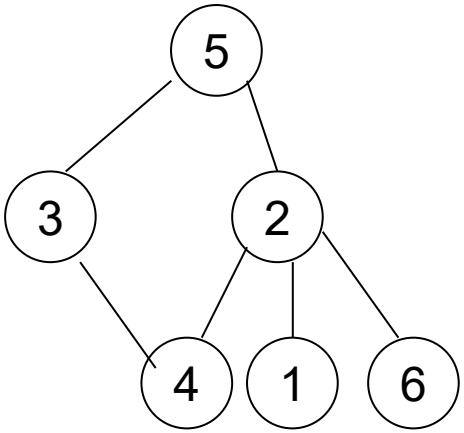
Definition of a Tree (Cont.)

- A tree satisfies the following properties:

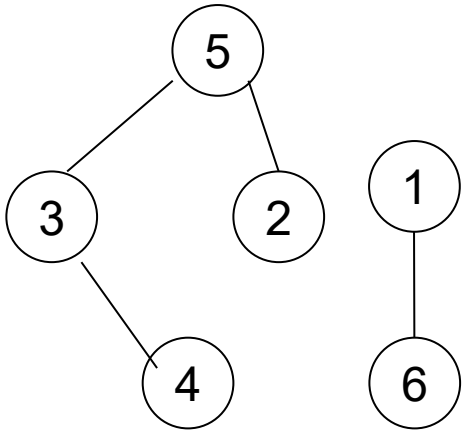
1. It has one designated node, called the root, that has no parent.
2. Every node, except the root, has exactly one parent.
3. A node may have zero or more children.
4. There is a unique directed path from the root to each node.



tree



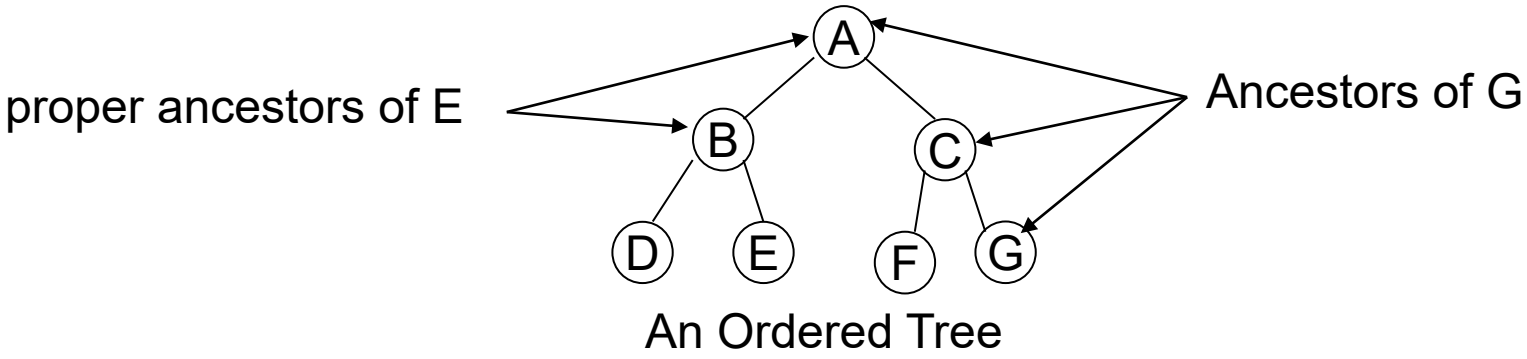
Not a tree



Not a tree

Tree Terminology

- **Ordered tree**: A tree in which the children of each node are linearly ordered (usually from left to right).



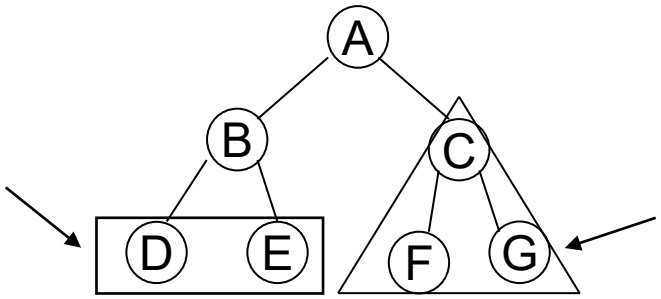
An Ordered Tree

- **Ancestor** of a node v: Any node, including v itself, on the path from the root to the node.
- **Proper ancestor** of a node v: Any node, excluding v, on the path from the root to the node.

Tree Terminology (Contd.)

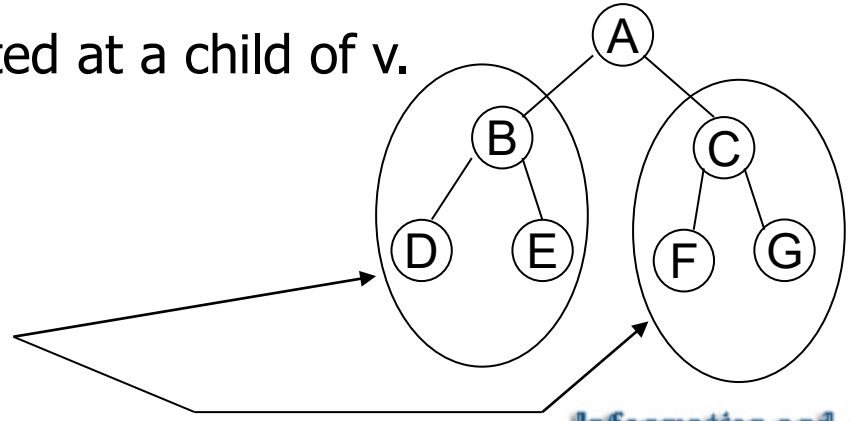
- **Descendant** of a node v: Any node, including v itself, on any path from the node to a leaf node (i.e., a node with no children).

Proper descendants of node B



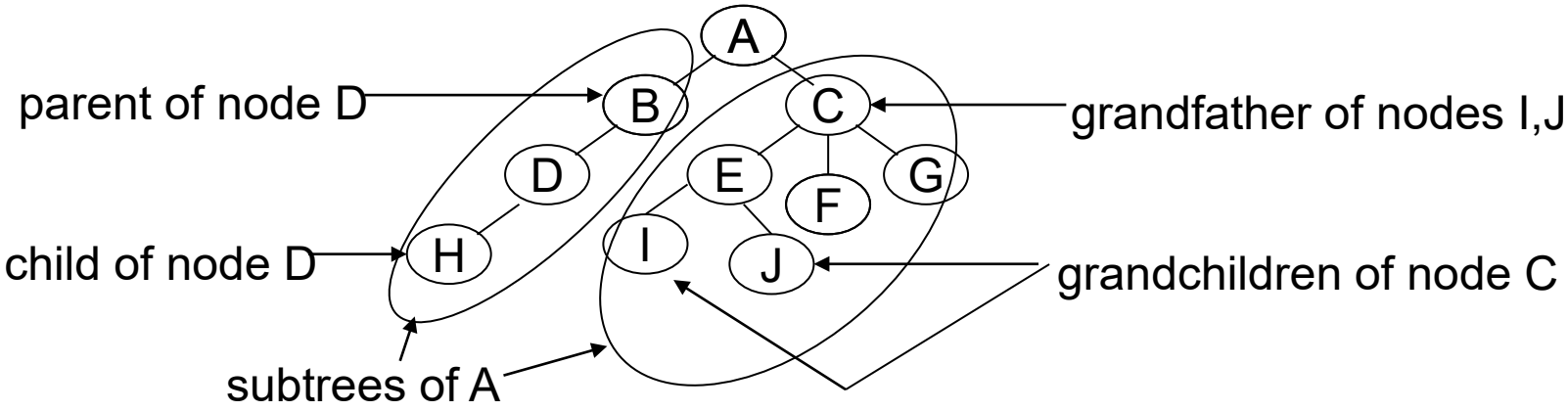
Descendants of a node C

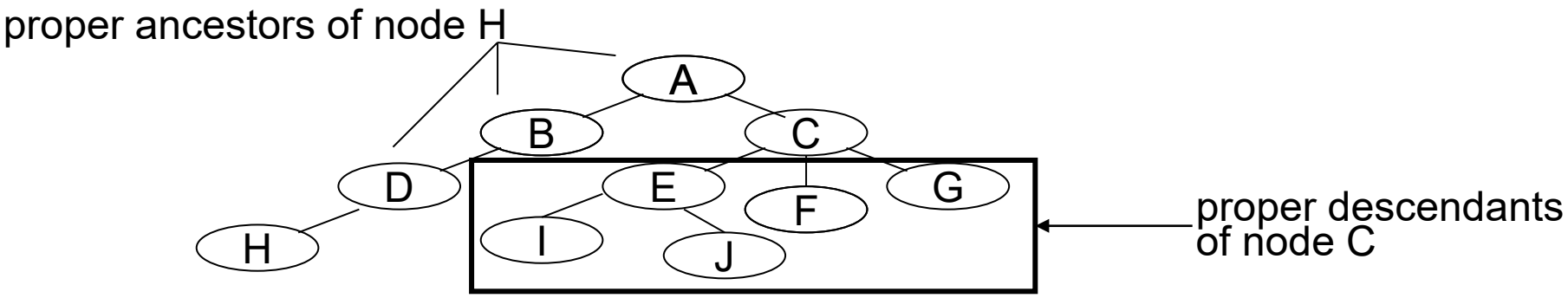
- **Proper descendant** of a node v: Any node, excluding v, on any path from the node to a leaf node.
- **Subtree** of a node v: A tree rooted at a child of v.



subtrees of node A

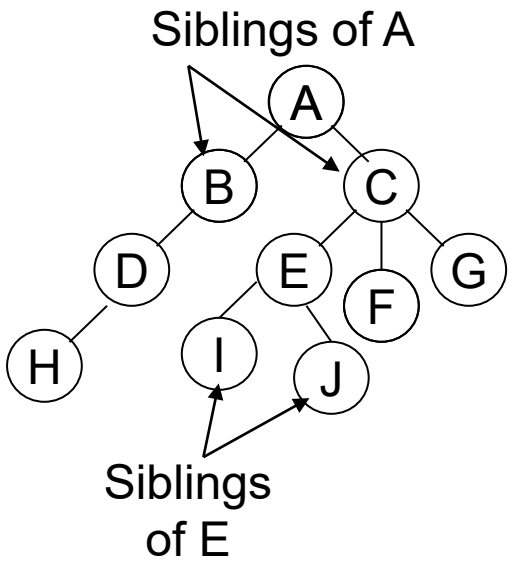
Tree Terminology (Contd.)





Tree Terminology (Contd.)

An Ordered Tree with size of 10

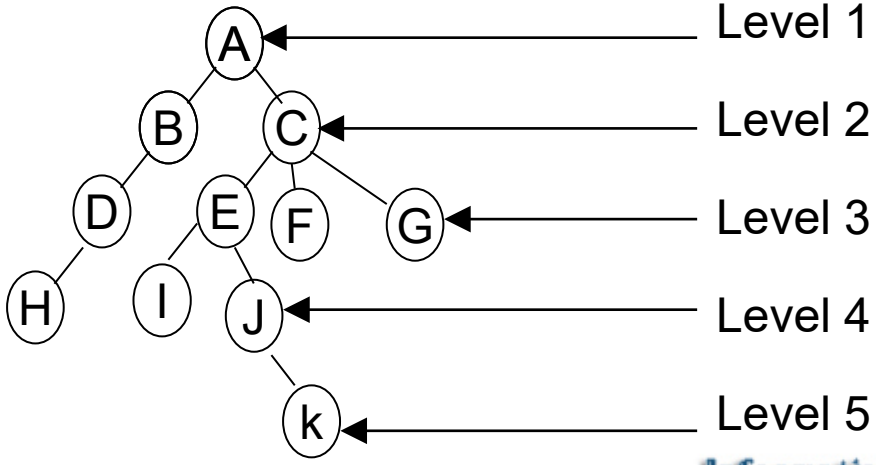


- **Degree**: The number of subtrees of a node
  - Each of node D and B has degree 1.
  - Each of node A and E has degree 2.
  - Node C has degree 3.
  - Each of node F,G,H,I,J has degree 0.
- **Leaf**: A node with degree 0.
- **Nonterminal** or internal node: a node with degree greater than 0.
- **Siblings**: Nodes that have the same parent.
- **Size**: The number of nodes in a tree.

Tree Terminology (Contd.)

- **Level** (or depth) of a node v: The length of the path from the root to node v plus one.
  - Same as the number of nodes in the path.
- **Height** of a nonempty tree: The maximum level of a node in a tree.
  - By definition the height of an empty tree is 0.

- The height of the tree is 5



Example Trees

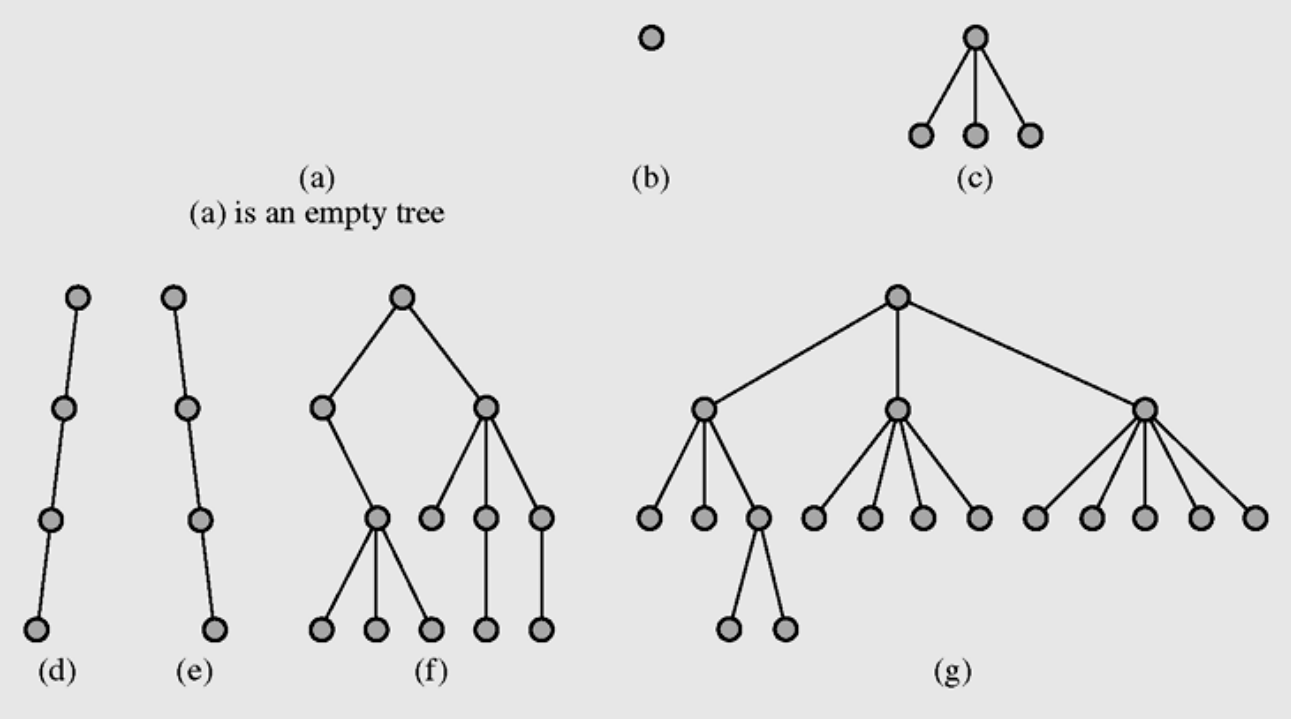


Figure 6-1 Examples of trees

Importance of Trees

- Trees are very important data structures in computing.
- They are suitable for:
  - Hierarchical structure representation, e.g.,
    - File directory.
    - Organizational structure of an institution.
    - Class inheritance tree.
  - Problem representation, e.g.,
    - Expression tree.
    - Decision tree.
  - Efficient algorithmic solutions, e.g.,
    - Search trees.
    - Efficient priority queues via heaps.

Hierarchical Structure Representation

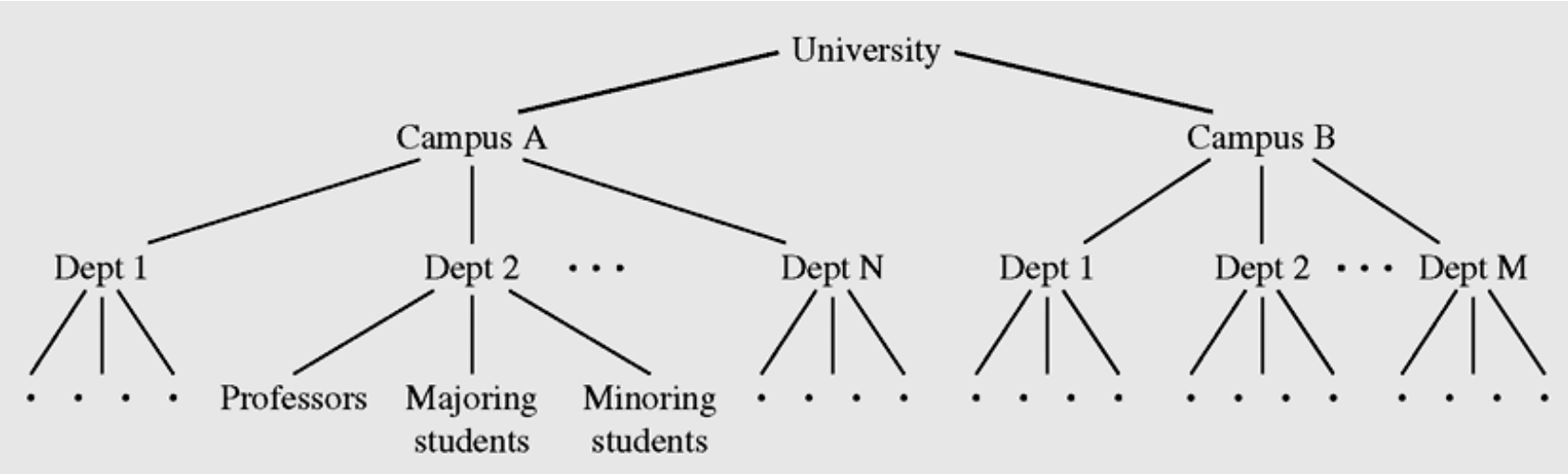


Figure 6-2 Hierarchical structure of a university shown as a tree

Orderly Trees

- An **orderly tree** is where all elements are stored according to some predetermined criterion of ordering

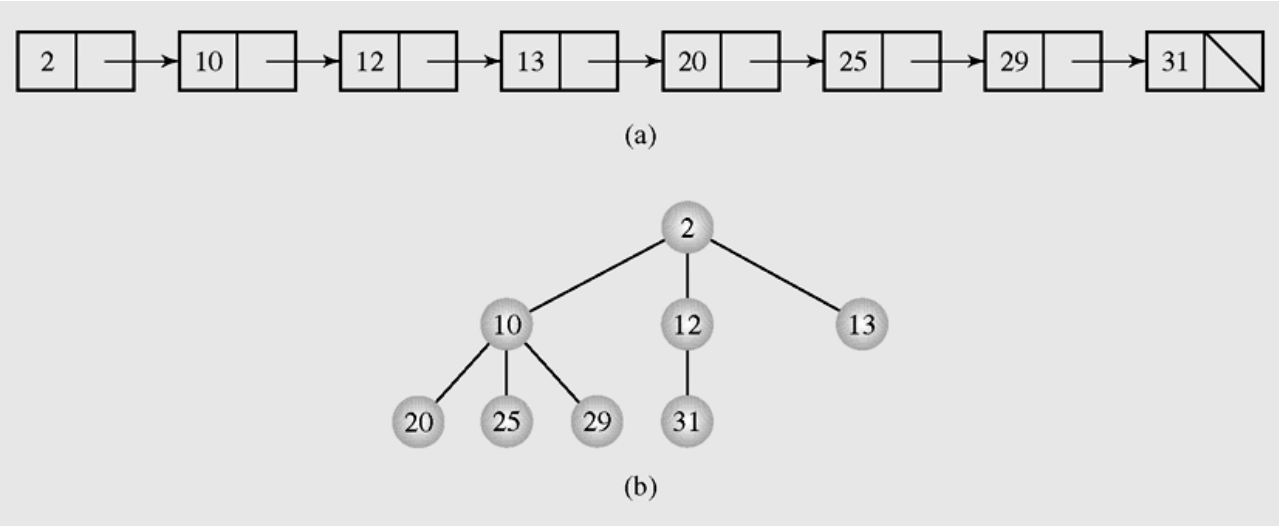


Figure 6-3 Transforming (a) a linked list into (b) a tree

Binary Trees

- A **binary tree** is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or a right child

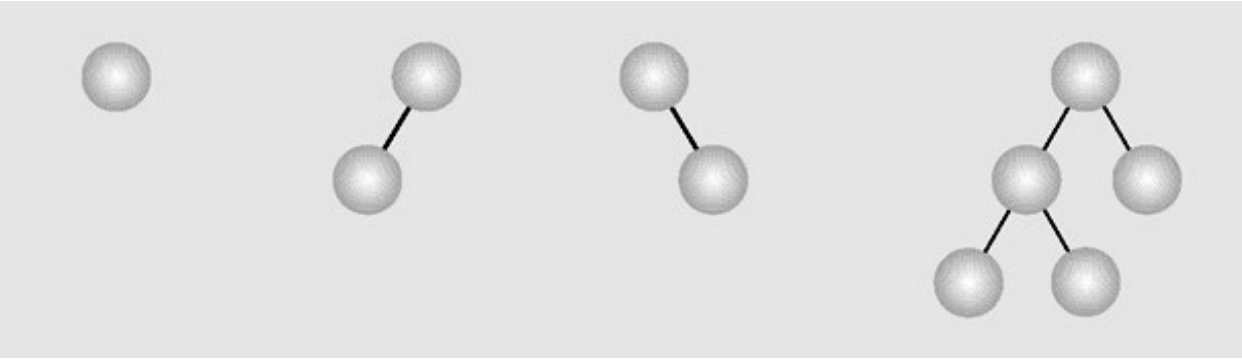
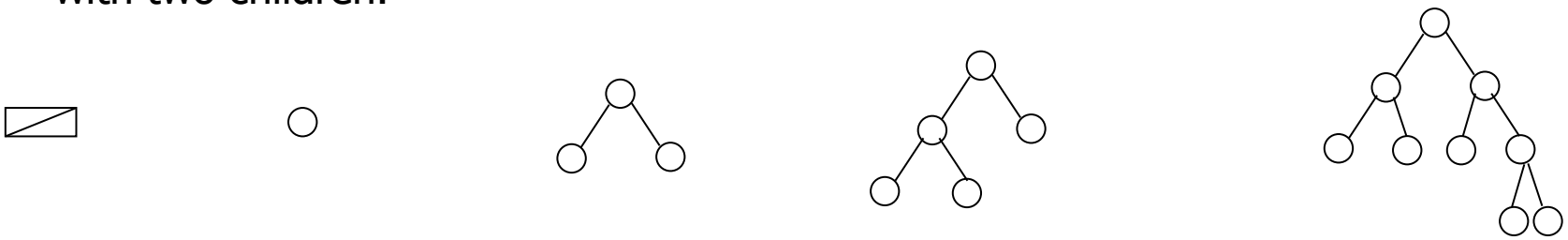


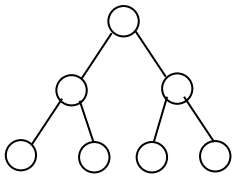
Figure 6-4 Examples of binary trees

Binary Trees

- **Definition**: A **decision tree (full binary tree)** is either an empty binary tree or a binary tree in which every node is either a leaf node or an internal node with two children.



- **Definition**: A **complete binary tree** is either an empty binary tree or a binary tree in which all nonterminal nodes have both their children, and all leaves are at the same level



Binary Trees

Theorem: The number of leaves in a non-empty decision tree is one more than the number of nonterminal nodes.

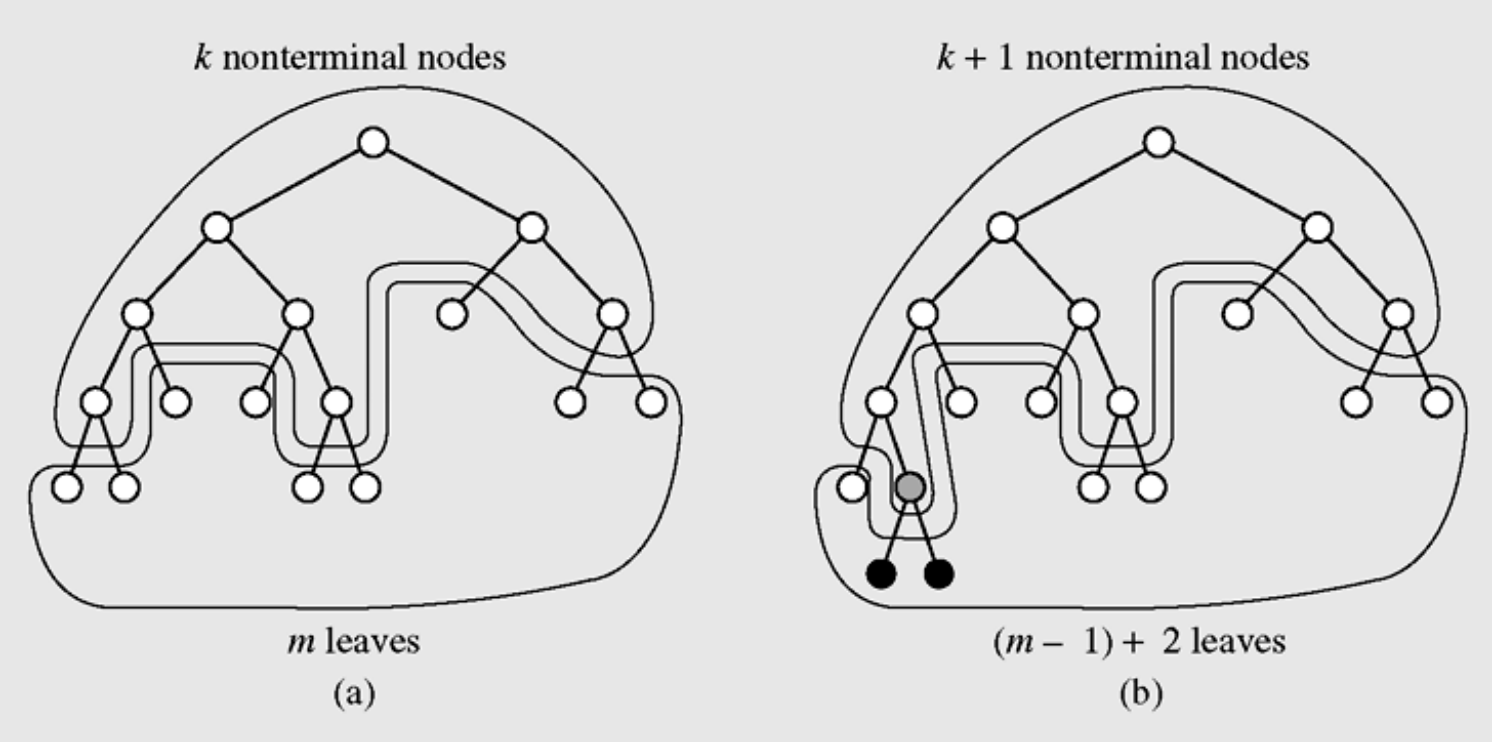


Figure 6-5 Adding a leaf to tree (a), preserving the relation of the number of leaves to the number of nonterminal nodes (b)

Binary Trees

- What is the maximum height of a binary tree with n elements?

$$n$$

- What is the minimum height of a binary tree with n elements?

$$\lceil \lg(n+1) \rceil$$

  - What is the minimum and maximum heights of a complete binary tree?

    Both are $\lceil \lg(n+1) \rceil$

  - What is the minimum and maximum heights of a decision (full binary) tree?

    $\lceil \lg(n+1) \rceil$ and $\lceil n/2 \rceil$

Binary Search Trees

- **Definition**: A **binary search tree (BST)** is a binary tree that is empty or that satisfies the BST ordering property:
  1. The key of each node is greater than each key in the left subtree, if any, of the node.
  2. The key of each node is less than each key in the right subtree, if any, of the node.
- Thus, each key in a BST is unique.

Binary Search Tree Examples

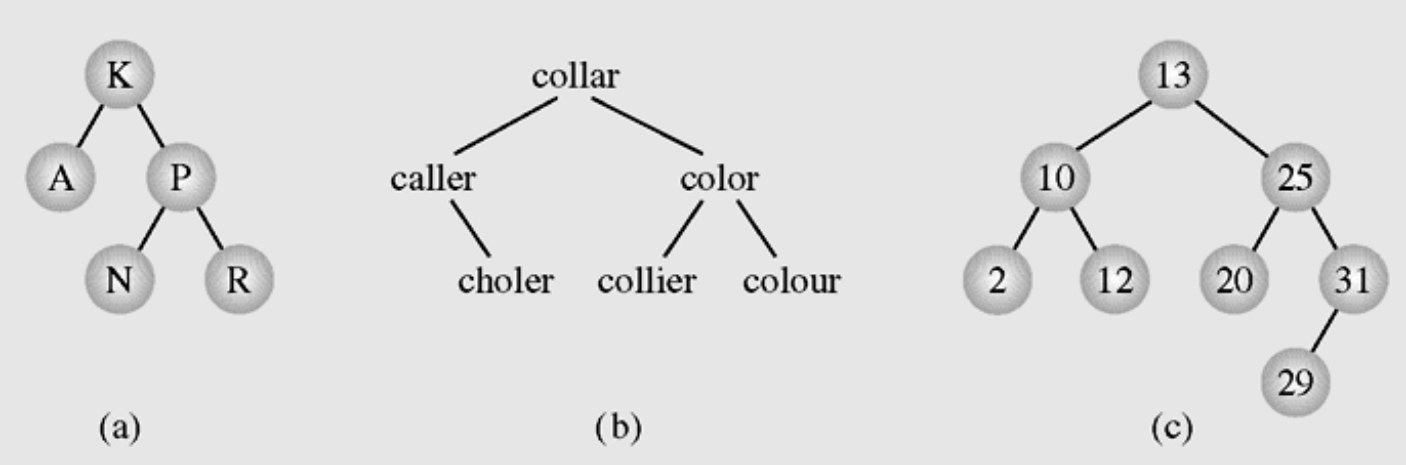
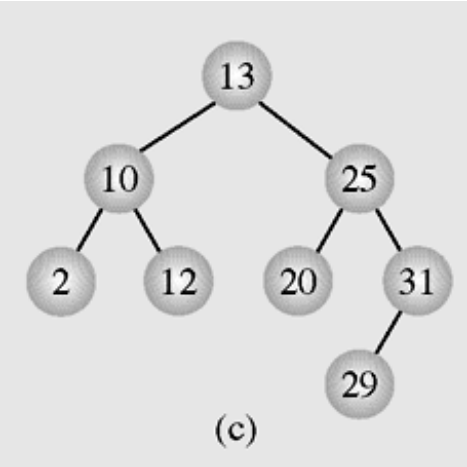


Figure 6-6 Examples of binary search trees

Implementing Binary Trees

- Binary trees can be implemented in at least two ways:
  - As arrays
  - As linked structures
- To implement a tree as an array, a node is declared as an object with an information field and two "reference" fields

Implementing Binary Trees



Index	Info	Left	Right
0	13	4	2
1	31	6	–1
2	25	7	1
3	12	–1	–1
4	10	5	3
5	2	–1	–1
6	29	–1	–1
7	20	–1	–1

Figure 6-7 Array representation of the tree in Figure 6.6c

Implementing Binary Trees

```
/************************  BSTNode.java  ***************************
 *              node of a generic binary search tree
 */

public class BSTNode<T extends Comparable<? super T>> {
    protected T el;
    protected BSTNode<T> left, right;
    public BSTNode() {
        left = right = null;
    }
    public BSTNode(T el) {
        this(el,null,null);
    }
    public BSTNode(T el, BSTNode<T> lt, BSTNode<T> rt) {
        this.el = el; left = lt; right = rt;
    }
}
```

Figure 6-8 Implementation of a generic binary search tree

Implementing Binary Trees

```
/************************  BST.java  ***************************
 *                  generic binary search tree
 */

public class BST<T extends Comparable<? super T>> {
    protected BSTNode<T> root = null;
    public BST() {
    }
    protected void visit(BSTNode<T> p) {
        System.out.print(p.el + " ");
    }
    protected T search(T el) {…}
    public void breadthFirst() {…}
    public void preorder() {
        preorder(root);
    }
    public void inorder() {
        inorder(root);
    }
    public void postorder() {
        postorder(root);
    }
```

Figure 6-8 Implementation of a generic binary search tree (continued)

Implementing Binary Trees

```
    protected void inorder(BSTNode<T> p) {…}
    protected void preorder(BSTNode<T> p) {…}
    protected void postorder(BSTNode<T> p) {…}
    public void deleteByCopying(T el) {…}
    public void deleteByMerging(T el) {…}
    public void iterativePreorder() {…}
    public void iterativeInorder() {…}
    public void iterativePostorder2() {…}
    public void iterativePostorder() {…}
    public void MorrisInorder() {…}
    public void MorrisPreorder() {…}
    public void MorrisPostorder() {…}
    public void balance(T data[], int first, int last) {…}
    public void balance(T data[]) {…}
}
```

Figure 6-8 Implementation of a generic binary search tree (continued)

Searching a Binary Search Tree

```
protected T search(T el) {
        BSTNode<T> p = root;
        while (p != null)
            if (el.equals(p.el))
                return p.el;
            else if (el.compareTo(p.el) < 0)
                p = p.left;
            else p = p.right;
        return null;
}
```

Figure 6-9 A function for searching a binary search tree

Searching a Binary Search Tree

- The **internal path length (IPL)** is the sum of all path lengths of all nodes
  - It is calculated by summing

    $\Sigma(i-1)L_i$

    over all levels *i,* where L_i is the number of nodes on level *L*
- A depth of a node in the tree is determined by the path length
- An average depth, called an **average path length**, is given by the formula IPL/*n,* which depends on the shape of the tree

Importance of BSTs

- BSTs provide good logarithmic time performance in the best and average cases.
- Average case complexities of using linear data structures compared to BSTs:

Data Structure	Retrieval	Insertion	Deletion
BST	O(log n) FAST	O(log n) FAST	O(log n) FAST
Sorted Array	O(log n) FAST*	O(n) SLOW	O(n) SLOW
Sorted Linked List	O(n) SLOW	O(n) SLOW	O(n) SLOW

*using binary search

Tree Traversal (Definition)

- The process of systematically visiting every node once in a tree and performing some computation at each node in the tree is called a tree traversal.

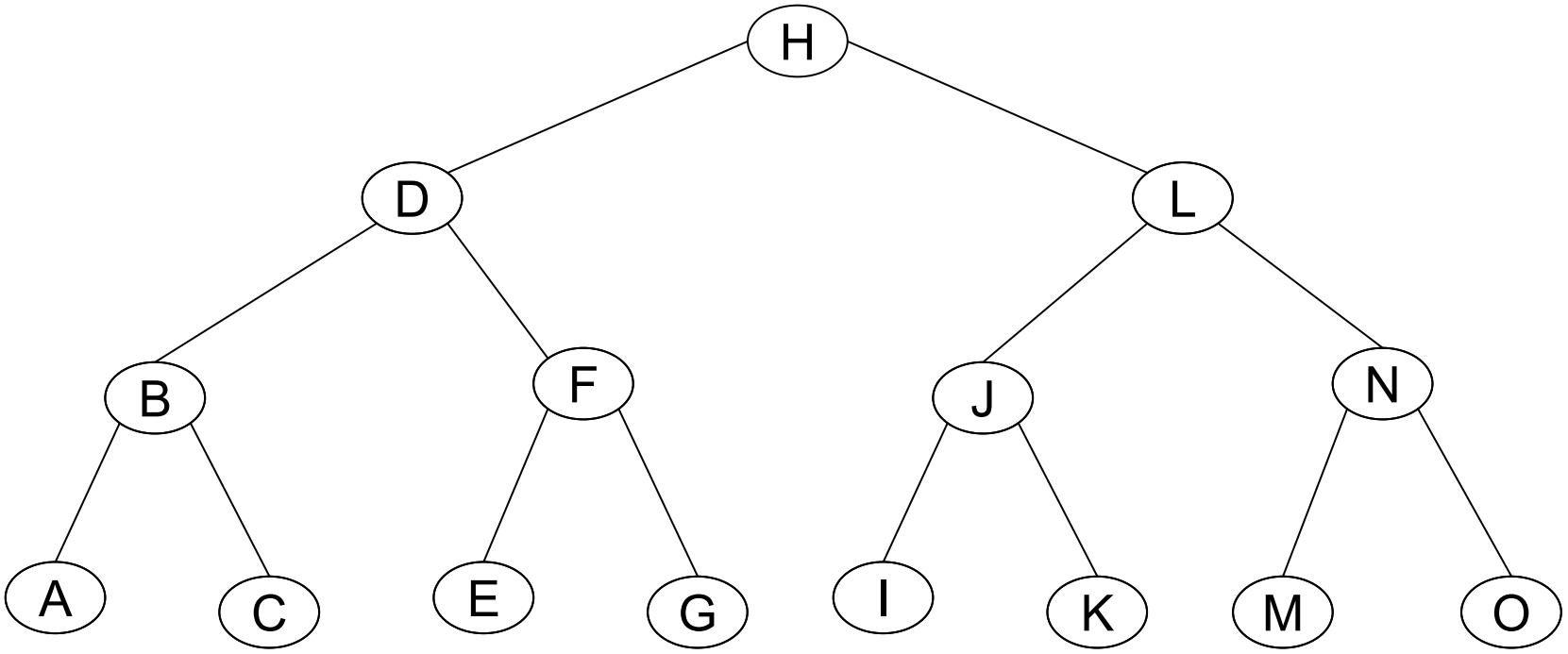
- There are two methods in which to traverse a tree:
  1. Breadth-First Traversal.
  2. Depth-First Traversal:
     - Preorder traversal
     - Inorder traversal (for binary trees only)
     - Postorder traversal

Breadth-First Traversal

```
public void breadthFirst() {
       BSTNode<T> p = root;
       Queue<BSTNode<T>> queue = new Queue<BSTNode<T>>();
       if (p != null) {
            queue.enqueue(p);
            while (!queue.isEmpty()) {
                p = queue.dequeue();
                visit(p);
                if (p.left != null)
                     queue.enqueue(p.left);
                if (p.right != null)
                     queue.enqueue(p.right);
            }
       }
   }
```

Figure 6-10 Top-down, left-to-right, breadth-first traversal implementation

Breadth-First Traversal



H	D	L	B	F	J	N	A	C	E	G	I	K	M	O

Depth-First Traversal

- **Depth-first traversal** proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right)
  - V — Visiting a node
  - L — Traversing the left subtree
  - R — Traversing the right subtree

Depth-First Traversals

Name	for each Node:
Preorder (V-L-R)	•Visit the node •Visit the left subtree, if any. •Visit the right subtree, if any.
Inorder (L-V-R)	•Visit the left subtree, if any. Visit the node •Visit the right subtree, if any.
Postorder (L-R-V)	•Visit the left subtree, if any. •Visit the right subtree, if any. •Visit the node

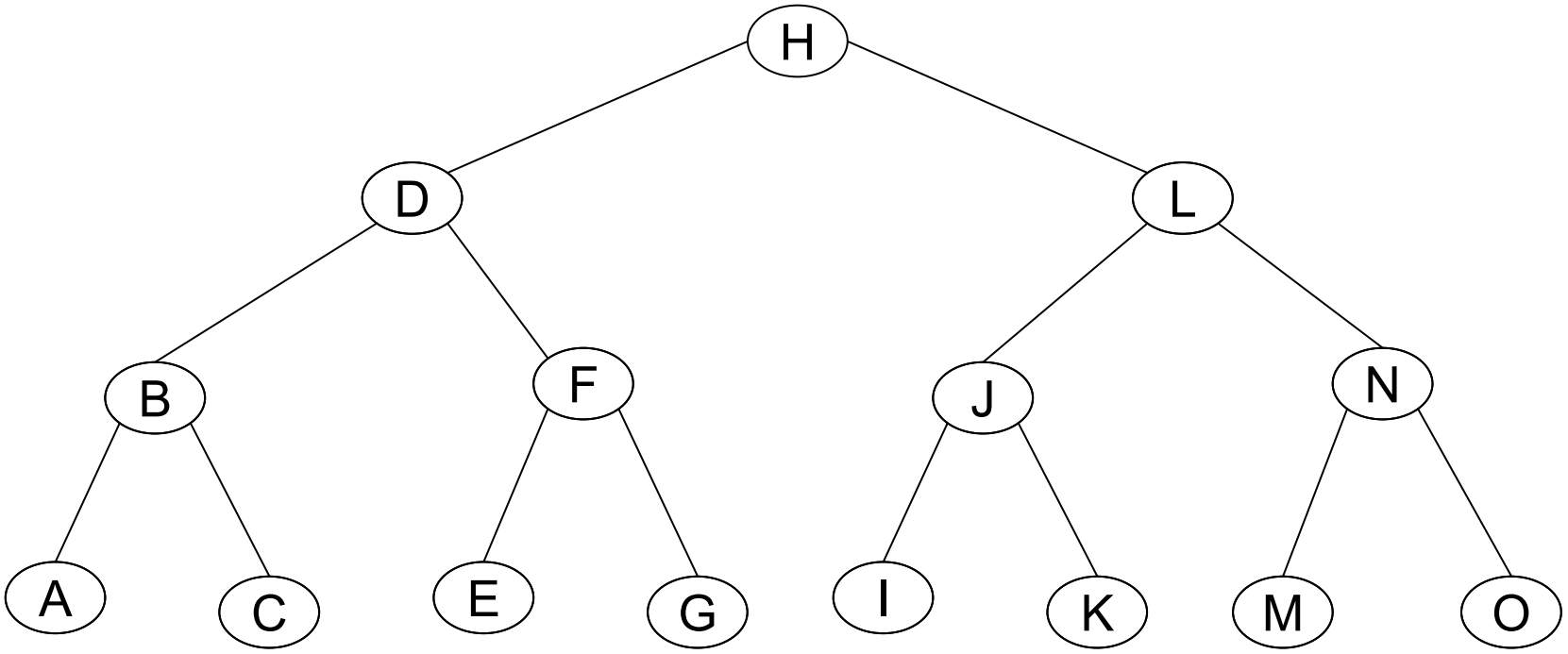
Depth-First Traversal (continued)

```
protected void inorder(BSTNode<T> p) {
        if (p != null) {
             inorder(p.left);
             visit(p);
             inorder(p.right);
        }
    }
    protected void preorder(BSTNode<T> p) {
        if (p != null) {
             visit(p);
             preorder(p.left);
             preorder(p.right);
        }
    }
    protected void postorder(BSTNode<T> p) {
        if (p != null) {
             postorder(p.left);
             postorder(p.right);
             visit(p);
        }
    }
```

Figure 6-11 Depth-first traversal implementation

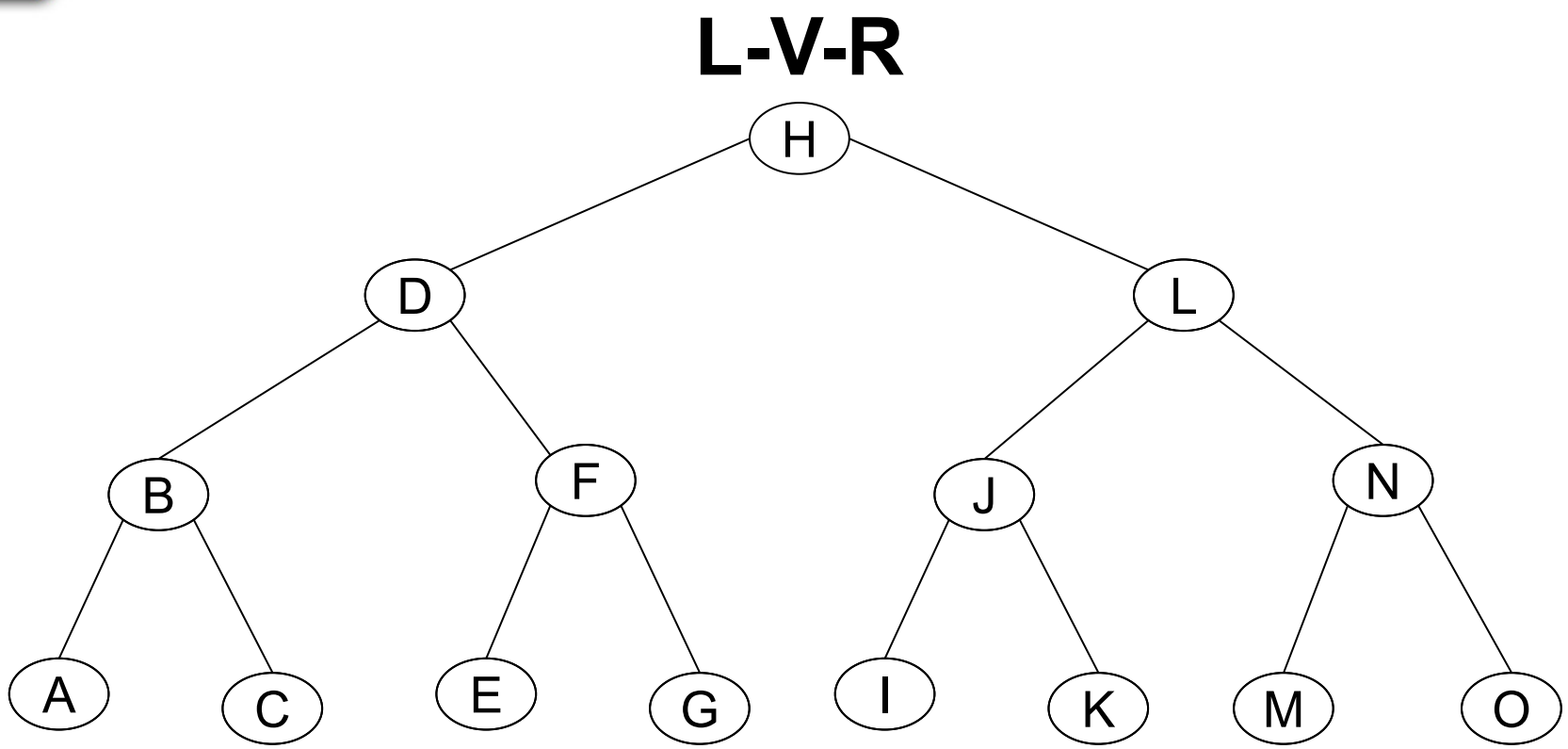
Depth-first Preorder Traversal

V-L-R



H	D	B	A	C	F	E	G	L	J	I	K	N	M	O

Depth-first Inorder Traversal

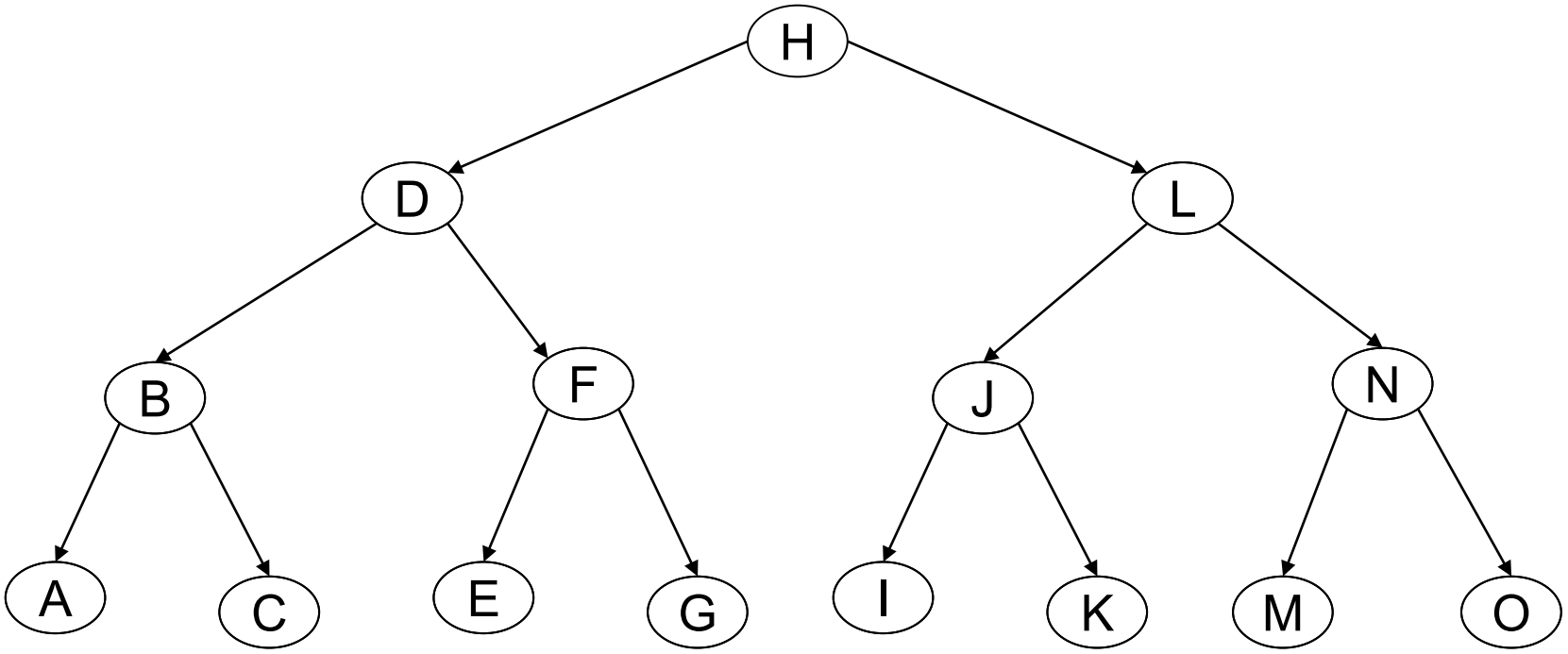


A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Note: An inorder traversal of a BST visits the keys sorted in increasing order.

Depth-first Postorder Traversal

L-R-V



A	C	B	E	G	F	D	I	K	J	M	O	N	L	H

Iterative Preorder Traversal

```
public void iterativePreorder() {
       BSTNode<T> p = root;
       Stack<BSTNode<T>> travStack = new Stack<BSTNode<T>>();
       if (p != null) {
            travStack.push(p);
            while (!travStack.isEmpty()) {
                p = travStack.pop();
                visit(p);
                if (p.right != null)
                     travStack.push(p.right);
                if (p.left  != null)         // left child pushed after right
                     travStack.push(p.left);// to be on the top of the stack;
            }
       }
   }
```

Figure 6-15 A nonrecursive implementation of preorder tree traversal

BST Insertion

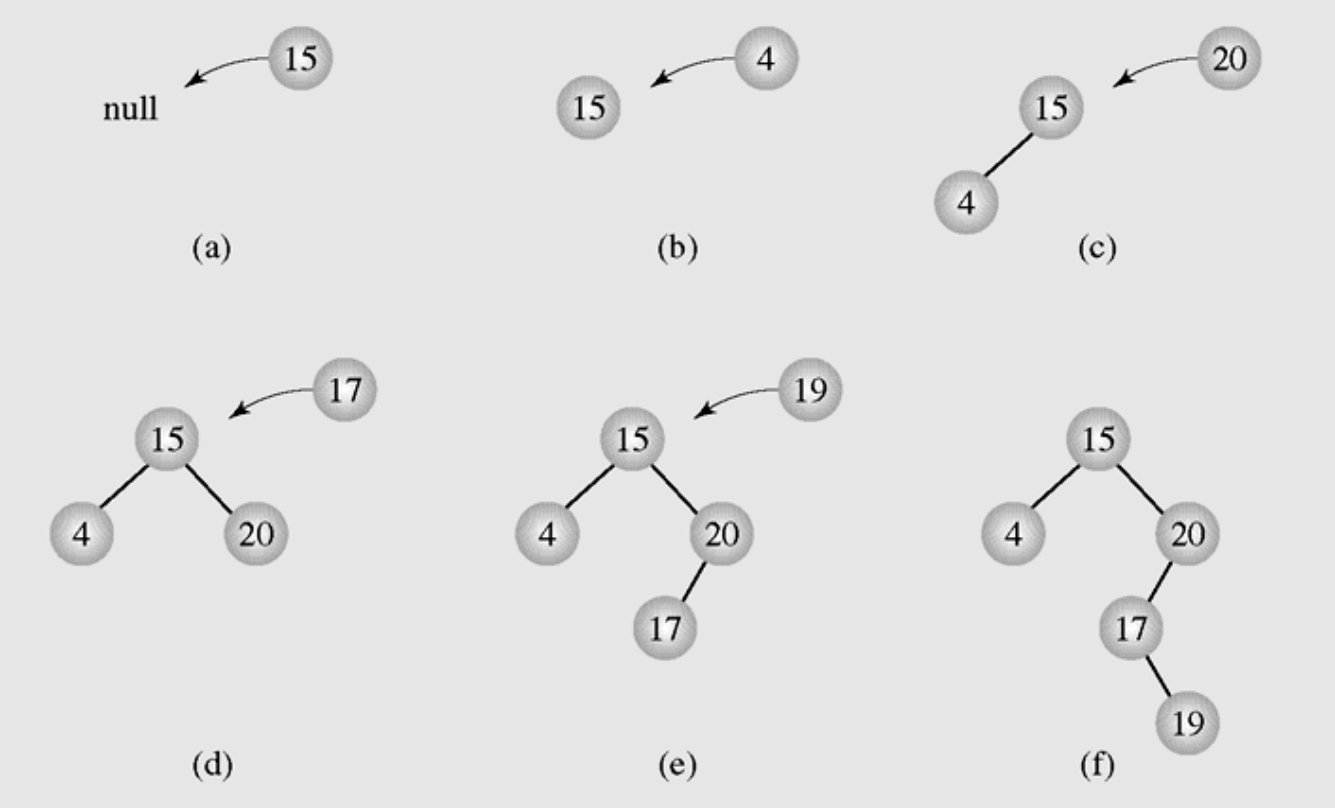


Figure 6-22 Inserting nodes into binary search trees

Insertion (continued)

```
public void insert(T el) {
       BSTNode<T> p = root, prev = null;
       while (p != null) {  // find a place for inserting new node;
           prev = p;
           if (el.compareTo(p.el) < 0)
                p = p.left;
           else p = p.right;
       }
       if (root == null)    // tree is empty;
            root = new BSTNode<T>(el);
       else if (el.compareTo(prev.el) < 0)
            prev.left  = new BSTNode<T>(el);
       else prev.right = new BSTNode<T>(el);
   }
```

Figure 6-23 Implementation of the insertion algorithm

Deletion

- There are three cases of deleting a node from the binary search tree:
  - The node is a leaf; it has no children
  - The node has one child
  - The node has two children

Deletion (continued)

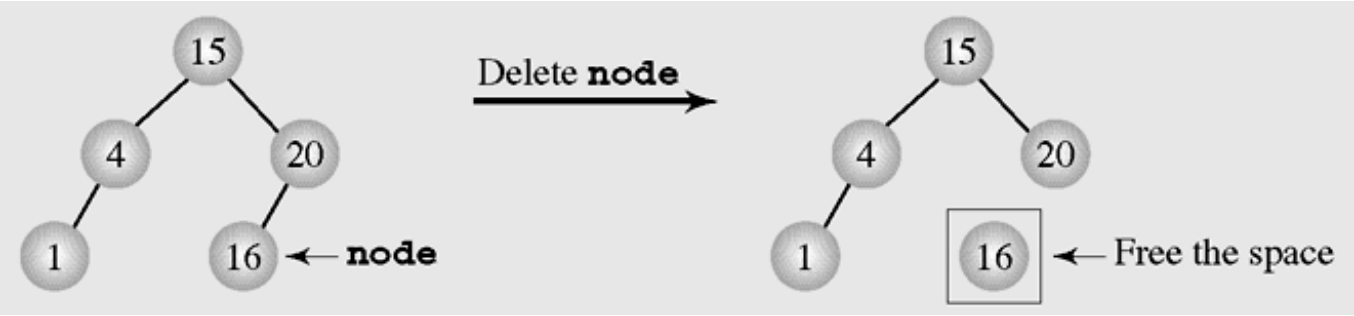


Figure 6-26 Deleting a leaf

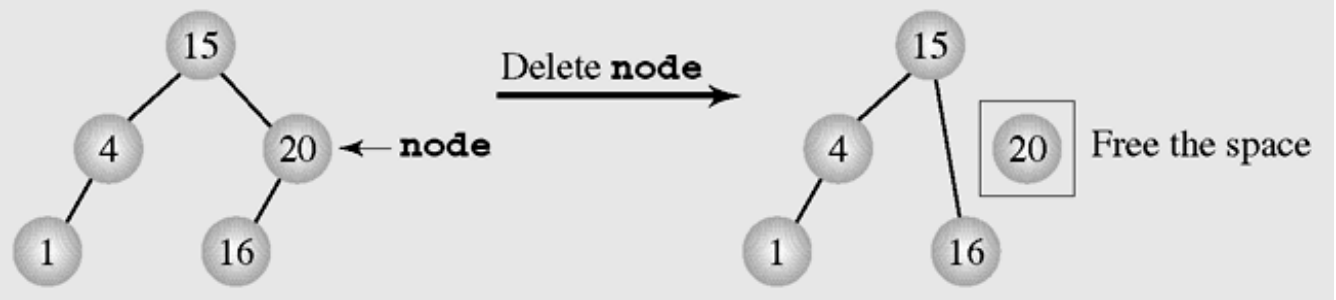
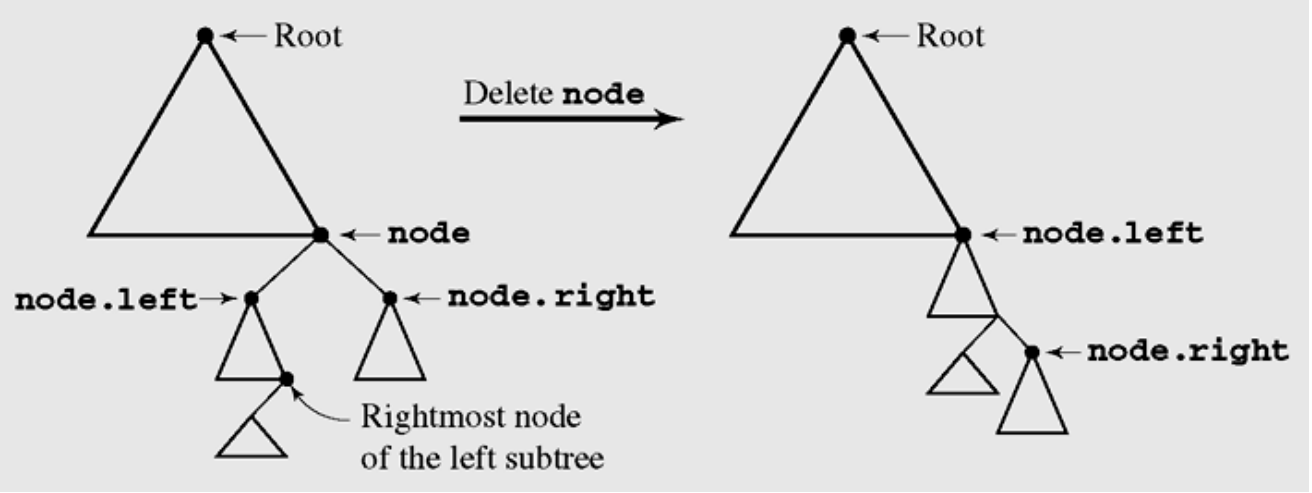


Figure 6-27 Deleting a node with one child

Deletion by Merging

- Making one tree out of the two subtrees of the node and then attaching it to the node's parent is called **deleting by merging**



Deletion by Merging (continued)

```
public void deleteByMerging(T el) {
       BSTNode<T> tmp, node, p = root, prev = null;
       while (p != null && !p.el.equals(el)) {  // find the node p
            prev = p;                           // with element el;
            if (el.compareTo(p.el) < 0)
                 p = p.right;
            else p = p.left;
       }
```

Figure 6-29 Implementation of algorithm for deleting by merging

Deletion by Merging (continued)

```
        node = p;
        if (p != null && p.el.equals(el)) {
             if (node.right == null) // node has no right child: its left
                  node = node.left;  // child (if any) is attached to its parent;
             else if (node.left == null) // node has no left child: its right
                  node = node.right; // child is attached to its parent;
             else {                  // be ready for merging subtrees;
                  tmp = node.left;   // 1. move left
                  while (tmp.right != null) // 2. and then right as far as
                      tmp = tmp.right;      //    possible;
                  tmp.right =        // 3. establish the link between
                      node.right;    //    the rightmost node of the left
                                     //    subtree and the right subtree;
                  node = node.left;  // 4.
             }
             if (p == root)
                  root = node;
             else if (prev.left == p)
                  prev.left = node;
             else prev.right = node; // 5.
        }
        else if (root != null)
             System.out.println("el " + el + " is not in the tree");
        else System.out.println("the tree is empty");
    }
```

Figure 6-29 Implementation of algorithm for deleting by merging (continued)

Deletion by Merging (continued)

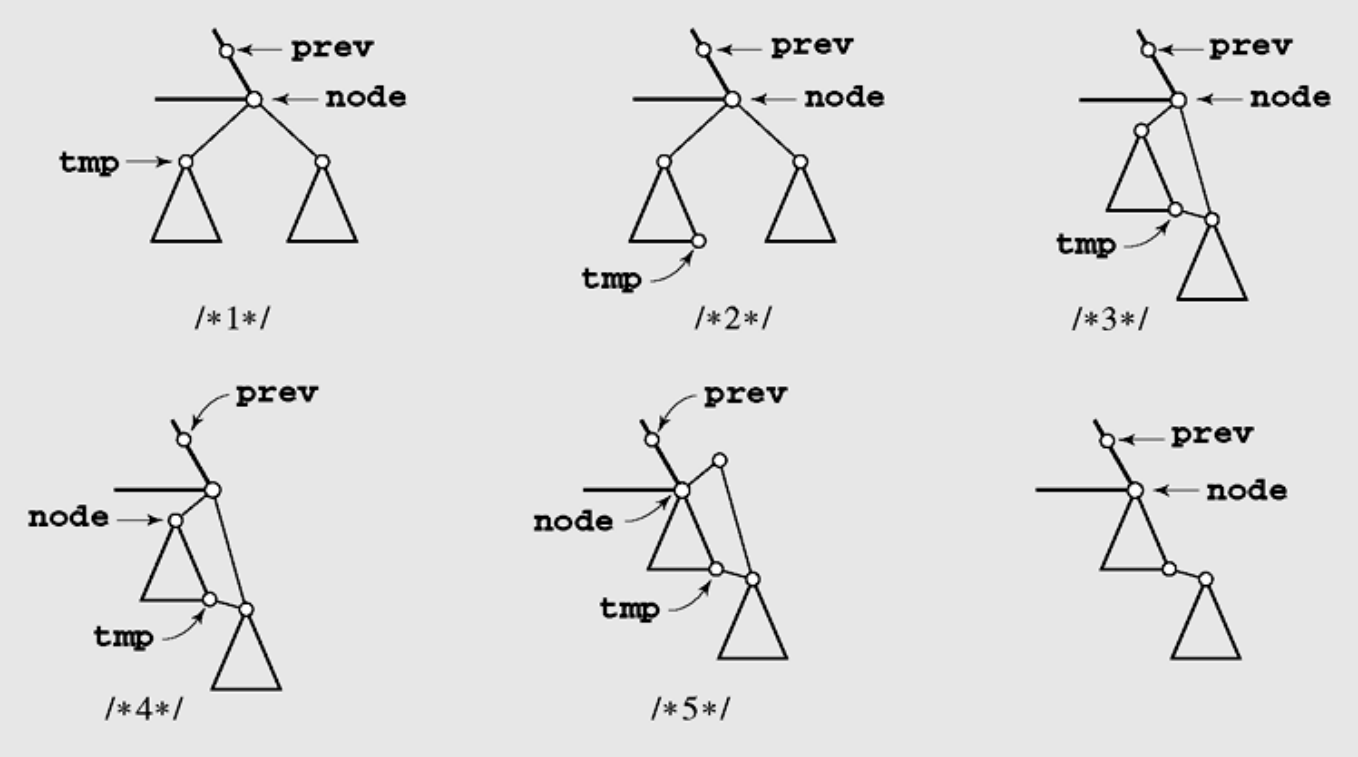


Figure 6-30 Details of deleting by merging

Deletion by Merging (continued)

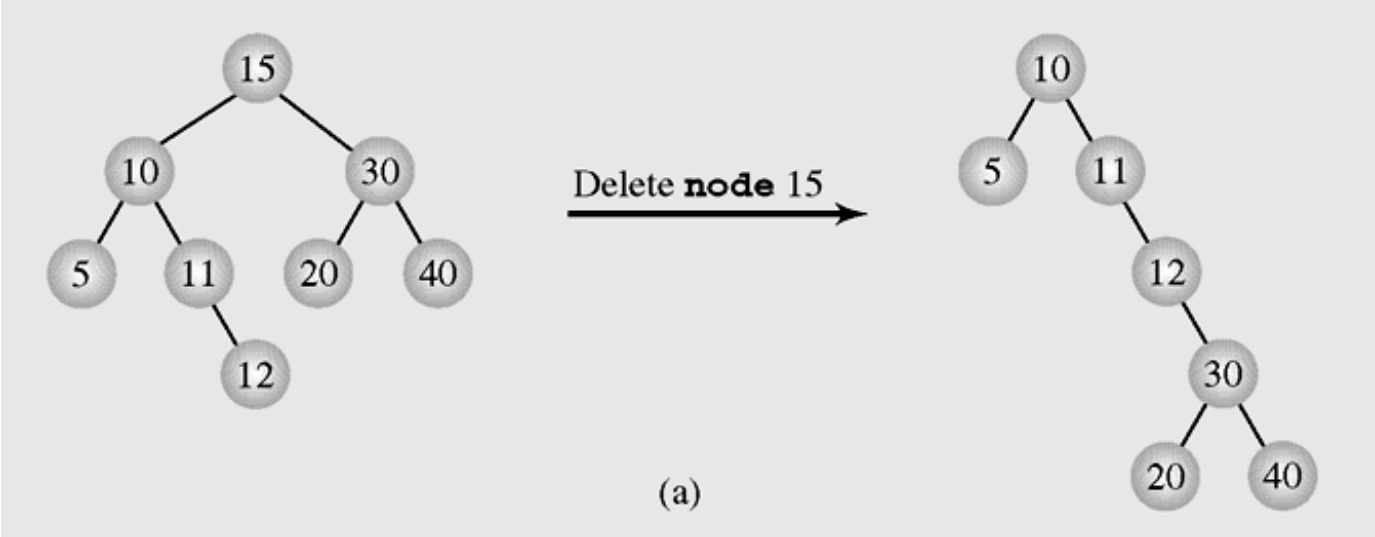


Figure 6-31 The height of a tree can be (a) extended or (b) reduced after deleting by merging

Deletion by Merging (continued)

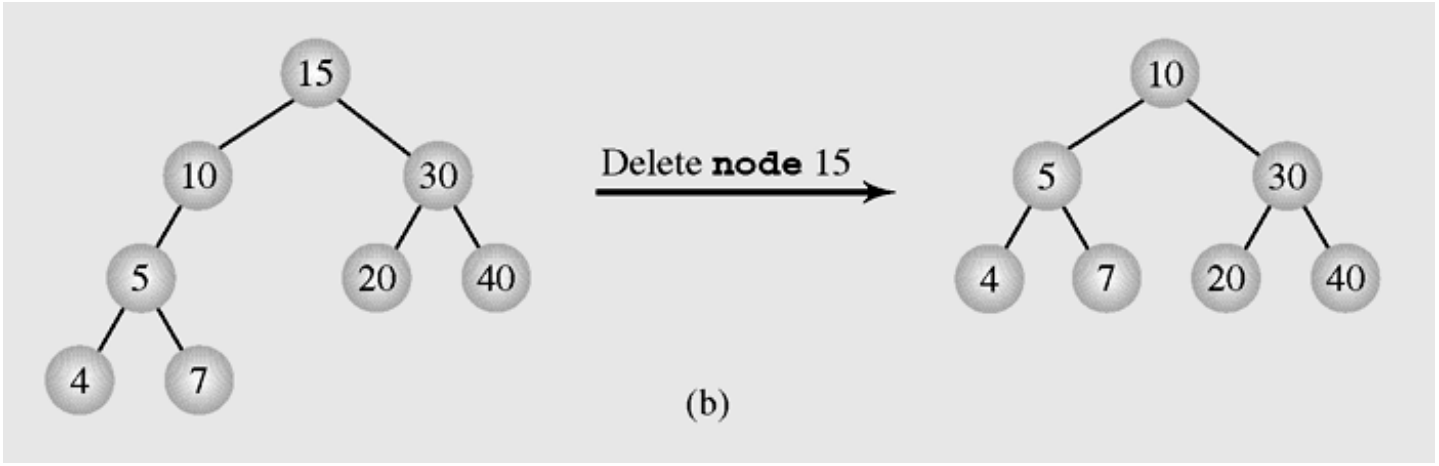


Figure 6-31 The height of a tree can be (a) extended or (b) reduced after deleting by merging (continued)

Deletion by Copying

- If the node has two children, the problem can be reduced to:
  - The node is a leaf
  - The node has only one nonempty child
- Solution: replace the key being deleted with its immediate predecessor (or successor)
- A key's predecessor is the key in the rightmost node in the left subtree

Deletion by Copying (continued)

```
public void deleteByCopying(T el) {
       BSTNode<T> node, p = root, prev = null;
       while (p != null && !p.el.equals(el)) {  // find the node p
            prev = p;                           // with element el;
            if (el.compareTo(p.el) < 0)
                 p = p.left;
            else p = p.right;
       }
```

Figure 6-32 Implementation of an algorithm for deleting by copying

```
    node = p;
    if (p != null && p.el.equals(el)) {
         if (node.right == null)              // node has no right child;
              node = node.left;
         else if (node.left == null)          // no left child for node;
              node = node.right;
         else {
              BSTNode<T> tmp = node.left;     // node has both children;
              BSTNode<T> previous = node;     // 1.
              while (tmp.right != null) {     // 2. find the rightmost
                  previous = tmp;             //    position in the
                  tmp = tmp.right;            //    left subtree of node;
              }
              node.el = tmp.el;               // 3. overwrite the reference
                                              //    to the element being deleted;
              if (previous == node)           // if node's left child's
                  previous.left  = tmp.left; // right subtree is null;
             else previous.right = tmp.left; // 4.
         }
         if (p == root)
              root = node;
         else if (prev.left == p)
              prev.left = node;
         else prev.right = node;
    }
    else if (root != null)
         System.out.println("el " + el + " is not in the tree");
    else System.out.println("the tree is empty");
}
```

Figure 6-32 Implementation of an algorithm for deleting by copying (continued)

Deletion by Copying (continued)

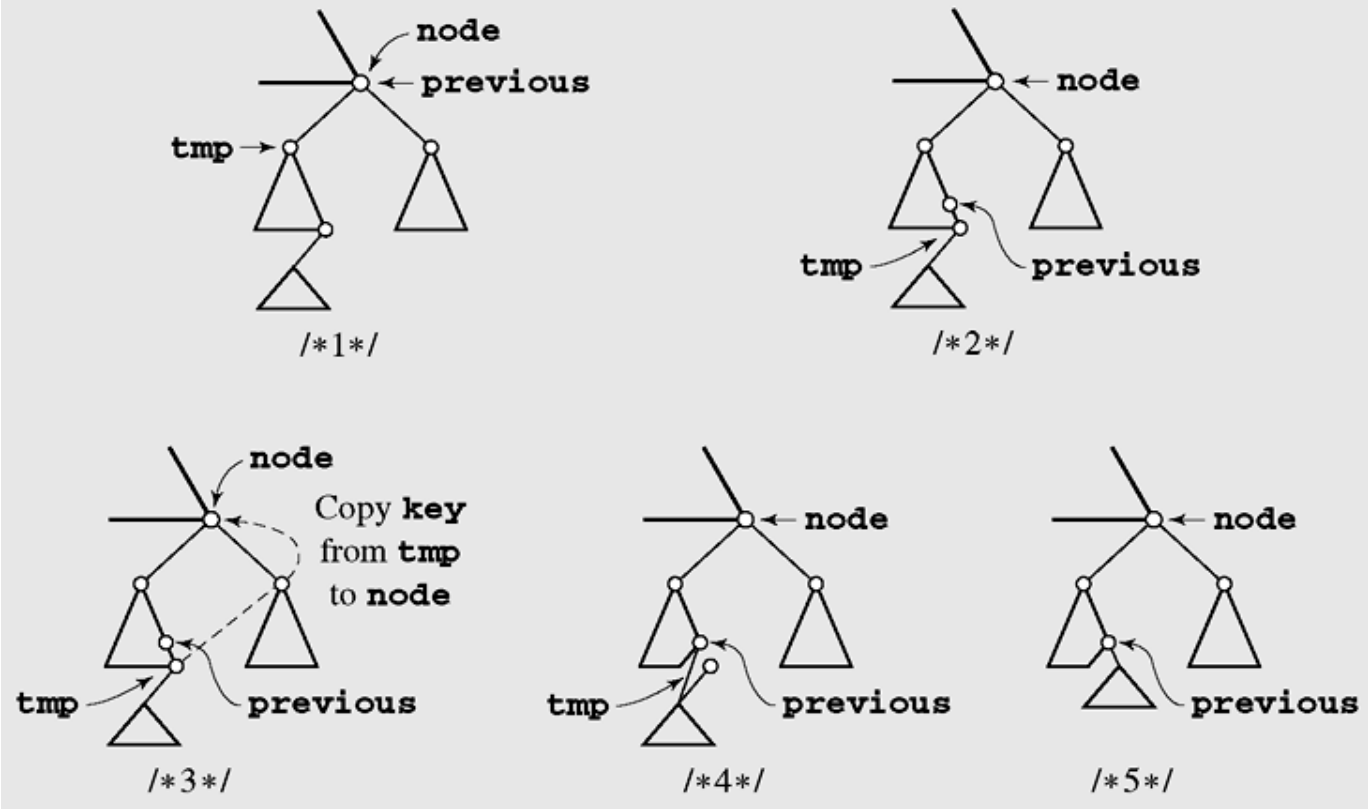


Figure 6-33 Deleting by copying