# 9-BINARY TREES

## Topics

- Why use Binary Trees ?
- What is a Tree ?
- Tree Terminology
- Binary Trees
  - Characteristics
  - Binary Search Trees
    - Finding a Node
    - Inserting a Node
    - Traversing a Tree
    - Deleting a Node

## Why use Binary Trees ?

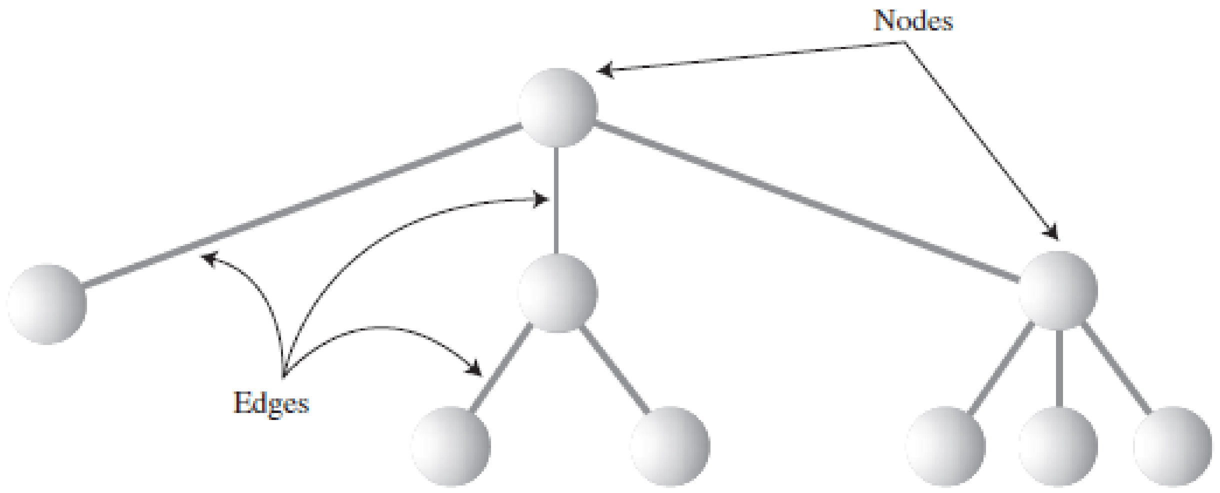
## Why use Binary Trees ?

- It combines the advantages of two other structures:
- An ordered array and a linked list.
  - You can search a tree quickly, as you can an ordered array,
  - and you can also insert and delete items quickly, as you can with a linked list.
- Avoids disadvantages of:
  - Slow insertion in an ordered array
  - Slow searching in a linked list

## What is a Tree ?

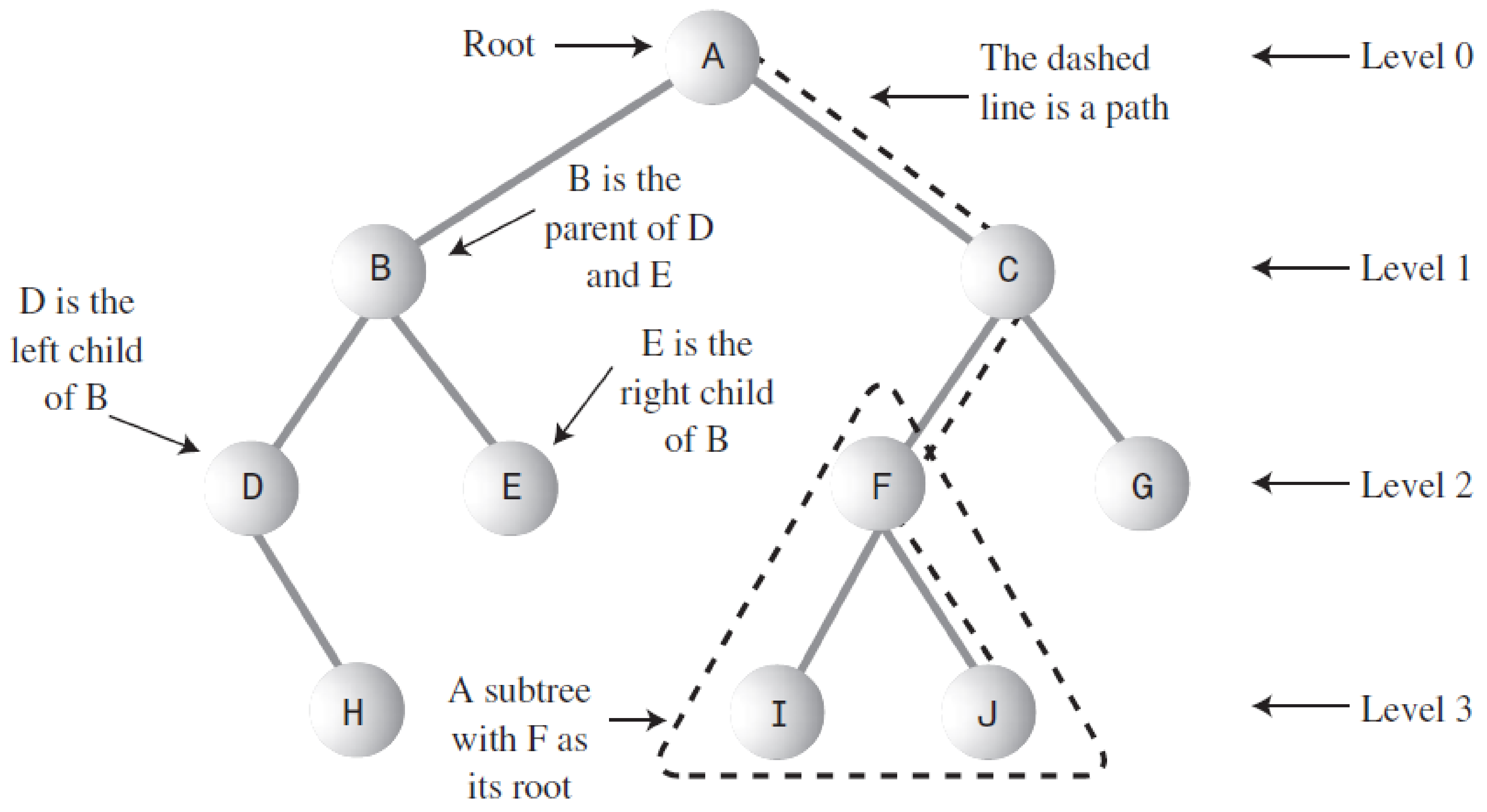
## What is a Tree ?

- A tree consists of *nodes* connected by *edges*.
  - Nodes often represent entities as people, car parts, airline reservations, etc.
  - The lines (edges) between the nodes represent the way the nodes are related.



## Tree Terminology

# Tree Terminology

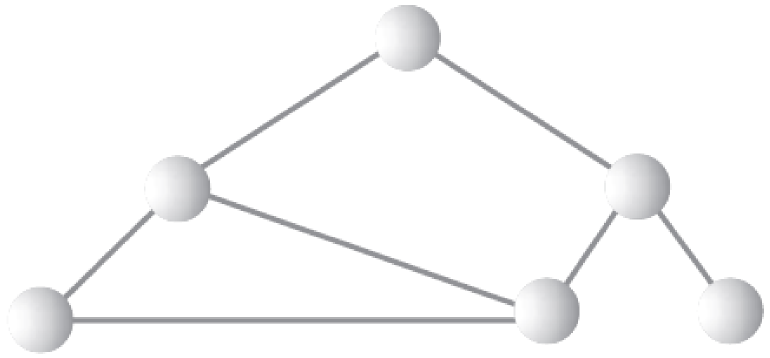

Root
A
The dashed line is a path
Level 0
B is the parent of D and E
B
C
Level 1
D is the left child of B
E is the right child of B
D
E
F
G
Level 2
H
A subtree with F as its root
I
J
Level 3

## Tree Terminology

- **Path**
  - Walking from node to node along the edges that connect them. The resulting sequence of nodes is called a *path*.
- **Root**
  - The node at the top of the tree is called the *root*. There is only one root in a tree.

## Tree Terminology

- For a collection of nodes and edges to be defined as a tree, there must be one (and only one!) path from the root to any other node.
- Is This a tree ?



- NON TREE !!

## Tree Terminology

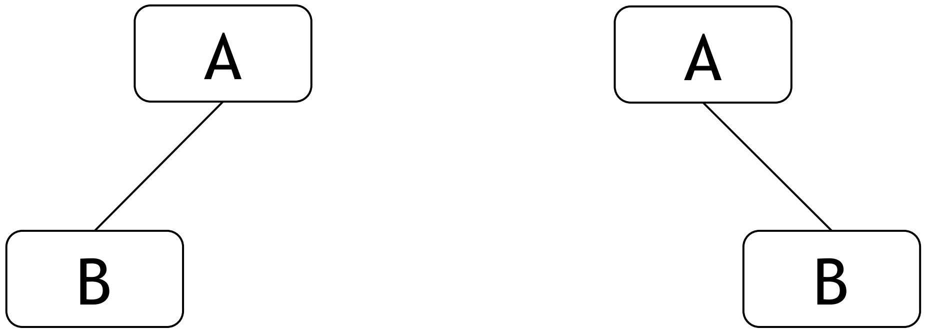
- **Parent**
  - Any node (except the root) has exactly one edge running upward to another node. The node above it is called the *parent* of the node.
- **Child**
  - Any node may have one or more lines running downward to other nodes. These nodes below a given node are called its *children*.
- **Leaf**
  - A node that has no children is called a *leaf node* or simply a *leaf*. There can be only one root in a tree, but there can be many leaves.

## More terminology

- Node A is the parent of node B if node B is a child of A
- Node A is an ancestor of node B if A is a parent of B, or if some child of A is an ancestor of B
  - In less formal terms, A is an ancestor of B if B is a child of A, or a child of a child of A, or a child of a child of a child of A, etc.
- Node B is a descendant of A if A is an ancestor of B
- Nodes A and B are siblings if they have the same parent

## Left ≠ Right

- The following two binary trees are *different*:



- In the first binary tree, node A has a left child but no right child; in the second, node A has a right child but no left child
- Put another way: Left and right are *not* relative terms

## Tree Terminology

- **Subtree**
  - Any node may be considered to be the root of a *subtree*, which consists of its children, and its children's children, and so on. If you think in terms of families, a node's subtree contains all its descendants.
- **Visiting**
  - A node is *visited* when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data fields or displaying it.
- **Traversing**
  - To *traverse* a tree means to visit all the nodes in some specified order.

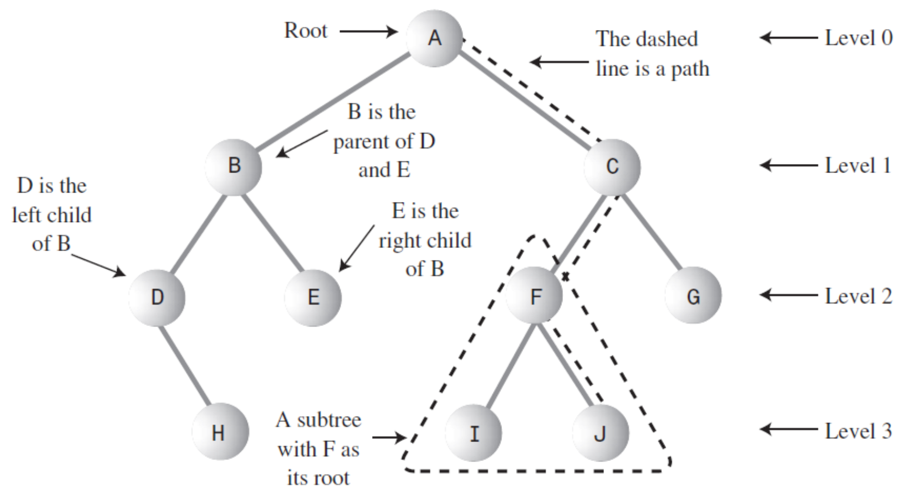
## Tree Terminology

- **Levels**
  - The *level* of a particular node refers to how many generations the node is from the root. If we assume the root is Level 0, then its children will be Level 1, its grandchildren will be Level 2, and so on.
- **Keys**
  - We' ve seen that one data field in an object is usually designated a *key value*. This value is used to search for the item or perform other operations on it. In tree diagrams, when a circle represents a node holding a data item, the key value of the item is typically shown in the circle. (We' ll see many figures later on that show nodes containing keys.)

## Binary Trees

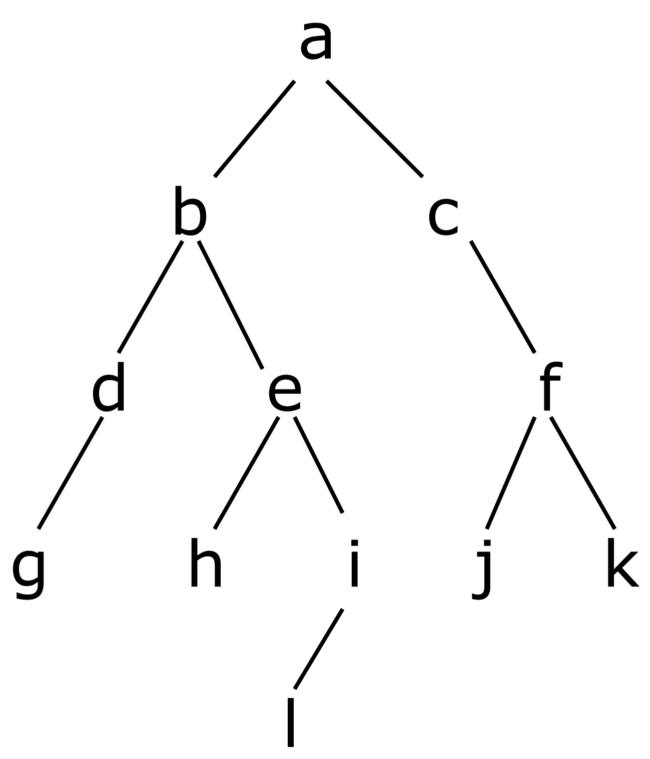
## Binary Trees

- If every node in a tree can have at most two children, the tree is called a *binary tree.*
- The two children of each node in a binary tree are called the *left child* and the *right child*


Root
A
The dashed line is a path
Level 0
B is the parent of D and E
B
C
Level 1
D is the left child of B
E is the right child of B
D
E
F
G
Level 2
H
A subtree with F as its root
I
J
Level 3

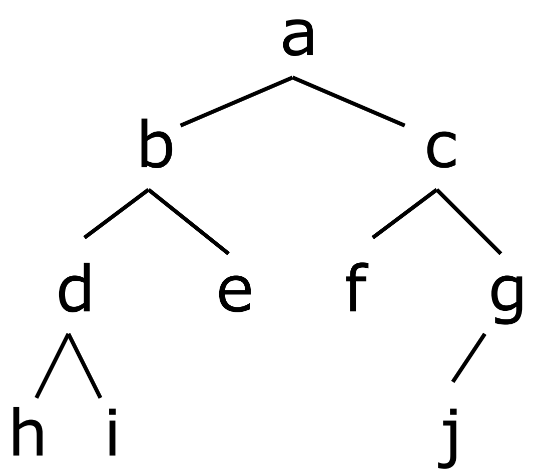
## Binary Trees - Characteristics

## Size and depth

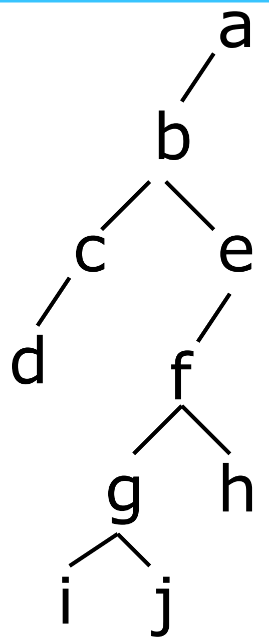


- The size of a binary tree is the number of nodes in it
  - This tree has size 12
- The depth of a node is its distance from the root
  - a is at depth zero
  - e is at depth 2
- The depth of a binary tree is the depth of its deepest node
  - This tree has depth 4

## Balance



A balanced binary tree



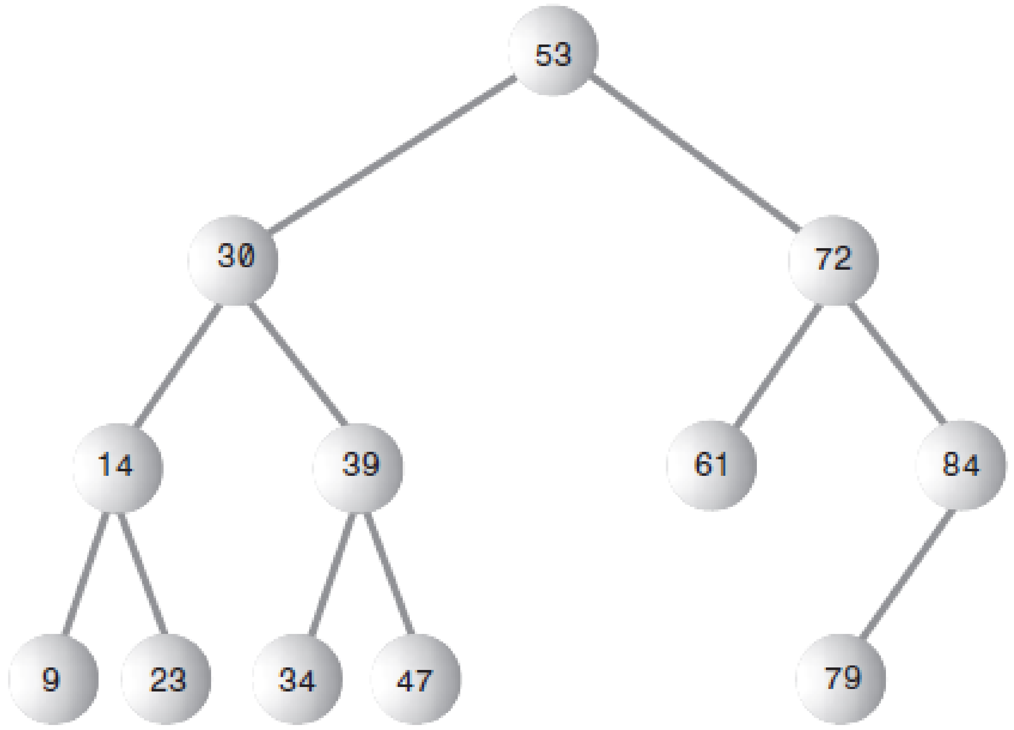
An unbalanced binary tree

- A binary tree is balanced if every level above the lowest is "full" (contains $2^n$ nodes, at the nth level)
- In most applications, a reasonably balanced binary tree is desirable

## Binary Search Trees

## Binary Search Trees

- The defining characteristic of a binary search tree is this:
  - A node's left child must have a key less than its parent,
  - and a node's right child must have a key greater than or equal to its parent.

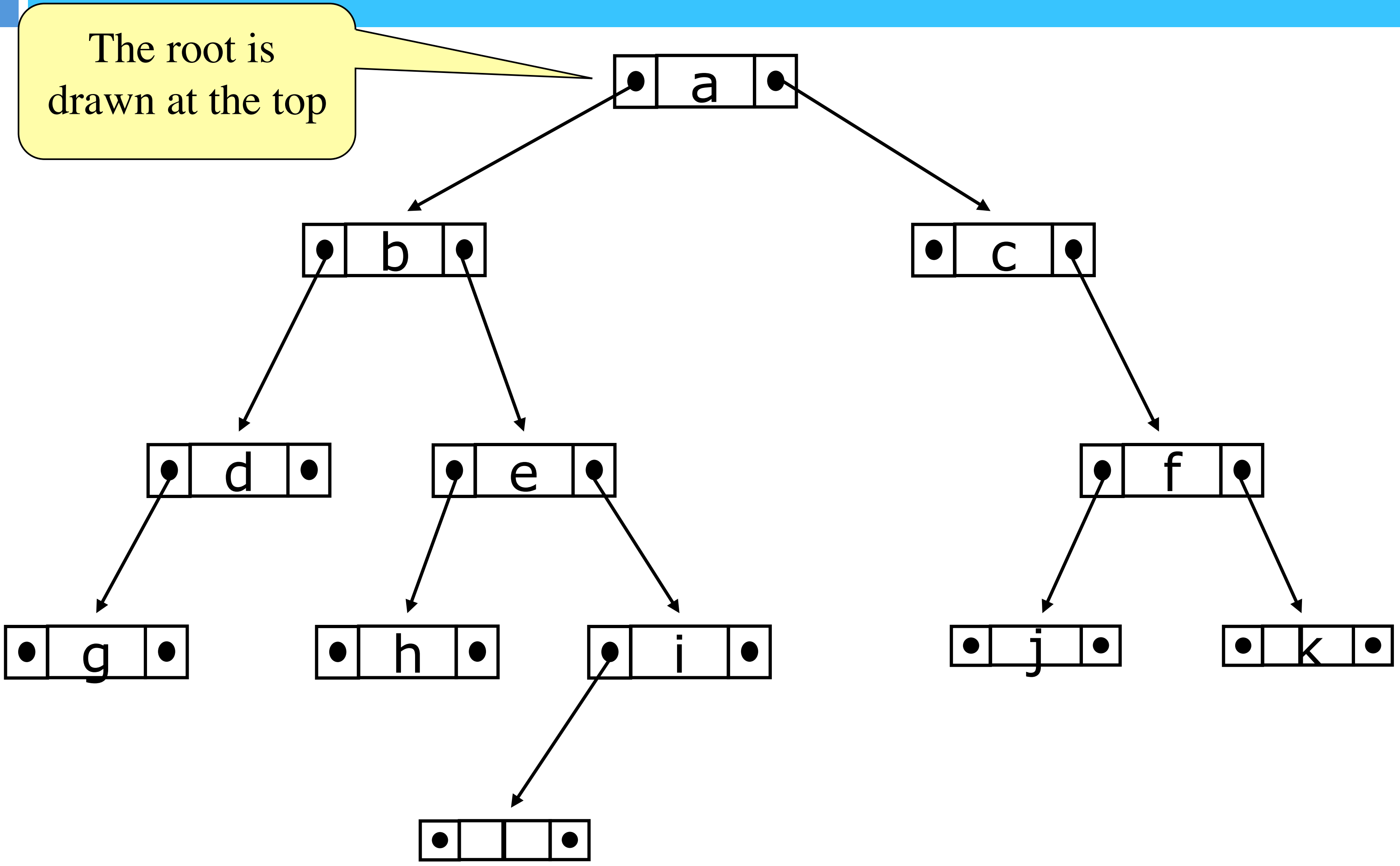


A binary search tree.

## How Binary Search Trees Work ?

- Common binary tree operations:
  - Finding a node with a given key
  - Inserting a new node
  - Traversing the tree
  - Deleting a node
- (See Binary Tree Workshop Applet )

## Binary Tree in Java Code



## Parts of a binary tree

- A binary tree is composed of zero or more nodes
  - In Java, a reference to a binary tree may be null
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be null), and
  - A reference or pointer to a right child (may be null)
- A binary tree may be *empty* (contain no nodes)
- If not empty, a binary tree has a root node
  - Every node in the binary tree is reachable from the root node by a *unique* path
- A node with no left child and no right child is called a leaf
  - In some binary trees, only the leaves contain a value

## Binary Tree Structure

- Here's a *recursive* Java representation for the Binary Tree data structure:

```
public class Node {
    public Object value;
    public Node leftChild;
    public Node rightChild;
    public Node(Object v,
                Node l,
                Node r)
    {value = v; leftChild = l; rightChild = r;}
}
```

## The Node class

- Here's a *recursive* Java representation for the Binary Tree data structure:

```
class Node
{
   Person p1;          // reference to person object
   Node leftChild; // this node's left child
   Node rightChild; // this node's right child
}
class Person
{
   int iData;
   double fData;
}
```

## The Tree class

```
class Tree
{
   private Node root; // the only data field in Tree

   public void find(int key) {   …  }
   public void insert(int id, double dd) {   …  }
   public void delete(int id) {   …  }
// various other methods
} // end class Tree
```

## The Treeapp class

- Listing 8.1, tree.java, page 405

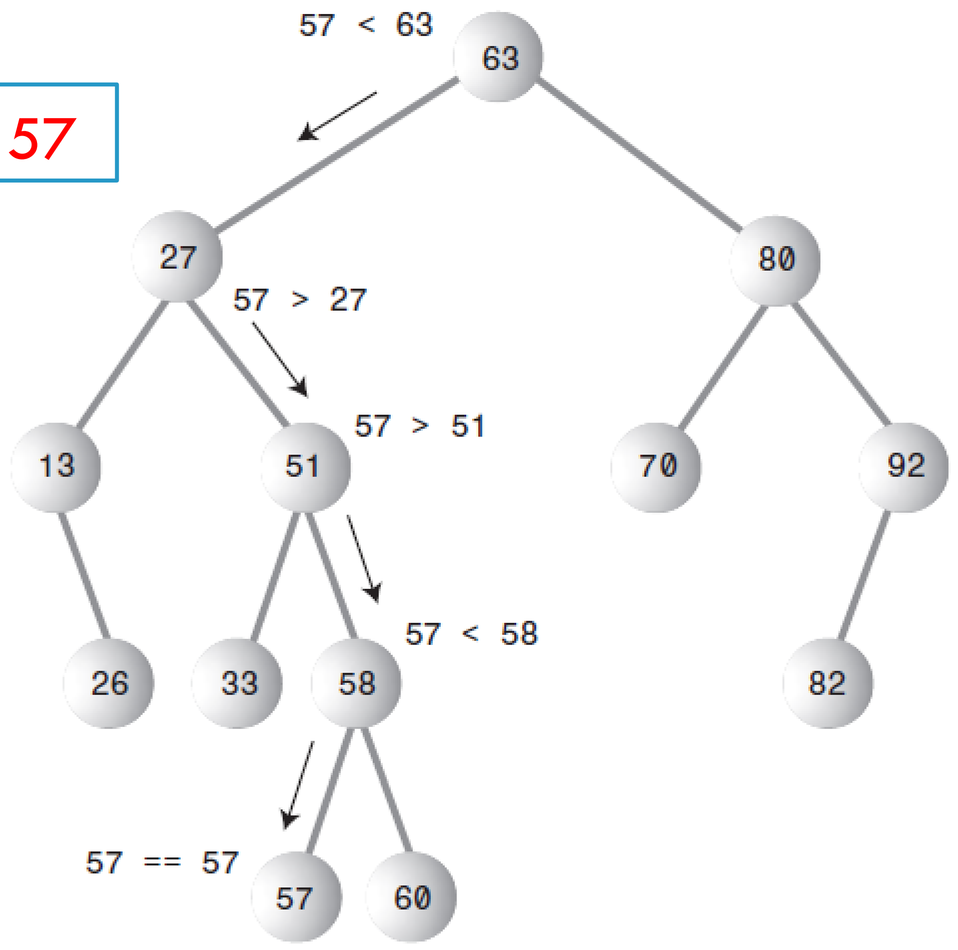
## Binary Trees – Finding a Node

## Finding a Node

- Finding a node with a specific key.
- Nodes are in a binary search tree, correspond to objects containing information.
  - Could be *person objects*, with an employee number as the key and also perhaps name, address, telephone number, salary, and other fields.
  - Or they could represent car parts, with a part number as the key value and fields for quantity on hand, price, and so on.

## Finding a Node

Finding node 57



Finding node 57.

## Java Code for Finding a Node

```
public Node find(int key)               // find node with given key
{  // (assumes non-empty tree)
   Node current = root;                 // start at root
   while ( current.iData != key ) {     // while no match,
      if(key < current.iData)           // go left?
         current = current.leftChild;
      else
         current = current.rightChild;  // or go right?
      if(current == null)               // if no child,
         return null;                   // didn’t find it
   }
   return current;                      // found it
}
```

## Efficiency of Finding a Node in Tree

- The time required to find a node depends on how many levels down it is situated
- O ( log N )   [Ideally if the tree is balanced]

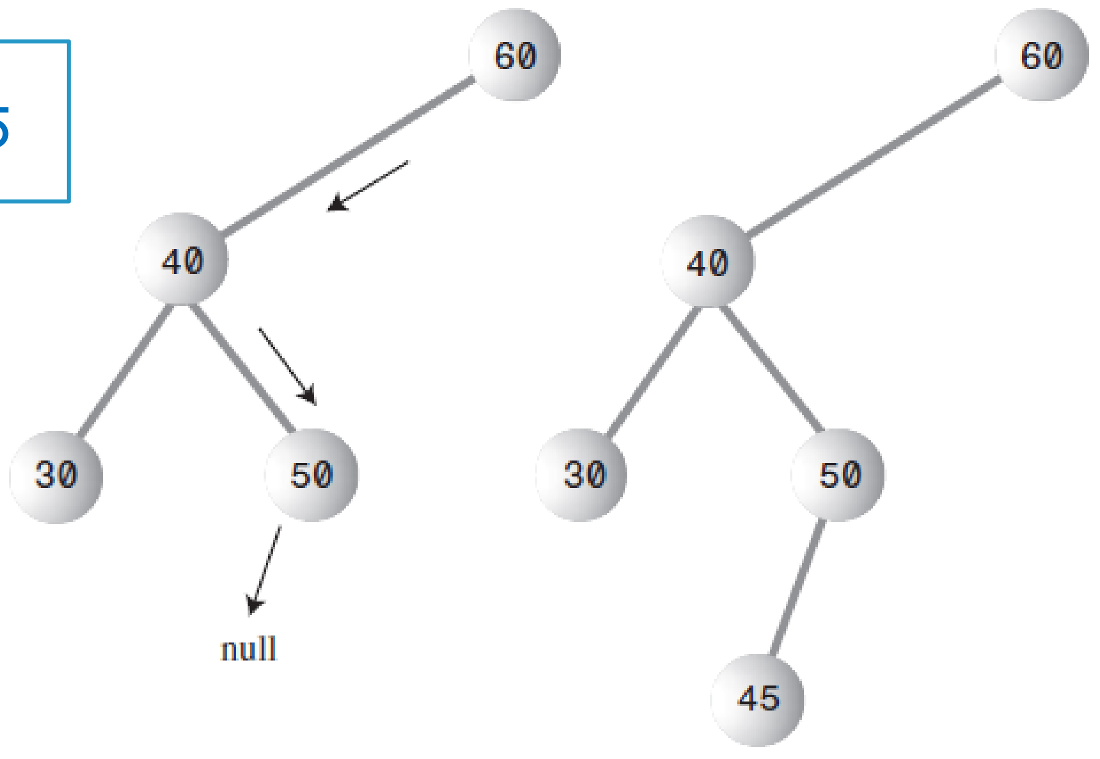
## Binary Trees – Inserting a Node

## Inserting a Node

- To insert a node, we must first find the place to insert it.
- We follow the path from the root to the appropriate node, which will be the parent of the new node.
- When this parent is found, the new node is connected as its left or right child, depending on whether the new node's key is less or greater than that of the parent.

# Inserting a Node

Insert 45



a) Before insertion

b) After insertion

Inserting a node.

# Java Code for Inserting a Node

```
public void insert (int id, double dd)   {
   // make new node
   Node newNode = new Node ( ) ;
  // no node in root
   if (root == null)   root = newNode;
   else  { // root occupied
      Node current = root; // start at root
      Node parent;
      while(true) { // (exits internally)
         parent = current;
         if(id < current.iData)  { // go left?
             current = current.leftChild;
             if(current == null) { // if end of the line,
                // insert on left
                parent.leftChild = newNode;
                return;
             }
      }
else { // or go right?
   current = current.rightChild;
   if(current == null)  { // if end of the line
      // insert on right
      parent.rightChild = newNode;
      return;
   }
} // end else go right
} // end while
} // end else not root
} // end insert()
```

## Binary Trees – Traversing a Tree

## Traversing a Tree

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive

## Traversing a Tree

- Since a binary tree has three “parts,” there are six possible ways to traverse the binary tree:
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root

## Inorder traversal – (1)

- In inorder, the root is visited *in the middle*
- The traversing method needs to do only three things:
  - 1. Call itself to traverse the node's left subtree.
  - 2. Visit the node.
  - 3. Call itself to traverse the node's right subtree.

## Inorder traversal – (1)

- Here's an inorder traversal to print out all the elements in the binary tree:

```
public void inorder(Node localRoot) {
    if (localRoot == null) return;

    inorder(localRoot.leftChild);
    System.out.println(localRoot.iData + " ");
    inorder(localRoot.rightChild);
}
```

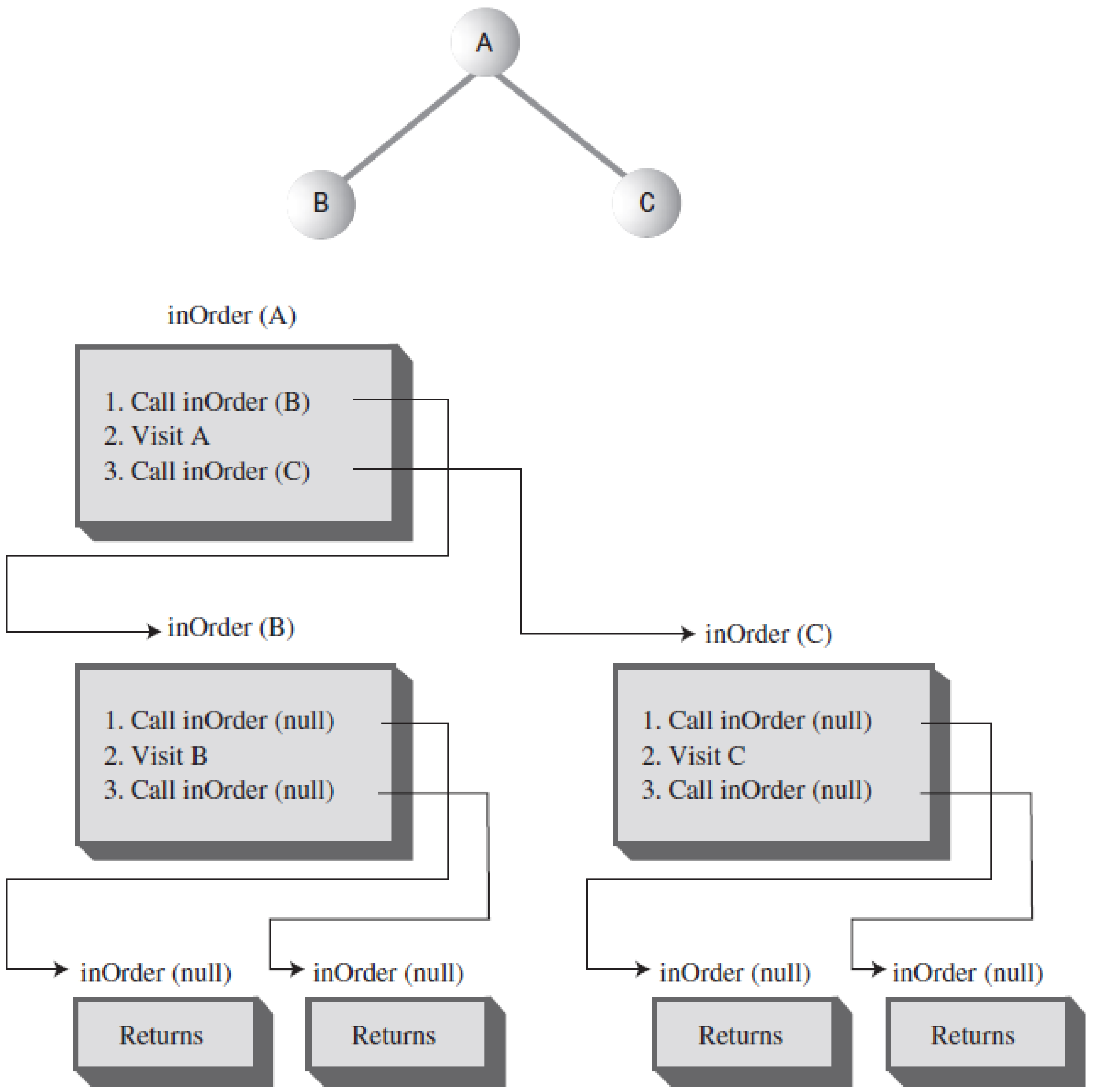
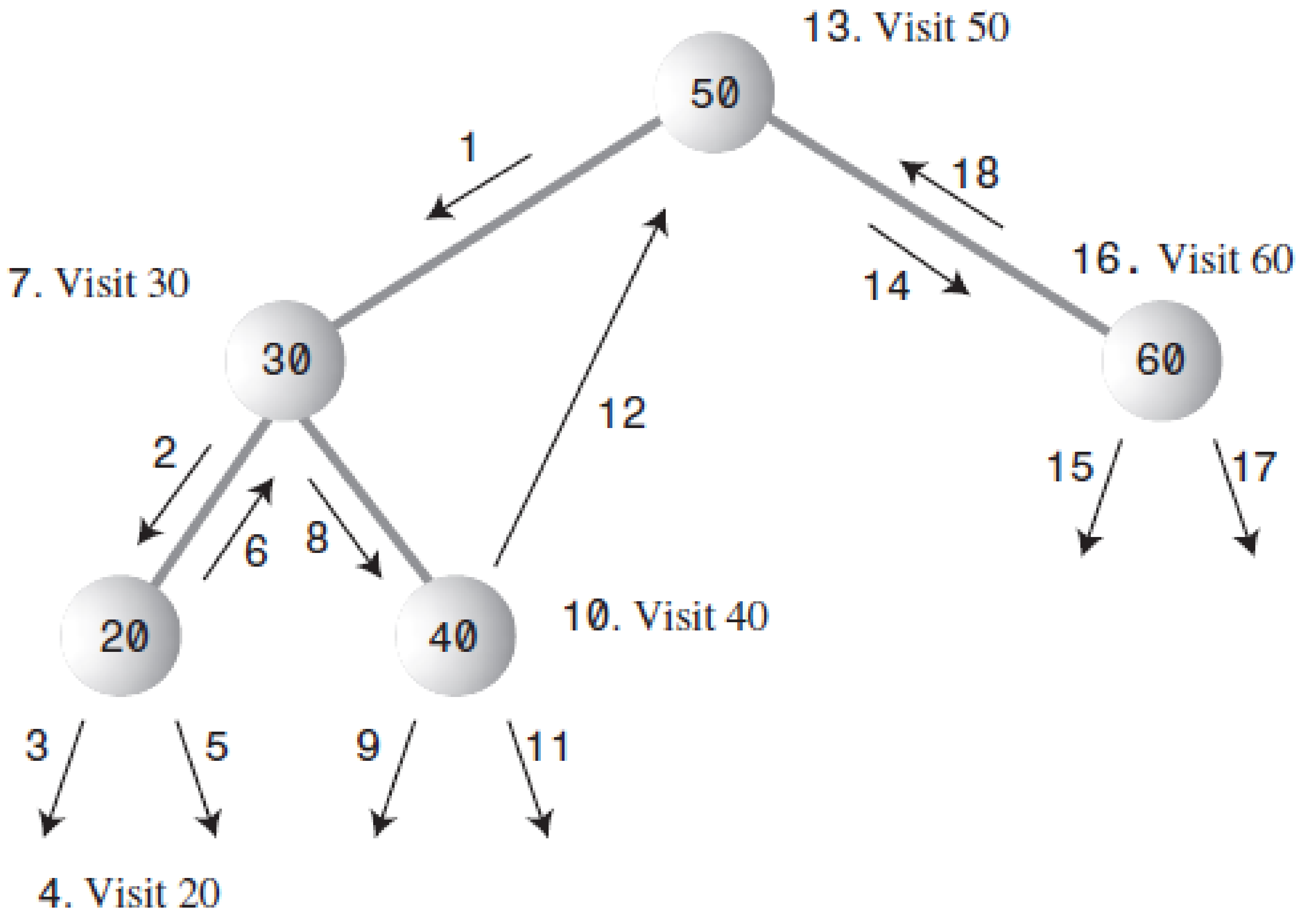


*FIGURE 8.9* The `inOrder()` method applied to a three-node tree.



Traversing a tree inorder.

## Preorder traversal – (1)

- In preorder, the root is visited *first*
- Here's the sequence for a preorder() method:
  - **1.** Visit the node.
  - **2.** Call itself to traverse the node's left subtree.
  - **3.** Call itself to traverse the node's right subtree.

## Preorder traversal – (2)

- Here's a preorder traversal to print out all the elements in the binary tree:

```
public void preorder(Node localRoot) {
    if (localRoot == null) return;

    System.out.println(localRoot.iData + " ");
    preorder(localRoot.leftChild);
    preorder(localRoot.rightChild);
}
```

## Postorder traversal – (1)

- In postorder, the root is visited *last*
- Here's the sequence for a postorder() method:
  - **1.** Call itself to traverse the node's left subtree.
  - **2.** Call itself to traverse the node's right subtree.
  - **3.** Visit the node.

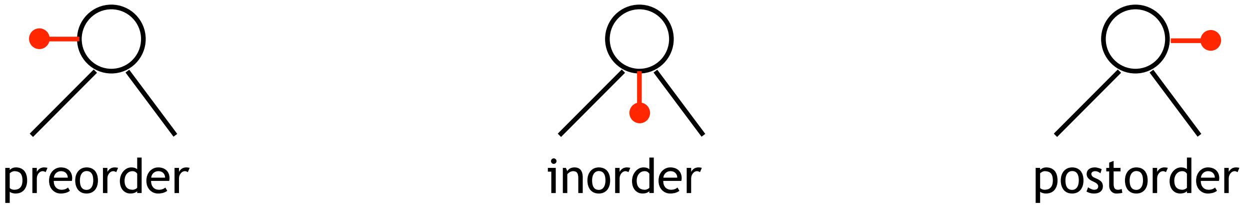
## Postorder traversal – (2)

- Here’s a postorder traversal to print out all the elements in the binary tree:

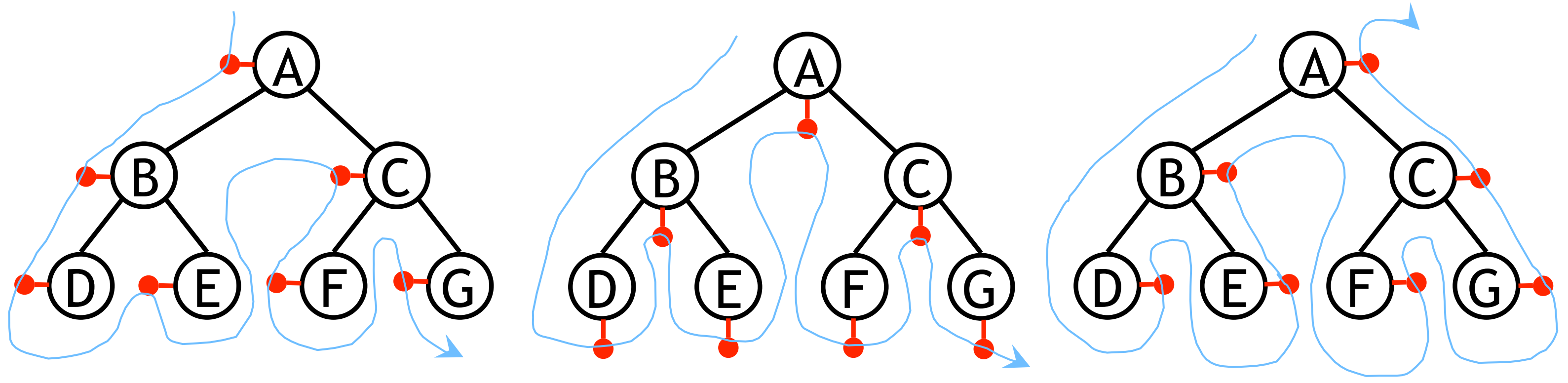
```
public void postorder(Node localRoot) {
    if (localRoot == null) return;

    postorder(localRoot.leftChild);
    postorder(localRoot.rightChild);
    System.out.println(localRoot.iData + " ");
}
```

## Tree traversals using "flags"

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:



- To traverse the tree, collect the flags:



A B D E C F G D B E A F C G D E B F G C A

## Copying a binary tree

- In postorder, the root is visited *last*
- Here’s a postorder traversal to make a complete copy of a given binary tree:

```
public Node copyTree(Node localRoot) {
    if (localRoot == null) return null;

    Node left = copyTree(localRoot.leftChild);
    Node right = copyTree(localRoot.rightChild);
// create Node and set its right and left children
    Node temp =  new Node(localRoot.iData);
    temp.leftChild = left;
    temp.rightchild = right;
    return temp;
}
```

## Other traversals

- The other traversals are the reverse of these three standard ones
  - That is, the right subtree is traversed before the left subtree is traversed
- Reverse preorder: root, right subtree, left subtree
- Reverse inorder: right subtree, root, left subtree
- Reverse postorder: right subtree, left subtree, root

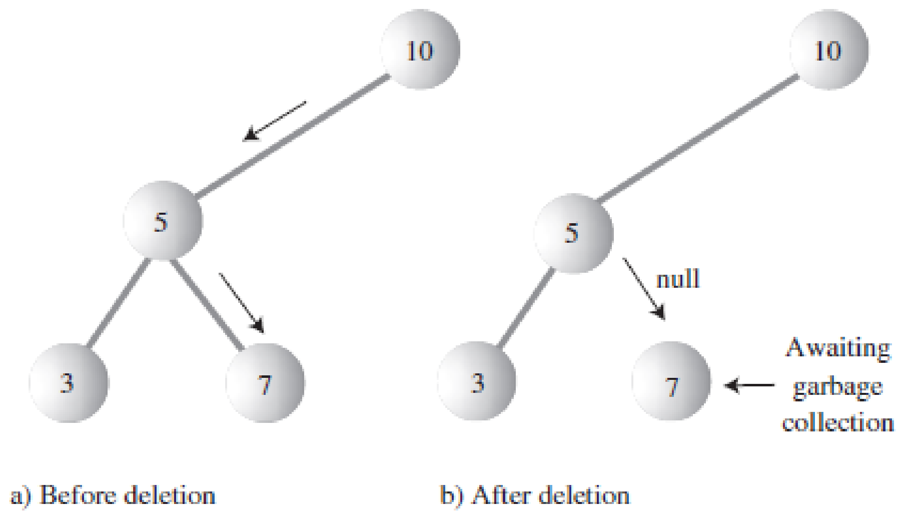
## Binary Trees – Deleting a Node

## Deleting a Node

- There are three cases to consider:
  - **1.** The node to be deleted is a leaf (has no children).
  - **2.** The node to be deleted has one child.
  - **3.** The node to be deleted has two children.

## Case 1: The Node to Be Deleted Has No Children

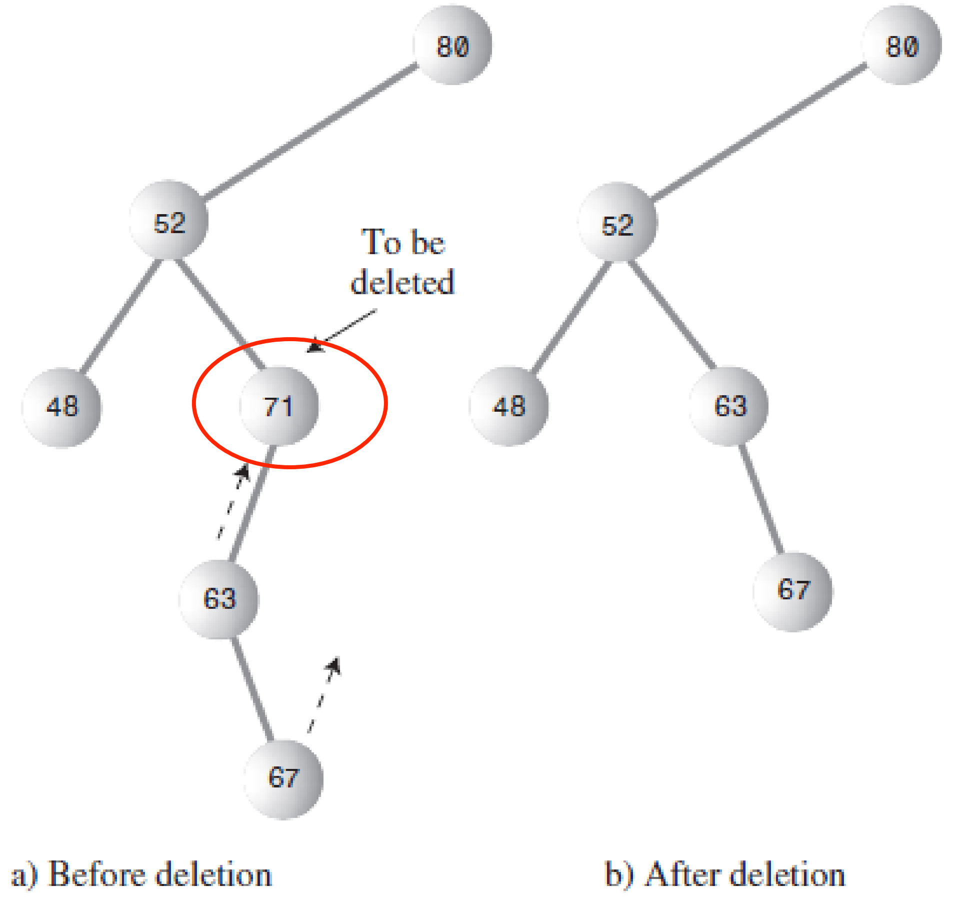
- To delete a leaf node, you simply change the appropriate child field in the node's parent to point to null, instead of to the node.
  - The node will still exist, but it will no longer be part of the tree.



Deleting a node with no children.

## Case 2: The Node to Be Deleted Has One Child

- The node has only two connections: to its parent and to its only child.
  - You want to "remove" the node out of this sequence by connecting its parent directly to its child.



Deleting a node with one child.

## Case 3: The Node to Be Deleted Has Two Children – (1)

- If the deleted node has two children, you can't just replace it with one of these children, at least if the child has its own children. Why not?

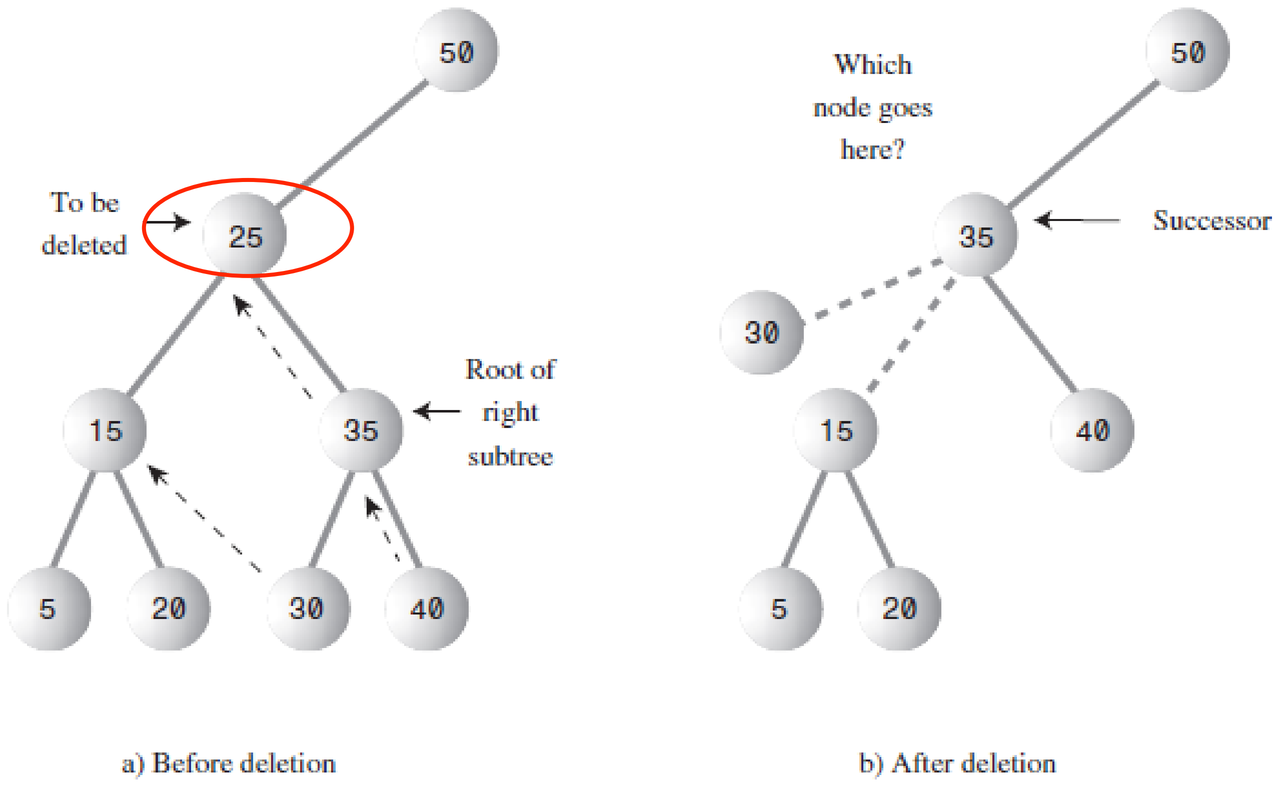


FIGURE 8.15 Cannot replace with subtree.

## Case 3: The Node to Be Deleted Has Two Children – (2)

- **Solution:** To delete a node with two children, *replace the node with its inorder successor*.

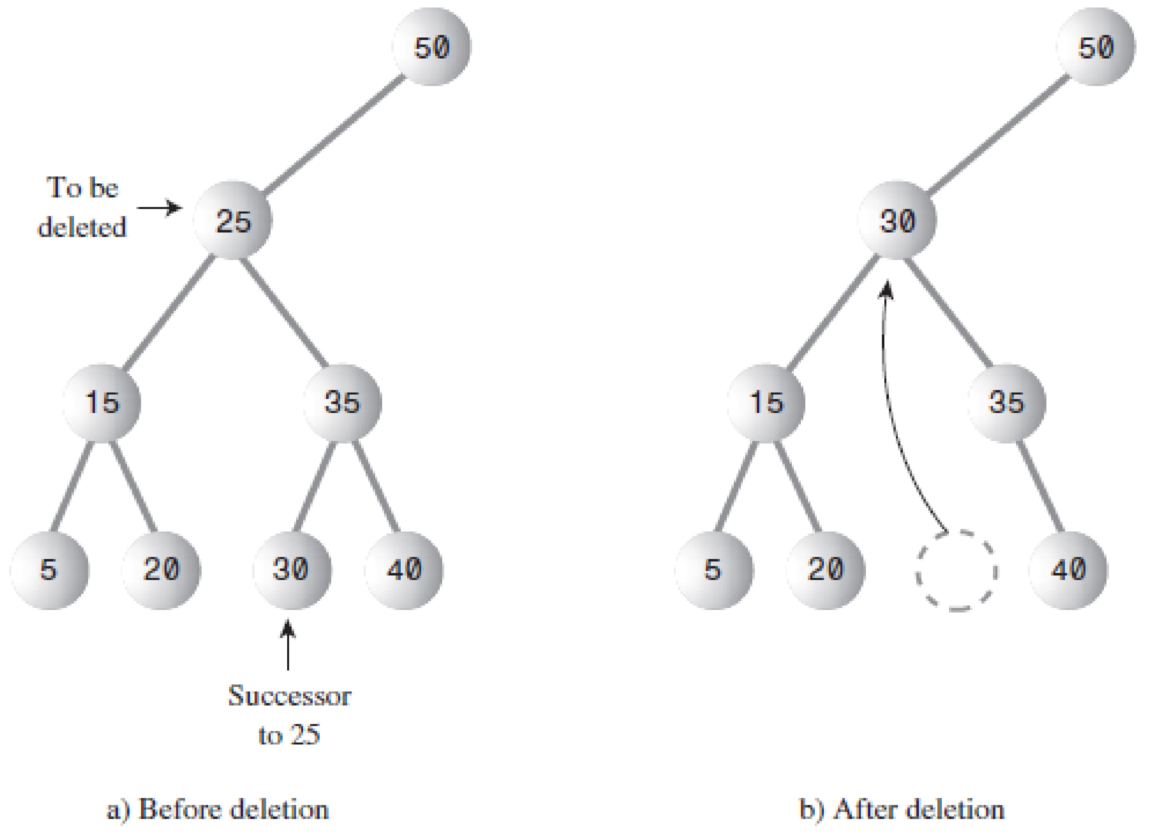
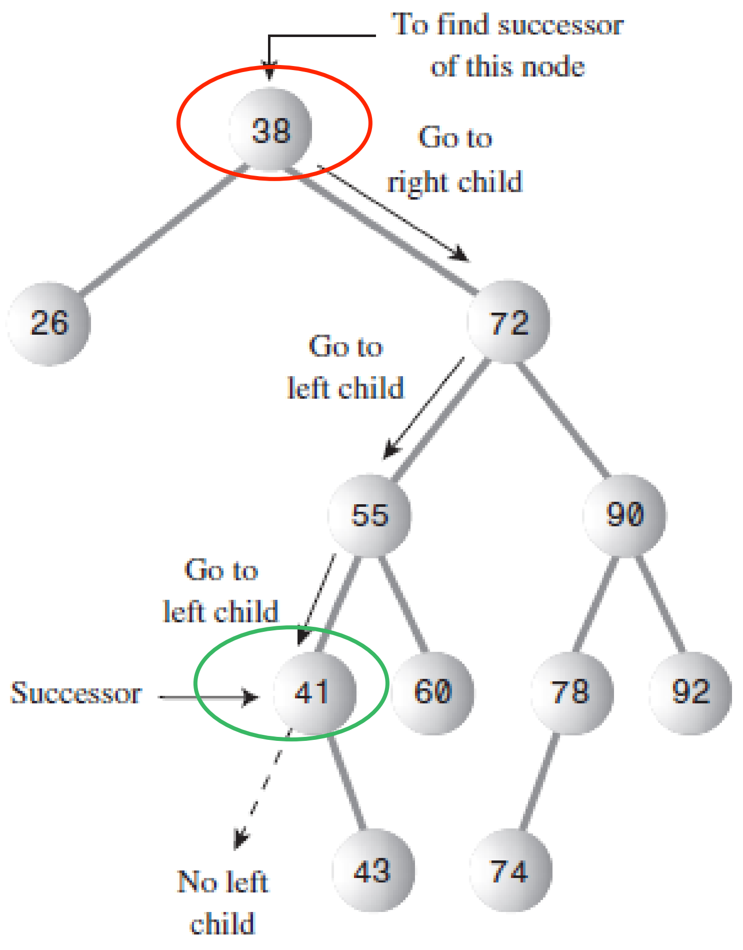


*FIGURE 8.16* Node replaced by its successor.

# Case 3: The Node to Be Deleted Has Two Children – (3)

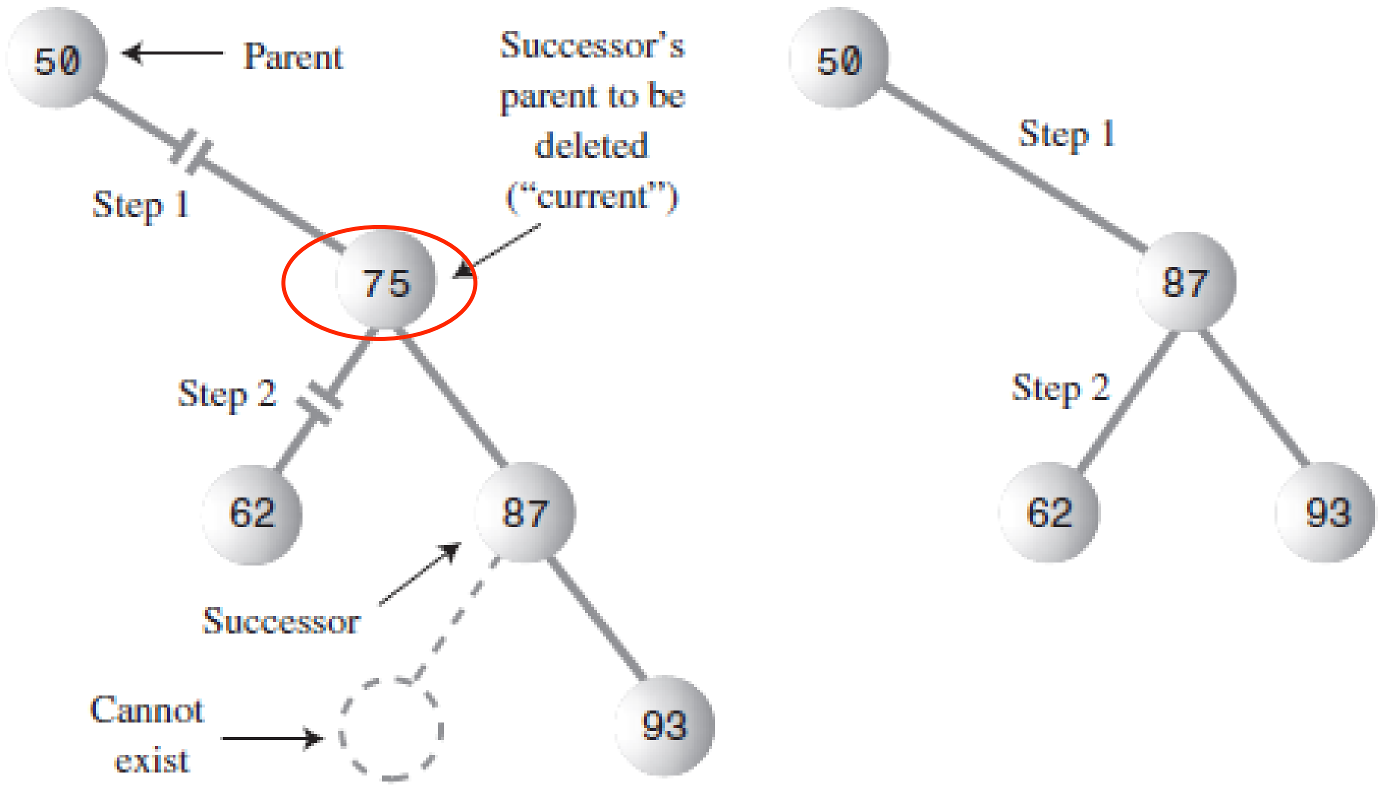
- **Finding a successor:**
- Go to the original node's right child, which must have a key larger than the node.
- Then it goes to this right child's left child (if it has one), and to this left child's left child, and so on.
- The last left child in this path is the successor of the original node



Finding the successor.

## Case 3: The Node to Be Deleted Has Two Children – (4)

- **Successor Is Right Child of** delNode



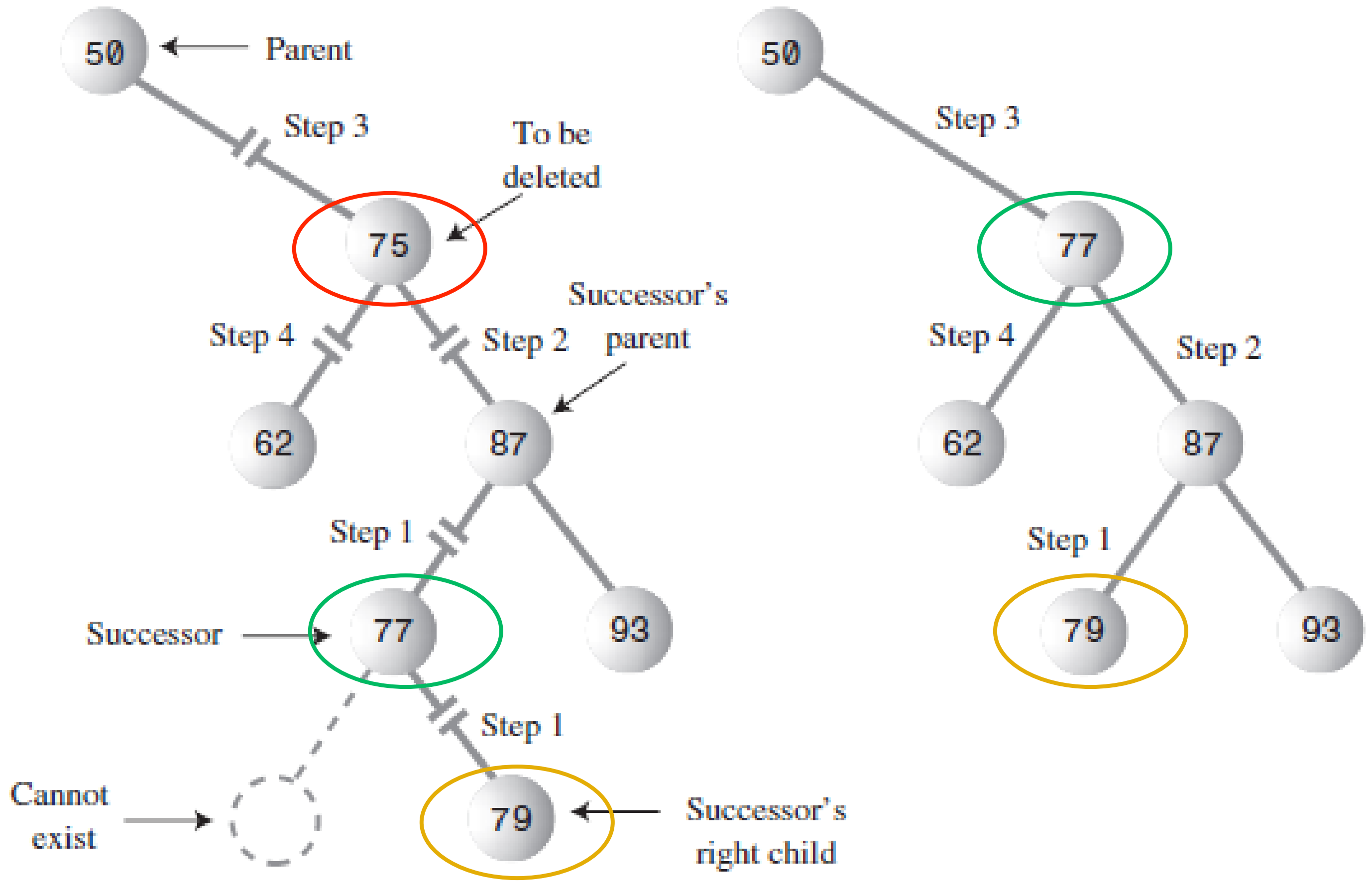
a) Before deletion b) After deletion

## Case 3: The Node to Be Deleted Has Two Children – (5)

- **Successor Is Right Child of** delNode
- This operation requires only two steps:
  - **1.** Unplug current from the rightChild field of its parent (or leftChild field if appropriate), and set this field to point to successor.
  - **2.** Unplug current's left child from current, and plug it into the leftChild field of successor.

## Case 3: The Node to Be Deleted Has Two Children – (6)

- Successor Is Left Descendant of Right Child of delNode



a) Before deletion

b) After deletion

## Case 3: The Node to Be Deleted Has Two Children – (7)

- **Successor Is Left Descendant of Right Child of** delNode
- Four steps are required to perform the deletion:
  - 1. Plug the right child of successor into the leftChild field of the successor's parent.
  - 2. Plug the right child of the node to be deleted into the rightChild field of successor.
  - 3. Unplug current from the rightChild field of its parent, and set this field to point to successor.
  - 4. Unplug current's left child from current, and plug it into the leftChild field of successor.

## The End