

# Tree Implementations

## Chapter 16

# Nodes in a Binary Tree

- Representing tree nodes
  - Must contain both data and “pointers” to node’s children
  - Each node will be an object
- Array-based
  - Pointers will be array indices
- Link-based
  - Use C++ pointers

# Array-Based Representation

- Class of array-based data members

```
TreeNode<ItemType> tree[MAX_NODES]; // Array of tree nodes
int root; // Index of root
int free; // Index of free list
```

- Variable **root** is index to tree's root node within the array tree
- If tree is empty, **root** = -1



# Array-Based Representation

- As tree changes (additions, removals) ...
  - Nodes may not be in contiguous array elements
- Thus, need list of available nodes
  - Called a free list
- Node removed from tree
  - Placed in free list for later use

# Array-Based Representation

```
template<class ItemType>
class TreeNode
{
private:
    ItemType  item;           // Data portion
    int       leftChild;     // Index to left child
    int       rightChild;    // Index to right child

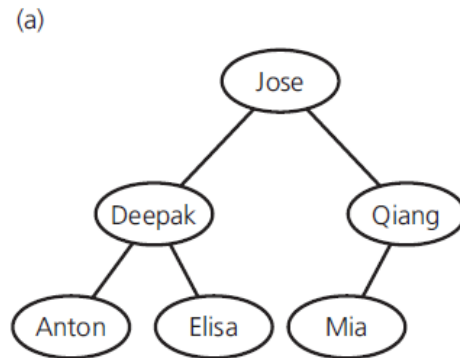
public:
    TreeNode();
    TreeNode(const ItemType& nodeItem, int left, int right);

    // Declarations of the methods setItem, getItem, setLeft, getLeft,
    // setRight, and getRight are here.

    . . .
}; // end TreeNode
```

LISTING 16-1 The class **TreeNode** for an array-based implementation of the ADT binary tree

# Array-Based Representation



(b)

	item	The array tree leftChild	rightChild	root
0	Jose	1	2	0
1	Deepak	3	4	free
2	Qiang	5	-1	6
3	Anton	-1	-1	
4	Elisa	-1	-1	
5	Mia	-1	-1	
6	?	-1	7	Free list
7	?	-1	8	
8	?	-1	9	
9	?	-1	-1	
•	•	•	•	
•	•	•	•	
•	•	•	•	

FIGURE 16-1 (a) A binary tree of names; (b) its implementation using the array **tree**



# Link-Based Representation

```
1  /** A class of nodes for a link-based binary tree.
2      @file BinaryNode.h */
3
4  #ifndef BINARY_NODE_
5  #define BINARY_NODE_
6  #include <memory>
7
8  template<class ItemType>
9  class BinaryNode
10 {
11 private:
12     ItemType                item;           // Data portion
13     std::shared_ptr<BinaryNode<ItemType>> leftChildPtr; // Pointer to left child
14     std::shared_ptr<BinaryNode<ItemType>> rightChildPtr; // Pointer to right child
15
16 public:
17     BinaryNode();
18     BinaryNode(const ItemType& anItem);
19     BinaryNode(const ItemType& anItem,
20                std::shared_ptr<BinaryNode<ItemType>> leftPtr,
21                std::shared_ptr<BinaryNode<ItemType>> rightPtr);
```

LISTING 16-2 The header file containing the class **BinaryNode** for a link-based implementation of the ADT binary tree

# Link-Based Representation

```
22
23     void setItem(const ItemType& anItem);
24     ItemType getItem() const;
25
26     bool isLeaf() const;
27
28     auto getLeftChildPtr() const;
29     auto getRightChildPtr() const;
30
31     void setLeftChildPtr(std::shared_ptr<BinaryNode<ItemType>> leftPtr);
32     void setRightChildPtr(std::shared_ptr<BinaryNode<ItemType>> rightPtr);
33 }; // end BinaryNode
34
35 #include "BinaryNode.cpp"
36 #endif
```

LISTING 16-2 The header file containing the class **BinaryNode** for a link-based implementation of the ADT binary tree



# Link-Based Representation

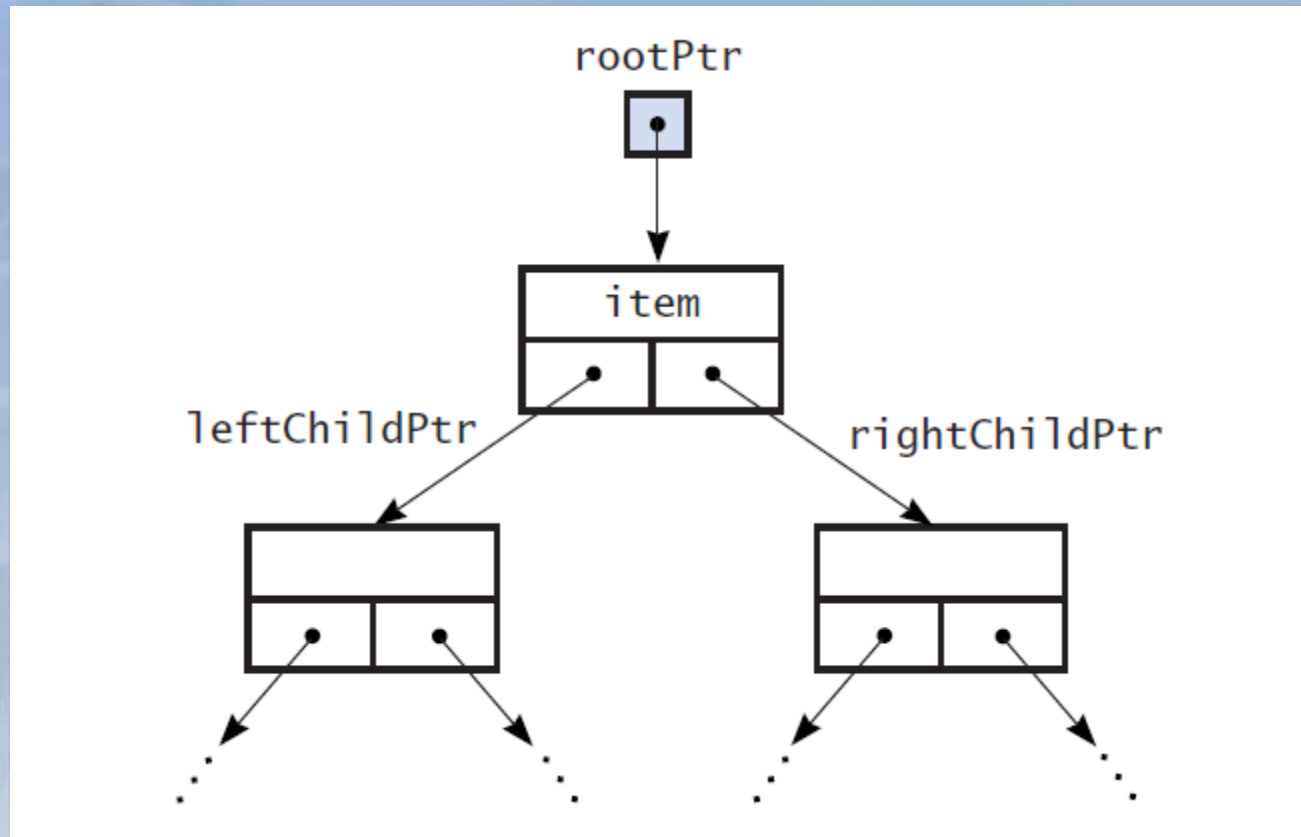


FIGURE 16-2 A link-based implementation of a binary tree

# The Header File

```
1  /** ADT binary tree: Link-based implementation.
2   * @file BinaryNodeTree.h */
3
4  #ifndef BINARY_NODE_TREE_
5  #define BINARY_NODE_TREE_
6
7  #include "BinaryTreeInterface.h"
8  #include "BinaryNode.h"
9  #include "PrecondViolatedExcept.h"
10 #include "NotFoundException.h"
11 #include <memory>
12
13 template<class ItemType>
14 class BinaryNodeTree : public BinaryTreeInterface<ItemType>
15 {
16 private:
17     std::shared_ptr<BinaryNode<ItemType>> rootPtr;
18 }
```

LISTING 16-3 A header file for the link-based implementation of the class **BinaryNodeTree**

# The Header File

```
18
19 protected:
20 //-----
21 //      Protected Utility Methods Section:
22 //      Recursive helper methods for the public methods.
23 //-----
24     int getHeightHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr) const;
25     int getNumberOfNodesHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr) const;
26
27     // Recursively adds a new node to the tree in a left/right fashion to keep tree balanced.
28     auto balancedAdd(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
29                     std::shared_ptr<BinaryNode<ItemType>> newNodePtr);
```

LISTING 16-3 A header file for the link-based implementation of the class **BinaryNodeTree**



# The Header File

```
30
31 // Removes the target value from the tree.
32 virtual auto removeValue(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
33                          const ItemType target, bool& isSuccessful);
34
35 // Copies values up the tree to overwrite value in current node until
36 // a leaf is reached; the leaf is then removed, since its value is stored in the parent.
37 auto moveValuesUpTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr);
38
39 // Recursively searches for target value.
40 virtual auto findNode(std::shared_ptr<BinaryNode<ItemType>> treePtr,
41                      const ItemType& target, bool& isSuccessful) const;
42
43 // Copies the tree rooted at treePtr and returns a pointer to the root of the copy.
44 auto copyTree(const std::shared_ptr<BinaryNode<ItemType>> oldTreeRootPtr) const;
45
46 // Recursively deletes all nodes from the tree.
47 void destroyTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr);
48
```

LISTING 16-3 A header file for the link-based implementation of the class **BinaryNodeTree**

# The Header File

```
48
49 // Recursive traversal helper methods:
50 void preorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;
51 void inorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;
52 void postorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;
53
54 public:
55 //-----
56 //      Constructor and Destructor Section.
57 //-----
58 BinaryNodeTree();
59 BinaryNodeTree(const ItemType& rootItem);
60 BinaryNodeTree(const ItemType& rootItem,
61               const std::shared_ptr<BinaryNodeTree<ItemType>> leftTreePtr,
62               const std::shared_ptr<BinaryNodeTree<ItemType>> rightTreePtr);
63 BinaryNodeTree(const std::shared_ptr<BinaryNodeTree<ItemType>>& tree);
64 virtual ~BinaryNodeTree();
65
```

LISTING 16-3 A header file for the link-based implementation of the class **BinaryNodeTree**



# The Header File

```
65
66 //-----
67 //      Public BinaryTreeInterface Methods Section.
68 //-----
69 bool isEmpty() const;
70 int getHeight() const;
71 int getNumberOfNodes() const;
72 ItemType getRootData() const throw (PrecondViolatedExcept);
73 void setRootData(const ItemType& newData);
74 bool add(const ItemType& newData); // Adds an item to the tree
75 bool remove(const ItemType& data); // Removes specified item from the tree
76 void clear();
77 ItemType getEntry(const ItemType& anEntry) const throw (NotFoundException);
78 bool contains(const ItemType& anEntry) const;
```

LISTING 16-3 A header file for the link-based implementation of the class **BinaryNodeTree**



# The Header File

```
79
80 //-----
81 //      Public Traversals Section.
82 //-----
83     void preorderTraverse(void visit(ItemType&)) const;
84     void inorderTraverse(void visit(ItemType&)) const;
85     void postorderTraverse(void visit(ItemType&)) const;
86
87 //-----
88 //      Overloaded Operator Section.
89 //-----
90     BinaryNodeTree& operator=(const BinaryNodeTree& rightHandSide);
91 }; // end BinaryNodeTree
92
93 #include "BinaryNodeTree.cpp"
94 #endif
```

LISTING 16-3 A header file for the link-based implementation of the class **BinaryNodeTree**

# The Implementation

```
template<class ItemType>
BinaryNodeTree<ItemType>::BinaryNodeTree() : rootPtr(nullptr)
{
} // end default constructor

template<class ItemType>
BinaryNodeTree<ItemType>::
BinaryNodeTree(const ItemType& rootItem)
    :rootPtr(std::make_shared<BinaryNode<ItemType>>(rootItem, nullptr, nullptr))
{
} // end constructor
```

## Constructors



# The Implementation

```
template<class ItemType>
BinaryNodeTree<ItemType>::
BinaryNodeTree(const ItemType& rootItem,
               const std::shared_ptr<BinaryNodeTree<ItemType>> leftTreePtr,
               const std::shared_ptr<BinaryNodeTree<ItemType>> rightTreePtr)
    :rootPtr(std::make_shared<BinaryNode<ItemType>>(rootItem,
                                                    copyTree(leftTreePtr->rootPtr),
                                                    copyTree(rightTreePtr->rootPtr)))
{
} // end constructor
```

## Constructors



# The Implementation

```
template<class ItemType>
std::shared_ptr<BinaryNode<ItemType>> BinaryNodeTree<ItemType>::copyTree(
    const std::shared_ptr<BinaryNode<ItemType>> oldTreeRootPtr) const
{
    std::shared_ptr<BinaryNode<ItemType>> newTreePtr;

    // Copy tree nodes during a preorder traversal
    if (oldTreeRootPtr != nullptr)
    {
        // Copy node
        newTreePtr = std::make_shared<BinaryNode<ItemType>>(oldTreeRootPtr->getItem(),
                                                            nullptr, nullptr);
        newTreePtr->setLeftChildPtr(copyTree(oldTreeRootPtr->getLeftChildPtr()));
        newTreePtr->setRightChildPtr(copyTree(oldTreeRootPtr->getRightChildPtr()));
    } // end if
    // Else tree is empty (newTreePtr is nullptr)

    return newTreePtr;
} // end copyTree
```

Protected method `copyTree` called by copy constructor

# The Implementation

```
template<class ItemType>
BinaryNodeTree<ItemType>::
    BinaryNodeTree(const BinaryNodeTree<ItemType>& treePtr)
{
    rootPtr = copyTree(treePtr.rootPtr);
} // end copy constructor
```

Copy constructor



# The Implementation

```
template<class ItemType>
void BinaryNodeTree<ItemType>::
    destroyTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr)
{
    if (subTreePtr != nullptr)
    {
        destroyTree(subTreePtr->getLeftChildPtr());
        destroyTree(subTreePtr->getRightChildPtr());
        subTreePtr.reset(); // Decrement reference count to node
    } // end if
} // end destroyTree
```

**destroyTree** used by destructor which simply calls this method



# The Implementation

```
template<class ItemType>
int BinaryNodeTree<ItemType>::
    getHeightHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr) const
{
    if (subTreePtr == nullptr)
        return 0;
    else
        return 1 + max(getHeightHelper(subTreePtr->getLeftChildPtr()),
                        getHeightHelper(subTreePtr->getRightChildPtr()));
} // end getHeightHelper
```

Protected method `getHeightHelper`

# The Implementation

```
template<class ItemType>
bool BinaryNodeTree<ItemType>::add(const ItemType& newData)
{
    auto newNodePtr = std::make_shared<BinaryNode<ItemType>>(newData);
    rootPtr = balancedAdd(rootPtr, newNodePtr);

    return true;
} // end add
```

Method **add**

# The Implementation

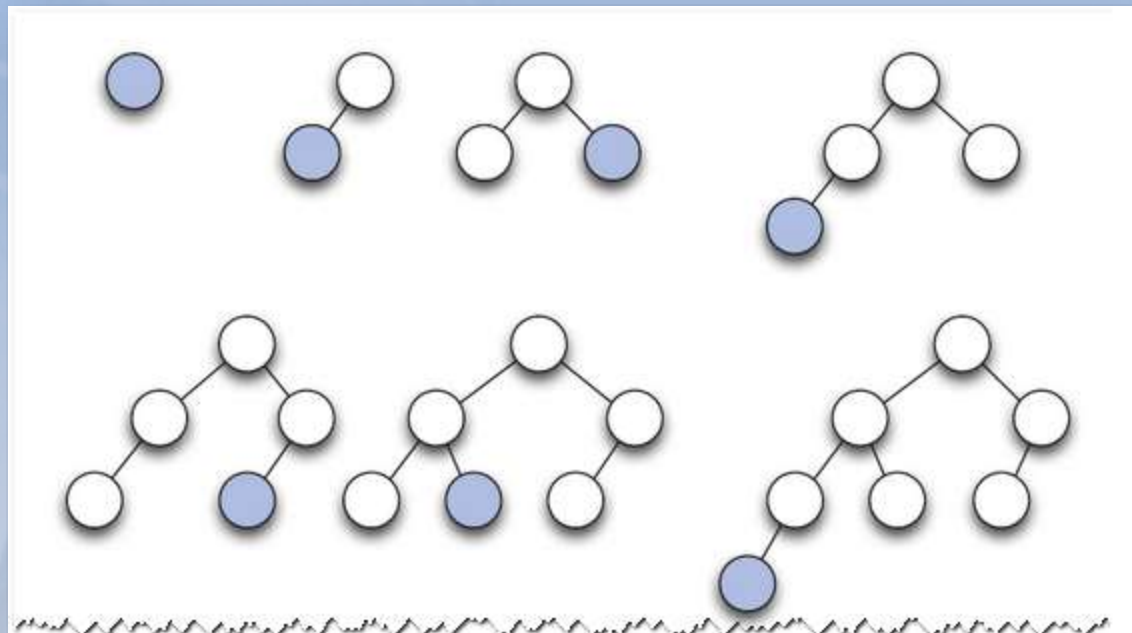


FIGURE 16-3 Adding nodes to an initially empty binary tree



# The Implementation

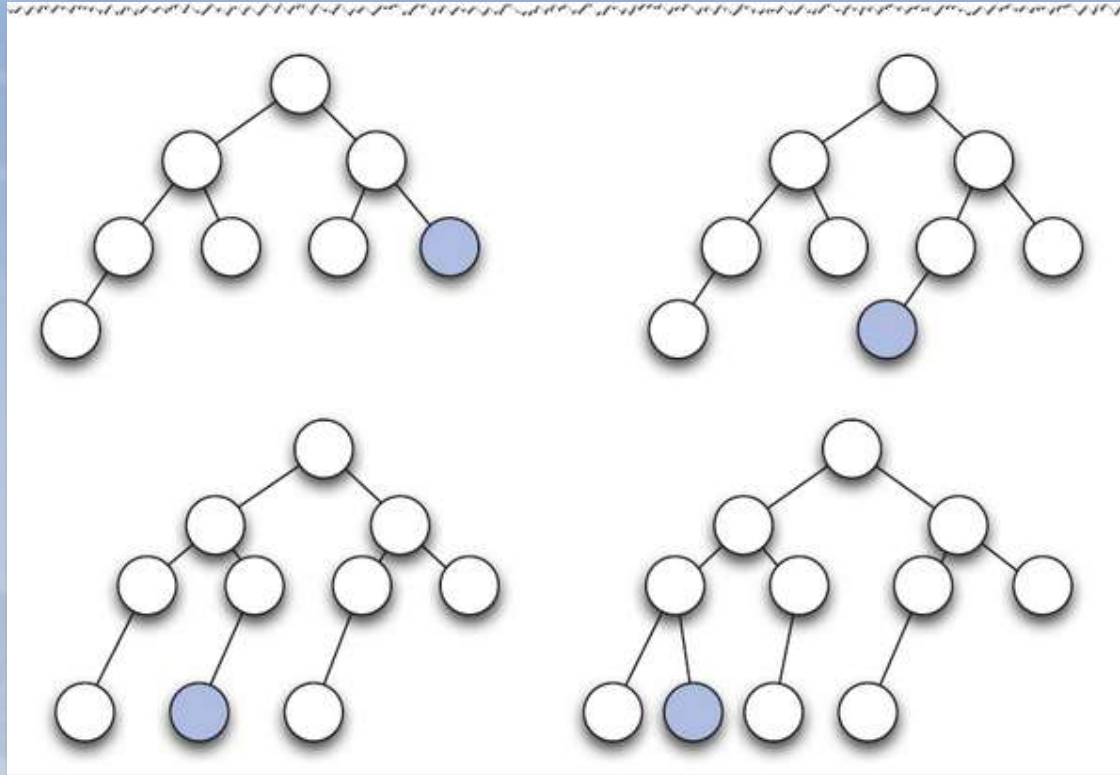


FIGURE 16-3 Adding nodes to an initially empty binary tree

# The Implementation

```
template<class ItemType>
void BinaryNodeTree<ItemType>::
    inorder(void visit(ItemType&),
            std::shared_ptr<BinaryNode<ItemType>> treePtr) const
{
    if (treePtr != nullptr)
    {
        inorder(visit, treePtr->getLeftChildPtr());
        ItemType theItem = treePtr->getItem();
        visit(theItem);
        inorder(visit, treePtr->getRightChildPtr());
    } // end if
} // end inorder
```

Protected method that enables recursive traversals.



# The Implementation

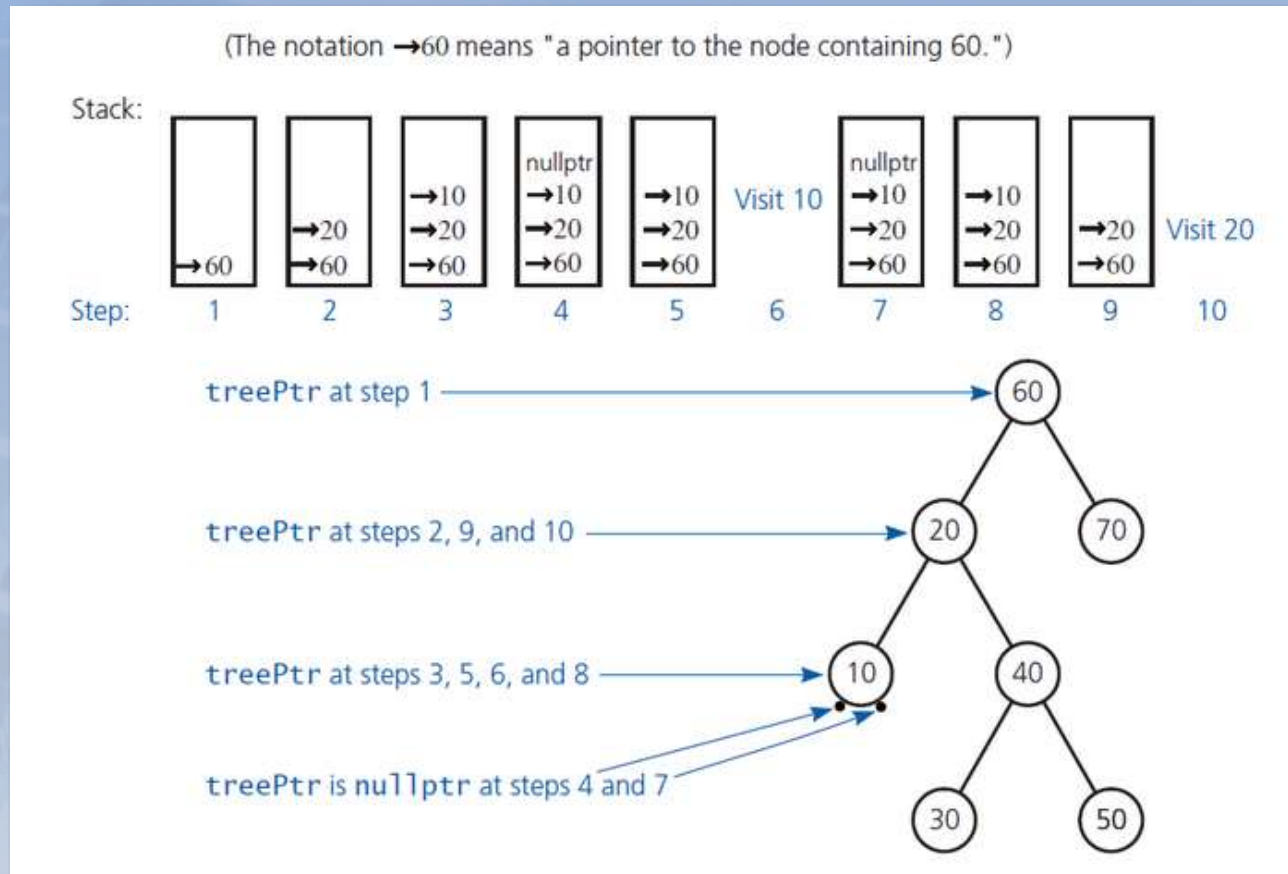
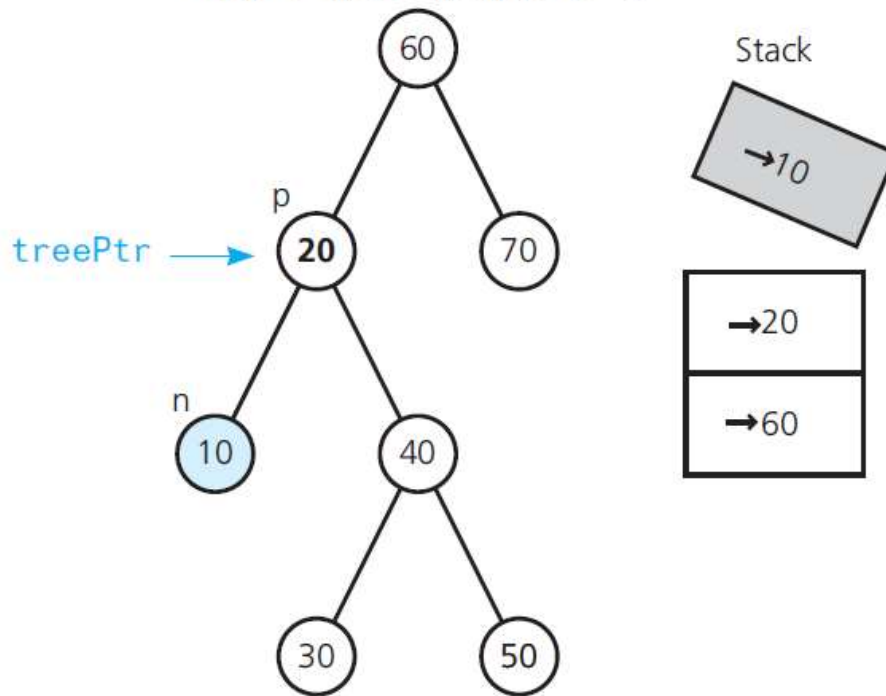


FIGURE 16-4 Contents of the implicit stack as **treePtr** progresses through a given tree during a recursive inorder traversal

# The Implementation

(a) Traversing 20's left subtree  
(steps 9 and 10 in Figure 16-4)



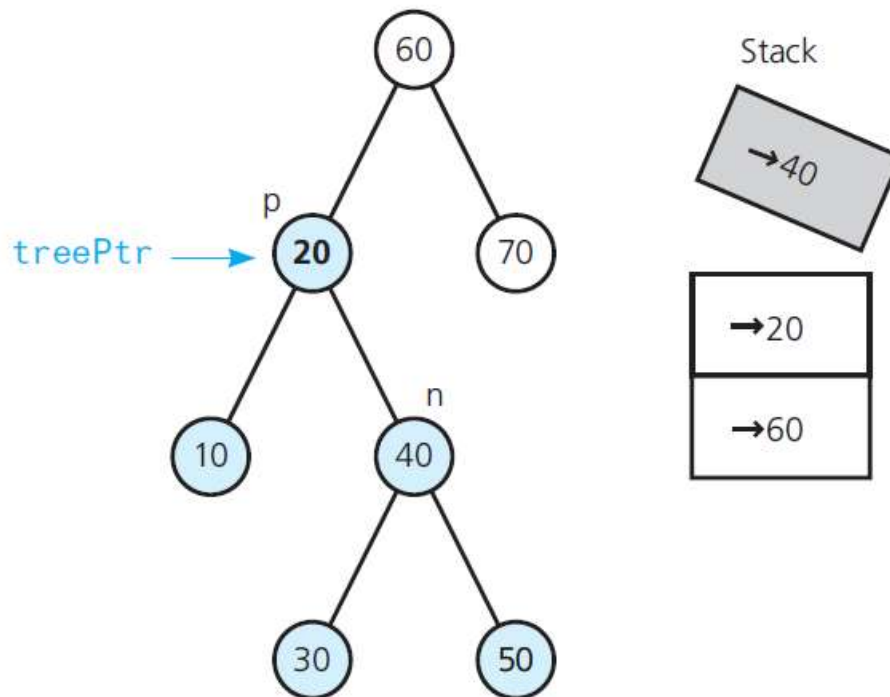
Left subtree of 20 has been traversed. Pop the reference to 10 from the stack, visit 20.

Figure 16-5 Steps during an **inorder** traversal of the subtrees of 20



# The Implementation

(b) Traversing 20's right subtree



Right subtree of 20 has been traversed. Pop the reference to 40 from stack.

Figure 16-5 Steps during an **inorder** traversal of the subtrees of 20

# The Implementation

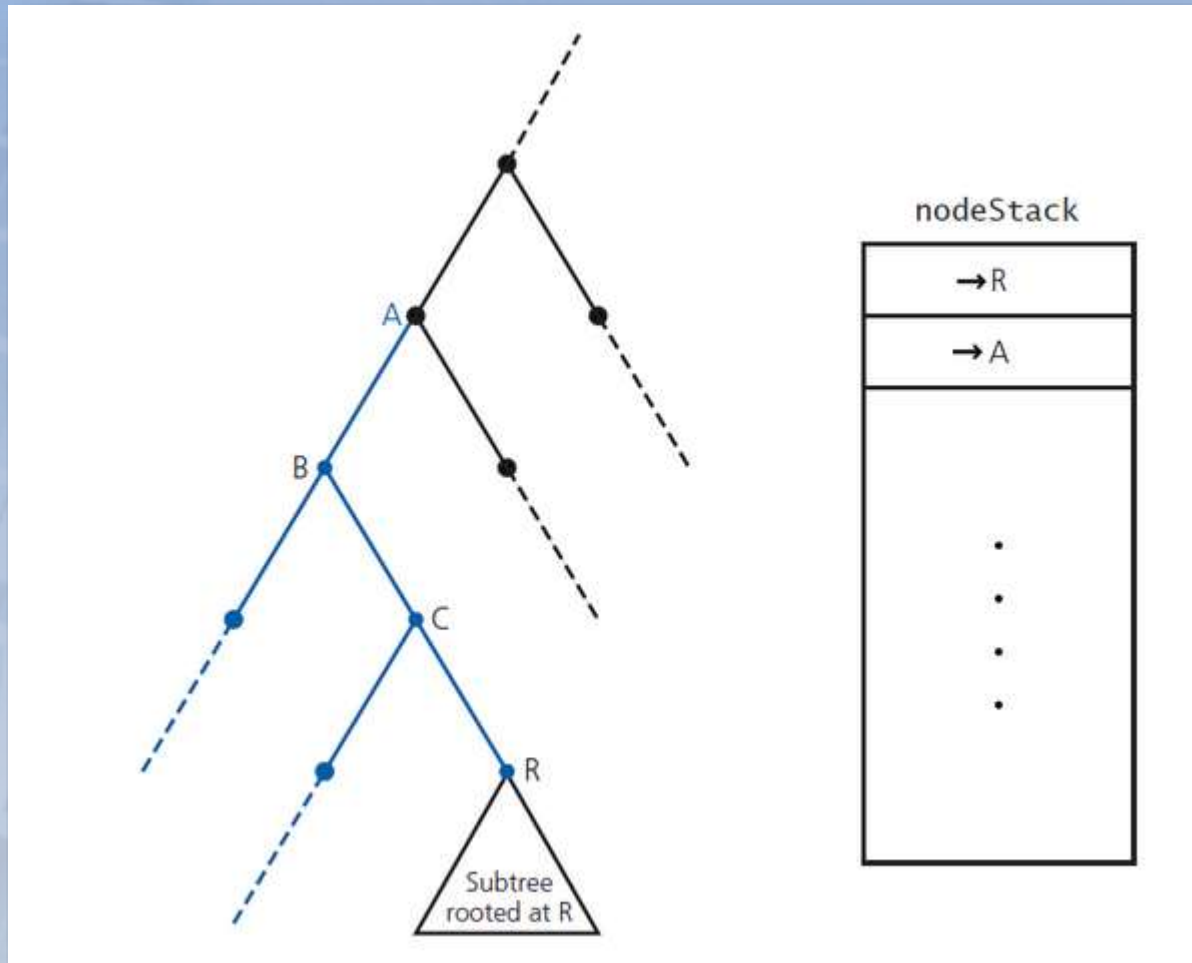


FIGURE 16-6 Avoiding returns to nodes B and C



# The Implementation

```
// Nonrecursively traverses a binary tree in inorder.
traverse(visit(item: ItemType): void): void
{
    // Initialize
    nodeStack = A new, empty stack
    curPtr = rootPtr // Start at root
    done = false

    while (!done)
    {
        if (curPtr != nullptr)
        {
            // Place pointer to node on stack before traversing the node's left subtree
            nodeStack.push(curPtr)

            // Traverse the left subtree
            curPtr = curPtr->getLeftChildPtr()
        }
        else // Backtrack from the empty subtree and visit the node at the top of

```

## Nonrecursive inorder traversal

# The Implementation

```
}  
else // Backtrack from the empty subtree and visit the node at the top of  
    // the stack; however, if the stack is empty, you are done  
{  
    done = nodeStack.isEmpty()  
    if (!done)  
    {  
        nodeStack.peek(curPtr)  
        visit(curPtr-&gtgetItem())  
        nodeStack.pop()  
  
        // Traverse the right subtree of the node just visited  
        curPtr = curPtr ->getRightChildPtr()  
    }  
}  
}  
}
```

Nonrecursive inorder traversal

# Link-Based Implementation of the ADT Binary Search Tree

- Uses same node objects as for binary-tree implementation.
- Class **BinaryNode** from Listing16-2 will be used
- Recursive search algorithm from Section15.3.2 is basis for operations



# Link-Based Implementation of the ADT Binary Search Tree

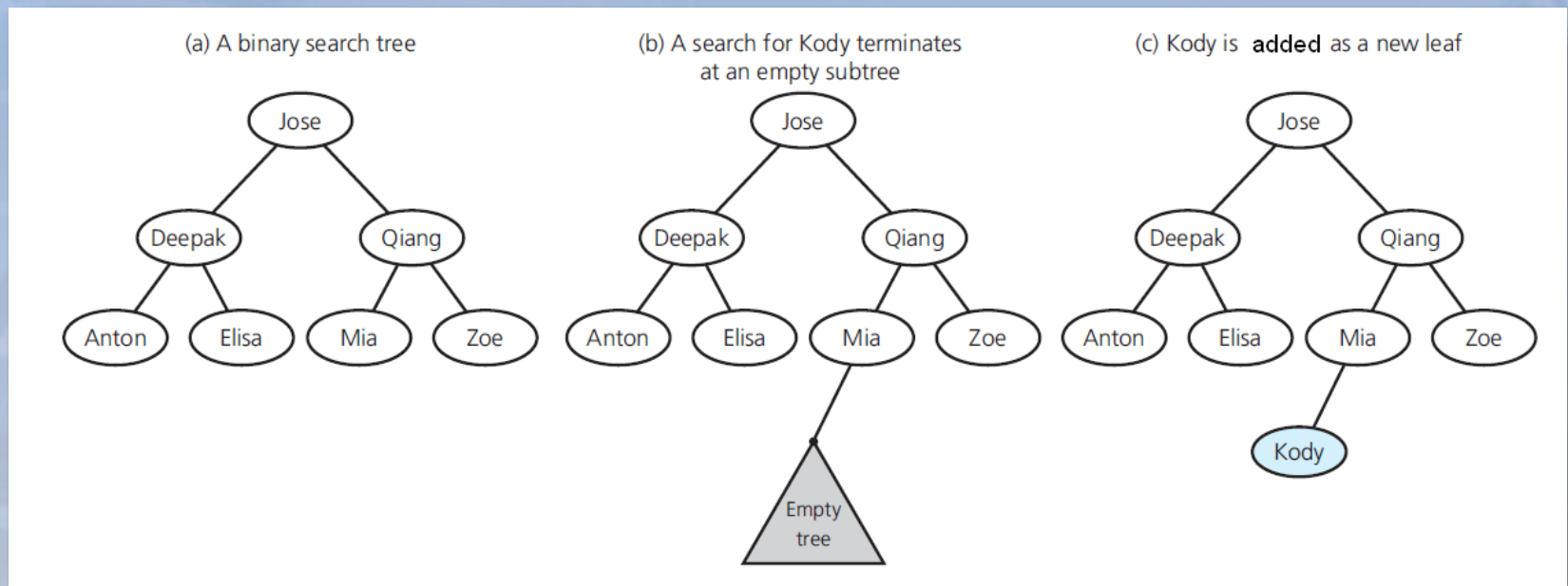


FIGURE 16-7 Adding Kody to a binary search tree

# Link-Based Implementation of the ADT Binary Search Tree

```
template<class ItemType>
bool BinarySearchTree<ItemType>::add(const ItemType& newData)
{
    auto newNodePtr = std::make_shared<BinaryNode<ItemType>>(newData);
    rootPtr = placeNode(rootPtr, newNodePtr);

    return true;
} // end add
```

Method `add`

# Link-Based Implementation of the ADT Binary Search Tree

*11 Recursively places a given new node at its proper position in a binary search tree.*

```
placeNode(subTreePtr: BinaryNodePointer,  
         newNodePtr: BinaryNodePointer): BinaryNodePointer  
{  
    if (subTreePtr is nullptr)  
        return newNodePtr  
    else if (subTreePtr->getItem() > newNodePtr->getItem())  
    {  
        tempPtr = placeNode(subTreePtr->getLeftChildPtr(), newNodePtr)  
        subTreePtr->setLeftChildPtr(tempPtr)  
    }  
    else  
    {  
        tempPtr = placeNode(subTreePtr->getRightChildPtr(), newNodePtr)  
        subTreePtr->setRightChildPtr(tempPtr)  
    }  
    return subTreePtr  
}
```

Refinement of addition algorithm



# Link-Based Implementation of the ADT Binary Search Tree

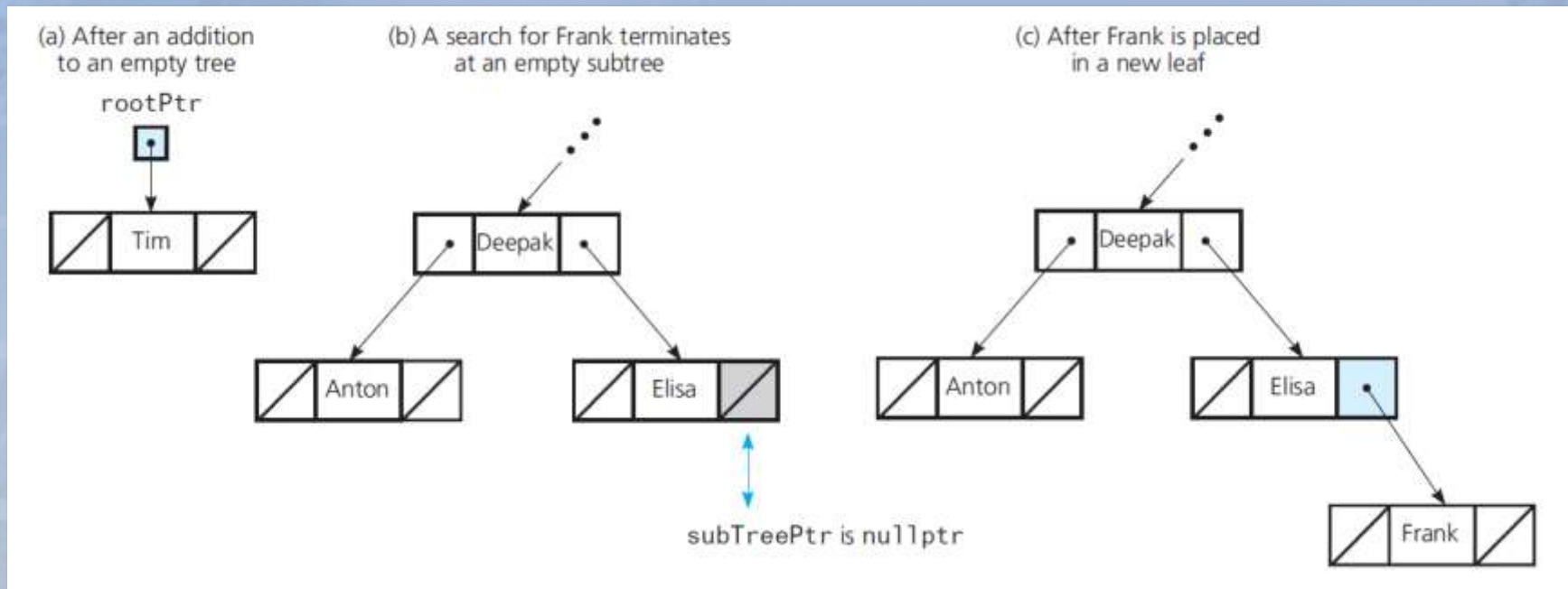


Figure 16-8 Adding new data to a binary search tree

# Link-Based Implementation of the ADT Binary Search Tree

```
// Removes the given target from a binary search tree.  
// Returns true if the removal is successful or false otherwise.  
removeValue(target: ItemType): boolean  
{  
    Locate the target by using the search algorithm  
    if (target is found)  
    {  
        Remove target from the tree  
        return true  
    }  
    else  
        return false  
}
```

First draft of the removal algorithm

# Link-Based Implementation of the ADT Binary Search Tree

- Cases for node  $N$  containing item to be removed

1.  $N$  is a leaf

- Remove leaf containing target
- Set pointer in parent to `nullptr`



# Link-Based Implementation of the ADT Binary Search Tree

- Cases for node  $N$  containing item to be removed
  2.  $N$  has only left (or right) child – cases are symmetrical
    - After  $N$  removed, all data items rooted at  $L$  (or  $R$ ) are adopted by root of  $N$
    - All items adopted are in correct order, binary search tree property preserved

# Link-Based Implementation of the ADT Binary Search Tree

- Cases for node  $N$  containing item to be removed
  3.  $N$  has two children
    - Locate another node  $M$  easier to remove from tree than  $N$
    - Copy item that is in  $M$  to  $N$
    - Remove  $M$  from tree

# Link-Based Implementation of the ADT Binary Search Tree

(a) Before removing the data item in node *N*

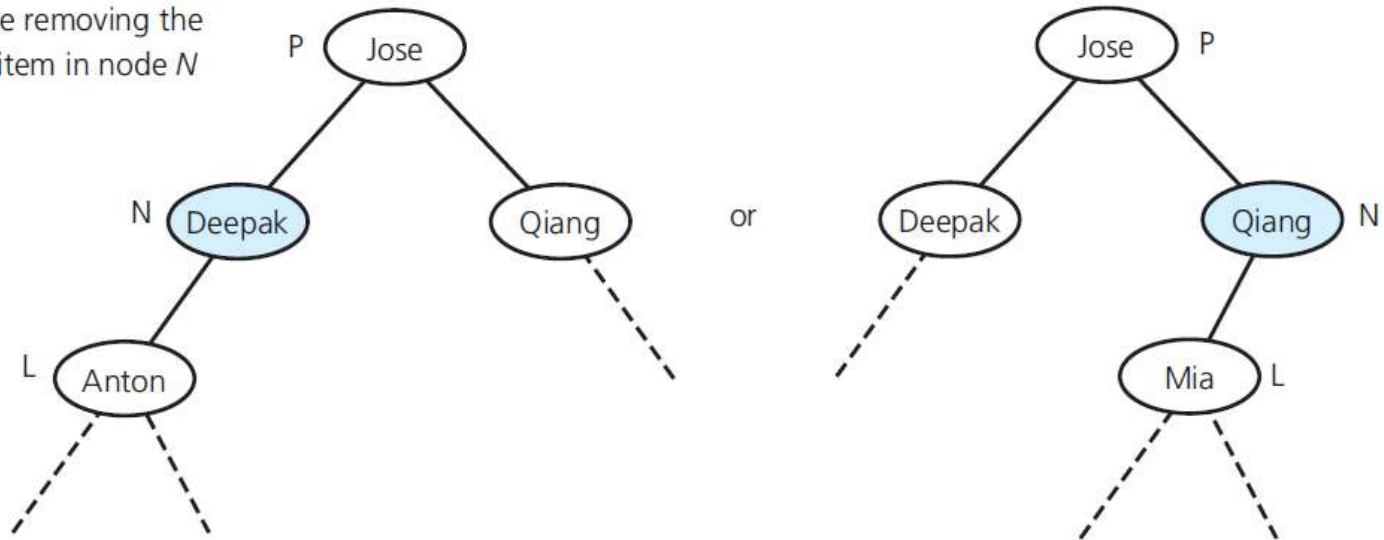


Figure 16-9 Case 2 for **removeValue**: The data item to remove is in a node *N* that has only a left child and whose parent is node *P*



# Link-Based Implementation of the ADT Binary Search Tree

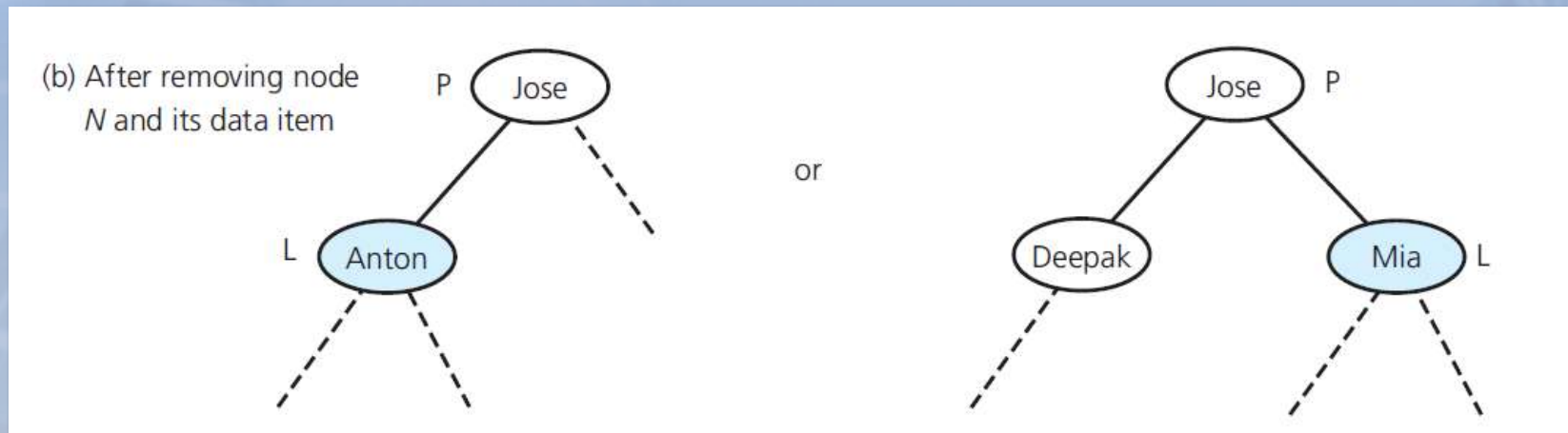


Figure 16-9 Case 2 for **removeValue**: The data item to remove is in a node *N* that has only a left child and whose parent is node *P*

# Link-Based Implementation of the ADT Binary Search Tree

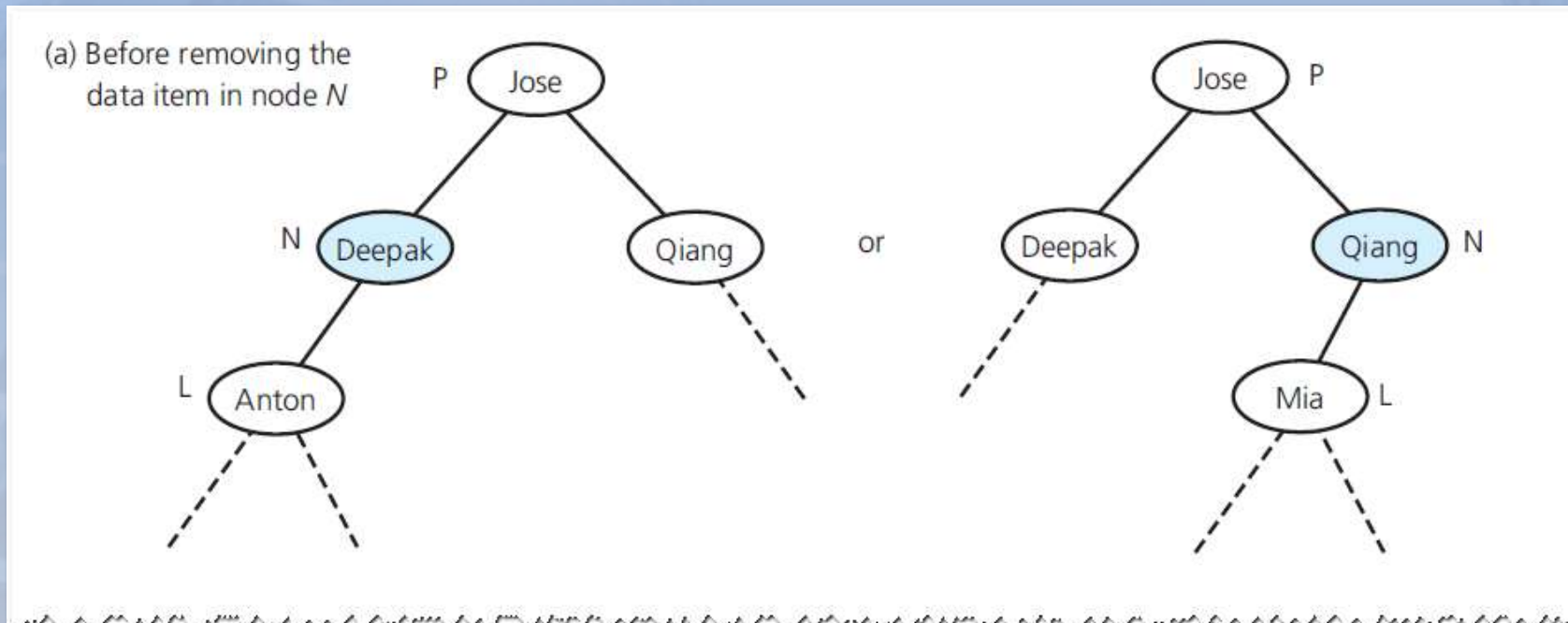


FIGURE 16-9 Case 2 for `removeValue`: The data item to remove is in a node *N* that has only a left child and whose parent is node *P*

# Link-Based Implementation of the ADT Binary Search Tree

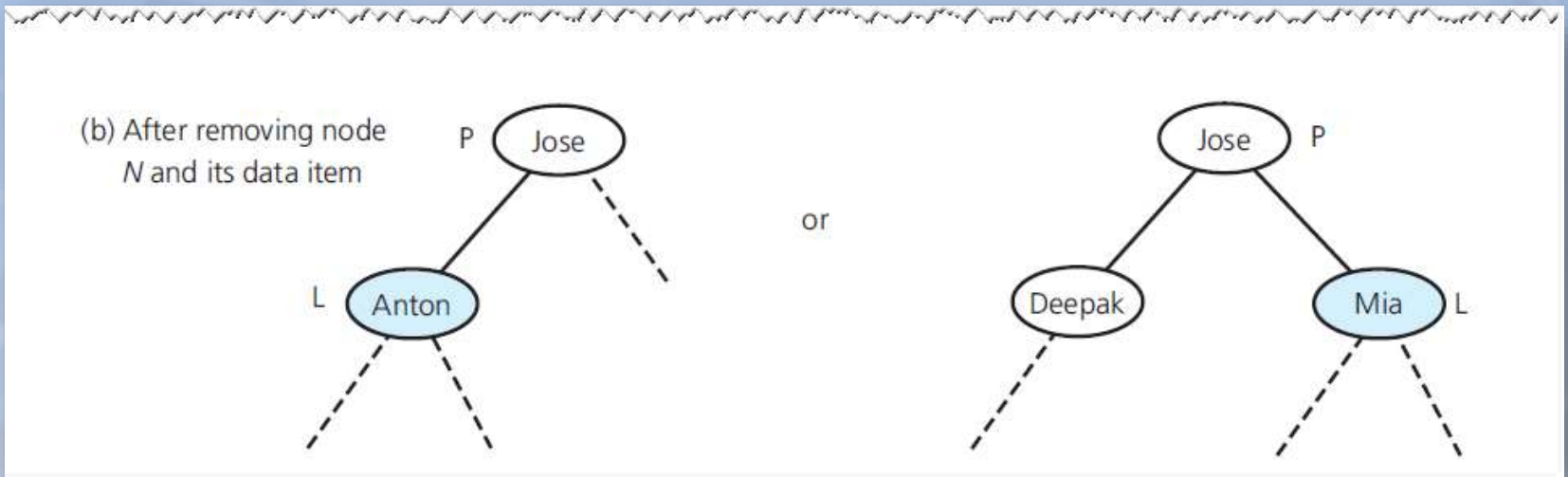


FIGURE 16-9 Case 2 for **removeValue**: The data item to remove is in a node *N* that has only a left child and whose parent is node *P*



# Link-Based Implementation of the ADT Binary Search Tree

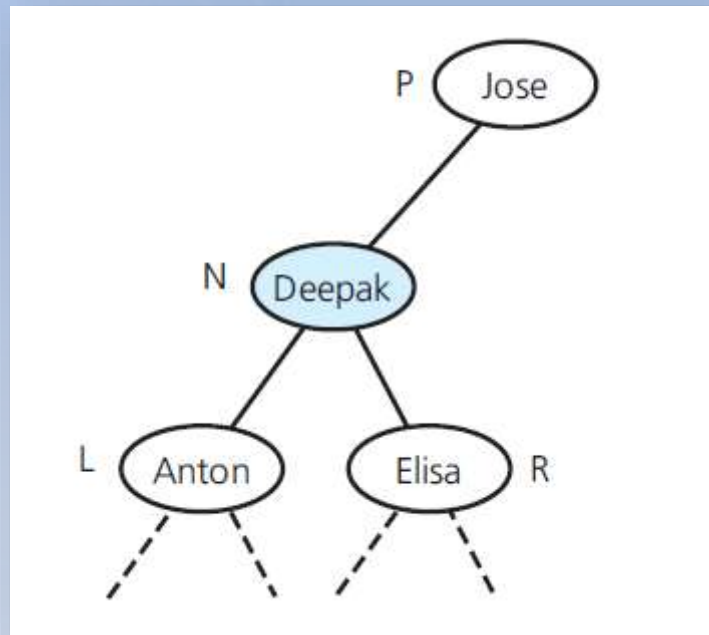


FIGURE 16-10 Case 3: The data item to remove is in a node *N* that has two children

# Link-Based Implementation of the ADT Binary Search Tree

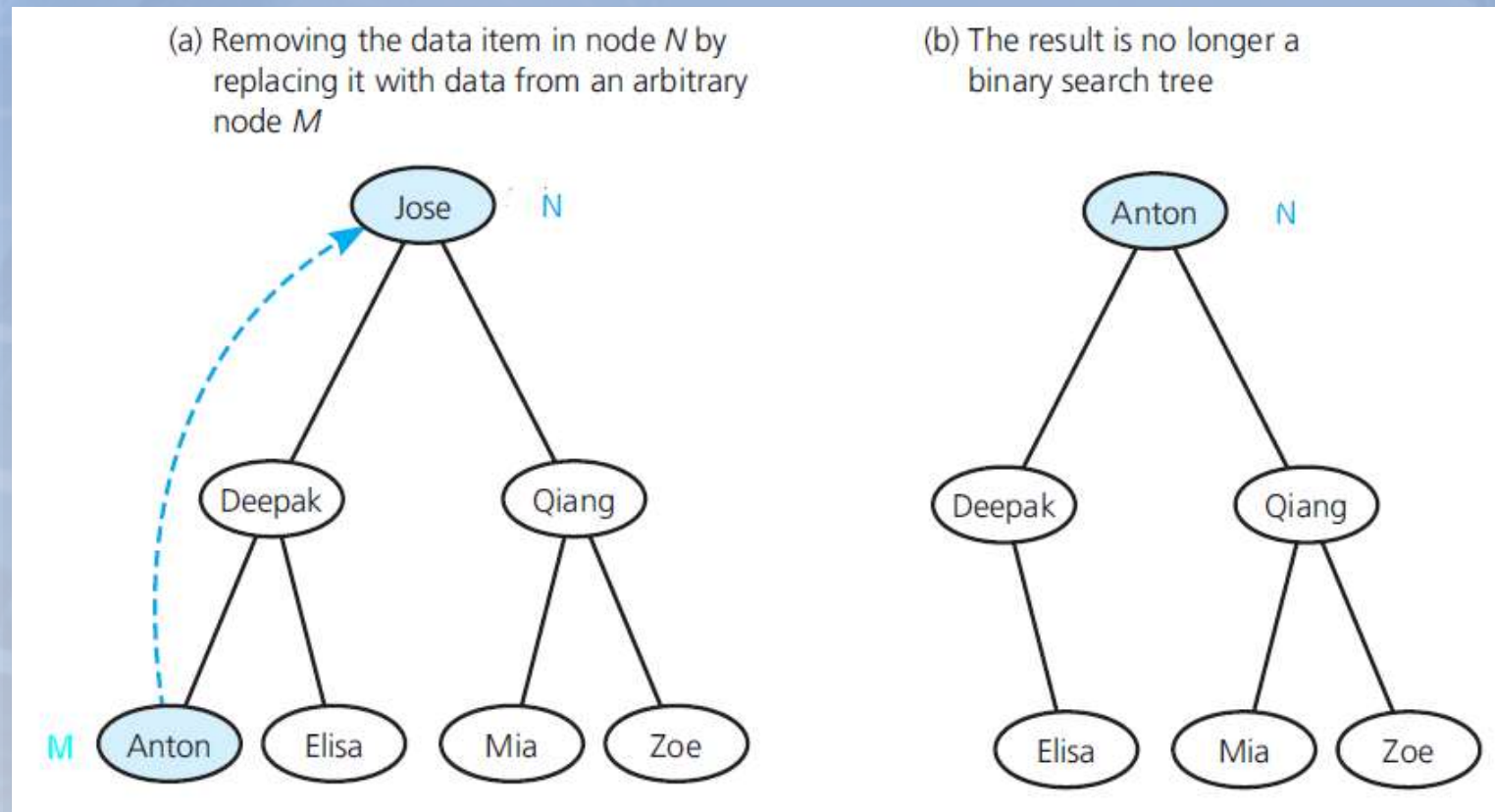
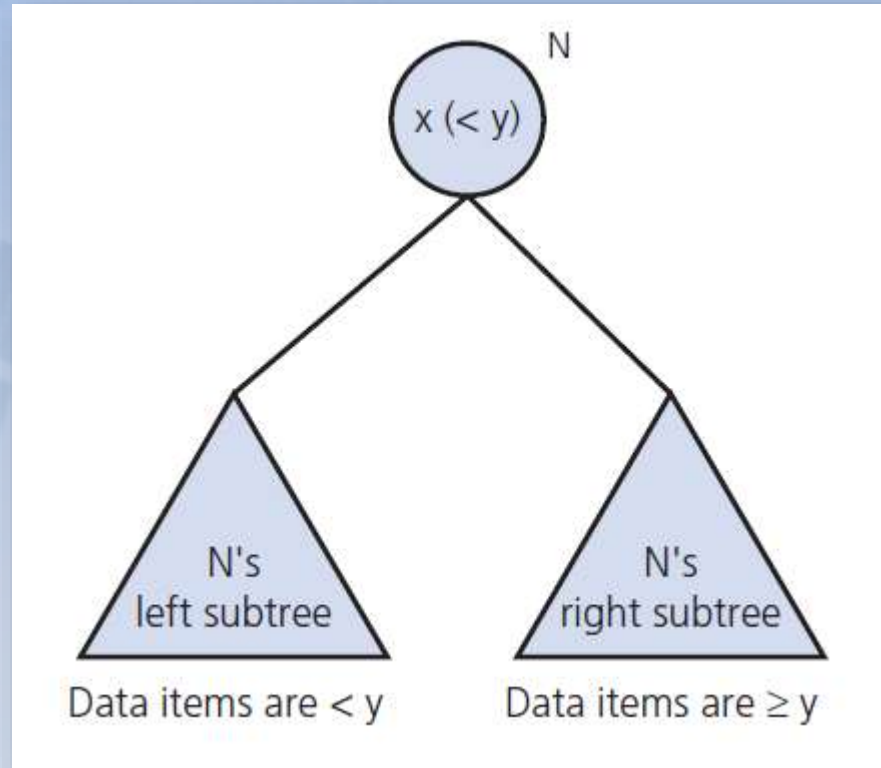


FIGURE16-11 Not any node will do

# Link-Based Implementation of the ADT Binary Search Tree



**FIGURE 16-12**



# Link-Based Implementation of the ADT Binary Search Tree

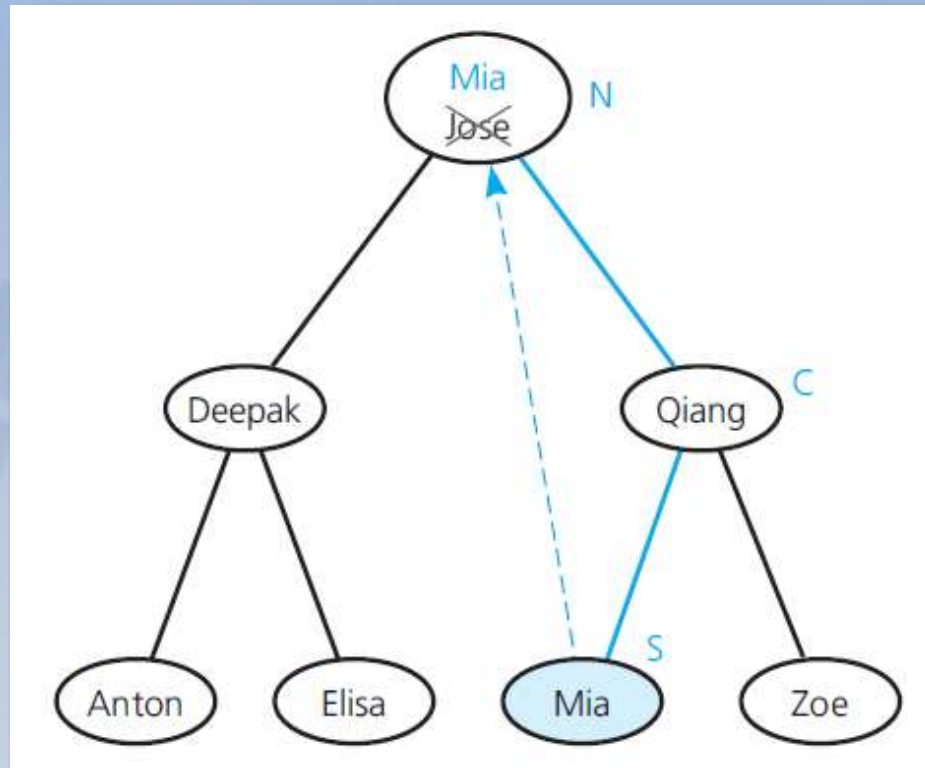


FIGURE 16-13 Replacing the data item in node *N* with its inorder successor

# Link-Based Implementation of the ADT Binary Search Tree

```
// Removes the given target from the binary search tree to which subTreePtr points.  
// Returns a pointer to the node at this tree location after the value is removed.  
// Sets isSuccessful to true if the removal is successful, or false otherwise.  
removeValue(subTreePtr: BinaryNodePointer, target: ItemType,  
            isSuccessful: boolean&): BinaryNodePointer  
{  
    if (subTreePtr == nullptr)  
    {  
        isSuccessful = false  
    }  
    else if (subTreePtr->getItem() == target)  
    {  
        // Item is in the root of some subtree  
        subTreePtr = removeNode(subTreePtr) // Remove the item  
        isSuccessful = true  
    }  
    else if (subTreePtr->getItem() > target)  
    {  
        // Search the left subtree  
        tempPtr = removeValue(subTreePtr->getLeftChildPtr(), target, isSuccessful)  
        subTreePtr->setLeftChildPtr(tempPtr)  
    }  
    else
```

Final draft of the removal algorithm



# Link-Based Implementation of the ADT Binary Search Tree

```
}  
else  
{  
    // Search the right subtree  
    tempPtr = removeValue(subTreePtr->getRightChildPtr(), target, isSuccessful)  
    subTreePtr->setRightChildPtr(tempPtr)  
}  
return subTreePtr  
}  
  
// Removes the data item in the node, N, to which nodePtr points.  
// Returns a pointer to the node at this tree location after the removal.  
removeNode(nodePtr: BinaryNodePointer): BinaryNodePointer  
{  
    if (N is a leaf)  
    {  
        // Remove leaf from the tree  
        Delete the node to which nodePtr points (done for us if nodePtr is a smart pointer)  
        return nodePtr  
    }  
    else if (N has only one child C)
```

Final draft of the removal algorithm



# Link-Based Implementation of the ADT Binary Search Tree

```
}  
else if (N has only one child C)  
{  
    // C replaces N as the child of N's parent  
    if (C is a left child)  
        nodeToConnectPtr = nodePtr->getLeftChildPtr()  
    else  
        nodeToConnectPtr = nodePtr->getRightChildPtr()  
  
    Delete the node to which nodePtr points (done for us if nodePtr is a smart pointer)  
    return nodeToConnectPtr  
}
```

Final draft of the removal algorithm

# Link-Based Implementation of the ADT Binary Search Tree

```
else // N has two children
{
    // Find the inorder successor of the entry in N: it is in the left subtree rooted
    // at N's right child
    tempPtr = removeLeftmostNode(nodePtr->getRightChildPtr(), newNodeValue)
    nodePtr->setRightChildPtr(tempPtr)
    nodePtr->setItem(newNodeValue) // Put replacement value in node N
    return nodePtr
}

// Removes the leftmost node in the left subtree of the node pointed to by nodePtr.
// Sets inorderSuccessor to the value in this node.
// Returns a pointer to the revised subtree.
removeLeftmostNode(nodePtr: BinaryNodePointer,
                    inorderSuccessor: ItemType&): BinaryNodePointer
{
```

Final draft of the removal algorithm



# Link-Based Implementation of the ADT Binary Search Tree

```
inorderSuccessor: ItemType&): BinaryNodePointer
{
    if (nodePtr->getLeftChildPtr() == nullptr)
    {
        // This is the node you want; it has no left child, but it might have a right subtree
        inorderSuccessor = nodePtr->getItem()
        return removeNode(nodePtr)
    }
    else
    {
        tempPtr = removeLeftmostNode(nodePtr->getLeftChildPtr(), inorderSuccessor)
        nodePtr->setLeftChildPtr(tempPtr)
        return nodePtr
    }
}
```

Final draft of the removal algorithm



# Link-Based Implementation of the ADT Binary Search Tree

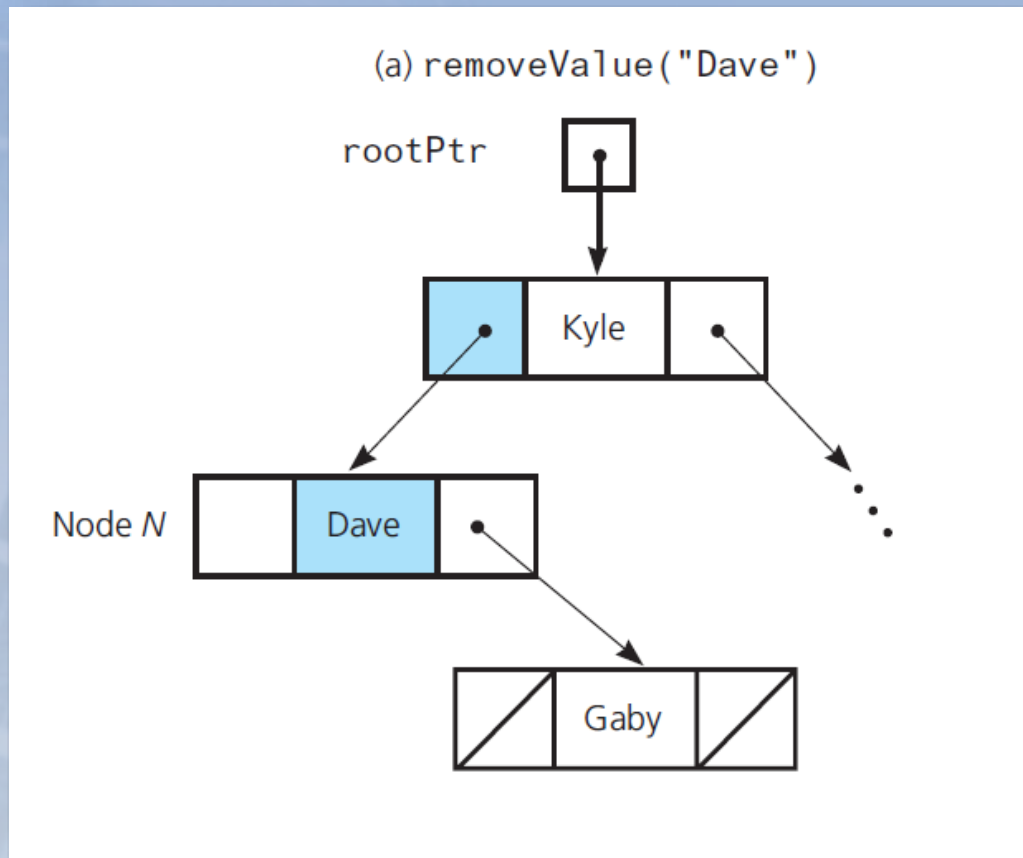


FIGURE 16-14 Recursive removal of node N

# Link-Based Implementation of the ADT Binary Search Tree

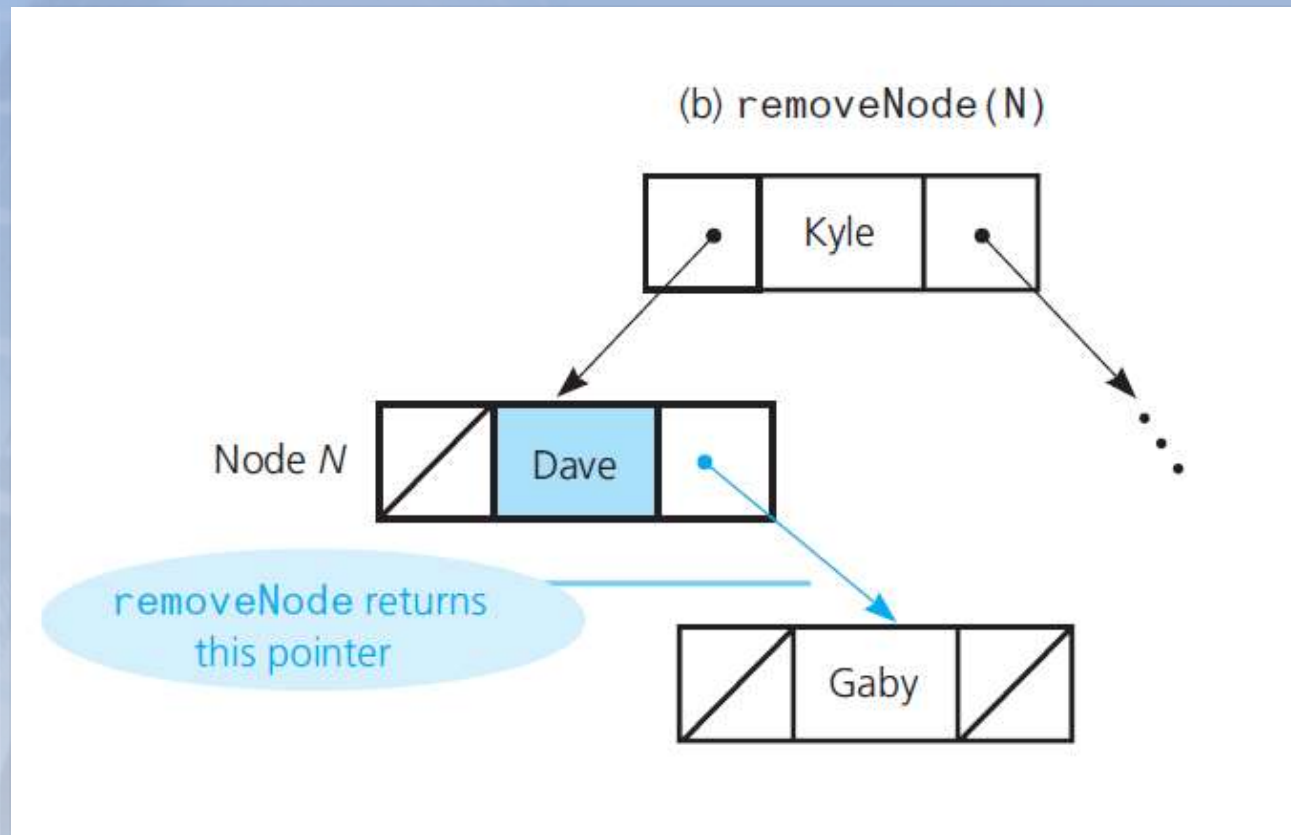


FIGURE 16-14 Recursive removal of node N

# Link-Based Implementation of the ADT Binary Search Tree

Pointer returned by `removeNode/removeValue` is assigned to left child pointer of Node *N*'s parent

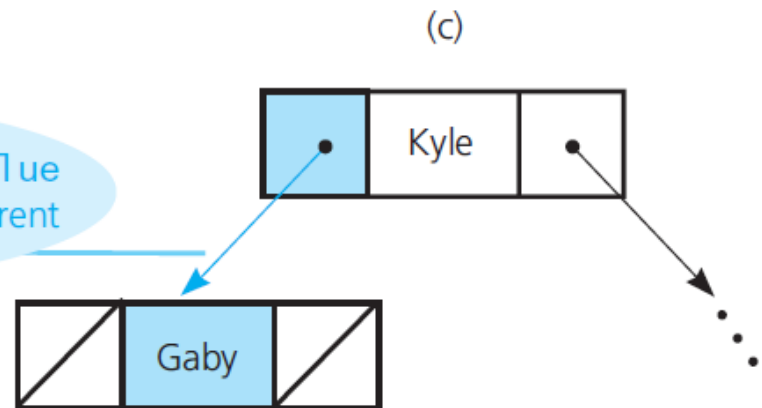


FIGURE 16-14 Recursive removal of node *N*



# Link-Based Implementation of the ADT Binary Search Tree

```
// Locates the node in the binary search tree to which subTreePtr points and that contains  
// the value target. Returns either a pointer to the located node or nullptr if such a  
// node is not found.  
findNode(subTreePtr: BinaryNodePointer, target: ItemType): BinaryNodePointer  
{  
    if (subTreePtr == nullptr)  
        return nullptr // Not found  
  
    else if (subTreePtr->getItem() == target)  
        return subTreePtr; // Found  
    else if (subTreePtr->getItem() > target)  
        // Search left subtree  
        return findNode(subTreePtr->getLeftChildPtr(), target)  
    else  
        // Search right subtree  
        return findNode(subTreePtr->getRightChildPtr(), target)  
}
```

## Algorithm for findNode

# The Class `BinarySearchTree`

```
1  /** Link-based implementation of the ADT binary search tree.
2   * @file BinarySearchTree.h */
3
4  #ifndef BINARY_SEARCH_TREE_
5  #define BINARY_SEARCH_TREE_
6
7  #include "BinaryTreeInterface.h"
8  #include "BinaryNode.h"
9  #include "BinaryNodeTree.h"
10 #include "NotFoundException.h"
11 #include "PrecondViolatedExcept.h"
12 #include <memory>
13
14 template<class ItemType>
15 class BinarySearchTree : public BinaryNodeTree<ItemType>
16 {
17 private:
18     std::shared_ptr<BinaryNode<ItemType>> rootPtr;
```

LISTING 16-4 A header file for the link-based implementation of the class `BinarySearchTree`



# The Class `BinarySearchTree`

```
19  protected:
20      //-----
21      //    Protected Utility Methods Section:
22      //    Recursive helper methods for the public methods.
23      //-----
24      // Places a given new node at its proper position in this binary
25      // search tree
26      auto placeNode(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
27                    std::shared_ptr<BinaryNode<ItemType>> newNode);
28
29      // Removes the given target value from the tree while maintaining a
30      // binary search tree.
31      auto removeValue(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
32                      const ItemType target,
33                      bool& isSuccessful) override;
34
35      // Removes a given node from a tree while maintaining a binary search tree.
36      auto removeNode(std::shared_ptr<BinaryNode<ItemType>> nodePtr);
37
```

LISTING 16-4 A header file for the link-based implementation of the class `BinarySearchTree`



# The Class `BinarySearchTree`

```
35 // Removes a given node from a tree while maintaining a binary search tree.
36 auto removeNode(std::shared_ptr<BinaryNode<ItemType>> nodePtr);
37
38 // Removes the leftmost node in the left subtree of the node
39 // pointed to by nodePtr.
40 // Sets inorderSuccessor to the value in this node.
41 // Returns a pointer to the revised subtree.
42 auto removeLeftmostNode(std::shared_ptr<BinaryNode<ItemType>>subTreePtr,
43                          ItemType& inorderSuccessor);
44
45 // Returns a pointer to the node containing the given value,
46 // or nullptr if not found.
47 auto findNode(std::shared_ptr<BinaryNode<ItemType>> treePtr,
48               const ItemType& target) const;
49
50 public:
51 //-----
52 //   Constructor and Destructor Section.
53 //-----
54 BinarySearchTree();
55 BinarySearchTree(const ItemType& rootItem);
56 BinarySearchTree(const BinarySearchTree<ItemType>& tree);
57 virtual ~BinarySearchTree();
```

LISTING 16-4 A header file for the link-based implementation of the class `BinarySearchTree`

# The Class `BinarySearchTree`

```
55 BinarySearchTree(const ItemType& rootItem);
56 BinarySearchTree(const BinarySearchTree<ItemType>& tree);
57 virtual ~BinarySearchTree();
58
59 //-----
60 //   Public Methods Section.
61 //-----
62 bool isEmpty() const;
63 int getHeight() const;
64 int getNumberOfNodes() const;
65 ItemType getRootData() const throw (PrecondViolatedExcept);
66 void setRootData(const ItemType& newData);
67 bool add(const ItemType& newEntry);
68 bool remove(const ItemType& target);
69 void clear();
70 ItemType getEntry(const ItemType& anEntry) const throw (NotFoundException);
```

LISTING 16-4 A header file for the link-based implementation of the class `BinarySearchTree`



# The Class `BinarySearchTree`

```
71     bool contains(const ItemType& anEntry) const;
72
73     //-----
74     //  Public Traversals Section.
75     //-----
76     void preorderTraverse(void visit(ItemType&)) const;
77     void inorderTraverse(void visit(ItemType&)) const;
78     void postorderTraverse(void visit(ItemType&)) const;
79
80     //-----
81     //  Overloaded Operator Section.
82     //-----
83     BinarySearchTree<ItemType>&
84         operator=(const BinarySearchTree<ItemType>& rightHandSide);
85 }; // end BinarySearchTree
86 #include "BinarySearchTree.cpp"
87 #endif
```

LISTING 16-4 A header file for the link-based implementation of the class `BinarySearchTree`



# Saving a Binary Search Tree in a File

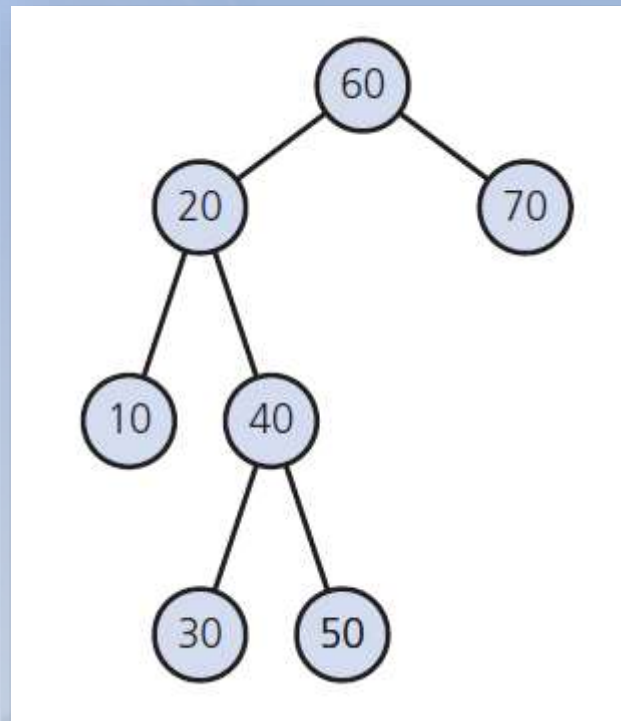


FIGURE 16-15 An initially empty binary search tree after the addition of 60, 20, 10, 40, 30, 50, and 70

# Saving a Binary Search Tree in a File

- Use preorder traversal to save binary search tree in a file
  - Restore to original shape by using method `add`
- Balanced binary search tree increases efficiency of ADT operations

# Saving a Binary Search Tree in a File

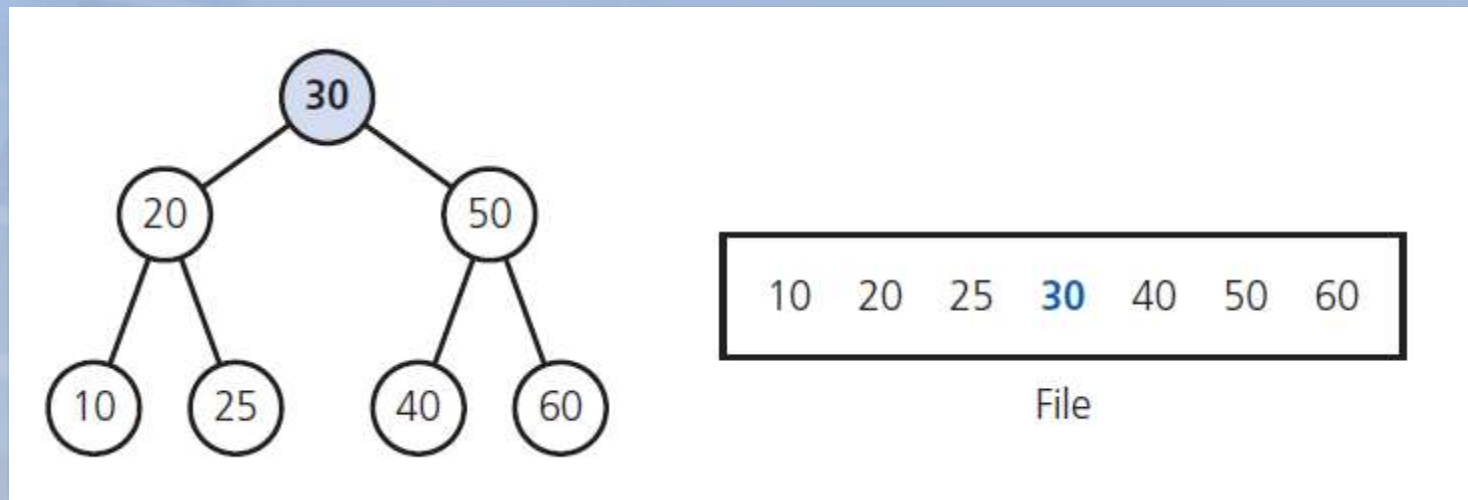


FIGURE 16-16 A full tree saved in a file by using inorder traversal



# Saving a Binary Search Tree in a File

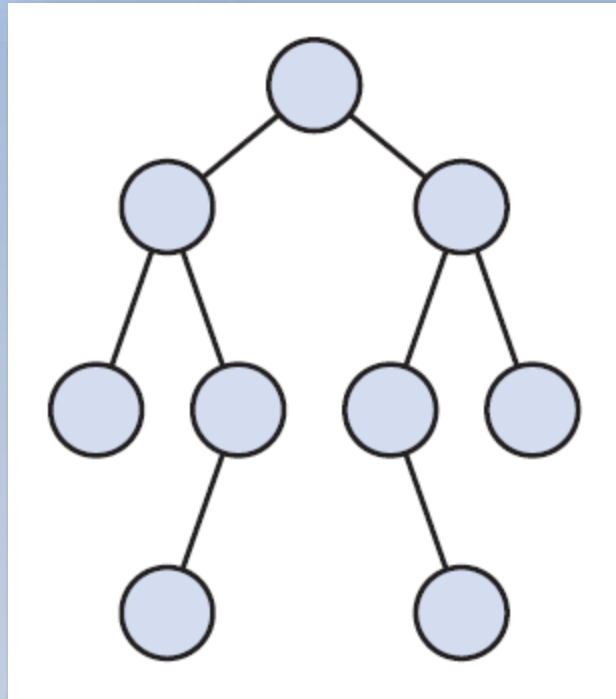


FIGURE 16-17 A tree of minimum height that is not complete

# Saving a Binary Search Tree in a File

```
// Builds a minimum-height binary search tree from n sorted values in a file.
// Returns a pointer to the tree's root.
readTree(treePtr: BinaryNodePointer, n: integer): BinaryNodePointer
{
    if (n > 0)
    {
        treePtr = pointer to new node with nullptr as its child pointers

        // Construct the left subtree
        leftPtr = readTree(treePtr->getLeftChildPtr(), n / 2)
        treePtr->setLeftChildPtr(leftPtr)
    }
}
```

Building a minimum-height binary search tree

# Saving a Binary Search Tree in a File

```
// Get the data item for this node
rootItem = next data item from file
treePtr->setItem(rootItem)

// Construct the right subtree
rightPtr = readTree(treePtr->getRightChildPtr(), (n - 1) / 2)
treePtr->setRightChildPtr(rightPtr)

return treePtr
}
else
    return nullptr
}
```

Building a minimum-height binary search tree



# Tree Sort

```
// Sorts the integers in an array into ascending order.  
treeSort(anArray: array, n: integer)  
{  
    Add anArray's entries to a binary search tree bst  
    Traverse bst in inorder. As you visit bst's nodes, copy their data items into successive  
    locations of anArray  
}
```

Tree sort uses a binary search tree.

# General Trees

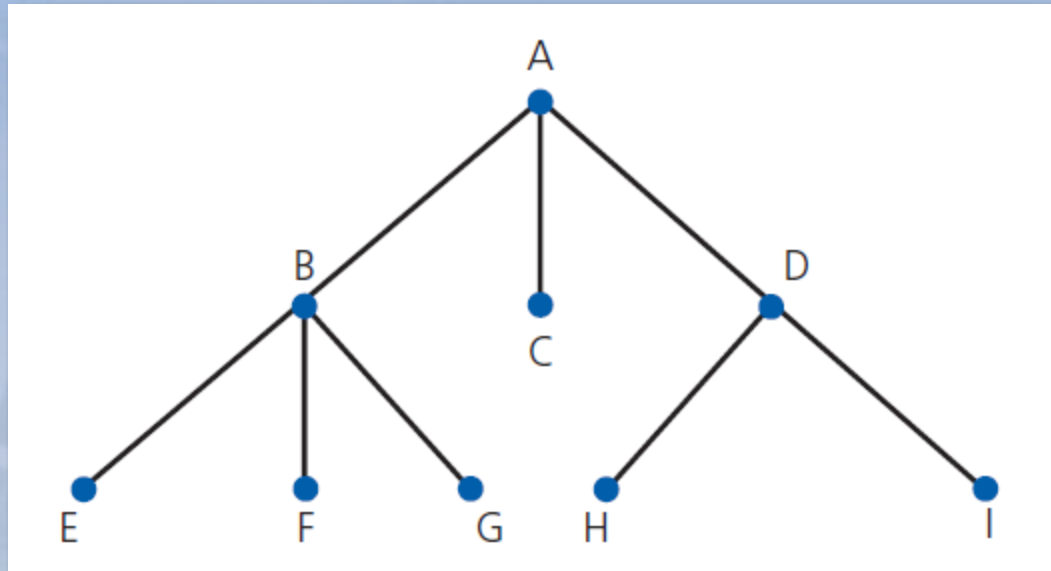


FIGURE 16-18 A general tree or an  $n$ -ary tree with  $n = 3$

# General Trees

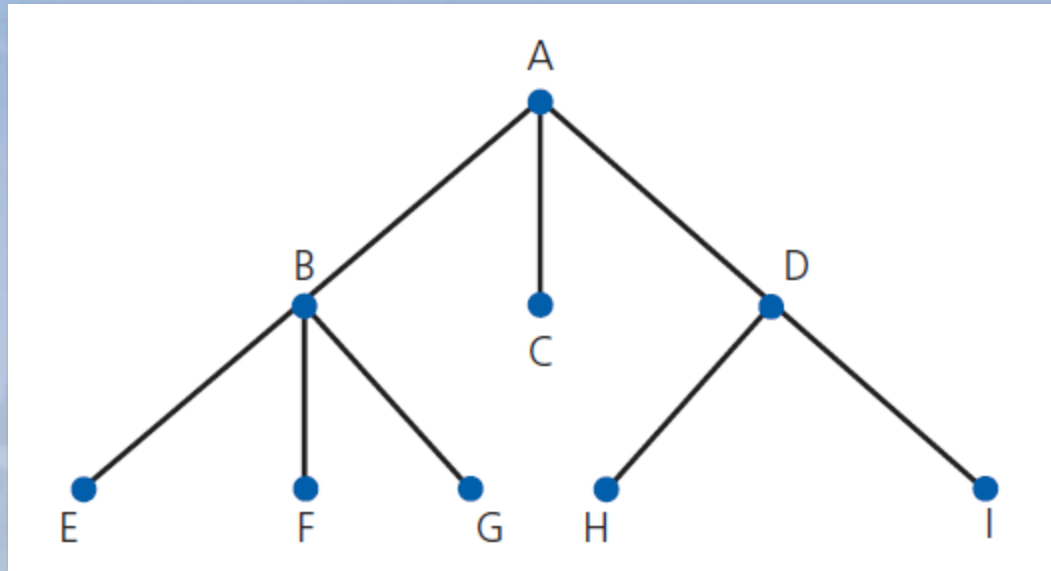


FIGURE 16-18 A general tree or an  $n$ -ary tree with  $n = 3$



# General Trees

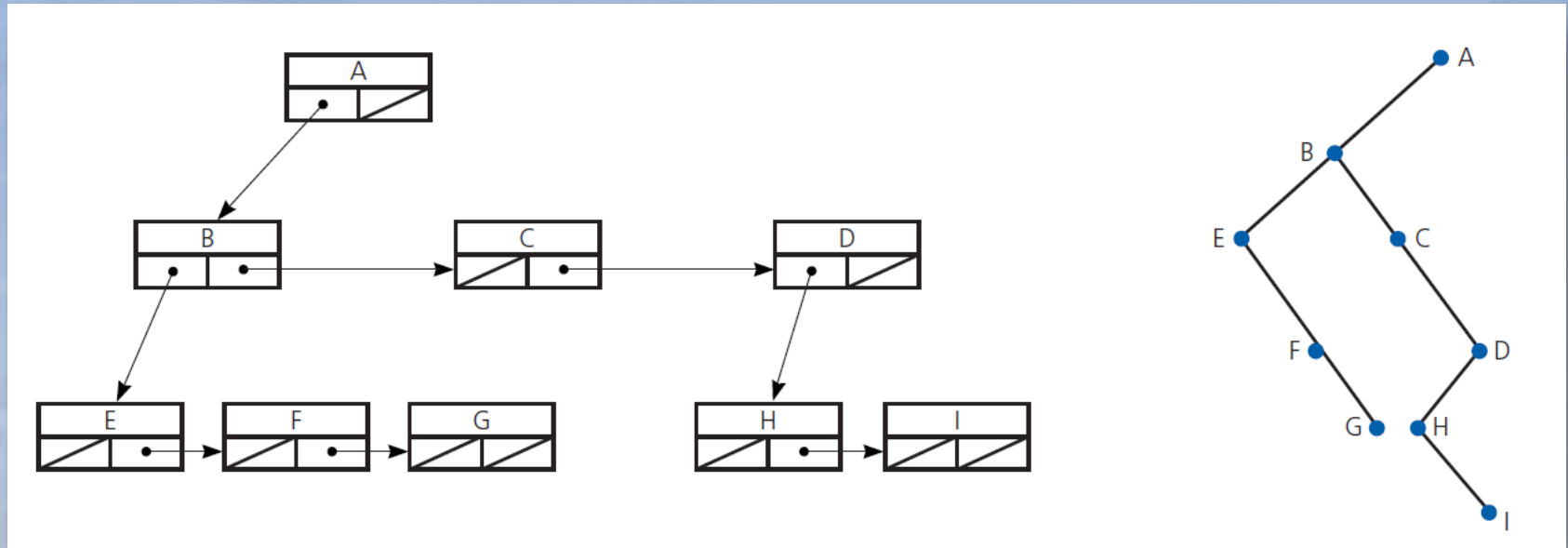


FIGURE 16-19 An implementation of a general tree and its equivalent binary tree

# General Trees

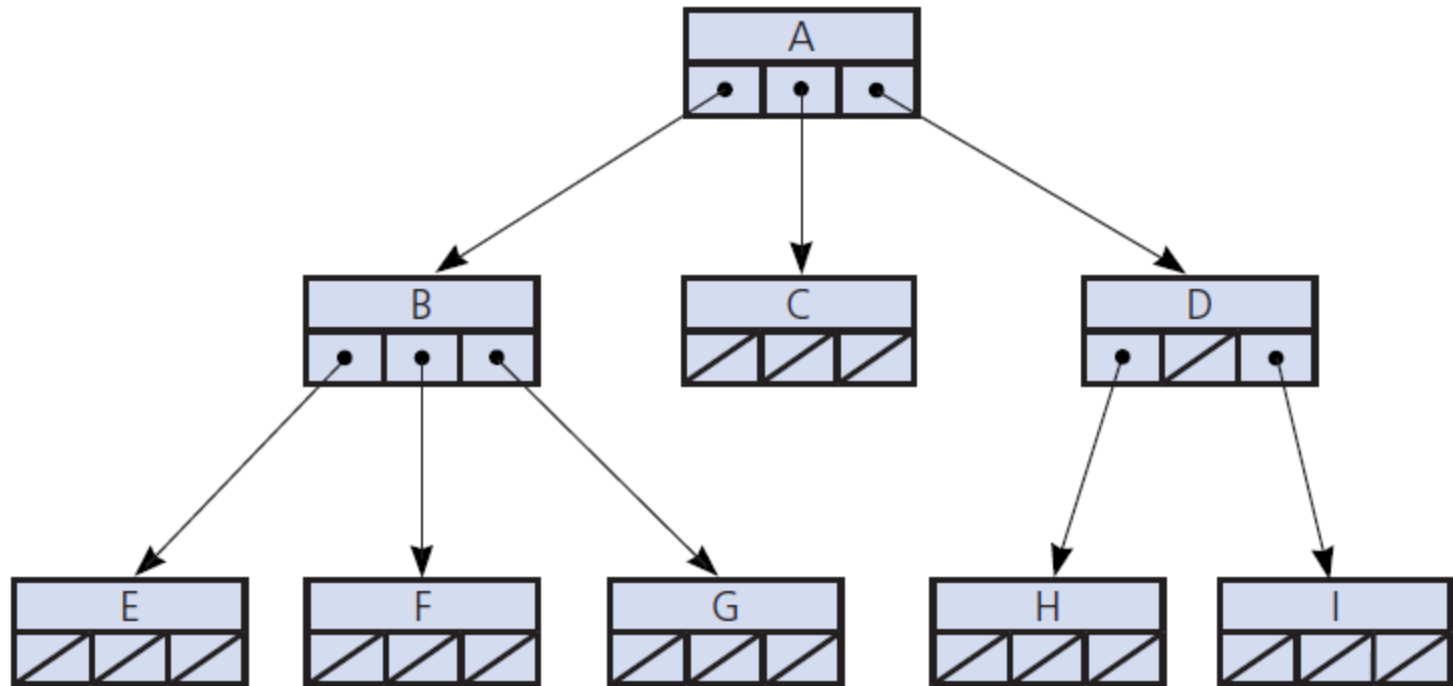
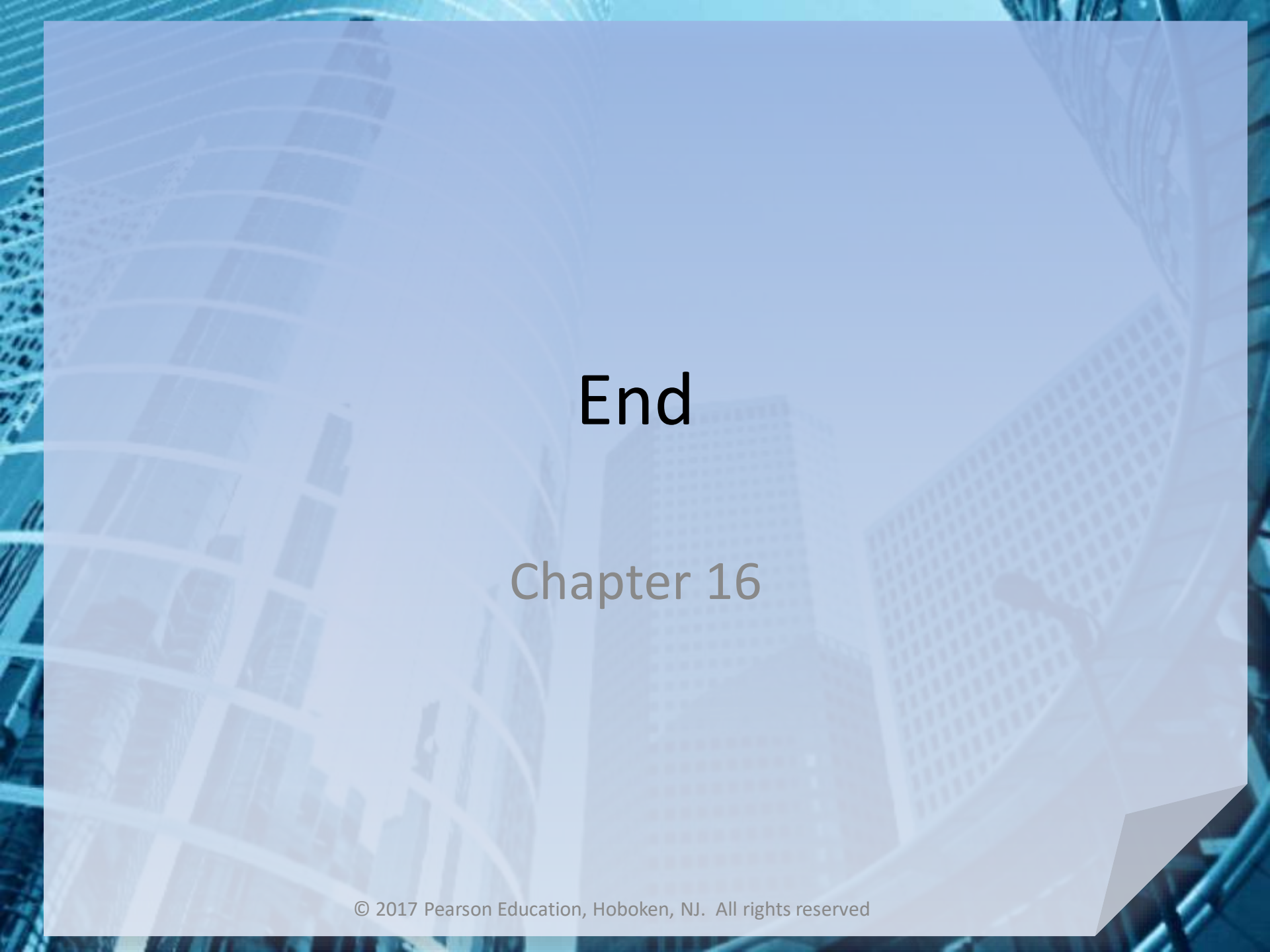


FIGURE 16-20 An implementation of the  $n$ -ary tree in Figure 16-18



# End

## Chapter 16