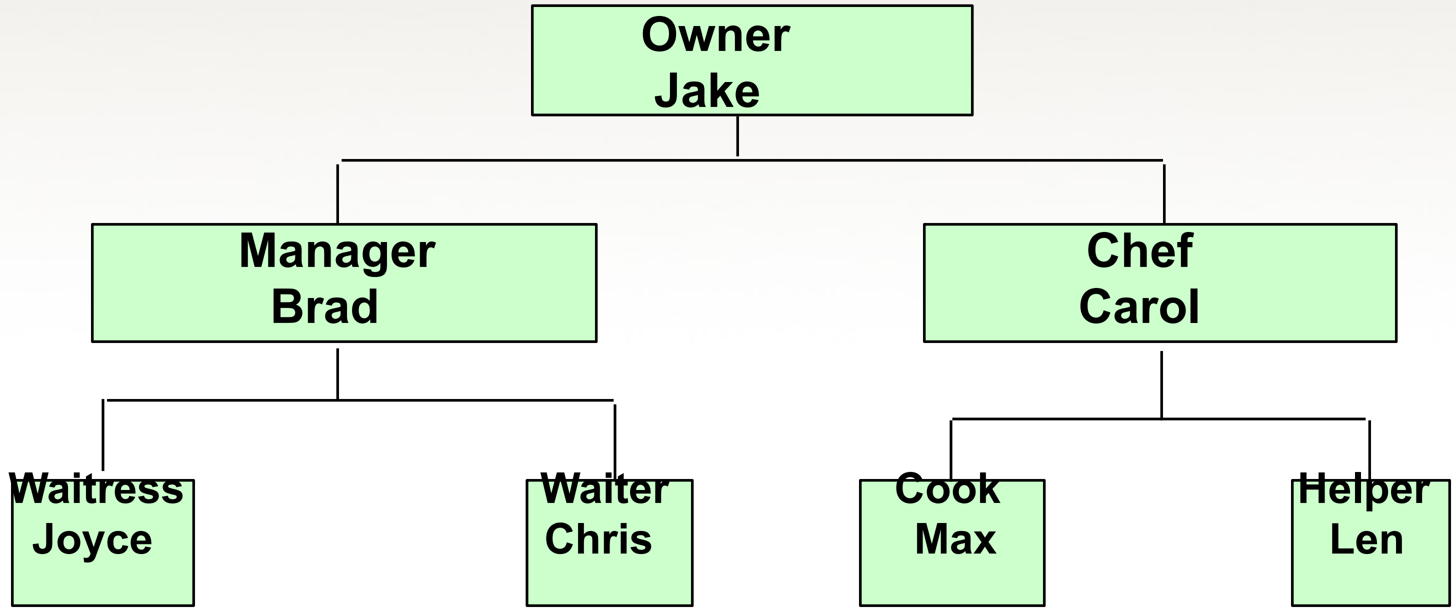
Nell Dale

C++ Plus Data Structures

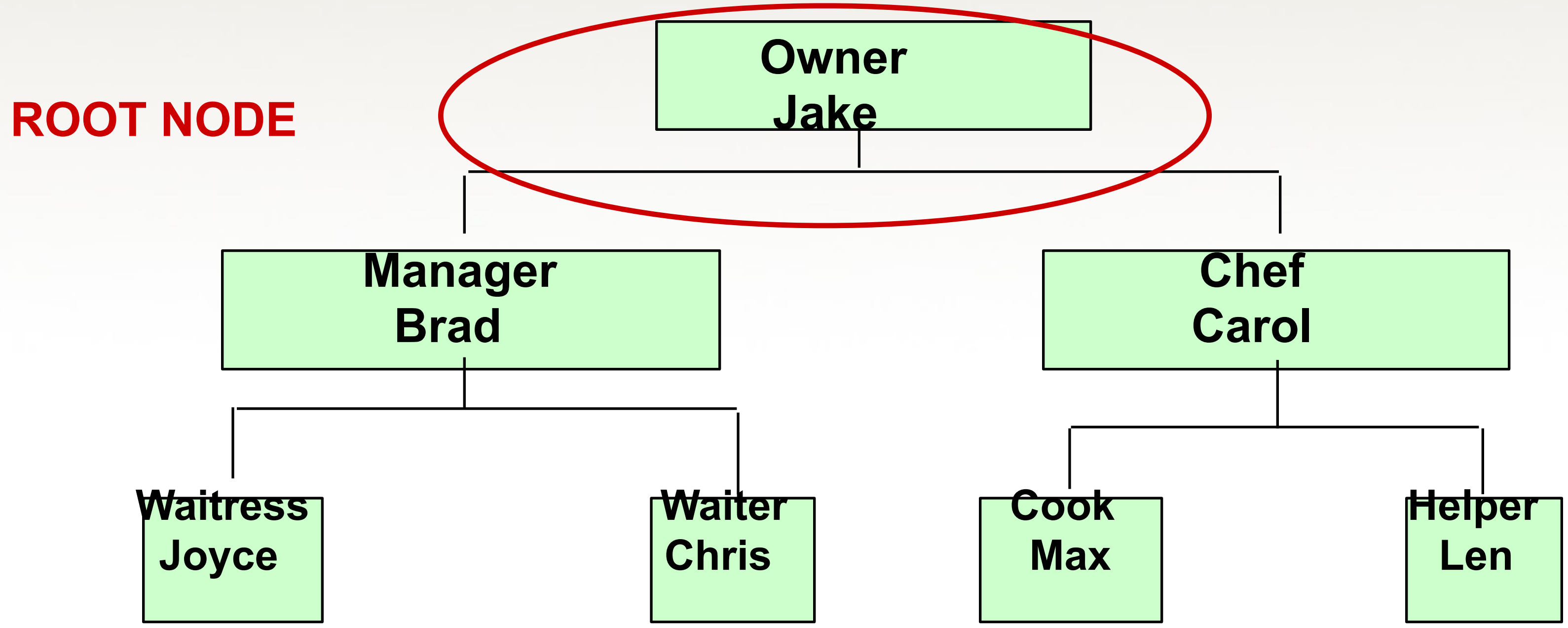
FIFTH EDITION

Chapter 8
Binary Search Trees

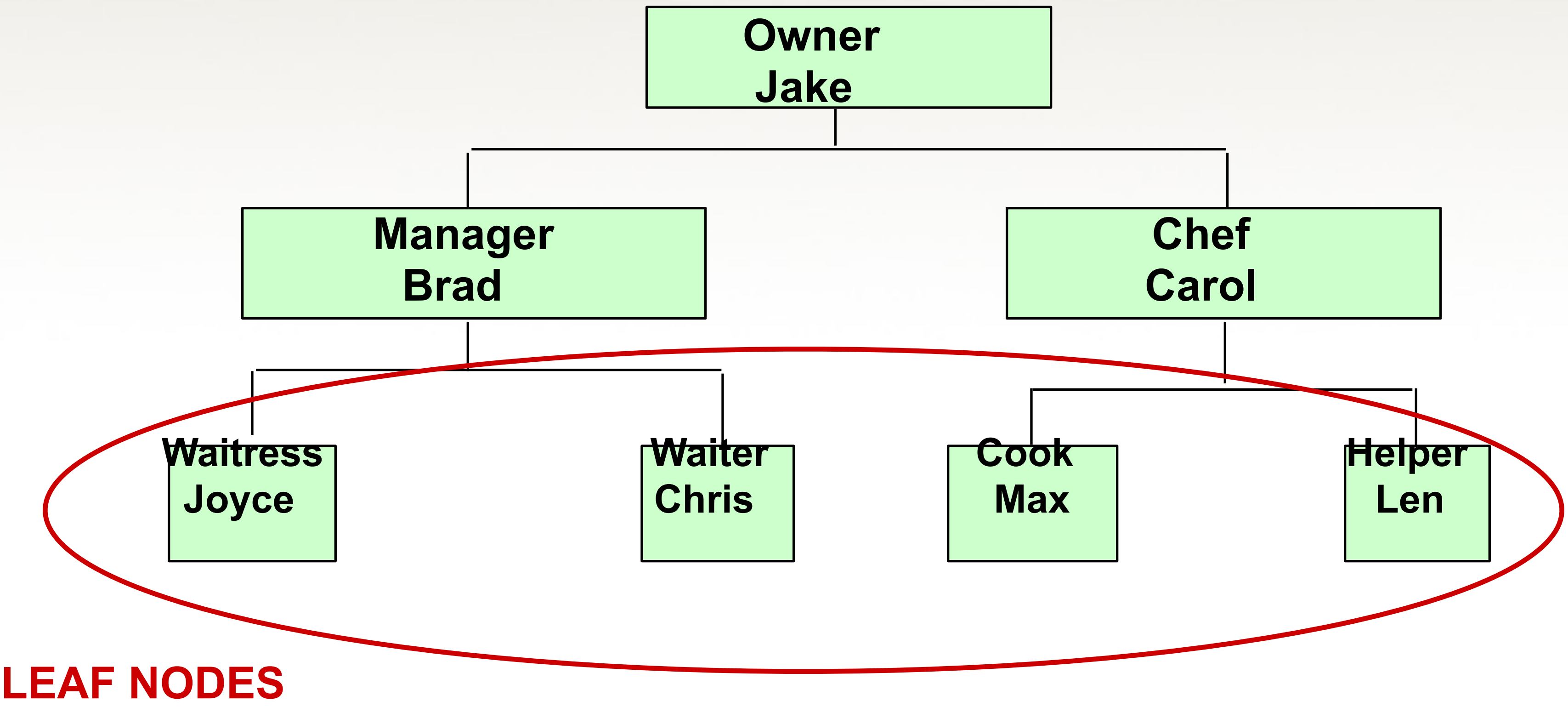
Jake's Pizza Shop



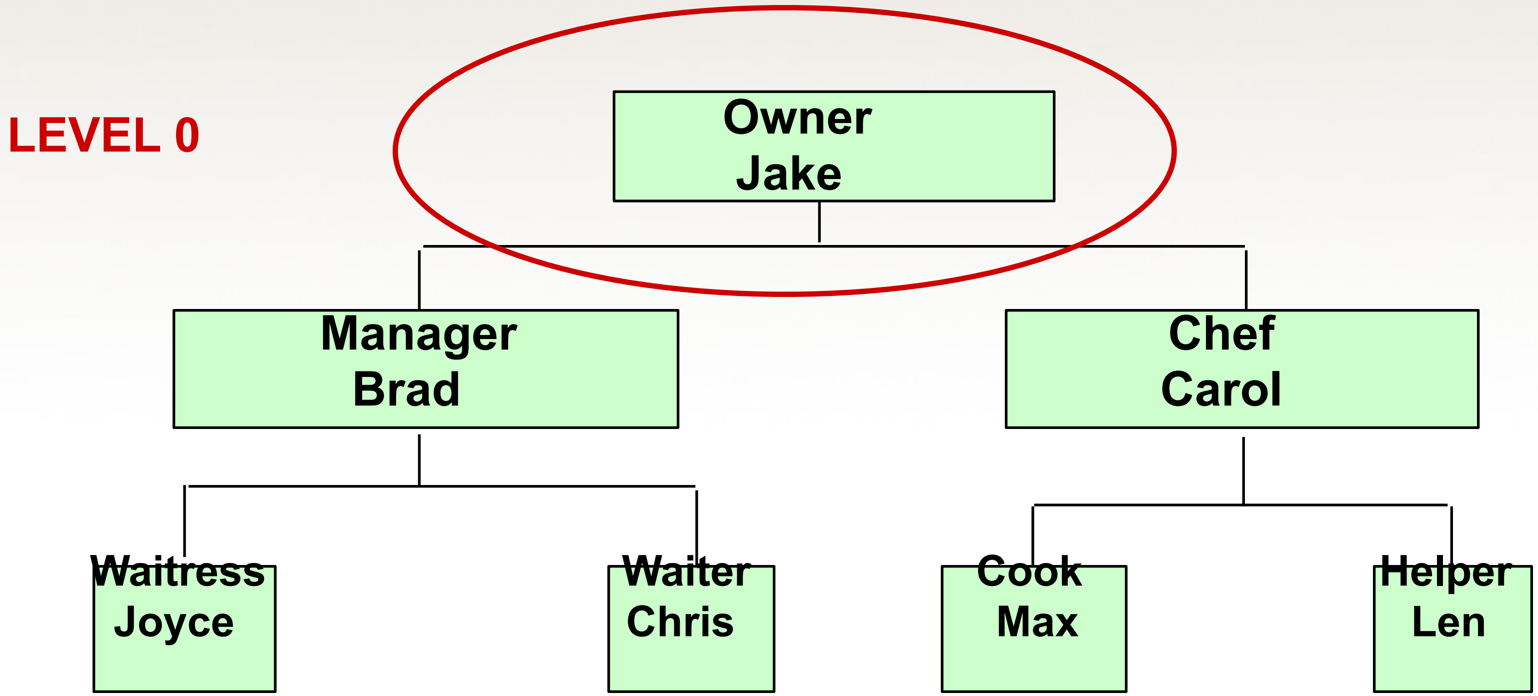
A Tree Has a Root Node



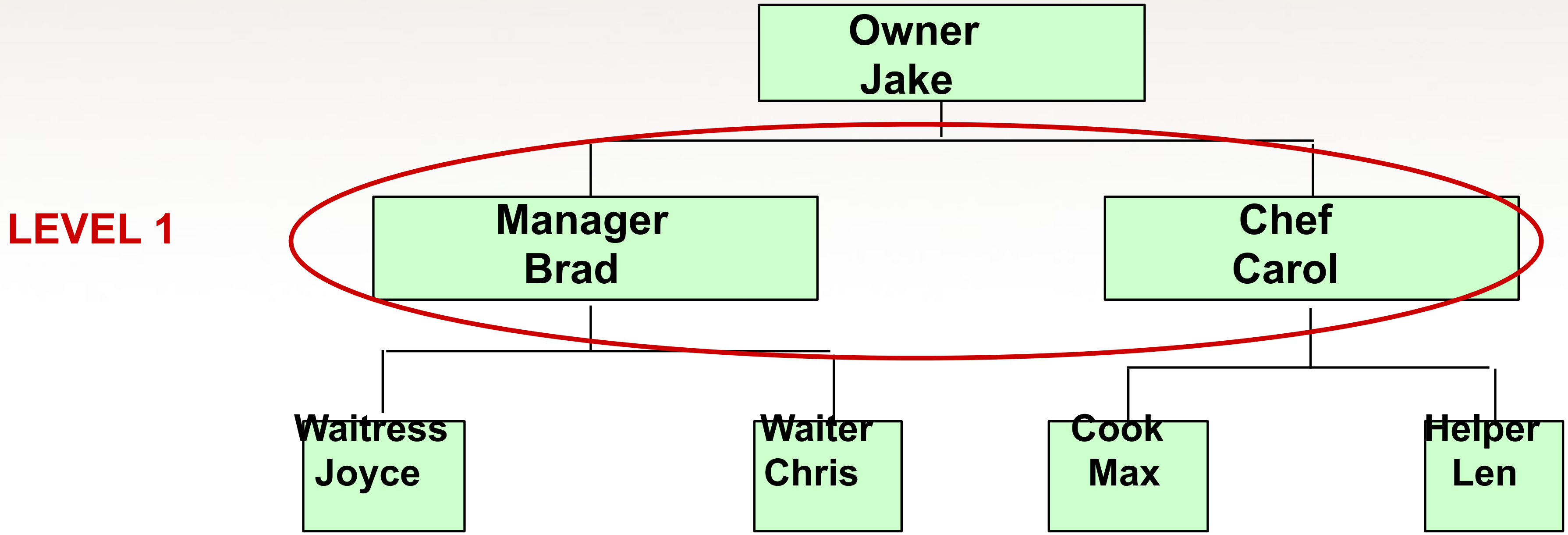
Leaf Nodes have No Children



A Tree Has Leaves

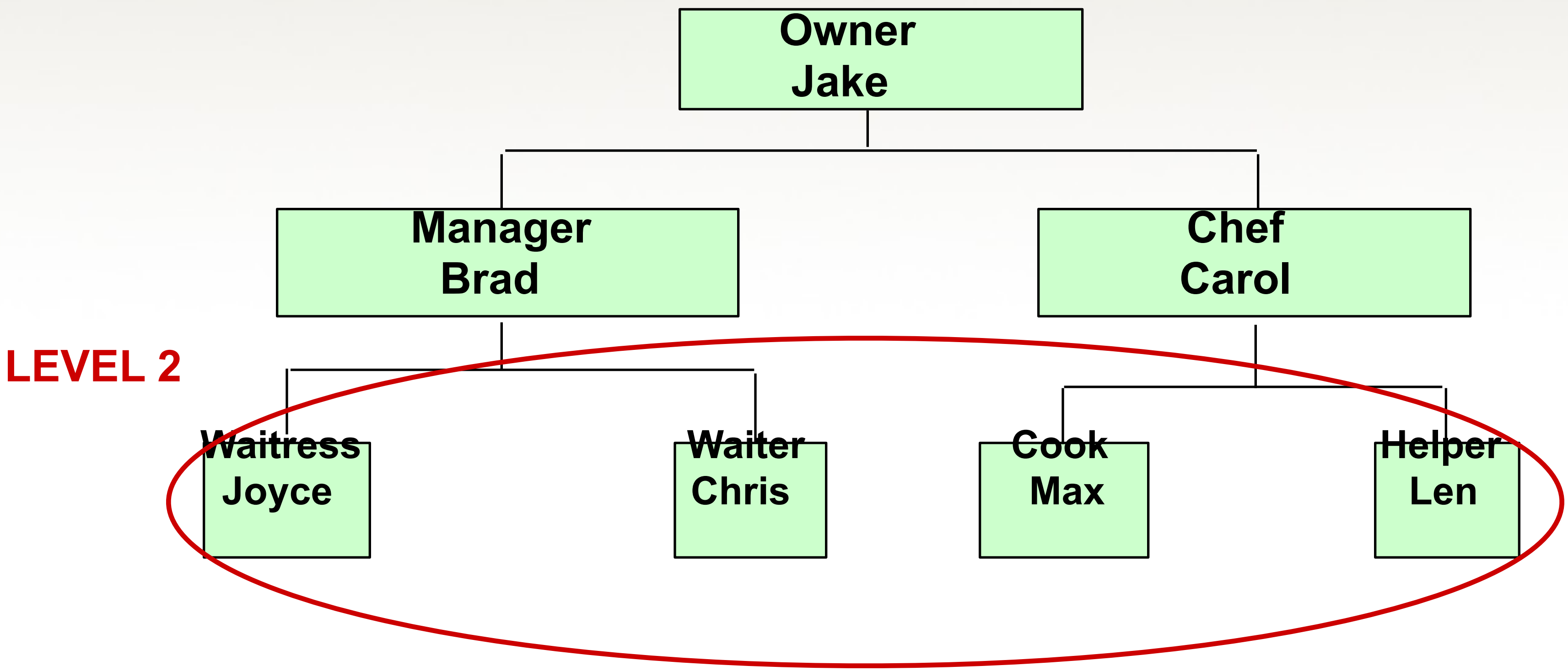


Level One



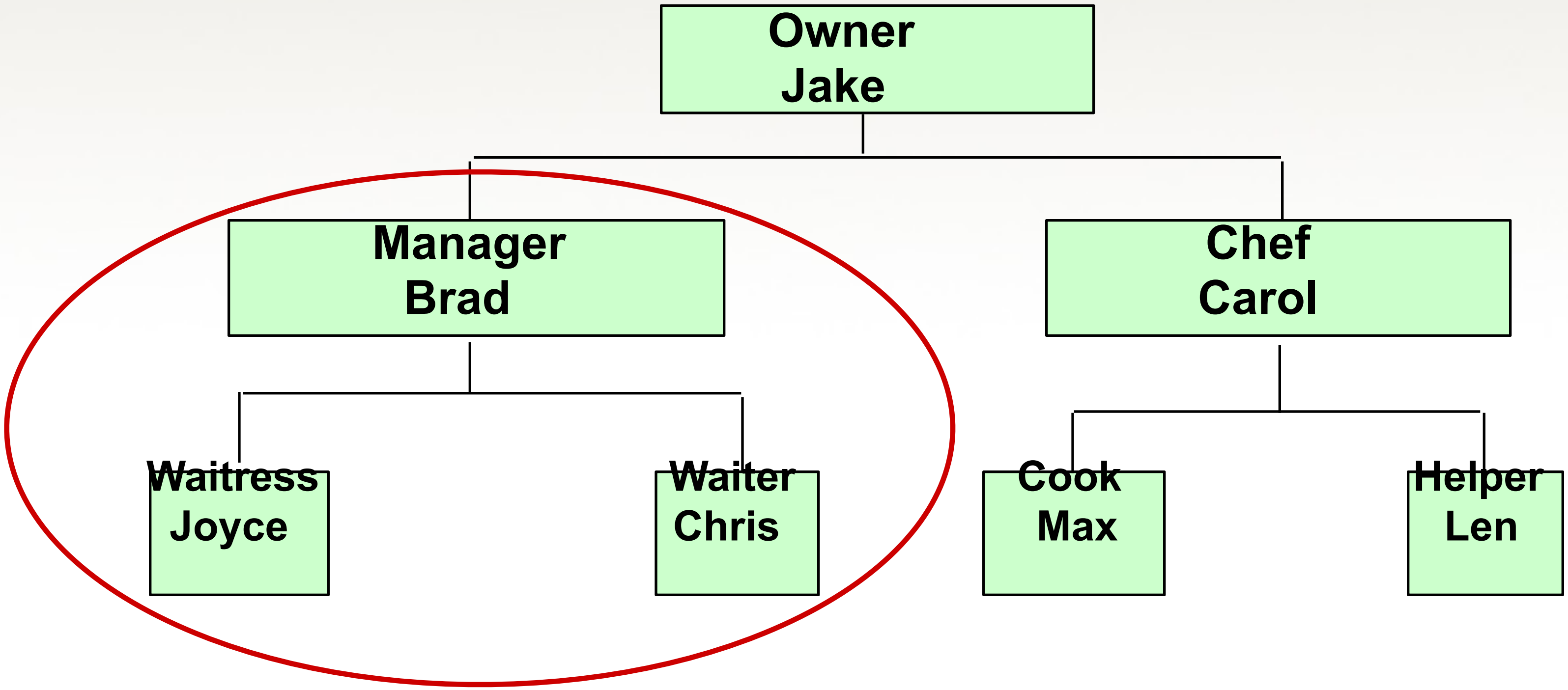
LEVEL 1
Owner
Jake
Manager
Brad
Chef
Carol
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

Level Two



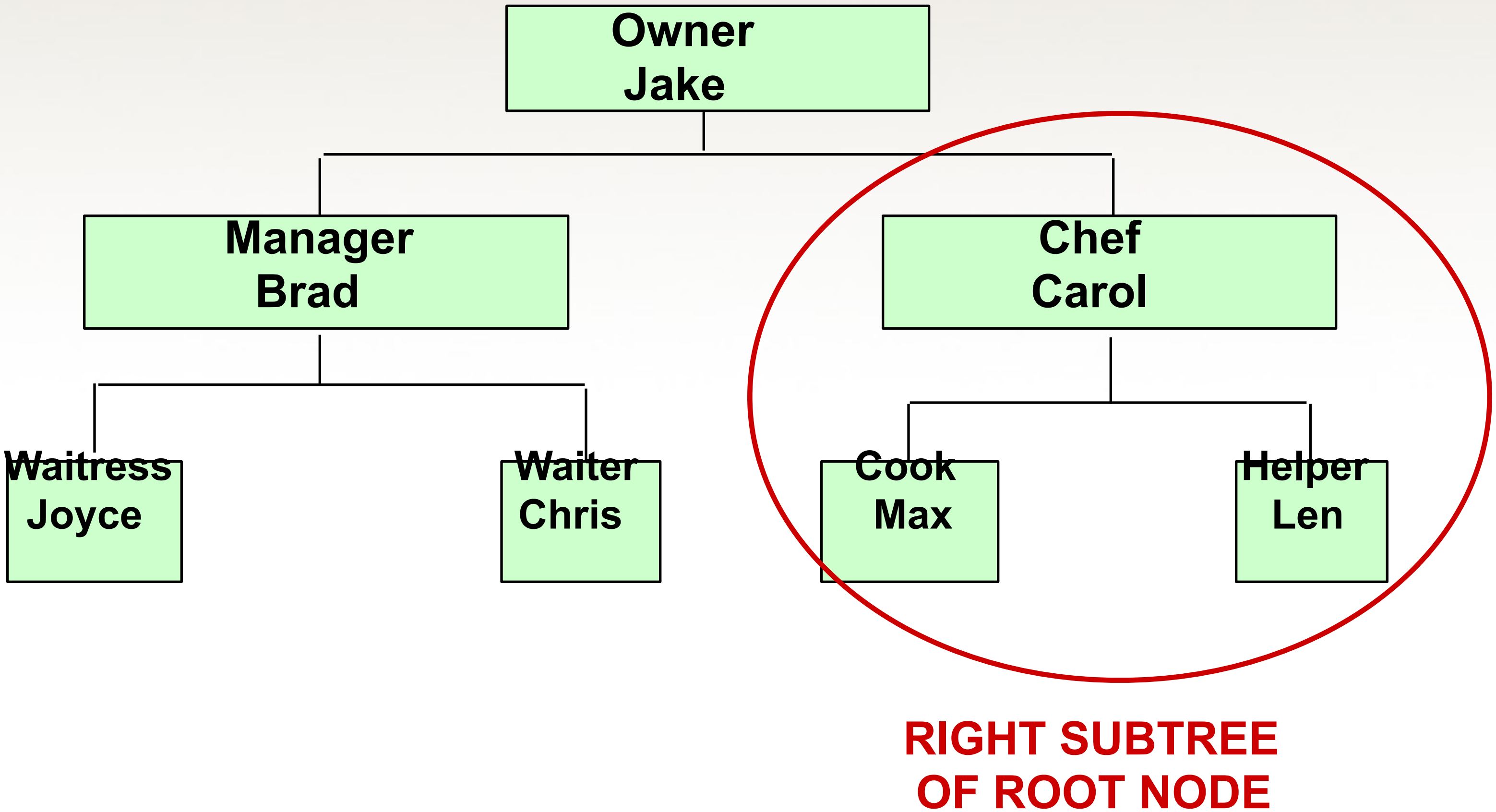
Owner
Jake
Manager
Brad
Chef
Carol
LEVEL 2
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

A Subtree



LEFT SUBTREE OF ROOT NODE

Another Subtree



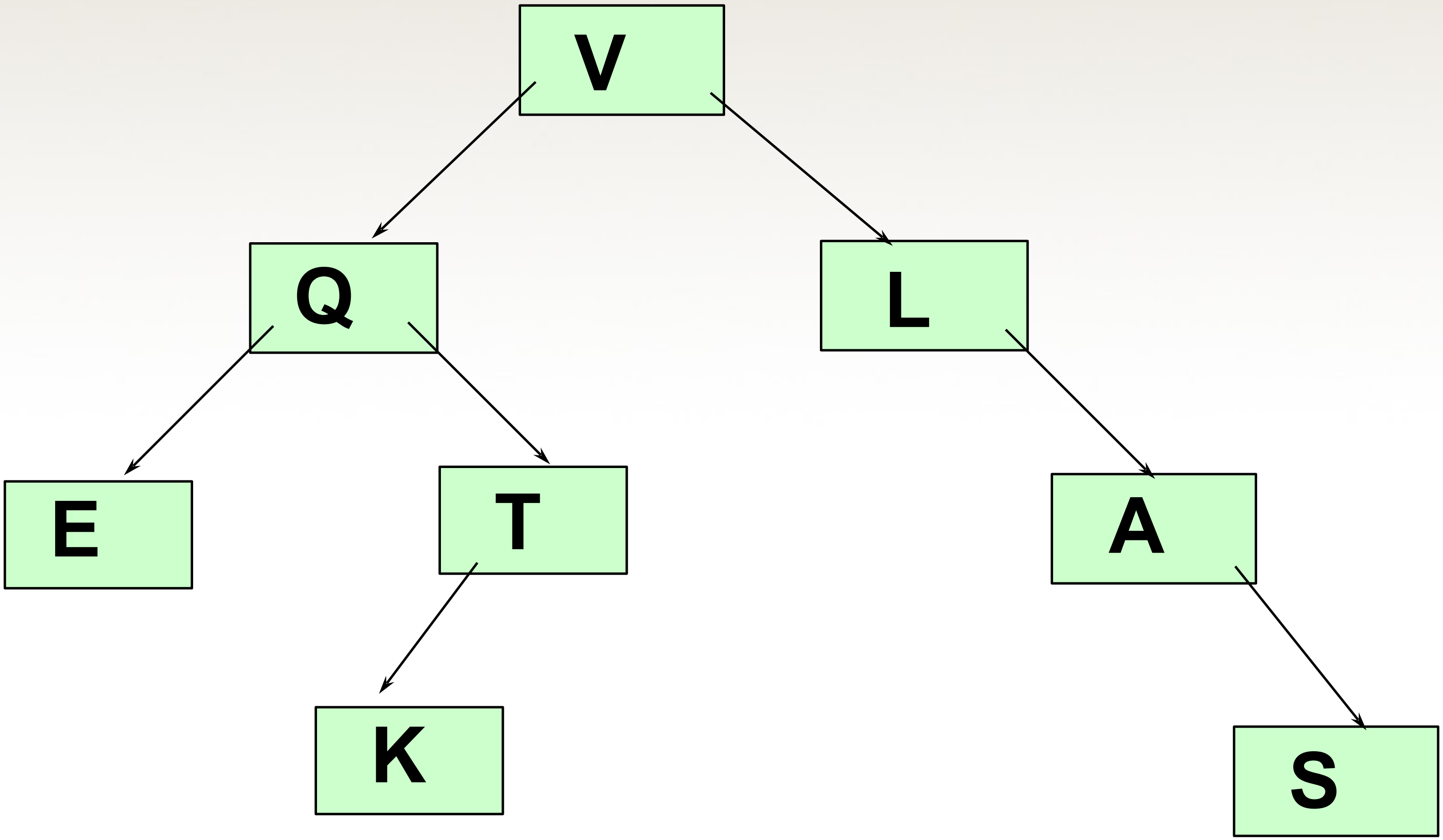
Binary Tree

A binary tree is a structure in which:

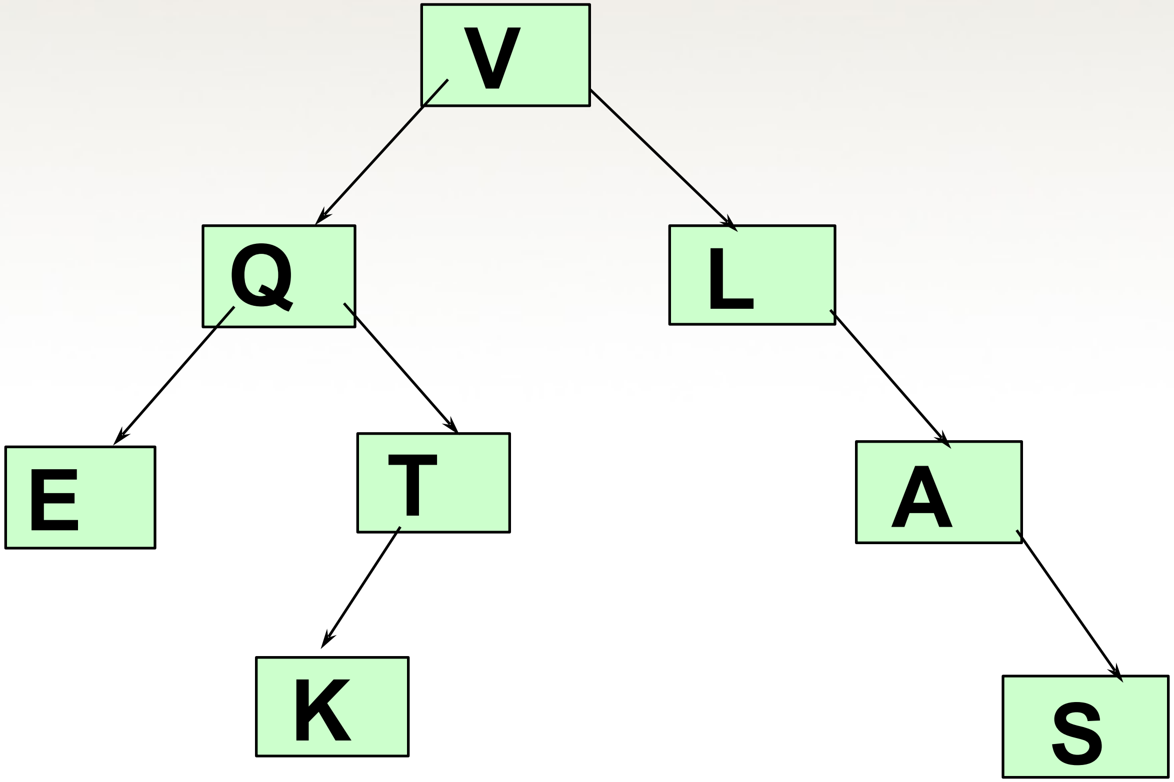
Each node can have at most two children, and in which a unique path exists from the root to every other node.

The two children of a node are called the left child and the right child, if they exist.

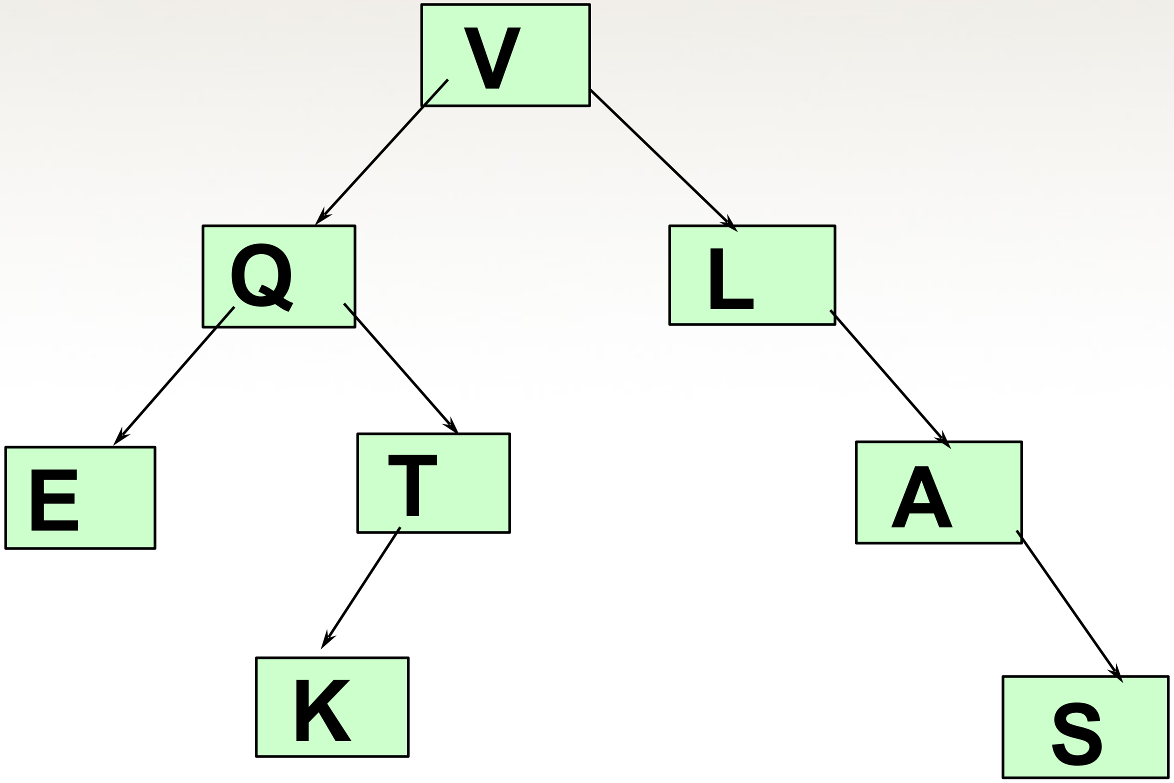
A Binary Tree



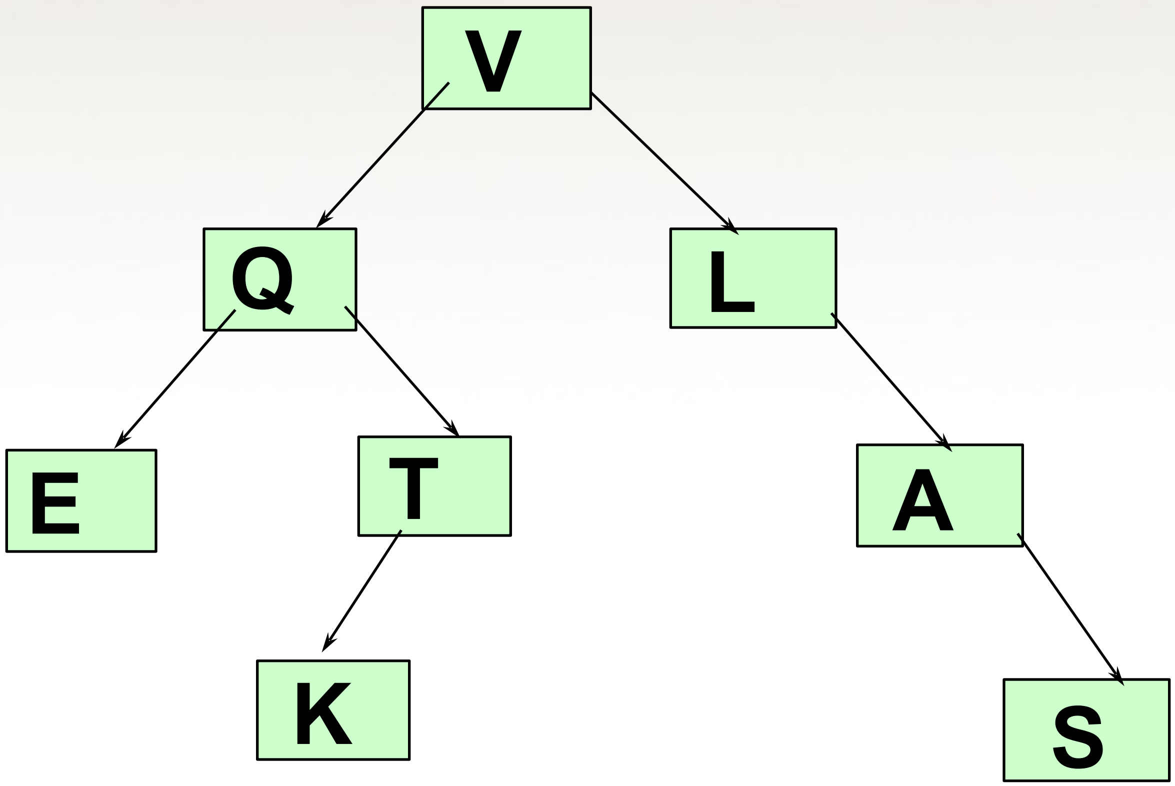
How many leaf nodes?



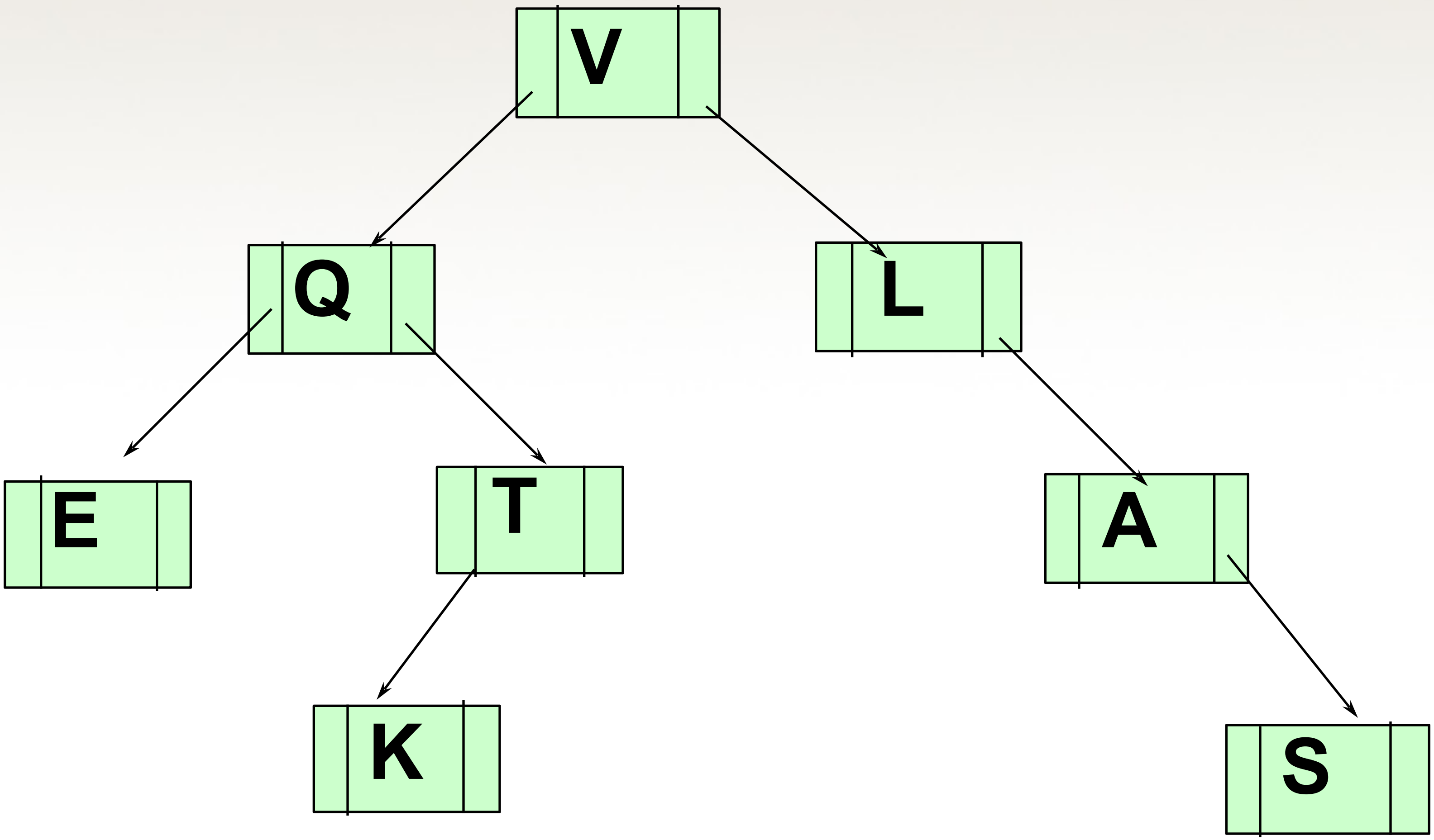
How many descendants of Q?



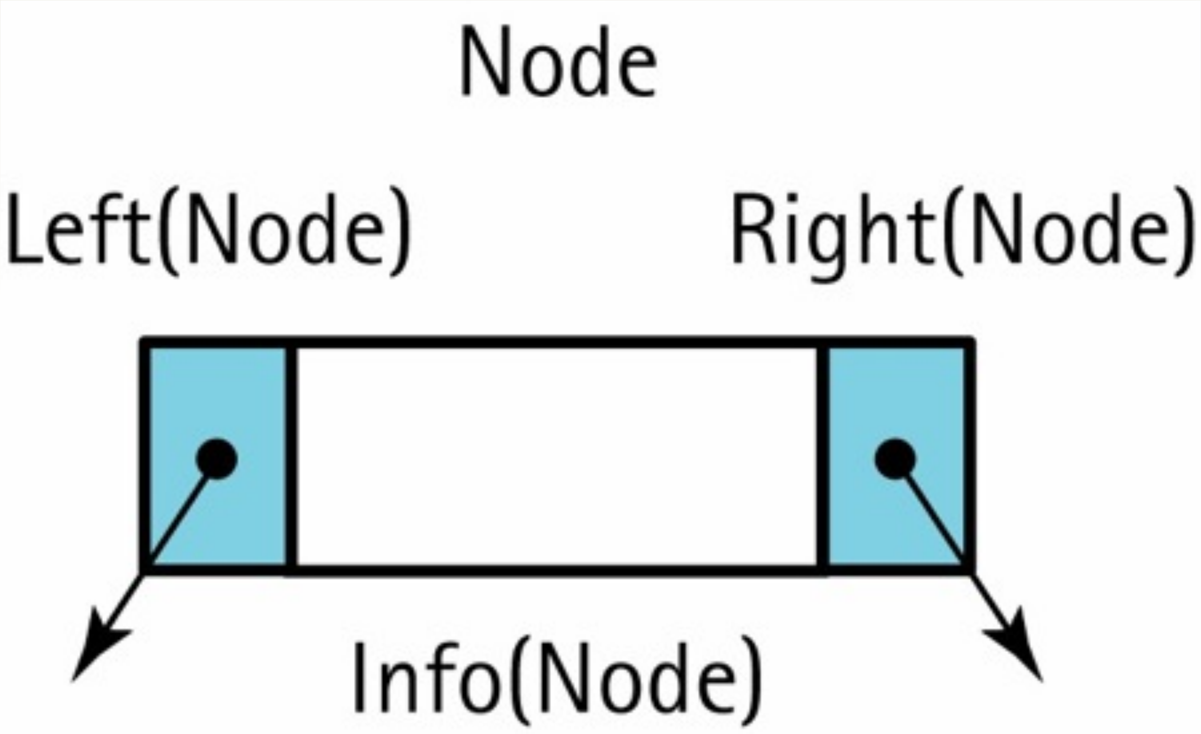
How many ancestors of K?



Implementing a Binary Tree with Pointers and Dynamic Data



Node Terminology for a Tree Node



A Binary Search Tree (BST) is . . .

A special kind of binary tree in which:

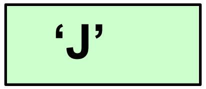
1. Each node contains a distinct data value,
2. The key values in the tree can be compared using "greater than" and "less than", and
3. The key value of each node in the tree is
   less than every key value in its right subtree, and
   greater than every key value in its left subtree.

Shape of a binary search tree . . .

Depends on its key values and their order of insertion.

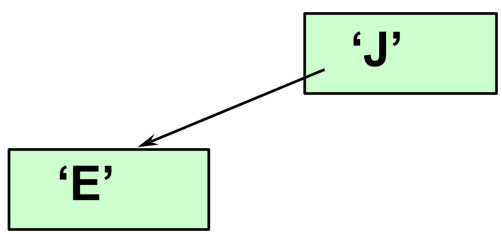
Insert the elements 'J' 'E' 'F' 'T' 'A' in that order.

The first value to be inserted is put into the root node.



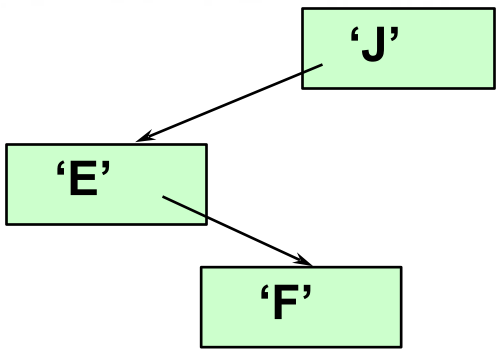
Inserting 'E' into the BST

Thereafter, each value to be inserted begins by comparing itself to the value in the root node, moving left it is less, or moving right if it is greater. This continues at each level until it can be inserted as a new leaf.



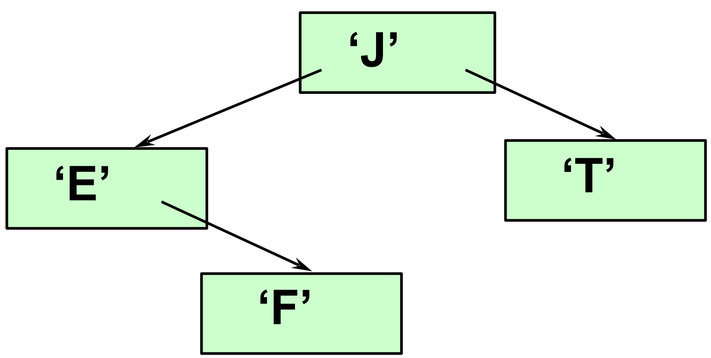
Inserting ‘F’ into the BST

Begin by comparing ‘F’ to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



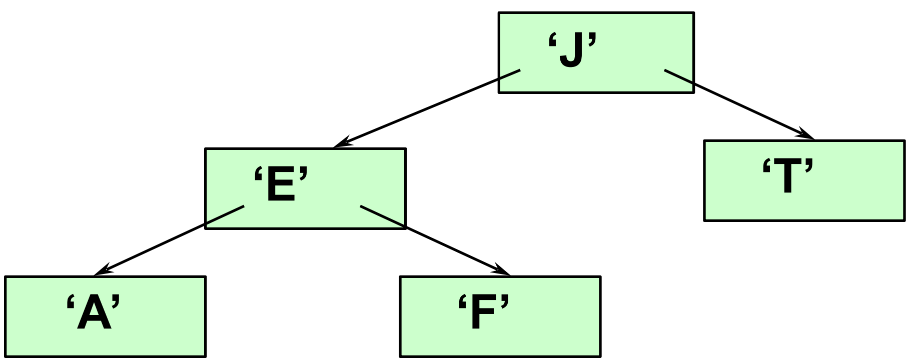
Inserting 'T' into the BST

Begin by comparing 'T' to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



Inserting 'A' into the BST

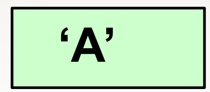
Begin by comparing 'A' to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



What binary search tree . . .

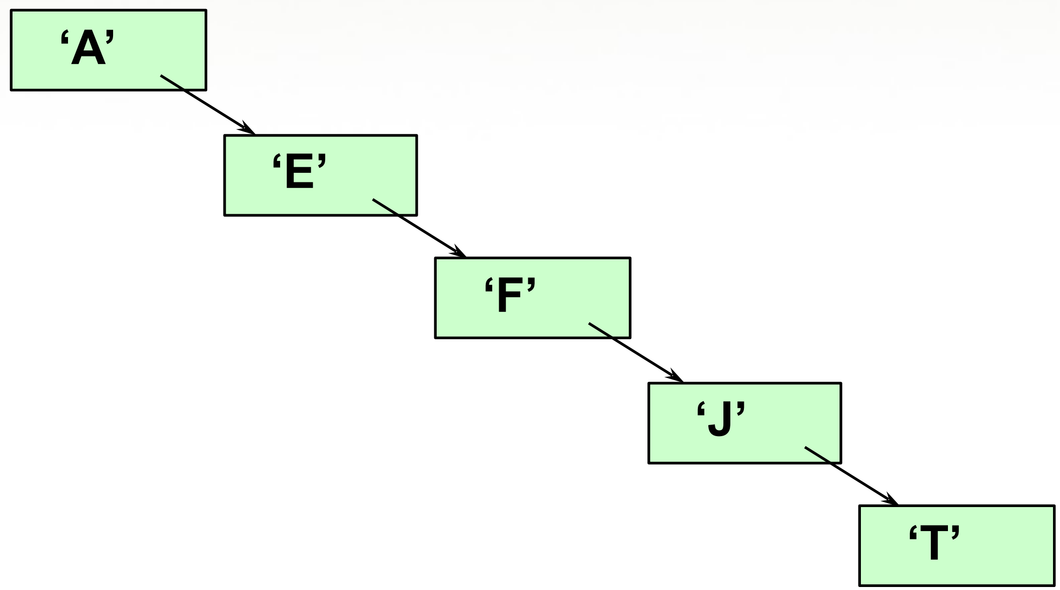
is obtained by inserting

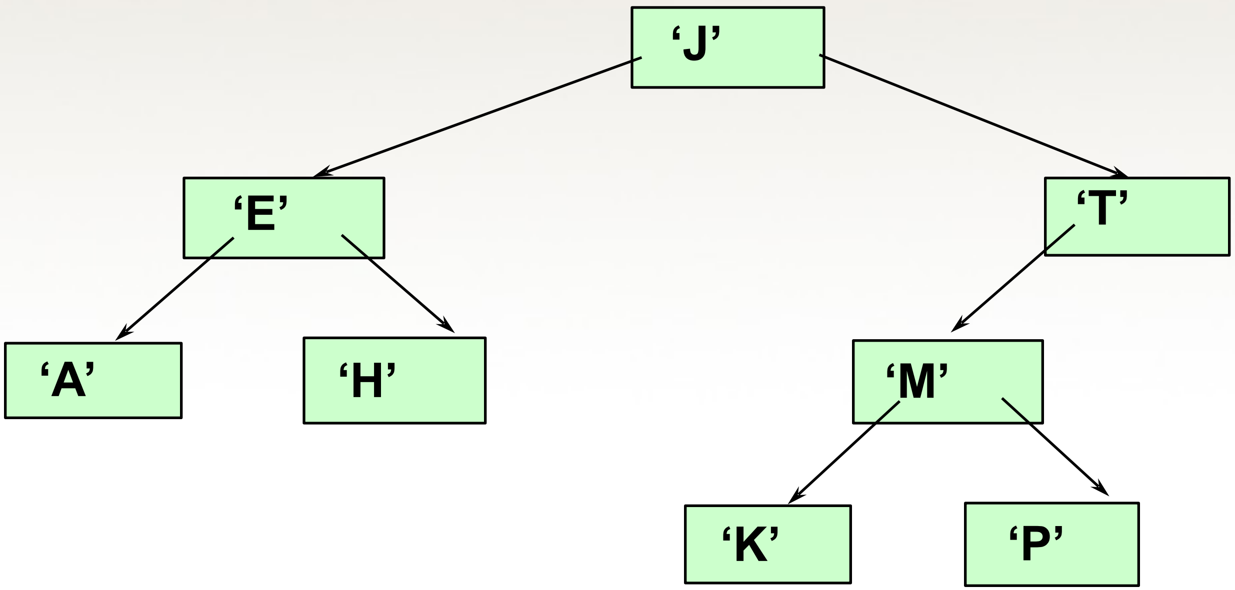
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order?



obtained by inserting

the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order.

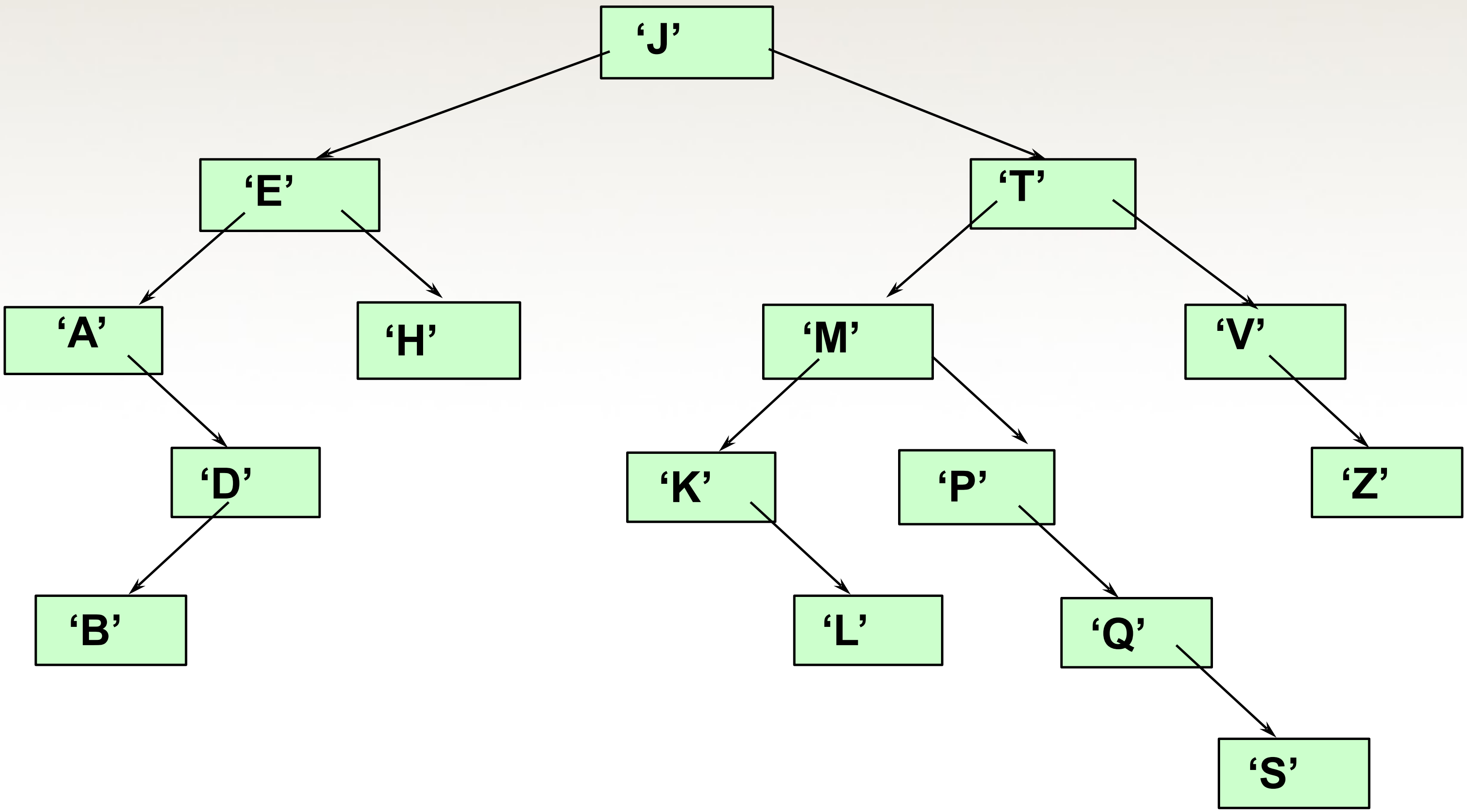




Add nodes containing these values in this order:

'D' 'B' 'L' 'Q' 'S' 'V' 'Z'

Is ‘F’ in the binary search tree?



Class TreeType

```
// Assumptions:  Relational operators overloaded
 class TreeType
 {
 public:
   // Constructor, destructor, copy constructor
   ...
   // Overloads assignment
   ...
   // Observer functions
   ...
   // Transformer functions
   ...
   // Iterator pair
   ...
   void Print(std::ofstream& outFile) const;
 private:
   TreeNode* root;
 };
```

```
bool TreeType::IsFull() const
{
  NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
}

bool TreeType::IsEmpty() const
{
  return root == NULL;
}
```

Tree Recursion

CountNodes Version 1

```
if (Left(tree) is NULL) AND (Right(tree) is NULL)
    return 1
else
    return CountNodes(Left(tree)) +
           CountNodes(Right(tree)) + 1
```

What happens when Left(tree) is NULL?

Tree Recursion

CountNodes Version 2

```
if (Left(tree) is NULL) AND (Right(tree) is NULL)
    return 1
else if Left(tree) is NULL
    return CountNodes(Right(tree)) + 1
else if Right(tree) is NULL
    return CountNodes(Left(tree)) + 1
else return CountNodes(Left(tree)) +
    CountNodes(Right(tree)) + 1
```

What happens when the initial tree is NULL?

Tree Recursion

CountNodes Version 3

```
if tree is NULL
    return 0
else if (Left(tree) is NULL) AND (Right(tree) is NULL)
    return 1
else if Left(tree) is NULL
    return CountNodes(Right(tree)) + 1
else if Right(tree) is NULL
    return CountNodes(Left(tree)) + 1
else return CountNodes(Left(tree)) +
    CountNodes(Right(tree)) + 1
```

Can we simplify this algorithm?

Tree Recursion

CountNodes Version 4

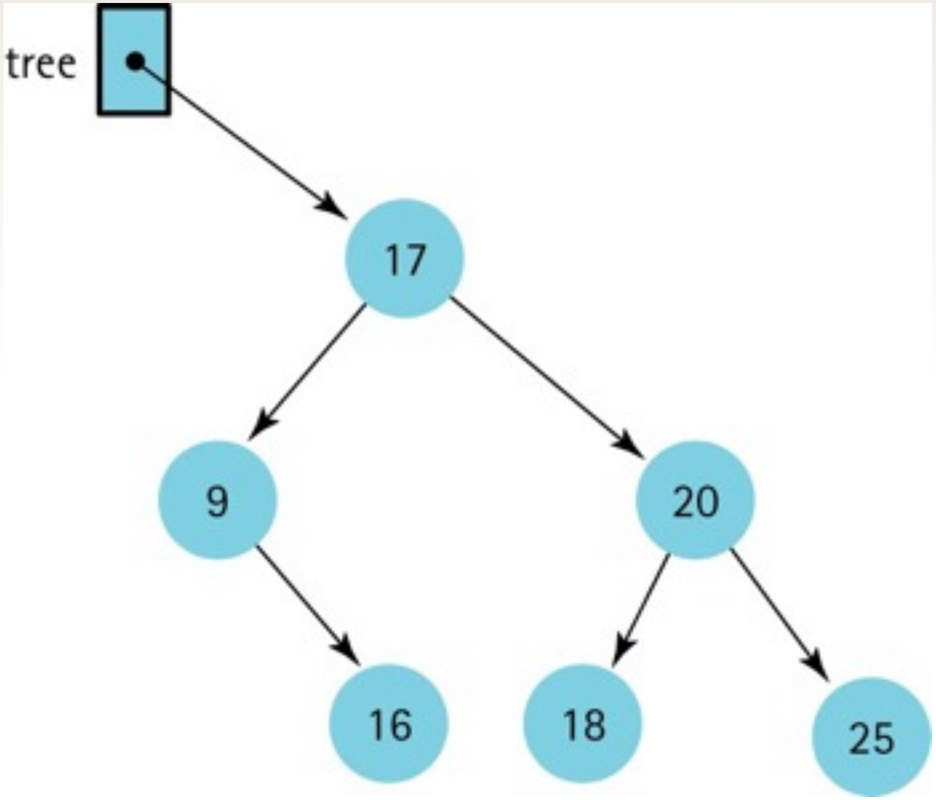
```
if tree is NULL
    return 0
else
    return CountNodes(Left(tree)) +
           CountNodes(Right(tree)) + 1
```

Is that all there is?

```
// Implementation of Final Version
int CountNodes(TreeNode* tree); // Pototype
int TreeType::GetLength() const
// Class member function
{
  return CountNodes(root);
}

int CountNodes(TreeNode* tree)
// Recursive function that counts the nodes
{
  if (tree == NULL)
    return 0;
  else
    return CountNodes(tree->left) +
      CountNodes(tree->right) + 1;
}
```

Retrieval Operation



Retrieval Operation

```
void TreeType::GetItem(ItemType& item, bool& found)
{
  Retrieve(root, item, found);
}

void Retrieve(TreeNode* tree,
     ItemType& item, bool& found)
{
  if (tree == NULL)
    found = false;
  else if (item < tree->info)
    Retrieve(tree->left, item, found);
```

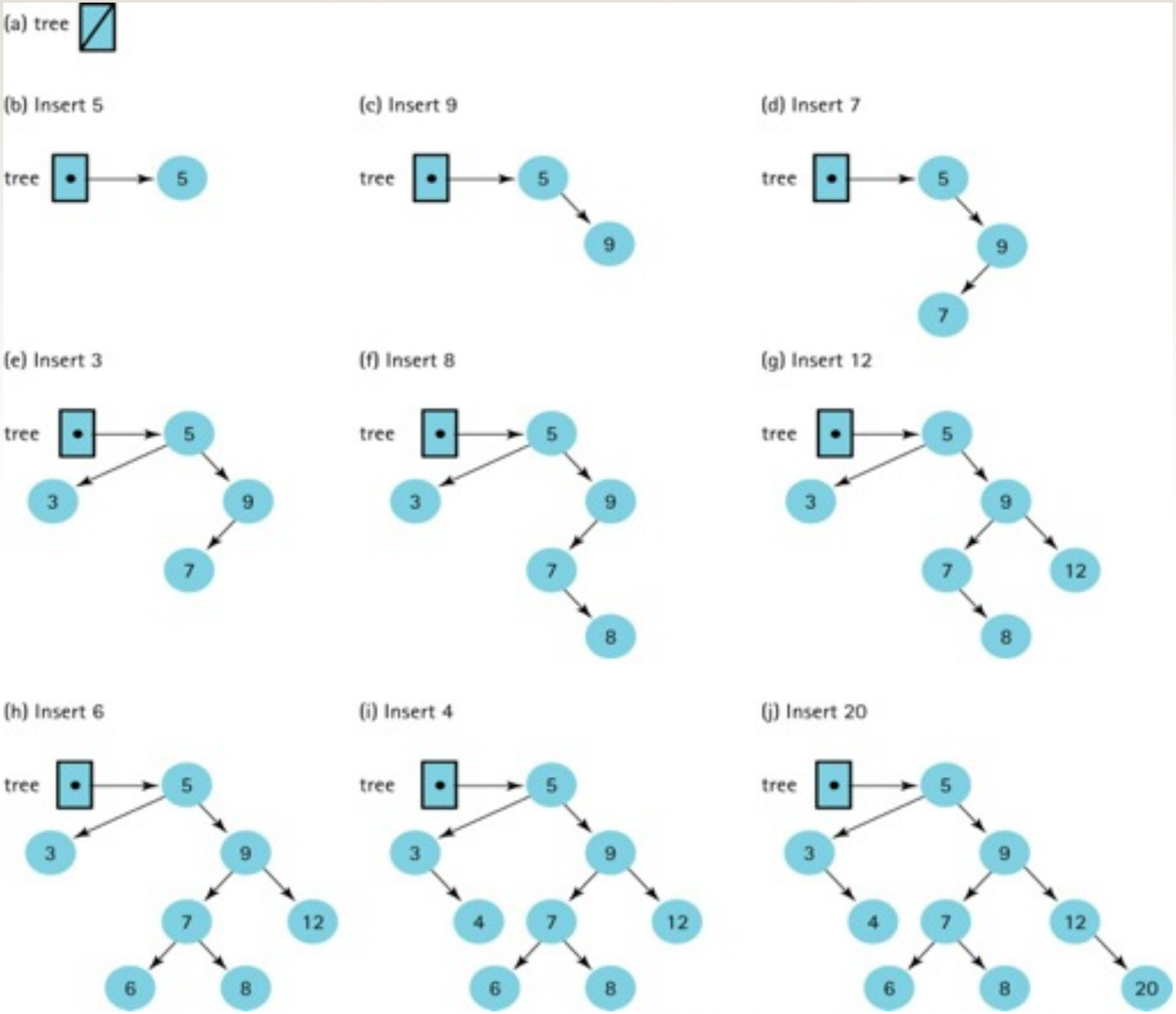
Retrieval Operation, cont.

```
else if (item > tree->info)
    Retrieve(tree->right, item, found);
  else
  {
    item = tree->info;
    found = true;
   }
}
```

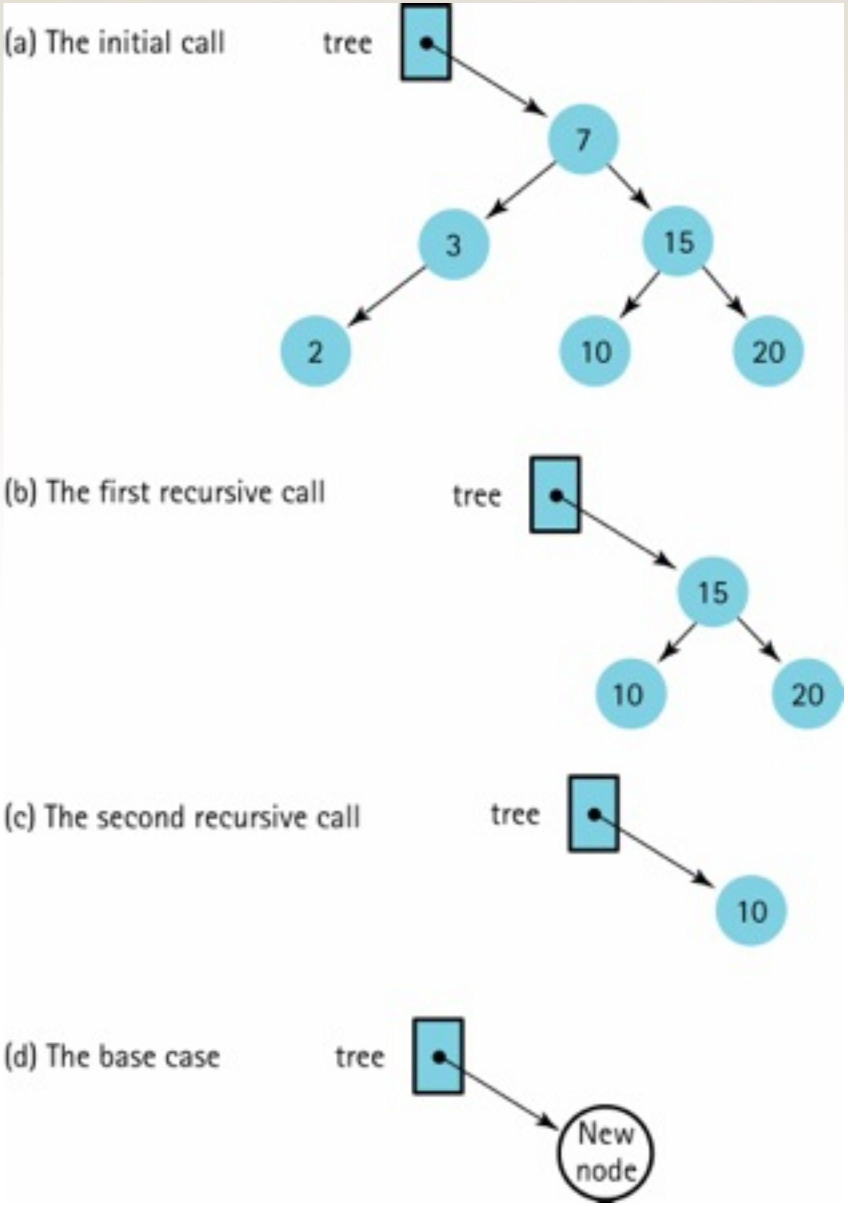
The Insert Operation

A new node is always inserted into its appropriate position in the tree as a leaf.

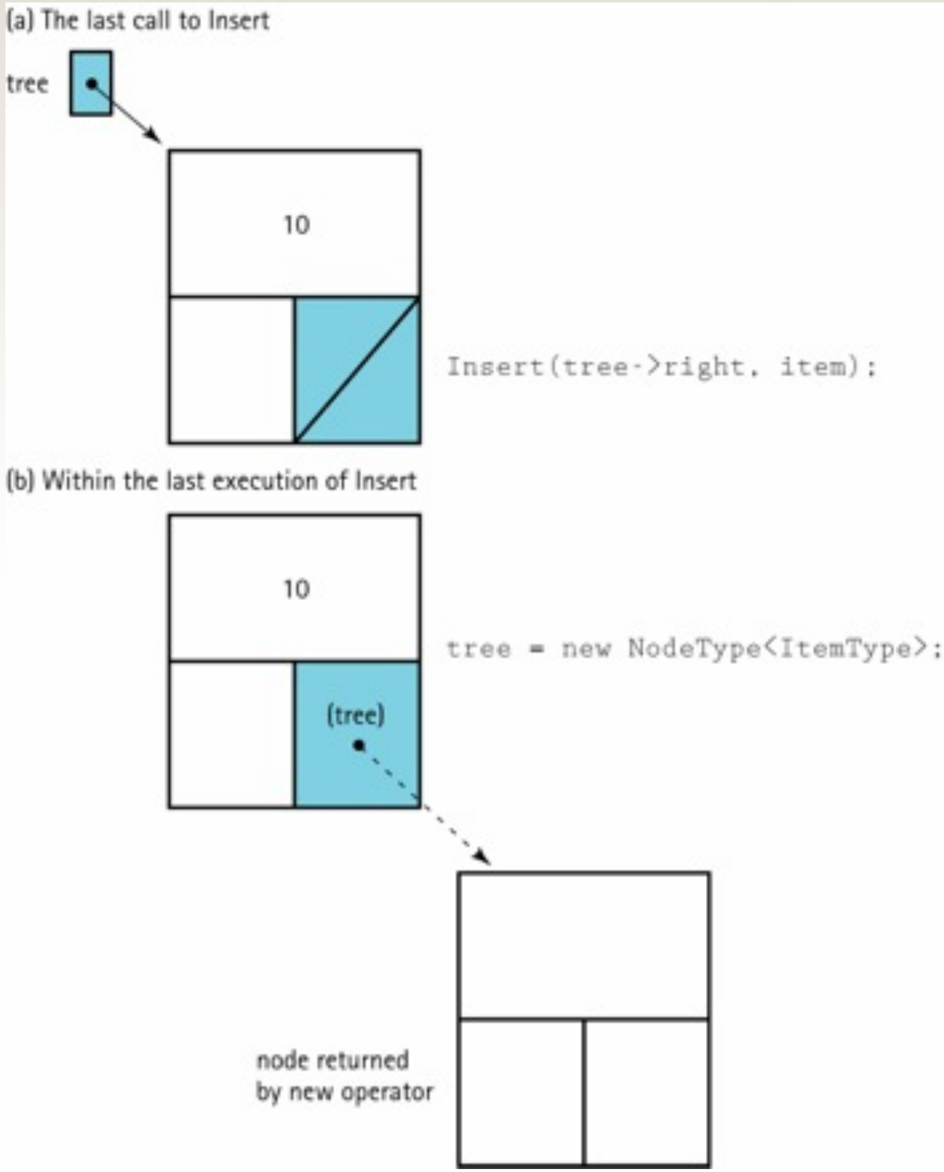
Insertions into a Binary Search Tree



The recursive Insert operation



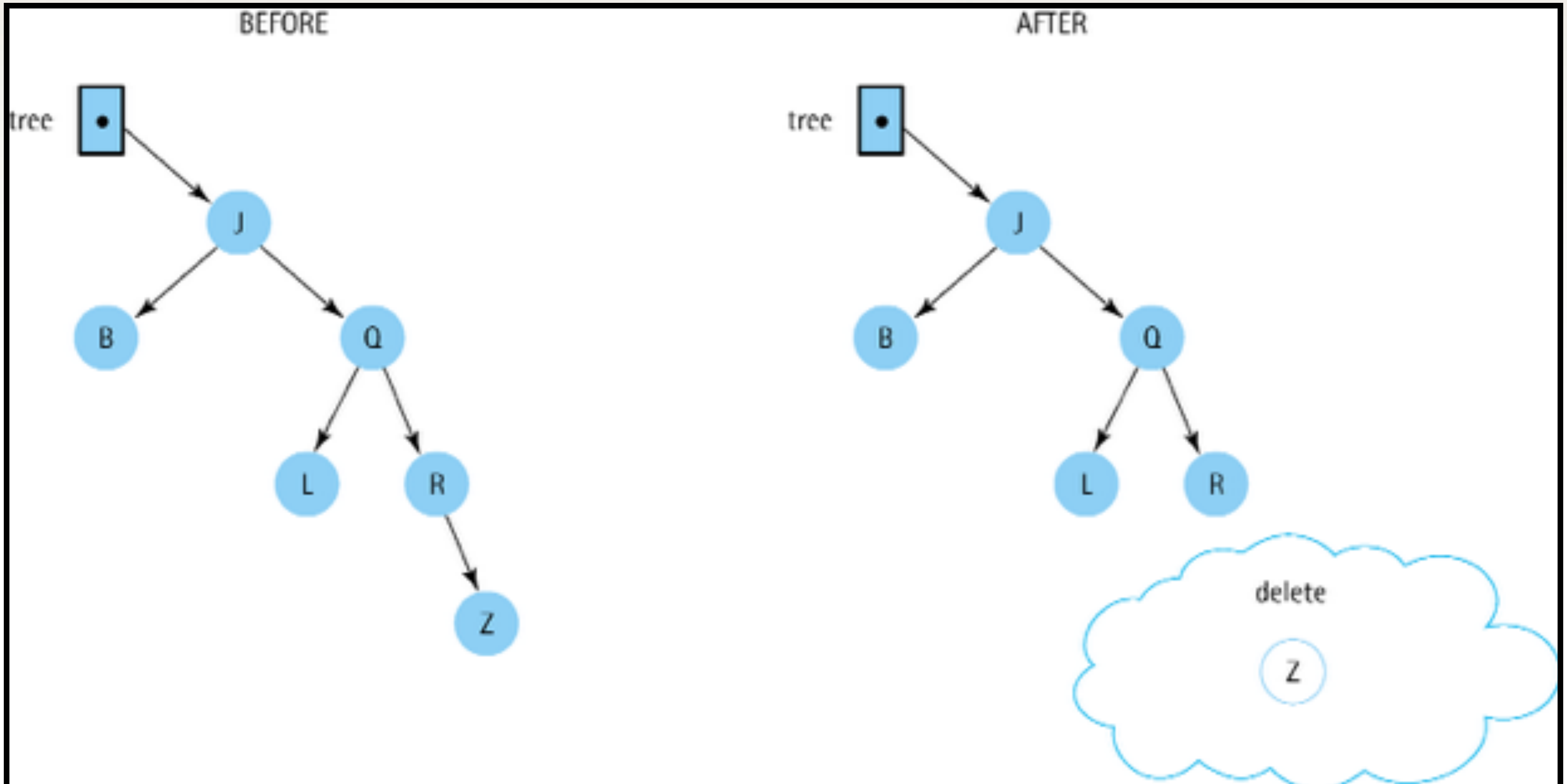
The tree parameter is a pointer within the tree



Recursive Insert

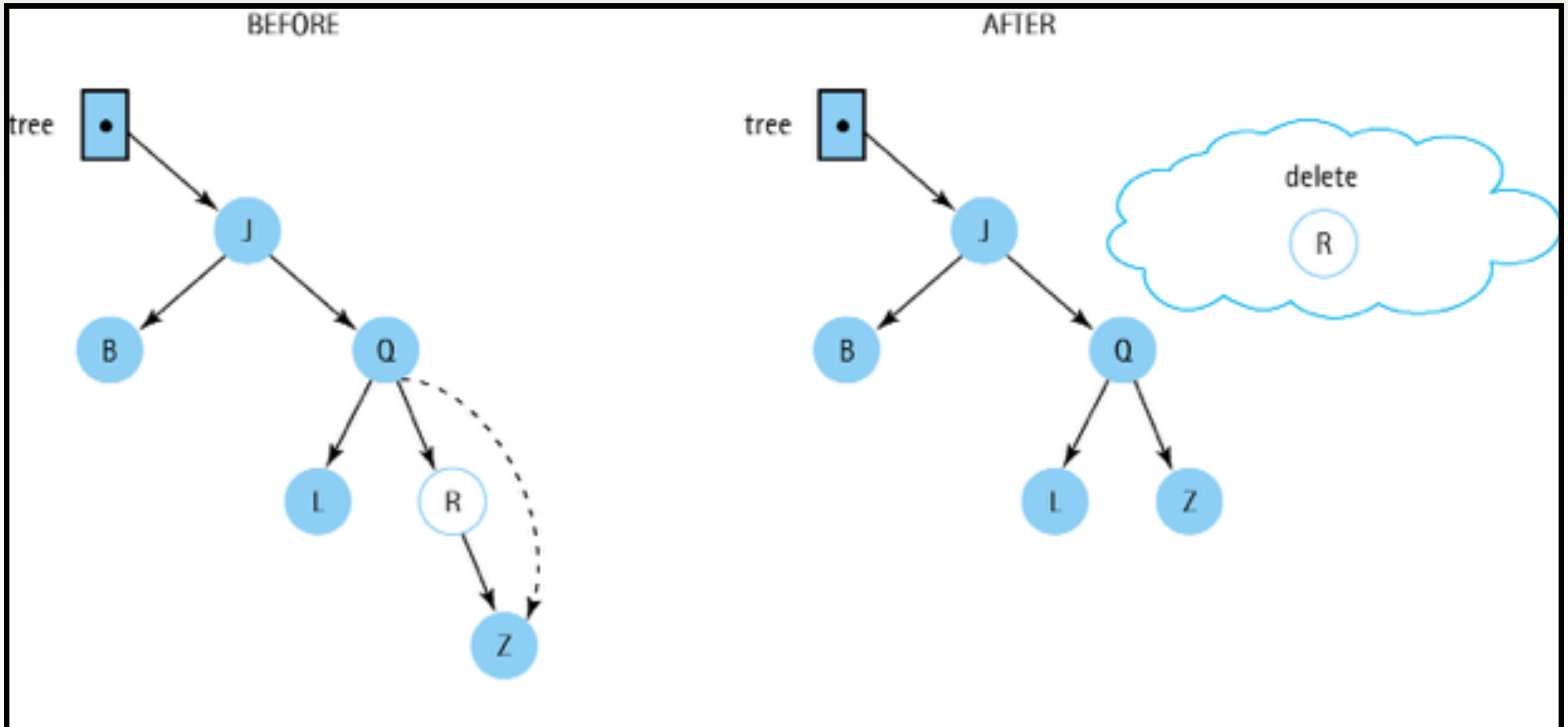
```
void Insert(TreeNode*& tree, ItemType item)
{
  if (tree == NULL)
  {// Insertion place found.
    tree = new TreeNode;
    tree->right = NULL;
    tree->left = NULL;
    tree->info = item;
  }
  else if (item < tree->info)
    Insert(tree->left, item);
  else
    Insert(tree->right, item);
}
```

Deleting a Leaf Node



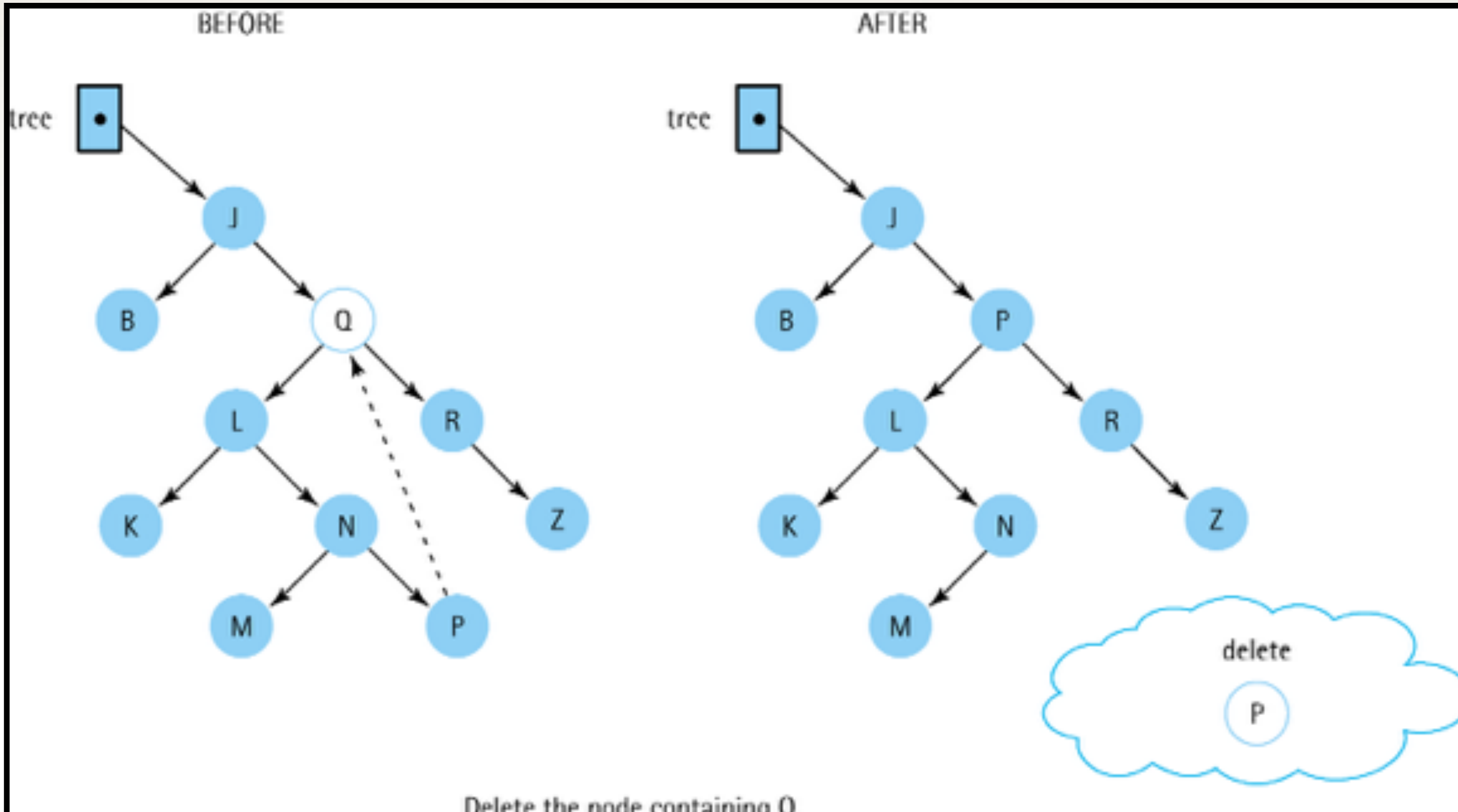
Delete the node containing Z

Deleting a Node with One Child



Delete the node containing R

Deleting a Node with Two Children



DeleteNode Algorithm

```
if (Left(tree) is NULL) AND (Right(tree) is NULL)
    Set tree to NULL
else if Left(tree) is NULL
    Set tree to Right(tree)
else if Right(tree) is NULL
    Set tree to Left(tree)
else
    Find predecessor
    Set Info(tree) to Info(predecessor)
    Delete predecessor
```

Code for DeleteNode

```
void DeleteNode(TreeNode*& tree)
{
  ItemType data;
  TreeNode* tempPtr;
  tempPtr = tree;
  if (tree->left == NULL) {
    tree = tree->right;
    delete tempPtr; }
  else if (tree->right == NULL){
    tree = tree->left;
    delete tempPtr;}
  els{
    GetPredecessor(tree->left, data);
    tree->info = data;
    Delete(tree->left, data);}
}
```

Definition of Recursive Delete

Definition: Removes item from tree

Size: The number of nodes in the path from the root to the node to be deleted.

Base Case: If item's key matches key in Info(tree), delete node pointed to by tree.

General Case: If item < Info(tree),

Delete(Left(tree), item);

else

Delete(Right(tree), item).

Code for Recursive Delete

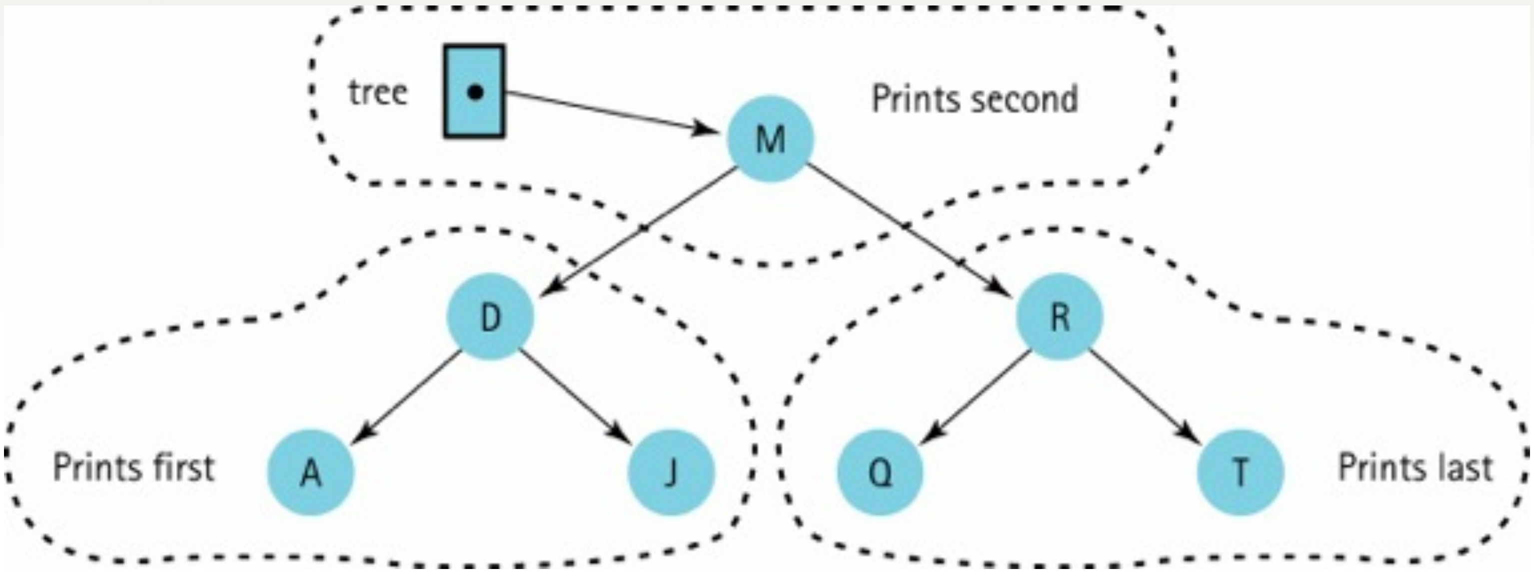
```
void Delete(TreeNode*& tree, ItemType
  item)
{
  if (item < tree->info)
    Delete(tree->left, item);
  else if (item > tree->info)
    Delete(tree->right, item);
  else
    DeleteNode(tree);  // Node found
}
```

Code for GetPredecessor

```
void GetPredecessor(TreeNode* tree,
    ItemType& data)
{
  while (tree->right != NULL)
    tree = tree->right;
  data = tree->info;
}
```

Why is the code not recursive?

Printing all the Nodes in Order



Function Print

Function Print

Definition: Prints the items in the binary search tree in order from smallest to largest.

Size: The number of nodes in the tree whose root is tree

Base Case: If tree = NULL, do nothing.

General Case: Traverse the left subtree in order.
Then print Info(tree).
Then traverse the right subtree in order.

Code for Recursive InOrder Print

```
void PrintTree(TreeNode* tree,
  std::ofstream& outFile)
{
  if (tree != NULL)
  {
    PrintTree(tree->left, outFile);
    outFile << tree->info;
    PrintTree(tree->right, outFile);
  }
}
```

Is that all there is?

Destructor

```
void Destroy(TreeNode*& tree);
TreeType::~TreeType()
{
  Destroy(root);
}

void Destroy(TreeNode*& tree)
{
  if (tree != NULL)
  {
    Destroy(tree->left);
    Destroy(tree->right);
    delete tree;
  }
}
```

Algorithm for Copying a Tree

if (originalTree is NULL)
  Set copy to NULL
else
  Set Info(copy) to Info(originalTree)
  Set Left(copy) to Left(originalTree)
  Set Right(copy) to Right(originalTree)

Code for CopyTree

```
void CopyTree(TreeNode*& copy,
     const TreeNode* originalTree)
{
  if (originalTree == NULL)
    copy = NULL;
  else
  {
    copy = new TreeNode;
    copy->info = originalTree->info;
    CopyTree(copy->left, originalTree->left);
    CopyTree(copy->right, originalTree->right);
  }
}
```

Inorder(tree)

if tree is not NULL

Inorder(Left(tree))

Visit Info(tree)

Inorder(Right(tree))

To print in alphabetical order

Postorder(tree)

```
if tree is not NULL
    Postorder(Left(tree))
    Postorder(Right(tree))
    Visit Info(tree)
```

Visits leaves first

(good for deletion)

Preorder(tree)

if tree is not NULL

Visit Info(tree)

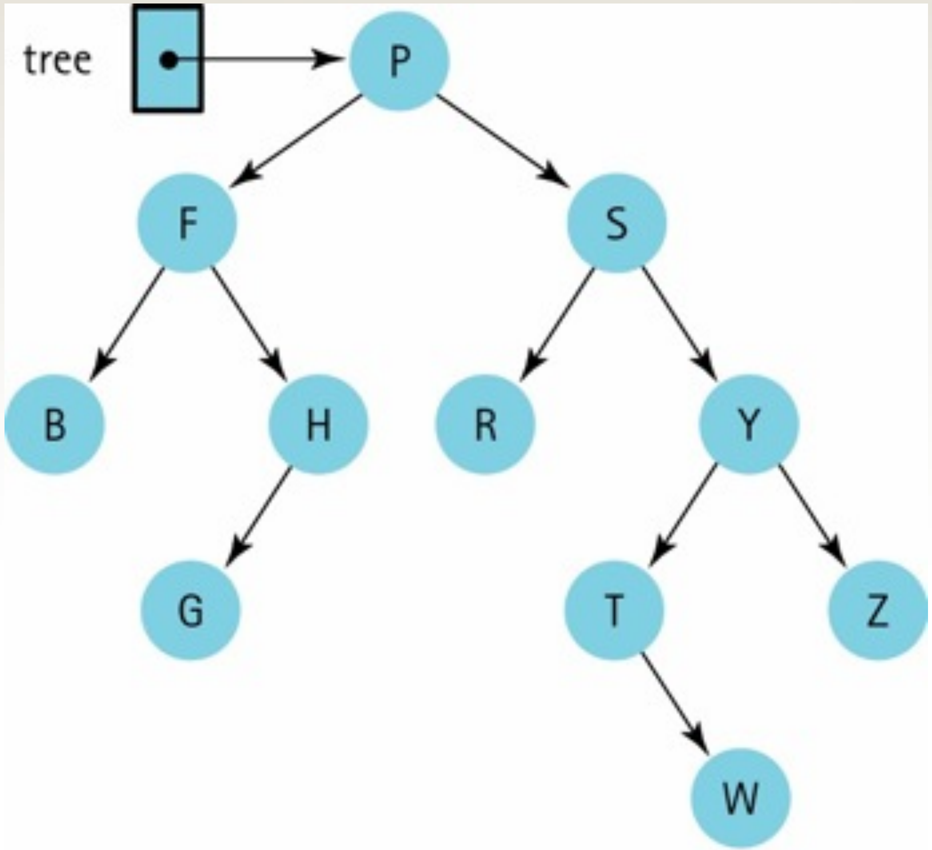
Preorder(Left(tree))

Preorder(Right(tree))

Useful with binary trees

(not binary search trees)

Three Tree Traversals



Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

Our Iteration Approach

- The client program passes the ResetTree and GetNextItem functions a parameter indicating which of the three traversals to use

- ResetTree generates a queues of node contents in the indicated order

- GetNextItem processes the node contents from the appropriate queue: inQue, preQue, postQue.

Code for `ResetTree`

```
void TreeType::ResetTree(OrderType order)
// Calls function to create a queue of the tree
// elements in the desired order.
{
  switch (order)
  {
    case PRE_ORDER : PreOrder(root, preQue);
                     break;
    case IN_ORDER  : InOrder(root, inQue);
                     break;
    case POST_ORDER: PostOrder(root, postQue);
                     break;
  }
}
```

Code for GetNextItem

```
ItemType TreeType::GetNextItem(OrderType order,bool& finished)
{
  finished = false;
  switch (order)
  {
    case PRE_ORDER  : preQue.Dequeue(item);
                      if (preQue.IsEmpty())
                        finished = true;
                      break;
    case IN_ORDER   : inQue.Dequeue(item);
                      if (inQue.IsEmpty())
                        finished = true;
                      break;
    case  POST_ORDER: postQue.Dequeue(item);
                      if (postQue.IsEmpty())
                        finished = true;
                      break;
  }
}
```

Iterative Versions

```
FindNode
Set nodePtr to tree
Set parentPtr to NULL
Set found to false

while more elements to search AND NOT found
   if item < Info(nodePtr)
        Set parentPtr to nodePtr
        Set nodePtr to Left(nodePtr)
   else if item > Info(nodePtr)
        Set parentPtr to nodePtr
        Set nodePtr to Right(nodePtr)
   else
        Set found to true
```

```
void FindNode(TreeNode* tree, ItemType item,
     TreeNode*& nodePtr, TreeNode*& parentPtr)
{
  nodePtr = tree;
  parentPtr = NULL;
  bool found = false;
  while (nodePtr != NULL && !found)
  { if (item < nodePtr->info)
    {
      parentPtr = nodePtr;
      nodePtr = nodePtr->left;
    }
    else if (item > nodePtr->info)
    {
      parentPtr = nodePtr;
      nodePtr = nodePtr->right;
    }
    else found = true;
  }
}
```

Code for FindNode

`PutItem`

Create a node to contain the new item.

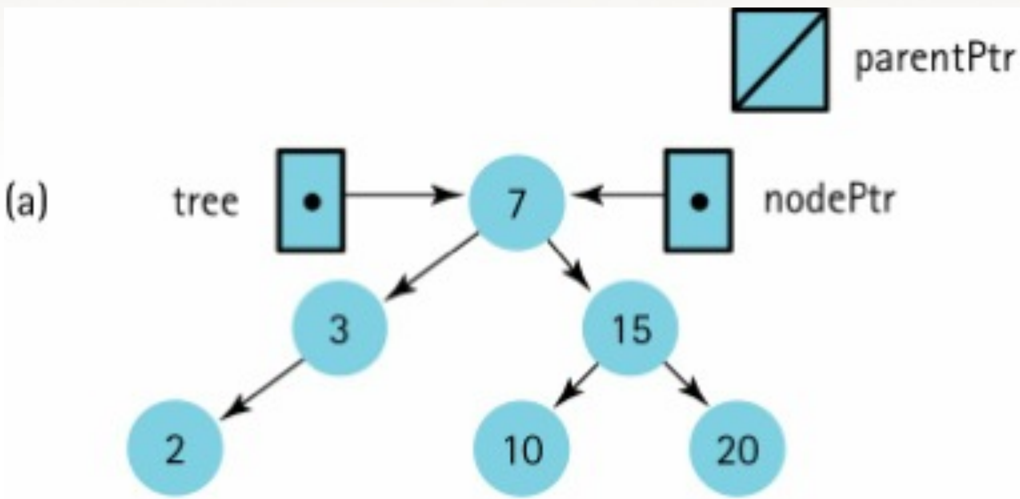
Find the insertion place.

Attach new node.

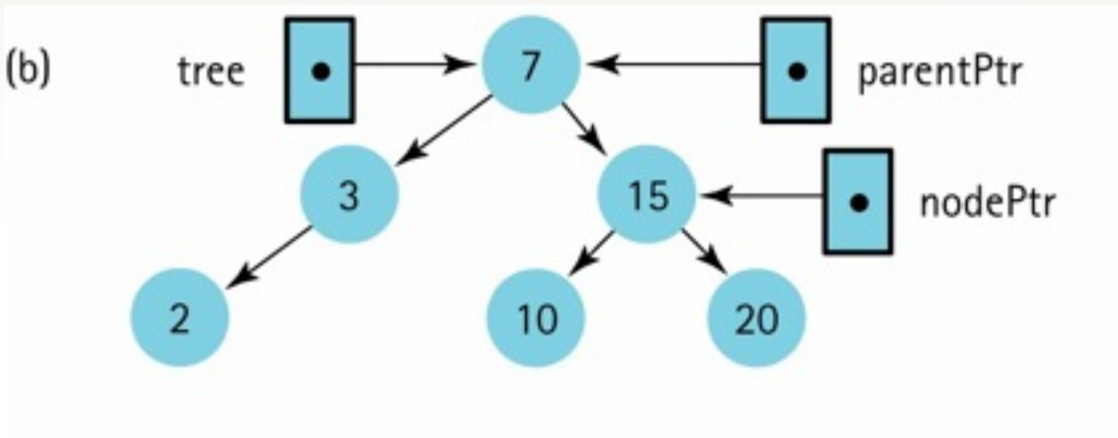
Find the insertion place

FindNode(tree, item, nodePtr, parentPtr);

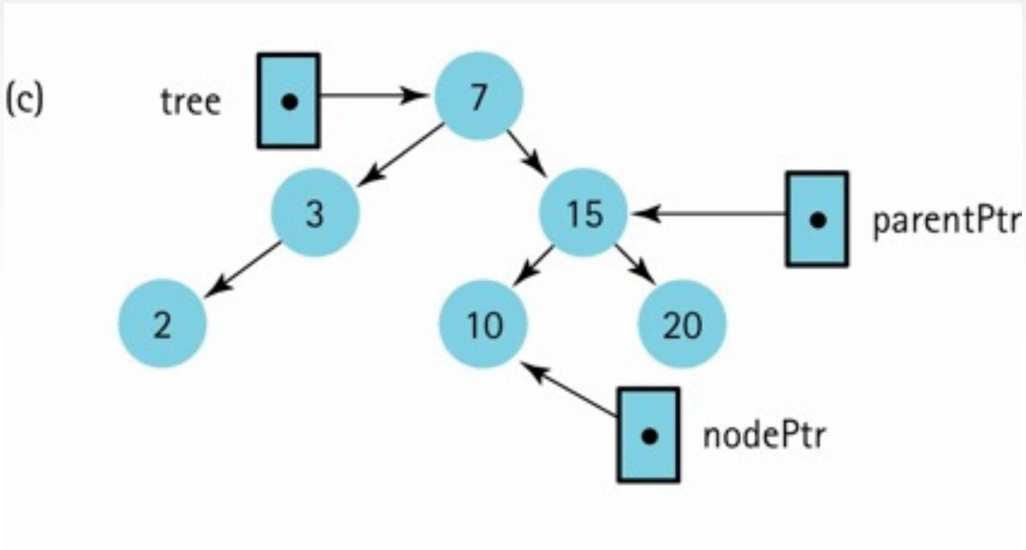
Using function FindNode to find the insertion point



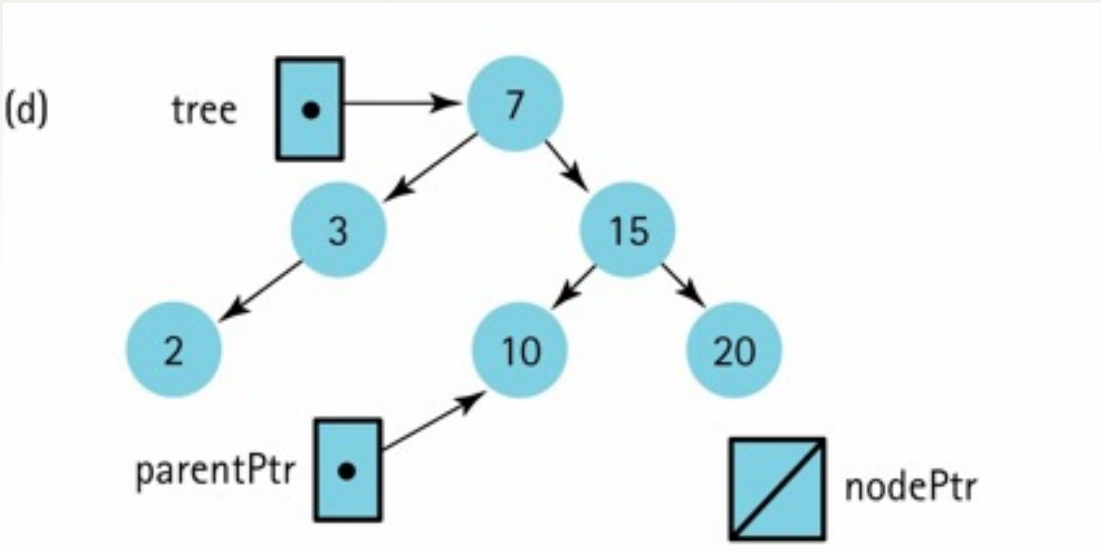
Using function FindNode to find the insertion point



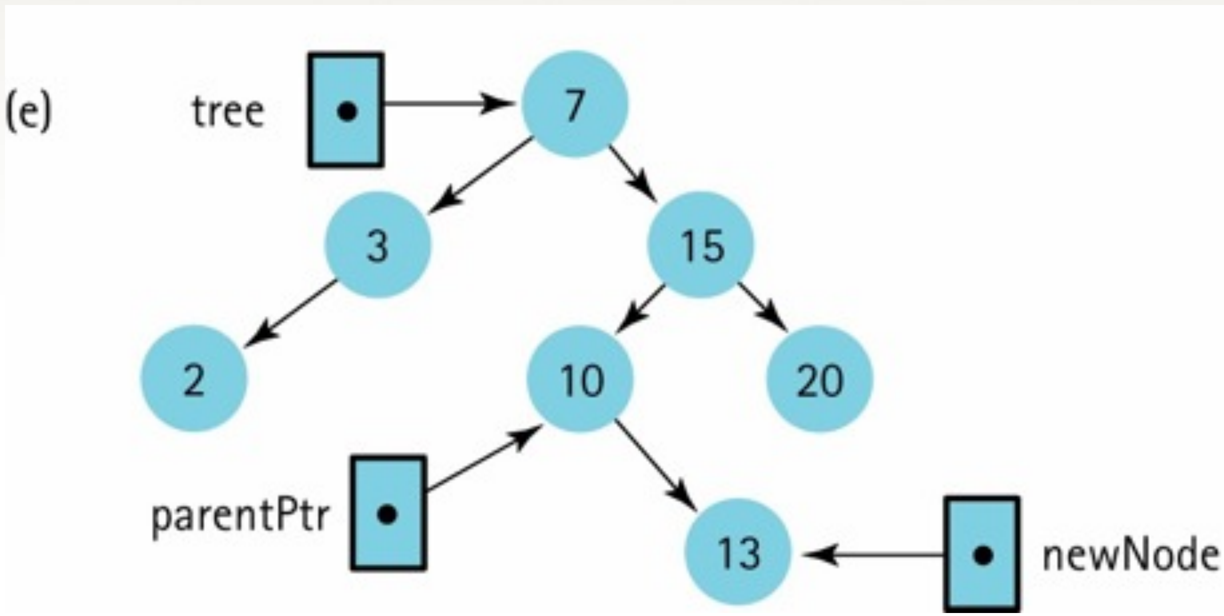
Using function FindNode to find the insertion point



Using function FindNode to find the insertion point



Using function FindNode to find the insertion point



AttachNewNode

if item < Info(parentPtr)

  Set Left(parentPtr) to newNode

else

  Set Right(parentPtr) to newNode

AttachNewNode(revised)

if parentPtr equals NULL

  Set tree to newNode

else if item < Info(parentPtr)

  Set Left(parentPtr) to newNode

else

  Set Right(parentPtr) to newNode

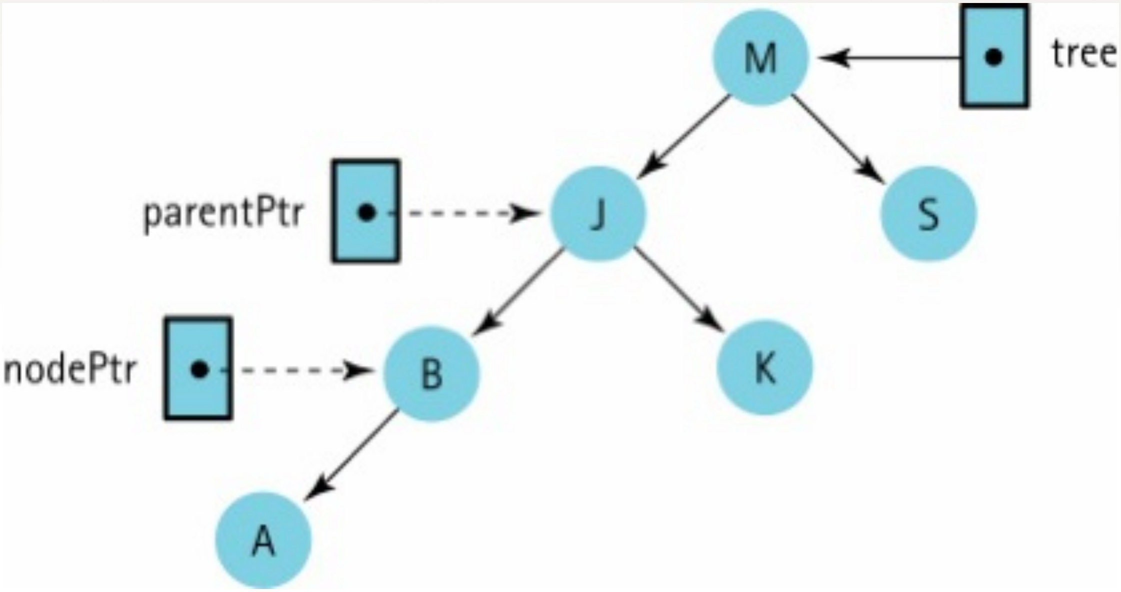
Code for PutItem

```
void TreeType::PutItem(ItemType item)
{
  TreeNode* newNode;
  TreeNode* nodePtr;
  TreeNode* parentPtr;
  newNode = new TreeNode;
  newNode->info = item;
  newNode->left = NULL;
  newNode->right = NULL;
  FindNode(root, item, nodePtr, parentPtr);
  if (parentPtr == NULL)
    root = newNode;
  else if (item < parentPtr->info)
    parentPtr->left = newNode;
  else parentPtr->right = newNode;
}
```

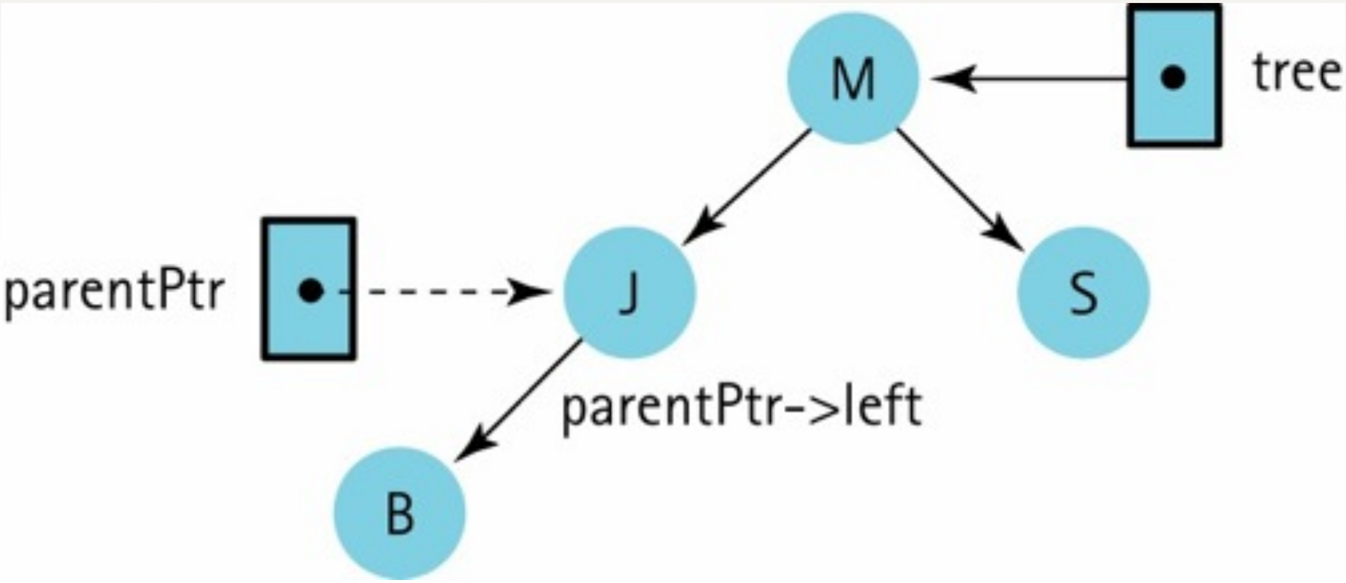
Code for `DeleteItem`

```
void TreeType::DeleteItem(ItemType item)
{
  TreeNode* nodePtr;
  TreeNode* parentPtr;
  FindNode(root, item, nodePtr, parentPtr);
  if (nodePtr == root)
    DeleteNode(root);
  else
    if (parentPtr->left == nodePtr)
      DeleteNode(parentPtr->left);
    else DeleteNode(parentPtr->right);
}
```

PointersnodePtr and parentPtr Are External to the Tree



Pointer parentPtr is External to the Tree, but parentPtr-> left is an Actual Pointer in the Tree

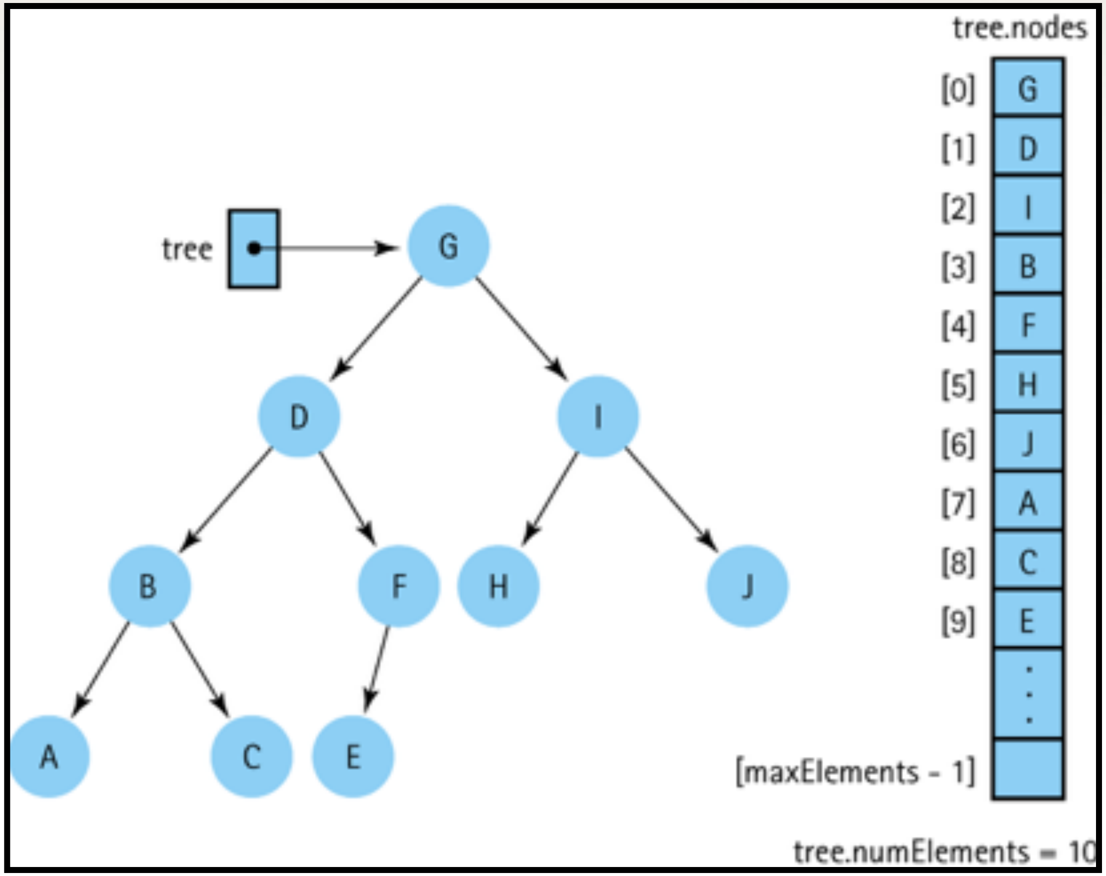


With Array Representation

For any node `tree.nodes[index]`

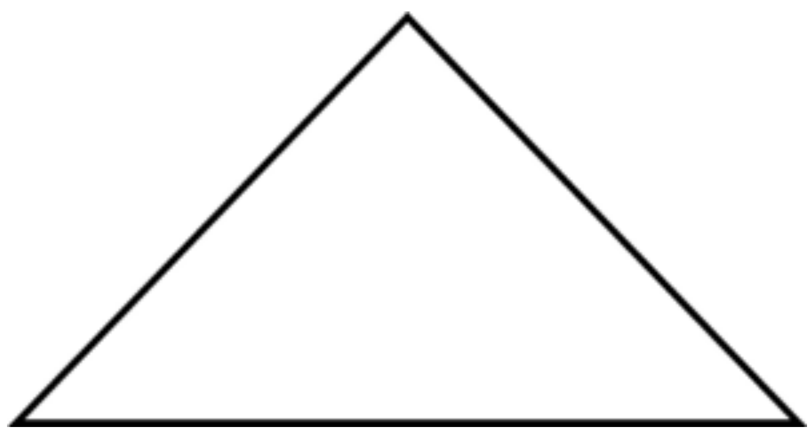
- its left child is in `tree.nodes[index*2 + 1]`
- right child is in `tree.nodes[index*2 + 2]`
- its parent is in `tree.nodes[(index – 1)/2]`.

A Binary Tree and Its Array Representation



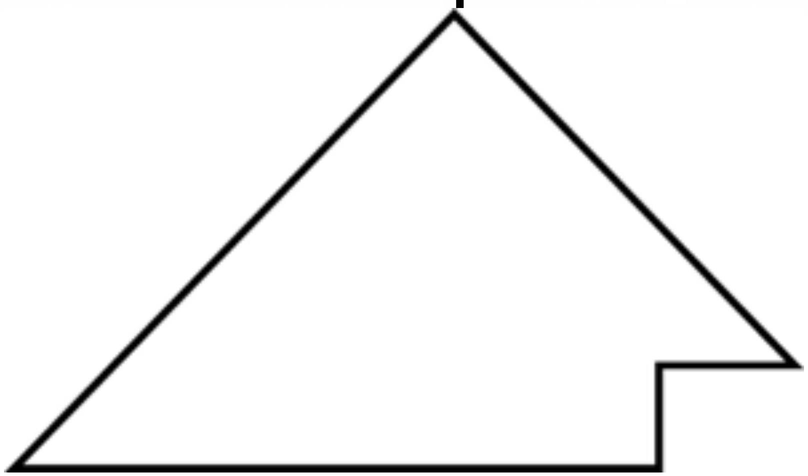
Definitions

Full Binary Tree: A binary tree in which all of the leaves are on the same level and every nonleaf node has two children

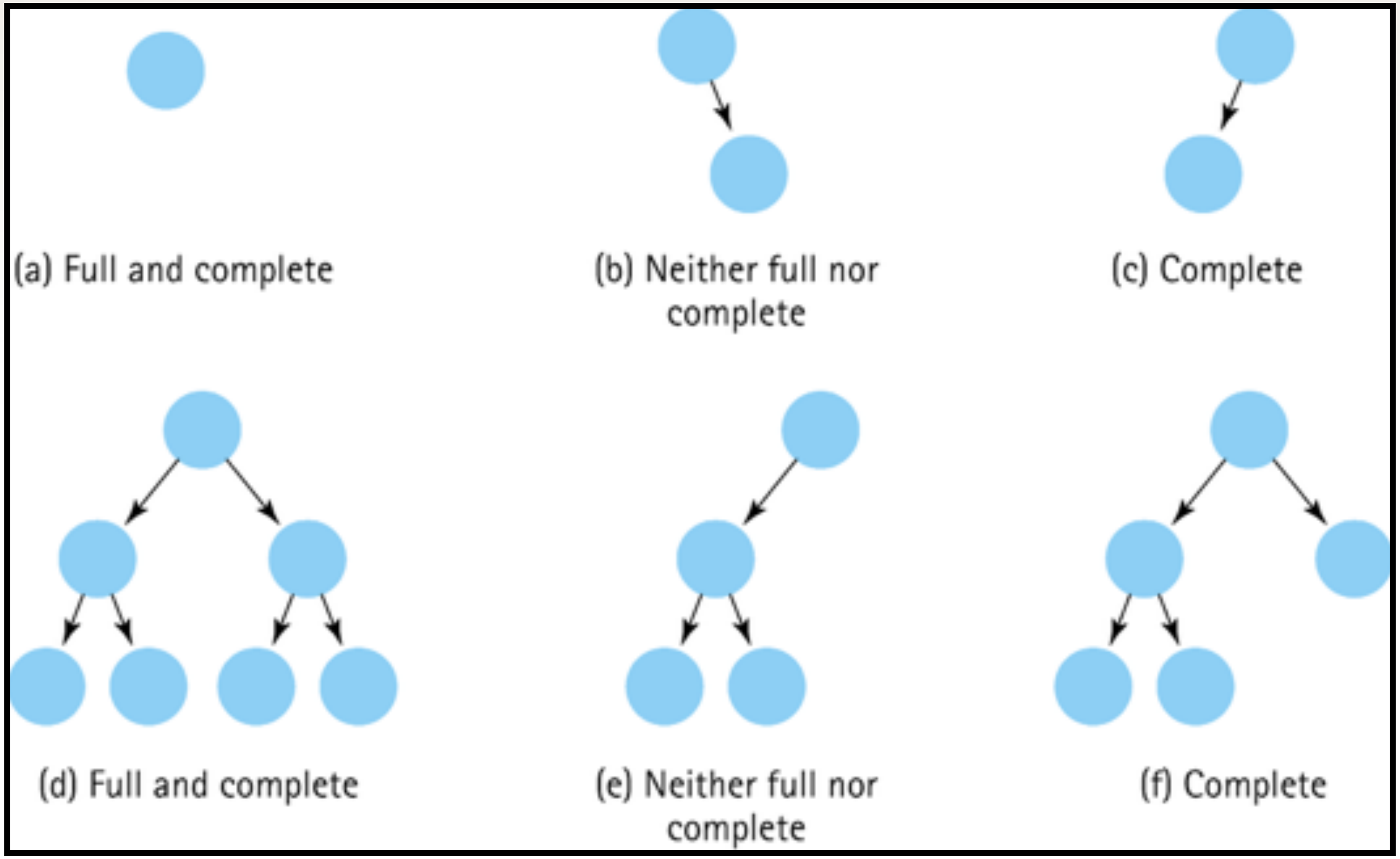


Definitions (cont.)

Complete Binary Tree: A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible



Examples of Different Types of Binary Trees



A Binary Search Tree Stored in an Array with Dummy Values