C++ Plus Data Structures

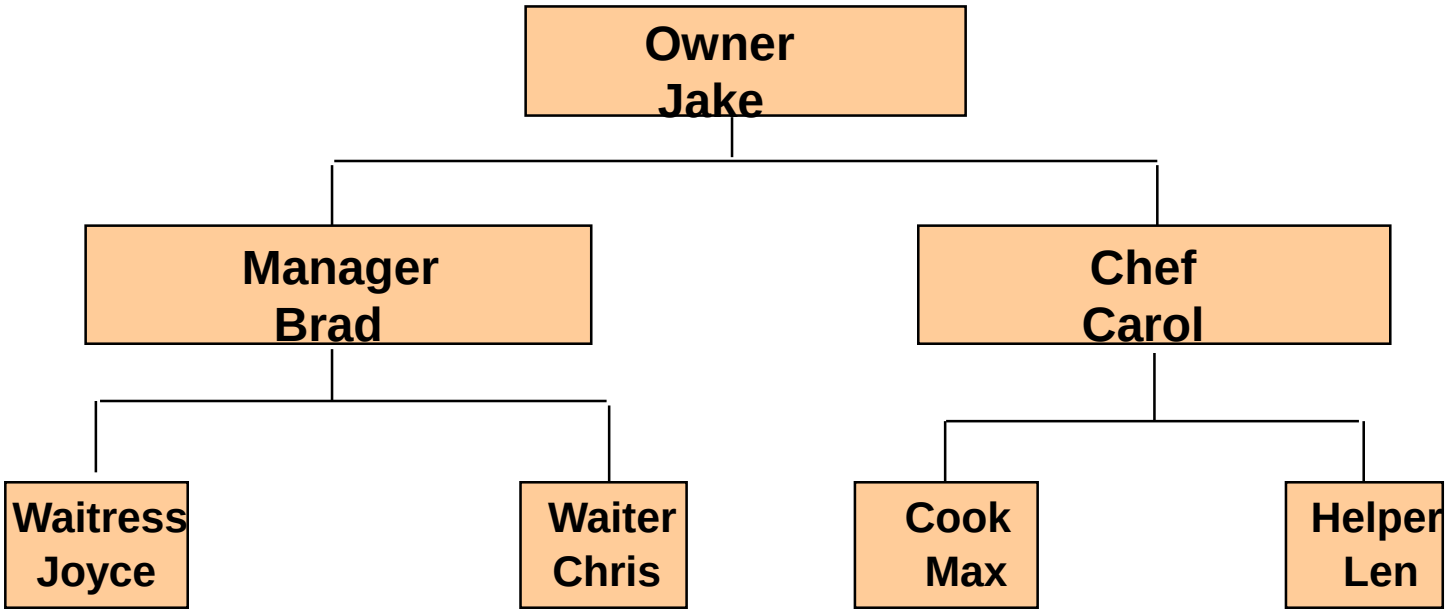
Nell Dale

David Teague

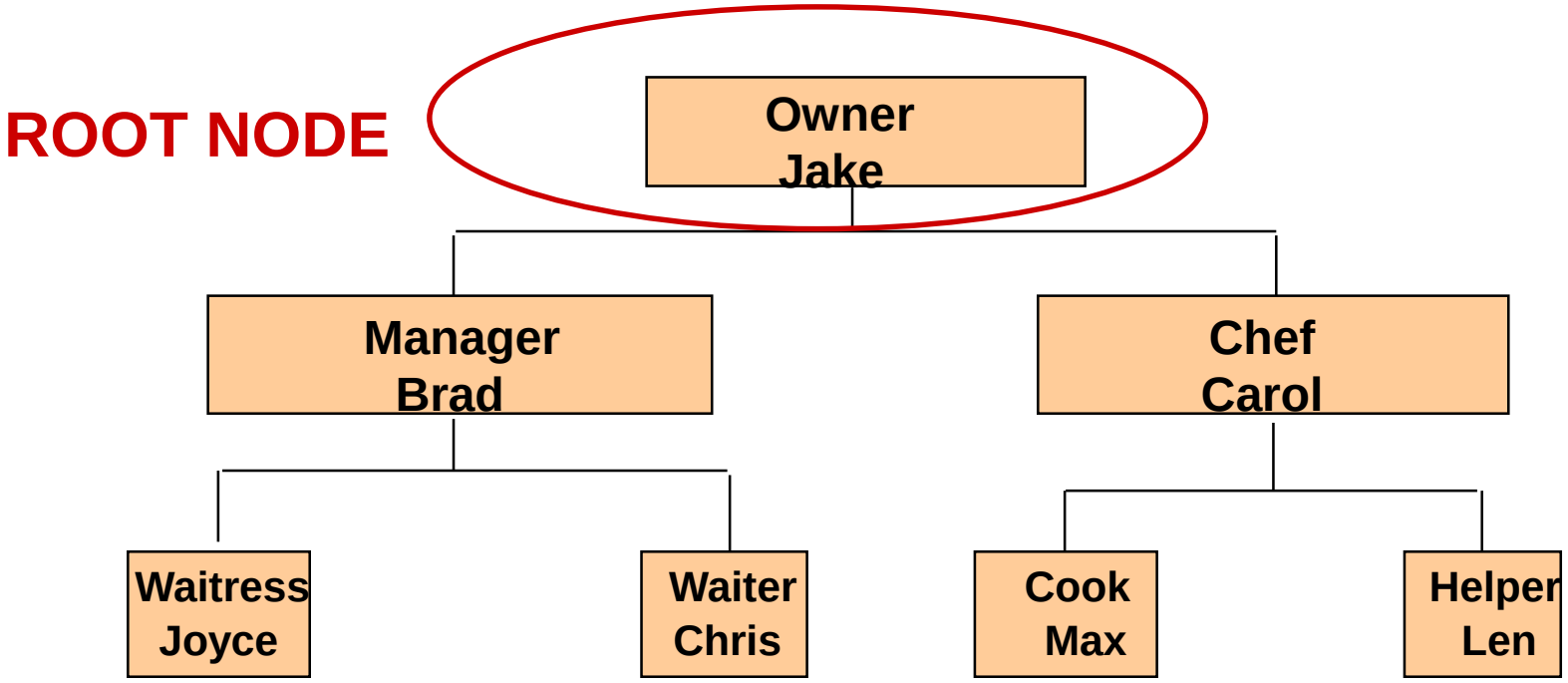
Chapter 8

Binary Search Trees

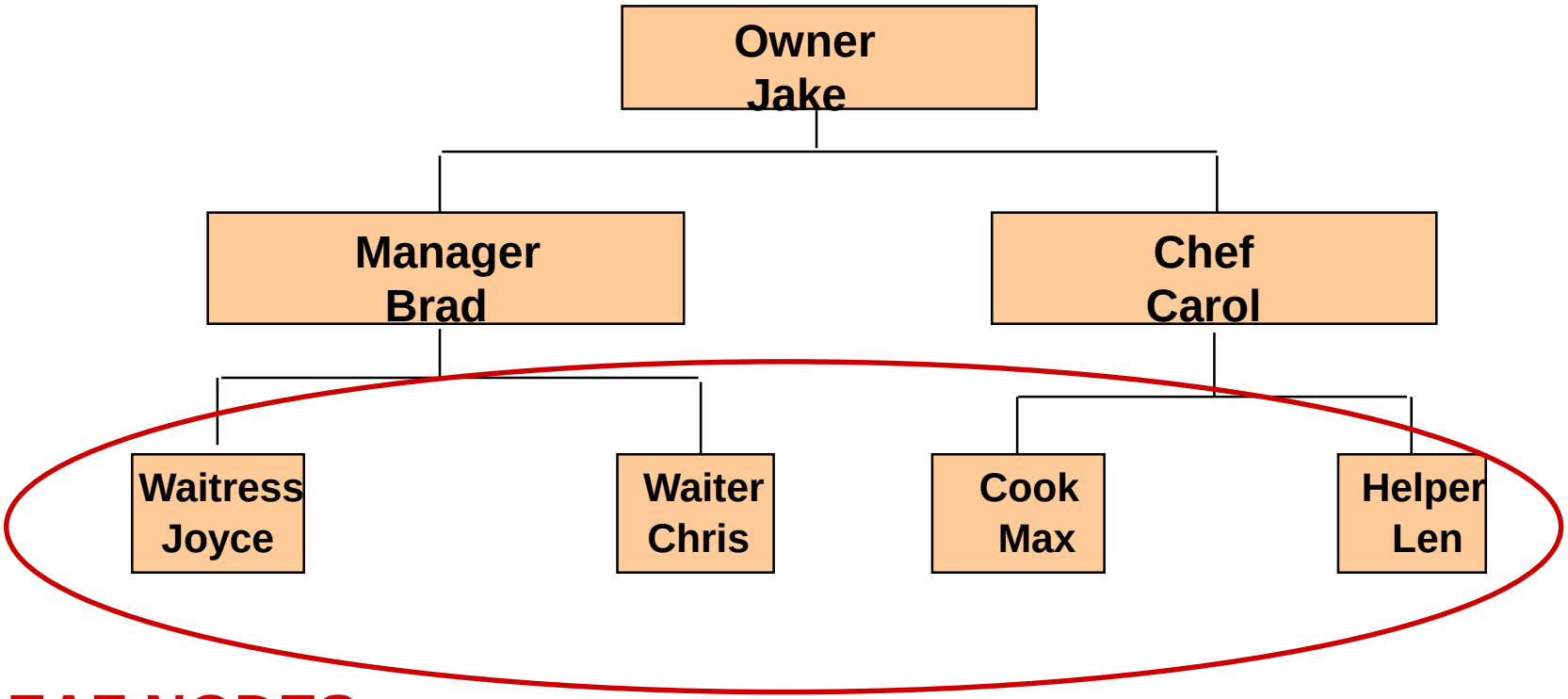
Jake's Pizza Shop



A Tree Has a Root Node

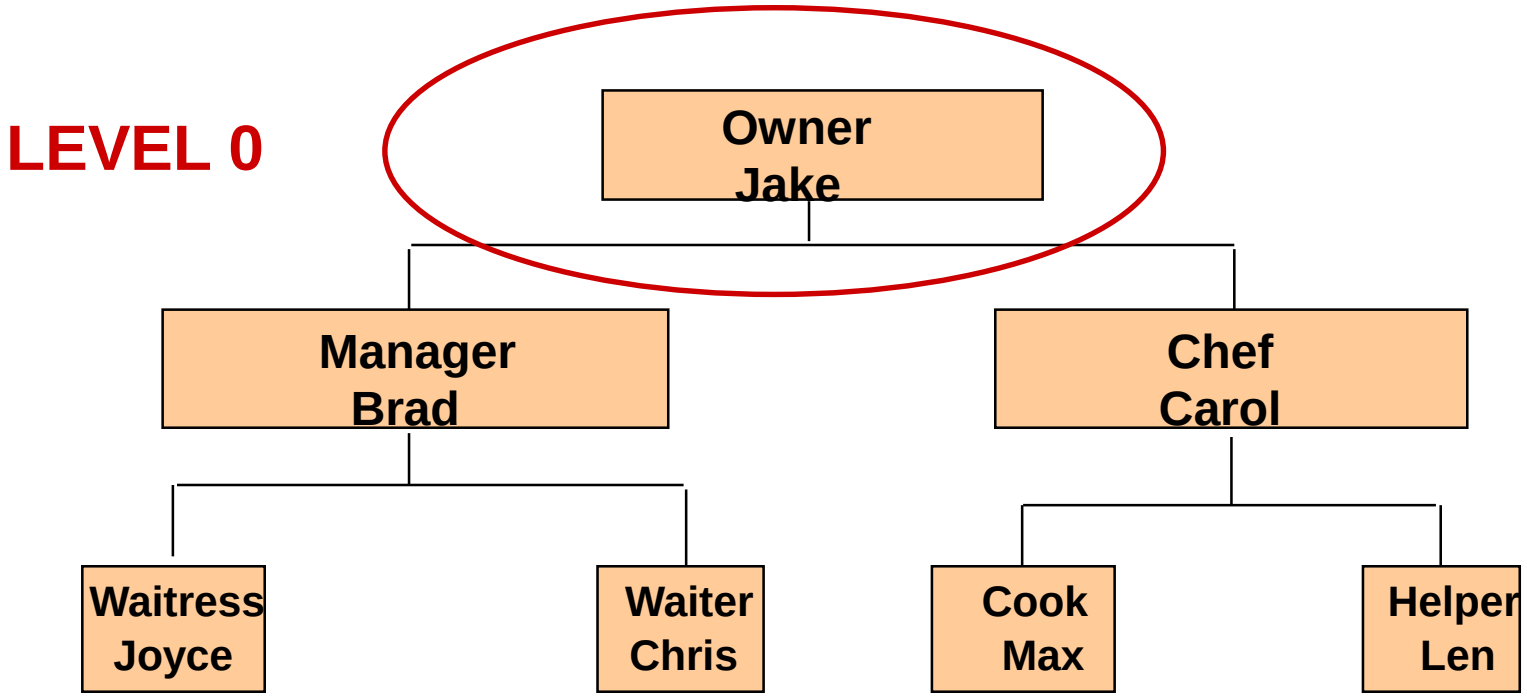


Leaf nodes have no children

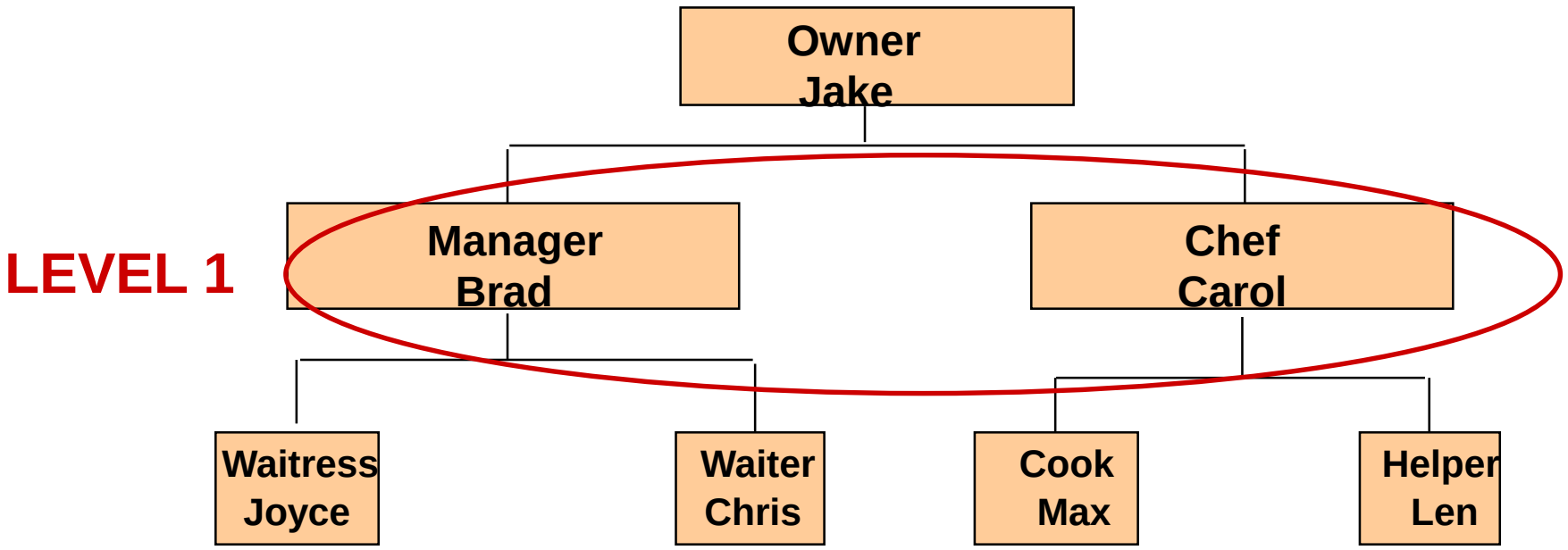


LEAF NODES

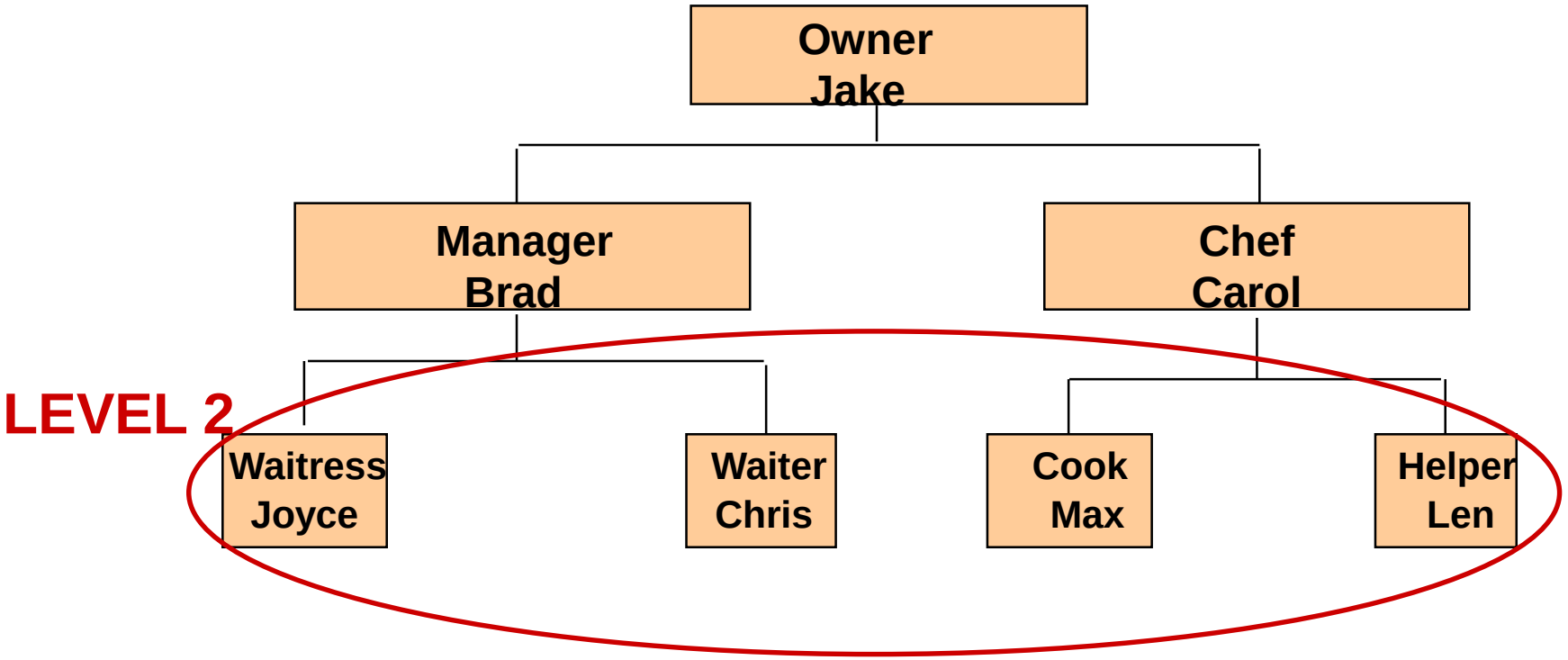
A Tree Has Levels



Level One

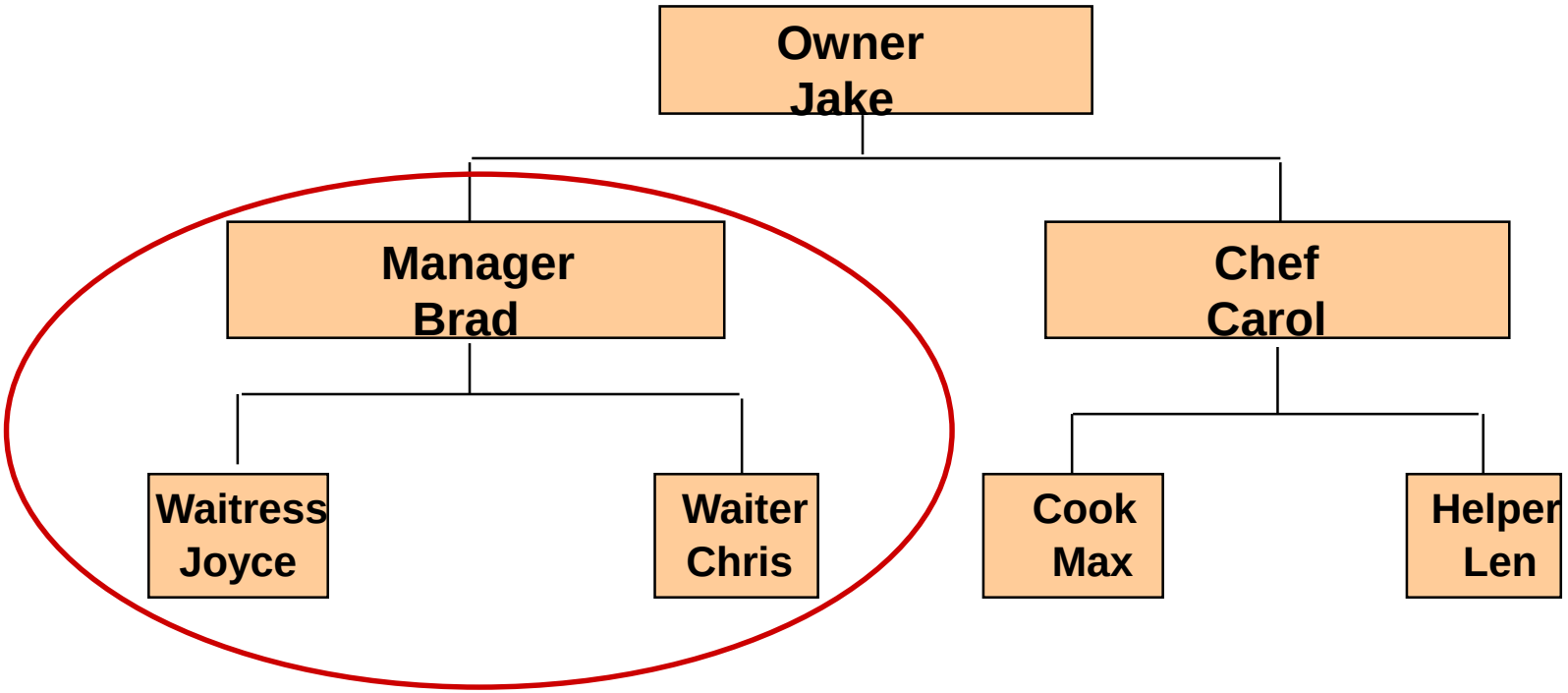

LEVEL 1
Owner
Jake
Manager
Brad
Chef
Carol
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

Level Two



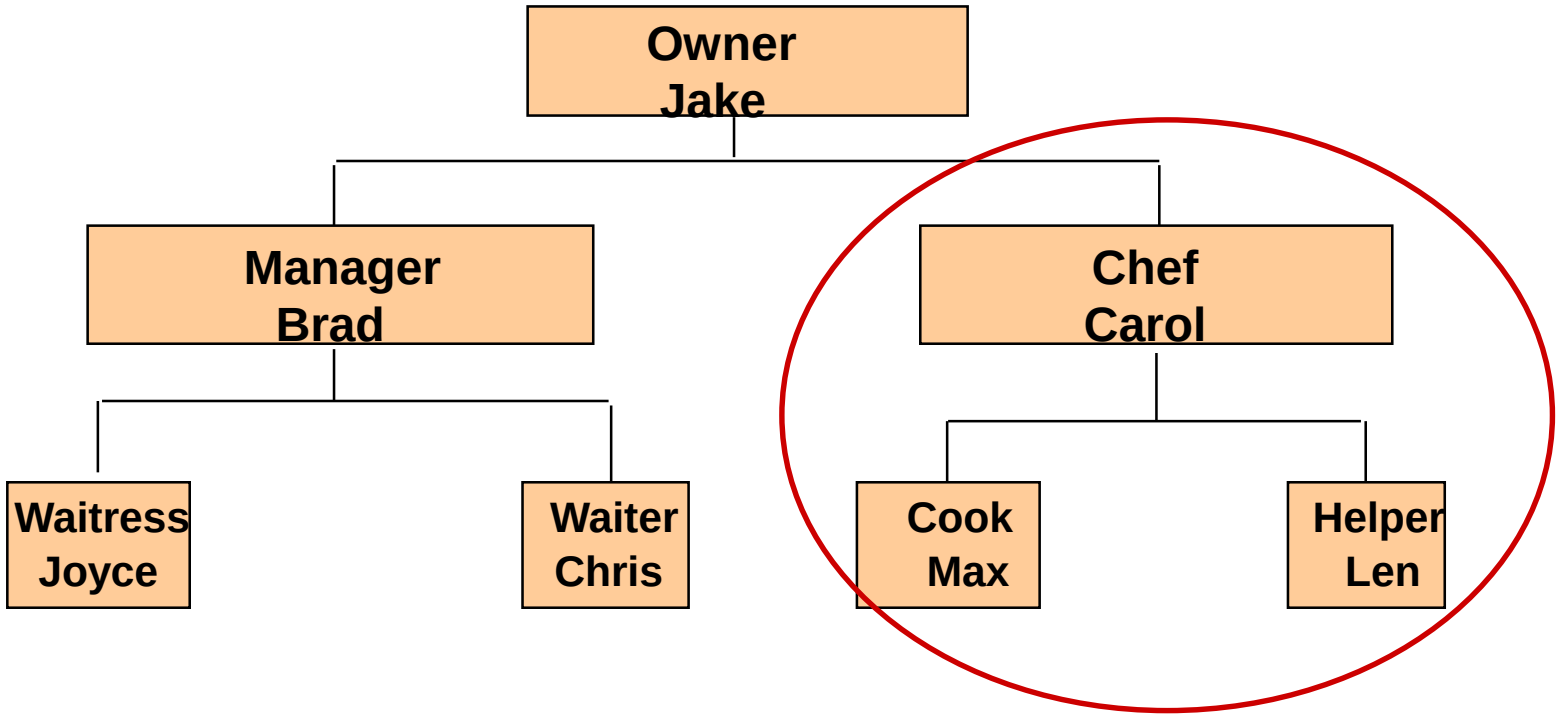
What is the Height of a tree?

A Subtree



LEFT SUBTREE OF ROOT NODE

Another Subtree



RIGHT SUBTREE OF ROOT NODE

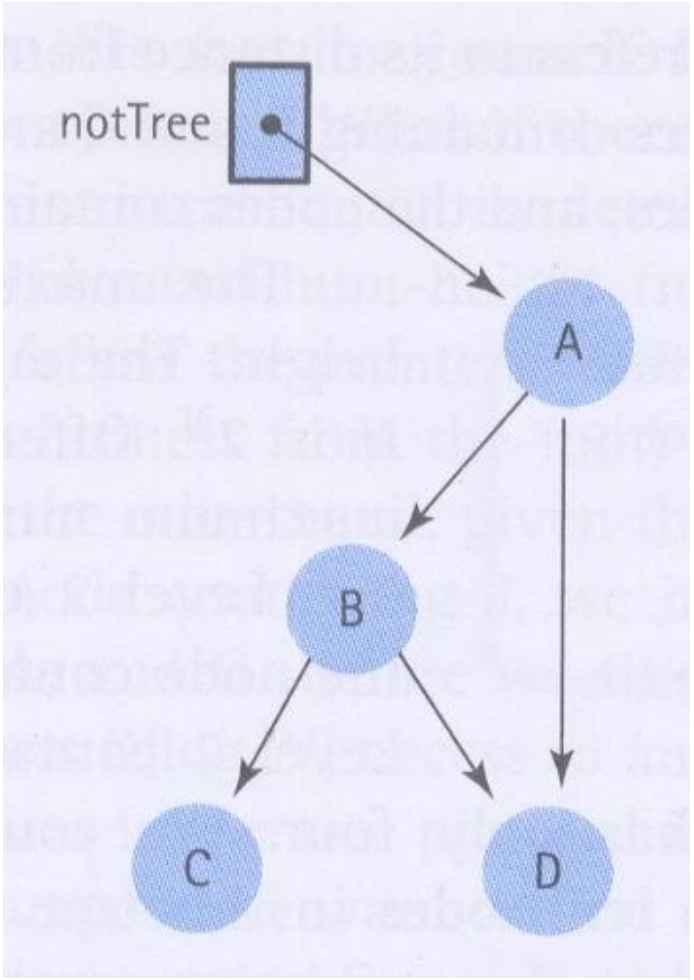
Binary Tree

A binary tree is a structure in which:

[1]Each node can have at most two children, and [2] in which a unique path exists from the root to every other node.

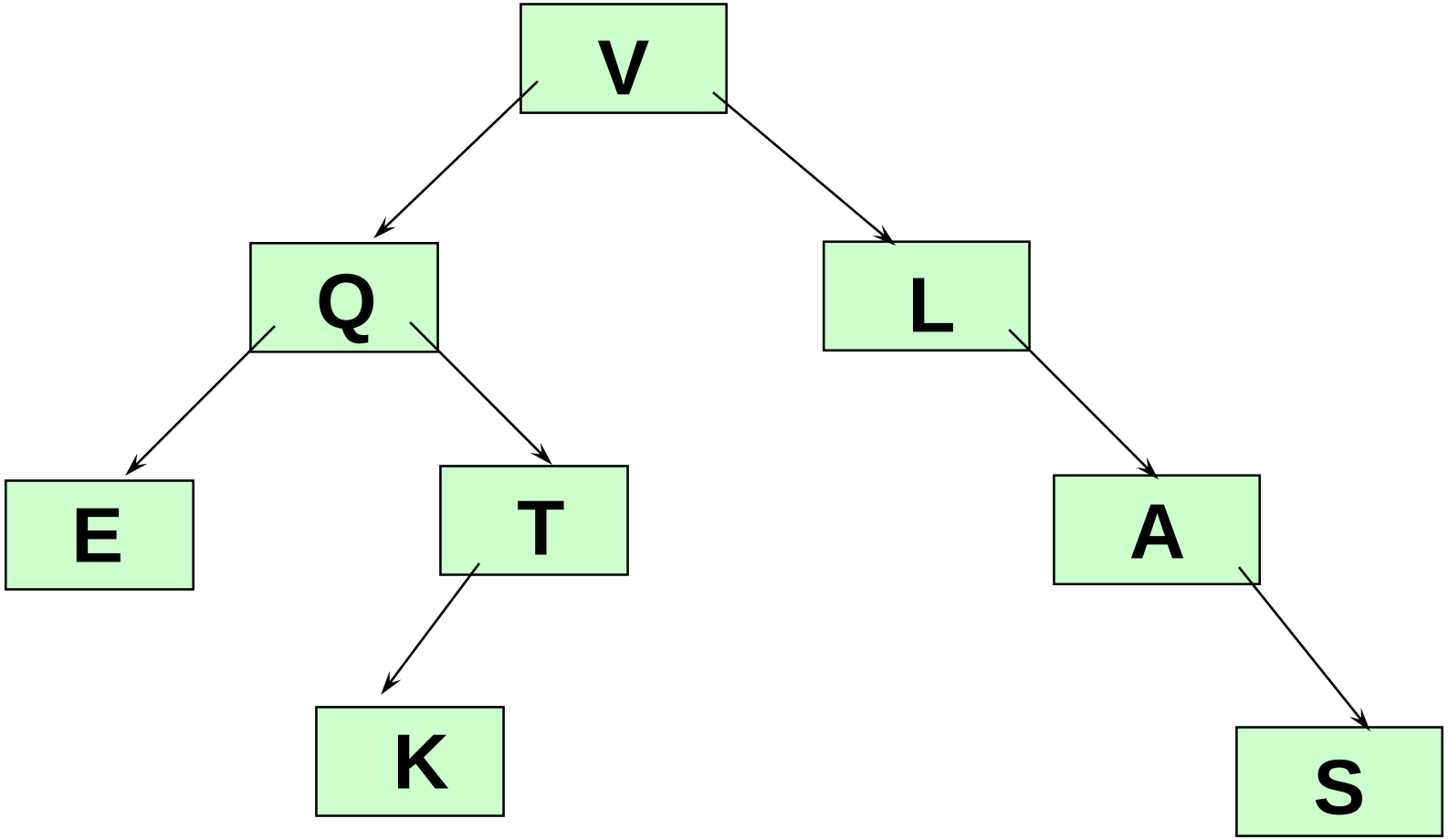
The two children of a node are called the left child and the right child, if they exist.

Binary Tree? If not, why?



A Binary Tree

- Height?



Level & Balance

- The max. number of nodes that a N level tree can have?
- Relation between the search time and the number of level.
- Max. number of nodes that each level can have.

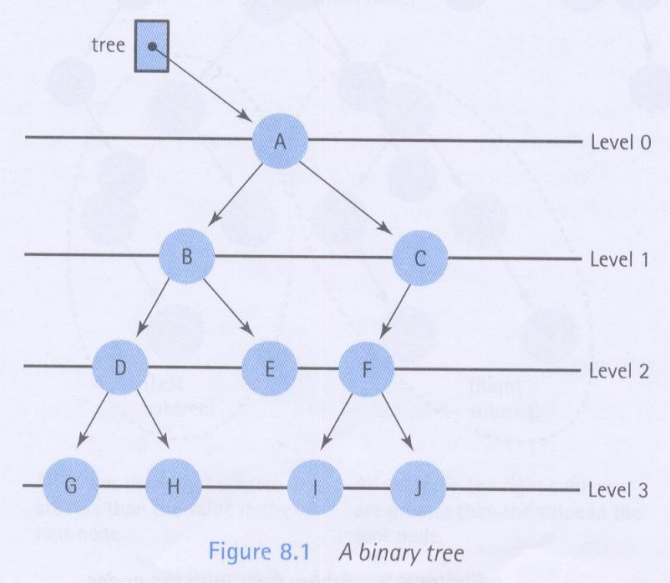


Figure 8.1 *A binary tree*

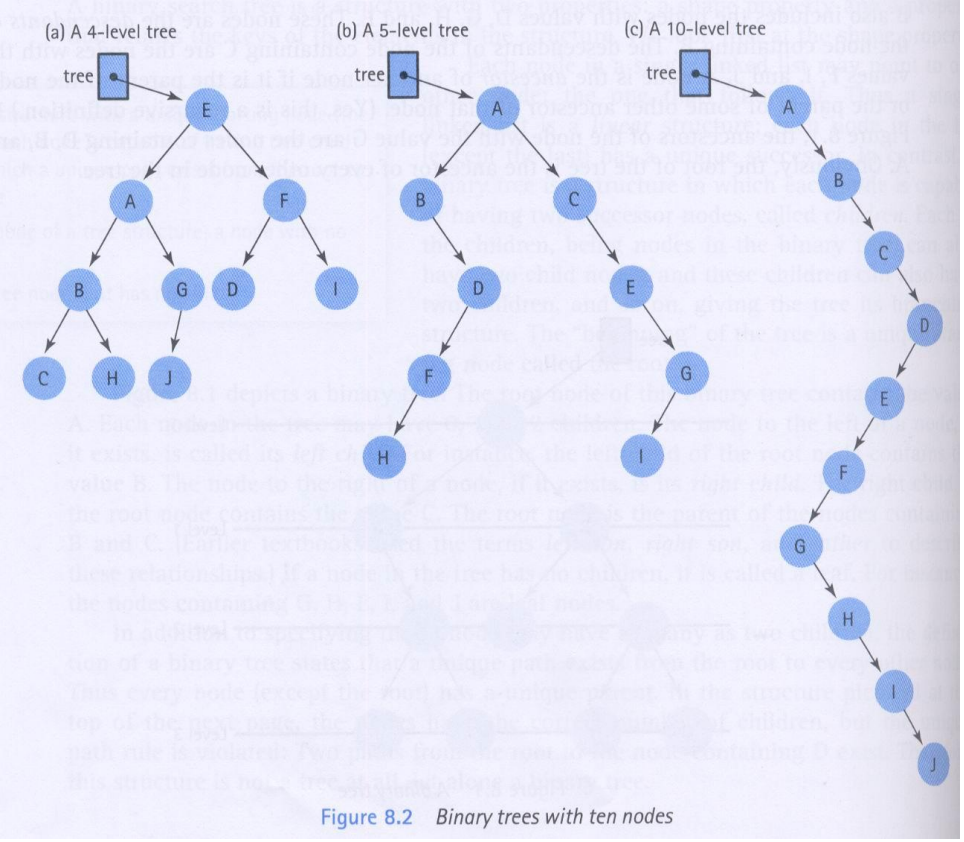
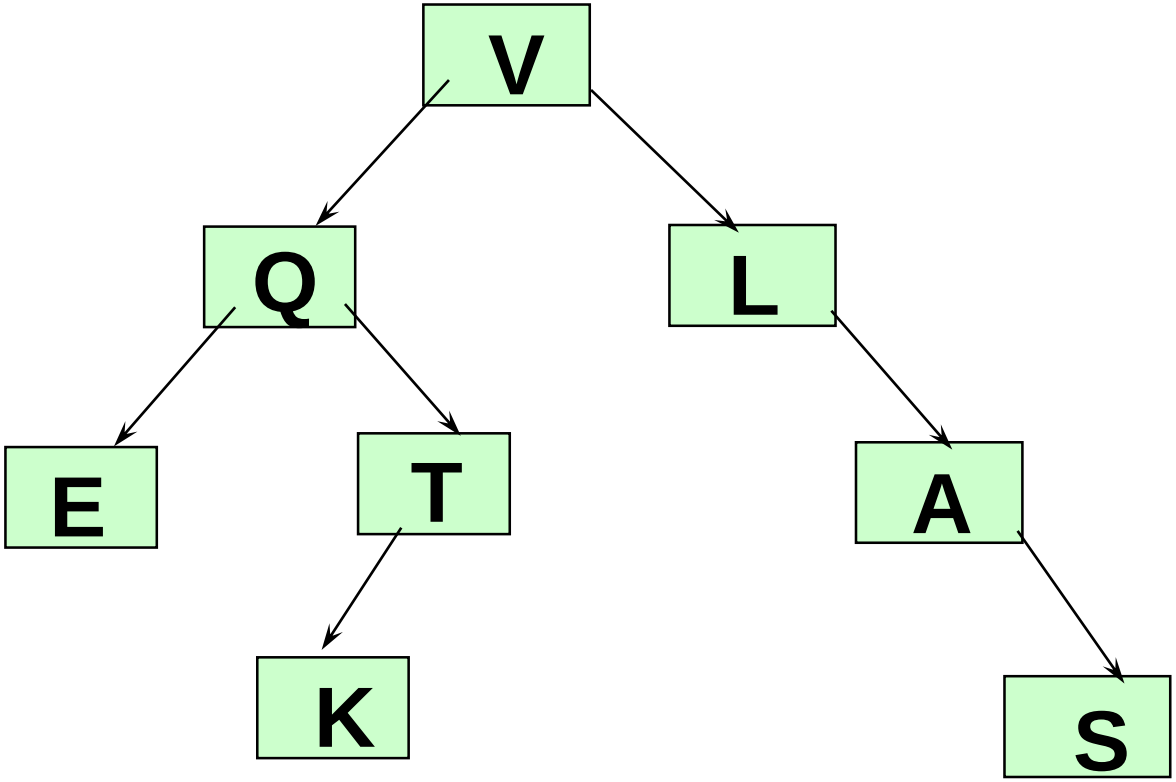
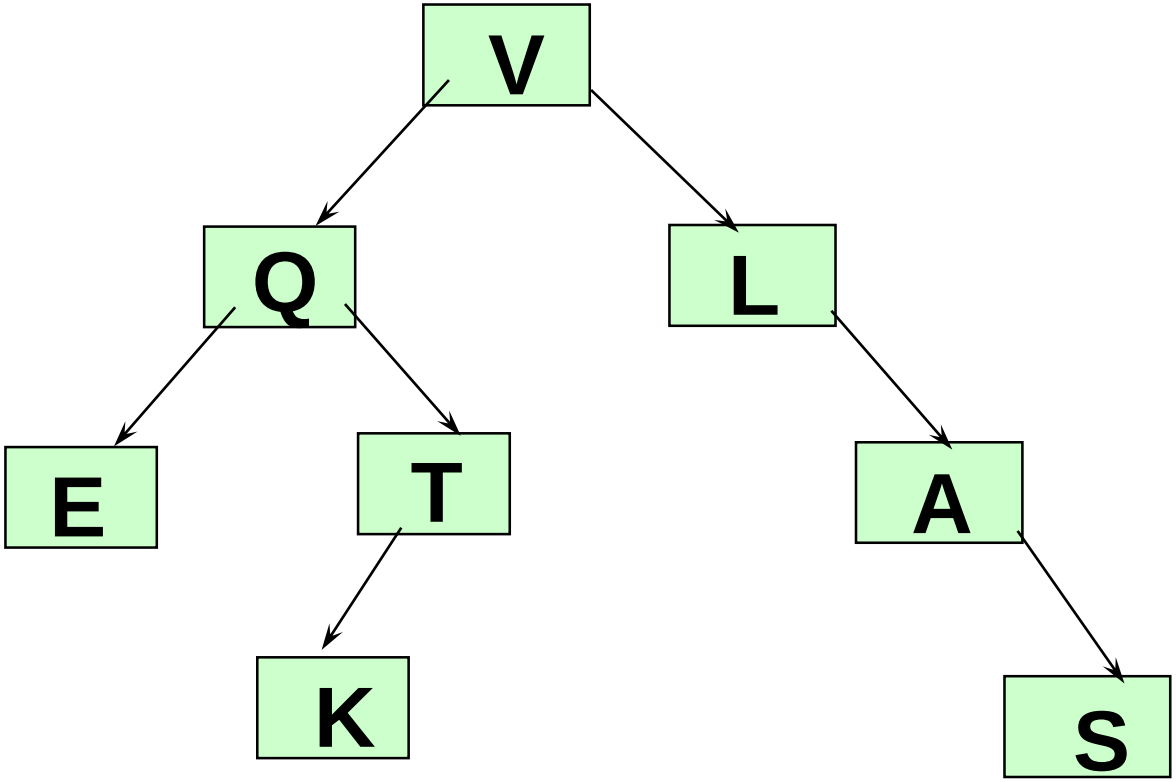


Figure 8.2 *Binary trees with ten nodes*

How many leaf nodes?

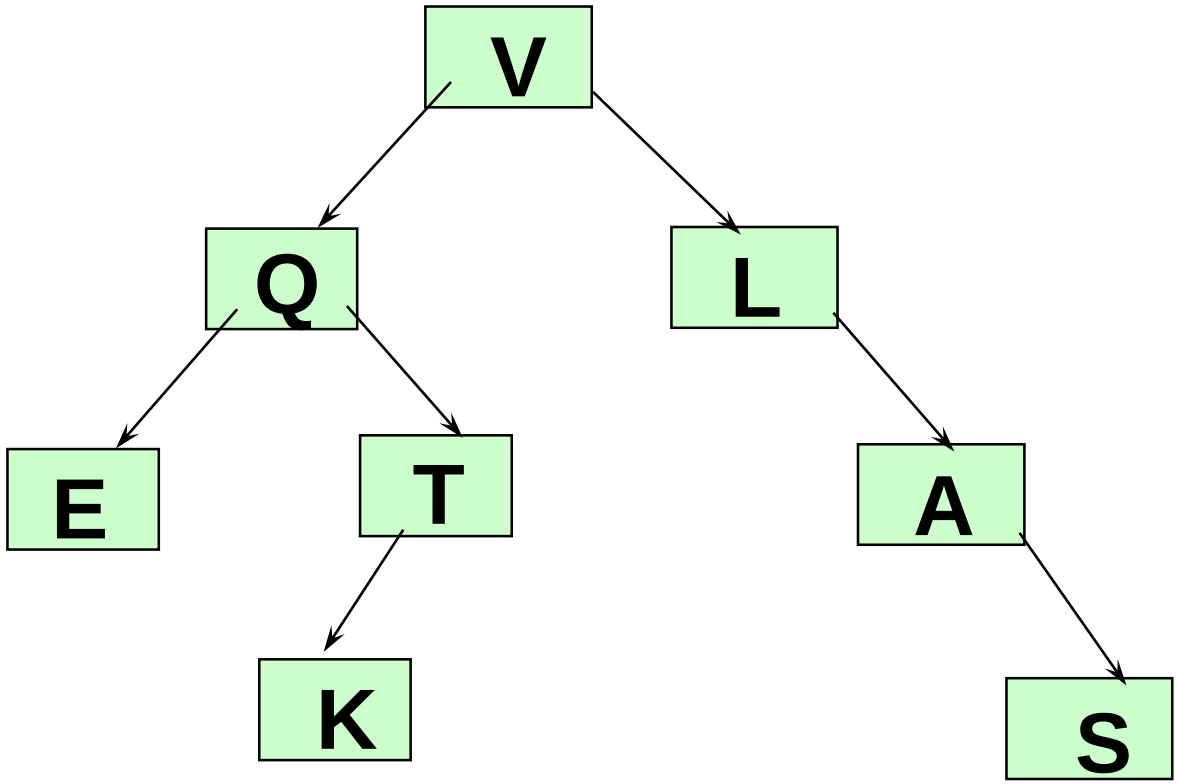


How many descendants of Q?



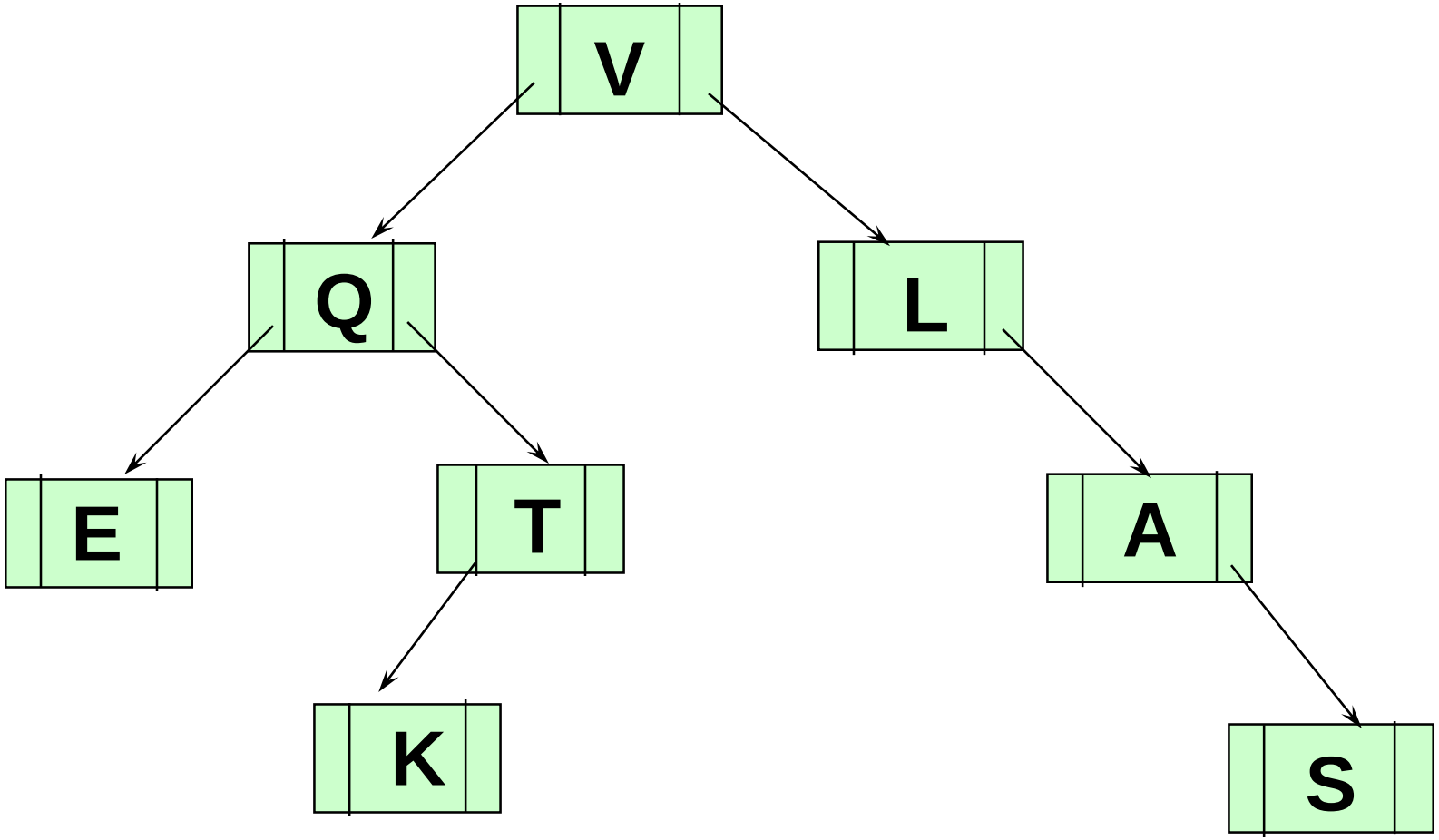
E, T, K

How many ancestors of K?



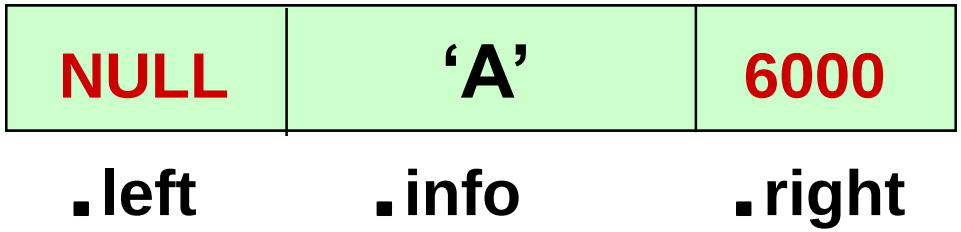
T, Q, V

Implementing a Binary Tree with Pointers and Dynamic Data



Each node contains two pointers

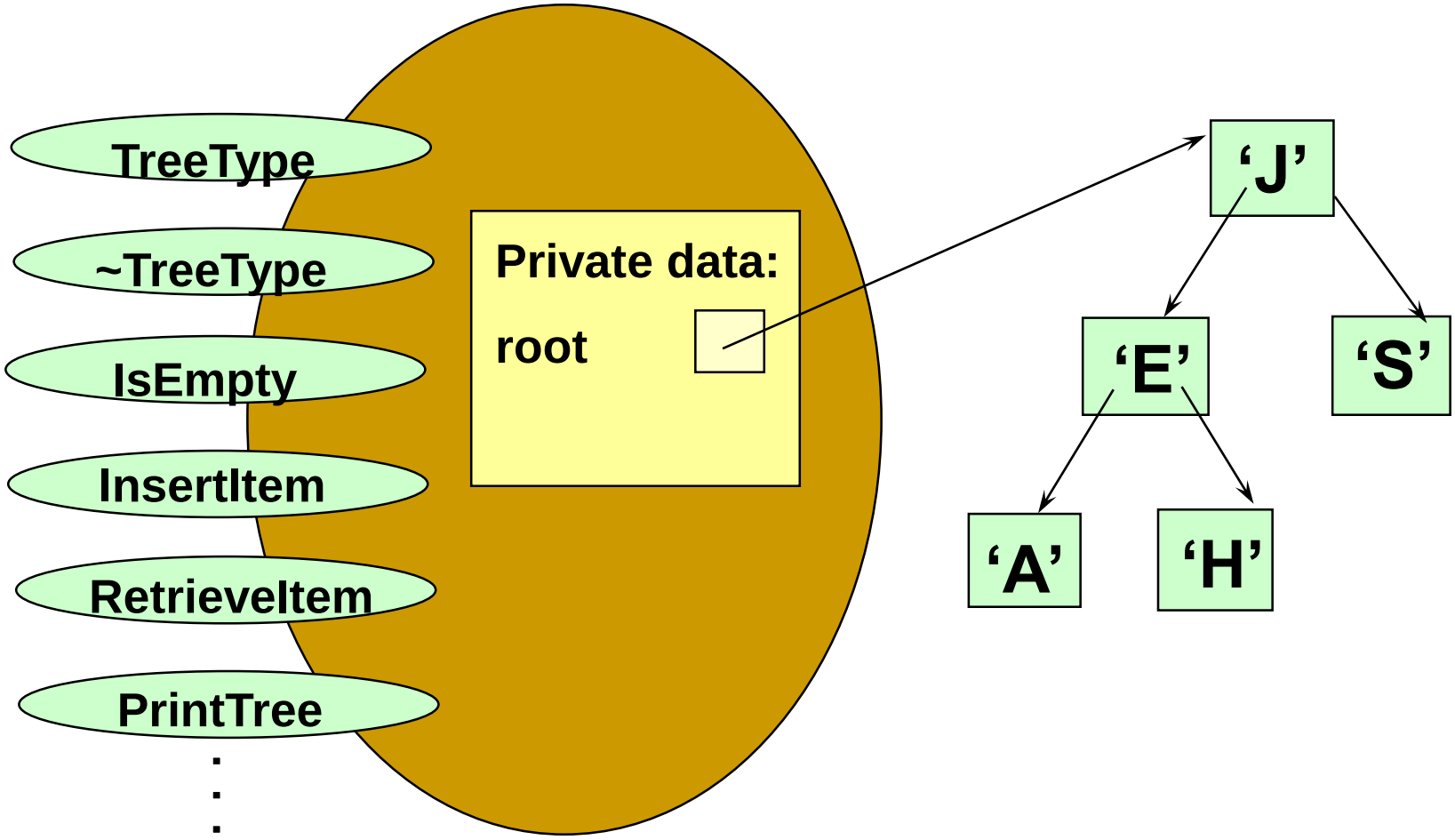
```
template< class  ItemType >
struct   TreeNode
 {
   ItemType     info;                    // Data member
   TreeNode<ItemType>*  left;            // Pointer to left child
   TreeNode<ItemType>*  right;           // Pointer to right child
};
```



```
class TreeType
{
public:
  TreeType();                          // constructor
 ~TreeType();                          // destructor
  TreeType(const  TreeType&  originalTree);  //  copy
   constructor
  void operator=(const TreeType& originalTree);
  void MakeEmpty();
  bool IsEmpty() const;
  bool IsFull() const;
  void ResetTree(OrderType order);
  int LengthIs() const;
  void RetrieveItem(ItemType& item, bool& found) const;
```

```
//   TreeType  계속
   void InsertItem(ItemType item);
   void DeleteItem(ItemType item);
   void GetNextItem (ItemType& item, OrderType order,
        bool& finished);
   void Print(std::ofstream& outFile) const;
private:
   TreeNode* root;
};

struct TreeNode
{
  ItemType info;
  TreeNode* left;
  TreeNode* right;
};
```

TreeType<char> CharBST;



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.

Subtree of a binary tree

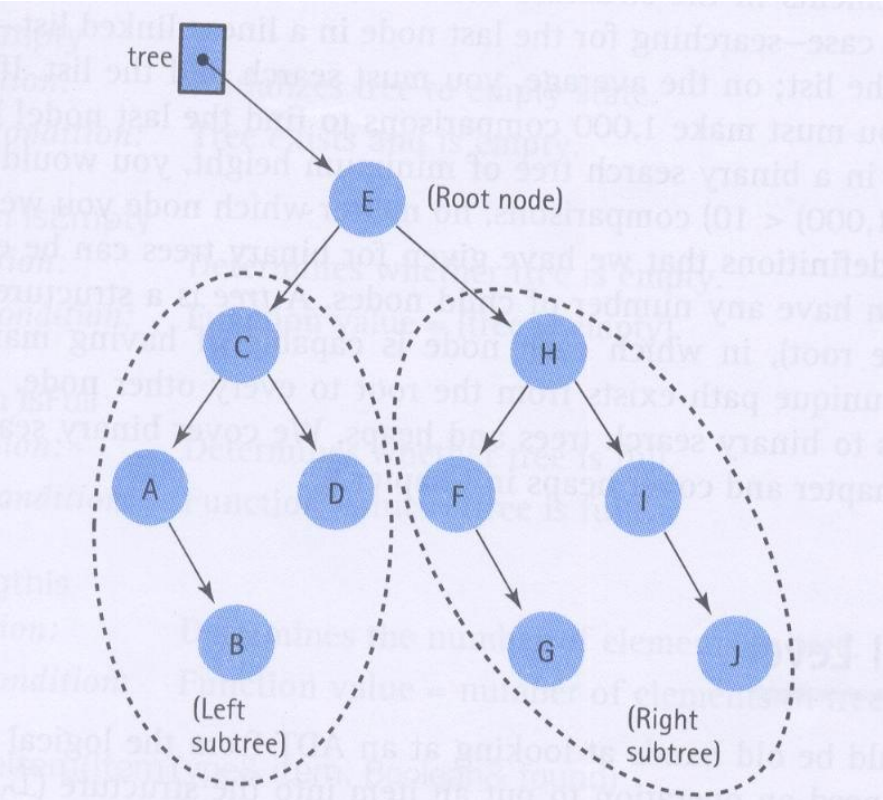


Figure 8.3 *A binary search tree*

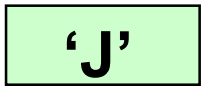
Left Tree < Right Tree

Shape of a binary search tree . . .

Depends on its key values and their order of insertion.

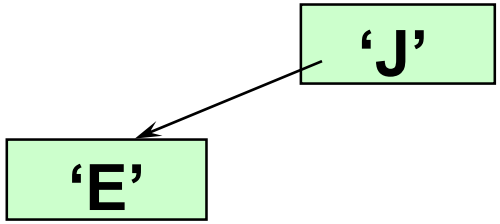
Insert the elements ‘J’ ‘E’ ‘F’ ‘T’ ‘A’ in that order.

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



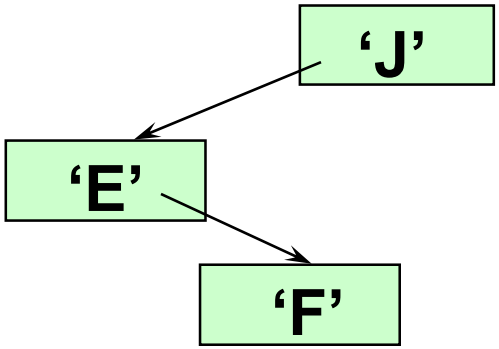
Inserting ‘E’ into the BST

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.



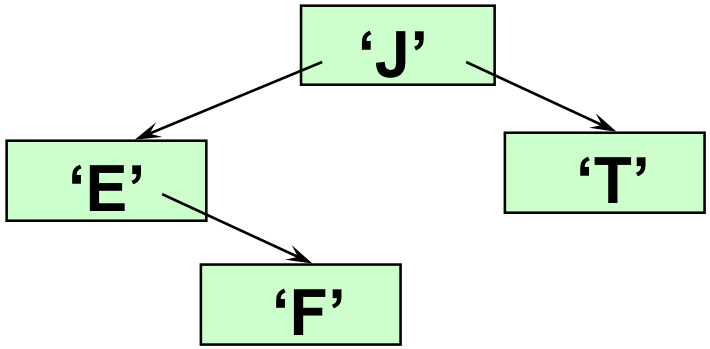
Inserting ‘F’ into the BST

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



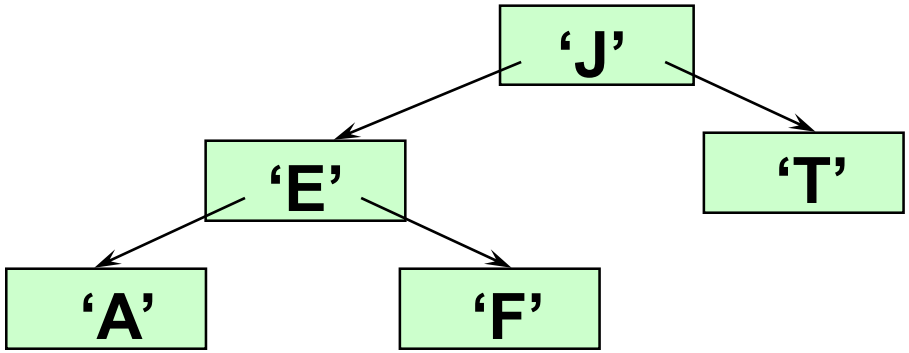
Inserting ‘T’ into the BST

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

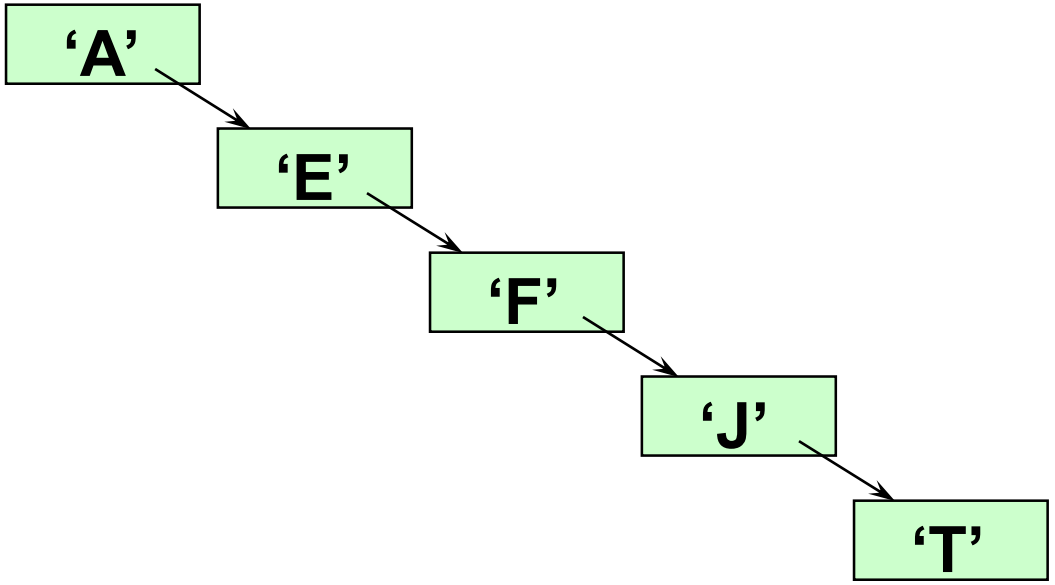
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order?

‘A’

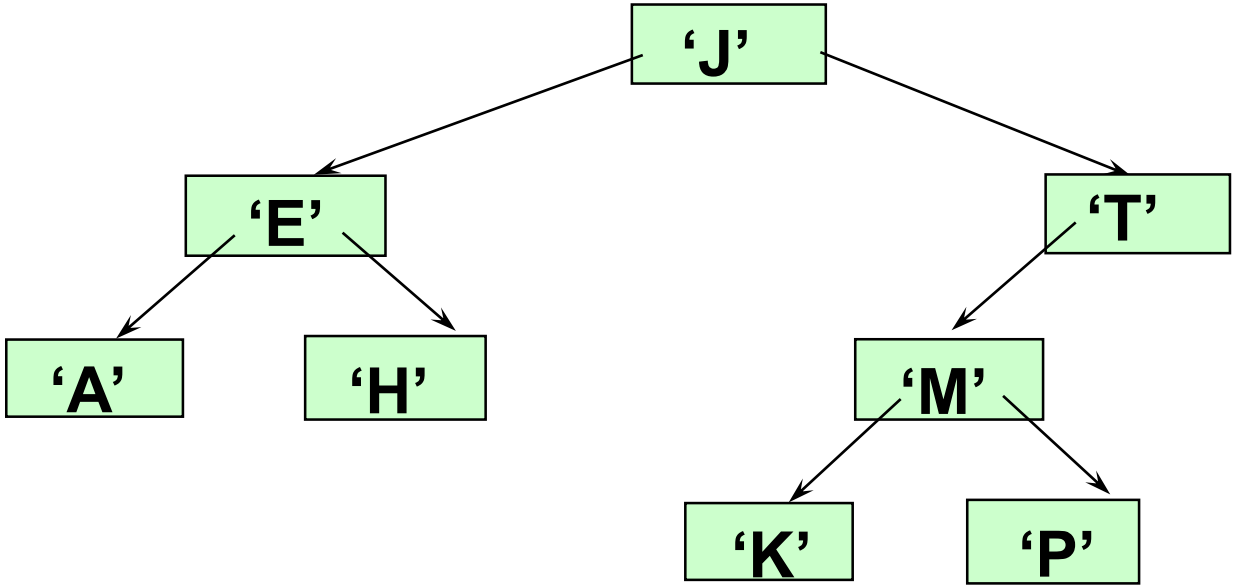
Binary search tree . . .

obtained by inserting

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



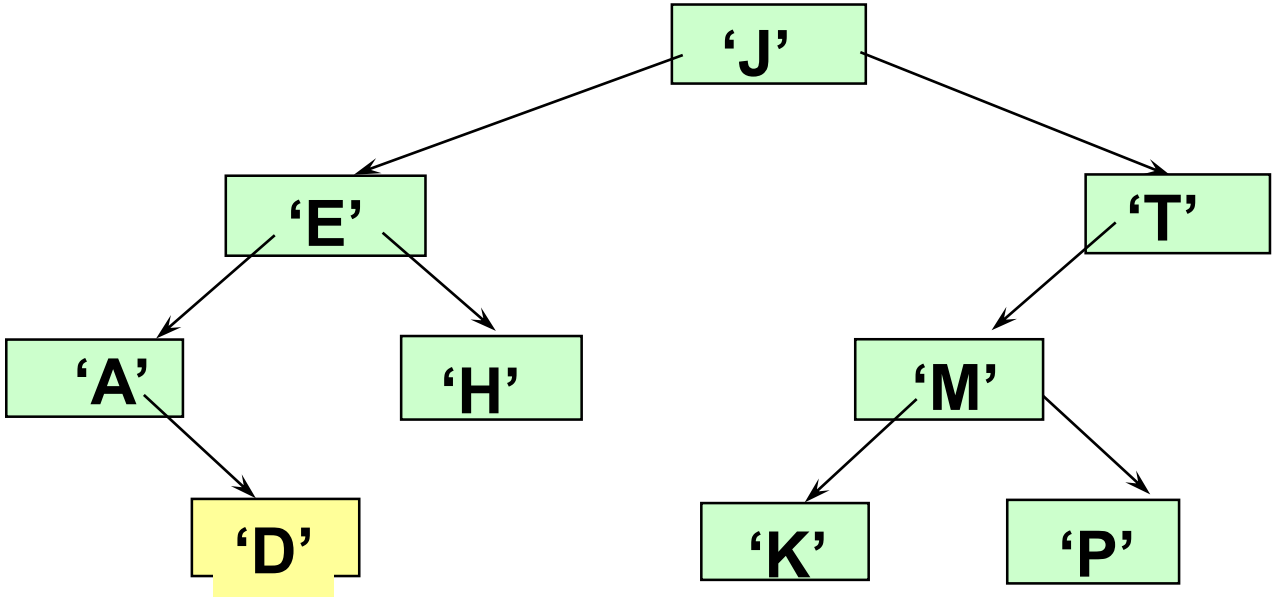
Another binary search tree



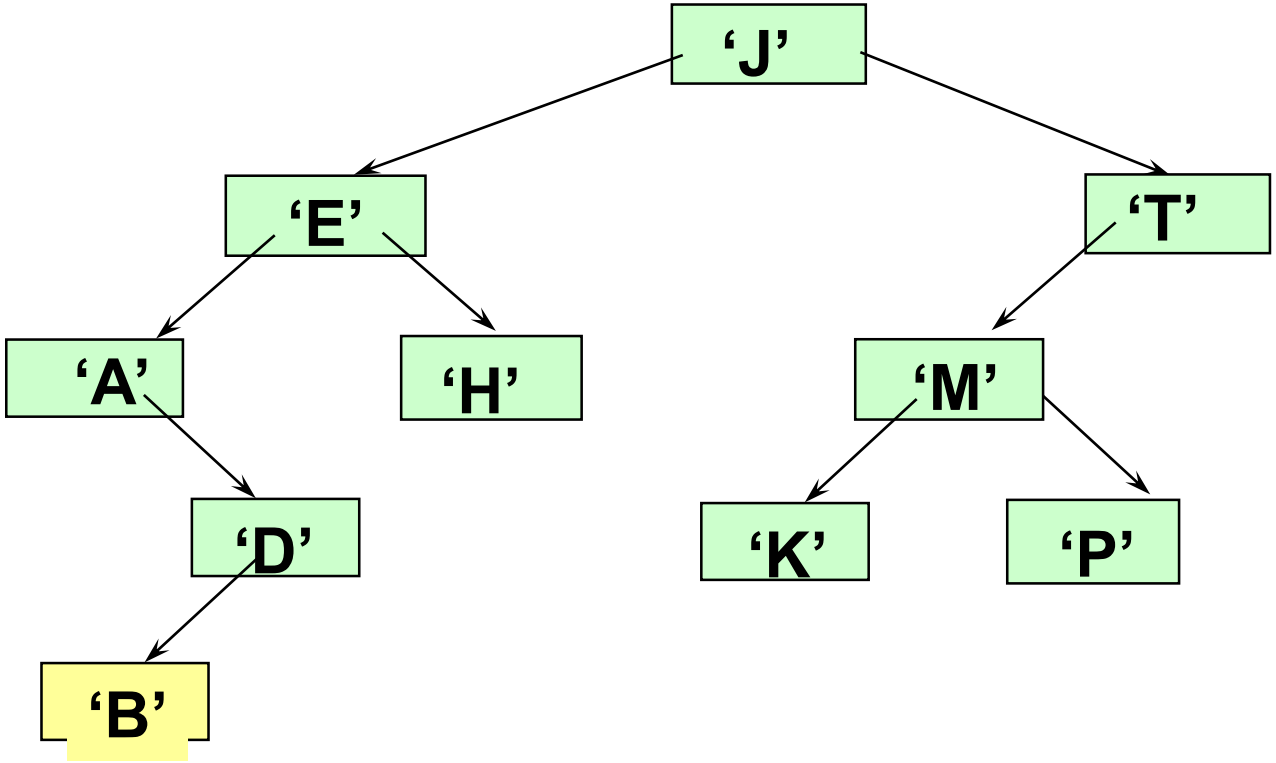
Add nodes containing these values in this order:

‘D’ ‘B’ ‘L’ ‘Q’ ‘S’ ‘V’ ‘Z’

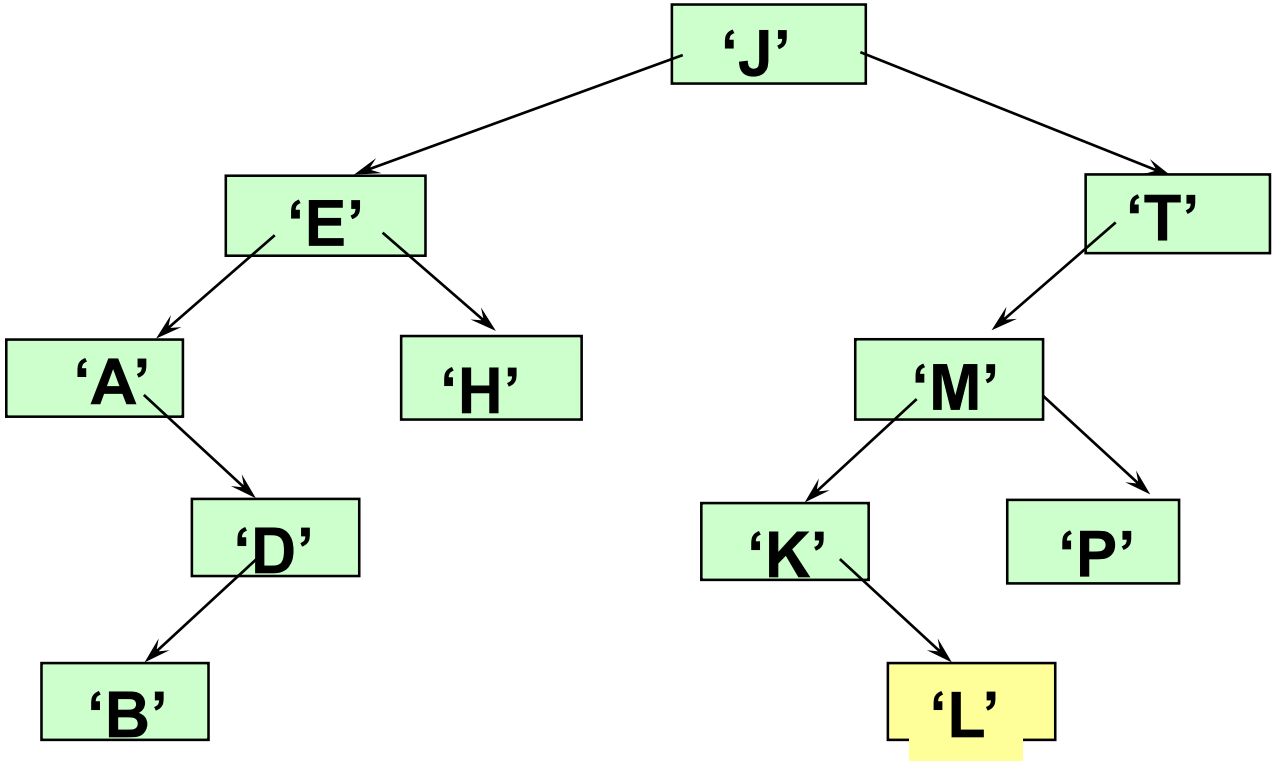
Insert ‘D’



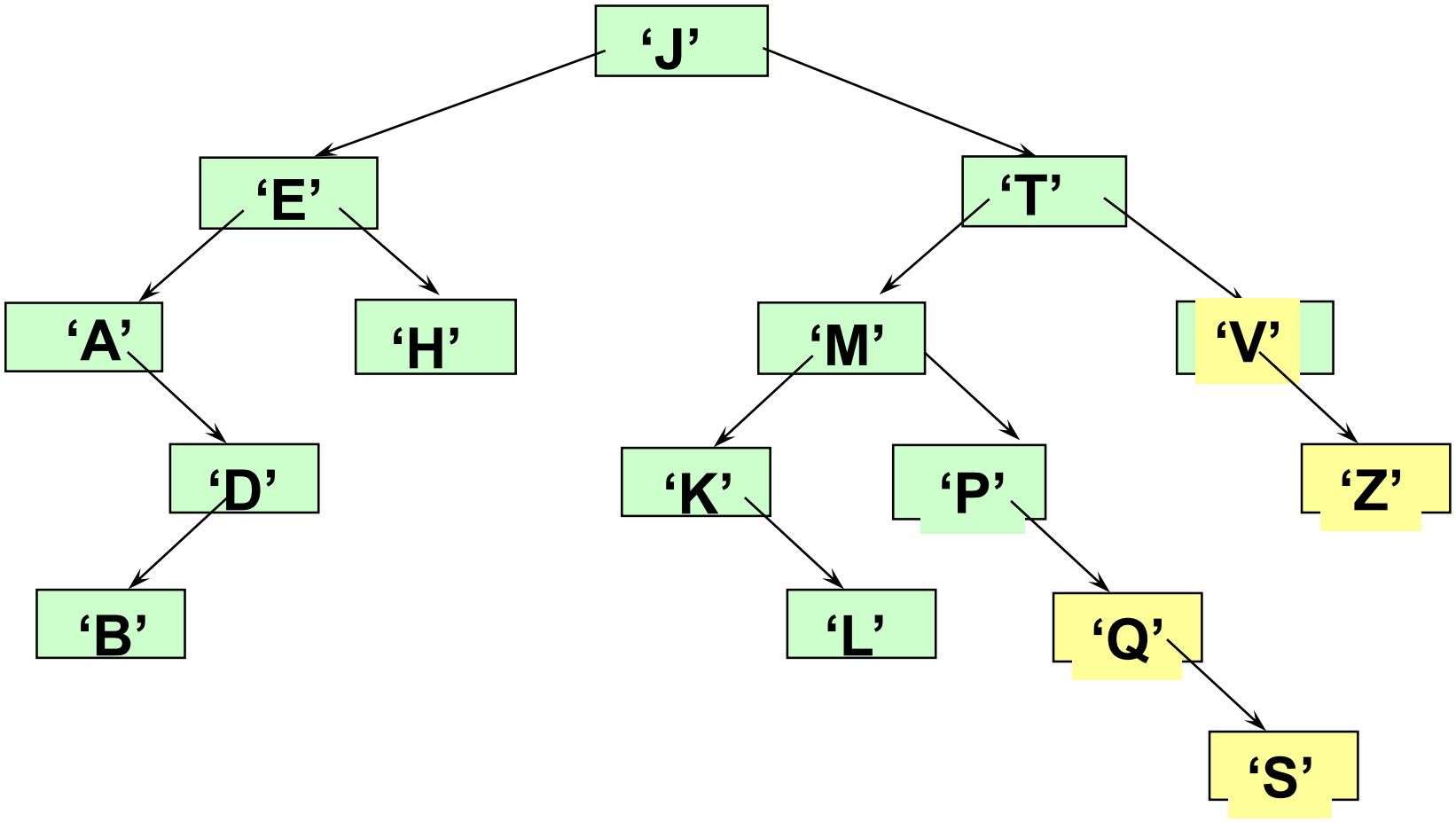
Insert ‘B’



Insert ‘L’



Insert ‘Q’ ‘S’ ‘V’ ‘Z’



```
Template<class ItemType>
class TreeType
{
public:
  TreeType();                          // constructor
 ~TreeType();                          // destructor
  TreeType(const TreeType<itemType>& originalTree);  // copy
    constructor
  void operator=(const TreeType<itemType>& originalTree);
  void MakeEmpty();
  bool IsEmpty() const;
  bool IsFull() const;
  void ResetTree(OrderType order);
  int LengthIs() const;
  void RetrieveItem(ItemType& item, bool& found) const;
  void InsertItem(ItemType item);
  void DeleteItem(ItemType item);
```

```
//   TreeType  계속
  void GetNextItem (ItemType& item, OrderType
            order, bool& finished);
  void Print(std::ofstream& outFile) const;
private:
    TreeNode<itemType>* root;
    QueType<itemType> preQue;
    QueType<itemType> inQue;
    QueType<itemType> postQue;
};

Template<class ItemType>
struct TreeNode
{
  ItemType info;
  TreeNode* left;
  TreeNode* right;
};
```

```
bool TreeType::IsFull() const
// Returns true if there is no room for another item
//  on the free store; false otherwise.
{
  TreeNode* location;
  try
  {
    location = new TreeNode;
    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
}

bool TreeType::IsEmpty() const
// Returns true if the tree is empty; false otherwise.
{
  return root == NULL;
}
```

The Function LengthIs

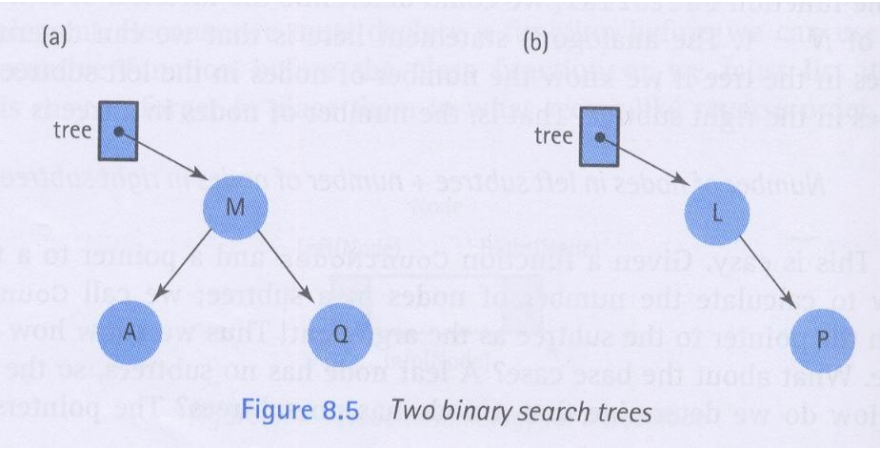


Figure 8.5 *Two binary search trees*

CountNodes Version 4

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

Definition: Count the Number of nodes in tree

Size: Number of nodes in tree

Base Case: If tree is NULL, return 0

General ,,: Return CountNodes(Left(tree))+CountNodes(Right(tree))+1

CountNodes Version 3

```
if tree is NULL
    return 0
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
```

Function LengthIs()

```
int CountNodes(TreeNode* tree);

int TreeType::LengthIs() const
// Calls recursive function CountNodes to count the
// nodes in the tree.
{
  return CountNodes(root);
}

int CountNodes(TreeNode* tree)
// Post: returns the number of nodes in the tree.
{
  if (tree == NULL)
    return 0;
  else
    return CountNodes(tree->left) + CountNodes(tree->right) + 1;
}
```

Retrieve Operation

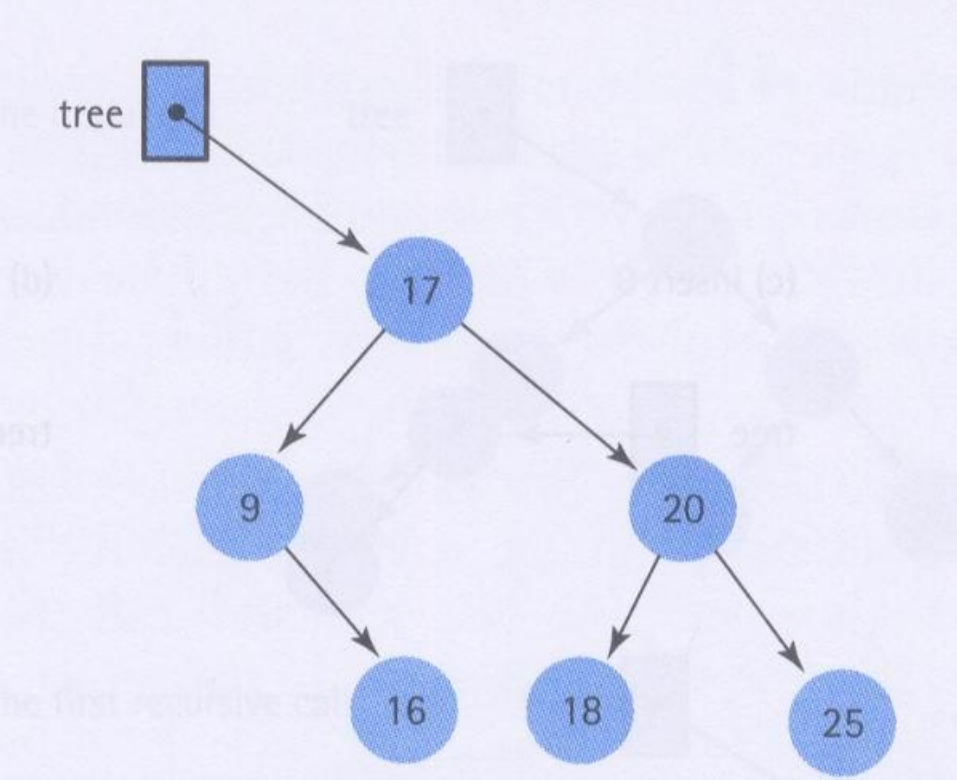


Figure 8.6 *Tracing the Retrieve operation*

-**Retrieve 18**

-**Retrieve 21**

Function Retrieve(tree, item,found)

Base Case:

-**If item's key matches key in Info(tree), item is set to Info(tree) and found is true**

-**If tree==NULL, found is false**

General Case:

If item's key is less than key in Info(tree), Retrieve(Left(tree), item, found);

Else Retrieve(Right(tree),item, found)

```
template< class  ItemType >
void   TreeType<ItemType> :: RetrieveItem ( ItemType&  item,
                                            bool&   found )
{
        Retrieve ( root, item,  found ) ;
}

template< class  ItemType >
void  Retrieve ( TreeNode<ItemType>* ptr, ItemType&  item,
                 bool&   found)
{     if  ( ptr == NULL )
          found = false ;
      else if  ( item  <  ptr->info )                    // GO LEFT
          Retrieve( ptr->left , item,  found ) ;
      else if  ( item  >  ptr->info )                    // GO RIGHT
          Retrieve( ptr->right , item,  found ) ;
      else
      {     item = ptr->info ;
          found = true ;
      }
}
```

Summary of Insertion Operation

- Recursive Operation
  - If current pointer==NUL, then insert.
  - Else If Key > current key, move to right subtree
  - Else If Key < current key, move to left subtree
  - Else (Key==current key), end with a error message.

The Function InsertItem

Insert 13

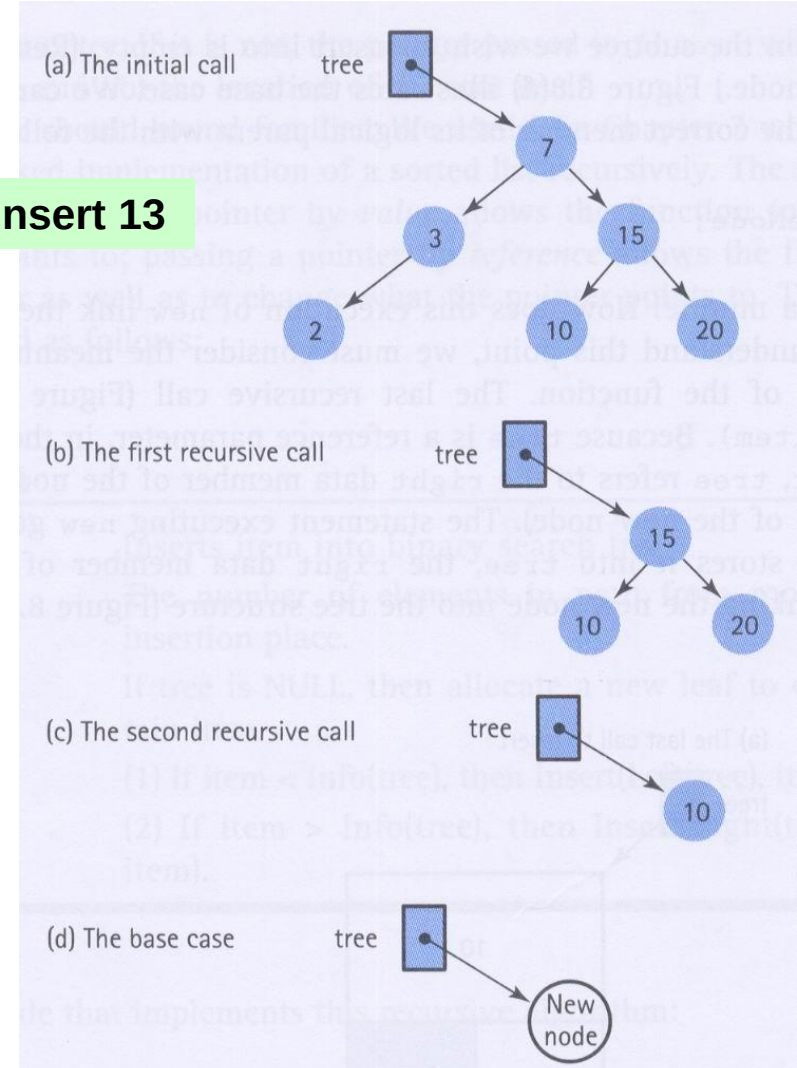


Figure 8.8 *The recursive Insert operation*

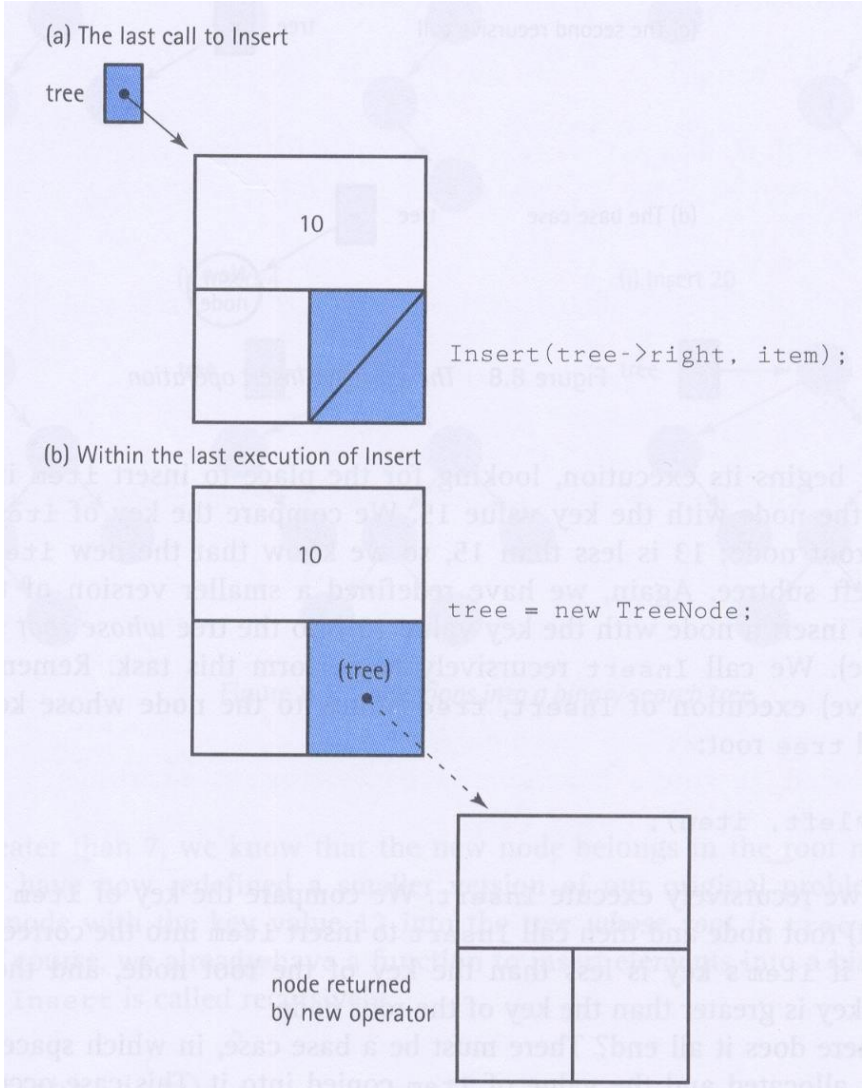
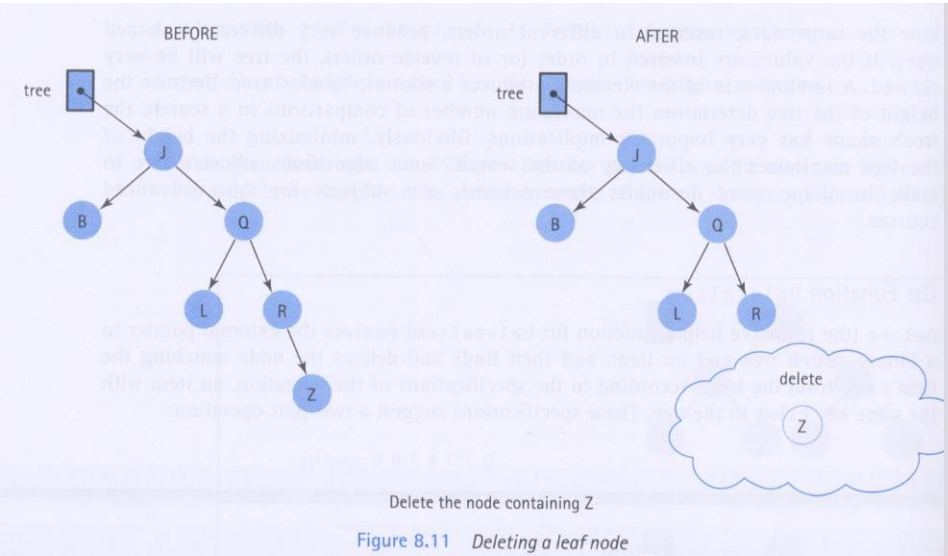


Figure 8.9 *The tree parameter is a pointer within the tree*

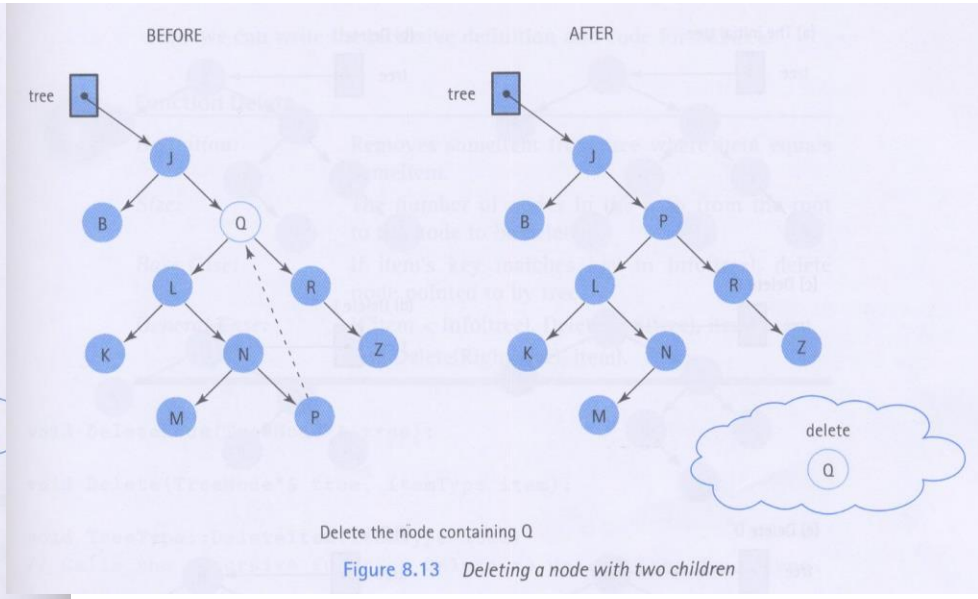
```
template< class  ItemType >
void   TreeType<ItemType> :: InsertItem ( ItemType  item )
{
        Insert ( root, item ) ;
}

template< class  ItemType >
void   Insert ( TreeNode<ItemType>*&  ptr,   ItemType  item )
{     if  ( ptr == NULL )
      {                                          // INSERT item HERE AS LEAF
          ptr = new TreeNode<ItemType> ;
          ptr->right = NULL ;
          ptr->left = NULL ;
          ptr->info = item ;
      }
      else if  ( item  <  ptr->info )                    // GO LEFT
          Insert( ptr->left , item ) ;
      else if  ( item  >  ptr->info )                    // GO RIGHT
          Insert( ptr->right , item ) ;
}
```

The Function DeleteItem



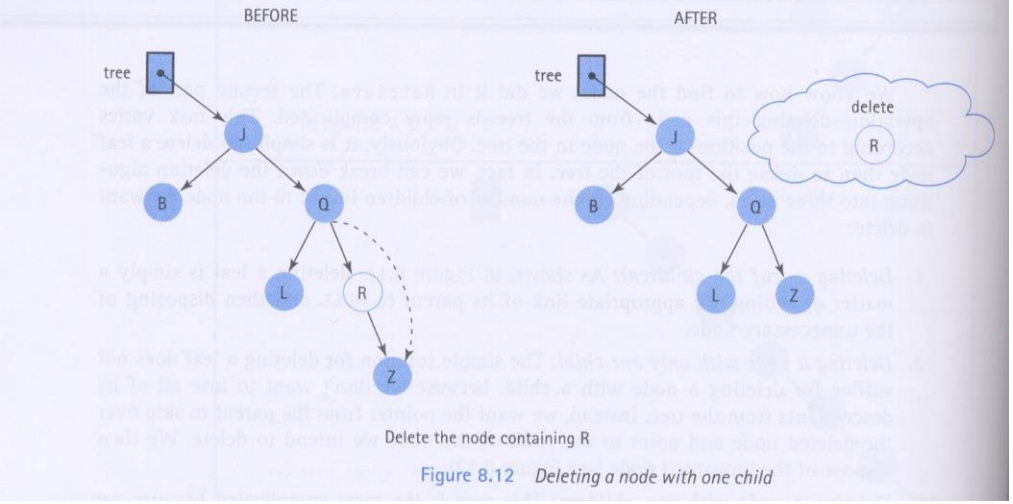
Delete the node containing Z

Figure 8.11 *Deleting a leaf node*



Delete the node containing Q

Figure 8.13 *Deleting a node with two children*



Delete the node containing R

Figure 8.12 *Deleting a node with one child*

Base Case:

If item's key matches key in Info(tree), delete node

General Case:

If item<Info(tree), Delete(Left(tree),item);

Else Delete(Right(tree),item);

Function Delete

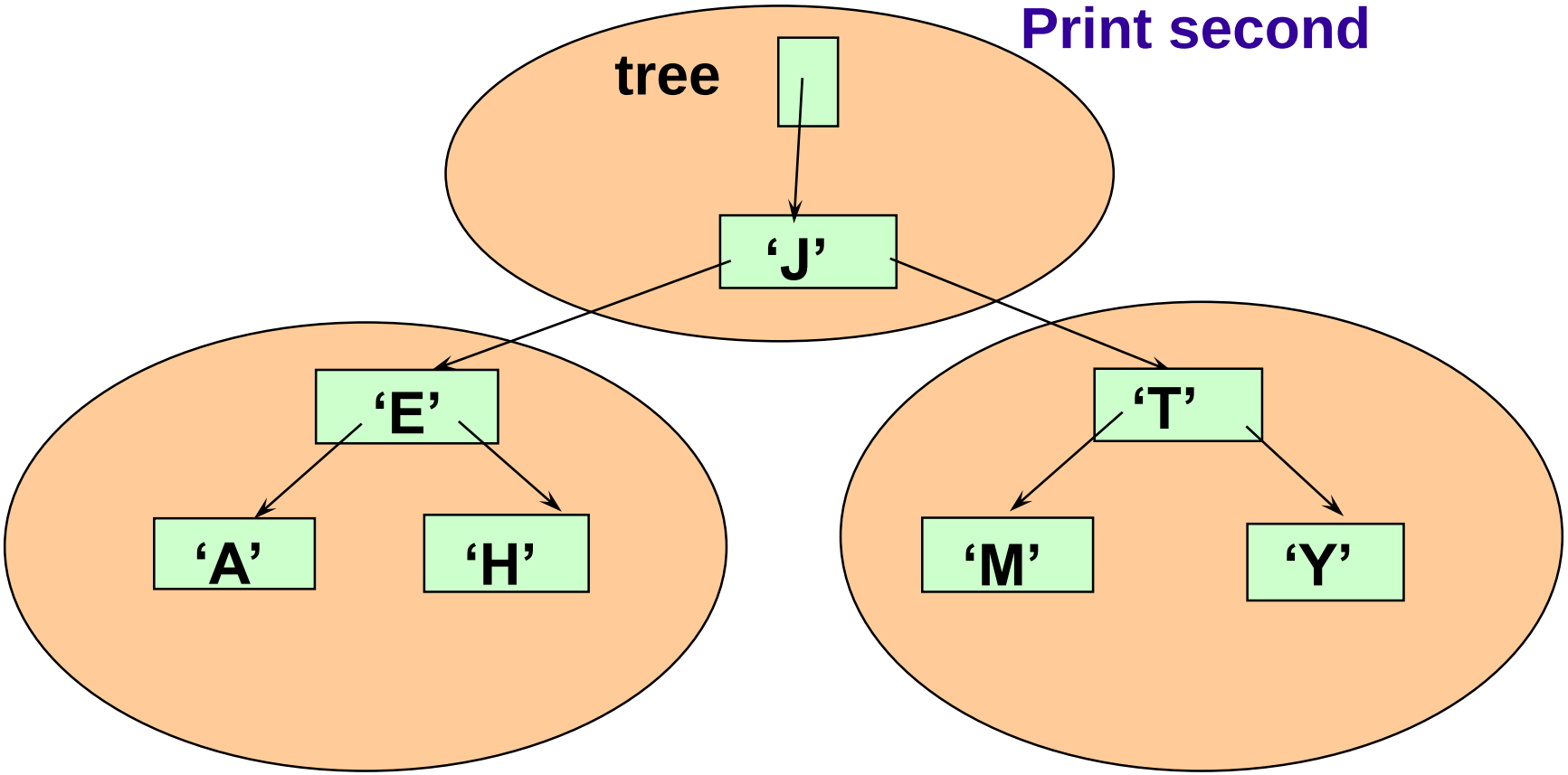
```
void Deleete(TreeNode*& tree, ItemType item);

void TreeType::DeleteItem(ItemType item)
// Calls the recursive function Delete to delete item from tree.
{
    Delete(root, item);
}

void Delete(TreeNode*& tree, ItemType item)
// Deletes item from tree
// Post : item is not in tree
{
    if (item < tree->info)
        Delete(tree->left, item);    // Look in left subtree.
    else if (item > tree ->info)
        Delete(tree->right, item); // Look in right subtree.
    else
        DeleteNode(tree);            // Node found; call DeleteNode.
}
```

Function DeleteNode()

```
void GetPredecessor(TreeNode* tree, ItemType data);
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; }
    else
    {     GetPredecessor(tree->left, data);
          tree->info = data;
          Delete(tree->left, data);//Delete predecessor node.
    }
}
```

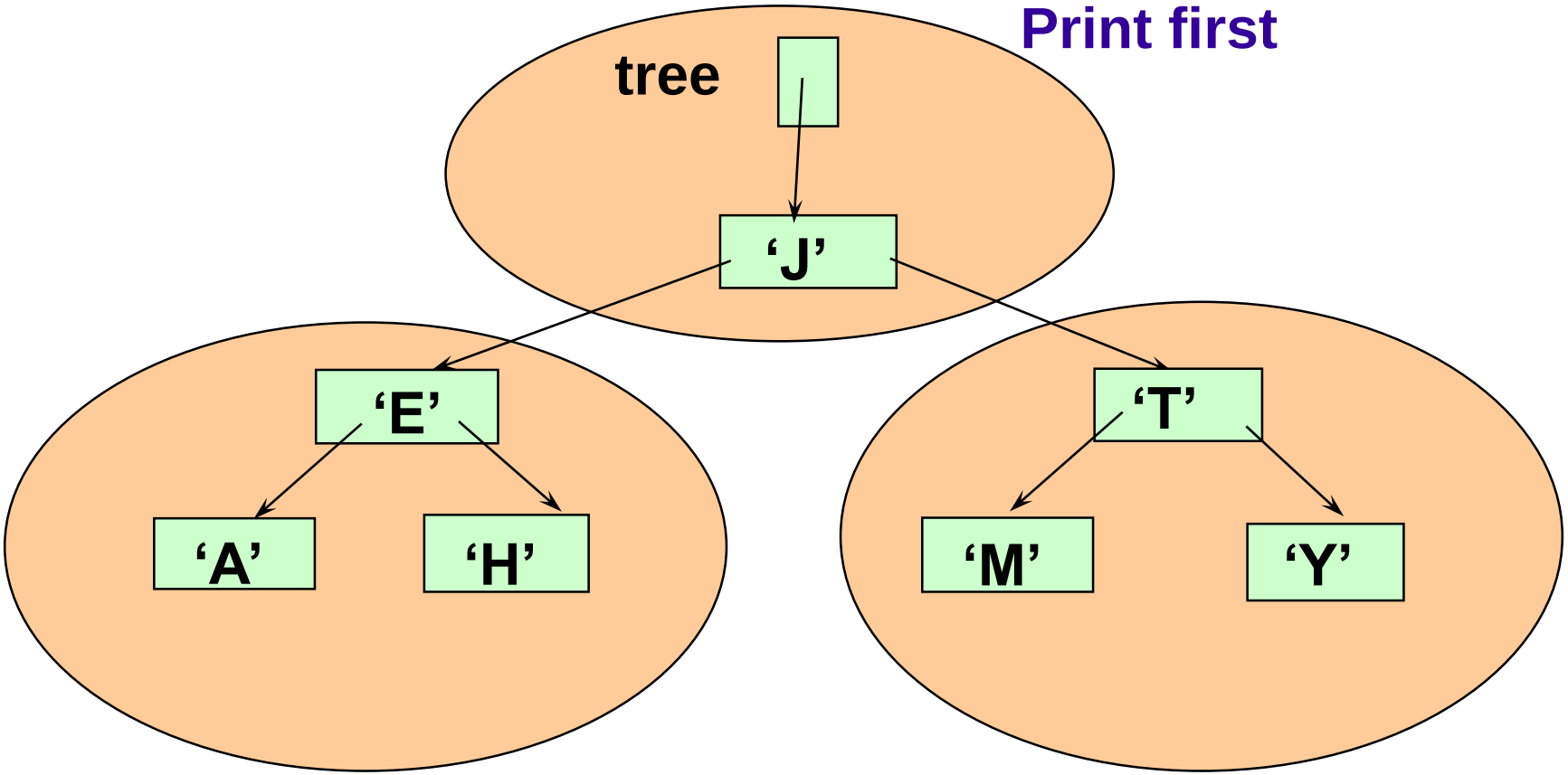
```
void GetPredecessor (TreeNode* tree, ItemType&
    data)
// Sets data to the info member of the rightmost
//  node in tree.
{
    while(tree->right != NULL)
        tree = tree->right;
    data = tree->info;
}
```

Inorder Traversal: A E H J M T Y

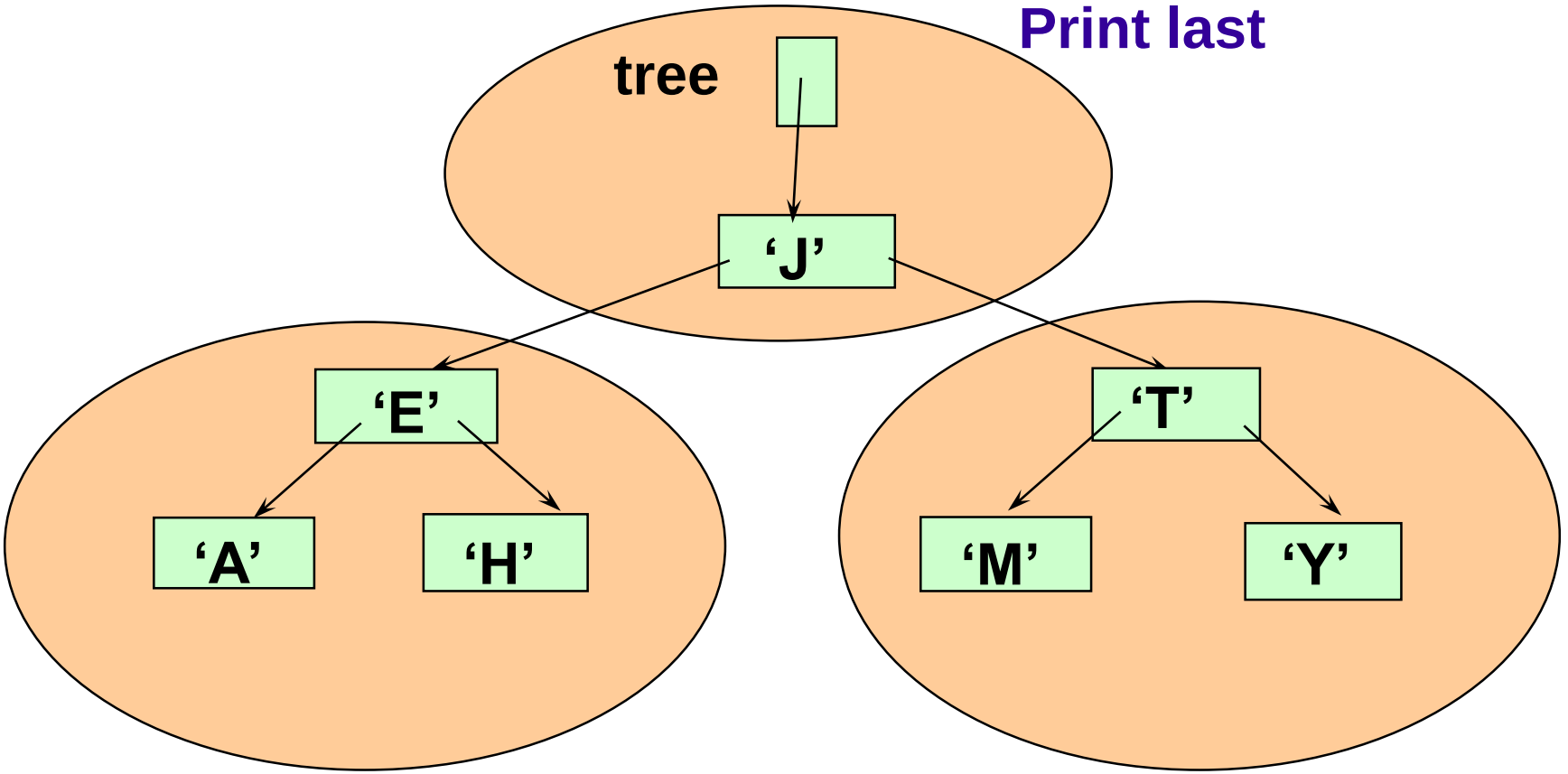


```
                                                    // INORDER TRAVERSAL
template< class  ItemType >
void   TreeType<ItemType> :: PrintTree ( ofstream&  outFile ) const
{
        Print ( root,  outFile ) ;
}


template< class  ItemType >
void  Print( TreeNode<ItemType>* ptr,  std::ofstream&  outFile )
{     if  ( ptr != NULL )
      {
          Print( ptr->left , outFile ) ;        // Print left subtree

          outFile  <<  ptr->info ;

          Print( ptr->right, outFile ) ;        // Print right subtree

       }
}
```

Preorder Traversal: J E A H T M Y



Postorder Traversal: A H E M Y T J



```
template< class  ItemType >
TreeType<ItemType> :: ~TreeType (  )            // DESTRUCTOR
{
     Destroy ( root ) ;
}

template< class  ItemType >
void   Destroy ( TreeNode<ItemType>* ptr )
// Post:  All nodes of the tree pointed to by ptr are deallocated.

{     if  ( ptr != NULL )
      {
          Destroy ( ptr->left ) ;
          Destroy ( ptr->right ) ;
          delete  ptr ;
      }
}
```

```
template< class  ItemType >
Void TreeType<ItemType> :: ResetTree (OrderType order )
{
    switch (order) {
            case PRE_ORDER: PreOrder(root, preQue); break;
            case IN_ORDER:    InOrder(root, inQue); break;
            case POST_ORDER: PostOrder(root, postQue); break;
}

template< class  ItemType >
void   GetNextItem ( ItemType& item, 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;
            }
}
```