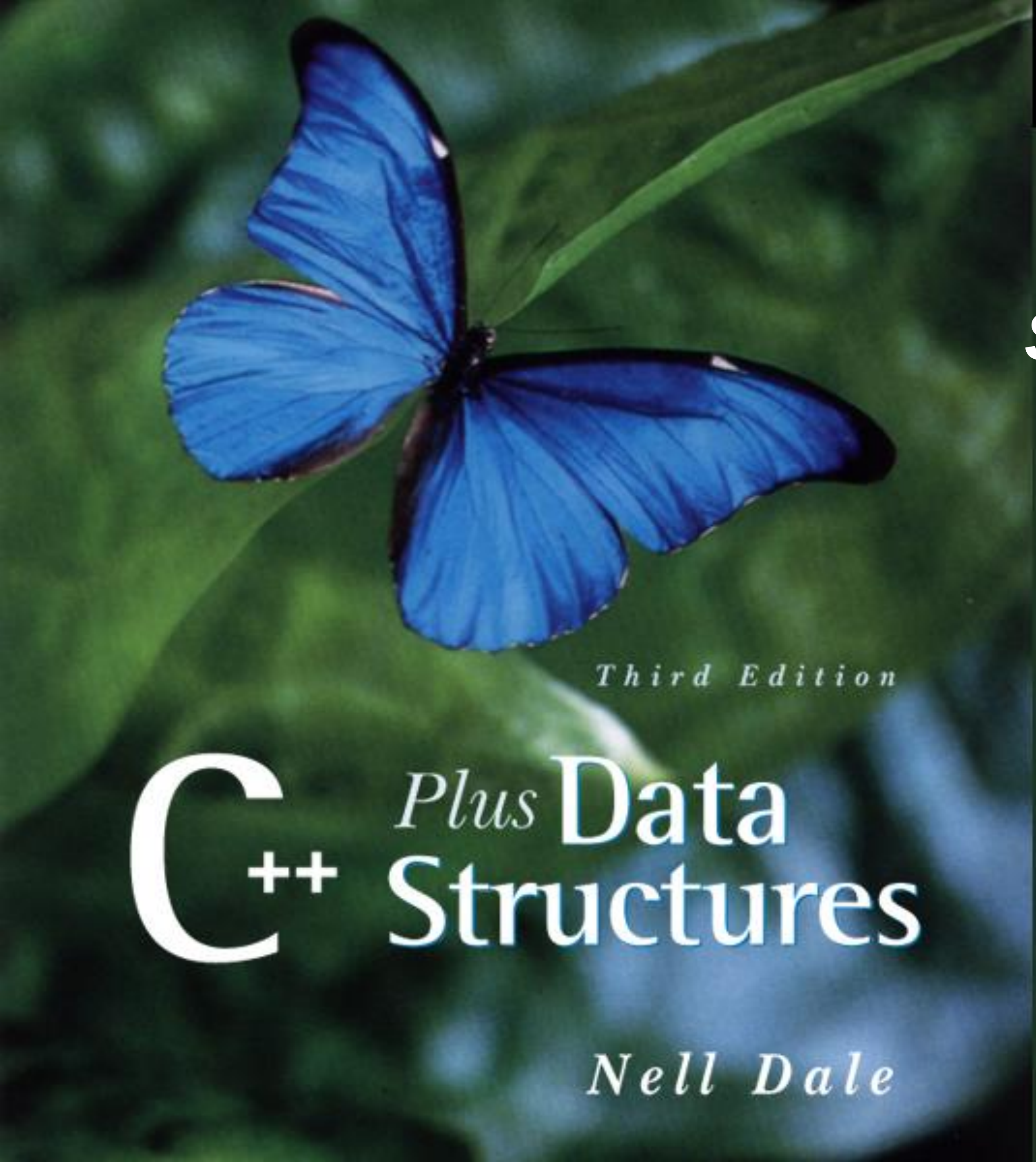
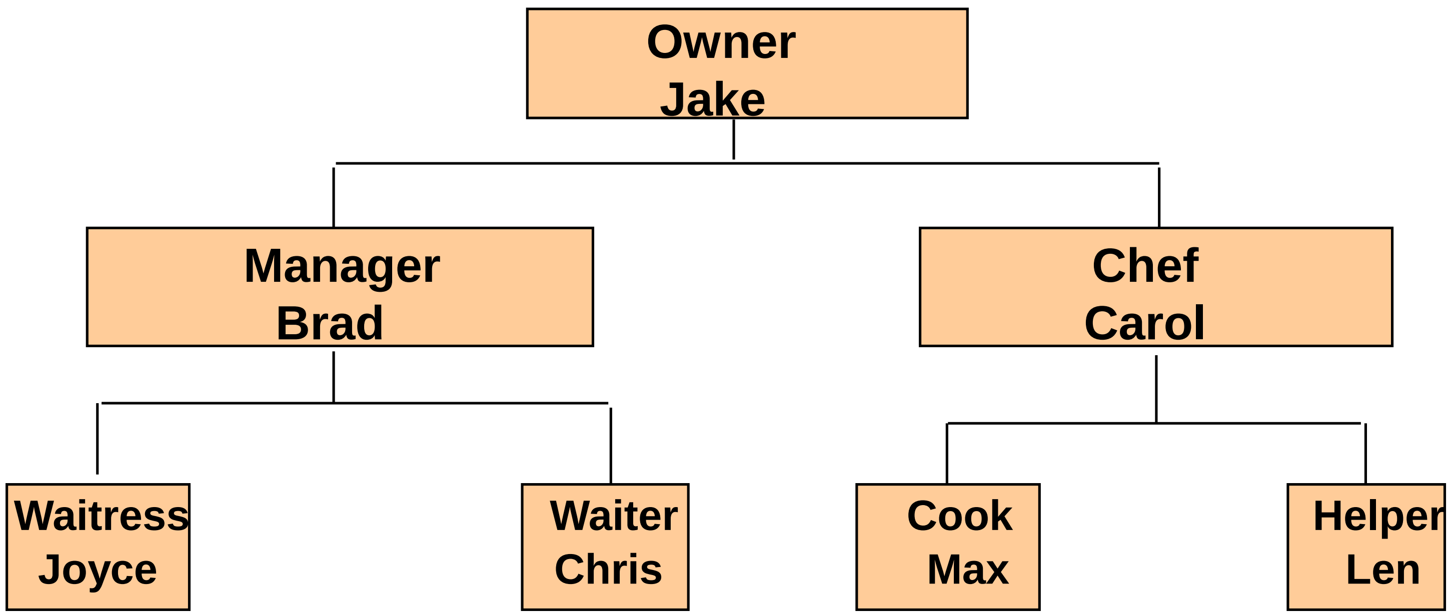


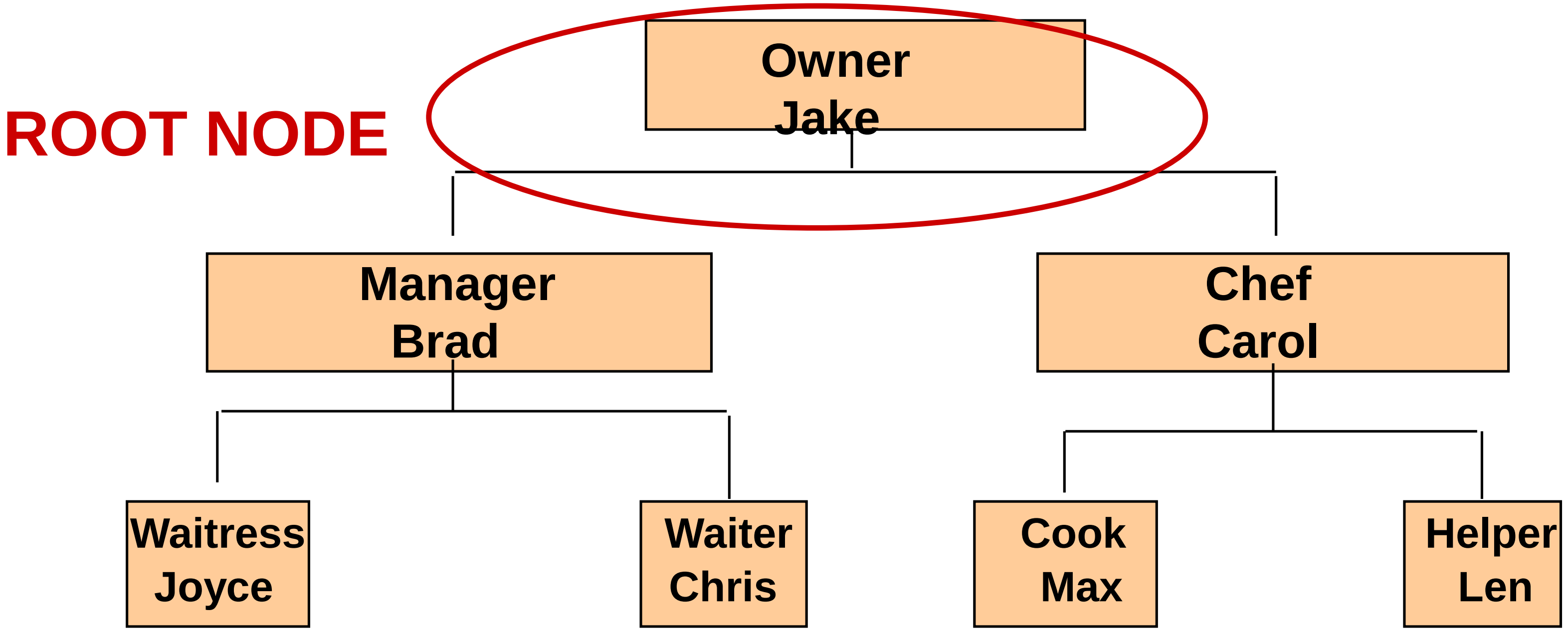
Chapter

8-1 Binary Search Trees

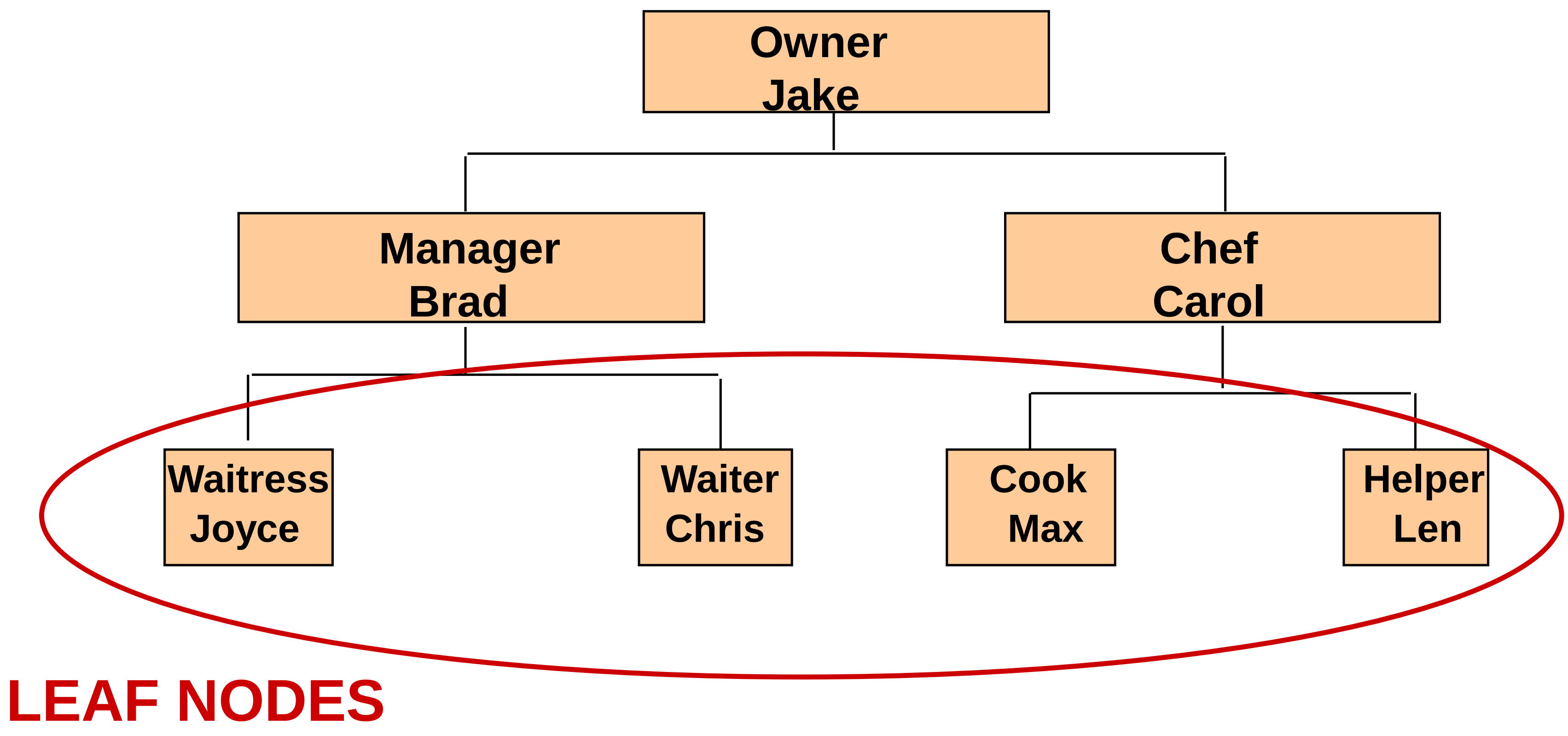
Jake’s Pizza Shop



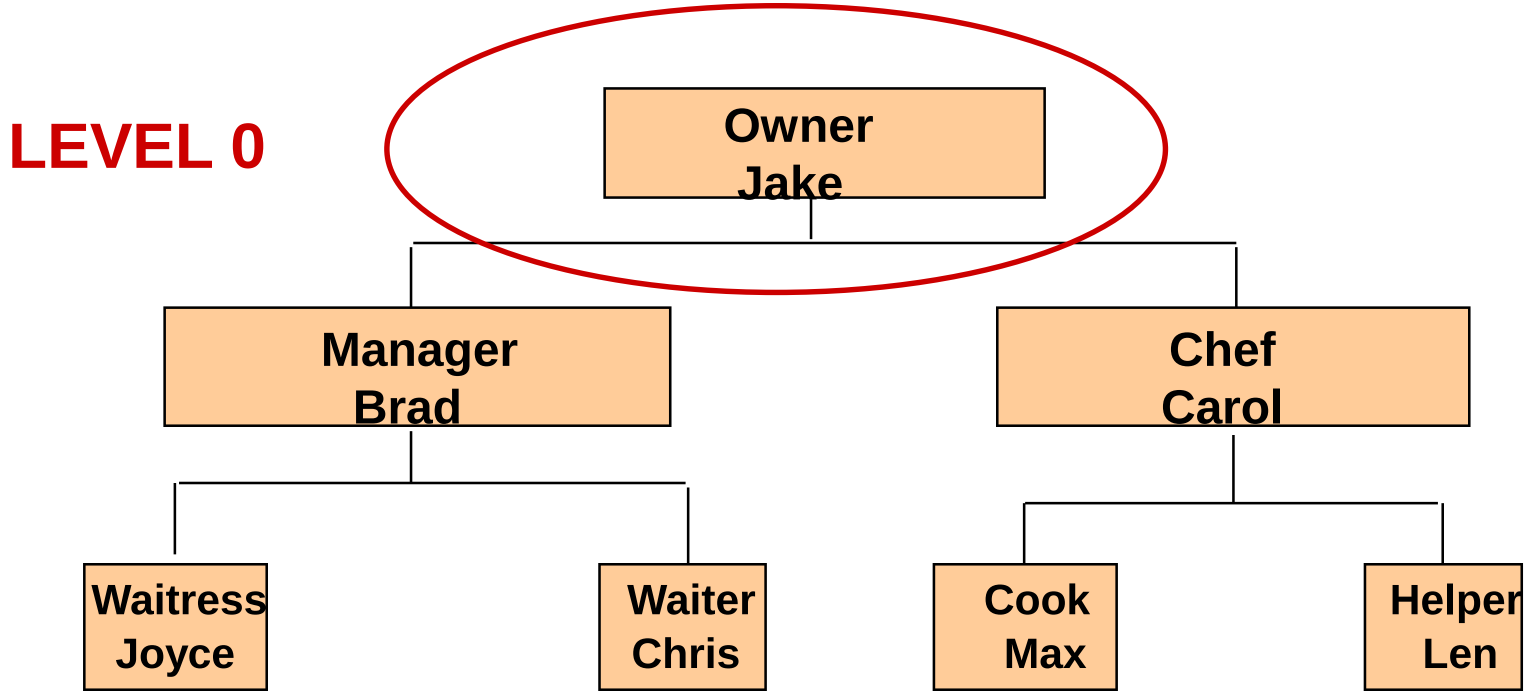
A Tree Has a Root Node



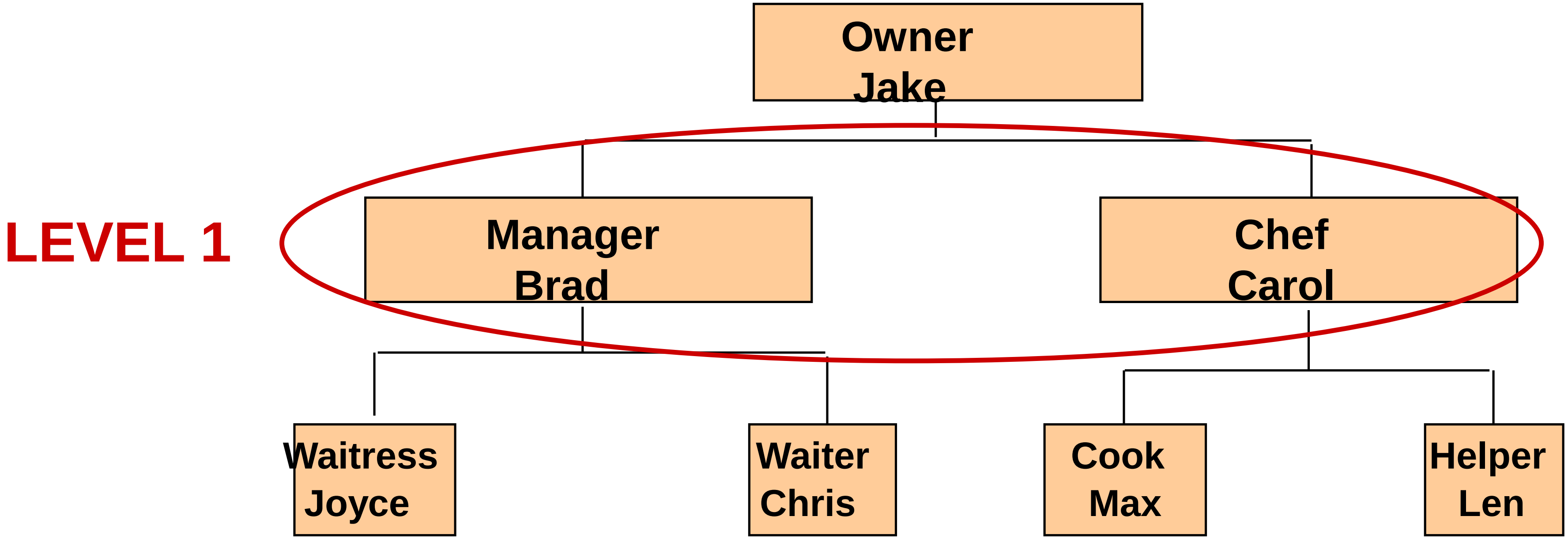
Leaf Nodes have No Children



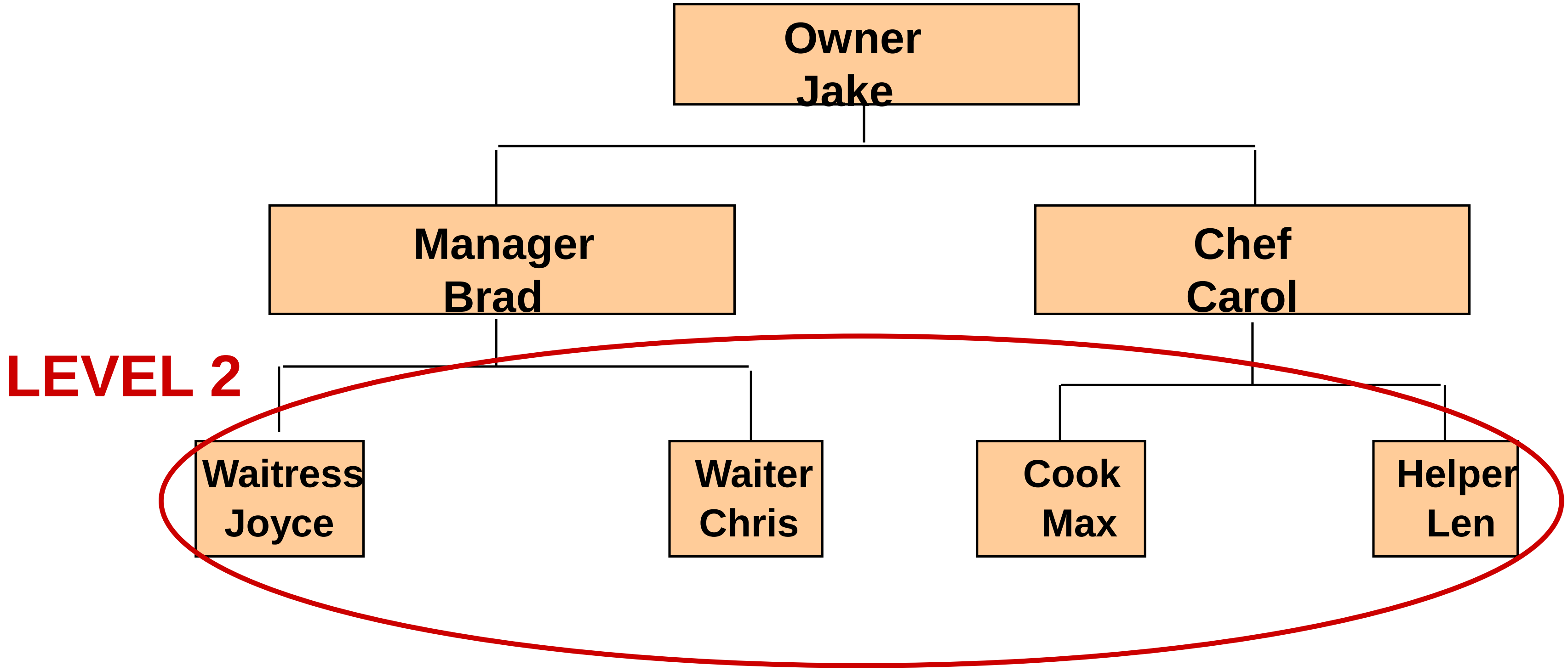
A Tree Has Leaves



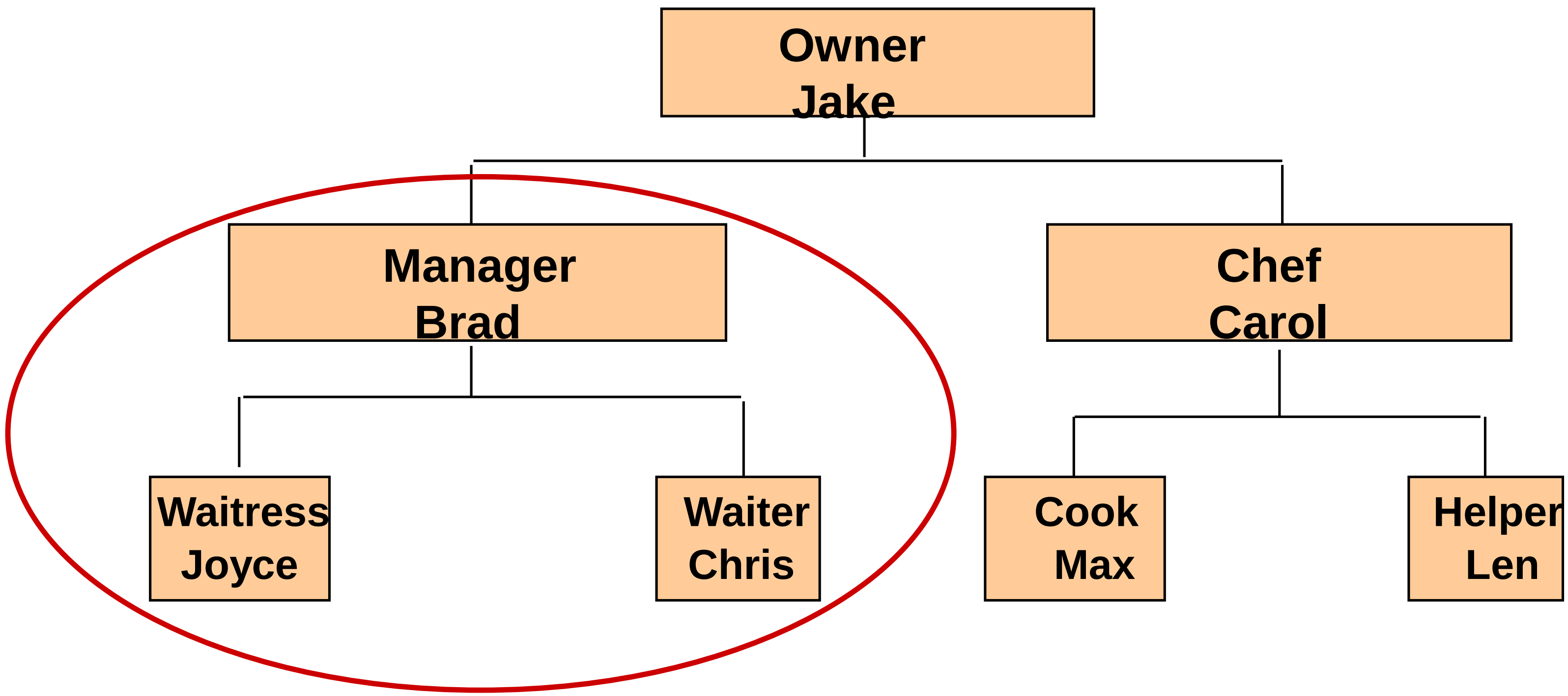
Level One


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

Level Two

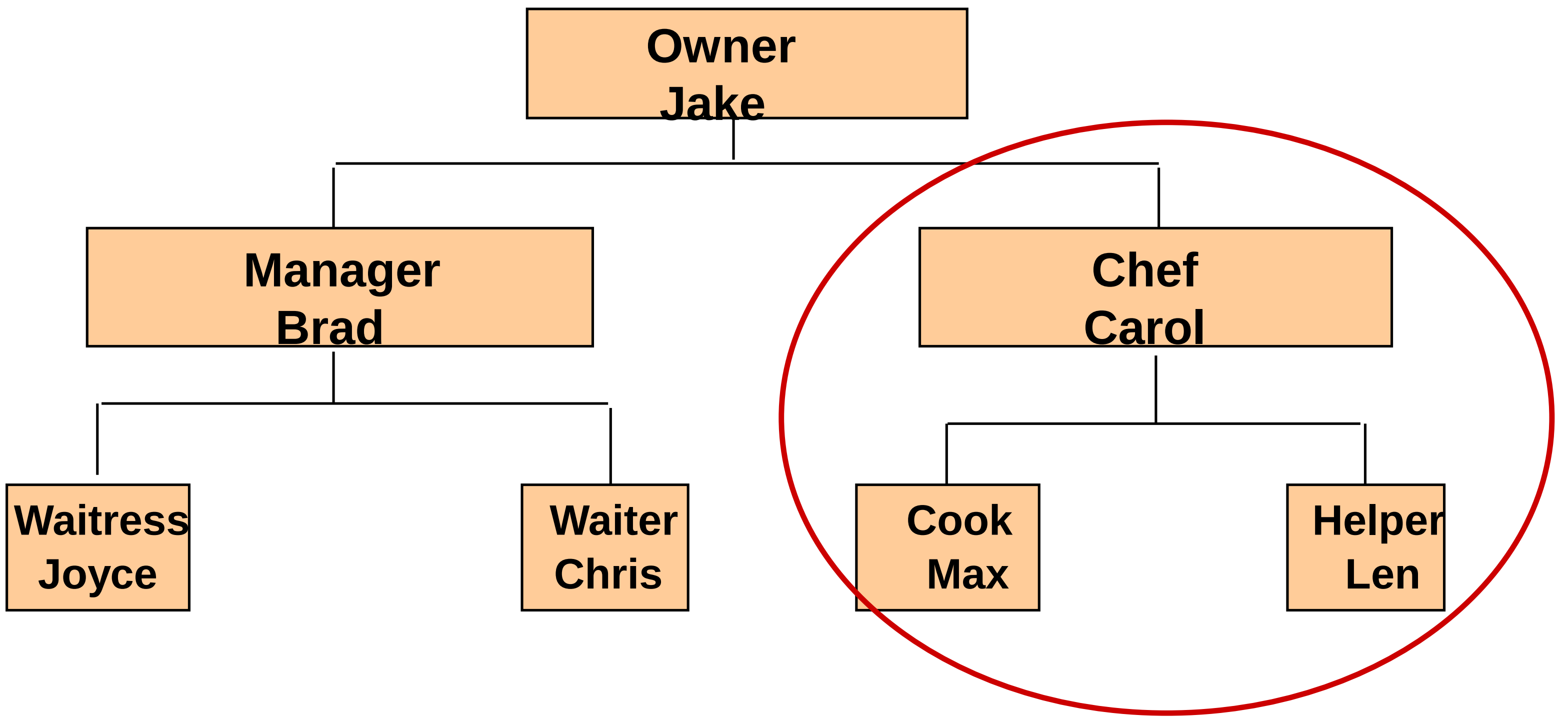

Owner
Jake
Manager
Brad
Chef
Carol
LEVEL 2
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

A Subtree



LEFT SUBTREE OF ROOT NODE

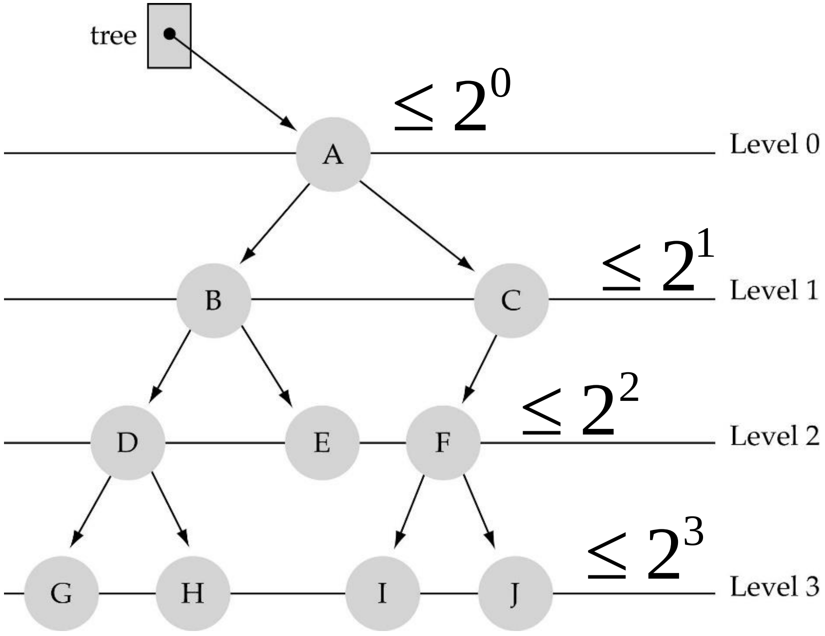
Another Subtree



RIGHT SUBTREE OF ROOT NODE

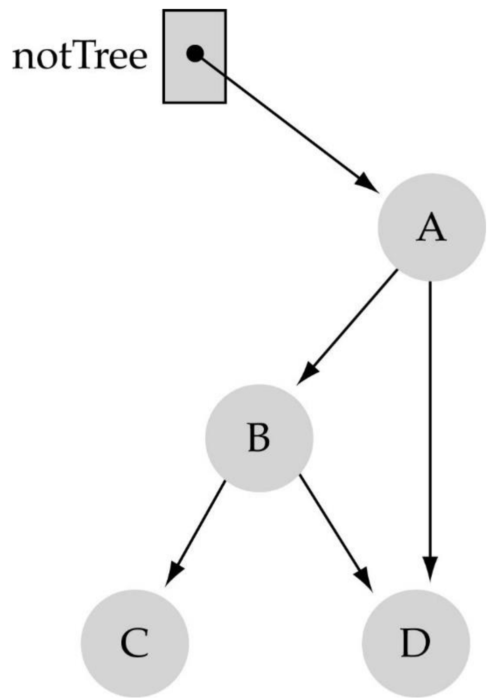
What is a binary tree?

- *Property1:* each node can have up to two successor nodes (*children*)
  - The predecessor node of a node is called its *parent*
  - The "beginning" node is called the *root* (no parent)
  - A node without *children* is called a *leaf*



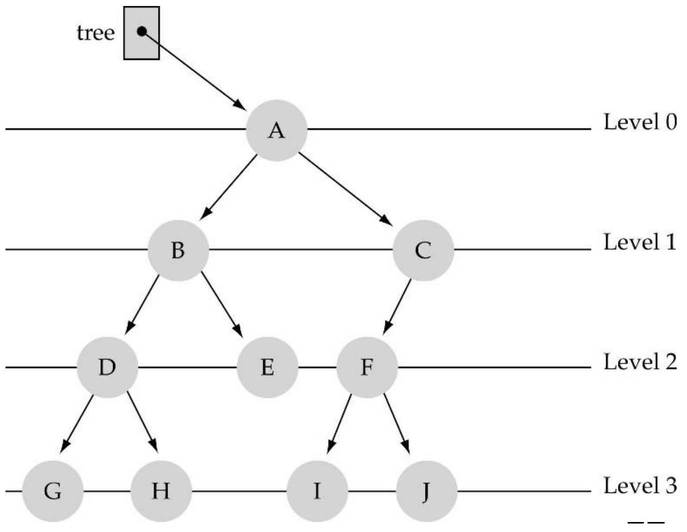
What is a binary tree? (cont.)

- *Property2:* a unique path exists from the root to every other node

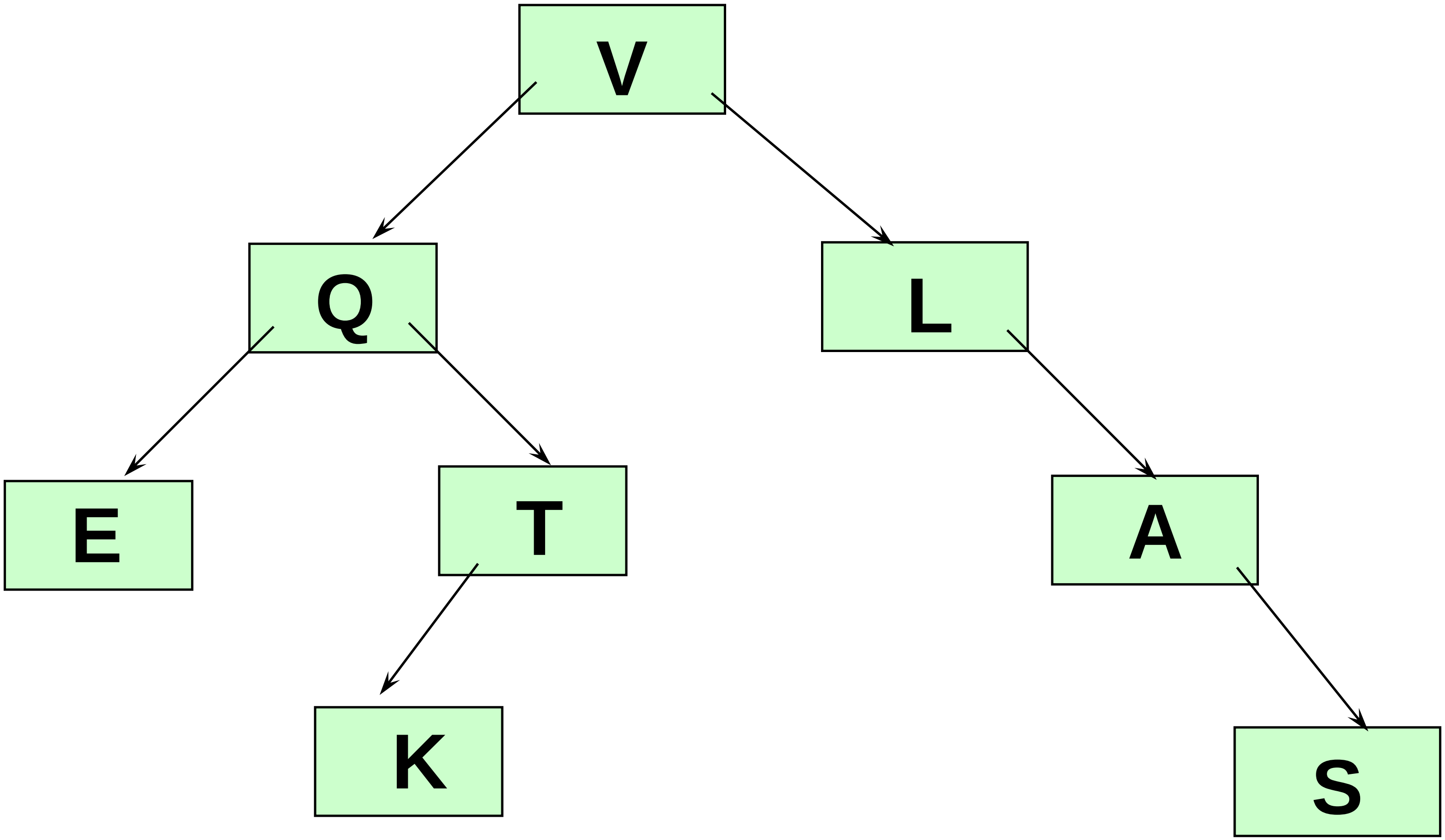


Some terminology

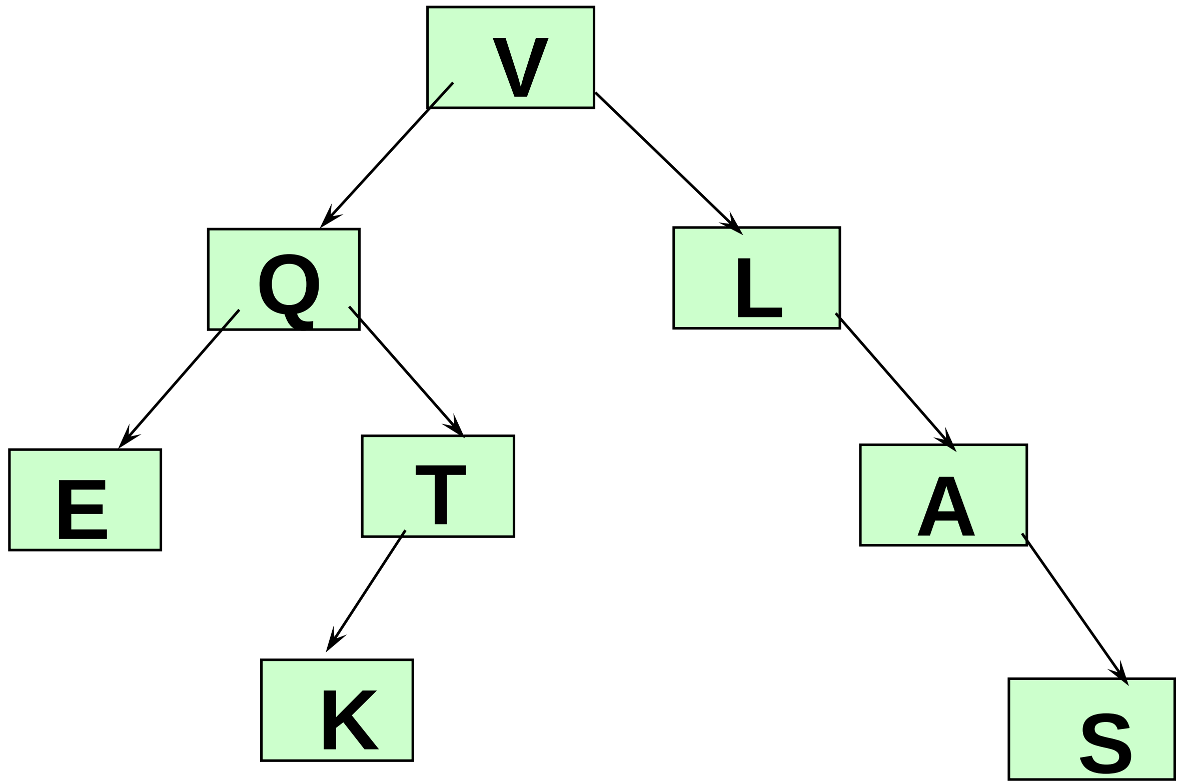
- Ancestor of a node: any node on the path from the root to that node (parent, grandparent, ...)
- Descendant of a node: any node on a path from the node to the last node in the path(child, grandchild, ...)
- Level (depth) of a node: number of edges in the path from the root to that node
- Height of a tree: number of levels (warning: some books define it as #levels - 1)



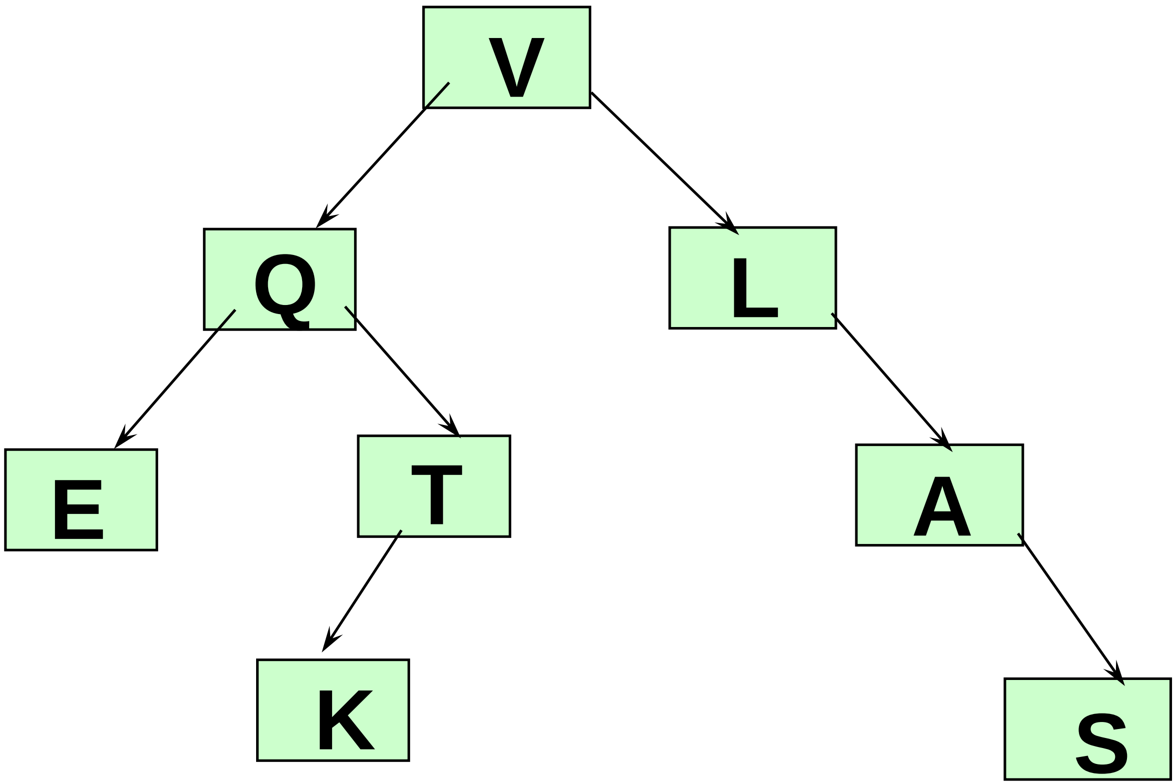
A Binary Tree



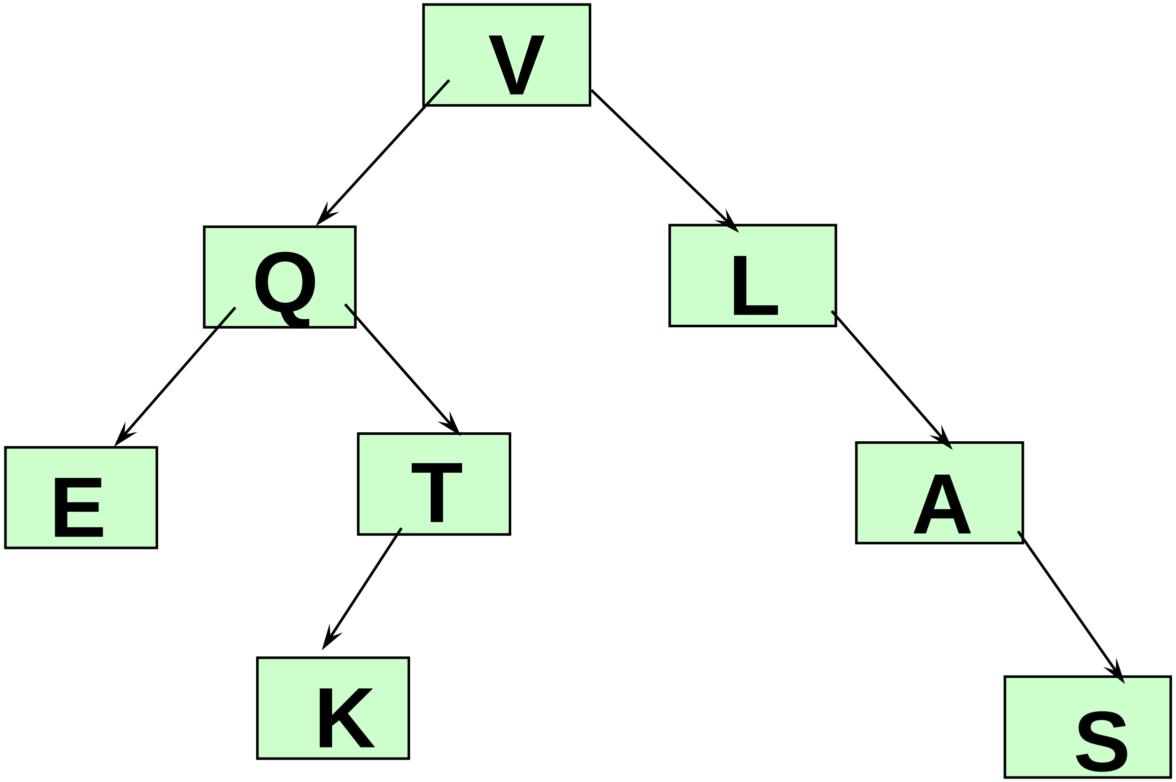
How many leaf nodes?



How many descendants of Q?



How many ancestors of K?



What is the # of nodes N of a full tree with height h?

full tree: a tree in which all of the leaves are on the same level and every nonleaf node has two children

The max #nodes at level l is 2^l

$$N = \underbrace{2^0}_{l=0} + \underbrace{2^1}_{l=1} + \ldots + \underbrace{2^{h-1}}_{l=h-1} = 2^h - 1$$

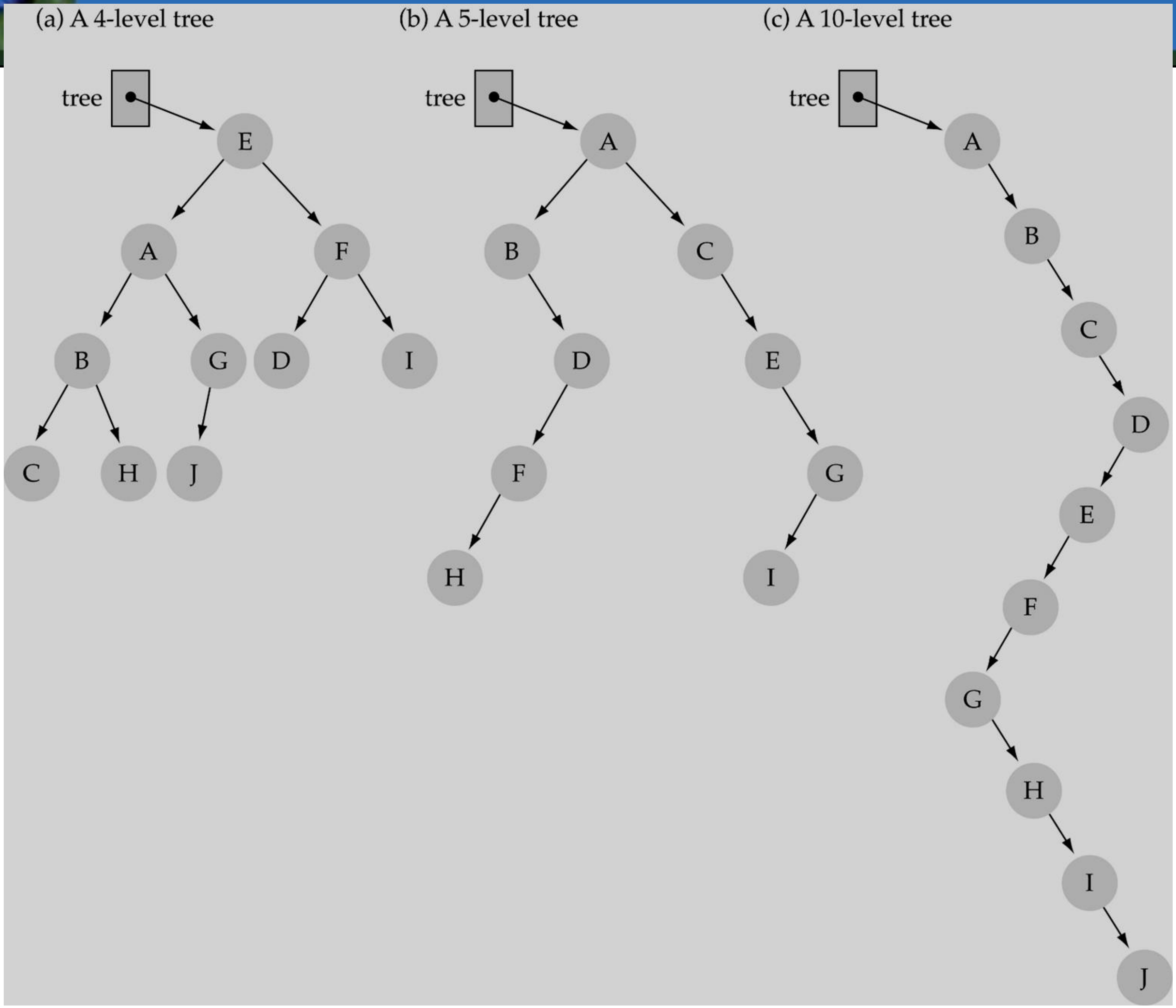
using the geometric series:

$$x^0 + x^1 + \ldots + x^{n-1} = \sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

What is the height h of a full tree with N nodes?

$$2^h - 1 = N$$
$$\Rightarrow 2^h = N + 1$$
$$\Rightarrow h = \log(N+1) \rightarrow O(\log N)$$

- The max height of a tree with *N* nodes is *N* (same as a linked list)
- The min height of a tree with *N* nodes is *log(N+1)*



(a) A 4-level tree
(b) A 5-level tree
(c) A 10-level tree
tree
E
A
F
B
G
D
I
C
H
J
tree
A
B
C
D
E
F
G
H
I
tree
A
B
C
D
E
F
G
H
I
J

Searching a binary tree

(1) Start at the root

(2) Search the tree level by level, until you find the element you are searching for
($O(N)$ time in worst case)

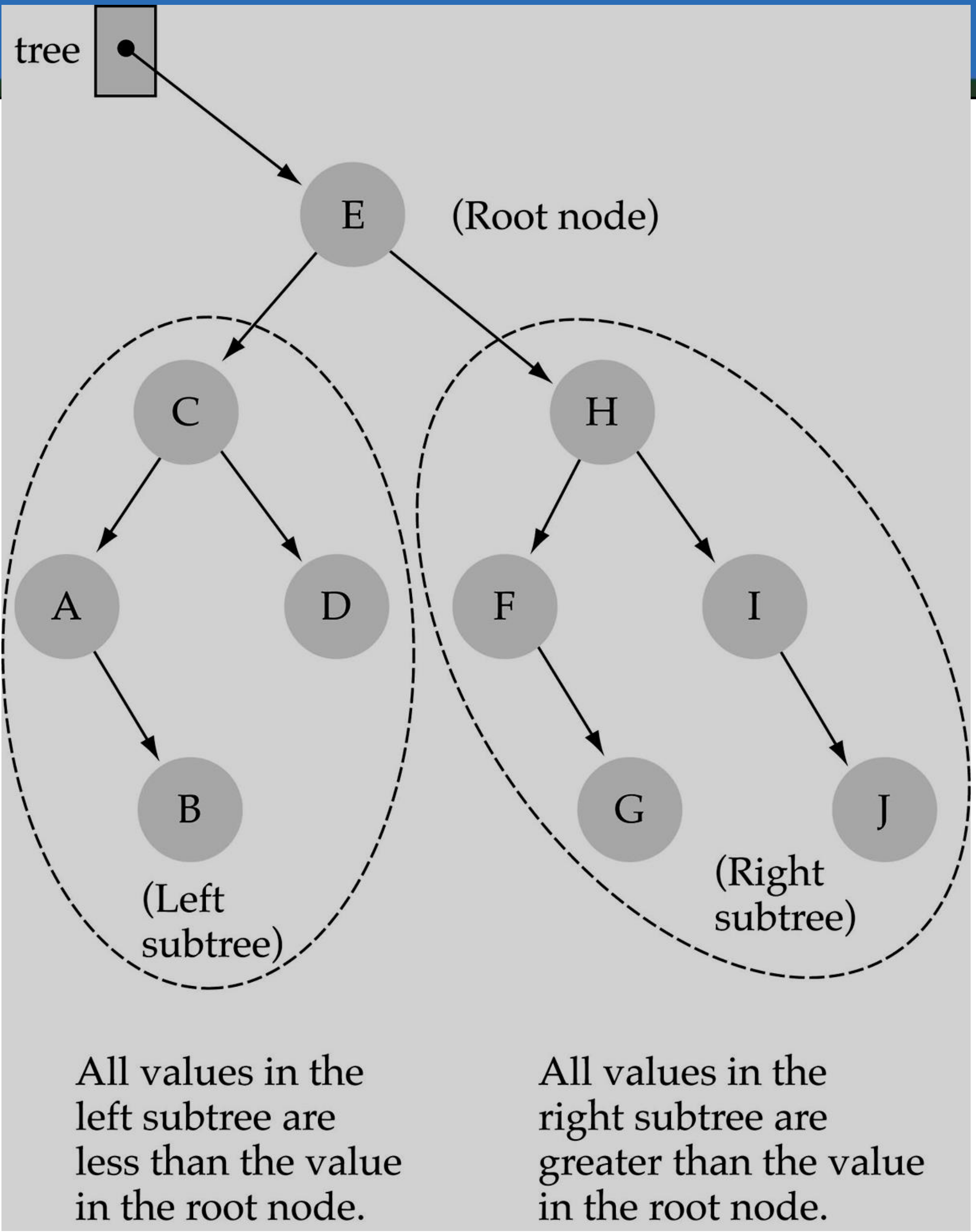
Is this better than searching a linked list?

No ---> O(N)

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.


tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

Shape of a binary search tree . . .

Depends on its key values and their order of insertion.

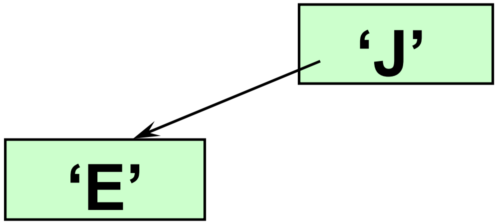
Insert the elements ‘J’ ‘E’ ‘F’ ‘T’ ‘A’ in that order.

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

‘J’

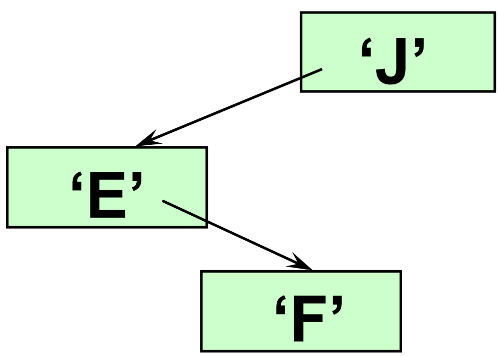
Inserting ‘E’ into the BST

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.



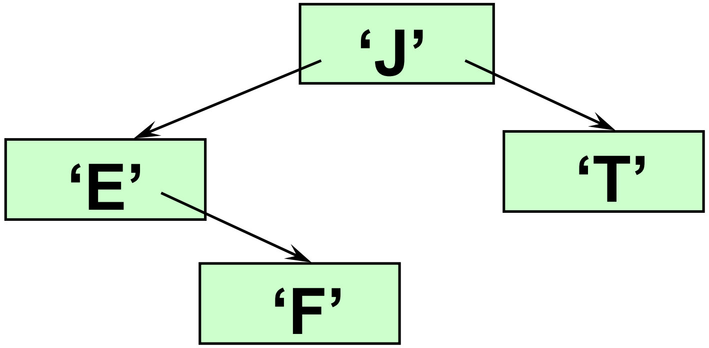
Inserting ‘F’ into the BST

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.



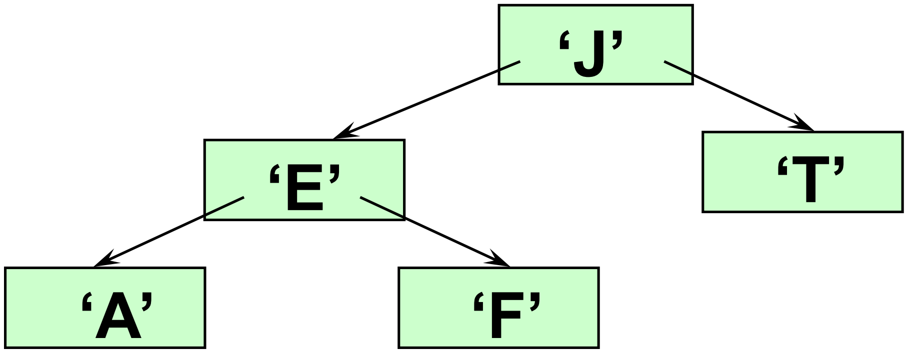
Inserting ‘T’ into the BST

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

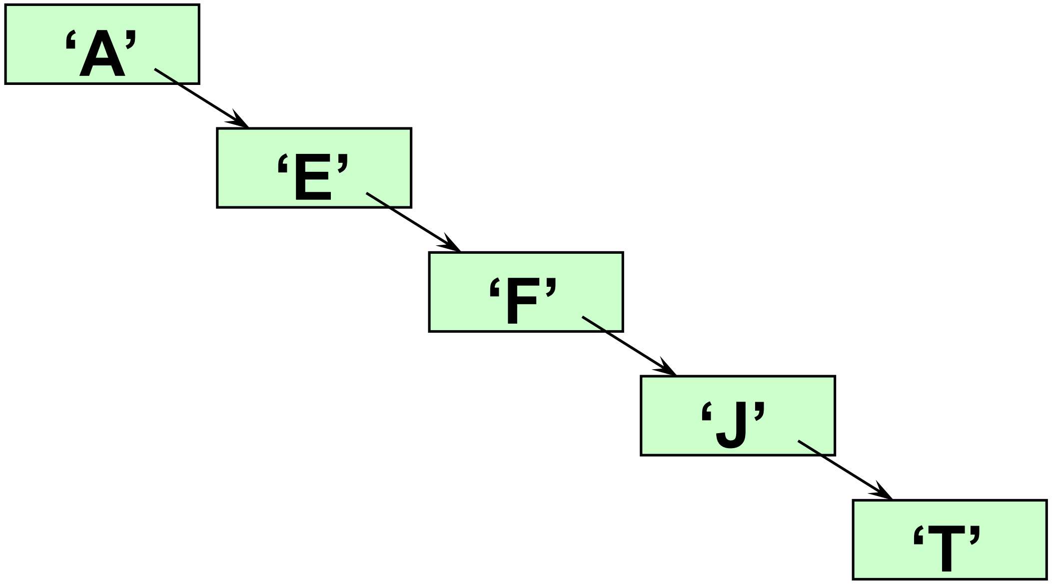
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order?

‘A’

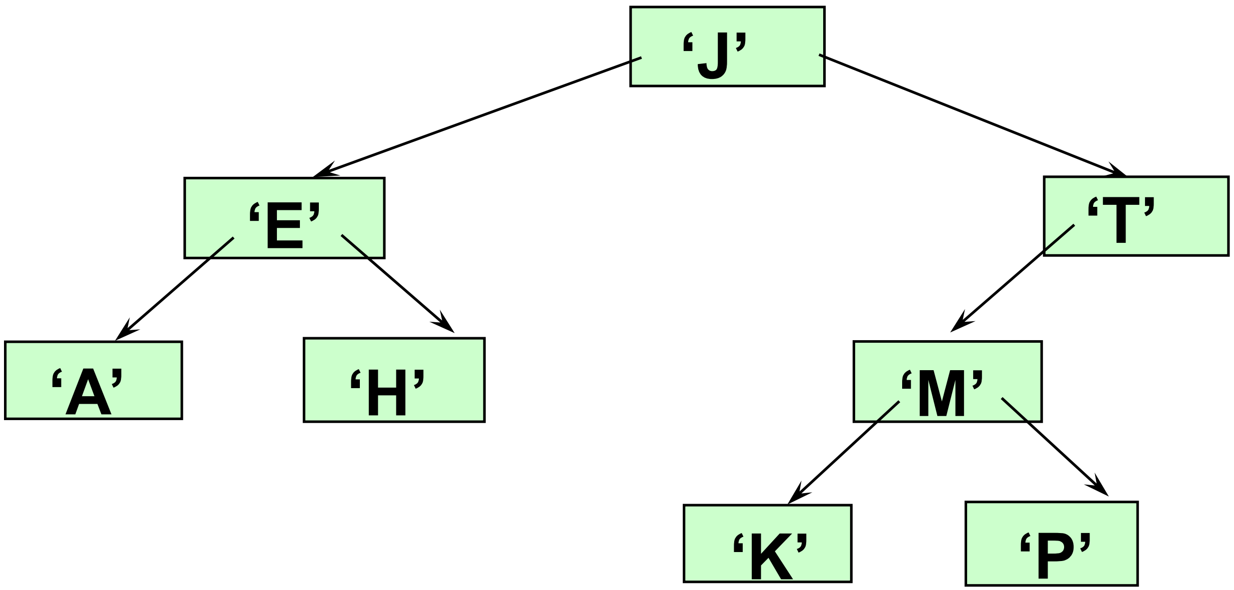
Binary search tree . . .

obtained by inserting

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



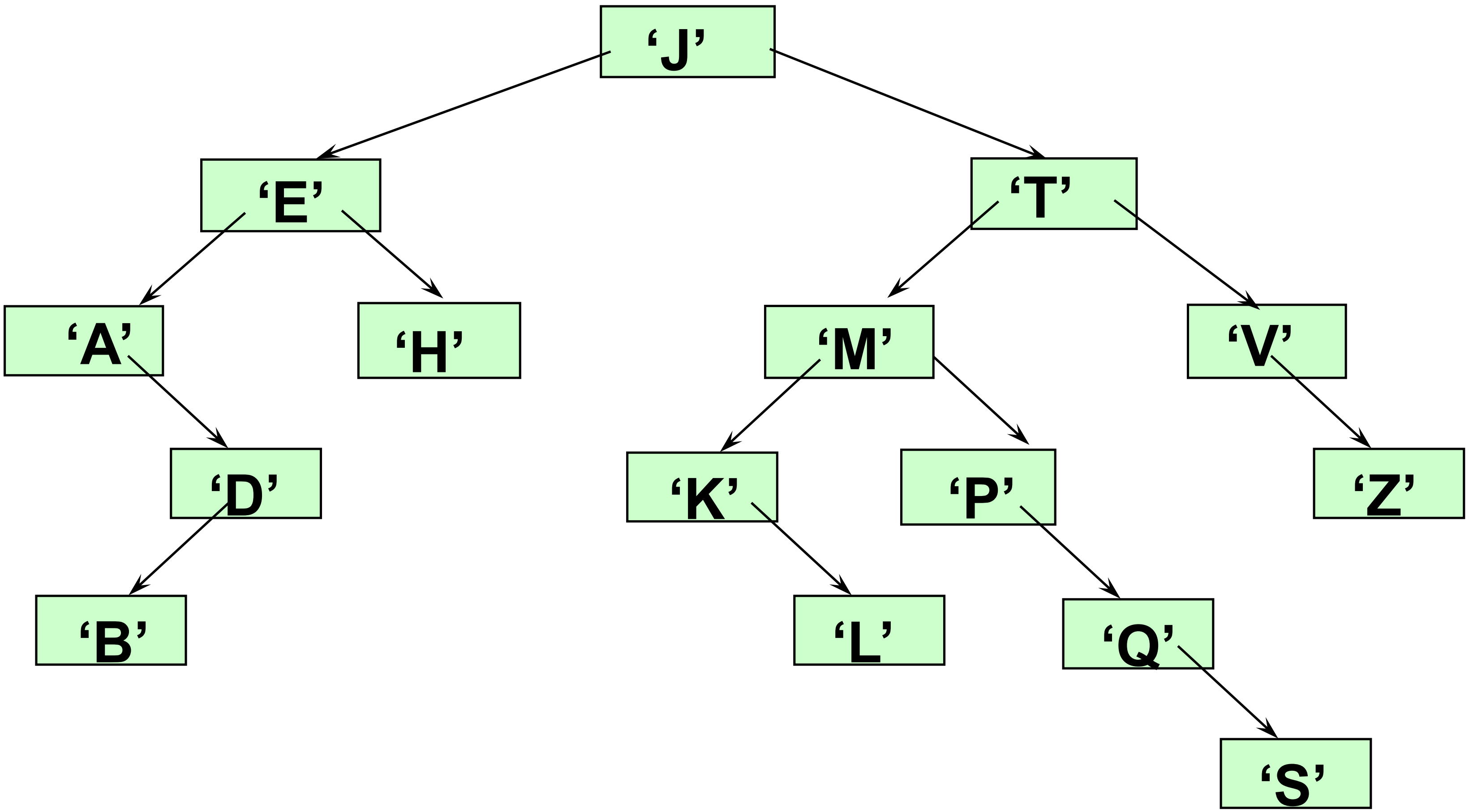
Another binary search tree



Add nodes containing these values in this order:

‘D’ ‘B’ ‘L’ ‘Q’ ‘S’ ‘V’ ‘Z’

Is ‘F’ in the binary search tree?



Searching a binary search tree

(1) Start at the root

(2) Compare the value of the item you are searching for with the value stored at the root

(3) If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*

Searching a binary search tree (cont.)

(4) If it is less than the value stored at the root, then search the left subtree

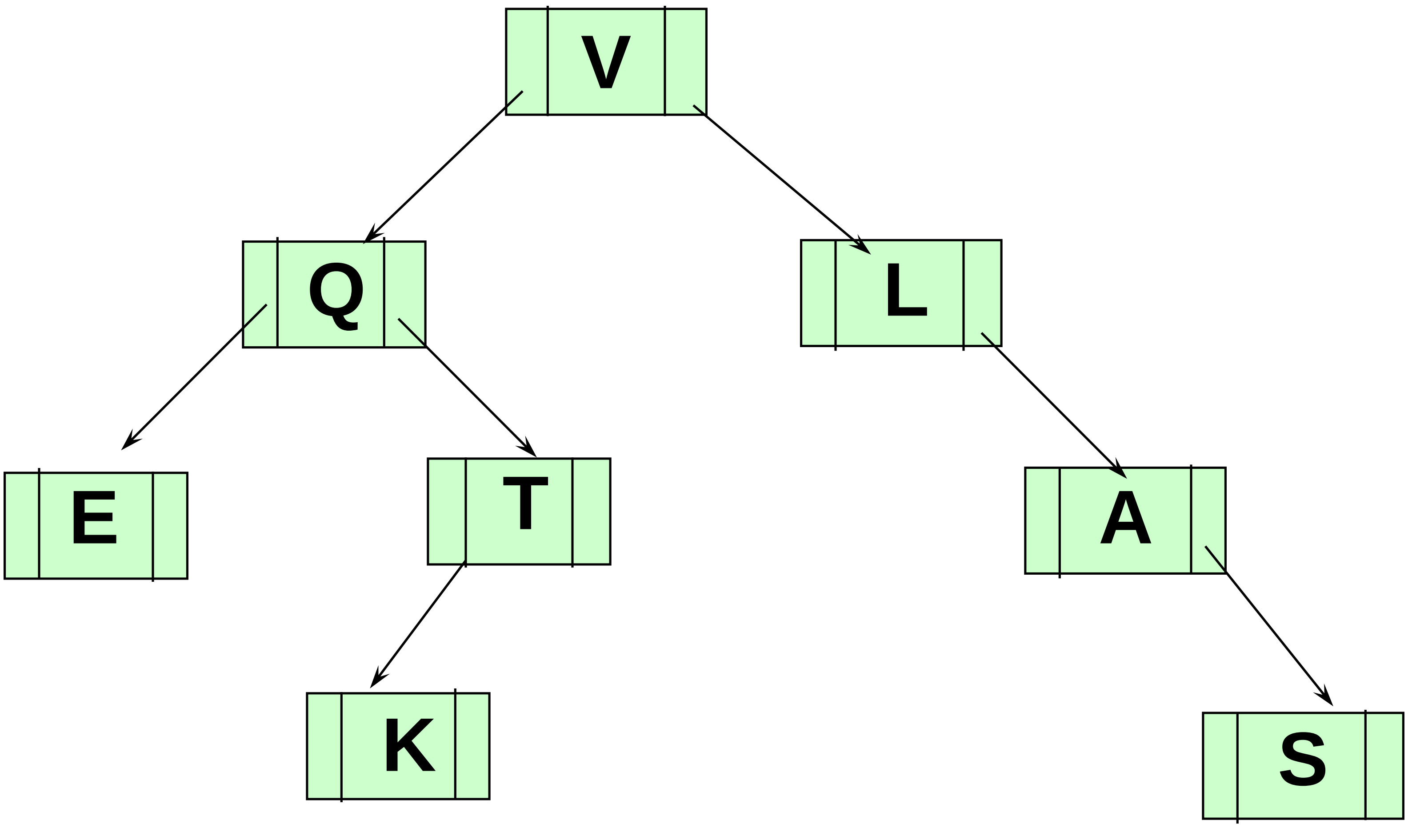
(5) If it is greater than the value stored at the root, then search the right subtree

(6) Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

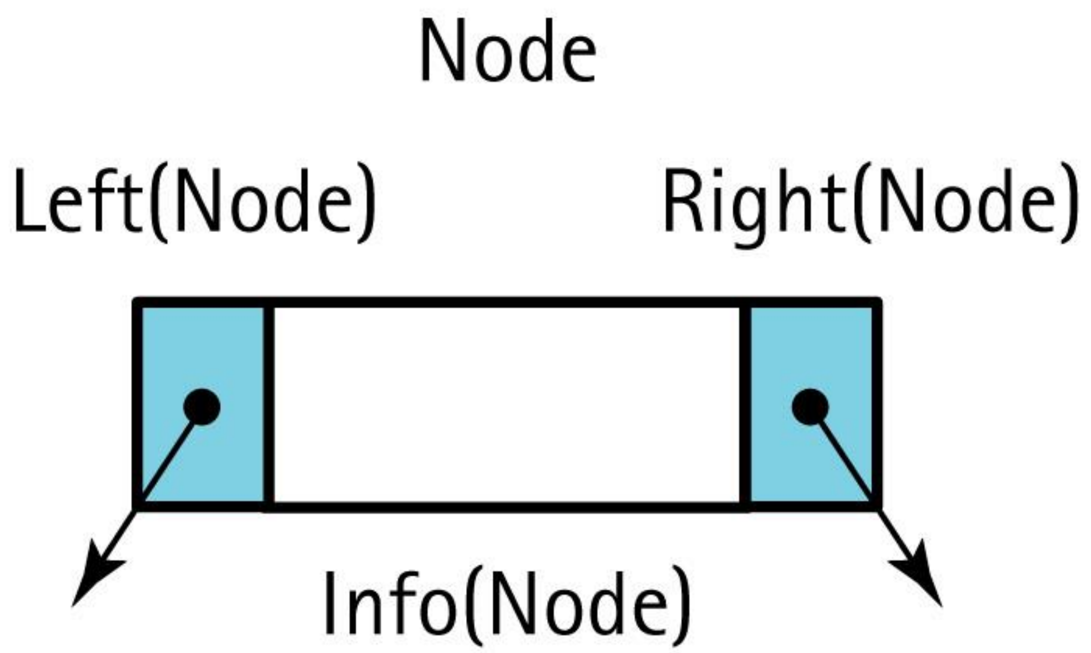
Is this better than searching a linked list?

Yes !! ---> O(logN)

Implementing a Binary Tree with Pointers and Dynamic Data



Node Terminology for a Tree Node




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

Binary Search Tree Specification

```
#include <fstream.h>

template<class ItemType>
struct TreeNode;

enum OrderType {PRE_ORDER, IN_ORDER, POST_ORDER};

template<class ItemType>
class TreeType {
  public:
    TreeType();
    ~TreeType();
    TreeType(const TreeType<ItemType>&);
    void operator=(const TreeType<ItemType>&);
```

(continues)

Binary Search Tree Specification

(cont.)

```
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    int LengthIs() const;
    void RetrieveItem(ItemType&, bool& found);
    void InsertItem(ItemType);
    void DeleteItem(ItemType);
    void ResetTree(OrderType);
    void GetNextItem(ItemType&, OrderType, bool&);
    void PrintTree(ofstream&) const;
  private:
    TreeNode<ItemType>* root;
};
```

Functions IsFull( ) & IsEmpty( )

```
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;
}
```

Function CountNodes

- Recursive implementation

  #nodes in a tree =

  #nodes in left subtree + #nodes in right subtree + 1
- What is the size factor?

  Number of nodes in the tree we are examining
- What is the base case?

  The tree is empty
- What is the general case?

  CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1

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?

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?

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?

CountNodes Version 4

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

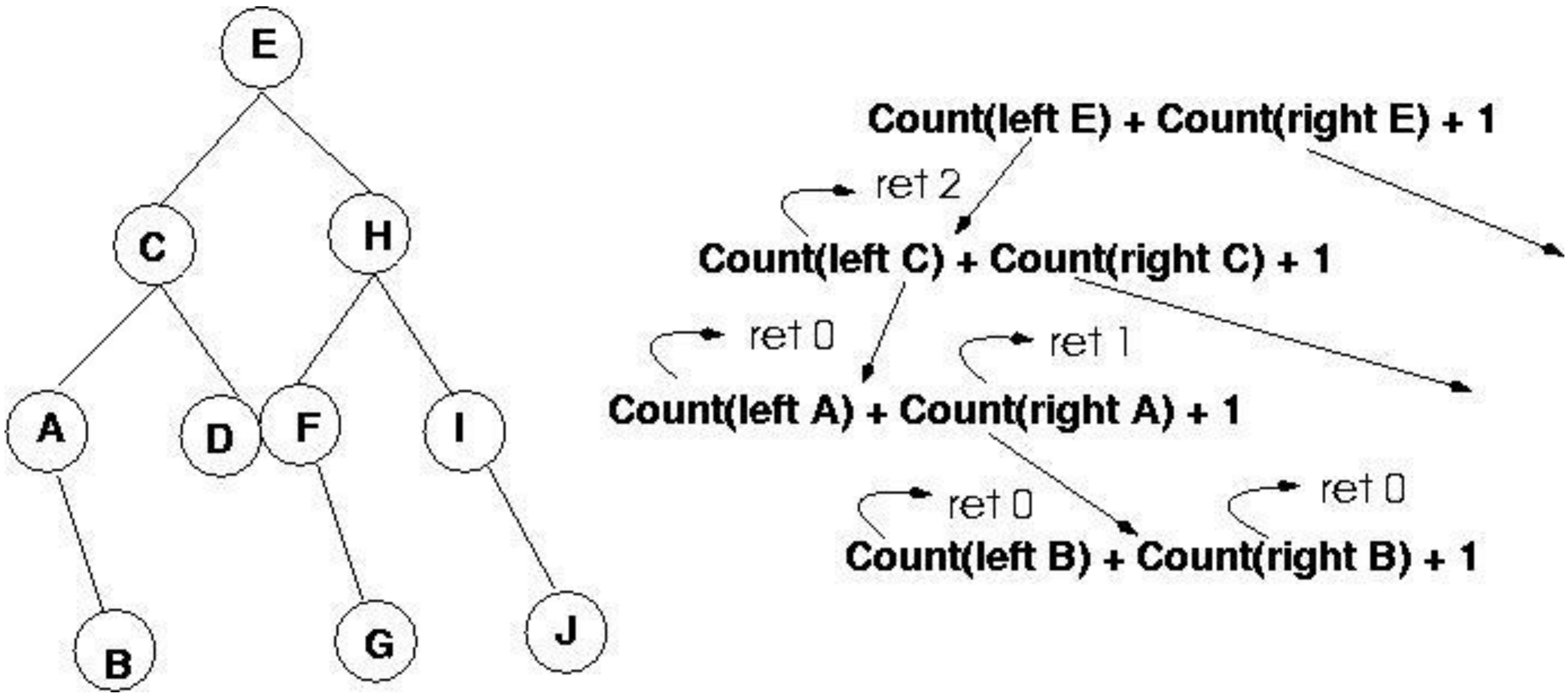
Is that all there is?

Function LengthIs()

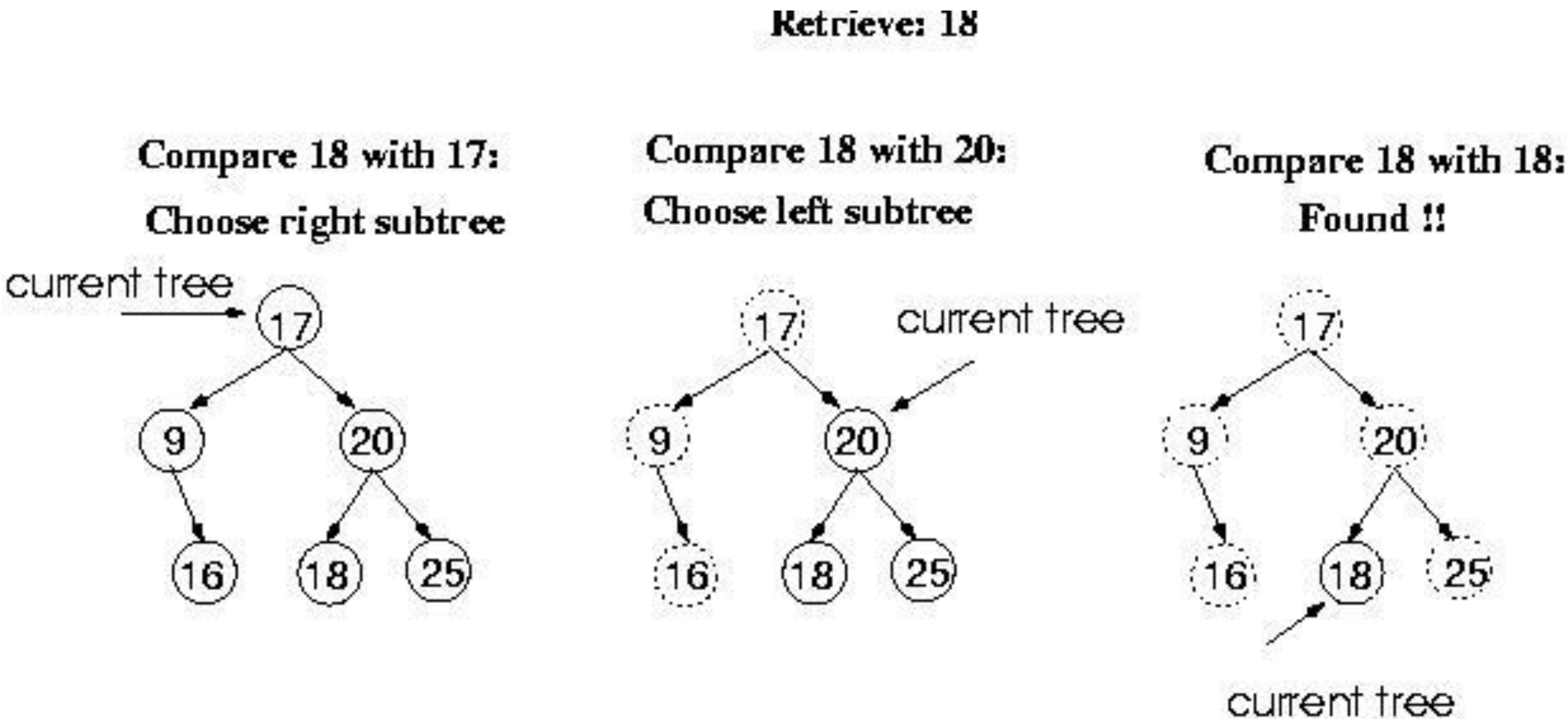
```
// Implementation of Final Version
int CountNodes(TreeNode* tree); // Pototype
int TreeType::LengthIs() const
// Class member function
{
  return CountNodes(root);
}

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

Let's consider the first few steps:



Function RetrieveItem



Function RetrieveItem

- What is the size of the problem?

  Number of nodes in the tree we are examining

- What is the base case(s)?

  1) When the key is found
  2) The tree is empty (key was not found)

- What is the general case?

  Search in the left or right subtrees

Retrieval Operation

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

void Retrieve(TreeNode* tree,
     ItemType& item, bool& found)
{
  if (tree == NULL)  // base case 2
    found = false;
  else if (item < tree->info)
    Retrieve(tree->left, item, found);
```

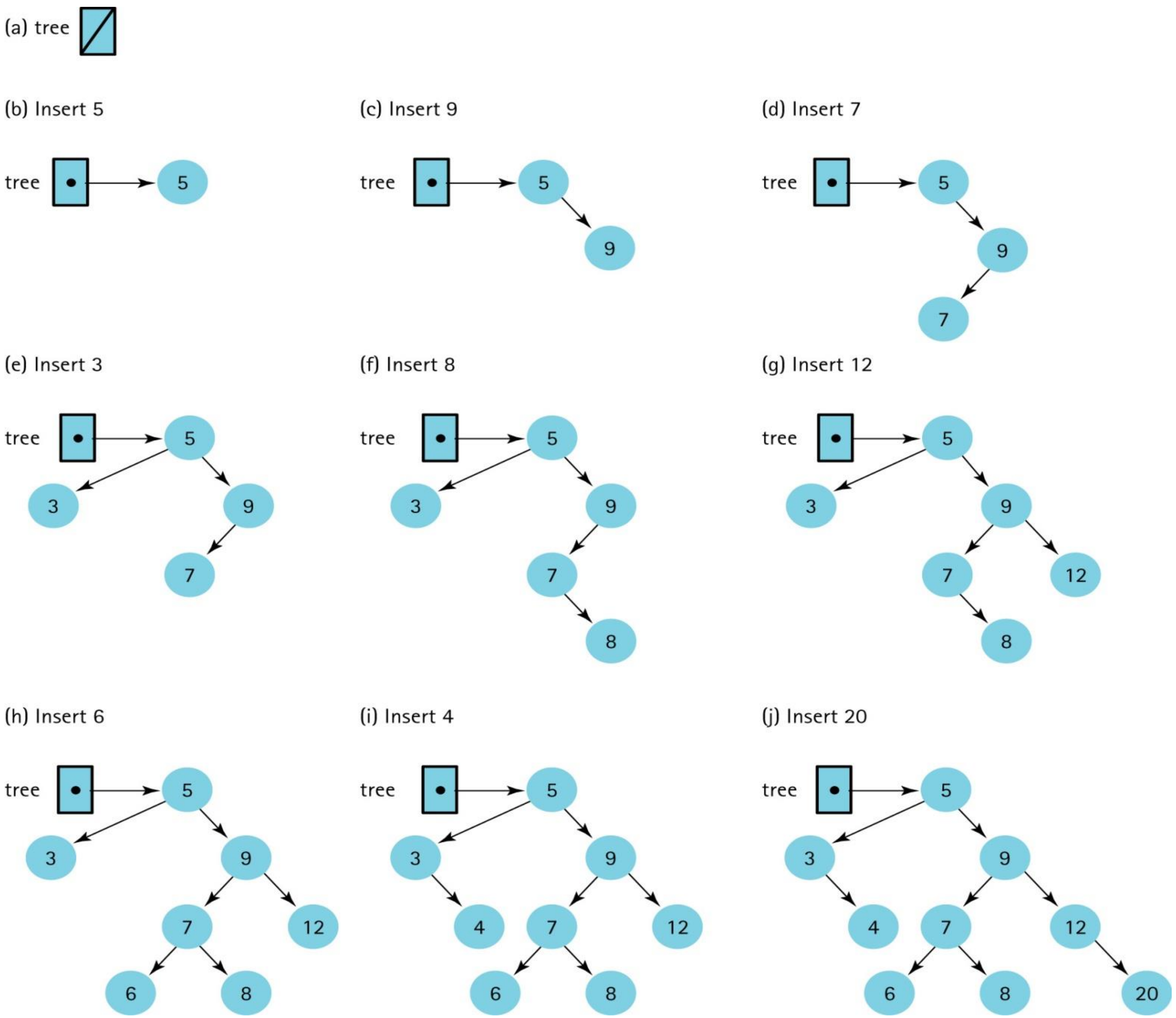
Retrieval Operation, cont.

```
else if (item > tree->info)
  Retrieve(tree->right, item, found);
else  // base case 1
   {
     item = tree->info;
     found = true;
    }
}
```

The Insert Operation

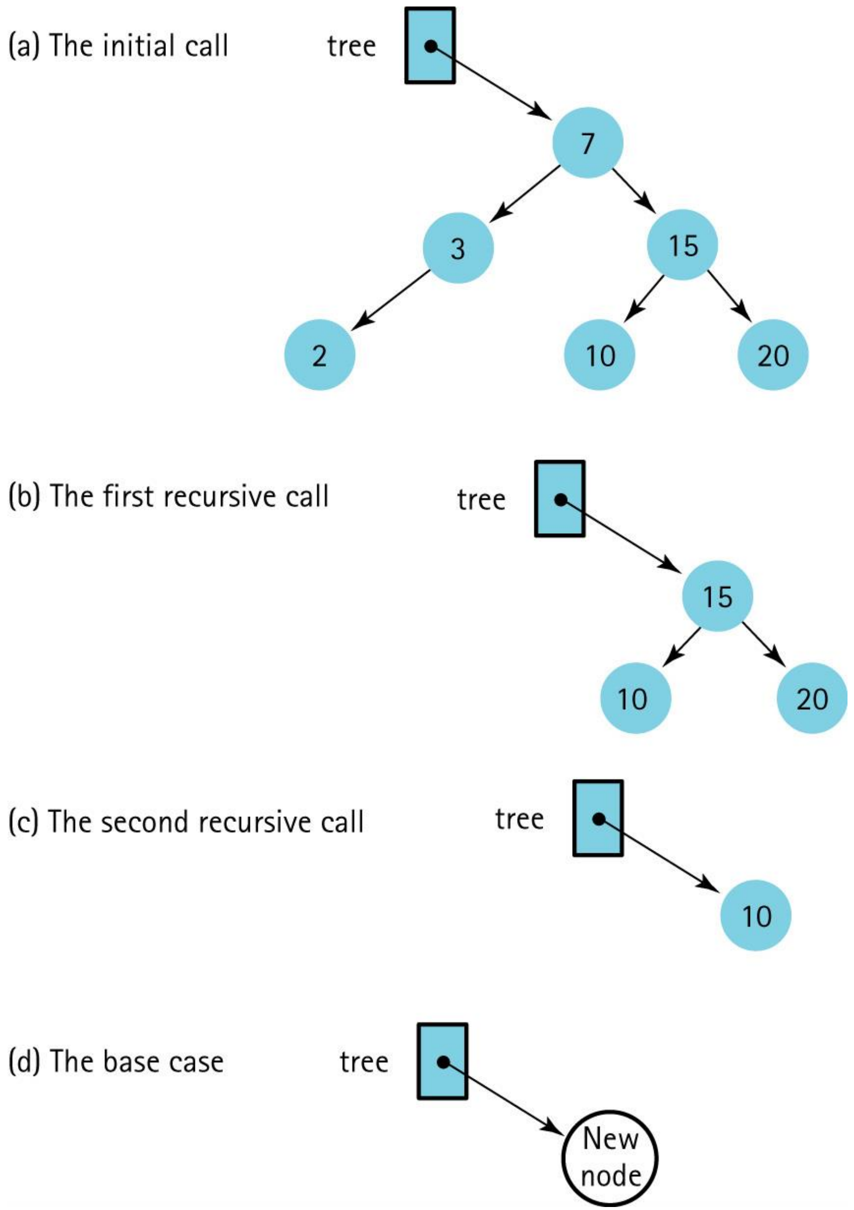
- A new node is always inserted into its appropriate position in the tree as a leaf.

Insertions into a Binary Search Tree



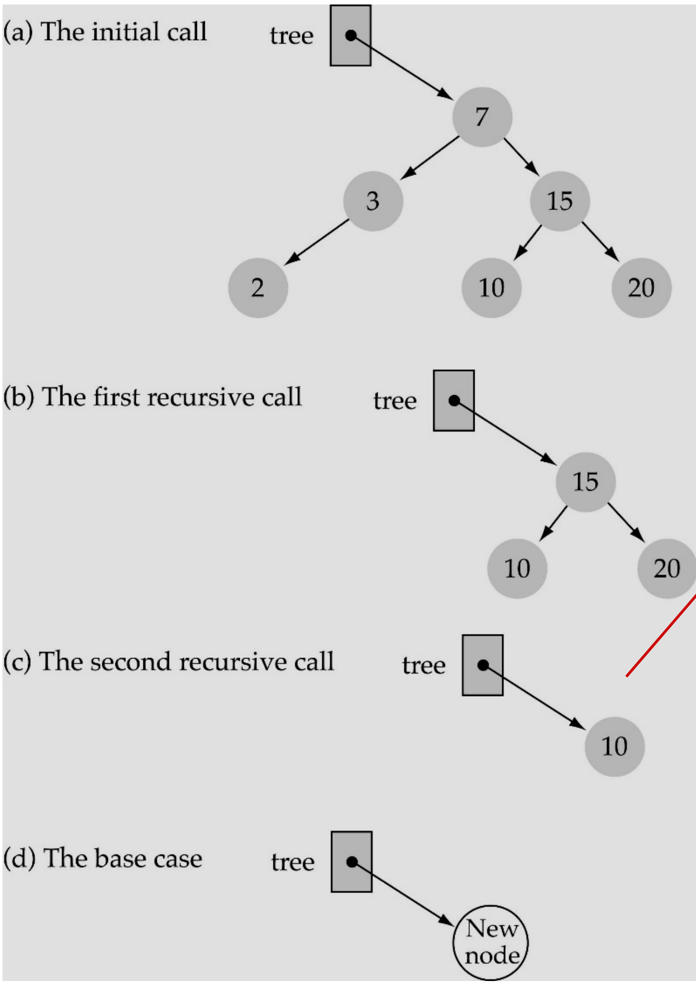
The recursive InsertItem operation

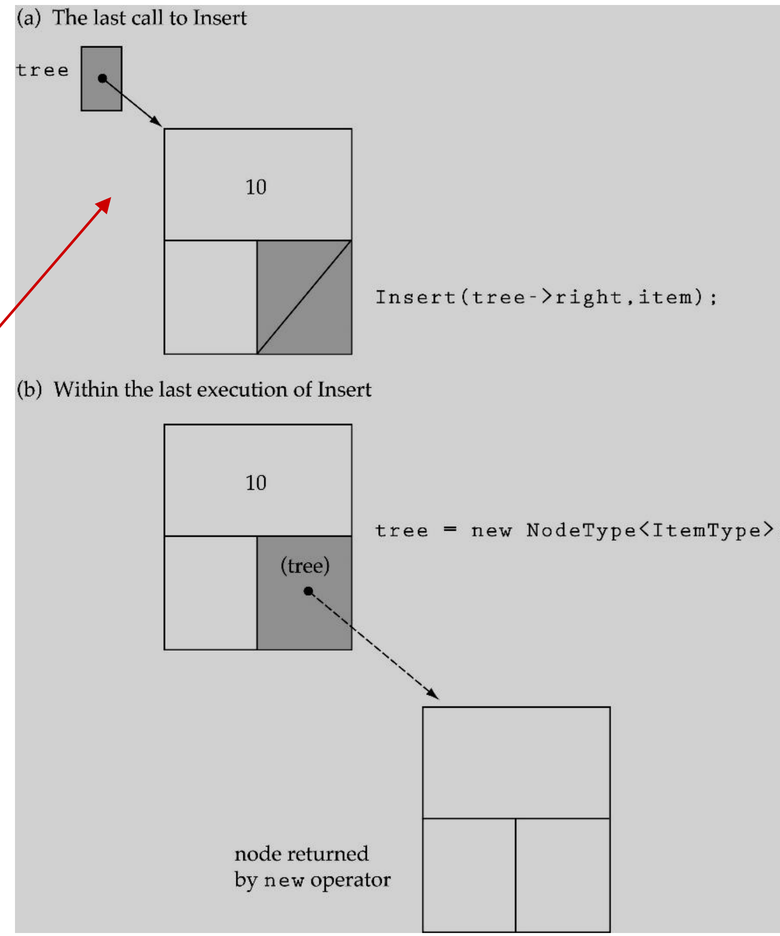
insert 11



The tree parameter
is a pointer within the tree

Insert 11





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);
}
```

referenc
e type

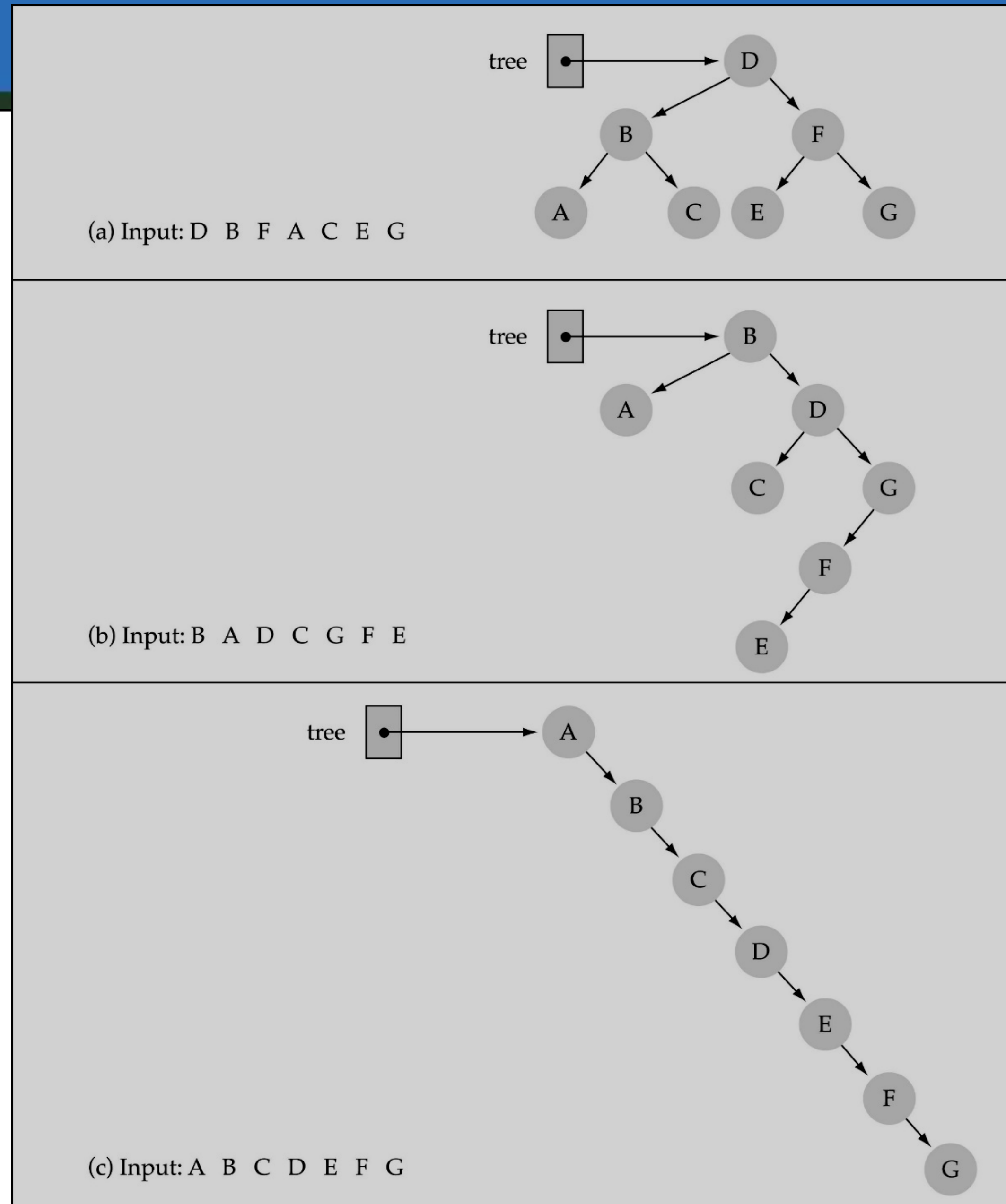
Does the order of inserting elements into a tree matter?

- Yes, certain orders produce very unbalanced trees!!
- Unbalanced trees are not desirable because search time increases!!
- There are advanced tree structures (e.g.,"red-black trees") which guarantee balanced trees

Does the order of inserting elements into a tree matter?

Yes!!!

A random mix of the elements produces a shorter, "bushy" tree



(a) Input: D B F A C E G

(b) Input: B A D C G F E

(c) Input: A B C D E F G

Function DeleteItem

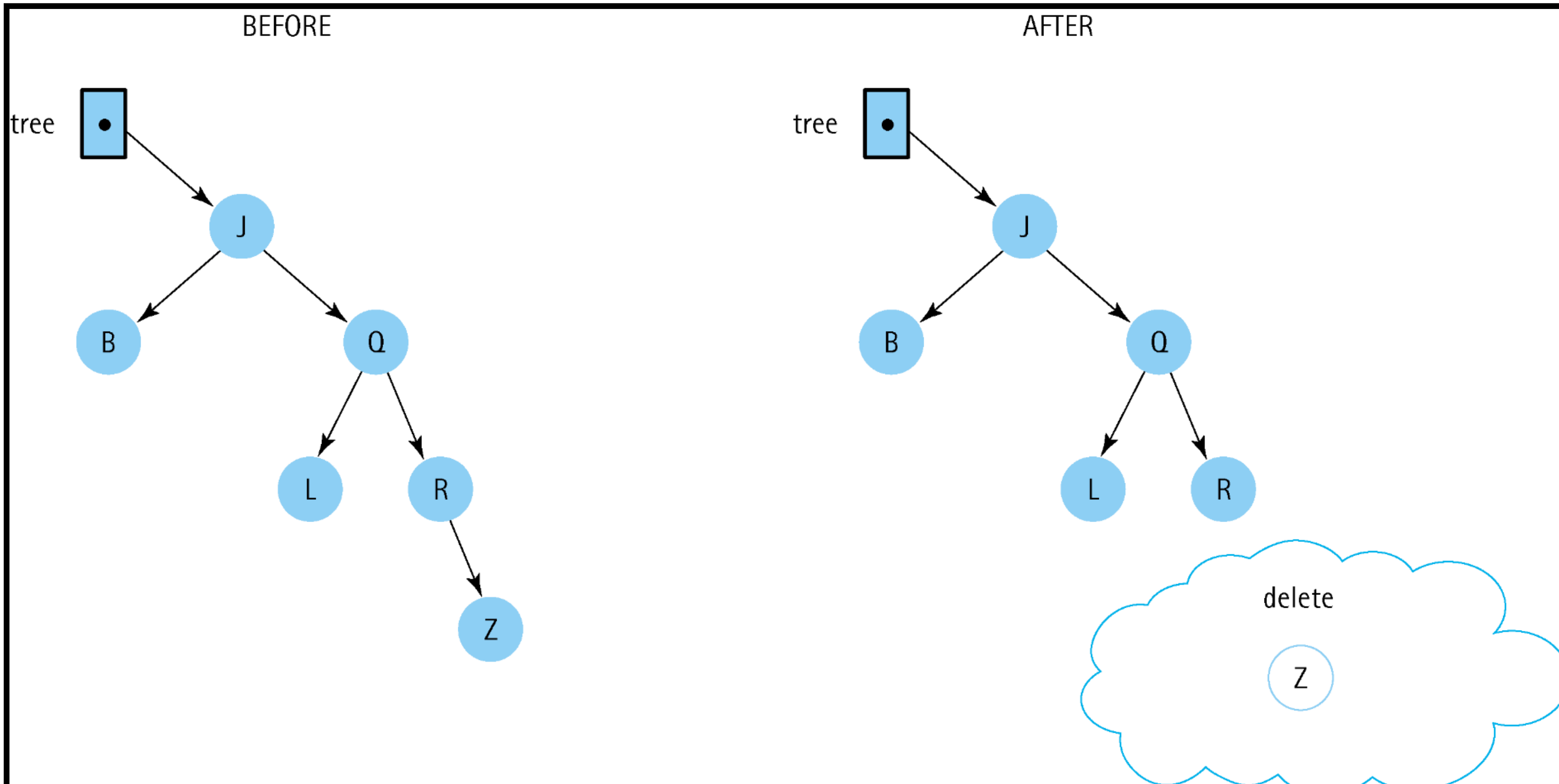
- First, find the item; then, delete it
- Important: binary search tree property must be preserved!!
- We need to consider three different cases:

(1) Deleting a leaf

(2) Deleting a node with only one child

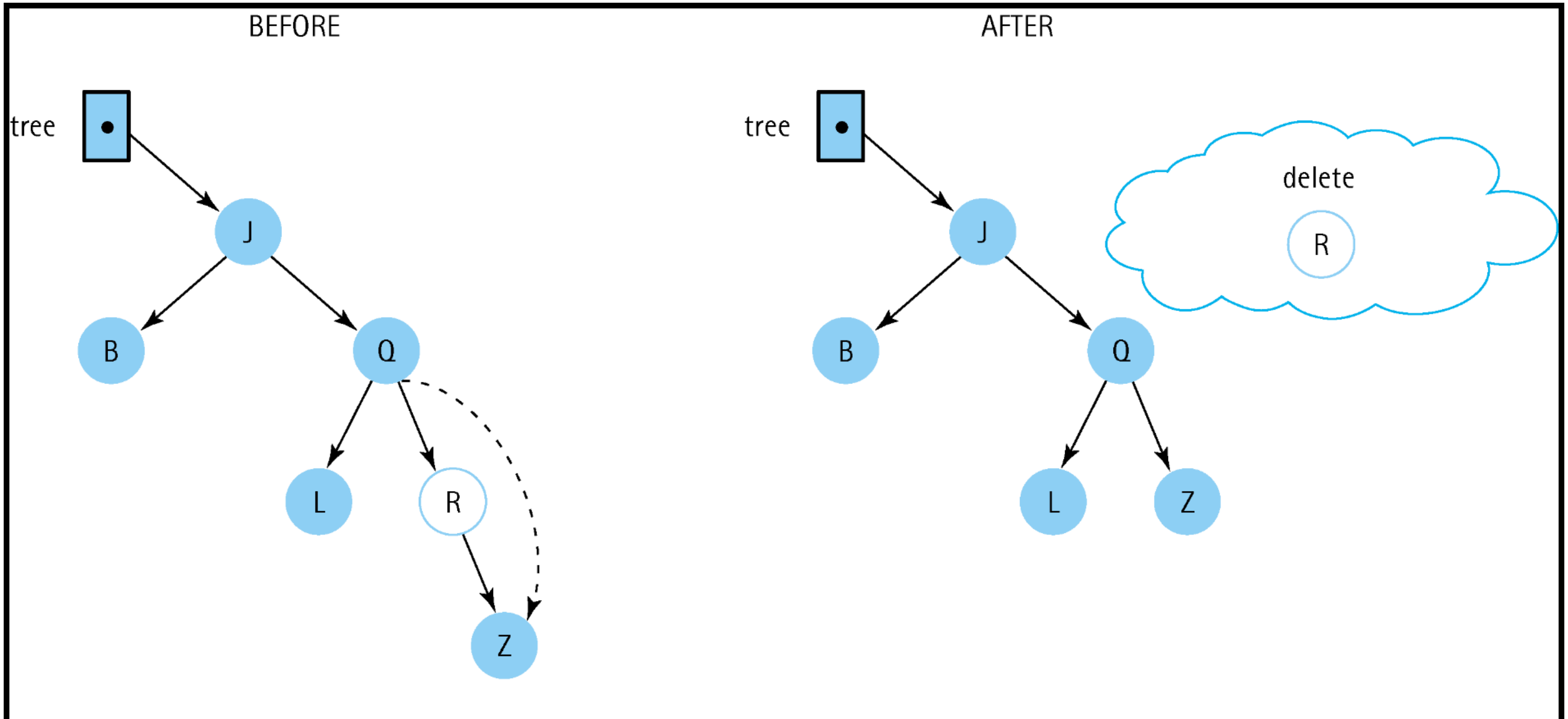
(3) Deleting a node with two children

Deleting a Leaf Node



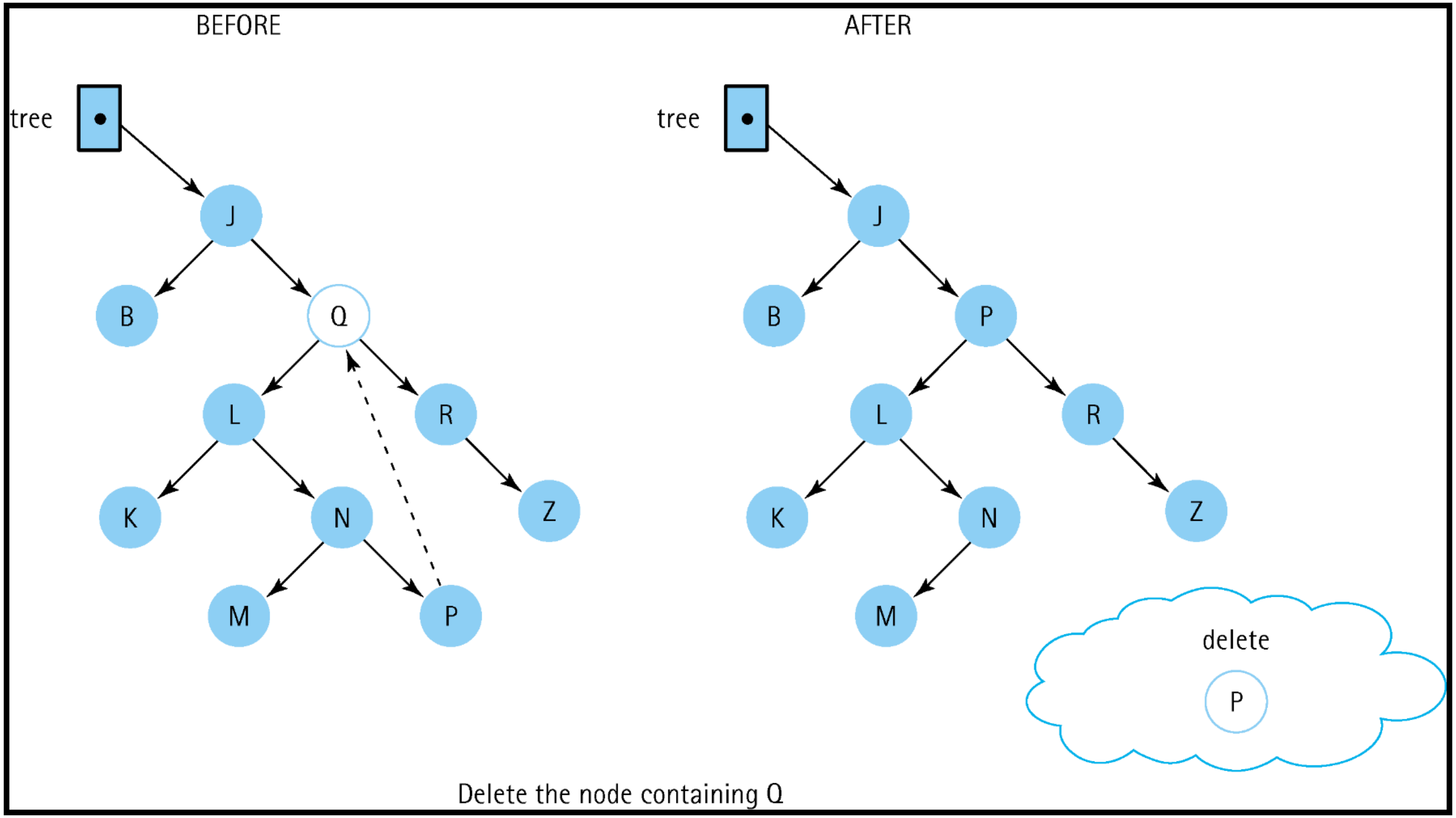
Delete the node containing Z

Deleting a Node with One Child



Delete the node containing R

Deleting a Node with Two Children



Deleting a Node with Two Children (cont'd)

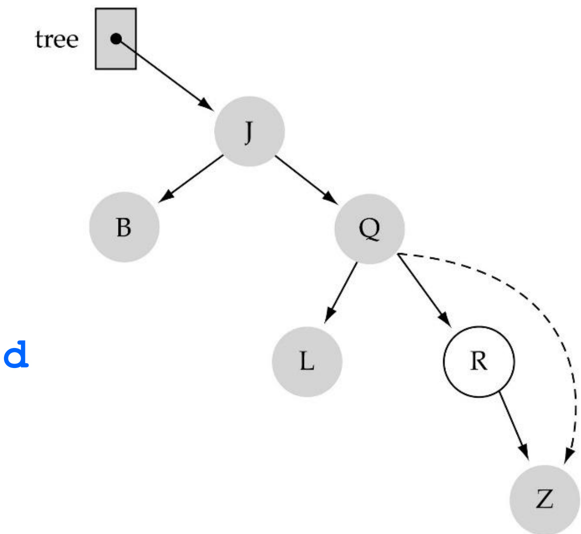
- Find predecessor (it is the rightmost node in the left subtree)
- Replace the data of the node to be deleted with predecessor's data
- Delete predecessor node

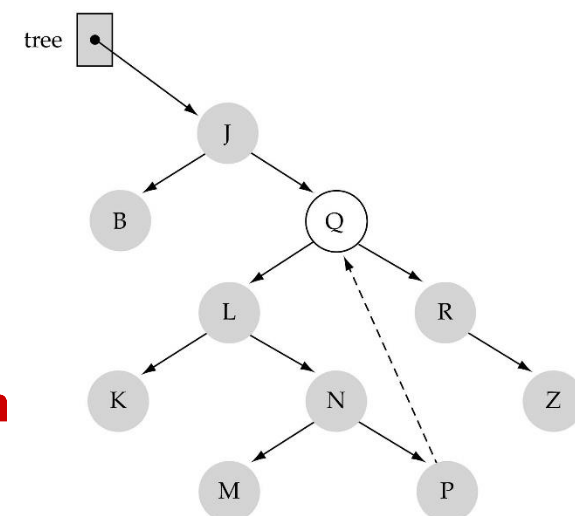
DeleteNode Algorithm

```
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) { //right child
    tree = tree->right;
    delete tempPtr; }        0 or 1 child
  else if (tree->right == NULL){//left child
    tree = tree->left;
    delete tempPtr;}        0 or 1 child
  else{
    GetPredecessor(tree->left, data);
    tree->info = data;
    Delete(tree->left, data);}  2 children
}
```





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?