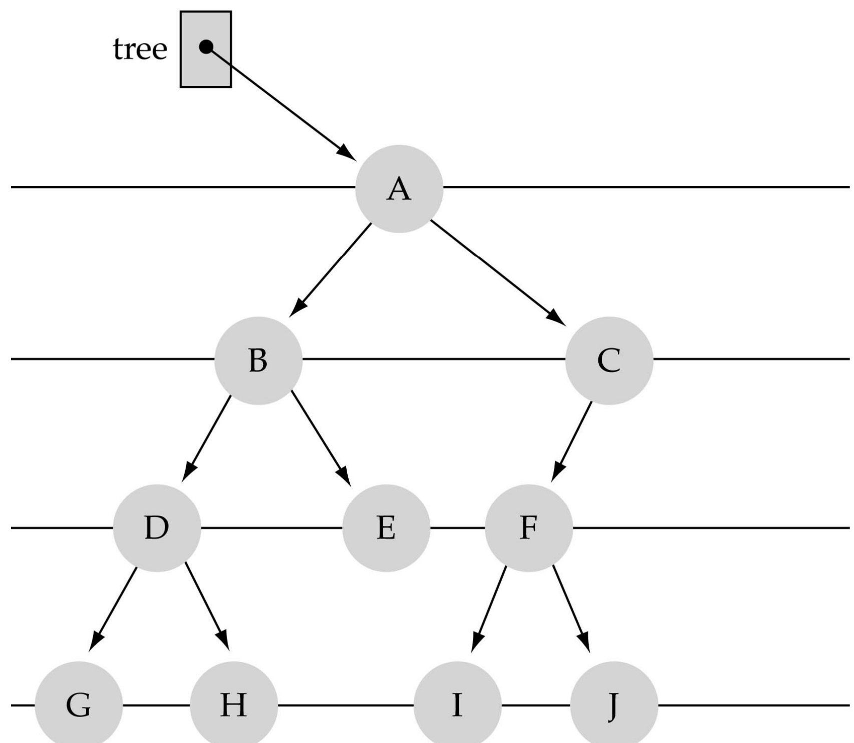
Binary Search Trees

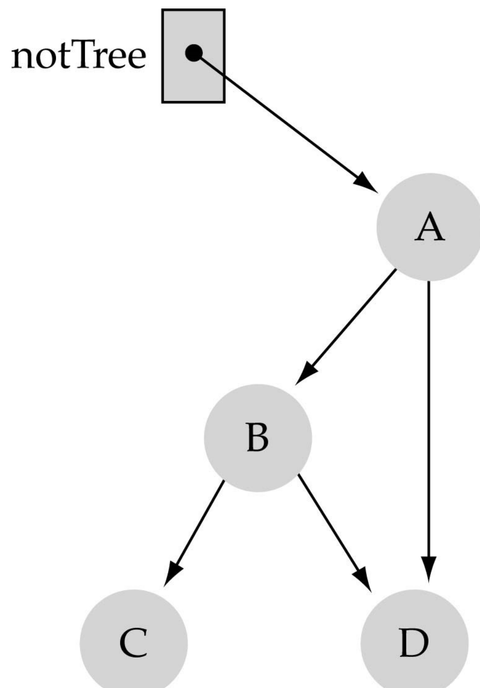
What is a binary tree?

- *Property 1*: **each node can have up to two successor nodes.**



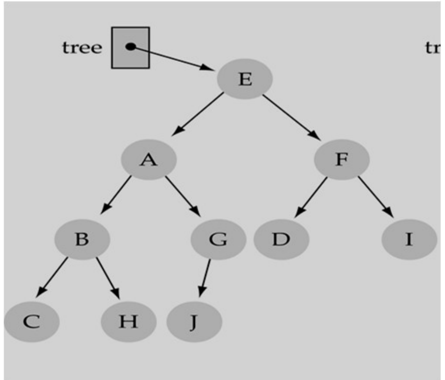
What is a binary tree? (cont.)

- **Property 2:** **a unique path exists from the root to every other node**



Some terminology

- **The successor nodes of a node are called its *children***
- **The predecessor node of a node is called its *parent***
- **The "beginning" node is called the *root* (has no parent)**
- **A node without *children* is called a *leaf***



Some terminology (cont'd)

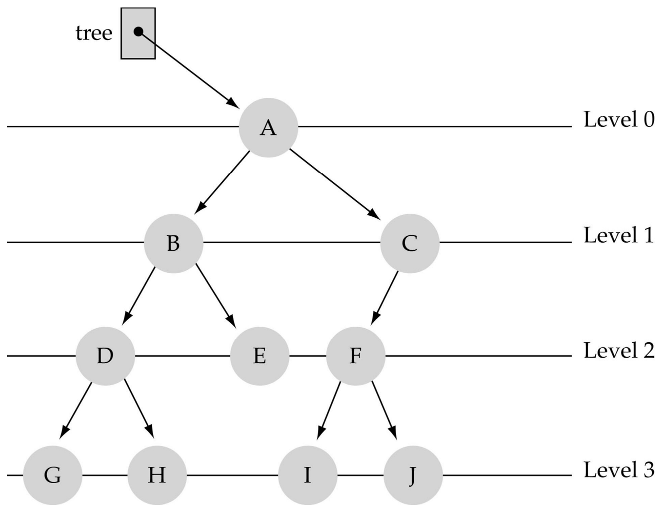
- **Nodes are organized in levels (indexed from 0).**

- **Level (or depth) of a node: number of edges in the path from the root to that node.**

- **Height of a tree h: #levels = L (*Warning:* some books define h as #levels-1).**

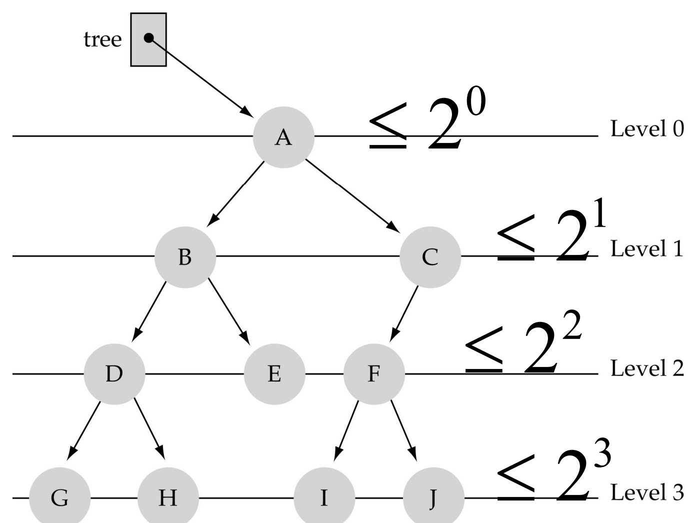
- **Full tree: every node has exactly two children *and* all the leaves are on the same level.**

not full!



What is the <u>max</u> #nodes at some level l?

The <u>max</u> #nodes at level l is 2^l where $l=0,1,2,\ \ldots,L-1$



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

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

l=0 l=1 l=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 <u>full</u> tree with N nodes?

$$2^h - 1 = N$$

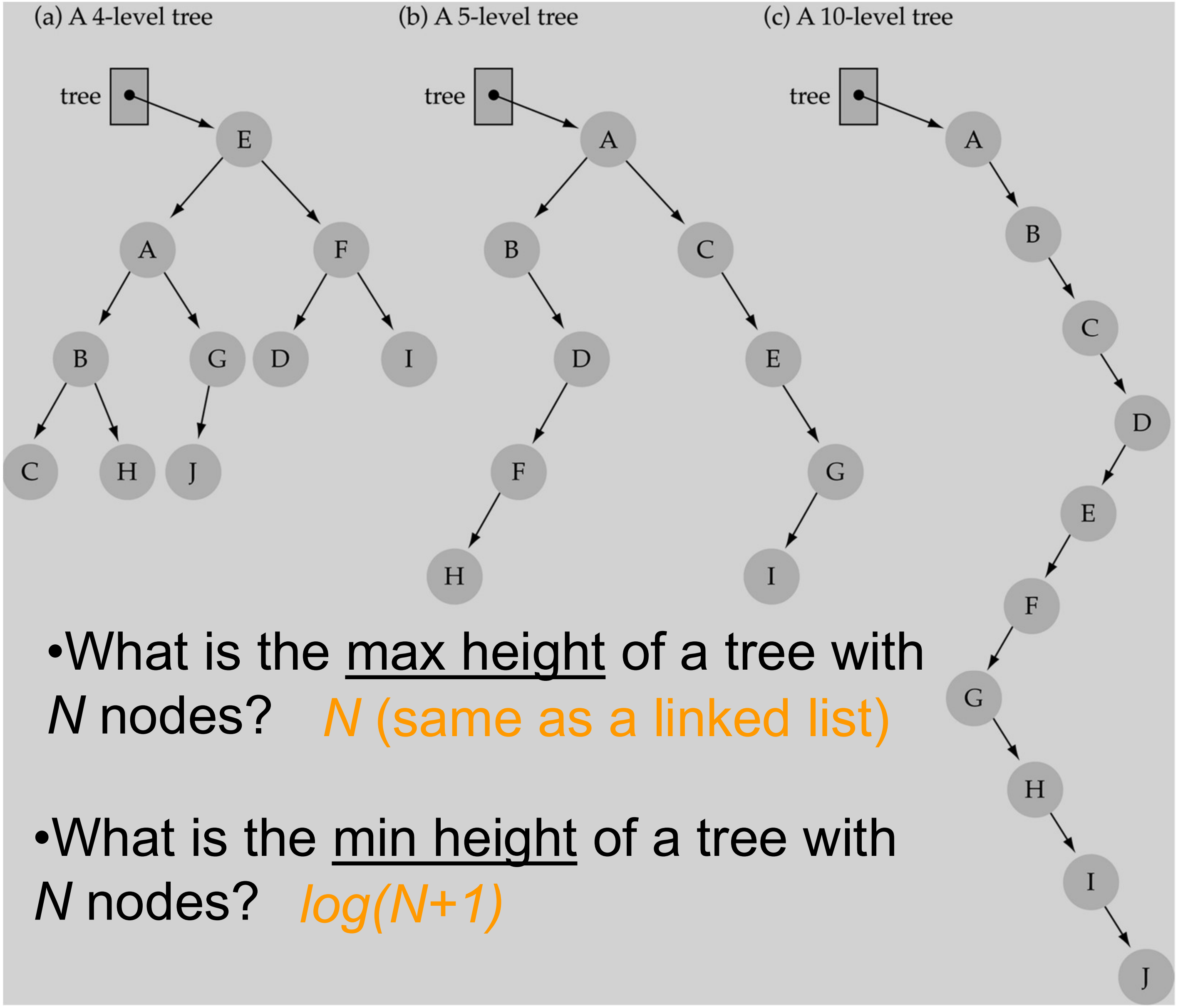
$$\Rightarrow 2^h = N + 1$$

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

Why is h important?

- **Tree operations (e.g., insert, delete, retrieve etc.) are typically expressed in terms of h.**

- **So, h determines the complexity or running time**



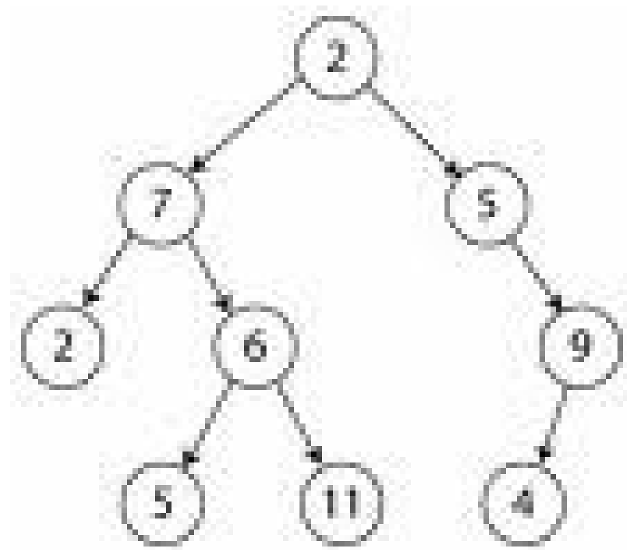
- What is the <u>max height</u> of a tree with *N* nodes? *N* (same as a linked list)

- What is the <u>min height</u> of a tree with *N* nodes? *log(N+1)*

How to search a binary tree?

(1) Start at the root

(2) Search the tree level by level, until you find the element you are searching for or you reach a leaf.

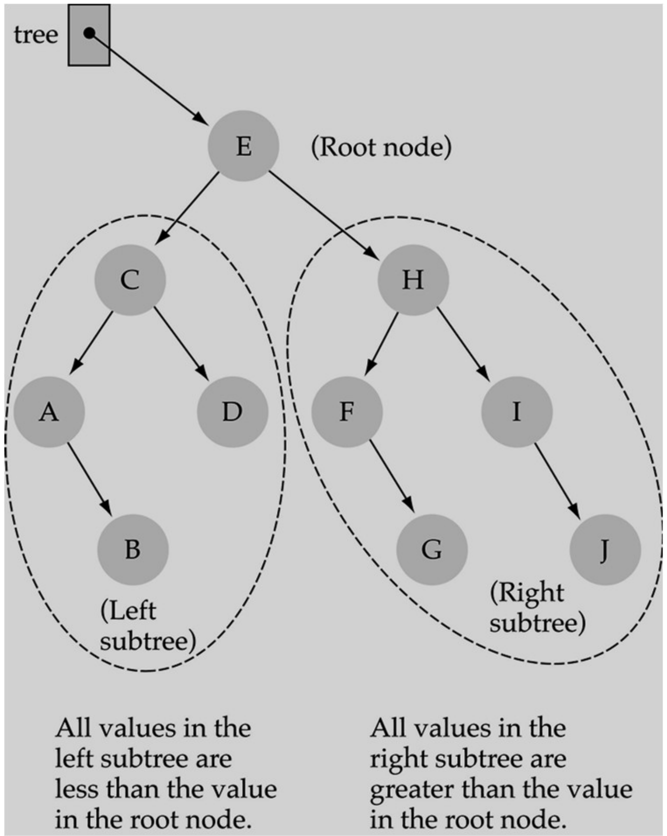


Is this better than searching a linked list?

No → O(N)

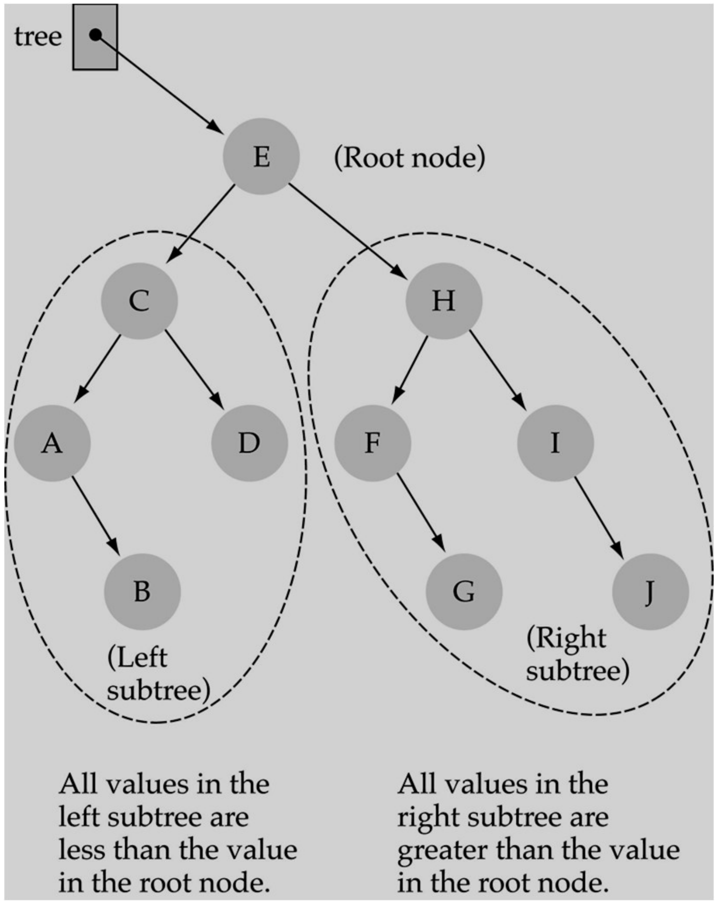
Binary Search Trees (BSTs)

- **Binary Search Tree Property:**
  The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



Binary **Search** Trees (BSTs)

In a BST, the value stored at the root of a subtree is *greater* than any value in its left subtree and *less* than any value in



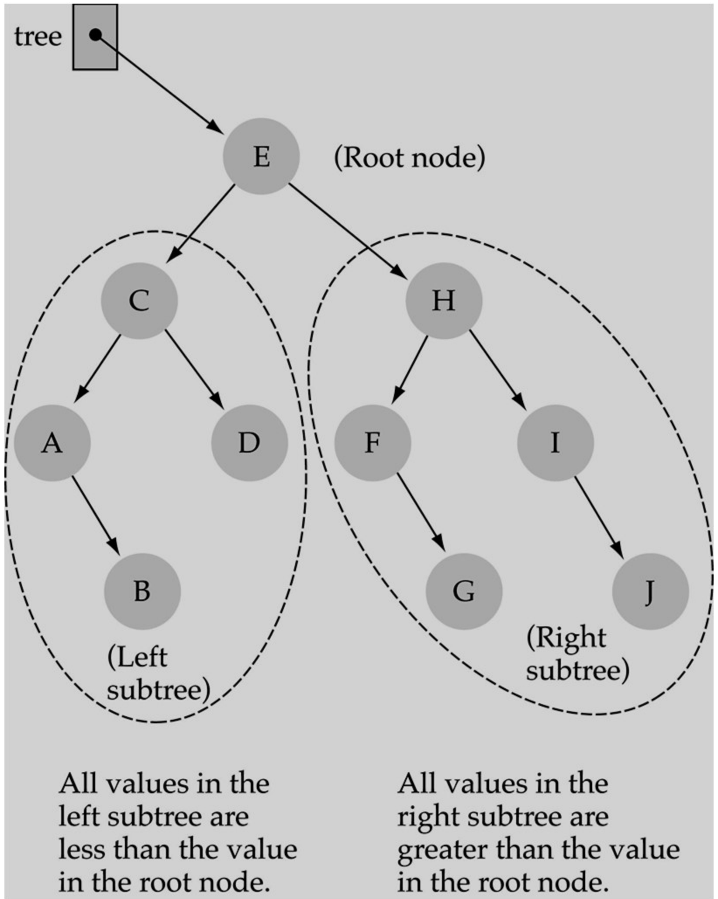
Binary **Search** Trees (BSTs)

Where is the smallest element?

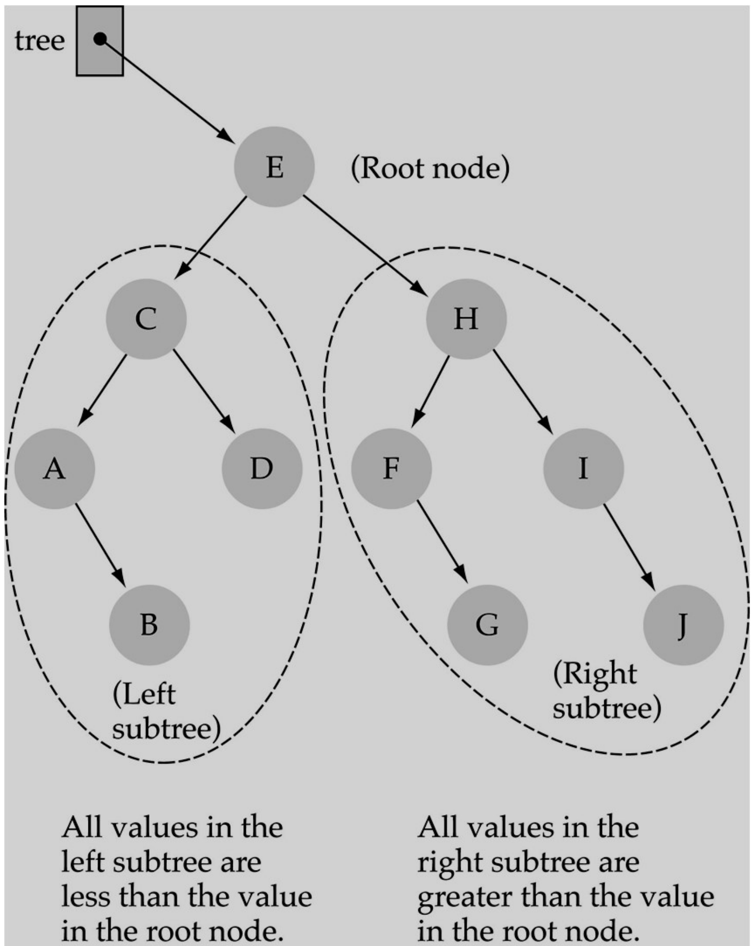
Ans: leftmost element

Where is the largest element?

Ans: rightmost element



How to search a binary search tree?

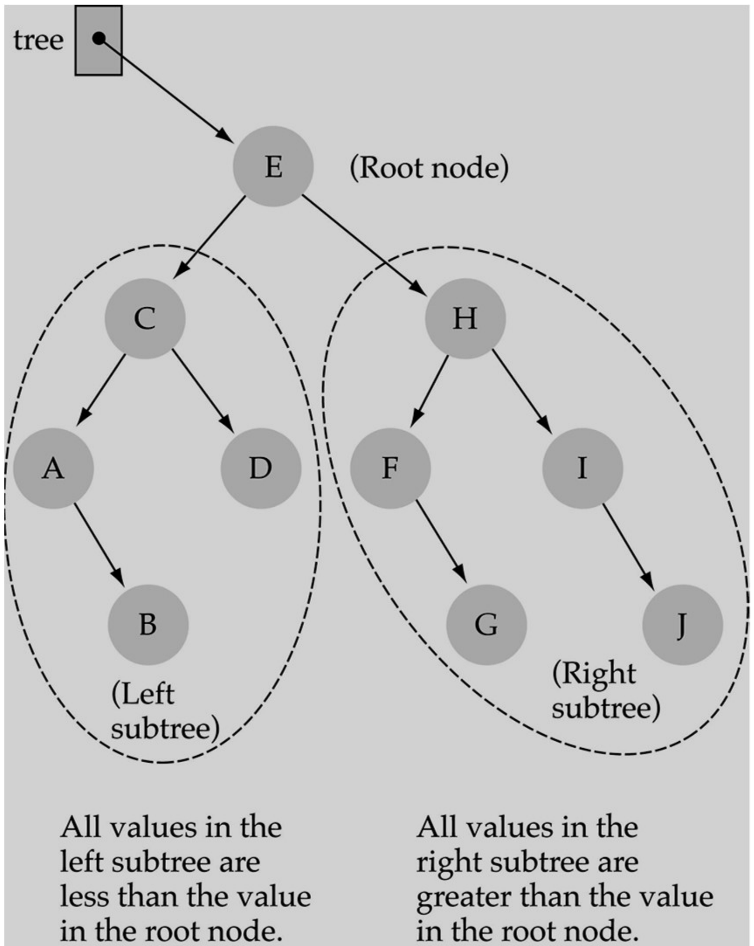


(1) Start at the root

(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*

How to search a binary search tree?

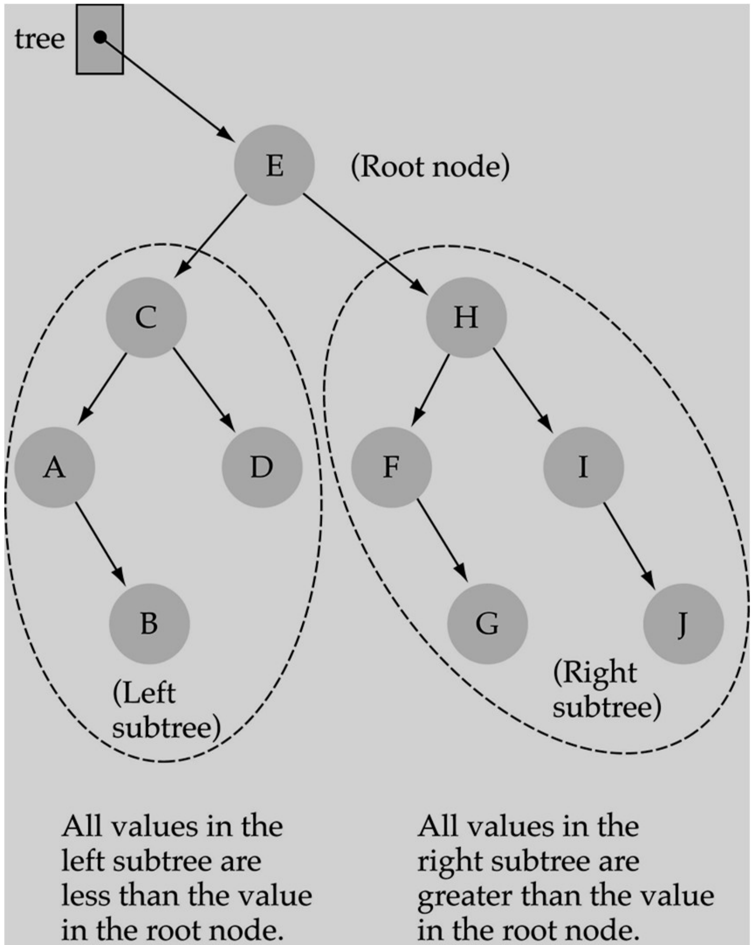


4) If it is less than the value stored at the root, then search the left subtree

(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

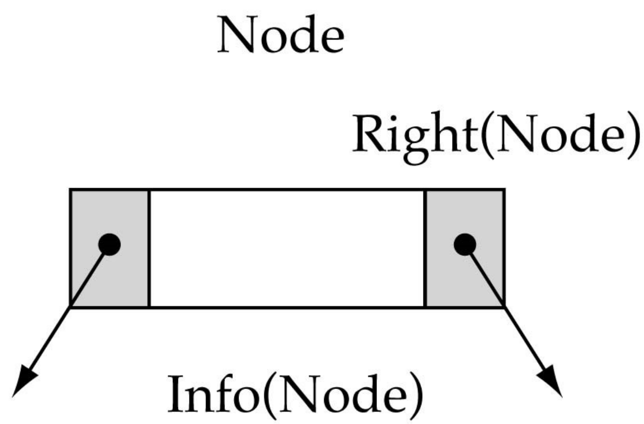
How to search a binary search tree?



this better than searching a linked list?

Yes !! ---> O(logN)

Tree node structure




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

Function NumberOfNodes (cont.)

```
template<class ItemType>
int TreeType<ItemType>::NumberOfNodes() const
{
 return CountNodes(root);
}

template<class ItemType>
int CountNodes(TreeNode<ItemType>* tree)
{
 if (tree == NULL)
   return 0;
 else
   return CountNodes(tree->left) + CountNodes(tree->right)
    + 1;
}
```

Function NumberOfNodes

- **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?**

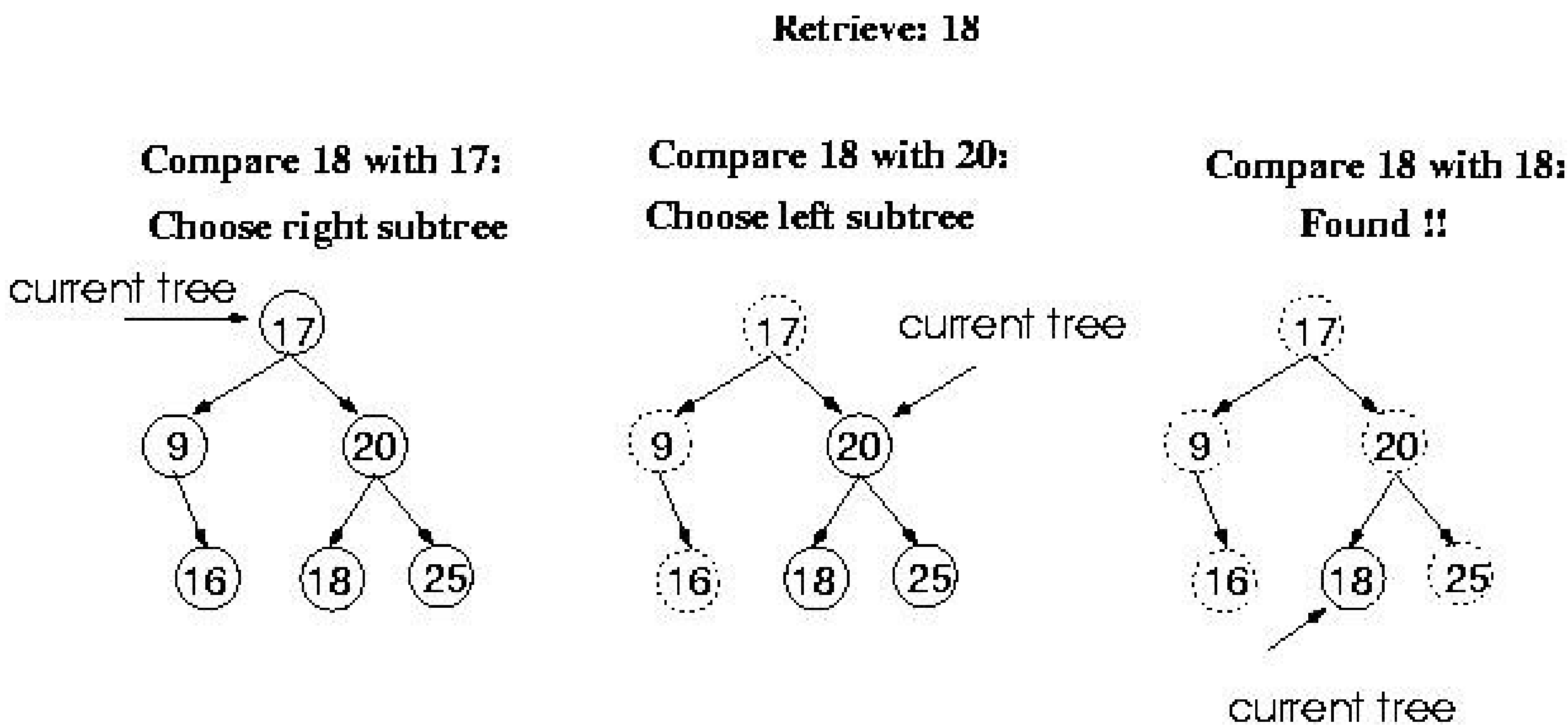
  CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1

Function RetrieveItem (cont.)

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

```
template<class ItemType>
void Retrieve(TreeNode<ItemType>* tree, ItemType& item, bool&
   found)
{
 if (tree == NULL) // base case 2
   found = false;
 else if(item < tree->info)
   Retrieve(tree->left, item, found);
 else if(item > tree->info)
   Retrieve(tree->right, item, found);
 else {  // base case 1
   item = tree->info;
   found = true;
 }
```

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

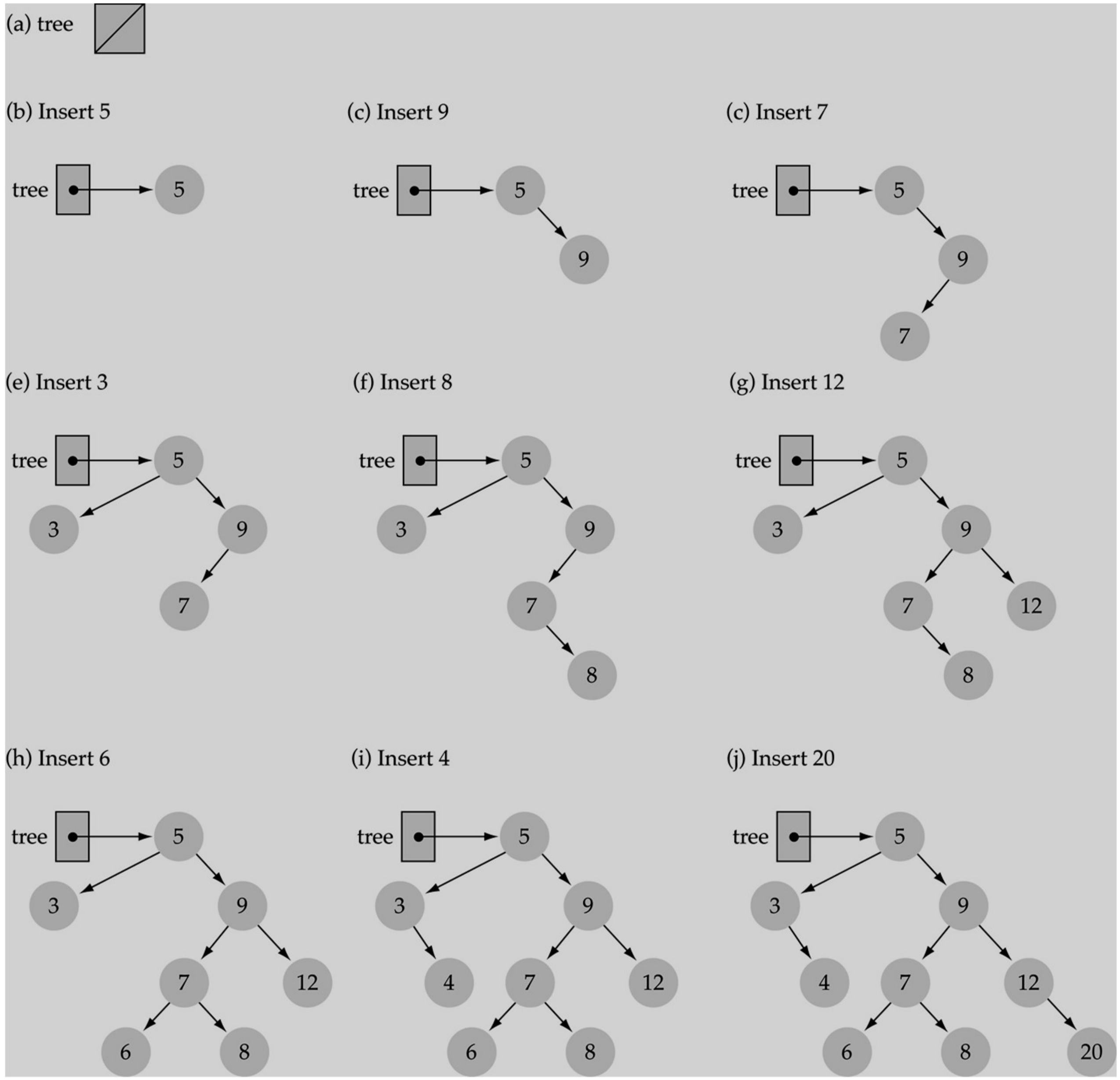
Function InsertItem (cont.)

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

```
template<class ItemType>
void Insert(TreeNode<ItemType>*& tree, ItemType item)
{
 if(tree == NULL) {  // base case
   tree = new TreeNode<ItemType>;
   tree->right = NULL;
   tree->left = NULL;
   tree->info = item;
 }
 else if(item < tree->info)
   Insert(tree->left, item);
 else
   Insert(tree->right, item);
}
```

Function InsertItem

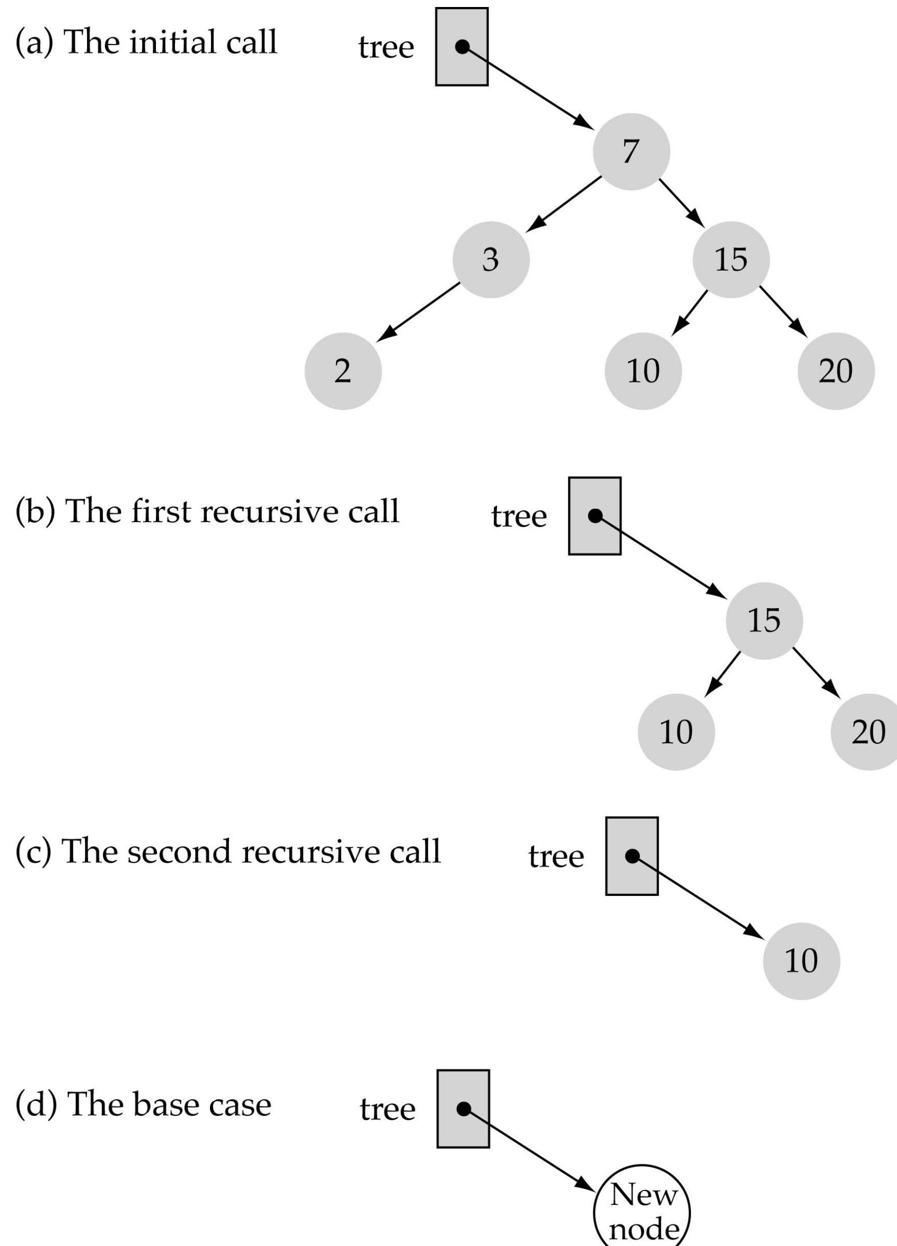
- **Use the binary search tree property to insert the new item at the correct place**



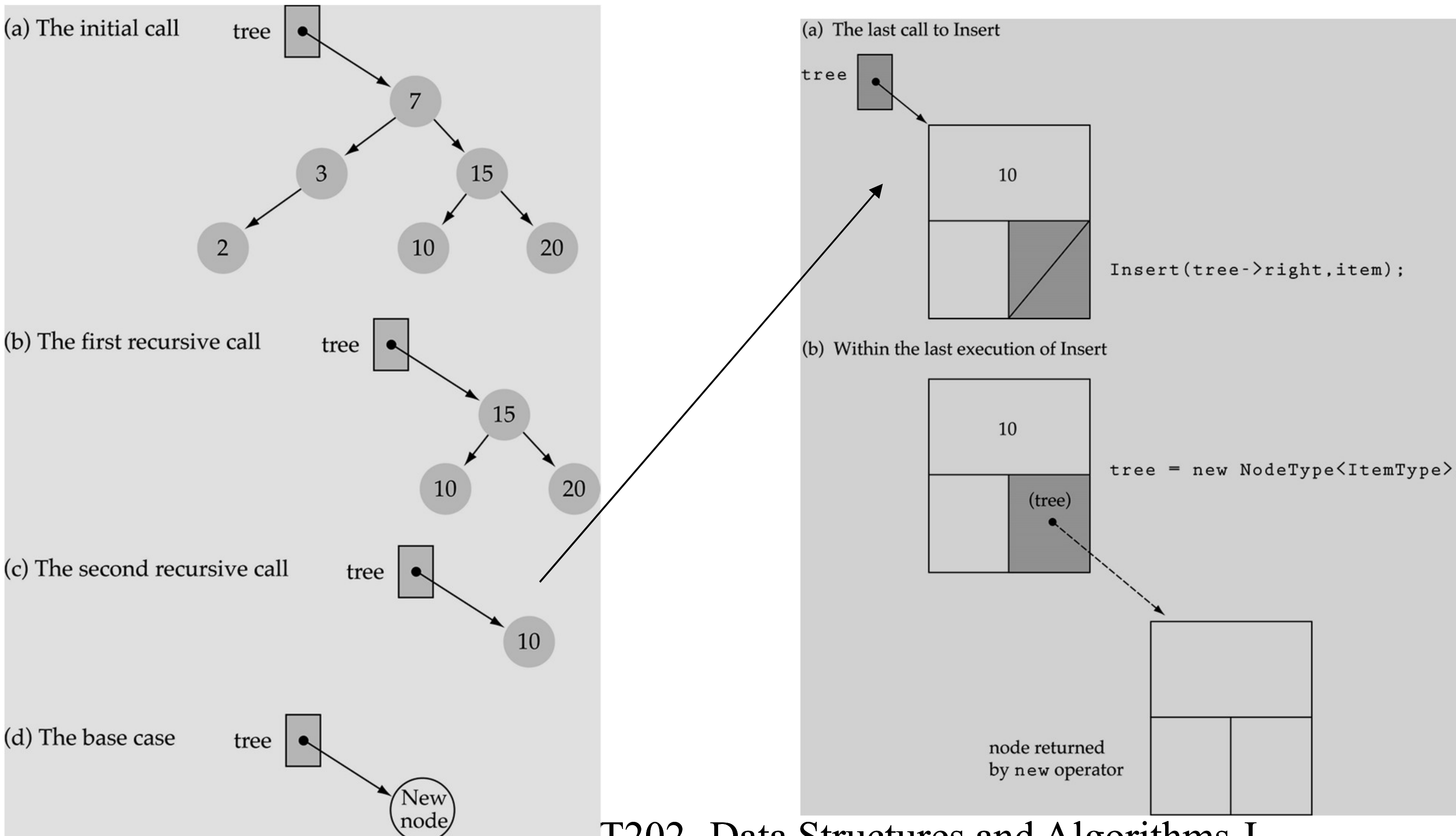
Function InsertItem (cont.)

- Implementing insertion resursively

e.g., insert 11



Function InsertItem (cont.)



Function InsertItem (cont.)

- **What is the size of the problem?**

  Number of nodes in the tree we are examining

- **What is the base case(s)?**

  The tree is empty

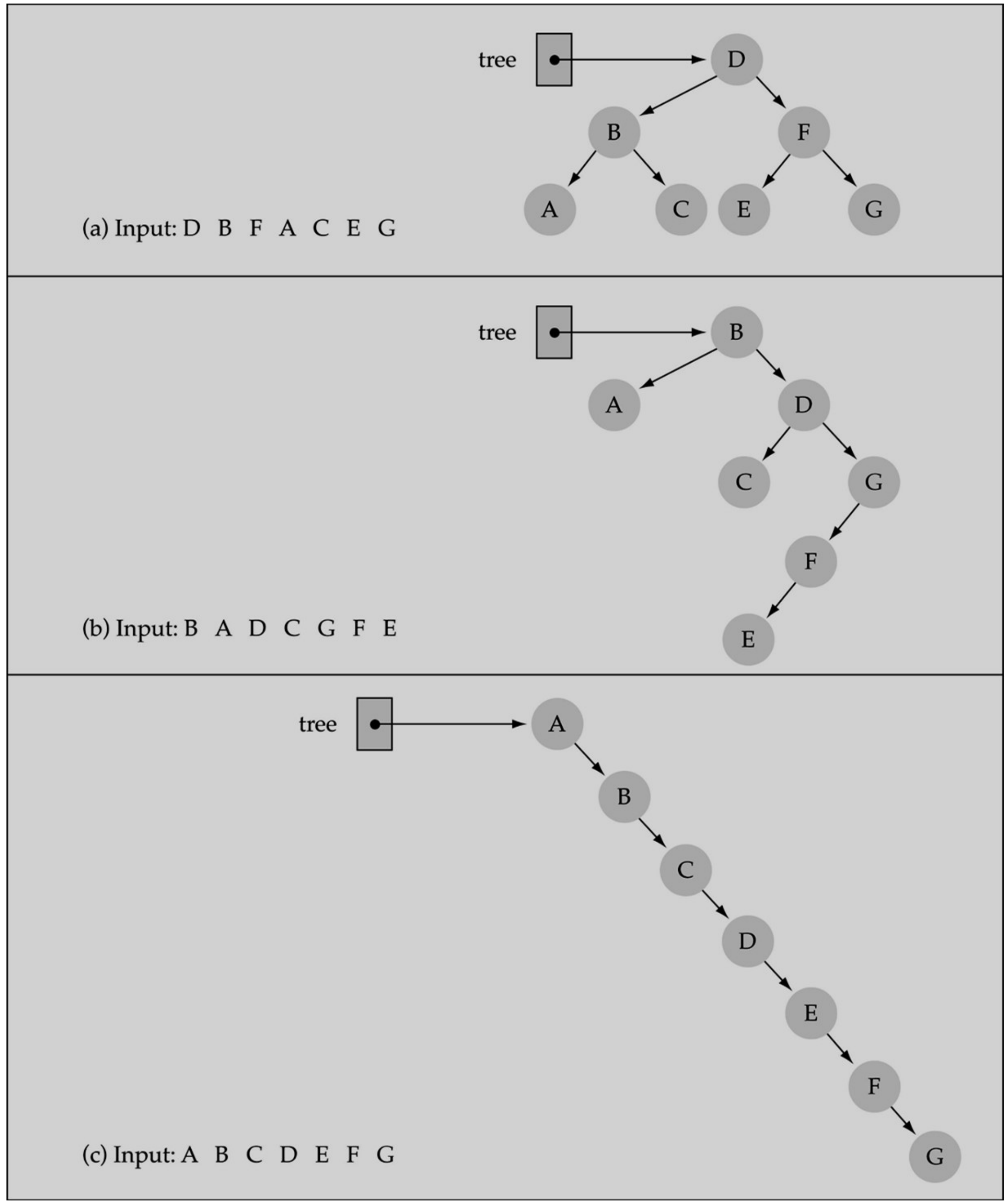
- **What is the general case?**

  Choose the left or right subtree

Does the order of inserting elements into a tree matter?

- **Yes, certain orders might produce very unbalanced trees!**

Does the order of inserting elements into a tree matter? (cont.)



Does the order of inserting elements into a tree matter? (cont'd)

- **Unbalanced trees are not desirable because search time increases!**
- **Advanced tree structures, such as red-black trees, guarantee balanced trees.**

Function DeleteItem

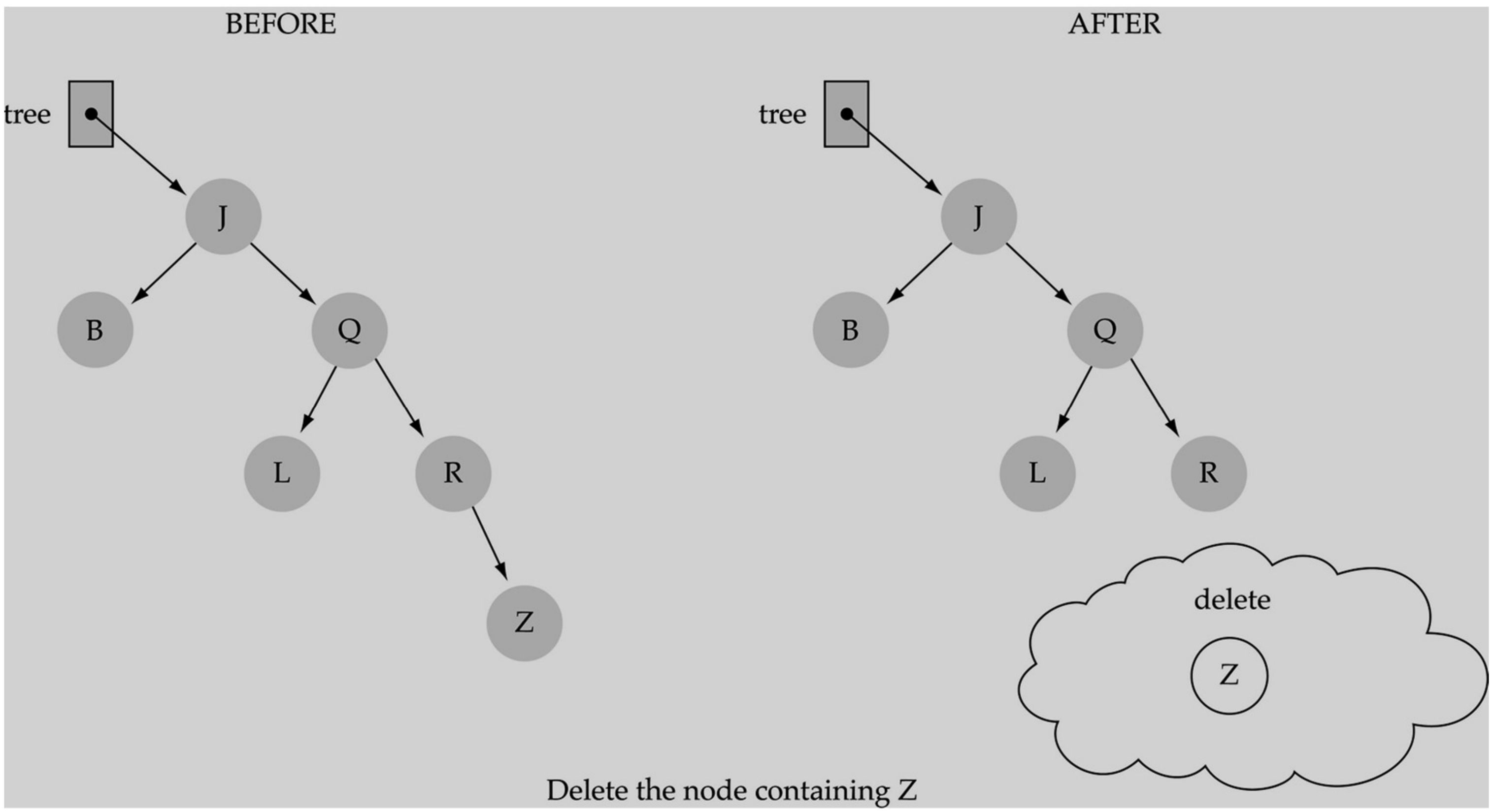
- **First, find the item; then, delete it**
- **Binary search tree property must be preserved!!**
- **We need to consider three different cases:**

  (1) Deleting a leaf

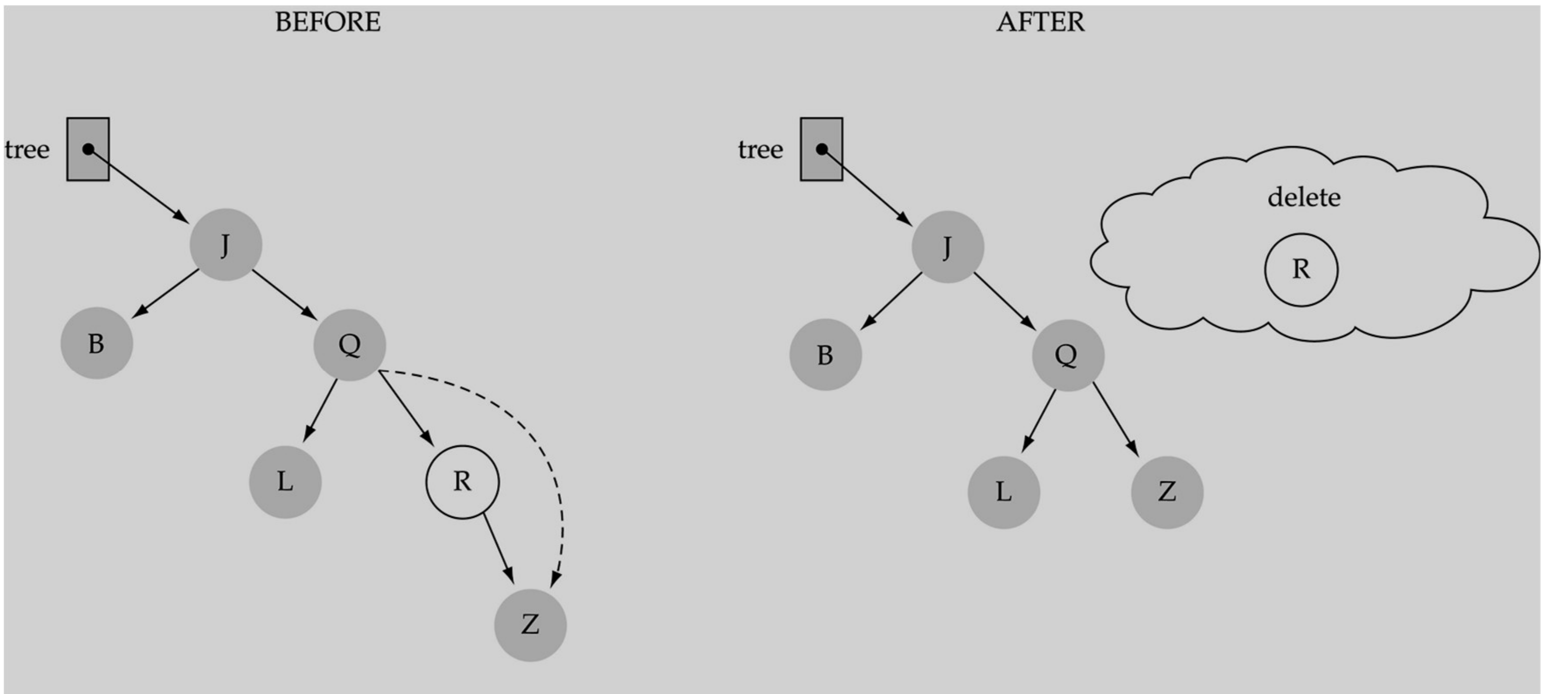
  (2) Deleting a node with only one child

  (3) Deleting a node with two children

(1) Deleting a leaf

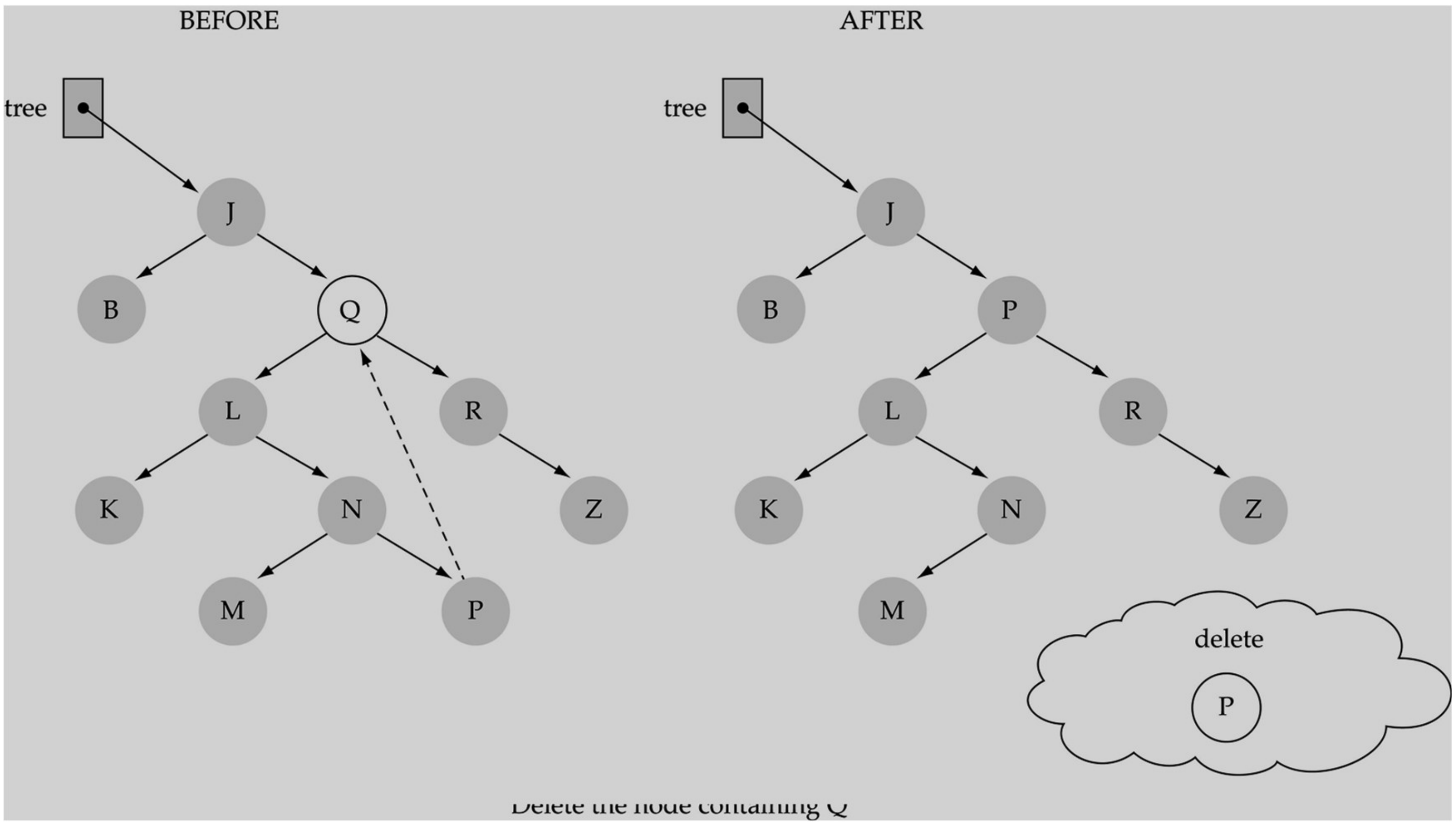


(2) Deleting a node with only one child



Delete the node containing R

(3) Deleting a node with two children



(3) Deleting a node with two children (cont.)

- **Find predecessor (i.e., rightmost node in the left subtree)**
- **Replace the data of the node to be deleted with predecessor's data**
- **Delete predecessor node**

Function DeleteItem (cont.)

```
template<class ItemType>
void TreeType<ItmeType>::DeleteItem(ItemType item)
{
 Delete(root, item);
}
```

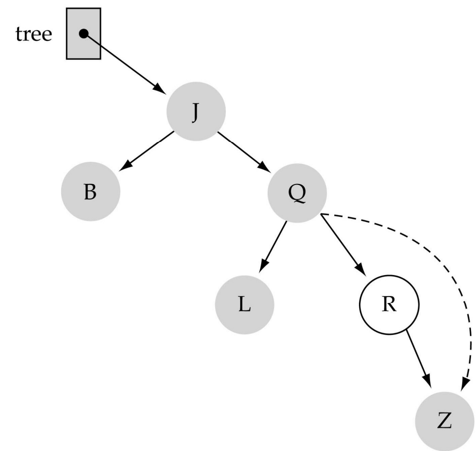
```
template<class ItemType>
void Delete(TreeNode<ItemType>*& tree, ItemType item)
{
 if(item < tree->info)
   Delete(tree->left, item);
 else if(item > tree->info)
   Delete(tree->right, item);
 else
   DeleteNode(tree);
}
```

Function DeleteItem (cont.)

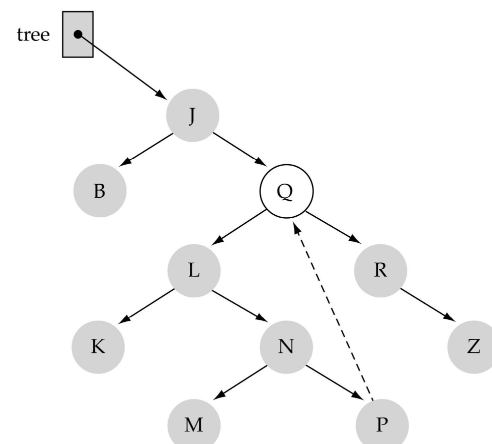
```
template <class ItemType>
void DeleteNode(TreeNode<ItemType>*& tree)
{
 ItemType item;
 TreeNode<ItemType>* tempPtr;

 tempPtr = tree;
 if(tree->left == NULL) { // right child
   tree = tree->right;
   delete tempPtr;
 }
 else if(tree->right == NULL) { // left child
   tree = tree->left;
   delete tempPtr;
 }
 else {
   GetPredecessor(tree->left, item);
   tree->info = item;
   Delete(tree->left, item);
 }
}
```

0 children or 1 child



0 children or 1 child



2 children

Function DeleteItem (cont.)

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

Function DeleteItem (cont.)

```
template<class ItemType>
void TreeType<ItmeType>::DeleteItem(ItemType item)
{
 Delete(root, item);
}
```

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

Tree Traversals

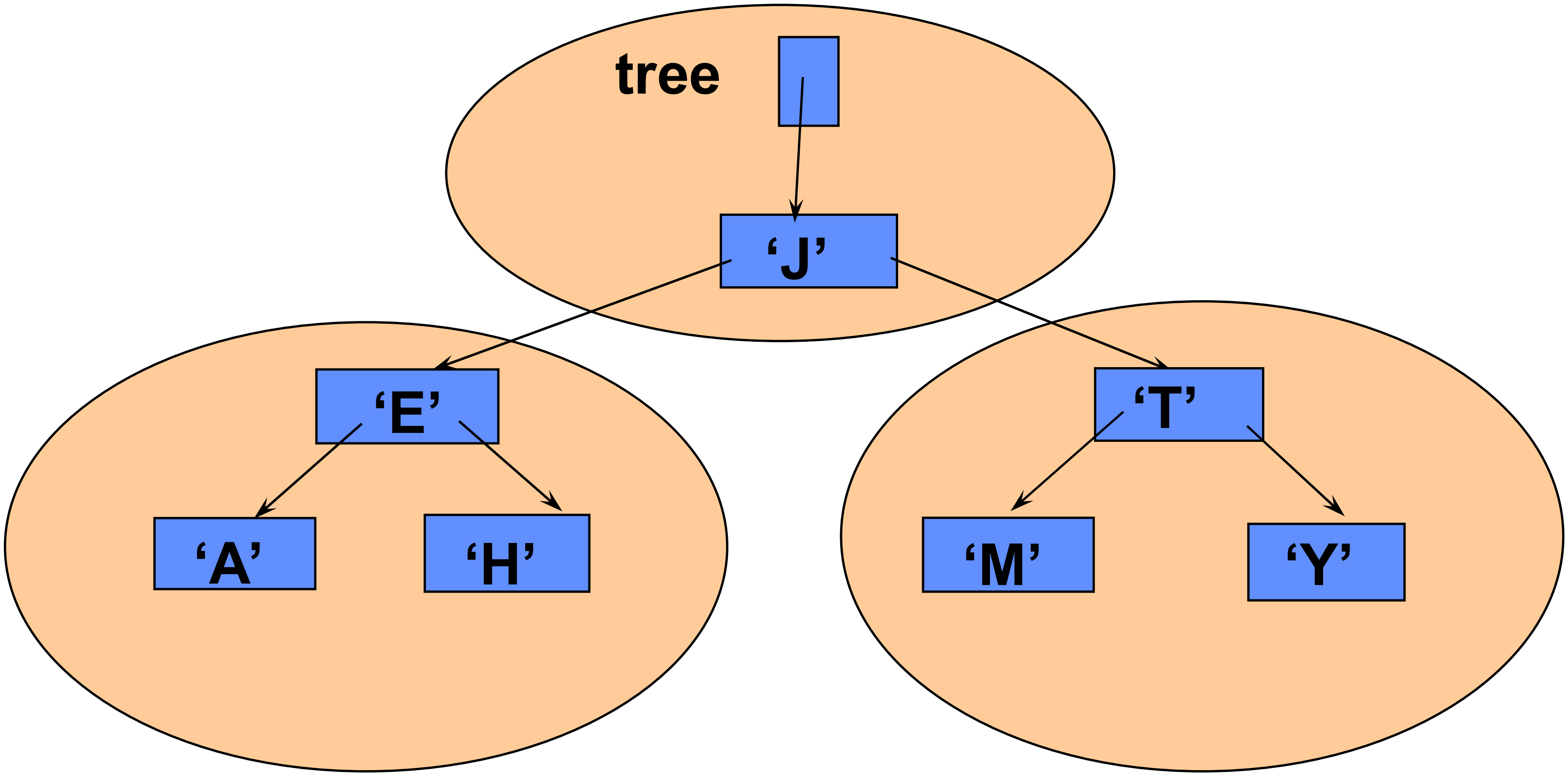
There are mainly three ways to traverse a tree:

Inorder Traversal

Postorder Traversal

Preorder Traversal

Inorder Traversal:



Visit left subtree first

Visit right subtree last

Inorder Traversal

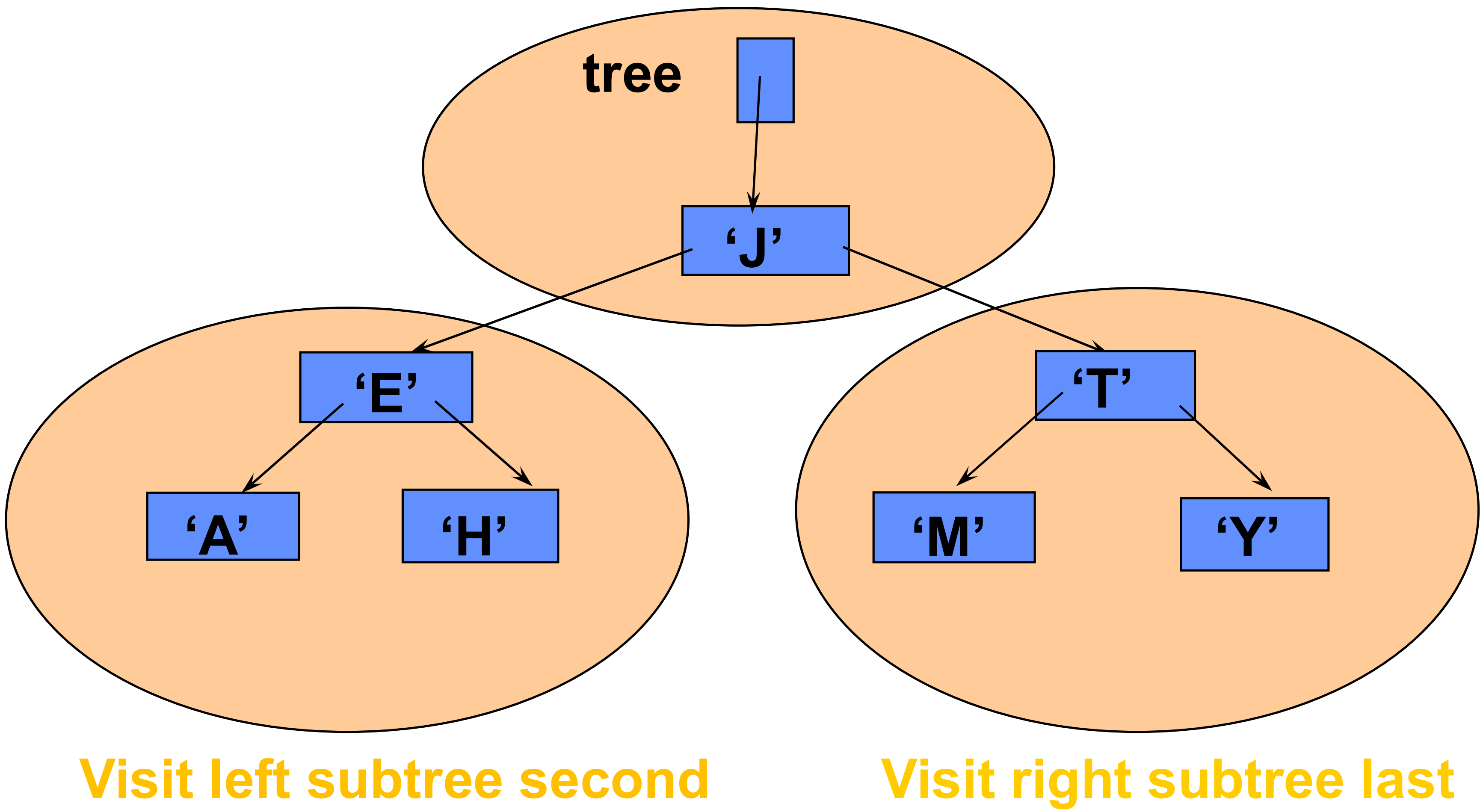
- **Visit the nodes in the left subtree, then visit the root of the tree, then visit the nodes in the right subtree**

```
Inorder(tree)

If tree is not NULL
  Inorder(Left(tree))
  Visit Info(tree)
  Inorder(Right(tree))
```

Warning: "visit" implies <u>do</u> something with the value at the node (e.g., print, save, update etc.).

Preorder Traversal:

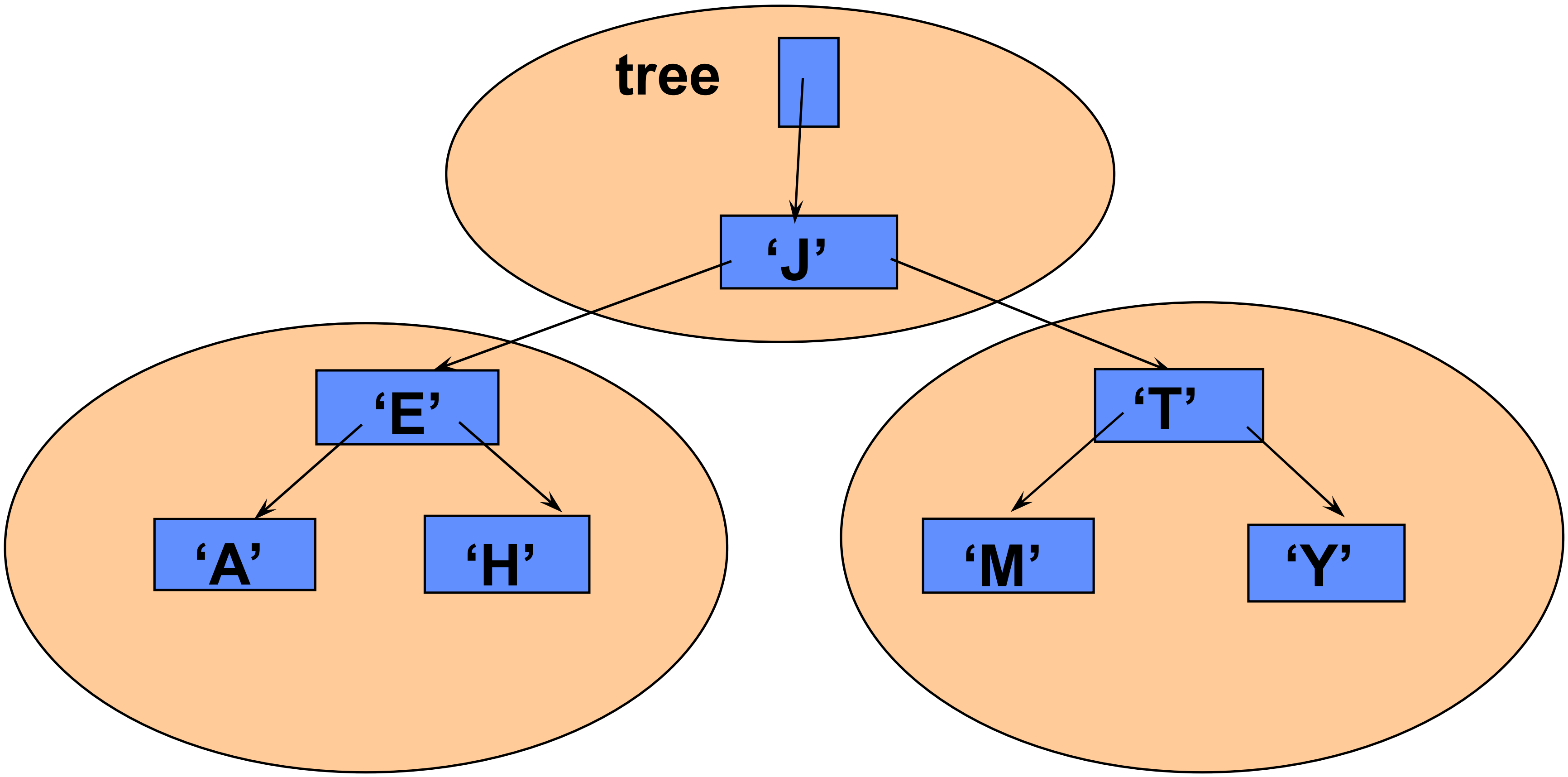


Preorder Traversal

- **Visit the root of the tree first, then visit the nodes in the left subtree, then visit the nodes in the right subtree**

```
Preorder(tree)
If tree is not NULL
  Visit Info(tree)
  Preorder(Left(tree))
  Preorder(Right(tree))
```

Postorder



Visit left subtree first

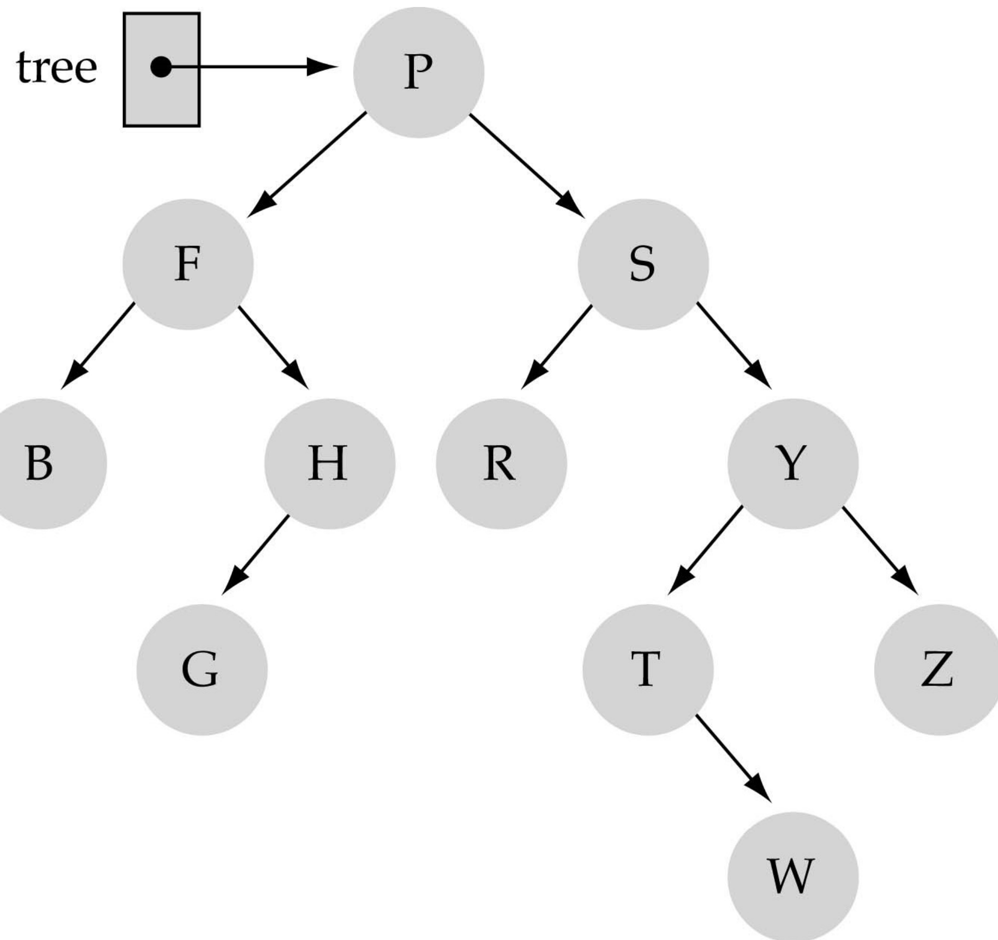
Visit right subtree second

Postorder Traversal

- **Visit the nodes in the left subtree first, then visit the nodes in the right subtree, then visit the root of the tree**

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

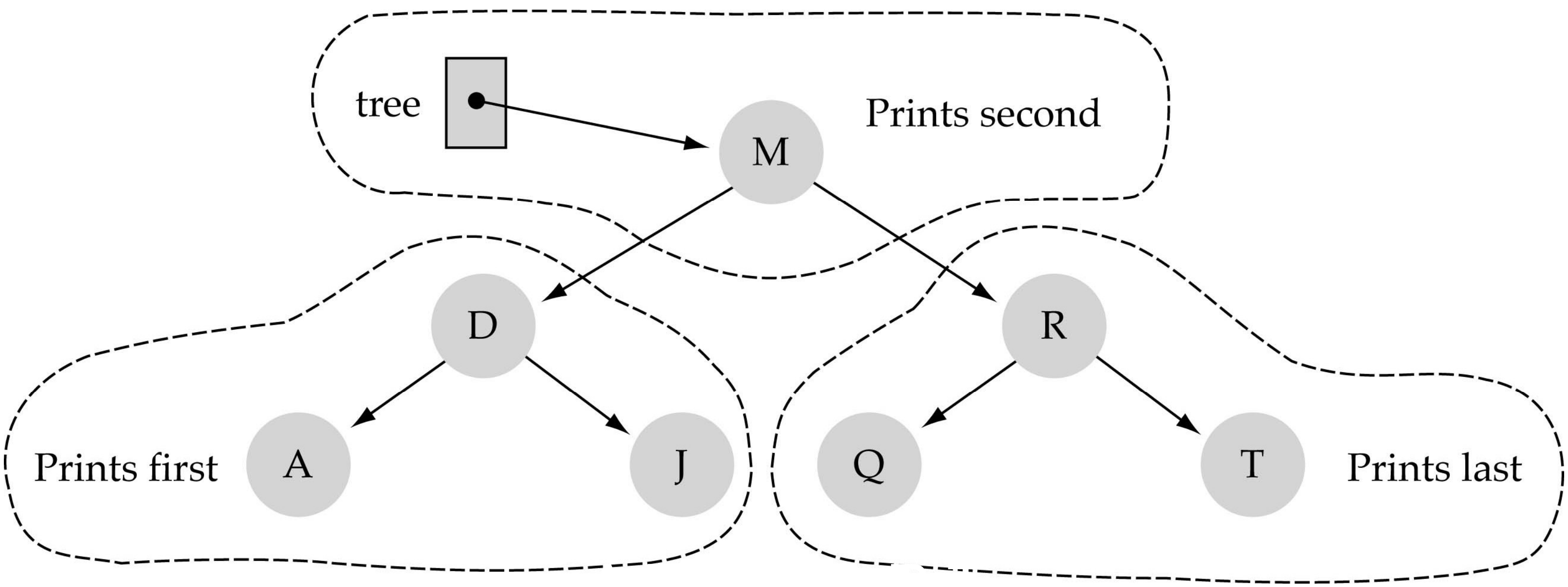
Tree Traversals: another example



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

Function PrintTree

- We use "inorder" to print out the node values.
- Keys will be printed out in sorted order.
- <u>Hint:</u> binary search could be used for sorting!

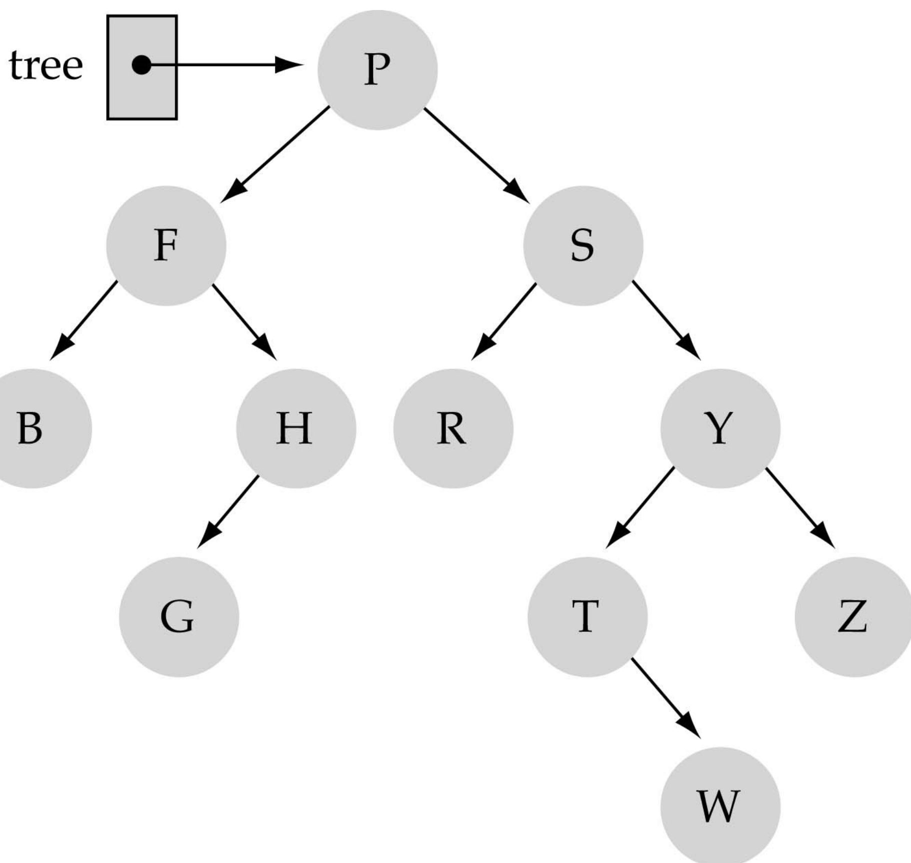


Function PrintTree (cont.)

```
void TreeType::PrintTree(ofstream& outFile)
{
 Print(root, outFile);
}
```

```
template<class ItemType>
void Print(TreeNode<ItemType>* tree, ofstream& outFile)
{
 if(tree != NULL) {
   Print(tree->left, outFile);
   outFile << tree->info; // “visit”
   Print(tree->right, outFile);
 }
}
```

Copy Constructor



Use preorder!

Copy Constructor (cont'd)

```
template<class ItemType>
TreeType<ItemType>::TreeType(const TreeType<ItemType>&
                                                     originalTree)
{
 CopyTree(root, originalTree.root);
}
```

```
template<class ItemType)
void CopyTree(TreeNode<ItemType>*& copy,
                      TreeNode<ItemType>* originalTree)
{
 if(originalTree == NULL)
   copy = NULL;
 else {
   copy = new TreeNode<ItemType>;      // "visit"
   copy->info = originalTree->info;
   CopyTree(copy->left, originalTree->left);
   CopyTree(copy->right, originalTree->right);
 }
}
```

preorder

GetNextItem

```
template<class ItemType>
void TreeType<ItemType>::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;
  }
}
```

Applications

- ***Preorder***
  - Tree copying
  - Counting the number of nodes
  - Counting the number of leaves
  - Prefix notation from a expression tree
- ***Postorder traversal***
  - Deleting a binary tree
  - All stack oriented programming language
  - Calculator programs
  - postfix notation in an expression tree – used in calculators

Comparing Binary Search Trees to Linear Lists

Big-O Comparison			
Operation	Binary Search Tree	Array-based List	Linked List
Constructor	O(1)	O(1)	O(1)
Destructor	O(N)	O(1)	O(N)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
RetrieveItem	O(logN)*	O(logN)	O(N)
InsertItem	O(logN)*	O(N)	O(N)
DeleteItem	O(logN)*	O(N)	O(N)

*assuming h=O(logN)