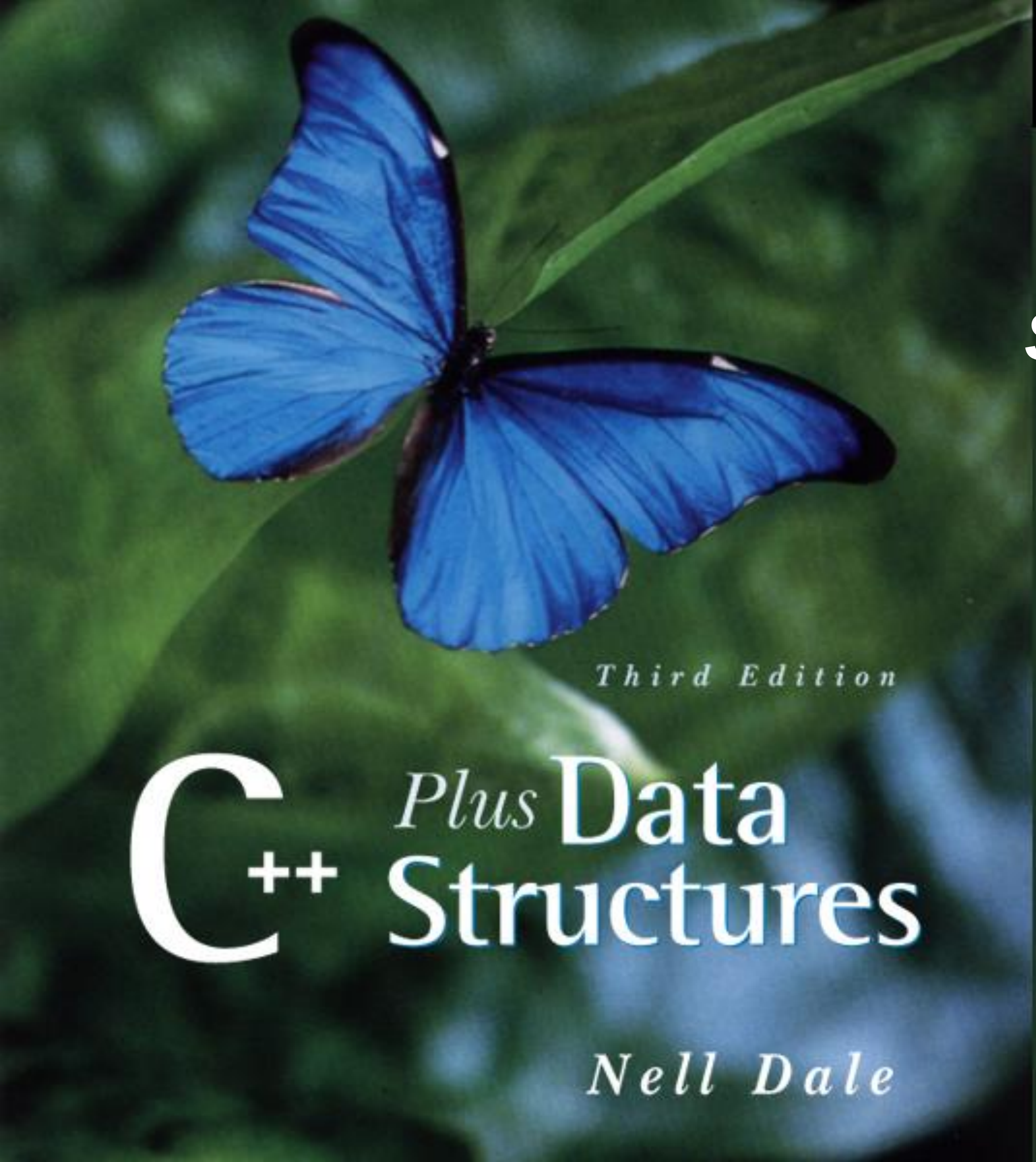


Chapter 8-2

Binary Search Trees

Function DeleteItem

First, find the item; then, delete it

Important: binary search tree property must be preserved!!

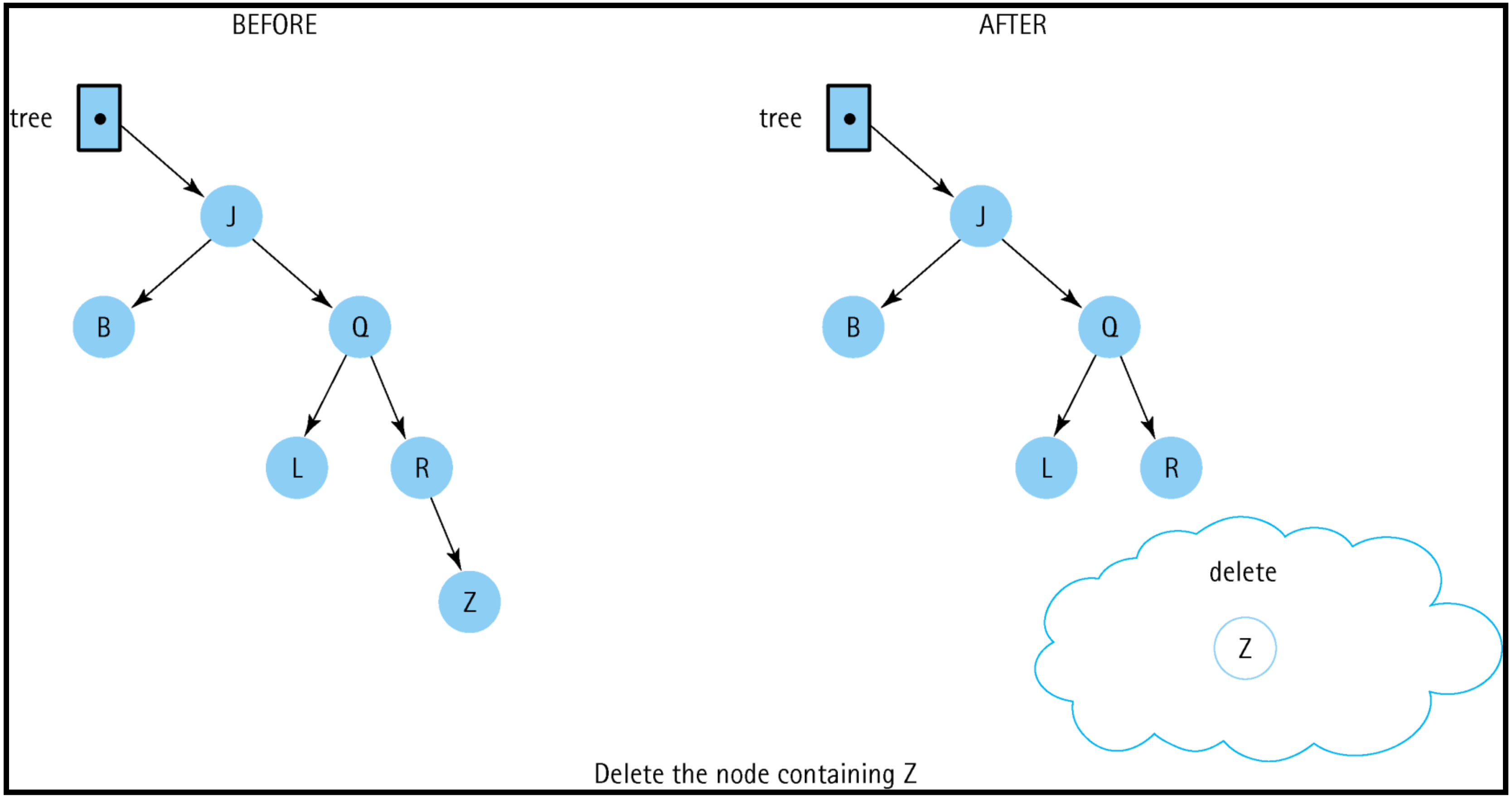
We need to consider three different cases:

(1) Deleting a leaf

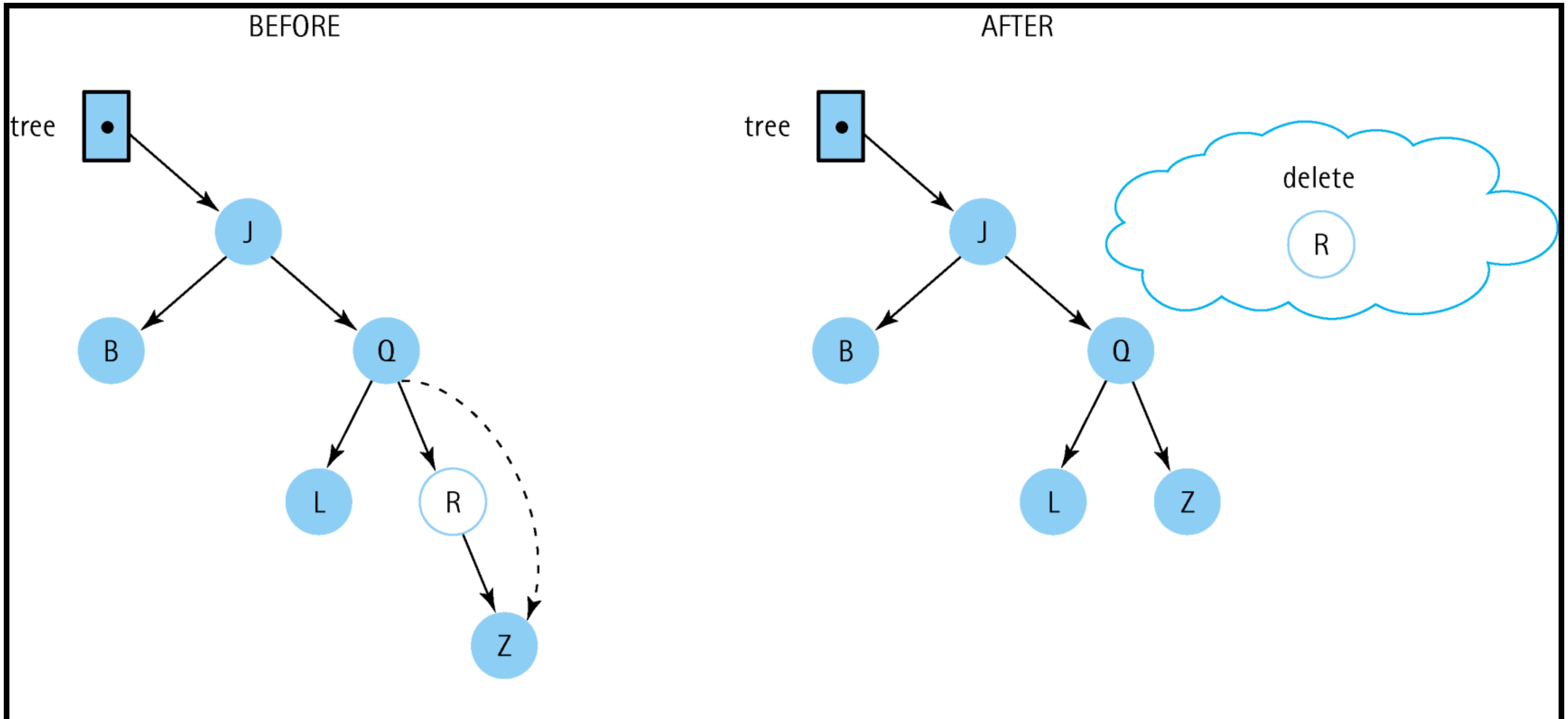
(2) Deleting a node with only one child

(3) Deleting a node with two children

Deleting a Leaf Node

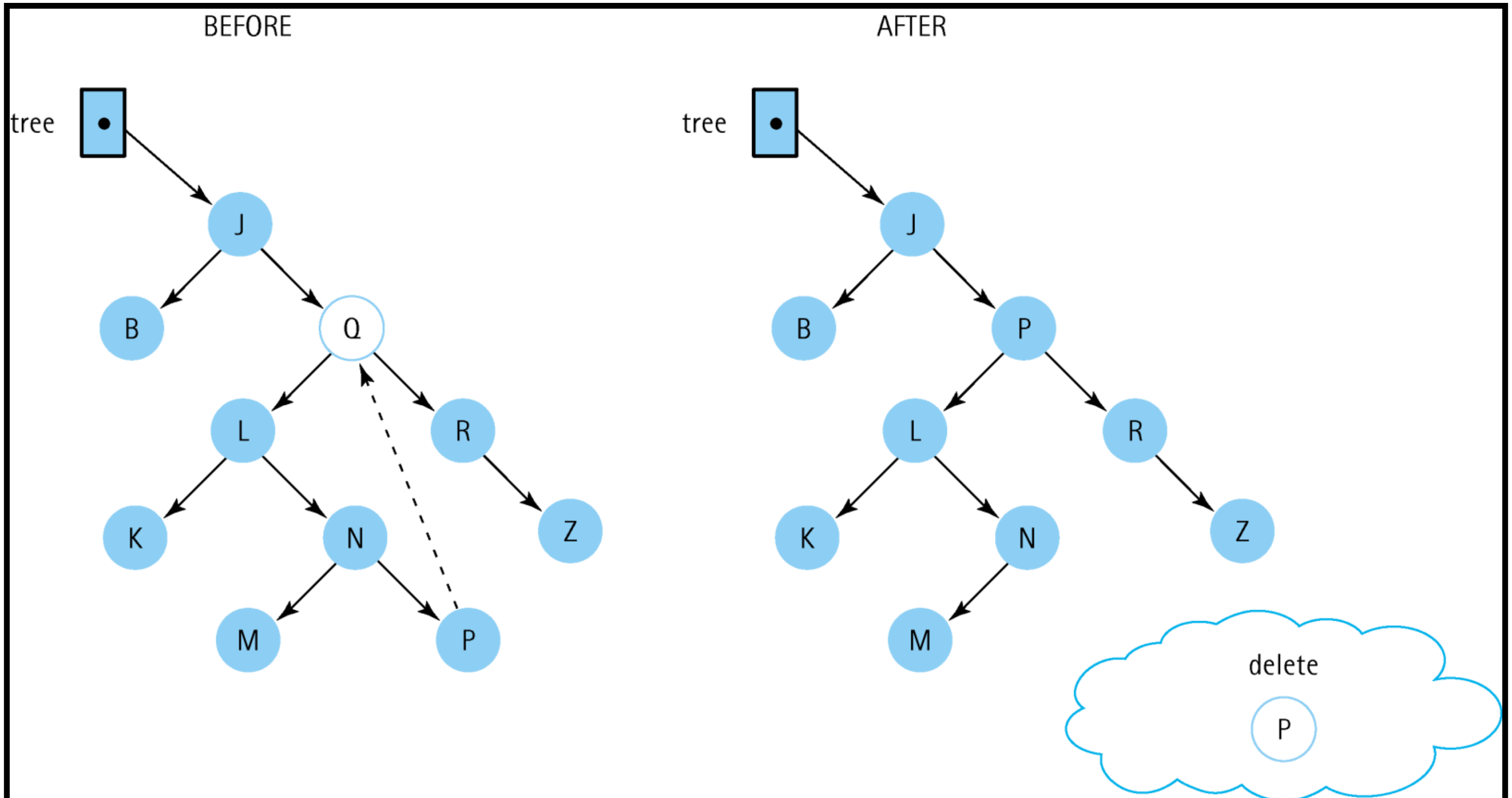


Deleting a Node with One Child



Delete the node containing R

Deleting a Node with Two Children



Delete the node containing Q

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

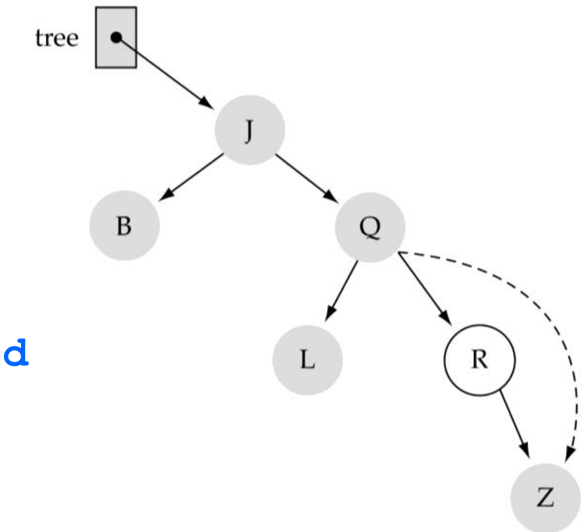
Delete predecessor node

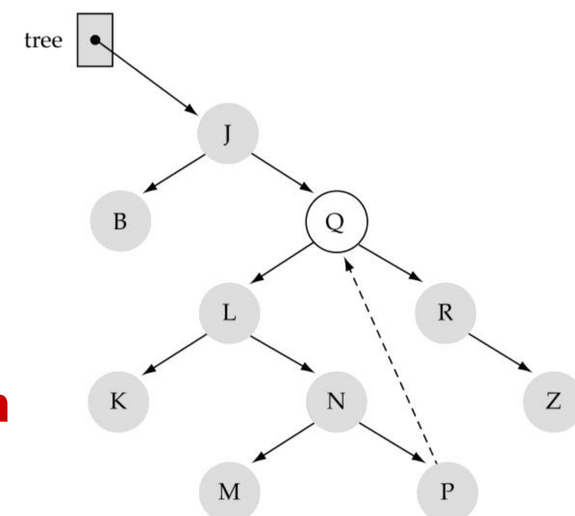
DeleteNode Algorithm

```
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

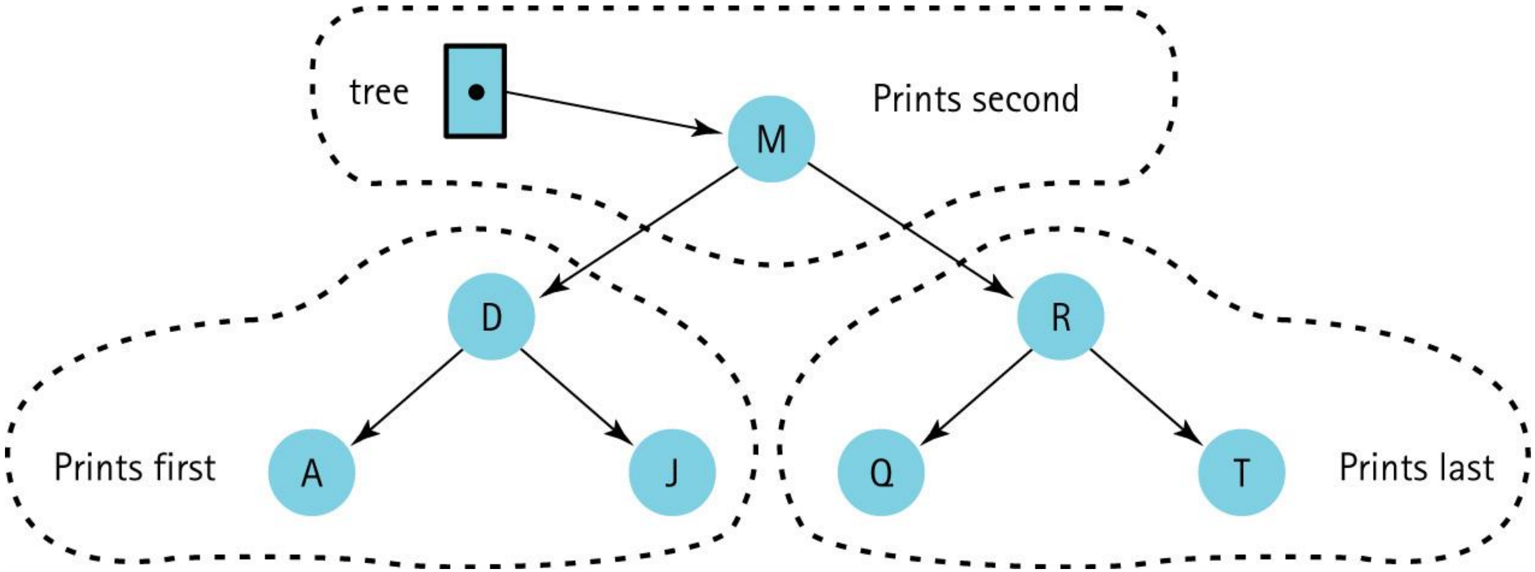
```
void Delete(TreeNode*& tree, ItemType
  item)
{
  if (item < tree->info)
    Delete(tree->left, item);
  else if (item > tree->info)
    Delete(tree->right, item);
  else
    DeleteNode(tree);  // Node found
}
```

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?

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

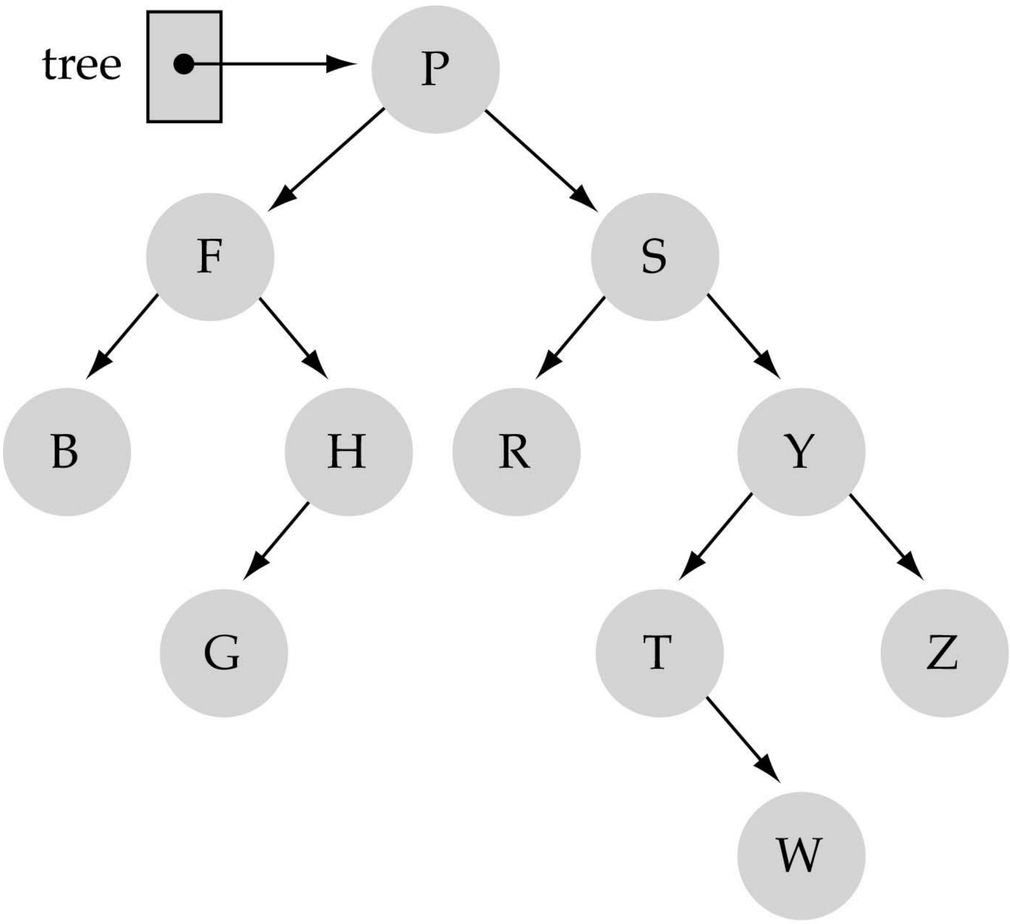
```
void PrintTree(TreeNode* tree,
 std::ofstream& outFile)
{
 if (tree != NULL)
 {
   PrintTree(tree->left, outFile);
   outFile << tree->info;
   PrintTree(tree->right, outFile);
 }
}
```

Is that all there is?

Class Constructor

```
template<class ItemType>
TreeType<ItemType>::TreeType()
{
 root = NULL;
}
```

Class Destructor



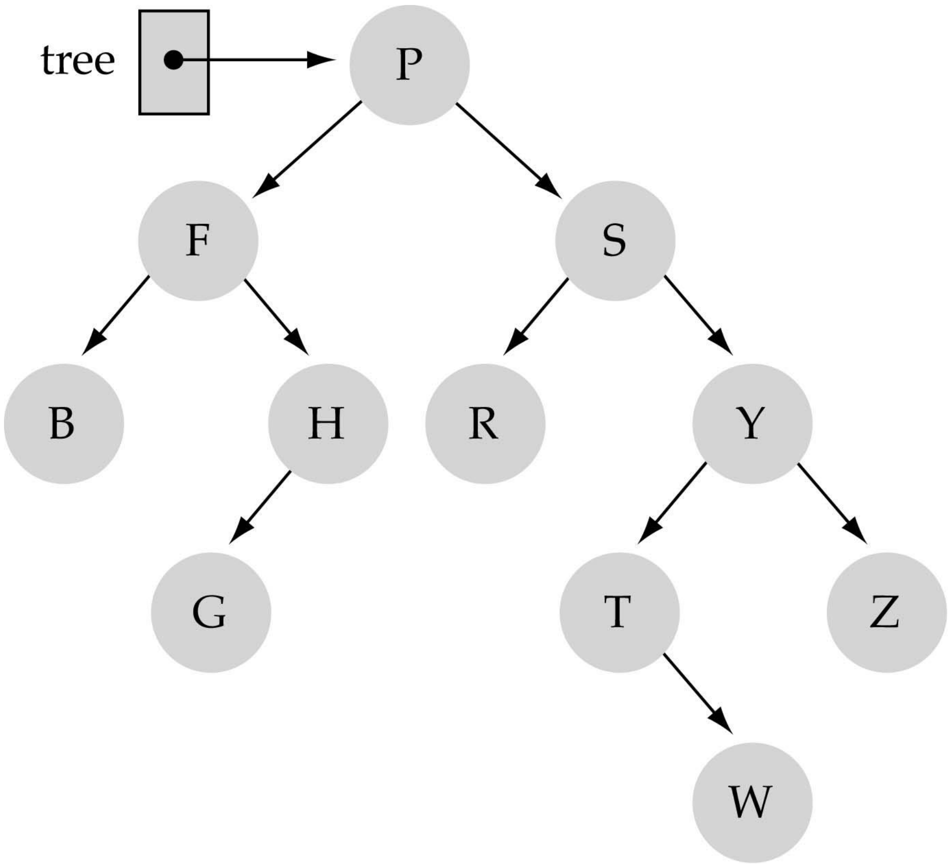
How should we delete the nodes of a tree?

Destructor

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

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

Copy Constructor



How should we create a copy of a 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)
```

What traversal order de we use?

Code for CopyTree

```
TreeType::TreeType(const TreeType& originalTree)
{
  CopyTree(root, originalTree.root);
}
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);
  }
}
```

Tree Traversal

A tree traversal means visiting all the nodes in the tree

"visit" means that the algorithm does something with the values in the node, e.g., print the value

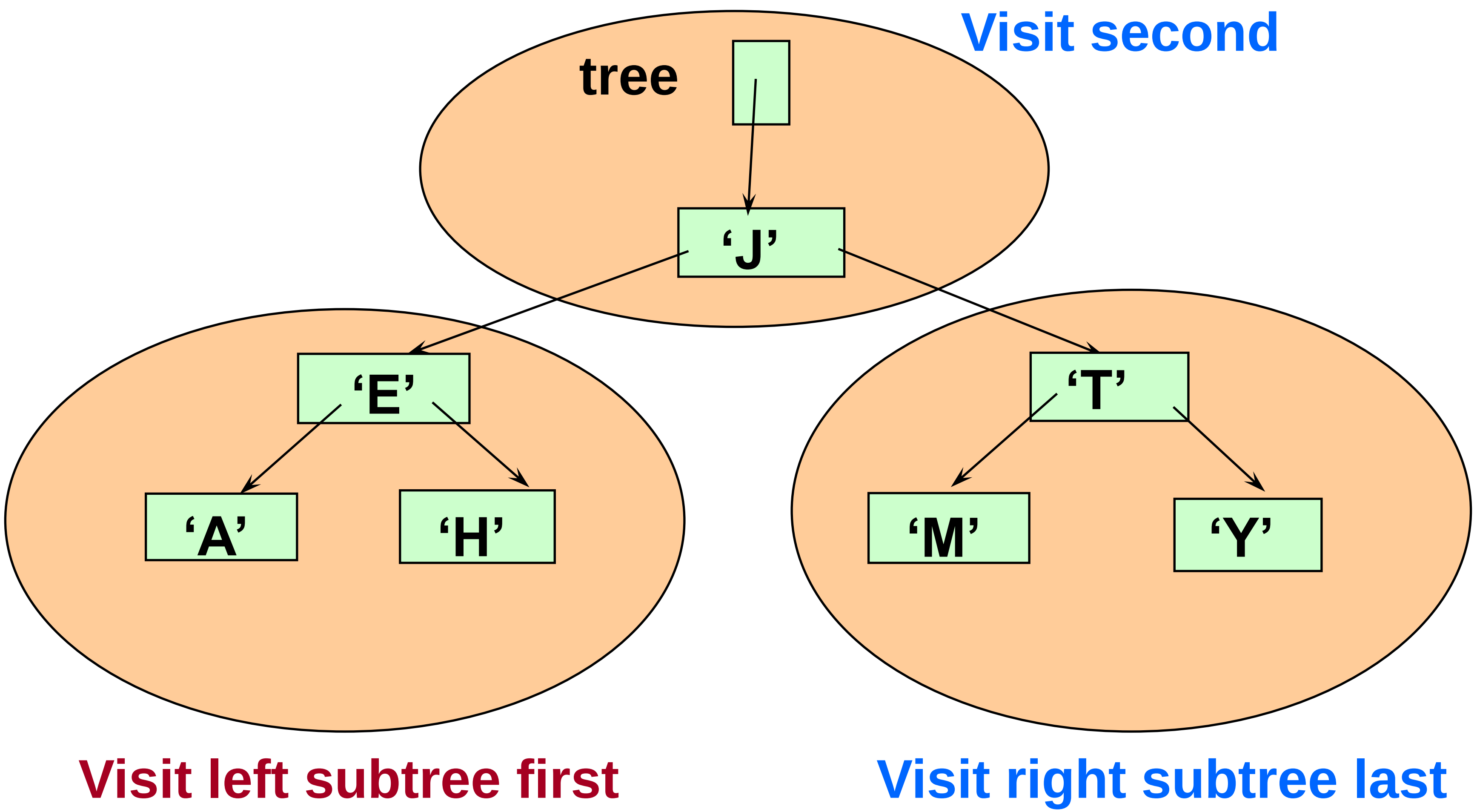
Tree Traversal Methods

There are mainly three ways to traverse a tree:

1) Inorder Traversal
2) Postorder Traversal
3) Preorder Traversal

Inorder Traversal

Inorder Traversal: A E H J M T Y



Inorder(tree)

if tree is not NULL

Inorder(Left(tree))

Visit Info(tree)

Inorder(Right(tree))

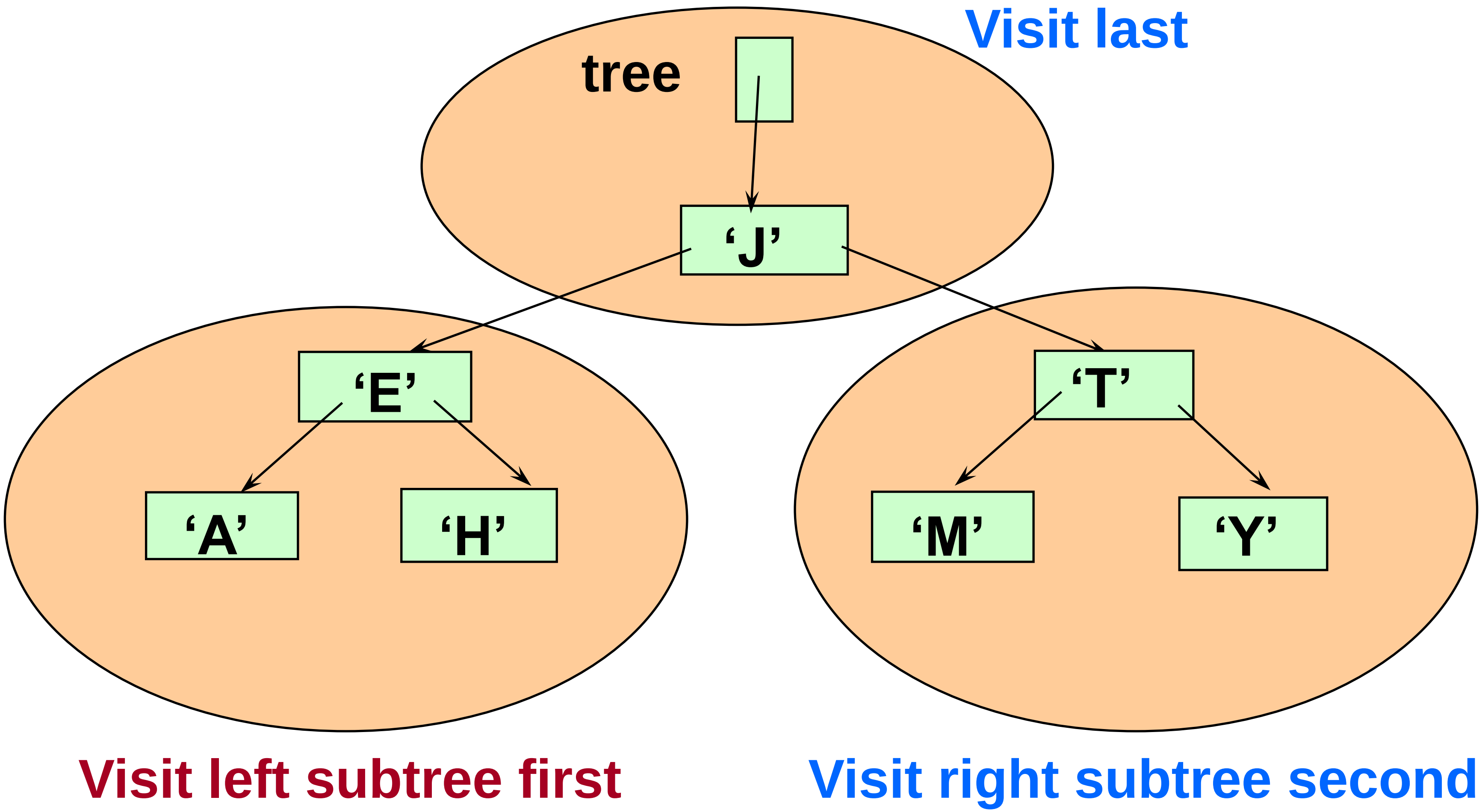
To print in alphabetical order

InOrder Implementation

```
void InOrder(TreeNode* tree,
  QueType& inQue)
{
 if(tree != NULL) {
   InOrder(tree->left, inQue);
   inQue.Enqueue(tree->info);
   InOrder(tree->right, inQue);
 }
}
```

Postorder Traversal

Postorder Traversal: A H E M Y T J



Postorder(tree)

if tree is not NULL

Postorder(Left(tree))

Postorder(Right(tree))

Visit Info(tree)

Visits leaves first

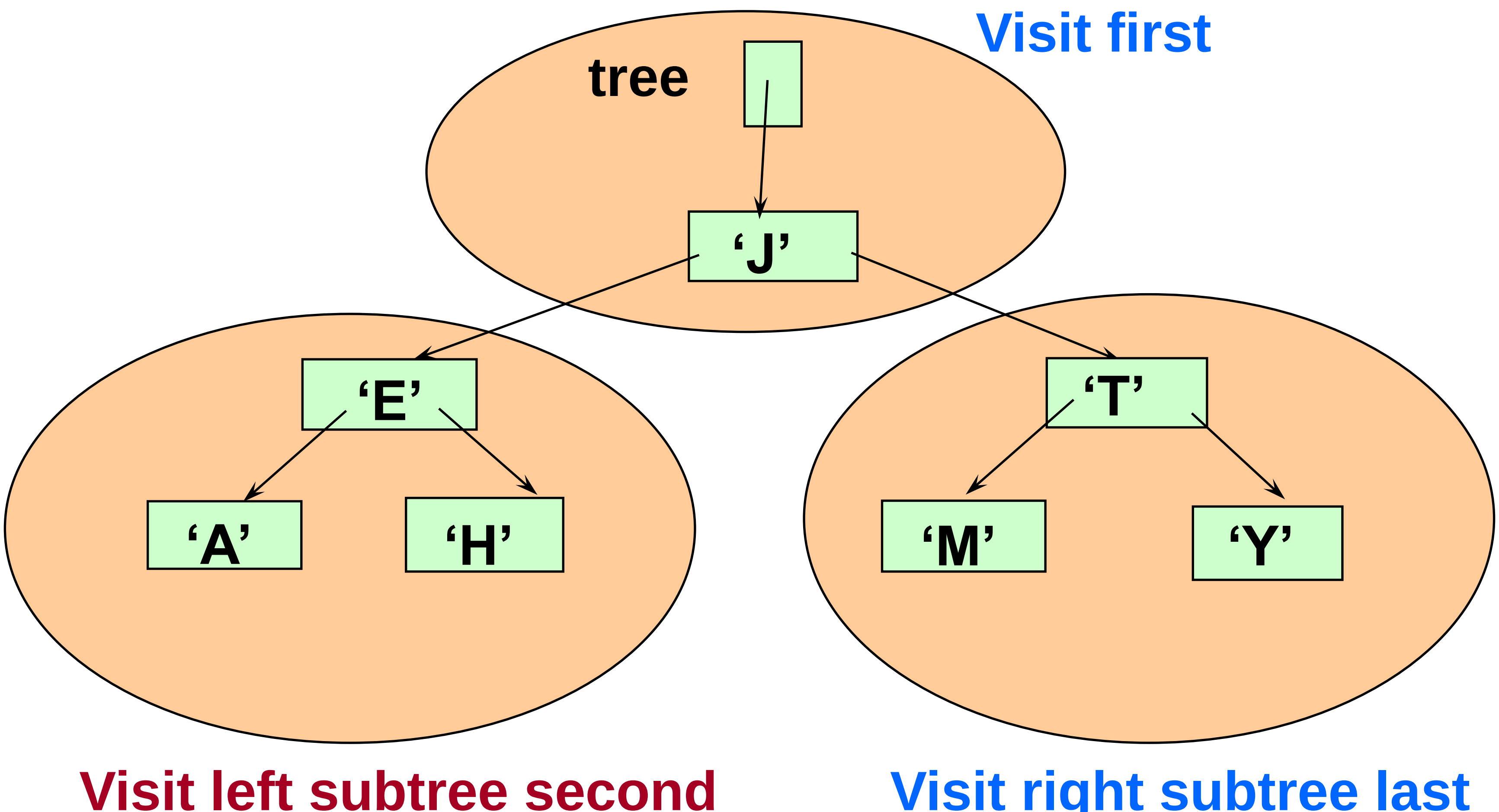
(good for deletion)

PostOrder Implementation

```
void PostOrder(TreeNode *tree,
  QueType& postQue)
{
 if(tree != NULL) {
   PostOrder(tree->left, postQue);
   PostOrder(tree->right, postQue);
   postQue.Enqueue(tree->info);
 }
}
```

Preorder Traversal

Preorder Traversal: J E A H T M Y



Preorder(tree)

if tree is not NULL

    Visit Info(tree)

    Preorder(Left(tree))

    Preorder(Right(tree))

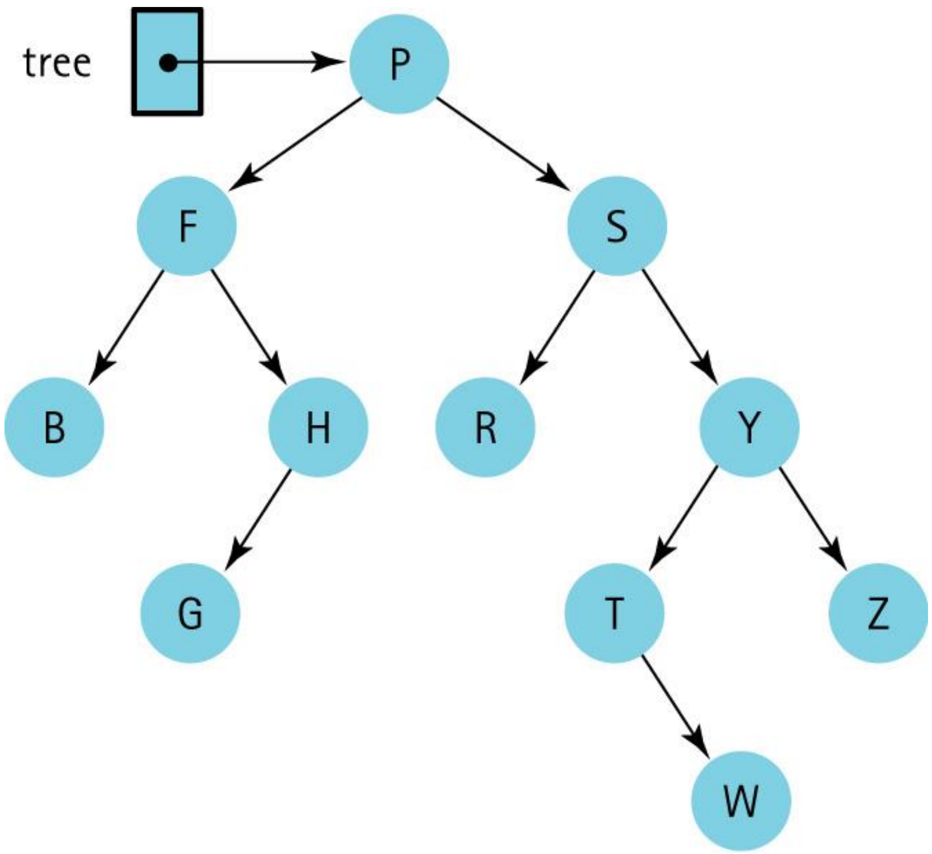
Useful with binary trees

(not binary search trees)

PreOrder Implementation

```
void PreOrder(TreeNode* tree,
  QueType& preQue)
{
 if(tree != NULL) {
   preQue.Enqueue(tree->info);
   PreOrder(tree->left, preQue);
   PreOrder(tree->right, preQue);
 }
}
```

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

For efficiency, ResetTree generates a queue of node contents in the indicated order

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

Modification of Class TreeType

```
enum OrderType {PRE_ORDER, IN_ORDER,
  POST_ORDER};

class TreeType {
  public:
   // same as before
  private:
    TreeNode* root;
    QueType preQue;
    QueType inQue;
    QueType postQue;
};
```

new private data

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

```
void TreeType::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;
  }
}
```

Prototypes of Traversal Functions

```
void PreOrder(TreeNode*,
  QueType&);

void InOrder(TreeNode*,
  QueType&);

void PostOrder(TreeNode*,
  QueType&);
```

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

InsertItem

Create a node to contain the new item.

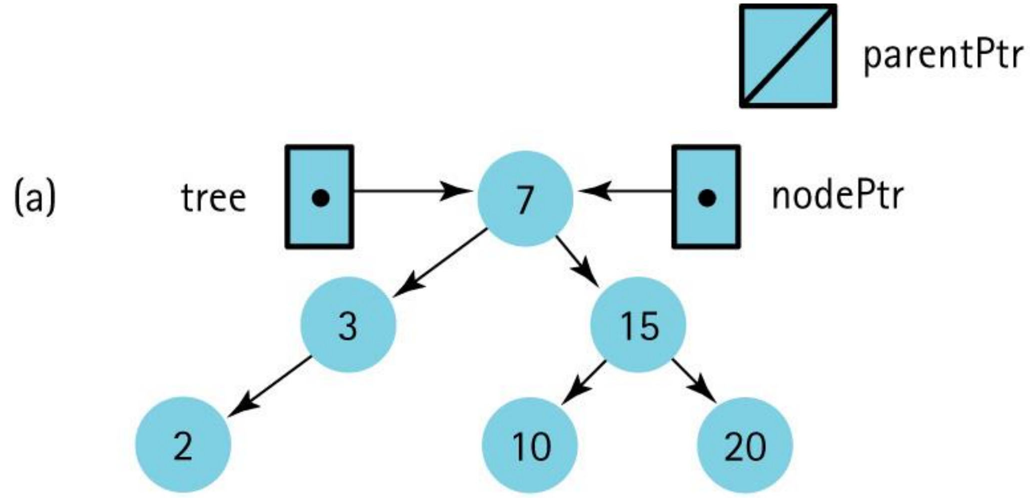
Find the insertion place.

Attach new node.

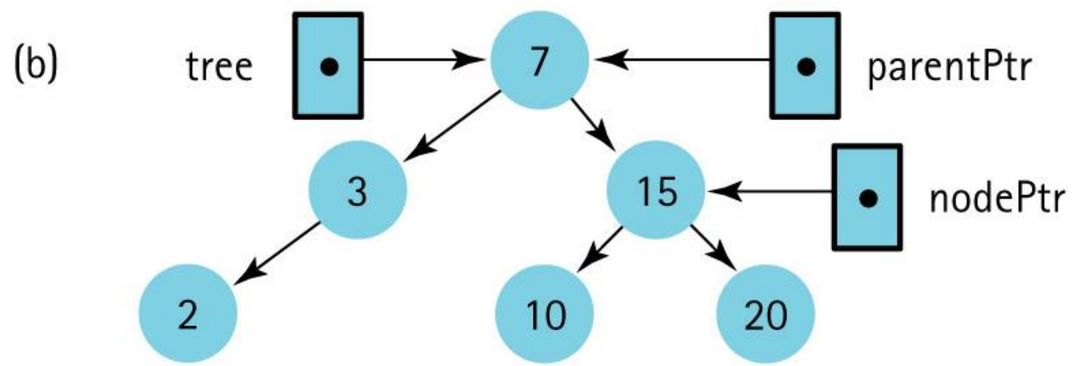
Find the insertion place

FindNode(tree, item, nodePtr, parentPtr);

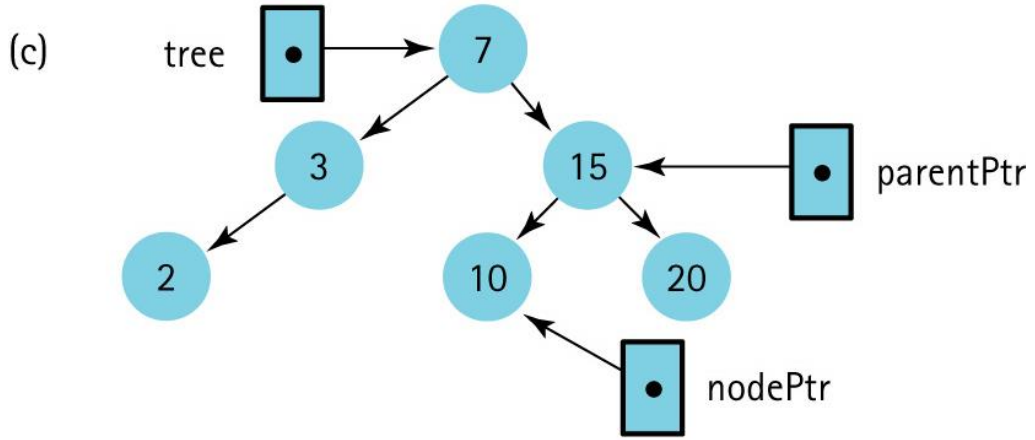
Using function FindNode to find the insertion point (Insert 13)



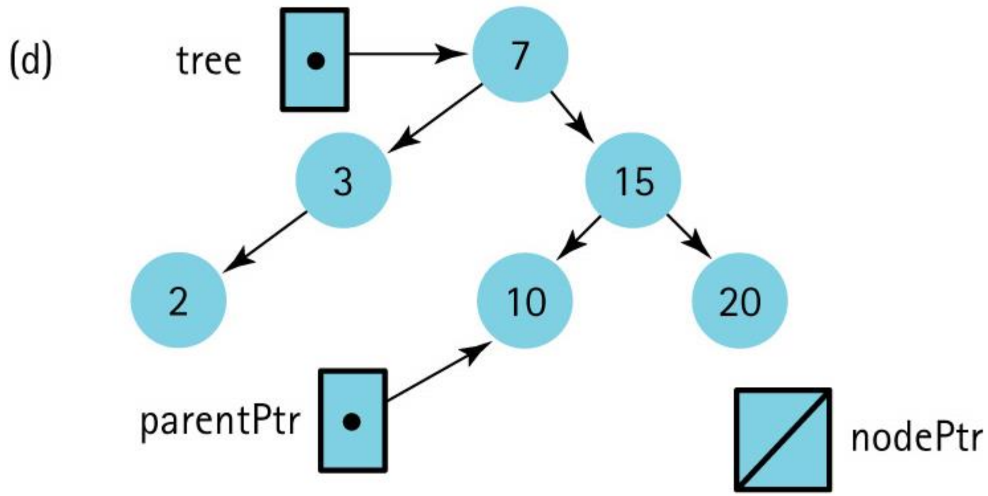
Using function FindNode to find the insertion point (Insert 13)



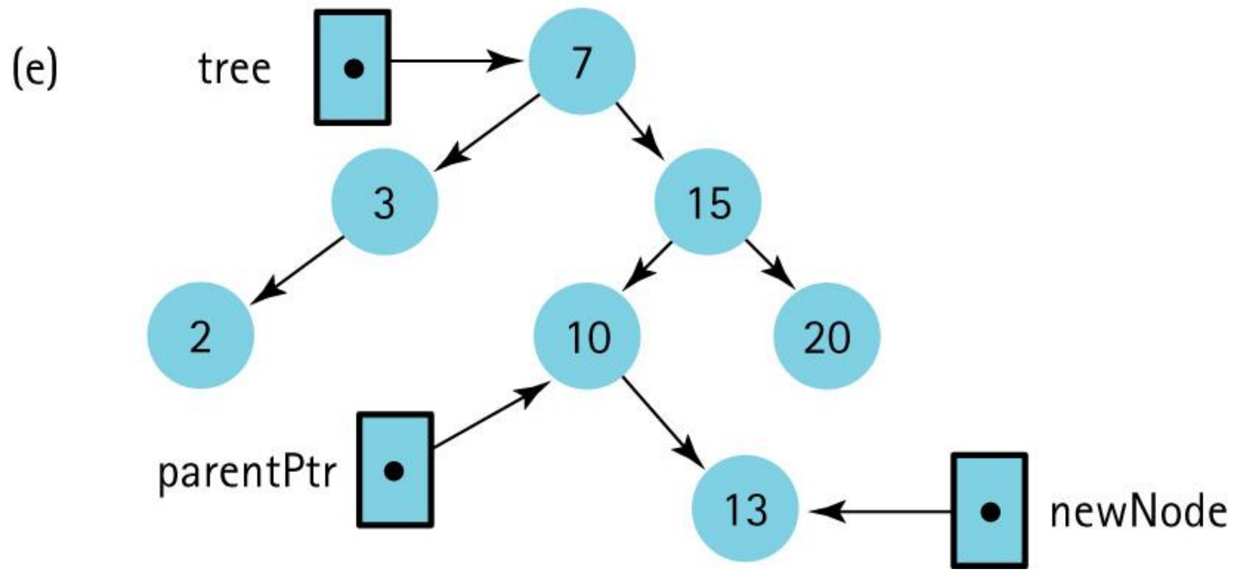
Using function FindNode to find the insertion point (Insert 13)



Using function FindNode to find the insertion point (Insert 13)



Using function FindNode to find the insertion point (Insert 13)



AttachNewNode

if item < Info(parentPtr)

    Set Left(parentPtr) to newNode

else

    Set Right(parentPtr) to newNode

What's wrong?

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
```

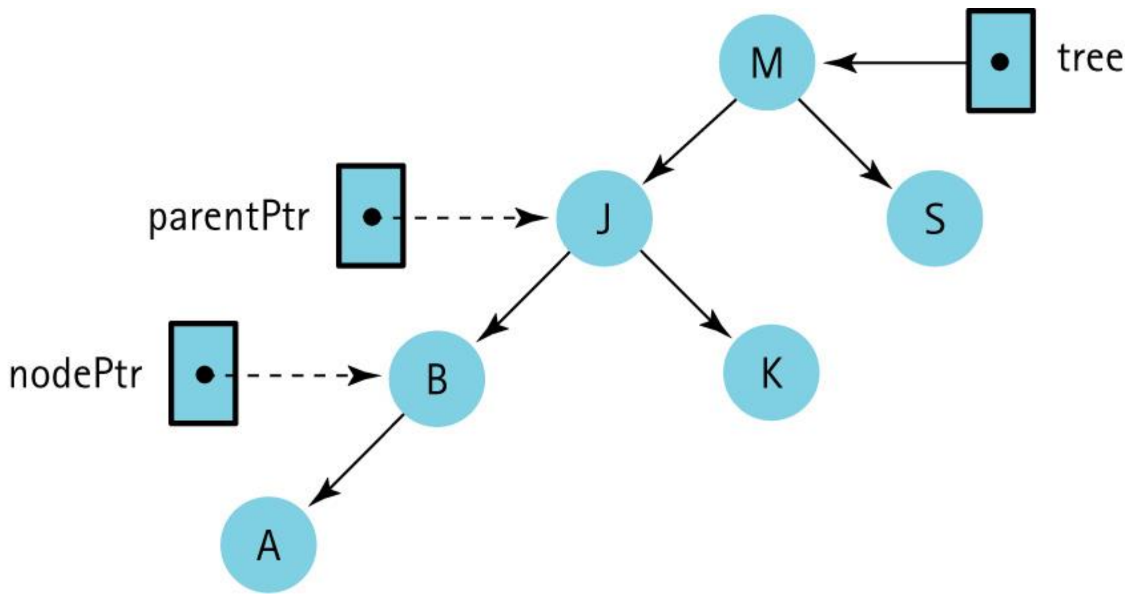
Code for InsertItem

```
void TreeType::InsertItem(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;
}
```

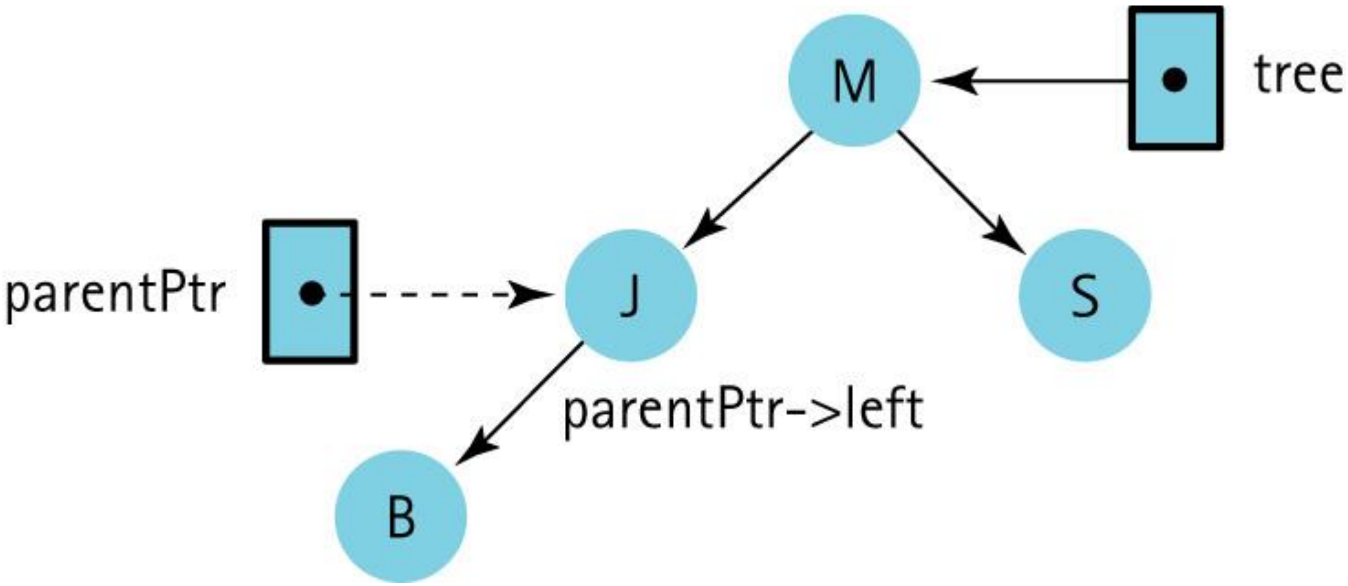
Code for `DeleteItem`

```
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);
}
```

Pointers nodePtr and parentPtr Are External to the Tree



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



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)

With Array Representation

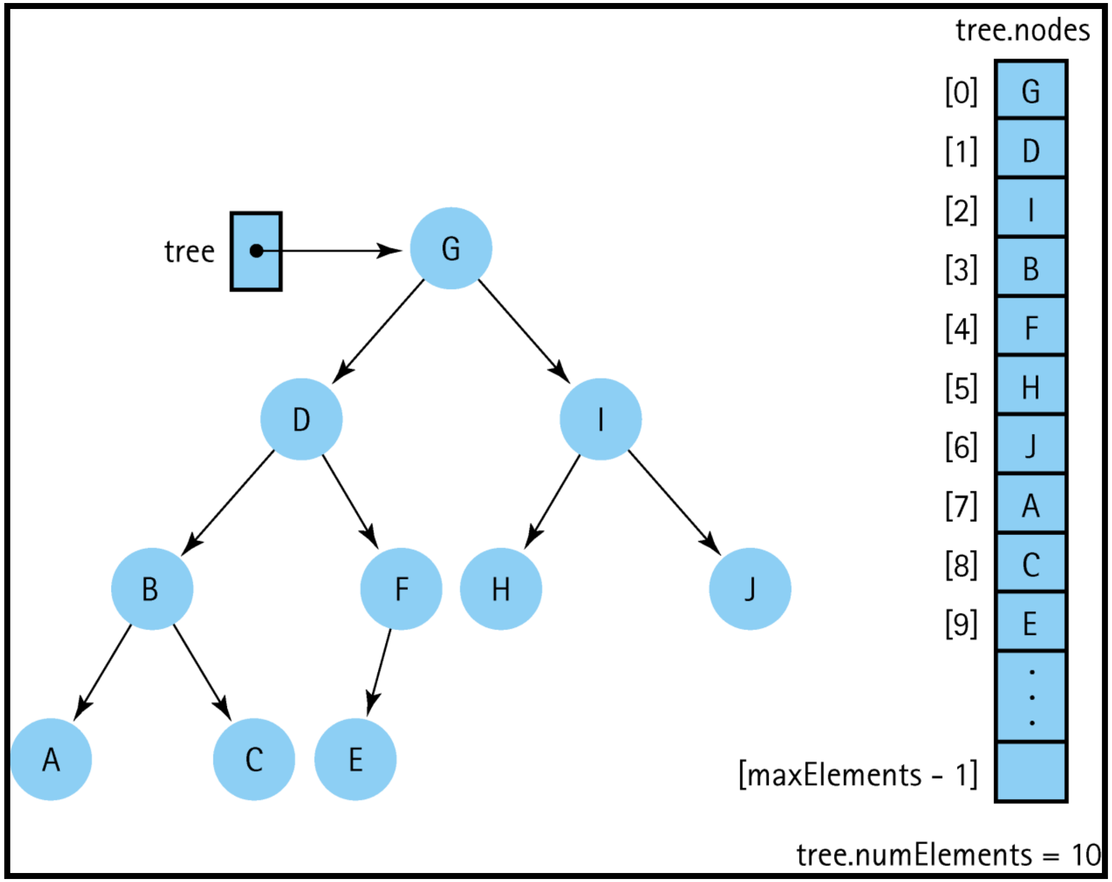
For any node `tree.nodes[index]`

its left child is in `tree.nodes[index*2 + 1]`

right child is in `tree.nodes[index*2 + 2]`

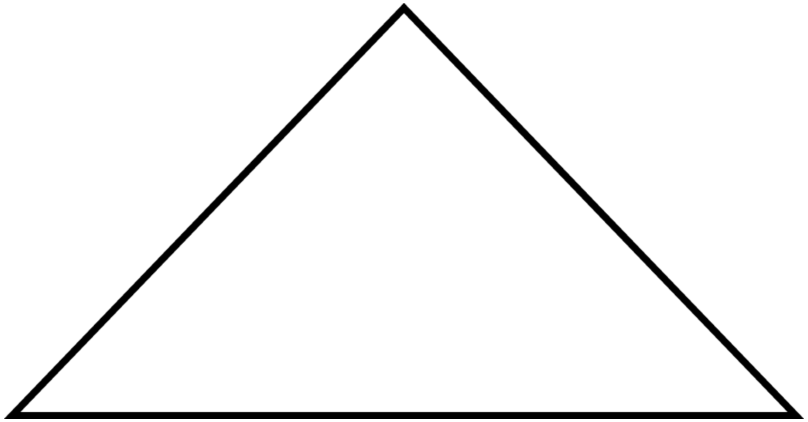
its parent is in `tree.nodes[(index – 1)/2]`.

A Binary Tree and Its Array Representation



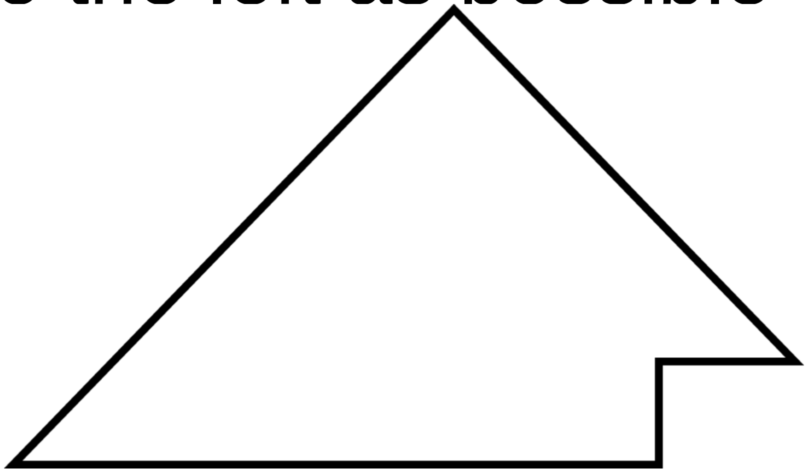
Definitions

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

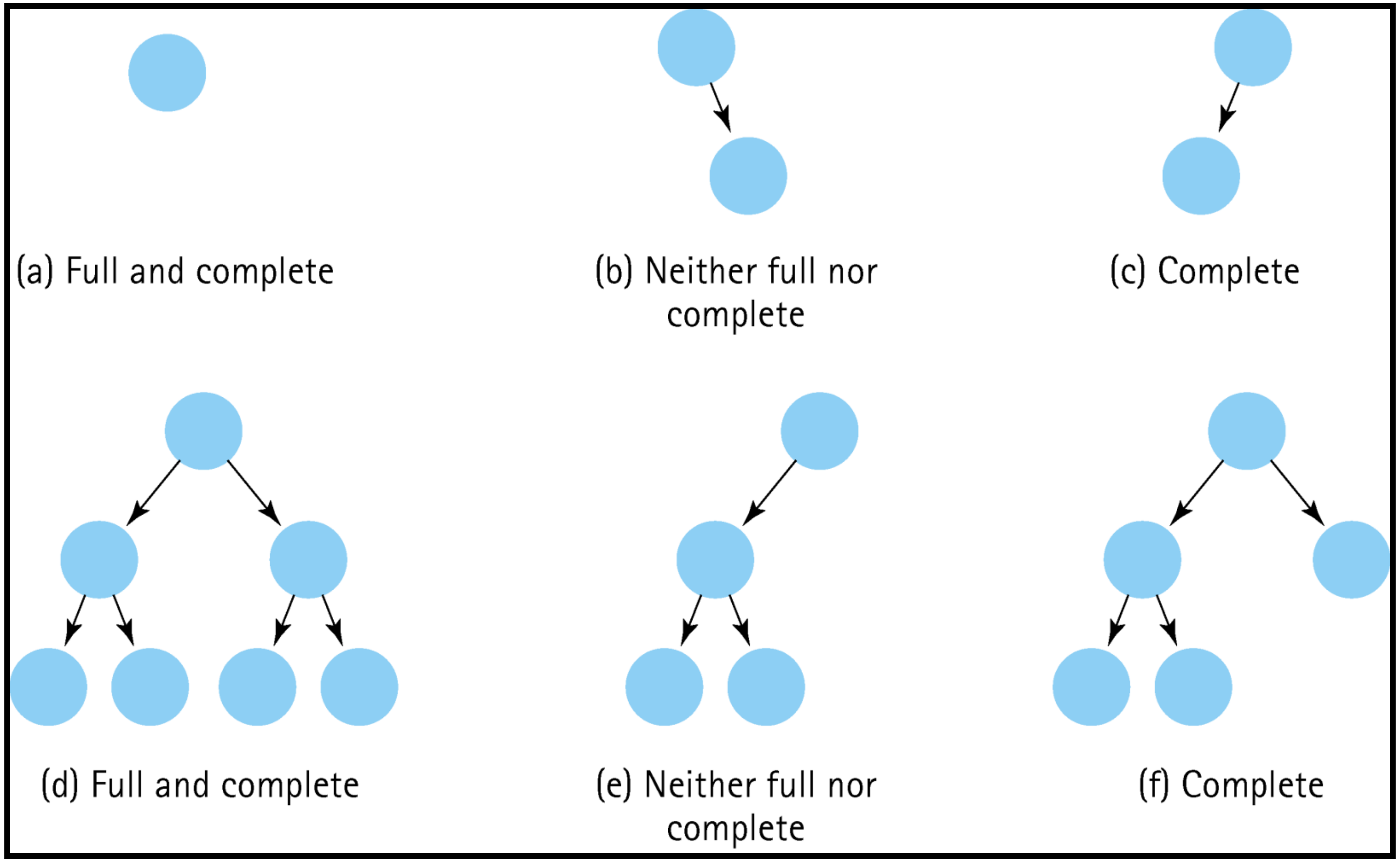


Definitions (cont.)

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