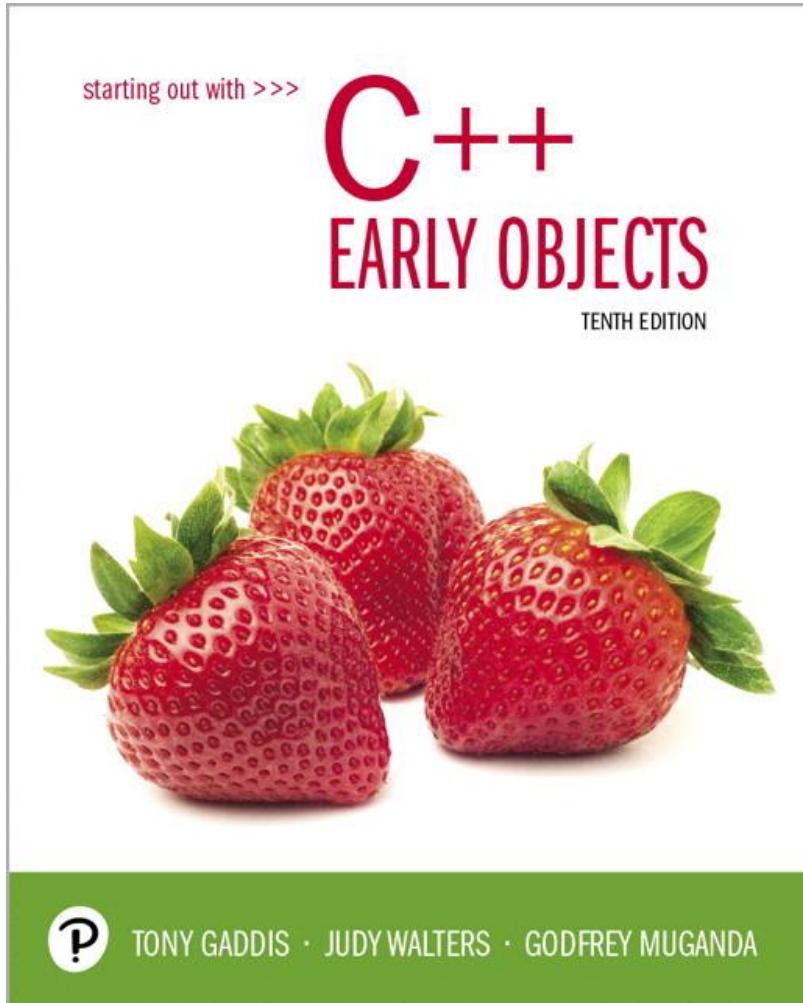


# Starting Out with C++ Early Objects

Tenth Edition



## Chapter 20

### Binary Trees

# Topics

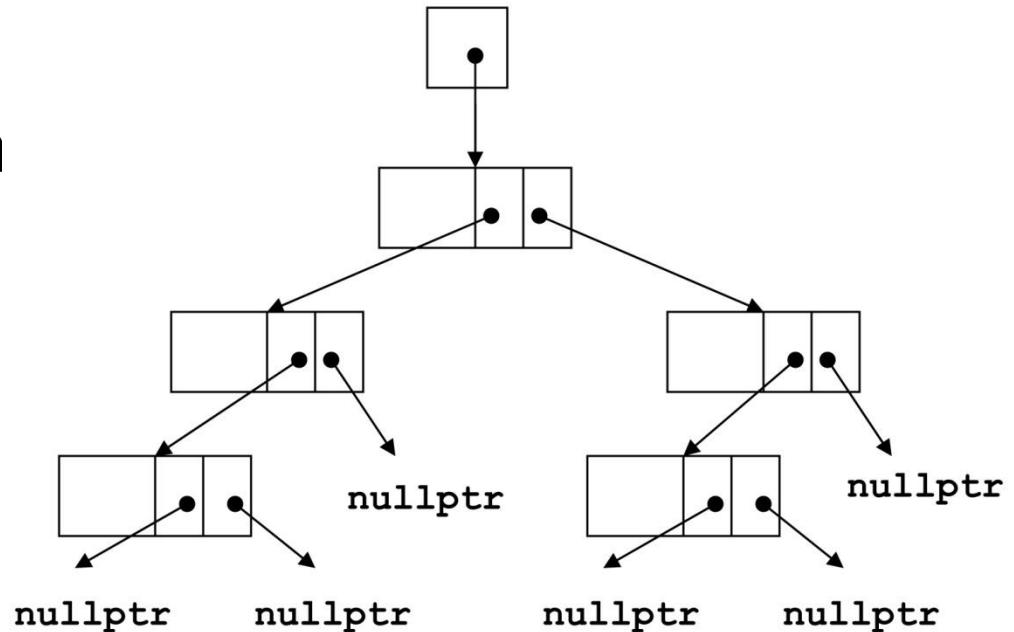
20.1 Definition and Applications of Binary Trees

20.2 Binary Search Tree Operations

20.3 Template Considerations for Binary  
Search Trees

# 20.1 Definition and Applications of Binary Trees

- **Binary tree**: a nonlinear data structure in which each node may point to 0, 1, or two other nodes
- The nodes that a node  $N$  points to are the (left or right) **children** of  $N$

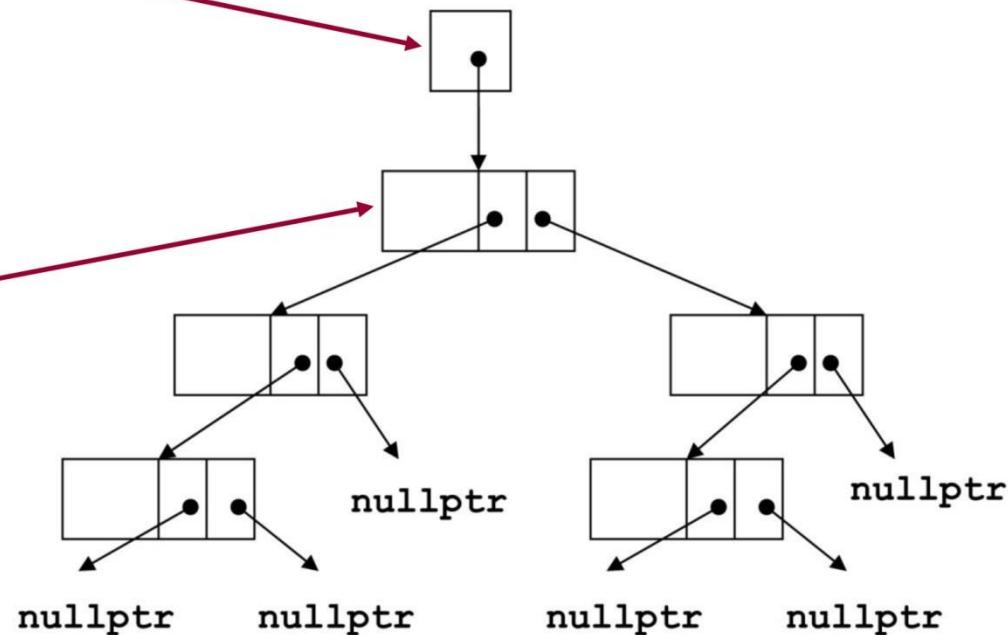


# Terminology

- If a node  $N$  is a child of another node  $P$ , then  $P$  is called the **parent** of  $N$
- A node that has no children is called a **leaf node**
- In a binary tree there is a unique node with no parent. This is the **root** of the tree. Every node other than the root of the tree has exactly one parent.

# Binary Tree Terminology

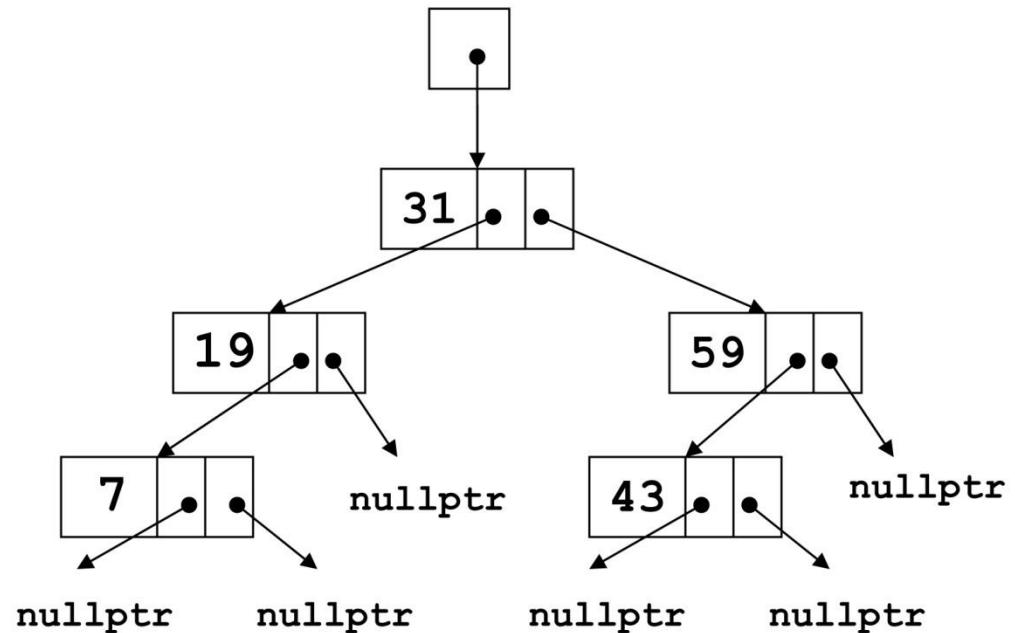
- **Root pointer:** like a head pointer for a linked list, it points to the root node of the binary tree
- **Root node:** the node with no parent. Similar to the head of a linked list.



# Binary Tree Terminology 1 of 5

**Leaf nodes:** nodes that have no children

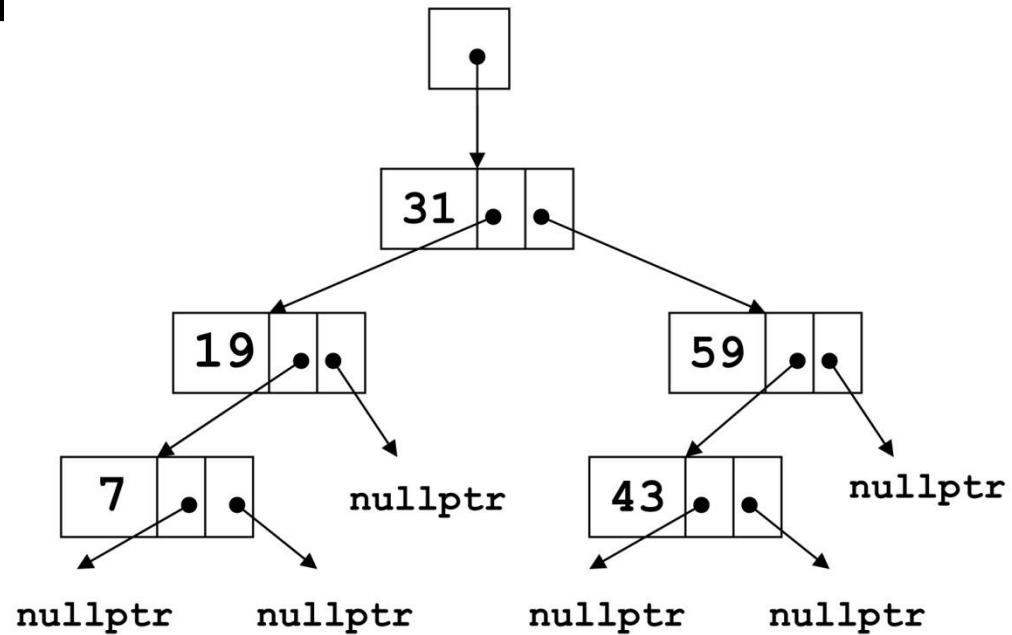
The nodes containing 7 and 43 are leaf nodes



# Binary Tree Terminology 2 of 5

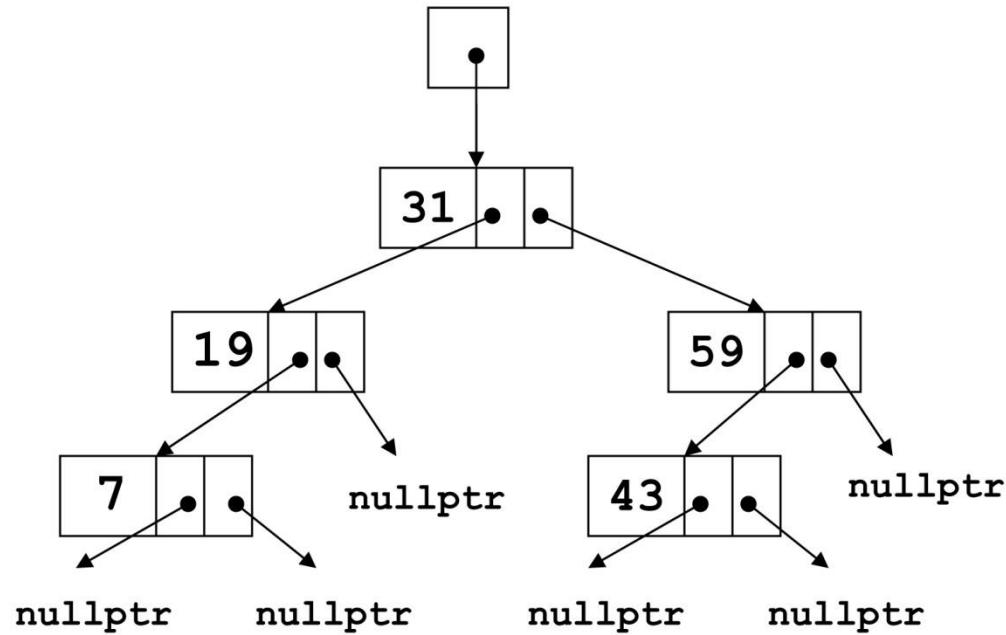
Child nodes, children:

The children of the node containing 31 are the nodes containing 19 and 59



# Binary Tree Terminology 3 of 5

The **parent** of the node containing 43 is the node containing 59



# Binary Tree Terminology 4 of 5

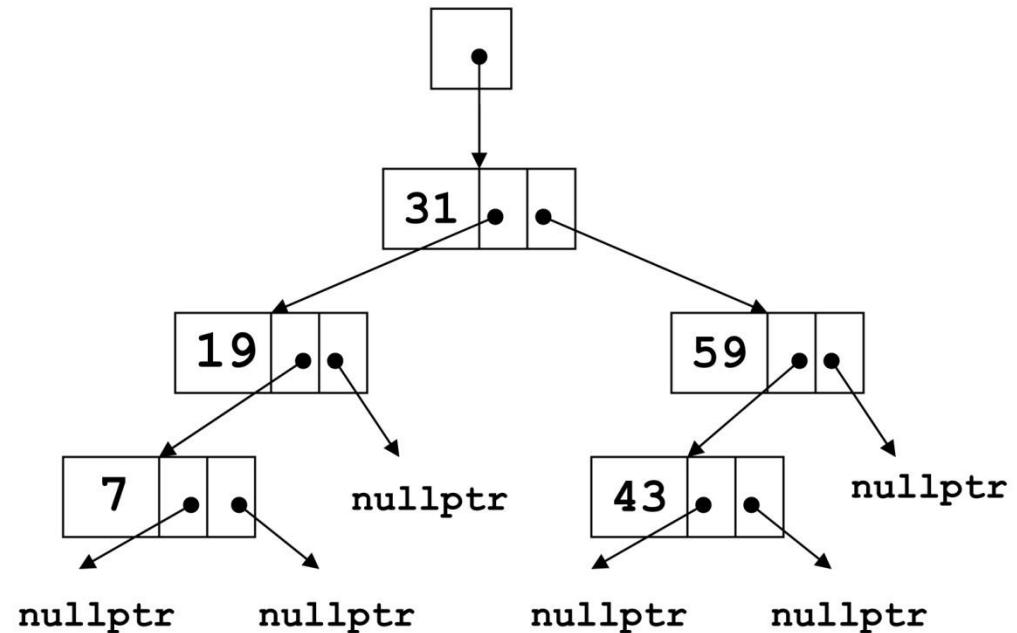
- A **subtree** of a binary tree is a part of the tree from a node  $N$  down to the leaf nodes
- Such a subtree is said to be rooted at  $N$ , and  $N$  is called the **root of the subtree**

# Subtrees of Binary Trees

- A subtree of a binary tree is itself a binary tree
- A nonempty binary tree consists of a root node, with the rest of its nodes forming two subtrees, called the left and right subtree

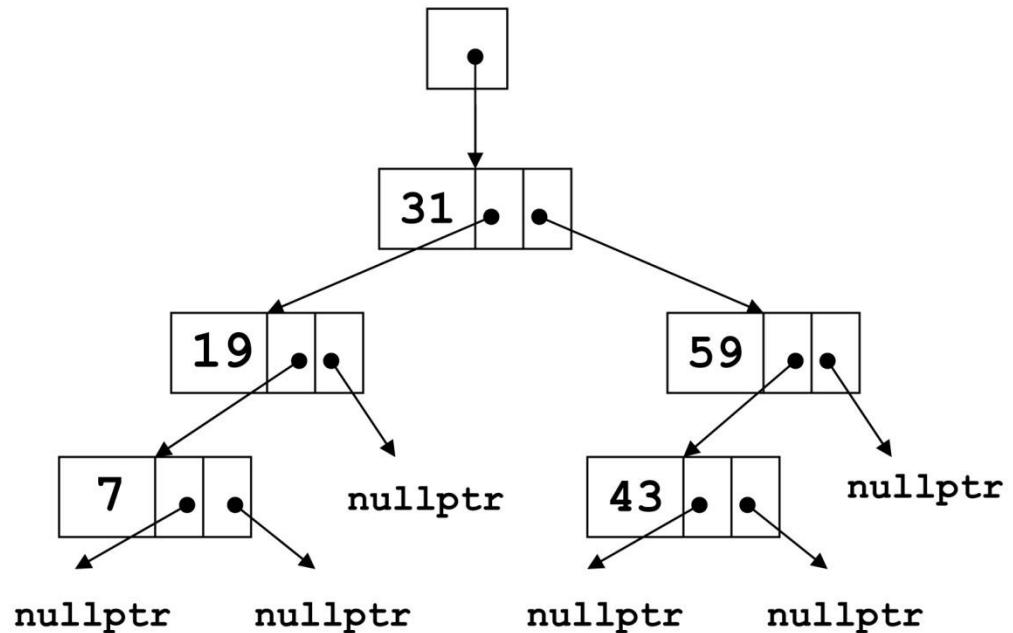
# Binary Tree Terminology 5 of 5

- The node containing 31 is the root
- The nodes containing 19 and 7 form the left subtree
- The nodes containing 59 and 43 form the right subtree



# Uses of Binary Trees

- **Binary search tree:** a binary tree whose data is organized to simplify searches
- Left subtree at each node contains data values less than the data in the node
- Right subtree at each node contains values greater than the data in the node



## 20.2 Binary Search Tree Operations

- **Create** a binary search tree
- **Insert a node** into a binary tree – put node into tree in its correct position to maintain order
- **Find a node** in a binary tree – locate a node with particular data value
- **Delete a node** from a binary tree – remove a node and adjust links to preserve the binary tree and the order

# Binary Search Tree Node

- A node in a binary tree is like a node in a linked list, except that it has two node pointer fields:

```
struct TreeNode  
{  
    int value;  
    TreeNode *left;  
    TreeNode *right;  
};
```

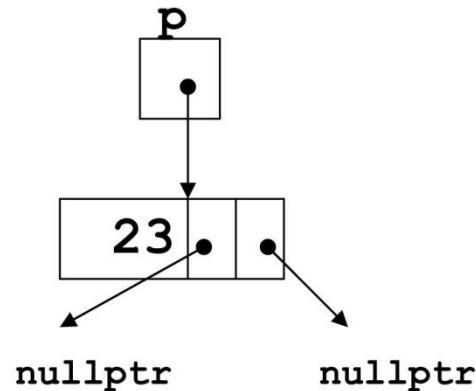
- A constructor with default values can aid in the creation of nodes.

# TreeNode Constructor

```
TreeNode::TreeNode(int val,  
                   TreeNode *l1=NULL,  
                   TreeNode *r1=NULL)  
{  
    value = val;  
    left = l1;  
    right = r1;  
}
```

# Creating a New Node

```
TreeNode *p;  
int num = 23;  
p = new TreeNode(num);
```



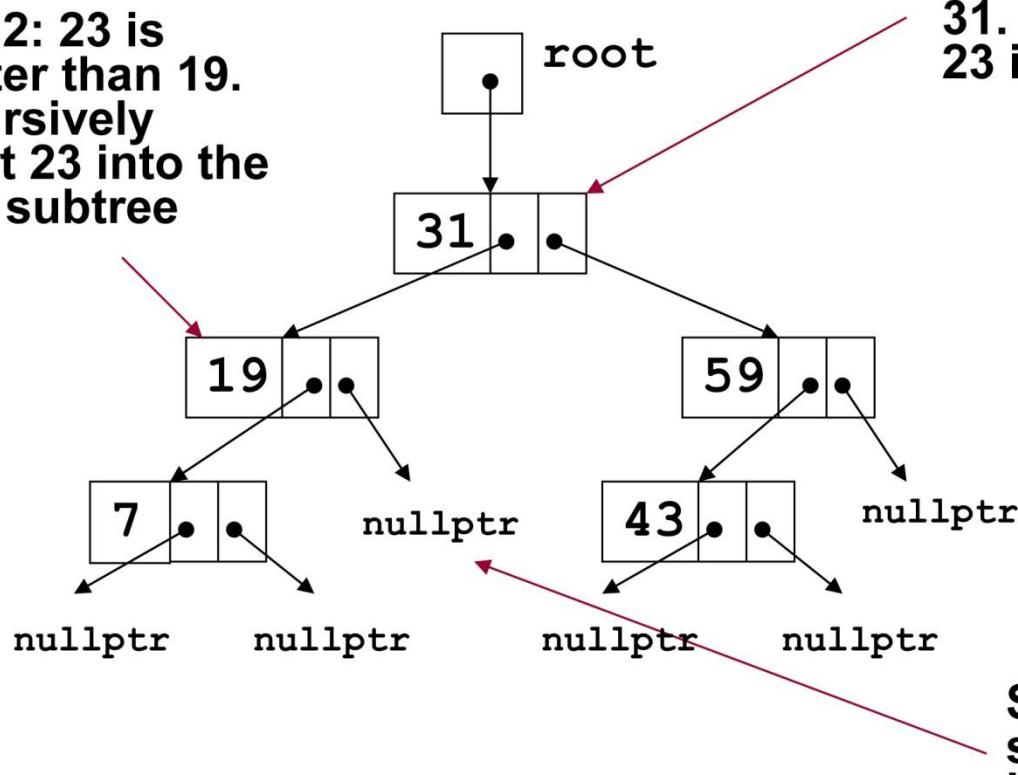
# Inserting an item into a Binary Search Tree 1 of 2

- 1) If the tree is empty, replace the empty tree with a new binary tree consisting of the new node as root, with empty left and right subtrees
- 2) Otherwise, if the item is less than the root node, recursively insert the item in the left subtree. If the item is greater than the root node, recursively insert the item into the right subtree

# Inserting an item into a Binary Search Tree 2 of 2

value to insert:  
23

**Step 2:** 23 is greater than 19. Recursively insert 23 into the right subtree



**Step 1:** 23 is less than 31. Recursively insert 23 into the left subtree

**Step 3:** Since the right subtree is NULL, insert 23 here

# Traversing a Binary Tree 1 of 2

Three traversal methods:

1) Inorder:

- a) Traverse left subtree of node
- b) Process data in node
- c) Traverse right subtree of node

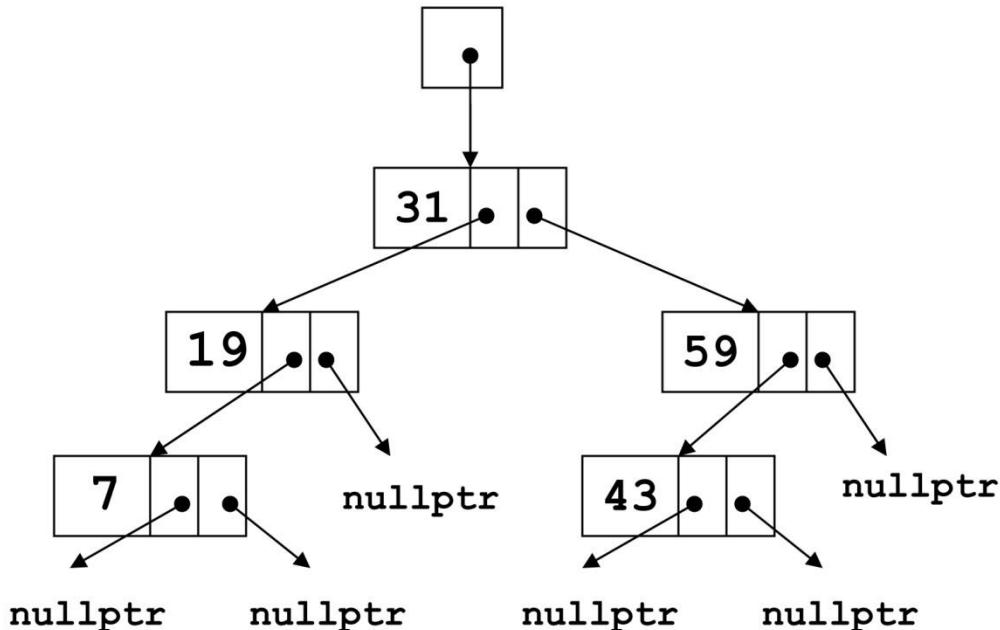
2) Preorder:

- a) Process data in node
- b) Traverse left subtree of node
- c) Traverse right subtree of node

3) Postorder:

- a) Traverse left subtree of node
- b) Traverse right subtree of node
- c) Process data in node

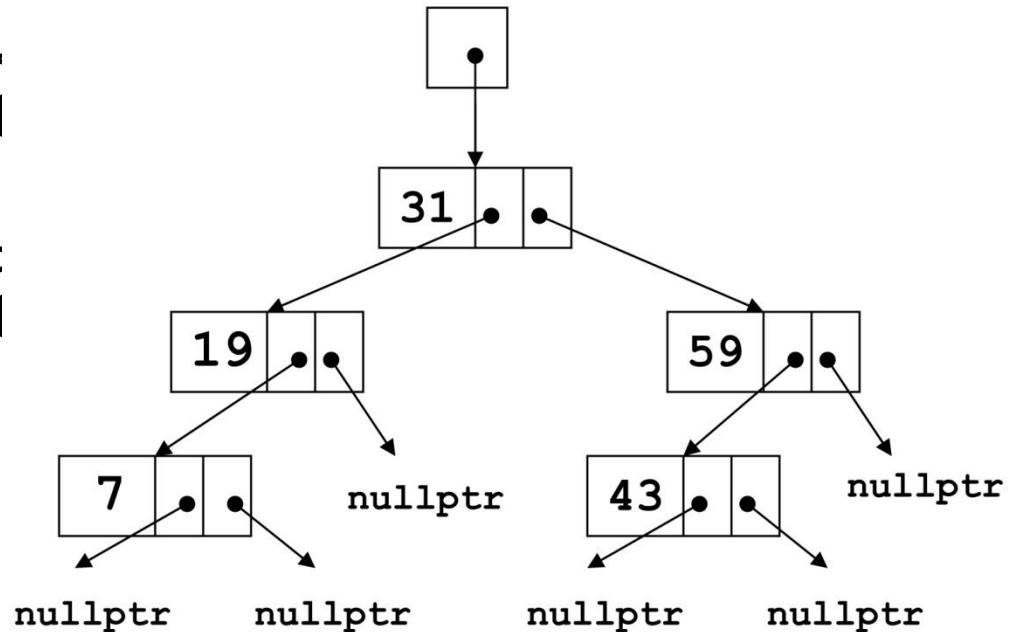
# Traversing a Binary Tree 2 of 2



TRAVERSAL METHOD	NODES ARE VISITED IN THIS ORDER
Inorder	7, 19, 31, 43, 59
Preorder	31, 19, 7, 59, 43
Postorder	7, 19, 43, 59, 31

# Searching in a Binary Tree 1 of 2

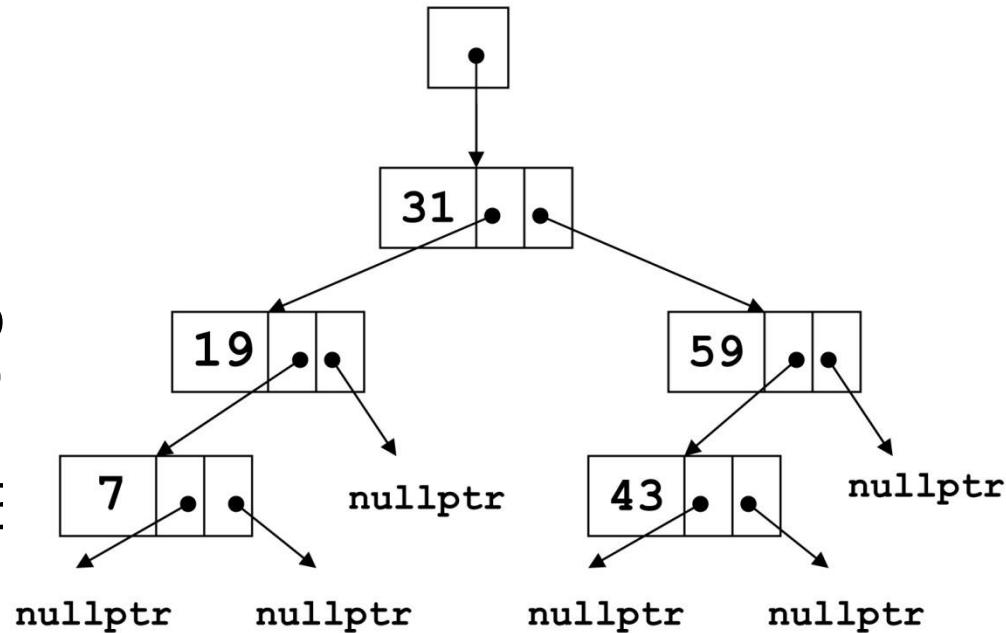
- 1) Start at root node
- 2) Examine node data:
  - a) Is it desired value?  
Done
  - b) Else, is desired data < node data? Repeat step 2 with left subtree
  - c) Else, is desired data > node data? Repeat step 2 with right subtree
- 3) Continue until desired value found or **NULL** pointer reached



# Searching in a Binary Tree 2 of 2

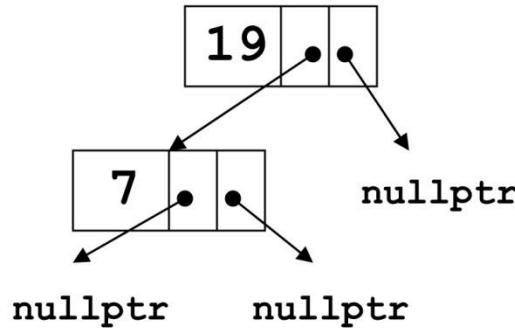
To locate the node containing 43,

1. Examine the root node (31)
2. Since  $43 > 31$ , examine the right child of the node containing 31, (59)
3. Since  $43 < 59$ , examine the left child of the node containing 59 (43)
4. The node containing 43 has been found

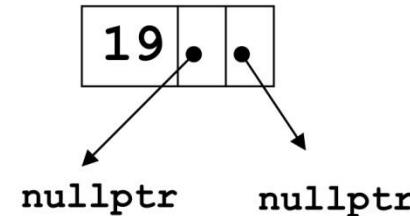


# Deleting a Node from a Binary Tree – Leaf Node

If node to be deleted is a leaf node, replace parent node's pointer to it with `nullptr`, then delete the node



**Deleting node with 7 – before deletion**

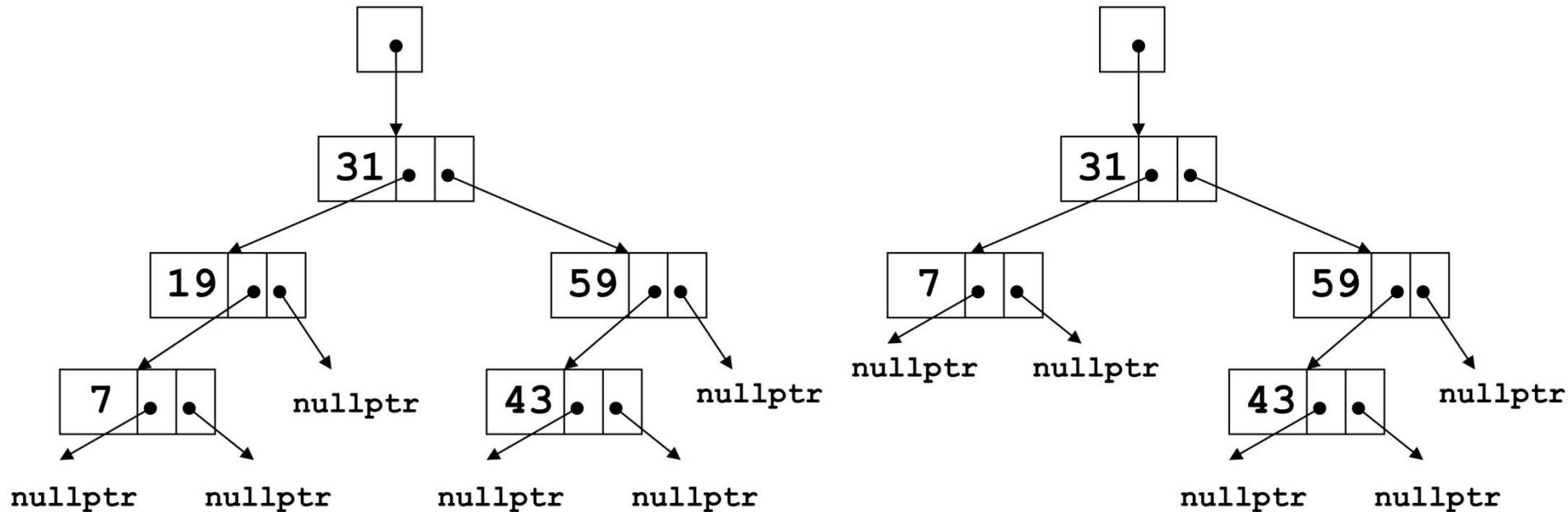


**Deleting node with 7 – after deletion**

# Deleting a Node from a Binary Tree – One Child 1 of 2

If the node to be deleted has one child node, adjust the pointers so that parent of the node to be deleted points to child of the node to be deleted, then delete the node

# Deleting a Node from a Binary Tree – One Child 2 of 2



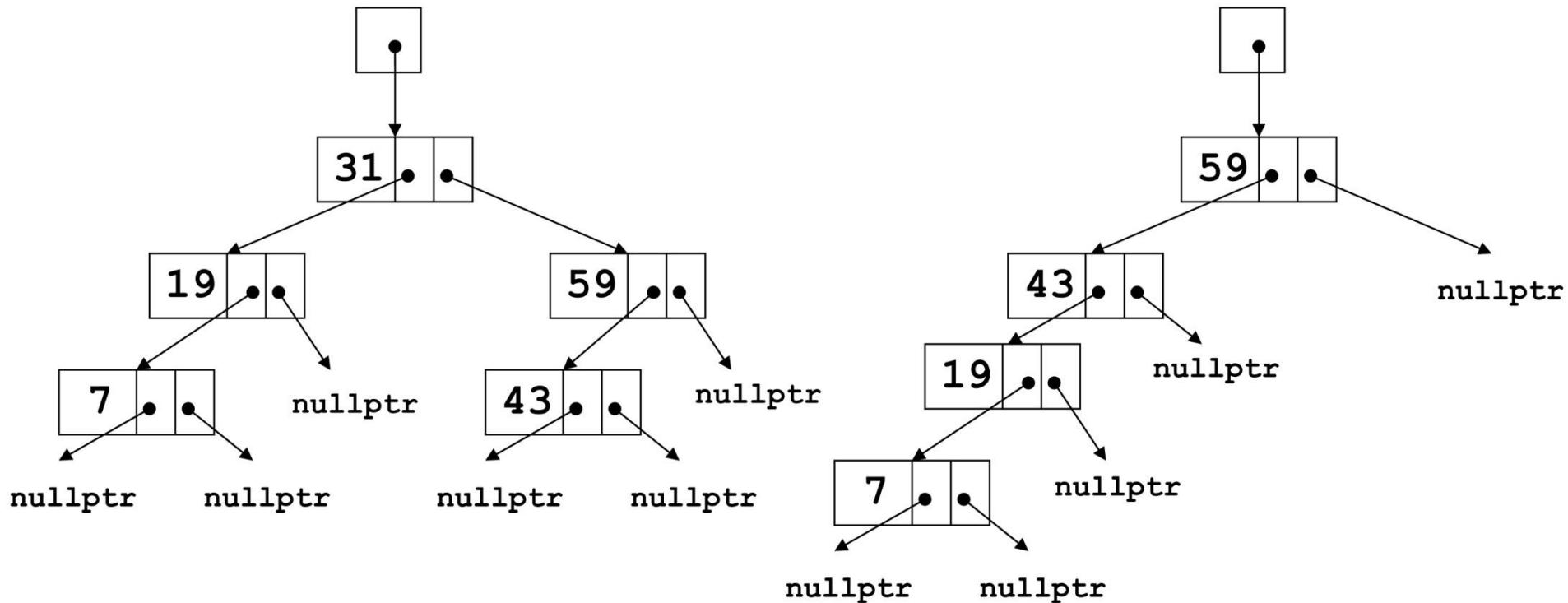
**Deleting node containing 19 – before deletion**

**Deleting node containing 19 – after deletion**

# Deleting a Node from a Binary Tree – Two Children 1 of 2

- If node to be deleted has left and right children,
  - ‘Promote’ one child to take the place of the deleted node
  - Locate correct position for other child in subtree of promoted child
- Convention in the text: “attach” the right subtree to its parent, then position the left subtree at the appropriate point in the right subtree.
- Care must be taken when re-attaching the left subtree to maintain the search property of a binary search tree.

# Deleting a Node from a Binary Tree – Two Children 2 of 2



**Deleting node with  
31 – before deletion**

**Deleting node with  
31 – after deletion**

## 20.3 Template Considerations for Binary Search Trees

- A binary tree can be implemented as a template, allowing flexibility in determining the type of data stored
- The implementation must support relational operators `>`, `<`, and `==` to allow for the comparison of nodes

# Copyright

