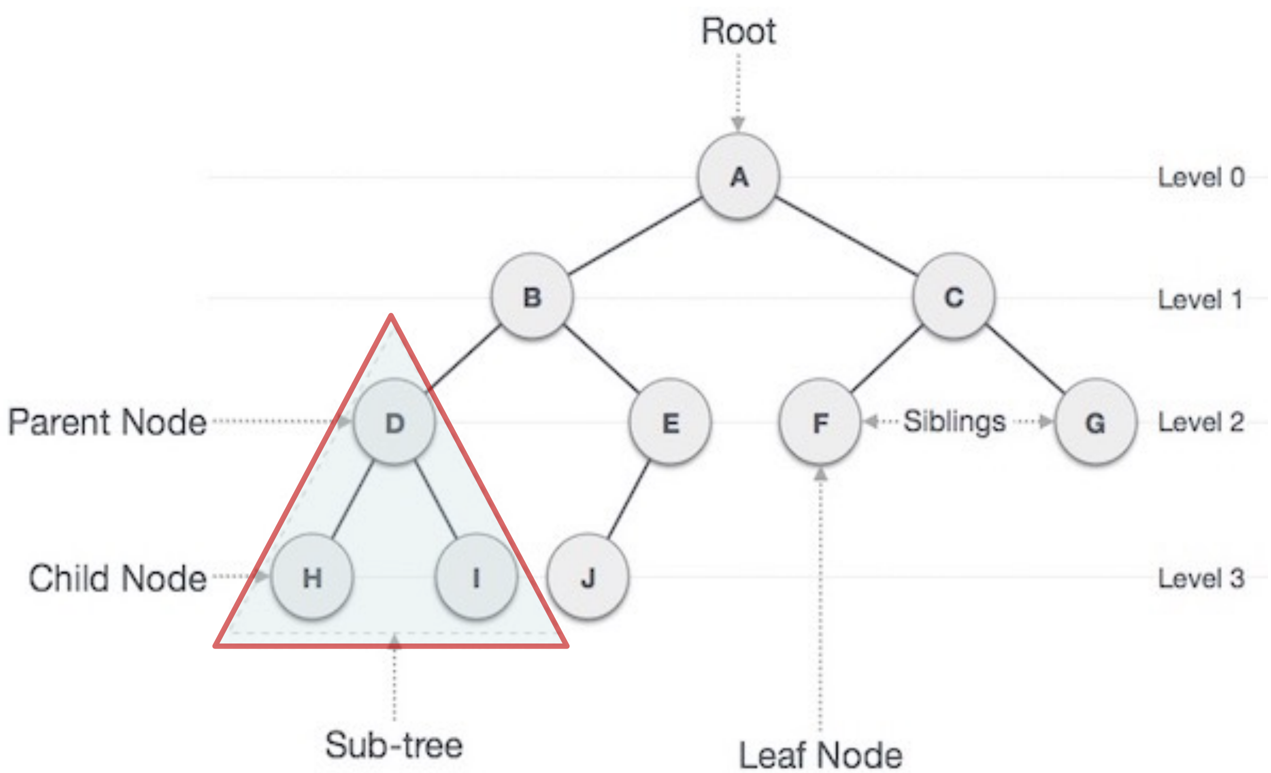
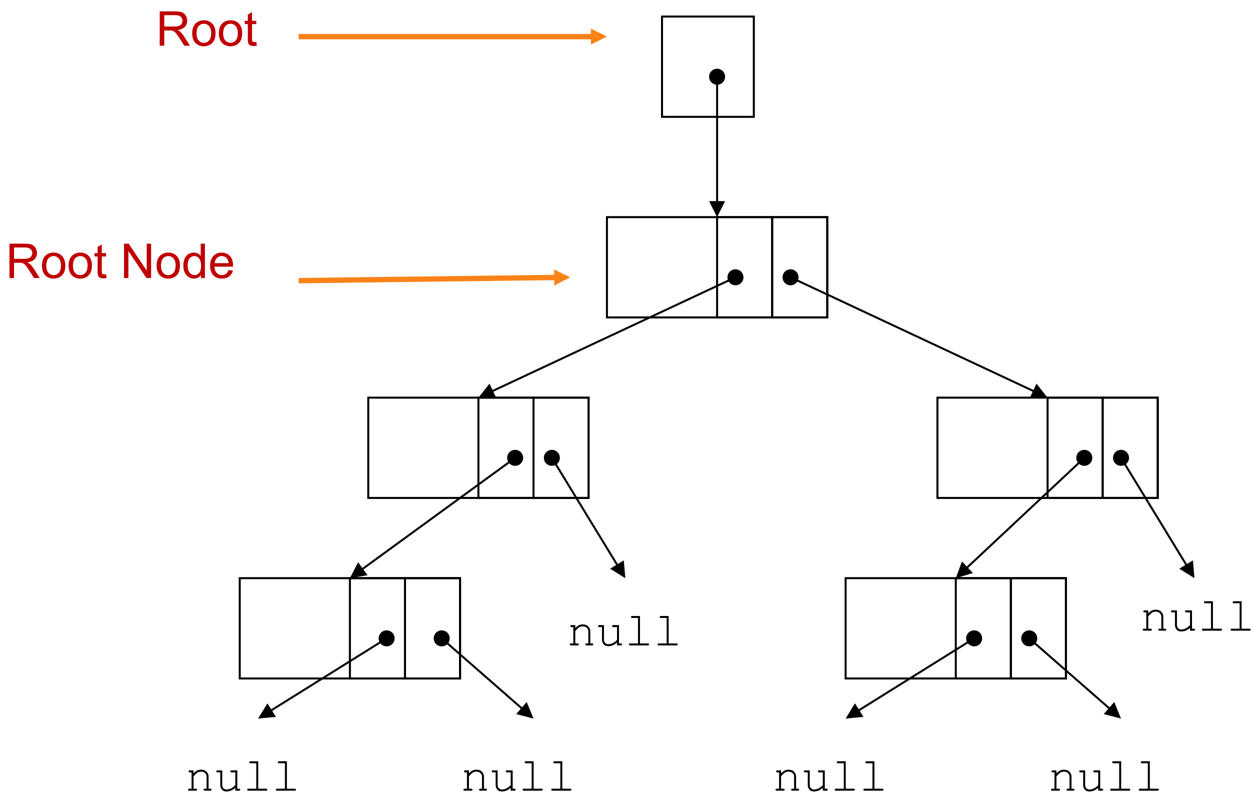
Binary Trees

- Binary tree: a nonlinear linked list in which each node may point to 0, 1, or two other nodes
- Each node contains one or more data fields and two pointers



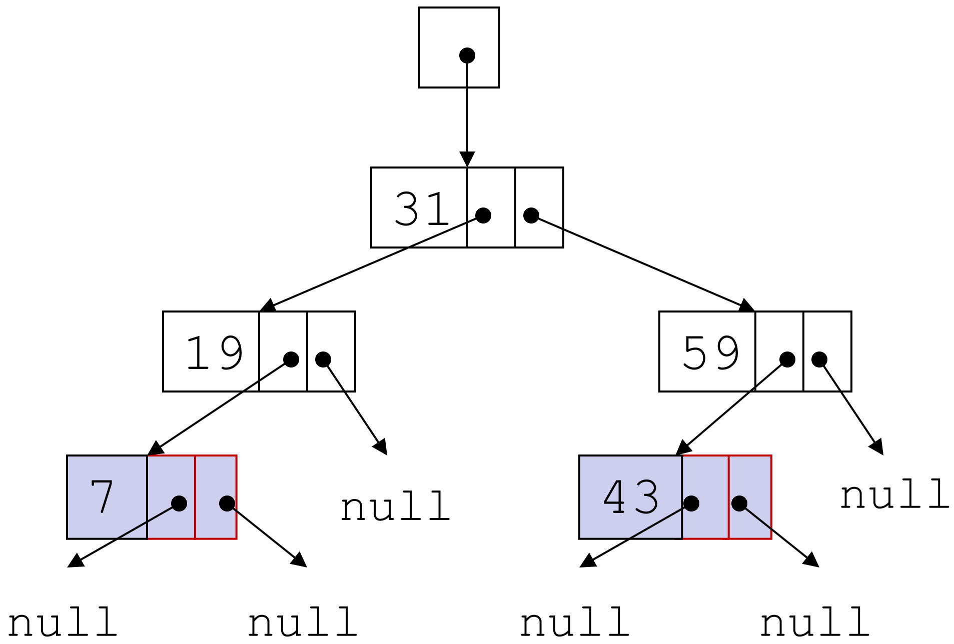
Binary Tree Terminology

- Tree pointer: like a head pointer for a linked list, it points to the first node in the binary tree
- Root node: the node at the top of the tree



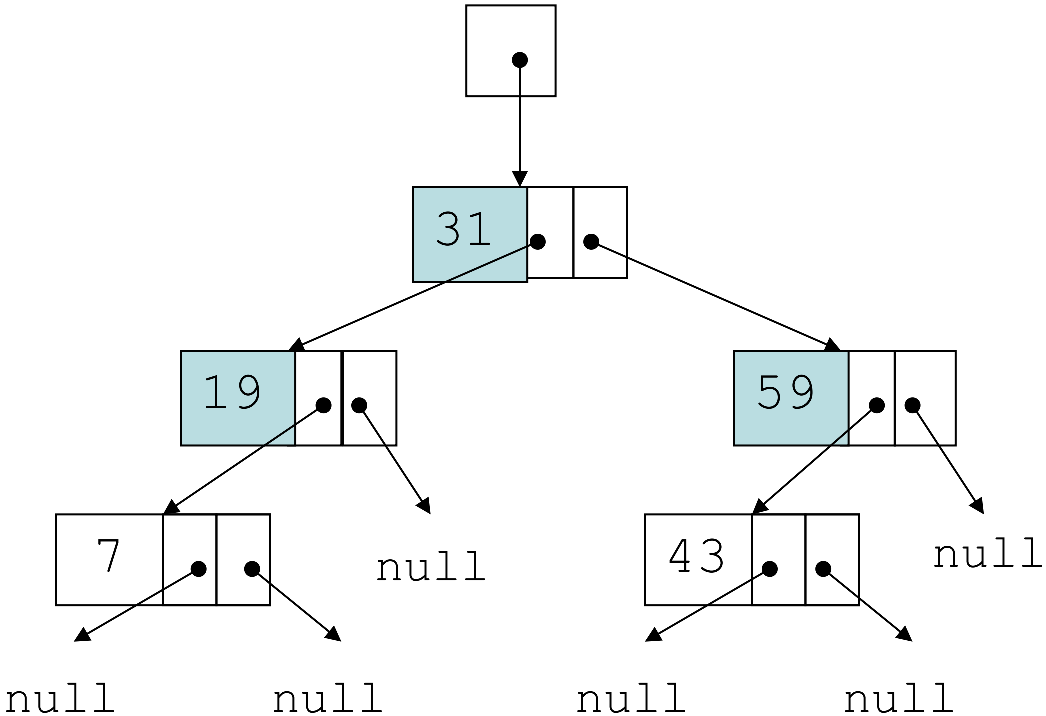
Binary Tree Terminology

- Leaf nodes: nodes that have no children
- The nodes containing 7 and 43 are leaf nodes



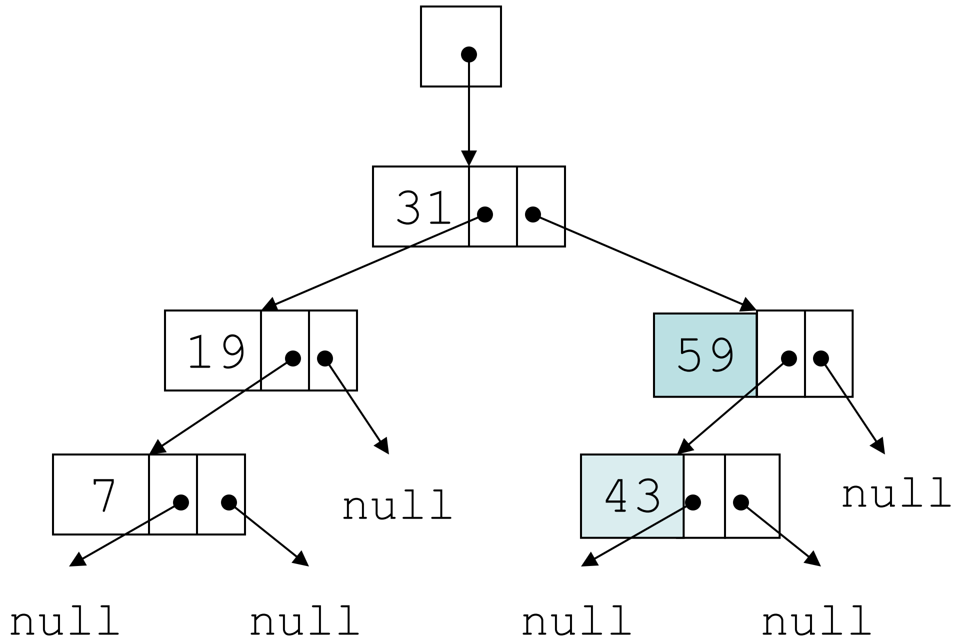
Binary Tree Terminology

- Child nodes, children: nodes below a given node
- The children of the node containing 31 are the nodes containing 19 and 59



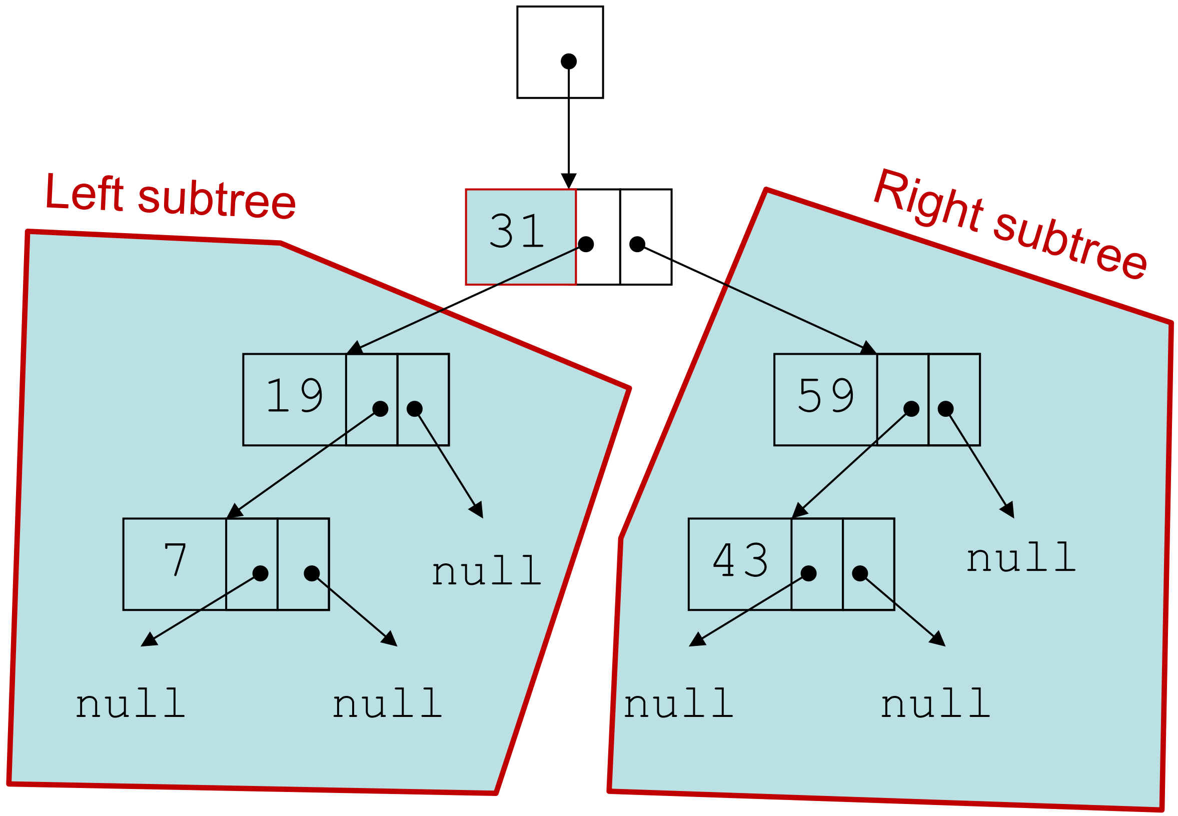
Binary Tree Terminology

- Parent node: node above a given node
- The parent of the node containing 43 is the node containing 59



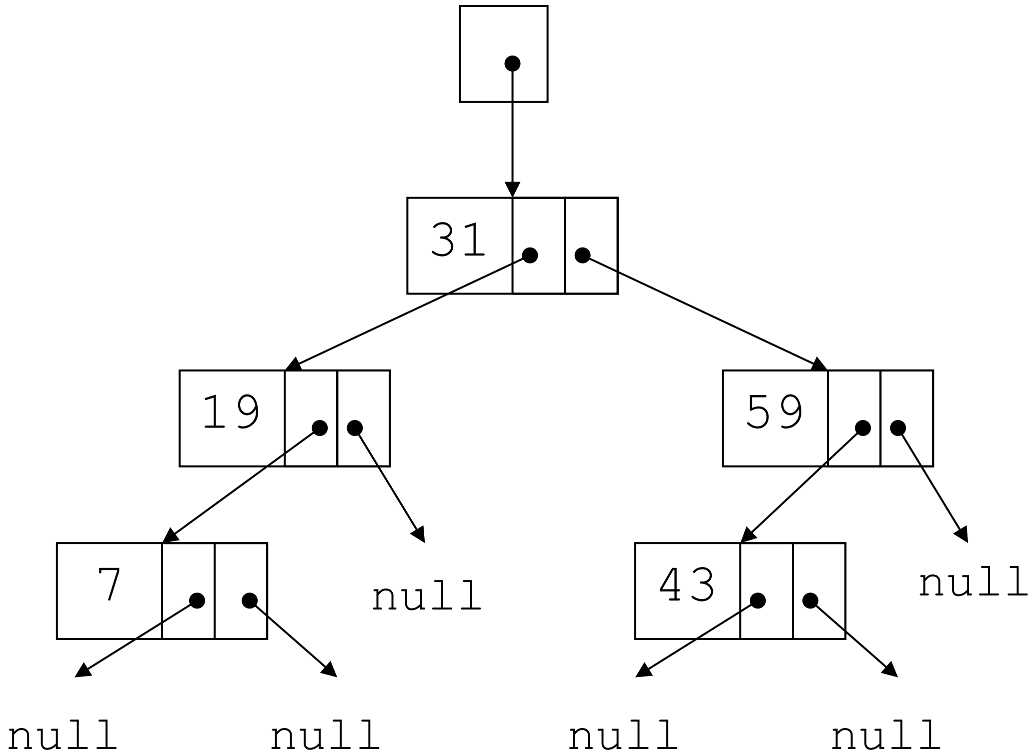
Binary Tree Terminology

- Subtree: the portion of a tree from a node down to the leaves
- The nodes containing 19 and 7 are the left subtree of the node containing 31



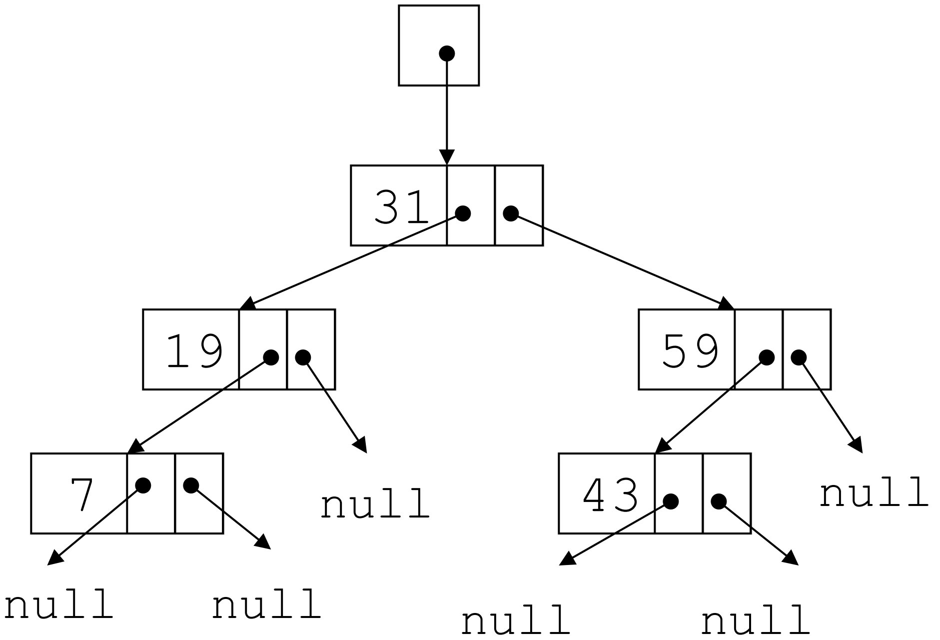
Binary Search Trees

- Binary search tree: data organized in a binary tree to simplify searches
- Left subtree of a node contains data values < the data in the node
- Right subtree of a node contains values >= the data in the node



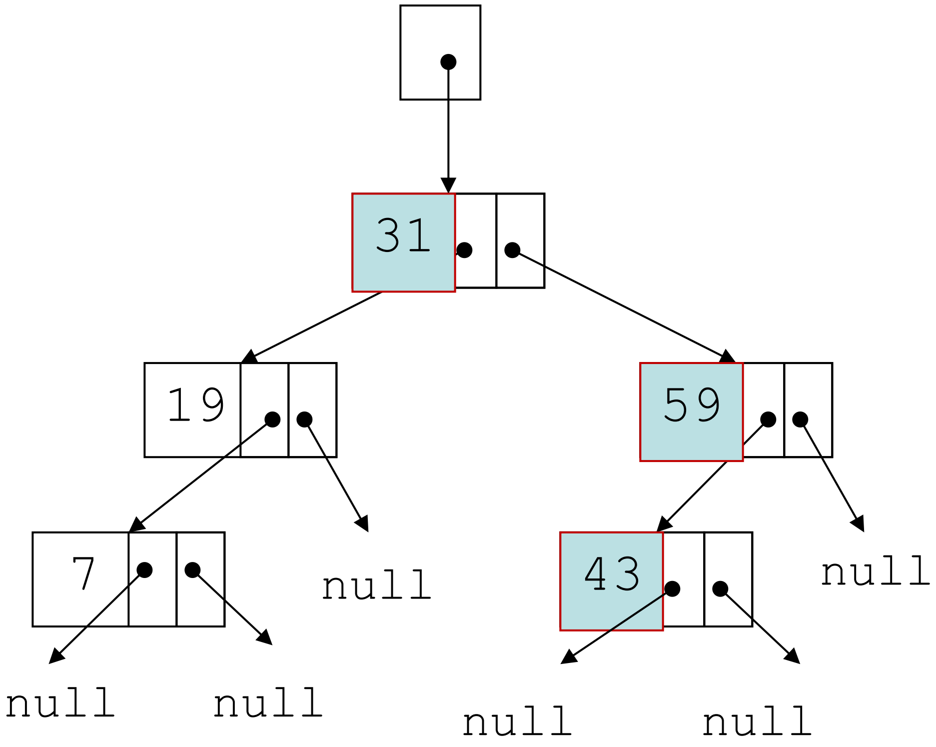
Searching a Binary Tree

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



To locate the node containing `43`,

- Examine the root node (`31`) first
- Since `43 > 31`, examine the right child of the node containing `31`, (`59`)
- Since `43 < 59`, examine the left child of the node containing `59`, (`43`)
- The node containing 43 has been found



Binary Tree Operations

- **Create** a binary search tree – organize data into 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 maintain binary tree

- A node in a binary tree is like a node in a linked list, with two node pointer fields:

```
template <class T>
struct Node {
    T info;
    Node *left;
    Node *right;
};
```

Create a Binary Tree

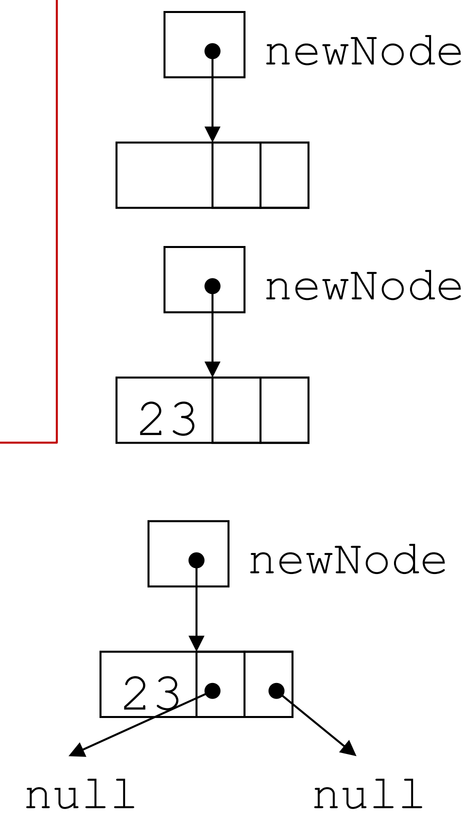
```
template<class T>
struct Node {
    T info;
    Node *left;
    Node *right;

    Node() = delete;
    Node(T v) {
        info = v;
        left = nullptr;
        right = nullptr);
     }
};
```

```
template <class T>
class BinaryTree
{
    Node<T> *root;

public:
    BinaryTree()=delete;
    BinaryTree(T v) {
      root = new Node<T>(v);
    }
};
```

```
int main()  {
    BinaryTree<int> btree(23);
    return 0;
}
```



Inserting a Node into a Binary Search Tree (BST)

If root is NULL
  then create root node
return

If root exists then
  compare the data with node.info

  while until insertion position is located

    If data is greater than node.info
      goto right subtree
    else
      goto left subtree

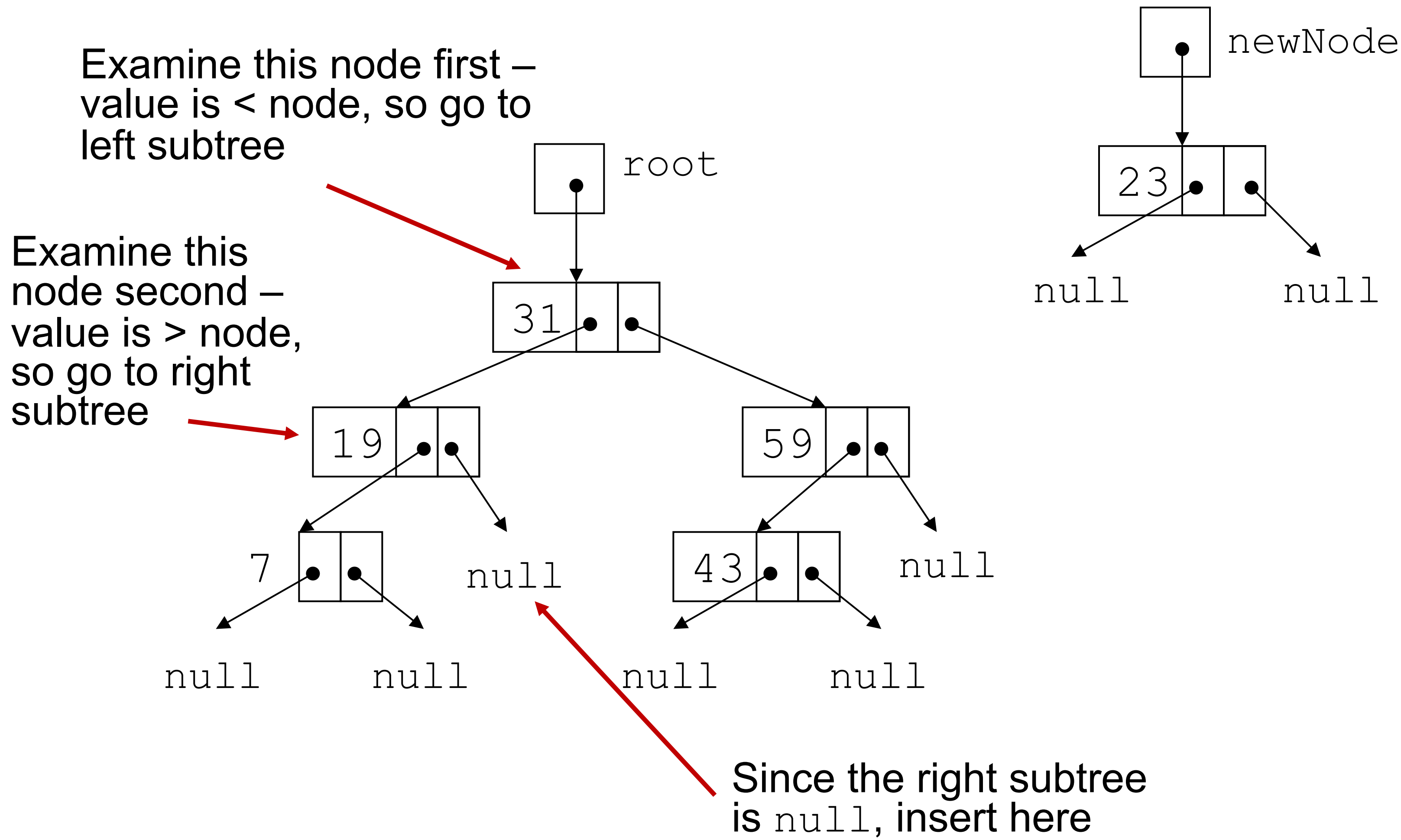
  endwhile

  insert info

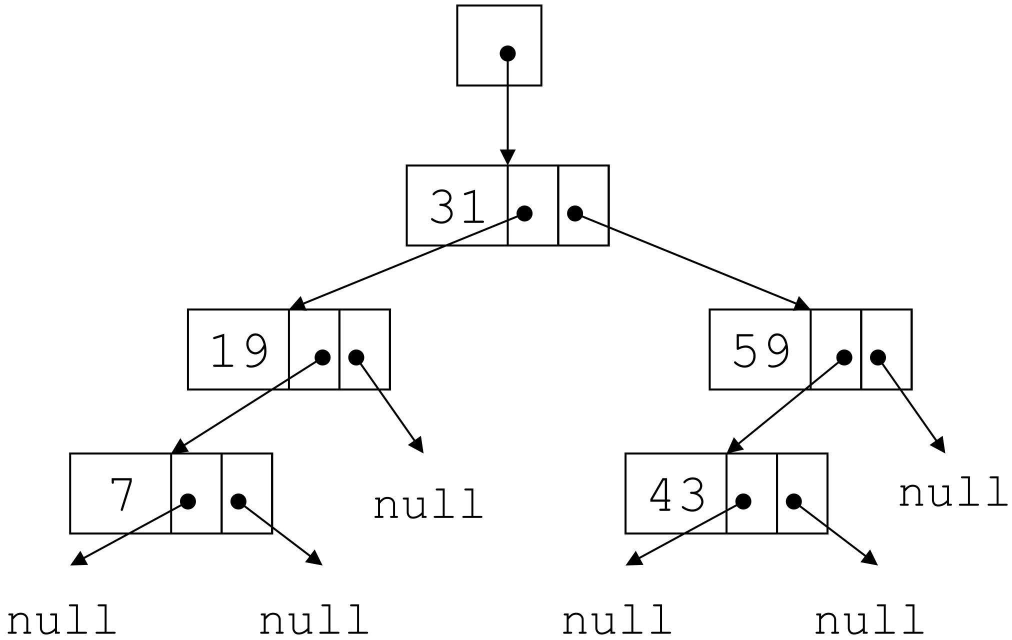
end If

```
void insert(T v) {
  Node<T> *temp = new Node<T>(v);
  Node<T> *cur;
  Node<T> *par;
  if(root == nullptr) {
    root = temp;
 } else {
    cur = root;
    par  = nullptr;
    while(1) {
      par = cur;
      if(v < par->info) {
        cur = cur->left;
        if(cur == nullptr) {
            par->left = temp;
            return;
        }
      } else {
        cur = cur->right;
        if(cur == nullptr) {
            par->right = temp;
            return;
        }
      }
    }
  }
}
```

Inserting a Node into a Binary Search Tree (BST)



Traversing a Binary Search Tree



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

Inorder Traversing a Binary Search Tree

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

```
void InOrder(Node<T> *ptr)
{
    if (ptr == nullptr) return;

    InOrder(ptr->left);

    cout << ptr->info << ":";

    InOrder(ptr->right);
}
```

```
void InOrder()
{
  InOrder(root);
}
```

Preorder Traversing a Binary Search Tree

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

```
void PreOrder(Node<T> *ptr)
{
    if (ptr == nullptr)  return;

    cout << ptr->info << ":";

    PreOrder(ptr->left);
    PreOrder(ptr->right);
}
```

```
void PreOrder()
{
    PreOrder(root);
}
```

Postorder Traversing a Binary Search Tree

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

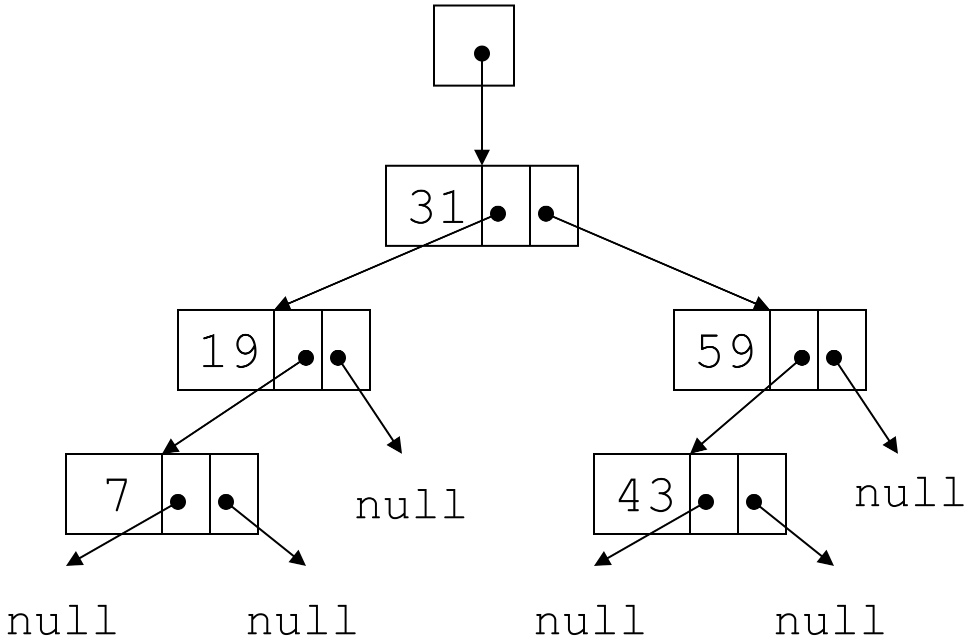
```
void PostOrder(Node<T> *ptr)
{
    if (ptr == nullptr) return;

    PostOrder(ptr->left);
    PostOrder(ptr->right);

    cout << ptr->info << ":";
}
```

```
void PostOrder() {
    PostOrder(root);
}
```

- Start at root node, traverse the tree looking for value
- Stop when value found or null pointer detected
- Can be implemented as a bool function



Search for `43`? return `true`
Search for `17`? return `false`

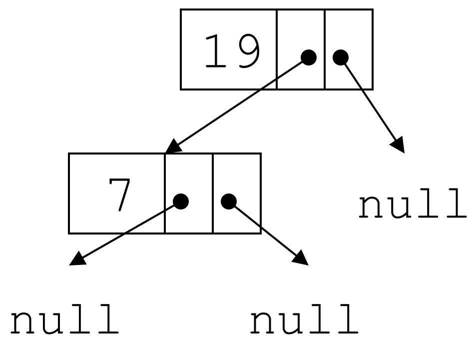
Searching a Binary Search Tree

```
If root.info is equal to search.info
   return root
else
   while info not found
      If info is greater than node.info
         goto right subtree
      else
         goto left subtree
      If info found
         return node
   endwhile
   return info not found
end if
```

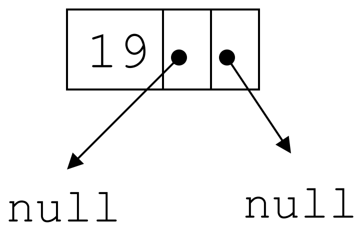
```
Node<T> *search(T v) {
  Node<T> *cur = root;
  while(cur->info != v) {
      if(cur->info > v) {
        cur = cur->left;
      }
      else
      {
         cur= cur->right;
      }
      if(cur == nullptr) {
        return nullptr;
     }
  }
  return cur;
}
```

Deleting a Node from a Binary Search Tree – Leaf Node

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



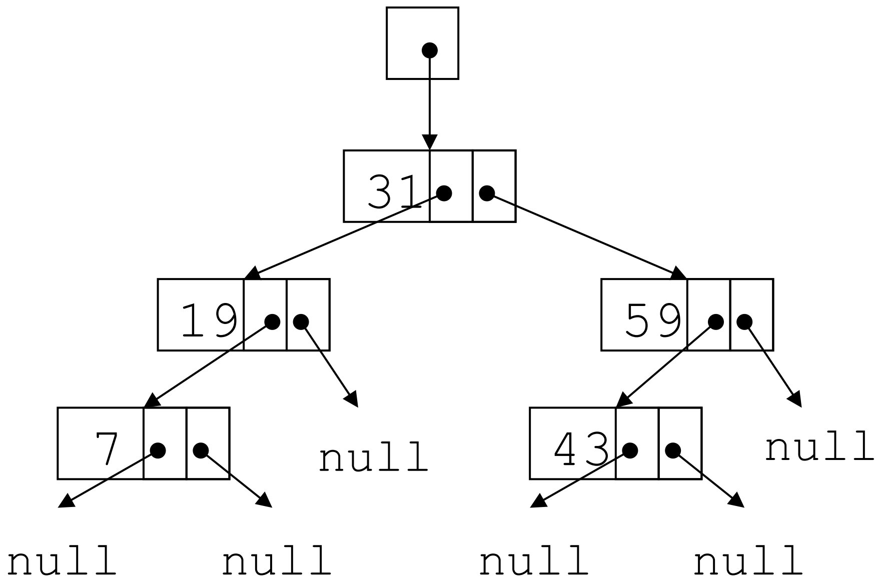
Deleting node with 7
– before deletion



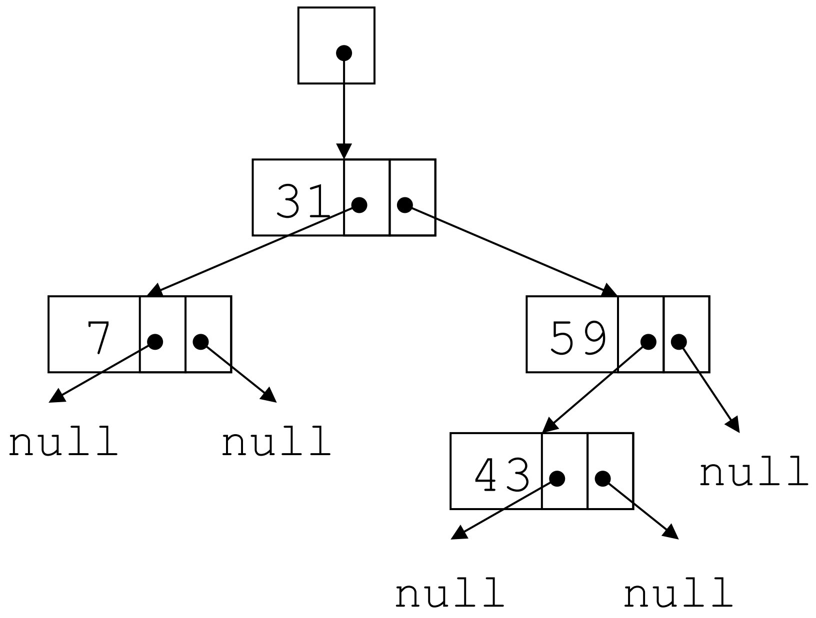
Deleting node with 7
– after deletion

Deleting a Node from a Binary Search Tree – One Child

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



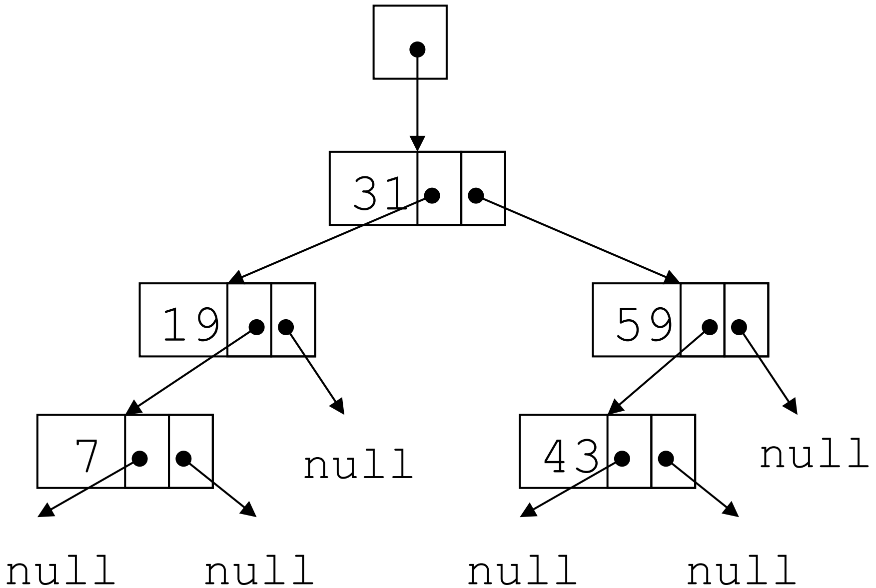
Deleting node with 19
– before deletion



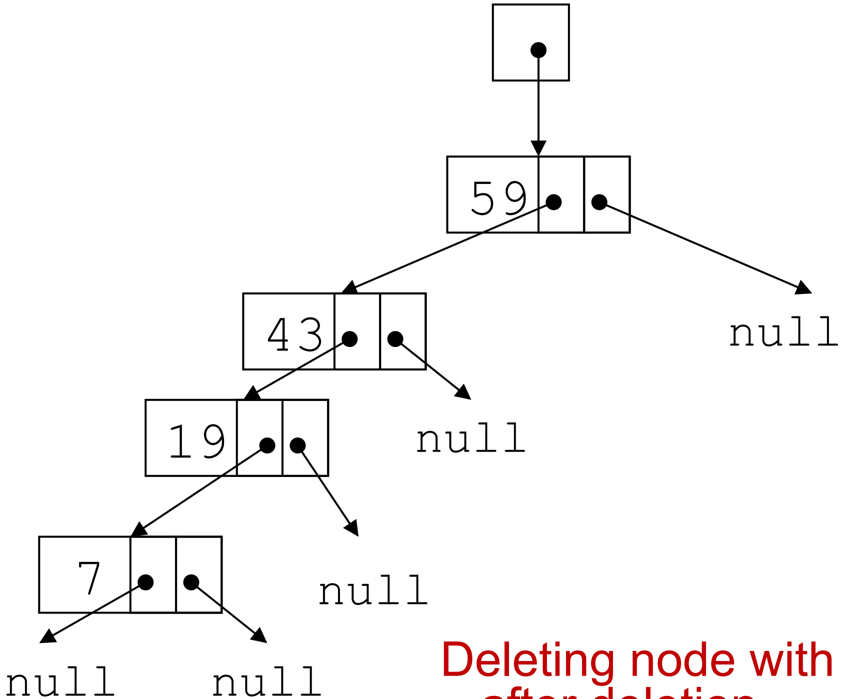
Deleting node with 19
– after deletion

Deleting a Node from a Binary Search Tree – Two Child

- 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 text: promote the right child, position left subtree underneath



Deleting node with 31 – before deletion



Deleting node with 31 – after deletion

Deleting a Node from a Binary Search Tree – Two Child

```
Node<T> * minValueNode
 (Node<T> *node)
{
    Node<T> *cur = node;
    while (cur->left != nullptr)
      cur = cur->left;
    return cur;
}
```

```
bool idelete (T v)
{
     if (idelete(root,v)!= nullptr)
        return true;
    else
        return false;
    }
```

```
Node<T> *idelete (Node<T> *ptr, T key)  {
  if(ptr == nullptr) return ptr;
  if(key < ptr->info)
      ptr->left = idelete(ptr->left, key);
  else
    if(key > ptr->info)
       ptr->right=idelete(ptr->right, key);
   else {
      if(ptr->left == nullptr) {
         Node<T> *temp = ptr->right;
         delete (ptr);
         return temp;
      } else if(ptr->right == nullptr) {
          Node<T> *temp = ptr->left;
          delete(ptr);
          return temp;
      }
      Node<T> *temp=minValueNode(ptr->right);
      ptr->info=temp->info;
      ptr->right=idelete(ptr->right,temp->info);
    }
    return ptr;
}
```

Template Considerations for BST

- Binary tree can be implemented as a template, allowing flexibility in determining type of data stored
- Implementation must support relational operators >, <, and == to allow comparison of nodes