CIT-237

Chapter 21: Binary Trees

December 2, 2019

Reminders / Announcement

- By popular request, Quiz 7 will be held at the start of class on

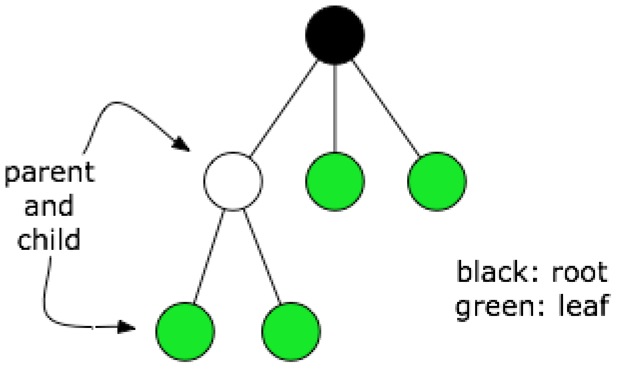
  Wednesday, December 4.
- The material covered on Quiz 7 will be:
  - The Lectures of November 18 and 20.
  - Chapters 17 and 18.
- Quiz 8 will be on Wednesday, December 11.
- Project 3: EXTENDED due date is December 9.
- Our last day of class is Monday, December 16.
  - In addition to a lecture, we will be demonstrating Lab solutions during that class.

Tree Data Structures

- We have seen arrays, vectors, and linked lists.
  - All of these data structures are linear in nature
- Sometimes we need to organize data as a "tree":

tree (data structure): A data structure accessed beginning at the *root* node. Each *node* is either a *leaf* or an *internal node*. An internal node has one or more *child* nodes and is called the *parent* of its child nodes. All children of the same node are *siblings*. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom.

(Source: https://xlinux.nist.gov/dads/HTML/tree.html)



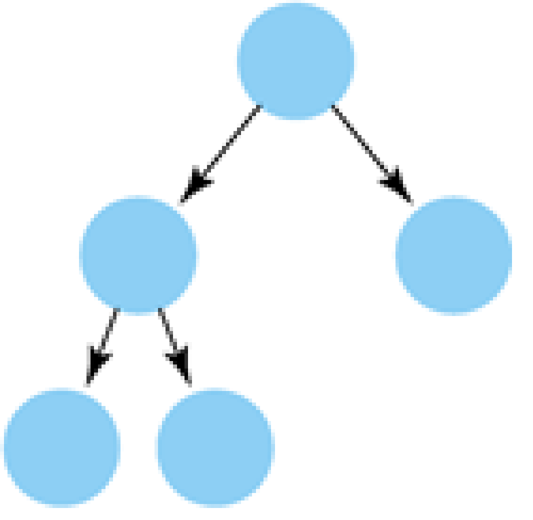
Binary Tree

- If each node in a tree is assumed to have no more that <u>two</u> child nodes, it is called a *binary tree*:

binary tree (data structure): A *tree* with at most two *children* for each *node*.

Formal Definition: A binary tree either is empty (no nodes), or has a *root* node, a left binary tree, and a right binary tree.

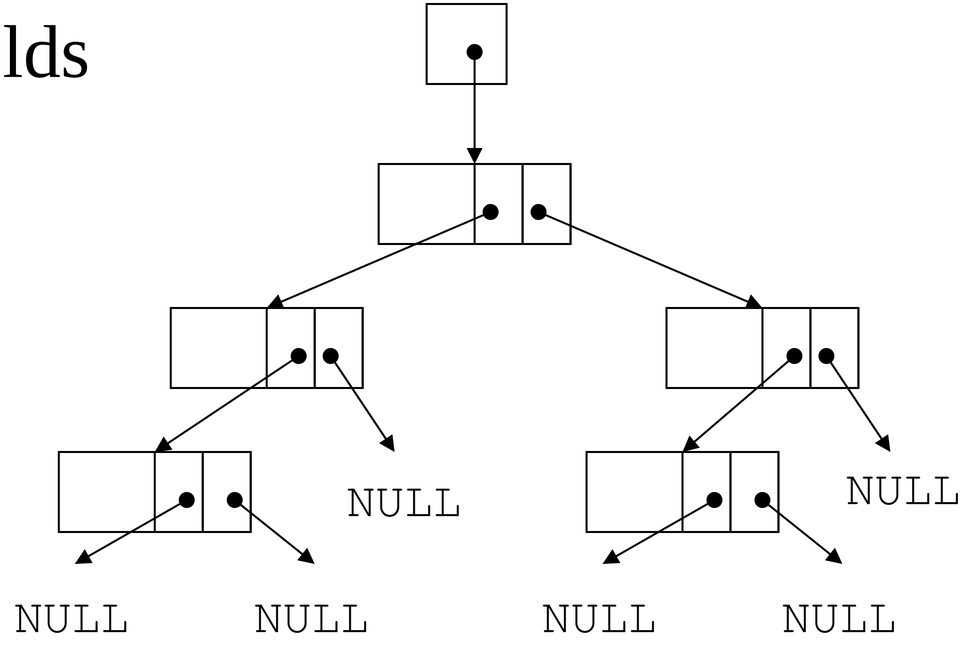
(Source: https://xlinux.nist.gov/dads/HTML/binarytree.html)



- Sometimes binary trees can be represented using an array, but first we are going to focus on *linked-node* implementations of binary trees.

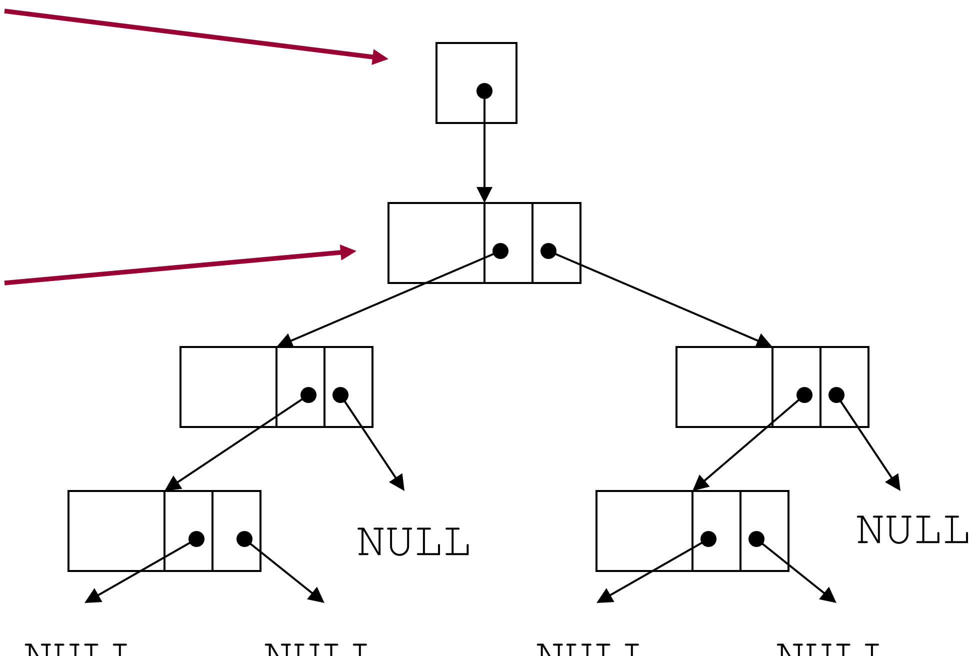
Linked-Node Binary Trees

- Binary tree: a nonlinear linked data structure in which each node may link to 0, 1, or 2 other nodes
- Each node contains:
  - one or more data fields
  - two pointers



Binary Tree Terminology (1)

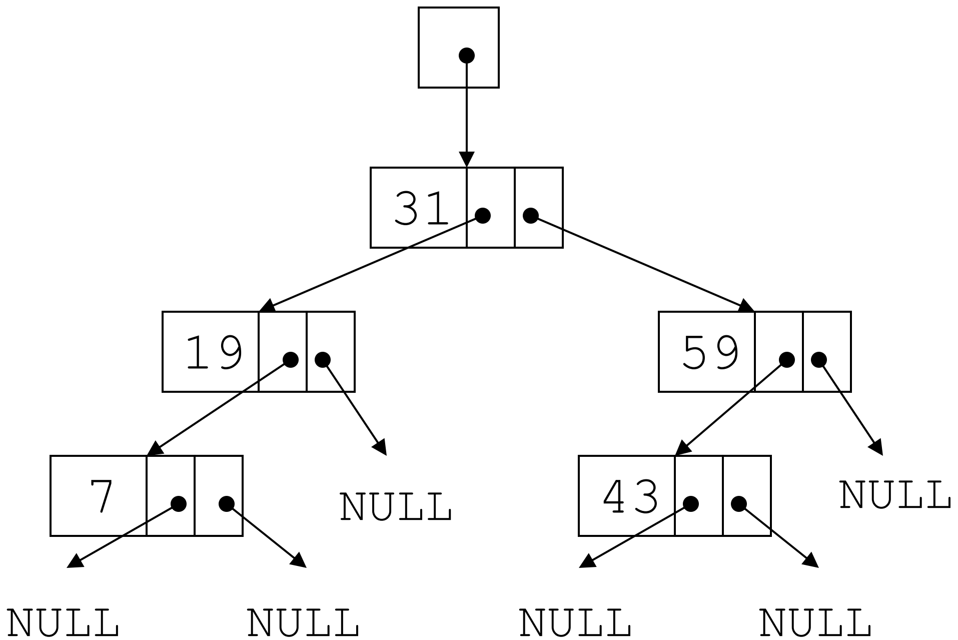
- Tree pointer: like a head pointer for a linked list, it points to the first node in the binary tree
- Root node: the node at the top of the tree



Binary Tree Terminology (2)

- Child nodes, children: nodes below a given node

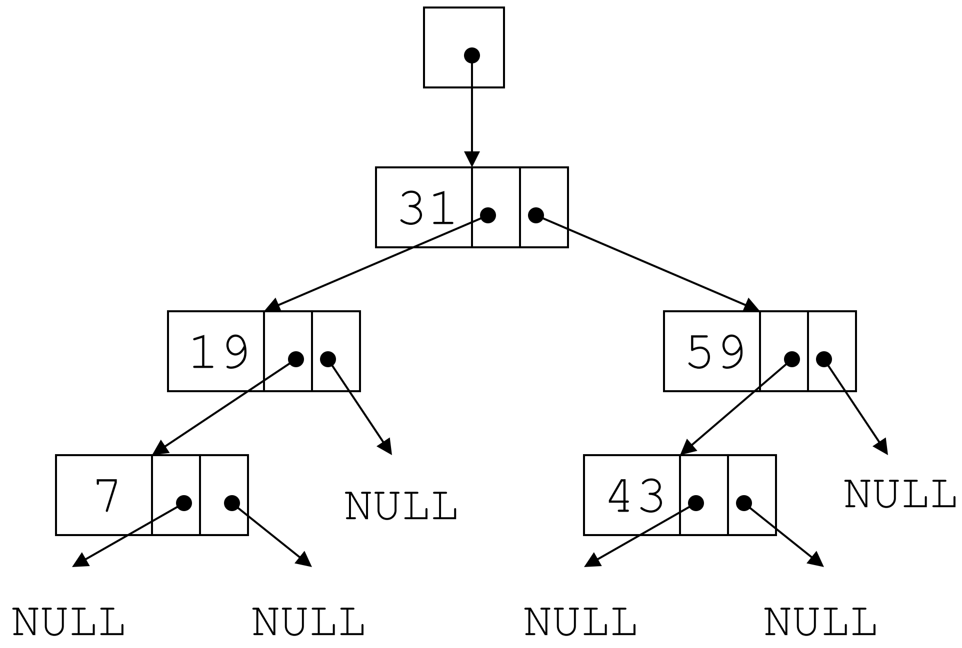
  The children of the node containing `31` are the nodes containing `19` and `59`



Binary Tree Terminology (3)

- Parent node: node above a given node

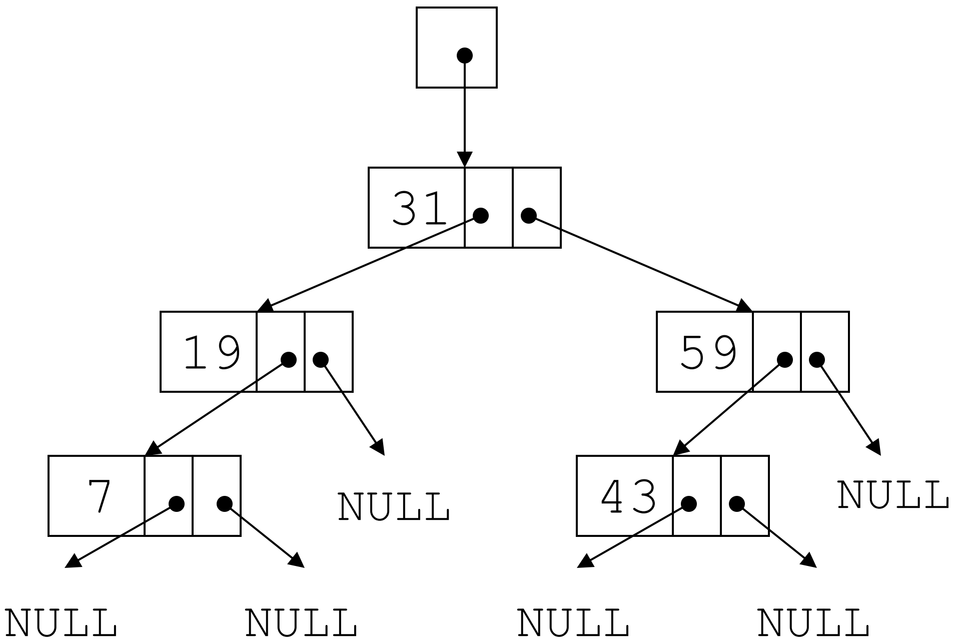
  The parent of the node containing `43` is the node containing `59`



Binary Tree Terminology (4)

- <u>Leaf nodes</u>: nodes that have no children

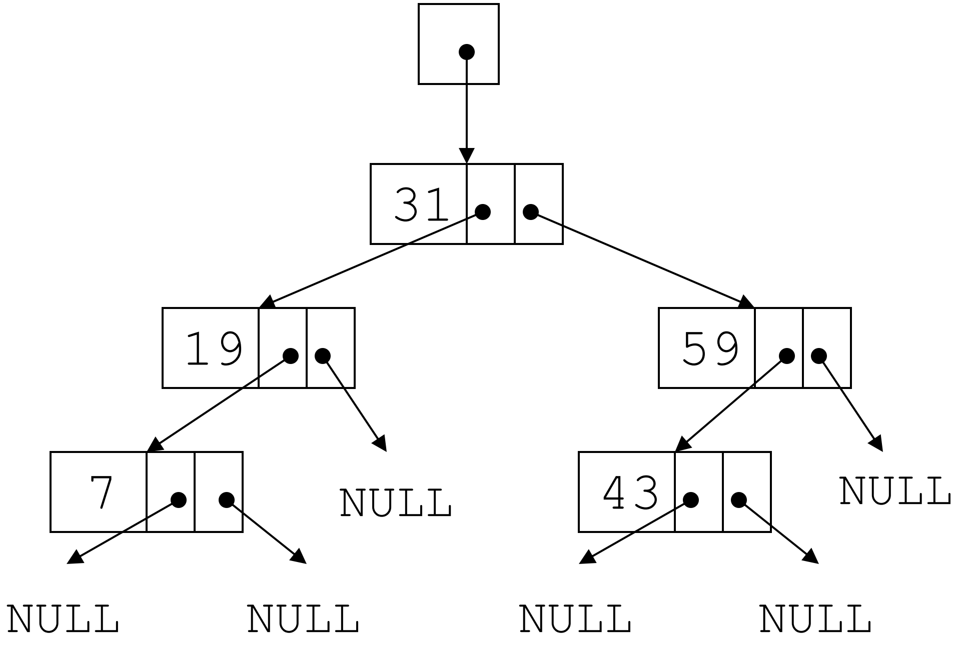
  The nodes containing `7` and `43` are leaf nodes



Binary Tree Terminology (5)

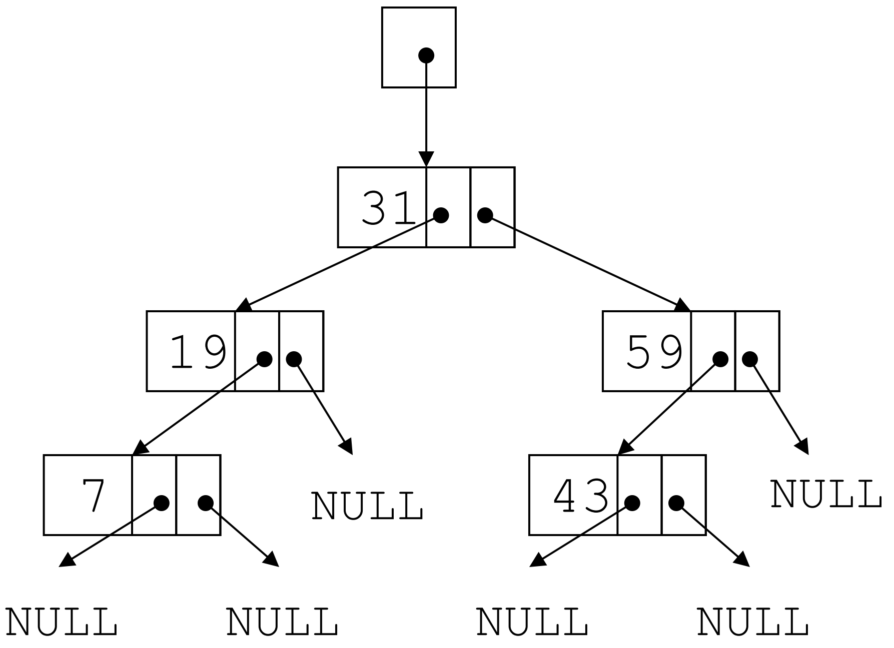
- <u>Subtree</u>: the portion of a tree from a node down to the leaves

The nodes containing `19` and `7` are the left subtree of the node containing `31`



Binary Tree Terminology (6)

- Depth: the "depth" of a node **`x`** in tree **`T`** is the length of the path from the root node to node **`x`**.
  - The node containing "19" has depth =1.
  - The node containing "43" has depth = 2.
- Height: The "height" of a tree **`T`** is the largest depth of any node in **`T`**. (The height of this tree is 2.)



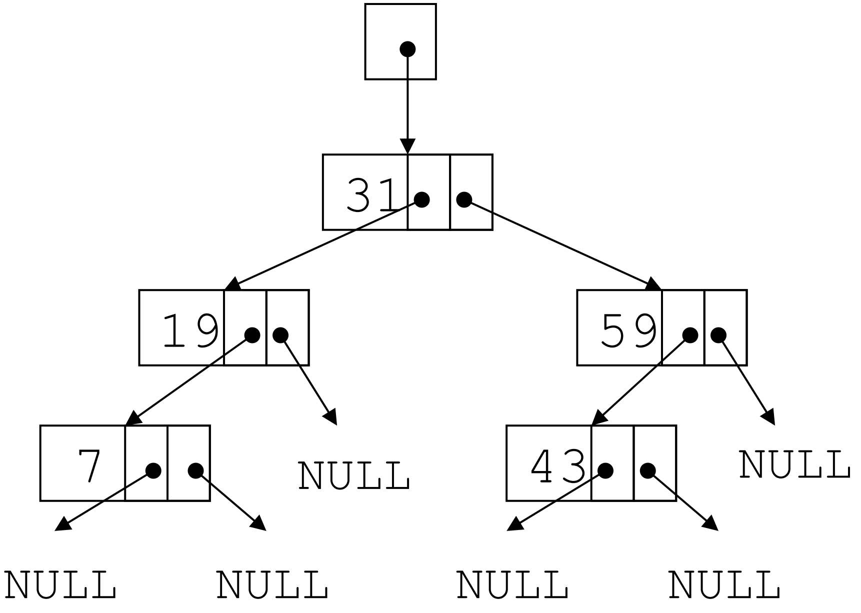
Binary Search Tree (1)

- One common use of a binary tree is as a *Binary Search Tree*:
  - Organized in a way to making searching the tree very efficient.

- In a *Binary Search Tree*:

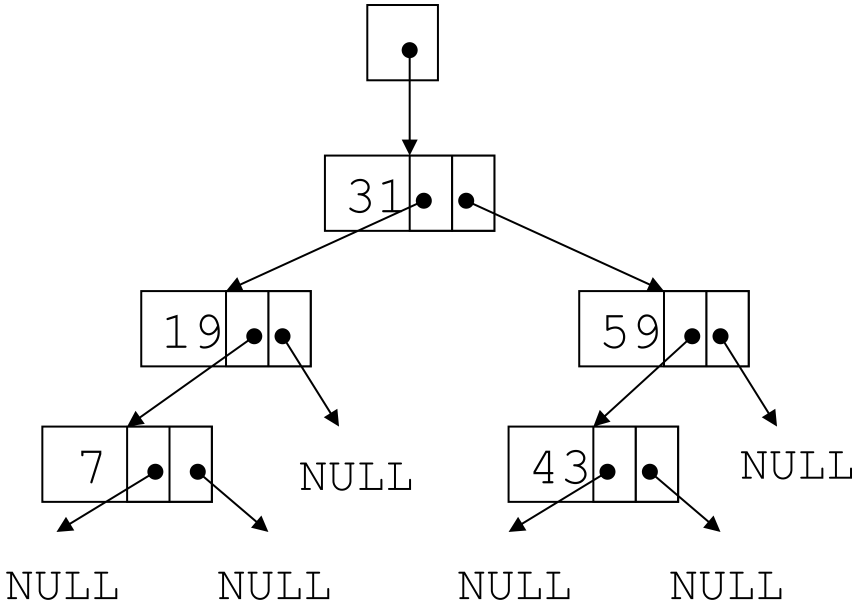
  The <u>left</u> subtree of some node **X** contains data values "less than" the data in node **X**.

  The <u>right</u> subtree of some node **X** contains data values "greater than" the data in node **X.**



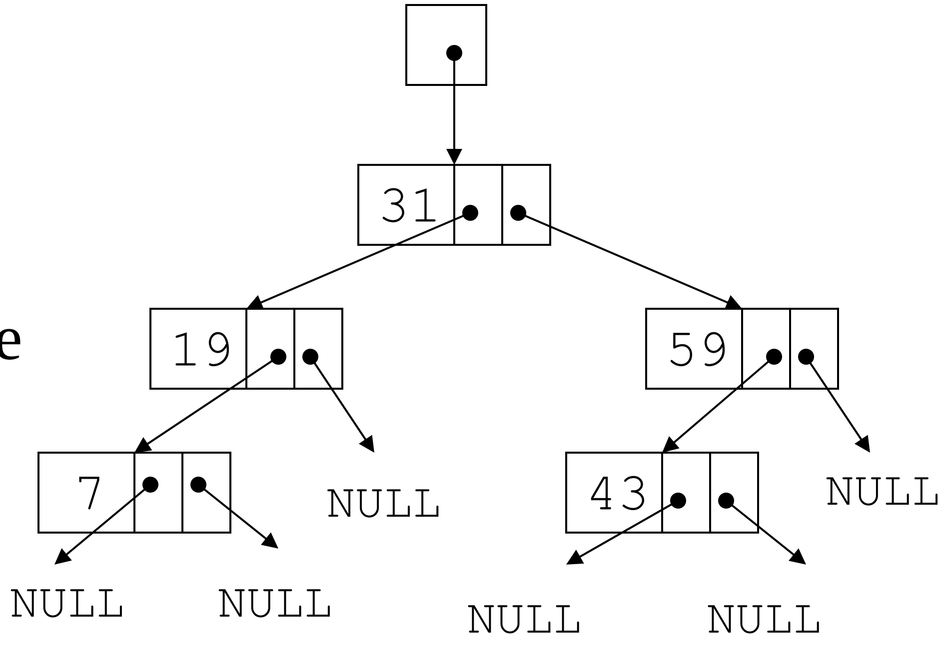
Binary Search Tree (2)

- The precise meaning of "less than" and "greater than" is obvious for numeric data, but in general depends on the nature of the data itself.
  - String data is usually compared by "alphabetical order".
  - A binary search tree of <u>objects</u> could compare those objects by whatever criterion is appropriate.
- For simplcity, we will use numeric examples.



Finding Data in a Binary Search Tree

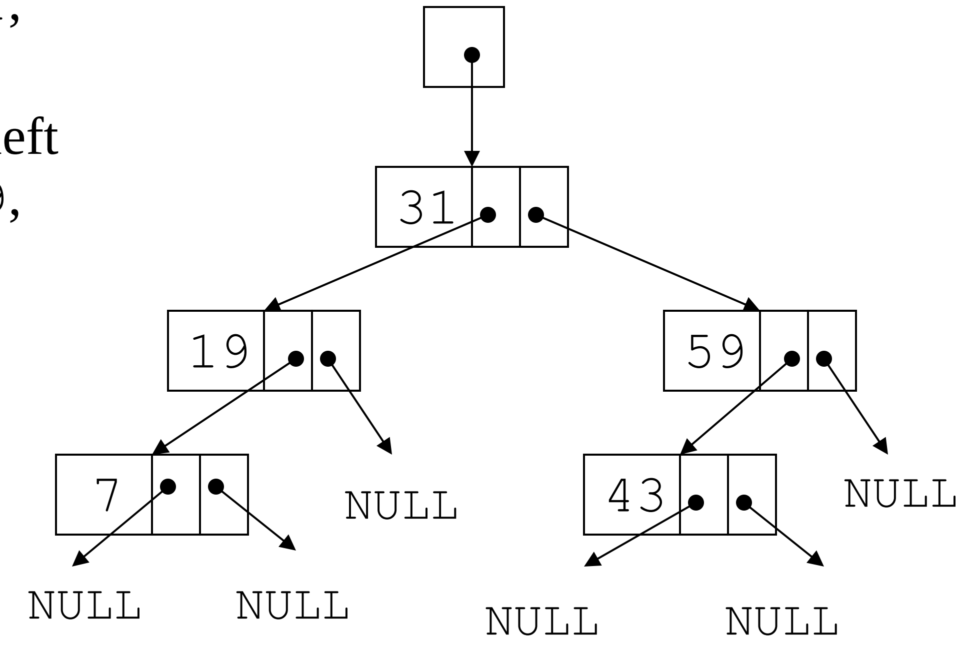
1) Start at root node
2) Examine node data:
    a) Is it desired value? Done
    b) Else, is desired data < node data? Repeat step 2 with <u>left</u> subtree
    c) Else, is desired data > node data? Repeat step 2 with <u>right</u> subtree
3) Continue until desired value found or `NULL` pointer reached



Binary Tree Search Example

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 Search Tree Operations

Considering the *Binary Search Tree* as an Abstract Data Type, we know that we must consider the operations required, in addition to the data representation:

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

Binary Search Tree Node

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

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

Creating a New Node

- Allocate memory for new node:

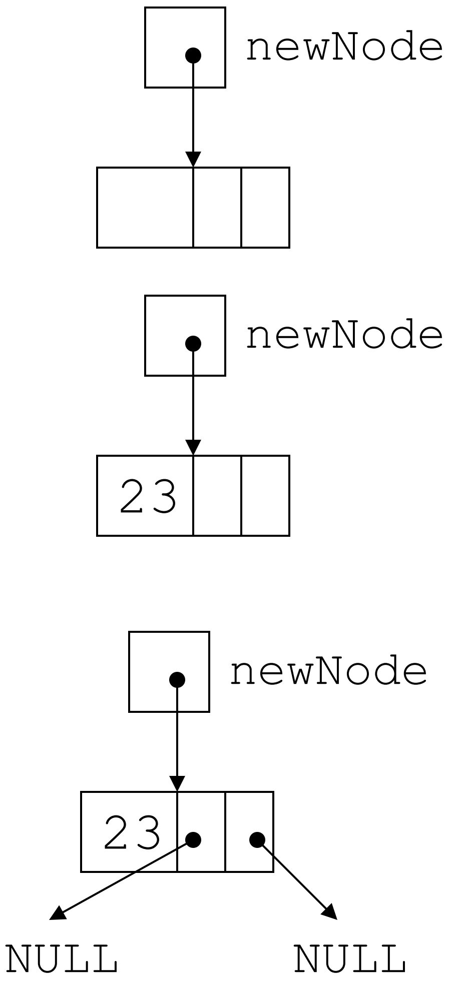
```
newNode = new TreeNode;
```

- Initialize the contents of the node:

```
newNode->value = num;
```

- Set the pointers to `NULL`:

```
newNode->Left
= newNode->Right
= NULL;
```



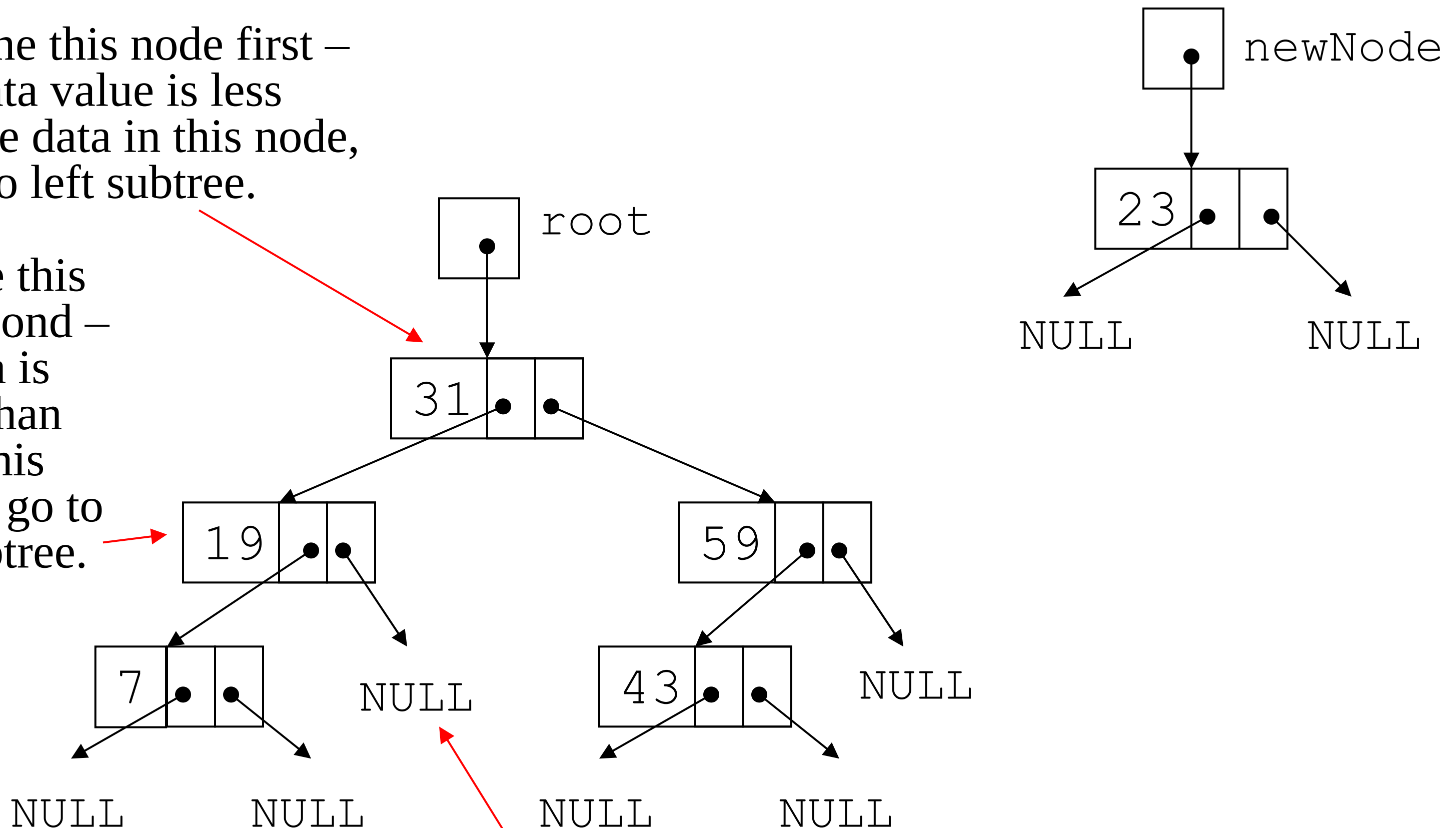
Inserting a Node in a Binary Search Tree

1) If tree is empty, insert the new node as the root node.
2) Otherwise, compare the data value in the new node with the data value in the "root" node, to determine whether the data value in the new node is **less than** or **greater than** the data in the root node.
3) Based on this comparison, choose to insert the new node into the **left** subtree or the **right** subtree.
4) Continue comparing and choosing **left** or **right** subtree until a **`NULL`** pointer is found.
5) Set this **`NULL`** pointer to point to new node.

Inserting a Node into a Binary Search Tree

Examine this node first – new data value is less than the data in this node, so go to left subtree.

Examine this node second – new data is greater than data in this node, so go to right subtree.



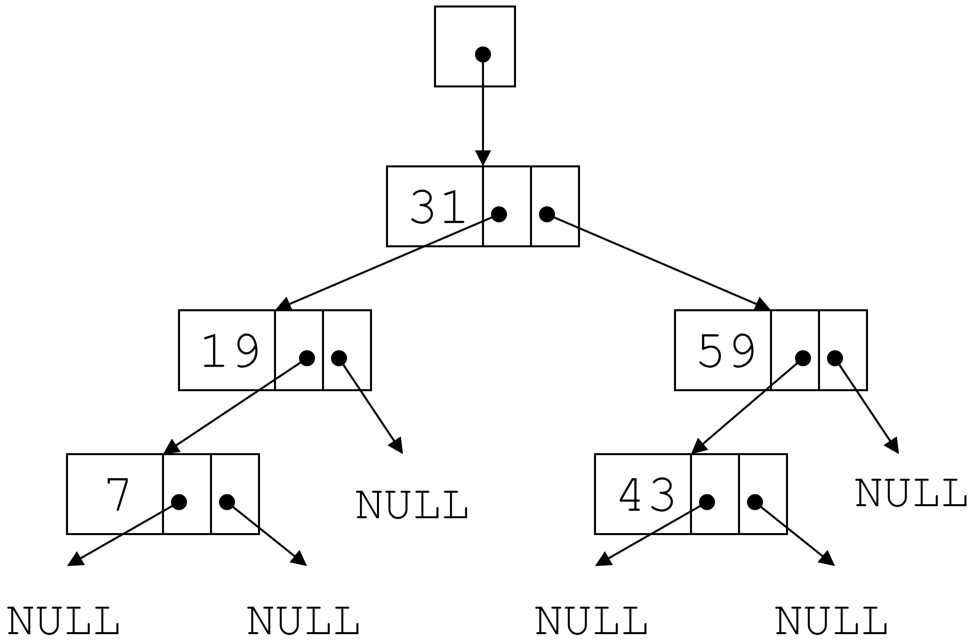
Since the right subtree is NULL, insert new node here.

Traversing a Binary Tree

Three popular *traversal methods* (<u>very</u> important):

1) <u>Inorder</u>:
    a) Traverse left subtree of node
    b) Process data in node
    c) Traverse right subtree of node
2) <u>Preorder</u>:
    a) Process data in node
    b) Traverse left subtree of node
    c) Traverse right subtree of node
3) <u>Postorder</u>:
    a) Traverse left subtree of node
    b) Traverse right subtree of node
    c) Process data in node

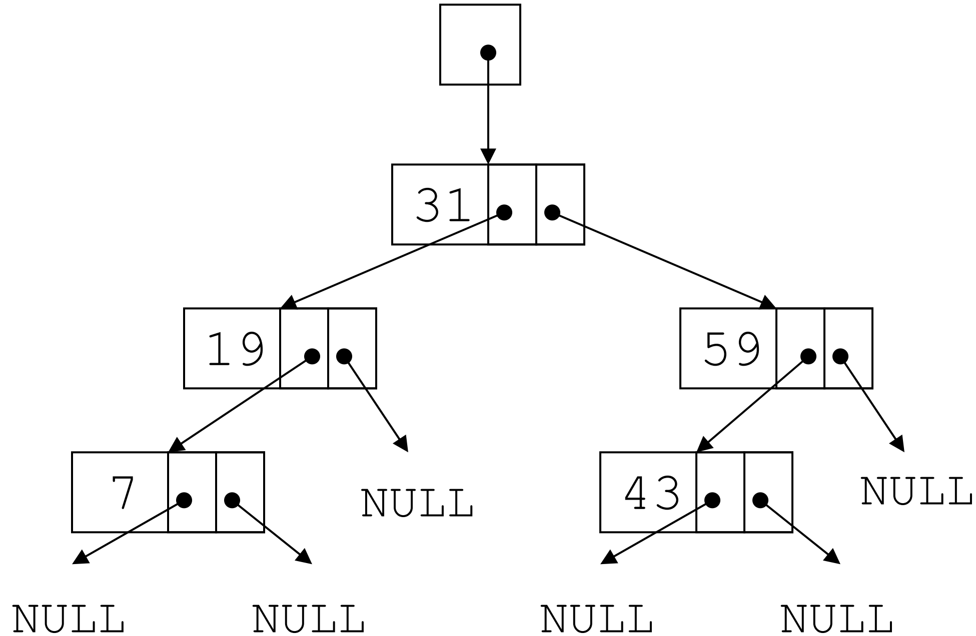
Traversing a Binary Tree



Traversal Method:	Nodes 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

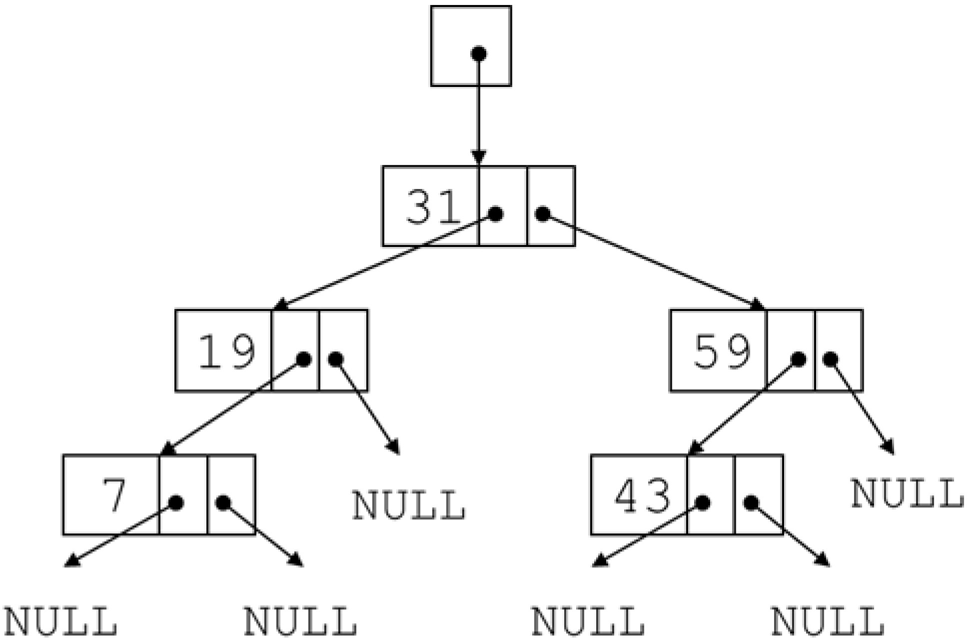
- Start at root node.
- Traverse the tree, looking for the desired value.
- Stop when desired value is found or a `NULL` pointer detected.
- The search function can be implemented as a function that returns a `bool` value



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

Counting Nodes in a Binary Tree

- Start at root node.
- Traverse the tree counting nodes as you go
- Stop when the entire tree has been counted.
- Can be implemented as a function that returns **`int`**

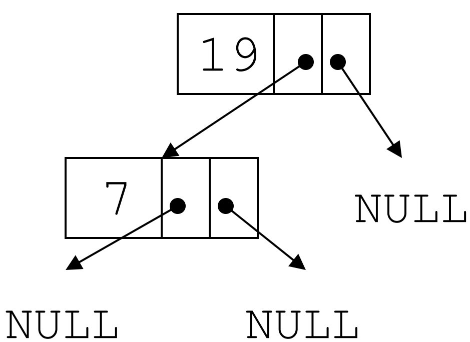


Node-counting code

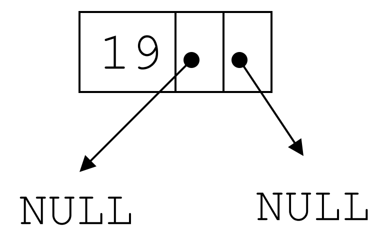
```
int IntBinaryTree::calculateNumberOfNodes(TreeNode
*nodePtr) const
{
   if (nodePtr)  {
      int leftSubTree, rightSubTree;

      leftSubTree =
           calculateNumberOfNodes(nodePtr->left);

      rightSubTree =
           calculateNumberOfNodes(nodePtr->right);
      return (leftSubTree + rightSubTree + 1);
   }  else  { return 0;  }
}
```

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 a `NULL` pointer, then delete the node



Deleting node with 7 – before deletion

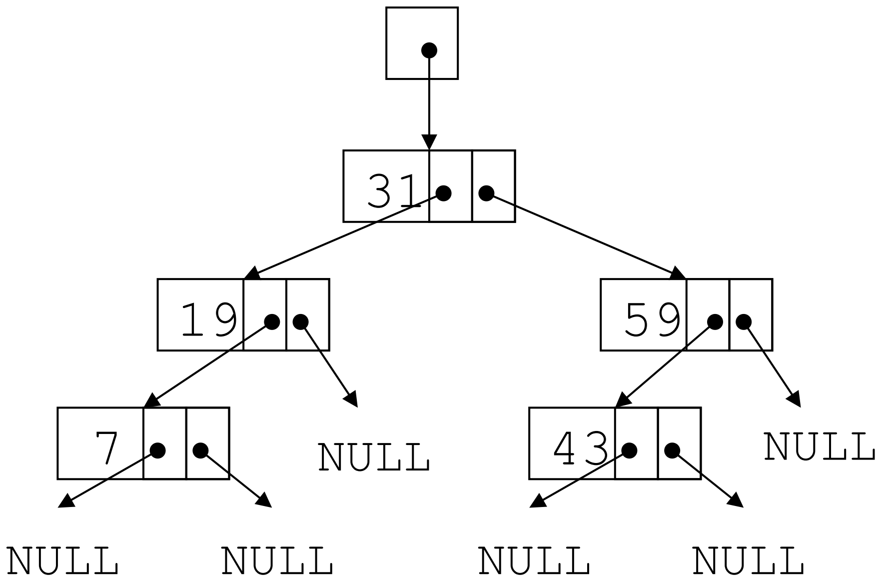


Deleting node with 7 – after deletion

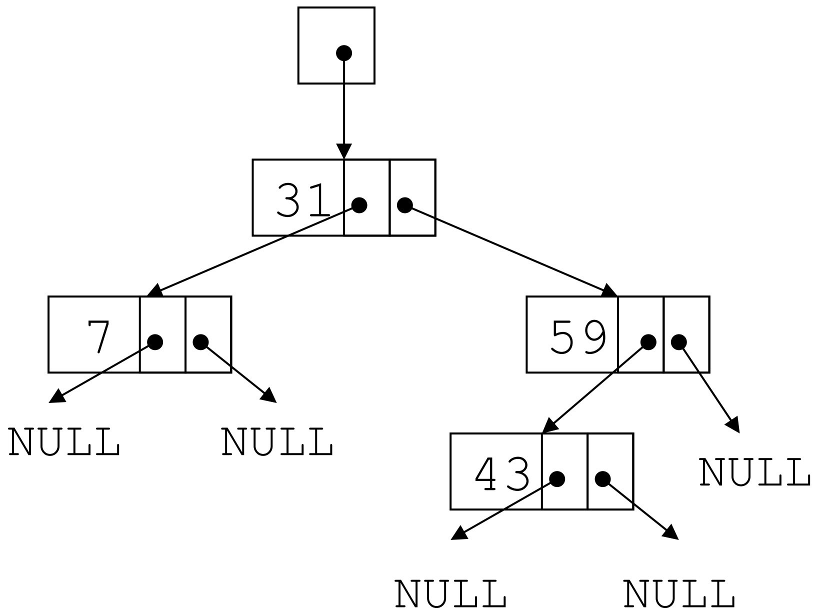
Deleting a Node from a Binary Tree – One Child

- 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 a Node from a Binary Tree – One Child



Deleting node with 19
– before deletion

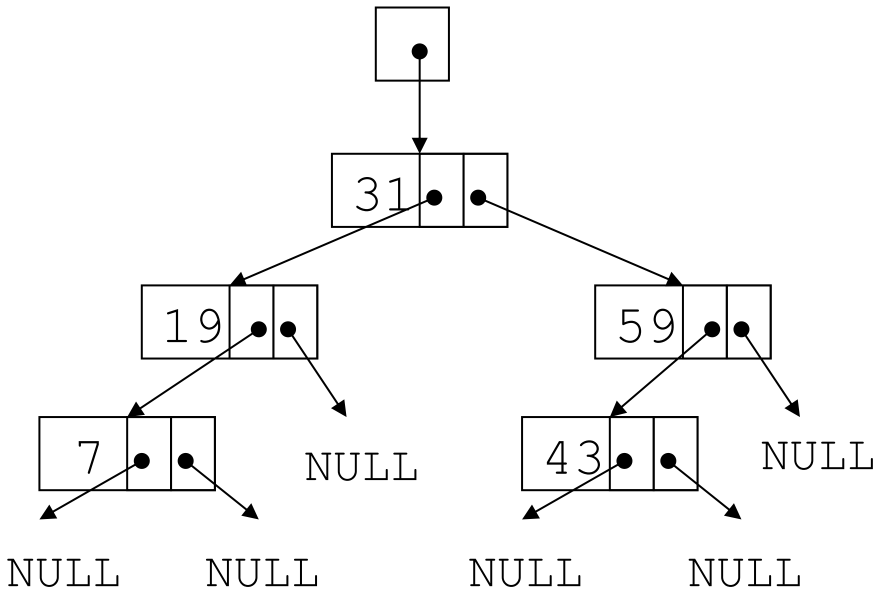


Deleting node with 19
– after deletion

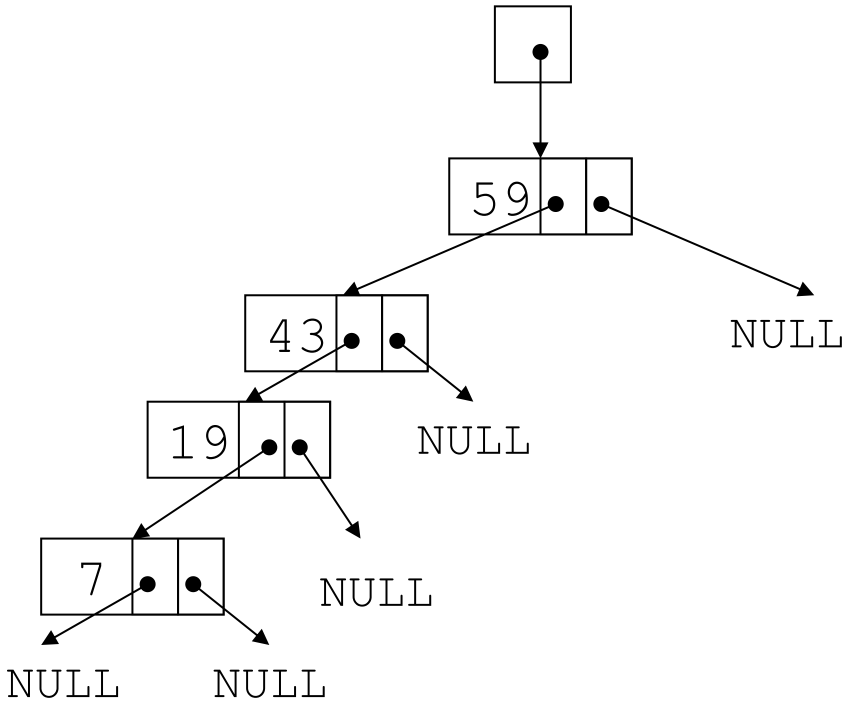
Deleting a Node from a Binary Tree – Two Children

- If node to be deleted has left <u>and</u> 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 our textbook:
  - promote the right child,
  - position left subtree underneath

Deleting a Node from a Binary Tree – Two Children



Deleting node with 31
– before deletion



Deleting node with 31
– after deletion

But it does not look like a tree anymore!

- An “unbalanced” binary tree can perform more like a linear linked list.
- How to balance a tree is really beyond the scope of CIT-237.

- Let’s think about how we might *design* code to balance a binary tree.

Disclaimer: I found this discussion on the internet, and I have **not** tested the solution.

Rebuilding A Binary Tree to Balance It

A “brute force” (and rather expensive) approach:

1. Perform an <u>inorder</u> traversal of the existing tree, inserting a pointer to each node in a vector.
2. Find the index of the <u>midpoint</u> of the vector:

   midpoint = vector size / 2

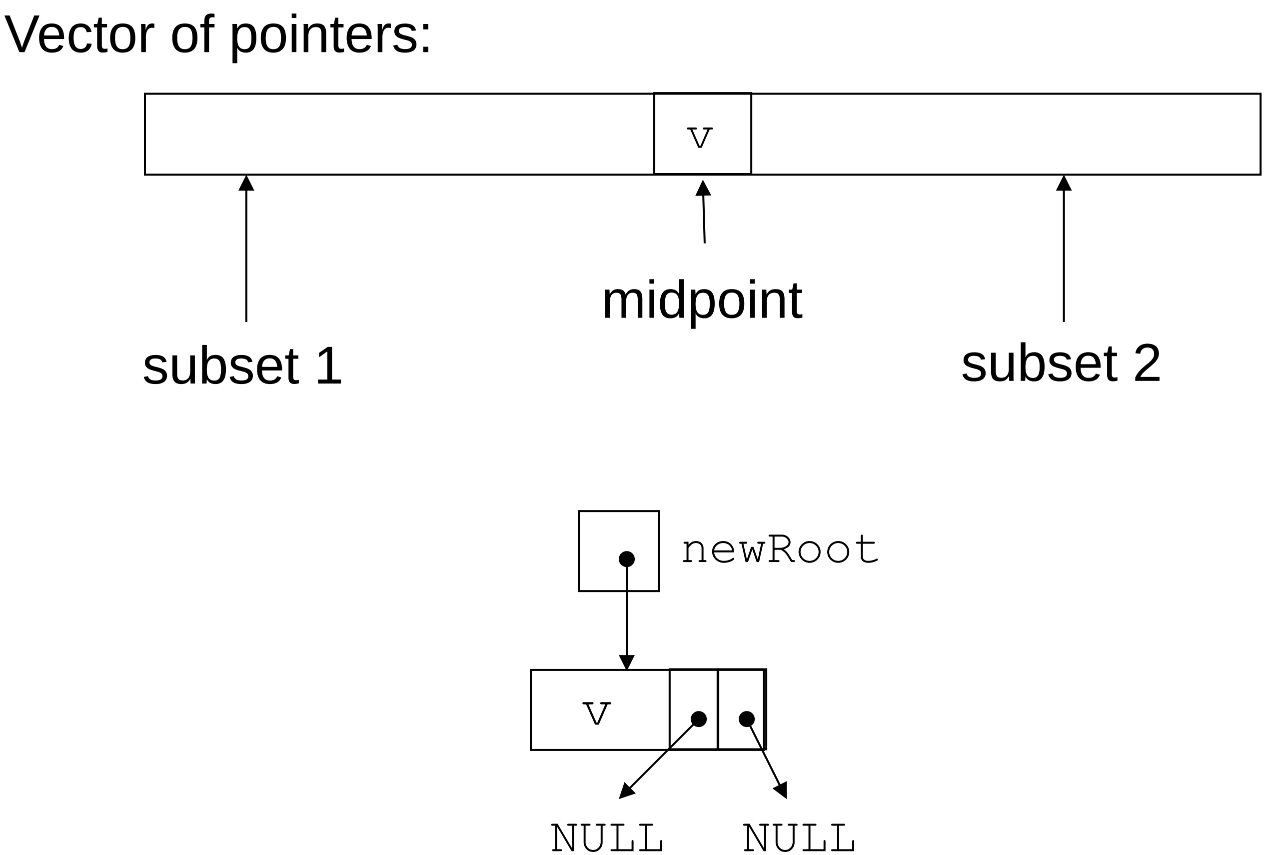
3. Insert the “midpoint node” into the new tree.
4. Define two subsets of the remaining vector elements:

   those whose index is <u>less</u> than the midpoint, and

   those whose index is <u>greater</u> than the midpoint.

5. Recursively insert each subset into the new tree.

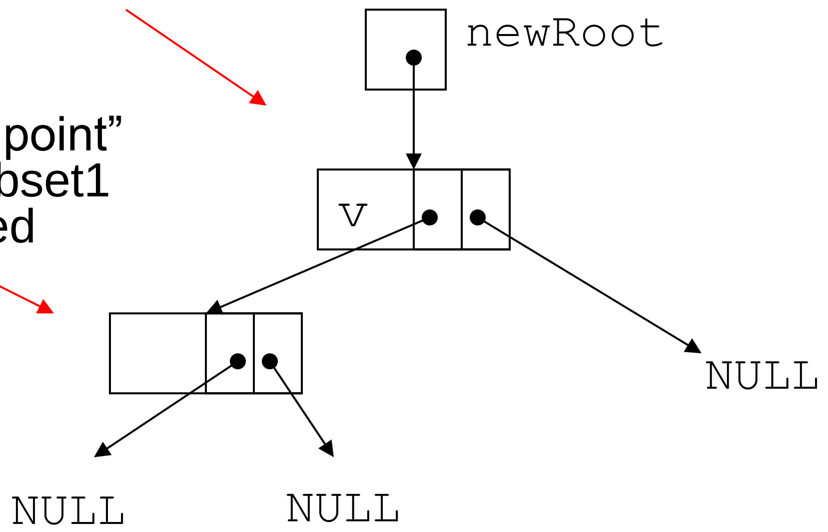
Begin to build the New Tree



Adding the subsets

1. The “midpoint” value gets inserted first

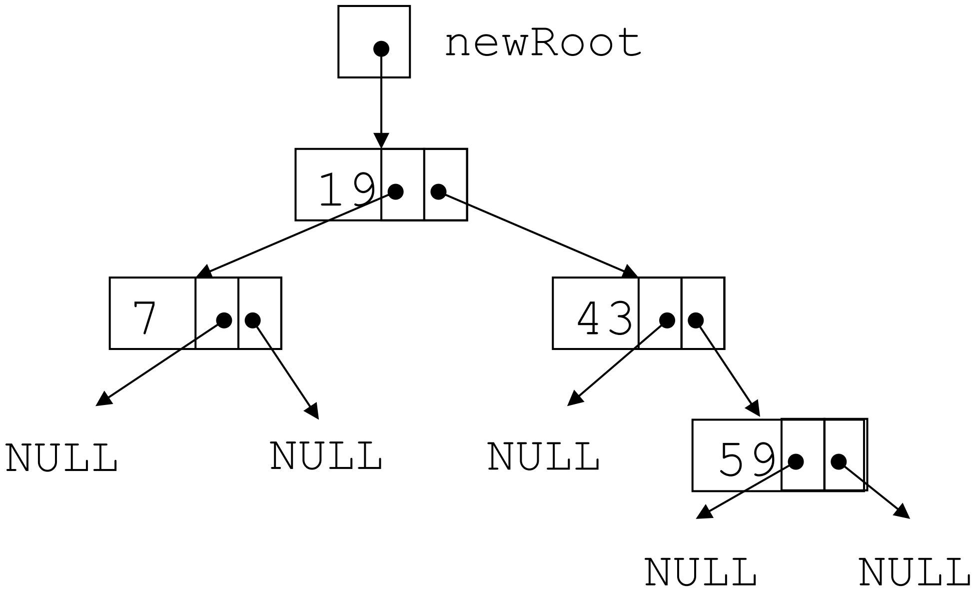
2. The “midpoint” value of subset1 gets inserted second



4. The subset2 nodes become the <u>right</u> sub-tree

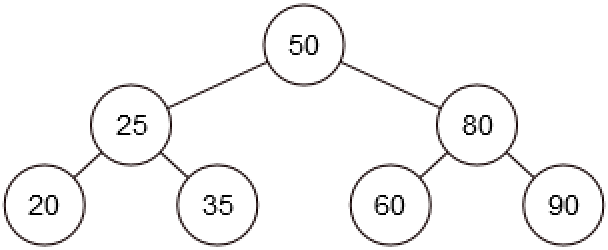
3. The remaining subset1 nodes complete the <u>left</u> sub-tree

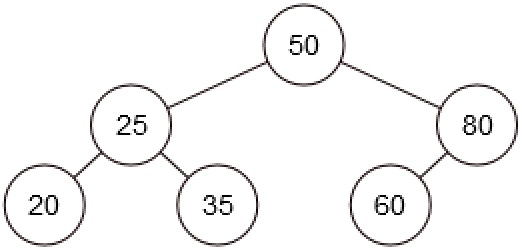
Final Result



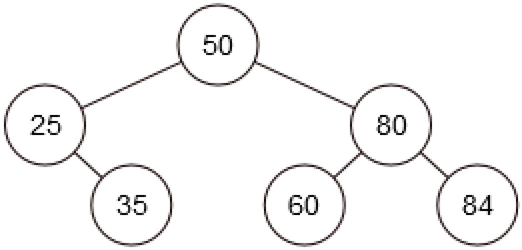
Complete Binary Tree

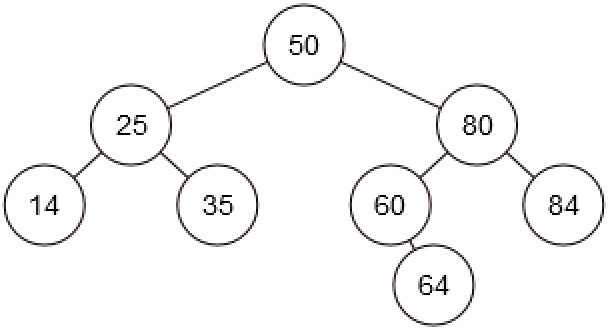
- A binary tree is *complete* if every level of the tree is full, or
- If the last level is not full, all the nodes on the last level are placed left-most.
- For example, these two example trees are *complete*:





- But these two are <u>not</u> *complete*:





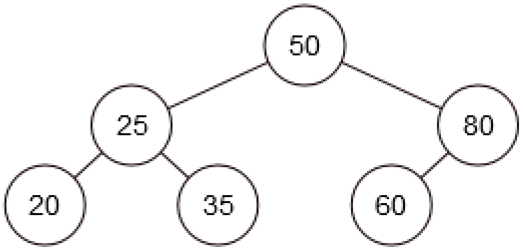
Complete Binary Tree Representation (1)

- Previously, we looked at binary trees represented by linked objects.
- An alternate representation of a <u>complete</u> binary tree, using an array, is sometimes preferable:
  - It saves space.
  - It provides an easier way for a program to find the <u>parent</u> node of any particular node.

Complete Binary Tree Representation (2)

- While our conceptual picture of any binary tree is as a tree, we can choose to represent a *complete* binary tree using an array.

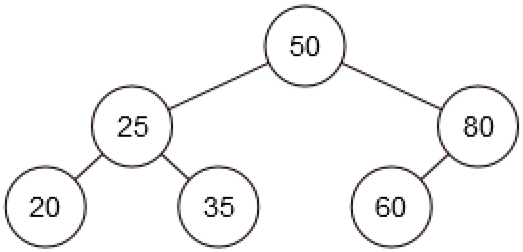
For example, the tree shown below:

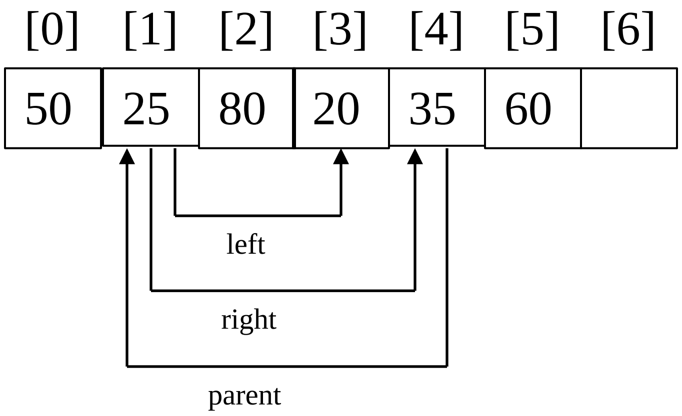


can be represented by the following array:

[0]	[1]	[2]	[3]	[4]	[5]	[6]
50	25	80	20	35	60	

Complete Binary Tree Representation (3)





- For a node at position *i*:
  - its left child is at position (*2*i)+1*
  - its right child is at position (*2*i)+2*
  - its parent is at index (*i-1)/2* (Integer division – remainder is discarded.)
- For example: The node for element 25 is at position 1.
  - its left child (element 20) is at position 3 (*2*1+1*)
  - its right child (element 35) is at position 4 (*2*1+2*)
  - Element 25 is the parent of element 35: position of element 35 is 4. The parent of the element at position 4 is at position (4-1)/2 = 1.

Binary Trees: Summary

- *Binary Search Trees* are more efficient to search than linear lists (unless they are excessively “unbalanced”).
- Binary Trees can be traversed in different ways. We looked at three examples:
  - Inorder Traversal
  - Preorder Traversal
  - Postorder Traversal
- Algorithms for manipulating Binary Trees are often implemented using recursion.
- Maintaining the correct order of a *Binary Search Tree* (as the tree is modified) can result in the tree becoming unbalanced.
- Rebalancing a Binary Tree can be time consuming.

Today's Lab Exercise: Lab 21.1

- The lab exercise for today has an extensive "starter" program.
  - Includes the **IntBinaryTree** class (adapted from the code in Chapter 21).
- The Lab assignment will be to complete a few "unfinished" features in the program.
- Let's get familiar with the <u>Lab 21.1</u> program:
  - Interactive commands to display and modify a binary tree.
  - Several sample trees, represented as text files.