BINARY TREES

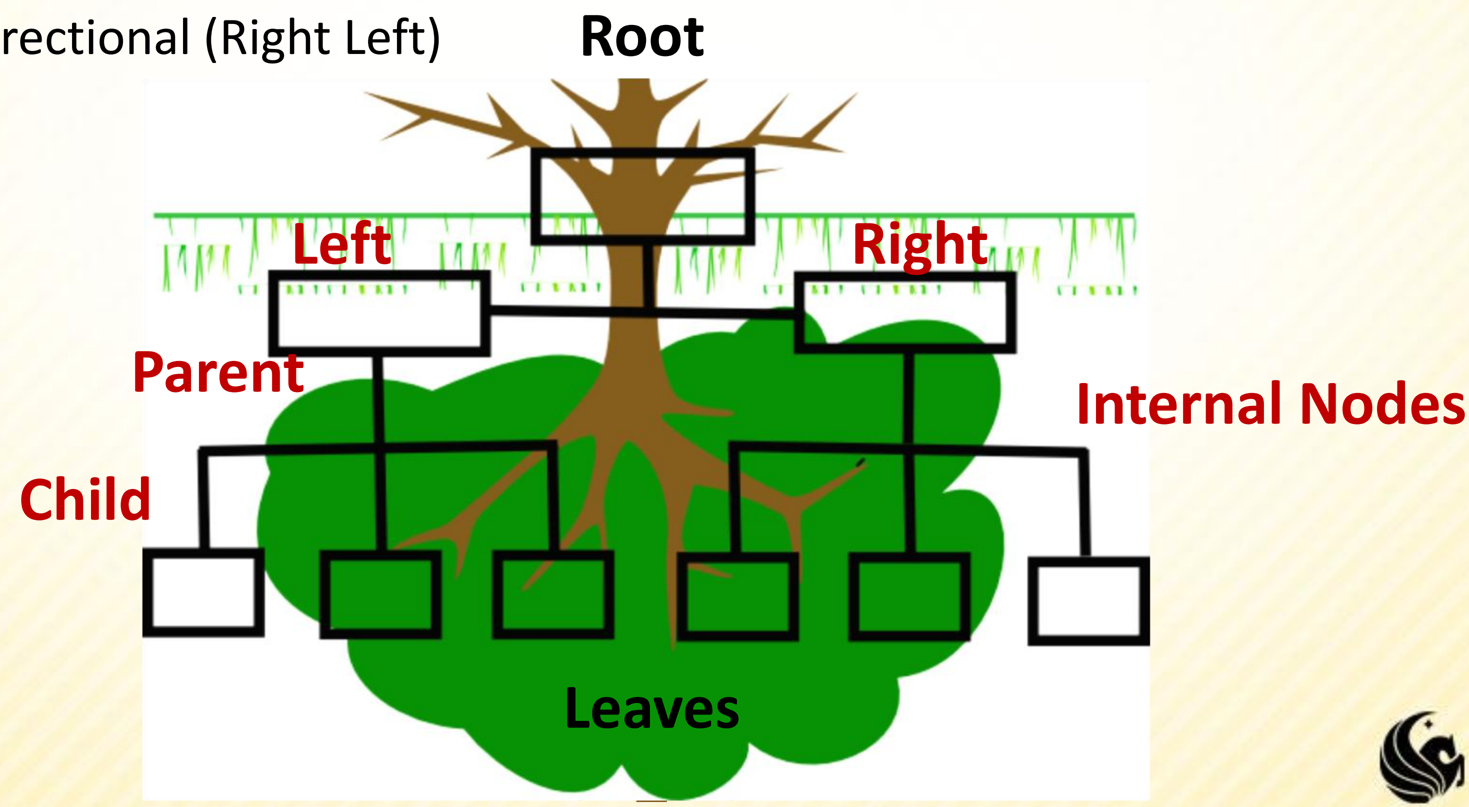
COP 3502

Trees

- We’ve already seen lists of linked nodes
  - But the problem was that it took a long time to get to an arbitrary node in a linked list.
  - It would be nice if we had a linked structure where nodes were more easily accessible.
- A tree is a widely used data structure that has a hierarchical set of linked nodes.
  - If you think of a tree with branches
    - And each point where branches intersect as a node
    - You find a structure with a huge number of nodes, but where each path is not too long.

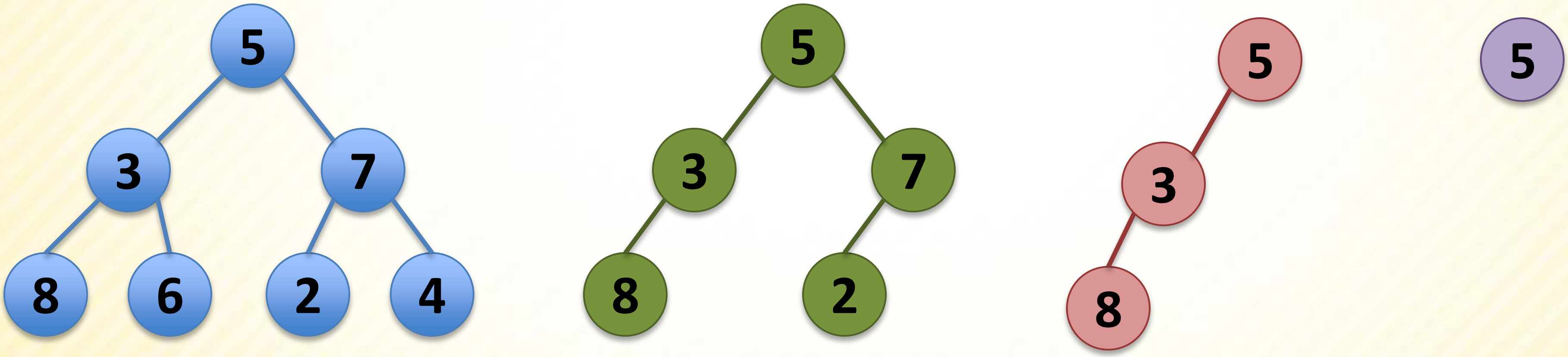
Trees

- A tree is a widely used data structure that has a hierarchical set of linked nodes.
  - We have several ways of referring to nodes:
    - Biological (Root, leaves)
    - Familial(Parent and child)
    - Directional (Right Left)



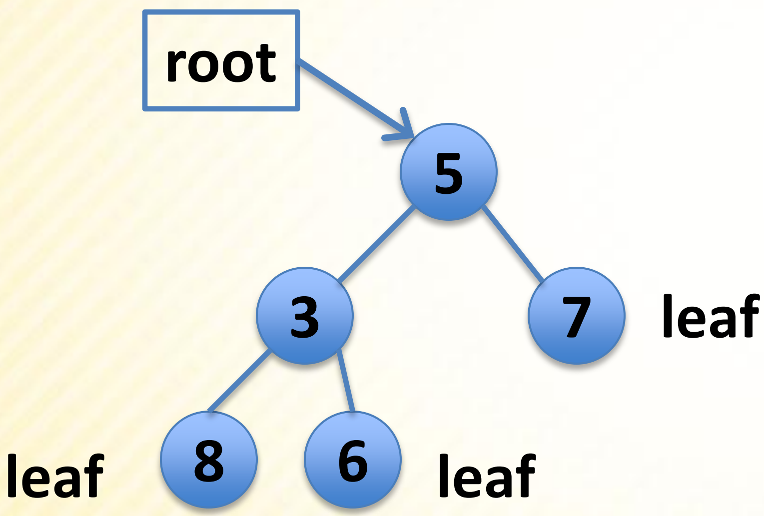
Binary Tree

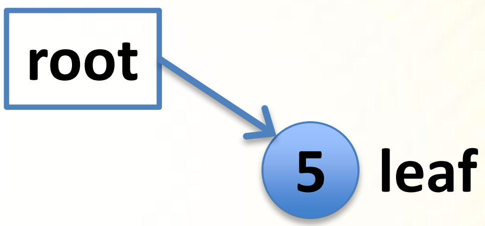
- A binary tree is a data structure in which each node has at most 2 child nodes
  - So these are examples of binary trees



Binary Trees

- A **leaf** node has no children.

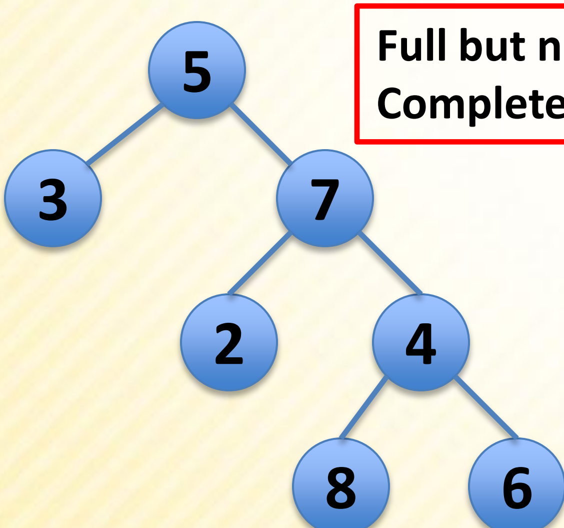




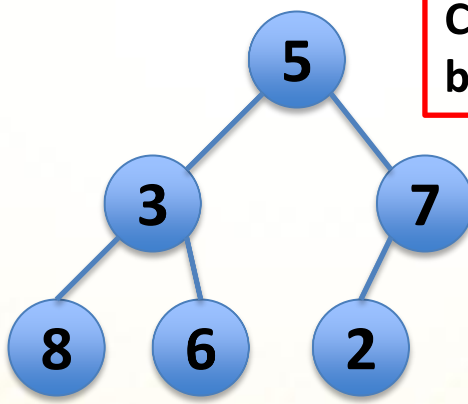
Binary Trees

- A Binary Tree is ***full*** if each node is either a leaf or has exactly two child nodes.

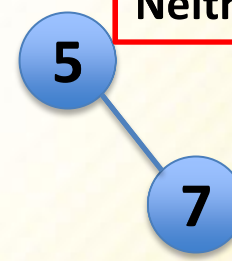
- A Binary Tree is ***complete*** if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



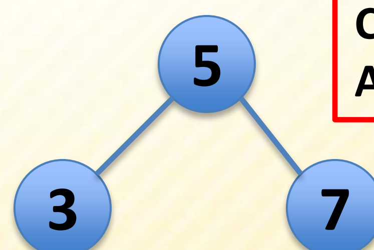
Full but not Complete



Complete but not Full



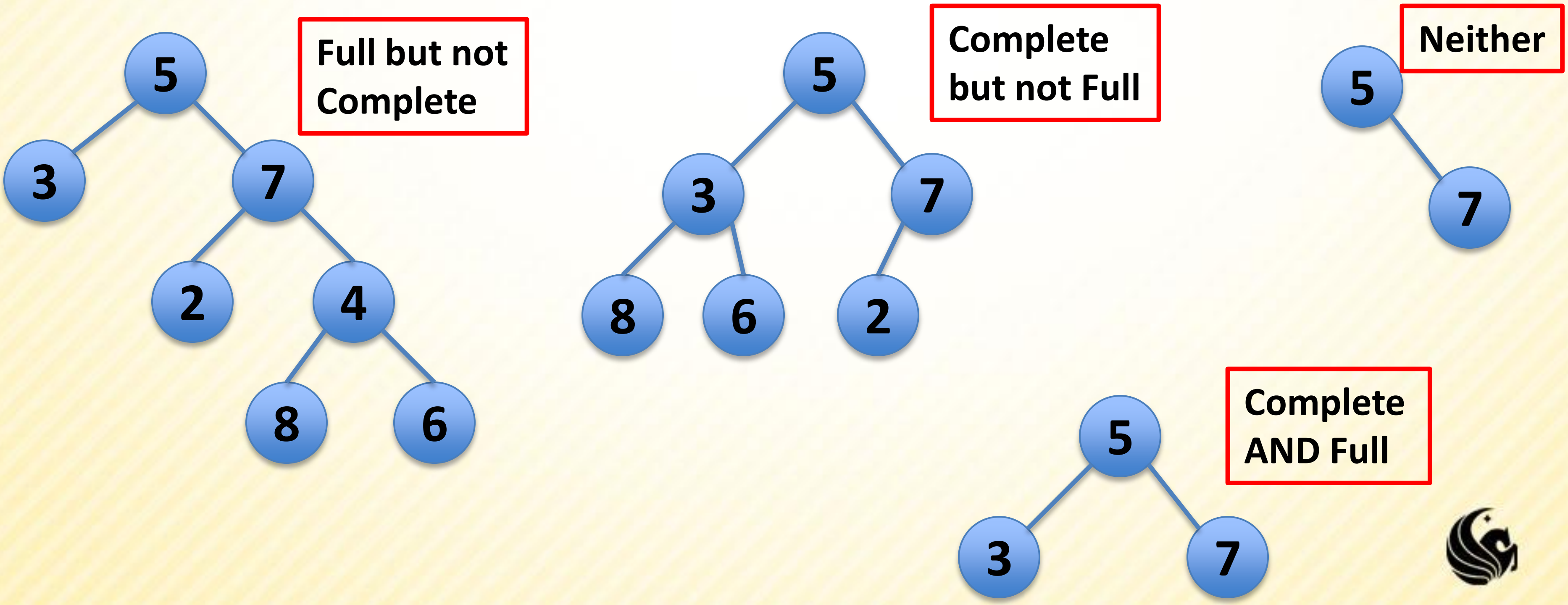
Neither



Complete AND Full

Binary Trees

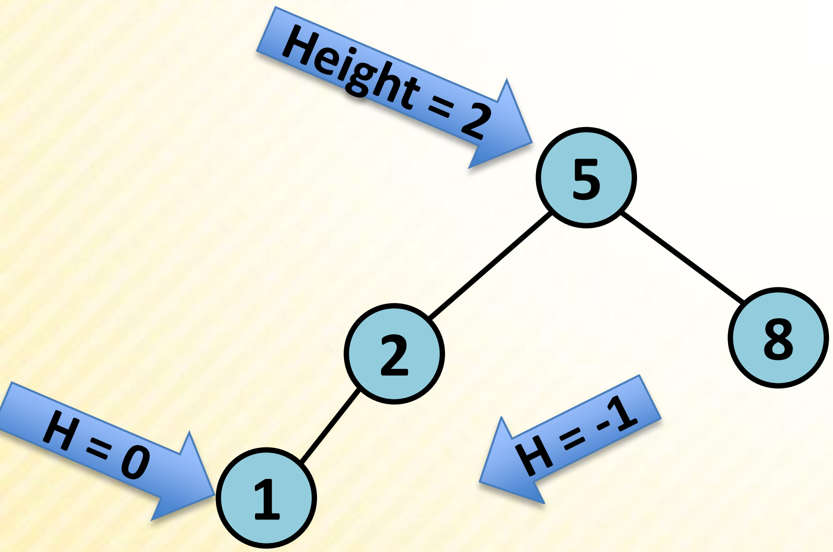
- The height of a binary tree is

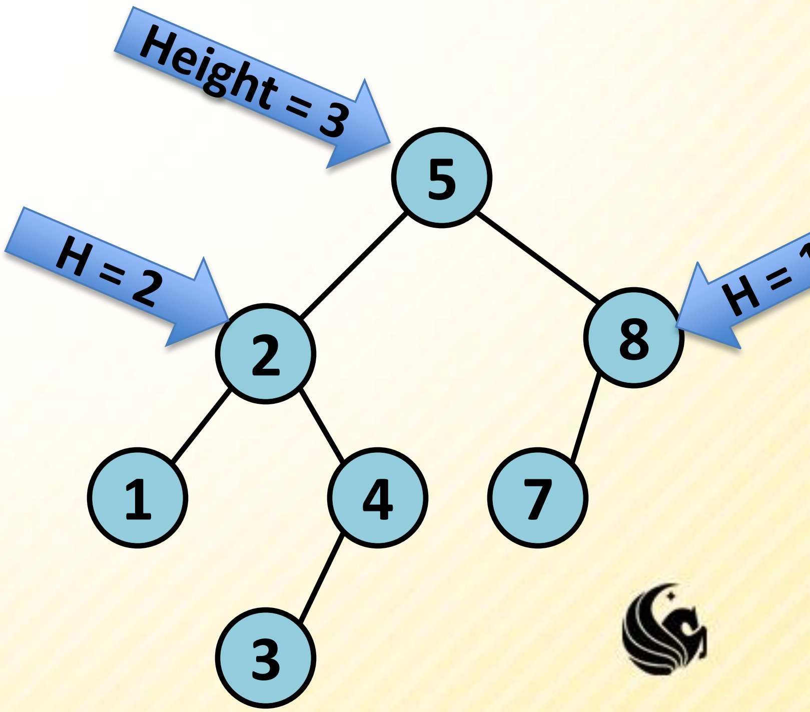


AVL Trees

height of a binary tree:
the length of the longest path from the root to a leaf.
(the height of an empty tree is -1)
(the height of a leaf is 0)

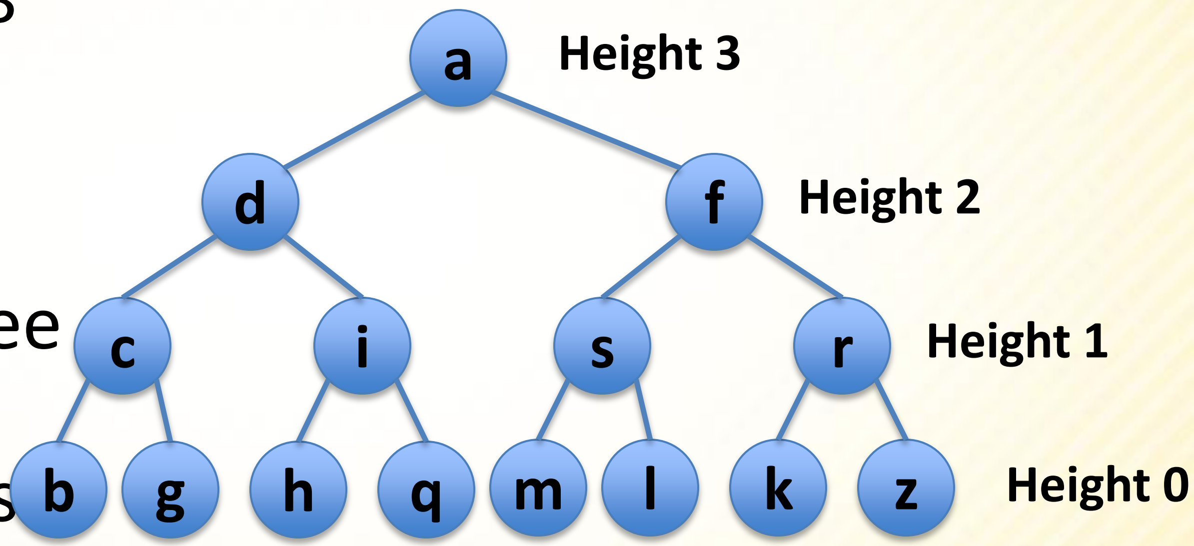
- The height of any node is 1 more than the max height of its children





Binary Trees

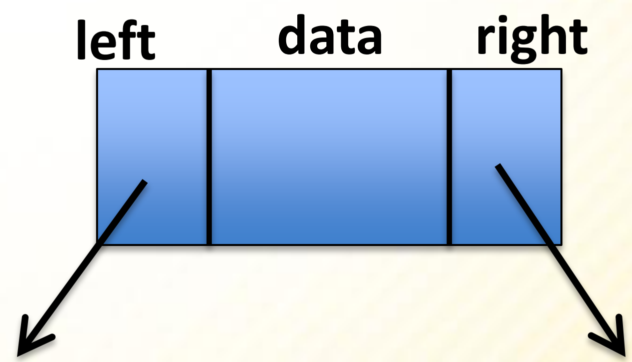
- Total # of nodes **n** is
  - $\mathbf{n = 2^{h+1} - 1}$ (maximum)
  - For example, if h = 3
  - The max nodes in a complete tree is:
  - $n = 2^4 - 1 = 15$
- Height of the full tree **h**, if there are **n** nodes
  - $\mathbf{h = \log_2((n+1)/2)}$
  - If we have 15 nodes
  - $\mathbf{h = \log_2(16/2)}$
  - $\mathbf{= \log_2(8) = 3}$



Binary Tree Node

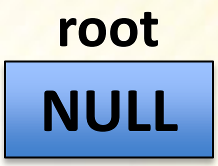
- A node of a binary tree is very similar to a node in a linked list.
  - Except instead of having 1 field as a pointer field,
  - we should have 2 pointer fields – a left and a right.

```
struct node {
    int data;
    struct node *left;
    struct node *right
};
```

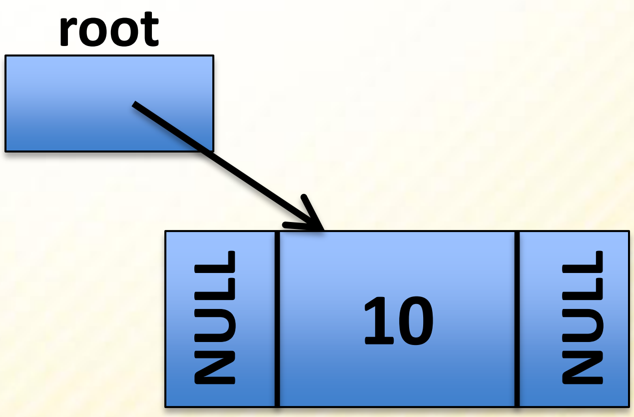


Binary Trees

- To declare an empty binary tree:
  - struct node *root = NULL;



- To add a single node to the tree, we could do:
  - `root = (struct node*)malloc(sizeof(struct node));`
  - `root->data = 10;`
  - `root->left = NULL;`
  - `root->right = NULL;`



Traversing a Binary Tree

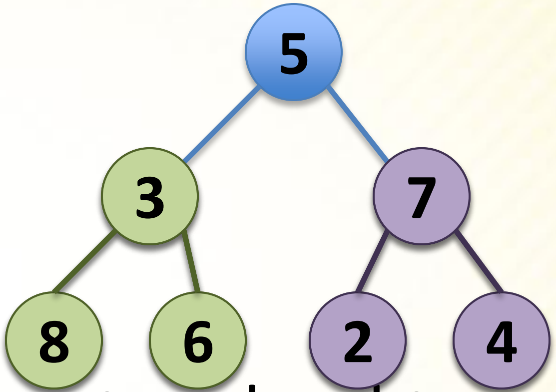
- In a linked list we could traverse starting with the head and stopping when we got to NULL.
  - We can’t really do that in a binary tree
    - Things are not so trivial for a tree.

- We will have to turn to our good old friend
  - Recursion

  - (Note: we’re covering traversing a tree before we cover inserting into a tree, so let’s assume we already have an existing tree.)

Traversing a Binary Tree

- Consider the 3 components of a binary tree:
  1) A node (the root node)
  2) A left subtree
  3) A right subtree

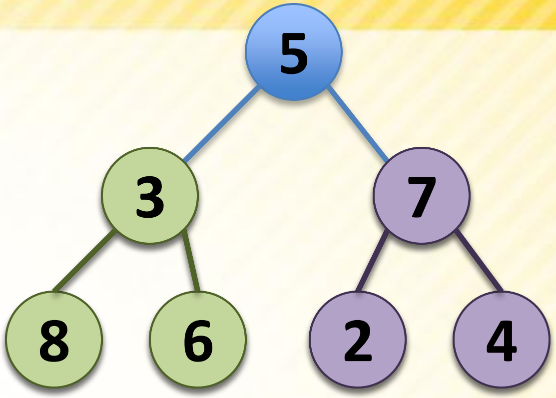


- What we notice is that we can treat each subtree as a binary tree with
  1) A root node
  2) A left subtree
  3) A right subtree

➢ This is where the recursion comes in, we'll traverse each subtree recursively.

Traversing a Binary Tree

- The 3 components of a binary tree:
  1) A node (the root node)
  2) A left subtree
  3) A right subtree



- We can traverse these 3 components in any order we want
  - Typically though the left is always traversed before the right.
    - This leaves us 3 options then:
  1) Root, Left, Right – Pre-Order Traversal
  2) Left, Root, Right – In-Order Traversal
  3) Left, Right, Root – Post-Order Traversal

Inorder Binary Tree Traversal

- An inorder tree traversal visits the 3 parts of a tree in this order:
  1) left subtree
  2) root node
  3) right subtree

This traversal is the most common because it is typically used to go through a sorted list in order stored in a binary tree.

- Here is a function that would print each node in a tree using an Inorder traversal:

```
void Inorder(struct node *curr)
if (curr != NULL) {
        Inorder(curr->left);
        printf("%d ", curr->data);
        Inorder(curr->right);
    }
}
```

Inorder Binary Tree Traversal

- We’ll show an example Inorder traversal on the board in class.

Preorder Binary Tree Traversal

- A preorder tree traversal visits the 3 parts of a tree in this order:
  1) root node
  2) left subtree
  3) right subtree
  - Here is a function that would print each node in a tree using a Preorder traversal:

```
void Preorder(struct node *curr)
if (curr != NULL) {
        printf("%d ", curr->data);
        Preorder(curr->left);
        Preorder(curr->right);
    }
}
```

Inorder Binary Tree Traversal

Postorder Binary Tree Traversal

- A postorder tree traversal visits the 3 parts of a tree in this order:
  1) left subtree
  2) right subtree
  3) root node

  - Here is a function that would print each node in a tree using a Postorder traversal:

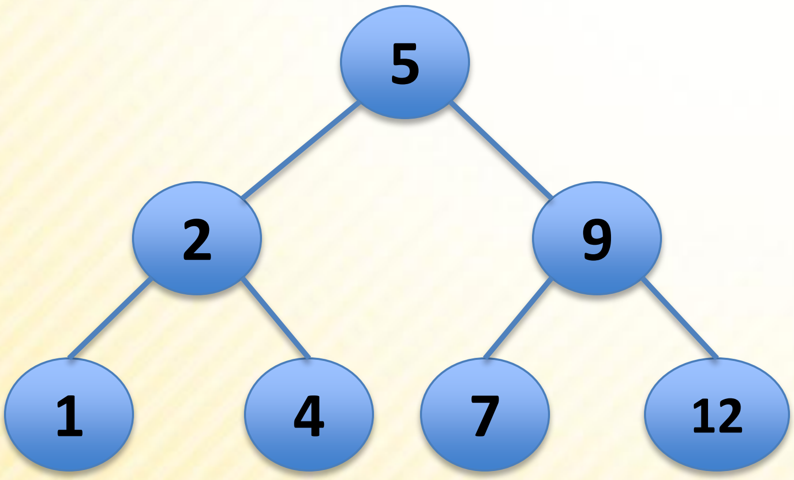
```
void Postorder(struct node *curr)
if (curr != NULL) {
        Postorder(curr->left);
        Postorder(curr->right);
        printf("%d ", curr->data);
   }
}
```

Inorder Binary Tree Traversal

- We’ll show an example Inorder traversal on the board in class.

Binary Search Tree

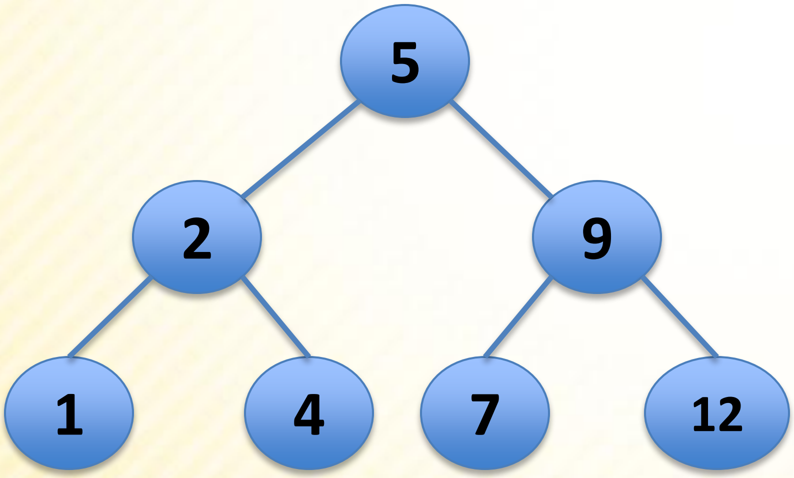
- Even though we now know how to traverse a binary tree
  - it's not clear how a binary tree can benefit us…
  - but what if we added a restriction to a binary tree?
- Consider the following binary tree:



- What patterns are true about each node in the tree?
  - For each node N all the values in the left subtree are LESS than the value in node N.
  - And the values in the right subtree are GREATER than the value stored in N.

Binary Search Tree

- Binary Search Tree property:
  - For each node N all the values in the left subtree are LESS than the value in node N.
  - And the values in the right subtree are GREATER than the value stored in N.

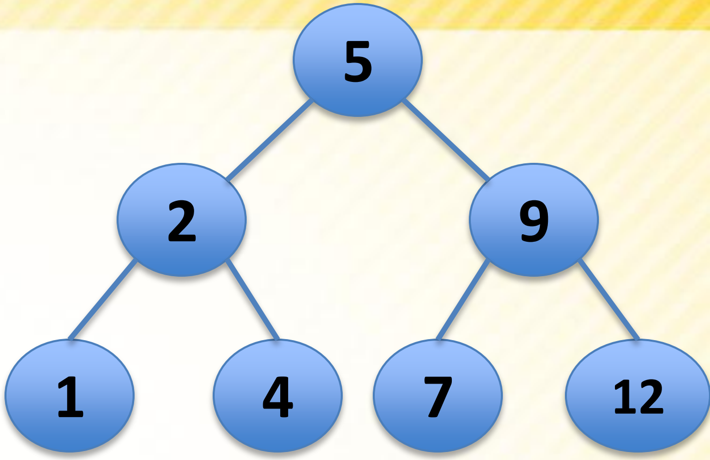


- Why might this property be a desirable one?
  - It's going to make searching much easier!
  - Rather than "looking" both directions after checking a node, we know EXACTLY which direction to go.

Notice the Binary Search Tree Property holds true recursively, so if we look at the left subtree as a separate tree the property holds, and same for the right.

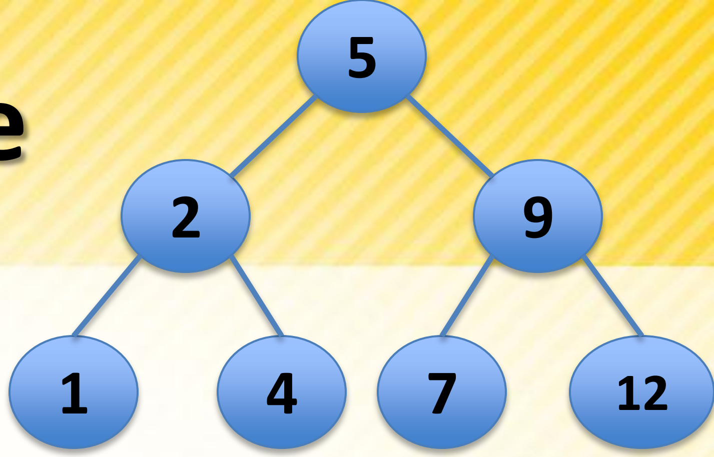
Binary Search Tree

- Searching a Binary Search Tree:
  - Let's see if we can come up with the code given the following algorithm.




```
int Find(struct node *curr, int val) {
   // 1) if the tree is NULL, return false
   // 2) Check root node, if we find val return true!
   // 3) else if the val is less than root's value,
   //    recursively search the left subtree
   // 4) else recursively search in the right subtree.
}
```

Binary Search Tree



- Searching a Binary Search Tree:

```
int Find(struct node *curr, int val) {

  if (curr != NULL)  {
     if (curr->data == val)
       return 1;
     if (val < curr->data)
       return Find(curr->left, val);
     else
       return Find(curr->right, val);
  }
  else
     return 0;
}
```