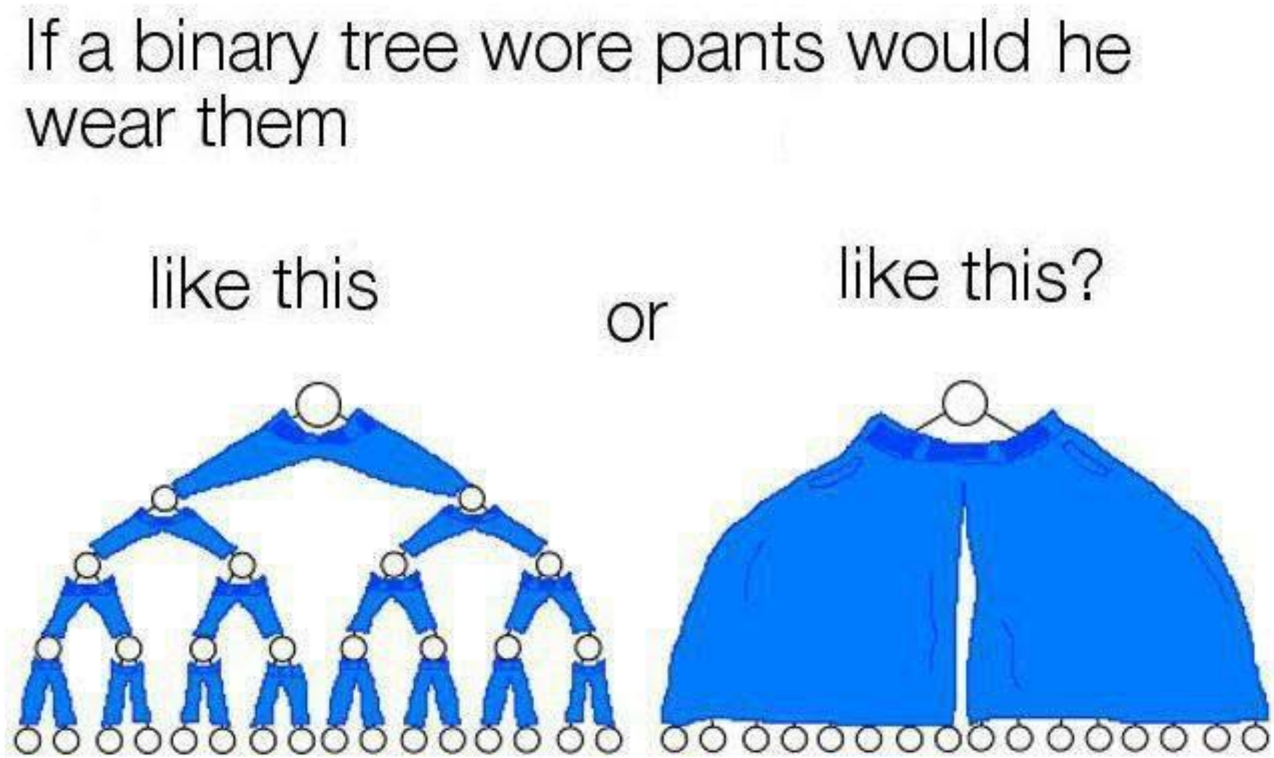
Lecture #12

- Binary Tree Traversals
- Evaluate Expressions Using

- Binary Search Trees
- Binary Search Tree Operations
  - Searching for an item
  - Inserting a new item
  - Finding the minimum and maximum items
  - Printing out the items in order
  - Deleting the whole tree

Binary Trees, Cont.



Binary Tree Traversals

When we process all the nodes in a tree, it's called a traversal.

There are four common ways to traverse a tree.

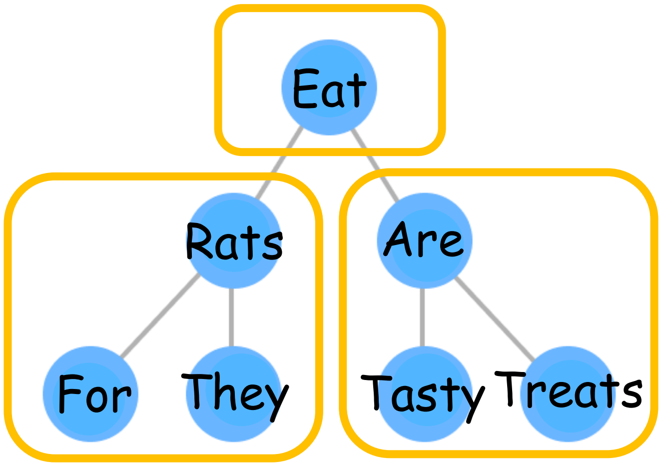
1. Pre-order traversal (we did this last time)
2. In-order traversal
3. Post-order traversal
4. Level-order traversal

Let's see an in-order traversal first!

The Pre-order Traversal: Refresher

PreOrder(current_node):

1. Process the current node.
2. Recursively process nodes in the left sub-tree.
3. Recursively process nodes in the right sub-tree.



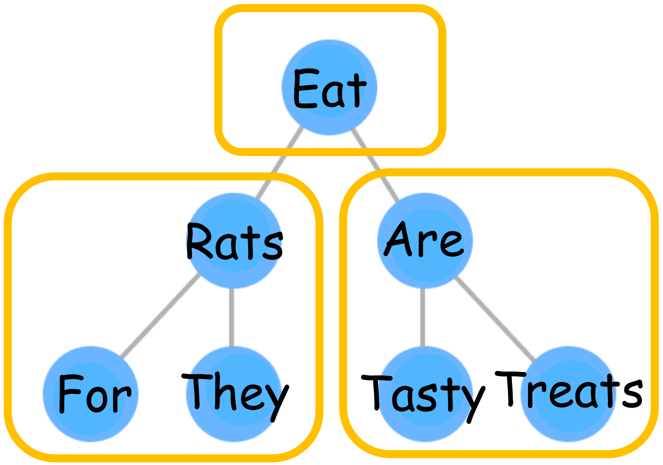
Can you guess why it's called a "pre-order" traversal?

Because we pre-process the current node... before processing its left and right subtrees.

The In-order Traversal

InOrder(current_node):

1. Recursively process nodes in the left sub-tree.
2. Process the current node.
3. Recursively process nodes in the right sub-tree.

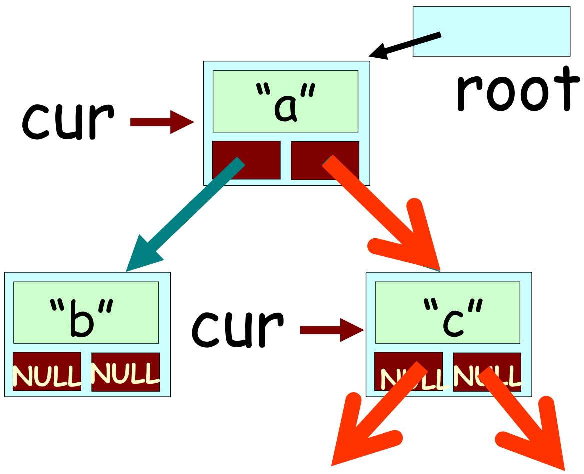


Can you guess why it's called an "in-order" traversal?

Because we process the current node...
in-between processing its left and right subtrees.

The In-order Traversal

- Here's our in-order traversal
- You can see that we have our same base-case which checks to see if we're processing an empty tree. This is when cur == nullptr, and there is no valid node.
- Otherwise, we first process the left subtree by passing in cur->left to our function.
- Then when the call to process the entire left subtree returns, we then process the current node.
- Finally, we process the right subtree by passing in cur->right to our function.
- Important note: If you use an in-order traversal on a binary SEARCH tree (which is a type of binary tree), it will visit all of the values in alphabetical order!




```
void InOrder(Node *cur)
{
   if (cur == nullptr)          // if empty, return…
        return;

   InOrder(cur->left);      // Process nodes in left sub-tree.

   cout << cur->value;      // Process the current node.

   InOrder(cur->right);  // Process nodes in right sub-tree.
}
```

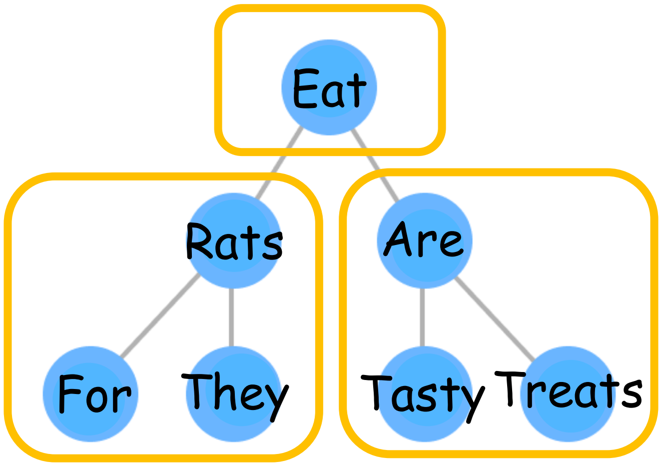
Output:

b a c

The Post-order Traversal

PostOrder(current_node):

1. Recursively process nodes in the left sub-tree.
2. Recursively process nodes in the right sub-tree.
3. Process the current node.

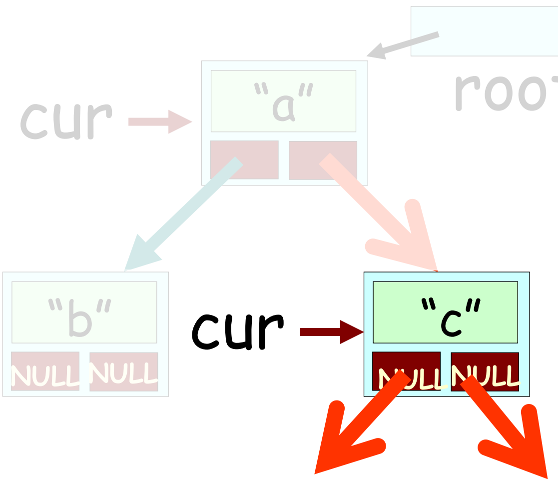


Can you guess why it's called a "post-order" traversal?

Because we first process its left and right subtrees… and only then post-process the current node…

The Post-order Traversal

- Here's our post-order traversal
- You can see that we have our same base-case which checks to see if we're processing an empty tree. This is when cur == nullptr, and there is no valid node.
- Otherwise, we first process the left subtree by passing in cur->left to our function.
- Then, we process the right subtree by passing in cur->right to our function.
- Finally, once we've processed both of the subtrees (if any), we then process the current node.
- Important note: Post-order traversals are useful for things like expression evaluation and freeing trees during destruction.




```
void PostOrder(Node *cur)
{
   if (cur == nullptr)          // if empty, return…
        return;

   PostOrder(cur->left);      // Process nodes in left sub-tree.

   PostOrder(cur-> right);  // Process nodes in right sub-tree.

   cout << cur->value;       // Process the current node.
}
```

Output:

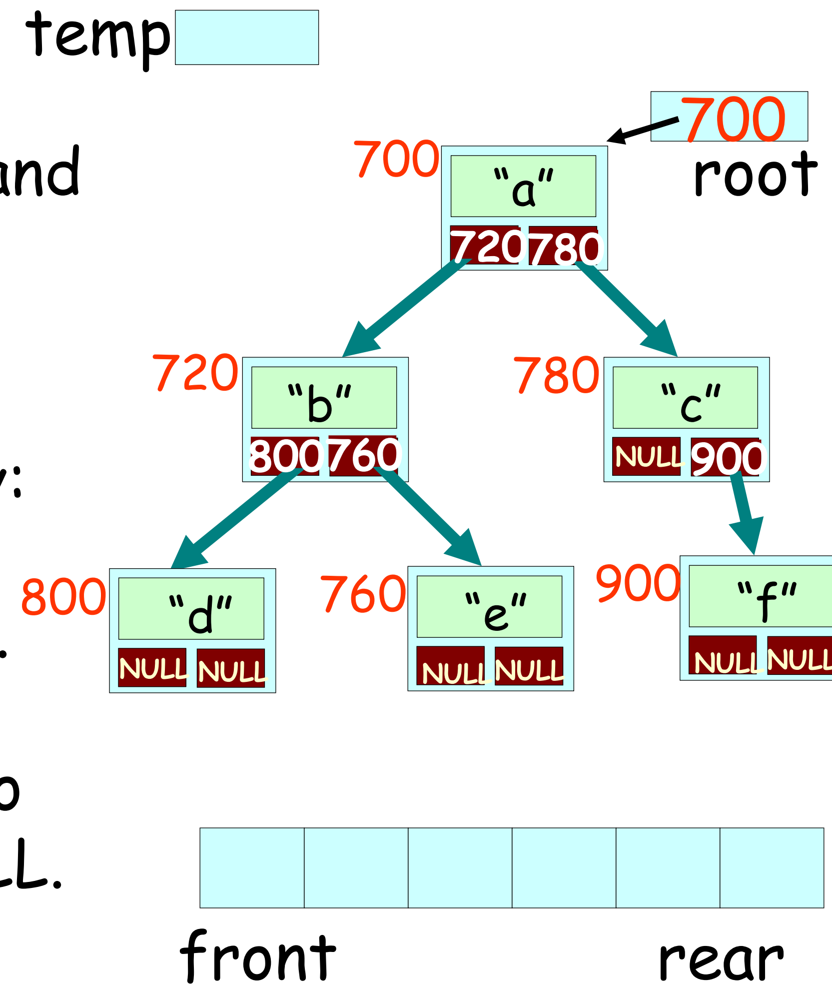
b c a

The Level Order Traversal

In a *level order traversal* we visit each level's nodes, from left to right, before visiting nodes in the next level.

Here's the algorithm:

1. Use a temp pointer variable and a queue of node pointers.
2. Insert the root node pointer into the queue.
3. While the queue is not empty:
   A. Dequeue the top node pointer and put it in temp.
   B. Process the node.
   C. Add the node's children to queue if they are not NULL.



a b c d e f

Big-O of Traversals?

Question: What're the big-ohs of each of our traversals?

Each of our traversals performs three operations per node:

They process the value in the current node.

They initiate processing of its left subtree.

They initiate processing of its right subtree.

So for a tree with n nodes, that's 3*n operations, or…

O(n)

We "process" the value in each node just once.

We initiate processing of the node's left subtree.

We initiate processing of the node's right subtree.

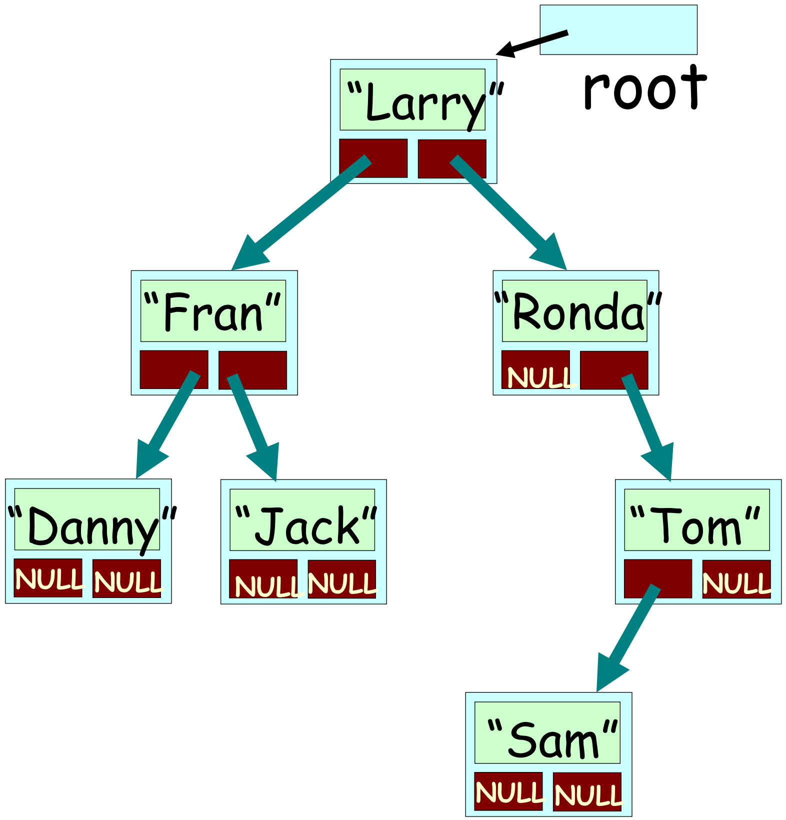
```
void PreOrder(Node *cur)
{
    if (cur == nullptr)
        return;

    cout << cur->value;
    PreOrder(cur->left);
    PreOrder(cur-> right);
}
```

Traversal Challenge

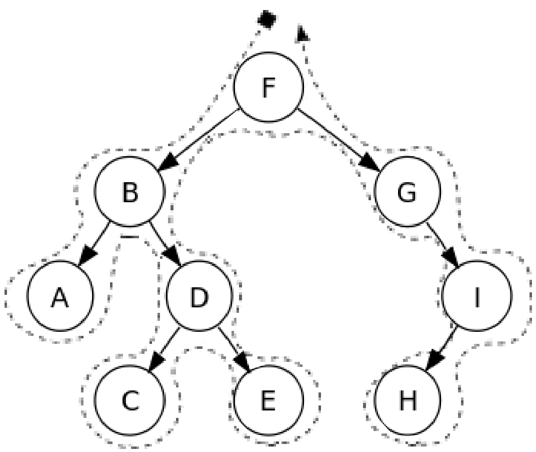
RULES

- The class will split into left and right teams
- One student from each team will come up to the board
- Each student can either
  - write one new item or
  - fix a single error in their teammates solution
- Then the next two people come up, etc.
- The team that completes their program first wins!

Challenge: What order will the following nodes be printed out if we use an in-order traversal?



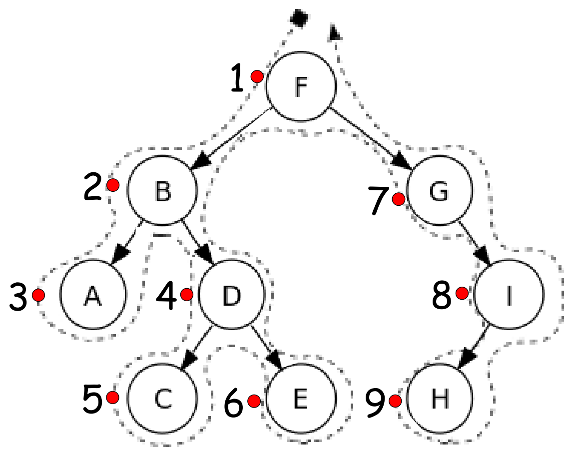
An Easy Way to Remember the Order of Pre/In/Post Traversals



Starting just above-left of the root node, draw a loop counter-clockwise around all of the nodes.

Ok, got that?

Pre-order Traversal: Dot on the LEFT



Pre-order:

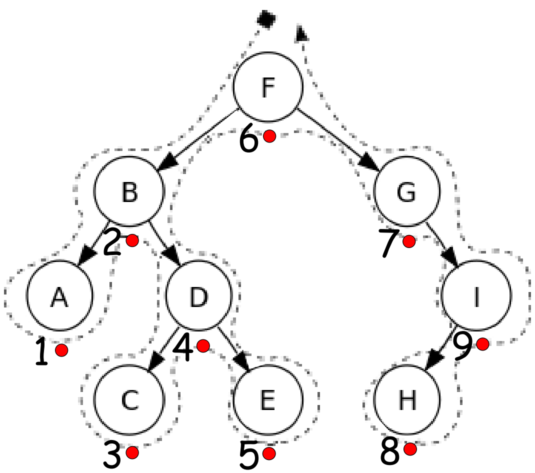
F B A D C E G I H

To determine the order of nodes visited in a pre-order traversal…

Draw a dot next to each node as you pass by its left side.

The order you draw the dots is the order of the pre-order traversal!

In-order Traversal: Dot UNDER the node



In-order:

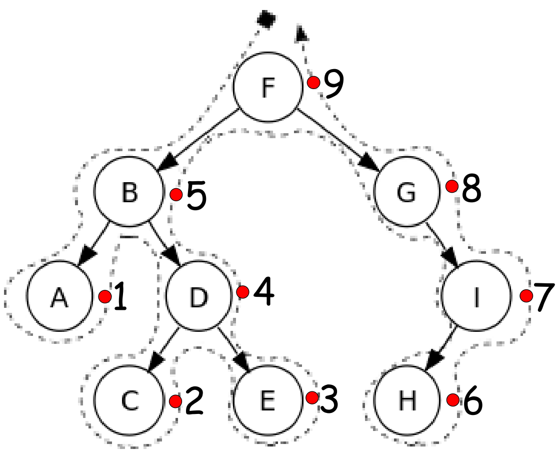
A B C D E F G H I

To determine the order of nodes visited in a in-order traversal…

Draw a dot next to each node as you pass by its under-side.

The order you draw the dots is the order of the in-order traversal!

Post-order Traversal: Dot on the RIGHT



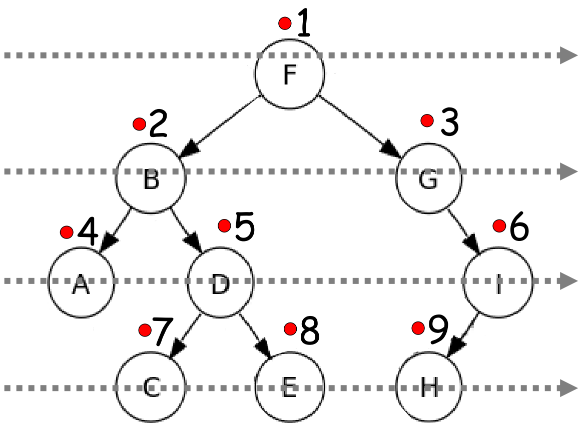
Post-order:
A C E D B H I G F

To determine the order of nodes visited in a post-order traversal…

Draw a dot next to each node as you pass by its right side.

The order you draw the dots is the order of the post-order traversal!

Level-order Traversal: Level-by-level



Level-order:
F B G A D I C E H

To determine the order of nodes visited in a level-order traversal...

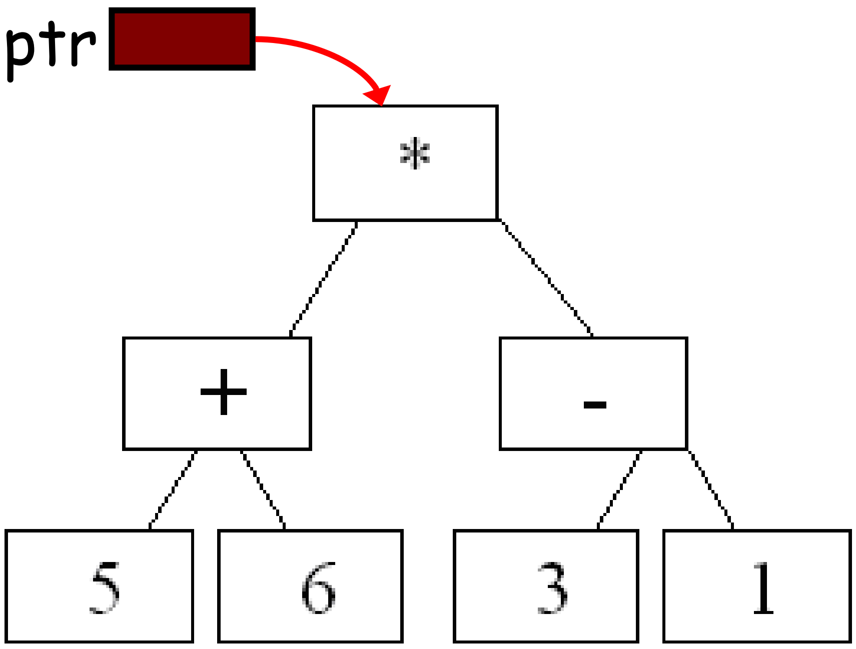
Start at the top node and draw a horizontal line left-to-right through all nodes on that row.

Repeat for all remaining rows.

The order you draw the lines is the order of the level-order traversal!

Expression Evaluation

We can represent arithmetic expressions using a binary tree.



For example, the tree on the left represents the expression: (5+6)*(3-1)

Once you have an expression in a tree, its easy to evaluate it and get the result.

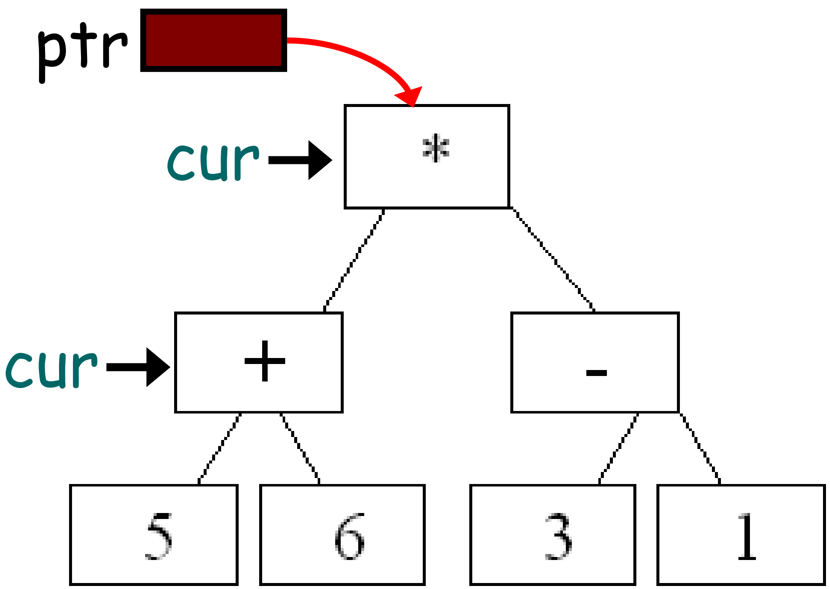
Let's see how!

Expression Evaluation

- Here's a function that computes the value of an expression tree. We start by passing in a pointer to the root of the tree.
- Step #1 is the base case. It checks for a number in a node, and if it finds one, just returns the value of the number. All leaf nodes will be numbers.
- Steps #2 and #3 evaluate the left and right subtrees of the current node, and gets their values.
- Then step #4 applies the operator (e.g., * sign) to the two results, and returns the overall result.
- This should look familiar! It's a post-order traversal in disguise. We process the left and right subtrees first, and then finally process the current node (the operator).

(5+6)*(3-1)

1. If the current node is a number, return its value.
2. Recursively evaluate the left subtree and get the result.
3. Recursively evaluate the right subtree and get the result.
4. Apply the operator in the current node to the left and right results; return the result.



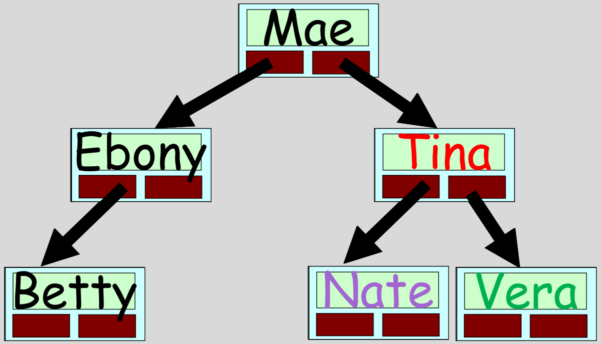
Binary Search Trees

Binary Search Trees

What's the big picture?

A binary search tree enables fast ($\log_2 N$) searches by ordering its data in a special way.

For <u>every</u> node j in the tree, all children to j's left must be less than it, and all children to j's right must be greater than it. e.g.,



To see if a value V is in the tree:

1. Start at the root node
2. Compare V against the node, moving down left or right if V is less or greater
3. Repeat until you find V or hit a dead end

Uses:

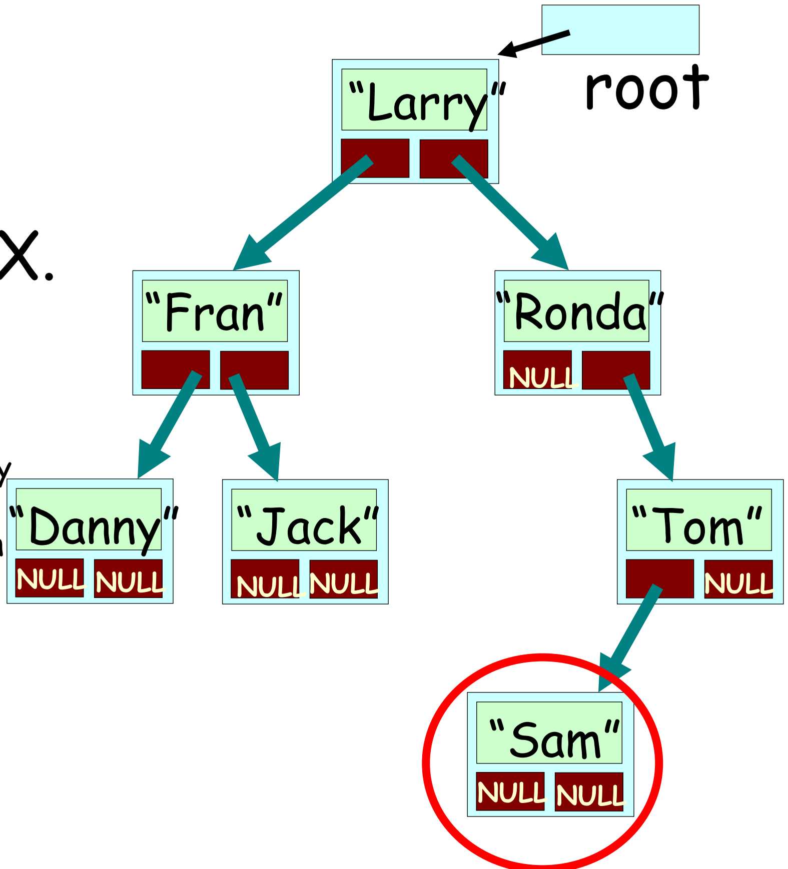
BSTs are used when you need to quickly search though loads of data, and also keep that data alphabetized.

Binary Search Trees

BST Definition: A Binary Search Tree is a binary tree with the following property:

For *every* node X in the tree:

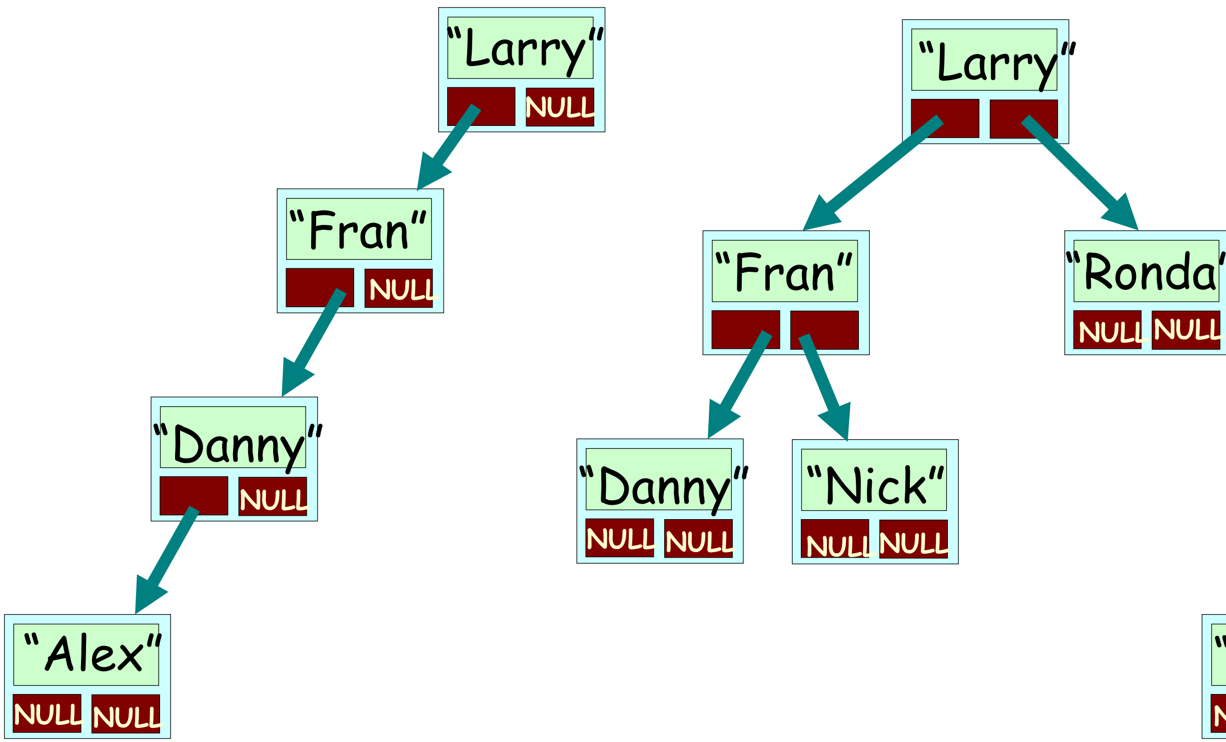
- All nodes in X's left sub-tree must be less than X.
- All nodes in X's right sub-tree must be greater than X.

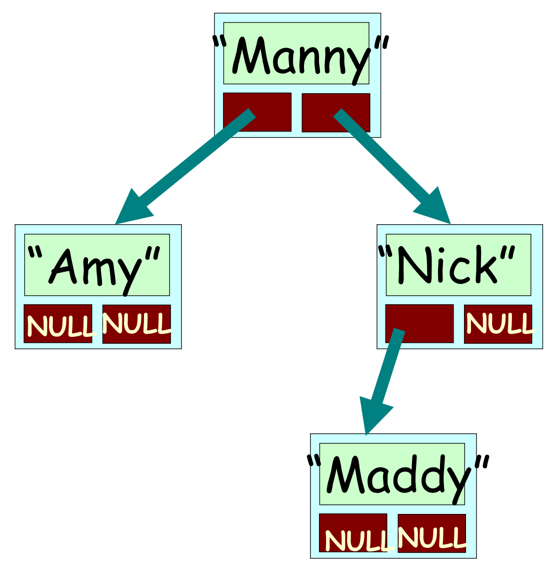


- The tree to the right is a valid binary search tree. Let's verify this.
- Every value in Larry's subtree is less than Larry. Every value in Larry's right subtree is greater than Larry.
- Every value in Fran's subtree is less than Fran. Every value in Fran's right subtree is greater than Fran.
- Every value in Rhonda's subtree is less than Rhonda (it's empty). Every value in Rhonda's right subtree is greater than Rhonda.
- The rest of the nodes are leaf nodes.

Binary Search Trees

Question: Which of the following are valid BSTs?





- From left to right:
  - Valid BST
  - Invalid BST: Nick is in Larry's left subtree, but Nick is > Larry!
  - Invalid BST: Maddy is less than Manny but is in Manny's right subtree.

Operations on a Binary Search Tree

Here's what we can do to a BST:

- Determine if the binary search tree is empty
- Search the binary search tree for a value
- Insert an item in the binary search tree
- Delete an item from the binary search tree
- Find the height of the binary search tree
- Find the number of nodes and leaves in the binary search tree
- Traverse the binary search tree
- Free the memory used by the binary search tree

Searching a BST

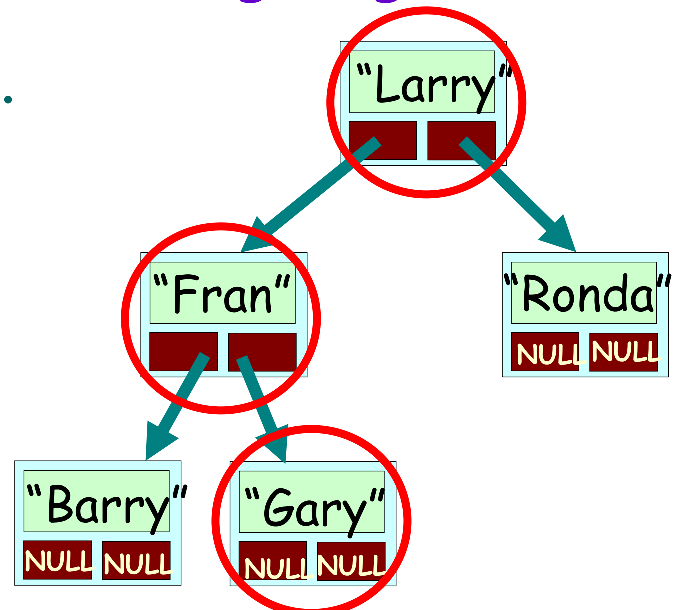
Input: A value V to search for
Output: TRUE if found, FALSE otherwise

Start at the root of the tree
Keep going until we hit the NULL pointer

If V is equal to current node's value, then found!
If V is less than current node's value, go left
If V is greater than current node's value, go right

If we hit a NULL pointer, not found.

- Let's search for Gary.
- We start by comparing Gary to Larry, and find Gary is less than Larry, so we go left.
- We then compare Gary to Fran, and find Gary is greater than Fran, so we go right.
- We then compare Gary to Gary, and find they're equal. We found our value!



Searching a BST

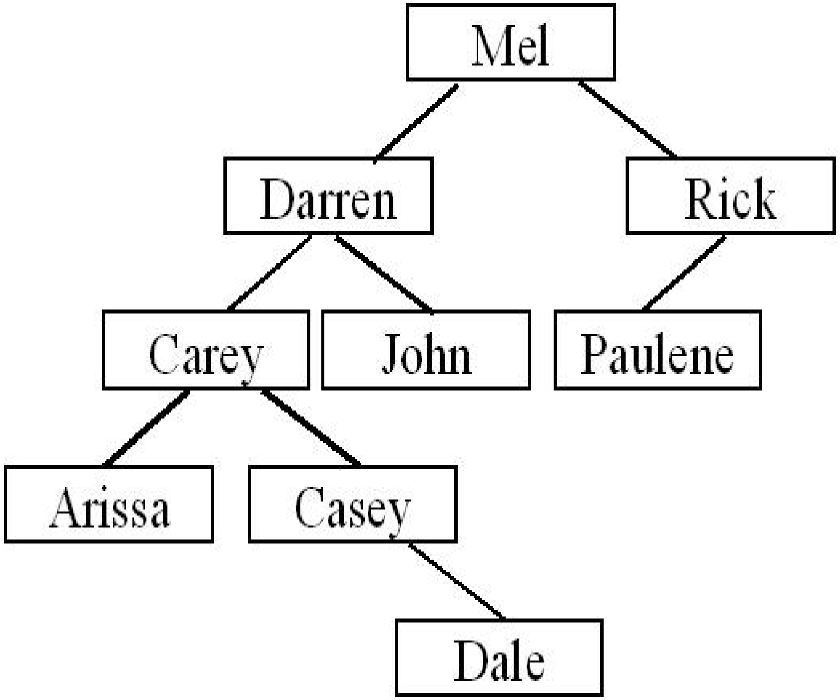
Start at the root of the tree
Keep going until we hit the NULL pointer

- If V is equal to current node's value, then found!
- If V is less than current node's value, go left
- If V is greater than current node's value, go right

If we hit a NULL pointer, not found.

Show how to search for:

1. Khang
2. Dale
3. Sam



Searching a BST

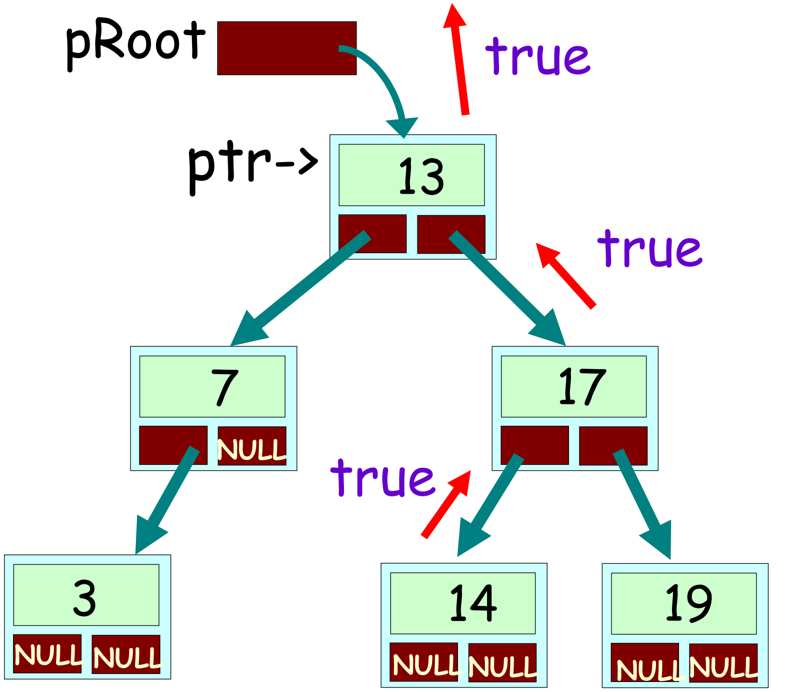
Here are two different BST search algorithms in C++, one iterative and one recursive:

```
bool Search(int v, Node *root)
{
  Node *ptr = root;
  while (ptr != nullptr)
  {
    if (v == ptr->value)
      return true;
    else if (v < ptr->value)
      ptr = ptr->left;
    else
      ptr = ptr->right;
  }
  return false; // nope
}
```

```
bool Search(int v, Node *ptr)
{
  if (ptr == nullptr)
    return false;  // nope
  else if (v == ptr->value)
    return(true);  // found!!!
  else if (v < ptr->value)
    return Search(V,ptr->left);
  else
    return Search(V,ptr->right);
}
```

Recursive BST Search

Lets search for 14.



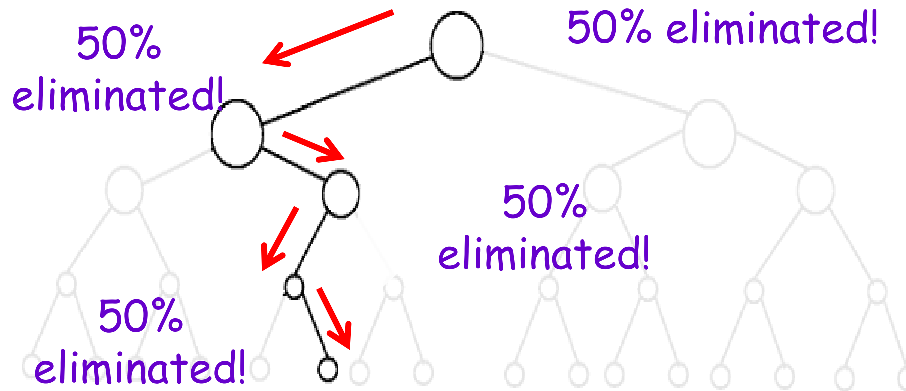

```
bool Search(int V, Node *ptr)
{
  if (ptr == nullptr)
    return(false);  // nope
  else if (V == ptr->value)
    return(true);  // found!!!
  else if (V < ptr->value)
    return(Search(V,ptr->left));
  else
    return(Search(V,ptr->right));
}
```

```
int main(void)
{
  bool bFnd;
  bFnd = Search(14,pRoot);
}
```

Big Oh of BST Search

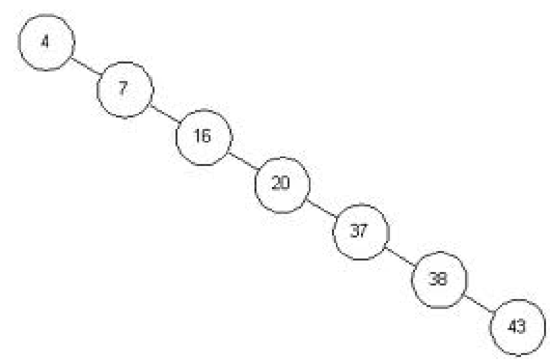
Question:
In the average BST with N values, how many steps are required to find our value?

Right! $\log_2(N)$ steps



Question:
In the worst case BST with N values, how many steps are required find our value?

Right! N steps



Question:
If there are 4 billion nodes in a BST, how many steps will it take to perform a search?

Just 32!

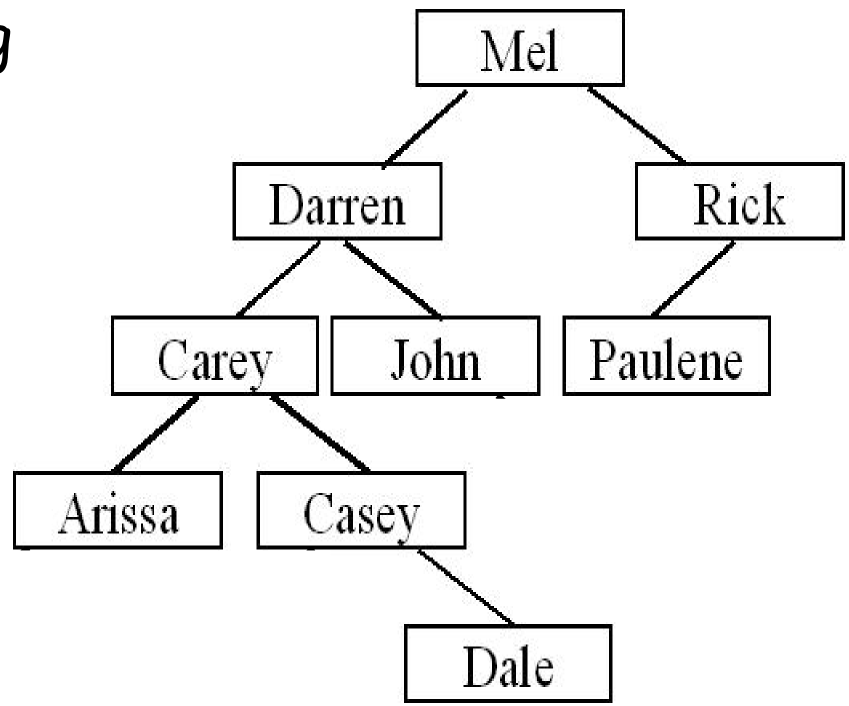
WOW!
Now that's PIMP!

Inserting A New Value Into A BST

To insert a new node in our BST, we must place the new node so that the resulting tree is still a valid BST!

Where would the following new values go?

Carly
Ken
Alice



- Answers:
  - Karly would be added as Casey's left child.
  - Ken would be added as John's right child
  - Alice would be added as Arissa's left child.

Inserting A New Value Into A BST

Input: A value V to insert

If the tree is empty
- Allocate a new node and put V into it
- Point the root pointer to our new node. DONE!

Start at the root of the tree
While we're not done…
- If V is equal to current node's value, DONE! (nothing to do…)
- If V is less than current node's value
  - If there is a left child, then go left
  - ELSE allocate a new node and put V into it, and
    set current node's left pointer to new node. DONE!
- If V is greater than current node's value
  - If there is a right child, then go right
  - ELSE allocate a new node and put V into it,
    set current node's right pointer to new node. DONE!

Now the C++ Code!

```
struct Node
{
  Node(const std::string &myVal)
   {
     value = myVal;
     left = right = nullptr;
   }

  std::string   value;
  Node        *left,*right;
};
```

- Just as with a regular binary tree, we use a node struct to hold our items.
- However let's add a constructor to our Node so we can easily create a new one!
- And our constructor initializes that root pointer to nullptr
  when we create a new tree. (This indicates the tree is empty)

Now the C++ Code!

```
class BinarySearchTree
{
public:

   BinarySearchTree()
   {
      m_root = nullptr;
   }

   void insert(const std::string &value)
   {
      ...
   }

private:

   Node *m_root;
};
```

- And here's our Binary Search Tree class.
- Our BST class has a single member variable – the root pointer to the tree.
- And our constructor initializes that root pointer to nullptr when we create a new tree. (This indicates the tree is empty)
- Now let's see our complete insertion function in C++.

```
void insert(const std::string &value)
{
   if (m_root == nullptr)
      {   m_root = new Node(value);    return; }

   Node *cur = m_root;
   for (;;)
    {
       if (value == cur->value)    return;
       if (value < cur->value)
       {
          if (cur->left != nullptr)
              cur = cur->left;
          else
          {
              cur->left = new Node(value);
              return;
          }
        }
        else if (value > cur->value)
        {
           if (cur->right != nullptr)
               cur = cur->right;
           else
           {
               cur->right = new Node(value);
               return;
           }
        }
    }
}
```

Co

If our tree is empty, allocate a new node and point the root pointer to it – then we're done!

Start traversing down from the root of the tree.
for(;;) is the same as an infinite loop.

If our value is already in the tree, then we're done - just return.

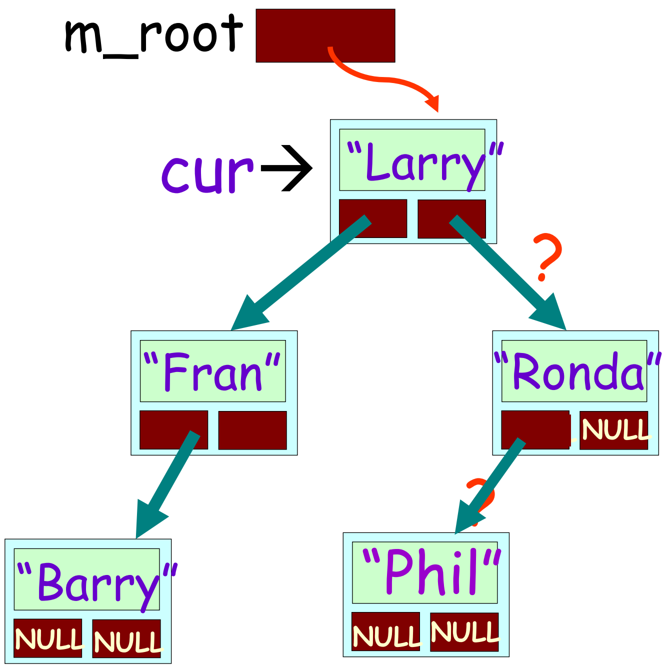
If the value to insert is less than the current node's value, then go left.

If there is a node to our left, advance to that node and continue.

Otherwise we've found the proper spot for our new value! Add our value as the left child of the current node.

If the value we want to insert is greater than the current node's value, then traverse/insert to the right.

```
void insert(const std::string &value)
{
   if (m_root == NULL)
     {   m_root = new Node(value);    return; }

   Node *cur = m_root;
   for (;;)
    {
       if (value == cur->value)    return;

       if (value < cur->value)
       {
           if (cur->left != NULL)
               cur = cur->left;
           else
           {
               cur->left = new Node(value);
               return;
           }
        }
        else if (value > cur->value)
        {
            if (cur->right != NULL)
                cur = cur->right;
            else
            {
                cur->right = new Node(value);
                return;
            }
        }
    }
}
```




```
void main(void)
{
   BinarySearchTree bst;
   bst.insert("Larry");
   ...
   bst.insert("Phil");
}
```

Inserting A New Value Into A BST

As with BST Search, there is a recursive version of the Insertion algorithm too. Be familiar with it!

Question:
Given a random array of numbers if you insert them one at a time into a BST, what will the BST look like?

Question:
Given a ordered array of numbers if you insert them one at a time into a BST, what will the BST look like?

- Answers:
  - Random: The numbers will form a "bushy" wide binary tree.
  - Ordered: The numbers will look like a linked list going either left (ordered in descending order) or right (ordered in ascending order).

Big Oh of BST Insertion

So, what's the big-oh of BST Insertion?

Right! It's also $O(\log_2 n)$

Why? Because we have to first use a binary search to find where to insert our node and binary search is $O(\log_2 n)$.

Once we've found the right spot, we can insert our new node in $O(1)$ time.

Groovy Baby!

Finding Min & Max of a BST

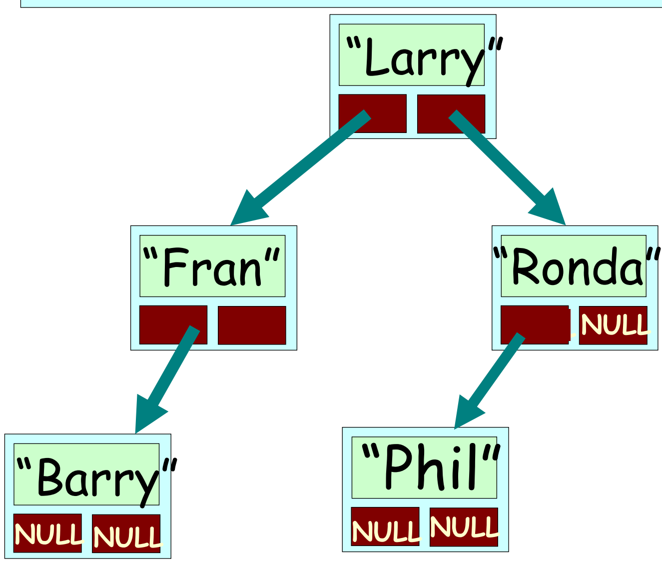
How do we find the minimum and maximum values in a BST?

The minimum value is located at the left-most node.

The maximum value is located at the right-most node.

```
int GetMin(node *pRoot)
{
   if (pRoot == NULL)
      return(-1);   // empty

   while (pRoot->left != NULL)
       pRoot = pRoot->left;

   return(pRoot->value);
}
```

```
int GetMax(node *pRoot)
{
   if (pRoot == NULL)
      return(-1);   // empty

   while (pRoot->right != NULL)
       pRoot = pRoot->right;

   return(pRoot->value);
}
```



Question: What's the big-oh to find the minimum or maximum element?

Answer: $O(\log_2(N))$

Finding Min & Max of a BST

And here are recursive versions for you...

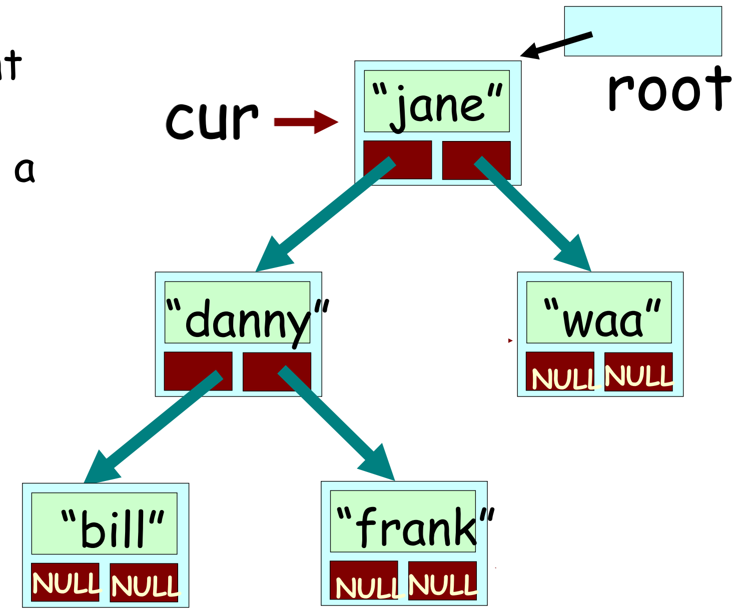
```
int GetMin(node *pRoot)
{
   if (pRoot == NULL)
     return(-1);   // empty

   if (pRoot->left == NULL)
     return(pRoot->value);

   return(GetMin(pRoot->left));
}
```

```
int GetMax(node *pRoot)
{
   if (pRoot == NULL)
     return(-1);   // empty

   if (pRoot->right == NULL)
     return(pRoot->value);

   return(GetMax(pRoot->right));
}
```

Hopefully you're getting the idea that most tree functions can be done recursively...

Printing a BST In Alphabetical Order

- Can anyone guess what algorithm we use to print out a BST in alphabetical order?
- You guessed it! Using an "in-order" traversal on a binary search tree will print it in alphabetical order! Neat!



Big-oh Alert!

So what's the big-Oh of printing all the items in the tree?

Right! O(n) since we have to visit and print all n items.

Output:

bill
danny
frank
jane
waa

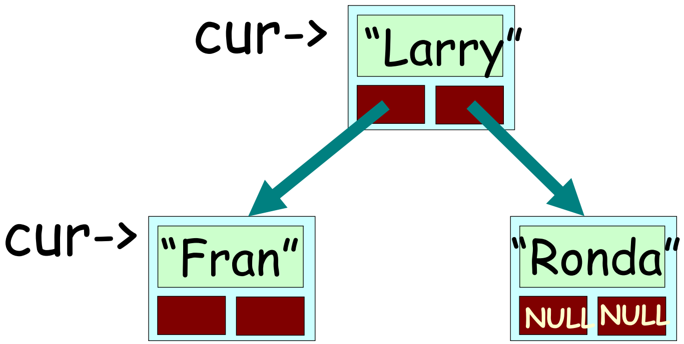
Freeing The Whole Tree

When we are done with our BST, we have to free every node in the tree, one at a time.

- The algorithm to free the whole tree is below.
- It is basically a post-order traversal.
- First we free the left subtree of the current node.
- Then we free the right subtree of the current node.
- Then we delete the current node once all of its children are gone.

```
void FreeTree(Node *cur)
{
   if (cur == nullptr)          // if empty, return...
        return;

   FreeTree(cur->left);     // Delete nodes in left sub-tree.

   FreeTree (cur->right);  // Delete nodes in right sub-tree.

   delete cur;                  // Free the current node
}
```

Freeing The Whole Tree




```
void FreeTree(Node *cur)
{
    if (cur == nullptr)
        return;

    FreeTree(cur->left);

    FreeTree (cur-> right);

   delete cur;
}
```

Big-oh Alert!

So what's the big-Oh of freeing all the items in the tree?

It's still O(n) since we have to visit all n items.