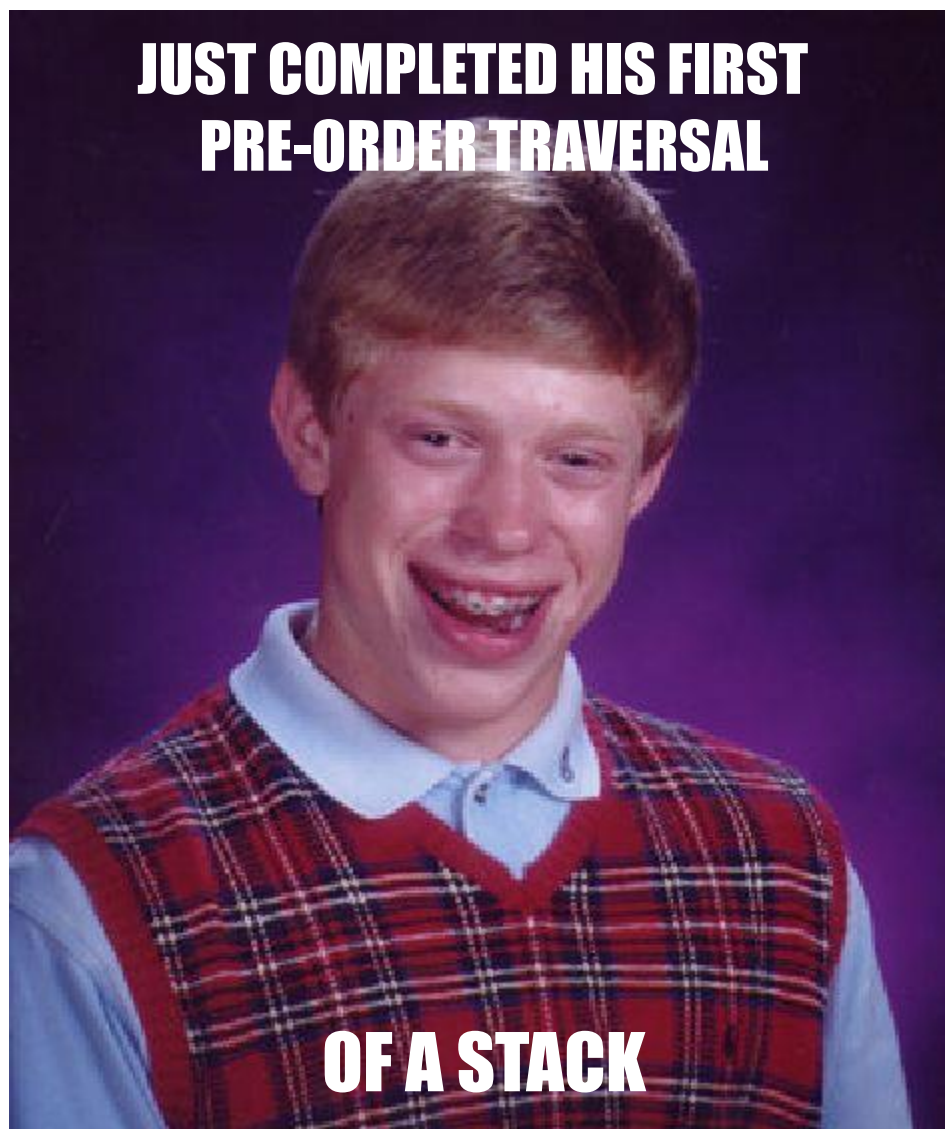


Lecture #12

- Binary Tree Traversals
- Evaluate Expressions Using Binary Trees
- Binary Search Trees
 - 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.

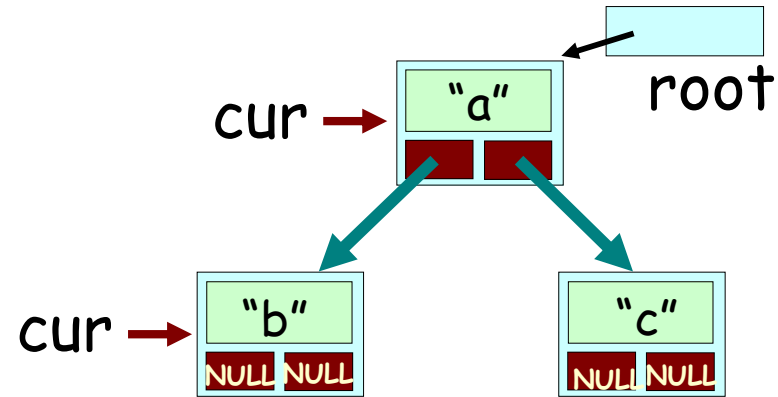
There are four common ways to traverse a tree.

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 In-order Traversal

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



```

void InOrder(Node *cur)
{
    if (cur == NULL)          // if empty, return...
        return;

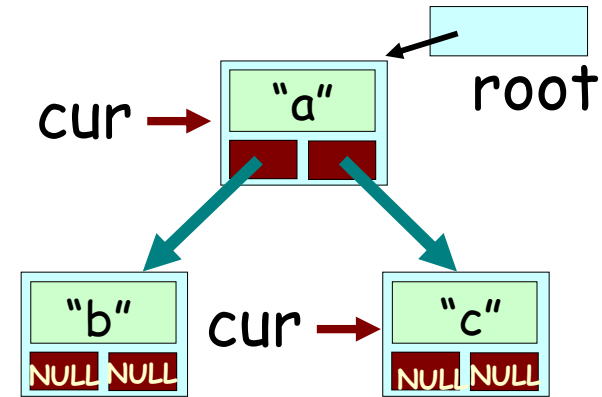
    InOrder(cur->left);        // Process nodes in left sub-tree.
    cout << cur->value;        // Process the current node.
    InOrder(cur->right);       // Process nodes in right sub-tree.
}
  
```

Output:

b

The In-order Traversal

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



```

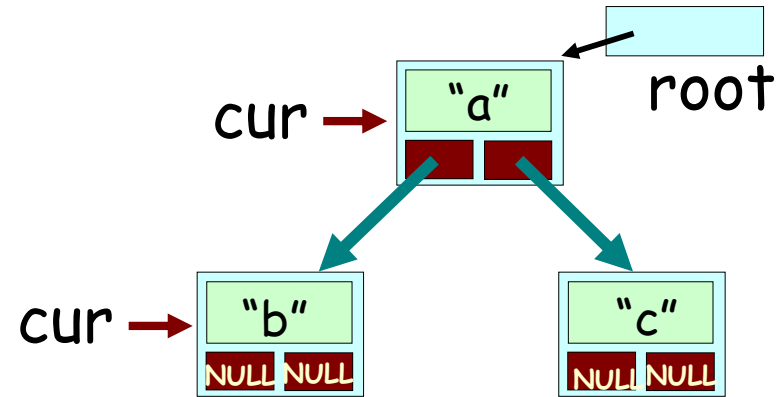
void InOrder(Node *cur)
{
    if (cur == NULL)          // if empty, return...
        return;

    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

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



```

void PostOrder(Node *cur)
{
    if (cur == NULL)          // if empty, return...
        return;

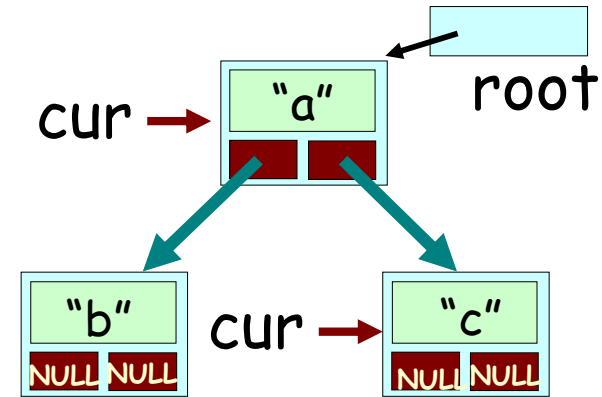
    PostOrder(cur->left);    // Process nodes in left sub-tree.
    PostOrder(cur->right);   // Process nodes in right sub-tree.
    cout << cur->value;      // Process the current node.
}
  
```

Output:

b

The Post-order Traversal

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



```

void PostOrder(Node *cur)
{
    if (cur == NULL)          // 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

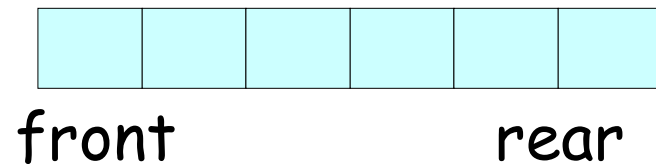
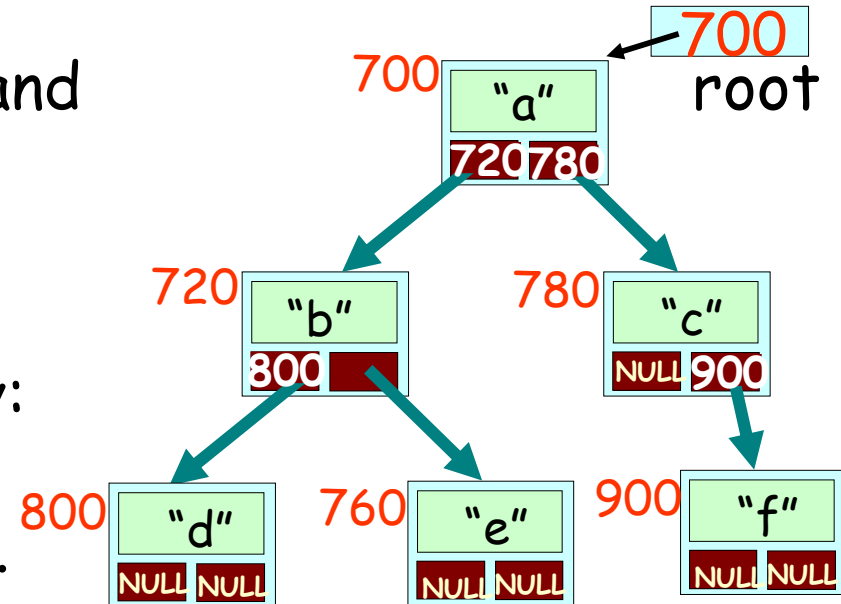
In a *level order traversal* we visit each level's nodes, from left to right, before visiting nodes in the next level.

Here's the algorithm:

temp

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.

abcd Etc...



Big-Oh of Traversals?

Question:

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

Answer:

Well, since a traversal **must** process each node exactly once...
and since there are **n nodes** in a tree...
the big-oh for any of the traversals is...

Each of our traversals
processes each node
just **once**.

$O(n)$

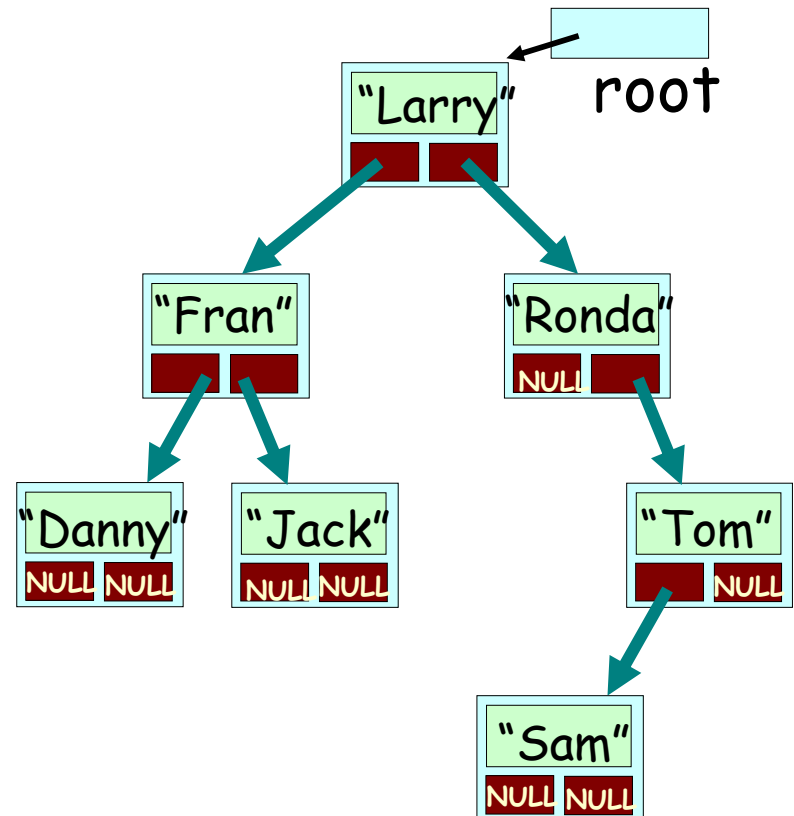
```
void PreOrder(Node *cur)
{
    if (cur == NULL)
        return;
    cout << cur->value;
    PreOrder(cur->left);
    PreOrder(cur->right);
}
```

Traversal Challenge

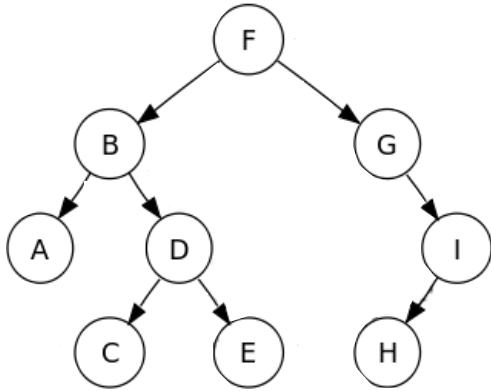
RULES

- 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**?



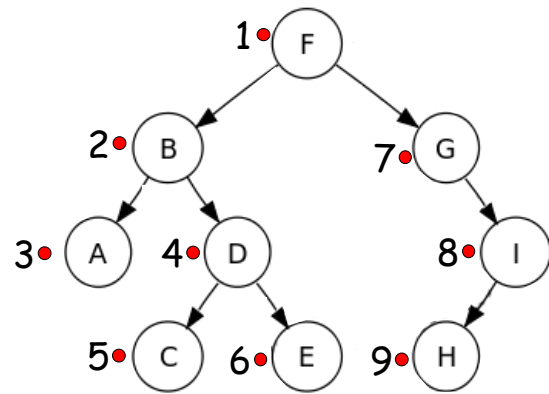
An Easy Way to Remember the Order of Pre/In/Post Traversals



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**



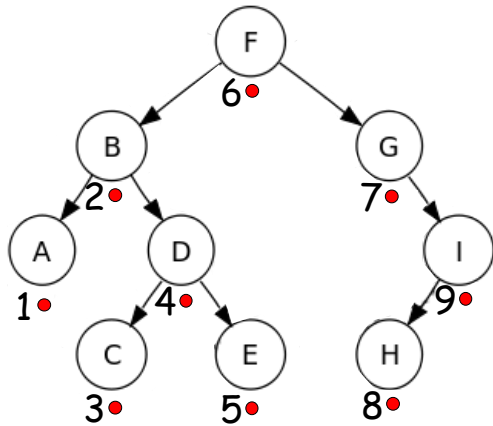
Pre-order:
F B A D C E G I H

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



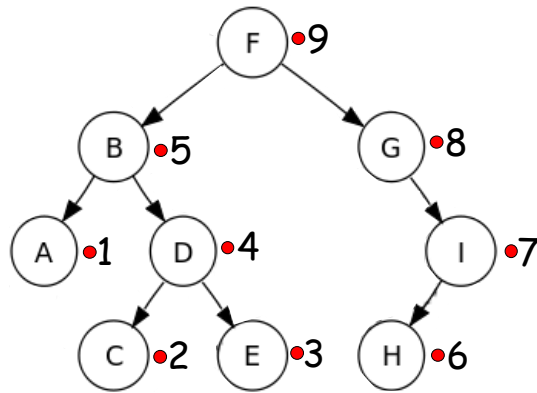
In-order:
A B C D E F G H I

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**



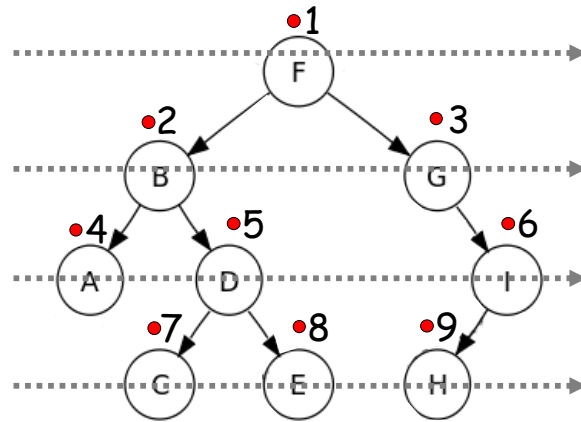
Post-order:
A C E D B H I G F

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**...

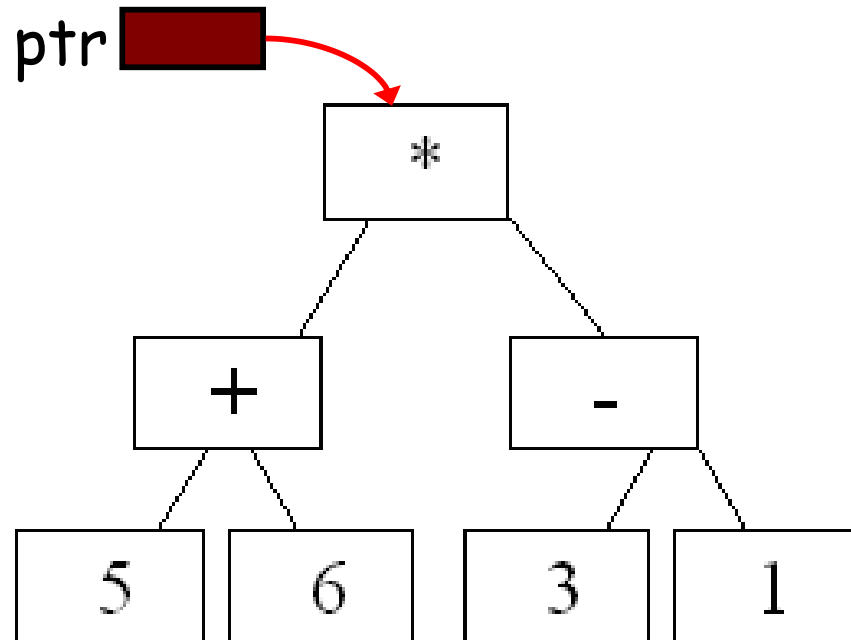
Start at the top node and **draw a horizontal line left-to-right** through all nodes on that row.

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.

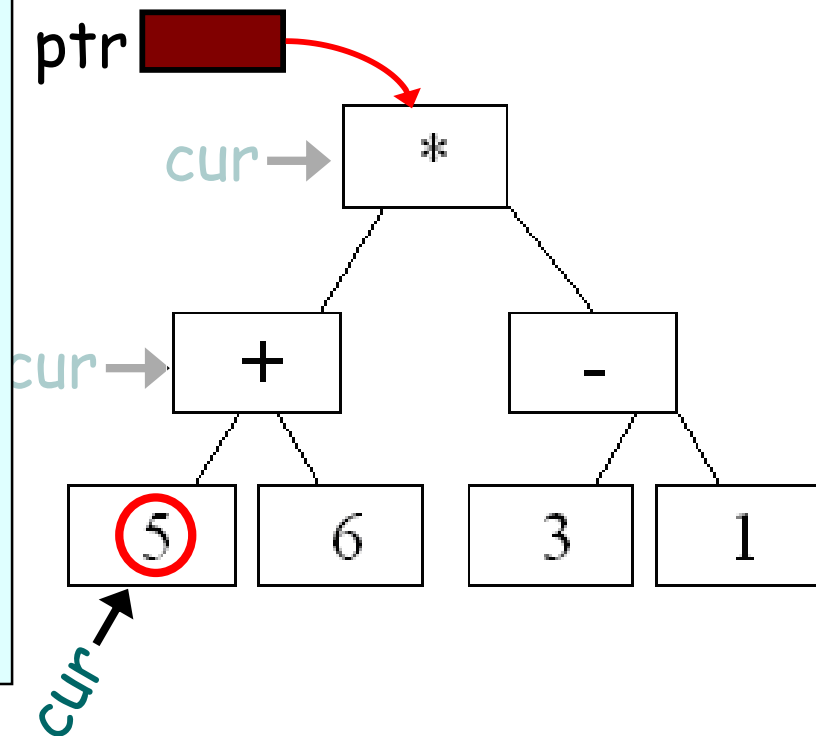
Let's see how!

Expression Evaluation

Here's our evaluation function. We start by passing in a pointer to the root of the tree.

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.

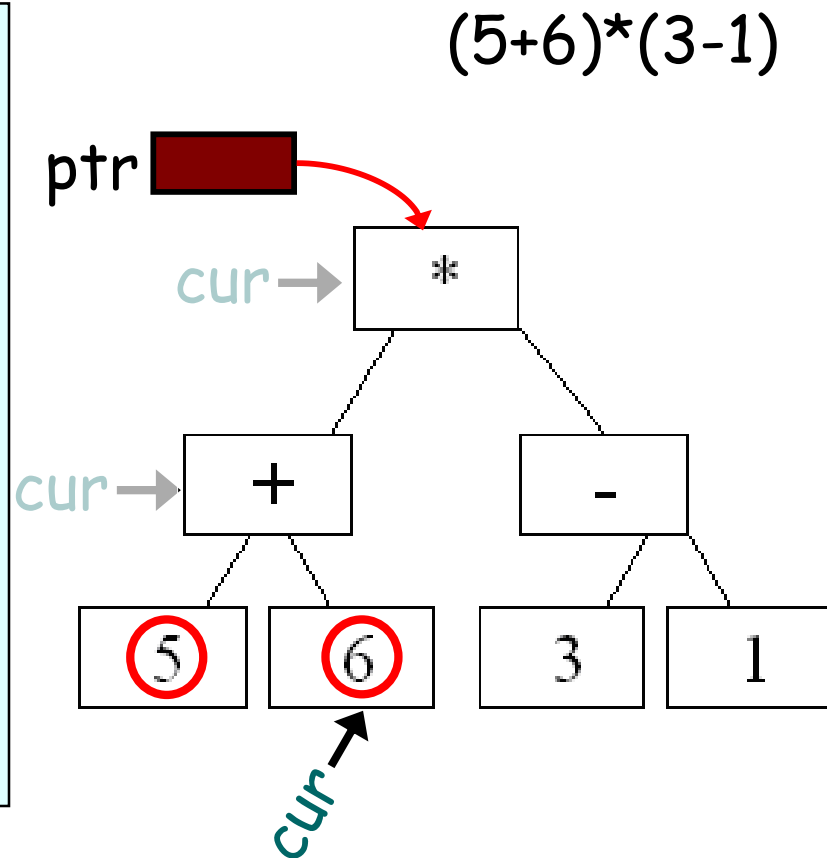
$(5+6)*(3-1)$



Expression Evaluation

Here's our evaluation function. We start by passing in a pointer to the root of the tree.

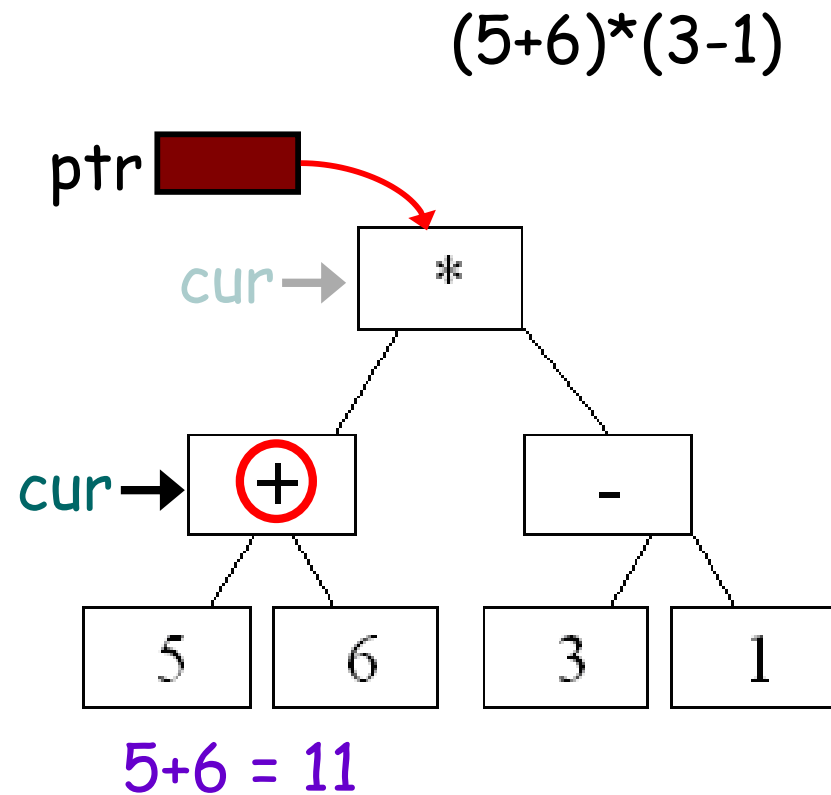
1. If the current node is a number, return its value.
2. Recursively evaluate the left subtree and get the result.
3. Recursively evaluate the right subtree and get the result.
4. Apply the operator in the current node to the left and right results; return the result.



Expression Evaluation

Here's our evaluation function. We start by passing in a pointer to the root of the tree.

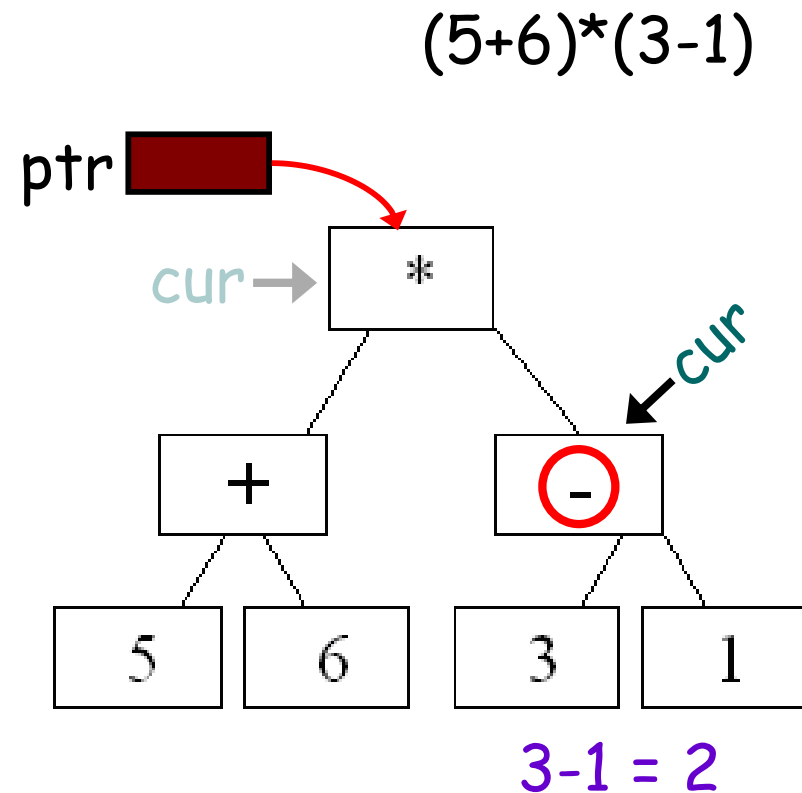
1. If the current node is a number, return its value.
2. Recursively evaluate the left subtree and get the result.
Result = 5
3. Recursively evaluate the right subtree and get the result.
Result = 6
4. Apply the operator in the current node to the left and right results; return the result.



Expression Evaluation

Here's our evaluation function. We start by passing in a pointer to the root of the tree.

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

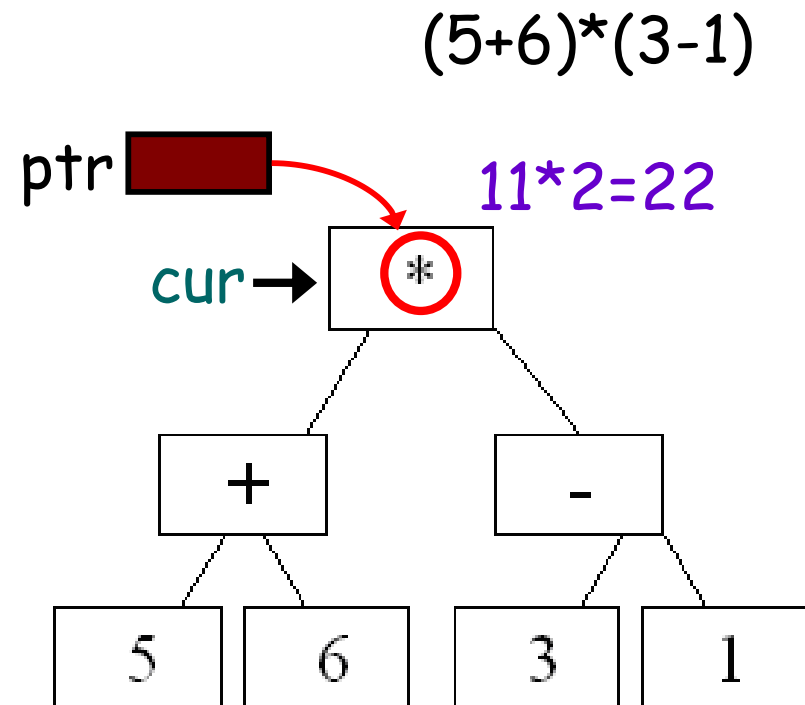


Expression Evaluation

Here's our evaluation function. We start by passing in a pointer to the root of the tree.

The result is 22.

1. If the current node is a number, return its value.
2. Recursively evaluate the left subtree and get the result.
Result = 11
3. Recursively evaluate the right subtree and get the result.
Result = 2
4. Apply the operator in the current node to the left and right results; return the result.



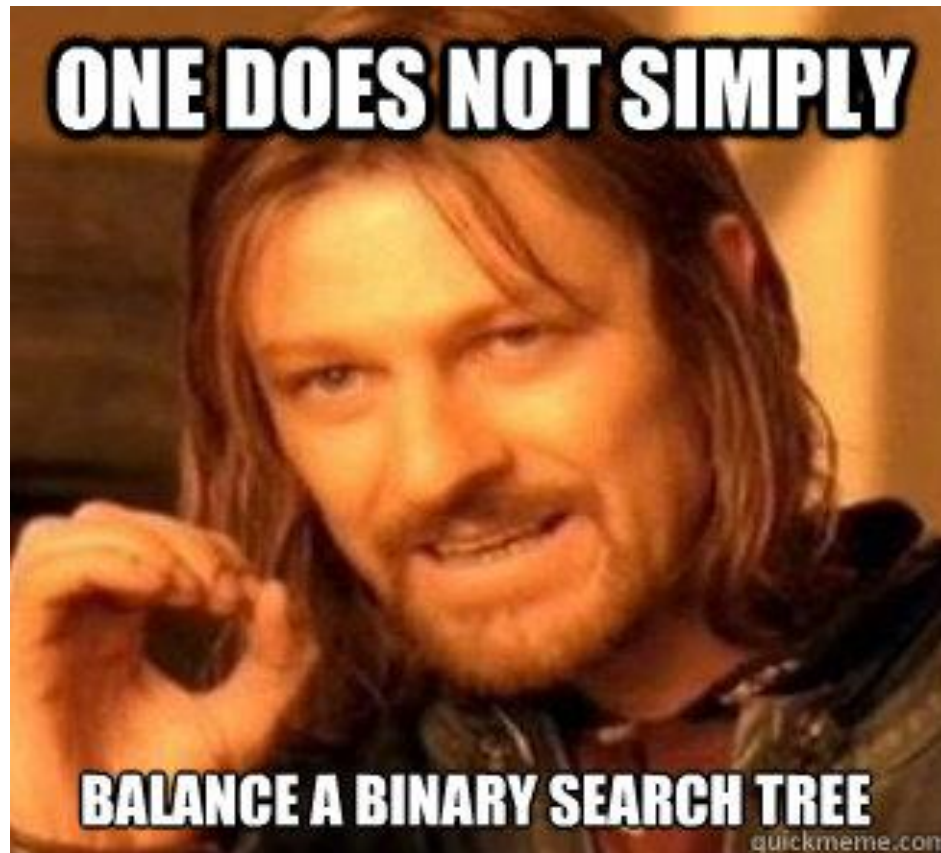
Expression Evaluation

Here's our evaluation function. We start by passing in a pointer to the root of the tree.

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.

Question: Which other algorithm does this remind you of?

Binary Search Trees



Binary Search Trees

Why should you care?

Binary Search Trees are an extremely efficient way to store/search for data!

You can search a BST with billions of items in just microseconds!

They're used in databases, operating systems, search engines, etc.

And because you'll be asked about them in **job interviews** and on **exams**.

So pay attention!

Why
should
I care?

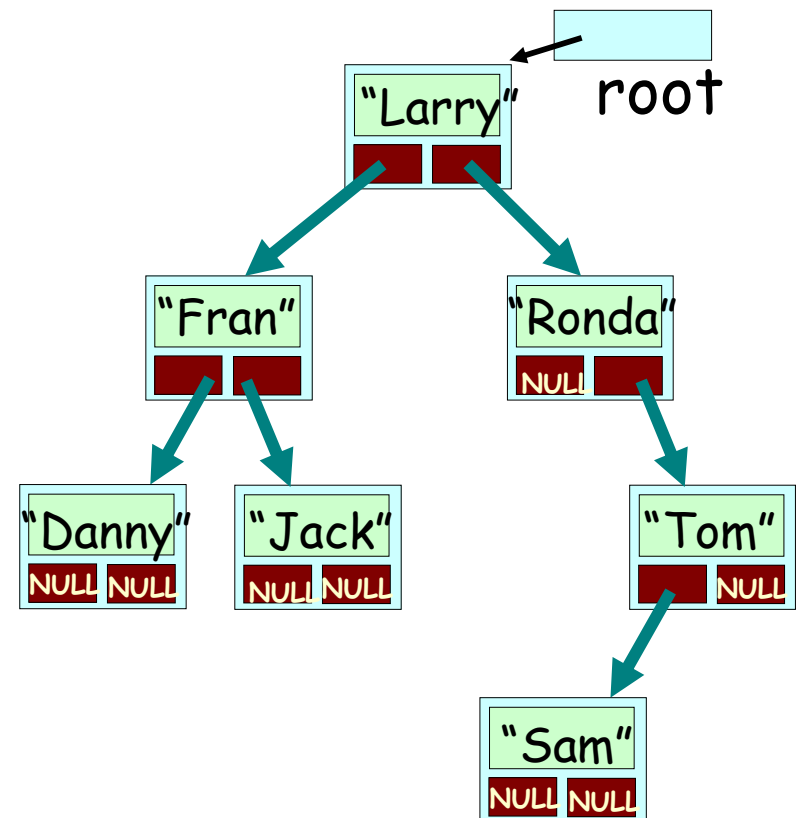


Binary Search Trees

Binary Search Trees are a type of binary tree with specific properties that make them very efficient to search for a value in the tree.

Like regular Binary Trees, we store and search for values in Binary Search Trees...

Here's an example BST...



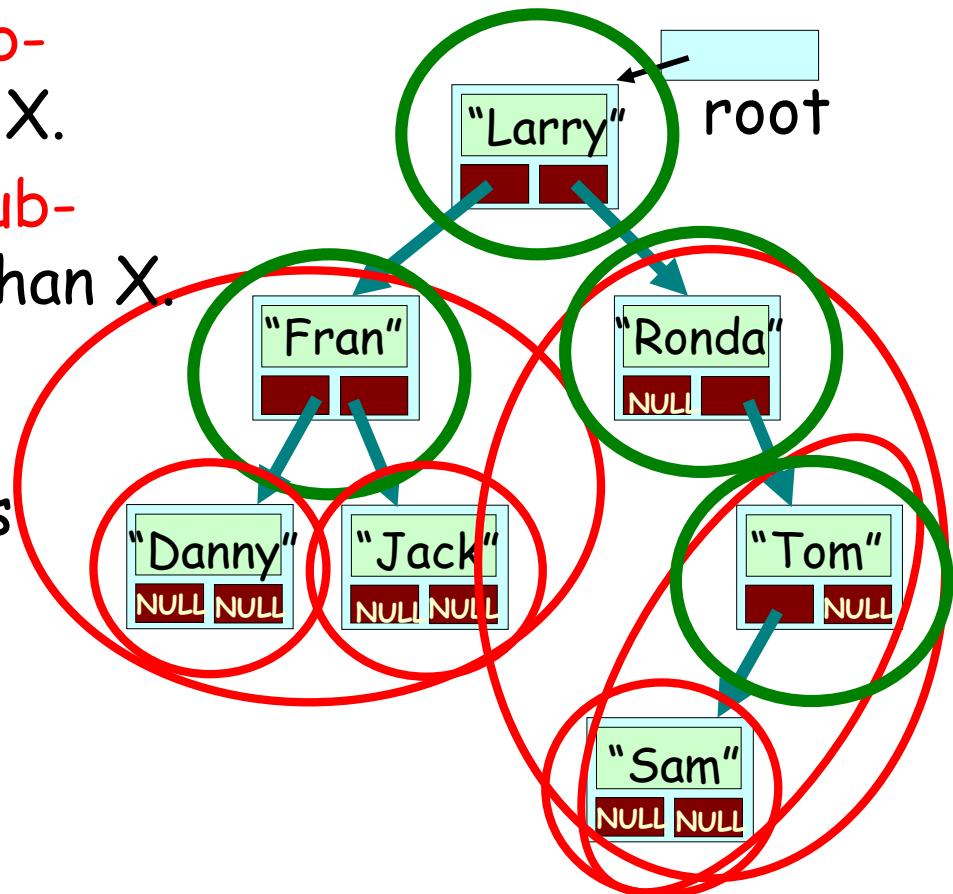
Binary Search Trees

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

For every **node X** in the tree:

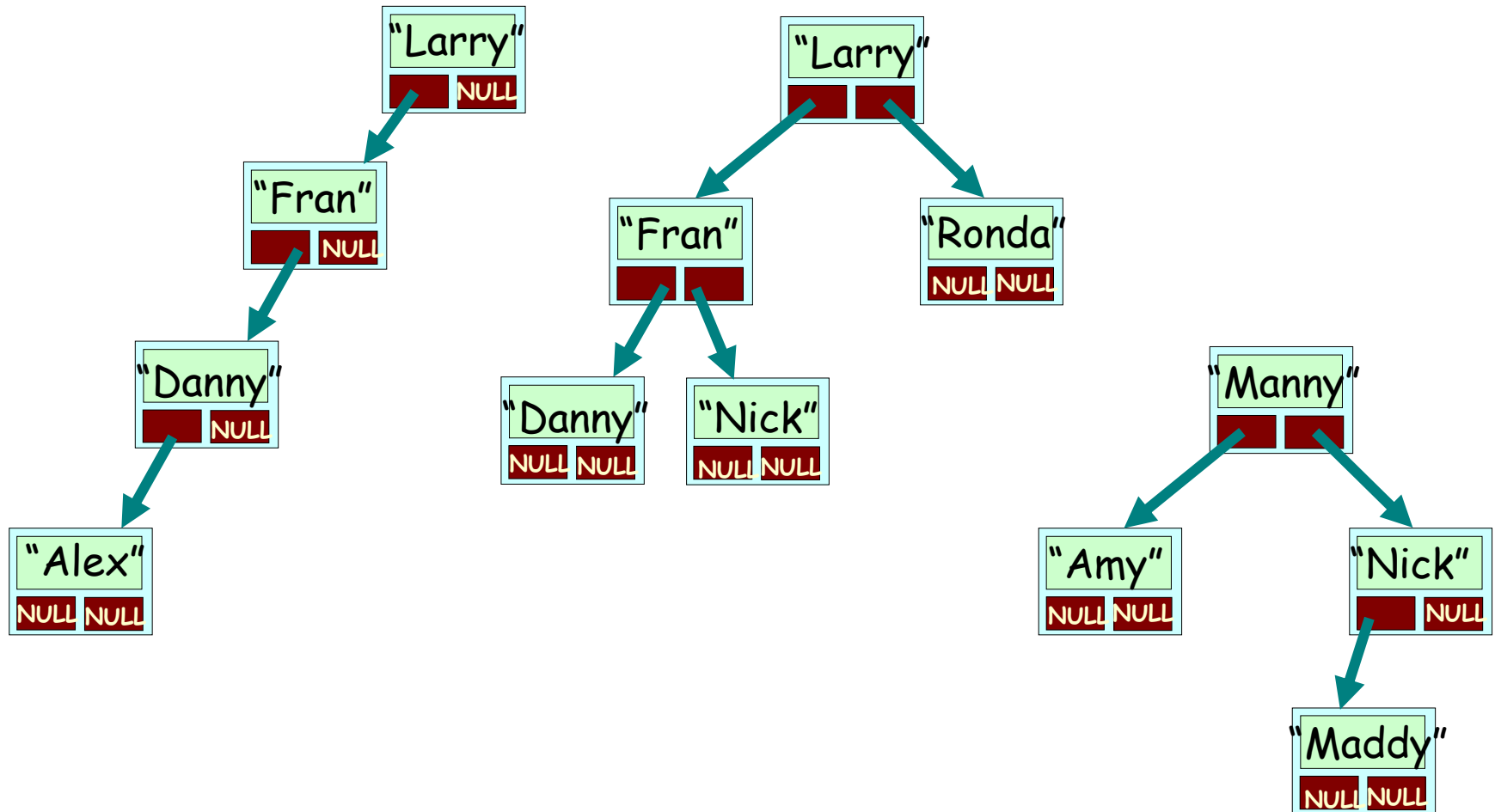
- All nodes in X's **left subtree** must be **less** than X.
- All nodes in X's **right subtree** must be **greater** than X.

Let's validate that this is a valid BST...



Binary Search Trees

Question: Which of the following are valid BSTs?



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.

Gary == Larry??

Gary < Larry??

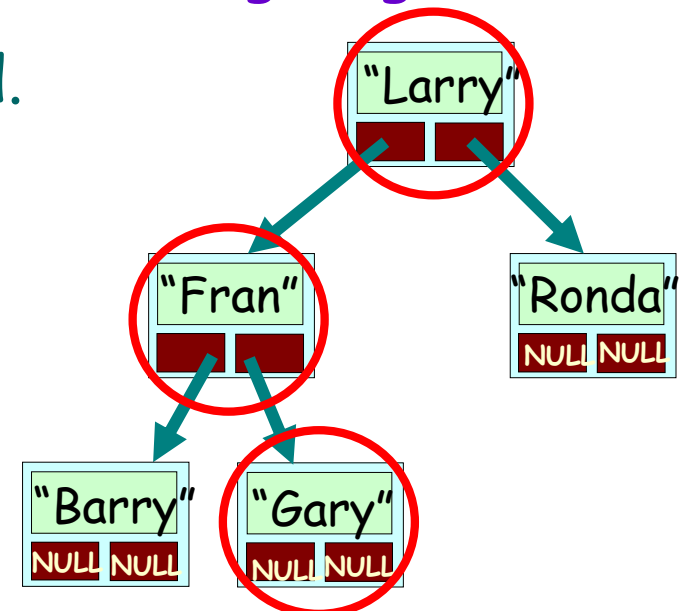
Gary == Fran??

Gary < Fran??

Gary > Fran??

Gary == Gary??

Let's search
for **Gary**.



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!**

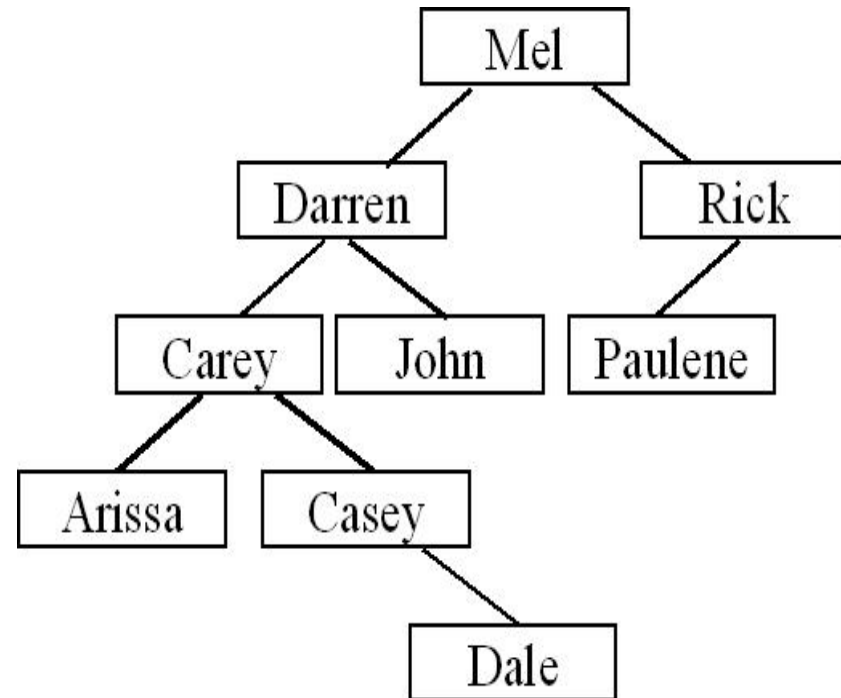
If V is **less** than current node's value, **go left**

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 recursive and one iterative:

```
bool Search(int V, Node *ptr)
{
    if (ptr == NULL)
        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));
}
```

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

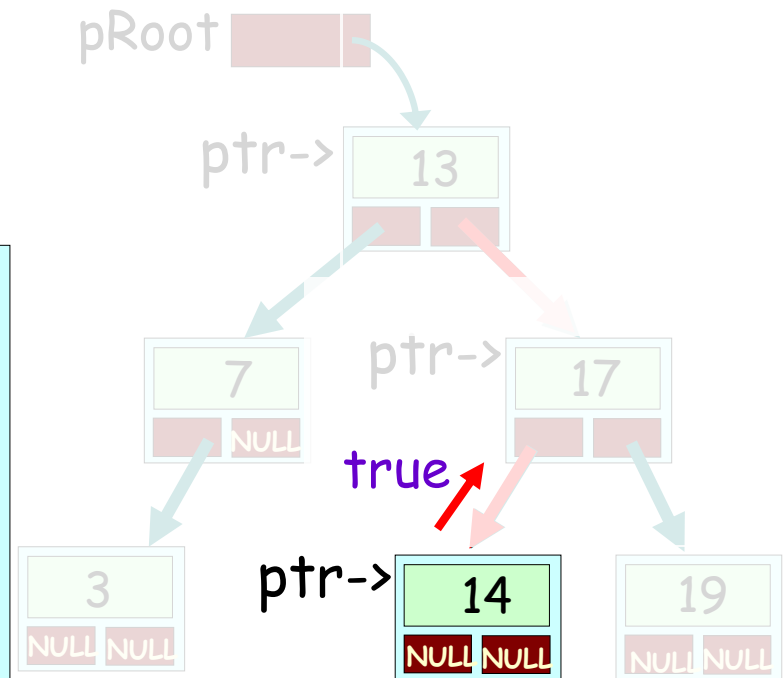
Let's trace through the recursive version...

Recursive BST Search

Lets search for 14.

```
bool Search(int V, Node *ptr)
{
    if (ptr == NULL)
        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));
}

return(Search(V, ptr->right));
}
```

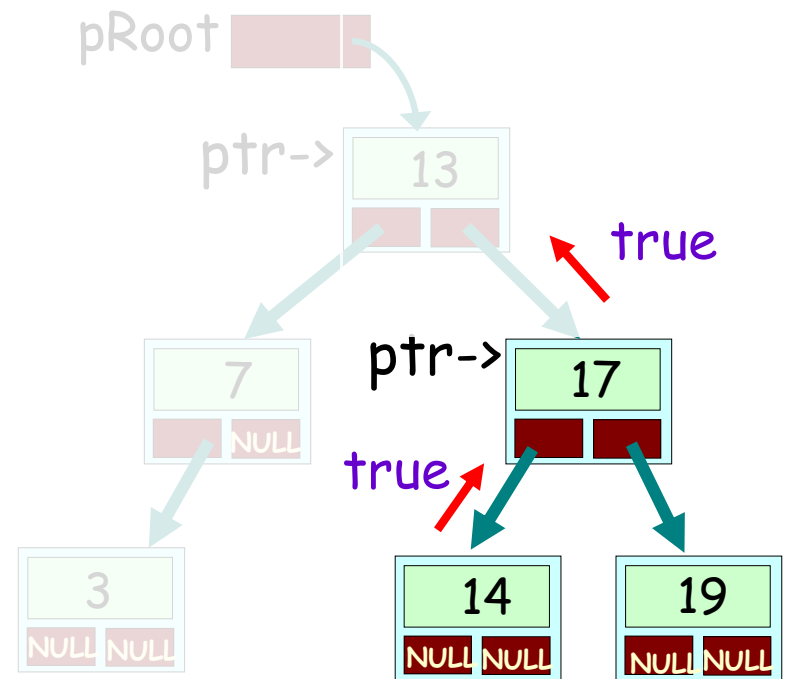


```
void main(void)
{
    bool bEnd;
    bEnd = Search(14, pRoot);
}
```


Recursive BST Search

Lets search for 14.

```
bool Search(int V, Node *ptr)
{
    if (ptr == NULL)
        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));
}
```

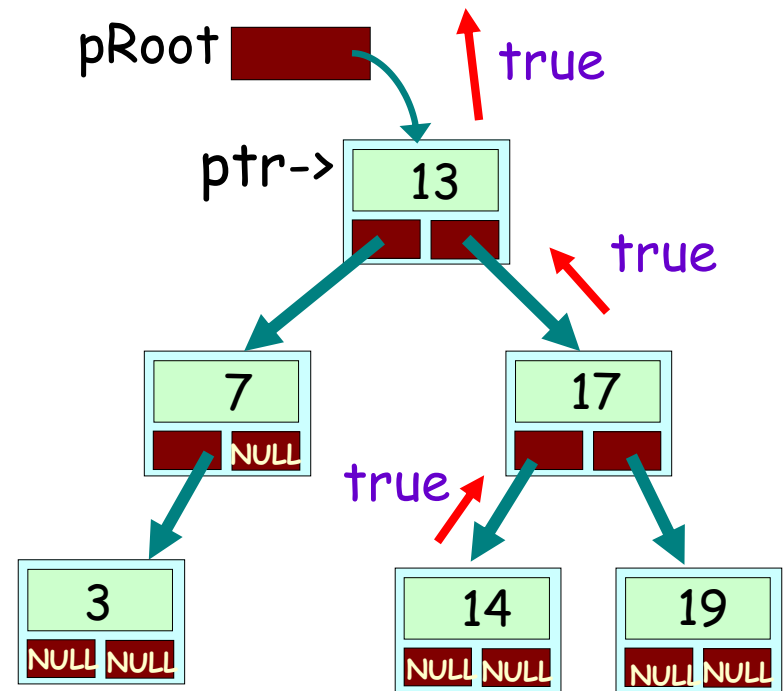


```
void main(void)
{
    bool bEnd;
    bEnd = Search(14, pRoot);
}
```

Recursive BST Search

Lets search for 14.

```
bool Search(int V, Node *ptr)
{
    if (ptr == NULL)
        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 bEnd;
    bEnd = 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:

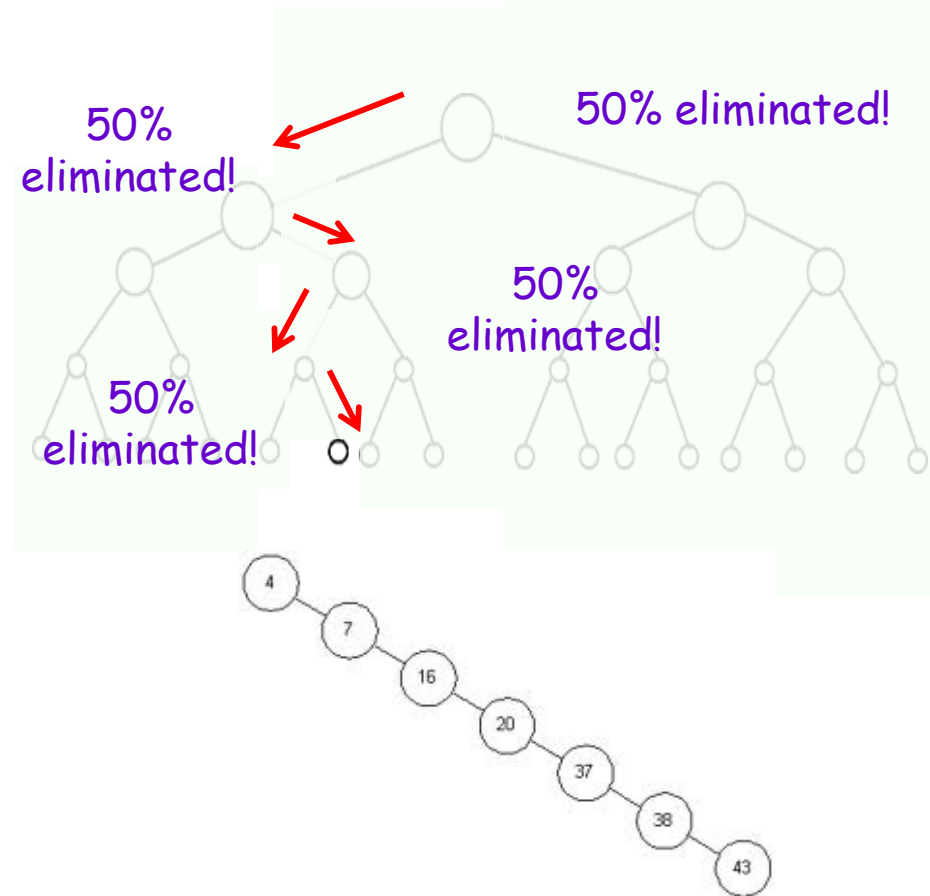
In the worst case BST with **N values**, how many steps are required find our value?

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!

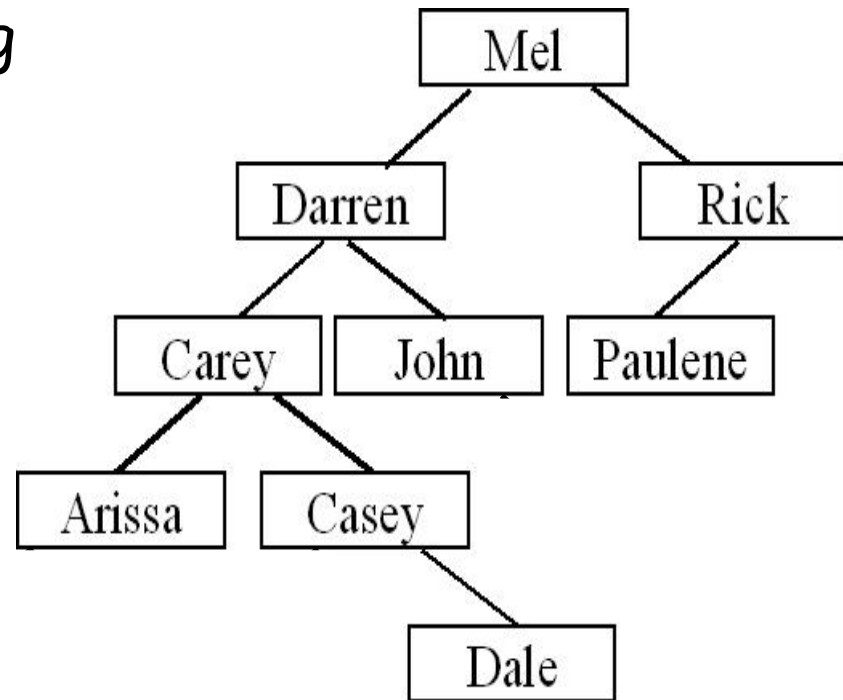
Now that's PIMP!

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



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!

```

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

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

```

```

class BinarySearchTree
{
public:
    BinarySearchTree()
    {
        m_root = NULL;
    }

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

private:
    Node *m_root;
};

```

And our constructor initializes that **root pointer** to **NULL** when we create a new tree. (This indicates the tree is empty)

Our BST class has a **single member variable** - the root pointer to the tree.

```

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;

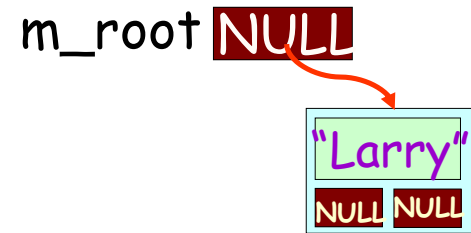
        if (value < cur->value)
        {
            if (cur->left != NULL)
                cur = cur->left;
            else
            {
                cur->left = new Node(value);
                return;
            }
        }
        else if (value > cur->value)
        {
            if (cur->right !=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

```

```

void 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");

    ...

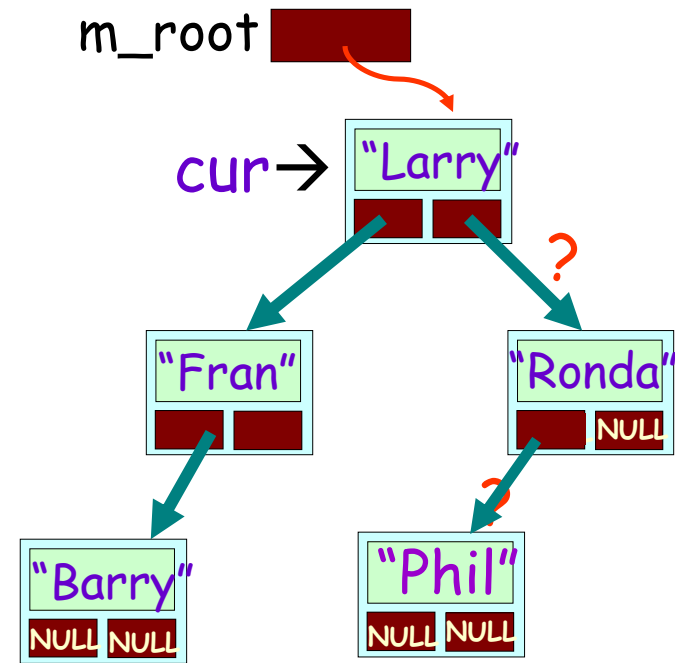
    bst.insert("Phil");
}

```

```

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?

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?

```
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?

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

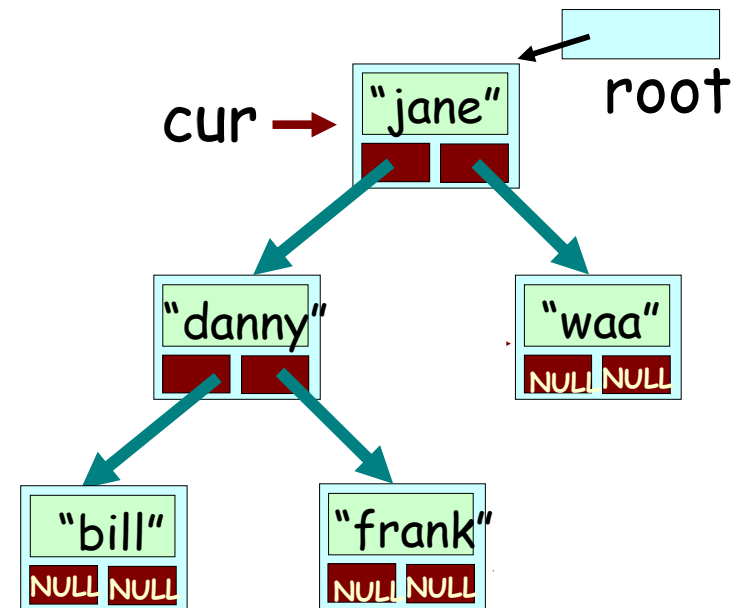
    if (pRoot->right == NULL)
        return(pRoot->value);

    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?



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.

Question: Can anyone think of an algorithm for this?

```
void FreeTree(Node *cur)
{
    if (cur == NULL)          // 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
}
```

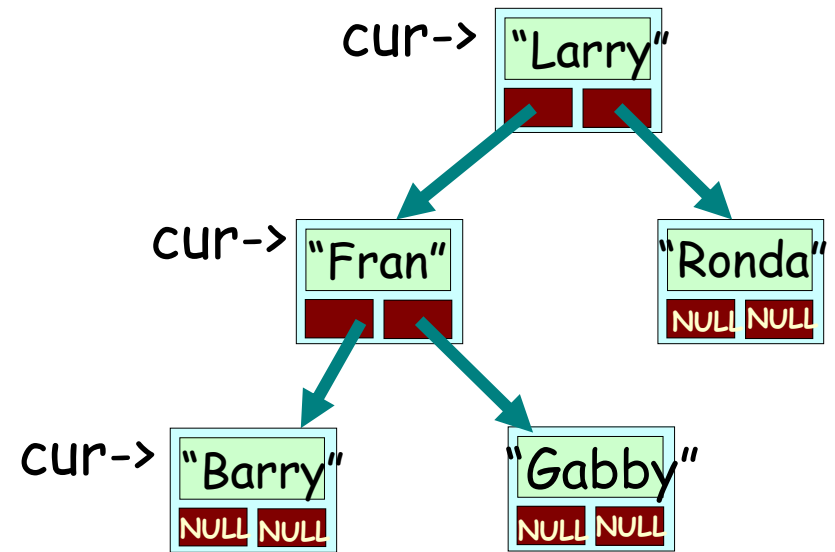
Freeing The Whole Tree

cur = **NULL**

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

    FreeTree(cur->left);
    FreeTree (cur-> right);

    delete cur;
}
```

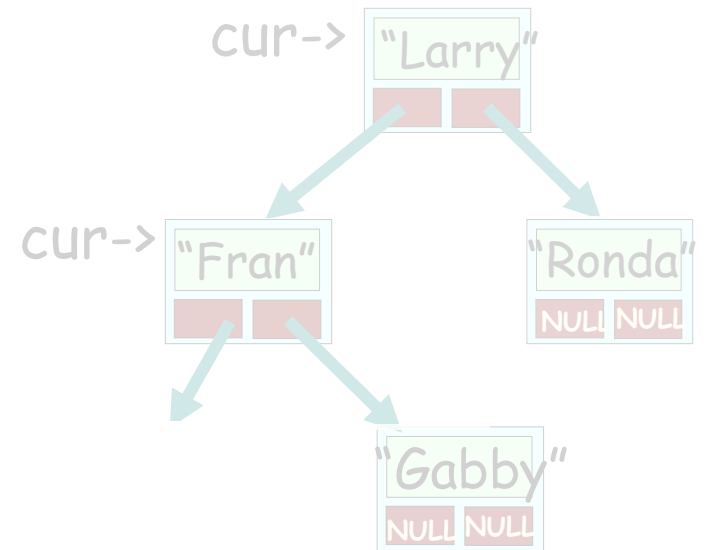


Freeing The Whole Tree

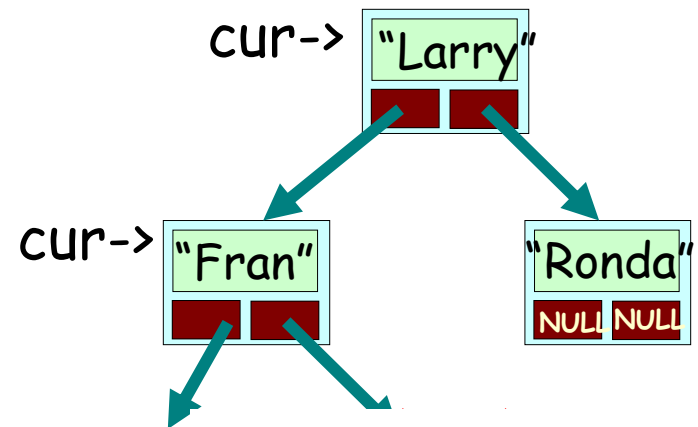
```
void FreeTree(Node *cur)
{
    if (cur == NULL)
        return;

    FreeTree(cur->left);
    FreeTree (cur-> right);

    delete cur;
}
```



Freeing The Whole Tree



```

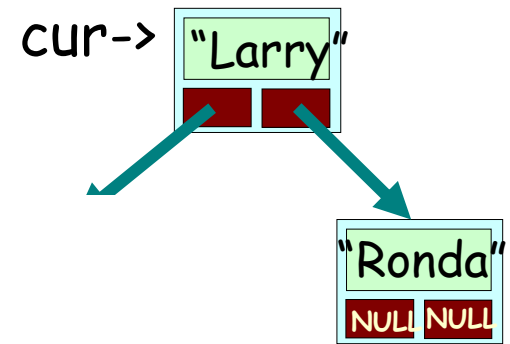
void FreeTree(Node *cur)
{
    if (cur == NULL)
        return;

    FreeTree(cur->left);
    FreeTree (cur-> right);

    delete cur;
}

```

Freeing The Whole Tree



```
void FreeTree(Node *cur)
{
    if (cur == NULL)
        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.