

# Lecture #11

- Sorting Algorithms, part II:
  - Quicksort
  - Mergesort
- Introduction to Trees

# But first... STL Challenge

Give me a data structure that I can use to maintain a bunch of people's names and for each person, allows me to easily get all of the streets they lived on.

Assuming I have  $P$  total people and each person has lived on an average of  $E$  former streets...

What is the Big-Oh cost of:

- A. Finding the names of all people who have lived on "Levering street"?
- B. Determining if "Bill" ever lived on "Westwood blvd"?
- C. Printing out every name along with each person's street addresses, in alphabetical order.
- D. Printing out all of the streets that "Tala" has lived on.

# Divide and Conquer Sorting

The last two sorts we'll learn (for now) are  
Quicksort and Mergesort.

These sorts generally work as follows:

1. **Divide** the elements to be sorted into two groups of roughly equal size.
2. **Sort** each of these smaller groups of elements (conquer).
3. **Combine** the two sorted groups into one large sorted list.

Any time you see "divide and conquer," you should think  
recursion... EEK!

# The Quicksort Algorithm

1. If the array contains only 0 or 1 element, **return**.
2. Select an arbitrary element **P** from the array (typically the **first element** in the array).
3. Move all elements that are **less than or equal** to **P** to the **left of the array** and all elements **greater than P** to the **right** (this is called **partitioning**).
4. Recursively repeat this process on the left sub-array and then the right sub-array.

Divide

Conquer

13	1	21	30	69	40	77
----	---	----	----	----	----	----

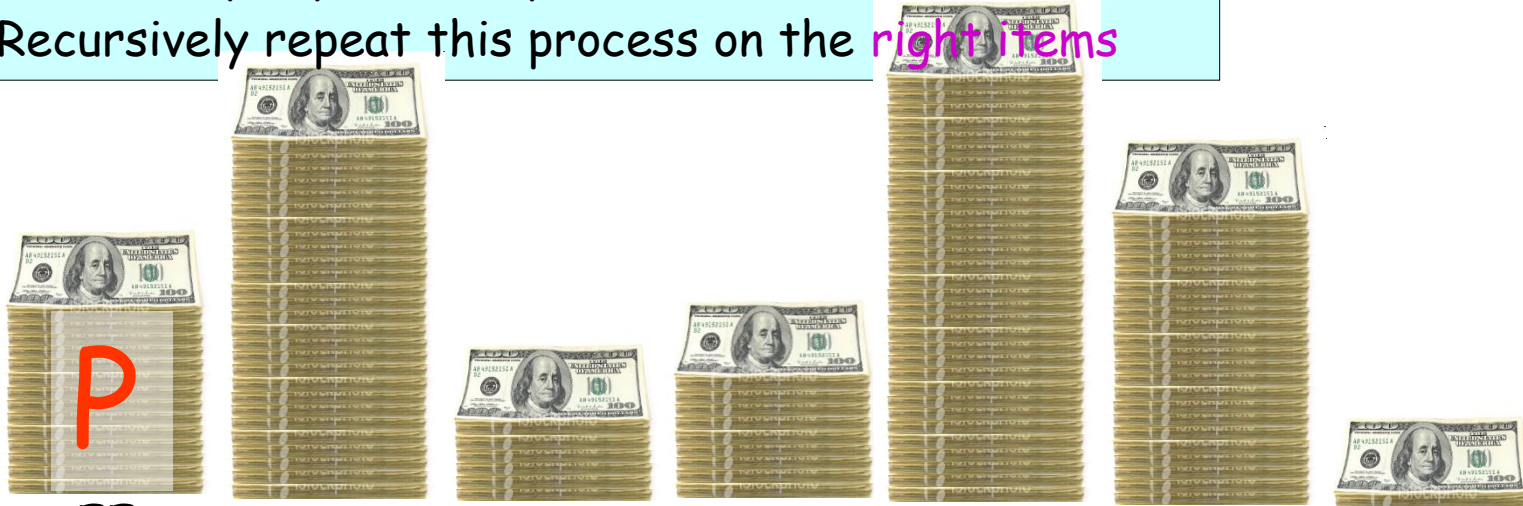
Select an arbitrary item **P** from the array.

Move items **smaller than or equal to P** to the **left** and **larger items** to the **right**; **P** goes in-between.

Recursively repeat this process on the **left items**

Recursively repeat this process on the **right items**

# QuickSort



EE  
Major

MBA

History  
Major

Bio  
Major

Drop-out

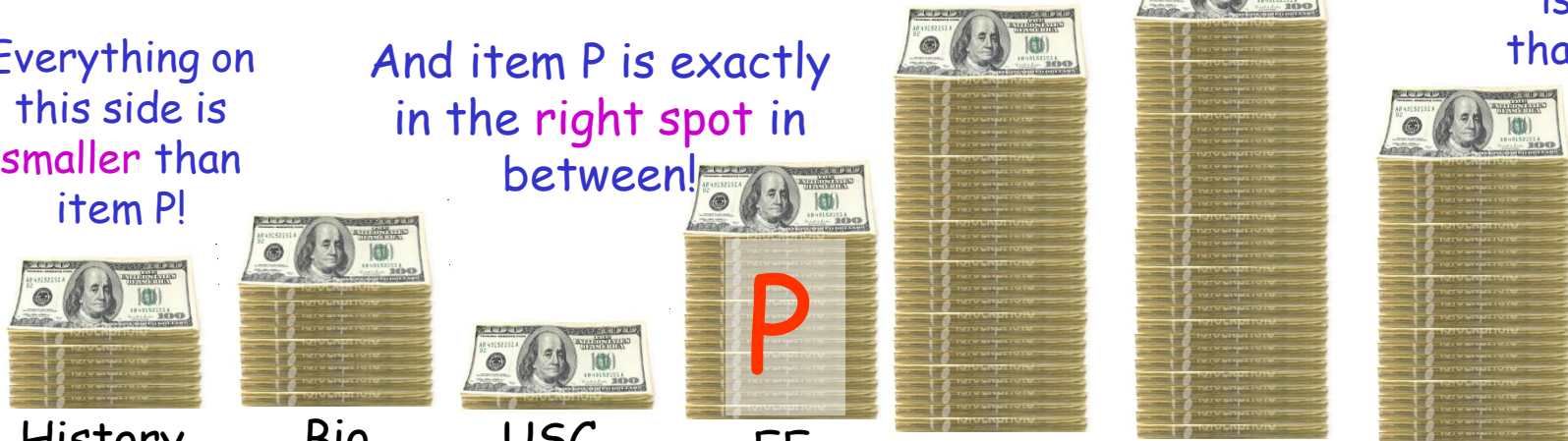
CS

USC  
rad

Everything on  
this side is  
**smaller** than  
item **P**!

And item **P** is exactly  
in the **right spot** in  
between!

Everything  
on this side  
is **larger**  
than item **P**!



History  
Major

Bio  
Major

USC  
Grad

EE  
Major

MBA

Drop-out

CS Major

# QuickSort

Select an arbitrary item **P** from the array.

Move items **smaller than or equal to P** to the **left** and **larger items** to the **right**; **P** goes in-between.

Recursively repeat this process on the **left items**

Recursively repeat this process on the **right items**



History  
Major



Bio  
Major



USC  
Grad



EE  
Major



MBA



Drop-out CS Major



USC  
Grad



History  
Major



Bio  
Major

Everything left of EE Major  
(our first P) is now **sorted**!



# QuickSort

Select an arbitrary item **P** from the array.

Move items **smaller than or equal to P** to the **left** and **larger items** to the **right**; **P** goes in-between.

Recursively repeat this process on the **left items**

Recursively repeat this process on the **right items**



USC  
Grad



History  
Major



Bio  
Major



EE  
Major



MBA



Drop-out CS Major



USC  
Grad



History  
Major



Bio  
Major



EE  
Major



CS Major



MBA



Drop-out

Finally, all items are sorted!

Everything right of  
Major (our first P) is now  
sorted!

# D Quickso

**First** specifies the starting element of the array to sort.

**Last** specifies the last element of the array to sort.

Only bother sorting arrays of **at least two** elements!

And here's an actual Quicksort C++ function:

0

7

```
void QuickSort(int Array[],int First,int Last)
{
    if (Last - First >= 1 )
    {
        int PivotIndex;

        3
        PivotIndex = Partition(Array,First,Last) ;
        QuickSort(Array,First,PivotIndex-1); // left
        QuickSort(Array,PivotIndex+1,Last);  // right
    }
}
```

## DIVIDE

Pick an element.  
Move ≤ items left  
Move > items right

## CONQUER

Apply our QS algorithm to the left half of the array.

## CONQUER

Apply our QS algorithm to the right half of the array.

13	1	21	30	69	40	77	46
0	1	2	3	4	5	6	7



# The QS Partition Function

The **Partition** function uses the first item as the pivot value and moves **less-than-or-equal items** to the **left** and **larger ones** to the **right**.

```
int Partition(int a[], int low, int high)
{
    int pi = low;
    int pivot = a[low];
    do
    {
        while ( low <= high && a[low] <= pivot )
            low++;
        while ( a[high] > pivot )
            high--;
        if ( low < high )
            swap(a[low], a[high]);
    }
    while ( low < high );
    swap(a[pi], a[high]);
    pi = high;
    return(pi);
}
```

And finally, return the pivot's index in the array (**4**) to the QuickSort function.

4	1	12	13	30	52	40	99	77	35	47	56
0	1	2	3	4	5	6	7	8	9	10	11

# Big-oh of Quicksort

We first **partition** the array, at a cost of  $n$  steps.

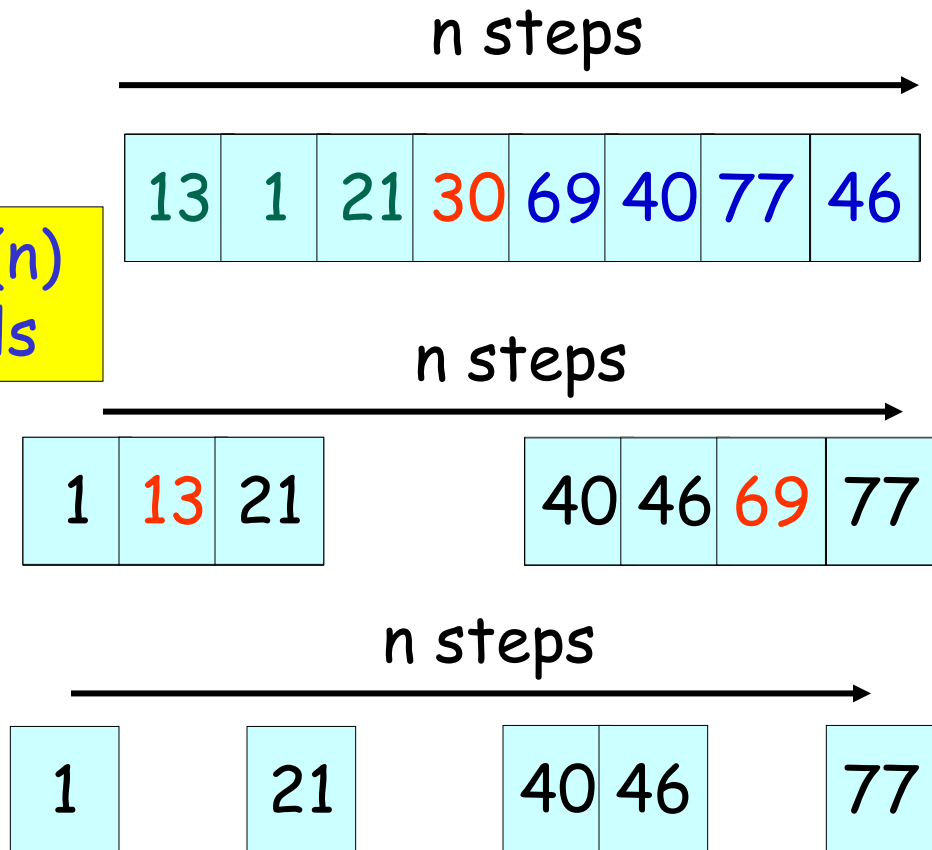
Then we repeat the process for each half...

We **partition** each of the  $2$  halves, each taking  $n/2$  steps, at a total cost of  $n$  steps.

Then we repeat the process for each half...

We **partition** each of the  $4$  halves, each taking  $n/4$  steps, at a total cost of  $n$  steps.

$\log_2(n)$   
levels



So at each level, we do  $n$  operations, and we have  $\log_2(n)$  levels, so we get:  $n \log_2(n)$ .

# Quicksort - Is It Always *Fast*?

Are there any kinds of input data where Quicksort is either **more** or **less efficient**?

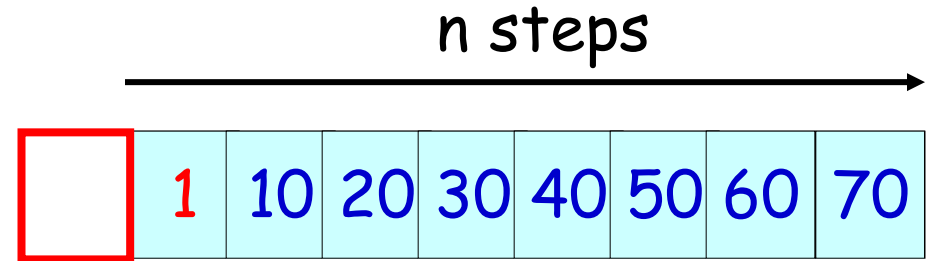
Yes! If our array is **already sorted** or **mostly sorted**, then quicksort becomes **very slow**!

1	10	20	30	40	50	60	70
---	----	----	----	----	----	----	----

Let's see why.

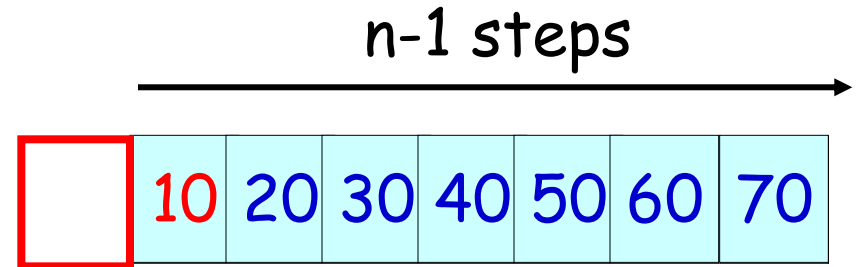
# Worst-case Big-oh of Quicksort

We first **partition** the array, at a cost of  $n$  steps.

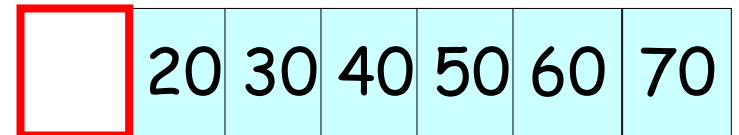


Then we repeat the process for the ~~left~~ & right groups...

Ok, let's **partition** our **right group** then.

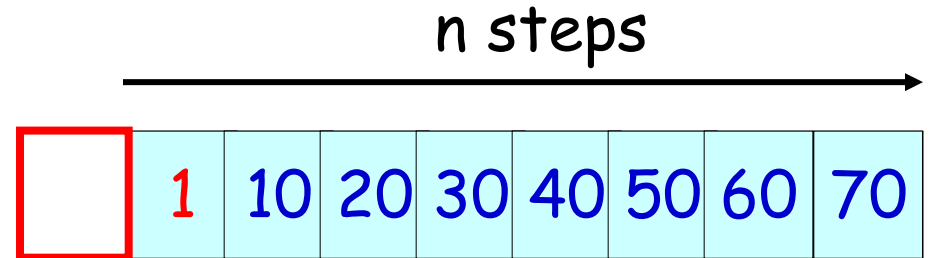


Then we repeat the process for the ~~left~~ & right groups...



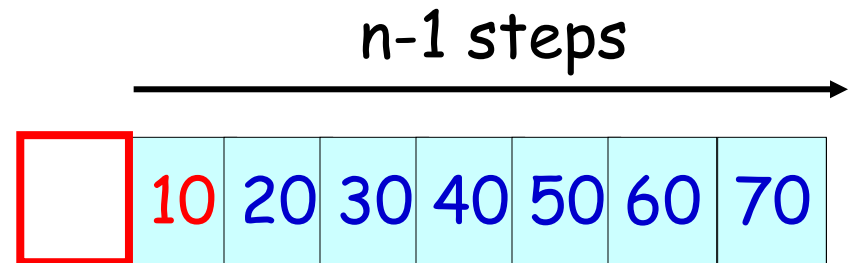
# Worst-case Big-oh of Quicksort

We first **partition** the array, at a cost of  $n$  steps.



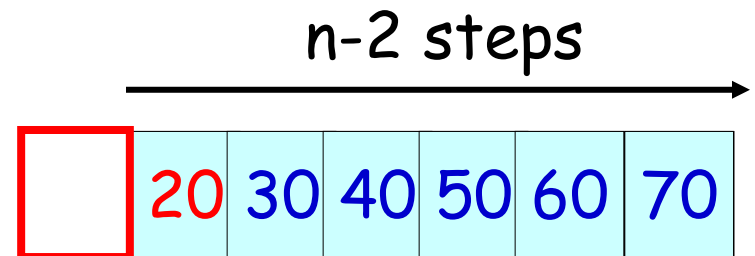
Then we repeat the process for the ~~left~~ & right groups...

Ok, let's **partition** our **right group** then.

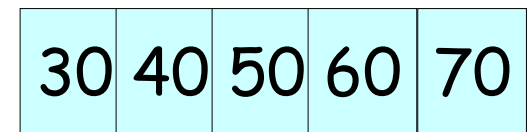


Then we repeat the process for the ~~left~~ & right groups...

Ok, let's **partition** our **right group** then.



Then we repeat the process for the ~~left~~ & right groups...





# Worst-case Big-oh of Quicksort

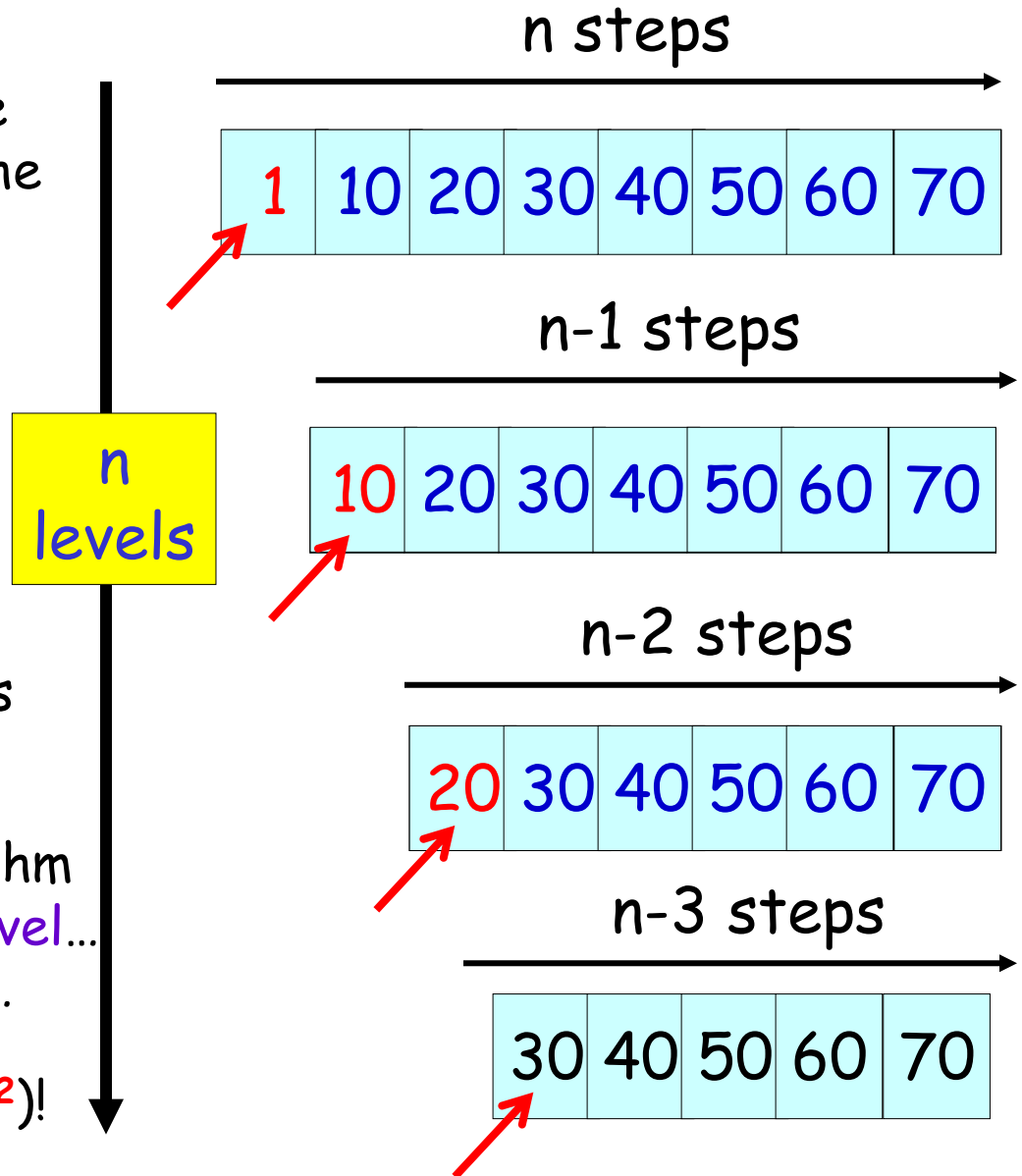
What you'll notice is that each time we partition, we remove **only one item** off the left side!

And if we only remove one item off the left side each time...

We're going to have to go through this **partitioning** process  **$n$  times** to process the entire array!

And if the partition algorithm requires  **$\sim n$  steps** at **each level**...  
And we go  **$n$  levels deep**...

Then our algorithm is  $O(n^2)$ !



# Other Quicksort Worst Cases?

So, as you can see, an array that's **mostly in order** will require an average of  **$N^2$  steps!**

As you can probably guess, **Quicksort** also has the same problem with arrays that are in **reverse order!**

So if you happen to know your data will be **mostly sorted** (or in **reverse**) **order**, avoid Quicksort!

It's a **DOG!**



# QuickSort Questions

Can QuickSort be applied easily to sort items within a linked list?

Is QuickSort a "stable" sort?

Does QuickSort use a fixed amount of RAM, or can it vary?

Can QuickSort be parallelized across multiple cores?

When might you use QuickSort?

# Mergesort

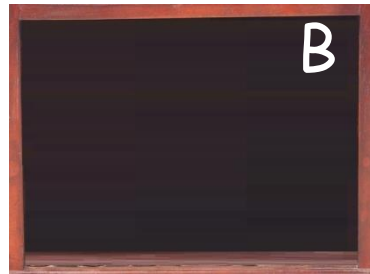
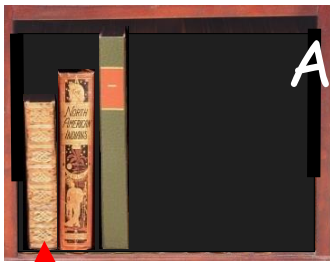
The Mergesort is another extremely efficient sort - yet it's pretty easy to understand.



But before we learn the **Mergesort**, we need to learn another algorithm called "**merge**".

# Mergesort

The basic **merge** algorithm takes **two-presorted arrays** as inputs and **outputs a combined, third sorted array**.



## Merge Algorithm

Consider the left-most book in both shelves  
 Take the smallest of the two books  
 Add it to the new shelf  
 Repeat the whole process until all books are moved

1. Initialize counter variables  $i1$ ,  $i2$  to zero
2. While there are more items to copy...
  - If  $A1[i1]$  is less than  $A2[i2]$ 
    - Copy  $A1[i1]$  to output array B and  $i1++$
  - Else
    - Copy  $A2[i2]$  to output array B and  $i2++$
3. If either array runs out, copy the entire contents of the other array over

By always selecting and moving the **smallest book** from either shelf we guarantee all of our books will end up sorted!



# Merge Algorithm in C++

```

void merge(int data[], int n1, int n2)
{
    int i=0, j=0, k=0;
    int *temp = new int[n1+n2];
    int *sechalf = data + n1;

    while (i < n1 || j < n2)
    {
        if (i == n1)
            temp[k++] = sechalf[j++];
        else if (j == n2)
            temp[k++] = data[i++];
        else if (data[i] < sechalf[j])
            temp[k++] = data[i++];
        else
            temp[k++] = sechalf[j++];
    }
    for (i=0; i<n1+n2; i++)
        data[i] = temp[i];
    delete [] temp;
}

```

Here's the C++ version of our **merge** function!

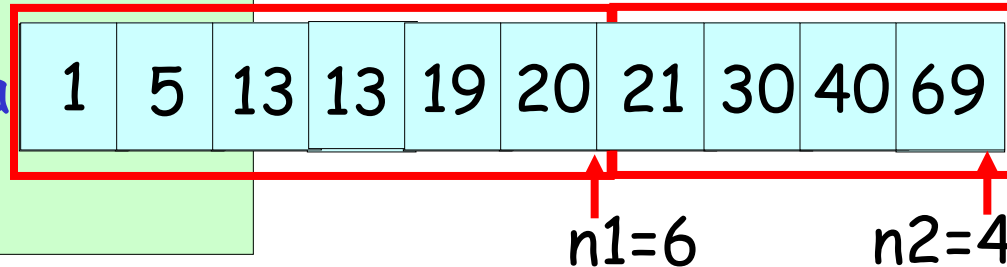
Instead of passing in **A1**, **A2** and **B...**

you pass in an array called **data** and two sizes: **n1** and **n2**

Notice how this function uses **new/delete** to allocate a temporary array for merging.

**data** holds the merged contents at the end.

**data**



# Mergesort

OK - so what's the full mergesort algorithm:

Mergesort function :

1. If array has one element, then return (it's sorted).
2. Split up the array into two equal sections
3. Recursively call Mergesort function on the left half
4. Recursively call Mergesort function on the right half
5. Merge the two halves using our **merge** function

Ok, let's see how to mergesort a shelf full of books!




1. If array has 1 item, then return
2. Split array in two equal sections
3. Call Mergesort on the left half
4. Call Mergesort on the right half
5. **Merge** the halves back together



1. If array has 1 item, then return
2. Split array in two equal sections
3. Call Mergesort on the left half
4. Call Mergesort on the right half
5. **Merge** the halves back together





1. If array has 1 item, then return
2. Split array in two equal sections
3. Call Mergesort on the left half
4. Call Mergesort on the right half
5. **Merge** the halves back together

- 
1. If array has 1 item, then return
  2. Split array in two equal sections
  3. Call Mergesort on the left half
  4. Call Mergesort on the right half
  5. **Merge** the halves back together



1. If array has 1 item, then return
2. Split array in two equal sections
3. Call Mergesort on the left half
4. Call Mergesort on the right half
5. **Merge** the halves back together

- 
- A single book is shown on the left, with a vertical red line passing through its center, indicating it is being split into two halves.
1. If array has 1 item, then return
  2. Split array in two equal sections
  3. Call Mergesort on the left half
  4. Call Mergesort on the right half
  5. **Merge** the halves back together

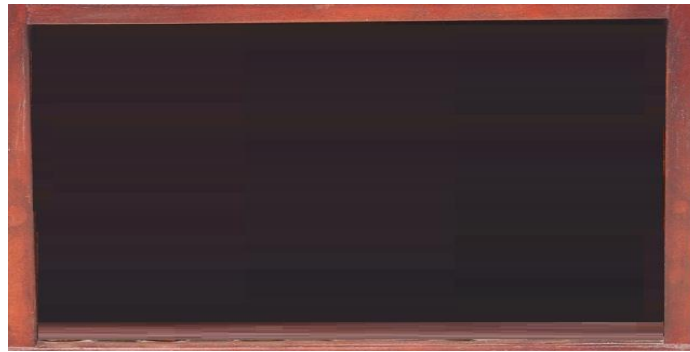
- 
- Two separate sorted sub-arrays are shown: a small grey book on the left and a taller orange book on the right.
1. If array has 1 item, then return
  2. Split array in two equal sections
  3. Call Mergesort on the left half
  4. Call Mergesort on the right half
  5. **Merge** the halves back together



i1



i2



1. If array has 1 item, then return
2. Split array in two equal sections
3. Call Mergesort on the left half
4. Call Mergesort on the right half
5. **Merge** the halves back together



↑  
i1



↑  
i2

And our array is  
sorted!!!!



# Mergesort - One Final Detail

While I showed the Mergesort moving books into a bunch of small piles...



i1 i2



The real algorithm sorts the data in-place in the array...

and only uses a separate array for merging.

Let's see how it really works!

# Big-oh of Mergesort



$\log_2 n$  levels deep

Why? Because we keep dividing our piles in half...

until our piles are just 1 book!



# Big-oh of Mergesort

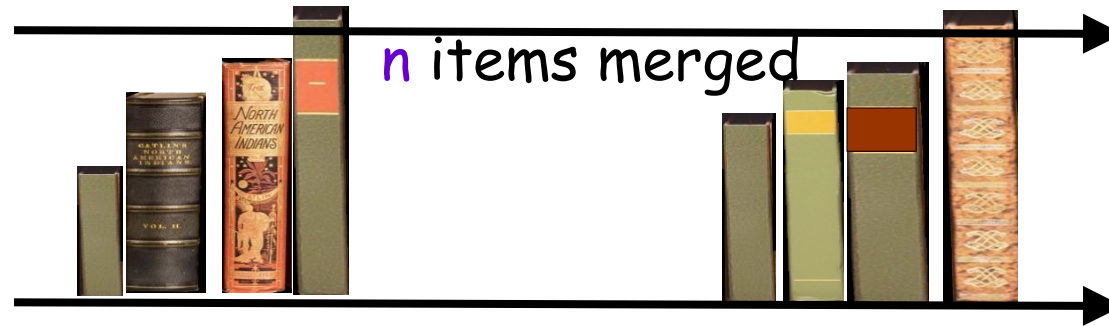


$\log_2 n$  levels deep

Why? Because we keep dividing our piles in half...

until our piles are just 1 book!

# Big-oh of Mergesort



$n$  items merged



$\log_2 n$  levels deep

Why? Because we keep dividing our piles in half...

until our piles are just 1 book!

Overall, this gives us  $n \cdot \log_2(n)$  steps to sort  $n$  items of data. Not bad! 😊

# Mergesort - Any Problem Cases

So, are there any cases where mergesort is less efficient?

No! Mergesort works equally well regardless of the ordering of the data...



However, because the merge function needs secondary arrays to merge, this can slow things down a bit...

In contrast, quicksort doesn't need to allocate any new arrays to work.



# MergeSort Questions

Can MergeSort be applied easily to sort items within a linked list?

Is MergeSort a "stable" sort?

Are there any special uses for MergeSort that other sorts can't handle?

Can MergeSort be parallelized across multiple cores?

# Sorting Overview

Sort Name	Stable/ Non-stable	Notes
Selection Sort	Unstable	Always $O(n^2)$ , but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow)
Insertion Sort	Stable	$O(n)$ for already or nearly-ordered arrays. $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement.
Bubble Sort	Stable	$O(n)$ for already or nearly-ordered arrays (with a good implementation). $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement. Rarely a good answer on an interview!
Shell Sort	Unstable	$O(n^{1.25})$ approx. OK for linked lists. Used in some embedded systems (eg, in a car) instead of quicksort due to fixed RAM usage.
Quick Sort	Unstable	$O(n \log_2 n)$ average, $O(n^2)$ for already/mostly/reverse ordered arrays or arrays with the same value repeated many times. Can be used with linked lists. Can be parallelized across multiple cores. Can require up to $O(n)$ slots of extra RAM (for recursion) in the worst case, $O(\log_2 n)$ avg.
Merge Sort	Stable	$O(n \log_2 n)$ always. Used for sorting large amounts of data on disk (aka "external sorting"). Can be used to sort linked lists. Can be parallelized across multiple cores. Downside: Requires $n$ slots of extra memory/disk for merging - other sorts don't need extra RAM.
Heap Sort	Unstable	$O(n \log_2 n)$ always. Sometimes used in low-RAM embedded systems because of its performance/low memory req'ts.

# Challenge Problems

1. Give an algorithm to efficiently determine which element occurs the largest number of times in the array.
2. What's the best algorithm to sort 1,000,000 random numbers that are all between 1 and 5?

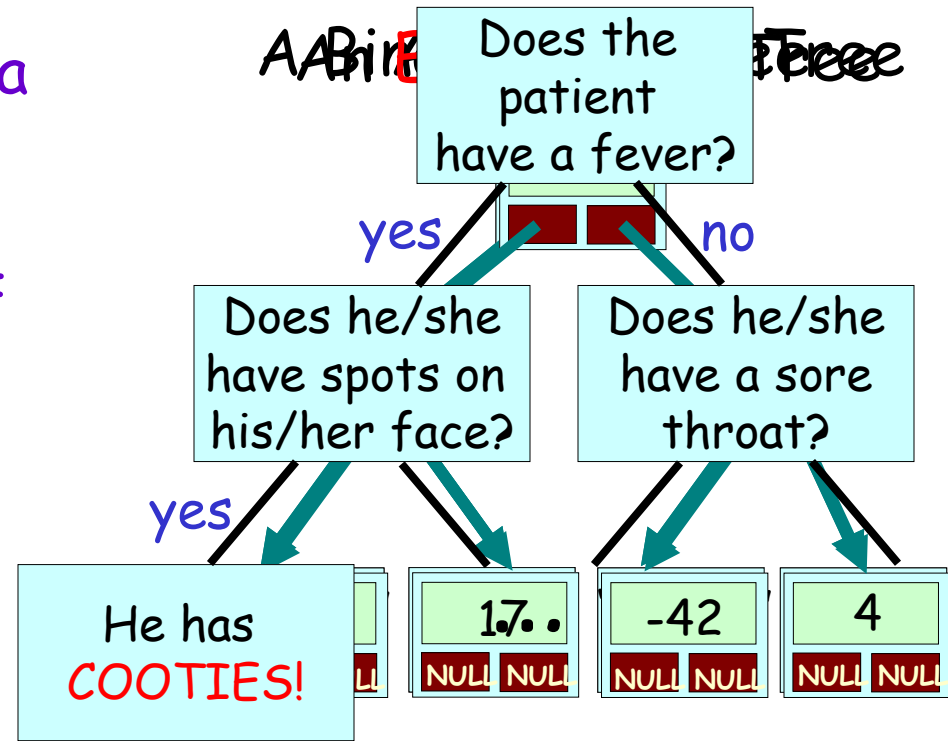
# Trees

"I think that I shall never see a data structure as lovely as a tree." - Carey Nachenberg

A **Tree** is a special **linked list-based data structure** that has many uses in Computer Science:

- To organize hierarchical data
- To make information easily searchable
- To simplify the evaluation of mathematical expressions
- To make decisions

## A **Decision** Tree



# Basic Tree Facts

1. Trees are made of **nodes** (just like linked list nodes).
2. Every tree has a "root" pointer.
3. The top node of a tree is called its "root" node.
4. Every node may have zero or more "children" nodes.
5. A node with 0 children is called a "leaf" node.
6. A tree with no nodes is called an "empty tree."

```

struct node
{
    int value;    // some value
    node *left, *right;
};

node *rootPtr;
  
```

0 children

Leaf node

2 children

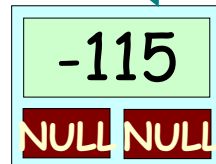
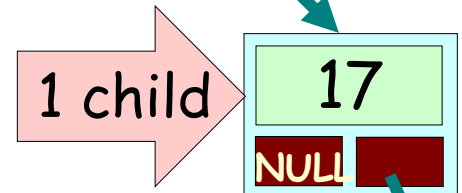
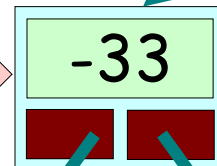
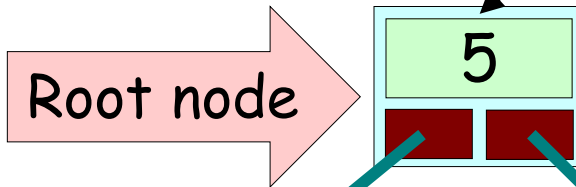
Root node

Empty tree

root ptr

NULL

root ptr



But instead of just one next pointer, a tree node can have **two or more next pointers!**

The tree's **root pointer** is like a linked list's **head pointer!**

# Tree Nodes Can Have Many Children

A tree node can have more than just two children:

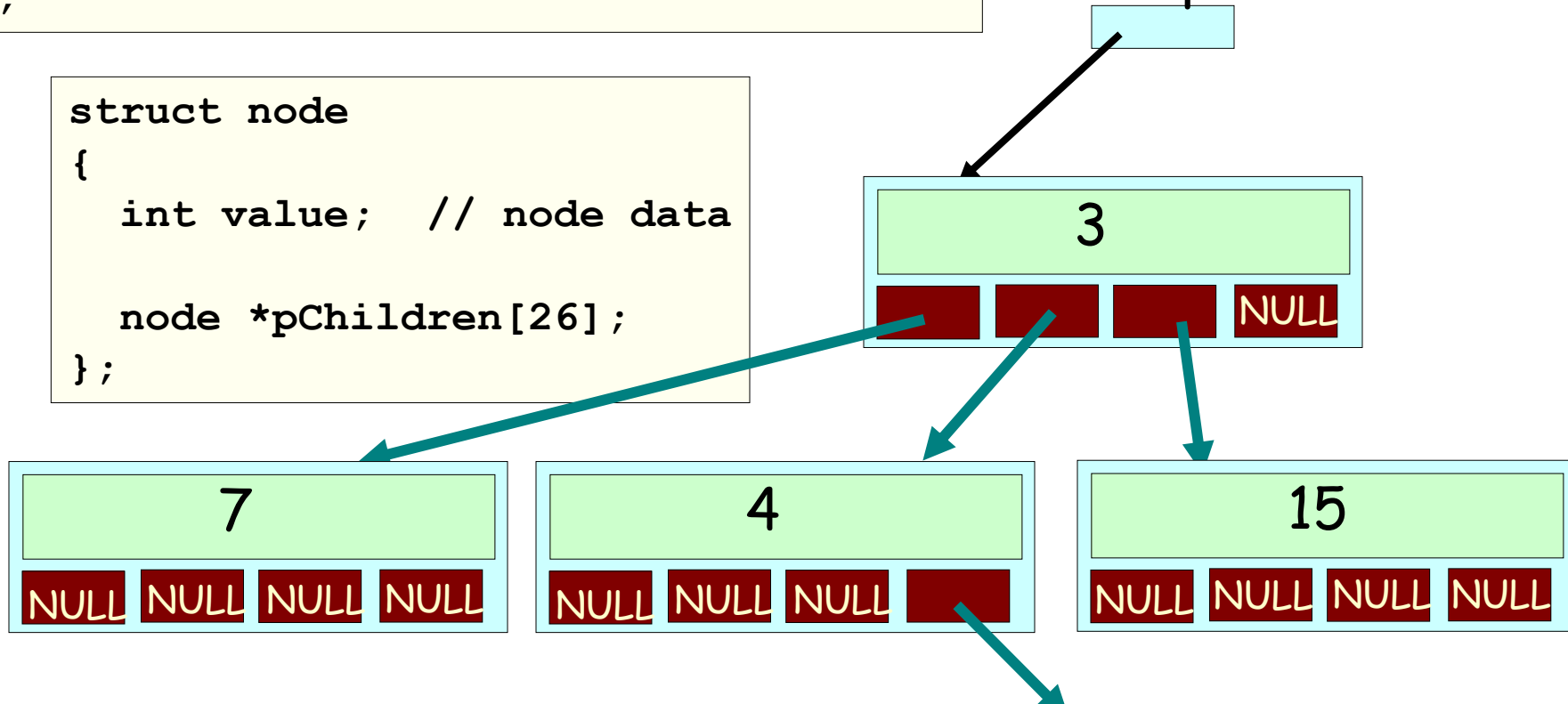
```
struct node
{
    int value;    // node data

    node *pChild1, *pChild2, *pChild3, ...;
};
```

```
struct node
{
    int value;    // node data

    node *pChildren[26];
};
```

root ptr



# Binary Trees

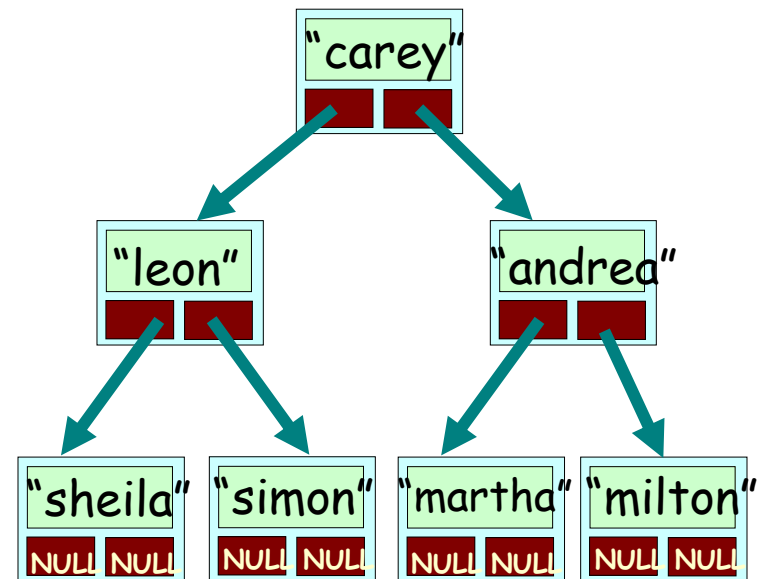
A **binary tree** is a special form of tree. In a binary tree, every node has at most **two children nodes**:

**A left child and a right child.**

```
struct BTNODE // binary tree node
{
    string value; // node data

    BTNODE *pLeft, *pRight;
};
```

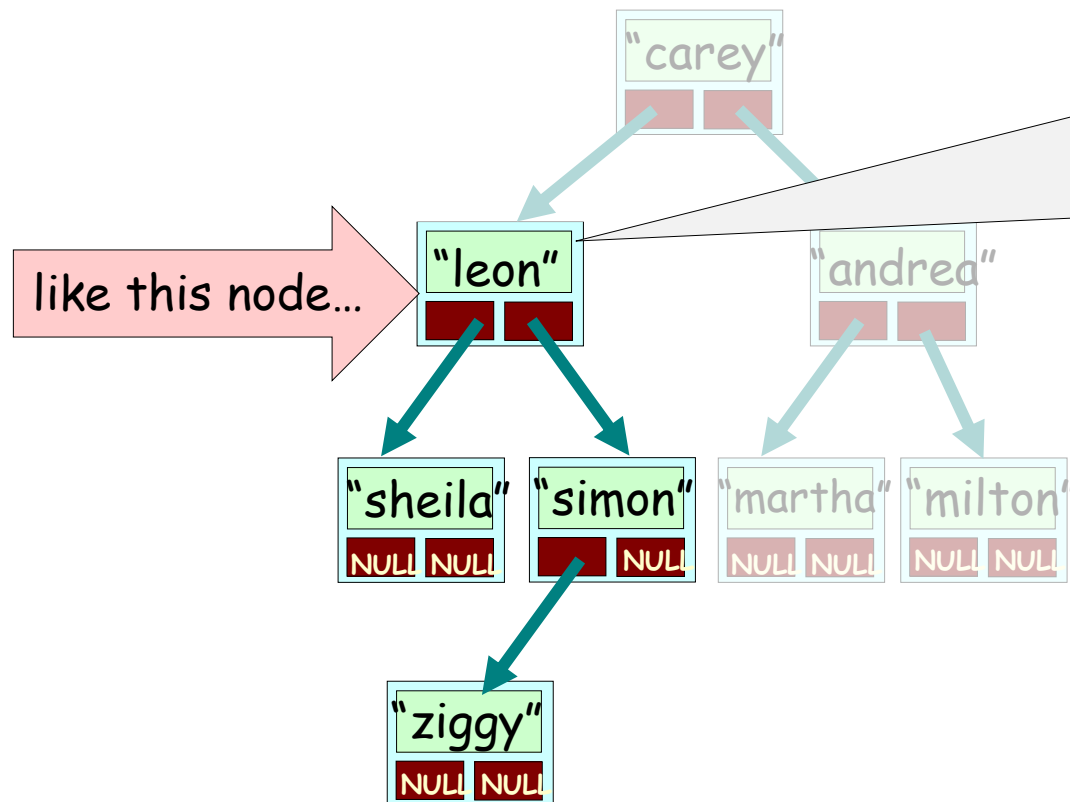
A **Binary** Tree



# Binary Tree Subtrees

We can pick any node in the tree...

And then focus on its "**subtree**" - which includes it and all of nodes below it.



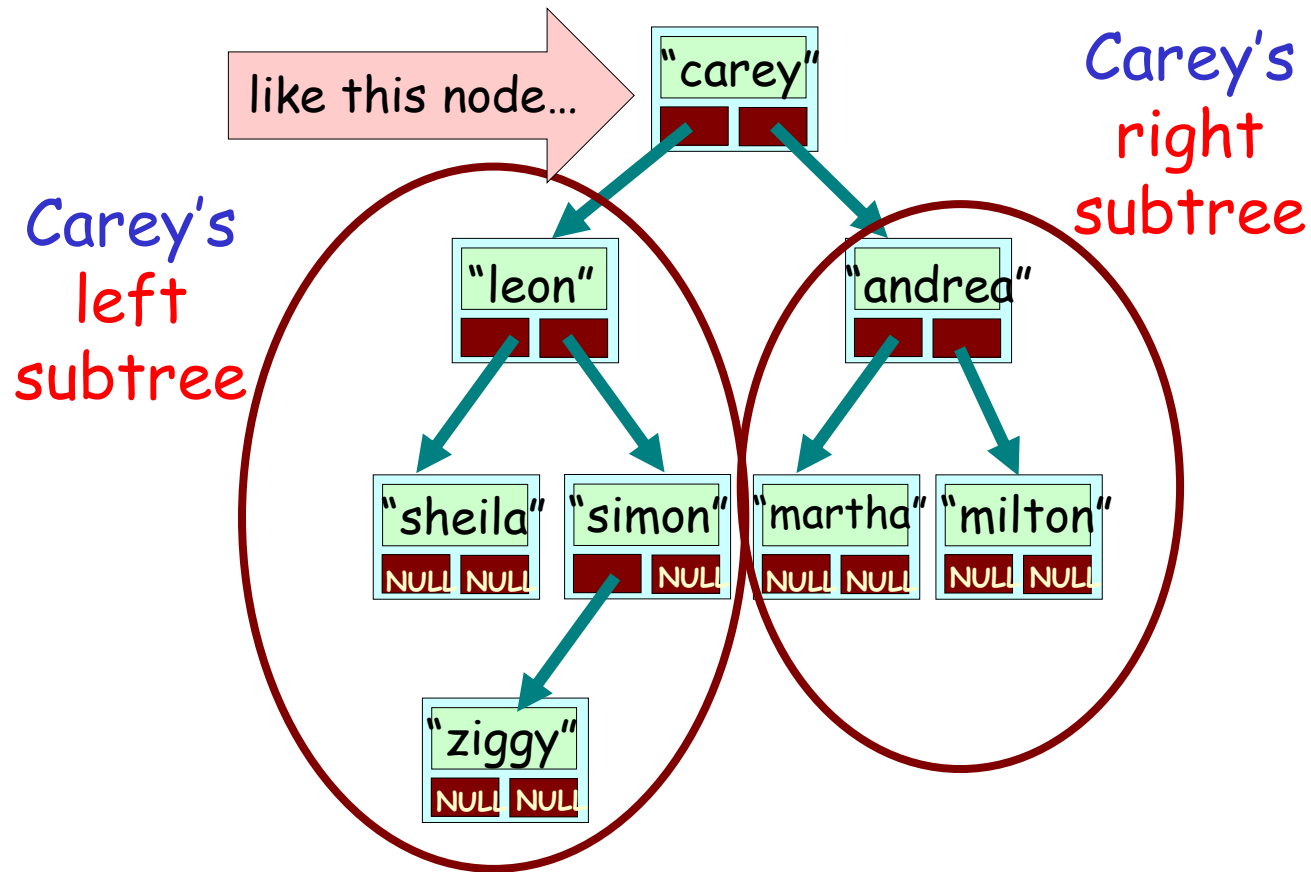
This subtree includes four different nodes...

It has the "leon" node as its root.



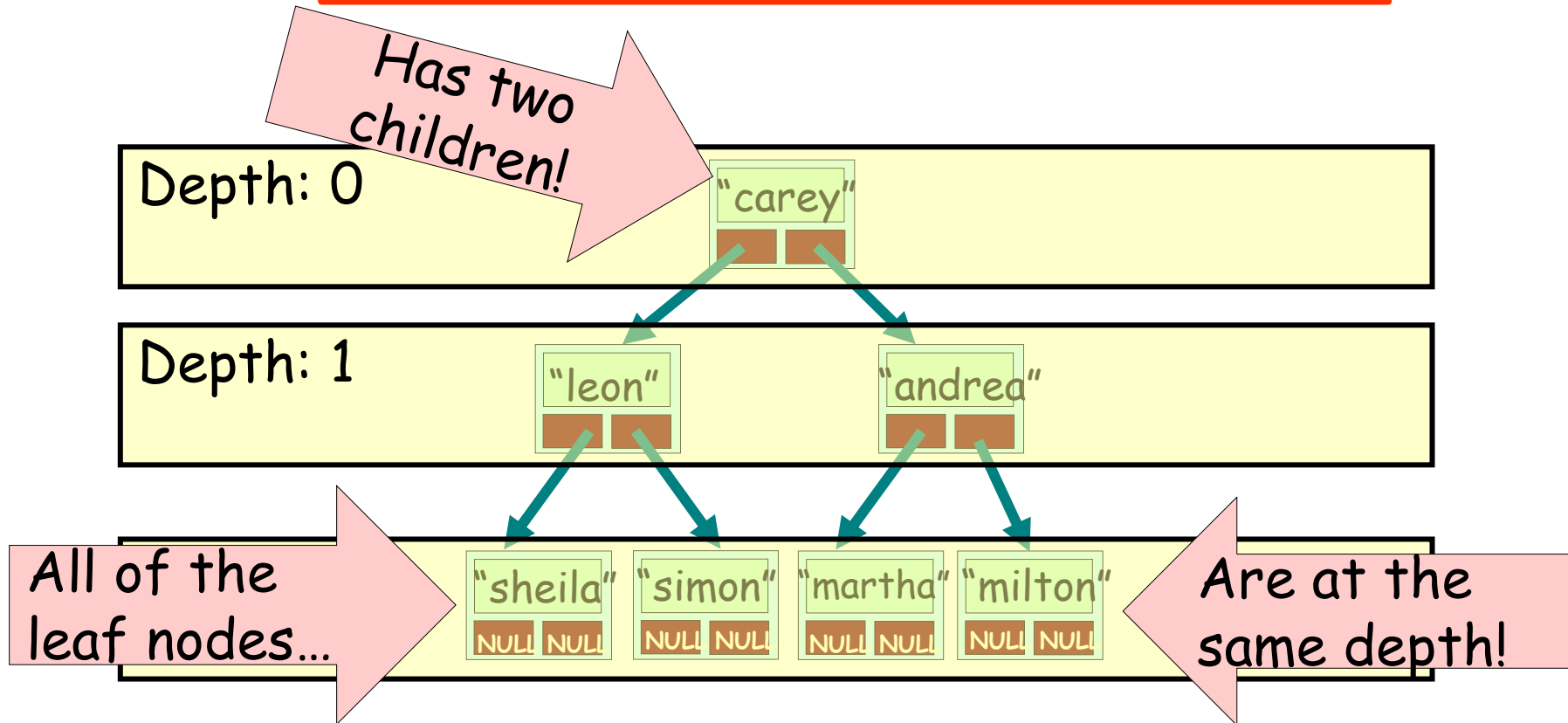
# Binary Tree Subtrees

If we pick a node from our tree...  
we can also identify its **left** and **right sub-trees**.



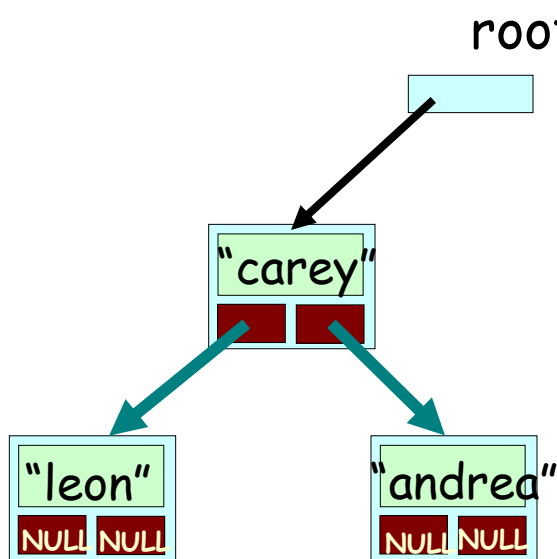
# Full Binary Trees

A **full binary tree** is one in which **every leaf node has the same depth,** and **every non-leaf has exactly two children.**

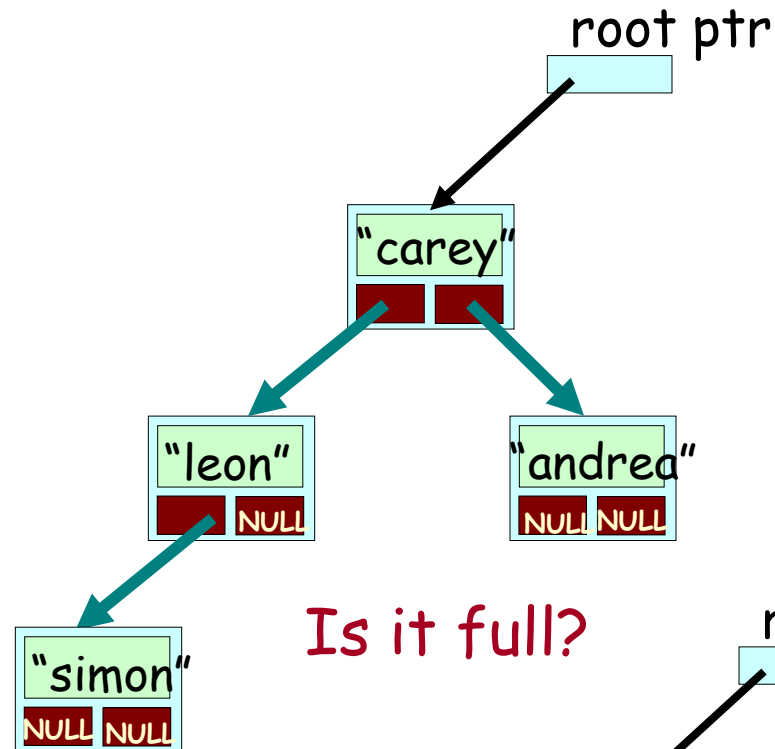


# Full Binary Trees

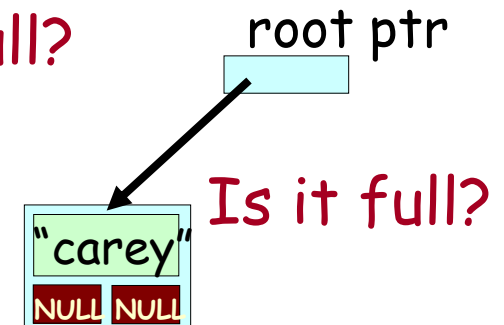
A **full binary tree** is one in which every leaf node has the same depth, and every non-leaf has exactly two children.



Is it full?



Is it full?



Is it full?

# Operations on Binary Trees

The following are common operations that we might perform on a Binary Tree:

- enumerating all the items
- searching for an item
- adding a new item at a certain position on the tree
- deleting an item
- deleting the entire tree (destruction)
- removing a whole section of a tree (called **pruning**)
- adding a whole section to a tree (called **grafting**)

We'll learn about many of these operations over the next two classes.

```

struct BTNODE // node
{
    int value; // data
    BTNODE *left, *right;
};

```

```

main()
{

```

```

    BTNODE *temp, *pRoot

```

```

    pRoot = new BTNODE;
    pRoot->value = 5;

```

```

    temp = new BTNODE;
    temp->value = 7;
    temp->left = NULL;
    temp->right = NULL;
    pRoot->left = temp;

```

```

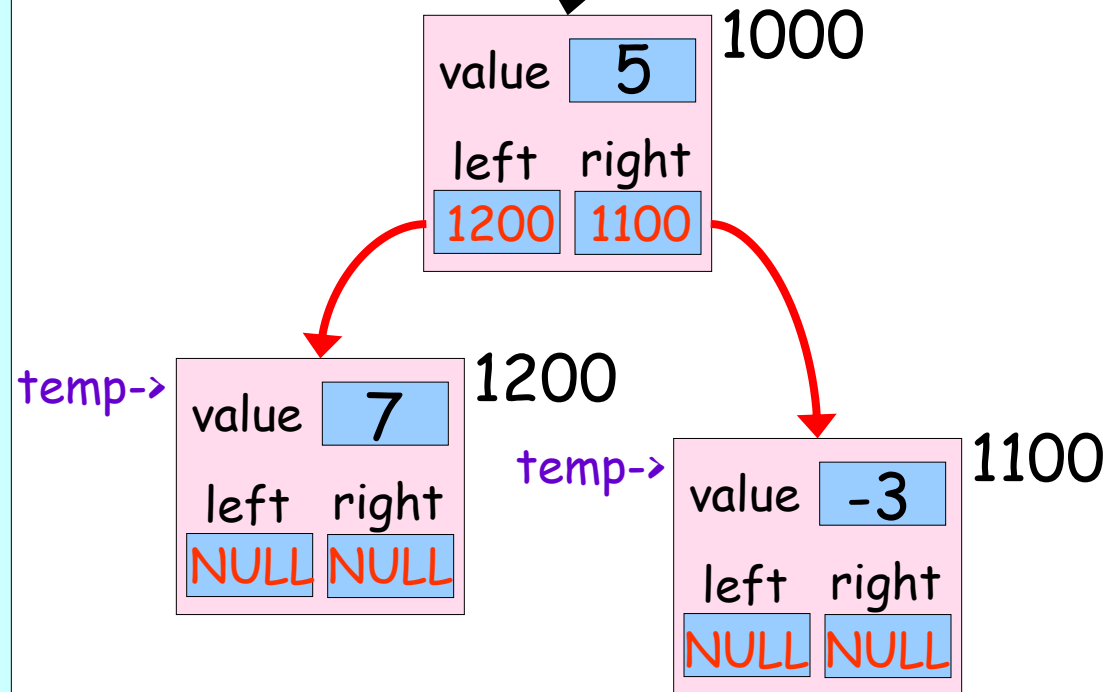
    temp = new BTNODE;
    temp->value = -3;
    temp->left = NULL;
    temp->right = NULL;
    pRoot->right = temp;
    // etc...

```

As with **linked lists**, we use **dynamic memory** to allocate our **nodes**.

# A Simple Tree Example

temp ████████  
pRoot 1000



And of course, later we'd have to delete our tree's nodes.

# We've created a binary tree... now what?

Now that we've created a binary tree, what can we do with it?

Well, next class we'll learn how to use the binary tree to speed up searching for data.

But for now, let's learn how to iterate through each item in a tree, one at a time.

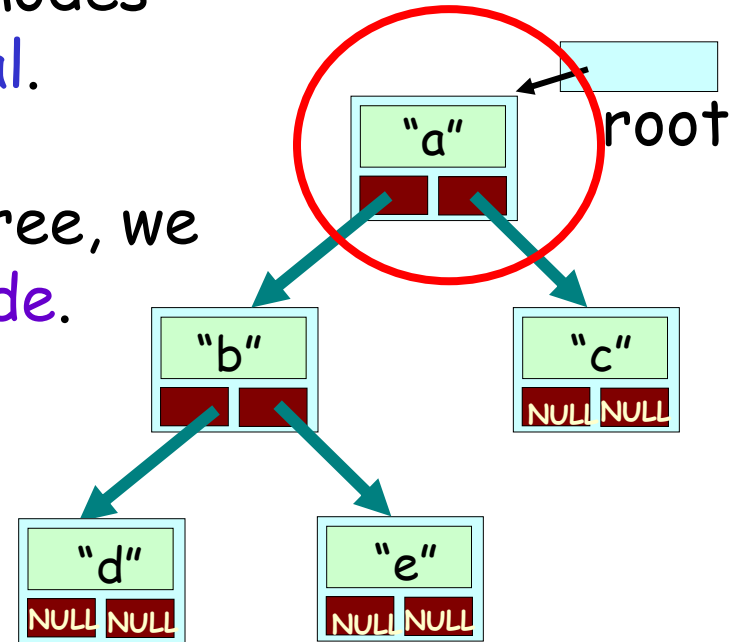
This is called "traversing" the tree, and there are several ways to do it.

# Binary Tree Traversals

When we iterate through all the nodes in a tree, it's called a **traversal**.

Any time we traverse through a tree, we always start with the **root node**.

There are four common ways to traverse a tree.



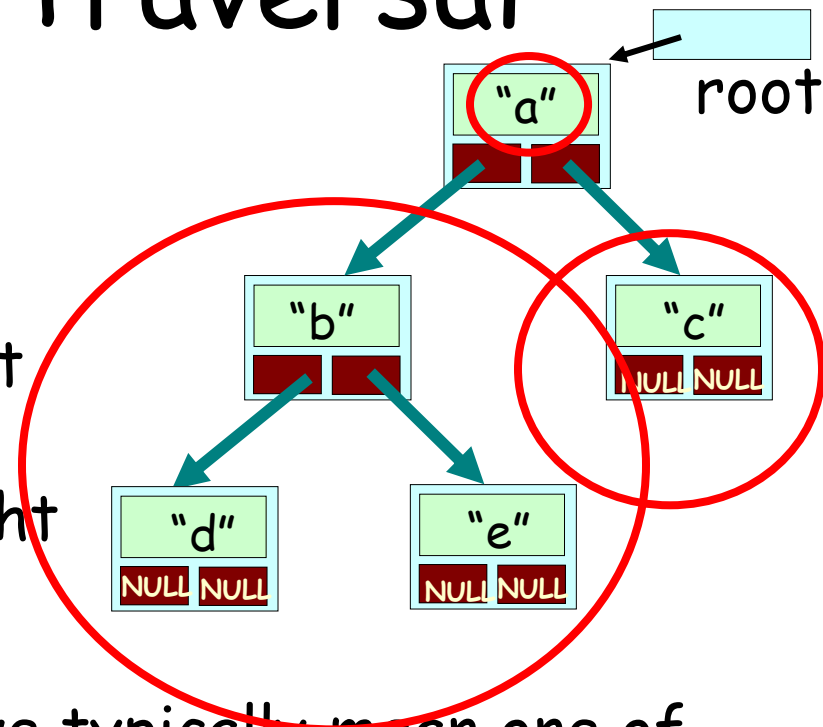
Each technique differs in the **order** that each node is visited during the traversal:

1. Pre-order traversal
2. In-order traversal
3. Post-order traversal
4. Level-order traversal

# The Preorder Traversal

## Preorder:

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



By “**process the current node**” we typically mean one of the following:

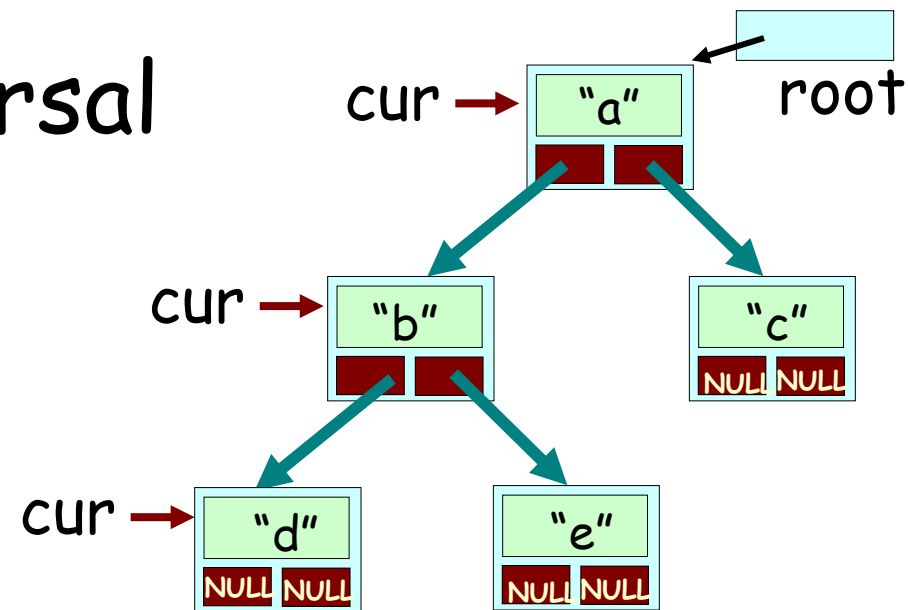
1. Print the current node's value out.
2. Search the current node to see if its value matches the one you're searching for.
3. Add the current node's value to a total for the tree
4. Etc...



# The Pre-order Traversal

Output:

a b d



```

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

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

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

```

main()
{
    Node *root;

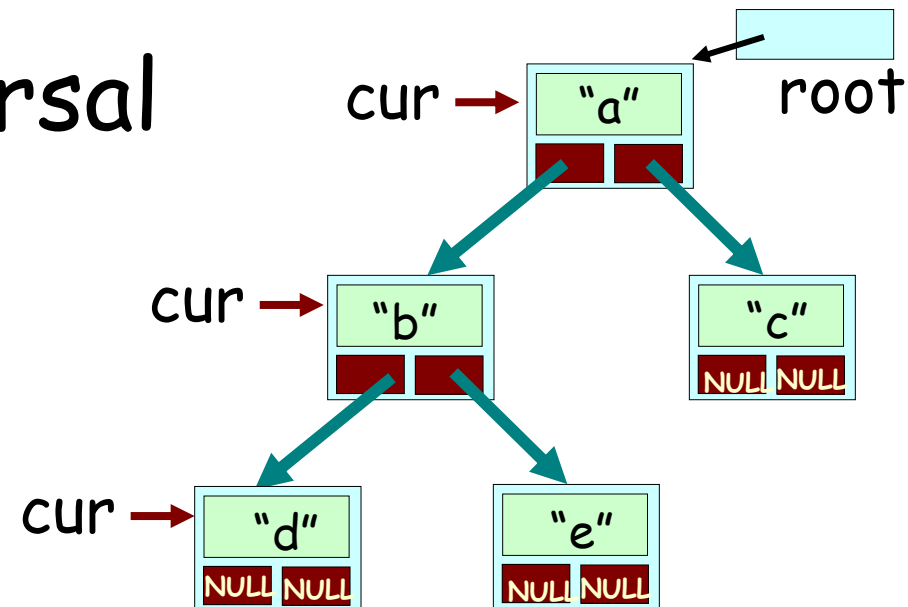
    ...

    PreOrder(root);
}
    
```

# The Pre-order Traversal

Output:

a b d



```
void PreOrder(Node *cur)
```

```
{
```

```
    if (cur == NULL)        // if empty, return...
        return;
```

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

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

```
    PreOrder(cur->right);    // Process nodes in left sub-tree.
```

```
}
```

```
}
```

```
}
```

```
}
```

```
n()
```

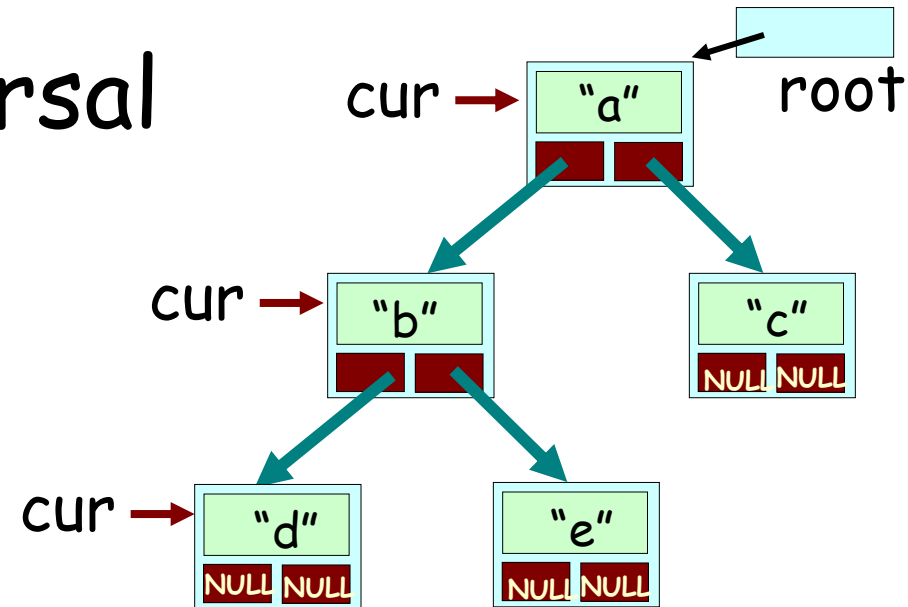
```
Node *root;
```

```
PreOrder(root);
```

# The Pre-order Traversal

Output:

a b d



```
void PreOrder(Node *cur)
```

```
{
{
{   if (cur == NULL)           // if empty, return...
    return;

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

```
main()
```

```
Node *root;
```

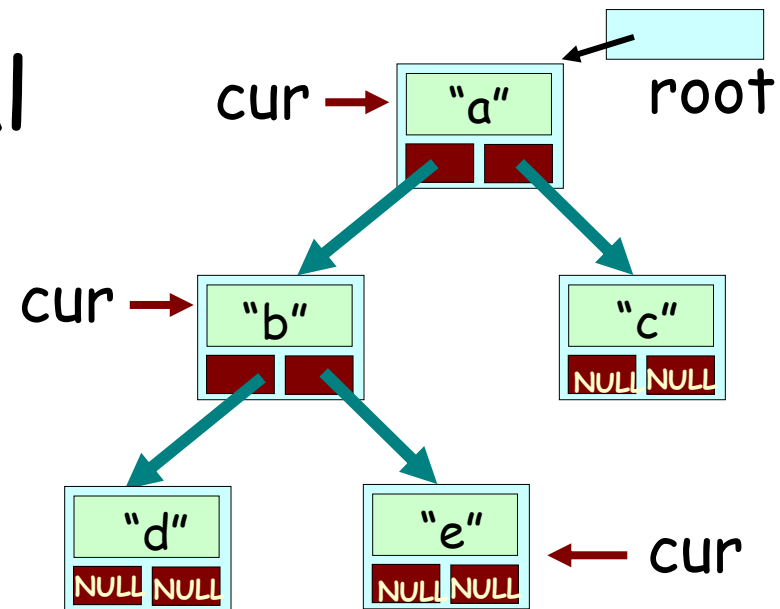
```
...
```

```
PreOrder(root);
```

# The Pre-order Traversal

Output:

a b d e



```
void PreOrder(Node *cur)
```

```
{
{
{   if (cur == NULL)           // if empty, return...
    return;

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

```
main()
```

```
Node *root;
```

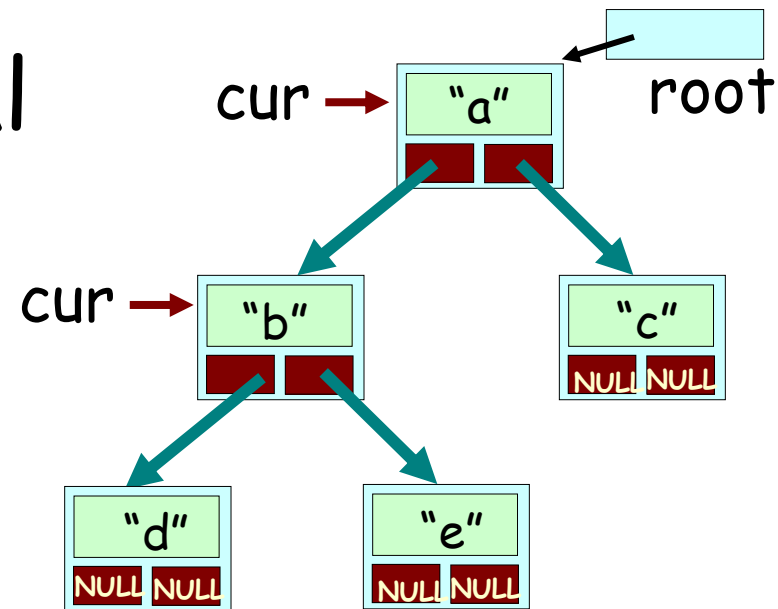
```
...
```

```
PreOrder(root);
```

# The Pre-order Traversal

Output:

a b d e



```
void PreOrder(Node *cur)
```

```
{
  if (cur == NULL)          // if empty, return...
    return;
```

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

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

```
  PreOrder(cur->right);     // Process nodes in left sub-tree.
```

```
}
```

```
}
```

```
main()
```

```
{
```

```
  Node *root;
```

```
  ...
```

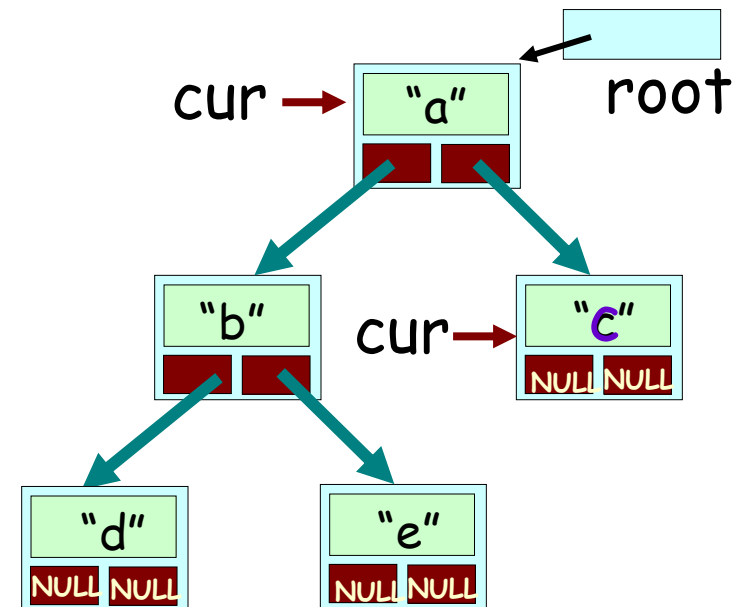
```
  PreOrder(root);
```

```
}
```

# The Pre-order Traversal

Output:

a b d e c



```

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

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

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

```

main()

Node *root;

...

PreOrder(root);
  
```