Lecture #11

- Sorting Algorithms, part II:
  - Quicksort
  - Mergesort
- Trees
  - Introduction
  - Implementation & Basic Properties
  - Traversals: The Pre-order Traversal
- On-your-own Study
  - Full binary trees

But first… STL Challenge

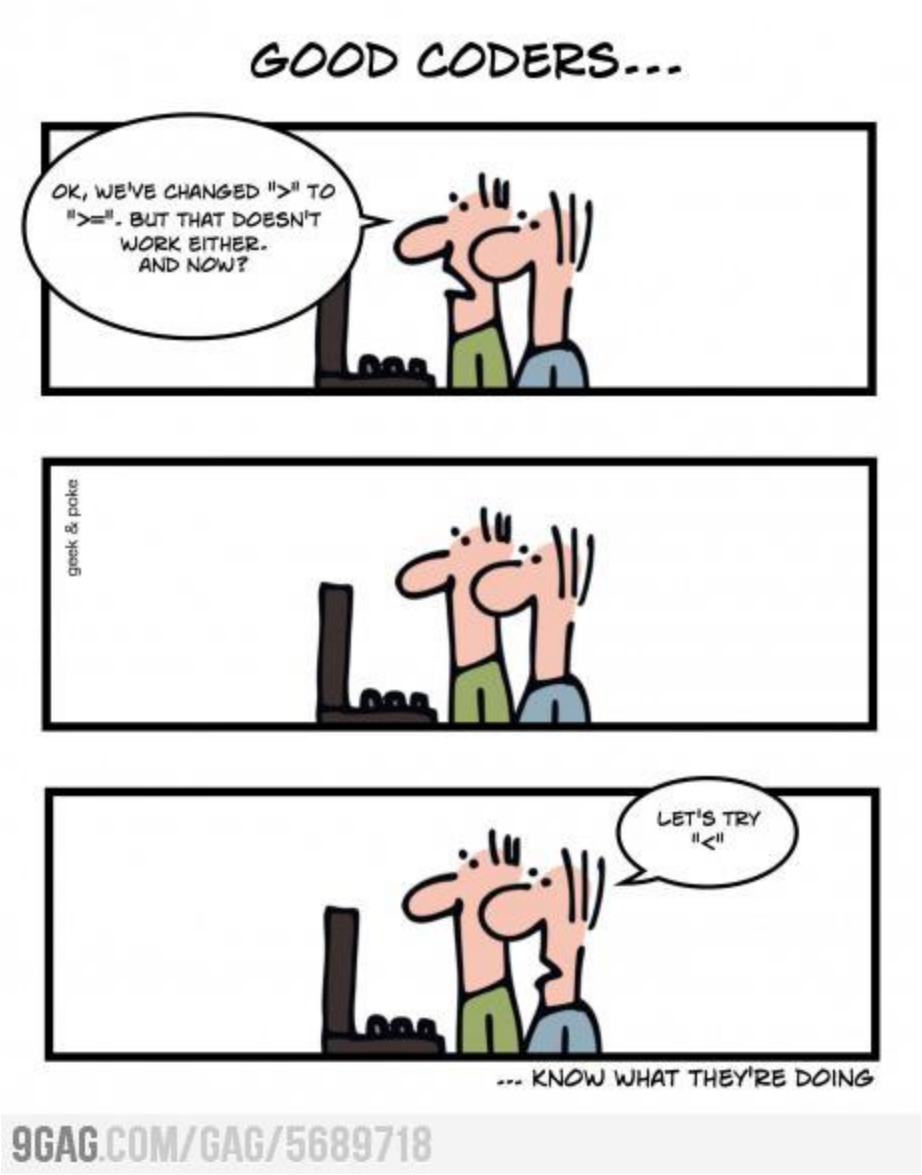
Give me a data structure that I can use to maintain a bunch of people's names and for each person, allows me to easily get all of the streets they lived on.

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 (names and addresses in alpha-order).
D. Printing out all of the streets that "Tala" has lived on.

There are several possible answers:
1. map<string,set<string>>
1.A. P * $\log_2(E)$
1.B $\log_2(P)$ + $\log_2(E)$
1.C. P*E
1.D $\log_2(P)$ + E
2. map<string,vector<string>> or map<string,list<string>>
2.A. P * E
2.B P + E
2.C. P * $\log_2(P)$ + P*E*$\log_2(E)$
2.D P + E


GOOD CODERS...
OK, WE'VE CHANGED ">" TO ">=". BUT THAT DOESN'T WORK EITHER. AND NOW?
geek & poke
LET'S TRY "<"
... KNOW WHAT THEY'RE DOING
9GAG.COM/GAG/5689718

Divide & Conquer Sorting Algorithms

What's the big picture?

Quicksort and Mergesort are efficient "divide and conquer" sorting algorithms.

They 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) using *recursion*.
3. Combine the two sorted groups into one large sorted group.

These sorts generally require $O(N*\log_2(N))$ steps.

Uses:

Used in virtually every C++ program that needs to order data.

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.

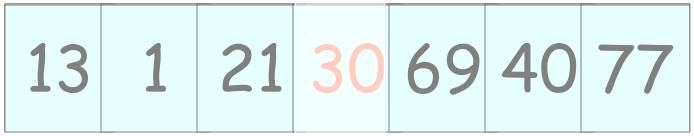
2. Select an arbitrary element P from the array (typically the first element in the array).

Divide

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).

Conquer

4. Recursively repeat this process on the left sub-array and then the right sub-array.



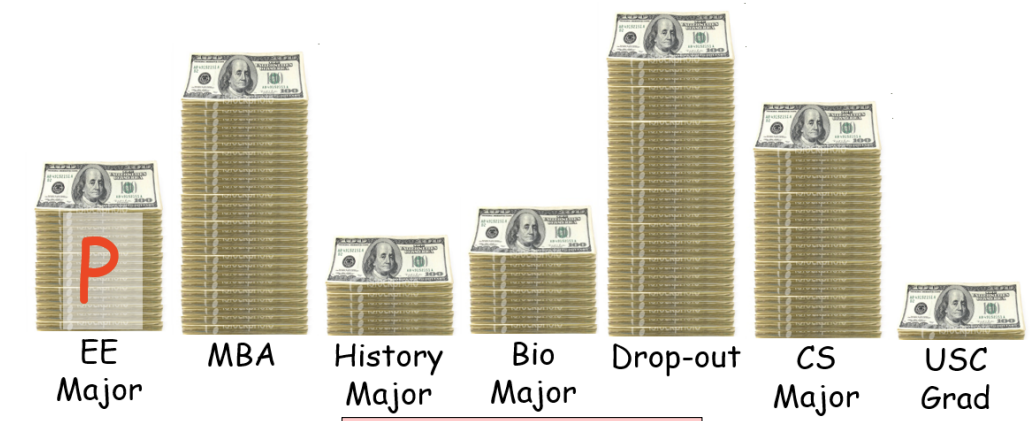
QuickSort

Select an arbitrary item P from the array.

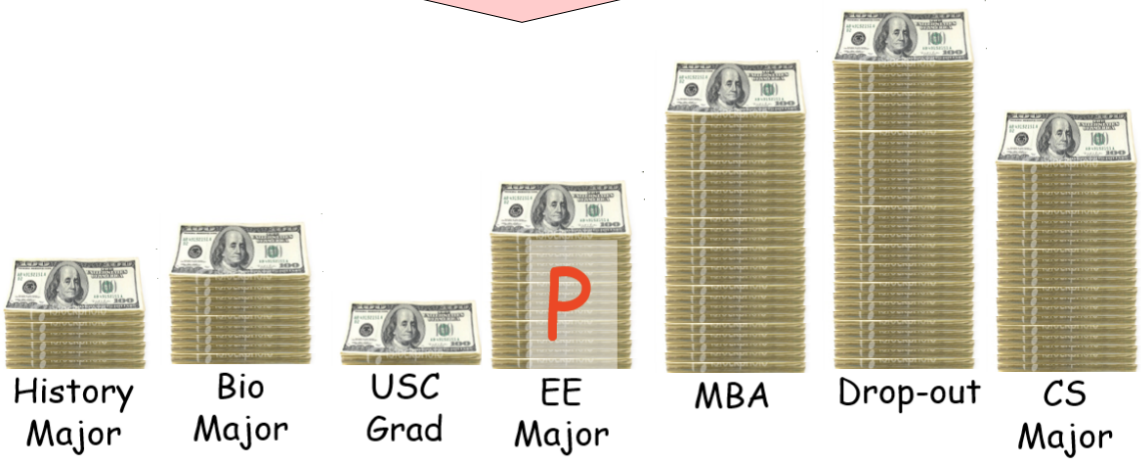
Move items smaller than or equal to P to the left and larger items to the right; P goes in-between.

Recursively repeat this process on the left items

Recursively repeat this process on the right items



After the first shift



- The top row of piles is the initial configuration before any sorting has taken place.
- The second row of piles is after we've selected EE major as our "arbitrary pile" P and moved all shorter or equal-height piles to the left, and all taller piles to the right
- Notice that while the second row is not fully sorted, pile P (EE major) is actually in the right place – it never needs to be moved again.
- Why? Because every pile left of P is smaller than P, and every pile than it is greater than it. So P is in the perfect position – the position it'll be in once everything is completely sorted.
- This means that we can independently sort the left three piles, then independently sort the right three piles, leaving P as-is.
- And then everything will be 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

History Major | Bio Major | USC Grad | EE Major | MBA | Drop-out | CS Major

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

USC Grad | History Major | Bio Major

- This slide shows us recursively sorting the left three piles.
- Again, we pick an arbitrary pile P (in this case, History major) and then move everything less than or equal to the left, and everything taller to the right.
- Since there are only three items, this results in the left part of the array being sorted!
- But if there were more items, we'd repeat this process over and over

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

Finally, all items are sorted!

Everything right of EE 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);
}
```

- Select the first item as our pivot value
- Find next value on the left that is > than the pivot.
- Find next value on the right <= than the pivot.
- Swap the two out of place items
- Swap our pivot into the right spot
- Return the slot # of our pivot item in the array

Big-oh of Quicksort

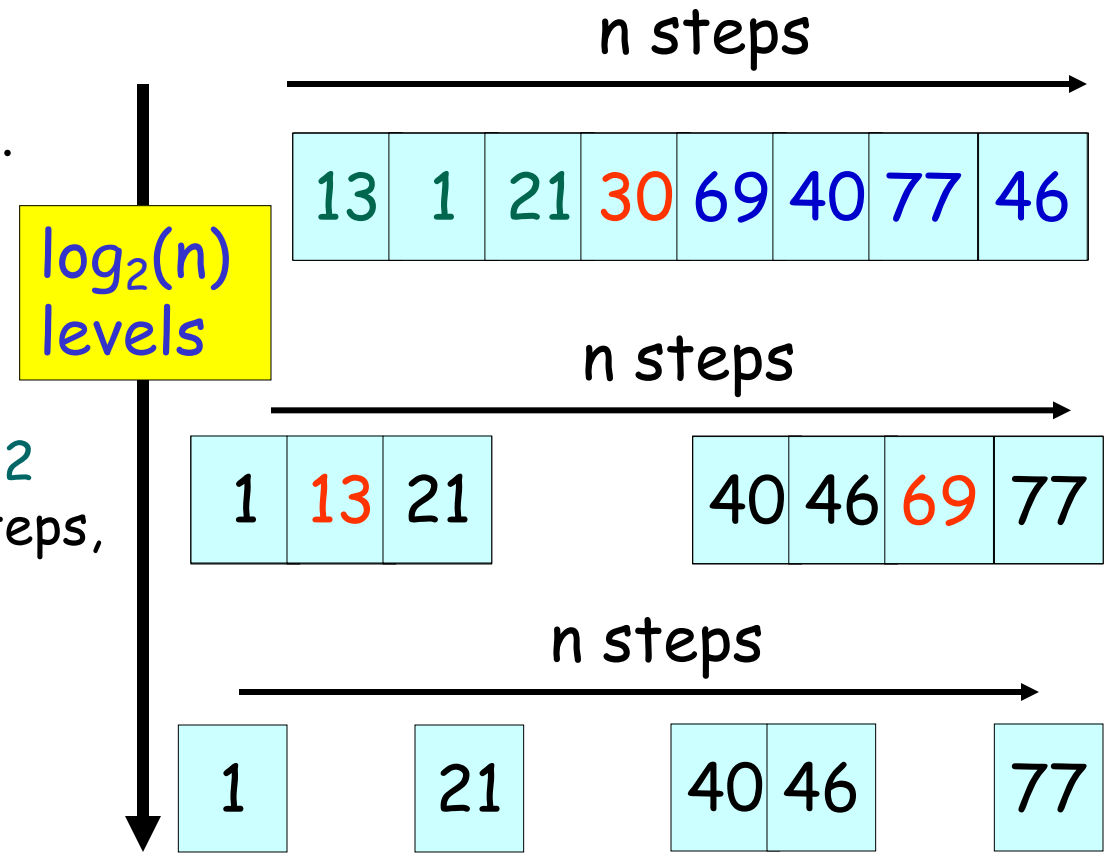
We first partition the array, at a cost of n steps.

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.



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.

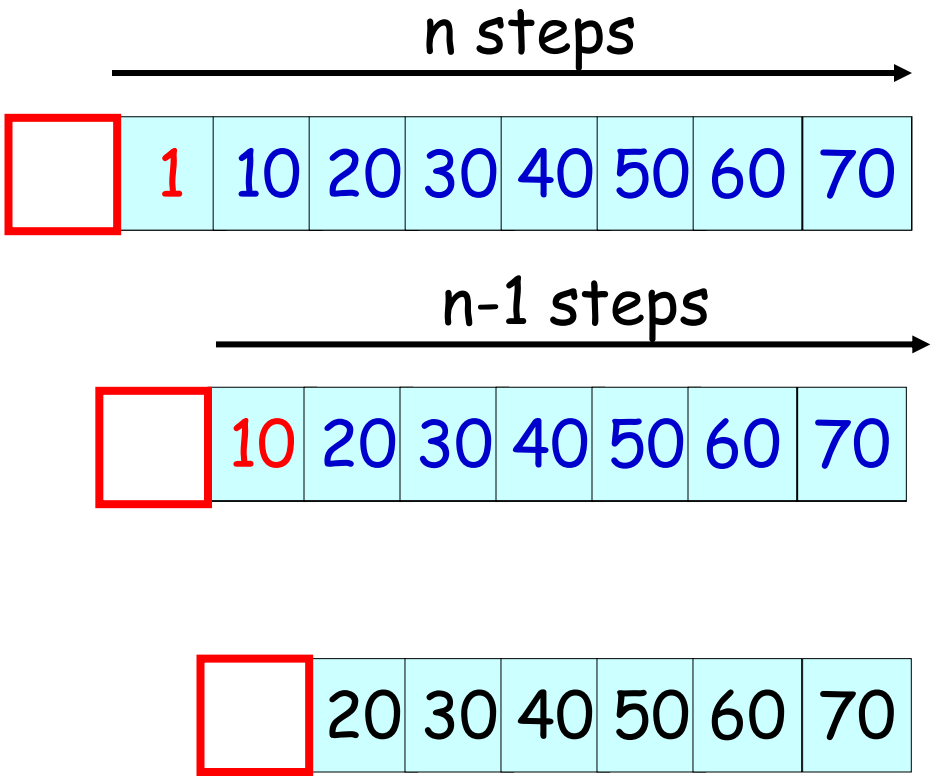
Worst-case Big-oh of Quicksort

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…



- When an array is already sorted, the smallest item will always be on the left
- So if we choose the first item as the pivot P, after our partition alg. P will stay all the way on the left!
- So rather than having roughly half the array moved left of the pivot P and half on the right side as we saw in our example with piles of cash, we'll have N-1 items on the right side of P and zero to its left!
- So now when we do recursion on the left side there's nothing to do, since there are zero items less than the pivot P to sort…
- And when we do recursion on the right side, we have N-1 items still to sort.
- So to fully sort the array, we have to recurse down N-1 levels deep!!

Worst-case Big-oh of Quicksort

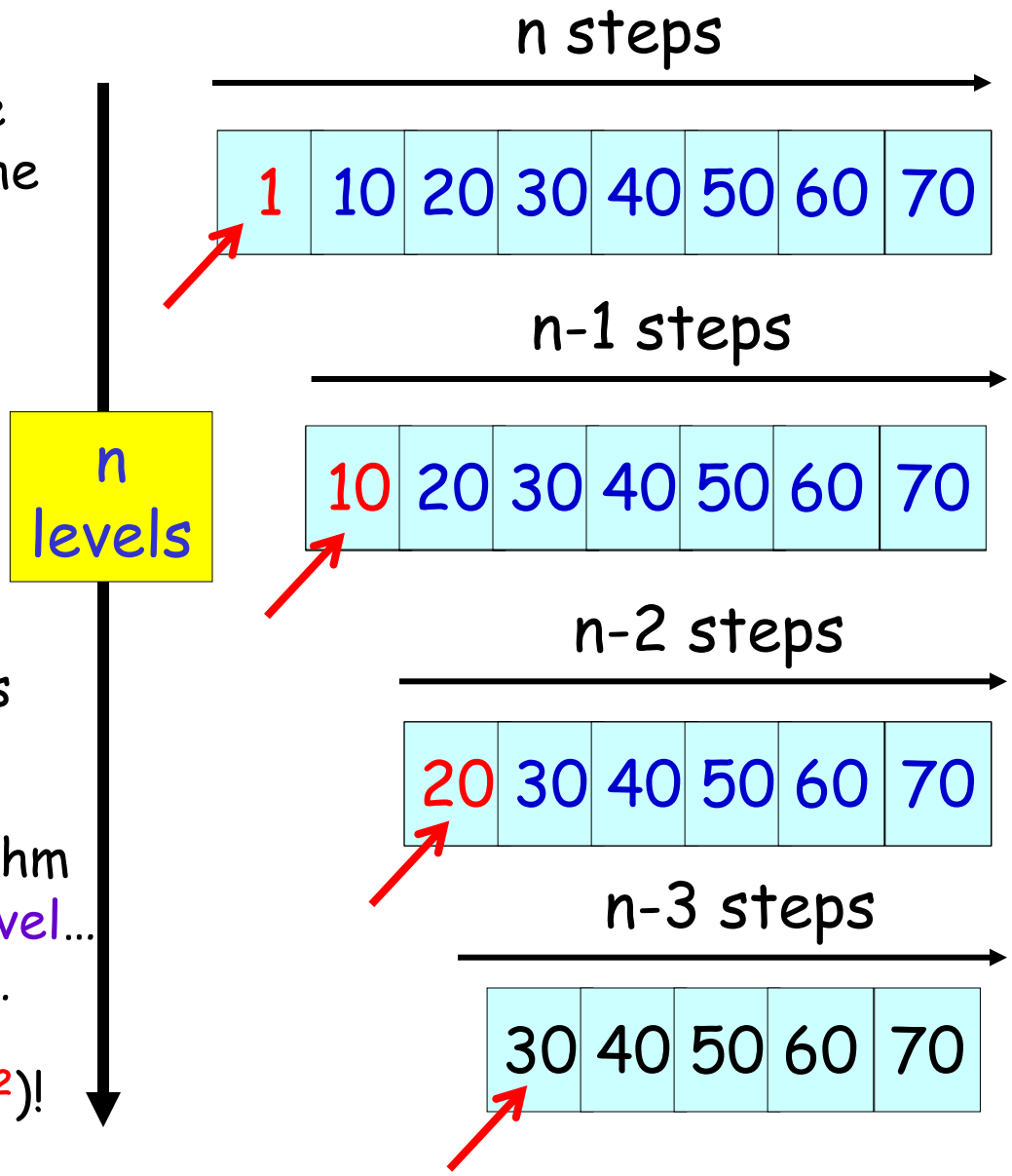
What you'll notice is that each time we partition, we remove only one item off the left side!

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 ~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

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

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

Merge Algorithm in C++

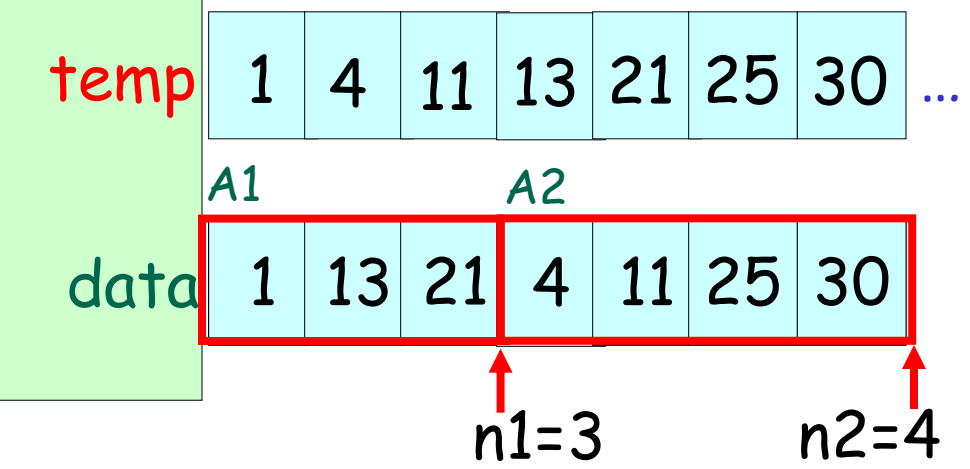
```
void merge(int data[], int n1, int n2,
           int temp[])
{
  int i1=0, i2=0, k=0;
  int *A1 = data, *A2 = data + n1;

  while (i1 < n1 || i2 < n2)
  {
    if (i1 == n1)
       temp[k++] = A2[i2++];
    else if (i2 == n2)
       temp[k++] = A1[i1++];
    else if (data[i1] <= A2[i2])
       temp[k++] = A1[i1++];
    else
       temp[k++] = A2[i2++];
  }
  for (int i=0;i<n1+n2;i++)
    data[i] = temp[i];
}
```

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

You pass in an input array called data and the sizes of the two parts of it to merge: n1 and n2

The last parameter, temp, is a temporary array of size n1+n2 that holds the merged results as we loop.

Finally, we copy our merged results back to the data array.



Mergesort

OK – so what's the full mergesort alogrithm:

Mergesort function :

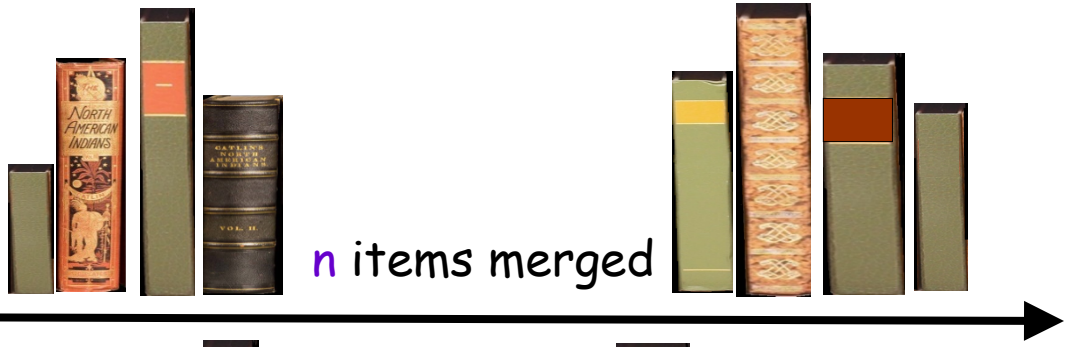
1. If array has one element, then return (it's sorted).
2. Split up the array into two equal sections
3. Recursively call Mergesort function on the left half
4. Recursively call Mergesort function on the right half
5. Merge the two halves using our merge function

It's difficult to show mergesort visually in static slides, so if you want to see it in action, download my PPT slides: lecture11-updated.pptx on www.careynachenberg.com

Big-oh of Mergesort

- This is visually how mergesort divides its piles.
- It divides the initial array in half, then recursively calls itself on each half to sort them, then merges the sorted two piles into one big pile
- Of course each of those halves is further broken in half, and passed to another recursive call, and so on.
- This breaking in half happens until we reach a single book as we see in the bottom row.
- Then we merge the sorted piles on the way back up
- We start by merging just two books, one book from the left pile and one book from the right pile (see the bottom row)
- At the next level up we'll merge two books from the left pile with two books from the right pile
- Then up a level we'll merge four books from the left pile and four from the right pile
- And so on…

Big-oh of Mergesort



- Note that if there are N total values to sort, we'll keep breaking the array in half until we get arrays of just 1 value each.
- That will be $\log_2 n$ levels deep, which is the # of times we can divide N by two until we get to 1.

- On the way back up, we merge each of the arrays.
- On each row, we merge N total values (it's O(N)).
- That might not be obvious, but it's what happens.
- On the bottom row, we merge N arrays of 1 value each together into N/2 arrays of two values each. That's O(N) steps
- On the second-to-last row, we merge N/2 arrays of 2 values each together into N/4 arrays of four values each. That's also O(N) steps
- And so on, until we merge the top two arrays of size N/2 into a single array of size N. That's also O(N) steps
- So log(N) levels of O(N) merges per level is N*logN

Big-oh of Mergesort

n items merged

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

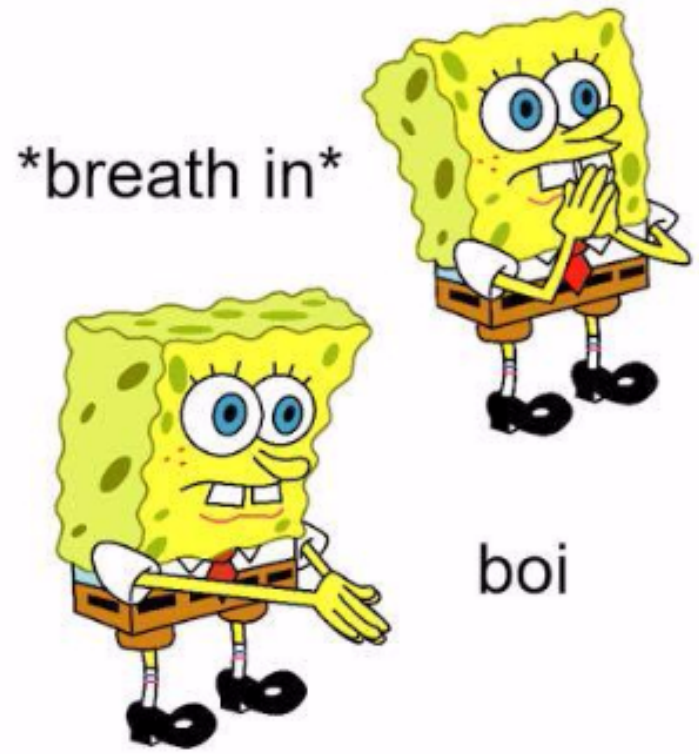
Sort Name	Stable/ Non-stable	Notes
Selection Sort	Unstable	Always $O(n^2)$, but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow)
Insertion Sort	Stable	$O(n)$ for already or nearly-ordered arrays. $O(n^2)$ otherwise. Can be used with 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 $O(n)$ 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 xtra 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

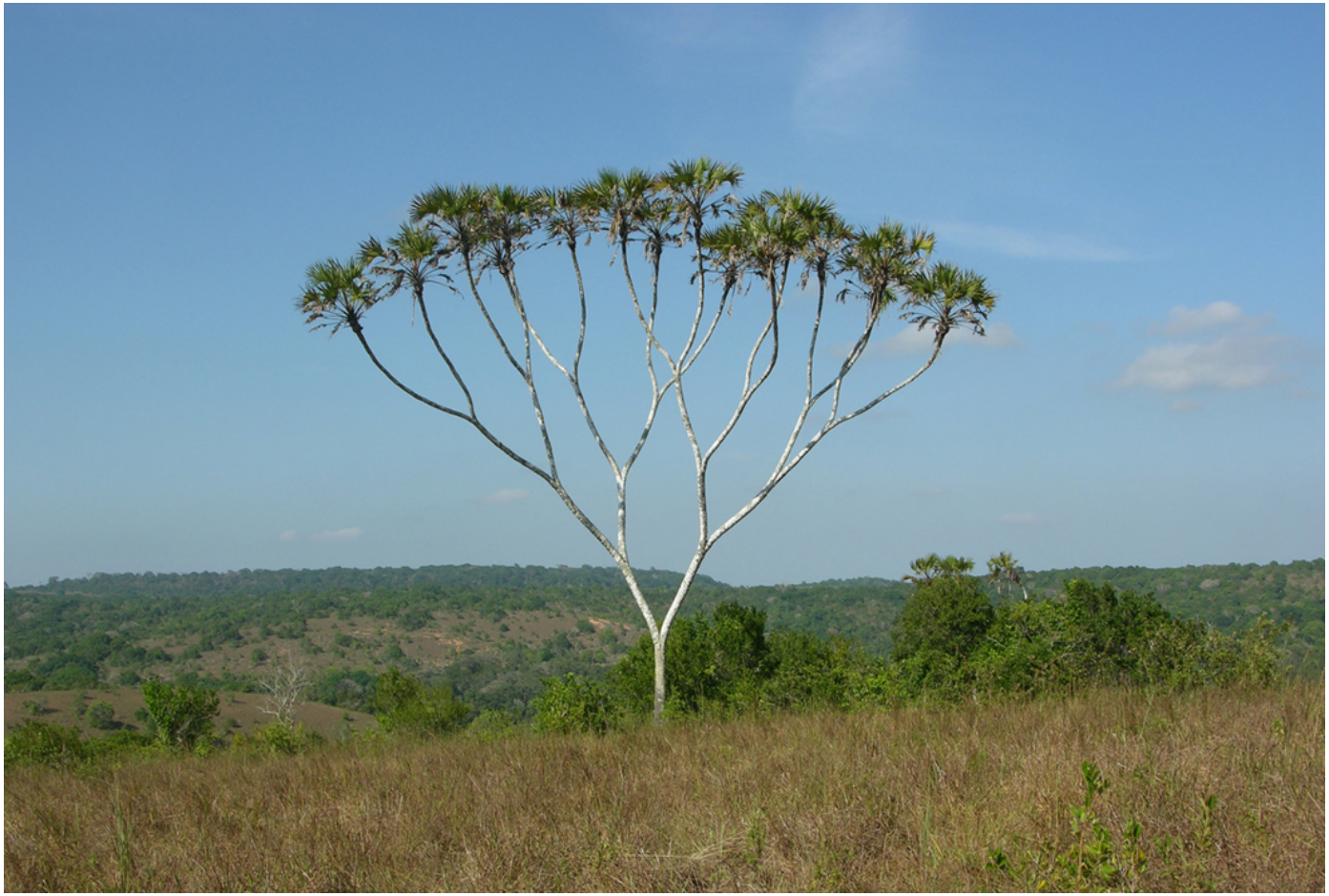
1. Give an algorithm to efficiently determine which element occurs the largest number of times in the array.

2. What's the best algorithm to sort 1,000,000 random numbers that are all between 1 and 5?

when your code is meant to be O(NlogN) but it's been 30 minutes and it still hasn't finished N=3



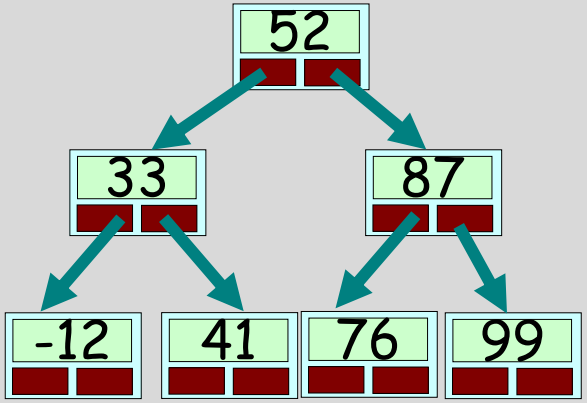
Trees



Tree Data Structures

What's the big picture?

A tree is a data structure that stores values in a hierarchical fashion, e.g.,



We often use linked lists to build trees. For instance, the tree above has nodes with two "next" pointers - one going left and one right.

Trees are an alternative to linked lists and arrays when you need more organization of your data.

Uses:

Efficient searching, compilers, generating spelling suggestions, processing graphical images (quadtrees), etc!

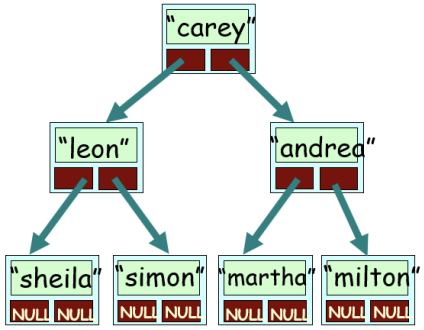
Trees

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

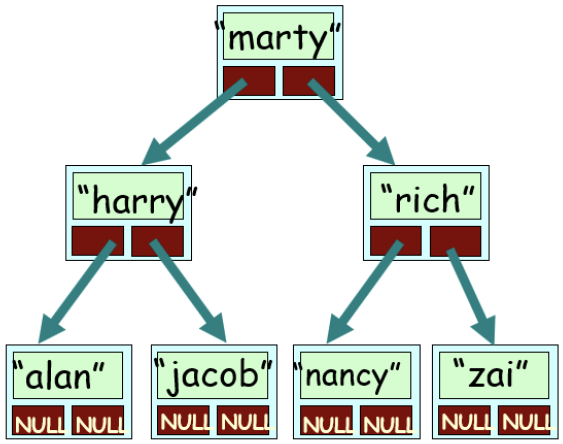
A Tree is a special linked list-based data structure that has many uses in Computer Science:

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

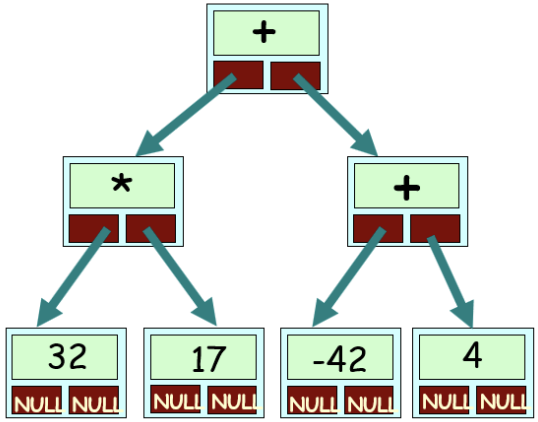
A Family Tree



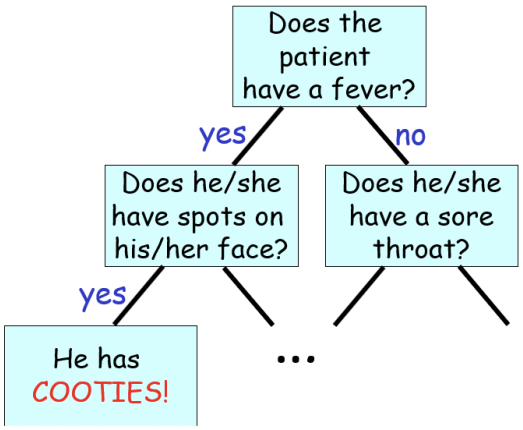
A Binary Search Tree



An Expression Tree

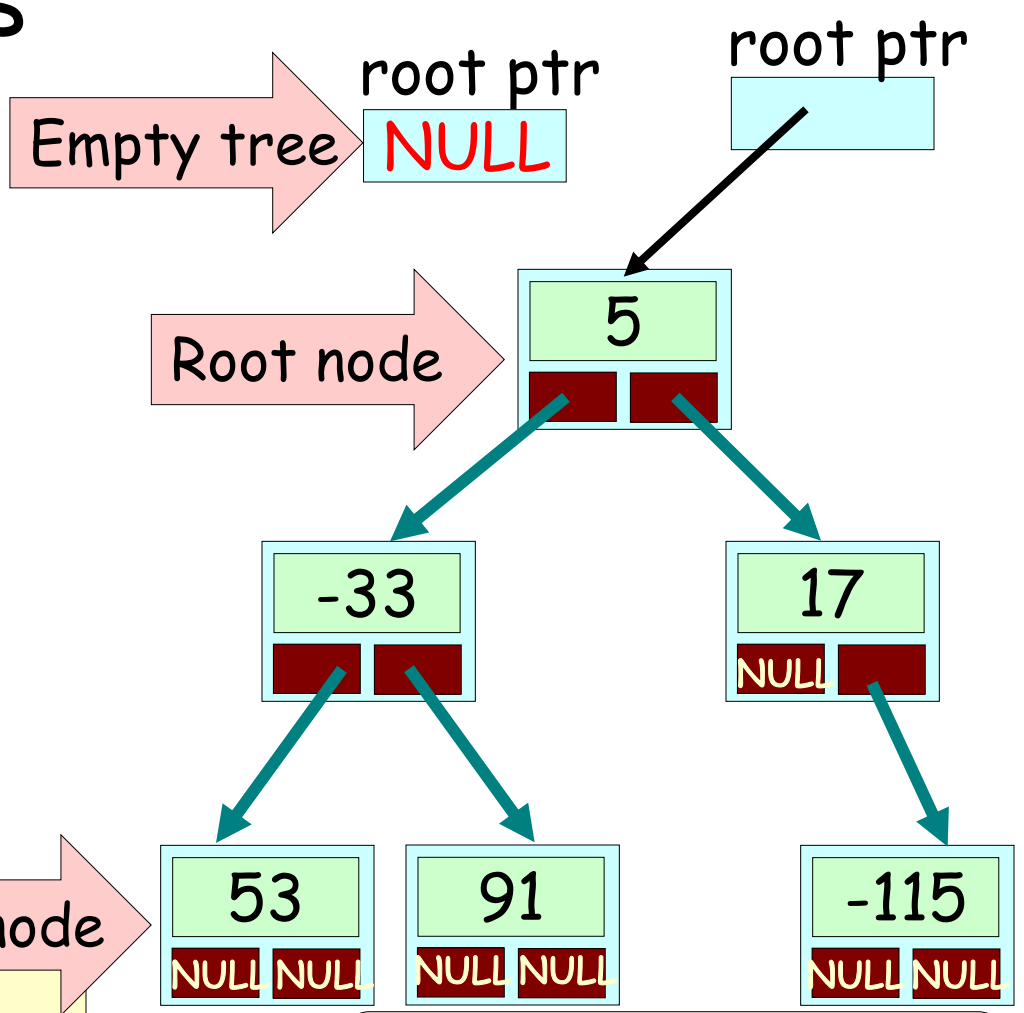


A Decision Tree



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;
```

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

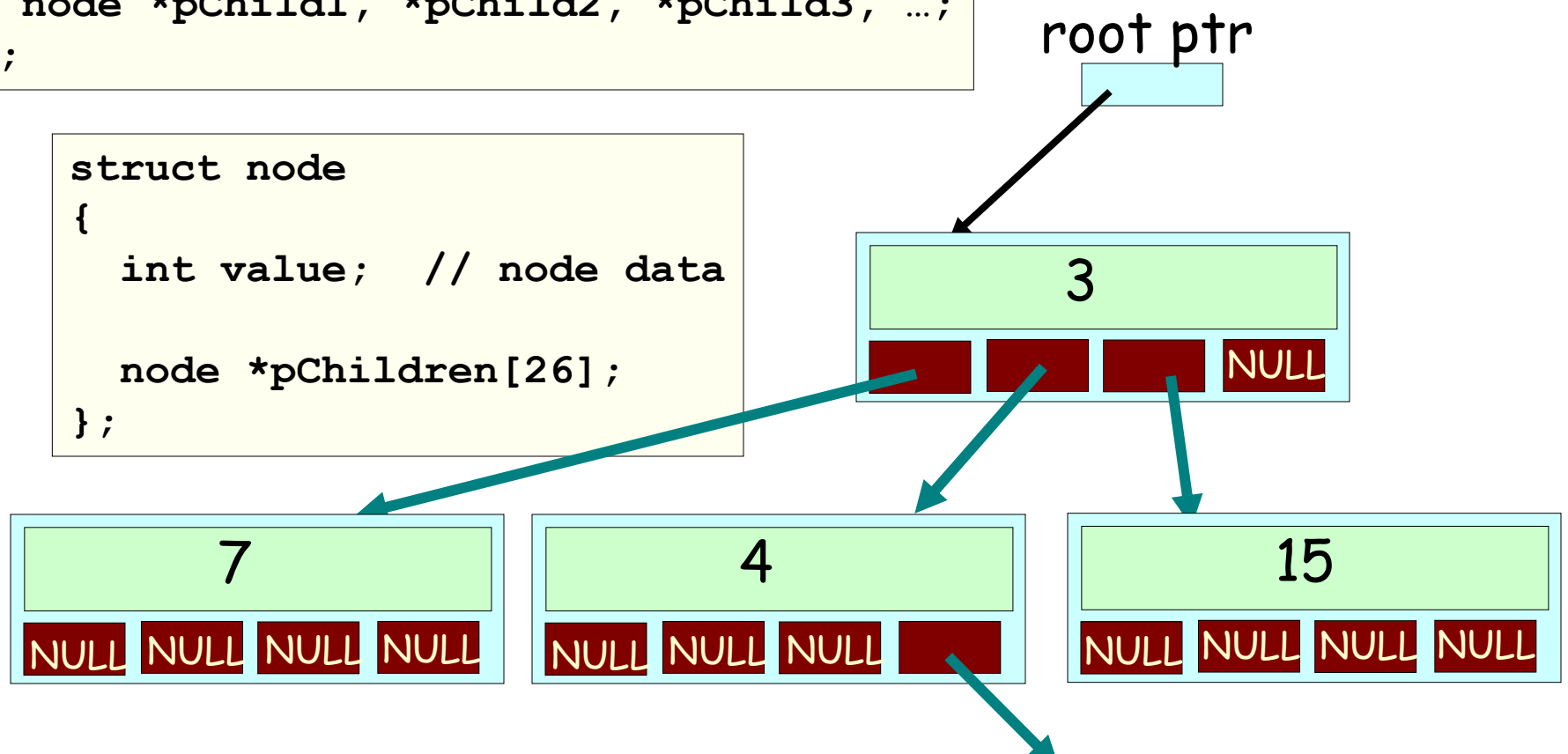
The tree's root pointer is like a linked list's head pointer!

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];
};
```



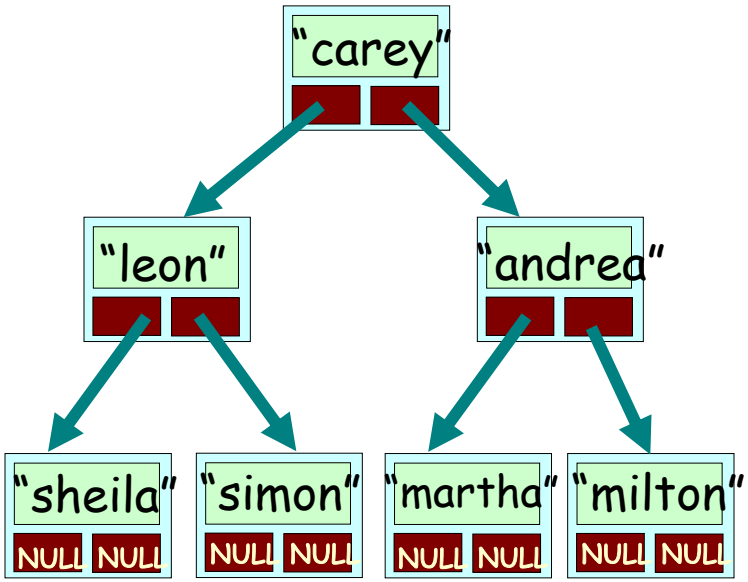
Binary Trees

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

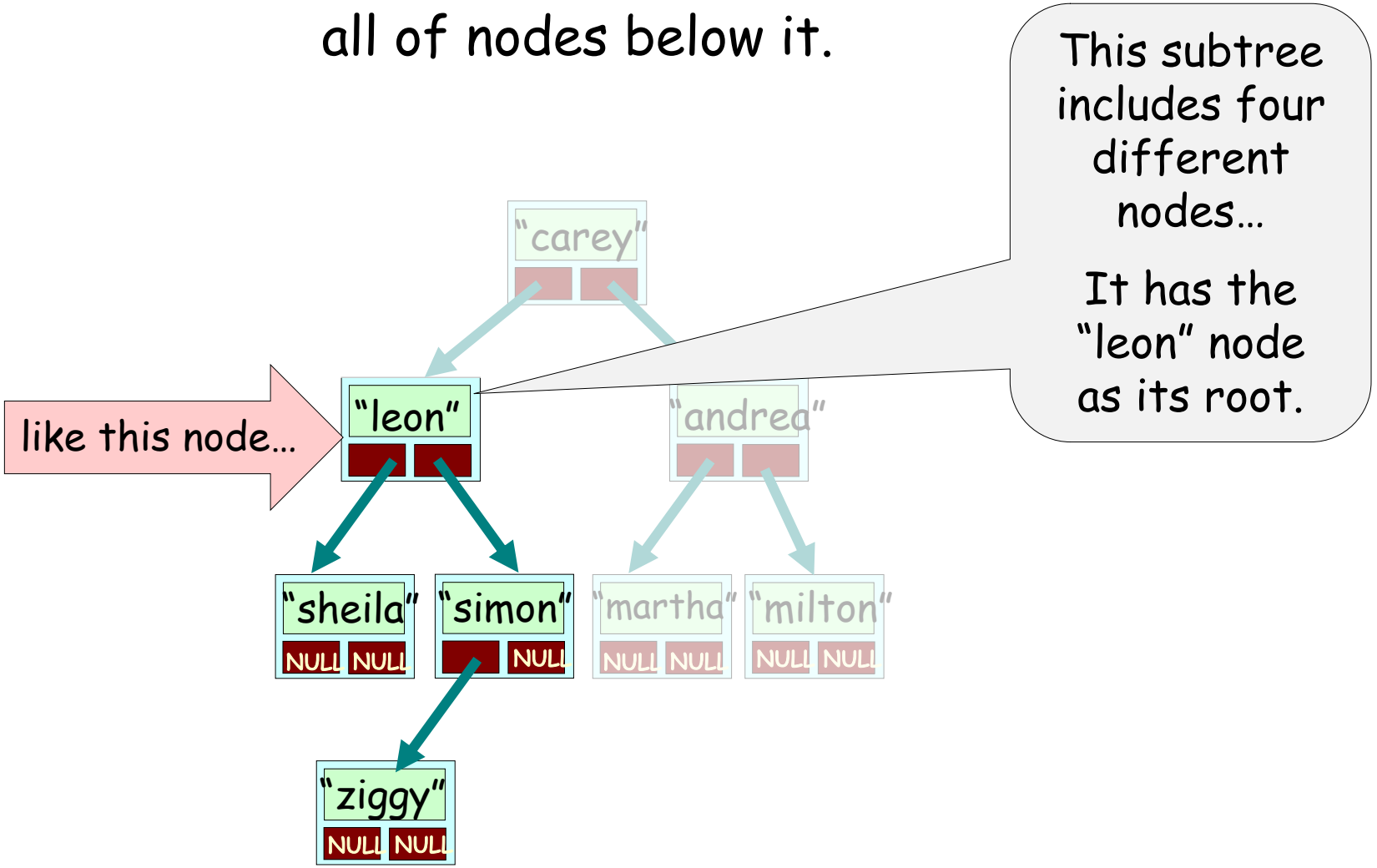


- It's important to note that not every binary tree is a binary search tree.
- For instance, the tree to the right is a binary tree but NOT a binary search tree.
- The only criteria required to have a binary tree is that each node has two children nodes.
- In contrast, a binary SEARCH tree is a binary tree where the organization of the nodes follows certain ordering rules.

Binary Tree Subtrees

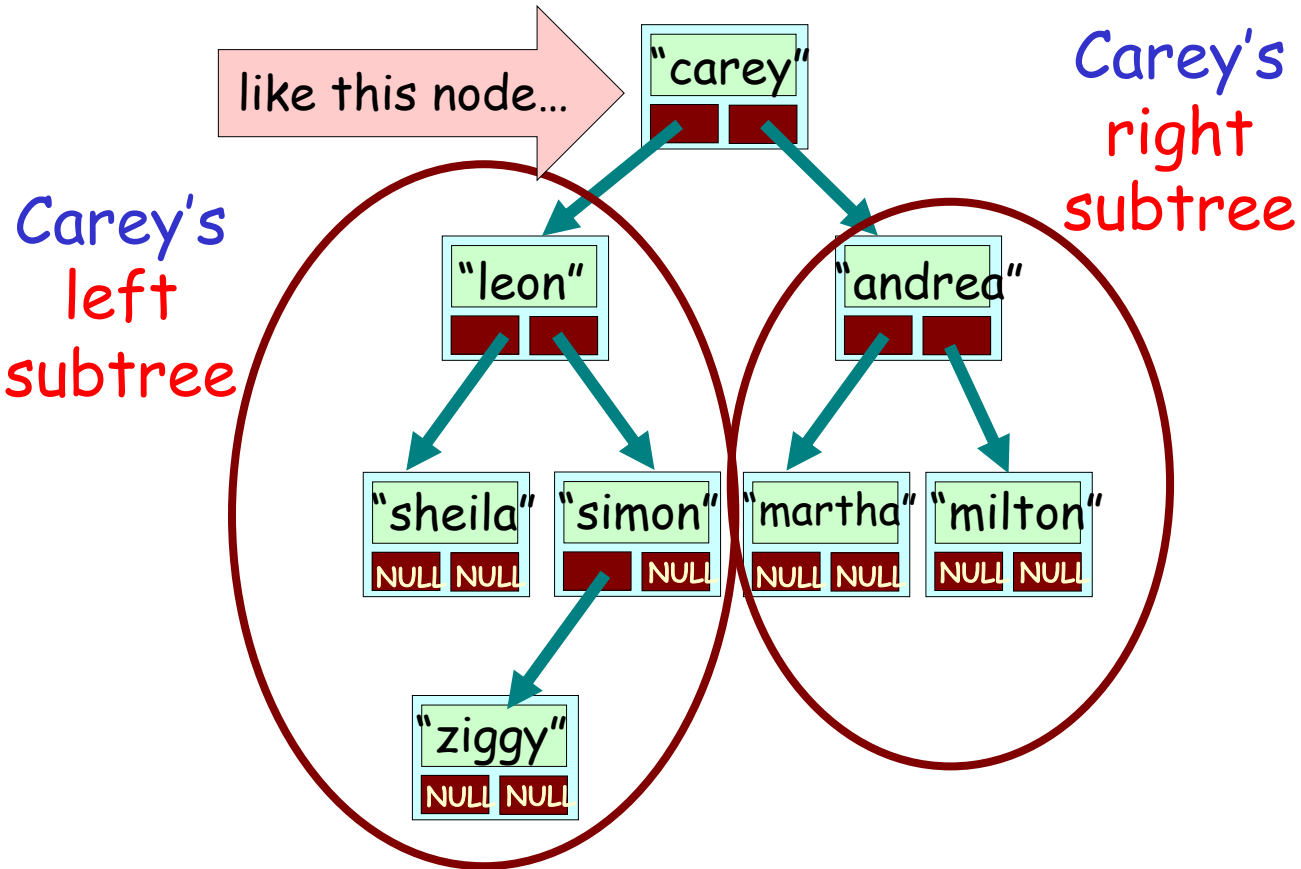
We can pick any node in the tree…

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



Binary Tree Subtrees

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



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**)

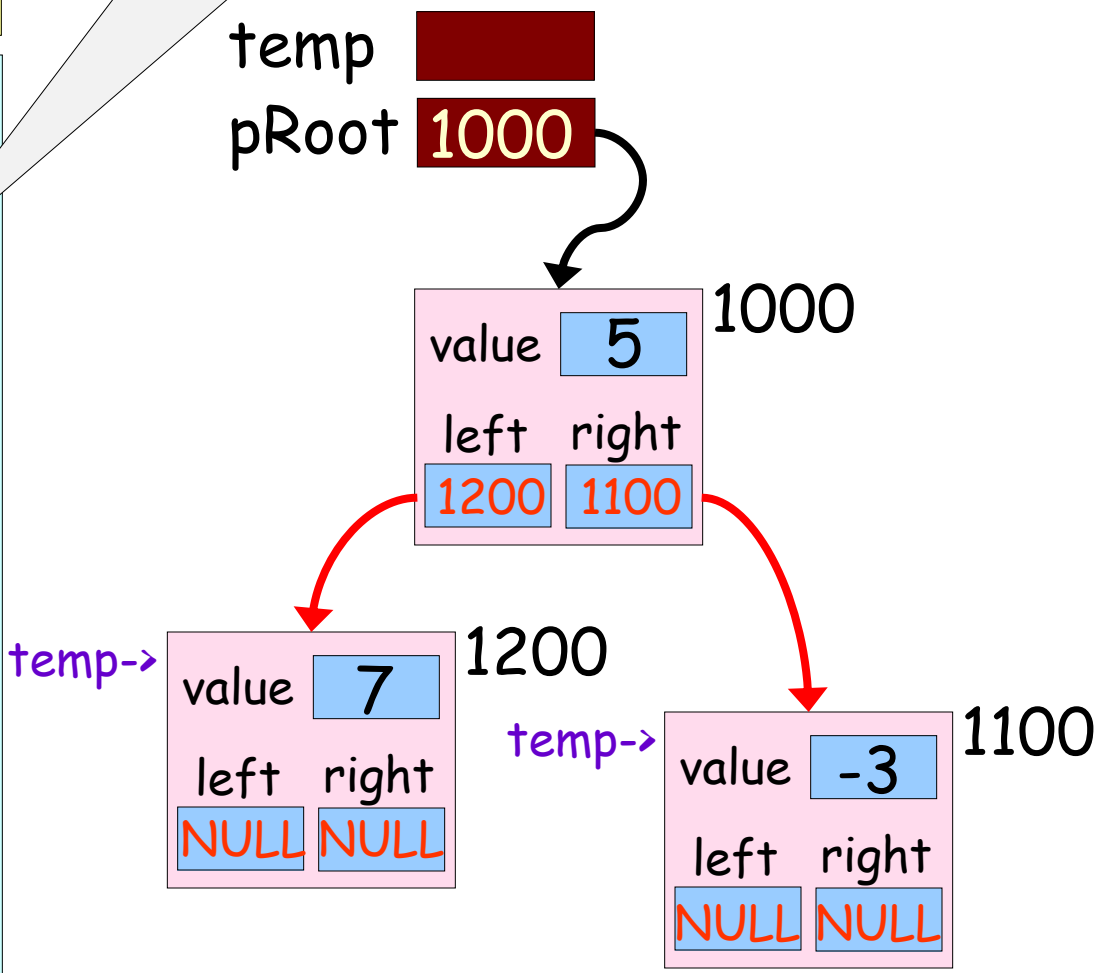
We'll learn about many of these operations over the next two classes.

A Simple Tree Example

```
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.



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.

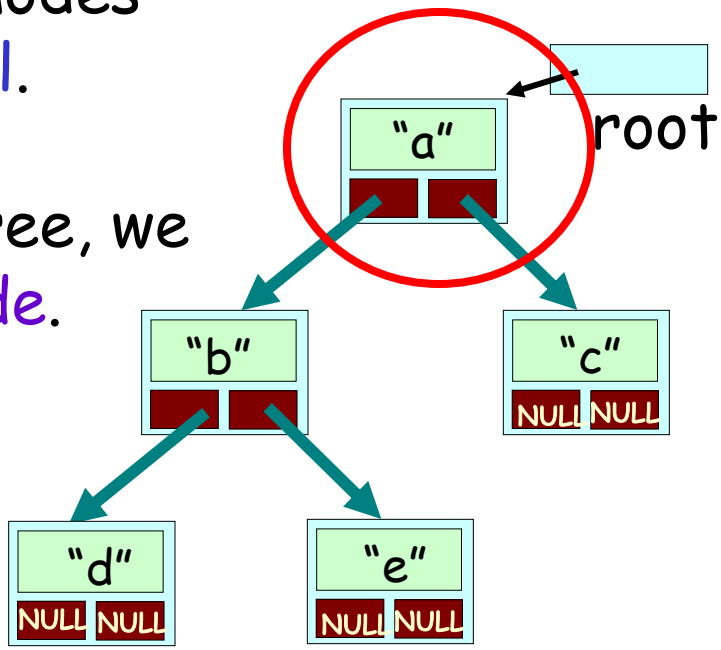
This is called "traversing" the tree, and there are several ways to do it.

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

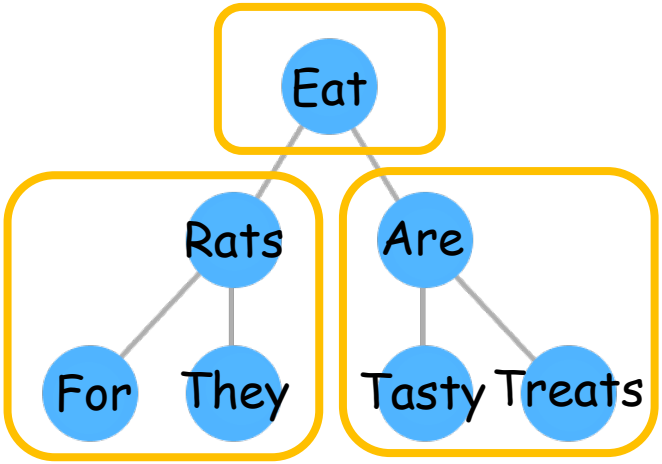
The Preorder Traversal

By process, we mean things like…

- Print the node's value out
- Search the node for a particular value
- Add the node's value to a total

PreOrder(node):

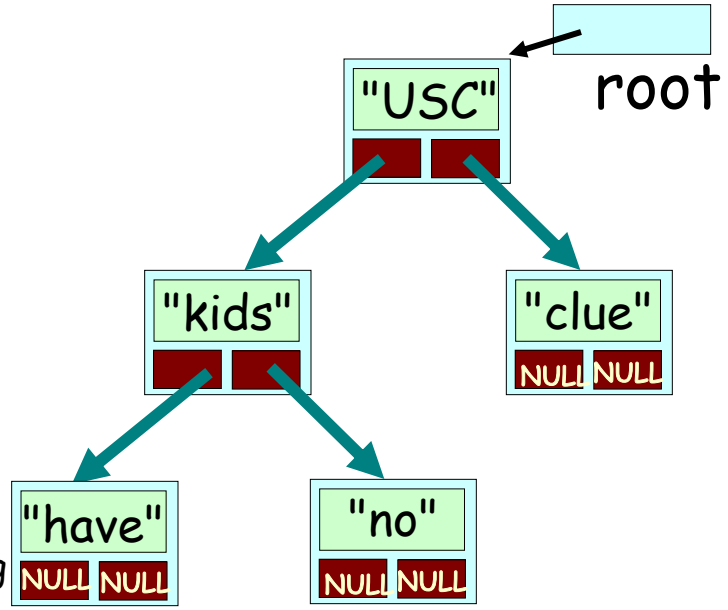
1. Process the current node.
2. Recursively call PreOrder on the **left** sub-tree.
3. Recursively call PreOrder on the **11** sub-tree.



- The PreOrder traversal is a recursive traversal that processes all of the nodes in a tree.
- Can you guess why it's called a "pre-order" traversal?
- Because at each node, we pre-process the current node before processing the node's left and right subtrees.
- So, for example, when we start at the "Eat" node, we process "Eat" first, then process the "Rats" subtree in its entirety, then process the "Are" subtree in its entirety.
- And the algorithm is asked to process the "Rats" node, it processes it first, then process the "For" subtree in its entirety, then process the "They" subtree in its entirety
- So the order the nodes would be processed by a pre-order traversal would be:
  - Eat, rats, for, they, are, tasty, treats

The Pre-order Traversal

Output: USC kids have no clue



- Below we see the PreOrder function – look how simple it is!
- The first line "if (cur == nullptr)" checks for the base case. If we are passed an empty tree/subtree, then we just return and do nothing. This is a super-common pattern for tree-based recursion. Always include a check for nullptr.
- Then we process the current node's value, in this case, printing it out
- Finally, we recursively call ourself on the left child of the current node (the root of the left subtree)
- When that's done, we recursively call ourself on the right child of the current node to process the right subtree.

```
void PreOrder(Node *cur)
{
    if (cur == nullptr)          // 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 right sub-tree.
}
```

```
main()
{
→ Node *root;

→ …

→ PreOrder(root);
}
```