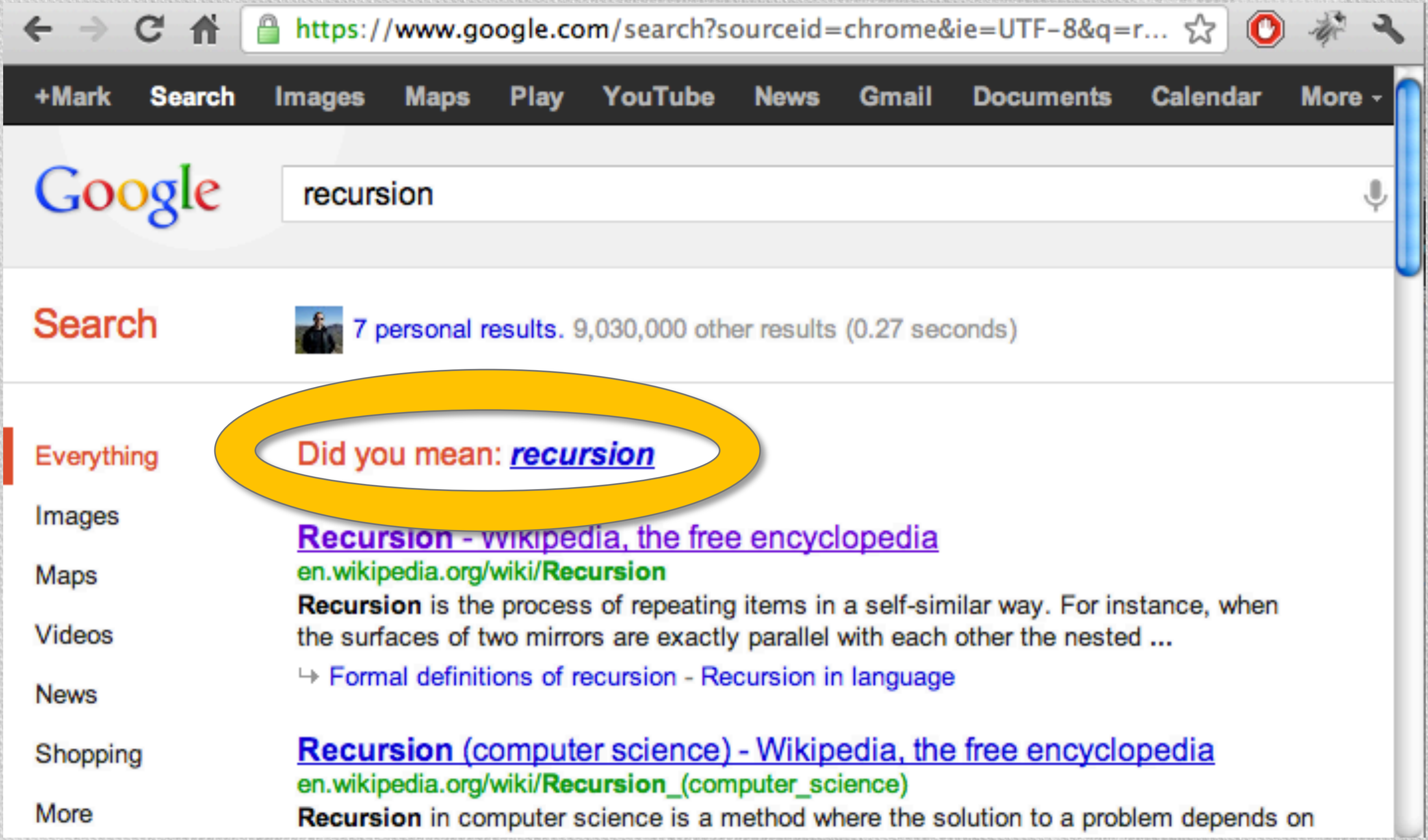
CS 211

RECURSION

Recursion



7 personal results. 9,030,000 other results (0.27 seconds)

Did you mean: ***recursion***

Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/**Recursion**
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested ...
↳ Formal definitions of recursion - Recursion in language

Recursion (computer science) - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/**Recursion**_(computer_science)
Recursion in computer science is a method where the solution to a problem depends on

What is **Recursion**?

- Recursion generally means that something is defined in terms of itself.

→ functions/methods can be recursive

    → if it calls itself

→ data can be recursive

    → if a class "has-a" field of its own type

Method Recursion

- We can call a method inside its own body.
- The **recursive call** should logically solve a "smaller" problem
- We must have some way to stop, called a **base case**. (It should be checked *before* the recursive call!)
  → otherwise, it's just like an infinite loop!

Example: Factorial

- In mathematics, the factorial n! is defined as $n!=n*(n-1)*\ldots*2*1$. It is defined for all non-negative numbers, and $0! = 1$. Examples:

$5! = 5*4*3*2*1$ **$100! = 100*99*98*\ldots*3*2*1$**

$3! = 3*2*1$ **$1! = 1$** **$0! = 1$**

- The **Base Case** is when n=0: we immediately know the answer. No recursion is necessary.
- The **Recursive Case** is when n>0: we know that whatever value n has, (n-1) will be one step closer to the base case of n=0.
  → assume the method is already correct; phrase $n! = n*(n-1)!$
  → call our method on (n-1), and multiply it by n.
  → let the recursive call do the rest!

Example: Factorial

```
public static int factorial (int n) {


        //base case, no recursion
        if (n==0) {   return 1;   }

        //recursive case: n! = n*(n-1)!
        else {
                int nfact = n *  factorial(n-1);
                return nfact;
  }
}
```

Recursive Calls: Details

- When a method calls itself, each call is distinct (separate)
  → each separate call has its ***own copy of local data***
  → for `factorial`, each call has its own value for parameter `n`.

Base Case Reached! Non-recursive call can complete.

factorial (0)	0! == 1
factorial (1)	1! == 1*0! == 1
factorial (2)	2! == 2*1! == 2
factorial (3)	3! == 3*2! == 6

Recursion Recipe

- To use recursion, you might want to follow this pattern:

1. Identify the base cases: times when you already know the answer
2. Identify the recursive cases: times when you can define one step of the solution in terms of others
   - Is the recursive step using the method on a "smaller" problem? (needs to be yes!)

3. Write code for the base case *first*
4. Write code for the recursive case *second*

→ handle any error conditions like base cases: e.g., factorial shouldn't be called on negative numbers, so choose how to exit meaningfully.

Recursion Example: Fibonacci

- The fibonacci sequence looks like:

  1, 1, 2, 3, 5, 8, 13, …

  → Its first two elements are each 1.

  → the nth element is the sum of the previous two elements.

- We can number them like array slots:

  fib(0)==1, fib(6)==13, etc.

Practice Problems

Implement the fibonacci method, which accepts the 'index' number `n`, and then returns that fibonacci number.

Questions (suggested solution path):

- What are the base cases?
- What are the recursive cases?
- What are some good test cases?

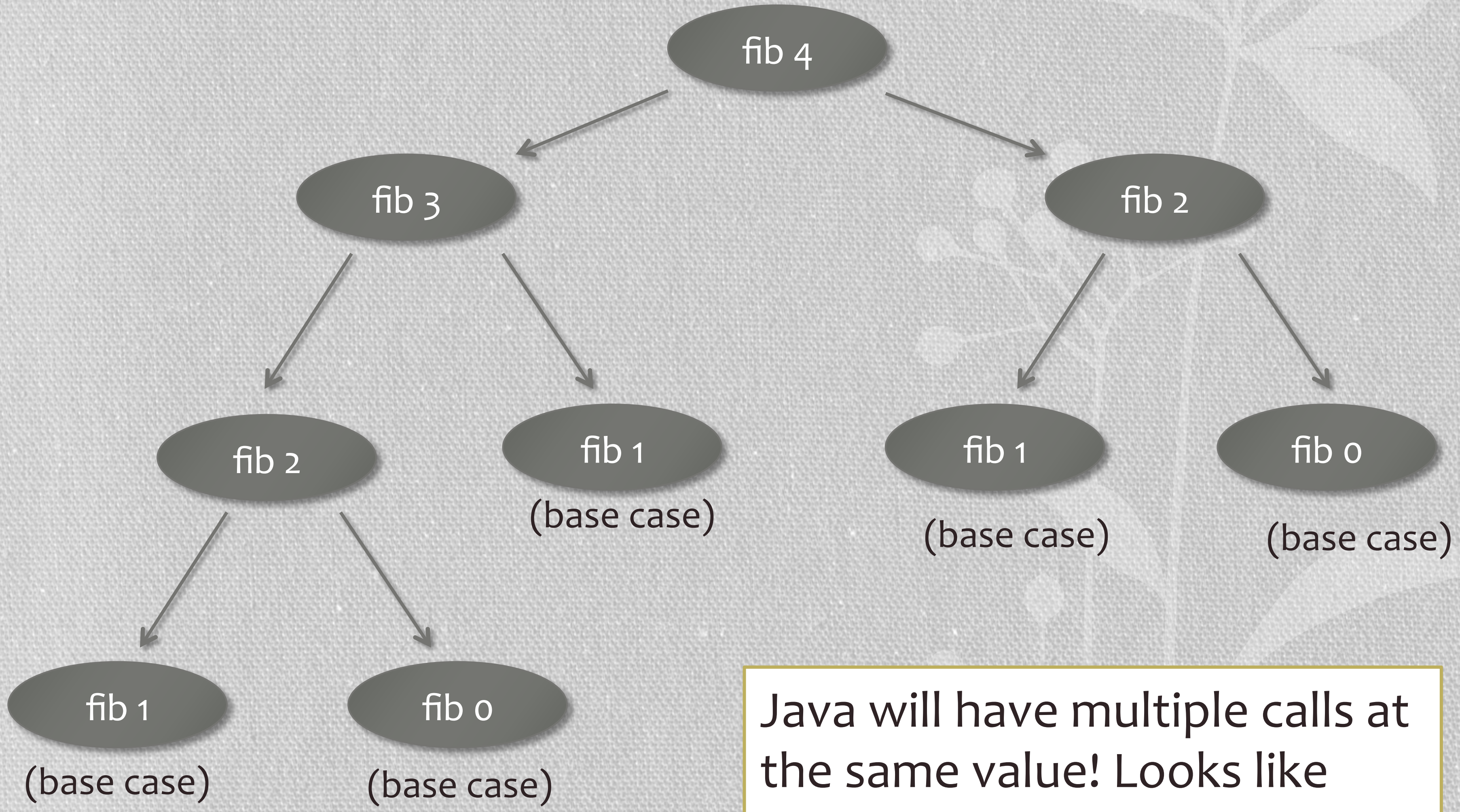
Fibonacci Code

```
public static int fib (int n) {

        // base cases
        if (n==1 || n==0) {  return 1;  }

        //recursive case
        else {
                return  fib(n-1) + fib(n-2) ;
        }
}
```

Visualizing Fibonacci Calls



Java will have multiple calls at the same value! Looks like wasted effort…

Iterative Version of Fibonacci

```
public static int fibIter (int n) {
        //base cases
        if (n==1 || n==0) { return 1; }

        //iterative cases
        int lower = 1;
        int higher = 1;
        for (int i = 2; i <=n; i++ ) {
            int temp = lower+higher;
            lower = higher;
            higher = temp;
        }
        return higher;
}
```

Considering Recursion

Recursion: **Pros**

- Sometimes much easier to reason about
- distinct method calls help separate concerns (separate our local data per call).
- Easy to maintain separate state (values) each recursive call

Recursion: **Cons**

- Sometimes, lots of work is duplicated (leading to quite long running time)
- Overhead of a method call is more than overhead of another loop iteration

Considering Iteration

Iteration: **Pros**

- quick, barebones.
- Simpler control flow (we perhaps see how iterations will follow each other easier than with recursion)
- no stack overflow errors

Iteration: **Cons**

- some tasks do not lend well to iterative definitions (especially ones with lots of bookkeeping/state)
- We tend to be given mathematical, "recursive" definitions to problems, and then have to translate to an iterative version.

Recursion versus Iteration

So, which one is better?

→ it depends on the situation.

- When might we prefer recursion?
- When might we prefer iteration?

Practice Problems

What are some cases that might merit recursion?

Thinking experiment: How can you use recursion to...
→ check if a number is even?
→ find the log of a number? (return an int)
→ solve a maze?
→ solve a sudoku?

Practice Problems

- How, in general, might we try to convert a loop to a recursive method call?

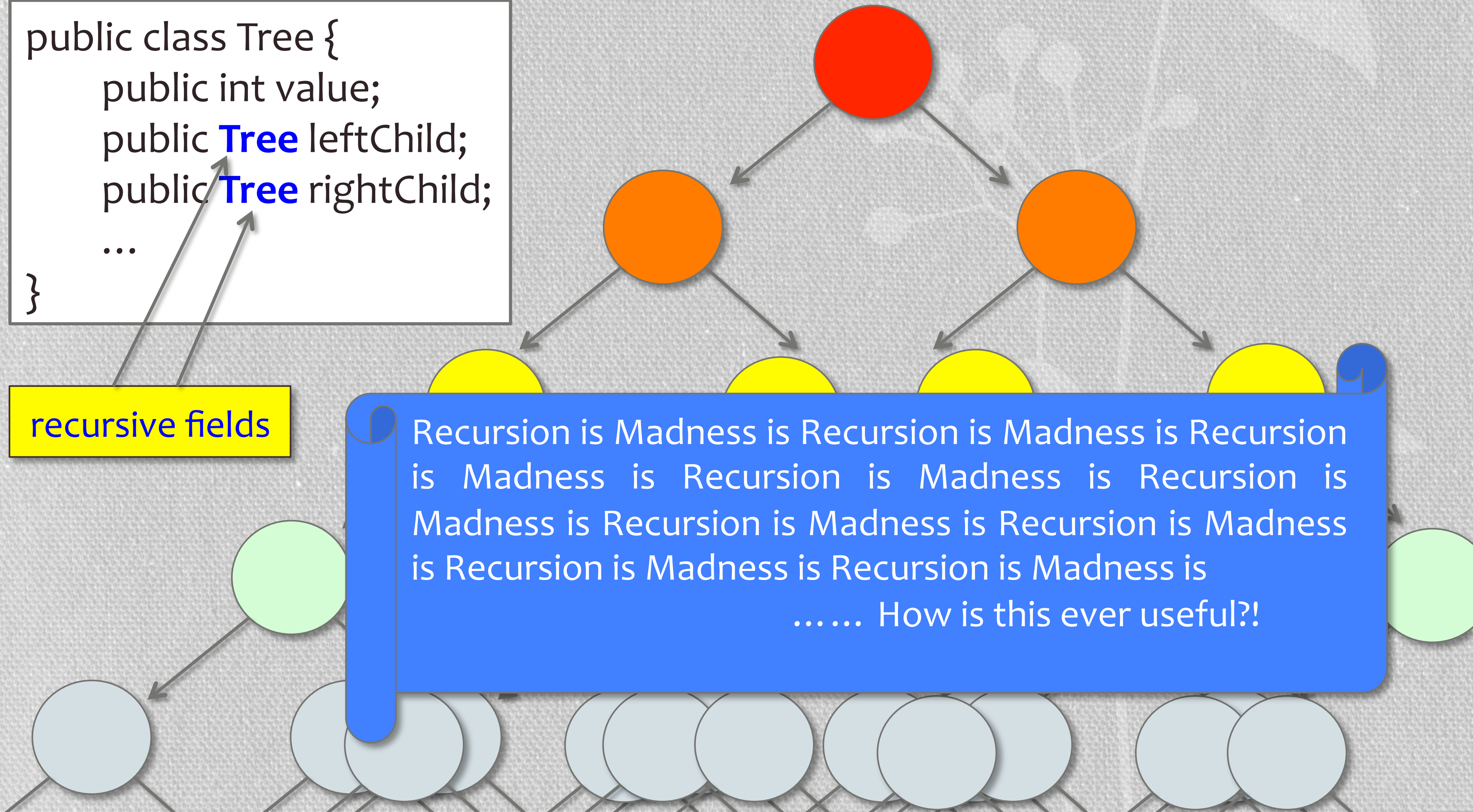
- Is there any problem that recursion or iteration can solve that we couldn't solve with the other?

Data Recursion

Data can also be recursive: when a class definition contains a field whose type is the same as the class being defined:

```
public class Tree {
    public int value;
    public Tree leftChild;
    public Tree rightChild;
    …
}
```

recursive fields

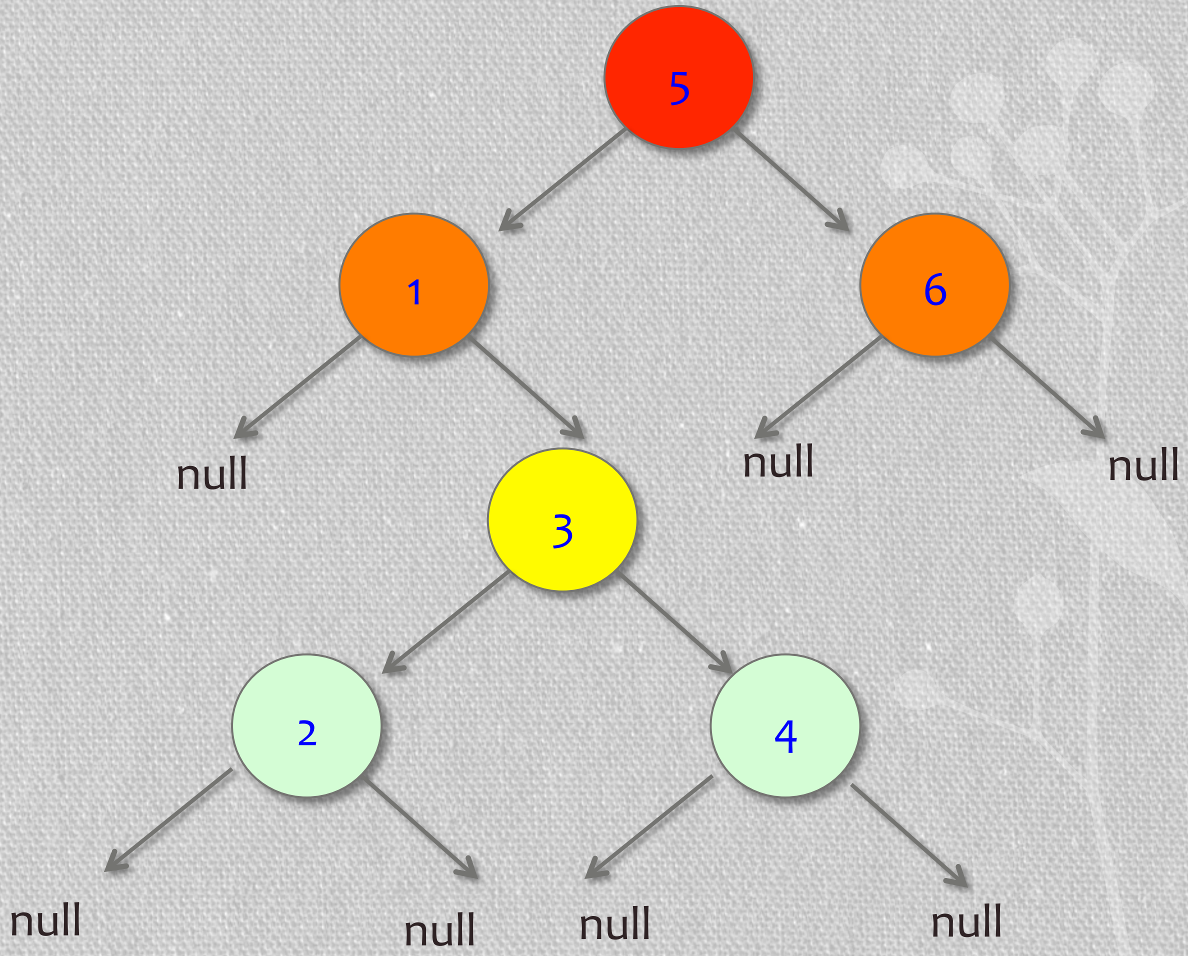


Recursion is Madness is Recursion is Madness is Recursion is Madness is Recursion is Madness is Recursion is Madness is Recursion is Madness is Recursion is Madness is Recursion is Madness is Recursion is Madness is

…… How is this ever useful?!

Base Cases in Data Recursion

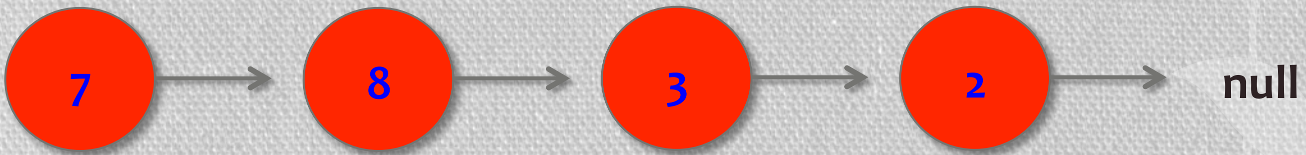
- We will again end the recursion with a base case: the `null` value.



Linked Lists

What if our `Tree` only had one branch? And we named it `IntList`?

```
public class IntList {
    public int val;
    public IntList next;
    …
}
```



It looks a lot like the array **`int[] xs = {7,8,3,2};`**
→ could we implement the usual operations over our IntList that are usually available on arrays?

Making a Linked List Useful

- We might want to add these sorts of operations over our IntList:

1. **size** (how many elements are in here?)
2. **add a value** (sorted? always at the end?)
3. **remove a value** (just tell us which value to remove)
4. **check if a value is present** (return a boolean)
5. **makeArrayVersion** (create an old-fashioned array out of it)

- We can approach these tasks thinking of the diagrams of "before" the operation, "after" the operation, and then write code that implements these changes. Now we're *really* programming with our own data structures!

More Data Structures

There are many common data structures. **CS 310** is a course entirely devoted to them! It. Is. Awesome.

Some others:

- Linear stuff → Lists, Stacks, Queues, …
- Trees → binary, balanced, …
- Graphs → networks, DiGraphs, …
- Hashes → hashes

Java Libraries for Data Structures

- We should take a look at the ArrayList class in Java:

  go to: *(google "Java ArrayList")*

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/ArrayList.html