CS 375 – Analysis of Algorithms

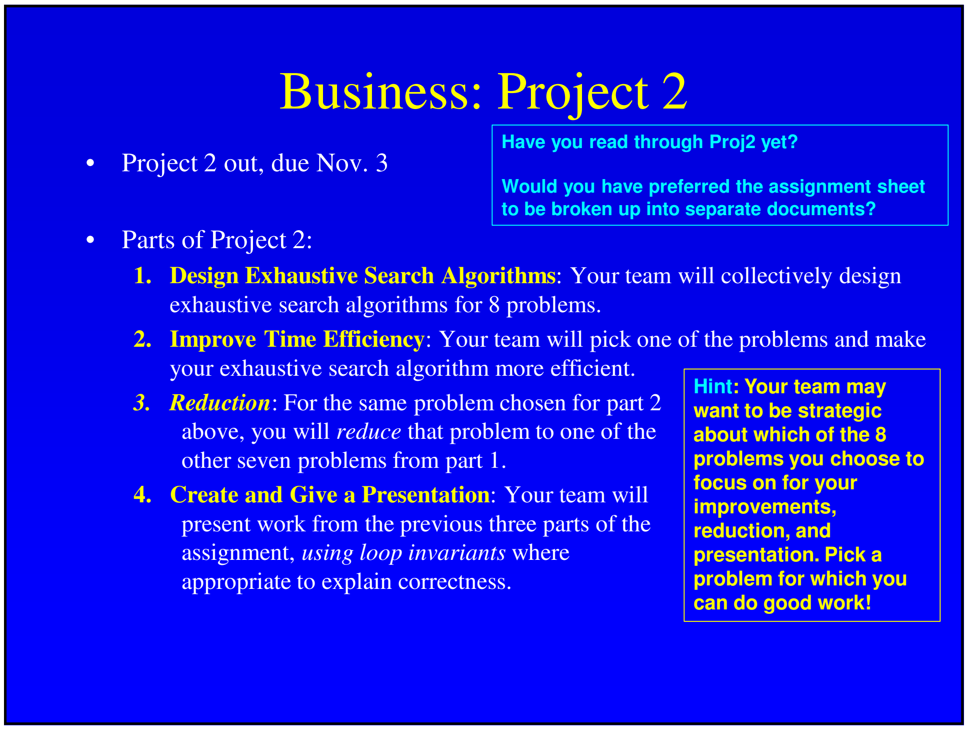
Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Grading update:
  - SA3 returned
- PS2 extended / unusual deadline
  - Due 11:59pm today
- Project 2
  - First part due already
  - Other parts due Nov. 3
  - Please schedule dress rehearsals with me for Friday / Saturday
  - *I will not be available for them next Tuesday or later*



Business: Project 2

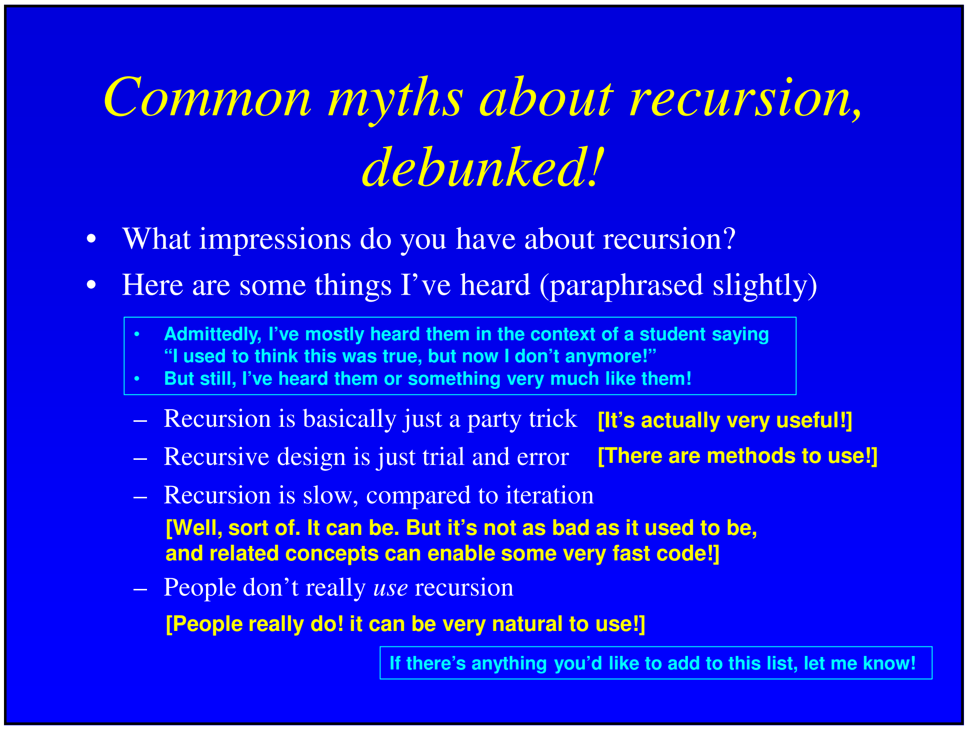
- Project 2 out, due Nov. 3

Have you read through Proj2 yet?

Would you have preferred the assignment sheet to be broken up into separate documents?

- Parts of Project 2:
  1. **Design Exhaustive Search Algorithms**: Your team will collectively design exhaustive search algorithms for 8 problems.
  2. **Improve Time Efficiency**: Your team will pick one of the problems and make your exhaustive search algorithm more efficient.
  3. ***Reduction***: For the same problem chosen for part 2 above, you will *reduce* that problem to one of the other seven problems from part 1.
  4. **Create and Give a Presentation**: Your team will present work from the previous three parts of the assignment, *using loop invariants* where appropriate to explain correctness.

Hint: Your team may want to be strategic about which of the 8 problems you choose to focus on for your improvements, reduction, and presentation. Pick a problem for which you can do good work!



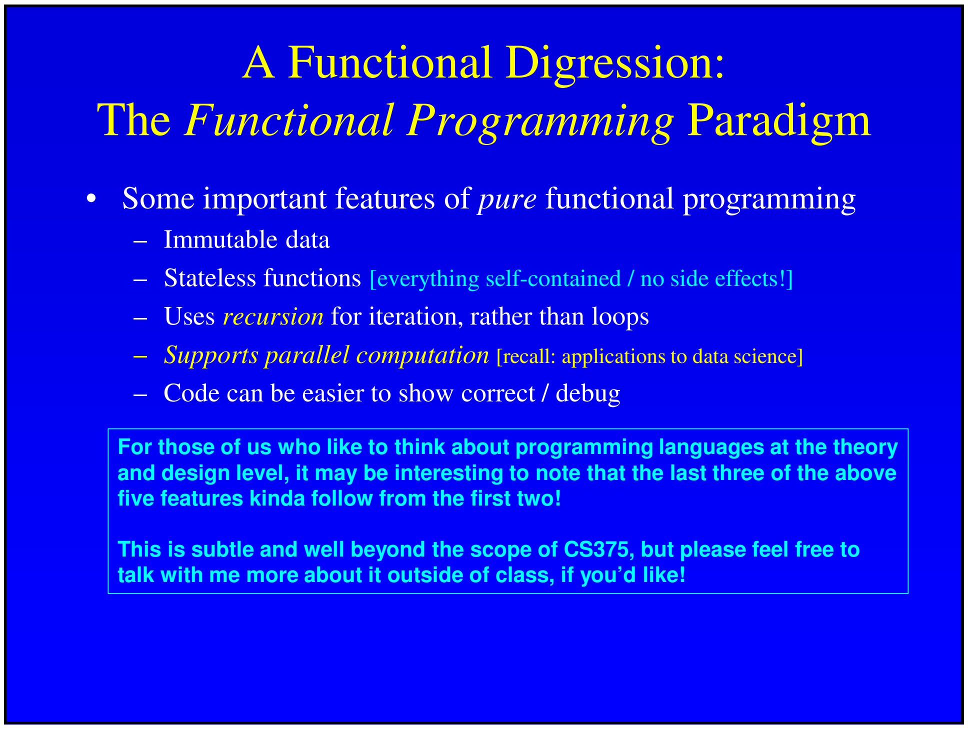
Common myths about recursion, debunked!

- What impressions do you have about recursion?
- Here are some things I've heard (paraphrased slightly)

  - **Admittedly, I've mostly heard them in the context of a student saying "I used to think this was true, but now I don't anymore!"**
  - **But still, I've heard them or something very much like them!**

  – Recursion is basically just a party trick **[It's actually very useful!]**
  – Recursive design is just trial and error **[There are methods to use!]**
  – Recursion is slow, compared to iteration
  [Well, sort of. It can be. But it's not as bad as it used to be, and related concepts can enable some very fast code!]
  – People don't really *use* recursion
  [People really do! it can be very natural to use!]

If there's anything you'd like to add to this list, let me know!



A Functional Digression: The *Functional Programming* Paradigm

- Some important features of *pure* functional programming
  – Immutable data
  – Stateless functions [everything self-contained / no side effects!]
  – Uses *recursion* for iteration, rather than loops
  – *Supports parallel computation* [recall: applications to data science]
  – Code can be easier to show correct / debug

For those of us who like to think about programming languages at the theory and design level, it may be interesting to note that the last three of the above five features kinda follow from the first two!

This is subtle and well beyond the scope of CS375, but please feel free to talk with me more about it outside of class, if you'd like!

Zen and The Art Of Algorithm Design

- A couple of Generally Good Ideas (principles) to help you design your algorithms (and their implementations)

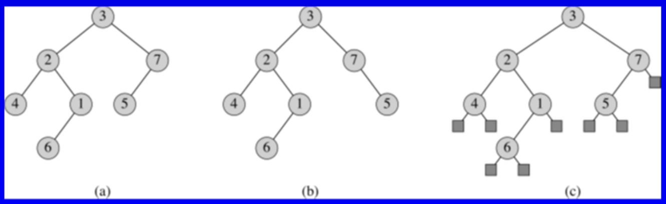
1. **The foundations—i.e., relevant definitions and data structures—should be as simple as possible while still providing all needed functionality**
2. **Let the foundations guide the development and analysis of algorithms based on them**

   – I might restate principle 1 as "*Keep your foundations simple*"
   – I might restate principle 2 as "*Let your definitions tell you what to do*"

- Let's apply this to binary trees…

Binary Trees: A Review

- Remember binary trees from CS231?

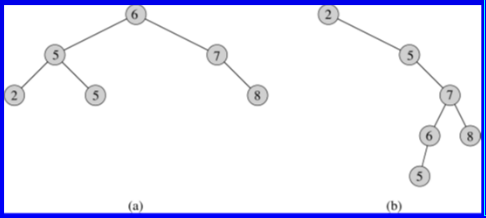


This figure from CLRS illustrates that binary trees could, in principle, contain the same data in different tree structures (parts a and b).

Part c shows *external* nodes, the *null* fields in a typical implementation of a binary tree. They can be thought of as empty sub-trees, in context. External nodes are hidden in parts a and b.

Binary Trees: A Review

- Remember binary trees from CS231?
  - Application: Binary search trees (BST's)



Just as a reminder of where you've seen binary trees before: Both of these are examples of a binary search tree—a specific kind of binary tree.

In our work over the several slides, we'll talk not about BST's, but about binary trees in general

- What's a good definition of a binary tree?

Note: This isn't asking about the definition of a BST, but about the more general data structure of a binary tree

One Possible Definition of Binary Tree

- Often, an implementation of a binary tree is based on two classes: a Node class, and a Tree class (as well as a type T of data to store)
- Node<T> class has fields:
  - *item*: T – the data stored at the node; a value of type T
  - *left*: Node – the left sub-tree, represented by its root node
  - *right*: Node – the right sub-tree, represented by its root node
  - … and perhaps others …
- Tree<T> class has fields:
  - *root*: Node<T> – the root node, which represents the tree
  - … and perhaps others, such as …
  - *size*: int – the number of nodes in the tree

Is this a *good* definition? Consider Principle 1: Keep your foundations simple….

If we wanted a data structure just to be a binary tree of integers, for example, would we need all of this structure?

Definition of our *IntBinTree* Data Structure

NOTE: This definition may show up on HW, too!

- Throughout CS375, we will sometimes refer to an *IntBinTree* data structure, representing a binary tree of integers
- In English, we'd say an IntBinTree is:
  - Either empty,
  - Or
    - an int, called *val*
    - and two *subtrees,* called *left* and *right*, that are also IntBinTrees

Is this a *good* definition? Consider Principle 1: Keep your foundations simple….

- Programmers might be used to seeing it more like this

```
Definition IntBinTree: Empty, or…
   int val # the int value; not empty
   intBinTree left # the left subtree
   intBinTree right # the right subtree
```

Is this definition equivalent to the English one above?

The fact that a tree could be empty is often implicit in many specifications

Definition of our *IntBinTree* Data Structure

NOTE: This definition may show up on HW, too!

- In English, we'd say an IntBinTree is:
  - Either empty,
  - Or
    - an int, called *val*
    - and two *subtrees,* called *left* and *right*, that are also IntBinTrees

We'll call them *IBT*s, for short

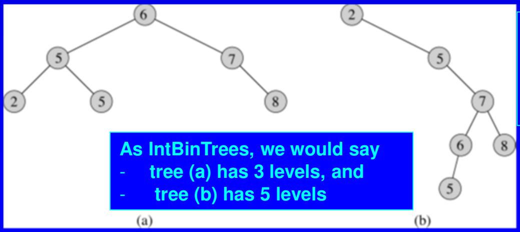
- To be unambiguous (*and consistent with functional programming*) about how we work with IBTs, these will be the primitive functions defined on IBTs:
  - val(*T*): returns the *val* element of an IBT *T*
  - left(*T*): returns the *left* subtree of an IBT *T*
  - right(*T*): returns the *right* subtree of an IBT *T*

For CS375, these are the only ways to access *val, left, right*. So, something like *T.val* is not permitted.

Important note: *val(T), left(T),* and *right(T)* are *functions* that return values; they are not fields of an object. Because of this, we cannot assign values to them—e.g., val(T) = 3 is not permitted.

IntBinTrees: An Exercise

- Design an algorithm to return the number of *levels* in an IntBinTree



As IntBinTrees, we would say
- tree (a) has 3 levels, and
- tree (b) has 5 levels

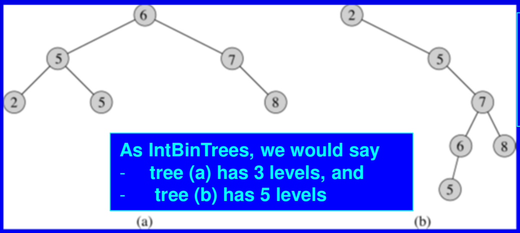
Definition IntBinTree: Empty, or
int val # int value
intBinTree left # left subtree
intBinTree right # right subtree

- What design paradigm will we use for this algorithm?
  – Will it be iterative or recursive?

Consider principle 2, “Let your definitions tell you what to do.”
How does our definition of an IntBinTree tell us what to do here?

IntBinTrees: An Exercise

- Design an algorithm to return the number of *levels* in an IntBinTree



As IntBinTrees, we would say
- tree (a) has 3 levels, and
- tree (b) has 5 levels

Definition IntBinTree: Empty, or
int val # int value
intBinTree left # left subtree
intBinTree right # right subtree

- *Because* the definition of IntBinTree is recursive, it makes sense that algorithms over it would be recursive

In fact, because an IntBinTree is defined recursively in terms of two IntBinTrees, it might make sense for an algo over IntBinTree to have two recursive calls! (That’s an example of Principle 2—letting definitions tell us what to do.)

A Review of Recursive Design (*Divide and Conquer*)

- Every recursive algorithm has the following components
  - *Base case(s)*: One or more small case(s) for which it is easy to identify (or compute) and return a solution
  - *Recursive case(s)*: One or more cases in which the algorithm calls itself on a smaller instance of its input
    - *Divide*: The algorithm must *break the original problem* (input) *down into smaller sub-problems* (sub-inputs) on which the algo can be called recursively
    - *"Conquer"*: The algorithm must solve each of the sub-problems
    - *Combine*: The algorithm must combine / employ the solutions of sub-problems into a solution of the original problem
  - **Recursive cases bring input closer to *terminating* in a base case**

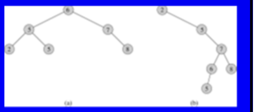
It's *really* important that recursions terminate

IntBinTrees: An Exercise

- Design an algorithm to return the number of *levels* in an IntBinTree
  - Reminder: In English, an IntBinTree
    is either empty or
    an int and two subtrees

Definition IntBinTree: Empty, or…
int val # int value
intBinTree left # left subtree
intBinTree right # right subtree

- For our recursive algorithm to compute the number of levels…
  - What are the input and output?
  - What input does the base case check for?
  - What's the intended output for the base case?
  - How many recursive calls will the algorithm make?
  - How do we use output from
    recursive call(s) to compute
    the output on the given input?



It can be helpful to write down a problem specification with input and output types….

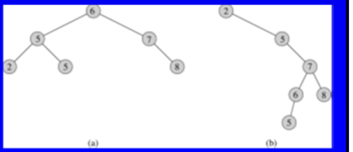
IntBinTrees: An Exercise

- Design an algorithm to return the number of *levels* in an IntBinTree
  - Reminder: In English, an IntBinTree is either empty or an int and two subtrees

```
Definition IntBinTree: Empty, or…
  int val # int value
  intBinTree left # left subtree
  intBinTree right # right subtree
```

- Our recursive algorithm to compute the number of levels:

```
Algorithm: Levels(T)
//Input: IntBinTree T
//Output: integer, number of levels in T
  if T is empty
    return 0
  else
    return max{Levels(left(T)), Levels(right(T))} + 1
```



- How would we explain this algorithm's correctness?

Correctness of Recursive Algorithms: Inductive Arguments

- When arguing the correctness of a recursive algorithm, the general form is that of an *inductive* argument
- The explanation follows the structure of the algorithm
  - Show that the algorithm's base case returns correct output
  - Show that the recursive cases return correct output… *under the assumption that all recursive calls return correct output*

Just like when creating recursive code—we assume recursive calls work in the recursive case!

- Here, how would that work?
- How would we explain the base case? The recursive case?

```
Algorithm: Levels(T)
//Input: IntBinTree T
//Output: integer, number of levels in T
  if T is empty
    return 0
  else
    return max{Levels(left(T)), Levels(right(T))} + 1
```

Correctness of Recursive Algorithms: Inductive Arguments

- When arguing the correctness of a recursive algorithm, the general form is that of an *inductive* argument
- The explanation follows the structure of the algorithm
  - Show that the algorithm's base case returns correct output
  - Show that the recursive cases return correct output…
    under the assumption that all recursive calls return correct output

Just like when creating recursive code—we assume recursive calls work in the recursive case!

- As part of explaining recursive case(s), also explain how we know the algo *terminates*
  - Show arguments in recursive calls get closer to base case

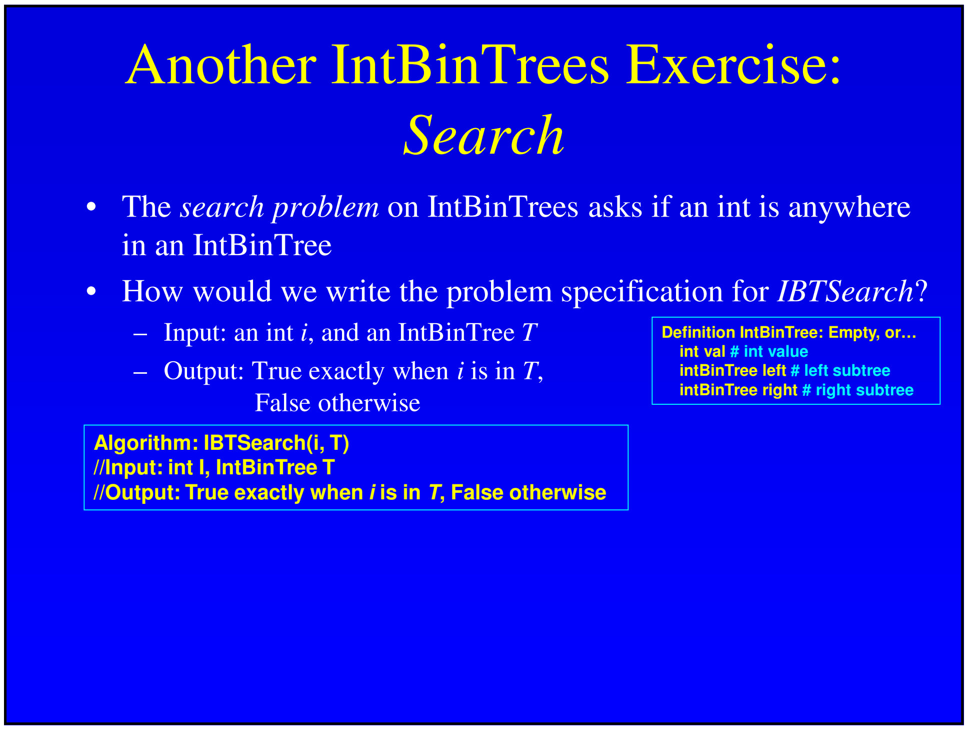
```
Algorithm: Levels(T)
//Input: IntBinTree T
//Output: integer, number of levels in T
  if T is empty
    return 0
  else
    return max{Levels(left(T)), Levels(right(T))} + 1
```

Another IntBinTrees Exercise: *Search*

- The *search problem* on IntBinTrees asks if an int is anywhere in an IntBinTree
- How would we write the problem specification for *IBTSearch*?
  - What would be the input?
  - What would be correct output?

```
Definition IntBinTree: Empty, or…
    int val # int value
    intBinTree left # left subtree
    intBinTree right # right subtree
```

- How would we design an algorithm to solve it?
  - Would the algorithm be iterative or recursive?
- How would we argue its correctness?

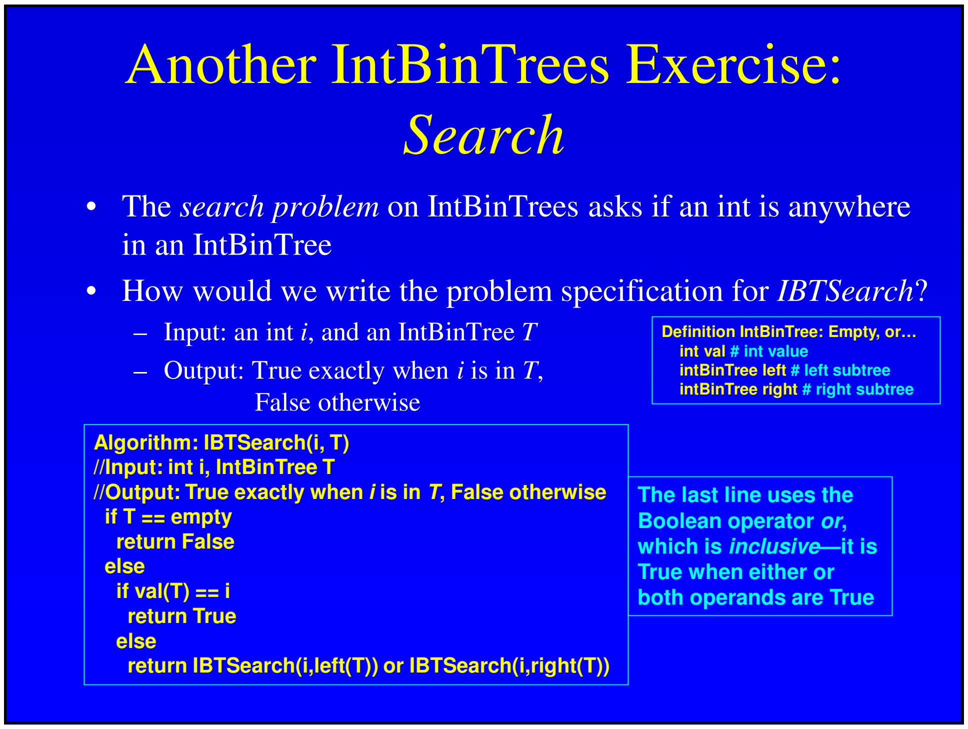


Another IntBinTrees Exercise: *Search*

- The *search problem* on IntBinTrees asks if an int is anywhere in an IntBinTree
- How would we write the problem specification for *IBTSearch*?
  - Input: an int *i*, and an IntBinTree *T*
  - Output: True exactly when *i* is in *T*, False otherwise

```
Definition IntBinTree: Empty, or…
   int val # int value
   intBinTree left # left subtree
   intBinTree right # right subtree
```

```
Algorithm: IBTSearch(i, T)
//Input: int I, IntBinTree T
//Output: True exactly when i is in T, False otherwise
```



Another IntBinTrees Exercise: *Search*

- The *search problem* on IntBinTrees asks if an int is anywhere in an IntBinTree
- How would we write the problem specification for *IBTSearch*?
  - Input: an int *i*, and an IntBinTree *T*
  - Output: True exactly when *i* is in *T*, False otherwise

```
Definition IntBinTree: Empty, or…
   int val # int value
   intBinTree left # left subtree
   intBinTree right # right subtree
```

```
Algorithm: IBTSearch(i, T)
//Input: int i, IntBinTree T
//Output: True exactly when i is in T, False otherwise
 if T == empty
   return False
 else
   if val(T) == i
     return True
   else
     return IBTSearch(i,left(T)) or IBTSearch(i,right(T))
```

The last line uses the Boolean operator *or*, which is *inclusive*—it is True when either or both operands are True