EECS 280 – Lecture 19

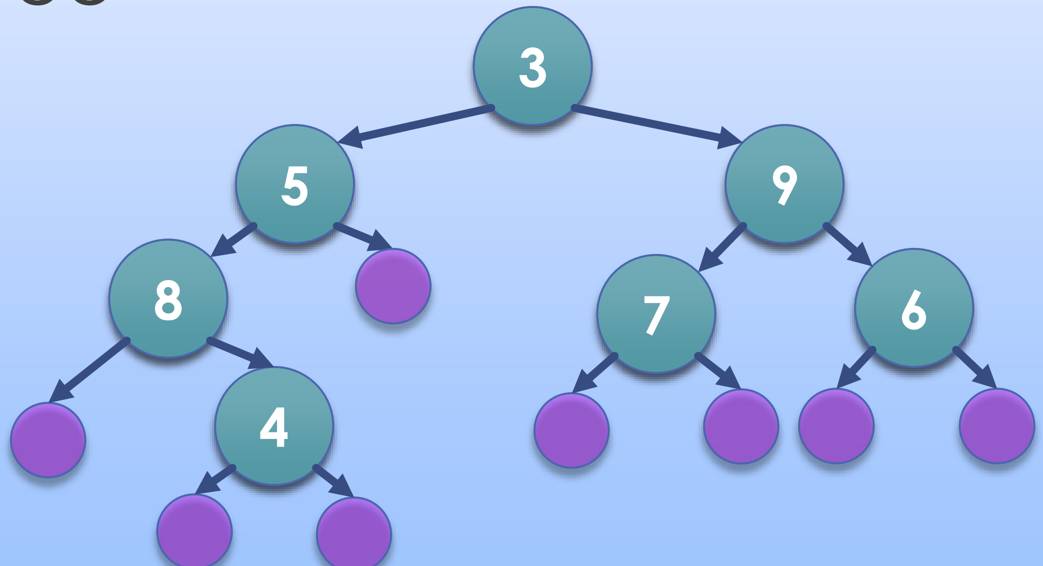
Structural Recursion

3/23/2022

Recursive Data Structures

- List

- Tree



Recall: Linked List Data Representation

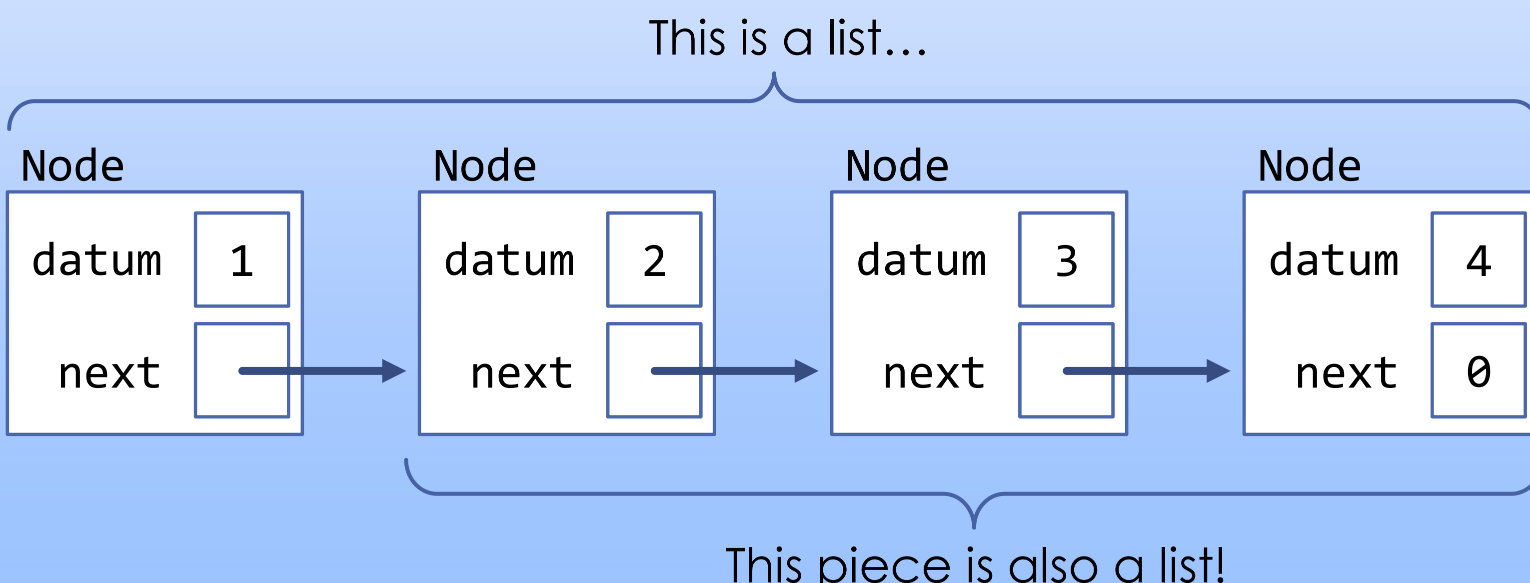
```
struct Node {
  int datum;
  Node *next;
};
```

Assume int elements for now.

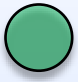
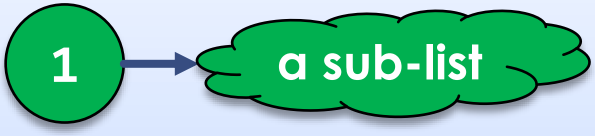
Used to store an element of the list.

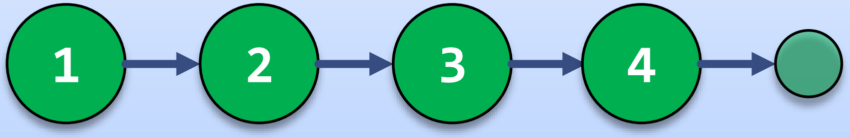
Contains the address of the next node in the list.

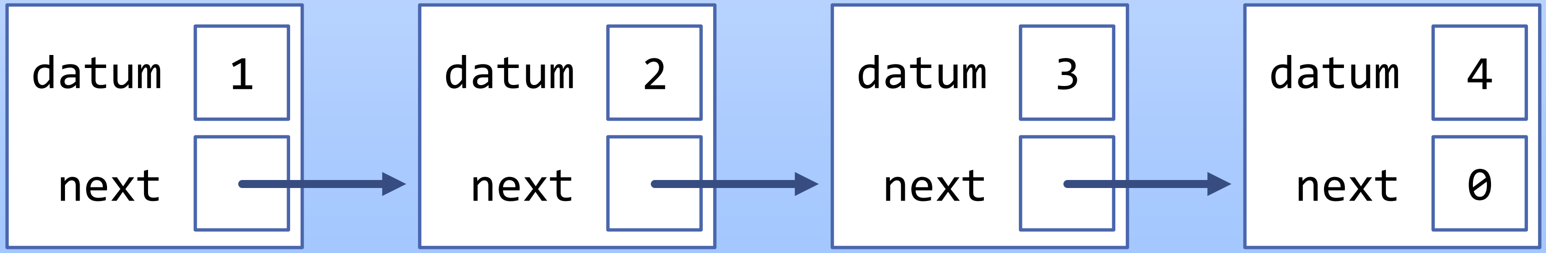
- The node structure of a linked list is self-similar!



Recursively Defining a List

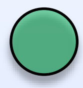
- Conceptually, any list is either:
  1. empty 
  2. A datum, followed by a sub-list 

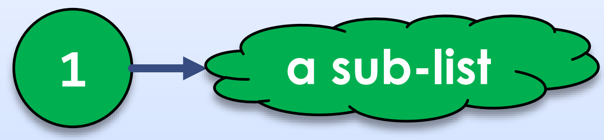




Processing a List Recursively

- For example, let's **compute the length of a list L.**
- Consider the two cases:

1. empty  easy enough: `length(L) = 0`

2. A datum, followed by a sub-list 


```
length(L) = 1 + length( the sub-list )
```

Recursion!

~~length(1 → 2 → 3 → 4 →)~~

= 1 + ~~length(2 → 3 → 4 →)~~

1 + ~~length(3 → 4 →)~~

1 + ~~length(4 →)~~

1 + ~~length( )~~

0

Processing a List Recursively

- For example, let's compute the length of a list L.
- Consider the two cases:

1. empty easy enough: `length(L) = 0`
2. A datum, followed by a sub-list

`length(L) = 1 + length( the sub-list )`

```
int length(Node *node) {
  if (node == nullptr) { // BASE CASE
    return 0;
  }
  else { // RECURSIVE CASE
    return 1 + length(node->next);
  }
}
```

This could also be written as !node.

Exercise: List Recurrence Relations

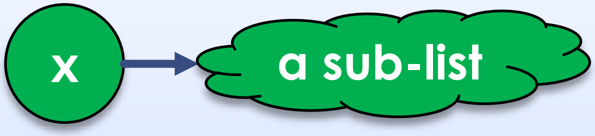
- See Exercise 19.1 in the accompanying worksheet:

bit.ly/3fQE21a

Exercise: max

```
struct Node {
  int datum;
  Node *next;
};
```

- Now, write a function to find the maximum element.
- Hint: You may need to use a different base case.
- Hint: Use the recursive leap of faith on the sub-list.



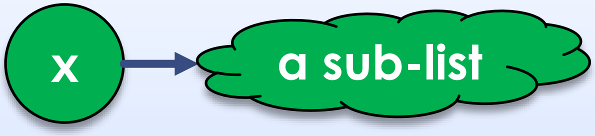

```
// Use this helper function if you want
int max(int x, int y) {
  if (x > y) { return x; }
  else { return y; }
}
```

```
// REQUIRES: 'node' must not be null (i.e. the list
//           starting at 'node' may not be empty)
// EFFECTS:  Returns the largest element in the list.
int max(Node *node) {
  // YOUR CODE HERE
}
```

Solution: max

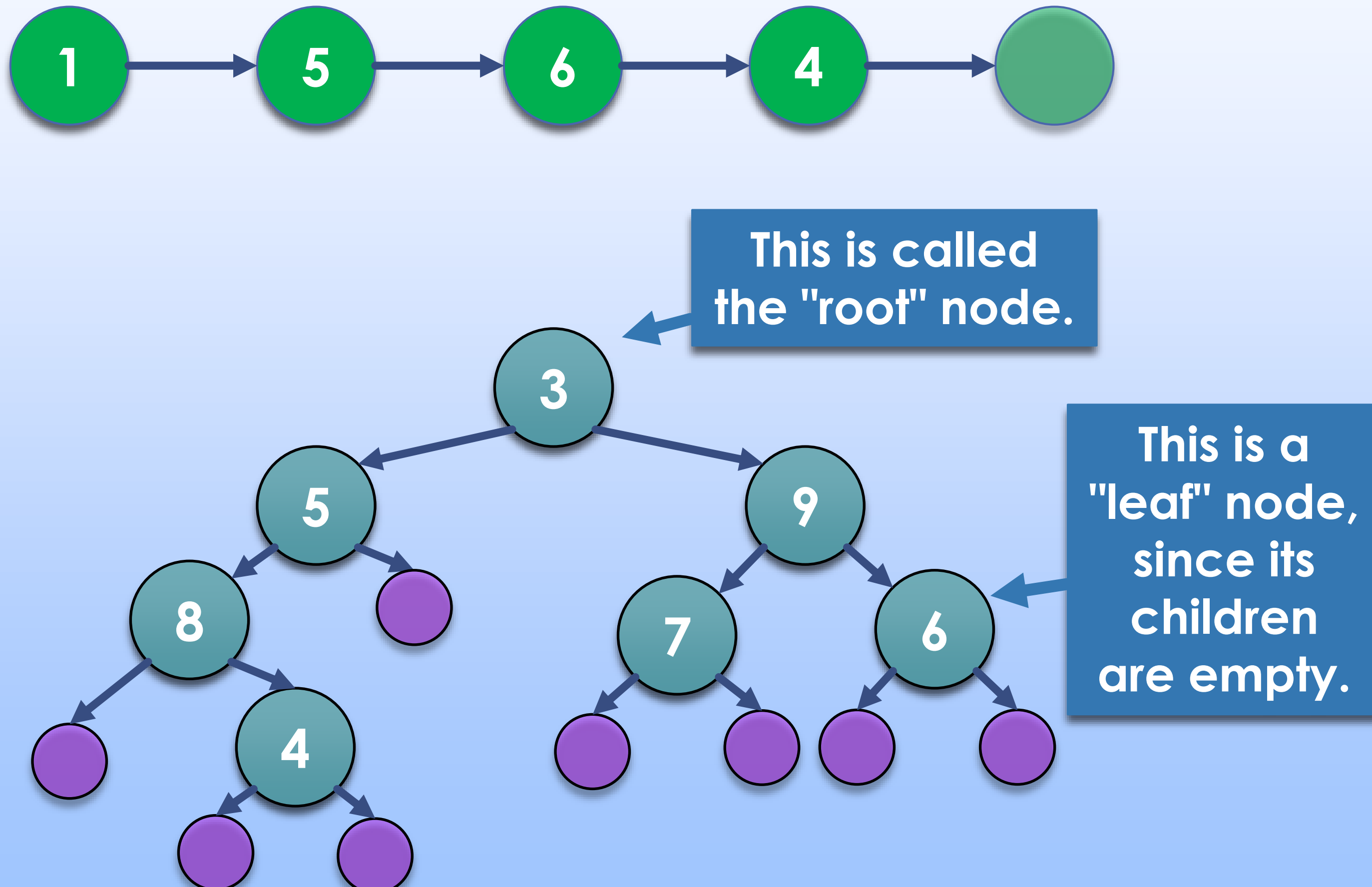
```
struct Node {
  int datum;
  Node *next;
};
```

- Now, write a function to find the maximum element.
- Hint: You may need to use a different base case.
- Hint: Use the recursive leap of faith on the sub-list.



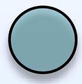

```
// REQUIRES: 'node' must not be null (i.e. the list
//           starting at 'node' may not be empty)
// EFFECTS:  Returns the largest element in the list.
int max(Node *node) {
  // BASE CASE – A single element list
  if (node->next == nullptr) {
    return node->datum;
  }
  else { // RECURSIVE CASE
    return std::max(node->datum, max(node->next));
  }
}
```

Recursive Data Structures

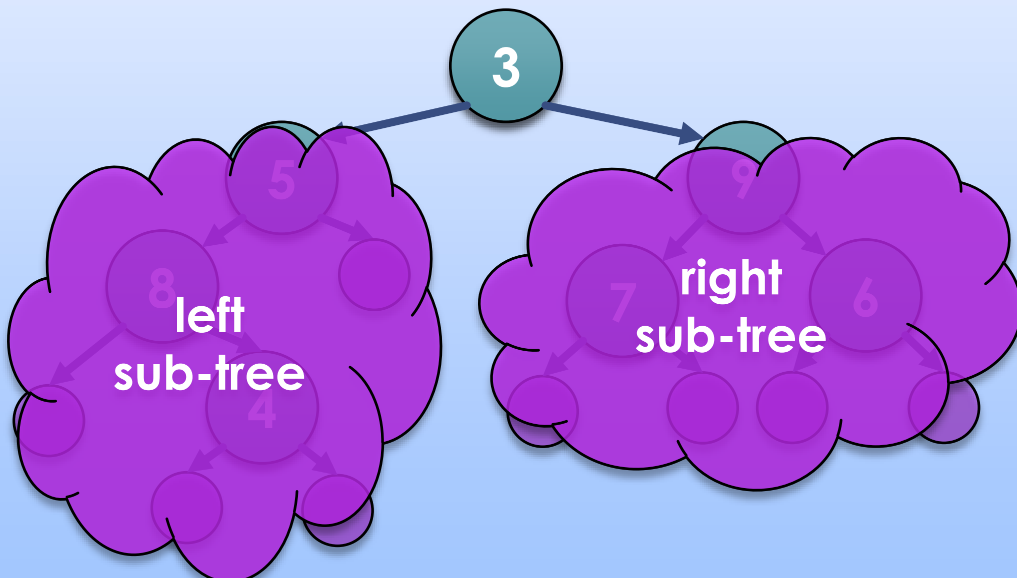


Recursively Defining a Tree

- Conceptually, a tree is either:

1. empty 

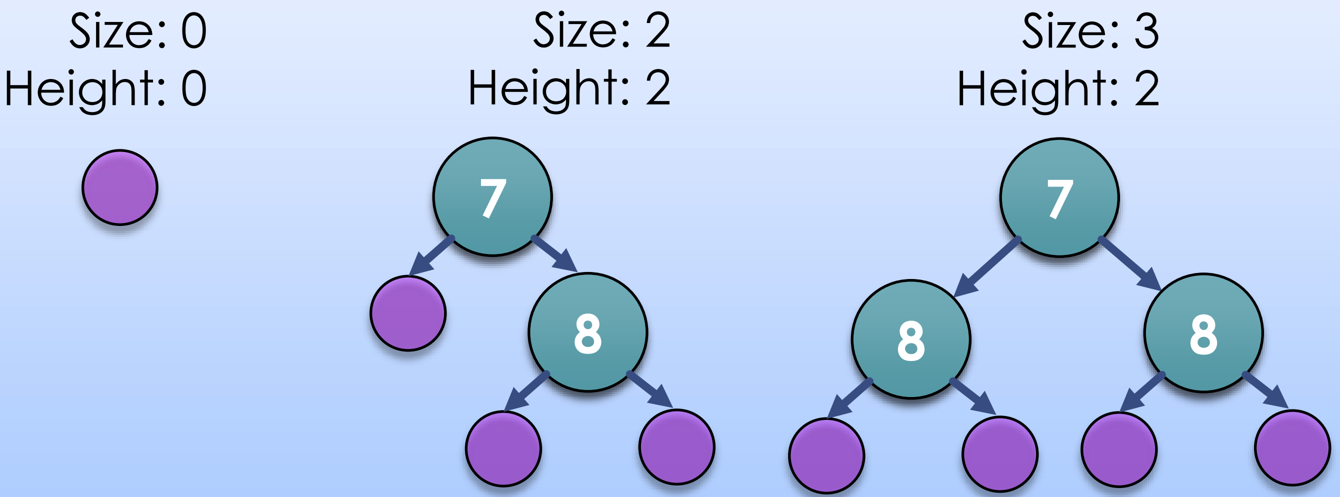
2. A datum, with left and right sub-trees



We will be working on *binary trees*, in which each node has two children.

Properties of Trees

- We can measure a tree in two ways:
  - Size: The total number of elements
  - Height[1]: The longest chain of nodes from root to leaf.

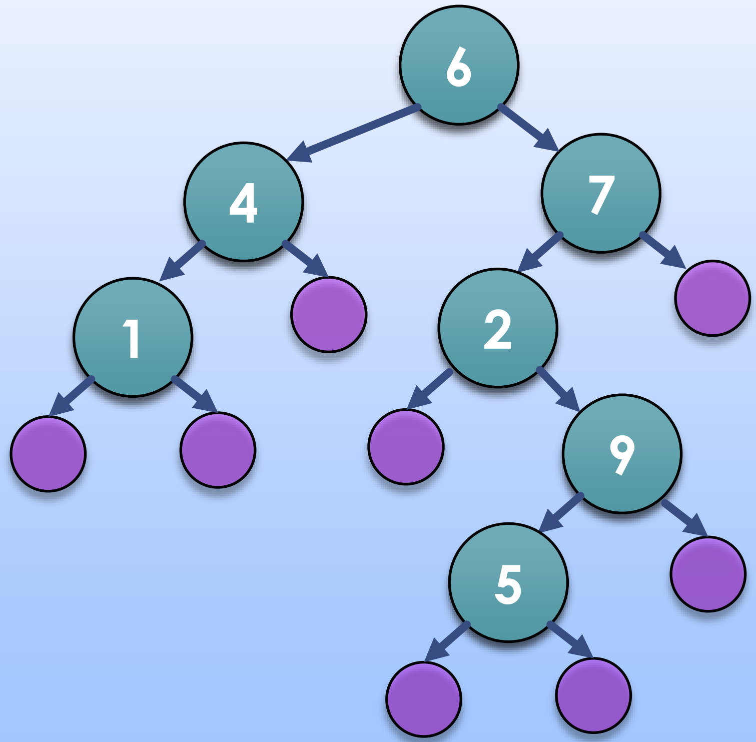


1 This is sometimes called the "maximum depth" of the tree.

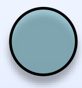
Properties of Trees

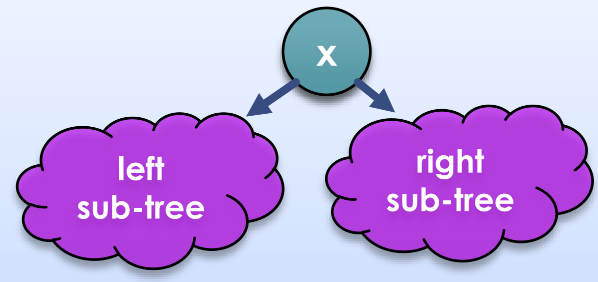
- What are the size and height of this tree?

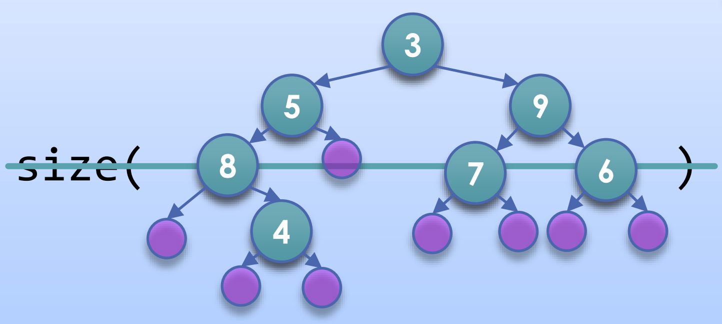
Size: 7 Height: 5



Processing a Tree Recursively

- For example, let's **compute the size of a tree T**:
- Consider the two cases:
  1. empty 
  2. A datum, with left and right sub-trees

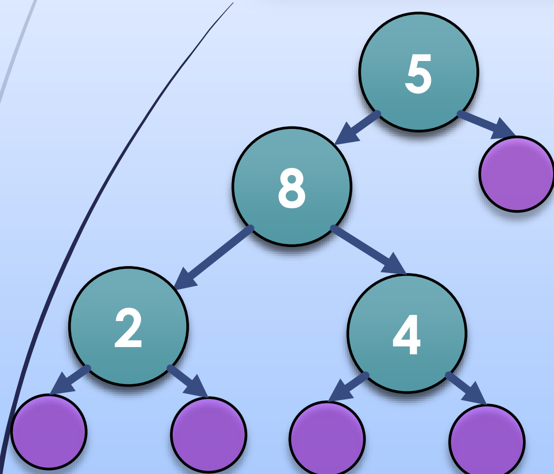


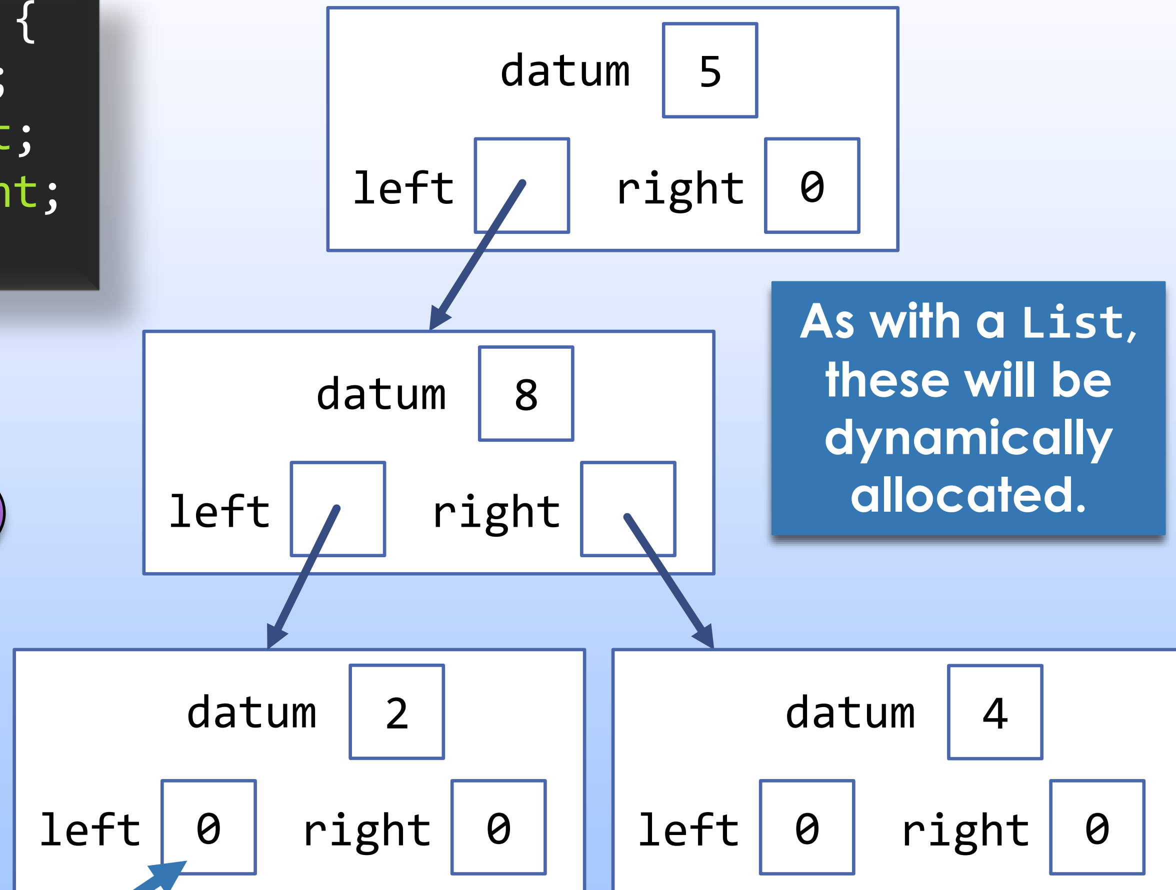


Need to count the root node!

Tree Data Representation

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```



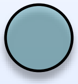


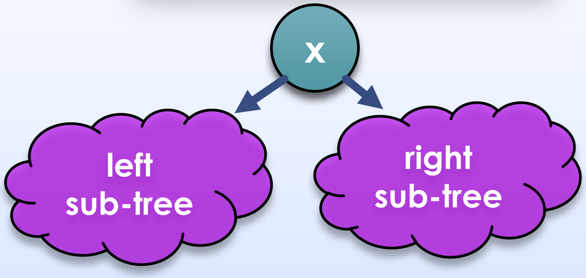
As with a List, these will be dynamically allocated.

Empty trees are represented by a null pointer.

Example: Tree size

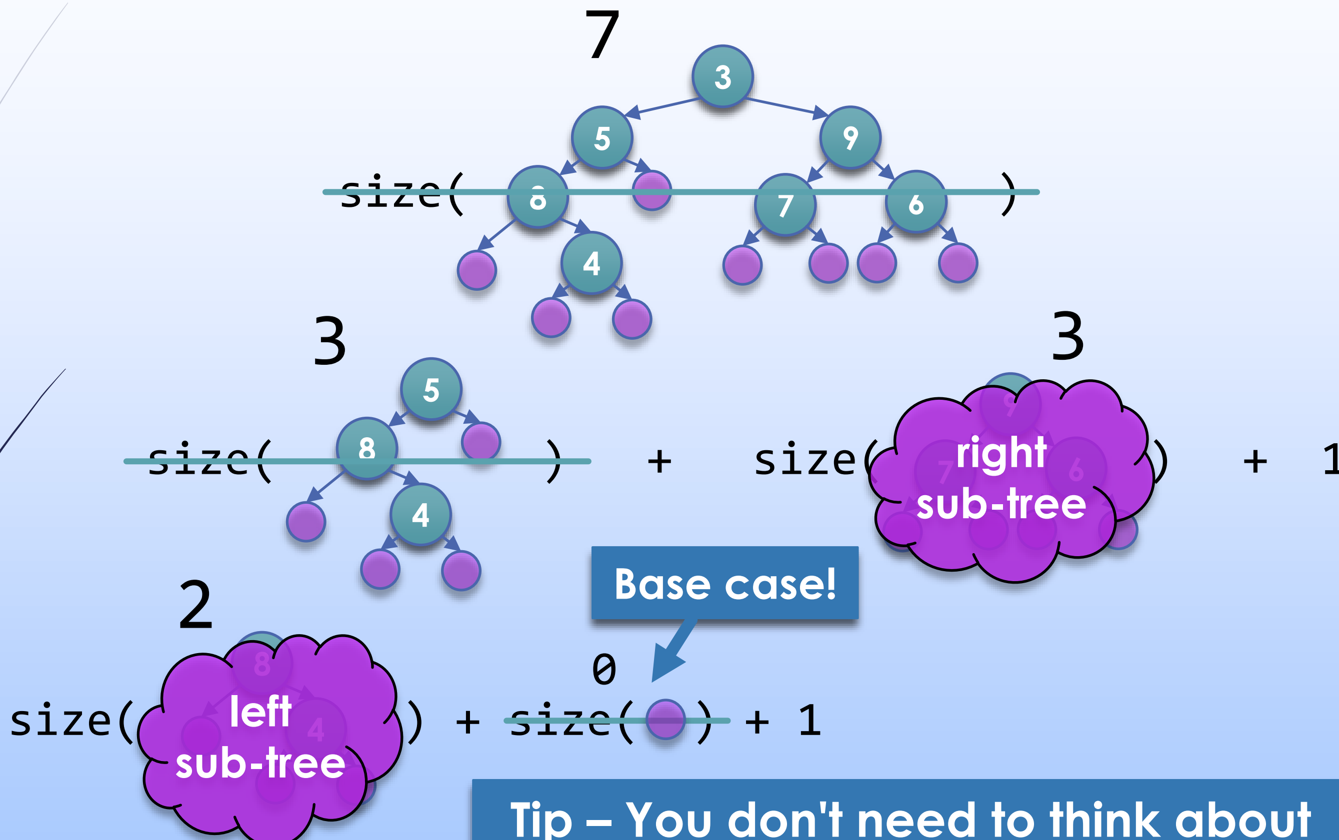
```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- Let's **compute the size of a tree T**:
- Consider the two cases:
  1. empty 
  2. A datum, with left and right sub-trees




```
// EFFECTS: Returns the size of the tree rooted at 'node'.
int size(Node *node) {
  // BASE CASE - Empty tree has size 0
  if (!node) {
    return 0;
  }

  // RECURSIVE CASE
  return 1 + size(node->left) + size(node->right);
}
```

Computing Tree Size



Tip – You don't need to think about the recursion "all the way down". Just take the recursive leap of faith!

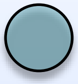
Exercise: Tree Recurrence Relations

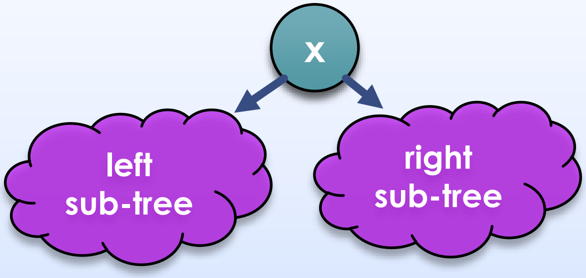
- See Exercise 19.2 in the accompanying worksheet:

bit.ly/3fQE21a

Exercise: Tree height

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

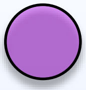
- Write a function to compute the **height** of a tree.
- Consider the two cases:
  1. empty 
  2. A datum, with left and right sub-trees




```
// EFFECTS: Returns the height of the tree rooted at 'node'.
int height(Node *node) {



  // YOUR CODE HERE




}
```

Tree Height

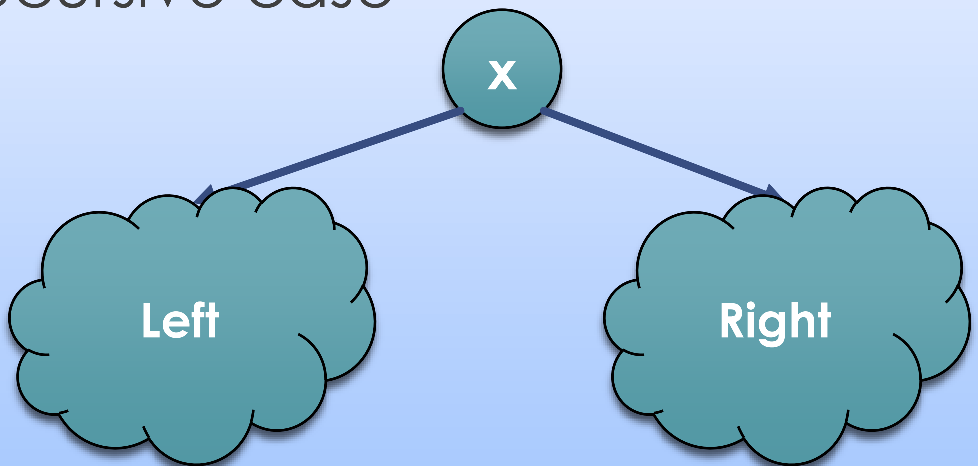
- Base case



```
if (!node) {
  return 0;
}
```

- Recursive case

```
height = 1 + max(L, R)
```

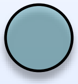


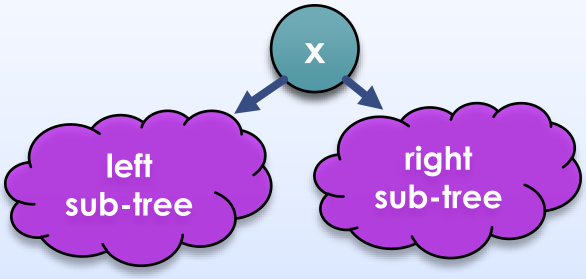

```
L = height(node->left)
```

```
R = height(node->right)
```

Solution: Tree height

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- Write a function to compute the **height** of a tree.
- Consider the two cases:
  1. empty 
  2. A datum, with left and right sub-trees



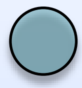

```
// EFFECTS: Returns the height of the tree rooted at 'node'.
int height(Node *node) {
  // BASE CASE - Empty tree has size 0
  if (!node) {
    return 0;
  }

  // RECURSIVE CASE
  return 1 + std::max(height(node->left),
                      height(node->right));
}
```

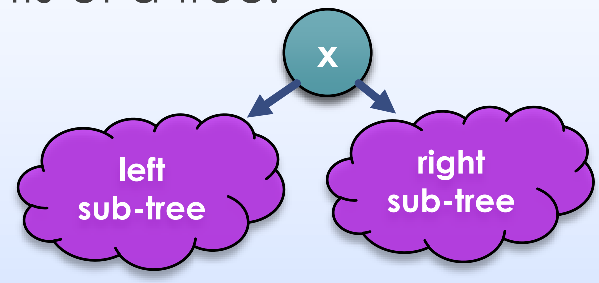
Break Time!

We'll start again in five minutes.

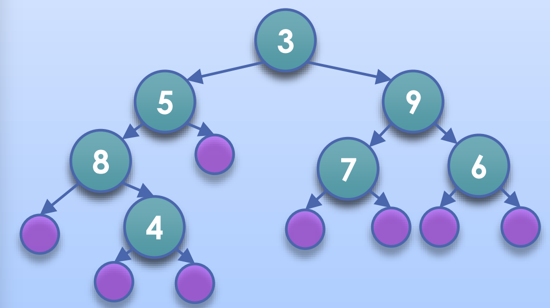
Tree print

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- Write a function to print the elements of a tree.
- Consider the two cases:
  1. empty 
  2. A datum, with left and right sub-trees

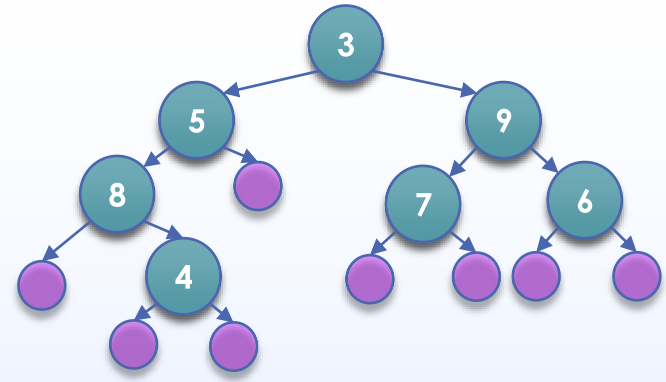



```
// EFFECTS: Prints the elements of
//          the tree rooted at
//          'node', with a space
//          after each element.
void print(Node *node) {
  if (node) { // RECURSIVE CASE
    cout << node->datum << " ";
    print(node->left);
    print(node->right);
  }
}
```



Prints 3 5 8 4 9 7 6

Tree Traversals



- For `print()`, we have a choice of when to process a datum
- A ***preorder traversal*** processes a datum before the recursive calls.

Prints 3 5 8 4 9 7 6

```
if (node) { // RECURSIVE CASE
  cout << node->datum << " ";
  print(node->left);
  print(node->right);
}
```

- An ***inorder traversal*** processes a datum between the calls.

Prints 8 4 5 3 7 9 6

```
if (node) { // RECURSIVE CASE
  print(node->left);
  cout << node->datum << " ";
  print(node->right);
}
```

- A ***postorder traversal*** processes a datum after the recursive calls.

Prints 4 8 5 7 6 9 3

```
if (node) { // RECURSIVE CASE
  print(node->left);
  print(node->right);
  cout << node->datum << " ";
}
```

Tree height

- Question: Can the height function be implemented tail recursively? If so, how? If not, why not?
  - Nope! You need to check both sides of the tree, which requires two recursive calls. They can't both be tail calls.

```
// EFFECTS: Returns the height of the tree rooted at 'node'.
int height(Node *node) {
  // BASE CASE - Empty tree has size 0
  if (!node) {
    return 0;
  }

  // RECURSIVE CASE
  return 1 + std::max(height(node->left),
                      height(node->right));
}
```

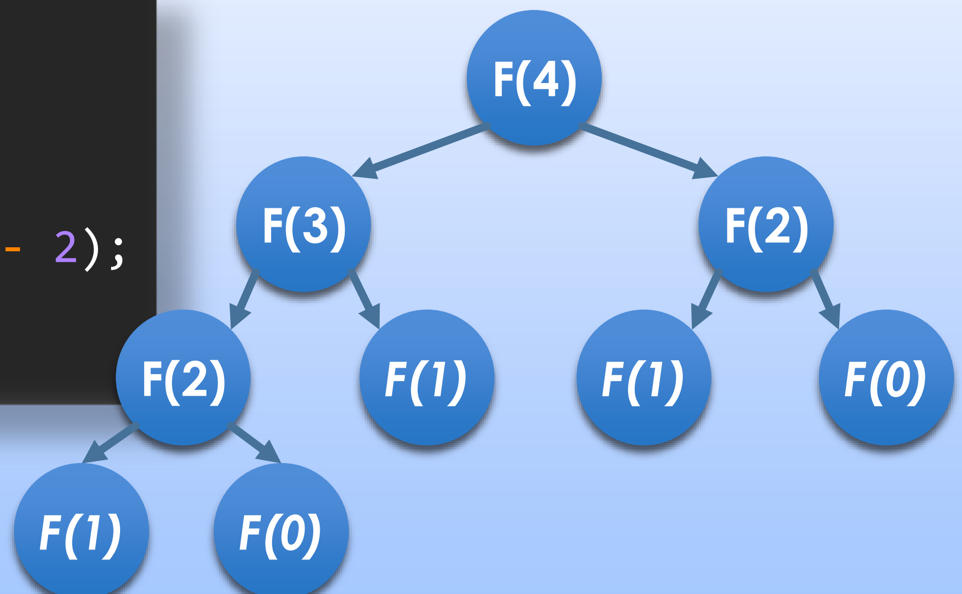
Types of Recursion

- A function is ***linear recursive*** if it makes at most one recursive call each time the function is called.
  - Example: `fact`, `List max`
- A function is ***tail recursive*** if it is linear recursive and all recursive calls are tail calls, so that no work is done after a recursive call.
  - Example: `fact_tail`, `max_tail`
- A function is ***tree recursive*** if it might make more than one recursive call each time the function is called.
  - Example: `Tree size`, `height`
  - A function doesn't have to operate on trees to be tree recursive! It is tree recursive if the structure of the recursive calls *branches* and thus looks like a tree.

Subproblem Graph: Fibonacci

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  return fib(n - 1) + fib(n - 2);
}
```



This `fib` function is tree recursive.

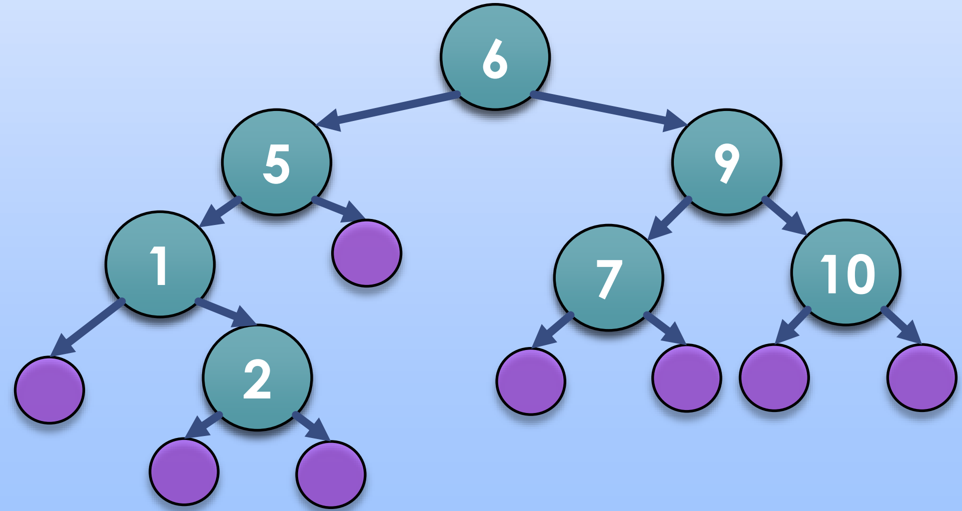
Fibonnaci can also be computed iteratively or tail recursively.

Binary Search Trees (BSTs)

- A tree is a binary search tree if…
  - It is empty

  OR
  - The left and right subtrees are binary search trees.
  - All elements in any **left** subtree are **less** than the root.
  - All elements in any **right** subtree are **greater** than the root.

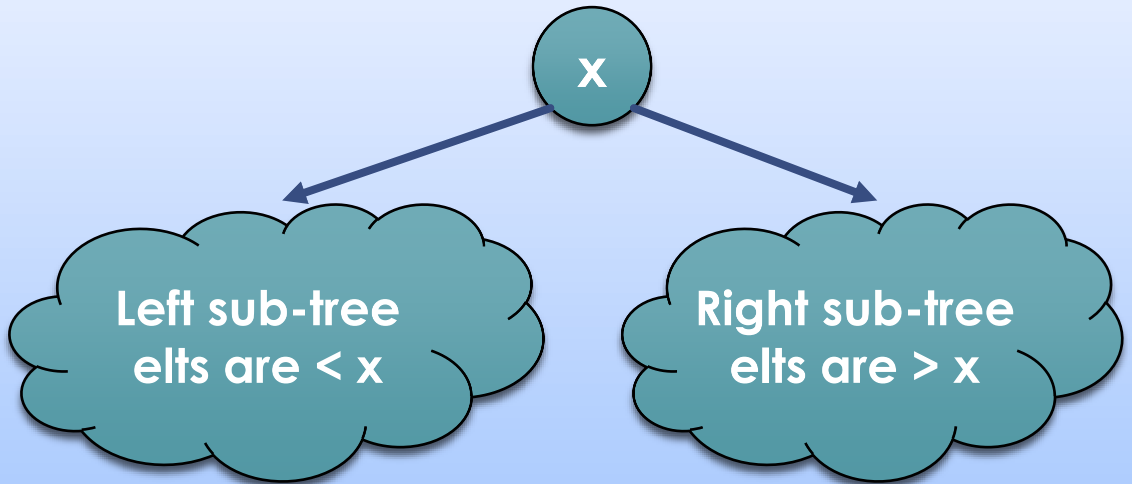
It is so called because searching for elements can be done efficiently.



Note: In the slides and on project 5, we make a simplifying assumption that there are no duplicates in our BSTs.

Searching in a Binary Search Tree

- If we're looking for an element in a BST, comparing with the root tells us where to look.
  - Thus the name "*Binary Search* Tree".



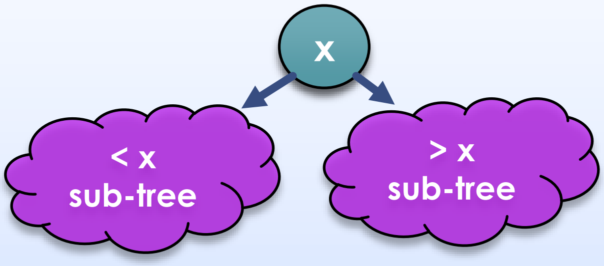
Simplifying assumption: no duplicate elements.

Building a Binary Search Tree

Example: BST `max`

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- The maximum element in a binary search tree is:
  1. The datum in the node if the right sub-tree is empty
  2. Otherwise, the maximum element in the right sub-tree

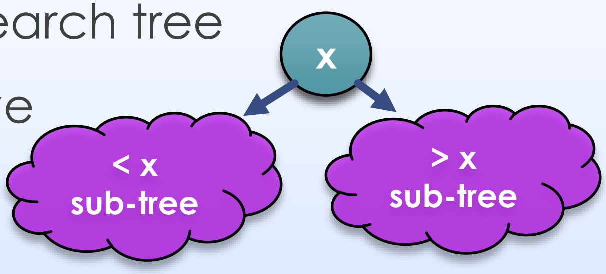



```
// REQUIRES: 'node' must be a binary search tree that
//           is not empty (i.e. 'node' is not null)
// EFFECTS:  Returns the largest element in the tree.
int max(Node *node) {
  // BASE CASE – Right sub-tree is empty
  if (!node->right) {
    return node->datum;
  }
  else { // RECURSIVE CASE - Right sub-tree not empty
    return max(node->right);
  }
}
```

Exercise: BST contains

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- Write a function that determines whether or not an element is contained in a binary search tree
- Your solution should be tail recursive

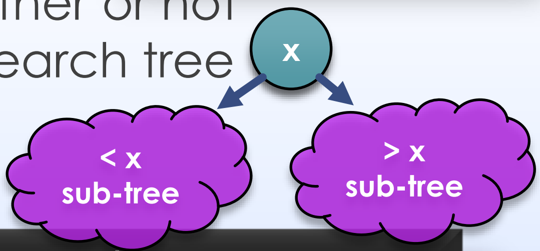



```
// REQUIRES: 'node' must be a binary search tree
// EFFECTS:  Returns whether or not the tree contains
//           the given item.
bool contains(Node *node, int item) {



  // YOUR CODE HERE



}
```

Solution: BST contains

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- Write a function that determines whether or not an element is contained in a binary search tree
- Your solution should be tail recursive




```
// REQUIRES: 'node' must be a binary search tree
// EFFECTS:  Returns whether or not the tree contains
             the given item.
bool contains(Node *node, int item) {
  if (!node) {
    return false;
  } else if (node->datum == item) {
    return true;
  } else if (node->datum > item) {
    return contains(node->left, item);
  } else {
    return contains(node->right, item);
  }
}
```

The BinarySearchTree Interface

```
template <typename T>
class BinarySearchTree {
public:
  BinarySearchTree();
  BinarySearchTree(const BinarySearchTree &other);
  BinarySearchTree & operator=(const BinarySearchTree &other);
  ~BinarySearchTree();

  bool empty() const;
  int size() const;
  bool contains(const T &item) const;
  void insert(const T &item);

private:
  struct Node {
    T datum;
    Node *left, *right;
  };

  Node *root;
};
```

BinarySearchTree Implementation

- We can implement the functions that operate on Nodes as `private`, `static` member functions

```
template <typename T>
class BinarySearchTree {
public:
  bool contains(const T &item) const {
    return contains_impl(root, item);
  }
private:
  Node *root;
  static bool contains_impl(Node *node, const T &item) {
    if (!node) {
      return false;
    } else if (node->datum == item) {
      return true;
    } else if (node->datum > item) {
      return contains_impl(node->left, i
    } else {
      return contains_impl(node->right, item);
    }
  }
};
```

static means the function can't access member variables. It shouldn't! The pointer to the root of the node structure is passed in.

Project 5 `BinarySearchTree.h`

- In project 5, you'll implement a BST.
- The starter files provide a code skeleton and we've implemented several pieces for you:
  - The overall structure
  - The data representation and `Node` struct
  - Iterators
  - The Big Three
- Many of these call static "`_impl`" functions for manipulating the recursive node structure.
  - You write the "`_impl`" functions!