**Intro to Programming II**  
**Recursion II**  
Chris Brooks  
Department of Computer Science  
University of San Francisco

---

### 18-0: Recursion in graphics

- We can use recursion to easily create interesting graphical effects.  
- For example, recursively tiling a surface.

---

### 18-1: Recursion in graphics: Exercise

- Add your own pictures to the applet.  
- Change the applet so that the recursive part of the picture is in the lower right.  
  - How can we do this?  
  - Let's try to trace the original program - maybe that'll give us some ideas.

### 18-2: Fractals

- We can also use recursion to draw fractals  
- Example: Koch snowflake  
- Rule: Each line segment is replaced by a "wedge" with sides that are the same length as the replaced piece.  
- As we increase the depth, it begins to look like a snowflake.

---

### 18-3: Koch Snowflake: exercise

- Change the color scheme of the applet.  
- Change the default max and min values  
- Change the initial triangle to have its "point" downward.

### 18-4: Trees

- Trees are a useful recursive data structure.  
- If we keep them sorted, we can find elements more quickly than in a list.  
- examples:
### 18-5: Tree Terminology

- Parent / Child
- Leaf node
- Root node
- Edge (between nodes)
- Path
- Ancestor / Descendant
- Depth of a node
  - Length of path from root to
- Height of a tree
  - (Depth of deepest node) + 1

### 18-6: Implementing a tree

- Treeneode class:
  ```java
  public class TreeNode {
    private Comparable data;
    private TreeNode left;
    private TreeNode right;
    public Object getData() { return data; }
  }
  ```

### 18-7: Binary Search Trees

- Binary Trees
  - For each node n, (value stored at node n) > (value stored in left subtree)
  - For each node n, (value stored at node n) < (value stored in right subtree)

### 18-8: Adding methods to BST

```java
public TreeNode insert(TreeNode tree, Comparable elem) {
  if (tree == null) {
    return new TreeNode(elem);
  }
  if ( elem.compareTo(tree.getData()) < 0 ) {
    tree.setLeft(insert(tree.left(), elem));
    return tree;
  }
  else {
    tree.setRight(insert(tree.right(), elem));
    return tree;
  }
}
```

### 18-9: Finding a node

- First, the Base Case – when is it easy to determine if an element is stored in a Binary Search Tree?

### 18-10: Finding an Element in a BST

- First, the Base Case – when is it easy to determine if an element is stored in a Binary Search Tree?
  - If the tree is empty, then the element can’t be there
  - If the element is stored at the root, then the element is there
18-11: Finding an Element in a BST

Next, the Recursive Case – how do we make the problem smaller?

18-12: Finding an Element in a BST

Next, the Recursive Case – how do we make the problem smaller?

- Both the left and right subtrees are smaller versions of the problem. Which one do we use?

18-13: Finding an Element in a BST

Next, the Recursive Case – how do we make the problem smaller?

- Both the left and right subtrees are smaller versions of the problem. Which one do we use?
- If the element we are trying to find is < the element stored at the root, use the left subtree. Otherwise, use the right subtree.

18-14: Finding an Element in a BST

Next, the Recursive Case – how do we make the problem smaller?

- Both the left and right subtrees are smaller versions of the problem. Which one do we use?
- If the element we are trying to find is < the element stored at the root, use the left subtree. Otherwise, use the right subtree.

18-15: Finding an Element in a BST

Next, the Recursive Case – how do we make the problem smaller?

- Both the left and right subtrees are smaller versions of the problem. Which one do we use?
- If the element we are trying to find is < the element stored at the root, use the left subtree. Otherwise, use the right subtree.

How do we use the solution to the subproblem to solve the original problem?

18-16: Finding an Element in a BST

Next, the Recursive Case – how do we make the problem smaller?

- Both the left and right subtrees are smaller versions of the problem. Which one do we use?
- If the element we are trying to find is < the element stored at the root, use the left subtree. Otherwise, use the right subtree.

How do we use the solution to the subproblem to solve the original problem?

The solution to the subproblem is the solution to the original problem (this is not always the case in recursive algorithms)
18-17: Finding an Element in a BST

To find an element $e$ in a Binary Search Tree $T$:
- If $T$ is empty, then $e$ is not in $T$
- If the root of $T$ contains $e$, then $e$ is in $T$
- If $e <$ the element stored in the root of $T$:
  - Look for $e$ in the left subtree of $T$
  - Otherwise
  - Look for $e$ in the right subtree of $T$

18-18: Exercise

- Copy the TreeNode class, including the insert method.
- Insert some numbers.
- Implement the find() method.
- How many recursive calls does it take to find something?
- How is this related to the number of elements in the tree?

18-19: Measuring efficiency

- How can we tell how long it takes an algorithm to run?
  - Implement and time it.
- What are some problems with this?

18-20: Big-O analysis

- We'd like to have a formal way to describe how long it takes for an algorithm to run, based on the size of the input.
- We'll focus on the worst-case situation.
- How long does it take to find something in an unsorted list of length $n$?
- How long does it take to insert something into a sorted list of length $n$?

18-21: Big-O analysis

- A $O(n)$ algorithm takes $C * n$ operations, where $C$ is a constant.
- A $O(n^2)$ algorithm takes $C * n^2$ operations, where $C$ is a constant.
- Since we want to get a rough idea of performance, we ignore constants and lower-order terms.

18-22: Lists and arrays

- How does it take to:
  - Find the nth element in a linked list.
  - Find the nth element in an array.
  - Insert an object at the front of a linked list
  - Insert an object at the front of an array.
  - Remove an element from a linked list
  - Remove an element from an array.
18-23: Searching in trees

- So, how long does it take to find something in a tree?
  - What if the tree is perfectly balanced?
  - What if it's completely unbalanced?

18-24: Counting nodes

- So how would we count the number of nodes in a tree?

18-25: Counting nodes

- If the tree is null, return 0.
- Otherwise, return 1 plus the number of nodes in the left subtree plus the number of nodes in the right subtree.
- Exercise: add a countNodes() method to our TreeNode class.