

# Introduction to Programming II

## Trees

Chris Brooks

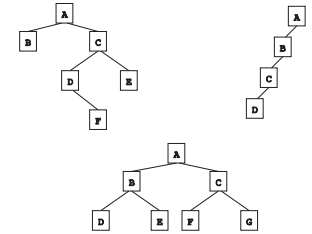
Department of Computer Science  
University of San Francisco

### 22-2: Trees

- Previously, we've talked about how to store objects in a linear sequence.
  - Arrays
  - ArrayLists
  - LinkedLists
- These are nice when we care about keeping everything in order.
- Finding particular elements can take a while, though.

### 22-3: Trees

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



### 22-4: Tree Terminology

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

### 22-5: Implementing a tree

- A struct representing a tree containing strings:

```
typedef struct treeNode *treeptr;

typedef struct treeNode {
    char *data;
    treeptr left;
    treeptr right;
} treeNode;
```

- We need to declare a type called "treeptr", because otherwise the C compiler doesn't know what a treeNode is until the definition is processed.

### 22-6: Implementing a tree

- Since C doesn't have constructors, it's often helpful to make them ourselves.
- Make a method called makeNode that takes a string as input and returns a new treeNode.

## 22-7: 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)

## 22-8: Adding methods to BST

```
void insert(treeptr root, char *newdata) {
    treeNode *newNode;

    if (strcmp(root->data, newdata) <= 0) {
        if (root->left == NULL) {
            newNode = makeNode(newdata);
            root->left = newNode;
            return;
        } else {
            return insert(root->left, newdata);
        }
    } else {
        if (root->right == NULL) {
            newNode = makeNode(newdata);
            root->right = newNode;
            return;
        } else {
            return insert(root->right, newdata);
        }
    }
}
```

## 22-9: Finding a node

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

## 22-10: 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

## 22-11: Finding an Element in a BST

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

## 22-12: 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?

#### 22-13: 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.

#### 22-14: 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.
- How do we use the solution to the subproblem to solve the original problem?

#### 22-15: 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?
  - The solution to the subproblem *is* the solution to the original problem (this is not always the case in recursive algorithms)

#### 22-16: 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  $T$

#### 22-17: Exercise

- Implement the `find(treeNode *root, char *object)` method.
- Write a `main()` that inserts names into the tree and tests `find()`.
- Count how many recursive calls it takes to find something.
- How is this related to the number of elements in the tree?

#### 22-18: 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?

#### 22-19: Counting nodes

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

#### 22-20: Counting nodes

- If the tree is null, return 0.
- Otherwise, return 1 + the number of nodes in the left subtree + the number of nodes in the right subtree.
- Exercise: add a `countNodes(treeNode *root)` function to our tree data structure.

#### 22-21: Counting leaves

- How would we change node counting to only count the number of leaves?
- Add a `countLeaves(treeNode *root)` function

#### 22-22: Tree traversals

- Many times when working with a tree, you need to visit every single node in a particular order.
  - For example, how would we print out all of the names in alphabetical order?

#### 22-23: Tree traversals

- Many times when working with a tree, you need to visit every single node in a particular order.
  - For example, how would we print out all of the names in alphabetical order?
  - Want to first visit the leftmost child, then its parent, then that parent's right child.
- This is called an *in order* traversal.

#### 22-24: In order traversal

```
void inorder(treeNode *root) {
    if (root == NULL) {
        return;
    } else {
        inorder(root->left);
        printf("%s", root->data);
        inorder(root->right);
    }
}
```

22-25: Exercise

- Modify the inorder function to make a:
  - Preorder traversal
  - Postorder traversal.