Data Structures

Today's Agenda

- **Continue Discussing Table Abstractions**
- But, this time, let's talk about them in terms of new non-linear data structures
  - trees
  - which require that our data be organized in a hierarchical fashion

Tree Introduction

- Remember when we learned about tables?
  - We found that none of the methods for implementing tables was really adequate.
  - With many applications, table operations end up not being as efficient as necessary.
  - We found that hashing is good for retrieval, but doesn't help if our goal is also to obtain a sorted list of information.

Tree Introduction

- We found that the binary search also allows for fast retrieval,
  - but is limited to array implementations versus linked list.
  - Because of this, we need to move to more sophisticated implementations of tables, using binary search trees!
  - These are "nonlinear" implementations of the ADT table.

Tree Terminology

- Trees are used to represent the relationship between data items.
  - All trees are hierarchical in nature which means there is a parent-child relationship between "nodes" in a tree.
  - The lines between nodes are called directed edges.
  - If there is a directed edge from node A to node B -- then A is the parent of B and B is a child of A.

Tree Terminology

- Children of the same parent are called siblings.
- Each node in a tree has at most one parent, starting at the top with the root node (which has no parent).
- Parent of n — The node directly above node n in the tree
- Child of n — The node directly below the node n in the tree

Tree Terminology

- Root — The only node in the tree with no parent
- Leaf — A node with no children
- Siblings — Nodes with a common parent
- Ancestor of n — A node on the path from the root to n

Tree Terminology

- Descendant of n
  - A node on a path from n to a leaf
- Empty tree
  - A tree with no nodes
- Subtree of n
  - A tree that consists of a child of n and the child's descendants
- Height
  - The number of nodes on the longest path from root to a leaf

Tree Terminology

- Binary Tree
  - A tree in which each node has at most two children
- Full Binary Tree
  - A binary tree of height h whose leaves are all at the level h and whose nodes all have two children; this is considered to be completely balanced

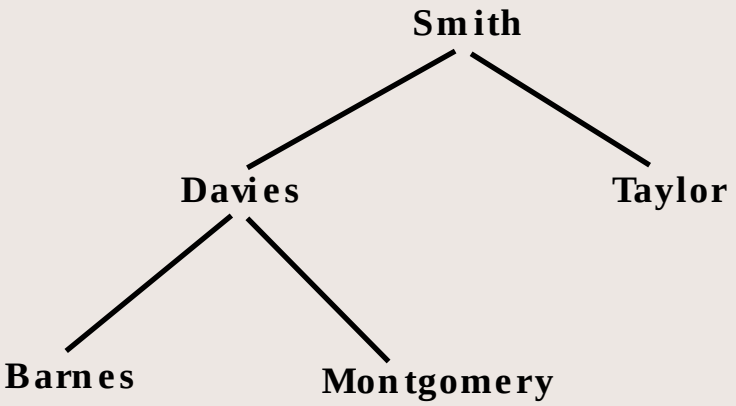
Binary Trees

- A binary tree is a tree where each node has no more than 2 children.
  - If we traverse down a binary tree -- for every node -- there are either no children (making this node a leaf) or there are two children called the left and right subtrees
  - (A subtree is a subset of a tree including some node in the tree along with all of its descendants).

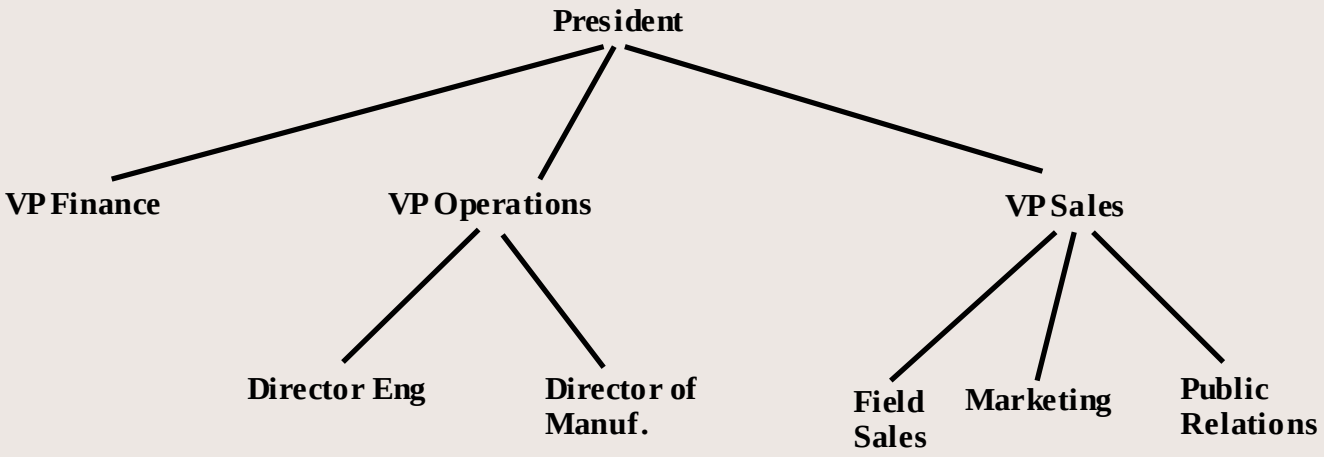
Binary Search Trees

- The nodes of a binary tree contain values.
- For a **binary search tree**, it is really sorted according to the key values in the nodes.
  - It allows us to traverse a binary tree and get our data in sorted order!
  - For example, for each node n, all values greater than n are located in the right subtree...all values less than n are located in the left subtree. Both subtrees are considered to be binary trees themselves.

Binary Search Trees



NOT a Binary Search Tree



Binary Search Trees

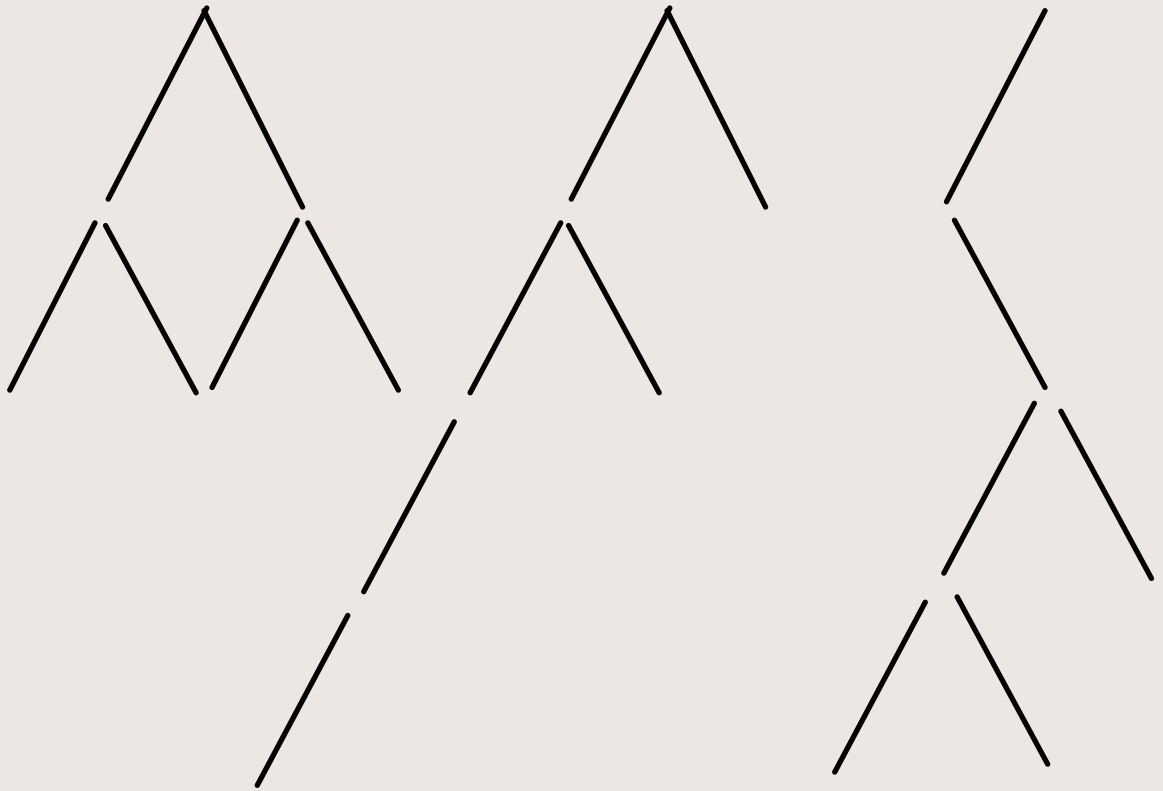
- Notice that a binary tree organizes data in a way that facilitates searching the tree for a particular data item.
- It ends up solving the problems of sorted-traversal with the linear implementations of the ADT table.
- And, if reasonably balanced, it can provide a logarithmic retrieval, removal, and insertion performance!

Binary Trees

- Before we go on, let's make sure we understand some concepts about trees.
- Trees can come in many different shapes. Some trees are taller than others.
- To find the height of a tree, we need to find the distance from the root to the farthest leaf. Or....you could think of it as the number of nodes on the longest path from the root to a leaf.

Binary Trees

- Each of these trees has the same number of nodes -- but different heights:



Binary Trees

- You will find that experts define heights differently.
- For example, just by intuition you would think that the trees shown previously have a height of 2 and 4.
- But, for the cleanest algorithms, we are going to define the height of a tree as the following (next slide)

Binary Trees

- If a node is a root, the level is 1. If a node is not the root,
  - then it has a level 1 greater than its parent.
- If the tree is entirely empty,
  - then it has a height of zero.
- Otherwise, its height is equal to the maximum level of its nodes.
- Using this definition,
  - the trees shown previously have the height of 3, 5, and 5.

Full Binary Trees

- Now let's talk about full, complete, and balanced binary trees.
- A full binary tree has all of its leaves at level h.
- In the previous diagram, only the left hand tree is a full binary tree!
- All nodes that are at a level less than the height of the tree have 2 children.

Complete Binary Trees

- A complete binary tree is one which is a full binary tree to a level of its height-1 ...
  - then at the last level, it is filled from left to right. For example:

Binary Search Trees

- This has a height of 4 and is a full binary tree at level 3.
- But, at level 4, the leaves are filled in from left to right!
- From this definition, we realize that a full binary tree is also considered to be a complete binary tree.
- However, a complete binary tree does not necessarily mean it is full!

Implementing Binary Trees

- Just like other ADTs,
  - we can implement a binary tree using pointers or arrays.
  - A pointer based implementation example:

```
struct node {
        data value;
        node * left_child;
        node * right_child;
};
```

Binary Search Trees

- In what situations would the data being “stored” in the node...
  - be represented by a pointer to the data?

```
struct node {
        data * ptr_value;
```

  - when more than a single data structure needs to reference the same tree (e.g., two binary search trees referencing the same data but organized on two different keys!)

Binary Search Trees

- In what situations would the data being "stored" in the node...
  - be represented by a pointer to a LLL node?

```
struct tree_node {
    LLL_node * head;
```

  - when each node's data is actually a list of items (a general purpose list, stack, queue, or other ordered list representation)

Binary Search Trees

- In what situations would the data being “stored” in the node...
  - be represented by an array of data?

```
struct tree_node {
    data ** array;
```

  - when each node’s data is actually a list of items (a general purpose list, stack, queue, or other ordered list representation), but where the size and efficiency of this data structure is preferred over a LLL

Implementing Binary Trees

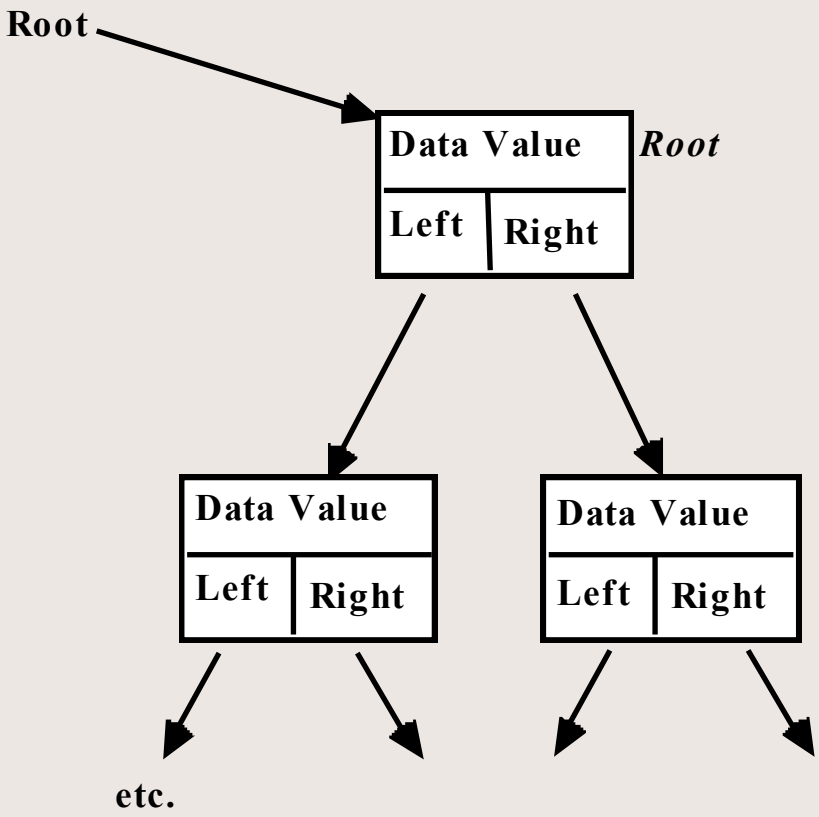
```
class binary_tree {
  public:
     binary_tree();
     ~binary_tree();
     int insert(const data &);
     int remove(const key &);
     int retrieve (const key &,
     data [], int & num_matches);
     void display();
```

Implementing Binary Trees

```
//continued....class interface
  private:
     node * root;
};
```

- Notice that instead of using “head” we use “root” to establish the “starting point” in the tree
- If the tree is empty, root is NULL.

Implementing Binary Trees



Implementing Binary Trees

- When we implement binary tree algorithms
  - we have a choice of using iteration or recursion and still have reasonably efficient results
  - remember why we didn't use recursion for traversing through a standard linear linked list?
  - now, if the tree is reasonably balanced, we can traverse through a tree with a minimal number of recursive calls

Traversal through BSTs

- Remember that a binary tree is either empty or it is in the form of a Root with two subtrees.
  - If the Root is empty, then the traversal algorithm should take no action (i.e., this is an empty tree -- a "degenerate" case).
  - If the Root is not empty, then we need to print the information in the root node and start traversing the left and right subtrees.
  - When a subtree is empty, then we know to stop traversing it.

Traversal through BSTs

- Given all of this, the recursive traversal algorithm is:

Traverse (Root)

If the Tree is not empty then

Visit the node at the Root (maybe display)

Traverse(Left subtree)

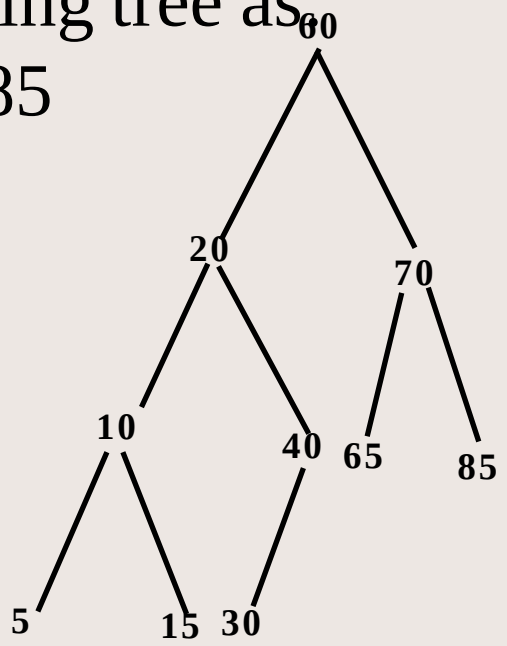
Traverse(Right subtree)

Traversal through BSTs

- But, this algorithm is not really complete.
- When traversing any binary tree, the algorithm should have 3 choices of when to process the root:
  - before it traverses both subtrees (like this algorithm),
  - after it traverses the left subtree,
  - or after it traverses both subtrees.
  - Each of these traversal methods has a name: preorder, inorder, postorder.

Traversal through BSTs

- You've already seen what the preorder traversal algorithm looks like...
  - it would traverse the following tree as: 60,20,10,5,15,40,30,70,65,85
  - but what would it be using inorder traversal?
  - or, post order traversal?



Traversal through BSTs

- The inorder traversal algorithm would be:

Traverse (Root)

If the Tree is not empty then

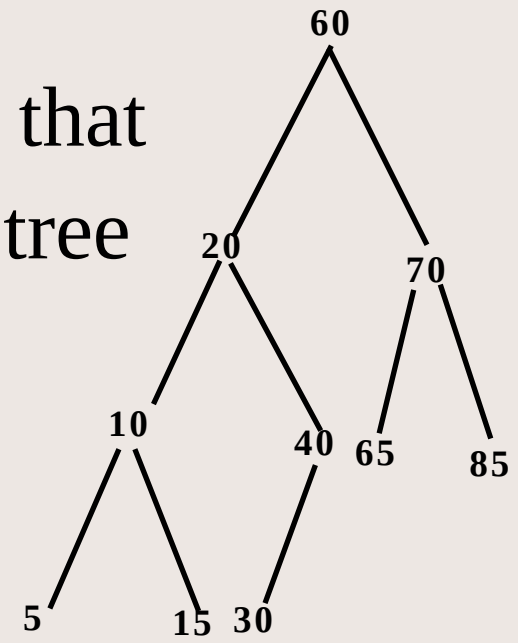
Traverse(Left subtree)

Visit the node at the Root (display)

Traverse(Right subtree)

Traversal through BSTs

- It would traverse the same tree as: 5,10,15,20,30,40,60,65,70,85;
- Notice that this type of traversal produces the numbers in order.
- Search trees can be set up so that all of the nodes in the left subtree are less than the nodes in the right subtree.



Traversal through BSTs

- The postorder traversal is:

  If the Tree is not empty then

  Traverse(Left subtree)

  Traverse(Right subtree)

  Visit the node at the Root (maybe display)
- It would traverse the same tree as:
  - 5, 15, 10,30,40,20,65,85,70,60

Traversal through BSTs

- Think about the code to traverse a tree inorder using a pointer based implementation:

```
void inorder_print(tree root) {
    if (root) {
     inorder_print(root->left_child);
     cout <<root->value.name);
     inorder_print(root->right_child);
    }
```

Traversal through BSTs

- Why do we pass root by value vs. by reference?

```
void inorder_print(tree root) {
```

- Why don't we say??

```
root = root->left_child;
```

- As an exercise, try to write a nonrecursive version of this!

Using BSTs for Table ADTs

- We can implement our ADT Table operations using a nonlinear approach of a binary search tree.
- This provides the best features of a linear implementation that we previously talked about plus you can insert and delete items without having to shift data.
- With a binary search tree we are able to take advantage of dynamic memory allocation.

Using BSTs for Table ADTs

- Linear implementations of ADT table operations are still useful.
- Remember when we talked about efficiency, it isn't good to overanalyze our problems.
- If the size of the problem is small, it is unlikely that there will be enough efficiency gain to implement more difficult approaches.
- In fact, if the size of the table is small using a linear implementation makes sense because the code is simple to write and read!

Using BSTs for Table ADTs

- For test operations, we must define a binary search tree where for each node -- the search key is greater than all search keys in the left subtree and less than all search keys in the right subtree.
  - Since this is implicitly a sorted tree when we traverse it inorder, we can write efficient algorithms for retrieval, insertion, deletion, and traversal.
  - Remember, traversal of linear ADT tables was not a straightforward process!

Using BSTs for Table ADTs

- Let's quickly look at a search algorithm for a binary search tree implemented using pointers (i.e., implementing our Retrieve ADT Table Operation):
- The following is pseudo code:

```
int retrieve (tree *root, key &k, data & value){
if (!root)                          //we have an empty tree
        return 0;
```

Using BSTs for Table ADTs

```
    else if (root->value == k) {
            value = root->value;
            return 1;
            }
else if (k < root->value)
            return retrieve(root->left_child, k,data);
else
    return retrieve(root->right_child, k, data);
    }
```

For Next Time...

- To prepare for next class
  - write C++ code to insert a new data item at a leaf in the appropriate sub-tree using the binary search tree concept
  - think about what you might need to do to then remove an item?
  - what special cases will we need to consider?
  - how might we make a copy of a binary search tree?