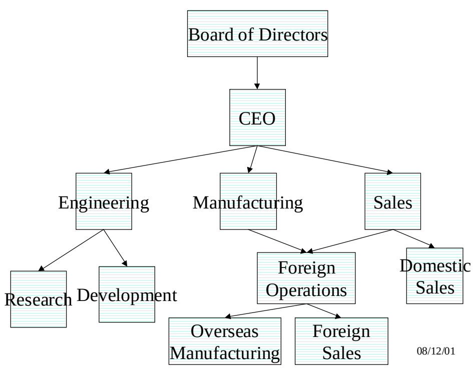
CSE 143

Trees

[Chapter 10]

Linear vs. Branching

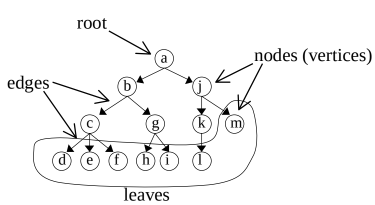
- Our data structures so far are linear
  - Have a beginning and an end
  - Everything falls in order between the ends
  - Arrays, linked lists, queues, stacks, priority queues, etc.
- Everyday life has **branching** structures, too.
  - Family genealogy
  - Biology: phylum/genus/species
  - Company organization chart
  - Table of contents



Branching Structures in CS

- **Trees** are a common branching structure in CS
- We've seen already:
  - Class hierarchies
  - Call graphs
  - Recursive function traces
- PS: Some of these won't quite be "trees" under our official definition
  - The org chart was not a tree (go back later and see why)

A Tree



What's in a Node?

- Answer: anything you want!
- Could have a tree of ints, tree of students, animals, appointments, etc.
  - All nodes will be of the same (base) type
- For simplicity, we often label the nodes with a single letter or an integer

Formal Textbook Definition

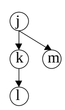
- A *general tree* T is either empty, or is a set of nodes such that T is partitioned into disjoint subsets:
  1. A subset with a single node r (called the root)
  2. Subsets that are themselves general trees (these are called the subtrees of T).
- Notes:
  - This definition is recursive!
  - The nodes are not defined. They can be anything, and still satisfy the definition.

Tree Terminology

- *Empty tree*: tree with no nodes
- *Child* of a node u
  - Any node reachable from u by 1 edge pointing away from u
  - Nodes can have zero, one, or more children
- *Leaf:* a node with no children
- If `b` is a child of `a`, then `a` is the *parent* of `b`
  - All nodes except root have exactly one parent
  - *Root* has no parent

Descendants

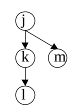
- *Descendant* of a node (recursive definition)
  - 1. P is a descendant of P for any node P
  - 2. If C is a child of P, and P is a descendant of A, then C is a descendant of A
- Puzzle: neither rule states explicitly that if C is a child of P, C is also a descendant of P. Is it? Do we need another rule?



- Example:
  - what are the descendents of `j`?
  - Of what is l a descendant?

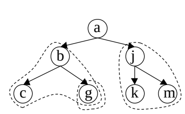
Ancestors

- *Ancestor* of a node
  - Definition: If D is a descendant of A, then A is an ancestor of D
- Example: `j`, `k`, and `l` are ancestors of `l`



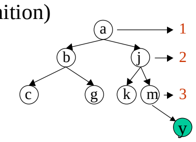
Subtree Terminology

- Subtree
  - Any node of a tree, with all of its descendants
  - Puzzle: is b-c a subtree of the tree starting at a? Is it a tree?



Height and Level

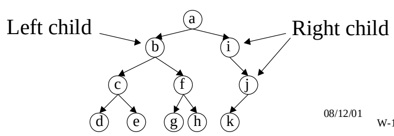
- Level or depth (recursive definition)
  - Level of root node is 1
  - Level of any node other than root is one greater than level of its parent



- Height
  - Height of a tree is maximum of all depths of its leaves
  - Height of empty tree is defined to be 0
- Warning: Definitions vary
  - Some textbooks define level of the root node as 0,
    - so root node height would be 0, empty tree height would be -1

Binary Trees

- A *binary tree* is a tree each of whose nodes has no more than two children
  - The two children are called the *left child* and *right child*
    - The trees which start with these children are called the *left subtree* and the *right subtree*
  - See textbook for formal recursive definition



Importance of Binary Trees

- Binary trees are widely used in Computer Science
- Much easier to represent (find a good data structure for) than general trees
- Much easier to manipulate (write and implement algorithms) than general trees
- Turns out that any general tree can be represented using a binary tree.
  - Won't discuss in this course

Binary Tree as an ADT

- Textbook lists 18 operations!
  - constructors and destructors
  - bool isEmpty
  - return/set root data
  - attach left or right child nodes
  - attach left or right subtrees
  - detach left or right subtrees
  - return a copy of left or right subtree
  - traversals (more late)

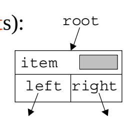
Implementing A Binary Tree

- Using an array
  - Efficient
  - See textbook 452-453 for details
    - won't discuss further in this course
  - Drawbacks: not flexible in terms of size; wastes space if tree is unbalanced
- Using dynamic memory
  - Similar to linked list implementation
  - Two pointers, one each for left and right subtrees
  - See textbook 455ff for details
    - Will use in this course

Binary Tree Data Structure

- Binary tree node (for a tree of ints):

```
struct BTreeNode {
  int item;
  BTreeNode *left;
  BTreeNode *right;
};
```



- Keep a `root` pointer to the root node
  - Analogous to *head* pointer for a linked list
  - Empty tree has a NULL root
    - will usually omit NULL pointers when drawing pictures
- This example shows node for a tree of ints
  - but "item" could be any type, even a class object

Example: Counting Nodes

- Base case: Empty tree has zero nodes
- Recursive case: Nonempty tree has one node (the root) plus nodes in left subtree plus nodes in right subtree

```
// return # of nodes in tree with given root
int CountNodes(BTreeNode *root)
{
  if ( root == NULL )
    return 0;      // base case
  else
    return 1 + CountNodes(root->left)
             + CountNodes(root->right);
}
```

Binary Trees and Recursion

```
struct BTreeNode {
  int item;
  BTreeNode *left;
  BTreeNode *right;
};
```

- Note the recursive data structure
- Algorithms often are recursive as well
- Don't fight it! Recursion is going to be the natural way to express the algorithms
  - Challenge: code CountNodes without using recursion

Finding the Height

- Base case: Empty tree has height `0`
- Recursive case: Nonempty tree has height `1` more than maximum height of left and right subtrees

```
// returns height of tree with given root
int Height(BTreeNode *root) {
  if ( root == NULL )
    return 0;
  else
    return 1 + max(Height(root->left),
                   Height(root->right));
}
```


Analyses

- What is running time of these algorithms?
  - Time to execute for one node: `O(1)`
  - Number of recursive calls: `O(N)`
    - `N` is the number of nodes in tree
    - There's no way to miss any node
    - There's no way to get to any node twice
      - Each node is called from its parent, and a node has only one parent

Exercises

Do try these at home!

- 1. Find the sum of all the values (items) in a binary tree of integers
- 2. Find the smallest value in a B.T. of integers
- 3. (A little harder) Count the number of leaf nodes in a B.T.
- 4. (A little harder) Find the average of all the values in a B.T. (one approach: think in terms of a "kickoff" function)

Recursive Tree Searching

- How to tell if a data item is in a binary tree?

```
// true iff "item appears in tree with given
   root"
bool find(BTreeNode *root, int item) {
  if ( root == NULL )
    return false;
  else if ( root->data == item )
    return true;
  else
    return ( find(root->left, item) ||
             find(root->right, item) );
}
```


Complexity of Find

- What is the running time of this algorithm?
  - Worst case: Has to visit every node in the tree, `O(N)`
- Can we do better?
  - Answer: not without changing the data structure
  - We will shortly look at a binary *search* tree
    - Items will have an order, which will make searching more efficient.
  - But first we take up another topic: traversals.

Tree Traversal

- Functions to count nodes, find height, sum, etc. systematically "visit" each node
- This is called a *traversal*
  - We also used this word in connection with lists.
- Traversal is a common pattern in many algorithms
  - The processing done during the "visit" varies with the algorithm
- What order should nodes be visited in?
  - Many are possible
  - Three have been singled out as particularly useful: *preorder, postorder, and inorder*

Pre and Post Order Traversals

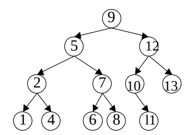
- Preorder traversal:
  - "Visit" the (current) node *first*
    - i.e., do what ever processing is to be done
  - Then, (recursively) do preorder traversal on its children, left to right
- Postorder traversal:
  - First, (recursively) do postorder traversals of children, left to right
  - Visit the node itself *last*
- PS: These algorithms make sense for non-binary trees, too.

Inorder

- Unlike pre- and post-, makes sense only for binary trees
- Inorder traversal:
  - (Recursively) do inorder traversal of left child
  - Then visit the (current) node
  - Then (recursively) do inorder traversal of right child

Example of Tree Traversal

Assume this question: in what order are the nodes visited, if we start the process at the root?



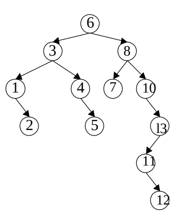
Preorder:

Inorder:

Postorder:

More Practice

What about this tree?



Preorder:

Inorder:

Postorder:

Two Traversals for Printing

```
void printInOrder(BTreeNode* t) {
  if (t != NULL) {
    printInOrder(t->left);
    cout << t->data << " ";
    printInOrder(t->right);
  }
}
```

```
void printPreOrder(BTreeNode* t) {
  if (t != NULL) {
    cout << t->data << " ";
    printPreOrder(t->left);
    printPreOrder(t->right);
  }
}
```

Traversing to Delete

- Use a postorder traversal to delete all the nodes in a tree

```
// delete binary tree with root t
void deleteTree(BTreeNode* t) {
  if (t != NULL) {
    deleteTree(t->left);
    deleteTree(t->right);
    delete t;
  }
}
```

- Puzzler: Would inorder or preorder work just as well??

Analysis of Tree Traversal

- How many recursive calls?
  - Two for every node in tree (plus one initial call);
  - `O(N)` in total for `N` nodes
- How much time per call?
  - Depends on complexity `O(V)` of the visit
  - For printing and most other types of traversal, visit is `O(1)` time
- Multiply to get total
  - `O(N)*O(V) = O(N*V)`
- Does tree shape matter?

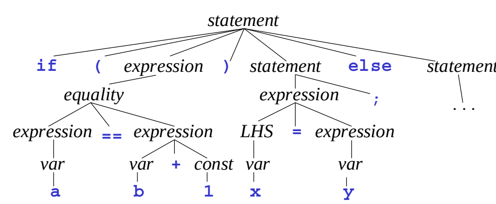
Sidebar: Syntax and Expression Trees

- Computer programs have a hierarchical structure
  - All statements have a fixed form
  - Statements can be ordered and nested almost arbitrarily (nested `if-then-else`)
- Can use a structure known as a *syntax tree* to represent programs
  - Trees capture hierarchical structure

A Syntax Tree

Consider the C++ statement:

```
if ( a == b + 1 ) x = y; else ...
```



Syntax Trees

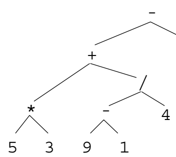
- Compilers usually use syntax trees when compiling programs
  - Can apply simple rules to check program for syntax errors
  - Easier for compiler to translate and optimize than text file
- Process of building a syntax tree is called *parsing*

Binary Expression Trees

- A *binary expression tree* is a syntax tree used to represent meaning of a mathematical expression
  - Normal mathematical operators like `+`, `-`, `*`, `/`
- Structure of tree defines result
- Easy to evaluate expressions from their binary expression tree

Example

`5 * 3 + (9 - 1) / 4 - 1`



08/12/01 W-37

Expression Magic

- Traverse in <u>postorder</u> for <u>postfix</u> notation!
  `5 3 * 9 1 - 4 / + 1 -`
- Traverse in <u>preorder</u> for <u>prefix</u> notation
  `- + * 5 3 / - 9 1 4 1`
- Traverse in <u>inorder</u> for <u>infix</u> notation
  `5 * 3 + 9 - 1 / 4 - 1`
  - Note that operator precedence may be wrong! Correction: add parentheses at every step

  `(((5*3) + ((9 - 1) / 4)) - 1)`

08/12/01 W-38

Trees Summary (1)

- Tree as new hierarchical ADT
  - Recursive definition
  - recursive data structure
- Tree terminology
  - Nodes; Root node, leaf nodes
  - Children, parents, ancestors, descendants
  - Depth of node, height of tree
  - Subtrees

08/12/01 W-39

Trees Summary (2)

- Binary Trees
  - Either 0, 1, or 2 children at any node
  - Recursive functions to manipulate them
- Binary Tree Implementation
  - Via node with two pointers
- Tree Traversals
  - Preorder traversal
  - Postorder traversal
  - Inorder traversal (binary trees only)
- Expression and Syntax Trees

08/12/01 W-40