CSCI 2270
Data Structures and Algorithms
Lecture 16

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wed 1-2pm
Fri 2-3pm

Administrivia

Thursday 2-3 office hours have moved to Friday 2-3

HW2 will post on Monday

linked list implementation of a biiig integer in any base

Read 10.2, Linked Lists

Skim Free Trees B.5.1

Read Rooted and Ordered Trees, B.5.2

Read Binary Search Trees, 12.1-12.4

Next week's lab: binary search trees

Colloquia: check the videos link if you need review…

Queues

Finish ‘wrapped’ array based queue

Trees

Trees are like linked lists that can branch out (but never in)

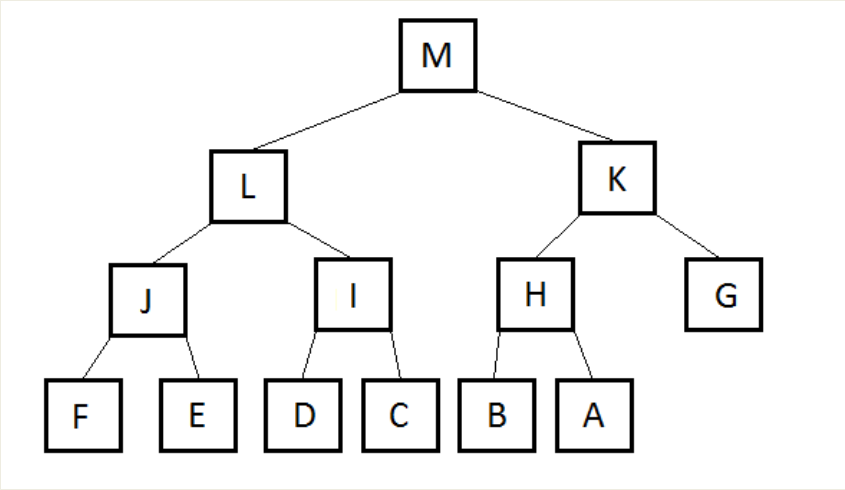
Tree nodes have more than one next pointer

Instead of a head_ptr, we keep a pointer to the tree's root

nodes farthest from the root are the leaves

Hierarchy: parent/child, ancestor, sibling

Trees contain subtrees (recursive structure)



Binary trees

Binary tree nodes:

contain data;

have 2 children per node (n-ary tree: n children)

We call the children the left and right subtrees

Used to represent arithmetic expressions

Tree height

Empty tree: height = 0

Else tree height is = 1 + height of largest subtree

(what's important about this definition?)

If binary tree is full, no nodes are missing at any height

height h >= 0; tree has $2^h - 1$ nodes

if tree has n nodes, height is $\log_2(n+1)$

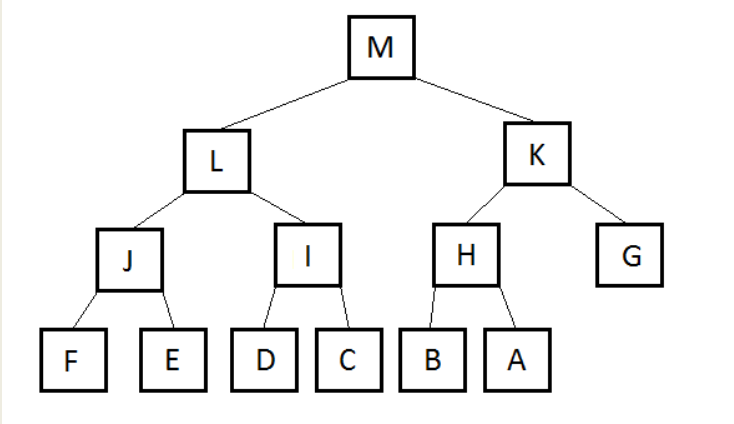
If it's complete, then the last row can be partially filled in,

left to right, with no skipped children.

Opposite of complete tree (degenerate case) looks like a list

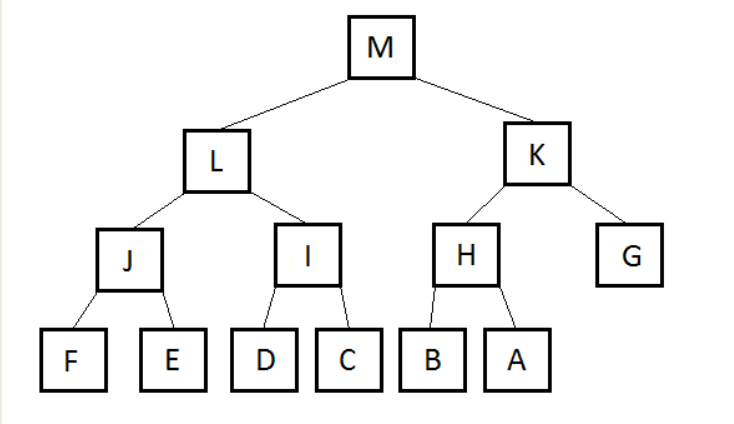
Tree traversals

Recursive process



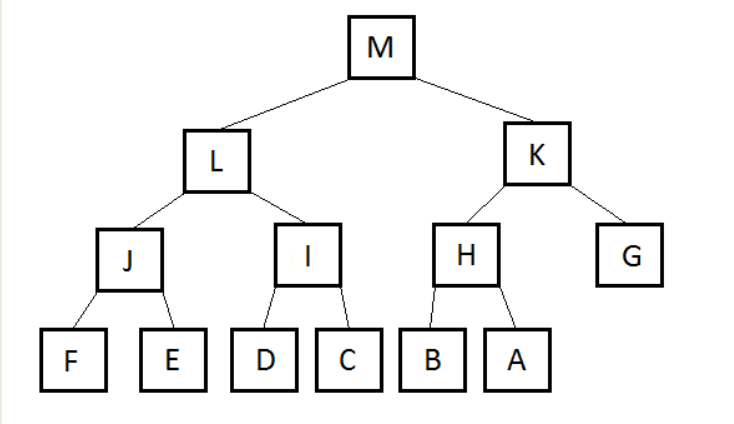

```
preorder_print (const binary_tree_node* bintree)
        if (bintree is not empty)
        {
                print out data at root
                preorder_print(left subtree of bintree's root)
                preorder_print(right subtree of bintree's root)
        }
```

Tree traversals




```
inorder_print (const binary_tree_node* bintree) :
        if (bintree is not empty)
        {
                inorder_print(left subtree of bintree's root)
                print out data at root
                inorder_print(right subtree of bintree's root)
        }
```

Tree traversals




```
postorder_print (const binary_tree_node* bintree) :
        if (bintree is not empty)
        {
                postorder_print(left subtree of bintree's root)
                postorder_print(right subtree of bintree's root)
                print out data at root
        }
```

Binary trees

Binary tree nodes:

- contain data;
- have 2 children per node (n-ary tree: n children)
- We call the children the left and right subtrees

Used to represent arithmetic expressions

Binary search trees: special binary trees

- *can* work faster than lists
- data at n is >= values in n's left subtree
- data at n is < values in n's right subtree

Binary *search* tree: search

Recall that data in node n is >= data in n's left subtree

data in node n is < data in n's right subtree

```
search(const binary_tree_node* bstree, int target) :
        if (bstree is empty)
                target not found, boo!
        else if (target equals data item at root)
                target found, yay!
        else if (target < data item at root)
                search(left subtree of bstree, target)
        else
                search(right subtree of bstree, target)
```

Binary search tree: insert

Find where item would go and then put it there as a new leaf

Shape of tree depends on order in which items are inserted

Trees offer the chance for log(n) performance if you're lucky

For a binary search tree of n nodes,

Search: O(log n) average, O(n) worst case

Insert: O(log n) average, O(n) worst case

Remove: O(log n) average, O(n) worst case

Traverse: O(n)

Another tree ADT, the B-tree, ensures that the tree is perfectly full—guarantees the O(log n) performance, worst case