Announcements

- Issues with midterm
  - I will be grading them this week, stop by oh to pick up end of the week
  - Extra OH to make up for last week
  - If you have an issue, please come by, I don’t read minds....
  - Homework 2 out

- Reading:
  - Chapter 4.1-4.3,4.4
  - Skip 4.3.5, 4.5
  - Next time 4.4, 4.7
Blast from the past

- We introduced the idea of a tree data structure
  - Root is top
  - Each node can have zero or more children up to some limit
  - Leaf are nodes with no children non-leafs are internal nodes

- Operations:
  - Insert
  - Find
BST

- Binary search tree is simply a tree, where we force (during insert) each item to be greater than everything in the left child, but less than everything on the right child

- Should be comfortable with insert, finding, and getting height of tree

- Any questions on printing ??
Reminder

- Tree can be drawn as a nice directed graph on the board or screen

- Printer and other devices have limited quality, so need to print it out in some manageable form

- Inorder, preorder, postorder are ways of printing out the tree
Question

- Can I have two trees with the same inorder printout but are not the same shape ??
Let me show you what I mean
delete

- Occasionally we need to do some clean up on a data structure
  - Although sometimes you will have a Tree where only inserts and finds happen

- Any ideas on what issues delete needs to deal with
- Runtime ??
Strategy

- Depending on the node type
  - Leaf:
    - Just delete
  - One child:
    - Simply move the link
  - Two children:
    - Switch with smallest child of right subtree
    - Delete it
Code example
Types of deletion

- For many data structures
  - Deletion might be complicated
  - Might want to mark it as deleted
  - So its there, but ignored

- Known as “LAZY DELETION”
Another flavor of binary trees are expression trees
- internal nodes: arithmetic operations
- leaves: numbers/variables
Can you turn this into postfix ??
What about infix ??
question

- can you write a pseudo code algorithm for converting postfix to expression trees ??
while ( /*some input*/) {
    // read in symbol
    if(isSymbol(s)) {
        // make tree t from s
        // push to a stack
    }
    else {
        // pop 2 trees t1, t2 from the stack
        // combine to form tree t3
        // push to stack
    }
}
}
One of the nice things about binary trees are some interesting mathematical properties inherit in the tree structure

- Help in analyzing our running times
- We can take advantage of them to further design DS
Prove

- Can you prove that at level $i$ the binary tree has $2^i$ slots??
Proof structure of binary tree

- **Show:**
  - at 0 it has $2^0$ slots

- **Assume**
  - ??

- **Prove**
  - At level $N$ each of the $2^N$ slots will have at most 2 children
  - so next level: $2 \times 2^N = 2^{N+1}$
  - which means level $N+1$ has $2^{N+1}$
Prove

- If we have N internal nodes in a binary tree, how many external nodes will we have??
If N internal nodes, we have N+1 external nodes

Idea:
- for the tree to move from N internal to N+1 internal, an external must add children.
- subtract one external (its now internal)
- can have up to +2 children
Definition

- Complete Binary Tree:
  - this is a binary tree which is filled in left to right on each level so no empty spots

- some examples

- Now the most important question:
WHO CARES ????
Advantages

- First advantage to complete binary trees is in representing them

- you can now use an array to represent the tree

- Let me draw it out:
  - Given a node A[i]
    - where would the left and right child be ??
- **A[i]**
  - left is A[i*2]
  - right A[i*2+1]
  - parent A[i/2]
  - root A[1]
    - can save A[0] for other stuff

- any ideas on how to check if A[i] is a leaf ??
Height

- You hopefully remember how to get height of a tree
- For a binary tree with N internal nodes, its height is between \( \log n \) and \( n-1 \)
  - worst case
    - linked list case
  - best
    - any ideas ??
run time on trees

- basic operations on trees:
  - Insert:
  - Find:

- What would an interface class for a tree look like?
  - Why write it?
  - How?

- what are the worst case, and best case??
- idea is to have the tree as balanced as possible as a BST to give us best case scenario

- need to keep it BST property while balanced
with BST

insert
  - 5, 14, 3, 12

now:
  - 14, 12, 5, 3
Overhead

- if you want to keep the BST tree balanced you might end up with a situation where inserts could end up being $n + \log n = O(n)$

- need a better way
Adelson-Velskii & Landis

- AVL Tree
  - data structure which keeps the trees (almost) balanced at all times, using AVL property

- AVL Property
  1. $H(\text{right}) == H(\text{left})$
  2. $H(\text{right})$ differs from $H(\text{left})$ by 1

  $H(\text{empty tree}) = -1$

  $H(\text{node}) = \text{longest path}$
run time

- because the tree is always balanced:
  - inserts = \( \log n \)
  - find = \( \log n \)
AVL Rotation

- Rotation is the operation of keeping an AVL tree balanced.

- Remember that because it's always an AVL tree, the only way to unbalance it is by insert/delete a node.

- If $\alpha$ is the node that needs rebalancing, it means $\alpha$ subtree was edited in some way.

- Divides itself into 4 cases.
1. Just inserted into left subtree of left child
2. Just inserted into right subtree of left child
3. Just inserted into left subtree of right child
4. Just inserted into right subtree of right child

- single rotations 1,4
- double rotations 2,3

lets show this graphically:
- insert 5,2,8
- 1,4,7,3

- now insert 6
  - which is unbalanced?
  - which rule is used?

- how do you code this??