Announcements

• A2 was posted Wed night
• Apply lessons learned from A1
  – Start early
  – Tear that handout apart
  – Get clarification
  – Don’t write any code until you understand the docstrings and the handout
  – Write good docstrings and doctest before code
  – Test, test, test
  – Working with your partner effectively, or work alone
Announcements

• The latest exercise hasn’t been testing yet
  – Very likely this weekend

• Midterm #2 is one week from today
  – We will post details on Piazza this weekend
  – Linked lists, trees, recursion will be emphasized
Binary trees

• Our general Tree class allows any number of children per node.
• The branching factor of a node is its number of children.
• Sometimes we impose a limit.
• A binary tree has branching factor 2.
• We call the children the left and right children.
Example binary tree
Tree traversal

• Tree traversal: visiting all nodes to do something at each.

• We’ve seen 2 orders already:
  – Pre-order: visit a node before its children.
  – Post-order: visit a node after its children.

• Another option for binary trees, since there are at most 2 children per node:
  – In-order: visit a node in between its children.
Representing a binary tree

- **General tree class:**
  - a list of subtrees
  - we can make the list the right length to hold the non-empty subtrees
  - so we have no empty trees among the subtrees

- **Binary tree class:**
  - two named attributes: _left and _right
  - if there is only one non-empty subtree, we still need to know which side it’s on
  - So we allow _left and _right to have empty subtrees
Specific to our implementation

• If \_root is None, so are \_left and \_right
  – This represents an empty tree.
  – Same as what we did for our general tree class.
  – Allows empty and non-empty trees to be instances of the same class.

• If \_root is not None, \_left and \_right are trees.
  – They cannot be None.
  – This means each leaf as two (empty tree) children.
  – And a node with one child has an empty tree on the other side.
  – This is not what we did for our general tree class.
  – It makes the recursion more elegant.
Binary search trees

• A **binary search tree** is a binary tree in which:
  For every node, its value is
  – greater than (or equal to) every value in its left subtree
  – less than (or equal to) every value in its right subtree

• It follows that any subtree of a BST is also a BST.

• Variations on the definition:
  – We are allowing duplicates to go on either side.
  – Another option is to specify which one side they can go on. Which one is an arbitrary choice – it doesn’t matter.
Example binary search tree
Search: method **contains**

- How will it be different for a BST than for a general tree?
  - At each node, if we don’t find what we’re looking for, we know whether to look left or right. We do not have to check both.
BST insertion

• In class we
  – analyzed the cases and wrote the code.
  – observed that insertions always add a new leaf.

• Challenge: Try to find a case where we can’t insert by adding a leaf. (There is no such case!)
BST deletion

• In class we considered the easier cases
  – The easiest case: delete a leaf.
  – The next-easiest case: delete a node one level up. The value in either child could go in its place.
  – In both, cases, little to no structural change is required.

• Then we considered deleting a root farther up
  – If we could find a leaf whose value could go in the root, we would avoid big structural changes.
  – We observed that the largest value on the left, or the smallest on the right, would work.
  – We wrote the values in the tree in order, and saw that this makes sense.