Outline

■ Binary Trees
■ Traversals
■ Binary Search Trees
Binary Tree Node

Change our generic Tree design so that we have two named children, left and right, and can represent an empty tree with None.

```python
class BinaryTree:
    """
    A Binary Tree, i.e. arity 2.
    """
    def __init__(self, data, left=None, right=None):
        """
        Create BinaryTree self with data and children left and right.
        @param BinaryTree self: this binary tree
        @param object data: data of this node
        @param BinaryTree|None left: left child
        @param BinaryTree|None right: right child
        @rtype: None
        """
        self.data, self.left, self.right = data, left, right
```

CSC148 L5102 Spring 2016
Special Methods...

We’ll want the standard special methods:

- __eq__
- __str__
- __repr__
Arithmetic Expression Trees

Binary arithmetic expressions can be represented as binary trees:
Evaluating a Binary Expression Tree

- There are no empty expressions
- If it’s a leaf, just return the value
- Otherwise...
  - Evaluate the left tree
  - Evaluate the right tree
  - Combine left and right with the binary operator

Python built-in `eval` might be handy.
Traversal

The functions and methods we have seen get information from every node of the tree – in some sense they traverse the tree.

Sometimes the order of processing tree nodes is important: do we process the root of the tree (and the root of each subtree...) before or after its children? Or, perhaps, we process along levels that are the same distance from the root?
Traversals

- Linear
  - *For loop* or *iterator*
- Binary Tree
  - *Preorder*
  - *Inorder*
  - *Postorder*
  - *Levelorder*
inorder

- Visit the left child
- Then visit the parent
- Visit the right child
preorder

- Visit the parent first
- Visit the left child
- Visit the right child
postorder

- Visit the left child
- Visit the right right child
- Visit the parent
levelorder

- Visit nodes at each level in left to right order
Definition

Add ordering conditions to a binary tree:

- Data are comparable
- Data in left subtree are less than node.data
- Data in right subtree are more than node.data
Why Binary Search Trees?

Searches that are directed along a single path are efficient:

- A BST with 1 node has height 1
- A BST with 3 nodes may have height 2
- A BST with 7 nodes may have height 3
- A BST with 15 nodes may have height 4
- A BST with \( n \) nodes may have height \([\log n]\)
BST_contains

If node is the root of a “balanced” BST, then we can check whether an element is present in about $\log n$ node accesses.

def bst_contains(node, value):
    """
    Return whether tree rooted at node contains value.
    Assume node is the root of a Binary Search Tree
    @param BinaryTree|None node: node of a Binary Search Tree
    @param object value: value to search for
    @rtype: bool
    >>> bst_contains(None, 5)
    False
    <<< bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(3))
    True
    """