Binary Trees Traversal, BST

Arnamoy Bhattacharyya
Agenda

1. Arithmetic expression tree *parenthesize()*
2. Binary Tree Traversal
3. Binary Search Trees
Parenthesizing expression trees

```python
def parenthesize(b: BinaryTree) -> str:
    """
    Parenthesize the expression rooted at b. If b is a leaf, return its float value. Otherwise, parenthesize b.left and b.right and combine them with b.value.
    """
    Assume:  
        — b is a non-empty binary tree
        — interior nodes contain value in {"+", "−", "∗", "/"}
        — interior nodes always have two children
        — leaves contain float value

@param BinaryTree b: binary tree representing arithmetic expression
@paramtype: str

>>> b = BinaryTree(3.0)
>>> print(parenthesize(b))
3.0

>>> b = BinaryTree("+", BinaryTree("∗", BinaryTree(3.0), BinaryTree(4.0)), BinaryTree(7.0))
>>> print(parenthesize(b))
((3.0 * 4.0) + 7.0)
"""
Binary Tree: Inorder Traversal

- A recursive definition:
  - visit the left subtree inorder
  - visit this node itself
  - visit the right subtree inorder
- The code is almost identical to the definition
Exercise: Inorder Traversal
Implementing inorder

```python
def inorder_visit(b, act):
    
    Visit each node of binary tree rooted at root in order and act.

@param BinaryTree|None root: binary tree to visit
@param (BinaryTree)->object act: function to execute on visit
@return: None

>>> def f(node): print(node.value)
>>> b = None
>>> inorder_visit(b, f) is None
True
>>> b = BinaryTree("+", BinaryTree("*", BinaryTree(3.0), BinaryTree(4.0)), BinaryTree(7.0))
>>> inorder_visit(b, f)
3.0
  *
  4.0
+
  7.0
  +
```
Implementing pre and post order
When to use pre, post and in-order

Post-order: Deleting a Binary tree

In-order: For generating human understandable equation from expression tree

Pre-order: While duplicating a binary tree
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
Find a value in a regular Binary Tree

How many nodes do we visit to find out the following:

- Find value 5, if present...
- Find value 13, if present...
- Find value 12, if present...
Find a value in a BST

How many nodes do we visit (say, in preorder) to find out the following:

- Find value 5, if present...
- Find value 13, if present...
- Find value 12, if present...
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 \( \log_2 n \)
  - \( 1,000,000 \) nodes \( \Rightarrow \) height < 20!

If the BST is “balanced”, then we can check whether an element is present in about \( \log n \) node accesses
Demo