CSC148 Lab#6, winter 2015

learning goals

In this lab you will continue implementing recursive functions. Unlike **lab05**, the input will now be a **BTNode**, the root of a binary tree, as presented in lecture. You may want to review lecture materials.

You should work on these on your own before Thursday, and you are encouraged to then go to your lab where you can get guidance and feedback from your TA . There will be a short quiz during the last 15 minutes of the lab, based on these exercises.

set-up

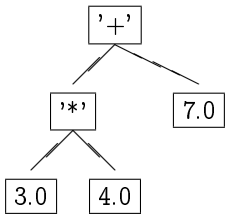
Open file bt.py in Wing and save it under a new sub-directory called **lab06**. This file provides you with a declaration of class **BTNode**, headers and docstrings for the functions you will implement, as well as implementations for functions **insert**, which inserts values into a binary search tree, and **is_leaf**, which reports whether a node in a binary tree is a leaf or not.

Once you have familiarized yourself with the **__init__** and **__str__** methods for class **BTNode**, you are ready to proceed to the implementation of the functions below. If you have questions, call your TA over.

Since you are learning recursion, you are **not** allowed to use calls to functions such as **vist_inorder** that we have implemented in class. You should implement your recursive code from scratch.

implement parenthesize

This function will take in an expression tree such as:



. . . and produce the corresponding parenthesized expression:

$$((3.0 * 4.0) + 7.0)$$

The spaces between operators and operands is omitted.

Read over the header and docstrings in **bt.py**, but **don't** fill in any implementation until you have answered the questions below, and shown your answers to your TA:

1. Give an example of a **BTNode** representing an expression that is so simple as not to require recursion. Write the Python code to detect such cases, and return the appropriate value.

2. Give a "typical" example of a **BTNode** representing an expression that is less simple and may be solved with recursion. Write the Python code to detect such cases, and return the appropriate value.

Now implement **parenthesize** and satisfy yourself that it works properly.

implement list_longeset_path

This function will list the data in a longest path of a tree rooted at a **BTNode**. For example, in the tree rooted at **t** from the previous function, a longest path might be **['+', '*', 4.0]**.

Before implementing the function, answer the same sort of questions as before and show them to your TA:

1. Give an example of a **BTNode** representing an expression that is so simple as not to require recursion. Be sure to consider empty trees! Write the Python code to detect such cases, and return the appropriate value.
2. Give a "typical" example of a **BTNode** representing an expression that is less simple and may be solved with recursion. Write the Python code to detect such cases, and return the appropriate value.

Now implement **list_longest_path** and satisfy yourself that it works properly.

binary search trees

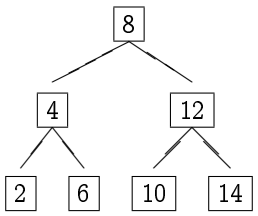
The remaining functions are **binary search trees**, a special case of binary trees. Recall that a binary search tree is a binary tree with the following conditions:

1. The data are comparable.
2. All data in the left subtree are less than this node's data.
3. All data in the right subtree are more than this node's data.

implement list_between

This function will take a binary search tree and return a list of its data between **start** and **end**, inclusive. You must do this as efficiently as possible, that is your code must not visit unecessary nodes.

Suppose **t** was the **BTNode** at the root of this tree:



. . . then **list_between(t, 5, 9)** should return **[6, 8]**, and the code should not visit the nodes with data 2 or 14 at all.

Read over the header and docstrings in **bt.py**, but **don't** fill in any implementation until you have answered the questions below, and shown your answers to your TA:

1. Give an example of a **BTNode** representing an expression that is so simple as not to require recursion. Be sure to consider the case of an empty tree. Write the Python code to detect such cases, and return the appropriate value.
2. Give a "typical" example of a **BTNode** representing an expression that is less simple and may be solved with recursion. Write the Python code to detect such cases, and return the appropriate value.

Now implement **list_between** and satisfy yourself that it works properly.

list_internal_between

This function takes a binary search tree rooted at a **BTNode** and returns a list of data between **start** and **end** (inclusive) that are in internal nodes.

You may **not** use a call to **list_between** in your implementation — the idea is to write your recursive code from scratch.

Read over the header and docstrings in **bt.py**, but **don't** fill in any implementation until you have answered the questions below, and shown your answers to your TA:

1. Give an example of a **BTNode** representing an expression that is so simple as not to require recursion. Be sure to consider the case of an empty tree! Write the Python code to detect such cases, and return the appropriate value.

2. Give a "typical" example of a **BTNode** representing an expression that is less simple and may be solved with recursion. Write the Python code to detect such cases, and return the appropriate value.

Now implement **list_internal_between** and satisfy yourself that it works properly.

additional exercises

Here are some additional exercises you should work on if you finish those above. As usual, at the end of the lab, there will be a 15-minute quiz.