CS234 S17 - Assignment 3
Coverage: Modules 4-6

This assignment consists of a written component and a programming component. Please read the course website carefully to ensure that you submit each component correctly.

Several of the questions will make use of a variant of ADT Binary Tree. In the list of operations below, $B$ is a binary tree and $node$ is a node. The list is not intended to be comprehensive, but should be sufficient for you to complete the exercises.

<table>
<thead>
<tr>
<th>Name</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateBT()</td>
<td>a new empty binary tree</td>
</tr>
<tr>
<td>IsEmptyBT($B$)</td>
<td>true if empty, else false</td>
</tr>
<tr>
<td>RootBT($B$)</td>
<td>the root node of $B$, if any</td>
</tr>
<tr>
<td>KeyAtNode($B$, $node$)</td>
<td>value of key at node</td>
</tr>
<tr>
<td>ParentBT($B$, $node$)</td>
<td>the parent of node, else None</td>
</tr>
<tr>
<td>LeftChildBT($B$, $node$)</td>
<td>the left child of node, else None</td>
</tr>
<tr>
<td>RightChildBT($B$, $node$)</td>
<td>the right child of node, else None</td>
</tr>
</tbody>
</table>

(ADT Binary Tree is credited to Naomi Nishimura, with minor modifications.)

Written component
For full marks, the implementation you describe must be reasonably efficient (in particular, not doing work that is obviously unnecessary).

W1. [4 marks] Using the ADT Binary Tree, write pseudocode for a function Cousins that consumes two nodes and returns true if they are cousins (i.e., share the same grandparent, but not the same parent), false otherwise.

W2. [6 marks] Using the ADT Binary Tree, write pseudocode for a recursive function Shortest that consumes a node in a binary tree, and determines (and returns) the length of the shortest path from that node downward to a leaf. The length of the path should be the number of nodes, including the beginning and ending nodes of the path.

W3. [10 marks] Using the ADT Binary Tree, write pseudocode for a recursive function PerfectBST that takes as input a node in a binary tree. Your function should return:
   • If the (sub)tree rooted at the node is a perfect binary search tree, a tuple $(min, max, h)$ where $min$ and $max$ are the lowest and highest keys in the tree respectively, and $h$ is the height of the tree
   • If the (sub)tree rooted at the node is not a perfect binary search tree, returns None.
Here, as in class, we will consider 'height' to be the number of levels.
Programming component
Please read the course website carefully to ensure that you are using the correct version of Python and the correct style.

P1. [20 marks] For this question, you will implement a variant of Ordered Tree. This tree will store only values (not key-value pairs); the ordering will be based on the operations performed on the tree, not the values themselves.

The interface for the OTree class is provided in the file otree.py, which accompanies this assignment. Implement each of the functions that do not have implementations (i.e., those with 'pass' as bodies); a handful of functions have been implemented for you (e.g., __str__( ), to help you in your work.

The node class, OTNode, has been provided for you, in the file otnode.py. You are required to use this class; do not alter the file in any way. OTNodes store the following:
- The value for the node;
- A list of the children of that node (where the ordering of the list is the ordering of the children of the node);
- A link to the parent of the node (which, if the OTNode is the root of the tree, has the value None).

The ordered tree (OTree) contains a link to the root node (which is None, if the tree is empty).

It is suggested that you implement the add_node() function first (and start early!), so that you can build trees with which to implement/test other functions. The delete function is the most difficult. Hint: Be careful managing child lists and parent links!

Three ‘string’ functions have been provided for you in the OTNode class. __str__ ( ) simply prints the value of the node. detailed_string() prints the node’s value, as well as its parent and children. tree_string() prints the structure of the entire subtree rooted at the node, where each node’s value is followed by a list of its children, enclosed in square brackets. (If the children have children of their own, and so on, then you will see nested brackets.)

An example of an interactive session, making use of OTree operations:
```python
>>> t = OTree()
>>> a = t.add_node(None, 'a')
>>> t
a
>>> t.add_node(a, 'b')
b
>>> c = t.add_node(a, 'c')
>>> t
```

a[b,c]

>>> d = t.add_node(c, 'd')
>>> t
a[b,c[d]]

>>> a.detailed_string()
'a Parent:None Children:b,c'

>>> c.detailed_string()
'c Parent:a Children:d'

>>> t.add_node(c, 'e')
e

>>> t.add_node(c, 'f')
f

>>> t
a[b,c[d,e,f]]

>>> t.delete_node(c)

>>> t
a[b,d[e,f]]

>>> t.find('e')
e

>>> t.find('j')