Outline
linked lists, conceptually

- **data:** Sequence of nodes, each with a value and reference to next (successor) node. List has reference to front (aka head) node.

- **operations:** insert(node), find(value), …
class LListNode:
    '''Linked List node that can reference next node.'''
    def __init__(self, value=None, nxt=None):
        '''Create a LListNode with value and reference to next LListNode'''
        self.value, self.nxt = value, nxt
    def __repr__(self):
        '''Represent this node as a string.'''
        return 'LListNode(' + str(self.value) + ', ' + str(self.nxt) + ')'
prefer repr to str

Two special methods for representing an object. If you omit `__str__`, Python will use `__repr__`. By convention, the latter should be able to produce an equivalent object.
an alternative Stack

Use LListNode to re-implement Stack, and compare performance.
### Histogram for Test #1

<table>
<thead>
<tr>
<th></th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>out of</td>
<td>10.0</td>
<td>15.0</td>
<td>15.0</td>
<td>45.0</td>
</tr>
<tr>
<td>average%</td>
<td>74.9</td>
<td>69.9</td>
<td>69.6</td>
<td>71.5</td>
</tr>
<tr>
<td>excl. dr.</td>
<td>74.9</td>
<td>69.9</td>
<td>69.6</td>
<td>71.5</td>
</tr>
<tr>
<td>&gt;=100%</td>
<td>84</td>
<td>34</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>90..&lt;100%</td>
<td>1</td>
<td>26</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>80..&lt;90%</td>
<td>5</td>
<td>50</td>
<td>58</td>
<td>26</td>
</tr>
<tr>
<td>70..&lt;80%</td>
<td>13</td>
<td>12</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>60..&lt;70%</td>
<td>17</td>
<td>30</td>
<td>43</td>
<td>34</td>
</tr>
<tr>
<td>50..&lt;60%</td>
<td>12</td>
<td>5</td>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>40..&lt;50%</td>
<td>35</td>
<td>14</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>35..&lt;40%</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>30..&lt;35%</td>
<td>14</td>
<td>7</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>20..&lt;30%</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>10..&lt;20%</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1..&lt;10%</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt; 1%</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
a linked list is a special case of a tree, with arity (AKA branching factor) of 1. Here's a more general tree node:

class TreeNode:
    """Node with children."""
    def __init__(self: 'TreeNode',
                 value: object =None, children: list =None):
        """Node with any number of children"""
        self.value = value
        if not children:
            self.children = []
        else:
            self.children = children[:] # quick-n-dirty copy of list
    def __repr__(self: 'TreeNode'):
        """Represent this TreeNode as a string"""
        return ('TreeNode(' + str(self.value) + ', ' +
                 repr(self.children) + ')')
In order to keep track of the root, and perhaps some methods, we need a Tree class:

class Tree:
    """Bare-bones Tree ADT"""
    
    def __init__(self, root=None):
        """Create a new tree rooted at root."""

        self.root = root  

        # allows for an empty tree
try to implement:

```python
Use helpers defined as nested functions.

arity(self: 'Tree') -> int:
    """Return the maximum branching factor of this tree""

__contains__(self: 'Tree', value: object) -> bool:
    """Return whether this tree has a node with value""
    this allows ~5 in Tree

node_count(self: 'Tree') -> int:
    """Number of nodes in this tree""
```