CSC148 winter 2014
linked structures
week 8

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W14/
416-978-5899

March 4, 2014
Outline
linked lists, two concepts

There are two useful, but different, ways of thinking of linked list structures

1. as lists made up of an item (value) and the remaining list (rest)

2. as objects (nodes) with a value and a reference to other similar objects

```
12  →  99  →  37  →  X
```
a node class

class LListNode:
    """Node to be used in linked list""

    def __init__(self: 'LListNode', value: object, 
                 nxt: 'LListNode' = None) -> None:
        """Create a new LListNode containing value 
        referring to next node nxt

        nxt --- None if and only if we are on the last node 
        value --- always a Python object, there are no empty nodes 
        """

        self.value, self.nxt = value, nxt
a wrapper class for list

The list class keeps track of information about the entire list — such as its front.

class LinkedList:
    """Collection of LListNodes""

def __init__(self: 'LinkedList') -> None:
    """Create an empty LinkedList""
    self.front = None
    self.size = 0
def insert(self: 'LinkedList', value: object) -> None:
    """Insert LListNode with value at front of self

    >>> lnk = LinkedList()
    >>> lnk.insert(0)
    >>> lnk.insert(1)
    >>> lnk.insert(2)
    >>> str(lnk.front)
    '2 -> 1 -> 0 -> None'
    >>> lnk.size
    3
    """
deletion

"""Delete front LListNode from self

self must not be None

>>> lnk = LinkedList()
>>> lnk.insert(0)
>>> lnk.insert(1)
>>> lnk.insert(2)
>>> lnk.delete_front()
>>> str(lnk.front)
'1 -> 0 -> None'
>>> lnk.size
2
"""
def reverse(ln: LListNode) -> LListNode:
    """Return the linked list starting
    at ln in reverse order

    ln is not None

    >>> ln = LListNode(0)
    >>> ln1 = LListNode(1, ln)
    >>> ln2 = LListNode(2, ln1)
    >>> ln3 = LListNode(3, ln2)
    >>> lnr = reverse(ln3)
    >>> str(lnr)
    '0 -> 1 -> 2 -> 3 -> None'
    """
Instead of single tree class, separate node and bst classes:

class BTNode:
    """Binary Tree node."""

    def __init__(self: 'BTNode', data: object, 
        left: 'BTNode'=None, 
        right: 'BTNode'=None) -> None:
        """Create BT node with data, children left and right."""
        self.data, self.left, self.right = data, left, right
Python `str` method is more informal than `repr`. I had to start with a helper function (why?)

```python
def _str(b: BTNode, i: str) -> str:
    """Return a string representing self inorder indent by i""
    return ((bt_str(b.right, i + ' ') if b.left else '') + i + str(b.data) + '\n' +
             (bt_str(b.left, i + ' ') if b.right else ''))
```
...now the __str__ method is easy

def __str__(self: 'BTNode') -> str:
    """Return a string representing self inorder""
    return _str(self, '')
Add a condition: data in left subtree is less than that in the root, which in turn is less than that in right subtree. Now search is more efficient...

class BST:
    """Binary search tree."""

    def __init__(self: 'BST', root: BTNode=None) -> None:
        """Create BST with BTNode root."""
        self._root = root
Careful reading of the example show that we expect `insert` to ensure this is a binary search tree:

```python
def insert(self: 'BST', data: object) -> None:
    """Insert data, if necessary, into this tree.

>>> b = BST()
>>> b.insert(8)
>>> b.insert(4)
>>> b.insert(2)
>>> b.insert(6)
>>> b.insert(12)
>>> b.insert(14)
>>> b.insert(10)

BST(BTNode(8, BTNode(4, BTNode(2, None, None), BTNode(6, None, None)),
            BTNode(12, BTNode(10, None, None), BTNode(14, None, None))))
"""
    self._root = _insert(self._root, data)
```
the wrapper/node design means that the recursive structures are **BTNodes** rather than **BST**, so write a module-level function as a helper:

```python
def _insert(node: BTNode, data: object) -> BTNode:
    """Insert data starting at node, and return root."""
    return_node = node
    if not node:
        return_node = BTNode(data)
    elif data < node.data:
        node.left = _insert(node.left, data)
    elif data > node.data:
        node.right = _insert(node.right, data)
    else:  # nothing to do
        pass
    return return_node
```