CSC148 winter 2014

linked structures

week 8

Danny Heap / Dustin Wehr
heap@cs.toronto.edu / dustin.wehr@utoronto.ca
BA4270 / SF4306D
http://www.cdf.toronto.edu/~heap/148/F13/

March 5, 2014
Outline

LinkedList again

linked list node class with wrapper class

binary search trees
Start by designing a class hierarchy. What information is needed for each type of regular expression tree? What information is specialized? What’s general? Look at Week 6’s Tree class in tree.py for ideas.
Alternative initializer for LinkedList class

Included in linked_list.py

class AltLinkedList:
    def __init__(self,*args):
        if len(args) == 0:
            self.empty = True
        elif len(args) == 1:
            self.empty = False
            self.head = args[0]
            self.rest = AltLinkedList()
        elif len(args) == 2:
            self.empty = False
            self.head = args[0]
            assert isinstance(args[1], AltLinkedList)
            self.rest = args[1]
        else: # len(args) > 2:
            raise Exception("AltLinkedList initializer takes ''
                "between 0 and 2 arguments.")
design choices

**AltLinkedList** initialization reveals design choices

- **AltLinkedList()** creates an empty list
- **AltLinkedList(obj)** creates a 1-element list with **head** = **obj**
- **AltLinkedList(obj, otherlist)** creates a linked list with **head** = **obj** and **rest** = **otherlist**.

- It’s possible for **head** to refer to None — why might you want this?

- **rest** refers to another **AltLinkedList** with the same structure

This isn’t the only design for a linked list, for example How to think like a computer scientist shows the “wrapper” approach.
implement `prepend(head)`

main goals are to preserve the list identity (same id) and preserve the previous contents

- start the rest of the list with the current attributes (shallow copy them)

- change the current head to the one passed in

- change the current rest to the copy!

Try drawing the result of `prepend(5)`

```
12 → 99 → 37 → [insert new node here]
```
def prepend(self: 'LinkedList', head: object) -> None:
    old_head = copy(self)
    self.rest = old_head
    self.head = head
    self.empty = False
what’s wrong with this implementation of prepend(head)?

def prepend(self: 'LinkedList', head: object) -> None:
    new_head = LinkedList(head, self)
    self = new_head
    self.empty = False
what about this `prepend(head)`?

It’s usage is different; if you want to add 9 to LinkedList x you do:

```python
x = x.prepend(9)
```

```python
def prepend(self: 'LinkedList', head: object) -> 'LinkedList':
    new_head = LinkedList(head, self)
    return new_head
```
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
```
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
    """
"""Delete front LLListNode 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_links(ln: LListNode) -> LListNode:
    """Reverse the (virtual) linked list that starts at ln, by modifying the next attributes of all the LListNodes reachable from ln, and return the front node of the (virtual) reversed linked list.

ln is not None

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

```python
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\_ methods are more informal than \_repr\_ methods. I had to start with a helper function (why?)

def \_str\_(b: BTNode, i: str) -> str:
    """Return a string representing self inorder
    indent by i""
    return (_str(b.right, i + '  ') if b.right else '') +
        i + str(b.data) + '\n' +
        (_str(b.left, i + '  ') if b.left else '')

...now the \texttt{\_str\_} method is easy

def \texttt{\_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
insert must obey condition

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)
>>> b
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)
```
helper function...

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
```
def _find(node: BTNode, data: object):
    """Return the node containing data, or else None."""
    if node is None or node.data == data:
        return node
    if data < node.data:
        return _find(node.left, data)
    else:
        return _find(node.right, data)