t2a 1-1

CSC148 term test #2, L0101/L0301

March 2017

Last name ....

First name ....

Utorid ....

U of T Email ....

Three questions:

Q1:	/ 5
Q2:	/ 6
Q3:	/ 9

t2a 1-2

1. **(5 pts)** Consider a post-order traversal, i.e. visiting each node in post-order, of a **Binary Search Tree**, with the act function defined as;

```
 def act(node):
     print(node.value, end=' ') # prints all on the same line
```

Recall that for a BST, a postorder traversal first visits the left subtree in postorder, then the right subtree in postorder, and finally the root.

Draw a representation of a Binary Search Tree that would produce the following output when traversed as described above.

```
15 10 25 20 50 45 33
```

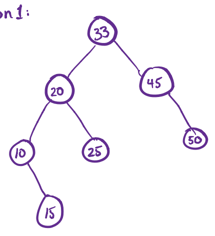
Hint: Recall that both postorder traversals and the Binary Search Tree property are defined recursively, so that they apply at the root of the main tree and of each subtree...

sample solution: Below are solutions for all three test versions (9/10 a.m., 1 p.m., 6 p.m.)

General question:
Given a post-order traversal of a BST, draw the tree.
Hints: give definition of post order

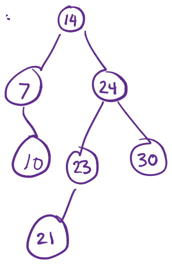
each tree should have 7 nodes, with no left leaf of leftmost interior node.

Version 1:



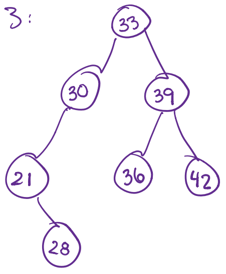
15 10 25 20 50 45 33

Version 2:



10 7 21 23 30 24 14

Version 3:



28 21 30 36 42 39 33

Use the recursive definitions of both postorder traversals and the BST property to help.

t2a 1-4

2. **(6 pts)** Read the definition of class **Tree** below, from module **tree**. **Notice** that a Tree's children is a list of 0 or more Tree objects, and does not contain any None objects. Also, the only functions or methods you may rely are on are below.

   Implement function **count_at_depth** on the next page. You may implement helper functions, if you wish, or implement it as one function.

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.
    """

    def __init__(self, value=None, children=None):
        """
        Create Tree self with content value and 0 or more children

        @param Tree self: this tree
        @param object value: value contained in this tree
        @param list[Tree] children: possibly-empty list of children
        @rtype: None
        """
        self.value = value
        # copy children if not None
        self.children = children.copy() if children is not None else []


    def __str__(self, indent=0):
        """
        Produce a user-friendly string representation of Tree self,
        indenting each level as a visual clue.

        @param Tree self: this tree
        @param int indent: amount to indent each level of tree
        @rtype: str

        >>> t = Tree(17)
        >>> print(t)
        17
        >>> t1 = Tree(19, [t, Tree(23)])
        >>> print(t1)
        19
           17
           23
        >>> t3 = Tree(29, [Tree(31), t1])
        >>> print(t3)
        29
           31
           19
              17
              23
        """
        root_str = indent * " " + str(self.value)
        return '\n'.join([root_str] +
                         [c.__str__(indent + 3) for c in self.children])
```

```
from tree import Tree


def count_at_depth(t, d):
    """ Return the number of nodes at depth d of t.

    @param Tree t: tree to explore --- cannot be None
    @param int d: depth to report from, non-negative
    @rtype: int

    >>> t = Tree(17, [Tree(0), Tree(1, [Tree(4)]), Tree(2, [Tree(5)]), Tree(3)])
    >>> print(t)
    17
       0
       1
          4
       2
          5
       3
    >>> count_at_depth(t, 0)
    1
    >>> count_at_depth(t, 1)
    4
    >>> count_at_depth(t, 2)
    2
    >>> count_at_depth(t, 5)
    0
    """
```

Hint: Any node that is at depth **d** from **t** is at depth **d-1** from **t**'s children.

sample solution(s): All three versions below.

```
        if d < 0:
            return 0
        elif d == 0:
            return 1
        else:
            return sum([count_at_depth(c, d - 1)
                        for c in t.children
                        if c is not None])


    def sum_at_depth(t, d):
        """ Return the sum of node values at depth d of t.

        Assume that node values are integers and that there are no
        None values in any list of children in t or its descendants.

        @param Tree t: tree to explore, cannot be None
```

t2a 1-6

```
    @param int d: depth to report from, non-negative
    @rtype: int

    >>> t = Tree(17, [Tree(0), Tree(1, [Tree(4)]), Tree(2, [Tree(5)]), Tree(3)])
    >>> print(t)
    17
       0
       1
          4
       2
          5
       3
    >>> sum_at_depth(t, 0)
    17
    >>> sum_at_depth(t, 1)
    6
    >>> sum_at_depth(t, 2)
    9
    >>> count_at_depth(t, 5)
    0
    """
    if d == 0:
        return t.value
    else:
        return sum([sum_at_depth(c, d - 1)
                    for c in t.children
                    if c is not None])


def concatenate_at_depth(t, d):
    """ Return the concatenation of node values at depth d of t.

    Assume that node values are strings and that there are no
    None values in any list of children in t or its descendants.

    @param Tree t: tree to explore, cannot be None
    @param int d: depth to report from, non-negative
    @rtype: str

    >>> t = Tree("a", [Tree("b"), Tree("c", [Tree("d")]), Tree("e", [Tree("f")]), Tree("g")])
    >>> print(t)
    a
       b
       c
          d
       e
          f
       g
    >>> concatenate_at_depth(t, 0)
    'a'
    >>> concatenate_at_depth(t, 1)
    'bceg'
    >>> concatenate_at_depth(t, 2)
    'df'
    >>> concatenate_at_depth(t, 5)
    ''
    """
    if d == 0:
```

```
    return t.value
else:
    return "".join([concatenate_at_depth(c, d - 1)
                    for c in t.children
                    if c is not None])
```

t2a 1-8

3. **(9 pts)** Read the declaration of the **LinkedList** and **LinkedListNode** classes below, from module **node**. **Notice** that we use **property** and **_get_value** to make sure the values of these LinkedListNodes are immutable: they cannot be changed after initialization!

On page 7 implement the function **reverse_list**. You may create new local names (variables) to refer to existing nodes (if you need to), **but** you may **not** create any new objects (LinkedLists, LinkedListNodes, or Python lists, etc.).

```
class LinkedListNode:
    """
    Node to be used in linked list
    === Attributes ===
    @param LinkedListNode next_: successor to this LinkedListNode
    @param object value: data this LinkedListNode represents
    """
    def __init__(self, value, next_=None):
        """
        Create LinkedListNode self with data value and successor next_.

        @param LinkedListNode self: this LinkedListNode
        @param object value: data of this linked list node
        @param LinkedListNode|None next_: successor to this LinkedListNode.
        @rtype: None
        """
        self._value, self.next_ = value, next_

    def _get_value(self):
        # to show value
        return self._value
    # no way to set value!
    value = property(_get_value)

    def __str__(self):
        """
        Return a user-friendly representation of this LinkedListNode.

        @param LinkedListNode self: this LinkedListNode
        @rtype: str

        >>> n = LinkedListNode(5, LinkedListNode(7))
        >>> print(n)
        5 -> 7 ->|
        """
        s = "{} ->".format(self.value)
        current_node = self.next_
        while current_node is not None:
            s += " {} ->".format(current_node.value)
            current_node = current_node.next_
        assert current_node is None, "unexpected non_None!!!"
        s += "|"
        return s
```

```
class LinkedList:
    """
    Collection of LinkedListNodes
    === Attributes ==
    @param: LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList
                        a non-negative integer
    """
    def __init__(self):
        """
        Create an empty linked list.

        @param LinkedList self: this LinkedList
        @rtype: None
        """
        self.front, self.back, self.size = None, None, 0

    def __str__(self):
        """
        Return a human-friendly string representation of
        LinkedList self.

        @param LinkedList self: this LinkedList

        >>> lnk = LinkedList()
        >>> print(lnk)
        I'm so empty...
        """
        if self.front is None:
            assert self.back is None and self.size is 0, "ooooops!"
            return "I'm so empty..."
        else:
            return str(self.front)

    def prepend(self, value):
        """
        Insert value before LinkedList self.front.

        @param LinkedList self: this LinkedList
        @param object value: value for new LinkedList.front
        @rtype: None

        >>> lnk = LinkedList()
        >>> lnk.prepend(0)
        >>> lnk.prepend(1)
        >>> lnk.prepend(2)
        >>> str(lnk.front)
        '2 -> 1 -> 0 ->|'
        >>> lnk.size
        3
        """
        new_node = LinkedListNode(value, self.front)
        self.front = new_node
        if self.size == 0:
            self.back = new_node
        self.size += 1
```

t2a 1-10

```
from node import LinkedList, LinkedListNode


def reverse_list(list_):
    """ Reverse the order of the nodes in list_.

    @param list_ LinkedList: linked list to modify
    @rtype: None

    >>> lnk = LinkedList()
    >>> lnk.prepend(1)
    >>> lnk.prepend(3)
    >>> lnk.prepend(5)
    >>> print(lnk)
    5 -> 3 -> 1 ->|
    >>> reverse_list(lnk)
    >>> print(lnk)
    1 -> 3 -> 5 ->|
    """

    # Hint: draw pictures.
```

sample solution(s): All three versions below...

```
def reverse_list(list_, arg=None):
    current = list_.front
    prev = None
    tail = list_.front
    while current:
        next_ = current.next_
        current.next_ = prev
        prev = current
        current = next_
        # Or in one line:
        # current.next_, current, prev = prev, current.next_, current

    list_.front, list_.back = prev, tail


def reverse_list_to_value(list_, value):
    """ Does not update size, discards other nodes. """
    current = list_.front
    prev = None
    tail = list_.front

    while current and (prev is None or prev.value != value):
        next_ = current.next_
        current.next_ = prev
        prev = current
        current = next_
        # Or in one line:
        # current.next_, current, prev = prev, current.next_, current
    list_.front, list_.back = prev, tail


def reverse_list_after_value(list_, value):
    """ Does not update size, discards other nodes. """
    current = list_.front

    while current and current.value != value:
        current = current.next_

    if current:
        prev = None
        tail = current
        while current:
            next_ = current.next_
            current.next_ = prev
            prev = current
            current = next_
            # Or in one line:
            # current.next_, current, prev = prev, current.next_, current

        list_.front, list_.back = prev, tail
```

This page is available for answers that don't fit elsewhere.

t2a 1-12