CSC148 winter 2017

binary trees

week 8

Danny Heap
heap@cs.toronto.edu / BA4270 (behind elevators)
http://www.teach.cs.toronto.edu/~csc148h/winter/
416-978-5899

March 8, 2017

Outline

general trees continued...

binary trees

traversals

binary *search* trees

traversal

The functions and methods we have seen get information from every node of the tree — in some sense they traverse the tree.

Sometimes the **order** of processing tree nodes is important: do we process the root of the tree (and the root of each subtree...) **before** or **after** its children? Or, perhaps, we process along **levels** that are the same distance from the root?

pre-order visit

```
def preorder_visit(t, act):
    """
    Visit each node of Tree t in preorder, and act on the nodes
    as they are visited.

    @param Tree t: tree to visit in preorder
    @param (Tree)->Any act: function to carry out on visited Tree node
    @rtype: None
```

assume act is defined on all nodes of t

```
    >>> t = descendants_from_list(Tree(0),
                                  [1, 2, 3, 4, 5, 6, 7], 3)
    >>> def act(node): print(node.value)
    >>> preorder_visit(t, act)
    0
    1
    4
    5
    6
    2
    7
    3
```

try tracing this

postorder

```
def postorder_visit(t, act):
    """
    Visit each node of t in postorder, and act on it when it is visited

    @param Tree t: tree to be visited in postorder
    @param (Tree)->Any act: function to do to each node
    @rtype: None

    >>> t = descendants_from_list(Tree(0),
                                  [1, 2, 3, 4, 5, 6, 7], 3)
    >>> def act(node): print(node.value)
    >>> postorder_visit(t, act)
    4
    5
    6
    1
    7
    2
    3
    0
    """
```

levelorder

```
def levelorder_visit(t, act):
    """
    Visit every node in Tree t in level order and act on the node
    as you visit it.

    @param Tree t: tree to visit in level order
    @param (Tree)->Any act: function to execute during visit

    >>> t = descendants_from_list(Tree(0),
                                  [1, 2, 3, 4, 5, 6, 7], 3)
    >>> def act(node): print(node.value)
    >>> levelorder_visit(t, act)
    0
    1
    2
    3
    4
    5
    6
    7
    """
```

queues, stacks, recursion

You may have noticed in the last slide there were no recursive calls, and a **queue** was used to process a recursive structure in level order.

Careful use of a **stack** allows you to process a tree in preorder.

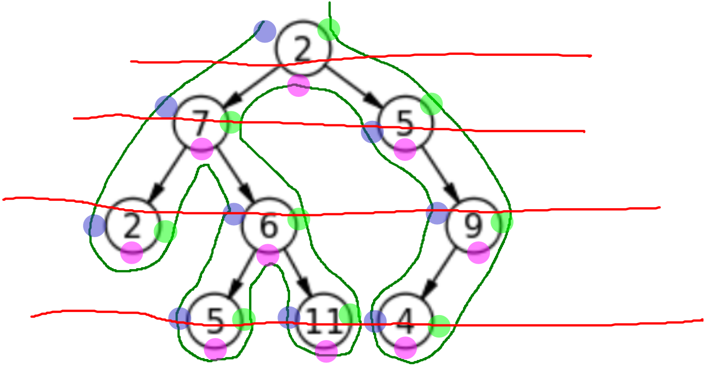
traversal tracing...

pre-order

postorder

levelorder

inorder



refactor

- make flatten and height methods

  choose a method over a module-level function when it is closely associated with each instance of a class

- tweak __str__ and __repr__ so that they'll work better for binary trees

- subclass Tree to BinaryTree

  began this, but see next slide...

tree inheritance issues

see Liskov Substitution Principle

one approach to **BinaryTree** would be to make it a subclass of **Tree**, but there are some design considerations

- any client code that uses **Tree** would be required not to violate the branching factor (2) of **BinaryTree**

  e.g. t.children.append(...) would be a no-no

- one way to achieve this would be to make **Tree** immutable: make sure there is no way to change **children** or **value**, and then have subclasses that might be mutable

we will take a different approach: a completely separate **BinaryTree** class

BinaryTree

Change our generic Tree design so that we have two named children, left and right, and can represent an empty tree with None

```
class BinaryTree:
    """
    A Binary Tree, i.e. arity 2.
    """

    def __init__(self, value, left=None, right=None):
        """
        Create BinaryTree self with value and children left and right.

        @param BinaryTree self: this binary tree
        @param object value: value of this node
        @param BinaryTree|None left: left child
        @param BinaryTree|None right: right child
        @rtype: None
        """
        self.value, self.left, self.right = value, left, right
```

special methods...

We'll want the standard special methods:

- `__eq__` done for you
- `__str__`
- `__repr__`

contains

you've implemented contains on linked lists, nested Python lists, general Trees before; implement this function, then modify it to become a method

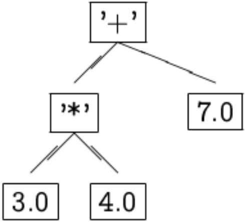
```
def contains(node, value):
    """
    Return whether tree rooted at node contains value.

    @param BinaryTree|None node: binary tree to search for value
    @param object value: value to search for
    @rtype: bool

    >>> contains(None, 5)
    False
    >>> contains(BinaryTree(5, BinaryTree(7), BinaryTree(9)), 7)
    True
    """
```

arithmetic expression trees

Binary arithmetic expressions can be represented as binary trees:



evaluating a binary expression tree

- there are no empty expressions
- if it's a leaf, just return the value
- otherwise...
  - evaluate the left tree
  - evaluate the right tree
  - combine left and right with the binary operator

Python built-in eval might be handy.

inorder

done

A **recursive** definition:

- visit the left subtree **inorder**
- visit this node itself
- visit the right subtree **inorder**

The code is almost identical to the definition.

preorder

- visit this node itself
- visit the left subtree in **preorder**
- visit the right subtree in **preorder**

postorder

- visit the left subtree in **postorder**
- visit the rightsubtree in **postorder**
- visit this node itself

level order

- visit root
- visit root's children
- visit root's grandchildren
- visit root's greatgrandchildren
- ...

definition

Add ordering conditions to a binary tree:

- data are comparable
- data in left subtree are less than node.data
- data in right subtree are more than node.data

why binary search trees?

Searchs that are directed along a single path are efficient:

- a BST with 1 one has height 1
- a BST with 3 nodes may have height 2
- a BST with 7 nodes may have height 3
- a BST with 15 nodes may have height 4
- a BST with n nodes may have height $\lceil \lg n \rceil$.

bst_contains

If node is the root of a "balanced" BST, then we can check whether an element is present in about $\lg n$ node accesses.

```
def bst_contains(node, value):
    """
    Return whether tree rooted at node contains value.

    Assume node is the root of a Binary Search Tree

    @param BinaryTree|None node: node of a Binary Search Tree
    @param object value: value to search for
    @rtype: bool

    >>> bst_contains(None, 5)
    False
    >>> bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(9)), 5)
    True
    """
    # use BST property to avoid unnecessary searching
```

mutation: insert

```
def insert(node, data):
    """
    Insert data in BST rooted at node if necessary, and return new root

    Assume node is the root of a Binary Search Tree.

    @param BinaryTree|None node: root of a binary search tree.
    @param object data: data to insert into BST, if necessary.

    >>> b = BinaryTree(8)
    >>> b = insert(b, 4)
    >>> b = insert(b, 2)
    >>> b = insert(b, 6)
    >>> b = insert(b, 12)
    >>> b = insert(b, 14)
    >>> b = insert(b, 10)
    >>> print(b)
            14
        12
            10
    8
```