CSC148 winter 2018

binary trees
week 8

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

April 25, 2018

Computer Science
UNIVERSITY OF TORONTO

Outline

general trees continued...

binary trees

traversals

binary *search* trees

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.

...or even process a tree in postorder using two stacks...

tree inheritance issues

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**
- 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

BTNode

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

```
class BTNode:
    """
    A Binary Tree, i.e. arity 2.
    data: data/value for this node
    left: left child
    right: right child
    """

    def __init__(self, data: object,
                 left: Union["BTNode", None]=None,
                 right: Union["BTNode, None]=None) -> None:
        """
        Create BTNode self with data and children left and right.
        """
        self.data, self.left, self.right = data, left, right
```

to represent empty children...

special methods...

We'll want the standard special methods:

- `__eq__`

  see code on web page...

- `__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

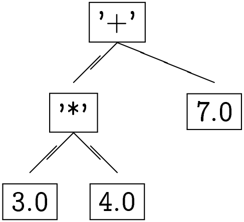
```
def contains(node: BTNode, data: object) -> bool:
    """
    Return whether tree rooted at node contains data.

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

an empty tree never contains data --- that's a base case!

general case: check whether node contains data, or its left child, or its right child.

arithmetic expression trees

Binary arithmetic expressions can be represented as binary trees:



((3.0 * 4.0) + 7.0)

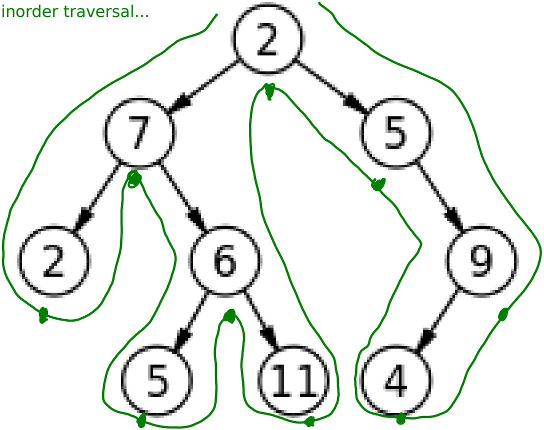
evaluating a binary expression tree

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

Python built-in eval might be handy.

one more diagram...

inorder traversal...



inorder

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.

see our posted code...

preorder

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

see our posted solution...

postorder

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

see our posted solution...

level order

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

use a queue...

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

with these definitions is it possible to have duplicate data? Why or why not?

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 $\lceil \lg n \rceil$. $\lceil \lg (n+1) \rceil$

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: BTNode, data: object) -> bool:
    """
    Return whether tree rooted at node contains data.

    Assume node is the root of a Binary Search Tree

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

see posted code...

mutation: insert

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

    Assume node is the root of a Binary Search Tree.

    >>> b = BTNode(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
            6
        4
            2
```

see posted code...