CSC148 winter 2016

binary trees

week 8

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

March 11, 2016

Outline

binary trees

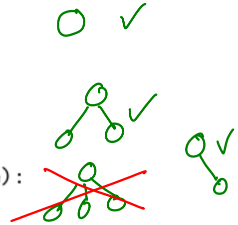
set data structure

traversals

binary *search* trees

Binary Tree

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, data, left=None, right=None):
        """
        Create BinaryTree self with data and children left and right.

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

max children

special methods...

We'll want the standard special methods:

- `__eq__`
- `__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

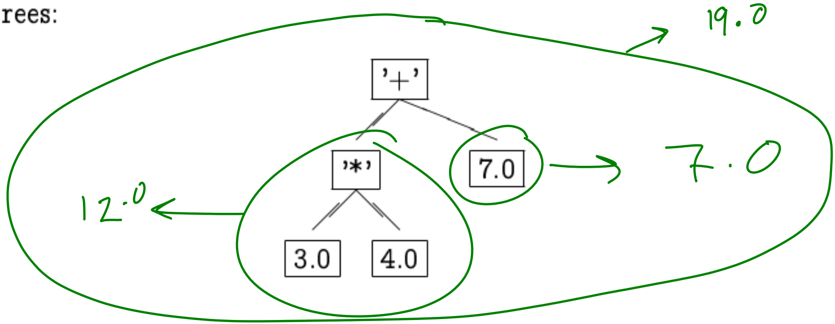
```
def contains(node, value):
    """
    Return whether tree rooted at node contains value.

    @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
    """
```

empty tree → False

otherwise → node.value == value
or contains(node.left, value)
or contains(node.right, value)

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

strings string

use eval

Python built-in eval might be handy.

A2 uses set

some uses:

```
>>> {1, 2, 3} == {2, 3, 1}
True
>>> {1, 2, 3} - {3, 4, 5}
{1, 2}
>>> 2 in {1, 2, 3}
True
>>> {1, 2, 3}.union({3, 4, 5})
{1, 2, 3, 4, 5}
>>> set([1, 2, 3, 2, 3, 4])
{1, 2, 3, 4}
```

— sets ignore element order

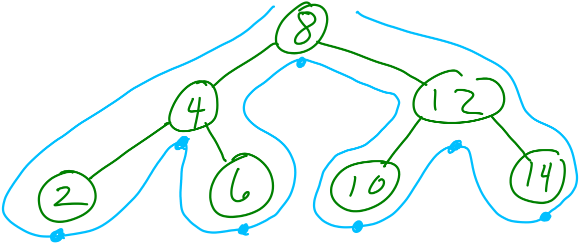
— set difference

— set membership

— set union

) eliminate duplicates

inorder



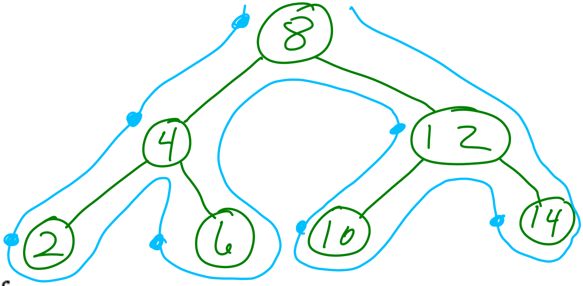
A recursive definition:

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

visit between children

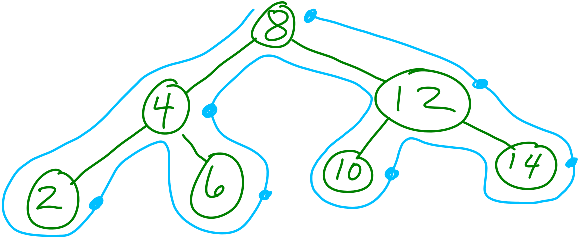
The code is almost identical to the definition.

preorder



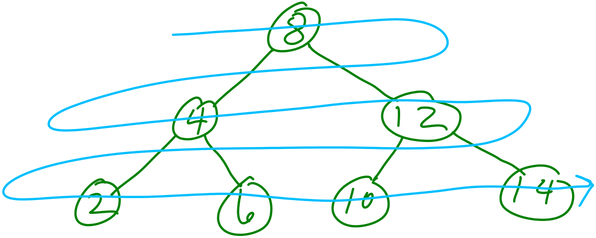
- visit this node itself
- visit the left subtree in **preorder**
- visit the right subtree in **preorder**

postorder



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

level order



- visit this node
- visit this node's children
- visit this node's grandchildren
- visit this node's greatgrandchildren
- ...

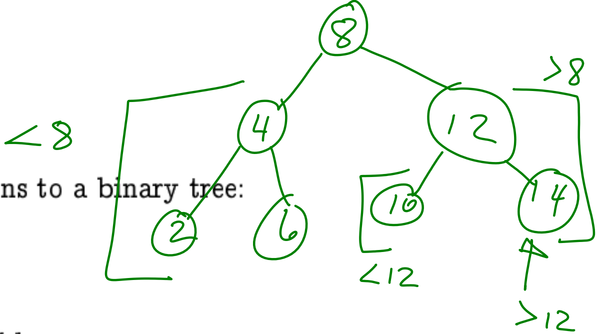
tracing redux

some students report being a bit bewildered by the execution of def **visit_level(t, n, act)**, which means tracing is needed...

- trace **visit_level(None, 7, act)** (for any function **act** you devise)
- trace **visit_level(t, 0, act)** (for some **BinaryTree** with a few levels)
- trace **visit_level(t, 1, act)** (for some **BinaryTree** with a few levels)
- trace **visit_level(t, 2, act)** (for some **BinaryTree** with a few levels)
- trace **visit_level(t, 3, act)** (for some **BinaryTree** with a few levels)
- ...

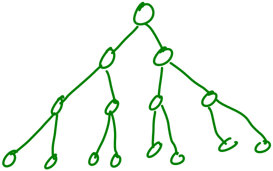
definition

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 $\lceil \lg n \rceil$.

1,000,000 nodes, height < 20

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
```

See code

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
```