

CSCI 48 *Intro. to Computer Science*

Lecture 8: *Binary Trees, BST*

Amir H. Chanaei, Winter 2016

Office Hours: W 16:00–17:45 BA4222

ahchinaei@cs.toronto.edu

<http://www.cs.toronto.edu/~ahchinaei/>

Course webpage:

<http://www.cdf.toronto.edu/~csci48h/winter>

Last week

- ❖ Tracing recursive programs
- ❖ Recursive functions, definitions, and structures
 - Trees
 - Trees terminology ()
- ❖ **Today**
 - Binary trees (arity=2)
 - Binary tree traversals
 - Binary search trees
 - Set

Binary Trees

- ❖ 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 & children left & 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
```

Special methods (eq)

```
def __eq__(self, other):  
    """
```

Return whether BinaryTree self is equivalent to other.

@param BinaryTree self: this binary tree

@param Any other: object to check equivalence to self

@rtype: bool

```
>>> BinaryTree(7).__eq__("seven")  
False
```

```
>>> b1 = BinaryTree(7, BinaryTree(5))
```

```
>>> b1.__eq__(BinaryTree(7, BinaryTree(5), None))  
True  
"""
```

```
return (type(self) == type(other) and  
        self.data == other.data and  
        (self.left, self.right) == (other.left, other.right))
```

Special methods (str)

```
def __str__(self, indent=""):
```

```
    """  
    Return a user-friendly string representing BinaryTree (self)  
    inorder. Indent by indent.
```

```
>>> b = BinaryTree(1, BinaryTree(2, BinaryTree(3)), BinaryTree(4))
```

```
>>> print(b)
```

```
    4
```

```
  1
```

```
    2
```

```
      3
```

```
<BLANKLINE>
```

```
    """
```

```
    right_tree = (self.right.__str__(indent + "    ") if self.right  
                  else "")
```

```
    left_tree = self.left.__str__(indent + "    ") if self.left  
                else ""
```

```
    return (right_tree + "{}{}\\n".format(indent, str(self.data)) +  
            left_tree)
```

Special methods (repr)

```
def __repr__(self):  
    """
```

Represent BinaryTree (self) as a string that can be evaluated to produce an equivalent BinaryTree.

@param BinaryTree self: this binary tree
@rtype: str

```
>>> BinaryTree(1, BinaryTree(2), BinaryTree(3))  
BinaryTree(1, BinaryTree(2, None, None), BinaryTree(3, None, None))  
    """
```

```
return "BinaryTree({}, {}, {})".format(repr(self.data),  
                                         repr(self.left),  
                                         repr(self.right))
```

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__(self, value):  
    """  
    Return whether tree rooted at node contains value.  
  
    @param BinaryTree self: binary tree to search for value  
    @param object value: value to search for  
    @rtype: bool  
  
    >>> BinaryTree(5, BinaryTree(7), BinaryTree(9)).__contains__(7)  
    True  
    """  
  
    return (self.data == value or  
            (self.left and value in self.left) or  
            (self.right and value in self.right))
```

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

```
>>> eval("2+3")
5
```

evaluating a binary expression tree

❖ **def** evaluate(b):
 """

*Evaluate the expression rooted at b. If b is a leaf,
return its float data. Otherwise, evaluate b.left and
b.right and combine them with b.data.*

*Assume: -- b is a non-empty binary tree
-- interior nodes contain data in {"+", "-", "*", "/"}
-- interior nodes always have two children
-- leaves contain float data*

*@param BinaryTree b: binary tree representing arithmetic expression
@rtype: float*

```
>>> b = BinaryTree(3.0)
>>> evaluate(b)
3.0
>>> b = BinaryTree("*", BinaryTree(3.0), BinaryTree(4.0))
>>> evaluate(b)
12.0
"""
```

```
if b.left is None and b.right is None:
    return b.data
else:
    return eval(str(evaluate(b.left)) +
                str(b.data) +
                str(evaluate(b.right)))
```

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.

inorder

```
def inorder_visit(root, act):  
    """
```

Visit each node of binary tree rooted at root in order and act.

@param BinaryTree root: binary tree to visit

@param (BinaryTree)->object act: function to execute on visit

@rtype: None

```
>>> b = BinaryTree(8)
```

```
>>> b = insert(b, 4)
```

```
>>> b = insert(b, 2)
```

```
>>> b = insert(b, 6)
```

```
>>> b = insert(b, 12)
```

```
>>> def f(node): print(node.data)
```

```
>>> inorder_visit(b, f)
```

```
2
```

```
4
```

```
6
```

```
8
```

```
12
```

```
"""
```

```
if root is not None:
```

```
    inorder_visit(root.left, act)
```

```
    act(root)
```

```
    inorder_visit(root.right, act)
```

preorder

- ❖ A recursive definition:
 - visit this node itself
 - visit the left subtree preorder
 - visit the right subtree preorder
- ❖ The code is almost identical to the definition.

preorder

```
def preorder_visit(root, act):  
    """
```

Visit each node of binary tree rooted at root in preorder and act.

@param BinaryTree root: binary tree to visit

@param (BinaryTree)->object act: function to execute on visit

@rtype: None

```
>>> b = BinaryTree(8)
```

```
>>> b = insert(b, 4)
```

```
>>> b = insert(b, 2)
```

```
>>> b = insert(b, 6)
```

```
>>> b = insert(b, 12)
```

```
>>> def f(node): print(node.data)
```

```
>>> preorder_visit(b, f)
```

```
8
```

```
4
```

```
2
```

```
6
```

```
12
```

```
"""
```

```
if root is not None:
```

```
    act(root)
```

```
    preorder_visit(root.left, act)
```

```
    preorder_visit(root.right, act)
```

postorder

- ❖ A recursive definition:
 - visit the left subtree postorder
 - visit the right subtree postorder
 - visit this node itself
- ❖ The code is almost identical to the definition.

postorder

```
def postorder_visit(root, act):
```

```
    """
```

```
    Visit each node of binary tree rooted at root in postorder and act.
```

```
    @param BinaryTree root: binary tree to visit
```

```
    @param (BinaryTree)->object act: function to execute on visit
```

```
    @rtype: None
```

```
    >>> b = BinaryTree(8)
```

```
    >>> b = insert(b, 4)
```

```
    >>> b = insert(b, 2)
```

```
    >>> b = insert(b, 6)
```

```
    >>> b = insert(b, 12)
```

```
    >>> def f(node): print(node.data)
```

```
    >>> postorder_visit(b, f)
```

```
    2
```

```
    6
```

```
    4
```

```
    12
```

```
    8
```

```
    """
```

```
    if root is not None:
```

```
        postorder_visit(root.left, act)
```

```
        postorder_visit(root.right, act)
```

```
    act(root)
```

level order

- ❖ visit this node
 - ❖ visit this node's children
 - ❖ visit this node's grandchildren
 - ❖ visit this node's great grandchildren
 - ❖ ...
-
- ❖ Let's have a helper function
 `visit_level (tree, level, act)`

visit_level

```
def visit_level(t, n, act):  
    """  
    Visit each node of BinaryTree t at level n and act on it. Return  
    the number of nodes visited.  
    @param BinaryTree|None t: binary tree to visit  
    @param int n: level to visit  
    @param (BinaryTree)->Any act: function to execute on nodes at level n  
    @rtype: int  
    >>> b = BinaryTree(8)  
    >>> b = insert(b, 4)  
    >>> b = insert(b, 2)  
    >>> b = insert(b, 6)  
    >>> b = insert(b, 12)  
    >>> def f(node): print(node.data)  
    >>> visit_level(b, 2, f)  
    2  
    6  
    2  
    """  
    if t is None: return 0  
    elif n == 0:  
        act(t)  
        return 1  
    elif n > 0: return (visit_level(t.left, n-1, act)  
                        + visit_level(t.right, n-1, act))  
    else: return 0
```

```

def levelorder(t, act):
    """
    Visit BinaryTree t in level order and act on each node.
    @param BinaryTree|None t: binary tree to visit
    @param (BinaryTree)->Any act: function to use during visit
    @rtype: None
    >>> b = BinaryTree(8)
    >>> b = insert(b, 4)
    >>> b = insert(b, 2)
    >>> b = insert(b, 6)
    >>> b = insert(b, 12)
    >>> def f(node): print(node.data)
    >>> levelorder_visit(b, f)
    8
    4
    12
    2
    6
    """
    # this approach uses iterative deepening

    visited, n = visit_level(t, 0, act), 0
    while visited > 0:
        n += 1
        visited = visit_level(t, n, act)

```

Binary Search Trees

- ❖ 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

Binary Search Trees

- ❖ a BST with 1 node 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 $\lceil \lg n \rceil$
-
- ❖ if the BST is “balanced”, then we can check whether an element is present in about $\lg n$ node accesses

bst_contains

```
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
```

```
"""
```

```
    if node is None:
```

```
        return False
```

```
    elif value < node.data:
```

```
        return bst_contains(node.left, value)
```

```
    elif value > node.data:
```

```
        return bst_contains(node.right, value)
```

```
    else:
```

```
        return True
```

bst_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 node: root of a binary search tree.
```

```
    @param object data: data to insert into BST, if necessary.
```

```
>>> b = BinaryTree(5)
```

```
>>> b1 = insert(b, 3)
```

```
>>> print(b1)
```

```
5
```

```
    3
```

```
<BLANKLINE>
```

```
    """
```

```
    return_node = node
```

```
    if not node:
```

```
        return_node = BinaryTree(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
```

A2 uses sets

a useful data structure

```
>>> {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}
```