CSC148 Intro. to Computer Science

Lecture 8: Binary Trees, BST

Amir H. Chinaei, Summer 2016

Office Hours: R 10-12 BA4222
ahchinaei@cs.toronto.edu
http://www.cs.toronto.edu/~ahchinaei/

Course page: http://www.cs.toronto.edu/~ahchinaei/teaching/20165/csc148/

Last week

- Tracing recursive programs

Last week

- Recursive structures
  - Trees terminology

Today

- Today
  - Binary trees (arity=2)
  - Examples of methods/functions on binary trees
  - Binary tree traversals
  - Introduction to Binary Search Trees (BST)

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
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.

```python
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.
        self.data, self.left, self.right = data, left, right

    def __eq__(self, other):
        # Return whether BinaryTree self is equivalent to other.
        return (type(self) == type(other) and
                self.data == other.data and
                (self.left, self.right) == (other.left, other.right))

    def __repr__(self):
        # Represent BinaryTree (self) as a string that can be evaluated to
        # produce an equivalent BinaryTree.
        return "BinaryTree({}, {}, {})".format(repr(self.data),
                                                repr(self.left),
                                                repr(self.right))

    def __str__(self, indent=
```

Special methods (eq)

```python
def __eq__(self, other):
    # Return whether BinaryTree self is equivalent to other.
    return (type(self) == type(other) and
            self.data == other.data and
            (self.left, self.right) == (other.left, other.right))

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

Special methods (str)

```python
def __str__(self, indent="
```

Special methods (repr)

```python
def __repr__(self):
    # Represent BinaryTree (self) as a string that can be evaluated to
    # produce an equivalent BinaryTree.
    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
contains

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

```python
def __contains__(self, value):
    """Return whether tree rooted at node contains value.
    :param object value: value to search for
    :type value: object
    :rtype: bool
    >>> BinaryTree(3, BinaryTree(7), BinaryTree(9)).__contains__(7)
    True
    >>> BinaryTree(3, BinaryTree(7), BinaryTree(9)).__contains__(1)
    False
    >>> (self.data == value or (self.left and value in self.left) or (self.right and value in self.right))
```

moving on to a new topic

arithmetic expression trees

- Binary arithmetic expressions can be represented as binary trees:

```plaintext```
```
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
```

moving on to a new topic

```python
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 b: binary tree representing arithmetic expression
    :type b: BinaryTree
    :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)))
```

Tree 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.

def inorder_visit(root, act):
    """Visit each node of binary tree rooted at root in order and act.
    :param root: binary tree to visit
    :type root: BinaryTree
    :param act: function to execute on visit
    :type act: (BinaryTree)object
    :type: 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)

Tree traversal: 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.

def preorder_visit(root, act):
    """Visit each node of binary tree rooted at root in preorder and act.
    :param root: binary tree to visit
    :type root: BinaryTree
    :param act: function to execute on visit
    :type act: (BinaryTree)object
    :type: 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:
    preorder_visit(root.left, act)
    act(root)
    preorder_visit(root.right, act)

Tree traversal: 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.

def postorder_visit(root, act):
    """Visit each node of binary tree rooted at root in postorder and act.
    :param root: binary tree to visit
    :type root: BinaryTree
    :param act: function to execute on visit
    :type act: (BinaryTree)object
    :type: 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)
    act(root)
    postorder_visit(root.right, act)
Tree traversal: 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

```python
def visit_level(tree, level, act):
    # Visit each node of BinaryTree t at level n and act on it.
    # param t: binary tree to visit
    # param n: level to visit
    # param act: function to execute on nodes at level n
    # type t: BinaryTree
    # type n: int
    # type act: (BinaryTree) -> Any
    # rtype: int
    return 0
```

```python
def levelorder(t, act):
    # Visit BinaryTree t in level order and act on each node.
    # param t: binary tree to visit
    # param act: function to use during visit
    # type t: BinaryTree
    # type act: (BinaryTree) -> Any
    # type n: None
    # type visited: int
    # type n: int
    # rtype: None
```

```
# this approach uses iterative deepening
visited, n = visit_level(t, 0, act), 0
while visited > 0:
    n += 1
    visited = visit_level(t, n, act)
```

moving on to a new topic

Intro to: 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 one 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**

```python
def bst_contains(node, value):
    """Return whether tree rooted at node contains value.
    Assume node is the root of a Binary Search Tree
    :param node: node of a Binary Search Tree
    :type node: BinaryTree|None
    :param value: value to search for
    :type value: object
    :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**

```python
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 node: root of a binary search tree.
    :type node: BinaryTree
    :param data: data to insert into BST, if necessary
    :type data: object
    >>> 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
```