CSC148-Section:L0301
Week#8-Friday

Instructed by
AbdulAziz Al-Helali
a.alhelali@mail.utoronto.ca
Office hours: Wednesday 11-1, BA2230.

Slides adapted from Professor Danny Heap course material
winter17
Announcement

• In addition to regularly-scheduled office hours we have:
  • Thursday March 1, 11 noon, 1-2, Anujan
  • Friday March 2, 9-11, 12-1, Anujan
  • Monday March 5, 10 a.m. - 8 p.m., Ali
  • Monday March 5, 10 a.m. - 4 p.m., Shahin
  • Monday March 5, 4 p.m. - 8 p.m., Mingjie
Outline

• Tree
  • Levelorder visit
• Binary Trees
def levelorder_visit(t: Tree, act: Callable[[Tree], Any]) -> None:
    
    Visit every node in Tree t in level order and act on the node as you visit it.

    >>> 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
    """
def levelorder_visit(t: Tree, act: Callable[[Tree], Any]) -> None:
    ""
    Visit every node in Tree t in level order and act on the node
    as you visit it.
    >>> 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
    ""

    q = Queue()
    q.add(t)
    while not q.is_empty():
        curr_tree = q.remove()
        act(curr_tree)
        for x in curr_tree.children:
            q.add(x)
Binary Trees

• Each node: has at most two children
• Called: left child and the right child

• Applications:
  • Search algorithms
  • Compression algorithms used in jpeg and .mp3
  • Compilers
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
```python
class BinaryTree:
    
    """
    A Binary Tree, i.e. arity 2.
    """

    def __init__(self, value: object, left: Union['BinaryTree', None]=None,
                 right: Union['BinaryTree', None]=None) -> None:
        """
        Create BinaryTree self with value and children left and right.
        """

        self.value, self.left, self.right = value, left, right
```
special methods...

• We'll want the standard special methods:
  • __eq__
  • __str__
  • __repr__
Contains – As Module Level functions

```python
def contains(node: BinaryTree, value: object) -> bool:
    """
    Return whether tree rooted at self contains value.
    
    >>> t = BinaryTree(5, BinaryTree(7), BinaryTree(9))
    >>> contains(t, 7)
    True
    """
```
Contains – As Module Level functions

```python
def contains(node: BinaryTree, value: object) -> bool:
    """
    Return whether tree rooted at self contains value.
    """
    if node.left is None and node.right is None:
        return node.value == value
    else:
        return (node.value == value
                or contains(node.left, value)
                or contains(node.right, value))
```

The code will NOT work if the BinaryTree has one child as None? No, we need to handle those cases see next slide
```python
def contains(node: Union[BinaryTree, None], value: object) -> bool:
    """
    Return whether tree rooted at self contains value.
    """

    if node.left is None and node.right is None:
        return node.value == value
    elif node.left is None:
        return node.value == value or contains(node.right, value)
    elif node.right is None:
        return node.value == value or contains(node.left, value)
    else:
        return (node.value == value
                or contains(node.left, value)
                or contains(node.right, value))
```

If either left or right nodes is None
We should not call contains on that branch of the tree
Contains – As BinaryTree class Method

```python
def __contains__(self, value: object) -> bool:
    """
    Return whether tree rooted at self contains value.
    """

>>> t = BinaryTree(5, BinaryTree(7), BinaryTree(9))
>>> 7 in t
True
>>> t = BinaryTree(5, BinaryTree(7), None)
>>> 3 in t
False
"""

if self.left is None and self.right is None:
    return self.value == value
elif self.left is None:
    return self.value == value or value in self.right
elif self.right is None:
    return self.value == value or value in self.left
else:
    return (self.value == value
             or value in self.left
             or 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.
def evaluate(b: BinaryTree) -> Union[float, object]:
    
    Evaluate the expression rooted at b. If b is a leaf, return its float value. Otherwise, evaluate b.left and b.right and combine them with b.value.

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

    >>> b = BinaryTree(3.0)
    >>> evaluate(b)
    3.0
    >>> b = BinaryTree("*", BinaryTree(3.0), BinaryTree(4.0))
    >>> evaluate(b)
    12.0
    """
def evaluate(b: BinaryTree) -> Union[float, object]:
    ""
    Evaluate the expression rooted at b. If b is a leaf, return its float value. Otherwise, evaluate b.left and b.right and combine them with b.value.
    
    Assume: -- b is a non-empty binary tree
    -- interior nodes contain value in {"+", "-", "*", "/"}
    -- interior nodes always have two children
    -- leaves contain float value
    
    >>> 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.value
    else:
        return eval("{} {} {}".format(evaluate(b.left),
                                   b.value,
                                   evaluate(b.right)))