Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

MCS 275 Lecture 14
Programming Tools and File Management
Jan Verschelde, 10 February 2017

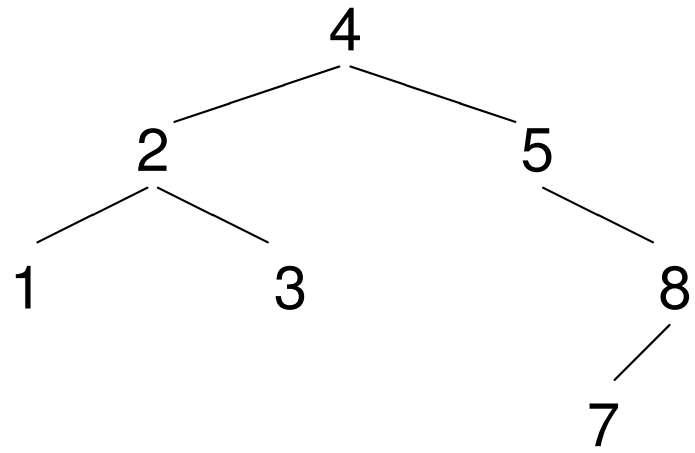
Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

sorting numbers using a tree

Consider the sequence 4, 5, 2, 3, 8, 1, 7

Insert the numbers in a tree:



Rules to insert x at node N:

- if N is empty, then put x in N
- if $x < N$, insert x to the left of N
- if $x \geq N$, insert x to the right of N

Numbers are sorted if we traverse the tree in *inorder*.

the class `Node`

Protected attributes are only used by methods of the class.

```
class Node(object):
    """
    Defines a node in a binary tree.
    """
    def __init__(self, data, *children):
        """
        Returns a node with data.
        Children, left and right, are optional.
        """
        self._data = data  # protected attribute
        if len(children) == 0:
            self._left = None
            self._right = None
        else:
            self._left = children[0]
            self._right = children[1]
```

The `*` in front of `*children` means that children are optional.

the `__str__` method

```
def __str__(self):
    """
    Data at node represented as string.
    """
    return str(self._data)

def value(self):
    """
    Returns the data at the node.
    """
    return self._data
```

For a node `nd`, instead of `nd._data`, we do `nd.value()`.

using the class `Node`

If the definition of class Node is in the file `classnode.py`,
then we may use it as

```
>>> from classtree import Node
>>> nd = Node(2016)
>>> nd
<classtree.Node instance at 0x6a670>
>>> nd.value()
2016
>>> str(nd)
'2016'
```

Notice the difference between the value
and the string representation of the data.

the left and right nodes

The left and right of a node are protected attributes.

```
def left(self):
    """
    Returns the node at the left.
    """
    return self._left

def right(self):
    """
    Returns the node at the right.
    """
    return self._right
```

inserting an item to a node *recursively*

```
def insert(self, item):
    """
    Inserts the item to the node.
    """
    if item != self._data:
        if item < self._data:
            if self._left is None:
                self._left = Node(item)
            else:
                self._left.insert(item)
        else:
            if self._right is None:
                self._right = Node(item)
            else:
                self._right.insert(item)
```

testing interactively

```
>>> from classnode import Node
>>> nd = Node("single")
>>> print(nd)
single
>>> left = Node("left")
>>> right = Node("right")
>>> root = Node("root", left, right)
>>> print(root.left())
left
>>> print(root.right())
right
>>> root.insert("next")
>>> print(root.left().right())
next
```

a test function defined by `main`

```
def main():
    """
    Simple test on the Node class.
    """
    node = Node("single")
    print('a single node :', node)
    left = Node("left")
    right = Node("right")
    root = Node("root", left, right)
    print('the root node :', root)
    print('-> its left :', root.left())
    print('-> its right :', root.right())
    root.insert("next")
    print('after inserting \"next\",')
    print('at the right of the left :', end =' ')
    print(root.left().right())
```

the class `Tree`

A object of the class `Node` cannot be `None`.

```
class Tree(object):
    """
    Defines a ordered binary tree.
    """
    def __init__(self):
        """
        Returns an empty tree.
        """
        self._root = None
```

Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

trees of strings

```
>>> from classtree import Tree
>>> t = Tree()
>>> t.add("here")
>>> t.add("comes")
>>> t.add("the")
>>> t.add("best")
>>> t.add("part")
>>> t.add("we")
>>> t.add("have")
>>> t
here
|->comes
|   |->best
|   |->have
|->the
|   |->part
|   |->we
```

the `__str__()` and `__repr__()` methods

```
def __str__(self):
    """
    Returns the string representation.
    """
    if self._root is None:
        return ''
    else:
        result = self.show(self._root, 0)
        return result[0:len(result)-1]

def __repr__(self):
    """
    The representation is the string representation.
    """
    return str(self)
```

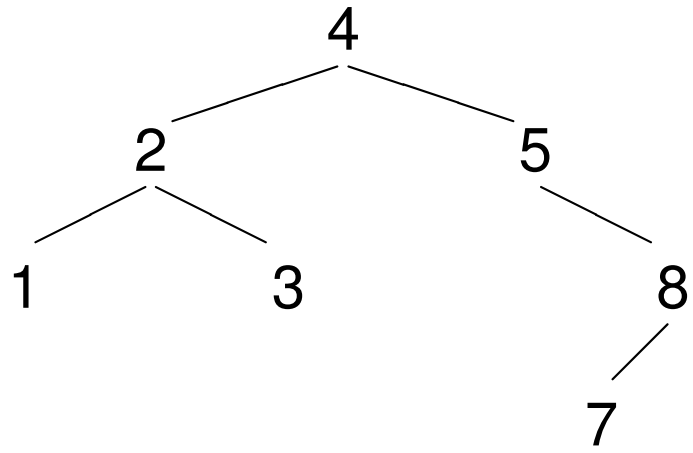
display of a tree

```
def show(self, node, k):
    """
    Returns a string to display a tree,
    for the current node and level k.
    """
    result = (k-1)*"|   "
    if k > 0:
        result = result + "|->"
    result = result + str(node) + "\n"
    if node.left() is not None:
        result += self.show(node.left(), k+1)
    if node.right() is not None:
        result += self.show(node.right(), k+1)
    return result
```

Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

Inorder Traversal



Inorder Traversal of binary tree:

1. inorder traverse left branch of node,
2. visit data at the node,
3. inorder traverse right branch of node.

The data comes out sorted, for an ordered binary tree.
For an arithmetic expression, for example: `3 + 4`,
this order corresponds to the *infix notation*: `3 + 4`.

code for inorder traversal

The method applies to a node.

```
def inorder_nodes(self, node):
    """
    Returns a list by traversing nodes in inorder.
    """
    result = []
    if node.left() is not None:
        result = self.inorder_nodes(node.left())
    result.append(node.value())
    if node.right() is not None:
        result += self.inorder_nodes(node.right())
    return result
```

the method on a `Tree` object

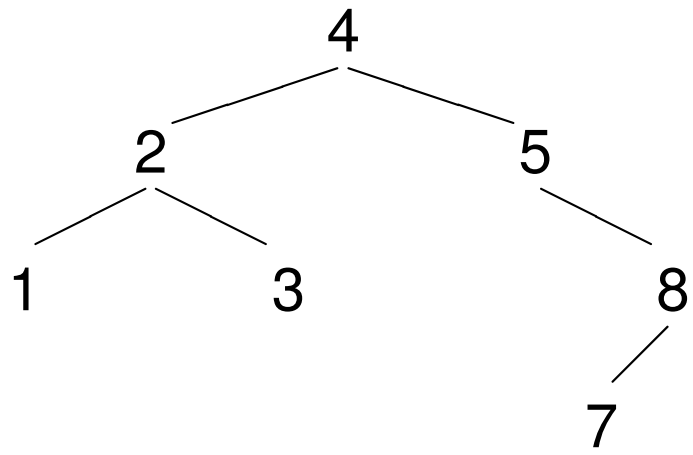
A tree can be `None`.

```
def inorder(self):
    """
    Returns the data as a list in inorder.
    """
    if self._root is None:
        return []
    else:
        return self.inorder_nodes(self._root)
```

Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

Preorder Traversal



Preorder Traversal of binary tree:

1. visit data at the node,
2. preorder traverse left branch of node,
3. preorder traverse right branch of node.

For an arithmetic expression, for example: `3 + 4`,
this order corresponds to the *prefix notation*: `+ 3 4`.

code for preorder traversal

The method applies to a node.

```
def preorder_nodes(self, node):
    """
    Returns a list by traversing nodes in preorder.
    """
    result = [node.value()]
    if node.left() is not None:
        result += self.preorder_nodes(node.left())
    if node.right() is not None:
        result += self.preorder_nodes(node.right())
    return result
```

the method on a `Tree` object

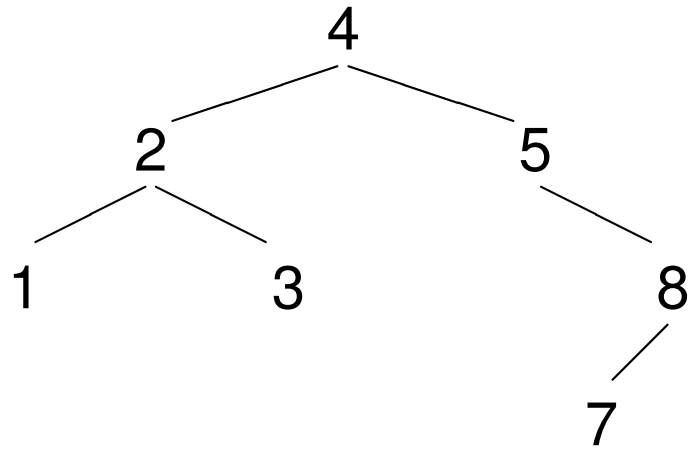
A tree can be `None`.

```
def preorder(self):
    """
    Returns the data as a list in preorder.
    """
    if self._root is None:
        return []
    else:
        return self.preorder_nodes(self._root)
```

Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

Postorder Traversal



Postorder Traversal of binary tree:

1. postorder traverse left branch of node,
2. postorder traverse right branch of node,
3. visit data at the node.

For an arithmetic expression, for example: `3 + 4`,
this order corresponds to the *postfix notation*: `3 4 +`.

code for postorder traversal

```
def postorder_nodes(self, node):
    """
    Returns a list by traversing nodes in postorder.
    """
    result = []
    if node.left() is not None:
        result += self.postorder_nodes(node.left())
    if node.right() is not None:
        result += self.postorder_nodes(node.right())
    result.append(node.value())
    return result
```

the method on a `Tree` object

```
def postorder(self):
    """
    Returns a list in postorder.
    """
    if self._root is None:
        return []
    else:
        return self.postorder_nodes(self._root)
```

running the main function

```
$ python classtree.py
here
|->comes
|   |->best
|   |->have
|->the
|   |->part
|   |->we
['here', 'comes', 'best', 'have', 'the', 'part', 'we']
['best', 'comes', 'have', 'here', 'part', 'the', 'we']
['best', 'have', 'comes', 'part', 'we', 'the', 'here']
```

storing expressions

How to convert `'5 * (x + (3 - 8*y))/z'`
into `'5 * [x + [3 - 8*y]]/z'` ?

```
>>> s = '5 * (x + (3 - 8*y))/z'
>>> L = s.split('(')
'5 * [x + [3 - 8*y]]/z'
>>> r = '['.join(L)
'5 * [x + [3 - 8*y]]/z'
```

The expression tree:

'[5,*,[x, + ,[3 - [8,*,y]],/,z]]'

When storing tree as list of lists, consider `*`, `x`, ...
as strings: `'*'`, `'x'`.

Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

substitution

```
def subs(form, x, y):
    """
    Replaces all occurrences of x
    in the string form by y.
    """
    data = form.split(x)
    return y.join(data)

def main():
    """
    Prompts user for string and two symbols.
    """
    ins = input("Give a string : ")
    x = input("   what to replace : ")
    y = input("replacement string : ")
    out = subs(ins, x, y)
    print("the new string \"%s\"" % out)

main()
```

Traversing Binary Trees

1. Binary Trees as Objects
   - classes for nodes and trees
   - representations of trees
2. Traversing Trees
   - inorder traversal
   - preorder traversal
   - postorder traversal
3. Expression Trees
   - substitution of variables
   - recursive evaluation of expressions
4. Exercises

Evaluation

Evaluation of a simple binary expression follows
its recursive definition:

```
< operand > ::= < variable >|< number >
    |< expression >

< operator > ::= < + >|< - >|< * >|< / >

< expression > ::= < operand >|
    < operand >< operator >< operand >
```

Either an expression evaluates directly to a number or the value of a variable (base cases),

or an expression consists of two expressions,
separated by an operator (recursive call).

Exercises

1. Write a recursive algorithm to generate a *complete* binary tree of k levels. The user determines the value for k. How many data elements does this complete tree have? Use a preorder traversal to assign an increasing sequence of integer numbers as data in the nodes.
2. Draw a complete binary tree with k levels on canvas. Let k be given by the user in an entry field.
3. For the tree drawn in exercise 2, write a GUI that allows the user to enter, view, and modify the elements at each node by pressing the mouse at the location of the node as shown on canvas.
4. Describe an algorithm to convert the infix notation of expressions into postfix, where the operand comes last, e.g.: `3 + 9` becomes `3 9 +`.