Data Structures in Python

10. Trees (II)

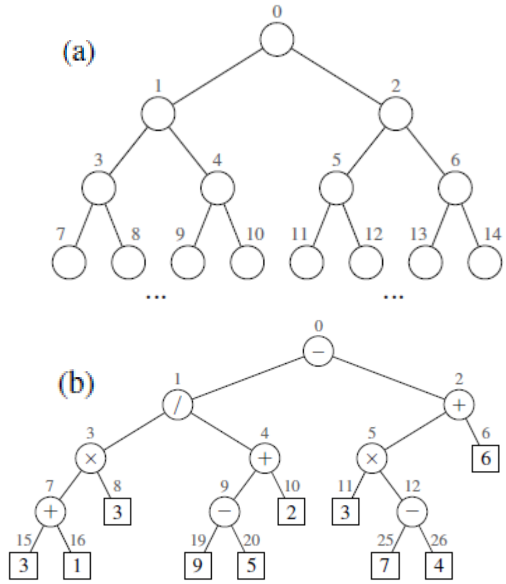
Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Array-Based Representation of a Binary Tree

- We have discussed the linked structure for binary trees.
- An alternative representation of a binary tree *T* is based on a way of numbering the positions of *T*.
- For every position *p* of *T*, let ***f (p)*** be the integer defined as follows.
  - If *p* is the ***root*** of *T*, then ***f (p) = 0***.
  - If *p* is the ***left child*** of position *q*, then ***f (p) = 2 f (q)+1***.
  - If *p* is the ***right child*** of position *q*, then ***f (p) = 2 f (q)+2***.
- The numbering function *f* is known as a ***level numbering*** of the positions in a binary tree *T*, for it numbers the positions on each level of *T* in increasing order from left to right. (See Figure (a).)
- Note that the level numbering is based on ***potential positions*** within the tree, not actual positions of a given tree, so they are not necessarily consecutive.

Array-Based Representation of a Binary Tree

- For example, in Figure (b), there are no nodes with level numbering 13 or 14, because the node with level numbering 6 has no children.
- One advantage of an array-based representation of a binary tree is that a position *p* can be represented by the single integer ***f (p)***, and that position-based methods such as root, parent, left, and right can be implemented using simple arithmetic operations on the number ***f (p)***.



Binary tree level numbering:
(a) general scheme; (b) an example.

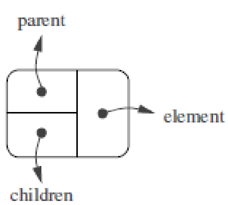
Array-Based Representation of a Binary Tree

- The indices of left child, right child, and parent of a position *p* of *T* are calculated based on the above formula for the level numbering.
- <u>The details of a complete implementation is left as an exercise</u>.
- A drawback of an array representation is that some update operations for trees cannot be efficiently supported.
- For example, for a binary tree of size n, deleting a node and promoting its child takes O(n) time because it is not just the child that moves locations within the array, but all descendants of that child.

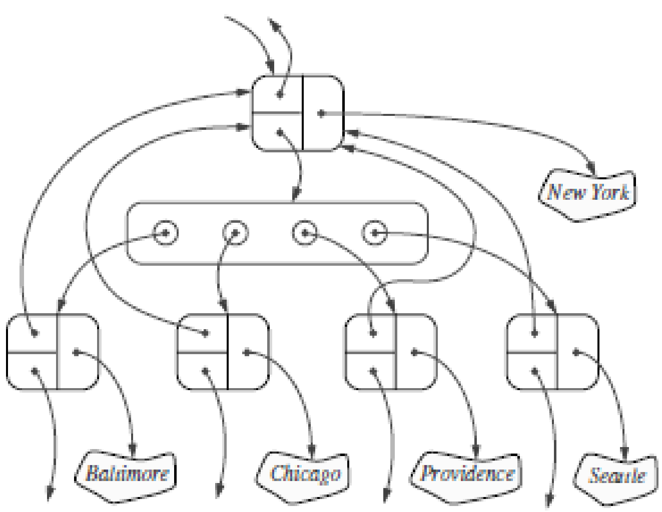
Linked Structure for General Trees

- When representing a ***binary tree*** with a linked structure, each node explicitly maintains fields left and right as references to individual children.
- For a ***general tree***, there is no a priori limit on the number of children that a node may have.
- A natural way to realize a general tree *T* as a linked structure is to have each node store a single *container* of references to its children.
- For example, a children field of a node can be a *Python list* of references to the children of the node (if any).
- Such a linked representation is schematically illustrated in the following figure.

Linked Structure for General Trees



the structure of a node



larger portion of the data structure associated with a node and its children.

The linked structure for a general tree:

Tree Traversal Algorithms

- A ***traversal*** of a tree *T* is a systematic way of accessing, or "visiting," all the positions of *T*.
- The specific action associated with the "visit" of a position *p* depends on the application of this traversal, and could involve anything from incrementing a counter to performing some complex computation for *p*.
- In this section, we describe several common traversal schemes for trees, implement them in the context of our various tree classes, and discuss several common applications of tree traversals.

❖ ***Preorder Traversal of General Trees***

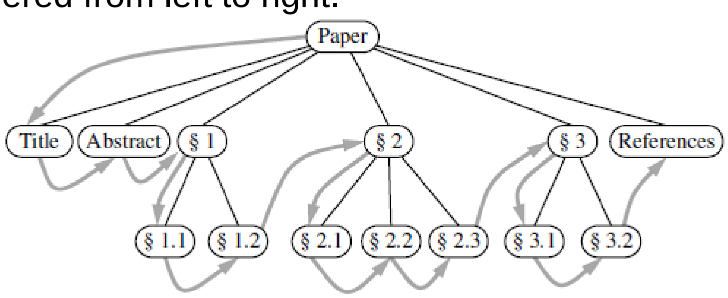
- In a ***preorder traversal*** of a tree *T*, *the root of T is visited first and then the subtrees rooted at its children are traversed recursively*.
- If the tree is ordered, then the subtrees are traversed according to the order of the children.

7

Tree Traversal Algorithms

- The pseudo-code for the ***preorder traversal*** of the subtree rooted at a position *p* of *T* is as follows:

  Algorithm preorder(T, p):
  perform the "visit" action for position p
  for each child c in T.children(p) **do**
  preorder(T, c) {recursively traverse the subtree rooted at c}
- The figure shows the order in which positions of a sample tree are visited during an application of the ***preorder traversal*** algorithm, where the children of each position are ordered from left to right.



8

Tree Traversal Algorithms

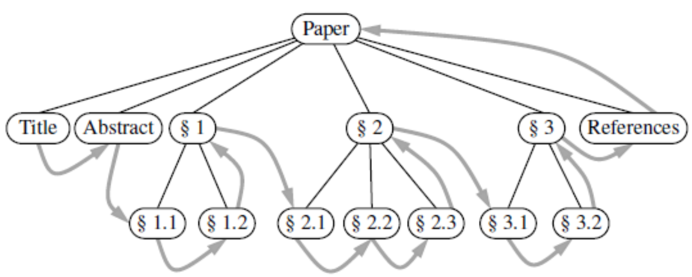
❖ *Postorder Traversal of General Trees*

- The ***postorder traversal algorithm***, can be viewed as the opposite of the preorder traversal, because *it recursively traverses the subtrees rooted at the children of the root first, and then visits the root* (hence, the name "postorder").
- Pseudo-code for the ***postorder traversal*** is as follows:

```
Algorithm postorder(T, p):
   for each child c in T.children(p) do
      postorder(T, c)  {recursively traverse the subtree rooted at c}
   perform the "visit" action for position p
```

- The following figure shows an example of the ***postorder traversal*** of a subtree rooted at position p of a tree T.

Tree Traversal Algorithms

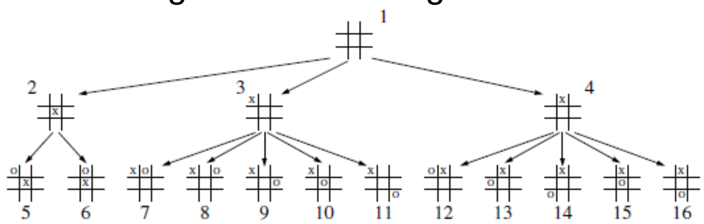


❖ *Breadth-First Tree Traversal*

- Another common traversal approach is to traverse a tree so that *we visit all the positions at depth d before we visit the positions at depth d +1*.
- Such an algorithm is known as a ***breadth-first traversal***.
- A ***breadth-first traversal*** is used in software for playing games.

Tree Traversal Algorithms

- A ***game tree*** represents the possible choices of moves that might be made by a player (or computer) during a game, with the root of the tree being the initial configuration for the game.
- For example, the figure displays a partial game tree for Tic-Tac-Toe,



  with annotations indicating the order in which positions are visited in a breadth-first traversal.
- A ***breadth-first traversal*** of such a game tree is often performed because a computer may be unable to explore a complete game tree in a limited amount of time.
- So the computer will consider all moves, then responses to those moves, going as deep as computational time allows.

Tree Traversal Algorithms

- Pseudo-code for a ***breadth-first traversal*** is given

```
Algorithm breadthfirst(T):
   Initialize queue Q to contain T.root( )
   while Q not empty do
      p = Q.dequeue( )      {p is the oldest entry in the queue}
      perform the "visit" action for position p
      for each child c in T.children(p) do
         Q.enqueue(c) {add p's children to the end of the queue
                                                  for later visits}
```

- The process is not recursive, since we are not traversing entire subtrees at once.
- We use a queue to produce a ***FIFO*** (i.e., first-in first-out) semantics for the order in which we visit nodes.

Tree Traversal Algorithms

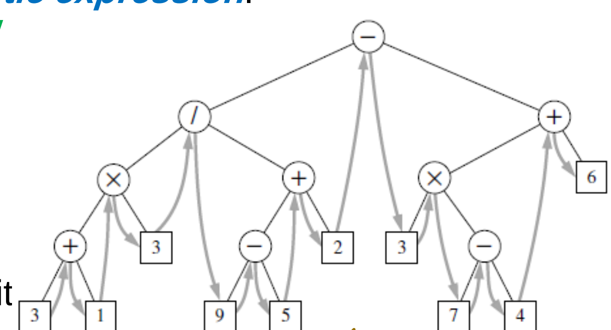
❖ ***Inorder Traversal of a Binary Tree***

- The standard ***preorder***, ***postorder***, and ***breadth-first traversals*** that were introduced for ***general trees***, can be directly applied to ***binary trees***.
- Now, we introduce another common traversal algorithm specifically for a ***binary tree***.
- During an ***inorder traversal***, we visit a position between the recursive traversals of its left and right subtrees.
- The inorder traversal of a binary tree *T* can be informally viewed as visiting the nodes of *T* "from left to right."
- Indeed, for every position *p*, the ***inorder traversal*** *visits p after all the positions in the left subtree of p and before all the positions in the right subtree of p*.
- Pseudo-code for the ***inorder traversal*** algorithm is given below:

Tree Traversal Algorithms

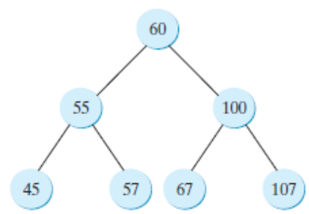
Algorithm inorder(p):
    if p has a left child lc **then**
      inorder(lc) {recursively traverse the left subtree of p}
    perform the "visit" action for position p
    if p has a right child rc **then**
      inorder(rc) {recursively traverse the right subtree of p}

- The figure shows the ***inorder traversal*** a ***binary tree*** that represents an ***arithmetic expression***.
- The ***inorder traversal*** visits positions in a consistent order with the standard representation of the expression, as in 3+1×3/9−5+2. . . (albeit without parentheses).



Binary Search Trees

- An important application of the ***inorder traversal*** algorithm arises when we store an ordered sequence of elements in a binary tree, defining a structure we call a ***binary search tree***.
- Let *S* be a set whose unique elements have an order relation. For example, *S* could be a set of integers.
- A ***binary search tree*** for *S* is a binary tree *T* such that, for each position *p* of *T*:
  - Position *p* stores an element of *S*, denoted as *e*(*p*).
  - Elements stored in the left subtree of *p* (if any) are less than *e*(*p*).
  - Elements stored in the right subtree of *p* (if any) are greater than *e*(*p*).
- The figure shows an example of a binary search tree.

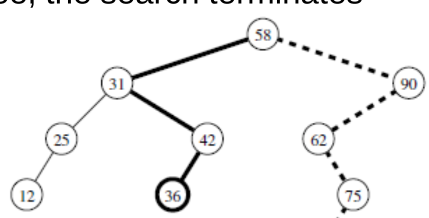


- The above properties assure that an ***inorder traversal*** of a ***binary search tree*** *T* visits the elements in increasing order

15

Binary Search Trees

- We can use a binary search tree *T* for set *S* to find whether a given search value *v* is in *S*, by traversing a path down the tree *T*, starting at the root.
- At each internal position *p* encountered, we compare our search value *v* with the element *e*(*p*) stored at *p*.
  - If $v < e(p)$, then the search continues in the left subtree of *p*.
  - If $v = e(p)$, then the search terminates successfully.
  - If $v > e(p)$, then the search continues in the right subtree of *p*.
  - Finally, if we reach an empty subtree, the search terminates unsuccessfully.
- The figure illustrates examples of the search operation.
  The solid path is traversed when searching (successfully) for 36.
  The dashed path is traversed when searching (unsuccessfully) for 70.



16

Implementing Tree Traversals in Python

- When first defining the ***tree ADT***, we stated that tree T should include support for the following methods:
  - ***T.positions( ):*** Generate an iteration of all ***positions*** of tree T.
  - ***iter(T):*** Generate an iteration of all ***elements*** stored within tree T.
- At that time, we did not make any assumption about the order in which these iterations report their results.
- In this section, we demonstrate how the ***tree traversal algorithms*** can be used to produce these iterations.
- Support for the ***iter(T)*** syntax can be formally provided by a concrete implementation of the special method ***__iter__*** within the ***abstract base class Tree***.
- We rely on Python's ***generator*** syntax as the mechanism for producing iterations.

Implementing Tree Traversals in Python

- The implementation of ***Tree.__iter__*** is given in the following code fragment:

```
def __iter__(self):
    """"Generate an iteration of the tree s elements.""""
    for p in self.positions( ):          # use same order as positions()
        yield p.element( )               # but yield each element
```

- This code iterates all ***elements*** of a ***Tree instance***, based upon an iteration of the ***positions*** of the tree.
- This code should be included in the body of the ***Tree class***.
- To implement the ***positions method***, we have to choose one of the tree traversal algorithms.
- We will provide independent implementations of each strategy that can be called directly by a user of the class.
- We can then trivially adapt one of those as a default order for the ***positions method*** of the ***tree ADT***.

Implementing Tree Traversals in Python

➢ ***Preorder Traversal***

- Firstly, we provide a public method with calling signature ***T.preorder( )*** for tree T, which generates a preorder iteration of all positions within the tree.
- The recursive algorithm for generating a ***preorder traversal***, described in Slide 9, must be parameterized by a specific position within the tree that serves as the root of a subtree to traverse.
- So, we define a ***nonpublic utility method*** with the desired recursive parameterization, and then the ***public method preorder*** invokes the nonpublic method upon the root of the tree.
- The implementation of such a design is given in the following code fragment.

Implementing Tree Traversals in Python

```
def preorder(self):
  """Generate a preorder iteration of positions in the tree."""
  if not self.is_empty( ):
   for p in self._subtree_preorder(self.root( )): # start recursion
    yield p

def _subtree_preorder(self, p):
  """Generate a preorder iteration of positions in subtree rooted at p."""
  yield p # visit p before its subtrees
  for c in self.children(p): # for each child c
   for other in self._subtree_preorder(c): # do preorder of c's subtree
    yield other # yielding each to our caller
```

- This code should be included in the body of the Tree class.
- Both ***preorder*** and the utility ***_subtree_preorder*** are generators.

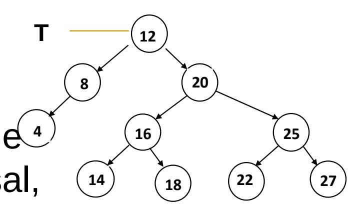
Implementing Tree Traversals in Python

- Rather than perform a “visit” action from within this code, we yield each position to the caller and let the caller decide what action to perform at that position.
- The ***_subtree_preorder*** method is recursive.
  - In order to yield all positions within the subtree of child *c*, we loop over the positions yielded by the recursive call ***self._subtree_preorder(c)***, and re-yield each position in the outer context.
  - Note that if *p* is a leaf, the for loop over ***self.children(p)*** is trivial (this is the base case for this recursion).
- The public ***preorder*** method re-yields all positions that are generated by the recursive process starting at the root of the tree; if the tree is empty, nothing is yielded.
- At this point, we have provided full support for the ***preorder generator***.

Implementing Tree Traversals in Python

- A user of the class can therefore write code such as

```
for p in T.preorder( ):
        # ”visit” position p
```



- For example, to print all the elements of the tree ***T***, generated at the end of the previous lecture, using preorder traversal, we can add the following code:

```
print("Preorder traversal of T:")
for p in T.preorder( ):
   print(p.element(), end =" ")    # print element at position p
```

Output

Preorder traversal of T:
12 8 4 20 16 14 18 25 22 27

- Now, we can provide an implementation of the ***positions method*** for the ***Tree class*** that uses a ***preorder traversal*** to generate the results, and include it within the ***Tree class***:

```
def positions(self):
  """Generate an iteration of the tree s positions."""
  return self.preorder( ) # return entire preorder iteration
```

Implementing Tree Traversals in Python

- Rather than loop over the results returned by the ***preorder*** call, we return the entire iteration as an ***object***.
- So, the following code calls the ***positions method*** to get an object containing all elements of ***T*** using preorder traversal, then iterates over it to display these elements as shown in the previous slide:

```
for p in T.positions( ):
    print(p.element(), end =" ")    # ”visit” position p
```

➢ *Postorder Traversal*

- We can implement a ***postorder traversal*** using very similar technique as with a ***preorder traversal***.
- The only difference is that within the recursive utility for a ***postorder*** we wait to yield position *p* until *after* we have recursively yield the positions in its subtrees.

Implementing Tree Traversals in Python

- An implementation of ***postorder traversal*** is given in the following code fragment, which should be included in the body of the ***Tree class***:

```
def postorder(self):
 """Generate a postorder iteration of positions in the tree."""
 if not self.is_empty( ):
  for p in self._subtree_postorder(self.root( )): # start recursion
   yield p

def _subtree_postorder(self, p):
 """Generate a postorder iteration of positions in subtree rooted at p."""
 for c in self.children(p): # for each child c
  for other in self._subtree_postorder(c): # do postorder of c’s subtree
   yield other # yielding each to our caller
 yield p
```

- The code below prints all the elements of our tree ***T*** using ***postorder traversal***

```
print("Postorder traversal of T:")
for p in T.postorder( ):
   print(p.element(), end =" ")    # print element at position p
```

Output

Postorder traversal of T:
4 8 14 18 16 22 27 25 20 12

Implementing Tree Traversals in Python

➢ *Breadth-First Traversal*

- The following code fragment provides an implementation of the breadth-first traversal algorithm in the context of our Tree class.
- Recall that the ***breadth-first traversal*** algorithm is not recursive; it uses a queue of positions to manage the traversal process.
- The implementation uses the **LinkedQueue class** from Lecture 8.

```
def breadthfirst(self):
  """Generate a breadth-first iteration of the positions of the tree."""
  if not self.is_empty( ):
   fringe = LinkedQueue( ) # known positions not yet yielded
   fringe.enqueue(self.root( )) # starting with the root
   while not fringe.is_empty( ):
    p = fringe.dequeue( ) # remove from front of the queue
    yield p # report this position
    for c in self.children(p):
     fringe.enqueue(c) # add children to back of queue
```

25

Implementing Tree Traversals in Python

- The code below prints all the elements of our tree ***T*** using ***breadth-first traversal***

```
print("breadth-first traversal of T:")
for p in T.breadthfirst( ):
    print(p.element(), end =" ")    # ”visit” position p
print()
```

Output ➔

breadth-first traversal of T:
12 8 20 4 16 25 14 18 22 27

➢ *Inorder Traversal for Binary Trees*

- The ***preorder***, ***postorder***, and ***breadth-first traversal*** algorithms are applicable to all trees, and so we include their implementations within the **Tree abstract base class**.
- Those methods are inherited by the **abstract BinaryTree class**, the concrete **LinkedBinaryTree class**, and any other dependent tree classes we might develop.
- The ***inorder traversal*** algorithm, because it explicitly relies on the notion of a left and right child of a node, only applies to ***binary trees***.

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

26

Implementing Tree Traversals in Python

- We therefore include its definition within the body of the **BinaryTree class**.
- The following code fragment provides an implementation of the ***inorder traversal***.

```
def inorder(self):
 """Generate an inorder iteration of positions in the tree."""
 if not self.is_empty( ):
  for p in self. _subtree_inorder(self.root( )):
   yield p

def _subtree_inorder(self, p):
 """Generate an inorder iteration of positions in subtree rooted at p."""
 if self.left(p) is not None: # if left child exists, traverse its subtree
  for other in self. _subtree_inorder(self.left(p)):
   yield other
 yield p # visit p between its subtrees
 if self.right(p) is not None: # if right child exists, traverse its subtree
  for other in self. _subtree_inorder(self.right(p)):
   yield other
```

- This code should be included in the **BinaryTree class**

Implementing Tree Traversals in Python

- The code below prints all the elements of our tree ***T*** using ***inorder traversal***

```
print("Inorder traversal of T:")
for p in T.inorder( ):
   print(p.element(), end =" ")   # ”visit” position p
print()
```

Output

Inorder traversal of T:
4 8 12 14 16 18 20 22 25 27

- We can make ***inorder traversal*** the default for the **BinaryTree class** by overriding the positions method that was inherited from the **Tree class** as follows:

```
# override inherited version to make inorder the default
def positions(self):
 """Generate an iteration of the tree s positions."""
 return self.inorder( ) # make inorder the default
```

- This code should be included in the **BinaryTree** class
- Now, running the following code gives the same result as the above code

```
for p in T.positions( ):
   print(p.element(), end =" ")   # ”visit” position p
```

Case Study: An Expression Tree

- Slide 14 showed an example of using a ***binary tree*** to represent the structure of an ***arithmetic expression***.
- In this section, we define a new **ExpressionTree class** to be used for constructing such trees, and for displaying and evaluating the ***arithmetic expression*** that such a tree represents.
- The **ExpressionTree class** is defined as a subclass of **LinkedBinaryTree**, and we use the nonpublic ***mutators*** to construct such trees.
- Each internal node stores a string that defines a binary operator (e.g., +), and each leaf stores a numeric value (or a string representing a numeric value).
- The goal is to build arbitrarily complex expression trees for compound arithmetic expressions such as (((3+1)×4)/((9−5)+2)).

Case Study: An Expression Tree

- It suffices for the **ExpressionTree class** to support two basic forms of initialization (***constructor***):
  - ***ExpressionTree(value)*: ** Create a tree storing the given value at the root.
  - ***ExpressionTree(op,E1,E2)*: ** Create a tree storing string op at the root (e.g., +), and with the structures of existing ExpressionTree instances E1 and E2 as the left and right subtrees of the root, respectively.
- This ***constructor*** is given in the code shown below.
- The class inherits from **LinkedBinaryTree**, so it has access to all the nonpublic update methods that were defined before.
- We use ***_add_root*** to create an initial root of the tree storing the token provided as the first parameter.

Case Study: An Expression Tree

- Then we perform run-time checking of the parameters to determine whether the caller invoked the one-parameter version of the ***constructor*** (in which case, we are done), or the three-parameter form.
- In that case, we use the inherited ***_attach*** method to incorporate the structure of the existing trees as subtrees of the root.
- The code for the beginning of an **ExpressionTree class**:

```
from LinkedBinaryTree import LinkedBinaryTree
class ExpressionTree(LinkedBinaryTree):
 """An arithmetic expression tree."""
 def __init__(self, token, left=None, right=None):
  """Create an expression tree.
  In a single parameter form, token should be a leaf value (e.g., 42 ),
  and the expression tree will have that value at an isolated node.
  In a three-parameter version, token should be an operator,
  and left and right should be existing ExpressionTree instances
  that become the operands for the binary operator.
  """
```


Case Study: An Expression Tree

```
  super().__init__( ) # LinkedBinaryTree initialization
  if not isinstance(token, str):
   raise TypeError('Token must be a string')
  self._add_root(token) # use inherited, nonpublic method
  if left is not None: # presumably three-parameter form
   if token not in '+-*x/':
    raise ValueError('token must be valid operator')
   self._attach(self.root( ), left, right) # use inherited, nonpublic method
 def __str__(self):
  """Return string representation of the expression."""
  pieces = [ ] # sequence of piecewise strings to compose
  self._parenthesize_recur(self.root( ), pieces)
  return ''.join(pieces)
 def _parenthesize_recur(self, p, result):
  """Append piecewise representation of p's subtree to resulting list."""
  if self.is_leaf(p):
   result.append(str(p.element( ))) # leaf value as a string
  else:
   result.append('(') # opening parenthesis
   self._parenthesize_recur(self.left(p), result)   # left subtree
   result.append(p.element()) # operator
   self._parenthesize_recur(self.right(p), result) # right subtree
   result.append(')') # closing parenthesis
```

Case Study: An Expression Tree

➢ *Composing a Parenthesized String Representation*

- A string representation of an existing expression tree instance, for example, as (((3+1)x4)/((9-5)+2)) , can be produced by displaying tree elements using an inorder traversal, but with opening and closing parentheses inserted with a preorder and postorder step, respectively.
- In the context of an **ExpressionTree class**, we support a special ***__str__ method*** that returns the appropriate string.
- Because it is more efficient to first build a sequence of individual strings to be joined together, the implementation of ***__str__*** relies on a nonpublic, recursive method named ***_parenthesize_recur*** that appends a series of strings to a list.
- These methods are included in the above code.

Case Study: An Expression Tree

➢ *Expression Tree Evaluation*

- The numeric evaluation of an expression tree can be accomplished with a simple application of a ***postorder*** traversal.
- If we know the values represented by the two subtrees of an internal position, we can calculate the result of the computation that position designates.
- Pseudo-code for the recursive evaluation of the value represented by a subtree rooted at position *p* is given below:

```
Algorithm evaluate_recur(p):
  if p is a leaf then
    return the value stored at p
  else
    let op be the operator stored at p
    x = evaluate_recur(left(p))
    y = evaluate_recur(right(p))
    return x op y
```

Case Study: An Expression Tree

- To implement this algorithm in the context of a Python **ExpressionTree class**, we provide a public evaluate method that is invoked on instance T as ***T.evaluate( )***.
- The following code fragment provides such an implementation, using a nonpublic ***_evaluate_recur*** method that computes the value of a designated subtree.

```
def evaluate(self):
 """Return the numeric result of the expression."""
 return self._evaluate_recur(self.root( ))
def _evaluate_recur(self, p):
 """Return the numeric result of subtree rooted at p."""
 if self.is_leaf(p):
  return float(p.element( )) # we assume element is numeric
 else:
  op = p.element( )
  left_val = self._evaluate_recur(self.left(p))
  right_val = self._evaluate_recur(self.right(p))
  if op == '+':
   return left_val + right_val
  elif op == '-':
   return left_val - right_val
```


Case Study: An Expression Tree

```
  elif op == '/':
   return left_val / right_val
  else:
   return left_val * right_val # treat 'x' or '*' as multiplication
```

➢ *Building an Expression Tree*

- The constructor for the **ExpressionTree class** provides basic functionality for combining existing trees to build larger expression trees.
- Now we see how to construct a tree that represents an expression for a given string, such as (((3+1)x4)/((9-5)+2)) .
- To automate this process, we use a bottom-up construction algorithm, assuming that a string can first be tokenized so that multidigit numbers are treated atomically (see the exercise at the end of the lecture), and that the expression is fully parenthesized.
- The algorithm uses a **stack *S*** while scanning tokens of the input expression *E* to find values, operators, and right parentheses. (Left parentheses are ignored.)

Case Study: An Expression Tree

- When we see an operator **op**, we push it on the stack.
- When we see a literal value *v*, we create a single-node expression tree *T* storing *v*, and push *T* on the stack.
- When we see a right parenthesis,) , we pop the top three items from the stack *S*, which represent a subexpression (E1 **op** E2). We then construct a tree *T* using trees for E1 and E2 as subtrees of the root storing **op**, and push the resulting tree *T* back on the stack.

- We repeat this until the expression *E* has been processed, at which time the top element on the stack is the expression tree for *E.*
- An implementation of this algorithm is given in the following code fragment in the form of a stand-alone function named ***build_expression_tree***, which produces and returns an appropriate **ExpressionTree** instance, assuming the input has been tokenized.

37

Case Study: An Expression Tree

```
def build_expression_tree(tokens):
 """Returns an ExpressionTree based upon a tokenized
 expression."""
 S = [ ] # we use Python list as stack
 for t in tokens:
  if t in '+-x*/' : # t is an operator symbol
   S.append(t) # push the operator symbol
  elif t not in '()' : # consider t to be a literal
   S.append(ExpressionTree(t)) # push trivial tree storing value
  elif t == ')' : # compose a new tree from three constituent parts
   right = S.pop( ) # right subtree as per LIFO
   op = S.pop( ) # operator symbol
   left = S.pop( ) # left subtree
   S.append(ExpressionTree(op, left, right)) # repush tree
   # we ignore a left parenthesis
 return S.pop( )
```

Case Study: An Expression Tree

- ***Exercise:***
- The above ***build_expression_tree*** method of the **ExpressionTree class** requires input that is an iterable of string tokens.
- We used a convenient example, (((3+1)x4)/((9-5)+2)) , in which each character is its own token, so that the string itself sufficed as input to build expression tree.
- In general, a string, such as ‘(35 + 14)’, must be explicitly tokenized into list [‘(‘ , ’35’ , ‘+’ , ’14’ , ‘)’] so as to ignore whitespace and to recognize multidigit numbers as a single token.
- Write a utility method, ***tokenize(raw)***, that returns such a list of tokens for a raw string.