Data structure and algorithm in Python

Tree

Seongjin Lee

Dept of Aerospace and Software Engineering, Gyeongsang National University

- 1. General Trees
- 2. Binary Tree
- 3. Implementing Trees
- 4. Tree Traversal Algorithms
- 5. Expression Tree

General Trees

Tree is one of the most important nonlinear data structures.

- Tree structures are indeed a breakthrough in data organization, for they allow us to implement a host of algorithms much faster than when using linear data structures, such as array-based lists or linked lists.
- Trees also provide a natural organization for data, and consequently have become ubiquitous structures in file systems, graphical user interfaces, databases, Web sites, and other computer systems.

When we say that trees are "nonlinear", we are referring to an organizational relationship that is richer than the simple "before" and "after" relationships between objects in sequences. The relationships in a tree are hierarchical, with some objects being "above" and some "below" others.

General Trees

Actually, the main terminology for tree data structures comes from family trees, with the terms "parent", "child", "ancestor" and "descendant" being the most common words used to describe rela- tionships.



4

General Trees Tree Definitions and Properties

A tree is an abstract data type that stores elements hierarchically.

With the exception of the top element, each element in a tree has a parent element and zero or more children elements.

We typically call the top element the root of the tree, but it is drawn as the highest element, with the other elements being connected below.



Figure 8.2: A tree with 17 nodes representing the organization of a fictitious corporation. The root stores *Electronics R'Us*. The children of the root store *R&D*, *Sales*, *Purchasing*, and *Manufacturing*. The internal nodes store *Sales*, *International*, *Overseas*, *Electronics R'Us*, and *Manufacturing*.

Definition : Formal Tree Definition

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following properties:

- If T is nonempty, it has a special node, called the root of T, that has no parent.
- Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.

According to the definition, a tree can be empty, meaning that it does not have any nodes. This convention also allows us to define a tree recursively such that a tree T is either empty or consists of a node r, called the root of T, and a (possibly empty) set of subtrees whose roots are the children of r.

- Two nodes that are children of the same parent are siblings.
- A node v is external if v has no children.
- A node v is internal if it has one or more children. External nodes are also known as leaves.
- A node *u* is an ancestor of a node *v* if *u* = *v* or *u* is an ancestor of the parent of *v*.

Conversely, we say that a node v is a descendant of a node u if u is an ancestor of v.

The subtree of T rooted at a node v is the tree consisting of all the descendants of v in T (including v itself).



Figure 8.3: Tree representing a portion of a file system.

- An edge of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa.
- A path of *T* is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.



Figure 8.4: A portion of Python's hierarchy of exception types.



Figure 8.5: Our own inheritance hierarchy for modeling various abstractions and implementations of tree data structures. In the remainder of this chapter, we provide implementations of Tree, BinaryTree, and LinkedBinaryTree classes, and highlevel sketches for how LinkedTree and ArrayBinaryTree might be designed.

Definition : Ordered Tree

A tree is ordered if there is a meaningful linear order among the children of each node; that is, we purposefully identify the children of a node as being the first, second, third, and so on. Such an order is usually visualized by arranging siblings left to right, according to their order.



Figure 8.6: An ordered tree associated with a book.

General Trees The Tree Abstract Data Type

We define a tree ADT using the concept of a position as an abstraction for a node of a tree. An element is stored at each position, and positions satisfy parent-child relationships that define the tree structure.

A position object for a tree supports the method:

• p.element(): Return the element stored at position p.

The tree ADT then supports the following accessor methods, allowing a user to navigate the various positions of a tree:

- T.root(): Return the position of the root of tree T or None if T is empty.
- T.is_root(p): Return True if position p is the root of tree T.
- T.parent(p): Return the position of the parent of position p, or None if p is the root of T.
- T.num_chidren(p): Return the number of children of position p.
- T.children(p): Generate an iteration of the children of position p.
- T.is_leaf(): Return True if position p does not have any chidlren.

- len(T): Return the number of positions (and hence elements) that are contained in tree T.
- T.is_empty(): Return True if tree T does not contain any positions.
- T.positions(): Generate an iteration of all positions of T.
- iter(T): Generate an iteration of all elements stored within T.

- len(T): Return the number of positions (and hence elements) that are contained in tree T.
- T.is_empty(): Return True if tree T does not contain any positions.
- T.positions(): Generate an iteration of all positions of T.
- iter(T): Generate an iteration of all elements stored within T.

Any of the above methods that accepts a position as an argument should generate a **ValueError** if that position is invalid for T.

- If T is ordered, then T.chidren(p) reports the chidren of p in the natural order.
- If p is a leaf, then T.chidren(p) generates an empty iteration.
- If T is empty, then both T.positions() and and iter(T) generate empty iterations.

A formal mechanism to designate the relationships between different implementations of the same abstraction is through the definition of one class that serves as an abstract base class, via inheritance, for one or more concrete classes.

A formal mechanism to designate the relationships between different implementations of the same abstraction is through the definition of one class that serves as an abstract base class, via inheritance, for one or more concrete classes.

We choose to define a Tree class that serves as an abstract base class corresponding to the tree ADT.

However, our Tree class does not define any internal representation for storing a tree, and five of the methods given in that code fragment remain abstract (root, parent, num_children, children, and __len__); each of these methods raises a NotImplementedError.

However, our Tree class does not define any internal representation for storing a tree, and five of the methods given in that code fragment remain abstract (root, parent, num_children, children, and __len__); each of these methods raises a NotImplementedError.

The subclasses are responsible for overriding abstract methods, such as children, to provide a working implementation for each behavior, based on their chosen internal representation.

However, our Tree class does not define any internal representation for storing a tree, and five of the methods given in that code fragment remain abstract (root, parent, num_children, children, and __len__); each of these methods raises a NotImplementedError.

The subclasses are responsible for overriding abstract methods, such as children, to provide a working implementation for each behavior, based on their chosen internal representation.

With the **Tree** class being abstract, there is no reason to create a direct instance of it, nor would such an instance be useful. The class exists to serve as a base for inheritance, and users will create instances of concrete subclasses.

```
class Tree:
  class Position:
    def element(self):
      raise NotImplementedError('must be
      implemented by subclass')
    def __eq_(self, other):
      raise NotImplementedError('must be
      implemented by subclass')
    def __ne__(self, other):
      return not (self == other)
```

The Tree Abstract Data Type

```
def root(self):
```

```
raise NotImplementedError('must be
implemented by subclass')
```

```
def parent(self, p):
    raise NotImplementedError('must be
    implemented by subclass')
```

```
def num_children(self, p):
    raise NotImplementedError('must be
    implemented by subclass')
```

```
def children(self, p):
    raise NotImplementedError('must be
    implemented by subclass')
```

```
def __len__(self):
 raise NotImplementedError('must be
  implemented by subclass')
def is root(self, p):
 return self.root() == p
def is leaf(self, p):
 return self.num_children(p) == 0
def is empty(self):
 return len(self) == 0
```

General Trees Computing Depth and Height

Definition : Depth

Let p be the position of a node of T. The depth of p is the number of ancestors of p, excluding p itself.

Note that this definition implies that the depth of the root of T is 0.

The depth of p can also be recursively defined as follows:

- If *p* is the root, then the depth of *p* is 0.
- Otherwise, the depth of *p* is one plus the depth of the pararent of *p*.

The depth of p can also be recursively defined as follows:

- If *p* is the root, then the depth of *p* is 0.
- Otherwise, the depth of *p* is one plus the depth of the pararent of *p*.

```
def depth(self, p):
    if self.is_root(p):
        return 0
    else:
        return 1 + self.depth(self.parent(p))
```
- The running time of T.depth(p) for a position p is $O(d_p+1)$, where d_p denotes the depth of p in T.
- T.depth(p) runs in O(n) worst-case time, where n is the total number of positions of T, because a position of T may have depth n-1 if all nodes form a single branch.

Definition : Height

The height of a position p in a tree T is also defined recursively:

- If p is a leaf, then the height of p is 0.
- Otherwise, the height of p is one more than the maximum of the heights of p' s children.

The height of a nonempty tree T is the height of the root of T.

Property

The height of a nonempty tree T is equal to the maximum of the depths of its leaf positions.

Property

The height of a nonempty tree T is equal to the maximum of the depths of its leaf positions.

We can compute the height of a tree more efficiently, in O(n) worst-case time, by relying instead on the original recursive definition. To do this, we will parameterize a function based on a position within the tree, and calculate the height of the subtree rooted at that position.

We can compute the height of a tree more efficiently, in O(n) worst-case time, by relying instead on the original recursive definition. To do this, we will parameterize a function based on a position within the tree, and calculate the height of the subtree rooted at that position.

```
def height(self, p=None):
    if p is None:
        p = self.root()
    return self._height2(p)
```

Definition : Binary Tree

A binary tree is an ordered tree with following properties:

- 1. Every node has at most two children.
- 2. Each child node is labled as being either a left child or a right child.
- 3. A left child precedes a right child in the order of children of a node.

Definition : Binary Tree

A binary tree is an ordered tree with following properties:

- 1. Every node has at most two children.
- 2. Each child node is labled as being either a left child or a right child.
- 3. A left child precedes a right child in the order of children of a node.
- The subtree rooted at a left or right child of an internal node v is called a left subtree or right subtree, respectively, of v.
- A binary tree is proper if each node has either zero or two children.
 Some people also refer to such trees as being full binary trees. Thus, in a proper binary tree, every internal node has exactly two children.
 A binary tree that is not proper is improper.



Figure 8.7: A decision tree providing investment advice.



Figure 8.8: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2)) - ((3\times (7-4))+6)))$. The value associated with the internal node labeled "/" is 2.

Definition : A Recursive Binary Tree Definition

A binary tree is either empty or consists of

- 1. A node r, called the root of T, that stores an element
- 2. A binary tree (possibly empty), called the left subtree of T
- 3. A binary tree (possibly empty), called the right subtree of T

Binary Tree The Binary Tree Abstract Data Type

As an abstract data type, a binary tree is a specialization of a tree that supports three additional accessor methods:

- T.left(p): Return the position that represents the left child of p, or None if p has no left child.
- T.right(p): Return the position that represents the right child of p, or None if p has no right child.
- T.sibling(p): Return the position that represents the sibling of p, or None if p has no sibling.

```
from tree import Tree
class BinaryTree(Tree):
    def left(self, p):
        raise NotImplementedError('must be
        implemented by subclass')
    def right(self, p):
        raise NotImplementedError('must be
        implemented by subclass')
```

```
def sibling(self, p):
  parent = self.parent(p)
  if parent is None:
    return None
  else:
    if p == self.left(parent):
      return self.right(parent)
    else:
      return self.left(parent)
```

```
def children(self, p):
    if self.left(p) is not None:
        yield self.left(p)
    if self.right(p) is not None:
        yield self.right(p)
```

Binary Tree Properties of Binary Trees

Binary trees have several interesting properties dealing with relationships between their heights and number of nodes.

We denote the set of all nodes of a tree T at the same depth d as level d of T . In a binary tree,

- level 0 has at most one node (the root),
- level 1 has at most two nodes (the children of the root),
- level 2 has at most four nodes,
-

In general, level d has at most 2^d nodes.

Properties of Binary Trees



Property

Let T be a nonempty binary tree, and let n, n_E , n_I and h denote the number of nodes, number of external nodes, number of internal nodes, and height of T, respectively. Then T has the following properties:

- $h+1 \le n \le 2^{h+1}-1$
- $1 \le n_E \le 2^h$
- $h \le n_I \le 2^h 1$
- $\log(n+1) 1 \le h \le n-1$

Property : Continue

If T is proper, then T has the following properties:

- $h+1 \le n \le 2^{h+1}-1$
- $1 \le n_E \le 2^h$
- $h \le n_I \le 2^h 1$
- $\log(n+1) 1 \le h \le n-1$

Property

In a nonempty proper binary tree T, with n_E external nodes and n_I internal nodes, we have $n_E = n_I + 1$.

Implementing Trees

The Tree and BinaryTree classes that we have defined are both formally abstract base classes.

- Neither of them can be directly instantiated.
- Have not yet defined key implementation details for how a tree will be represented internally, and how we can effectively navigate between parents and children.
- A concrete implementation of a tree must provide methods
 - root
 - parent
 - num_children
 - children
 - __len__

and in the case of BinaryTree, the additional accessors

- left
- right

Implementing Trees Linked Structure for Binary Trees

Linked Structure for Binary Trees





A natural way to realize a binary tree T is to use a linked structure, with a node that maintains references

- to the element stored at a position p
- to the nodes associated with the children and parent of *p*.

- If p is the root of T, then the parent field of p is None.
- If p does not have a left child (respectively, right child), the associated field is None.

The tree itself maintains

- an instance variable storing a reference to the root node (if any),
- a variable size that represents the overall number of nodes of T.

T.add_root(e):

Create a root for an empty tree, storing e as the element, and return the position of that root; an error occurs if the tree is not empty.

T.add_left(p, e):

Create a new node storing element e, link the node as the left child of position p, and return the resulting position; an error occurs if p already has a left child.

T.add_right(p, e):

Create a new node storing element e, link the node as the right child of position p, and return the resulting position; an error occurs if p already has a right child.

T.replace(p, e):

Replace the element stored at position p with element e, and return the previously stored element.

T.delete(p):

Remove the node at position p, replacing it with its child, if any, and return the element that had been stored at p; an error occurs if p has two children.

T.attach(p, T1, T2):

Attach the internal structure of trees T1 and T2, respec- tively, as the left and right subtrees of leaf position p of T, and reset T1 and T2 to empty trees; an error condition occurs if p is not a leaf.

```
from binary_tree import BinaryTree
class LinkedBinaryTree(BinaryTree):
 class _Node:
    __slots__ = '_element', '_parent', '_left',
    ' right'
    def __init__(self, element, parent=None,
    left=None, right=None):
      self. element = element
      self._parent = parent
      self. left = left
      self. right = right
```

```
class Position(BinaryTree.Position):
 def init (self, container, node):
    self._container = container
    self. node = node
 def element(self):
    return self._node._element
 def __eq_(self, other):
    return type(other) is type(self) and other
    . node is self. node
```

Linked Structure for Binary Trees

```
def _validate(self, p):
 if not isinstance(p, self.Position):
    raise TypeError('p must be proper Position
    type')
 if p._container is not self:
    raise ValueError('p does not belong to
   this container')
 if p._node._parent is p._node:
   raise ValueError('p is no longer valid')
 return p._node
def make position(self, node):
 return self.Position(self, node) if node is
 not None else None
```
```
def __init__(self):
 self. root = None
 self. size = 0
def len (self):
 return self. size
def root(self):
 return self. make position(self. root)
def parent(self, p):
 node = self._validate(p)
 return self._make_position(node._parent)
```

```
def left(self, p):
    node = self._validate(p)
    return self._make_position(node._left)
def right(self, p):
    node = self._validate(p)
    return self._make_position(node._right)
```

```
def num_children(self, p):
  node = self._validate(p)
  count = 0
  if node._left is not None:
    count += 1
  if node._right is not None:
    count += 1
  return count
```

```
def _add_root(self, e):
    if self._root is not None:
        raise ValueError('Root exists')
    self._size = 1
    self._root = self._Node(e)
    return self._make_position(self._root)
```

```
def _add_left(self, p, e):
  node = self._validate(p)
  if node._left is not None:
    raise ValueError('Left child exists')
  self._size += 1
  node._left = self._Node(e, node)
  return self._make_position(node._left)
```

```
def _add_right(self, p, e):
   node = self._validate(p)
   if node._right is not None:
      raise ValueError('Right child exists')
   self._size += 1
   node._right = self._Node(e, node)
   return self._make_position(node._right)
```

```
def _replace(self, p, e):
   node = self._validate(p)
   old = node._element
   node._element = e
   return old
```

```
def _delete(self, p):
  node = self._validate(p)
  if self.num_children(p) == 2:
    raise ValueError('Position has two
    children')
  child = node._left if node._left else node.
  _right
  if child is not None:
    child._parent = node._parent
```

```
if node is self._root:
    self._root = child
else:
    parent = node._parent
    if node is parent._left:
        parent._left = child
    else:
        parent._right = child
```

Linked Structure for Binary Trees

```
def _attach(self, p, t1, t2):
 node = self._validate(p)
 if not self.is leaf(p):
    raise ValueError('position must be leaf')
 if not type(self) is type(t1) is type(t2):
    raise TypeError('Tree types must match')
 self._size += len(t1) + len(t2)
 if not t1.is empty():
   t1._root._parent = node
   node._left = t1._root
   t1. root = None
   t1. size = 0
 if not t2.is_empty():
   t2._root._parent = node
   node._right = t2._root
   t2._root = None
   t2. size = 0
```

Implementing Trees 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.

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.

Linked Structure for General Trees



Figure 8.14: The linked structure for a general tree: (a) the structure of a node; (b) a larger portion of the data structure associated with a node and its children.

Operation	Running Time
len, is_empty	O(1)
root, parent, is_root, is_leaf	<i>O</i> (1)
children(p)	$O(c_p + 1)$
depth(p)	$O(d_p + 1)$
height	O(n)

Table 8.2: Running times of the accessor methods of an *n*-node general tree implemented with a linked structure. We let c_p denote the number of children of a position *p*. The space usage is O(n).

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 increment- ing a counter to performing some complex computation for p.

Tree Traversal Algorithms Preorder and Postorder Traversals of General Trees

Preorder and Postorder Traversals 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.



Figure 8.15: Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

```
class Tree:
```

```
def preorder(self):
 if not self.is empty():
    for p in self._subtree_preorder(self.root
    ()):
     yield p
def subtree preorder(self, p):
 yield p
 for c in self.children(p):
    for other in self._subtree_preorder(c):
      yield other
```

```
class Tree:
```

```
def postorder(self):
 if not self.is empty():
    for p in self._subtree_postorder(self.root
    ()):
      yield p
def subtree postorder(self, p):
 for c in self.children(p):
    for other in self._subtree_postorder(c):
      yield other
 yield p
```

Both preorder and postorder traversal algorithms are efficient ways to access all the positions of a tree.

Both preorder and postorder traversal algorithms are efficient ways to access all the positions of a tree.

At each position p, the nonrecursive part of the traversal algorithm requires time $O(c_p + 1)$, where c_p is the number of children of p, under the assumption that the "visit" itself takes O(1) time. Both preorder and postorder traversal algorithms are efficient ways to access all the positions of a tree.

At each position p, the nonrecursive part of the traversal algorithm requires time $O(c_p + 1)$, where c_p is the number of children of p, under the assumption that the "visit" itself takes O(1) time.

The overall running time for the traversal of tree T is O(n), where n is the number of positions in the tree. This running time is asymptotically optimal since the traversal must visit all the *n* positions of the tree.

Tree Traversal Algorithms Breadth-First Tree Traversal

Definition : Breadth-First Tree Traversal

Although the preorder and postorder traversals are common ways of visiting the positions of a tree, another common 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.

Breadth-First Tree Traversal



Figure 8.17: Partial game tree for Tic-Tac-Toe, with annotations displaying the order in which positions are visited in a breadth-first traversal.

class Tree:

```
def breadthfirst(self):
    if not self.is_empty():
        fringe = LinkedQueue()
        fringe.enqueue(self.root())
        while not fringe.is_empty():
        p = fringe.dequeue()
        yield p
        for c in self.children(p):
            fringe.enqueue(c)
```

Tree Traversal Algorithms Inorder Traversal of 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".

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.

Inorder Traversal of a Binary Tree



Figure 8.18: Inorder traversal of a binary tree.

Inorder Traversal of a Binary Tree

class BinaryTree(Tree):

```
def inorder(self):
  if not self.is_empty():
    for p in self. subtree inorder(self.root()
    ):
      yield p
def _subtree_inorder(self, p):
  if self.left(p) is not None:
    for other in self._subtree_inorder(self.
    left(p)):
      yield other
  yield p
  if self.right(p) is not None:
    for other in self._subtree_inorder(self.
    right(p)):
      vield other
```

Expression Tree

from linked_binary_tree import LinkedBinaryTree

```
class ExpressionTree(LinkedBinaryTree):
  def __init__(self, token, left=None, right=
 None):
    super(). init ()
    if not isinstance(token, str):
      raise TypeError('Token must be a string')
    self. add root(token)
    if left is not None:
      if token not in '+-*x/':
        raise ValueError('token must be valid
        operator')
      self._attach(self.root(), left, right)
```

```
def parenthesize recur(self, p, result):
 if self.is leaf(p):
    result.append(str(p.element()))
 else:
    result.append('(')
    self._parenthesize_recur(self.left(p),
    result)
    result.append(p.element())
    self._parenthesize_recur(self.right(p),
    result)
    result.append(')')
```
```
def __str__(self):
   pieces = []
   self._parenthesize_recur(self.root(), pieces
   )
   return ''.join(pieces)
```

def evaluate(self):
 return self._evaluate_recur(self.root())

Expression Tree

```
def _evaluate_recur(self, p):
 if self.is_leaf(p):
   return float(p.element())
 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
   elif op == '/':
     return left_val / right_val
   else:
     return left val * right val
```

Expression Tree

```
def tokenize(raw):
 SYMBOLS = set ('+-x*/()')
 mark = 0
 tokens = []
n = len(raw)
 for j in range(n):
   if raw[j] in SYMBOLS:
     if mark != j:
       tokens.append(raw[mark:j])
     if raw[j] != ' ':
       tokens.append(raw[j])
     mark = j+1
 if mark != n:
   tokens.append(raw[mark:n])
 return tokens
```

```
def build_expression_tree(tokens):
 S = []
 for t in tokens:
   if t in '+-x*/':
     S.append(t)
   elif t not in '()':
     S.append(ExpressionTree(t))
   elif t == ')':
     right = S.pop()
     op = S.pop()
     left = S.pop()
     S.append(ExpressionTree(op, left, right))
 return S.pop()
```