Data Structures in Python

9. Trees (I)

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

Introduction

- In this lecture, we discuss one of the most important ***nonlinear data structures*** in computing—***trees***.
- 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.
- Actually, the main terminology for tree data structures comes from family trees, where the terms "***parent***," "***child***," "***ancestor***," and "***descendant***" are used to describe relationships.

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.
- The top element of the tree is called the ***root***.

➢ ***Formal Tree Definition***

- Formally, a ***tree T*** *is* defined as 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*.

Tree Definitions and Properties

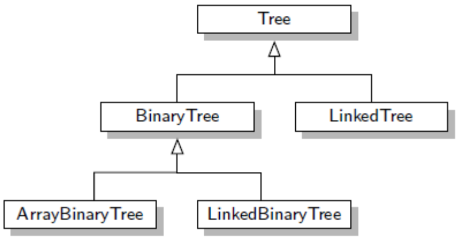
- A tree can be empty, meaning that it does not have any nodes.
- A tree can be defined recursively such that a tree *T* is either empty or consists of a node *r*, called the *root* of *T*, and a (possibly empty) *set of subtrees* whose roots are the children of *r*.

➢ ***Other Node Relationships***

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

Tree Definitions and Properties

- The ***subtree*** of *T* rooted at a node *v* is the tree consisting of all the descendants of *v* in *T* (including *v* itself).
- 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***.
- As a preview of the remainder of this lecture, the figure portrays a hierarchy of classes for representing various forms of a tree.
- We will provide implementations of Tree, ***BinaryTree***, and ***LinkedBinaryTree*** classes, and Highlevel sketches for how ***LinkedTree*** and ***ArrayBinaryTree*** might be designed.



Tree Definitions and Properties

➢ ***Ordered Trees***

- A tree is ***ordered*** if there is a meaningful linear order among the ***children*** of each node; that is, we 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.

The Tree Abstract Data Type

- We will 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 p*** for a tree supports the method:
  p.element( ) **:** Return the element stored at position ***p***.

The Tree Abstract Data Type

- 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_children(p): Return the number of children of position *p*.

T.children(p): Generate an iteration of the children of position *p*.

T.is_leaf(p): Return *True* if position p does not have any children.

T.len(): 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 tree T.

T.iter( ): Generate an iteration of all *elements* stored within tree T.

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

A Tree Abstract Base Class in Python

- We will define a ***Tree class*** that serves as an abstract base class corresponding to the ***tree ADT***.
- The ***Tree class*** provides a definition of a nested ***Position class*** (which is also ***abstract***), and declarations of many of the ***accessor*** methods included in the ***tree ADT***.
- The ***Tree class*** does not define any internal representation for storing a tree, and five of its methods 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.
- Although the ***Tree class*** is an ***abstract base class***, it includes several ***concrete*** methods with implementations that rely on calls to the ***abstract*** methods of the class.

A Tree Abstract Base Class in Python

- The code of the Tree abstract base class

```
class Tree:
 """Abstract base class representing a tree structure."""
 #---------------------------- nested Position class -----------------------------
 class Position:
  """An abstraction representing the location of a single
  element."""
  def element(self):
   """Return the element stored at this Position."""
   raise NotImplementedError('must be implemented by subclass')
  def __eq__(self, other):
   """Return True if other Position represents the same
   location."""
   raise NotImplementedError('must be implemented by subclass')
  def __ne__(self, other):
   """Return True if other does not represent the same location."""
   return not (self == other) # opposite of eq
```


A Tree Abstract Base Class in Python

```
 # --- abstract methods that concrete subclass must support -----
 def root(self):
  """Return Position representing the tree s root (or None if
  empty)."""
  raise NotImplementedError('must be implemented by subclass')
 def parent(self, p):
  """Return Position representing p s parent (or None if p is
  root)."""
  raise NotImplementedError('must be implemented by subclass')
 def num_children(self, p):
  """Return the number of children that Position p has."""
  raise NotImplementedError('must be implemented by subclass')
 def children(self, p):
  """Generate an iteration of Positions representing p s
  children."""
  raise NotImplementedError('must be implemented by subclass')
 def len(self):
  """Return the total number of elements in the tree."""
  raise NotImplementedError('must be implemented by subclass')
```

A Tree Abstract Base Class in Python

```
# ---------- concrete methods implemented in this class ----------
def is_root(self, p):
 """Return True if Position p represents the root of the tree."""
 return self.root() == p
def is_leaf(self, p):
 """Return True if Position p does not have any children."""
 return self.num_children(p) == 0
def is_empty(self):
 """Return True if the tree is empty."""
 return self.len() == 0
```

- ***Computing Depth and Height***
- Let *p* be the position of a node of a tree *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.

A Tree Abstract Base Class in Python

- The ***depth*** of *p* can 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 ***parent*** of *p*.
- Based on this definition, we present a simple, recursive algorithm, ***depth***, for computing the ***depth*** of a position *p* in Tree *T*.

```
def depth(self, p):
 """Return the number of levels separating Position p from the root."""
 if self.is_root(p):
  return 0
 else:
  return 1 + self.depth(self.parent(p))
```

- The ***height*** of a position ***p*** in a tree *T* is defined as *the length of the longest path to a leaf under p*.

A Tree Abstract Base Class in Python

- The ***height*** of a position *p* in a tree *T* can be defined recursively as follows:
  - 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*.
- Based on this recursive definition, we can compute the ***height*** of a tree as follows:
  - Parameterize a function based on a position within the tree, and calculate the height of the subtree rooted at that position.
- Algorithm ***height***, shown below as nonpublic method ***_height***, computes the height of tree *T* in this way.

A Tree Abstract Base Class in Python

- Method ***_height*** for computing the height of a subtree rooted at a position *p* of a Tree:

```
def _height(self, p): # time is linear in size of subtree
  """Return the height of the subtree rooted at Position p."""
  if self.is_leaf(p):
   return 0
  else:
   return 1 + max(self._height(c) for c in self.children(p))
```

- The ability to compute heights of subtrees is beneficial, but a user might expect to be able to compute the height of the entire tree without explicitly designating the tree root.
- We can wrap the nonpublic ***_height*** in our implementation with a public ***height*** method that provides a default interpretation when invoked on tree *T* with syntax ***T.height()***.

A Tree Abstract Base Class in Python

- Public method ***Tree.height*** that computes the height of the entire tree by default, or a subtree rooted at given position, if specified.

```
def height(self, p=None):
 """Return the height of the subtree rooted at Position p.
    If p is None, return the height of the entire tree.   """
 if p is None:
  p = self.root( )
 return self._height(p) # start height2 recursion
```


Binary Trees

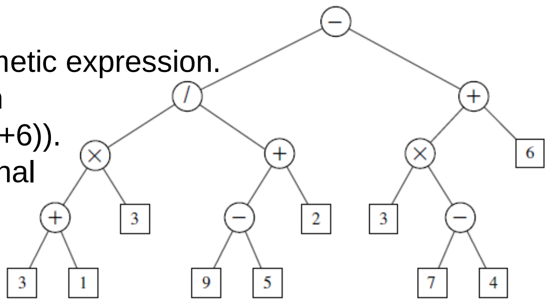
- A ***binary tree*** is an ***ordered tree*** with the following properties:
  1. Every node has at most two children.
  2. Each child node is labeled as being either a ***left child*** or a ***right child***.
  3. A left child precedes a right child in the order of children of a node.
- The subtree rooted at a left or right child of an internal node *v* is called a ***left subtree*** or ***right subtree*** of *v*, respectively.
- 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.

Binary Trees

- Thus, in a ***proper*** binary tree, every ***internal*** node has exactly two children.
- A binary tree that is not proper is ***improper***.
- **Example:** An arithmetic expression can be represented by a binary tree whose leaves are associated with variables or constants, and whose internal nodes are associated with one of the operators +, −, ×, and /. (See the figure)
- Each node in such a tree has a value associated with it.
  - If a node is ***leaf***, then its value is that of its variable or constant.
  - If a node is ***internal***, then its value is defined by applying its operation to the values of its children.
- An arithmetic expression tree is a *proper binary tree*, since each operator +, −, ×, and / takes exactly two operands.
- Of course, if we were to allow unary operators, like negation (−), as in "−*x*," then we could have an *improper binary tree*.

Binary Trees

A binary tree representing an arithmetic expression. This tree represents the expression ((((3 +1)×3)/((9 −5)+2))−((3×(7−4)) +6)). The value associated with the internal node labeled "/" is 2.



➢ *A Recursive Binary Tree Definition*

- We can define a binary tree in a recursive way such that a binary tree is either empty or consists of:
  - A node ***r***, called the ***root*** of *T*, that stores an element
  - A binary tree (possibly empty), called the ***left subtree*** of *T*
  - A binary tree (possibly empty), called the ***right subtree*** of *T*

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.

The BinaryTree Abstract Base Class in Python

- We will define a new ***BinaryTree class*** associated with the binary tree ADT based upon the existing Tree class.
- ***BinaryTree class*** remains ***abstract***, as we still do not provide complete specifications for how such a structure will be represented internally, nor implementations for some necessary behaviors.

The BinaryTree Abstract Base Class in Python

- By using inheritance, a binary tree supports all the functionality that was defined for general trees (e.g., ***parent***, ***is_leaf***, ***root***).
- The new class also inherits the nested ***Position class*** that was originally defined within the ***Tree class*** definition.
- In addition, the new class provides declarations for new abstract methods ***left*** and ***right*** that should be supported by concrete subclasses of ***BinaryTree***.
- The new class also provides two concrete implementations of methods: ***sibling*** and ***children***.
- The new ***sibling*** method is derived from the combination of ***left***, ***right***, and ***parent***.
- The sibling of a position *p* is identified as the "other" child of *p*'s parent.
- However, if *p* is the root, it has no parent, and thus no sibling.

The BinaryTree Abstract Base Class in Python

- Also, *p* may be the only child of its parent, and thus does not have a sibling.
- *The new **children** method*: Although we have still not specified how the children of a node will be stored, we derive a generator for the ordered children based upon the implied behavior of abstract methods ***left*** and ***right***.
- The code for the ***BinaryTree*** abstract base class that extends the existing ***Tree*** abstract base class

```
class BinaryTree(Tree):
 """Abstract base class representing a binary tree structure."""
 # -------------------- additional abstract methods --------------------
 def left(self, p):
  """Return a Position representing p s left child.
     Return None if p does not have a left child.  """
  raise NotImplementedError('must be implemented by subclass')
 def right(self, p):
  """Return a Position representing p s right child.
     Return None if p does not have a right child.  """
  raise NotImplementedError('must be implemented by subclass')
```

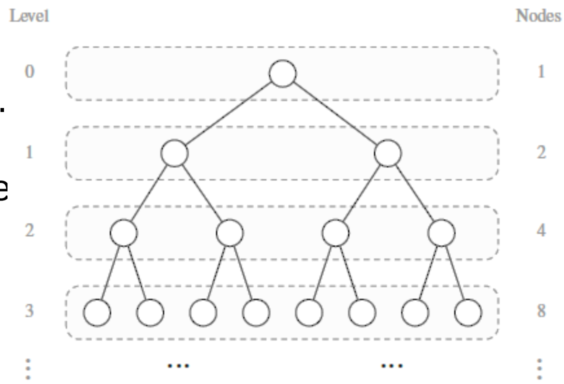
The BinaryTree Abstract Base Class in Python

```
 # ---------- concrete methods implemented in this class ----------
 def sibling(self, p):
  """Return a Position representing p s sibling (or None if no sibling)."""
  parent = self.parent(p)
  if parent is None: # p must be the root
   return None # root has no sibling
  else:
   if p == self.left(parent):
    return self.right(parent) # possibly None
   else:
    return self.left(parent) # possibly None

 def children(self, p):
  """Generate an iteration of Positions representing p s children."""
  if self.left(p) is not None:
   yield self.left(p)
  if self.right(p) is not None:
   yield self.right(p)
```

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*.
- The maximum level in a tree determines its height.
- 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, and so on, as shown in the figure.

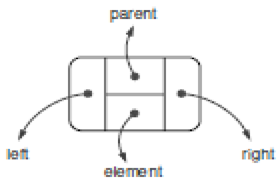


- In general, level *d* has at most 2^d nodes. Often, however, levels do not contain the maximum number of nodes.

Implementing Trees

- The ***Tree*** and ***BinaryTree*** classes that we have defined are both ***abstract base classes***.
- Although they provide a great deal of support, neither of them can be directly instantiated.
- We 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.
- Specifically, 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*** and ***right***.
- We begin with the case of a ***binary tree***, since its shape is more narrowly defined.

Linked Structure for Binary Trees

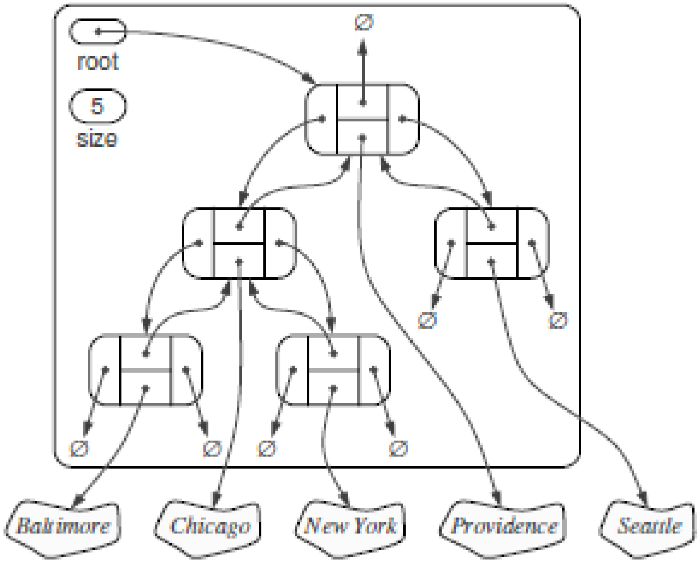


(a)

- 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* and to the nodes associated with the children and parent of *p*. (See Figure (a))
- If *p* is the root of *T*, then the parent field of *p* is None. Likewise, 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), and a variable, called size, that represents the overall number of nodes of *T*.
- Such a linked structure representation of a binary tree is shown in Figure (b).

Linked Structure for Binary Trees

A linked structure for representing a binary tree.



(b)

Python Implementation of a Linked Binary Tree Structure

- Now, we define a concrete ***LinkedBinaryTree*** class that implements the ***binary tree ADT*** by subclassing the ***BinaryTree*** class.
- We define a simple, nonpublic ***_Node*** class to represent a node, and a public ***Position*** class that wraps a node.
- We provide:
  - ***_validate*** utility for robustly checking the validity of a given position instance when unwrapping it, and
  - ***_make_position*** utility for wrapping a node as a position to return to a caller.
- The new ***Position*** class is declared to inherit immediately from ***BinaryTree.Position***.
- Note that the ***BinaryTree*** class definition does not declare such a nested class; it inherits it from ***Tree.Position***.
- The ***LinkedBinaryTree*** class definition continues with a ***constructor*** and concrete implementations for the methods that remain abstract in the ***Tree*** and ***BinaryTree*** classes.

Python Implementation of a Linked Binary Tree Structure

- The constructor creates an empty tree by initializing ***_root*** to ***None*** and ***_size*** to zero.
- The accessor methods are implemented with careful use of the ***_validate*** and ***_make_position*** utilities to safeguard against boundary cases.

➢ ***Operations for Updating a Linked Binary Tree***

- Thus far, we have provided functionality for examining an existing binary tree.
- However, the constructor for the ***LinkedBinaryTree*** class results in an empty tree.
- For linked binary trees, a reasonable set of update methods to support for general usage are the following:
  - ***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.

Python Implementation of a Linked Binary Tree Structure

- ***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, respectively, 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.

Python Implementation of a Linked Binary Tree Structure

- The code for ***LinkedBinaryTree*** class:

```
class LinkedBinaryTree(BinaryTree):
 """Linked representation of a binary tree structure."""
 class _Node: # Lightweight, nonpublic class for storing a 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):
  """An abstraction representing the location of a single element."""
  def __init__(self, container, node):
   """Constructor should not be invoked by user."""
   self._container = container
   self._node = node
  def element(self):
   """Return the element stored at this Position."""
   return self._node._element
```

Python Implementation of a Linked Binary Tree Structure

```
def __eq__(self, other):
 """Return True if other is a Position representing the same location."""
 return type(other) is type(self) and other._node is self._node

def _validate(self, p):
 """Return associated node, if position is valid."""
 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: # convention for deprecated nodes
  raise ValueError('p is no longer valid')
 return p._node
def _make_position(self, node):
 """Return Position instance for given node (or None if no node)."""
 return self.Position(self, node) if node is not None else None
#------------------------- binary tree constructor -------------------------
def __init__(self):
 """Create an initially empty binary tree."""
 self._root = None
 self._size = 0
```


Python Implementation of a Linked Binary Tree Structure

```
#------------------------- public accessors -------------------------
def len(self):
 """Return the total number of elements in the tree."""
 return self._size
def root(self):
 """Return the root Position of the tree (or None if tree is empty)."""
 return self._make_position(self._root)
def parent(self, p):
 """Return the Position of p's parent (or None if p is root)."""
 node = self._validate(p)
 return self._make_position(node._parent)
def left(self, p):
 """Return the Position of p's left child (or None if no left child)."""
 node = self._validate(p)
 return self._make_position(node._left)
def right(self, p):
 """Return the Position of p's right child (or None if no right child)."""
 node = self._validate(p)
 return self._make_position(node._right)
```

Python Implementation of a Linked Binary Tree Structure

```
def num_children(self, p):
 """Return the number of children of Position p."""
 node = self._validate(p)
 count = 0
 if node._left is not None: # left child exists
  count += 1
 if node._right is not None: # right child exists
  count += 1
 return count

def _add_root(self, e):
 """Place element e at the root of an empty tree and return new Position.
    Raise ValueError if tree nonempty."""
 if self._root is not None: raise ValueError('Root exists')
 self._size = 1
 self._root = self._Node(e)
 return self._make_position(self._root)
```


Python Implementation of a Linked Binary Tree Structure

```
def _add_left(self, p, e):
 """Create a new left child for Position p, storing element e.
 Return the Position of new node.
 Raise ValueError if Position p is invalid or p already has a left child. """
 node = self._validate(p)
 if node._left is not None: raise ValueError('Left child exists')
 self._size += 1
 node._left = self._Node(e, node) # node is its parent
 return self._make_position(node._left)

def _add_right(self, p, e):
 """Create a new right child for Position p, storing element e.
    Return the Position of new node.
 Raise ValueError if Position p is invalid or p already has a right child. """
 node = self._validate(p)
 if node._right is not None: raise ValueError('Right child exists')
 self._size += 1
 node._right = self._Node(e, node) # node is its parent
 return self._make_position(node._right)
```

Python Implementation of a Linked Binary Tree Structure

```
def _replace(self, p, e):
 """Replace the element at position p with e, and return old element."""
 node = self._validate(p)
 old = node._element
 node._element = e
 return old

def _delete(self, p):
 """Delete the node at Position p, and replace it with its child, if any.
   Return the element that had been stored at Position p.
 Raise ValueError if Position p is invalid or p has two children."""
 node = self._validate(p)
 if self.num_children(p) == 2: raise ValueError('p has two children')
 child = node._left if node._left else node._right # might be None
 if child is not None:
  child._parent = node._parent # child s grandparent becomes parent
 if node is self._root:
  self._root = child # child becomes root
 else:
  parent = node._parent
```

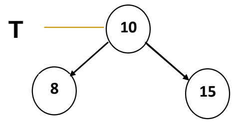

Python Implementation of a Linked Binary Tree Structure

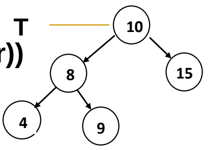
```
  if node is parent._left:
   parent._left = child
  else:
   parent._right = child
 self._size -= 1
 node._parent = node # convention for deprecated node
 return node._element
def _attach(self, p, t1, t2):
 """Attach trees t1 and t2 as left and right subtrees of external p."""
 node = self._validate(p)
 if not self.is_leaf(p): raise ValueError('position must be leaf')
 if not type(self) is type(t1) is type(t2): # all 3 trees must be same type
  raise TypeError('Tree types must match')
 self._size += t1.len() + t2.len()
 if not t1.is_empty( ): # attached t1 as left subtree of node
  t1._root._parent = node
  node._left = t1._root
  t1._root = None # set t1 instance to empty
  t1._size = 0
 if not t2.is_empty( ): # attached t2 as right subtree of node
  t2._root._parent = node
  node._right = t2._root
  t2._root = None # set t2 instance to empty
  t2._size = 0
```

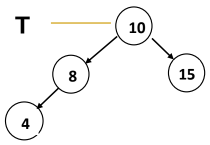
Python Implementation of a Linked Binary Tree Structure

- The following code exercises the operations of the ***LinkedBinaryTree*** class:

```
T = LinkedBinaryTree()
root = T._add_root(10)
l = T._add_left(root, 8)
r = T._add_right(root, 15)
print("Tree Size: ", T.len())
print("Num of root\'s children:", T.num_children(root))
print("Root element: ", T.root().element())
print("Root left element: ", T.left(root).element())
print("Root right element: ", T.right(root).element())
print("Num of children of root left child:", T.num_children(l))
print("Num of children of root right child:", T.num_children(r))
ll = T._add_left(l, 4)
lr = T._add_right(l, 9)
print("Tree Size: ", T.len())
print("Root left left element: ", T.left(l).element())
print("Root left right element: ", T.right(l).element())
print("Deleted Root left right element: ", T._delete(lr))
print("Tree Size: ", T.len())
old = T._replace(root, 12)
print("Root element:", old, " replaced by:", T.root().element())
```

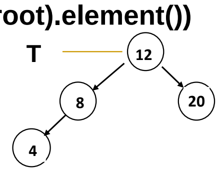


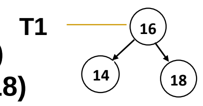


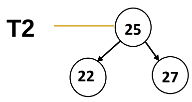


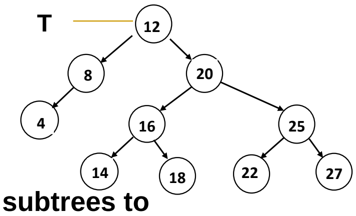
Python Implementation of a Linked Binary Tree Structure

```
old_r = T._replace(r, 20)
print("Root right element:", old_r, " replaced by:", T.right(root).element())
# create new tree with root 16, left 14, right 18
T1 = LinkedBinaryTree()
root1 = T1._add_root(16)
l1 = T1._add_left(root1, 14)
r1 = T1._add_right(root1, 18)
# create new tree with root 25, left 22, right 27
T2 = LinkedBinaryTree()
root2 = T2._add_root(25)
l2 = T2._add_left(root2, 22)
r2 = T2._add_right(root2, 27)
# attach the new trees T1 and T2 as left and right subtrees to
# right node of original tree T
T._attach(T.right(root), T1, T2)
print("Tree Size after adding the new subtrees: ", T.len())
print("Root right left element: ", T.left(r).element())
print("Root right left left element: ", T.left(T.left(r)).element())
print("Root right left right element: ", T.right(T.left(r)).element())
print("Root right right element: ", T.right(r).element())
print("Root right right left element: ", T.left(T.right(r)).element())
print("Root right right right element: ", T.right(T.right(r)).element())
```









Python Implementation of a Linked Binary Tree Structure

Output

```
Tree Size:  3
Num of root's children: 2
Root element:  10
Root left element:  8
Root right element:  15
Num of children of root left child: 0
Num of children of root right child: 0
Tree Size after adding two children to the root left node:  5
Root left left element:  4
Root left right element:  9
Deleted Root left right element:  9
Tree Size after deleting Root left right element:  4
Root element: 10  replaced by: 12
Root right element: 15  replaced by: 20
Tree Size after adding the new subtrees:  10
Root right left element:  16
Root right left left element:  14
Root right left right element:  18
Root right right element:  25
Root right right left element:  22
Root right right right element:  27
```

39