```
from binary_tree import BinaryTree

class LinkedBinaryTree(BinaryTree):
  """Linked representation of a binary tree structure."""

  #-------------------------- nested _Node class --------------------------
  class _Node:
    """Lightweight, nonpublic class for storing a node."""
    __slots__ = '_element', '_parent', '_left', '_right' # streamline memory
usage

    def __init__(self, element, parent=None, left=None, right=None):
      self._element = element
      self._parent = parent
      self._left = left
      self._right = right

  #-------------------------- nested Position class --------------------------
  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

    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

  #------------------------------ utility methods ------------------------------
--
  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      raise ValueError('p is no longer valid')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
```

```
  #---------------------------- 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)

  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

  #---------------------------- nonpublic mutators ----------------------------
  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)

  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)

  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('Position 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
      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, respectively, as the left and right subtrees of
the external Position p.

    As a side effect, set t1 and t2 to empty.
    Raise TypeError if trees t1 and t2 do not match type of this tree.
    Raise ValueError if Position p is invalid or not external.
    """
    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 += len(t1) + len(t2)
    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

As a side effect, set t1 and t2 to empty.
Raise TypeError if trees t1 and t2 do not match type of this tree.
Raise ValueError if Position p is invalid or not external.
"""
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 += len(t1) + len(t2)
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
```