```
from linked_queue import LinkedQueue
import collections

class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -----------------------
--------
  class Position:
    """An abstraction representing the location of a single element within a
tree.

    Note that two position instaces may represent the same inherent location in
a tree.
    Therefore, users should always rely on syntax 'p == q' rather than 'p is q'
when testing
    equivalence of positions.
    """

    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__

  # ---------- 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')

  # ---------- 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 len(self) == 0

  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))

  def _height1(self):                 # works, but O(n^2) worst-case time
    """Return the height of the tree."""
    return max(self.depth(p) for p in self.positions() if self.is_leaf(p))

  def _height2(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._height2(c) for c in self.children(p))

  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._height2(p)       # start _height2 recursion

  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

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

  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

  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                                         # visit p after its
subtrees

  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
```