Week 6: Trees

General Trees

树是非线性数据结构，在许多情况下，它允许比线性结构更快地进行数据操作。树中的关系是分层的

树定义:

树是一种抽象数据类型，它按层次hierarchically结构存储元素。 树中的每个元素（根除外）都有一个父元素和零个或多个子元素。父级和子级通常使用线连接，祖先在后代之上。 我们通常将 top 元素称为树的根，但它被绘制为最高的元素 正式定义：树 T 是一组存储元素的节点，这些节点具有满足以下属性的父子关系： • 如果 T 为非空，则它具有一个特殊节点，称为 T 的根 ，该节点没有父项。• T 的每个节点 v 与根不同，都有一个唯一的父节点 w;具有父节点 W 的每个节点都是 W 的子节点。根据这个定义，一棵树可以是空的，这意味着它没有任何节点

其他节点关系

同级节点Siblings：作为同一父节点的子节点的两个节点是同级节点。 外部节点External node：如果 v 没有子节点，则节点 v 为外部节点。外部节点也称为叶子。 内部节点Internal node：如果节点 v 具有一个或多个子节点，则该节点 v 为 internal 节点

文件树:

文件系统的树表示 树的内部节点与目录相关联，叶子与常规文件相关联。在 UNIX 和 Linux 操作系统中，树的根被恰当地称为 "根目录"，并由符号 "/" 表示。

Ancestors and Descendants：

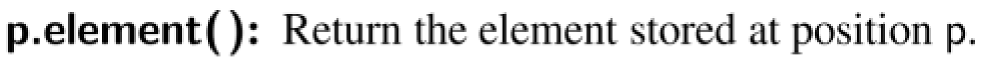
Ancestor：如果 u = v 或 u 是 v 父级的上级，则节点 u 是节点 v 的上级Ancestor。 Descendants后代：如果您是 v 的上级，则节点 v 是节点 u 的后代。 Subtree子树：以节点 v 为根的 T 子树是由 T 中 v 的所有后代（包括 v 本身）组成的树。 edge边：树 T 的边是一对节点 （u，v），使得 u 是 v 的父节点，反之亦然。 path路径：T 的路径是一个节点序列，使得序列中的任意两个连续节点形成一条边

Ordered Trees有序树

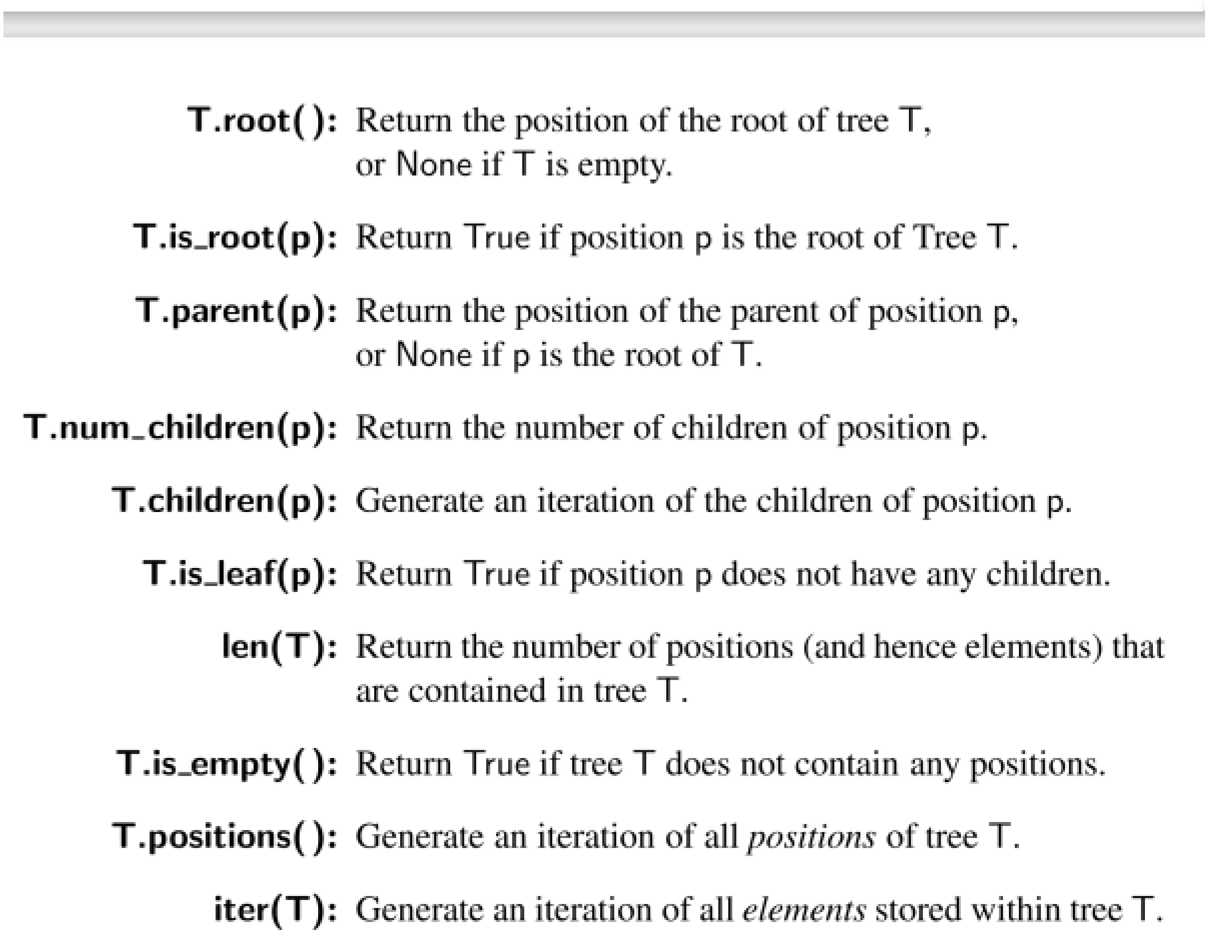
如果每个节点的子节点之间存在有意义的线性顺序，则树是有序的;也就是说，我们有意将节点的子节点标识为 first、second、third 等

树的抽象类型:

我们使用位置的概念作为树节点的抽象来定义树 ADT。每个位置都存储了一个元素，位置满足定义树结构的父



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

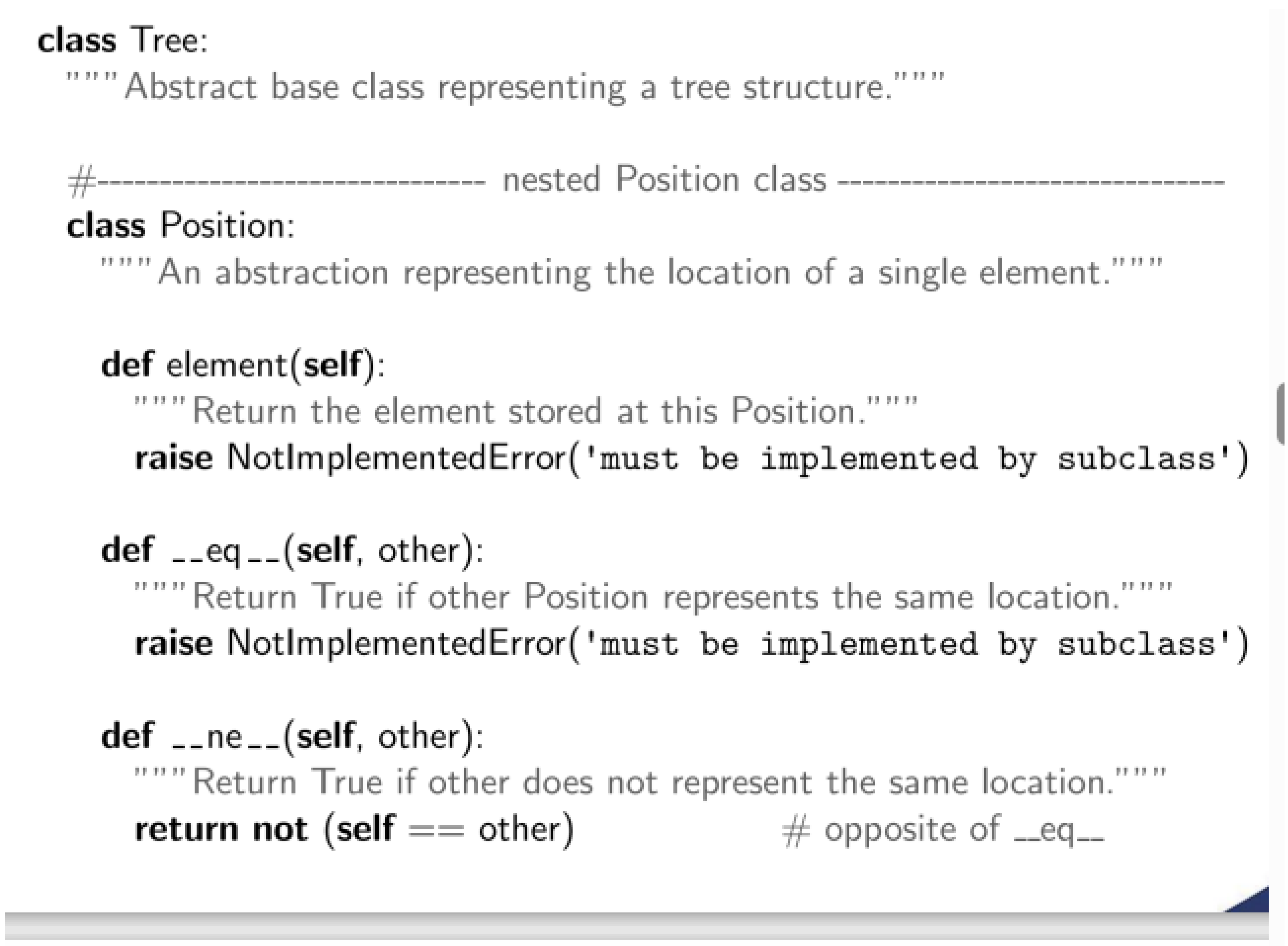


T.root( ):	Return the position of the root of tree T, or None if T is empty.
T.is_root(p):	Return True if position p is the root of Tree T.
T.parent(p):	Return the position of the parent of position p, or None if p is the root of T.
T.num_children(p):	Return the number of children of position p.
T.children(p):	Generate an iteration of the children of position p.
T.is_leaf(p):	Return True if position p does not have any children.
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 tree T.
iter(T):	Generate an iteration of all *elements* stored within tree T.

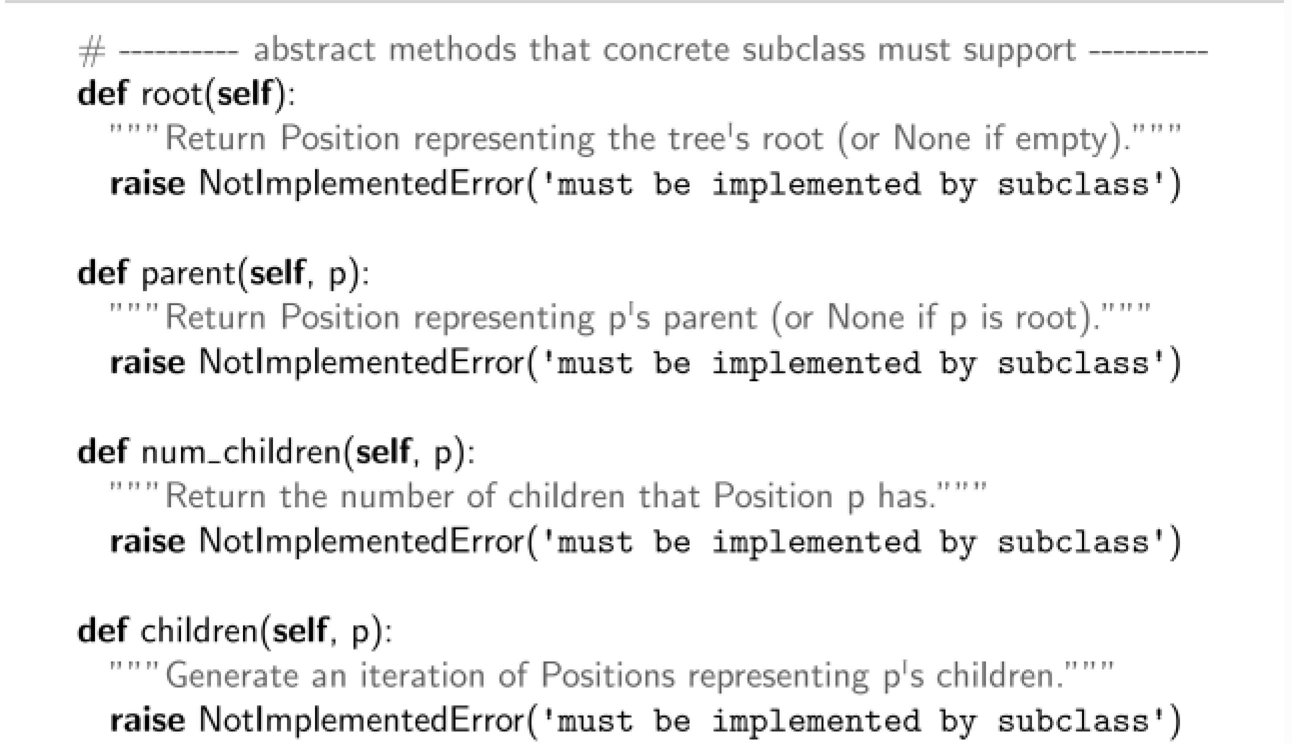
子关系。 位置position是对树的一次包装

树的抽象基类：

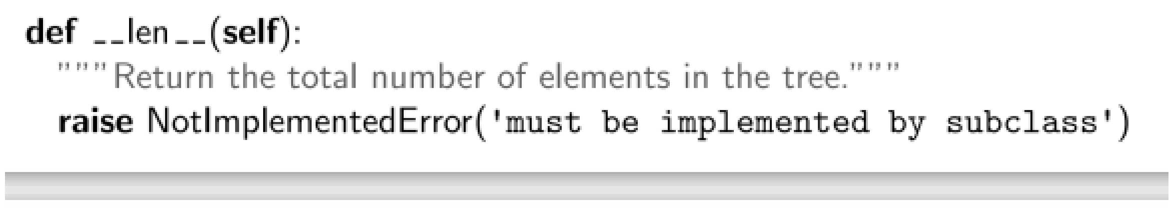
指定同一抽象的不同实现之间关系的正式机制是通过继承定义一个类，该类充当一个或多个具体类的抽象基类。 我们选择定义一个 Tree 类，它用作对应于树 ADT 的抽象基类。 但是，我们的 Tree 类没有定义用于存储树的任何内部表示，并且该代码片段中给出的 5 个方法仍然是抽象的。 尽管 Tree 类是一个抽象基类，但它包含多个具体方法，这些方法的实现依赖于对类的抽象方法的调用



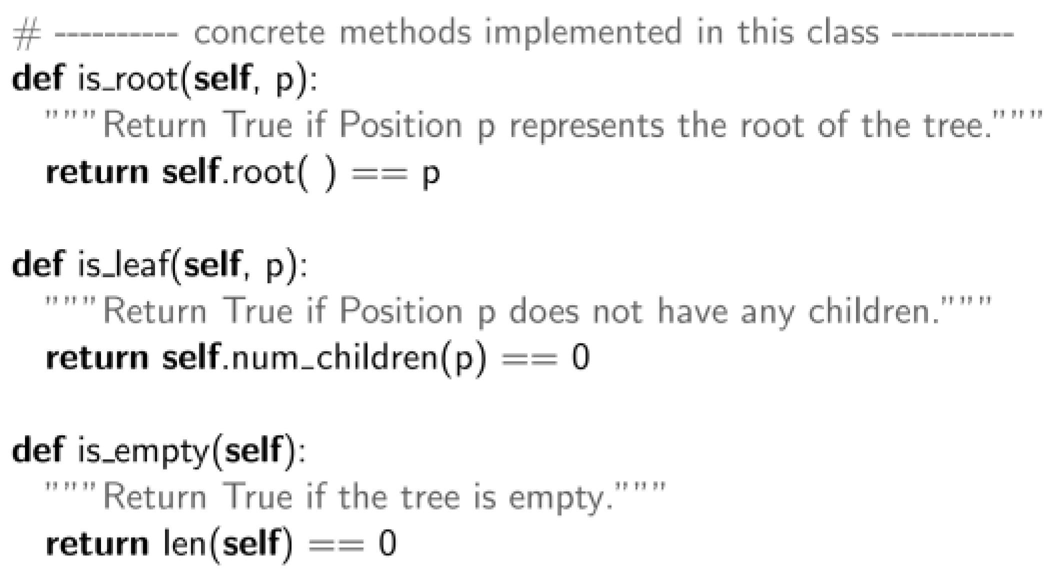
```python
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- 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__
```



```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')
```



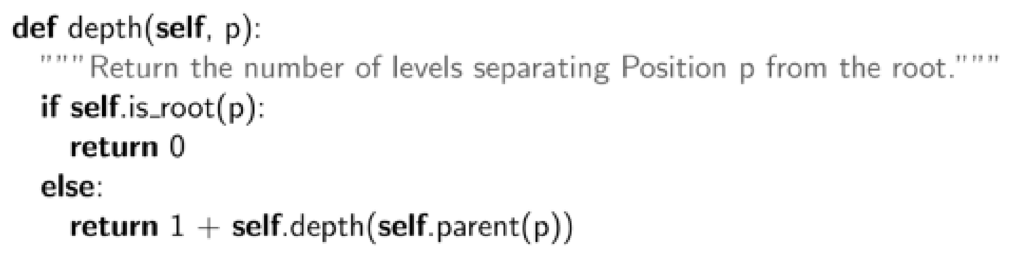
```
def __len__(self):
  """Return the total number of elements in the tree."""
  raise NotImplementedError('must be implemented by subclass')
```



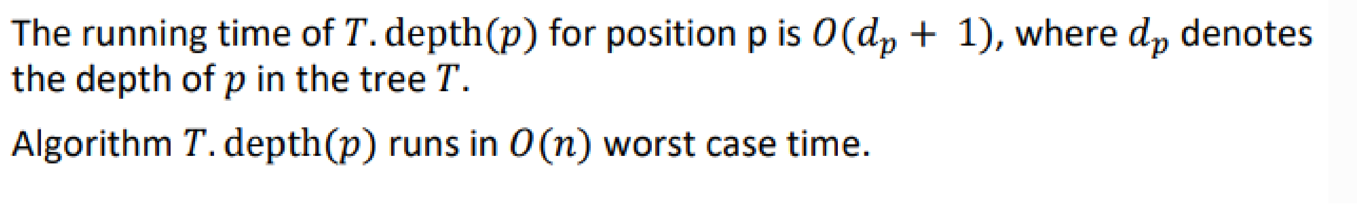
```
# ---------- concrete methods implemented in this class ----------
def is_root(self, p):
  """Return True if Position p represents the root of the tree."""
  return self.root( ) == p

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
```

深度计算：

Computing Depth计算深度深度： 设 p 为树 T 的节点位置。p 的深度是 p 的祖先数，不包括 p 本身。 请注意，此定义意味着 T 的根的深度为 0。 p 的深度也可以递归定义如下： • 如果 p 是根，则 p 的深度为 0。• 否则，p 的深度为 1 加上 p 的父项的深度。



```
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))
```

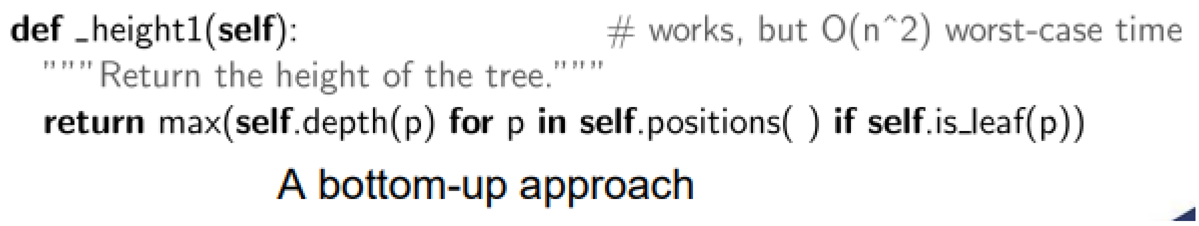


The running time of T.depth(p) for position p is $O(d_p + 1)$, where d_p denotes the depth of p in the tree T.

Algorithm T.depth(p) runs in $O(n)$ worst case time.

树当中的的高度p也是递归定义的 如果 p 是一片叶子，则 p 的高度为 0。• 否则，p 的身高比 p 的子项的最大值高大 1。 非空树的高度 T 是 T 的根的高度。

定理：非空树 T 的高度等于其叶子位置深度的最大值

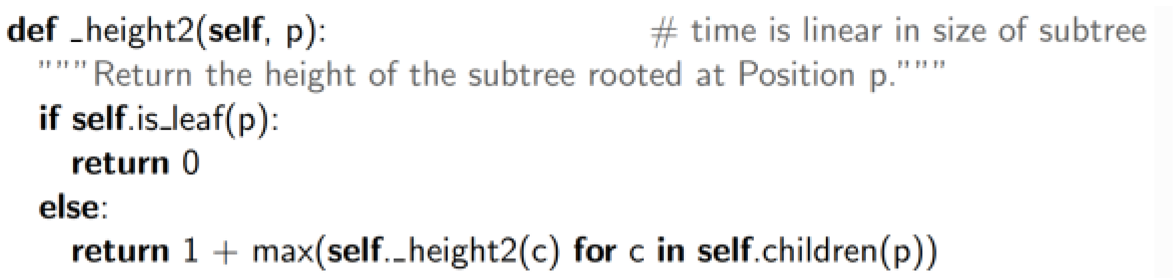


```
def _height1(self):                    # works, but O(n^2) worst-case time
  """Return the height of the tree."""
  return max(self.depth(p) for p in self.positions( ) if self.is_leaf(p))
```

A bottom-up approach

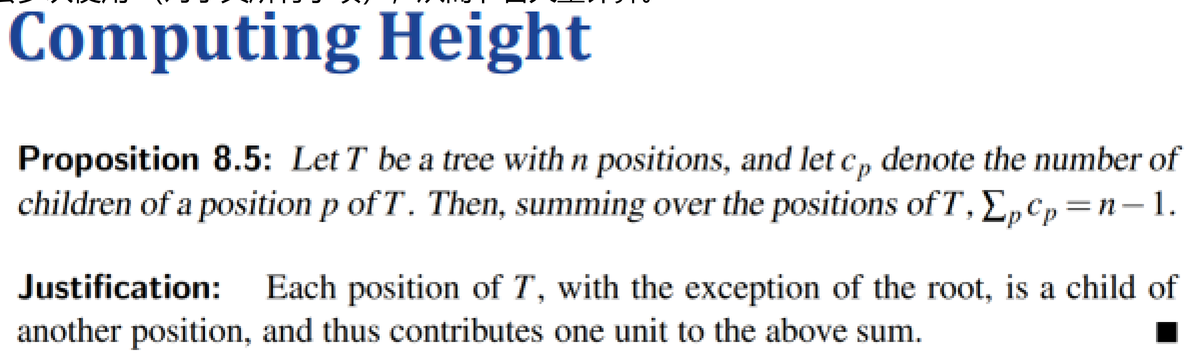
自底向上的方法

一种更高效的计算高度算法：一种自上而下的方法



```
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))
```

为什么自上而下的方法更有效？ • 请注意，父子关系是 1 对 N 的映射。 • 以自下而上 （子-父） 方式计算节点的深度不会重复使用上级的深度，因此，内部节点的深度可能会多次计算。 • 相比之下，使用自上而下的方式，节点的高度会多次使用 （对于其所有子项）， 从而节省大量计算。



Computing Height

Proposition 8.5: *Let T be a tree with n positions, and let c_p denote the number of children of a position p of T. Then, summing over the positions of T, $\sum_p c_p = n - 1$.*

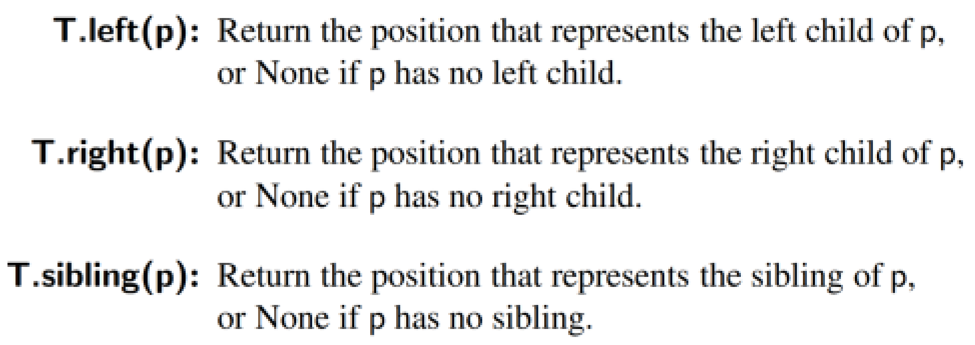
Justification: Each position of T, with the exception of the root, is a child of another position, and thus contributes one unit to the above sum. ■

Binary Trees二叉树

二叉树是具有以下属性的有序树： • 每个节点最多有两个子节点。 • 每个子节点都标记为左子节点或右子节点。 • 按节点的子项顺序，左子项位于右子项之前。左/右子树：以内部节点 v 的左子树或右子节点为根的子树分别称为 v 的左子树或右子树。 正确的二叉树：如果每个节点都有零个或两个子节点，则二叉树是正确的。 eg：决策树、算术表达式树

递归二叉树定义

我们还可以用递归方式定义二叉树，使二叉树要么是空的，要么由以下部分组成： • 一个节点 r，称为 T 的根，存储一个元素 • 一棵二叉树（可能为空）， 称为 T 的左子树 • 一棵二叉树（可能为空）， 称为 T 的右子树

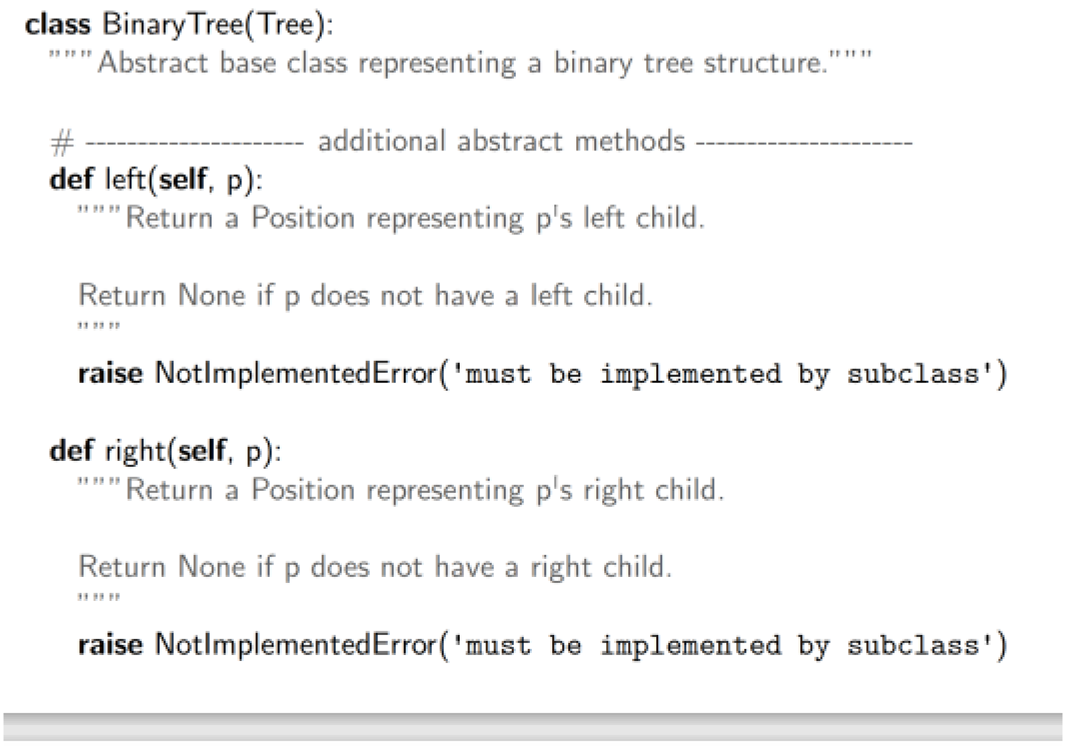


T.left(p): Return the position that represents the left child of p, or None if p has no left child.

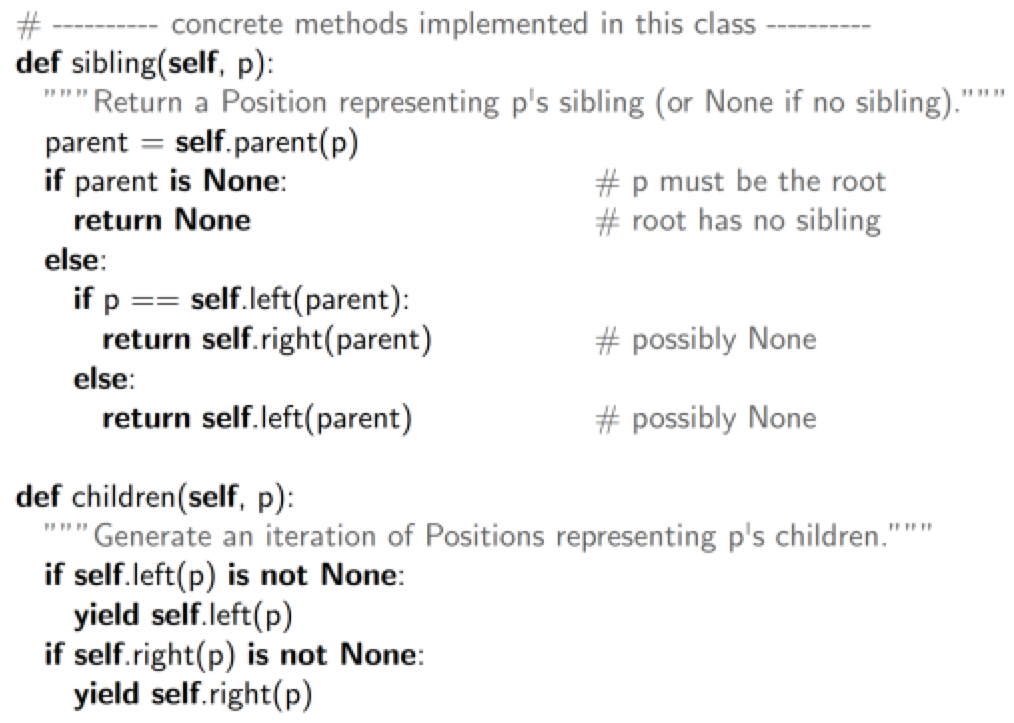
T.right(p): Return the position that represents the right child of p, or None if p has no right child.

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

三种额外的方法:

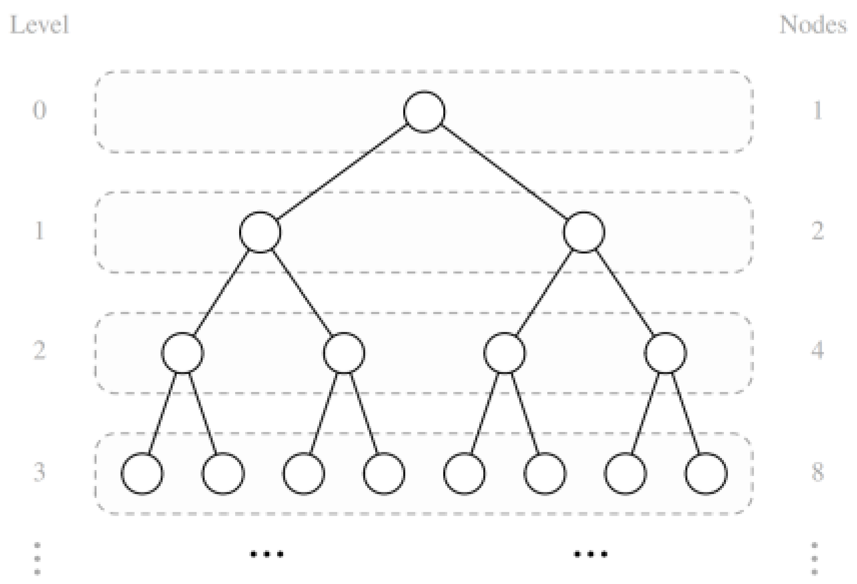


```python
class BinaryTree(Tree):
  """Abstract base class representing a binary tree structure."""

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



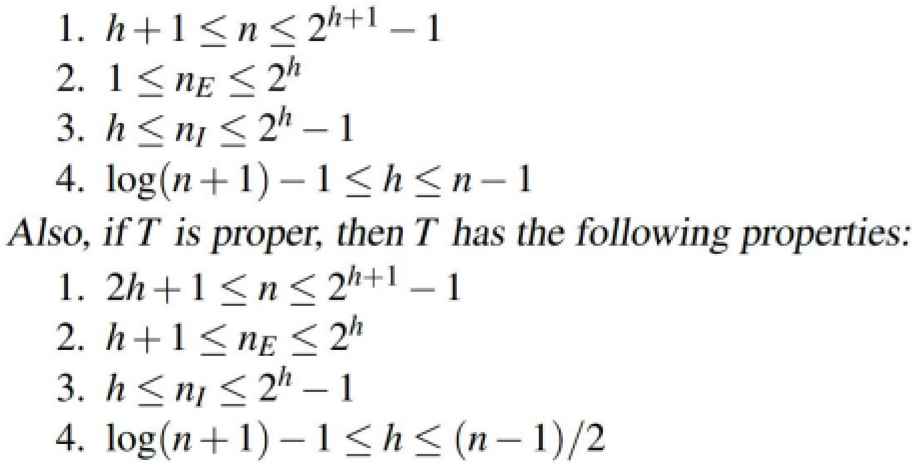
```python
  # ---------- concrete methods implemented in this class ----------
  def sibling(self, p):
    """Return a Position representing p's sibling (or None if no sibling)."""
    parent = self.parent(p)
    if parent is None:                    # p must be the root
      return None           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

二叉树的性质:



1，级别 d 最多有 2d 节点。

2，设 T 为非空二叉树，设 n、nE、nI 和 h 分别表示节点数、外部节点数、内部节点数和 T 的高度。则 T 具有



1. $h+1 \leq n \leq 2^{h+1}-1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq n-1$

Also, if T is proper, then T has the following properties:

1. $2h+1 \leq n \leq 2^{h+1}-1$
2. $h+1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq (n-1)/2$

以下属性 3，有a和外部节点，n个内部节点，则a=n+1(外部节点=内部节点+1)

Implementing Binary Trees

链表结构：

每个节点都保留对存储在位置 p 的元素及其父位置和子位置的引用。树本身维护对根节点（如果有）和树的大小（树中的节点总数）的引用

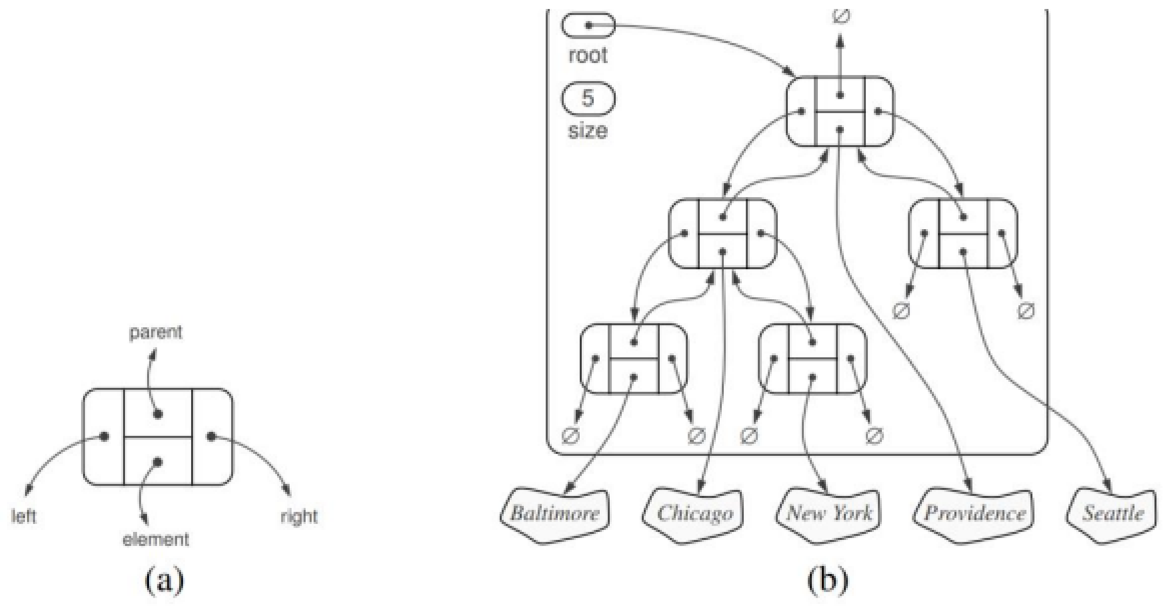
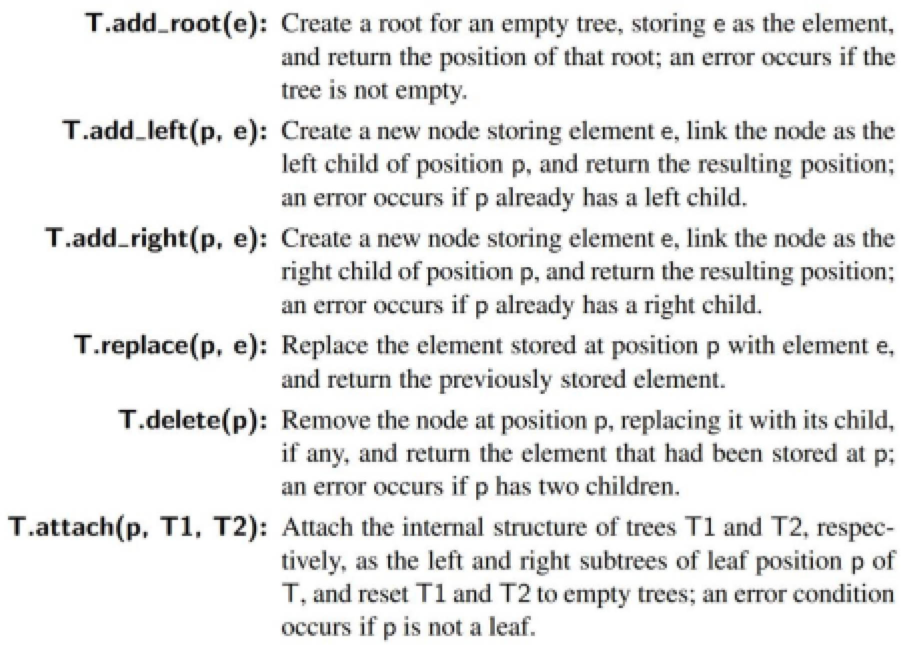
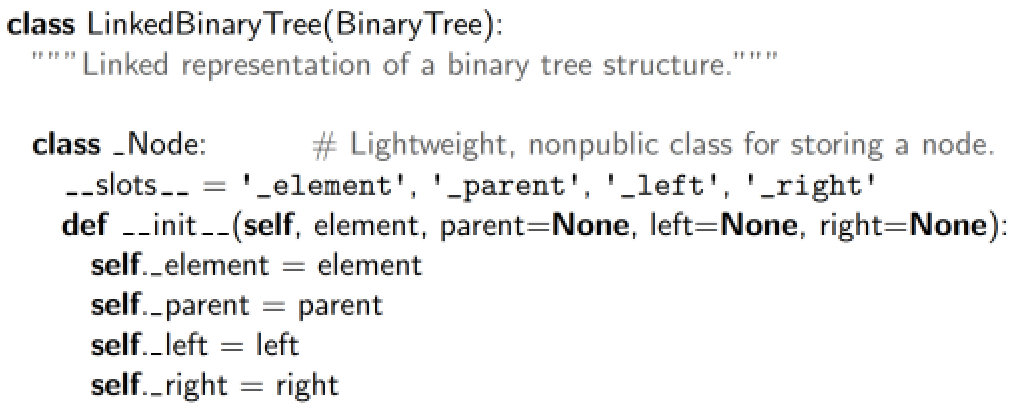


Figure 8.11: A linked structure for representing: (a) a single node; (b) a binary tree.



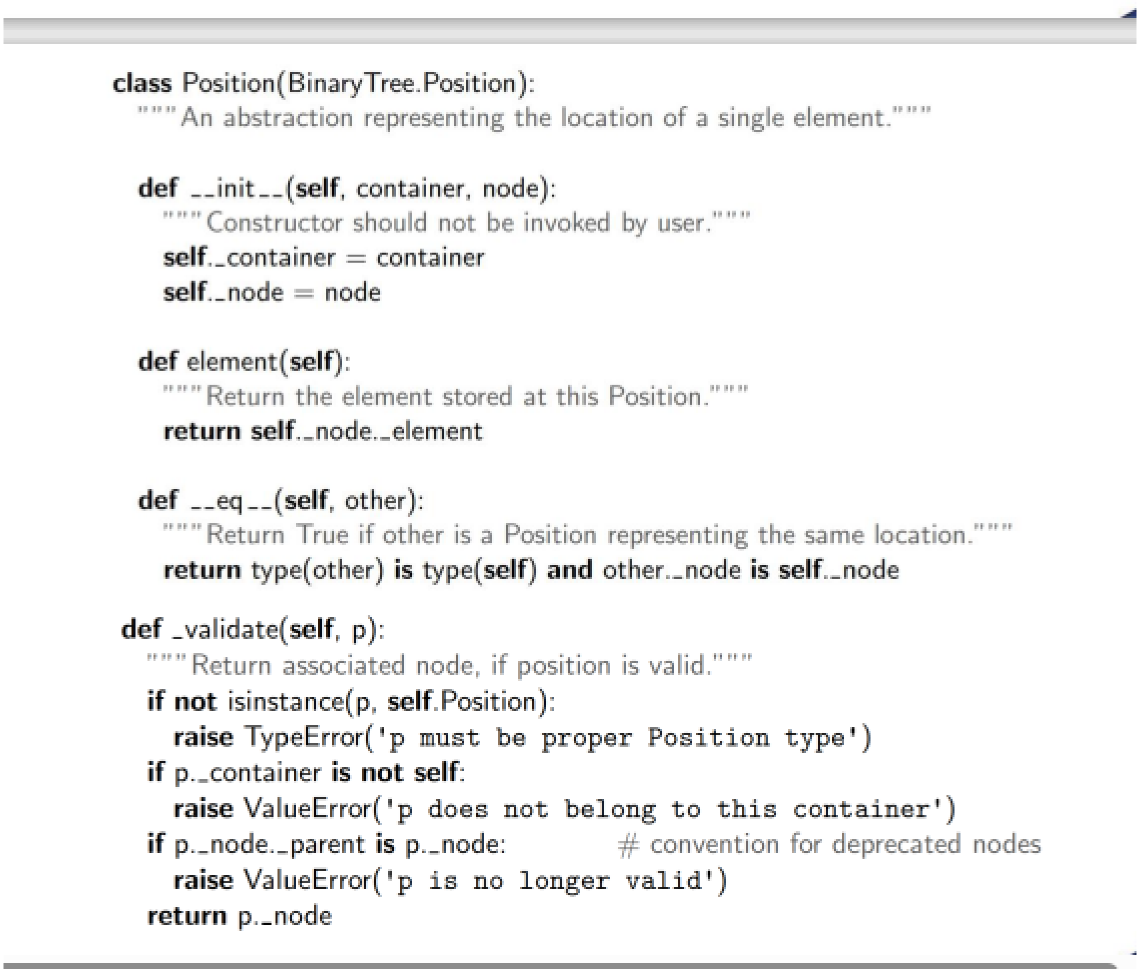
T.add_root(e):	Create a root for an empty tree, storing e as the element, and return the position of that root; an error occurs if the tree is not empty.
T.add_left(p, e):	Create a new node storing element e, link the node as the left child of position p, and return the resulting position; an error occurs 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, 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.

ach of them can be implemented in O(1) worse-case time.

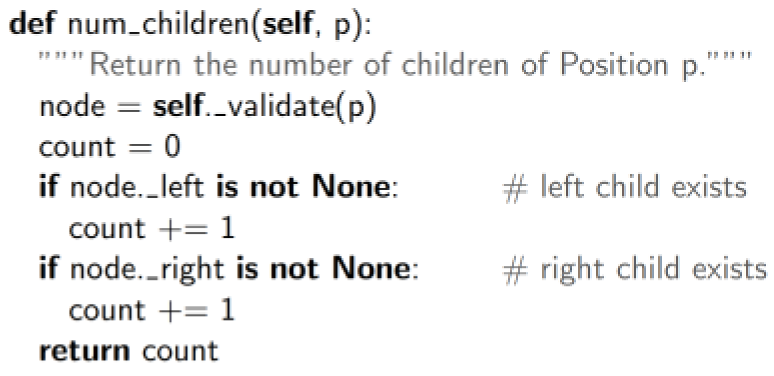


```
class LinkedBinaryTree(BinaryTree):
  """Linked representation of a binary tree structure."""

  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
```

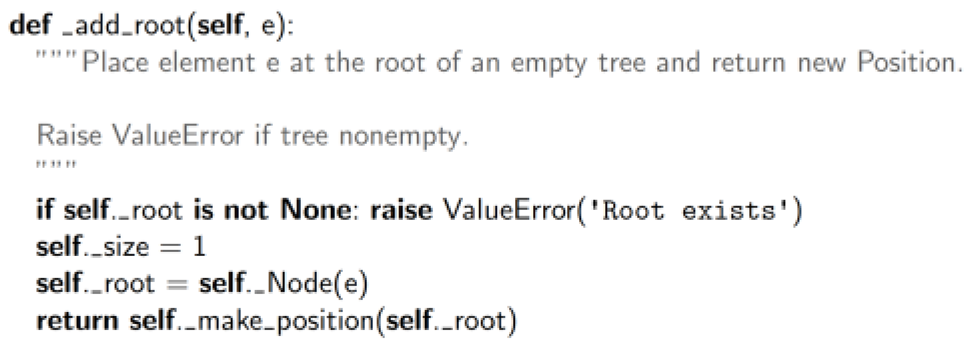
he class has four variables: _element, _parent, _left, _right



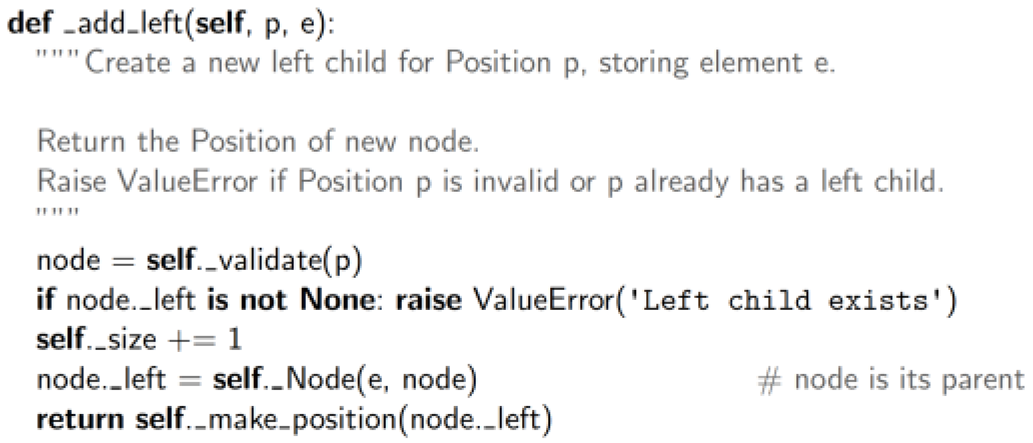
```
  class Position(BinaryTree.Position):
    """An abstraction representing the location of a single element."""

    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

    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
```



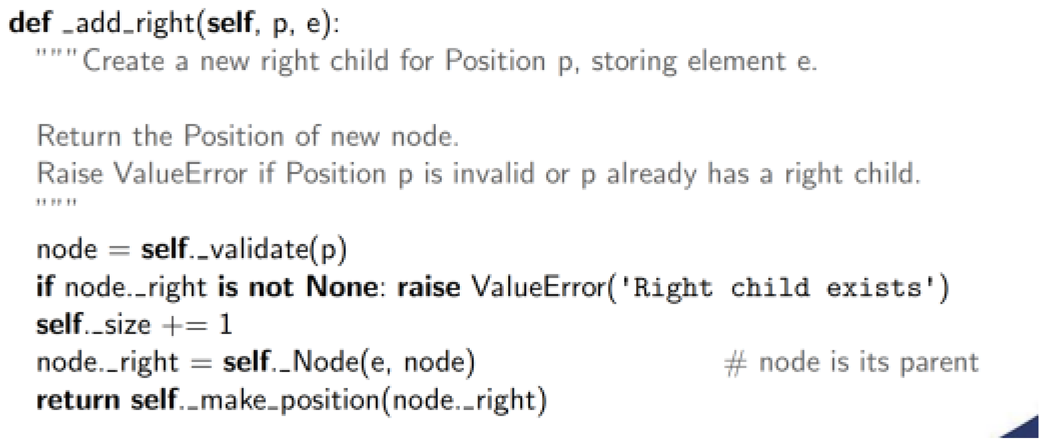
```
def num_children(self, p):
  """Return the number of children of Position p."""
  node = self._validate(p)
  count = 0
  if node._left is not None:        # left child exists
    count += 1
  if node._right is not None:       # right child exists
    count += 1
  return count
```



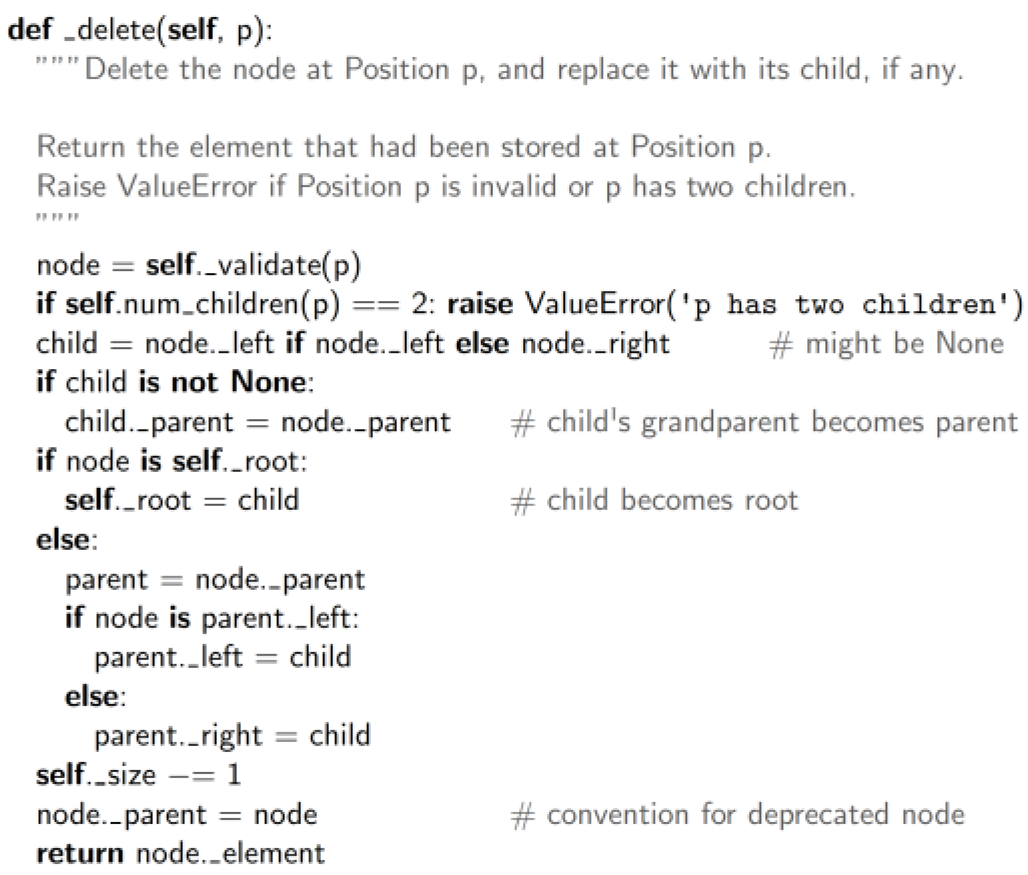
```
def _add_root(self, e):
  """Place element e at the root of an empty tree and return new Position.

  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)
```



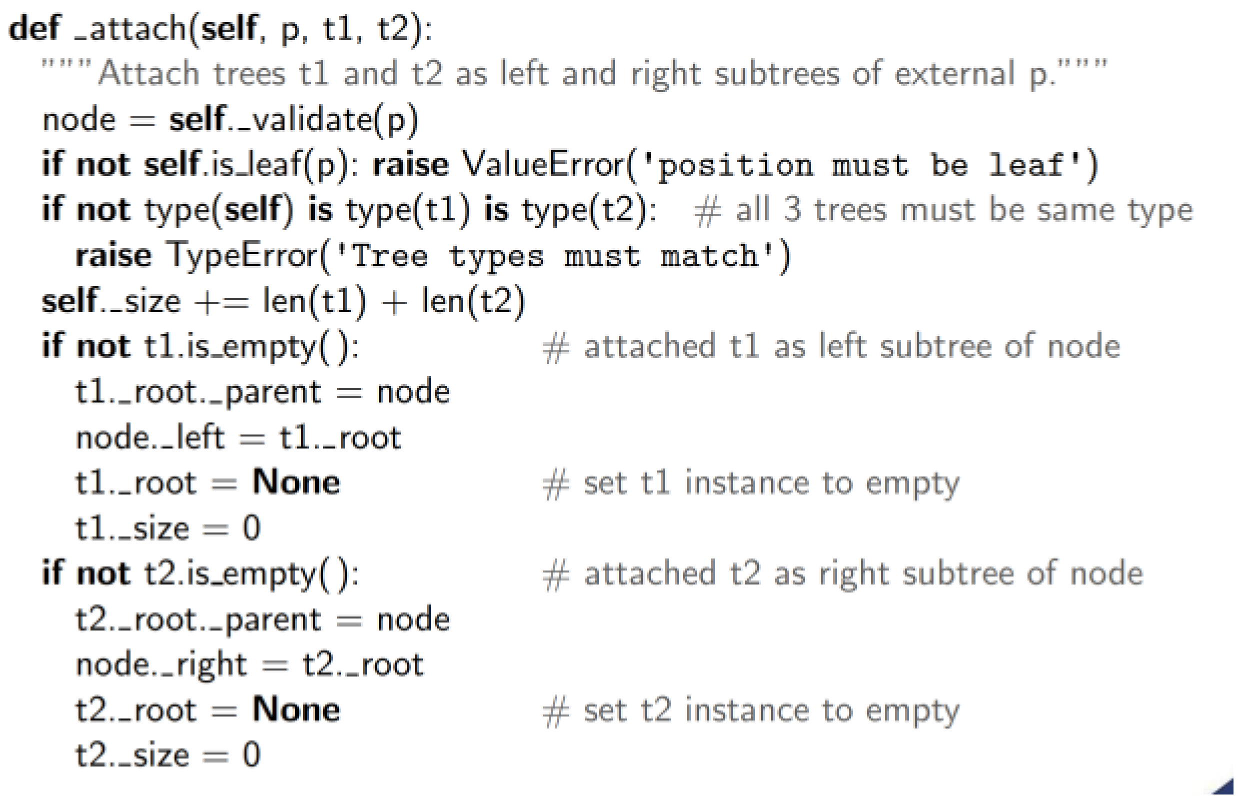
```
def _add_left(self, p, e):
  """Create a new left child for Position p, storing element e.

  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
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
    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
```



```python
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 += 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
```

基于数组的方法：

- If p is the root of T , then $f\ (p)\ =\ 0$
- If p is the left child of position q, then $f(p)\ =\ 2f(q)\ +\ 1.$
- If p is the right child of position q, then $f(p)\ =\ 2f(q)\ +\ 2.$

Numbers the positions on each level of T in **increasing order** from left to right.

Level numbering is based on **potential positions** within the tree, not actual positions of a given tree, so they are **not necessarily consecutive**.

按从左到右的升序对 T 的每个级别上的位置进行编号。级别编号基于树中的潜在位置，而不是给定树的实际位置，因此它们不一定是连续的。

Level numbering suggests a way to represent a binary tree with an array.

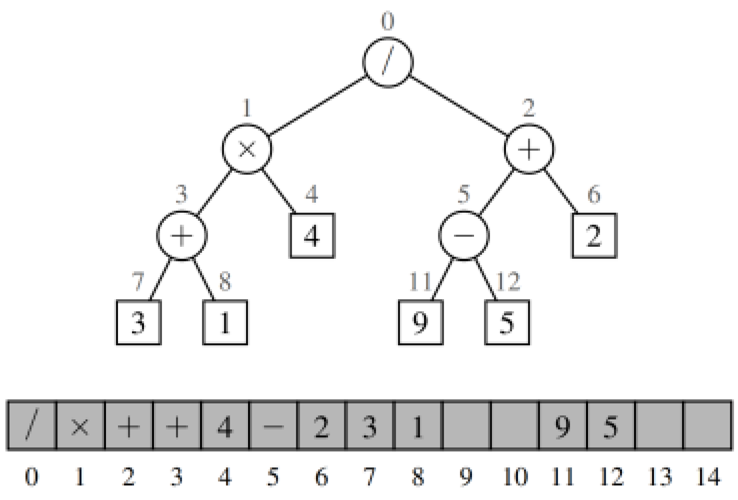


Figure 8.13: Representation of a binary tree by means of an array.

基于数组的表示的优缺点： 优点： 位置 p 可以用单个整数 f（p） 表示。根据我们的级别编号公式，p 的左子级的索引为 2f（p） + 1，p 的右子级的索引为 2f（p） + 2，p 的父级的索引为 L（f（p） − 1）/2」。 基于数组的表示的空间使用在很大程度上取决于树的形状。设 n 为 T 的节点数，设 fM 为 T 的所有节点的 f（p） 的最大值。数组 A 需要长度 N = 1 + fM。请注意，A 可能有许多空单元格，这些单元格不引用 T 的现有节点

```
print("Hello:")
```