数据结构与算法

树

张晓平

武汉大学数学与统计学院

Table of contents

1. 树
2. 二叉树 (Binary Tree)
3. 树的实现
4. 树的遍历算法
5. Expression Tree

树

树

树是最重要的一种非线性数据结构。

- 树结构是数据组织的一个突破，使用它能构造出一系列比使用线性数据结构更快的算法。
- 树为数据提供了一种自然的组织方式，它已经成为文件系统、图形用户界面、数据库、网站和其他计算机系统中普遍存在的结构。

树

树是最重要的一种非线性数据结构。

- 树结构是数据组织的一个突破，使用它能构造出一系列比使用线性数据结构更快的算法。
- 树为数据提供了一种自然的组织方式，它已经成为文件系统、图形用户界面、数据库、网站和其他计算机系统中普遍存在的结构。

“非线性”指的是一种组织关系，它比序列中对象之间简单的“前”和“后”关系更丰富。树中的关系是分层的，有些对象“高”，有些对象“低”.

树

树数据结构的主要术语来自于家谱，其中“父”、“子”、“祖先”和“后代”是描述关系最常用的词。

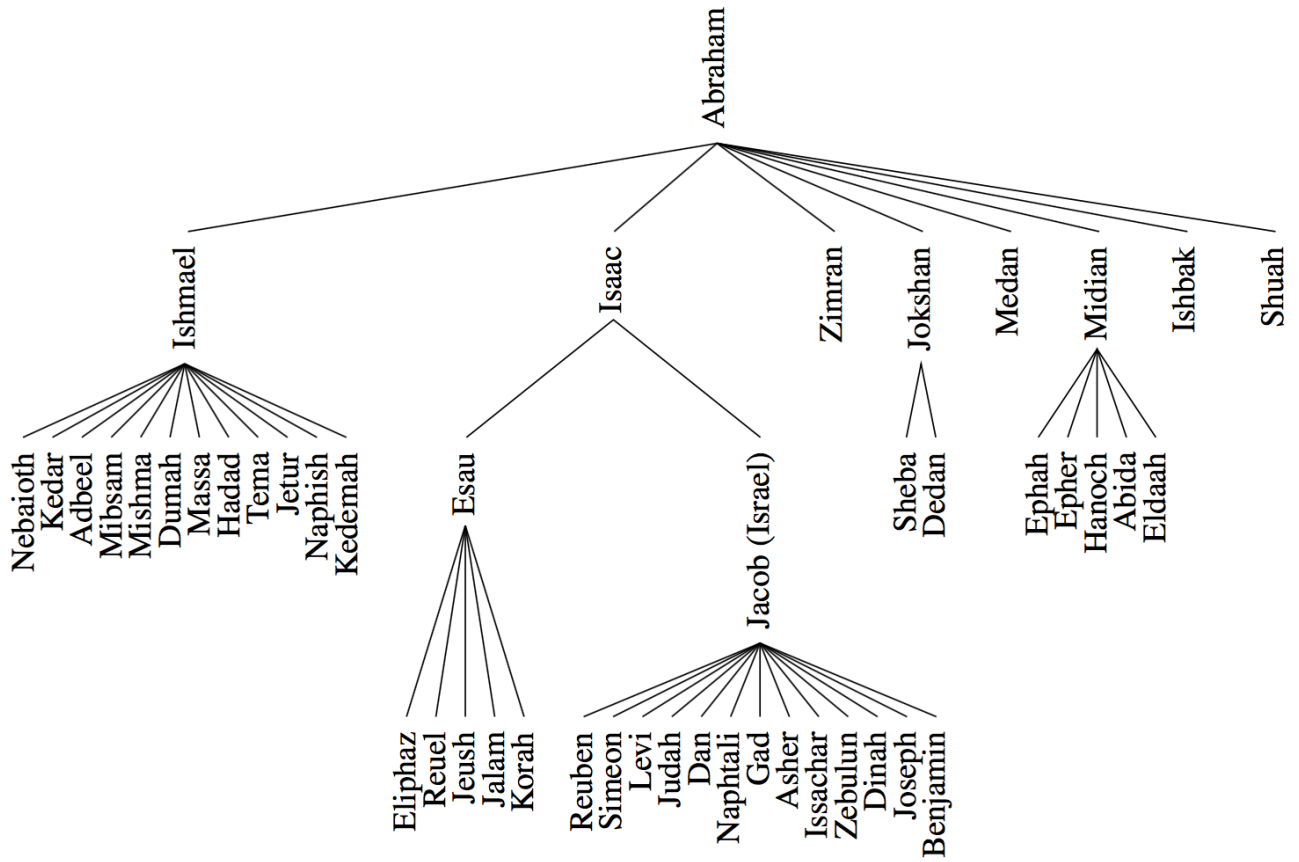

Figure 8.1: A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

树

树的定义及其性质

树的定义及其性质

树是一种抽象的数据类型，它按层次结构存储元素。

除了顶部元素之外，树中的每个元素都有一个父元素和零个或多个子元素。

通常将顶部元素称为树的根。

树的定义及其性质

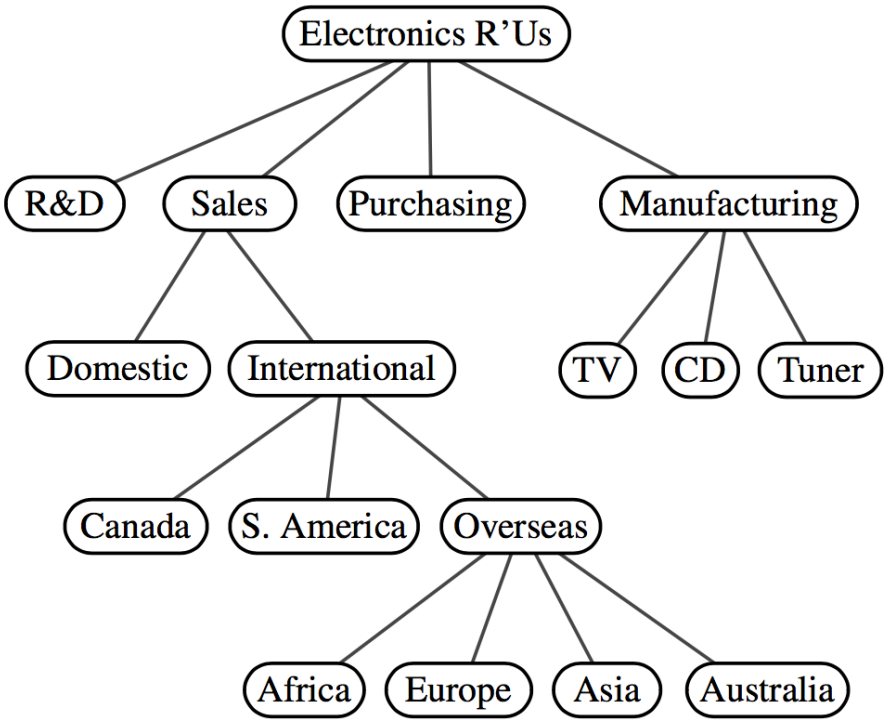

Figure 8.2: A tree with 17 nodes representing the organization of a fictitious corporation. The root stores *Electronics R'Us*. The children of the root store *R&D*, *Sales*, *Purchasing*, and *Manufacturing*. The internal nodes store *Sales*, *International*, *Overseas*, *Electronics R'Us*, and *Manufacturing*.

树的定义及其性质

定义：树的定义

树 T 是一组存储元素的节点，这些节点满足以下属性：

- 若 T 非空，则它有一个特殊节点，即 T 的根，它没有父节点。
- 非根节点 v 都有一个唯一的父节点 w；以 w 为父节点的每个节点都是 w 的子节点。

树的定义及其性质

定义：树的定义

树 T 是一组存储元素的节点，这些节点满足以下属性：

- 若 T 非空，则它有一个特殊节点，即 T 的根，它没有父节点。
- 非根节点 v 都有一个唯一的父节点 w；以 w 为父节点的每个节点都是 w 的子节点。

由以上定义可知，树可以为空，即它没有任何节点。此约定还允许以递归的方式来定义树，即：树 T 要么为空，要么由根节点 r 和一组以 r 为根节点的子树（可能为空）组成。

树的定义及其性质

- 具有相同父节点的两节点称为兄弟节点。
- 若 v 没有子节点，则称它为外部节点，也称叶子节点。

  若 v 有一个或多个子节点，则称它为内部结点.
- 如果 $u = v$ 或 u 是 v 的父节点的祖先，则 u 是 v 的祖先。

  反之，若 u 是 v 的祖先，则 v 是 u 的后代。
- 以 v 为根节点的 T 子树是由 v 在 T 中的所有后代（包括 v 本身）组成的树。
- 称 (u, v) 为树 T 的一条边，若 u 是 v 的父节点，或者 v 是 u 的父节点。
- 路径 T 是一个节点序列，且序列中的任何两个连续节点形成一条边。

树的定义及其性质

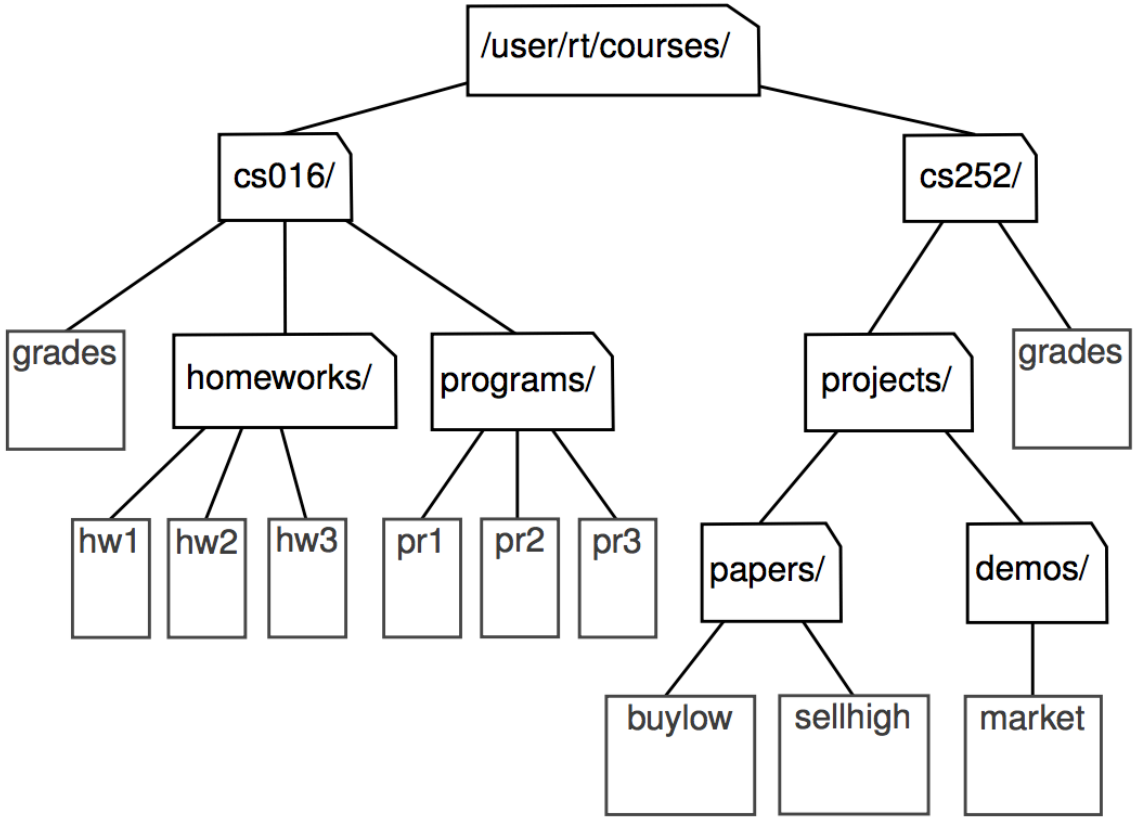

Figure 8.3: Tree representing a portion of a file system.

树的定义及其性质

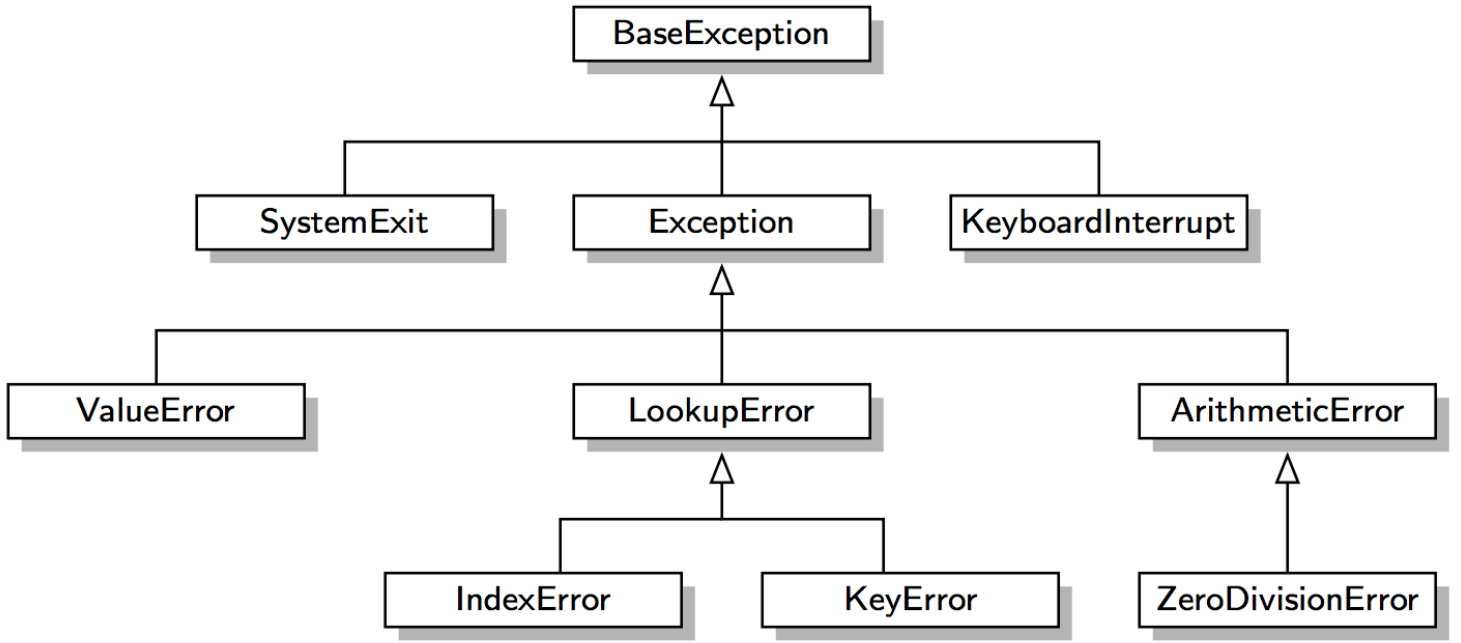

Figure 8.4: A portion of Python's hierarchy of exception types.

树的定义及其性质

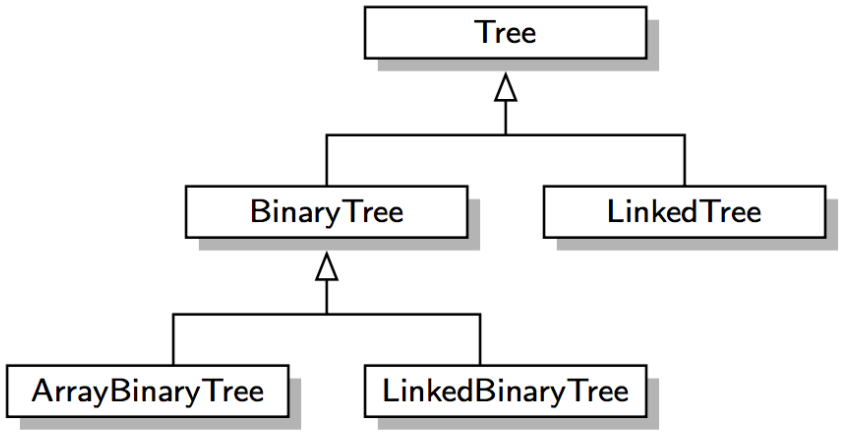

Figure 8.5: Our own inheritance hierarchy for modeling various abstractions and implementations of tree data structures. In the remainder of this chapter, we provide implementations of Tree, BinaryTree, and LinkedBinaryTree classes, and high-level sketches for how LinkedTree and ArrayBinaryTree might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

树的定义及其性质

定义：有序树 (ordered tree)

若每个节点的子节点之间存在一个有意义的线性顺序，则称该树是有序的。此时，可将节点的子节点标识为第一、第二、第三等等，通常按长幼次序从左到右进行排列。

树的定义及其性质

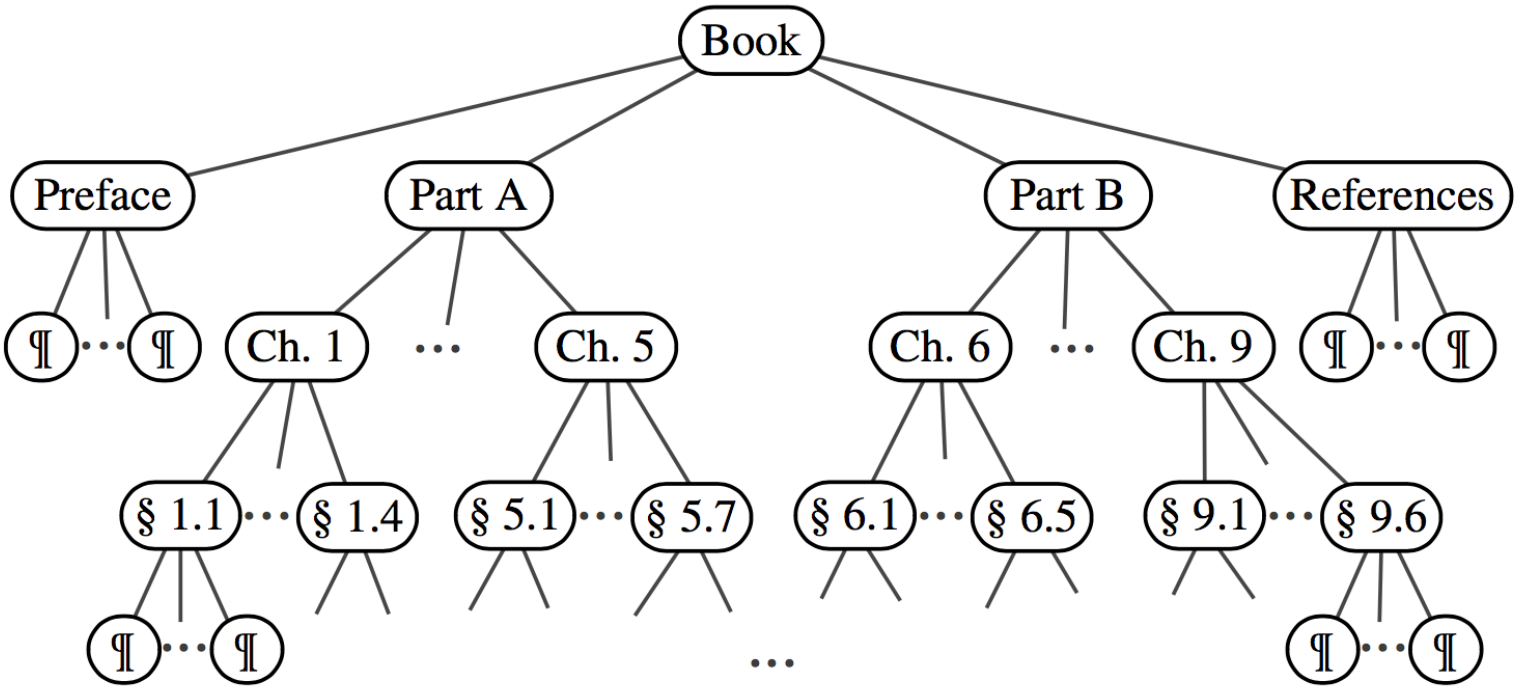

Figure 8.6: An ordered tree associated with a book.

树

树抽象数据类型

树抽象数据类型

定义树的抽象数据类型，可使用位置的概念来作为节点的抽象。元素存储在每个位置，位置满足定义树结构的父子关系。

树的位置对象支持以下方法：

- `p.element()`：返回存储在 p 位置的元素。

树抽象数据类型

树的抽象数据类型支持以下访问器方法，允许用户确定树的各个位置：

- `T.root()`：返回 T 的根节点的位置，若 T 为空，则返回 *None*。
- `T.is_root(p)`：若位置 p 为 T 的根，返回 *True*。
- `T.parent(p)`：返回位置 p 的父节点的位置，若 p 为 T 的根节点，则返回 *None*。
- `T.num_chidren(p)`：返回位置 p 的子节点的个数。
- `T.children(p)`：生成位置 p 的子节点的迭代。
- `T.is_leaf()`：若位置 p 没有任何子节点，则返回 *True*。
- `len(T)`：返回 T 中包含的位置数（以及相应的元素）。
- `T.is_empty()`：若 T 不包含任何位置，则返回 *True*。
- `T.positions()`：生成 T 的所有位置的迭代。
- `iter(T)`：生成存储在 T 中的所有元素的迭代。

树抽象数据类型

树的抽象数据类型支持以下访问器方法，允许用户确定树的各个位置：

- `T.root()`：返回 T 的根节点的位置，若 T 为空，则返回 *None*。
- `T.is_root(p)`：若位置 p 为 T 的根，返回 *True*。
- `T.parent(p)`：返回位置 p 的父节点的位置，若 p 为 T 的根节点，则返回 *None*。
- `T.num_chidren(p)`：返回位置 p 的子节点的个数。
- `T.children(p)`：生成位置 p 的子节点的迭代。
- `T.is_leaf()`：若位置 p 没有任何子节点，则返回 *True*。
- `len(T)`：返回 T 中包含的位置数（以及相应的元素）。
- `T.is_empty()`：若 T 不包含任何位置，则返回 *True*。
- `T.positions()`：生成 T 的所有位置的迭代。
- `iter(T)`：生成存储在 T 中的所有元素的迭代。

若位置 p 对 T 无效，则接受 p 作为参数的上述任何方法都应抛出一个 **ValueError**。

树抽象数据类型

- 若 T 有序，则`T.children(p)`会按自然顺序生成 p 的子节点的一个迭代。
- 若 p 为叶子，则`T.children(p)`将生成一个空迭代。
- 若 T 为空，则`T.positions()`和`iter(T)`均生成空迭代。

树抽象数据类型

定义一个类作为一个抽象基类 (abstract base class)，通过继承 (inheritance)来定义一个或多个具体类 (concrete classes)。

树抽象数据类型

定义一个类作为一个抽象基类 (abstract base class)，通过继承 (inheritance)来定义一个或多个具体类 (concrete classes)。

这里我们定义一个*Tree*类，作为对应于树 ADT 的抽象基类。*Tree*类并不定义内部数据的存储方式，其中的方法 (如*root*, *parent*, *num_children*, *children*, *__len__*) 是抽象的，不给出具体定义，均抛出一个 **NotImplementedError** 异常.

树抽象数据类型

定义一个类作为一个抽象基类 (abstract base class)，通过继承 (inheritance)来定义一个或多个具体类 (concrete classes)。

这里我们定义一个*Tree*类，作为对应于树 ADT 的抽象基类。*Tree*类并不定义内部数据的存储方式，其中的方法 (如*root*, *parent*, *num_children*, *children*, *__len__*) 是抽象的，不给出具体定义，均抛出一个 **NotImplementedError** 异常.

子类负责重写抽象方法，以便根据所选的数据存储方式为每个行为提供具体实现。

树抽象数据类型

定义一个类作为一个抽象基类 (abstract base class)，通过继承 (inheritance)来定义一个或多个具体类 (concrete classes)。

这里我们定义一个*Tree*类，作为对应于树 ADT 的抽象基类。*Tree*类并不定义内部数据的存储方式，其中的方法 (如*root*, *parent*, *num_children*, *children*, *__len__*) 是抽象的，不给出具体定义，均抛出一个 **NotImplementedError** 异常.

子类负责重写抽象方法，以便根据所选的数据存储方式为每个行为提供具体实现。

由于 *Tree* 类是抽象的，不能直接创建它的实例。它通常作为继承的基础，用户将创建具体子类的实例。

树抽象数据类型

```
class Tree(object):
  class Position(object):
    def elem(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)
```

树抽象数据类型

```
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')
```

树抽象数据类型

```
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
```

树

树的深度 (depth) 与高度 (height)

树的深度 (depth) 与高度 (height)

定义：深度

设 p 为 T 中某节点的位置，称 p 的祖先 (不包含其自身) 的个数为 p 的深度。

注意：T 根节点的深度为 0.

树的深度 (depth) 与高度 (height)

定义：深度

设 p 为 T 中某节点的位置，称 p 的祖先 (不包含其自身) 的个数为 p 的深度。

注意：T 根节点的深度为 0.

定义：深度的递归定义

- 若 p 为根，则其深度为 0；
- 否则，p 的深度等于 p 父节点的深度加 1.

树的深度 (depth) 与高度 (height)

定义：深度

设 p 为 T 中某节点的位置，称 p 的祖先 (不包含其自身) 的个数为 p 的深度。

注意：T 根节点的深度为 0.

定义：深度的递归定义

- 若 p 为根，则其深度为 0;
- 否则，p 的深度等于 p 父节点的深度加 1.

```
def depth(self, p):
    if self.is_root(p):
        return 0
    else:
        return 1 + self.depth(self.parent(p))
```

树的深度 (depth) 与高度 (height)

时间复杂度分析

- T.depth(p)的时间复杂度为 $O(d_p+1)$，其中 d_p 表示 p 在 T 中的深度。
- 若 T 中的所有节点构成一个线性表，则T.depth(p)的最坏时间复杂度为 $O(n)$，其中 n 表示 T 中的位置总数。

树的深度 (depth) 与高度 (height)

定义：位置 p 高度

T 中某位置 p 的高度可递归定义为:

- 若 p 为叶子，其高度为 0;
- 否则，p 的高度等于所有 p 的子节点的高度的最大值加 1。

树的深度 (depth) 与高度 (height)

定义：位置 p 高度

T 中某位置 p 的高度可递归定义为:

- 若 p 为叶子，其高度为 0;
- 否则，p 的高度等于所有 p 的子节点的高度的最大值加 1。

定义：树 T 的高度

非空树 T 的高度为 T 根节点的高度。

树的深度 (depth) 与高度 (height)

```
def _height2(self, p):  # time is linear in
size of subtree
  if self.is_leaf(p):
    return 0
  else:
    return 1 + max(self._height2(c)
                   for c in self.children(p))
```

树的深度 (depth) 与高度 (height)

```
def _height2(self, p):  # time is linear in
size of subtree
  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):
  if p is None:
    p = self.root()
  return self._height2(p)
```

最坏时间复杂度为 $O(n)$。

树的深度 (depth) 与高度 (height)

性质

非空树 T 的高度等于所有叶子位置的深度的最大值。

树的深度 (depth) 与高度 (height)

性质

非空树 T 的高度等于所有叶子位置的深度的最大值。

```
def _height1(self): # O(n^2) worst-case time
  return max(self.depth(p)
             for p in self.positions()
             if self.is_leaf(p))
```

二叉树 (Binary Tree)

二叉树 (Binary Tree)

定义：Binary Tree

二叉树是具有以下属性的顺序树：

1. 每个节点最多有两个孩子；
2. 每个孩子都标记为左孩子或右孩子。
3. 左孩子先于右孩子。

二叉树 (Binary Tree)

定义：Binary Tree

二叉树是具有以下属性的顺序树：

1. 每个节点最多有两个孩子；
2. 每个孩子都标记为左孩子或右孩子。
3. 左孩子先于右孩子。

- 以 v 的左孩子为根节点的子树称为 v 的左子树；以 v 的右孩子为根节点的子树称为 v 的右子树；

二叉树 (Binary Tree)

定义：Binary Tree

二叉树是具有以下属性的顺序树：

1. 每个节点最多有两个孩子；
2. 每个孩子都标记为左孩子或右孩子。
3. 左孩子先于右孩子。

- 以 v 的左孩子为根节点的子树称为 v 的左子树；以 v 的右孩子为根节点的子树称为 v 的右子树；
- 若每个节点有零个或两个孩子，则称该二叉树为真二叉树，也称满二叉树。在真二叉树中，每个内部结点刚好有两个孩子。

二叉树 (Binary Tree)

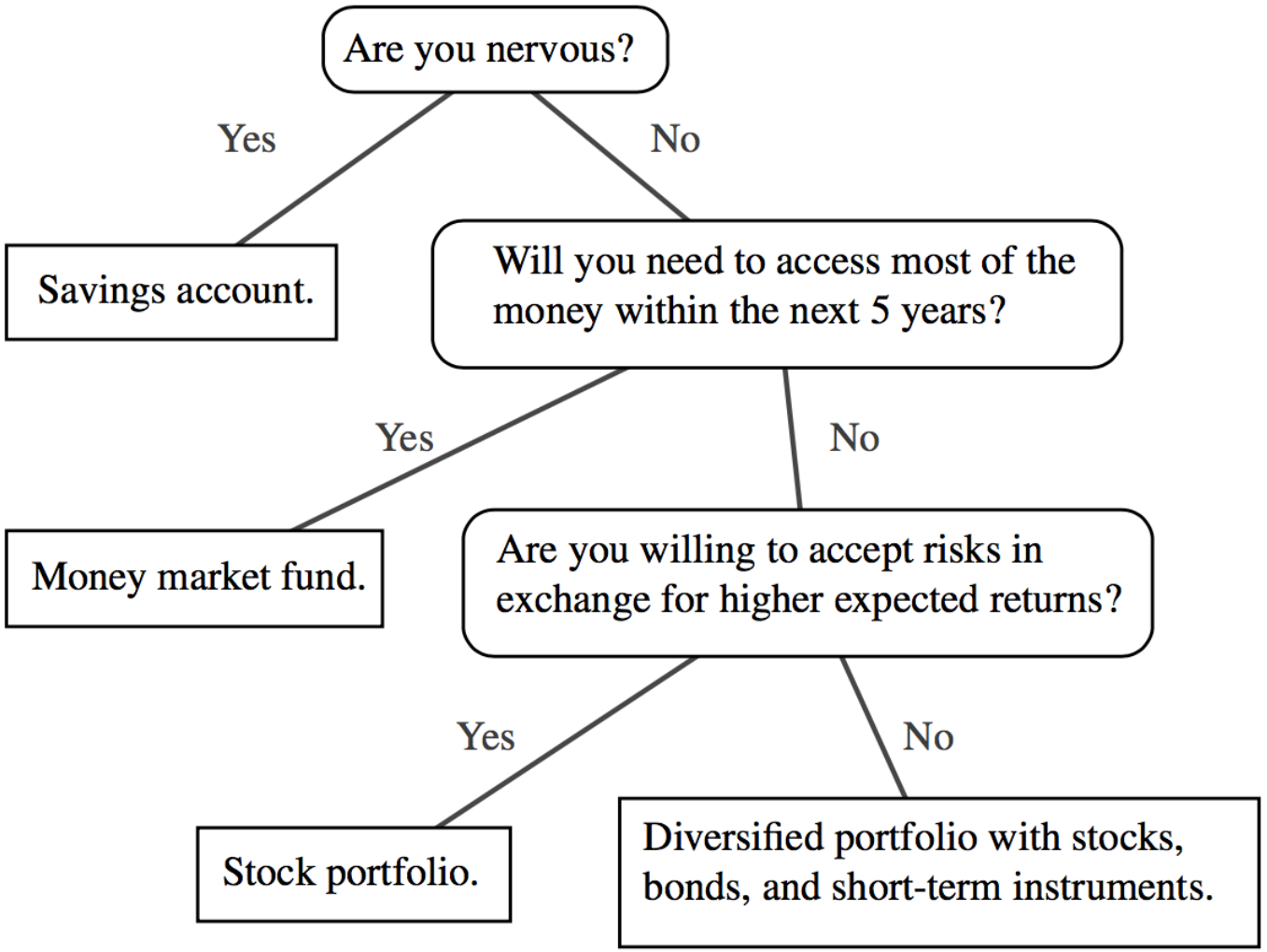

Figure 8.7: A decision tree providing investment advice.

二叉树 (Binary Tree)

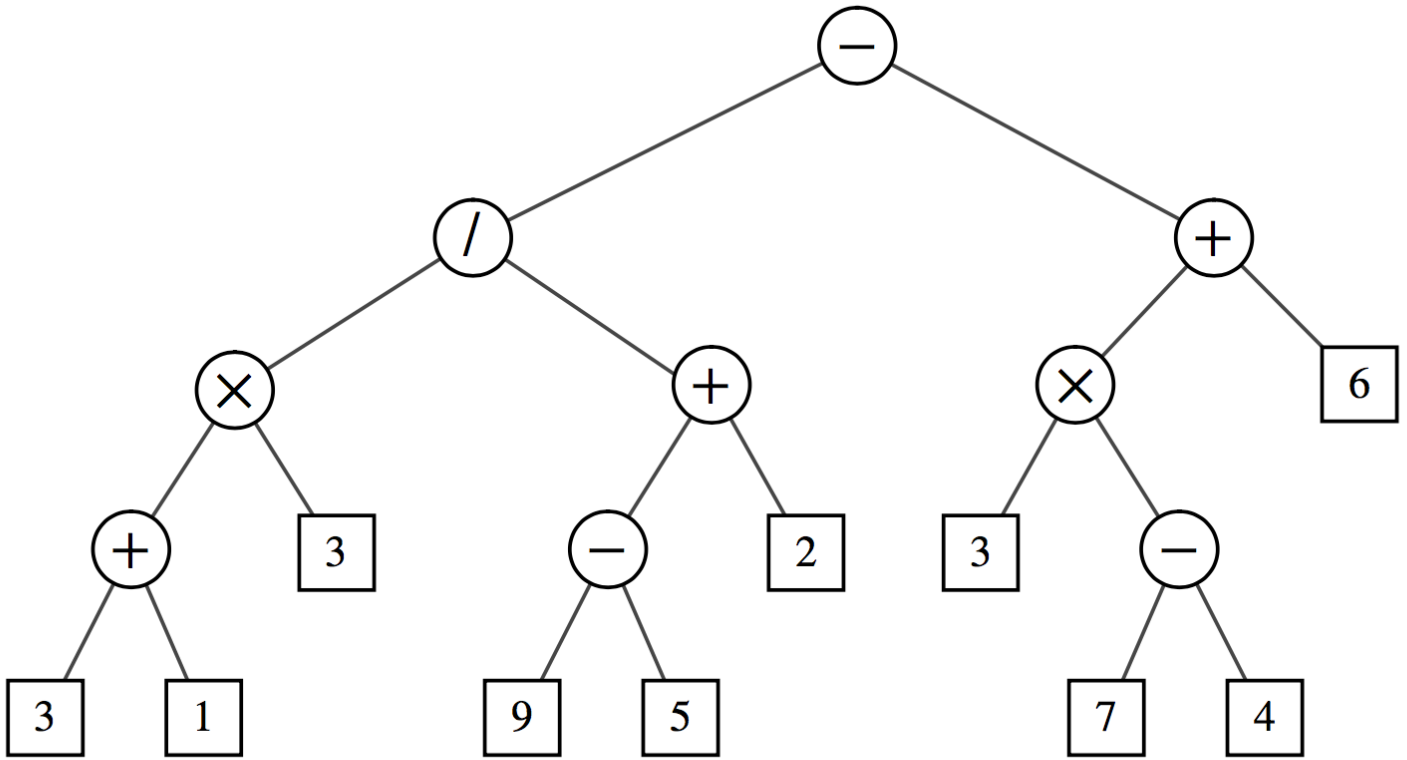

Figure 8.8: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

二叉树 (Binary Tree)

定义：二叉树的递归定义

二叉树 T 要么为空，要么由

1. 一个称为 T 的根的节点 r，它存储一个元素；
2. 一个称为 T 的左子树的二叉树 (可能为空)；
3. 一个称为 T 的右子树的二叉树 (可能为空)

组成。

二叉树 (Binary Tree)

二叉树数据类型

二叉树数据类型

作为抽象数据类型，二叉树是树的特例，它支持三个额外的访问器方法：

- `T.left(p)`：返回位置 p 的左孩子的位置。若 p 没有左孩子，则返回 **None**。
- `T.right(p)`：返回位置 p 的右孩子的位置。若 p 没有右孩子，则返回 **None**。
- `T.sibling(p)`：返回位置 p 的兄弟的位置。如果 p 没有兄弟，则返回 **None**。

二叉树数据类型

```
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')
```

二叉树数据类型

```
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)
```

二叉树数据类型

```
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)
```

二叉树 (Binary Tree)

二叉树的性质

二叉树的性质

记树 T 的第 d 层的节点的深度为 d。在二叉树中，

- 第 0 层最多有一个节点（根节点），
- 第 1 层最多有两个节点（根节点的两个孩子），
- 第 2 层最多有四个节点，
- ……

一般地，第 d 层最多有个 2^d 个节点。

二叉树的性质

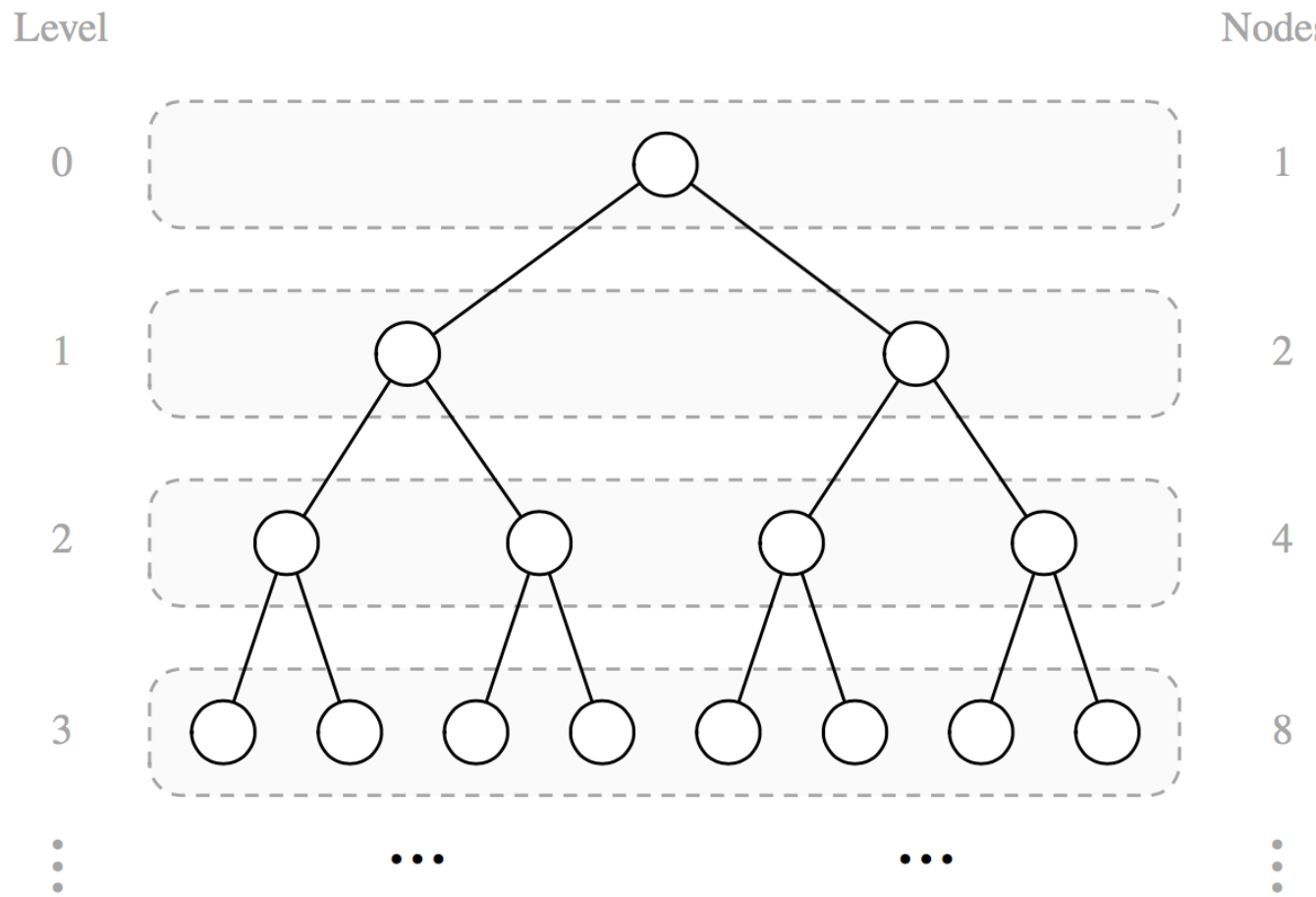

Figure 8.9: Maximum number of nodes in the levels of a binary tree.

二叉树的性质

性质

设 T 为非空二叉树，令 n, n_E, n_I 和 h 分别为 T 的节点数，外部节点数，内部节点数和高度，则

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

若 T 为真二叉树，则

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

二叉树的性质

性质

在非空真二叉树 T 中，$n_E = n_I + 1$。

二叉树的性质

性质

在非空真二叉树 T 中，$n_E = n_I + 1$。

证明

树的实现

树的实现

前面定义的 Tree 类和 BinaryTree 类都是抽象基类 (abstract base class)。

- 它们都不能直接实例化；
- 尚未定义如何在内部表示树以及如何在父树和子树之间有效定位的关键信息。
- 树的具体实现必须提供方法
  - `root`
  - `parent`
  - `num_children`
  - `children`
  - `__len__`

  而对于二叉树，还需实现
  - `left`
  - `right`

树的实现

二叉树的链式结构

二叉树的链式结构

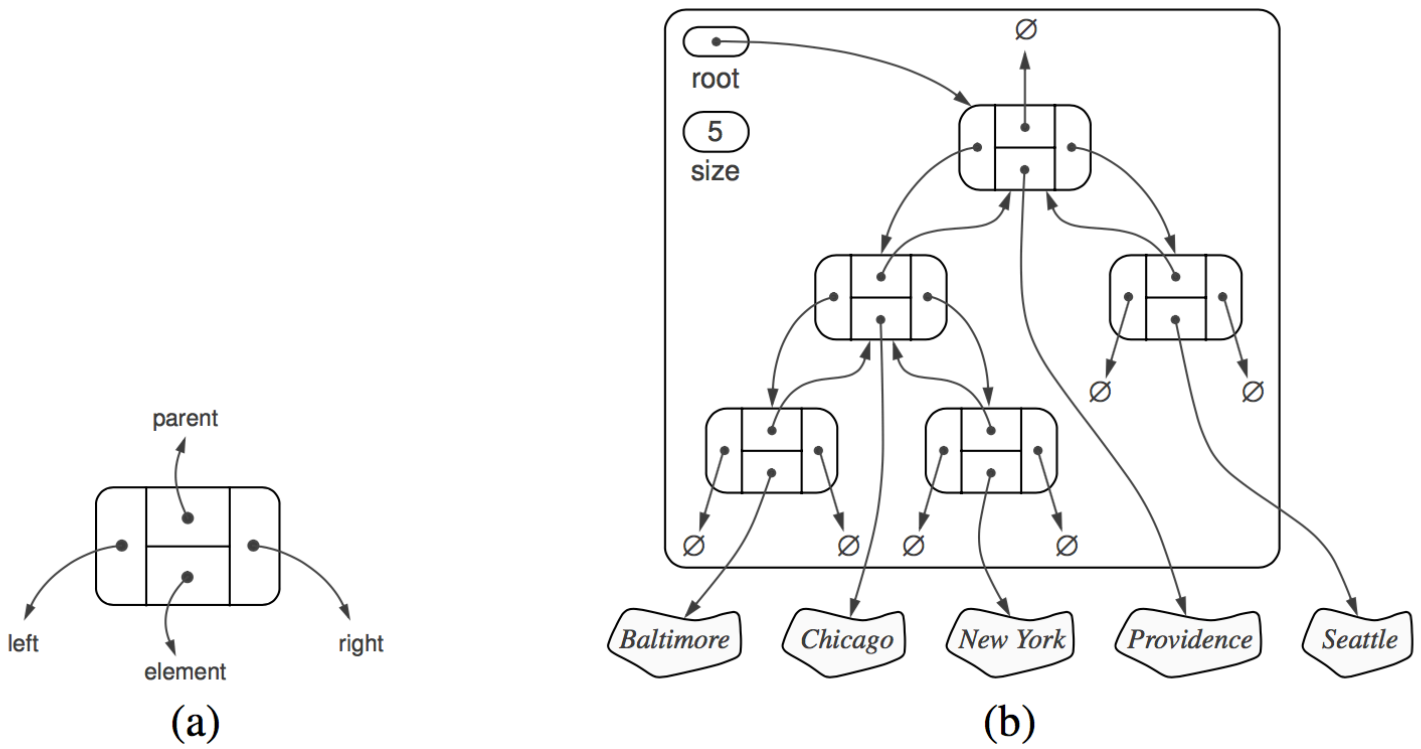

Figure 8.11: A linked structure for representing: (a) a single node; (b) a binary tree.

二叉树的链式结构

实现二叉树 T 的一种自然方法是使用链式结构，其中的节点维护

- 对存储在 p 位置的元素的引用；
- 对 p 的父节点和左右孩子节点的引用。

二叉树的链式结构

实现二叉树 T 的一种自然方法是使用链式结构，其中的节点维护

- 对存储在 p 位置的元素的引用；
- 对 p 的父节点和左右孩子节点的引用。

注意

- 若 p 是 T 的根，则其父引用是**None**。
- 若 p 没有左孩子 (或右孩子)，则其相关引用为**None**。

二叉树的链式结构

树本身维护

- 存储对根节点（如果有）的引用的实例变量，
- 表示 T 的节点总数的可变大小。

Operations for Updating a Linked Binary Tree

- `T.add_root(e)`:

  为空树创建一个根，存储元素 e，并返回该根的位置；若 T 非空，则报错。

Operations for Updating a Linked Binary Tree

- `T.add_root(e)`:

  为空树创建一个根，存储元素 e，并返回该根的位置；若 T 非空，则报错。

- `T.add_left(p, e)`:

  创建一个存储元素 e 的新节点，将其作为 p 的左孩子，并返回其位置；若 p 已有左孩子，则报错。

Operations for Updating a Linked Binary Tree

- `T.add_root(e)`:

  为空树创建一个根，存储元素 e，并返回该根的位置；若 T 非空，则报错。

- `T.add_left(p, e)`:

  创建一个存储元素 e 的新节点，将其作为 p 的左孩子，并返回其位置；若 p 已有左孩子，则报错。

- `T.add_right(p, e)`:

  创建一个存储元素 e 的新节点，将其作为 p 的右孩子，并返回其位置；若 p 已有右孩子，则报错。

Operations for Updating a Linked Binary Tree

- `T.replace(p, e)`:

  将存储在位置 p 的元素替换为元素 e，并返回原存储元素。

Operations for Updating a Linked Binary Tree

- `T.replace(p, e)`:

  将存储在位置 p 的元素替换为元素 e，并返回原存储元素。

- `T.delete(p)`:

  移除位置 p 处的节点，替换为其子节点 (如果有)，并返回存储在 p 处的元素；若 p 有两个孩子，则报错。

Operations for Updating a Linked Binary Tree

- `T.replace(p, e)`:

  将存储在位置 p 的元素替换为元素 e，并返回原存储元素。

- `T.delete(p)`:

  移除位置 p 处的节点，替换为其子节点 (如果有)，并返回存储在 p 处的元素；若 p 有两个孩子，则报错。

- `T.attach(p, T1, T2)`:

  将树 T_1 和 T_2 分别附加为 T 的叶子位置 p 的左右子树，并将 T_1 和 T_2 重置为空树；若 p 不是叶子，则报错。

二叉树的链式结构

```
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
```

二叉树的链式结构

```
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
```

二叉树的链式结构

```
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
```

二叉树的链式结构

```
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)
```

二叉树的链式结构

```
  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)
```

二叉树的链式结构

```
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
```

二叉树的链式结构

```
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)
```

二叉树的链式结构

```
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)
```

二叉树的链式结构

```
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)
```

二叉树的链式结构

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

二叉树的链式结构

```
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
```

二叉树的链式结构

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

二叉树的链式结构

```
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
```

树的实现

一般树的链式结构

一般树的链式结构

用链接结构表示二叉树时，每个节点都显式地维护对其左、右孩子的引用。

一般树的链式结构

用链接结构表示二叉树时，每个节点都显式地维护对其左、右孩子的引用。

对于一般树，节点可能拥有的孩子的数量没有先验限制。

用链式结构实现树 T 的自然方法是让每个节点存储对其子节点的引用的单个容器。

一般树的链式结构

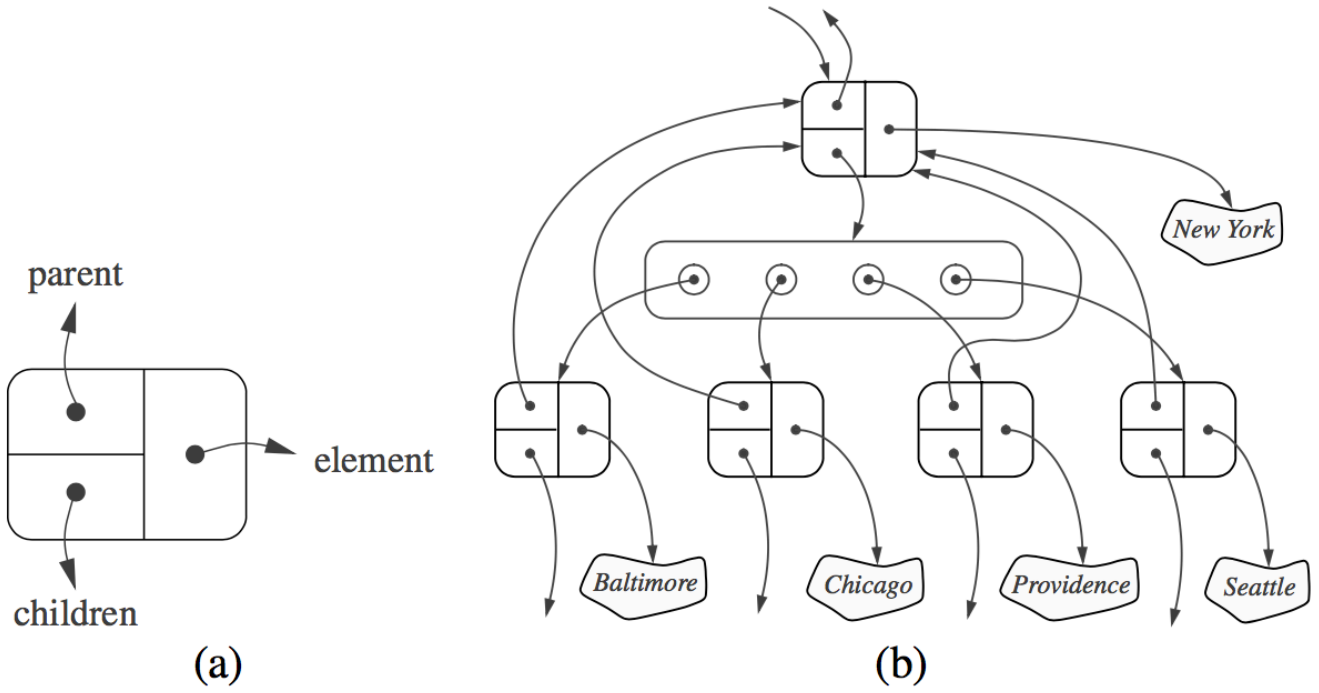

Figure 8.14: The linked structure for a general tree: (a) the structure of a node; (b) a larger portion of the data structure associated with a node and its children.

一般树的链式结构

Operation	Running Time
len, is_empty	$O(1)$
root, parent, is_root, is_leaf	$O(1)$
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 $O(n)$.

树的遍历算法

树的遍历算法

树的遍历是指访问树的所有位置，对位置 p 的访问操作取决于此遍历的应用.

树的遍历算法

一般树的前序遍历与后序遍历

一般树的前序遍历与后序遍历

在树 T 的前序遍历中，首先访问 T 的根，然后递归地遍历其子树。如果树是有序的，那么子树就按照子树的顺序遍历。

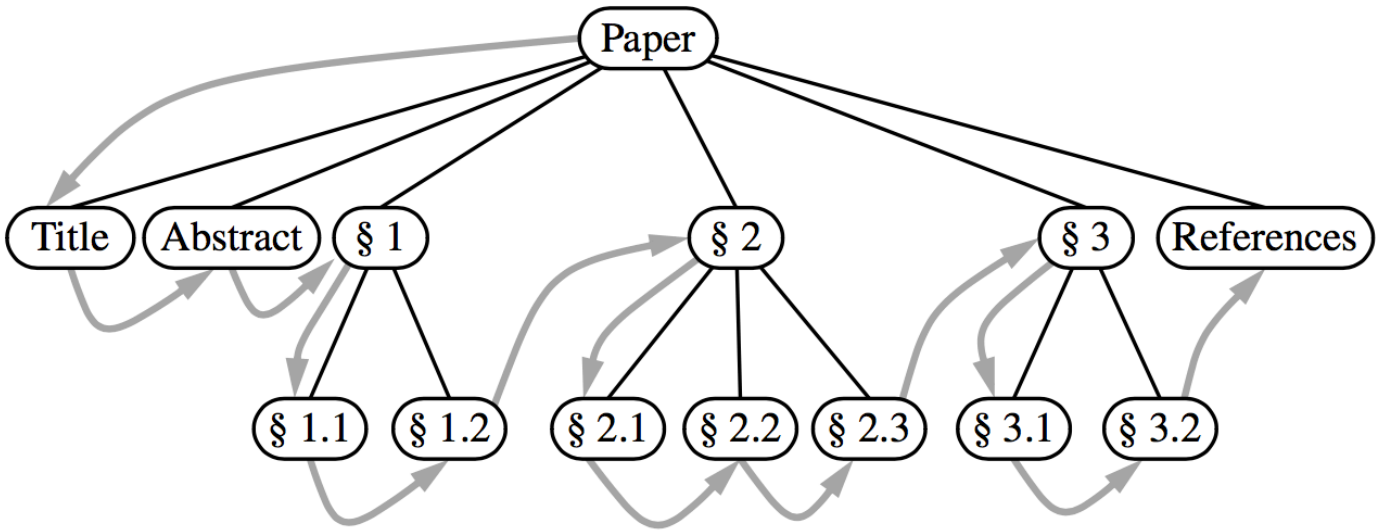

Figure 8.15: Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

一般树的前序遍历与后序遍历

```
class Tree(object):
```

```
  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
```

一般树的前序遍历与后序遍历

```
class Tree(object):
```

```
  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
```

一般树的前序遍历与后序遍历

前序和后序遍历算法都是访问树的所有位置的有效方法。

一般树的前序遍历与后序遍历

前序和后序遍历算法都是访问树的所有位置的有效方法。

在每个位置 p，遍历算法的非递归部分需要时间 $O(c_p+1)$，其中 c_p 是 p 的孩子数，这里假设“访问”本身需要 $O(1)$ 时间。

一般树的前序遍历与后序遍历

前序和后序遍历算法都是访问树的所有位置的有效方法。

在每个位置 p，遍历算法的非递归部分需要时间 $O(c_p + 1)$，其中 c_p 是 p 的孩子数，这里假设“访问”本身需要 $O(1)$ 时间。

遍历树 T 的总运行时间是 $O(n)$，其中 n 是树中的位置数。这个运行时间是渐近最优的，因为遍历必须访问树的所有 n 位置。

树的遍历算法

树的广度优先搜索

树的广度优先搜索

虽然前序遍历和后序遍历是访问树位置的常见方法，但还存在另一种常见方法-广度优先搜索

定义：广度优先搜索（Breadth-First Tree Traversal)

在访问深度 $d+1$ 的位置之前访问深度 d 的所有位置，这样的遍历算法被称为广度优先遍历。

树的广度优先搜索

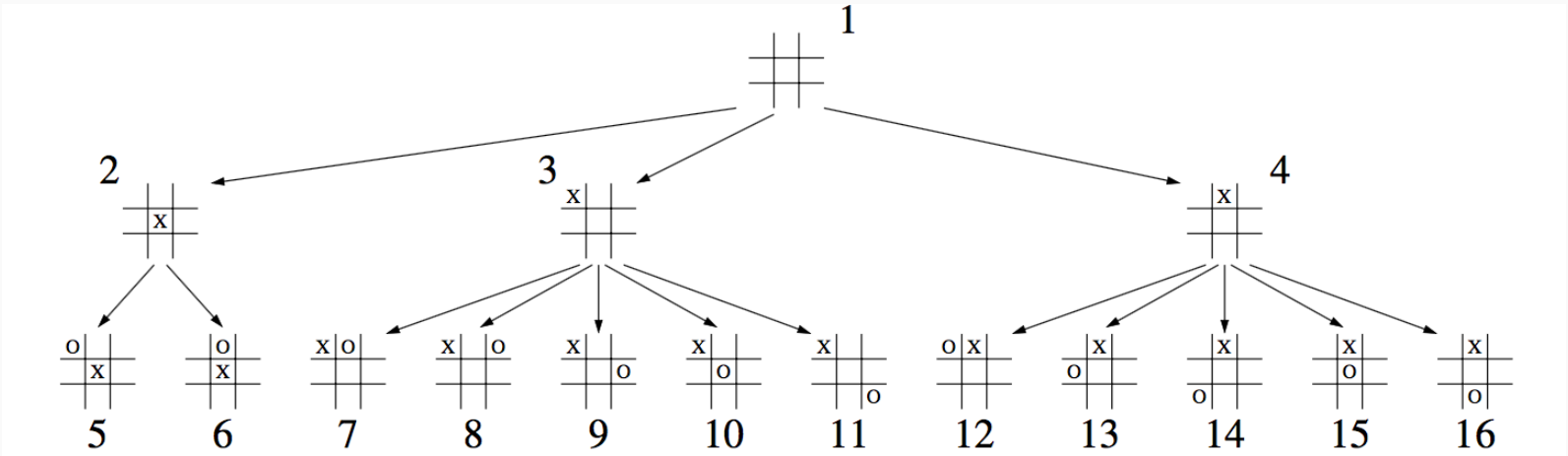

Figure 8.17: Partial game tree for Tic-Tac-Toe, with annotations displaying the order in which positions are visited in a breadth-first traversal.

树的广度优先搜索

```
class Tree(object):
```

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

树的遍历算法

二叉树的中序遍历

二叉树的中序遍历

在中序遍历期间，我们访问其左子树和右子树的递归遍历之间的位置。
二叉树 T 的顺序遍历可以非正式地看作是从左到右访问 T 的节点。

二叉树的中序遍历

在中序遍历期间，我们访问其左子树和右子树的递归遍历之间的位置。二叉树 T 的顺序遍历可以非正式地看作是从左到右访问 T 的节点。

实际上，对于每个位置 p，顺序遍历在 p 的左子树中的所有位置之后，在 p 的右子树中的所有位置之前访问 p。

二叉树的中序遍历

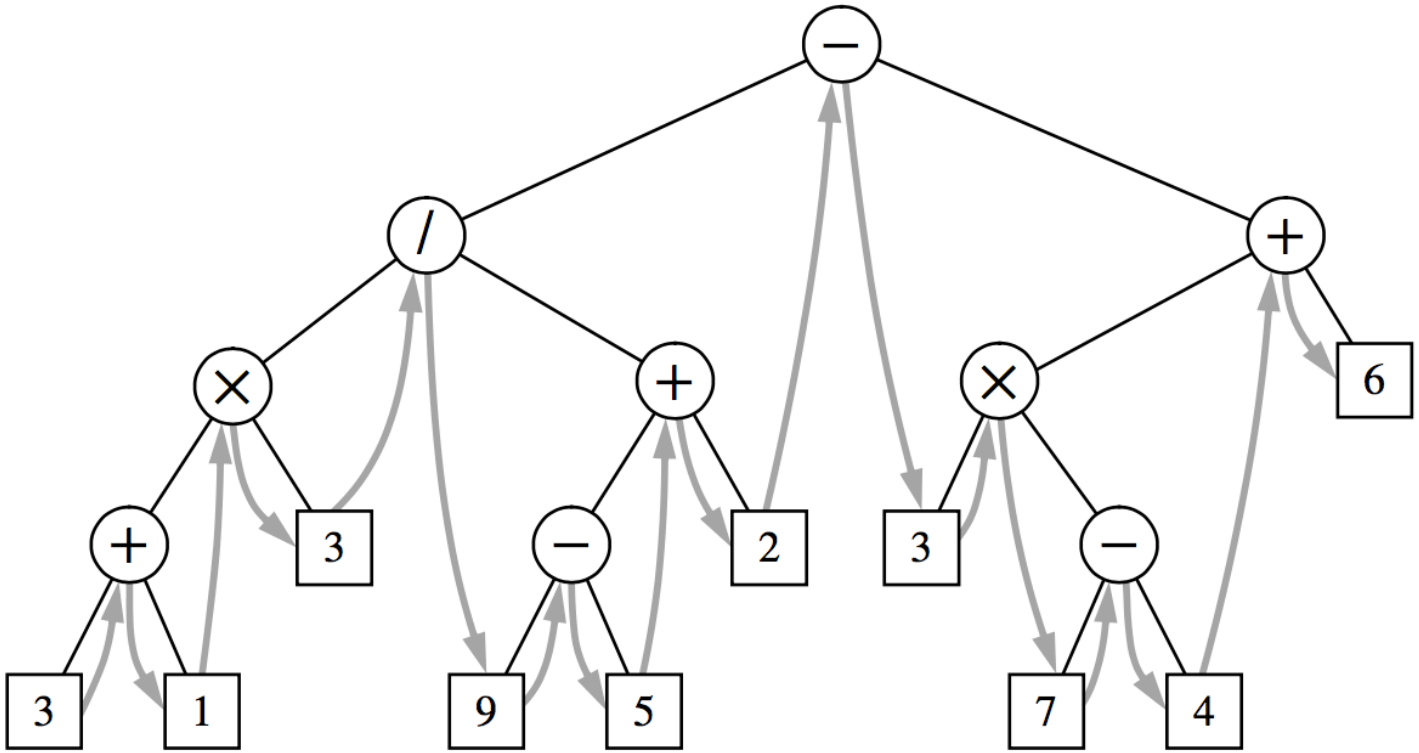

Figure 8.18: Inorder traversal of a binary tree.

二叉树的中序遍历

```
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)):
        yield other
```

Expression Tree

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)
```

Expression Tree

```
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(')')
```

Expression Tree

```
def __str__(self):
  pieces = []
  self._parenthesize_recur(self.root(), pieces
  )
  return ''.join(pieces)
```

Expression Tree

```
  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
```

Expression Tree

```
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()
```

Expression Tree

```
if __name__ == '__main__':
  big = build_expression_tree(tokenize('((((3 +
  1) * 3)/((9-5)+2))-((3x(7-4))+6))'))
  print(big, '=', big.evaluate())
```