

右子树。若每个节点都有零个或两个节点，则这样的二叉树为完全二叉树。一些人也把这种树称为满二叉树。因此，在完全二叉树中，每个内部节点都恰好有两个孩子。若二叉树不完全，则称为不完全二叉树。

例题 8-6：二叉树的一个重要的类适用于这样的情形：我们希望能（使用此类）表示许多种不同的输出结果，这些结果可以作为一系列 yes-or-no 问题的答案。每个内部节点对应一个问题。从根节点开始，我们根据该问题的答案是“Yes”还是“No”来决定当前节点是左孩子还是右孩子。对于每次决定，相当于选择了从父节点到子节点的一条边，最终能形成一条从根节点到叶节点的路径。这样的二叉树被称为决策树，因为若与树中叶节点 p 的祖先节点相关的问题都被回答，以得到 p 的结果，那么 p 即表示为一种需要做什么的决策。决策树是完全二叉树。图 8-7 给出了能给未来投资者提供建议的一棵决策树。

例题 8-7：二叉树能用于表示算术表达式，叶子对应变量或常数，内部节点对应 $+$ 、 $-$ 、 \times 和 $/$ 操作（见图 8-8）。树中的每个节点都对应一个值。

- 若节点为叶节点，则其值为变量或常数。
- 若节点为内部节点，则其值为对其孩子节点值的操作所得。

算术表达式树是完全二叉树，因为每个 $+$ 、 $-$ 、 \times 、 $/$ 都需要两个操作数。当然，如果允许一元操作符，例如负号 $(-)$ ，表示为“ $- \times$ ”，也可以得到不完全二叉树。

递归二叉树的定义

我们也能够顺便使用递归方式定义二叉树，此时二叉树或者为空树，或者由以下条件组成：

- 二叉树 T 的根为节点 r ，其存储一个元素。
- 二叉树（可能为空）称为 T 的左子树。
- 二叉树（可能为空）称为 T 的右子树。

8.2.1 二叉树的抽象数据类型

作为抽象数据类型，二叉树是树的一种特殊化，其支持 3 种额外的访问方法：

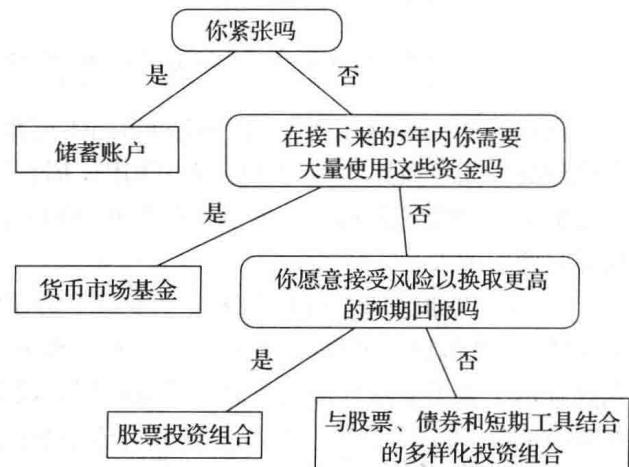


图 8-7 提供投资建议的决策树

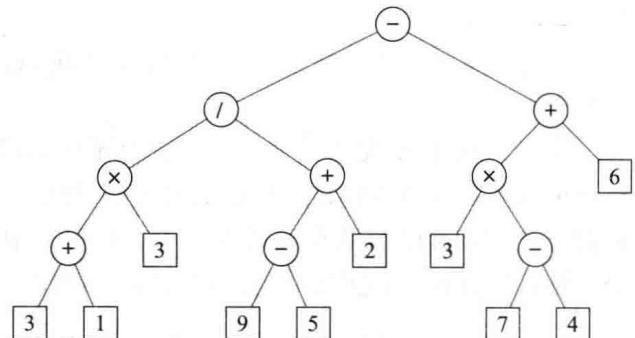


图 8-8 使用二叉树表示算术表达式。该树所表示的表达式为 $((((3 + 1) \times 3) / ((9 - 5) + 2)) - ((3 \times (7 - 4)) + 6))$ 。内部节点 “/” 对应的值是 2

- $T.\text{left}(p)$: 返回 p 左孩子的位置，若 p 没有左孩子，则返回 None 。
- $T.\text{right}(p)$: 返回 p 右孩子的位置，若 p 没有右孩子，则返回 None 。
- $T.\text{sibling}(p)$: 返回 p 兄弟节点的位置，若 p 没有兄弟节点，则返回 None 。

类似于 8.1.2 节对树 ADT 的处理，此处不专门对二叉树定义更新方法。而是在描述二叉树具体的实现和应用时，才去考虑一些可能的更新方法。

Python 中的抽象基类 `BinaryTree`

在 8.1.2 节中，我们将 `Tree` 定义为抽象基类。类似地，我们在已存在的 `Tree` 类基础上，依据继承性，对二叉树 ADT 定义一个新的 `BinaryTree` 类。然而，`BinaryTree` 类保持抽象性，因为对于这样的一个结构，我们并没有提供完整的内部细节描述，也没有实现一些必要的行为。

在代码段 8-7 中，我们给出了 `BinaryTree` 类的 Python 实现。根据继承性，二叉树支持在一般的树中定义的所有功能（例如，`parent`、`is_leaf` 和 `root`）。新类也继承嵌套的 `Position` 类，该类一开始就定义在 `Tree` 类的定义中。另外，新类声明了新的抽象方法 `left` 和 `right`，这些方法应能在 `BinaryTree` 类的具体子类中实现。

新类也给出了两种方法的具体实现。新的 `sibling` 方法由 `left`、`right` 和 `parent` 结合产生。具有代表性的是，我们把位置 p 的兄弟节点定义为 p 双亲节点的“另一个”孩子节点。若 p 是根节点，因为没有双亲节点，所以也没有兄弟节点。另外， p 可能是其双亲节点唯一的儿子，因而此时也无兄弟节点。

最后，代码段 8-7 给出了 `children` 方法的具体实现，该方法在 `Tree` 类中是抽象的。尽管我们仍未具体说明节点的孩子是如何存储的，但能通过抽象的 `left` 和 `right` 方法的隐含行为产生有序的孩子。

代码段 8-7 从代码段 8-1 和 8-2 已存在的 Tree 抽象基类中扩展的 `BinaryTree` 抽象基类

```

1 class BinaryTree(Tree):
2     """Abstract base class representing a binary tree structure."""
3
4     # ----- additional abstract methods -----
5     def left(self, p):
6         """Return a Position representing p's left child.
7
8         Return None if p does not have a left child.
9         """
10    raise NotImplementedError('must be implemented by subclass')
11
12    def right(self, p):
13        """Return a Position representing p's right child.
14
15        Return None if p does not have a right child.
16        """
17    raise NotImplementedError('must be implemented by subclass')
18
19    # ----- concrete methods implemented in this class -----
20    def sibling(self, p):
21        """Return a Position representing p's sibling (or None if no sibling)."""
22        parent = self.parent(p)
23        if parent is None:
24            return None
25        else:
26            if p == self.left(parent):
27                return self.right(parent)

```

```

28     else:
29         return self.left(parent)           # possibly None
30
31     def children(self, p):
32         """Generate an iteration of Positions representing p's children."""
33         if self.left(p) is not None:
34             yield self.left(p)
35         if self.right(p) is not None:
36             yield self.right(p)

```

8.2.2 二叉树的属性

二叉树在处理其高度和节点数的关系时有几个有趣的性质。我们将位于树 T 同一深度 d 的所有节点都视为位于 T 的 d 层。在二叉树中，0 层至多有一个节点（根节点），1 层至多有两个节点（根节点的孩子），2 层至多有 4 个节点，以此类推（见图 8-9）。总之， d 层至多有 2^d 个节点。

我们注意到，当沿着二叉树往下遍历时，每层的最大节点数呈指数增长。通过这个简单的观察，我们可以得出二叉树 T 的高度与节点数之间的性质。这些性质的详细证明留作练习 R-8.8。

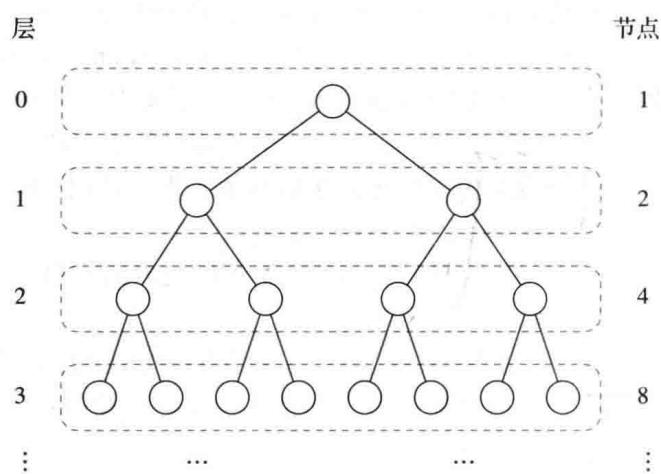


图 8-9 二叉树每层之间的最大节点数

命题 8-8：设 T 为非空二叉树， n 、

n_E 、 n_I 和 h 分别表示 T 的节点数、外部节点数、内部节点数和高度，则 T 具有如下性质：

- 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) $\log(n + 1) - 1 \leq h \leq n - 1$

另外，若 T 是完全二叉树，则 T 具有如下性质：

- 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) $\log(n + 1) - 1 \leq h \leq (n - 1)/2$

完全二叉树中内部节点与外部节点的关系

除了前面二叉树的性质，下述关系存在于完全二叉树中内部节点数与外部节点数之间。

命题 8-9：在非空完全二叉树 T 中，有 n_E 个外部节点和 n_I 个内部节点，则有 $n_E = n_I + 1$ 。

证明：从 T 中取下节点，并把它们分别放入两个“桩”，即内部节点桩和外部节点桩，直到 T 为空。两个桩初始都为空。执行到最后，我们会发现外部节点桩比内部节点桩多一个节点。考虑以下两种情况。

情况 1：若 T 仅有一个节点 v ，我们将 v 取下，并把它放入外部节点桩。因此，外部节

点桩有一个节点，而内部节点桩为空。

情况 2：另外 (T 多于一个节点)，我们从 T 中取下一个（任意的）外部节点 w 和其父母节点 v ， v 为内部节点。我们将 w 放入外部节点桩，将 v 放入内部节点桩。若 v 有父母节点 u ，则将 u 与 w 之前的兄弟节点 z 连接起来，如图 8-10 所示。此次操作取下了一个内部节点和一个外部节点，并使树变成新的完全二叉树。重复上述操作，我们最后将会得到仅有一个节点的最终树。注意，在经过这样一系列操作并得到最终树的过程中，相同数目的外部节点和内部节点被分别放入各自的桩中。现在，我们将最终树的节点取下并放入外部节点桩中。因此，外部节点桩比内部节点桩多一个节点。 ■

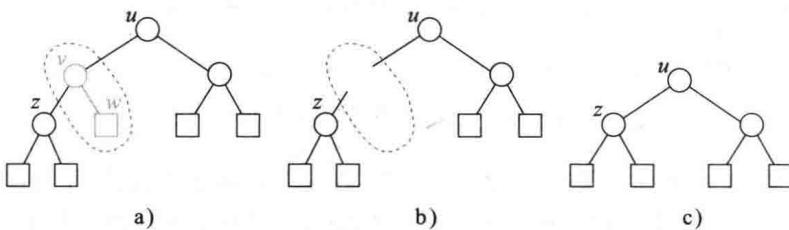


图 8-10 取下外部节点和其父母节点的操作，该操作运用于命题 8-9 的证明过程

注意：上述关系一般不适用于不完全二叉树和非二叉树，而其他有趣的关系则能适用（见练习 C-8.32 ~ C-8.34）。

8.3 树的实现

到目前为止，本章所定义的 Tree 和 BinaryTree 类都只是形式上的抽象基类。尽管给出了许多支持操作，但它们都不能直接被实例化。对于树内部如何表示，以及如何高效地在父母节点和孩子节点之间进行切换，我们还没有定义关键的实现细节。特别地，具体实现树要能提供 Root、parent、num_children、children 和 __len__ 这些方法，对于 BinaryTree 类，还要提供额外的访问器 left 和 right。

对于树的内部表示有几种选择。本节介绍最普遍的表示方法。我们先以二叉树为例进行介绍，因为它的形状更有局限性。

8.3.1 二叉树的链式存储结构

实现二叉树 T 的一个自然方法便是使用链式存储结构，一个节点（见图 8-11a）包含多个引用：指向存储在位置 p 的元素的引用，指向 p 的孩子节点和双亲节点的引用。若 p 是 T 的根节点，则 p 的 parent 字段为 None。同样，若 p 没有左孩子（或右孩子），则相关字段即为 None。树本身包含一个实例变量，存储指向根节点（假如存在根节点）的引用，还包含一个 size 变量，表示 T 的所有节点数。在图 8-11b 中，我们给出了表示二叉树的链式存储结构。

链式二叉树结构的 Python 实现

在本节中，我们定义 BinaryTree 类的一个具体子类 LinkedBinaryTree，该类能够实现二叉树 ADT。通用方法非常类似于 7.4 节中开发 PositionalList 时所采用的方法：定义一个简单、非公开的 _Node 类表示一个节点，再定义一个公开的 Position 类用于封装节点。我们提供 _validate 方法，在所给的 position 实例未封装前，能够强有力地验证该实例的有效性。另外，我们也提供 _make_position 方法，把节点封装进 position 实例，并返回给调用者。

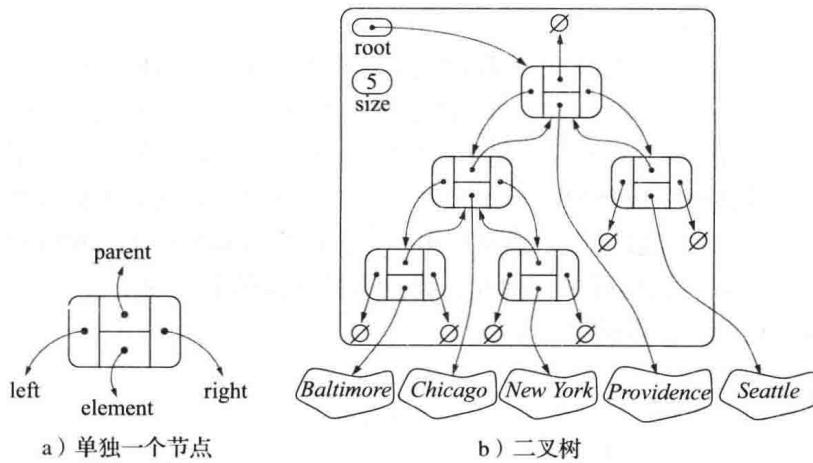


图 8-11 链式存储结构表示

代码段 8-8 给出了这些定义。从形式上说，新的 Position 类被声明为直接继承 BinaryTree.Position 类。而从技术上说，BinaryTree 类的定义（见代码段 8-7）并未正式声明这样的一个内嵌类，它仅仅平凡地继承于 Tree.Position。这样设计的一个细微优势在于：Position 类能够继承 `__ne__` 这一特殊方法，以至于相对于 `__eq__` 方法，语句 `p!=q` 能够自然地执行。

代码段 8-8 LinkedBinaryTree 类的开始 (后接代码段 8-9 ~ 8-11)

```

1 class LinkedBinaryTree(BinaryTree):
2     """Linked representation of a binary tree structure."""
3
4     class _Node:      # Lightweight, nonpublic class for storing a node.
5         __slots__ = '_element', '_parent', '_left', '_right'
6         def __init__(self, element, parent=None, left=None, right=None):
7             self._element = element
8             self._parent = parent
9             self._left = left
10            self._right = right
11
12    class Position(BinaryTree.Position):
13        """An abstraction representing the location of a single element."""
14
15        def __init__(self, container, node):
16            """Constructor should not be invoked by user."""
17            self._container = container
18            self._node = node
19
20        def element(self):
21            """Return the element stored at this Position."""
22            return self._node._element
23
24        def __eq__(self, other):
25            """Return True if other is a Position representing the same location."""
26            return type(other) is type(self) and other._node is self._node
27
28        def __ne__(self, other):
29            """Return associated node, if position is valid."""
30            if not isinstance(p, self.Position):
31                raise TypeError('p must be proper Position type')
32            if p._container is not self:
33                raise ValueError('p does not belong to this container')

```

```

34     if p._node._parent is p._node:          # convention for deprecated nodes
35         raise ValueError('p is no longer valid')
36     return p._node
37
38 def _make_position(self, node):
39     """Return Position instance for given node (or None if no node)."""
40     return self.Position(self, node) if node is not None else None

```

在代码段 8-9 中，对类继续定义了构造函数，并且对 Tree 和 BinaryTree 类的抽象方法做了具体实现。构造函数通过将 _root 初始化为 None、将 _size 初始化为 0，能够创建一棵空树。实现访问方法时，谨慎使用了 _validate 和 _make_position，防止出现边界问题。

代码段 8-9 LinkedBinaryTree 类的公开访问方法。该类从代码段 8-8 开始，且
在代码段 8-10 和 8-11 中继续

```

41 #----- binary tree constructor -----
42 def __init__(self):
43     """Create an initially empty binary tree."""
44     self._root = None
45     self._size = 0
46
47 #----- public accessors -----
48 def __len__(self):
49     """Return the total number of elements in the tree."""
50     return self._size
51
52 def root(self):
53     """Return the root Position of the tree (or None if tree is empty)."""
54     return self._make_position(self._root)
55
56 def parent(self, p):
57     """Return the Position of p's parent (or None if p is root)."""
58     node = self._validate(p)
59     return self._make_position(node._parent)
60
61 def left(self, p):
62     """Return the Position of p's left child (or None if no left child)."""
63     node = self._validate(p)
64     return self._make_position(node._left)
65
66 def right(self, p):
67     """Return the Position of p's right child (or None if no right child)."""
68     node = self._validate(p)
69     return self._make_position(node._right)
70
71 def num_children(self, p):
72     """Return the number of children of Position p."""
73     node = self._validate(p)
74     count = 0
75     if node._left is not None:      # left child exists
76         count += 1
77     if node._right is not None:    # right child exists
78         count += 1
79     return count

```

更新链式二叉树的操作

至此，我们已经给出了用于操作已存在二叉树的函数。而 LinkedBinaryTree 类的构造函数创建了一棵空树，我们没有提供任何改变这种结构的方法，也没有提供任何填充这棵树的方法。

在 Tree 和 BinaryTree 抽象基类中，我们没有声明更新方法的原因如下。

首先，虽然封装原则表明类的外部行为不需要依赖于类的内部实现，而操作的效率却极大地取决于实现方式。我们更倾向于 Tree 类的每个具体实现都能提供更合适的选择方式来更新一棵树。

其次，我们可能不希望更新方法成为公开接口。树有许多应用，适用于其中一个应用的更新操作可能不被另一个应用所接受。而假如我们在基类中声明更新方法，继承于该基类的任何子类都将继承这一方法。例如，考虑方法 `T.replace(p, e)` 的可能性，该方法用元素 `e` 替换存储于位置 `p` 的元素。这种一般性的方法可能不适用于算术表达式树（见例题 8-7，在 8.5 节中，我们将会学习另一个例子）的情形，因为我们可能会强制内部节点仅存储一个运算符。

对于链式二叉树，支持日常使用的合理更新方法如下：

- `T.add_root(e)`：为空树创建根节点，存储元素 `e`，并返回根节点的位置。若树非空，则抛出错误。
- `T.add_left(p, e)`：创建新的节点，存储元素 `e`，将该节点链接为位置 `p` 的左孩子，返回结果位置。若 `p` 已经有左孩子，则抛出错误。
- `T.add_right(p, e)`：创建新的节点，存储元素 `e`，将该节点链接为位置 `p` 的右孩子，返回结果位置；若 `p` 已经有右孩子，则抛出错误。
- `T.replace(p, e)`：用元素 `e` 替换存储在位置 `p` 的元素，返回之前存储的元素。
- `T.delete(p)`：移除位置为 `p` 的节点，用它的孩子代替自己，若有，则返回存储在位置 `p` 的元素；若 `p` 有两个孩子，则抛出错误。
- `T.attach(p, T1, T2)`：将树 `T1, T2` 分别链接为 `T` 的叶子节点 `p` 的左右子树，并将 `T1` 和 `T2` 重置为空树；若 `p` 不是叶子节点，则抛出错误。

之所以专门选择这组操作，是因为使用链接表示时，每个操作的最坏运行时间为 $O(1)$ 。其中最复杂的操作是 `delete` 和 `attach` 操作，因为要分析有关的各种双亲 - 孩子关系的问题和边界条件问题，还要保证执行固定的操作数。（类似于对位置列表的处理，若使用树的前哨节点表示法，则这两种方法的实现过程将大大简化，见练习 C-8.40。）

为了避免不必要的更新方法被 `LinkedBinaryTree` 的子类所继承，我们选择所有方法均不采用公开支持的实现方式。换言之，我们对每种方法都提供非公开的形式，例如，使用带下划线的 `_delete` 方法来替换公开的 `delete` 方法。代码段 8-10 和代码段 8-11 给出了 6 种更新方法的实现方式。

代码段 8-10 `LinkedBinaryTree` 类的非公开更新方法（后接代码段 8-11）

```

80  def _add_root(self, e):
81      """Place element e at the root of an empty tree and return new Position.
82
83      Raise ValueError if tree nonempty.
84      """
85      if self._root is not None: raise ValueError('Root exists')
86      self._size = 1
87      self._root = self._Node(e)
88      return self._make_position(self._root)
89
90  def _add_left(self, p, e):
91      """Create a new left child for Position p, storing element e.
92

```

```

93     Return the Position of new node.
94     Raise ValueError if Position p is invalid or p already has a left child.
95     """
96     node = self._validate(p)
97     if node._left is not None: raise ValueError('Left child exists')
98     self._size += 1
99     node._left = self._Node(e, node)           # node is its parent
100    return self._make_position(node._left)
101
102   def _add_right(self, p, e):
103       """Create a new right child for Position p, storing element e.
104
105       Return the Position of new node.
106       Raise ValueError if Position p is invalid or p already has a right child.
107       """
108       node = self._validate(p)
109       if node._right is not None: raise ValueError('Right child exists')
110       self._size += 1
111       node._right = self._Node(e, node)          # node is its parent
112       return self._make_position(node._right)
113
114   def _replace(self, p, e):
115       """Replace the element at position p with e, and return old element."""
116       node = self._validate(p)
117       old = node._element
118       node._element = e
119       return old

```

代码段 8-11 LinkedBinaryTree 类的非公开更新方法（接代码段 8-10）

```

120  def _delete(self, p):
121      """Delete the node at Position p, and replace it with its child, if any.
122
123      Return the element that had been stored at Position p.
124      Raise ValueError if Position p is invalid or p has two children.
125      """
126      node = self._validate(p)
127      if self._num_children(p) == 2: raise ValueError('p has two children')
128      child = node._left if node._left else node._right           # might be None
129      if child is not None:
130          child._parent = node._parent    # child's grandparent becomes parent
131          if node is self._root:
132              self._root = child          # child becomes root
133          else:
134              parent = node._parent
135              if node is parent._left:
136                  parent._left = child
137              else:
138                  parent._right = child
139              self._size -= 1
140              node._parent = node          # convention for deprecated node
141              return node._element
142
143  def _attach(self, p, t1, t2):
144      """Attach trees t1 and t2 as left and right subtrees of external p."""
145      node = self._validate(p)
146      if not self._is_leaf(p): raise ValueError('position must be leaf')
147      if not type(self) is type(t1) is type(t2): # all 3 trees must be same type
148          raise TypeError('Tree types must match')
149      self._size += len(t1) + len(t2)
150      if not t1._is_empty():                 # attached t1 as left subtree of node
151          t1._root._parent = node

```

```

152     node._left = t1._root
153     t1._root = None          # set t1 instance to empty
154     t1._size = 0
155     if not t2.is_empty():      # attached t2 as right subtree of node
156         t2._parent = node
157         node._right = t2._root
158     t2._root = None          # set t2 instance to empty
159     t2._size = 0

```

在特定的应用程序中，`LinkedBinaryTree` 的子类能调用内部非公开的方法，并提供适用于应用程序的公开接口。子类也可以使用公开方法封装一个或多个非公开更新方法供用户调用。我们将会在练习 R-8.15 中要求定义 `MutableLinkedBinaryTree` 这一子类，该子类能够提供封装 6 种公开更新方法的任意一种。

链式二叉树实现方式的性能

为了总结链式结构表示法的效率，我们分析 `LinkedBinaryTree` 方法的运行时间，其中包括从 `Tree` 和 `BinaryTree` 类派生的方法：

- `len` 方法，在 `LinkedBinaryTree` 内部实现，使用一个实例变量存储 T 的节点数，花费 $O(1)$ 的时间。`is_empty` 方法继承自 `Tree` 类，对 `len` 方法进行一次调用，因此需要花费 $O(1)$ 的时间。
- 访问方法 `root`、`left`、`right`、`parent` 和 `num_children` 直接在 `LinkedBinaryTree` 中执行，花费 $O(1)$ 的时间。`sibling` 和 `children` 方法从 `BinaryTree` 类派生，对其他访问方法做固定次的调用，因此，它们的运行时间也是 $O(1)$ 。
- `Tree` 类的 `is_root` 和 `is_leaf` 方法都运行 $O(1)$ 的时间，因为 `is_root` 调用 `root` 方法，之后判定两者的位置是否相等；而 `is_leaf` 调用 `left` 和 `right` 方法，并验证二者是否返回 `None`。
- `depth` 和 `height` 方法在 8.1.3 节中已做过分析。`depth` 方法在位置 p 处运行 $O(d_p + 1)$ 的时间，其中 d_p 是它的深度；`height` 方法在树的根节点处运行 $O(n)$ 的时间。
- 各种更新方法 `add_root`、`add_left`、`add_right`、`replace`、`delete` 和 `attach`（即它们的非公开实现方式）都运行 $O(1)$ 的时间，因为它们每次操作都仅仅重新链接到固定的节点数。

表 8-1 总结了二叉树链式存储结构实现方式的性能。

表 8-1 使用链接结构表示的 n 节点二叉树的各种方法的运行时间。空间占用为 $O(n)$

操作	运行时间
<code>len, is_empty</code>	$O(1)$
<code>root, parent, left, right, sibling, children, num_children</code>	$O(1)$
<code>is_root, is_leaf</code>	$O(1)$
<code>depth(p)</code>	$O(d_p + 1)$
<code>height</code>	$O(n)$
<code>add_root, add_left, add_right, replace, delete, attach</code>	$O(1)$

8.3.2 基于数组表示的二叉树

二叉树 T 的一种可供选择的表示法是对 T 的位置进行编号。对于 T 的每个位置 p ，设 $f(p)$ 为整数且定义如下：

- 若 p 是 T 的根节点，则 $f(p)=0$ 。
- 若 p 是位置 q 的左孩子，则 $f(p)=2f(q) + 1$ 。
- 若 p 是位置 q 的右孩子，则 $f(p)=2f(q) + 2$ 。

编号函数 f 被称为二叉树 T 的位置的层编号，因为它将 T 每一层的位置从左往右按递增顺序编号（见图 8-12）。注意，层编号是基于树内的潜在位置，而不是所给树的实际位置，因此编号不一定是连续的。例如，在图 8-12b 中，没有层编号为 13 或 14 的节点，因为层编号为 6 的节点没有孩子。

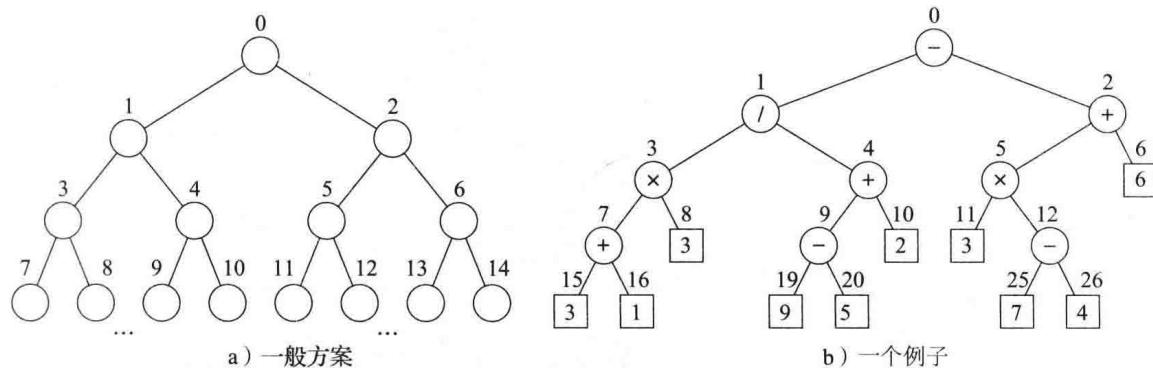


图 8-12 二叉树的层编号

层编号函数 f 是一种二叉树 T 依据基于数组结构 A （例如，Python 列表）的表示方法， T 的 p 位置元素存储在数组下标为 $f(p)$ 的内存中。在图 8-13 中，我们给出了一个二叉树基于数组表示的例子。

二叉树基于数组的表示方式的一个优势在于位置 p 能用简单的整数 $f(p)$ 来表示，且基于位置的方法（如 `root`、`parent`、`left` 和 `right` 方法）能采用对编号 $f(p)$ 进行简单算术操作的方法来执行。根据层编号的公式， p 左孩子的下标为 $2f(p) + 1$ ，右孩子的下标为 $2f(p) + 2$ ，而 p 父母的下标为 $\lfloor f(p) - 1/2 \rfloor$ 。我们将完整实现方式的细节留作练习 R-8.18。

基于数组表示的空间使用情况极大地依赖于树的形状。设 n 为树 T 的节点数， f_M 为 $f(p)$ 对于 T 所有节点的最大值。数组 A 所需长度为 $N = 1 + f_M$ ，因为元素范围为从 $A[0]$ 到 $A[f_M]$ 。注意， A 可以有多个空单元，未指向 T 的已有节点。事实上，在最坏情况下， $N=2^n - 1$ ，证明过程留作练习 R-8.16。在 9.3 节中，我们将学习二叉树的 `heaps` 类，其中 $N = n$ 。因此，即使是最坏情况下的空间使用，仍有应用程序指明二叉树的数组表示是空间高效的。而对于一般的二叉树而言，这种表示方式的指数级最坏空间需求是不允许的。

数组表示的另一个缺点是不能有效地支持树的一些更新方法，例如删除节点且提升自己的孩子节点的编号需要花费 $O(n)$ 的时间，因为在数组中，不仅有孩子节点需要移动位置，该孩子节点的所有子孙也都要移动。

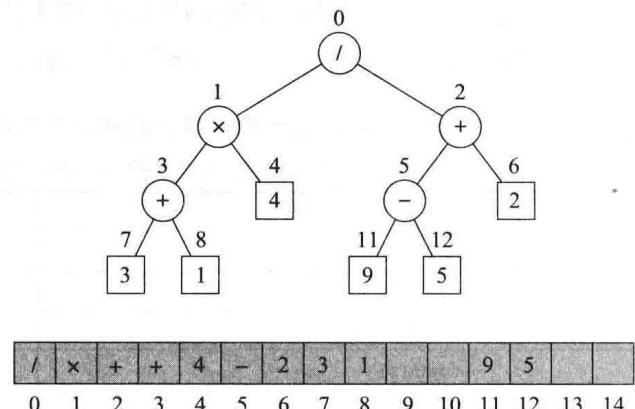


图 8-13 二叉树基于数组的表示方式

8.3.3 一般树的链式存储结构

当使用链式存储结构表示二叉树时，每个节点都明确包含了 left 和 right 字段，用于指向各自的孩子节点。对于一般树，一个节点所拥有的孩子节点之间没有优先级限制。使用链式存储结构实现一般树 T 的一个很自然的方法是：使每个节点都配置一个容器，该容器存储指向每个孩子的引用。例如，节点的 children 字段可以是一张 Python 列表，用于存储指向该节点孩子（若有）的引用。图 8-14 阐明了这种链式表示。

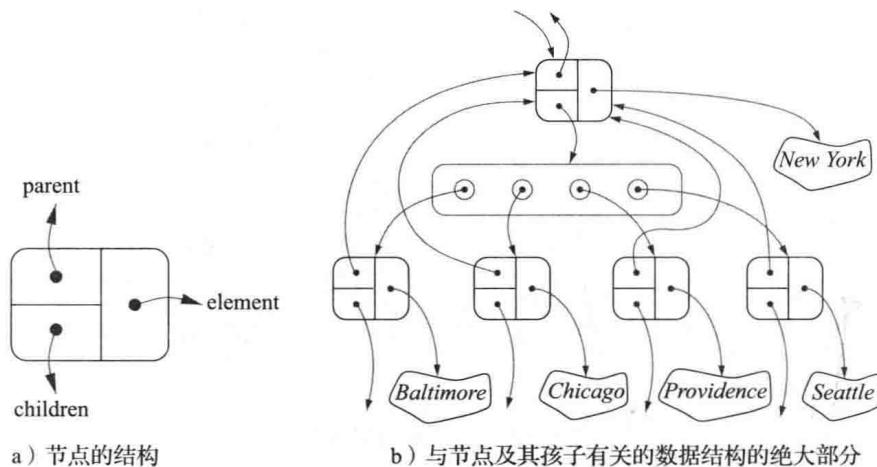


图 8-14 一般树的链式结构

表 8-2 总结了使用链式存储结构实现一般树的性能。分析过程留作练习 R-8.14，但需要注意，使用集合存储每个位置 p 的孩子时，我们可以使用简单的迭代来实现 $\text{children}(p)$ 。

表 8-2 使用链式存储结构实现的具有 n 个节点的一般树的各种访问方法的运行时间。

我们设 c_p 表示位置 p 的孩子节点数。空间占用为 $O(n)$

操作	运行时间
<code>len, is_empty</code>	$O(1)$
<code>root, parent, is_root, is_leaf</code>	$O(1)$
<code>children(p)</code>	$O(c_p + 1)$
<code>depth(p)</code>	$O(d_p + 1)$
<code>height</code>	$O(n)$

8.4 树的遍历算法

树 T 的遍历是访问或者“拜访” T 的所有位置的一种系统化方法。“访问” p 位置的相关具体行动取决于遍历的应用程序，并且可能包括任何计数器为 p 执行一些复杂的运算。在本节中，我们描述了几种常见的树的遍历方案，并在各种树类的环境中实现它们，还讨论了几种树遍历的常见应用。

8.4.1 树的先序和后序遍历

在树 T 的先序遍历中，首先访问 T 的根，然后递归地访问子树的根。如果这棵树是有序的，则根据孩子的顺序遍历子树。对于 p 位置处子树的根的先序遍历，其伪代码如代码段

8-12 所示。

代码段 8-12 T 树 p 位置的子树根的先序遍历的 preorder 算法

```

Algorithm preorder( $T, p$ ):
    perform the “visit” action for position  $p$ 
    for each child  $c$  in  $T.\text{children}(p)$  do
        preorder( $T, c$ ) {recursively traverse the subtree rooted at  $c$ }

```

图 8-15 描述了在一个先序遍历算法的应用中样本树的位置被顺序访问。

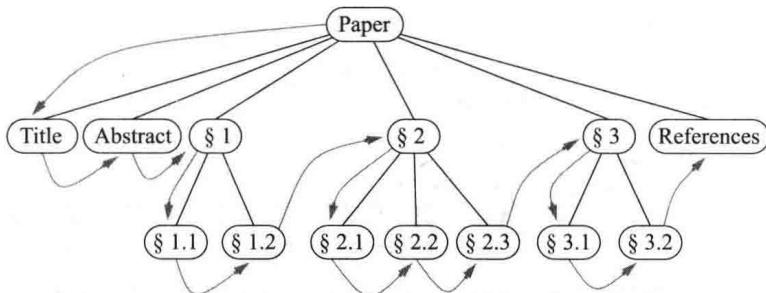


图 8-15 顺序树的先序遍历，每个位置处的孩子节点从左到右被排序

后序遍历

另一个重要的树的遍历算法是后序遍历。在某种程度上，这种算法可以看作相反的先序遍历，因为它优先遍历子树的根，即首先从孩子的根开始，然后访问根（因此叫作后序）。后序遍历的伪代码如代码段 8-13 所示，图 8-16 描绘了一个后序遍历的例子。

代码段 8-13 执行树 T 根在 p 位置处的后序遍历的 postorder 算法

```

Algorithm postorder( $T, p$ ):
    for each child  $c$  in  $T.\text{children}(p)$  do
        postorder( $T, c$ ) {recursively traverse the subtree rooted at  $c$ }
    perform the “visit” action for position  $p$ 

```

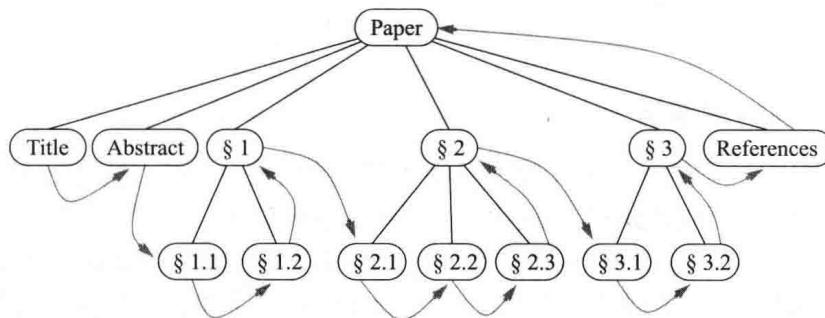


图 8-16 图 8-15 所示的顺序树的后序遍历

运行时间分析

先序遍历和后序遍历的算法对于访问树的所有位置都是有效的，对这两种算法的分析和 height2 算法是相似的，如 8.1.3 节的代码段 8-5 所示。在每个 p 位置，遍历算法中的非递归部分所需的时间为 $O(c_p + 1)$ ， c_p 是指 p 位置处孩子的个数，假设访问本身需要 $O(1)$ 的时间。由命题 8-5 可知，树 T 的整体运行时间为 $O(n)$ ，其中 n 是树中位置的数量。这个运行时间是

最佳的，因为遍历必须经过树的 n 个位置。

8.4.2 树的广度优先遍历

在访问树的位置时先序遍历和后序遍历是常见的方法，另一种常见的是遍历树的方法是在访问深度 d 的位置之前先访问深度 $d+1$ 的位置。这种算法称为广度优先遍历。

广度优先遍历广泛应用于游戏软件上，在游戏（或计算机）中，博弈树代表了可选择的一些动作，树的根是游戏的初始配置。例如，图 8-17 所示即为井字棋的部分博弈树。

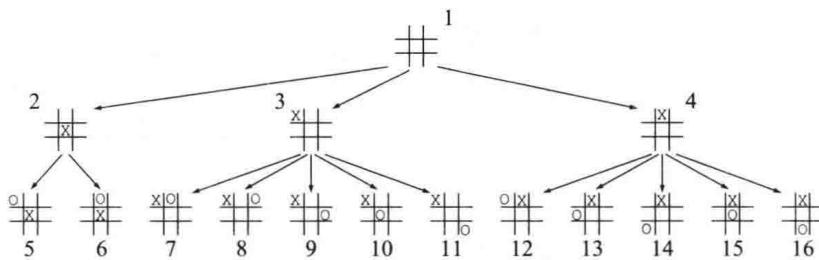


图 8-17 井字棋的部分博弈树，注释位置显示的访问顺序是广度优先遍历

之所以常执行这样一个博弈树的广度优先遍历，是因为计算机无法在有限的时间内去挖掘完整的博弈树。所以计算机要考虑所有动作，然后在允许的计算时间内对这些动作进行回馈。

广度遍历的伪代码如代码段 8-14 所示。这个过程不是递归的，因为我们不是首先遍历整个子树。我们使用一个队列来产生 FIFO（即先进先出）访问节点的顺序语义。整体的运行时间为 $O(n)$ ，因为对 enqueue 和 dequeue 操作各调用了 n 次。

代码段 8-14 执行树的广度优先遍历算法

```

Algorithm breadthfirst(T):
    Initialize queue Q to contain T.root()
    while Q not empty do
        p = Q.dequeue()           {p is the oldest entry in the queue}
        perform the "visit" action for position p
        for each child c in T.children(p) do
            Q.enqueue(c)          {add p's children to the end of the queue for later visits}

```

8.4.3 二叉树的中序遍历

前面介绍的对于一般树的标准先序、后序和广度的优先遍历能直接应用在二叉树中。在这节中，我们介绍另一种常见的专门应用于二叉树的遍历算法。

在中序遍历中，我们通过递归遍历左右子树去访问一个位置。二叉树的中序遍历可以看作“从左到右”非正式地访问 T 的节点。事实上，对于每个位置 p ， p 将在其左子树之后及其右子树之前被中序遍历访问。中序遍历算法的伪码如代码段 8-15 所示。图 8-18 描述了中序遍历的一个例子。

代码段 8-15 根在二叉树 p 位置处的子树的 inorder 算法的执行

```

Algorithm inorder(p):
    if p has a left child lc then
        inorder(lc)           {recursively traverse the left subtree of p}
    perform the "visit" action for position p
    if p has a right child rc then
        inorder(rc)          {recursively traverse the right subtree of p}

```

中序遍历的算法有几个重要的应用。使用二叉树表示一个算术表达式，如图 8-18 所示，中序遍历访问的位置与标准的顺序表达式的顺序一致，例如 $3 + 1 \times 3 / 9 - 5 + 2 \dots$ （尽管没有括号）。

二叉搜索树

中序遍历算法的一个重要应用是把有序序列的元素存储在二叉树中，所定义的这种结构称为二叉搜索树。设 S 为一个集合，其独特的元素存在次序关系。例如， S 可能是一组整数。 S 的二叉搜索树是 T ，对于 T 的每一个位置 p ，有：

- 位置 p 存储 S 的一个元素，记作 $e(p)$ 。
- 存储在 p 的左子树的元素（如果有的话）小于 $e(p)$ 。
- 存储在 p 的右子树的元素（如果有的话）大于 $e(p)$ 。

图 8-19 所示为二叉搜索树的例子。上述性能保证二叉搜索树 T 的中序遍历可以按照非递减次序访问元素。

我们可以为 S 使用二叉搜索树 T ，来寻找 S 中的元素 v ，从根开始遍历树 T 下的路径。在 p 遇到的每个内部位置，我们比较搜索值 v 和存储在 p 位置的 $e(p)$ 。如果 $v < e(p)$ ，则继续搜索 p 的左子树。如果 $v = e(p)$ ，则搜索成功。如果 $v > e(p)$ ，则搜索 p 的右子树。最后，如果我们到达一个空的子树，则搜索失败。换句话说，二叉搜索树可以看作一棵二叉决策树（回忆例题 8-6），在内部节点处，要考虑元素是小于、等于还是大于被搜索的元素。在图 8-19 中说明了几个搜索操作的例子。

注意，二叉搜索树 T 的运行时间是和 T 的高度成正比的。回忆命题 8-8， n 个节点二叉树的高度可以小到 $\log(n+1) - 1$ 或者大到 $n-1$ 。因此，当二叉树高度最小时是最有效的。第 11 章将主要介绍搜索树。

8.4.4 用 Python 实现树遍历

在 8.1.2 节中，我们第一次定义树 ADT。树 T 应该支持下列方法：

- $T.positions()$: 树 T 的所有位置生成一个迭代器。
- $iter(T)$: 生成一个迭代器用树 T 存储所有元素。

之前，我们不对这些迭代器报告的结果的顺序做任何假设。在本节中，我们将演示如何让任意一种之前介绍的树遍历算法都能用于产生这些迭代。

一开始，我们注意到树的所有元素很容易产生一个迭代器，前提是依赖一个所有位置的假定迭代器。因此，支持 $iter(T)$ 语法可以正式地通过带有抽象基本树类的特殊方法的 $iter$ 的

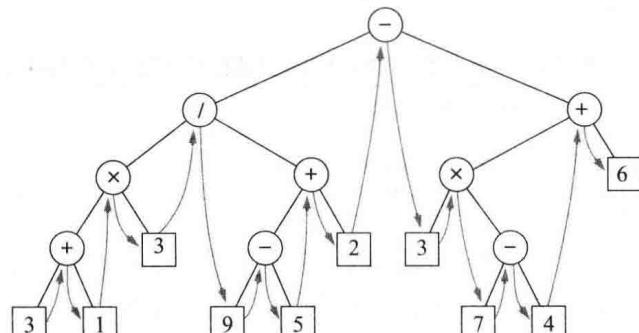


图 8-18 二叉树的中序遍历

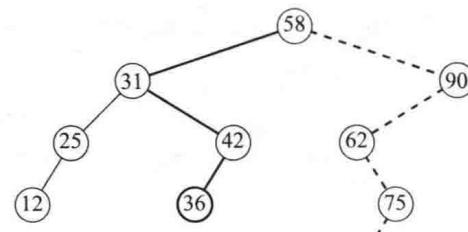


图 8-19 存储整数的二叉搜索树。当搜索（成功地）36 时，实线路径被遍历；当搜索（不成功）70 时，虚线路径被遍历

具体实现给出。我们以 Python 的生成器语法作为迭代产生的机制（见 1.8 节）。代码段 8-16 给出了 Tree.__iter__ 的实现。

代码段 8-16 基于迭代树的位置的所有元素树的实例。这段代码应该包含在 Tree 类的结构体内

```
75 def __iter__(self):
76     """Generate an iteration of the tree's elements."""
77     for p in self.positions():
78         yield p.element() # but yield each element
```

为了实现 positions 方法，我们可以选择树遍历算法。考虑到这些遍历顺序的优点，我们将提供每个策略的独立实现，这些实现可以被类的使用者直接调用。我们可以选择其中一个作为树 ADT 的 positions 方法的默认顺序。

先序遍历

首先考虑先序遍历的算法。我们通过调用树 T 的 $T.\text{preorder}()$ 来给出一个公共的方法，该方法生成一个关于树的所有位置的先序迭代器。然而，像代码段 8-12 中描述的生成先序遍历的递归算法，必须由树的特定位置参数化作为子树的根遍历。对于这种情况，一种标准的解决方案是用所需的递归参数化来定义非公开的应用程序方法，然后由公共方法 preorder 在树根上调用非公开方法。这样设计的实现在代码段 8-17 中给出。

代码段 8-17 支持执行树的先序遍历。这段代码应该包含在 Tree 类的结构体中

```
79 def preorder(self):
80     """Generate a preorder iteration of positions in the tree."""
81     if not self.is_empty():
82         for p in self._subtree_preorder(self.root()): # start recursion
83             yield p
84
85     def _subtree_preorder(self, p):
86         """Generate a preorder iteration of positions in subtree rooted at p."""
87         yield p # visit p before its subtrees
88         for c in self.children(p): # for each child c
89             for other in self._subtree_preorder(c): # do preorder of c's subtree
90                 yield other # yielding each to our caller
```

从形式上讲，preorder 和应用 _subtree_preorder 是生成器。我们把位置给调用者，然后让调用者决定在该位置执行什么操作，而不是在这段代码中执行“访问”行为。

_subtree_preorder 方法是递归的。然而，递归形式略有不同，因为我们依赖于生成器而不是传统的函数。为了生成孩子 c 的子树的所有位置，我们在通过递归调用 self._subtree_preorder(c) 产生的位置上执行循环，并在外环境中重新生成每个位置。注意，如果 p 是叶子，self.children(p) 上的 for 循环是散乱的（这是递归的基本情况）。

我们用相似的技巧从树的根部应用公共的 preorder 方法重新生成所有位置。如果树是空的，什么都不产生。在这点上，我们为 preorder 迭代器提供全面支持，所以类的用户可以编写代码如下：

```
for p in T.preorder():
    # "visit" position p
```

官方树 ADT 要求所有树支持 positions 方法。为了用先序遍历作为默认的迭代顺序，我们在代码段 8-18 中给出了 Tree 类的定义。我们返回整个迭代作为对象，而不是通过先序调用循环返回结果。

代码段 8-18 位置方法依赖于先序遍历产生的结果

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

后序遍历

我们可以应用与先序遍历相似的技巧来实现后序遍历。唯一不同的是后序递归应用，直到递归地产生子树的位置之后，才生成位置 p 。代码段 8-19 给出了一个实例。

代码段 8-19 支持执行树的后序遍历。这段代码应包含在 Tree 类的结构体内

```
94 def postorder(self):
95     """Generate a postorder iteration of positions in the tree."""
96     if not self.is_empty():
97         for p in self._subtree_postorder(self.root()): # start recursion
98             yield p
99
100    def _subtree_postorder(self, p):
101        """Generate a postorder iteration of positions in subtree rooted at p."""
102        for c in self.children(p): # for each child c
103            for other in self._subtree_postorder(c): # do postorder of c's subtree
104                yield other # yielding each to our caller
105        yield p # visit p after its subtrees
```

广度优先遍历

在代码段 8-20 中，我们给出了一个在 Tree 类的上下文中执行广度优先遍历的实现。广度优先遍历算法不是递归的，它借助位置队列来管理递归程序。尽管任何队列 ADT 的实现已经够了，但从 7.1.2 节开始，我们用 LinkedQueue 类来实现。

代码段 8-20 树的广度优先遍历的实现。这段代码应包含在 Tree 类的结构体内

```
106 def breadthfirst(self):
107     """Generate a breadth-first iteration of the positions of the tree."""
108     if not self.is_empty():
109         fringe = LinkedQueue() # known positions not yet yielded
110         fringe.enqueue(self.root()) # starting with the root
111         while not fringe.is_empty():
112             p = fringe.dequeue() # remove from front of the queue
113             yield p # report this position
114             for c in self.children(p):
115                 fringe.enqueue(c) # add children to back of queue
```

中序遍历二叉树

先序、后序和广度优先遍历算法可应用于所有树，所以我们在 Tree 的抽象基类中包含了它们的所有实现。这些方法可以被抽象二叉树类、具体的链二叉树类和其他派生的类继承。

由于中序遍历算法显式地依赖于左和右孩子节点的概念，只适用于二叉树，因此我们在 BinaryTree 类的结构体中包含了该算法的定义。我们使用一个与先序和后序遍历相似的技巧实现中序遍历（见代码段 8-21）。

代码段 8-21 支持执行二叉树的中序遍历，这段代码应包含在 BinaryTree 类中（代码段 8-7 中给出）

```
37 def inorder(self):
38     """Generate an inorder iteration of positions in the tree."""
39     if not self.is_empty():
40         for p in self._subtree_inorder(self.root()):
```

```

41     yield p
42
43     def _subtree_inorder(self, p):
44         """Generate an inorder iteration of positions in subtree rooted at p."""
45         if self.left(p) is not None:      # if left child exists, traverse its subtree
46             for other in self._subtree_inorder(self.left(p)):
47                 yield other
48             yield p                      # visit p between its subtrees
49             if self.right(p) is not None:  # if right child exists, traverse its subtree
50                 for other in self._subtree_inorder(self.right(p)):
51                     yield other

```

对于二叉树的许多应用，中序遍历提供了自然的迭代。我们可以通过重写继承自 Tree 类的 positions 方法来将其作为 BinaryTree 类的默认值（见代码段 8-22）。

代码段 8-22 定义二叉树的位置方法以实现中序遍历节点位置

```

52     # override inherited version to make inorder the default
53     def positions(self):
54         """Generate an iteration of the tree's positions."""
55         return self.inorder()           # make inorder the default

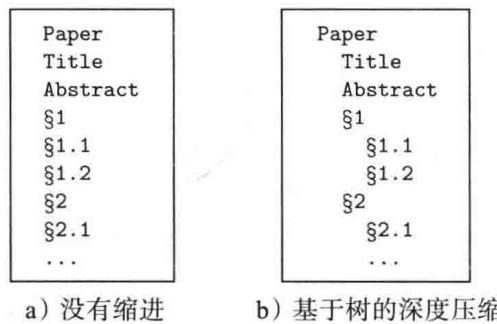
```

8.4.5 树遍历的应用

在本节中，我们将演示几个树遍历的代表应用程序，其中包括一些标准遍历算法的定制。

目录表

我们使用树来表示文档的层次结构，树的先序遍历可以自然地被用于产生一个文档的目录表。例如，图 8-15 中与树相关联的目录表如图 8-20 所示。图 8-20a 按每行一个元素的样式进行了简单表示。图 8-20b 则基于树的深度，通过缩进元素给出了一种更醒目的表示形式。类似的表示可用于展示计算机文件系统目录（见图 8-3）。



a) 没有缩进 b) 基于树的深度压缩

图 8-20 图 8-15 中用树表示的文档的目录表

给定树 T 没有缩进版本的目录表，可以用下面的代码：

```
for p in T.preorder():
    print(p.element())
```

为了生成图 8-20b 的表示样式，我们将每个元素的空间缩进树中元素深度的 2 倍（因此根元素是不被缩进的）。尽管我们可以替换语句打印的循环体 ($2 * T.\text{depth}(p) * ' + \text{str}(p.\text{element}())$)，但这种方法会造成不必要的效率低下。基于 8.4.1 节的分析，虽然产生的先序遍历运行时间为 $O(n)$ ，调用深度会产生一个隐含的成本。从树的每一个位置调用深度都会产生最坏运行时间 $O(n^2)$ ，如 8.1.3 节中 height 算法的分析。

生成一个缩进目录表的首选方法是重新设置一个自顶向下的递归，其中将当前的深度作为额外的参数。代码段 8-23 给出了这个实现。这个实现最坏的运行时间为 $O(n)$ （除去技术上将花费打印增加长度的字符串的时间）。

代码段 8-23 用于打印先序遍历的缩进版本的高效递归。在一个完整的树 T 上，遍历应该从 `preorder_indent(T, T.root(), 0)` 开始

```

1 def preorder_indent(T, p, d):
2     """Print preorder representation of subtree of T rooted at p at depth d."""
3     print(2*d*' ' + str(p.element()))           # use depth for indentation
4     for c in T.children(p):
5         preorder_indent(T, c, d+1)               # child depth is d+1

```

考虑图 8-20 所示的例子，幸运的是编号是嵌入树中的元素。一般来讲，我们可能有兴趣用先序遍历展示树的结构，缩进和树上没有显式呈现的编号。例如，我们可以按照以下样式开始展示图 8-2 所示的树。

```

Electronics R'Us
1 R&D
2 Sales
  2.1 Domestic
  2.2 International
    2.2.1 Canada
    2.2.2 S. America

```

这更具有挑战性，因为数字被用作标签隐含在树的结构中。标签取决于位置的索引，相对于其兄弟姐妹，沿着路径从根到当前位置。为了实现这个任务，我们将路径作为一个额外的参数添加到递归签名。尤其是，我们使用一个 0 索引数字列表，其每个位置沿着向下的路径，而不是根（我们将这些数据转换成索引形式打印）。

在实现层级，我们希望在将一个新参数从递归的一个层级传递到下一个层级时，避免这样低效率的列表。一个标准的解决方案是通过递归共享相同的列表实例。在递归的层级上，一个新的条目在做进一步递归调用之前被暂时添加到列表的末尾。为了“不留下痕迹”，相同的代码块在完成任务之前必须移除多余的条目。代码段 8-24 给出了基于这种方法的实现。

代码段 8-24 用于打印先序遍历的缩进和标记表示

```

1 def preorder_label(T, p, d, path):
2     """Print labeled representation of subtree of T rooted at p at depth d."""
3     label = '.'.join(str(j+1) for j in path)  # displayed labels are one-indexed
4     print(2*d*' ' + label, p.element())
5     path.append(0)                           # path entries are zero-indexed
6     for c in T.children(p):
7         preorder_label(T, c, d+1, path)      # child depth is d+1
8         path[-1] += 1
9     path.pop()

```

树的附加说明表示

如图 8-20a 所示，如果只给定元素的先序序列，那么不可能重建一般的树。要更好地定义树的结构，一些附加的上下文是必需的。用缩进或者编了号的标签提供这样的环境是非常人性化的表现。不管怎样，有些更简明的树的字符串是对计算机友好的。

在本节中，我们探索这样一个表示。树 T 的附加说明的字符串表示 $P(T)$ 以如下方式递归定义。如果 T 由单一的位置 p 组成，则

$$P(T) = \text{str}(p.\text{element}())$$

否则，它将递归定义为

$$P(T) = \text{str}(p.\text{element}()) + ' (' + P(T_1) + ', ' + \dots + ', ' + P(T_k) + ')'$$

其中 p 是 T 的根， T_1, T_2, \dots, T_k 是 p 的孩子的子树根。如果 T 是有序树，则按序给出。我们用“+”来表示字符串连接。例如，图 8-2 所示的树的附加说明表示如下（换行符是修饰）：

```
Electronics R'Us (R&D, Sales (Domestic, International (Canada,
S. America, Overseas (Africa, Europe, Asia, Australia))),
Purchasing, Manufacturing (TV, CD, Tuner))
```

虽然附加说明本质上是一个先序遍历，但是我们不能用之前代码段 8-17 给出的 preorder 实现轻易生成额外的标点符号。左括号必须在循环该位置的孩子之前产生，右括号必须在循环该位置的孩子之后产生。进一步来讲，逗号必须产生。Python 函数 parenthesize 是一个自定义的遍历，用于输出树 T 的附加说明字符串表示，如代码段 8-25 所示。

代码段 8-25 输出树的附加说明字符串表示函数

```
1 def parenthesize(T, p):
2     """Print parenthesized representation of subtree of T rooted at p."""
3     print(p.element(), end=' ')
4     if not T.is_leaf(p):
5         first_time = True
6         for c in T.children(p):
7             sep = ' (' if first_time else ', '
8             print(sep, end=' ')
9             first_time = False
10            parenthesize(T, c)
11            print(')', end=' ')
# include closing parenthesis
```

计算磁盘空间

在例 8-1 中，我们用树作为文件系统结构的模型，用内部位置代表目录，用叶子代表文件。事实上，在第 4 章中介绍递归的使用时，我们专门研究过文件系统（见 4.1.4 节）。虽然当时没有明确地将文件系统模型化为一棵树，但我们给出了计算磁盘使用率的一个算法的实现（见代码段 4-5）。

磁盘空间的递归计算是后序遍历的一个象征，正如我们不能有效地计算总的使用空间直到了解子目录的使用空间之后。不幸的是，代码段 8-19 给出的 postorder 的正式实施并不满足这一目的。访问一个目录的位置时，没有简单的方法来辨别之前的哪个位置代表孩子的目录，也无法辨别有多少递归磁盘空间被分配。

我们想要将孩子向父亲返回信息的机制作为遍历过程的一部分。每层递归为调用者提供一个返回值，来自定义解决磁盘空间问题，如代码段 8-26 所示。

代码段 8-26 树的磁盘空间的递归计算，假设每个树元素的 space() 方法给出在这个位置的本地空间使用情况

```
1 def disk_space(T, p):
2     """Return total disk space for subtree of T rooted at p."""
3     subtotal = p.element().space()
# space used at position p
4     for c in T.children(p):
5         subtotal += disk_space(T, c)
# add child's space to subtotal
6     return subtotal
```

8.4.6 欧拉图和模板方法模式 *

8.4.5 节描述的各种应用程序展示了树递归遍历的强大功能。不幸的是，它们也表明 Tree 类的 preorder 和 postorder 方法的具体实现，或者 BinaryTree 类的 inorder 方法的实现，一般不足以采集我们期望的计算范围。在有些情况下，我们需要更多的混合方法，初始工作执行之前重复执行子树，额外的工作执行在递归执行之后，对于二叉树，工作执行两种可能的递归。进一步来讲，在某些情况下，知道位置的深度，或者从根到该位置的完整的路径，或者返回从递归的一个层级到另一个层级的信息，这些是很重要的。对于前面的每个应用程序，我们可以开发一个正确适用递归思想的实现，但是面向对象编程（见 2.1.1 节）原则包括适应性和可重用性。

在本节中，我们开发了一个更通用的框架，即基于概念实现树的遍历——欧拉遍历。一般树 T 的欧拉遍历可以非正式地定义为沿着 T “走”，从根开始“走”向最后一个孩子，我们保持在左边，像“墙”一样查看 T 的边缘，如图 8-21 所示。

遍历的复杂度为 $O(n)$ ，因为恰好两次沿着树的 $n - 1$ 条边进行——一次沿着边缘向下走，一次沿着边缘向上走。为了统一先序和后序遍历的概念，对于每个位置 p ，我们可以考虑两个值得注意的“访问”：

- 当到达第一个位置，即当遍历立刻通过可视化节点的左边时，“先访问”出现。
- 当从该位置向上遍历，即当遍历通过可视化节点的右边时，“后访问”发生。

欧拉遍历的过程很容易被看成递归，在给定位置的“先访问”和“后访问”之间将是每个子树的递归遍历。以图 8-21 为例，整个遍历的连续部分本身就是节点带元素“/”的子树的欧拉遍历。遍历包含两个连续的子遍历，一个遍历左子树，一个遍历右子树。对于根在 p 位置处的子树的欧拉遍历，其伪代码如代码段 8-27 所示。

代码段 8-27 根在 p 位置处的子树的欧拉遍历的算法实现 .

```
Algorithm eulertour( $T, p$ ):
    perform the “pre visit” action for position  $p$ 
    for each child  $c$  in  $T.\text{children}(p)$  do
        eulertour( $T, c$ ) {recursively tour the subtree rooted at c}
    perform the “post visit” action for position  $p$ 
```

模板方法模式

为了提供一个可重用的和适应性强的框架，我们借用了一种有趣的面向对象软件设计模式——模板方法模式。模板方法模式通过精简某些步骤描述了一个通用的计算机制。在指定步骤的过程中，为了允许自定义，基本算法调用称为钩子（hook）的辅助函数。

在欧拉遍历的上下文中，我们定义了两个单独的钩子。在子树被访问之前，“先序访问”钩子被调用；在子树完成遍历之后，“后续访问”钩子被调用。我们的实现将采用 EulerTour 类管理进程，并简单定义什么也不做的钩子。遍历可以通过定义 EulerTour 的子类和重载一个或两个用以提供特殊性能的钩子来进行个性化设置。

Python 实现

代码段 8-28 提供了 EulerTour 类的实现，主要的递归过程被定义为非公开的 `_tour` 方法。

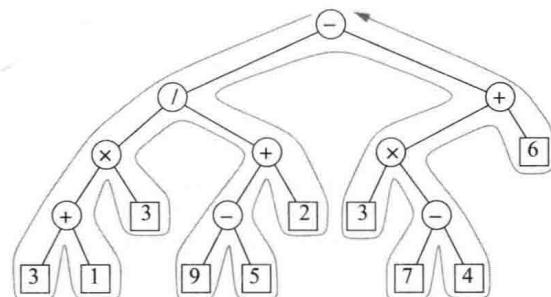


图 8-21 树的欧拉遍历

通过发送引用一个特定的树的构造函数创建遍历实例，然后通过调用公共执行方法去遍历返回一个计算的最终结果。

代码段 8-28 EulerTour 基类提供了一个框架，用于执行树的欧拉遍历

```

1  class EulerTour:
2      """Abstract base class for performing Euler tour of a tree.
3
4      _hook_previsit and _hook_postvisit may be overridden by subclasses.
5      """
6
7      def __init__(self, tree):
8          """Prepare an Euler tour template for given tree."""
9          self._tree = tree
10
11     def tree(self):
12         """Return reference to the tree being traversed."""
13         return self._tree
14
15     def execute(self):
16         """Perform the tour and return any result from post visit of root."""
17         if len(self._tree) > 0:
18             return self._tour(self._tree.root(), 0, [ ])           # start the recursion
19
20     def _tour(self, p, d, path):
21         """Perform tour of subtree rooted at Position p.
22
23         p      Position of current node being visited
24         d      depth of p in the tree
25         path   list of indices of children on path from root to p
26
27         self._hook_previsit(p, d, path)                      # "pre visit" p
28         results = []
29         path.append(0)           # add new index to end of path before recursion
30         for c in self._tree.children(p):
31             results.append(self._tour(c, d+1, path))    # recur on child's subtree
32             path[-1] += 1                         # increment index
33             path.pop()                          # remove extraneous index from end of path
34         answer = self._hook_postvisit(p, d, path, results)    # "post visit" p
35         return answer
36
37     def _hook_previsit(self, p, d, path):                  # can be overridden
38         pass
39
40     def _hook_postvisit(self, p, d, path, results):        # can be overridden
41         pass

```

8.4.5 节的简单应用基于定制遍历的经验，我们在代码段 8-24 中介绍了支持主要的 EulerTour 遍历以维护递归遍历的深度和路径。我们还为递归层级提供了一个机制，用于在进行后续处理时返回值。在形式上，框架依赖于专业化的两个钩子：

- **method _hook_previsit(p, d, path)**

每个位置调用这个函数一次——在子树遍历之前立即调用（如果有的话）。参数位置 p 是树上的位置，d 是位置的深度，path 是索引的列表，使用代码段 8-24 中所描述的约定。这个函数没有返回值。

- **method _hook_postvisit(p, d, path, results)**

这个函数在每个位置被调用一次——在其子树被遍历后立即调用，前三个参数使用与 _hook_previsit 相同的约定。最后一个参数是以 p 的子树后序遍历的返回值作为

列表对象。任何通过此调用的返回值可以被其父节点 p 所利用。

对于更复杂的任务，EulerTour 的子类能够以实例变量可以在带有钩子的本体类中被访问的形式选择初始化和维护额外的状态。

使用欧拉遍历框架

为了展示欧拉遍历的灵活性，我们重新审视 8.4.5 节中的示例应用程序。举一个简单的例子，一个缩进的先序遍历（类似于代码段 8-23），可以由代码段 8-29 中给出的简单子类生成。

代码段 8-29 EulerTour 的子类生成树元素的缩进先序列表

```

1 class PreorderPrintIndentedTour(EulerTour):
2     def _hook_previsit(self, p, d, path):
3         print(2*d*' ' + str(p.element()))

```

对于给定的树 T，通过创建子类的实例来开始遍历并调用 execute 方法。代码如下：

```

tour = PreorderPrintIndentedTour(T)
tour.execute()

```

缩进标记版本类似于代码段 8-24，可能通过 EulerTour 的新子类生成，如代码段 8-30 所示。

代码段 8-30 EulerTour 的子类生成树元素的标记和缩进先序列表

```

1 class PreorderPrintIndentedLabeledTour(EulerTour):
2     def _hook_previsit(self, p, d, path):
3         label = '.'.join(str(j+1) for j in path)    # labels are one-indexed
4         print(2*d*' ' + label, p.element())

```

为了生成附加说明的字符串表示，最初实现如代码段 8-25 所示，我们通过重写先序遍历和后序遍历的钩子定义了一个子类。新的实现如代码段 8-31 所示。

代码段 8-31 EulerTour 的子类，用于打印树的附加说明字符串的表示

```

1 class ParenthesizeTour(EulerTour):
2     def _hook_previsit(self, p, d, path):
3         if path and path[-1] > 0:                # p follows a sibling
4             print(', ', end='')                  # so preface with comma
5             print(p.element(), end='')          # then print element
6         if not self.tree().is_leaf(p):          # if p has children
7             print('(', end='')                 # print opening parenthesis
8
9     def _hook_postvisit(self, p, d, path, results):
10        if not self.tree().is_leaf(p):          # if p has children
11            print(')', end='')                # print closing parenthesis

```

注意，在这个实现中，我们在树的实例中调用一个从内部钩子遍历的方法。欧拉遍历类的公共 tree() 方法作为树的访问器。

最后，计算磁盘空间的任务（如代码段 8-26 所示）可以用代码段 8-32 所示的 EulerTour 子类很容易地实现。根的后序遍历通过调用 execute() 返回结果。

代码段 8-32 欧拉遍历子类计算树的磁盘空间

```

1 class DiskSpaceTour(EulerTour):
2     def _hook_postvisit(self, p, d, path, results):
3         # we simply add space associated with p to that of its subtrees
4         return p.element().space() + sum(results)

```

二叉树的欧拉遍历

在 8.4.6 节中，我们介绍了一般图的欧拉遍历的概念，使用模板方法模式设计 EulerTour 类。类提供的 _hook_previsit 和 _hook_postvisit 方法可以被重载来定制遍历。代码段 8-33 给出了一个 BinaryEulerTour 特性，包括额外的 _hook_invisit 方法——被每个位置调用一次，在遍历左子树之后、右子树之前调用。

BinaryEulerTour 的实现代替了原来的 _tour，仅限于一个节点至多有两个孩子的情况。如果一个节点只有一个孩子，遍历将区分是左孩子还是右孩子。访问发生在一个左孩子访问之后，且在一个右孩子访问之前。在一片叶子的情况下，会连续调用三个钩子。

代码段 8-33 BinaryEulerTour 基类为二叉树提供专门的遍历。最初的 EulerTour 基类在代码段 8-28 中给出

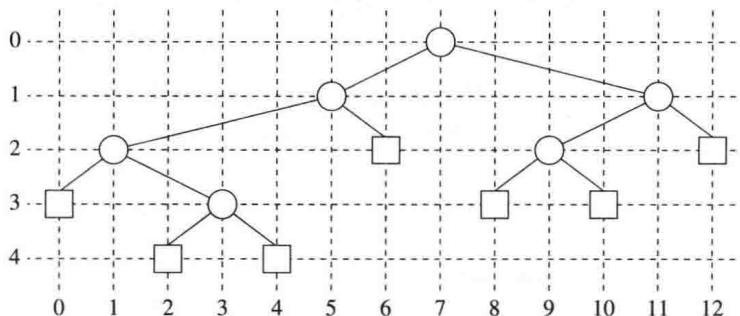
```

1 class BinaryEulerTour(EulerTour):
2     """Abstract base class for performing Euler tour of a binary tree.
3
4     This version includes an additional _hook_invisit that is called after the tour
5     of the left subtree (if any), yet before the tour of the right subtree (if any).
6
7     Note: Right child is always assigned index 1 in path, even if no left sibling.
8     """
9
10    def __init__(self, p, d, path):
11        self._tour(p, d, path)           # will update with results of recursions
12        self._hook_previsit(p, d, path)   # "pre visit" for p
13        if self._tree.left(p) is not None: # consider left child
14            path.append(0)
15            self._tour(self._tree.left(p), d+1, path)
16            path.pop()
17            self._hook_invisit(p, d, path)      # "in visit" for p
18            if self._tree.right(p) is not None: # consider right child
19                path.append(1)
20                self._tour(self._tree.right(p), d+1, path)
21                path.pop()
22        self._hook_postvisit(p, d, path)      # "post visit" p
23
24    def _hook_invisit(self, p, d, path): pass      # can be overridden

```

为了演示 BinaryEulerTour 的框架，我们开发了一个用于计算二叉树的图形布局的子类，如图 8-22 所示。几何图形由一个算法确定，该算法用以下两条规则为二叉树 T 的每个位置 p 指定 x 坐标和 y 坐标。

- $x(p)$ 是在 p 之前 T 的中遍历中访问的位置数量。
- $y(p)$ 是 T 中 p 的深度。



在这个应用中，我们采用计算机图形学中的一个公认约定，即 x 坐标从左到右增加， y 坐标从上到下增加，所以原点在计算机屏幕的左上角。

代码段 8-34 给出了一个 BinaryLayout 子类的实现，用于前面的算法，即实现为存储在二叉树每个位置的元素分配 (x, y) 坐标。我们以 `_count` 实例变量（表示我们已执行“in visits”的数量）的形式引入额外的状态，从而调整 BinaryEulerTour 框架。每个位置的 x 坐标根据计数器设置。

代码段 8-34 用于计算坐标绘出二叉树图形布局的 BinaryLayout 类，假设原来树的元素类型支持 `setX` 和 `setY` 方法

```

1 class BinaryLayout(BinaryEulerTour):
2     """Class for computing (x,y) coordinates for each node of a binary tree."""
3     def __init__(self, tree):
4         super().__init__(tree)           # must call the parent constructor
5         self._count = 0                # initialize count of processed nodes
6
7     def _hook_invisit(self, p, d, path):
8         p.element().setX(self._count)  # x-coordinate serialized by count
9         p.element().setY(d)          # y-coordinate is depth
10        self._count += 1            # advance count of processed nodes

```

8.5 案例研究：表达式树

在例 8-7 中，我们介绍了使用二叉树来表示算数表达式的结构。在本节中，我们定义一个新类 ExpressionTree 为构建树提供支持，显示和评估树呈现的算术表达式。ExpressionTree 类被定义为 LinkedBinaryTree 类的子类。我们用非公开调整器来构建这样的树。每个内部节点必须存储一个用于定义二进制操作（如 +）的字符串，每片叶子必须存储一个数值（或者一个字符串代表一个数值）。

最终目的是将任意复杂度的表达式树建立为复合运算表达式，如 $((3 + 1) \times 4) / ((9 - 5) + 2)$ 。然而，它仅支持两种基本形式来初始化表达式树类。

- ExpressionTree(value): 创建一棵在根处存储给定值的树。
- ExpressionTree(op, E_1, E_2): 创建一棵在根处存储字符串 op (如 +) 的树，ExpressionTree 的实例 E_1 和 E_2 分别作为根的左子树和右子树。

ExpressionTree 的构造函数在代码段 8-35 中给出，该类正式继承自 LinkedBinaryTree，所以它访问 8.3.1 节中定义的非公开更新方法。我们使用 `_add_root` 方法来创建树的初始根，用以将令牌作为第一个参数存储，然后执行运行时参数来检查调用者是调用构造函数的单个参数版本（在这种情况下，我们已经做完了）还是 3 个参数形式。在这种情况下，我们结合树的结构使用继承的 `_attach` 方法作为根的子树。

组成一个括号字符串表示

现有表达式树实例的字符串表示，例如 $((3 + 1) \times 4) / ((9 - 5) + 2)$ ，可以通过中序遍历树的方法来产生，但左括号和右括号分别用先序和后序步骤插入。ExpressionTree 类的上下文中，我们支持一个特殊的 `__str__` 方法（见 2.3.2 节）返回一个合适的字符串。因为它是更高效地先将一系列独立的字符串连接在一起（见 5.4.2 节中“组合字符串”的讨论），`str` 的实现依赖于一个非公开、递归的方法 `_parenthesize_recur`，该方法用于在一个列表中添加一系列字符串。这些方法被包含在代码段 8-35 中。

代码段 8-35 ExpressionTree 类的开始部分

```

1 class ExpressionTree(LinkedBinaryTree):
2     """An arithmetic expression tree."""
3
4     def __init__(self, token, left=None, right=None):
5         """Create an expression tree.
6
7         In a single parameter form, token should be a leaf value (e.g., '42'),
8         and the expression tree will have that value at an isolated node.
9
10    In a three-parameter version, token should be an operator,
11    and left and right should be existing ExpressionTree instances
12    that become the operands for the binary operator.
13    """
14    super().__init__()                      # LinkedBinaryTree initialization
15    if not isinstance(token, str):
16        raise TypeError('Token must be a string')
17    self._add_root(token)                   # use inherited, nonpublic method
18    if left is not None:                   # presumably three-parameter form
19        if token not in '+-*x/':
20            raise ValueError('token must be valid operator')
21        self._attach(self.root(), left, right) # use inherited, nonpublic method
22
23    def __str__(self):
24        """Return string representation of the expression."""
25        pieces = []                         # sequence of piecewise strings to compose
26        self._parenthesize_recur(self.root(), pieces)
27        return ''.join(pieces)
28
29    def _parenthesize_recur(self, p, result):
30        """Append piecewise representation of p's subtree to resulting list."""
31        if self.is_leaf(p):
32            result.append(str(p.element()))      # leaf value as a string
33        else:
34            result.append('(')                # opening parenthesis
35            self._parenthesize_recur(self.left(p), result) # left subtree
36            result.append(p.element())        # operator
37            self._parenthesize_recur(self.right(p), result) # right subtree
38            result.append(')')                # closing parenthesis

```

表达式树的评估

表达式树的数值评估可以用先序遍历的简单应用完成。如果知道两个子树内部节点的位置，我们可以计算指定位置的计算结果。代码段 8-36 给出了根在 p 位置处子树的评估值的递归伪代码。

代码段 8-36 根在 p 位置处的子树的评估算法 evaluate_recur

```

Algorithm evaluate_recur(p):
    if p is a leaf then
        return the value stored at p
    else
        let o be the operator stored at p
        x = evaluate_recur(left(p))
        y = evaluate_recur(right(p))
        return x o y

```

为了用 Python 的 ExpressionTree 类实现这个算法，我们提供了一个公共的 evaluate 方法，它用 $T.evaluate()$ 调用实例 T 。代码段 8-37 给出了这样一个实现——用一个非公开评估方法 _evaluate_recur 计算指定子树的值。

代码段 8-37 评估 ExpressTree 的实例

```

39 def evaluate(self):
40     """ Return the numeric result of the expression."""
41     return self._evaluate_recur(self.root())
42
43 def _evaluate_recur(self, p):
44     """ Return the numeric result of subtree rooted at p."""
45     if self.is_leaf(p):
46         return float(p.element()) # we assume element is numeric
47     else:
48         op = p.element()
49         left_val = self._evaluate_recur(self.left(p))
50         right_val = self._evaluate_recur(self.right(p))
51         if op == '+': return left_val + right_val
52         elif op == '-': return left_val - right_val
53         elif op == '/': return left_val / right_val
54         else: return left_val * right_val # treat 'x' or '*' as multiplication

```

创建一棵表达式树

代码段 8-35 中 ExpressionTree 的构造函数，提供了结合现有树构建更大表达式树的基本功能。然而，对于给定的字符串，如 $((3+1) \times 4)/((9-5)+2)$ ，如何构建一棵表示该表达式的树，这一问题尚未解决。

为了将这个过程自动化，我们使用一个自上而下的构造算法，假设一个字符串可以先被标记化，这样多位数字就可以自动处理（见练习 R-8.30），从而这个表达式就完全被括起来了。算法使用栈 S 扫描输入表达式 E 来查找值、操作符和右括号（左括号被忽略）。

- 当看到一个操作符时，我们将字符串推入栈。
- 当看到一个文本值 v 时，我们创建一个单个节点表达式树 T 存储 v ，并将 T 推入栈中。
- 当看到一个右括号 “)” 时，我们从栈 S 的最顶端抛出三个元素，它代表子表达式 $(E_1 \circ E_2)$ 。我们构造树 T ，使用根的子树存储 E_1 和 E_2 ，并把结果树 T 放回栈中。

我们重复这个过程直到表达式 E 被处理完，每一次栈顶元素都是表达式树 E 。总共的运行时间为 $O(n)$ 。

算法的实现在代码段 8-38 中以独立函数 build_expression_tree 的形式给出，该函数返回一个适当的 ExpressionTree 实例，假设输入已经被标记化。

代码段 8-38 build_expression_tree 的实现，该函数用表示一个算术表达式的一系列
标记生成 ExpressionTree

```

1 def build_expression_tree(tokens):
2     """ Returns an ExpressionTree based upon by a tokenized expression."""
3     S = [] # we use Python list as stack
4     for t in tokens:
5         if t in '+-*/':
6             S.append(t) # push the operator symbol
7         elif t not in '()' :
8             S.append(ExpressionTree(t)) # push trivial tree storing value
9         elif t == ')':
10            # compose a new tree from three constituent parts
11            right = S.pop() # right subtree as per LIFO
12            op = S.pop() # operator symbol
13            left = S.pop() # left subtree
14            S.append(ExpressionTree(op, left, right)) # repush tree
15    # we ignore a left parenthesis
16    return S.pop()

```

8.6 练习

请访问 www.wiley.com/college/goodrich 以获得练习帮助。

巩固

R-8.1 下列问题基于图 8-3 中的树。

- 哪个节点是根节点?
- 哪些是内部节点?
- 节点 cs016 有多少子孙节点?
- cs016 有多少祖先节点?
- homeworks 有哪些兄弟节点?
- 哪些节点在以 projects 为根节点的子树中?
- papers 节点的深度是多少?
- 树的高度是多少?

R-8.2 对于树的 depth 算法, 给出一棵树, 实现最坏情况运行时。

R-8.3 给出命题 8-4 的证明。

R-8.4 当在一棵树的位置 p 处而不是根节点处调用 `T.height2(p)` 方法时的运行时间是多少? (详见代码段 8-5)

R-8.5 给出一个仅依靠二叉树操作的算法, 该算法能够统计二叉树中作为左孩子的叶子节点的个数。

R-8.6 假设 T 是一棵有 n 个节点的二叉树, 该二叉树可能不规则。请说明如何通过一棵有 $O(n)$ 节点数的完全二叉树 T' 来表示 T 。

R-8.7 在一个拥有 n 个节点的不完全二叉树中, 内部节点和外部节点的最多和最少的个数分别是多少?

R-8.8 回答如下的问题以给出命题 8-8 的证明:

- 对于一棵高度为 h 的完全二叉树, 外部节点的最少个数是多少。证明你的答案。
- 对于一棵高度为 h 的完全二叉树, 外部节点的最多个数是多少。证明你的答案。
- 假设 T 是一棵有 n 个节点并且高度为 h 的完全二叉树, 请证明:

$$\log(n + 1) - 1 \leq h \leq (n - 1)/2$$

- d) 当 n 和 h 取什么值时, 上边的不等式两边取等号?

R-8.9 请给出命题 8-9 的证明。

R-8.10 在 `BinaryTree` 类中给出 `num_children` 方法的实现过程。

R-8.11 找出与图 8-8 所示二叉树中每个子树相关的值的算术表达式。

R-8.12 画出一棵算术表达式的树, 其含有 4 个外部节点, 分别存储数字 1、5、6 和 7 (每个外部节点存储一个数字, 但不一定按照这样的顺序) 并且有 4 个内部节点, 分别存储来自操作集 $\{+, -, *, /\}$ 中的字符, 使得通过计算得到根节点的值为 21。这些操作符可能在局部被使用, 并且一个操作符可能被使用不止一次。

R-8.13 画出如下算术表达式的二叉树:

$$(((5 + 2)*(2 - 1))/((2 + 9) + ((7 - 2) - 1)))*8$$

R-8.14 根据表 8-2, 通过给出每一个方法实现的描述和执行时间的分析, 总结用链式结构来表达树的运行时间。

R-8.15 8.3.1 节中的类 `LinkedBinaryTree` 仅提供了一个非公共的更新操作方法。请实现一个能够为每个继承非公共的更新操作方法提供公共函数的 `MutableLinkedBinaryTree` 子类。

R-8.16 假设 T 是一棵有 n 个节点的二叉树, 并且 $f()$ 是树某个位置同一水平中节点个数计数的函数 (参见 8.3.2 节)。

- a) 试证明对于树 T 中任何一个位置 p , $f(p) \leq 2^n - 2$ 。
 b) 在一棵拥有 7 个节点的树中, 试着给出在哪些位置下上面的不等式两边能够取等号。

- R-8.17 试说明如何通过使用欧拉遍历来计算处于树 T 中每个位置 p 的 $f(p)$ 。
- R-8.18 假设 T 是一棵通过数组 A 表示的拥有 n 个节点的二叉树, $f()$ 是计算树 T 中某个位置同一级节点的个数的函数。试给出 `root`、`parent`、`left`、`right`、`is_leaf` 和 `is_root` 方法的伪代码。
- R-8.19 我们在 8.3.2 节给出的计算同一级节点个数的函数 $f(p)$ 在根节点的位置时的结果是 0。一些作者更喜欢使用函数 $g(p)$, 当 p 是根节点时, 其结果为 1, 因为其简化了寻找相邻位置的方法。请使用函数 $g(p)$ 重做练习 R-8.18。
- R-8.20 画一棵二叉树 T , 使其同时满足如下条件:
- 树 T 的每个内部节点存储一个字符。
 - 对树 T 先序遍历产生 EXAMFUN。
 - 对树 T 中序遍历产生 MAFXUEN。
- R-8.21 对图 8-8 中的树进行先序遍历时, 访问树中节点的顺序是怎样的?
- R-8.22 对图 8-8 中的树进行后序遍历时, 访问树中节点的顺序是怎样的?
- R-8.23 假设 T 是一棵有不止一个节点的有序树, 试问是否可能对其中序遍历和后续遍历都以相同的顺序访问其中的节点? 如果你认为可能, 请给出一个例子; 反之, 请解释为什么不可能。类似地, 是否存在可能使得对树进行中序遍历和后序遍历时以相反的顺序访问树中的节点? 如果你认为可能, 给出具体的例子; 反之, 请解释为什么不会发生。
- R-8.24 当 T 是一棵有不止一个节点的完全二叉树时, 试回答 R-8.23 中的问题。
- R-8.25 考虑图 8-17 中给出的对树进行广度优先遍历的例子, 用图中标注的数字, 描述在每一次执行代码段 8-14 中的循环之前队列当中的内容。一开始, 在第一次执行循环之前队列当中的内容为 {1}, 第二次执行前的内容是 {2, 3, 4}。
- R-8.26 类 `collection.deque` 支持一次性将一个集合的元素添加到队列尾部的 `extend` 方法。重新实现 `Tree` 类的广度优先遍历的方法, 使其充分利用这个特性。
- R-8.27 如图 8-8 给出的树, 试写出代码段 8-25 中函数 `parenthesize(T, T.root())` 的输出。
- R-8.28 对于一棵有 n 个节点的树, 代码段 8-25 中的函数 `parenthesize(T, T.root())` 的执行时间是多少?
- R-8.29 请用伪代码描述一个算法, 该算法用于计算在先序遍历中给出一个计算二叉树中每个节点的子孙节点个数的算法。该算法需要基于欧拉遍历。
- R-8.30 `ExpressionTree` 类 `build_expression_tree` 方法中的输入需要一个字符串标记。举个简单的例子, $'((3 + 1)*4)/((9 - 5) + 2))'$, 其中每个字符是它本身的标记, 因此这个字符串本身足以作为 `build_expression_tree` 的输入。例如, 字符串 '(35+14)' 需要放入链表 ['(', '35', '+', '14', ')'], 使得可以忽略空格, 并能识别多维字符作为令牌。写一个可用的方法 `tokenize()`, 以返回这样一个令牌。

创新

- C-8.31 将一棵树 T 所有内部节点的深度之和定义为内部路径长度 $I(T)$ 。类似地, 将一棵树 T 所有外部节点的深度之和定义为外部路径长度 $E(T)$ 。试证明一棵有 n 个节点的完全二叉树满足公式 $E(T) = I(T) + n - 1$ 。
- C-8.32 假设 T 是一棵有 n 个节点的二叉树, 并假设 D 是树 T 所有外部节点深度的总和。是否存在树 T 有最少的外部节点使得 D 的运行时间为 $O(n)$, 并且是否存在树 T 有最多的外部节点使得 D 为 $O(n \log n)$?
- C-8.33 假设 T 是一棵有 n 个节点的二叉树, 并假设 D 是树 T 所有外部节点的深度之和。试给出一棵

树，使得代码段 8-4 中的 `_height1` 方法运行在最坏情况下。

C-8.34 对于一棵树 T ，假设 n_i 表示内部节点的个数，并假设 n_E 表示外部节点的个数。试证明如果每个内部节点有 3 个孩子节点，那么 $n_E=2n_i+1$ 。

C-8.35 如果两个有序树 T' 和 T'' 中满足如下两点的任何一个，则称它们是同构的：

- T' 和 T'' 是空树。
- T' 和 T'' 的根节点有相同数量的 k 个子树 ($k \geq 0$) 并且 T' 和 T'' 的第 i 个子树也是同构的，其中 $i=1, 2, \dots, k$ 。

试设计一个测试两棵给定的有序树是否是同构的算法，并说明该算法的运行时间。

C-8.36 试证明有 n 个内部节点的 2^n 个不完全二叉树中没有一对是同构的（参见 C-8.35）。

C-8.37 如果排除同构树，那么有多少棵完全二叉树存在 4 个叶子节点？

C-8.38 给 `LinkedBinaryTree` 增加一个 `_delete_subtree(p)` 方法。该方法能够移除以 p 节点为根节点的整个子树，并维持整棵树所有节点的个数的不变性。这个方法的运行时间是多少？

C-8.39 在 `LinkedBinaryTree` 中增加一个 `_swap(p, q)` 方法，该方法能够交换节点 p 和节点 q ，反之亦然。需要考虑节点是邻边节点的情况。

C-8.40 如果充分利用哨兵节点（该哨兵节点指的是树实例的 `_sentinel` 数量），我们可以简化 `LinkedBinaryTree` 的实现过程。哨兵是树的根节点的父节点，而根节点是哨兵的左孩子节点。此外，哨兵将取代 `None` 来表示节点中 `_left` 或 `_right` 的数量，而不需要这样的孩子节点。请给出更新方法 `_delete` 和 `_attach` 的新的实现。

C-8.41 请描述如何通过使用 `_attach` 方法来复制一个 `LinkedBinaryTree` 的完全二叉树实例。

C-8.42 请描述如何通过使用 `_add_left` 和 `_add_right` 方法来复制一个 `LinkedBinaryTree` 的完全二叉树实例。

C-8.43 对于一棵有序树，我们可以定义一种二叉树表示 T' （见图 8-23）：

- 对于树 T 中的每个位置 p ，树 T' 中都有一个与其相关的位置 p' 。
- 如果 p 是树 T 的叶子节点，那么 T' 中的 p' 没有任何孩子节点；否则， p' 的左孩子节点是 q' ，其中 q 是树 T 中 p 的第一个孩子节点。
- 如果树 T 中 p 有一个相邻节点 q ，那么 q' 是树 T 中 p' 的右孩子节点；否则， p' 没有右孩子节点。

假设树 T' 是常见有序树 T 的一种表示，请回答如下的问题：

- T' 的先序遍历和 T 的先序遍历是否相同？
- T' 的先序遍历和 T 的中序遍历是否相同？
- T' 的中序遍历是否是树 T 标准遍历中的一种？如果是，是下面的哪一种（见图 8-23）？

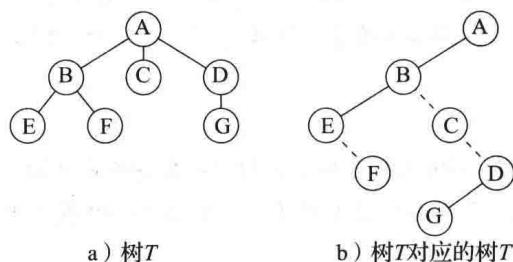


图 8-23 一棵二叉树的表示方法

C-8.44 对于树 T 中的每个位置 p ，给出一个计算并打印 p 后面子树元素的高效算法。

- C-8.45 给出一个计算树 T 中所有节点深度的运行时间为 $O(n)$ 的算法，其中 n 是树中节点的个数。
- C-8.46 树 T 的路径长度是其中所有节点的深度之和。请给出一个计算树路径长度的线性时间算法。
- C-8.47 一棵完全二叉树的内部节点 p 的左子树与右子树的高度（深度）差即为该节点的平衡因子。试描述如何利用 8.4.6 节的欧拉遍历来打印一棵平衡二叉树的所有内部节点的平衡因子。
- C-8.48 给定一棵完全二叉树 T ，定义 T' 是树 T 的镜像，其中 T 的每个节点 v 同样也是 T' 的节点，但是树 T 中节点 v 的左孩子是树 T' 中 v' 节点的右孩子。树 T 中节点 v 的右孩子是树 T' 的节点 v' 的左孩子。试说明对一棵完全二叉树 T 的先序遍历和树 T' 镜像的后序遍历完全一样，只是顺序相反。
- C-8.49 假设在遍历时给位置为 p 的元素定义一个排名，第一个被遍历到的元素排名第一，第二个被遍历到的元素排名第二，以此类推。对于每一个处于位置 p 的元素，我们假设 $\text{pre}(p)$ 是对树 T 中处于位置 p 的元素在先序遍历时的排名， $\text{post}(p)$ 是对树 T 中处于位置 p 的元素在后续遍历时的排名。假设 $\text{depth}(p)$ 是处于位置 p 的深度， $\text{desc}(p)$ 是处于位置 p 的后代的数量（包括 p 本身）。对于树 T 中的每一个节点，请给出 $\text{post}(p)$ 、 $\text{desc}(p)$ 、 $\text{depth}(p)$ 和 $\text{pre}(p)$ 的公式。
- C-8.50 对一棵给定的二叉树设计支持如下操作的算法：
- `preorder_next(p)`：对树 T 进行先序遍历时，返回访问 p 后下一个将要访问的节点的位置（如果 p 是最后一个节点，则返回空）。
 - `inorder_next(p)`：对树 T 进行中序遍历时，返回访问 p 后下一个将要访问的节点的位置（如果 p 是最后一个节点，则返回空）。
 - `postorder_next(p)`：对树 T 进行后序遍历时，返回访问 p 后下一个将要访问的节点的位置（如果 p 是最后一个节点，则返回空）。
- 在最坏情况下，这些算法的执行时间是多少？
- C-8.51 为了实现 `LinkedBinaryTree` 类的 `preorder` 方法，我们需要利用 Python 的生成器句法和 `yield` 状态。请给出 `preorder` 的另一种实现，返回嵌套迭代器类的显式实例。（参见 2.3.4 节关于迭代器的讨论。）
- C-8.52 算法 `preorder_draw` 通过指定每个位置 p 的横纵坐标来生成一棵二叉树，其中 $x(p)$ 是先序遍历中 p 前的节点的数量。 $y(p)$ 是树中 p 的深度。
- 试说明通过算法 `preorder_draw` 产生的二叉树没有两条相交的边。
 - 通过算法 `preorder_draw` 重新生成图 8-22 中的树。
- C-8.53 通过与算法 `preorder_draw` 相似的 `postorder_draw` 算法重做先前的问题，其中 $x(p)$ 是后续遍历中 p 前的节点的数量。
- C-8.54 使用与中序遍历生成一棵二叉树相同的方法设计一个生成普通树的算法。
- C-8.55 练习 P-4.27 描述了 `os` 模块中的 `walk` 函数。该函数对文件系统的树形结构执行遍历。查看该函数的文档，特别是其中一个可选的称为 `topdown` 的布尔参数的使用。试描述其与本章讨论的树遍历算法有何关系。
- C-8.56 树 T 的缩进表示是树 T 附带说明（详见代码段 8-25）表示的另一种形式，其使用如图 8-24 所示的缩进表示。请给出一个能打印出这种树的算法。
- C-8.57 假设树 T 是一棵有 n 个节点的二叉树，定义罗马位置 p 为这样一个位置：该位置的左子树的数量与其右子树的数量最多不少于 5。给出一个寻找树 T 中每个位置的线性时间算法，在树 T 中， p 不是罗马位置，但所有 p 的后代节点是罗马位置。
- C-8.58 假设 T 是一棵有 n 个节点的树，定义这个最低的共同祖先（LCA）是在树 T 中两个最低位置

之间都有 p 和 q 作为祖先的节点（其中我们允许一个位置自己作为祖先）。给定两个位置 p 和 q ，给出一个寻找 p 和 q 的 LCA 的高效算法。该算法的运行时间是多少？

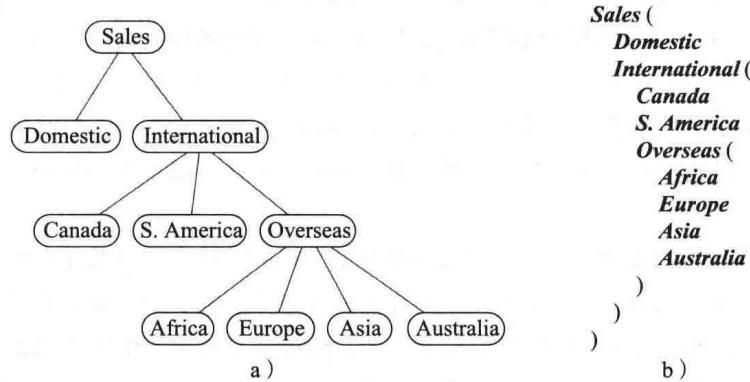


图 8-24 a) 树 T ; b) 树 T 的缩进表示

- C-8.59 假设 T 是一个有 n 个节点的二叉树，并且对于 T 中的任何一个位置 p ，假设 d_p 代表树 T 中 p 的深度，那么两个节点 p 和 q 之间的长度用 $d_p + d_q - 2d_a$ 表示。其中 a 是 p 和 q 的 LCA。树 T 的直径是树中两个节点的最远距离。给出一个寻找树 T 直径的高效算法。其运行时间是多少？
- C-8.60 假设一棵二叉树 T 中的每个节点附有一个值 $f(p)$ ，试设计一个快速决定 LCA 的 $f(a)$ 的算法，其中 $f(q)$ 和 $f(p)$ 给定，你不需要寻找到位置 a ，只要得出 $f(a)$ 。
- C-8.61 请给出一个 ExpressionTree 类中 `_expression_tree` 方法的可选方法。该方法依赖构建树欧拉环路的递归。
- C-8.62 在 ExpressionTree 类 `build_expression_tree` 方法中的叶子节点的令牌可以是任何字符串，例如，其解析表达式 ' $(a*(b+c))$ '，然而在评估方法中，当尝试将一个叶子令牌转换为一个数字时会产生一个错误。修改这个评估方法，使其能够接受一个可选的 Python 中的字典，可以使用这样的字符串映射到数值。比如这样一个表达，`T.evaluate({'a': 3, 'b': 1, 'c': 5})`。通过这种方式，这个相同的代数表达式可以使用不用的值评估。
- C-8.63 正如 C-6.22 中提到的，后缀表达法是一种明确的算式表达式的方法。如果对于表达式 “ $(exp_1 \text{ op } exp_2)$ ”，这是一个正常的算术表达式的表示方法，其后缀表达式是 “ $pexp_1\ pexp_2\ \text{op}$ ”，其中 $pexp_1$ 和 $pexp_2$ 分别是 exp_1 和 exp_2 的后缀表示法。一个数字和一个常量的后缀表示法就是其本身。例如，表达式 “ $((5+2)*(8-3))/4$ ” 的后缀表达式是 “ $5\ 2\ +\ 8\ 3\ -\ *\ 4\ /$ ”。请实现 8.5 节给出的 ExpressionTree 类中实现了这种后缀表达式的方法 `postfix`。

项目

- P-8.64 使用 8.3.2 节描述的基于数组的表示实现二叉树的抽象数据类型。
- P-8.65 使用 8.3.3 节描述的链表结构实现树的抽象数据类型，并给出一个合理的更新方法集。
- P-8.66 `LinkedBinaryTree` 类的内存使用能够通过移除每个节点的双亲节点的指针来改进。不用保存每个位置实例的位置，而是用一系列节点代表从根节点到每个节点的整个路径（这通常可以节省内存是因为这样可以存储更少的指针）。使用这种策略重新实现 `LinkedBinaryTree` 类。
- P-8.67 切片平面图将一个矩形分割成垂直和平行切片两种形式（见图 8-25a）。一种切片平面图能够通过一个恰当的二叉树表示——称为切片二叉树。其内部节点代表切片，并且整个外部节点表示整个切片中的基本矩形（见图 8-25b）。这个压缩问题的描述为：假定一个切片平面图中

每个基本的平面图像被指定了一个最小的宽度 w 和一个最小的高度 h 。这个压缩问题就是找到其中宽度最小和高度最小的矩形。这个问题需要对每个位置 p 指定 $h(p)$ 和 $w(p)$ ，如下所示：

$$w(p) = \begin{cases} w & \text{如果 } p \text{ 是叶子节点, 其基本矩形有最小宽度 } w \\ \max(w(l), w(r)) & \text{如果 } p \text{ 是内部节点, 则是一个水平切片, 左孩子为 } l, \text{ 右孩子为 } r \\ w(l) + w(r) & \text{如果 } p \text{ 是内部节点, 则是一个垂直切片, 左孩子为 } l, \text{ 右孩子为 } r \end{cases}$$

$$h(p) = \begin{cases} h & \text{如果 } p \text{ 是叶子节点, 其基本矩形有最小高度 } h \\ h(l) + h(r) & \text{如果 } p \text{ 是内部节点, 则是一个水平切片, 左孩子为 } l, \text{ 右孩子为 } r \\ \max(h(l), h(r)) & \text{如果 } p \text{ 是内部节点, 则是一个垂直切片, 左孩子为 } l, \text{ 右孩子为 } r \end{cases}$$

设计一个支持如下操作的数据结构：

- 创建一个切片平面图以包含一个基本矩形。
- 通过水平方式分解一个基本图形。
- 通过垂直方式分解一个基本图形。
- 给一个基本图形分配最小的长宽。
- 画出一个切片平面图的树。
- 画出一个切片平面图。

P-8.68 写一个能够有效地玩三连棋（或称井字棋）（见 5.6 节）的程序，为了实现这个程序，你需要创建一棵博弈树 T ，其中每个节点都需要进行参数配置。在 8.4.2 节描述的情况下，根节点是最初的配置。对于每个内部节点 p ， p 的孩子节点反映了我们能够从 p 节点获取的游戏状态，即 A （第一个玩家）或者 B （第二个玩家）的一步符合规则的移动。偶数深度的位置与 A 的移动有关，奇数深度的位置与 B 的移动有关。叶子节点要么是最终游戏的状态，要么就是我们不想继续探索的状态。我们计算每一个叶子节点的值，以表示玩家 A 状态的好坏。在大型游戏中（如象棋），我们需要使用一个启发式的函数，但是对于小游戏（如井字棋），我们能够构造整个博弈树并且为叶子节点赋值 $+1$ 、 -1 和 0 ，以此来表明玩家 A 获胜、平局还是失败。在选择移动方式时，一个好的算法是极大极小算法。在这个算法中，我们分配一个分数给每一个内部节点 p ，这样 p 就代表 A 的方向。我们计算 p 的最高分数作为 p 的孩子节点。如果内部节点代表 B 的方向，则计算 p 的最小分数作为 p 的孩子节点。

P-8.69 使用练习 C-8.43 描述的二叉树实现树的抽象数据结构。你需要使用 `LinkedBinaryTree` 实现。

P-8.70 试编写一个程序，用于将一棵树和树中的一个节点 p 作为输入，将其转换成另外一棵有相同节点的树，但这次 p 是根节点。

扩展阅读

经典先序、中序和后序树遍历方法的讨论可以在 Knuth 的《Fundamental Algorithms》一书^[64] 中找到。欧拉遍历技术源于并行算法社区，它由 Tarjan 和 Vishkin^[93] 引入，由 JáJá^[54] 和 Karp 和 Ramachandran^[58] 进行了讨论。建树的算法通常被认为是建图算法的一部分。对建图感兴趣的读者可以参考由 Di Battista、Eades、Tamassia 和 Tollis 编写的书^[34] 以及 Tamassia 和 Liotta^[92] 的调查。

优先级队列

9.1 优先级队列的抽象数据类型

9.1.1 优先级

在第 6 章，我们介绍了队列 ADT 是一个根据先进先出（FIFO）策略在队列中添加和移除数据的对象集合。公司的客户呼叫中心实现了这样一个模型：在该模型中，客户被告知“呼叫将按照呼叫中心接受的顺序来应答”。在其设置中，一个新的呼叫被追加到队列的末尾，每当一个客户服务代表可以提供服务时，他将应答等待队列最前端的客户。

在现实生活中，有许多应用使用类似队列的结构来管理需要顺序处理的对象，但仅有先进先出的策略是不够的。比如，假设一个空中交通管制中心必须决定在众多即将降落的航班中先为哪次航班清理跑道。这个选择可能受到各种因素的影响，比如每个飞机跑道之间的距离、着陆过程中所用的时间或燃料的余量。着陆决定纯粹基于一个 FIFO 策略是不太可能的。

“先来先服务”策略在某些情况下是合理的，但在另一些情况下，优先级才是起决定作用的。现在，我们用另一个航空公司的例子加以说明，假设一个航班在起飞前一个小时被订满，由于有旅客取消的可能，航空公司维护了一个希望获得座位的候补等待（standby）旅客的队列。尽管等待旅客的优先级受到其检票时间的影响，但包括支付机票和是否频繁飞行（常飞乘客）在内的其他因素都需要考虑。因此，如果某位乘客被航空公司代理赋予了更高的优先级，那么当飞机上出现空闲座位时，即使他比其他乘客到得晚，他也有可能买到这张机票。

在本章中，我们介绍一个新的抽象数据类型，那就是优先级队列。这是一个包含优先级元素的集合，这个集合允许插入任意的元素，并允许删除拥有最高优先级的元素。当一个元素被插入优先级队列中时，用户可以通过提供一个关联键来为该元素赋予一定的优先级。键值最小的元素将是下一个从队列中移除的元素（因此，一个键值为 1 的元素将获得比键值为 2 的元素更高的优先级）。虽然用数字表示优先级是相当普遍的，但是任何 Python 对象，只要对象类型中的任何实例 `a` 和 `b`，对于 `a < b` 都支持一个一致的释义，那么该对象就可以用于定义键的自然顺序。有了这样的普遍性，应用程序可以为每个元素定义它们自己的优先级概念。比如，不同的金融分析师可以给特定的资产指定不同的评级（即优先级），如股票的份额。

9.1.2 优先级队列的抽象数据类型的实现

我们形式化地将一个元素和它的优先级用一个 key-value 对进行建模。我们在优先级队列 `P` 上定义优先级队列 ADT，以支持如下的方法：

- `P.add(k, v)`: 向优先级队列 P 中插入一个拥有键 k 和值 v 的元组。
- `P.min()`: 返回一个元组 (k, v), 代表优先级队列 P 中一个包含键和值的元组, 该元组的键值是最小值 (但是没有移除该元组); 如果队列为空, 将发生错误。
- `P.remove_min()`: 从优先级队列 P 中移除一个拥有最小键值的元组, 并且返回这个被移除的元组, (k, v) 代表这个被移除的元组的键和值; 如果优先级队列为空, 将发生错误。
- `P.is_empty()`: 如果优先级队列不包含任何元组, 将返回 True。
- `len(P)`: 返回优先级队列中元组的数量。

一个优先级队列中可能包含多个键值相等的条目, 在这种情况下 `min` 和 `remove_min` 方法可能从具有最小键值的元组中任选一个返回。值可以是任何对象类型。

在优先级队列的初始模型中, 假设一个元素一旦被加入优先级队列, 它的键值将保持不变。在 9.5 节中, 我们考虑对这个初始模型进行扩展, 扩展后允许用户更新优先级队列中的元素的键。

例题 9-1: 下表展示了一个初始为空的优先级队列 P 中的一系列操作及其产生的效果。由于它将条目以键排序的元组形式列出, 因此“优先级队列”一列是有误的。这样的一个内部表示不需要优先级队列。

操作	返回值	优先级队列
<code>P.add(5, A)</code>		<code>{(5, A)}</code>
<code>P.add(9, C)</code>		<code>{(5, A), (9, C)}</code>
<code>P.add(3, B)</code>		<code>{(3, B), (5, A), (9, C)}</code>
<code>P.add(7, D)</code>		<code>{(3, B), (5, A), (7, D), (9, C)}</code>
<code>P.min()</code>	(3, B)	<code>{(3, B), (5, A), (7, D), (9, C)}</code>
<code>P.remove_min()</code>	(3, B)	<code>{(5, A), (7, D), (9, C)}</code>
<code>P.remove_min()</code>	(5, A)	<code>{(7, D), (9, C)}</code>
<code>len(P)</code>	2	<code>{(7, D), (9, C)}</code>
<code>P.remove_min()</code>	(7, D)	<code>{(9, C)}</code>
<code>P.remove_min()</code>	(9, C)	<code>{}</code>
<code>P.is_empty()</code>	True	<code>{}</code>
<code>P.remove_min()</code>	“error”	<code>{}</code>

9.2 优先级队列的实现

在本节中, 我们将展示如何通过给一个位置列表 L 中的条目排序来实现一个优先级队列 (见 7.4 节)。根据在列表 L 中保存条目时是否按键排序, 我们提供了两种实现。

9.2.1 组合设计模式

即使在数据结构中已经重新定义了元组, 我们仍需要同时追踪元素和它的键值, 这是实现优先级队列的挑战之一。这一点让我们想起在 7.6 节的案例讨论中, 我们为每个元素维护一个访问计数器的做法。在那种设定下, 我们介绍了组合设计模式, 定义了一个 `_Item` 类, 用它来确保在主要的数据结构中每个元组保存它相关计数值。

对于优先级队列，我们将使用组合设计模式来存储内部元组，该元组包含键 k 和值 v 构成的数值对。为了在所有优先级队列中实现这种概念，我们给出了一个 `PriorityQueueBase` 类（见代码段 9-1），其中包含一个嵌套类 `_Item` 的定义。对于元组实例 `a` 和 `b`，我们基于关键字定义了语法 $a < b$ 。

代码段 9-1 `PriorityQueueBase` 类包含一个嵌套类 `_Item`，它将键和值组成单独的对象。为了方便，我们给出了 `is_empty` 的具体实现，它是在一个假定的 `__len__` 的实现的基础上实现的

```

1  class PriorityQueueBase:
2      """Abstract base class for a priority queue."""
3
4      class _Item:
5          """Lightweight composite to store priority queue items."""
6          __slots__ = '_key', '_value'
7
8          def __init__(self, k, v):
9              self._key = k
10             self._value = v
11
12         def __lt__(self, other):
13             return self._key < other._key      # compare items based on their keys
14
15     def is_empty(self):                  # concrete method assuming abstract len
16         """Return True if the priority queue is empty."""
17         return len(self) == 0

```

9.2.2 使用未排序列表实现优先级队列

在第一个具体的优先级队列实现中，我们使用一个未排序列表存储各个条目。代码段 9-2 中给出了 `UnsortedPriorityQueue` 类，它继承自代码段 9-1 中的 `PriorityQueueBase` 类。对于内部存储，键 - 值对是使用继承类 `_Item` 的实例进行组合表示的。这些元组是用 `PositionalList` 存储的，它们被视为类中的 `_data` 成员。在 7.4 节中，我们假设位置列表用一个双向链表实现，因此所有 ADT 操作执行的时间复杂度为 $O(1)$ 。

在构建一个新的优先级队列时，我们从一个空的列表开始。无论何时，列表的大小都等于存储在优先级队列中键 - 值对的数量。由于这个原因，优先级队列 `__len__` 方法能够简单地返回内部 `_data` 列表的长度。通过设计我们的 `PriorityQueueBase` 类，可以继承 `is_empty` 方法的具体实现，这种方法依赖于调用我们的 `__len__` 方法。

通过 `add` 方法，每次将一个键 - 值对追加到优先级队列中，对于给定的键和值，我们创建了一个新的 `_Item` 的元组（组成），并且将这个元组追加到列表的末端。这一实现的时间复杂度为 $O(1)$ 。

当 `min` 或者 `remove_min` 方法被调用时，我们必须定位键值最小的元组，这是另一个挑战。由于元组没有被排序，我们必须检查所有元组才能找到键值最小的元组。为了方便，我们定义了一个非公有的方法 `_find_min`，它用于返回键值最小的元组的位置。获得了位置信息，就允许 `remove_min` 方法可以在位置列表上调用 `delete` 方法。当准备返回一个键 - 值对元组时，`min` 方法可以简单地使用位置来检索列表元组。由于是用循环查找最小键值的，因此 `min` 和 `remove_min` 方法的时间复杂度均为 $O(n)$ ，其中 n 为优先级队列中元组的数量。

对于 `UnsortedPriorityQueue` 类的时间复杂度的总结见表 9-1。

表 9-1 长度为 n 的优先级队列中各方法最坏情况下的运行时间。以未排序的双向链表实现，空间需求为 $O(n)$

操作	运行时间
len	$O(1)$
is_empty	$O(1)$
add	$O(1)$
min	$O(n)$
remove_min	$O(n)$

代码段 9-2 使用未排序列表实现的优先级队列。父类 PriorityQueueBase 由代码段 9-1 给出，PositionalList 类来源于 7.4 节

```

1 class UnsortedPriorityQueue(PriorityQueueBase): # base class defines _Item
2     """A min-oriented priority queue implemented with an unsorted list."""
3
4     def _find_min(self):           # nonpublic utility
5         """Return Position of item with minimum key."""
6         if self.is_empty():        # is_empty inherited from base class
7             raise Empty('Priority queue is empty')
8         small = self._data.first()
9         walk = self._data.after(small)
10        while walk is not None:
11            if walk.element() < small.element():
12                small = walk
13                walk = self._data.after(walk)
14        return small
15
16    def __init__(self):
17        """Create a new empty Priority Queue."""
18        self._data = PositionalList()
19
20    def __len__(self):
21        """Return the number of items in the priority queue."""
22        return len(self._data)
23
24    def add(self, key, value):
25        """Add a key-value pair."""
26        self._data.add_last(self._Item(key, value))
27
28    def min(self):
29        """Return but do not remove (k,v) tuple with minimum key."""
30        p = self._find_min()
31        item = p.element()
32        return (item._key, item._value)
33
34    def remove_min(self):
35        """Remove and return (k,v) tuple with minimum key."""
36        p = self._find_min()
37        item = self._data.delete(p)
38        return (item._key, item._value)

```

9.2.3 使用排序列表实现优先级队列

优先级队列的另一个替代实现是使用位置列表，列表中的元组以键值非递减的顺序进行排序。这样可以保证列表的第一个元组是拥有最小键值的元组。

代码段 9-3 给出了 SortedPriorityQueue 类。方法 min 和 remove_min 的实现相当直接地

给出了列表的第一个元素拥有最小键值的信息。我们根据位置列表的 `first` 方法来找到第一个元组的位置，并使用 `delete` 方法来删除列表中的元组。假设列表是使用一个双向链表实现的，那么 `min` 和 `remove_min` 操作的时间复杂度为 $O(1)$ 。

然而，这个好处是以 `add` 方法花费更多的时间成本为代价的，我们需要扫描列表来找到合适的位置，以插入新的元组。实现从列表的结尾开始反方向查找，直到新的键值比当前元组的键值小为止；在最坏情况下，这个操作会一直扫描到列表的最前端。因此，`add` 方法在最坏情况下的时间复杂度是 $O(n)$ ， n 是执行该方法时优先级队列元组的数量。总之，当使用一个已排序列表来实现优先级队列时，插入操作的运行时间是线性的，而查找和移除最小键值的元组的操作则能在常数时间内完成。

比较两种基于列表的实现

表 9-2 详细地比较了分别通过已排序列表和未排序列表实现的优先级队列的各方法的运行时间。当使用列表来实现优先级队列 ADT 时，我们看到一个有趣的权衡。一个未排序的列表会支持快速插入操作，但是查询和删除操作就会比较慢；相反，一个已排序列表实现的优先级队列支持快速查询和删除操作，但是插入操作就比较慢。

表 9-2 大小为 n 的优先级队列的各方法在最坏情况下的运行时间。假设列表是由双向链表实现的，其空间使用量为 $O(n)$

操作	未排序列表	排序列表
Len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

代码段 9-3 使用排序列表实现的优先级队列。父类 `PriorityQueueBase` 在代码段 9-1 中给出，`PositionalList` 类在 7.4 节给出

```

1 class SortedPriorityQueue(PriorityQueueBase): # base class defines _Item
2     """A min-oriented priority queue implemented with a sorted list."""
3
4     def __init__(self):
5         """Create a new empty Priority Queue."""
6         self._data = PositionalList()
7
8     def __len__(self):
9         """Return the number of items in the priority queue."""
10        return len(self._data)
11
12    def add(self, key, value):
13        """Add a key-value pair."""
14        newest = self._Item(key, value)           # make new item instance
15        walk = self._data.last()                # walk backward looking for smaller key
16        while walk is not None and newest < walk.element():
17            walk = self._data.before(walk)
18        if walk is None:
19            self._data.add_first(newest)          # new key is smallest
20        else:
21            self._data.add_after(walk, newest)   # newest goes after walk
22
23    def min(self):
24        """Return but do not remove (k,v) tuple with minimum key."""

```

```

25     if self.is_empty():
26         raise Empty('Priority queue is empty.')
27     p = self._data.first()
28     item = p.element()
29     return (item._key, item._value)
30
31 def remove_min(self):
32     """ Remove and return (k,v) tuple with minimum key."""
33     if self.is_empty():
34         raise Empty('Priority queue is empty.')
35     item = self._data.delete(self._data.first())
36     return (item._key, item._value)

```

9.3 堆

在前面的两节中，实现优先级队列 ADT 的两种策略展示了一个有趣的权衡。当使用一个未排序列表来存储元组时，我们能够以 $O(1)$ 的时间复杂度实现插入，但是查找或者移除一个具有最小键值的元组则需要时间复杂度为 $O(n)$ 的循环操作来遍历整个元组集合。相对应地，如果使用一个已排序列表实现的优先级队列，则可以以 $O(1)$ 的时间复杂度查找或者移除具有最小键值的元组，但是向队列追加一个新的元素就需要 $O(n)$ 的时间来重新存储这个排序列表的序列。

在本节中，我们使用一个称为二进制堆的数据结构来给出一个更加有效的优先级队列的实现。这个数据结构允许我们以对数时间复杂度来实现插入和删除操作，这相对于 9.2 节讨论的基于列表的实现有很大的改善。利用堆实现这种改善的基本方式是使用二叉树的数据结构来在元素是完全无序和完全排好序之间取得折中。

9.3.1 堆的数据结构

堆（见图 9-1）是一棵二叉树 T ，该树在它的位置（节点）上存储了集合中的元组并且满足两个附加的属性：关系属性以存储键的形式在 T 中定义；结构属性以树 T 自身形状的方式定义。关系属性如下：

Heap-Order 属性：在堆 T 中，对于除了根的每个位置 p ，存储在 p 中的键值大于或等于存储在 p 的父节点的键值。

作为 Heap-Order 属性的结果， T 中从根到叶子的路径上的键值是以非递减顺序排列的。也就是说，一个最小的键总是存储在 T 的根节点中。这使得调用 `min` 或 `remove_min` 时，能够比较容易地定位这样的元组，一般情况下它被认为“在堆的顶部”（因此，给这种数据结构命名为“堆”）。顺便说一下，这里定义的数据结构堆与被用作支持一种程序语言（如 Python）的运行环境的内存堆（见 15.1.1 节）没并无任何关系。

由于效率的缘故，我们想让堆 T 的高度尽可能小，原因后面就会清楚。我们通过坚持让堆 T 满足结构属性中的附加属性，来强制满足让堆的高度尽可能小这一需求。——它必须是完全二叉树。

完全二叉树属性：一个高度为 h 的堆 T 是一棵完全二叉树，那么 T 的 $0, 1, 2, \dots, h-1$ 层上有可能达到节点数的最大值（即， i 层上有 2^i 个节点，且 $0 \leq i \leq h-1$ ），并且剩余的节点在 h 级尽可能保存在最左的位置。

图 9-1 中的树是完全二叉树，因为树的 0、1、2 层都是满的，并且 3 层的 6 个节点都处在

该层的最左边位置上。对于最左边位置的正式说法，我们可以参考 8.3.2 节中有关层级编号的讨论，即基于数组的二叉树表示的相关内容（事实上，在 9.3.3 节中，我们将会讨论使用数组来表示堆）。一棵含有 n 个节点的完全二叉树，是一棵含有从 0 到 $n - 1$ 层级编号的位置的树。比如，在一个基于数组的完全二叉树的表示中，它的 13 个元组将被连续地存储在 $A[0]$ 到 $A[12]$ 中。

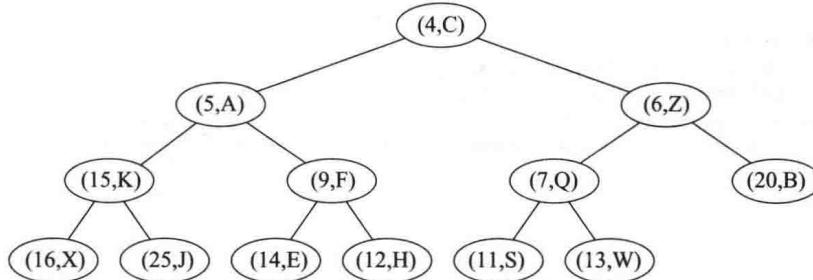


图 9-1 一个含有 13 个条目的堆排序的例子。最后一个节点保存的元组为 (13, W)

堆的高度

使用 h 表示 T 的高度。 T 为完全二叉树一定会有一个重要的结论，如命题 9-2 所示。

命题 9-2：堆 T 有 n 个元组，则它的高度 $h = \lfloor \log n \rfloor$ 。

证明：由 T 是完全二叉树可知，完全二叉树 T 0 ~ $h - 1$ 层节点的数量是 $1 + 2 + 4 + \dots + 2^{h-1} = 2^h - 1$ ，并且在 h 层的节点数最少为 1 个最多为 2^h 个。因此可得：

$$n \geq 2^h - 1 + 1 = 2^h \text{ 和 } n \leq 2^h - 1 + 2^h = 2^{h+1} - 1$$

给不等式 $2^h \leq n$ 两边取对数，得到高度 $h \leq \log n$ 。给不等式 $n \leq 2^{h+1} - 1$ 两边取对数，得到 $\log(n+1) - 1 \leq h$ 。由于 h 为整数，因此这两个不等式可简化为 $h = \lfloor \log n \rfloor$ 。 ■

9.3.2 使用堆实现优先级队列

命题 9-2 有一个重要的结论，那就是如果能以与堆的高度成比例的时间执行更新操作，那么这些操作将在对数级的时间内完成。现在，我们来讨论如何有效地使用堆来实现优先级队列中的各个方法。

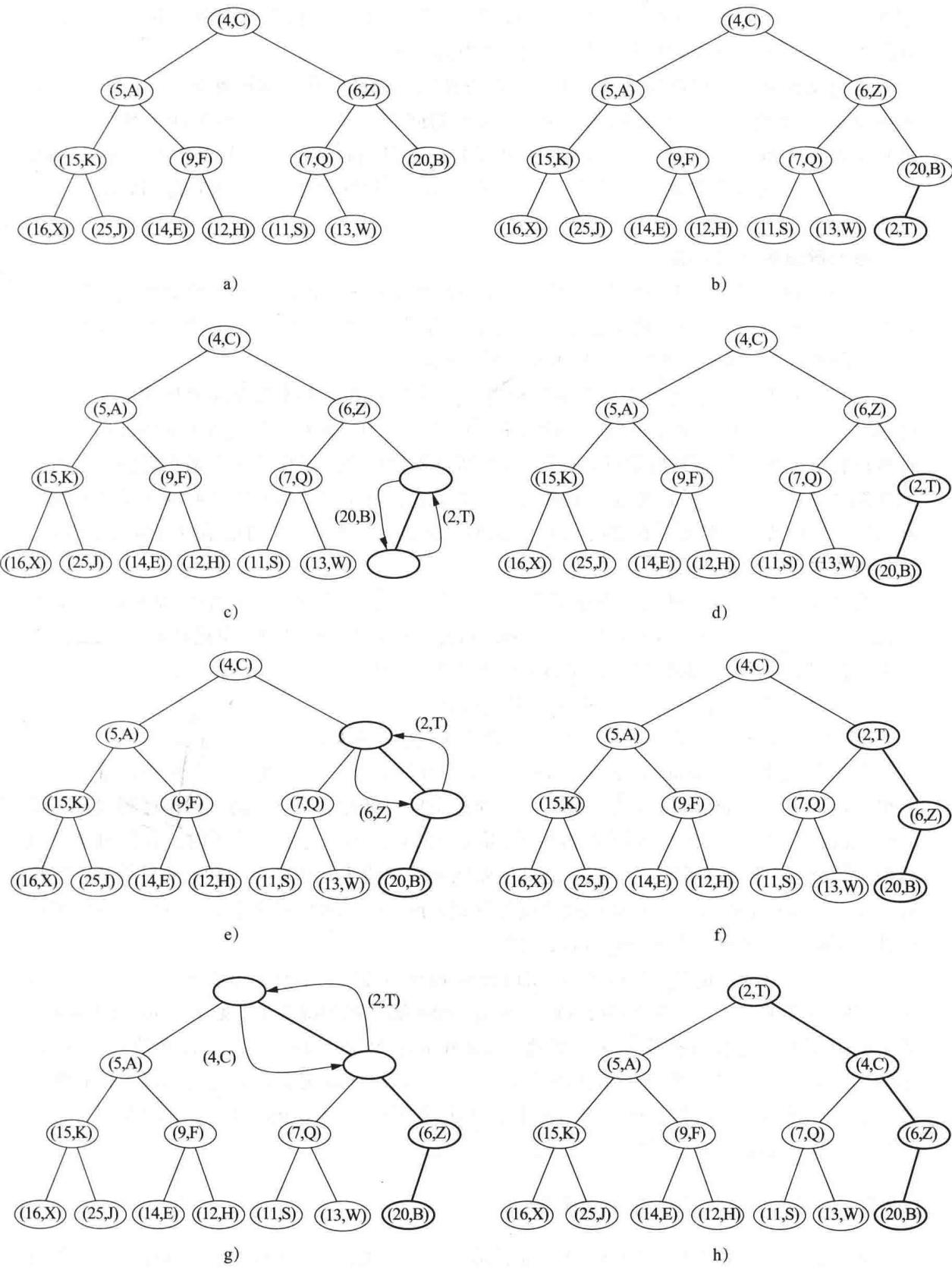
我们将使用 9.2.1 节的组合模式来在堆中存储键 – 值对的元组。`len` 和 `is_empty` 方法能够基于对树的检测来实现。`min` 操作相当简单，因为堆的属性保证了树的根部元组有最小的键值。`add` 和 `remove_min` 的实现方法都是有趣的算法。

在堆中增加一个元组

让我们考虑如何在一个用堆 T 实现的优先级队列上实现 `add(k, v)` 方法。我们把键值对 (k, v) 作为元组存储在树的新节点中。为了维持完全二叉树属性，这个新节点应该被放在位置 p 上，即树底层最右节点相邻的位置。如果树的底层节点已满（或堆为空），则应存放在新一层的最左位置上。

插入元组后堆向上冒泡

在这个操作之后，树 T 为完全二叉树，但是它可能破坏了 heap-order 属性。因此，除非位置 p 是树 T 的根节点（也就是说，优先级队列在插入操作前是空的），否则我们将对 p 位置上的键值与 p 的父节点 q （定义 p 的父节点为 q ）上的键值进行比较。如果 $k_p \geq k_q$ ，则满足 heap-order 属性且算法终止。如果 $k_p < k_q$ ，则需要重新调整树以满足 heap-order 属性，我们通过调换存储在位置 p 和 q 的元组来实现（见图 9-2c 和图 9-2d）。这个交换导致新元组的



9-2 向图 9-1 的堆中插入一个键值为 2 的元组: a) 初始堆; b) 执行 add 操作之后; c) 和 d) 通过交换恢复局部的有序属性; e) 和 f) 另一次交换; g) 和 h) 最后一次交换

层次上移一层。而 heap-order 属性可能再次被破坏，因此，我们需要在树 T 重复以上操作，直到不再违背 heap-order 属性位置（见图 9-2 中的 e 和图 9-2h）。

通过交换方式上移新插入的元组是非常方便的，这种操作被称作堆向上冒泡（up-heap bubbling）。交换既解决了破坏 heap-order 属性的问题，又将元组在堆中向上移一层。在最坏情况下，堆向上冒泡会导致新增元组向上一直移动到堆 T 的根节点位置。所以，add 方法所执行的交换次数在最坏情况下等于 T 的高度。根据命题 9-2，我们得知高度的上界是 $\lfloor \log n \rfloor$ 。

移除键值最小的元组

让我们现在考虑优先级队列 ADT 的 remove_min 方法。我们知道键值最小的元组被存储在堆 T 的根节点 r 上（即使有多于一个元组含有最小键值）。但是，一般情况下我们不能简单删除节点 r ，因为这将产生两棵不相连通的子树。

相反，我们可以通过删除堆 T 最后位置 p 上的叶子节点来确保堆的形状满足完全二叉树属性，这个最后位置 p 是树最底层的最靠右的位置。为了保存最后位置 p 上的元组，我们将该位置上的元组复制到根节点 r （就是那个即将要执行删除操作的含有最小键值的元组）。图 9-3a 和图 9-3b 展示了有关这些步骤的一个例子，含最小键值的元组（4, C）被从根部删除之后，该位置由来自最后位置的元组（13, W）所填充。在最后位置的节点被从树中删除。

删除操作后堆向下冒泡

在还没有做任何处理时，即使 T 现在是完全二叉树，它也很有可能已经破坏了 heap-order 属性。如果 T 只有一个节点（根），那么 heap-order 属性可以很简单地满足且算法终止。否则，我们需要区分两种情况，这里将 p 初始化为 T 的根：

- 1) 如果 p 没有右孩子，令 c 表示 p 的左孩子。
- 2) 否则 (p 有两个孩子)，令 c 作为 p 的具有较小键值的孩子。

如果 $k_p \leq k_c$ ，则 heap-order 属性已经满足，算法终止；如果 $k_p > k_c$ ，则需要重新调整元组位置来满足 heap-order 属性。我们可以通过交换存储在 p 和 c 上的元组来使得局部满足 heap-order 属性（见图 9-3c 和图 9-3d）。值得注意的是，当 p 有两个孩子时，我们着重考虑两个孩子节点中较小的那个。不仅 c 的键值要比 p 的键值小，还要至少和 c 的兄弟节点的键值一样小。这样能够确保当较小的键值被提升到 p 或 c 的兄弟位置之上的位置时，我们能够通过局部调整的方式来满足 heap-order 属性。

在恢复了节点 p 相对于其孩子节点的 heap-order 属性后，节点 c 可能违反了该属性。因此，我们必须继续向下交换直到没有违反 heap-order 属性的情况发生（见图 9-3e ~ 图 9-3h）。这个向下交换的过程被称作堆向下冒泡（down-heap bubbling）。交换可以解决违反 heap-order 属性的问题或者导致该键值在堆中下移一层。在最坏情况下，元组会一直下移到堆的最底层（见图 9-3）。这样，在最坏情况下，在执行方法 remove_min 中交换的次数等于堆 T 的高度，即根据命题 9-2 可知，这个最大值是 $\lfloor \log n \rfloor$ 。

9.3.3 基于数组的完全二叉树表示

基于数组的二叉树表示（8.3.2 节）非常适合完全二叉树 T 。在这部分实现中我们还使用它， T 的元组被存储在基于数组的列表 A 中，因此，存储在 T 中位置 p 的元素的索引等于层数 $f(p)$ ， $f(p)$ 是 p 的函数，其定义如下：

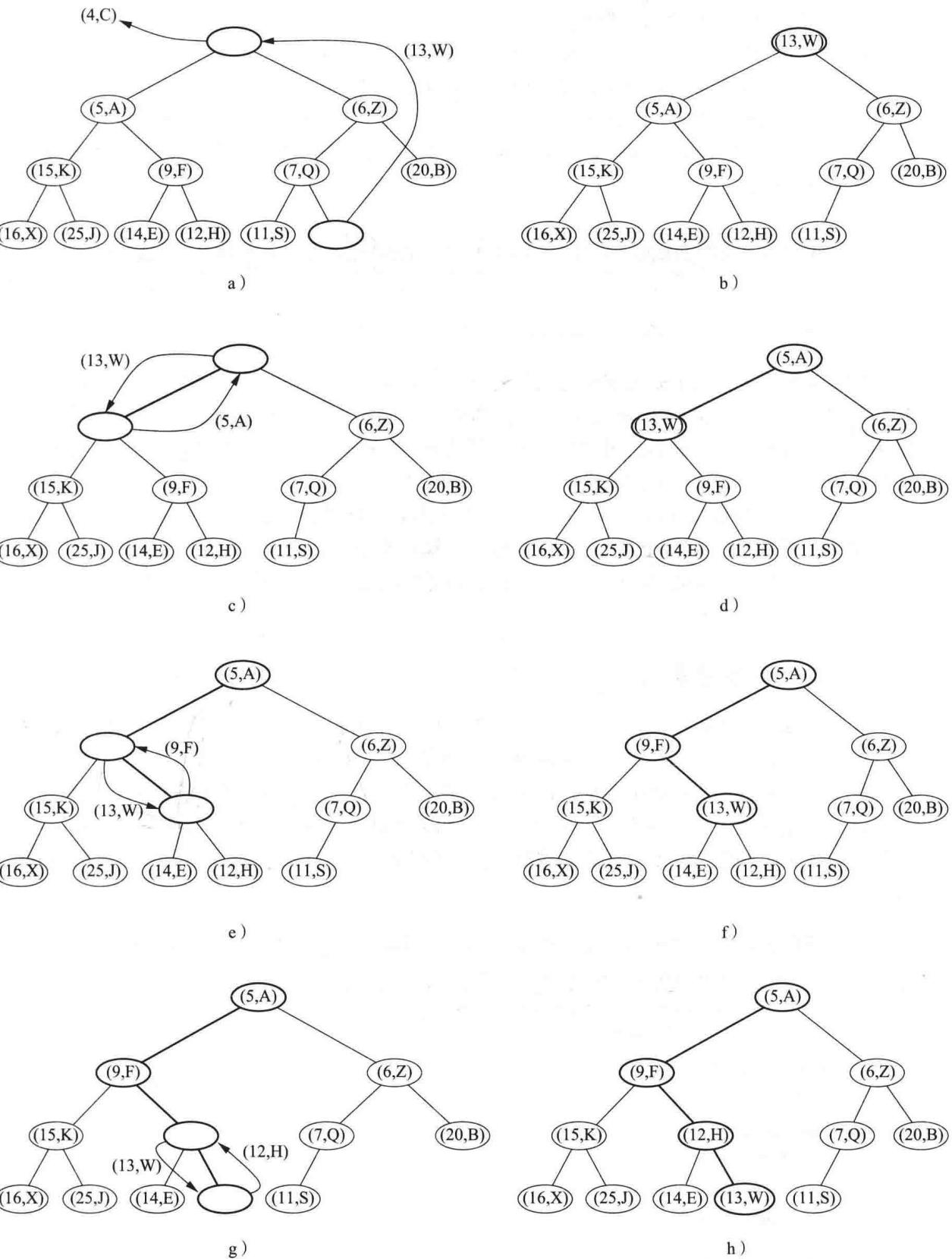


图 9-3 移除堆中键值最小的元组: a) 和 b) 删除最后的节点, 即存储在跟中的元组; c) 和 d) 通过交换恢复局部的 heap-order 属性; e) 和 f) 另一次交换; g) 和 h) 最后一次交换

- 如果 p 是 T 的根节点，则 $f(p) = 0$ 。
- 如果 p 是位置 q 的左孩子，则 $f(p) = 2f(q) + 1$ 。
- 如果 p 是位置 q 的右孩子，则 $f(p) = 2f(q) + 2$ 。

通过这种实现， T 的元组有落在 $[0, n - 1]$ 范围内相邻的索引，而且 T 最后节点的总是在索引 $n - 1$ 的位置上，其中 n 是 T 的元组数量。例如，图 9-1 基于数组表示的堆结构示意图如图 9-4 所示。

(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(8,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

图 9-4 图 9-1 基于数组表示的堆结构示意图

用基于数组表示的堆来实现优先级队列使我们避免了基于节点树结构的一些复杂性。尤其是优先级队列的 `add` 和 `remove_min` 操作都依靠定位大小为 n 的堆的最后一个索引位置。使用基于数组的表示，最后位置是数组中下标为 $n - 1$ 的位置。通过链结构实现定位完全二叉树的最后位置需要付出更多的代价（见练习 C-9.34）。

如果事先不知道优先级队列的大小，基于数组表示堆的使用就会引入偶尔动态重新设置数组大小的需要，就像 Python 列表一样。这样一个基于数组表示的节点数为 n 的完全二叉树的空间使用复杂度为 $O(n)$ ，而且增加和删除元组的方法的时间边界也需要考虑摊销（amortized）（见 5.3.1 节）。

9.3.4 Python 的堆实现

代码段 9-4 和代码段 9-5 提供了一个基于堆的优先级队列的 Python 实现。我们使用基于数组的表示，保存了元组组合表示的 Python 列表。虽然没有正式使用二叉树 ADT，但是代码段 9-4 包含了非公有效用函数，该函数能够计算父节点或另一个孩子节点的层次编号。这样就可以使用父节点、左孩子和右孩子等树相关术语来描述剩下的算法。但是，相关变量是整数索引（不是“位置”对象）。我们采用递归来实现 `_upheap` 和 `_downheap` 中的重复调用。

代码段 9-4 用基于数组的堆实现优先级队列（后接代码段 9-5），是代码段 9-1 中 PriorityQueueBase 类的扩展

```

1 class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
2     """A min-oriented priority queue implemented with a binary heap."""
3     #----- nonpublic behaviors -----
4     def _parent(self, j):
5         return (j-1) // 2
6
7     def _left(self, j):
8         return 2*j + 1
9
10    def _right(self, j):
11        return 2*j + 2
12
13    def _has_left(self, j):
14        return self._left(j) < len(self._data)      # index beyond end of list?
15
16    def _has_right(self, j):

```

```

17     return self._right(j) < len(self._data)    # index beyond end of list?
18
19 def _swap(self, i, j):
20     """Swap the elements at indices i and j of array."""
21     self._data[i], self._data[j] = self._data[j], self._data[i]
22
23 def _upheap(self, j):
24     parent = self._parent(j)
25     if j > 0 and self._data[j] < self._data[parent]:
26         self._swap(j, parent)
27         self._upheap(parent)           # recur at position of parent
28
29 def _downheap(self, j):
30     if self._has_left(j):
31         left = self._left(j)
32         small_child = left          # although right may be smaller
33         if self._has_right(j):
34             right = self._right(j)
35             if self._data[right] < self._data[left]:
36                 small_child = right
37         if self._data[small_child] < self._data[j]:
38             self._swap(j, small_child)
39             self._downheap(small_child) # recur at position of small child

```

代码段 9-5 用基于数组的堆实现优先级队列（接代码段 9-4）

```

40 #----- public behaviors -----
41 def __init__(self):
42     """Create a new empty Priority Queue."""
43     self._data = []
44
45 def __len__(self):
46     """Return the number of items in the priority queue."""
47     return len(self._data)
48
49 def add(self, key, value):
50     """Add a key-value pair to the priority queue."""
51     self._data.append(self._Item(key, value))
52     self._upheap(len(self._data) - 1)      # upheap newly added position
53
54 def min(self):
55     """Return but do not remove (k,v) tuple with minimum key.
56
57     Raise Empty exception if empty.
58     """
59     if self.is_empty():
60         raise Empty('Priority queue is empty.')
61     item = self._data[0]
62     return (item._key, item._value)
63
64 def remove_min(self):
65     """Remove and return (k,v) tuple with minimum key.
66
67     Raise Empty exception if empty.
68     """
69     if self.is_empty():
70         raise Empty('Priority queue is empty.')
71     self._swap(0, len(self._data) - 1)      # put minimum item at the end
72     item = self._data.pop()                # and remove it from the list;
73     self._downheap(0)                     # then fix new root
74     return (item._key, item._value)

```

9.3.5 基于堆的优先级队列的分析

表 9-3 显示了基于堆实现的优先级队列 ADT 各方法的运行时间，其中，假设两个键的比较能够在时间复杂度 $O(1)$ 内完成，而且堆 T 是基于数组表示的树或基于链表表示的树实现的。

简言之，每个优先级队列 ADT 方法能够在时间复杂度 $O(1)$ 或 $O(\log n)$ 内完成，其中 n 是执行方法时堆中元组的数量。这些方法的运行时间的分析是基于以下结论得出的：

- 堆 T 有 n 个节点，每个节点存储一个键 - 值对的引用。
- 由于堆 T 是完全二叉树，所有堆 T 的高度是 $O(\log n)$ （命题 9-1）。
- 由于树的根部包含最小元组，因此 min 操作运行的时间复杂度是 $O(1)$ 。
- 如 add 和 remove_min 操作中所需要的，定位堆的最后一个位置的操作，在基于数组表示的堆上完成需要的时间复杂度为 $O(1)$ ，在基于链表树表示的堆上需要以 $O(\log n)$ 的时间复杂度完成（见练习 C-9.34）。
- 堆向上冒泡和堆向下冒泡执行交换的次数在最坏情况下等于 T 的高度。

表 9-3 利用堆实现的优先级队列 P 的性能。 n 表示执行一个操作时优先级队列中元组的数量，空间需求量为 $O(n)$ 。操作 min 和 remove_min 的运行时间在基于数组表示的实现中是摊销的结果，因为动态数组有时候会调整大小；对于链表树结构，运行时间的边界是最坏情况下的结果

操作	运行时间
$\text{len}(P), P.\text{is_empty}()$	$O(1)$
$P.\text{min}()$	$O(1)$
$P.\text{add}()$	$O(\log n)^*$
$P.\text{remove_min}()$	$O(\log n)^*$

* 如果是基于数组的，则为摊销结果

我们可以得出这样的结论：无论堆使用链表结构还是数组结构实现，堆数据结构都是优先级队列 ADT 非常有效的实现方式。与基于未排序或已排序列表的实现不同，基于堆的实现插入和移除操作中均能快速地获得运行结果。

9.3.6 自底向上构建堆 *

如果以一个初始为空的堆开始，在最坏情况下，连续 n 次调用 add 操作的时间复杂度为 $O(n \log n)$ 。但是，如果所有存储在堆中的键 - 值对都事先给定，比如在堆排序算法的第一阶段，可以选择运行的时间复杂度为 $O(n)$ 的自下而上的方法构建堆（但是，堆排序仍然需要 $\Theta(n \log n)$ 的时间复杂度，因为在第二阶段我们仍然是重复地移除剩余元组中具有最小键值的一个）。

在这一节，我们描述了自底向上地构建堆，并给出了一个实现方法，基于堆的优先队列的构造函数可以使用这个实现方法来构建堆。

为了使叙述简单，我们在描述这种自底向上的堆构建时，假设键的数量为 n ，并且 n 为整数， $n = 2^{h+1} - 1$ 。也就是说，堆是一个每层都满的完全二叉树，所以堆的高度满足 $h = \log(n + 1) - 1$ 。以非递归的方法描述，自底向上构建堆包含以下 $h + 1 = \log(n + 1)$ 个步骤。

1) 第一步（见图 9-5b），我们构建 $(n + 1)/2$ 个基本堆，每个堆中仅存储一个元组。

2) 第二步（见图 9-5c ~ 图 9-5d），我们通过将基本堆成对连接起来并增加一个新元组来构建 $(n + 1)/4$ 个堆，这种堆的每个堆中存储了 3 个元组。新增的元组放在根部，并且它很有可能不得不与堆中某一个孩子节点存储的元组进行交换以保持 heap-order 属性。

3) 第三步(见图9-5e~图9-5f), 我们通过成对连接含3个元组的堆(该堆在上一步中构建), 并且增加一个新的元组, 从而构建 $(n+1)/8$ 个堆, 每个堆存储7个元组。新增的元组存储在根节点, 但是它可能通过堆向下冒泡算法下移以保持堆的heap-order属性。

.....

i) 第*i*步, $2 \leq i \leq h$, 我们通过成对连接存有 $(2^{i-1}-1)$ 个元组的堆(该堆是在前一步中构建的), 并且在每个合并的堆上增加一个新的元组来构建 $(n+1)/2^i$ 个堆, 每个堆存储 2^i-1 个元组。新增元组被存储在根节点上, 但是它很可能需要通过堆向下冒泡算法进行下移以保持堆的heap-order属性。

.....

$h+1$) 最后一步(见图9-5g~图9-5h), 我们通过连接两个存储了 $(n-1)/2$ 个元组的堆(该堆是在上一步中构建的), 并且增加新一个的元组来构建最终的堆, 该堆存储了所有*n*个元组。新增的元组开始存储在根节点, 但是它可能需要通过堆向下冒泡的算法下移以保持堆的heap-order属性。

$h=3$ 时, 自底向上的建堆过程如图9-5所示。

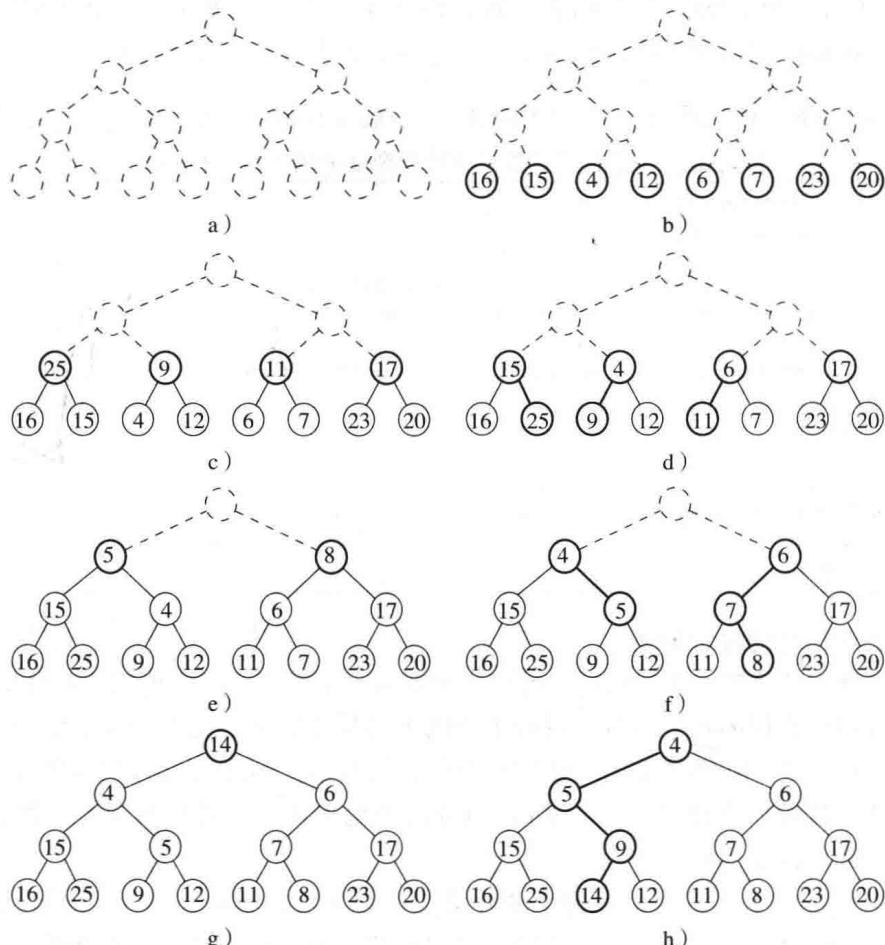


图9-5 含15个元组的自底向上构建堆: a) 和 b) 从最底层构建只含一个元组的堆开始构建; c) 和 d) 将这些堆合并成含3个元组的堆, 然后 e) 和 f) 构建含7个元组的堆, 直到 g) 和 h) 构建最终状态堆的构建。在 d)、f) 和 h) 中, 已经将堆向下冒泡的路径着重显示出来。为简单起见, 我们仅显示了每个节点的键值, 而不是显示整个元组的内容

自底向上构建堆的 Python 实现

当给定了“下堆”(down-heap)效用函数(utility function)时,实现自底向上构建堆是非常容易的。正如本章开头所描述的那样,相等大小的两个堆的“合并”就是公共位置 p 的两棵子树的合并,可以简单地通过 p 元组的下堆来完成,正如键值 14 在图 9-5f~图 9-5g 中所发生的变化。

在使用数组来表示堆时,如果我们初始化时将 n 个元组以任意顺序存储在数组中,就能够通过一个单层循环来实现自底向上的堆构造,该循环在树的每个位置上调用 `_downheap`,并且这些调用是有序进行的,从最底层开始并在树的根节点处结束。事实上,由于下堆被调用对叶节点无影响,因此这些循环可以从最底层的非叶节点开始。

在代码段 9-6 中,我们对 9.3.4 节的原始类 `HeapPriorityQueue` 进行了加强,从而支持一个初始化集合自底向上堆构造。我们介绍了一个非公有的方法 `_heapify`,它在每个非叶位置上调用 `_downheap`,从最底层开始,直到树的根节点结束。我们已经重新设计了该类的构造函数,以使其能接收一个可选的参数,该参数可以是任何 (k, v) 元组的序列。我们使用列表理解语法(见 1.9.2 节)来建造一个由给定内容的组合元组构成的初始化列表,而不是将 `self._data` 初始化为一个空列表。我们声明了一个空序列作为参数的默认值,作为 `HeapPriorityQueue()` 默认的语法,使其能够处理空的优先级队列并输出结果。

代码段 9-6 重写代码段 9-4 和代码段 9-5 中的类 `HeapPriorityQueue`,使其支持对给定的元组序列实现线性复杂度的优先级队列构建

```
def __init__(self, contents=()):
    """Create a new priority queue.

    By default, queue will be empty. If contents is given, it should be as an
    iterable sequence of (k,v) tuples specifying the initial contents.
    """
    self._data = [self._Item(k,v) for k,v in contents]      # empty by default
    if len(self._data) > 1:
        self._heapify()

def _heapify(self):
    start = self._parent(len(self) - 1)          # start at PARENT of last leaf
    for j in range(start, -1, -1):                # going to and including the root
        self._downheap(j)
```

自底向上堆构建的渐近分析

自底向上堆构建比向一个初始的空堆中逐个插入 n 个键值元组要更快,而且是渐近式的。直观地说,我们是在树的每个位置上进行单个的下堆操作,而不是单个的上堆操作。由于与树底部更近的节点多于离顶部近的,向下路径的总和是线性变化的,正如下面的命题所示。

命题 9-3: 假设两个键值可以在 $O(1)$ 的时间内完成比较,则使用 n 个元组自底向上构建堆需要的时间复杂度为 $O(n)$ 。

证明: 构建堆的主要成本是在每个非叶节点位置下堆的构造步骤上。用 π_v 表示堆从非叶节点 v 到其“中序后继”叶节点的路径,也就是说,该路径是从 v 节点开始,沿着 v 的右孩子,然后继续沿着最左方向下直至到达叶节点。虽然 π_v 不需要一定是从 v 节点向下冒泡步骤产生的路径,但是它的长度 $||\pi_v||$ (即 π_v 的边的个数)与以 v 为根的子树的高度成比例,因此,这也是节点 v 下堆操作的复杂度的边界。我们用路径大小的总和 $\sum_v ||\pi_v||$ 来限制自底

向上堆构造算法总的运行时间。直观地，图 9-6 展示了“可视化”的证明，用标签标记非叶节点 v 的路径 π_v 中所包含的每条边。

我们声明对于所有非叶节点 v 的路径 π_v 是不相交的，因此路径长度的和受到树的总边数的限制，即为 $O(n)$ 。为了展示这一结论，我们考虑定义的“向右学习”(right-leaning) 边和“向左学习”(left-leaning) 边(这些边从父节点到右孩子和左孩子)。一个特别的向右学习边 e 只能是节点 v 的路径 π_v 的一部分，在由 e 表示的关系中，该节点 v 是父节点。如果持续地向左向下直至到达叶节点，那么

所到达的叶节点可以用来对向左学习的边进行划分。每个非叶节点只使用在同组中的向左学习边将生成非叶节点的中序后继。由于每个非叶节点必须有不同的中序后继，因此没有两个路径包含相同的向左学习边。因此，我们断定自底向上构造堆的时间复杂度为 $O(n)$ 。 ■

9.3.7 Python 的 heapq 模块

Python 的标准分布包含一个 `heapq` 模块，该模块提供对基于堆的优先级队列的支持。该模块不提供任何优先级队列类，而是提供一些函数，这些函数把标准 Python 列表作为堆进行管理。它的模型与我们自己的基本相同：基于层次编号的索引，将 n 个元素存储在 $L[0] \sim L[n - 1]$ 的单元中，并且最小元素存储在根 $L[0]$ 中。我们注意到 `heapq` 并不是单独地管理相关的值，即元素作为它们自己的键值。

`Heapp` 模块支持如下函数，假设所有这些函数在调用之前，现有的列表 L 已经满足 `heap-order` 属性：

- `heappush(L, e)`：将元素 e 存入列表 L ，并重新调整列表以满足 `heap-order` 属性。该函数执行的时间复杂度为 $O(\log n)$ 。
- `heappop(L)`：取出并返回列表 L 中拥有最小值的元素，并且重新调整存储以满足 `heap-order` 属性。该函数执行的时间复杂度为 $O(\log n)$ 。
- `heappushpop(L, e)`：将元素 e 存入列表 L 中，同时取出和返回最小的元组。该函数执行的时间复杂度为 $O(\log n)$ ，但是它较分别调用 `push` 和 `pop` 方法的效率稍微高一些，因为列表的大小在处理过程中不发生变化。如果最新被插入列表的元素值是最小的，那么该函数立刻返回；否则，新增的元素将会替换在根节点处取出的元素，随后，函数会执行下堆操作。
- `heareplace(L, e)`：与 `heappushpop` 方法相类似，但相当于在插入操作前执行 `pop` 操作(换言之，即使新插入的元素是最小值也不能被返回)。该函数执行的时间复杂度为 $O(\log n)$ ，但是它比分别调用 `push` 和 `pop` 方法效率更高。

该模块还支持在不满足 `heap-order` 属性的序列上进行操作的其他函数。

- `heapify(L)`：改变未排序的列表，使其满足 `heap-order` 属性。这个函数使用自底向上的堆构造算法，时间复杂度为 $O(n)$ 。

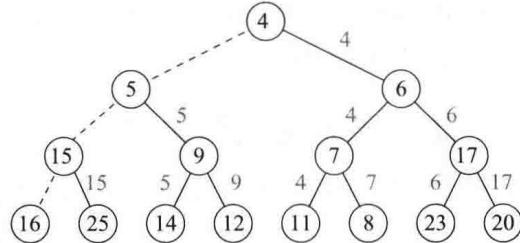


图 9-6 自底向上堆构建运行时间为线性的“可视化”证明。路径 π_v 所包含的每条边 e (如果有的话) 都加上含有节点 v 的标签

- `nlargest(k, iterable)`：从一个给定的迭代中生成含有 k 个最大值的列表。执行该函数的时间复杂度为 $O(n + k \log n)$ ，这里使用 n 来表示迭代的长度（见练习 C-9.42）。
- `nsmallest(k, iterable)`：从一个给定的迭代中生成含有 k 个最小值的列表。该函数使用与 `nlargest` 相同的技术，其时间复杂度为 $O(n + k \log n)$ 。

9.4 使用优先级队列排序

在定义优先级队列 ADT 时，我们注意到任何类型的对象都能够被定义为键，但是任何一对键之间必须是可比较的，这样这个键集自然是可排序的。在 Python 中，我们常用“ $<$ ”操作符来定义这样的序列，在定义过程中，必须满足属性：

- 漫反射特性： $k \prec k$ 。
- 传递属性：如果 $k_1 < k_2$ ，并且 $k_2 < k_3$ ，则 $k_1 < k_3$ 。

这种关系被正式地定义为严格弱序（strict weak order），因为它允许各个键值是相等的，但更广泛的等价类是完全有序的，因为它们可以根据传递属性排列成唯一的从最小值到最大值的序列。

作为优先级队列的第一个应用，我们展示了它们如何被用在对一个可比较元素集合 C 的排序上。也就是说，我们能够生成集合 C 中元素的一个递增排序的序列（或者如果存在重复数据，则至少是非递减的顺序）。这个算法非常简单——我们将所有元素插入一个最初为空的优先级队列中，然后重复调用 `remove_min`，从而以非递减的顺序获取所有元素。

我们在代码段 9-7 中给定了这种算法的一个实现，其中假定 C 是一个位置列表（见 7.4 节）。调用方法 `P.add` 时，我们把集合的原始元素 `element` 同时作为键和值，即 `P.add(element, element)`。

代码段 9-7 函数 `pq_sort` 的实现，这里假设已经有了一个合适的 `PriorityQueue` 类的实现。注意输入列表 C 的每个元素都充当了其在优先级队列 P 的键

```

1 def pq_sort(C):
2     """Sort a collection of elements stored in a positional list."""
3     n = len(C)
4     P = PriorityQueue()
5     for j in range(n):
6         element = C.delete(C.first())
7         P.add(element, element)      # use element as key and value
8     for j in range(n):
9         (k,v) = P.remove_min()      # store smallest remaining element in C
10        C.add_last(v)

```

如果对以上代码做个小小的改动：将元素按照一定的规则排序而不是保留其默认的顺序，这样便可以使该函数更为通用。例如，当处理字符串时，“ $<$ ”操作符定义一个字典序列，这是将一个字母序扩展到 Unicode 上。比如，我们定义 '`'2' < '4'`'，因为是根据每个字符串的第一个字母的顺序定义的，就像 '`'apple' < 'banana'`' 一样。假设有一个应用，在应用中我们有一个众所周知的代表整数值（如 '`'12'`'）的字符串列表，那么我们的目标就是根据这些对应的整数值给这些字符串排序。

Python 中提供了为一个排序算法自定义顺序的标准方法，作为排序函数的一个可选参数，一个对象自身就是一个给定的元素计算键的单参数函数（见 1.5 和 1.10 节，在内置 `max` 函数的上下文中有关于该方法的讨论）。比如，在使用一个（数字）字符串列表时，我

们很可能希望将 `int(s)` 的数值作为列表中字符串 `s` 的键。在这种情况下，`int` 类的构造函数可以作为计算键的单参数函数。在这种方式下，字符串 '4' 将排在字符串 '12' 的前面，因为它们的键的关系是 `int('4') < int('12')`。我们把用这种方法为 `pq_sort` 函数提供可选键参数的问题留作一个练习（见练习 C-9.46）。

9.4.1 选择排序和插入排序

对于任意给定的优先级队列类的有效实现，`pq_sort` 函数都能正确地处理。但是，排序算法的运行时间复杂度取决于给定的优先级队列类的 `add` 方法和 `remove_min` 方法的时间复杂度。接下来我们讨论一种优先级队列的实现，该实现实际上使得 `pq_sort` 计算成为经典的排序算法之一。

选择排序

如果用一个未排序的列表实现 `P`，那么由于每增加一个元素都能在 $O(1)$ 的时间复杂度内完成，所以在 `pq_sort` 的第一阶段所花费的时间复杂度为 $O(n)$ 。在第二阶段，每次 `remove_min` 操作的时间复杂度与 `P` 的大小成正比。因此，计算的瓶颈是在第二阶段重复地选择最小元素。由于这个原因，这个算法被命名为选择排序（见图 9-7）。

如上面提到的，算法的瓶颈就是我们在第二阶段重复地从优先级队列 `P` 中移除拥有最小键值的元组。`P` 的大小开始为 n ，随着每次调用 `remove_min`，持续递减，直到变为 0。所以，第一次操作的时间复杂度为 $O(n)$ ，第二次操作的时间复杂度为 $O(n - 1)$ ，以此类推。因此，第二阶段所需要的总时间为：

$$O(n + (n-1) + \dots + 2 + 1) = O\left(\sum_{i=1}^n i\right)$$

由命题 3-3 可知， $\sum_{i=1}^n i = n(n+1)/2$ 这一结论。因此，第二阶段的时间复杂度为 $O(n^2)$ ，故整个选择排序算法的时间复杂度为 $O(n^2)$ 。

插入排序

如果用一个排序列表实现优先级队列，由于此时每次在 `P` 上执行 `remove_min` 操作所花费的时间复杂度为 $O(1)$ ，因此我们可以将第二阶段的时间复杂度降低到 $O(n)$ 。不幸的是，第一阶段将会变成整个算法的瓶颈，因为在最坏情况下，每次 `add` 操作的时间复杂度与当前 `P` 的大小成正比。这种排序算法被称作插入排序（见图 9-8）。实际上，在优先级队列中增加一

	集合 C	优先级队列 P
输入	(7, 4, 8, 2, 5, 3)	()
阶段1	(a) (4, 8, 2, 5, 3) (b) (8, 2, 5, 3) ⋮ (f) 0	(7) (7, 4) ⋮ (7, 4, 8, 2, 5, 3)
阶段2	(a) (2) (b) (2, 3) (c) (2, 3, 4) (d) (2, 3, 4, 5) (e) (2, 3, 4, 5, 7) (f) (2, 3, 4, 5, 7, 8)	(7, 4, 8, 5, 3) (7, 4, 8, 5) (7, 8, 5) (7, 8) (8) 0

图 9-7 在集合 $C = (7, 4, 8, 2, 5, 3)$ 上执行选择排序

	集合 C	优先级队列 P
输入	(7, 4, 8, 2, 5, 3)	()
阶段1	(a) (4, 8, 2, 5, 3) (b) (8, 2, 5, 3) (c) (2, 5, 3) (d) (5, 3) (e) (3) (f) 0	(7) (4, 7) (4, 7, 8) (2, 4, 7, 8) (2, 4, 5, 7, 8) (2, 3, 4, 5, 7, 8)
阶段2	(a) (2) (b) (2, 3) ⋮ (f) (2, 3, 4, 5, 7, 8)	(3, 4, 5, 7, 8) (4, 5, 7, 8) ⋮ 0

图 9-8 在集合 $C = (7, 4, 8, 2, 5, 3)$ 上执行插入排序

个元素的实现与之前 7.5 节给出的插入算法的步骤几乎完全相同。

插入排序算法的第一阶段在最坏情况下的运行时间为：

$$O(1+2+\cdots+(n-1)+n) = O(\sum_{i=1}^n i)$$

同样，根据命题 3-2，这意味着最坏情况下第一阶段的时间复杂度为 $O(n^2)$ ，并且整个插入排序算法的时间复杂度也为 $O(n^2)$ 。但是，不同于选择排序，插入排序在最好情况下的时间复杂度为 $O(n)$ 。

9.4.2 堆排序

正如我们之前所看到的，使用堆实现的优先级队列比较有优势，因为优先级队列 ADT 中的所有方法都是在对数级时间或更短时间内完成。因此，这种实现非常适合那些所有优先级队列方法都追求快速的运行时间的应用。现在，让我们再次考虑 pq_sort 的设计，这次使用基于堆的优先级队列的实现方式。

在第一阶段，由于第 i 次 add 操作完成后堆有 i 个元组，所以第 i 次 add 操作的时间复杂度为 $O(\log i)$ 。因此，这一阶段整体的时间复杂度为 $O(n \log n)$ （采用 9.3.6 节所描述的自底向上堆构造的方法，第一阶段的时间复杂度能够被提升到 $O(n)$ ）。

在 pq_sort 的第二阶段，由于在第 j 次 remove_min 操作执行时堆中有 $(n - j + 1)$ 个元组，因此第 j 次 remove_min 操作的时间复杂度为 $O(\log(n - j + 1))$ 。将所有这些 remove_min 操作累加起来，这一阶段的时间复杂度为 $O(n \log n)$ 。所以，当使用堆来实现优先级队列时，整个优先级队列排序算法的时间复杂度为 $O(n \log n)$ 。这个排序算法就称为堆排序，以下命题总结了它的性能。

命题 9-4：假设集合 C 的任意两个元素能在 $O(1)$ 时间内完成比较，则堆排序算法能在 $O(n \log n)$ 时间内完成含有 n 个元素的集合 C 的排序。

显然，堆排序的 $O(n \log n)$ 时间复杂度比起选择排序和插入排序（见 9.4.1 节）的 $O(n^2)$ 时间复杂度性能是相当好的。

实现原地堆排序

如果集合 C 的排序由基于数组序列的方法实现，Python 列表就是一个典型的代表，我们可以通过引入一个常量因子以列表自身的一部分存储堆的方法来加速堆排序并减小空间需求，以避免使用辅助堆数据结构。这可以通过如下所示的算法修改进行实现：

1) 通过使每个位置的键值不小于其孩子节点的键值，我们重新定义堆的操作，使其成为面向最大值的堆（maximum-oriented heap）。这可以通过重新编码算法或者调整键的概念为相反方向的来实现。在算法执行过程中的任意时间点，我们始终使用 C 的左半部分（即 0 到一个确定的索引 $i - 1$ ）来存储堆中的元组，并且使用 C 的右半部分（即索引 $i \sim n - 1$ ）来存储序列的元素。也就是说， C 的前 i 个元素（在索引 $0, \dots, i - 1$ 处）提供了堆的数组列表表示。

2) 在算法的第一阶段，我们从一个空堆开始，并从左向右移动堆与序列之间的边界，一次一步。在第 i 步，这里 $i = 1, \dots, n$ ，我们通过在索引 $i - 1$ 处追加元素来对堆进行扩展。

3) 在算法的第二阶段，我们从一个空的序列开始，并从右到左移动堆与序列之间的边界，一次一步。在第 i 步，这里 $i = 1, \dots, n$ ，我们将最大值元素从堆中移除并将其存储到索引为 $n - i$ 的位置上。

一般来说，如果一个排序算法除了存储对象已排序的序列，仅额外使用一小部分内存，我们就说该算法为原地（in-place）算法。上述调整过的堆排序算法就是原地算法。相对于将元素移出序列再重新移入，我们简单地对序列进行了重新组织。我们在图 9-9 中对原地堆排序第二阶段的处理过程进行了说明。

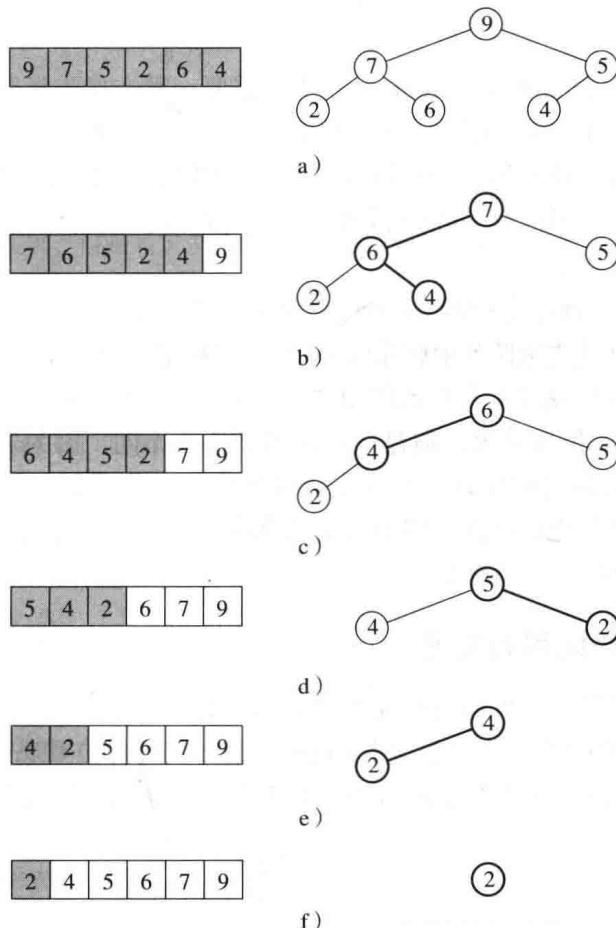


图 9-9 原地堆排序的第二阶段。序列中堆的部分做了突出表示。在每个表示序列的二叉树图表示中，最新的向下堆冒泡形成的路径做了突出表示

9.5 适应性优先级队列

9.1.2 节给出的优先级队列 ADT 的方法对于大多数优先级队列的基本应用（比如排序）来说已经很完善了。但是，有些场景还需要一些附加方法，比如下面所示的涉及航班候补等待（standby）乘客的应用场景。

- 持有消极态度的待机乘客可能会因为对等待感到疲倦而决定在登机时间到来之前离开，并请求从等待列表中移除。因此，我们将与该乘客相关的元组从优先级队列中移除。由于要离开的乘客不需要最高优先级，因此 `remove_min` 操作不能完成此任务。所以，我们需要一个新的操作 `remove`，用来删除优先级队列中的任意一个元组。
- 另一个待机乘客拿出她的常飞乘客金卡并出示给售票代理，因此她的优先级将被相应地更改。为了完成这个优先级的变更，我们需要一个新的操作 `update`，使我们能用一个新的键去替换元组现有的键。

在实现 14.6.2 节和 14.7.1 节的特定图算法时，我们将看到可适应性优先级队列的另一种应用。

在本节中，我们构建了一个可适应性优先级队列 ADT，并展示了如何将这个抽象概念作为基于堆的优先级队列的扩展来实现。

9.5.1 定位器

为了有效地实现方法 `update` 和 `remove`，我们需要一种在优先级队列中找到用户元组的机制，该机制可以避免在整个元组集合进行线性搜索。为了实现这一目标，当一个新的元素追加到优先级队列中时，我们返回一个特殊的对象给调用者，该对象称为定位器（locator）。对于一个优先级队列 P ，当执行 `update` 或者 `remove` 方法时，我们需要用户提供一个合适的定位器作为参数，详情如下：

- $P.update(loc, k, v)$: 用定位器 loc 代替键和值作为元组的标识。
- $P.remove(loc)$: 从优先级队列中删除以 loc 标识的特定元组，并返回它的 $(key, value)$ 对。

定位抽象类似于我们从 7.4 节开始使用的位置列表 ADT 中使用的位置抽象和第 8 章介绍的树的 ADT 中使用的位置抽象。但是，定位器和位置不同，因为优先级队列的定位器并不代表结构中一个元素的具体位置。在优先级队列中，一些看似与元素没有直接关系的操作，一旦执行，该元素可能在数据结构中被重新定位。只要一个元组项一直在队列中的某个地方，这个元组的定位器将一直有效。

9.5.2 适应性优先级队列的实现

在本节中，我们提供一个可适应性优先级队列的 Python 实现，将它作为 9.3.4 节所讨论的 `HeapPriorityQueue` 类的扩展。为了实现 `Locator` 类，我们将扩展现有 `_Item` 的组成来增加一个额外的字段，该字段指定在基于数组表示的堆中的元素的当前索引，如图 9-10 所示。

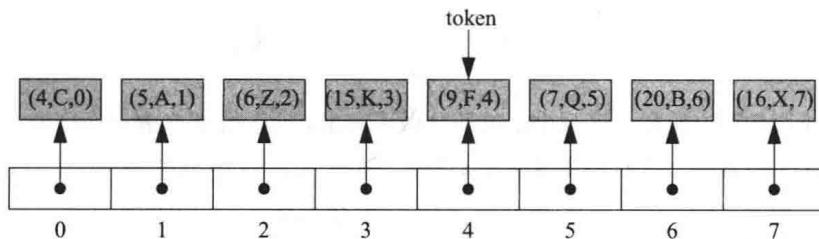


图 9-10 用一个定位器序列表示堆。数组中每个元组的索引对应每个定位器实例中的第三个元素。假定标识符 `token` 是用户域中的一个定位器的引用（reference）

该列表是一个指向定位器实例的序列，每个定位器都存储一个 `key`, `value` 和列表内元组的当前索引。用户会获得每个插入的元素的定位器实例的引用，如图 9-10 中的 `token` 标识所示。

在堆上执行优先级队列操作时，元组在结构中被重新定位，我们重新设置列表中各定位器实例的位置，并更新每个定位器的第三个字段以反映该定位器在列表中的新索引。图 9-11 展示了上述的堆在调用 `remove_min()` 方法后状态的一个例子。堆操作使得最小元组 $(4, C)$ 被删除，并使元组 $(16, X)$ 暂时从最后一个位置移到根位置，这之后是向下冒泡的处理阶段。在下堆阶段，元素 $(16, X)$ 与它在列表索引为 1 的位置的左孩子 $(5, A)$ 做了交换，

然后又与它在列表的索引值为 4 的右孩子元组 (9, F) 交换。在最后的配置中，所有受影响的元组的定位器实例都已经被修改了，以反映它们的新位置。

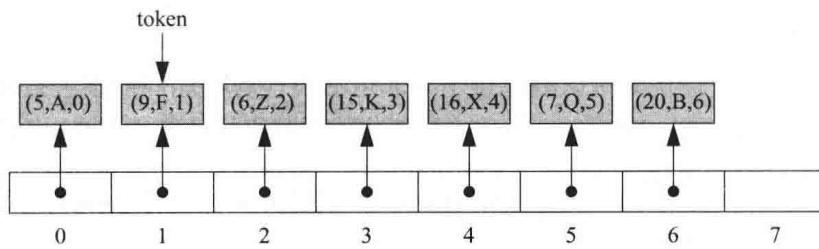


图 9-11 在图 9-10 中所描述的堆上调用 `remove_min()` 的结果。在初始配置中，标识 `token` 继续指向同一个定位器实例，但是当定位器增加了第三个域时，它在列表中的位置会发生变化

强调定位器实例没有改变元组标识非常重要。如图 9-10 和图 9-11 所示，用户 `token` 的指针将继续指向相同的实例。我们只是简单地改变了实例的第三个域，并改变了列表序列中引用该实例的索引的位置。

通过这种新的表示，对可适应性优先级队列 ADT 提供额外的支持更加直接。当一个定位器实例被当作参数传给方法 `update` 或 `remove` 时，我们可以借助该结构的第三个域来指明该元素在堆中的位置。根据前面的讨论我们知道，一个键的 `update` 操作仅需要简单的一次堆向上冒泡或堆向下冒泡来重新满足 `heap-order` 属性（完全二叉树属性保持不变）。为了实现移除任意一个元素的操作，我们把在最后位置的元素移到腾空的位置，并再次执行适当的冒泡操作来满足 `heap-order` 属性。

Python 实现

代码段 9-8 和代码段 9-9 展示了可适应性优先级队列的 Python 实现，它是 9.3.4 节 `HeapPriorityQueue` 类的子类。我们在原始类上做的修改非常小。我们定义了一个公有的 `Locator` 类，该类继承非公有的 `_Item` 类并通过额外的 `_index` 域增强它。之所以将它定义为公有类，是因为我们要同时用 `locators` 作为返回值和参数，但是，用户定位器类的公有接口不包括任何其他功能。

为了在堆操作的过程中更新定位器，我们借助一个特定的设计决策，即在原始类在所有数据移动中都使用一个非公有的方法 `_swap`。在两个互换的定位器实例中，我们重写该实用程序来执行更新定位器中所存储的索引的附加步骤。

我们提供一个新的 `_bubble` 程序，该程序负责一个在堆中任意位置的键改变时恢复 `heap-order` 属性，不管这个改变是由于键的更新，还是因为从树的最后一个位置移除元素及其对应的元组。`_bubble` 程序根据给定的位置是否有一个更小的父节点来决定是否进行堆向上冒泡或者堆向下冒泡（如果一个更新的键恰巧保存了有效的当前位置，我们在技术上调用 `_downheap` 但没有交换结果）。

代码段 9-9 给出了公有的方法。现有的 `add` 方法被覆盖，两者都是使用一个 `Locator` 实例（而不是存储新元素的 `_Item` 实例），并将定位器返回给调用者。该方法的其余部分与原有的方法相类似，即通过 `_swap` 新版本的使用来制定管理定位器的索引。由于对于可适应性优先级队列在行为上唯一需要的改变已经在重载 `_swap` 方法中提供，因此没有必要再重写 `remove_min` 方法。

`update` 和 `remove` 方法为可适应性优先级队列提供了核心的新功能。我们对一个被调用

方发送的定位器的有效性进行鲁棒性检查（为了节省篇幅，我们给出的代码不做确保参数确实是一个 Locator 实例的初步类型检查）。为了确保定位器与给定优先级队列中的当前元素相关联，我们检查被封装在定位器对象中的索引，然后验证在该索引处的列表的元组正是这个定位器。

综上所述，可适应性优先级队列提供了与非可适应性版本相同的渐近效率和空间使用，并且为新的基于定位器的 update 和 remove 方法提供了对数级的性能。表 9-4 给出了性能总结。

代码段 9-8 一个可适应性优先级队列的实现（后接代码段 9-9）。这是代码段 9-4 和代码段 9-5 中 HeapPriorityQueue 的扩展

```

1 class AdaptableHeapPriorityQueue(HeapPriorityQueue):
2     """A locator-based priority queue implemented with a binary heap."""
3
4     #----- nested Locator class -----
5     class Locator(HeapPriorityQueue._Item):
6         """Token for locating an entry of the priority queue."""
7         __slots__ = '_index'           # add index as additional field
8
9     def __init__(self, k, v, j):
10        super().__init__(k,v)
11        self._index = j
12
13    #----- nonpublic behaviors -----
14    # override swap to record new indices
15    def _swap(self, i, j):
16        super().__swap(i,j)          # perform the swap
17        self._data[i]._index = i      # reset locator index (post-swap)
18        self._data[j]._index = j      # reset locator index (post-swap)
19
20    def _bubble(self, j):
21        if j > 0 and self._data[j] < self._data[self._parent(j)]:
22            self._upheap(j)
23        else:
24            self._downheap(j)

```

代码段 9-9 一个可适应性优先级队列的实现（接代码段 9-8）

```

25 def add(self, key, value):
26     """Add a key-value pair."""
27     token = self.Locator(key, value, len(self._data)) # initiaize locator index
28     self._data.append(token)
29     self._upheap(len(self._data) - 1)
30     return token
31
32 def update(self, loc, newkey, newval):
33     """Update the key and value for the entry identified by Locator loc."""
34     j = loc._index
35     if not (0 <= j < len(self) and self._data[j] is loc):
36         raise ValueError('Invalid locator')
37     loc._key = newkey
38     loc._value = newval
39     self._bubble(j)
40
41 def remove(self, loc):
42     """Remove and return the (k,v) pair identified by Locator loc."""
43     j = loc._index
44     if not (0 <= j < len(self) and self._data[j] is loc):
45         raise ValueError('Invalid locator')

```

```

46     if j == len(self) - 1:           # item at last position
47         self._data.pop()            # just remove it
48     else:
49         self._swap(j, len(self)-1)  # swap item to the last position
50         self._data.pop()            # remove it from the list
51         self._bubble(j)           # fix item displaced by the swap
52     return (loc._key, loc._value)

```

表 9-4 一个用基于数组堆表示实现的大小为 n 的可适应性优先级队列 P 的各方法运行时间表。空间需求量是 $O(n)$

操作	运行时间
$P.\text{len}()$, $P.\text{is_empty}()$, $P.\text{min}()$	$O(1)$
$P.\text{add}(k, v)$	$O(\log n)^*$
$P.\text{update}(\text{loc}, k, v)$	$O(\log n)$
$P.\text{remove}(\text{loc})$	$O(\log n)^*$
$P.\text{remove_min}()$	$O(\log n)^*$

* 动态数组摊销

9.6 练习

请访问 www.wiley.com/college/goodrich 以获得练习帮助。

巩固

- R-9.1 使用 `remove_min` 操作从含有 n 个元组的堆中删除第 $\lceil \log n \rceil$ 最小元组需要花费多长时间?
- R-9.2 假设使用等于先序排序的键值来标识二叉树 T 的每个位置 p , 在什么情况下 T 是堆?
- R-9.3 在下列优先级队列 AD, T 方法中, 每次调用 `remove_min` 将返回什么? 这些函数是: `add(5, A)`、`add(4, B)`、`add(7, F)`、`add(1, D)`、`remove_min()`、`add(3, J)`、`add(6, L)`、`remove_min()`、`remove_min()`、`add(8, G)`、`remove_min()`、`add(2, H)`、`remove_min()` 和 `remove_min()`。
- R-9.4 某机场正在开发一个空中交通管制模拟系统, 该系统用于处理诸如飞机着陆和起飞等事件, 每个事件有一个标记事件什么时候发生的时间戳。模拟程序需要能够有效地处理如下两个基本操作:
- 插入一个带有给定时间戳的事件 (即增加一个未来的事件)。
 - 取出拥有最短时间戳的事件 (即确定下一个处理的事件)。
- 哪种数据结构适合处理上述操作? 为什么?
- R-9.5 如表 9-2 所示, `UnsoredPriorityQueue` 类的方法 `min` 的时间复杂度为 $O(n)$ 。试简单修改该类, 使 `min` 的时间复杂度变为 $O(1)$ 。请解释对于类的其他方法需要做哪些必要的改动。
- R-9.6 能否通过调整上一问题的解决方案, 使 `UnsoredPriorityQueue` 类中的 `remove_min` 方法时间复杂度也为 $O(1)$? 简单描述如何调整。
- R-9.7 试描述在输入序列 (22, 15, 36, 44, 10, 3, 9, 13, 29, 25) 上执行选择排序算法的过程。
- R-9.8 试说明在上一问题中的输入序列上执行插入排序算法的过程。
- R-9.9 请给出一个会出现插入排序最坏情况的含 n 个元组的序列的例子, 并证明在这样的序列上执行插入排序的时间复杂度为 $\Omega(n^2)$ 。
- R-9.10 堆中哪个位置可能存储着第三小的键?
- R-9.11 堆中哪个位置可能存储最大键?

- R-9.12 考虑这样的情况，用户有数值型的键并希望有一个面向最大值导向的优先级队列。如何用一个标准（面向最小值）的优先级队列实现这一目的？
- R-9.13 试说明在输入序列（2, 5, 16, 4, 10, 23, 39, 18, 26, 15）上执行原地堆排序算法的过程。
- R-9.14 假设 T 为完全二叉树，位置 p 存储以 $f(p)$ 为关键字的元组， $f(p)$ 是 p 的层次编号（见 8.3.2 节）。请问该树 T 是堆吗？为什么？
- R-9.15 试解释为什么堆向下冒泡算法的描述中不考虑位置 p 有右孩子但是没有左孩子的情况。
- R-9.16 是否存在一个含有 7 个元组且键值唯一的堆 H ，这个堆 H 可以根据先序遍历生成按键值递增或递减排序的序列？中序遍历呢？后序遍历呢？如果存在，请给出一个例子；如果不存在，请说明原因。
- R-9.17 假设 H 是一个基于数组表示的完全二叉树的堆，堆中存储了 15 个元组。以先序遍历 H 的数组标识序列是什么？中序遍历 H 呢？后续遍历 H 呢？
- R-9.18 证明在一个堆排序中的和 $\sum_{i=1}^n \log i$ 的复杂度是 $\Omega(n \log n)$ 。
- R-9.19 Bill 认为一个堆的先序遍历将以非降序的顺序列出它所有元组的键。画图给出一个例子，证明他是错误的。
- R-9.20 Hillary 认为一个堆的后序遍历将以非升序的顺序列出它的键。请给出一个例子，证明她是错误的。
- R-9.21 试给出从图 9-1 的堆中删除元组（16, X）算法的所有步骤，假设该元组已经由一个定位器标识。
- R-9.22 试给出在图 9-1 的堆中，用 18 替换元组（5, A）的键的算法的所有步骤，假设该元组已经由定位器标识。
- R-9.23 假设一个堆的所有元组均是 1 ~ 59（不重复）的奇数，画出插入键值为 32 的元组时所引起的自底向上到根节点的孩子节点（用 32 替换这个孩子节点的键值）的堆向上冒泡的过程。
- R-9.24 描述向堆中插入 n 个节点的序列需要在 $O(n \log n)$ 时间内处理。
- R-9.25 写出对图 9-9 中的堆原地堆排序算法的所有步骤。给出在每一步结束时数组和相关的堆的状态。

创新

- R-9.26 如何能仅使用一个优先级队列和一个额外的整型实例变量来实现堆栈 ADT？请给出方法。
- R-9.27 如何仅使用一个优先级队列和一个额外的整型实例变量来实现 FIFO 队列 ADT？请给出方法。
- R-9.28 对于以上问题，Idle 教授建议使用如下解决方案：在一个元组被插入队列的时候，给它分配一个等于当前队列长度的键值。这样的策略能产生 FIFO 语义吗？证明这个方法是可行的，或者提供一个反例来否定这个方法。
- R-9.29 使用一个 Python 列表重新实现 SortedPriorityQueue。确保维持 `remove_min` 的时间复杂度为 $O(1)$ 。
- R-9.30 给出类 `HeapPriorityQueue` 中 `_upheap` 方法的一个非递归的实现。
- R-9.31 给出类 `HeapPriorityQueue` 中 `_downheap` 方法的一个非递归的实现。
- R-9.32 假设使用完全二叉树 T 的链表示，并使用一个额外的指针指向树的最后一个节点。假定 n 是当前树的节点数，则在 `add` 和 `remove_min` 操作之后如何在 $O(\log n)$ 时间复杂度内更新指向最后节点的指针？就像在图 9-12 所描述的那样。请确保能够处理所有可能的情况。

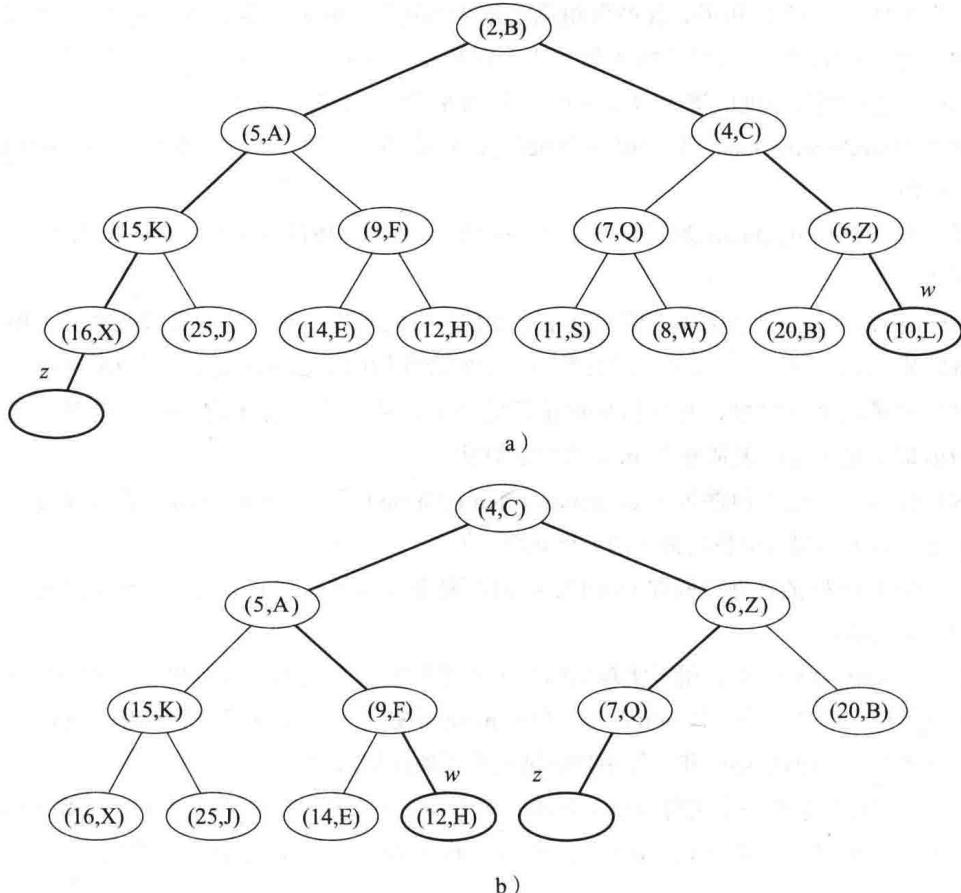


图 9-12 在 add 或 remove 操作之后，更新完全二叉树的最后一个节点。节点 w 是执行 add 操作前或 remove 操作后树中的最后一个节点。节点 z 是执行 add 操作后或 remove 操作前树中的最后一个节点。

- R-9.33 当使用基于链表的树表示堆时，在一个堆 T 的插入过程中找到最后一个节点的另一种方法是存储，在最后一个节点和 T 中的每个叶节点，指向叶节点的引用立即指向它的右边节点（包装下一层的第一个节点为最右叶节点）。假设用链表结构实现 T ，展示如何在每个优先级队列 ADT 操作中以 $O(1)$ 的时间复杂度维护这样的引用。
- R-9.34 我们能够通过二进制字符串的方法表示二叉树从根节点到给定节点的路径，在这个路径中 0 表示“沿左孩子走”，1 表示“沿右孩子走”。比如，在图 9-12a 的堆中从根节点到存储 $(8, W)$ 的节点的路径表示为“101”。基于以上表示，设计一个 $O(\log n)$ 时间复杂度的算法来寻找拥有 n 个节点的完全二叉树的最后一个节点。展示这种算法怎样能被用在通过链表结构实现且没有指向最后节点指针的完全二叉树中。
- R-9.35 给定一个堆 T 和一个键 k ，给出一个算法来计算在 T 中所有元组中有一个键值小于等于 k 。比如，给定图 9-12a 中的堆和请求键值 $k = 7$ ，该算法应该给出拥有键值 2、4、5、6 和 7 的元组（但不需要以这种顺序）。该算法应该运行在与返回元组数量成正比的时间内，并且不应该改变堆。
- R-9.36 请给出表 9-4 中的时间范围的一个理由。
- R-9.37 通过显示以下的总和是 $O(1)$ ，给出自下而上的堆结构的另一种分析，对于任何正整数 h ：

$$\sum_{i=1}^h (i / 2^i)$$

- R-9.38 假设两棵二叉树 T_1 和 T_2 , 保持元组满足堆序列属性(但是不需要满足完全二叉树属性)。描述一种连接 T_1 和 T_2 为二叉树 T 的方法, 它的节点为 T_1 和 T_2 中的节点并且满足堆顺序属性。你的算法时间复杂度应该为 $O(h_1 + h_2)$, h_1 和 h_2 为 T_1 和 T_2 的高度。
- R-9.39 对于 HeapPriorityQueue 类实现一个 heappushpop 方法, 并且与 9.3.7 节 heapq 模块的描述语义相似。
- R-9.40 对于 HeapPriorityQueue 类实现一个 heapreplace 方法, 并且与 9.3.7 节 heapq 模块的描述语义相似。
- R-9.41 Tamarindo 航空公司想给他们最高 $\log n$ 飞行频率的常客一张一流的升级优惠券, 根据里程数量的累积, n 为航空公司常客的总数量。他们现在用的算法时间复杂度为 $O(n \log n)$, 按飞行里程数量给常客排序, 并且扫描被排序的列表, 从中选出最高的 $\log n$ 个常客。描述一种在 $O(n)$ 时间复杂度内识别最高 $\log n$ 常客的算法。
- R-9.42 解释使用面向最大值的堆 (maximum-oriented heap) 在 $O(n + k \log n)$ 时间复杂度内从拥有 n 个元组的无序列表中找出最大的 k 个元组。
- R-9.43 解释使用 $O(k)$ 的辅助空间在 $O(n \log k)$ 时间复杂度内从拥有 n 个元组的无序列表中找出最大的 k 个元组。
- R-9.44 给定 PriorityQueue 类, 用于实现面向最小值优先级队列 ADT, 并提供一个 MaxPriorityQueue 类的实现, 它适合用方法 add、max 和 remove_max 来提供面向最大值抽象。你的实现不应该对原始类 PriorityQueue 和可能用到的键值类型做任何假设。
- R-9.45 写一个非负整数的一个关键函数, 该函数根据每个整数的二进制扩展中的 1 个数来确定顺序。
- R-9.46 给出在代码段 9-7 部分 pq_sort 函数的可替代实现, 该函数接受一个关键字函数作为可选参数。
- R-9.47 对于一个数组, 描述一个选择排序算法的原地版本, 并且该数组除了本身只是用 $O(1)$ 的实例变量空间。
- R-9.48 在数组 A 中, 假设排序问题的输入已给定, 试描述如何只用数组 A 和至多一个常数数量的附加变量来实现插入排序算法。
- R-9.49 使用标准的面向最小值优先级队列(代替面向最大值优先级队列)来给出原地堆排序算法一个可替代的描述。
- R-9.50 交易股票的网上计算机系统需要处理从“以 $\$x$ 每份价格买 100 份”到“以 $\$y$ 每份价格卖 100 份”的订单。买 $\$x$ 的订单只有在存在价格 $\$y$ 的卖订单并且 $y \leq x$ 时才会被处理。同样, 卖 $\$y$ 的订单只有在存在价格 $\$x$ 的买订单并且 $y \leq x$ 时才会被处理。如果一个购买或出售订单到来但不能被处理, 它必须等待一个未来的允许它进行处理的订单。请描述一种方案, 允许买或卖订单以 $O(\log n)$ 的时间进入系统, 与它们是否被立即处理无关。
- R-9.51 针对之前的问题扩展一个解决方案, 以便让用户可以更新他们的购买或出售的还没有被处理的订单价格。
- R-9.52 一群孩子想要玩一个被称作反垄断的游戏, 在该游戏中, 每回拥有最多钱的玩家必须把他 / 她一半的钱给拥有最少钱的玩家。什么数据结构能被用来高效地玩这种游戏? 为什么?
- 项目**
- P-9.53 实现原地堆排序算法, 并通过实验与非原地的标准堆排序算法的运行时间做比较。
- P-9.54 使用练习 C-9.42 或 C-9.43 的方法重新实现 7.6.2 节的 FavoritesListMTF 类的 top 方法。确保结果从最小到最大生成。

- P-9.55 编写一个程序，就像在练习 C-9.50 中描述的可以处理一系列股票买卖订单。
- P-9.56 S 表示在平面上拥有不同整数 x 和 y 坐标的 n 个点的集合。用 T 表示存储 S 外部节点中的点的完全二叉树。以便于这些点能够以 X 坐标增加的方式从左到右排序。对于每一个在 T 中的节点 v ，用 $S(v)$ 表示包含存储以 v 为根的子树的点。对于 T 的根 r ，定义 $\text{top}(r)$ 是在拥有最大 y 坐标的 $S = S(r)$ 上的点。对于每一个其他节点 v ，定义 $\text{top}(r)$ 为在 $S(v)$ 中拥有最高 y 坐标以及不在 $S(u)$ 中拥有最高的 y 坐标的点，在 T 中 u 是 v 的父节点（如果这样的节点存在）。如此标记使 T 成为一个优先级搜索树。请针对把 T 变成一个优先级搜索树描述一个线性算法，并实现这个方法。
- P-9.57 优先级队列的一个主要应用是操作系统——在 CPU 上的调度工作。在这个项目中，你将创建一个类似于 CPU 调度工作的程序。你的程序应该运行在一个循环中，它的每一次遍历相当于 CPU 的一个时间片。每个工作被设定一个优先级，它是 -20 （最高优先级）~ 19 （最低优先级）的整数，从在一个时间片等待被执行的所有工作中，CPU 必须调用拥有最高优先级的工作。在这个模拟中，每个工作也将包含一个长度值，它是 $1 \sim 100$ 的一个整数值，表示处理这个工作需要的时间片数。为了简单，你可以假设工作不能被打断——一旦它被 CPU 调用，一个工作运行需要等于它长度的时间片。模拟必须在每个被调用的时间片输出运行在 CPU 上的工作的名字，并且必须处理一个命令序列，每个时间片一个，每一个命令由“增加长度 n 的工作的名字和优先级 p ”或“在这个时间片没有新的工作”组成。
- P-9.58 开发一个可适应优先级队列的 Python 实现，该队列基于一个未排序的列表，并且支持位置感知元组。

扩展阅读

Knuth 关于排序和搜索的书^[65]描述了选择排序、插入排序和堆排序算法的动机和历史。堆排序算法由 Willoams^[103]完成，线性时间堆构造算法由 Floyd^[39]完成。堆和堆排序变化更多的算法和分析参见 Bentley^[15]、Carlsson^[24]、Gonnet 和 Munro^[45]、McDiarmid 和 Reed^[74]以及 Schaffer 和 Sedgewick^[88]所撰写的论文。