heapq --- 堆队列算法

源码： Lib/heapq.py

这个模块提供了堆队列算法的实现，也称为优先队列算法。

堆是一个二叉树，它的每个父节点的值都只会小于或等于所有孩子节点（的值）。 它使用了数组来实现：从零开始计数，对于所有的 *k*，都有 `heap[k] <= heap[2*k+1]` 和 `heap[k] <= heap[2*k+2]`。 为了便于比较，不存在的元素被认为是无限大。 堆最有趣的特性在于最小的元素总是在根结点：`heap[0]`。

这个API与教材的堆算法实现有所不同，具体区别有两方面： （a）我们使用了从零开始的索引。这使得节点和其孩子节点索引之间的关系不太直观但更加适合，因为 Python 使用从零开始的索引。 （b）我们的 pop 方法返回最小的项而不是最大的项（这在教材中称为“最小堆”；而“最大堆”在教材中更为常见，因为它更适用于原地排序）。

基于这两方面，把堆看作原生的Python list也没什么奇怪的： `heap[0]` 表示最小的元素，同时 `heap.sort()` 维护了堆的不变性！

要创建一个堆，可以使用list来初始化为 `[]`，或者你可以通过一个函数 `heapify()`，来把一个list转换成堆。

定义了以下函数：

heapq.**heappush**(*heap*, *item*)

将 *item* 的值加入 *heap* 中，保持堆的不变性。

heapq.**heappop**(*heap*)

弹出并返回 *heap* 的最小的元素，保持堆的不变性。如果堆为空，抛出 `IndexError`。使用 `heap[0]`，可以只访问最小的元素而不弹出它。

heapq.**heappushpop**(*heap*, *item*)

将 *item* 放入堆中，然后弹出并返回 *heap* 的最小元素。该组合操作比先调用 `heappush()` 再调用 `heappop()` 运行起来更有效率。

heapq.**heapify**(*x*)

将list *x* 转换成堆，原地，线性时间内。

heapq.**heapreplace**(*heap*, *item*)

弹出并返回 *heap* 中最小的一项，同时推入新的 *item*。 堆的大小不变。 如果堆为空则引发 `IndexError`。

这个单步骤操作比 `heappop()` 加 `heappush()` 更高效，并且在使用固定大小的堆时更为适宜。 pop/push 组合总是会从堆中返回一个元素并将其替换为 *item*。

返回的值可能会比添加的 *item* 更大。 如果不希望如此，可考虑改用 `heappushpop()`。 它的 push/pop 组合会返回两个值中较小的一个，将较大的值留在堆中。

该模块还提供了三个基于堆的通用功能函数。

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)

将多个已排序的输入合并为一个已排序的输出（例如，合并来自多个日志文件的带时间戳的条目）。 返回已排序值的 iterator。

类似于 `sorted(itertools.chain(*iterables))` 但返回一个可迭代对象，不会一次性地将数据全部放入内存，并假定每个输入流都是已排序的（从小到大）。

key 指定带有单个参数的 key function，用于从每个输入元素中提取比较键。 默认值为 None (直接比较元素)。

reverse 为一个布尔值。 如果设为 `True`，则输入元素将按比较结果逆序进行合并。 要达成与 `sorted(itertools.chain(*iterables), reverse=True)` 类似的行为，所有可迭代对象必须是已从大到小排序的。

在 *3.5* 版更改: 添加了可选的 *key* 和 *reverse* 形参。

heapq.**nlargest**(n, iterable, key=None)

从 *iterable* 所定义的数据集中返回前 *n* 个最大元素组成的列表。 如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。 等价于: `sorted(iterable, key=key, reverse=True)[:n]`。

heapq.**nsmallest**(n, iterable, key=None)

从 *iterable* 所定义的数据集中返回前 *n* 个最小元素组成的列表。 如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。 等价于: `sorted(iterable, key=key)[:n]`。

后两个函数在 *n* 值较小时性能最好。 对于更大的值，使用 `sorted()` 函数会更有效率。 此外，当 `n==1` 时，使用内置的 `min()` 和 `max()` 函数会更有效率。 如果需要重复使用这些函数，请考虑将可迭代对象转为真正的堆。

基本示例

堆排序 可以通过将所有值推入堆中然后每次弹出一个最小值项来实现。

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这类似于 `sorted(iterable)`，但与 `sorted()` 不同的是这个实现是不稳定的。

堆元素可以为元组。 这适用于将比较值（例如任务优先级）与跟踪的主记录进行赋值的场合:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

优先队列实现说明

优先队列 是堆的常用场合，并且它的实现包含了多个挑战:

- 排序稳定性：你该如何令相同优先级的两个任务按它们最初被加入时的顺序返回?
- 如果优先级相同且任务没有默认比较顺序，则 (priority, task) 对的元组比较将会中断。
- 如果任务优先级发生改变，你该如何将其移至堆中的新位置?
- 或者如果一个挂起的任务需要被删除，你该如何找到它并将其移出队列?

的，元组比较将永远不会直接比较两个任务。

不可比较任务问题的另一种解决方案是创建一个忽略任务条目并且只比较优先级字段的包装器类:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

其余的挑战主要包括找到挂起的任务并修改其优先级或将其完全移除。 找到一个任务可使用一个指向队列中条目的字典来实现。

移除条目或改变其优先级的操作实现起来更为困难，因为它会破坏堆结构不变量。 因此，一种可能的解决方案是将条目标记为已移除，再添加一个改变了优先级的新条目:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

理论

堆是通过数组来实现的，其中的元素从 0 开始计数，对于所有的 *k* 都有 `a[k] <= a[2*k+1]` 且 `a[k] <= a[2*k+2]`。 为了便于比较，不存在的元素被视为无穷大。 堆最有趣的特性在于 `a[0]` 总是其中最小的元素。

上面的特殊不变量是用来作为一场锦标赛的高效内存表示。 下面的数字是 *k* 而不是 `a[k]`:

```
                               0

             1                                 2

      3             4                 5               6
```

15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

在上面的树中，每个 *k* 单元都位于 `2*k+1` 和 `2*k+2` 之上。 体育运动中我们经常见到二元锦标赛模式，每个胜者单元都位于另两个单元之上，并且我们可以沿着树形图向下追溯胜者所遇到的所有对手。 但是，在许多采用这种锦标赛模式的计算机应用程序中，我们并不需要追溯胜者的历史。 为了获得更高的内存利用效率，当一个胜者晋级时，我们会用较低层级的另一条目来替代它，因此规则变为一个单元和它之下的两个单元包含三个不同条目，上方单元“胜过”了两个下方单元。

如果此堆的不变量始终受到保护，则序号 0 显然是最后的赢家。 删除它并找出“下一个”赢家的最简单算法方式是家某个输家（让我们假定是上图中的 30 号单元）移至 0 号位置，然后将这个新的 0 号沿树下行，不断进行值的交换，直到不变量重新建立。 这显然会是树中条目总数的对数。 通过迭代所有条目，你将得到一个 O(n log n) 复杂度的排序。

此排序有一个很好的特性就是你可以在排序进行期间高效地插入新条目，前提是插入的条目不比你最近取出的 0 号元素“更好”。 这在模拟上下文时特别有用，在这种情况下树保存的是所有传入事件，“胜出”条件是最小调度时间。 当一个事件将其他事件排入执行计划时，它们的调试时间向未来方向延长，这样它们可方便地入堆。 因此，堆结构很适宜用来实现调度器，我的 MIDI 音序器就是用的这个 :-)。

用于实现调度器的各种结构都得到了充分的研究，堆是非常适宜的一种，因为它们的速度相当快，并且几乎是恒定的，最坏的情况与平均情况没有太大差别。 虽然还存在其他总体而言更高效的实现方式，但其最坏的情况却可能非常糟糕。

堆在大磁盘排序中也非常有用。 你应该已经了解大规模排序会有多个“运行轮次”（即预排序的序列，其大小通常与 CPU 内存容量相关），随后这些轮次会进入合并通道，轮次合并的组织往往非常巧妙 [1]。 非常重要的一点是初始排序应产生尽可能长的运行轮次。 锦标赛模式是达成此目标的好办法。 如果你使用全部有用内存来进行锦标赛，替换和安排恰好适合当前运行轮次的条目，你将可以对于随机输入生成两倍于内存大小的运行轮次，对于模糊排序的输入还会有更好的效果。

另外，如果你输出磁盘上的第 0 个条目并获得一个可能不适合当前锦标赛的输入（因为其值要“胜过”上一个输出值），它无法被放入堆中，因此堆的尺寸将缩小。 被释放的内存可以被巧妙地立即重用以逐步构建第二个堆，其增长速度与第一个堆的缩减速度正好相同。 当第一个堆完全消失时，你可以切换新堆并启动新的运行轮次。 这样做既聪明又高效！

总之，堆是值得了解的有用内存结构。 我在一些应用中用到了它们，并且认为保留一个 'heap' 模块是很有意义的。 :-)

备注

[1] 当前时代的磁盘平衡算法与其说是巧妙，不如说是麻烦，这是由磁盘的寻址能力导致的结果。 在无法寻址的设备例如大型磁带机上，情况则相当不同，开发者必须非常聪明地（极为提前地）确保每次磁带转动都尽可能地高效（就是说能够最好地加入到合并“进程”中）。 有些磁带甚至能够反向读取，这也被用来避免倒带的耗时。 请相信我，真正优秀的磁带机排序看起来是极其壮观的，排序从来都是一门伟大的艺术！ :-)