9.4. heapq — Heap queue algorithm¶

New in version 2.3.

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k*, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `heap[0]` is always its smallest element.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

To create a heap, use a list initialized to `[]`, or you can transform a populated list into a heap via function `heapify()`.

The following functions are provided:

`heapq.heappush`(*heap*, *item*)¶
Push the value *item* onto the *heap*, maintaining the heap invariant.

`heapq.heappop`(*heap*)¶
Pop and return the smallest item from the *heap*, maintaining the heap invariant. If the heap is empty, `IndexError` is raised.

`heapq.heappushpop`(*heap*, *item*)¶

Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than `heappush()` followed by a separate call to `heappop()`.

New in version 2.6.

`heapq.heapify`(*x*)¶
Transform list *x* into a heap, in-place, in linear time.

`heapq.heapreplace`(*heap*, *item*)¶

Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn't change. If the heap is empty, `IndexError` is raised. This is more efficient than `heappop()` followed by `heappush()`, and can be more appropriate when using a fixed-size heap. Note that the value returned may be larger than *item*! That constrains reasonable uses of this routine unless written as part of a conditional replacement:

```
if item > heap[0]:
   item = heapreplace(heap, item)
```

Example of use:

```
>>> from heapq import heappush, heappop
>>> heap = []
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> for item in data:
...     heappush(heap, item)
...
>>> ordered = []
>>> while heap:
...     ordered.append(heappop(heap))
...
>>> print ordered
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> data.sort()
>>> print data == ordered
True
```

Using a heap to insert items at the correct place in a priority queue:

```
>>> heap = []
>>> data = [(1, 'J'), (4, 'N'), (3, 'H'), (2, 'O')]
>>> for item in data:
...     heappush(heap, item)
...
>>> while heap:
...     print heappop(heap)[1]
J
O
H
N
```

The module also offers three general purpose functions based on heaps.

`heapq.merge`(**iterables*)¶

Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an *iterator* over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

New in version 2.6.

`heapq.nlargest`(*n*, *iterable*[, *key*])¶

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: `key=str.lower` Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`

New in version 2.4.

Changed in version 2.5: Added the optional *key* argument.

`heapq.nsmallest`(*n*, *iterable*[, *key*])¶

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: `key=str.lower` Equivalent to: `sorted(iterable, key=key)[:n]`

New in version 2.4.

Changed in version 2.5: Added the optional *key* argument.

The latter two functions perform best for smaller values of *n*. For larger values, it is more efficient to use the `sorted()` function. Also, when `n==1`, it is more efficient to use the built-in `min()` and `max()` functions.

9.4.1. Theory¶

(This explanation is due to François Pinard. The Python code for this module was contributed by Kevin O'Connor.)

Heaps are arrays for which `a[k] <= a[2*k+1]` and `a[k] <= a[2*k+2]` for all *k*, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `a[0]` is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are *k*, not `a[k]`:

```
                               0

             1                                 2

     3               4                 5               6

 7       8       9       10      11      12      13      14

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

In the tree above, each cell *k* is topping `2*k+1` and `2*k+2`. In an usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the “next” winner is to move some loser (let’s say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an O(n log n) sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not “better” than the last 0’th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the “win” condition means the smallest scheduled time. When an event schedule other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing “runs” (which are pre-sorted sequences, which size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised [1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you’ll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0’th item on disk and get an input which may not fit in the current tournament (because the value “wins” over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a ‘heap’ module around. :-)

Footnotes

[1] The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at “progressing” the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

Table Of Contents

Previous topic

Next topic

This Page

- Show Source

Navigation