heapq – Heap Sort Algorithm

Purpose: The heapq implements a min-heap sort algorithm suitable for use with Python’s lists.

A *heap* is a tree-like data structure in which the child nodes have a sort-order relationship with the parents. *Binary heaps* can be represented using a list or array organized so that the children of element N are at positions 2 * *N* + 1 and 2 * *N* + 2 (for zero-based indexes). This layout makes it possible to rearrange heaps in place, so it is not necessary to reallocate as much memory when adding or removing items.

A max-heap ensures that the parent is larger than or equal to both of its children. A min-heap requires that the parent be less than or equal to its children. Python’s heapq module implements a min-heap.

Example Data

The examples in this section use the data in heapq_heapdata.py.

```
# heapq_heapdata.py

# This data was generated with the random module.

data = [19, 9, 4, 10, 11]
```

The heap output is printed using heapq_showtree.py.

```
# heapq_showtree.py

import math
from io import StringIO


def show_tree(tree, total_width=36, fill=' '):
    """Pretty-print a tree."""
    output = StringIO()
    last_row = -1
    for i, n in enumerate(tree):
        if i:
            row = int(math.floor(math.log(i + 1, 2)))
        else:
            row = 0
        if row != last_row:
            output.write('\n')
        columns = 2 ** row
        col_width = int(math.floor(total_width / columns))
        output.write(str(n).center(col_width, fill))
        last_row = row
    print(output.getvalue())
    print('-' * total_width)
    print()
```

Creating a Heap

There are two basic ways to create a heap: heappush() and heapify().

```
# heapq_heappush.py

import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

heap = []
print('random :', data)
print()
print()

for n in data:
    print('add {:>3}:'.format(n))
    heapq.heappush(heap, n)
    show_tree(heap)
```

When heappush() is used, the heap sort order of the elements is maintained as new items are added from a data source.

```
$ python3 heapq_heappush.py

random : [19, 9, 4, 10, 11]

add  19:

                 19
------------------------------------

add   9:

                 9
        19
------------------------------------

add   4:

                 4
        19                9
------------------------------------

add  10:

                 4
        10                9
    19
------------------------------------

add  11:

                 4
        10                9
    19       11
------------------------------------
```

If the data is already in memory, it is more efficient to use heapify() to rearrange the items of the list in place.

```
# heapq_heapify.py

import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print('random    :', data)
heapq.heapify(data)
print('heapified :')
show_tree(data)
```

The result of building a list in heap order one item at a time is the same as building an unordered list and then calling heapify().

```
$ python3 heapq_heapify.py

random    : [19, 9, 4, 10, 11]
heapified :

                 4
        9                 19
    10       11
------------------------------------
```

Accessing the Contents of a Heap

Once the heap is organized correctly, use heappop() to remove the element with the lowest value.

```
# heapq_heappop.py

import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print('random    :', data)
heapq.heapify(data)
print('heapified :')
show_tree(data)
print()

for i in range(2):
    smallest = heapq.heappop(data)
    print('pop    {:>3}:'.format(smallest))
    show_tree(data)
```

In this example, adapted from the stdlib documentation, heapify() and heappop() are used to sort a list of numbers.

```
$ python3 heapq_heappop.py

random    : [19, 9, 4, 10, 11]
heapified :

                 4
        9                 19
    10       11
------------------------------------


pop      4:

                 9
        10                19
    11
------------------------------------

pop      9:

                 10
        11                19
------------------------------------
```

To remove existing elements and replace them with new values in a single operation, use heapreplace().

```
# heapq_heapreplace.py

import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

heapq.heapify(data)
print('start:')
show_tree(data)

for n in [0, 13]:
    smallest = heapq.heapreplace(data, n)
    print('replace {:>2} with {:>2}:'.format(smallest, n))
    show_tree(data)
```

Replacing elements in place makes it possible to maintain a fixed-size heap, such as a queue of jobs ordered by priority.

```
$ python3 heapq_heapreplace.py

start:
```

```
start:

                 4
        9                 19
    10       11
------------------------------------

replace  4 with  0:

                 0
        9                 19
    10       11
------------------------------------

replace  0 with 13:

                 9
        10                19
    13       11
------------------------------------
```

Data Extremes from a Heap

heapq also includes two functions to examine an iterable and find a range of the largest or smallest values it contains.

```
# heapq_extremes.py

import heapq
from heapq_heapdata import data

print('all       :', data)
print('3 largest :', heapq.nlargest(3, data))
print('from sort :', list(reversed(sorted(data)[-3:])))
print('3 smallest:', heapq.nsmallest(3, data))
print('from sort :', sorted(data)[:3])
```

Using `nlargest()` and `nsmallest()` is efficient only for relatively small values of *n* > 1, but can still come in handy in a few cases.

```
$ python3 heapq_extremes.py

all       : [19, 9, 4, 10, 11]
3 largest : [19, 11, 10]
from sort : [19, 11, 10]
3 smallest: [4, 9, 10]
from sort : [4, 9, 10]
```

Efficiently Merging Sorted Sequences

Combining several sorted sequences into one new sequence is easy for small data sets.

```
list(sorted(itertools.chain(*data)))
```

For larger data sets, this technique can use a considerable amount of memory. Instead of sorting the entire combined sequence, `merge()` uses a heap to generate a new sequence one item at a time, determining the next item using a fixed amount of memory.

```
# heapq_merge.py

import heapq
import random


random.seed(2016)

data = []
for i in range(4):
    new_data = list(random.sample(range(1, 101), 5))
    new_data.sort()
    data.append(new_data)

for i, d in enumerate(data):
    print('{}: {}'.format(i, d))

print('\nMerged:')
for i in heapq.merge(*data):
    print(i, end=' ')
print()
```

Because the implementation of `merge()` uses a heap, it consumes memory based on the number of sequences being merged, rather than the number of items in those sequences.

```
$ python3 heapq_merge.py

0: [33, 58, 71, 88, 95]
1: [10, 11, 17, 38, 91]
2: [13, 18, 39, 61, 63]
3: [20, 27, 31, 42, 45]

Merged:
10 11 13 17 18 20 27 31 33 38 39 42 45 58 61 63 71 88 91 95
```

See also

- Standard library documentation for heapq
- Wikipedia: Heap (data structure) – A general description of heap data structures.
- Priority Queue – A priority queue implementation from Queue in the standard library.

array — Sequence of Fixed-type Data

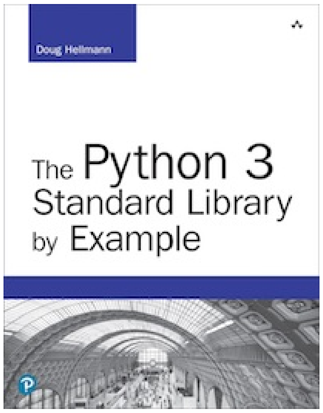
bisect — Maintain Lists in Sorted Order

Quick Links

This page was last updated 2017-07-30.

Navigation

array — Sequence of Fixed-type Data
bisect — Maintain Lists in Sorted Order



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

- Module Index
- Index

Other Writing

- Blog
- The Python Standard Library By Example