1 heappush(heap,item)建立大、小根堆

heapq.heappush()是往堆中添加新值，此时自动建立了小根堆，代码如下

import heapq

a = []   #创建一个空堆

heapq.heappush(a,18)

heapq.heappush(a,1)

heapq.heappush(a,20)

heapq.heappush(a,10)

heapq.heappush(a,5)

heapq.heappush(a,200)

print(a)

    1

    2

    3

    4

    5

    6

    7

    8

    9

输出

[1, 5, 20, 18, 10, 200]

    1

但heapq里面没有直接提供建立大根堆的方法，可以采取如下方法：每次push时给元素加一个负号（即取相反数），此时最小值变最大值，反之亦然，那么实际上的最大值就可以处于堆顶了，返回时再取负即可。

a = []

for i in [1, 5, 20, 18, 10, 200]:

    heapq.heappush(a,-i)

print(list(map(lambda x:-x,a)))

    1

    2

    3

    4

输出

[200, 18, 20, 1, 10, 5]

    1

2 heapify(heap)建立大、小根堆

heapq.heapfy()是以线性时间讲一个列表转化为小根堆

a = [1, 5, 20, 18, 10, 200]

heapq.heapify(a)

print(a)

    1

    2

    3

输出

[1, 5, 20, 18, 10, 200]

    1

用第一节中同样的方法建立大根堆：

a = [1, 5, 20, 18, 10, 200]

a = list(map(lambda x:-x,a))

heapq.heapify(a)

print([-x for x in a])

    1

    2

    3

    4

输出

200, 18, 20, 5, 10, 1]

    1

与第一节所得大根堆相比，虽然二叉树的第三层元素顺序不同，但都符合大根堆的定义

3 heappop(heap)

heapq.heappop()是从堆中弹出并返回最小的值

3.1 利用heappop进行堆排序

堆排序当然也要利用到heappush或者heapify，可参考排序算法总结中的堆排序，这里只贴代码

import heapq

def heap\_sort(arr):

    if not arr:

        return []

    h = []  #建立空堆

    for i in arr:

        heapq.heappush(h,i) #heappush自动建立小根堆

    return [heapq.heappop(h) for i in range(len(h))]  #heappop每次删除并返回列表中最小的值

    1

    2

    3

    4

    5

    6

    7

    8

若是从大到小排列，有两种方法：

1）先建立小根堆，然后每次heappop()，此时得到从小大的排列，再reverse

2）利用相反数建立大根堆，然后heappop(-元素)。即push(-元素)，pop(-元素)

3.2 普通list的heapop()

普通list（即并没有进行heapify等操作的list），对他进行heappop操作并不会弹出list中最小的值，而是弹出第一个值

>>> a=[3,6,1]

>>> heapify(a)                  #将a变成堆之后，可以对其操作

>>> heappop(a)

1

>>> b=[4,2,5]                   #b不是堆，如果对其进行操作，显示结果如下

>>> heappop(b)                  #按照顺序，删除第一个数值并返回,不会从中挑选出最小的

4

>>> heapify(b)                  #变成堆之后，再操作

>>> heappop(b)

2

    1

    2

    3

    4

    5

    6

    7

    8

    9

    10

4 heappushpop(heap,item)

heapq.heappushpop()是heappush和haeppop的结合，同时完成两者的功能，先进行heappush()，再进行heappop()

>>>h =  [1, 2, 9, 5]

>>> heappop(h)

1

>>> heappushpop(h,4)            #增加4同时删除最小值2并返回该最小值，与下列操作等同：

2

>>> h

[4, 5, 9]

    1

    2

    3

    4

    5

    6

    7

5 heapreplace(heap.item)弹出并返回 heap 中最小的一项，同时推入新的 item

heapq.heapreplace()与heapq.heappushpop()相反，先进行heappop()，再进行heappush()

堆的大小不变。 如果堆为空则引发 IndexError。这个单步骤操作比依次执行heappop() + 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。

>>> a=[]

>>> heapreplace(a,3)            #如果list空，则报错

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: index out of range

>>> heappush(a,3)

>>> a

[3]

>>> heapreplace(a,2)            #先执行删除（heappop(a)->3),再执行加入（heappush(a,2))

3

>>> a

[2]

>>> heappush(a,5)

>>> heappush(a,9)

>>> heappush(a,4)

>>> a

[2, 4, 9, 5]

>>> heapreplace(a,6)            #先从堆a中找出最小值并返回，然后加入6

2

>>> a

[4, 5, 9, 6]

>>> heapreplace(a,1)            #1是后来加入的，在1加入之前，a中的最小值是4

4

>>> a

[1, 5, 9, 6]

    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

6 merge(\*iterables)

heapq.merge()合并多个堆然后输出

输入的list无序，merge后无序，若输入的list有序，merge后也有序

在这里插入图片描述

heapq.merge()的迭代性质意味着它对所有提供的序列都不会做一次性读取。这意味着可以利用它处理非常长的序列，而开销却非常小

7 nlargest(n , iterbale, key=None) 和nsmallest(n , iterbale, key=None)

获取列表中最大、最小的几个值，key的作用和sorted( )方法里面的key类似

>>>a = [0, 1, 2, 3, 4, 5, 5, 7, 8, 10, 15, 20, 25]

>>>heapq.nlargest(5,a)

[25, 20, 15, 10, 8]

>>>b = [('a',1),('b',2),('c',3),('d',4),('e',5)]

>>>heapq.nlargest(1,b,key=lambda x:x[1])

[('e', 5)]

    1

    2

    3

    4

    5

    6

    7

8 复杂度分析

8.1 各方法复杂度

1）heapq.heapify(x): O(n)

2）heapq.heappush(heap, item): O(logn)

3）heapq.heappop(heap): O(logn)

即插入或删除元素时，所有节点自动调整，保证堆的结构的复杂度为O(log n)

4）heapq.nlargest(k,iterable)和nsmallest(k,iterable)：O(n \* log(t))

8.2 关于排序和取TopN时各方法的快慢比较

在关于排序和取Top N值时，到底使用什么方法最快，python3 cookbook给出了非常好的建议：

1）当要查找的元素个数相对比较小的时候，函数nlargest() 和 nsmallest()。

2）仅仅想查找唯一的最小或最大(N=1)的元素的话，那么使用min()和max()函数。

3）如果N的大小和集合大小接近的时候，通常先排序这个集合然后再使用切片操作会更快点 (sorted(items)[:N] 或者是 sorted(items)[-N:])。