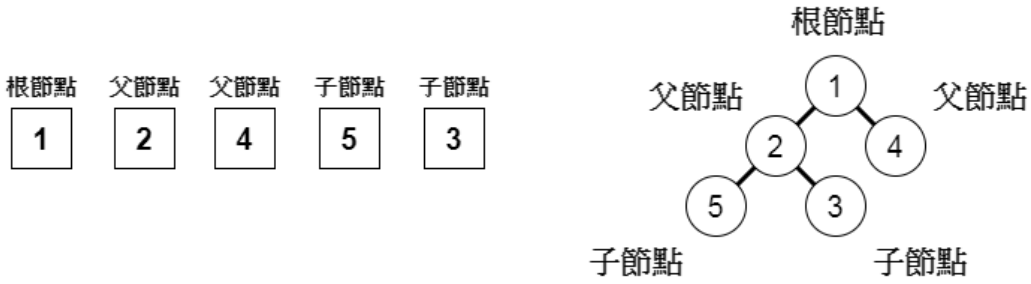
Heap Sort 流程圖、學習筆記、文字說明

一、流程圖

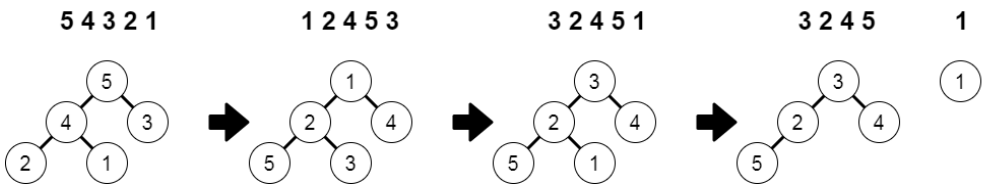
Heap Sort 的原理是先將數列排成堆積樹的形態，將位於樹根節點的數值與位於最後一個子節點的數值對調，再將對掉到最後一個子節點的數值剔除，作為新數列的第一順位，完成一次的 Heap Sort 排列。接著，因為原本堆積樹型態的數列已經剔除了位於最後一個節點的數值並且原本位於樹根的數值也被移往最後一個節點的位置，因此原本堆積樹的型態已被破壞掉，此時將位於數根的數值向下依序與其較小的子節點比較，當該數值大於子節點時，兩數值則互換，互換後原位於數根的數值再繼續與更下層較小的子節點比較，直到比到最後一個節點為止，全部比較完後再將位於樹根節點的數值與位於最後一個子節點的數值對調，並將對調到最後一個子節點的數值剔除，排進新數列的第二順位。以此類堆，直到堆積樹型態的數列的數值完全被剔除，都被排到新數列中。而此時的新數列將會是一個經大小排序過的數列。

以 Min Heap Tree 為例，堆積樹的型態分成根節點、父節點、子節點三個元素。根節點為該數列的最小值並且位於數量的首端，而每個父節點底下連接最多兩個，最少一個子節點，並且父節點的數值必須小於子節點的數值。下圖為堆積樹型態的數列與樹狀圖的是意圖。

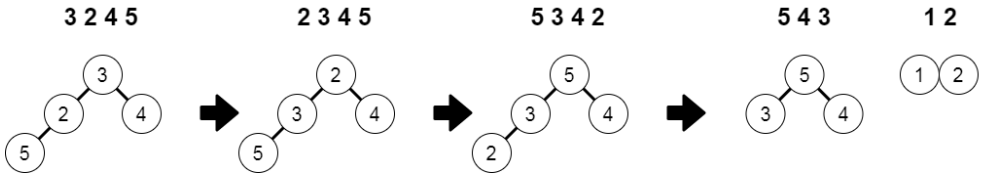



以下為 Min Heap Sort 所有的步驟:

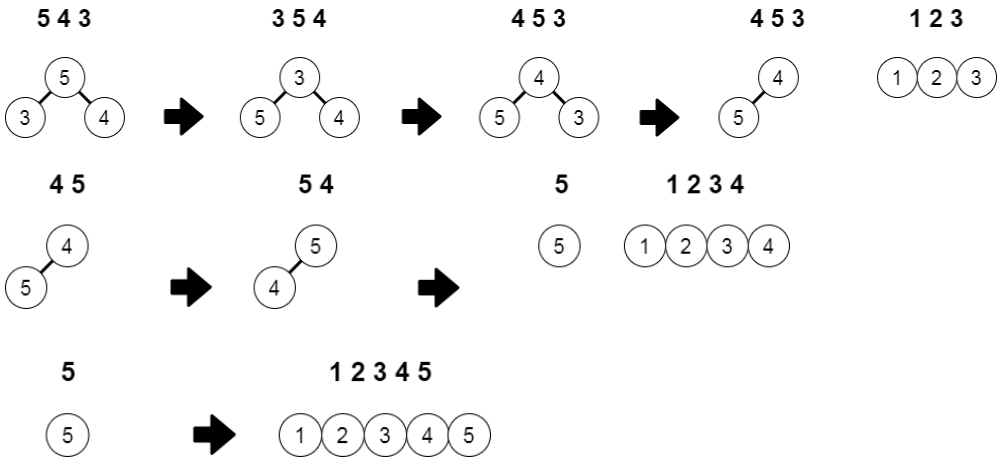
1. 將數列 5,4,3,2,1 排成堆積樹的形式；將位於樹根節點的數值 1 與位於最後一個子節點的數值 3 對調；將對掉到最後一個子節點的數值剔除，作為新數列的第一順位，完成一次的 Heap Sort 排列。



2. 接著，因為原本堆積樹型態的數列已經剔除了位於最後一個節點的數值並且原本位於樹根的數值也被移往最後一個節點的位置，因此原本堆積樹的型態已被破壞掉，此時將位於數根的數值向下依序與其子節點比較，當該數值大於子節點時，兩數值則互換，互換後原位於數根的數值再繼續與更下層的子節點比較，直到比到最後一個節點為止，全部比較完後再將位於樹根節點的數值與位於最後一個子節點的數值對調，並將對掉到最後一個子節點的數值剔除，排進新數列的第二順位。




3. 以此類堆，直到堆積樹型態的數列的數值完全被剔除，都被排到新數列中。而此時的新數列將會是一個經大小排序過的數列。

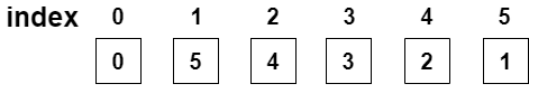


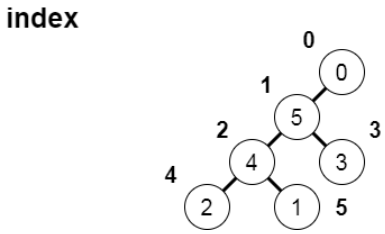
二、學習筆記

繳交的作業為原創的程式碼，無任何抄襲。 參考的程式碼僅作為延伸學習。對照參考程式碼和原創程式碼兩部分可以看出兩者的思考邏輯並不相同。以下的程式皆為 min heap sort。

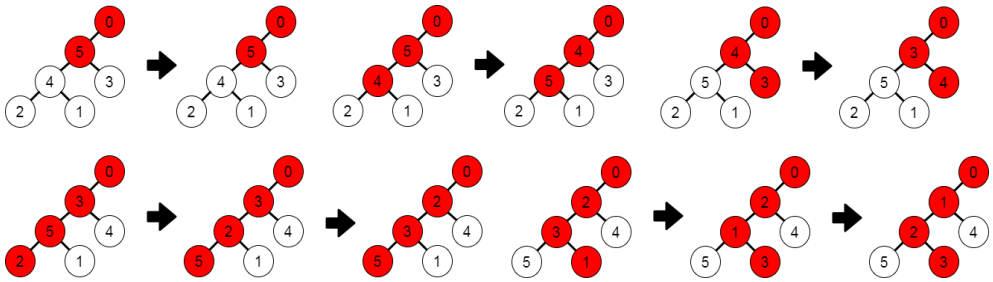
1. 原創程式的邏輯細節

程式的邏輯與流程圖介紹的原理相同。其中在設計上比較特別的是會在最一開始時增加一個數值在輸入的數列中的最前面，而該新增的數值會是原數列中最小的數值減掉 1，例如輸入 [5,4,3,2,1] 進入 heap_sort 函式進行排序，程式一開始會先在該陣列的開頭增加一個數值 0，變成 [0,5,4,3,2,1]。原因在於當數值最前面插入一個數值後其他所有的數值的 index 都會加 1，這時除了新的數值外所有的數值的 index 除 2 的商就是該數值的父節點的 index，如此就能方便呼叫父節點的數值；另外因為該新數值為數列中最小的數值，所以之後不管怎麼進行堆積樹排列，該數值都會在數列的最前面不動，因此之後都可以順利用 index 除 2 的商來呼叫。




另外在排序成堆積樹型態時，是從樹根由上往下排序，讓每個父節點都小於子節點，因此由上到下依序訪尋到每個節點時，都要從該節點開始由下至上走訪所有父節點，直到走訪到樹根節點的下一個節點為止，走訪時依序與父節點比較大小，如果走訪到的節點小於其父節點，該節點就與其父節點對調。



2. 不完整的程式碼

特別要注意的是，Heap Sort 只有從頭到尾搭建過一次 heap tree，第二次開始只是將位於樹根的數值向下與較小的子節點比較並交換已維持 heap tree 的型態。兩個步驟的差異在於時間複雜度的不同，從頭到尾搭建過一次 heap tree 的時間複雜度為 O(n)；與較小的子節點比較並交換的時間複雜度為 O( $nlog_2n$)。細節於「Heap Sort / Merge Sort 之比較」的檔案有詳細說明。而下圖的程式碼則是在每次最後一個子節點與根節點的數值對調，並將對掉到最後一個子節點的數值剔除後都從頭到尾搭建一次 heap tree，這將使時間複雜度上升至 O(n)。

```
In [327]: data= [8,7,6,5,4,3,2,1]
     ...: Solution().heap_sort(data)
Out[327]: [1, 2, 3, 4, 5, 6, 7, 8]

In [328]: data= [-7,6,-5,4,-3,2,1]
     ...: Solution().heap_sort(data)
Out[328]: [-7, -5, -3, 1, 2, 4, 6]

In [329]: data= [2,2,4,4,-3,2,1]
     ...: Solution().heap_sort(data)
Out[329]: [-3, 1, 2, 2, 2, 4, 4]

In [330]: data=[1]
     ...: Solution().heap_sort(data)
Out[330]: [1]

In [331]: data=[]
     ...: Solution().heap_sort(data)
Out[331]: 'no element in the list'
```

```python
class Solution(object):
    def heap_sort(self,data):
        if len(data)==0:
            return 'no element in the list'
        dummy=min(data)-1
        data=[dummy]+data
        D=[]
        while len(data)>1:
            data=self.Tree(data)
            self.Pop(data,D)
        return D

    def Tree(self,data):
        p=0
        for i in data[1:]:
            p+=1
            index=p
            element=i
            while data[index//2] > element:
                data[index]=data[index//2]
                index=index//2
            data[index]=element
        return data

    def Pop(self,data,D):
        data[1],data[-1]=data[-1],data[1]
        k=data.pop(-1)
        D.append(k)
        return D
```

3. 原創程式碼

```
1  class Solution(object):
2      def heap_sort(self,data):
3          if len(data)==0:
4              return 'no element in the list'
5          dummy=min(data)-1
6          data=[dummy]+data
7          D=[]
8          self.Tree(data)
9          self.Pop(data,D)
10         while len(data) > 1:
11             self.DeleteMin(data)
12             self.Pop(data,D)
13         return D
14
15     def Tree(self,data):
16         p=0
17         for i in data[1:]:
18             p+=1
19             index=p
20             element=i
21             while data[index//2] > element:
22                 data[index]=data[index//2]
23                 index=index//2
24             data[index]=element
25         return data
```

```
27     def DeleteMin(self,data):
28         last=len(data)-1
29         element=data[1]
30         index_root=1
31         index_c=index_root*2
32         while index_c <= last:
33             if index_c < last:
34                 if data[index_c] > data[index_c+1]:
35                     index_c=index_c+1
36             if element > data[index_c]:
37                 data[index_root]=data[index_c]
38                 index_root=index_c
39                 index_c=index_c*2
40             else:
41                 break
42         data[index_root]=element
43         return data
44
45     def Pop(self,data,D):
46         data[1],data[-1]=data[-1],data[1]
47         k=data.pop(-1)
48         D.append(k)
49         return D
```

```
In [20]: data= [-7,6,-5,4,-3,2,1]
    ...: Solution().heap_sort(data)
Out[20]: [-7, -5, -3, 1, 2, 4, 6]

In [21]: Solution().heap_sort(data)
    ...: data= [6,5,4,3,2,1]

In [22]: data= [6,5,4,3,2,1]
    ...: Solution().heap_sort(data)
Out[22]: [1, 2, 3, 4, 5, 6]

In [23]: data= [2,1]
    ...: Solution().heap_sort(data)
Out[23]: [1, 2]

In [24]: data=[]
    ...: Solution().heap_sort(data)
Out[24]: 'no element in the list'
```

以下為 Heap Sort 程式碼的所有步驟:

i. 第 3、4 行。檢查所輸入的數列是否為空，如果為空數列則輸出 no element in the list。

ii. 增加一個數值在輸入的數列中的最前面，該新增的數值會是原數列中最小的數值減掉 1。

iii. 第 7~10 行。將數列排序成堆積樹型態，接著將樹根與最後的子節點對調，然後把調換到最後節點的數值從堆積樹型態的數列中剔除，另外再額外創一個新的空陣列，將剔除的數值放到新的陣列中，該數值則作為這個新陣列的首位數值。

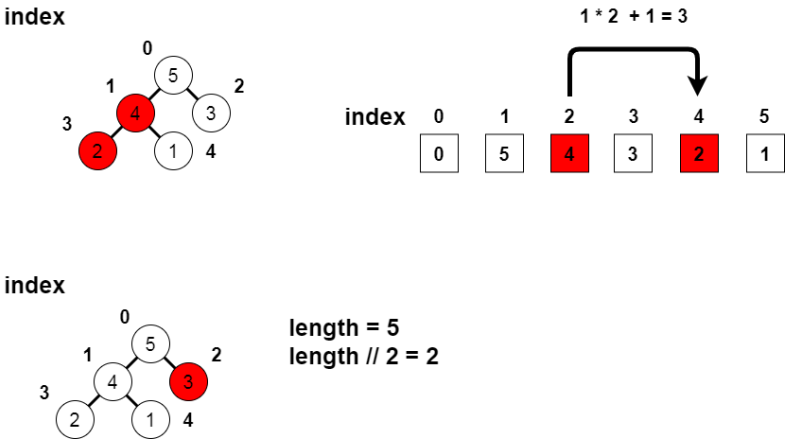
其中堆積樹排序的方式是透過 Tree 函式進行排序，其原理是從數列的樹根由上往下排序，讓每個父節點都小於子節點。因此由上到下依序訪尋到每個節點時 (第 17 行) ，都要從該節點開始由下至上運用該節點

的 index 除 2 的商來走訪所有父節點，直到走訪到樹根節點的下一個節點為止 (第 18~20 行、21 行、23 行) 。走訪時依序與父節點比較大小，如果走訪到的節點小於其父節點，該節點就與其父節點對調 (第 22、24 行) 。

iv. 第 10~12 行。將位於數根的數值向下依序與其較小的子節點比較，當該數值大於子節點時，兩數值則互換，互換後原位於數根的數值再繼續與更下層較小的子節點比較，直到比到最後一個節點為止，全部比較完後再將位於樹根節點的數值與位於最後一個子節點的數值對調，並將對調到最後一個子節點的數值剔除，排進新數列中。每次都用 DeleteMin 這個函式來完成。

4. 延伸學習的程式碼

參考 <使用資料結構 Python，李淑馨著，深石數位科技> 第九章 9.3.3 節第 9-32 頁中 HeapSort 的程式碼。該程式碼與原創程式碼的差異在於進行堆積樹排序的方式。在排序成堆積樹型態時，該程式是從最後一個父節點由下往上排序，讓每個父節點都小於子節點，因此在由下到上依序訪尋到每個節點時，都要從該節點開始由上至下走訪所有的子節點，直到走訪到最後一個子節點為止，走訪時依序與子節點比較大小，如果走訪到的節點大於其子節點，該節點就與其子節點對調。其中，由於該程式並未在數列最前面新增一個數值，所以每個節點的 index 乘 2 加 1 才會是該節點的子節點的索引。另外，數列的長度除 2 即為該數列最後一個父節點在數列中下一個數值的索引，因此在由下到上依序訪尋時會先從該數值的節點開始。




```
In [335]: data= [1,1,2,2,3,3,4,4]
     ...: SortHeap(data)
Out[335]: [1, 1, 2, 2, 3, 3, 4, 4]

In [336]: data= [5,4,6,8,2,5,9,2]
     ...: SortHeap(data)
Out[336]: [2, 2, 4, 5, 5, 6, 8, 9]

In [337]: data=[7,6,5,4,3,2,1]
     ...: SortHeap(data)
Out[337]: [1, 2, 3, 4, 5, 6, 7]

In [338]: data=[-7,6,-5,4,-3,2,-1]
     ...: SortHeap(data)
Out[338]: [-7, -5, -3, -1, 2, 4, 6]

In [339]: data=[1]
     ...: SortHeap(data)
Out[339]: [1]

In [340]: data=[]
     ...: SortHeap(data)
Out[340]: 'no element in the list'
```

```python
def sortHeap(Ary):
    length=len(Ary)-1
    leastParent=length // 2
    for k in range(leastParent,-1,-1):
        heapDown(Ary,k,length)
    for k in range(length,0,-1):
        if Ary[0] > Ary[k]:
            Ary[0],Ary[k]=Ary[k],Ary[0]
            heapDown(Ary,0,k-1)

def heapDown(Ary,first,last):
    largest=2*first+1
    while largest <= last:
        if largest < last and Ary[largest] < Ary[largest+1]:
            largest+=1
        if Ary[largest] > Ary[first]:
            Ary[largest],Ary[first]=Ary[first],Ary[largest]
            first=largest
            largest=2*first+1
        else:
            break
```