Assignment 4

Construction of Heap

โครงสร้างพื้นฐานของ Heap โดยการใส่ attribute พื้นฐานคือ array กับ size

```
class Heap:
    def __init__(self):
        self.heaparray=[0]
        self.size=0
```

1) Insert method

```
def insert(self,k):
    self.heaparray.append(k)
    self.size=self.size+1
    self.Upheap(self.size)
```

Pseudocode:

Insert k in heaparray

size ← size+1

Upheap

2) Upheap method

```
def Upheap(self,i):
    while i // 2 >0:
        if self.heaparray[i] < self.heaparray[i//2]:
            A=self.heaparray[i//2]
            self.heaparray[i//2]=self.heaparray[i]
            self.heaparray[i]=A
        i=i//2
```

Pseudocode:

While quotient of i divided by 2 greater than 0

    If heaparray[i] < heaparray[i//2]

        A←heaparray[i//2]

        heaparray[i//2]=heaparray[i]

        heaparray[i]←A

    i=i//2

end

3) Downheap Method

```
def Downheap(self,i):
    while (i*2) <= self.size:
        min=self.minchild(i)
        if self.heaparray[i] > self.heaparray[min]:
            A=self.heaparray[i]
            self.heaparray[i]=self.heaparray[min]
            self.heaparray[min] = A
        i=min
```

Pseudocode:

While (i*2) <= size

min=minchild(i)

If heaparray[i] > heaparray[min]

A←heaparray[i]

heaparray[i]←heaparray[min]

heaparray[min] ← A

i←min

end

end

4) minchild method

```
def minchild(self,i):
    if i * 2 +1 > self.size:
        return i*2
    else:
        if self.heaparray[i*2] < self.heaparray[i*2+1]:
            return i*2
        else:
            return i*2+1
```

Pseudocode:

```
If i*2+1 > size
        return
else
        if heaparray[i*2] < heaparray[i*2+1]
                return i*2
        else
                return i*2+1
```

5) removemin method

```python
def removemin(self):
    H=self.heaparray[1]
    self.heaparray[1]=self.heaparray[self.size]
    self.size= self.size-1
    self.heaparray.pop()
    self.Downheap(1)
    return H
```

Pseudocode:

```
H ← heaparray[1]
heaparray[1] ← heaparray[size]
size ← size-1
remove last element in heaparray
Downheap
return H
```

6) Construct_heap method

```
def Construct_heap(self,L):
    i=len(L)//2
    self.size=len(L)
    self.heaparray=[0]+L[:]
    while (i>0):
        self.Downheap(i)
        i-=1
```

Pseudocode:

i ← size of List L //2

size of heaparray=size of List L

heaparray ← [0] + L[:]

While i>0

Downheap(i)

i ←i-1

end

Implementation of Heap

1) Heap Sort

```
def Heap_Sort(L):
    H=Heap()
    H.Construct_heap(L)
    A=[H.removemin() for i in range (len(L))]
    return A

A=Heap_Sort([1,5,9,-1,10,6,7,8,55])
print(A)
```

โดยการ input List ดังภาพเข้าไปใน Heap Sort algorithm

ผลที่ได้:

```
Sorted element from minimum to maximum
[-1, 1, 5, 6, 7, 8, 9, 10, 55]
PS C:\Users\yoshi\Desktop\Code>
```

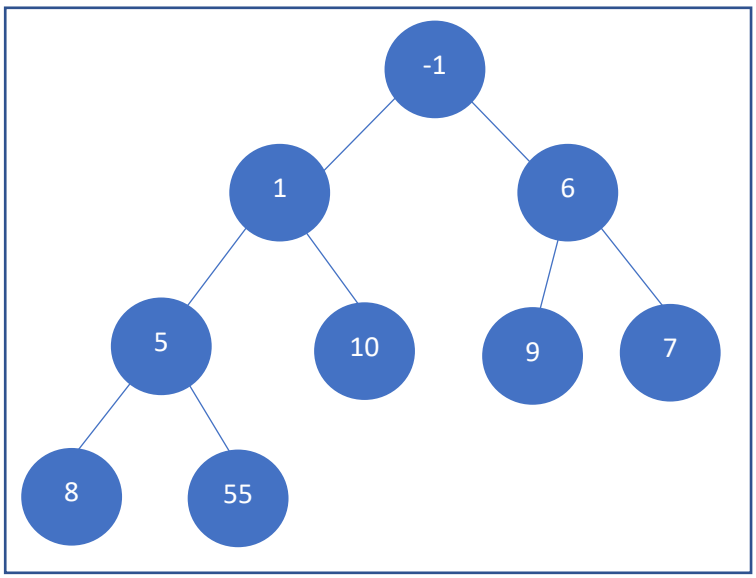
จะเห็นได้ว่าทุกสมาชิกใน List ถูกเรียงเป็นที่เรียบร้อยแล้วจากน้อยไปมาก

2) Construct heap from any list of integers

```
H=Heap()
H.Construct_heap([1,5,9,-1,10,6,7,8,55])
print(H.heaparray)
```

ผลที่ได้คือ:

```
[0, -1, 1, 6, 5, 10, 9, 7, 8, 55]
PS C:\Users\yoshi\Desktop\Code>
```

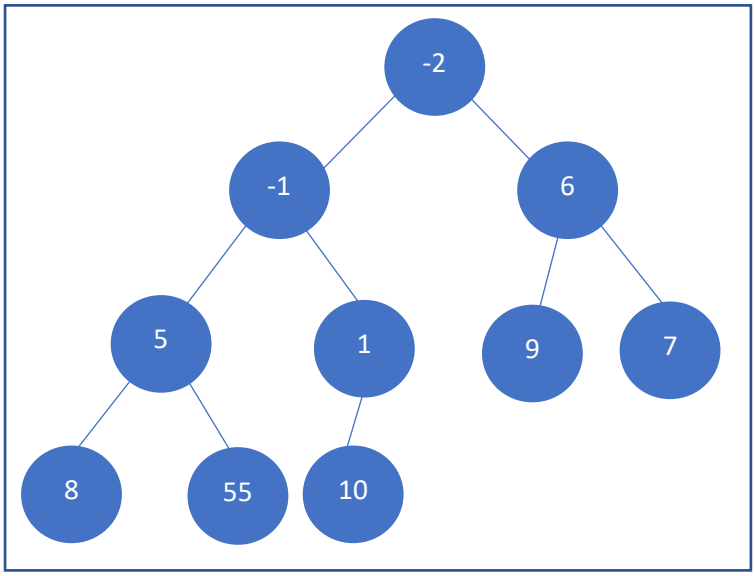


3) Insert

```
H=Heap()
H.Construct_heap([1,5,9,-1,10,6,7,8,55])
H.insert(-2)
print(H.heaparray)
```

ผลที่ได้:

```
[0, -2, -1, 6, 5, 1, 9, 7, 8, 55, 10]
PS C:\Users\yoshi\Desktop\Code>
```

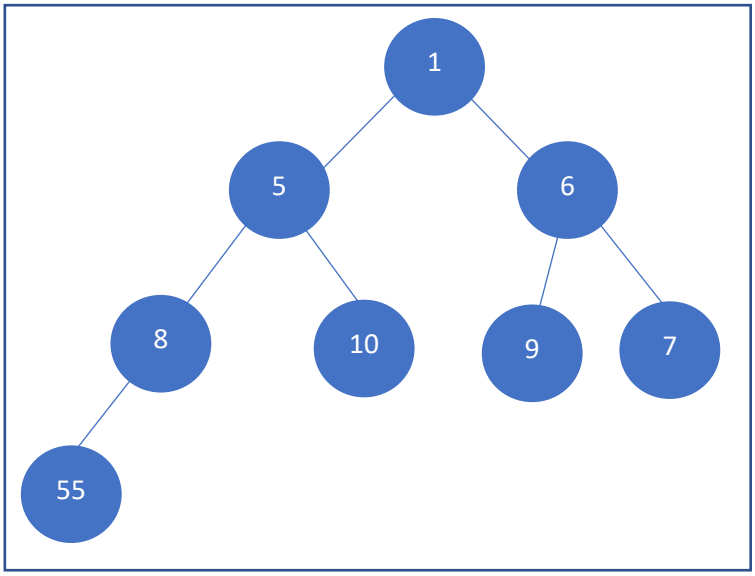


4) Removemin

```
H=Heap()
H.Construct_heap([1,5,9,-1,10,6,7,8,55])
H.insert(-2)
H.removemin()
H.removemin()
print(H.heaparray)
```

ผลที่ได้:

```
[0, 1, 5, 6, 8, 10, 9, 7, 55]
PS C:\Users\yoshi\Desktop\Code
```



Complexity analysis

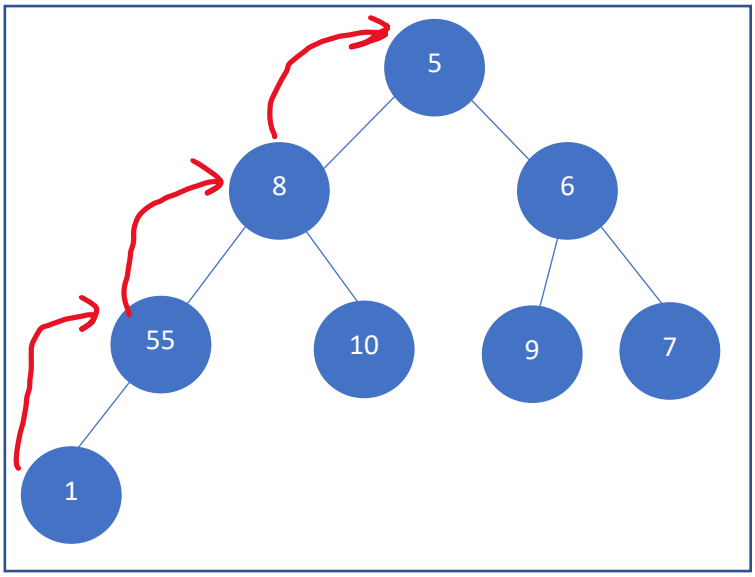
1) Upheap method

```
def Upheap(self,i):
    while i // 2 >0:
        if self.heaparray[i] < self.heaparray[i//2]:
            A=self.heaparray[i//2]
            self.heaparray[i//2]=self.heaparray[i]
            self.heaparray[i]=A
        i=i//2
```

เนื่องจาก Algorithm นั้นจะ insert element ตัวล่าสุดไว้ที่ child ตัวสุดท้ายดังนั้น child ที่ถูก insert จะ swap ขึ้นไปถ้าคุณสมบัติของ heap นั้นไม่ถูกต้อง กล่าวคือ ถ้า value ของ child มีค่าน้อยกว่า parent จะต้องถูก swap กับ parent ตัวนั้นจนกระทั่งมีคุณสมบัติของ heap หรืออีกกรณีที่ child ตัวที่ถูก insert มีค่าน้อยที่สุดจะต้อง swap กับ parents จนถึง root ของ heap

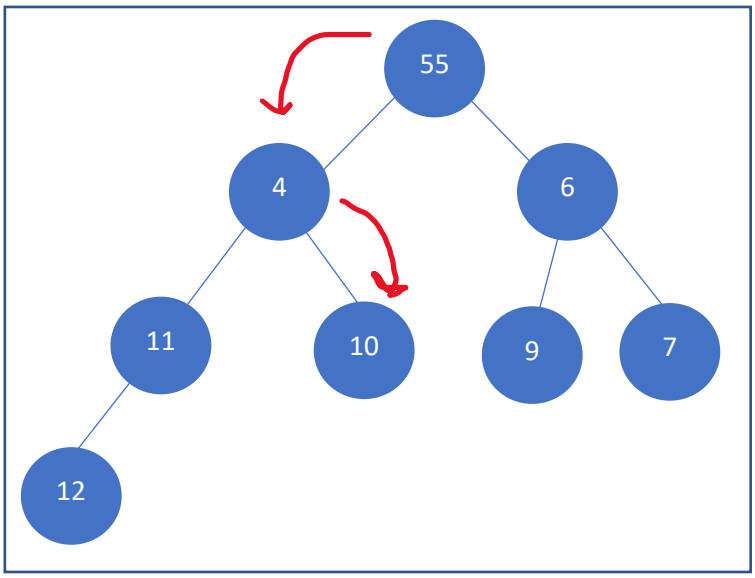
จากค่าของ ความลึกของขั้นของ heap จะมีค่าเท่ากับ h ซึ่งเท่ากับ $O(logn)$

ดังนั้นถ้า element ถูก swap ไปตามชั้นต่างๆ มากที่สุด h ชั้น จะได้ running time เป็น $O(logn)$



2) Downheap method

```
def Downheap(self,i):
    while (i*2) <= self.size:
        min=self.minchild(i)
        if self.heaparray[i] > self.heaparray[min]:
            A=self.heaparray[i]
            self.heaparray[i]=self.heaparray[min]
            self.heaparray[min] = A
        i=min
```



จากการลบ element ตัวที่น้อยที่สุดจะนำ element ตัวท้ายสุดมาไว้ที่ root ดังนั้นถ้า สภาพ child กับ parent ตัวอื่นยังคงคุณสมบัติ heap ดังนั้นมีแค่ตัว root ที่ยังทำให้ ไม่เป็น heap จึงต้อง swap root เช่นดังภาพ 55 สลับกับ child 4 เพราะน้อยกว่า 6 แล้วก็ สลับ กับ 10 เพราะ 10 น้อยกว่า 11

ดังนั้นจะสลับลงไปลึกมากสุด h ชั้น ดังนั้น running time จะได้ $O(logn)$

3) Removemin method

```
def removemin(self):
    H=self.heaparray[1]
    self.heaparray[1]=self.heaparray[self.size]
    self.size= self.size-1
    self.heaparray.pop()
    self.Downheap(1)
    return H
```

การ removemin จะแทนค่าให้ heaparray[1]=heaparray[size] (ตัวท้ายสุด) ขั้นตอนนี้จะต้องใช้เวลาในการหา element ตัวแรกเป็น $O(n)$ เนื่องจากเราใช้ Array-based แล้วลบ element ท้ายสุด แล้วจึงไป Downheap ต่อจะได้ว่า Running time จากการ Downheap จะเป็น $O(logn)$ แต่การ removemin จะใช้เวลา $O(n)$

4) Heap Sort

```
def Heap_Sort(L):
    H=Heap()
    H.Construct_heap(L)
    A=[H.removemin() for i in range (len(L))]
    return A

A=Heap_Sort([1,5,9,-1,10,6,7,8,55])
print(A)
```

การใช้ Heap Sort จะทำการ construct heap จะใช้เวลา $O(n)$ ซึ่งการ remove root แล้วทำการ Down heap ต่อซึ่งการ Down heap จะใช้เวลา $O(logn)$ ดังนั้น จะทำการ Down heap จำนวน n ครั้ง เพราะดึง element ไปทั้งหมด n ตัวจะได้ว่า running time คือ $O(logn) * O(n)$ ซึ่งก็คือ $O(nlogn)$

Implementation Priority Queue by array

```python
class PriorityQueue():
    def __init__(self):
        self.queue = []

    def isEmpty(self):
        return len(self.queue) == 0

    def insert(self, data):
        if (type(data) == dict) and (len(data) == 1):
            new_key = list(data.keys())[0]
            if (type(new_key) == str) and (type(data[new_key]) == int):
                self.queue.append(data)
        print(data)

    def delete(self):
        if self.isEmpty():
            return [None, None]
        max_index = None
        max_int = None
        max_key = None
        for i in range(len(self.queue)):
            pair_key = list(self.queue[i].keys())[0]
            pair_int = self.queue[i][pair_key]
            if (max_index == None) or (pair_int < max_int):
                max_index = i
                max_int = pair_int
                max_key = pair_key
        del self.queue[max_index]
        return [max_key, max_int]
```

```python
A = PriorityQueue()
A.insert({"A": 0})
A.insert({"A": 1})
A.insert({"D": 2})
A.insert({"A": 3})
A.insert({"D": 4})
A.insert({"D": 5})

print(A.delete())
```

```
{'A': 0}
{'A': 1}
{'D': 2}
{'A': 3}
{'D': 4}
{'D': 5}
['A', 0]
```