R8 Priority Queus

3.25 https://github.com/GUMI-21/MIT6.006_note

Priority Queus

algorithm	data structure	insertion	extraction	total
Selection Sort	Array	$O(1)$	$O(n)$	$O(n^2)$
Insertion Sort	Sorted Array	$O(n)$	$O(1)$	$O(n^2)$
Heap Sort	Binary Heap	$O(\log n)$	$O(\log n)$	$O(n \log n)$

A interface of python code

```
class PriorityQueen:
    def __init__(self):
        self.A = []
    def insert(self, x):
        self.A.append(x)
    def delete_max(self):
        if len(self.A) < 1:
            rais IndexError('pop from enpty queue')
        return self.A.pop()
    @classmethod
    def sort(Queue, A):
        pq = Queue()  # make empty priority queue
        for x in A:
            pq.insert(x)
        out = [pq.delete_max() for _ in A]
        out.reverse()
        return out
```

Array Heaps

- Array

```
class PQ_Array(PriorityQueue):
    # def insert just append
    def delete_max(self):    # O(n)
        n, A, m = len(self.A), slef.A, 0
        for i in range(1,n):
            if A[m].key < A[i].key
                m = i
```

```
        A[m], A[n] = A[n],A[m]  # swap max with end of array
        return super().delete_max()   #pop from end of array
```

- sortedArray

```
class PQ_SortedArray(PriorityQueue):
    # delete_max: pop from end
    # O(n)
    def insert(self, *args):
        super.insert(*args)
        i, A = len(self.A) - 1, self.A  #restore
            while 0 < i and A[i+1].key < A[i].key:
                A[i+1],A[i] = A[i], A[i+1]
                i -= 1
```

Binary Heaps

```
class PQ_Heap(PriorityQueue):
    def insert(self, *args):   #O(logn)
        super().insert(*args)
        n, A = self.n, self.A
        max_heapify_up(A,n,n-1)
    def delete_max(self):   #O(logn)
        n , A = self.n, self.A
        A[0], A[n] = A[n], A[0]
        max_heapify_down(A,n,0)
        return supter().delete_max() # pop from end of array
```

- *find parent and child*

```
def parent(i):
    p = (i - 2) // 2
    return p if 0 < i else i
def left(i,n):
    l = 2*i+1
    return l if l < n else i
def right(i,n)
    r = 2*i+2
    return r if r < n else i
```

- max_heapify_up & max_heapify_down:

```
def max_heapify_up(A, n, c):  O(log c)
    p = parent(c)
    if A[p].key < A[c].key
        A[c], A[p] = A[p], A[c]
        max_heapify(A, n, p)
def max_heapify_down(A, n, p):  O(log n)
    l,r = left(p,n), right(p,n)
    c = l if A[r].key < A[l].key else r
    if A[p].key < A[c].key
        A[c], A[p] = A[p], A[c]
        max_heapify_down(A,n,c)
```

O(n) Build Heap

construct the heap in reverse level order, from the leaves to root.
if from root to leaves run max_heapify_down will cost much more time.

```
def build_max_heap(A):
    n = len(A)
    for i in range(n//2, -1, -1)  # becase we don't need to proccess leaf
        max_heapify_down(A, n, i)    # O(logn - log i)
```

O(n)

To see that this procedure takes $O(n)$ instead of $O(n \log n)$ time, we compute an upper bound explicitly using summation. In the derivation, we use Stirling's approximation: $n! = \Theta(\sqrt{n}(n/e)^n)$.

$$T(n) < \sum_{i=0}^{n} (\log n - \log i) = \log\left(\frac{n^n}{n!}\right) = O\left(\log\left(\frac{n^n}{\sqrt{n}(n/e)^n}\right)\right)$$
$$= O(\log(e^n/\sqrt{n})) = O(n \log e - \log \sqrt{n}) = O(n)$$

logn - logi means node i need do max_heapify_down times.

In-Place Heaps

```
class PriorityQueue:
    def __init__(self, A):
        self.n, self.A = 0, A
    def insert(self):
        if not self.n < len(self.A):
            rais IndexError('insert into full priority queue')
        self.n += 1
    def delete_max(self):
        if self.n < 1
            raise IndexError('pop from empty priority queue')
```

```
        self.n -= 1
    @classmethod
    def sort(Queue, A):
        pq = Queue(A)
        for i in range(len(A))
            pq.insert()
        for i in range(len(A)):
            pq.delete_max()
        return pq.A
```

instead, it inserts the item already stored in A[n], and incorporates it into the now-larger queue. Similarly, delete max does not return a value; it merely deposits its output into A[n] before decreasing its size.

can seem as a dynamic Array AND maintain lenth of bianry heap n.