Heaps - III

Priority queue

↳ is a data structure for maintaining a set S of elements, each with an associated value called the key (or priority)

— A max-priority queue uses a max-heap

Operations: Insert(S, x, k):
insert element x with key k
$\Rightarrow S = S \cup \{x\}$

Maximum(S): returns the element with largest key

Extract-Max(S): removes & returns element with largest key

Increase-key(S, x, k): increases value of x's key to new value k (assumed that $k \geq x.key$)

— Applications of max-priority queue:

↳ Schedule job on a computer, keeping priorities of jobs

a) once a job is finished, the scheduler selects the highest priority job by calling Extract-Max

b) New jobs are added with Insert()

Max-Heap-Maximum(A)

```
if A.heap-size < 1
    error "heap underflow"
return A[1]
```

→ $\Theta(1)$

Max-Heap-Extract-Max(A)

```
max = Max-Heap-Maximum(A)
A[1] = A[A.heap size]
A.heap size = A.heap size - 1
Max-Heapify(A, 1)
return max
```

→ $O(\lg n)$

→ When we implement a priority queue, we treat each array element as the keys to be sorted and assume that the satellite data moves with corresponding keys.

Max-Heap-Increase-key (A, x, k)

```
if k < x.key
        error "New key is smaller than the existing"
x.key = k
Find the index i in array A where object x occurs
while i > 1 and A[parent(i)].key < A[i].key
        exchange A[i] with A[parent(i)],
            (update info that maps priority queue
             objects to array indices)
        i = parent(i)
```

$O(\lg n)$ + overhead of mapping priority queue object to indices

Max-Heap-Insert (A, x, n)

```
if A.heap-size == n
        error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = -∞
A[A.heap-size] = x
map x to index heap-size in the array
Max-Heap-Increase-key (A, x, k)
```

$O(\lg n)$ + overhead of mapping objects to indices.