

Lecture 12: Heapsort, Priority Queue & Quicksort

Agenda:

- Heapsort (review)
- Priority queue
- Quicksort
 - A divide-and-conquer algorithm
 - The ideas & execution

Reading:

- Textbook pages 138 – 149

Heapsort algorithm:

- Make a heap from the elements. This takes $\Theta(n)$.
- The root (i.e. $A[1]$), is the max. Exchange it with $A[n]$ and decrease heapsize.
- Now we work with array $A[1..(n-1)]$ which is almost heap.
- Call Max-Heapify to turn $A[1..(n-1)]$ into a heap. This takes $\Theta(\lg n)$.
- Repeat until all elements are placed in the position.
- The total running time is $\Theta(n \lg n)$.

Priority Queue:

- An abstract data structure for maintaining a set S of *elements* each associated with a *key*
- Key — represents the priority of the element
- Example: a set of jobs to be scheduled on a shared computer.
 - The jobs arrive and should be placed in the queue.
 - Each has a priority. Queue should be with respect to this.
 - To perform a job, we “extract” the one in the queue with highest priority.
- In general, a PQ supports these operations:
 - `initialize` — insert all keys at once
 - `insert` — a new element
 - `maximum` — return the element with the maximum key
 - `extract maximum` — return the maximum and remove the element from the queue
 - `increase key` — increase the priority for an element
- Implementation? **Heap !!!**
Arrange the priorities (keys) into a max-heap, and
make a link from each priority to the corresponding element
— rearrangement of priorities $\iff$ rearrangement of elements.

Lecture 12: Priority Queue

- `Initialize( $A$ )` — Build-Max-Heap. So this takes $\Theta(n)$ time.
- `Maximum( $A$ )` — return $A[1]$. Takes $\Theta(1)$ time.
- `Extract-Maximum( $A$ )` — return $A[1]$ and decrease heapsize, pull the last key to the top, Max-Heapify. $\Theta(\lg n)$ time.

```
procedure Heap-Extract-Max( $A$ )    **p 139
    if  $\text{heap} - \text{size}[A] < 1$  then Error ‘‘heap underflow’’
     $\text{max} \leftarrow A[1]$ 
     $A[1] \leftarrow A[\text{heap} - \text{size}[A]]$ 
     $\text{heap} - \text{size}[A] \leftarrow \text{heap} - \text{size}[A] - 1$ 
    Max-Heapify( $A, 1$ )
    return  $\text{max}$ 
```

- `Increase-Key( $A, i, \text{new\_key}$ )` — increase the priority value for $A[i]$ and bubble up to the right position. $\Theta(\lg n)$ time.

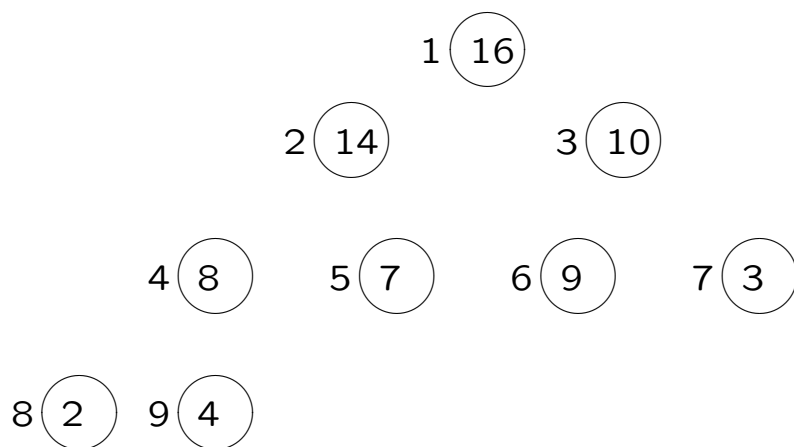
```
procedure Heap-Increase-Key( $A, i, \text{key}$ )    **p 140
    if  $\text{key} < A[i]$  then Error ‘‘new key is smaller’’
     $A[i] \leftarrow \text{key}$ 
    while  $i > 1$  and  $A[\text{Parent}(i)] < A[i]$  do
        exchange  $A[i] \leftrightarrow A[\text{Parent}(i)]$ 
         $i \leftarrow \text{Parent}(i)$ 
```

- `Insert( $A, \text{new\_key}$ )` — increase heapsize, add a new key with priority value equal to `new_key`, and bubble up. $\Theta(\lg n)$ time.

```
procedure Heap-Insert( $A, \text{key}$ )                **p 140
     $\text{heap} - \text{size}[A] \leftarrow \text{heap} - \text{size}[A] + 1$ 
     $A[\text{heap} - \text{size}[A]] \leftarrow -\infty$ 
    Heap-Increase-key ( $A, \text{heap} - \text{size}[A], \text{key}$ )
```

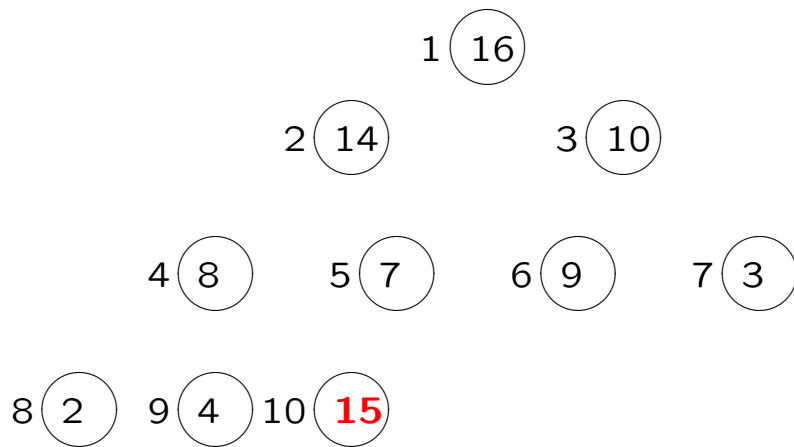
Lecture 12: Priority Queue

Starting with a heap:



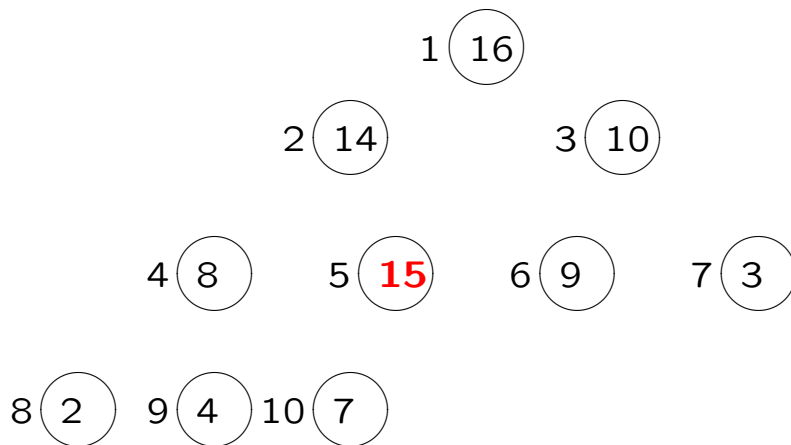
Lecture 12: Priority Queue

Max-heap-Insert ($A, 15$):



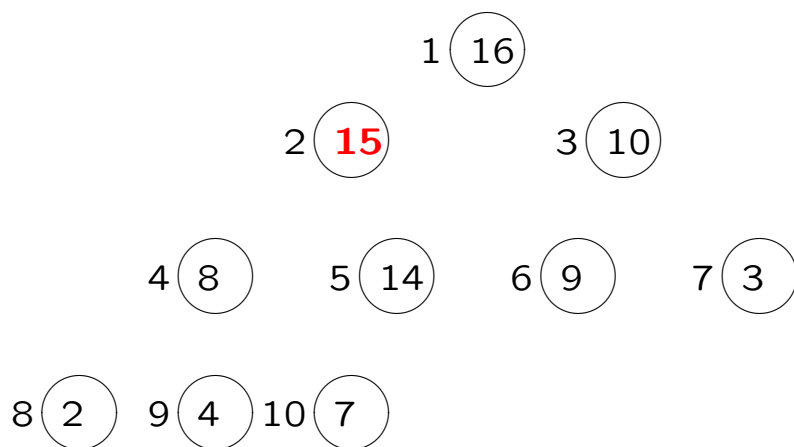
Lecture 12: Priority Queue

Heap-Increase-Key ($A, \text{heap-size}[A], 15$) is called:



Lecture 12: Priority Queue

Bubbling up key 15:



Another sorting meets divide-and-conquer:

- The ideas:
 - Pick one key
 - Compare to others: partition into 'smaller' and 'greater' sublists
 - Recursively sort two sublists

- Pseudocode:

```
procedure Quicksort( $A, p, r$ )          **p 146
```

```
    if  $p < r$  then
         $q \leftarrow \text{Partition}(A, p, r)$ 
        Quicksort( $A, p, q - 1$ )
        Quicksort( $A, q + 1, r$ )
```

```
procedure Partition( $A, p, r$ )          **p 146
```

```
    **  $A[r]$  is the key picked to do the partition
```

```
     $x \leftarrow A[r]$ 
     $i \leftarrow p - 1$ 
    for  $j \leftarrow p$  to  $r - 1$  do
        if  $A[j] \leq x$  then
             $i \leftarrow i + 1$ 
            exchange  $A[i] \leftrightarrow A[j]$ 
    exchange  $A[i + 1] \leftrightarrow A[r]$ 
    return  $i + 1$ 
```

$\text{Partition}(A, p, r)$:

- The invariant:
 - $A[p..i]$ contains keys $\leq A[r]$
 - $A[(i + 1)..(j - 1)]$ contains keys $> A[r]$
- Ideas:
 - $A[j]$ is the current key under examination — $j \geq i$
 - If $A[j] \leq A[r]$, exchange $A[j] \leftrightarrow A[i + 1]$ and increment i to maintain the invariant
 - At the end, exchange $A[r] \leftrightarrow A[i + 1]$ such that:
 - * $A[p..i]$ contains keys $\leq A[i + 1]$
 - * $A[(i + 2)..r]$ contains keys $> A[i + 1]$
 - * After $A[p..i]$ and $A[(i + 2)..r]$ been sorted, $A[p..r]$ is sorted.
- An example: $A[1..8] = \{3, 1, 7, 6, 4, 8, 2, 5\}$, $p = 1$, $r = 8$

3	1	7	6	4	8	2	5	$i = 0, j = 1$
3	1	7	6	4	8	2	5	$i = 1, j = 2$
3	1	7	6	4	8	2	5	$i = 2, j = 3$
3	1	7	6	4	8	2	5	$i = 2, j = 4$
3	1	4	6	7	8	2	5	$i = 3, j = 5$
3	1	4	6	7	8	2	5	$i = 3, j = 6$
3	1	4	6	7	8	2	5	$i = 3, j = 7$
3	1	4	2	7	8	6	5	$i = 4, j = 7$
3	1	4	2	5	8	6	7	$i = 4, j = 7$

Lecture 12: Quicksort

Have you understood the lecture contents?

well	ok	not-at-all	topic
☐	☐	☐	BC running time (two cases)
☐	☐	☐	priority queue
☐	☐	☐	priority queue operations (via heap)
☐	☐	☐	quicksort idea
☐	☐	☐	quicksort pseudocode(s), execution