Lecture 10: Heaps

Agenda:

- Heap
  - a data structure
  - an array of keys organized in some specific way
  - viewing heap as a (binary) tree
- Heap property maintaining
- Heap building

Reading:

- Textbook pages 127 – 135

(Binary-)Heap data structure:

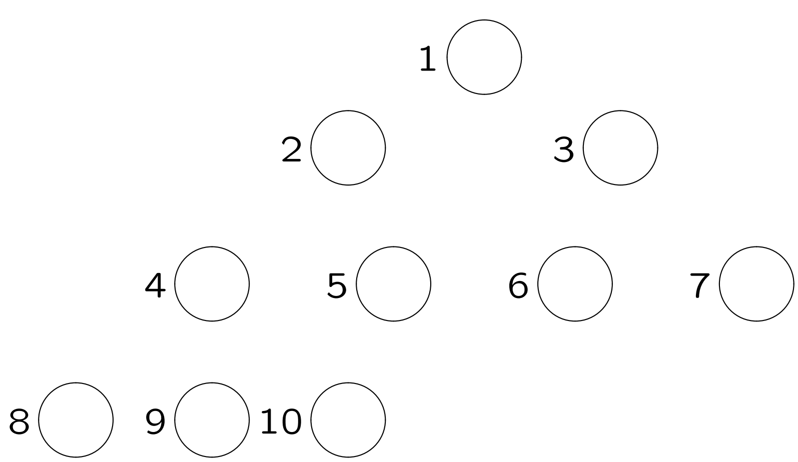
- An array $A[1..n]$ of n *comparable* keys
  either '$\geq$' or '$\leq$'

- An implicit binary tree, where
  - $A[2j]$ is the left child of $A[j]$
  - $A[2j+1]$ is the right child of $A[j]$
  - $A[\lfloor\frac{j}{2}\rfloor]$ is the parent of $A[j]$

- Keys satisfy the *max-heap* property: $A[\lfloor\frac{j}{2}\rfloor] \geq A[j]$

- Examples:

j	1	2	3	4	5	6	7	8	9	10	
$A[j]$	4	1	3	2	16	9	10	14	8	7	heap ? *no.*
$A[j]$	16	14	10	8	7	9	3	2	4	1	heap ? *yes.*



(Binary-)Heap data structure:

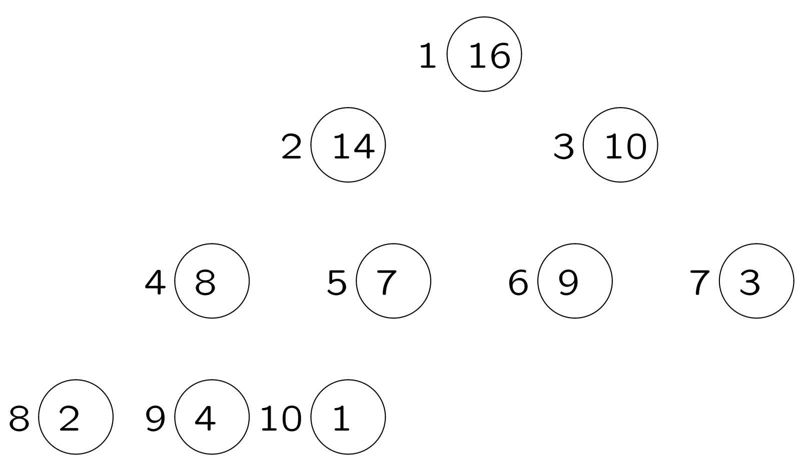
- An array $A[1..n]$ of n *comparable* keys
  either '$\geq$' or '$\leq$'

- An implicit binary tree, where
  - $A[2j]$ is the left child of $A[j]$
  - $A[2j+1]$ is the right child of $A[j]$
  - $A[\lfloor \frac{j}{2} \rfloor]$ is the parent of $A[j]$

- Keys satisfy the *max-heap* property: $A[\lfloor \frac{j}{2} \rfloor] \geq A[j]$

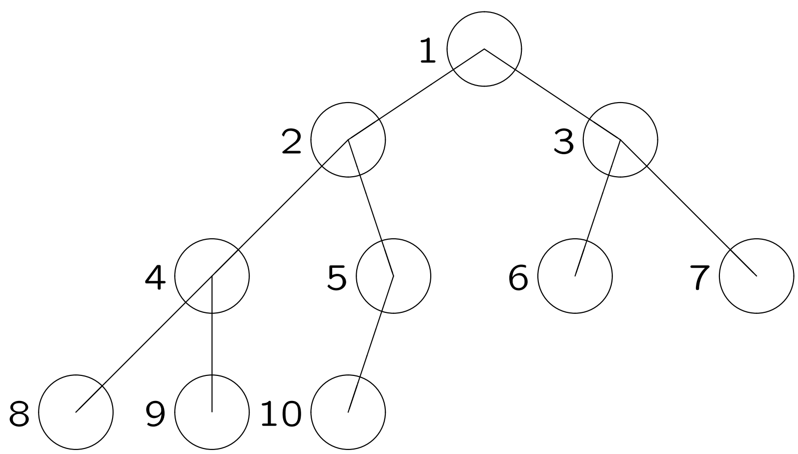
- Examples:

j	1	2	3	4	5	6	7	8	9	10	
$A[j]$	4	1	3	2	16	9	10	14	8	7	heap ? *no*.
$A[j]$	16	14	10	8	7	9	3	2	4	1	heap ? *yes*.



Some heap properties:

- There are max-heap and min-heap. We use max-heap.
- $A[1]$ is the maximum among the n keys.



- Viewing heap as a binary tree, height of the tree is $h = \lfloor \lg n \rfloor$.
  [— the number of edges on the longest root-to-leaf path]

  Call the *height* of the heap.

- Question:
  How many keys can be held into a heap of height k?

Max-Heapify:

- Pre-condition: Suppose we have an array that is almost a heap, except the first key does NOT satisfy the heap property.

- Goal: Suppose we want to make it into a heap.

  How ?

- Compare its two children and *exchange* the larger with the parent.

  This process

  1. does not violate the heap property of the subtree rooted at the unexchanged child,
  2. makes the first position satisfy the heap property,
  3. "trickle-down" the problem to the larger child

- Therefore, repeat this "trickle-down" process will eventually resolve the problem.

- How many steps?

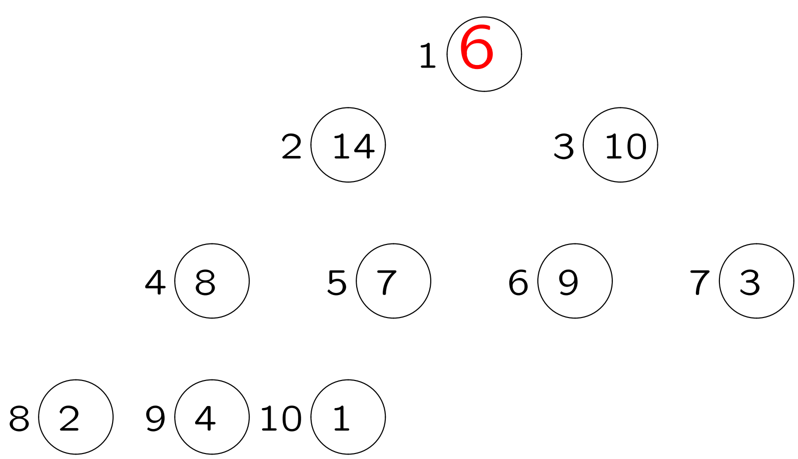
  WC: $\lg n$

Max-Heapify (cont'd):

- Pseudocode:

```
procedure Max-Heapify(A, i)      **p 130
      **turn almost-heap into a heap
      **pre-condition:  tree rooted at A[i] is almost-heap
      **post-condition:  tree rooted at A[i] is a heap

   lc ← leftchild(i)
   rc ← rightchild(i)
   if lc ≤ heapsize(A) and A[lc] > A[i] then
      largest ← lc
   else
      largest ← i
   if rc ≤ heapsize(A) and A[rc] > A[largest] then
      largest ← rc
   if largest ≠ i then
      exchange A[i] ↔ A[largest]
      Max-Heapify(A, largest)
```

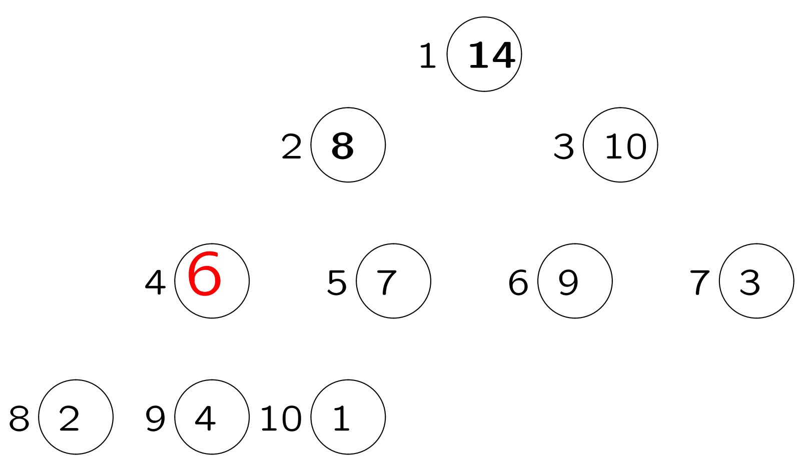


Max-Heapify (cont'd):

- Pseudocode:

procedure Max-Heapify(A, i)   **p 130
        **turn almost-heap into a heap
        **pre-condition: tree rooted at $A[i]$ is almost-heap
        **post-condition: tree rooted at $A[i]$ is a heap

    $lc \leftarrow leftchild(i)$
    $rc \leftarrow rightchild(i)$
    if $lc \leq heapsize(A)$ and $A[lc] > A[i]$ then
        $largest \leftarrow lc$
    else
        $largest \leftarrow i$
    if $rc \leq heapsize(A)$ and $A[rc] > A[largest]$ then
        $largest \leftarrow rc$
    if $largest \neq i$ then
        exchange $A[i] \leftrightarrow A[largest]$
        Max-Heapify($A, largest$)



Building a heap from an array:

- Given: an array of n keys $A[1], A[2], \ldots, A[n]$
- Output: a permutation which is a heap
- Ideas:
  1. Consider the bottom-level nodes in the binary tree:
     Each of them is a single-key heap!
  2. So, the subtrees rooted at the nodes at the second last level are almost-heaps:
     Max-Heapify them into heaps!
  3. So, now the subtrees rooted at the nodes at the third last level are almost-heaps:
     Max-Heapify them into heaps!
  4. ......
  5. The whole tree becomes an almost heap:

     Max-Heapify it into a heap!

     DONE!

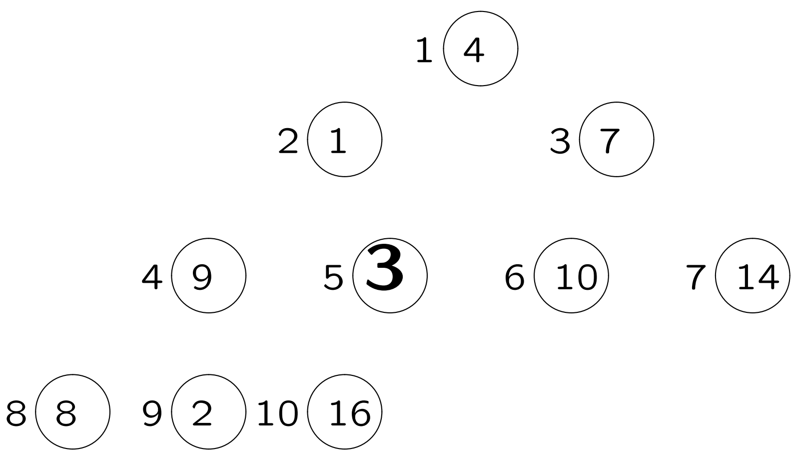
Building a heap from an array (cont'd):

- Pseudocode:

```
procedure Build-Max-Heapify(A) **p 133
       **turn an array into a heap

   heapsize(A) ← length[A]
   for i ← ⌊length[A]/2⌋ downto 1
       do Max-Heapify(A, i)
```

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$



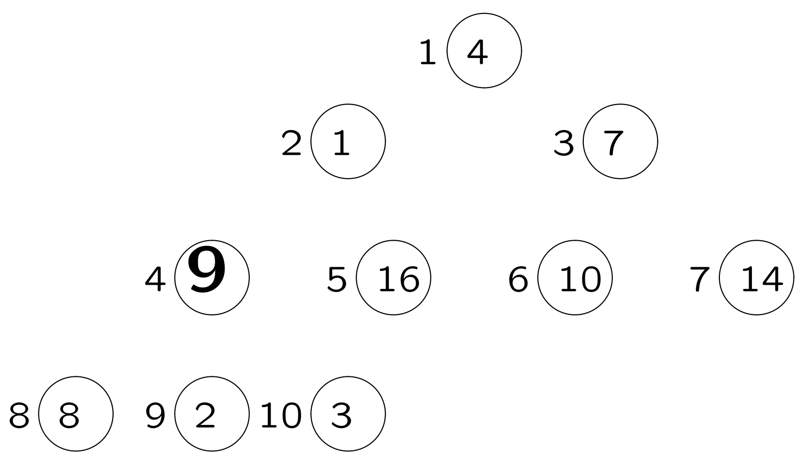
Building a heap from an array (cont'd):

- Pseudocode:

```
procedure Build-Max-Heapify(A) **p 133
       **turn an array into a heap

    heapsize(A) ← length[A]
    for i ← ⌊length[A]/2⌋ downto 1
        do Max-Heapify(A, i)
```

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$



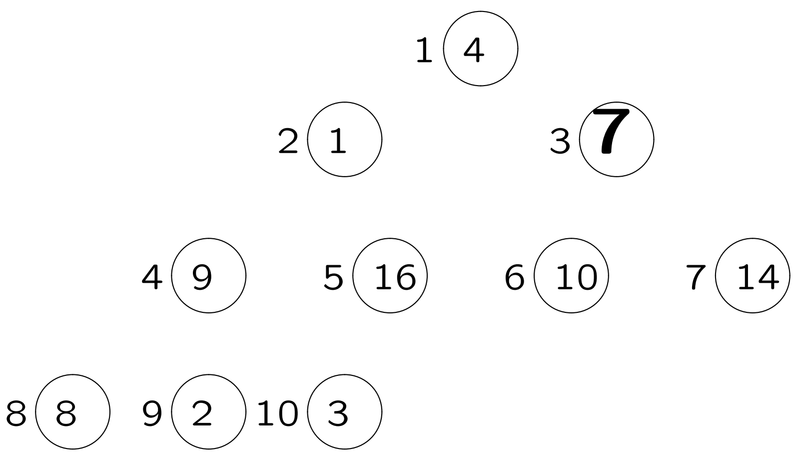
Building a heap from an array (cont'd):

- Pseudocode:

```
procedure Build-Max-Heapify(A) **p 133
      **turn an array into a heap

   heapsize(A) ← length[A]
   for i ← ⌊length[A]/2⌋ downto 1
      do Max-Heapify(A, i)
```

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$



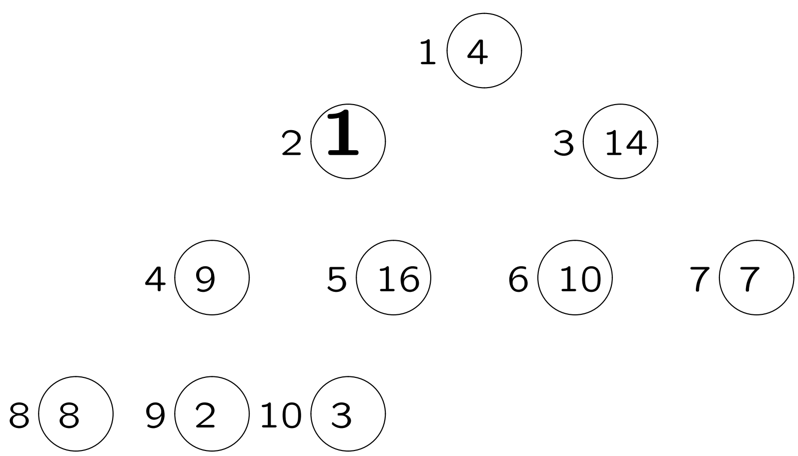
Building a heap from an array (cont'd):

- Pseudocode:

`procedure Build-Max-Heapify(`A`) **p 133`
`**turn an array into a heap`

$heapsize(A) \leftarrow length[A]$
`for` $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ `downto 1`
`do Max-Heapify(`A, i`)`

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$



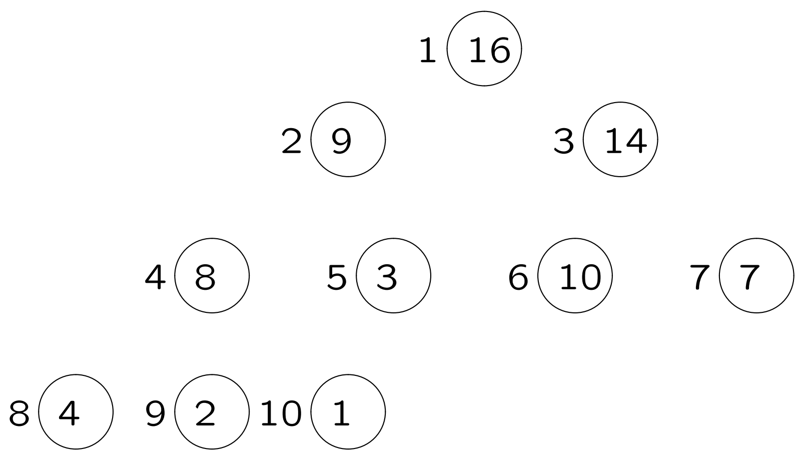
Building a heap from an array (cont'd):

- Pseudocode:

procedure Build-Max-Heapify(A) **p 133
    **turn an array into a heap

$heapsize(A) \leftarrow length[A]$
for $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ downto 1
    do Max-Heapify(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$



Have you understood the lecture contents?

well	ok	not-at-all	topic
☐	☐	☐	what is a (binary, max-) heap
☐	☐	☐	what is an almost-heap
☐	☐	☐	how Max-Heapify works
☐	☐	☐	using Max-Heapify to build a heap