Lecture 11: Heapsort & Its Analysis

Agenda:

- Heap recall:
  - Heap: definition, property
  - Max-Heapify
  - Build-Max-Heap
- Heapsort algorithm
- Running time analysis

Reading:

- Textbook pages 127 – 138

(Binary-)Heap data structure (recall):

- 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: $A[\lfloor \frac{j}{2} \rfloor] \geq A[j]$

- 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 $h = \lfloor \lg n \rfloor$.
  Call the *height* of the heap.
  [— <u>the number of edges</u> on the longest root-to-leaf path]

- A heap of height k can hold 2^k —— $2^{k+1} - 1$ keys.
  Why ???

  Since $\lg n - 1 < k \leq \lg n$
  $\Longleftrightarrow n < 2^{k+1}$ and $2^k \leq n$
  $\Longleftrightarrow 2^k \leq n < 2^{k+1}$

Max-Heapify (recall):

- It makes an almost-heap into a heap.
- 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)
```

- WC running time: $\lg n$.

Build-Max-Heap (recall):

- Given: an array of n keys $A[1], A[2], \ldots, A[n]$

- Output: a permutation which is a heap

- Ideas:

  Repeatedly apply Max-Heapify to nodes in the binary tree representation

  — bottom up

- 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)

- WC running time:

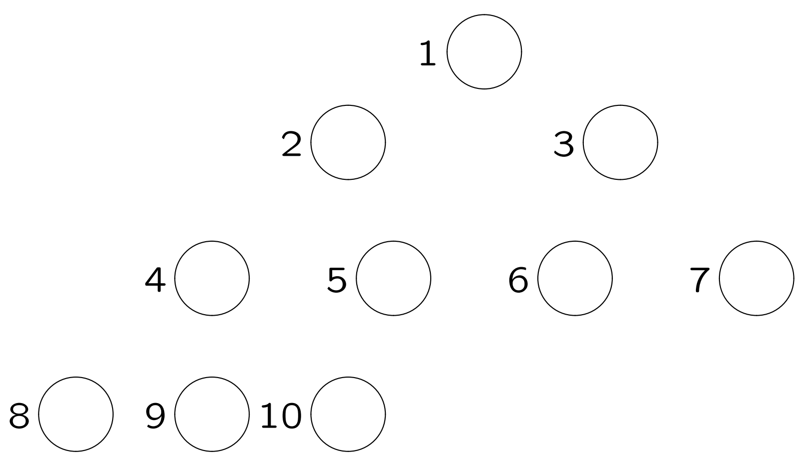
  $\lg n + 2(\lg n - 1) + 2^2(\lg n - 2) + \ldots + 2^{(\lg n - 1)} \cdot 1 = 2n - \lg n - 2.$

Heapsort algorithm:

- `Heapsort` is a data structure algorithm.

- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$,
    which is an almost-heap

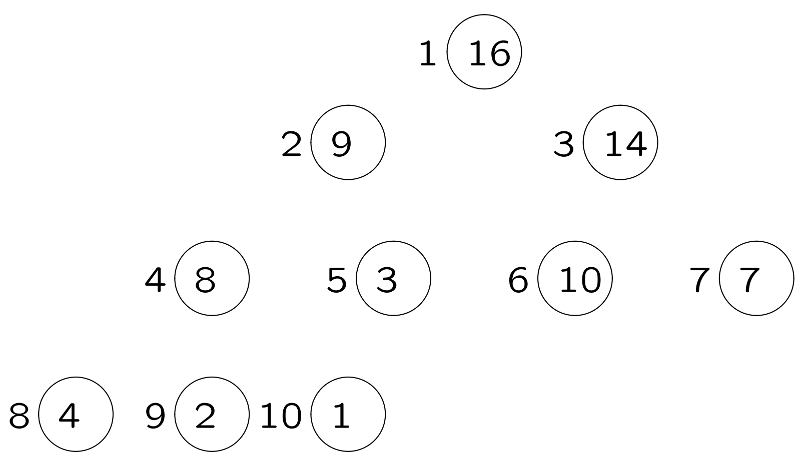
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Build into a heap:



Heapsort algorithm (cont'd):

- `Heapsort` is a data structure algorithm.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$

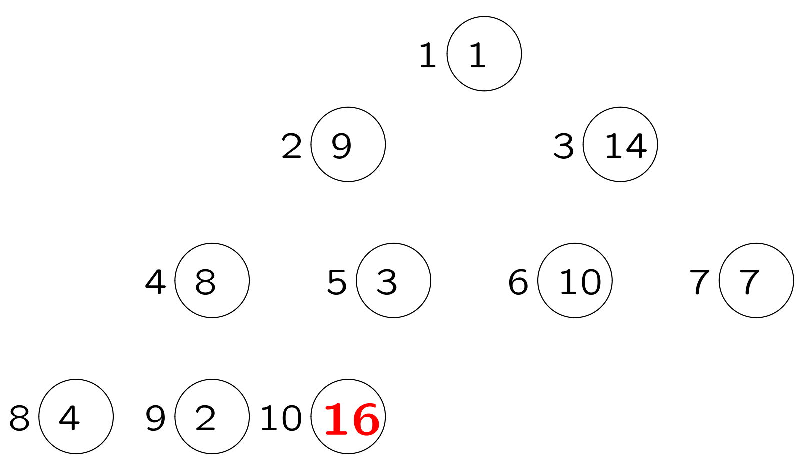
  Heapsize = 10:



Heapsort algorithm (cont'd):

- `Heapsort` is a data structure algorithm.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$

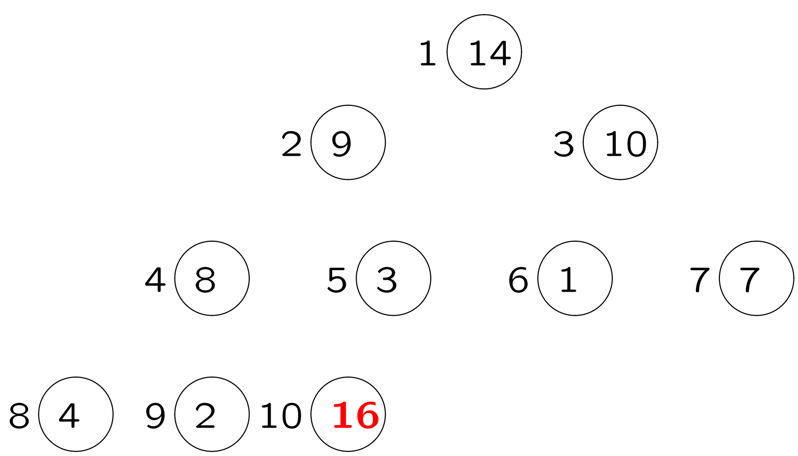
  Exchange $A[1]$ and $A[10]$, decrement Heapsize to 9, and Max-Heapify it (re-install the heap property):



Heapsort algorithm (cont'd):

- `Heapsort` is a data structure algorithm.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$

  Resultant tree: Heapsize = 9:



Heapsort algorithm (cont'd):

- Pseudocode:

```
procedure Heapsort(A)            **p 136
       **post-condition:  sorted array

    Build-Max-Heap(A)
    for i ← length[A] downto 2 do
        exchange A[1] ↔ A[i]
        heapsize(A) ← heapsize(A) − 1
        Max-Heapify(A, 1)
```

- WC running time analysis:
  - Build-Max-Heap in $2n - \lg n - 2$
  - For each i, Max-Heapify in $\lg i$ sum to $\sum_{i=2}^{n} \lg i \in \Theta(n \log n)$
  - So, in total $\Theta(n \log n)$

- Questions:
  1. What is the Worst Case (*array*) for Build-Max-Heap?
  2. What is the Worst Case (*heap*) for the `for` loop?
  3. What is the Worst Case (*array*) for Heapsort?

Heapsort algorithm (cont'd):

- BC running time analysis:
  - all keys equal:
    $\Theta(n)$
  - all keys distinct:
    $\Theta(n \log n)$ — next lecture

- AC running time analysis — very complicated, not required
  - But when *all keys distinct*:
    $\Theta(n \log n)$ — why ???

- Space requirement:
  $\Theta(1)$ — in space sorting algorithm

- Correctness:
  By Loop Invariants:
  - correctness for Max-Heapify (which is a recursion)
  - LI for Build-Max-Heap (p. 133)
  - LI for heapsort (p. 136, Ex 6.4-2)

Have you understood the lecture contents?

well	ok	not-at-all	topic
☐	☐	☐	heap, almost-heap
☐	☐	☐	Max-Heapify
☐	☐	☐	Build-Max-Heap
☐	☐	☐	heapsort algorithm & idea
☐	☐	☐	heapsort analysis (WC running time)