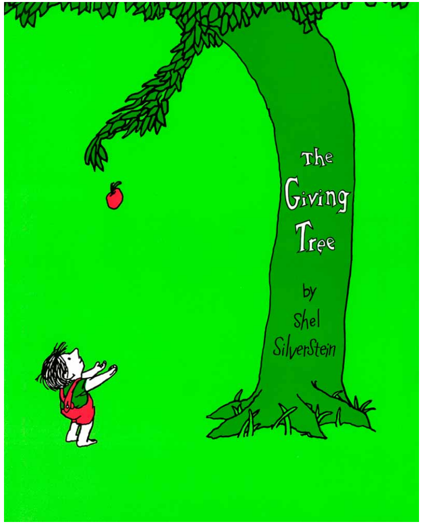
Week04: Solving Recurrences, Master Theorem, Heaps

Agenda:

- Solving Recurrences (Cont'd, using notes of last week)
- Heaps (CLRS Ch. 6.1-6.5)
  - Max-Heapify
  - Build-Max-Heap
  - Heapsort
  - Priority Queues



Heaps 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 $A[j]$
  - So: $A[\lfloor \frac{j}{2} \rfloor]$ is the parent of $A[j]$
- We focus on *max-heap* (there is also min-heap). Keys satisfy the *max-heap property*: for every node j we have $A[\lfloor \frac{j}{2} \rfloor] \geq A[j]$ (i.e., key of parent $\geq$ key of node)
- So the root ($A[1]$) is the maximum among the n keys.
- This gives the alternative definition of a heap: In any *sub-heap*, the root is the largest key
- Viewing heap as a binary tree, height of the tree is $h = \lfloor \lg n \rfloor$. h is called the *height* of the heap (the number of edges on the longest root-to-leaf path)
- All layers i from 0 to $h-1$ are full.
- A heap of height h can hold $[2^h, ..., 2^{h+1}-1]$ keys. Since
  $\lg n - 1 < k \leq \lg n$
  $\iff n < 2^{k+1}$ and $2^k \leq n$
  $\iff 2^k \leq n < 2^{k+1}$

Heaps - examples:

- Examples:
  - $A = [31]$, or any array with a single element
  - $A = [2, 1]$
  - $A = [6, 3, 5]$
  - $A = [6, 3, 5, 1, 2, 4]$
  - $A = [100, 42, 78, 13, 41, 77, 12]$
- Non-examples:

12

  - $A = [1, 2]$
  - $A = [4, 3, 5]$
  - $A = [100, 42, 78, 13, 41, 77, 12, 14]$
- Remember: The heap is stored in an array. The tree is *implicit*.
- Thus, all layers except for maybe the last are full.

Max-Heapify:

- It makes an almost-heap into a heap.
  - Almost-heap: only the root of the heap might violates the heap-property
- Pseudocode:

```
procedure Max-Heapify(A, i)
        **turns almost-heap into a heap
        **pre-condition:  tree rooted at A[i] is an almost-heap
        **post-condition:  tree rooted at A[i] is a heap
    lc ← leftchild(i)
    rc ← rightchild(i)
    largest ← i
    if (lc ≤ heapsize(A) and A[lc] > A[largest]) then
        largest ← lc
    if (rc ≤ heapsize(A) and A[rc] > A[largest]) then
        largest ← rc          **largest = index of max{A[i], A[rc], A[lc]}
    if (largest ≠ i) then
        exchange A[i] ↔ A[largest]
        Max-Heapify(A, largest)
```

- WC running time: $O(h) = O(\lg n)$.

Building a heap from an array:

- Given an array of n keys $A[1], A[2], \ldots, A[n]$, permute the keys in A so that A is a heap
- Outline:
  1. Look at the implicit binary tree that A induces
  2. Consider the leafs (the bottom-level nodes in the binary tree):
     Each of them has a single-key $\Rightarrow$ each of them is a heap
  3. Consider the nodes on the second-to-last level:
     The subtrees rooted at these nodes are almost-heaps:
     `Max-Heapify` them into heaps!
  4. Now, consider the nodes on the third-to-last level:
     The subtrees rooted at those nodes are almost-heaps:
     `Max-Heapify` them into heaps!

     $\vdots$
  5. The whole tree becomes an almost heap:
     Max-Heapify tree root into a heap!

     DONE!

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

- Pseudocode:

<u>procedure Build-Max-Heap(A)</u>

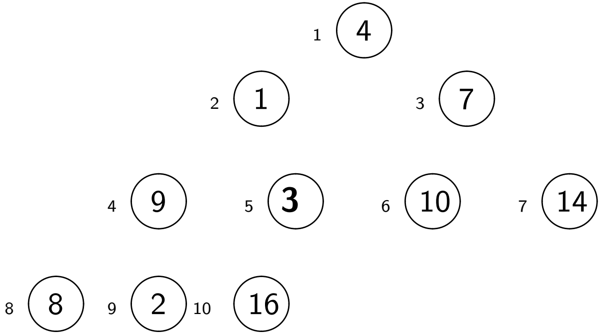
    **turn an array into a heap

  $heapsize(A) \leftarrow length[A]$

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

    Max-Heapify(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$: Max-Heapify($A, 5$):



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

- Pseudocode:

  procedure Build-Max-Heap(A)

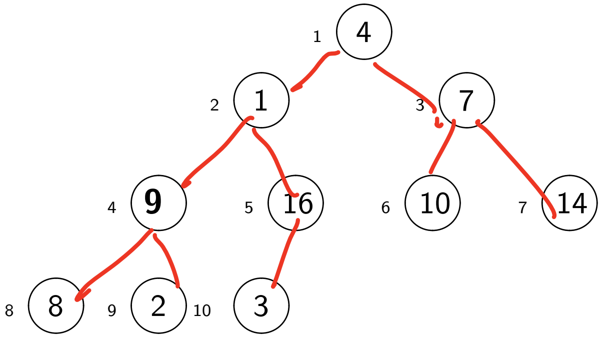
  **turn an array into a heap

  $heapsize(A) \leftarrow length[A]$

  for $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ downto 1

  do Max-Heapify(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ Max-Heapify$(A, 4)$:

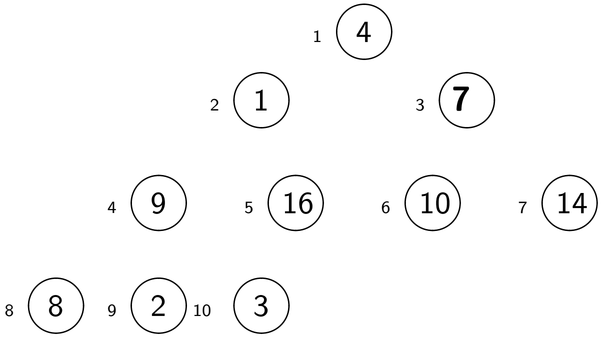


Building a heap from an array (cont'd):

- Pseudocode:

  `procedure Build-Max-Heap`(A)
          `**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)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ `Max-Heapify`$(A, 3)$:

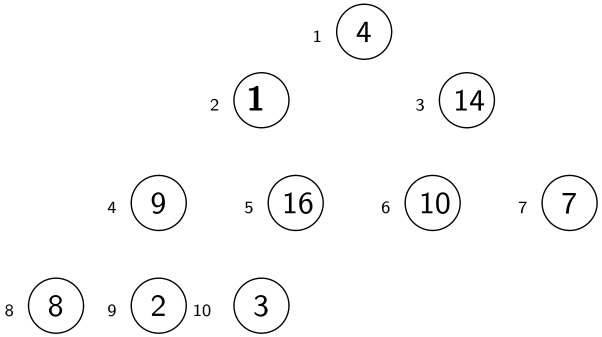


Building a heap from an array (cont'd):

- Pseudocode:

procedure Build-Max-Heap(A)
    **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)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ Max-Heapify$(A, 2)$:

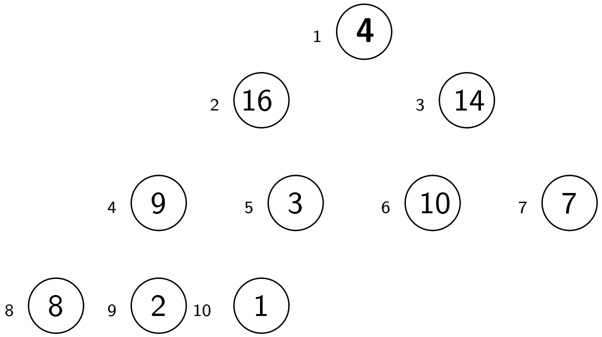


Building a heap from an array (cont'd):

- Pseudocode:

procedure Build-Max-Heap(A)
  **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)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ Max-Heapify$(A, 1)$:

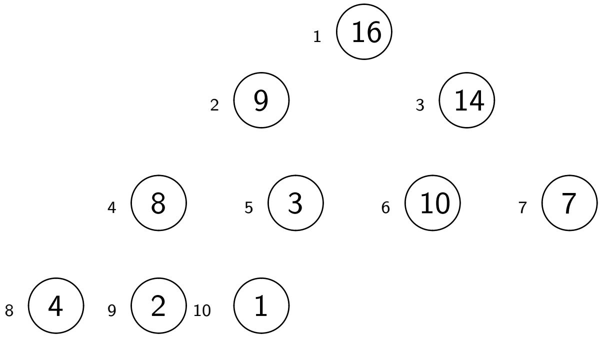


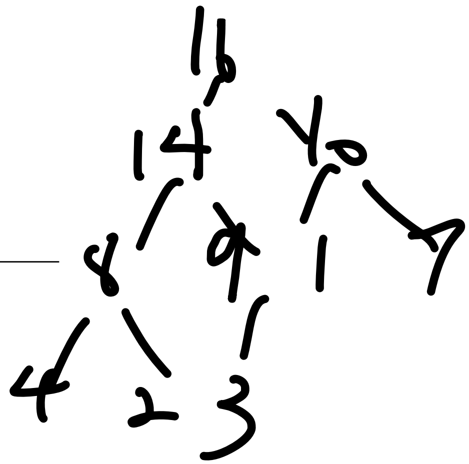
Building a heap from an array (cont'd):

- Pseudocode:

  procedure `Build-Max-Heap`(A)
          `**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)

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





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

- Pseudocode:

procedure Build-Max-Heap(A)
    **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)

- Worst case running time: because we make at most $\frac{n}{2}$ calls to Max-Heapfiy, each takes $O(\lg(n))$ we have $O(n \log(n))$.

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

- Correct bound is $O(n)$:
- `Max-Heapify`'s runtime is $O(k)$ for a node at height k.
  - At height 1 we have at most $n/2$ nodes.
  - At height 2 we have at most $n/4$ nodes.
  - At height 3 we have at most $n/8$ nodes.
  - ...
  - At height $\lg(n)$ we have at most 1 node.
- So runtime is upper bounded by:

$$\begin{aligned}
\sum_{k=1}^{\lg(n)} k \cdot \frac{n}{2^k} &= n \sum_{k=1}^{\lg(n)} \frac{k}{2^k} = n\left(\sum_{k=1}^{3} \frac{k}{2^k} + \sum_{k=4}^{\lg(n)} \frac{k}{2^k}\right) \le n\left(\sum_{k=1}^{3} \frac{k}{2^k} + \sum_{k=4}^{\lg(n)} \frac{2^{k/2}}{2^k}\right) \\
&= n\left(\sum_{k=1}^{3} \frac{k}{2^k} + \sum_{k=4}^{\lg(n)} \frac{1}{2^{k/2}}\right) \le n\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \sum_{k=4}^{\infty} \frac{1}{2^{k/2}}\right) \\
&\le n\left(2 + \sum_{k=0}^{\infty} \left(\sqrt{\frac{1}{2}}\right)^k\right) = n\left(2 + \frac{1}{1-\sqrt{\frac{1}{2}}}\right) \le n\left(2 + \frac{1}{1-0.75}\right) \le 6n
\end{aligned}$$

4

2 3

- Tighter analysis will yield running time is actually $2n - \lg n - 2$.

Heapsort algorithm:

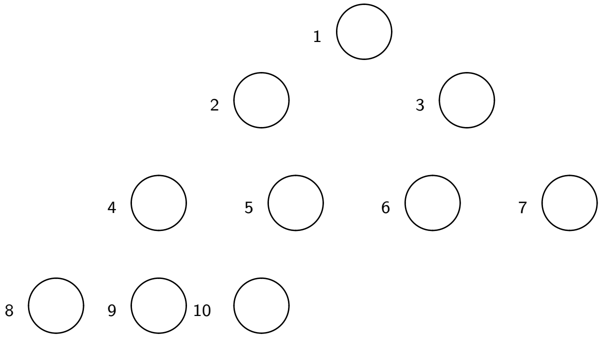
- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

4

L 3

2 3 4

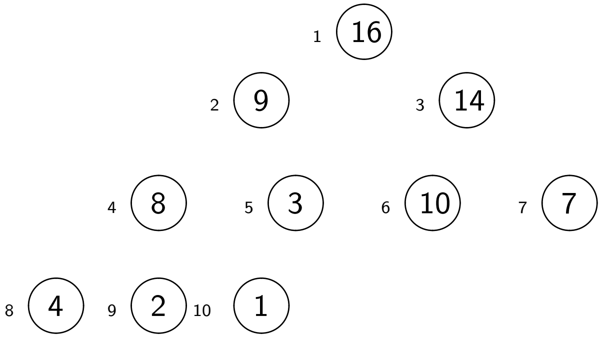
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Build into a heap:



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- 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, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

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



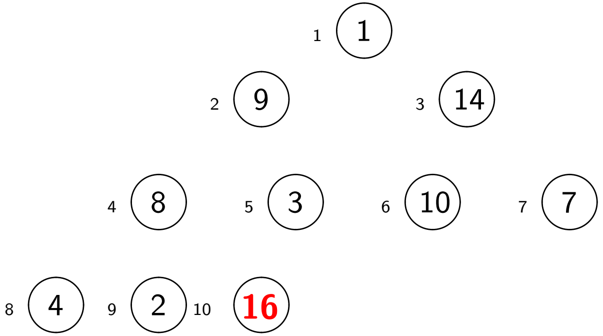
16

16

Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- 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, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

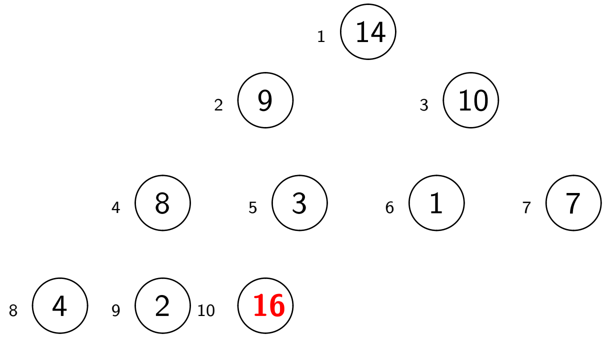
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Exchange $A[1]$ and $A[10]$, decrement Heapsize to 9, and Max-Heapify it (restore the heap property):



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- 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, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

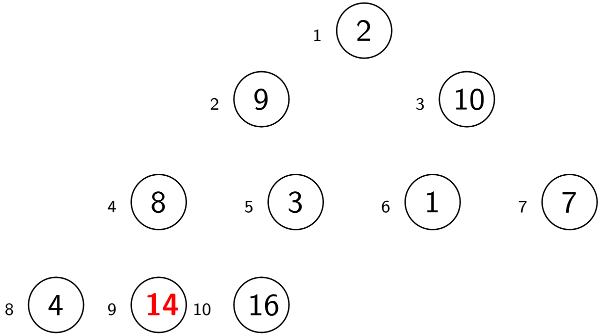
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Resultant tree: Heapsize $= 9$:



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- 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, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

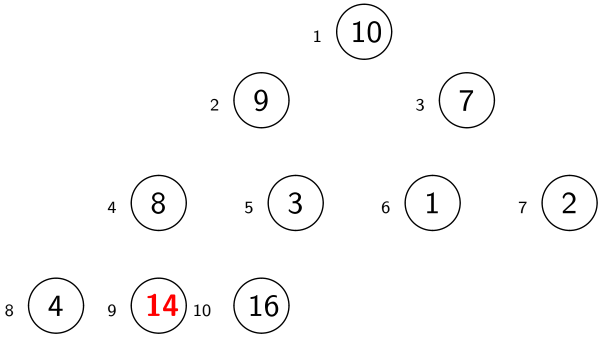
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Exchange $A[1]$ and $A[9]$, decrement Heapsize to 9, and Max-Heapify it (restore the heap property):



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- 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, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Resultant tree: Heapsize $= 8$:



2

3 4

Heapsort algorithm (cont'd):

- Pseudocode:

```
procedure Heapsort(A)
        **post-condition:  sorted array
    Build-Max-Heap(A)
    for (i ← heapsize(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 $O(n)$
  - For each position $i = n, n-1, ..., 2$, Max-Heapify takes $O(\lg i)$, so in total this is $\Theta(n \log(n))$.

$$\sum_{i=2}^{n} \log(i) \leq \sum_{i=1}^{n} \log(n) = n\log(n)$$

$$\sum_{i=2}^{n} \log(i) \geq \sum_{i=\lceil n/2 \rceil}^{n} \log(n/2) = \lfloor \tfrac{n}{2} \rfloor \cdot (\log(n) - 1) \geq \tfrac{1}{3} n \log(n)$$

  - So, in total $\Theta(n \log n)$

Heapsort algorithm — Conclusion:

- WC running time:
  - Build-Max-Heap takes $O(n)$.
  - $n-1$ calls to Max-Heapify: $O(n \log n)$.
  - Overall $O(n \log n)$.
  - In the worst-case, It is easy to see that Max-heapify can take $\Omega(\log n)$.
  - Thus the WC running time is also $\Omega(n \log n)$.
  - Total: $\Theta(n \log n)$.
- Correctness – prove on your own:
  - Correctness for Max-Heapify?
    (a recursion, use induction on height of i)
  - LI for Build-Max-Heap?
    For any $j \geq i$, the subtree rooted at j is heap (CLRS p.157)
  - LI for heapsort
    $A[1, ..., i]$ is a heap & $A[i+1, ...n]$ contains the $n-i$ largest keys, sorted.

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 !!!

1

2 3

4 5 6

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) —
  Like deleting from an array: put $A[n]$ as the new first element before returning the max.
  The difference: we `Max-Heapify`$(A, 1)$ to make this array into a heap.
  $\Theta(\lg n)$ time.

  procedure `Heap-Extract-Max`(A)

  $**$ precondition: A isn't empty
  $max \leftarrow A[1]$
  $A[1] \leftarrow A[heapsize[A]]$
  $heapsize[A] \leftarrow heapsize[A] - 1$
  if $(heapsize[A] > 0)$ **then**
      `Max-Heapify`$(A, 1)$
  return max
- `Increase-Key`(A, i, new_key)
- `Insert`(A, new_key)

1 2

Priority Queue:

- `Initialize`(A)
- `Maximum`(A)
- `Extract-Maximum`(A)
- `Increase-Key`(A, i, new_key) — The inverse of `Max-Heapify`: Increase the priority value for $A[i]$ and bubble up to till max-heap property is restored. $\Theta(\lg n)$ time.

  procedure Heap-Increase-Key(A, i, key)

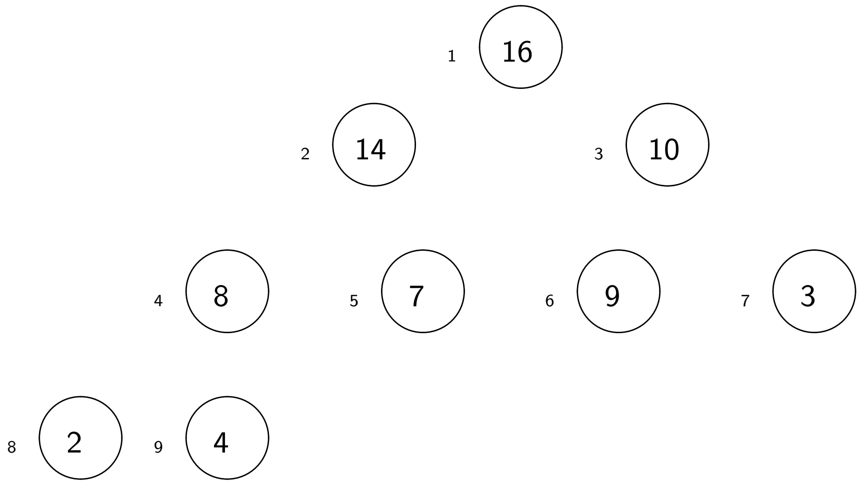
  $\quad$ ** Precondition: $key \geq A[i]$
  $\quad$ $A[i] \leftarrow key$
  $\quad$ while $(i > 1$ and $A[Parent(i)] < A[i])$ do
  $\quad\quad$ exchange $A[i] \leftrightarrow A[Parent(i)]$
  $\quad\quad$ $i \leftarrow Parent(i)$

- `Insert`(A, new_key) — Add a new key with lowest priority, increase its priority to new_key. $\Theta(\lg n)$ time.

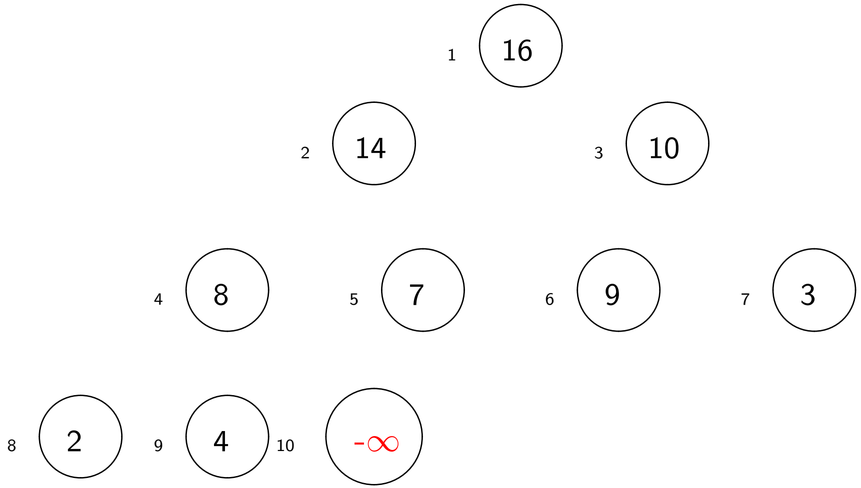
  procedure Heap-Insert(A, key)

  $\quad$ $heapsize[A] \leftarrow heapsize[A] + 1$
  $\quad$ $A[heapsize[A]] \leftarrow -\infty$ $\quad$ ** or any value smaller than all keys in A
  $\quad$ Heap-Increase-key $(A, heapsize[A], key)$

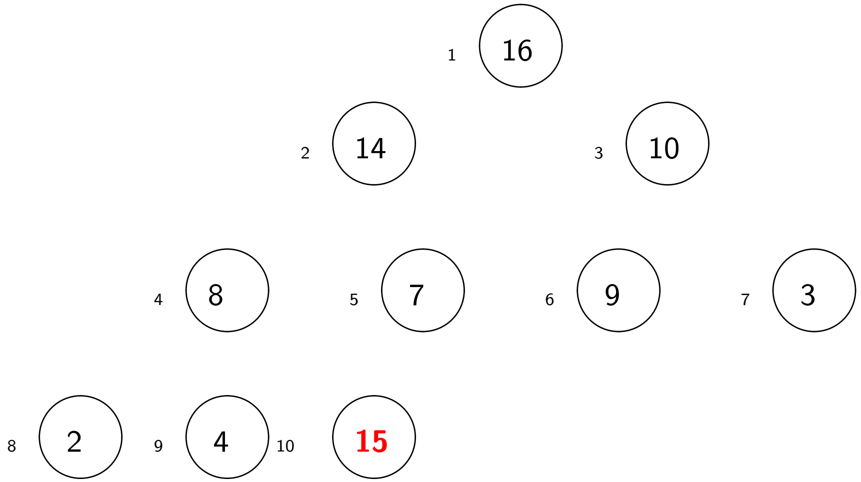
Starting with a heap:



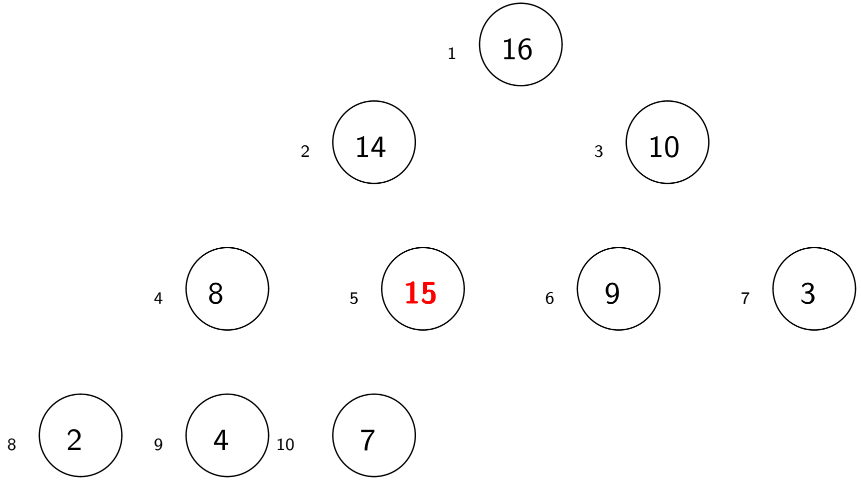
Max-heap-Insert $(A, 15)$:



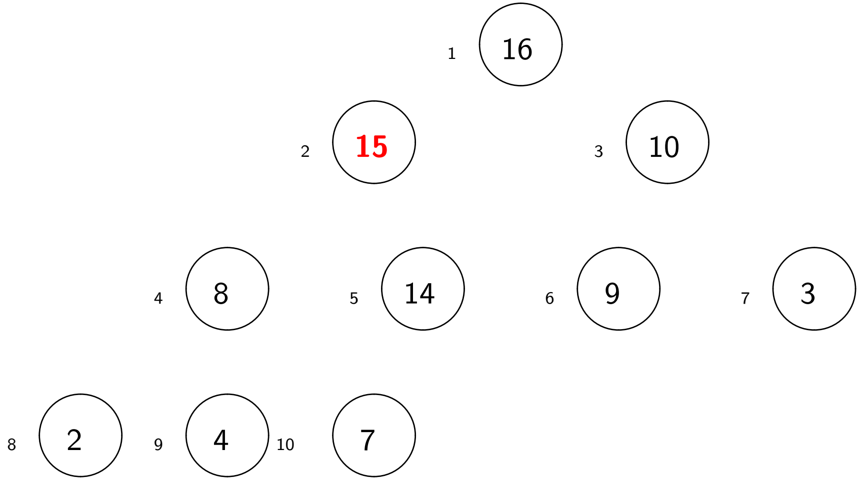
Heap-Increase-Key $(A, heapsize[A], 15)$ is called:

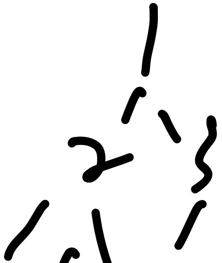


Bubbling up key 15:



Bubbling up key 15:





& 6

Priority Queue:

- `Initialize`(A) — Build-Max-Heap. Takes $\Theta(n)$ time.
- `Maximum`(A) — Return $A[1]$. Takes $\Theta(1)$ time.
- `Extract-Maximum`(A) — Like deleting from an array: put $A[n]$ as the new first element before returning the max. The difference: we `Max-Heapify`$(A, 1)$ to make this array into a heap. $\Theta(\lg n)$ time.
- `Increase-Key`(A, i, new_key) — The inverse of `Max-Heapify`: Increase the priority value for $A[i]$ and bubble up to till max-heap property is restored. $\Theta(\lg n)$ time.
- `Insert`(A, new_key) — Add a new key with lowest priority, increase its priority to new_key. $\Theta(\lg n)$ time.

- Note that we didn't mention `Decrease-key`(A, i, new_key). Why?
- Because we already know how to deal with it.
  Once i's key is set to a new *smaller* value, then the subheap rooted at i becomes an almost-heap.
  Run `Max-Heapify`(A, i).

Sorting on the Fly

- So far — we have considered the notion of a static problem: someone gives you an array of n items and we have to sort them.
- However, the problem can be studied also in the dynamic setting: the set of items changes, keys are added and removed (inserted and deleted), and every now and then, we wish to sort them (or find a value x among them).
- Option 1: use a data-structure that does insertion and deletion fast (array, list, hash-table), and upon a sorting request - run a sorting algorithm.
- Option 2: use a data-structure that keeps the elements sorted. (a binary search tree)
- Which is the better option: depends on the sequence of calls.
  If a `find()/sort()` request comes once in a long while, after many insertion/deletions, then use option 1.
  If there are many `find()/sort()` requests, or the they appear after the insertion/deletion of only a few elements — use option 2.