Week 4: Heapsort, Quicksort

Agenda:

- Heaps:
  - Max-Heapify
  - Build-Max-Heap
  - Heapsort
- Priority Queues
- Quicksort

Reading:

- Text: 127-158

Heaps 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)
      **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: $\lg n$.

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!

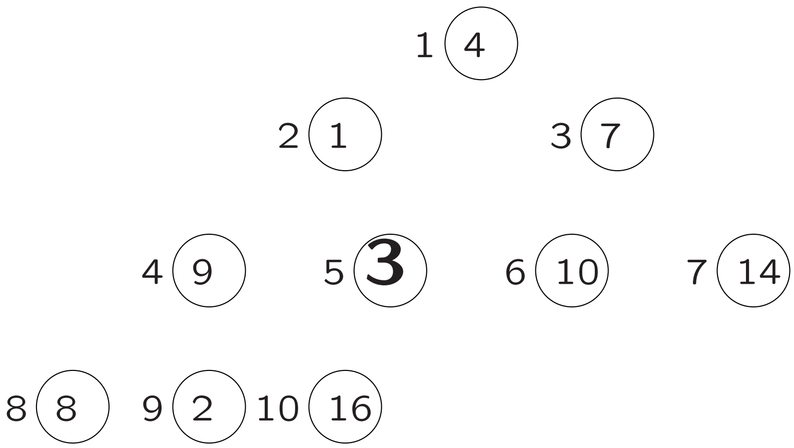
Building a heap from an array (cont'd):

- Pseudocode:

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

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

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



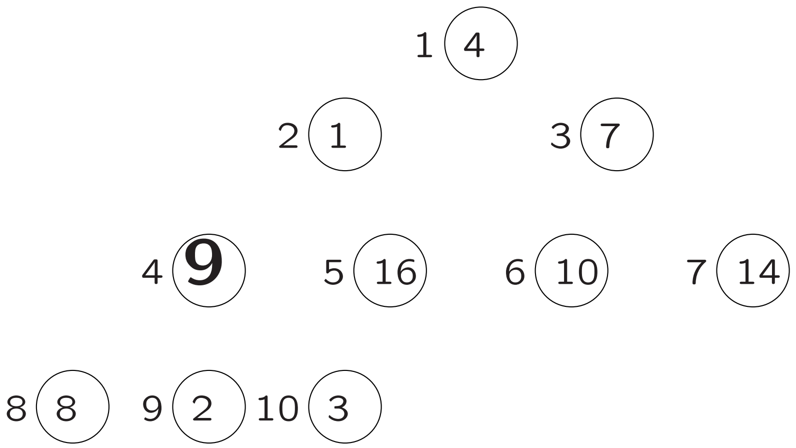
Building a heap from an array (cont'd):

- Pseudocode:

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

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



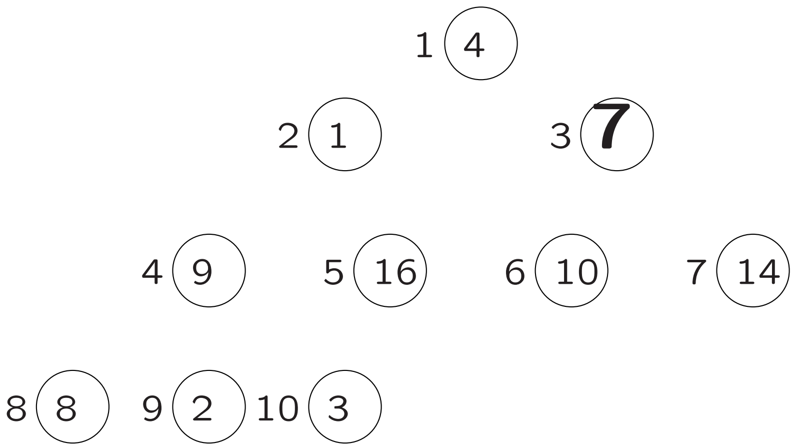
Building a heap from an array (cont'd):

- Pseudocode:

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

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



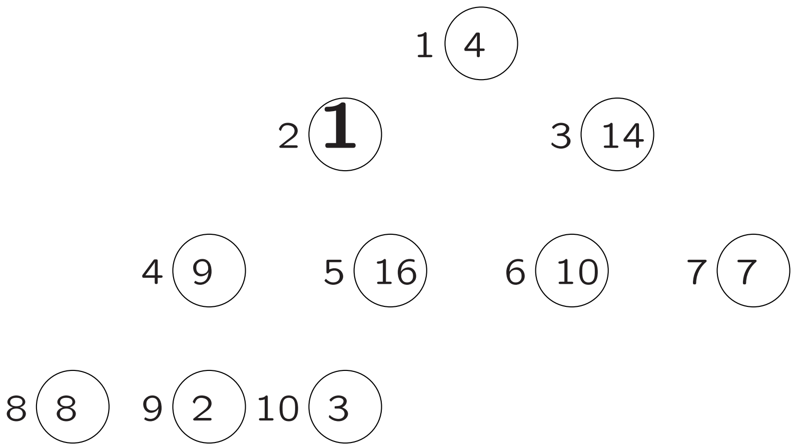
Building a heap from an array (cont'd):

- Pseudocode:

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

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



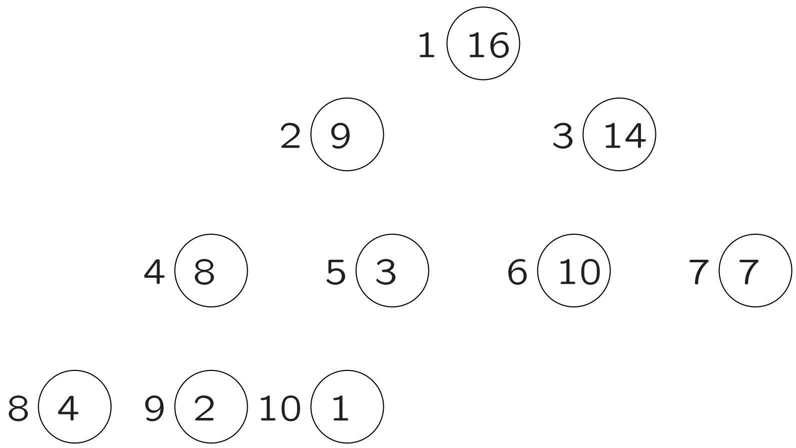
Building a heap from an array (cont'd):

- Pseudocode:

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

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



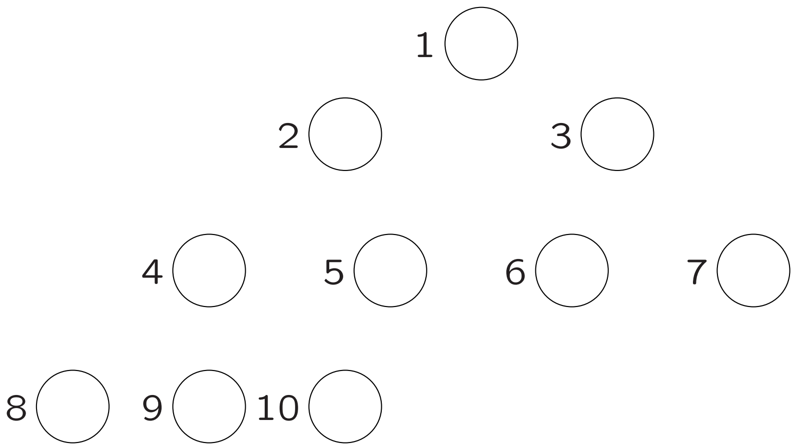
- Worst case running time: Easy guess $O(n \log n)$, because at most $\frac{n}{2}$ calls to Max-Heapfiy.

- Correct bound: $O(n)$ because about half of the calls (for the lowest level) take only $O(1)$ time, for about quarter of them the height is 2 which is $O(1)$, etc.

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

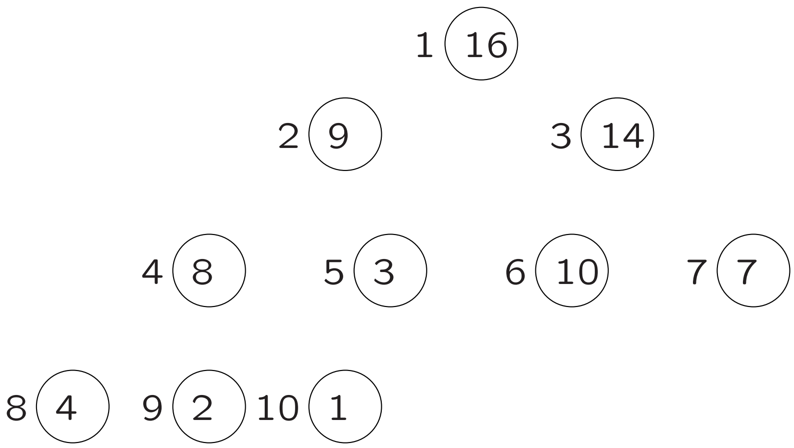
  Build into a heap:



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

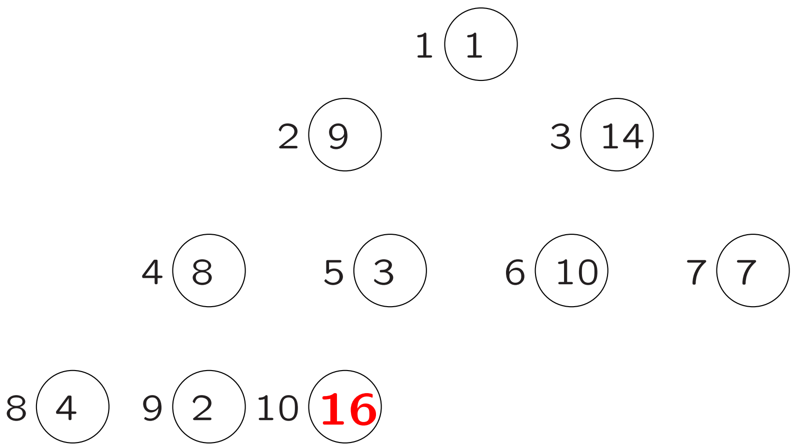
  Heapsize = 10:



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

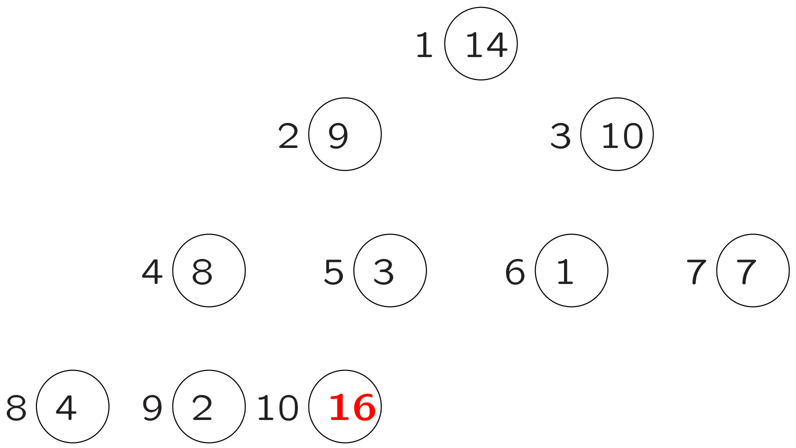
  Exchange $A[1]$ and $A[10]$, decrement Heapsize to 9, and Max-Heapify it (re-install the heap property):



Heapsort algorithm (cont'd):

- 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
- 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)
      **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):

- WC running time:
  - Buil-Max-Heap takes $O(n)$.
  - $n-1$ calls to Max-Heapify: $O(n \log n)$.
  - Overall $O(n \log n)$.
  - It is not hard to show that Max-heapify can take $\Omega(\log n)$.
  - Thus the WC running time is also $\Omega(n \log n)$.
  - Total: $\Theta(n \log n)$.
- BC running time analysis:
  - all keys distinct: $\Theta(n \log n)$ (not proved here).
  - So WC and BC are both $\Theta(n \log n)$.
- Correctness:
  By Loop Invariants:
  - correctness for Max-Heapify (which is a recursion)
  - LI for Build-Max-Heap (p. 133 from the handout)
  - LI for heapsort.

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 $\Longleftrightarrow$ rearrangement of elements.

- `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)
    if heap − size[A] < 1 then Error ``heap underflow''
    max ← A[1]
    A[1] ← A[heap − size[A]]
    heap − size[A] ← heap − size[A] − 1
    Max-Heapify(A, 1)
    return max
```

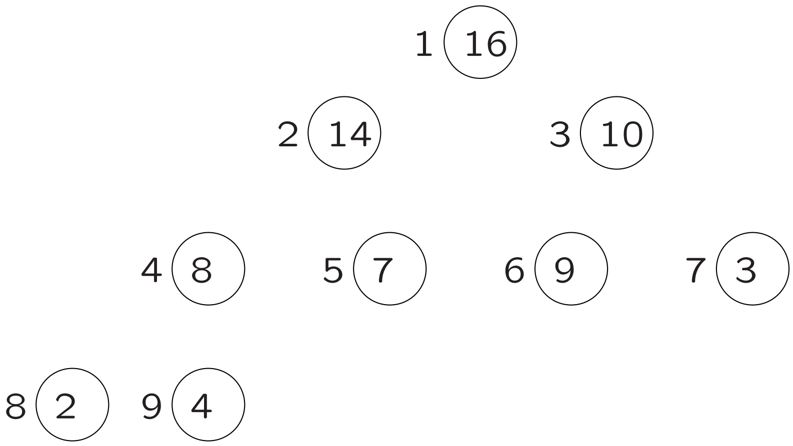
- `Increase-Key`(A, i, new_key) — increase the priority value for $A[i]$ and bubble up to the right position. $\Theta(\lg n)$ time.

```
procedure Heap-Increase-Key(A, i, key)
    if key < A[i] then Error ``new key is smaller''
    A[i] ← key
    while i > 1 and A[Parent(i)] < A[i] do
        exchange A[i] ↔ A[Parent(i)]
        i ← Parent(i)
```

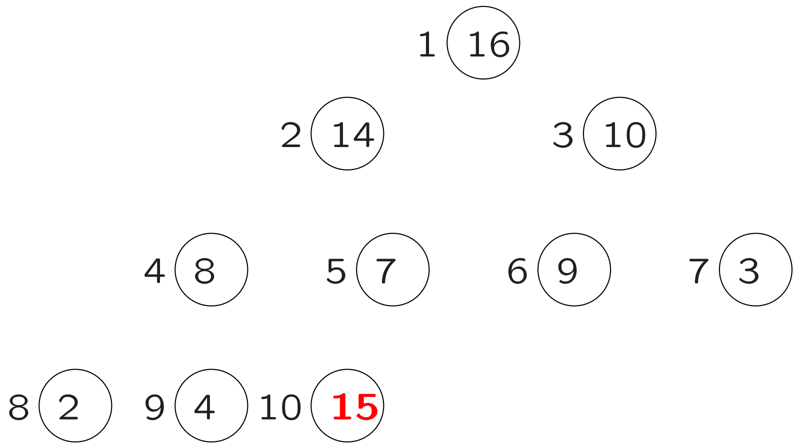
- `Insert`(A, new_key) — increase heapsize, add a new key with priority value equal to new_key, and bubble up. $\Theta(\lg n)$ time.

```
procedure Heap-Insert(A, key)
    heap − size[A] ← heap − size[A] + 1
    A[heap − size[A]] ← −∞
    Heap-Increase-key (A, heap − size[A], key)
```

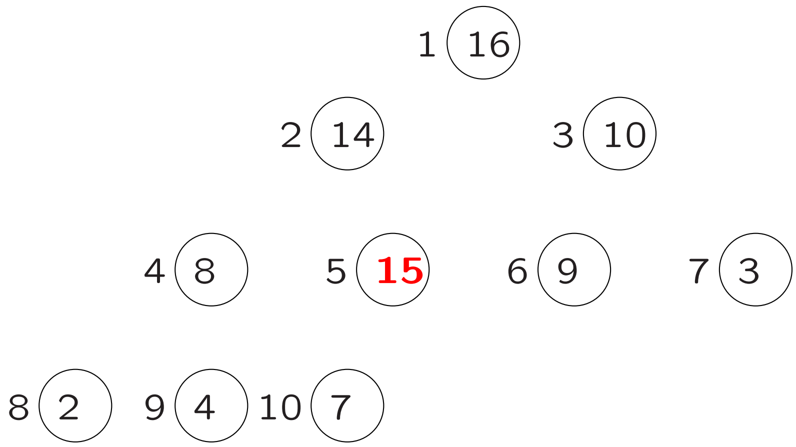
Starting with a heap:



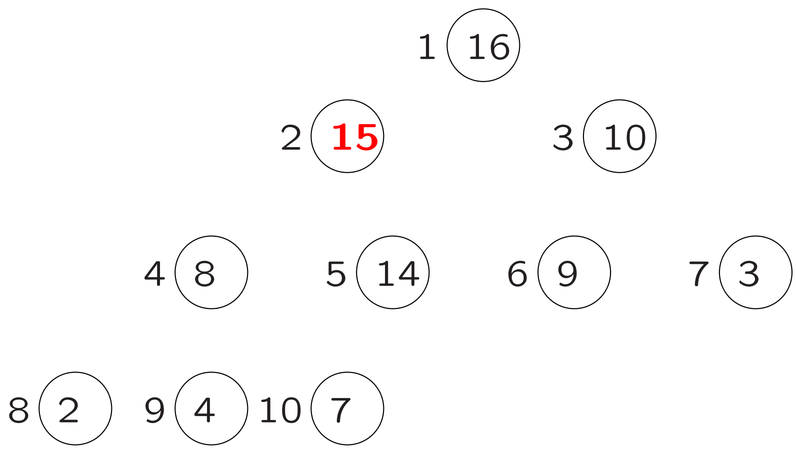
Max-heap-Insert $(A, 15)$:



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



Bubbling up key 15:



Quicksort: 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)

    if p < r then
        q ← Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)


procedure Partition(A, p, r)
    ** A[r] is the key picked to do the partition

    x ← A[r]
    i ← p − 1
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            exchange A[i] ↔ A[j]
    exchange A[i + 1] ↔ A[r]
    return i + 1
```

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	$i = 4, j = 7$

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	$i = 4, j = 7$