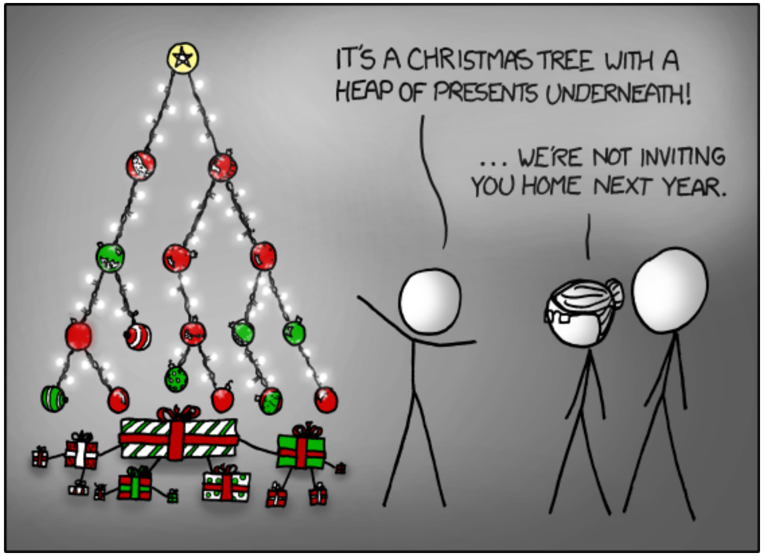
Priority queues and Heaps

Carlos Moreno

cmoreno@uwaterloo.ca

EIT-4103



http://xkcd.com/835/

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Priority queues and Heaps

Standard reminder to set phones to silent/vibrate mode, please!

Priority queues and Heaps

- During today's lesson:
  - Introduce the notion of priority queues
  - Consider some “obvious” implementations (an array of queues; a balanced binary search tree)
  - Introduce the Heap data structure, which provides a more efficient implementation alternative for priority queues
    - Discuss the various operations on a heap and their run time.

Priority queues and Heaps

- Priority queues are rather easy to define:
  - As the name suggests, they're queues where the elements have priorities associated to them.
  - We could look at it by analogy with real-life examples of FIFO structures: a line waiting to be served at a bank (or for the cash registers at a store, etc.)
    - It makes sense that they will serve first those who have been waiting the longest.

Priority queues and Heaps

- Priority queues are rather easy to define:
  - As the name suggests, they're queues where the elements have priorities associated to them.
  - We could look at it by analogy with real-life examples of FIFO structures: a line waiting to be served at a bank (or for the cash registers at a store, etc.)
    - It makes sense that they will serve first those who have been waiting the longest.
      - Except if, for example, a senior or a disabled person arrives in the line — either by policy or by simple courtesy, the common practice is: even if they arrived after, they are served first.

Priority queues and Heaps

- In this example, we're all following the scheme of “first-arrive-first-served” — but seniors have higher priority than non-seniors.
  - So, as long as there are seniors in line, they will be served first, no matter how long we've been waiting, and regardless of whether a senior person arrived just two seconds ago.
- Disabled persons presumably have higher priority than seniors — same principle.

Priority queues and Heaps

- We could visualize this as a set of queues:
  - Disabled persons have a designated line; seniors have a designated, separate, line; and the rest have a separate line.

Priority queues and Heaps

- We could visualize this as a set of queues:
  - Disabled persons have a designated line; seniors have a designated, separate, line; and the rest have a separate line.
  - The serve protocol is: check first the line for disabled persons — if there is someone, serve the first one in line there. If no-one in line, then check the line for seniors; and so on, until checking the line with lowest priority.

Priority queues and Heaps

- This automatically suggests an obvious implementation strategy:
  - Use an array of queues.
  - Assign a non-negative integer number to represent the priority: 0 represents the highest priority; the larger the number, the lower the priority.
  - Use the priority as the subscript for the array (to get to the corresponding queue)

Priority queues and Heaps

- Two disadvantages:
  - Limited — it is feasible (in a reasonable way) if we have a fixed number of priorities (good enough for many applications, but not good enough — mostly because we *can* do better than that)

Priority queues and Heaps

- Two disadvantages:
  - Limited — it is feasible (in a reasonable way) if we have a fixed number of priorities (good enough for many applications, but not good enough — mostly because we *can* do better than that)
  - Not efficient
    - Can you see why?

Priority queues and Heaps

- Two disadvantages:
  - Limited — it is feasible (in a reasonable way) if we have a fixed number of priorities (good enough for many applications, but not good enough — mostly because we *can* do better than that)
  - Not efficient
    - Can you see why?
      - If we have *m* priorities, then we have an array of *m* queues, and looking for the "next-in-line" takes $\Theta(m)$

Priority queues and Heaps

- We could turn this into a sorting scheme by thinking of the queue as a sorted list, where we sort by two criteria:
  - First, by priority
  - Then, for equal priorities, sort by arrival order

- This is a lexicographical order ... right? (why?)

Priority queues and Heaps

- If we keep a counter *k* and increase it every time we insert an element, then the pair (*p*,*k*), where *p* is the priority, provides the appropriate order:

$$(p_{1,} k_1) < (p_{2,} k_2) \iff \begin{array}{l} p_1 < p_2 \quad \text{or} \\ p_1 = p_2 \ \text{ and } \ k_1 < k_2 \end{array}$$

Priority queues and Heaps

- With this, we could simply use a balanced binary search tree (e.g., an AVL tree) using that pair as the value being inserted.
- An AVL tree maintains the elements in order with insertions and removals taking logarithmic time.
- However, the implementation is more complicated than it could be, as we'll see next, when looking into Heaps.

Priority queues and Heaps

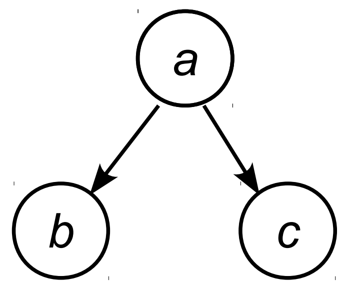
- Heaps are a particular type of binary trees.
- We'll provide a recursive definition:
- A binary tree of height 0 is a heap.
- A non-empty binary tree is a heap if:
  - The root node is less than the values in either of the sub-trees (if present).
  - Both sub-trees are themselves heaps.

Priority queues and Heaps

- An alternative way to phrase that is:
  - A non-empty binary tree is a heap if for every internal (non-leaf) node, every strict descendant is greater than the node.

Priority queues and Heaps

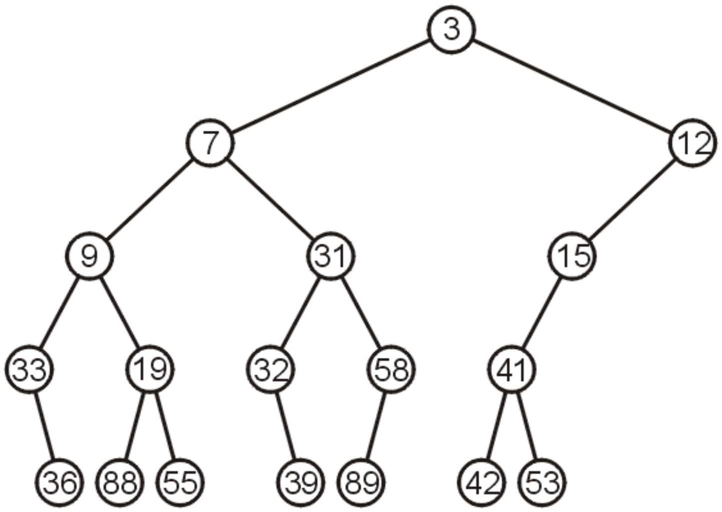
- Important "fine print" in that definition:
  - Sibling elements — or in general elements in the two sub-trees have NO RELATIONSHIP WHATSOEVER!!



We know that $b < a$ and that $c < a$; that says absolutely nothing about b as compared to, or related to, c.

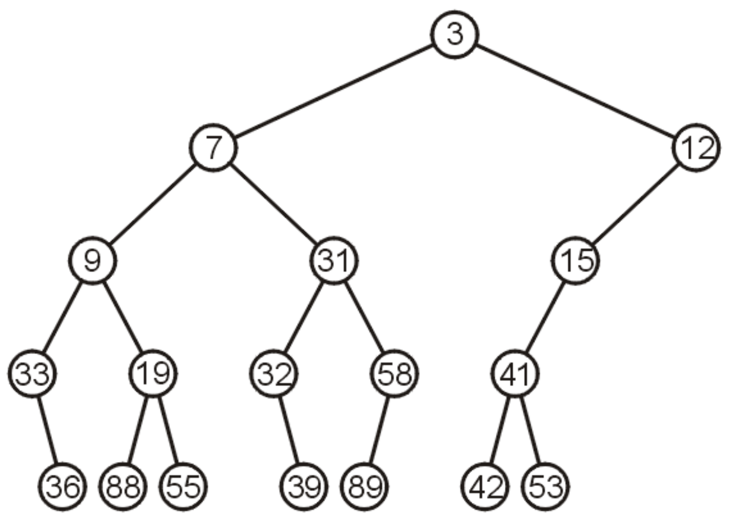
Priority queues and Heaps

- This is an example of a heap:



Priority queues and Heaps

- This is an example of a heap:
- We have to keep this notion completely apart from the notions of binary search trees. For example:
  - The smallest value (7) and the largest value (89) are both in the left sub-tree.



Priority queues and Heaps

- We can obviously find the lowest value in constant time: right? how?

Priority queues and Heaps

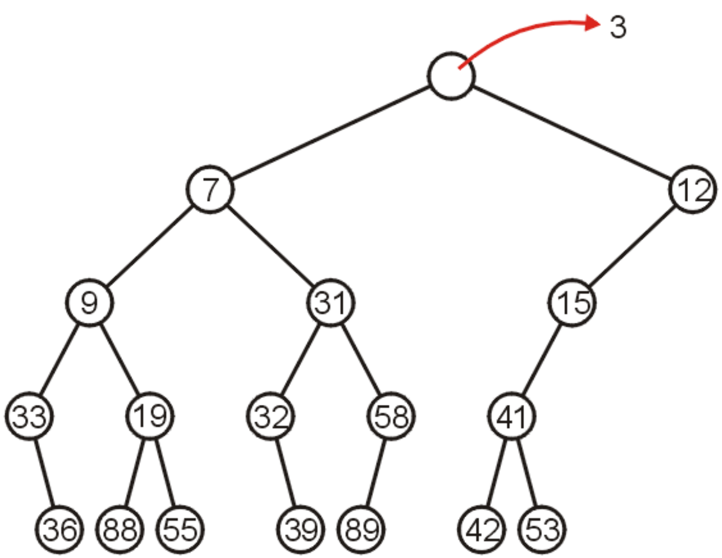
- We can obviously find the lowest value in constant time: right? How?
  - Yes, this one is too obvious to bother writing the answer in the slides :-)

Priority queues and Heaps

- Removing the lowest element (which is the operation corresponding to “serve the next in line”) is rather simple, and presumably efficient:
  - Promote the node at the root of the sub-tree which has the least value.
  - Repeat the same for that sub-tree, all the way down until reaching a leaf node.

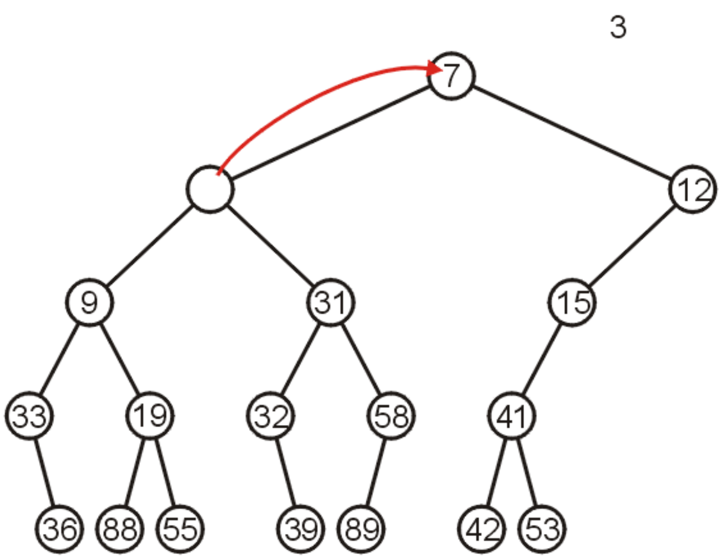
Priority queues and Heaps

- An example:



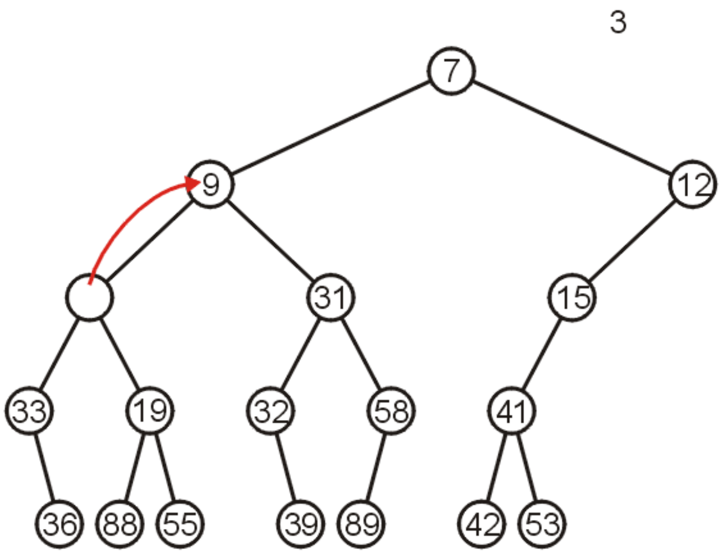
Priority queues and Heaps

- An example:



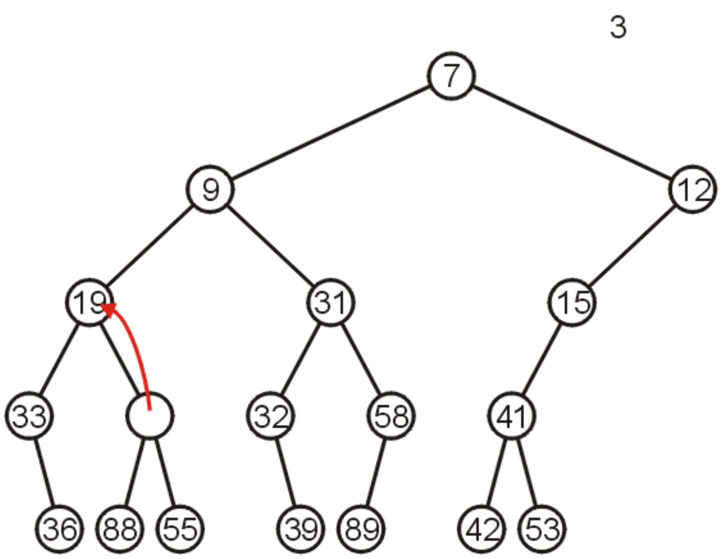
Priority queues and Heaps

- An example:



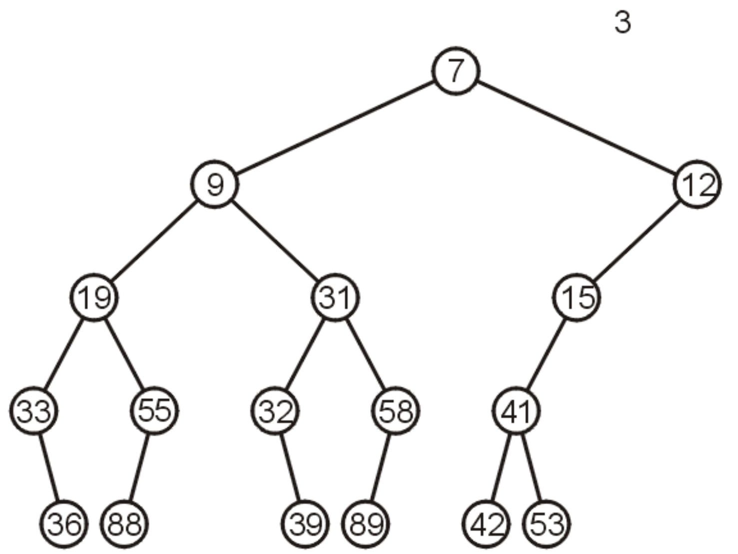
Priority queues and Heaps

- An example:



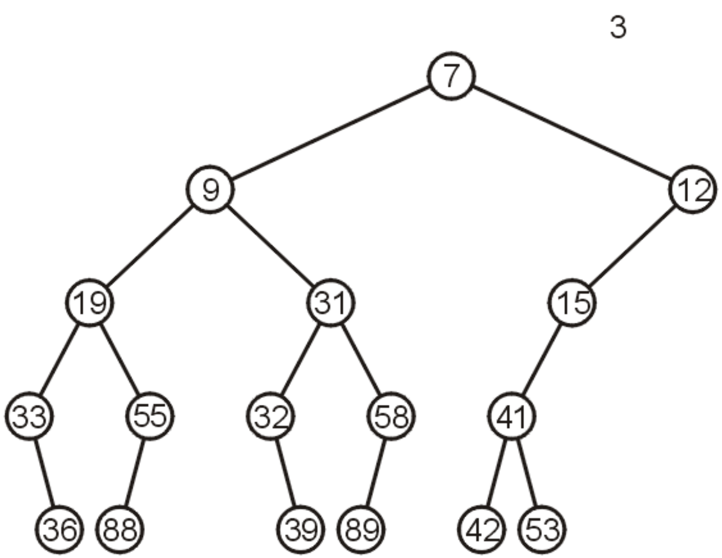
Priority queues and Heaps

- An example:



Priority queues and Heaps

- Question: why is it efficient? (why do I say *presumably*?)



Priority queues and Heaps

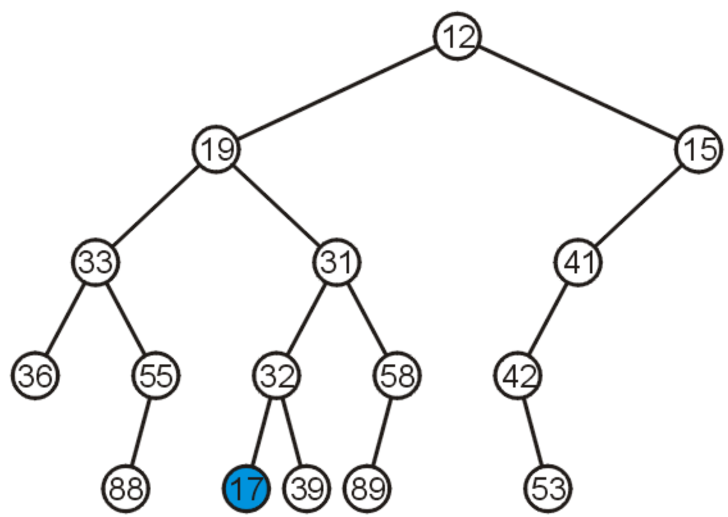
- Question: why is it efficient? (why do I say *presumably*?)
  - As you may have noticed, removal takes O(h), and *presumably*, h is small (right? why? And again, why *presumably*?)

Priority queues and Heaps

- Inserting an element can also be presumably efficient:
  - Create a leaf node with the inserted value, and then adjust.

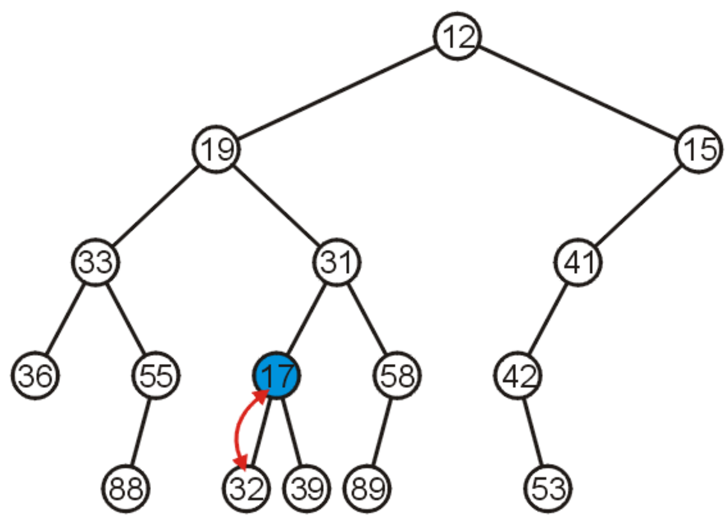
Priority queues and Heaps

- Let's look at an example — inserting 17 in the heap below:



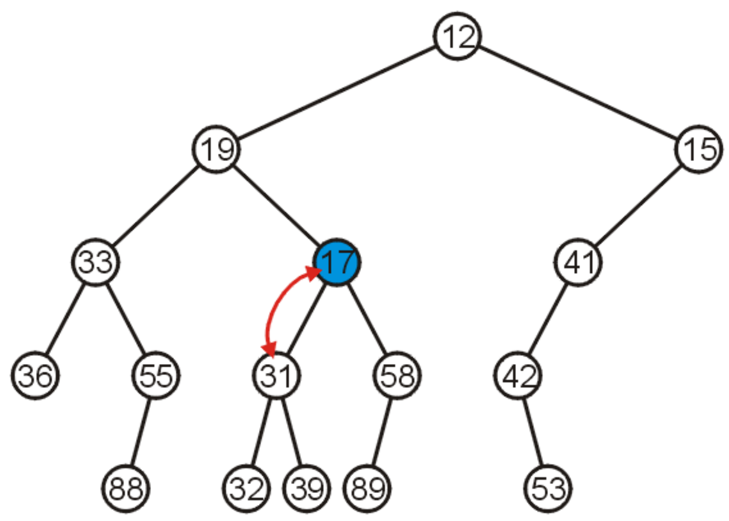
Priority queues and Heaps

- 17 < 32, so we need to swap them:



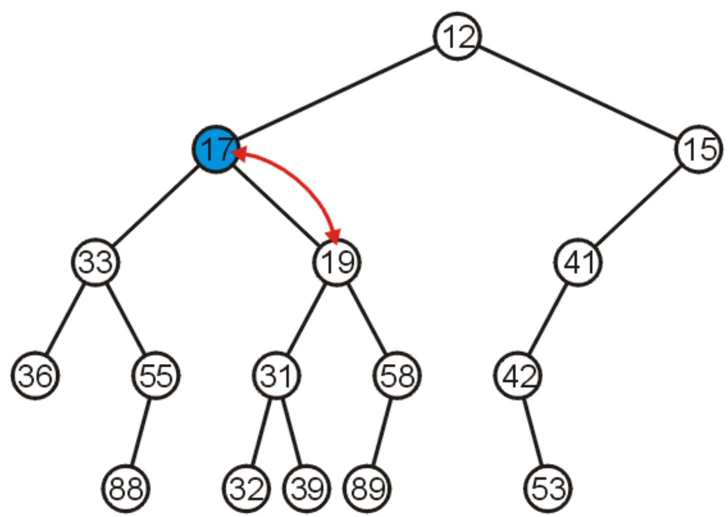
Priority queues and Heaps

- It is also < 31, so we swap these as well:



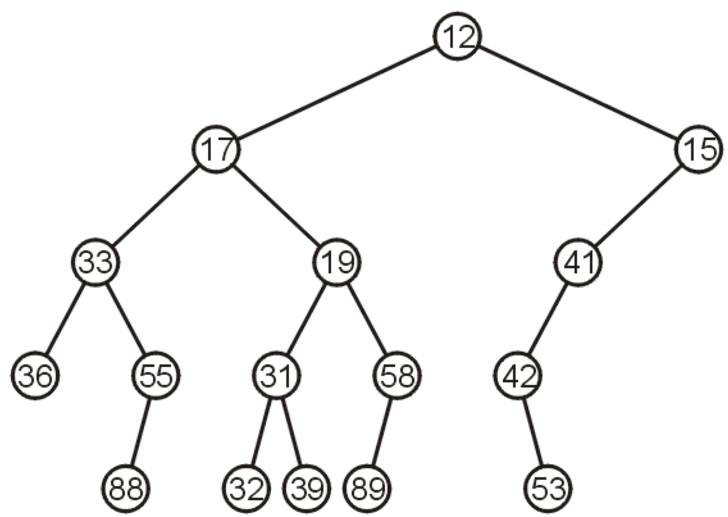
Priority queues and Heaps

- < 19, so we swap, and we're done, since 17 > 12:



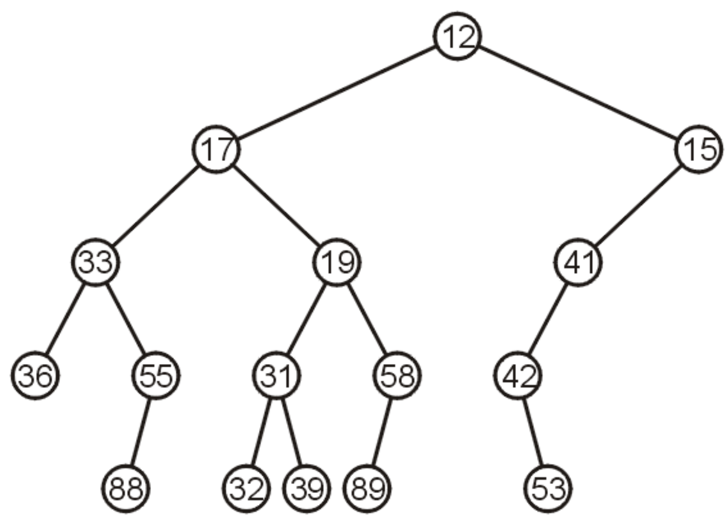
Priority queues and Heaps

- Notice that when swapping down a node, we don't need to check anything further for that node — for example, 19 was already less than anything in that sub-tree, so it can not need any further swaps:



Priority queues and Heaps

- BTW, this process is called *percolation* — the heavier (higher) elements “percolate” down:



Priority queues and Heaps

- However ....

Priority queues and Heaps

- However .... we want to do insertions in a way that maintains a balanced tree!

Priority queues and Heaps

- However .... we want to do insertions in a way that maintains a balanced tree!
  - One rather neat way to do this is ensuring that we always have a *complete* binary tree!
    - (and BTW, when we say a complete binary tree, this carries a piece of good news with it ... right?)

Priority queues and Heaps

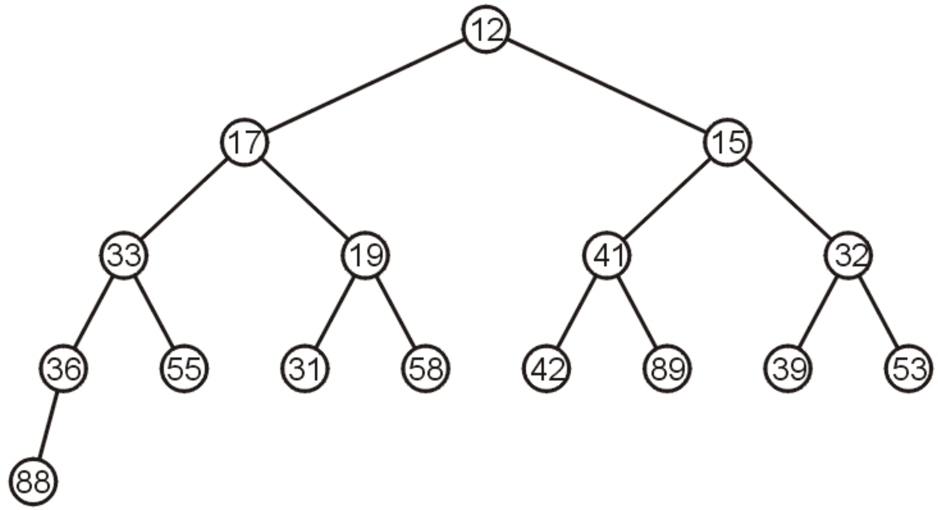
- However .... we want to do insertions in a way that maintains a balanced tree!
  - One rather neat way to do this is ensuring that we always have a *complete* binary tree!
    - (and BTW, when we say a complete binary tree, this carries a piece of good news with it ... right?)
  - So, we insert a leaf node, and adjust (you recall that with complete trees, you can only insert at the position following a “breadth-first” traversal — or rather, *as if* we were doing a breadth-first traversal)

Priority queues and Heaps

- BTW, the fact that we can get away with ensuring a complete binary tree is one of the advantages over a balanced binary search tree such as an AVL — maintaining a complete binary tree ensures balance with *much* lower overhead than that of a general balanced BST such as AVL trees.

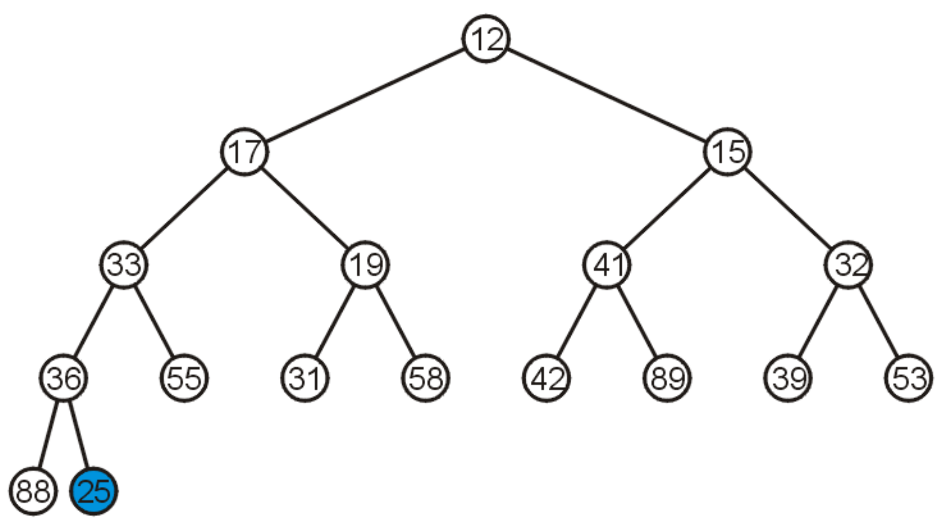
Priority queues and Heaps

- Example: let's try inserting 25 into the following heap:



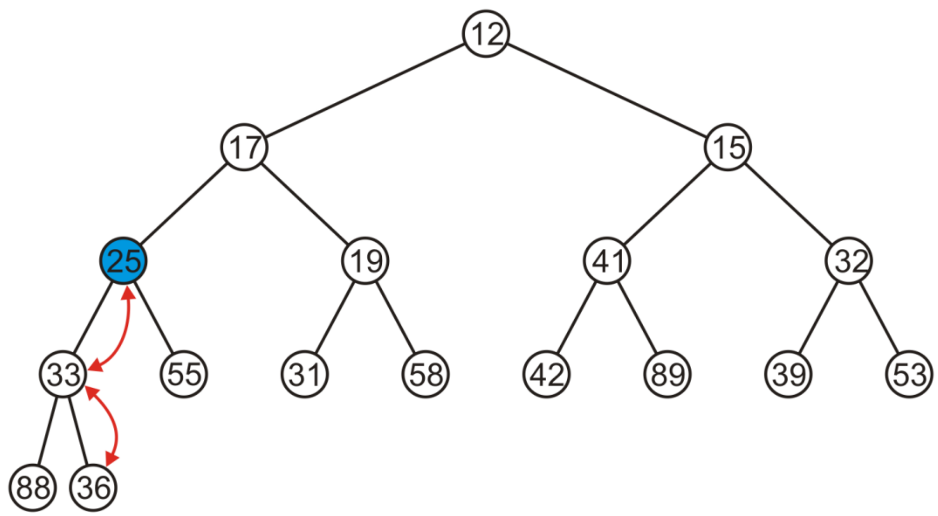
Priority queues and Heaps

- Example: let's try inserting 25 into the following heap:



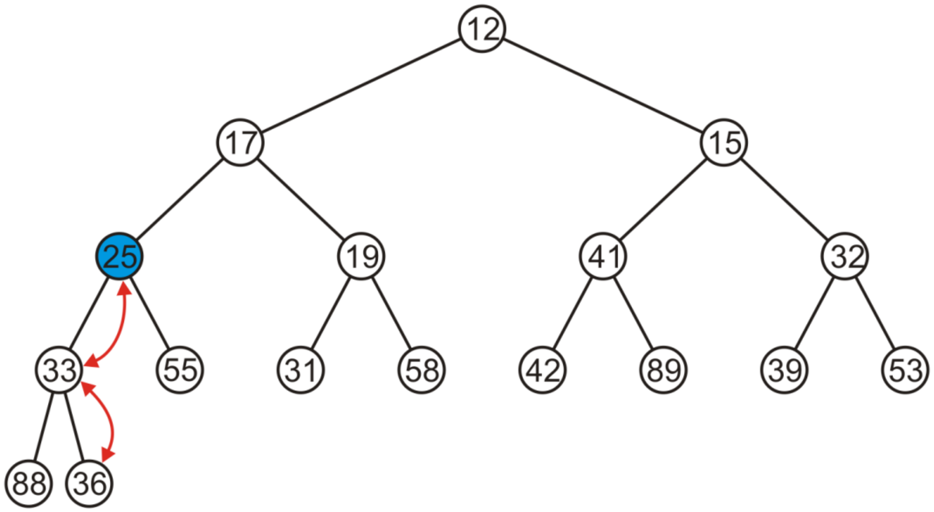
Priority queues and Heaps

- Since 25 < 36, we have to swap those. Now, 25 < 33, so we need to swap those at the second iteration — then we're done, since 25 > 17



Priority queues and Heaps

- And the resulting tree is a complete tree.



Priority queues and Heaps

- So, as long as we maintain a complete binary tree, we know its height $h = \Theta(\log n)$, and thus the two important operations (enqueue and dequeue) have a worst-case run time $\Theta(\log n)$

Priority queues and Heaps

- So, as long as we maintain a complete binary tree, we know its height $h = \Theta(\log n)$, and thus the two important operations (enqueue and dequeue) have a worst-case run time $\Theta(\log n)$
  - Well... except that it gets better!!

Priority queues and Heaps

- When we insert an element at the bottom (as a leaf node), do we need to do *h* swaps?

Priority queues and Heaps

- When we insert an element at the bottom (as a leaf node), do we need to do *h* swaps?
  - We certainly need to in the worst-case.... But what about the average case?

Priority queues and Heaps

- When we insert an element at the bottom (as a leaf node), do we need to do *h* swaps?
  - We certainly need to in the worst-case.... But what about the average case?
  - Would it stop half way on average?

Priority queues and Heaps

- When we insert an element at the bottom (as a leaf node), do we need to do *h* swaps?
  - We certainly need to in the worst-case.... But what about the average case?
  - Would it stop half way on average?
    - That wouldn't be such great news — it would still be $\Theta(\log n)$ ... That is, it would be good, but not *that* good.

Priority queues and Heaps

- Given the exponential nature of the number of leaf nodes (there are as many leaf nodes as internal nodes in a perfect tree — so, at depth *d*+1, there are twice as many nodes at as depth *d*)

Priority queues and Heaps

- Given the exponential nature of the number of leaf nodes (there are as many leaf nodes as internal nodes in a perfect tree — so, at depth d+1, there are twice as many nodes at as depth d)
- So, when a node is swapped to one level up, how does it compare against the rest of the elements at higher depth?

Priority queues and Heaps

- Given the exponential nature of the number of leaf nodes (there are as many leaf nodes as internal nodes in a perfect tree — so, at depth d+1, there are twice as many nodes at as depth d)
- So, when a node is swapped to one level up, how does it compare against the rest of the elements at higher depth?
  - By definition, there is absolutely no relationship between the data in different branches .... But ....

Priority queues and Heaps

- Because of the constraint that sub-trees below one given node have all values greater than the node, we have that, *on average*, assuming random data, evenly distributed, then the behaviour is that nodes at depth d are less than nodes at depth d+1.

Priority queues and Heaps

- Because of the constraint that sub-trees below one given node have all values greater than the node, we have that, *on average*, assuming random data, evenly distributed, then the behaviour is that nodes at depth *d* are less than nodes at depth *d*+1.
  - Thus, each time a node goes up one level, it is, *on average*, past half of the remaining elements!
  - So, after the first swap, on half of the cases it won't require any additional swaps; thus, average number of swaps is 1 !!

Priority queues and Heaps

- The actual math goes more or less as follows:
  - After k swaps ($0 \le k \le h$), we are at lower depth than $2^{(h-k)}$ elements, and we're interested in the probability of being above the parent node; this probability is given by the fraction $2^{(h-k)} / n$.

Priority queues and Heaps

- The actual math goes more or less as follows:
  - After k swaps ($0 \leq k \leq h$), we are at lower depth than $2^{(h-k)}$ elements, and we're interested in the probability of being above the parent node; this probability is given by the fraction $2^{(h-k)} / n$.
  - We get the average case by computing the weighted average of the number of swaps (the weights being those probabilities):

Priority queues and Heaps

- The actual math goes more or less as follows:
  - After k swaps ($0 \le k \le h$), we are at lower depth than $2^{(h-k)}$ elements, and we're interested in the probability of being above the parent node; this probability is given by the fraction $2^{(h-k)} / n$.
  - We get the average case by computing the weighted average of the number of swaps (the weights being those probabilities):

$$Avg.\ swaps = \sum_{k=0}^{h} k \cdot \frac{2^{(h-k)}}{n} = \frac{2^{h+1} - h - 2}{n} = \Theta(1)$$

Priority queues and Heaps

- So, this is great news — we have the following run times:
  - Insertion: $\Theta(\log n)$ worst-case
    $\Theta(1)$ average-case
  - Removal: $\Theta(\log n)$ worst-case and average-case

- This is definitely much better than with balanced binary search trees.

Priority queues and Heaps

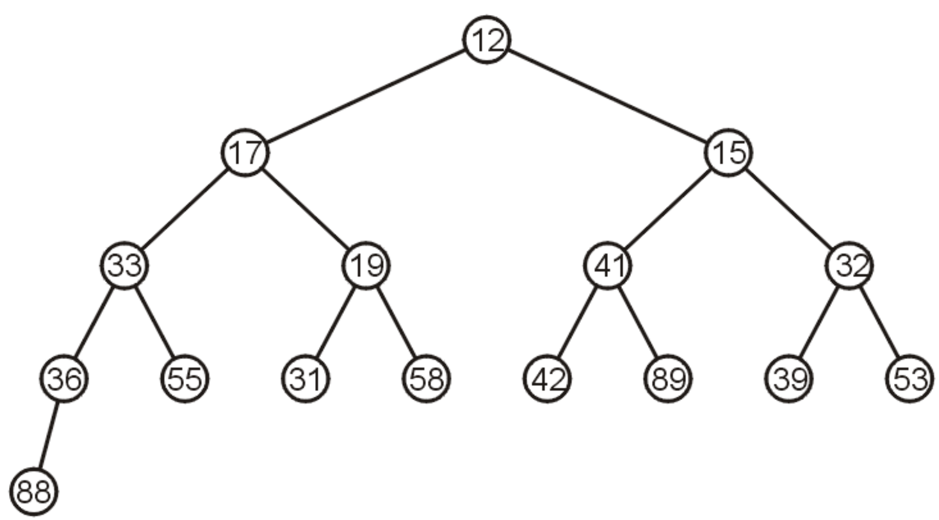
- Food for thought: why did this happen? Why, if we have a binary search tree structure in both cases, and we're in a sense putting data in order, why did we get something so radically faster with binary heaps?

Priority queues and Heaps

- I'm actually leaving that one for you guys to think about it ....

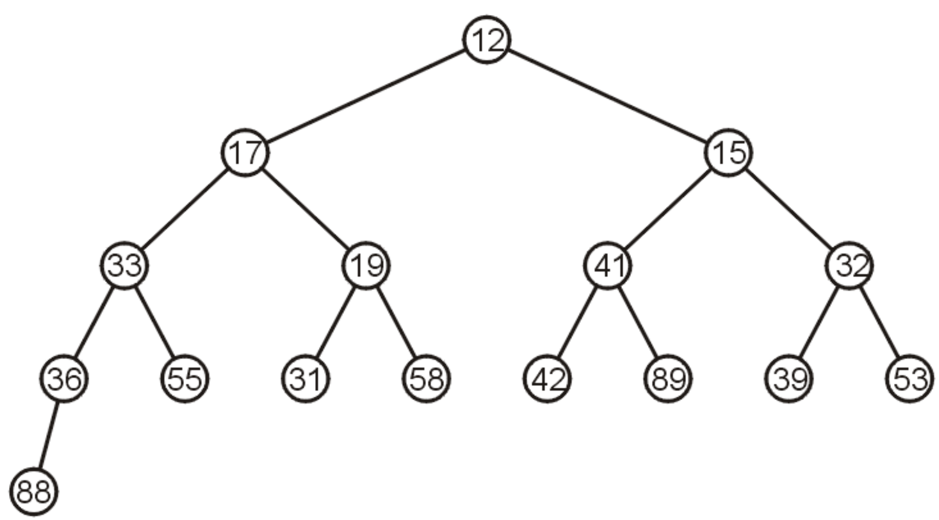
Priority queues and Heaps

- There's one additional detail, though:
  - When removing (the root element), how do we guarantee that the resulting tree will be a complete binary tree? (in the tree below, we won't end up promoting 88 — which is the only way in which we would end up with a complete binary tree)



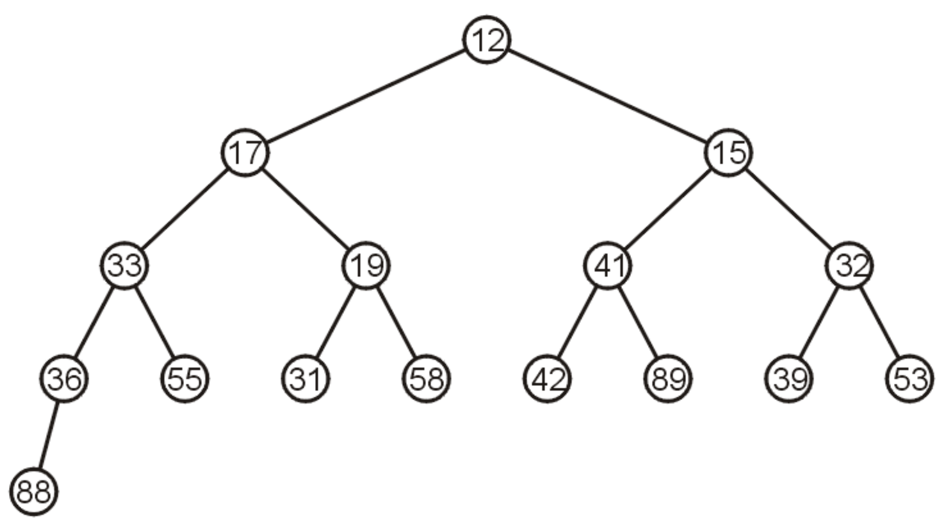
Priority queues and Heaps

- There's one additional detail, though:
  - We'd end up promoting 39 and leaving a hole there (in this case, we could move 88 to that position; but in general, we have no guarantee that we'll be able to ... right? why?).



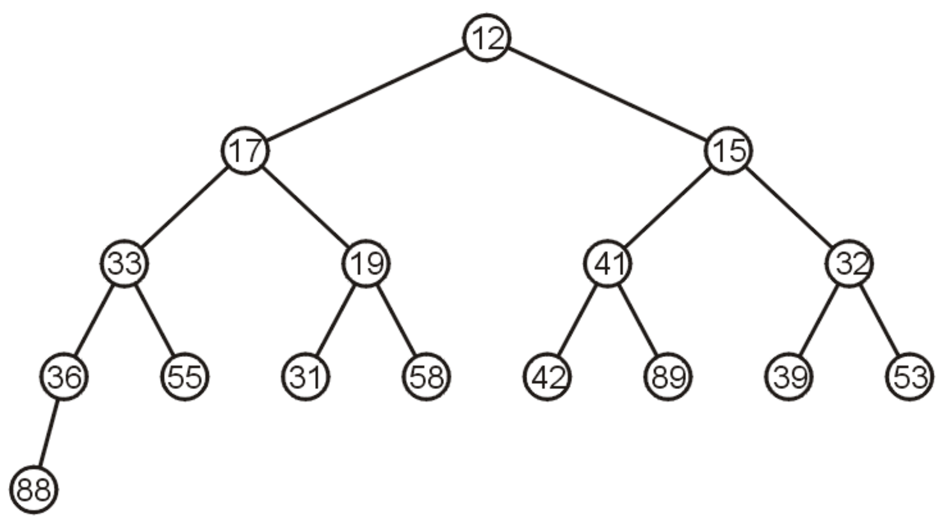
Priority queues and Heaps

- There's one additional detail, though:
  - We'd end up promoting 39 and leaving a hole there (in this case, we could move 88 to that position; but in general, we have no guarantee that we'll be able to ... right? why?). So, any ideas ?



Priority queues and Heaps

- There are actually two possibilities:
  - Either move the 88 to the hole left by the removal and then adjust (as if we were doing an insertion; percolate it to its corresponding position).
  - Or move the last entry to the root, and then percolate it down.



Priority queues and Heaps

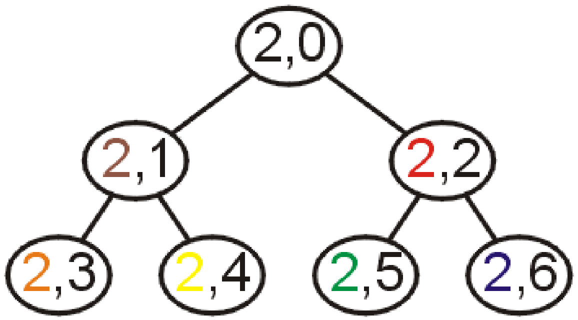
- Coming back to the issue of ordering by comparing pairs (p, k) — priority + order of arrival...
- Let's look at how this works with heaps:

Priority queues and Heaps

- Coming back to the issue of ordering by comparing pairs (p, k) — priority + order of arrival...
- Let's look at how this works with heaps:
  - Let's say that we insert 7 elements, all with priority 2. The counter k would then go from 0 to 6, and the heap could end up as follows:

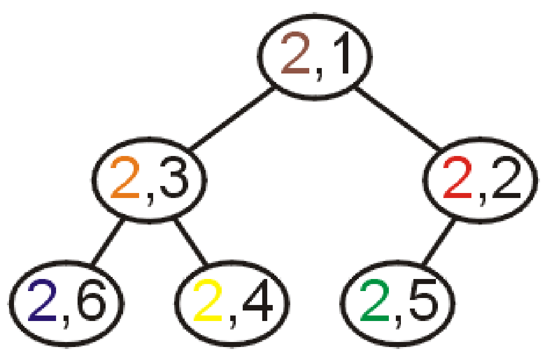
Priority queues and Heaps

- Now removing them:
  - We extract the element at the root:



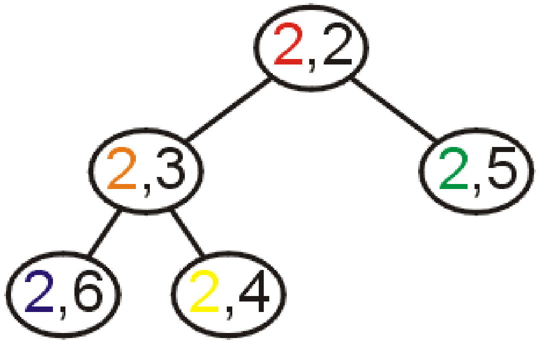
Priority queues and Heaps

- Now removing them:
  - We extract the element at the root: (2,6) goes up, then swapped with (2,1), then (2,3)



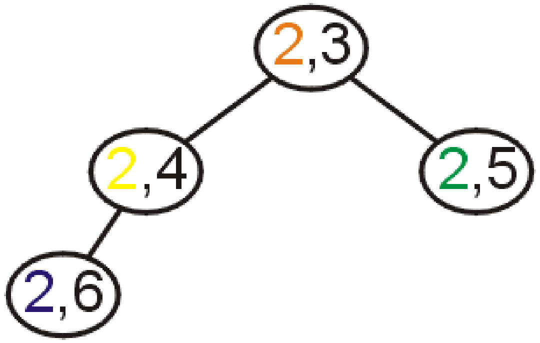
Priority queues and Heaps

- Now removing them:
  - Then we'd go in the following order:



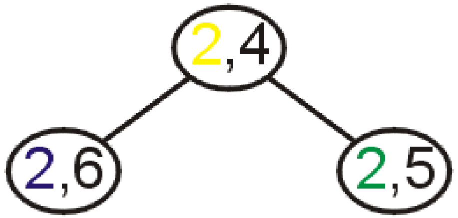
Priority queues and Heaps

- Now removing them:
  - Then we'd go in the following order:



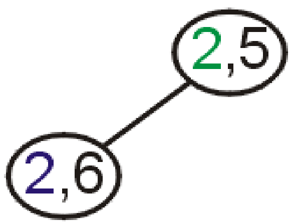
Priority queues and Heaps

- Now removing them:
  - Then we'd go in the following order:



Priority queues and Heaps

- Now removing them:
  - Then we'd go in the following order:



Priority queues and Heaps

- One last detail — perhaps the "cool upon cool" of all features of this binary heap:
  - Because we always maintain a complete binary tree, then we can implement it as an array !!
  - Leaving the first cell (subscript 0) unused:
    - Children of node k (i.e., value at subscript k of the array) are $2k$ and $2k+1$.
    - Parent of node k is $k \div 2$ (as in, integer division)

Priority queues and Heaps

- One last detail — perhaps the "cool upon cool" of all features of this binary heap:
  - Because we always maintain a complete binary tree, then we can implement it as an array !!
  - Leaving the first cell (subscript 0) unused:
    - Children of node k (i.e., value at subscript k of the array) are $2k$ and $2k+1$.
    - Parent of node k is $k \div 2$ (as in, integer division)
  - So here's a radical idea: maybe we will be able to use heaps to sort data! Since it is all in an array, sounds like we're in business...

Priority queues and Heaps

- Sorting using heaps ...
  - One important obstacle we'd have to clear:
    - Can it be done in-place? Sounds like we'd need to take the data from the array (arbitrary and unconstrained data) and insert the elements into a heap.

Priority queues and Heaps

- Sorting using heaps ...
  - One important obstacle we'd have to clear:
    - Can it be done in-place? Sounds like we'd need to take the data from the array (arbitrary and unconstrained data) and insert the elements into a heap.
- That will actually be our next topic — the truly remarkable *Heap Sort* algorithm!

Summary

- During today's lesson:
  - Introduced the notion of priority queues
  - Discussed some “obvious”, but not too efficient, implementations (array of queues; balanced binary search tree)
  - Looked into the Heap data structure, for a more efficient implementation alternative for priority queues.
    - Discussed operations on a heap and their run time.