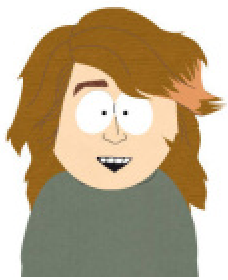
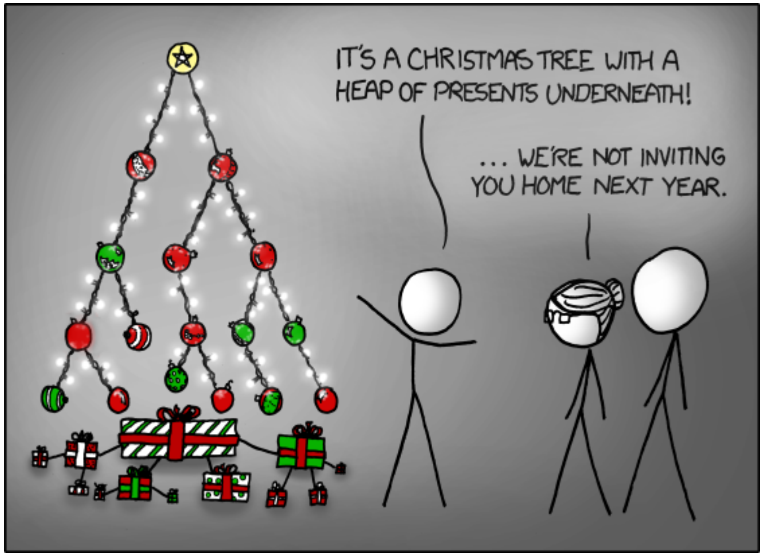
Priority queues and Heaps



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.
      - 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.
  - 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

- Disadvantage: Not efficient
  - Can you see why?

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

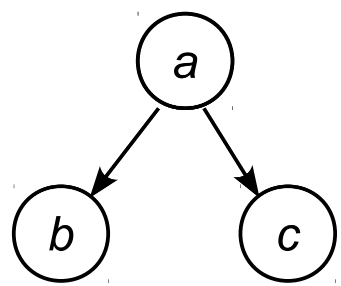
- With this, we could simply use a balanced binary search tree (e.g., an AVL tree) using that pair as the value being inserted.
- An AVL tree maintains the elements in order with insertions and removals taking logarithmic time.
- However, the implementation is more complicated than it could be, as we'll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

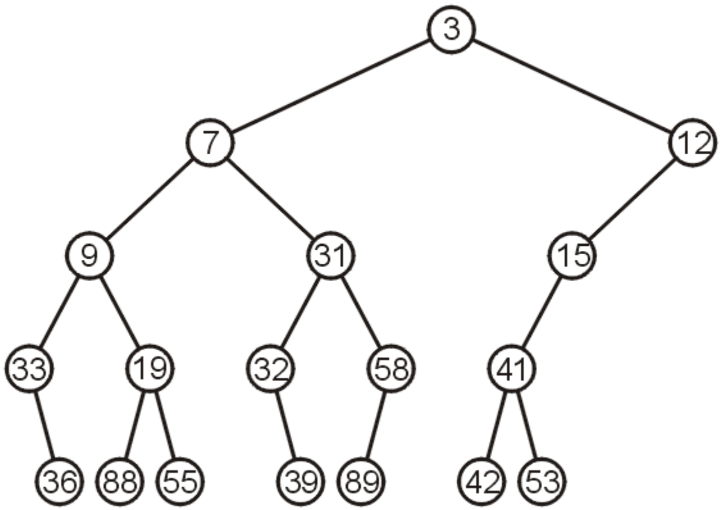
- Important "fine print" in that definition:
  - Sibling elements — or elements in the two sub-trees have NO RELATIONSHIP WHATSOEVER!!



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

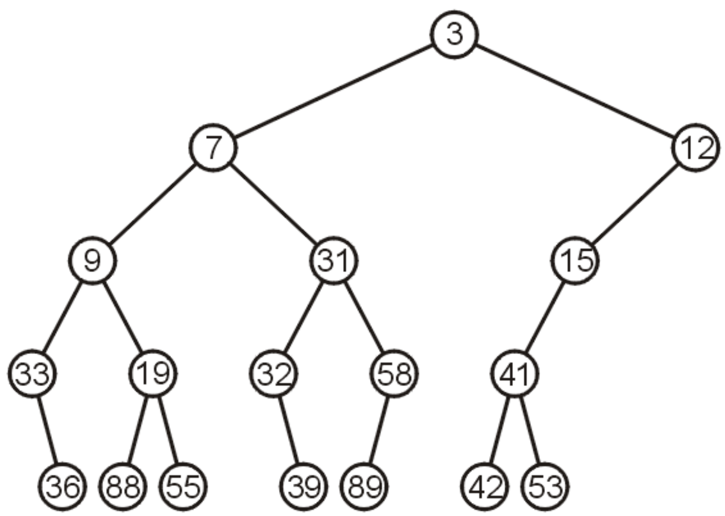
Priority queues and Heaps

- 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

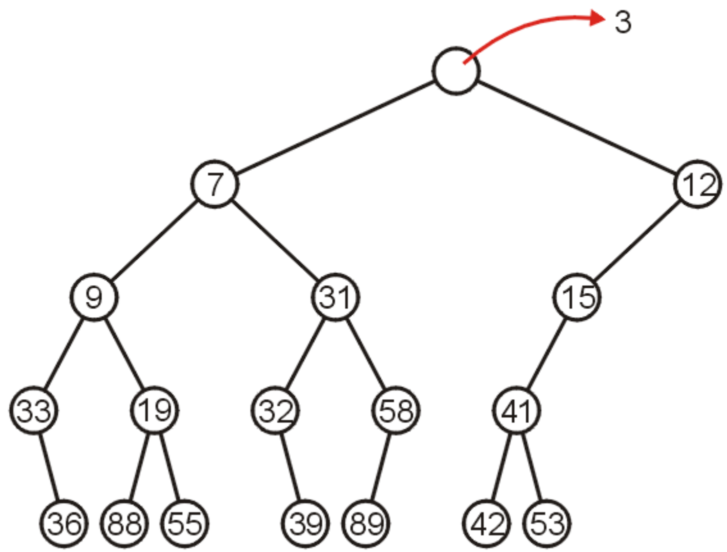
- We can obviously find the lowest value in constant time: right? how?

Priority queues and Heaps

- 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.

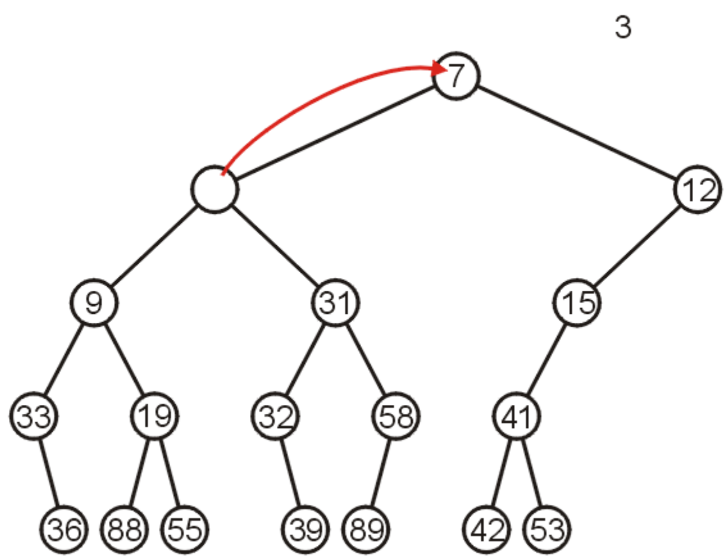
Priority queues and Heaps

- An example:



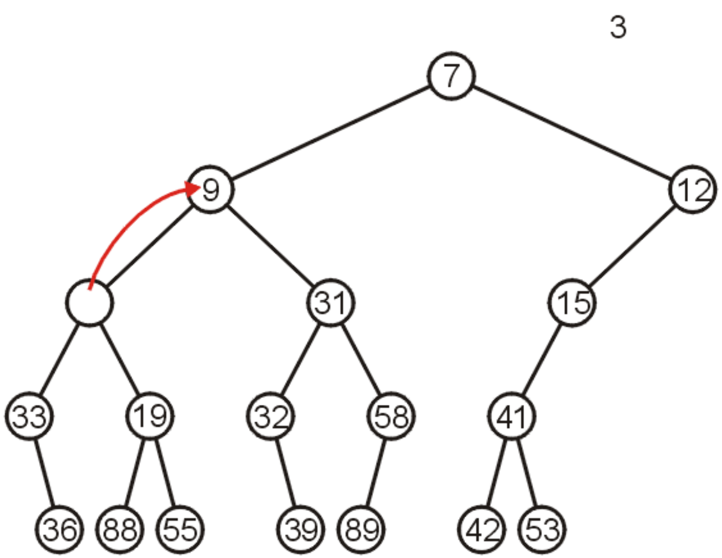
Priority queues and Heaps

- An example:



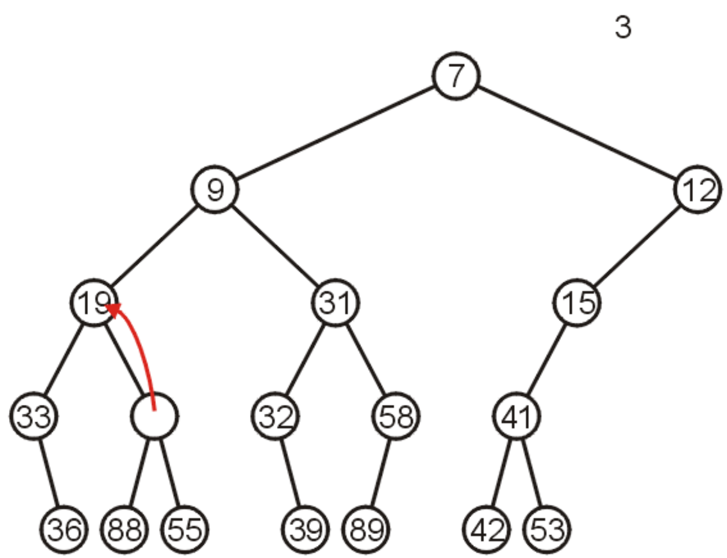
Priority queues and Heaps

- An example:



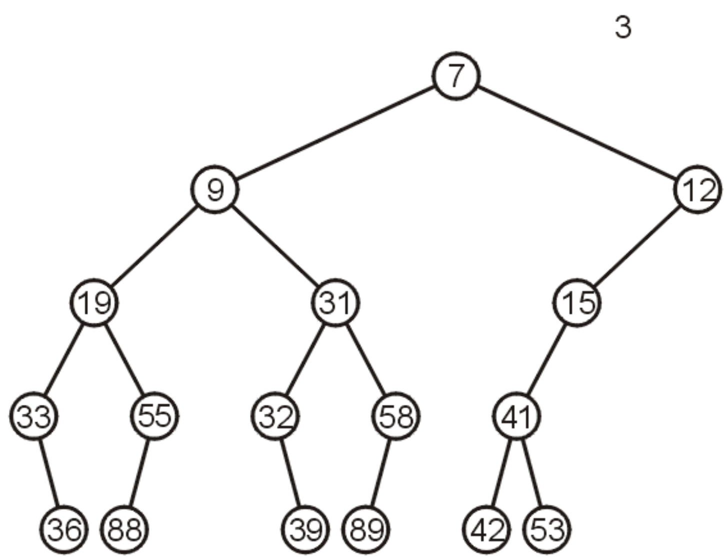
Priority queues and Heaps

- An example:



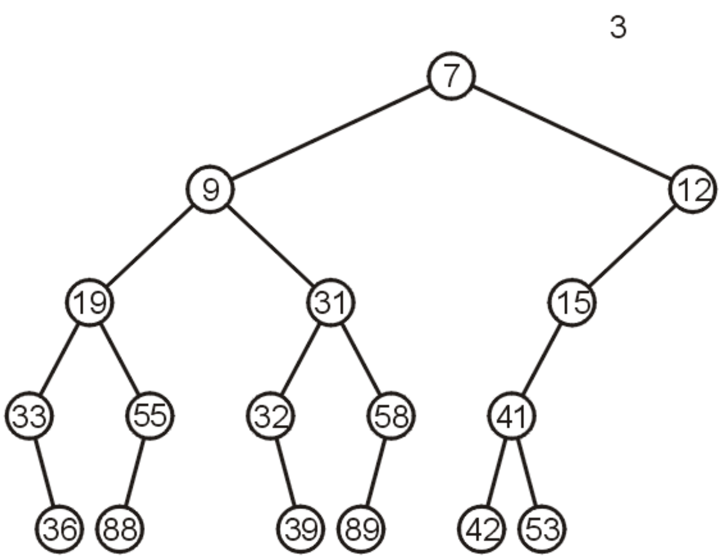
Priority queues and Heaps

- An example:



Priority queues and Heaps

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



Priority queues and Heaps

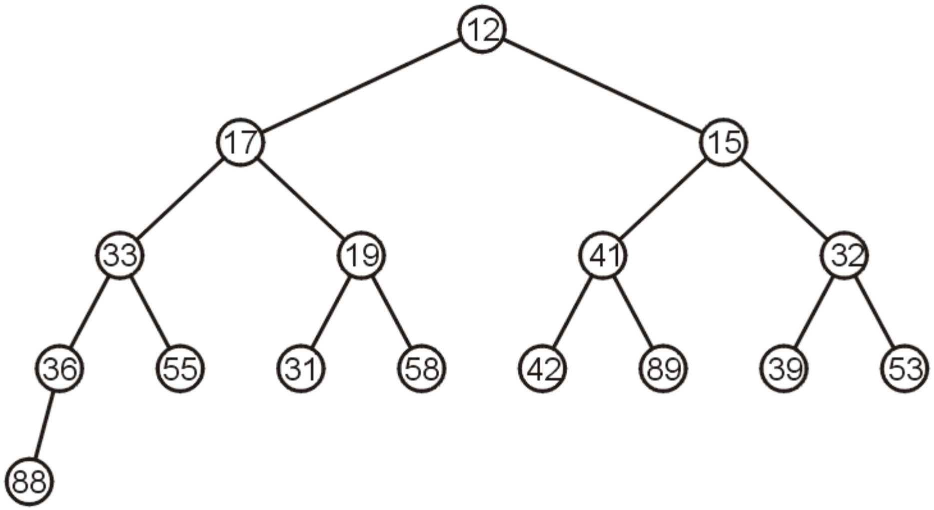
- Inserting an element has to be done with care: we want to maintain balance!
  - One rather neat way to do this is ensuring that we always have a *complete* binary tree!
    - (and BTW, when we say a complete binary tree, this carries a piece of good news with it ... right?)
  - So, we insert a leaf 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 the main advantage 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.

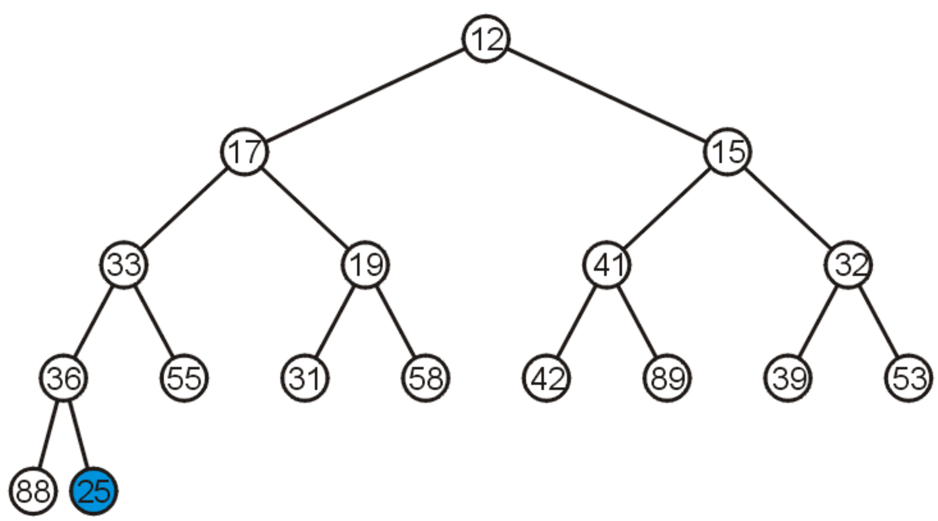
Priority queues and Heaps

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



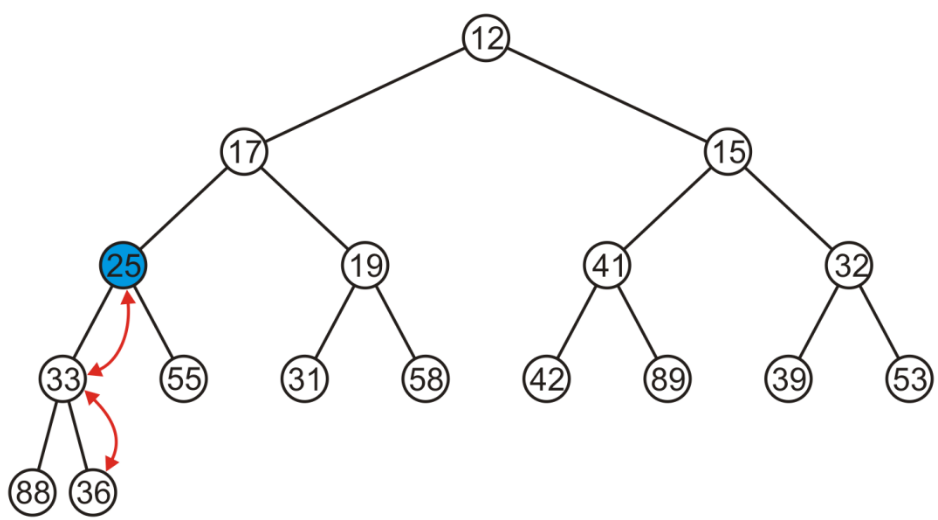
Priority queues and Heaps

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



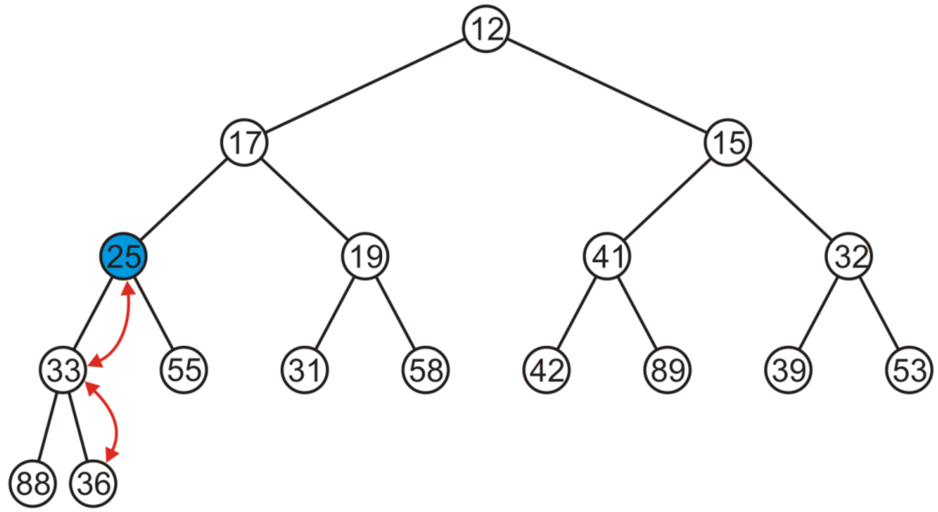
Priority queues and Heaps

- 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

- Notice that we don't need to worry about 33 when we swap it down — it was already lower than anything in that sub-tree.



Priority queues and Heaps

- We'll look at array implementation in class.
- We'll also discuss the run times more in detail in class.

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.