Heaps (of Fun)

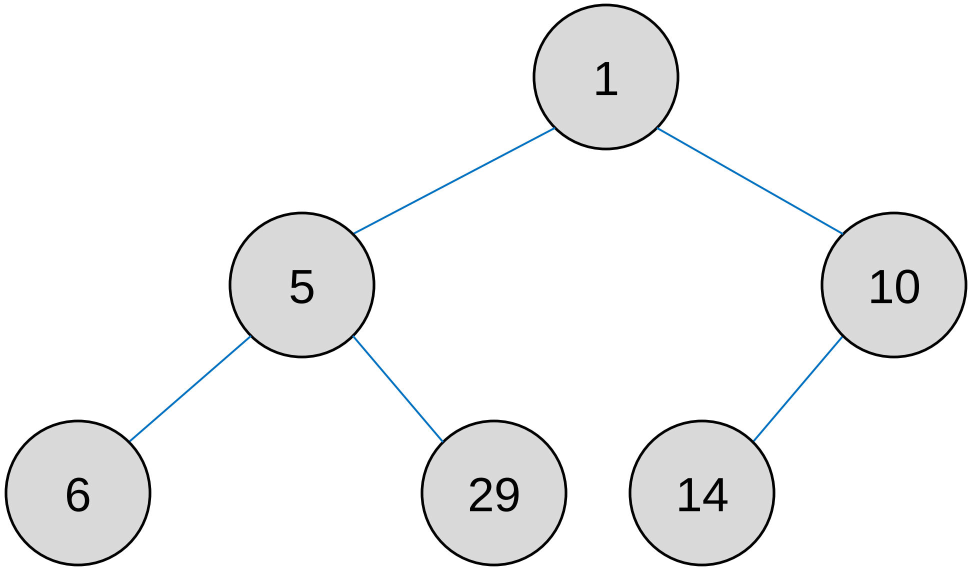
Heaps

Heap of sloths! (with a few teddy bears thrown in, I think…)

Heaps

- A **heap** is a special case of a complete tree
  - In this course, we'll assume "heap" refers to a **binary heap** (i.e., each node has 0, 1, or 2 children)
  - Remember that **complete** means that all "rows" of the tree are filled except possibly the last one; the last row's nodes are all placed toward the left
- Two types of heaps:
  - In a **min-heap**, each node must be <u>less than</u> both of its children
  - In a **max-heap**, each node must be <u>greater than</u> both of its children
- We will discuss heap algorithms using a min-heap as the example. The same algorithms can be easily applied to max-heaps – just use the opposite comparisons!

Example of a min-heap



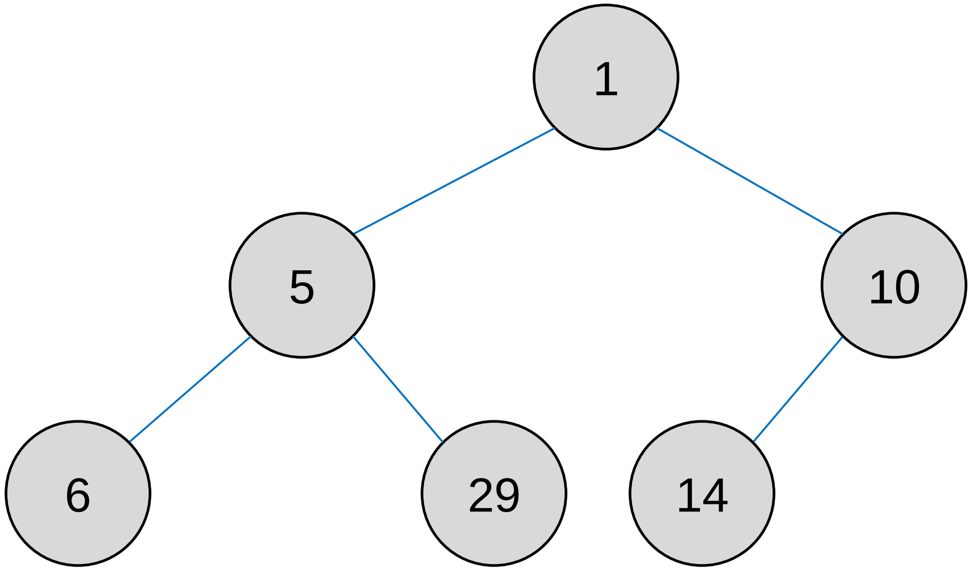
Note that in general, the level of a node does not imply anything about its value relative to nodes in other levels. For example, the 6 in level 3 is less than the 10 in level 2, while the 29 (also in level 3) is greater than the 10. However, we do know for sure that the root is the smallest element in the heap.

Inserting elements into a heap

- To insert a new element into a heap:
  - Place the element into the next available spot on the heap's last row (start a new row if the last row is already filled)
  - Compare the new element vs. its parent. If it's <u>less than</u> the parent, swap the element with its parent. Repeat as many times as necessary.
    - Two stopping conditions: 1) the new element is no longer less than its parent, or 2) the new element is brought all the way up to the root and hence no longer has a parent

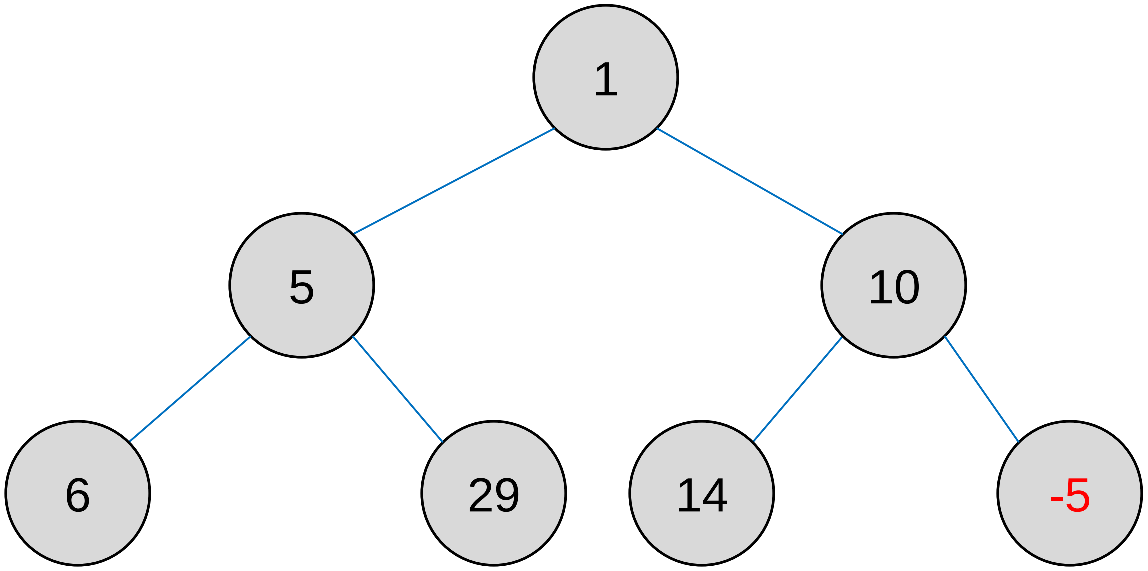
Inserting elements into a heap

Example: Insert -5 into the min-heap below.



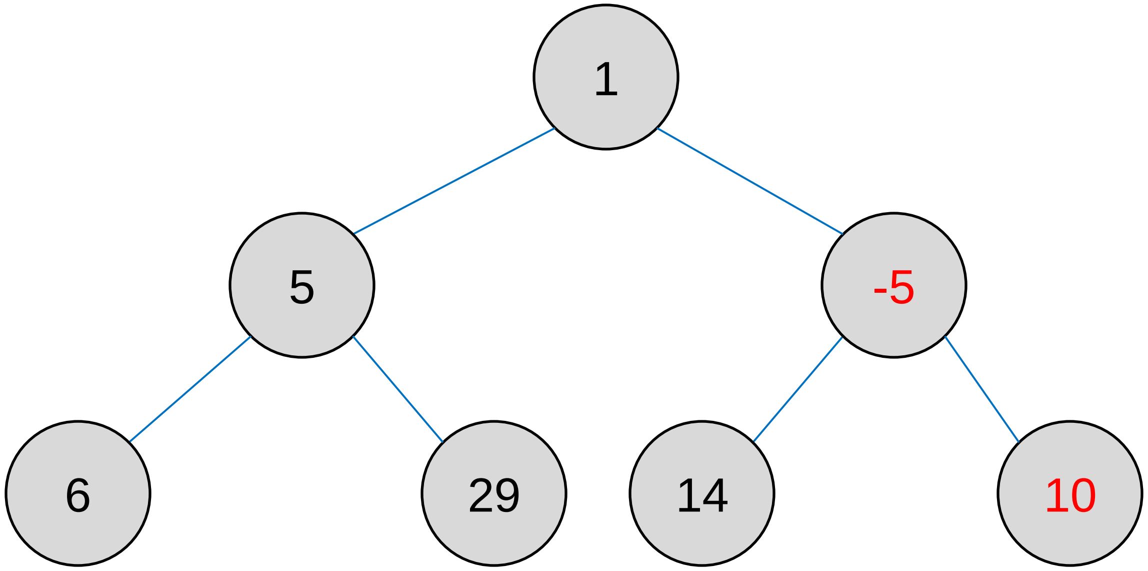
Inserting elements into a heap

Start by placing -5 into the next available spot on the last row. Compare -5 vs. its parent (10).



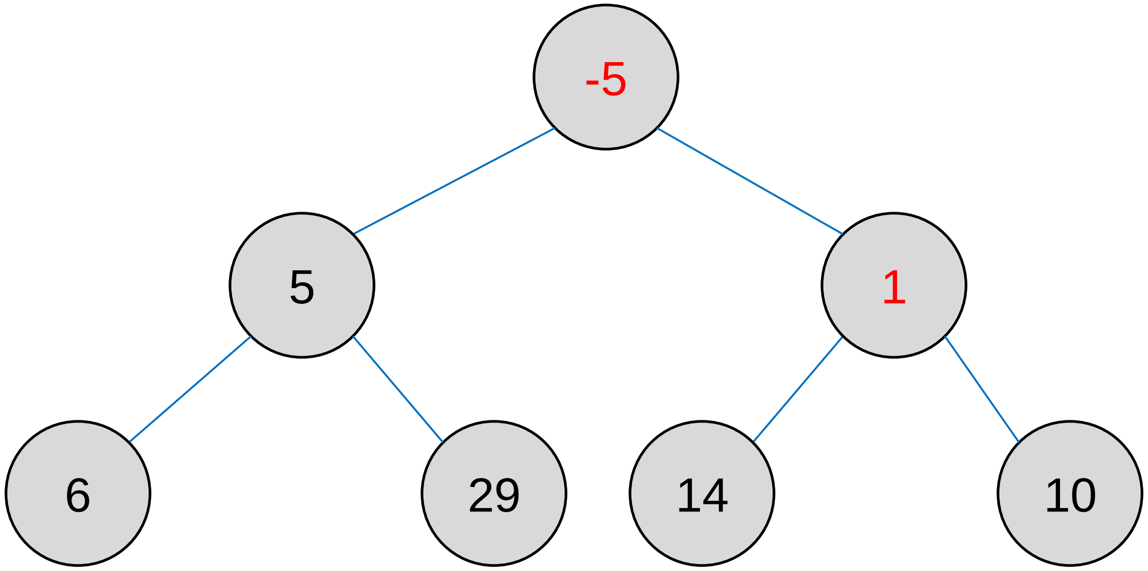
Inserting elements into a heap

Since -5 is less than 10, swap the two elements. Now compare -5 vs. its new parent (1).



Inserting elements into a heap

Since -5 is less than 1, swap the two elements. -5 is now the root, so it no longer has a parent. We are done!

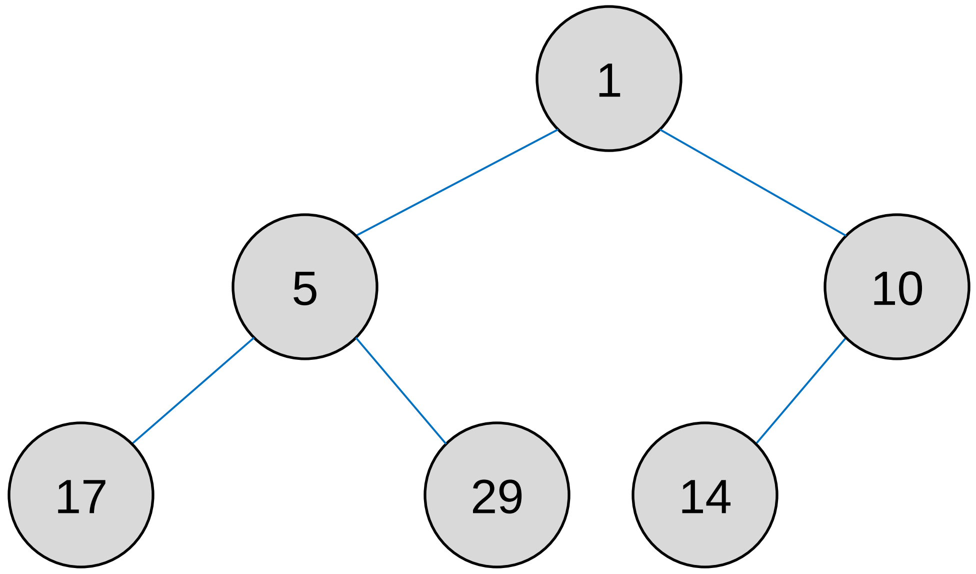


Deleting elements from a heap

- Elements are always deleted from the top (root) of a heap
  - Similar to how elements in a stack are removed only from the top
- To delete the top element from a heap:
  - Replace the top element with the “last” element in the heap (the rightmost element in the last row). Remove this last element from the heap.
  - Compare the new root vs. the <u>lesser</u> of its two children, and swap the two elements if the new root is <u>greater than</u> the lesser child. Repeat as many times as necessary.
    - Two stopping conditions: 1) the new root is no longer greater than its lesser child, or 2) the new root no longer has any children

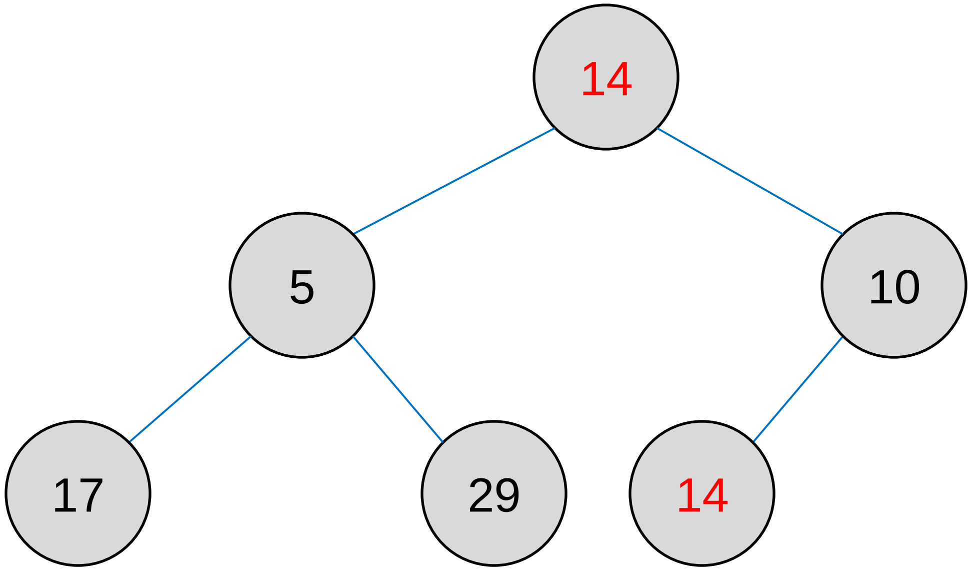
Deleting elements from a heap

Example: Delete the top element from the min-heap below.



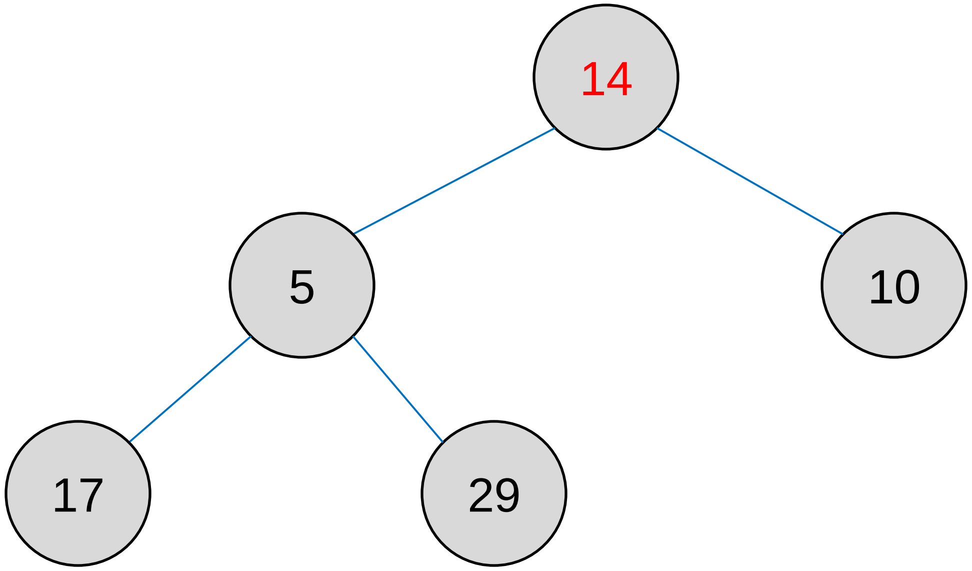
Deleting elements from a heap

Start by replacing 1 with last element in the heap (14).



Deleting elements from a heap

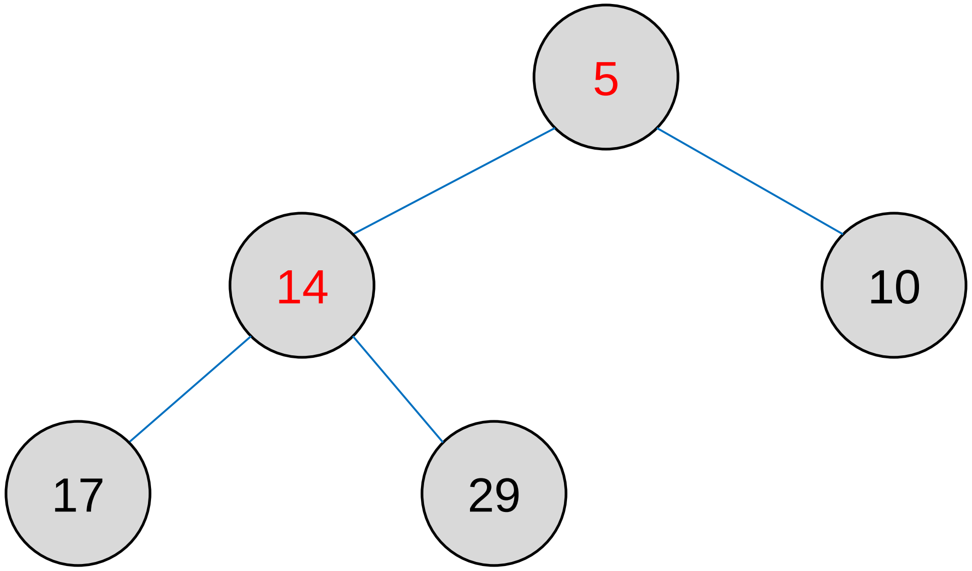
Remove the original 14 from the heap. Now compare 14 vs. the lesser of its children (5).



Note that the last element in the heap will never have any children, so deleting it is always easy!

Deleting elements from a heap

14 is greater than 5, so swap the two elements. Now compare 14 vs. the lesser of its children (17). 14 is not greater than 17, so we are done!



Analysis of heap operations

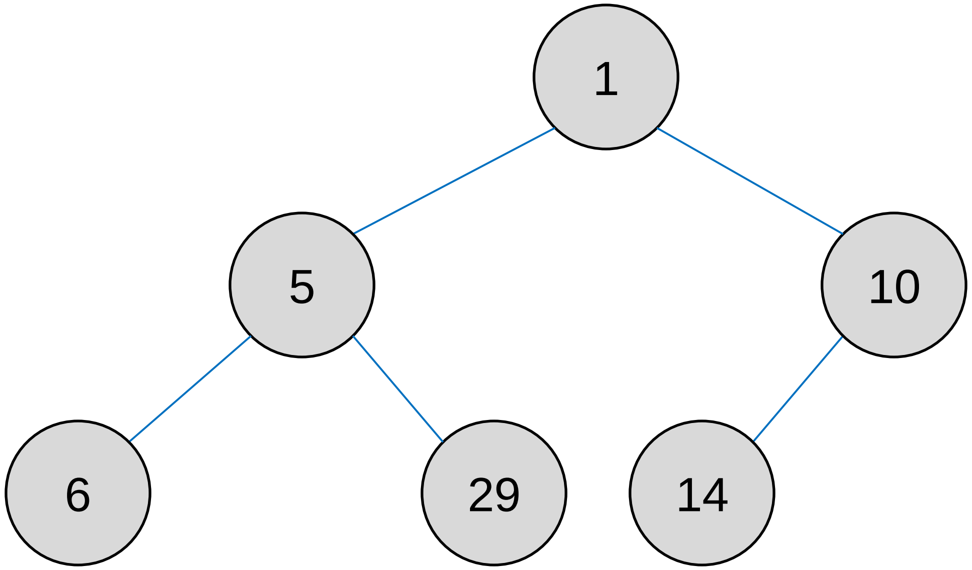
- Insertion involves adding something to the end of the heap and then working back up the heap (potentially all the way up to the root)
- Deletion involves replacing the root with the last element and then working back down the heap (potentially all the way down to the last level of the heap)
- Thus, the amount of work done by insertion and deletion depends on the <u>height</u> of the heap. Since a heap is a complete binary tree, this height is guaranteed to be log *n* (where *n* = the number of elements in the heap).
- Both heap insertion and deletion are *O*(log *n*) operations

Representing heaps with arrays

- Since a heap is just a binary tree, we could code something involving nodes linked to other nodes, similar to what we did for binary search trees
- However, because heaps are complete trees, there is an efficient array-based implementation!

Representing heaps with arrays

Graphical / tree representation:



Array representation (root is at index 0):

1	5	10	6	29	14				

Representing heaps with arrays

- Seems pretty simple, right?!
- Note that for any index *n* in the array:
  - Node *n*'s left child is located at index 2*n* + 1
  - Node *n*'s right child is located at index 2*n* + 2
  - Node *n*'s parent is located at index (*n* – 1) / 2 (use integer division)
- To implement a heap, we can just store all its elements into an array and use the above relationships to navigate between parent and child nodes

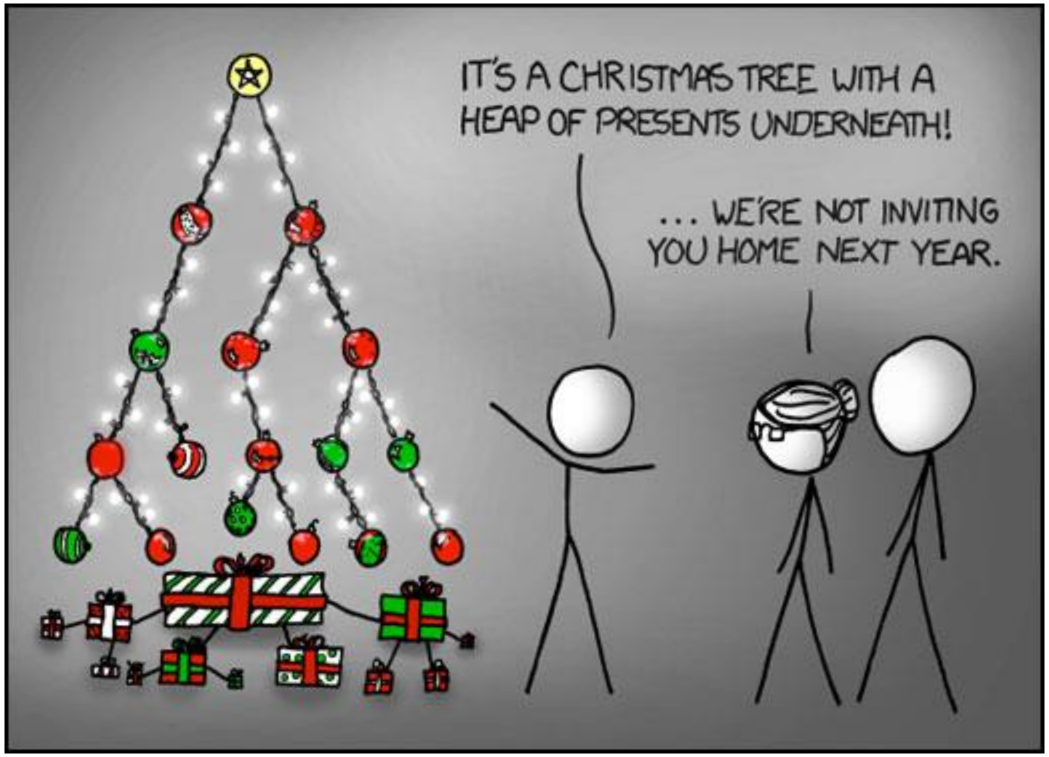
Priority queues

- Remember that in a standard queue, enqueue is always done from the back of the queue, while dequeue is always done from the front of the queue. The elements will be dequeued in the same order that they were enqueued (FIFO).
- In a **priority queue**, each element in the queue is assigned a priority. Dequeue always removes the highest-priority element from the queue, regardless of when that element was enqueued.
  - Useful in many computer science applications (e.g., print queues, process scheduling in an operating system)

Priority queues and heaps

- As you might've guessed, we can easily implement a priority queue using a heap!
  - The highest-priority element is always kept on top of the heap
    - With a min-heap, we assume the "smallest" element has the highest priority
  - Enqueueing a new element corresponds to inserting it into the heap
  - Dequeueing an element corresponds to removing the top element from the heap (which is always the highest-priority element)

Relevant xkcd


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.