Priority Queue

Priority Queue:

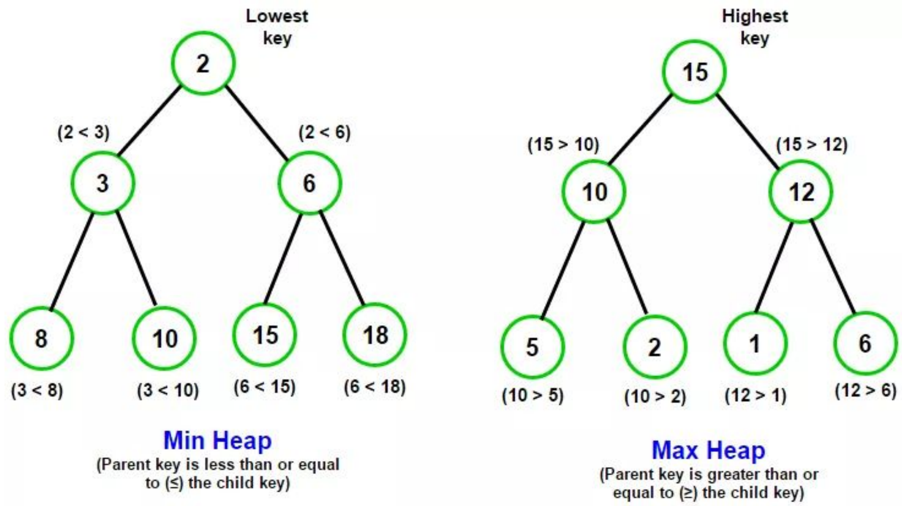
- A priority queue is an ADT with the following major methods:
    - void add (T x);
    - T removeMin();
- Similar to a regular Queue but now each element has a priority or a ranking such that the one with the highest priority will be removed first.
- It is important that the object being add have a compareTo method that specifies the comparisons between two instances of that class.

Implementation:

- Unsorted Array:
    - We can insert the elements into the queue one by one in no particular order.
        - void add (T x) takes O(1) time
    - But we must do extra work and make sure we remove each the element with the highest priority.
        - T removeMin() takes O(n) time
- Sorted Array:
    - We can insert the elements in order of their priorities.
        - void add (T x) takes O(n) time
    - We do the extra work in add so we can save time in removing. Since the Queue is already sorted, we can just remove normally.
        - T removeMin() takes O(1) time
- Binary Heap:
    - We can do some work in adding and some work in removing.
    - The runtime of adding and removing take O(lgn)

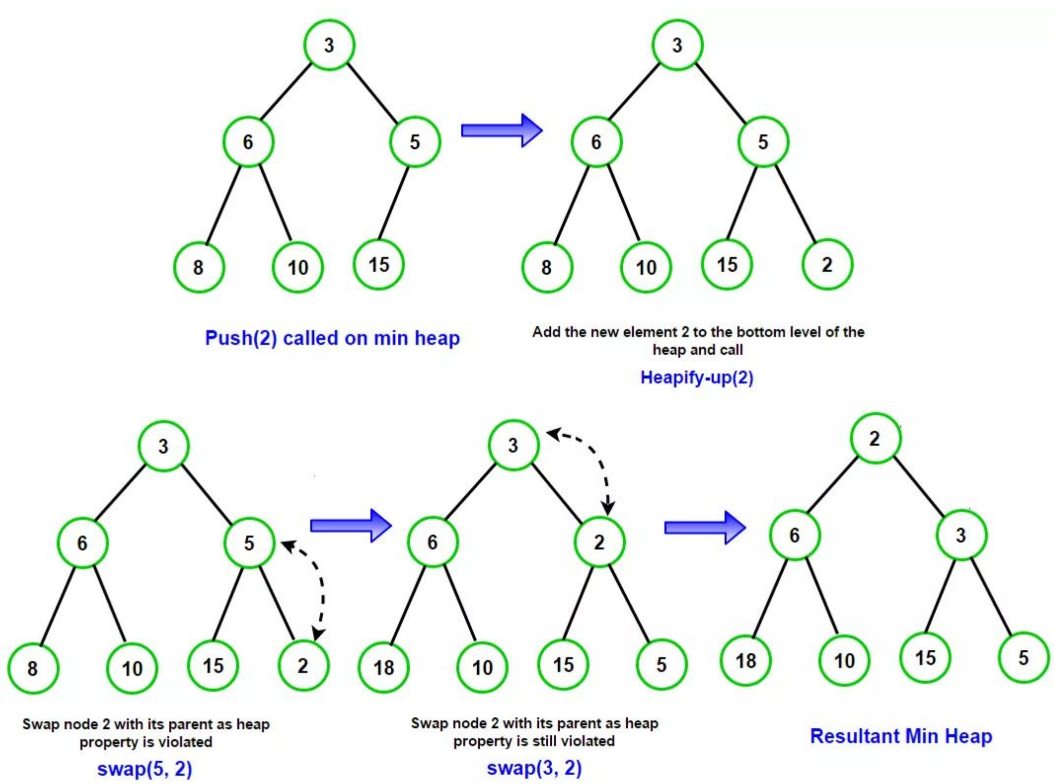
Binary Heap:

- A Heap is a binary tree that has the following 2 properties:
    - **Heap Order:** At any node the data is smaller than any data of its childrens subtree.
    - **Heap Shape:** All levels of the tree are completely full except the last level which can be partially filled from left to right.
        - A complete tree can be efficiently represented an array. This saves space because we do not have to store pointers.
    - Implementation:
        - Using an array. Where the root is located at index 0
        - Where are the children of node N?
            - Located at index 2n+1 and 2n+2
        - Parents of node N?
            - (n-1)/2

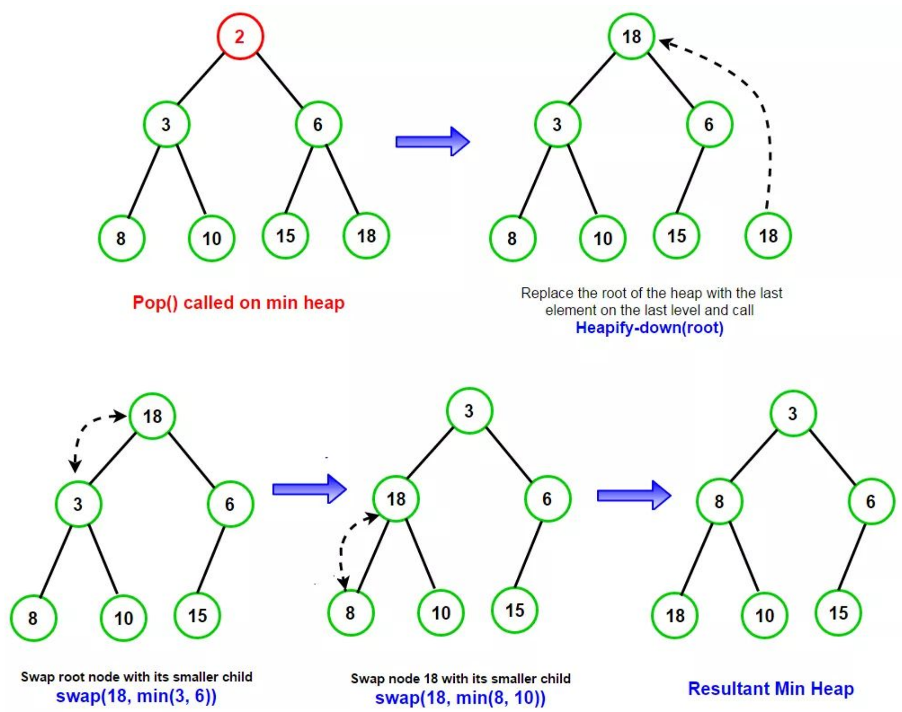


Source: https://www.techiedelight.com/introduction-priority-queues-using-binary-heaps/

- **Insertion (Min Heap):**
    - Place the new element one spot after the last element (the new element becomes the rightmost element of the bottom level).
    - While this element is smaller than its parent:
        - Move this element upward by swapping it with its parent.
        - Known as Bubble Up
        - This process takes O(lgn)



- **Remove (Min Heap)**
    - Move the last element (the rightmost element of the bottom level) to the root of the heap (replacing the previous root).
    - While this element is larger than one or both children:
        - Move this element downward by swapping it with its smaller child.
        - Known as Bubble Down
        - This process takes O(lgn)



Runtime:

- The height of the heap is O(logn), so in the worst case, both insertion and deletion perform O(logn) comparisons and swaps. Therefore the worst-case runtime is O(logn) (assuming the heap doesn't run out of space).

Bottom Up Heap/ Heapify:

- We can see that if we insert n elements into a priority queue, that would take O(nlgn).
- We can make this fast if we know all n elements that we want to insert due to the nature of a heap structure.
- Bottom up heap construction takes O(n)
- Before we begin, note that every leaf is already a 1-element heap.
    - For each position i, starting at the last non-leaf and ending at the root, bubble-down heap[i].
    - Both subtrees of heap[i] are already heaps, so after bubbling-down, the subtree rooted at heap[i] is a heap.

- Intuitively, this is faster because most of the elements of a heap are close to the bottom, so when we bubble-down each element, most elements do not have to be moved far.
- On the other hand, if we build the heap by inserting one element at a time, and bubble-up each element, in the worst case every element must be moved all the way to the root.

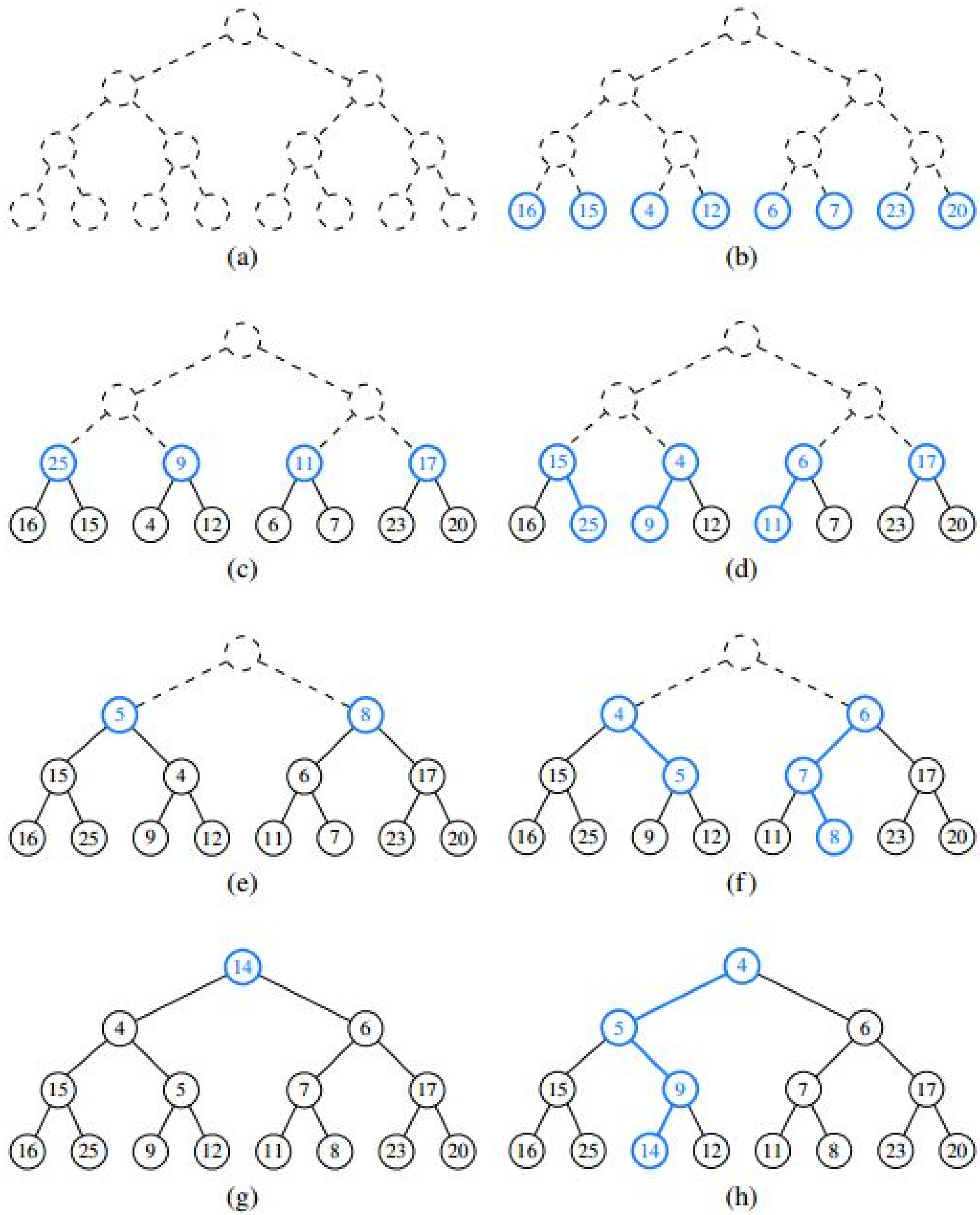


Figure 9.5: Bottom-up construction of a heap with 15 entries: (a and b) we begin by constructing 1-entry heaps on the bottom level; (c and d) we combine these heaps into 3-entry heaps; (e and f) we build 7-entry heaps; (g and h) we create the final heap. The paths of the down-heap bubblings are highlighted in (d, f, and h). For simplicity, we only show the key within each node instead of the entire entry.