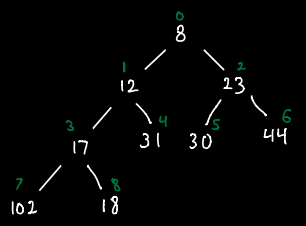
Theory

- A binary heap is a special type of binary tree that satisfies two additional properties:
  1. It is complete: The tree is filled up from Top to Bottom and Left to Right
  2. The heap property (distinguishes the difference b/w min and max heap):
     - Min heap: Every node in the heap has it's value $\leq$ it's children's node value
     - Max heap: Every node in the heap has it's value $\geq$ it's children's node value
- Note: A heap is not sorted
- A min (or max) heap can be represented very nicely using an array:

0	1	2	3	4	5	6	7	8
8	12	23	17	31	30	44	102	18



- If the current node is at index i
  $\Rightarrow$ It's first child is at index $2i + 1$
  $\Rightarrow$ It's second child is at index $2i + 2$
- Ex: node with value 23 is at 2
  So it's children are at:
  $2(2) + 1 = 5$ (30) $\quad 2(2) + 2 = 6$ (44)

Code:

① **Sift Up**: siftUp(currentIdx, heap)
- Find parent using floor((i-1)/2)
- While currentIdx > 0 (not at the root) and current value < parent value (min heap so children should be > parent)
  - swap(currentIdx, parentIdx, heap)
  - currentIdx = parentIdx
  - parentIdx = floor((currentIdx - 1)/2)

② **Sift down**: siftDown(currentIdx, endIdx, heap)
- Find first child index using 2(currentIdx) + 1
- While childOneIdx $\leq$ endIdx (there is at least one child node)
  - Find second child index using 2(currentIdx) + 2 (IF it is > endIdx set childTwoIdx to -1)
  - Find which index to swap with (childOne or childTwo)
    - If childTwoIdx != -1 && childTwo's value < childOne's value
      set idxToSwap as childTwoIdx
    - else set idxToSwap as childOneIdx
  - If the the idxToSwap's value < currentIdx's value:
    - swap(idxToSwap, currentIdx, heap)
    - update current Idx to idxToSwap
    - update childOneIdx to its left child node using 2(childOneIdx) + 1
  - else return

③ **Insert**: insert(value)
- push onto the heap (into our array)
- Sift up accordingly

④ **Remove**: remove()
- Swap first and last value in our heap
- pop off last value (min value) and store it
- Sift down value at index 0
- return popped off value

⑤ **Build Heap**: buildHeap(array)
- Start at the last parent node, found using
  floor((array.length - 1 - 1)/2)
- While the currIdx (starts at the last parent node Idx) is >0, siftDown starting at the last parent's Idx
- return array at end

```
class MinHeap {
  constructor(array) {
    this.heap = this.buildHeap(array);
  }

  // O(n) time | O(1) space
  buildHeap(array) {
    let lastParentIndex = Math.floor((array.length - 2) / 2);
    let currentIdx = lastParentIndex;
    while (currentIdx >= 0) {
      this.siftDown(currentIdx, array.length - 1, array);
      currentIdx--;
    }
    return array;
  }

  // O(logn) time | O(1) space
  siftDown(currentIdx, endIdx, heap) {
    let childOneIdx = currentIdx * 2 + 1;
    while (childOneIdx <= endIdx) {
      let childTwoIdx = currentIdx * 2 + 2 <= endIdx ? currentIdx * 2 + 2 : -1;
      let idxToSwap;
      if (childTwoIdx !== -1 && heap[childTwoIdx] < heap[childOneIdx]) {
        idxToSwap = childTwoIdx;
      } else {
        idxToSwap = childOneIdx;
      }
      if (heap[idxToSwap] < heap[currentIdx]) {
        this.swap(currentIdx, idxToSwap, heap);
        currentIdx = idxToSwap;
        childOneIdx = currentIdx * 2 + 1;
      } else {
        return;
      }
    }
  }

  // O(logn) time | O(1) space
  siftUp(currentIdx, heap) {
    let parentIdx = Math.floor((currentIdx - 1) / 2);
    while (currentIdx > 0 && heap[currentIdx] < heap[parentIdx]) {
      this.swap(currentIdx, parentIdx, heap);
      currentIdx = parentIdx;
      parentIdx = Math.floor((currentIdx - 1) / 2);
    }
  }

  // O(1) time | O(1) space
  peek() {
    return this.heap[0];
  }

  // O(logn) time | O(1) space
  remove() {
    this.swap(0, this.heap.length - 1, this.heap);
    const valueToRemove = this.heap.pop();
    this.siftDown(0, this.heap.length - 1, this.heap);
    return valueToRemove;
  }

  // O(logn) time | O(1) space
  insert(value) {
    this.heap.push(value);
    this.siftUp(this.heap.length - 1, this.heap);
  }

  swap(i, j, heap) {
    const temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
  }
}
```

Time: O(n) (where n is the # of nodes) since the majority of the nodes are at the bottom so O(logn) ≈ O(1) therefore it is not O(nlogn) but O(n)

O(nlogn) with SiftUp bc the majority of the nodes that will take the longest are at the bottom (parent Node is O(1), child is O(1), etc.)

Time: O(logn) (where n is # of nodes in the heap) since at each node, we only consider half the tree going forward (so time ∝ height of tree)

Space: O(1) since done in place

Due to SiftDown / siftUp