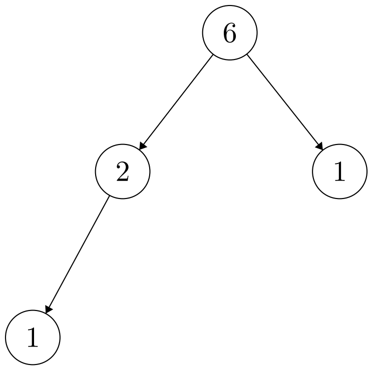
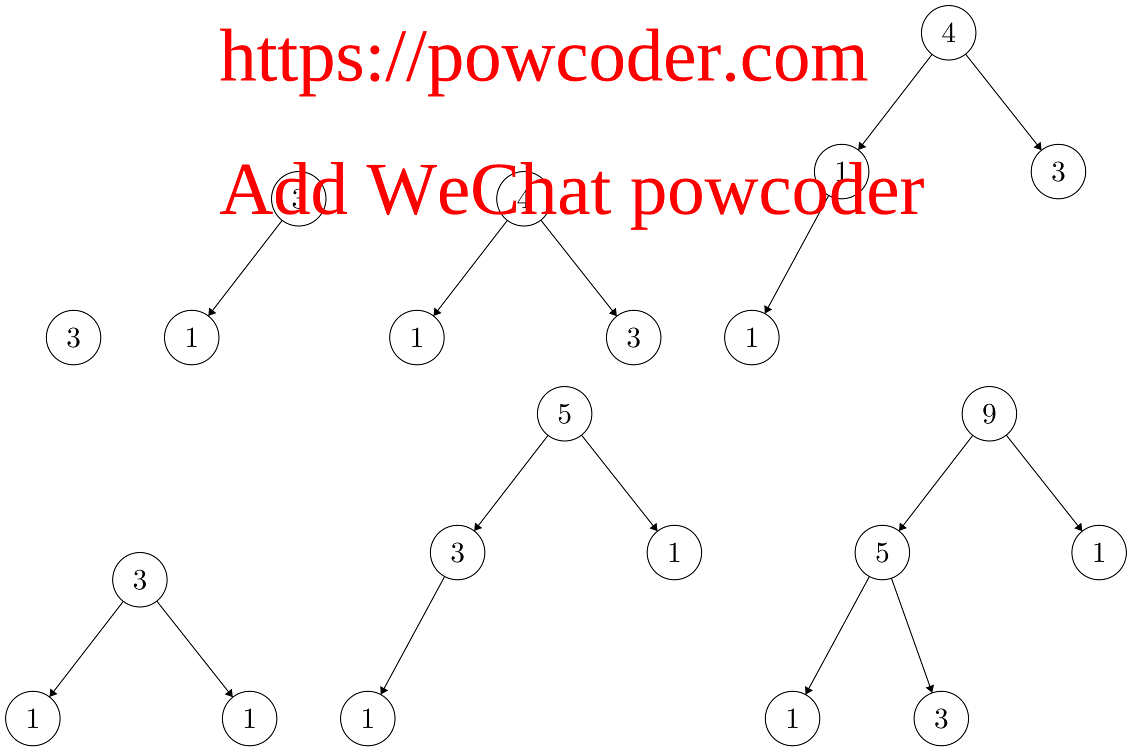
CS 124 **Section 1 Solutions** 2/3/21

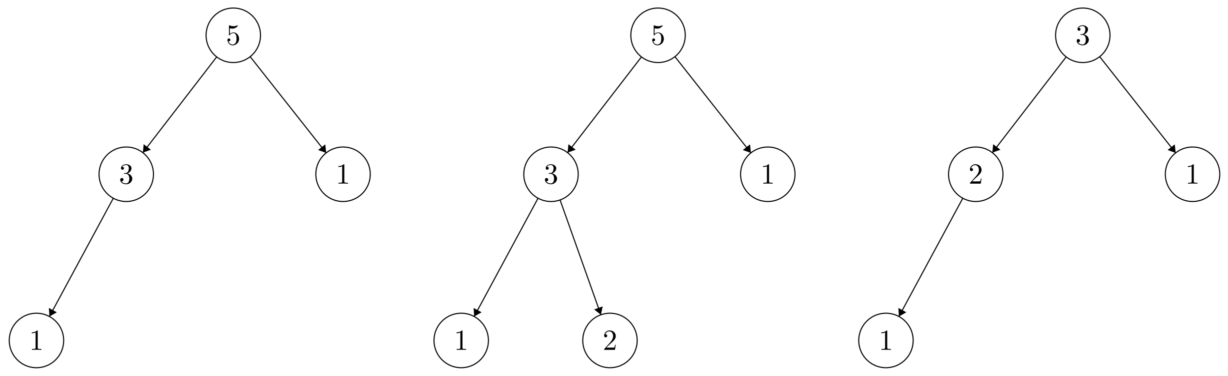
Problem 0

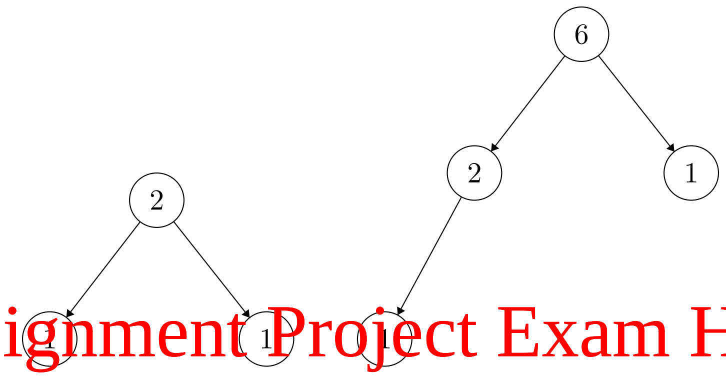
The final `MAX-HEAP` is



The sequence of intermediate `MAX-HEAPS` is below (the 4 `Extract-Max` calls return 4, 9, 5, and 3, in that order):




5
3
1
1
5
3
1
1
2
3
2
1
1


6
2
1
2
1
1
1

Problem 1

Consider the following algorithm to find the largest of a node N and its two children, if they exist, in a binary tree H:

```
Find-Max($H, N$)
Require: $N$ is a node in the binary tree $H$
 1: $(l, r) \leftarrow (\text{Left}(N), \text{Right}(N))$
 2: if Exists($l$) and $H[l] > H[N]$ then
 3:     $largest \leftarrow l$
 4: else
 5:     $largest \leftarrow N$
 6: end if
 7: if Exists($r$) and $H[r] > H[largest]$ then
 8:     $largest \leftarrow r$
 9: end if
10: return $largest$
```

We then use `Find-Max` as a subroutine in our iterative implementation of `Max-Heapify`:

```
Iterative-Max-Heapify($H, N$)
Require: $N$ is the root of a Max-Heap except, possibly, at $N$ (i.e., $N$ could be smaller than its
    children)
 1: $largest \leftarrow \text{Find-Max}(H, N)$
 2: while $largest \neq N$ do
 3:     Swap($H[N], H[largest]$)
 4:     $N \leftarrow largest$
 5:     $largest \leftarrow \text{Find-Max}(H, N)$
 6: end while
Ensure: $N$ is the root of a Max-Heap
```

As both SWAP and Find-Max take constant time, and the while loop in lines 2–4 is executed no more times than the height of the tree, the running time of Iterative-Max-Heapify is the same as that of Max-Heapify: $O(\log n)$.

Problem 2

a.) Take $A_n = [0, n-1, n-2, \dots, 1]$ (any array, which satisfies Max-Heapify's input conditions, with the root as its minimum element will work). Since the root is the minimum element in the binary tree, every (recursive) call of Max-Heapify will find N to be less than its children, resulting in a swap and another recursive call. As this occurs at each level of the tree, Max-Heapify will perform at least as many comparisons and swaps as the height of the tree, which is $\Omega(\log n)$, giving the desired lower bound.

b.) We've already shown that the upper bound $O(n)$ holds for any family $\{A_n\}_{n=1}^{\infty}$. The lower bound of $\Omega(n)$ holds as Max-Heapify, whose running time is clearly $\Omega(1)$ on any family $\{A_n\}_{n=1}^{\infty}$, is called $\lfloor n/2 \rfloor \in \Omega(n)$ times in the for loop in Build-Heap.

Problem 3

We claim that the running time is $O(n \log n)$. First, recall that Build-Heap takes time $O(n)$ and Extract-Max takes time $O(\log n)$. Since Extract-Max is repeatedly called n times, the running time of this algorithm is at most $O(n \log n)$ as the repeated calling of Extract-Max dominates both the time to Build-Heap as well as that to append elements to the array. The tightness follows from the information theoretic lower bound for comparison-based sorting. In particular, we note that the algorithm described is comparison-based. That is to say, for any two input arrays whose entries may be different numbers, but are in the same overall order, the algorithm will act in precisely the same way; the algorithm (in particular, Build-Heap and Extract-Max) makes decision based only on comparisons between elements, and not their actual values. We can lower bound the running time of the algorithm by the number of comparisons it makes. Specifically, let $C(n)$ be the number of comparisons (i.e. questions of the form "is $A[i] > A[j]$?") made by the algorithm on input arrays of size n. Since, on an input array A_n with n distinct elements, any comparison question admits exactly two answers, the total number of answers to the $C(n)$ comparison questions is $2^{C(n)}$, which gives an upper bound on the total number of possible permutations of the input array the algorithm can return as output (if two different permutations of the input array produce the same answers to all the comparison questions, then the algorithm cannot possibly act differently on them). However, since the algorithm must clearly act differently on any two different permutations of A_n (since otherwise, its output on at least one of them cannot be correct!), the total number of output permutations of the input is at least $n!$, thus implying that

$$2^{C(n)} \geq n! \implies C(n) \geq \log(n!) \in \Omega(\log(n) + \log(n-1) + \cdots + \log(1))$$

$$\geq \Omega(\log(n) + \log(n-1) + \cdots + \log(n/2)) = \Omega((n/2)\log(n/2)) \geq \Omega(n \log n).$$

Since $C(n)$ is a lower bound on the running time, the algorithm's running time is indeed $\Omega(n \log n)$ on input arrays of distinct elements, thus proving tightness.