Opening Questions

1. Explain the difference between the Binary Search Tree Parent-Child value relationship and the Heap Property Parent-Child value relationship. BST: the right node is larger than the parent node and the left node is smaller than the parent node. In Heaps, the parent is either larger than both children (MaxHeap) or smaller than both children (MinHeap).

2. Binary search trees are a dynamic data structure that uses left-child and right-child pointers to represent the tree. How is this different from a heap? In a heap, you can take the index of a node, apply $\lfloor \frac{\text{node}}{2} \rfloor$ to get the parent node. So you can wall up or down a heap whereas a binary tree can only walk down.

Heaps

Since a heap is a nearly complete binary tree and will always grow and shrink the rightmost bottom leaf, you can implement a head with an array instead of needing pointers (as is needed for a binary search tree which can grow/shrink at any node. Example of a MaxHeap showing array implementation:

1. At what index position is the largest element in a MaxHeap? We have to know how to easily move around a Heap. Can you devise a formula to relate the index of a parent to the indexes of its children? How about a formula for the index of a child to the index of its parent? The largest element in a Max Heap is at index 1 (given 1-based indexing). The formula is $\text{index\_parent} = \left\lfloor \frac{\text{child\_index}}{2} \right\rfloor$.

2. If a heap was one larger, where does the tree have to gain a node from when done? If a heap was one smaller, where does the three have to lose a node when done?

3. Considering your answer to #2, try to devise a way to ExtractMax from this maxheap. What is the runtime in terms of heapsize?

Algorithm 1 ExtractMax

```
1: function ExtractMax(heap)                    ▷ 1 based indexing, not 0
2:     Save the value at the root
3:     Move heap[heapsize] to heap[1]
4:     MaxHeapify(heap)
5: end function
```

Save the current value at the root. Then move 5 (which is at the heapsize index) to the root, and then heapify.

4. Considering your answer to #2, try to devise a way to Insert(20) into this maxheap. What is the runtime in terms of heapsize?

Both the above ExtractMax and Insert assumed we already had a heap. To efficiently build a heap, we put all the items to insert in an array. Call MaxHeapify (walk value down) from index $\frac{\text{heapsize}}{2}$ up to the root (index 1).

5. Write pseudocode for this BuildHeap algorithm and demonstrate on this data. What is the runtime in terms of n, the size of B? $B = [15\ 8\ 4\ 9\ 3\ 16]$ B is not a max heap.

Algorithm 2 BuildHeap

```
1: function BuildHeap(heap)
2:     MaxHeapify(heap, 1, heapsize/2) ▷ Precondition: Both children of that index are valid max heaps
3:                                     ▷ Any index position larger than heapsize/2 is a leaf→ valid max heap
4: end function
```

Runtime: must call $MaxHeapify$: $\frac{heapsize}{2}$ times. $MaxHeapify$ in the worst case runs at $O(\lg heapsize) = O(\lg n)$. However, working at $heapsize$, at most $MaxHeapify$ works once (incase the only child is larger than the node).

$$\begin{aligned}\theta(n) &= \lg(n) + 2\lg(n-1) + 4\lg(n-2) + \cdots + \left(\frac{n}{4}\right)(1) + \frac{n}{2}(0) \\ &= \text{linear time.}\end{aligned}$$

6. Write the loop invariant for BuildHeap and prove that it works.After the jth iteration (index of the item we call MaxHeapify on: (range $\frac{n}{2} \to 1$). Before call on MaxHeapify on jth iindex in the heap $\frac{n}{2} \to 1$, $\frac{n}{2} + 1 \to n$ are max heaps, which means child of j index are max heaps.
INIT: We know all index positions $> n/2$ are leafs or empty, therefore, they are already maxheaps.
MAINT: If $p(j) \to p(next\_j)$

7. How can we use a maxHeap and extractMax to sort? Build the heap in $O(n)$ time, then extractMax which saves the root, then sorts the heap. The initially saved root gets moved to another array into the index $heapsize - 1$. Then extract max is run again, and the saved value is moved into $heapsize - 2$. The runtime is $O(n \lg n)$.