Readings

- [Lecture Notes Chapter 14: Binary Heaps and Heapsort]

Review: Heaps

A heap is a tree-like data structure that implements the priority queue ADT, which allows us to maintain a set of elements, each with an associated key, and select the element with highest/lowest priority. Heaps satisfy the two following properties:

**Heap Property:** In a max-heap, for each node $i$, we have $A[\text{Parent}(i)] \geq A[i]$, so the maximum value is stored at the root. In a min-heap, for each node $i$, we have $A[\text{Parent}(i)] \leq A[i]$, so the minimum value is stored at the root.

**Shape Property:** A heap is an almost complete binary tree, meaning that every level of the tree is completely filled except for the last, which must be filled from left to right.

Implementation Details

Because a heap is an almost complete binary tree, we are able implement it using an array as shown below:

```
null  16  14  10  8  7  9  3  2  4  1
```

Observe that we can populate the array from left to right by doing a level-order traversal of the tree, where we start from the root and go through each level of the tree from left to right. Additionally, because of the shape property, if the root is stored at index 1 of the array, given a node at index $i$, its left child can be found at index $2i$, its right child can be found at index $2i + 1$, and its parent can be found at index $\lfloor i/2 \rfloor$. 
Operations (Max-Heaps)

**Max-Heapify** allows us to maintain the max-heap property at the node it is called on. More specifically, given a node whose children are both max-heaps, we can call **Max-Heapify** on the node so the entire subtree rooted at the node will now be a max-heap. It works by allowing the node to “float-down” through the tree; at each level, we swap it with its largest child, or if it is larger than both of its children, we terminate since the max-heap property holds. It runs in $O(\log n)$ time or $O(h)$ time, where $h$ is the height of the heap, since in the worst-case, the node must “float-down” through the entire tree.

**Build-Max-Heap** constructs a max-heap from an unsorted array by repeatedly calling **Max-Heapify** on each node from the “bottom-up”, starting at the nodes right above the leaves (which by definition are max-heaps!). It runs in $O(n)$ time. We know that an $n$-element heap has height $\lceil \log n \rceil$, calling **Max-Heapify** at any height $h$ takes $O(h)$ time, and there are at most $\lceil n/2^{h+1} \rceil$ nodes at any height $h$ of an $n$-element heap. So, we can express the running time as

$$
\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) \leq O \left( \frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} \right)
$$

$$
= O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)
$$

$$
= O(n \cdot 2) \quad \text{(arithmetico-geometric series)}
$$

$$
= O(n)
$$

**Extract-Max** allows us to remove and return the element with the maximum key. It works by removing the root, replacing it with the right-most element in the bottom level/last element in the array, and then calling **Max-Heapify** on the “new” root to maintain the max-heap property. Removing the root and replacing it with the “last” element in the heap takes $O(1)$ time, and calling **Max-Heapify** takes $O(\log n)$ time, so it runs in $O(\log n)$ time.

**Insert** allows us to add an element to our max-heap. It works by adding the element to the end of the array/max-heap, and then allowing it to “float-up” to its correct position in the max-heap by repeatedly swapping it with its parent as necessary to maintain the max-heap property. It runs in $O(\log n)$ time, since the path it takes while it “floats-up” has length $O(\log n)$.

**Peek** returns the maximum element in the heap, which is stored at the root. Since we implement a heap with an array, this runs in $O(1)$ time because we just have to index into the array.

### Problems

**Problem 1**

Given a data stream of $n$ test scores, design an $O(n \log k)$ time algorithm to find the $k$-th highest test score. Since PEFS provides minimal monetary resources, CIS 121 staff have limited access to storage space and can only afford you $O(k)$ space, where $k \ll n$.

**Midterm 1 Review Problems**

**Problem 2**

Provide a running time analysis of the following loop. That is, find both Big-O and Big-$\Omega$:

```cpp
for (int i = 0; i < n; i++)
    for (int j = i; j <= n; j++)
        for (int k = i; k <= j; k++)
```
Problem 3

Assume $n$ is a power of 3.

$$T(n) = \begin{cases} 
2T(\frac{n}{3}) + n & n > 1 \\
1 & \text{otherwise}
\end{cases}$$

Solve this recurrence by expansion (do not use Simplified Master Theorem).