Problem Set 3

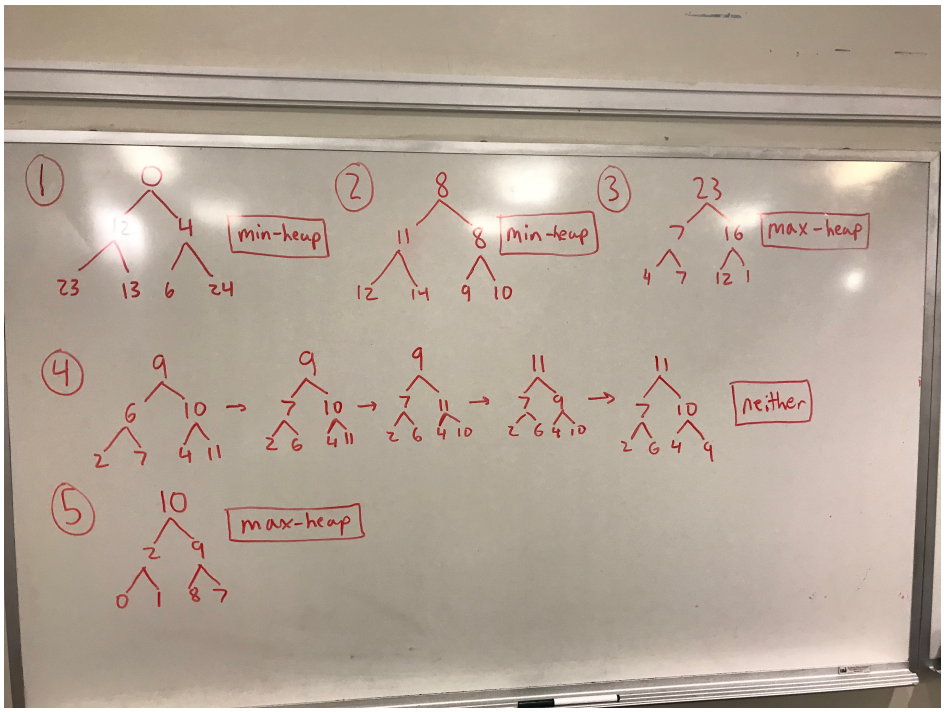
All parts are due Thursday, September 27 at 11PM.

Name: Kevin Jiang

Collaborators: Suraj Srinivasan, Jason Lu

Problem 3-1.

(a)



(b) Since $k = 1$ is the minimum of K, 1 has to be at A since it has no parents. Key $k = 2$ has to be in (B, C) since it is the next smallest. Key $k = 7$ has to be in (D, E, F, G) since it is the largest. Key $k = 6$ has to be in (D, E, F, G) too since only $k = 7$ is larger (can't be a parent to 2 children). Keys $k = 3, 4, 5$ can be anywhere but A.

$$\boxed{1 : A, 2 : (B, C), (3, 4, 5) : (B, C, D, E, F, G), (6, 7) : (D, E, F, G)}$$

(c) Note that the second largest element must be either the second or third element. Thus, we compare these two elements to see which is larger ($O(1)$ time) to find the second largest element. Next, we replace the larger element with k ($O(1)$ time) and perform at most $O(\log n)$ swaps downward (max heapify downwards) to maintain the max heap structure. Therefore, we can do setsecondlargest(A,k) in $O(\log n)$ time.

Problem 3-2.

(a) We can use a linked list data structure. Each item in the structure would contain a tuple $(\text{weight}, \text{species})$. Recording the weight and species of a catch takes $O(1)$ time since inserting a tuple to the left or right of a linked list takes $O(1)$ time (items are connected via pointers). Querying the species of the heaviest catch also takes $O(1)$ time since we can continuously compare the newest catch with the first element in the linked list. If the new catch is heavier, then we can insert it to the left. Otherwise, we can just add it to the end since we only care about the biggest catch. Returning the first element will give us the biggest catch.

(b) We can use a max heap data structure. Each element can be stored as a tuple containing the fish's weight and species. Recording the weight and species of a catch takes $O(\log n)$ time since we perform at most $\log n$ (the depth) swaps to maintain a max heap structure. Thus, we will always have the heaviest catch at the top of the tree in $O(\log n)$. To remove the heaviest catch, we can pop off the first element and replace it with the last element. We can then perform at most $O(\log n)$ swaps to maintain the max heap structure.

(c) We can use a max-heap and min-heap data structure. Note that to find the median, the element immediately to the left must be the biggest in the left half, and the element immediately to the right must be the smallest in the right half. Therefore, we can create a max-heap and a min-heap, where the max element in the max-heap is smaller than the min element in the min-heap. If we keep the sizes of the max-heap and min-heap as close as possible to each other, then the median will be either the max of the max heap, x, or the min of the min heap, y. We will keep a counter c for the number of total elements. For each element e, we will compare it to x and y.

If $e < x$, then we put e into the max heap, and max heapify up to maintain the max heap. If the size of the max heap is greater than the size of the min heap, then we remove the max x and put it in the min heap (max heapify down and min heapify up if necessary).

If $e > y$, then we put e into the min heap, and min heapify up to maintain the min heap. If the size of the min heap is greater than the size of the max heap, then we remove the min y and put it in the max heap (min heapify down and max heapify up if necessary).

If c is odd, then the median will be x if the size of the max heap is bigger and y otherwise. If c is even, then the median can be either x or y. WLOG suppose x is the median. We can remove x by swapping it with a leaf, popping it off, and max heapify down to maintain a max heap structure.

This algorithm is correct since the median is either the maximum of the left half or the minimum of the right half. Each of our steps keeps track of the maximum of the left half and minimum of the right half and removing the median still holds our

structure. The runtime of this algorithm is amortized $O(\log n)$ time since we are only making constant work comparisons and max/min heapifying up/down, which is $O(\log n)$ work, for each element.

(d) We can use a min-heap structure. Since we wish to keep the k heaviest fish, we need to remove the smallest fish every time we catch something. Therefore, we use a min-heap structure to keep track of the smallest fish. Whenever a fish f is caught, we add it to the min-heap and min-heapify up. Next, we switch the absolute min with a leaf, remove it, and then min-heapify down to maintain the k heaviest fish.

This algorithm is correct since we are always removing the absolute min (due to the nature of a min-heap), so we are keeping track of the k heaviest fish. This algorithm has an amortized run time of $O(\log n)$ since we are only adding/removing fish in $O(1)$ time and doing min-heapify up/down in $O(\log n)$ time.

Problem 3-3.

(a) If each element in a k-proximate array of length n needs at most k swaps to be sorted, then sorting will take $O(nk)$ time since we do at most k swaps for each of the n elements.

(b) We claim the following algorithm can sort a k-proximate array in $O(n \log k)$ time:

Given a k-proximate array A of length n, we know that the absolute minimum element must lie within the first $k+1$ elements. We can generate a min-heap from these $k+1$ elements and pop off the minimum element into our result list. By repeatedly adding the rest of the elements individually to our min-heap and popping off the minimum, we can sort a k-proximate array.

This algorithm is correct since we are always popping off the absolute minimum of the remaining array (due to the fact that a k-proximate array guarantees an absolute minimum within $k+1$ elements). This algorithm has a runtime of $O((k+1)\log(k+1) + n\log(k+1)) = O(n \log k)$ since we are first creating a min-heap with $k+1$ elements, which takes $O((k+1)\log(k+1))$ times, and min-heapifying up $O(n \log(k))$ work.

(c) Submit your implementation to `alg.mit.edu/PS3`