Problem Set 3

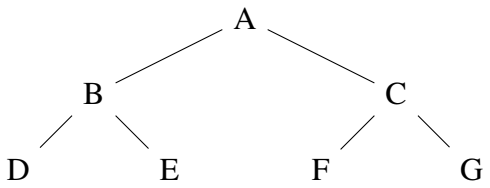
All parts are due on September 27, 2018 at 11PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 3-1. [25 points] **Heap Practice**

(a) [10 points] For each array below, draw it as a left-aligned complete binary tree (according to the transformation from Lecture 5) and state whether the tree is a max-heap, a min-heap, or neither. If the tree is neither, turn the tree into a max-heap by repeatedly swapping adjacent nodes of the tree. You should communicate your swaps by drawing a sequence of trees, with each tree depicting one swap.

1. `[0, 12, 4, 23, 13, 6, 24]`
2. `[8, 11, 8, 12, 14, 9, 10]`
3. `[23, 7, 16, 4, 7, 12, 1]`
4. `[9, 6, 10, 2, 7, 4, 11]`
5. `[10, 2, 9, 0, 1, 8, 7]`

(b) [10 points] Consider the following binary tree on seven nodes labeled A – G.



Assume the tree contains the keys $K = \{1, 2, 3, 4, 5, 6, 7\}$ (each occurring exactly once) such that the min-heap property is satisfied. For each key $k \in K$, list the node(s) that could contain key k.

(c) [5 points] Given a max-heap represented by array A, and an integer key k, describe an algorithm SET_SECOND_LARGEST(A, k) that finds the item currently stored in A having the second-largest key, and modifies it to have key k, while preserving the max-heap property on A. Your algorithm should run in $O(\log n)$ time, where $n \geq 2$ is the length of input heap A. Remember to argue correctness and running time of your algorithm. Your algorithm may make use of any heap operations described in lecture or recitation as a black box, without having to repeat their description, analysis, or correctness arguments.

Problem 3-2. [30 points] **Fishing Frameworks**

Frankie the Fisher frequently fishes in the fjords of Finland. To aid her business, she wants to build a mobile app called Fishr to help keep track of her inventory. For each of the following questions, describe a data structure to support the requested operations. When describing a data structure in this class, you must first describe how to store the data, and then describe an algorithm operating on the data to support each requested operation. As with any algorithm in this class, you should briefly argue correctness and running time for each supported data structure operation. For parts (a), (b), and (c), operation running times depend on the number n of fish stored in the data structure **at the time of the operation**.

(a) [5 points] Whenever Frankie catches a fish, she wants to be able to record its weight and species on the app. In addition, at any point during a fishing trip, she wants to be able to query the species of her heaviest catch on the trip so far. Describe a data structure that supports two operations: recording the weight and species of a catch, and querying the species of the heaviest catch. Both operations should run in worst-case $O(1)$ time per operation.

(b) [5 points] Frankie's fishing trips often last for months, so when she gets hungry, she will eat the heaviest fish in her inventory (Frankie dislikes eating small fish). Describe a data structure supporting the same two operations as part (a), plus an operation that removes the heaviest fish from her current inventory. All operations should run in worst-case or amortized $O(\log n)$ time per operation.

(c) [10 points] Frankie the Fisher has finally figured out that fleshier fish fetch fatter fees! To finance her future funding, Frankie decides to stop eating the heaviest fish in her inventory when she's hungry, and to eat the fish with **median** weight instead. Describe a data structure supporting the same operations as part (b), except that the second and third operations apply to the median weight fish instead of the heaviest. (For even n, the operations may reference either of the two middle fish in the sorted order by weight.) All operations should run in worst-case or amortized $O(\log n)$ time per operation.

(d) [10 points] Frustration! Many of Frankie's fish tanks have flung overboard in a furious storm, and she now only has enough space to store at most k fish on her boat at any given time, even though she may catch $n \gg k$ fish during her trip. She resolves to continue fishing, but to only keep the k heaviest fish that she sees on the trip. After catching the first k fish, each time she catches a new fish, she must choose some fish to throw back (or eat), which may or may not be the fish she just caught. Describe a data structure supporting two operations: adding a new catch to her inventory, and identifying a fish to discard, in order to maintain the heaviest k fish on her boat at all times. Be sure to argue why Frankie ends up with the k heaviest fish at the end of her trip! All operations should run in worst-case or amortized $O(\log k)$ time per operation.

Problem 3-3. [45 points] **Proximate Sorting**

An array of **distinct** integers is ***k-proximate*** if every integer of the array is at most k places away from its place in the array after being sorted, i.e., if the ith integer of the unsorted input array is the jth largest integer contained in the array, then $|i - j| \leq k$. In this problem, we will show how to sort a k-proximate array faster than $\Theta(n \log n)$.

(a) [5 points] Prove that insertion sort (as presented in this class, without any changes) will sort a k-proximate array in $O(nk)$ time.

(b) [15 points] $\Theta(nk)$ is asymptotically faster than $\Theta(n^2)$ when $k = o(n)$, but is not asymptotically faster than $\Theta(n \log n)$ when $k = \omega(\log n)$. Describe an algorithm to sort a k-proximate array in $O(n \log k)$ time, which is always nonstrictly faster than $\Theta(n \log n)$. (Remember to argue the correctness and running time of your algorithm.)

(c) [25 points] Write a Python function `proximate_sort` that implements your algorithm from part (b). You can download a code template from the webiste containing some test cases. You may adapt any code presented in lecture or recitation, but for this problem, **you may NOT import external packages** and **you may NOT use Python's built-in sort functionality** (the code checker will remove `List.sort` and `sorted` from Python prior to running your code). Submit your code online at `alg.mit.edu`.

```
def proximate_sort(A, k):
    '''
    Return an array containing the elements of
    input tuple A appearing in sorted order.
    Input:  k | an integer < len(A)
            A | a k-proximate tuple
    '''
    ##################
    # YOUR CODE HERE #
    ##################
    return []
```