This homework is due Thurday, April 16, at the start of class. The questions are drawn from the material in the lectures and Chapter 17 of the text on *amortized analysis*.

The homework is worth a total of 100 points. When questions with several parts do not specify the points for each part, each part has equal weight.

Remember to write on just one side of a page, do not use scrap paper, put your answers in the correct order, and staple your pages together. If you can't solve a problem, state this, and write only what you know to be correct. Neatness and conciseness count.

(1) **(Simulating a queue using stacks)** (10 points) Show how to implement the *queue* data structure by using two *stacks*, so that the amortized time for queue operations in the stack-based implementation matches their worst case time in a standard queue implementation. More specifically, show how to implement the operations

- `Put`(x, Q), which adds element x to the rear of queue Q, and
- `Get`(Q), which removes the element x on the front of queue Q and returns x,

so that both operations run in $O(1)$ amortized time. Use the potential function method for your analysis.

(2) **(Deleting the larger half)** (20 points) Design a data structure that supports the following two operations on a set S of integers:

- `Insert`(x, S), which inserts element x into set S, and
- `DeleteLargerHalf`(S), which deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Show how to implement this data structure so both operations take $O(1)$ amortized time. Use the accounting method for your analysis.

(3) **(Constant amortized time extract)** (10 points) Show that, by an appropriate choice of a potential function, the standard implementation of the implicit heap used in heap sort takes $O(1)$ amortized time for an `Extract`, and $O(\log n)$ amortized time for an `Insert`.

(Note: In your solution, (a) specify how you concretely measure the real time for these two operations, (b) specify your potential function for the heap, and (c) analyze the amortized time for both operations. Implicit heaps are described in Section 6.5 of the text.)

(4) **(Binary search with insertions)** (25 points) Storing a set of n key-item pairs (k_i, x_i) in an array sorted by keys allows us to efficiently *find* an item x_i by its key k_i using binary search in $O(\log n)$ time. To *insert* a new element into such a sorted array is very inefficient, however, and takes $\Theta(n)$ time in the worst case. By using multiple sorted arrays, we can achieve a much better balance between the time for finding and inserting elements.

Suppose we store the n elements in a data structure D that uses $\ell = \lceil \lg(n{+}1) \rceil$ different arrays. We refer to these arrays as $A_1, A_2, \ldots, A_\ell$. Array A_i has length 2^{i-1}, and is *full* if bit i is 1 in the *binary representation* of n; otherwise, if bit i is 0, array A_i is *empty*. (So for $n = 5$, which is `101` in binary, arrays A_1 and A_3 are full and have 1 and 4 elements respectively, while array A_2 is empty.) Each array A_i is *sorted* by its keys, but we do not maintain any ordering relationship between the keys in different arrays.

We wish to support two operations on this data structure D:

- `Find`(k, D), which returns the item x associated with key k in D, and
- `Insert`(k, x, D), which inserts the pair (k, x) into D.

You may assume that for `Find`, key k is in D, and for `Insert`, key k is not already in D.

(a) (5 points) Show how to implement `Find` so it takes $O(\log^2 n)$ worst-case time.

(b) (20 points) Show how to implement `Insert` so it takes $O(\log n)$ *amortized* time, even though it can take $\Theta(n)$ worst-case time.

(Hint: Review the material in Section 17.1 of the text on incrementing a binary counter. The aggregate method may be convenient for your amortized analysis in Part (b).)

(5) **(Amortized search trees)** (35 points) For binary search tree T and node x in T, let

- $s(x)$ be the size of the subtree rooted at x,
- $\ell(x)$ be the left child of x, and
- $r(x)$ be the right child of x.

For constant α, where $\frac{1}{2} \leq \alpha < 1$, tree T is said to be *α-balanced* if at every node x of T,

$$\begin{aligned} s(\ell(x)) &\leq \alpha\, s(x), \\ s(r(x)) &\leq \alpha\, s(x). \end{aligned}$$

Below we develop a very simple implementation of α-balanced search trees in which the operations `Insert` and `Delete` take $O(\log n)$ *amortized* time.

(a) (10 points) Show that an arbitrary n-node tree can be made $\frac{1}{2}$-balanced in $\Theta(n)$ time using $\Theta(n)$ space.

(b) (5 points) Show that performing a `Find` operation in an n-node α-balanced binary search tree takes $O(\log n)$ worst-case time.

(c) (10 points) Consider the following amortized approach for supporting the `Insert` and `Delete` operations on a search tree. Suppose `Insert` and `Delete` are implemented in the standard straightforward way for an ordinary search tree that is *not* balanced, except that now after an `Insert` or `Delete`, the tree is rebalanced by the following approach. After the `Insert` or `Delete`, find the highest node x in the tree that is not α-balanced, and rebuild the subtree rooted at x so it becomes $\frac{1}{2}$-balanced, using your solution to Part (a). We call this task of rebuilding the subtree at x in this way, *rebalancing* the tree.

Prove that rebalancing an n-node α-balanced tree, where $\alpha > \frac{1}{2}$, takes $O(1)$ *amortized* time.

To prove this, use the *potential method* with the following potential function $\Phi(T)$. For a node x in T, let

$$d(x) \;:=\; \Big| s(\ell(x)) \;-\; s(r(x)) \Big|.$$

Then

$$\Phi(T) \;:=\; \frac{1}{2\alpha - 1} \sum_{\substack{x \in T \\ d(x) \geq 2}} d(x).$$

(d) (10 points) Using your answer to Part (c), show that an `Insert` or a `Delete` on an n-node α-balanced tree, where $\alpha > \frac{1}{2}$, takes $O(\log n)$ *amortized* time.