Problem 2.1  [Constant-Time Concatenate].

Solution: This solution will use two stacks to represent the queue at large. One stack, $H$, will contain the head elements, and the other stack, $T$, will contain the tail elements. For each version of the DS, we will simply keep pointers to the “top” element of each of these stacks (top element for this version).

- $\text{insert-last}(Q, x)$: Insert $x$ onto stack $T$ and set the pointer for the head of this version’s $T$ to $x$.
- $\text{delete-first}(Q)$: Move the pointer for the head of $H$ down one element, and return the updated stacks.

The problem that now needs to be solved is shifting elements from $T$ to $H$ so that the head of $H$ actually contains the first inserted element. To do this, we maintain the invariant that $H$ must always contain more elements than $T$. If an insert or delete operation ever causes $T$ to be larger than $H$, we spawn a process to “fix” the size disparity.

1. Set $T$ to a new empty stack.
2. Create a copied stack of the old $T$ elements but reversed. (copy each element off the top of $T$ and insert into a new stack.) Call this new stack $H_{\text{new}}$.
3. Create a copied stack of the old $H$ elements but reversed. Call this new stack $T_{\text{rev}}$.
4. Copy each of the elements off the top of $T_{\text{rev}}$ onto $H_{\text{new}}$.
5. Set $H$ to $H_{\text{new}}$.

This entire process will produce a new $H$ that contains all of the elements in the order of least recently inserted.

To achieve a de-amortized runtime, this entire process must be split up into chunks. Thus, for every insert and delete operation, we will do 3 of the operations required for this process. We will do this in order of the above steps. We will keep track of which “step” we are on and keep a pointer to the current stack location of whichever stack we are currently working on. After the invariant of $T$ becoming larger than $H$ occurs, we will do 3 of these operations per insert or delete until it is complete.

This introduces another case for step 4 as $T$ may have changed and lost elements during this whole process. To keep track of this, we keep a counter of the elements in $T$ throughout the whole process. The final size of $T$ when the process in complete is the same amount of elements that should be copy popped and added to $H_{\text{new}}$ in step 4. This is to account for the elements in $T$ that are no longer in the structure.

Runtime: Each of these ops is $O(1)$ normally (when we are not “fixing” and balancing the stacks) as we simply push or pop onto a stack and increment a pointer. During the rebalancing process, runtime is still $O(1)$ as we are only doing 3-4 pops/copies and pointer increments per insert/delete.

We can also show that only one of these processes will ever be ongoing for any given insert/delete. Steps 2-4 of the process each take $n$ copy pops, where $n$ is the amount of elements in $T$ when the process is initiated (number of elements in $T$ is equal to $H$ when this occurs). Thus, the total needed operations of the process is $3n$. By doing three of these operations per insert/delete, we complete the process in $n$ inserts/deletes. Since the beginning of a new rebalancing process resets $T$ to be empty, the difference in size between $T$ and $H$ is $n$ when the process starts. Each insert/delete operation during the process decreases the size difference between by $T$ and $H$ by 1 ($H$ either decreases by 1 or $T$ grows by 1). Thus, the process will not be spawned again until at least $n$ inserts/deletes have occurred, at which point the previous process will be completed.

This structure is functional because elements are never removed or modified, only pointers moved and elements copied.