Functional Queues

Gaurav Arya

May 22, 2021

1 Functional data structures

A functional data structure can never be modified. Instead, when performing an operation, a new copy of the data structure is returned. This implies a property called *full persistence*: previous versions of the data structure remain accessible forever, and we can perform operations on any version we choose.

2 Functional stacks

A functional stack is probably the simplest example of a functional data structure. It consists solely of a pointer to the head of a one-way linked list, which contains the stack's elements from top to bottom. To push, we add a new element to the one-way linked list, and return the pointer to this new element. To pop, we return a pointer to the second element of the linked list.

3 What about a functional queue?

Functional queues are a natural next step. Ideally, we would like to implement them with a constant number of changes per enqueue/dequeue. This turns out to be much harder than for stacks! But there is a solution: using our functional stacks as a black box, we can simulate each queue operation using a constant number of operations on a set of six stacks. The primary goal of our visualizer is to show these operations in action, and how they come together to make a functional queue.

4 Outline of functional queue implementation

4.1 Overview

We will represent our functional queue `Q` by a tuple ($\texttt{INS}, \texttt{POP}, \texttt{POP}_{rev}, \texttt{POP}_2, \texttt{INS}_2, \texttt{HEAD}, \texttt{n}, \texttt{ops\_left}$). Here, `INS`, `POP`, $\texttt{POP}_{rev}$, $\texttt{POP}_2$, $\texttt{INS}_2$ and `HEAD` are functional stacks, and `n` and `ops_left` are integers. When `Q` is created, all stacks are empty.

Broadly speaking, our queue `Q` works by maintaing a stack `INS` that contains the most recently added elements of `Q`, and a stack `POP` that contains the remaining elements. `INS` keeps more recently added elements higher, whereas `POP` keeps less recently added elements higher. Thus, `INS` will allow for easy insertions into our queue, while `POP` will allow for easy pops from our queue.

At any moment, `Q` is either in "normal mode" or "transfer mode". In normal mode, `Q` maintains the invariants above, and does not worry about the other stacks: inserted elements are placed into INS[1], and deletions pop from POP. In normal mode, operations are clearly $O(1)$, and as long as POP is non-empty these operations will be valid.

However, if the size n of INS becomes equal to the size of POP, Q will switch into transfer mode for the next $2\,\mathtt{n} - \mathtt{d}$ operations, where $\mathtt{d} \geq 0$ is the number of deletions that occur during the transfer mode. At the end of transfer mode, all elements of INS will be moved into POP. We now describe how this happens.

4.2 Transfer Mode

4.2.1 Initializing

To begin, HEAD points to the top of the POP stack, and $\mathtt{POP}_{rev}$, $\mathtt{POP}_2$, and $\mathtt{INS}_2$ are empty. We set ops_left to $2\,\mathtt{n}$, which will keep track of how many operations are left in transfer mode.

4.2.2 Passive operations

The following happens independent of the operation type. We always reduce ops_left by 1. For the first n operations of transfer mode, we will:

- Pop an element from POP and add it into $\mathtt{POP}_{rev}$.
- Pop an element from INS and add it into $\mathtt{POP}_2$.

For the next $\mathtt{n} - \mathtt{d}$ operations of transfer mode, we will pop an element from $\mathtt{POP}_{rev}$ and add it into $\mathtt{POP}_2$. We are able to tell when transfer mode ends by checking when ops_left is 0. Note that since we stop after $\mathtt{n} - \mathtt{d}$ operations, we do not copy elements of $\mathtt{POP}_{rev}$ into $\mathtt{POP}_2$ if they have been deleted.

4.2.3 Insertion

We simply place the element into $\mathtt{INS}_2$.

4.2.4 Deletion

First, we reduce ops_left by 1. Then, we move the HEAD pointer down by one (apply the tail operation to the stack it points to), and return the value it points to. Note that there can be at most n deletions before termination of transfer mode, and hence that we can always apply the tail operation.

4.2.5 Cleanup

At the end of transfer mode, $\mathtt{POP}_2$ has become the amalgation of the original INS and POP, with elements in decreasing order of recency, as desired. Meanwhile, $\mathtt{INS}_2$ has collected all the elements that have been inserted during transfer mode. So we simply assign POP to $\mathtt{POP}_2$ and INS to $\mathtt{INS}_2$, and return to normal mode. Note that transfer mode lasts for $2\,\mathtt{n} - \mathtt{d}$ operations, and when it ends POP has size $2\,\mathtt{n} - \mathtt{d}$. Thus, after transfer mode ends, the size of INS cannot exceed the size of POP. This justifies our assumption that during normal mode INS has size at most the size of POP, and that transfer mode is triggered when INS grows in size to become equal in size to POP.

[1] as a special case, however, when an element is inserted into an empty queue we place it directly into POP.

References

[1] Robert Hood and Robert Melville. “Real-time queue operations in pure LISP”. In: *Information Processing Letters* 13.2 (1981), pp. 50–54. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(81)90030-2. URL: https://www.sciencedirect.com/science/article/pii/0020019081900302.