CSC265 Fall 2020 Homework Assignment 6

due Tuesday, October 27, 2020

1. Consider the abstract data type MAXQUEUE that maintains a sequence S of integers and supports the following three operations:

   DEQUEUE(S) , which removes the first element of S and returns its value. If S is empty, it returns NIL.

   ENQUEUE(S, x) , which appends the integer x to the end of S.

   MAXIMUM(S) returns the largest integer in S, but does not delete it.

   An element x is a *suffix maximum* in a sequence if all elements that occur after x in the sequence are smaller in value. In other words, when all elements closer to the beginning of the sequence are dequeued, x is the unique maximum in the resulting sequence. For example, in the sequence 1,6,3,5,3, the suffix maxima are the second, fourth, and fifth elements.

   One way to implement the MAXQUEUE abstract data type is to use a singly-linked list to represent the sequence, with pointers *first* and *last* to the first and last elements of that list, respectively. In addition, have a doubly-linked list of the suffix maxima, sorted in decreasing order, with a pointer *maxima* to the beginning of this list. Note that the elements in the suffix maxima list are in the same order as they are in the sequence and the last element of the suffix maxima list is the last element of the sequence.

   Assume that each element of the sequence is represented as a record with four fields:

   - *num*, containing the integer value of the element;
   - *next*, containing a pointer to the record of the next element in the sequence (or NIL if it is the last element of the sequence);
   - *nextSuffixMax*, containing a pointer to the record of the next suffix-maximum in the sequence (or NIL if the element is not a suffix-maximum or is the last suffix-maximum); and
   - *prevSuffixMax*, containing a pointer to the record of the previous suffix-maximum in the sequence (or NIL if the element is not a suffix-maximum or is the first suffix-maximum).

   (a) Draw a picture of this data structure for the sequence $S = 1, 6, 3, 5, 3$.

   (b) Explain how to perform the operations DEQUEUE, ENQUEUE, and MAXIMUM so that they all have $O(1)$ amortized time complexity. Explain why your algorithms are correct.

   (c) Using the accounting method, prove that, starting from an empty sequence, all three operations have O(1) amortized time complexity.

2. It is easy to store 2 stacks in an array $A[1..N]$ provided the sum of the number of items in both stacks is always at most N. The idea is to have stack S_1 growing to the right from the left side of the array and have stack S_2 growing to the left from the right side of the array. One would also need two values $z_1, z_2 \in \{0, 1, \ldots, N\}$, where $z_i \neq 0$ indicates the location in A of the top element in stack S_i and $z_i = 0$ indicates that stack S_i is empty. PUSH and POP can be performed in $O(1)$ time.

Now suppose we want to store 3 stacks in the array $A[1..N]$ provided the sum of the number of items in the three stacks is always at most N. This is more difficult. Here's one way to do it, assuming $N = b^2$, where $b \geq 4$. The array is viewed as being divided into b blocks of size b, $B_1, \ldots, B_b$, so $B_k[j] = A[(k-1)b + j]$.

Each block B_i has an associated value $P_i \in \{0, \ldots, b\}$. In addition, there are 4 values, $F_0, F_1, F_2, F_3 \in \{0, \ldots, b\}$. These values represent four singly linked lists of blocks, where i denotes block B_i and 0 denotes a NIL pointer. Every block is in exactly one of these linked lists.

For $1 \leq i \leq 3$, the blocks in the list pointed to by F_i have been allocated to stack S_i. If $F_i \neq 0$, then the first $f_i \in \{0, \ldots, b\}$ items in block B_{F_i} are in stack S_i with item $B_{F_i}[j]$ immediately below item $B_{F_i}[j+1]$ in the stack, for $1 \leq j < f_i$. All of the items in the remaining blocks B_k in this list are in stack S_i, with item $B_k[j]$ immediately below item $B_k[j+1]$ in the stack, for $1 \leq j < b$. Furthermore, if $P_h = k \neq 0$, then item $B_k[b]$ is immediately below item $B_h[1]$ in the stack. The values f_1, f_2, and f_3 are stored explicitly in the data structure. The blocks in the list pointed to by F_0 have not been allocated to any of the three stacks. This list is called the *free list*. The first block in the free list, B_{F_0}, is called the first free block, the second block in the free list is called the second free block, etc.

For $1 \leq i \leq 3$, the items in stack S_i that are not stored in blocks allocated to stack S_i (i.e., not in blocks in the list pointed to by F_i) are called residual. The top $t_i \in \{0, \ldots, 2b-1\}$ items in stack S_i are residual. The total number of residual elements in all three stacks is $t_1 + t_2 + t_3 \leq 4b - 3$. The values t_1, t_2, and t_3 are stored explicitly in the data structure.

If $1 \leq i \leq 3$ and $F_i \neq 0$, then some residual items may be stored in $B_{F_i}[f_i + 1..b]$. Residual item may also be stored in the first 4 free blocks (or in all free blocks, if there are fewer than 4 free blocks). Let x be the number of these locations that do NOT contain a residual item. The first x elements of the array $X[1..7b]$ represents a stack containing these x unused locations in A.

For $1 \leq i \leq 3$, the locations in A of the top t_i items in stack S_i are stored in the array $T_i[1..2b-1]$. If $t_i \neq 0$, then $A[T_i[t_i]]$ is the top item of stack S_i and, if $1 \leq j < t_i$, item $A[T_i[j]]$ is immediately below item $A[T_i[j+1]]$ in stack S_i.

Note that, in addition to the array $A[1..N]$, the data structure uses $\Theta(b \log N) = \Theta(\sqrt{N} \log N)$ bits of memory.

Initially, the three stacks S_1, S_2, and S_3 are empty, $P_i = i + 1$ for $1 \leq i < b$, $P_b = 0$, $F_0 = 1$, $F_1 = F_2 = F_3 = 0$, $t_1 = t_2 = t_3 = 0$, $x = 4b$, $X[i] = i$ for $i = 1, ..., x$, and the rest of the data structure doesn't matter.

(a) Draw a possible picture of this data structure for $b = 4$ when $S_1 = 17$, $S_2 = 21, 22, 23, 24, 25$, and $S_3 = 31, 32, 33, 34, 35, 36, 37, 38, 39$.

(b) Consider the following operations, where $i \in \{1, 2, 3\}$ and v is an item.

POP(i): If S_i is nonempty, pop and return the top item from stack S_i. Otherwise, return ERROR.

PUSH(i, v): If the number of items in stacks S_1, S_2 and S_3 combined is less than N, push item v onto the top of stack S_i. Otherwise, return ERROR.

Explain how to perform any sequence of these operations on this data structure, starting from three empty stacks. The amortized time per operation must be $O(1)$. To do so, partially reorganize the data structure after every b operations. Make sure that you explain why the stated properties of the data structure hold after each operation is performed. It may be helpful to state some additional properties.

(c) Prove that the amortized time per operation for your implementation is $O(1)$, using the potential function $\Phi(D) = cr$, where r is the number of POP and PUSH operations performed since the last reorganization of the data structure and c is a constant that you choose.