CSC265 Fall 2020 Homework Assignment 6

Student 1: Jiawei Chen
Student 2: Yizhou Yang

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:

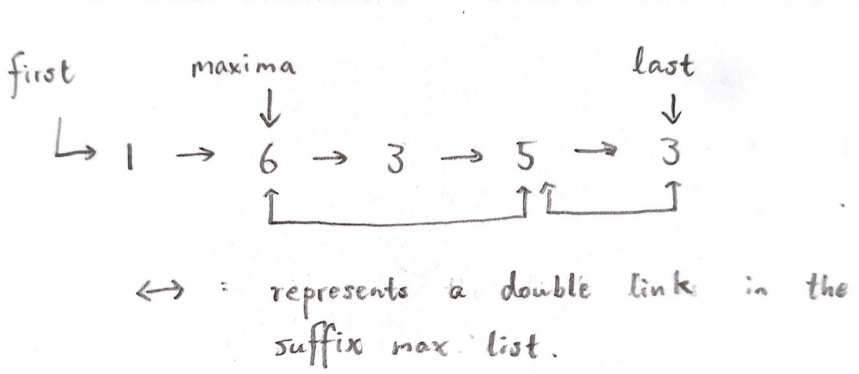
   - *num*, containing the integer value of the element;
   - *next*, containing a pointer to the record of the next element in the sequence (or NIL if it is the last element of the sequence);
   - *nextSuffixMax*, containing a pointer to the record of the next suffix-maximum in the sequence (or NIL if 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 $S = 1, 6, 3, 5, 3$.

   Solution:

   See figure 1.

   The NIL's at the end of the queue and in *nextSuffixMax*, *prevSuffixMax* of non-suffix-max nodes are omitted.



(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.
Solution:

DEQUEUE(S) First check the length of the queue. If it is empty, return NIL.
If it has only one number, return that number. Then, remove this node and set *maxima*, *first*, and *last* to NIL.
Otherwise, it has more than one element (that is, *first* and *last* point to different nodes). Use *first* to get the first node.

1. If it's not a suffix max, proceed to the next step.
2. If it happens to be a suffix max, then its clearly that it is the *maxima*. Remove it from the suffix-max list. And make the next node of the suffix-max list the new *maxima*. (Note that *nextSuffixMax* cannot be NIL because the last node is always a suffix max. And *first* and *last* point to different nodes because we have more than one element.)

Then, return its *num*. Remove it from the list. Set the next node to be the new *first* (In particular, if *next* is NIL, *first* becomes NIL as the queue is now empty).

ENQUEUE(S, x) If the queue is empty, let the new node be the *maxima*, *first*, and *last*.
If the queue is not empty, we first attach the new node to the end of the list. Make it the new *last*. Since it is the last number, it is a suffix-max for sure.
We need to verify whether the previous suffix maxima are still suffix maxima. So, starting from the right end, we go through the suffix-max list and compare every number to x until we find a number larger than x or the list is exhausted. Remove any number smaller than or equal to x from the suffix-max list. If the list is exhausted and no number is greater than x, make x the new *maxima* (it is also the only number in the suffix-max list as all other numbers are removed).

MAXIMUM(S) Simply return *maxima*.

DEQUEUE(S) is trivially correct in the empty case and one element case. In case that the first number is not suffix-max, simply removing it clearly causes no trouble. In case that it is a suffix-max, the algorithm handles it by making the next suffix max the new *maxima*. This is correct because the property suffix max only depends

on numbers on the right side. Removing the first (i.e. leftmost) number is not going to affect the suffix max status other numbers.
ENQUEUE(S, x) is trivially correct in the empty case. The algorithm also handles the change in the suffix-max list by iterating and checking. The *maxima* will be updated if the new number is greater than all other suffix maxima.
The MAXIMUM(S) algorithm is correct given that the other two functions maintain the record of *maxima* correctly.

To see that the amortized time is $O(1)$, first notice MAXIMUM(S) and DEQUEUE(S) only takes constant time. But for ENQUEUE(S, x), we need to loop through the suffix-max list. In the worst case, the suffix-max list can be as long as the entire queue, and all are less than the new number. So the worst case time complexity is $O(n)$. However, note that if ENQUEUE(S, x) checks k nodes, it implies that the $k-1$ nodes are smaller than x and thus will be removed from the suffix-max list. The length of the suffix-max list only increases if the new number is less than all other suffix max, in which case ENQUEUE(S, x) only checks 1 node. Let's call this special case of enqueue "$k = 1$ enqueue". The lenght of the suffix-max list is at most the number of times "$k = 1$ enqueue" is executed. So the amortized time complexity will not actually reach $O(n)$. I will explain this in detail in the next part.

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

(Let's say each loop in ENQUEUE(S, x) is one unit of cost. The cost of DEQUEUE(S) and MAXIMUM(S) is always constant, so it doesn't really matter what we assign as long as we assign a constant.)
The actuall cost of the operations are:
ENQUEUE(S, x): Number of the suffix max checked in the loop
DEQUEUE(S): 1
MAXIMUM(S): 1
Assign the allocated cost for each operation:
ENQUEUE(S, x): 2
DEQUEUE(S): 1
MAXIMUM(S): 1
DEQUEUE(S) and MAXIMUM(S) always have their actual cost equal to the allocated cost. The change in credits stored only come from ENQUEUE(S, x).
Note that if the loop in ENQUEUE(S, x) checks k numbers (i.e. its actual cost is k), that means $k-1$ numbers are removed from the suffix-max list, and then x is attached to the end of the suffix-max list (increasing its length by 1). So the length of the suffix-max list decreases by $k-2$ overall. In particular, when $k = 1$, the length increases by 1, when $k = 2$, the length is unchanged.
Each time, a $k = 1$ enqueue is executed. The actual cost is 1 and we have 1 credit stored. When a $k = 2$ enqueue is executed, it's actual cost is equal to the allocated cost 2, the stored credits neither increases nor decreases. When $k \geq 3$, it uses up $k-2$

stored credits. As we start with an empty queue (i.e. length of suffix-max list is 0 in the initial state), the credits stored correspond to the length of the suffix-max list. Since ENQUEUE(S, x)'s cost is at most the length of the suffix-max list (which cannot go to negative), the total credits stored cannot go to negative either.

To formulate this rigorously,

Lemma: Starting with an empty queue, and execute a sequence of n ENQUEUE commands. Then the credits stored is always at least the length of the suffix-max list.

Proof of lemma: Base case: When $n = 0$, no operation is executed. Credit stored is 0 and length is 0. The claim is trivially true.

Inductive step: Assume this is true for $n - 1$, verify n.

That is, the credits stored is at least the length of the suffix-max list after the $n - 1$-th enqueue.

Let's say the n-th enqueue has checks k numbers.

Then the credits stored increases by 2 (allocated cost). And then decreases by k (actual cost of running k loops). Overall, it decreases by $k - 2$.

For the change in length, there are two cases:

1. The loop stops because a number greater than x is found.
   Then the k-th number is larger than x. And the other $k - 1$ numbers are removed from the suffix-max list. Then, the x is added to the list. Overall, decreases by $k - 2$.
2. The loop checks all numbers in the suffix-max list and none is greater than x. Then, k numbers are removed, and x is added to the list. Overall, decreases by $k - 1$

In case 1, the length and credits decreases by the same number.

In case 2, the length decreases by one more.

So in both cases, the credits stored is at least the length of the suffix-max list.

By induction, the claim holds for all n.

□

There is another fact that is worth noting: DEQUEUE(S) can decrease the length of the suffix-max list if the first number happens to be a suffix max, but this has no contribution to the credits stored. So the credits stored is still at least the length of the suffix-max list. As stated before, the length of the suffix-max list cannot go to negative, therefore total credits stored (greater than the length) cannot go to negative either.

In conclusion, the total actual cost is bounded by total allocated cost. The total allocated cost of a sequence of n operations is at most $2n$. Dividing by n, we get a constant number 2. Therefore, the amortized time complexity is $O(1)$.

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.

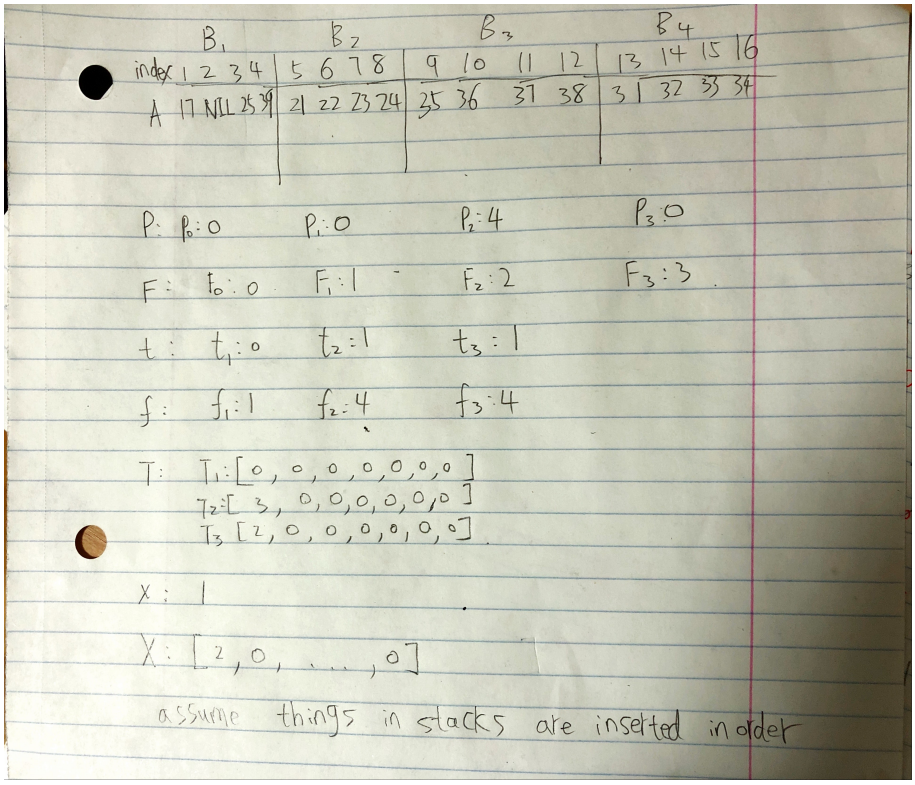
For $1 \leq i \leq 3$, the locations in A of the top t_i items in stack S_i are stored in the array $T_i[1..2b-1]$. If $t_i \neq 0$, then $A[T_i[t_i]]$ is the top item of stack S_i and, if $1 \leq j < t_i$, item $A[T_i[j]]$ is immediately below item $A[T_i[j+1]]$ in stack S_i.

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 $S_3 = 31, 32, 33, 34, 35, 36, 37, 38, 39$.

Solution:



(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.

Solution:

define SWAP(a,b) as swapping the values stored in indices a and b, this takes constant time, so we'll assume it takes 1 step.

PUSH(i,v): the general idea is that while there is still free block, do not occupy the blocks assigned to other stacks. There are 3 cases when making a PUSH.

If the stack is empty(F_i=0), then set F_i to F_0(occupy the first free block in the free list). Set F_0 to its parent, whose block number is recorded in P_{F_0}. Make $P_{F_i=0}$, since it is the bottom of the stack with no parent.

case 1: $f_i < b$

We want to extend the block by putting it in the F_ith block, however, we do not know

if the position is occupied or not. We seek $B_{F_i}[f_i+1]$, if its index in A is bigger than x, when we seek $X[X[F_i+f_i+1]]$. due to restructuring, if it does not equal to its index, then we know that the space is occupied. Add to A[X[1]], and swap X[1] with X[x]. Increase t_i by 1, and add this position to T[t_i]

Consider how the properties are maintained: Since X is an array of unoccupied residual places, this element will be added as a residual element, so as long as it is properly tracked by t and T, no property will be violated.

If it is not occupied, then add it to $B_{F_i}[f_i+1]$ and increase f_i by 1. Swap the X[added element's index] with X[x].

Consider how the properties are maintained: assume all other parts maintain all the properties, since the $B_k[f_i-1]$(the last element of inserted element) is right below the added element, and that this block is the point pointed to by F_i, the restraint is maintained.

case 2:$f_i = b$
If the current block is full, then first we try to find a new block. If F_0 is not 0, then set F_i to F_0(occupy the first free block in the free list). Set F_0 to its parent, whose block number is recorded in P_{F_0}. Make $P_{B_{F_0}}$ to be B_i, and swap the A[index of the inserted place]. with A[x].

If there is no more free blocks, then we add to occupied blocks. Now, the first x values in X indicate where we can add, add to A[X[1]], and swap X[1] with X[x]. Increase t_i by 1, and add this position to T[t_i]

In all cases, since we occupied a new space which was free, x need to decrease by 1.
Consider how the properties are maintained: We added a new block from a free block, so we pop out the top element from the free list, assigned it to the head of the linked list F_i, and adjusted its head so that this added properly keep track of its parent.

POP(i): we go into the following cases in order
case 1: $t_i > 0$
If there still are residual elements in stack i, then we pop them first.
The residual element at the top of the stack is $A[T[t_i]]$, pop it out, set $T[t_i] = 0, t_i = t_i - 1$.
Consider how the properties are maintained: we poped out an residual element and changed T and t, so after we fix X and x below, everything will be maintained.

case 2: case 1 doesn't apply, and $f_i > 1$
In this case, we just pop out $B_{F_i}[f_i]$, decrease f_i by 1.
Consider how the properties are maintained: We poped out an element from the top block of the assigned stack without freeing the block. We changed f to properly keep track of the top element, so after we fix X and x below, everything will be maintained.

case 3: case 2 doesn't apply, and $f_i = 1$
In this case, we need to also free the entire block. first pop out $B_{F_i}[1]$, and then assign B_{F_i} to F_0(set P_{F_0} to be F_i), freeing the block. Then, we reassign the top block of the stack to be the parent of the deleted stack, by assigning $F_i = P_{F_i}$. Change f_i to be b.
Consider how the properties are maintained: We freed a block. We need to assign it to the free list (by changing F_0), change the top block of the free list and F_i to be the freed block and its parent, and change f_i to indicate that the new top block is full.
In all cases of pop, we manipulate X. Denote the index that we poped out to be j. If X[j]!= j swap X[X[j]] with X[x+1], else swap X[j] with X[x+1].
In all cases, since we released a new space, x need to increase by 1.

We need to restructure X every b operations.

Invariant: the first x elements are free, and all elements after that are not free since they do not repeat. Each free element either stores its index, or some number that is bigger than x due to the way I switched in push and pop.

RESTRUCTURE(): Iterate b elements of X starting from x. For the i-th element, if it stores k$\neq$ i and k<x, then set X[k] = i. In this way, if X[m]!=m for any m smaller than x, then X[X[m]]= m.

(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.
Solution:
let c equal to 1, then the potential function will denote the maximum pairs of elements in X with index i that stores values k such that k not equal to i and X[k] does not equal to i (misplaced) since the last restructure.

Consider a push or pop. The operations themselves takes constant steps, but each operation can also create a maximum of 2 misplaced elements. The amortized cost is
$c_i' = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$
$= 1 + (r_i - (r_i - 1))$
$= 2$

now we prove the RESTRUCTURE also takes amortized constant steps.
Every RESTRUCTURE goes through b iterations, but for each iteration they the misplaced elements.
It happens after every b times of push/pop.

$c_i' = c_i + (\Phi(D_i) - \Phi(D_{i-b}))$
$= b + (r_i - (r_i - b))$
$= 0$

Since push, pop and restructure all takes amortized O(1) time to perform, the amortized time per operation is O(1).