CSC265 Fall 2020 Homework Assignment 6

Student 1: Kunal Chawla
Student 2: Andrew Feng

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:

   (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:

   MAXIMUM**:** One can simply return the element pointed to by *maxima*. This is greater than all subsequent terms by the definition of a suffmix maximum, and is greater than all previous terms, else another element would be the first suffix maximum.

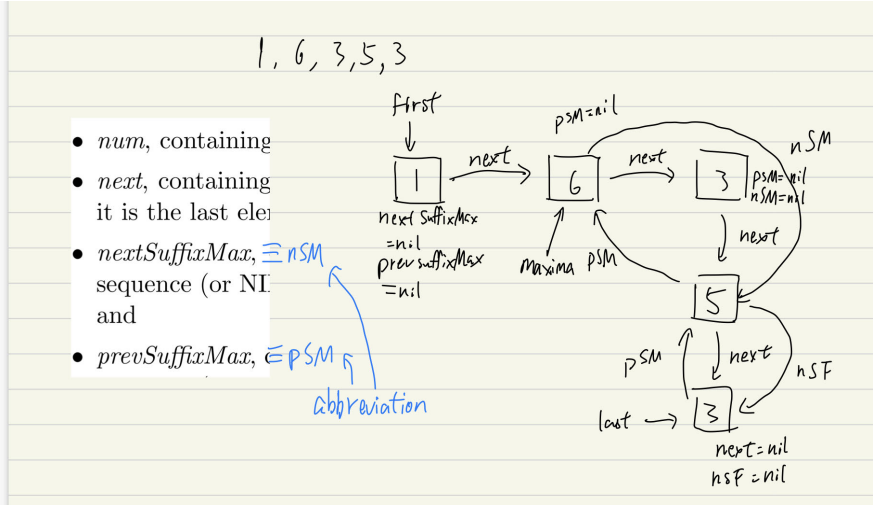


Figure 1: MAXQUEUE. All pointers pointing to NIL are the corresponding 'prevSuffixMax' and 'nextSuffixMax' for the nodes, unlabelled as to not create more clutter in the diagram.

This takes constant time.

DEQUEUE**:** Recall from page 238 of CLRS we have the routine LIST-DELETE(L, x) which, given a doubly linked list and a pointer x to an element of the list, deletes it in constant time. For the sake of clarity we'll call this 'DOUBLY-LINKED-DELETE(L, x)' to show that it is for doubly linked lists.

Now we also have a way to delete the first item of a singly linked list. Indeed, we a constant time routine SINGLY-LINKED-DELETE-FIRST(L) as follows:

SINGLY-LINKED-DELETE-FIRST(L)

1 L.head = L.head.next

which clearly runs in constant time as well.

Therefore we can implement DEQUEUE(L) as follows

i. If the element pointed to by *first* is equal to the element pointed to by *maxima*, delete the first item of the doubly-linked list using DOUBLY-LINKED-DELETE(L, $L.head$) where L is the list containing the suffix maxima

ii. Delete the first item of the singly-linked-list using SINGLY-LINKED-DELETE-FIRST(L)

both steps run in constant time, therefore this runs in constant time in the worst case. Now to see why the algorithm is correct.

Clearly it removes the first item of the linked list, so we need to check that it preserves the doubly linked list of suffix maxima.

There are two cases: either (i) the first element was not a suffix maximum, or (ii) it was.

In the first case, the doubly linked list of suffix maxima is not changed, and the first suffix maxima was greater than or equal to all previous elements. Therefore the doubly linked list of suffix maxima is correctly maintained.

In the second case, the first element of our list is the first suffix maximum. Say that the second suffix maximum is the ith element of our sequence. Then after DEQUEUE(L, x) is run, the first element of the list of suffix maxima is now the $(i-1)$th element of the list, which is at least as large as the first $i-2$ elements (otherwise this would not be the maxima). Therefore the doubly linked list of suffix maxima is correctly maintained.

ENQUEUE**:** We can perform ENQUEUE(L, x) as follows:

i. Append x to the end of the singly linked list.
ii. Append x to the end of the doubly linked list. Call the corresponding element of the sequence y, therefore y.num $= x$.
iii. While y.num is greater than y.prevSuffixMax, delete y.prevSuffixMax using DOUBLY-LINKED-DELETE(L, y.prevSuffixMax)

The first two lines take constant time, and so does the comparison used while continuing the while loop.

Each iteration of the while loop takes constant time as well, therefore the time taken is linear in the number of iterations of the while loop.

We can also see why this is correct, it correctly appends x to the end of the list, so we need to check that it properly maintains the list of suffix maxima.

Observe that y is the last node in the linked list, therefore y clearly contains a suffix maxima, so all that's left to check is that y.prevSuffixMax.num is greater than y.num.

However, this is precisely the halting condition for the while loop, which clearly halts as the number of suffix maxima strictly decreases at each iteration.

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

Solution:

Observe that as the running time of any operations is linear in the number of times an element is inserted or deleted from a doubly linked list, deleted from the beginning of a singly linked list, or when MAXIMUM is called, so we can count the running time of a sequence in terms of these.

Therefore we claim that each of the operations DEQUEUE, ENQUEUE, and MAXIMUM take $\mathcal{O}(1)$ amortized running time.

We assign the following amortized costs to each operation:

- ENQUEUE: 3
- DEQUEUE: 1
- MAXIMUM: 1

We now justify why we can pay for any sequence of operations using only the amortized costs.

Starting from an empty, list, whenever we add an element to our list of suffix maxima, we place a 1 dollar bill on that node.

We claim that the our credit after any sequence of n operations is nonnegative, which we prove by induction on n.

In our base case $n = 0$ both our actual and amortized costs are 0. Now assume the statement is true for some n, we show that it is true for $n + 1$.

Either the $(n+1)$th operation is a call to (i) MAXIMUM, (ii) ENQUEUE, or (iii) DEQUEUE.

In the first case, we do not modify the list, so our actual cost is 1 and so is our amortized cost. Therefore our credit invariant is maintained.

In the second case, we use 1 dollar to pay for appending our new value v to the end of our linked list, another dollar to append it to our linked list of suffix maxima, and we place a dollar on top of it. Therefore the only operations left are removing all necessary elements from our doubly linked list as specified in part (b).

As we placed a dollar on each element of the doubly linked list, and each deletion costs 1 dollar, we have enough credit to pay for the operation, therefore our credit invariant is maintained

In the third case, either our first element is a suffix maxima, or it is not. If it is a suffix maxima, we need to delete it from our doubly linked list which costs 1 dollar. We can simply use the dollar already stored on top of the element to pay for this. In either case we delete the first element from our singly linked list, which costs the single dollar we paid for when calling DEQUEUE. Therefore our credit invariant is maintained.

Overall, this shows that at each step in a sequence of operations, our credit is non-negative. Therefore our amortized cost is an upper bound for our worst case cost in a sequence of operations. Since our allocated costs are all constant (3,1, and 1), we know the amortized cost is $O(3n) = O(n)$, meaning that the amortized complexity is $O(n/n) = O(1)$.

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 $1 \le i \le 3$, 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 $t_1 + t_2 + t_3 \le 4b - 3$. The values t_1, t_2, and t_3 are stored explicitly in the data structure.

If $1 \le i \le 3$ 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.

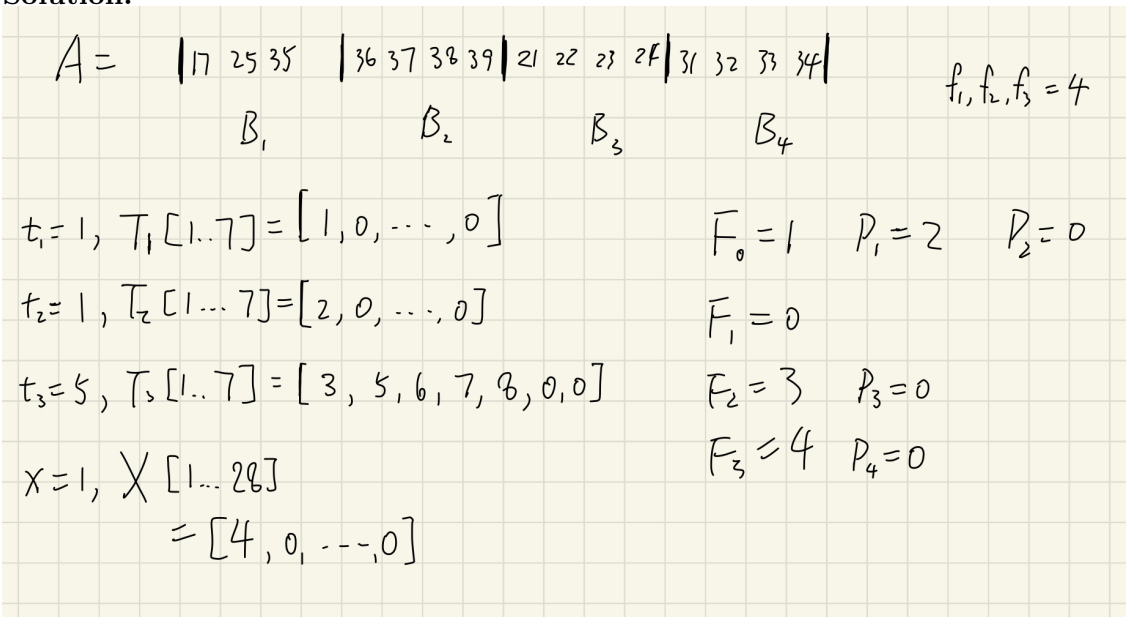
For $1 \le i \le 3$, the locations in A of the top t_i items in stack S_i are stored in the array $T_i[1..2b-1]$. If $t_i \neq 0$, then $A[T_i[t_i]]$ is the top item of stack S_i and, if $1 \le j < t_i$, item $A[T_i[j]]$ is immediately below item $A[T_i[j+1]]$ in stack S_i.

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 $1 \le i < b$, $P_b = 0$, $F_0 = 1$, $F_1 = F_2 = F_3 = 0$, $t_1 = t_2 = t_3 = 0$, $x = 4b$, $X[i] = i$ 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:

For convenience, the first 4 free blocks (or, if the free list is of length less than 4, the entire free list) are called the residual blocks. We will have two kinds of rearrangements: REARRANGEMENT-A is called automatically after every b operations—its purpose is to allocate new blocks; REARRANGEMENT-B(i) is called within POP to deallocate blocks if needed—we will prove that it will be called at most three times once every b operations.

Before we describe the algorithm, we state some helpful properties that our data structure should satisfy.

i. For $1 \leq i \leq 3$. Every block in the linked list starting at B_{F_i} is filled (a slot is filled when what is stored in that slot is an element of a stack and a block is filled when all its slots are filled). An exception is B_{F_i}: we require that $B_{F_i}[1..f_i]$ is filled, but $B_{F_i}[f_i..b]$ might not be. This property is maintained by our algorithm.

ii. $x = 0$ exactly when A is filled. Obviously, when A is filled, $B_{F_i}[f_i + 1..b]$ is filled $(1 \leq i \leq 3)$ and the residual blocks are filled. This means $x = 0$ by definition of x. Conversely, when $x = 0$, this means, again by definition of x, that $B_{F_i}[f_i + 1..b]$ is filled $(1 \leq i \leq 3)$ and the residual blocks are filled. Notice that by the above property, all blocks in the linked lists starting at $B_{F_1}, B_{F_2}, B_{F_3}$ are filled (including $B_{F_1}, B_{F_2}, B_{F_3}$ since $x = 0$). Now, consider the remaining blocks. We know that they must be free blocks. We claim that there are less than 4 free blocks. Indeed, if there are at least 4 free blocks, then the first four of which will be the residual blocks. So $x \geq$ (the number of unfilled slots in the residual blocks). But we know there can be at most $4b - 3$ residual elements, leaving at least 3 unfilled slots. By the above inequality, we should have $x \geq 3$, a contradiction. Thus, there are less than 4 free blocks. So all of them will be residual blocks, and $x = 0$ simply implies all of them are filled. We have show all blocks are filled. Thus A is filled. This property follows directly from the data structure specification, and our algorithm will maintain it.

iii. After every REARRANGEMENT-A, each stack will have at most $b - 1$ residual elements.

iv. A stack S_i is nonempty iff $f_i > 0$ or $t_i > 0$. This will be maintained by POP.

v. A stack S_i has $B_{F_i} \neq 0$ iff $f_i \neq 0$ (in our algorithm, we initialize $f_i = 0$). Clearly if $B_{F_i} = 0$ then $f_i = 0$. For the other direction, our algorithm will maintain that if $f_i = 1$ and POP(i) is called, then a block is deallocated from S_i, therefore either S_i is empty or $B_{F_i} \neq 0$ and now $f_i = b$.

Now, we describe our algorithms for PUSH, POP, REARRANGE-A, and REARRANGE-B, and their correctness. We also briefly discuss the worst case running time of each.

- PUSH(i, v). We return ERROR if $x = 0$, this is justified by the invariant that $x = 0$ iff A is filled. Now, we increment t_i. We know, by property [2(b)iii] and the fact that

REARRANGE-A is called every b operations, that $t_i < 2b - 1$. Indeed, there could have been at most $b - 1$ operations called since the last rearrangement, therefore at this point $t_i \leq b - 1 + b - 1 = 2b - 2$. Thus incrementing t_i does not result in an invalid $t_i > 2b - 1$. Knowing this, we set $T_i[t_i] = X[x]$, decrement x, and set $A[T_i[t_i]] = v$. Notice that every operation in PUSH runs in $O(1)$ time.
Notice that PUSH maintains the fact that x counts the number of unused residual slots, thus property [2(b)ii] is maintained.

- POP(i). We return ERROR if $f_i = 0$ and $t_i = 0$. This is justified by property [2(b)iv]. One case is that $t_i = 0$. This means there are no residual elements and $f_i > 0$. We simply pop from the block B_{F_i}:

  1 *return-val* $= B_{F_i}[f_i]$
  2 $X[{+}{+}x] = b(F_i - 1) + f_i$
  3 f_i--

  [1] Otherwise, we must have $t_i \neq 0$, in which case we pop off the residual element $T_i[t_i]$:

  1 *return-val* $= T_i[t_i]$
  2 $X[{+}{+}x] = T_i[t_i]$
  3 t_i--

  However, since we are decreasing f_i or t_i, it is possible that we violate property [2(b)iv]. So, if $f_i = 0$, $t_i = 0$, but $P_{F_i} \neq 0$ for some $i \in \{1, 2, 3\}$, we call REARRANGE-B(i) to fix this:

  1 **for** $i = 1, 2, 3$, **if** $f_i = 0, t_i = 0, P_{F_i} \neq 0$
  2 REARRANGE-B(i)
  3 **return** *return-val*

  As we will see, REARRANGE-B has worst case running time $O(b)$. And since everything else in POP is constant time, we know that POP has worst case time complexity $O(b)$.

- REARRANGE-A. Without some rearrangement, we might run out of residual slots prematurely, violating property [2(b)ii]. Or, we might exceed the limit $\sum_{i=1}^{3} t_i \leq 4b - 3$. We solve this by calling REARRANGE-A after every b operations.
  We will now describe REARRANGE-A.
  Let $i = 1, 2, 3$ (this notation corresponds to a for loop inside REARRANGE-A). REARRANGE-A first fills up B_{F_i} with residuals in S_i. This is done by relocating all residuals that lie in $B_{F_i}[f_i + 1..b]$ to elsewhere and move residuals at indices $T_i[1..b - f_i]$ to $B_{F_i}[f_i + 1..b]$. More specifically, we need to loop through $j = 1, 2, 3$ and $k = 1, \ldots, t_j$: if the index $T_j[k]$ lies in $B_{F_i}[f_i + 1..b]$, we perform swaps

  $$T_j[k] \longleftrightarrow T_i[(T_j[k] \bmod b) - f_i] \text{ and } A[T_j[k]] \longleftrightarrow A[T_i[(T_j[k] \bmod b) - f_i]].$$

  This looks ugly but the idea is simple: say $T_j[k]$ refers to the lth element in B_{F_i}, we want the residual element at index $T_i[l - f_i]$ to be there instead. And to find out l we need to take remainder mod b. Notice that we iterate through at most $4b - 3$

[1] We use the convenient notation where ${+}{+}x$ means incrementing x first and then evaluating the code. Placing ${+}{+}$ after x means evaluating and then increment. Similar for --.

elements because $\sum_{j=1}^{3} t_j \leq 4b-3$. And swapping takes constant time. Thus, this part of our algorithm takes $\Theta(b)$.

By now, we would have relocated all residual elements in B_{F_i} to appropriate positions, but slots with no residual elements remain "empty". Thus we will loop through $k = 1, \ldots, b - f_i$ to seek those that still have not been moved to B_{F_i}. In particular, if $T_i[k]$ does not lie in B_{F_i}, we will set $B_{F_i}[k] = A[T_i[k]]$.

Finally, we can set $f_i = b$, $t_i = t_i - (b - f_i)$, and $T_i[1..b - f_i]$ all to 0. We will also need to apply FIXUP(T_i)[2].

We shall denote the aforementioned part of our procedure as FILL$(B_{F_i}[f_i{+}1..b], L_i[1..b{-}f_i])$. We will reuse FILL again shortly.

The problem is that after all steps described above, it is still possible that $t_i \geq b$. The first case here is that LENGTH$(F_0) > 4$, in which case we simply perform FILL$(B_s, L_i[1..b])$, where B_s is the last free block, and then update $P_s = F_i$,$F_i = s$. (Here, LENGTH(F_i) gives the length of the singly linked list of blocks starting at F_i, $i \in \{0, \ldots, 3\}$. This is $O(b)$ since we simply traverse the entire linked list, which has length at most b.). The second case is LENGTH$(F_0) \leq 4$. Now, if $B_{F_0} \neq 0$, then we perform FILL$(B_{F_0}, L_i[1..b])$. However, it is possible to have $B_{F_0} = 0$. When this happens, we know that there are no more free blocks, and so all residual elements are stored in $B_{F_1}[f_1{+}1..b], B_{F_2}[f_2{+}1..b], B_{F_3}[f_3{+}1..b]$ (this is assuming that $F_1, F_2, F_3 \neq 0$ but the case where at least one of these is 0 can be handled exactly the same way). To solve the issue of $t_i \geq b$, we first perform FILL$(B_{F_i}[F_i{+}1..b], T_i[1..\max\{b{-}f_i, t_i\}])$. Then, if this still doesn't solve the problem $t_i \geq b$, we let $j \neq i$ and "make B_{F_j} the new B_{F_i}" by doing the following. For every $k = 1, \ldots, f_j$, we increment t_j, set $T_j[t_j] = T_i[k]$, and swap $A[T_j[t_j]] \longleftrightarrow B_{F_j}[k]$. This turns the first f_j elements into residuals of stack S_j, and swap their positions with those in S_i. Finally, we perform FILL$(B_{F_j}[f_j + 1..b], T_i[f_j + 1..b])$. Having done all of this, we are left with a B_{F_j} containing only elements of S_i, in the same order they appear in T_i. To clean up, we should correctly update $F_i = F_j$, $F_j = P_{F_j}$, $f_j = b$, and FIXUP(T_i).

Assuming that $t_i \leq 2b - 1$, the above procedure ensures now that $t_i \leq b - 1$. Also notice that since this algorithm runs in worst case $O(n)$ since we are traversing lists of length $O(n)$ and performing constant operations within each iteration.

- REARRANGE-B. As mentioned in the specification of POP(i), it is possible that $f_i = 0, t_i = 0, P_{F_i} \neq 0$. In this case, we would like the deallocate the first block B_{F_i}. Notice that it is not enough for us to simply add the block B_{F_i} to the end of the free list and reset $B_{F_i} = P_{F_i}$, because if the free list is already of length 4, this will introduce a free block containing residuals but which is not a residual block.

  Now, if LENGTH$(F_0) < 4$, we can of course skip this and do exactly what is outlined above, since the newly added block would be in the first four blocks, making it a residual block. So suppose LENGTH$(F_0) \geq 4$. We do the following. Set counter $k' = x$, and for each $i \neq j \in \{1, 2, 3\}$, we loop through $k = 1, \ldots, t_j$. If $T_j[k]$ lies in B_{F_i}, then:

[2] FIXUP(L) is a linear time procedure that removes all zeros in an array L. It works by keeping counters c and f (both 0 initially), where c stores index of the next 0 in L and f gets incremented until $L[f]$ is nonzero. Then the value at index f is moved to index c, $L[f]$ is set to 0, and c gets incremented until $L[c] = 0$. The procedure terminates in $O(length(L))$ steps after either c or f reaches the end of L. In particular it runs in $O(b)$ steps.

- decrease k' until the index $X[k']$ does not lie in B_{F_i}, and
- set $X[{+}{+}x] = T_j[k], T_j[k] = X[k'], A[T_j[k]] = A[X[x]], X[k'] = 0$ (this moves what was stored at index $T_j[k]$ to outside of B_{F_i}).

After this process, all of B_{F_i} should be cleared and it should be ready to be added to the end of the free list. To clean up, we perform FIXUP(X). Set $F_i = P_{F_i}$, $f_i = b$.

(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: Recall from the previous part of our questions the worst case costs for each operation

- PUSH(i, v) takes constant time.
- REARRANGE-A(i) takes $O(b)$ time.
- POP(i) takes constant time if REARRANGE-B is not called, and $O(b)$ time if it is.

Also observe that our potential function is nonnegative and our initial potential is equal to 0. Therefore after a sequence of n operations, our potential difference $\Phi(D_{n-1}) - \Phi(D_0)$ is nonnegative, therefore the amortized costs we calculate below are valid, in the sense that the sum of amortized cost for a sequence gives an upper bound for our worst case performance.

As REARRANGE-A(i) takes linear time in b, there exists some constants $M, N > 0$ such that the procedure takes less than $Mb + N$ steps. So we let $c = M$.

We calculate:

- PUSH(i, v): Our amortized cost is

$$O(1) + c \cdot (r) - c \cdot (r-1) = O(1) + c \in O(1).$$

- We also have the following amortized cost for REARRANGE-A(i):

$$Mb + N + c \cdot 0 - c \cdot b \in O(1).$$

- For POP(i), when REARRANGE-B is not called, our amortized time is

$$O(1) + c \cdot r - c \cdot (r-1) \in O(1).$$

- When REARRANGE-B is called, it takes linear time in b, however observe that this can occur at most 3 times in a sequence of b operations. Indeed, for $1 \leq i \leq 3$, the stack i can have at most one block deallocated as a result of POP(i) operations. Therefore this case occurs $O(n/b)$ times taking $O(b)$ steps each time. So this case occurs with $O(1)$ amortized complexity.