Comp 301 - Project 3 Report

Aslı Koçer- 63999
Mehmet Üstek - 64782
Yasemin Melek - 63929

Workload Distribution

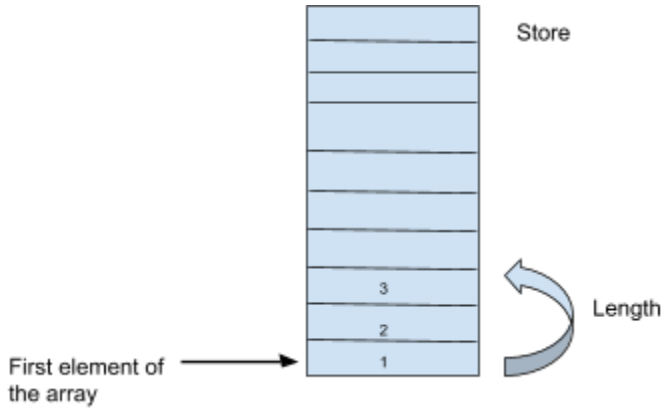
We worked together while discussing the implementation and collaborated for challenging parts of the code. The workload distribution for the coding part is as follows.
Aslı: Queues
Mehmet: Arrays, queues
Yasemin: Stacks

Parts that work properly, and that do not work properly

Every part of our code works properly.

Our Approach to Implementations

Array:
Our array works as a list of elements with the first element is the pointer to the rest of the list. It is the almost equivalent of LinkedList structure in a sense. In this way, we do not have to keep every element's position, instead we know the first element's position in the store and the length. Using this length and first element, we are eligible to obtain the rest of the list.
For further understanding, please analyze the visual 1.



Visual 1: Array Structure in the Store.
In this visual, the given array contains 3 elements which are 1,2,3. While putting these elements or retrieving these elements, we do not know their exact location. We only know the position of the first element on the array.

In this way, when I call read-array with a given index, I will iterate through my list which I obtain from the first element's address.

Queue:
Our queue works as a list of elements that is pushed through the list. We add new elements to the rear of the list and push from front. In our queue structure, we check our queue's size by counting initialized variables. For example, when we have a queue such as (10 20 30 #f #f #f ….), we increase the count by one for every index until we meet a #f value.

When we create a new queue, we formed an array that has 1000 not initialized elements (every element is #f in this case). Thus, we create enough space to have 1000 push operations.

In the push method, we add the element to the end of the queue by this method;
(update-array queue (expval->num (queue-size queue)) value)

To pop the first element of the queue, we return the value stored in the first index of the array. Then all the remaining elements of the queue have to be shifted leftward so that the front index of the queue is not empty.
When we shift the array to left by one regarding queue-size - 1, the number of the end stays still, so we update that last element to be a dummy value at the end.
i.e assume we have queue of (10 20 30 #f #f …) we shift all the elements to the left by one regarding queue-size - 1. So now, we have (20 30 30 #f #f), we have one extra element left in the 2nd index.
To get rid of the last element left we apply this code:
(update-array queue (- expval->num (queue-size queue) 1) (bool-val #f))).
Now with the pop operation completed, we have;
(20 30 #f #f #f …)
We do this shift of queue by **shift_left** method.

```
(define shift_left
   (lambda (array index )
     (if (equal? index (- (expval->num (queue-size array)) 1))
        (display "")
        (begin
          (update-array array index (read-array array (+ index 1)))
          (shift_left array (+ index 1))))))
```

shift_left method takes an array and an iteration variable index. It iterates every element to the left by one except for the last element to preserve list bounds.

To find the size of the stack, we find the location of the element at the end of the queue which is stored in the last (rightmost) non-empty index of the array and estimate the difference of it with the first index of the array.

Stack:
Our stack also works as a list of elements that is pushed through the list. As we push, we add new elements to the top of the stack by inserting the new element to the end of the list.

To create a new stack, we formed an array that has 1000 not initialized elements (every element is #f in this case) since the maximum push operations was stated as 1000 in hw constraints.

To pop the topmost element of the stack, we return the element located in the index that is equal to the stack-size minus 1. To remove the element, we update the array by assigning a #f boolean value to that index.
To find the size of the stack, we followed the same method we used in the queue.