\*2.1.

My choice of data structure for this question is an array because the space for the collection of books will neither expand nor shrink. And methods like swap is much more faster when an array is used for this question. Transversing through a list for some (implemented) methods would be much slower than getting a object from an array. This is because book’s unique ID is equal to array’s index.

Run-time:

checkout-> O(1)

checkin-> O(1)

status-> O(1)

details-> O(1)

removeOld-> O(1)

addNew-> O(array.length)

swap-> O(1)

booksOwned-> O(1)

inStock-> O(1)

OutForRent-> O(1)

As the library computers are not fast these functions are good match, where, only one function (addNew) has run-time of O(n) and all the other functions has run-time of O(1).

\*2.2.

Function:

enqueue- My enqueue function pops all the elements from Calendar stack and pushes them to Diary stack one-by-one. Then it pushes the element need to be queued in Calendar stack. Lastly, it pops all the elements from Dairy stack and pushes them back to Calendar stack one-by-one. That’s how finally the newest element (entry) ends up at the bottom of the Calendar stack.

Run-time-> O(size)

dequeue- My dequeue function pops an element from Calendar stack and pushes it to Dairy stack. That’s how the oldest element (entry) of Calendar stack ends up at to top Dairy stack.

Run-time-> O(1)

The run-time of enqueue function changed compared to the basic enqueue function of Queue’s ADT (O(1)) because a stack’s ADT only allows to add/remove elements at the top and in this case the newest item must be at the bottom to Calendar stack. Therefore, the elements needed to be moved from one stack to the other in order to enqueue a new element.

\*2.3.

Run-time:

iterative-> O(size)

recursive-> O(size\*log(size))

As the run-time of the iterative method of reversing a list of strings is smaller than the recursive method, iterative method is the better option for reversing a linked list of strings. On the other hand, the recursive method creates a new singly-linked list to pass a new list as list.head = list.head.next as an argument on every recursion. This is an expensive operation, although the memory- space is cleared as soon as the function ends. Overall, the iterative method for reversing a linked list is the better choice in this scenario.

\*2.4.

Run-time:

size-> O(1)

addItem-> O(1)

addItemPriority-> O(1)

removeItem-> O(1)

All of my function’s run-time is the same as it would be for a queue or a deque’s usual function under it’s ADT. My addItem function uses tail pointer to add item at the back of a list. And my addItemPriority and removeItem function uses head pointer to add and remove item at the front of a list.