Alexander Shah
Homework 5: More on Lists
EN.605.202.81 Section 84

1.

//If we consider items contain data, and pointers to previous and next, we can implement a deque as follows.

```
Define left, right = deque.head, deque.tail

Method InsertLeft(item):
   //an empty left item means list is empty, left and right are the same
   if left == null:
      left = item
      right = item
   else:
      //otherwise insert item and move pointers
      item.next = left
      left.prev = item
      left = item

Method DeleteRight(item):
   //remove if found
   if right != null:
      //single element
      if right == left:
            right = null
            left = null
      else:
         //adjust pointers to remove item
         right = right.prev
         right.next = null
```

2.

//Implementing a deque as a doubly-linked circular list with a dummy header

```
// header – items – tail
Method InsertRight(item):
   // wrap around the list by pointing item's previous to header's previous
   item.prev = header.prev
   item.next = header
   // adjust header
   header.prev.next = item
   header.prev = item

Method DeleteLeft(item):
   //if not empty
   if header.next != header:
      //rearrange pointers to remove the item
      header.next = header.next.next
      header.next.prev = header
```

3.

//In order to implement a hybrid structure to handle multiple stacks and queues in a single array, we need an array of a sufficient size to hold the elements in all structures, a stack to keep track of free spaces, and methods to handle assignment and cleanup of space usage. Then we need to keep track of head and tail pointers for our structures and define methods for adding and removing elements from them.

```
Define size as available array space
Define array as List[size]
Define free_list as Stack

// Initialize available space and the free list
for i from 0 to size - 1:
   array[i].data = null
   array[i].next = i + 1
   free_list.push(i)

AllocateSpace():
   //Return the next available space
   if not free_list.is_empty():
      return free_list.pop()
   else:
      return -1

DeallocateSpace(index):
   //Reset data and pointer
   array[index].data = null
   array[index].next = -1

   //Put the index on the free list
   free_list.push(index)

Insert(item):
   index = AllocateSpace()

   if index != -1:
      array[index].data = item.data
      array[index].next = item.next

Delete(index):
   if index >= 0 and index < size:
      DeallocateSpace(index)

//Define data structures for multiple stacks and queues
Define num as the number of stacks and queues
Define stacks as List[num]
Define queues as List[num]
```

```
//Initialize stack and queue pointers
for i from 0 to num-1:
    stacks[i].head = -1
    queues[i].head = i * (size // num)
    queues[i].tail = i * (size // num)

Push(stack_num, item):
    index = AllocateSpace()
    if index != -1:
        // Get the prev item's index
        prev_index = stacks[stack_num].head

        // Update the new item's next pointer to the previous item
        array[index].next = prev_index

        // Update the head to the new item
        stacks[stack_num].head = index

        array[index].data = item.data

Pop(stack_num):
    head = stacks[stack_num].head
    if head != -1:
        data = array[head].data
        index = array[head].next

        // Update the head to the next item
        stacks[stack_num].head = index

        DeallocateSpace(head)
        return data

Enqueue(queue_num, item):
    index = AllocateSpace()
    if index != -1:
        last_index = queues[queue_num].tail

        // Update the last item's next pointer to the new item
        array[last_index].next = index

        // Update the tail
        queues[queue_num].tail = index

        array[index].data = item.data
        array[index].next = -1

Dequeue(queue_num):
    head = queues[queue_num].head
    if head < queues[queue_num].tail:
```

```
data = array[head].data
index = array[head].next
queues[queue_num].head = index
DeallocateSpace(head)
return data
```