Ron Cox

HW5
605.202.81
Data Structures

1. Deque as a Doubly-Linked List

```
Class Node:
      data  // The data stored in this node
      next  // Pointer to the next node in the deque
      prev  // Pointer to the previous node in the deque

Class Deque:
      head  // Pointer to the first node in the deque
      tail  // Pointer to the last node in the deque

      Method __init__():
      head = null  // Initialize head to null
      tail = null  // Initialize tail to null

      Method InsertLeft(item):
      newNode = new Node(item)  // Create a new node with the given item
      if head == null:
            head = newNode  // Set both head and tail to the new node if deque is empty
            tail = newNode
      else:
            newNode.next = head  // Insert new node at the left end
            head.prev = newNode
            head = newNode

      Method DeleteRight():
      if tail == null:
            return null  // Return null if deque is empty
      deletedItem = tail.data  // Store the data to be deleted
      if head == tail:
            head = null  // If there's only one element, set head and tail to null
            tail = null
      else:
            tail = tail.prev  // Update the tail pointer
            tail.next = null
      return deletedItem
```

2. Deque as a Doubly-Linked Circular List with a Header

```
Class Node:
      data  // The data stored in this node
      next  // Pointer to the next node in the deque
      prev  // Pointer to the previous node in the deque

Class Deque:
      header  // Header node of the circular list

      Method __init__():
      header = new Node(null)  // Initialize the header node
      header.next = header  // Point header.next to itself
      header.prev = header  // Point header.prev to itself

      Method InsertRight(item):
      newNode = new Node(item)  // Create a new node with the given item
      last = header.prev  // The current last node
      last.next = newNode  // Update pointers to insert new node at the right end
      newNode.prev = last
      newNode.next = header
      header.prev = newNode

      Method DeleteLeft():
      if header.next == header:
            return null  // Return null if deque is empty
      first = header.next  // The current first node
      deletedItem = first.data  // Store the data to be deleted
      header.next = first.next  // Update pointers to remove the first node
      first.next.prev = header
      return deletedItem
```

3. Implementing Several Stacks and Queues within a Single Array

```
Class HybridArray:
      array  // The underlying array
      stackTops  // Array to keep track of the top indices of stacks
      queueFronts  // Array to keep track of the front indices of queues
      queueRears  // Array to keep track of the rear indices of queues
      size  // Total size of the underlying array

      Method __init__(totalSize, numStacks, numQueues):
      array = new Array[totalSize]  // Initialize the array
      stackTops = new Array[numStacks]  // Initialize stackTops
      queueFronts = new Array[numQueues]  // Initialize queueFronts
      queueRears = new Array[numQueues]  // Initialize queueRears
      size = totalSize  // Set the total size
      for i from 0 to numStacks-1:
            stackTops[i] = -1  // Initialize stack tops to -1
      for j from 0 to numQueues-1:
            queueFronts[j] = -1  // Initialize queue fronts to -1
            queueRears[j] = -1  // Initialize queue rears to -1

      Method Push(stackNum, item):
      stackTops[stackNum] += 1  // Increment the top index of the stack
      array[stackTops[stackNum]] = item  // Add the item to the array

      Method Pop(stackNum):
      if stackTops[stackNum] == -1:
            return "Stack is empty"
      item = array[stackTops[stackNum]]  // Retrieve the item from the
stack
      stackTops[stackNum] -= 1  // Decrement the top index of the stack
      return item

      Method Enqueue(queueNum, item):
      if queueFronts[queueNum] == -1:
            queueFronts[queueNum] = findFirstFreeIndex()  // Set the front
if queue is empty
      queueRears[queueNum] = (queueRears[queueNum] + 1) % size  // Update
the rear index
      array[queueRears[queueNum]] = item  // Add the item to the array

      Method Dequeue(queueNum):
      if queueFronts[queueNum] == -1:
            return "Queue is empty"
      item = array[queueFronts[queueNum]]  // Retrieve the item from the
queue
      if queueFronts[queueNum] == queueRears[queueNum]:
            queueFronts[queueNum] = -1  // Reset front and rear if the
queue is now empty
```

```
            queueRears[queueNum] = -1
      else:
            queueFronts[queueNum] = (queueFronts[queueNum] + 1) % size  //
Update the front index
      return item

      Method findFirstFreeIndex():
      for i from 0 to size-1:
            if array[i] == null:
                  return i  // Return the first free index
      return -1
```