Name: Pheakdey Luk
ID: 986591

Assignment 2

R-2.1 Describe, using pseudo-code, implementations of the methods insertBefore(p,e), insertFirst(e), and insertLast(e) of the List ADT, assuming the list is implemented using a doubly-linked list.

```
Algorithm insertBefore(p,e)
    Input : position of node p where  newnode will be inserted before this
node and element of newNode
    Output: newNode with element e will be inserted into list.

    newNode<-createNewNode(e)
    tmp<-p.prev
    tmp.next <- newNode
    newNode.next <- p
    newNode.prev <- tmp

    p.prev <- newNode



Algorithm insertFirst(e)
    Input : element e,which will be inserted into first of the linked list
    Output: newNode with element e will be inserted into first position of
list.

    newNode<-createNewNode(e)
    tmp<-head.next
    head.next<-newNode
    newNode.next<-tmp
    tmp.prev <- newNode

    newNode.prev <- head



Algorithm insertLast(e)
    Input : element e,which will be inserted into last position of the linked
list
    Output: newNode with element e will be inserted into last position of
list.
    newNode<-createNewNode(e)
    tmp<-tail.prev
    tail.prev <-newNode
    newNode.prev<-tmp
    tmp.next <- newNode

    newNode.next <- tail
```

C-2.1 Describe, in pseudo-code, a link-hopping method for finding the middle node of a doubly linked list with header and trailer sentinels, and an odd number of real nodes between them. (Note: This method can only use link-hopping; it cannot use a counter.) What is the running time of this method?

```
Algorithm findMiddle(L)
    Input : List L with odd number of nodes
    Output : middle position of L
    p<-L.first()
    q<-L.last()
    while p != q do
        p<-L.after(p)
        q<-L.before(q)

    return p
```

C-2.2 Describe, in pseudo-code, how to implement the queue ADT using two stacks. What is the running time of the enqueue() and dequeue() methods in this case?

```
S1<-Empty Stack                              | For, enqueuer(),running time =
S2<-Empty Stack                              | O(1)
                                             |
enqueue(val)                                 |
    if size() = N - 1 then                   |
        throw FullQueueException             |
    S1.push(val)                             |
                                             |
                                             |
                                             |
dequeue()                                    |
    if S2.isEmpty() then                     | O(1)
        while !S1.isEmpty() do               | O(n)
            S2.push(S1.pop())                | O(n)
    if !S2.isEmpty() then                    | O(1)
        return S2.pop()                      | O(1)
    else                                     |
        throw EmptyStackException            | So, total running time = O(n)
```

C-2.3 Describe how to implement the stack ADT using two queues. What is the running time of the push() and pop() methods in this case?

```
Q1<-Empty Queue
Q2<-Empty Queue

push(val)
    if size() = N - 1 then
        throw FullStackException
    Q1.enqueue(val)

pop()
    if Q2.isEmpty() then
        while !Q1.isEmpty()
            Q2.enqueue(Q1.dequeue())
    if !Q2.isEmpty() then
        Q2.dequeue()
    else
        throw EmptyQueueException
```

Running time of push() operation = O(1)

O(1)
O(n)
O(n)
O(1)
O(1)
So, total running time for pop operation = O(n)

C-2-4 Describe a recursive algorithm for enumerating all permutations of the numbers {1,2,...,n}. What is the running time of your method?

```
Algorithm perm(S, int n)
    Input: Sequence S with n elements
    Output: List L containing all the permutation

    if n = 1 then
        L.insertLast(S)
        return;
    while i < S.size() do
        S.swapElements(S.rankOf(i),S.rankOf(n-1))
        perm(S,n-1)
        S.swapElements(S.rankOf(i),S.rankOf(n-1))

Running time = O(n!)
```

C-2-5 Describe the structure and pseudo-code for an array-based implementation of the vector ADT that achieves $O(1)$ time for insertions and removals at rank 0, as well as insertions and removals at the end of the vector. Your implementation should also provide for a constant-time elemAtRank method.

```
Algorithm insertAtRank0(obj)
    Input: the object obj for inserting
    if V.size() = n-1
        throw fullException
    f<-(f - 1 + n ) mod n

    V[f] <- obj



Algorithm removeAtRank0()
    if !V.isEmpty()
        f<-(f + 1 ) mod n
    else

        throw emptyVectorException



Algorithm insertAtRankEnd(obj)
    Input: the object obj for inserting
    if V.size() = n-1
        throw fullException
    V[r] <- obj

    r <- (r + 1) mod n



Algorithm removeAtRankEnd()
    if !V.isEmpty()
        r<-(r - 1 + n) mod n
    else

        throw emptyVectorException
```