R-2.1

Algorithm insertBefore(p, e)

```
Create new node v
v.element ← e
v.next ← p                      {link v to its successor}
v.prev ← p.prev                 {link v to its predecessor}
(p.prev).next ← v               {link p old predecessor to its new successor}
p.prev ← v                      {link p to its predecessor}
return v
```

Algorithm insertFirst(e)

```
firstPosition ← L.first()               {get the position of the first element in the list}
 firstNode ← insertBefore(firstNode, e)
return firstNode
```

Algorithm insertLast (e)

```
Create new node v
lastPosition ← L.last()
v.element ← e
v.prev ← lastPosition           {link v to its predecessor}
lastPosition.next ← v           {link lastPosition to its new successor}
return v
```

C-2.1

Algorithm findMiddle(L)

{Input: L is a doubly linked list}

{output: middle node of L}

```
h ← L.header                    1
t ← L.trailer                   1
while h ¬ = t do                n/2
        h ← L.after(h)          n/2
        t ← L.before(t)         n/2
return h                        1
```

The running time for findMiddle(L) is O(n)

C-2.2

```
Algorithm enqueue(o)
        S1.push(o)                          1

Algorithm dequeue()
        If S2.Empty() then                  1
                While ¬ S1.isEmpty() do     n
                        S2.push(S1.pop())   2n
        Return S2.pop()                     1
```

The running time of enqueue is O(1)
The running time of dequeue is O(n)

C-2.3

```
Algorithm push(o)
        Q1.enqueue(o)                       1

Algorithm pop()
        While Q1.size()>1 do                n
                Q2.enqueue(Q1.enqueue())    2n
        e ← Q1.dequeue()                    1
        tmp ← Q2                            1
        Q2 ← Q1                             1
        Q1← tmp                             1
        Return e                            1
```

The running time of enqueue is O(1)
The running time of dequeue is O(n)

C-2-4

```
Algorithm permuteNumbers(s)
        {Input sequence s}
        {output sequence containing permutations of s}

        create new sequence permutedList
        create new sequence permutedListInner

        t ← skipFirstElement(s)                 {copy all of the elements in s except the first one to t}

        if s.Size()>1 then
                permutedListInner ← permuteNumbers (t)
        else
                permutedListInner.addLast(t)

        for each permutation in permutedListInner
                for i←0 to  s.size()-1 do
                        singlePermutation ← copy(permutation)
                        singlePermutation.addAtRank(i, s.first())
                        permutedList.add(singlePermutation)

        return permutedList

Algorithm skipFirstElement (s)
        {Input sequence s}
        {copy all of the elements in s except the first one to t}

        Create new sequence t
        For i←1 to s.size()-1 do
                t.addLast(s. elemAtRank(i))

        return t
```

C-2-5

Algorithm size()
    Return (N-f +t) mod N

Algorithm isEmpty()
    return (f = t)

Algorithm insertFront(o)
    If size() = N-1 then
        Throw vectorFullException()
    else
        f ← (f-1) mod N
        V[f] ← o

Algorithm deleteFront()
    If isEmpty() then
        Throw vectorEmptyException()
    else
        f ← (f+1) mod N
        V[f] ← null

Algorithm insertLast(o)
    If size() = N-1 then
        Throw vectorFullException()
    else
        t ← (t+1) mod N
        V[t] ← o

Algorithm deleteLast()
    If isEmpty() then
        Throw vectorEmptyException()
    else
        t ← (t-1) mod N
        V[t] ← null

Algorithm elementAtRank(r)
    If r<0 V r > size() then
        Throw outOfIndexException()
    Else
        Pos ← (N-f +r) mod N
        Return V[pos]