R.1

```
Algorithm findMiddle(L):
    first := L.first()                      O1
    last := L.last()                        O1
    while first != last do                  On
        first := L.after(first)             On
        last := L.before(last)              On
    return last.                            O1
```

Running time : O(n)

R2

stackOne := []
stackTwo := []

Algorithm enqueue (L)
    stackOne.push(L) O(1)

Algorithm dequeue()
    if !stackTwo.isEmpty then O(n)
        return stackTwo.pop() O(n)
    While !stackOne.isEmpty() do O(n)
        removed := stackOne.pop() O(n)
        stackTwo.push(removed) O(n)
    return stackTwo.pop(). O(1)
runtime time: O(n)

2.3

$Q_1 := []$
$Q_2 := []$

Algorithm push(element)
    if !Q1.isEmpty() then O(1)
        Q1.enqueue(element) O(1)
    else
        Q2.enqueue(element) O(1)

Algorithm pop()
    temp := null O(1)
    if Q1.isEmpty() && Q2.isEmpty() then O(1)
        return temp. O(1)
    if !Q1.isEmpty() then O(1)
        while Q1.size > 1 do O(n)
            Q2.enqueue(Q1.deque) O(n)
        temp := Q1.deque() O(1)
    else
        while Q2.size > 1 do O(n)
            Q1.enqueue(Q2.deque) O(n)
        temp := Q2.deque() O(1)
    return temp O(1)
Runtime Time : O(n)

R.2.1

```
Algorithm Insert Before (p, e)
    newNode := createNode                    O(1)
    newNode.element := e                     O(1)
    p.previous.next := newNode               O(1)
    newNode.previous := p.previous           O(1)
    newNode.next := p                        O(1)
    p.previous := newNode                    O(1)

Algorithm Insert First (e)
    newNode := createNode
    newNode.element := e
    newNode.next := header.next
    header.next.previous := newNode
    newNode.previous := header
    header.next := newNode

Algorithm Insert Last (e)
    newNode := createNode
    newNode.element := e
    if header == null then header := newNode
    else
        temp := header
        while temp.next != null do
            temp := temp.next
        temp.next := newNode
        newNode.previous := temp
```

A. Algorithm removeDups(S)

L	A	
O_1	O_1	if S.size < 2 then
O_1	O_1	return S
O_1	O_1	pointcheck := S.after(S.first())
O_n	O_n	DO
O_n	O_n	first := S.first()
$O(n^2)$	$O(n^2)$	while first != pointcheck do
$O(n^2)$	$O(n^2)$	if first.element == pointcheck.element() then
$O(n^2)$	$O(n^2)$	first := S.after(pointcheck)
$O(n^2)$	$O(n^3)$	S.remove(pointckeck)
		first := S.before(First)
		break
$O(n^2)$	$O(n^2)$	first := S.after(first)
$O(n^2)$	$O(n^2)$	pointcheck := S.after(first)
$O(n)$	$O(n)$	while ~~S.first() != pointcheck~~ pointcheck != null
$O(1)$	$O(1)$	return S

In Array $O(n^3)$
In List $O(n^2)$

the best ADT is hashSet

B Algorithm PowerSet (n)

O(1) result := new Sequence();

O(n²) powerSetRecursive (n , result, new Sequence())

O(1) return result

O(n³) Algorithm powerSetRecursive (n, result, subset)

if n==0 then

O(n) result.insertLast(subset)

else

O(n³) powerSetRecursive (n-1, result, subset)

O(n) newSubset := new Sequence()

O(n²) for i := 0 to subset.size() do

O(n³) newSubset.insertLast(subset.elemAtRank(i))

O(1) newSubset.insertLast(n)

O(n²) powerSetRecursive(n-1, result, newSubset).