Pouria Moradpour - 4024023040

Question 1

Please Note That;

It is assumed that we already have a linked list. (There is already a pseudo-code *implementation* of linked list and it's queries in the lecture notes. Therefore, I won't copy-paste them here to avoid clutter and instead just it)

$$L : Pre - Defined\ Linked\ List$$

Solution

Since we are required to do both $Push$ and Pop in $O(1)$ and the implementation is based on a singly linked list, I put the top of the stack to be the $head$ of the linked list as if I were to use $tail$ It would require traversal and it would take $O(n)$ time to pop.

Setup:

```
Stack {
        L    // Pre-defined linked list
        top: int    // indicating the top of the stack

}
```

Pseudo-code for $push$: (S is the Stack defined above and x is the item that we want to push)

$StackPush(S, x)$

```
x.next = L.head
L.head = x
top = top + 1
```

Pseudo-code for pop: (S is the Stack defined above)

$StackPop(S)$

```
if S.top == 0:
        return "underflow"
else:
        temp = L.head
        L.head = L.head.next
```

```
        top = top - 1
        return temp
```

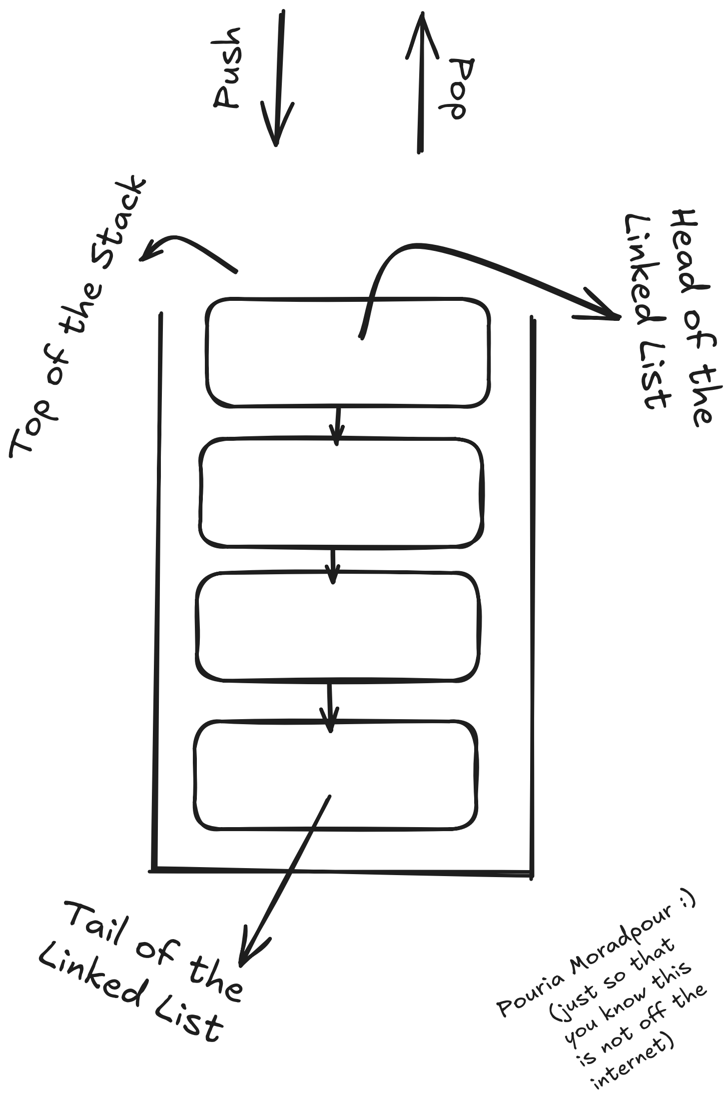
Pseudo-code for $peek$ (S is the stack defined above, I know this is not mandatory but I define it so I can use it later in **Question 3**)

$StackPeek(S)$

```
if S.top == 0:
        return "underflow"
else:
        peek = S.StackPop(S)
        S.StackPush(S, peek)
        return peek
```

This operation returns the top value of stack without removing it.

Visual Representation



Question 2

Please Note That;

It is assumed that we already have a linked list. (There is already a pseudo-code *implementation* of linked list and it's queries in the lecture notes. Therefore, I won't copy-paste them here to avoid clutter and instead just use it)

$$L : Pre - Defined\ Linked\ List$$

Solution

Since we are required to do both $Enqueue$ and $DeQueue$ in $O(1)$ and the implementation is based on a singly linked list, I put the $rear$ of the stack to be the $tail$ of the linked list as if I were to use $head$ It would require traversal and it would take $O(n)$ time to $Dequeue$.

Setup:

```
Queue {
        L    // Pre-defined linked list
        front = L.head   // indicating the front of the queue
        rear = L.tail   // indicating the rear of the queue

}
```

Pseudo-code for $EnQueue$: (Q is the Queue defined above and x is the item that we want to EnQueue)

$EnQueue(Q, x)$

```
if Q.rear == Q.maxsize then:
        return "overflow"
else:
        L.tail.next = x
        x.next = NULL
        L.tail = x
        rear = L.tail
```

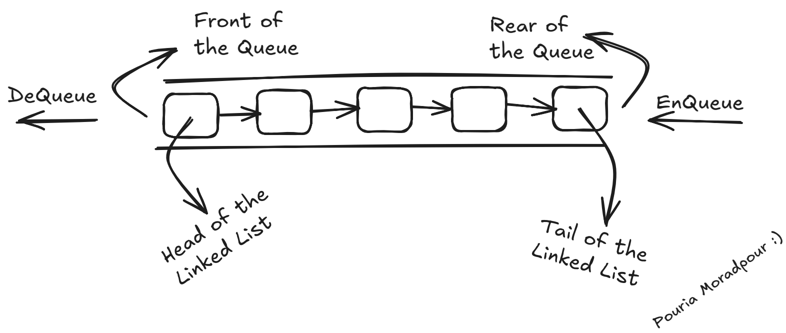
Pseudo-code for $DeQueue$: (Q is the Queue defined above)

$DeQueue(Q)$

```
if front == reaar then:
        return "underflow"
else:
```

```
        temp = L.head
        L.head = L.head.next
        front = L.head
        return temp
```

Visual Representation



Question 3

First Part

in order to implement the desired data structure I am using the $Stack$ Data Structure that I defined in **Question 1**.

$$S : Stack$$

Solution

Setup:

```
MinDS {
        MainStack: Stack // an instance of the Stack data structure that I defined above
        MinStack: Stack  // an instance of the Stack data structure that I defined above
}
```

$MinStack$ is defined to hold the minimum value of the $MainStack$ on it's top at any given point.

Now, I define the operations required in the assignment:

x is the item we intend to push to $MinDS$ and M is an instance of $MinDS$ defined above.

$MinDSPush(M, x)$

```
M.MainStack.StackPush(MainStack, x)
if M.MinStack.top == 0 OR x < M.MinStack.peek(MinStack):
        M.MinStack.StackPush(MinStack, x)
```

this pushes x to the $MainStack$ but only pushes it to the $MinStack$ if it is the new minimum

M is an instance of $MinDS$ defined above.

$MinDSPop(M)$

```
if M.MainStack.top == 0:
        return "underflow"
else:
        popped = M.MainStack.pop(M.MainStack)
        if popped == M.MinStack.StackPeek(M.MinStack):
                M.MinStack.pop(M.MinStack)
        return popped
```

pops from $MainStack$ but also pops from $MinStack$ if the popped value is minimum

M is an instance of $MinDS$ defined above.

$MinDSGetMin(M)$

```
if M.MainStack.top == 0:
        return "underflow"
else:
        return M.MinStack.StackPeek(M.MinStack)
```

returns the top value of $MinStack$ if it is not empty

Second Part

The best thing that I came up with (albeit using external sources as well) was to implement the desired DS on top of a $queue$ (implemented in **Question 2**) and a $deque$ (implemented in **Question 5**).

Solution

Setup

```
MaxDS{
        MainQueue: Queue  // an instance of queue defined in Question 2
```

```
        MaxDeque: Deque  // an instance of deque defined in Question 5
}
```

Operations

x is the item we intend to push to $MaxDS$ and M is an instance of $MaxDS$ defined above.

$MaxDSEnQueue(M, x)$

```
M.MainQueue.EnQueue(M.MainQueue, x)

while M.MaxDeque AND M.MaxDeque.getRear() < x:
        M.MaxDeque.DeleteRear()
M.MaxDeque.addRear(x)
```

$MaxDSDeQueue(M)$

```
if not M:
        return "underflow"
else:
        temp = M.MainQueue.DeQueue(M.MainQueue)
        if temp == M.MaxDeque.getFront():
                M.MaxDque.deleteFront()
        return temp
```

$MaxDSGetMax(M)$:

```
if not M:
        return "underflow"
else:
        temp = M.MaxDeque.getFront()
        M.MaxDeque.deleteFront()
        return temp
```

Question 4

4-A Solution

In order to Implement a $Queue$ using two $Stack$s I use one to store $EnQueue$ items and the other one to $DeQueue$ from.

Setup

```
Queue{
        Stack1: Stack //  an instance of Stack defined in Question 1
        Stack2: Stack  // an instance of Stack defined in Question 1
}
```

`Stack1` is used to handle $EnQueue$ while `Stack2` is mostly used to handle $DeQueue$

Q is the Queue defined above and x is the item that we want to EnQueue

$EnQueue(Q, x)$

```
Q.Stack1.StackPush(Q.Stack1, x)
```

this happens in $O(1)$

$DeQueue(Q)$

```
if not stack2:
        while stack1:
                stack2.push(stack1.pop())
if stack2:
        return stack2.pop()
else:
        return "underflow"
```

this one however, has a time complexity of $O(n)$ (worst case)
(By the way, I did some research and it seems like the overall amortized **time complexity is** $O(1)$ as each element is moved at most once)

for n operations, the time complexity would be $O(n)$ for both $EnQueue$ and $DeQueue$.

4-B Solution

it is in fact possible to implement a $Stack$ based on a single $Queue$ (which I implemented in **Question 2**).

Setup

```
Stack{
        Q: Queue  // an instance of Queue which I defined in Question 2
        Size =
}
```

Operations

Pseudo-code for $push$: (S is the Stack defined above and x is the item that we want to push)

$StackPush(S, x)$

```
S.Queue.EnQueue(S.Queue, x)
for i = 1 to length(S.Queue):
        S.Queue.Enqueue(S.Queue, S.Queue.DeQueue(S.Queue))
```

This pushed the desired element into the queue and then reverses it so that the element in the front would be the same as top of the stack.
this operation happens in $O(n)$ time

Pseudo-code for pop: (S is the Stack defined above)

$StackPop(S)$

```
if not S.Queue:
        return "underflow"
else:
        return S.Queue.DeQueue(S.Queue)
```

this $Dequeue$s the queue which returns the element on top of the stack
this operation happens in $O(1)$ time

for n number of operations;

- $StackPush$'s time complexity would be $O(n^2)$
- $StackPop$'s time complexity would be $O(n)$
  so it *is* different to part A

Question 5

Solution

In order to *actually* implement a deque, I am using a circular array to implement the deque.

That's why I increment like this:

$$(current\ index + 1)||n$$

and decrement like this:

$$(current\ index - 1 + n)||n$$

Setup

```
Deque{
        arr: circular array   // The circular array on top of which the deque is defined
        size: int   // The size of the array (also the last index when 1-indexing)
}
```

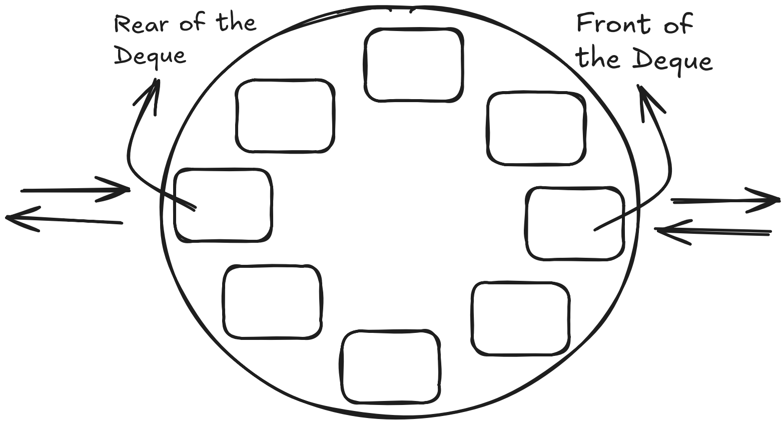
Operations

I am defining $Deque$ and it's operations as a class and it's methods. (some methods are not necessary for this question but I have implemented them so I can use them to conveniently solve **Question 3 - Part 2**)

```
class Deque:
    def __init__(capacity):
        arr = new array of size capacity
        front = -1
        rear = -1
        max_size = capacity

    def isEmpty():
        return front == -1

    def isFull():
        return (rear + 1) % max_size == front

    def insertFront(value):
        if isFull():
            print("Deque is full")
            return
        if isEmpty():
            front = 0
            rear = 0
        else:
            front = (front - 1 + max_size) % max_size
        arr[front] = value

    def insertRear(value):
        if isFull():
            print("Deque is full")
            return
        if isEmpty():
            front = 0
```

```
            rear = 0
        else:
            rear = (rear + 1) % max_size
        arr[rear] = value

    def deleteFront():
        if isEmpty():
            print("Deque is empty")
            return
        if front == rear:  // Only one element
            front = -1
            rear = -1
        else:
            front = (front + 1) % max_size

    def deleteRear():
        if isEmpty():
            print("Deque is empty")
            return
        if front == rear:  // Only one element
            front = -1
            rear = -1
        else:
            rear = (rear - 1 + max_size) % max_size

    def getFront():
        if isEmpty():
            print("Deque is empty")
            return None
        return arr[front]

    def getRear():
        if isEmpty():
            print("Deque is empty")
            return None
        return arr[rear]
```

Visual Representation



Question 6

Looks like it is actually possible to implement $Quack$ using only 2 stacks, here is the solution:

Solution

Setup

```
Quack{
        Stack1: Stack  // an instance of Stack defined in Question 1
        Stack2 : Stack  // an instance of Stack defined in Question 1
}
```

Operations

Q is an arbitrary instance of $Quack$ defined above. x is the element we want to $Push$

$PushToRight(Q, x)$

```
Q.Stack1.StackPush(Q.Stack1, x)
```

$PopFromRight(Q, x)$

```
if Q.Stack1:
        return Q.Stack1.StackPop(Q.Stack1)
else if Q.Stack2:
        while Q.Stack2:
                Q.Stack1.StackPush(Q.Stack2.StackPop())
        return Q.Stack1.StackPop()
```

```
else:
        return "underflow"
```

empties `Stack2` and pushes it in reverse order to `Stack1

$PullFromLeft(Q, x)$

```
if Q.Stack2:
        return Q.Stack2.StackPop()
else if Q.Stack1:
        while Q.Stack:
                Q.Stack2.StackPush(Q.Stack1.StackPop())
        return Stack2.StackPop()
else:
        return "underflow"
```

empties `Stack1` and pushes it in reverse order to `Stack1`