Problem: To simulate a deque (unrestricted input/output) using stacks, where stack is implemented as an array

Deque operations:
a) Element insertion at rear or front if queue not full
b) Element deletion from rear or front if queue not empty

Stack operations:
a) Push element on top if stack not full
b) Pop element from top if stack not empty

Sub-algorithms for Stack operations to emulate Deque operations:

Assumptions:

1. A linear array is used as the stack home for the deque
2. *Queue_size* = size of the deque to be designed
3. *Stack_size* = size of the stacks to be used for deque handling
4. *Queue_size = Stack_size*
5. *rear*, *front, Queue_rear, Queue_front*: global variables
   a. *rear* = pointer to deque rear for enqueue operations
   b. *front* = pointer to deque front for enqueue operations
   c. *Queue_rear =* pointer to deque rear for dequeue operations
   d. *Queue_front =* pointer to deque front for dequeue operations
6. *rear* operates in between 0 to *Queue_size*-1
7. *Queue_rear* is set to zero as long as Queue-rear is non-empty
8. *front* operates from *Queue_size* – 1 to 0
9. *Queue_front* is set to *Queue_size* – 1 as long as Queue-front is non-empty
10. Empty queue conditions:
    a. *front* == *rear* == *Queue_front* == *Queue_rear* == -1
    b. Queue-rear empty: *rear* == *Queue_rear* == *-1* && *Queue_front* == *Queue_size – 1* && *front* >=0
    c. Queue-front empty: *front* == *Queue_front* == *-1* && *Queue_rear* == *0* && *rear* >=0
11. Full queue conditions:
    a. *Queue_rear* == 0 && *rear* = *Queue_size* – 1 && *front* == *Queue_front* == -1
    b. *front* == 0 && *Queue_front* == *Queue_size* – 1 && *rear* == *Queue_rear* == -1
    c. *rear >=0* && *rear* == *front* – 1 && *Queue_rear* == 0 && *Queue_front* == *Queue_size* – 1
12. *val* = value to be entered into or collected from queue
13. Stack_rear: Stack to handle enqueue operations at Queue-rear end
14. Stack_front: Stack to handle enqueue operations at Queue-front end
15. Stack: Stack to support dequeue operations at both ends of deque (to maintain FIFO ordering of basic queues)

Index = 0				Index = *Queue_size-1*
Queue_rear *rear* (is incremented for every push) Enqueue at *rear*	…			*Queue_front* *front* (is decremented for every push) Enqueue at *front*

Dequeue from *Queue_rear* Stack_rear maintains Queue-rear		Dequeue from *Queue_front* Stack_front maintains Queue-front

a) Algo_stack_push_rear_Enqueue:

Input: *val* to be inserted at *rear*

Output: 'Successful' or 'Unsuccessful' entry message

Steps:

1. If (*rear* == *Queue_rear* == -1) then
   a. The Queue-rear end of deque is empty
   b. Update *rear* to make space for *val*
      [*rear* = 0]
   c. Update *Queue_rear* to indicate Queue-rear is not empty
      [*Queue_rear* = 0]
   d. Goto Step 4
2. If (*Queue_rear* == 0 && *rear* = *Queue_size* – 1 && *front* == *Queue_front* == -1) ||
   (*rear >=0* && *rear* == *front* – 1 && *Queue_rear* == 0 && *Queue_front* == *Queue_size* – 1)
   then
   a. Display "The deque is full"
   b. Goto Step 5
3. Increment *rear* to make space for *val*
   [*rear* = *rear* + 1]
4. Place *val* at *rear* in Stack_rear
   [Stack_rear[*rear*] = *val*]
5. Return

b) Algo_stack_pop_rear_Dequeue:

Input: NULL

Output: *val* at *Queue_rear*

Steps:

1. If (*rear == Queue_rear == -1*) then
   a. Display "The Queue-rear end of deque is empty
   b. Return
2. Preparing for dequeue operation from Queue-rear using Stack
   a. If *rear* == 0 then
      i. There is a single element in Stack_rear
      ii. Assign *val* = Stack_rear[*rear*]
      iii. Stack_rear is now rendered empty
         1. *rear* = *Queue_rear* = -1
         2. Goto Step 3
   b. Tranfer elements from Stack_rear to Stack
      i. Initialize pointer for Stack elements
         [*i* = 0]
      ii. Initialize counter for Stack elements
         [*Stack_top* = *rear*]

iii. Push Stack_rear[*Stack_top*] into Stack[*i*]
[Stack[*i*] = Stack_rear[*Stack_ top*]]
iv. Increment *i* to make space for subsequent element in Stack
[*i* = *i* + 1]
v. Decrement *Stack_top* to indicate new stack-top of Stack_rear after Step 2.b.iii.
[*Stack_top* = *Stack_top* – 1]
vi. If (*Stack_top* >= 0) then Goto Step 2.b.iii.

c. Assign *val* = Stack[*i*-1]
[Stack[*i*-1] holds the element at *Queue_rear* = the element being popped from Queue-rear. *i* gets incremented by a point higher and *Stack_top* gets decremented to a point lower than required as guided by Step 2.b.vi]

d. Transfer elements from Stack to Stack_rear
i. Initialize counter for Stack_rear elements
[*rear* = 0]
ii. Initialize *Stack_top* to indicate stack-top of Stack after pop operation in Step 2.c
[*Stack_top* = *i* – 2]
iii. Push Stack[*Stack_top*] into Stack_rear
[Stack_rear[*rear*] = Stack[*Stack_top*]
iv. Increment *rear* to make space for subsequent entries in Stack_rear
[*rear* = *rear* + 1]
v. Decrement Stack_top to indicate new stack-top of Stack after Step 2.d.iii
[*Stack_top* = *Stack_top* – 1]
vi. If (*Stack_top* >= 0) then Goto Step 2.d.iii.

e. Correct *rear* since *rear* is a point higher than required as guided by Step 2.d.vi
[*rear* = *rear* – 1]

3. Return *val*

c) Algo_stack_push_front_Enqueue:

Input: *val* to be inserted at *front*
Output: 'Successful' or 'Unsuccessful' entry message
Steps:

1. If (*front* == *Queue_front* == -1) then
   a. The Queue-front end of deque is empty
   b. Update *front* to make space for *val*
   [*front* = *Queue_size* – 1]
   c. Update *Queue_front* to indicate Queue-front is not empty
   [*Queue_front* = Queue_size – 1]
   d. Goto Step 4
2. If (*front* == 0 && *Queue_front* = *Queue_size* – 1 && *rear* == *Queue_rear* == -1) ||
   (*rear >=0* && *rear* == *front* – 1 && *Queue_rear* == 0 && *Queue_front* == *Queue_size* – 1) then
   a. Display "The deque is full"
   b. Goto Step 5
3. Decrement *front* to make space for *val* [*front* = *front* - 1]
4. Place *val* at *front* in Stack_front [Stack_front[*front*] = *val*]
5. Return

d) Algo_stack_pop_front_Dequeue:

Input: NULL
Output: *val* at *Queue_front*
Steps:

1. If (*front == Queue_front == -1*) then
   a. Display "The Queue-front end of deque is empty
   b. Return
2. Preparing for dequeue operation from Queue-front using Stack
   a. If *front* == *Queue_size* – 1 then
      i. There is a single element in Stack_front
      ii. Assign *val* = Stack_front[*front*]
      iii. Stack_front is now rendered empty
         1. *front* = *Queue_front* = -1
         2. Goto Step 3
   b. Tranfer elements from Stack_front to Stack
      i. Initialize pointer for Stack elements
         [*i* = 0]
      ii. Initialize counter for Stack elements
         [*Stack_top* = *front*]
      iii. Push Stack_front[*Stack_top*] into Stack[*i*]
         [Stack[*i*] = Stack_front[*Stack_ top*]]
      iv. Increment *i* to make space for subsequent element in Stack
         [*i* = *i* + 1]
      v. Increment *Stack_top* to indicate new stack-top of Stack_front after Step 2.b.iii.
         [*Stack_top* = *Stack_top* + 1]
      vi. If (*Stack_top* <= *Queue_size* – 1) then Goto Step 2.b.iii.
   c. Assign *val* = Stack[*i*-1]
      [Stack[*i*-1] holds the element at *Queue_front* = the element being popped from Queue-front. *i* gets incremented by a point higher and *Stack_top* gets incremented to a point higher than required as guided by Step 2.b.vi]
   d. Transfer elements from Stack to Stack_front
      i. Initialize counter for Stack_front elements
         [*front* = *Queue_size* - 1]
      ii. Initialize *Stack_top* to indicate stack-top of Stack after pop operation in Step 2.c
         [*Stack_top* = *i* – 2]
      iii. Push Stack[*Stack_top*] into Stack_front
         [Stack_front[*front*] = Stack[*Stack_top*]
      iv. Decrement *front* to make space for subsequent entries in Stack_front
         [*front* = *front* - 1]
      v. Decrement Stack_top to indicate new stack-top of Stack after Step 2.d.iii
         [*Stack_top* = *Stack_top* – 1]
      vi. If (*Stack_top* >= 0) then Goto Step 2.d.iii.
   e. Correct *front* since *front* is a point lower than required as guided by Step 2.d.vi
      [*front* = *front* + 1]
3. Return *val*

Main algorithm:
Algo_main:
Input: *choice* to indicate enqueue/ dequeue at *Queue_front*/ *Queue_rear*;

val if enqueue operation;
change to indicate user's request to update *choice* entry;
cont to indicate user's wish to continue deque operations

Output: *val* if dequeue operation; deque status

Steps:

1. Initialize queue *front*, *rear* pointers for new queue operations
2. Display available user choices:
   a. Display "Choice 1: Enqueue at queue rear"
   b. Display "Choice 2: Dequeue at queue rear"
   c. Display "Choice 3: Enqueue at queue front"
   d. Display "Choice 4: Dequeue at queue front"
3. Request user *choice* of queue operations
4. Check validity of *choice* entry
   a. If (choice < 1 || choice > 4) then
      i. Display "Incorrect Choice!"
      ii. Ask user if wants to update entry (1 = Yes | 0 = No) [Input *change*]
      iii. If change == 1 Then Goto Step 2 Else Goto Step 8
5. If *choice* == 1 then
   a. Display "Enqueue at queue rear"
   b. Request user input for queue [Input *val*]
   c. Invoke Algo_stack_push_rear_Enqueue(*val*)
   d. Goto Step 6

   If *choice* == 2 then
   e. Display "Dequeue from queue rear"
   f. Display results of Algo_stack_pop_rear_Dequeue
   g. Goto Step 6

   If *choice* == 3 then
   h. Display "Enqueue at queue front"
   i. Request user input for queue [Input *val*]
   j. Invoke Algo_stack_push_front_Enqueue(*val*)
   k. Goto Step 6

   If *choice* == 4 then
   a. Display "Dequeue from queue front"
   b. Display results of Algo_stack_pop_front_Dequeue
   c. Goto Step 6
6. Inquire if user wants to continue deque operations (1 = Yes | 0 = No) [Input *cont*]
7. If *cont* == 1 then Goto Step 2 else Goto Step 8
8. Display "End of Deque operations"
9. Stop

Operation Sequence:
Initializations:
Queue_size = 10
rear = front = -1
Queue_rear = Queue_front = -1

Enqueue in Queue-rear: 20
Queue contents: 20

rear = Queue_rear = 0

Enqueue at Queue-rear: 30
Queue contents: 20   30
rear = 1
Queue_rear = 0

Enqueue at Queue-rear: 40
Queue contents: 20   30   40
rear = 2
Queue_rear = 0

Dequeue at Queue-rear: 20
Queue contents: 30   40
rear = 1
Queue_rear = 0

Enqueue at Queue_front: 50
Queue contents: 30   40   50
rear = 1
Queue_rear = 0
front = 9
Queue_front = 9

Enqueue at Queue_front: 60
Queue contents: 30   40   60   50
rear = 1
Queue_rear = 0
front = 8
Queue_front = 9

Enqueue at Queue_front: 70
Queue contents: 30   40   70   60   50
rear = 1
Queue_rear = 0
front = 7
Queue_front = 9

Dequeue at Queue_front: 50
Queue contents: 30   40   70   60
rear = 1
Queue_rear = 0
front = 8
Queue_front = 9

Enqueue at Queue_rear: 40
Queue contents: 30   40   40   70   60
rear = 2

```
Queue_rear = 0
front = 8
Queue_front = 9

Enqueue at Queue_front: 80
Queue contents: 30   40   40                  80   70   60
rear  = 2
Queue_rear = 0
front = 7
Queue_front = 9

Enqueue at Queue_rear: 50
Queue contents: 30   40   40   50             80   70   60
rear  = 3
Queue_rear = 0
front = 7
Queue_front = 9

Enqueue at Queue_front: 90
Queue contents: 30   40   40   50        90   80   70   60
rear  = 3
Queue_rear = 0
front = 6
Queue_front = 9

Dequeue at Queue_rear: 30
Queue contents: 40   40   50             90   80   70   60
rear  = 2
Queue_rear = 0
front = 6
Queue_front = 9

Dequeue at Queue_front: 60
Queue contents: 40   40   50                  90   80   70
rear  = 2
Queue_rear = 0
front = 7
Queue_front = 9
```