STACKS & QUEUES.

Stack.

- (A stack is a linear data structure and is alternatively termed as LIFO (Last-In-First-Out).

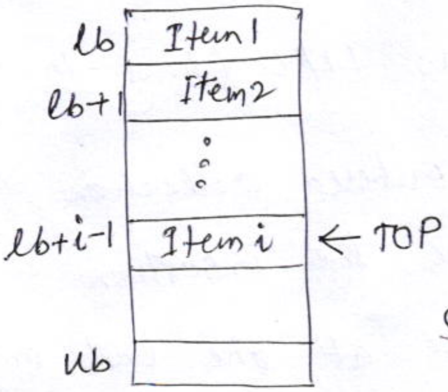
- A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only.

- The insertion and deletion operations in a stack are termed as PUSH and POP respectively. and the position of the stack where these operations are performed is known as the TOP of the stack). An element in a stack is termed an ITEM. The maximum number of elements that a stack can accommodate is termed SIZE.

- There are two ways of representing a stack in memory
  - i) using an array
  - ii) using a linked list

Array representation of Stack.

First we have to allocate a memory block of sufficient size to accomodate the full capacity of the stack. Then starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion



$SIZE = ub + lb - 1$

Array representation of a stack

In the figure,

- ITEM i denotes i^{th} item in stack
- lb and ub denote index range of the array with values 1 and SIZE respectively.
- TOP is a pointer to point the position of the array up to which it is filled

With this representation we have.

EMPTY : $TOP < 1$

FULL : $TOP \geq SIZE$

⊛ Operations on Stacks

The basic operations required to manipulate a stack are:

1. PUSH : To insert an item into a stack.
2. POP : To remove an item from a stack
3. STATUS: To know the present state of a stack.

⊛ PUSH_Array.

Input - The new item ITEM to be pushed onto it

Output - A stack with a newly pushed ITEM at the TOP position

Data Structure - An array A with TOP as the pointer.

1. If TOP ≥ SIZE then
2. Print "Stack is Full"
3. Else
4. TOP = TOP + 1
5. A[TOP] = ITEM.
6. Endif
7. Stop.

⊛ POP_Array.

Input - A stack with elements

Output - Removes an ITEM from the top of the Stack if it is not empty.

Data Structure - An array A with TOP as the pointer.

1. If TOP < 1 then.
2. Print "Stack is Empty"
3. Else
4. ITEM = A[TOP]
5. TOP = TOP - 1

6. Endif

7. Stop.

⊛ STATUS_Array.

Input - A Stack with elements.

Output - States whether it is empty or full, available free space and item at TOP.

Data Structure - An array A with TOP as the pointer

1. If TOP < 1 then.
2. Print "Stack is Empty"
3. Else.
4. If (TOP >= SIZE) then.
5. Print " Stack is Full"
6. Else
7. Print " The element at TOP is ", A[TOP].
8. free = (SIZE - TOP)/ SIZE * 100
9. Print "Percentage of free stack is", free
10. Endif.
11. End if.
12. Stop

Queue

- A queue is a linear list of elements in which deletion (dequeue) can take place at only end called the front and insertion (enqueue) can take place only at the other end called the rear.
- Queues are also called First-In-First-Out (FIFO) lists since the first element in a queue will be the first element out of the queue.
- There are two ways of representing a queue in memory.
  - Using an array.
  - Using a linked list
- A one dimensional array Q[1 ... N] can be used to represent a queue. Here two pointers FRONT and REAR are used to indicate the two ends of the queue.
- With this representation:
  - Queue is empty, if FRONT = REAR = 0.
  - Queue is full, if FRONT = 1 and REAR = N.

⊛ Enqueue Array.

1. If (REAR = N) then
2. Print " Queue is Full "
3. Exit
4. Else
5. If (REAR = 0) and (FRONT = 0) then
6. FRONT = 1
7. Endif
8. REAR = REAR+1
9. Q[REAR] = ITEM
10 Endif.
11 Stop.

⊛ Dequeue - Array.

1. If (FRONT = 0) then.
2. Print "Queue is Empty"
3. Exit
4. Else.
5. ITEM = Q[FRONT]
6. If (FRONT = REAR] // contains single element
7. REAR = 0
8. FRONT = 0.
9. Else
10. FRONT = FRONT + 1

11. Endif
12. Endif.
13. Stop.

Circular Queue

- For a queue represented using an array, when the REAR pointer reaches the end, insertion will be denied even if room is available at the front. One way to avoid this is to use a circular array. Physically, a circular array is the same as an ordinary array, say A[1 - N] but logically it implies that A[1] comes after A[N] or after A[N], A[1] appears.
- The two pointers FRONT & REAR will both move in clockwise direction and this is controlled by the 'mod' operation. For example, if the current pointer is at i, then shift to the next location will be i MOD LENGTH + 1, $1 \le i \le$ LENGTH (where LENGTH is the queue length). If i = LENGTH, then next position for the pointer is 1.

With this representation:

CQueue is empty : FRONT = REAR = 0.

CQueue is Full : FRONT = (REAR MOD LENGTH) + 1

FRONT=0
REAR=0

Initially Empty.

FRONT=1
REAR=3

A	B	C		
↑ F		↑ R		

A, B, C inserted.

FRONT=2
REAR=3

	B	C		
	↑ F	↑ R		

A deleted.

FRONT=2
REAR=5

	B	C	D	E
	↑ F			↑ R

D, E inserted.

FRONT=4
REAR=5

			D	E
			↑ F	↑ R

B, C deleted.

FRONT=4
REAR=1

F			D	E
↑ R			↑ F	

F Inserted

FRONT=5
REAR=1

F				E
↑ R				↑ F

D deleted.

FRONT=5
REAR=3

F	G	H		E
		↑ R		↑ F

G, H Inserted

FRONT=1
REAR=3

F	G	H		
↑ F		↑ R		

E deleted.

⊛ CQInsert

1. If (FRONT = 0) then // Queue is Empty.
    1. FRONT = 1
    2. REAR = 1
    3. CQ[REAR] = ITEM.
2. Else.
    1. next = (REAR Mod LENGTH) + 1
    2. If (next ≠ FRONT) then.
        1. REAR = next.
        2. CQ[REAR] = ITEM.
    3. Else
        1. Print "CQueue is Full"
    4. Endif.
3. Endif.
4. Stop.

⊛ CQDelete

1. If (FRONT = 0) then.
    1. Print "Queue is Empty"
    2. Exit
2. Else.
    1. ITEM = CQ[FRONT]
    2. If (FRONT = REAR) then // Queue has a single element.
        1. FRONT = 0.
        2. REAR = 0.

3. Else

1. FRONT = (FRONT mod LENGTH) + 1

4. Endif

3. Endif.

4. Stop.

Deque

- A deque is a linear list in which elements can be added or removed at either end but not in the middle.

- Deque term comes from double ended queue

- Deque can be represented using a circular array or a double linked list.

Two variations of deque are

1. Input restricted deque : which allows insertions at one end only, but allows deletions at both ends.

II. Output restricted deque : which allows deletions at one end only but allows insertions at both ends.

Priority Queue.

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Recursion.

A function that calls itself directly or indirectly and approaches towards the terminating condition or base condition is called a recursive function.

A recursive function must satisfy the following 2 conditions:

1. There must be certain criteria (terminating condition) called base criteria, for which the function does not call itself.
2. Each time when the function calls itself directly or indirectly it must be closer to the base criteria.

Example: We can express the factorial function in general as.

$$n! = n * (n-1)!$$

$$n! = 1 \quad \text{if } n \leq 1 \quad -(a)$$

$$n! = n * (n-1)! \quad \text{if } n > 1 \quad -(b)$$

In the above definition one can conclude that (a) is the base value and (b) is again redefining the problem in terms of smaller values of n, which are closer to the base values

Tail Recursion

In tail recursion, the very last action of the function is a recursive call to itself and none of the recursive call does additional work like printing, addition etc after the recursive call is complete except to return the value of the recursive call.

The following is tail-recursion

return func-rec (x, y); // no work after recursive call, just return the value of call.

The following is not tail-recursion

return func-rec (x, y) + 1 ; // work after recursive call, add 1 to result

Because once func-rec is done, it must add 1 and then return that value. This is an additional work, hence the above example is not tail recursion.

☒ The Tower of Hanoi Problem.