OXFORD
HIGHER EDUCATION

Data Structures Using C

Reema Thareja, Assistant Professor
Institute of Information Technology and Management
GGS IP University

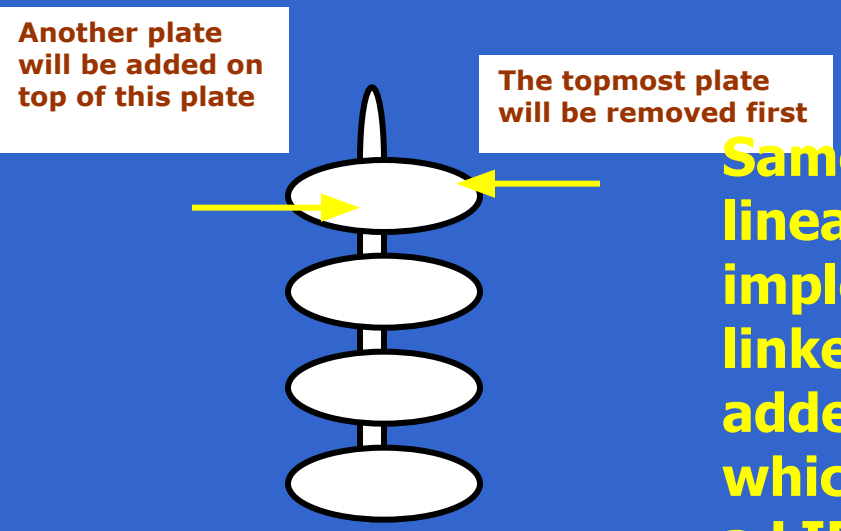
OXFORD
HIGHER EDUCATION

CHAPTER 9

STACKS AND QUEUES

Introduction to Stacks

- Stack is an important data structure which stores its elements in an ordered manner. Take an analogy of a pile of plates where one plate is placed on top of the other. A plate can be removed from the topmost position. Hence, you can add and remove the plate only at/from one position that is, the topmost position.



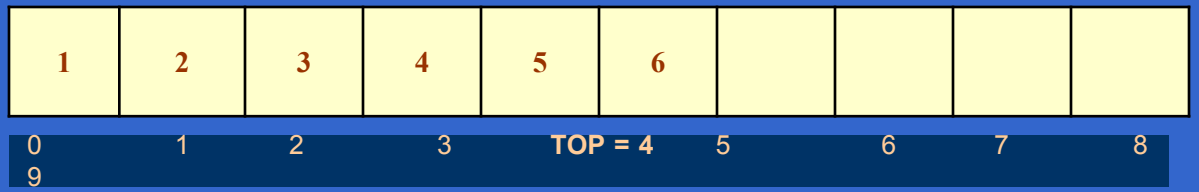
Same is the case with stack. A stack is a linear data structure which can be implemented either using an array or a linked list. The elements in a stack are added and removed only from one end, which is called *top*. Hence, a stack is called a LIFO (Last In First Out) data structure as the element that was inserted last is the first one to be taken out.

Array Representation Of Stacks

- In computer's memory stacks can be represented as a linear array.
- Every stack has a variable TOP associated with it. TOP is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted.
- There is another variable MAX which will be used to store the maximum number of elements that the stack can hold.
- If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX -1, then the stack is full.

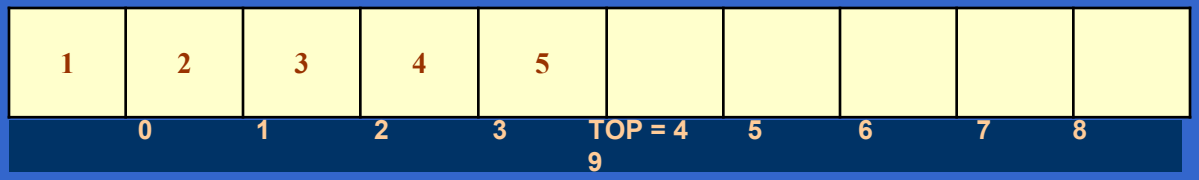
Push Operation

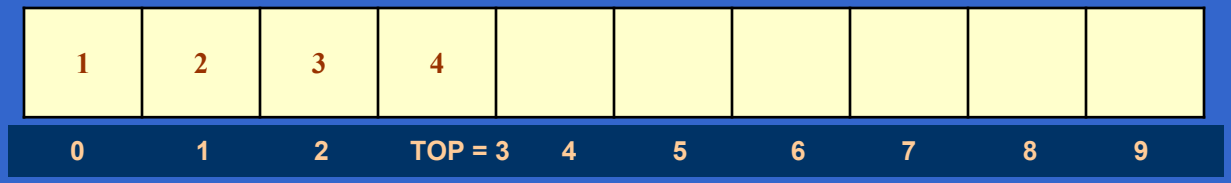
- The push operation is used to insert an element in to the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if TOP=MAX-1, because if this is the case then it means the stack is full and no more insertions can further be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.



Pop Operation

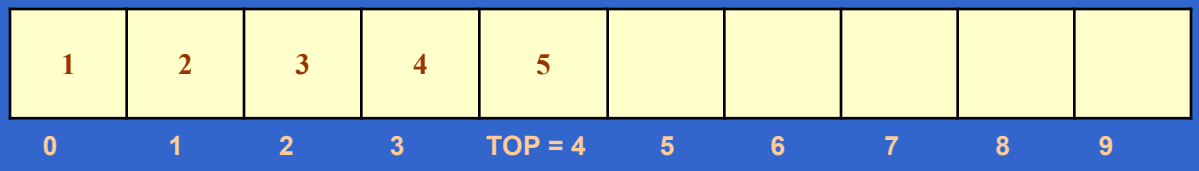
- The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case then it means the stack is empty so no more deletions can further be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.





PeeK Operation

- PeeK is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the peep operation first checks if the stack is empty or contains some elements. For this, a condition is checked. If TOP = NULL, then an appropriate message is printed else the value is returned.



Here PeeK operation will return 5, as it is the value of the topmost element of the stack.

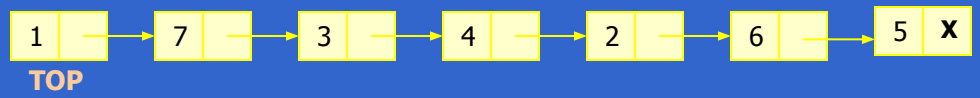
```
Algorithm to PUSH an element in to the stack

Step 1: IF TOP = MAX-1, then
          PRINT "OVERFLOW"
     [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

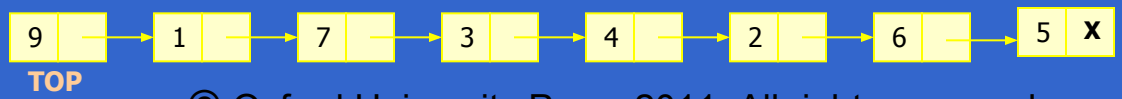
```
Algorithm to POP an element from the stack

Step 1: IF TOP = NULL, then
          PRINT "UNDERFLOW"
     [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

```
Algorithm for Peek Operation

Step 1: IF TOP =NULL, then
     PRINT "STACK IS EMPTY"
          Go TO Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Push Operation on a Linked Stack

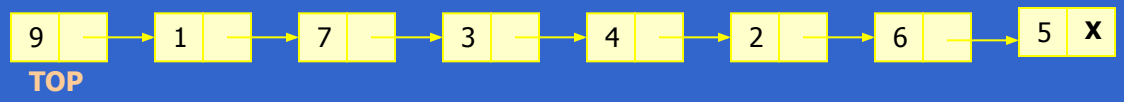
```
Algorithm to PUSH an element in to a linked stack

Step 1: Allocate memory for the new node and name it as New_Node
Step 2: SET New_Node->DATA = VAL
Step 3: IF TOP = NULL, then
           SET New_Node->NEXT = NULL
           SET TOP = New_Node
      ELSE
           SET New_node->NEXT = TOP
           SET TOP = New_Node
      [END OF IF]
Step 4: END
```

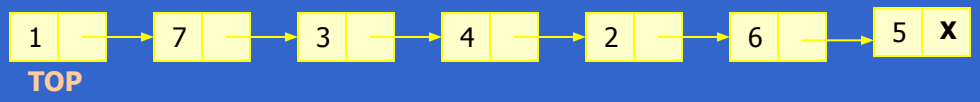




Pop Operation on a Linked Stack

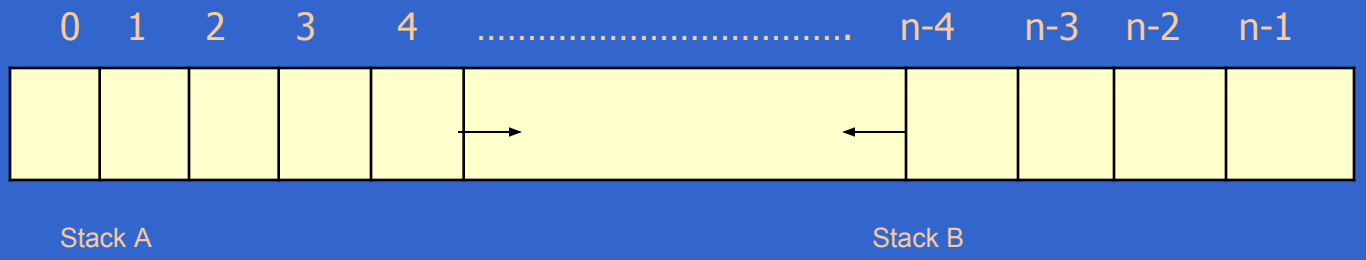
```
Algorithm to POP an element from the stack

Step 1: IF TOP = NULL, then
           PRINT "UNDERFLOW"
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP ->NEXT
Step 4: FREE PTR
Step 5: END
```





Multiple Stacks

- When we had implemented a stack array, we have seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent OVERFLOW conditions will be encountered.
- In case, we allocate a large amount of space for the stack, it will result in sheer wastage of memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- So a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size.



Infix, Postfix And Prefix Notation

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B; here, plus operator is placed between the two operands A and B.
- Although for us it is easy to write expressions using infix notation but computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence, associativity rules and brackets which overrides these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix notation

- Postfix notation was given by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation which is better known as Reverse Polish Notation or RPN.
- In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as A+B in infix notation, the same expression can be written AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets can not alter the order of evaluation.
- Similarly, the expression- (A + B) * C is written as –
- [AB+]*C
- AB+C* in the postfix notation.
- A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

Prefix Notation

- Although a Prefix notation is also evaluated from left to right but the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands. For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator. Like postfix, prefix expressions also do not follow the rules of operator precedence, associativity and even brackets cannot alter the order of evaluation.

- Convert the following infix expressions into prefix expressions
- (A + B) * C
- (+AB)*C
- *+ABC

Evaluation Of An Infix Expression

- **STEP 1: Convert the infix expression into its equivalent postfix expression**

```
Algorithm to convert an Infix notation into postfix notation

Step 1: Add ')" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
     IF a "(" is encountered, push it on the stack
          IF an operand (whether a digit or an alphabet) is encountered,
     add it to the postfix expression.
     IF a ")" is encountered, then;
         aRepeatedly pop from stack and add it to the postfix expression
          until a "(" is encountered.
bDiscard the "(". That is, remove the "(" from stack and do not add it to
 the postfix expression
     IF an operator O is encountered, then;
         aRepeatedly pop from stack and add each operator (popped from the
          stack) to the postfix expression until it has the same precedence
          or a higher precedence than O
bPush the operator O to the stack
     [END OF IF]
     Step 4: Repeatedly pop from the stack and add it to the postfix
     expression until the stack is empty
Step 5: EXIT
```

Exercise: Convert the following infix expression into postfix expression using the algorithm given in figure 9.21.

A - (B / C + (D % E * F) / G)* H

A - (B / C + (D % E * F) / G)* H)

Infix Character Scanned	STACK	Postfix Expression
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
(	(- (+ (	A B C /
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

STEP 2: Evaluate the postfix expression

```
Algorithm to evaluate a postfix expression

Step 1: Add a ")" at the end of the postfix expression
Step 2: Scan every character of the postfix expression and
repeat steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered, push it on the stack
IF an operator O is encountered, then
         pop the top two elements from the stack as A and B
         Evaluate B O A, where A was the topmost element and B
was the element below A.
        Push the result of evaluation on the stack
       [END OF IF]
Step 4: SET RESULT equal to the topmost element of the stack
Step 5: EXIT
```

Let us now take an example that makes use of this algorithm. Consider the infix expression given as "9 - ((3 * 4) + 8) / 4". Evaluate the expression.

The infix expression "9 - ((3 * 4) + 8) / 4" can be written as "9 3 4 * 8 + 4 / -" using postfix notation. Look at table I which shows the procedure.

Character scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Convert Infix Expression To Prefix Expression

```
Step 1: Reverse the infix string. Note that while reversing the
string you must interchange left and right parenthesis.
Step2: Obtain the corresponding postfix expression of the infix
expression obtained as a result of Step1.
Step 3: Reverse the postfix expression to get the prefix expression
```

For example, given an infix expression- (A – B / C) * (A / K – L)

Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parenthesis.

(L – K / A) * (C / B – A)

Step2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step1.

The expression is: (L – K / A) * (C / B – A)

Therefore, [L – (K A /)] * [(C B /) - A]

= [LKA/-] * [CB/A-]

= L K A / - C B / A - *

Step 3: Reverse the postfix expression to get the prefix expression

Therefore, the prefix expression is * - A / B C - / A K L

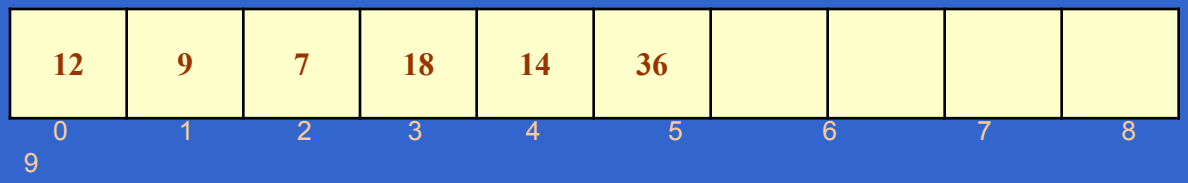
QUEUES

- Queue is an important data structure which stores its elements in an ordered manner. Take for example the analogies given below.
- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for bus. The first person standing in the line will be the first one to get into the bus.

- A queue is a FIFO (First In First Out) data structure in which the element that was inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other one end called front.

Array Representation Of Queue

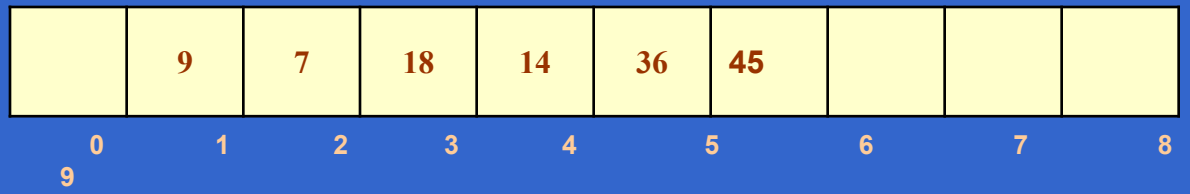
- Queues can be easily represented using linear arrays. As stated earlier, every queue will have front and rear variables that will point to the position from where deletions and insertions can be done respectively.
- Consider a queue shown in figure



Here, front = 0 and rear = 5. If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear. The queue after addition would be as shown in figure

Array Representation Of Queue contd.

- Here, front = 0 and rear = 6. Every time a new element has to be added, we will repeat the same procedure.
- Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue. The queue after deletion will be as shown in figure



Here, front = 1 and rear = 6.
However, before inserting an element in the queue we must check for overflow conditions. An overflow will occur when we will try to insert an element into a queue that is already full. When Rear = MAX – 1, where MAX is the size of the queue that is, MAX specifies the maximum number of elements that the queue can hold.

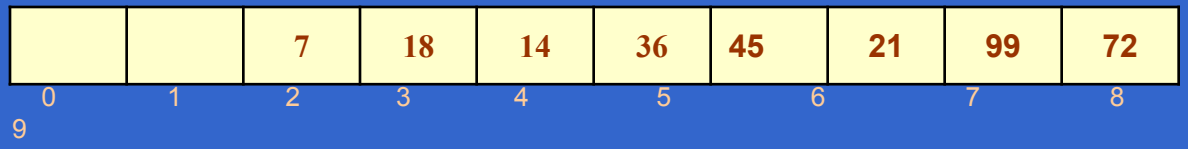
Similarly, before deleting an element from the queue, we must check for underflow condition. An underflow condition occurs when we try to delete an element from a queue that is already empty. If front = -1 and rear = -1, this means there is no element in the queue.

Algorithms to insert and delete an element from the Queue

```
Algorithm to insert an element in the queue

Step 1: IF REAR=MAX-1, then;
          Write OVERFLOW
     [END OF IF]
Step 2: IF FRONT == -1 and REAR = -1, then;
          SET FRONT = REAR = 0
     ELSE
          SET REAR = REAR + 1
     [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: Exit
```

```
Algorithm to delete an element from the queue

Step 1: IF FRONT = -1 OR FRONT > REAR, then;
          Write UNDERFLOW
     ELSE
          SET FRONT = FRONT + 1
          SET VAL = QUEUE[FRONT]
     [END OF IF]
Step 2: Exit
```

Circular Queue

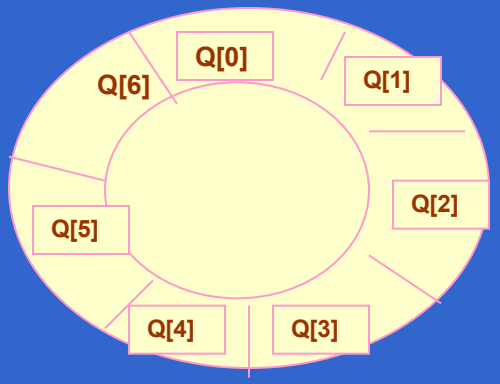


Here, front = 2 and rear = 9.

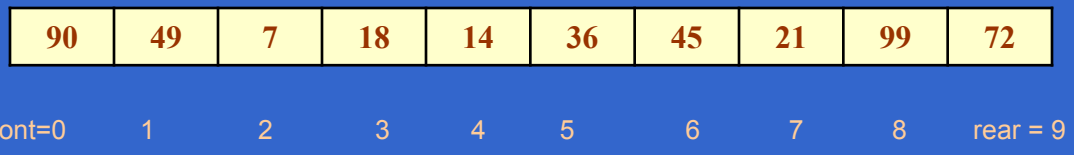
Now, if you want to insert any new element, although there is space but insertion cannot be done because the space is available on the left side. In our algorithm, we have said if rear = MAX – 1, then write OVERFLOW. So as per that, OVERFLOW condition exists. This is the major drawback of a linear queue. Even if space is available, no insertions can be done once rear becomes equal to MAX – 1. Finally, this leads to wastage of space. To cater to this situation, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time consuming especially when the queue is quite large.

The second option is to use a circular queue. In circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in figure

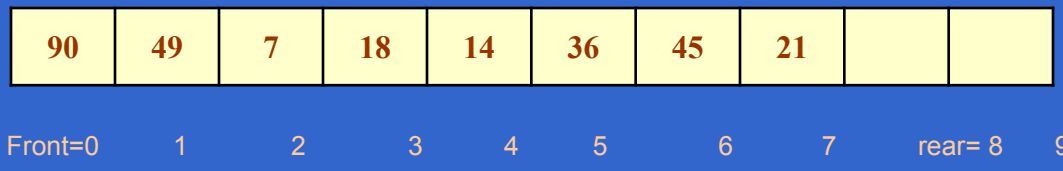
Circular Queue contd.



The circular queue will be full, only when front=0 and rear = Max – 1. A circular queue is implemented in the same manner as the linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations. For insertion we will now have to check for three conditions which are as follows: If front=0 and rear= MAX – 1, then print that the circular queue is full. Look at queue given in figure which illustrates this point



If rear != MAX – 1, then the value will be inserted and rear will be incremented as illustrated in figure



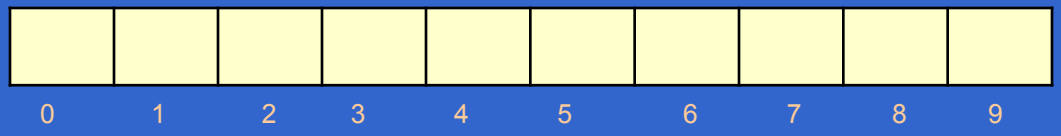
Circular Queue contd.

- If front!=0 and rear=MAX -1, then it means that the queue is not full. So, set rear = 0 and insert the new element there as shown in figure

```
Algorithm to insert an element in the circular queue

Step 1: IF FRONT = 0 and Rear = MAX - 1, then
          Write "OVERFLOW"
     ELSE IF FRONT = -1 and REAR = -1, then;
          SET FRONT = REAR = 0
     ELSE IF REAR = MAX - 1 and FRONT != 0
          SET REAR = 0
     ELSE
          SET REAR = REAR + 1
[END OF IF]
Step 2: SET QUEUE[REAR] = VAL
Step 3: Exit
```

Circular Queue contd.

- After seeing how a new element is added in a circular queue, let us talk about how deletions are performed in this case. To delete an element again we will check for three conditions.
- Look at the figure. If front = -1, then it means there are no elements in the queue. So an underflow condition will be reported.



If the queue is not empty and after returning the value on front, if front = rear, then it means now the queue has become empty and so front and rear is set to -1. This is illustrated in figure

Delete this element and set rear = front = -1

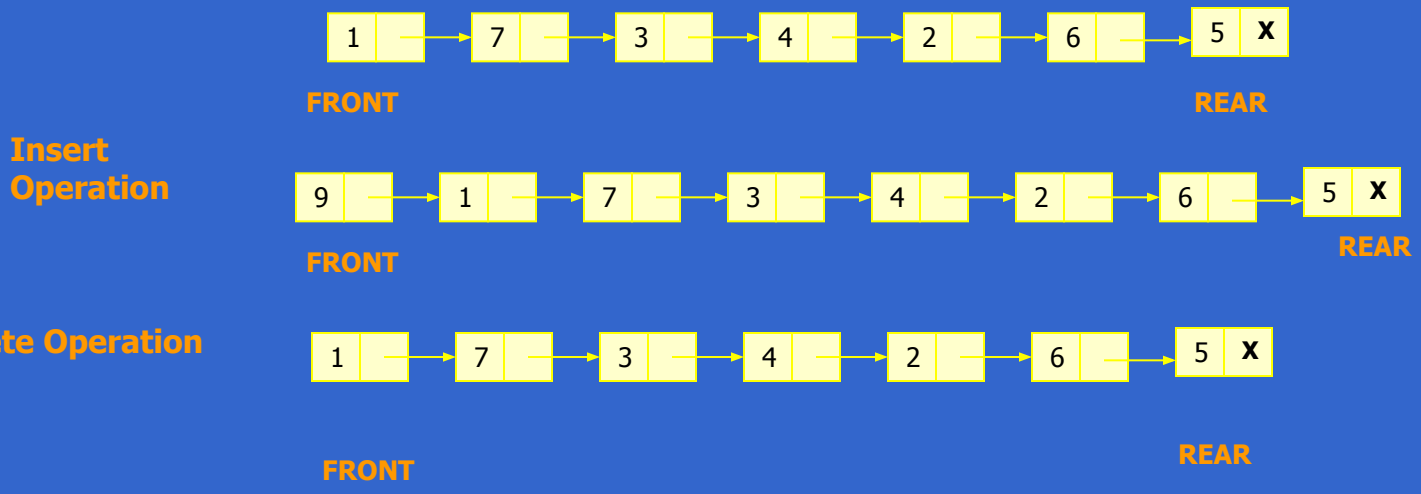
Circular Queue contd.

- If the queue is not empty and after returning the value on front, if front = MAX -1, then front is set to 0. This is shown in figure


```
Algorithm to delete an element from a circular queue

Step 1: IF FRONT = -1, then
          Write "Underflow"
          SET VAL = -1
     [End of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
          SET FRONT = REAR = -1
     ELSE
          IF FRONT = MAX -1
               SET FRONT = 0
          ELSE
               SET FRONT = FRONT + 1
          [End of IF]
     [END OF IF]
Step 4: Exit
```

Linked Representation of a Queue

- In a linked queue, every element has two parts- one that stores data and the other that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions are done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.
- The storage requirement of linked representation of queue with n elements is O(n) and the typical time requirement for operations is O(1).

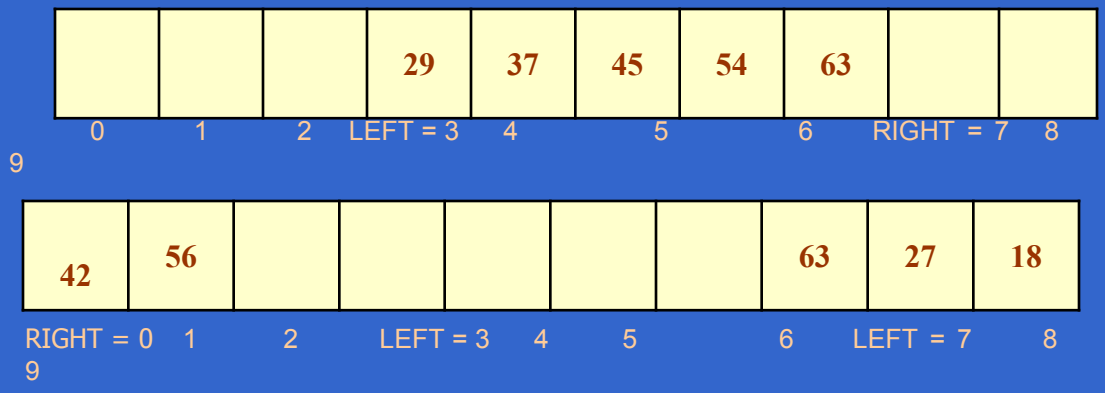


```
Algorithm to delete an element from a linked queue

Step 1: IF FRONT = NULL, then
          Write "Underflow"
          Go to Step 3
     [END OF IF]
Step 2: SET PTR = FRONT
Step 3: FRONT = FRONT->NEXT
Step 4: FREE PTR
Step 5: END
```

```
Algorithm to insert an element in to a linked queue

Step 1: Allocate memory for the new node and name it as PTR
Step 2: SET PTR->DATA = VAL
Step 3: IF FRONT = NULL, then
          SET FRONT = REAR = PTR;
          SET FRONT->NEXT = REAR->NEXT = NULL
     ELSE
          SET REAR->NEXT = PTR
          SET REAR = PTR
          SET REAR->NEXT = NULL
     [END OF IF]
Step 4: END
```

DEQUES

- A deque is a list in which elements can be inserted or deleted at either end. It is also known as a head-tail linked list, because elements can be added to or removed from the front (head) or back (tail).
- However, no element can be added and deleted from the middle. In computer's memory, a deque is implemented either using a circular array or a circular doubly linked list. In a deque, two pointers are maintained, LEFT and RIGHT which points to either end of the deque. The elements in a deque stretch from LEFT end to the the RIGHT and since it is circular, Dequeue[N-1] is followed by Dequeue[0].
- Basically, there are two variants of a double ended queue. They are:
- *Input restricted deque:* In this dequeue insertions can be done only at one of the dequeue while deletions can be done from both the ends.
- *Output restricted deque:* In this dequeue deletions can be done only at one of the dequeue while insertions can be done on both the ends.



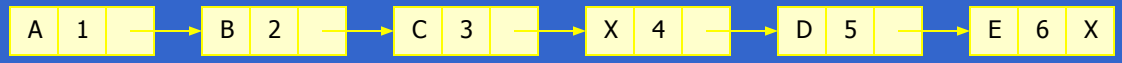
Priority Queues

- A priority queue is an abstract data type in which the each element is assigned a priority. The priority of the element will be used to determine the order in which these elements will be processed. The general rule of processing elements of a priority queue can be given as:
- An element with higher priority is processes before an element with lower priority
- Two elements with same priority are processed on a first come first served (FCFS) basis
- A priority queue can be thought of as a modified queue in which when an element has to be taken off the queue, the highest-priority one is retrieved first. The priority of the element can be set based upon distinct factors.
- A priority queue is widely used in operating systems to execute the highest priority process first. The priority of the process may be set based upon the CPU time it needs to get executed completely.

Priority Queue contd.

- In computer's memory a priority queue can be represented using arrays or linked lists. When a priority queue is implemented using a linked list, then every node of the list will have three parts: (1) the information or data part (ii) the priority number of the element (iii) address of the next element. If we are using a sorted linked list, then element having higher priority will precede the element with lower priority.
- Note: lower priority number means higher priority.

Priority queue after insertion of a new node



Array representation of a priority queue

When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.

We use a two dimensional array for this purpose where each queue will be allocated same amount of space. Given the front and rear values of each queue, the two dimensional matrix can be formed.

Multiple Queues

- When we had implemented a queue array, we have seen that the size of the array must be known in advance. If the queue is allocated less space, then frequent OVERFLOW conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.
- In case, we allocate a large amount of space for the queue, it will result in sheer wastage of memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size.
- While operating on these queues, one thing is important to note. While queue A will grow from left to right, the queue B on the same time will grow from right to left.