Unit – 2

Stack and Queue

Content

Stack: Introduction, Array Representation of Stacks, Operations and Implementation, Applications of Stacks-Reversing list, Infix to Postfix Conversion, Evaluating Postfix Expressions.

Queues: Introduction, Array Representation of Queues, Operations and Implementation, Types of Queues: Circular Queues, Deques and Priority Queues, Application of Queues.

---

1. Stack Data Structure introduction

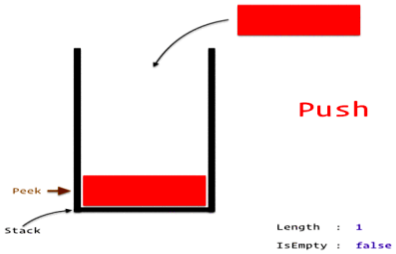
- ✓ A stack is a linear data structure.
- ✓ It uses the same principle, that the elements in a stack are added and removed only from one end, which is called the TOP.
- ✓ Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

2. Array representation of stacks

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

3. Operations on Stack

- ✓ A stack supports three basic operations:
  - ➢ Push
  - ➢ Pop
  - ➢ Peek.
- ✓ The push operation adds an element to the top of the stack.
- ✓ The pop operation removes the element from the top of the stack.
- ✓ The peek operation returns the value of the topmost element of the stack.



a. Push Operation

- ✓ The push operation is used to insert an element into the stack.
- ✓ The new element is added at the topmost position (TOP) of the stack.
- ✓ Before inserting the value, we must check if TOP = MAX–1, because if the stack is full and no more insertions can be done.
- ✓ If an attempt is made to insert a value in a stack that is already full, an *overflow* message is printed.

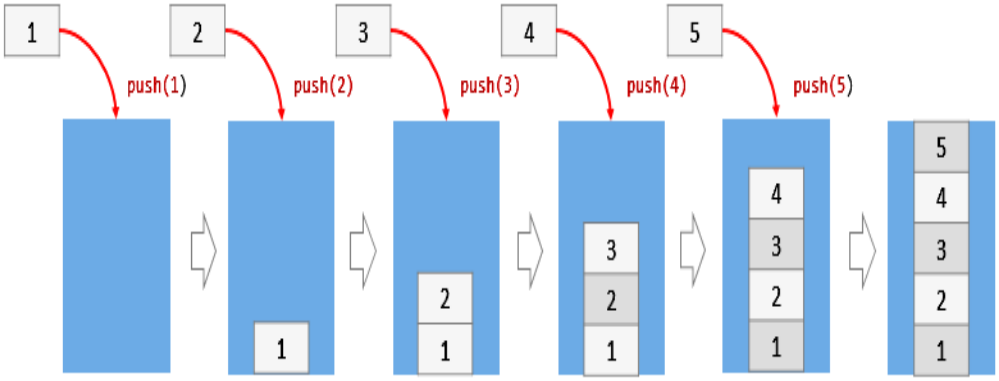
Push Operation Algorithm:

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

Step 1: Check for the OVERFLOW condition.

Step 2: TOP is incremented so that it points to the next location in the array.

Step 3: The value is stored in the stack at the location pointed by TOP.



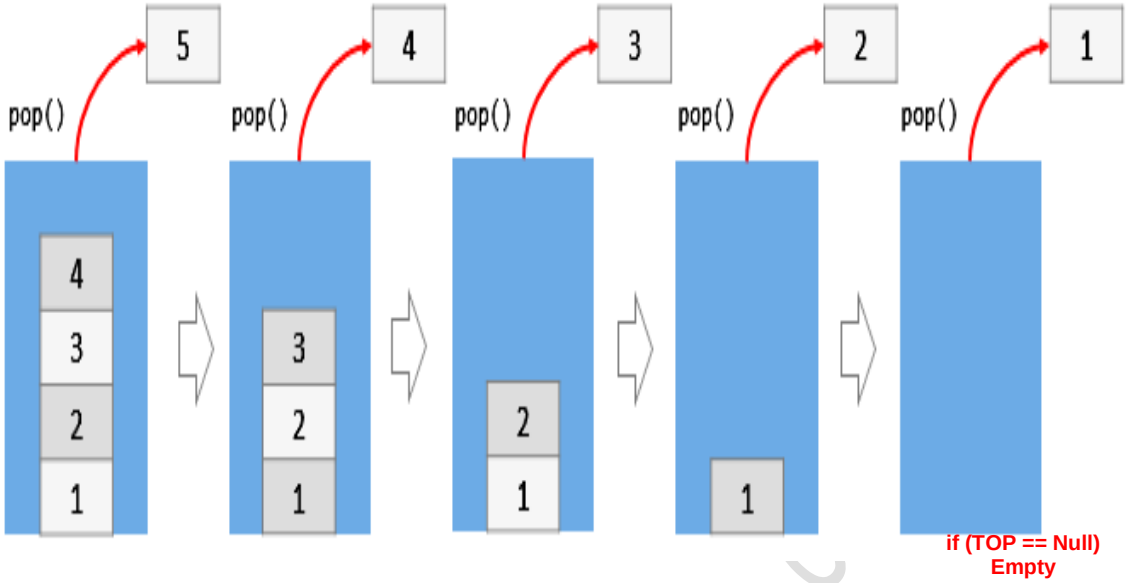
Pop Operation Algorithm:

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

Step 1: Check for the UNDERFLOW condition.

Step 2: The value of the location in the stack pointed by TOP is stored in VAL.

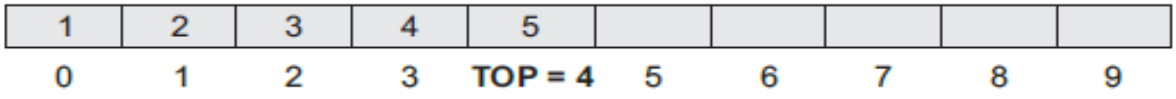
Step 3: TOP is decremented.



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 Peek operation first checks if the stack is empty.
- ✓ If TOP = NULL, then an appropriate message is printed, else the value is returned.

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

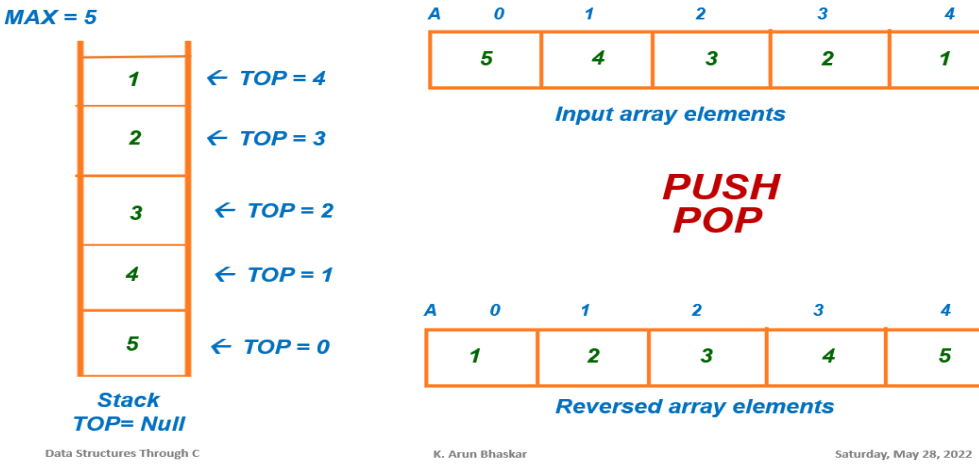


4. Applications of Stack

- ➢ *Reversing a list*
- ➢ *Parentheses checker*
- ➢ *Conversion of an infix expression into a postfix expression*
- ➢ *Evaluation of a postfix expression*
- ➢ *Conversion of an infix expression into a prefix expression*
- ➢ *Evaluation of a prefix expression*
- ➢ *Recursion*
- ➢ *Tower of Hanoi*

5. Reversing a List using a Stack:

- ✓ A list of numbers can be reversed by reading each number from an array from the first index and pushing it on a stack.
- ✓ Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.



Implementing Parentheses Checker using Stack

- ✓ Stack data structure can be used to check the validity of parentheses in any algebraic expression.
- ✓ For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.
- ✓ Example: The expression (A+B} is invalid but an expression {A + (B – C)} is valid.

6. Evaluation of Arithmetic Expressions (Infix to Postfix Conversion):

- ✓ Algebraic expressions can evaluated in three different ways:
  - ✓ Infix,
  - ✓ prefix, and
  - ✓ postfix notations.
- ✓ **Infix notation:** While writing 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 it is easy for us to write expressions using infix notation.
- ✓ **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 as AB+ in postfix notation.
- ✓ The order of evaluation of a postfix expression is always from *left to right*.
- ✓ Even brackets cannot alter the order of evaluation.
- ✓ **Prefix notation** : The operator is placed before the operands.

- ✓ For example: if an expression is written as A+B in infix notation, the same expression can be written as +AB in prefix notation.

Examples of Infix, Prefix, and Postfix:

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
(A + B) * C	* + A B C	A B + C *
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

6. Conversion of an Infix Expression into a Postfix Expression

- ✓ Let 'X' be an algebraic expression written in infix notation.
- ✓ The expression, may contain parentheses, operands, and operators.
- ✓ For simplicity of the algorithm, we will use only +, –, *, /, % operators.
- ✓ The precedence of these operators can be given as follows:
- ➢ Higher priority *, /, %
- ➢ Lower priority +, –
- ➢ **Example : 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 * –

Conversion of an Infix Expression into a Postfix Expression:

```
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 a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push 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
```

7. Evaluation of a Postfix Expression

Consider the infix expression: **9 – ((3 * 4) + 8) / 4.**

- ✓ Evaluate the expression.
- ✓ The above infix expression can be written in postfix notation:

9 3 4 * 8 + 4 / –

Evaluation of a postfix expression

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

7. Evaluation of a Postfix Expression

Algorithm :

```
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
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. 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
```

Conversion of an Infix Expression into a Prefix Expression

```
Step 1: Reverse the infix string. Note that
        while reversing the string you must
        interchange left and right parentheses.
Step 2: Obtain the postfix expression of the
        infix expression obtained in Step 1.
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 parentheses.*

(L – K / A) * (C / B – A)

Step 2: *Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.*

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**

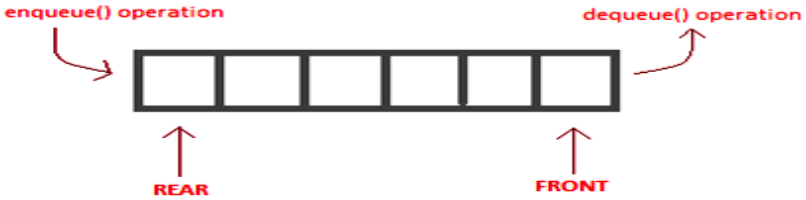
```
Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
        (a) Scan the prefix expression
            from right, one character at a
            time.
        (b) If the scanned character is an
            operand, push it on the
            operand stack.
        (c) If the scanned character is an
            operator, then
              (i) Pop two values from the
                  operand stack
             (ii) Apply the operator on
                  the popped operands
            (iii) Push the result on the
                  operand stack
Step 3: END
```

Example: Consider the prefix expression + – 2 7 * 8 / 4 12. Let us now apply the algorithm to evaluate this expression.

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
-	24, 5
+	29

Queue Data Structure:

- ✓ A *queue* is a linear data structure
- ✓ A *queue* is a FIFO (First-In, First-Out) data structure in which the element that is 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 end called the FRONT.
- ✓ Queues can be implemented by using either arrays or linked lists.



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

Array representation of Queue

- ✓ Queues can be easily represented using arrays.

- ✓ Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

Rear:

- ✓ Insert the element into queue from rear end.
- ✓ Before inserting an element in a queue, we must check for overflow conditions.
- ✓ An overflow will occur when we try to insert an element into a queue that is already full.
- ✓ When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition.
- ✓ Note that we have written MAX – 1 because the index starts from 0.

To insert an element in a queue:

- ✓ First check for the overflow condition.
- ✓ Check if the queue is empty.
- ✓ In case the queue is empty, then both FRONT and REAR are set to zero, so that the new value can be stored at the 0th location.
- ✓ Otherwise, if the queue already has some values, then REAR is incremented so that it points to the next location in the array.
- ✓ The value is stored in the queue at the location pointed by REAR.

Algorithm:

Step 1: IF REAR = MAX-1

Write OVERFLOW

Goto step 4

[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE

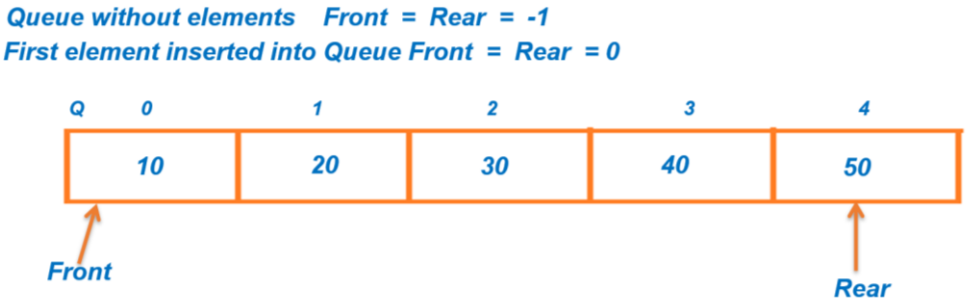
SET REAR = REAR + 1

[END OF IF]

Step 3: SET QUEUE[REAR] = NUM

Step 4: EXIT

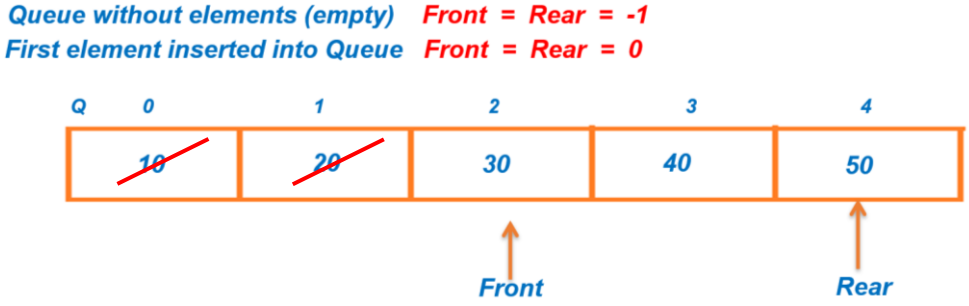
Operations on Queue Cont...



Front:

- ✓ Delete the element from the front of queue.
- ✓ Before deleting an element from a queue, we must check for underflow conditions.
- ✓ An underflow condition occurs when we try to delete an element from a queue that is already empty.
- ✓ If FRONT = –1 and REAR = –1, it means there is no element in the queue.

Array representation of Queue Cont...



Delete operation Process:

- ✓ To delete an element from a queue.
- ✓ In Step 1, we check for underflow condition.
- ✓ An underflow occurs if FRONT = –1 or FRONT > REAR.
- ✓ However, if queue as some values, then FRONT is incremented so that it now points to the next value in the queue.

Algorithm:

Step 1: IF FRONT = -1 OR FRONT > REAR

Write UNDERFLOW

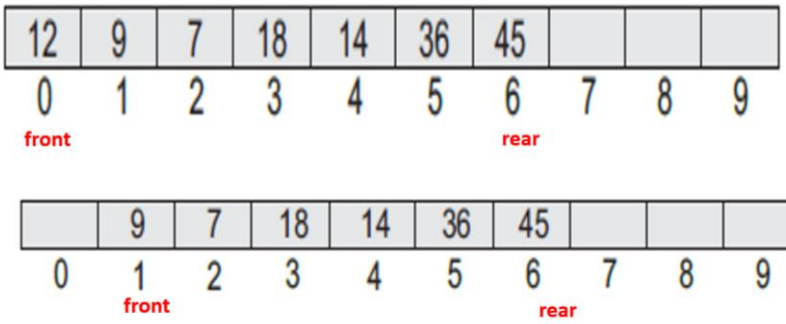
ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

Step 2: EXIT



Types of queues

A queue data structure can be classified into the following types:

1. Simple Queue
2. Circular Queue
3. Double-Ended Queue (Deque)
4. Priority Queue
5. Multiple Queue

Drawback of simple Queues:

In linear queues, insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.

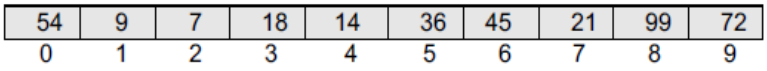


Figure Linear queue

Here, FRONT = 0 and REAR = 9.

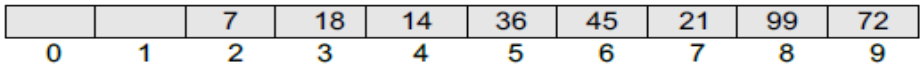
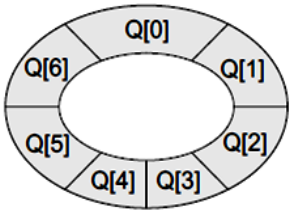


Figure Queue after two successive deletions

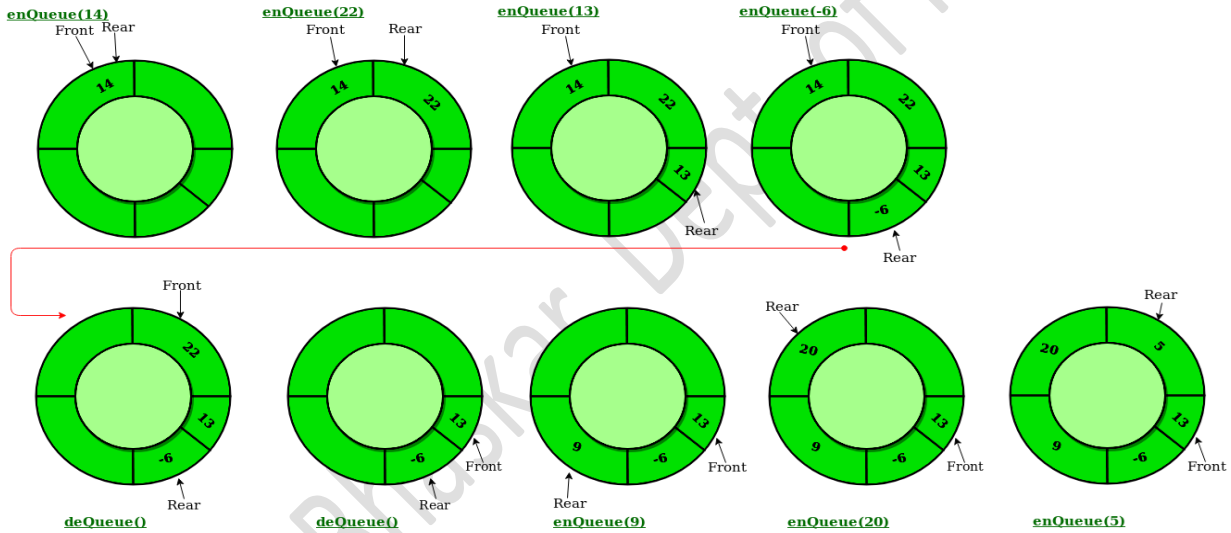
- ✓ Suppose we want to insert a new element in the queue:
- ✓ Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true.
- ✓ This is a major drawback of a linear queue.
- ✓ Two solutions are there to resolve this problem.
- ✓ First, 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 the circular queue, the first index comes right after the last index.

A. Circular Queues Introduction:

- ✓ Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- ✓ It is also called 'Ring Buffer'.
- ✓ In a normal Queue, we can insert elements until queue becomes full. The circular queue will be full only when front = 0 and rear = Max – 1.
- ✓ A circular queue is implemented in the same manner as a linear queue is implemented.



Circular Queues Introduction:



Step 1: First check for the overflow condition.

Step 2: We must check for two conditions:

- ➢ First to see if the queue is empty, and
- ➢ Second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end.

Step 3: The value is stored in the queue at the location pointed by REAR.

Algorithm:

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Delete an element from a circular queue:

Step 1: Check for the underflow condition.

Step 2: The value of the queue at the location pointed by FRONT is stored in VAL.

Step 3: Make two checks.

- First to see if the queue has become empty after deletion.
- Second to see if FRONT has reached the maximum capacity of the queue.
- The value of FRONT is then updated based on the outcome of these checks.

Algorithm:

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [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
```

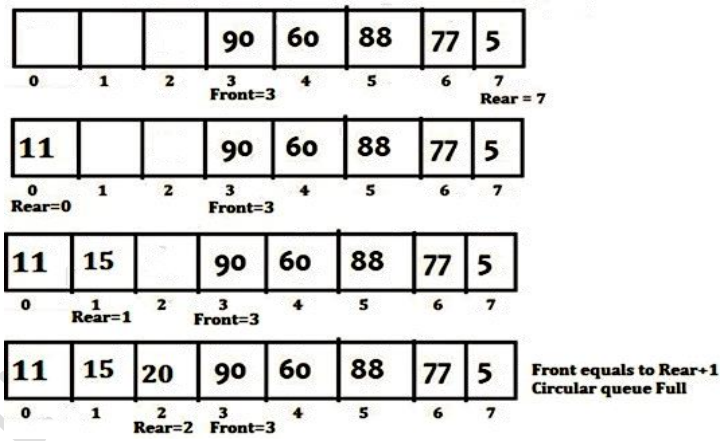


Figure 8.20 Empty queue

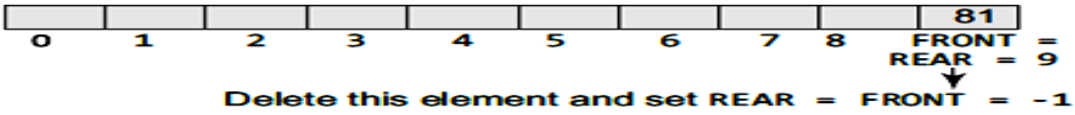


Figure 8.21 Queue with a single element

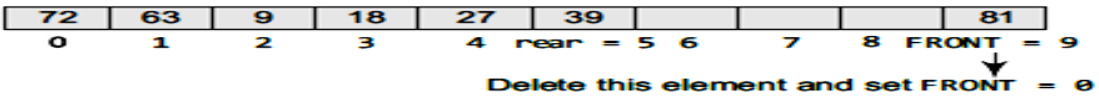


Figure 8.22 Queue where FRONT = MAX–1 before deletion

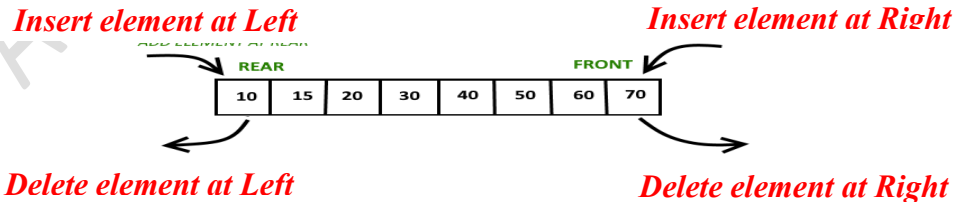
B. Deques Introduction

- ✓ A deque (pronounced as 'deck' or 'dequeue') is a list in which the 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 either the front (head) or the back (tail) end.
- ✓ However, no element can be added and deleted from the middle.
- ✓ A deque is implemented using either a circular array or a circular doubly linked list.
- ✓ In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque.
- ✓ The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0].
- ✓ There are two variants of a double-ended queue. They include :
- ✓ *Input restricted deque:* In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.

- ✓ *Output restricted deque :* In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

Deque operations:

1. *Insert element at right*
2. *Insert element at left*
3. *Delete / Remove element at right*
4. *Delete element at left.*



Algorithm to Insert an element at left:

Step 1: Start

Step 2: Check the queue is full, as if (RIGHT== max-1) &&(LEFT==0)

Step 3: If false update the pointer LEFT as LEFT=LEFT-1

Step 4: Insert the element at pointer LEFT as Q[LEFT] = element

Step 5: Stop

Algorithm to Insert an element at Right:

Step 1: Start

Step 2: Check the queue is full, as if (RIGHT == max-1) &&(LEFT==0)

Step 3: If false update the pointer RIGHT as RIGHT = RIGHT +1

Step 4: Insert the element at pointer RIGHT as Q[RIGHT] = element

Step 5: Stop

Algorithm to Delete an element at Left:

Step 1: Start

Step 2: Check the queue is empty, as if (RIGHT==LEFT)

Step 3: If false update the pointer LEFT as LEFT = LEFT +1 and delete element at LEFT as element Q[LEFT].

Step 4: If (LEFT==RIGHT)reset point LEFT and RIGHT as LEFT=RIGHT= -1.

Step 5: Stop.

Algorithm to Delete an element at Right:

Step 1: Start

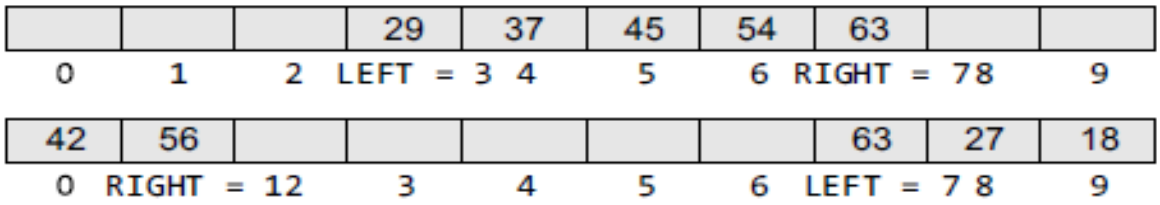
Step 2: Check the queue is empty, as if (RIGHT==LEFT)

Step 3: If false delete element at position RIGHT as element = Q[RIGHT].

Step 4: Update pointer RIGHT as RIGHT = RIGHT -1

Step 5: If (LEFT == RIGHT) reset pointer f and r as LEFT = RIGHT = -1.

Step 6: Stop



Applications of Deque:

1. Palindrome-checker
2. Steal job scheduling algorithm
3. Undo-Redo operations in Software applications.

C. Priority Queue Introduction

- ✓ Priority Queue is more specialized data structure than Queue.
- ✓ Like ordinary queue, priority queue has same method but with a major difference.
- ✓ In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear.
- ✓ So, we're assigned priority to item based on its key value.
- ✓ Lower the value, higher the priority.
- ✓ Priority Queue is an abstract datatype in which element is assign a priority.
- ✓ The general rule of processing the elements of a priority queue is:
  - ✓ *An element with higher priority is processed before an element with a lower priority.*
  - ✓ *Two elements are with same priority are processed on a First-Come First-Served (FCFS) basis.*

Priority Queue operations:

- ✓ Enqueue
- ✓ Dequeue

Applications of Queues:

- ✓ Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- ✓ Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- ✓ Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- ✓ Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- ✓ Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.