UNIT II

What is a Stack?

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.

It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Basic features of Stack

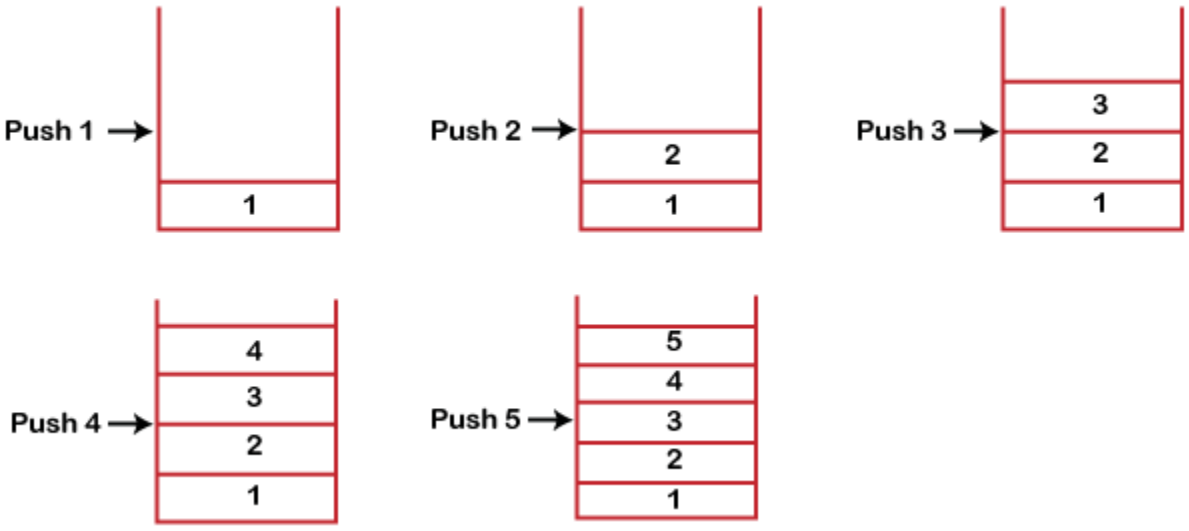
1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).

3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

The following are some common operations implemented on the stack:

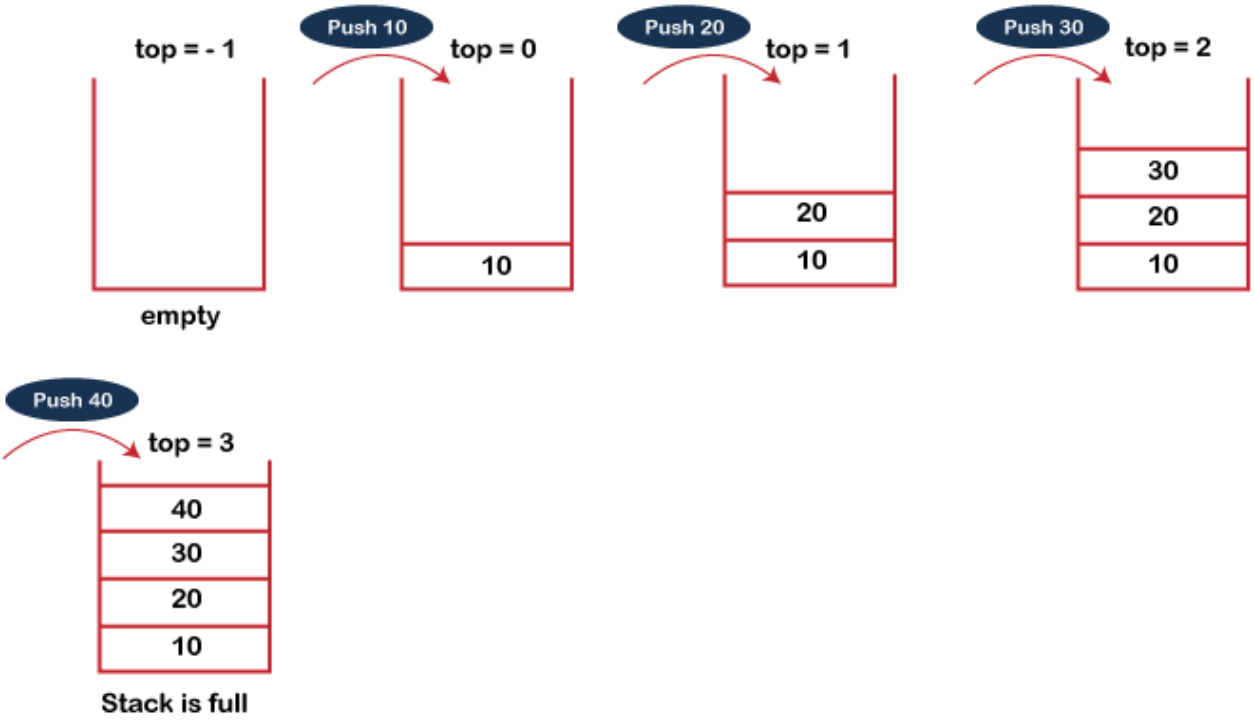
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.

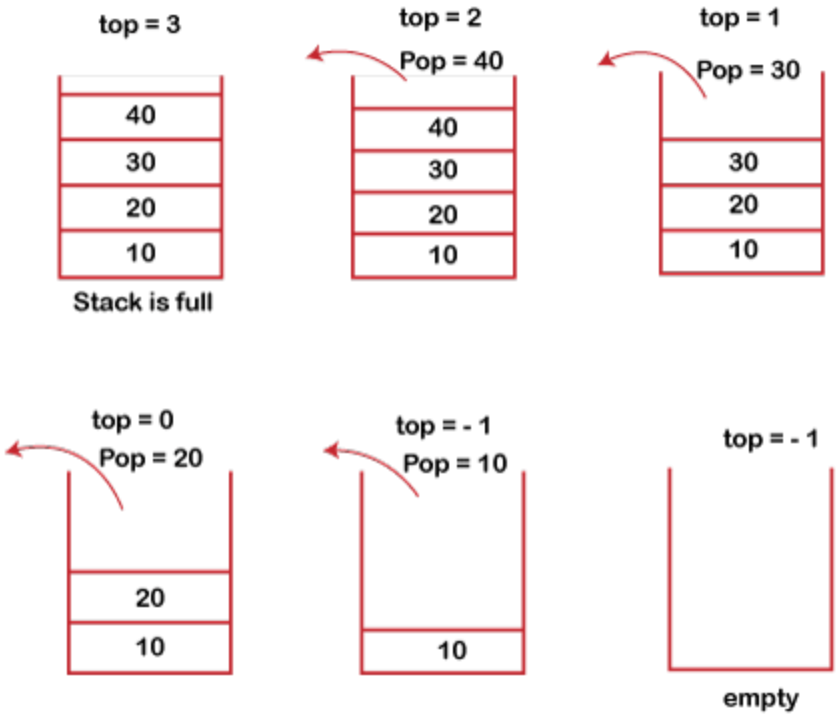
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

begin

1. if top = n then stack full
2. top = top + 1
3. stack (top) : = item;
4. end

implementation of push algorithm in C language

```
1. void push (int val,int n) //n is size of the stack
2. {
3.    if (top == n )
4.    printf("\n Overflow");
5.    else
6.    {
7.    top = top +1;
8.    stack[top] = val;
9.    }
10. }
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack.

The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
1. begin
2.     if top = 0 then stack empty;
3.     item := stack(top);
4.     top = top - 1;
5. end;
```

Implementation of POP algorithm using C language

```
1. int pop ()
2. {
3.     if(top == -1)
4.     {
5.         printf("Underflow");
6.         return 0;
7.     }
8.     else
9.     {
10.        return stack[top - - ];
11.    }
12. }
```

Queue

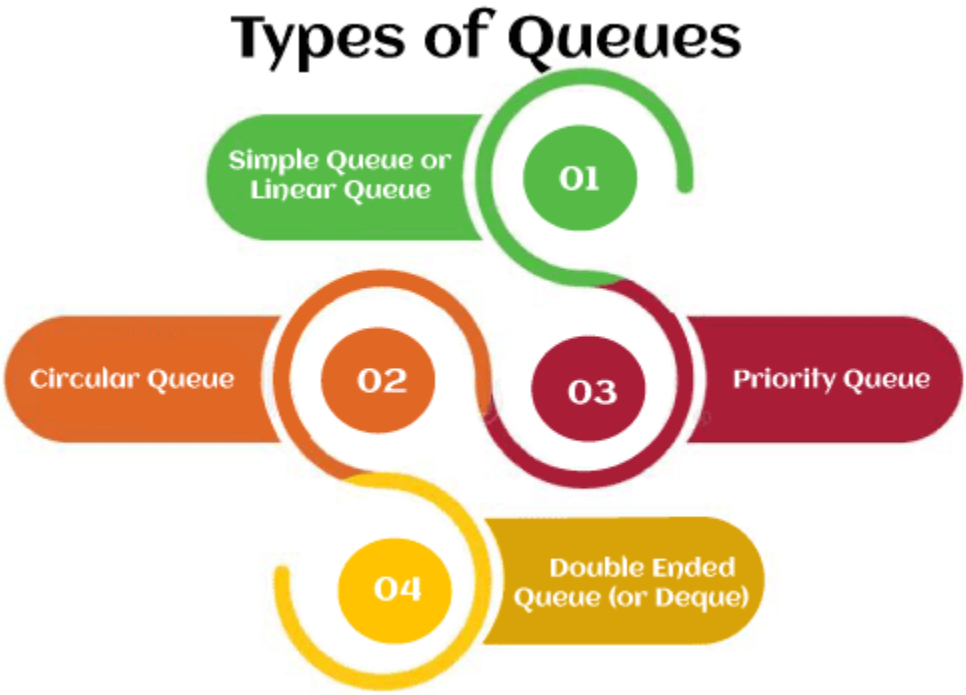
1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.

Types of Queue

There are four different types of queue that are listed as follows -



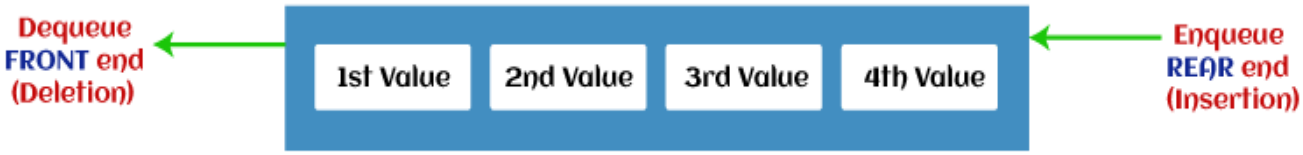
- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and

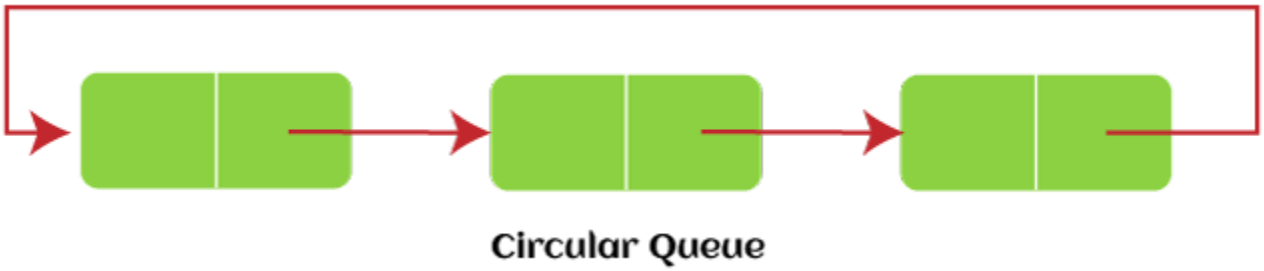
the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Circular Queue

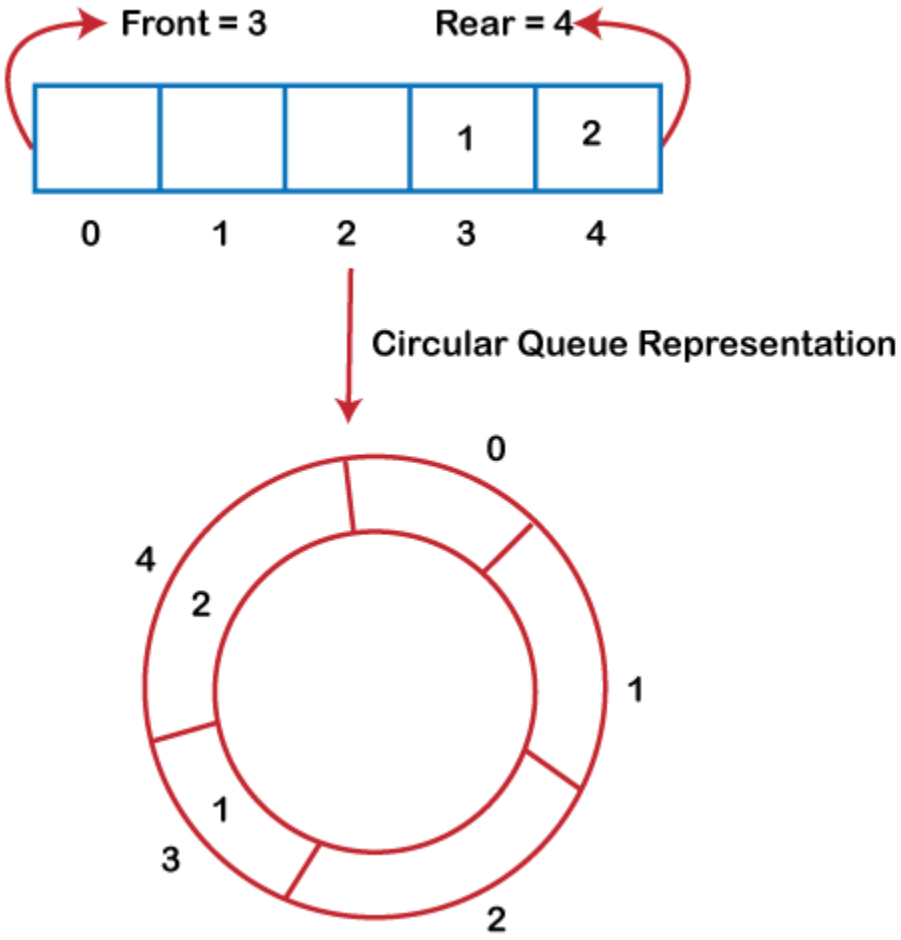
In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Why was the concept of the circular queue introduced?

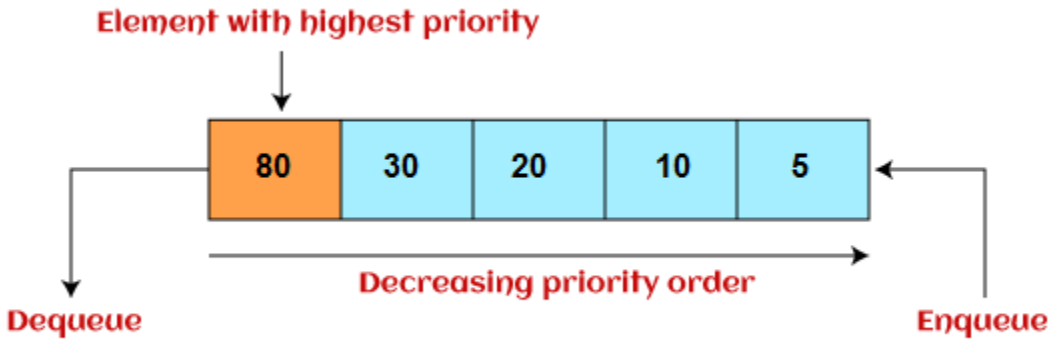
There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0^{th} position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

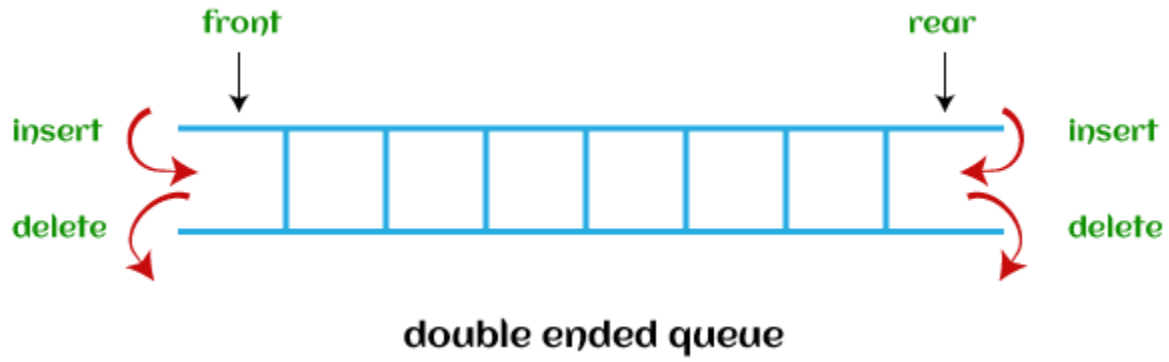
- **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

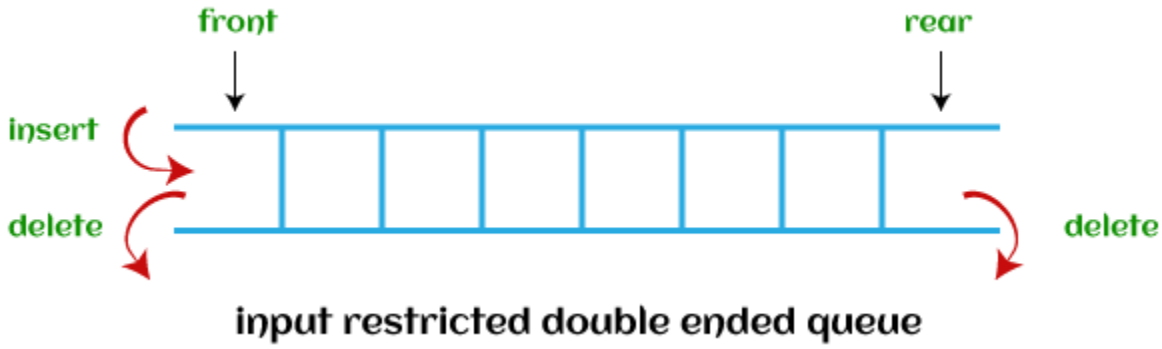
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -

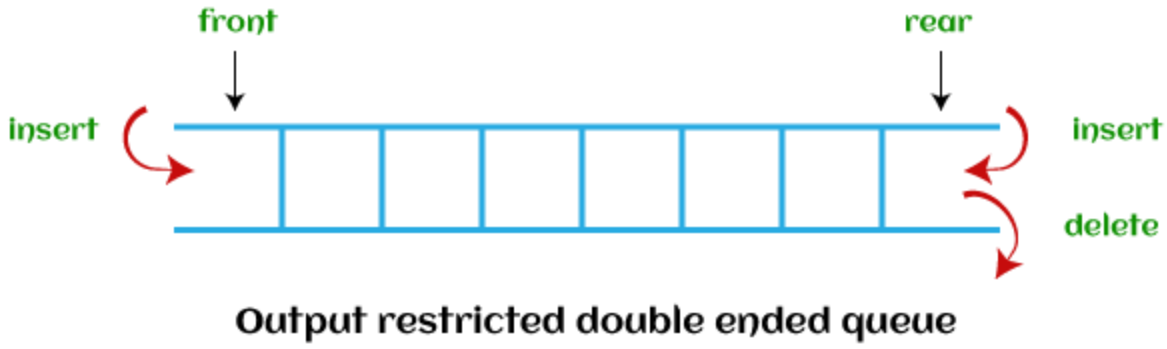


There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Now, let's see the operations performed on the queue.

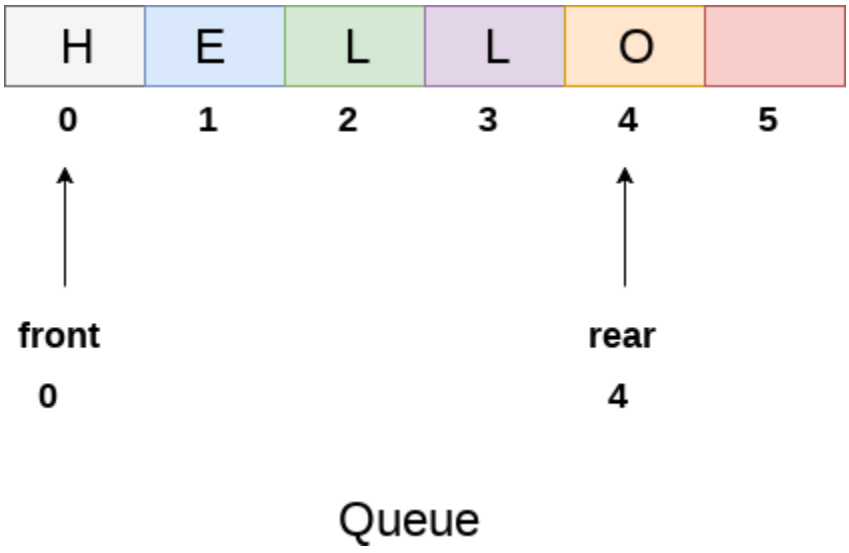
Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

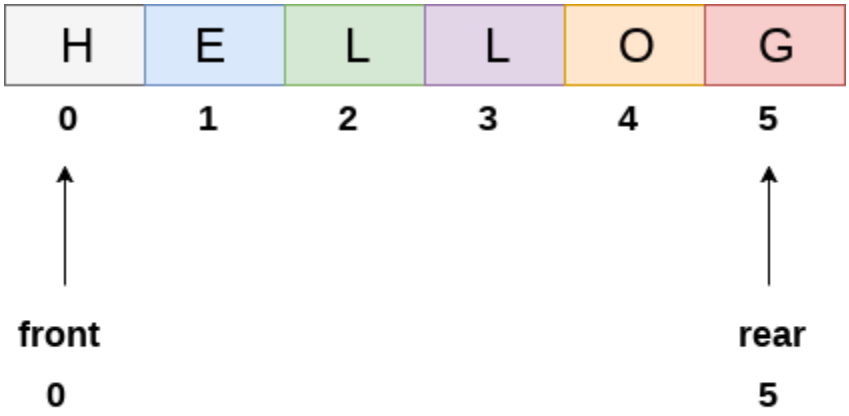
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



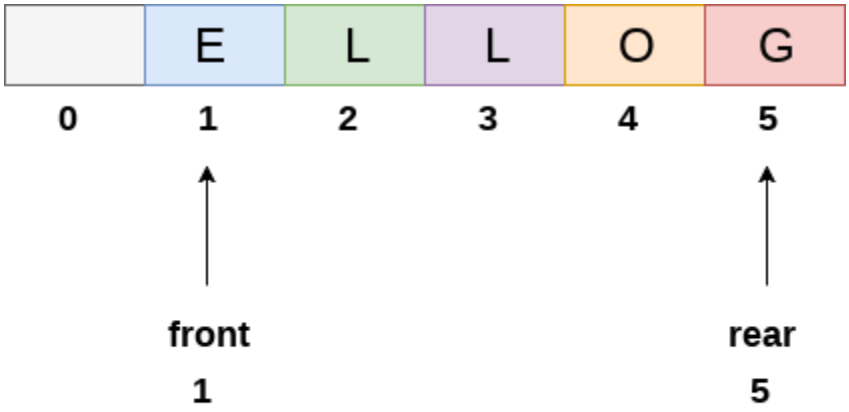
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

C Function

```
1.  void insert (int queue[], int max, int front, int rear, int item)
2.  {
3.      if (rear + 1 == max)
4.      {
5.          printf("overflow");
6.      }
7.      else
8.      {
9.          if(front == -1 && rear == -1)
10.         {
11.             front = 0;
12.             rear = 0;
13.         }
14.         else
15.         {
16.             rear = rear + 1;
17.         }
18.         queue[rear]=item;
19.     }
20. }
```

Algorithm to delete an element from the queue

C Function

```
1.  int delete (int queue[], int max, int front, int rear)
2.  {
3.      int y;
4.      if (front == -1 || front > rear)
5.
6.      {
7.          printf("underflow");
8.      }
9.      else
10.     {
11.         y = queue[front];
12.         if(front == rear)
13.         {
14.             front = rear = -1;
15.             else
16.             front = front + 1;
17.
18.         }
19.         return y;
20.     }
21. }
```

POLISH EXPRESSSIONS

Polish Notation in the data structure is a method of expressing mathematical, logical, and algebraic equations universally. This notation is used by the compiler to evaluate mathematical equations based on their order of operations. When parsing mathematical expressions, three types of notations are commonly used : **Infix** Notation, **Prefix** Notation, and **Postfix** Notation.

Introduction to Polish Notation in Data Structure

This type of notation was introduced by the **Polish mathematician Lukasiewicz**. Polish Notation in data structure tells us about different ways to write an arithmetic expression. An arithmetic expression contains 2 things, i.e., **operands** and **operators**. Operands are either numbers or variables that can be replaced by numbers to evaluate the expressions. Operators are symbols symbolizing the operation to be performed between operands present in the expression. Like the expression (1+2) * (3+4)(1+2)∗(3+4) standard becomes * +12 +34∗+12+34 in Polish Notation. Polish notation is also called **prefix notation**. It means that operations are written before the operands. The operators are placed left for every pair of operands. Let's say for the expression a+b, the prefix notation would be +ab.

Types of Notations

Infix Notation

The operator symbol is placed between its two operands in most arithmetic operations. This is called infix expression. For example,

(**A** + **B**); here the operator + is placed between the two operands a and b.

Although we find it simple to write expressions in infix notation, computers find it difficult to parse. So, the computer normally evaluates arithmetic expressions written in infix notation after converting them into postfix notation. The stack is the primary tool used to complete the given task in each phase.

Prefix Notation (Polish Notation)

Computers perform better when expressions are written in prefix and postfix notations.

Prefix notation refers to the notation in which the operator is placed before its two operands. For example, if

```
A + B
```

is an expression in infix notation, then

```
+AB
```

is the equivalent expression in prefix notation.

Postfix Notation (Reverse Polish Notation)

In postfix notation, the operator is placed after the operands. For example, if an expression is written in infix notation as

```
A + B
```

it can be written in postfix notation as

```
AB+
```

The evaluation of a postfix and prefix expressions are always performed from left to right.

There are 3 levels of precedence for 5 binary operators as given below:

```
Highest: Exponentiation (^)

Next highest: Multiplication (*) and division (/)

Lowest: Addition (+) and Subtraction (-)
```

For example –

```
Infix notation: (A-B)*[C/(D+E)+F]

Post-fix notation: AB- CDE +/F +*
```

Here, we first perform the arithmetic inside the parentheses (A-B) and (D+E). The division of C/(D+E) must be done prior to the addition with F. After that multiply the two terms inside the parentheses and bracket.

Now we need to calculate the value of these arithmetic operations by using a stack.

The procedure for getting the result is:

1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Push the operands into the stack in the order they appear.
3. When any operator encounters then pop two topmost operands for executing the operation.
4. After execution push the result obtained into the stack.
5. After the complete execution of expression, the final result remains on the top of the stack.

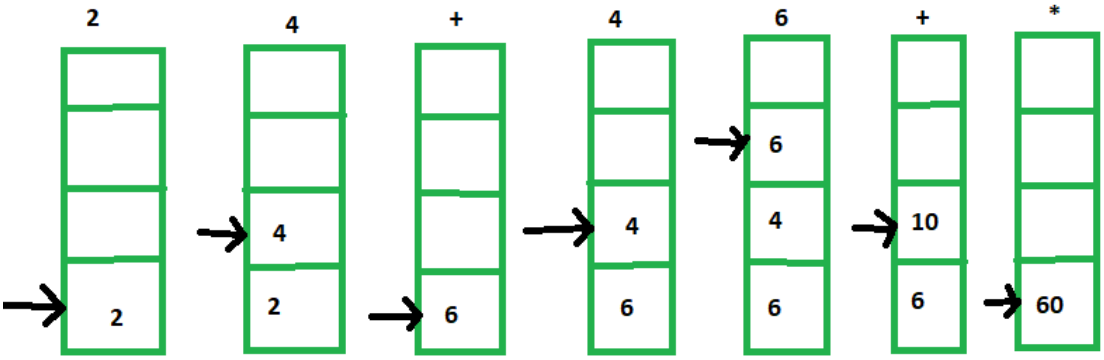
For example –

```
Infix notation: (2+4) * (4+6)

Post-fix notation: 2 4 + 4 6 + *

Result: 60
```

The stack operations for this expression evaluation is shown below:



Stack operations to evaluate (2+4)*(4+6)

Question 4: Evaluate the prefix expression or polish expression × 5 – 4 3 using stack?

Solution: Inserting the symbol # at the left end of the expression. Now the expression becomes # × 5 – 4 3.

Prefix Expression	Symbol Scanned	Stack
# × 5 – 4 3	None	Empty
# × 5 – 4	3	3
# × 5 –	4	3, 4
# × 5	–	4 – 3 = 1
# ×	5	1, 5
#	×	5 × 1 = 5
	#	Result = 5

Question 5: Evaluate the Polish expression -+-x43250/x784 using stack?

Solution: Put symbol # at the left end of the expression.

Polish Expression: # – + – × 4 3 2 50 / × 7 8 4

Prefix expression	Symbol scanned	Stack
# – + – × 4 3 2 50 / × 7 8 4	None	Empty
# – + – × 4 3 2 50 / × 7 8	4	4

# – + – × 4 3 2 50 / × 7	8	4, 8
# – + – × 4 3 2 50 / ×	7	4, 8, 7
# – + – × 4 3 2 50 /	×	4, 56 (8 × 7 = 56)
# – + – × 4 3 2 50	/	14 (56 / 4 = 14)
# – + – × 4 3 2	50	14, 50
# – + – × 4 3	2	14, 50, 2
# – + – × 4	3	14, 50, 2, 3
# – + – ×	4	14, 50, 2, 3, 4
# – + –	×	14, 50, 2, 12 (4 × 3 = 12)
# – +	–	14, 50, 10 (12 – 2 = 10)
# –	+	14, 60 (50 + 10 = 60)
#	–	46 (60 – 14 = 46)
	#	Result = 46

Reverse Polish Notation

Reverse polish notation is also known as postfix notation is defined as: In postfix notation operator is written after the operands. Examples of postfix notation are AB + and CD –. Here A and B are two operands and the operator is written after these two operands. The conversion from infix expression into postfix expression is shown below.

Reverse Polish Notation Algorithm

Now we will see how to evaluate a Reverse Polish Expression or Postfix Expression using stack. The algorithm for evaluation of it is given below.

Step 1: Insert a symbol (say #) at the right end of the postfix expression.

Step 2: Scan the expression from left to right and follow the Step 3 and Step 4 for each of the symbol encountered.

Step 3: if an element is encountered insert it into the stack.

Step 4: if an operator (say &) is encountered pop the top element A (say) and next to top element B (say) perform the following operation x = B & A. Push x into the top of the stack.

Step 5: if the symbol # is encountered then stop scanning.

Question 8: Evaluate the post fix expression or reverse polish expression 50 4 3 × 2 – + 7 8 × 4 / – using stack?

Solution: Put symbol # at the right end of the expression.

Postfix Expression: 50 4 3 × 2 – + 7 8 × 4 / – #

Postfix expression	Symbol scanned	Stack
50 4 3 × 2 – + 7 8 × 4 / – #	None	Empty
4 3 × 2 – + 7 8 × 4 / – #	50	50
3 × 2 – + 7 8 × 4 / – #	4	50, 4
× 2 – + 7 8 × 4 / – #	3	50, 4, 3
2 – + 7 8 × 4 / – #	×	50, 12 (4 × 3 = 12)
– + 7 8 × 4 / – #	2	50, 12, 2

+ 7 8 × 4 / – #	–	50, 10 (12 – 2 = 10)
7 8 × 4 / – #	+	60 (50 + 10 = 60)
8 × 4 / – #	7	60, 7
× 4 / – #	8	60, 7, 8
4 / – #	×	60, 56 (7 × 8 = 56)
/ – #	4	60, 56, 4
– #	/	60, 14 (56 / 4 = 14)
#	–	46 (60 – 14 = 46)
_	#	Result = 46

Dynamic Memory Management in C

When a C program is compiled, the compiler allocates memory to store different data elements such as constants, variables (including pointer variables), arrays and structures. This is referred to as **compile-time or static memory allocation.** There are several limitations in such static memory allocation:

1. This allocation is done in memory exclusively allocated to a program, which is often limited in size.

2. A static array has fixed size. We cannot increase its size to handle situations requiring more elements. As a result, we will tend to declare larger arrays than required, leading to wastage of memory. Also, when fewer array elements are required, we cannot reduce array size to save memory.

3. It is not possible (or efficient) to create advanced data structures such as linked lists, trees and graphs, which are essential in most real-life programming situations.

The C language provides a very simple solution to overcome these limitations: **dynamic memory allocation** in which the memory is allocated at run-time, i. e., during the execution of a program. Dynamic memory management involves the use of pointers and four standard library functions, namely, malloc, calloc, realloc and free. The first three functions are used to allocate memory, whereas the last function is used to return memory to the system (also called freeing/deallocating memory). The pointers are used to point to the blocks of memory allocated dynamically.

Standard library functions for dynamic memory management

Recall that the C language provides four functions for dynamic memory management, namely, malloc, calloc, realloc and free. These functions are declared in stdli.b .h header file. They are summarized in Table and are described below.

Function	Typical call	Description
malloc	malloc (sz)	Allocate a block of size *sz* bytes from memory heap and return a pointer to the allocated block
calloc	calloc *in (sz)*	Allocate a block of size *n* x *sz* bytes from memory heap, initialize it to zero and return a pointer to the allocated block
realloc	realloc *(bl,; sz)*	Adjust the size of the memory block *blk* allocated on the heap to *sz*, copy the contents to a new location if necessary and return a pointer to the allocated block
free	free *(blk)*	Free block of memory *blk* allocated from memory heap

The malloc, calloc and realloc functions allocate a contiguous block of memory from *heap.* If memory allocation is successful, they return a pointer to allocated block (i. e., starting address of the block) as a void (i. e., a typeless) pointer; otherwise, they return a NULL pointer.

The malloc function

The malloc (memory *allocate)* function is used to allocate a contiguous block of memory from heap. If the memory allocation is successful, it returns a pointer to the allocated block as a void pointer; otherwise, it returns a NULL pointer. The prototype of this function is given below.

```
1    void *malloc(size_t size);
```

The malloc function has only one argument, the *size* (in bytes) of the memory block to be allocated. Note that size_t is a type, also defined in stdlib.h, which is used to declare the sizes of memory objects and repeat counts. We should be careful while using the malloc function as the allocated memory block is uninitialized, i. e., it contains garbage values.

As different C implementations may have differences in data type sizes, it is a good idea to use the sizeof operator to determine the size of the desired data type and hence the size of the memory block to be allocated. Further

The calloc function

The calloc function is similar to malloc. However, it has two parameters that specify the number of items to be allocated and the size of each item as shown below.

```
1    void *callee ( size_t n_items, size_t size ) ;
```

Another difference is that the calloc initializes the allocated memory block to zero. This is useful in several situations where we require arrays initialized to zero. It is also useful while allocating a pointer array to ensure that the pointers will not have garbage values.

The realloc function

The realloc (reallocate) function is used to adjust the size of a dynamically allocated memory block. If required, the block is reallocated to a new location and the existing contents are copied to it. Its prototype is given below.

```
1    void *realloc( void *block; size_t size);
```

Here, *block* points to a memory block already allocated using one of the memory allocation functions (malloc, calloc or realloc). If successful, this function returns the address of the reallocated block; or NULL otherwise.

The free function

The free function is used to deallocate a memory block allocated using one of the memory allocation functions (malloc, calloc or realloc). The deallocation actually returns that memory block to the system heap. The prototype of free function is given below.

```
void free (void *block);
```

when a dynamically allocated block is no longer required in the program, we must return its memory to the system.