UNIT-4(STACKS,QUEUES AND RECURSSION)

Stacks:→ A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end. In stack data items are added or removed only at one end, called the top of the stack.

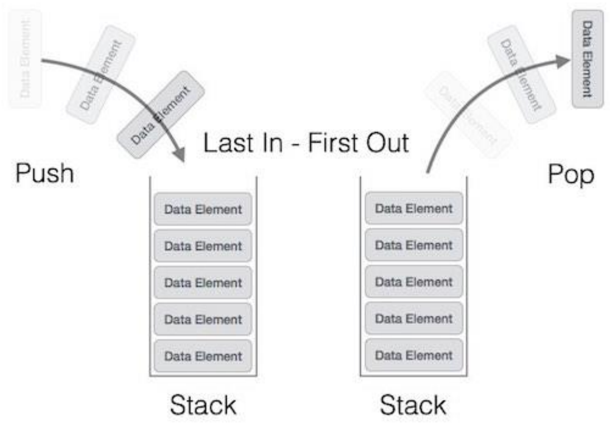
In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Some key points related to stack

- Stack is an ordered list of similar data type.
- It is called as stack because it behaves like a real-world stack, piles of books, stack of boxes etc.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.
- 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.

Stack Memory Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays.

An array is used to store an ordered list of elements. Using an array for representation of stack is one of the easy techniques to manage the data. But there is a major difference between an array and a stack.

- Size of an array is fixed.
- While, in a stack, there is no fixed size since the size of stack changed with the number of elements inserted or deleted to and from it.

Despite the difference, an array can be used to represent a stack by taking an array of maximum size; big enough to manage a stack.

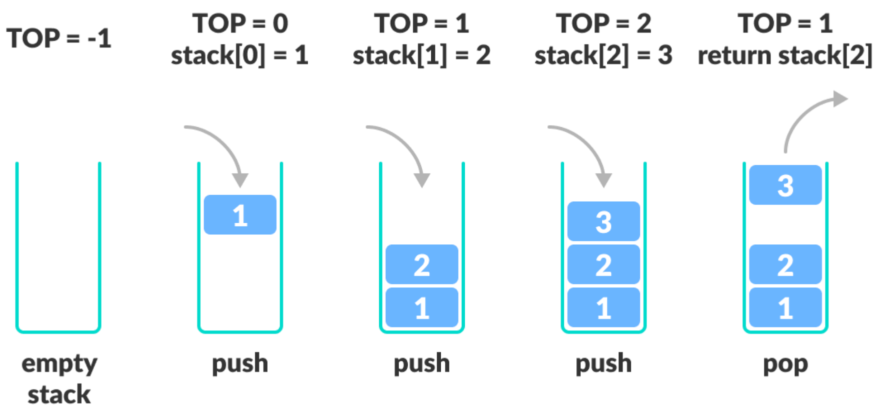
Standard Stack Operations

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():** get the top data element of the stack, without removing it
- **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.

Working of Stack Data Structure

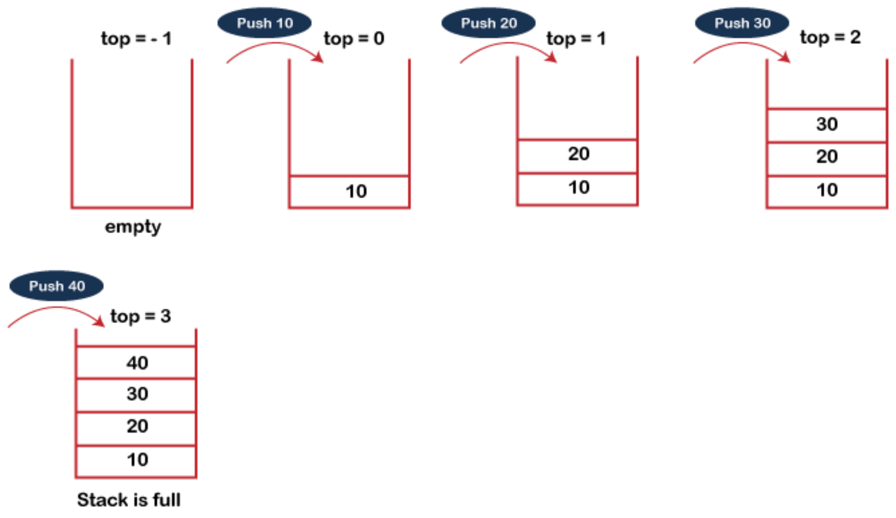
- A pointer called `TOP` is used to keep track of the top element in the stack.
- When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
- On pushing an element, we increase the value of `TOP` and place the new element in the position pointed to by `TOP`.
- On popping an element, we return the element pointed to by `TOP` and reduce its value.
- Before pushing, we check if the stack is already full
- Before popping, we check if the stack is already empty



PUSH operation

The steps involved in the PUSH operation is given below:

- Check if the stack is **full** or not.
- If the stack is full, then print error of overflow and exit the program.
- If the stack is not full, then increment the top and add the element.

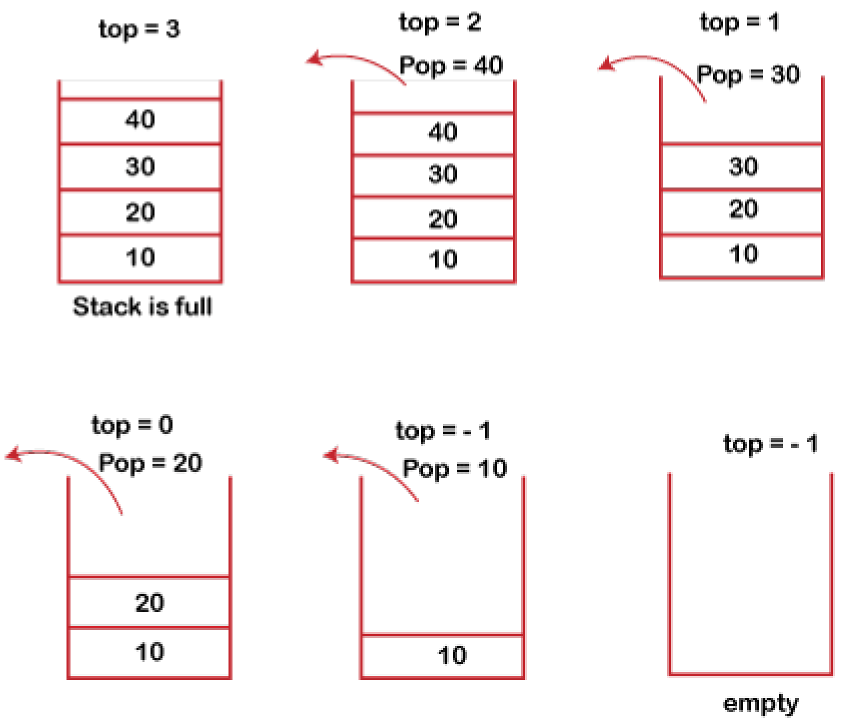


POP operation

The steps involved in the POP operation is given below:

1. Check if the stack is empty or not.

2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.



Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

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:

```
1.  start
2.     if top = n then stack full overflow condition;
3.     top = top + 1
4.     stack (top) : = item;
5.  end
```

Time Complexity : o(1)

implementation of push algorithm in C language

```
void push (int val,int n) //n is size of the stack
{
   if (top == n )
   printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
   }
}
```

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

Time Complexity : o(1)

Implementation of POP algorithm using C language

```c
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

```
1. Begin
2.     if top = -1 then stack empty
3.     item = stack[top]
4.     return item
5. End
```

Time complexity: o(n)

Implementation of Peek algorithm in C language

```c
int peek()
{
    if (top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack [top];
    }
}
```

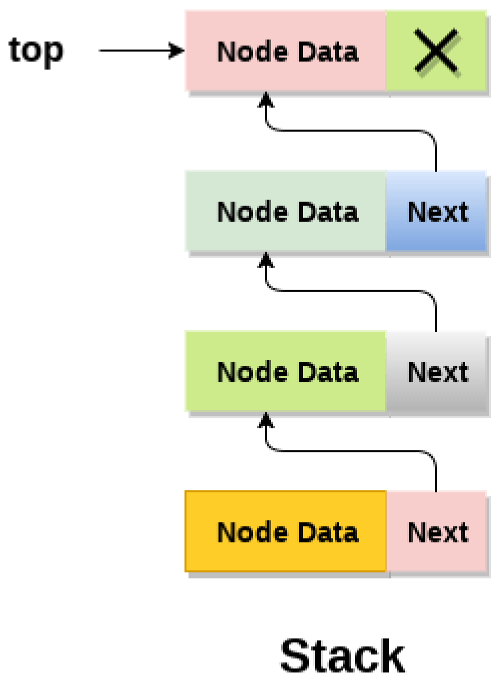
Program showing the use of push,pop and show function:→

```c
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");

printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
        };
    }
}

void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

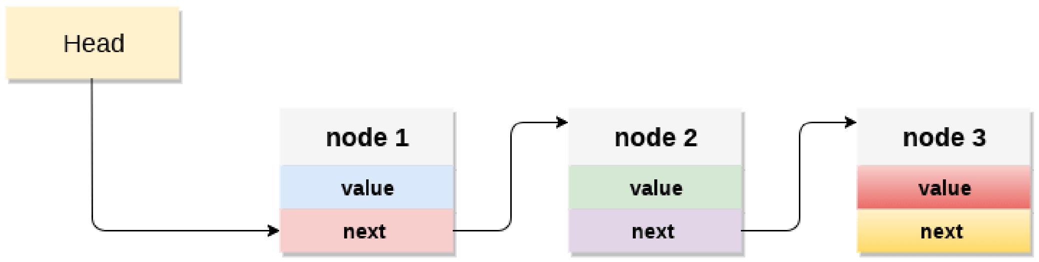
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

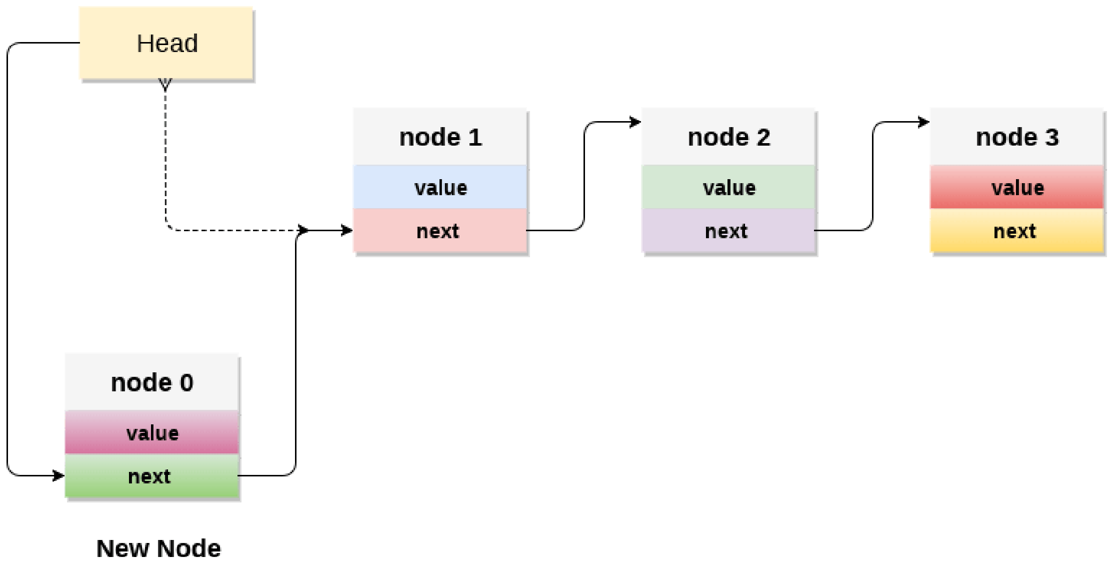
1. Create a node first and allocate memory to it.

2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)





Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(1)

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

Applications of Stack

The following are the applications of the stack:

Expression Evaluation:→evaluation of postfix expression

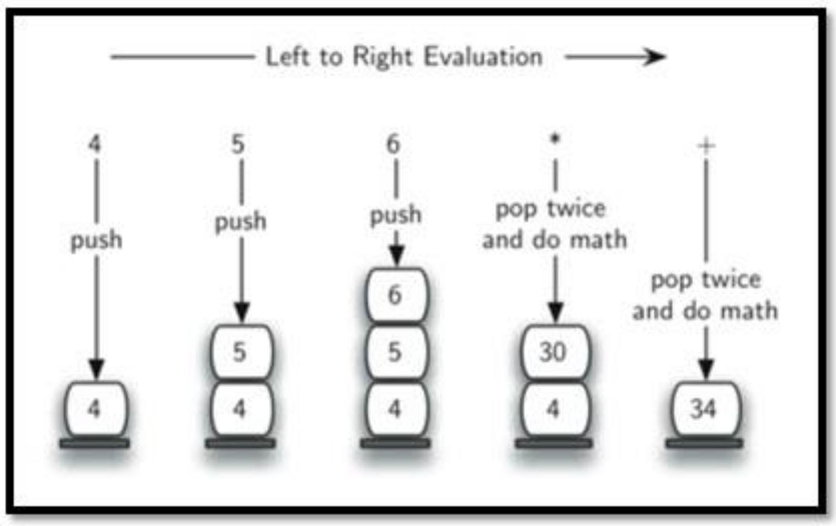
Algorithm

1) Add `(` to postfix expression.
2) Read postfix expression Left to Right until `)` encountered
3) If operand is encountered, push it onto Stack
[End If]
4) If operator is encountered, Pop two elements
i) `A -> Top` element
ii) `B-> Next to Top` element
iii) Evaluate `B` operator `A`
push `B` operator `A` onto Stack

5) Set `result = pop`
6) END

Let's see an example to better understand the algorithm:

Expression: 456*+



Que:1)P:→12,7,3,-,/,2,1,5,+,*,+

Solution: Scan p from left to right

Symbol	**stack**
12	**12**
7	**12,7**
3	**12,7,3**
-	**12,4**
/	**3**
2	**3,2**
1	**3,2,1**
5	**3,2,1,5**
+	**3,2,6**
*****	**3,12**
+	**15**

DFS(Depth First Search): This search is implemented on a Graph, and Graph uses the stack data structure.

Backtracking: Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

Memory management: The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

UNDO/REDO: It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

String reversal: Stack is also used for reversing a string. For example, we want to reverse a "**hello**" string, so we can achieve this with the help of a stack.
First, we push all the characters of the string in a stack until we reach the null character.
After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

Balancing of symbols: Stack is used for balancing a symbol. One of the most important applications of stacks is to check if the parentheses are balanced in a given expression. The compiler generates an error if the parentheses are not matched.

Here are some of the balanced and unbalanced expressions:

BALANCED EXPRESSION	UNBALANCED EXPRESSION
(a + b)	(a + b
[(c – d) * e]	[(c – d * e]
{ ( ) } [ ]	{ [( ]) }

Expression Representation and Conversion

There are three popular methods used for representation of an expression:

Infix	A + B	Operator between operands.
Prefix	+ AB	Operator before operands.
Postfix	AB +	Operator after operands.

1. Conversion of Infix to Postfix

Algorithm for Infix to Postfix

Step 1: Consider the next element in the input.

Step 2: If it is operand, display it.

Step 3: If it is opening parenthesis, insert it on stack.

Step 4: If it is an operator, then

- If stack is empty, insert operator on stack.
- If the top of stack is opening parenthesis, insert the operator on stack
- If it has higher priority than the top of stack, insert the operator on stack.
- Else, delete the operator from the stack and display it, repeat Step 4.

Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.

Step 6: If there is more input, go to Step 1.

Step 7: If there is no more input, delete the remaining operators to output.

Example: Suppose we are converting 3*3/(4-1)+6*2 expression into postfix form.

Following table shows the evaluation of Infix to Postfix:

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(	/(	33*
4	/(	33*4
-	/(-	33*4
1	/(-	33*41
)	/	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+*	33*41-/6
2	+*	33*41-/62
	Empty	**33*41-/62*+**

So, the Postfix Expression is **33*41-/62*+**

2. Infix to Prefix

Algorithm for Infix to Prefix Conversion:

Step 1: Insert ")" onto stack, and add "(" to end of the A .

Step 2: Scan A from right to left and repeat Step 3 to 6 for each element of A until the stack is empty .

Step 3: If an operand is encountered, add it to B .

Step 4: If a right parenthesis is encountered, insert it onto stack .

Step 5: If an operator is encountered then,
a. Delete from stack and add to B (each operator on the top of stack) which has same or higher precedence than the operator.
b. Add operator to stack.

Step 6: If left parenthesis is encountered then ,
a. Delete from the stack and add to B (each operator on top of stack until a left parenthesis is encountered).
b. Remove the left parenthesis.

Step 7: Exit

Example:→(A-B)*(D/E)

SOLUTION:→

1. [-AB]*[/DE]
2. *-AB/DE

The list of the expression conversion is given below:

- Infix to prefix

- Infix to postfix
- Prefix to infix
- Prefix to postfix

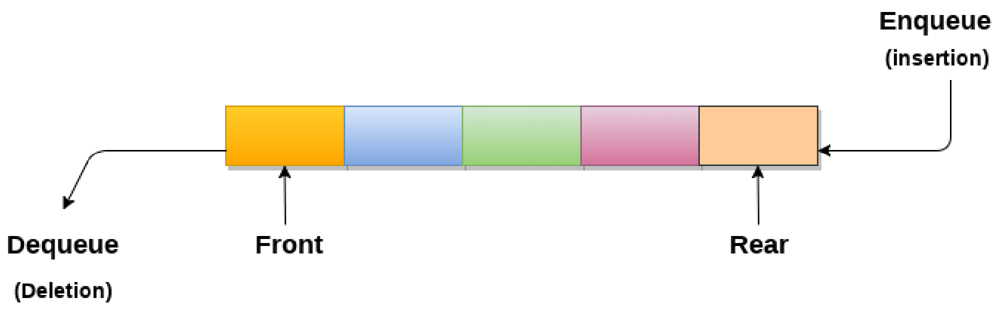
  Postfix to infix

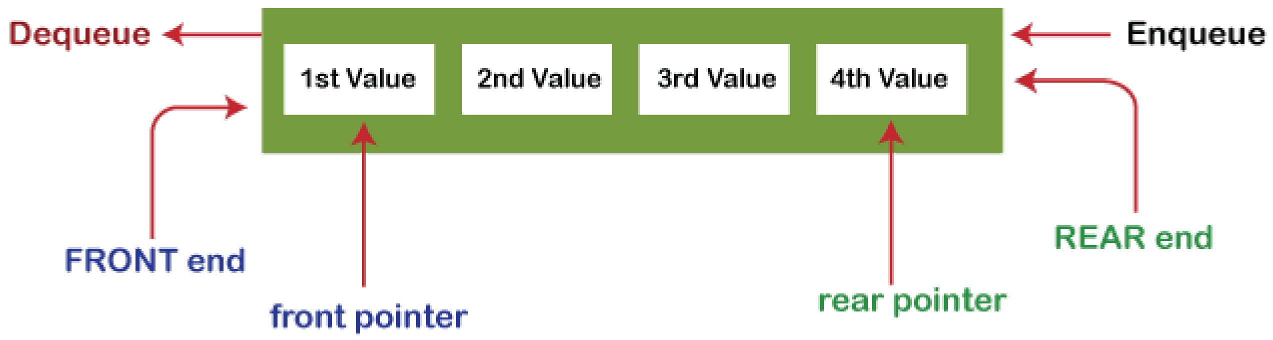
Queue

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.

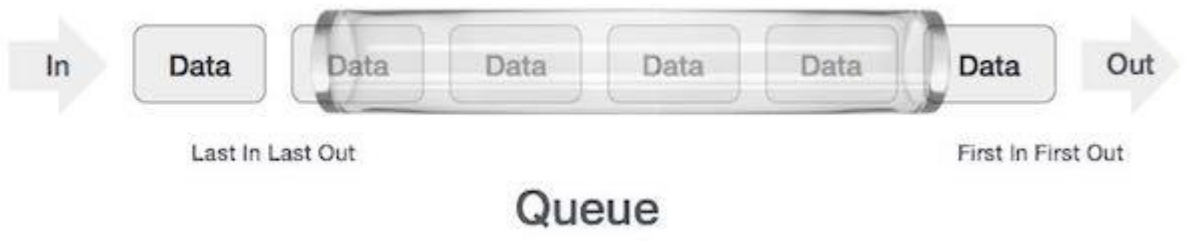
3. For example, people waiting in line for a rail ticket form a queue.





Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



Operations on Queue

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Noel:**→is an integer value of numbers of elements in a queue.And its value is 0 for an empty or newly created queue.
- **Dequeue:** The dequeue operation 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. The dequeue operation can also be designed to void.
- **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):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

What are the use cases of Queue?

Here, we will see the real-world scenarios where we can use the Queue data structure. The Queue data structure is mainly used where there is a shared resource that has to serve the multiple requests but can serve a single request at a time. In such cases, we need to use the Queue data structure for queuing up the requests. The request that arrives first in the queue will be served first. The following are the real-world scenarios in which the Queue concept is used:

- Suppose we have a printer shared between various machines in a network, and any machine or computer in a network can send a print request to the printer. But, the printer can serve a single request at a time.. If the requests are available in the Queue, the printer takes a request from the front of the queue, and serves it.

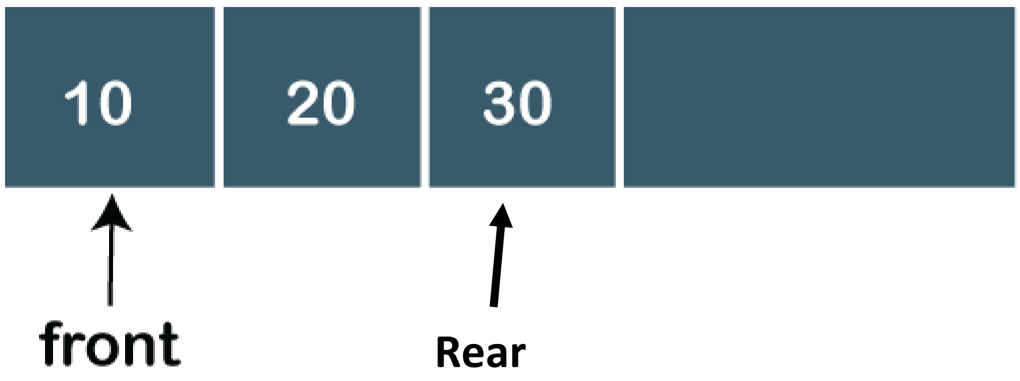
- The processor in a computer is also used as a shared resource. There are multiple requests that the processor must execute, but the processor can serve a single request or execute a single process at a time. Therefore, the processes are kept in a Queue for execution.

Types of Queue

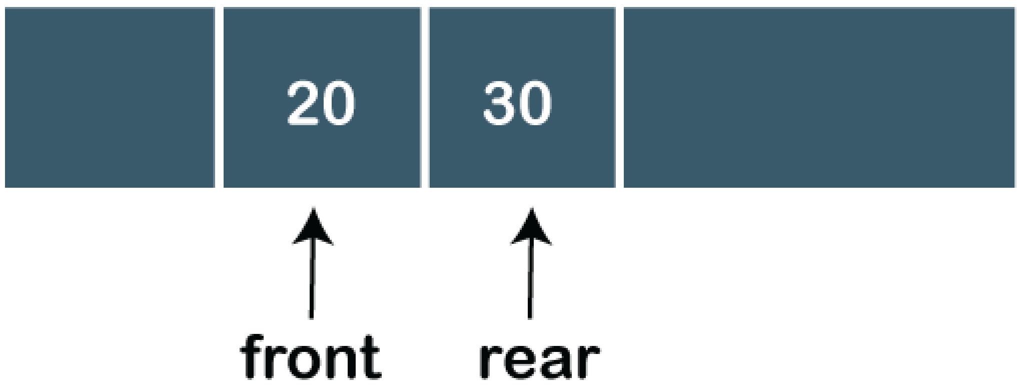
There are four types of Queues:

- **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 linear Queue can be represented, as shown in the below figure:



The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:

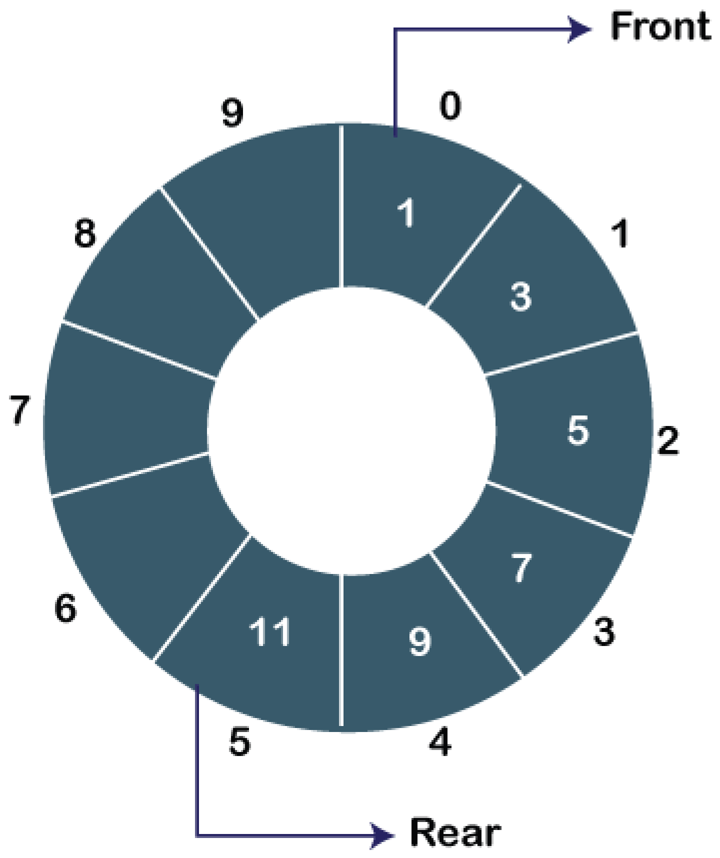


In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

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 circular queue can be represented as:



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.

- **Priority Queue**

A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:

The highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

- **Deque**

Both the Linear Queue and Deque are different as the linear queue follows the FIFO principle whereas, deque does not follow the FIFO principle. In Deque, the insertion and deletion can occur from both ends.

Implementation of Queue:→

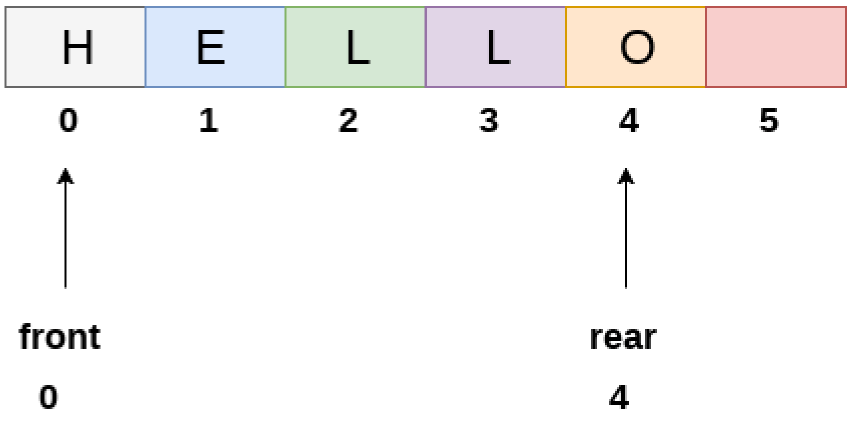
There are two ways of implementing the Queue:

- **Sequential allocation:** The sequential allocation in a Queue can be implemented using an array.
- **Linked list allocation:** The linked list allocation in a Queue can be implemented using a linked list.

Array representation of Queue

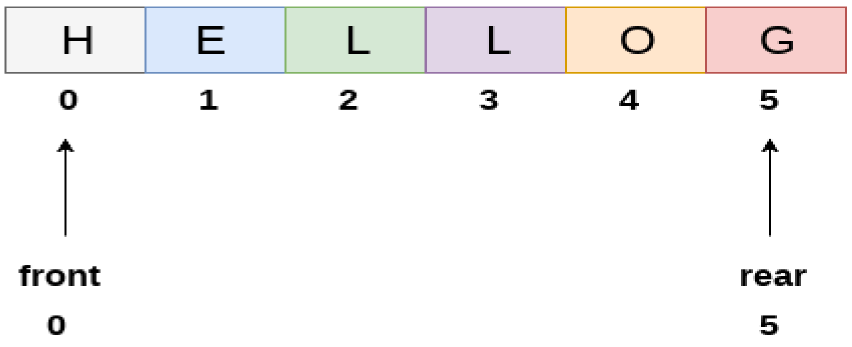
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front end queue is -1 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.



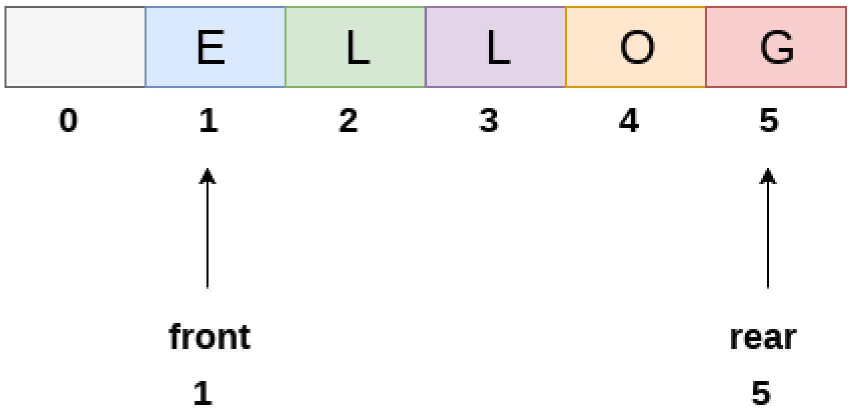
Queue

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 0 . 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 0 to 1. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
○ Step 1: IF REAR = MAX - 1
      Write OVERFLOW
      Go to step
      [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
```

C Function

```c
void insert (int queue[], int max, int front, int rear, int item)
```

```
{
    if (rear + 1 == max)
    {
        printf("overflow");
    }
    else
    {
        if(front == -1 && rear == -1)
        {
            front = 0;
            rear = 0;
        }
        else
        {
            rear = rear + 1;
        }
        queue[rear]=item;
    }
}
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

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
```

C Function

```
int delete (int queue[], int max, int front, int rear)
```

```
{
    int y;
    if (front == -1 || front > rear)

    {
        printf("underflow");
    }
    else
    {
        y = queue[front];
        if(front == rear)
        {
            front = rear = -1;
            else
            front = front + 1;

        }
        return y;
    }
}
```

Linked List implementation of Queue

the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

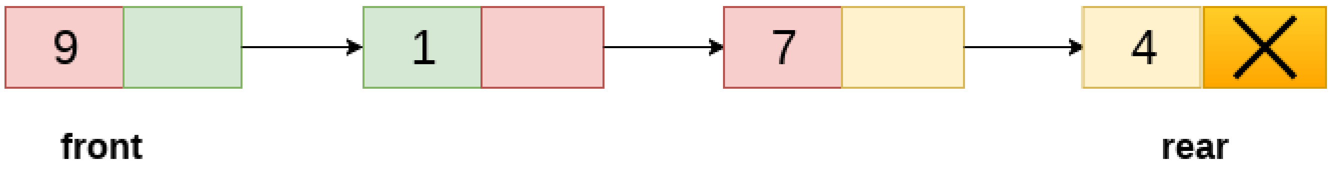
In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively.

- If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

Firstly, allocate the memory for the new node ptr by using the following statement.

```
Ptr = (struct node *) malloc (sizeof(struct node));
```

Algorithm

```
Step 1: Allocate the space for the new node PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            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
```

C Function

```c
void insert(struct node *ptr, int item; )
{


    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
{
    printf("\nOVERFLOW\n");
    return;
}
else
{
    ptr -> data = item;
    if(front == NULL)
    {
        front = ptr;
        rear = ptr;
        front -> next = NULL;
        rear -> next = NULL;
    }
    else
    {
        rear -> next = ptr;
        rear = ptr;
        rear->next = NULL;
    }   } }
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Algorithm

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

C Function

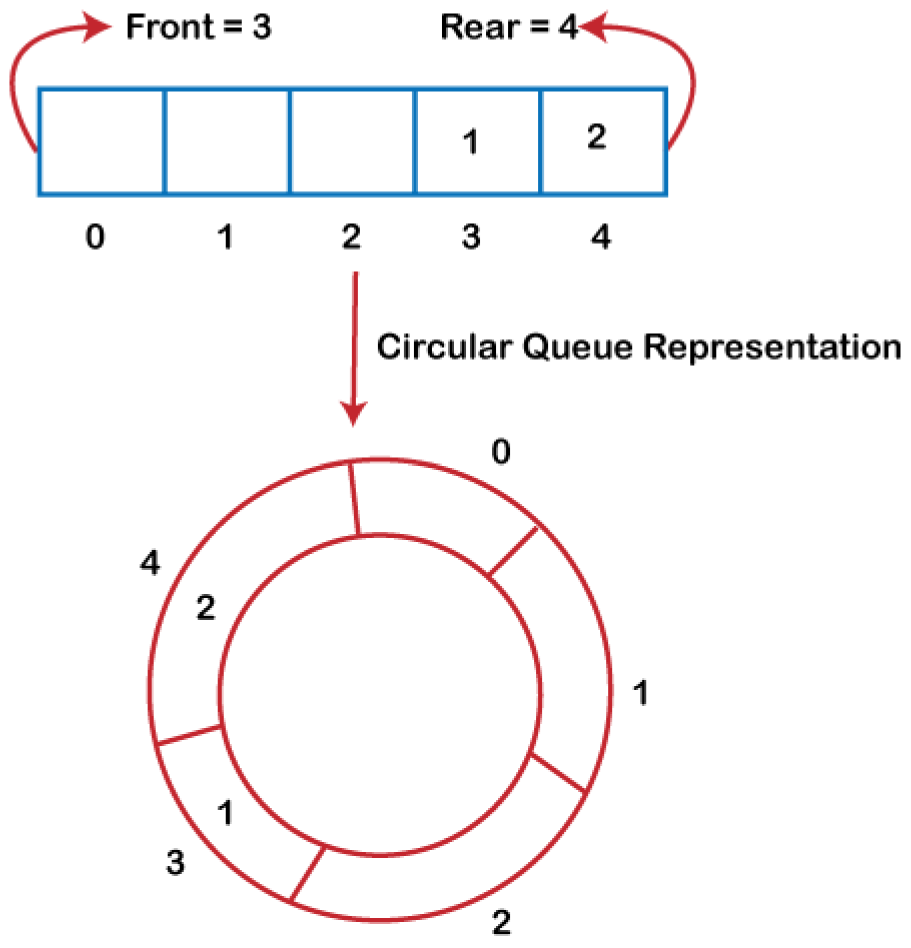
```c
void delete (struct node *ptr)
{
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
```

Circular Queue

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 0th positi

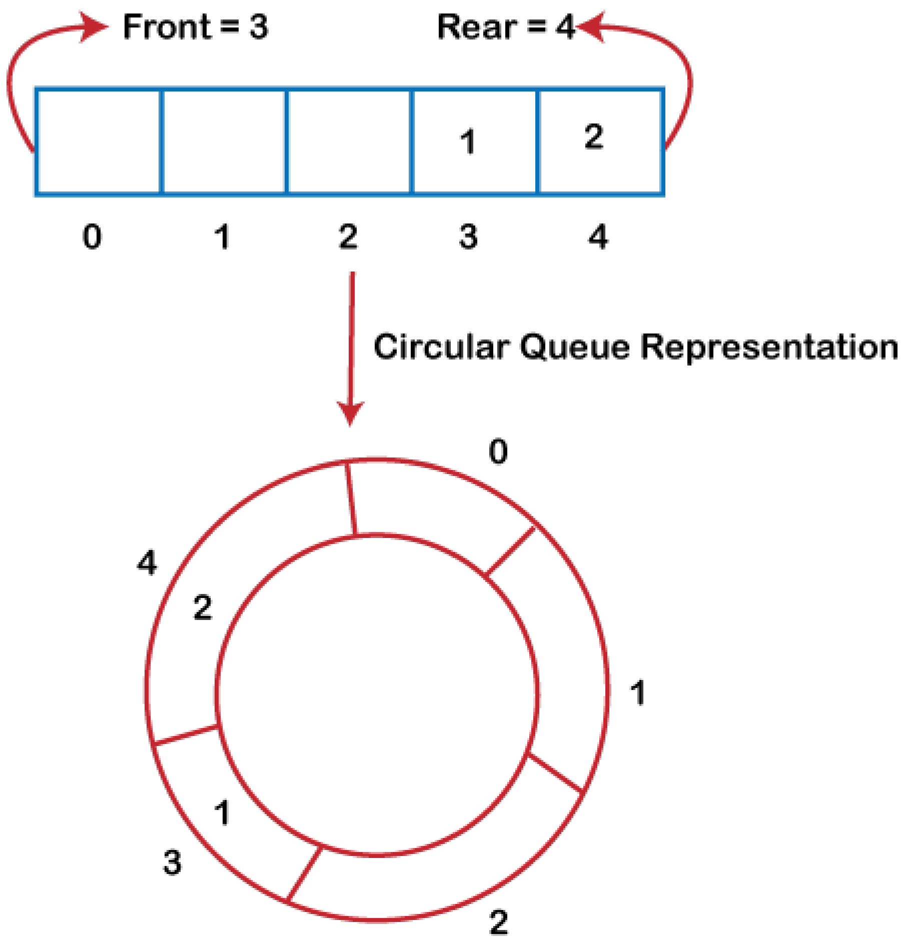


on. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting 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.

Circular Queue

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 0th 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.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.

- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.

- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- front== rear + 1;

Algorithm to insert an element in a circular queue

```
Step 1: IF (REAR+1)%MAX = FRONT
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) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT
```

Dequeue Operation

The steps of dequeue operation are given below:

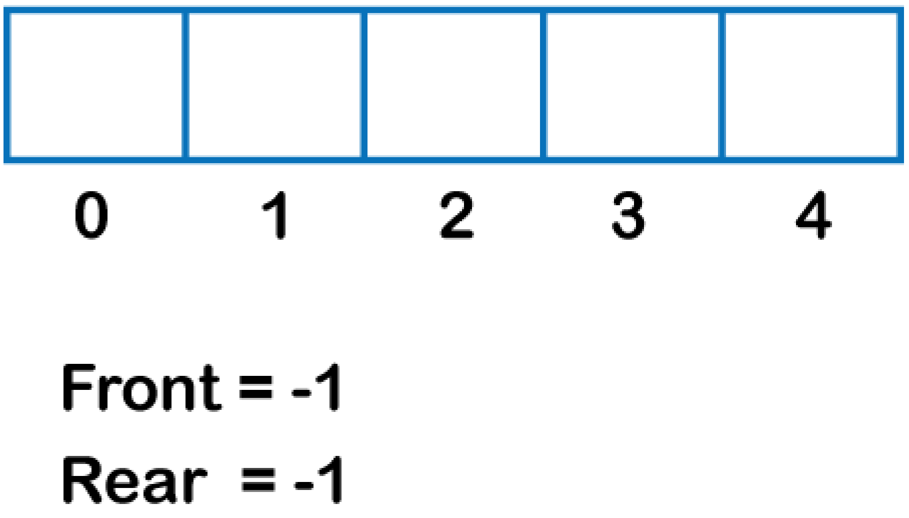
- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

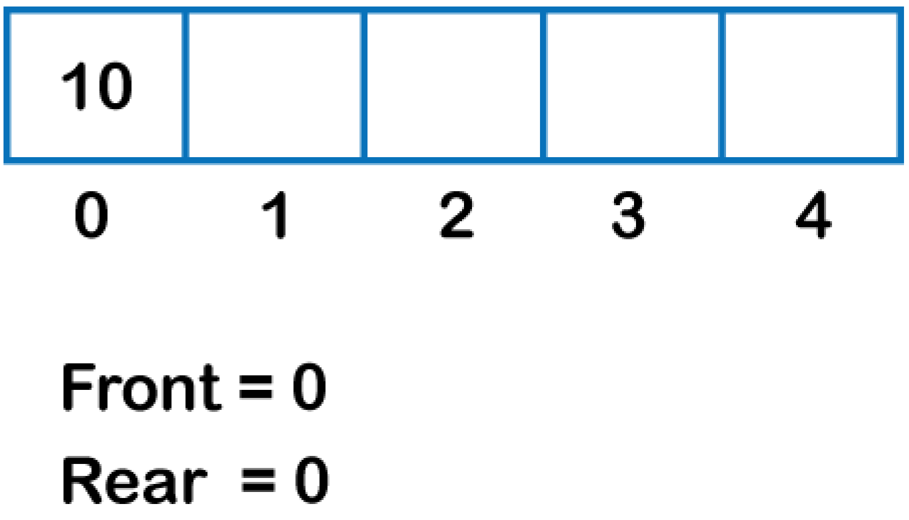
Algorithm to delete an element from the circular queue

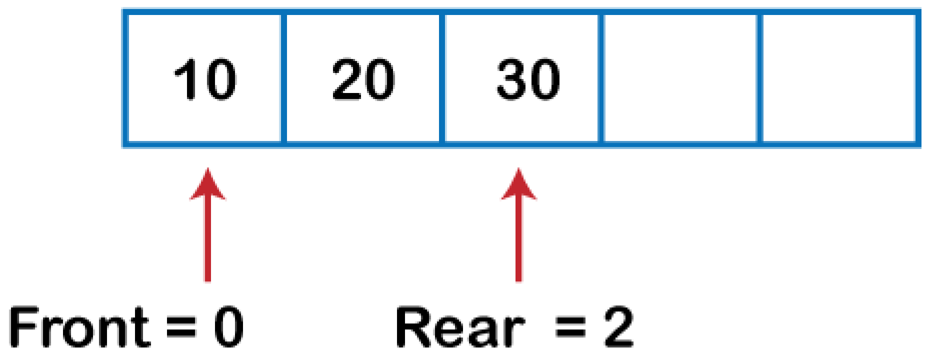
```
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]
```

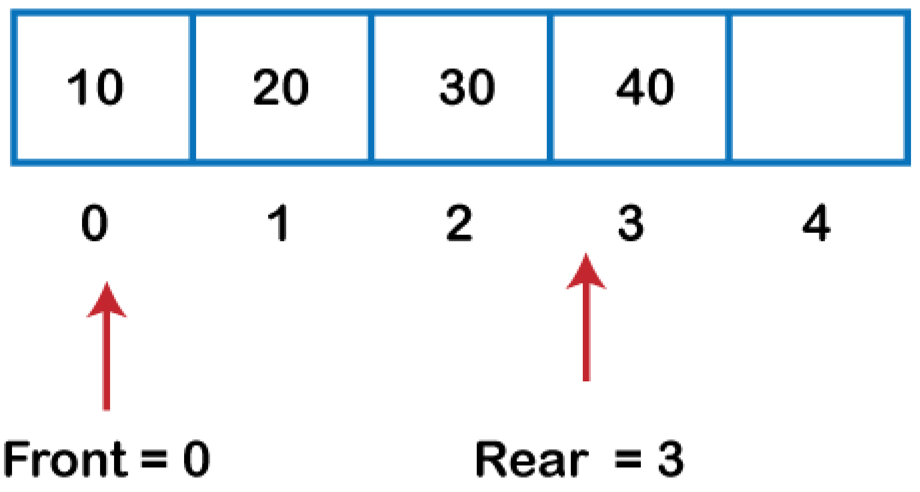
```
Step 2: SET VAL = QUEUE[FRONT]

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
```

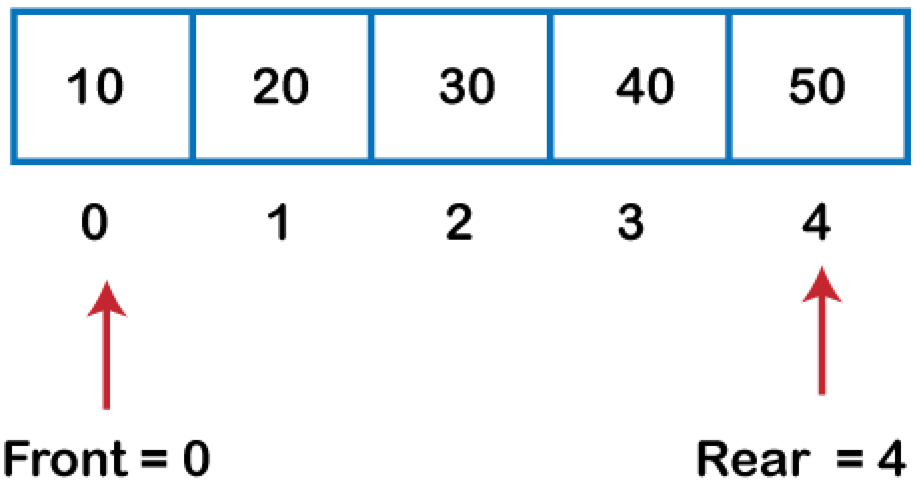
Let's understand the enqueue and dequeue operation through the diagrammatic representation.

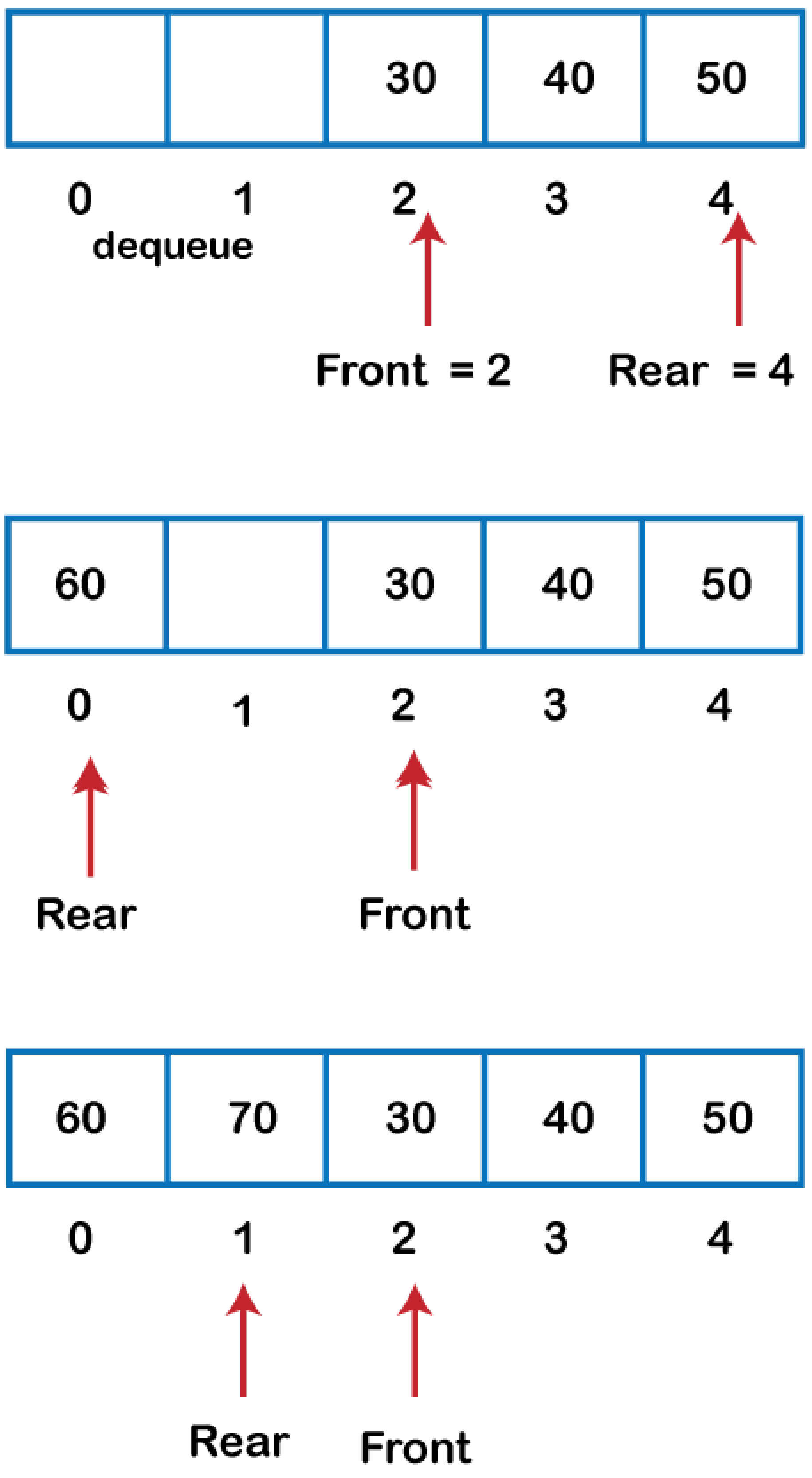





10
20
30
Front = 0
Rear = 2

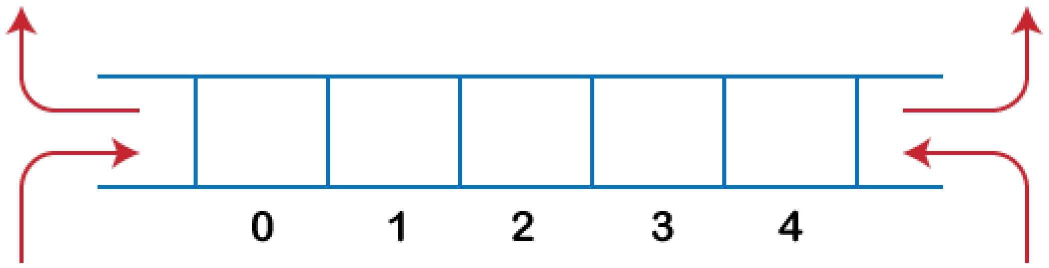

10
20
30
40
0
1
2
3
4
Front = 0
Rear = 3


10
20
30
40
50
0
1
2
3
4
Front = 0
Rear = 4


30
40
50
0
1
2
3
4
dequeue
Front = 2
Rear = 4
60
30
40
50
0
1
2
3
4
Rear
Front
60
70
30
40
50
0
1
2
3
4
Rear
Front

Deque

The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

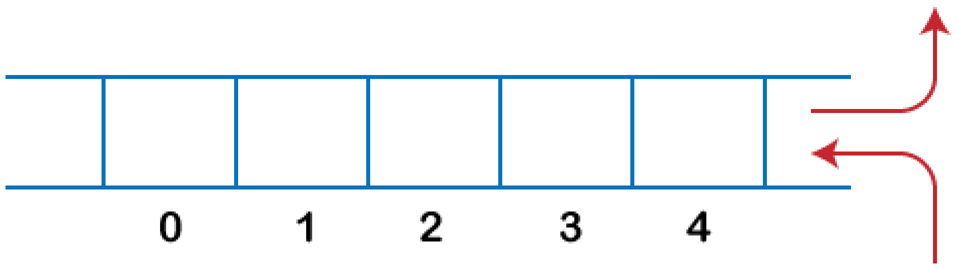


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

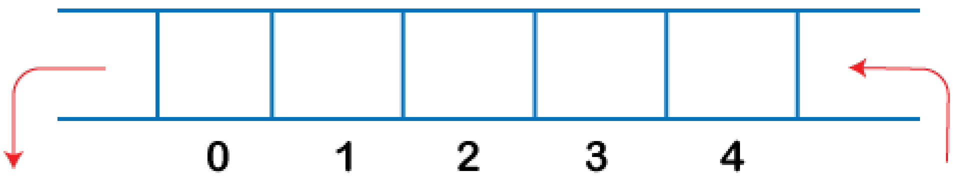
Let's look at some properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.

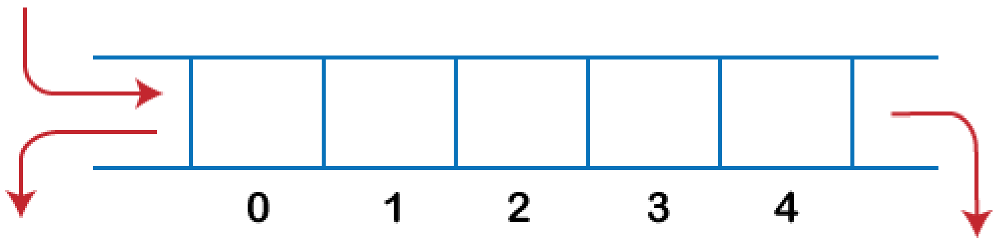


In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

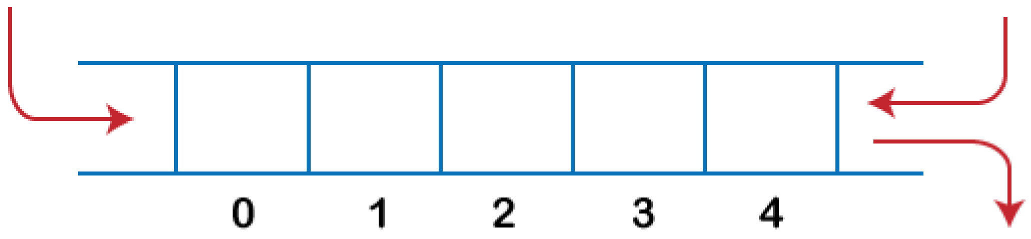


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Operations on Deque

The following are the operations applied on deque:

- **Insert at front**
- **Delete from front**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

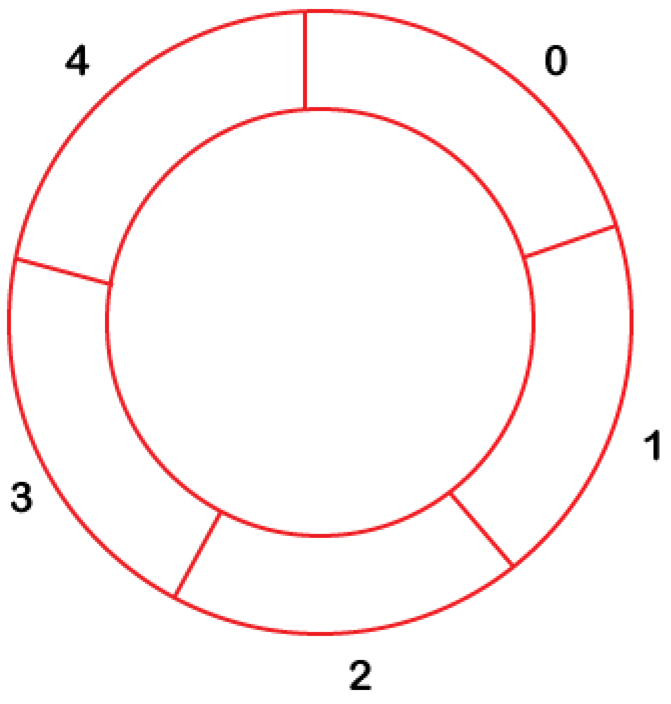
- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Memory Representation

The deque can be implemented using two data structures, i.e., **circular array**, and **doubly linked list**. To implement the deque using circular array, we first should know **what is circular array**.

What is a circular array?

An array is said to be **circular** if the last element of the array is connected to the first element of the array. Suppose the size of the array is 4, and the array is full but the first location of the array is empty. If we want to insert the array element, it will not show any overflow condition as the last element is connected to the first element. The value which we want to insert will be added in the first location of the array.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. **In Networks:**
   a) Queues in routers/ switches
   b) Mail Queues
5. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
6. Queues are used in operating systems for handling interrupts.

Recursion in data structures

- When function is called within the same function, it is known as **recursion** in C. The function which calls the same function, is known as **recursive function**.
- Recursion is defined as defining anything in terms of itself. Recursion is used to solve problems involving iterations, in reverse order.
- Recursion code is shorter than iterative code however it is difficult to understand.
- Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

  Example:→Algorithm for factorial of number

```
Step 1: Start
Step 2: Declare Variable n, fact, i
Step 3: Read number from User
Step 4: Initialize Variable fact=1 and i=1
Step 5: Repeat Until i<=number
                5.1 fact=fact*i
                5.2 i=i+1
Step 6: Print fact
Step 7: Stop
```

```
#include <stdio.h>
int fact (int);
int main()

{
   int n,f;
   printf("Enter the number whose factorial you want to calculate?");
   scanf("%d",&n);
   f = fact(n);
   printf("factorial = %d",f);
}
int fact(int n)
{     if (n==0)
   {        return 0;    }
   else if ( n == 1)
   {
      return 1;    }
   else
   {        return n*fact(n-1);    } }
```

Advantages of recursion

1. The code may be easier to write.
2. To solve such problems which are naturally recursive such as tower of Hanoi.
3. Reduce unnecessary calling of function.
4. Extremely useful when applying the same solution.
5. Recursion reduce the length of code.
6. It is very useful in solving the data structure problem.
7. Stacks evolutions and infix, prefix, postfix evaluations etc.

Disadvantages of recursion

- Recursive functions are generally slower than non-recursive function.
- It may require a lot of memory space to hold intermediate results on the system stacks.
- Hard to analyze or understand the code.
- It is not more efficient in terms of space and time complexity.
- The computer may run out of memory if the recursive calls are not properly checked.
- Recursive solution is always logical and it is very difficult to trace.(debug and understand).
- Recursion uses more processor time.
- The computer may run out of memory if the recursive calls are not checked.
-