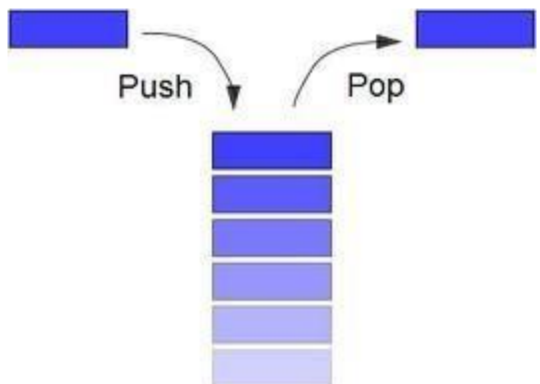
# UNIT – III Stacks and Queues

## Stacks :

A stack is a data structure where elements are inserted and removed according to the last-in first-out (LIFO) mechanism.




### Stack Data Structure:

Stack data structure works on the principle Last In First Out (LIFO). The last element added to the stack is the first element to be deleted. Insertion and deletion takes place at one end called TOP. It looks like one side closed tube.

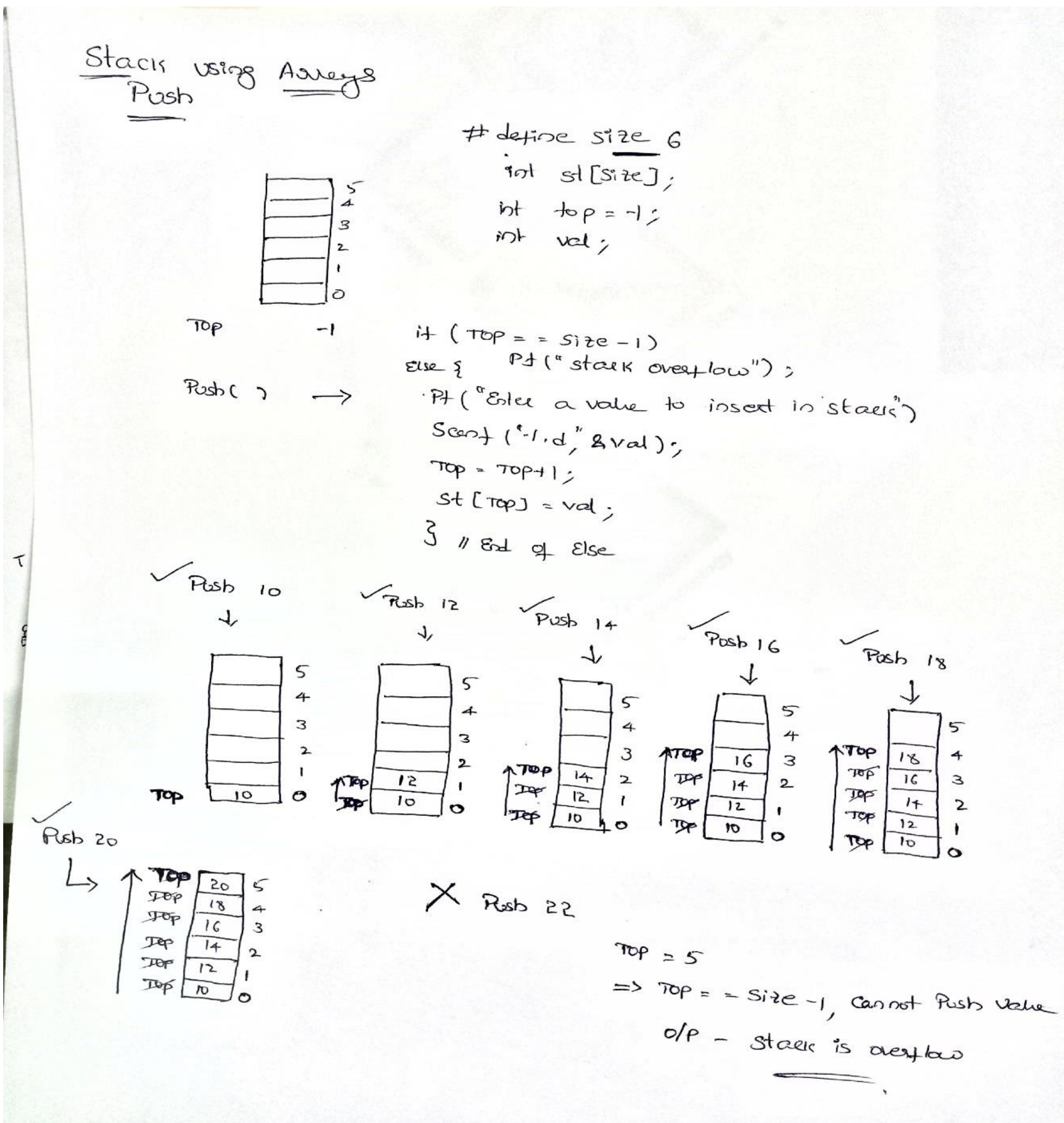
Operations that can be performed on a stack include

- adding an element into stack is called **push** operation
- The delete operation is called as **pop** operation.
- Push operation on a full stack causes **stack overflow**.
- Pop operation on an empty stack causes **stack underflow**.
- **Top (or) stack pointer(sp)** is a variable/pointer, which is used to access the top element of the stack.

## push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function reads an integer value and inserts that value into the stack. We can use the following steps to push an element on to the stack.

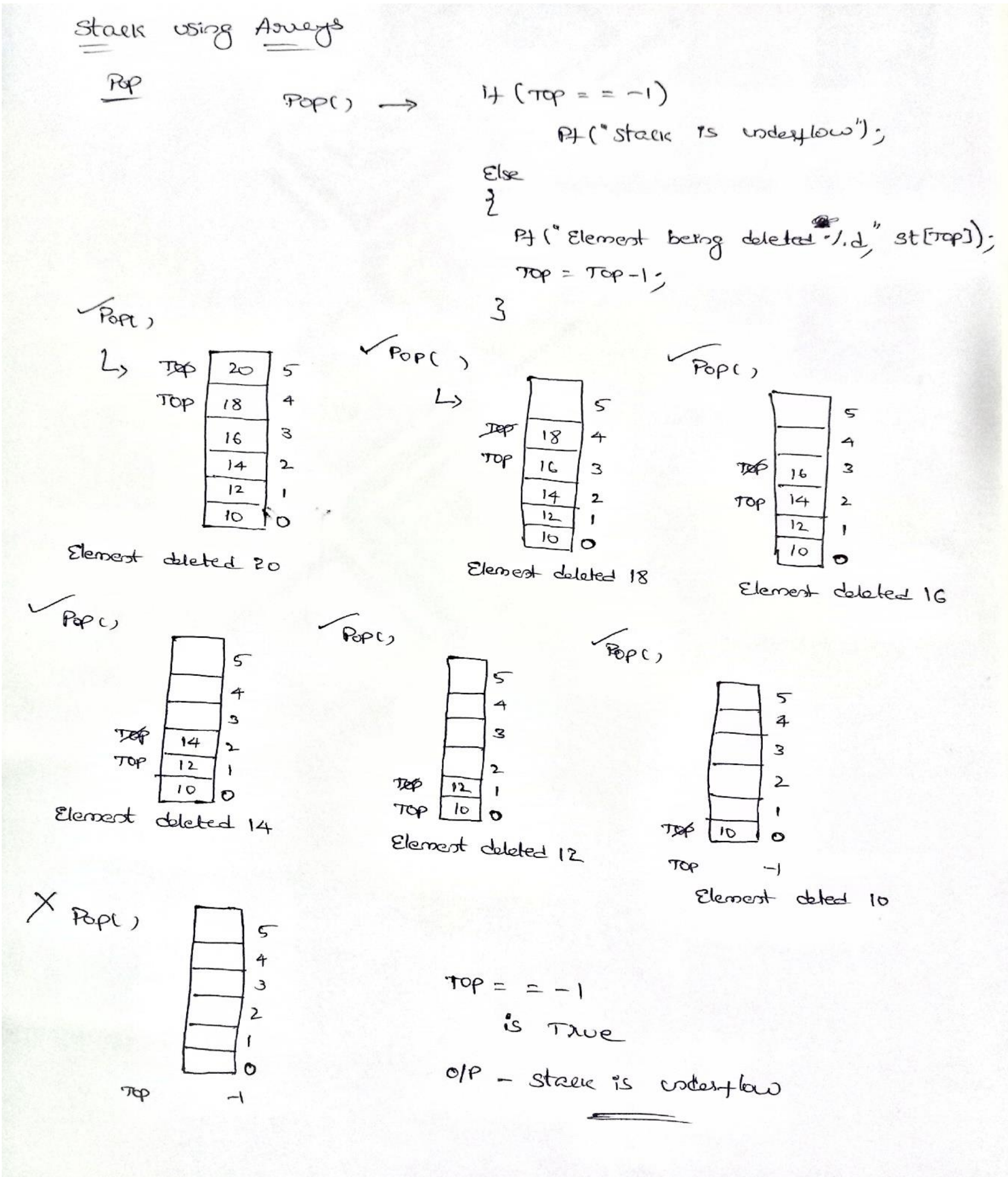
- Check whether **stack** is **FULL**. **if** (**top == SIZE-1**)
    - If it is **FULL**, then
        - display **"Stack is FULL Insertion is not possible!"** and terminate.
    - If it is **NOT FULL**
        - read a value to store in stack, say scanf(val)
        - then increment **top** value by one (**top++**)
        - set **stack[top]** to value (**stack[top] = val**).



## pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter.

- Check whether **stack** is **EMPTY**. **if**(**top == -1**)
  - If it is **EMPTY**,
    - then display **"Stack is EMPTY! Deletion is not possible!"** and terminate.
  - If it is **NOT EMPTY**,
    - then display **stack[top]** is deleted and decrement **top** value by one (**top--**).



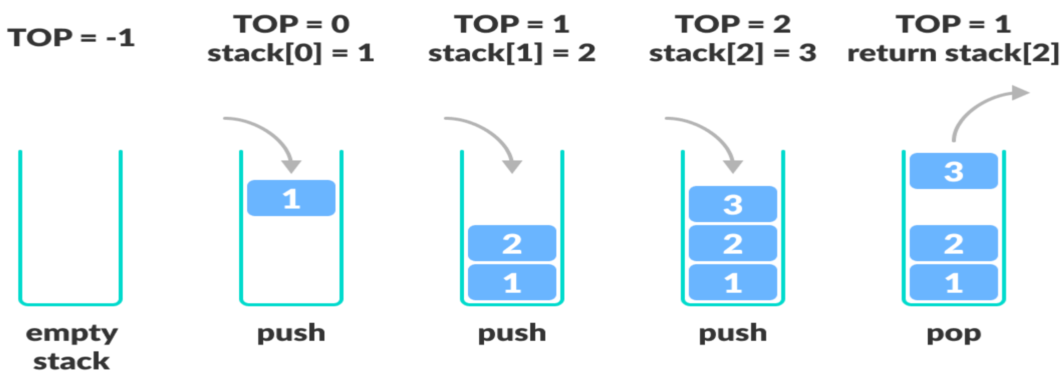
## display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack.

- Check whether **stack** is **EMPTY**. **if** (**top == -1**)
  - If it is **EMPTY**
    - then display **"Stack is EMPTY!"** and terminate.
  - If it is **NOT EMPTY**
    - then define a variable '**i**' and initialize with current top value.
    - Display **stack[i]** value and decrement **i** value by one (**i--**).
    - Repeat above step until **i** value becomes '0'

### Representation of a Stack using Arrays:

When a element is added to a stack, the operation is performed by push().



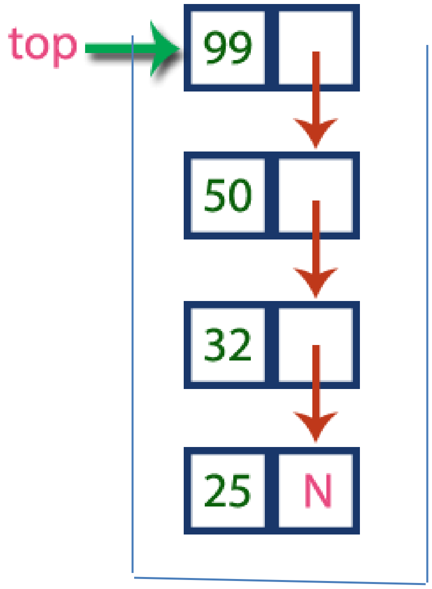

### Linked List Implementation of Stack:

stack implemented using an array works only for a fixed number of data values(size of array). Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. The Stack implemented using linked list can organize as many data values as we want.

First we create a node structure. below is the Linked list node, which will have data in it and a node pointer to store the address of the next node element.

```
struct node
{
        int data;
        node *next;
};
```

In linked list implementation of a stack, every new element is inserted as '**top**' element. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first inserted element will be always **NULL**.




In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

### Stack Applications:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.

## push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **temp node** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**,
  - then set **temp** → **next** = **NULL**
  - **top = temp**
- **Step 4 -** If it is **Not Empty**,
  - then set **temp** → **next** = **top**.
  - Finally, set **top** = **newNode**.

Stack Linked List

Push()

```
struct Node
{
    int data;
    struct Node *next;
};
    struct Node *top = NULL
    struct node * temp = NULL
```

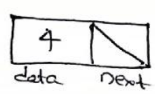
Pop()

top = = NULL, is True // no nodes in ~~List~~ Stack

O/P - stack is underflow

Push()

① {
```
temp = (struct Node *)malloc (sizeof (struct Node));
Pf (" Enter a value to Push");
scanf ("%d", &val);        // Given val = 4
temp -> data = val;
temp -> next = NULL;
```
}

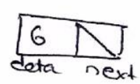
temp → 


if ( top = = NULL) is true // no nodes in stack

top = temp;

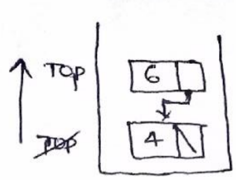

Push()

Execute statements under ①, Given val = 6, now

temp → 


if ( top = = NULL )

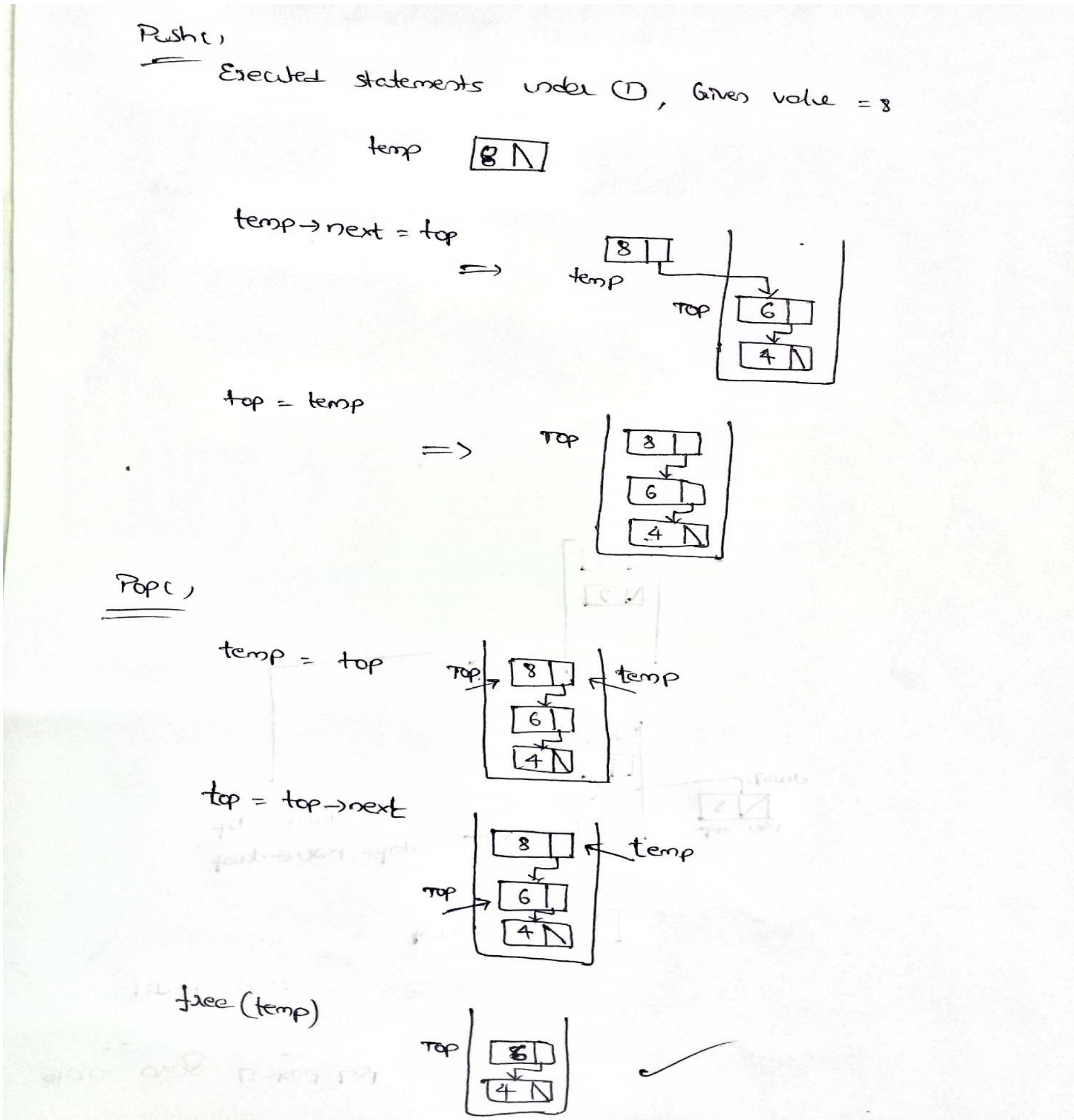
is False

✓ Else
{
temp → next = top
top = temp;
}
→ 

## pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- Check whether **stack** is **Empty** (**top == NULL**).
- If it is **Empty**,
    - then display **"Stack is underflow"** and terminate
- If it is **Not Empty**,
    - then define a **Node** pointer '**temp**' and set **temp = top**.
    - set '**top** = **top** → **next**'.
    - Finally, delete '**temp**'. (**free(temp)**).



## display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- Check whether stack is **Empty** (**top** == **NULL**).
- If it is **Empty**,
    - then display **'Stack is Underflow'** and terminate
- If it is **Not Empty**,
    - then define a Node pointer **'temp'** and set **temp = top**.
    - Display **temp** → **data** and move it to the next node(**temp=temp->next**).
    - Repeat the above step until **temp** reaches to the first node in the stack.
    - i.e., **while(temp** != **NULL**).

## In-fix- to Postfix Transformation:

### Procedure:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

   c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

   d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

| Symbol | Postfix string | Stack | |
|---|---|---|---|
| A | A | | |
| + | A | + | |
| B | A B | + | |
| * | 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 / | - * | |
| H | A B C * + D E / H | - * | |
| End of string | A B C * + D E / H * - | The input is now empty. Pop the output symbols from the stack until it is empty. | |

## Evaluating Postfix Expressions:

**Procedure:**

The postfix expression is evaluated easily by the use of a stack.

Step 1: When a operand/number is seen, it is pushed onto the stack

Step 2: When an operator is seen, the top two opearnds are popped from the stack and the operator is applied on them. The result is pushed back onto the stack.

Step 3: At the end of expression final result in the stack will be the output.

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

| Symbol | Operand 1 | Operand 2 | Value | Stack | Remarks |
|---|---|---|---|---|---|
| 6 | | | | 6 | |
| 5 | | | | 6, 5 | |
| 2 | | | | 6, 5, 2 | |
| 3 | | | | 6, 5, 2, 3 | The first four symbols are placed on the stack. |
| + | 2 | 3 | 5 | 6, 5, 5 | Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed |
| 8 | 2 | 3 | 5 | 6, 5, 5, 8 | Next 8 is pushed |
| * | 5 | 8 | 40 | 6, 5, 40 | Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed |
| + | 5 | 40 | 45 | 6, 45 | Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed |
| 3 | 5 | 40 | 45 | 6, 45, 3 | Now, 3 is pushed |

| | | | | | |
|---|---|---|---|---|---|
| + | 45 | 3 | 48 | 6, 48 | Next, ‘+’ is seen so 3 and 45 are popped and 45 + 3 = 48 is pushed |
| * | 6 | 48 | 288 | **288** | Finally, a ‘*’ is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed |

## Queue

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. To implement queue data structure using array define a one dimensional array of specific size then insert or delete the values into that array by using **FIFO (First In First Out) mechanism** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

# Queue Operations using Array

**create an empty queue.**

- **Step 1** - Define a constant size for array, say #define size 10
- **Step 2 -** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 3 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**.
    - (**int front = -1, rear = -1**)

## enQueue() - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function reads one integer value(val) and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1 -** Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display
    - **"Queue is FULL, Insertion is not possible"** and terminate.
- **Step 3 -** If it is **NOT FULL**, then
    - then check '**front == -1**' if it is **TRUE**, then set **front = 0**.
    - increment **rear** value by one (**rear++**) and set **queue[rear]** = **val**.

## deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2 -** If it is **EMPTY**,
    - then display **"Queue is EMPTY, Deletion is not possible"** and terminate.
- **Step 3 -** If it is **NOT EMPTY**,
    - Then display **queue[front]** is the element being deleted,
    - then increment the **front** value by one (**front ++**).
    - Then check whether both **front** and **rear** are equal (**front** == **rear**),
    - if it **TRUE**,
        - then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

## display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

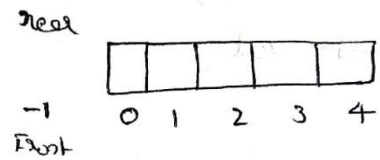
- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1** )
- **Step 2 -** If it is **EMPTY**, then
        - display **"Queue is EMPTY!!!"** and terminate.
    - If it is **NOT EMPTY**,
        - then define an integer variable '**i**' and set '**i** = **front**'.
        - Display '**queue[i]**' value and increment '**i**' value by one (**i++**).
        - Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

## Queue using Arrays



```
# define size 5
int Q[size];
int front = -1;
int rear = -1;
```

Queue initial state




**dequeue()**

Let's go with dequeue at this state.

```
if (front == -1 || front == rear+1)
```

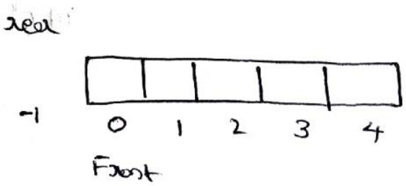
True

O/P - queue is underflow

**Enqueue()**

For the very first insertion of Element, bring front from -1 to 0

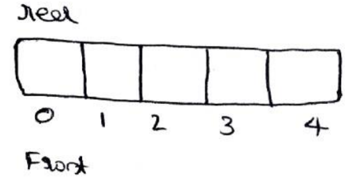
```
if (Front == -1)
    Front = 0;
```



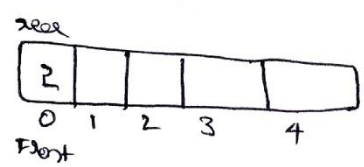

```
Pf("Enter a value to insert in Q");
Scanf (val);
```

// Say given val = 2

```
rear = rear + 1;
```




```
Q[rear] = val;
```



(2)

Enqueue()

Given val = 4, then

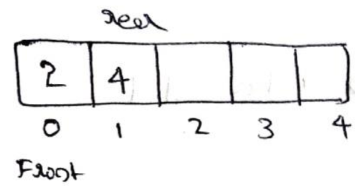
if (Front == -1)

False // for Second Enque onward it will be false

Scanf(val) // 4

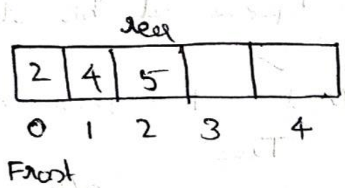
rear = rear+1;

Q[rear] = val




Enqueue()

val = 5, then




dequeue()

if (Front == -1 || Front == rear+1)

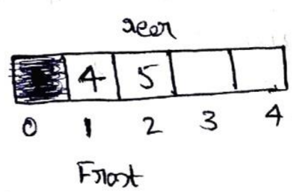
False False

Pf(" Element being deleted %d", Q[Front]

// O/P -

Elemented deleted 2

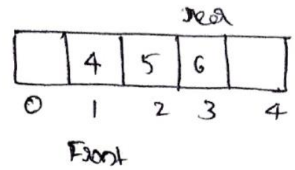
Front = Front+1;



Queue using Arrays

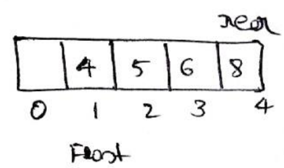
Enqueue()  val = 6

=>




Enqueue()  val = 8

=>

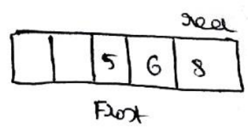



Enqueue()  val = 10

=> if ( rear = = Size-1)  // size is 5
True

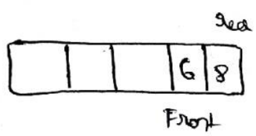
O/P - Q is overflow

dequeue()



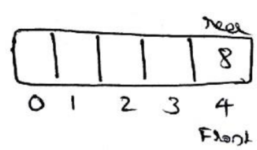

O/P - Element deleted is 4

dequeue()



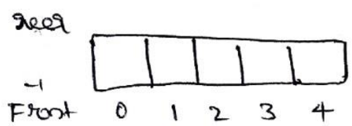

O/P - Element deleted is 5

dequeue()




O/P- Element deleted is 6

* dequeue()



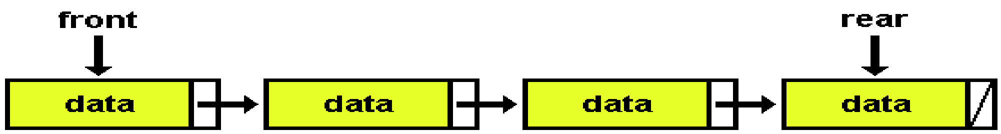

O/P - Element deleted is 8

if ( Front = = rear )  // when we delete all Elements upto rear
Front = -1  no more Elements Exist in queue, so reinitialize
rear = -1  Front and rear to -1, so Q can be reused again

# Queue Operations using Linked List

**queue implemented using an array will work for an only fixed number of data values. A queue data structure can be implemented using a linked list data structure. The Queue implemented using linked list can organize as many data values as we want.**

**In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.**




To implement queue using linked list,

- **Step 1 -** Define a '**Node**' structure with two members **data** and **next**.

```
struct Node
{
        int data;
        struct Node * next;
};

struct Node *front = NULL;
struct Node *rear = NULL;
```

- **Step 2 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.

## enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **temp** with given value and set '**temp** → **next**' to **NULL**.

```
struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
scanf("%d",&val);
temp -> data = val;
temp -> next = NULL;
```

- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
    - If it is **Empty** then,
        - set **rear** = **temp** and **front** = **temp**. // need to update front as it is first node
    - If it is **Not Empty** then,
        - set **rear** → **next** = **temp** and **rear** = **temp**.

## deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Check whether **queue** is **Empty** (**front == NULL**).

- If it is **Empty**, then
  - display **"Queue is Empty, Deletion is not possible"** and terminate.
- If it is **Not Empty** then
  - define a Node pointer '**temp**' and set it to '**front**', **temp = front.**
  - then set '**front** = **front** → **next**' and delete '**temp**' (**free(temp)**).

## display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
  - If it is **Empty** then,
    - display **'Queue is Empty!!!'** and terminate.
  - If it is **Not Empty** then,
    - define a Node pointer **'temp'** and initialize with **front**.
    - Display '**temp** → **data** ' and move it to the next node. **temp = temp -> next**
    - Repeat the same until '**temp**' reaches to '**rear**'/**lastnode** (**temp** != **NULL**).

## Queue using Linked List

(1)

```
Struct Node
{
   int data;
   Struct Node *next;
}
Struct Node * front = NULL;
Struct Node * rear = NULL;
Struct Node * temp = NULL;
```

Present state of queue List

rear = NULL
Front = NULL

**dequeue()**

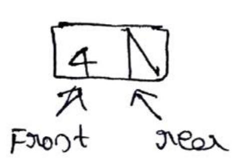
if ( Front == NULL)
   True      o/p - No nodes in Q List

First Enqueue Some Values into Q Linked List

**Enqueue()**

(1) { allocate memory for node, Say temp
Scan Value, Store in data member
temp->next = NULL; }


if (rear == NULL)
   True   => then make both Front and rear = temp
// will be Executed only when list Empty.



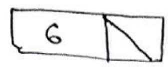
## Enqueue()

Executed (1) statements

read a value, Create temp node with given value

// Say val = 6

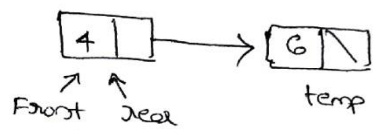
temp



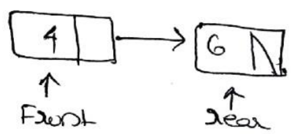

~~if(Front =~~ if (rear == NULL)

False, // List is not Empty

rear->next = temp



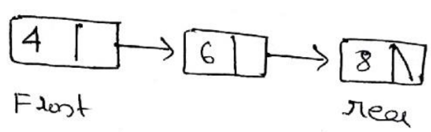

rear = temp




## Enqueue()

Say val = 8

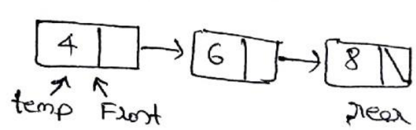
=>




. . . . . .

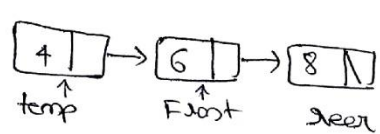
## dequeue()

temp = Front =>

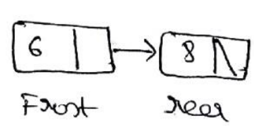



Front = Front->next

=>




✓ free(temp) =>



## Circular Queue:

In a Queue Data Structure, we can insert elements until queue becomes full, once the queue becomes full, we can not insert the next element until all the elements are deleted from the queue.

A circular queue is a data structure in which the last position is connected back to the first position to make a circle. we can insert the elements in previously deleted positions by moving front to beginning again.

## Implementation of Circular Queue

To implement a circular queue

- **Step 1 -** Create a one dimensional array with constant SIZE (**int cQueue[SIZE]**)
- **Step 2 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**.
    - (**int front = -1, rear = -1**)

## enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. The new element is always inserted at **rear** position. In enQueue() function an integer value will be read and inserted into the circular queue.

- **Step 1 -** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+1)**)
    - If it is **FULL**, then
        - display **"Queue is FULL. Insertion is not possible!"** and terminate.
    - If it is **NOT FULL**
        - then check '**front == -1**' if it is **TRUE**, then set **front = 0**.
        - Increment **rear = (rear+1)%size.**
        - set **queue[rear]** = **value**

## deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter.

- Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
    - If it is **EMPTY**,
        - then display **"Queue is EMPTY! Deletion is not possible!"** and terminate.
    - If it is **NOT EMPTY**,
        - then display **queue[front]** is the element being deleted element
        - increment the **front** value by one **front = (front+1)%size.**
        - Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**),
        - if it **TRUE**,
            - then set both **front** and **rear** to '**-1**' (**front** = -1 and **rear** = **-1**).

## display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

- Check whether **queue** is **EMPTY**. (**front == -1**)
    - If it is **EMPTY**,
        - then display **"Queue is EMPTY"** and terminate.
    - If it is **NOT EMPTY**,
        - then define an integer variable '**i**' and set '**i** = **front**'.
        - Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value
        - increment '**i**' value by one (**i++**).
        - Repeat the same until '**i <= rear**' becomes **FALSE**.

## Circular queue using Arrays

(1)

Problem with normal queue –

Let's Consider state of queue as follows

| | | | | rear |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 |
| Front | | | | |

After two deque operations, state of queue will be as follows

| | | | | rear |
|---|---|---|---|---|
| | | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 |
| | | Front | | |

Now, if we want to insert a new Element say 6,

because of our terminating condition

if (rear == Size-1)

True, we can not insert any Element

→ but Space at index 0 and 1 is free as Previous Elements 1 and 2 are dequeued

* So to make an Efficient utilization, bring rear back to initial Position again and reuse the cells until we reach 'Front'

$rear = (rear+1) \% size$ // To increment rear in cir Q

$\Rightarrow rear = (4+1) \% 5 = 0$

| rear | | | | |
|---|---|---|---|---|
| | | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 |
| | | Front | | |

# Circular queue using Arrays

rear

-1

Front

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | | | | |

**Enqueue**

val = 1

rear

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | |

Front

if (Front == -1)
    Front = 0

rear = (rear + 1) % size
     = (-1 + 1) % 5
     = 0

**Enqueue()**

Val = 2

rear

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 2 | | | |

Front

rear = (rear + 1) % size
     = (0 + 1) % 5
     = 1

After Enqueue of 3, 4, 5

rear

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |

Front

dequeue() for 4 times,

rear

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | | | | 5 |

Front

Front = (Front + 1) % size

Enqueue 6, 7

rear

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 6 | 7 | | | 5 |

Front

Again dequeue,

Front = (Front + 1) % size => Front = (4 + 1) % 5 = 0

o/p - Element deleted is 5

rear

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 6 | 7 | | | |

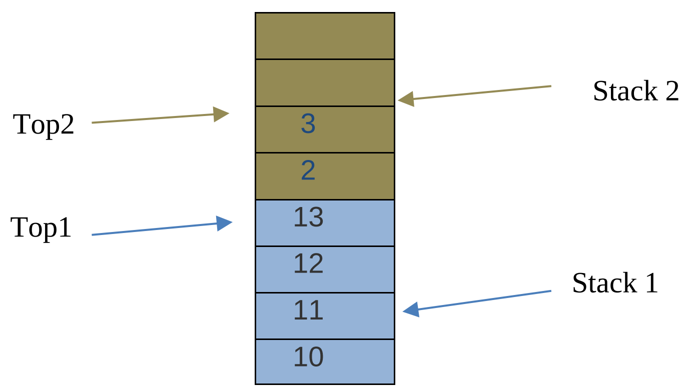
Front

# Multistack

- It is process of implementing multiple stacks in a single array

**Method** 1

A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

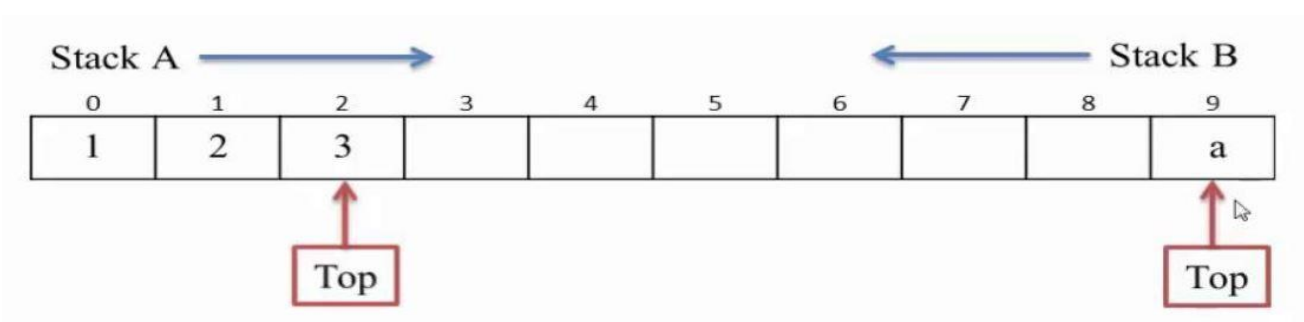



Note: The problem with this method is inefficient use of array space.

Say, now we want to store element 15 in stack1, we can not insert as space allocated for stack1 is completely utilized, even though space is available under stack 2 which is never used by stack 2

**Method 2**

This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks.



# Applications of Queues

- Direct applications:-
    - Waiting lists
    - Access to shared resources (e.g., printer)
    - Multiprogramming
- Indirect applications:-
    - Auxiliary data structure for algorithms
    - Component of other data structures