

# Lecture Notes

On

## UNIT I

2020 – 2021

II B.Sc (CS) III Semester

Mrs.S.Sheela, Assistant Professor of Computer Science



**Government College for Women (Autonomous), Kumbakonam**  
**Department of Computer Science**

# STACK ADT

## STACK

A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is often known as top of stack. When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop. Stack is also called as Last- In-First- Out (LIFO) list.

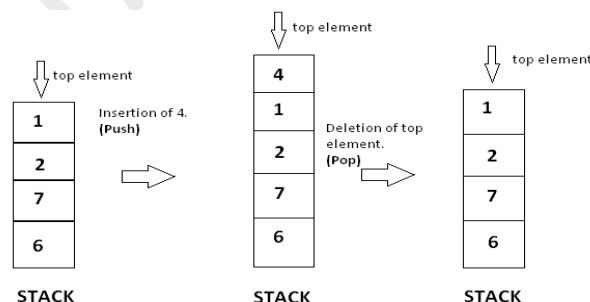
## Operations on Stack

There are two possible operations done on a stack. They are pop and push operation.

- **Push:** Allows adding an element at the top of the stack.
- **Pop:** Allows removing an element from the top of the stack.

## Representation of Stack

- A linear list which allows insertion and deletion of an one element at one end only is called as stack.
- The insertion operation is called as PUSH and deletion operation is called as POP.
- The most and least accessible elements in stack are known as top and bottom of the stack respectively.
- Since insertion and deletion operations are performed at one end of a stack, the elements can only be removed in the opposite orders from that in which they were added to the stack; such a linear list is referred to as a LIFO (Last in First out) list.



- A pointer TOP keeps track of the top element in the stack initially, when the stack is empty, TOP has a value of “-1” and so on.

- Each time a new element is inserted in the stack, the pointer is incremented by “one” before the element is placed on the stack. The pointer is decremented by “one” each time a deletion is made from the stack.

## Implementation of Stack

There are two ways to implement a stack:

1. Using Array
2. Using Linked List

## Array Implementation of Stack

- A stack data structure can be implemented using a one dimensional array
- Stack implemented using array stores only a fixed number of data values.
- Insert or delete the values into that array by using a variable called '**top**'. Initially, the top is set to -1.
- Whenever we want to insert a value into the stack, increment the top value by one and then insert.
- Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

## Operation on Stack

- MakeEmpty – create an empty stack.
- peek – get the top data element of the stack, without removing it.
- IsEmpty - check if stack is empty.
- IsFull – Check if stack is full.
- Push - insert an element into a stack.
- Pop - delete an element from a stack.

## Peek

Peek function is used to return the top element in the stack.

```
int peek() {  
    return stack[top];  
}
```

If an element is present in the stack it returns the position of top element otherwise it returns empty of stack.

## isEmpty

isEmpty() method is used to check and verify if a Stack is empty or not. It returns True if the Stack is empty else it returns False.

```
bool isEmpty() {  
    if(top == -1)  
        return true;  
    else  
        return false;  
}
```

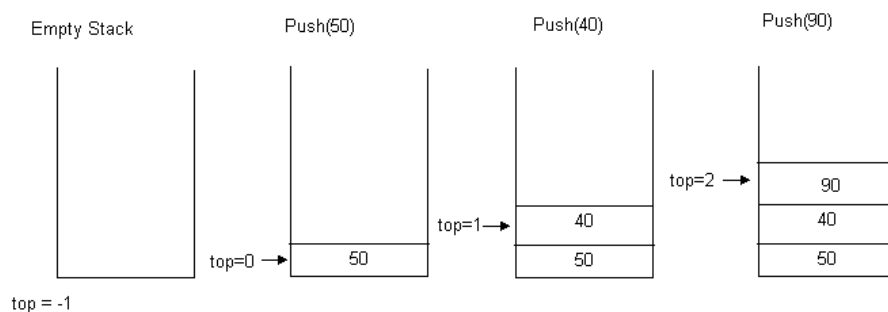
## isFull

IsFull method is used to check and verify if a Stack is full or not. it returns true if the stack is full else it returns False.

```
bool IsFull() {  
    if(top == MAXSIZE)  
        return true;  
    else  
        return false;  
}
```

## Push operation

If the elements are added continuously to the stack using the push operation then the stack grows at one end. Initially when the stack is empty the top = -1. The top is a variable which indicates the position of the topmost element in the stack

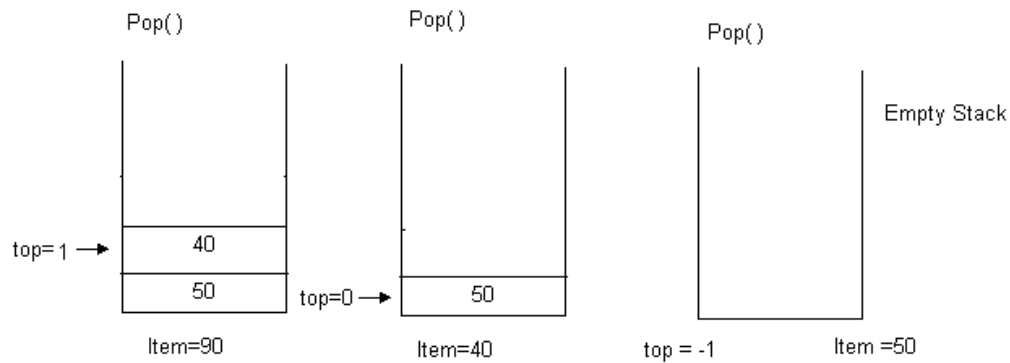


### Void Push(x)

```
    if top == MAX - 1  
    then  
        print "Stack is full"  
    return  
    else  
        top = top + 1  
        A[top] = x  
    End if  
End Push()
```

## POP Operation

Accessing the content while removing it from stack, is known as pop operation. In array implementation of pop operation, data element is not actually removed, instead **top** is decremented to a lower position in stack to point to next value. But in linked-list implementation, pop actually removes data element and deallocates memory space.



```
int Pop( )
if top == -1
then
print "Stack is empty"
return
else
item = A[top]
top = top - 1
return item
End if
End Pop()
```

## Linked List Implementation of Stack

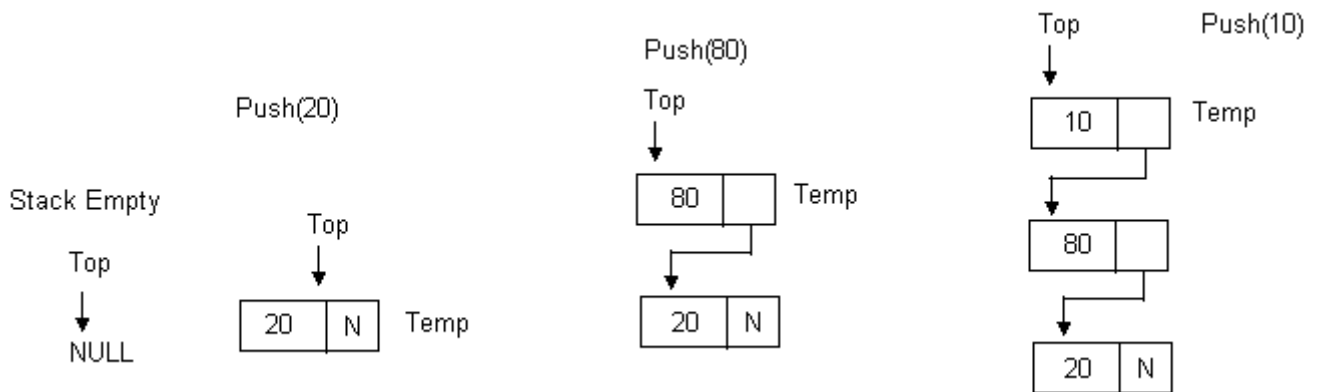
Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek. In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack.

Initially, when the stack is empty `top` pointer points to NULL. When an element is added using the push operation, `top` is made to point to the latest element whichever is added.

## Push Operation

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.



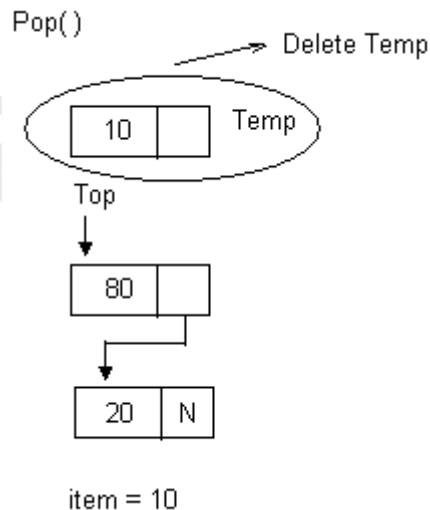
```

PUSH(x)
temp ->data=x
temp->next= top
top=temp
End PUSH()

```

### Pop Operation

The data in the topmost node of the stack is first stored in available called item. Then a temporary pointer is created to point to top. The top is now safely moved to the next node below it in the stack. Temp node is deleted and the item is returned.



```

POP()
if top = NULL then
    print "Stack is empty"
    return
else
    item = top->data

```

```
temp = top
top = top->next
delete temp
return item
End if
End POP()
```

## Application of Stack

- Evaluation of Arithmetic operation
- Conversion of infix to postfix expression
- Evaluation of postfix expression

## Evaluation of Arithmetic expression

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations. These notations are –

- Infix Notation - operators are used **in-between** operands. For example **a+b**
- Prefix (Polish) Notation - operator is **prefixed** to operands, i.e. operator is written ahead of operands. For example, **+ab**.
- Postfix (Reverse-Polish) Notation - the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

- Precedence - When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

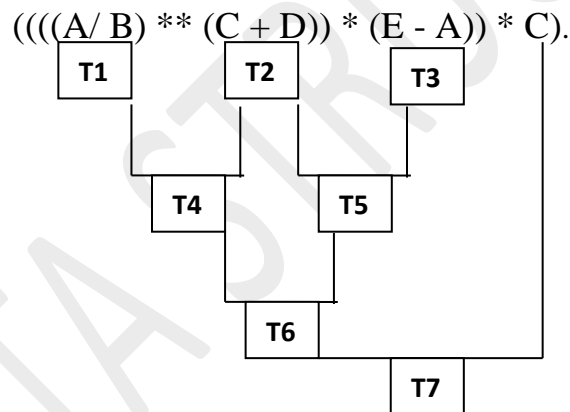
$$a + b * c \quad \rightarrow \quad a + ( b * c )$$

- Associativity - Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression  $a + b - c$ , both  $+$  and  $-$  have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both  $+$  and  $-$  are left associative, so the expression will be evaluated as  $(a + b) - c$ .

**Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –**

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication ( * ) & Division ( / )	Second Highest	Left Associative
3	Addition ( + ) & Subtraction ( - )	Lowest	Left Associative

To specify the latter order of evaluation by using parentheses:  $X (((A/B) ** (C + D)) * (E - A)) * C$ . To fix the order of evaluation, we assign to each operator a priority. Then within any pair of parentheses we understand that operators with the highest priority will be evaluated first.



Procedure EVAL(E)

Top=0

X=NEXT\_TOKEN(E)

Case

: x= '∞' : **return**

: x is an operand: **call Push(x)**

: else: remove the correct number of operands

For operator x from STACK, perform



The operations and store the result, if

Any, onto the stack

End

Forever

End EVAL

## Conversion of infix to postfix expression

The function to convert an expression from infix to postfix consists following steps:

1. Every character of the expression string is scanned in a while loop until the end of the expression is reached.
2. Following steps are performed depending on the type of character scanned.
  - (a) If the character scanned happens to be a space then that character is skipped.
  - (b) If the character scanned is a digit or an alphabet, it is added to the target string pointed to by t.
  - (c) If the character scanned is a closing parenthesis then it is added to the stack by calling push( ) function.
  - (d) If the character scanned happens to be an operator, then firstly, the topmost element from the stack is retrieved. Through a while loop, the priorities of the character scanned and the character popped 'opr' are compared. Then following steps are performed as per the precedence rule.
    - i. If 'opr' has higher or same priority as the character scanned, then opr is added to the target string.
    - ii. If opr has lower precedence than the character scanned, then the loop is terminated. Opr is pushed back to the stack. Then, the character scanned is also added to the stack.
  - (e) If the character scanned happens to be an opening parenthesis, then the operators present in the stack are retrieved through a loop. The loop continues till it does not encounter a closing parenthesis. The operators popped, are added to the target string pointed to by t.
2. Now the string pointed by t is the required postfix expression.

### Example

**Expression:**  $A * (B + C * D) + E$  becomes  $A B C D * + * E +$

	Current symbol	Operator Stack	Postfix string
1	A		A

2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* ( +	A B
6	C	* ( +	A B C
7	*	* ( + *	A B C
8	D	* ( + *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

## Evaluation of Postfix expression

### Algorithm for evaluation of Postfix Expression

```

Initialize(Stack S)
x = ReadToken();
while(x)
{
If(x is Operand)
Push(x) onto Stack S;
If(x is Operator)
{
value2=Pop();
value1=Pop();

```










Result = value1 operator value2;

Push(Result);

}

x=ReadNextToken();

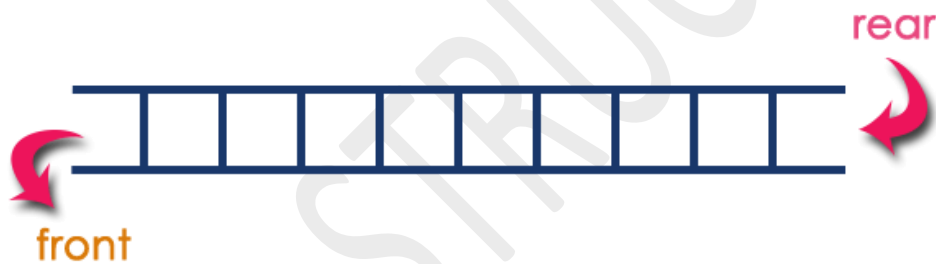
}

Infix Expression <b>(5 + 3) * (8 - 2)</b>		
Postfix Expression <b>5 3 + 8 2 - *</b>		
Above Postfix Expression can be evaluated by using Stack Data Structure as follows...		
Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing
+	value1 = pop() value2 = pop() result = value2 + value1 push(result) 	value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push( 8 ) <b>(5 + 3)</b>
8	push(8) 	(5 + 3)
2	push(2) 	(5 + 3)
-	value1 = pop() value2 = pop() result = value2 - value1 push(result) 	value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push( 6 ) <b>(8 - 2)</b> <b>(5 + 3) , (8 - 2)</b>
*	value1 = pop() value2 = pop() result = value2 * value1 push(result) 	value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push( 48 ) <b>(6 * 8)</b> <b>(5 + 3) * (8 - 2)</b>
\$ End of Expression	result = pop() 	Display (result) <b>48</b> As final result
Infix Expression <b>(5 + 3) * (8 - 2) = 48</b>		
Postfix Expression <b>5 3 + 8 2 - *</b> value is <b>48</b>		

# QUEUE ADT

## What is a Queue?

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- In a queue data structure, adding and removing elements are performed at two different positions.
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.
- In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.
- In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".



## Example

Queue after inserting 25, 30, 51, 60 and 85.

After Inserting five elements...



## Operations on a Queue

The following operations are performed on a queue data structure...

1. enQueue(value) - (To insert an element into the queue)
2. deQueue() - (To delete an element from the queue)
3. display() - (To display the elements of the queue)

## Implementation of Queue

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
  2. **Using Linked List**
- When a queue is implemented using an array, that queue can organize an only limited number of elements.
  - When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

## Queue Data Structure Using Array

- A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values.
- The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** 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

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1** - Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- **Step 2** - Declare all the **user defined functions** which are used in queue implementation.
- **Step 3** - Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4** - Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)

- **Step 5** - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

```
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
```

### EnQueue(value) - 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 takes one integer value as a parameter 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 the function.
- **Step 3** - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear] = value**.

```
void enQueue(int value)
{
    if(rear == SIZE-1)
        cout<<"\nQueue is Full!!! Insertion is not possible!!!";
    else{
        if(front == -1)
            front = 0;
        rear++;
        queue[rear] = value;
        cout<<"\nInsertion success!!!";
    }
}
```

## 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 == rear**)
- Step 2 - If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. 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**).

```
void deQueue(){
    if(front == rear)
        cout<<"\nQueue is Empty!!! Deletion is not possible!!!";
    else{
        cout<<"\nDeleted : %d", queue[front]);
        front++;
        if(front == rear)
            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 == rear**)
- Step 2 - If it is **EMPTY**, then display "**Queue is EMPTY!!!**" and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then define an integer variable **'i'** and set **'i = front+1'**.
- Step 4 - Display **'queue[i]'** value and increment **'i'** value by one (**i++**). Repeat the same until **'i'** value reaches to **rear** (**i <= rear**).

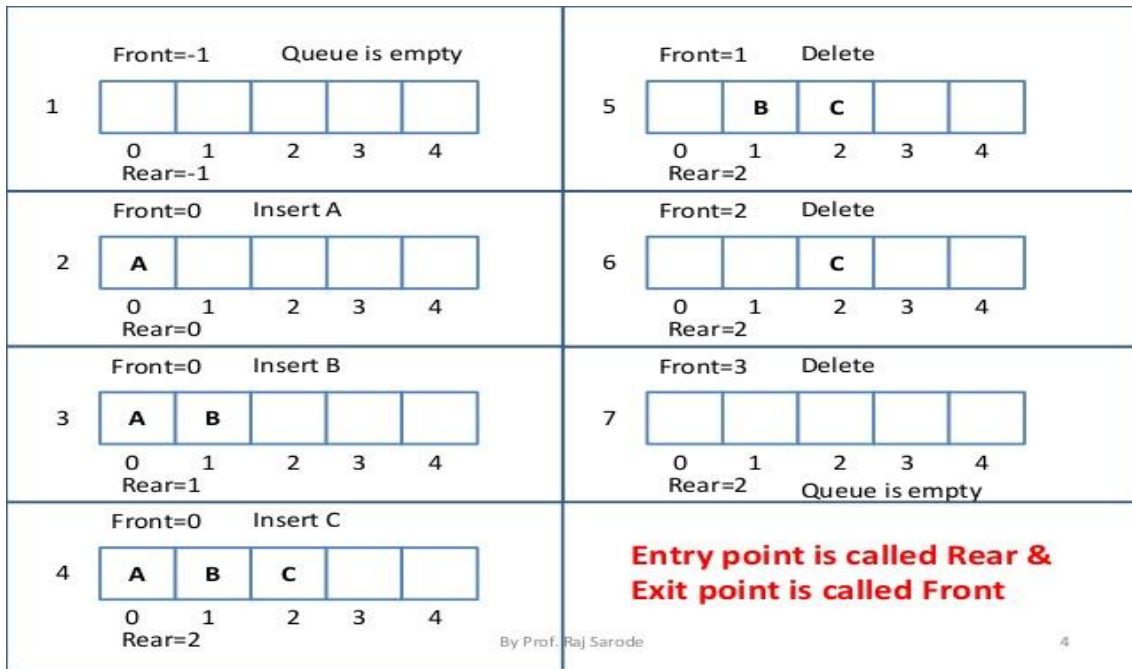
```
void display(){
    if(rear == -1)
        cout<<"\nQueue is Empty!!!";
    else{
        int i;
        cout<<"\nQueue elements are:\n";
```

```

for(i=front; i<=rear; i++)
    cout<<"\t"<<queue[i];
}
}

```

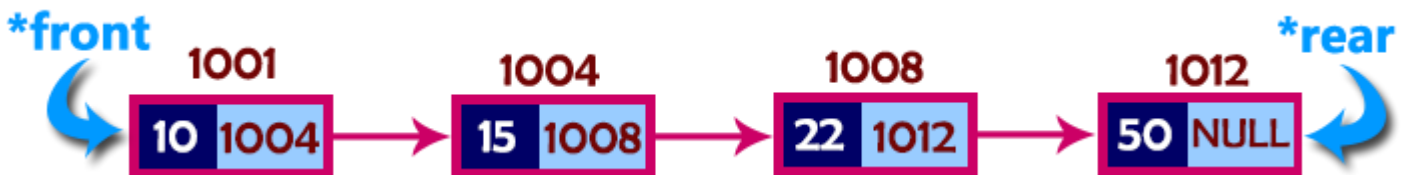
**Example**



**Queue Using Linked List**

- The major problem with the queue implemented using an array is, It 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 which is implemented using a linked list can work for an unlimited number of values
- 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'.

**Example**





In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

## Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1** - Define a '**Node**' structure with two members **data** and **next**.
- **Step 2** - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 3** - declare all the **user defined functions**.

```
struct Node
{
    int data;
    struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();
```

## 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 **newNode** with given value and set '**newNode** → **next**' to **NULL**.
- Step 2 - Check whether queue is **Empty** (**rear == NULL**)
- Step 3 - If it is **Empty** then, set **front = newNode** and **rear = newNode**.
- Step 4 - If it is **Not Empty** then, set **rear** → **next = newNode** and **rear = newNode**.

```
void insert(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode -> next = NULL;
    if(front == NULL)
        front = rear = newNode;
```

```

else{
    rear -> next = newNode;
    rear = newNode;
}
cout<<"\nInsertion is Success!!!\n";
}

```

## deQueue() - Deleting an Element from Queue

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

- Step 1 - Check whether **queue** is **Empty** (**front == NULL**).
- Step 2 - If it is **Empty**, then display "**Queue is Empty!!! Deletion is not possible!!!**" and terminate from the function
- Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- Step 4 - Then set '**front = front → next**' and delete '**temp**' (**free(temp)**).

```

void delete()
{
    if(front == NULL)
        cout<<"\nQueue is Empty!!!\n";
    else{
        struct Node *temp = front;
        front = front -> next;
        cout<<"\nDeleted element: %d\n", temp->data;
        free(temp);
    }
}

```

## Circular Queue Data structure

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

The queue after inserting all the elements into it is as follows...

## Queue is Full



Now consider the following situation after deleting three elements from the queue...

## Queue is Full (Even three elements are deleted)

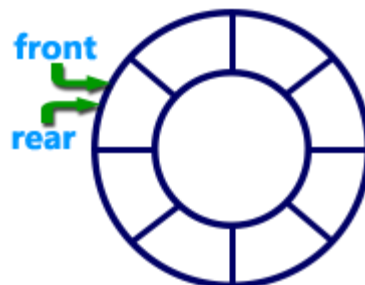


- This situation also says that Queue is Full and we cannot insert the new element because '**rear**' is still at last position. In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element.
- This is the major problem in a normal queue data structure. To overcome this problem we use a circular queue data structure.

## What is Circular Queue

A Circular queue is a linear data structure in which the operations are performed based on FIFO principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows



## Implementation of Circular Queue

To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

- **Step 1** - Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- **Step 2** - Declare all **user defined functions** used in circular queue implementation.
- **Step 3** - Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4** - Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)
- **Step 5** - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

### 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. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue.

We can use the following steps to insert an element into the circular queue

- **Step 1** - Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+1)**)
- **Step 2** - If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible!!!**" and terminate the function.
- **Step 3** - If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- **Step 4** - Increment **rear** value by one (**rear++**), set **queue[rear] = value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

```
void enQueue(int value)
{
    if((front == 0 && rear == SIZE - 1) || (front == rear+1))
        printf("\nCircular Queue is Full! Insertion not possible!!!\n");
    else{
        if(rear == SIZE-1 && front != 0)
            rear = -1;
        cQueue[++rear] = value;
```

```

        printf("\nInsertion Success!!!\n");
        if(front == -1)
            front = 0;
    }
}

```

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

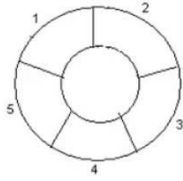
In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular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

- Step 1 - Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- Step 2 - If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. 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 = rear = -1**).

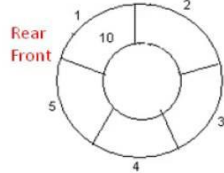
```

void deQueue()
{
    if(front == -1 && rear == -1)
        printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
    else{
        printf("\nDeleted element : %d\n",cQueue[front++]);
        if(front == SIZE)
            front = 0;
        if(front-1 == rear)
            front = rear = -1;
    }
}

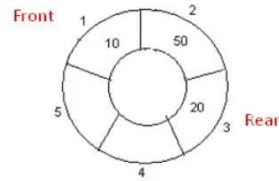
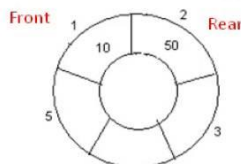
```



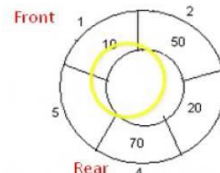
2. Insert 10, Rear = 1, Front = 1.



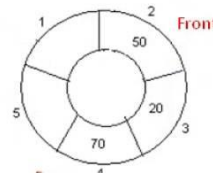
3. Insert 50, Rear = 2, Front = 1.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.



## Multiple Stacks

When a stack is created using single array, we cannot able to store large amount of data, thus this problem is rectified using more than one stack in the same array of sufficient array. This technique is called as Multiple Stack.

When an array of STACK[n] is used to represent two stacks, say Stack A and Stack B. Then the value of n is such that the combined size of both the Stack[A] and Stack[B] will never exceed n. Stack[A] will grow from left to right, whereas Stack[B] will grow in opposite direction (ie) right to left.

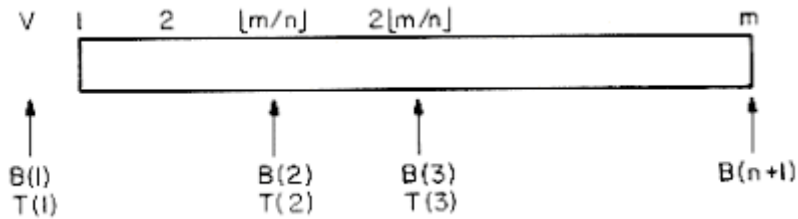
When more than two stacks, say  $n$ , are to be represented sequentially, we can initially divide out the available memory  $V(1:m)$  into  $n$  segments and allocate one of these segments to each of the  $n$  stacks.

For each stack  $i$  we shall use  $B(i)$  to represent a position one less than the position in  $V$  for the bottommost element of that stack.  $T(i)$ ,  $1 \leq i \leq n$  will point to the topmost element of stack  $i$ .

We shall use the boundary condition  $B(i) = T(i)$  iff the  $i$ 'th stack is empty. If we grow the  $i$ 'th stack in lower memory indexes than the  $i + 1$ 'st, then with roughly equal initial segments we have

$$B(i) = T(i) = m/n (i - 1), 1 \leq i \leq n$$

Initially,  $\text{boundary}[i]=\text{top}[i]$ .



### **procedure ADD( $i, X$ )**

//add element  $X$  to the  $i$ 'th stack,  $1 \leq i \leq n$ //

**if**  $T(i) = B(i + 1)$  **then call** *STACK-FULL* ( $i$ )

$T(i) = T(i) + 1$

$V(T(i)) = X$  //add  $X$  to the  $i$ 'th stack//

**end** *ADD*

### **procedure DELETE( $i, X$ )**

//delete topmost element of stack  $i$ //

**if**  $T(i) = B(i)$  **then call** *STACK-EMPTY* ( $i$ )

$X = V(T(i))$

$T(i) = T(i) - 1$

**end** *DELETE*

## **Representation of Polynomial**

A polynomial  $p(x)$  is the expression in variable  $x$  which is in the form  $(ax^n + bx^{n-1} + \dots + jx^k)$ , where  $a, b, c, \dots, k$  fall in the category of real numbers and ' $n$ ' is non negative

integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

```

typedef struct node *nptr;
struct node
{
    int coef;
    int expo;
    nptr next;
};

```

**Creation of polynomial list and Addition of Polynomial Lists:**

**Algorithm: Creation of polynomial list for n terms**

1. Create head node.
2. Read the number of terms in an expression (read n)
3. Repeat the following steps for n terms.
4. Create new node and add new node to list.

```

new=(nptr)(malloc(sizeof(struct node)));

new->coef=co;

new->expo=pow;

new->next=temp->next;

temp->next=new;

temp=new;

```

**Example:**

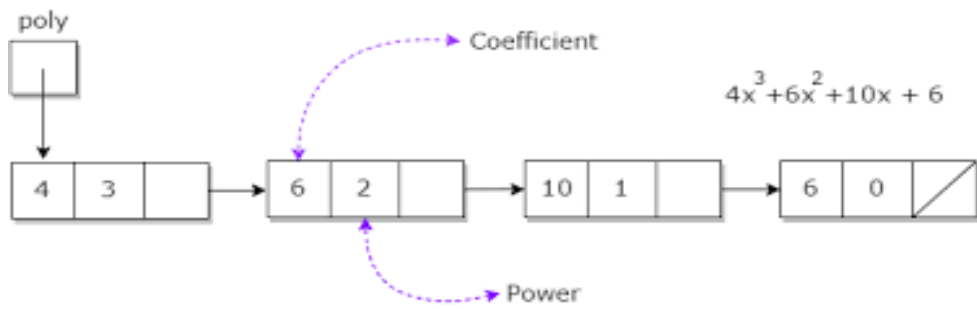
$10x^2 + 26x$ , here 10 and 26 are coefficients and 2, 1 is its exponential value.

Each term will be represented by a node. A node will be of fixed size having 3 fields which represent the coefficient and exponent of a term and a pointer to the next term.

COEF	EXP	LINK
------	-----	------

A linked list structure that represents polynomials  $4x^3 + 6x^2 + 10x + 6$



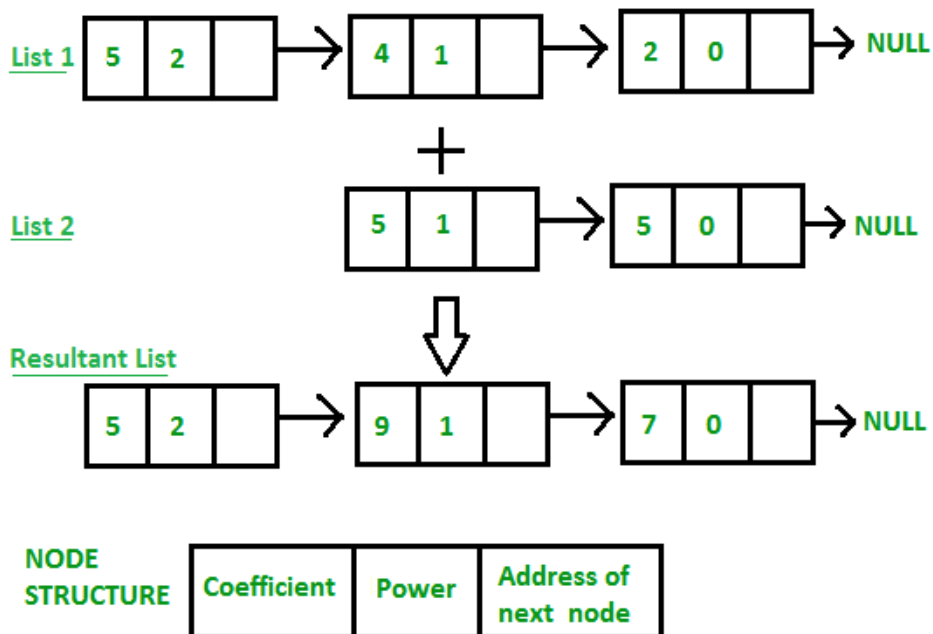


## Polynomial Addition

Input:

1st number =  $5x^2 + 4x^1 + 2x^0$

2nd number =  $5x^1 + 5x^0$



Algorithm:

Let p and q be the two polynomials represented by linked lists.

Let r represent result list of the addition of two polynomials

1. while p and q are not null, repeat step 2.

2. If powers of the two terms are equal

then insert the sum of the terms into the sum Polynomial

if(p ->expo==q->expo)

{

```
r->co-eff=p->co-eff+q->co-eff
```

```
r->expo=p->expo
```

```
p=p->next
```

```
q=q->next
```

```
}
```

Else if the expo of the polynomial  $p >$  expo of polynomial  $q$

Then insert the term from polynomial  $p$  into sum polynomial

Advance  $p$

```
Else if(p->expo>q->expo)
```

```
{
```

```
r->co-eff=p->co-eff
```

```
r->expo=p->expo
```

```
p=p->next
```

```
}
```

Else insert the term from  $q$  polynomial into sum polynomial

Advance  $q$

```
{
```

```
r->co-eff=q->co-eff
```

```
r->expo=q->expo
```

```
q=q->next
```

```
}
```

3. copy the remaining terms from the non empty polynomial into the sum polynomial. The

third step of the algorithm is to be processed till the end of the polynomials has not been

reached.

```
While(p!=NULL)
{
r->co-eff=p->co-eff
r->expo=p->expo
p=p->next
}
While(q!=NULL)
{
r->co-eff=q->co-eff
r->expo=q->expo
q=q->next
}
```

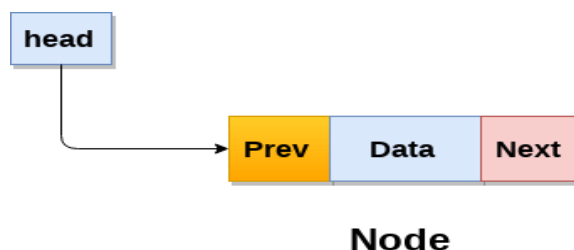
## Double Linked List

What is Double Linked List?

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list.

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



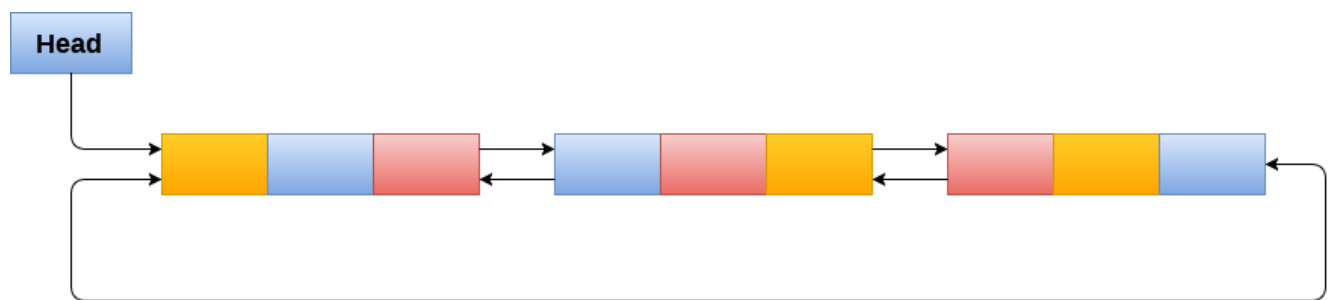
## declarations:

```
typedef struct node
*node_pointer;
typedef struct node{
node_pointer llink;
element
data;
node_pointer rlink;
};
```

## Doubly linked circular list with header node

Header node allows us to implement operations more easily.

```
ptr = ptr->llink->rlink = ptr->rlink->llink
```

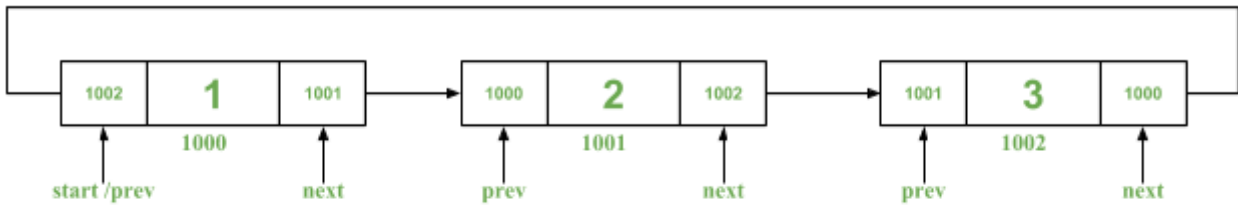


## Circular Doubly Linked List

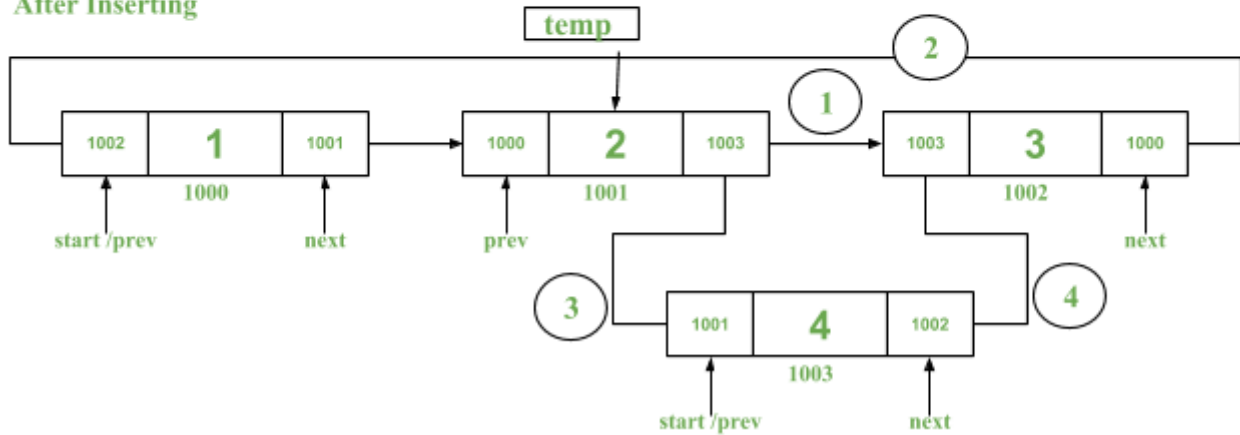
### Insertion into A Doubly Linked Circular List

```
Void dinsert(node_pointer node, node_pointer newnode)
{
/*insert newnode to the right of node */
newnode->llink =node;
newnode->rlink =node->rlink;
node->rlink->llink =newnode;
node->rlink =newnode;
}
```

## Linked List | Insert At Location 3:



After Inserting

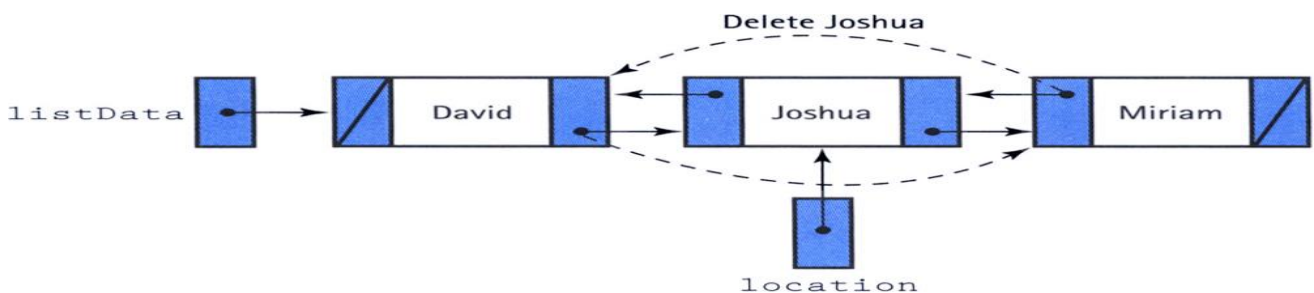


## Deletion from A Doubly Linked Circular List

```

Void ddelete(node_pointer node, node_pointer deleted)
{
/*delete from the doubly linked list */
if(node==deleted)
printf("Deletion of head node not permitted. \n");
else
{
deleted->llink->rlink = deleted->rlink;
deleted->rlink->llink =deleted->llink;
free(deleted);
}
}

```



## Dynamic Storage Management

**Dynamic Memory Allocation.** **Dynamic memory allocation** is when an executing program requests that the operating system give it a block of main **memory**. The program then uses this **memory** for some purpose. Usually the purpose is to add a node to a **data structure**.

DATA STRUCTURE