

UNIT - 2

STACKS AND QUEUES

STACK

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- 1.5. *Push*: is the term used to insert an element into a stack.
- 1.6. *Pop*: is the term used to delete an element from a stack.

“Push” is the term used to insert an element into a stack. “Pop” is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by `push()`. Figure 4.1 shows the creation of a stack and addition of elements using `push()`.

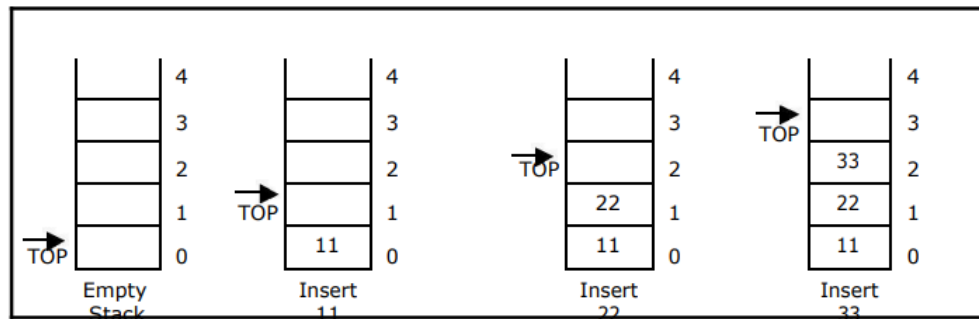


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by `pop()`. Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using `pop()`.

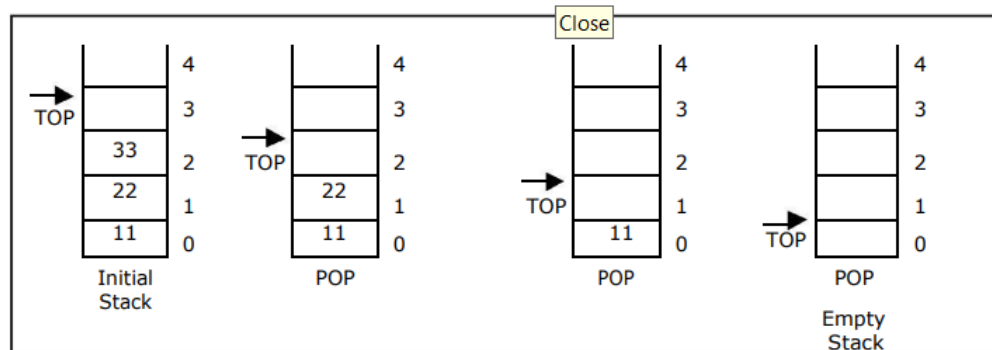


Figure 4.2. Pop operations on stack

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, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

peek()

Algorithm of peek() function –

```
begin procedure peek
  return stack[top]
end procedure
```

Implementation of peek() function in C programming language –

Example

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

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

  if top equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

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

isempty()

Algorithm of isempty() function –

```
begin procedure isempty
```

```
    if top less than 1
```

```
        return true
```

```
    else
```

```
        return false
```

```
    endif
```

```
end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

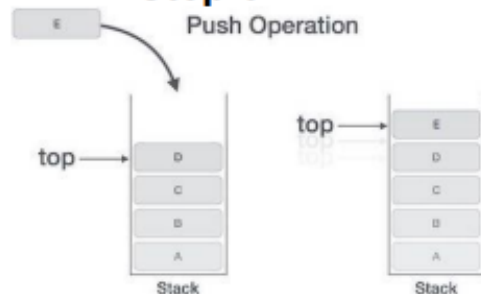
Example

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

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data
```

```
if stack is full
```

```
    return null  
endif  
  
top ← top + 1  
stack[top] ← data
```

```
end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

Example

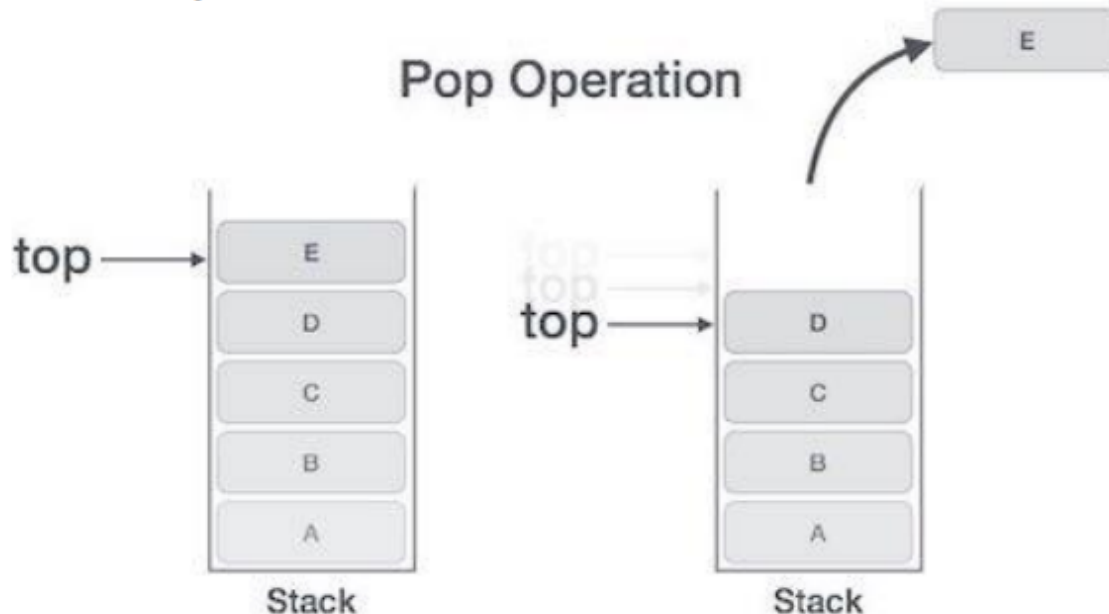
```
void push(int data) {  
    if(!isFull()) {  
        top = top + 1;  
        stack[top] = data;  
    } else {  
        printf("Could not insert data, Stack is full.\n");  
    }  
}
```

Pop Operation

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

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack
```

```
    if stack is empty  
        return null  
    endif
```

```
    data ← stack[top]  
    top ← top - 1  
    return data
```

```
end procedure
```

Implementation of this algorithm in C, is as follows –

Example

```
int pop(int data) {  
  
    if(!isempty()) {  
        data = stack[top];  
        top = top - 1;  
        return data;  
    } else {  
        printf("Could not retrieve data, Stack is empty.\n");  
    }  
}
```

// C program for array implementation of stack

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
// A structure to represent a stack
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
```

// Stack is full when top is equal to the last index

```
int isFull(struct Stack* stack)
{
    return stack->top == stack->capacity - 1;
}
```

// Stack is empty when top is equal to -1

```
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}
```

// Function to add an item to stack. It increases top by 1

```
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}
```

// Function to remove an item from stack. It decreases top by 1

```
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}
```

// Function to return the top from stack without removing it

```
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}
```

// Driver program to test above functions

```
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n",
pop(stack));

    return 0;
}
```

Output

10 pushed into stack

20 pushed into stack

30 pushed into stack

30 Popped from stack

Top element is : 20

Elements present in stack : 20 10

Applications of Stack

- Expression Parsing: Stacks help evaluate and check programming expressions, ensuring balanced parentheses.
- Backtracking: Used in algorithms like maze-solving and the "Eight Queens" puzzle.
- Function Calls: Manage function details during calls in programming languages.
- Undo Feature: Implement undo in text editors and browsers.
- Syntax Checking: Compilers use stacks to match syntax elements like 'if' with 'else'.

Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x , y , z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include $+$, $-$, $*$, $/$, $^$ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: $(A + B) * (C - D)$

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: $* + A B - C D$

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: $A B + C D - *$

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: $+$, $-$, $*$, $/$ and $\$$ or $\uparrow$ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($\$$ or $\uparrow$ or $\wedge$)	Highest	3
$*$, $/$	Next highest	2
$+$, $-$	Lowest	1

Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

- 1 Infix to postfix
- 2 Infix to prefix
- 3 Postfix to infix
- 4 Postfix to prefix
- 5 Prefix to infix
- 6 Prefix to postfix

Conversion from infix to postfix:

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).

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.

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 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.

Example 1:

Convert $((A - (B + C)) * D) \uparrow (E + F)$ infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

- Conversion from infix to prefix:**

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression $A + B - C$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example

Convert the infix expression $A \uparrow B * C - D + E / F / (G + H)$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	//	
E	E F + G H	//	
+	// E F + G H	+	
D	D // E F + G H	+	
-	D // E F + G H	+ -	
C	C D // E F + G H	+ -	
*	C D // E F + G H	+ - *	
B	B C D // E F + G H	+ - *	
$\uparrow$	B C D // E F + G H	+ - * $\uparrow$	
A	A B C D // E F + G H	+ - * $\uparrow$	
End of string	+ - * $\uparrow$ A B C D // E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion from postfix to infix:

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

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression $A\ B\ C\ *\ D\ E\ F\ \wedge\ /\ G\ *\ -\ H\ *\ +$ into its equivalent infix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) - ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) - ((D/(E^F))*G)) H	Push H
*	A (((B*C) - ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) - ((D/(E^F))*G)) * H))	

End of
string

The input is now empty. The string formed is infix.

Conversion from postfix to prefix:

Procedure to convert postfix expression to prefix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in front of the operands and push it onto the stack.
5. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression $A B C * D E F ^ / G * - H * +$ into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A *BC	Pop two operands and place the operator in front the operands and push the string.
D	A *BC D	Push D
E	A *BC D E	Push E
F	A *BC D E F	Push F
^	A *BC D ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A *BC /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A *BC /D^EF G	Push G
*	A *BC */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A - *BC*/D^EFG H	Push H
*	A *- *BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*- *BC*/D^EFGH	

End of
string

The input is now empty. The string formed is prefix.

Conversion from prefix to infix:

Procedure to convert prefix expression to infix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression $+ A * - * B C * / D ^ E F G H$ into its equivalent infix expression.

Symbol

Stack

Remarks

H	H					Push H
G	H	G				Push G
F	H	G	F			Push F
E	H	G	F	E		Push E
^	H	G	(E^F)			Pop two operands and place the operator in between the operands and push the string.
D	H	G	(E^F)	D		Push D
/	H	G	(D/(E^F))			Pop two operands and place the operator in between the operands and push the string.
*	H	((D/(E^F))*G)				Pop two operands and place the operator in between the operands and push the string.
C	H	((D/(E^F))*G)	C			Push C
B	H	((D/(E^F))*G)	C	B		Push B
*	H	((D/(E^F))*G)	(B*C)			Pop two operands and place the operator in front the operands and push the string.
-	H	((B*C)-((D/(E^F))*G))				Pop two operands and place the operator in front the operands and push the string.
*	((B*C)-((D/(E^F))*G))*H					Pop two operands and place the operator in front the operands and push the string.
A	((B*C)-((D/(E^F))*G))*H	A				Push A
+	(A+((B*C)-((D/(E^F))*G))*H)					Pop two operands and place the operator in front the operands and push the string.

End of string

The input is now empty. The string formed is infix.

Conversion from prefix to postfix:

Procedure to convert prefix expression to postfix expression is as follows:

- Scan the prefix expression from right to left (reverse order).
- If the scanned symbol is an operand, then push it onto the stack.
- If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator after the operands and push it onto the stack.
- Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression $+ A * - * B C * / D ^ E F G H$ into its equivalent postfix expression.

Symbol	Stack	Remarks					
H	<table><tr><td>H</td><td></td><td></td><td></td><td></td></tr></table>	H					Push H
H							
G	<table><tr><td>H</td><td>G</td><td></td><td></td><td></td></tr></table>	H	G				Push G
H	G						
F	<table><tr><td>H</td><td>G</td><td>F</td><td></td><td></td></tr></table>	H	G	F			Push F
H	G	F					
E	<table><tr><td>H</td><td>G</td><td>F</td><td>E</td><td></td></tr></table>	H	G	F	E		Push E
H	G	F	E				
^	<table><tr><td>H</td><td>G</td><td>EF^</td><td></td><td></td></tr></table>	H	G	EF^			Pop two operands and place the operator after the operands and push the string.
H	G	EF^					
D	<table><tr><td>H</td><td>G</td><td>EF^</td><td>D</td><td></td></tr></table>	H	G	EF^	D		Push D
H	G	EF^	D				

/	H	G	DEF^/		
*	H	DEF^/G*			
C	H	DEF^/G*	C		
B	H	DEF^/G*	C	B	
*	H	DEF^/G*	BC*		
-	H	BC*DEF^/G*-			
*	BC*DEF^/G*-H*				
A	BC*DEF^/G*-H*	A			
+	ABC*DEF^/G*-H*+				

End of
string

The input is now empty. The string formed is postfix.

Pop two operands and place the operator after the operands and push the string.

Pop two operands and place the operator after the operands and push the string.

Push C

Push B

Pop two operands and place the operator after the operands and push the string.

Pop two operands and place the operator after the operands and push the string.

Pop two operands and place the operator after the operands and push the string.

Push A

Pop two operands and place the operator after the operands and push the string.

Queue:

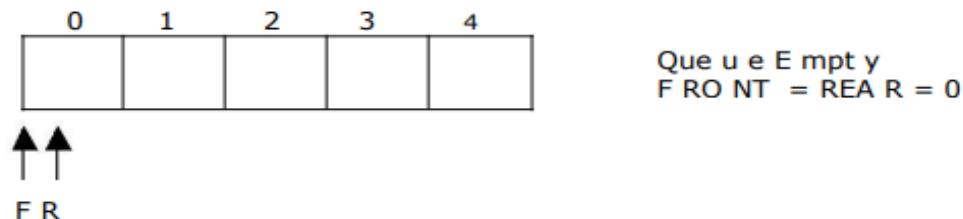
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

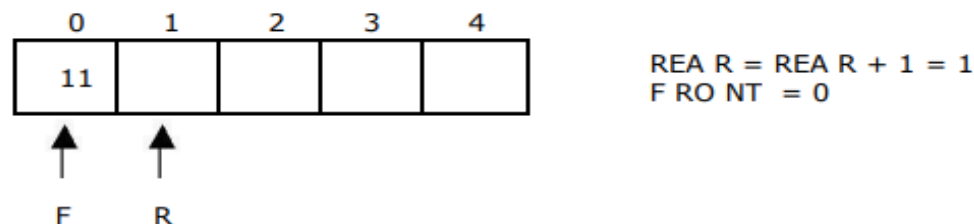
- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

Representation of Queue:

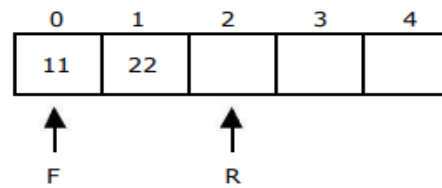
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



Now, insert 11 to the queue. Then queue status will be:

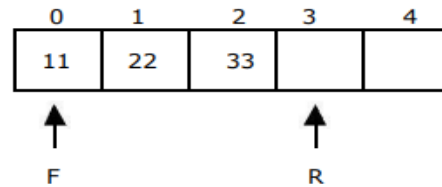


Next, insert 22 to the queue. Then the queue status is:



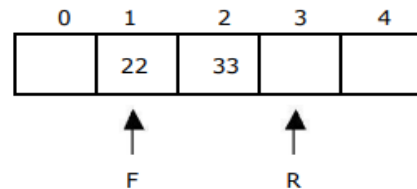
$REA R = REA R + 1 = 2$
 $FRO NT = 0$

Again insert another element 33 to the queue. The status of the queue is:



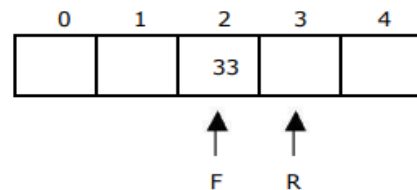
$REA R = REA R + 1 = 3$
 $FRO NT = 0$

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



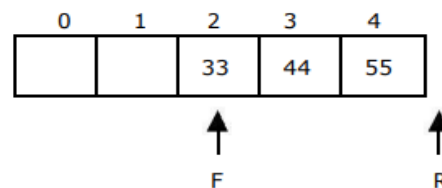
$REA R = 3$
 $FRO NT = FRO NT + 1 = 1$

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



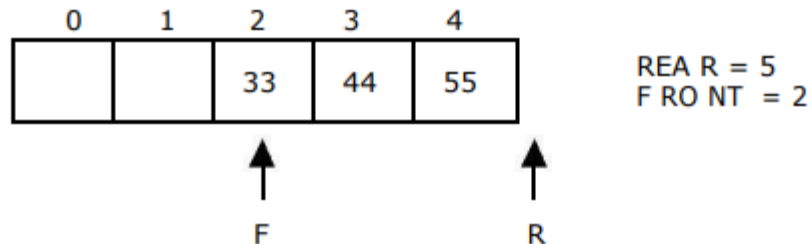
$REA R = 3$
 $FRO NT = FRO NT + 1 = 2$

Now, insert new elements 44 and 55 into the queue. The queue status is:

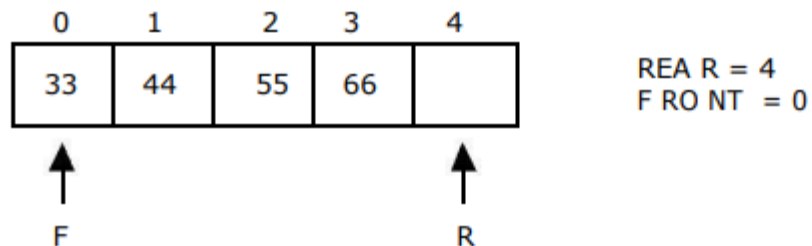


$REA R = 5$
 $FRO NT = 2$

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue**.

Basic Operations in Queue

Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue.

Queue uses two pointers – **front** and **rear**. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

Queue Insertion Operation: Enqueue()

The *enqueue()* is a data manipulation operation that is used to insert elements into the stack. The following algorithm describes the enqueue() operation in a simpler way.

Algorithm

1. START
2. Check if the queue is full.
3. If the queue is full, produce overflow error and exit.
4. If the queue is not full, increment rear pointer to point the next empty space.
5. Add data element to the queue location, where the rear is pointing.
6. return success.
7. END

Queue Deletion Operation: dequeue()

The *dequeue()* is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.

Algorithm

1. START
2. Check if the queue is empty.
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
5. Increment front pointer to point to the next available data element.
6. Return success.
7. END

Queue - The peek() Operation

The peek() is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.

Algorithm

1. START
2. Return the element at the front of the queue
3. END

Queue - The isFull() Operation

The isFull() operation verifies whether the stack is full.

Algorithm

1. START
2. If the count of queue elements equals the queue size,
return true
3. Otherwise, return false
4. END

Queue - The isEmpty() operation

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the count of queue elements equals zero, return true
3. Otherwise, return false
4. END

Queue Complete Implementation

Following are the complete implementations of Queue in various programming languages –

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
int peek(){
    return intArray[front];
}
bool isEmpty(){
    return itemCount == 0;
}
bool isFull(){
    return itemCount == MAX;
}
int size(){
    return itemCount;
}
```

```
void insert(int data){
    if(!isFull()) {
        if(rear == MAX-1) {
            rear = -1;
        }
        intArray[++rear] = data;
        itemCount++;
    }
}
int removeData(){
    int data = intArray[front++];
    if(front == MAX) {
        front = 0;
    }
    itemCount--;
    return data;
}
```

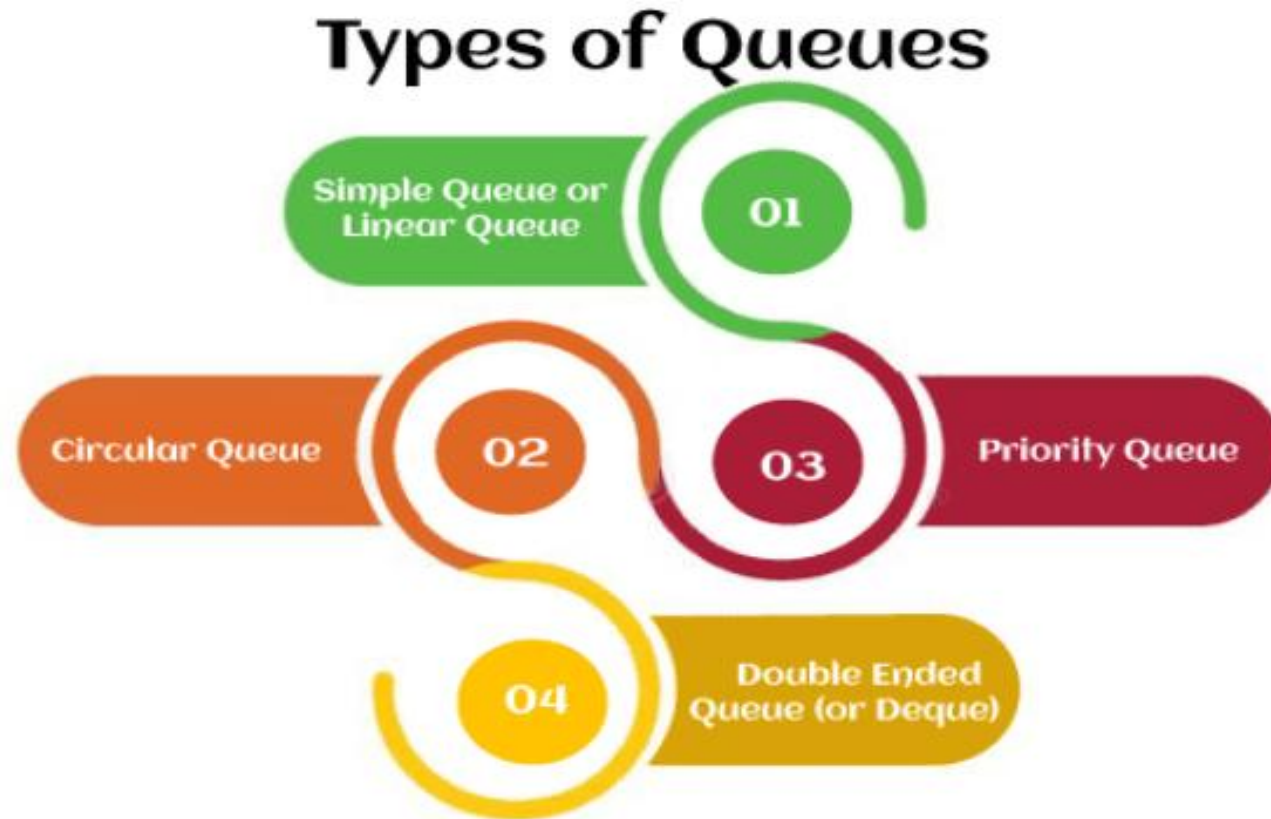
```
int main(){  
    /* insert 5 items */  
    insert(3);  
    insert(5);  
    insert(9);  
    insert(1);  
    insert(12);  
    insert(15);  
    printf("Queue size: %d", size());  
    printf("\nQueue: ");  
    for(int i = 0; i < MAX; i++){  
        printf("%d ", intArray[i]);  
    }  
    if(isFull()) {  
        printf("\nQueue is full!");  
    }  
    // remove one item  
    int num = removeData();  
    printf("\nElement removed: %d", num);  
    printf("\nSize of Queue after deletion: %d", size());  
    printf("\nElement at front: %d", peek());  
}
```

Output

```
Queue size: 6  
Queue: 3 5 9 1 12 15  
Queue is full!  
Element removed: 3  
Size of Queue after deletion: 5  
Element at front: 5
```

Types of Queue

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



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

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

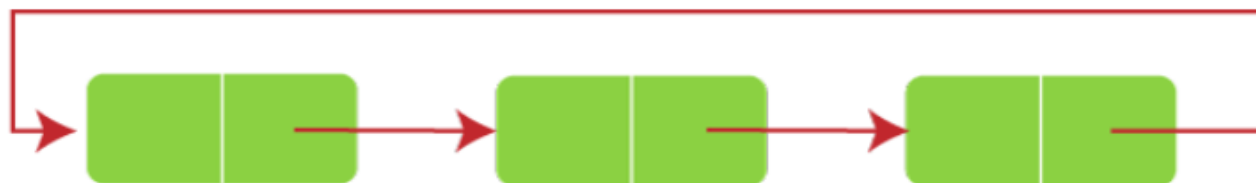
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



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

Circular Queue

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

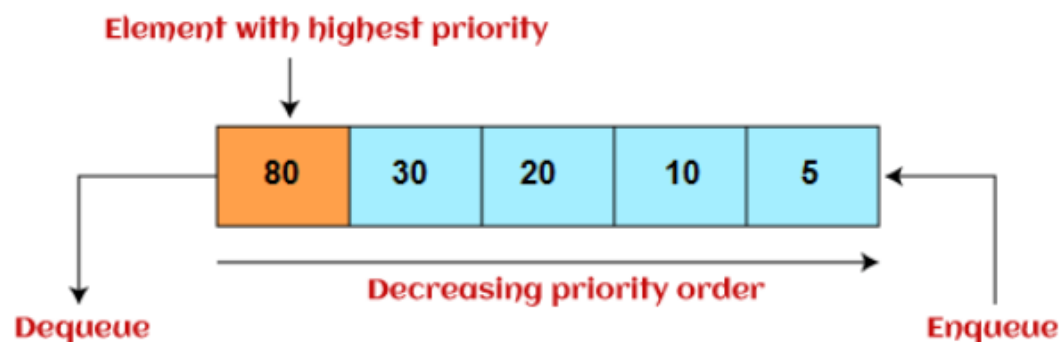


Circular Queue

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

Priority Queue

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



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

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

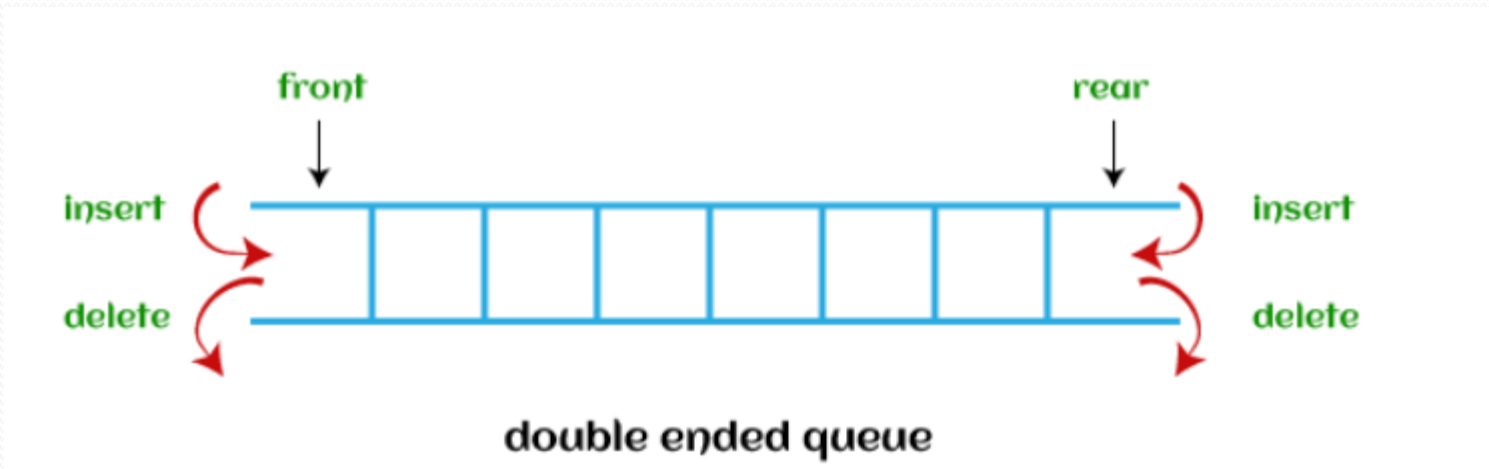
- **Ascending priority queue** - In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is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

Deque (or, Double Ended Queue)

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

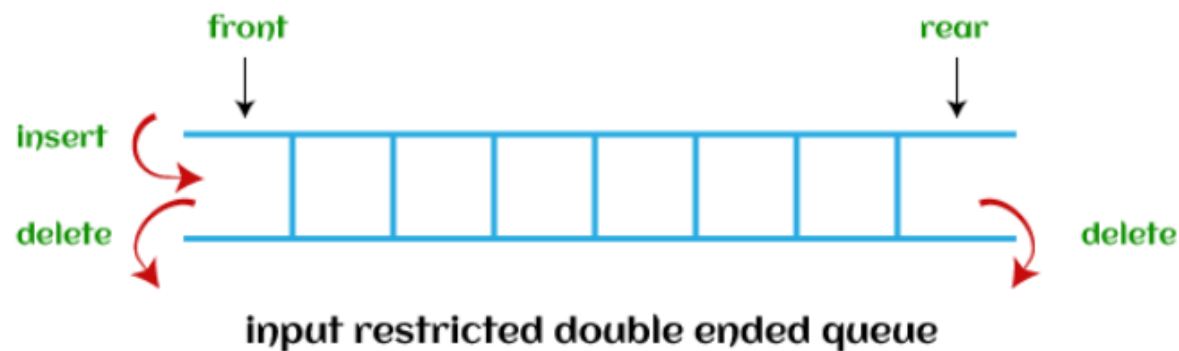
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible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

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

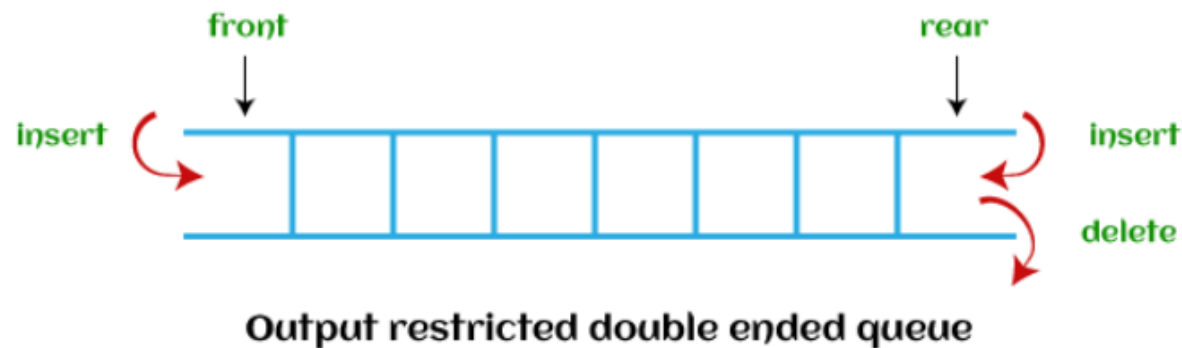


There are two types of deque that are discussed as follows -

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



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





UNIT – 2

COMPLETED