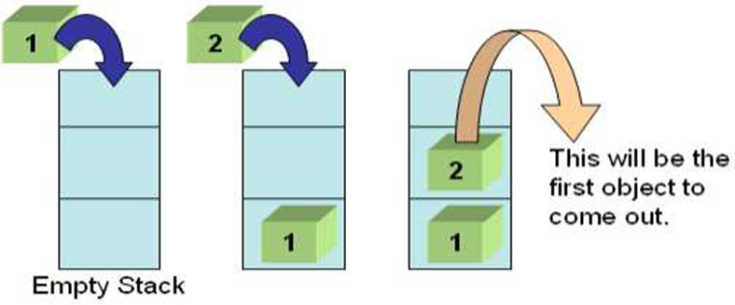
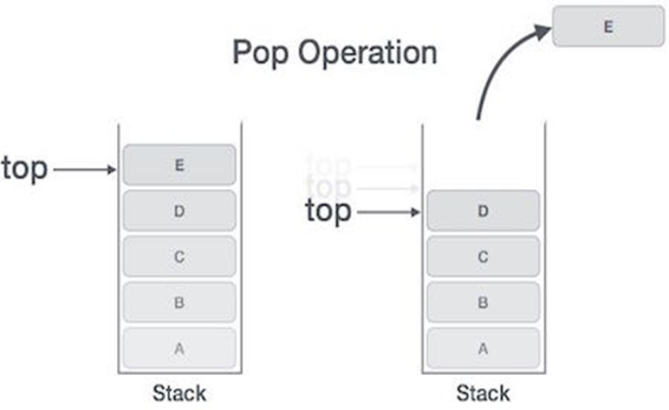
Unit - 3. **Stack & Queues**

**Stack**: -

* A stack is one of the most essential non-primitive linear data structure. Implementation of most of the system programs is based on stack data structure.
* The stack data structure obeys the principle of **Last-In-First-Out (LIFO)**.
* In which the last item pushed onto the stack is always the first to be removed from the stack.
* The end of the stack from where the insertion or deletion operation is carried out is called “**top**” of the stack.



* A stack is an ordered list with the restriction that elements are added or deleted from only one end of  
  the list termed TOP of stack.
* The insertion of element onto the stack is called as PUSH and deletion is called POP.

**Operation on the Stack**: -

1. **Create()**: Creates a new empty stack. With top = -1
2. **Push()** : add new element to top of the stack.
3. **Pop()** : deletes an elements from to of the stack.
4. **isEmpty()**: check whether stack is empty or not
5. **isFull()** : check whether stack is full or not
6. **Some additional operation on stack are:-**
7. **init()** :- initialize the stack pointer.
8. **Display()** : display the contents of stack.
9. **Size()** : display the size of the stack.
10. **Top()** : return the 1st element of the stack.

**Stack as an Abstract Data type**: -

* Stacks can also be defined as Abstract Data Type (ADT). The stack abstract data type is defined by the writing its structure and operations.
* Stacks are ordered LIFO.
* In order to create stack as ADT one can write following functions and their functionality.

1. **CREATE**: This operation creates a new stack. This operation is used to create an empty stack. Initially set the value of Top is - l which is used to show the stack is empty.

Algorithm:

CREATE (Stack)   
**Step 1**: Initialization begins  
 Set TOP=-1  
**Step 2**: Exit

1. **PUSH**: The process of adding a new element to the top of stack is called push operation. After every push operation the top is incremented by one.

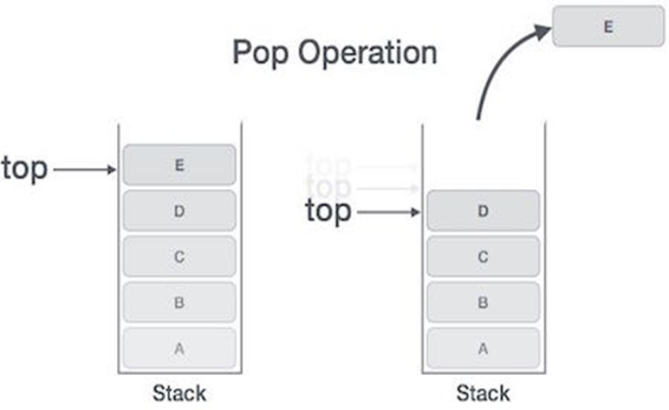
Algorithm:

PUSH (stack, TOP, size, item)   
**Step 1**: Check for stack overflow,  
 If TOP > size  
 message "stack overflow”   
 return  
**Step 2**: Set the position of TOP pointer

Set TOP = TOP + 1  
**Step 3**: Apply insertion  
 stack [TOP] = item  
**Step 4**: Exit

1. **POP**: The process of deleting an element from the top of stack is called pop operation. After every POP operation the top pointer of the stack is decremented by one.

Algorithm:

POP (Stack, TOP, size, item) **Step 1**: Check underflow condition,   
 If TOP = -1 then  
 message "underflow”  
 exit

**Step 2:** Delete the TOP element.

Set item = Stack [TOP].  
**Step 3**: Set the value of TOP  
 Set TOP = TOP - 1  
**Step 4**: Return

1. **isEmpty()**: check whether stack is empty or not, This operation returns TRUE if the stack is empty and false otherwise.
2. **isFull()** : check whether stack is full or not

**Some additional operation on stack are:-**

1. **init()** :- initialize the stack pointer.
2. **Display()** : display the contents of stack.
3. **Size()** : display the size of the stack.
4. **Top()** : return the 1st element of the stack.

**PUSH Operation** : The process of adding a new element to the top of stack is called push operation. After every push operation the top is incremented by one.

Algorithm:

  
PUSH (stack, TOP, size, item)   
**Step 1**: Check for stack overflow,  
 If TOP > size  
 message "stack overflow”   
 return  
**Step 2**: Set the position of TOP pointer

Set TOP = TOP + 1  
**Step 3**: Apply insertion  
 stack [TOP] = item  
**Step 4**: Exit

**PROGRAM FOR PUSH OPERATION IN STACK: -**

#include <stdio.h>  
#define MAX 5 //Maximum number of elements  
int main( )  
{  
int top=-1, stack[MAX],va1, i;  
char ch;  
do  
{  
if(top==MAX-1)  
 printf( "\n Stack is full I ! " );  
else  
{  
 printf("\n Enter element to push:',);  
 scanf( "%d ", &val );  
 top=top+1;  
 stack [top] = val; //element inserted in stack at top

}

printf("Do you want to insert more elements?”);  
scanf(" %c",&ch); //use a space before %c to clean stdin  
}whi1e(ch=='y' || ch=='y' );  
//code for display  
printf ( "\n\*\*\* Elements in Stack\*\*\*\n”);

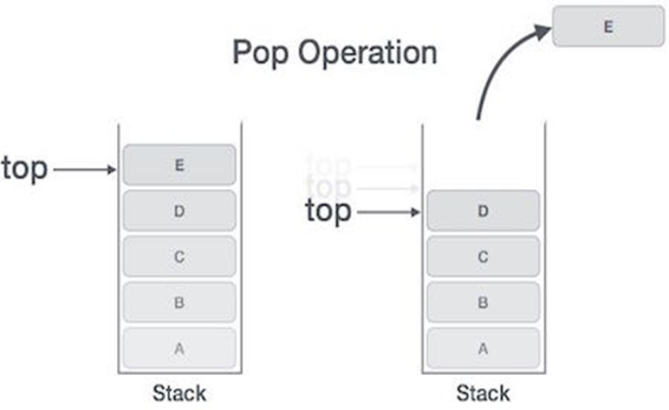
for(i=top; i>=0; i--)  
printf( "%d\n ", stack [i]);

}

Output:  
Enter element to push:10  
Do you want to insert more elements ? y  
Enter element to push:20  
Do you want to insert more elements ? y  
Enter element to push:30  
Do you want to insert more elements ? y  
Enter element to push:40  
Do you want to insert more elements ? n  
\*\*\*Elements in Stack\*\*\*. 40 30 20 10

**POP Operation**: The process of deleting an element from the top of stack is called pop operation. After every POP operation the top pointer of the stack is decremented by one.

Algorithm:

  
POP (Stack, TOP, size, item)   
**Step 1**: Check underflow condition,   
 If TOP = -1 then  
 message "underflow”  
 exit

**Step 2:** Delete the TOP element.

Set item = Stack [TOP].  
**Step 3**: Set the value of TOP  
 Set TOP = TOP - 1  
**Step 4**: Return

**PROGRAM FOR POP OPERATION IN STACK**

#include <stdio.h>  
#define MAX 5 / /t4axinun number of elements that can be stoned  
int main()  
{  
int top=MAX-1;  
int stack[MAX] ={10,20,30,40,50};  
char ch;  
int i;  
//code for display  
printf("\n\*\*\*\* Elements in Stack\*\*\*\*.\n");  
for(i=top; i>=0; i-- )  
printf ( "%d\n", stack[i]);  
//code fon Pop  
printf("\n Do you want to delete element?”);  
scanf("%c",&ch); // use a space before %c to clean stdin  
while(ch=='y' || ch=='y' )  
{  
if(top==-1;  
{  
 printf("\n Stack is Empty! !',);  
 return 0;  
}  
else  
{  
 printf("Deleted element is %d”, stack[top]); //element at top is deleted.

top=top-1;  
}

printf("\n Do you want to delete element?,');  
scanf("%c",&ch); //use a space before %c to clean stdin  
}  
//code fon display  
printf("\n\*\*\* Elements in Stack\*\*\*\*.\n");  
for(i=top; i>=0;i--)  
 printf ( "%d\n", stack[i]);  
}  
**Output**:  
\*\*\*\* Elements in stack\*\*\*\*\*. 50 40 30 20 10   
Do you want to delete element ? y  
Deleted element is 50  
Do you want to delete element ? n  
\*\*\*x Elements in stack\*\*\*\*. 40 30 20 10

**Stack Operation Conditions** :

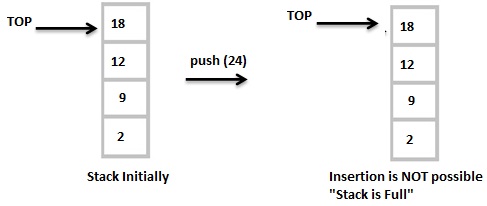
* There are 2 condition to be tested before performing any operation on Stack.
  + **stack Full/ stack overflow** : checked before inserting new data on stack. Means before push().
  + **stack Empty / stack underflow**: checked before deleting data from stack. Means before pop().

**Stack Full/ stack overflow condition** : -

* If the stack is full, it is not possible to push elements into it.
* make sure that the stack is not full. Before push operation compare the value of top with stack size, if it is more than [stack size-1] then overflow occurred.
* To check the status of stack as overflow the code is as follows:

if(top>=MAX-1);  
printf("\n Overflow or Stack is full!!!”);

* In Fig initially the stack size is 4 and position of top is 3 (index) which depict that stack is full.
* Now when user will try to push more elements then top will be compared with [size-1] and get "overflow”  
  message.
* This condition is known as '**Stack full**'**/** **'Stack overflow'** condition.

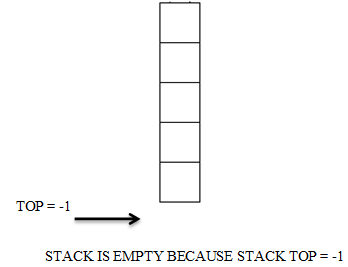


**Stack Empty / stack underflow condition: -**

* It is not possible to DELETE elements from the empty stack.
* Before pop() operation check the position of TOP, if it is -1 then underflow occurred.
* To check the status of stack as underflow the code is as follows:

if(top == -1);  
printf("\n Underflow or Stack is Empty! !");

* In the Fig. the stack has no elements. This condition is known as **'Stack Empty /Stack Underflow'** condition.



**Advantages of Stack** :

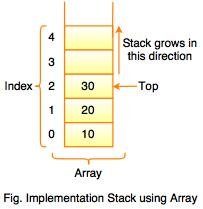
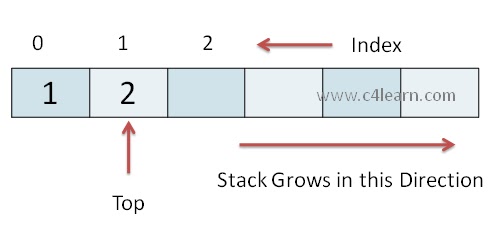
1. Stack provides last-in, first-out access.
2. We can use stack internally.

**Disadvantages of Stack** :

1. Stack has slow access to other items.
2. A stack is a restricted data structure, because only a small number of operations are performed it.
3. We cannot resize on stack.
4. It takes more memory.
5. There is no searching method

**Representation of Stack** : -

**Static representation of stack using an array**: -

* In array implementation of a stack, every new element is inserted is pointed by 'TOP'.
* Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'TOP' by decrementing ‘TOP’ to previous element in array.

**In ‘C’ representation of stack using array : -**

#include <stdio.h>  
#include <stdlib.h>  
#include<process. h>  
#define MAX 5 //Maximun number of elements that can be stored  
int top=-1,stack[MAX];  
void push();  
void pop();  
void display();  
int main()  
{  
int ch;  
printf("\n \*\*\* Stack Menu \*\*\*”);  
printf (" \n1 . Push\n2 . Pop\n3. Display\n4. Exit" );  
while(1) //infinite loop  
 {  
 printf ("\n Enter your choice(1-4) : ");  
 scanf( "%d", &ch );

switch ( ch )  
{  
case 1; push();  
 break;

case 2: pop();

break;  
case 3: display();

break;  
case 4: exit(0);  
default: printf(“\n Wrong Choice!!”);  
}

}  
return 0;  
}

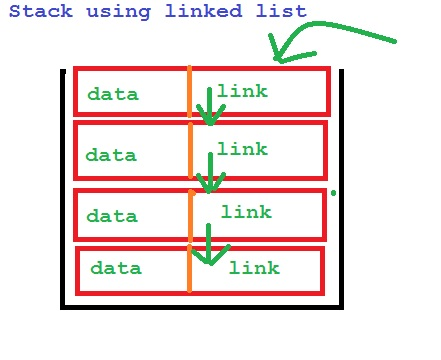
void push( )  
{  
int val;  
if(top== MAX-1);  
 printf( "\n Stack is full ! !'” );  
else  
{  
printf ( "\n Enter element to push :”);

scanf( "%d ", &val ); I  
top= top+1;  
stack[top ] = val;  
}  
}  
void pop( )  
{  
if(toP ==-1);  
 printf( "\n Stack is Empty ! “);  
else  
{  
 printf("\n Deleted element is %d", stack[top]); //delete top element  
 top= top-1;  
}  
}

void display()  
{  
int i;  
if(top==-1);  
 printf("\n Stack is Empty!!”);  
else  
{  
 printf( " \n\*\* Elements in Stack\*\*. \n” );  
 for(i=top; i>=0; i- - )  
 printf( "%d\n", stack[i]);  
}  
**Output:**  
\*\*\* Stack Menu \*x\*  
1.Push  
2.Pop  
3.Display  
4.Exit  
Enter your choice(1-4) :1  
Enter element to push:10  
Enter your choice(1-a):1  
Enter element to push:20  
Enter your choice(1-4) :1  
Enter element to push:30  
Enter your choice(1-4):3  
\*\*\* Elements in stack\*\*\*.  
30  
20  
10  
Enter your choice(1-4) :2  
Deleted element is 30  
Enter your choice(1-4):3  
\*\*\* Elements in stack\*\*\*.  
20  
10

**Dynamic Representation of stack (using Linked List) : -**

* Singly linked list is used to represent a stack.
* Singly linked list only requires that the head node or TOP node.
* The linked-list implementation of a stack requires the following.
  + 1. Structure declaration containing some n fields, and
  + 2. Pointer to the previous node or the next node.



* Linked List implementation uses structure which is collection of heterogeneous data items. A single node of Linked list is represented as follows:

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

**PUSH ( ) on stack using linked list** : -

* PUSH operation works by
  + receiving a data value to push onto the stack, along with a target stack, creating a new node(temp) by allocating memory for it, and
  + Then inserting it into a linked list by checking the status of top.
* If top is NULL , then first element is inserting into stack and its next field will be NULL. If top is not NULL then temp->next point to existing top and now temp will be new top.

int val;  
printf("\n Enten the element to be insert:”);  
scanf("%d", &val);  
struct Node \*temp;  
temp = (struct Node\*)malloc(sizeof(struct Node));  
temp->data = val;  
if(top == NULL)  
 temp->next = NULL;  
else  
 temp->next = top;  
top = temp;

**POP() on stack using linked list** : -

* POP operation checks whether the list is empty before popping from it.
* Then removes the head/top from the linked list, and assigns top pointer to next node.

if(top == NULL)  
 printf("\n Stack is Empty!!!\n");  
else  
{  
 struct Node \*temp = top;  
 printf("\n Deleted element : %d”, temp->data);  
 top = temp->next;  
free(temp);  
}

**traverse operation using linked list** : -

* traverse operation checks whether the list is empty before traversing the list.
* It will traverse all the element while temp->next is not equal to NULL {as we know last element->next field is NULL) using loop.
* Then it will print up to last elements data where temp->next=NULL.

if(top == NULL)  
 printf( "\n Stack is Empty !!\n" );  
else  
{  
 struct Node \*temp = top;  
 while(temp->next ! = NULL)  
 {  
 printf( "%d ",temp->data);  
 temp = temp -> next;  
}  
printf ( "%d”,temp- >data);  
}

|  |  |  |
| --- | --- | --- |
| **Sr No** | **Stack using Array** | **Stack using Linked List** |
| **1** | It is called static implementation of stack. | It is called dynamic implementation of stack. |
| **2** | Pointer is NOT involved. | Pointer is involved. |
| **3** | This method does not provide flexibility because the size of stack is fixed. | This method does provide flexibility as the size of list can grow infinitely. |
| **4** | Memory space is wasted (because of storing less number of elements in the array than the maximum size of array). | Memory space is not wasted, because memory space is allocated only when the user wants to push an element into the stack |
| **5** | Insertion and deletion operation in a stack using an array involve more data movement. | The first advantage of this method is that insertion and deletion operations do not involve more data movement, Only the of pointers. |
| **6** | Stack has slow operation. | Linked list has fast operation. |
| **7** | Less space is used for each data element (no pointer is used). | More space is used for each node |

**Applications of Stack**: -

1. Reversing a string.
2. Polish notations.
3. Conversion of Infix to postfix expression.
4. Evaluation of postfix expression.
5. Conversion of Infix to prefix expression.
6. Evaluation of prefix expression.
7. Recursion.
8. Backtracking (game playing, finding paths, exhaustive searching).

* Stacks are widely used in operating systems, by compiler and by applications. Some of the applications are

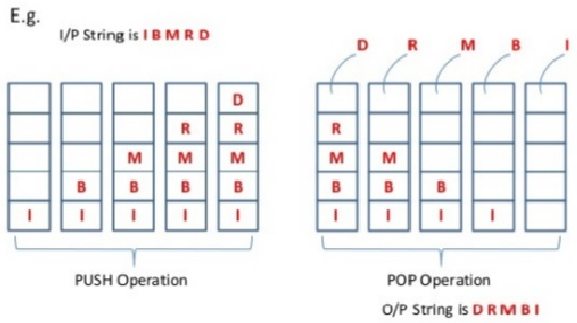
1. Subroutine calls
2. Interrupt handling.
3. Matching parentheses in an expression.

**Reversing a String**: -

* A simple application of stacks is reversing strings.
* This can be achieved very easily by reading the input string character by character and push that onto stack, till end of the string is reached.
* Once, all the characters of the string are pushed onto the stack, they are popped one by one.

**Algorithm for String Reversal**: Let str[] is character array/string to be reversed, stack[] is character to array to store string,

1. Set i=0  
2. While(i < length\_of\_str)  
 PUSH str[i] onto the stack[i]  
 Set i+1  
 End While  
3. Set i=TOP  
4. While(i >= 0)  
 POP the TOP element of the stack[i] and store it in strli]  
 Set i=i-1  
 End While  
5. Print "The reversed string is: ", str  
6. Exit



**Program to reverse a string using stack**

#include< stdio. h>  
#include< conio. h>  
#include< stning. h >  
char stk[10];  
char str[10];  
int top = '1;

int main()  
{  
int i, n;  
char z;  
printf( "Enter the string\n" );  
gets ( str) ;  
n = strlen(str);  
for(i=0; i<n; i++)  
push(str[i] );  
printf("The original string is:%s\n", str);  
printf( "The Reverse string is: " );  
for(i = top; i>=0; i--)  
{  
 z = pop();  
 printf("%c", z);  
}  
return 0;  
}

void push(char ch)  
{  
 top++;  
 stk[top] = 66;  
}

char pop( )  
{  
char temp;  
temp= stk[top];  
top -- ;  
return ( temp );  
}

Output:  
Enter the string : - GPD

The original string is: GPD  
The Reverse string is : DPG

**-------------------------------------ONLY FOR UNDERSTANDING.---------------------------------**

**Polish Notation** : ­-

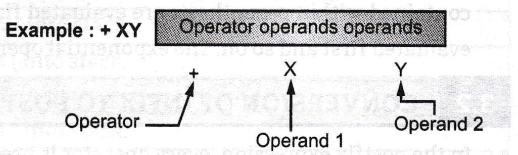
* Polish notation is a symbolic logic invented by Polish mathematician Jan Lukasiewicz in 1920’s.
* A Polish mathematician found a way to represent the same expression in different forms is called polish notation.
* It is the way of writing the operator before OR after the operand is called polish notation.
* Polish notation, also known as Polish prefix notation or simply prefix notation, is a form of notation for logic, arithmetic, and algebra.

**What is an expression……?**

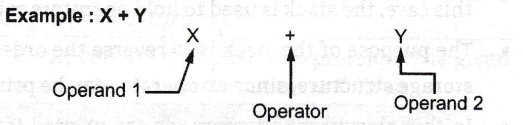
* An expression is defined as, "a number of operands or data items combined using several operators".
* One of the prominent applications of **stacks** is to evaluate the expression.
* **Operators**: Operators indicate the operation to be carried out on operands. Examples of operators are +, -, \*, / , %, etc
* **Operands**: They are the values going to be operated. Examples of operands are A, B, 2, 3 etc.

**Types of notation for an expression.**

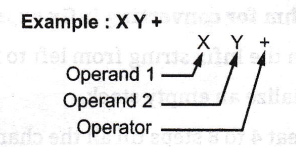
Prefix Notation : - It is a notation in which operator appears before its operands.



Infix Notation: - It is a notation in which operator appears in between the operands.



Post fix Notation : - It is a notation in which operator appears after its operands.



Example: Let us consider the infix expression A + B \* C expression for values A= 4, B = 3, C =7 , respectively.

Ans …..?

A+B\*C  
= 4+3\*7  
= 7\*7  
= 49 Wrong !!!

* For correct answer we have to follow BODMAS rules in order to have right result.
* B = brackets
* O = order ( means Square Or Square roots also called as Exponentiation OR powers )
* D = division.
* M= Multiplication
* A= Addition.
* S= Subtraction.
* Now the correct answer is: 4 + (3\*7) = 4 + 27= 25.

**Operator Precedence**: The order of evaluation is carried cut according to the priority of arithmetic operators.

|  |  |  |
| --- | --- | --- |
| **Sr No** | **Operators** | **Priority** |
| **1** | Exponentiation ( **^** ) OR ( **↑** ) | 3 |
| **2** | Division ( **/** ), Multiplication ( **\*** ) | 2 |
| **3** | Addition ( + ), Subtraction ( **--** ) | 1 |

* The operators of same priority like (\*, /) or (+, -) are performed from left to right.
* Operators contained within parenthesis are evaluated first. when the parentheses are nested, the innermost is  
  evaluated first and so on.
* The exponential operators are evaluated from **Right to Left.**

**----------------------------------------------------------------END. -------------------------------------------------------------------------**

**Conversion of Infix to Postfix Expression**: -

* In the postfix expression, every operator is preceded by two operands on which it operates.
* There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than operands.

**Algorithm for converting infix expression to postfix** : -

1. Scan the Infix string from left to right.
2. Initialize an empty stack.
3. Repeat 4 to 8 steps till all the characters are scanned
4. If the scanned character is operand, add it to the postfix string.
5. If the scanned character is left parenthesis push into stack.
6. If the scanned character is right parenthesis then,

Repeatedly pop from stack and add to Postfix string each operator until corresponding left parenthesis encountered.

7. If the scanned character is Operator  
 If top stack has higher precedence over the scanned character,

pop the stack  
 else

push the scanned character to stack.

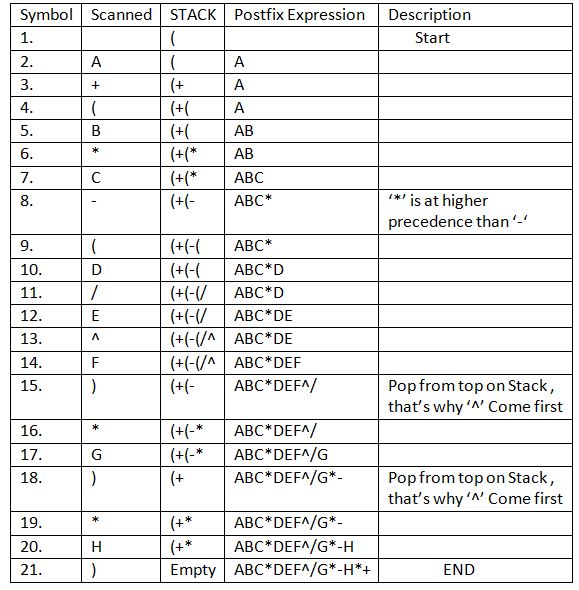
Repeat this step until the stack is not empty and top Stack has precedence over the character.

8. If stack is not empty add top stack to Postfix string and pop the stack. until stack is not empty.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Character** | **Stack** | **Post fix String** | **Operation** |
| ( | ( | Empty | Push ( into stack. |
| A | ( | A | Add A to postfix string. |
| + | (+ | A | Push + into stack. |
| B | (+ | AB | Add B to postfix string. |
| \* | (+\* | AB | Push \* into stack. |
| C | (+\* | ABC | Add C to postfix string. |
| - | (- | ABC\*+ | Character '-' is lower precedence than \* and +, \* and + and add into postfix string. |
| D | (- | ABC\*+D | Push D into stack. |
| ) | Empty | ABC\*+D- | Pop till '(' and add '-' to postfix string. |
| / | / | ABC\*+D- | Push / into stack. |
| ( | /( | ABC\*+D- | Push (into stack. |
| E | /( | ABC\*+D-E | Add E to postfix string. |
| \* | /(\* | ABC\*+D-E | Push \* into stack. |
| F | /(\* | ABC\*+D-EF | Add F to postfix string. |
| ) | Empty | ABC\*+D-EF\*/ | Pop till '(' and add to postfix. |

ABC\*+D-EF\*/ is final postfix expression corresponding to (A+B\*C-D) **/** (E-F).

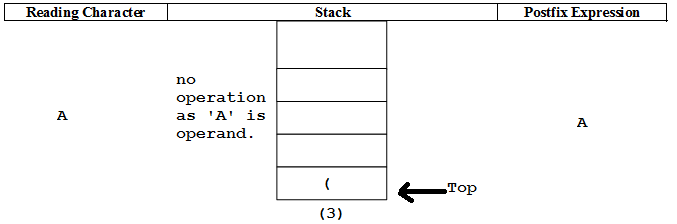
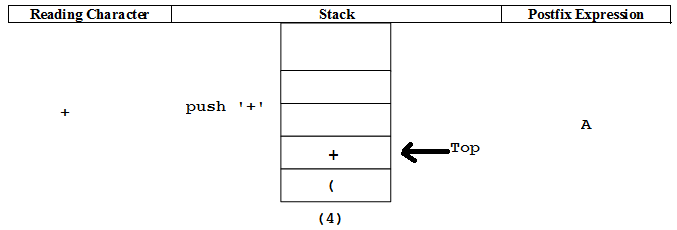
**Example- 2- -** Infix Expression: **A+ (B\*C-(D/E^F)\*G)\*H**, where **^** is an exponential operator.

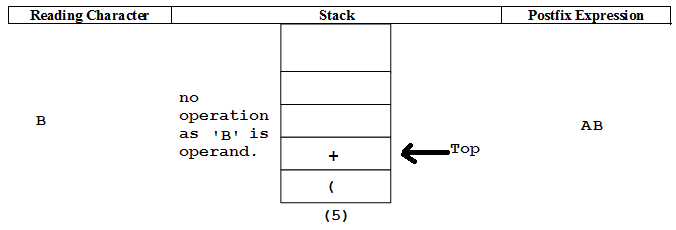
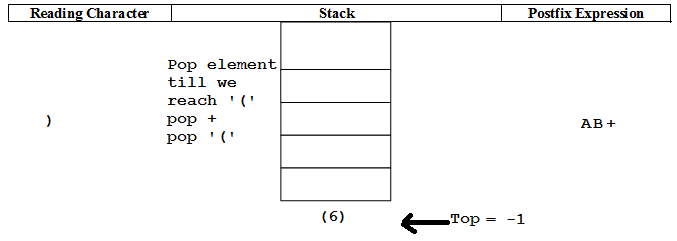


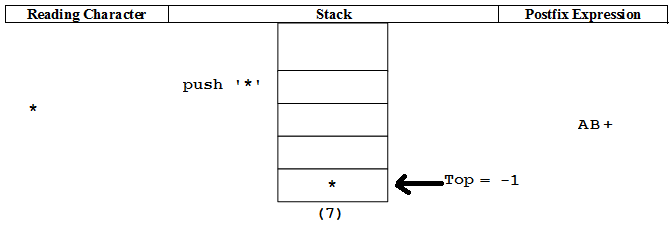
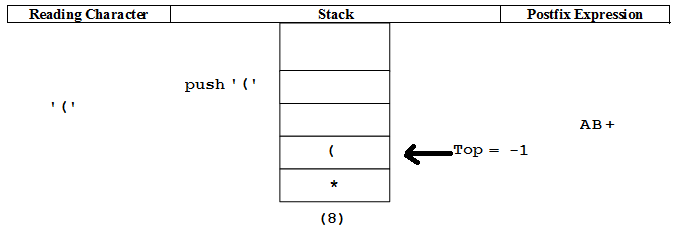
**Example-** Consider the following Infix Expression...**( A + B ) \* ( C - D )**

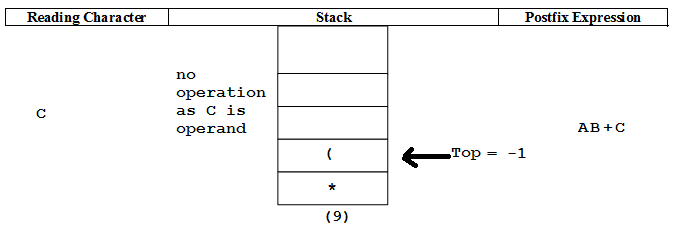
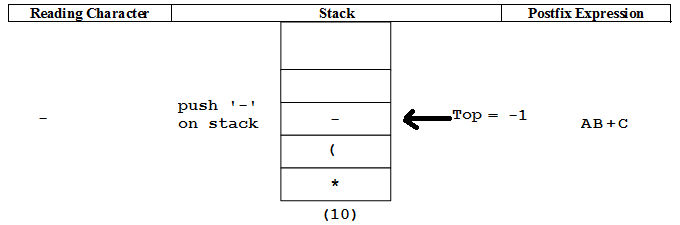
The given infix expression can be converted into postfix expression using Stack data Structure as follows...

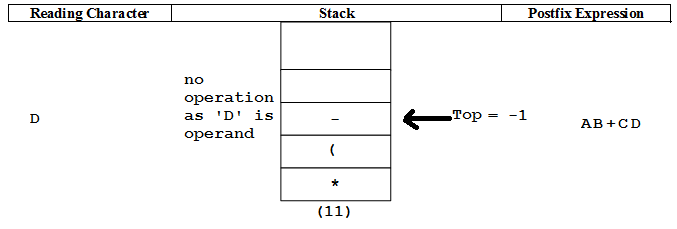
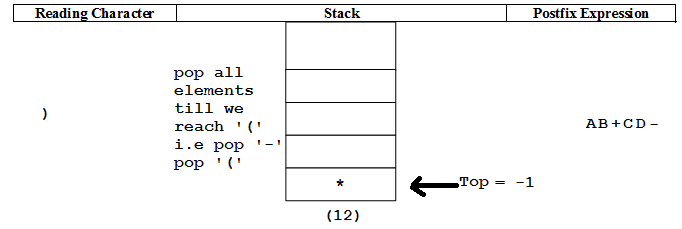


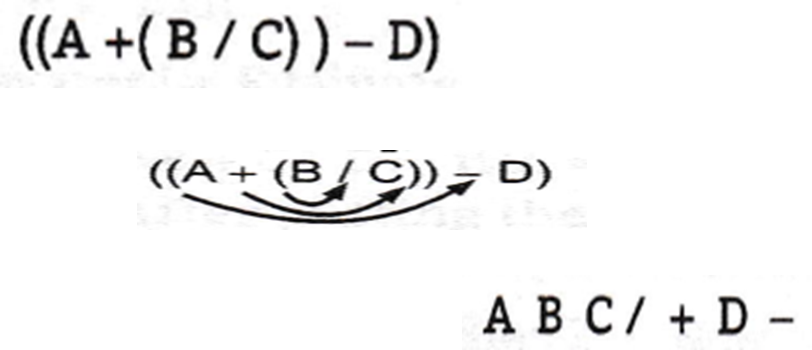
* At the last step pop all the remaining elements from the stack, stack become empty.
* And final postfix expression will be **‘AB+CD-\*’**

**--------------------------------------------------------------ONLY FOR UNDERSTANDING.----------------------------------------------------------**

**Direct Method for Converting Infix to postfix Form : -**

* **Parenthesize the expression starting from left to right.**
* **Move each operator to its corresponding right Parenthesis.**

**E.g. A + B / C – D**

****

**---------------------------------------------------------END.--------------------------------------------------------------------------------**

**Program for conversion of infix to postfix form.**

Program for conversion of infix to postfix form.

#define SIZE 50 /\* Size of Stack \*/

#include <ctype. h>

#include<stdio. h>

char s[SIZE];

int top=-1; /\* Global declarations \*/

void push(char elem)

{ /\* Function for push operation \*/

s [++top]=elem;

}

char pop()

{ /\* Function for pop operation \*/

return(s[top--]);

}

int pr(char elem)

{ /\* Function for precedence x/

switch ( elem)

{

case‘ #‘ : return 0;

case '(‘ : return 1:

case'+'

case '-‘ : return 2:

case'\*‘ :

case '/‘ : return 3:

}

}

int main( )  
{ /\* Main Pnognam \*/  
char infx[50], postfx[50], ch,elem;  
int i=0, k=0;  
printf("\n\Enter Infix Expression: ");  
scanf( "%s ", infx );  
push('#'); /\* # represent end of input expression \*/

while( (ch=infx[i++]) != '\0')

{

if( ch == '(') push(ch);

else

if(isalnum(ch)) postfx[k++]=ch;

else

if( ch == ')')

{

while( s[top] != '(' )

postfx[k++]=pop();

elem=pop(); /\* Remove ( \*/ }

}

else

{ /\* Checking Operator \*/

while( pr(s[top]) >= pr(ch) )

postfx[k++]=pop();

push(ch);

}

}

while( s[top] != '#') /\* Pop from stack till empty \*/

postfx[k++]=pop();

postfx[k]='\0'; /\* Make pofx as valid string \*/

printf("\n Postfix Expression: %s\n",postfx);

return 0;

}

**Output:**

Enter Infix Expression: (A+B\*C-D)/(E\*F)

Postfix Expression: ABC\*+D-EF\*/

**Evaluation of postfix Expressions** : - [ Conversion from Postfix to Infix.]

Algorithm:  
Step 1 : Scan expression from left to right and repeat steps 2 to 3 for each element of an

expression until the end of the expression \0, is encountered.

Step 2 : If an operand is encountered, put it on stack.

Step 3 : If an operator is encountered then

(i) Remove the two top elements of stack, i.e operand-1and operand-2.  
(ii) perform the operation given by the operator on operand-1 and operand-2.  
(iii) Place the result back on stack.

Step 4 : Set value equal to the top element on stack to get final outcome.  
Step5 : Exit.

For example, evaluate the following example:

AB +C- D/ if A =2,8 =3, C =4and D=5. After putting the value expression is :23+4\*5/

Step 1 : First element 2 push onto the stack.(fig (a)).  
Step 2 : Second element 3 push onto the stack. (fig (b)).  
Step 3 : Now operator + is encountered. Pop two elements from stack, perform addition and

push result into stack, (fig (c)).

Step 4 : Push element 4 onto the stack, (fig (d)).  
Step 5 : Now operator \* is encountered. Pop two elements from stack perform multiplication and push result into stack, (fig (e)).

Step 6 : Now 5 push onto the stack, (fig (f)).  
Step 7 : Now operator / is encountered. Pop two elements from stack, perform division and push

result into stack, (fig (g)).

Step 8 : '\0' is encountered in expression and pop the stack and get the final result i.e. 4.

**Example 2**: evaluate the following postfix expression using stacks: abc+d\*f/-, where a=6,b=3,c=6,d=5,f=g. After putting value we will get this postfix expression 636+5\*9/- which can be evaluated by following way.

|  |  |  |
| --- | --- | --- |
| **Symbol Scanned** | **Stack** | **Action** |
| **6** | 6 | PUSH to stack. |
| **3** | 6,3 | PUSH to stack. |
| **6** | 6,3,6 | PUSH to stack. |
| **+** | 6,9 | POP two elements and perform + operation and PUSH intermediate result in stack. |
| **5** | 6,9,5 | PUSH to stack. |
| **\*** | 6,45 | POP two elements and perform \* operation and PUSH intermediate result in stack. |
| **9** | 6,45,9 | PUSH to stack. |
| **/** | 6,5 | POP two elements and perform / operation and PUSH intermediate result in stack. |
| **-** | 1 | POP two elements and perform - operation and PUSH intermediate result in stack. |

**Example 3**: Evaluation of postfix expression consider an expression 552 + \*8 4 / -

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Symbol** | **Operand 2** | **Operand 1** | **Value** | **Stack** |
| 5 |  |  |  | 5 |
| 6 |  |  |  | 5 6 |
| 2 |  |  |  | 5 6 2 |
| + | 6 | **2** | 6 + 2 = 8 | 5 8 |
| \* | 5 | **8** | 5\*8=40 | 40 |
| 8 |  |  |  | 40 8 |
| **4** |  |  |  | 40 8 4 |
| / | 8 | **4** | 8/4 = 2 | 40 2 |
| - | 40 | **2** | 40-2 = 38 | 38 |
| End of Input | - | **-** | Print the result = 38 | Stack Empty |

**Program for evaluation of postfix expression**.

#include<stdio.h>

#include<ctype.h>

int stack[20];

int top = -1;

void push(int x)

{

stack[++top] = x;

}

int pop()

{

return stack[top--];

}

int main()

{

char exp[20];

char \*temp;

int nl,n2,n3,num;

printf("Enter postfix expression without space :: ");

scanf("%s",exp);

temp = exp;

while(\*temp != '\0')

{

if(isdigit(\*temp))

{

num = \*temp - 48; //convert exp. string to integer

push(num);

}

else

{

n1 = pop(); //operandi

n2 = pop(); //operand2

switch(\*temp)

{

case '+': {

n3 = n2 + n1;

break;

}

case '-':

{

n3 = n2 –n1; break;

}

case '\*':

{

n3 = n2 \* nl; break;

}

case '/':

{

n3 = n2 / n1;

break;

}

}

push(n3);

}

temp++;

}

printf("\n The result of postfix expression %s = %d\n\n",exp,pop());

return 0;

}

**Output:**

Enter postfix expression without space ::636+5\*9/-

The result of postfix expression 636+5\*9/- = 1

**Conversion of an Infix Expression to prefix expression** : -

**Algorithm**:  
1. Scan the Infix string from right to left.  
2. Initialize an empty stack.  
3. Repeat 4 and 8 steps till all the characters are scanned  
4. If the scanned character is operand, add it to the postfix string.  
5. if the scanned character is right parenthesis push into stack.  
5. If the scanned character is left parenthesis then,  
 Repeatedly POP from STACK and add to Postfix string each operator until corresponding left parenthesis encountered.

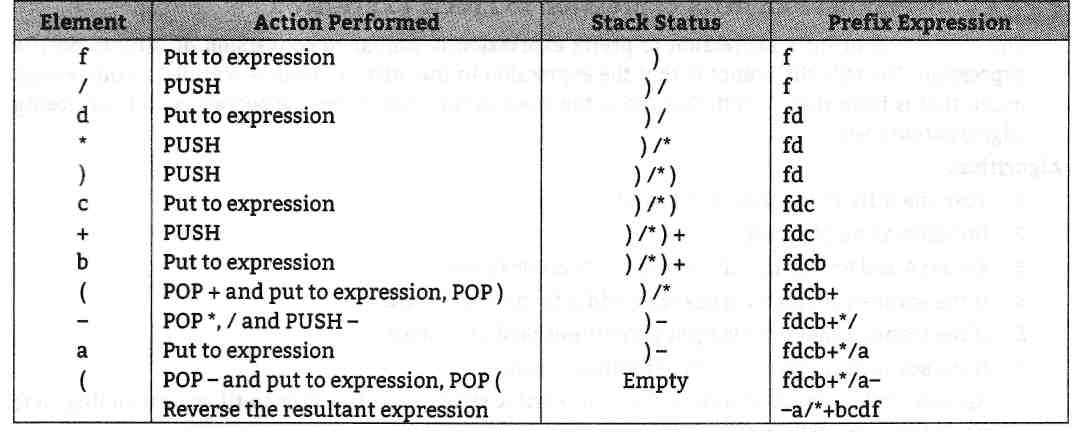
7. If the scanned character is Operator  
 If top Stack has higher precedence over the scanned character,  
 POP the stack  
 else  
 PUSH the scanned character to stack.  
 Repeat this step until the stack is not empty and top Stack has precedence over the character.  
8. If stack is not empty add top Stack to Postfix string and Pop the stack, Repeat this step as long as stack is not empty.  
9. **Read the Postfix string in reverse to get prefix string**.

Example 1: Covert infix expression (A+B\*C-D)/(E\*F) into prefix expression using stack

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Character** | **Stack** | **Postfix String** | **Operation** |  |
| **)** | **)** |  | PUSH) into stack. |  |
| **F** | **)** | F | Add F to postfix string. |  |
| **\*** | **)\*** | F | PUSH \* into stack. |  |
| **E** | )\* | FE | Add E to postfix string. |  |
| **(** | empty | FE\* | Pop till')' and add \* to postfix string. |  |
| **/** | / | FE\* | PUSH \* into stack. |  |
| **)** | /) | FE\* | PUSH) into stack. |  |
| **D** | /) | FE\*D | Add D to postfix string. |  |
| **-** | /)- | FE\*D | PUSH - into stack. |  |
| **C** | /)- | FE\*DC | Add C to postfix string. |  |
| **\*** | /)-\* | FE\*DC | PUSH \* into stack. |  |
| **B** | /)-\* | FE\*DCB | Add B to postfix string. |  |
| **+** | /)-+ | FE\*DCB\* | PUSH + into stack. |  |
| **A** | /)-+ | FE\*DCB\*A | Add A to postfix string. |  |
| **(** | / | FE\*DCB\*A+- | POP till')' and add +,- to postfix string. |  |
| **#** |  | FE\*DCB\*A+-/ | Add top of Stack to Postfix string and Pop the stack. | |

Final prefix string is reverse of postfix string i.e. /-+A\*BCD\*EF

Example 2: Consider the conversion of the (a - (b + c) \* d/f) infix expression to prefix expression:



* The equivalent prefix expression is - a/\*+bcdf.

**--------------------------------------------------------------ONLY FOR UNDERSTANDING.----------------------------------------------------------**

**Direct Method for Converting Infix to prefix Form : -**

1. Parenthesize the expression starting from left to right. Remember the operands associated with operator having higher precedence are first parenthesized.
2. Move each operator to its corresponding left Parenthesis. Remove all the parentheses and shift ah the operators at their appropriate places.

**Example:** Convert the expressions into postfix form: A+B/C-D

* After Parenthesize above expression ((A+(B/C))-D)
* Move each operator to its corresponding left parentheses



* Arrows indicates where to place the operators. Now, remove all the parentheses and shift all the operators at their appropriate places.

**-+A/BCD**

**-----------------------------------------------------------------END.---------------------------------------------------------------------------**

**Program: Program for conversion of infix to prefix expression**

#define SIZE 50 /\* size of Stack \*/

#include<string.h>

#include <ctype.h>

#include<stdio.h>

char s[SIZE];

int top=-l; /\* Global declarations \*/

void push(char elem)

{ /\* Function for PUSH operation \*/

s[++top]=elem;

}

char pop()

{ /\* Function for POP operation \*/

return(s[top--]);

}

int pr(char elem)

{ /\* Function for precedence \*/

switch(elem)

{

case '#': return 0;

case ')': return 1;

case '+':

case '-': return 2;

case '\*':

case '/': return 3;

}

}

int main()

{ /\* Main Program \*/

Setbuf(stdout, NULL); //turn off buffering of stdout

char infx[50],prefx[50],ch,elem;

int i=0,k=0;

printf("\n\n Enter Infix Expression:");

scanf("%s",infx);

push('#');

strrev(infx);

while( (ch=infx[i++]) != '\0')

{

if( ch == ‘)’)

push(ch);

else

if(isalnum(ch))

prefx[k++]=ch;

else

if( ch == '(')

{

while( s[top] != ')')

prefx[k++]=pop();

elem=pop(); /\* Remove ) \*/

}

else

{ /\* Operator \*/

while( pr(s[top]) >= pr(ch) ) prefx[k++]=pop();

push(ch):;

}

}

while( s[top] != '#') /\* Pop from stack till empty \*/

prefx[k++]=pop();

prefx[k]='\0'; /\* Make prfx as valid string \*/ strrev(prefx);

printf("\n\n Prefix Expression: %s\n",prefx);

return 0;

}

**Output:** Enter Infix Expression :( A+B\*C-D)/ (E\*F)

Prefix Expression: /+A-\*BCD\*EF

**Evaluation Of Prefix Expression** : - [Evaluation Of Prefix Expression.]

* To evaluate prefix expression, steps are :

1. First reverse the given infix expression.
2. Then read the expression from left to right.
3. If the element is an operand, then the element is pushed on to the stack.
4. If the element is an operator, then the two operands are popped from the stack and the desired operation is done.
5. The result of the operations is again pushed onto the stack.
6. This process is repeated till the end of the expression is encountered.
7. The final result is popped from the stack.

* **Example:** Consider following prefix expressions with values (-\*+ABCD), let A=4, B=3, C=2, D=5. After putting value in expression we will get (-\*+4325).
* Reverse the expression and follow the above mentioned steps.

|  |  |  |
| --- | --- | --- |
| **Symbol Scanned** | **Stack** | Action |
| **5** | 5 | PUSH into stack. |
| **2** | 5,2 | PUSH into stack. |
| **3** | 5,2,3 | PUSH into stack. |
| **4** | 5,2,3,4 | PUSH into stack. |
| **+** | 5,2,7 | POP two elements and perform + operation and PUSH intermediate result in stack.  PUSH |
| **\*** | 5,14 | POP two elements and perform \* operation and PUSH intermediate result in stack.  \* operation and  PUSH |
| **-** | -9 | POP two elements and perform - operation and PUSH intermediate result in stack.  - operation and  PUSH |

**Program :** Program for evaluation of Prefix expression.

#include<stdio.h>

#include<ctype.h>

#include<string.h>

int stack[20];

int top = -1;

void push(int x)

{

stack[++top] = x;

}

int pop()

{

return stack[top--];

}

int main()

{

char exp[20];revexp[20];

setbuf(stdout, NULL); //turn off buffering of stdout

char \*temp=NULL;

int nl,n2,n3,num;

printf("Enter prefix expression without space :: );

scanf("%s",exp);

strcpy(revexp,strrev(exp)); //convert prefix to postfix

temp=revexp;

while(\*temp != '\0')

{

if(isdigit(\*temp))

{

num = \*temp - 48;//convert exp. string to integer

push(num);

}

else

{

nl = pop();

n2 = pop();

switch(\*temp)

{

case '+':

{

n3 = n2 + nl;

break;

}

case '-':

{

n3 = n2 - nl;

break;

}

case '\*':

**{**

n3 = n2 \* nl;

break;

}

case '/':

{

n3 = n2 / nl;

break;

}

}

push(n3);

}

temp++;

}

printf("\n The result of prefix expression %s = %d\n\n",exp, pop());

return 0;

}

**Output:**

Enter prefix expression without space :: -\*+4325

The result of prefix expression 5234+\*- = -9

**Recursion**: -

* When a function calls itself repeatedly for some input then it is called RECURSION or CIRCULAR DEFINITION.
* When a function calls itself, new set of local variables and parameters are stored on the stack. They are NOT executed immediately, all the recursive calls are pushed on to the stack until the terminating condition is detected, and then the recursion call stored on to the stack are popped and executed one by one.
* Recursive call does NOT make any new copy of the function.
* Recursion is the method where solution to the problem depends on the solution to the smaller instances of the same problem.

* Direct Recursion: - when a function call itself.
* Indirect Recursion: - when function A calls function B, function B calls function C and function C calls the function A again.

**Essential condition of Recursion OR Rules of Recursion**: -

1. Only user defined function can be involved in recursion, NO library function are allowed in recursion.
2. Recursive function must include a condition or a statement to terminate the function.
3. Recursive function MUST have certain base value, so each time function calls itself the arguments of the function must get closer to the base value.

**Advantage of recursion**: -

1. Using recursion the length of the program can be reduced.
2. It is very flexible in data structures like stack, queues, linked list and quick sort.

**Disadvantages**: -

1. It requires extra storage space.
2. If programmer forgot to include the exit/terminating condition then recursive program will executes infinitely.
3. Recursive function is not efficient in execution speed and time.

**Program : Program to calculate factorial of a given number using recursion in C**

#include<stdio.h>

#include<conio.h>

int fact(int);

int main()

{

int num,f;

setbuf(stdout., NULL);

printf("Enter any Number\n");

scanf("%d".,&num);

f=fact(num);

printf("Factorial of a number %d is %d", num, f);

return 0;

}

int fact(int f)

{

if (f==l)

return f;

else

return f \* fact(f-l);

**}**

**Output:**

Enter any Number : 5

Factorial of a number 5 is 120

Program : Program to compute the nth number of Fibonacci series. Fibonacci series is given by 0, 1,1,2,3,5,8…..

#include<stdio.h>

#include<conio.h>

void fibonacci(int);

int main()

{

int n;

setbuf(stdout, NULL);

printf("Enter Range of the fibonacci series:");

scanf("%d",&n);

printf("Fibonacci Series:");

printf("%d\t%d",0,l);

fibonacci(n);

getch();

return 0;

}

void fibonacci(int n)

{

static int first=0, second=l, sum;

if(n>0)

{

sum=first+second;

first=second;

second=sum;

printf("\t%d",sum);

fibonacci(n-l);

}

}

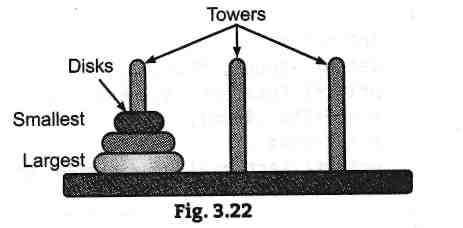
**Output:**

Enter Range of the fibonacci series:8

Fibonacci Series: 0112358 13 21 34

**Tower of Hanoi**: -

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted in Fig

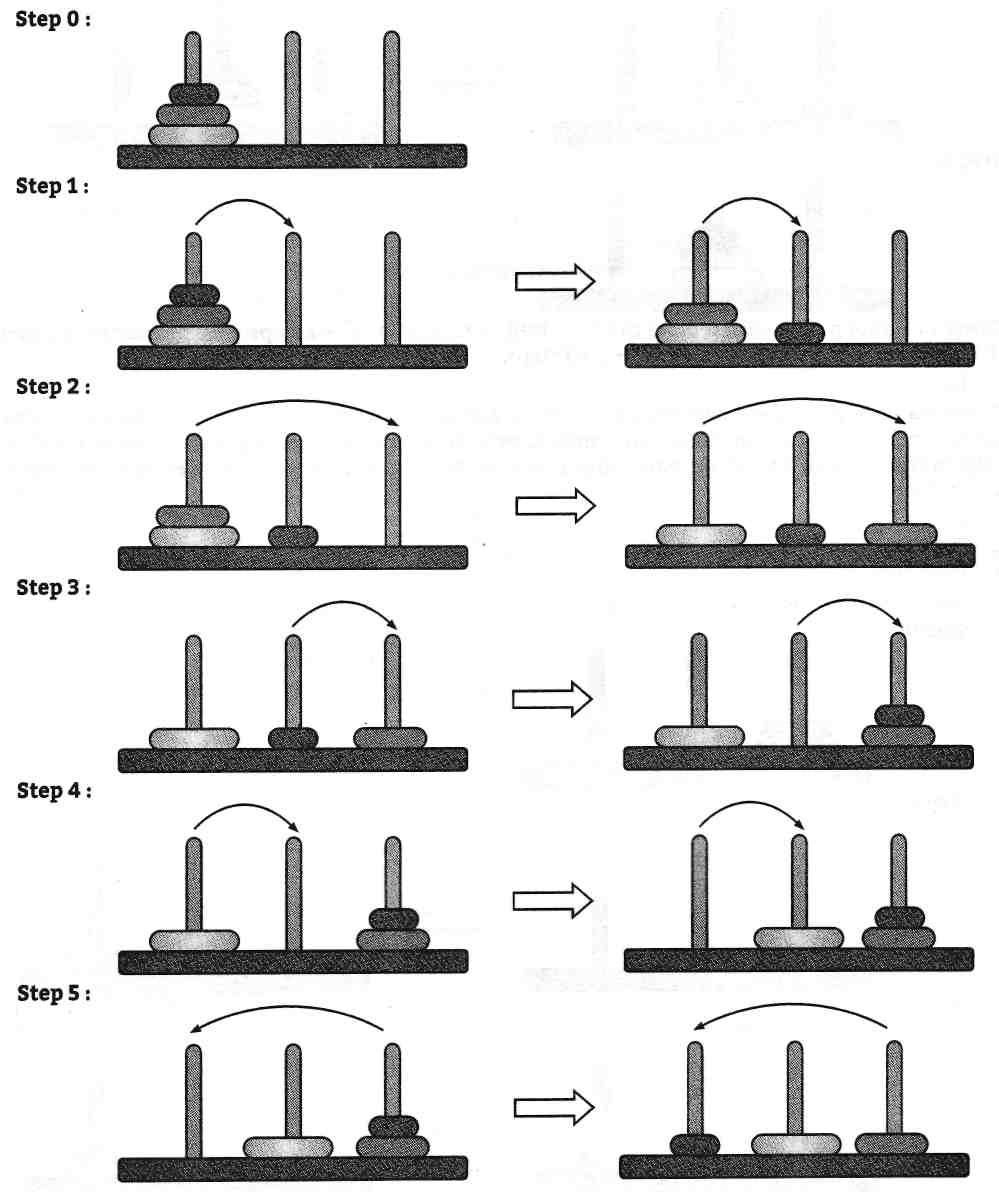


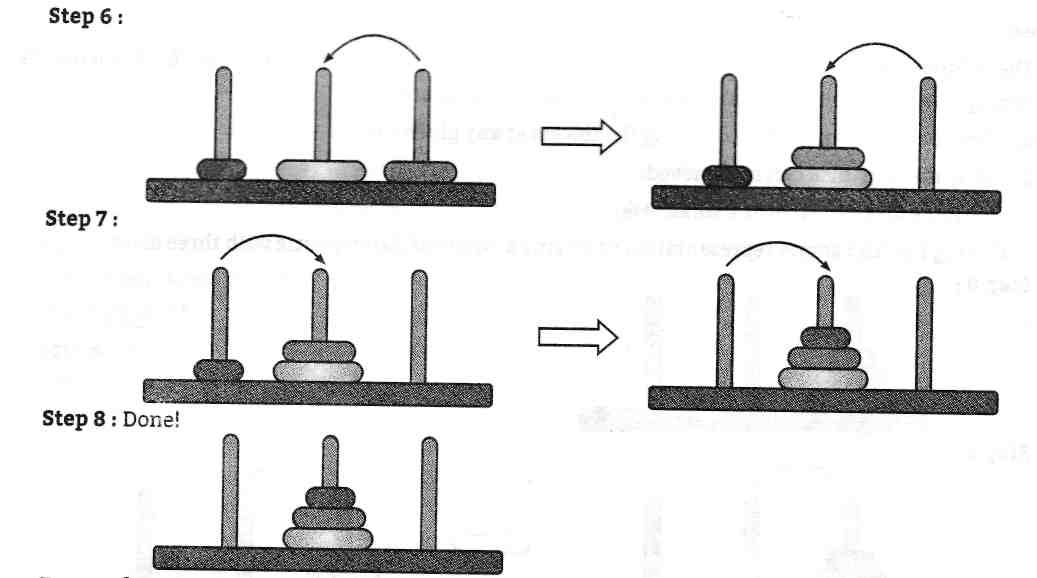
**Rules:**

* The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are:

1. Only one disk can be moved among the towers at any given time.
2. **Only** the "top" disk can be removed.
3. No **large** disk can sit over **a** small disk.

* Following Fig. shows representation of solving a Tower of Hanoi puzzle with three disks.





* Tower of Hanoi puzzle with n disks can be solved in minimum 2n - 1 steps. This presentation shows that a puzzle with 3 disks has taken 23 -1 = 7 steps.

Algorithm : the recursive algorithm for tower of Hanoi problem for n disks

Tower (n,first,mid,last)

Step 1 – if n=1 then

1. Write : first 🡪 last
2. Return.

End if.

Step 2 – [ move n-1 disks from rod first to rod mid]

Call Tower ( n-1,first, mid, last)

Step 3 - Write : first 🡪 last

Step 4 – [ move n-1 disks from rod mid to rod last]

Call Tower ( n-1,first, mid, last)

Step – 5 Return.

**program - Program implementation of Tower of Hanoi in C.**

#include <stdio.h>

#include <conio.h>

void hanoi(int,char,char,char);

int main()

{

int disk;

setbuf(stdout, NULL);

printf("\n ENTER NUMBER OF DISKS: ");

scanf(“%d”,&disk);

printf("\n TOWER OF HANOI FOR %d NUMBER OF DISKS:\n", disk);

hanoi(disk,'A','B', 'C');

return 0;

**}**

void hanoi(int n, char src, char dest, char aux)

{

if(n<=0)

printf("\n ILLEGAL NUMBER OF DISKS"); if(n==l)

printf("\n MOVE DISK FROM %c TO %c",src,aux); if(n>l)

{

hanoi(n-1,src,aux,dest);

printf("\n MOVE DISK FROM %c TO %c",src,aux);

hanoi(n-1,dest,Src,aux);

**}**

**} Output:**

ENTER NUMBER OF DISKS: 3

TOWER OF HANOI FOR 3 NUMBER OF DISKS:

MOVE DISK FROM A TO C

MOVE DISK FROM A TO B

MOVE DISK FROM C TO B

MOVE DISK FROM A TO C

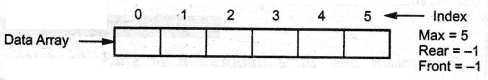
MOVE DISK FROM B TO A

MOVE DISK FROM B TO C

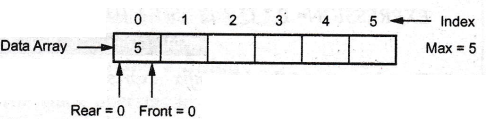
MOVE DISK FROM A TO C

**Queue**: -

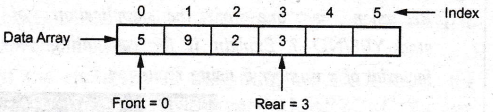
* A queue is a non-primitive, linear and homogenous data structure in which deletions can take place at one end called front end and insertion can take place only at other end called rear end.
* Queues are also called FIFO (First in First Out) lists, since the first element in a queue will the first element out of a queue.
* In other words, the order in which elements enter a queue is the order in which they leave.



* Consider the fig above, During Initialization of a queue, its front and rear are set to -1.
* Below Figure shows the status of queue after insertion of the element '5'.



* Whenever we insert any element in queue, the value of rear is incremented by one Rear = Rear +1
* front remains at the same place,



* Whenever we delete any element in queue, the value of front is incremented by one front = front +1.
* **Note:** During the insertion of first element in the queue, we always increment the front by one i.e. Front = Front +1. Afterwards, the front will not be changed during the entire addition operation.

**For Example:**

* The typical example can be a queue of people who are waiting for a city bus at the bus stop. Any new person is joining at one end of the queue, you can call it as the rear end. When the bus arrives the person at the other end first enters in the bus. You can call it as the front end of the queue.
* 'C representation of the queue

struct queue

{

int qu[size];

int front;

int rear;

}Q;

**Advantages of Queues**:

1. Queues are flexible, requiring no communication programming.
2. Adding or removing elements can be done quickly.
3. Queue is used in many applications such as printing documents
4. Queue provides first-in, first-out access.

**Disadvantages of Queues**:

1. A queue is not readily searchable.
2. Adding or removing elements from the middle of the queue is very complex.
3. Slow access to other items.

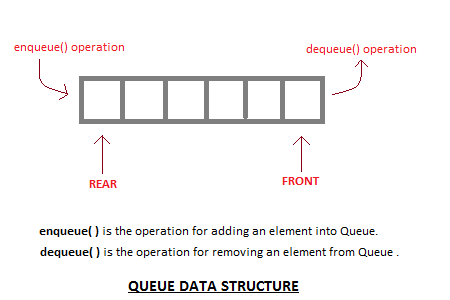
**Applications of Queue**

1. Simulation.
2. CPU scheduling in multiprogramming and time sharing environment.
3. Multilevel queue scheduling
4. Round Robin scheduling
5. **Printing** using the **Computer:** When several documents are to be printed, the printer prints the documents in FIFO method only. For this, i.e. to keep track of documents in FIFO order, a queue is used.
6. **Reservation System:** Reservations such as air ticket or railway reservation systems, use queue to issue reservations on first come first serve basis i.e. FIFO method.
7. **Computer Networks:** Computer networks use queues to give access from server to several clients, who are connected to the server in FIFO method.
8. Operating system maintain queue in **allocating the process to each unit by storing them in buffer.**
9. The interrupts that occurs during the system operation are also maintained in queue.
10. The allocation **of memory for a particular resources (Printer, Scanner, any hand held devices)** are also maintained in queue.
11. All types of **customer service (such as railways ticket reservation centre software's** are designed using queue to store customers information.
12. Round robin technique for processor scheduling is implemented using queues.
13. Queues are useful in **job scheduling algorithm** in the operations system.
14. Queues are also useful for **categorization of data.**

**Queue as an ADT**: -

* The queue is collections of elements in which the element can be inserted by one end called rear and elements can be deleted by other end called front.
* This makes the queue a First-In-First-out (FIFO) data structure.
* If the following operation are implemented on queue then it can be used as an ADT,

1. **Enqueue( )** : - adds a new item to the rear of the queue,
2. **Dequeue ( )**: - removes the front item from the queue.
3. **Queue ( )**: - creates a new queue that is empty.
4. **isEmpty( )** : - tests to see whether the queue is empty.
5. **Size ( )**: - returns the number of items in the queue.
6. **Front ( )**: - returns front element from queue.
7. **Rear ( )**: - returns rear element from queue.



* 'C representation of the queue

struct queue

{

int qu[size];

int front;

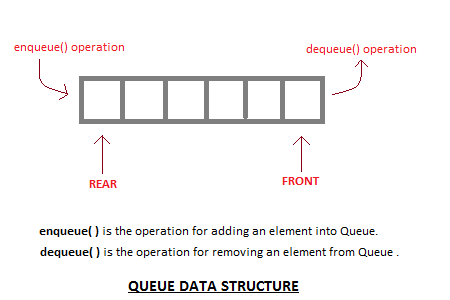
int rear;

}Q;

**Representation / Implementation of queues**: -

1. **Static implementation using Array : -**

* The static implementation is done with an array.
* In this implementation, we should know the exact number of elements to be stored in the queue.
* The size of the array decided before execution at compile time.
* Once the array is declared, its size cannot be altered during the run time of the program.



**Program :** Program to implement a queue using an array.

#include<stdio.h>

#include<conio.h>

#include<process.h>

int main()

{

int queue[5];

int rear=0;

while(rear<=4)

{

printf("Enter Queue element:");

scanf(“%d”, &queue[rear]);

rear++;

}

printf("[\n Elements](file:///nEiements) of Queue");

int front=0;

while(front<=4)

{

printf("%d *",* queue[front]);

front++;

}

return 0;

}

**Output:**

Enter Queue element:10

Enter Queue element:20

Enter Queue element:30

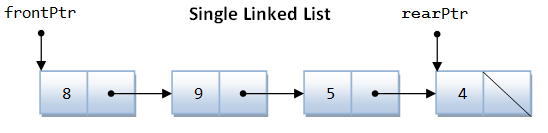
Enter Queue element:40

Enter Queue element:50

Elements of Queues : 10 20 30 40 50

**Dynamic implementation using Linked List**: -

* The dynamic implementation is done with linked list.
* The queue which is implemented using linked list can work for unlimited number of values.
* That means, queue using linked list can work for variable size of data.



Queue is represented using linked list as :

struct Node  
{  
 int data;  
 stnuct Node \*next;  
}\*front, \*rear;

* **enqueue( )** : - operation works by receiving a data value to insert onto the queue, along with a target queue, creating a new node(temp) by allocating memory for it, and then inserting it into a linked list by checking the status of front.
* If front is NULL, then first element is inserting into queue and front and rear will point to that element.
* If front is not NULL then temp will assigned to existing rear->next and now this temp will be new rear.

int value;

printf("Enter the element to be insert: ");

scanf("%d", &value);

struct Node \*temp;

temp = (struct Node\*)malloc(sizeof(struct Node));

temp->data = value:

temp -> next = NULL;

if(front ==NULL)

front = rear = temp; else

{

rear -> next = temp;

rear = temp;

}

• **dequeue ( ): - operation** removes the front element from the linked list, and assigns front pointer to next node. It checks whether the list is empty before deletion.

if(front == NULL)

printf("[\n Queue](file:///nQueue) is Empty!!!\n");

else

{

struct Node \*temp = front;

front = front -> next;

printf("[\n Deleted](file:///nDeleted) element: %d\n", temp->data);

free(temp);

}

**Traverse ( )**: -

* Operation checks whether the list is empty before traversing. It will traverse the entire element while temp->next is not equal to NULL using loop.
* Then it will print last elements data where temp->next=NULL if(front == NULL)

printf("\n Queue is Empty!!!\n");

else

{

struct Node \*temp = front;

while(temp->next != NULL)

{

printf("%d ",temp->data);

temp = temp -> next:

}

printf("%d ",temp->data);

}

------------------------------PROGRAM ONLY FOR UNDERSTANDING---------------------

#include<stdio.h>

#include<conio.h>

struct Node

{

int data;

struct Node \*next;

}\*front = NULL,\*rear = NULL;

void insert(int);

void delete();

void display();

void main()

{

int choice, value;

clrscr();

printf("\n:: Queue Implementation using Linked List ::\n");

while(1){

printf("\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n");

printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d",&choice);

switch(choice){

case 1: printf("Enter the value to be insert: ");

scanf("%d", &value);

insert(value);

break;

case 2: delete(); break;

case 3: display(); break;

case 4: exit(0);

default: printf("\nWrong selection!!! Please try again!!!\n");

}

}

}

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;

}

printf("\nInsertion is Success!!!\n");

}

void delete()

{

if(front == NULL)

printf("\nQueue is Empty!!!\n");

else{

struct Node \*temp = front;

front = front -> next;

printf("\nDeleted element: %d\n", temp->data);

free(temp);

}

}

void display()

{

if(front == NULL)

printf("\n Queue is Empty!!!\n");

else{

struct Node \*temp = front;

while(temp->next != NULL){

printf("%d--->",temp->data);

temp = temp -> next;

}

printf("%d--->NULL\n", temp->data);

}

}

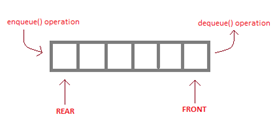
--------------------------------------------------------------------------------

**Types of Queues**: -

**Linear Queue**: -

* A queue is an ordered list in which all insertions take place at one end called the rear and deletion takes place at front.
* Following are basic features of linear queue:

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO structure.



* From above example we can conclude following things:
* When front=-1and rear=-1then queue is empty.
* When front =rear then there exist a single element in queue.
* when front=O and rear=size-1then queue is full.
* The queue data structure supports two operations:

1. Enqueue/insertion or addition of element in queue, and
2. Dequeue/deletion or removal of element from a queue.

**Insertion of an Element in a Queue**

* insertion takes place at rear side and deletion takes place at front side of queue.
* so to insert check the condition for overflow.
* first element will be stored at index 0 in queue and last element will be stored at MAX-1 so if rear = MAX-1 then print queue is full or overflow.
* Else increase the value of rear by one and insert element at rear.
* as initial value of front is -1 so you have to set it to 0 as first element is inserted.

**Algorithm:**

Let queue [Max] is an array for implementation of queue. Let front and queue initialize to -1.

1 if rear=MAX-l then //[check queue is overflow]

2. print queue is overflow

3. else

1. if front = -1 then set front to 0
2. set rear = rear + l

6. Set Queue[Rear] = Item //Insert element at rear position of queue.

**Deletion from a Queue**

* To delete an element from queue using array first you have to check the condition for underflow.
* As first element will be stored at index 0 in queue and last element will be stored at MAX-1.
* So if front= -1 then print queue is empty or underflow occurred otherwise remove front element from queue and increase the value of front by one.
* When there is only one element in queue, the value of front and rear will be 0.
* After last delete operation, the value of front will be one more than rear i.e. front >rear so need to set front and rear to -1 to represent empty queue.

**Dequeue Algorithm:**

Let queue [Max] is an array for implementation of queue. Let front and queue initialize to -1.

1. If front = -1 then //[check queue is underflow]

2. Print queue is underflow

3. Else

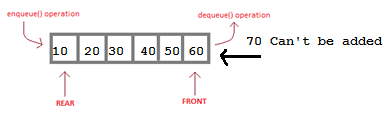
1. Set value = Queue [front] //Delete the element which is at the front of queue
2. Set front = front +1
3. If front > rear then set both front and rear to -1

**Queue Operation Condition**: -

1. Queue Full condition
2. Queue Empty condition
3. **Queue Full condition** : - we must check whether the queue is full or not. If the rear Pointer is going beyond the maximum size of the queue then the queue overflow occurs.

It is checked using following code :

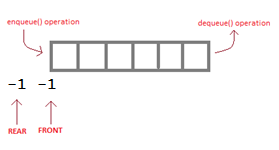
if(near == MAX-1)  
printf( "\n Queue is Fu11 ! ! " );



1. **Queue Empty condition : - we** must check whether the Queue is empty or not. If the Queue is empty, you can not perform the deletion. Empty queue is called the queue underflow condition.

It is checked using following code

if(front == -1)  
printf( "\n Queue is Empty ! ! " );



#include<stdio.h>

#include<conio.h>

#define SIZE 10

void enQueue(int);

void deQueue();

void display();

int queue[SIZE], front = -1, rear = -1;

void main()

{

int value, choice;

clrscr();

while(1){

printf("\n\n\*\*\*\*\* MENU \*\*\*\*\*\n");

printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");

printf("\nEnter your choice: ");

scanf("%d",&choice);

switch(choice){

case 1: printf("Enter the value to be insert: ");

scanf("%d",&value);

enQueue(value);

break;

case 2: deQueue();

break;

case 3: display();

break;

case 4: exit(0);

default: printf("\nWrong selection!!! Try again!!!");

}

}

}

void enQueue(int value){

if(rear == SIZE-1)

printf("\nQueue is Full!!! Insertion is not possible!!!");

else{

if(front == -1)

front = 0;

rear++;

queue[rear] = value;

printf("\nInsertion success!!!");

}

}

void deQueue(){

if(front == rear)

printf("\nQueue is Empty!!! Deletion is not possible!!!");

else{

printf("\nDeleted : %d", queue[front]);

front++;

if(front == rear)

front = rear = -1;

}

}

void display(){

if(rear == -1)

printf("\nQueue is Empty!!!");

else{

int i;

printf("\nQueue elements are:\n");

for(i=front; i<=rear; i++)

printf("%d\t",queue[i]);

}

}



**CIRCULAR QEUEUE**:-

* Circular Queues are the queues implemented in circular form rather than in a straight line.
* Circular Queues overcome the problem of unutilized space in linear queue implemented as an array.
* A Circular Queue is one in which the insertion of a new element is done at very first location of Queue if last location.
* The main advantage of circular queue is we can utilize the space of the queue fully.
* There is a formula which has to applied for setting the front and rear pointers for circular queue.

**Rear = (Rear+1) % maxsize**

**Front = (Front+1) % maxsize**

**Need of Circular Queue**

* In a circular queue, if rear=max-l and beginning position of queue is empty, then new elements can be added at beginning.
* **Circular Increment : -** Rear can be incremented circularly by the following code.

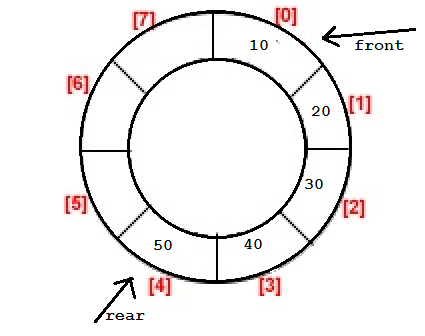
if((rear=MAX-l) and (front != 0))

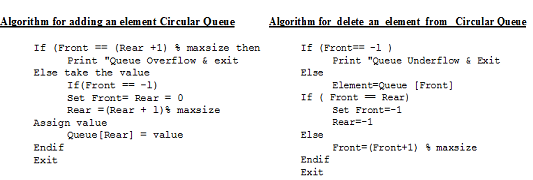
rear=0;

else

rear=(rear+l) %MAX

The same applies for front.





-----------------------------------------------------------------------------------------

**Circular Queue Representation Using Array** : -

#include<stdio.h>

#include<conio.h>

#define SIZE 5

void enQueue(int);

void deQueue();

void display();

int cQueue[SIZE], front = -1, rear = -1;

void main()

{

int choice, value;

clrscr();

while(1){

printf("\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n");

printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d",&choice);

switch(choice){

case 1: printf("\nEnter the value to be insert: ");

scanf("%d",&value);

enQueue(value);

break;

case 2: deQueue();

break;

case 3: display();

break;

case 4: exit(0);

default: printf("\nPlease select the correct choice!!!\n");

}

}

}

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;

}

}

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;

}

}

void display()

{

if(front == -1)

printf("\nCircular Queue is Empty!!!\n");

else{

int i = front;

printf("\nCircular Queue Elements are : \n");

if(front <= rear){

while(i <= rear)

printf("%d\t",cQueue[i++]);

}

else{

while(i <= SIZE - 1)

printf("%d\t", cQueue[i++]);

i = 0;

while(i <= rear)

printf("%d\t",cQueue[i++]);

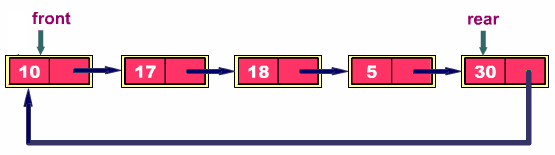
}

}

}

**Circular Queue Using Linked List:**

* In a linked list circular queue, the rear node always has the reference of the front node.
* Even if the front node is removed the rear node has the reference of the new front node. Fig. below shows circular queue with linked list.



---------------------------------------------------------------------------------

**Following program implements circular queue using linked list**.

#include <stdio.h>

#include <process.h>

#include<malloc.h>

struct node

{

int info;

struct node \*ptr;

}\*front, \*rear, \*temp, \*front;

int main()

{

int data;

char ch;

front = rear = NULL;

do

{

printf("[\n Enter](file:///nEnter) element to Insert:\n");

scanf("%d",&data);

temp = (struct node \*)malloc(sizeof(struct node));

temp->ptr = NULL;

temp->info = data;

if (rear == NULL)

{

Rear = temp;

front = rear;

}

else

{

Rear -> ptr = temp;

rear = temp;

}

printf("\n Do you want to insert more elements?");

scanf(" %c",&ch); //use a space before %c to clear stdin }while(ch==,y,||ch==,Y,);

printf("\n\*\*\*\* Elements in Circular Queue\*\*\*\*\*\n");

if ((front == NULL) && (rear == NULL))

{

printf("Queue is empty");

return 0;

}

while (front != rear)

{

printf("%d ", front->info);

front = front -> ptr;

}

printf("%d", front->info);

return 0;

}

**Output**

Enter element to Insert:

10

Do you want to insert more elements?y

Enter element to Insert:

20

Do you want to insert more elements?n

\*\*\*\* Elements in Circular Queue\*\*\*\*\*

10 20

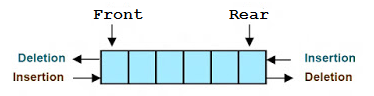
---------------------------------------------------------------------------------------------------------------------------------------

**Applications of Circular Queue**:

1. **Adding Large Integers**: Large integers can be added very easily using circular queues. Here the rightmost digit is placed in the front node and leftmost digit is placed in the rear node.
2. **Memory Management**: The unused memory Iocations in the case of ordinary queues can be utilized in circular queues.
3. **Computer Controlled Traffic System**: In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.

# Double Ended Queue (Dequeue) : -

* In the doubly ended queue we can make use of both the ends for insertion of the elements as well as we can use both the ends for deletion of elements.
* That means it is possible to insert the elements by rear as well as by front. Similarly it is possible to delete the elements from front as well as from rear.
* It is a homogeneous list of elements in which insertion and deletion operations performed from both the ends.



**Types of Dequeue**: -

1. **Input Restricted Dequeue-** It allow insertions at only one end of an array but deletions allow at both the ends.
2. **Output Restricted Dequeue-** It allow deletion at only one end of an array but insertions allow at both the ends.

* Since both insertion and deletion are performed from either end, it is necessary to design algorithm to perform following 4 operations :

1. Insertion of an element at the REAR end of the Queue
2. Deletion of an element from the FRONT end of the Queue
3. Insertion of an element at the FRONT end of the Queue
4. Deletion of an element from the REAR end of the Queue.

**a) Algorithm to insert an element at the REAR end of the Queue-**

Step 1 : if (Rear == (max-1)) then

Print "Queue is Full"

Step 2 : Set Rear = Rear + l

Step 3 : Queue[Rear] = Element

Step 4 : Exit

**b) Algorithm to delete an element from the FRONT end of the Queue**

Step 1 : if (Front = Rear) then Print "Queue is empty"

Return

Step 2 : Set Front = Front +1

Step 3 : Element = Queue [Front]

Step 4 : Exit

**c) Algorithm to insert an element at the FRONT end of Queue**

Step 1 : if front=0 then

Print "Queue is full"

Return

Step 2 : Front=Front-l

Step 3 : Queue[Front] = Element

Step 4 : Exit

**d) Algorithm to delete an element from the REAR end of Queue**

Step 1 : if( front=rear) then

Print "Queue is empty"

Return

Return Step 2 :

Rear=Rear +1

Step 3 : element=Q[Rear]

Step 4 : Exit

--------------------------------------------------------------------------------------------------------------------------------------------------

**Program to implement Double Ended Queue (Dequeue)**

#include<stdio.h>

#include<conio.h>

#define SIZE 100

void enQueue(int);

int deQueueFront();

int deQueueRear();

void enQueueRear(int);

void enQueueFront(int);

void display();

int queue[SIZE];

int rear = 0, front = 0;

int main()

{

char ch;

int choice1, choice2, value;

printf("\n\*\*\*\*\*\*\* Type of Double Ended Queue \*\*\*\*\*\*\*\n");

do

{

printf("\n1.Input-restricted deque \n");

printf("2.output-restricted deque \n");

printf("\nEnter your choice of Queue Type : ");

scanf("%d",&choice1);

switch(choice1)

{

case 1:

printf("\nSelect the Operation\n");

printf("1.Insert\n2.Delete from Rear\n3.Delete from Front\n4. Display");

do

{

printf("\nEnter your choice for the operation in c deque: ");

scanf("%d",&choice2);

switch(choice2)

{

case 1: enQueueRear(value);

display();

break;

case 2: value = deQueueRear();

printf("\nThe value deleted is %d",value);

display();

break;

case 3: value=deQueueFront();

printf("\nThe value deleted is %d",value);

display();

break;

case 4: display();

break;

default:printf("Wrong choice");

}

printf("\nDo you want to perform another operation (Y/N): ");

ch=getch();

}while(ch=='y'||ch=='Y');

getch();

break;

case 2 :

printf("\n---- Select the Operation ----\n");

printf("1. Insert at Rear\n2. Insert at Front\n3. Delete\n4. Display");

do

{

printf("\nEnter your choice for the operation: ");

scanf("%d",&choice2);

switch(choice2)

{

case 1: enQueueRear(value);

display();

break;

case 2: enQueueFront(value);

display();

break;

case 3: value = deQueueFront();

printf("\nThe value deleted is %d",value);

display();

break;

case 4: display();

break;

default:printf("Wrong choice");

}

printf("\nDo you want to perform another operation (Y/N): ");

ch=getch();

} while(ch=='y'||ch=='Y');

getch();

break ;

}

printf("\nDo you want to continue(y/n):");

ch=getch();

}while(ch=='y'||ch=='Y');

}

void enQueueRear(int value)

{

char ch;

if(front == SIZE/2)

{

printf("\nQueue is full!!! Insertion is not possible!!! ");

return;

}

do

{

printf("\nEnter the value to be inserted:");

scanf("%d",&value);

queue[front] = value;

front++;

printf("Do you want to continue insertion Y/N");

ch=getch();

}while(ch=='y');

}

void enQueueFront(int value)

{

char ch;

if(front==SIZE/2)

{

printf("\nQueue is full!!! Insertion is not possible!!!");

return;

}

do

{

printf("\nEnter the value to be inserted:");

scanf("%d",&value);

rear--;

queue[rear] = value;

printf("Do you want to continue insertion Y/N");

ch = getch();

}

while(ch == 'y');

}

int deQueueRear()

{

int deleted;

if(front == rear)

{

printf("\nQueue is Empty!!! Deletion is not possible!!!");

return 0;

}

front--;

deleted = queue[front+1];

return deleted;

}

int deQueueFront()

{

int deleted;

if(front == rear)

{

printf("\nQueue is Empty!!! Deletion is not possible!!!");

return 0;

}

rear++;

deleted = queue[rear-1];

return deleted;

}

void display()

{

int i;

if(front == rear)

printf("\nQueue is Empty!!! Deletion is not possible!!!")

else{

printf("\nThe Queue elements are:");

for(i=rear; i < front; i++)

{

printf("%d\t ",queue[i]);

}

}

}

--------------------------------------------------------------------------------------------------------------------------------------------------

**Priority Queue** : -

* In a priority queue, each item has a priority and when it’s time to select an item, the item with highest priority is the one that is chosen.

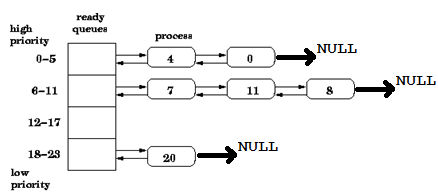
**Example** :

* Suppose you are sitting in the waiting room of a doctor's dispensary. The doctor will examine the patients in the order in which they have come. However, if a critical patient comes, the doctor will have to examine that patient first. This means that FIFO ordering should be violated in this case.

**Definition**: -

* The priority queue is a data structure having a collection of elements which are associated with specific ordering i.e the ordering among the elements decides the manner in which Add and Delete Operations will be performed.
* A priority queue is a collection of elements where element has assigned a priority value such that the order in which elements are deleted and processed comes from the following rules:

1. **An element of higher priority is processed before any element of lower priority.**
2. **Two elements with the same priority are processed according to the order in which they were added to the queue**.

****

**Types of Priority Queue : -**

1. Ascending Priority Queue
2. Descending Priority Queue
3. **Ascending Priority Queue-** It is a collection of elements in which the elements can be inserted arbitrarily but only the smallest element can be **removed.**
4. **Descending Priority Queue-** It is a collection of elements in which insertion of elements can be in any order but only largest element can be removed.

* In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed deleting each time.

**Application of Priority Queue**

1. **Job scheduling**: In the application of job scheduling, jobs with priority are entered into the queue and the job with highest priority is chosen for execution.
2. **In network communication**, to manage limited bandwidth for transmission the priority queue is used.
3. **In simulation modeling**, to manage the discrete events the priority queue is used.
4. **To** find **k\* largest element:** To find kth biggest element in a list of elements a priority queue can be maintained to indicate their corresponding degree. And the required element can be directly picked up.
5. **Sorting**: Some sorting techniques use priority queues for implementing sorting.

**Operations on Priority Queue**: -

* In C language, we define priority queue using following structures:

struct node

{

int info;

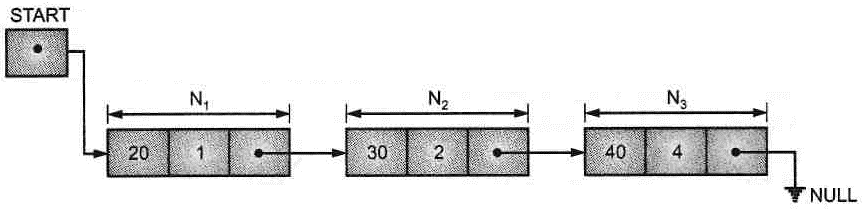
int priority;

struct node\*link;

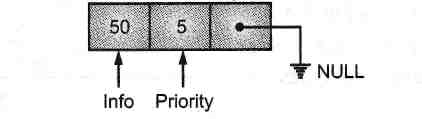
}

**Insert Operation:** Elements can be inserted in any order, but are arranged in order of their priority value in the queue.

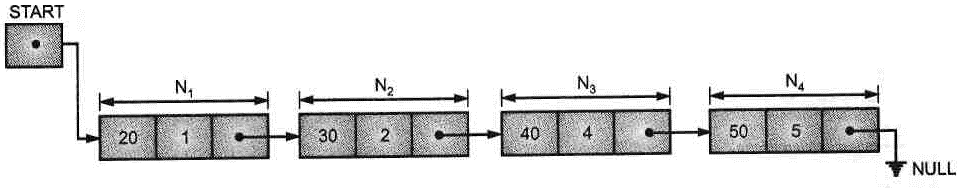
Example: fig shows the priority queue.



* Suppose one wants to add new node N4 having priority value 5 which is shown in Fig.

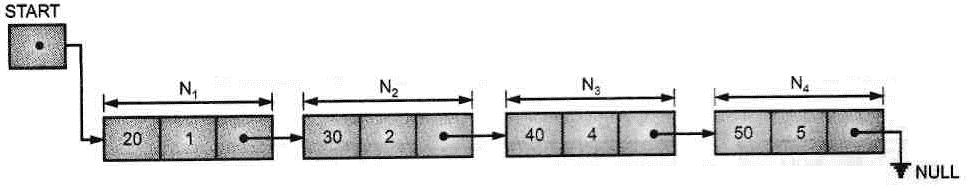


* Since, new node has priority 5 then we insert this node at the end of existing queue as its priority is less compare to existing elements.

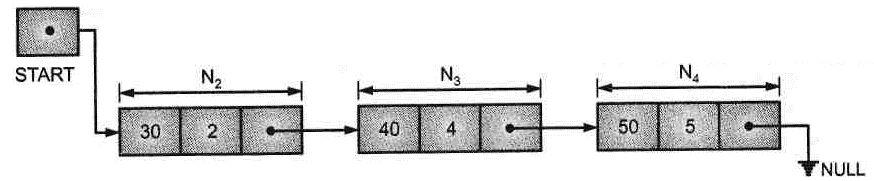


**Delete operation:**

* The elements are deleted from the queue in the order of their priority. A highest priority element removed first. The elements with the same priority are given equal importance and processed accordingly. For example, suppose we have a list of priority queue as shown in Fig.



* Fig. below shows priority queue after deletion node N**1**.

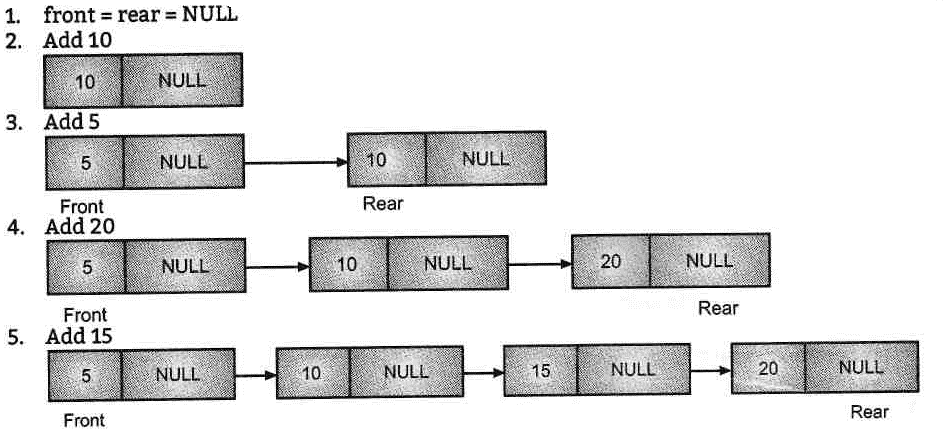


**Representation / Implementation of Priority Queue** : -

A priority queue can be implemented dynamically in 2 ways:

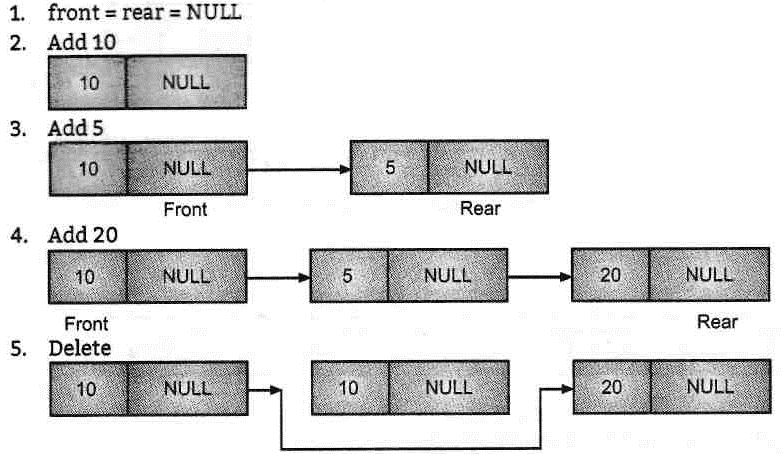
**Ordered List Implementation:**

* In this method the elements are added to the list in such a way that the list is in the sorted order.
* When an element is to be added to the priority queue, it is inserted in its correct position in the queue.
* For deletion the first element (which is smallest element) is removed.
* **Example:**



**Unordered List Implementation:**

* In this method of implementation, elements are added to the rear of the queue.
* This is simple insertion process. The insertion is not dependent on sorted order.
* For deletion the entire list has to be examined in order to find the smallest element, which is then removed from the list.
* **Example:**



* At the time of deletion, the smallest element is deleted from the list.

**Program : Program for Insertion in Priority Queue.**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data, priority;

struct node \*next;

}\*start; //start is declared

int main()

{

int data, pr;

char ch;

do

{

printf("[\n Enter](file:///nEnter) element and its priority: ");

scanf("%d%d",&data,&pr); //input from user

struct node \*temp, \*t;

temp = (struct node \*)malloc(sizeof(struct node));

temp->data=data;

temp->priority=pr;

temp->next=NULL;

if(start==NULL)

start = temp;

else if(start->priority>pr)

{

temp->next=start;

start=temp;

}

else

{

t=start;

while( t->next != NULL && (t->next)->priority <= pr )

t = t->next;

temp->next = t-> next;

t->next = temp;

}

printf("\n Do you want to insert more elements?");

scanf(" %c",&ch); //use a space before %c to clean stdin }while(ch=='y'||ch=='Y');

printf("\n\*\*\*\* Elements in Circular Queue\*\*\*\*\*\n");

struct node \*templ = start;

while(temp1!= NULL)

{

printf("\n Data = %d Priority = %d ",templ->data, temp1->priority); temp1=temp1->next;

}

return 0;

} //end of Main

**Output:**

Enter element and its priority: 10 4

Do you want to insert more elements? y

Enter element and its priority: 20 3

Do you want to insert more elements? y

Enter element and its priority: 30 1

Do you want to insert more elements? n

\*\*\*\* Elements in Circular Queue\*\*\*\*\*

Data=30 Priority=l

Data=20 Priority=3

Data=10 Priority=4

---------------------------------------------------------------------------------------------------------------------------------------------------

**Program: Program for deletion in priority queue.**

#include<stdio.h>

#include<stdlib.h>

void insert();

void delete ();

void display();

struct node

{

int data, priority;

struct node \*next;

}\*start; //start is declared

int main()

{

setbuf(stdout, NULL);

int ch;

printf("\n\*\*\* Priority Queue Menu \*\*\*");

printf("\nl. Insertion \n2.Deletion \n3.Display [\n4.Exit](file:///n4.Exit)");

while(l)

{

printf("[\n Enter](file:///nEnter) the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):"); scanf("%d",&ch);

switch(ch)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

display();

break;

case 4:

exit(0);

default:

printf("[\n Wrong](file:///nWrong) Choice!!");

}

}

return 0;

}

void insert ()

{

int data, pr;

printf("[\n Enter](file:///nEnter) element and its priority: ");

scanf("%d%d",&data,&pr); //input from user

struct node \*temp, \*t;

temp = (struct node \*)malloc(sizeof(struct node));

temp->data = data;

temp->priority = pr;

temp->next = NULL;

if(start == NULL)

start = temp;

else if(start->priority > pr)

{

temp->next = start;

start=temp;

}

else

{

t=start;

while(t->next != NULL && (t->next)->priority <= pr )

t=t->next;

temp->next = t->next;

t->next = temp;

}

}

void delete ()

{

if(start != NULL)

{

printf("\n Removed Element: %d", start->data);

start = start->next;

}

else

printf("\nQueue is Empty");

}

void display()

{

printf("\n\*\*\*\* Elements in Circular Queue\*\*\*\*\*\n");

struct node \*templ = start;

while(templ!=NULL)

{

printf("\n Data=%d Priority=%d ",temp1->data, temp1->priority);

temp1 = temp1->next;

}

}

**Output:**

\*\*\* Priority Queue Menu \*\*\*

1.Insertion

2.Deletion

3.Display

4.Exit

Enter the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):l

Enter element and its priority: 10 3

Enter the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):l

Enter element and its priority: 20 1

-

Enter the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):l

Enter element and its priority: 30 2

Enter the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):3

\*\*\*\* Elements in Circular Queue\*\*\*\*\*

Data=20 Priority=l

Data=30 Priority=2

Data=10 Priority=3

Enter the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):2

Removed Element: 20

Enter the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):3

\*\*\*\* Elements in Circular Queue\*\*\*\*\*

Data=30 Priority=2

Data=10 Priority=3

Enter the Choice(l.Insertion 2.Deletion 3.Display 4.Exit):