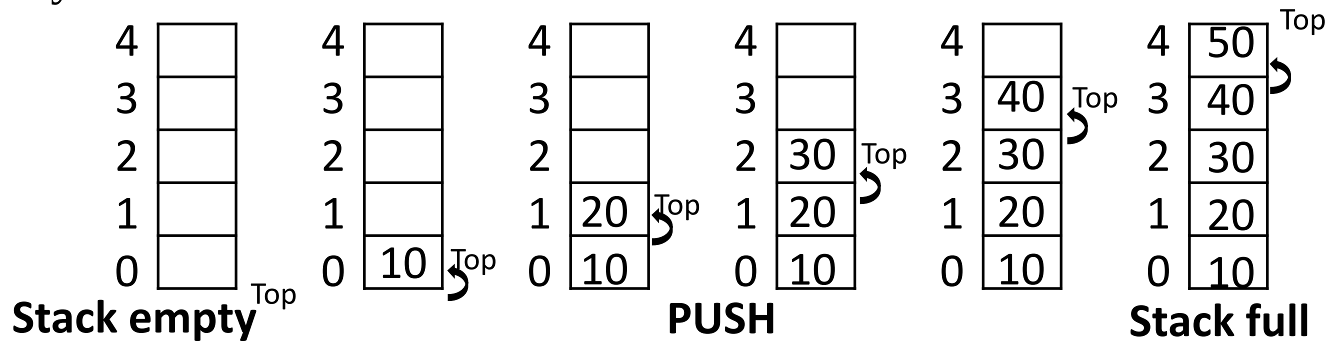
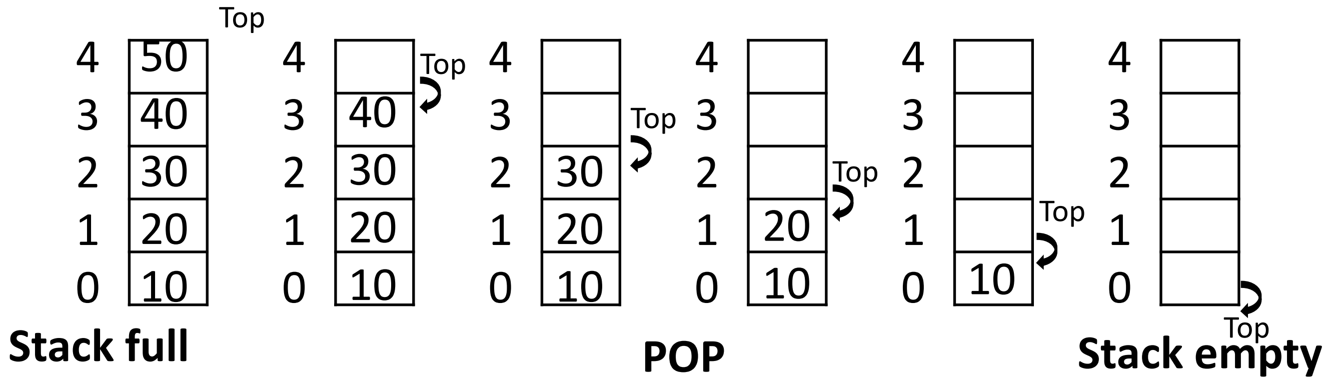
Definition & Diagram of Stack

A stack is an ordered collection of items into which new item can be inserted one end, and can be removed from same end called as a top of the stack.

This implies that stack is different than array as it is ordered collection of elements having property last in pass Out. Hence stack also uses different terminologies for operations, which are referred as Push and Pop instead of Store and Retrieve in array.





Stack as an ADT & Operations on Stack

Stack S is a finite collection of elements having same data type and all operations are carried out at one end called a top

Operations on Stack

1. **Push (item x, Stack S) -> Stack S';**

   If stack is not full then,

   This function inserts an item x into top of the stack and returns new stack S' with Top pointing to the position of x.

2. **Pop (Stack S) -> Stack S';**

   if stack is not empty then,

   This function deletes the item x pointed by the top of the stack and returns new Stack S' with Top pointing to the item down to the deleted item.

3. **isEmpaty (Stack S) -. TRUE / FALSE;**

   This function returns TRUE when S is empty else FALSE.

4. **isFull (Stack S) -. TRUE / FALSE;**

   This function returns TRUE when S is full else FALSE.

Arrays & Stack (Difference)

SR	Array	Stack
i	We can insert an element at any location in an array.	A new element can be added only at the top of a stack.
ii	Any element of an array can be accessed.	Only the topmost element can be accessed.
iii	An array essentially contains homogenous elements.	A stack may contain diverse elements.
iv	An array is a static data structure i.e. Its size remains constant.	A stack is a dynamic data structure i.e. its size grows and shrinks as elements are pushed and popped.
v	There is an upper limit on the size of the array, which is specified during declaration.	Logically, a stack can grow to any size.

Declaring a Stack

We can use typedef to create STACK as a user defined type.

```
#define MAX 100
Typedef struct
{
            int top;
            int data [MAX];
}           STACK;
```

1. **Creating a Stack**: To create a stack, we should create a variable of the abc structure.

   Example

   STACK s;

   This will create a stack named s.

2. **Initializing a Stack**: When a stack is created, top has to be initialized to indicate that the stack is empty at the beginning.

   Since we are using an array, the first element will be stored at position 0. Hence to indicate an empty stack, top should be initialized to -1. This can be done as follows:

   s.top = -1;

Declaring a Stack

3. **The Push Operation**: To push an element "num" into the stack, we should first increment top and then store the element in the array at position top. The steps are shown bellows:

    s.top++;
    s.data[s.top] = num;
    This can also be written as:
    s.data [++s.top] + num;

4. **The Pop Operation**: To pop the topmost element from the stack, we should first access the internet at the top position and then decrement top. The steps are shown below:

    num = s.data [s.top];
    s.top--;
    This can also be written as:
    num = s.data [s.top--];

Declaring a Stack

5. **The Isempty Operation**: An element can be popped only if the stack is not empty. To whether the stack is empty, we can chek if top has-1.

    if (s.top == -1)

6. **The Isfull Operation**: If the value of top reaches the maximum array index i.e. MAX-1, no more elements can be pushed into the stack. This indicates a full stack.

    if (s.top == MAX-1)

We can define all the operations in the form of functions. However, since the function will modify the stack variable, we should pass the address of al the stack variable to the function. Hence, the function will use a pointer to STACK to access the stack members.

Static implementation of Stack Algorithm

```
1.  Start
2.  Top = -1
3.  Accept choice “Push” or “Pop”
4.  If choice == “Push” then
          if top == max-1 then
          printf ”stack overflow”

          else
                    top = top+1
                    stack [top] = num
5.  If choice == “Pop” then
          if top ==-1 then
          printf “stack underflow”

          else
                    top = top-1
6.  Stop
```

Static implementation of Stack (Example)

We can illustrate the concepts above with the help of diagrams. We will accept of size 3.

1. Initially, the stack is empty

2. Push 10

   top = top+1=-1+1 = 0

3. Push 20

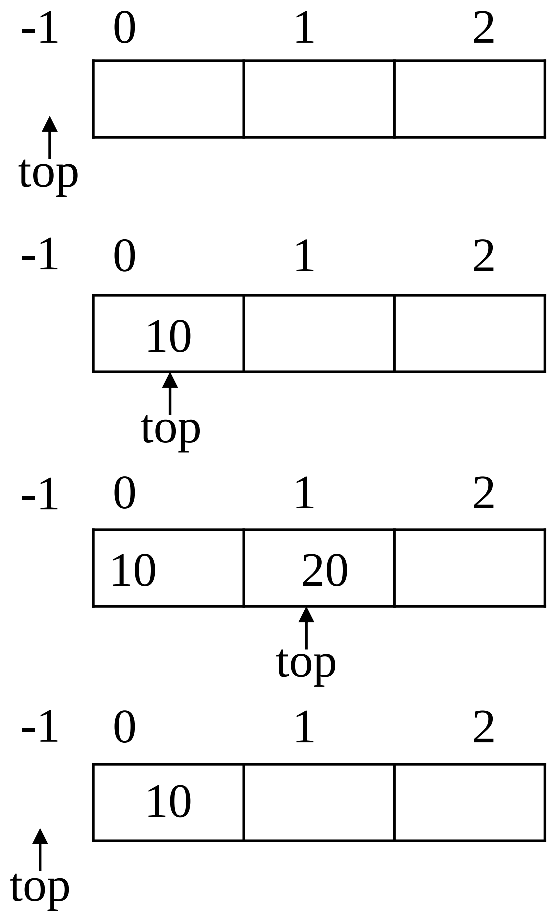
   top = 0+1 = 1

4. Pop

   top = top-1 = 1-1 = 0

5. Pop

   top = top-1 = 0-1 = -1



Application of Stack

1. Reversing a list.
2. Polish Notations.
3. Conversion of an infix to postfix expression.
4. Evaluation of postfix expression.
5. Conversion of an infix to prefix expression.
6. Evaluation of prefix expression.
7. Recursion.

Infix Notations & Prefix Notations

(i) Infix Notations: When the operators exist between two operands then the expression is called infix expression.
Example: The expression to add two numbers A and B is written in infix notation as:

A+B

(ii) Prefix Notations: When the operators are written before their operands, than the expression is called the prefix notation or prefix polish notation.
Example: The expression to add two numbers A and B is written in prefix notation as:

+AB

(iii) Postfix Notations: When the operators are written after the operands, it is called postfix or suffix notations. It is also known as reverse polish notation.
Example: The expression to add two numbers A and B is written in postfix notation as:

AB+

Notations

Infix Expression	Prefix Expression	Postfix Expression
A+B	+AB	AB+
(A -C)*B	*- ACD	AC-*B
(A +B)/(C - D)	/+ AB - CD	AB + CD -/
A + (C*D)	+A*CD	ACD*+

Operator Precedence

Operator	**Symbol**	**Precedence**
Exponential	$	Highest
Multiplication / Division	*,/	Next Highest
Addition / Subtraction	+,-	Lowest

Converting Infix to Postfix Expression Algorithm

```
1.  Start
2.  opstk is an empty stack.
3.  ch is the next character from the infix string.
4.  If ch is an operand
        add ch to postfix string.
5.  If ch is ‘(’
        push ch into opstk
6.  If ch is an operator
        while priority (top-of-stack) > = priority (ch) pop form opstk and add to postfix string.
        push ch into opstk
7.  If ch is ‘)’
        ch1 = pop from opstk
        while (ch1 is not ‘(’ and opstk is not empty)
                add ch1 to postfix string
                ch1 = pop (opstk)
8.  If infix string has not ended
        go to Step 3
9.  While opstk is not empty.
        pop ch and add to postfix string
10. Display postfix string.
11. stop
```

Evaluation of Postfix Expression

Given Expression: $((A + B) * D) \uparrow (E - F)$

SR	Symbol scanned	Stack	Expression
1	(	(	~
2	(	((	~
3	(	(((	~
4	A	(((	A
5	+	(((+	A
6	B	(((+	AB
7	)	((	AB+
8	*	((*	AB+
9	D	((*	AB + D
10	)	(	AB + D*
11	↑	(↑	AB + D*
12	(	(↑(	AB + D*
13	E	(↑(	AB + D * E
14	-	(↑(-	AB + D * E
15	F	(↑(-	AB + D * EF
16	)	(↑	AB + D * EF-
17	)		$\boxed{\text{AB + D * EF-}}$

Given Expression: (A * B – C$\uparrow$D + E/F)

SR	Symbol scanned	Stack	Expression
1	(	(	~
2	A	(	A
3	*	(*	A
4	B	(*	AB
5	-	(-	AB*
6	C	(-	AB*C
7	$\uparrow$	(- $\uparrow$	AB*C
8	D	(- $\uparrow$	AB*CD
9	+	(- +	AB*CD $\uparrow$
10	E	(- +	AB*CD$\uparrow$E
11	/	(- +/	AB*CD$\uparrow$ E
12	F	(- +/	AB*CD $\uparrow$ EF
13	)		$\boxed{\text{AB*CD}\uparrow\text{EF/ + -}}$

Given Expression: 6,2,3,+,-,3,8,2,/,+,*,2,↑,3,+

SR	Symbol scanned	Stack
1	6	6
2	2	6,2
3	3	6,2,3
4	+	6,5
5	-	1
6	3	1,3
7	8	1,3,8
8	2	1,2,8,2
9	/	1,3,4
10	+	1,7
11	*	7
12	2	7,2
13	↑	49
14	3	49,3
15	+	$\boxed{52}$

Example: Convert infix string ((A+B) * (C - D) into prefix string with stack operations show the content of stack in each step.

SR	Symbol scanned	Stack	Expression
1	)	)	~
2	)	))	~
3	D	))	D
4	-	))-	D
5	C	))-	CD
6	(	)	-CD
7	*	)*	-CD
8	)	)*)	-CD
9	B	)*)	B – CD
10	+	)*)+	B – CD
11	A	)*)+	AB – CD
12	(	)*	+AB – CD
13	(	-	$\boxed{\text{*+AB – CD}}$

Example: Convert the given string to prefix exprssio0n and shows the details of stack at each step.
(A- B/C) * (D * E - F)

SR	Symbol scanned	Stack	Expression
1	)	)	~
2	)	))	~
3	F	))	F
4	-	))-	F
5	E	))-	EF
6	*	))-*	EF
7	D	))-*	DEF
8	(	)	-*DEF
9	*	)*	-*DEF
10	)	)*)	-*DEF
11	C	)*)	C - *DEF
12	/	)*)/	C - *DEF
13	B	(*)/	BC - *DEF
14	-	)*)-	/BC - *DEF
15	A	)*)-	A/BC - *DEF
16	(	)*	-A/BC - *DEF
17	(		$\boxed{\text{*-A/BC - *DEF}}$

Recursion (Definition)

Recursion is a process of expressing a fraction. In them of itself in recursion.
It is essential for a fraction to call itself, otherwise recursion will not into replace only user depend function can be involved in the recursion library function cannot be involved in recursion became their source code cannot be viewed.

Recursion (Example)

Write a program to find factorial of number using recursion.

```
/* program to find factorial of given number */
#include <studio.h>
#include <conio.h>
Voide main()
{
        int fact(int);
        int num;
        clrscr();
        printf(“Enter the number\n”);
        scanf(“%d”, & num);
        printf(“The factorial of %d=%d”,num,fact(num));
        getch();
        }
        int fact(int n)
        {
                if(n==0)
                        return(1);
                return(n * fact(n-1));
}
```

Tower of Hanoi (Diagram)