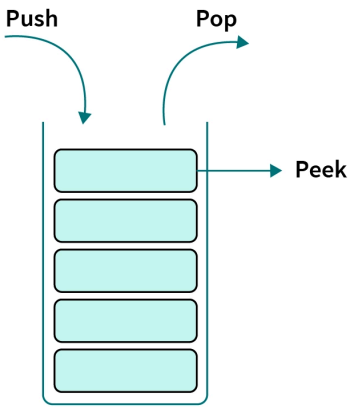
Practical-4

Stack Operations & Application

It is an Abstract data type (ADT) in which data is inserted and deleted from a single end which is the stack's top. By definition, a stack is an ordered list in which insertion and deletion are done at one end, where the end is generally called the top. The last inserted element is available first and is the first one to be deleted. Hence, it is known as Last In, First Out (LIFO) or First In, Last Out (FILO). In a stack, the element at the top is known as the top element.

A Stack is an abstract linear data structure serving as a collection of elements that are inserted (also known as push operation) and removed (also known as pop operation) according to the Last in First Out (LIFO) approach or First In Last Out (FILO) approach.

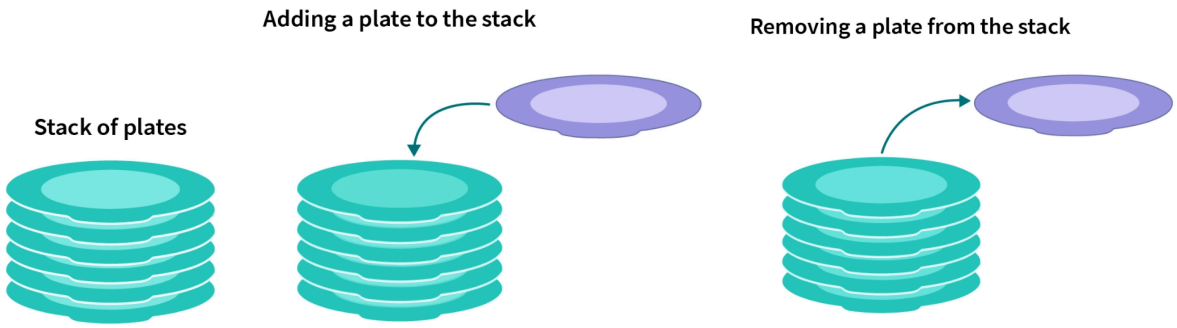
Insertion and deletion happen on the same end (top) in a Stack. The top of the stack is returned using the peek operation.



In C, the Stack data structure is an ordered, linear sequence of items. It is a LIFO (Last In First Out) data structure, which means that we can insert or remove an item at the top of the stack only. It is a sequential data type, unlike an array.

In an array, we can access any of its elements using indexing, but we can only access the top most element in a stack. The name "Stack" for this data structure comes from the analogy to a set of physical items stacked on top of each other.

An example of this data structure would be a stack of plates: We can only add a plate to the top of the stack and take a plate from the top of the stack. A stack of coins or a stack of books can also be a real-life example of the stack data structure.



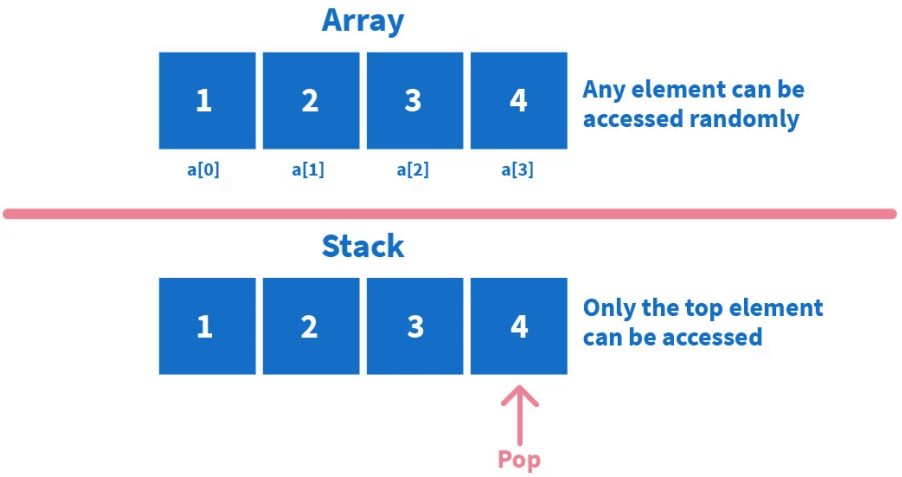
Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns the top element of the stack.
- **isEmpty:** Returns true if the stack is empty, else false.

"What is the difference between arrays and stacks?"

Stacks differ from the arrays in a way that in arrays random access is possible; this means that we can access any element just by using its index in the array. Whereas, in a stack only limited access is possible and only the top element is directly available.

Consider an array, arr = [1, 2, 3, 4]. To access 2, we can simply write arr[1], where 1 is the index of the array element, 2. But if the same is implemented using a stack, we can only access the topmost element that is 4.



Stacks are dynamic in nature; this means that they do not have a fixed size and their size can be increased or decreased depending upon the number of elements.

A Stack data structure can be implemented/ grouped in two types Static and Dynamic

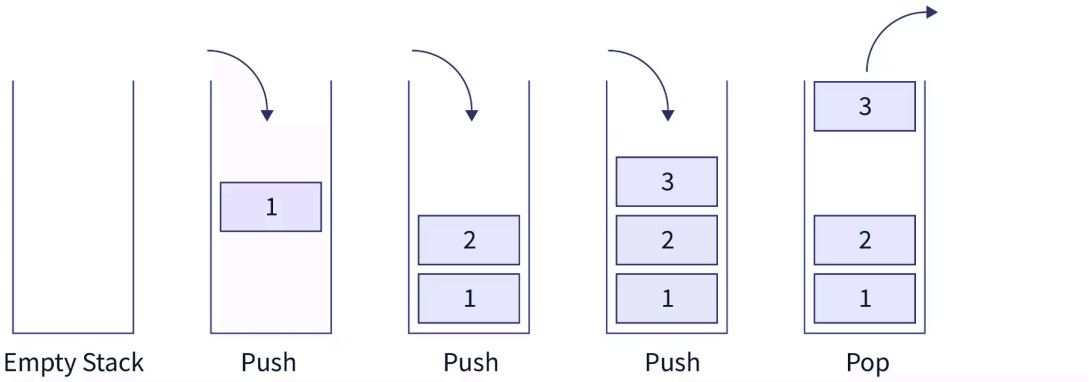
- A Static Stack (also known as a bounded stack) has a bounded capacity. It can contain a limited number of elements. If a static stack is full and does not have any space remaining for another element to be pushed to it, it is then called to be in an Overflow State. In C, a static stack is implemented using an Array, as arrays are static.

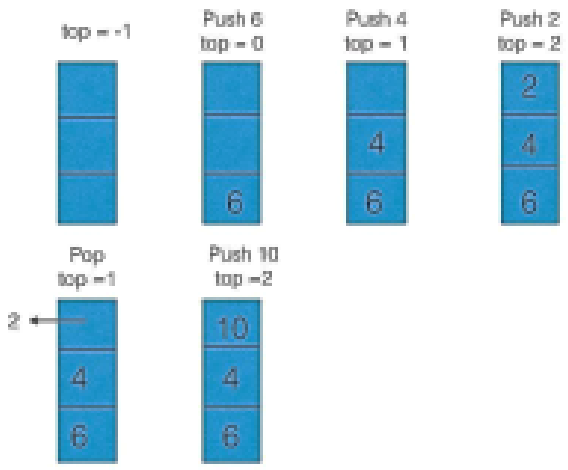
- A Dynamic Stack is a stack data structure whose capacity (maximum elements that can be stored) increases or decreases in runtime, based on the operations (push or pop) performed on it. In C, a dynamic stack is implemented using a Linked List, as linked lists are dynamic data structures. (....?)

In C, the stack data structure works using the LIFO (Last In First Out) approach. Initially, we set a Peek pointer to keep track of the topmost element of the stack. Then the stack is initialized to -1 as its peek, as we add (push) elements to the stack, the peek gets updated to point to its topmost element, and if we remove (pop) elements from the stack, the peek gets reduced.

- We use the **Push** operation to add an element to the top of the stack.
- We use the **Pop** operation to return and remove the topmost element of the stack.
- We use the **Peek** Operation to display the topmost element of the stack.
- We use the **IsEmpty** Operation to check whether the stack is empty or not.
- We can use the **IsFull** Operation to check whether the stack is full or not.

This operation can only be used with the static implementation of the stack (using an array), also called a bounded stack.



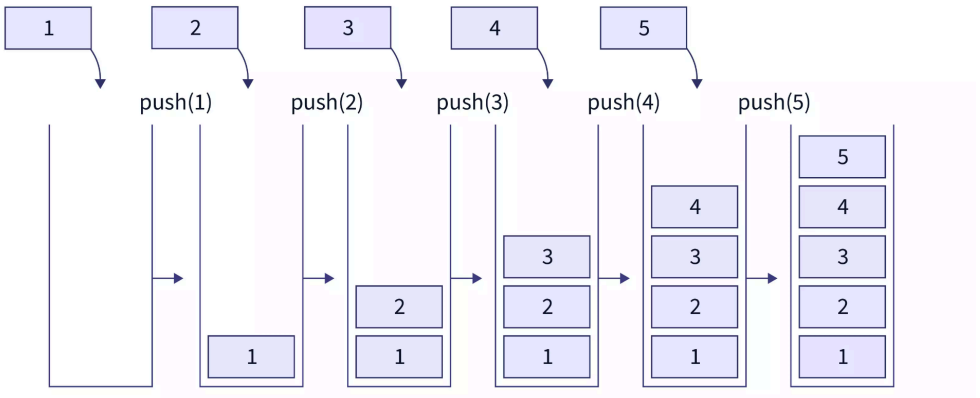


Stack Operations

1. push(x)

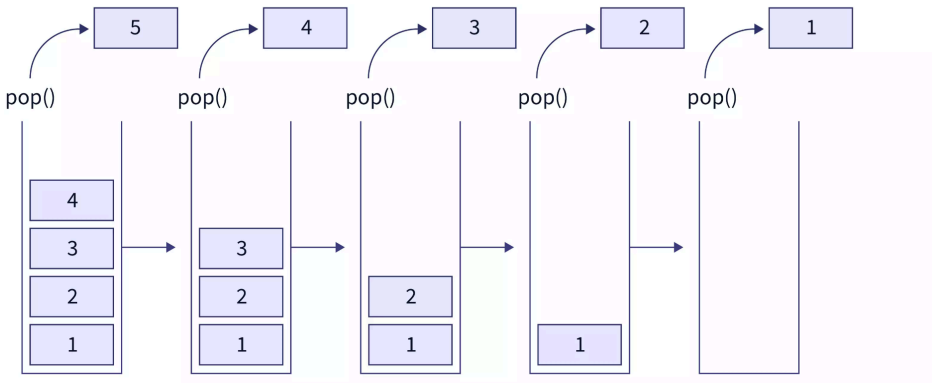
   The process of inserting new data into a stack is known as push. The push (x) operation takes the element to be added as a parameter. If we are implementing a stack using an array, it may have a pre-defined capacity , which means that only a specific number of elements can be stored in it. In this case, if the stack is full, it results in a condition called stack overflow. It indicates that we have utilized the entire capacity of the given stack, and there is no space for any new element.

   The steps involved in a push operation are –
   - Before inserting the element, check whether the stack is full.
   - If the stack is full, print “stack overflow” and terminate the program.
   - If the stack is not full, add data into the stack.
   - We can repeat the above steps until the maximum capacity of the stack is achieved.



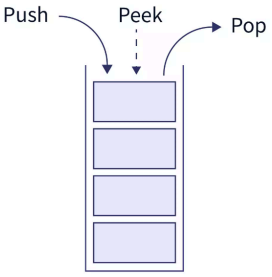
2. pop()

   The process of deleting the topmost element from the stack is known as pop. In stacks, if we try to pop or remove elements if the stack is empty, it results in a condition known as underflow. It means that there is no element in the stack that can be removed.
   The steps involved in a pop operation are –
   - Before removing the topmost element, check whether the stack is empty
   - If the stack is empty, print “Stack underflow” and terminate the program
   - If the stack is not empty, access the topmost element and let the top point to the element just before it
   - We can repeat the above until all the elements from the stack are removed



3. topElement()/peek()

   This operation on a stack returns the topmost element in a stack. This operation is also known as the peek() operation on the stack.



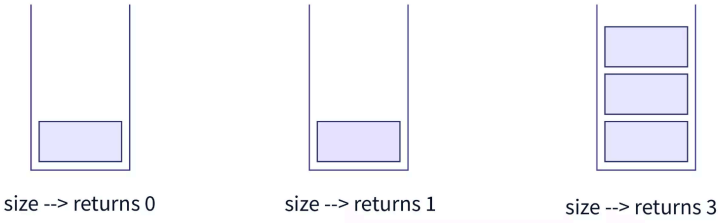
4. isEmpty()

   This operation is used to check if the given stack is empty. It returns a boolean value; that is true when the stack is empty, otherwise false.

5. isFull()

   This operation is used to check if the given stack is full. It returns a boolean value; that is true when the stack is full, otherwise false.

6. size()
   This operation determines the current size of the stack



Time Complexity of Stack Operations

Push Operation: O(1)

- The time complexity of the Pop operation would be O(1) as in this operation, we are inserting an element at the top of the stack only.

Pop Operation: O(1)

- The time complexity of the Push operation would be O(1) as in this operation, we are removing and returning an element from the top of the stack only.

Peek Operation: O(1)

- The time complexity of the Peek operation would be O(1) as in this operation, we are returning only the topmost element of the stack.

IsEmpty Operation: O(1)

- The time complexity of the IsEmpty operation would be O(1) as in this operation, we are checking whether the topmost element is null or not.

IsFull Operation: O(1)

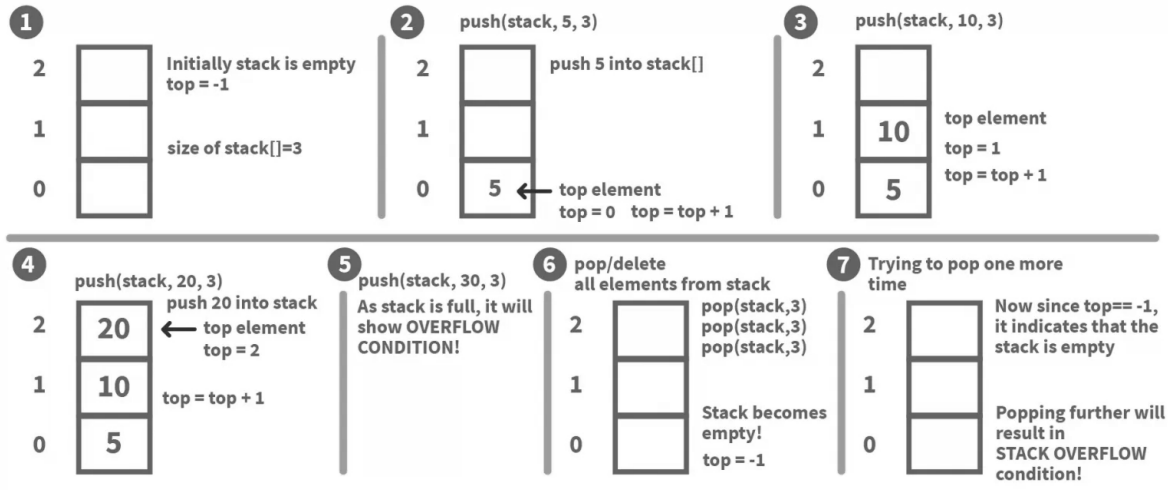
- The time complexity of the IsFull operation would be O(1) as in this operation, we are checking whether the topmost element is at the maximum position or not.

Implementation of Stack using Array

A stack is a linear data structure that follows LIFO(Last In First Out) approach, which means the element added at last in the stack will be removed first and the specific order can't be changed. The stack can be implemented in multiple ways. In stack implementation using an array, we will do all the operations of the stack data structure using an array.

Implementation of Stack using a One-dimensional Array

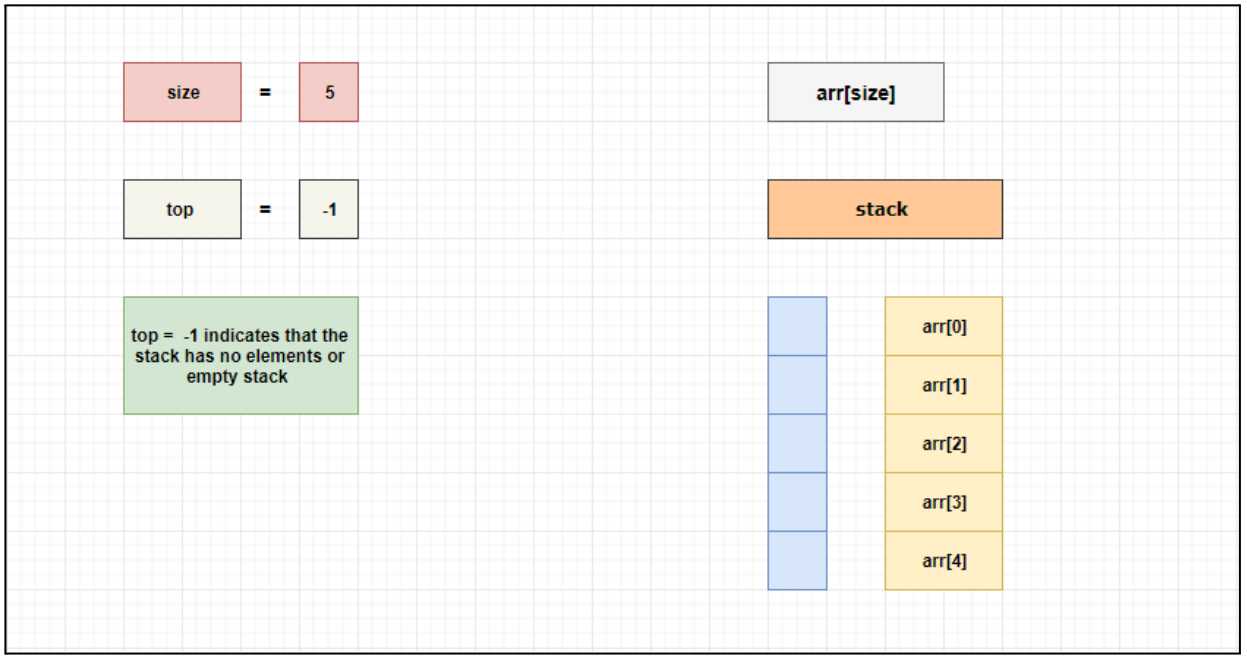
- A variable called top is used to keep track of the top element in the stack
    - When initializing the stack, we set the value of top equal to -1 so that we can check if the stack is full or empty using its value
- Declare a function push() that takes an array called stack[], the size of the array called n, and the element to be inserted called x as its parameters.
    - Check if the stack is already full. If it is full, return from the function
    - If the stack is not full, increase the value of top and place the new element in the position pointed to by top
- Declare a function called pop() that removes the top element from the stack.
    - Check if the stack is empty. If it is, return from the function
    - If the stack is not empty, return the element at the top index in the array stack[] and reduce the value of top Note – we are not deleting the value at top index during pop() because once the pointer is moved to point to the previous element, it does not matter whatever is contained in that memory location.
- Declare a function called topElement()/ peek() that returns the element at the top index from the array stack[]
- Declare a function size() that returns the occupied size of the array stack[]. It returns top + 1.
- Declare a function isEmpty() that checks if top is equal to -1. If it is, it returns true, otherwise false
- Declare a function isFull() that checks if the top is equal to the maximum size of the array stack[]. If it is, it returns true, otherwise false



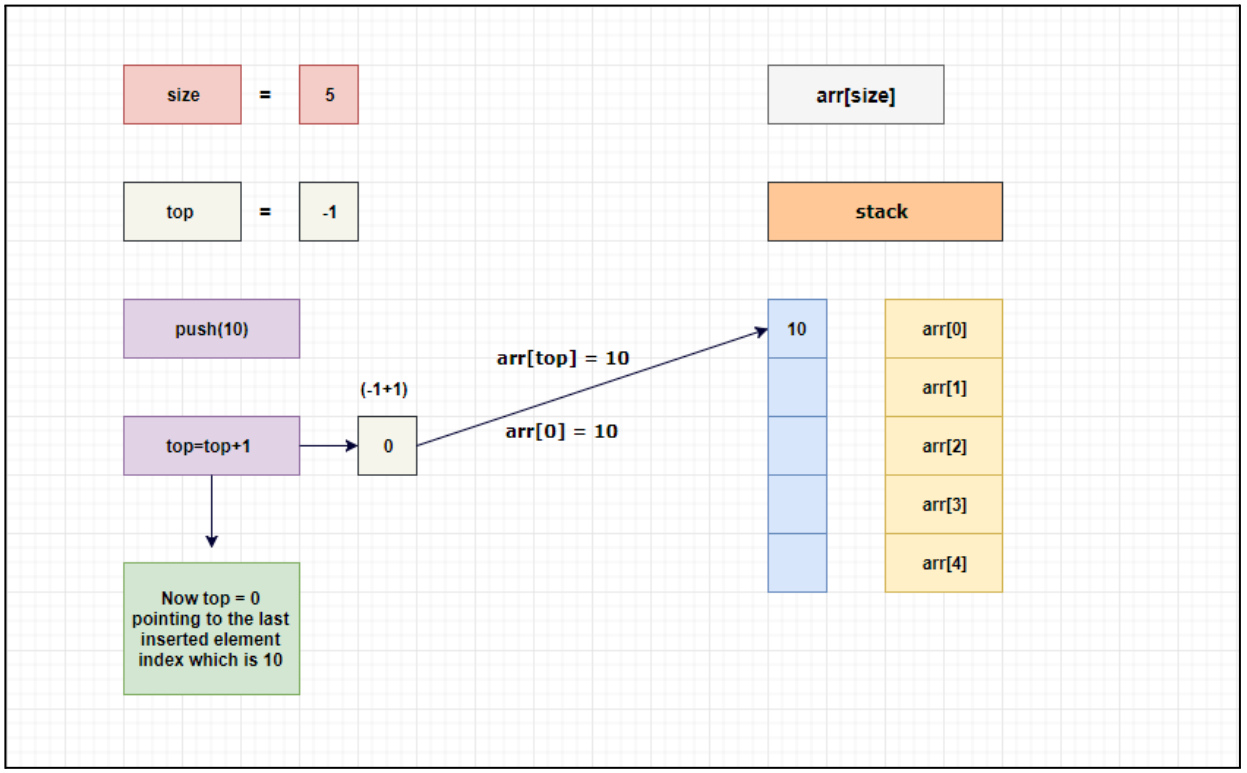
Limitations of implementing stacks using an array –

- The maximum size of the array must be defined in advance
- The size of the array cannot be changed during the implementation

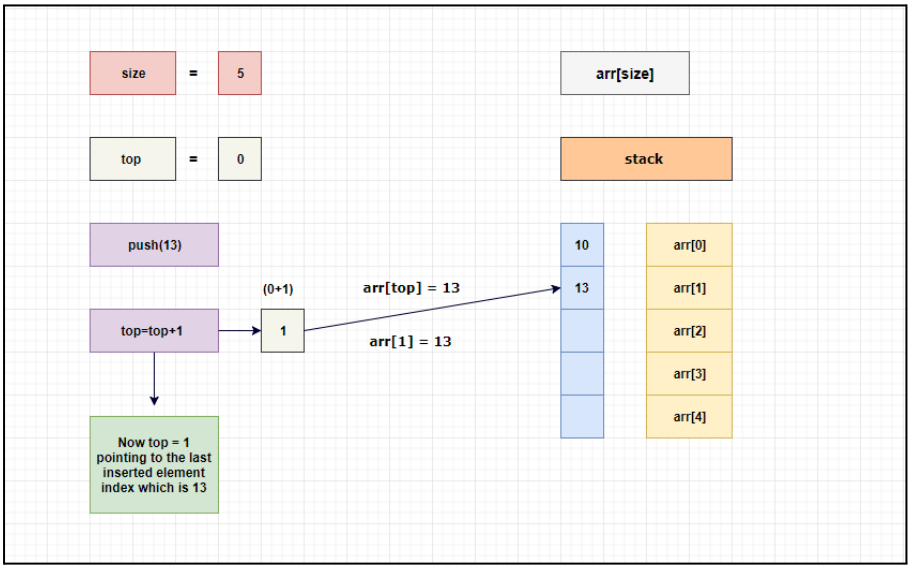
Initialize the variables



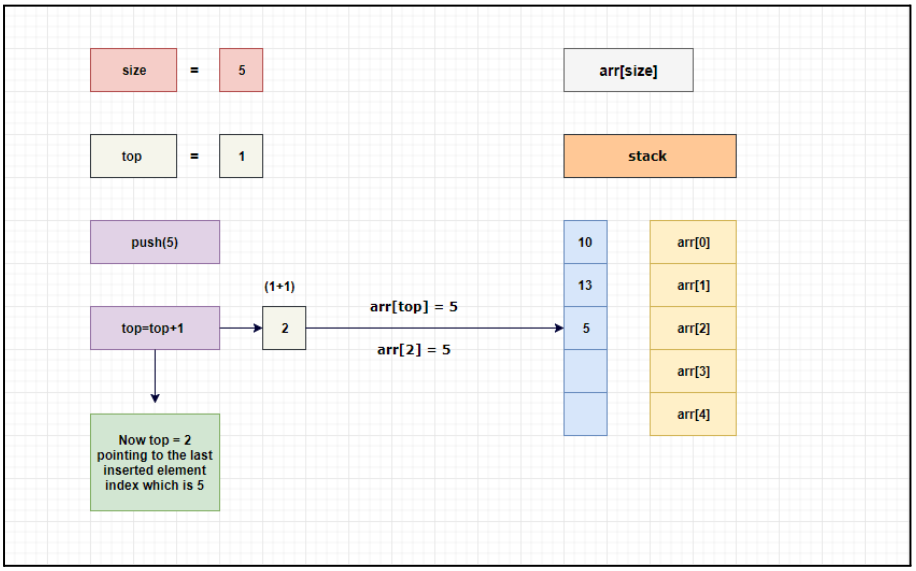
Push 10



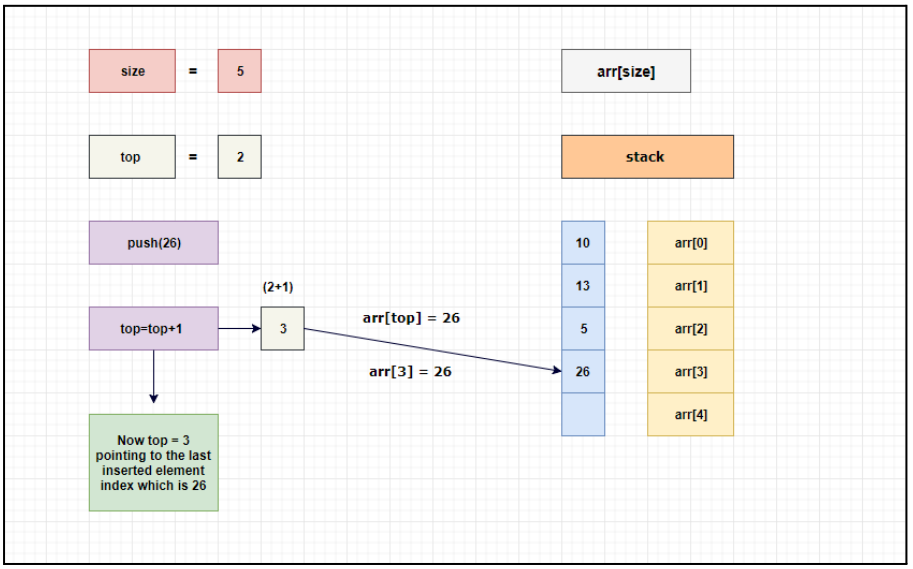
Push 13



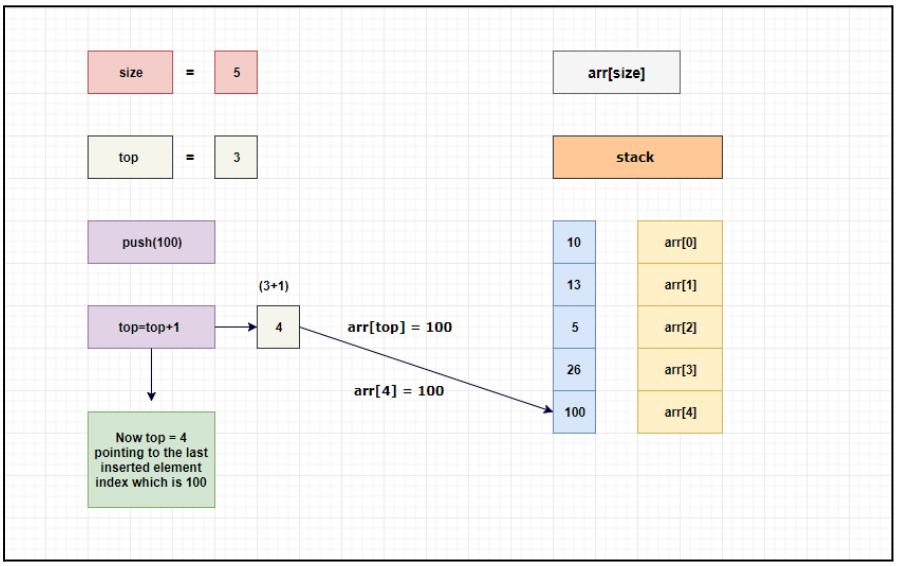
Push 5



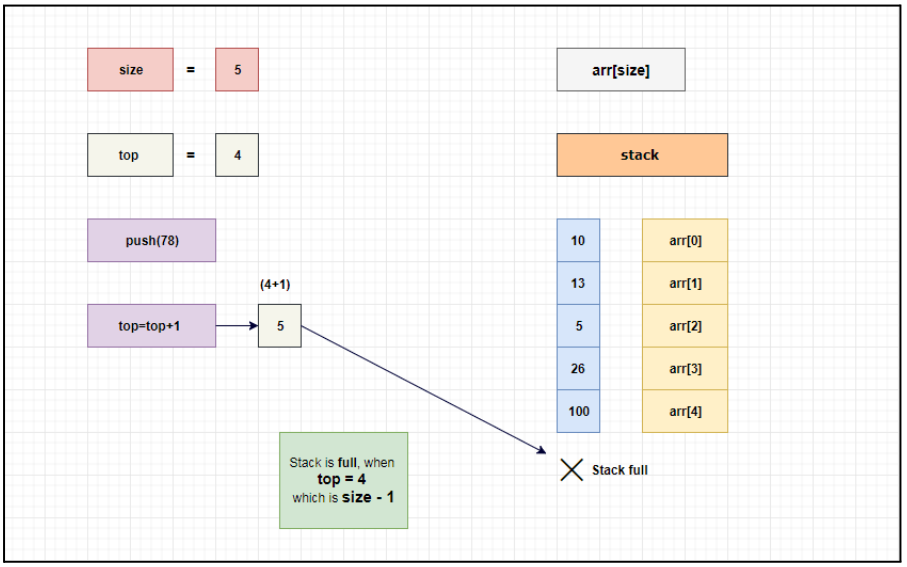
Push 26



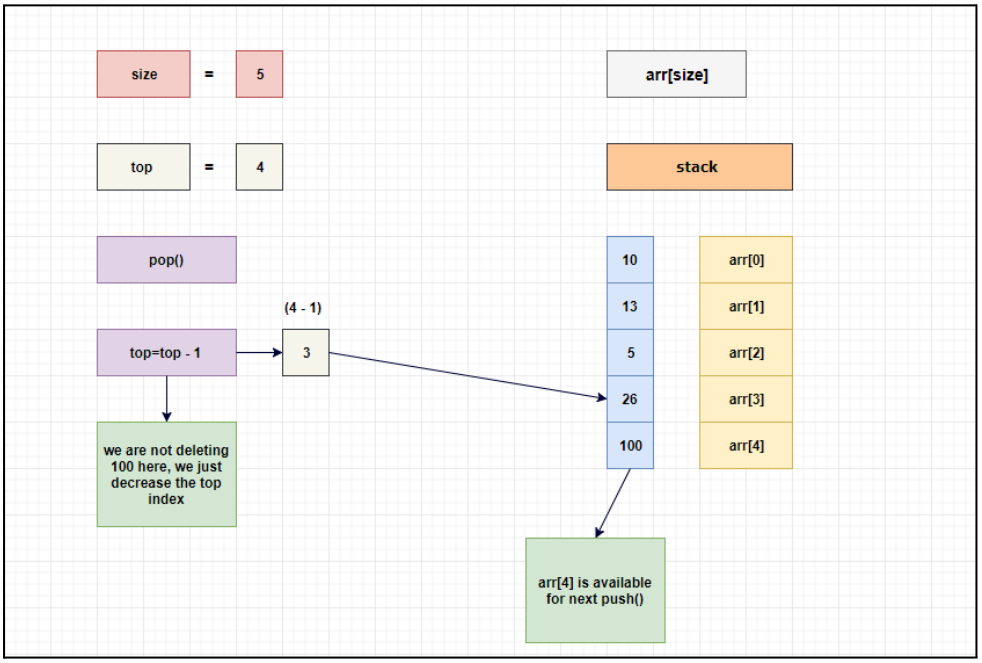
Push 100

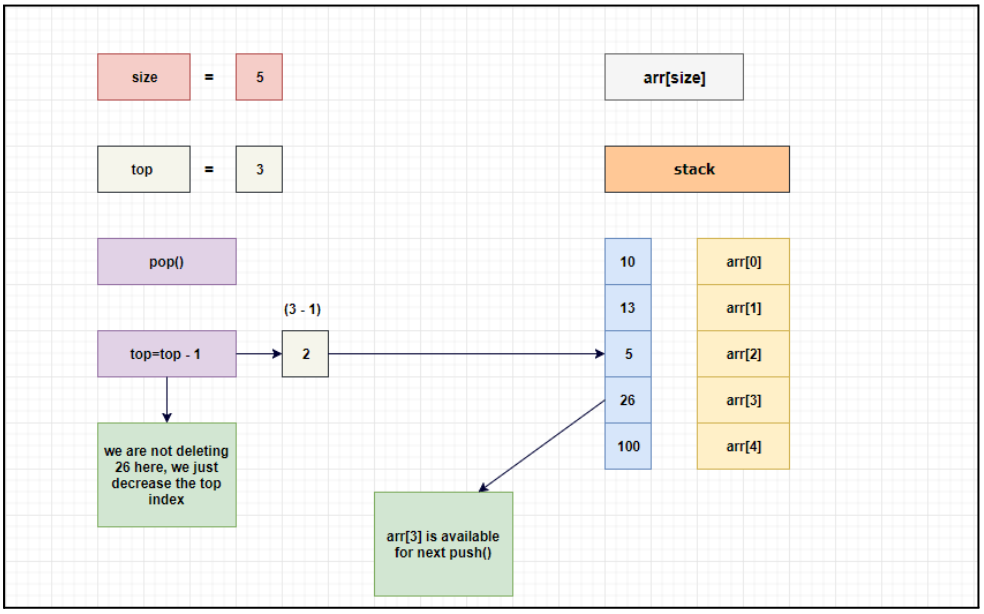


Push 78



Pop


size
=
5
arr[size]
top
=
4
stack
pop()
(4 - 1)
top=top - 1
3
10
13
5
26
100
arr[0]
arr[1]
arr[2]
arr[3]
arr[4]
we are not deleting 100 here, we just decrease the top index
arr[4] is available for next push()


size
=
5
arr[size]
top
=
3
stack
pop()
(3 - 1)
top=top - 1
2
10
13
5
26
100
arr[0]
arr[1]
arr[2]
arr[3]
arr[4]
we are not deleting 26 here, we just decrease the top index
arr[3] is available for next push()

Define Stack

```
// N will be the capacity of the Static Stack
#define N 100

// Initializing the top of the stack to be -1
int top = -1;

// Initializing the stack using an array
int stack[N];

// Function prototypes
void push();   // Push element to the top of the stack

int pop();     // Remove and return the top most element of
                    //the stack

int peek();    // Return the top most element of the stack

bool isEmpty();// Check if the stack is in Underflow state
                    //or not

bool isFull(); // Check if the stack is in Overflow state
                    //or not
```

Adding an Element Onto the Stack/ Push Operation

Push operation executes in two steps:

1. Increment the variable top (the pointer that points to the top of the stack). Now it will point to a new memory location.
2. Add the new element at the updated top, the new memory location. And increase the size of the stack.

- Push operation can cause stack overflow conditions if we try to perform push operation on an already full stack.

Algorithm:

```
begin
    if top = n it means the stack is full
    else
        top = top + 1
        stack (top)  = new_item;
End
```

Code:

```
void push (int value, int n)
{
    if (top == n )
        printf("\n Overflow");
```

```
    else
    {
        top = top +1;
        stack[top] = val;
    }
}
```

Deletion of an Element From a Stack/ Pop Operation

Retrieving the value while removing it from the top of the stack, is known as a pop operation.

1. In an array implementation of pop() operation in the stack data structure, the data element is not removed, instead, the top pointer is decremented to a lower position in the stack to point to the next value, and simultaneously the size is decreased.
2. The pop function will return this removed value.

- Underflow conditions can occur in pop operation if we try to perform a pop operation on an empty stack.

Algorithm:

```
begin
    if top = 0 it means stack is empty;
    else
        top_item = stack(top);
        top = top - 1;
end;
```

Code:

```
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top-- ];
    }
}
```

Accessing the Top Element of the Stack/ Peek Operation

1. In peek operation we return the element which is present at the top of the stack without deleting it.

- If we try to iterate over an empty stack then the stack underflow condition occurs.

Algorithm:

```
Begin
    if top = -1 it means stack is empty
    else
        item = stack[top]
    return item
End
```

Code:

```
int peek()
{
    if (top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack [top];
    }
}
```

IsEmpty and IsFull:

```
bool isEmpty(){
    if(top == -1){
        printf("Stack is empty: Underflow State\n");
        return true;
    }
    printf("Stack is not empty\n");
    return false;
}

bool isFull(){
    if(top == N-1){
        printf("Stack is full: Overflow State\n");
        return true;
    }
    printf("Stack is not full\n");
    return false;
}
```

Sample 1:

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 10
void push(int);
void pop();
void display();
int stack[SIZE], top = -1;

void main()
{
    int value, choice;
    while(1)
    {
         printf("\n\n***** MENU *****\n");
         printf("1. Push\n2. Pop\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);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("\nWrong selection!!! Try again!!!");
         }
    }
}
void push(int value){
    if(top == SIZE-1)
        printf("\nStack is Full!!! Insertion is not
possible!!!");
    else
    {
        top++;
        stack[top] = value;
        printf("\nInsertion success!!!");
    }
}
```

```
void pop()
{
    if(top == -1)
        printf("\nStack is Empty!!! Deletion is not
possible!!!");
    else
    {
        printf("\nDeleted : %d", stack[top]);
        top--;
    }
}
void display()
{
    if(top == -1)
        printf("\nStack is Empty!!!");
    else
    {
        int i;
        printf("\nStack elements are:\n");
        for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
    }
}
```

Sample 2:

```
// Implementing Static Stack using an Array in C
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define N 1000
int top = -1;
int stack[N];

void push();
int pop();
int peek();
bool isEmpty();
bool isFull();

int main(){
    printf("STATIC ARRAY (Total Capacity: %d)\n", N);
    int choice;

    while(1){
        printf("\nChoose any of the following options:\n");
        printf(" 0: Exit            1: Push            2: Pop
3: Peek\n");
```

```
        printf(" 4: Check if the stack is empty     5: Check
if the stack is full\n\n");
        scanf("%d", &choice);

        switch(choice){
            case 0: exit(0);
            case 1: push(); break;
            case 2: pop(); break;
            case 3: peek(); break;
            case 4: isEmpty(); break;
            case 5: isFull(); break;
            default: printf("Please choose a correct
option!");
        }
    }
    return 0;
}

void push(){
    // Checking overflow state
    if(top == N-1)
        printf("Overflow State: can't add more elements into
the stack\n");
    else{
        int x;
        printf("Enter element to be pushed into the stack: ");
        scanf("%d", &x);
        top+=1;
        stack[top] = x;
    }
}

int pop(){
    // Checking underflow state
    if(top == -1)
        printf("Underflow State: Stack already empty, can't
remove any element\n");
    else{
        int x = stack[top];
        printf("Popping %d out of the stack\n", x);
        top-=1;
        return x;
    }
    return -1;
}

int peek(){
    int x = stack[top];
    printf("%d is the top most element of the stack\n", x);
    return x;
```

```
}

bool isEmpty(){
    if(top == -1){
        printf("Stack is empty: Underflow State\n");
        return true;
    }
    printf("Stack is not empty\n");
    return false;
}

bool isFull(){
    if(top == N-1){
        printf("Stack is full: Overflow State\n");
        return true;
    }
    printf("Stack is not full\n");
    return false;
}
```

Applications of Stack

In this section we will discuss typical problems where stacks can be easily applied for a simple and an efficient solution. Some of the common application so stack are:

- ➔ Reversing a list
- ➔ Parentheses checker
- ➔ Conversion of an infix expression into a postfix expression
- ➔ Evaluation of a postfix expression
- ➔ Conversion of an infix expression into a prefix expression
- ➔ Evaluation of a prefix expression
- ➔ Recursion
- ➔ Tower of Hanoi

- **Expression Handling**

  Stack data structures are highly effective in evaluating arithmetic operations, basically, it evaluates the expression by converting it into a specific notation (prefix or postfix) and then calculating its values.
  There are three common notations for expressing an arithmetic expression.

  1. Infix Notation Each operator is placed between the operands in the infix notation, which is a convenient manner of constructing an expression. Depending on the situation, infix expressions may or may not be parenthesized.

     ```
     (A+B)+C*(E-F) is an example of Infix Notation
     ```

  2. Prefix Notation The operator is placed before the operands in the prefix notation. This system was created by a Polish mathematician, and is thus known as polish notation.

     ```
     + A B and *+ABC are the examples of Polish notation
           Where *+ABC = (A+B)*C in infix notation
     ```

  3. Postfix Notation The operator is placed after the operands in postfix notation. This notation is known as Reverse Polish notation since it is the inverse of Polish notation.

     ```
     AB +   // example
     ```

- **Backtracking**

  One of the algorithm design techniques is backtracking. For that aim, we dig into one path; if that path is inefficient, we return to the previous state and explore alternative options. We must save the previous state in order to return from the current state. Stack is required for this purpose. A famous example is N-queens problems.

- **Reverse a Data**

  We already know that the data is processed in a LIFO (last in, first out) way by the stack i.e. the last element will automatically be returned first, hence resulting in the reversed data.

  Consider a string s = "Apple" and we want to reverse it. The approach would be to push all the elements from the beginning of the string and pop out then, as the last entered element will be the first to come out of the stack, and the same is followed for the rest of the elements, resulting in the reversed string "elppA".

- **Parenthesis Checking**

  One of the most common uses of the stack is to see if the provided expression's parentheses are balanced.

  Stack is one way to check for balanced parentheses. When you come across an open parentheses, put it into the stack, and when you come across a closed parenthesis, match it to the top of the stack and pop it, if the stack is empty, return unbalanced. Return balanced if the stack is empty at the end, unbalanced Otherwise.

  This is one of the stack's most common applications, and its logic may be applied to any parenthesis checking problem.

- **Processing Function Calls**

  We often see terms like call stack and recursion stack often being used in programming. Whenever a function(A) calls another function(B), the address of caller function is stored and when the call of function B is finished, the computation for function A is performed and address of A is removed from the stack and the appropriate output is returned. Consider the example of two functions A() and B(), the function A() calls B() first and then prints a statement, on the other hand, B() only prints a single statement and terminates the call from there.

Evaluation of Postfix Expressions

The compiler finds it convenient to evaluate an expression in its postfix form. The virtues of postfix form include elimination of parentheses which signify priority of evaluation and the elimination of the need to observe rules of hierarchy, precedence and associativity during evaluation of the expression. As **Postfix expression** is without parentheses and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

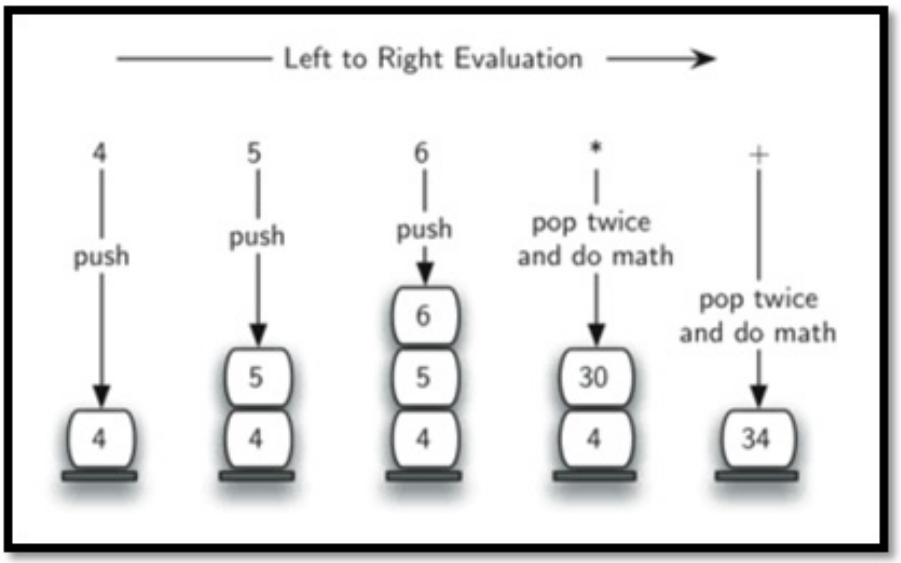
Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

1. Add ‘)’ to the postfix expression.
2. Read postfix expression Left to Right until ‘)' encountered
3. If operand is encountered,
   a. push it onto Stack [End If]
4. If operator is encountered, Pop two elements
   a. i) A -> Top element
   b. ii) B-> Next to Top element
   c. iii) Evaluate B operator A
   d. iv) push B onto Stack
5. Set result = pop
6. END

Example Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Result: 34

Input	**Stack** *(top is on the left)*	
2 3 4 * +	*empty*	Push 2
3 4 * +	2	Push 3
4 * +	3 2	Push 4
* +	4 3 2	Pop 4, pop 3, do 3 *4 , push 12
+	12 2	Pop 12, Pop 2, do 2 + 12, push 14
	14	

Input	**Stack**	
3 4 * 2 5 * +	*empty*	Push 3
4 * 2 5 * +	3	Push 4
* 2 5 * +	4 3	Pop 4, pop 3, do 3*4, Push 12
2 5 * +	12	Push 2
5 * +	2 12	Push 5
* +	5 2 12	Pop 5, Pop 2, do 2*5, Push 10
+	10 12	Pop 10, Pop 12 do 12 + 10, push 22
	22	

The following algorithm converts infix to postfix.

- Scan input string from left to right character by character.
- If the character is an operand, put it into output stack.
- If the character is an operator and the operator's stack is empty, push the operator into operators' stack.
- If the operator's stack is not empty, there may be the following possibilities.
    - If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operand's stack.
    - If the precedence of the scanned operator is less than or equal to the top most operator of operator's stack, pop the operators from operand's stack until we find a lower precedence operator than the scanned character.
    - Never pop out (**'('**) or (**')'**) whatever may be the precedence level of scanned character.
    - If the character is an opening round bracket (**'('**), push it into the operator's stack.
    - If the character is a closing round bracket (**')'**), pop out operators from the operator's stack until we find an opening bracket (**'('**).
    - Now pop out all the remaining operators from the operator's stack and push into the output stack.

Code:

```
#include<stdio.h>
#include<ctype.h>
char stack[100];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}
char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}
int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}
int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
```

```
    scanf("%s",exp);
    printf("\n");
    e = exp;
    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c ",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c ", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c ",pop());
            push(*e);
        }
        e++;
    }
    while(top != -1)
    {
        printf("%c ",pop());
    }
    return 0;
}
```

Input	Stack	Postfix
2*3 + 4*5	empty	
*3+4*5	empty	2
3+4*5	*	2
+4*5	*	23
4*5	+	23*
*5	+	23*4
5	*+	23*4
	*+	23*45
	+	23*45*
	empty	23*45*+

Input	Stack	Postfix
2-3+4-5*6	empty	
-3+4-5*6	empty	2
3+4-5*6	-	2
+4-5*6	-	23
4-5*6	+	23-
-5*6	+	23-4
5*6	-	23-4+
*6	-	23-4+5
6	*-	23-4+5
	*-	23-4+56
	-	23-4+56*
	empty	23-4+56*-

Input	Stack	Postfix
(2-3+4)*(5+6*7)	empty	
2-3+4)*(5+6*7)	(	
-3+4)*(5+6*7)	(	2
3+4)*(5+6*7)	(-	2
+4)*(5+6*7)	(-	23
4)*(5+6*7)	(+	23-
)*(5+6*7)	(+	23-4
*(5+6*7)	empty	23-4+
(5+6*7)	*	23-4+
5+6*7)	(*	23-4+
+6*7)	(*	23-4+5
6*7)	+(*	23-4+5
*7)	+(*	23-4+56
7)	*+(*	23-4+56
)	*+(*	23-4+567
	*	23-4+567*+
	empty	23-4+567*+*

Another Example

Postfix

- Step 1: Add ")" to the end of the infix expression
- Step 2: Push "(" onto the stack
- Step 3: Repeat until each character in the infix notation is scanned
    - 3.1: IF a "(" is encountered, push it on the stack
    - 3.2: IF an operand (whether a digit or a character) is encountered, add it postfix expression.
    - 3.3: IF a ")" is encountered, then
      a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
      b. Discard the "(". That is, remove the(from stack and do not add it to the postfix expression
    - 3.4: IF an operator O is encountered, then
      a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has equal or higher precedence than O
      b. Push the operator O to the stack.

  [END OF IF]

- Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
- Step 5: EXIT

Prefix

- Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
- Step 2: Obtain the postfix expression of the expression obtained from Step 1 using the above given postfix algorithm with slight change in Step 3.4

IF an operator O is encountered, then
a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has ~~equal or~~ higher precedence than O

- Step 3: Reverse the postfix expression to get the prefix expression

Infix: 2+4/5*(5-3)^5^4
Postfix : 245/53-5^4^*+

Postfix

Sr. no.	Expression	Stack	Postfix
0		(	
1	2	(	2
2	+	(+	2
3	4	(+	24
4	/	(+ /	24
5	5	(+ /	245
6	*	(+ *	245/
7	(	(+ * (	245/
8	5	(+ * (	245/5
9	-	(+ * (-	245/5
10	3	(+ * (-	245/53
11	)	(+ *	245/53-
12	^	(+ * ^	245/53-
13	5	(+ * ^	245/53-5
14	^	(+ * ^	245/53-5^
15	4	(+ * ^	245/53-5^4
16	)		245/53-5^4^*+

Infix: 2+4/5*(5-3)^5^4
Prefix : +2*/45^^-5354

Prefix
1. Reversed the Infix expression along with swithing (with) and vice versa.
2. Find the Postfix Expression as below table.
3. Reverse the resultant expression like in step 1.

Sr. no.	Expression	Stack	Prefix (Reversed)
0		(	
1	4	(	4
2	^	(^	4
3	5	(^	45
4	^	(^ ^	45
5	(	(^ ^ (	45
6	3	(^ ^ (	453
7	-	(^ ^ (-	453
8	5	(^ ^ (-	4535
9	)	(^ ^	4535-
10	*	(*	4535-^^
11	5	(*	4535-^^5
12	/	(* /	4535-^^5
13	4	(* /	4535-^^54
14	+	(+	4535-^^54/*
15	2	(+	4535-^^54/*2
16	)		4535-^^54/*2+

Postfix Evaluation

- Step 1: Add a ")" at the end of the postfix expression
- Step 2: Scan every character of the postfix expression and repeat Step 3 and 4 until ")" is encountered.
- Step 3: IF an operand is encountered, Push it on the stack
  IF an operator O is encountered, then
    - a: Pop the top two elements from the stack as A and B
    - b: Evaluate B O A, where A is the topmost element and B is the element below A.
    - c: Push the result of evaluation on the stack
      [END OF IF]
- Step 4: Set result = stack's top elements
- Step 5: Exit

Prefix Evaluation

- Step 1: Get Prefix expression as it is
- Step 2: Repeat until all the characters in prefix
  expression have been scanned
    - a: Read the prefix expression from right to left one at a time
    - b: If the readed character is an operand, push it on the stack
    - c: If the readed character is an operator, then
        - i: pop two values from the stack.
        - ii: Apply the operation on the operands.
        - iii: Push the result onto the stack.
- Step 3: Exit

Exercise

1. Write a program to implement following character stack operations using arrays with MAX elements. Use a Character array. [Try to create generic function so that it can be reused for later programs]

   - PUSH
   - POP
   - PEEP
   - DISPLAY
   - CHANGE
   - IS_FULL
   - IS_EMPTY

2. Add a functionality to your stack program and provide an option for the user to test an application to reverse the string. Here the user will be asked for a string and using stack operations your code should provide a revered string as output. [You have to use code for stack implementation and operation created in earlier program]
3. Add a functionality to your stack program and provide an option for the user to test an application to convert an infix expression to a postfix expression. Here the user will be asked for an infix expression as a string and using stack operations your code should provide converted postfix expression as output. [You have to use code for stack implementation and operation created in earlier program]
4. Add a functionality to your stack program and provide an option for the user to test an application to evaluate a postfix expression. Here the user will be asked for a postfix expression as a string and using stack operations your code should provide evaluated output. [You have to use code for stack implementation and operation created in earlier programs. Consider all the operands will be single digit in expression]

--

- Write a program to implement the following stack operations. The implementation should allow users to create dynamic sized stacks.

  - PUSH
  - POP
  - PEEP
  - DISPLAY
  - CHANGE
  - IS_FULL
  - IS_EMPTY

- Add a functionality to your stack program for infix to prefix conversion of an expression. [You can use functions for stack implementation and operation created in earlier program]
- Add a functionality to your stack program for infix to postfix conversion of an expression. [You can use functions for stack implementation and operation created in earlier program]
- Add a functionality to your stack program for evaluation of postfix expression. [You can use functions for stack implementation and operation created in earlier programs. Consider all the operands will be single digit in expression]
- Add a functionality to your stack program for evaluation of prefix expressions. [You can use functions for stack implementation and operation created in earlier programs. Consider all the operands will be single digit in expression]