05

Stack Data Structure

DATA STRUCTURE LAB

SESSION - 05

CSE135 | DAFFODIL INTERNATIONAL UNIVERSITY

Stack

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple linear data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

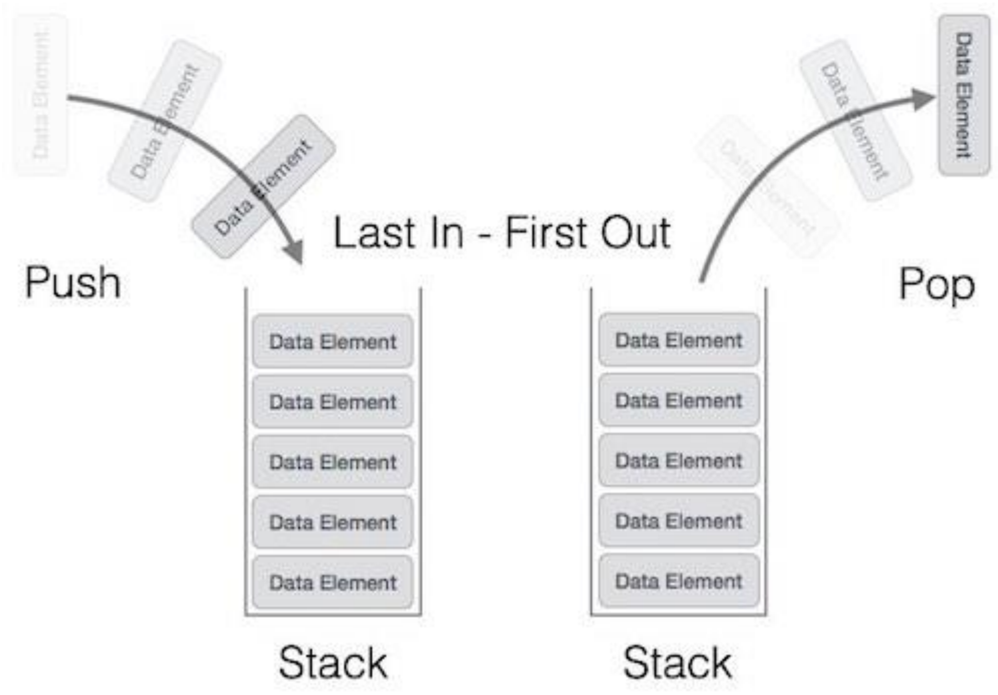
A real-world stack allows operations at one end only. Firstly take real time example, Suppose we have created stack of the book like this shown in the following fig –

How books are arranged in that stack???

1. Books are kept one above the other
2. Book which is inserted first is Taken out at last.(Brown)
3. Book which is inserted Lastly is served first.(Light Green)

Stack is first in last out (LIFO) last in first out(LIFO) Data Structure. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

The following diagram depicts a stack and its operations −



Basic Operations:

A stack is used for the following two primary operations −

- **push()** − Pushing (storing/inserting) an element on the stack.
- **pop()** − Removing (accessing/removing) an element from the stack.

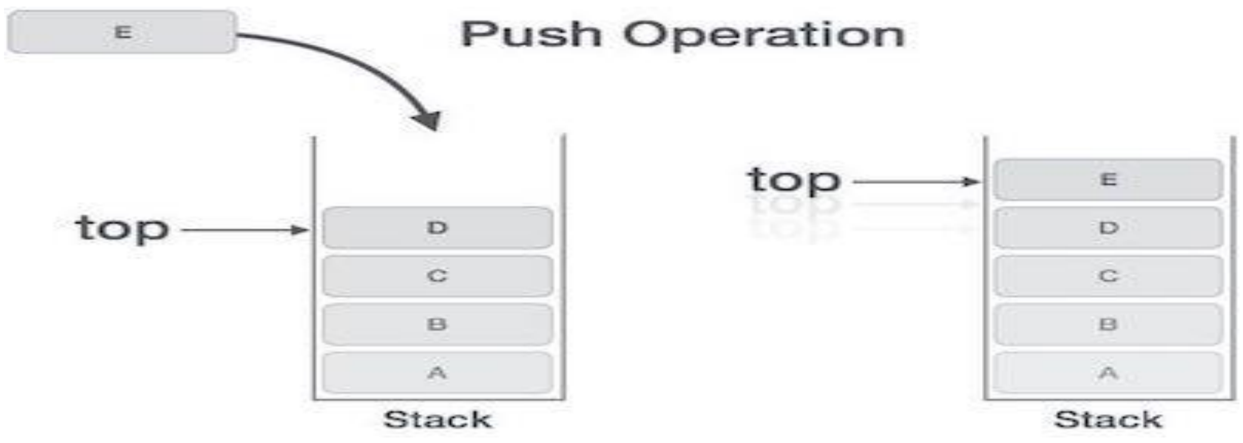
To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.(if top is equal to maximum size then stack is full)
- **isEmpty()** − check if stack is empty.(if top is less than 0 then stack is empty)

Push Operation

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

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



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

PUSH Operation implementation in array

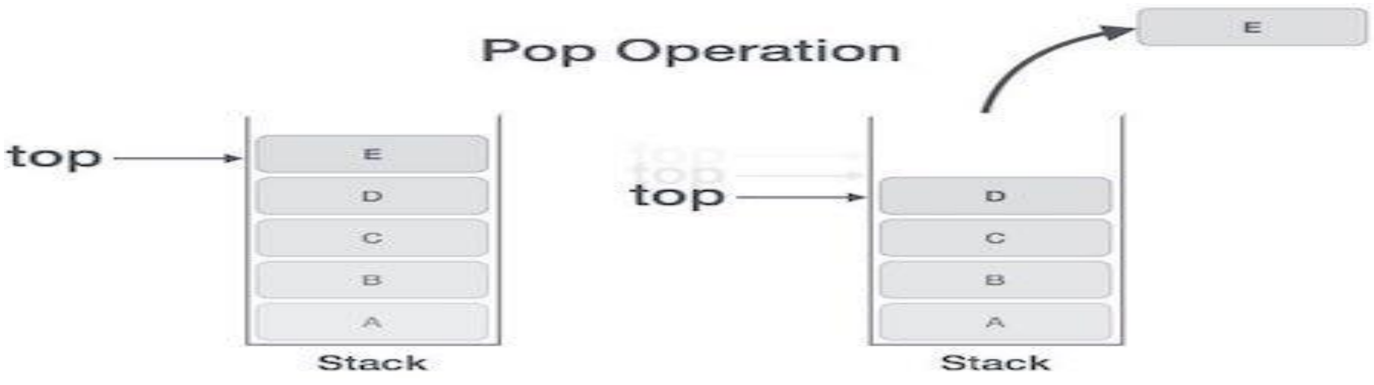
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   }
   Else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

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

A Pop operation may involve the following steps −

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



Pop Operation implementation in array

```
int pop(int data)
{
    if(top<0){
        printf("stack is empty!\n");
        return -1;
    }
    top--;
    return stack[top]
}
```

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Implementing Stack using Arrays

```
#include<stdio.h>
#define max 10
int stack[max];
/*
```


```
   0      1       2      3      4       5       6     7       8       9
*/
int top=-1;
void push(int data)
{
    if(top==max)
    {
        printf("Stack is full\n");
        return;
    }
    top++;
    stack [top]=data;
    printf("%d is pushed\n",data);
}

int pop()
{
    if(top<0)
    {
        printf("Stack is empty\n");
        return -1;
    }
    return stack [top--];
}
int main()
{
    push(7);
/*
```

7					

```
0(top)   1       2      3       4       5       6     7      8       9
*/
     push(9);
/*
```

7	9				

```
0       1(top)   2       3      4       5      6      7      8      9
*/
     push(11);
/*
```

7	9	11			

```
0        1     2(top)    3      4       5     6       7      8       9
*/
     printf("%d\n",pop());
/*
```

7	9				

```
0      1(top)   2      3       4       5       6      7       8       9
*/
```

```
    printf("%d\n",pop());
/*
```

7					

```
0(top)    1      2      3      4      5      6     7      8      9
*/
    printf("%d\n",pop());
/*
```


```
0         1       2      3      4      5      6      7      8      9
top becomes -1 now */
    printf("%d\n",pop());
/*stack is empty now so it will return -1 and print stack is empty*/
     return 0;
}
```

Implementation of stack using linklist:

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node node;
void push(node *stack,int data)
```

```
{
     node *temp;
      temp=(node*)malloc(sizeof(node));
/*     temp
        |
        |
    +---+-----+
    | #  | #  |
    +---+-----+
*/
    temp->data=data;
    temp->next=NULL;
/*     temp
        |
        |
    +-----+-----+
    | data |   ●----->Ø
    +-----+-----+
*/
     while(stack->next !=NULL)
     {
          stack=stack->next;
     }
     stack->next=temp;
/*     stack              temp
        |                  |
        |                  |
    +-----+-----+     +-----+-----+
    |  #  |   ●-----> | data|   ●-----> Ø
    +-----+-----+     +-----+-----+
*/
    printf("%d is pushed!\n",data);
}
int pop(node *stack)
{
    int data;
    node *temp;
    if(stack->next==NULL)
    {
        printf("Stack is empty\n");
        return -1;
    }
    while(stack->next->next!=NULL)
    {
        stack=stack->next;
    }
    temp=stack->next;
    data=temp->data;
    stack->next=NULL;
    free(temp);
    return data;
}
int main()
{
    node *stack;
    stack=(node*)malloc(sizeof(node));
    stack->next=NULL;
```

```
/*     stack(top)
         |
         |
    +-----+-----+
    |  #  |   ●----->Ø
    +-----+-----+
*/
```

push(stack,9);

```
/*     stack            stack->next(top)
         |                 |
         |                 |
    +-----+-----+     +-----+-----+
    |  #  |   ●-----> |  9  |   ●-----> Ø
    +-----+-----+     +-----+-----+
*/
```

push(stack,11);

```
/*     stack             stack->next(top)
         |                  |
         |                  |
    +-----+-----+     +-----+-----+     +-----+-----+
    |  #  |   ●----→ |  9  |   ●-----→| 11  |   ●---→Ø
    +-----+-----+     +-----+-----+     +-----+-----+
*/
```

push(stack,7);

```
/*  stack      stack->next    stack->next->next  stack->next->next->next(top)
       |             |                 |                |
       |             |                 |                |
 +-----+-----+    +-----+-----+    +-----+-----+     +---+----+
 |   # |   ●---> |  9  |   ●----> | 11  |   ●----> |  7 |   ●--> ∅
 +-----+-----+    +-----+-----+    +-----+-----+     +---+----+
*/
    printf("\n%d is popped\n",pop(stack));
  stack           stack->next       stack->next->next(top)
      |             |                  |
      |             |                  |
 +-----+-----+    +-----+-----+    +-----+-----+
 |   #   |   ●-->|   9 |   ●---> | 11   |   ●--->∅
 +-----+-----+    +-----+-----+    +-----+-----+
    printf("%d is popped\n",pop(stack));
      stack             stack->next(top)
        |                 |
        |                 |
    +-----+-----+     +-----+-----+
    |  #  |   ●-----> |  9  |   ●-----> ∅
    +-----+-----+     +-----+-----+
    printf("%d is popped\n",pop(stack));


   stack(top)
        |
        |
    +-----+-----+
    |   # |   ●----->∅
    +-----+-----+

     return 0;
}
```

Applications of Stack

Expression Parsing (Infix to Postfix, Postfix to Prefix etc) and many more.

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

Expression Parsing

1+2-5 ←-arithmetic expression

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are,

- Infix Notation
- Prefix Notation
- Postfix Notation

Expression	**Example**	**Note**
Infix	a + b	Operator Between Operands
Prefix	+ a b	Operator before Operands
Postfix	a b +	Operator after Operands

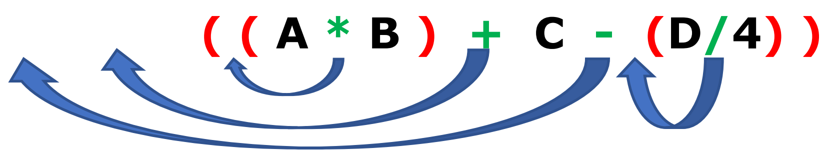
Infix to Prefix:

Infix → A*b+C-D/4

Step1: Parenthesize(put parenthesis) according to precedence of operator.

((A*B)+C-(D/4))

Step2: Move operator to the left out of enclosing parenthesis.



Step3: write the prefix.

+ - * A B C / D 4 ← PREFIX

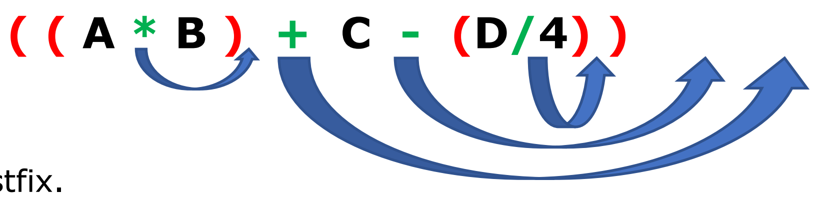
Infix to Postfix:

Infix → A*b+C-D/4

Step1: Parenthesize(put parenthesis) according to precedence of operator.

((A*B)+C-(D/4))

Step2: Move operator to the right out of enclosing parenthesis.



Step3: write the postfix.

AB*CD4/-+ ← POSTFIX

The following table briefly tries to show the difference in all three notations –

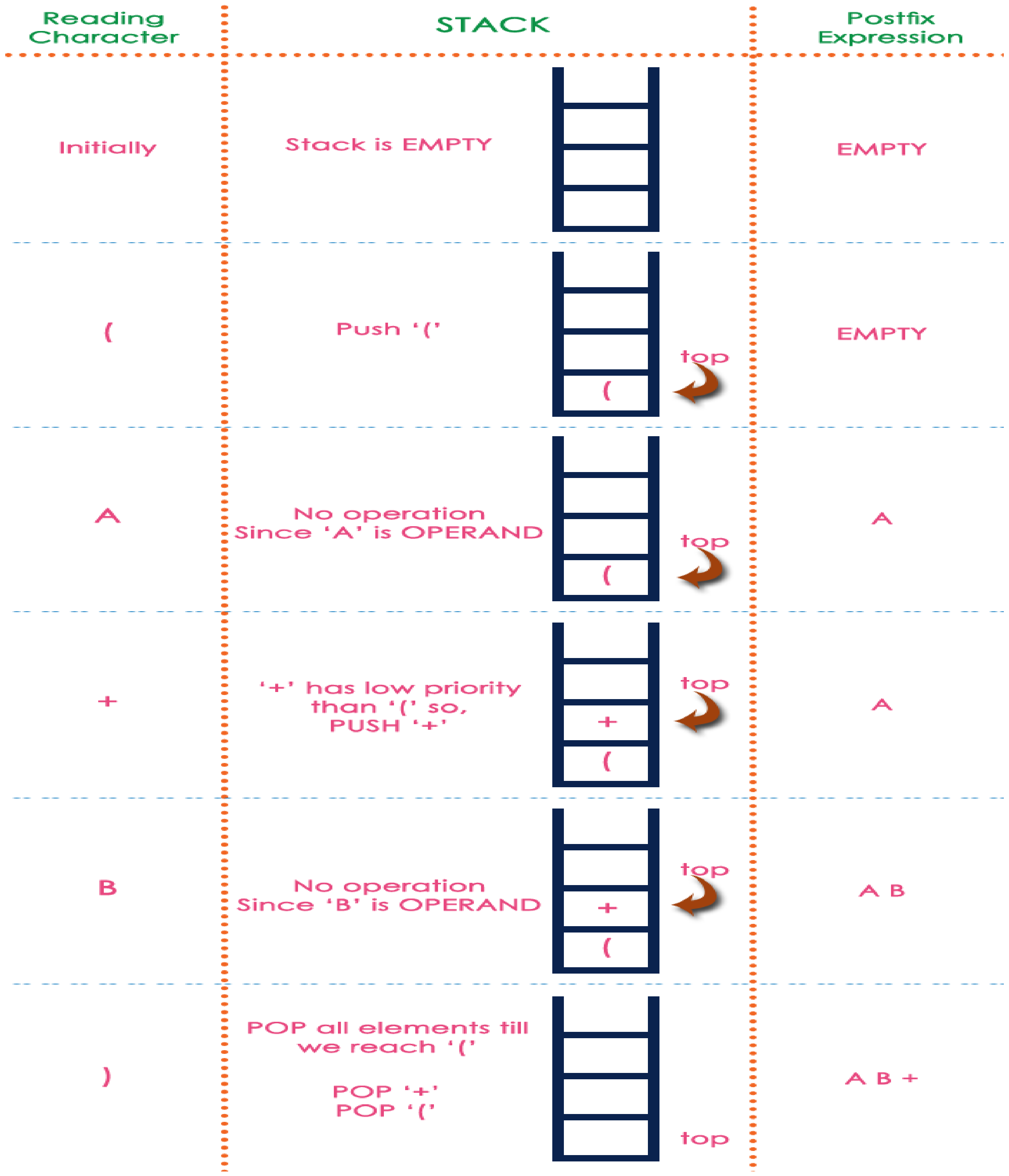
Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

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

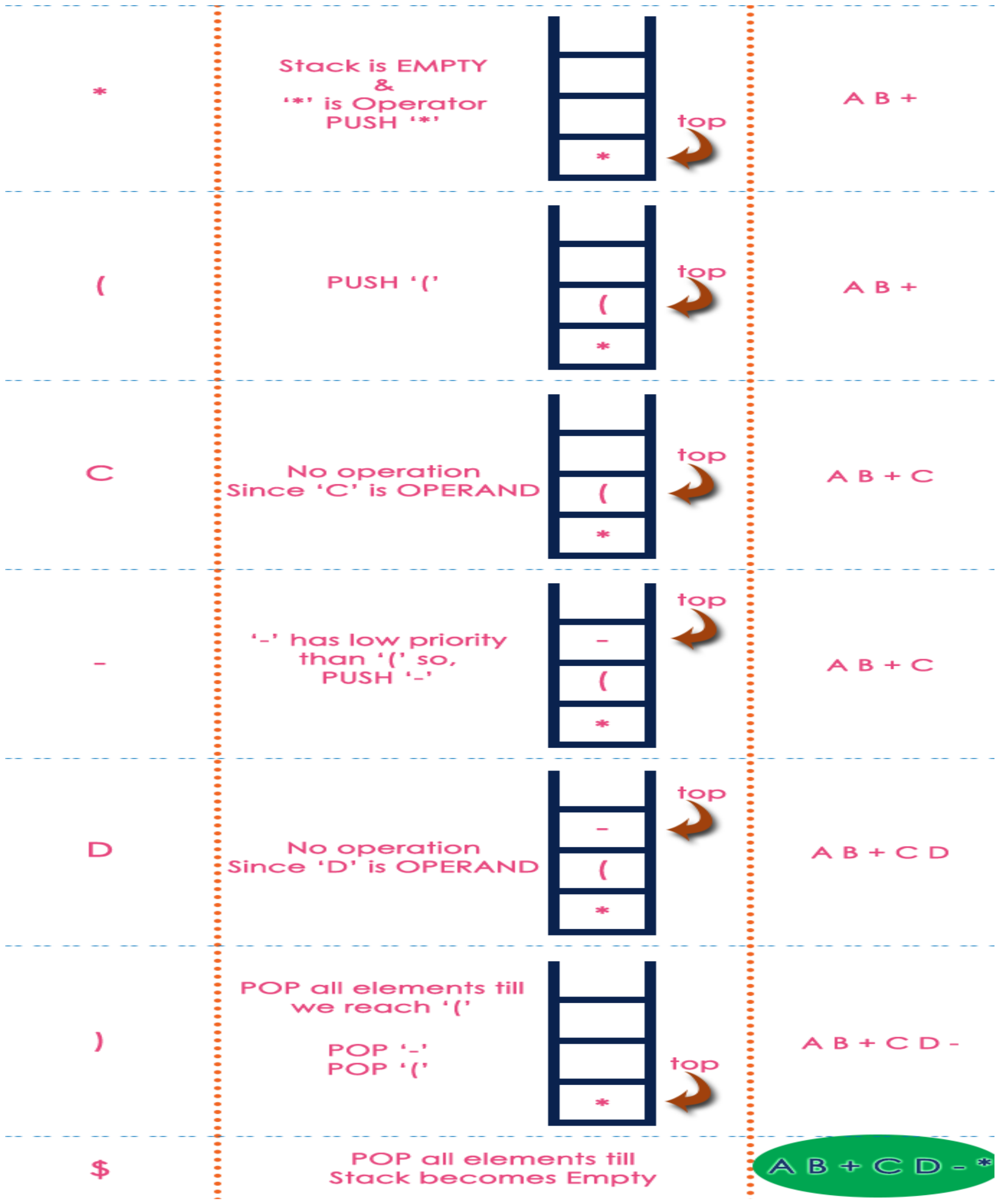
In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c** .

Infix to postfix conversion using Stack:

```
Considering the following Infix Expression     (A+B)*(C-D)
```



Reading Character	STACK		Postfix Expression
Initially	Stack is EMPTY		EMPTY
(	Push '('	(top	EMPTY
A	No operation Since 'A' is OPERAND	(top	A
+	'+' has low priority than '(' so, PUSH '+'	(+ top	A
B	No operation Since 'B' is OPERAND	(+ top	A B
)	POP all elements till we reach '(' POP '+' POP '('	top	A B +



*	Stack is EMPTY & '*' is Operator PUSH '*'	top *	A B +
(	PUSH '('	top (*	A B +
C	No operation Since 'C' is OPERAND	top (*	A B + C
-	'-' has low priority than '(' so, PUSH '-'	top - (*	A B + C
D	No operation Since 'D' is OPERAND	top - (*	A B + C D
)	POP all elements till we reach '(' POP '-' POP '('	top *	A B + C D -
$	POP all elements till Stack becomes Empty		A B + C D - *

The final postfix expression is as follows:

```
A B + C D - *
```

Infix to postfix conversion implementation:

```
#include<stdio.h>
#include<string.h>
#define max 100
char stack[max];
int top=-1;
void push(char k){
    if(top+1==max)
        printf("Stack is full!");
    else{
        top++;
        stack[top]=k;
    }
}
char pop(){
    char a;
    if(top<0)
    {
        printf("Stack is empty\n");
        return '\0';
    }
    else
    {
        a=stack[top];
        top--;
        return a;
    }
}
int main()
```

```
{
    int i,l;
    char ch,e[max];
    printf("Enter string with():");
    gets(e);
    l=strlen(e);
    printf("Postfix is:\n");
    for(i=0; i<=l-1; i++){
        ch=e[i];
        if(ch=='('||ch=='*'||ch=='-'||ch=='+'||ch=='/')
            push(ch);
        else if(ch!=')')
            putchar(ch);
        else{
            do{
                ch=pop();
                if(ch!='(')
                    putchar(ch);
            }
        while(ch!='(');
        }
    }
    return 0;
}
```

Stack Evaluation:

```
            (5+3)*(8-2)
            Value: 48
```

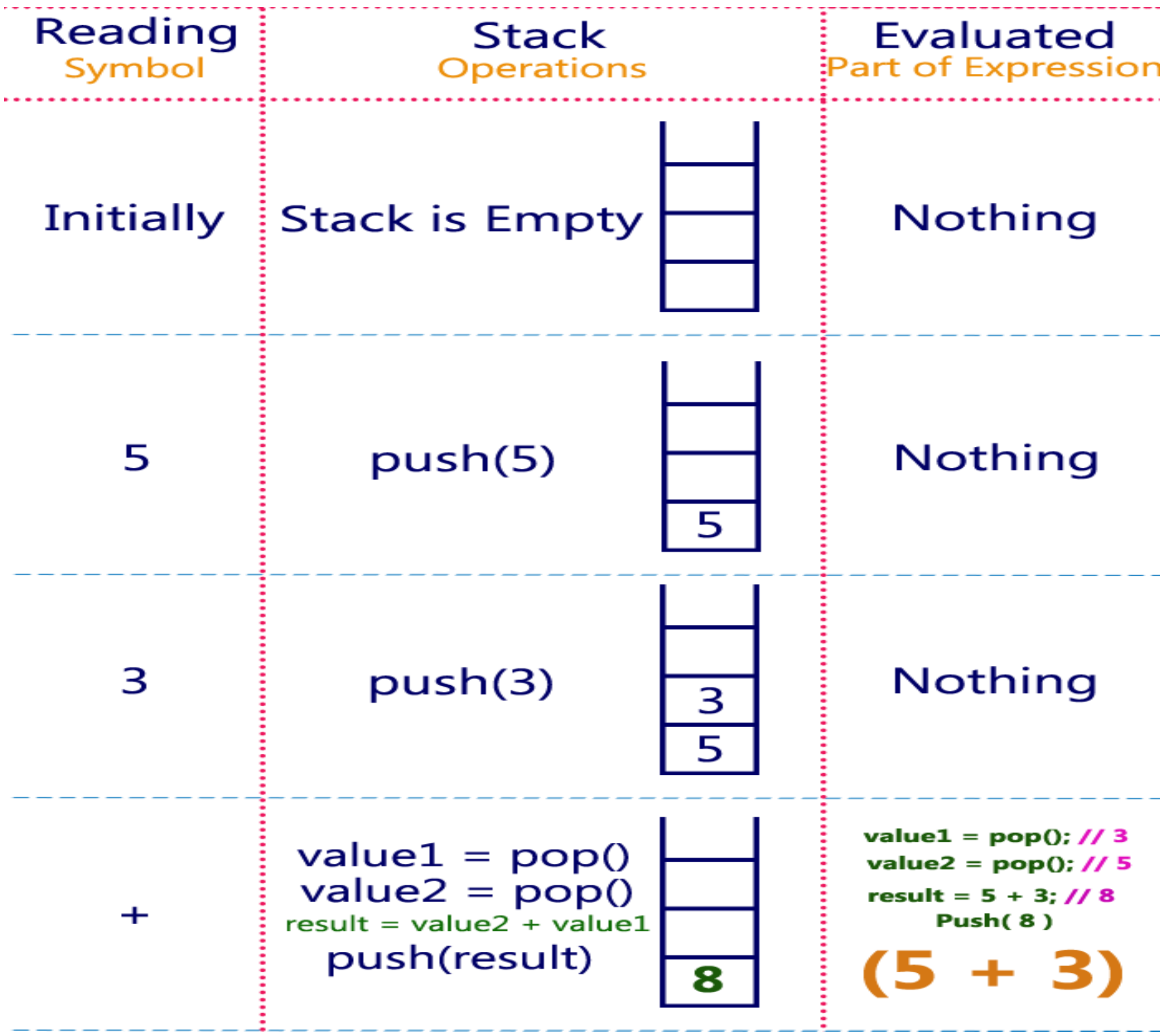
Now lets evaluate given expression using stack.

Step 1.Postfix → **43/232*-+**

Step 2. Evaluation→

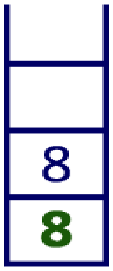
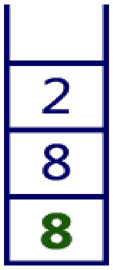
Rules:

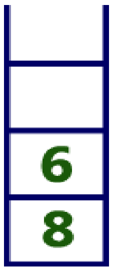
- If operand then push into stack.
- If operator (^, +, -, *, /) then pop two operands and do the operation and push into stack.

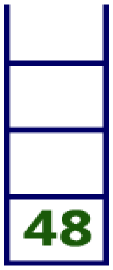


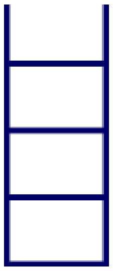
Reading Symbol	Stack Operations		Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)		(5 + 3)
2	push(2)		(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
			value1 = pop(); // 6
			value2 = pop(); // 8
			result = 8 * 6; // 48
			Push(48)
			(6 * 8)
			(5 + 3) * (8 - 2)

$ End of Expression	result = pop()		Display (result) **48** As final result

Infix Expression **(5 + 3) * (8 - 2) = 48**

Postfix Expression **5 3 + 8 2 - *** value is **48**

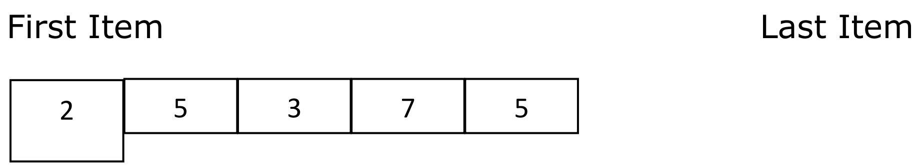
Infix to postfix conversion implementation:

```
#include<stdio.h>
#include<ctype.h>
int stack[50];
int top=-1;
void push(int data){
    stack[++top]=data;
}
int pop(){
    return (stack[top--]);
}
nt main()
{
    char post[50],ch;
    int i=0, op1,op2;
    printf("Enter postfix:");
    scanf(" %s",post);
    while((ch=post[i++])!='\0'){
        if(isdigit(ch)){
            push(ch-'0');
        }
        else{
            op2=pop();
            op1=pop();
            printf("%d %d\n",op1,op2);
            switch(ch){
                   case'+':push(op1+op2);
                break;
                case'-':push(op1-op2);
                break;
                case'*':push(op1*op2);
                break;
                case'/':push(op1/op2);
                break;
            }
        }
    }
    printf("result=%d\n",stack[top]);
    return 0;
}
```

##Exercise:

STACK:

1. Implement c code for the following diagram using Stack data structure using array:



2. Show the result evaluating the following postfix expression using stack.

“10 5 + 60 6 / * 8 – “

3.convert to postfix and prefix.

a. 6/3+4-2*3
b. ((8/4)+6-(3*3))

4. Use stack to convert the following into postfix

a. 4*2+6/3-5*2
b. 3^2+9/3-4*2

5. Evaluate the following arithmetic operation using stack.

a. 4/2+3-2*2
b. 2*3-5+3-9/3

6. Write a C program to create a **Stack data structure**. This Stack data structure is to store the integer values. Your program should display a menu of choices to operate the Stack data structure. The menu given below.

1. Add items

2. Delete items

3. Show the number of items

4. Show min and max items

5. Find an item

6. Print all items

7. count how many items are there

8. Exit