Vivekananda Institute
of Professional Studies

Stacks

Data Structures
Dr. Nivedita Palia

STACKS

- **Stack** is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows **adding and removing elements** in a particular order.

- A stack is an **abstract data type** that holds an ordered, linear sequence of items. A stack is a **last in, first out** (LIFO) structure.

- Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack.

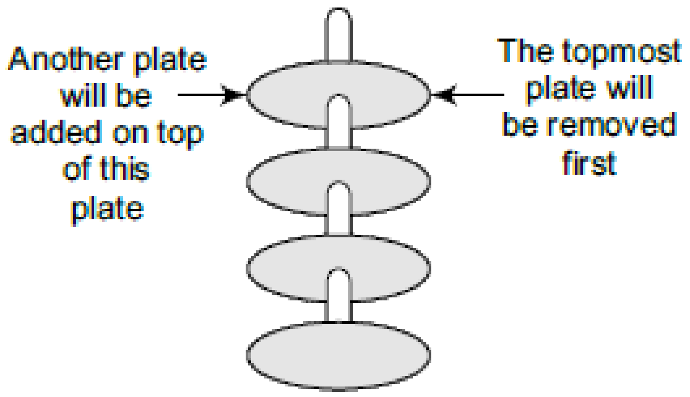
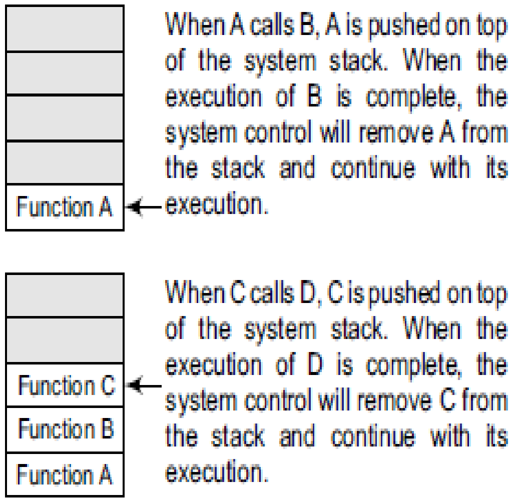

Another plate will be added on top of this plate
The topmost plate will be removed first

Figure 7.1 Stack of plates

Need of Stack


Function A
When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.
Function C
Function B
Function A
When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.

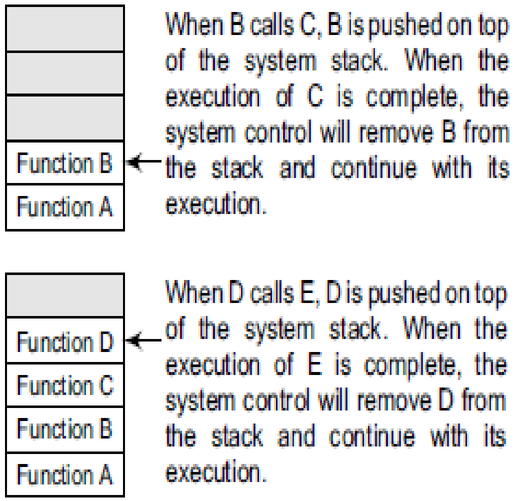

Function B
Function A
When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.
Function D
Function C
Function B
Function A
When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.

Figure 7.2 System stack in the case of function calls

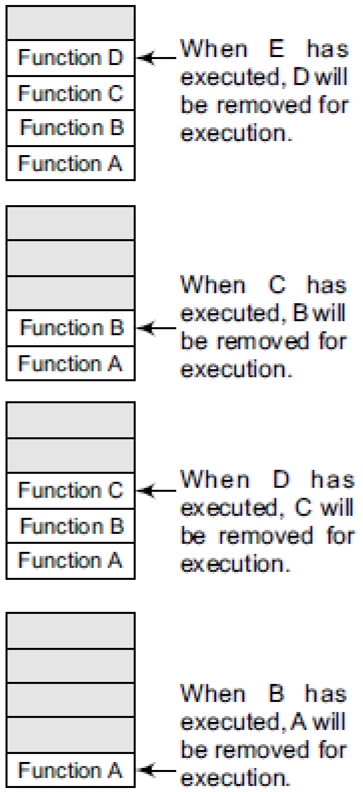
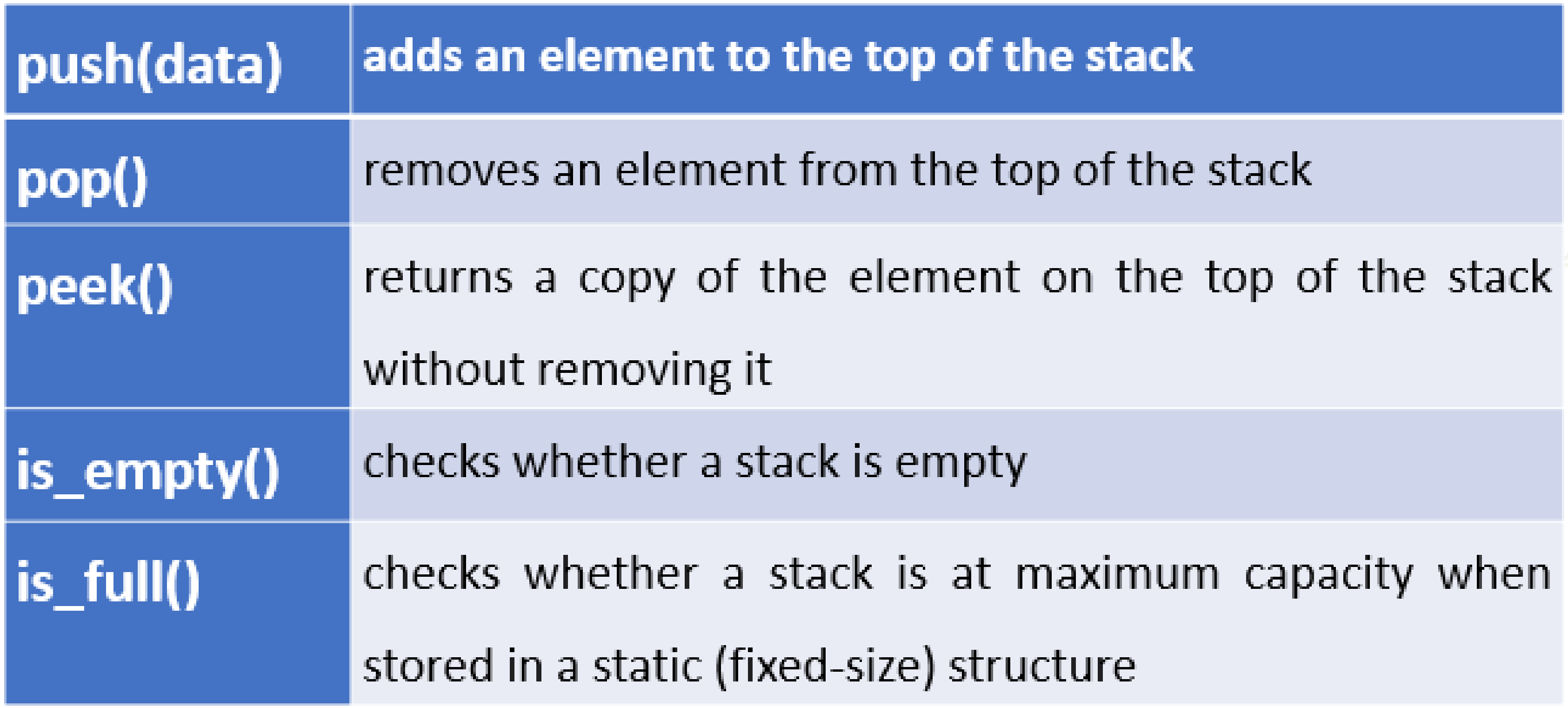

Function D
Function C
Function B
Function A
When E has executed, D will be removed for execution.
Function B
Function A
When C has executed, B will be removed for execution.
Function C
Function B
Function A
When D has executed, C will be removed for execution.
Function A
When B has executed, A will be removed for execution.

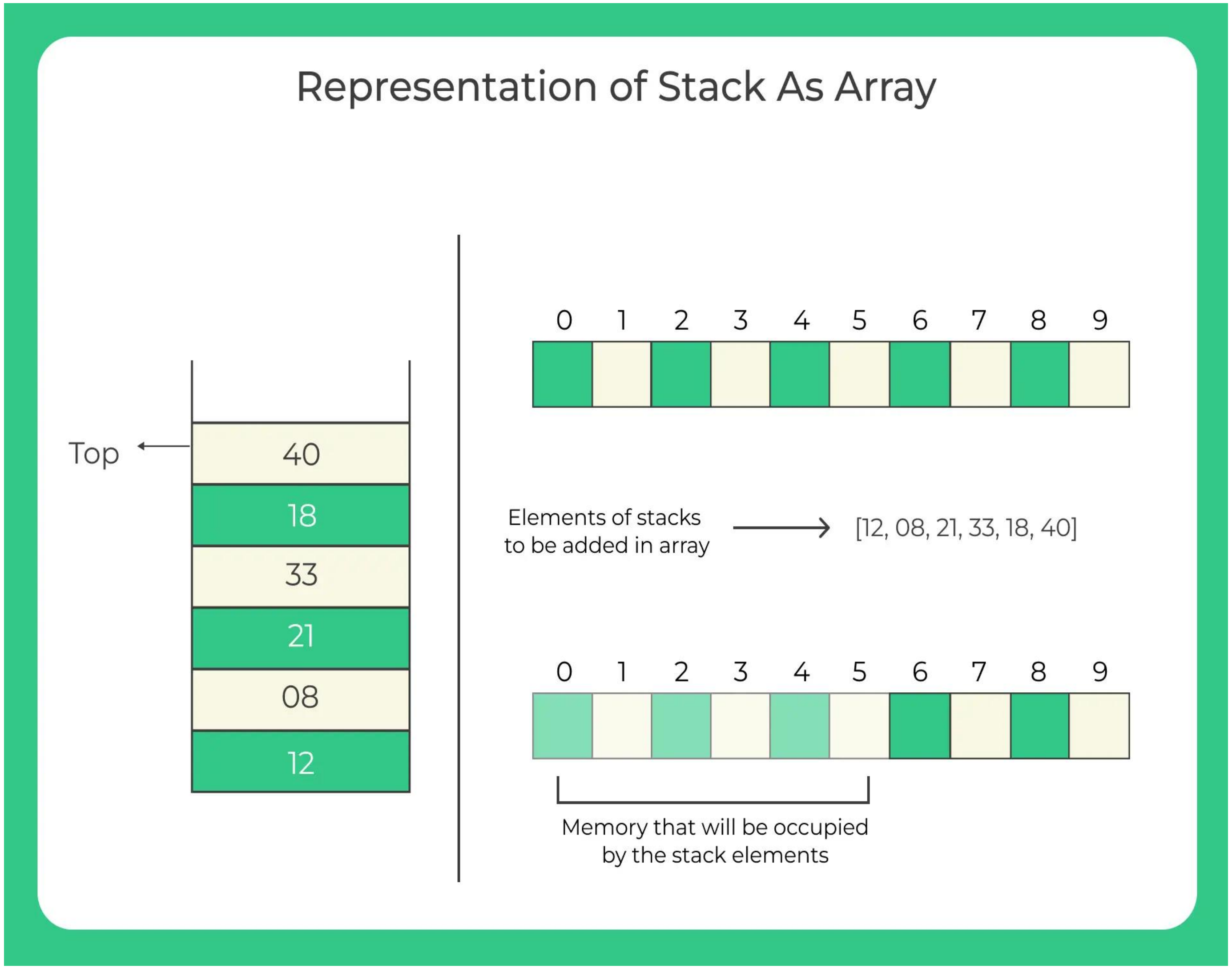
Figure 7.3 System stack when a called function returns to the calling function

Operations on stack



push(data)	**adds an element to the top of the stack**
pop()	removes an element from the top of the stack
peek()	returns a copy of the element on the top of the stack without removing it
is_empty()	checks whether a stack is empty
is_full()	checks whether a stack is at maximum capacity when stored in a static (fixed-size) structure

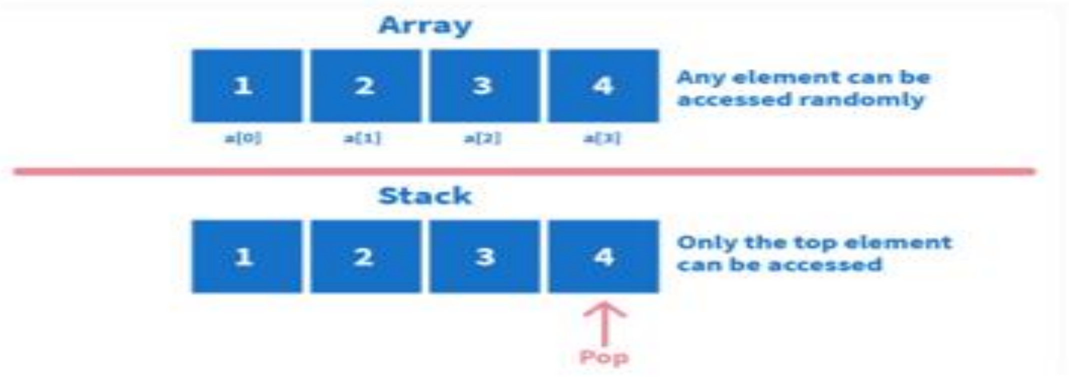
Implementation of Stack


Representation of Stack As Array
Top
40
18
33
21
08
12
0 1 2 3 4 5 6 7 8 9
Elements of stacks to be added in array
[12, 08, 21, 33, 18, 40]
0 1 2 3 4 5 6 7 8 9
Memory that will be occupied by the stack elements

Stack VS Array

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.


Array
1
2
3
4
a[0]
a[1]
a[2]
a[3]
Any element can be accessed randomly
Stack
1
2
3
4
Only the top element can be accessed
Pop

Vivekananda Institute of Professional Studies

Operation on Stack using Arrays

Push

```
void push(int st[], int val)
{
         if(top == MAX-1)
         {
                  printf("\n STACK OVERFLOW");
         }
         else
         {
                  top++;
                  st[top] = val;
         }
}
```

```
Step 1: IF TOP = MAX-1
              PRINT "OVERFLOW"
              Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

1	2	3	4	5	6				
0	1	2	3	4	TOP = 5	6	7	8	9

Pop

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

1	2	3	4						
0	1	2	TOP = 3	4	5	6	7	8	9

```
int pop(int st[])
{
         int val;
         if(top == -1)
         {
                  printf("\n STACK UNDERFLOW");
                  return -1;
         }
         else
         {
                  val = st[top];
                  top--;
                  return val;
         }
}
```

```
Step 1: IF TOP = NULL
              PRINT "UNDERFLOW"
              Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Peek

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

```
int peek(int st[])
{
         if(top == -1)
         {
                  printf("\n STACK IS EMPTY");
                  return -1;
         }
         else
         return (st[top]);
}
```

```
Step 1: IF TOP = NULL
              PRINT "STACK IS EMPTY"
              Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Traverse

```
void display(int st[])
{
         int i;
         if(top == -1)
         printf("\n STACK IS EMPTY");
         else
         {
                  for(i=top;i>=0;i--)
                  printf("\n %d",st[i]);
                  printf("\n"); // Added for formatting purposes
         }
}
```

Push O(1), Pop O(1),Peek O(1) and Traverse O(n)

Stack implementation using Linked List

Start as a Top of Stack

Push- Insertion at the beginning of the linked list

Pop- Deletion at the beginning of the linked list

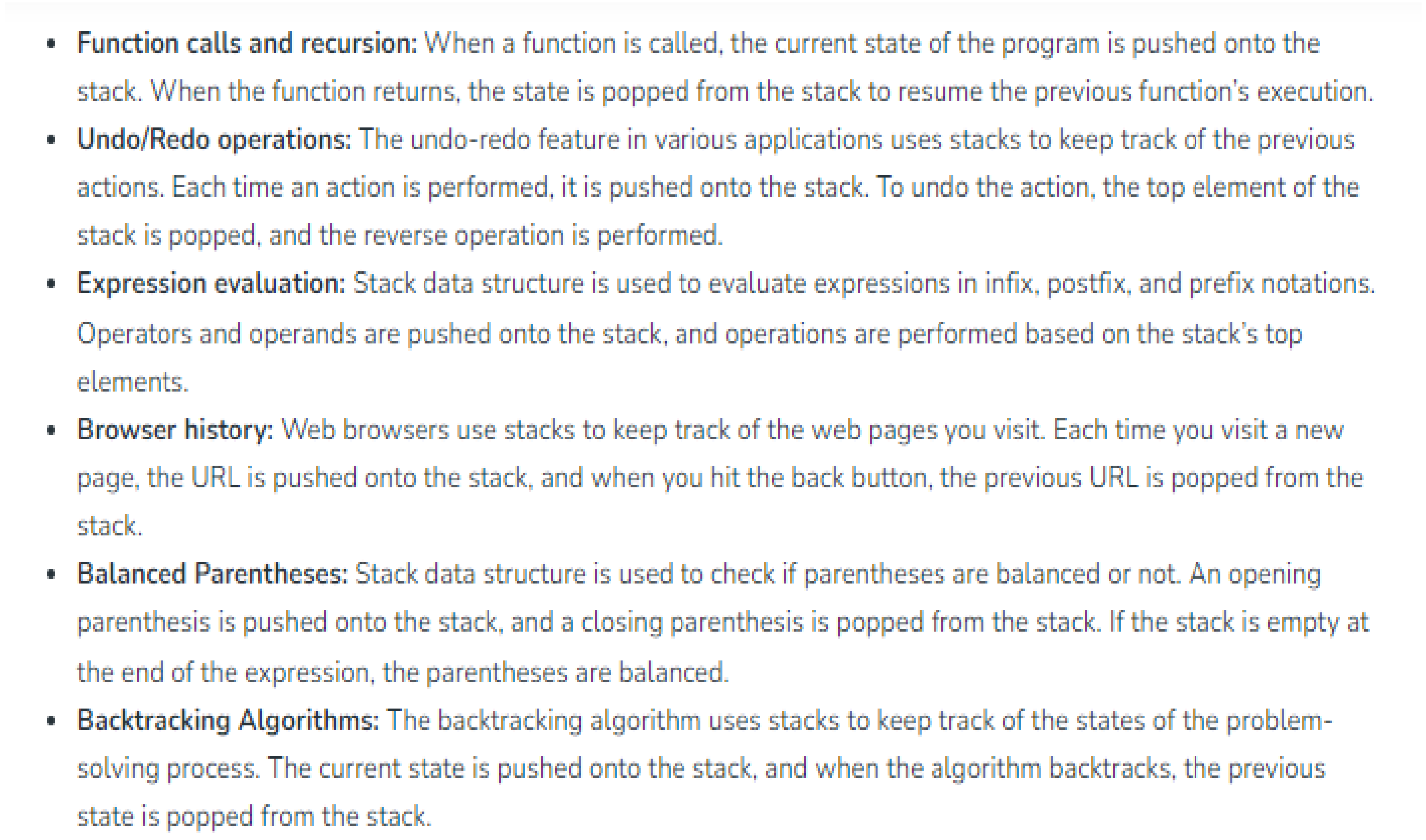
peek- Print the value of the first node

Overflow: when no free space

Underflow: when start is null

Vivekananda Institute of Professional Studies

Applications



- **Function calls and recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.
- **Undo/Redo operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.
- **Expression evaluation:** Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.
- **Browser history:** Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.
- **Balanced Parentheses:** Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.
- **Backtracking Algorithms:** The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.

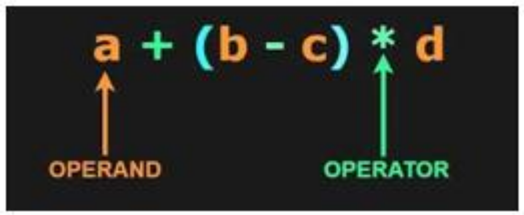
Vivekananda Institute of Professional Studies

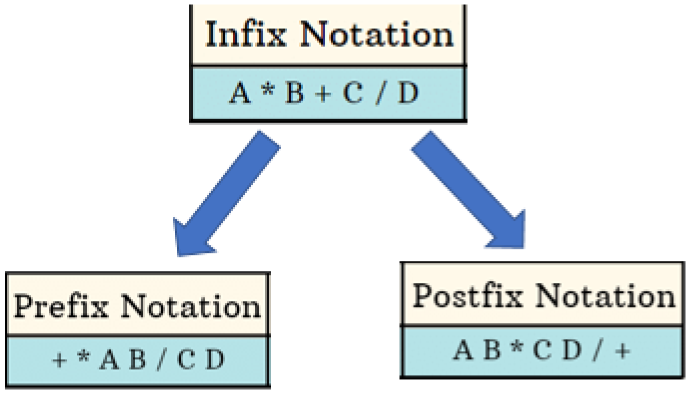
Arithmetic Expression

Infix notation : <operands><operator><operands>

Postfix notation (Reverse Polish notation): <operands> <operands> <operator>

Prefix notation (Polish notation): <operator><operands><operands>





Binary operations have different levels of precedence and association .

• First :	Exponentiation (^)
• Second:	Multiplication (*) and Division (/)
• Third :	Addition (+) and Subtraction (-)

Vivekananda Institute of Professional Studies

योगः कर्मसु कौशलम्

IN PURSUIT OF PERFECTION

Evaluation of Arithmetic Expression

- Evaluate the following Arithmetic Expression:

  5 ^ 2 + 3 * 5 – 6 * 2 / 3 + 24 / 3 + 3
- First:

  25 + 3 * 5 – 6 * 2 / 3 + 24 / 3 + 3
- Second:

  25 + 15 – 4 + 8 + 3
- Third:

  47

Vivekananda Institute of Professional Studies

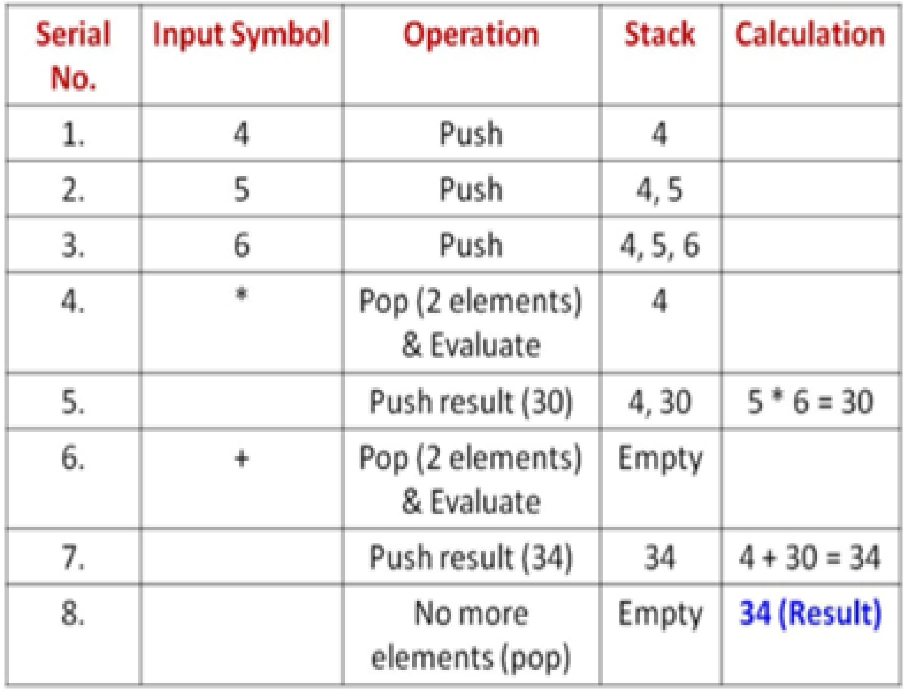
Evaluation of Postfix Expression

❑ P is an arithmetic expression in Postfix Notation.

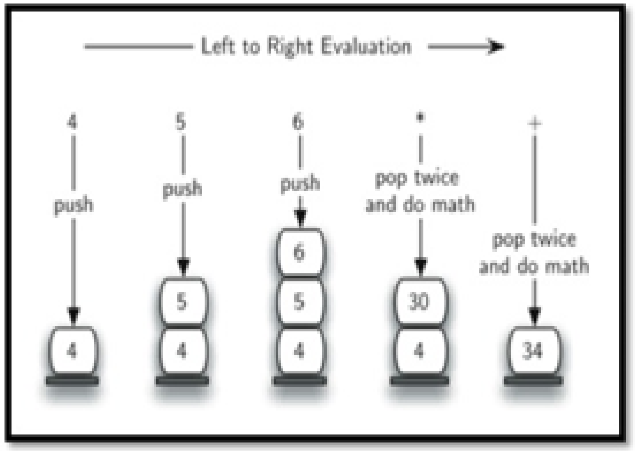
```
1.  Add a right parenthesis ")" at the end of P.

2.  Scan P from left to right and Repeat Step 3 and 4 for each element of
    P until the sentinel ")" is encountered.

3.          If an operand is encountered, put it on STACK.
4.          If an operator @ is encountered, then:
             (a) Remove the two top elements of STACK, where A
                 is the top element and B is the next to top element.
             (b) Evaluate B @ A.
             (c) Place the result of (B) back on STACK.
            [End of if structure.]
    [End of step 2 Loop.]
5.  Set VALUE equal to the top element on STACK.
6.  Exit.
```

Evaluation of Postfix Expression

Example: 456*+



Serial No.	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.		Push result (30)	4, 30	5 * 6 = 30
6.	+	Pop (2 elements) & Evaluate	Empty	
7.		Push result (34)	34	4 + 30 = 34
8.		No more elements (pop)	Empty	34 (Result)

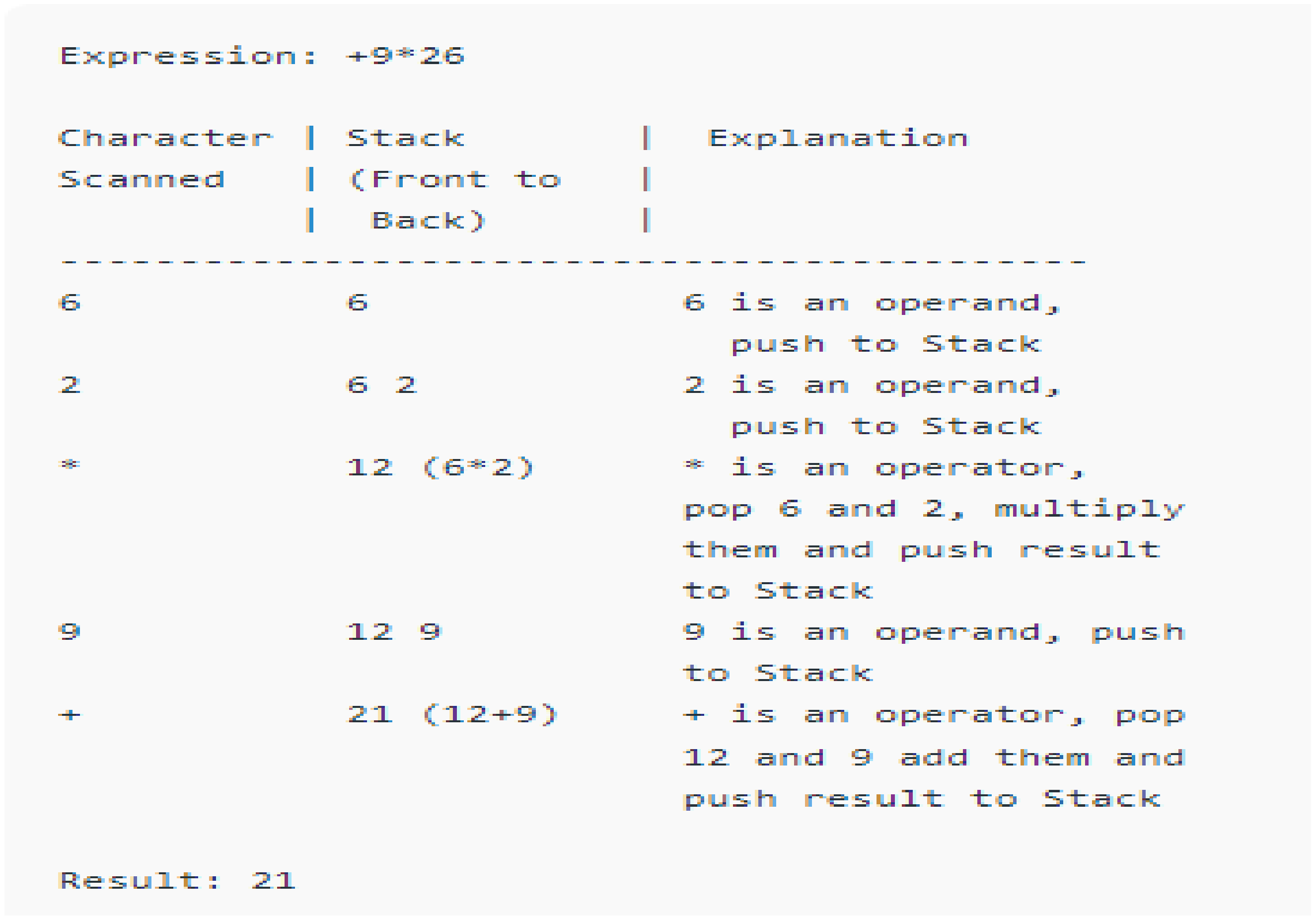


2 3 4 + 5 6 - - *

Evaluation of Prefix Expression

Step 1: Start from the last element of the expression.
Step 2: check the current element.
Step2.1: if it is an operand, push it to the stack.
Step 2.2: If it is an operator, pop two operands from the stack. A is top element and B is next to top element. Perform the operation (A op B) and push the result back to the stack.
Step 3: Do this till all the elements of the expression are traversed and return the top of stack which will be the result of the operation.

Evaluation of Prefix Expression



```
Expression: +9*26

Character | Stack          |  Explanation
Scanned   | (Front to      |
          |  Back)         |
--------------------------------------------
6           6                6 is an operand,
                               push to Stack
2           6 2              2 is an operand,
                               push to Stack
*           12 (6*2)         * is an operator,
                             pop 6 and 2, multiply
                             them and push result
                             to Stack
9           12 9             9 is an operand, push
                             to Stack
+           21 (12+9)        + is an operator, pop
                             12 and 9 add them and
                             push result to Stack

Result: 21
```

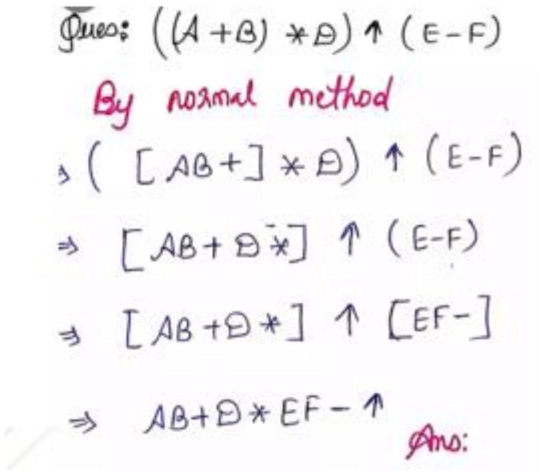
```
Input : -+7*45+20
```

Prefix Expression :

* + 6 9 - 3 1

Infix to Postfix Conversion

Method 1:



Ques: ((A + B) * D) ↑ (E − F)

By normal method

⇒ ([AB+] * D) ↑ (E − F)

⇒ [AB+D*] ↑ (E − F)

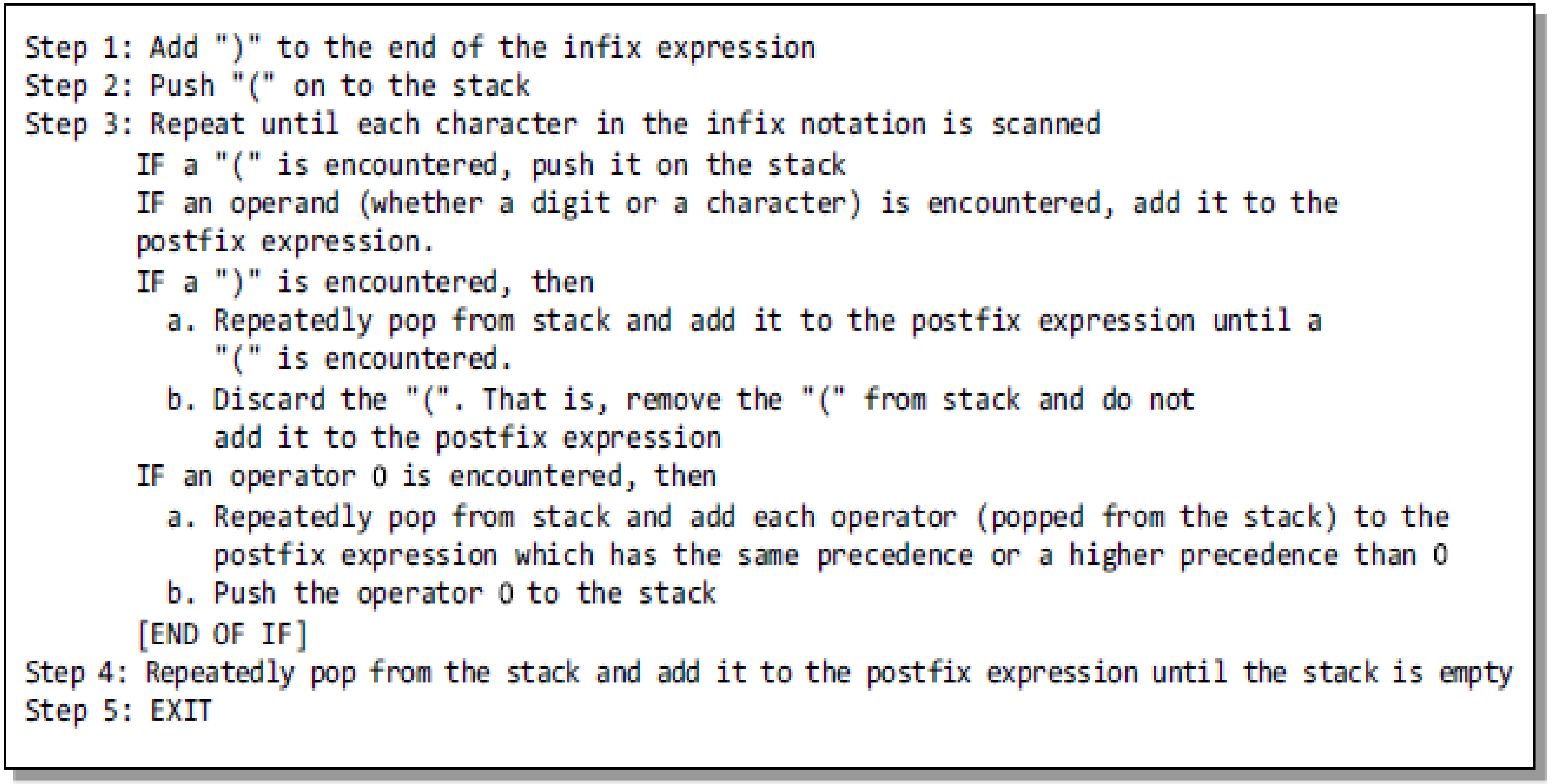
⇒ [AB+D*] ↑ [EF−]

⇒ AB+D*EF−↑ Ans:

- A + B * C
- → (A + (B * C))
- → (A + (B C *))
- → A B C * +

- A + B * C + D
- → ((A + (B * C)) + D)
- → ((A + (B C*))+ D)
- → ((A B C *+) + D)
- → A B C * + D +

Infix to Postfix Conversion using Stack



```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
        IF a "(" is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the
        postfix expression.
        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
        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 the same precedence or a 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
```

Infix to Postfix

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

(a) A – (B / C + (D % E * F) / G)* H

(b) A – (B / C + (D % E * F) / G)* H)

Infix to Prefix

1. Push) on to stack and add (to the end of expression
2. Scan infix expression from Right to left and repeat steps 3 to 6 for each character of expression until stack is empty.
3.If the current character is an operand ,append it to the output string.
4.If the current character is a closing bracket ‘)’, push it onto the stack.
5.If the current character is an operator then
A) Repeatedly pop from stack and add to the expression , each operator which has higher precedence than the current operator.
B) Push current operator to the stack.
6)If the current character is an opening bracket ‘(‘, pop the operators from the stack and append them to the output string until the corresponding closing bracket ‘)’ is encountered. Discard the closing bracket.

7)Reverse the output string to obtain the prefix expression.

Infix to Prefix Conversion

Infix Expression: (P + (Q * R) / (S - T))

Note: - Read the infix string in revers

Symbol Scanned	Stack	Output
)	)	-
)	))	-
T	))	T
-	))-	T
S	))-	ST
(	)	-ST
/	)/	-ST
)	)/)	-ST
R	)/)	R-ST
*	)/)*	R-ST
Q	)/)*	QR-ST
(	)/	*QR-ST
+	)+	/*QR-ST
P	)+	P/*QR-ST
(	Empty	+P/*QR-ST

Prefix Expression: +P/*QR-ST

Vivekananda Institute of Professional Studies

Multiple Stacks

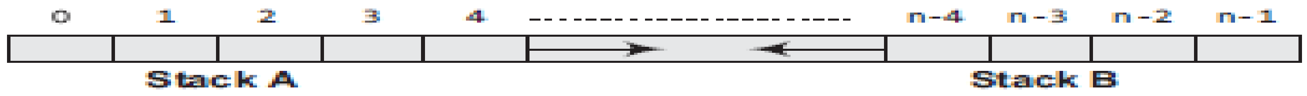
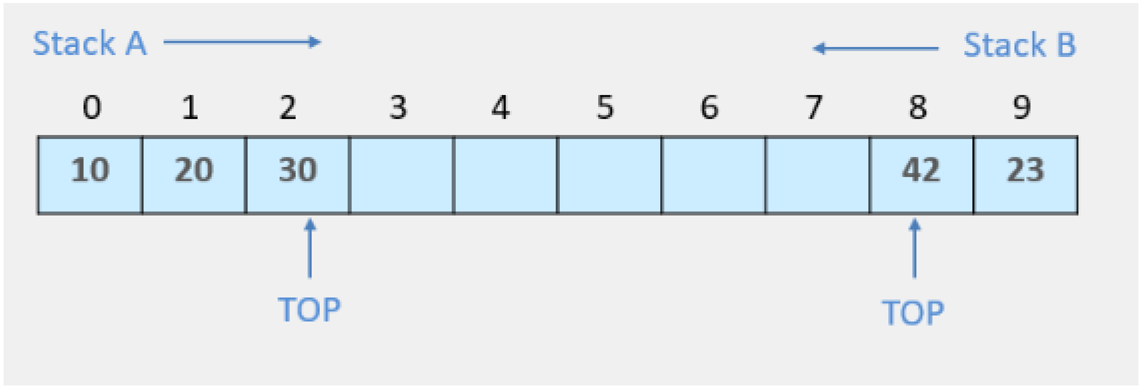


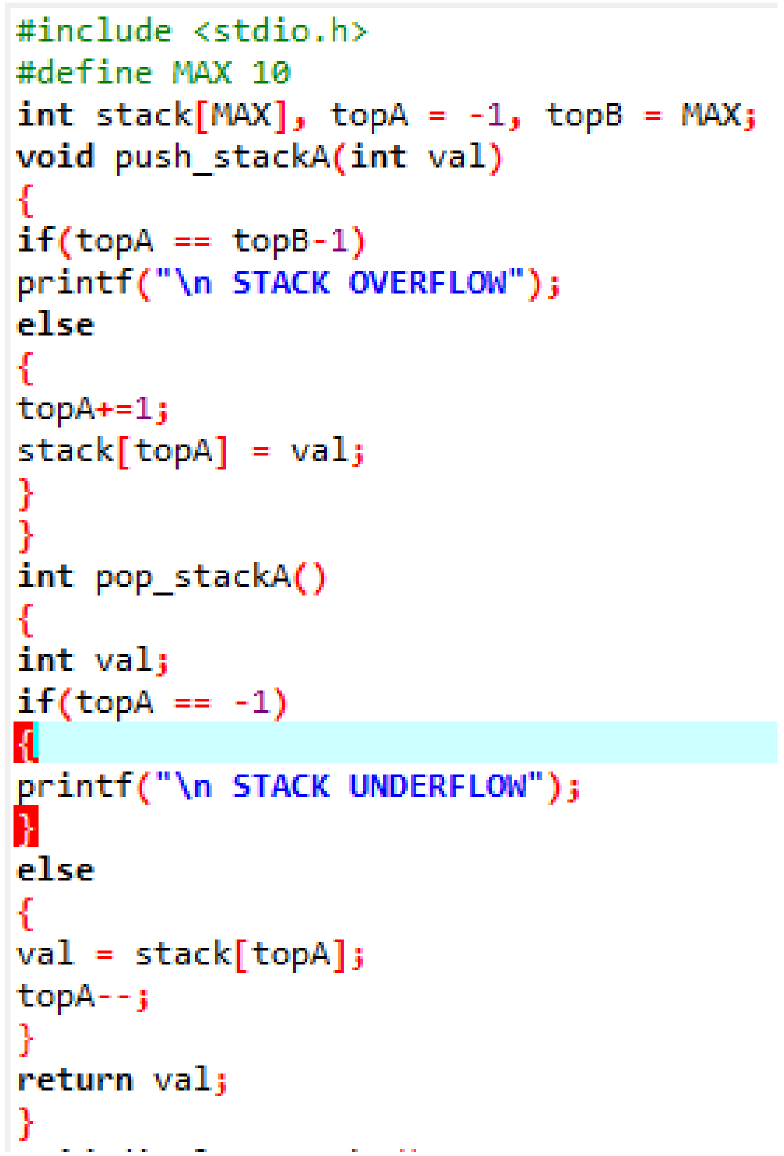
Figure 7.20 Multiple stacks

A single stack is sometimes not sufficient to store a large amount of data. To overcome this problem, we can use multiple stack. For this, we have used a single array having more than one stack. The array is divided for multiple stacks.

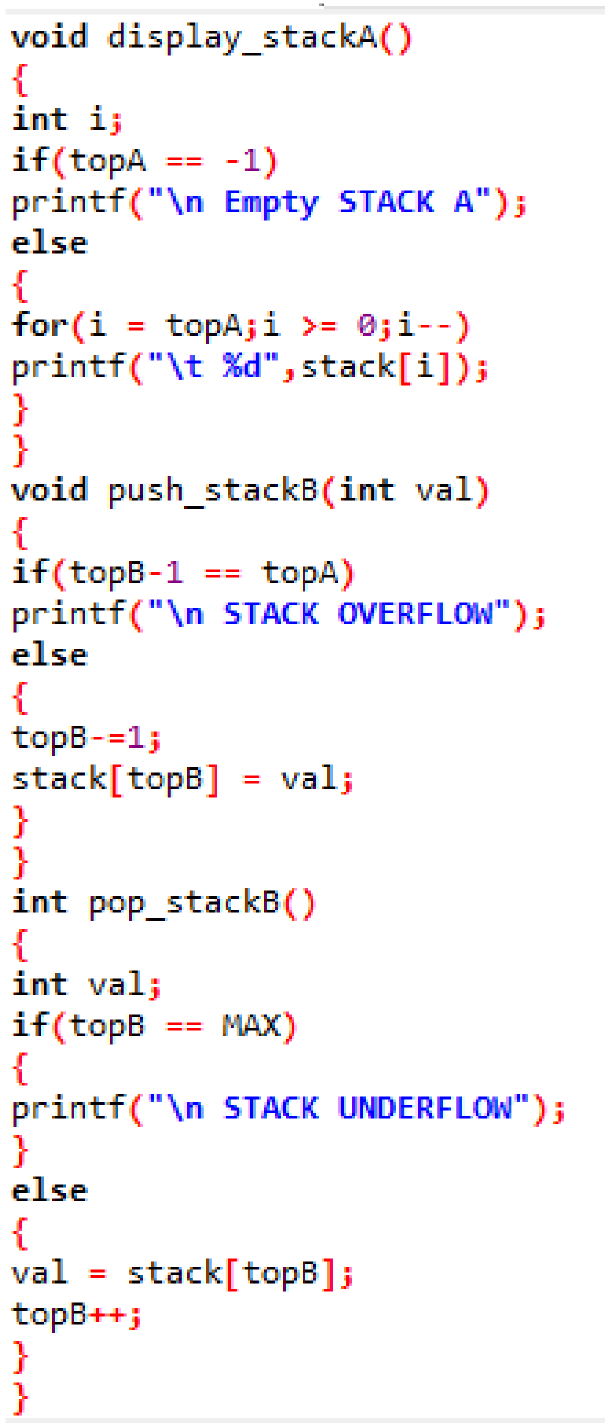
Suppose there is an array **STACK[n]** divided into two stack **STACK** and **STACK B**, where **n = 10**.

- **STACK A** expands from the left to the right, i.e., from 0th element.
- **STACK B** expands from the right to the left, i.e., from 10th element.
- The combined size of both **STACK A** and **STACK B** never exceeds 10.



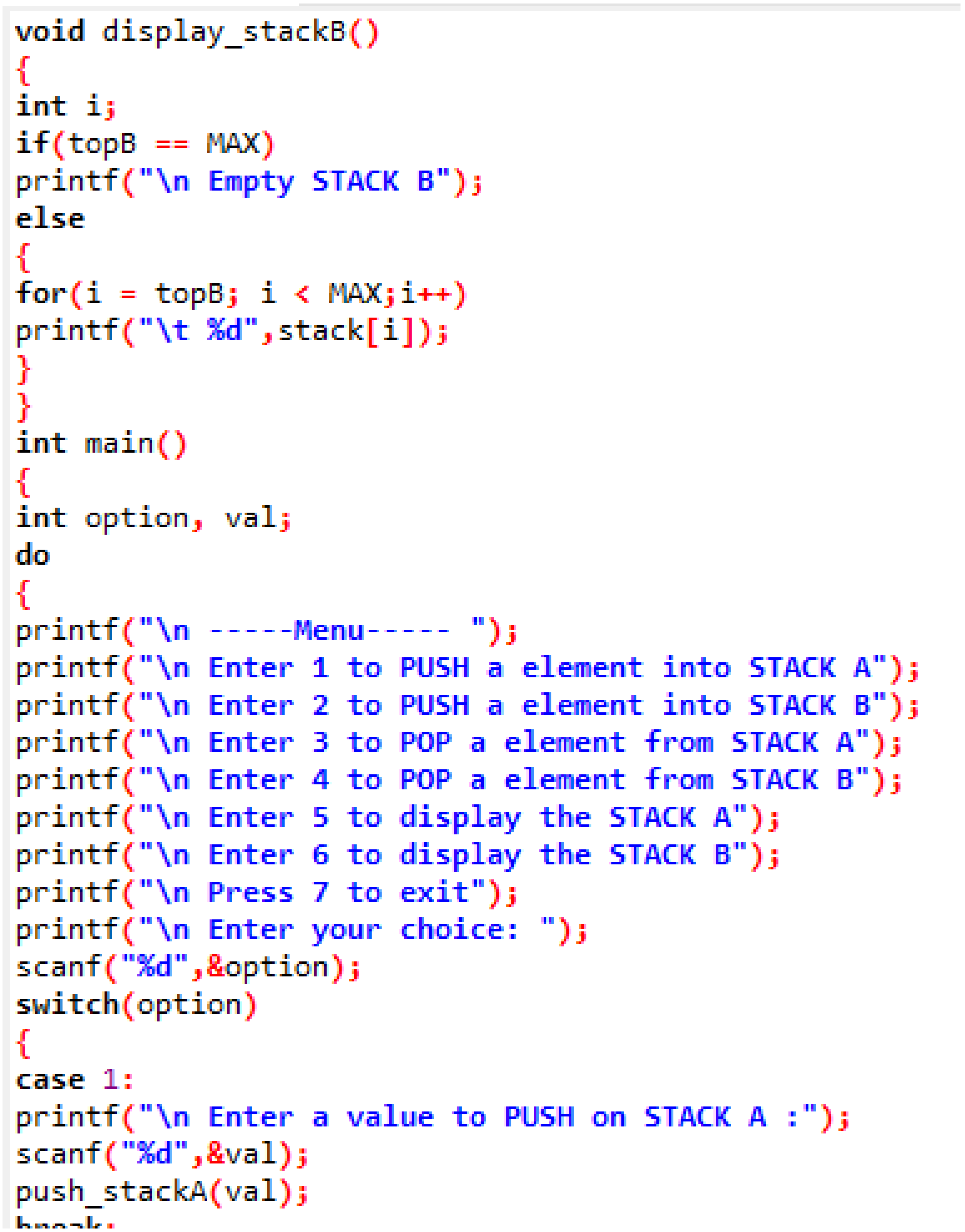


```
#include <stdio.h>
#define MAX 10
int stack[MAX], topA = -1, topB = MAX;
void push_stackA(int val)
{
if(topA == topB-1)
printf("\n STACK OVERFLOW");
else
{
topA+=1;
stack[topA] = val;
}
}
int pop_stackA()
{
int val;
if(topA == -1)
{
printf("\n STACK UNDERFLOW");
}
else
{
val = stack[topA];
topA--;
}
return val;
}
```

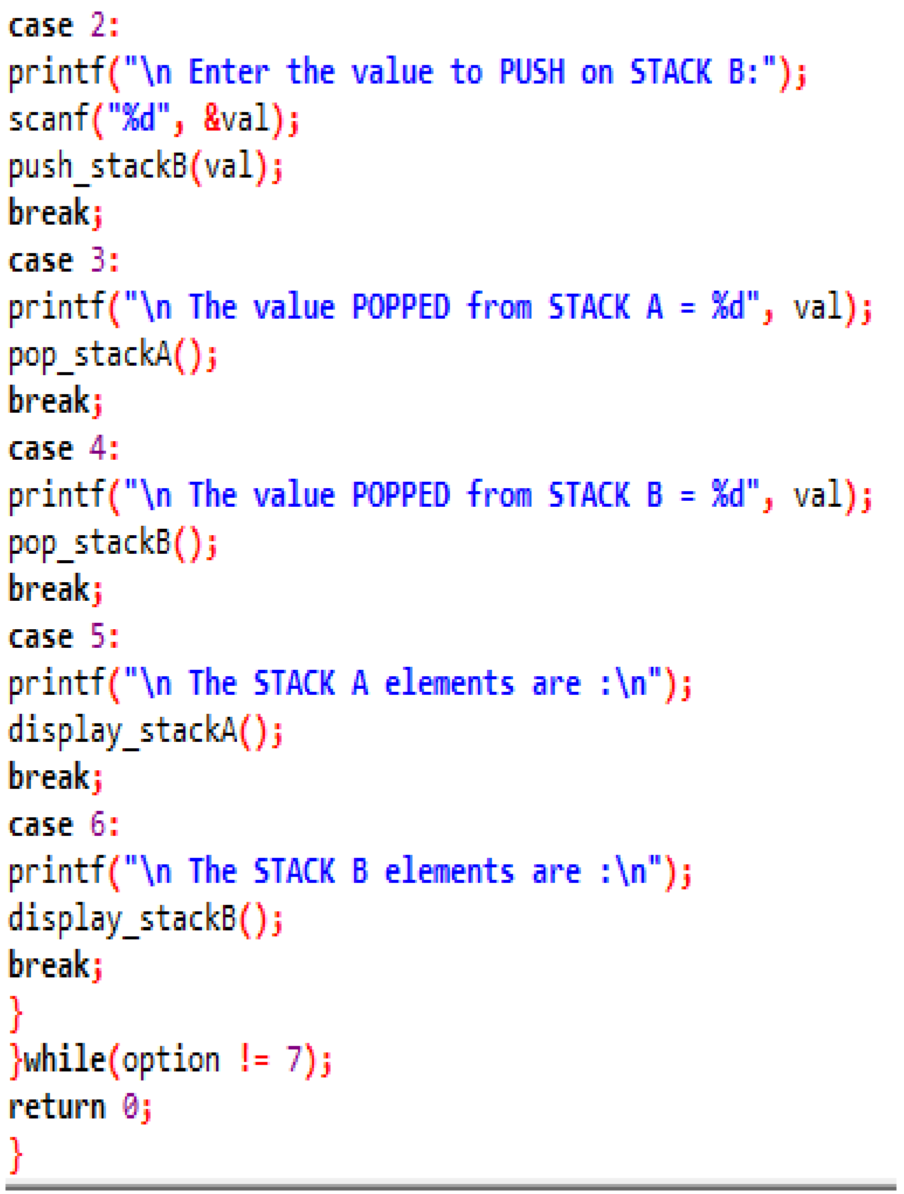


```
void display_stackA()
{
int i;
if(topA == -1)
printf("\n Empty STACK A");
else
{
for(i = topA;i >= 0;i--)
printf("\t %d",stack[i]);
}
}
void push_stackB(int val)
{
if(topB-1 == topA)
printf("\n STACK OVERFLOW");
else
{
topB-=1;
stack[topB] = val;
}
}
int pop_stackB()
{
int val;
if(topB == MAX)
{
printf("\n STACK UNDERFLOW");
}
else
{
val = stack[topB];
topB++;
}
}
```

Vivekananda Institute
of Professional Studies



```
void display_stackB()
{
int i;
if(topB == MAX)
printf("\n Empty STACK B");
else
{
for(i = topB; i < MAX;i++)
printf("\t %d",stack[i]);
}
}
int main()
{
int option, val;
do
{
printf("\n -----Menu----- ");
printf("\n Enter 1 to PUSH a element into STACK A");
printf("\n Enter 2 to PUSH a element into STACK B");
printf("\n Enter 3 to POP a element from STACK A");
printf("\n Enter 4 to POP a element from STACK B");
printf("\n Enter 5 to display the STACK A");
printf("\n Enter 6 to display the STACK B");
printf("\n Press 7 to exit");
printf("\n Enter your choice: ");
scanf("%d",&option);
switch(option)
{
case 1:
printf("\n Enter a value to PUSH on STACK A :");
scanf("%d",&val);
push_stackA(val);
break;
```



```
case 2:
printf("\n Enter the value to PUSH on STACK B:");
scanf("%d", &val);
push_stackB(val);
break;
case 3:
printf("\n The value POPPED from STACK A = %d", val);
pop_stackA();
break;
case 4:
printf("\n The value POPPED from STACK B = %d", val);
pop_stackB();
break;
case 5:
printf("\n The STACK A elements are :\n");
display_stackA();
break;
case 6:
printf("\n The STACK B elements are :\n");
display_stackB();
break;
}
}while(option != 7);
return 0;
}
```

Recursion

A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are

1.Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.

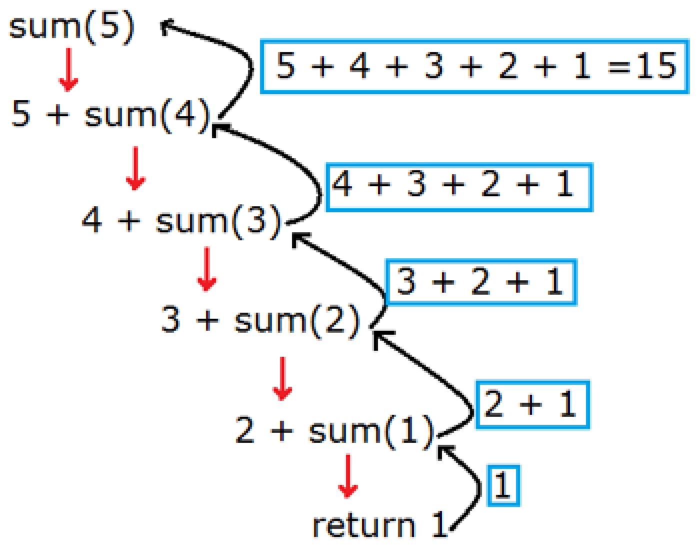
2. Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Every recursive function must have at least one base case. Otherwise, the recursive function will generate an infinite sequence of calls, thereby resulting in an error condition known as an infinite stack.

Sum of natural number using recursion

This exit condition inside a recursive function is known as base condition.

```
int sum(int n)
{
  if(n==1)  //Base Condition
    return 1;
  return n+ sum(n-1); // Function calling itself
}
```

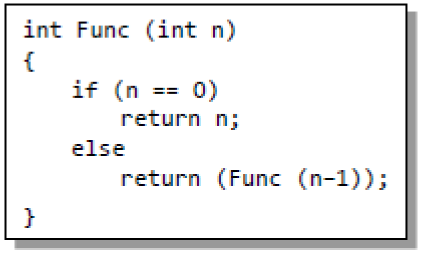


Types of Recursion

- Direct vs Indirect recursive function :

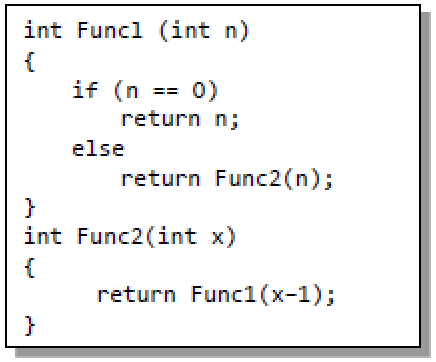
A function is said to be *directly* recursive if it explicitly calls itself.

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it.



```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

Figure 7.28 Direct recursion



```
int Func1 (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
      return Func1(x-1);
}
```

Figure 7.29 Indirect recursion

- Linear Vs Tree recursive function :If a recursive function calling itself for one time then it's known as Linear Recursion. Otherwise if a recursive function calling itself for more than one time then it's known as Tree Recursion. For example :

Linear : factorial Tree: fibnoacci

END