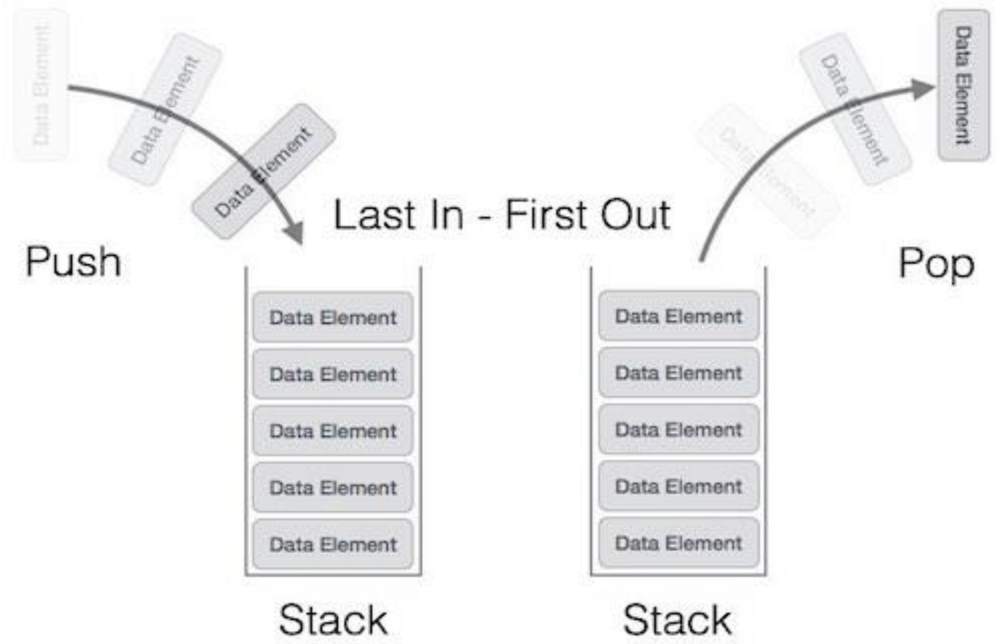
04-09-2023

Stacks

Stacks

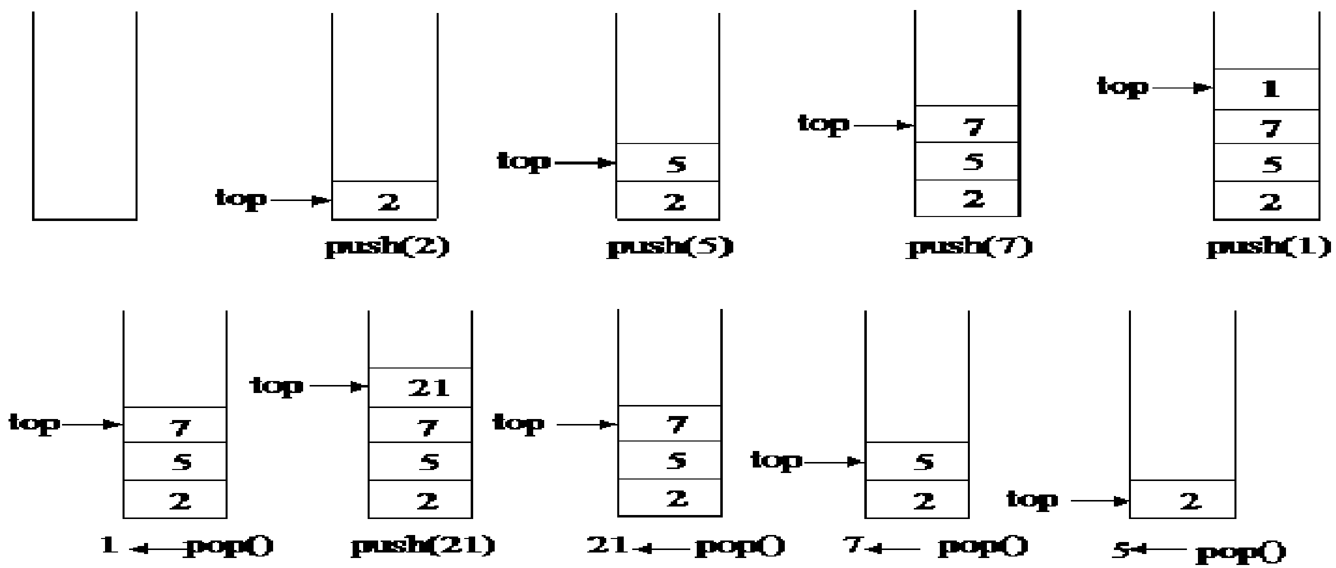
- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).



Operation on Stacks

- **Push operation**-The insertion of an element into stack
- **Pop operation** - Deletion of an element from the stack

- In stack we always keep track of the last element present in the list with a pointer called top.



Array Implementation of Stack

`top` Index of element at the top of stack

Stack is empty → `top` is -1

Push operation

`top` is increased by 1

New element is placed at index `top`

Pop operation

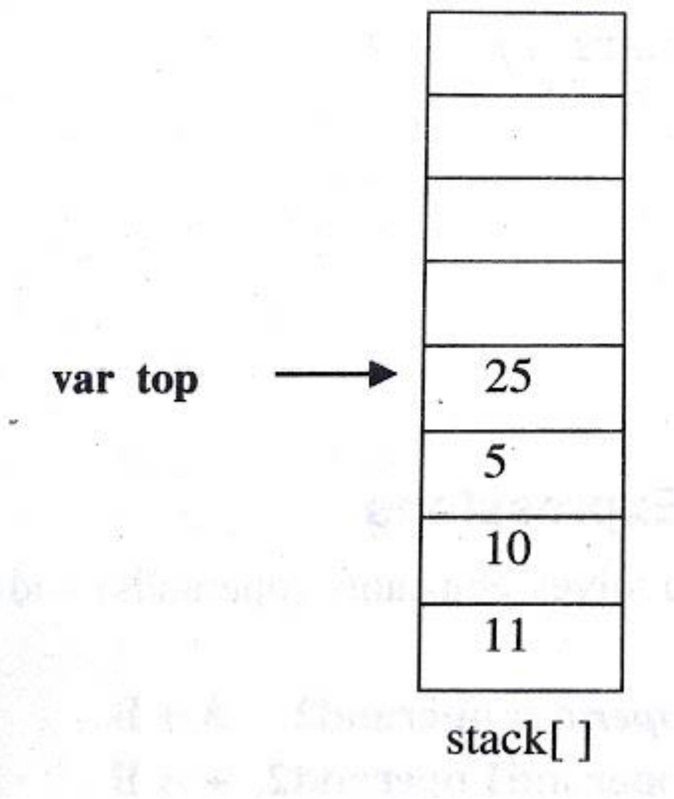
Element at index `top` is taken out

`top` is decreased by 1

Array Implementation of Stack

Array Implementation of Stack

- Push element one by one onto stack from 0th position to n-1 th position.
- Variable top=position of the top element in the array
- If there is no element in the stack, value of top will be -1



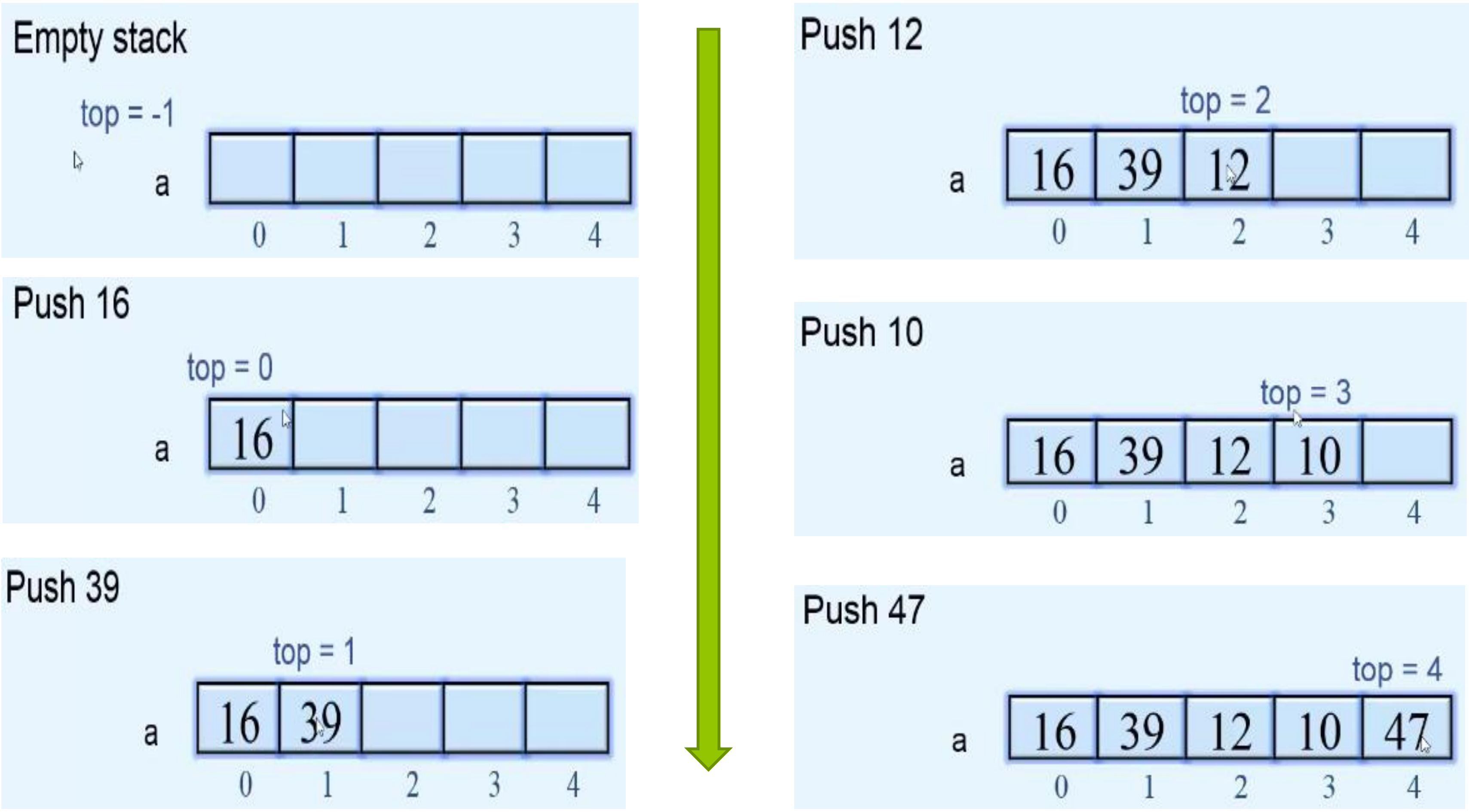
Stack Overflow /FullCondition

If(top==MAX-1)
Then print stack is full or overflowing

Stack Underflow /Empty Condition

If(top== -1)
Then print stack is empty or underflowing

PUSH Operation on Stack



Algorithm for PUSH Operation

1) Check Stack Full condition

   If(top==MAX-1)

   Then print stack is full or overflow

2) Otherwise increase the top value by 1

   top=top+1

3) Input the value

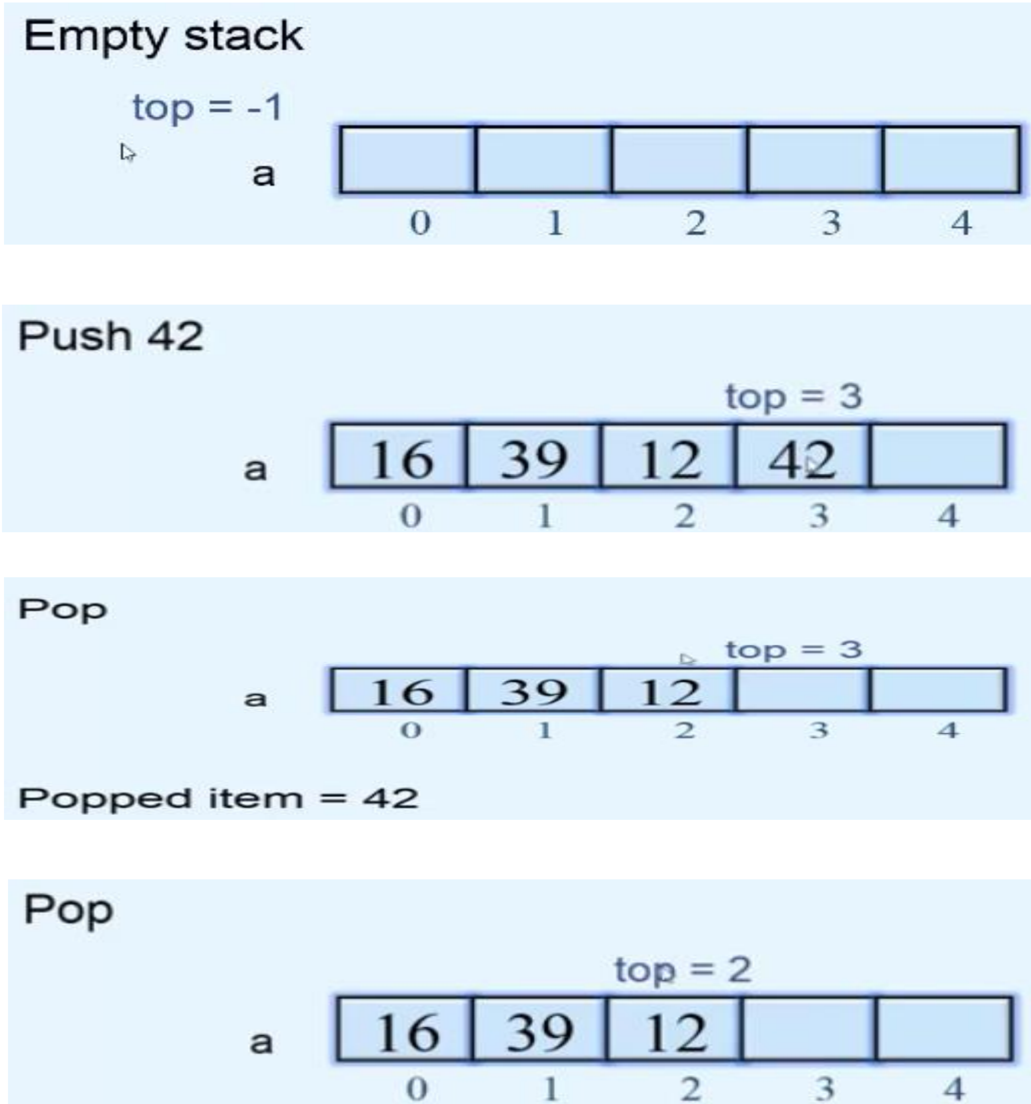
4) Assign the item at top position
   stack_arr[top]=pushed_item

Try writing the code for Push Operation function

PUSH Operation

```
push()
{
        int pushed_item;
        if(top == (MAX-1))
                printf("Stack Overflow\n");
        else
        {
                printf("Enter the item to be pushed in stack : ");
                scanf("%d",&pushed_item);
                top=top+1;
                stack_arr[top] = pushed_item;
        }
}/*End of push()*/
```

POP Operation on Stack



Stack Underflow condition

If(top== -1)

Algorithm for POP Operation

1) Check Stack Underflow condition

   If(top== -1)

   Then print stack is underflow

2) Otherwise, delete the top position element

   Popped_item=stack_arr[top]

3) Decrease the position of top

   top=top-1

4) Print the popped_item

Try writing the code for Pop Operation function

POP Operation

```
pop()
{
        if(top == -1)
                printf("Stack Underflow\n");
        else
        {
                printf("Popped element is : %d\n",stack_arr[top]);
                top=top-1;
        }
}/*End of pop()*/
```

Try writing the code for Display Operation function

Display fn

```
display()
{
        int i;
        if(top == -1)
                printf("Stack is empty\n");
        else
        {
                printf("Stack elements :\n");
                for(i = top; i >=0; i--)
                        printf("%d\n", stack_arr[i] );
        }
}/*End of display()*/
```

Peek Operation

- An operation on certain abstract data types, specifically sequential collections such as stacks and queues,
  - which returns the value of the top ("front") of the collection
  - without removing the element from the collection.
  - It thus returns the same value as operations such as "pop" or "dequeue", but does not modify the data.

Peek Operation

- The name "peek"
  - the name for this operation varies depending on data type and language.

- Peek is generally considered an inessential operation,
  - not included in the basic definition of these data types.

Peek Operation

- Sequential types for which peek is often implemented include:
  - Stack
  - Queue
  - Priority queue (such as a heap)
  - Double-ended queue (deque)
  - Double-ended priority queue (DEPQ)

Peek Operation

- Single-ended types, such as stack,
  - generally only admit a single peek, at the end that is modified.
- Double-ended types, such as deques,
  - admit two peeks, one at each end.

- Names for peek vary.
  - For queues "front" is common.
  - Dequeues have varied names, often "front" and "back" or "first" and "last".
  - The name "peak" is also occasionally found

04-09-2023

Linked List Representation of Stack

Linked List Representation of Stack

- The info field of the node holds the elements of the stack
- Link fields hold pointers to the neighbouring element in the stack
- **The start pointer behaves as the TOP pointer variable of the stack**
- **Null pointer of the last node in the linked list signals the bottom of the stack**

Linked List Representation of Stack

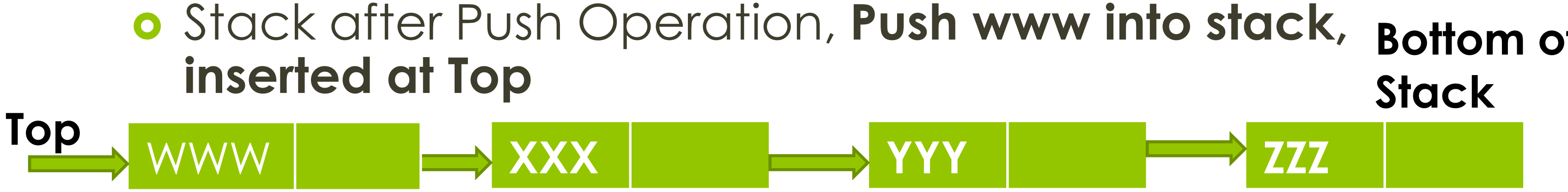
```
struct node
{
        int info;
        struct node *link;
} *top=NULL;
```

Push Operation

- Inserting a node into the front or start of the list. i.e. at the top of the stack

- Stack after Before Push Operation,

- Stack after Push Operation, **Push www into stack, inserted at Top**



Linked List Representation of Stack

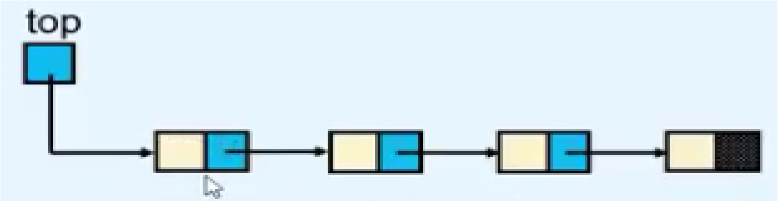
Beginning of linked list → top of the stack

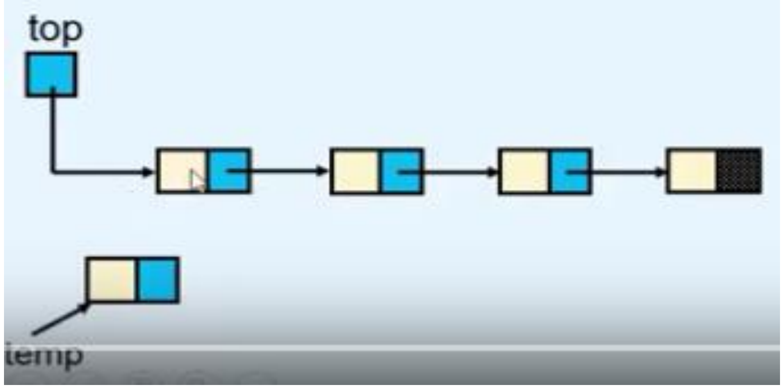
top Point to first node of the list

Stack is empty → top is NULL

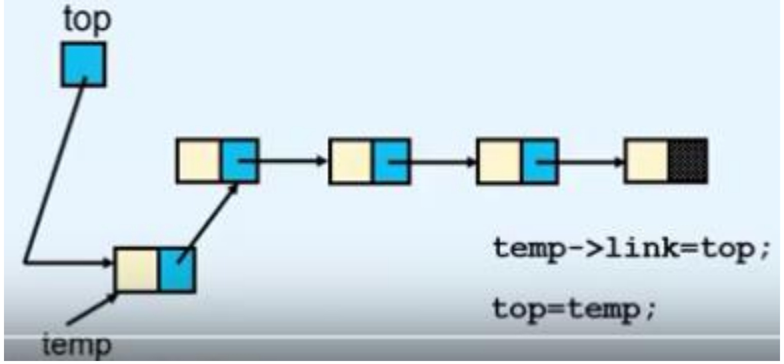
Push operation	Pop operation
Node is inserted in the beginning of the list	First node of the list is deleted

PUSH Operation on Stack





tmp = (struct node *)malloc(sizeof(struct node));



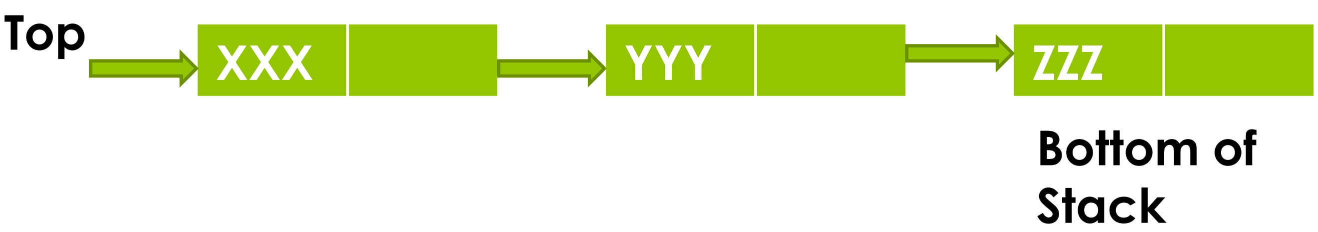
tmp->link=top;
top=tmp;

Push Operation

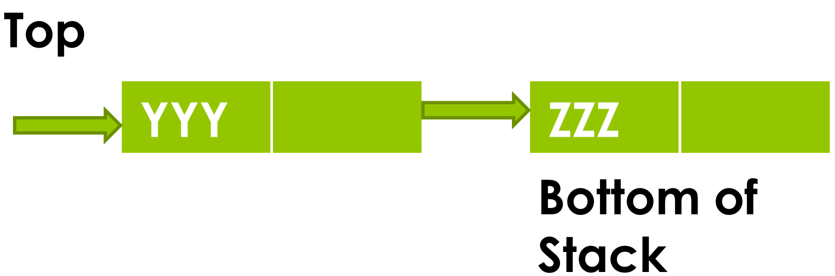
```
push()
{
        struct node *tmp;
        int pushed_item;
        tmp = (struct node *)malloc(sizeof(struct node));
        printf("Input the new value to be pushed on the stack : ");
        scanf("%d",&pushed_item);
        tmp->info=pushed_item;
        tmp->link=top;
        top=tmp;
}/*End of push()*/
```

Pop Operation

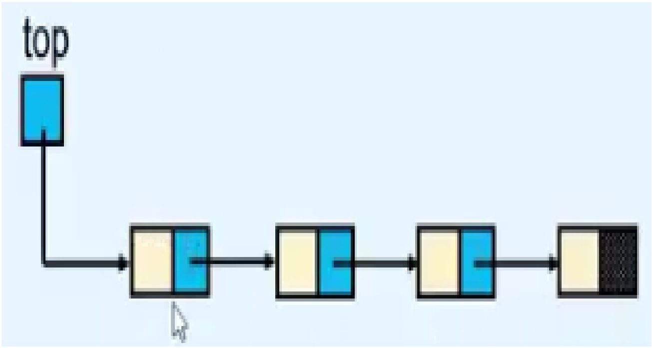
- Deleting a node pointed to by start pointer of the list. i.e. at the top of the stack
- Stack after Before Pop Operation

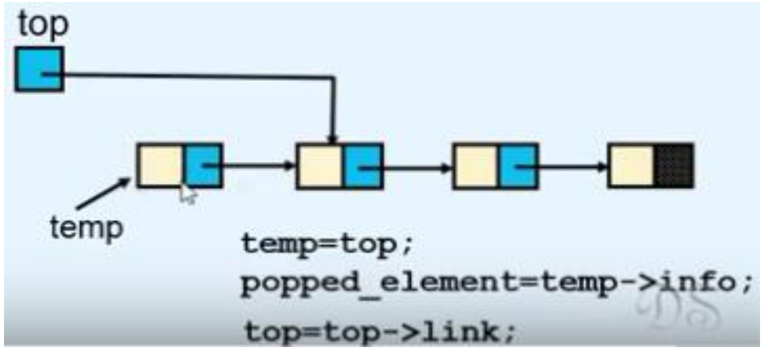


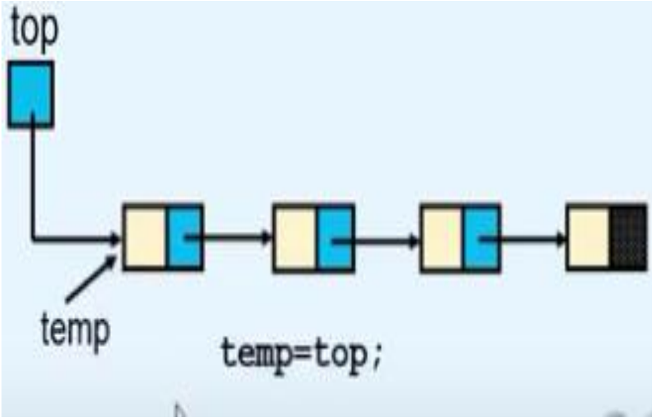
- Stack after Pop Operation, element at top deleted

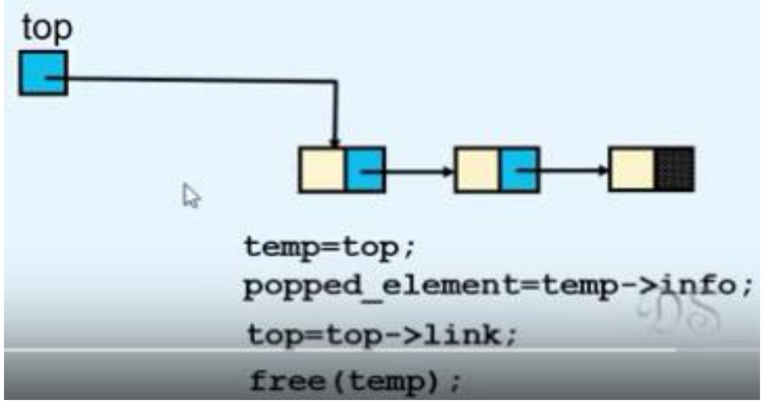


POP Operation on Stack









POP Operation

```
pop()
{
        struct node *tmp;
        if(top == NULL)
                printf("Stack is empty\n");
        else
        {
                tmp=top;
                printf("Popped item is %d\n",tmp->info);
                top=top->link;
                free(tmp);
        }

}/*End of pop()*/
```

Display Operation

```
display()
{       struct node *ptr;
            ptr=top;
            if(top==NULL)
                        printf("Stack is empty\n");
            else
            {
                        printf("Stack elements :\n");
                        while(ptr!= NULL)
                        {
                                    printf("%d\n",ptr->info);
                                    ptr = ptr->link;
                        }/*End of while */
            }/*End of else*/
}/*End of display()*/
```

04-09-2023

Application of Stack

Application of Stack –Reversal of a String

- Push each character of the string on the stack.
- When whole string is pushed on the stack
- Pop the characters from the stack to get the reversed string

Application of Stack –Reversal of a String

- Try Writing the Code snippet for it……..

Application of Stack –Polish Notation

- Polish notation (PN), also known as
  - normal Polish notation (NPN)
  - Łukasiewicz notation
  - Warsaw notation,
  - Polish prefix notation or
  - simply prefix notation,

Polish Notation-Prefix Notation

- Is a mathematical notation in which operators precede their operands,
- The description "Polish" refers to the nationality of logician Jan Łukasiewicz who invented Polish notation in 1924.

Reverse Polish Notation-Postfix Notation

- Reverse Polish notation (RPN), in which operators follow their operands.
- **The term Polish notation is sometimes taken (as the opposite of infix notation) to also include reverse Polish notation.**

Expression Representation

- An expression is defined as a number of operands or data items combined using several operators.
- 3 popular methods for representation:
  - Infix Notation
  - Prefix Notation
  - Postfix Notation

Infix Notation

- Used in General Mathematics
- Operator is written in between the operands
- Eg- a+b, x+y*z
- Called infix because of the position of the operator in expression.

Evaluation-

- Evaluated left to right but operator precedence must be taken into consideration
- **Not used inside computer, due to additional complexity of handling of precedence**

Prefix Notation

- Operator is written before the operands
- Also called Polish Notation
- Eg- +ab, +x*yz

Postfix Notation

- Operator is written after the operands
- Also called Reverse Polish or Suffix Notation
- Eg- ab+, xyz*+

Polish Notation

- Prefix and Postfix are free from any precedence
- Computers use postfix form

Notation Conversion

- **Scan the expression from left to right**
- **Operator Precedence-**
  - **Paranthesis evaluated first**
  - **After that evaluation is on the basis of operator precedence**
    - **Logical Not**
    - **Exponential Operator**
    - **Multiplication/division/modulus**
    - **Addition/Subtraction**
    - **Left shift,Right Shift**
    - **Relational**
    - **Logical And**
    - **Logical Or**

Operator Precedence Table

Description	Operator	Associativity
Function expression	( )	Left to Right
Array Expression	[ ]	Left to Right
Structure operator	->	Left to Right
Structure operator	.	Left to Right
Unary minus	-	Right to left
Increment/Decrement	++ --	Right to Left
One's compliment	~	Right to left
Negation	!	Right to Left
Address of	&	Right to left
Value of address	*	Right to left
Type cast	(type)	Right to left
Size in bytes	sizeof	Right to left
Multiplication	*	Left to right
Division	/	Left to right
Modulus	%	Left to right
Addition	+	Left to right
Subtraction	-	Left to right
Left shift	<<	Left to right
Right shift	>>	Left to right
Less than	<	Left to right
Less than or equal to	<=	Left to right
Greater than	>	Left to right
Greater than or equal to	>=	Left to right
Equal to	==	Left to right
Not equal to	!=	Left to right

Description	Operator	Associativity
Bitwise AND	&	Left to right
Bitwise exclusive OR	^	Left to right
Bitwise inclusive OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	? :	Right to left
Assignment	=	Right to left
	*= /= %=	Right to left
	+= -= &=	Right to left
	^= |=	Right to left
	<<= >>=	Right to left
Comma	,	Right to left

Manual Conversion -Infix to Postfix

- A+ B * C — Given infix form
- Left to right scan, Brackets first then Operator Precedence
- A + <u>B C *</u> — Convert the multiplication
- A B C * + — Convert the addition

Manual Conversion -Infix to Postfix

- A+[(B+C)+(D+E)*F]/G

Manual Conversion -Infix to Postfix

- A+[(B+C)+(D+E)*F]/G
- Left to right scan, Brackets first then Operator Precedence
- A+[(<u>BC+</u>)+(D+E)*F]/G
- A+[(<u>BC+</u>)+(<u>DE+</u>)*F]/G
- A+[(<u>BC+</u>)+(<u>DE+</u>)F*]/G
- A+[(BC+)(DE+)F*+]/G
- A+[(BC+)(DE+)F*+]G/
- A[(BC+)(DE+)F*+]G/+
- ABC+DE+F*+G/+

Manual Conversion -Infix to Postfix

Convert the following expressions from Infix to Postfix;

1) **A + B + C + D**
2) **(A + B) / (C – D)**
3) **(A + B) * C - (D - E) * (F + G)**
4) **((A+B) — C * (D/E)) + F**
5) **((A + B) * (C– D) + E) / (F + G)**

Manual Conversion -Infix to Postfix

Convert the following expressions from Infix to Postfix;

1) **A + B + C + D=A B + C + D +**

=<u>AB+</u> +C +D

=<u>AB+C+</u> +D

=AB+C+D+

Manual Conversion -Infix to Postfix

Convert the following expressions from Infix to Postfix;

1) **A + B + C + D=A B + C + D +**
2) **(A + B) / (C – D) =A B + C D - /**

Manual Conversion -Infix to Postfix

Convert the following expressions from Infix to Postfix;

1) **A + B + C + D=A B + C + D +**
2) **(A + B) / (C – D) =A B + C D - /**
3) **(A + B) * C - (D - E) * (F + G)=AB+C*DE-FG+*-**

Manual Conversion -Infix to Postfix

Convert the following expressions from Infix to Postfix;

1) **A + B + C + D=A B + C + D +**
2) **(A + B) / (C – D) =A B + C D - /**
3) **(A + B) * C - (D - E) * (F + G)=AB+C*DE-FG+*-**
4) **((A+B) — C * (D/E)) + F=A B + C D E / * - F +**

Manual Conversion -Infix to Postfix

Convert the following expressions from Infix to Postfix;

1) **A + B + C + D=A B + C + D +**
2) **(A + B) / (C – D) =A B + C D - /**
3) **(A + B) * C - (D - E) * (F + G)=AB+C*DE-FG+*-**
4) **((A+B) — C * (D/E)) + F=A B + C D E / * - F +**
5) **((A + B) * (C– D) + E) / (F + G)=A B + C D - * E + F G + /**

Manual Conversion -Infix to Prefix

- A/B^C+D
- Left to right scan, Brackets fist then Operator Precedence
- A/^BC+D
- /A^BC+D
- +/A^BCD

Manual Conversion -Infix to Prefix

Convert the following expressions from Infix to Prefix;

1) **(P*Q^R+S)**
2) **(A-B/C)*(D*E-F)**
3) **(A*B+(C/D))-F**
4) **A+B*C-(D/E^F)*G*H**
5) **(A+B)*C/D+E^F/G**

Manual Conversion -Infix to Prefix

Convert the following expressions from Infix to Prefix;

1) **(P*Q^R+S)=+*P^QRS**

Manual Conversion -Infix to Prefix

Convert the following expressions from Infix to Prefix;

1) **(P*Q^R+S)=+*P^QRS**
2) **(A-B/C)*(D*E-F)=*(-A/BC)(-*DEF)**

Manual Conversion -Infix to Prefix

Convert the following expressions from Infix to Prefix;

1) **(P*Q^R+S)=+*P^QRS**
2) **(A-B/C)*(D*E-F)=*(-A/BC)(-*DEF)**
3) **(A*B+(C/D))-F=-+*AB/CDF**

Manual Conversion -Infix to Prefix

Convert the following expressions from Infix to Prefix;

1) **(P*Q^R+S)=+*P^QRS**
2) **(A-B/C)*(D*E-F)=*(-A/BC)(-*DEF)**
3) **(A*B+(C/D))-F=-+*AB/CDF**
4) **A+B*C-(D/E^F)*G*H**

=A+*BC-(/D^EF)*G*H

=A+*BC-*/D^EFG*H

=A+*BC-/D^EFGH**

=+A*BC-/D^EFGH**

=-+A*BC/D ^EFGH**

Manual Conversion -Infix to Prefix

Convert the following expressions from Infix to Prefix;

1) **(P*Q^R+S)=+*P^QRS**
2) **(A-B/C)*(D*E-F)=*(-A/BC)(-*DEF)**
3) **(A*B+(C/D))-F=-+*AB/CDF**
4) **A+B*C-(D/E^F)*G*H**

=A+*BC-(/D^EF)*G*H
=A+*BC-*/D^EFG*H
=A+*BC-/D^EFGH**
=+A*BC-/D^EFGH**
=-+A*BC/D ^EFGH**

5) **(A+B)*C/D+E^F/G**

=+AB*C/D+E^F/G
=+AB*C/D+^EF/G
=*+ABC/D+^EF/G
=/*+ABCD+^EF/G
=/*+ABCD+/^EFG
=+/*+ABCD/^EFG

Algorithm for Infix to Postfix using stack

Step 1-Put the opening bracket '(' on the start of the expression and add the closing bracket ')' at the end of the input expression 'P'

Step 2-Scan the Expression P from left to right and repeat steps 3 to 6 for each element of P until the stack is empty

Step 3-If an operand is encountered, add it to the output expression 'Q'

Step 4-If a left bracket is encountered push it onto the stack

Algorithm for Infix to Postfix using stack

Step 5-If an operator is encountered then

5a)Repeatedly pop from the stack and add to the output expression Q, **each operator which has the same or higher priority** compared to the operator just scanned.

5b) then Add the operator just scanned to the stack

Step 6-Ifa right bracket ')' is encountered then repeatedly pop from the stack and add to the output expression Q until a left bracket is encountered.

Pop the left bracket also

Infix to Postfix Conversion using stack

Eg- P=(A+B)*C

Infix to Postfix Conversion using stack

Eg- P=(A+B)*C

P=((A+B) * C)

Scan	Stack	Expression Q
(	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*	AB+C
)	NULL	AB+C*

Infix to Postfix Conversion using stack

Convert P/(Q-R)*S+T

Infix to Postfix Conversion using stack

Convert P/(Q-R)*S+T=(P/(Q-R)*S+T)

Scan	Stack	Expression Q
(	(	
P	(	P
/	(/	P
(	(/ (	P
Q	(/ (	PQ
-	(/ (-	PQ
R	(/ (-	PQR
)	(/	PQR-
*	(*	PQR-/
S	(*	PQR-/S
+	(+	PQR-/S*
T	(+	PQR-/S*T
)	NULL	PQR-/S*T+

Infix to Postfix Conversion using stack

Convert (P/(Q-R)*S+T)

(P/(Q–R)*S+T))

Symbol	Stack	Expression
(	(	—
P	(	P
/	(/	P
(	(/(	P
Q	(/(	PQ
–	(/(–	PQ
R	(/(–	PQR
)	(/	PQR–
*	(*	PQR–/
S	(*	PQR–/S
+	(+	PQR–/S*
T	(+	PQR–/S*T
	null	PQR-/S*T+

Infix to Postfix Conversion using stack

Convert P=(a && b||c || ! (e > f))

Notation Conversion

- **Scan the expression from left to right**
- **Operator Precedence-**
  - **Paranthesis evaluated first**
  - **After that evaluation is on the basis of operator precedence**
    - **Logical Not**
    - **Exponential Operator**
    - **Multiplication/division/modulus**
    - **Addition/Subtraction**
    - **Left shift,Right Shift**
    - **Relational**
    - **Logical And**
    - **Logical Or**

Infix to Postfix Conversion using stack

Convert P=(a && b| |c | | ! (e > f))

Scan	Stack	Expression
(	(	
a	(	a
&&	(&&	a
b	(&&	ab
| |	(| |	ab&&
c	(| |	ab&&c
| |	(| |	ab&&c| |
!	(| |!	ab&&c| |
(	(| |!(	ab&&c| |
e	(| |!(	ab&&c| |e
>	(| |!(>	ab&&c| |e
f	(| |!(>	ab&&c| |ef
)	(| |!	ab&&c| |ef>
)	NULL	ab&&c| |ef>!| |

- **Operator Precedence**
- **Logical Not**
- **Exponential Operator**
- **Multiplication/division /modulus**
- **Addition/Subtraction**
- **Left shift,Right Shift**
- **Relational**
- **Logical And &&**
- **Logical Or | |**

Infix to Postfix Conversion using stack

Convert P=(A+(B*C-(D/E^F)*G*H))

Infix to Postfix Conversion using stack

Convert P=(A+(B*C-(D/E^F)*G*H))

Scan	Stack	Expression Q
(	(	
A	(	A
+	(+	A
(	(+(	AB
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/ ^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
*	(+(-*	ABC*DEF^/G*
H	(+(-*	ABC*DEF^/G*H
)	(+	ABC*DEF^/G*H*-
)	null	ABC*DEF^/G*H*-+

Convert P=(A+(B*C-(D/E^F)*G*H))

Scan	Stack	Expression Q
(	(	
A	(	A
+	(+	A
(	(+(	AB
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
*	(+(-*	ABC*DEF^/G*
H	(+(-*	ABC*DEF^/G*H
)	(+	ABC*DEF^/G*H*-
)	null	ABC*DEF^/G*H*-+

Manual Evaluation of a Prefix notation

- Lets take an eg: +5*32
- Find an operator from left to right having 2 operands after it
- Multiplication of 3 and 2 is carried out
- Expression becomes +56
- Now + has two operands so evaluated
- Exp=11

Manual Evaluation of a Postfix notation

- Lets take an eg: 532*+
- Find first operator from left to right having 2 operands before it
- perform the operation
- Multiplication of 3 and 2 is carried out
- Expression becomes 56+
- Now + is evaluated
- Exp=11

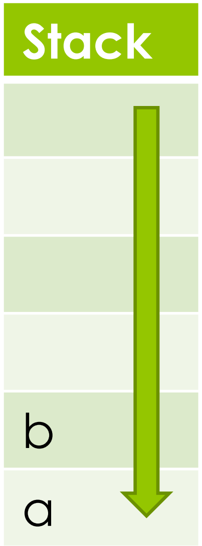
Algorithm for Evaluation of Postfix using stack

Step 1-Scan the Expression P from left to right and repeat steps 2 and 3 for each element of 'P' until the last element

Step 2- If an operand is encountered, push it on the stack

1st operand say 'a' is pushed

2nd operand say 'b' is pushed



Step 3-If an operator is encountered then

3a) Remove the top two elements of stack and perform

a operator b

(where b is the top element and a is the next element to top element)

3b)Push the result onto the stack

Algorithm for Evaluation of Postfix using stack

P=653+9*+
P=53+9*+
P=3+9*+
P=+9*+
Evaluate 5+3=8
Push 8
P=9*+
P=*+
Evaluate
8*9=72
Push 72
P=+
Evaluate 6+72=78
Ans=78

Stack

Stack
6

Stack
5
6

Stack
3
5
6

Stack
8
6

Stack
9
8
6

Stack
72
6

Algorithm for Evaluation of Postfix using stack

Evaluate the following expression-

1) 432*+5-
2) 532*+4-5+
3) 53+82-*
4) Evaluate 562+*(12)4/-
   taking 12 as a single number

Evaluate 432*+5-

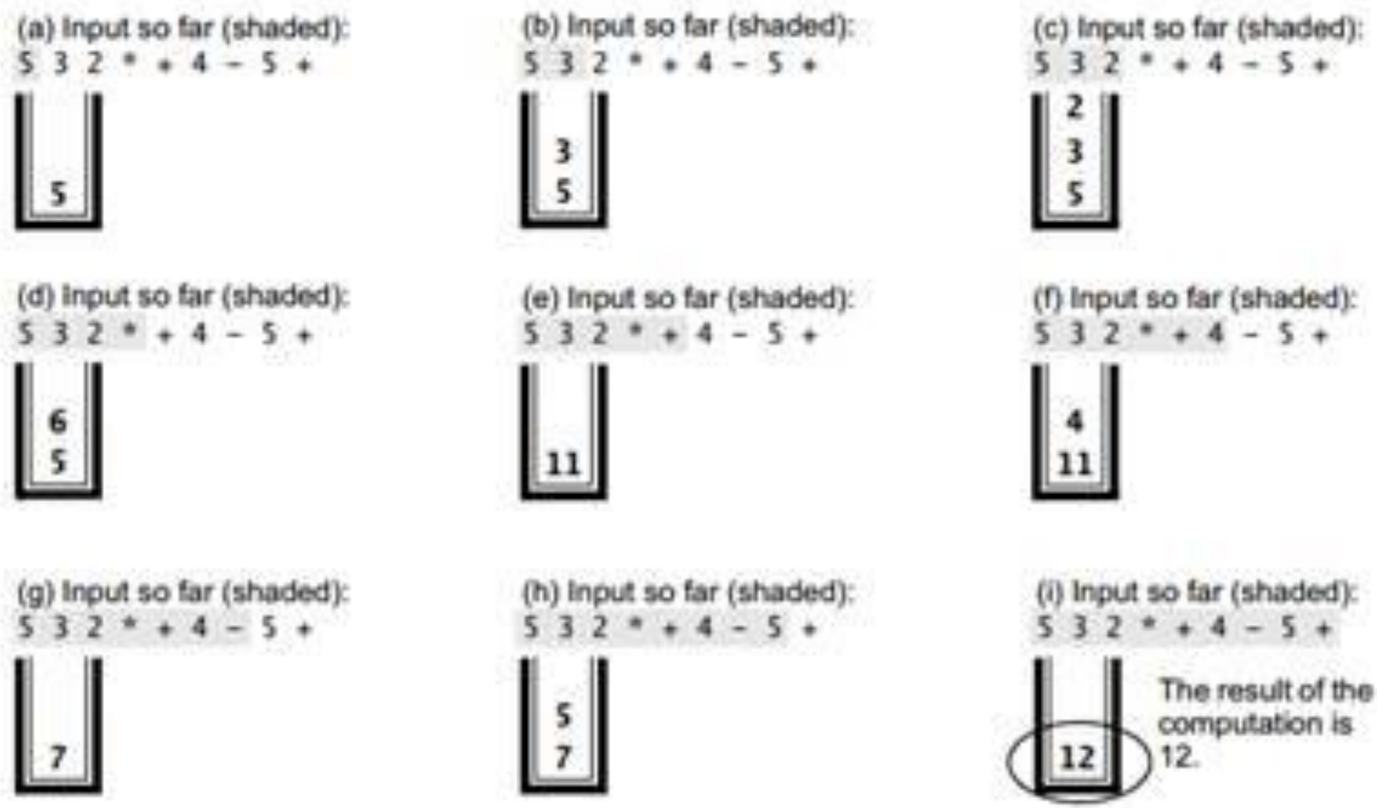
Postfix → a b c * + d -   Let, a = 4, b = 3, c = 2, d = 5

Postfix → 4 3 2 * + 5 -

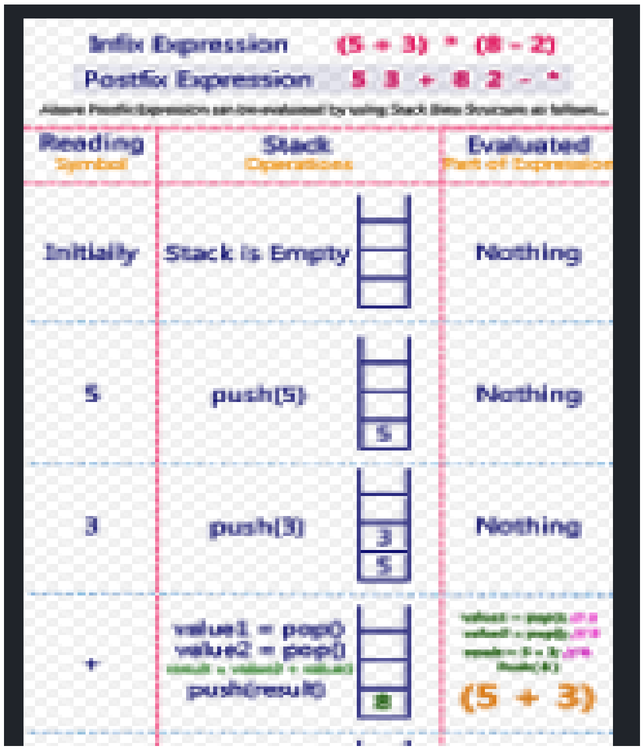
Operator/Operand	Action	Stack
4	Push	4
3	Push	4, 3
2	Push	4, 3, 2
*	Pop (2, 3) and 3*2 = 6 then Push 6	4, 6
+	Pop (6, 4) and 4+6 = 10 then Push 10	10
5	Push	10, 5
-	Pop (5, 10) and 10-5 = 5 then Push 5	5

Example

Evaluate the postfix expression
5 3 2 * + 4 - 5 +



Postfix Expression 5 3 + 8 2 - *



Courtesy:http://btechsmartclass.com/data_structures/postfix-evaluation.html

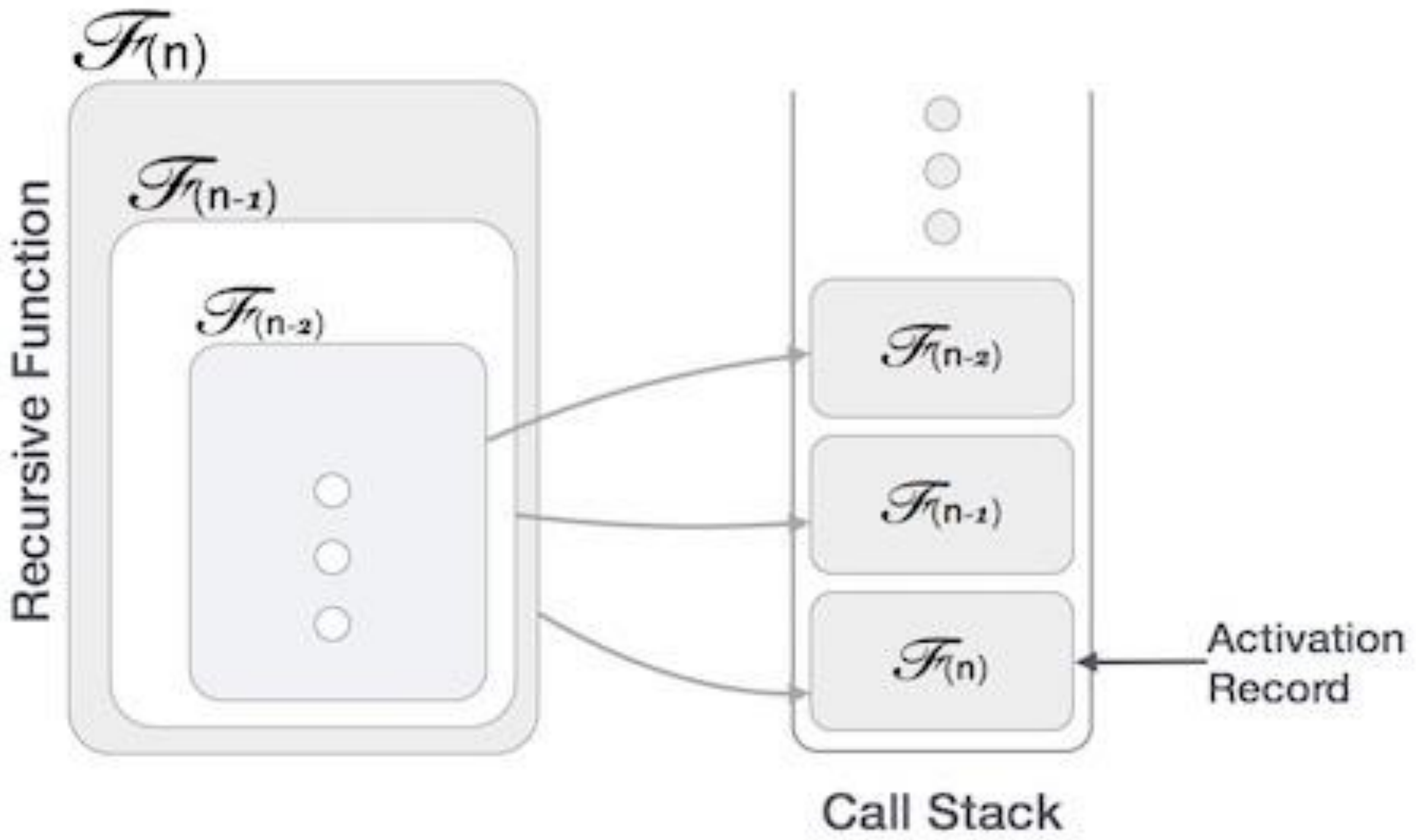
Evaluate 562+*(12)4/- taking 12 as a single number

5 6 2 + * 12 4 / -

Step	Input Symbol/Element	Stack	Intermediate Calculations Output
1	5 Push	5	
2	6 Push	5, 6	
3	2 Push	5, 6, 2	
4	+ Pop 2 elements and evaluate	5	6 + 2 = 8
5	Push result 8	5, 8	
6	* Pop 2 elements and evaluate	# empty	5 x 8 = 40
7	Push result 40	40	
8	12 Push	40,12	
9	4 Push	40, 12, 4	
10	/ Pop 2 elements and evaluate	40	12 / 4 = 3
11	Push result 3	40, 3	
12	- Pop 2 elements and evaluate	# empty	40 – 3 = 37
13	Push result 37	37	
14	No more elements		37

Courtesy:https://unacademy.com/lesson/evaluation-of-a-postfix-expression-in-tabular-form/8Z5PDFL5

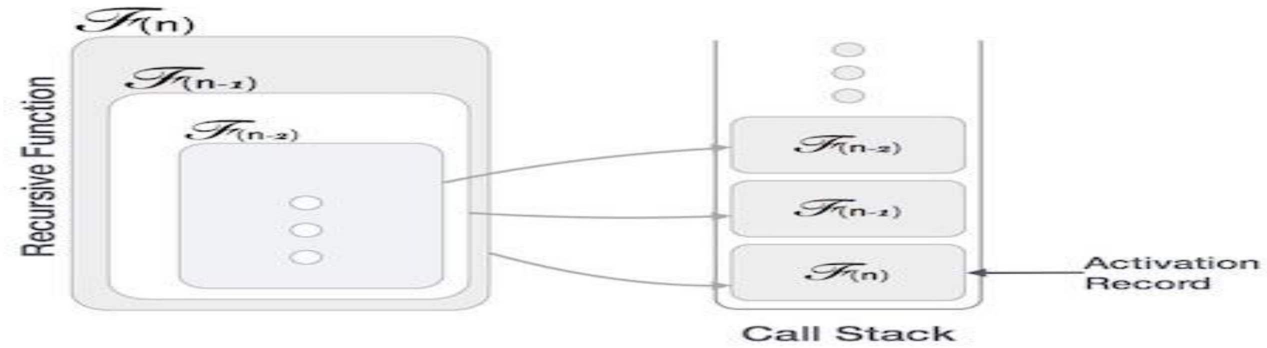
Application of Stack –Recursion



Application of Stack –Recursion

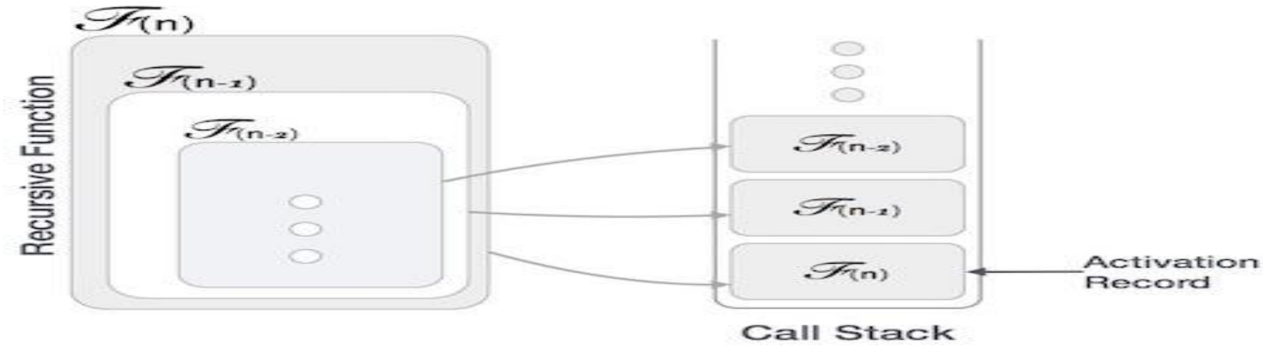
- Many programming languages implement recursion by means of **stacks**.

- A function (**caller**) calls another function (**callee**) or itself as callee,
  - The caller function transfers execution control to the callee.
  - This transfer process may also involve some data to be passed from the caller to the callee.



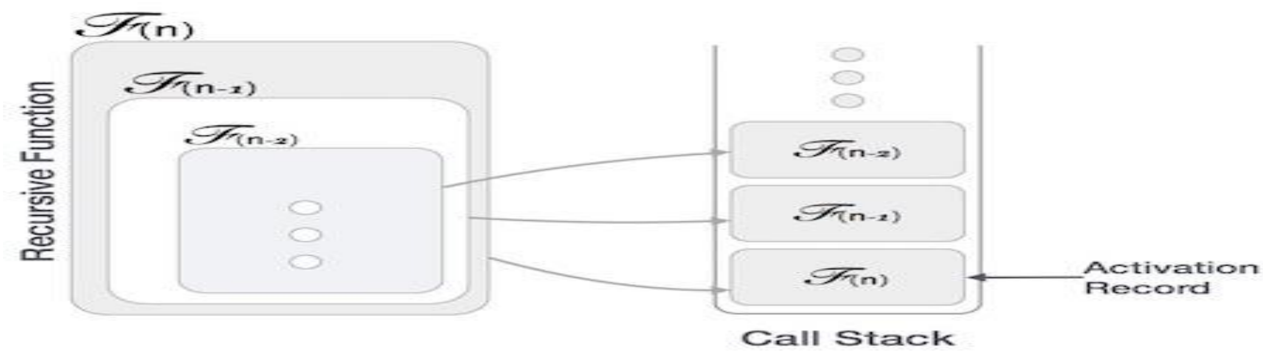
Application of Stack –Recursion

- The caller function has
  - to suspend its execution temporarily and
  - resume later
  - when the execution control returns from the callee function.



Application of Stack –Recursion

- Here, the caller function needs to start exactly from the point of execution where it puts itself on hold.
  - It also needs the exact same data values it was working on.
  - So, an activation record (or stack frame) is created for the caller function.
  - Activation record keeps the information about
    - local variables,
    - formal parameters,
    - return address and
    - all information passed to the callee function.

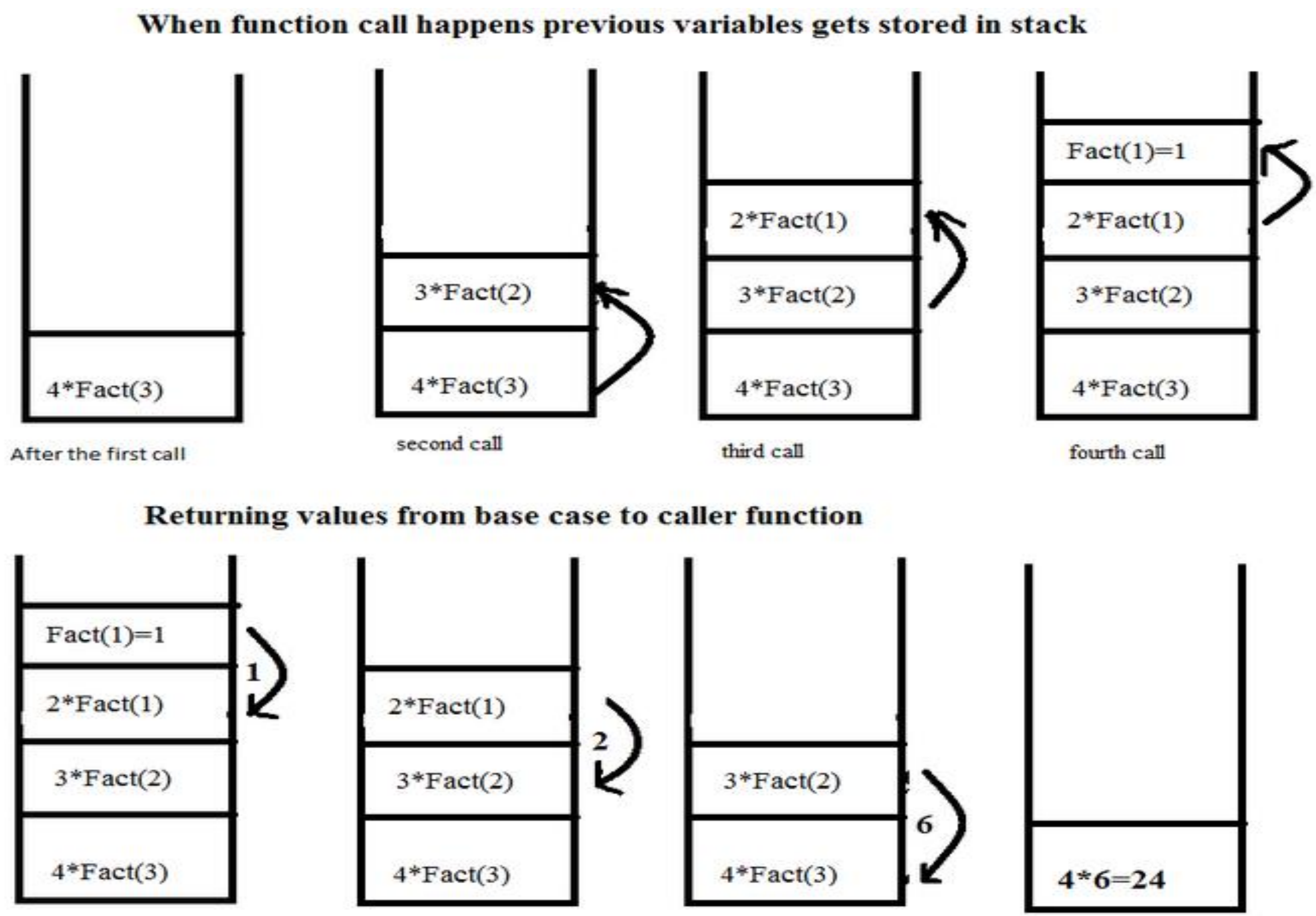


Application of Stack –Recursion

A recursive function to find the factorial of a positive whole number

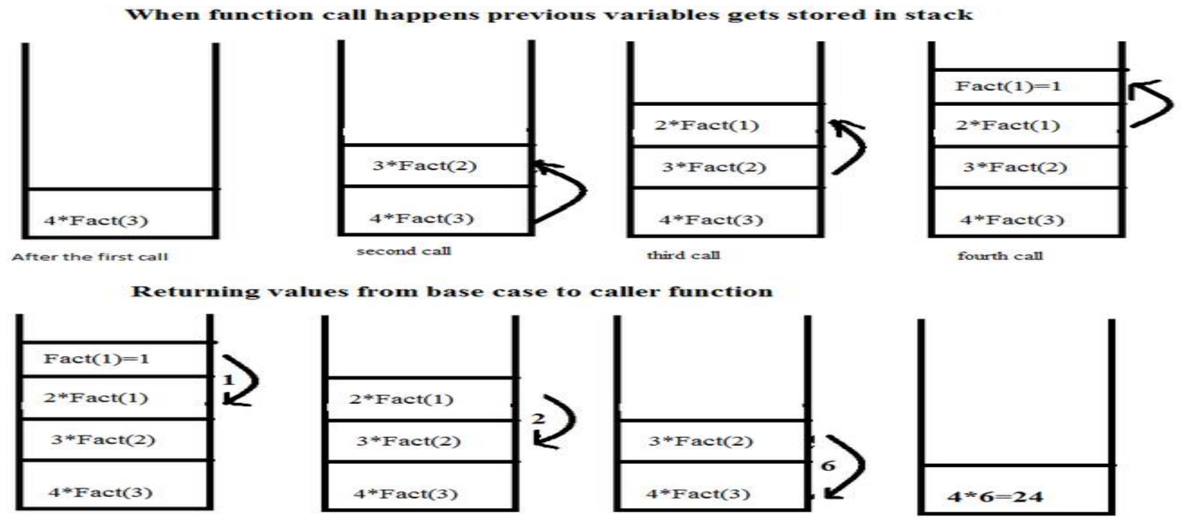
```
 int factorial(n)
{
        if (n == 1)
        {
                // base case
                return 1;
        }
        else
        {
                return n * factorial(n - 1);
                // function calls itself
        }
}
```

Application of Stack –Recursion



Application of Stack –Recursion

- Functions calls are 'stacked' one on top of the other,
- This is called the call stack (or execution stack)
- The call stack operates on a "Last In, First Out" basis. An item is "pushed" onto a stack on function call, and an item is "popped" off the stack when that function returns a value.



Application of Stack –Recursion

Time Complexity

- In iterations,
  - we take number of iterations to count the time complexity.
- In recursion,
  - assuming everything is constant,
  - count the number of times a recursive call is being made.

Space Complexity

- Space complexity is counted as
  - what amount of extra space is required for a module to execute.
- In iterations,
  - the compiler hardly requires any extra space.
  - The compiler keeps updating the values of variables used in the iterations.

Application of Stack –Recursion

Time Complexity

- A call made to a function is O(1),
  - hence the (n) number of times a recursive call is made
  - makes the recursive function O(n).

Space Complexity

- In recursion,
  - the system needs to store activation record each time a recursive call is made.
  - Thus, space complexity of recursive function
    - may go higher than that of a function with iteration.