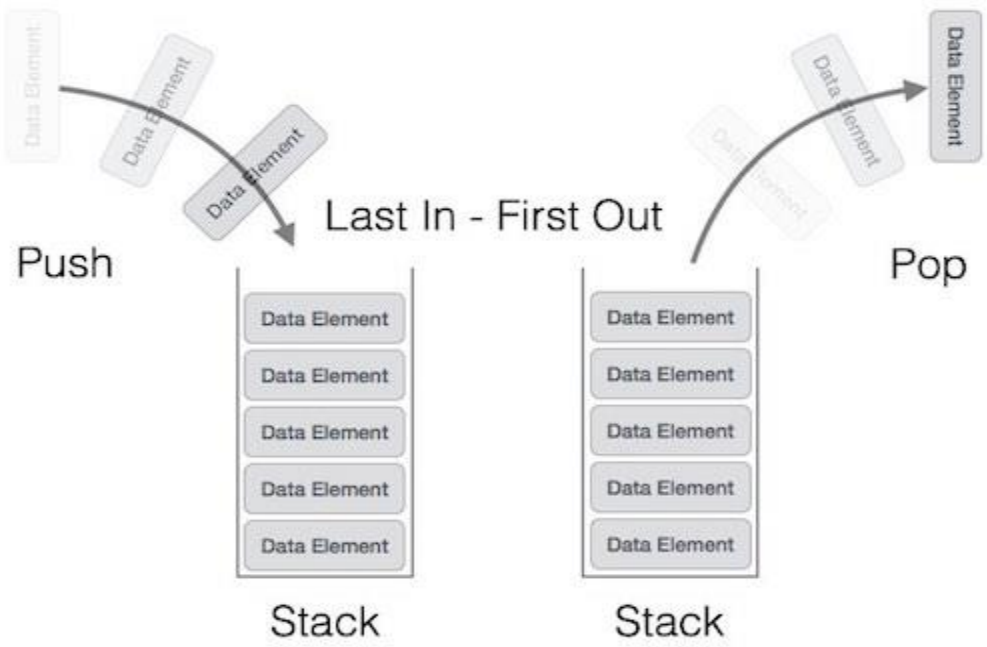
Stacks

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).



What is this good for ?

- To store history in a Web browser
- Undo sequence in a any application software or text editor
- Saving local variables during function calls
- Recursions
- Watchlists?

A Stack

- Definition:
  - An ordered collection of homogenous data items
  - Can be accessed at only one end (the top)
- Operations:
  - Create an empty stack
  - check if it is empty
  - Push: add an element to the top
  - Pop: remove the top element
  - Peek: retrieve the top element(Not the deletion)
  - Destroy : remove all the elements one by one and destroy the data structure

The Stack ADT: Value definition

Abstract typedef StackType(ElementType ele)

Condition: none

Stack ADT: Operator definition

1.Abstract StackType CreateStack()

Precondition: none

Postcondition: EmptyStack is created

2. Abstract StackType PushStack(StackType Stack, ElementType Element)

Precondition: Stack not full or NotFull(Stack)= True

Postcondition: stack= stack + Element at the top

Or Stack= original stack with new Element at the top

Stack ADT: Operator definition

3.Abstract ElementType PopStack(StackType stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: PopStack= element at the top,

Stack = stack - Element at the top

Or Stack= original stack without top Element

4.Abstract DestroyStack(StackType Stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: Element from the stack are removed one by one starting from top to bottom.

Empty(Stack)= True

Stack ADT: Operator definition

5.Abstract Boolean NotFull(StackType stack)

Precondition: none

Postcondition: NotFull(Stack)= true if Stack is not full

NotFull(Stack)= False if Stack is full.

6.Abstract Boolean NotEmpty(StackType stack)

Precondition: none

Postcondition: NotEmpty(Stack)= true if Stack is not empty

~Empty(Stack)= False if Stack is empty.

Stack ADT: Operator definition

7. Abstract ElementType Peek(StackType stack)

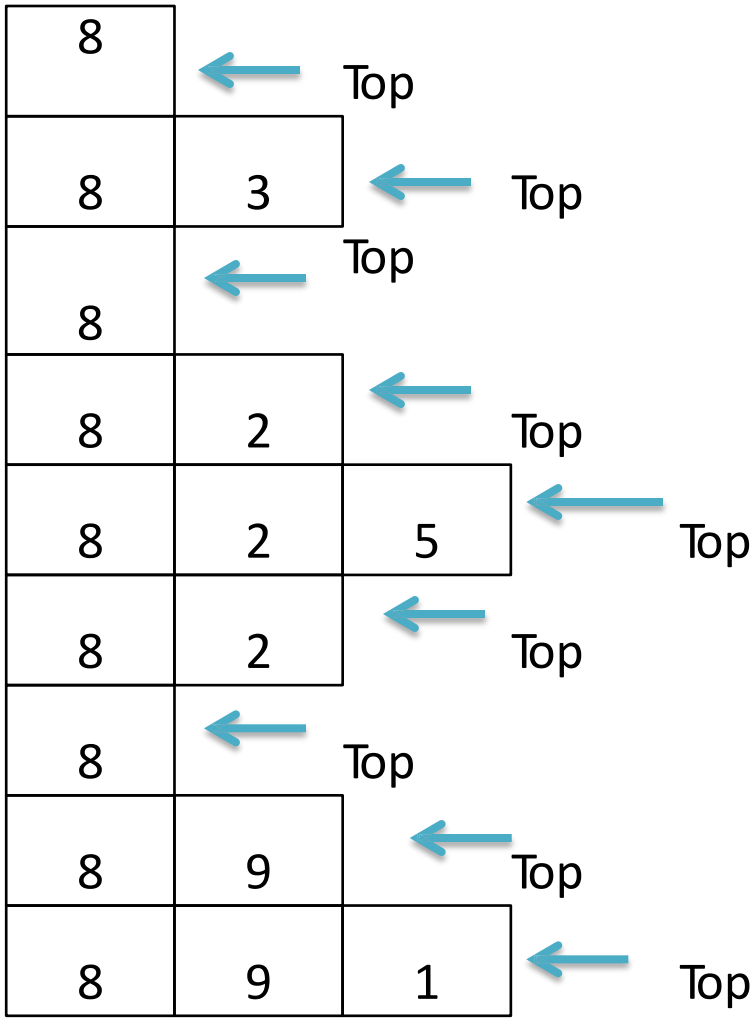
Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: Peek= element at the top,

Stack = original stack

Exercise: Stacks

- Push(8)
- Push(3)
- Pop()
- Push(2)
- Push(5)
- Pop()
- Pop()
- Push(9)
- Push(1)



Implementing a Stack

- At least three different ways to implement a stack
  - array
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

Implementing Stacks: Array

- Advantages -best performance
- Disadvantage - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list

Writing an algorithm

- Specify algorithm name, list of inputs, data types of the inputs and return data types clearly
- Specify purpose of the algorithm
- Algo should produce at least one Output
- Definiteness: Each step must be clear and unambiguous.
- Should react correctly to all valid and invalid inputs
- Finiteness: If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- Effectiveness:
- Comment Session: Comment is additional info of program for easily modification. In algorithm comment would be appear between two square bracket []. For example: [this is a comment of an algorithm].

Stack ADT: Array Implementation

1. Algorithm StackType CreateStack()

//This Algorithm returns an empty stack- stack

{ integer StackTop =-1;

Return stack;

}

2.Algorithm StackType PushStack(StackType Stack, ElementType Element)

// This algorithm accepts a StackType stack and ElementType Element as input and adds ‘Element’ at the top of ‘stack’. StackTop is an integer index that holds current value of StackTop position.

{

if NotFull(Stack)= True

stack[++StackTop]= Element

Else “Error Message”

}

Stack ADT: Array Implementation

3. Algorithm ElementType PopStack(StackType stack)

// This algorithm accepts a stack as input and returns ‘Element’ at the top of ‘stack’.

```
{  if NotEmpty(Stack)= True
Return Stack[StackTop--]
Else  print “Error Message”
}
```

4. Abstract DestroyStack(StackType Stack)

//This algorithm returns all the elements from Stack in LIFO order and destroys the data structure

```
{ if NotEmpty(Stack) = true
     while(NotEmpty(Stack))
          print PopStack(Stack)
   else print “Error Message”
}
```

Stack ADT: Array Implementation

5. Abstract Boolean NotFull(StackType stack)

```
// This algorithm returns true if the stack is not full, false otherwise.
{ if NotFull(Stack)
        retrun True
  else
        return False
}
```

6. Abstract Boolean NotEmpty(StackType stack)

```
// This algorithm returns true if the stack is not empty, false otherwise.
{ if NotEmpty(Stack)
        retrun True
  else
        return False
}
```

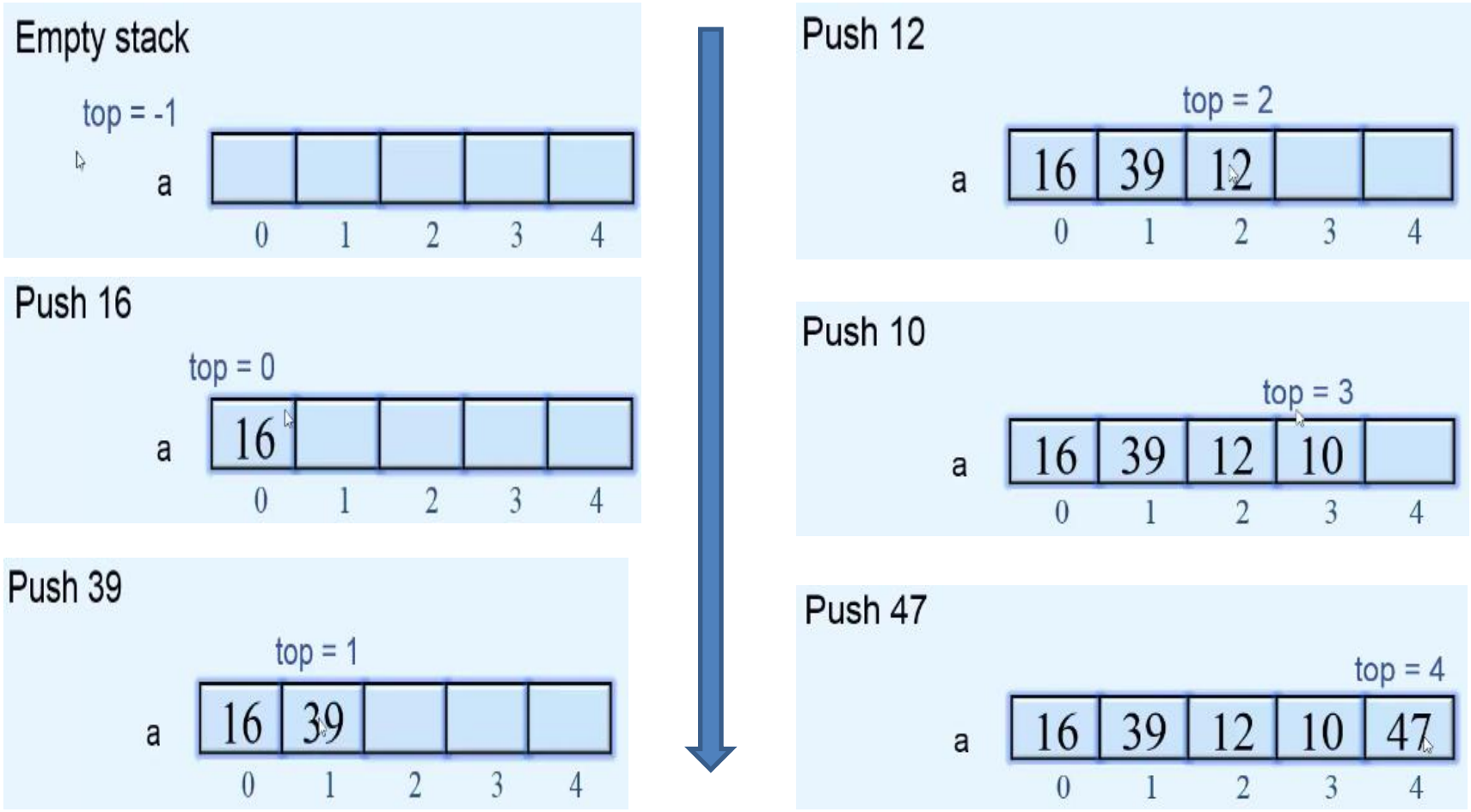
Stack ADT: Array Implementation

7. Abstract ElementType Peek(StackType stack)

//// This algorithm accepts a stack as input and returns 'Element' at the top of 'stack'.

```
{  if NotEmpty(Stack)= True
Return Stack[StackTop]
Else  print "Error Message"
}
```

PUSH Operation on Stack

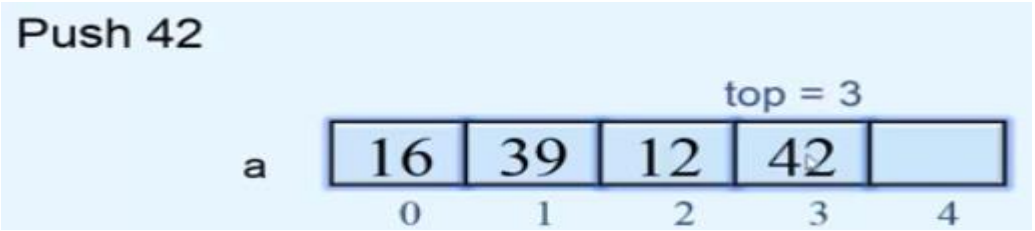


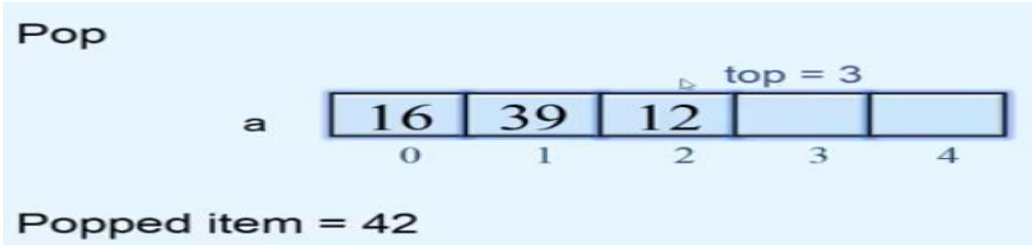
POP Operation on Stack

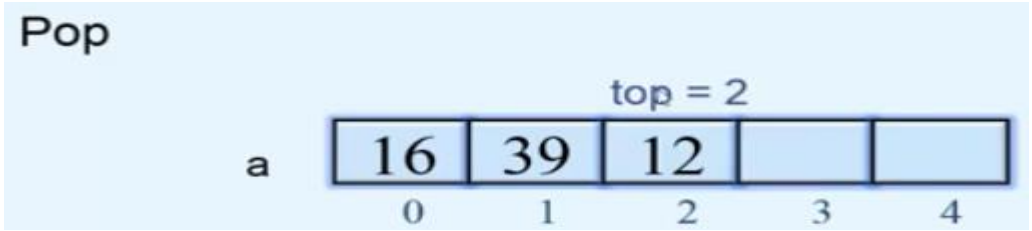
24-08-2023

Stack Underflow condition

If(top== -1)





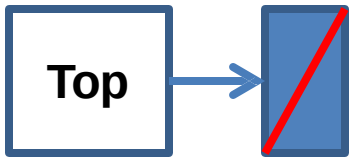


Implementing Stacks: Linked List

```
Struct NodeType{
                ElementType Element;
                NodeType Next;
        }
```

1. Algorithm StackType CreateStack()

```
//This Algorithm creates and returns an empty stack- pointed by a pointer-Top
{ createNode(Top);
Top =NULL;
}
```



Implementing Stacks: Linked List

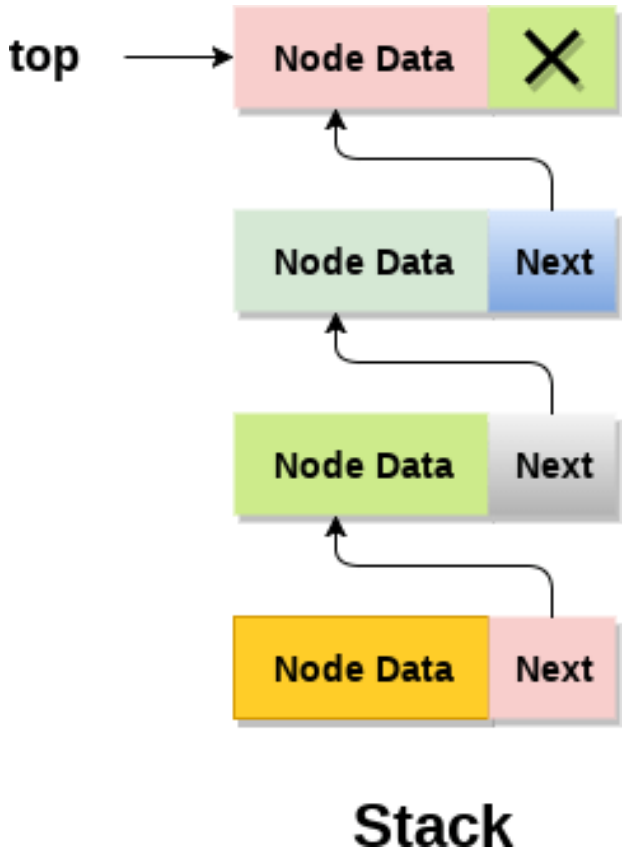


Image Courtesy: https://www.javatpoint.com/ds-linked-list-implementation-of-stack

Implementing Stacks: Linked List

2. StackType PushStack(StackType Stack, NodeType NewNode)

// This Algorithm adds a NewNode at the top of ‘stack’. Top is an pointer that points to the topmost Stack node.

```
{ if Top ==NULL // first element in stack
        NewNode->Next = NULL;
        Top=NewNode;
 Else NewNode->Next=Top;// General case
        Top=NewNode;
}
```

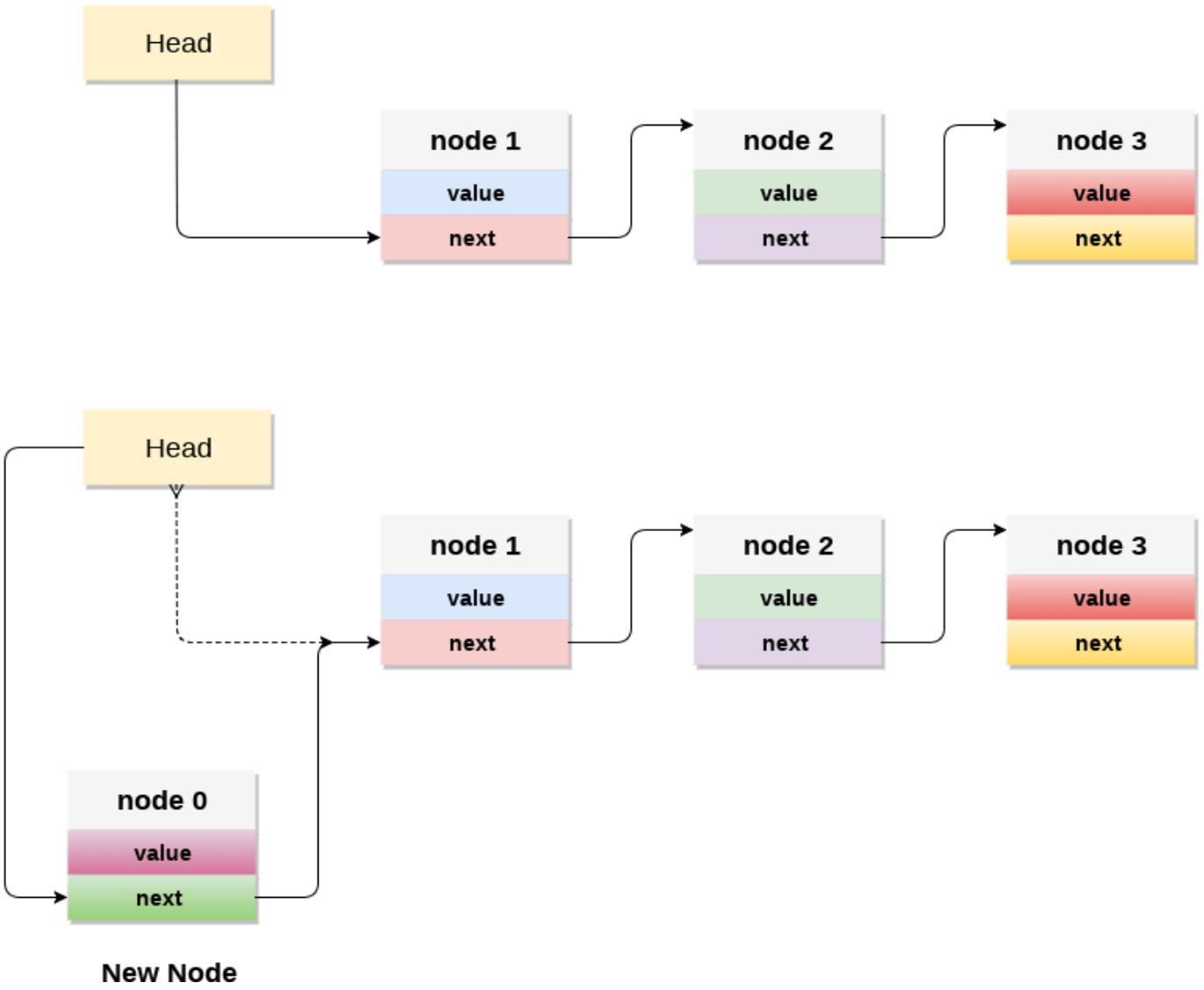


Image Courtesy: https://www.javatpoint.com/ds-linked-list-implementation-of-stack

Implementing Stacks: Linked List

3. Algorithm ElementType PopStack(StackType stack)

//This algorithm returns value of ElementType stored in topmost node of stack. Temp is a temproary node used in pop process.

```
{ if Top==NULL
        Print “Underflow”
 Else
        {
        createNode(Temp);
        Temp=Top;
        Top= Top->next;
        Return(temp->Data);
        }
}
```

Implementing Stacks: Linked List

4. Abstract DestroyStack(StackType Stack)

//This algorithm returns values stored in data structure and free the memory used in data structure implementation.

```
{{ if Top==NULL
        Print “Underflow”
 Else    createNode(Temp);
        while(NotEmpty(stack))
        {
        Temp=Top;
        Top= Top->next;
        Return(temp->Data);
        }
}
```

Implementing Stacks: Linked List

5. Abstract ElementType Peek(StackType stack)

//This algorithm returns value of ElementType stored in topmost node of stack.

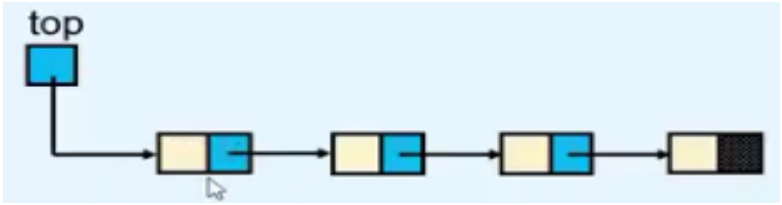
```
{ if Top==NULL
        Print “Error Message”
  Else
        Return(Top->Data);
}
```

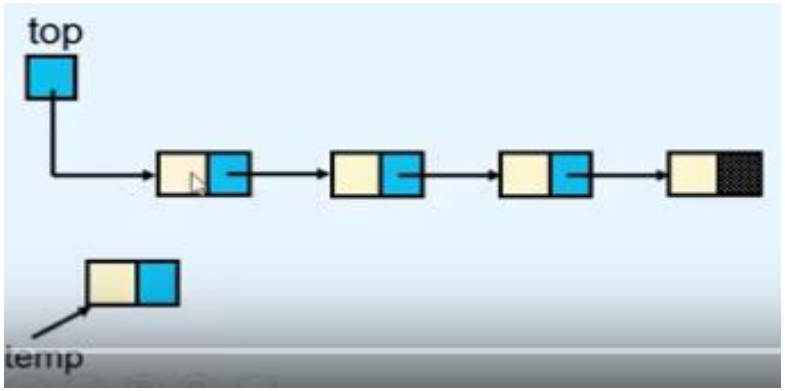
6. Abstract DisplayStack(StackType stack)

//This algorithm Prints all the Elements stored in stack. Temp purpose?

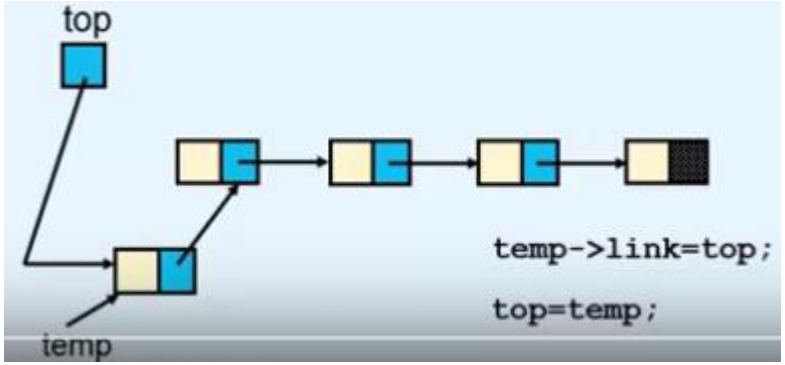
```
{ if Top==NULL
        Print “Error Message”
 Else {createNode(Temp)
        Temp=Top;
        While(Temp!=Null)
                Print(Temp->Data);
                Temp= Temp->next;
        }
}
```

PUSH Operation on Stack



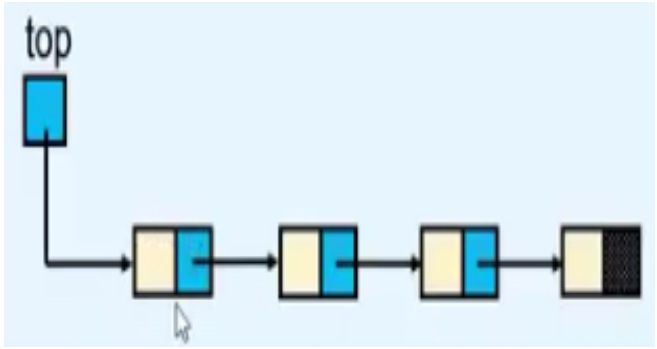


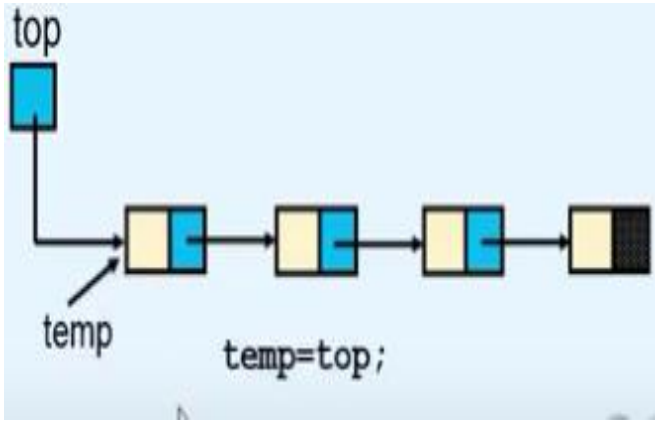
tmp = (struct node *)malloc(sizeof(struct node));

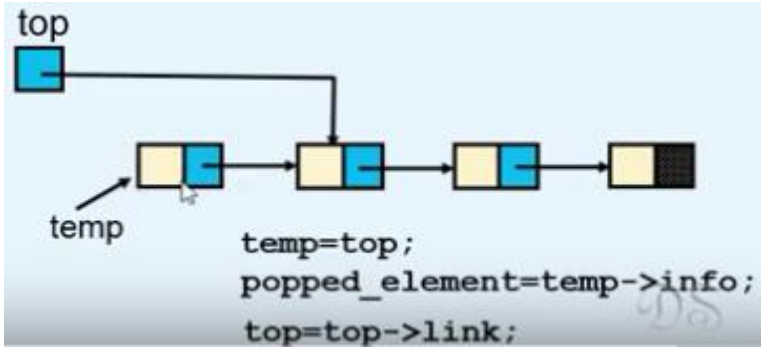


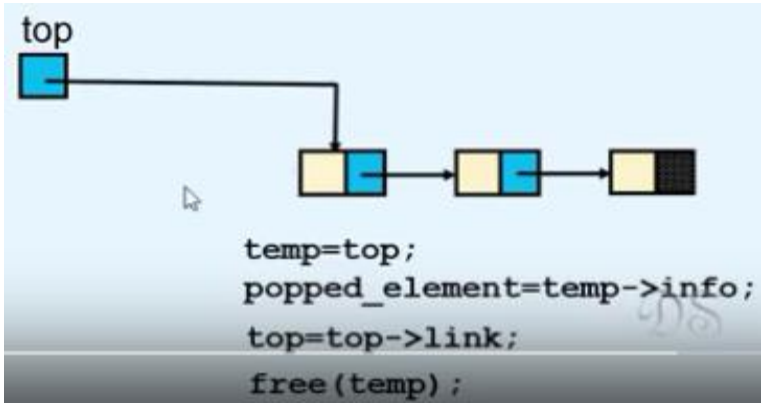
tmp->link=top;
top=tmp;

POP Operation on Stack









Application of Stack

Parentheses Matching Algorithm

Algorithm Boolean ParenMatch(*X*,*n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
    if *X*[*i*] is an opening grouping symbol **then**
        S.push(*X*[*i*])
    else if *X*[*i*] is a closing grouping symbol **then**
        if *S*.isEmpty() **then**
            return false *{nothing to match with}*
        if *S*.pop() does not match the type of *X*[*i*] **then**
            return false *{wrong type}*
if *S*.isEmpty() **then**
    return true *{every symbol matched}*
else
    return false *{some symbols were never matched}*

Parentheses Matching Algorithm

i/p string= {()()}

step 1	Step 2	step 3	Step 4	step 5	step 6
i/p= {, push	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= }, pop; ToS= {, match= true
{	({	{	({	{	

After step 6, stack is empty. So given string of parenthesis is balanced

Parentheses Matching Algorithm

i/p string= {(){()}

step 1	Step 2	step 3	Step 4	step 5	step 6	step 7	step 8
i/p= {, push	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= {, push	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= }, pop; ToS=}, match= true	i/p= }, pop; underflow; Error
				(			
	(		{	{	{		
{	{	{	{	{	{	{	

After step 8, stack is nonempty but there are more characters in input string. So given string of parenthesis is not balanced

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
  - simply prefix notation,

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*+

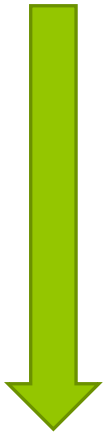
Polish Notation

- 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**



Manual Conversion -Infix to Postfix

- A+ B * C Given infix form
- Left to right scan, Brackets first then Operator Precedence
- A + B C * 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+[(BC+)+(D+E)*F]/G
- A+[(BC+)+(DE+)*F]/G
- A+[(BC+)+(DE+)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 +**

=AB+ +C +D

=AB+C+ +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

Infix to postfix process with parenthesis

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Scan X from left to right and repeat Step 2 to 5 for each element of X until the Stack is empty.
2. If an operand is encountered, add it to Y
3. If a left parenthesis is encountered, push it onto Stack.
4. If an operator is encountered ,then:
   - Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator until an opening parenthesis is encountered.
   - Add operator to Stack.
5. If a right parenthesis is encountered ,then:
   - Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
   - Remove the left Parenthesis.

2. END.

Input: input expression:(((A + B) * C) - ((D - E) * (F + G)))

Input char	stack	Output
(	(	
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	AB
)	((	AB+
*	((*	AB+
C	((*	AB+C
)	(	AB+C*
-	(-	AB+C*
(	(-(	AB+C*
(	(-((	AB+C*
D	(-((	AB+C*D
-	(-((-	AB+C*D
E	(-((-	AB+C*DE
)	(-(	AB+C*DE-
*	(-(*	AB+C*DE-
(	(-(*(	AB+C*DE-
F	(-(*(	AB+C*DE-F
+	(-(*(+	AB+C*DE-F
G	(-(*(+	AB+C*DE-FG
)	(-(*	AB+C*DE-FG+
)	(-	AB+C*DE-FG+*
)	EMPTY	AB+C*DE-FG+*-

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)

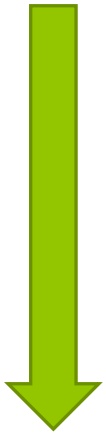
$(P/(Q-R)*S+T)$

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+

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-(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

Evaluation of postfix expression

- Create a stack for storing operands
- Scan the input expression from left to right
  - If the element is operand, push it onto the stack
  - If the element is operator, pop two operands, evaluate and push the result onto the stack
- If the expression is over, the stack contains the final answer

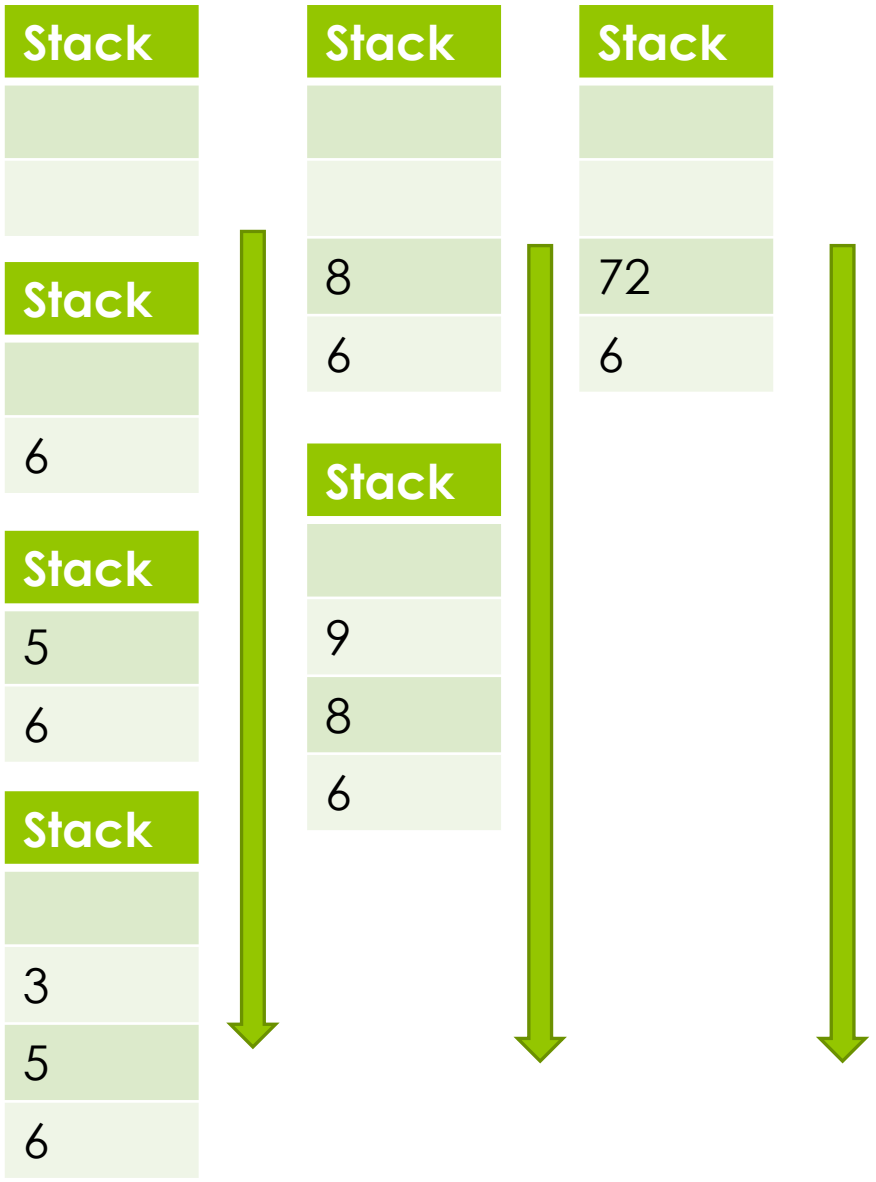
Input: input expression: AB+C*DE-FG+*-

e.g. A=2, B=3,C=1,D=4,E=5, F=7, G=8

Input char	stack
2	2
3	2, 3
+	(2+3)=5
1	5, 1
*	(5*1)= 5
4	5,4
5	5,4,5
-	5,-1
7	5,-1,7
8	5,-1,7,8
+	5,-1,15
*	5,-15
-	20

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



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 +

(a) Input so far (shaded):
5 3 2 * + 4 - 5 +

(b) Input so far (shaded):
5 3 2 * + 4 - 5 +

(c) Input so far (shaded):
5 3 2 * + 4 - 5 +

(d) Input so far (shaded):
5 3 2 * + 4 - 5 +

(e) Input so far (shaded):
5 3 2 * + 4 - 5 +

(f) Input so far (shaded):
5 3 2 * + 4 - 5 +

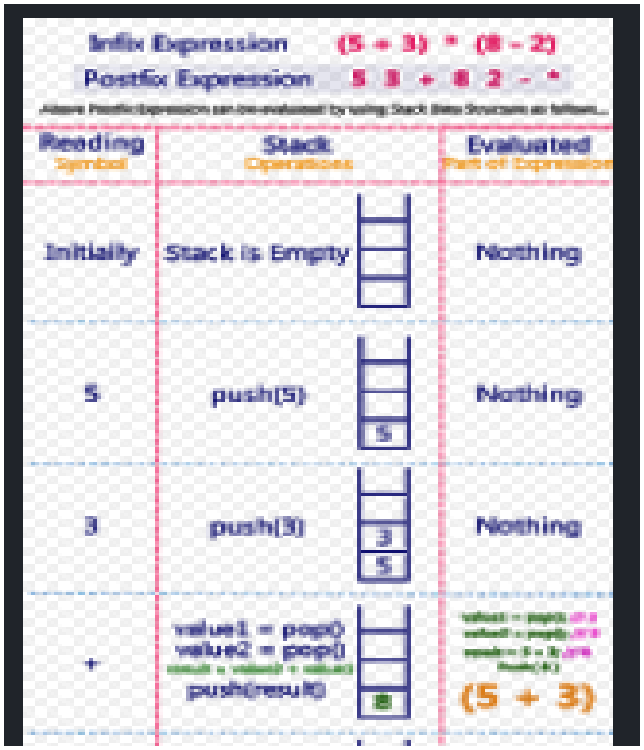
(g) Input so far (shaded):
5 3 2 * + 4 - 5 +

(h) Input so far (shaded):
5 3 2 * + 4 - 5 +

(i) Input so far (shaded):
5 3 2 * + 4 - 5 +

The result of the computation is 12.

Postfix Expression 5 3 + 8 2 - *



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

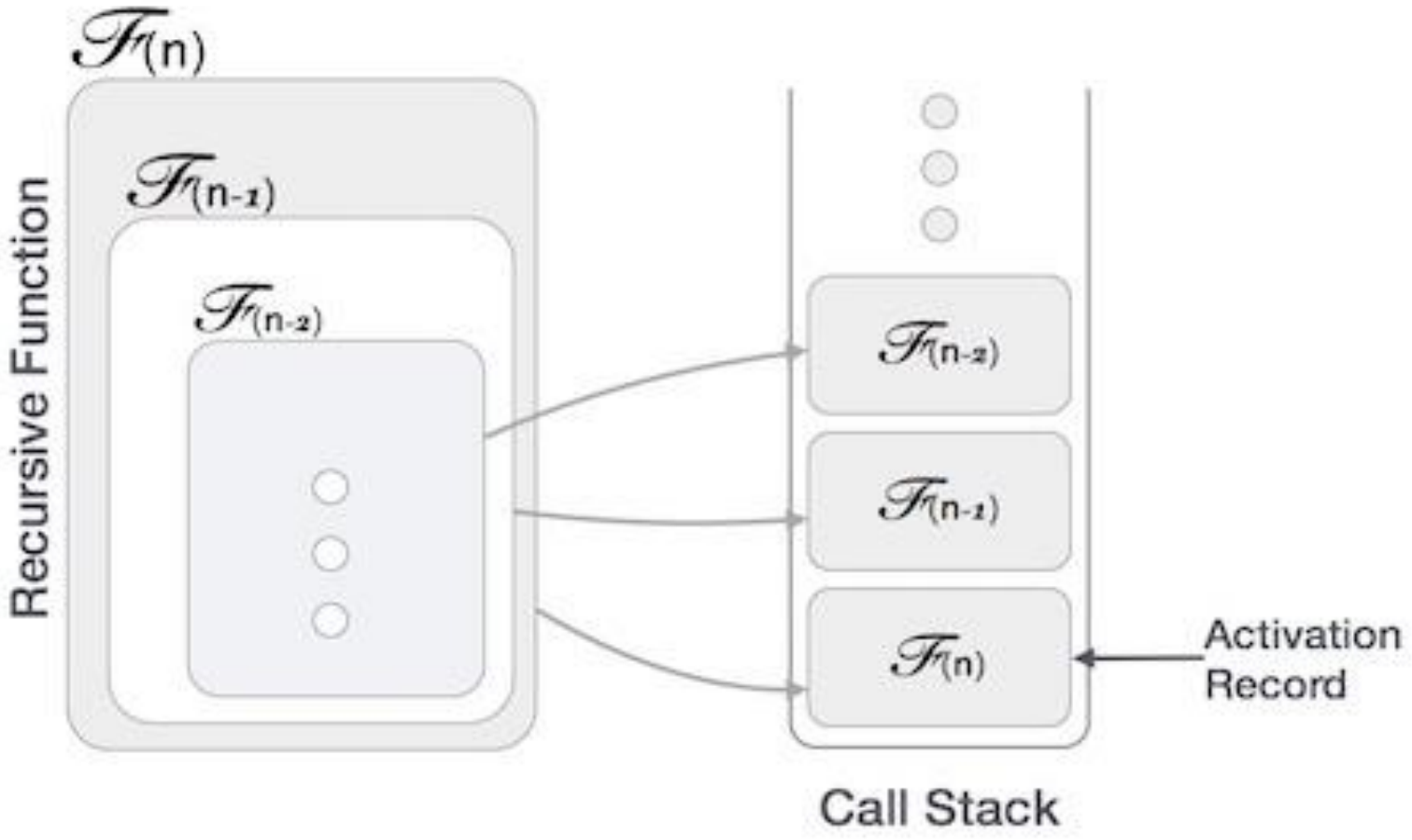
Evaluate 562+*(12)4/- taking 12 as a single number

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

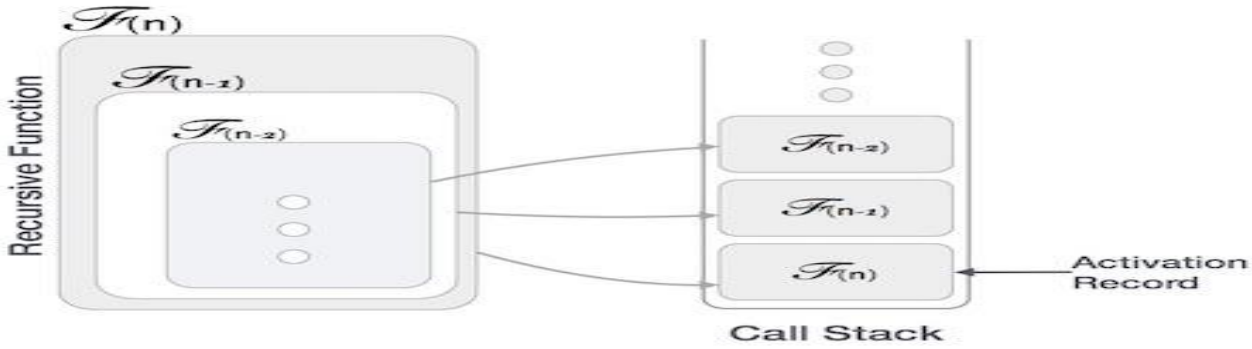
Application of Stack –Recursion



Application of Stack –Recursion

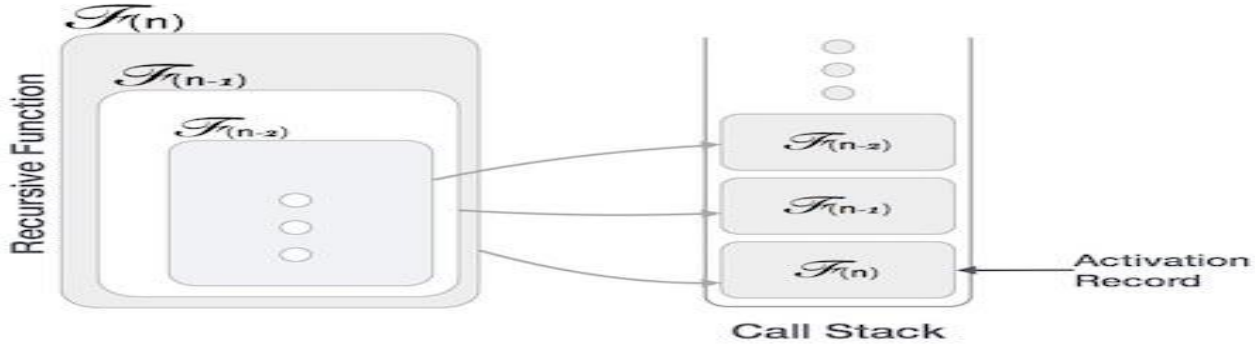
- Many programming languages implement recursion by means of **stacks**.

- 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.



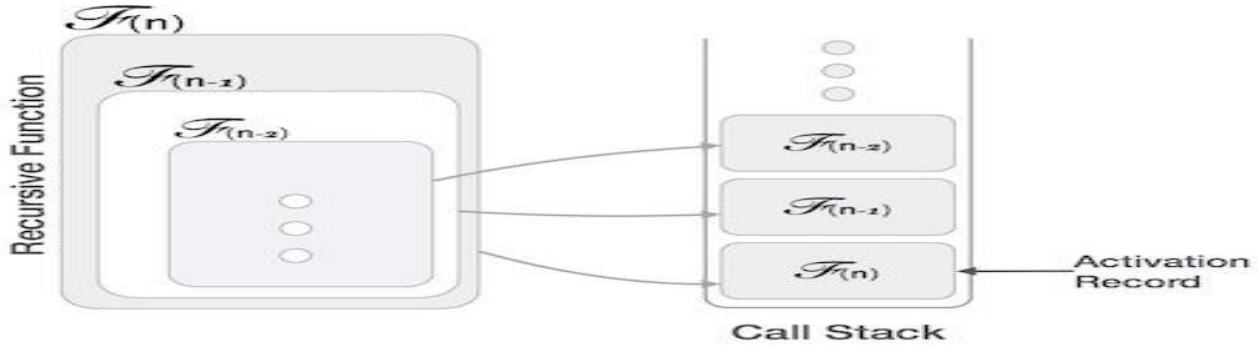
Application of Stack –Recursion

- 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.



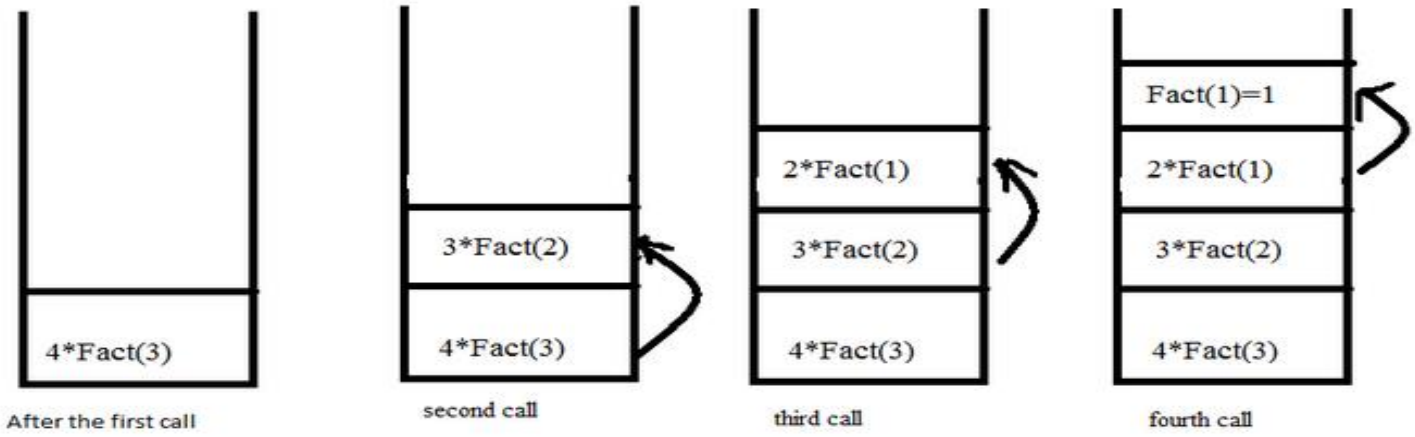
Application of Stack –Recursion

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

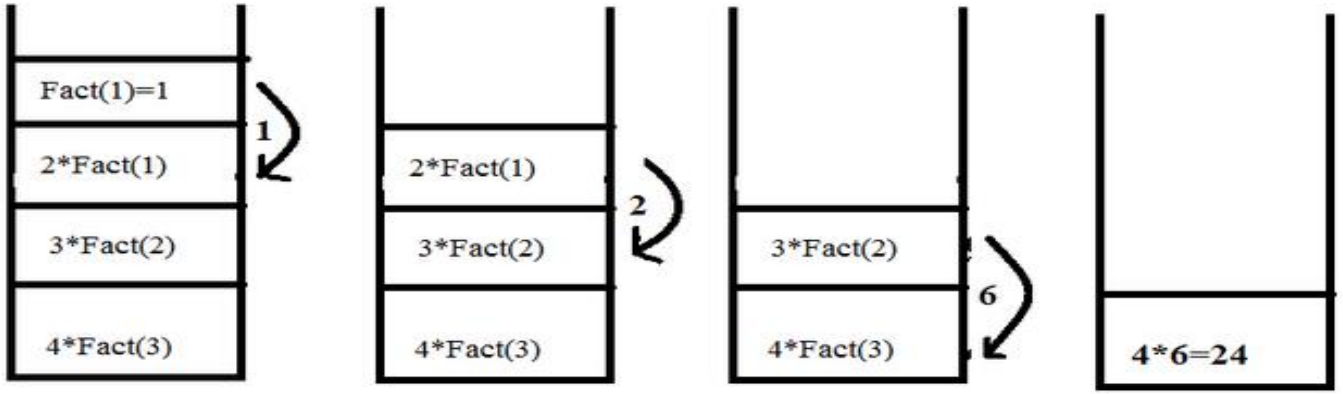
```
int factorial(n)
{
        if (n == 1)
        {
                // base case
                return 1;
        }
        else
        {
                return n * factorial(n - 1);
                // function calls itself
        }
}
```

Application of Stack –Recursion

When function call happens previous variables gets stored in stack



Returning values from base case to caller function



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.