Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Fundamentals of Data Structures

S. Y. B. Tech CSE

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

Syllabus

Unit 5 :

Stacks: Stack as an Abstract Data Type, Representation of Stack Using Sequential Organization, stack operations, Multiple Stacks Applications of Stack- Expression Conversion and Evaluation, Linked Stack and Operations, Recursion.

Queues: Queue as Abstract Data Type, Representation of Queue Using Sequential Organization, Queue Operations Circular Queue, Advantages of Circular queues, Linked Queue and Operations Deque-Basic concept, types (Input restricted and Output restricted), Application of Queue: Job scheduling.

Stack

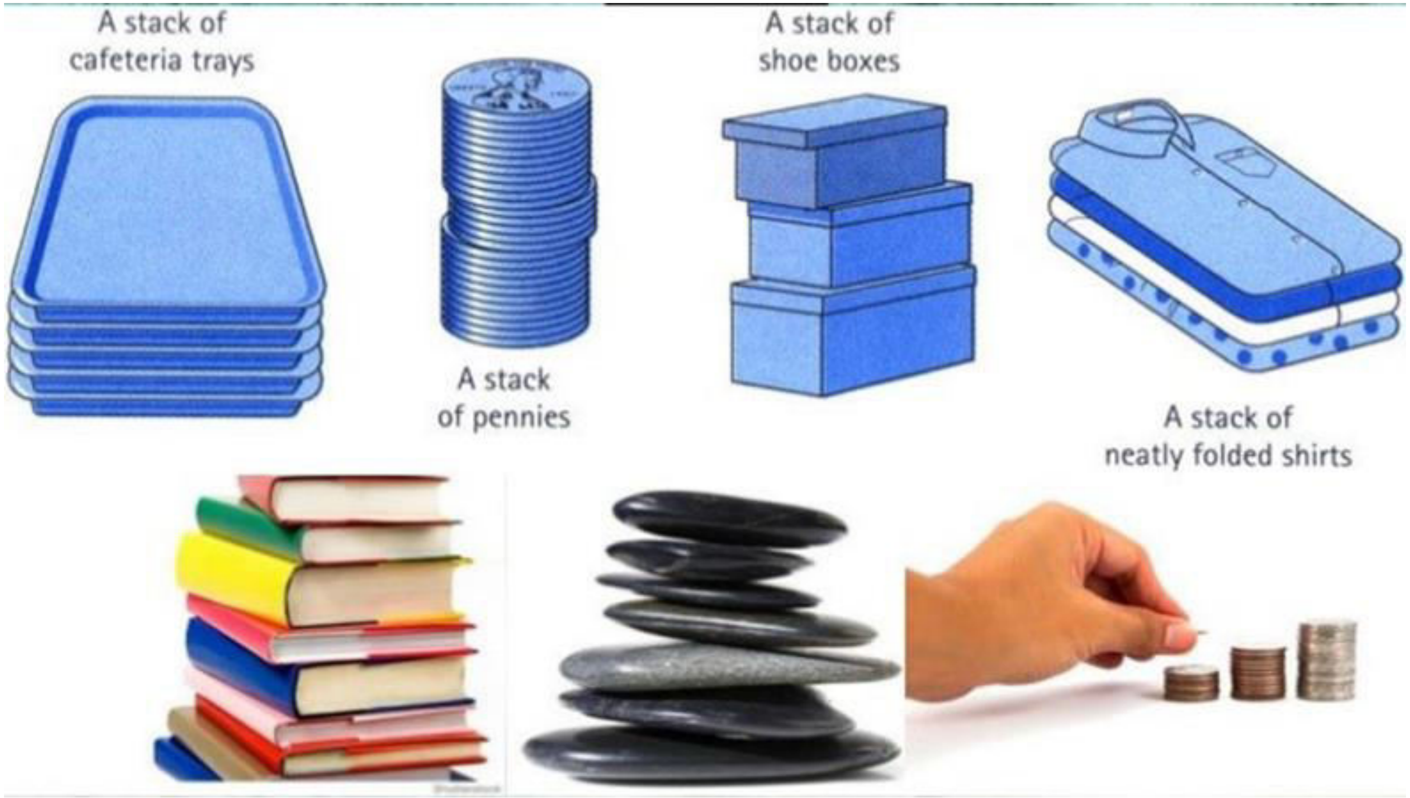
Topics to be Covered

- Stack as an Abstract Data Type
- Representation of Stack Using Sequential Organization
- Applications of Stack- Expression Conversion and Evaluation
- Recursion

Unit-V

Stacks

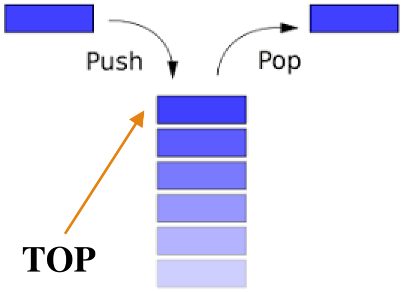
Real life Applications of Stack



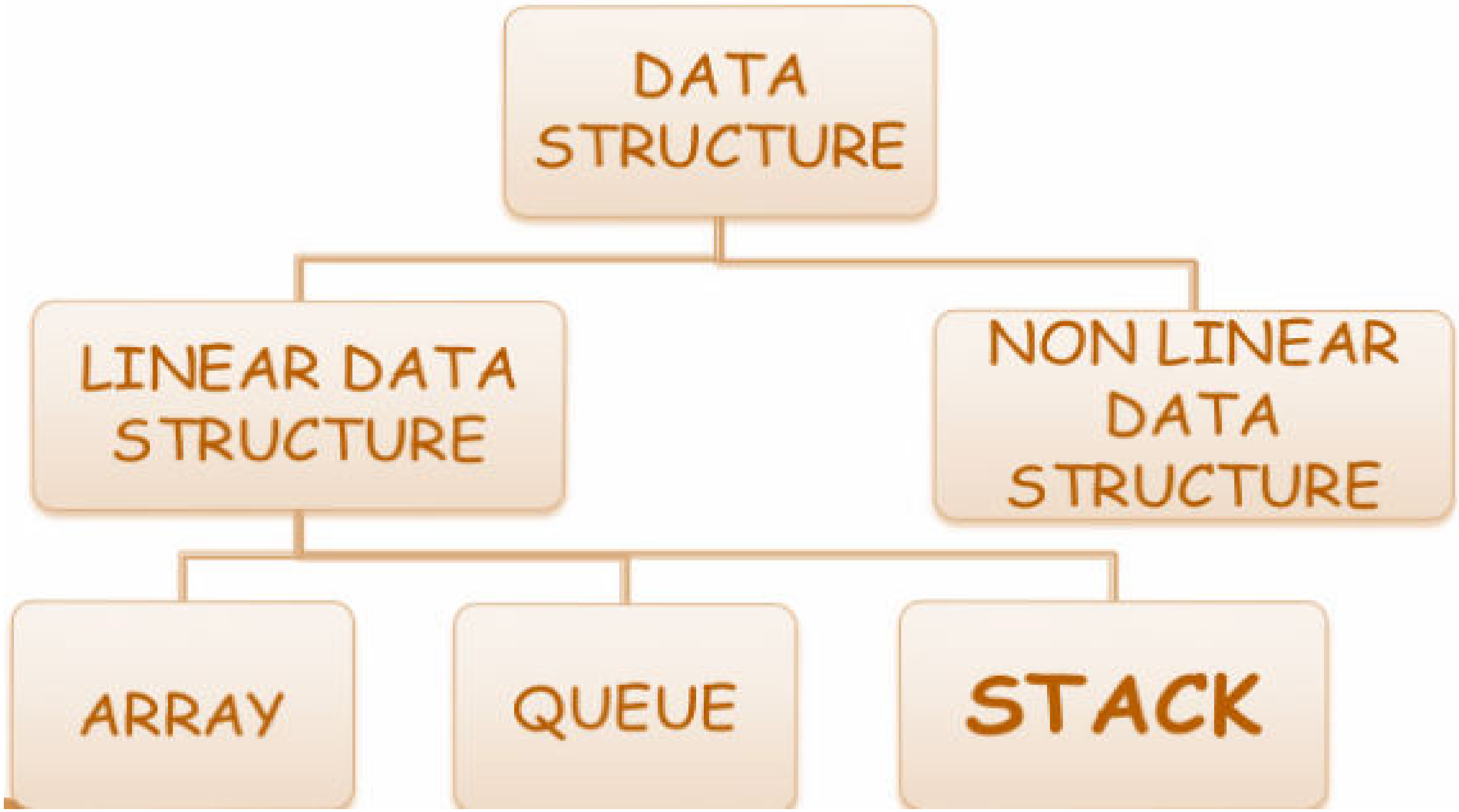
Stack

- Stack : Special case of ordered list also called as restricted/controlled list where insertion and deletion happens at only one end called as top of stack (homogeneous collection of elements.)
- Elements are added to and removed from the top of the stack *(the most recently added items are at the top of the stack).*
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).
- Only access to the stack is the top element
  - consider trays in a cafeteria

    --to get the bottom tray out, you must first remove all of the elements above



Where stack resides in data structure family?



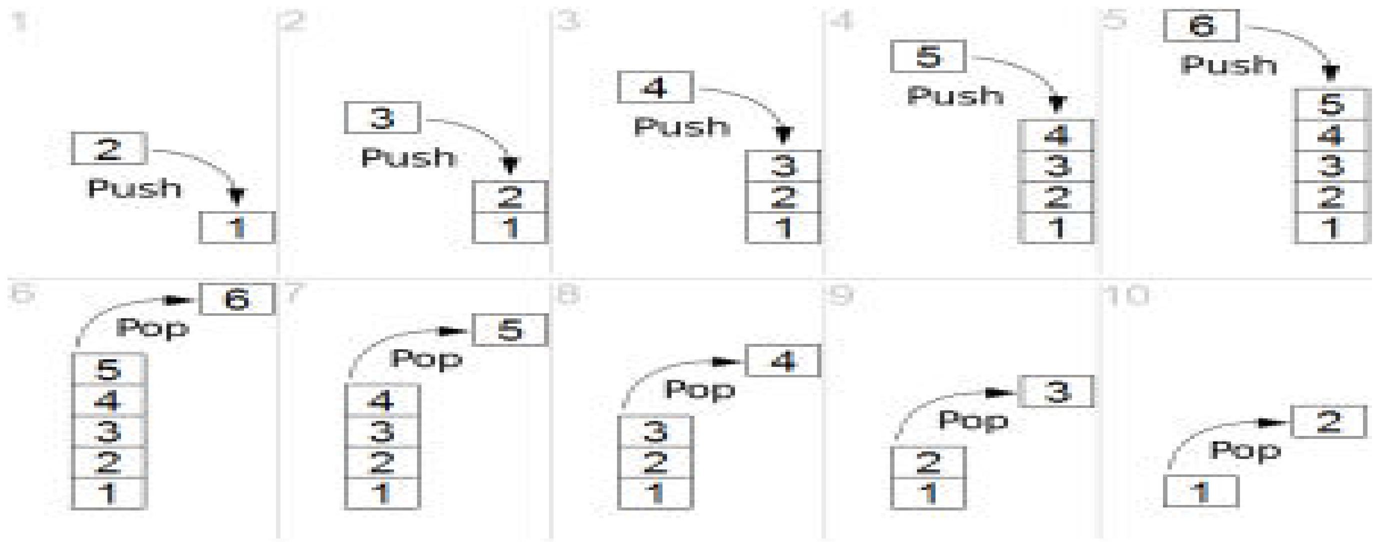
Stack ADT

```
structure STACK (item)
  declare CREATE() -> stack
              ADD(item, stack) -> stack
              DELETE(stack) -> stack
              TOP(stack) -> item
              ISEMPS(stack) -> boolean;
  for all S ∈ stack,  i ∈ item let
          ISEMPS(CREATE) ::= true
          ISEMPS(ADD(i,S)) ::= false
          DELETE(CREATE) ::= error
          DELETE(ADD(i,S)) ::= S
          TOP(CREATE)
          TOP(ADD(i,S)) ::=  i
  end
end STACK
```

Basic Stack Operations

Operations-

- ➢ isEmpty()-Checking stack is empty
- ➢ IsFull()-Checking stack is full
- ➢ push()-Pushing Element on top of stack
- ➢ pop() – Popping top element from stack



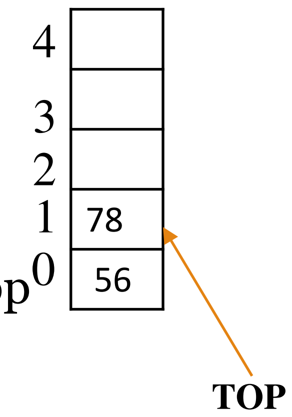
Representation of stack

Stack can be represented(implemented) using two data structures

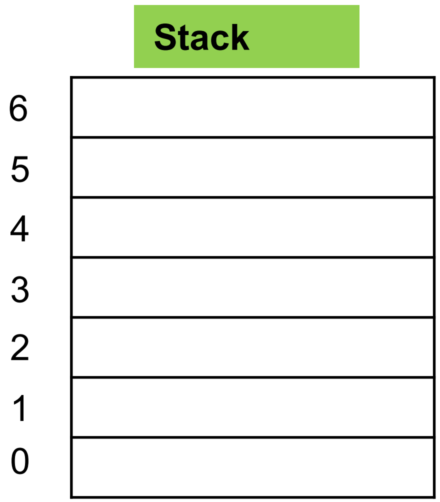
- **Array (Sequential Organization)**
- **Linked List (Linked Organization)**

Representation of Stack using Sequential Organization

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Index of bottom most element of stack is 0
- Increment *top* when one element is pushed, decrement after pop
- Index of the most recently added item is given by *top*



Stack Operations using Array : Example



Stack Max Size=07

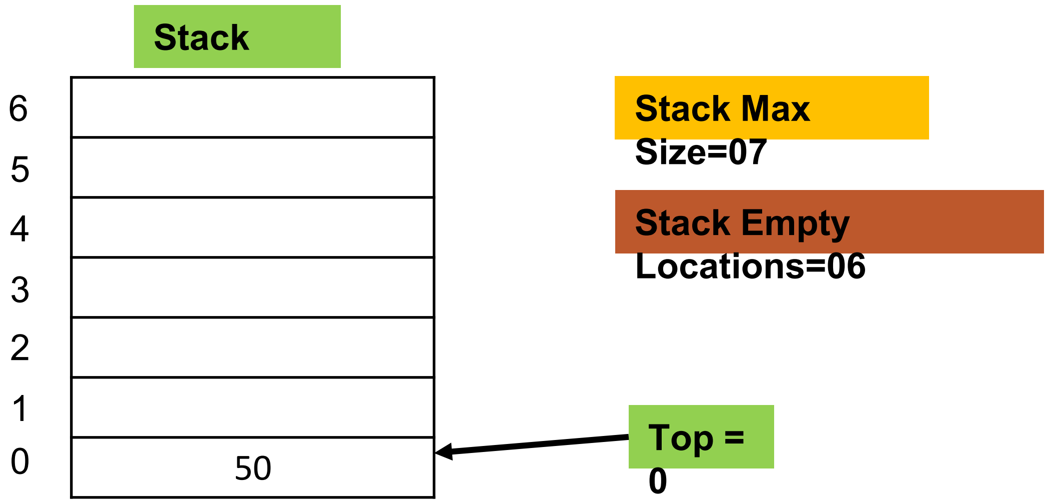
Stack Empty Locations=07

Top = -1

Stack Empty

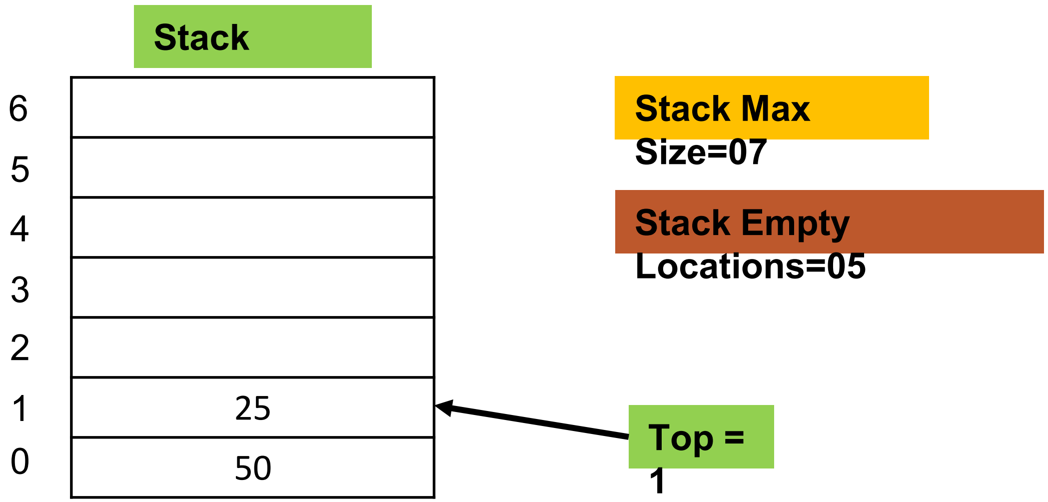
Empty Stack

Stack Operations using Array : Example



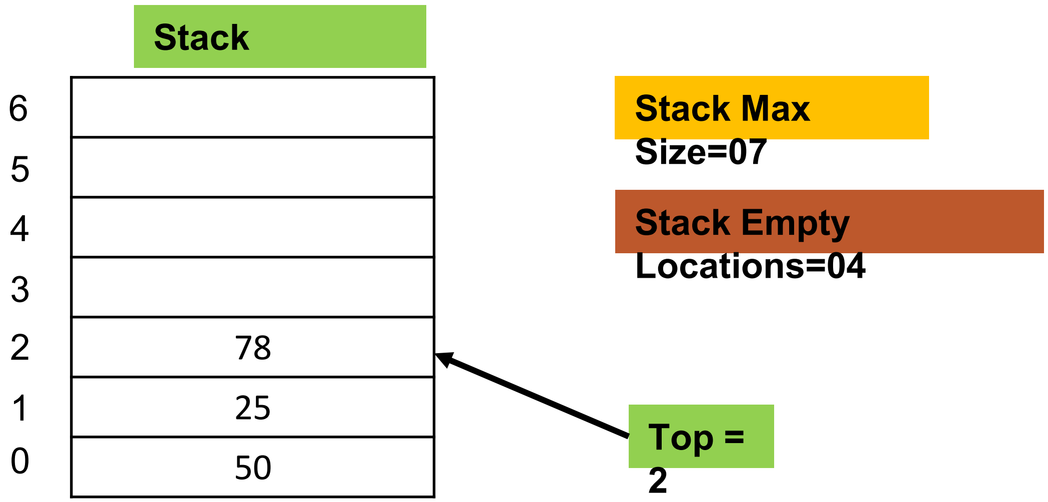
Push (50)

Stack Operations using Array : Example



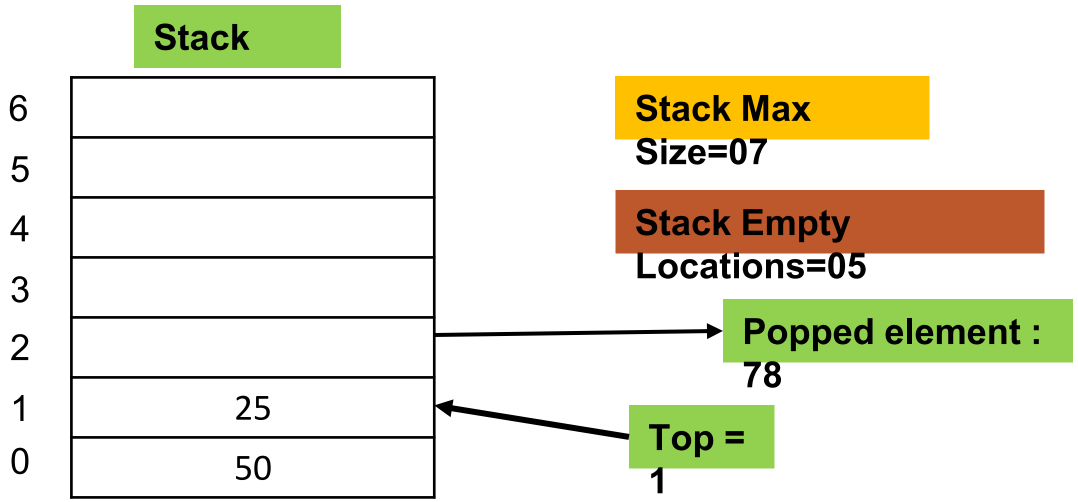
Push (25)

Stack Operations using Array : Example



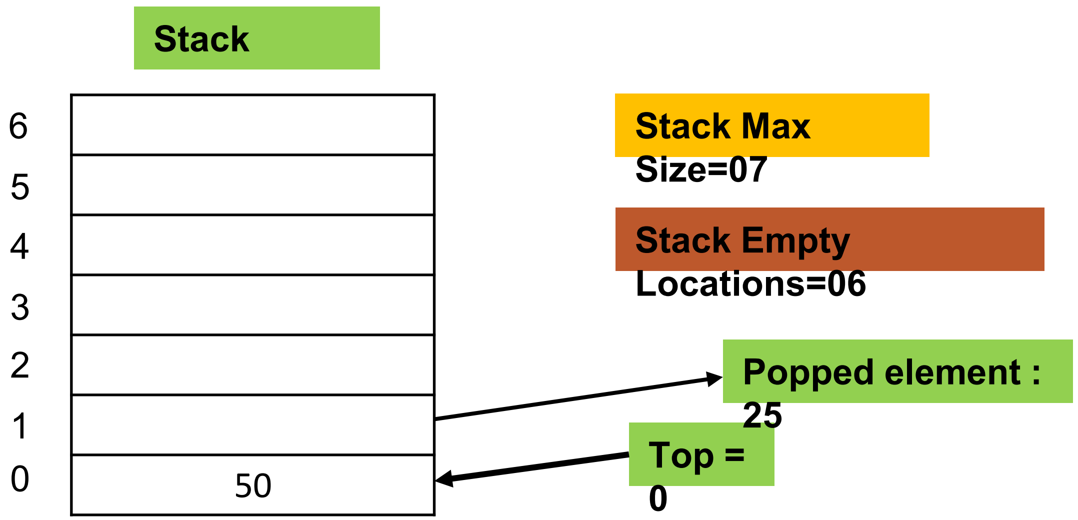
Push (78)

Stack Operations using Array : Example



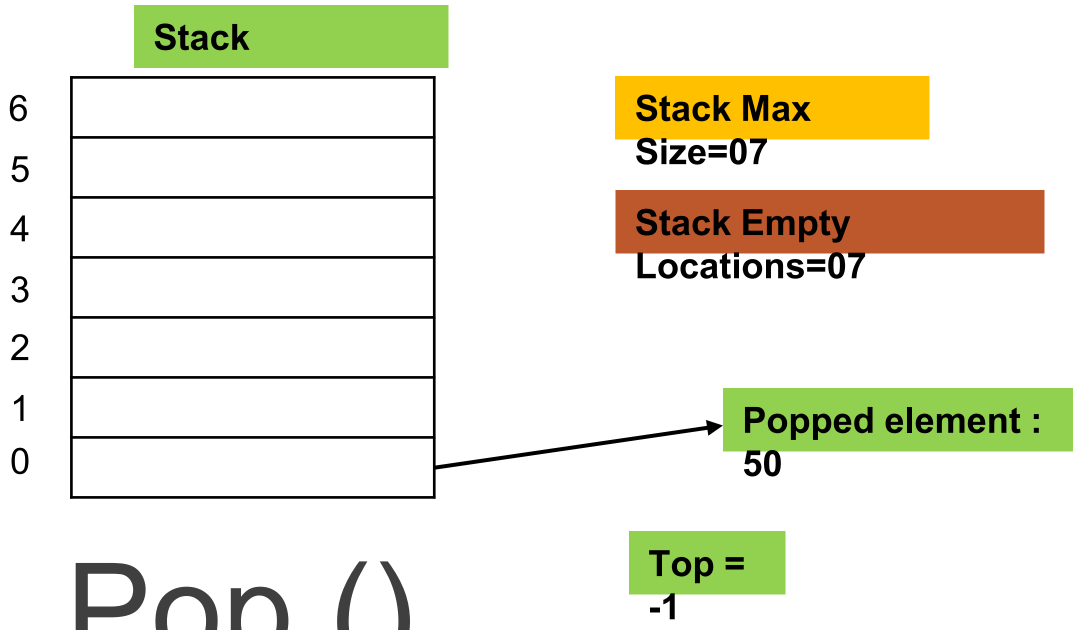
Pop ()

Stack Operations using Array : Example



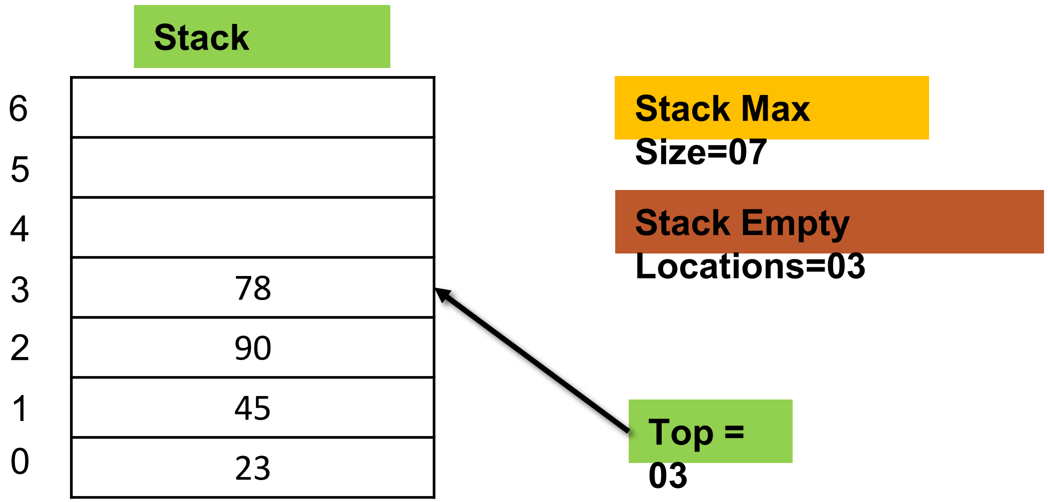
Pop ()

Stack Operations using Array : Example



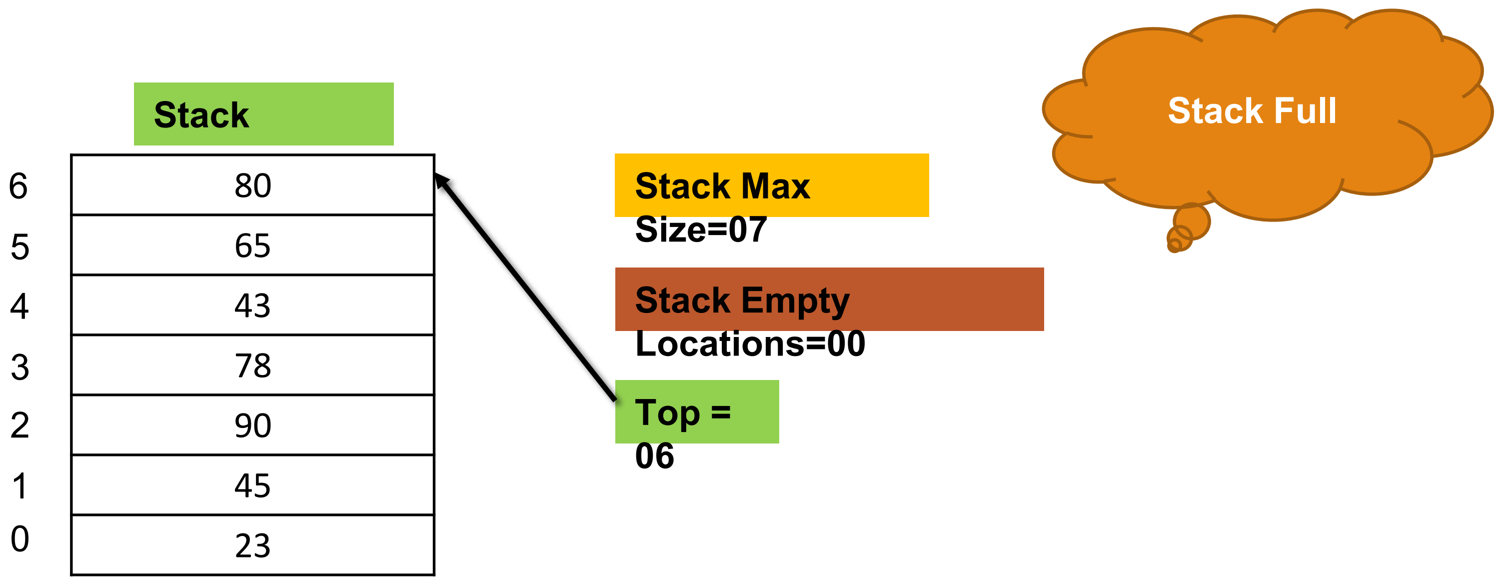
Pop ()

Stack Operations using Array : Example



Push (23),Push(45),Push(90),Push(78)

Stack Operations using Array : Example

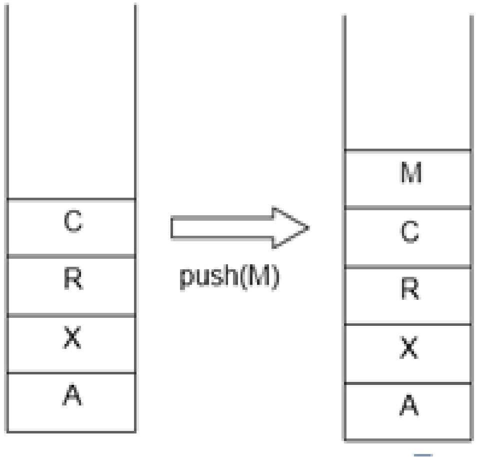


Push (43),Push(65),Push(80)

Operations on Stack (Algorithm and Pseudo Code)

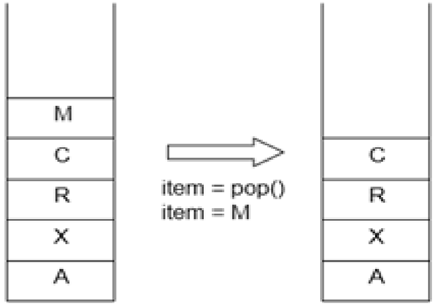
Push (ItemType newItem)

- ***Function***: Adds newItem to the top of the stack.
- ***Preconditions***: Stack has been initialized and is not full
- ***Postconditions***: newItem is at the top of the stack.



Operations on Stack (Algorithm and Pseudo Code)

Pop ()



- ***Function***: Removes top Item from stack and returns it .
- ***Preconditions***: Stack has been initialized and is not empty.
- ***Postconditions***: Top element has been removed from stack

Operations on Stack(Algorithm and Pseudo Code)

isFull()

Stack overflow

The condition resulting from trying to push an element onto a full stack.

Pseudo of isFull() function −

```
Algorithm isFull()
  {
        if (top ==MAXSIZE-1)
        return true ;
        else
        return false ;
  }
```

Operations on Stack (Algorithm and Pseudo Code)

Pseudo Code for Push

```
Algorithm push (stack, item)
  {
    if ( ! isFull ())
   {
   top=top+1
   stack[top]=item;
   }
 }
```

Operations on Stack (Algorithm and Pseudo Code)

isEmpty()

Stack underflow (check if stack is empty.)

The condition resulting from trying to pop an empty stack.

Pseudo of isEmpty() function −

```
Algorithm isEmpty( )
    {
    if (top = = -1)
    return true ;
    else
    return false ;
    }
```

Operations on Stack (Algorithm and Pseudo Code)

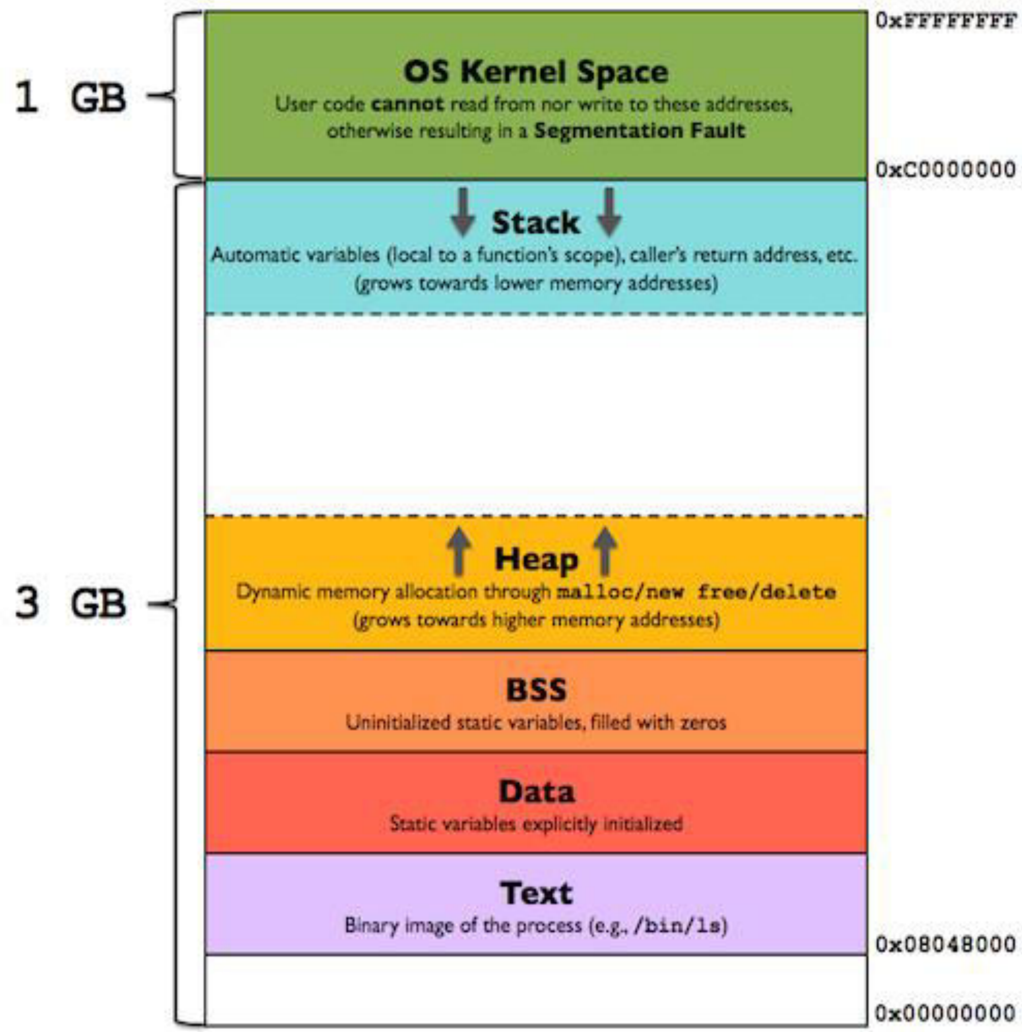
Pseudo Code for Pop

```
Algorithm  pop ()
  {
    if ( ! isEmpty())
   {
   temp=stack[top];
   top=top-1;
   return (temp);
   }
}
```

Applications of Stacks in Computer Science

- Process Function Calls
- Recursive Functions Calls
- Converting Expressions
- Evaluating expressions
- String Reverse
- Number Conversion
- Backtracking
- Parenthesis Checking

Program/Process in memory

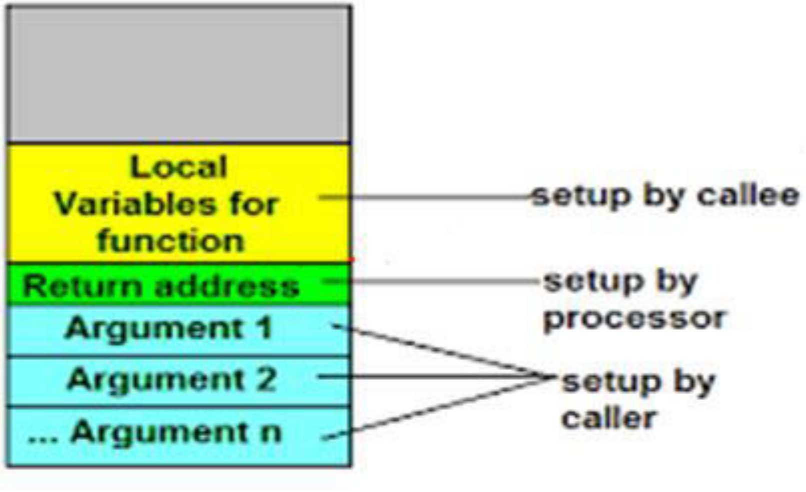


Applications of Stacks *(cont'd)*

1. **Processing Function calls:**

A stack is useful for the compiler/operating system to store local variables used inside a function block, so that they can be discarded once the control comes out of the function block.

When function execution completes, it is popped from stack



Applications of Stacks *(cont'd)*

```
main()
{
        int  a=10,b=15;
        float avg;
        avg = average(a,b,2);
        printf("%f",avg);
}
```

```
float average(int x,y,n)
{       float avg;
        int sum=add(x,y);
        avg = sum/n;
        return avg;
}
```

```
int add(int x,int  y)
{
        int sum=x+y;
        return sum;
}
```

Stack
Local variable of add
return address of add
Parameters of add
Local variables of average
return address of average
Parameters of average
Local variables
Return Address of Main()

Pushing add() Call

Pushing average() Call

Applications of Stacks *(cont'd)*

2. Recursive functions:

The stack is very much useful while implementing recursive functions.

The return values and addresses of the function will be pushed into the stack and the lastly invoked function will first return the value by popping the stack.

```
factorial(int x)
{
If(x==1)
return(1);
else
return(x* factorial(x-
1));
}
```

```
main()
{
factorial(4)
;
}
```

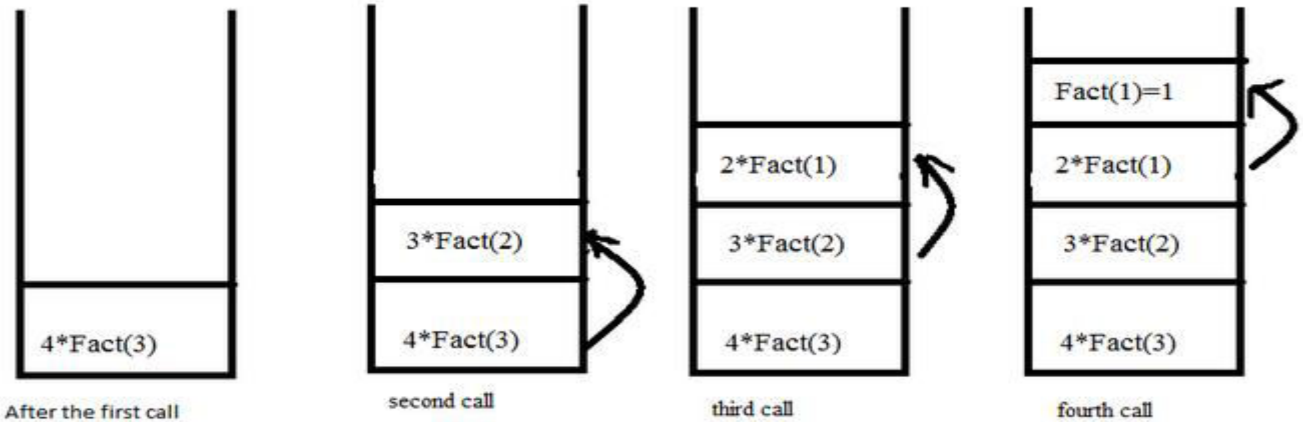
factorial(4)=
4*factorial(3);
4*3*factorial(2);
4*3*2*factorial(1);
4*3*2*1

factorial(1)
factorial(2)
factorial(3)
factorial(4)
main()

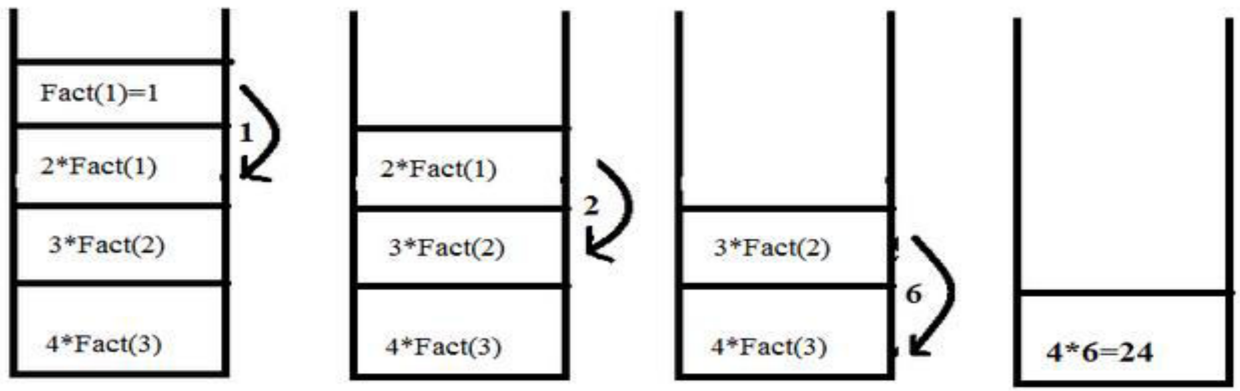
Will return value 1 and exit

Applications of Stacks *(cont'd)*

When function call happens previous variables gets stored in stack



Returning values from base case to caller function



Applications of Stacks *(cont'd)*

Expression Conversion: There are different types of Expressions

1) Infix expression :- It is the general notation used for representing expressions.

"In this expression the operator is fixed in between the operands"

Ex: a + b*c

2) Postfix fix expression :- (Reverse polish notation)

"In this expression the operator is placed after the operands".

Ex : abc*+

3) Prefix expression :- (Polish notation)

"In this expression the operators are followed by operands i.e the operators are fixed before the operands"

Ex : *+abc

All the infix expression will be converted into postfix or prefix notation with the help of stack in any program.

Expression Conversion

Why to use PREFIX and POSTFIX notations when we have simple INFIX notation?

- INFIX notations are not as simple as they seem specially while evaluating them.
- To evaluate an infix expression we need to consider Operators' Priority and Associative Property

E.g. expression 3+5*4 evaluate to 32 i.e. (3+5)*4 or to 23 i.e. 3+(5*4).

To solve this problem Precedence or Priority of the operators were defined

Expression Conversion *(cont'd)*

Operator Precedence

- Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Rank	Operator
1	^
2	* / %
3	+ - (Binary)

Expression Conversion *(cont'd)*

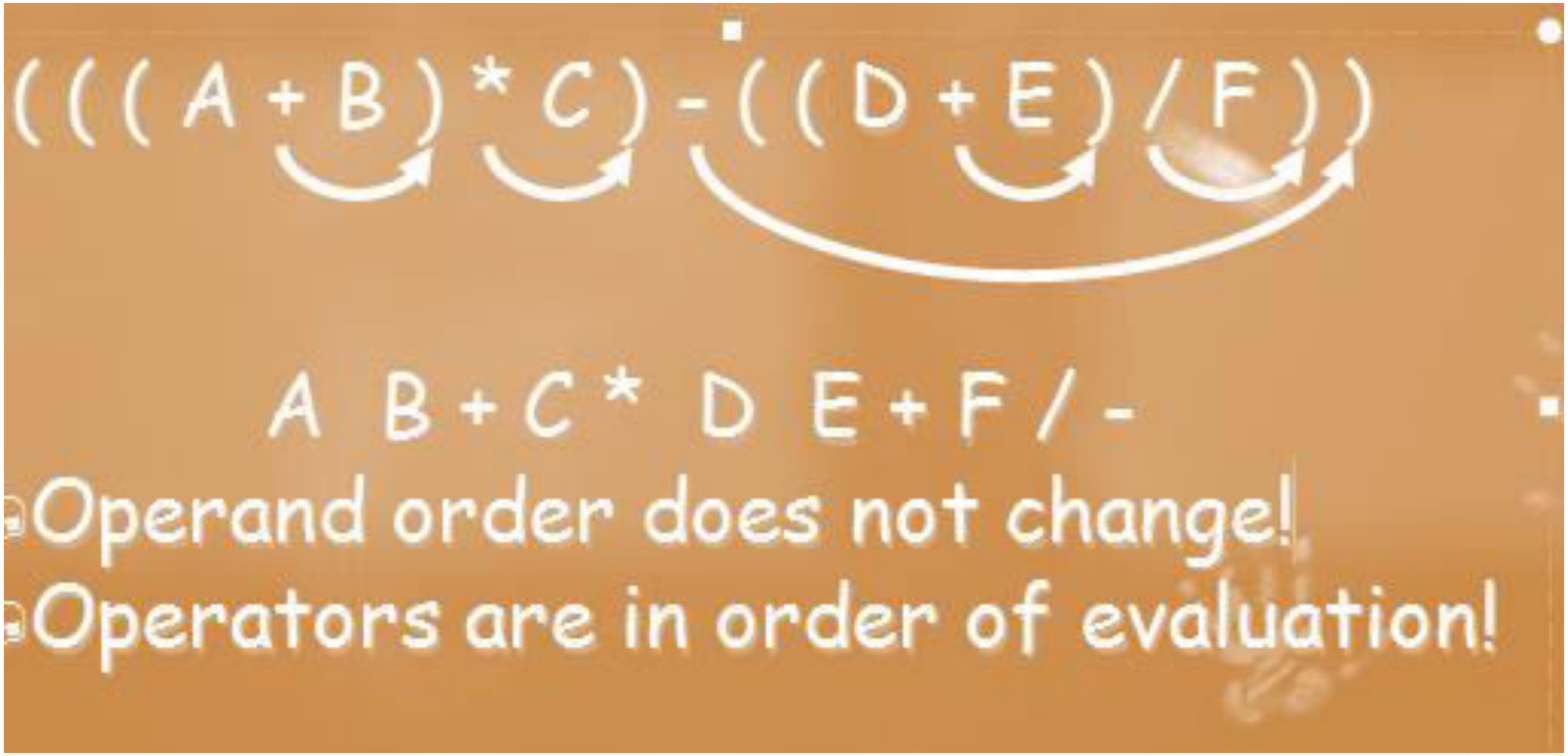
Expression Conversion Forms

Infix	Postfix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

We can convert any expression to any other two forms using Stack Data Structure

Infix to Postfix

Expression Conversion Infix to Postfix Example



Infix to Postfix

Operator Precedence (In stack and Incoming precedence)

- Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Operator	ICP	ISP
(	5	0
^	4	3
* / %	2	2
+ - (Binary)	1	1

Algorithm

Scan the symbols of the expression from left to right and for each symbol, do the following:

a. **If symbol is an operand**

- store in postfixexp that symbol onto the screen.

b. **If symbol is a left parenthesis**

- Push it on the stack.

C. **If symbol is a right parenthesis**

  - Pop all the operators from the stack upto the first left parenthesis and store in postfixexp
  - Discard the left and right parentheses.

d. **If symbol is an operator**

- **If** the precedence of the operators in the stack are greater than or equal to the current operator, then
  - Pop the operators out of the stack and store in postfixexp , and push the current operator onto the stack.

else

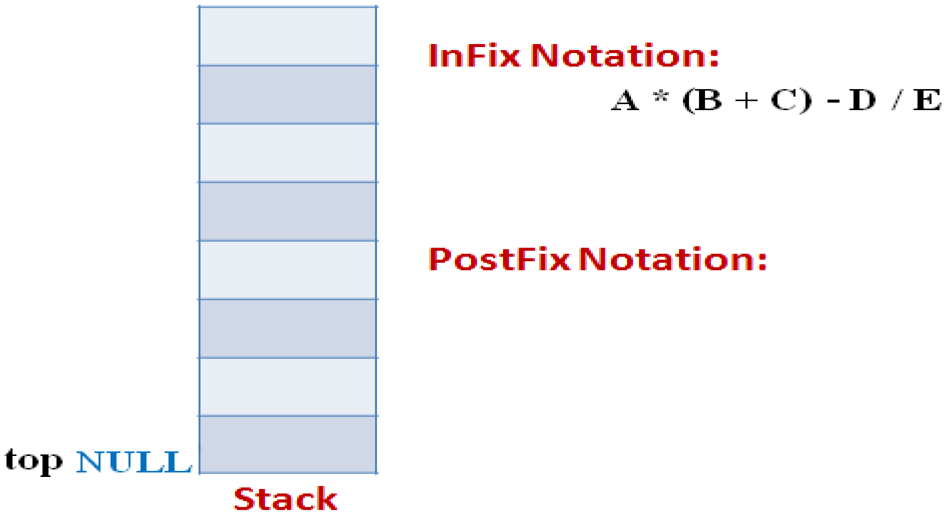
- Push the current operator onto the stack.

Infix to Postfix : Example 1

★Let the incoming the Infix expression be:

A * (B + C) – D / E

Stage 1: **Stack is empty** and we only have the Infix Expression.



Infix to Postfix : Example 1

Stage 2

★The first token is **Operand A** Operands are Appended to the Output as it is.

InFix Notation:

* (B + C) - D / E

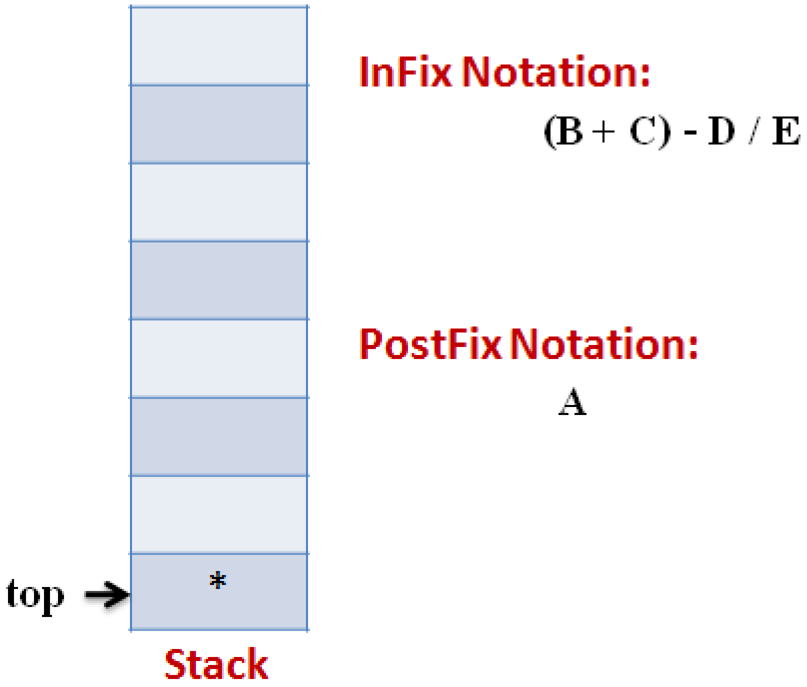
PostFix Notation:

A

Infix to Postfix : Example 1

Stage 3

★Next token is ***** Since **Stack is empty (top==-1)** it is **pushed into the Stack**

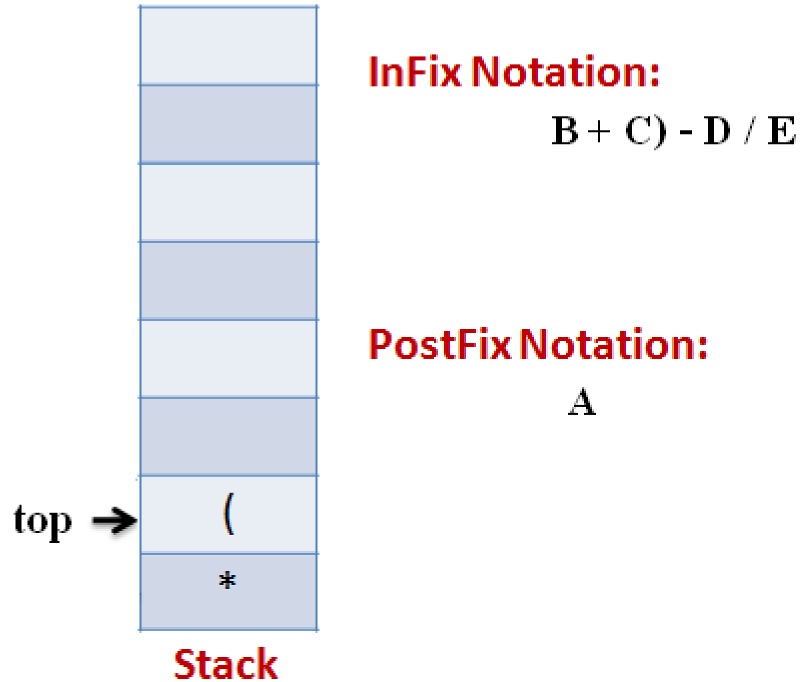


Infix to Postfix : Example 1

Stage 4

★Next token is **(** the precedence of open-parenthesis, when it is to go inside, is maximum.

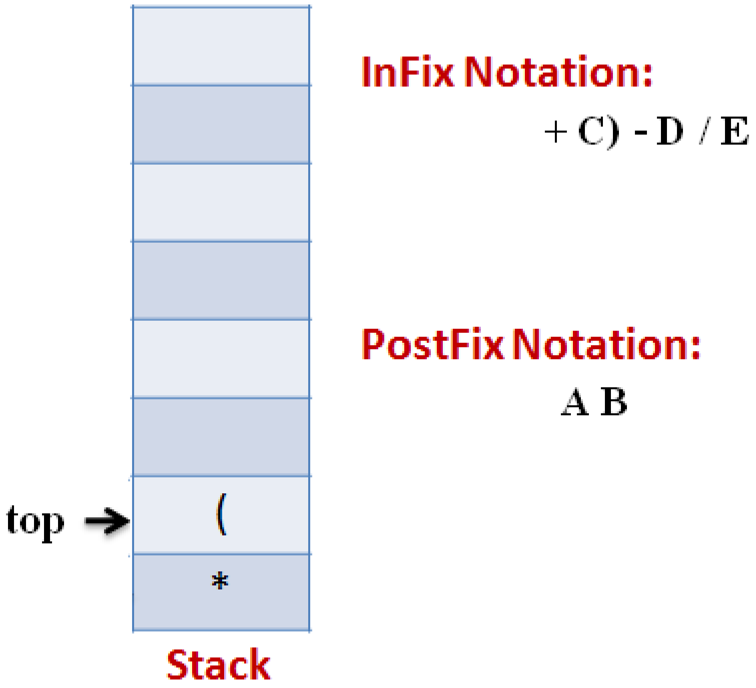
★But when another operator is to come on the top of **'('** then its precedence is least.



Infix to Postfix : Example 1

Stage 5

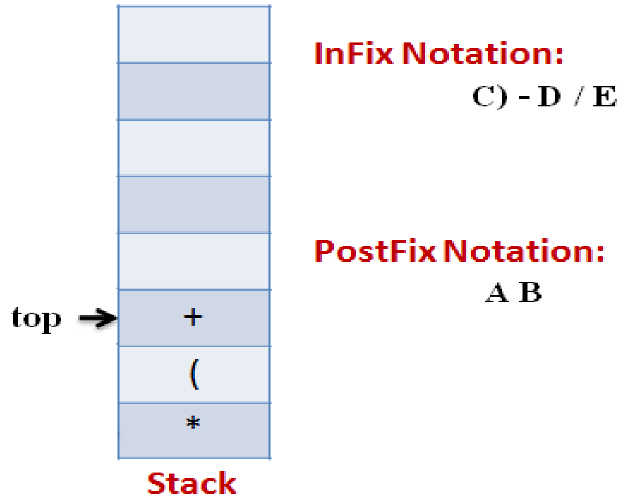
★Next token, **B** is an operand which will go to the Output expression as it is



Infix to Postfix : Example 1

Stage 6

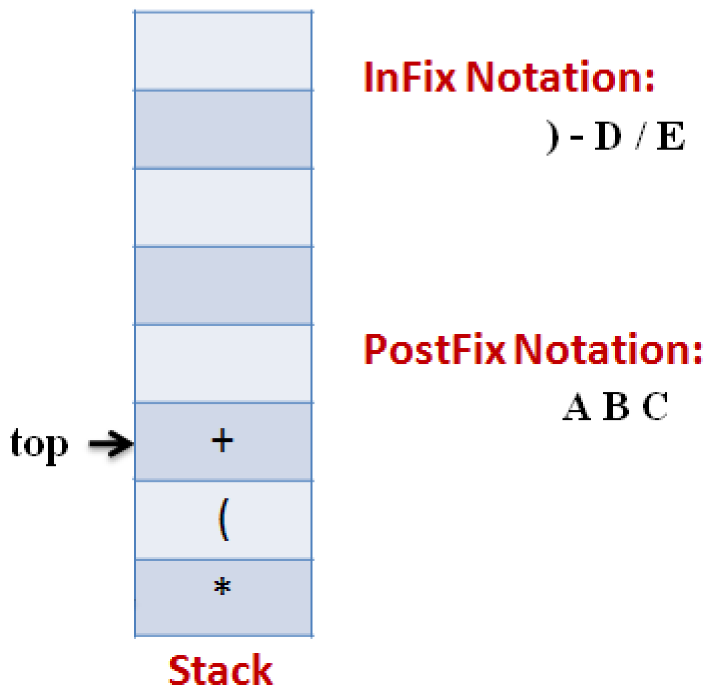
★Next token, **+** is operator, We consider the precedence of **top element in the Stack**, **'('**. The outgoing precedence of open parenthesis is the least (refer point 4. Above). So **+** gets **pushed into the Stack**



Infix to Postfix : Example 1

Stage 7

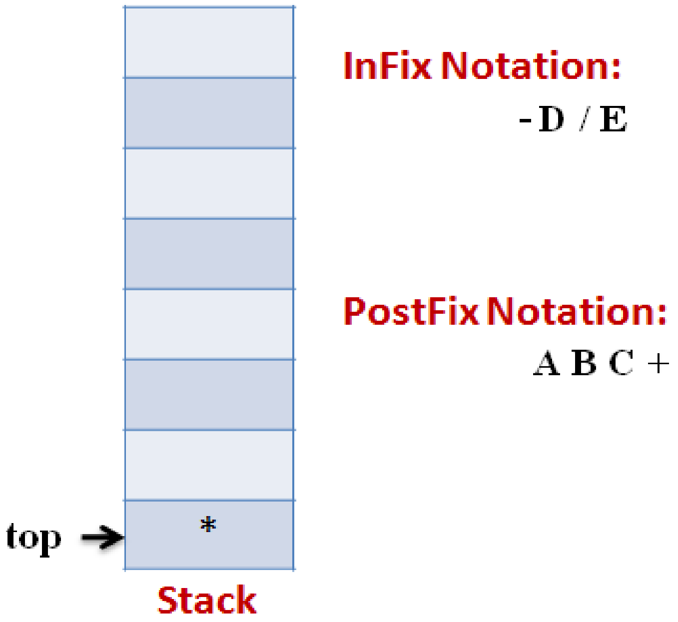
★ Next token, **C**, is appended to the output



Infix to Postfix : Example 1

Stage 8

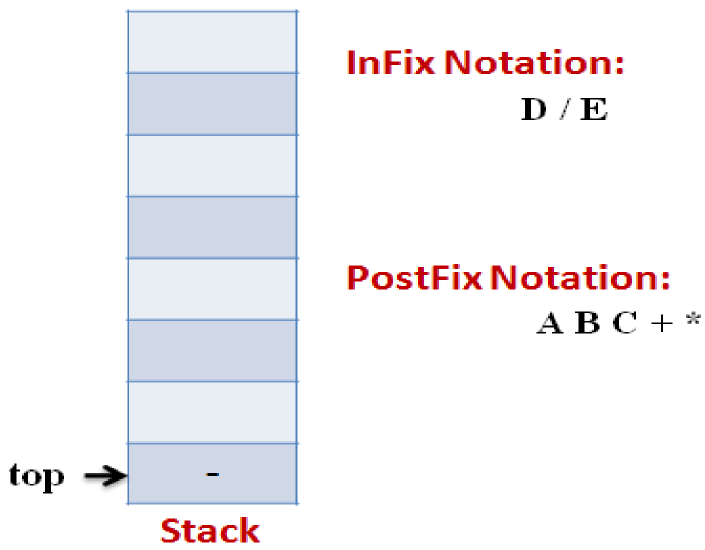
★Next token **)**, means that **pop all the elements from Stack** and **append them to the output** expression till we read an opening parenthesis.



Infix to Postfix : Example 1

Stage 9

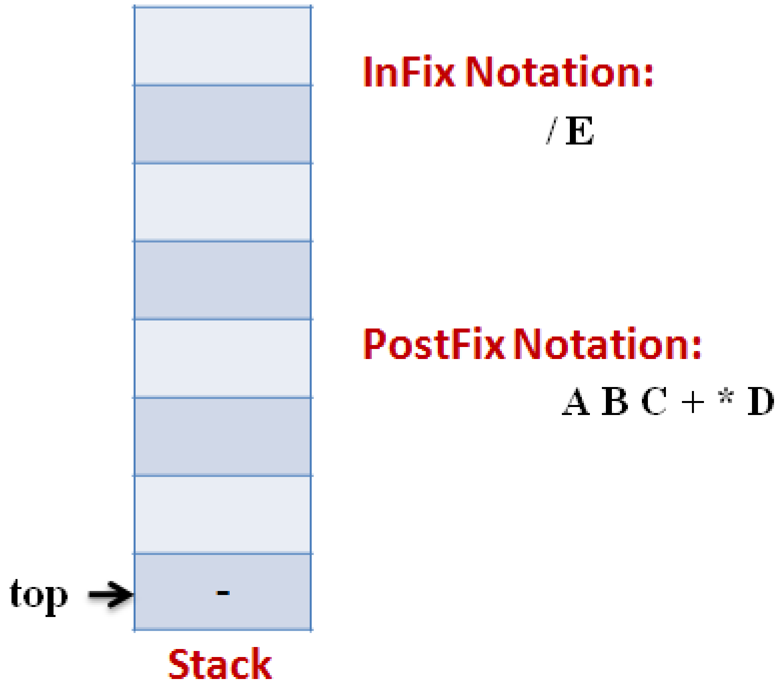
★Next token, **-**, is an operator. The precedence of operator on the top of Stack '*****' is more than that of Minus. So we **pop multiply** and **append it to output** expression. Then **push minus in the Stack**.



Infix to Postfix : Example 1

Stage 10

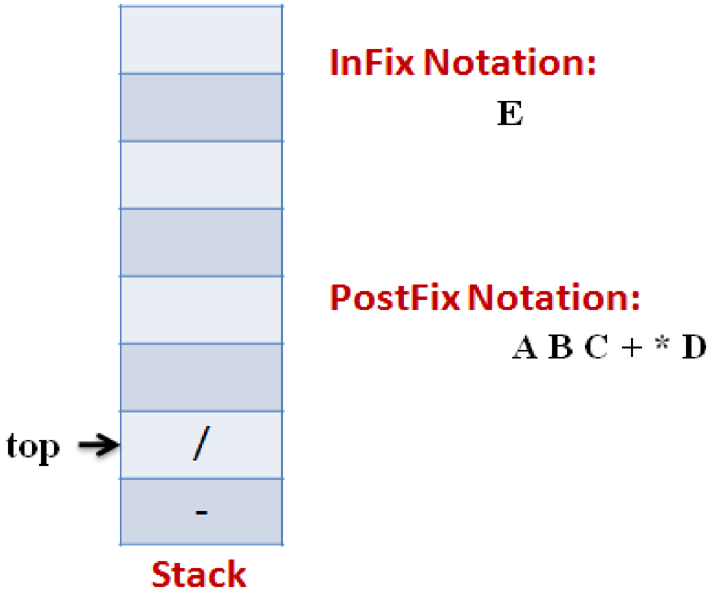
★Next, Operand ‘**D**‘ gets **appended to the output**.



Infix to Postfix : Example 1

Stage 11

★Next, we will insert the **division** operator into the Stack because its precedence is more than that of minus.

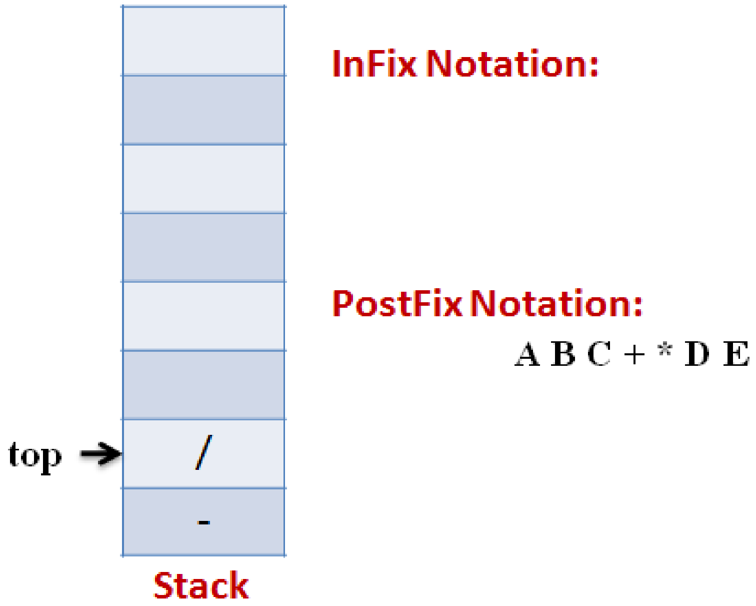


Infix to Postfix : Example 1

Stage 12

★The last token, **E**, is an operand, so we **insert it to the output**

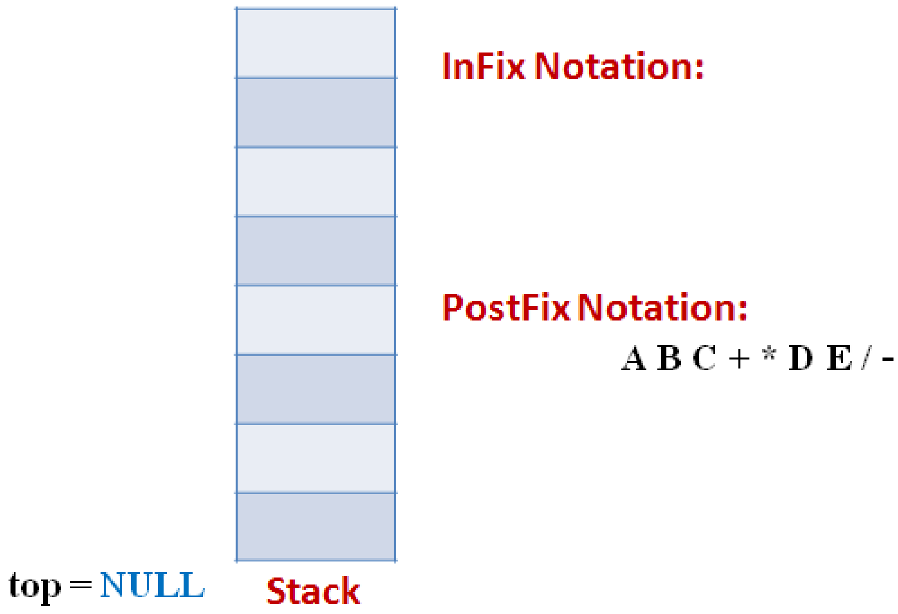
Expression as it is.



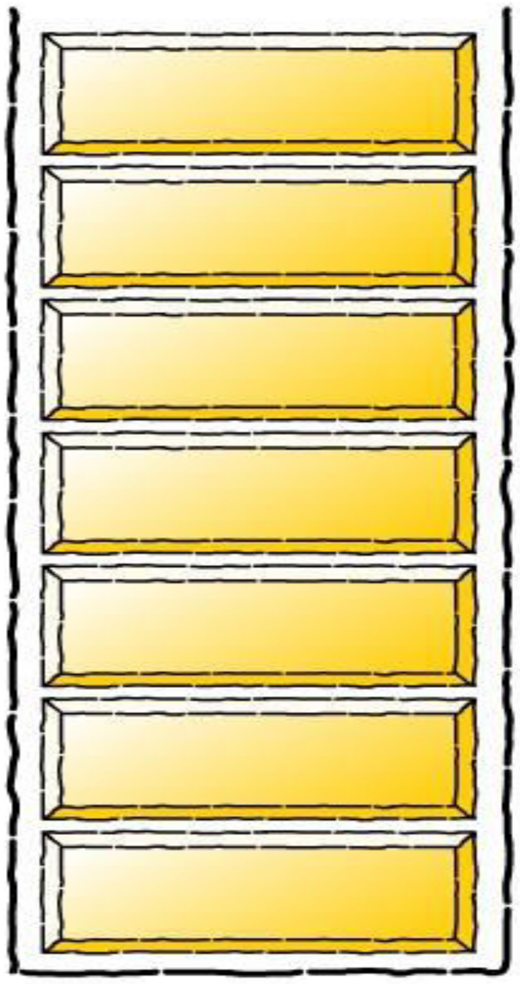
Infix to Postfix : Example 1

Stage 13

★The input Expression is complete now. So we **pop the Stack** and **Append it to the Output Expression** as we pop it.



Infix to Postfix : Example 2



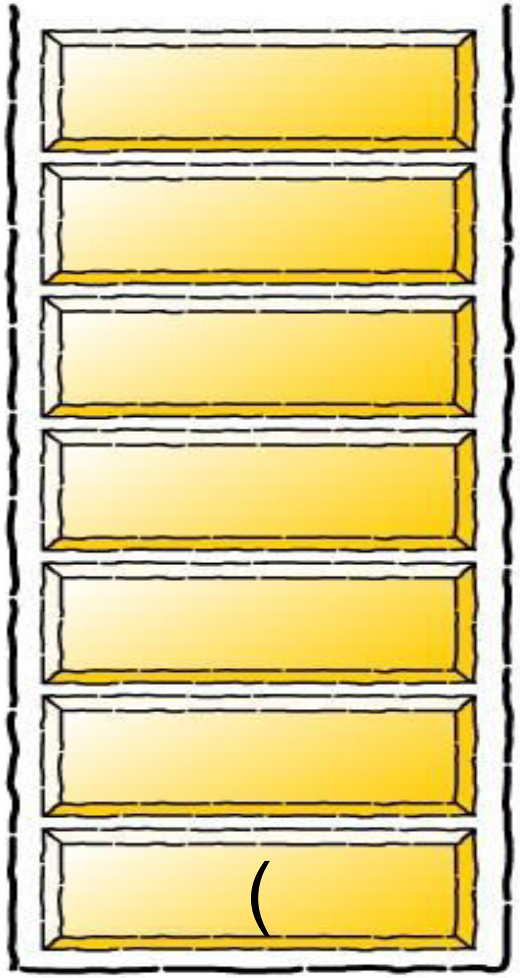
infixVect

(a + b - c) * d – (e + f)

postfixVect

Infix to Postfix : Example 2

stackVect



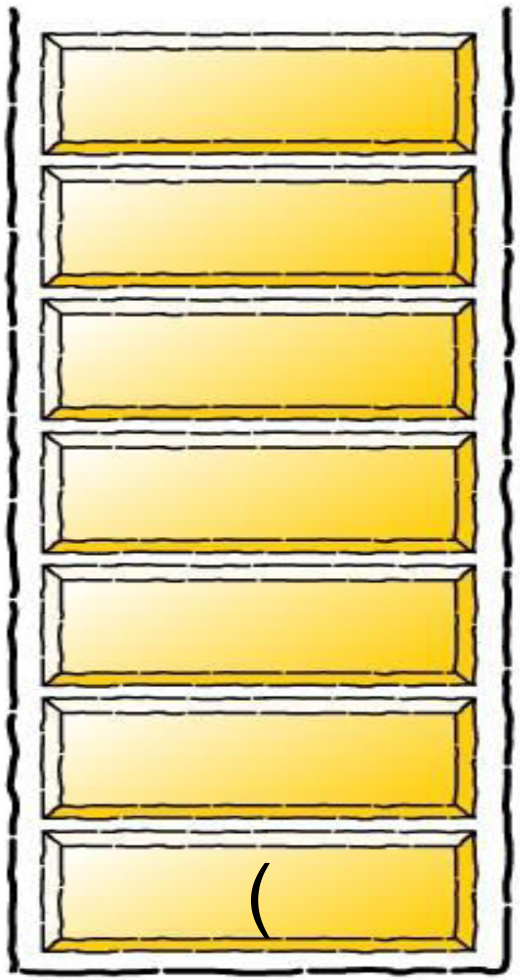
infixVect

a + b - c) * d – (e + f)

postfixVect

Infix to Postfix : Example 2

stackVect



infixVect

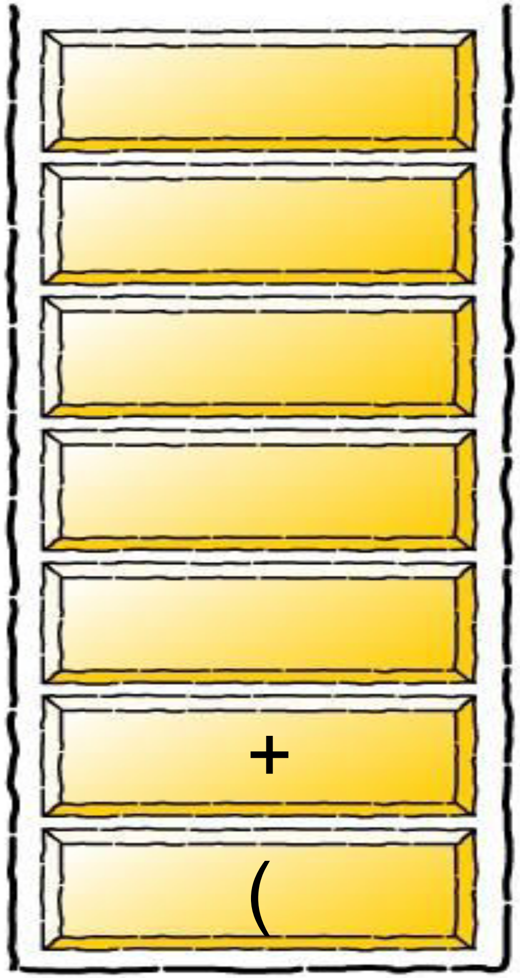
+ b - c) * d – (e + f)

postfixVect

a

Infix to Postfix : Example 2

stackVect



infixVect

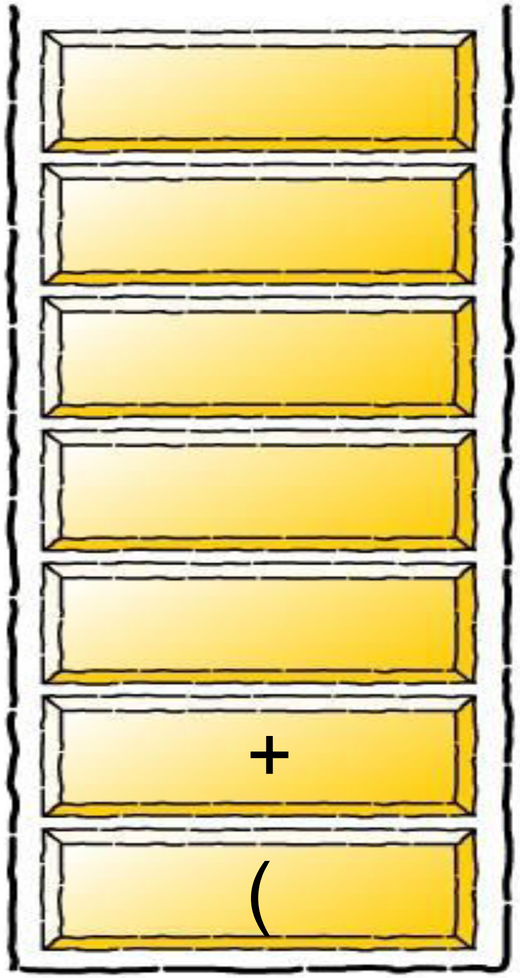
b - c) * d – (e + f)

postfixVect

a

Infix to Postfix : Example 2

stackVect



infixVect

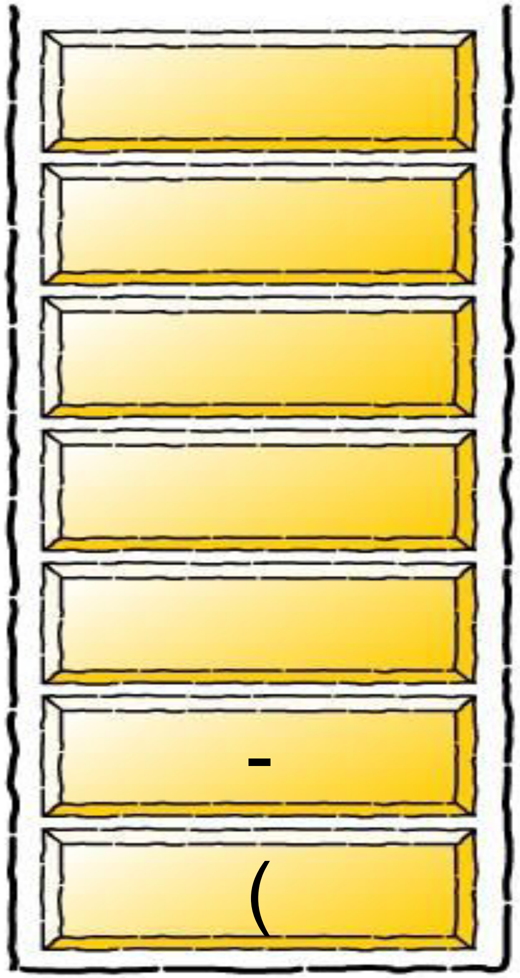
- c) * d – (e + f)

postfixVect

a b

Infix to Postfix : Example 2

stackVect



infixVect

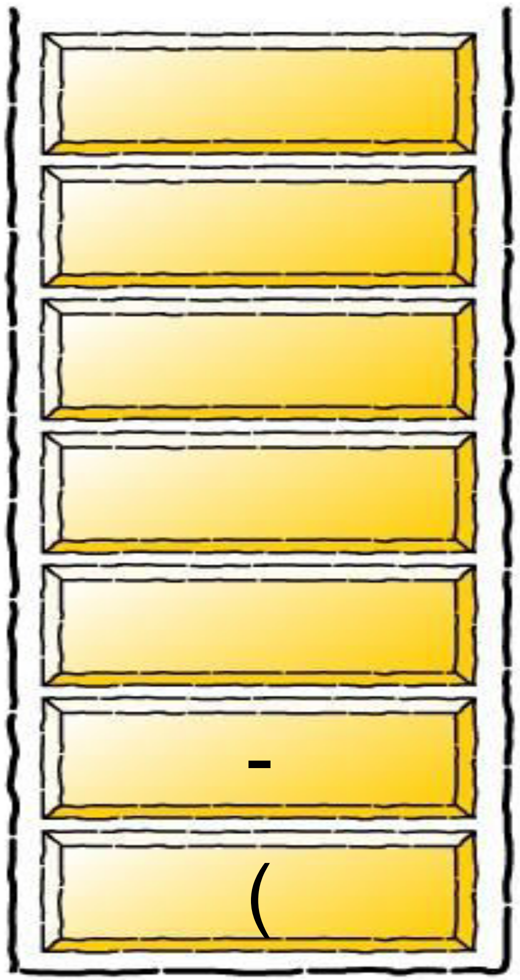
c) * d – (e + f)

postfixVect

a b +

Infix to Postfix : Example 2

stackVect



infixVect

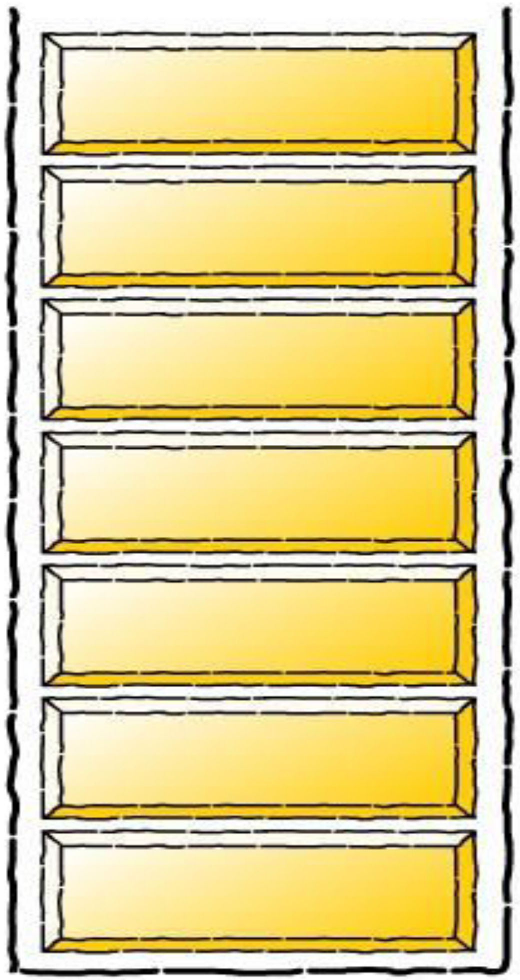
) * d – (e + f)

postfixVect

a b + c

Infix to Postfix : Example 2

stackVect



infixVect

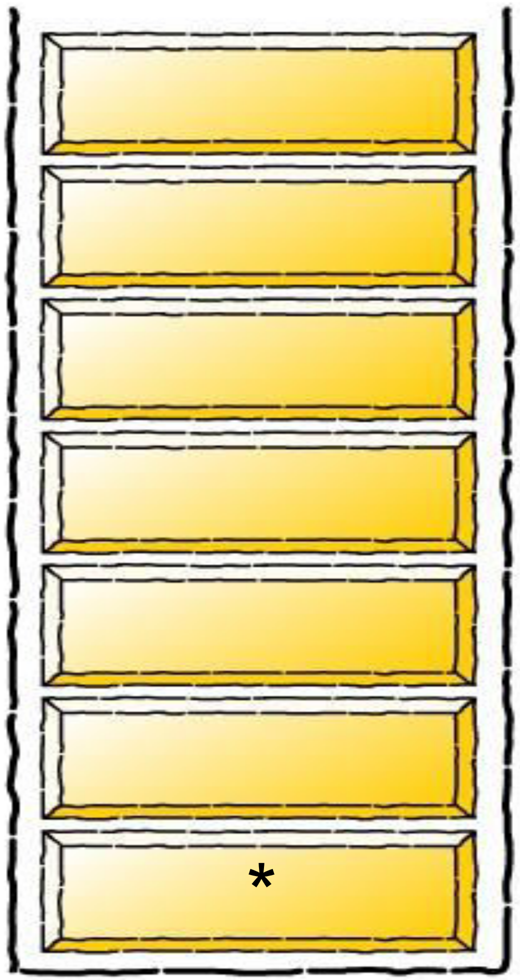
* d – (e + f)

postfixVect

a b + c -

Infix to Postfix : Example 2

stackVect



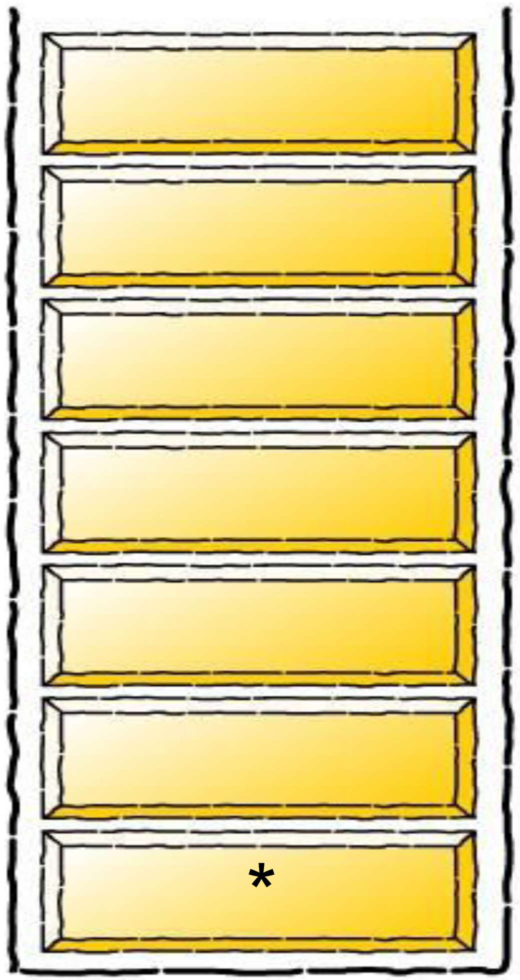
infixVect

d – (e + f)

postfixVect

a b + c -

Infix to Postfix : Example 2



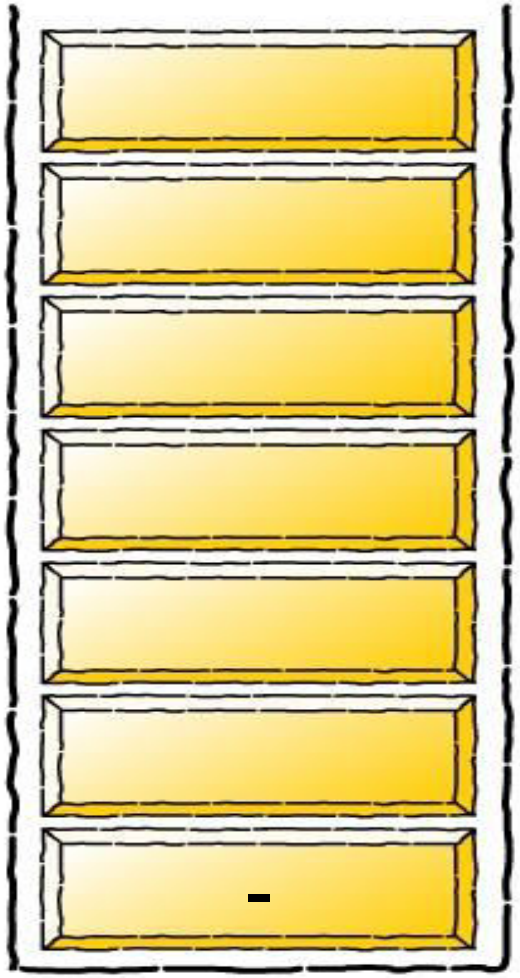
infixVect

– (e + f)

postfixVect

a b + c - d

Infix to Postfix : Example 2



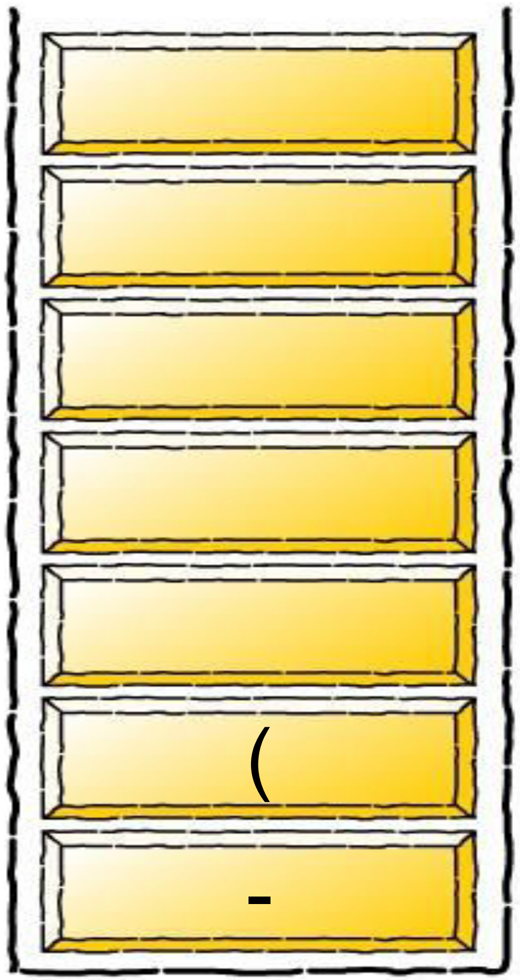
infixVect

(e + f)

postfixVect

a b + c – d *

Infix to Postfix : Example 2



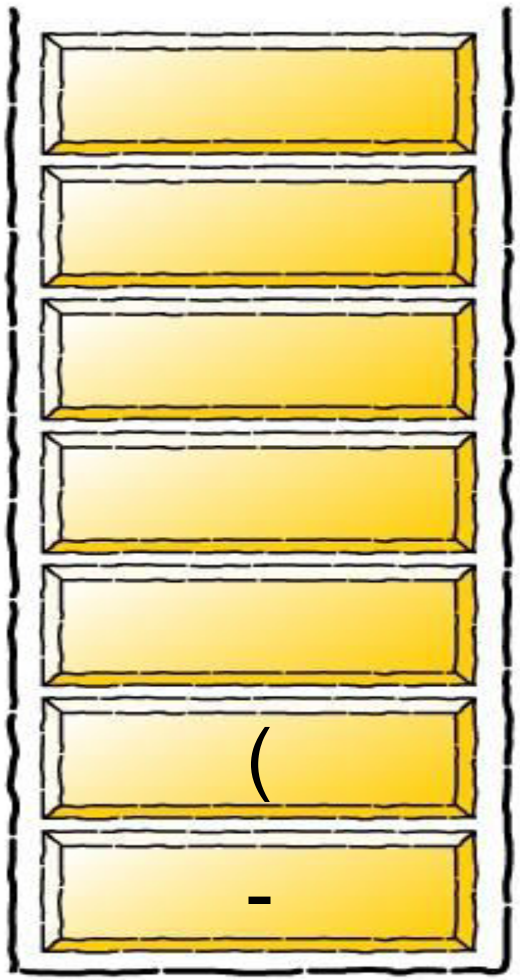
infixVect

e + f)

postfixVect

a b + c – d *

Infix to Postfix : Example 2



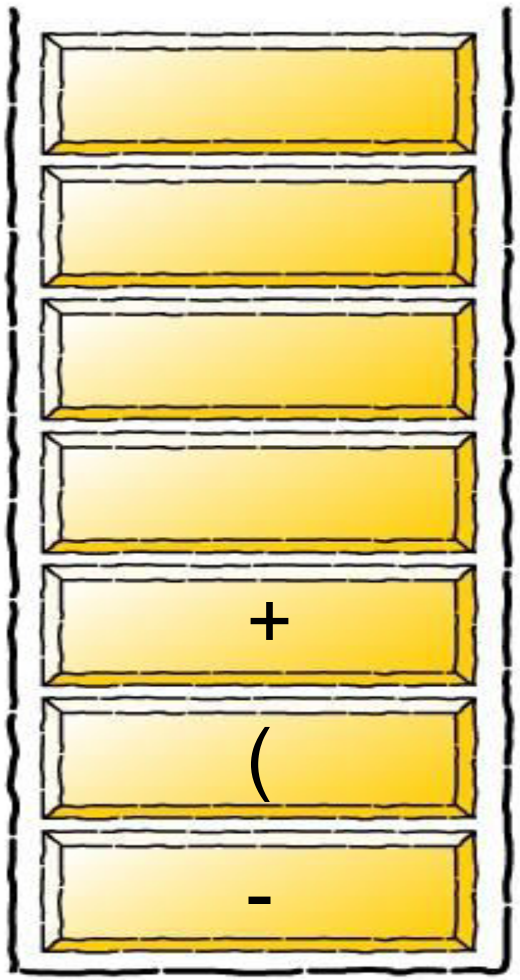
infixVect

+ f)

postfixVect

a b + c – d * e

Infix to Postfix : Example 2



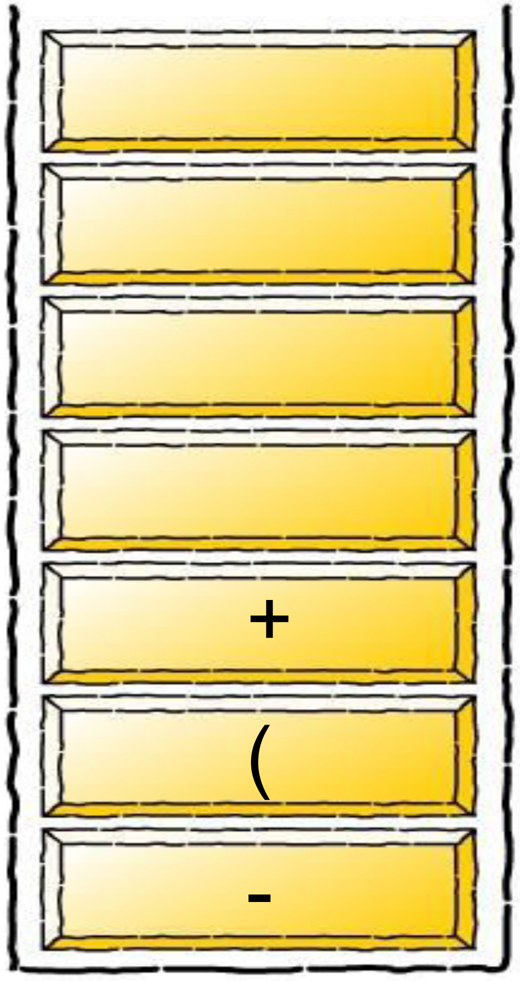
infixVect

f)

postfixVect

a b + c – d * e

Infix to Postfix : Example 2



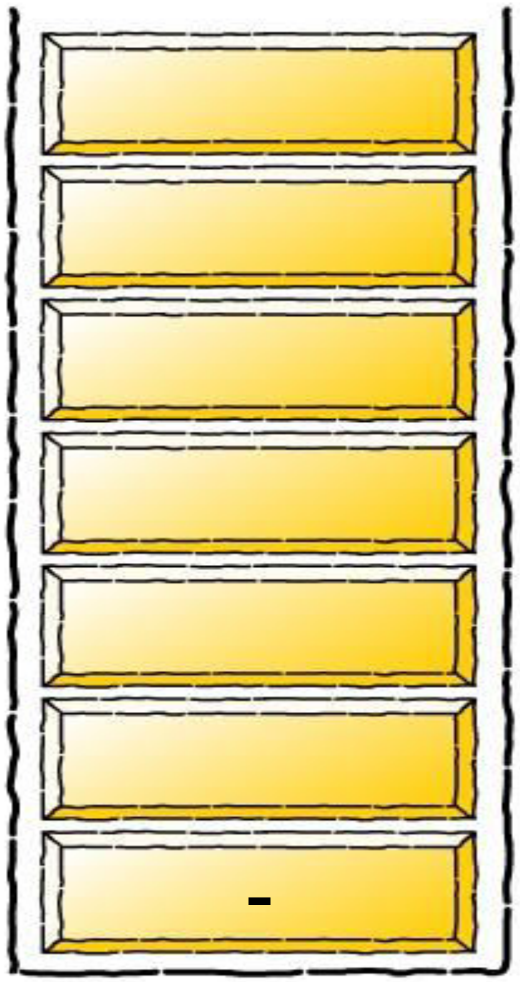
infixVect

)

postfixVect

a b + c – d * e f

Infix to Postfix : Example 2

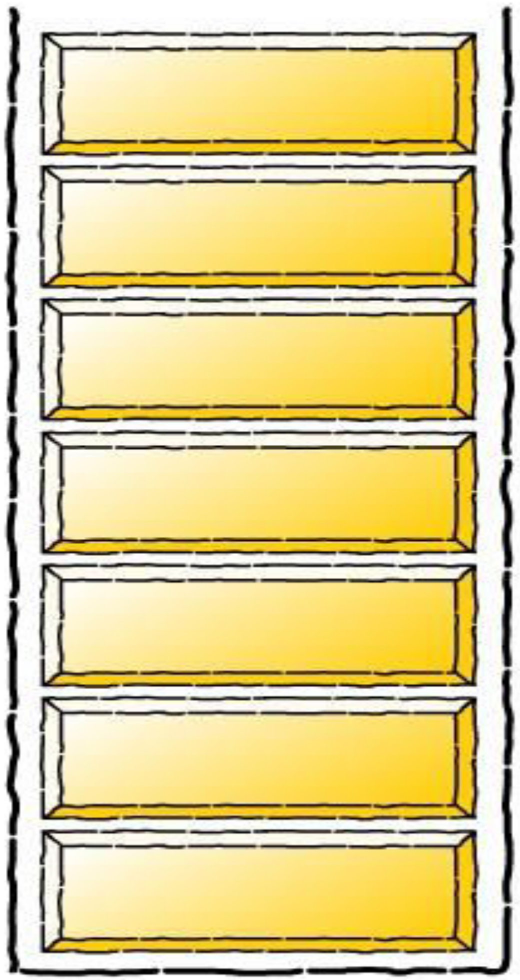


infixVect

postfixVect

a b + c – d * e f +

Infix to Postfix : Example 2



infixVect

postfixVect

a b + c – d * e f + -

Infix to Postfix : Example 3

Expression Conversion Infix to Postfix

Convert 2*3/(2-1)+5*3 into Postfix form

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

In Stack and Incoming priorities functions (Infix→Postfix)

```
icp(ch)
{
 if(ch=='+' || ch=='-')
        return 1;
 if(ch=='*' || ch=='/')
        return 2;
if(ch=='^')
        return 4;
if(ch=='(')
        return 5;
 else
        return 0;
}
```

```
isp( ch)
{
  if(ch=='+' || ch=='-')
        return 1;
  if(ch=='*' || ch=='/')
        return 2;
  if(ch=='^')
        return 3;
  else
        return 0;
}
```

```
Algorithm in_post(inexp[ ])
{ // postexp[ ] has the postfix expression
   k=0; i=0;
  tkn=inexp[i];
  while ( tkn!='\0')
  {   if  tkn is an operand
      {  postexp[k]=inexp[i];
         k++;
      }
      else //1st
      {   if tkn=='('   //open parathensis
          {   push('(');   }
          else //2nd
          {
              if tkn==')'   //open paranthesis
              {  while (tkn=pop()) !='('
                  {     postexp[k]=tkn; k++; }
              }
              else //3rd
              {  while (stack not empty &&
                          isp(stk[top]) >=    icp (tkn) )
                     {   postexp[k]=pop(); k++;
                     }
                 push(tkn);
              } // end of  3rd else
          }//end of 2nd  else
      }// end of 1st else

      // read next token
         i++;
        tkn=inexp[i];
  }//end  of outer while
     while stack not empty
         { postexp[k]=pop(); k++ }
}
```

Infix to Prefix/Postfix

- Higher priority operators should be assigned higher values of ISP and ICP.

- For right associative opeartors,ISP should be lower than ICP.
- A^B^C---ABC^^

- If ICP is higher than ISP,operator should be stacked.

- ISP and ICP should be equal for left associative operators.

Infix to Prefix

Expression = (A+B^C)*D+E^5

Step 1. Reverse the infix expression.

5^E+D*)C^B+A(

Step 2. Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3. use following algorithm in_pre(infix) for Infix to Prefix conversion (Given on next to next slide)

Infix to Prefix

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

+-*^ABCC//DA+EF

```
Input : A * B + C / D

Output : + * A B/ C D


Input : (A - B/C) * (A/K-L)
Output : *-A/BC-/AKL
```

```
Algorithm in_pre(inexp[ ])
{//Input: reversed  infix expression
// Output : pre_exp[ ] has the prefix expression
k=0; i=0;
  tkn=inexp[i];
  while ( tkn!='\0')
  {   if  tkn is an operand
        {pre_exp[k]=inexp[i];
          k++;
        }
        else //1st
        {   if tkn=='('   //open parenthesis
            {   push('(');   }
            else //2nd
             {
                 if tkn==')'   //open parenthesis
                   {  while (tkn=pop()) !='('
                        {pre_exp[k]=tkn; k++; }
                   }
                 else //3rd
                   {  while (stack not empty &&
                              isp(stk[top]) >    icp(tkn) )
                          { pre_exp[k]=pop(); k++;
                          }
                      push(tkn);
                   } // end of  3rd else
                 }//end of 2nd  else
               }// end of 1st else
               // read next token
                   i++;
                  tkn=inexp[i];
               }//end  of outer while
                 while stack not empty
                     {pre_exp[k]=pop(); k++ }
                // reverse pre_exp to get prefix expression
               }
```

Infix to Prefix : Example 1

Expression	Stack	Output	Operation
5^E+D*(C^B+A)	Empty	-	
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop And Push
***(C^B+A)**	+	5E^D	Print
(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

Infix to Prefix : Example 2

A+B*C+D/E
E/D+C*B+A

Ch	prefix	stack
E	E	-
/	E	/
D	ED	/
+	ED/	+
C	ED/C	+
*	ED/C	+, *
B	ED/CB	+, *
+	ED/CB*	+
	ED/CB*+	+
A	ED/CB*+A	+
	ED/CB*+A+	-

Postfix to Infix

Algorithm Post_infx(E)

```
l=length of E
for i =0 to l-1
            x=next_token E
             case x:  =  operand push(x)
                   x:  =  operator
                        op2=pop();
                       op1=pop();
                 E1=strcat('(',op1,x,op2,')');
                     push (E1)
end
```

End Post_infx(E)

```
void postinfx(char post[SIZE])
{
   char s1[SIZE] , s[SIZE],s3[SIZE],temp[SIZE], inf[SIZE];
     for(int i=0;post[i]!='\0';i++)
      {
            s1[0]=post[i];
            s1[1]='\0';
      if(isalpha(post[i])!=0)
            push(s1);
      else
       {
            pop(s2);
            pop(s3);
            strcpy(inf,"(");
            strcat(inf,s3);
            strcat(inf,s1);
            strcat(inf,s2);
            strcat(inf,")");
            push(inf);
        }
    }
    pop(inf);
    printf("\nthe infix expr is: ");
    printf("\n%s",inf);
```

char s[30][30];

```
void push(char str[SIZE])
 {
    if(isFull())
        printf("\nStack is full !");

    else
        strcpy(s[++top],str);

 }
```

```
void pop(char str[20])
{
        if(isEmpty())
            printf("\nStack is empty !");
         else
          {
            strcpy(str,s[top]);
            top--;
         }

}
```

Postfix to prefix

```
Algorithm Post_pre(E)
 l=length of E
 for i =0 to l-1
   x=next_token E
        case x:  =  operand push(x)
             x:  =    operator
                   op2=pop();
                   op1=pop();
                     E1=strcat(x,op1,op2);
                       push (E1)
 end
end Post_pre(E)
```

prefix to infix prefix to postfix

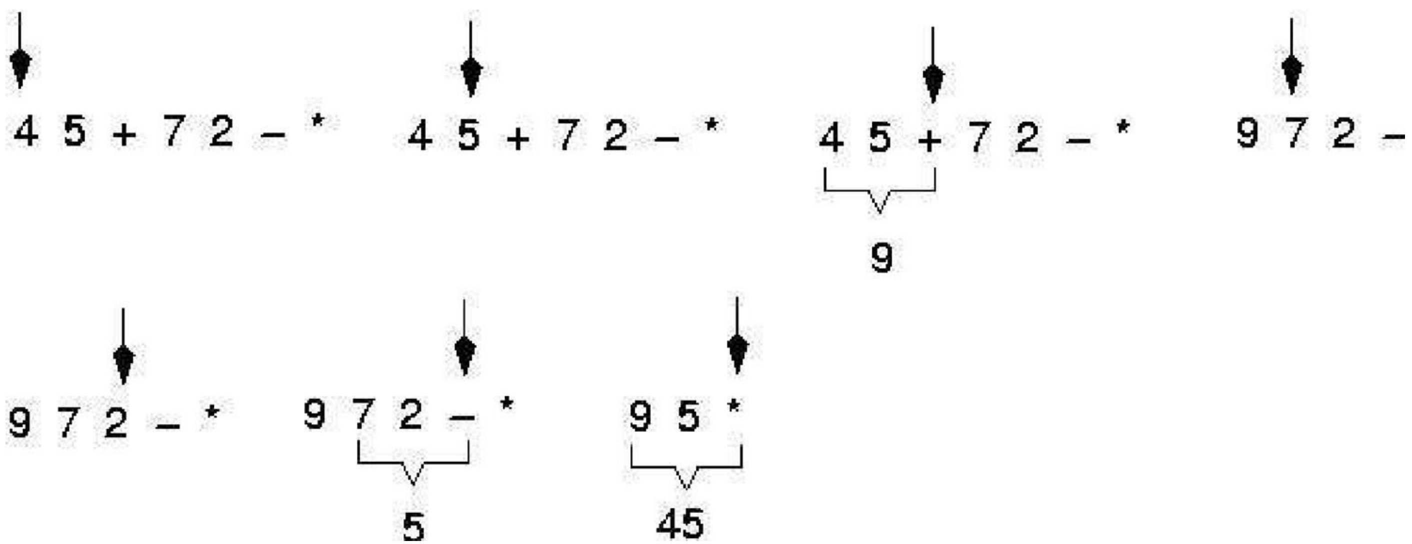
```
Algorithm Pre_infx(E)
 l=length of E
 for i =l-1 to 0
        x=next_token E
        case x: = operand push(x)
             x: = operator
                op1=pop();
               op2=pop();
          E1=strcat('(',op1,x,op2,')');
             push (E1)
 end
End Pre_infx(E)
```

```
Algorithm pre_post(E)
  l=length of E
   for i =l-1 to 0
       x=next_token E
       case x: = operand push(x)
           x: =  operator
                op1=pop();
                op2=pop();
       E1=strcat(op1,op2,x);
              push (E1)
   end
End pre_post(E)
```

Applications of Stacks *(cont'd)*

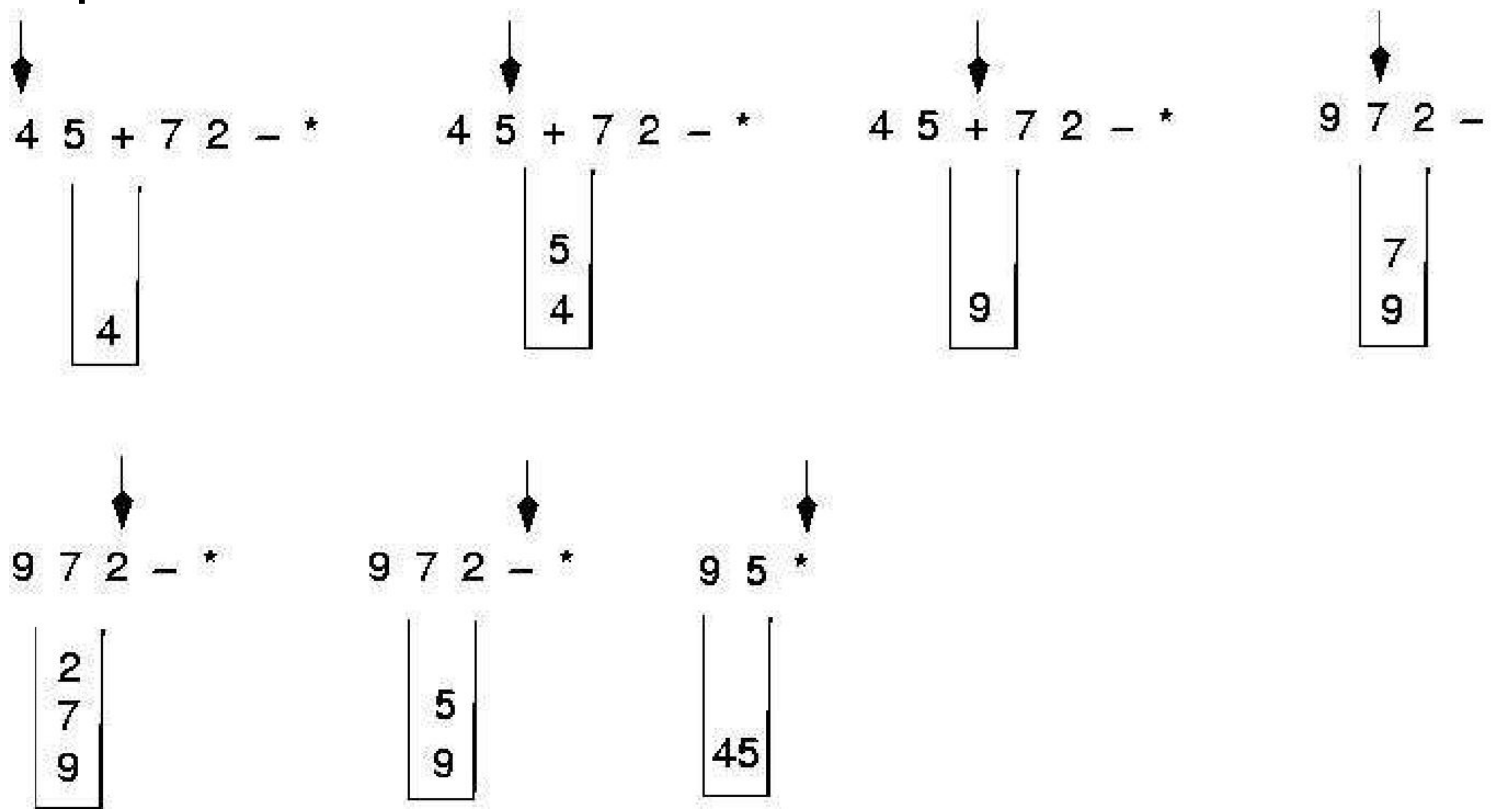
3. Expression Evaluation

- For evaluating expression ,it is converted into prefix or postfix form.
- Expression in postfix or prefix form can be easily evaluated by computer because no precedence of operators is required in this.



Applications of Stacks *(cont'd)*

3. Expression Evaluation
Example



Algorithm for evaluating a postfix expression

```
WHILE more input items exist
{
 If symb is an operand
          then push (opndstk,symb)
 else     //symbol is an operator
 {
          Opnd2=pop(opndstk);
          Opnd1=pop(opndnstk);
          Value = result of applying symb to opnd1 & opnd2
          Push(opndstk,value);
 }                  //End of else
} // end while
Result = pop (opndstk);
```

```
void eval(char post[SIZE])
{
    for(i=0;post[i]!='\0';i++)
  {
    if(isalpha(post[i])!=0)
    {
      printf("\nEnter value of %c",post[i]);
      scanf("%d",&z);
      pushval(z);
    }
    else
    {
      op1=popval();
      op2=popval();
      ans=calc(op1,op2,post[i]);
      pushval(ans);
    }
  }
    printf("\nEvaluation is: ");
    printf("%d",stack[top]);
}
```

```
int calc(int c1,int c2,char op)
{
   switch(op)
  {
    case '+':
      ans=a+b;
      break;
    case '-':
      ans=a-b;
      break;
    case '*':
      ans=a*b;
      break;
    case '/':
      ans=a/b;
      break;
    default:;
  }
  return(ans);
}
```

Question : Evaluate the following expression in postfix : 623+-382/+*2^3+

Final answer is

- 49
- 51
- 52
- 7
- None of these

Evaluate- 623+-382/+*2^3+

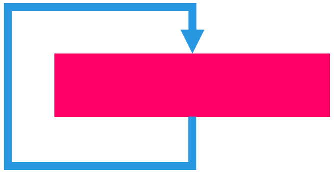
Symbol	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2				7,2
^	7	2	49	49
3				49,3
+	49	3	52	52

Concept of Recursion

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first

- Recursion is a technique that solves a problem by solving a smaller problem of the same type

- A procedure that is defined in terms of itself



Concept of Recursion

What's Behind this function
?

```
int f(int a){
        if (a==1)
          return(1);
        else
          return(a * f( a-1));
}
```

It computes f! (factorial)

Concept of Recursion

Factorial:

$$a! = 1 * 2 * 3 * ... * (a-1) * a$$

Note:

$$a! = a * (a-1)!$$

remember:

...splitting up the problem into a smaller problem of the same type...

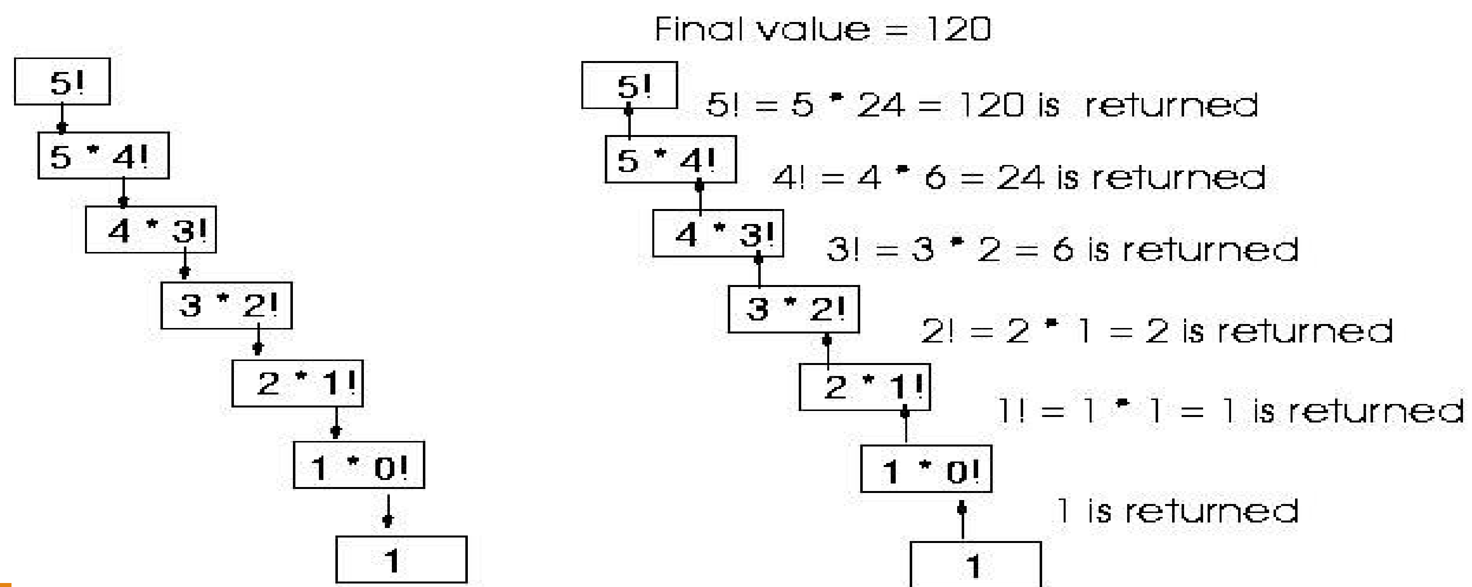
$$a!$$

$$a \quad * \quad (a-1)!$$

Concept of Recursion

```
int factorial(int a){
            if (a==0)
              return(1);
            else
              return(a * factorial( a-1));
}
```

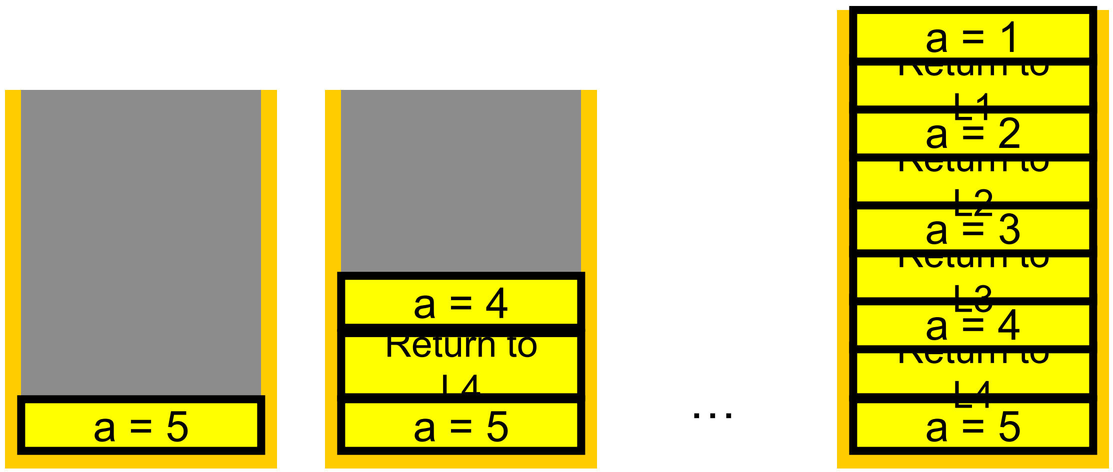
RECURSION !



Concept of Recursion

```
int factorial(int a){
        if (a==1)
          return(1);
        else
          return(a * factorial( a-1));
}
```

Watching the Stack



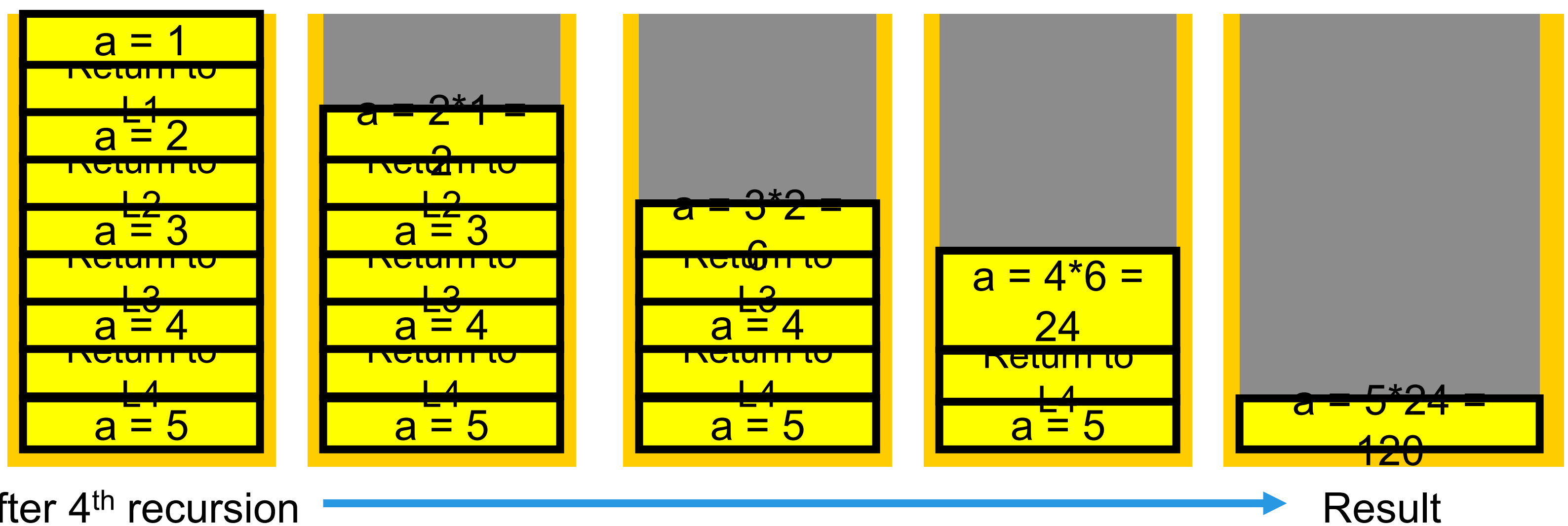
Initial   After 1 recursion   After 4th recursion

Every call to the method creates a new set of local variables !

Concept of Recursion

```
int factorial(int a){
        if (a==1)
          return(1);
        else
          return(a * factorial( a-1));
}
```

Watching the Stack



Properties of Recursion

Problems that can be solved by recursion have these characteristics:

- One or more stopping cases have a simple, nonrecursive solution
- The other cases of the problem can be reduced (using recursion) to problems that are closer to stopping cases
- Eventually the problem can be reduced to only stopping cases, which are relatively easy to solve

Follow these steps to solve a recursive problem:

- Try to express the problem as a simpler version of itself
- Determine the stopping cases
- Determine the recursive steps

Concept of Recursion

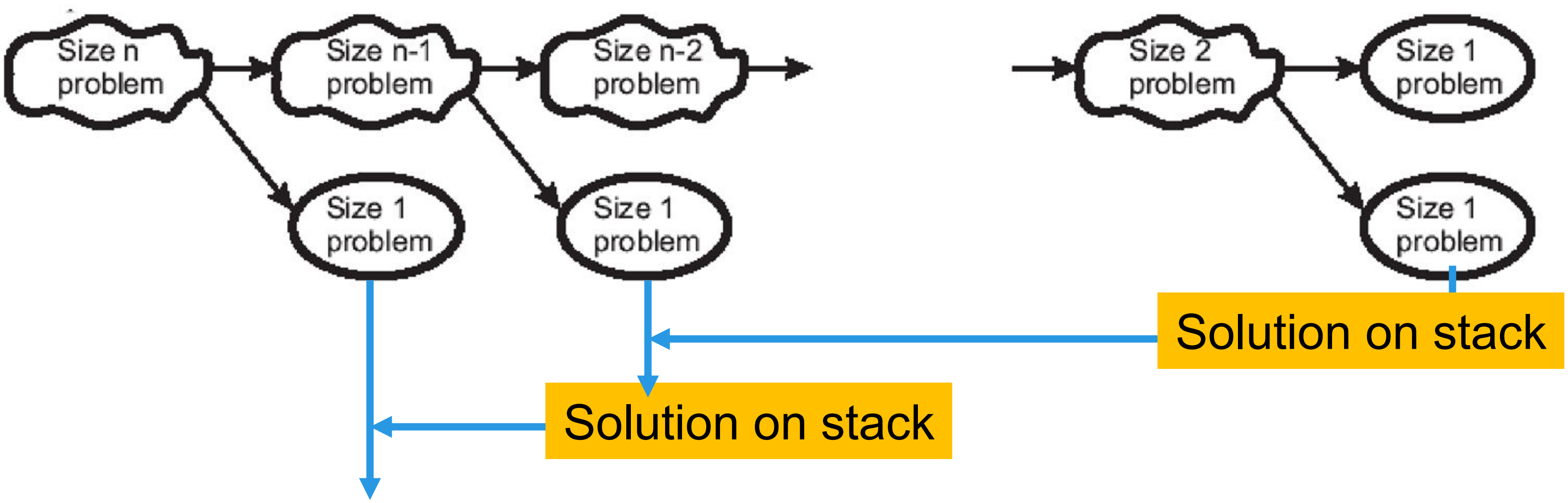
The recursive algorithms we write generally consist of an if statement:

IF

the stopping case is reached solve it

ELSE

split the problem into simpler cases using recursion



Concept of Recursion

Common Programming Error

Recursion does not terminate properly: Stack Overflow !

Exercise

Define a recursive solution for the following function:

$$f(x) = x^n$$

Recursion vs. Iteration

You could have written the power-function *iteratively*, i.e. using a loop construction

Where's the difference ?

Recursion vs. Iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code
- (Nearly) every recursively defined problem can be solved iteratively iterative optimization can be implemented after recursive design

Deciding whether to use a Recursive Function

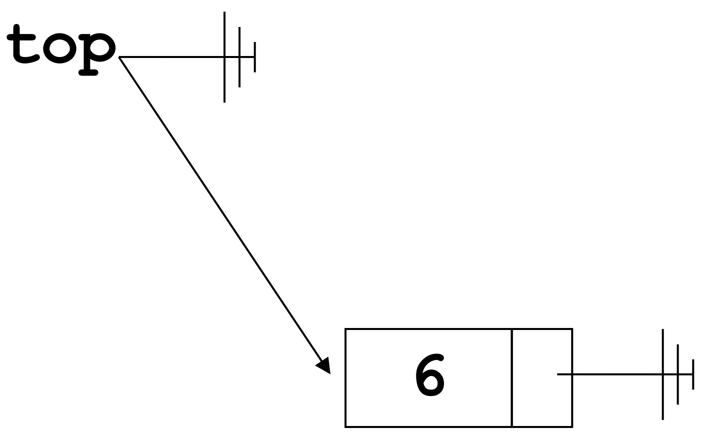
- When the depth of recursive calls is relatively "shallow"
- The recursive version does about the same amount of work as the non recursive version
- The recursive version is shorter and simpler than the non recursive solution

Stack: Linked List Implementation

- Store the items in the stack in a linked list
- The top of the stack is the head node, the bottom of the stack is the end of the list
- *push* by adding to the front of the list
- *pop* by removing from the front of the list

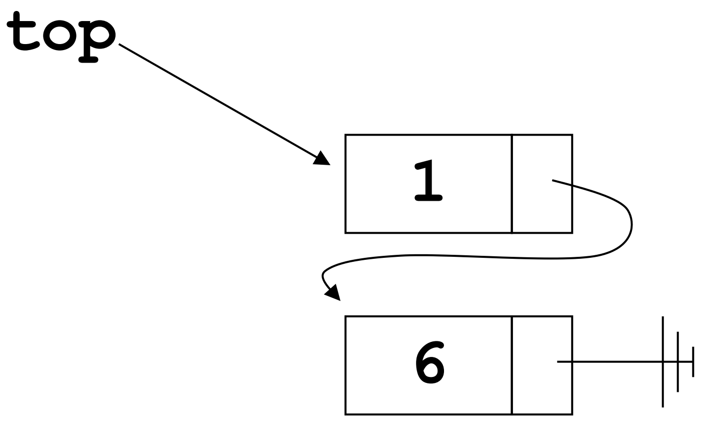
List Stack Example

```
st.push(6);
```



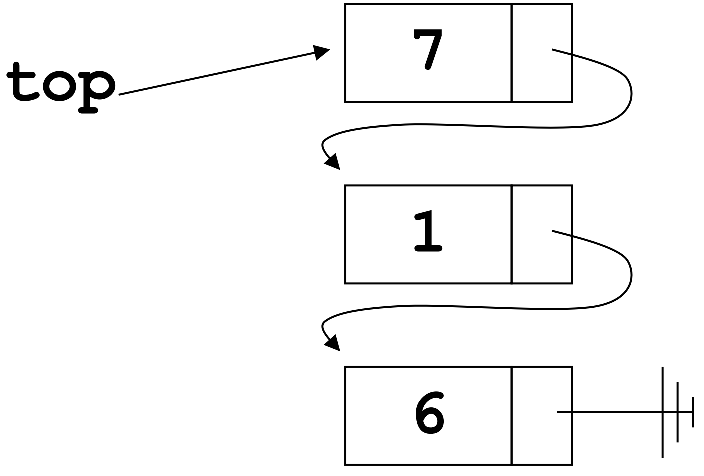
List Stack Example

```
st.push(6);
st.push(1);
```

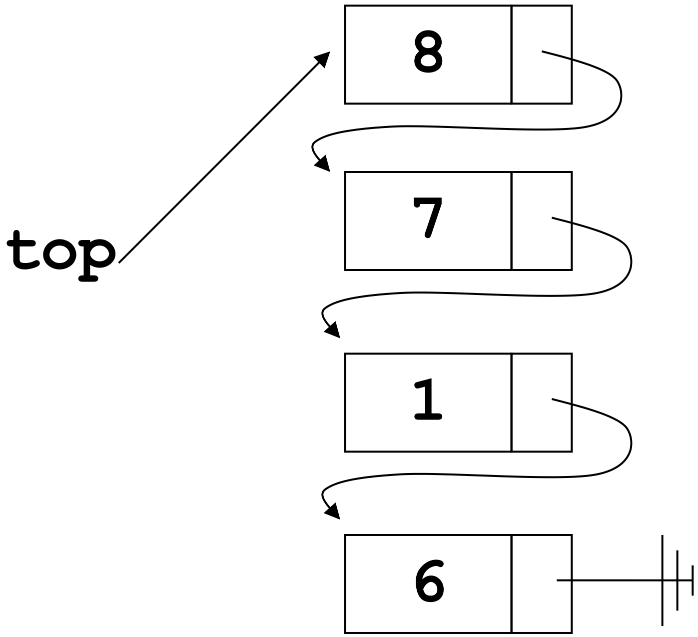


List Stack Example

```
st.push(6);
st.push(1);
st.push(7);
```

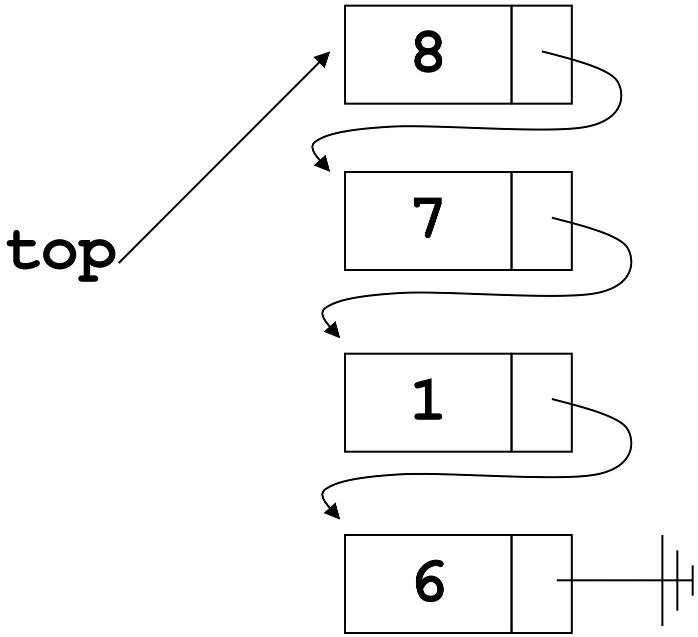


List Stack Example



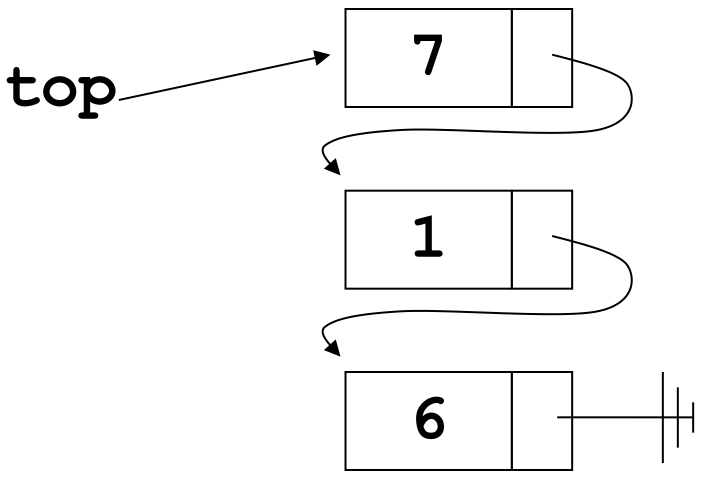

```
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```

List Stack Example




```
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

List Stack Example




```
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

Stack using Linked List

```
typedef struct Stack
{
  int data;
  struct Stack *link;
}stack;


stack *top;
```

```
stack* getnode(int val)
{
  stack *ptr;
  allocate memory for ptr;
  ptr->data = val;
  ptr->link = NULL;
  return(ptr);
}
```

Stack using Linked List

```
void push(int val)
{
  stack *temp;
  temp = getnode(val);
  temp->link=top;
  top=temp;
}
```

```
int pop()
{
  stack *temp;
  int val;
  temp=top;
  val = temp->data;
  top=top->link;
  free(temp);
  return(val);
}
```

Multiple Stack

- **Multiple Stack** computers have two or more stacks supported
- One stack is usually intended to store return addresses, the other stack is for expression evaluation and subroutine parameter passing

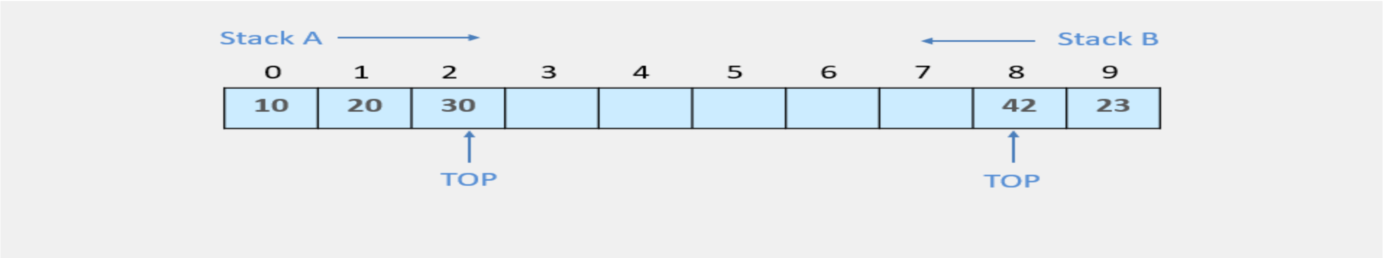
---

- An important advantage of having multiple stacks is one of speed.
- A machine that has simultaneous access to both a data stack and a return address stack can perform subroutine calls and returns in parallel with data operations.
- **STACK[n]** divided into two stack **STACK A** 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.



MULTIPLE STACKS

- Two stacks

m[0], m[1], …, m[n-2], m[n-1]

bottommost

stack 1

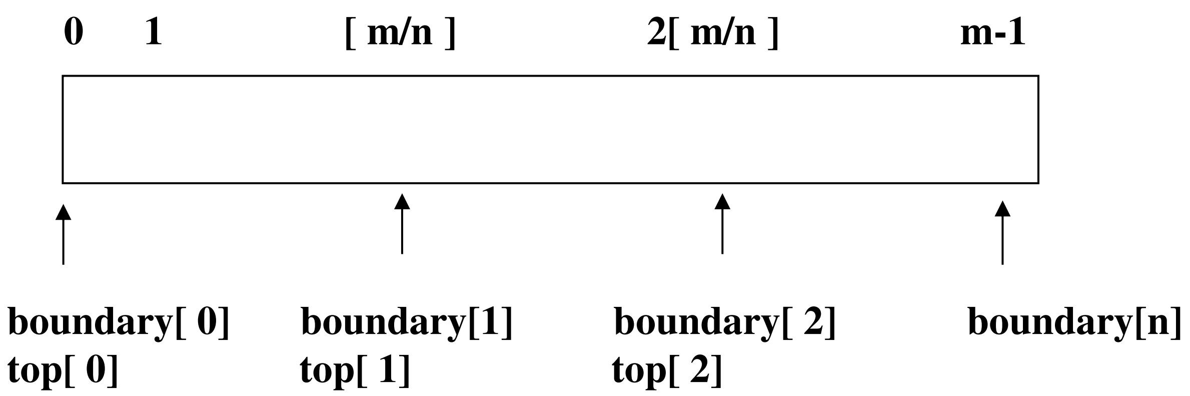
bottommost

stack 2

- More than two stacks (n)
  - memory is divided into n equal segments
  - boundary[stack_no]
    - 0 ≤ stack_no < MAX_STACKS
  - top[stack_no]
    - 0 ≤ stack_no < MAX_STACKS

MULTIPLE STACKS

Initially, boundary[i]=top[i].



All stacks are empty and divided into roughly equal segments.

Multiple Stack

```
#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; }
```

Multiple Stack

```
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++;
  }
}
```

Multiple Stack

If we have only 2 stacks to represent. then the solution is simple.

We can use V (1) for the bottom most element in stack 1 and V(m) for the corresponding element in stack 2.

Stack 1 can grow towards V(m) and stack 2 towards V (1).

It is therefore, possible to utilize efficiently all the available space.

Can we do the same when more than 2 stacks are to be represented?

The answer is no, because:

- a one-dimensional array has only two fixed points V (1) and V(m) and each stack requires a fixed point for its bottommost element.
- When more than two stacks, say n, are to be represented sequentially, we can initially divide out the available memory V(1:m) into n segments and allocate one of these segments to each of the n stacks.
- This initial division of V(1:m) into segments may be done in proportion to expected sizes of the various stacks if the sizes are known.
- In the absence of such information, V(1:m) may be divided into equal segments.
- For each stack i, we shall use B(i) to represent a position one less than the position in V for the bottommost element of that stack.

MULTIPLE STACKS

```
#define MEMORY_SIZE 100          /* size of memory */
#define MAX_STACK_SIZE 100       /* max number of stacks plus 1 */
```

```
element memory [MEMORY_SIZE];     /* global memory declaration */
int top [MAX_STACKS];
int boundary [MAX_STACKS];
int n; /* number of stacks entered by the user */
```

To divide the array into roughly equal segments :

```
top[0] = boundary[0] = -1;
for (i = 1; i < n; i++)
    top [i] =boundary [i] =(MEMORY_SIZE/n)*i;
boundary [n] = MEMORY_SIZE-1;
```

MULTIPLE STACKS

an item to the stack *stack-no*

```
void add (int i, element item) {
 /* add an item to the ith stack */
   if (top [i] == boundary [i+1])
             stack_full (i);     may have unused storage
   memory [++top [i] ] = item;
}
```

an *item* from the stack *stack-no*

```
element delete (int i) {
/* remove top element from the ith stack */
   if (top [i] == boundary [i])
      return stack_empty (i);
   return memory [ top [i]--];
}
```

Takeaways

- Stack is Last in First Out(LIFO) Data Structure
- Primitive operations on Stack are push, pop, isEmpty and isFull
- Stack can be represented by using Array as well as linked list.
- Stack is commonly used in expression conversion and Evaluation.
- Recursion is one of the important application of stack

FAQS

1. Write an ADT for Stack
2. What are the primitive operations of stack.
3. Explain with example stack applications.