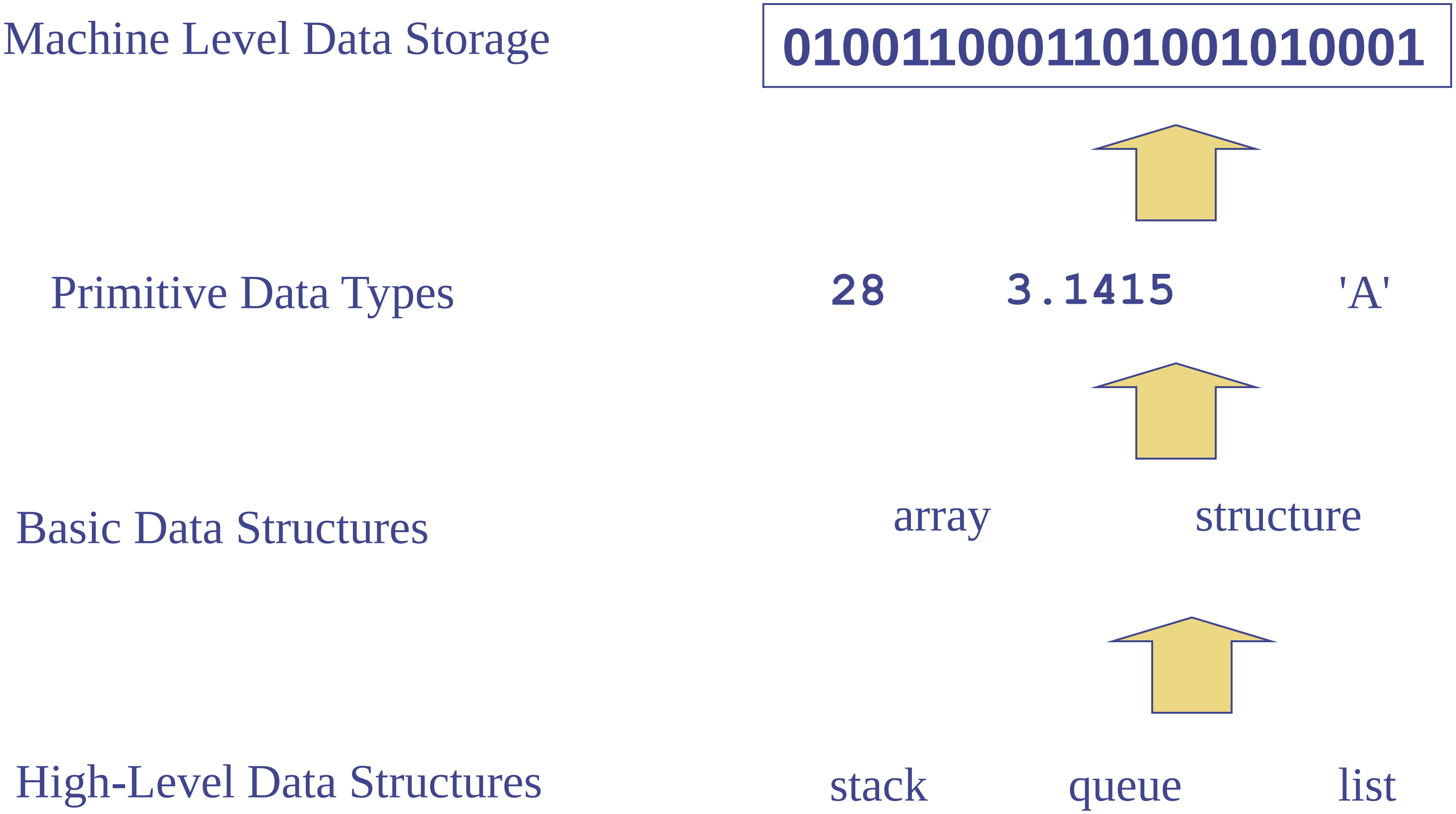
From Data to Data Structures



Machine Level Data Storage 0100110001101001010001

Primitive Data Types `28` `3.1415` 'A'

Basic Data Structures array structure

High-Level Data Structures stack queue list

Important Data Structures

- *Linear structures*
  - Array: Fixed-size
  - Linked-list: Variable-size
  - Stack: Add to top and remove from top
  - Queue: Add to back/end and remove from front
  - Priority queue: Add anywhere, remove the highest priority

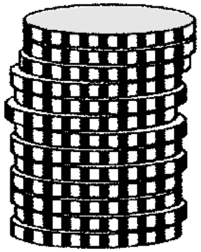
Important Data Structures

- *Hash tables*: Unordered lists which use a ‘hash function’ to insert and search
- *Tree*: A branching structure with no loops
- *Graph*: A more general branching structure, with less stringent connection conditions than for a tree

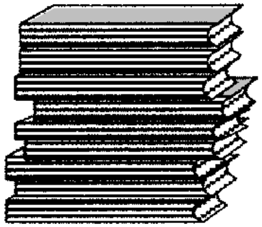
Stack

STACK

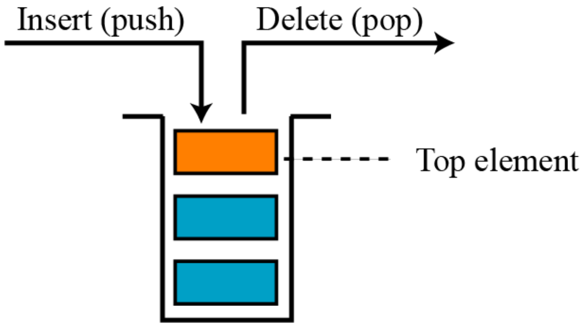
- A stack is a linear data structure in which addition and deletion of an existing element always takes place at the same end.
- A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- This end is known as top of the stack.



Stack of coins



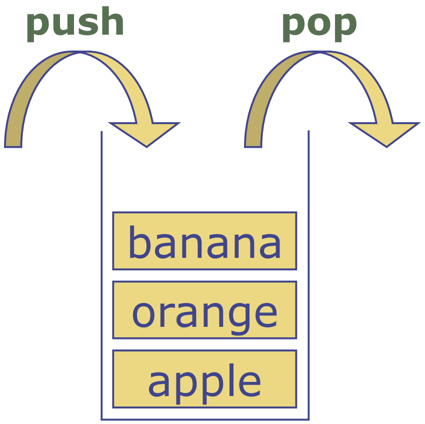
Stack of books



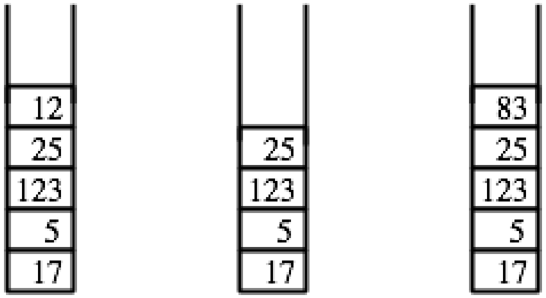
Computer stack

STACK

- *Addition of an element in the stack is called as push operation.*
- *Deletion of an element from stack is known as pop operation.*
- *Stack is also called as last-in first-out (LIFO) list.*
- *Example: Cafeteria Trays*



In a stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the stack. The "pop" operation removes the item on the top of the stack and returns it.



Original stack.   After pop().   After push(83).

Terminology for stacks

- **Stack** = a list in which entries are removed and inserted only at the head
- **LIFO** = last-in-first-out
- **Top** = head of list
- **Bottom** or **base** = tail of list
- **Pop** = remove entry from the top
- **Push** = insert entry at the top

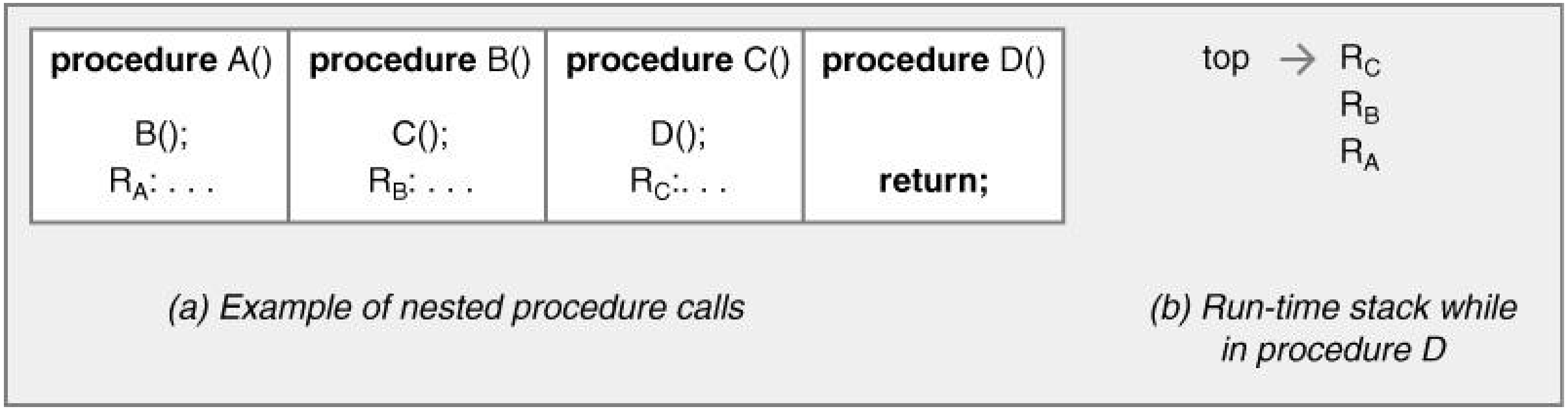
Applications of Stacks

- Direct applications
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine or C++ runtime environment (Program execution)
  - Parsing
  - Reversing a string
  - postponing data usage and backtracking steps.
  - Evaluating an expression
  - Call stack (recursion).
  - Searching networks, traversing trees (keeping a track where we are).

  Examples:
    - Checking balanced expressions
    - Recognizing palindromes
    - Evaluating algebraic expressions
- Indirect applications
  - Component of other data structures

Stack Applications

- Run-time procedure information



procedure A()	**procedure** B()	**procedure** C()	**procedure** D()
B(); R_A: . . .	C(); R_B: . . .	D(); R_C:. . .	**return;**

(a) Example of nested procedure calls

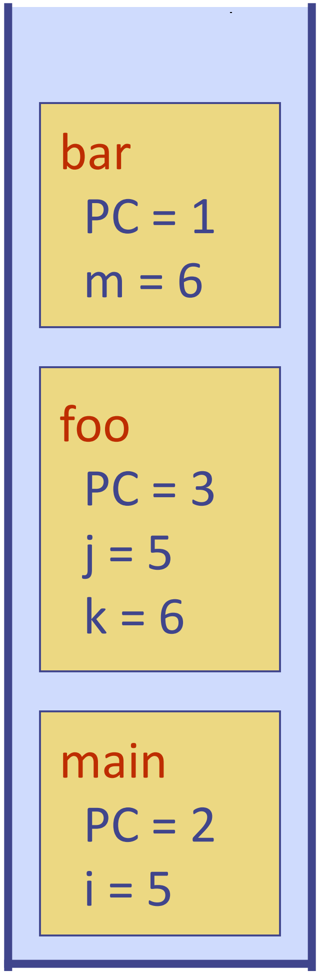
(b) Run-time stack while in procedure D

- Arithmetic computations
  - Postfix notation

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

```
main()
{
int i = 5;
foo(i);
}
foo(int j)
{
int k;
k = j+1;
bar(k);
}
bar(int m)
{
…
}
```



Stack Operation

- All the data item is accessed at one end and it is called top of the stack

◆ Operations

1. **Stack: create an empty stack**
2. **Push : insert the element at the top of the stack**
3. **Pop : Delete the element from the top of the stack**
4. Peep : Function returns the value of the ith element from top of the stack
5. Change: It changes the value of the ith element from the top of the stack to new value.
6. **Empty : Check whether stack is empty or not**

Procedure : PUSH (S, TOP, X)

- This procedure inserts an element **X** to the top of a stack.
- Stack is represented by a vector **S** containing **N** elements.
- A pointer **TOP** represents the top element in the stack.

```
1. [Check for stack overflow]
        If      TOP ≥ N
        Then    write ('STACK OVERFLOW')
                Return
2. [Increment TOP]
        TOP ← TOP + 1
3. [Insert Element]
        S[TOP] ← X
4. [Finished]
        Return
```

Stack is empty, TOP = 0, N=3

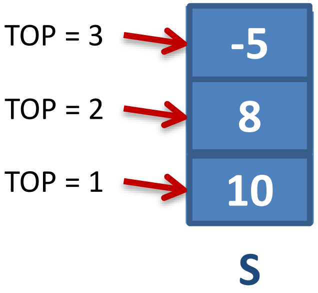
PUSH(S, TOP, 10)

PUSH(S, TOP, 8)

PUSH(S, TOP, -5)

PUSH(S, TOP, 6)

Overflow



Procedure : POP (S, TOP)

- This function ***removes & returns*** the top element from a stack.
- Stack is represented by a vector **S** containing **N** elements.
- A pointer **TOP** represents the top element in the stack.

```
1. [Check for stack underflow]
        If      TOP = 0
        Then    write ('STACK UNDERFLOW')
                Return (0)
2. [Decrement TOP]
        TOP ← TOP - 1
3. [Return former top element of stack]
        Return(S[TOP + 1])
```

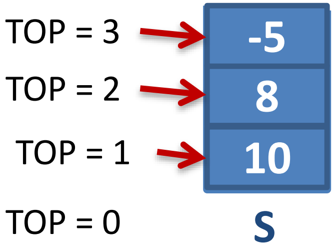
POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

Underflow



Procedure : PEEP (S, TOP, I)

- This function returns the value of the I^{th} element from the **TOP** of the stack. The element is not deleted by this function.
- Stack is represented by a vector **S** containing **N** elements.

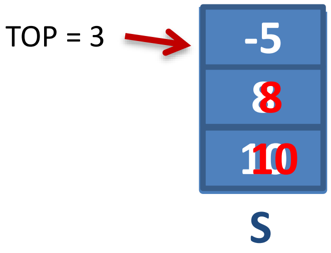
```
1. [Check for stack underflow]
        If      TOP-I+1 ≤ 0
        Then    write ('STACK UNDERFLOW')
                Return (0)
2. [Return I^th element from top of the stack]
        Return(S[TOP–I+1])
```

PEEP (S, TOP, 2)

PEEP (S, TOP, 3)

PEEP (S, TOP, 4)

Underflow



Procedure : CHANGE (S, TOP,X,I)

- This procedure changes the value of the **I**th element from the top of the stack to **X**.
- Stack is represented by a vector **S** containing **N** elements.

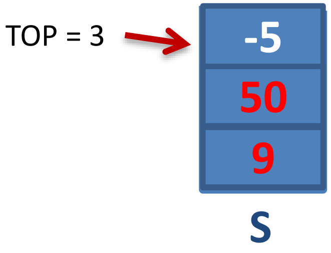
```
1.  [Check for stack underflow]
        If      TOP-I+1 ≤ 0
        Then    write ('STACK UNDERFLOW')
                Return
2. [Change Ith element from top
   of the stack]
        S[TOP–I+1] ← X
3. [Finished]
        Return
```

CHANGE (S, TOP, 50, 2)

CHANGE (S, TOP, 9, 3)

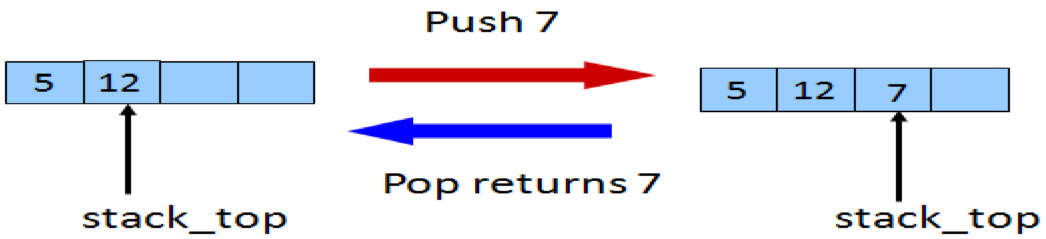
CHANGE (S, TOP, 25, 8)

Underflow



Representation of Stack

- To implement stack we can use
  - Array
  - Linked list
- For Array representation
  - Use an element array of MAX size to represent a stack.
  - Use a variable TOP to represent the index/or address of the top element of the stack in the array. It is this position from where the element will be added or removed
  - TOP = -1 indicates that the stack is empty
  - TOP = MAX -1 indicates that the stack is full



Program for Stack Operations using Array

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(){
   int a[20],i,c,t,x,d,j;
   char ch='y';  t=0;
   while(ch=='Y' || ch=='y'){
          printf("\t\t\t\t");
          printf("Welcome\n\n");
          printf("\t\t\t[1]. Push.\n");
          printf("\t\t\t[2]. Pop.\n");
          printf("\t\t\t[3]. Peep.\n");
          printf("\t\t\t[4]. Change(by y).\n");
          printf("\t\t\t[5]. Change(interchange).\n");
          printf("\t\t\t[6]. Display.\n");
          printf("\t\t\t[7]. Exit.\n");
```

Program for Stack Operations using Array

```
printf("Top= %d",t);
printf("\n\nEnter your choice:");
fflush(stdin);
scanf("%d",&c);

switch(c){
        case 1:  //PUSH
                if(t>=20)
                        printf("An overflow has been occurred");
                else{
                        printf("\n\nEnter the number:");
                        scanf("%d", &x);
                        a[t]=x;
                        t+=1;
                        printf("\nThe number is added.");    }
                break;
```

Program for Stack Operations using Array

```
case 2:  //  POP
        if(t==0)
                printf("An underflow has been occurred");
        else{
                printf("\n\n Number is a[%d] = %d", t, a[t-1]);
                t-=1;
                printf("\n\n The top record %d has been
                        popped",a[t]);
                }
        break;
```

Program for Stack Operations using Array

```
                case 3:  //PEEP
                        printf("\n\nEnter which element u want to
see:");
                        scanf("%d",&i);
                        if((t-i)<0)
                                printf("\n\n\n Underflow");
                        else
                                printf("\n\n\n The number is %d",a[t-i]);
                        break;
```

Program for Stack Operations using Array

```
    case 4:  //CHANGE
            printf("\n\nEnter which element you want to change :");
            scanf("%d",&i);
            if((t-i)==0)
                    printf("\n\n 'Underflow' ");
            else{
                    printf("\n\n Enter a new value:");
                    scanf("%d",&x);
                    a[t-i]=x;
            }
            break;
```

Program for Stack Operations using Array

```
case 5:  //INTERCHANGE
        printf("\n\nEnter which element to place:");
        scanf("%d",&i);
        if((t-i)<=0)
                printf("\n\n 'Underflow' ");
        else{
                printf("Enter second element to place:");
                scanf("%d",&j);
                if((t-j)<=0)
                        printf("\n\n 'Underflow' ");
                else{
                        c=a[t-i];
                        a[t-i]=a[t-j];
                        a[t-j]=c;                       }
        }
        break;
```

Program for Stack Operations using Array

```
                case 6:  //DISPLAY
                        printf("\n\n\nStack is---->\n\n\n");
                        for(i=t-1; i>=0; i--){
                                printf("\t\t\t|%d |\n",a[i]);
                                printf("\t\t\t|_________|\n");
                        }
                        break;
                case 7:
                        scanf("Exit");
                default:
                        ch=‘n’;
                        printf("\t\t\t\t");
                        printf("Thank You");
                        break;

        }
    }
}
```

Infix Notation

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed between the operands. For example, *A+B;* here, plus operator is placed between the two operands A and B.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix Notation

- Postfix notation also known as **Polish** notation and a postfix notation which is better known as Reverse Polish Notation or RPN.
- In postfix notation, the operator is placed after the operands. For example, if an expression is written as *A+B* in infix notation, the same expression can be written as *AB+* in postfix notation.
- The order of evaluation of a postfix expression is always from left to right.
- The expression (A + B) * C is written as:
  - AB+C* in the postfix notation.
- A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands.
- For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

Prefix Notation

- In a prefix notation, the operator is placed before the operands.
- For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence, associativity, and even brackets cannot alter the order of evaluation.
- The expression (A + B) * C is written as:

  *+ABC in the prefix notation

Arithmetic Priority

Parentheses() -- 1
Exponent(^,$,|) – 2
Multiplication/ division – 3
Addition/Substraction -- 4

In case of equal precedence/priority of operators ,expression solved from left to right.

<table>
<tr><th>Infix</th><th>Postfix</th><th>Prefix</th></tr>
<tr><td>A+B*C</td><td>ABC*+</td><td>+A*BC</td></tr>
<tr><td>((A-(B+C))*D)|(E+F)</td><td>ABC+-D*EF+|</td><td>|*-A+BCD+EF</td></tr>
<tr><td>(A+B)*(C-D)$E*F</td><td>AB+CD-E$*F*</td><td>**+AB$-CDEF</td></tr>
</table>

Example(((A-(B+C))*D)|(E+F)

- Post fix

((A-BC+)*D)|(E+F)

(ABC+- * D)|(E+F)

ABC+-D* | (E+F)

ABC+-D* | EF+

ABC+-D*EF+|

- Prefix

((A - +BC)*D)|(E+F)

(-A+BC * D) |(E+F)

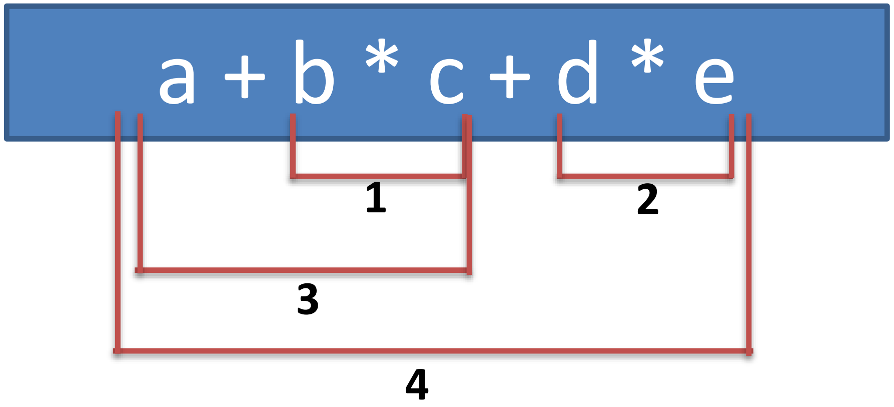
*-A+BCD |(E+F)

*-A+BCD | +EF

|*-A+BCD+EF

Polish Expression & their Compilation

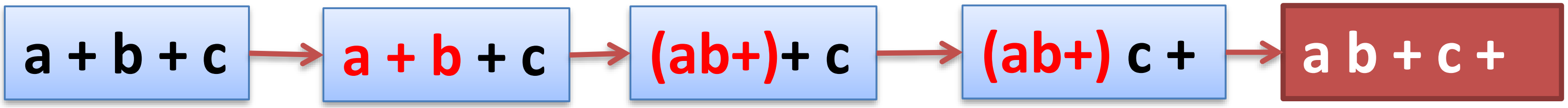
- Evaluating Infix Expression



- A **repeated scanning from left to right is needed** as operators appears inside the operands.
- ***Repeated scanning is avoided*** if the **infix expression** is first **converted** to an equivalent parenthesis free ***prefix or suffix (postfix) expression***.
- **Prefix Expression: Operator**, Operand, Operand
- **Postfix Expression:** Operand, Operand, **Operator**

Polish Notation

Sr.	Infix	Postfix	Prefix
1	a	a	a
2	a + b	a b +	+ a b
3	a + b + c	a b + c +	+ + a b c
4	a + (b + c)	a b c + +	+ a + b c
5	a + (b * c)	a b c * +	+a * b c
6	a * (b + c)	a b c + *	* a + b c
7	a * b * c	a b *c*	** a b c



a + b + c → a + b + c → (ab+)+ c → (ab+) c + → a b + c +

Evaluation of postfix expression

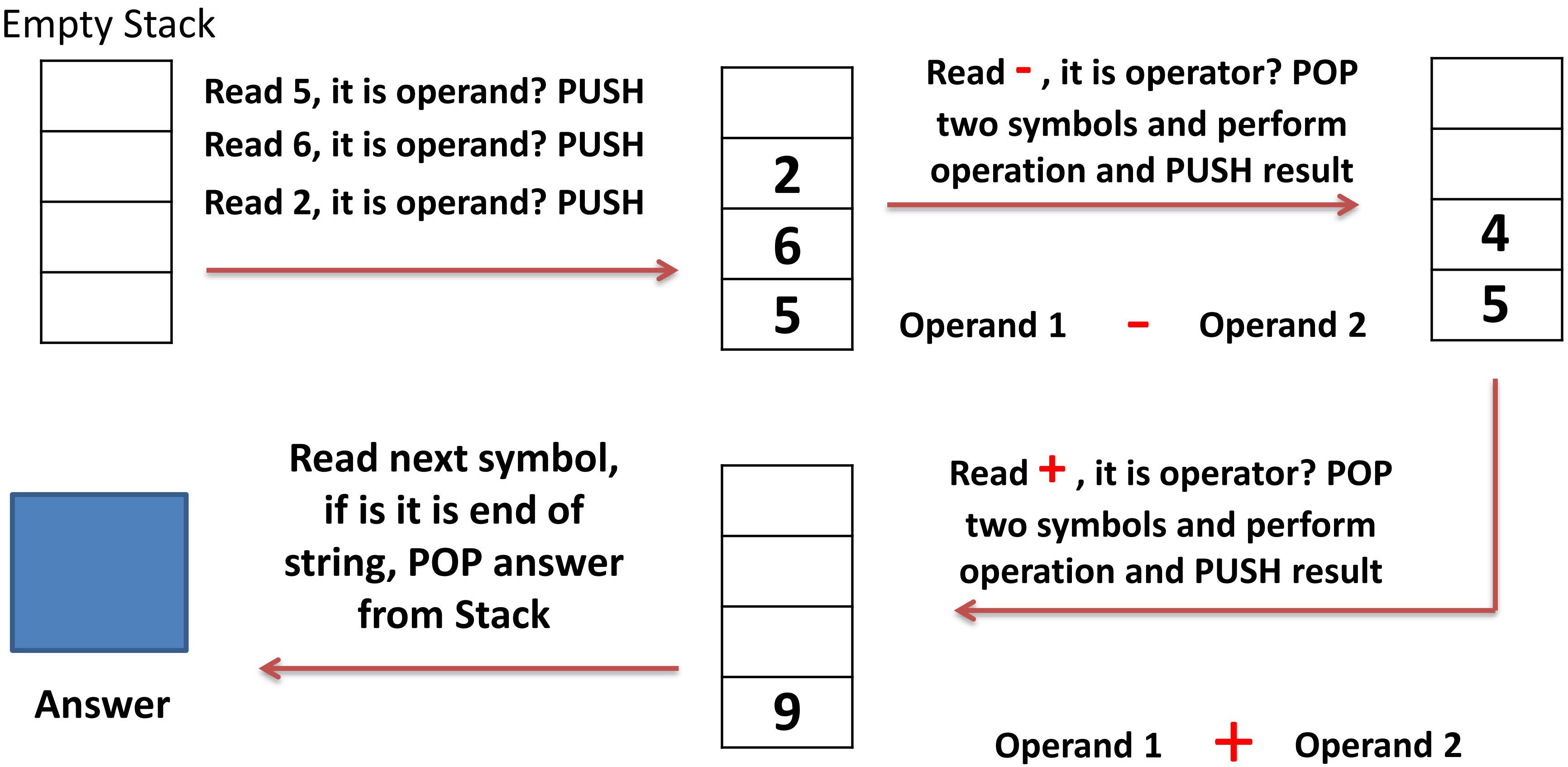
- Each **operator** in **postfix** string **refers** to the *previous two operands* in the string.
- Each time we **read** an **operand**, we **PUSH** it onto **Stack**.
- When we reach an **operator**, its **operands** will be **top two elements** on the stack.
- We can then **POP** these two elements, perform the indicated operation on them and PUSH the result on the stack so that it will available for use as an operand of the next operator.

Algorithm: EVALUAE_POSTFIX

```
1.  Add ) to postfix expression.
2.  Read postfix expression Left to Right until ) encountered
3.  If operand is encountered, push it onto Stack
    [End If]
4.  If operator is encountered, Pop two elements
    i) A -> Top element
    ii) B-> Next to Top element
    iii) Evaluate B operator A
    push B operator A onto Stack
5.  Set result = pop
6.  END
```

Evaluation of postfix expression

Evaluate Expression: 5 6 2 - +



Read 5, it is operand? PUSH

Read 6, it is operand? PUSH

Read 2, it is operand? PUSH

Read - , it is operator? POP two symbols and perform operation and PUSH result

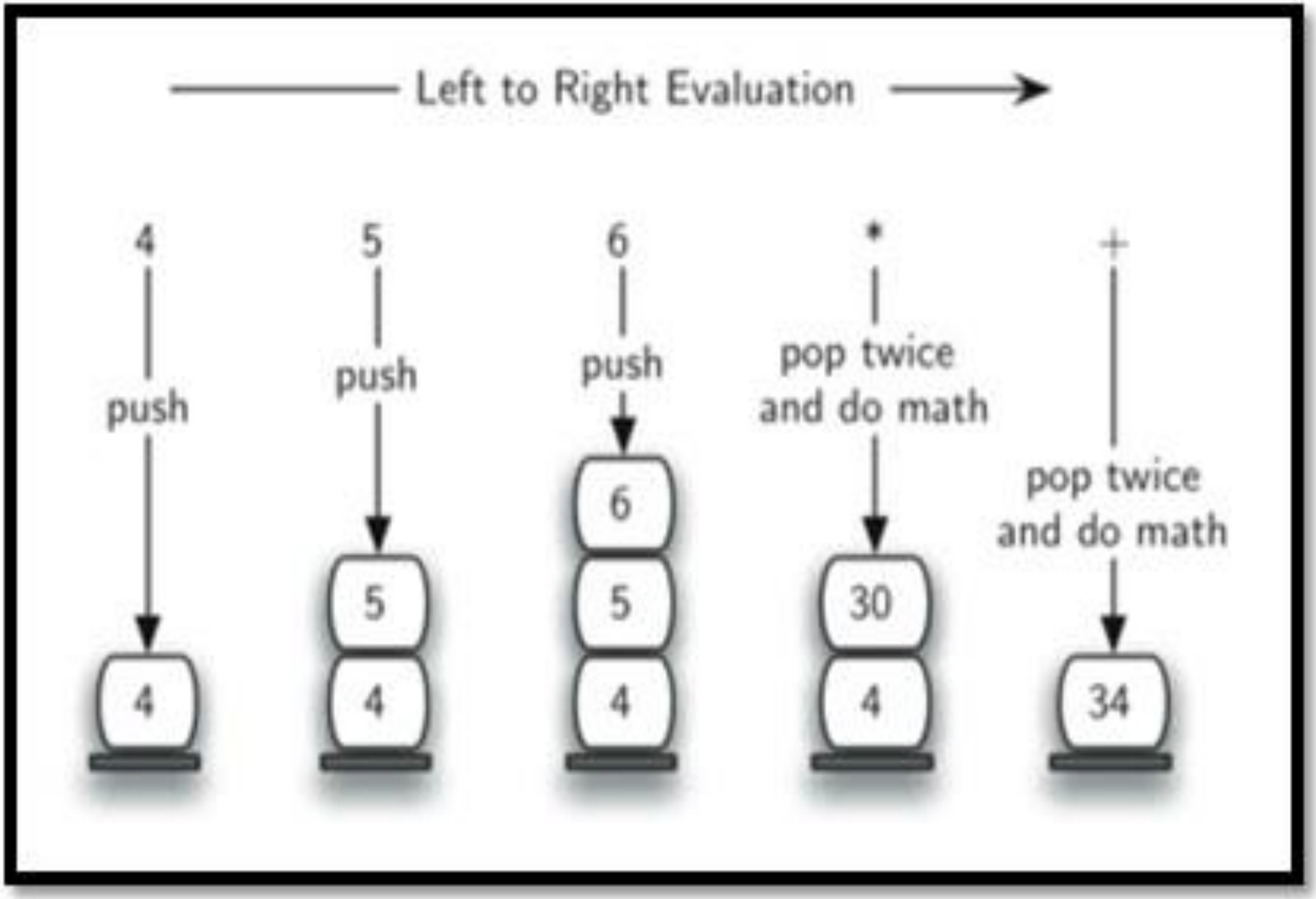
Operand 1 - Operand 2

Read + , it is operator? POP two symbols and perform operation and PUSH result

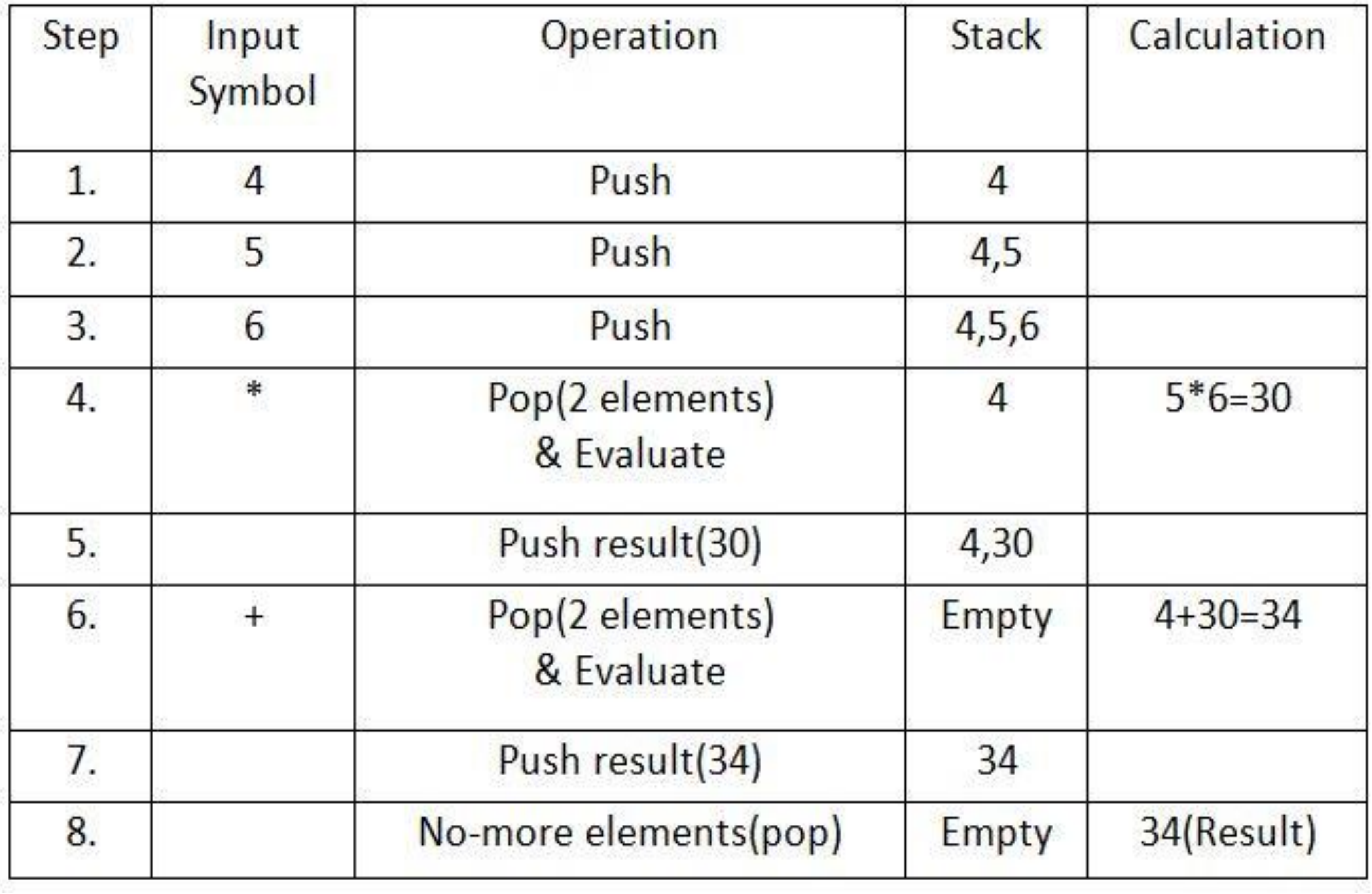
Operand 1 + Operand 2

Read next symbol, if is it is end of string, POP answer from Stack

Expression: 456*+



Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Algorithm: EVALUAE_POSTFIX

```
1. [Initialize Stack]
     TOP ← 0
     VALUE ← 0
```

```
2. [Evaluate the postfix expression]
Repeat until last character
     TEMP ← NEXTCHAR (POSTFIX)
     If  TEMP is DIGIT
     Then PUSH (S, TOP, TEMP)
     Else OPERAND2 ← POP (S, TOP)
          OPERAND1 ← POP (S, TOP)
          VALUE ← PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
          PUSH (S, TOP, VALUE)
```

```
3. [Return answer from stack]
    Return (POP (S, TOP))
```

Evaluation of postfix expression Program

```
/* This program is for evaluation of postfix expression.  This program
assume that there are only four operators * (*, /, +, -) in an expression
and operand is single digit only. * Further this program does not do any
error handling e.g. it does not check that entered postfix expression is
valid or not. */
#include <stdio.h>
#include <ctype.h>

#define MAXSTACK 100  /* for max size of stack */
#define POSTFIXSIZE 100
            /* define max number of characters in postfix expression */

/* declare stack and its top pointer to be used during postfix expression
evaluation*/
int stack[MAXSTACK];
int top = -1;         /* because array index in C begins at 0 */
```

Evaluation of postfix expression Program

```
/* define push operation */
void push(int item)
{

    if (top >= MAXSTACK - 1) {
        printf("stack over flow");
        return;
    }
    else {
        top = top + 1;
        stack[top] = item;
    }
}
```

Evaluation of postfix expression Program

```
/* define pop operation */
int pop()
{
   int item;
   if (top < 0) {
      printf("stack under flow");
   }
   else {
      item = stack[top];
      top = top - 1;
      return item;
   }
}
```

Evaluation of postfix expression Program

```
/* define function that is used to input postfix expression and to
evaluate it */
void EvalPostfix(char postfix[]){
   int i, val, A,B;
   char ch;

   /* evaluate postfix expression */
   for (i = 0; postfix[i] != ')'; i++) {
      ch = postfix[i];
      if (isdigit(ch)) {
         /* we saw an operand,push the digit onto stack
                                                                          */
         push(ch);
      }
```

Evaluation of postfix expression Program

```
   else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
/* we saw an operator  pop top element A and next-to-top
  element B from stack and compute B operator A*/
A = pop();
B = pop();
switch (ch)                    /* ch is an operator */
{
   case '*':   val = B * A;
      break;
   case '/':   val = B / A;
      break;
    case '+':   val = B + A;
       break;
     case '-':      val = B - A;
       break;
}
```

Evaluation of postfix expression Program

```
            /* push the value obtained above onto the stack */
           push(val);
        }
    }
    printf(" \n Result of expression evaluation : %d \n", pop());
}
 int main(){
    int i;
    /* declare character array to store postfix expression */
    char postfix[POSTFIXSIZE];
    printf("ASSUMPTION: There are only four operators(*, /, +, -) in
             an expression and operand is single digit only.\n");
    printf(" \nEnter postfix expression,\n press right parenthesis ')'
              for end expression : ");
```

•

Evaluation of postfix expression Program

```
              /* take input of postfix expression from user */
    for (i = 0; i <= POSTFIXSIZE - 1; i++) {
        scanf("%c", &postfix[i]);
        if (postfix[i] == ')')       {
            break;
        }
    }

    /* call function to evaluate postfix expression */

    EvalPostfix(postfix);

    return 0;
}
```

Evaluation of postfix expression Program [output]

- Enter postfix expression,
- press right parenthesis ')' for end expression : 934*8+4/-)
- Result of expression evaluation : 4

- Enter postfix expression, press right parenthesis ')' for end expression : 456*+)
- Result of expression evaluation : 34
-
- Enter postfix expression, press right parenthesis ')' for end expression: 12345*+*+)
- Result of expression evaluation: 47

Algorithm: Converting Infix to Postfix

- The following algorithm converts infix to postfix.
- Scan input string from left to right character by character.
- If the character is an operand, put it into output stack.
- If the character is an operator and operator's stack is empty, push operator into operators' stack.
- If the operator's stack is not empty, there may be following possibilities.
  - If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operand's stack.
  - If the precedence of scanned operator is less than or equal to the top most operator of operator's stack, pop the operators from operand's stack until we find a low precedence operator than the scanned character. Never pop out (**'('**) or (**')'**) whatever may be the precedence level of scanned character.
  - If the character is opening round bracket (**'('**), push it into operator's stack.
  - If the character is closing round bracket (**')'**), pop out operators from operator's stack until we find an opening bracket (**'('**).
  - Now pop out all the remaining operators from the operator's stack and push into output stack.

Algorithm: Converting Infix to Postfix

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

1. Push “(“onto Stack, and add “)” to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. 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.
    2. Add operator to Stack.
       [End of If]

Algorithm: Converting Infix to Postfix

```
6. If a right parenthesis is encountered ,then:
   1. Repeatedly pop from Stack and add to Y each operator
      (on the top of Stack) until a left parenthesis is
      encountered.
   2. Remove the left Parenthesis.
      [End of If]
      [End of If]
7. END.
```

Algorithm: Converting Infix to Postfix

- **Let's take an example**
- Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

A+ (B*C-(D/E^F)*G)*H Postfix Expression: ABC*DEF^/G*-H*+

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	‘*’ is at higher precedence than ‘-‘
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that’s why ‘^’ Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that’s why ‘^’ Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Program Converts Infix Expression To Postfix Expression.

```
/* This program converts infix expression to postfix expression. This
program assume that there are  four operators: (*, /, +, -)             in infix
expression and operands can be of single-digit only. This program will
not work for fractional numbers. Further this program does not check
whether infix expression is  valid or not in terms of number of operators
and operands.*/


#include<stdio.h>
#include<stdlib.h>      /* for exit() */
#include<ctype.h>     /* for isdigit(char ) */
#include<string.h>
char stack[100];
int top = -1;
 void push(char x){
    stack[++top] = x;
}
```

Program Converts Infix Expression To Postfix Expression.

```
char pop(){
   if(top == -1)
      return -1;
   else
      return stack[top--];
}

int priority(char x){
   if(x == '(')
      return 0;
   if(x == '+' || x == '-')
      return 1;
   if(x == '*' || x == '/')
      return 2;
   return 0;
}
```

Program Converts Infix Expression To Postfix Expression.

```
int main()
{
   char exp[100];
   char *e, x;
   printf("Enter the expression : ");
   scanf("%s",exp);
   printf("\n");
   e = exp;

   while(*e != '\0')
   {
      if(isalnum(*e))
         printf("%c ",*e);
      else if(*e == '(')
         push(*e);
```

Program Converts Infix Expression To Postfix Expression.

```
      else if(*e == ')')
    {
       while((x = pop()) != '(')
          printf("%c ", x);
    }
    else
    {
       while(priority(stack[top]) >= priority(*e))
          printf("%c ",pop());
       push(*e);
    }
 e++;
}
```

Program Converts Infix Expression To Postfix Expression.

```
        while(top != -1)
          {
              printf("%c ",pop());
        }
        return 0;
}
```

Output

```
Enter the expression : a+b*c
a b c * +

Enter the expression : (a+b)*c+(d-a)
a b + c * d a - +

Enter the expression : ((4+8)(6-5))/((3-2)(2+2))
4 8 + 6 5 - 3 2 - 2 2 + /
```

Program Converts Infix Expression To Postfix Expression.

- /* This program converts infix expression to postfix expression. This program assume that there are Five operators: (*, /, +, -,^)   in infix expression and operands can be of single-digit only. This program will not work for fractional numbers. Further this program does not check whether infix expression is valid or not in terms of number of operators and operands.*/
-
- #include<stdio.h>
- #include<stdlib.h>   /* for exit() */
- #include<ctype.h>   /* for isdigit(char) */
- #include<string.h>
- #define SIZE 100
- /* declared here as global variable because stack[] is used by more than one functions */
- char stack[SIZE];
- int top = -1;

Program Converts Infix Expression To Postfix Expression.

```
/* define push operation */

void push(char item)
{
        if(top >= SIZE-1)
        {
                printf("\nStack Overflow.");
        }
        else
        {
                top = top+1;
                stack[top] = item;
        }
}
```

Program Converts Infix Expression To Postfix Expression.

```
•   /* define pop operation */
•   char pop() {
•           char item ;
•           if(top <0)          {
•                       printf("stack under flow: invalid infix expression");
•                       getchar();
•                       /* underflow may occur for invalid expression */
•                       /* where ( and ) are not matched */
•                       exit(1);
•           }
•           else      {
•                       item = stack[top];
•                       top = top-1;
•                       return(item);
•           }
•   }
```

Program Converts Infix Expression To Postfix Expression.

```
/* define function that is used to determine whether any symbol is
operator or not (that is symbol is operand) this function returns 1 if
symbol is operator else return 0 */

int is_operator(char symbol)
{
        if(symbol == '^' || symbol == '*' || symbol == '/' || symbol
                == '+' || symbol =='-')
        {
                return 1;
        }
        else
        {
                return 0;
        }
}
```

Program Converts Infix Expression To Postfix Expression.

- /* define function that is used to assign precedence to operator. Here ^ denotes exponent operator. In this function we assume that higher integer value means higher precedence */

```
int precedence(char symbol){
        if(symbol == '^')/* exponent operator, highest precedence*/
        {                   return(3);          }
        else if(symbol == '*' || symbol == '/')
        {                   return(2);          }
        else if(symbol == '+' || symbol == '-')  /* lowest precedence */
        {                   return(1);          }
        else
        {                   return(0);          }
}
```

Program Converts Infix Expression To Postfix Expression.

```
•   void InfixToPostfix(char infix_exp[], char postfix_exp[])
•   {
•           int i, j;
•           char item, x;
•           push('(');                          /* push '(' onto stack */
•           strcat(infix_exp,")");        /* add ')' to infix expression */
•           i=0;
•           j=0;
•           item=infix_exp[i];        /* initialize before loop*/
•           while(item != '\0')     /* run loop till end of infix expression */
•           {
•                   if(item == '(')
•                   {           push(item);       }

•
•
```

Program Converts Infix Expression To Postfix Expression.

```
•               else if( isdigit(item) || isalpha(item))
•               {
•                       postfix_exp[j] = item;
•                     /* add operand symbol to postfix expr */
•                       j++;
•               }
•               else if(is_operator(item) == 1)
•                   /* means symbol is operator */        {
•                       x=pop();
•                       while(is_operator(x) == 1 &&
                           precedence(x)>= precedence(item))    {
•                               postfix_exp[j] = x;
•                             /* so pop all higher precedence operator and */
•                               j++;
•                               x = pop();
•                               /* add them to postfix expression */
•                       }
```

Program Converts Infix Expression To Postfix Expression.

```
•                               push(x);
•                               /* because just above while loop will
                      terminate we have opped one extra item
•                               for which condition fails and loop terminates,
                      so that one*/
•
•                               push(item);
•                                /* push current operator symbol onto stack */
•                     }
•                     else if(item == ')')     /* if current symbol is ')' then */
•                     {
•                               x = pop();     /* pop and keep popping until */
•                               while(x != '(')          /* '(' encounterd */
•                               {         postfix_exp[j] = x;
•                                         j++;
•                                         x = pop();     }
•                     }
```

Program Converts Infix Expression To Postfix Expression.

```
                    else
                    { /* if current symbol is neither operand not '(' nor ')'
               and nor       operator */
                              printf("\nInvalid infix Expression.\n");
                         /* the it is illegal  symbol */
                              getchar();
                              exit(1);
                    }
                    i++;
                    item = infix_exp[i];
                    /* go to next symbol of infix expression */
          } /* while loop ends here */
```

Program Converts Infix Expression To Postfix Expression.

```
•         if(top>0)        {
•                 printf("\nInvalid infix Expression.\n");
•                  /* the it is illegal  symbol */
•                 getchar();
•                 exit(1);          }
•                 }
•         postfix_exp[j] = '\0'; /* add sentinel else puts() function */
•         /* will print entire postfix[] array up to SIZE */
•
•   }
```

Program Converts Infix Expression To Postfix Expression.

```
/* main function begins */
int main()
{
        char infix[SIZE], postfix[SIZE];
        /* declare infix string and postfix string */

        /* why we asked the user to enter infix expression in
        parentheses ( ) What changes are required in program to get
        rid of this restriction since it is not in algorithm */

        printf("ASSUMPTION: The infix expression contains single
          letter variables and single digit constants only.\n");
        printf("\nEnter Infix expression : ");
        gets(infix);
```

Program Converts Infix Expression To Postfix Expression.

```
        InfixToPostfix(infix,postfix);
            /* call to convert */
        printf("Postfix Expression: ");
        puts(postfix);                  /* print postfix expression */

        return 0;
}
```

- Output

```
Enter Infix expression : A+(B*C-(D/E^F)*G)*H
Postfix Expression: ABC*DEF^/G*-H*+

Enter Infix expression : (3^2*5)/(3*2-3)+5
Postfix Expression: 32^5*32*3-/5+
```

Algorithm Infix to Prefix

1. Push “)” onto STACK, and add “(“ to end of the A
2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
3. If an operand is encountered add it to B
4. If a right parenthesis is encountered push it onto STACK
5. If an operator is encountered then:
   - Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
   - Add operator to STACK
6. If left parenthesis is encountered then
   - Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered)
   - Remove the left parenthesis
7. Exit

Infix to prefix conversion
Algorithm: Alternative method

- Expression = A + (B * C - (D / E ^ F) * G) * H
- **Step 1.** Reverse the infix expression.
  H *) G *) F ^ E / D (- C * B (+ A
- **Step 2.** Make Every '(' as ')' and every ')' as '('
  H * (G * (F ^ E / D) - C * B) + A
- **Step 3.** Convert expression to postfix form.
  HGFE^D/∗CB∗-∗A+
- **Step 4.** Reverse the postfix expression to get prefix form.
  +A*-*BC*/D^EFGH

Infix to Prefix Program

```
/* This program converts infix expression to prefix expression.
This program assume that there are Five operators: (*, /, +, -,^).
This program will not work for fractional numbers. Further this
program does not check whether infix expression is valid or not
in terms of number of operators and operands.*/
#include<stdio.h>
#include<stdlib.h> // for exit() function
#include<ctype.h> // for isdigit(char)function
#include<string.h>
#define SIZE 100

// Global Variable Declaration
char stack[SIZE];
int top = -1;
```

Infix to Prefix Program

-
  ```
  //Global Function Declaration
  void push(char c);
  char pop();
  int isoperator(char symbol);
  int precedence(char symbol);
  void InfixToPrefix(char infix_exp[], char prefix_exp[]);

  // main() function begins
  void main()
  {
          // Declare infix string and prefix string
          char infix[SIZE], prefix[SIZE];
          printf("\n\n Enter Infix expression : ");
          gets(infix);
          InfixToPrefix(infix,prefix); // Call to convert
          printf("\n Prefix Expression: ");
          puts(prefix); }
  ```

Infix to Prefix Program

-
  ```
  void InfixToPrefix(char infix_exp[], char prefix_exp[])
  {
          int i, j, k, pos, len;
          char item, x, rev[SIZE];
          // Reverse the infix expression
          pos=0;
          len=strlen(infix_exp);
          for(k=len-1;k>=0;k--)
          {
                  rev[pos]=infix_exp[k];
                  pos++;
          }
          rev[pos]='\0';
  ```

-
  ```
          strcpy(infix_exp,rev);
  ```

Infix to Prefix Program

-
  ```
  // Make Every “ ( ” as “ ) ” and every “ ) ” as “ ( ”
  for(i=0; infix_exp[i]!='\0'; i++)
  {
          if(infix_exp[i] == ')')
                  infix_exp[i] = '(';
          else if(infix_exp[i] == '(')
                  infix_exp[i] = ')';
  }
  //Convert expression to postfix form.
  // push '(' onto stack
  push('(');
  // add ')' to infix expression
  strcat(infix_exp,")");
  ```

Infix to Prefix Program

```
•         i=0;
          j=0;
          // Initialize before loop
          item=infix_exp[i];
          // Run loop till end of infix expression
          while(item != '\0')      {
                    if(item == '(')
                              {  push(item); }
                    else if( isdigit(item) || isalpha(item))
                    {
                              // Add operand symbol to postfix expression
                              prefix_exp[j] = item;
                              j++;
                    }
```

Infix to Prefix Program

```
•   else if(isoperator(item) == 1)
        {
              // pop all higher precedence operator and
              //add them to postfix expression
              x=pop();
              while(isoperator(x) == 1 && precedence(x)>=
                                precedence(item))
              {
                    prefix_exp[j] = x;
                    j++;
                    x = pop();
              }
              // push the last pop operator symbol onto  stack
              push(x);
              // push current operator symbol onto stack
              push(item); }
```

Infix to Prefix Program

-
  ```
  // if current symbol is ')' then pop and keep popping until '('
  //encounterd
  else if(item == ')')        {
          x = pop();
          while(x != '(') {
                  prefix_exp[j] = x;
                  j++;
                  x = pop();
          }
  }
  else
  {
  // if current symbol is neither operand not '(' nor ')' and nor
  //operator
  printf("\nInvalid infix Expression.\n");
  break;
  }
  ```

Infix to Prefix Program

```
•                       i++;
                        // Go to next symbol of infix expression
                        item = infix_exp[i];
            } //End while loop
            if(top > 0)
                        printf("\n Invalid infix Expression.");
            prefix_exp[j] = '\0';
            // Reverse the prefix expression.
            pos=0;
            len=strlen(prefix_exp);
            for(k=len-1;k>=0;k--)   {
                        rev[pos]=prefix_exp[k];
                        pos++; }
            rev[pos]='\0';
•           strcpy(prefix_exp,rev);
    }
```

Infix to Prefix Program

-
  ```
  // Define push operation
  void push(char c)
  {
          if(top >= SIZE-1)
                  printf("\n Stack Overflow.");
          else
          {
                  top++;
                  stack[top] = c;
          }
  }
  ```

Infix to Prefix Program

```
// Define pop operation
char pop()
{
        char c;
        c='\0';
        if(top < 0)
                printf("\n Stack Underflow.");
        else
        {
                c = stack[top];
                top--;
        }
        return c;
}
```

Infix to Prefix Program

-
  ```
  // Define function that is used to determine whether any
  symbol is //operator or not
  int isoperator(char symbol)
  {
          if(symbol == '^' || symbol == '*' || symbol == '/' || symbol
          == '+' || symbol == '-')
                  return 1;
          else
                  return 0;
  }
  ```

Infix to Prefix Program

-
  ```
  // Define function that is used to assign precedence to operator.
  // In this function we assume that higher integer value means
  //higher precedence
  int precedence(char symbol) {
          if(symbol == '^')
                  return(5);
          else if(symbol == '/')
                  return(4);
          else if(symbol == '*')
                  return(3);
          else if(symbol == '+')
                  return(2);
          else if(symbol == '-')
                  return(1);
          else
                  return(0);
  }
  ```
-

Infix to Prefix Program

-
  ```
  // Define function that is used to assign precedence to operator.
  // In this function we assume that higher integer value means
  //higher precedence
  int precedence(char symbol) {
          if(symbol == '^')
                  return(5);
          else if(symbol == '/')
                  return(4);
          else if(symbol == '*')
                  return(3);
          else if(symbol == '+')
                  return(2);
          else if(symbol == '-')
                  return(1);
          else
                  return(0);
  }
  ```
-

Infix to Prefix Program Output

- Enter Infix expression : A+(B*C-(D/E^F)*G)*H
- Prefix Expression: +A*-*BC*/D^EFGH

Algorithm: EVALUAE_PREFIX

```
1. [Initialize Stack]
     TOP ← 0
     VALUE ← 0

2. [Evaluate the prefix expression]
Repeat from last character up to first
     TEMP ← NEXTCHAR (PREFIX)
     If  TEMP is DIGIT
     Then PUSH (S, TOP, TEMP)
     Else OPERAND1 ← POP (S, TOP)
          OPERAND2 ← POP (S, TOP)
          VALUE ← PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
          PUSH (S, TOP, VALUE)

3. [Return answer from stack]
    Return (POP (S, TOP))
```