MODULE-1

CLASSIFICATION OF DATA STRUCTURES, STACK

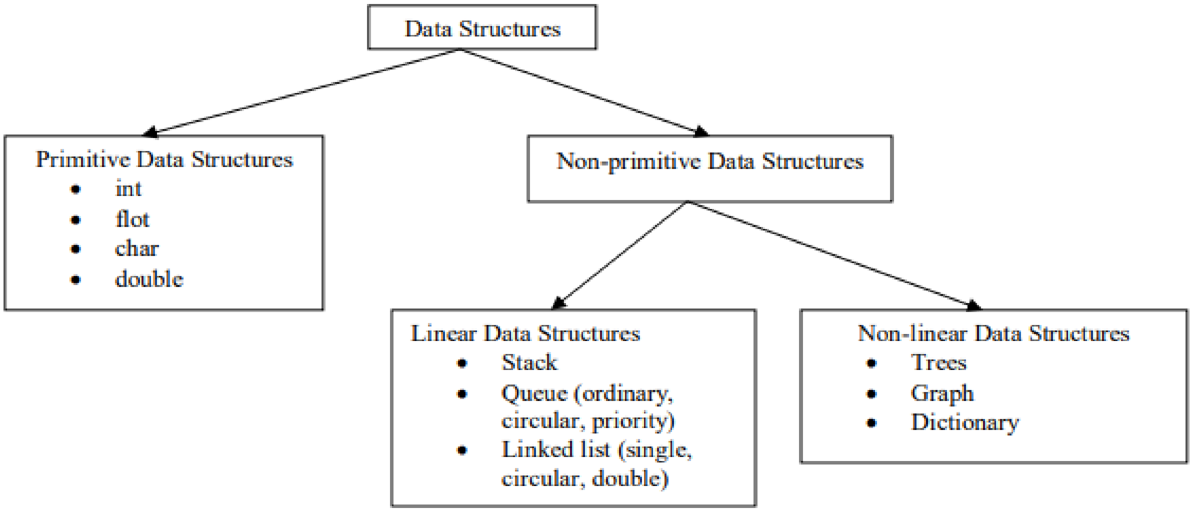
1.1 Classification of data structures

The study of data structures in C deals with the study of **how the data is organized in the memory, how efficiently the data can be retrieved from the memory, how the data is manipulated in the memory and the possible ways in which different data items are logically related.** Thus, we can understand that the study of data structure involves the study of memory as well. Irrespective of the programming language, the structure of the data can be studied. C is one programming language which throws light into in-depth of this concept, as C facilitates hardware interaction and memory management to the programmers.

The study of data structures also involves the study of how to implement the developed data structures using the available data structures in C. Since the problems that arise which implementing high-level data structures are quite complex, the study will allow to investigate the C language more thoroughly and to gain valuable experience in the use of this language. While implementing data structure, one should take care of efficiency, which involves two facts viz. time and space. That is, a careful evaluation of time complexity and space complexity should be made before data structure implementation.

Irrespective of the programming language, the structure of the data can be studied. C is one programming language which throws light into in-depth of this concept, as C facilitates hardware interaction and memory management to the programmers.

Data structures can be categorized as below –



Data Structures
Primitive Data Structures
int
flot
char
double
Non-primitive Data Structures
Linear Data Structures
Stack
Queue (ordinary, circular, priority)
Linked list (single, circular, double)
Non-linear Data Structures
Trees
Graph
Dictionary

Basic data types like int, float etc. are called as **primitive data structures**, because values in are directly accessible. The data types (or data structures) in which values are not directly accessible, instead, they are pointers (or references) are called as **non-primitive data structures.** In C, non-primitive data structures can be further divided into – linear and non-linear data structures. When the relationship between data elements is **linear**, it is called as **linear data structure**. For example, stack, queue, linked list etc. If the relationship between data-elements is **hierarchical,** then it is called as **nonlinear data structure**. For example, trees, graphs, sets, dictionaries etc.

Operations performed on Data structures:

1. **Traversing:** Traversing a Data Structure means to visit the element stored in it. This can be done with any type of DS.

2. **Searching:** Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found. Searching is the operation which we can performed on data-structures like array, linked-list, tree, graph, etc.

3. **Insertion:** It is the operation which we apply on all the data-structures. Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure. It is unsuccessful in some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element.

4. **Deletion:** It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure.

5. **Sorting:** Sorting means arranging the data elements either in ascending or descending order. We have different sorting that helps in arranging the elements in order. For example bubble sort, selection sort, insertion sort, merge sort, quick sort etc.

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

- **Processor speed:** To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.
- **Data Search:** Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.
- **Multiple requests:** If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Advantages of Data Structures

- **Efficiency:** Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.
- **Reusability:** Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
- **Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

1.2 STACKS

Stack is non-primitive linear data structure into which, new items may be inserted and from which items may be deleted. In this data structure, the insertion and deletion of the elements are carried out at one end and is called as 'top' of the stack. In stack, the element last inserted will be the first to be deleted. Hence, stack is known as Last In First Out (LIFO) structure.

For example, consider the task of keeping books on a table one above the other. When a person wants to take the book, he has to take the book which was kept last. Thus, the first book kept on the table will be the last book to be taken out.

In the study of data structures, insertion of new item into the stack is called as ***push*** operation; whereas, deletion of an item at the top of the stack is ***pop*** operation. When the stack is full, we can't insert any more elements. This situation is called as ***stack overflow***. Similarly, when stack is empty, we can't delete element from it. This condition is known as ***stack underflow.***

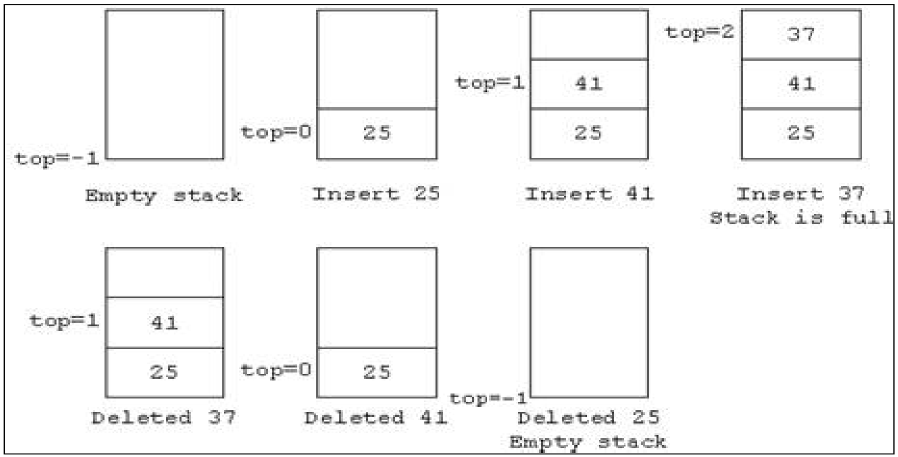


Figure 1.1 Demo of stack operations

An integer variable ***top*** is normally used for denoting current status of the stack – that is, the value of ***top*** gives the number of items of the stack. For programming purpose, an empty stack is denoted by setting the value of ***top*** as -1. Each time the ***push*** operation is encountered, the ***top*** will be incremented. When the ***pop*** operation is done, ***top*** will be decremented. Usually, we define the size of the stack at the beginning. For example, stack of 5 elements, stack of 10 elements etc. When the value of ***top*** becomes -1 during deletion, it is stack underflow. When the ***top*** reaches the predefined size, it is stack overflow.

Figure 1.1 depicts the example of primitive operation on an integer stack of size 3.

Program 1.1 Primitive operations on stack

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void push(int st[], int item, int *t)
{
        if(*t==MAX-1)
        {
                printf("Stack full!!");
                return;
        }
        st[++(*t)]=item;
}

int pop(int st[], int *t)
{
        int item;

        if(*t== -1)
        {
            printf("\nStack Empty!!");
            return ;
        }
        item=st[(*t) - -];
        return item;
}

void disp(int st[], int *t)
{
        int i;

        if(*t== -1)
        {
                printf("\nStack Empty!!");
                return;
        }
        printf("\nStack contents:\n");
        for(i=*t ; i>=0; i- -)
                printf("\n%d",st[i]);
}

void main()
{
        int st[MAX], top= -1, opt, item;

        for(;;)
        {
                printf("\n*****Stack Operations****\n");
                printf("\n1. Push \n 2. Pop \n 3. Display \n 4. Exit\n");
                printf("Enter your option:");
                scanf("%d", &opt);

                switch(opt)
                {
                        case 1: printf("\nEnter item:");
                                scanf("%d",&item);
                                push(st, item, &top);
                                break;
                        case 2: item=pop(st, &top);
                                 printf("\n Deleted item is %d", item);
                                  break;
                        case 3: disp(st, &top);
                                  break;
                        case 4:
                        default:exit(0);
                }
        }
}
```

1.1.1 Application of Stacks

The concept of stacks has various applications in the field of computer science. Some of them includes conversion of arithmetic expressions, evaluation of expressions, recursion etc. Few of these applications are discussed in the following sections.

Conversion of Expressions

Let us consider an arithmetic expression

x op y

Here, x and y are two arithmetic expressions or operands and ‘op’ is the arithmetic operator like +, -, *, / etc. For example, consider a simple arithmetic expression, (x+y). Here x and y

are operands and + is an operator. Representing the equation like this is known as 'infix' expression. There are two more representations for denoting an arithmetic expression: xy+,
known as postfix expression
+xy, known as prefix expression

The words 'pre', 'post' and 'in' specifies the relative position of the operator in the expression.

Thus, any expression having an operator in between two operands is called as an ***infix expression***. Any expression having an operator followed by two operands is ***postfix expression***. An expression, in which the operator precedes the two operands, is a ***prefix expression***.

Note that an infix expression may have parentheses in-between. But, postfix and prefix expressions will not be having any parentheses. While converting a particular infix expression into either prefix or postfix expression, one should consider precedence of operators, as given below:

Precedence	Operator
1	(or) -> parentheses
2	^ or $ -> exponentiation
3	* or / -> Multiplication or division
4	+ or - -> Addition or subtraction

Example:
Convert the following infix expression into postfix and prefix expression:
((A+B)$C-(D/E)*F)

<u>**Solution**</u>*:* ***Conversion into Postfix:***

- The given expression is ((A+B) $ C – (D/E) * F)
- We have to resolve inner brackets first.
- Consider, (A+B). This will be AB+ in postfix. Let P = AB+
- Consider, (D/E). This is DE/ in postfix. Let Q= DE/
- Now, the expression will be (P$C – Q * F)
- Now resolve operators with high precedence. That is $ (power symbol) and * (multiplication)
- That is, P$C = PC$ Let R = PC$
- And, Q * F = QF* Let S= QF*
- Then, expression is (R-S), which when converted into postfix, gives RS –
- Now, by replacing the values of R and S and in turn, the values of P and Q we will get –
    - PC$QF* –
    - AB+C$DE/F*–
- The required postfix expression is **AB+C$DE/F*–**

Conversion into Prefix:

- Consider, ((A+B)$C – (D/E)*F)
- As we did in conversion into postfix notation, here also, we make use of some temporary variables like P, Q etc.
- Consider (A+B). In prefix notation, this will be +AB. Let P = +AB.
- Let (D/E) = /DE = Q.

- Now, the expression is (P$C – Q*F)
- Again, P$C=$PC = R, say and, Q*F = *QF = S
- Then, expression is (R-S) = –RS
- Now, putting the values of P, Q, R and S, we will get –

    ⇨ –$PC*QF

    ⇨ **–$+ABC*/DEF**, is the required prefix expression.

Procedure for converting an Infix expression to Postfix expression programmatically:

- For converting an infix expression to postfix expression, one has to analyze the precedence value of a symbol or element both in input and in stack.
- If an operator is left associative, then the input precedence value is less than the stack precedence value.
- If the operator is right associative, then, the input precedence value is greater than the stack precedence value.

The following table is used for writing an algorithm and program for conversion of infix to postfix.

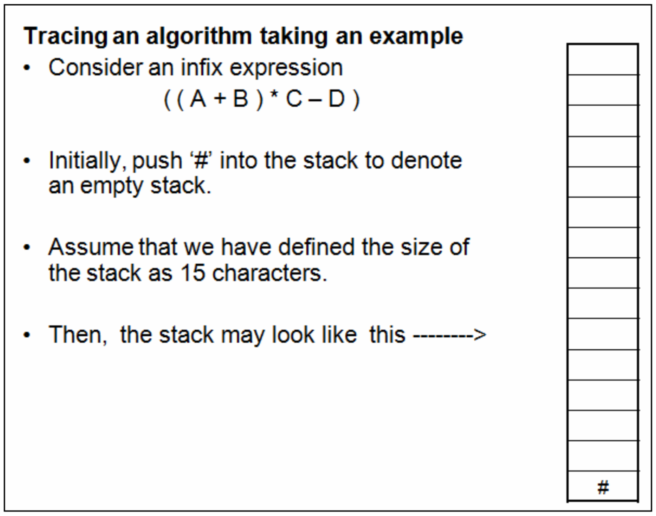
Symbols	**Input precedence (IP)**	**Stack precedence (SP)**
+ or –	1	2
* or /	3	4
$ or ^	6	5
Operands	7	8
(	9	0
)	0	-
#	-	–1

Algorithm/Pseudo code for converting infix expression into postfix expression:

```
1. Initialize empty stack with symbol ‘#’.
       i.e. st[0]=‘#’
2. Read the character from infix expression.
              i.e. symbol=infix[ i]
```

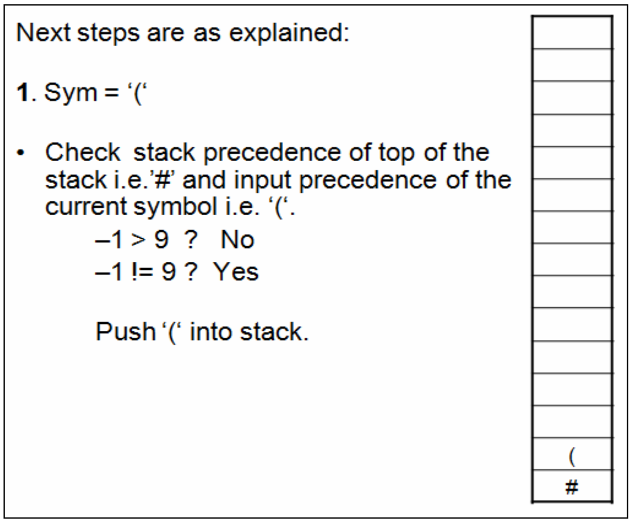
```
3.  while SP(st[top]) > IP(symbol)
            postfix [ j]=pop(st[top])

4.  if SP(st[top]) != IP(symbol)
                   push(symbol)
    else
                   pop(st[top])

5.  Repeat the steps (2) to (4) till the last character of infix expression.

6.   While stack becomes empty,              //only for partially parenthesized expression
         postfix [ j]=pop(st[top])
```

Observe the following figures to understand the tracing of the above algorithm.



Tracing an algorithm taking an example

- Consider an infix expression
  ((A + B) * C – D)
- Initially, push '#' into the stack to denote an empty stack.
- Assume that we have defined the size of the stack as 15 characters.
- Then, the stack may look like this -------->

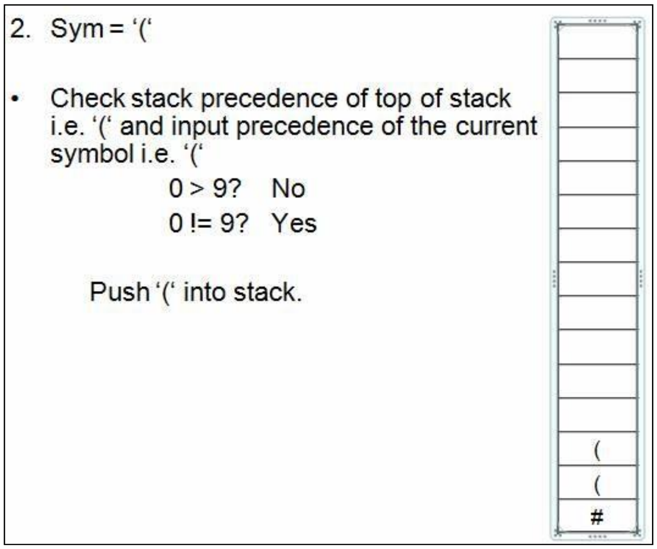


Next steps are as explained:

1. Sym = '('

- Check stack precedence of top of the stack i.e.'#' and input precedence of the current symbol i.e. '('.
  –1 > 9 ? No
  –1 != 9 ? Yes

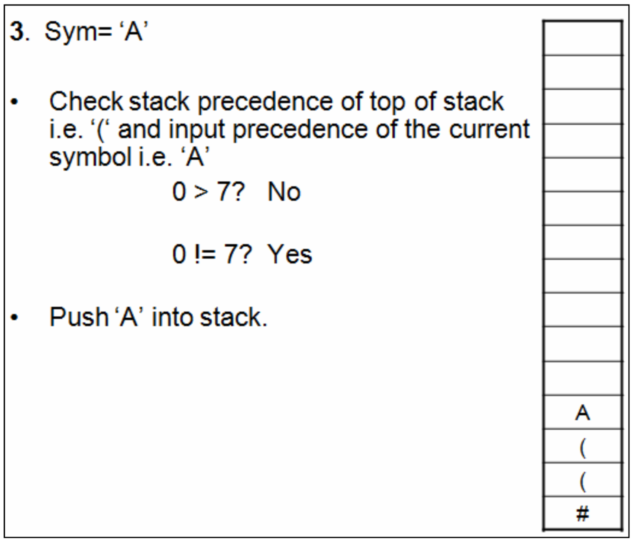
  Push '(' into stack.



2. Sym = '('

- Check stack precedence of top of stack i.e. '(' and input precedence of the current symbol i.e. '('
  0 > 9? No
  0 != 9? Yes

  Push '(' into stack.

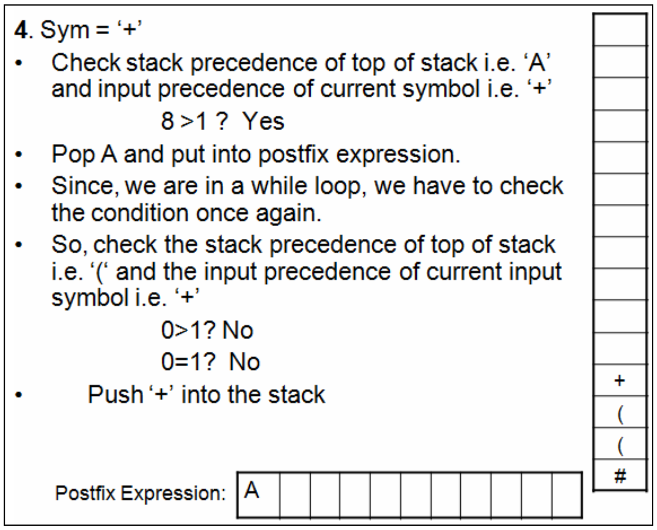


3. Sym= 'A'

- Check stack precedence of top of stack i.e. '(' and input precedence of the current symbol i.e. 'A'
  0 > 7? No

  0 != 7? Yes

- Push 'A' into stack.



4. Sym = '+'

- Check stack precedence of top of stack i.e. 'A' and input precedence of current symbol i.e. '+'

  8 >1 ? Yes
- Pop A and put into postfix expression.
- Since, we are in a while loop, we have to check the condition once again.
- So, check the stack precedence of top of stack i.e. '(' and the input precedence of current input symbol i.e. '+'

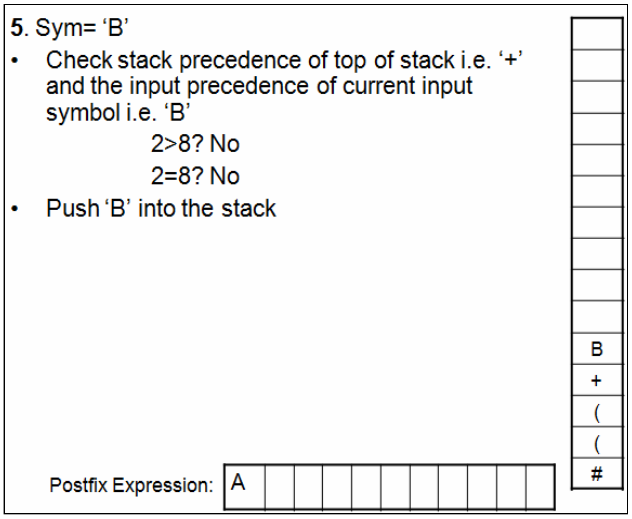
  0>1? No

  0=1? No
- Push '+' into the stack

+
(
(
#

Postfix Expression:

A										



5. Sym= 'B'

- Check stack precedence of top of stack i.e. '+' and the input precedence of current input symbol i.e. 'B'

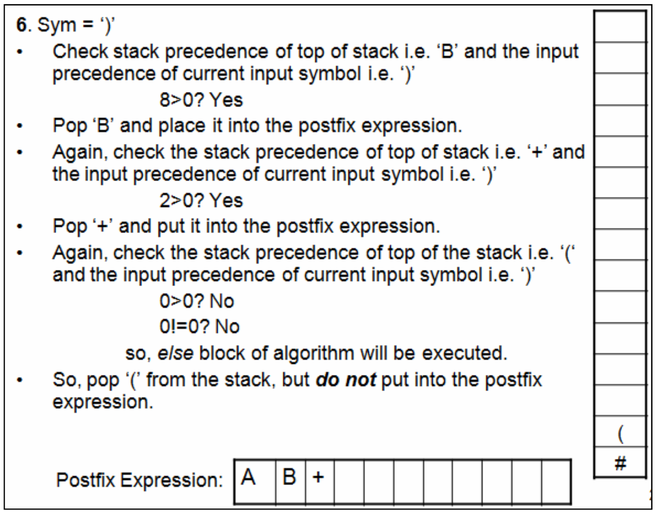
  2>8? No

  2=8? No
- Push 'B' into the stack

B
+
(
(
#

Postfix Expression:

A										



6. Sym = ')'

- Check stack precedence of top of stack i.e. 'B' and the input precedence of current input symbol i.e. ')'

  8>0? Yes
- Pop 'B' and place it into the postfix expression.
- Again, check the stack precedence of top of stack i.e. '+' and the input precedence of current input symbol i.e. ')'

  2>0? Yes
- Pop '+' and put it into the postfix expression.
- Again, check the stack precedence of top of the stack i.e. '(' and the input precedence of current input symbol i.e. ')'

  0>0? No

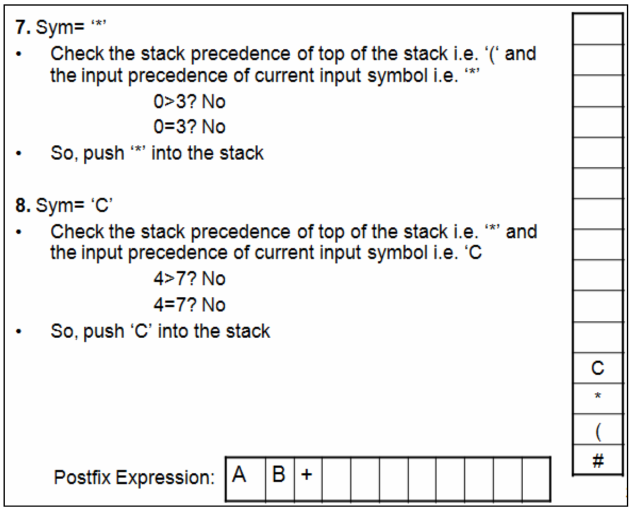
  0!=0? No

  so, *else* block of algorithm will be executed.
- So, pop '(' from the stack, but ***do not*** put into the postfix expression.

(
#

Postfix Expression:

A	B	+							



7. Sym= '*'

- Check the stack precedence of top of the stack i.e. '(' and the input precedence of current input symbol i.e. '*'

  0>3? No

  0=3? No
- So, push '*' into the stack

8. Sym= 'C'

- Check the stack precedence of top of the stack i.e. '*' and the input precedence of current input symbol i.e. 'C

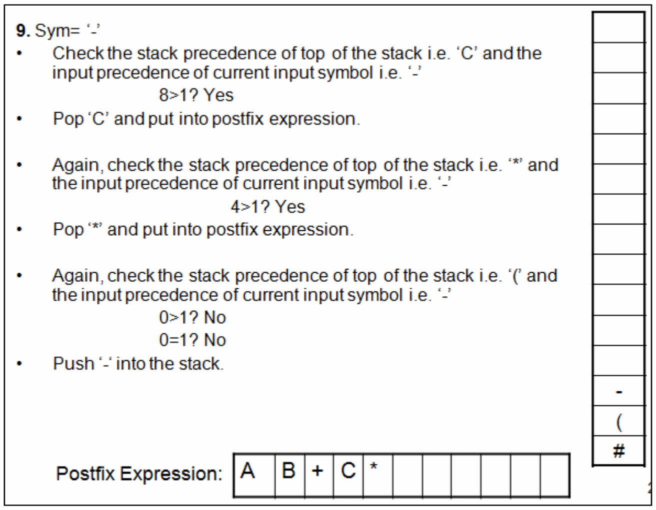
  4>7? No

  4=7? No
- So, push 'C' into the stack

C
*
(
#

Postfix Expression:

A	B	+								



9. Sym= '-'

- Check the stack precedence of top of the stack i.e. 'C' and the input precedence of current input symbol i.e. '-'

  8>1? Yes
- Pop 'C' and put into postfix expression.
- Again, check the stack precedence of top of the stack i.e. '*' and the input precedence of current input symbol i.e. '-'

  4>1? Yes
- Pop '*' and put into postfix expression.
- Again, check the stack precedence of top of the stack i.e. '(' and the input precedence of current input symbol i.e. '-'

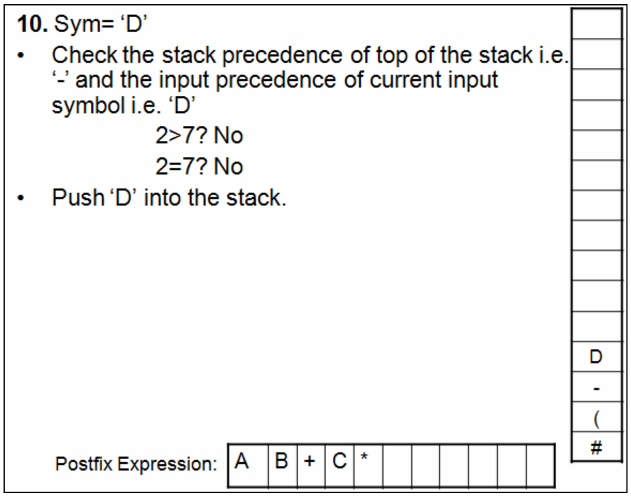
  0>1? No

  0=1? No
- Push '-' into the stack.

-
(
#

Postfix Expression:

A	B	+	C	*						



10. Sym= 'D'

- Check the stack precedence of top of the stack i.e. '-' and the input precedence of current input symbol i.e. 'D'

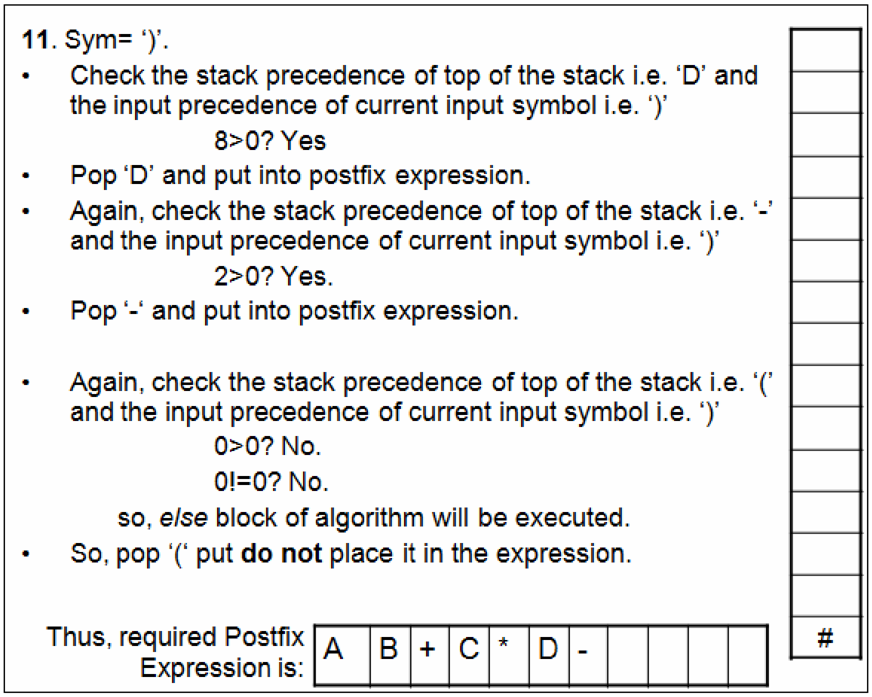
  2>7? No

  2=7? No
- Push 'D' into the stack.

D
-
(
#

Postfix Expression:

A	B	+	C	*						



11. Sym= ‘)’.

- Check the stack precedence of top of the stack i.e. ‘D’ and the input precedence of current input symbol i.e. ‘)’

  8>0? Yes
- Pop ‘D’ and put into postfix expression.
- Again, check the stack precedence of top of the stack i.e. ‘-’ and the input precedence of current input symbol i.e. ‘)’

  2>0? Yes.
- Pop ‘-‘ and put into postfix expression.
- Again, check the stack precedence of top of the stack i.e. ‘(’ and the input precedence of current input symbol i.e. ‘)’

  0>0? No.

  0!=0? No.

  so, *else* block of algorithm will be executed.
- So, pop ‘(‘ put **do not** place it in the expression.

Thus, required Postfix Expression is:

A	B	+	C	*	D	-				

#

Program 1.2 Converting a valid infix expression into the postfix expression:

```
#include<stdio.h>
#include<conio.h>

int inputpre(char sym)          //Function for input precedence
{
        switch(sym)
        {
                case '+' :
                case '-' : return 1;
                case '*' :
                case '/' : return 3;
                case '^' :
                case '$' : return 6;
                case '(' : return 9;
                case ')' : return 0;
                default : return 7;
        }
}

int stackpre(char sym)          //Function for stack precedence
{
        switch(sym)
        {
                case '+' :
```

```
                case '-' : return 2;
                case '*' :
                case '/' : return 4;
                case '^' :
                case '$' : return 5;
                case '(' : return 0;
                case '#' : return -1;
                default : return 8;
        }
}

void push (char item, int *top, char s[])
{
        s[++(*top)] = item;
}

char pop(int *top, char s[])
{
          return s[(*top)--];
}

void infix_to_postfix (char ifix[], char pfix[])
{
        int top = -1, i, j = 0;
        char s[30] , sym;

        push('#',&top,s);

        for(i=0;i < strlen(ifix);i++)
        {
                sym = ifix[i];
                while (stackpre(s[top]) > inputpre(sym))
                        pfix[j++] = pop(&top,s);

                if(stackpre(s[top]) != inputpre(sym))
                        push(sym,&top,s);
                else
                        pop(&top,s);
        }

        while(s[top] != '#')
                pfix[j++] = pop(&top,s);

        pfix[j] = '\0';
}
```

```
void main()
{
        char ifix[20], pfix[20];
        clrscr();

        printf("Enter valid infix expression\n");
        scanf("%s", ifix);
        infix_to_postfix (ifix,pfix);
        printf("The postfix expression is = %s", pfix);
}
```

Procedure for converting an Infix expression to Prefix expression:
To convert an infix expression into prefix expression, we have to use the following precedence table.

Symbols	Input precedence (IP)	Stack precedence (SP)
+ or -	2	1
* or /	4	3
$ or ^	5	6
Operands	7	8
(	0	--
)	9	0
#	-	-1

Algorithm: The algorithm to convert an infix expression into prefix expression is given below:

- Reverse the given infix expression.
- Follow the steps used for obtaining postfix expression, but use the above precedence table.
- Reverse the result obtained.

Program 1.3 Converting a valid infix expression into the prefix expression:

```
#include<stdio.h>
#include<conio.h>

int inputpre(char sym)
{
        switch(sym)
        {
                case '+' :
                case '-' : return 2;
                case '*' :
                case '/' : return 4;
                case '^' :
                case '$' : return 5;
                case '(' : return 0;
                case ')' : return 9;
                default : return 7;
        }
}

int stackpre(char sym)
{
        switch(sym)
        {
                case '+' :
                case '-' : return 1;
                case '*' :
                case '/' : return 3;
                case '^' :
                case '$' : return 6;
                case ‘)' : return 0;
                case '#' : return -1;
                default : return 8;
        }
}

void push (char item, int *top, char s[])
{
        s[++(*top)] = item;
}

char pop(int *top, char s[])
{
          return s[(*top)--];
}

void infix_to_prefix (char ifix[], char pfix[])
{
        int top = -1, i, j = 0;
        char s[30] , sym;

        push('#',&top,s);
        strrev(ifix);

        for(i=0;i < strlen(ifix);i++)
        {
                sym = ifix[i];
                while (stackpre(s[top]) > inputpre(sym))
                        pfix[j++] = pop(&top,s);
                if(stackpre(s[top]) != inputpre(sym))
                        push(sym,&top,s);
                else
                        pop(&top,s);
        }

        while(s[top] != '#')
                pfix[j++] = pop(&top,s);

        pfix[j] = '\0';
        strrev(pfix);
}

void main()
{
        char ifix[20], pfix[20];
        clrscr();

        printf("Enter valid infix expression\n");
        scanf("%s", ifix);
        infix_to_prefix (ifix,pfix);
        printf("The prefix expression is = %s", pfix);
}
```

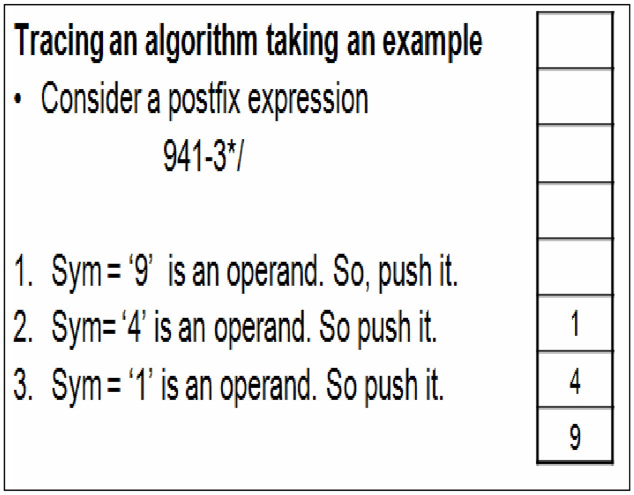
Evaluation of Postfix expression:
To evaluate an infix expression, we will scan from left to right repeatedly. But if the expression contains parentheses, evaluation becomes complex as the parentheses changes the order of precedence. So, it is always easy to evaluate an infix expression by converting it into either prefix or postfix expression.

Algorithm to evaluate postfix expression:

1. Scan a symbol in postfix expression from left to right.
2. If the symbol is operand, push it into stack.
3. If the symbol is an operator, pop two elements from the stack and perform the operation indicated.
4. Push the result of step (3) into the stack.
5. Repeat the above steps till all the symbols get exhausted in the given postfix expression.
6. Now, pop the element from the stack, which will be the result of entire postfix expression.

Note that, here, a single digit is treated as an operand and scanned.

Observe the following figures to understand the working of above algorithm.



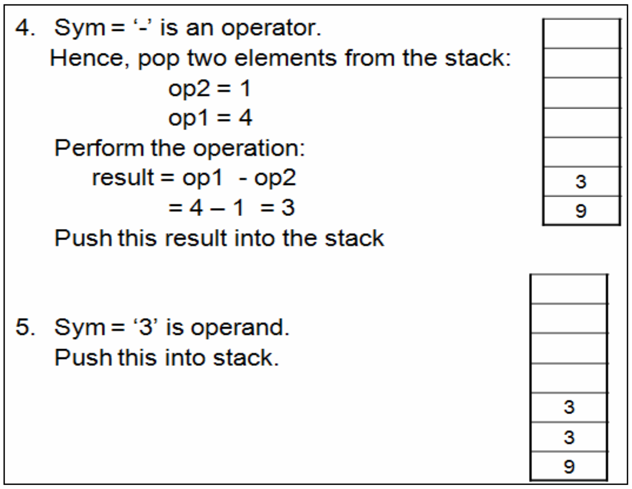
Tracing an algorithm taking an example

- Consider a postfix expression

  941-3*/

1. Sym = '9' is an operand. So, push it.
2. Sym= '4' is an operand. So push it.
3. Sym = '1' is an operand. So push it.

1
4
9

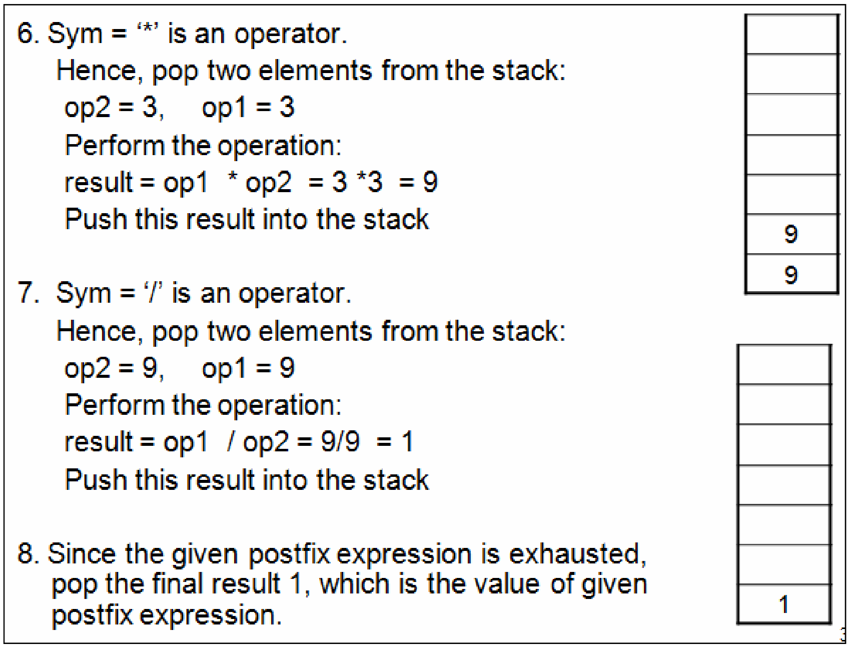


4. Sym = '-' is an operator.
   Hence, pop two elements from the stack:
   op2 = 1
   op1 = 4
   Perform the operation:
   result = op1 - op2
   = 4 – 1 = 3
   Push this result into the stack

3
9

5. Sym = '3' is operand.
   Push this into stack.

3
3
9



6. Sym = '*' is an operator.
   Hence, pop two elements from the stack:
   op2 = 3, op1 = 3
   Perform the operation:
   result = op1 * op2 = 3 *3 = 9
   Push this result into the stack

9
9

7. Sym = '/' is an operator.
   Hence, pop two elements from the stack:
   op2 = 9, op1 = 9
   Perform the operation:
   result = op1 / op2 = 9/9 = 1
   Push this result into the stack

8. Since the given postfix expression is exhausted, pop the final result 1, which is the value of given postfix expression.

1

Program 1.4 Evaluating a valid postfix expression:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

float oper(char sym, float op1, float op2)
{
        switch(sym)
        {
                case '+': return op1 + op2;
                case '-': return op1 - op2;
                case '*': return op1 * op2;
                case '/': if(op2== 0)
                          {
                            printf(“Can’t evalueate”);
                            exit(0);
                          }
                          return op1 / op2;
                case '^':
                case '$': return pow(op1,op2);
        }
}

void push(float item, int *top, float s[ ])
{
        s[++(*top)] = item;
}

float pop(int *top, float s[ ])
{
        return s[(*top)--];
}

void main()
{
        float s[20], result, op1, op2, x;
        int top = -1, i;
        char postfix[20], sym;

        printf("Enter valid postfix expression\n");
        scanf("%s",postfix);

        for(i=0;i<strlen(postfix);i++)
        {
                sym = postfix[i];

                if(isdigit(sym))
                        push(sym-'0', &top, s);   // character to digit conversion
                else if (isalpha(sym))
                {
                        printf("Enter the value of %c: ", sym);
                        scanf("%f",&x);
                        push(x,&top,s);
                }
```

```
                    else
                    {
                            op2 = pop(&top,s);
                            op1 = pop(&top,s);
                                  result = oper(sym,op1,op2); push(result,&top,s);

                    }
          }
          result = pop(&top,s);
          printf("Result =%.4f",result);
}
```

Sample Output 1:

```
Enter valid postfix expression: 941-3*/
Result = 1.0000
```

Sample Output 2:

```
Enter valid postfix expression: abc-d*/
Enter value of a:  -54
Enter value of b: 23
Enter value of c: 5
Enter value of d: 8
Result = -0.3750
```

NOTE: The **sample output 1** takes the postfix expression with digits. Hence, each digit has to be treated as one operand. Whereas, in the **Sample Output 2,** the postfix expression is a series of alphabets (variables). So, the program will allow you to read the values for each of these variables. And hence, the user can give an operand containing more than one digit, a floating point number, a negative number etc. In the **Sample Output2,** the meaning of expression (in infix format) will be –

- 54 / ((23-5)*8) = - 0.3750.

POSTFIX to INFIX Expression:

Postfix expression is an expression in which the operator is after operands, like **operand operator.**

Postfix expressions are easily computed by the system but are not human readable. So this conversion is required. Generally reading and editing by the end-user is done on infix notations as they are parenthesis separated hence easily understandable for humans.

Let’s take an example to understand the problem

Input − xyz/*

Output − (x * (y/z))

To solve this problem, we will use the stack data structure.

Algorithm For converting postfix expression to infix expression:

Step 1: Read the symbol from the given postfix expression from left to right.
Step 2: if symbol is operand, then push the symbol in to the stack.
Step 3: if symbol is operator pop the two operands and perform the operation and push the result into the stack (s2 operator s1) format.
Step 4: Repeat the step from 1 to 3 until you finish final symbol in the given expression.

Example: **xyz/***
Using the algorithm we will the solve the above postfix expression:

1. First symbol we read is **'x'**
2. Since symbol is an operand we are pushing operand into the stack.

x

3. Next we read the symbol which is **'y'**
4. Again the symbol is operand we push that into the stack

y
x

5. Next we read the symbol which is **'z'**
6. Again symbol would found to be operand push it into stack

z
y
x

7. Next we read **'/'** symbol which is operator now we need to follow step 3 in The algorithm i.e., we need to pop two operands and perform the operation. First we will pop z and then y now we are going to perform Division operation in the format s2 operator s1 i.e., **y/z** where **s2=y and s1=z** After performing the operation again push the result into the stack

y/z
x

8. Now read the symbol which is **'*'** since it is an operator pop two operand and perfrom the operation now **s1=y/z** and **s2=x , x*(y/z)** now push the result into stack.

X*(y/z)

Since we don't any symbol to read in the expression, pop the top of the stack which is final infix expression

The infix expression for above postfix expression is **: x*(y/z)**