STACK:

A stack is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the top of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called Last-in-First-out (LIFO). Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.
The operation of the stack can be illustrated as in Fig. 3.1.

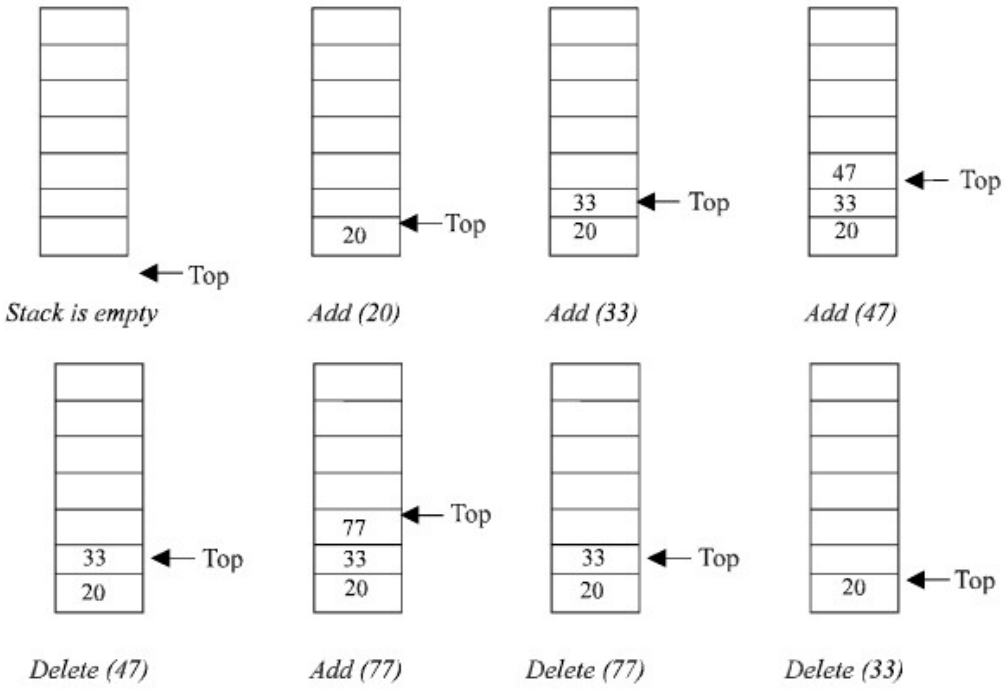

Fig. 3.1. Stack operation.

The insertion (or addition) operation is referred to as push, and the deletion (or remove) operation as pop. A stack is said to be empty or underflow, if the stack contains no elements. At this point the top of the stack is present at the bottom of the stack. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:
PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.
POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

STACK IMPLEMENTATION

Stack can be implemented in two ways:
1. Static implementation (using arrays)
2. Dynamic implementation (using pointers)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (*i.e.,* increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (*i.e.,* memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity.

The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

STACK USING ARRAYS

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

Algorithm for push

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE – 1, then:
(*a*) Display "The stack is in overflow condition"
(*b*) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit

Algorithm for pop

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

1. If TOP < 0, then
(*a*) Display "The Stack is empty"
(*b*) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5.Exit.

```
//THIS PROGRAM IS TO DEMONSTRATE THE OPERATIONS PERFORMED ON THE STACK AND IT IS
//IMPLEMENTATION USING ARRAYS
#include<stdio.h>
#include<conio.h>
//Defining the maximum size of the stack
#define MAXSIZE 100
//Declaring the stack array and top variables in a structure
struct stack
{
int stack[MAXSIZE];
int Top;
};
//type definition allows the user to define an identifier that would represent an existing data type. The user-defined data type
//identifier can later be used to declare variables.
typedef struct stack NODE;

//This function will add/insert an element to Top of the stack
void push(NODE *pu)
```

```
{
int item;
//if the top pointer already reached the maximum allowed size then we can say that the stack is full or overflow
if (pu->Top == MAXSIZE–1)
{
printf(“\nThe Stack Is Full”);
getch();
}
//Otherwise an element can be added or inserted by incrementing the stack pointer Top as follows
else
{
printf(“\nEnter The Element To Be Inserted = ”);
scanf(“%d”,&item);
pu->stack[++pu->Top]=item;
}
}
//This function will delete an element from the Top of the stack
void pop(NODE *po)
{
int item;
//If the Top pointer points to NULL, then the stack is empty That is NO element is there to delete or pop
if (po->Top == -1)
printf(“\nThe Stack Is Empty”);
//Otherwise the top most element in the stack is popped or deleted by decrementing the Top pointer
else
{
item=po->stack[po->Top--];
printf(“\nThe Deleted Element Is = %d”,item);
}
}
//This function to print all the existing elements in the stack
void traverse(NODE *pt)
{
int i;
//If the Top pointer points to NULL, then the stack is empty. That is NO element is there to delete or pop
if (pt->Top == -1)
printf(“\nThe Stack is Empty”);
//Otherwise all the elements in the stack is printed
else
{
printf(“\n\nThe Element(s) In The Stack(s) is/are...”);
for(i=pt->Top; i>=0; i--)
printf (“\n %d”,pt->stack[i]);
}
}
void main( )
{
int choice;
char ch;
//Declaring an pointer variable to the structure
NODE *ps;
//Initializing the Top pointer to NULL
ps->Top=–1;
do
{
clrscr();
//A menu for the stack operations
printf(“\n1. PUSH”);
printf(“\n2. POP”);
printf(“\n3. TRAVERSE”);
printf(“\nEnter Your Choice = ”);
scanf (“%d”, &choice);
```

```
switch(choice)
{
case 1://Calling push() function by passing
//the structure pointer to the function
push(ps);
break;
case 2://calling pop() function
pop(ps);
break;
case 3://calling traverse() function
traverse(ps);
break;
default:
printf("\nYou Entered Wrong Choice") ;
}
printf("\n\nPress (Y/y) To Continue = ");
//Removing all characters in the input buffer for fresh input(s), especially <<Enter>> key
fflush(stdin);
scanf("%c",&ch);
}while(ch == 'Y' || ch == 'y');
}
```

APPLICATIONS OF STACKS

There are a number of applications of stacks; three of them are discussed briefly in the preceding sections. Stack is internally used by compiler when we implement (or execute) any recursive function. If we want to implement a recursive function non-recursively, stack is programmed explicitly. Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

RECURSION

Recursion occurs when a function is called by itself repeatedly; the function is called recursive function. The general algorithm model for any recursive function contains the following steps:

1. *Prologue:* Save the parameters, local variables, and return address.
2. *Body:* If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call).
3. *Epilogue:* Restore the most recently saved parameters, local variables, and return address.

RECURSION vs ITERATION

Recursion of course is an elegant programming technique, but not the best way to solve a problem, even if it is recursive in nature. This is due to the following reasons:

1. It requires stack implementation.
2. It makes inefficient utilization of memory, as every time a new recursive call is made a new set of local variables is allocated to function.
3. Moreover it also slows down execution speed, as function calls require jumps, and saving the current state of program onto stack before jump.

Though inefficient way to solve general problems, it is too handy in several problems as discussed in the starting of this chapter. It provides a programmer with certain pitfalls, and quite sharp concepts about programming. Moreover recursive functions are often easier to implement d maintain, particularly in case of data structures which are by nature recursive. Such data structures are queues, trees, and linked lists. Given below are some of the important points, which differentiate iteration from recursion.

No	*Iteration*	*Recursion*
1	It is a process of executing a statement or a set of statements repeatedly, until some specified condition is specified.	Recursion is the technique of defining anything in terms of itself.

2	Iteration involves four clear-cut Steps like initialization, condition, execution, and updating.	There must be an exclusive if statement inside the recursive function, specifying stopping condition.
3	Any recursive problem can be solved iteratively.	Not all problems have recursive solution.
4	Iterative counterpart of a problem is more efficient in terms of memory utilization and execution speed.	Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.

DISADVANTAGES OF RECURSION

1. It consumes more storage space because the recursive calls along with automatic variables are stored on the stack.
2. The computer may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed and execution time.
4. According to some computer professionals, recursion does not offer any concrete advantage over non-recursive procedures/functions.
5. If proper precautions are not taken, recursion may result in non-terminating iterations.
6. Recursion is not advocated when the problem can be through iteration. Recursion may be treated as a software tool to be applied carefully and selectively.

POLISH NOTATION

The process of writing the operators of an expression either before their operands or after their operands is called Polish notation. It was given by Polish mathematician JanLuksiewicz. The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.
There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)
1. Infix notation
2. Prefix notation
3. Postfix notation

The *infix notation* is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as:
A + B
Note that the operator '+' is written in between the operands A and B.

The *prefix notation* is a notation in which the operator(s) is written before the operands,
The same expression when written in prefix notation looks like:
+ A B
As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the *postfix notation* the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as *suffix notation* or *reverse polish notation*. The above expression if written in postfix expression looks like:
A B +
The prefix and postfix notations are not really as awkward to use as they might look.
For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction: add(A, B)
Note that the operator *add* (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

AdvantagesF of using postfix notation

Human beings are quite used to work with mathematical expressions in *infix* notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS , and associativity. Using infix notation, one cannot tell the order in which operators should be applied. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first. But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

Meaning of BODOMAS

B	**B**rackets first
O	**O**rders (ie Powers and Square Roots, etc.)
DM	**D**ivision and **M**ultiplication (left-to-right)
AS	**A**ddition and **S**ubtraction (left-to-right)

Divide and Multiply rank equally (and go left to right).

Add and Subtract rank equally (and go left to right)

After you have done "B" and "O", just go from left to right doing any "D" ***or*** "M" as you find them.

Then go from left to right doing any "A" ***or*** "S" as you find them.

Examples

Example: How do you work out **3 + 6 × 2** ?
Multiplication before **A**ddition:
First **6 × 2 = 12**, then **3 + 12 = 15**

Example: How do you work out **(3 + 6) × 2** ?
Brackets first: First **(3 + 6) = 9**, then **9 × 2 = 18**

Example: How do you work out **12 / 6 × 3 / 2** ?
Multiplication and **D**ivision rank equally, so just go left to right:
First **12 / 6 = 2**, then **2 × 3 = 6**, then **6 / 2 = 3**

Oh, yes, and what about 7 + (6 × 5^2 + 3) ?

$7 + (6 \times 5^2 + 3)$	
$7 + (6 \times 25 + 3)$	Start inside Brackets, and then use "Orders" First
$7 + (150 + 3)$	Then Multiply
$7 + (153)$	Then Add
$7 + 153$	Brackets completed, last operation is add
160	DONE

Notation Conversions

Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

A + B * C = 4 + 3 * 7 = 7 * 7 = 49

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

CONVERTING INFIX TO POSTFIX EXPRESSION

The method of converting infix expression A + B * C to postfix form is:

A + B * C Infix Form
A + (B * C) Parenthesized expression
A + (B C *) Convert the multiplication
A (B C *) + Convert the addition
A B C * + Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression
B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}
A + { [(BC +) + (DE + F *] / G}
A + { [(BC + (DE + F * +] / G} .
A + [BC + DE + F *+ G /]
ABC + DE + F * + G / + Postfix Form

Give postfix form for (A + B) * C / D + E ^ A / B

Solution. Evaluation order is

[(AB +) * C / D] + [(EA ^) / B]
[(AB +) * C / D] + [(EA ^) B /]

[(AB +) C * D /] + [(EA ^) B /]

(AB +) C * D / (EA ^) B / +
AB + C * D / EA ^ B / + Postfix Form

Algorithm

Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-).

The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

1. Push "(" onto stack, and add")" to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator is encountered, then:
(*a*) Repeatedly pop from stack and add P each operator (on the top of stack), which has the same precedence as, or higher precedence than operator
(*b*) Add operator to stack.
6. If a right parenthesis is encountered, then:
(*a*) Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis is encountered.
(*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit.

EVALUATING POSTFIX EXPRESSION

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator is encountered, then:
(*a*) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
(*b*) Evaluate B operator A.
(*c*) Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.

QUEUE

A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre. It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are
1. Insert (or add) an element to the queue (push)
2. Delete (or remove) an element from a queue (pop)