Chapter

TWO Arrays

LEARNING OBJECTIVES

Having studied this chapter, the students will be able to :

- *Understand the concept of arrays.*
- *The ways arrays are represented in memory.*
- *Various operations performed on linear arrays.*
- *Understand the sparse matrices and representation.*

2.1 INTRODUCTION

This chapter discusses the common data structure known as the array. Arrays are basic building blocks for more complex data structure. We can represent every kind of complex structure using an array. In applications, where we have a small number of elements, we tend to specify separate variable name for each element but when there is large number of elements, it is not possible to define separate variables name for each element. To overcome this problem, the concept of arrays are used. One major feature of the array is that it takes the same amount of time to access any element in the list.

An array is ordered set of homogeneous elements. By ordered set, we mean that each element in the list has unique address and by homogeneous, we mean that each element in the list consists of same data type. Each element of an array is referenced by a subscripted variable.

2.2 TYPES OF ARRAYS

- Single or one dimensional array
- Two dimensional array
- Multi-dimensional array

One Dimensional Arrays

These arrays are also called linear arrays or vectors. If one subscript is required to reference an element, it is called one-dimensional array. The one-dimensional array A is represented as follow :

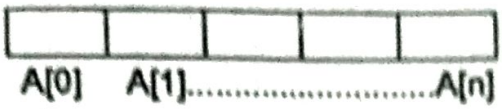


Fig. 2.1 : Rpresentation of 1 – D array

The subscript of an element specifies its position in the array's ordering e.g., A[i] or A[5].

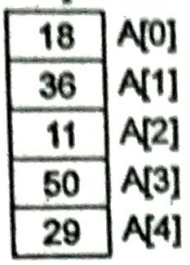


Fig. 2.2 : Insertion of elements

The number of elements is called the size of array. Smallest index value is called LOWER BOUND and largest index value is called UPPER BOUND.

$\therefore$ Size of array = UB – LB + 1.

It is to be noted that the size of array equal to UB if LB = 1.

In the above example, LB = 0, UB = 4

$\therefore$ Size = UB – LB + 1

= 4 – 0 + 1 = 5

Two Dimensional Arrays

Arrays can be multi-dimensional. An array defined to have more than one index is said to be two-dimensional arrays. An array can be 2-dimensional, 3-dimensional, 4-dimensional or N-dimensional.

Two dimensional arrays are also called matrices. These arrays are represented as follow :

	Column 0	Column 1
Row 0	a_{00}	a_{01}
Row 1	a_{10}	a_{11}
Row 2	a_{20}	a_{21}

Fig. 2.3 : Representation of two-dimensional Array

The best way to visualize this array means rows and columns. The first dimension in the array refers to the rows and second dimension refers to the columns. e.g., a[3, 3] is the element of a that is in the 3rd row and 3rd column of array B (as indicated in the fig.).

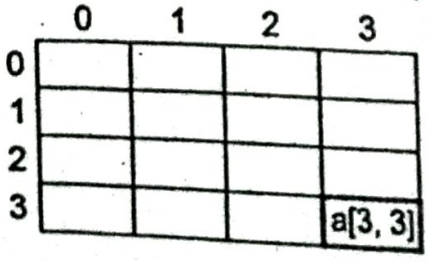


Fig. 2.4 :

2.3 CALCULATING ADDRESSES IN ARRAY

Let us now see how the data represented in an array is actually stored in memory cells of the machine.

One Dimensional Array

Let us suppose a be the linear array with n elements and these elements are stored in consecutive memory location. The system needs to keep track of address of first element only. This address is called base address. The formula for calculating address in one dimensional array is

$$\text{Location [a[i]]} = \text{Base Address} + S(i - LB)$$

Where S is number of words to store one element.

Example 2.1 :

Consider an array with 15 elements. Find the address of A[4] with base address = 200 and the number of bytes to store an element will be 4.

Solution :

If we consider LB = 0,

$$\text{Location [a[4]]} = 200 + 4\ (4 - 0) = 200 + 16 = 216$$

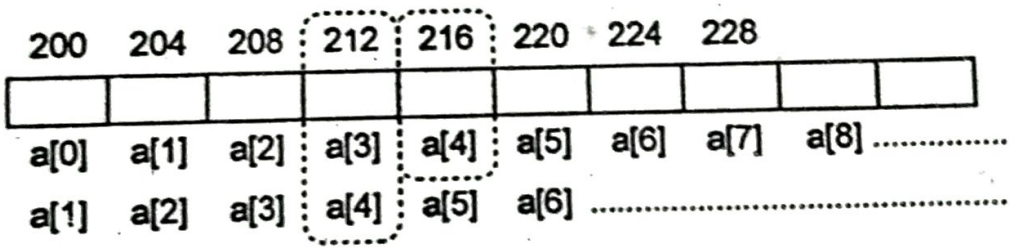


Fig. 2.5 : Address structure

IF we consider LB = 1,

$$A[4] = 200 + 4\ (4 - 1)$$
$$= 200 + 4.3 = 200 + 12 = 212$$

Example 2.2 :

Suppose A is a linear array with LB = 1. Find the address of A[37] if address of A[40] = 525 and A[35] = 500.

Solution :

The formula for determining the address is

$$A[i] = \text{Base Address} + S[i - LB]$$

Now, $a[35] = \text{Base Address} + S[35 - 1] = 500$

and $a[40] = \text{Base Address} + S[40 - 1] = 525$

We get the equations,

$$\text{Base Address} + 34\ S = 500$$

$$\text{Base Address} + 39\ S = 525$$

After solving these equation, we get

$$S = 5$$

So, $\text{Base Address} + 34\ S = 500$

$$\text{Base Address} = 500 - 34S$$

$$= 500 - 34 \times 5$$

$$= 500 - 170$$

$$\text{Base Address} = 330$$

Now,

$$A[37] = 330 + 5\ [37 - 1].$$

$$= 330 + 5 \times 36$$

$$= 330 + 180$$

$$= 510$$

Two Dimensional Array

Let us suppose a be two-dimensional array. The elements can be stored in two different ways.

(a) Column Major Order : In this order, the elements are stored column-by-column, that is first store the first column, then the second column, the third column and so on.

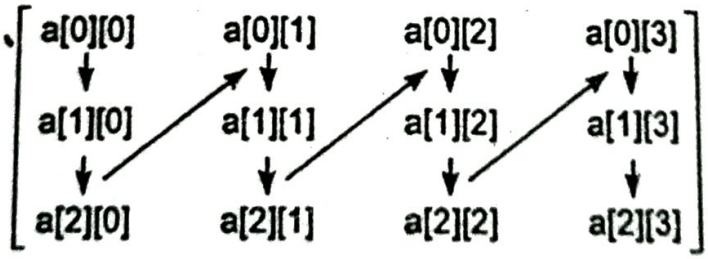


Fig. 2.6 : Column major representation

(b) Row Major Order : In this order, elements, are stored row-by-row that is first store the first row, then the second row, the third row and so on.

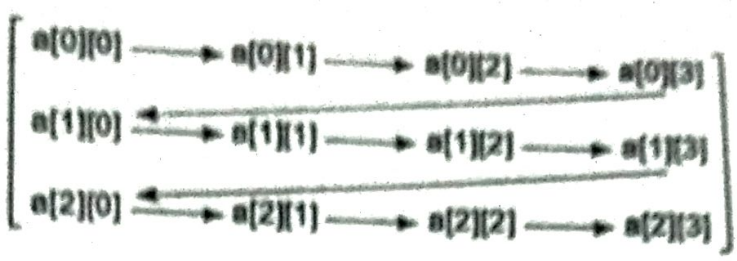


Fig. 2.7 : Row major representation

It is to be noted that in Pascal and C/C++, the storage order is row-major order whereas in FORTRAN, it is of column-major order.

The formula for calculating addresses are as follow :

In column-major order,

Loc[a(J, K)] = Base Address + S[M(K-1) + (J - 1)].

and in row-major order,

Loc[A(J, K)] = Base Address + S[N(J-1) + (K - 1)].

Where, s denotes the number of words per memory location.

M represents number of columns and N represents the number of rows.

Example 2.3 :

Consider an array of a[4][6]. Suppose, base = 200 and there are 2 words per memory cell. Find the address of a [2][4].

Solution :

By Using Column Major Order

Here, M = 6, J = 2, K = 4, S = 2, Base = 200, N = 4

$$\begin{aligned} Loc[A(2, 4)] &= 200 + 2[6(4 - 1) + (2 - 1)] \\ &= 200 + 2[6.3 + 1] \\ &= 200 + 2[18 + 1] \\ &= 200 + 2[19] = 200 + 38 \\ &= 238 \end{aligned}$$

By Using Row Major Order

$$\begin{aligned} Loc[A(2, 4)] &= 200 + 2[4(2 - 1) + (4 - 1)] \\ &= 200 + 2\ [4][1 + 3] \\ &= 200 + 2[7] = 200 + 14 = 214 \end{aligned}$$

200+14 =

2.5 OPERATIONS OF ARRAYS

We can perform various operations on arrays

(i) **Traversing :** Traversing means accessing each element exactly once. The process of traversing is very simple and straightforward. Let us suppose A be the linear array and we want to print the contents of each element of A.

Algorithm :

1. For i = LB to UB
   Apply PROCESS to a[i]
2. End

Insertion :

(ii) It means to add a new element to existing list. After insertion, the size of linear array is increased by one. This operation can be possible if the memory space allocated for the array is large enough.

Inserting an element at the end of the list can be done very easily. But at the middle of list, the elements are to be moved downward to new locations to accommodate the new element and keep the order of other elements.

Algorithm : Let us suppose, we want to insert new element at jth position.

(i) Set, i = n

(ii) Repeat step (a) & (b) until i > = J.

(a) a[i + 1] = a[i]

(b) i = i - 1

(iii) Set, a[J] = NEW item.

(iv) n = n + 1

(v) Exit

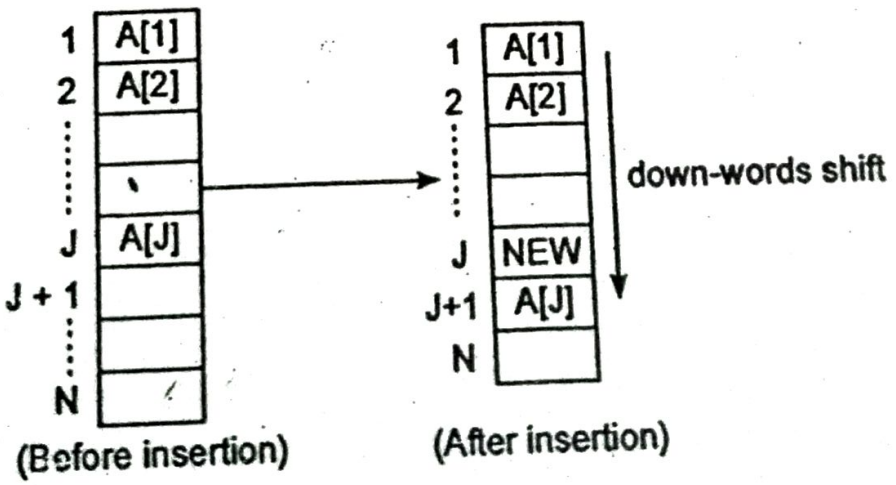


Fig. 2.8 ; Insertion process

Deletion :

(iii) It refers to the operation of removing an element from existing list. After deletion operation, the size of array is decreased by one. The elements are to be moved upward in order to fill up the location vacated by removed element.

Algorithm : Let us suppose we want to delete an element from position K.

1. Set, J = K
2. Repeat step (a) and (b) until J < N.
   (a) a[j] = a[j + 1]
   (b) j++
3. Set, n = n - 1
4. Exit

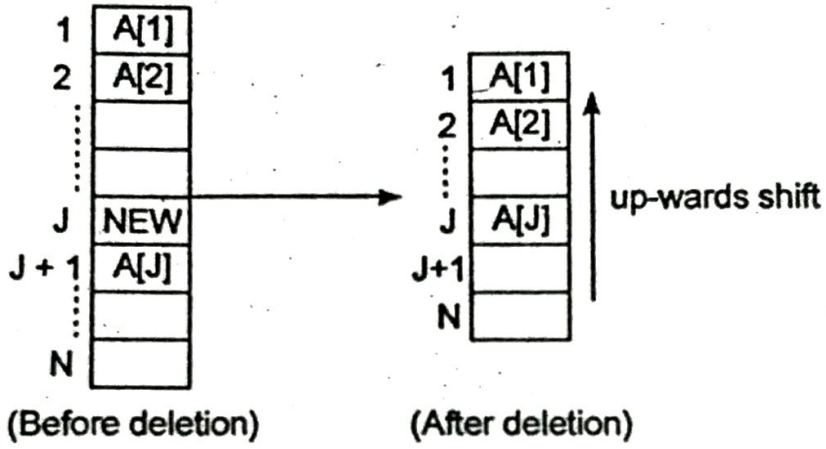


Fig. 2.9 : Deletion process

Searching and Sorting :

(iv) For details, please refer to chapter 3

2.6 SPARSE ARRAYS

In numerical computing, we come across matrices with maximum zeros. Such types of matrices are called sparse matrices. The matrix which is not sparse is called dense matrix. Sparse arrays are special arrays in which most of the elements are 0. For example, the following fig. shows a 6 × 5 sparse matrix with 6 nonzero elements.

$$\begin{bmatrix} 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 6 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \end{bmatrix}$$

Fig. 2.10 : Sparse matrix

There are following ways to represent sparse arrays :

(a) Array Representation

(b) Linked List Representation

Array Representation : Here, the elements are stored in the form of triplets.

<row, column, element>

The first field row is used to specify the row position, second field column is used to mention the column position and third field element is used to specify the nonzero element. The following fig. shows the array representation of above mentioned example.

Row	Column	Value
1	4	4
2	5	6
3	2	1
4	5	9
5	3	2
6	4	7

Fig. 2.11 : Representation of sparse Arrays

Linked list Representation : Here is an example of linked representation which is as follow :

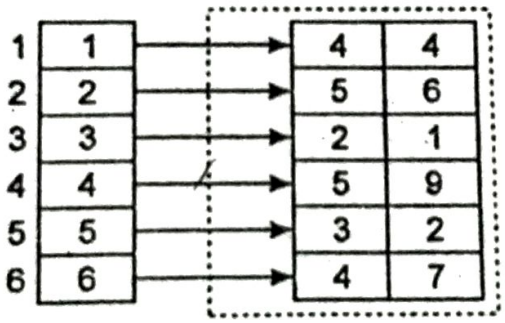


Fig. 2.12 : Representation of sparse Arrays through linked list

2.7 BENEFITS AND LIMITATIONS

Benefits

(a) Arrays are very easy and simple to create.

(b) Searching is easy.

(c) Arrays are linear, compile time data structures

Limitations

(a) Arrays are of fixed size. If we are not sure of size, it can be a potential waste of memory.

(b) Insertion and deletion operations are very time consuming due to shifting of elements.
(c) Arrays can store similar data type.

Salient Features

- An array is a finite ordered set of homogeneous elements. The elements of an array are referred to by their positions.
- One dimensional arrays are also called linear arrays or vectors. It is necessary to find base address of the array if we are calculating the address of a particular element.
- Two dimensional arrays are stored in the memory in row-major order and column-major order.
- Special type of arrays like sparse arrays are also described in this chapter which are used to represent a maximum no. of zero's.
- Insertion and deletion operation performs shifting that is downwards and upwards transfer of elements.

REVIEW QUESTIONS

A : Multiple choice questions

1. Which of the following sorting algorithm is of divide-and-conquer type?
   (a) Bubble sort (b) Insertion sort
   (c) Quick sort (d) All of the above
2. The number of interchanges required to sort 5, 1, 6, 2, 4 in ascending order using bubble sort is
   (a) 6 (b) 5
   (c) 7 (d) 4
3. The smallest element of an array's index is called its
   (a) Lower Bound (b) Upper bound
   (c) Range (d) Extraction
4. If the address of A[1][1] and A[2][1] are 1000 and 1010 respectively and each element occupies 2 bytes, then the array has been stored in _____ order.
   (a) Row Major (b) Column Major
   (c) Matrix Major (d) None of these
5. A two dimensional array A[6][8] is stored in Row major order with Base address 551. What is the address of A[3][4]?
   (a) 404 (b) 405
   (c) 406 (d) 407

SEARCHING

Searching is a process of searching the records in a data structure.
we can search the elements either by
- Linear Search.
- Binary Search..

LINEAR SEARCH

- It is done when the elements are not sorted in an array or list.
- starts at the beginning of an array and moves sequentially and checks the element one by one to see if it matches the element or not

Algorithm :- LS (a, N, item, Loc)

LS is the name of the algorithm as Linear Search.
a is the array with N elements Item is the element which we want to search in an array.
and Loc is the location of an element in an array

Step-1 Set i = 1

Step-2 ~~if i<=N then exit~~ if a[i] = Item,
then print Element found.
Set loc = i and Exit.
else
Go to step-3

Step-3 Increment i

$i = i + 1$

Go to step-2.

Complexity

worst Case = $O(n)$

Average Case = $O\left(\frac{n+1}{2}\right)$

Best Case = $O(1)$.

BINARY SEARCH

- Binary search works when an array is sorted.
- It is a fast search algorithm which works on the principle of Divide and Conquer.

Algorithm :- BS (a, N, item, Beg, end, mid)

BS is the name of algorithm as Binary Search a is an array with N elements and item is the element to search.

Step-1 :- Set Beg = 0 , end = N-1

and $mid = \frac{Beg + End}{2}$

Step-2 :- If a[mid] = Item,

then print that Item found and Exit

• The ways arrays are

ep-3 Repeat steps while (End >= Beg)

(a) If (Item < a[mid])

then end = mid - 1

and mid = $\frac{Beg + End}{2}$.

Go to step - 2

else

beg = mid + 1.

and mid = $\frac{beg + End}{2}$

Go to step - 2.

Complexity :-

Best Case = O(1)

Average Case = O(log n).

Worst Case = O(n)

STACK:

A stack is a linear non-primitive data structure in which items may be added (PUSH) or deleted (POP) only at one end. And this end is called top of stack (TOP).

- This means that the last item to be inserted to a stack is the first element to be deleted. So, stacks are called LIFO ie; Last in - first - out structure.

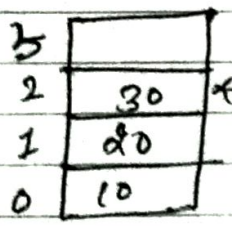
Operations on Stack

There are two basic operation associated with stack

1. PUSH : to insert an element into stack
2. POP : to delete an element from stack

Example of stack :

1. stack of dishes
2. stack of books
3. stack of chairs.



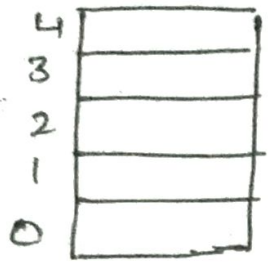
stack

Here we have a stack of size = 4, only 3 elements are there Initially And Top most element is at position 2. So TOP will point at location 2 (as it is the current TOP most position.

Take another example:

- Suppose, we want to insert 5 elements in stack ie; 10, 7, 5, 20, 15. & initially we have no element in stack.

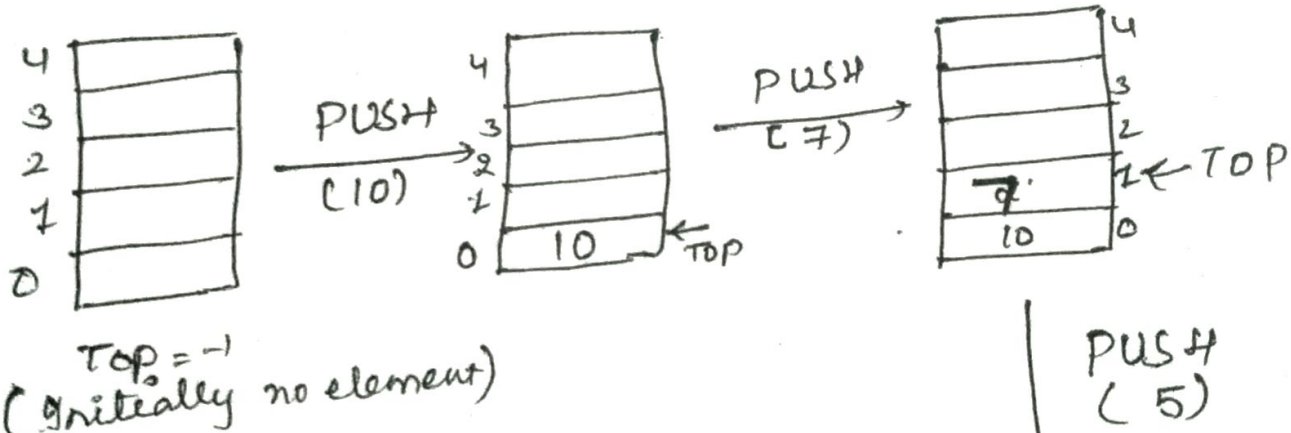
So in that case, stack is

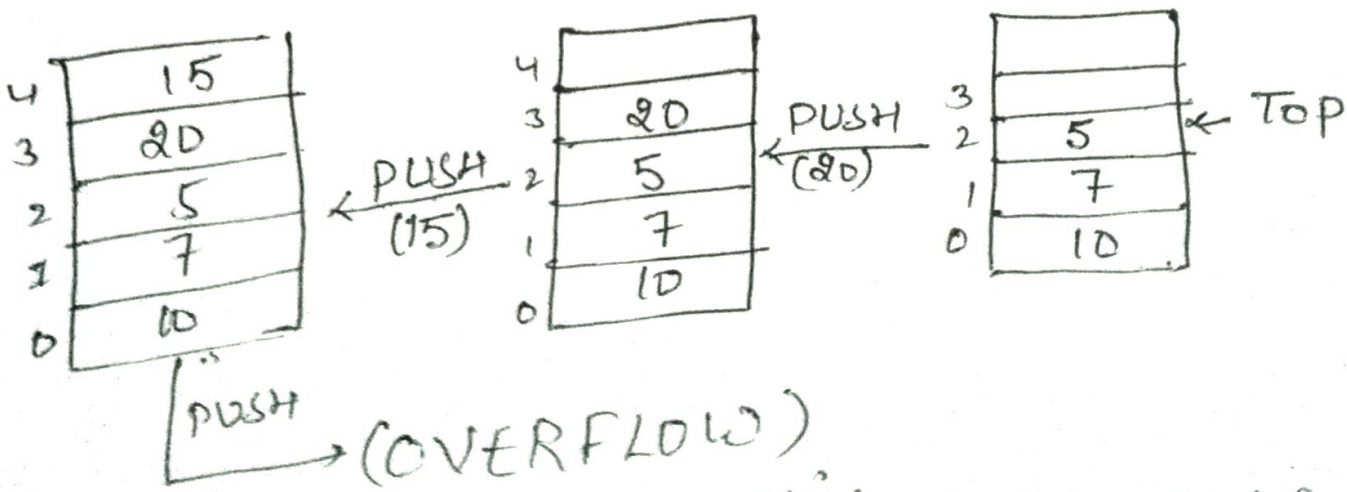


Top = -1

← Here size of stack = 5.
As there is no element in stack, so TOP will not point to any position of stack so we can say. Top = -1

Now we will PUSH elements one by one, as.





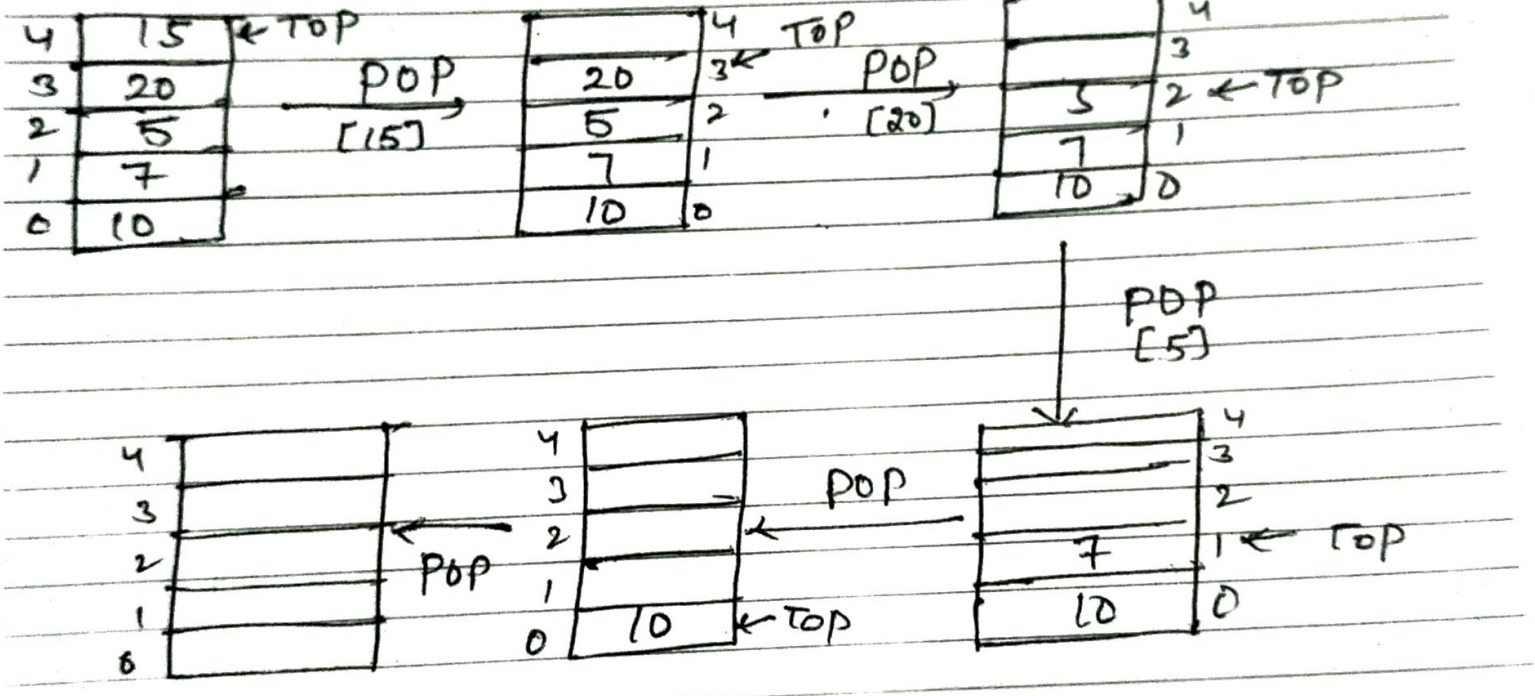
As stack is full, still we are pushing elements in it. Then this condition is

called "overflow".

- condition of overflow arise when Top reaches to maximum size of stack.
or we can say.

If (TOP == (size-1)) then
"OVERFLOW".

Now, we will delete elements from it, or perform <u>POP</u>.



TOP = -1

POP → "UNDER FLOW"

when stack becomes empty and still you are performing POP (there is no element to delete) then it is called "underflow".

So condition for underflow is

If (TOP == -1) then
"UNDERFLOW".

There are 3 conditions for Top

1. Top = -1 ⟶ underflow / stack empty
2. Top = size-1 ⟶ overflow / stack full.
3. Top = 0 to (size-1) ⟶ normal operation.

Stack implementation

Stack can be implemented in two ways :-

1) static implementation
2) Dynamic implementation

Static implementation: It uses arrays to create a stack. It is very simple technique but is not a flexible.
Here size of stack is has to be mentioned.

Dynamic implementation: It uses link-list representation & uses pointers to implement stack type of data structure.
Here size of stack can be increased at run time.

Algorithm For PUSH

go_PUSH (stack, TOP, size, item)

Here stack is the name of array, Top is the the pointer, pointing to topmost element. Item is the value to be PUSH.
Initial value of Top = -1

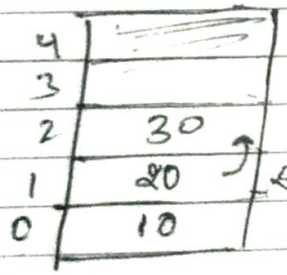
Step1 : [check for overflow]
If (Top == (size-1) then
Print (overflow) else go to next step.

Step2 : Set Top = Top + 1

Step3 : set stack [TOP] = item

step4 : exit.

Eg: consider size of stack is 5.

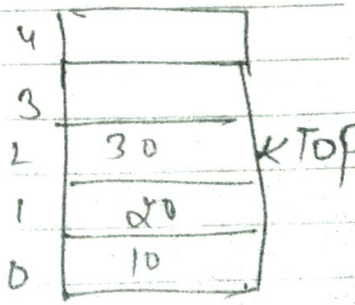


Top = 1 , item = 30

Step1 if (TOP == (5-1)) → false

Step2: TOP = Top + 1
= 1 + 1 = 2

step3 : stack [TOP] = item
stack [2] = 30



step4 exit.

Algorithm for Pop operation

POP (stack[size], item, top)

Here stack is name of array of maximum size of size. Item is the value deleted. and TOP is the stack pointer.

step1: [check for underflow condition]

if (TOP = = -1) then
Print ("Underflow") else go to next step.

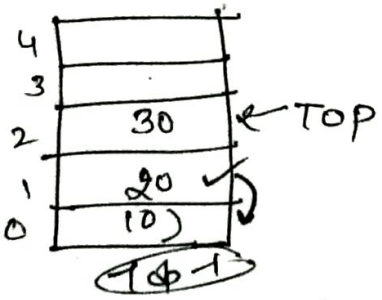
step2

item = stack [TOP]

step3 TOP = TOP - 1 [decrement TOP]

step4 EXIT.

Example: Suppose we have a stack of size 5 as



Here TOP = 2

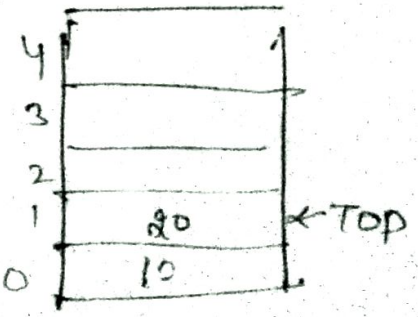
So

step1 if (TOP = = -1) ← false

step2 item = STACK [TOP]
item = STACK [2] // value at index 2
item = 30

step3 TOP = TOP - 1
TOP = 2 - 1 = 1

step4 exit



[As top is decremented, it comes one place down]

Applications of Stack

(1) Stack Frame: Stacks are used to pass parameters between functions. Programs compiled from high-level language (like C) make use of stack frame for the working memory of each procedure or function invocation. When any procedure or function is called - ~~a no. of words~~ - the stack frame - is pushed on to a program stack. When the procedure or function returns, this frame of data is poped off the stack.

(3) As a function calls another function, first its arguments, then the return address and finally space for local variable is pushed onto the stack. Since each function runs in its own "environment" or context, it becomes possible for a function to call itself — a technique known as recursion.

(2) The stack frame generally includes the following components -
1. The return address
2. argument variable of the f'ns
3. local variables of the f'n.

(4) Example:

```
increment ( int a)
{
  int c;  // local variable of fn increment.
    c = a++;
  return decrement (c);
  decrement (int d)
    {
      int e; // local variable of function decrement.
      d = e--;
    return increment (d);
```

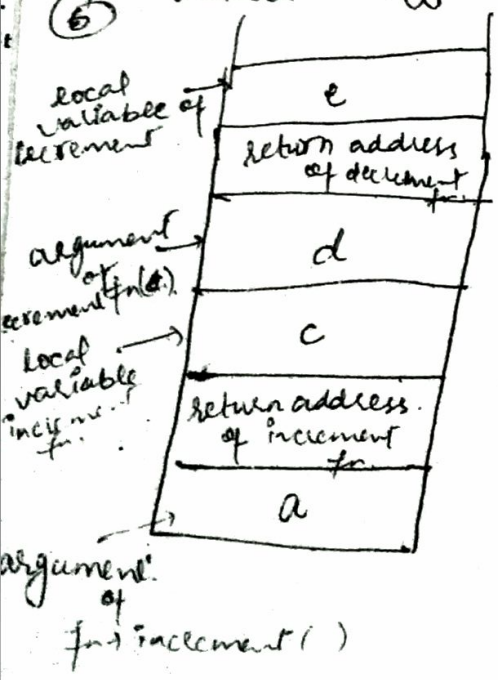
(5) So in stack frame, firstly increment fn's data will be pushed in order as: first its arguments then ~~return address~~, return address, then its local variables & then will be pushed onto stack. Since increment fn is calling decrement fn. so in same ~~variable~~ order, decrement fn's argument list, its return address and local variables will get pushed

Do j=1 to i
Do i=1 to i

Chapter

onto stack. As it is very much clear that for performing the task of fn increment, firstly decrement fn's task should be completed only then increment fn will return an output. So basically when a fn calls another fn calling another fn. , - the inside fn. will get evaluated first & reverts back its result to the calling fn which eventually gives back the result to main callee fn.

(6)



"A stack frame" is a frame of data that gets pushed onto the stack.

When a stack is called, a stack frame would represent a function call and its argument data.

The function return addresses, its arguments and space for local variables is pushed onto stack.

Reversing a string: As the characteristic of stack is reversing the order of execution. Stack helps us to generate reverse of a string.

ie a string 'PRIYA' can be converted to 'AYIRP' with the help of stack.

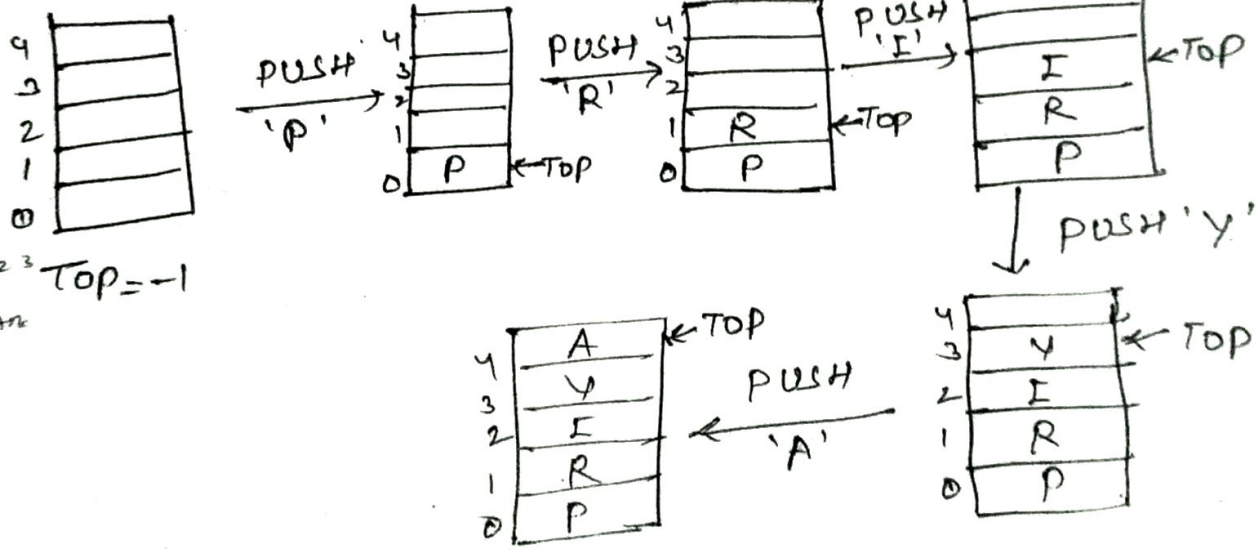
For this, we will perform two step:

Step1 First PUSH all the characters of string onto stack one by one.

Step2: start POP operation on stack.

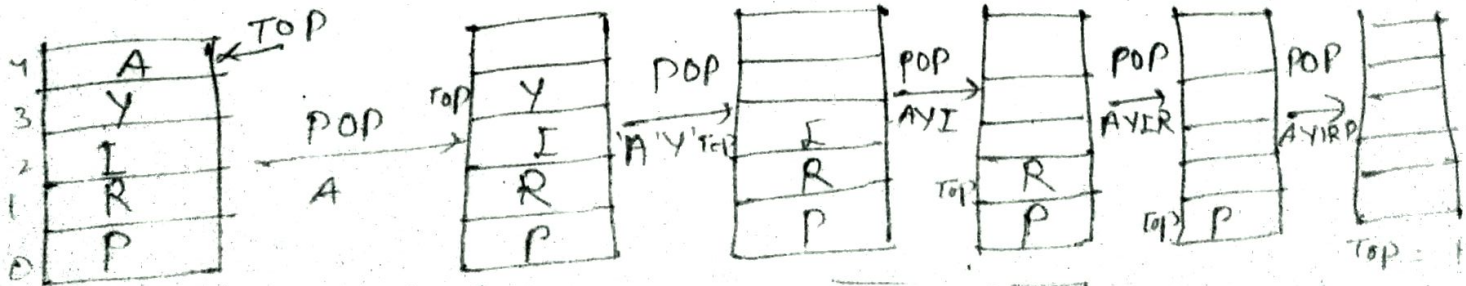
Eg: string (input) = PRIYA.
'P', 'R', 'I', 'Y', 'A'

Step1 PUSH



(All elements are pushed)

Step POP - one by one



String comes: AYIRP - Reversed

5) Evaluation of Postfix Expression ; POLISH NOTATIONS

There are 3 notations for an expression.

- an expression is defined as a no. of operands or data items combined using several operators

→ Infix expression ⟶ (A+B)
→ Prefix expression ⟶ +AB
→ Postfix expression ⟶ AB+

NOTE (Later explained in detail)

Algorithm :

Eval_Postfix (P, value)

Here P is the arithmetic expression in postfix notation & value is the calculated result of P.

Step1 : add a right parenthesis ')' at the end of P.

step2. scan P from left to right & repeat steps 3 & 4 for each element of P until the right parenthesis ")" is encountered.

Step3 : if an operand is encountered, put it on stack.

step4 : if an operator ⊗ is encountered, then

a. Remove the two top elements of stack where A is the top element & B is the next to top element.

b. evaluate B ⊗ A

c Place result of step 4(b) on top of stack

step5 : set value equal to top of stack.

step6 : exit.

13 + 4 * 3 +

Q: Consider the postfix expression

P = (1 2 + 4 * 3 +)

Step 1. add) at end of P. So

P = 1,2+4 * 3 +)

Step 2 Now scan it from left to right until ')' is encountered.

Symbol scanned	Stack	
1	1	← Operand, PUSH
2	1, 2	← operand, PUSH
+	3	← operator, POP
4	3, 4	
*	12	
3	12, 3	
+	15	
)	—	

value = 15

Step3 PUSH 1.

Step3 PUSH 2

Step4 '+' encountered, POP top most two element. As

A = 2

B = 1

evaluate B + A

1 + 2 = 3

Push it on stack

(Repeat same)

Ex.2.

P :- 5, 6, 2, +, * , 12, 4, /, -

Symbol Scanned	Stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37
)	

Ans - 37

AB+CD/

Expression Notations: Expression Notation is the representation of mathematical expressions. An expression is defined as a number of operands combined with several operators.

Eg $A+B$

Here A & B are operand & '+' is operator.

There are basically three types of expression notation

1. Infix notation.
2. prefix notation
3. Postfix notation

1) **Infix Notation:** Infix notation is what we come across in general maths, where the operator is written in-between the operands.

Eg 1. $A+B$

Here '+' is in between A & B.

2. $A+B*C$

$A+B*C$

Post $AB+C*$

2. **Prefix Notation:** the Notation in which the operator is written before the operands.

The same example of infix, can be written as in prefix as

Eg 1. $+AB$ (Here operator '+' is before A & B)

2 $+A*BC$

This notation is also called as **Polish Notation**

3. Postfix Notation : In this notation operator are written after the operands, so it is called postfix.

The same example can be written as:

Eg ① AB+

Here '+' operator comes after operand A & B.

2. ABC*+

This Notation is also called as Reverse Polish Notation.

Notation conversion / Expression conversion :

To calculate the expression for value, we must know the BODMAS Rule to know the precedence / priority of one operator over other.

Eg 4+3*7

It can solved as

4+3*7
= 7*7
= 49
(wrong)

Or

4+3*7
= 4+21
= 25
(correct)

First method is wrong, as it is not following the precedence rule. while second is correct.

Precedence order / priority order of operators :

Exponential operator	↑	Highest precedence
Multiply / Divide	*, /	Next precedence
Addition / subtraction	+, −	Least precedence.

Infix to postfix conversion

In this we will convert the infix expression into its equivalent postfix expression

Eg $(A+B) \xrightarrow[\text{conversion}]{\text{After}} AB+$
(Infix) (postfix)

Algorithm:

Postfix (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algo finds the equivalent postfix expression P.

Step1: Push left parenthesis "(" on to stack & add right parenthesis ")" at end of Q

Step2: scan Q from left to right and repeat steps 3 to 6 for each element of Q until the stack is empty.

Step3: if an operand is encountered, add it to P.

step4: if a left parenthesis '(' is encountered, push it on top of stack

step5: if an operator ⊗ is encountered, then:

a. Repeatedly POP from STACK and add each operator to P which has the same or higher precedence than ⊗.

b. add ⊗ to stack.

Step6: if a right parenthesis ')' is encounterd, then:

a. Repeatedly Pop from STACK and add to P each operator to P until a left parenthesis is encounterd on stack.

b. Remove the left Parenthesis from stack [Do not add it to P]

Step7 EXIT.

Example.

Q = A * B + C / D

P = ?

Soln:

Step1 add '(' on stack & ')' at end of Q.

Q = A * B + C / D)

Step2 scan Q from left to right until stack is empty

Step3. operand is encounter, ie. A, so add it to P.

Now operator (*) encounter

Step 4 : will skip.

Step 5 : operator encounter. Push it on stack.

No Pop., as there are no elements on stack

SYMBOL SCAN	STACK	P(OUTPUT)
—	C	—
A	C	A
*	C*	A
B	C*	AB
+	C*+	AB*
C	C+	AB*C
/	C+/	AB*C
D	C+/	AB*CD
)	(+/	AB*CD/+

stack is empty

B encounter

Step 6 will skip. Loop repeat. and again an operand is encounter

Step3 Add B to P

'+' encounter

step4 : will execute as operator encountered.

[Now '*' is already on top of stack which has higher precedence than '+'. so 4(a) will execute. ie Pop from stack & add to P. * will poped out & added to P]

encounter
step 5 & 6 will skip. loop repeat.
step 3 will execute as operand is encountered

step 3: add 'C' to P.

'/' encounter:

step 4 will execute as operator '/' is encountered.
Now we already have '+' on top stack & operator encountered is '/' which has hegher precedence than '+'. So no pop from stack.

D encounter:

step 3: will execute, as operand 'D' is encountered so add it to P.

')' encounter:

step 6 will execute as right parethesis is encountred
now repeatedy pop from stack & add to P until a '(' encounter. so '/' & '+' will poped out and added to P.

remore left parenthesis from stack.

Now stack becomes empty. loop terminate. and A is scanned completely so P is required output.

So

$\underbrace{A+B*C}_{\text{Infix}}$ $\xrightarrow[\text{conversion}]{\text{After}}$ $\underbrace{AB*CD/+}_{\text{Postfix}}$

Q:- $A+(B*C-(D/E\uparrow F)*G)*H$

Ans:- $ABC*DEF\uparrow/G*-H*+$

Take another example.

Q = ((A+B^D)/(E-F)+G)

P = ?

Apply step 1.

Q = ((A+B^D)/(E-F)+G))

SYMBOL SCANNED	STACK	P (OUTPUT)	
	(		
(	((		
(	(((		
A	(((	A	
+	((+	A	
B	((+	AB	
^	((+^	AB	→ ^ ← big than it
D	((+^	ABD	
)	((+	ABD^	← POP
/	((/	ABD^+	
(	((/(	ABD^+	
E	((/(	ABD^+E	
-	((/(-	ABD^+E	
F	((/(-	ABD^+EF	
)	((/	ABD^+EF-	← POP
+	((+	ABD^+EF-/	+ ← Small than /
)	(	ABD^+EF-/+	← POP
)		ABD^+EF-/+	← POP

STACK empty, Q is scanned completely. P is required output. ie

ABD^+EF-/+

+/-FE+^DBA

Infix to Prefix Conversion:

In this we will convert the infix expression into prefix expression.

i.e.

(A+B) — After conversion → (+AB)

Infix — Prefix

Algorithm:

prefix (Q, P)

// Q is Infix expression
// P is equivalent Prefix expression.

Step 1: Reverse the input string Q to Q'.

Step 2 Push left parenthesis "(" on to stack & add right parenthesis ")" at end of Q'

Step 3 Scan Q' from left to right and Repeat steps 4 to 7 for each element of Q' until the stack is empty

step 4: if an operand is encountered, add it to P' (Postfix expression)

step 5: if a left parenthesis '(' is encountered, add it to P'.

step 6: if an operator ⊗ is encountered, then:

a. Repeatedly Pop from STACK and add to P' each operator which has same or higher precedence than ⊗.

b. add ⊗ to stack.

step 7: if a right parenthesis ')' is encountered then:

a. Repeatedly Pop from STACK & add to P' each operator until a left parenthesis is encountered on stack.

b. Remove the left parenthesis from stack.

step 8: Now reverse P' to get prefix exp' P

step 9: EXIT.

Eg let Q = A*B+C/D Q' = D/C+B*A)

P = ?

step1 reverse Q to Q'

Q' = -D/C+B*A)

step2 add ')' as end of Q' & '(' on top of stack.

step3 repeat steps 4 to 7 until stack is empty & scan Q'

NOTE: (Apply steps as in previous example of postfix convesion)

SYMBOL	STACK	P' (Postfix) Pre	o/p
-	(	- ↓	
D	(	D	
/	(/	D	
C	(/	DC	
+	(+	DC/	← / Pop out as />+
B	(+	DC/B	
*	(+*	DC/B	← No Pop as *>+
A	(+*	DC/BA	
)	—	DC/BA*+	Pop

So P' is DC/BA*+

which is postfix. +*AB/CD

Apply step 8: Reverse P' to P

i.e P = |+*AB/CD|

Prefix