GATE CSE NOTES

by

UseMyNotes

Introduction.

* **Data** (Datum - fact) - In computers, data is the value assigned to a variable.

* **Data Structure** - Data structure is a data organization & storage format that enables efficient access & modification.

Data structures can implement one or more particular Abstract data types. DS is a concrete implementation of the space provided by an ADT.

* **Abstract Data Type**: An ADT is a mathematical model for data types, where a data is defined by its behaviour from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type & the behaviour of these operations.

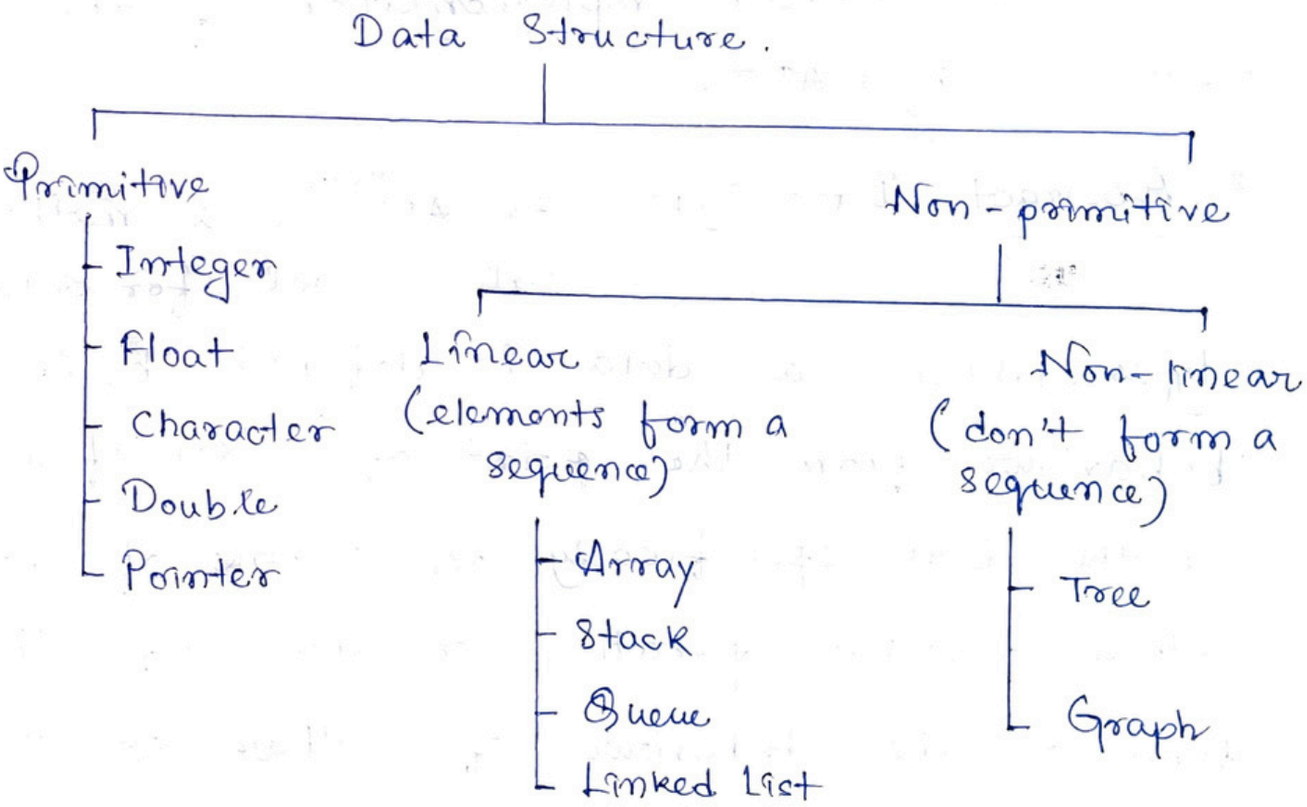
This contrasts with DS, which are concrete representations of data & are the point of view of an implementer, not a user.

Formal def": (Class of objects whose logical behavior is defined by a set of values & a set of operations.)

• Data abstraction is the separation between the specification of a data object from the outside world & its implementation.

Eg. For the set ADT, operations possible are union, intersection, size, complement & also the possible values are $\{\} = \phi$, $\{0,1\}$, $\{-2, 3, 39\}$ etc.

* Classification of Data Structure :



* Operations on Data Structure :

1. Inserting
2. Searching
3. Traversing
4. Sorting
5. Deleting.
6) Selection
7. Merging
8. Splitting.

* Abstraction & implementation :

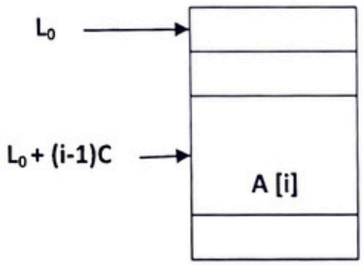
ADT	vs.	DS
1. Vehicle		Golf cart
		Bicycle
		Smart car
2. Map		Tree Map
		Hash Map
		Hash table
3. List		Dynamic array
		Linked list
4. Queue		Linked list based queue
		Array based queue
		Stack based queue

Array

Explain Array in detail

One Dimensional Array

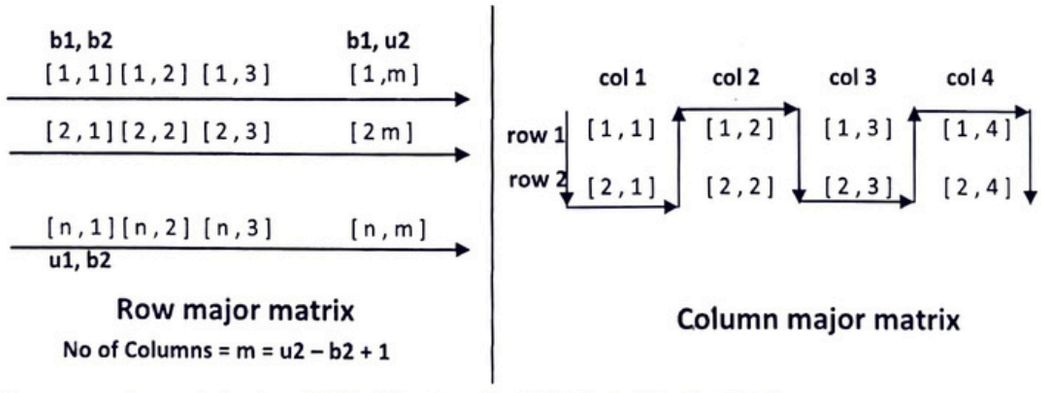
- Simplest data structure that makes use of computed address to locate its elements is the one-dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.
- Vector A with subscript lower bound of "one" is represented as below....



- L_0 is the address of the first word allocated to the first element of vector A.
- C words are allocated for each element or node
- The address of A_i is given equation **Loc (A_i) = L_0 + C (i-1)**
- Let's consider the more general case of representing a vector A whose lower bound for it's subscript is given by some variable b. The location of Ai is then given by **Loc (A_i) = L_0 + C (i-b)**

Two Dimensional Array

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts
- Two dimensional array in which elements are stored column by column is called as column major matrix
- Two dimensional array in which elements are stored row by row is called as row major matrix
- First subscript denotes number of rows and second subscript denotes the number of columns
- Two dimensional array consisting of two rows and four columns as above Fig is stored sequentially by columns : A [1, 1], A [2 , 1], A [1 , 2], A [2 , 2], A [1 , 3], A [2 , 3], A [1, 4], A [2, 4]
- The address of element A [i , j] can be obtained by expression **Loc (A [i , j]) = L_0 + (j-1)*2 + i-1**
- In general for two dimensional array consisting of n rows and m columns the address element A [i , j] is given by **Loc (A [i , j]) = L_0 + (j-1)*n + (i – 1)**
- In row major matrix, array can be generalized to arbitrary lower and upper bound in its subscripts, assume that b1 ≤ I ≤ u1 and b2 ≤ j ≤u2



- For row major matrix : **Loc (A [i , j]) = L_0 + (i – b1) *(u2-b2+1) + (j-b2)**

Applications of Array

1. Symbol Manipulation (matrix representation of polynomial equation)
2. Sparse Matrix

Symbol Manipulation using Array

- We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc...
- We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc... can be performed in efficient manner
- Array can be used to represent Polynomial equation
- **Matrix Representation of Polynomial equation**

	Y	Y^2	Y^3	Y^4
X	$X\,Y$	$X\,Y^2$	$X\,Y^3$	$X\,Y^4$
X^2	$X^2\,Y$	$X^2\,Y^2$	$X^2\,Y^3$	$X^2\,Y^4$
X^3	$X^3\,Y$	$X^3\,Y^2$	$X^3\,Y^3$	$X^3\,Y^4$
X^4	$X^4\,Y$	$X^4\,Y^2$	$X^4\,Y^3$	$X^4\,Y^4$

e.g. $2x^2+5xy+Y^2$
is represented in matrix form as below

	Y	Y^2	Y^3	Y^4
X	0	0	**1**	0	0
X^2	0	**5**	0	0	0
X^3	**2**	0	0	0	0
X^4	0	0	0	0	0
	0	0	0	0	0

e.g. $x^2+3xy+Y^2+Y-X$
is represented in matrix form as below

	Y	Y^2	Y^3	Y^4
X	0	0	**1**	0	0
X^2	**-1**	**3**	0	0	0
X^3	**1**	0	0	0	0
X^4	0	0	0	0	0
	0	0	0	0	0

- Once we have algorithm for converting the polynomial equation to an array representation and another algorithm for converting array to polynomial equation, then different operations in array (matrix) will be corresponding operations of polynomial equation

---

What is sparse matrix? Explain

- An mXn matrix is said to be sparse if "many" of its elements are zero.
- A matrix that is not sparse is called a dense matrix.
- We can device a simple representation scheme whose space requirement equals the size of the non-zero elements.

- **Example:-**
  - The non-zero entries of a sparse matrix may be mapped into a linear list in row-major order.
  - For example the non-zero entries of 4X8 matrix of below fig.(a) in row major order are 2, 1, 6, 7, 3, 9, 8, 4, 5

0	0	0	2	0	0	1	0
0	6	0	0	7	0	0	3
0	0	0	9	0	8	0	0
0	4	5	0	0	0	0	0

Fig (a) 4 x 8 matrix

Terms	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Row	1	1	2	2	2	3	3	4	4
Column	4	7	2	5	8	4	6	2	3
Value	2	1	6	7	3	9	8	4	5

Fig (b) Linear Representation of above matrix

- To construct matrix structure we need to record
  (a) Original row and columns of each non zero entries
  (b) No of rows and columns in the matrix
- So each element of the array into which the sparse matrix is mapped need to have three fields: row, column and value
- A corresponding amount of time is saved creating the linear list representation over initialization of two dimension array.

A =

0	0	6	0	9	0	0
2	0	0	7	8	0	4
10	0	0	0	0	0	0
0	0	12	0	0	0	0
0	0	0	0	0	0	0
0	0	0	3	0	0	5

- Here from 6X7=42 elements, only 10 are non zero. A[1,3]=6, A[1,5]=9, A[2,1]=2, A[2,4]=7, A[2,5]=8, A[2,7]=4, A[3,1]=10, A[4,3]=12, A[6,4]=3, A[6,7]=5.
- One basic method for storing such a sparse matrix is to store non-zero elements in one dimensional array and to identify each array elements with row and column indices fig (c).

	ROW	COLUMN	A
1	1	3	6
2	1	5	9

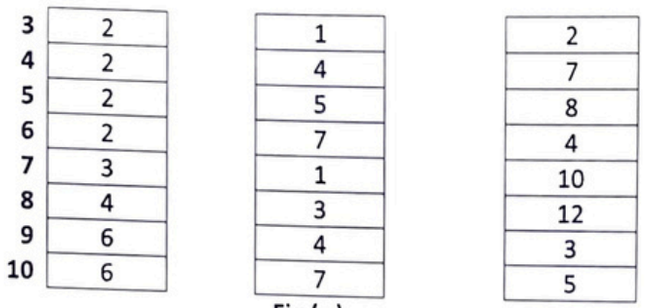
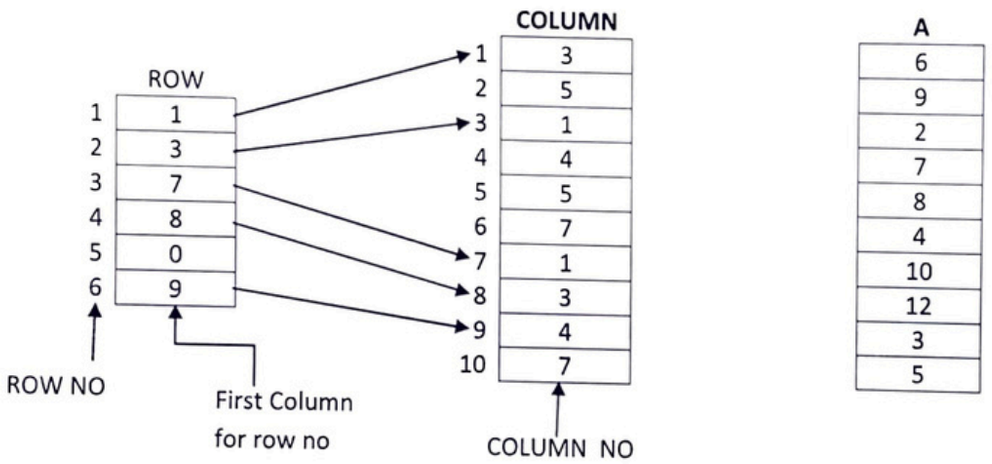


Fig (c)

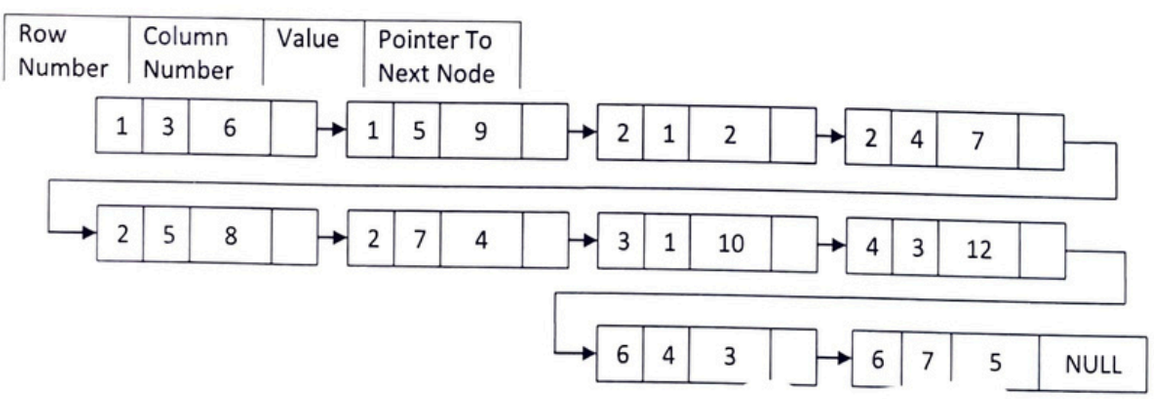


Fig(d)

- A more efficient representation in terms of storage requirement and access time to the row of the matrix is shown in fid (d). The row vector changed so that its i^{th} element is the index to the first of the column indices for the element in row I of the matrix.

Linked Representation of Sparse matrix

Typical node to represent non-zero element is



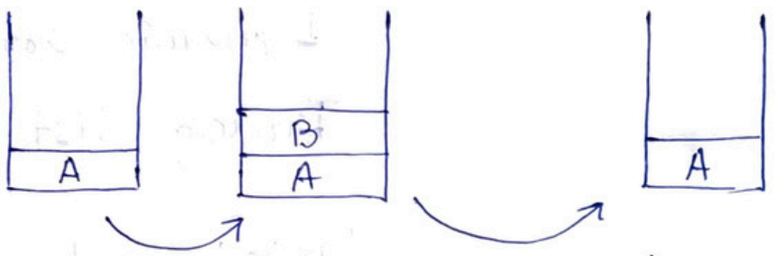
DS1 (Code, Th) DS2 (Adv.)

The Stack.

* **Stack** : A stack is a linear data structure where the elements are added & removed only from one end, that is called the TOP.

LIFO - Last in First out — The element that was inserted last is the first one to be taken out.

* We need stacks in function calls.

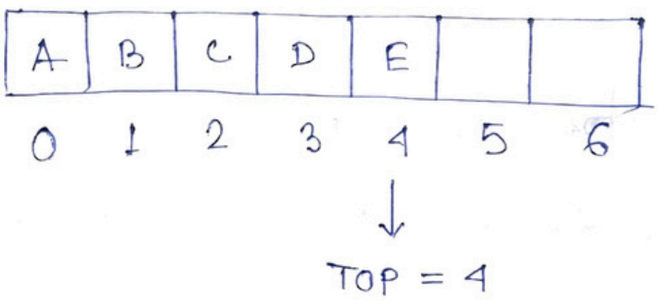


A calls B
B pushed

B executed
B popped.

* **Array Representation of Stack:**

- Every stack has a variable called **top** associated with it, that is used to store the address of the topmost element of the stack.
- **Max** - variable to store the max. no. of elements that the stack can hold.
- TOP = NULL — stack empty
  TOP = MAX - 1 — stack full



Array representation.

* Operations on Stack:

1. PUSH : Used to insert element into the stack. The new element is added to the topmost position of the stack.

Algo

Step 1: If- TOP = MAX-1
Print Overflow
Goto step 4
End if-

Step 2: Set Top = Top + 1

Step 3: Set stack [Top] = Value

Step 4: End.

2. POP : Used to delete element of the topmost position of stack.

Algo

1: If- TOP = NULL
Print Underflow
Goto 4
End if-

2: Set val = Stack [TOP]

3: SET TOP = TOP - 1

4: End.

3. PEEK : Used to return value of the topmost positioned element of stack without deleting it.

Algo

1: If TOP = NULL
   Print Stack empty
   Goto 3
2: Return Stack [TOP]
3: End.

* Program to perform PUSH, POP, PEEK.

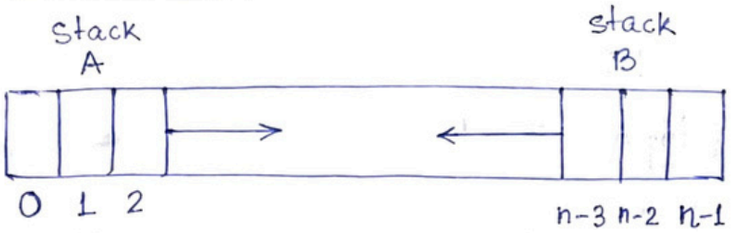
→

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 3
int stack [MAX], TOP = -1;
void PUSH (int stack [], int value)
{
    if (TOP == MAX-1)
    {
        printf ("\n Stack overflow! ");
    }
    else
    {
        TOP ++;
        stack [TOP] = value;
    }
}
```

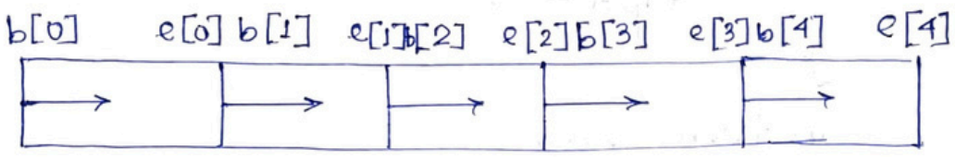
```
int POP (int stack[])
{
        int value;
        if (top == -1)
        {
            printf ("\n Stack underflow!");
            return -1;
        }
        else
        {
            value = stack [TOP];
            TOP --;
            return value;
        }
}
void display (int stack[])
{
    int i;
    if (TOP == -1)
        printf ("\n Stack empty!");
    else
    {
        for (i=TOP; i>=0; i--)
        {
            printf ("\n %d", stack[i]);
        }
    }
}
int peek (int stack[])
{   if (TOP == -1)
    { printf ("Stack empty!");
      return -1; }
    else {
        return (stack [TOP]); }
}
```

```
int main (
{
    int value, option;
    do
    {
        printf ("1. PUSH");
        printf ("2. POP");
        printf ("3. PEEK");
        printf ("4. DISPLAY");
        printf ("5. EXIT");
        printf ("Enter option - ");
        scanf ("%d", &option);
        switch (option)
        {
            case 1: printf ("Enter element: ");
                    scanf ("%d", &value);
                    PUSH (stack, value);
                    break;
            case 2: value = POP (stack);
                    if (value != -1)
                        printf ("%d deleted", value
                    break;
            case 3: value = peek (stack);
                    if (value != -1)
                        printf ("In %d at top", value)
                    break;
            case 4: display (stack);
                    break;
        }
    } while (option != 5)
    return 0;
}
```

* Multiple Stack :



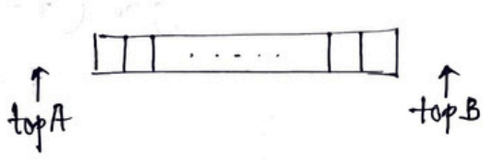
- Double Stack – n will be greater than the size of both the stacks. ~~with~~ Stack A & stack B grow from different directions.

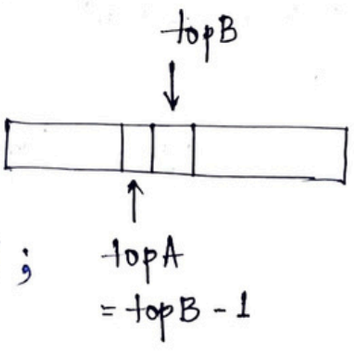


A stack can be used to represent n number of stacks in the same array. Each stack # i will be allocated an equal amount of space bounded by indices b[i] & e[i].

* Program to implement multiple stack.

→
```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int stack [MAX] , topA = -1, topB = MAX ;
void pushA (int val)
{
    if (topA == topB - 1)
        printf ("\n Overflow) ;
    else
    {
        topA += 1 ;
        stack [topA] = val ;
    }
}
```





```
int popA ()
{
    int val;
    if (topA == -1)
    {
        printf ("\n Underflow");
        val = -999;
    }
    else
    {
        val = stack [topA];
        topA --;
    }
    return val;
}
void displayA ()
{
    int i;
    if (topA == -1)
        printf ("\n Stack A empty");
    else
    {
        for (i = topA; i >= 0; i--)
            printf ("\t %d", stack [i]);
    }
}
void pushB (int val)
{
    if (topB - 1 == topA)  // topB == topA + 1
        printf ("\n Overflow");
    else
    {
        topB -= 1;
        stack [topB] = val;
    }
}
```

```
int popB()
{
    int val;
    if (topB == MAX)
    {
        printf("\n Underflow");
        val = -999;
    }
    else
    {
        val = stack[topB];
        topB++;
    }
}
void displayB()
{
    int i;
    if (topB == MAX)
        printf("\n stack B empty");
    else
    {
        for(i= topB; i<MAX; i++)
            printf("\t %d", stack[i]);
    }
}
void main()
{
    int option, val;
    clrscr;
    do {
        printf("PUSHA-1, PUSHB-2, POPA-3,
                POPB-4, DISA-5, DISB-6, Exit-7");
        printf("\n Enter choice - ");
        scanf("%d", &option
        switch operation ---} while(option!=7)
}
```

* Application of Stacks.

1. Reversing a list : [P]

Read each # from array & push into stack. Then pop one by one & store @ array.

```
int stk [10];
int top = -1;
int pop (); void push (int);
int main ()
{
    int val, n, i;
  int arr [10];
    printf ("\n Elements No:");
    scanf ("%d", &n);
    printf ("\n Elements: ");
    for (i = 0; i < n ; i++)
        scanf ("%d" &arr [i]);
    for (i = 0; i < n; i++)
        push (arr [i]);
    for (i = 0; i < n; i++)
    {
        val = pop ();
        arr [i] = val;
    }
    printf ("\n The reversed array: ");
    for (i = 0; i < n; i++).
        printf ("\n %d ", arr [i]);
    return 0;
}
void push (int val)
{
    stk [++top] = val;
}
int pop ()
{
    return (stk [top--]);
}
```

2. Implementing Parentheses Checker:

{A + (B + C)} (A + B}

VALID INVALID.

```
#define MAX 10
int top = -1;
int stk [MAX];
void push (char);
char pop ();
int main ()
{
```

Push all left paranthesis into stack. When a right paranthesis is encountered, try to match with an eqv. left paranthesis. If matched, then valid till that. At end, empty stack if valid.

```
    char exp [MAX], temp;
    int i, flag = 1; // flag to check if valid or not
    printf ("Enter expression: ");
    gets (exp);
    for (i=0; i<strlen (exp); i++)
    {
      if (exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
            push (exp [i]); // push (, {, [
      if (exp[i] == ')' || exp [i] == '}' || exp[i] == ']')
      {
          if (top == -1) // if empty stack, invalid
1. A) rejected ------>      flag = 0;
          else           // if non-empty stack
          {
              temp = pop(); // pop & try to match
              if (exp[i] = ')' && (temp == '{' ||
                                   temp == '['))
                  flag = 0;
              if (exp[i] = '}' && (temp = '(' ||
                                   temp == '['))
                  flag = 0;
              if (exp[i] == ']' && (temp == '(' ||
                                    temp == '{'))
                  flag = 0;
          }
      }
```

```
    }
    if (top >=0) //if non empty stack
        flag = 0;
    if (flag == 1)
        printf ("\n Valid");
    else
        printf ("\n Invalid");
    return 0;
}
void push (char c)
{
    if (top == (MAX -1))
        printf ("\n Overflow");
    else
    {
        top+= 1;
        stk [top] = c;
    }
}
Char pop
{
    if (top == -1)
        {printf ("\n Underflow");
        return (-999); }
    else
    {
        return stk [top--];
    }
}
```

2. (A) rejected

3. Evaluation of Arithmetic Expressions :

- **Infix** – operand operator operand.
- **Postfix** – (Developed by Jan Łukasiewicz, Polish logician) . Polish Notation → prefix.

  ↓

  Reverse Polish Notation. (RPN).

  operand operand operator. ⟶ left to right evaluation
- **Prefix** –

  operator operand operand ⟶

Eg. Infix (A+B)/(C+D) – (D*E).

Postfix [AB+] / [CD+] – [DE*]

[AB+ CD+ /] – [DE*]

AB+CD+/DE*–

Prefix **–/+AB+CD*DE**

Algo [Infix ⟶ Postfix] Operator stack

1. Add ')' to the end of infix.
2. Push '(' to the stack.
3. Repeat until each character in the infix is scanned.
   - 3a) If a '(', push it to stack.
   - 3b) If an operand add to postfix.
   - 3c) If a ')' is encountered,
     - i) Repeatedly pop from stack & add to postfix until a '(' is encountered.

ii) Discard the '(' from stack & do not push to postfix

3d) If an operator ø is encountered, then

i) Repeatedly pop from stack & add each operator to the postfix that has the same precedence than ø.

ii) Push ø to the stack.

4. Repeatedly pop from the stack & add it to the postfix expression until the stack is empty.

5. Exit.

Algo [Evaluation of Postfix]

1. Add a ')' at the end of postfix.
2. Scan every character of postfix & repeat (3) & (4) until ')' is encountered.
3. If an operand is encountered, push to stack.

If an operator O is encountered, then

a) Pop top two elements from the stack as A & B and evaluat B O A, where A is the topmost element & B below A.

b. Push the result on the stack.

4. Set result equal to the topmost element of the stack.
5. Exit.

* Expression evaluation.

Most of the compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

✓ Most programming languages are context-free languages allowing them to be parsed with stack based machines.

✓ Note: natural languages are context sensitive & stacks alone are not enough to interpret their meaning.

- Operator precedence in arithmetic exp^{n}s:

B Bracket
O Order or power
D Division
M Multiplication
A Addition
S Subtraction

- Evaluation of arith. exp^n in computers:
  1. Infix → Postfix conversion
  2. Evaluating postfix exp^n

Evaluation of infix:

1. Read one character.
2. Actions at end of each i/p:
   a) Opening brackets: Push into stack & goto (1)

(b) Number : Push into stack & goto (1).

(c) Operator : Push into stack & goto (1).

(d) Closing brackets : Pop from stack.
(popped char → c)

i. If c is closing bracket discard, goto (1).

ii. Pop four times otherwise.

First pop operand$_2$

Second pop operator

Third pop operand$_1$

Fourth popped elem as assigned the remaining

Opening bracket : Discarded.

Evaluate op_1 op op_2

Push result into stack & goto d.

(e) Newline : pop from stack & print answer.

eg. (((2 * 5) - (1 * 2)) / (11 - 9)) \n

Symbol	Stack
(	(
(	((
(	(((
2	(((2
*	(((2*
5	(((2*5
)	((10
-	((10-
(	((10-(
1	((10-(1
*	((10-(1*
2	((10-(1*2
)	((10-2
)	(8
/	(8/
(	(8/(
11	(8/(11
-	(8/(11-
9	(8/(11-9
)	(8/2
)	4
\n	pop ↓ 4

- Another algorithm : Using 2 stacks.

Approach : Use 2 stacks, operand & operator stack

Process :

1. Pop out 2 values from operand stack.
2. Pop out operator from operator stack.
3. Evaluate & push result to the operand stack.

Algorithm :

Iterate through expression, one character at a time.

1. If character is an operand, push it to the operand stack.
2. If character is operator,
   i) If operator stack is empty, push it to operator stack.
   ii) Else if operator stack is not empty,
      - if character's precedence is greater than or equal to the precedence of the stack top of operator stack, then push char. to operator stack.
      - otherwise pop 2 value from operand stack, pop out 1 operator from operator stack

& evaluate until character's precedence is less or stack is not empty. Then push result to operand sta

3. If character is '(', push into operat stack.

4. If character is ')', then do process until the corresponding '(' is encountered in operator stack. Now, pop the '('.

Once the expression is iterated fully, if stack is not empty, do process until operator stack is empty. Value left at operand stack is the final result.

TC O(n)

eg. 2 * (5 * (3 + 6)) / 15 - 2

Token	Op^nd	Op^r
2	2	
*	2	*
(	2	*(
5	25	*(
*	25	*(*
(	25	*(*(
3	253	*(*(
+	253	*(*(+
6	2536	*(*(+
)	259	*(*
)	2 45	*

Token	Op^nd	Op^r
/	2 45	*/
15	2 45 15	
-	2 3	*
	6	-
2	6 2	-

4

◐ Infix to postfix conversion.

Approach: Use an operator stack.

Algorithm:

Initialize result as a blank string. Iterate through given expression, one character at a time:

1. If the character is an operand, add it to the result.

2. If the character is an operator,

- If the operator stack is empty then push it to the stack.
- Else if the operator stack isn't empty.
  - If the operator's precedence is $\geq$ to that of the stack top of the stack, then push to the operator stack.
  - If the operator's precedence is $<$ that of the stack top then pop out an operator from the stack & add it to the result until the stack is empty or operator's precedence is $\geq$ to that of the stack top. At last, push the operator to stack.
    - If top = (, push the operator.

3. If character is '(', push

4. If ')', then pop an operator & add to result until corresponding '(' is encountered. Now, pop out the '('.

Once the expression iteration is completed & the operator stack is not empty, pop out an operator from the stack & add to the result until the operator stack is empty. Result is an

eg. A + B * (C ^ D - E)

Token	Stack	Result
A		A
+	+	
B	+	AB
*	+ *	AB
(	+ * (	AB
C	+ * (	ABC
^	+ * (^	ABC
D	+ * (^	ABCD
-	+ * (	ABCD^
	+ * (-	
E	+ * (-	ABCD^E
)	+ *	ABCD^E-

ABCD^E-*+

◐ Evaluation of postfix expression

Approach: Use operand stack.

Algorithm:

Iterate through given expression, one character at a time.

1. If the character is a digit, initialize number = 0.
   - While the next character is digit, do number = number * 10 + digit
   - Push number to stack.
2. If the character is an operator,
   - Pop operand from stack, say op1
   - Pop operand from stack, say op2
   - Perform op2 operator op1 & push.
3. Once the expression iteration is completed, Stack will have the final result.

eg. 20 50 3 6 + * * 300 / 2 -

Token	Stack
20	20
50	20 50
3	20 50 3
6	20 50 3 6
+	20 50 9
*	20 450
*	9000
300	9000 300
/	30
2	30 2
-	28

Evaluation of prefix expressions

Operand stack

Algorithm: Reverse the expression & iterate.

1. If character is an operand, push it to stack.
2. If character is an operator,
   i) Pop operand, op1
   ii) Pop operand, op2
   iii) Perform op1 operator op2 & push.
3. Once iteration is complete, stack has the result.

Infix to prefix conversion

Operator stack

Algorithm:

1. Reverse expression.
2. Infix to postfix.
3. Reverse to get prefix.

eg. Infix A + B * (C ^ D - E)

Reverse) E - D ^ C (* B + A

↓ Reverse brackets

(E - D ^ C) * B + A

Inf → Postf E D C ^ - B * A +

Reverse for prefix + A * B - ^ C D E.

● Prefix to postfix conversion — Operand Stack

Iterate in reverse.

1. If operand, push.
2. If operator,
   pop op1, pop op2, push (op1 op2 op)
3. Once iteration is complete, initiatize the result string & pop out from the stack & add it to the result.

eg. Prefix : * - A / B C - / A K L

Token	Stack
L	L
K	LK
A	LKA
/	L (AK/)
-	(AK/L-)
C	(AK/L-) C
B	(AK/L-) CB
/	(AK/L-) (BC/)
A	(AK/L-) (BC/) A
-	(AK/L-) (ABC/-)
*	ABC/-AK/L-*

● Prefix to infix. Operand stack

Iterate exp^n in reverse order,

1. If character is operand, push.
2. If character is operator,
   i) Pop op1
   ii) Pop op2
   iii) Perform (op1 operator op2) & push.
3. Once iteration is completed, initialize result string & pop out from stack & add to the result.

eg. Prefix * - A / B C - / A K L ←

Token	Stack
L	L
K	L K
A	L K A
/	L (A/K)
-	((A/K) - L)
C	((A/K) - L) C
B	((A/K) - L) C B
/	((A/K) - L) (B/C)
A	((A/K) - L) (B/C) A
-	((A/K) - L) (A - (B/C))
*	((A - (B/C)) * ((A/K) - L))

- **Postfix to infix :** Operand stack

Iterate, one char at a time,

1. If operand, push.
2. If operator,
   pop op1, pop op2, push (op2 op op1)
3. Once iteration is complete, initialize the result string & pop out & add to result.

eg. Postfix : A B C / - A K / L - *

Token	Stack
A	A
B	AB
C	ABC
/	A (B/C)
-	(A - (B/C))
A	(A - (B/C)) A
K	(A - (B/C)) A K
/	(A - (B/C)) (A/K)
L	(A - (B/C)) (A/K) L
-	(A - (B/C)) ((A/K) - L)
*	((A - (B/C)) * ((A/K) - L))

● Postfix to prefix Operand stack

Iterate

1. If operand, push.
2. If operator,
   pop op1, pop op2, push (op op2 op1)
3. Once iteration is complete, initialize the result string & pop out from the stack & add it to the result.

* Sort stack using temporary stack :

1. While input stack is not empty, do
   i) Pop from input stack, say temp.
   ii) While temporary stack is not empty & top of temporary stack is > temp, pop from temporary stack & push into the input stack.
   iii) Push temp into temporary stack.
2. Sorted numbers in temporary stack.

eg. (34, 3, 31, 98, 92, 23)

Elem. popped from input stack	Input	Temp. Stack
23	34, 3, 31, 98, 92	23
92	34, 3, 31, 98	23, 92
98	34, 3, 31	23, 92, 98
31	34, 3, 98, 92	23, 31
92	34, 3, 98	23, 31, 92

Popped elem.	Input	Temp. stack.
98	34, 3	23, 31, 92, 98
3	34, 98, 92, 31, 23	3
23	34, 98, 92, 31.	3, 23
31	34, 98, 92	3, 23, 31
92	34, 98	3, 23, 31, 92
98	34	3, 23, 31, 92, 98
34	98, 92	3, 23, 31, 34
92	98	3, 23, 31, 34, 92
98	[]	3, 23, 31, 34, 92, 98

* Implement stack using queues.

S, q_1, q_2

Method 1. Newly entered element is always at the front of q_1, so that pop just dequeues from q_1.

Enqueue - to rear
Dequeue - at front

push (s, x)

1. enqueue x to q_2.
2. One by one dequeue everything from q_1 & enqueue to q_2.
3. Swap names of q_1 & q_2. (to avoid one more movement of all elements from q_2 to q_1)

pop(s)

1. Dequeue an item from q_1 & return.

Method 2

New elem enqueued to q_1. In pop(), if q_2 is empty then all the elems except the last, are moved to q_2. Finally last elem is dequeued from q_1 & returned.

push (s, x)

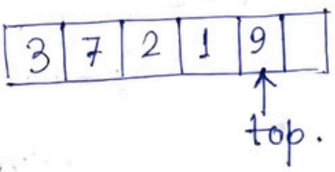
Enqueue x to q_1.

pop

1. One by one dequeue everything except the last elem from q_1 & equeue to q_2.
2. Dequeue last elem of q_1 & it is the result.
3. Swap names $q_1 \leftrightarrow q_2$.
4. Return item of step 2.

NS

- **Application of Stack.** : Balancing of symbols, infix-postfix conversion, evaluation of postfix expression, implementing function calls, finding of spans (spans in stock markets), page visited history in web browser (back buttons), undo sequence in text editor, matching tags in HTML & XML, parsing, backtracking.
- **Simple Array Implementation.** S



Code

```
typedef struct ArrayStack {
        int top;
        int capacity;
        int *array;
} Stack;

Stack *CreateStack() {
        Stack *S = (Stack*) malloc (sizeof(Stack));
        if (!S)
            return NULL;
        S → capacity = 1;
        S → top = -1;
        S → array = (int*) malloc (S → capacity * sizeof(int));
        if (!S → array)
            return NULL;
        return S;
}
int IsEmptyStack (Stack *S) {
        return (S → top == -1);
}
```

```
int IsfullStack ( Stack *S) {
    return ( S→top == S→capacity - 1);
}
void Push ( Stack *S , int data) {
    if (InfullStack (S))
        printf ("Stack overflow!");
    else {
        (S→top)++;
        S→array [S→top] = data;
    }
}
```

```
S→array [++S→top]
    = data ;
```

```
int Pop (Stack.*S) {
    if (IsEmpty Stack(S)) {
        printf ("Stack Underflow!");
        return 0;
    }
    else {
        (S→top)++;
    return S→array [S→top]
        (S→top)--;
    }
}
```

```
return (S→array [S→top--]);
```

```
void DeleteStack ( Stack *S) {
    of (S) {
        if (S→array)
            free ( S→array);
        free (S);
    }
}
```

```
void Display (Stack *S)
{
    ant i;
    if ( S→top != -1) {
    for ( i=S→top; i >= 0; i--)
    {printf ("%d", S→array[i])
    }
}
```

* Performance.

→ Space Complexity (for n push) - O(n)

→ TC of push, pop, isempty, isfull, delete - O(1).

* Limitations.

The maximum size of the stack must be defined in prior & cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

• Dynamic Array Implementation.

By array doubling technique; if the array is full, create a new array of twice the size & copy items then push.

For n push operations we double the array size logn times.

T(n) for n push operations =

$$1 + 2 + 4 + 8 + \cdots + \frac{n}{4} + \frac{n}{2} + n$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right).$$

$$= 2n = O(n).$$

Amortised time of push operation is O(1).

[DS2,]
Intro

Code

```
typedef struct DynArrayStack {
    int top;
    int capacity;
    int *array;
} Stack;

Stack *CreateStack() {
    Stack *S = (Stack *) malloc (sizeof (Stack));
    if (!S)
        return NULL;
    S→capacity = 1;
    S→top = -1;
    S→array = (int *) malloc (S→capacity * sizeof (int));
    if (!S→array)
        return NULL;
    return S;
}
int IsFullStack( Stack *S) {
    return ( S→top == S→capacity - 1);
}
int IsEmptyStack ( Stack *S) {
    return ( S→top == -1);
}
Void DoubleStack (Stack *S) {
    S→capacity *= 2;
    S→array = (int *) realloc (S→array, S→capacity);
}
```

```
void Push(Stack *S, int data) {
    if (IsFullStack(S))
        DoubleStack(S);
    S->array[++S->top] = data;
}
void Peek(Stack *S) {
    if (IsEmptyStack(S))
        printf("Underflow");
    printf("Top element - %d", S->array[S->top]);
}
int Pop(Stack *S) {
    if (IsEmptyStack(S))
        return INT_MIN;
    return S->array[S->top--];
}
void DeleteStack(Stack *S) {
    if (S) {
        if (S->array)
            free(S->array);
        free(S);
    }
}
void Display(Stack *S) {
    int i;
    if (S->top != -1) {
        for (i = S->top; i >= 0; i--)
            printf("%d\t", S->array[i]);
    }
}
```

* Performance.

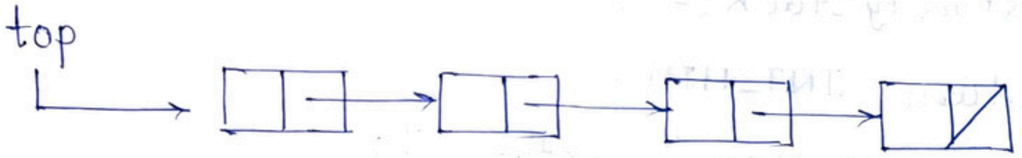
Space complexity (for n push) - O(n).

TC of CreateStack, push, pop, peek, isempty, isfull, deletestack - O(1)

→ Too many doublings may cause memory overflow exception.

- Linked List Implementation.

Push is implemented by inserting element at the front of the list. Pop is implemented by deleting from front.



Code

```
typedef struct ListNode {
    int data;
    struct ListNode *next;
} Stack;

Stack *CreateStack () {
    return NULL;
}
void Push ( Stack **top, int data) {
    Stack *temp;
    temp = (Stack *) malloc ( sizeof (Stack));
    if (!temp) {
        return ;
    }
```

```
        temp -> data = data;
        temp -> next = *top ;      // Imp.
        *top = temp;
}
int IsEmptyStack (Stack *top) {
        return top == NULL;
}
int Pop ( Stack **top) {
        int data;
        Stack *temp;
        if (IsEmptyStack (*top))
                return INT_MIN;
        temp = *top;
        *top = *top -> next;
        data = temp -> data;
        free (temp);
        return (data);
}
int Peek ( Stack *top) {
        if (IsEmptyStack ( top))
                return INT_MIN;
        return top -> data;
}
void DeleteStack (Stack **top) {
        Stack *temp, *p;
        p = *top;
        while (p -> next) {
                temp = p -> next;
                p -> next = temp -> next;
                free (temp);
        }
```

```
    free(p);
}
void Display (Stack **top) {
    Stack *temp = *top;
    while (temp != NULL) {
        printf ("%d\n", temp->data);
        temp = temp->next;
    }
}.
```

* Performance.

Space complexity for n push - O(n).

TC for create stack, push, pop, peek, isempty, delete - O(1).

Comparison of Implementations.

→ Incremental Strategy. -

The amortized time (average time per operation) of a push operation is O(n).

→ Doubling strategy -

The amortized time of a push operation is O(n)/n or O(1).

→ Array Implementation. -

- Operations take constant time.
- Expensive doubling operation.
- Amortised bound takes time proportional to n.

→ Linked List Implementation.

- Every operation takes constant time.
- Every operation takes extra space & time to deal with references

Q. Let the min. no. of stacks reqd. to evaluate a prefix expression is A & the value of the prefix expression --+2*34+/8215 evaluated using the same # of stacks is B. A, B = ?

→

--+2*34+/8215

--+2 12 + 4 1 5

-- 14 5 5

- 9 5

4.

1 stack (operator stack)

A = 1
B = 4

↓ or reverse ↓ operand stk