Data Structures

- way to organize and store data in computer to access it efficiently
  (main memory)
  - efficiently → time
  - efficiently → space

* Classification:
  (i) Linear Data Structures — array, linked list, stack, queue
  (ii) Non-linear Data Structures — tree, graph

Data type
- set of elements/values that can be stored in that type of variable
- set of operations that can be performed to that type of variable

```
int a;          float a;
a%2  ✓          a%2  X (NOT allowed)
```

Linear List

- a collection of similar (same data type) elements

e.g. – array, linked list

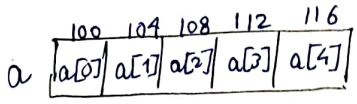
Array

- a linear list in which data is stored contiguously in memory
- insertion and deletion can be done from any index
- size is fixed at declaration (in many languages)

Linked List

- a linear list but isn't stored contiguously in memory
- size is not fixed at declaration

int a[5];



(if int takes 4 bytes)

&a[0] = base address of array = 100

&a[1] = 104

&a[4] = 116

* a → pointer to the base address of array (&a[0])
* a[i] is the same as i[a]

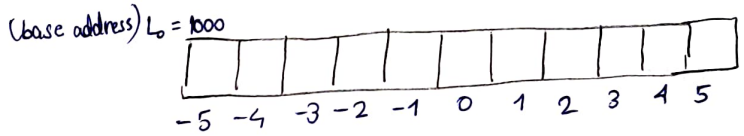
& → address operator

* → value at address operator

a[i] = *(a+i)

i[a] = *(i+a)

Q.) A 1-D array is defined as A: Array [-5...5] (lower bound L_b, upper bound U_b). Each element takes 2B and base address is 1000. Find &A[0]. (indexing)



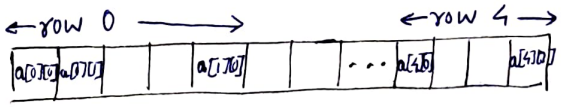
&A[0] = 1000 + 2*5

= 1010

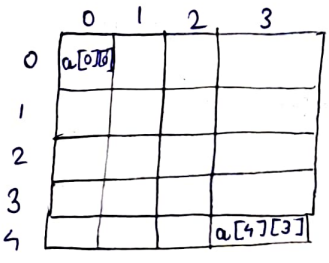
&A[i] = L_0 + (size of each element)* ($i - L_b$)

2-D array

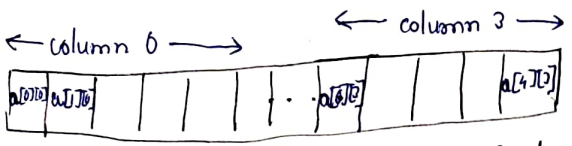
int a[5][4];



Memory (Row-major Order)



Logical View

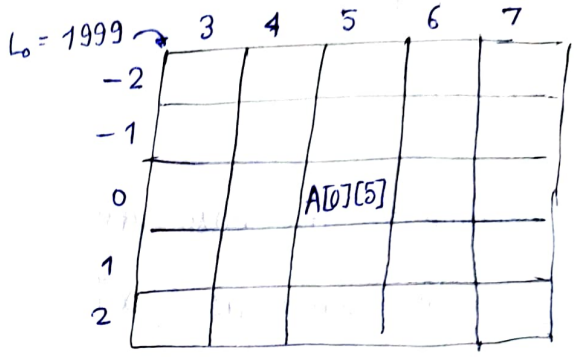


(Column-major Order)

Q.) An array is declared as A : Array [-2 ... 2, 3 ... 7]. Each element is 2B. (row indexing, column indexing; elements indexing)
Base address is 1999. Find

&A[0][5]

in R.M.O and C.M.O



(i) R.M.O

$$\&A[0][5] = 1999 + 12*2$$
$$= 2023$$

(ii) C.M.O

$$\&A[0][5] = 1999 + 12*2$$
$$= 2023$$

$$\&A[i][j] = \begin{cases} L_0 + size * [(i - L_{b1})(U_{b2} - L_{b2} + 1) + (j - L_{b2})], & \text{R.M.O} \\ L_0 + size * [(j - L_{b2})(U_{b1} - L_{b1} + 1) + (i - L_{b1})], & \text{C.M.O} \end{cases}$$

* Diagonal elements in square matrix will have same address in both R.M.O and C.M.O.

Introduction to Algorithms

- Algorithms are finite set of steps to solve a problem.

Criteria for Algorithms:

(i) Input (0 or more external inputs)
(ii) Output (1 or more outputs)
(iii) Definiteness (every instruction is unambiguous)
(iv) Finiteness (algorithm must terminate)
(v) Effectiveness (all steps must be feasible)

Analysis of Algorithms

↳ Apriori analysis
↳ Apost analysis

Asymptotic Notation

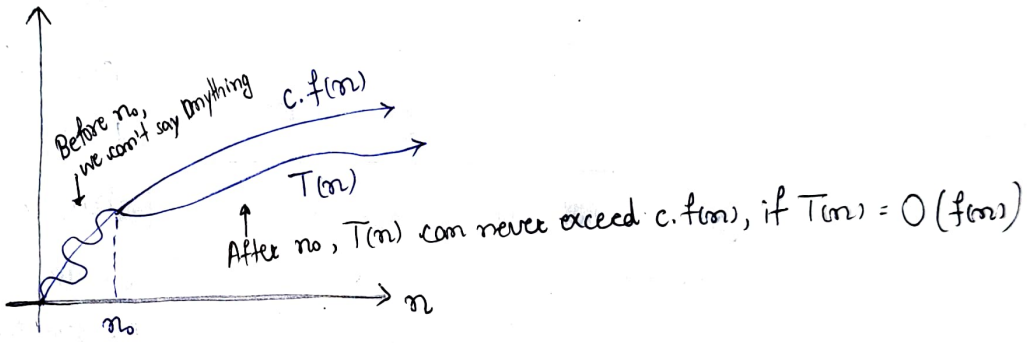
– to estimate the time and space needed by an algorithm

(I) Big oh (O) Notation

$T(n) = O(f(n))$ iff, $\exists$ (+ve) constants 'c' and 'n_0' such that

('c' → real no.; 'n_0' → integer)

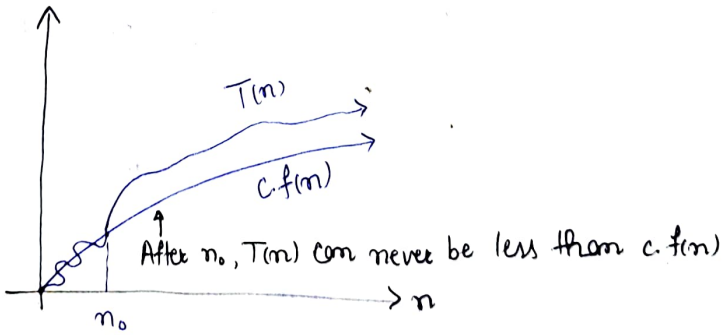
$$T(n) \leq c.f(n) \quad \forall n > n_0$$



(II) Big Omega (Ω) notation

$T(n) = \Omega(f(n))$ iff, $\exists$ (+ve) constants 'c' and 'n_0' such that

$$T(n) \geq c.f(n) \quad \forall n > n_0$$



Linked List

```
struct Node{
    int info;
    struct Node* link;
};
```

} self-referential structure

```
int Do(struct Node* p){
    if(p){
        return (1+Do(p->link));
    }
    return 0;
}
main(){
    printf("%d", Do(s));
}
```

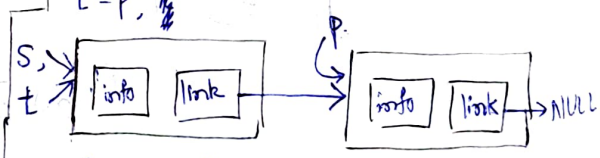
↳ Output: No. of nodes

"Singly Linked List" — only one link field
- can only move forward

* to display information:

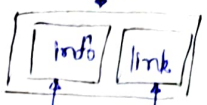
```
t = s
while(t){
    printf("%d", t->info);
    t = t->link;
}
```

```
struct Node *s, *t, *p;
s = (struct Node*)malloc(sizeof(struct Node));
scanf("%d", s->info);
t = s;
char ch = 'y';
while(ch == 'y'){
p = (struct Node*)malloc(sizeof(struct Node));
scanf("%d", p->info);
t->link = p;
t = p;
```




```
ch = getchar();
}
t->NULL;
```

```
struct Node s;
```




```
struct Node *sp;
s.info = (*sp).info
       = sp->info
s.link = (*sp).link
       = sp->link
```

---

* Unlike arrays, space isn't wasted in linked lists
* Arrays can access elements in constant time, while linked list take linear time

Insertion Sort

	Cost	Times
Algo Is(A,n){		
for i=2 to n{	c_1	n ($n-1$ → success + 1 → failure)
$j = i-1$	c_2	$n-1$
$x = A[i]$	c_3	$n-1$
while ($j>0$ AND $A[j]>x$){	c_4	$\sum_{i=2}^{n} t_i$, t_i → cost involved in ith iteration
$A[j+1] = A[j]$	c_5	$\sum_{i=2}^{n} (t_i - 1)$
$j = j-1$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
}		
$A[j+1] = x$	c_7	$n-1$
}		
}		

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\sum_{i=2}^{n} t_i\right) + c_5\left(\sum_{i=2}^{n} (t_i - 1)\right) + c_6\left(\sum_{i=2}^{n} (t_i - 1)\right) + c_7(n-1)$$

* Running time depends on input size and also nature of input

- Best Case: $t_i = 1$ (already sorted in ascending order)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \cdot 0 + c_6 \cdot 0 + c_7(n-1)$$

$$T(n) = an + b$$

If, ~~Here~~ $T(n) = O(n) = O(n^2)$

$T(n) = \Omega(n) \neq \Omega(n^2)$

$\therefore T(n) = O(n)$ and $T(n) = \Omega(n)$

$\therefore T(n) = \Theta(n)$

* Best case, worst case, average case depend on nature of input, not size

(iii) Big Theta (Θ) notation

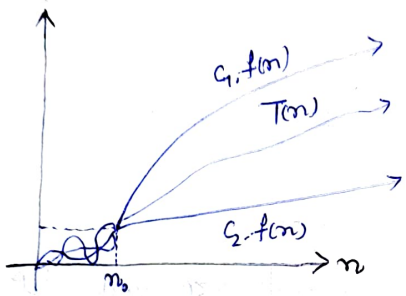
$T(n) = \Theta(f(n))$, iff

$T(n) = O(f(n))$ and, $T(n) = \Omega(f(n))$

$\Downarrow$ $\Downarrow$

$T(n) \leq c_1 \cdot f(n), \forall n \geq n_1$ and, $T(n) \geq c_2 \cdot f(n), \forall n \geq n_2$

$$c_2 \cdot f(n) \leq T(n) \leq c_1 \cdot f(n), \forall n \geq \max(n_1, n_2)$$



- Worst case: $t_i = i$ (elements sorted in descending order)

$$\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i$$

$$= \frac{n(n+1)}{2} - 1$$

$$\sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i-1)$$

$$= \frac{(n-1)(n)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left[\frac{n(n+1)}{2} - 1\right] + c_5\left[\frac{(n-1)n}{2}\right] + c_6\left[\frac{(n-1)n}{2} - \right.$$

$$+ c_7 \cdot (n-1)$$

$$T(n) = an^2 + bn + c$$

Here, $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$

$$\therefore T(n) = \Theta(n^2)$$

Examples:

(I)
```
main(){
    x=y+z;        // 1
}
```
→ O(1) ⇒ "constant time complexity"

(II)
```
main(){
    x=y+z;        // 1
    for(i=1; i<=n; i++){        // n+1
        x=x+y;    // n
    }
}
```
→ O(n) ⇒ "linear time complexity"

(III)
```
main(){
    x=a+b;        // 1
    for(i=1; i<=n; i++){        // n+1
        x=y+z;    // n
    }
    for(i=1; i<=n; i++){        // n+1
        for(j=1; j<=n; j++){    // n(n+1)
            a=a+b;  // n^2
        }
    }
}
```
→ O(n²) ⇒ "quadratic time complexity"

(IV)
```
main(){
    while(i<=n){        // n+1 (if 'i' is initialized as 1)
        i=i+1;          // n
        x=y+z;          // n
    }
}
```
→ O(n)

(V)
```
main(){
    i=1;
    while(i<=n){
        i=i+1;
        i=i+5;
        i=i+8;
    }
    a=b+c;
}
```
→ O(n)

(VI)
```
main(){
    i=1;
    while(i<=n){        // k+1
        i=2*i;          // k
        a=b+c;          // k
    }
}
```
$k = \log_2 n$

→ $O(\log_2 n)$

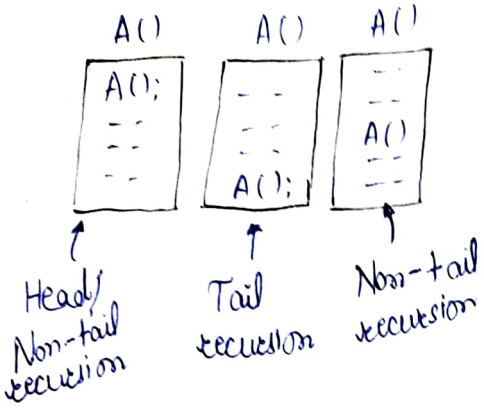
⇓

"logarithmic time complexity"

Recursion

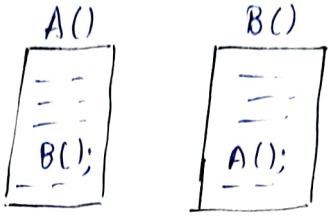
(I) Direct Recursion

- Function calling itself



(II) Indirect Recursion

- Function calls another function which then calls the first function



(III) Excessive Recursion

- recursive calls occur more than once

e.g. - $fib(n) = fib(n-1) + fib(n-2)$

(IV) Nested Recursion

- recursive call itself has a recursive call as argument

e.g. - $f(f(n-1))$

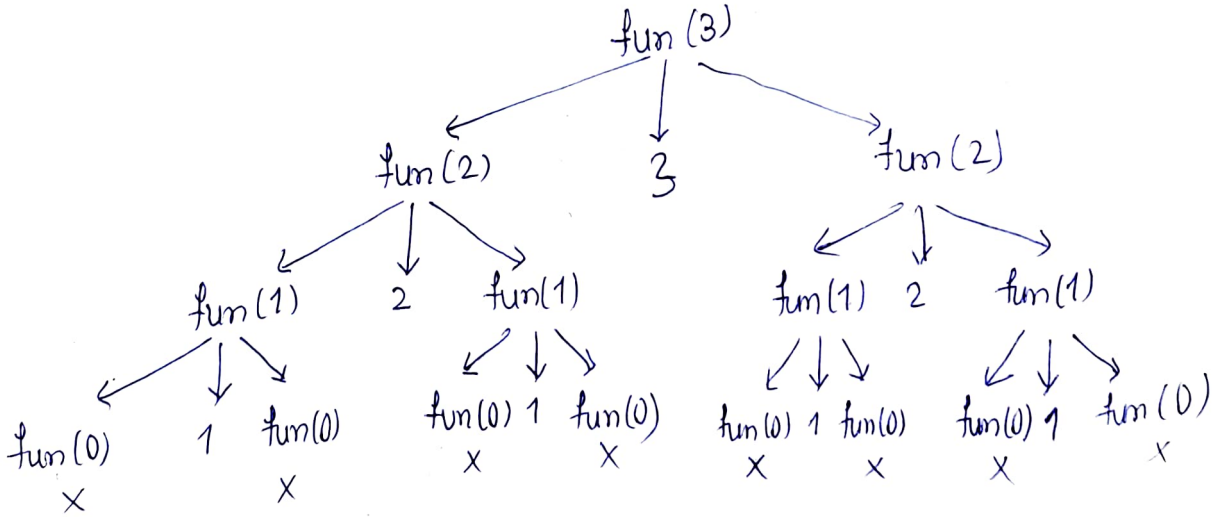
Example:

```
void fun(int x) {
    if (x > 0) {
        fun(x-1);
        printf("%d", x);
        fun(x-1);
    }
}
```

- fun(3)

↳ 1 2 1 3 1 2 1

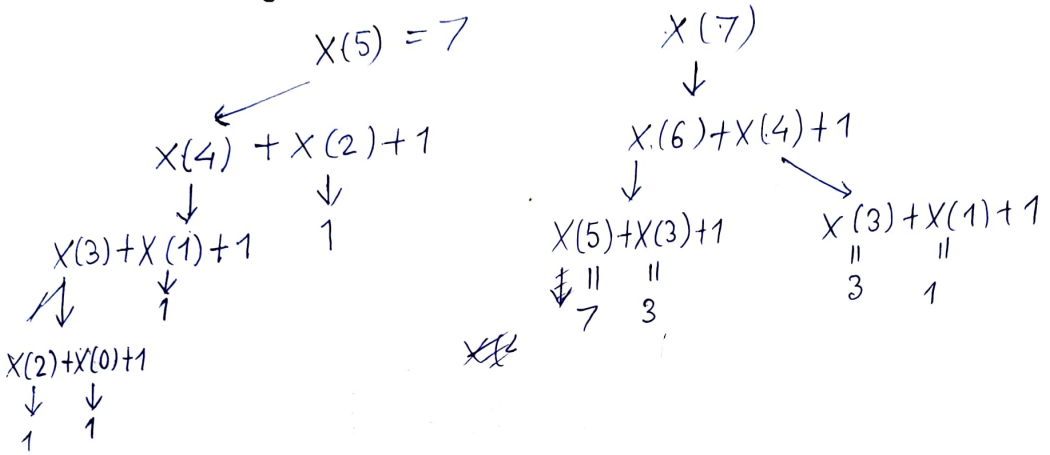
- no. of invocations = 15



Example:

```
int X (int N) {
    if (N < 3)
        return 1
    return ( X(N-1) + X(N-3) + 1 );
}
```

Find no. of invocations that take place while evaluating in X(X(5))



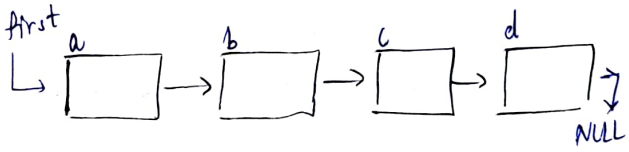
Ans: 24 ← No. of invocations

Value = 17

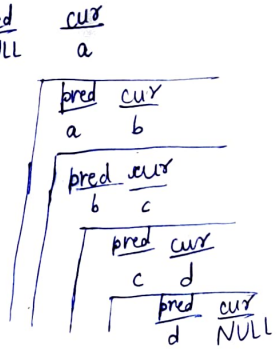
Example:

```
void R (struct Node* pred, struct Node* cur){
    if (cur){                          // address of next node
        R (cur, cur -> link);
        cur -> link = pred;
    }
    first = pred;
}
```

```
main(){
    R (NULL, first);                   // (global) pointer to first node of linked list
}
```



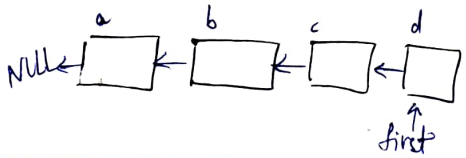
* Reversal of a singly-linked list



first

~~a b c d NULL~~

d



Examples:

(I)
```
main () {
    i = n;
    while (i > 1) {
        i = i/2;
    }
}
```
→ $O(\log_2 n)$

(II)
```
main () {
    i = 1;
    while (i <= n) {
        i = i * 2;
    }
}
```
←

(III)
```
main () {
    i = 0; s = 0;
    while (s < n) {
        i = i + 1;
        s = s + i;
    }
}
```
→ $O(\sqrt{n})$

i	s
0	0
	$0 < n$
1	$0+1 < n$
2	$0+1+2 < n$
3	$0+1+2+3 < n$
⋮	⋮
K	$0+1+2+\dots+K = n$

$$\Rightarrow \frac{K(K+1)}{2} = n$$

$$\Rightarrow K^2 + K = 2n$$

$$\Rightarrow K^2 \approx n$$

$$\Rightarrow K \approx \sqrt{n}$$

(iv)
```
main(){
    int i;
    for(i=1; i<=n; i++){          // 2 times
        for(i=1; i<=n^2; i++){    // 2 times
            for(i=1; i<=n^3; i++){ // n^3+1 times   -> O(n^3)
                u=y+z;
            }
        }
    }
}
```

i
✗
✗
✗ ✗ ✗ ... ✗

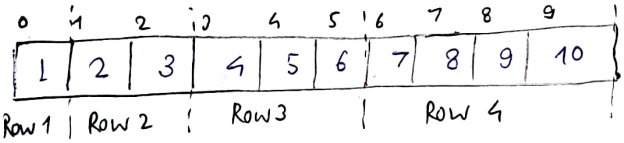
n^3+1

Lower Triangular Matrix Representation

	1	2	3	4
1	1	0	0	0
2	2	3	0	0
3	4	5	6	0
4	7	8	9	10

4x4

→ Lower triangular matrix
(sq. mat A; A[i][j]=0 iff i<j)



Storing in memory using 1D array (row-major order)

- Each element can be uniquely identified by:
  - row no.
  - column no.
  - value itself

```
Data(i,j){
    if (i<j) return 0;
    else{
        return a[i(i-1)/2 + j - 1];
    }
}
```

// If the matrix starts index at i=0, j=0

return a[i(i+1)/2 + j];

Sparse Matrix Representation

	0	1	2	3	4
0	7	0	0	0	1
1	0	0	2	0	0
2	0	0	0	0	0
3	0	3	0	0	0
4	1	0	0	4	0

M — 5x5

Row	0	0	1	3	4	4
Column	0	4	2	1	0	3
Value	7	1	2	3	1	4

SM

2Xsize ↓ no. of non-zero elements

* Sparse Matrix: more than half elements are zero
- Size can be estimated as $\frac{mn}{2}$, because by def" no. of non-zero elements can't be more than that.

```
int S.M.[3][size];
```

```
//Code to create efficient representation of sparse matrix.
int i=0,j1=0;
for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        if (M[i][j]!=0){
            S.M.[0][j1]=i;
            S.M.[1][j1]=j;
            S.M.[2][j1]=M[i][j];
            j1++;
        }
    }
}
```

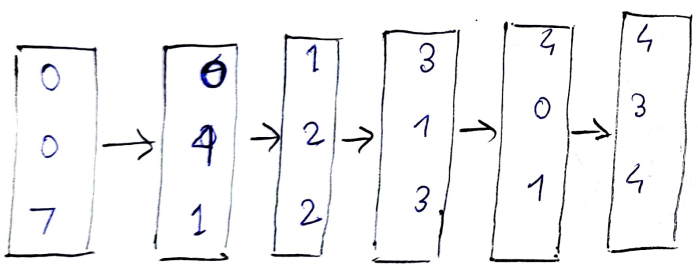
//Code to perform transpose of M

Row	0	4	2	1	0	3
Column	0	0	1	3	4	4
Value	7	1	2	3	1	4

```
for(int k=0; k<size; k++){
    int temp = SM[0][k];
    SM[0][k] = SM[1][k];
    SM[1][k] = temp;
}
```

// Linked List Implementation

```
struct SM_Node{
        int row;
        int col;
        int val;
        struct SM_Node* link;
};
```



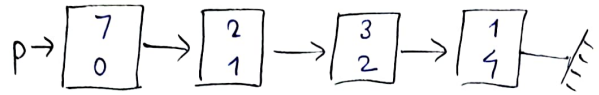
Q.) Write code to add two sparse matrices when implemented in (i) array form (ii) linked list form

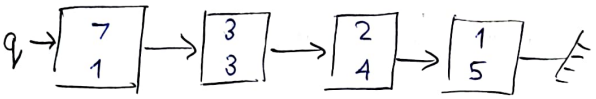
Representation of Polynomial using linked list

$$a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

```
struct pnode {
    int coeff;
    int exp;
    struct pnode * link;
};
```

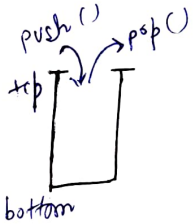
e.g. - $7 + 2x + 3x^2 + x^4$; $7x + 3x^3 + 2x^1 + x^5$





Stack

- Linear List
- LIFO (Last-In-First-Out)
  - ↳ data is added and removed from same side (top of stack)



push() → adds element to top of stack
pop() → removes element from top of stack

Array Implementation:

```
int stack[5];
int top = -1 (signifying stack is empty)
```

Push():

```
top++;
stack[top] = element
```

(top points to lastly inserted element / holds its index)

```
// Check before pushing
if (top == N-1) printf("Stack Overflow");
```

Pop():

```
top--;
//We can also return the popped element
int temp = stack[top--];
return temp;
```

```
// Check before popping
if (top == -1) printf("Stack Empty");
```

Linked list Implementation (Linked Stack):

```
struct Node{
    int data;
    struct Node* next;
};
struct Node* top=NULL;
void push(ele){
    struct Node* temp=(struct Node*)malloc(sizeof(struct Node)); if (p==NULL) return;
                                                                 //Stack Overflow
    temp->next=top;
    top=temp;
    temp->data=ele;
}
int pop(){ if(top==NULL) return; //Stack Underflow
    temp=top;
    top=top->next;
    return temp->data;
}
```

Application of Stacks

a=8, b=4, c=2, d=3, e=5, f=2, g=1

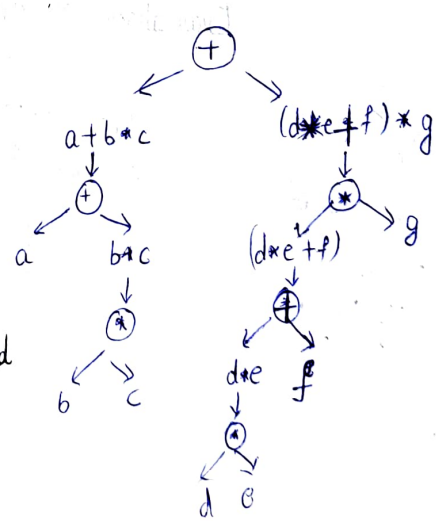
x = a+b*c +(d*e+f)*g //Infix notn

Converting Infix to Postfix:

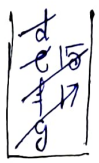
a + b * c + (d * e + f) * g
⑤ ③ ⑥ ① ② ④

root, left, right //Prefix

++a*bc*+*defg



← scan from operand end



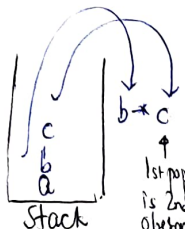
Stack

left, right, ~~root~~:

abc*+de*f+g*+

→ scan from operand end

* Push operand to Stack and perform operation when operator is encountered.



Infix	Prefix	Postfix
$-b$	$-b$	$b-$
$\log x$	$\log x$	$x \log$
$x!$	$!x$	$x!$

Operators	Associativity
$\wedge / \uparrow / **$	Right
$*, /$	Left
$+, -$	Left
$=$	Right

precedence ↓es

$a \uparrow b \uparrow c = a^{b^c}$

Q) Evaluate:

8 2 3 ^ / 2 3 * + 5 1 * +

$= 8 + 2 \wedge 3 / 2 * 3 + 5 * 1$

$= 8 + 8 / 2 * 3 + 5 * 1$

$= 8 + 4 * 3 + 5$

$= 8 + 12 + 5$

$= 25$

$= 8 / 2 \wedge 3 + 2 * 3 + 5 * 1$

$= 8 / 8 + 6 + 5$

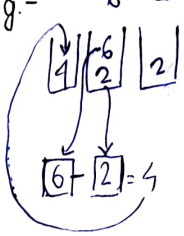
$= 1 + 6 + 5$

$= 12$

Evaluation of Arithmetic Expression

Prefix
(Right to left)

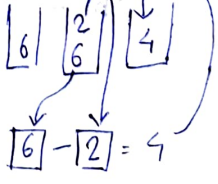
e.g. - - 6 2



operand: push
operator: (i) pop (twice for binary ops)
(ii) evaluate
(iii) push (result)

Postfix
(Left to right)

e.g. - 6 2 -



* NEVER push operator onto the stack while evaluating.

Q.) If the input sequence is given as 1, 2, 3, 4, 5, then identify valid or invalid stack permutations

(a) 3, 4, 2, 5, 1 ✓

(b) 4, 3, 5, 2, 1 ✓

(c) 2, 5, 4, 3, 1 ✓

(d) 5, 4, 3, 1, 2 ✗

* Application of Stack: Generating permutations

Multiple Stacks

Q.) In an array of size 'm', there are 'n' stacks. The initial configuration is as follows:

$B[i] = T[i] = i\left\lfloor \frac{m}{n} \right\rfloor - 1$; for $0 \le i < n$

where, B: bottom : fixed

T: top : moving

```
void push(int i, int x) {
    if ( ? )  // T[i] == B[i+1]
        printf("Stack[i] is Overflow");
    else
        s[++T[i]] = x;
}
```

```
int pop(int i){
    if ( ? ) { // T[i] == B[i]
        printf("Stack[i] is Underflow");
        return -1;
    }else{
        return s[T[i]--];
    }
}
```

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

(b) T[i] = B[i+1] ← push (top of i^{th} stack = bottom of $(i+1)^{th}$ stack)

(c) T[i] = B[i] ← pop (top of ith stack = bottom of ith stack)

(d) T[i] = B[i-1]

* Application of Stack: Conversion of infix to postfix/prefix

• An arithmetic expression is a valid combination of operators and operands.

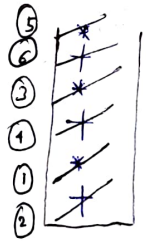
Infix to Postfix Conversion using Stack

1. Read the infix from left to right
2. Initialize an empty stack
3. While (more input):
   (a) if '(' read : push onto stack
   (b) if operand is read : ~~push onto~~ stack add to output
   (c) if ')' read: (i) pop the stack and add to output until '(' is encountered
      (ii) pop the '(' and discard it
      (iii) discard the ')'.
   (d) if operator is read:
      (i) while (precedence (operator) < predence (top of stack)):
         pop the stack and add to the output
      (ii) push operator onto stack
4. Pop the stack until stack is empty and add each element to output.

* Never push ~~higher~~ lower precedence operator over higher precedence operator

e.g. - a+b*c+d*e+f*g

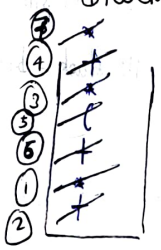
O/P: abc*+de*+fg*+



Stack

a+b*c+(d*e+f)*g

O/P: abc*+de*f+g*+



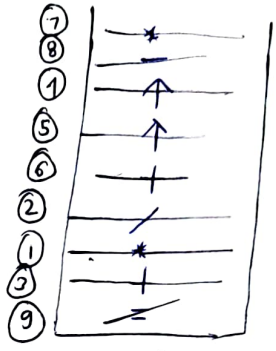
Stack

Modifications:

* if precedence = same (of read operator and top)
   (a) if left associative, pop and add, then push
   (b) if right associative, push
* if top == '(' : push
* if operator is read and top == '(' : push

e.g.- a = b + c * d / e + f ↑ g ↑ h - i * j

o/p: a b c d * e / + f g h ↑ ↑ + i j * - =



Stack

Q.) Write an algorithm to convert infix to prefix using stack.