DATA STRUCTURES AND ALGORITHMS

BSE : 3rd SEMESTER

23/1/24

* Data Storage, Representation and Retriere.
* Data Structures in the form of class (organized form).
* Array is a data structure. (String, date is also data structure).
* Algorithm (Step by step procedure). (code is algorithm to solve problem)
* Date:

```
Class
{ DD
  MM
  YYYY
}
```

3/11/2024
D M Y

* Problems to be solved efficiently. (Timely)

⊙ Data Structure:

Data structure is a way of collecting and organizing

data in such a way that we can perform operations on these data in an effective way.

→ Data Structures are the programmatic way of storing data so that data can be used efficiently or effectively.

⊙ ALgorithm:

A collection of steps to solve a particular problem. It takes input, process input and gives output.

(*) ① Take two numbers
② Add the number
③ Output the final result.

Properties

Any algorithm possess these properties:

①(*) Input: Algorithm should have some input

: class structure
variables, template class
(learn before lab).

(2)(*) At least one output should be returned after completion of task.

(3)(*) Definiteness: Every statement of the algorithm should be unambigious.

(4)(*) Finiteness: After finite number of steps program should terminate.

(5)(*) Effectiveness: Every step of the algorithm should be feasible that it is possible to perform each step in a finite time.

(.) Abstract Data Type (ADT)
Data structure is programmatically version of ADT.

(*) Complex data type:

$z = x + iy$ $\quad i = \sqrt{-1}, \; i^2 = -1$

{-> Domain -> Operations
-> Axioms (Rules)} (Major three aspects of data structure.

-> User enters imaginary number and real number. We convert it into complex form.

operations

$0 = \oplus$

$\ominus$

$\otimes$

conjugate ∇

$\leftarrow$ || magnitude

(*) Complex Z

{ Domain { X real
Y real
}

⊙ Axiom 1:

$z_1 + z_2$

$z_1 = x_1 + iy_1$

$z_2 = x_2 + iy_2$

$z_1 + z_2 = z_3 = (x_1 + x_2) + i(y_1 + y_2)$

```
Add_complex (z1 complex, z2 complex)
{ Complex z3;
  z3.x = z1.x + z2.x;
  z3.y = z1.y + z2.y;
  return z3;
}

Display_complex ()
{
  cout << x << "+" << "i"
       << y << endl;
}
```

Minus_complex (z1 complex, z2 complex)

{ Complex z3;

z3.x = z1.x - z2.x;

z3.y = z1.y - z2.y;

return z3;

}

⊙ Axiom 2:

z1 ⊗ z2

$(x_1 x x_2 - y_1 x y_2) + i(x_1 x y_2 + x_2 x x_1)$

Multiply_complex (z1 complex, z2 complex)

{ Complex z3;

z3.x = ((z1.x * z2.x) - (z1.y * z2.y)

z3.y = (z1.x * z2.y) + (z2.x * z1.y)

return z3;

}

⊛ Axiom 3:

z1 ⊘ z2

$\left(\frac{X_1 x X_2 + Y_1 x Y_2}{X_2^2 + Y_2^2}\right) + i\left(\frac{X_2 x Y_1 - X_1 x Y_2}{X_2^2 + Y_2^2}\right)$

: sqrt functions
: atleast one quiz every week.

Divide_Complex (Z1 complex, Z2 complex

{ Complex Z3;

$Z_3.x = (((Z_1.x * Z_2.y) + (Z_1.y + Z_2.y$

$/ (Z_2.x)(Z_2.x) + (Z_2.y)(Z_2.$

$Z_3.y = (Z_2.x * Z_1.y) -$

* Axiom 4:

⊙ Conjugate = $\nabla Z_1 = \frac{X_1}{x_1^2 + Y_1^2} + i \frac{Y_1}{x_1^2 + Y_1^2}$

$|Z_1| = \sqrt{x_1^2 + y_1^2}$: sqrt($x_1^2 + y_2^2$

25/1/24

* Domain (Primitive Data type).
* Algorithm performance analysis (feasibl

Space Complexity (memory consumption)

Time Complexity (Time consumption)

→ Less space and less time is an efficient program.

(*) Space complexity of an algorithm is the total space taken by algorithm with respect to input size.

(*) Time complexity is the total time taken by algorithm to complete its execution.

(*) (1) Instruction Space ✓
(2) Environmental Space (Stack) ✗
(3) Data space (Input + Auxillary space) ✓

(*) Data space depends upon input data type.

- char - 1
- short - 2
- int - 4
- long --
- float - 4
- Double - 8

(*)
```
sum (int a, int b, int c)
{
    int z = a+b+c;
    return z;
}
```
a b c z = 16 B
4 4 4 4

```
int sum(int a[], int N)
{
    int sum, i=0;
    for (i=0; i<N; i++)
    {
        sum=sum+a[i];
    }
    return sum;
}
```

: Complexity depends on N

a = N x 4
N : 4
sum : 4
i = 4

= 16N

(space complexity)

* Basic Time Complexity is focused. In paper Complexity mean time complexity.

* Time complexity:

- Compile Time (Do not depend upon instance (input) Characters)
- Run Time (Depend on input) (It is mainly focused)

+, -, *

Time required to perform operations.

⊙ Method to find Time Complexity:
1. Seconds count of different operations.
2. Step-count

(*) int sum(int *a, const int N)
{ ——— (0)
int s=0; ——— (1)
for(int i=0; i<N; i++) ——— (N+1)
{ s=s+a[i]; ——— (N)
}
return s; ——— (1)
} ——— (0)

= 0 + 1 + N+1 + N + 1
= 2N + 3 (Time Complexity).

(*) int Abc(int a, int b, int c, int d)
{ ——— (0)
return a*b*(c-d) + 4*(a*d); ——— (1)
} ——— (0)

Big O(1)

= 0+1=0 = (1) - Constant

: Constant Complexity (Not depends)
: Linear Complexity (Depends on input variable)

(*) 3 Arrays, 2 Array into

(*) void add(int **a, int **b, int **c, int m, int n)
{ for(int i=0; i<m; i++) ——— (m+1)
{ for(int j=0; j<n; j++) ——— m(n+1)
{ c[i][j] = a[i][j] + b[i][j]; ——— mn
} }
}

$= m+1 + m(n+1) + mn$ Big O(mn)

$= m+1 + mn+m + mn$

$= (2m + 2mn + 1)$ Time Complexity

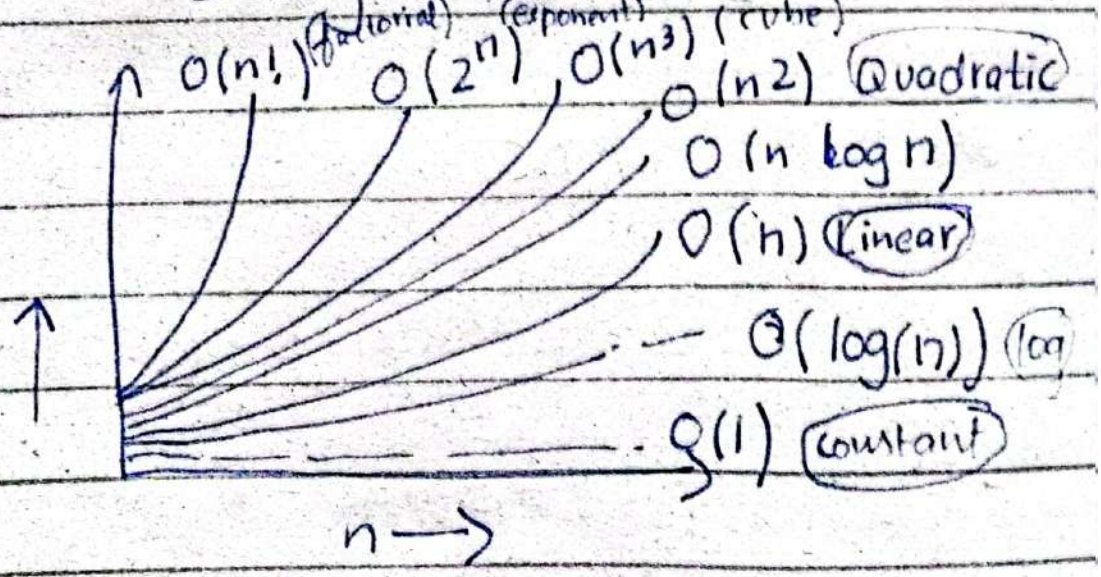
⊙ If $m = n$:

$= n+1 + n(n+1) + n(n)$ Big O(n^2)

$= n+1 + n^2 + n + n^2$

$= (2n^2 + 2n + 1)$ → Quadratic Complexity

⊙ Big-O → Higher terms



⊙ Multiply two matrix:

```
void multiplyMatrix(int** m, int** n,
                    int** r, int mRows,
                    int mColumns,
                    int nRows, int nColumns)
{
    for (int i=0; i< mRows; i++)      (mRows +1)
    { for (int j=0; j<nColumns; j++)  (mRows * nColumns)
```

30/1/24

- ⊙ Logarithmic Complexity
- ⊙ Quadratic Complexity

```
for ( int j=0,i=1 , i<=n; i*=2) — log(n)+1
  { for (j=1 ; j<=n; j++)—
```

(4) for (int j , i=1 ; i<=n; i++) — (n+1)

{ for (j=1, j<=i ; j++) — $\frac{n(n+1)}{2}$

{ cout<< ... — independent i series

} }

: $O(n^2)$

for (cnt4=0, i=1; i<=n; i=*2) — (log(n)+1)

: $1+2+4+\dots \log n$

$= 2^{\log n}$

$\Sigma 2^{\log n} = N-1$

if n=3

1, 2, 4

3-1

{ for (j=1 ; j<=i ; j++) — (N-1)(log N)

{ cnt4++ ;

} }

(5) Series :

(1) $\sum_{K=1}^{N} K = \frac{K(K+1)}{2}$

(2) $\sum_{K=1}^{N} K^2 = \frac{n(n+1)(2n+1)}{6}$ ...

(3) $\sum_{n=0}^{K} x^n = 1 + x + x^2 + x^3 \dots$

(4) $1 + 2^1 + 2^2 + 2^3 \dots 2^n = \sum_{n=0}^{K} 2^n$

⊛ Algorithm to find element (Linear search Algorithm)

```
int findElement (int * arr, int N,
                 int element)
{
    for (int i=0; i<N; i++)  — (N+1)
    { if (arr[i] == element) — (size)
      {
          return i;
      }
    }
    return -1 — (1)
}
```

Worst case check for Big O

⊛ Algorithm Binary search. (Sorted Array)
(Find from mid)

```
int Binarysearch (int *a, int X)
{
    int left=0, right=size, mid;
    while (left <= right)
    { mid = (left + right)/2;
      if (a[mid] == x)
      { return mid;
```

```
else if ( a[mid] > x)
    {   right = mid-1;
    }
  else
      {   left.  = mid+1;
      }
  }
  return -1;
}
```

⊙. In-it Print (int n)
{ n = 1000
Print n; : Constant Complexity
}

In-it Print (int n)
{ n = 1000
for (int i=0; i<4; i++) : Not depend on any variable
{ Print n; : Constant Complexity
}
}

* ⊛ Next Class Quiz.

⊛ Polynomials ADT:

$P_1 = 4x^3 + 3x^2 + 2$: (coefficient, exponent)

$P_2 = 3x^2 + 3$

$a_0 x^{e_0} + a_1 x^{e_1} + a_2 x^{e_2} \ldots\ldots a_n x^{e_n}$

⊙ Operations:

① Add ② Subtract
③ Multiply
④ Evaluate (int type)

→ $P_1 + P_2$:

$P_1 = 4x^3 + 3x^2 + 2$
$P_2 = 3x^2 + 3$
$= 4x^3 + 6x^2 + 5$

: zero coefficient term not display ($2x^0 = 2$)

⊙ $a(x) = \Sigma a_i x^{e_i}$, $b(x) = \Sigma b_j x^{e_i}$

$a+b = \Sigma (a_i + b_j) x_i$

$a-b = \Sigma (a_i - b_j) x_i$

$a \times b = \Sigma (a_i x^{e_i} \cdot \Sigma (b_j \cdot x^{e_j}))$

⊙ Evaluate:

$4x^3 + 3x^2 + 2$: $x = 2$

$4(2)^3 + 3(2)^2 + 2$
$= 32 + 12 + 2 = 46$

(*) We use array index as exponent.

(*) Degree of polynomial (maximum power). : degree

: Maxdegree = n

: coefficient [Maxdegree + 1]

(*) $P_1 = 4x^3 + 3x^2 + 2$

4	3	0	2
0	1	2	3

: nth term = (n - i)

3 → variable

2 → coefficient not displays

0 → not displays.

(*) Display; : degree = n

```
for (int i=0; i<=n; i=i+1)
    { cout << a[i] << "X" << n-i << "+";
    }
```

(*) Polynomial add (Polynomial p)

```
{ if (this.degree > poly.degree)
    { max.poly = this;
      min.poly = poly;
    }
```

else

max.poly = poly;

min.poly = this;

for (int i=0; i< max.poly.degree, i++)

{

max.poly[i] = max.poly[i] + min.poly(i);

⋮

$4x^3 + 6x^2 + 5$

1	2	3	4
0	6	0	5

(*) for (i=0; i<= maxpoly.degree; i++)

{ res.poly[i] = maxpoly[i] + minpoly[i];

}

(*) for (int i=0; i<=max.poly.degree; i++)

{ res.poly[i] = max poly[i];

}

for (int i=0, i<= min.poly degree; i++)

{ res.poly[i] = res.poly[i] + min.poly[i];

}

⊛ $i = max.degree - min.degree$

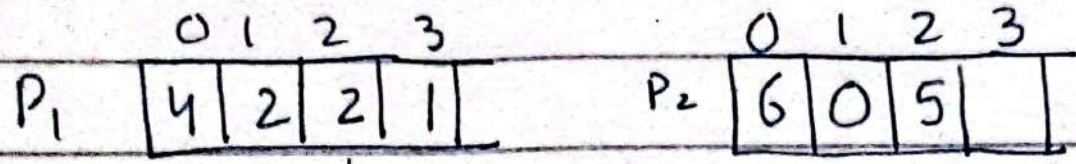
13/2/24

* Polynomial:

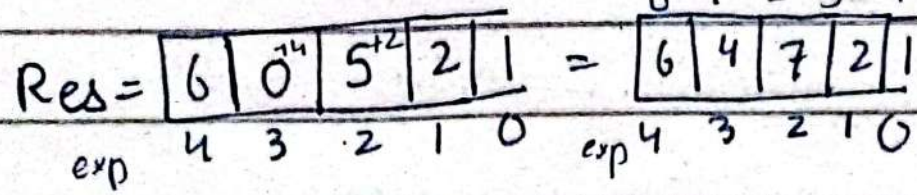
(max degree+1) : Coeff[ ]
Max Degree
degree

P_1 $4x^3 + 2x^2 + 2x + 1$ $d=3$

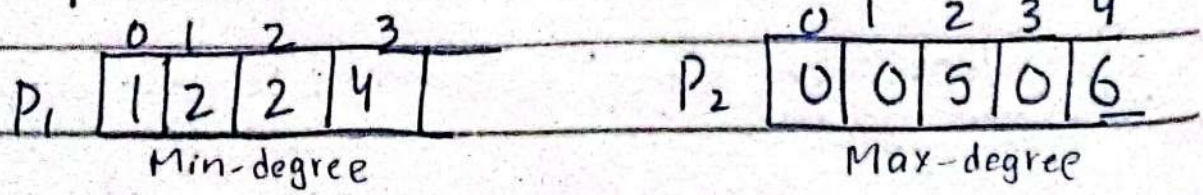
P_2 $6x^4 + 5x^2$ $d=4$



: a_{n-i}



⊛ Reverse-Order:



∴ Res = max-degree

⊙ Add-Poly (Polynomial poly)

```
{ if (degree < poly.degree)
    { Max-degree = poly;
      Min-degree = this;
    }
  else
    { Max-degree = this;
      Min-degree = poly;
    }
```

```
for (int i=0; i<= Mindegree.degree; i++)
    {
      Res.poly[i] = Res.poly[i] + Min.degree[i];
    }
```

0	1	2	3	4
0	0	5	0	6

1	2	7	4	6

→ Resultant

① $P_1 = 4x^3 + 3x^2 + 1$

$6x^4 + 5x$

$24x^7 + 12x^6 + 6x^4 + 20x^4 + 10x^3 + 5x$

$24x^7 + 12x^6 + 26x^4 + 10x^3 + 5x$

```
Multiply-Poly (Polynomial poly)
{
  for (int i=0; i< min.degree; i++)
    {
      Respoly[i]=0;
      for(int j=0; j< max.degree; j++)
        {
          Respoly[i+j] = Respoly[i+j] +
                         (min.degree * max.degree);
        }
    }
```

② `Res-Poly = new Polynomial;`

`Res.Poly.degree = this.degree + poly.degree;`

```
for (i=0; i< this.degree; i=i+1)
  { for (j=0; j< poly.degree; j=j+1)
    {
      Res.poly[i+j] = Res.poly[i+j] +
                      (this.coeff[i] *
                       poly.coeff[j]);
  }  }
```

: $O(n^2 + n)$

** $O(n^2)$

④ Evaluate

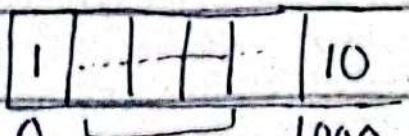
$4x^3 + 2x^2 + 1$

$x = 2$

$4(2)^3 + 2(2)^2 + 1$

: Drawback of Array data in polynomial

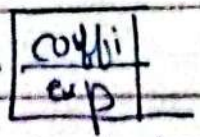
$100x^{1000} + 1$



Waste of memory

(make objects of coefficient and exponents).

④ Term



④ class Polynomial

```
{ Term TermArray[];
  int Capacity;
  int Term;
}
```

$4x_0^3 + 2x_1^2 + 1$

coeff:	4	2	1
exp:	3	2	0

20/2/24

: Term
```
{ coefficient;
  exp;
}
```

: Polynomial
```
{ Term TermArray[];
  capacity;
  term;
}
```

→ $2x^3 + 6x^2 + 1$

coeff	2	6	1
exp	3	2	0

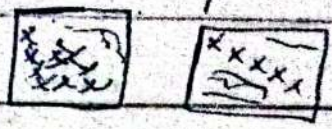
→ $7x^4 - 2x^3$

coeff	7	-2
exp	4	3

: Linear Big O

: Order

: sparse



→ Polynomial Add(Polynomial b)
```
{ Polynomial c;
  c.term = 0;
  c.capacity = a.capacity + b.capacity;
```
(terms) (terms)

C.TermArray = new Term[c.capacity];
int j=0;
for(int i=0; i< a.Term && i<b.Term; i++)
{ if(a.TermArray[i].exp == b.TermArray[i].exp)
{ C.TermArray[j].coefficient = a.TermArray[i].coefficient + b.TermArray[i].coefficient
C.TermArray[j].exp[j] = a.TermArray.exp[i];
j++;
}
}

→ while(apos < Term && bpos < b.Term)
{ TermArray[apos].cap == b.TermArray[bpos].cap;
{ C.TermArray[i] = new Term
((TermArray[apos].coefficient +
b.TermArray[bpos].coefficient),
(TermArray[apos].exp)
i++, apos++, bpos++
}

```
else if ( TermArray[apos].exp < b.TermArray[bpos].exp )
    {  C.TermArray[i] = new Term
                (b.TermArray(bpos].coeff;
                 b.TermArray[bpos].exp);
    }
```

```
=> for(i; apos<= Term; apos++)
    {
        C.TermArray[i] = TermArray[i];
        i++;
    }
for (j; bpos<= b.Term; bpos++)
    {
        C.TermArray[i]=b.TermArray[bpos]
        i++;
    }                          : Arranging.
```

-> $2x^3 - 10x^2 + 15$ — P1

$x^2 - 3x + 2$ — P2

2	-10	15
3	2	0

1	-3	2
2	1	0

Dry Run

a pos = 0 Term = 3

b pos = 0 Term = 3

i = 0;

Term Array [apos] . cap =

C . Term Array [i] = apos

$P_1 - 2x^3 - 10x^2 + 15$

$P_2 - x^2 - 3x + 2$

2	-10	15
3	2	0

1	-3	2
2	1	0

bpos

2	-9	-3	17
3	2	1	0

-> Int Evaluate (int x)

```
{ int res = 0;
  for ( i = 0 ; i < degree ; i++ )
      {
        res = res + This.coff [i] * Pow(x
      }
  return res;
}
```

22/2/24

* (*) Access each object without declaration.

Classical Data structures

Data Structure

Linear D.S
- Detailed study:
  - → Array
  - → Stack
  - → Queue
  - → Linked list

Non-Linear D.S
- → Trees
- → Graphs
- → Tables
- → sets

→ Array:

Return 4th member → A[4]

0	1	2				
6	5	4	3	2	1	0

A[2] = 4

: Strong Data structure

A[0] = 6

DS = Array int

A [L ..... U]

L: lower

U: upper (maximum index)

① Traversing:

Print or reading etc.....

→ Input — Array A with n elements
→ Output — Depend on Procedure Transverse
→ Data structure — Int array with n elements

④ while (i=L; i<=U)
{ process(A(i)); i++;
}
End

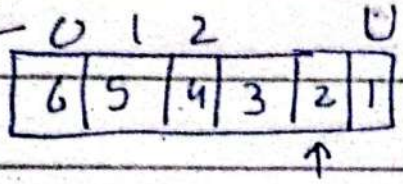
: Big O(n)

⊙ Sorting:

Input : An Array
Output : Sorted Array
D.S = Int Array A[L ..... U]

→ Bubble sort, Linear sort, Insertion sort.

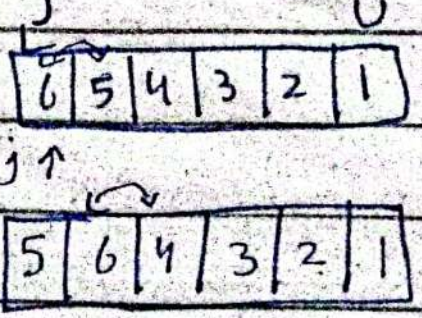
→ Bubble Sort:

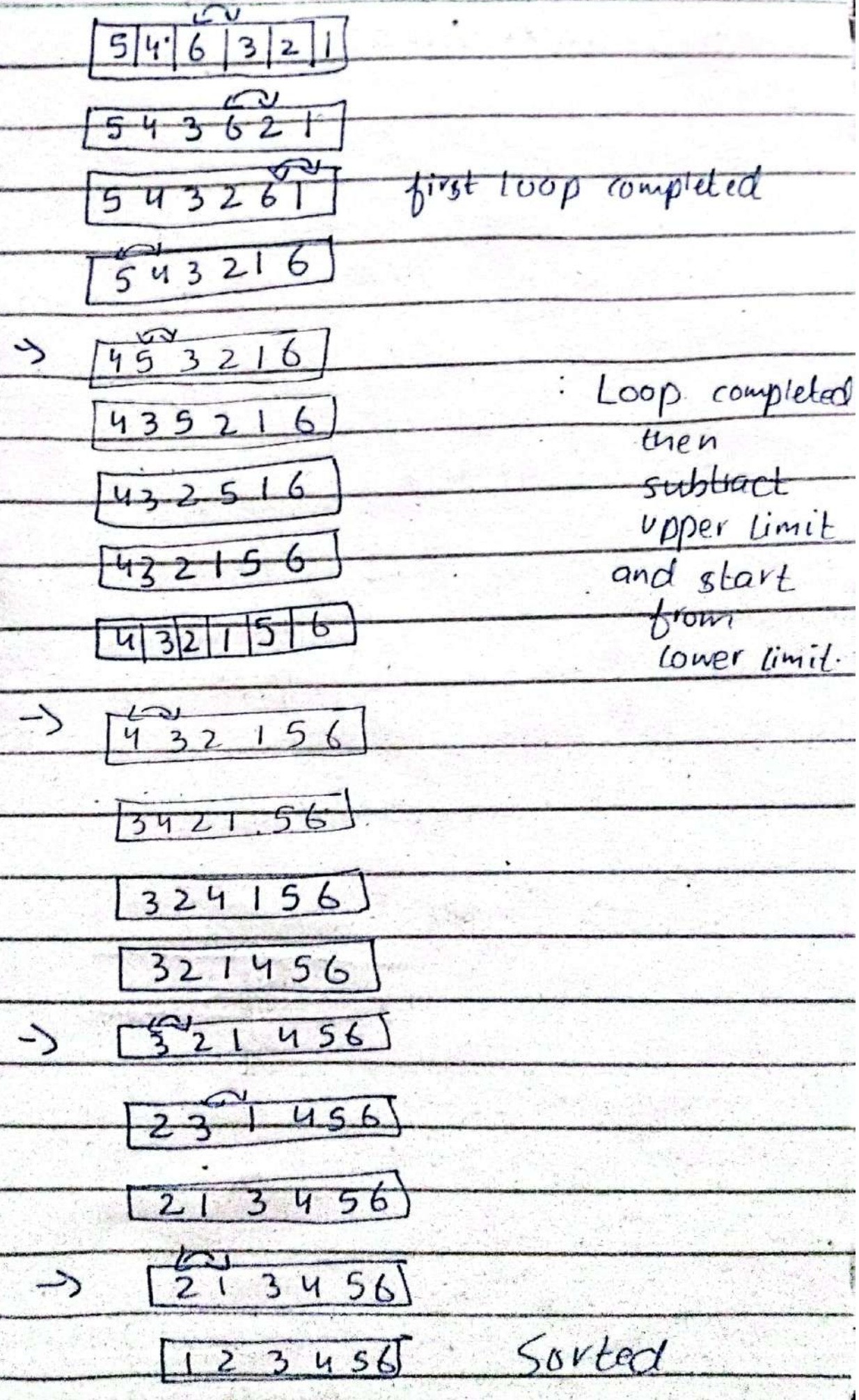



```
=> i = U                                   : Starts from upper limit and sort it
while (i >= L)
{  j = L
   while (j < i)
   {  if (a[j] > a[j+1])
      {  swap(a[j], a[j+1]);
      }
      j++;
   }
   i = i - 1;
}
```

swap a b T

: Big O(n^2)



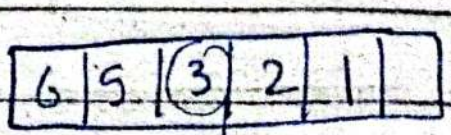

5 4 6 3 2 1
5 4 3 6 2 1
5 4 3 2 6 1
first loop completed
5 4 3 2 1 6
4 5 3 2 1 6
4 3 5 2 1 6
4 3 2 5 1 6
4 3 2 1 5 6
4 3 2 1 5 6
4 3 2 1 5 6
3 4 2 1 5 6
3 2 4 1 5 6
3 2 1 4 5 6
3 2 1 4 5 6
2 3 1 4 5 6
2 1 3 4 5 6
2 1 3 4 5 6
1 2 3 4 5 6
Loop completed then subtract upper limit and start from lower limit.
Sorted

→ Search Array (Key):

Input: Key passes in Array to be searched
Output: Index of Key
D.S : Array int A[L ..... U]

```
→ i=1, found=0, index=-1;
  while(i<=U)
    { if(a[i]==key)
        { found=1;
          index=i;
        }
      i++;
    }
  return index;
```

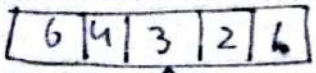
→ Insert element:




```
→ If (U==capacity)
    { return;
    }
  i=U
  while(i>=location)
    { A[i+1]=A[i];
      i--;
```

```
A[index] = E;
U = U+1;
}
```

-> Remove element:

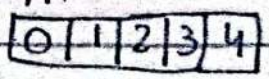


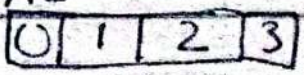
found=0 ss found==0

```
for (int i=0; i<size; i++)
{ if [a[i] == Key)
    j=i; while (j<size)
    { a[i] = a[i+1]
      j++;
    }
```

-> `i = U;`
`while (i<U)`
`{ a[i] = a[i+1];`
`i--;`
`}`

⊛ Merge Arrays (A_1, A_2)





$A_1 = L_1 \text{ ---- } U_1$

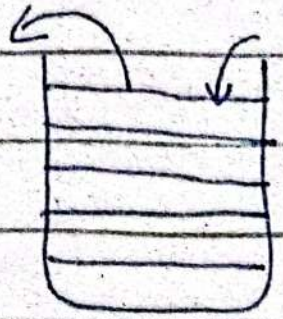
$A_2 = L_2 \text{ ---- } U_2$

Soln:
```
Array a = new Array[U1+U2]
int i=0;
for(. ; i=0; i<U1 ; i++)
{ a[i] = A1[i];
}
for(int j=0; j<U2, j++;i++)
{ a[i] = A2[j];
```

* (*) Array sorted by putting higher element of array in starting index.

⊙ STACK:

Last In First out (LIFO)

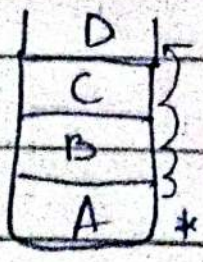


: push (insert data in stack)

: pop (remove data from stack)

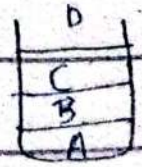
→ Stack is managed through array but any central data can't be taken in.

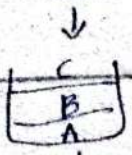
→ stack pointer (points the top of stack).

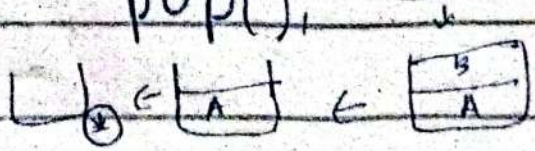


: Push A
Push B
Push C
Push D

pop();
pop();
pop();
pop();





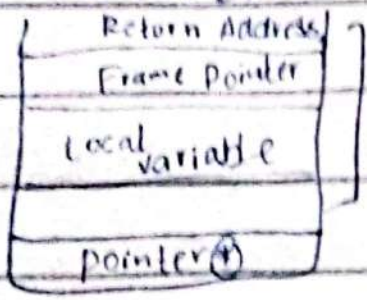


(*) · All operations are constant time operations in Stack ADT : Big O(1).

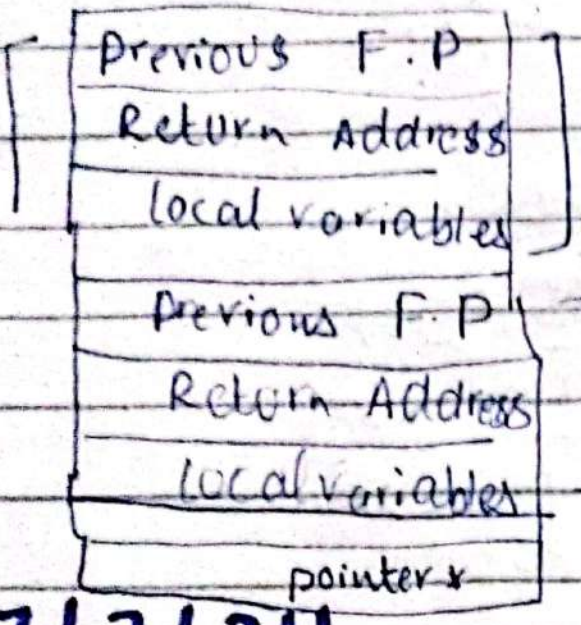
: Ctrl Z (Undo) : Plates

Eg

(*) System Stack:



∴ Recursion calls Frame Pointer increment After call Frame Pointer decrement



→ Activation frame
Stack frame

27/2/24

⊙ STACK:

Push(A);

Pop();

Recursion (calling of function).

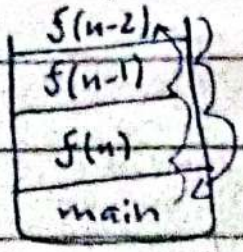
(*) Function recursion:

(calls function by changing condition). (calls itself by changing parameter).

① - Itself
② Changed/different parameter

$2^3 = 2^2 \cdot 2^1$

(*)



(When base calls then it will return to that function called).

(*) Base case are important for writing

in recursive function.

* Stack

```
{ int *stack
  int capacity;
  int top;
public:
  Stack( );
  isEmpty();
  isFull ();
  int Top();
  int Pop();
  void Push (int s);
}
```

: top pointer acts as function pointer

: made it for template class (char stack etc....).

(top most element shows) (return top most and also remove it)

```
-> Stack(int capacity)
{  if(capacity<0)
   { return 0;
     OR
     print capacity >0;
   }
  Stack = new int [capacity];
  top = -1;
}
```

isEmpty Top==-1

isFull (Top!=-1).

```
-> bool isEmpty()
    { return top == -1;
    }
-> bool isFull()
    { return top == capacity;
    }
-> int Top()
    { if (isEmpty)
        { return -1;
        }
      return stack[Top];
    }
-> int Pop()
    { if (isEmpty)
        { return -1;
        }
      else
        { int item = stack[Top];
          Top = Top - 1;
          return item;
        }
```

: ADT set top by itself

: push & pop only handle top.

: insert: top+1 then add.

** : Dry Run + Conceptual Paper

: all operations are constant time operations.

→ void Push(int e)
{ if (is FULL)
{ Print "stack is Full";
}
else
{ top = top+1;
stack[top] = e;
}
}

(*) Dry - Run:

capacity = 4

top = ~~-1 0 1 2 1 0 1 2 3~~

top=-1

→ Push(int) → top=0
Push(int) → top=1
Push(int) → top=2
pop() → top=1
pop() → top=0
push() → top=1
push() → top=2
push() → top=3
pop() → top=2
pop() → top=1
pop() → top=0
pop() → top=-1
pop() → top=-1

return -1 as stack is Empty.

→ Application:

⊛ Evaluating arithmetic expression

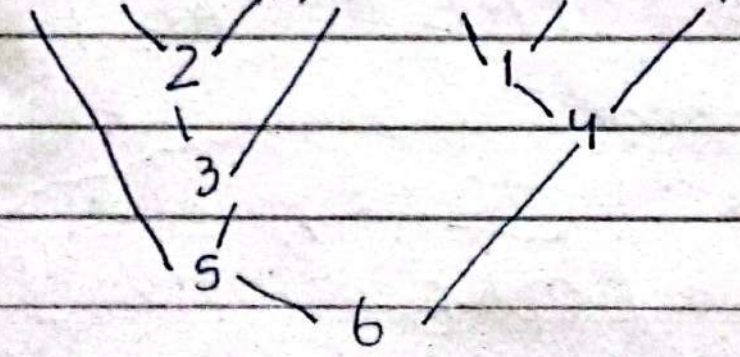
⊛ Recursive / Recursion

⊛ Arithmetic (Mathematical expression)
(BODMAS Rule)

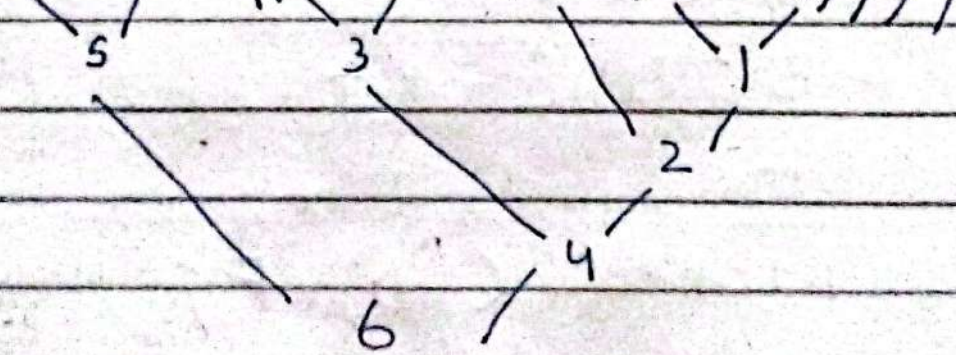
→ A+D-C×D , A*B-C/D

→ Precedence (Exponent, Divide etc...)

→ A+B * C/D - E^F * G



⊛ ((A+B) * ((C/D) - (E^(F*G))))



⊛ Lab Stack (General Application)

E.g: Palindrome etc... (Any)

(101), (DAD)

(MOM)

5/3/24

* Sparse Matrix multiplication. (ADT).

* Precedence (left to right)
  (^ , * , / , + , -).

*

Infix	Prefix	Postfix
A-B	-AB	AB-
A+B	+AB	AB+
A*B	*AB	AB*
A/B	/AB	AB/

*

Postfix

A+B*(C^D-E)

A+B*(CD^E-)

A+B CD^E- *

A B C D^E - * +

Prefix

A+B*(^CD-E)

A+B*(-^CDE)

+A*B-^CDE

: operator stack
: value stack.

* ((A+B)*((C^D)-F)

Postfix

(AB+ *(CD^-F))

(AB+ *CD^F-)

(AB+CD^F-*)

(*) (+AB*(^CD-E))

(+AB*-^CDE) Prefix

*+AB-^CDE.

(*) A/B-C+D*F-A*C

Postfix

A/BC-+DF*-AC*

(x) ABC-/DF*+AC*-

(*) -+-/ABC*DF*AC Prefix

(*) AB-C/+DF*-AC*

(x) ABC/-DF*+AC*- Postfix

ABC-DF*+AC*-

→ Parenthesized :

(((A/B)-C)+(D*F))-(A*C))

(*) Sparse Matrix Multiplication Assignment (Transpose, Multiplication, Addition)
(All functions important for lab).

7/3/24

* Infix Operations:

(1) Input arithmetic operations.

(2) Read char from left to right.

(If oper then display it).

-> If operator precedence is less than stack operator precedence then display.

->Steps:

(1) -> Scan expression from left to right.

(2) -> If scanned character is an operand, then output else if scanned character is operator and if the precedence of scanned character is greater than the operator on the stack or the stack is empty or there is an opening parentheses on top of stack, then pu

that character on the stack.

(3) → Pop all operators from the stack having precedence greater than equals to the precedence of scanned character.

(4) → If the scanned character is opening parentheses, then push it on stack and if the scanned character is closing, pop the character from the stack until opening parentheses present and the discard it.

(5) Output the remaining operators from the stacks if any.

⊙ **Example:**

$$a*(b+c+a)$$

$$a*(bc++d)$$

$$a*\ bc+d+$$

$$abc+d+*$$

(Postfix)

	Stack	O/P
a	#	a
*	# *	a
(	# * (	a
b	# * (	ab
+	# * (+	ab
c	# * (	abc +
+	# * (+	abc +
d	# * (+	abc + d +
)	# *	abc + d + *

-> A+B * C/D - F + A ^ E

Sol:-

A+B * C/D - F + AE^

× [A+B * C/D - FAE^ +

A+B * C/D FAE^ +

A+B * CD/ - F + AE^ : ABC*D/+

F-AE^+

=>

	Stack	O/P
A	#	A
+	# +	A
B	# +	AB
*	# + *	AB
C	# + *	ABC
/	# + /	ABC *
D	# + /	ABC * D / +
-	# +] # -	ABC *
F	# -	
+	# +	
A	# +	
^	# + ^	
E	# + ^	

=> (A+B) * D + E / (F + A * D) + C

Sol:-

(AB+) * D + E / (F + AD*) + C

(AB+) * D + E / (FAD*+) + C

AB+ * D + E / FAD*+ + C

AB+ * D + EFAD*+ / + C

AB + D * + EFAD*+ / C +

ABD * + EFAD *+ / C ++

	stack	o/p
(	#(	
A	#(	A
+	#(+	A
B	#(+	AB
)	#(	AB+
*	#*	AB+
D	#*	AB+D
+		
E		
/		
(		
F		
+		
A		
*		
D		
)		
+		
C		

1+2+3 => 123++

12+3 => 123+

1+23 => 123+

1 — 2 — 3 — ++

12 — 3 — +

1 — 23 — +

⊛ Infix → Prefix:

→ Not equal precedence
(greater precedence)

① → Reverse expression
② → Nearly postfix.
③ → Reverse it, then prefix.

→ (A+B^C)*D+E^5

Sol:-

← 5^E+D*(C^B+A) : Reverse

Postfix (by token). 5E^+D*(CB^+A) : converting postfix

5E^+D*(CB^A+)

5E^+D CB^A+ *

5E^ D CB^A+ * + : Reversing

+*+A^BCD^E5

⊛ Prefix and Postfix Evaluate
(one minor change)

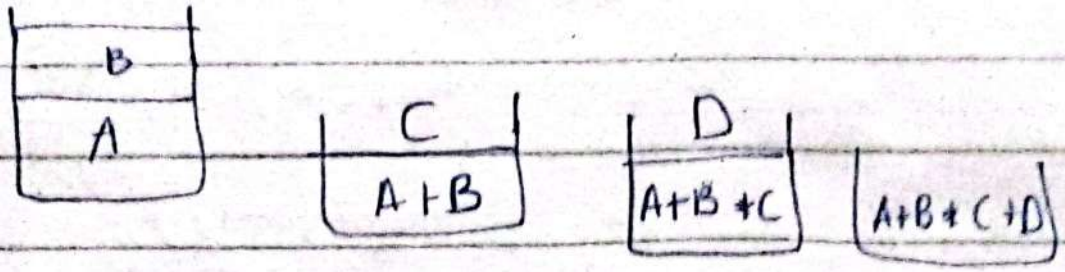
① If postfix (left → right)
prefix (right → left)
② Operand then push to stack.
If operator then apply operator on top most operands and return result at stack.

→ (A+B) * C + D Prefix → +*+ABCD

AB+ * C + D

AB+ C* + D

AB+ C+D +



12/3/24

* Evaluation:

→ Postfix (one operand → second operand).

A B C D E *: Practice

* Recursion:

The ability of the function to call itself.

→ Factorial:

$5! = 5 \times 4 \times 3 \times 2 \times 1$

$n! = n(n-1)(n-2)(n-3)\ldots$

$n! = n(n-1)$

↳ $n(n-1)(n-2)$

↳ $n(n-1)(n-2)(n-3)$

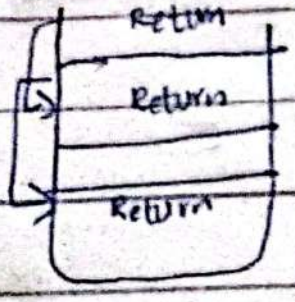
(*) Base Case (Anchor / Ground case):

(*) Inductive step

(*) Factorial Iterating:

```
Fact=1;
for(int i=0; i<N; i++)
    { fact(N(N-1));
    }
```

(*) Base case is used as terminating point.



Backtracking

```
factorial
→ if(n==0 || n==1)
    { return 1
    }
return fact( (n-1)); }
```

Base case

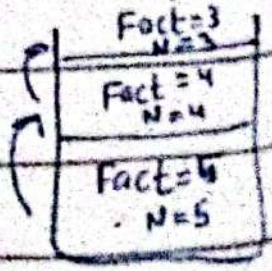
→ Inductive step

(*) Fact(5)

↳ Fact(5(5-1))

↳ Fact(4(4-1))

↳ Fact(3(3-1))



(*) Power function:

⊙ x^n

```
result = 1;
for (int i=0; i<N; i++)
  { result = result * x;
  }
```

⊙ x^4 -> $x(x^3)$

↳ $x(x^2)$

↳ $x(x)$

↳ $x(x^0)$

```
calculate Pow (int N, int pow)
  { if (pow == 1)
      { return N;
      }
    else if (pow == 0)
      { return 1;
      }
```

-> If one function calls other (Direct Recursion)

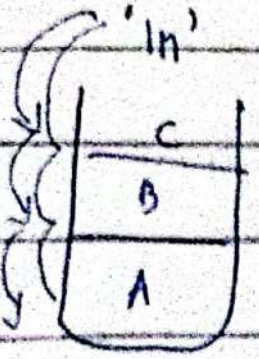
-> If a function calls itself without intervention of some other

function is called Direct recursion.
-> If a function itself with intervention of other function is called indirect recursion.
-> Tail Recursion (Recursive function is the last call)(Recursive call).
(can be converted into iterated solution)
-> Non-Tail Recursion (Recursive call is not last call). (Not be converted easily into Iterative function)(without stack).

E.g:

```
void display()
{ char ch;
  cin >> ch;
  if(ch != '\n')
  { display reverse();
  }
  else
  {cout << ch;
  }
}
```

(Non-Tail Recursion)



(Iterative)

-> Stack<char> Reverse ()
{ stack <char> S;
char ch;
cin>>ch

```
while (cin.get() != '\n')
    { s.push(n);
    }
while (!s.isEmpty())
    {
        cout<< s.pop;
    }
}
```

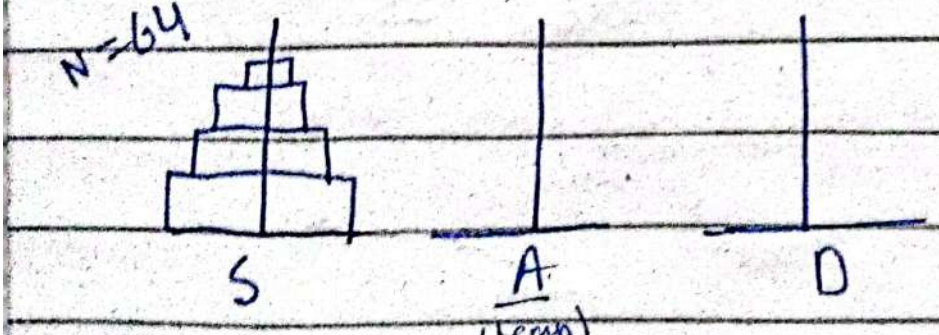
: without stack Iterative process is not possible in Non-tail Recursion.

↑ Next class Quiz (Recursion)

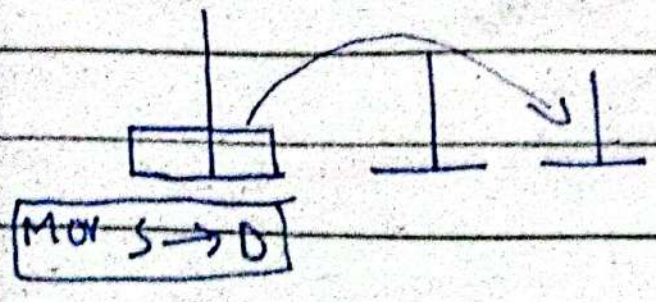
14/3/23

* (*) Assignment (Base Case + Inductive step)
* (↓) Space Complexity Increases of Recursive Function but some problems can't be solved by iteration.

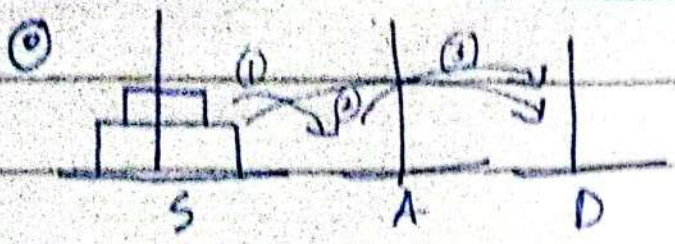
→ Towers of Hanoi:



① Only top most disk can be moved
② Only smaller disk
③ Only one disk can be moved at a time



i) two disks



N → Disk

S → A
S → D
A → D
→ N-1, source To AUX (using Dest) (Helping disk)
1, source to Dest
N-1, AUX to Destination (using Source)

* ⊙ HANOI (N, S, A, D)

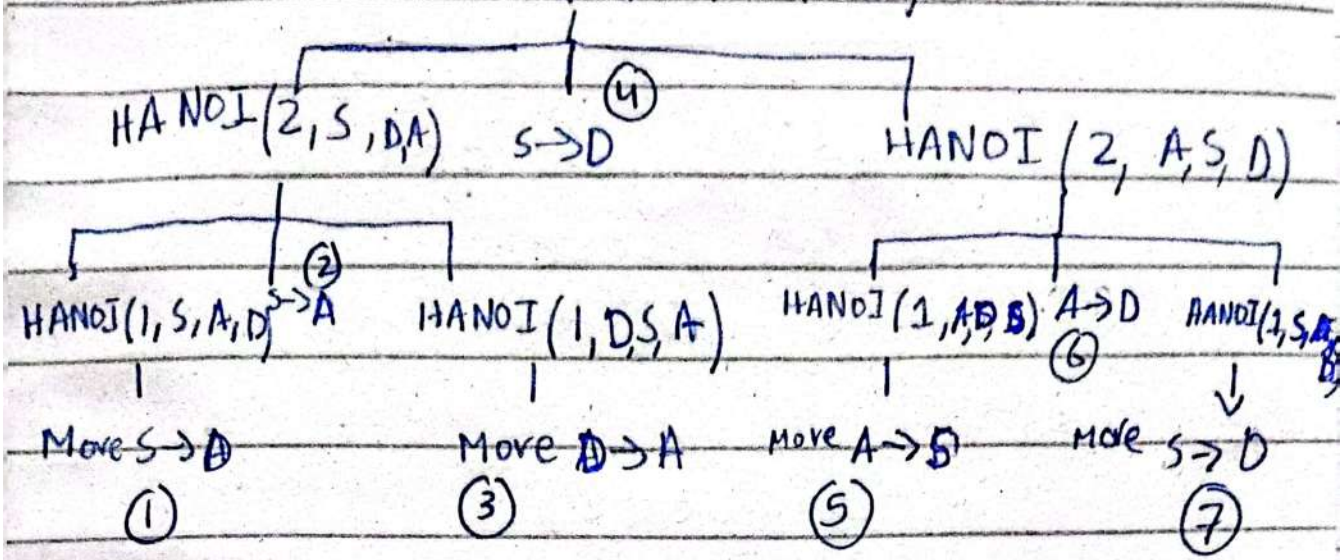
```
{ if (N == 1)   → Base case
  { Move S → D;
  }
  else          → Inductive step
  { HANOI (N-1, S, D, A);
    Move S → D
    HANOI (N-1, A, S, D);
  }
}
```

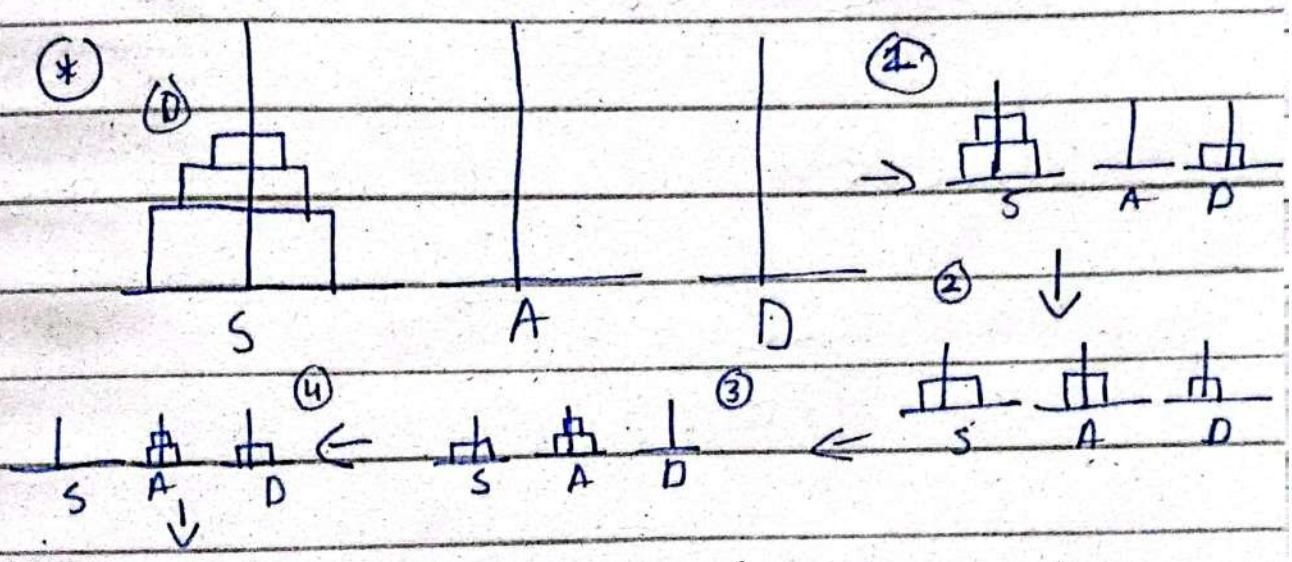
⊙ HANOI (3, S, A, D)

* Assignment
: N=5 (HANOI)
(Dry Run and Tree)

* Iterative Solution

→ HANOI (3, S, A, D)

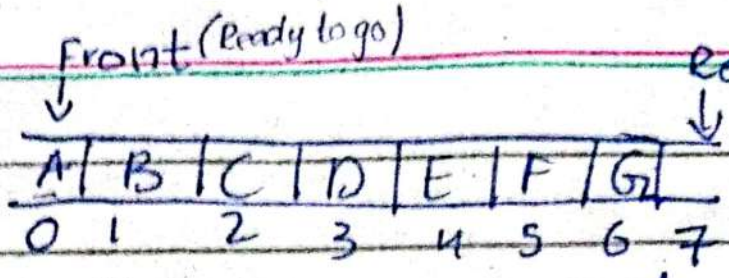




21|3|24

* Linear Data Structure :
  - ↳ Array
  - ↳ Stack → LIFO
  - ↳ Queue → Ordered & Linear (FIFO).

(*) Front (1st element), Rear (where element to be added



(*) Two pointers: front and rear are used in queue.

E.g: (: Print, Car Parking).

(*) ADT

```
{ Enqueue(e);      (Insert)
  Dequeue();       (Removes front)
}
```

:(Front and rear operations are possible not from center or any other)

(*) Queue

```
{ Queue[size];
  Rear = -1;
  Front = -1;

bool isFull()
  { if (Rear == size-1)
      { return true;
      }
    else return false;
  }
bool isEmpty()
  { if (Front == Rear == -1)
      { return true;
  } else return false;
```

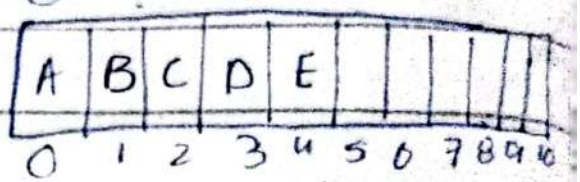
```
void Enqueue (int val)
{   if (isFull())
    { Print "Queue is full";
      return;
    }
    else
    { Rear = Rear+1;                    : Insertion Rear
      Queue[Rear] = val;                : Removal Front
      if( Front == -1)                  : Starting condition.
      { Front=0;
      }
    }

int Dequeue ( )
{   if( isEmpty())
    { Print "Queue is Empty";
      Return;
    }
    else
    { element = Queue[Front];
      (Front = Front+1;)
      If (Front == Rear)
      { Front = -1;
        Rear = -1;
      }
      else
      { Front = Front+1;
      }
    return element;
}
```

F R
↓ ↓
[A]

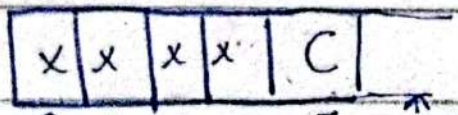
④ Enque(A)
Enque(B)
" (C)
" (D)
" (E)
Deque()
Deque()
Deque()
Deque()
Deque()

F=-1, F=0



F=R

④ Issue:



Enqueue(D)

Error: Full

④ int shifted_dequeue()
{ ⋮

• Front will never go next to rear.

×(else
{Front = Front+1; ×
})×

else
{ for(i=0; i<=Rear; i++)
{ Queue[i] = Queue[i+1];
}
Rear = Rear-1;
} }

:Next class
Circular Enqueue

26|3|24

(*) Paper Mid:

→ ADT's (Array, stack, Queue)

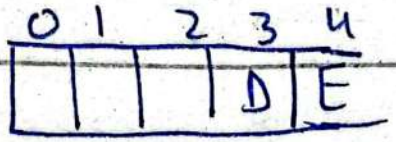
→ 10-15% code + dry Run (Structure)

→ Recursion (Tower of Hanoi)

→ Syllabus (till Queue).

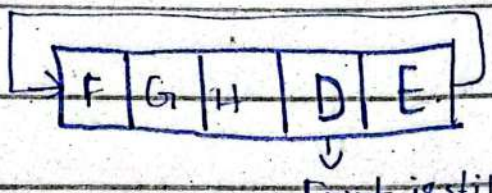
(*) Static DS (Array), Dynamic (Linked list)

(*) **Circular Queue:**



Enqueue(F) x (error) (Full)

So we take circular queue to solve issue.



Enqueue(F), (G), (H)

→ By taking mod we can solve it:

0%5=0, 1%5=1, 2%5=2

3%5=3, 4%5=4, 5%5=0

node = i = (i+1)%5

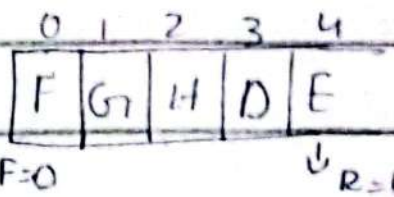
i=0 → 1, i=1 → 2, i=2 → 3 ....

i=5 → 0 ....

```
→bool isEmpty()
{   if(Front==Rear==-1)
    {  Return true;
    }
    Return false;
}
```

-> is Full {Front==0 & Rear == size-1
Rear ← Front}

```
bool isFull()
(F==0 & R==3-1)
{ if (Front == ((Rear+1) % size))
    { Print " Full";
    }
}
```




```
-> void Enqueue (e)
   {
      if (isFull())
         { Print "Queue is Full";
         }
      else
         {
            if (F == -1)
               { F = 0; }
            Reare = (Rear+1) % size;
            Queue [Rear] = e;
   }  }
```

```
-> int Dequeue()
   { if (is Empty)
      { return; (Print)
      else
         e = Queue [Front]
      if (Front == Rear)
            { F == R = -1 }
      }
```

```
else
    Front= (Front +1)%. size;        : change
    }
}
```

→ **Priority Queue** (Printing Higher Authority first and then students).

elements	A	B	
Priorities	P1	P2	P3

* : Included in Exam
** : Kindly Practice
*** : Lab included