Data Structures & Applications

Data & Information
↓ ↓
value meaningful data

Data: Data are simply values or set of values.

Information: Meaningful or processed data.

Data Structures: Logical or mathematical model of a particular organisation of data in the memory. (term)

Implementation

Array → Static Implementation

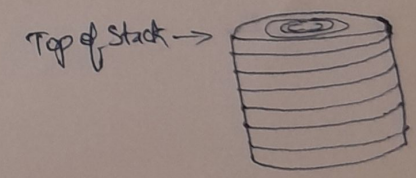
Linked list → Dynamic Implementation

Types:
- Stack, Queue } Linear
- Tree, Graph } Non-Linear

Stack (LIFO)

- A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack.

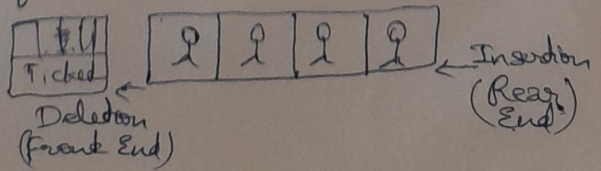
Example: Stack of disk



Queue (FIFO)

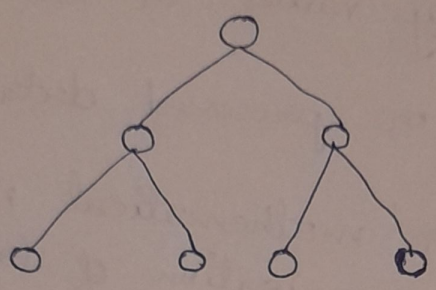
- A queue is a linear list of elements in which deletions can take place only at one end, called the front, and insertions can take place only at the other end, called the rear.

Example- Queue waiting for ticket at the ticket counter



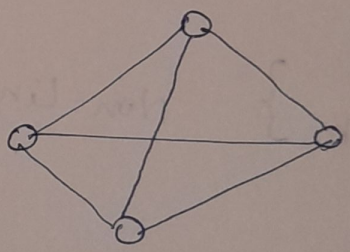
Tree: (Hierarchy)

Tree is a non-linear data structure, used to represent data containing a hierarchial relationship between elements.



Example: records, family tree, table of contents

Graph (Network Type of Structure)



Example: Airline flights

* Each tree is a graph but each graph is not a tree.

ALGORITHM

. Finite set of instructions to accomplish a task.

SPARKS : Smart Programmers Are Required to Know SPARKS

Structured Programming: A Reasonably Komplete Set

statements

cin : Input/Read

cout : Output/write/Display

=: ←

Procedure: Name (parameters list)
Statement

End

Algorithm:

Characteristics

Algorithm must satisfy
- INPUT
- OUTPUT
- DEFINITENESS (clear/unambiguous)
- FINITENESS (Termination after finite no. of steps)

Algorithm + Datastructure = Program

Complexity
- Time (Running Time) (Default)
  - Worst Case
  - Average Case
  - Best Case
- Space (Storage Space)

Time Space Tradeoff:-

The increasing space for storing data processing time may be reduced or vice-versa.

Unless stated complexity means the function which gives running time of the worst case in terms of input size.

Big Oh Notation (Capital O)

Constant Time Algorithm – $O(1)$

Logarithmic Time Algorithm – $O(\log n)$

Linear Time Algorithm – $O(n)$

Polynomial Time Algorithm – $O(n^k)$ for $k>1$

Exponential Time Algorithm – $O(k^n)$ for $k>1$

To study any data structure:-

Basics

Representation of data structure

Implementation (static/Dynamic)

Operations: Insertion + Deletion + Traversal
(Stack/Queue) (Tree/Graph)
- Inorder
- Preorder
- Postorder

(Tree/Graph)
- BFS
- DFS

Applications

Playing Cards: Insertion Sort
BFS: Queue
DFS: Stack

Time Space Tradeoff:

The increasing space for storing data processing time may be reduced or vice-versa.

Unless stated complexity means the function which gives running time of the worst case in terms of input size.

Big Oh Notation (Capital O)

Constant Time Algorithm - $O(1)$

Logarithmic Time Algorithm - $O(\log n)$

Linear Time Algorithm - $O(n)$

Polynomial Time Algorithm - $O(n^k)$ for $k>1$

Exponential Time Algorithm - $O(k^n)$ for $k>1$

To study any data structure:

Basics

Representation of data structure

Implementation ~~static~~ (static/Dynamic)

Operations: Insertion + Deletion + Traversal
(Stack/Queue) (Tree/Graph)
- Inorder
- Preorder
- Postorder
- BFS
- DFS

Applications

Playing Cards: Insertion Sort
BFS: Queue.
DFS: Stack

Array
- Homogeneous Element
- Continuous memory allocation / arranged in contiguous way

A[ ]

Value →	10	20	30	40	50
Index →	0	1	2	3	4

A[0] = 10

A[1] = 20

'A'	'B'	'10'	'C'	'D'	10

LB UB

printf("%u", A);
↳ Base Address

%d → decimal

Address Calculation:-

1D:

LOC(A[K]) = Base Address + W(K - LB)

W = size
K = Kth
LB : Lower Bound

2D:

2D Array
- Row Major
- Column Major

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}_{2\times2}$$

Row Major: 1, 2, 3, 4

Column Major: 1, 3, 2, 4

Applications:

- To list some elements of a record
- Matrices in Mathematics
- Tables in Business Applications (MATLAB)
- String Manipulation

Sparse Matrix:

Matrices with a relatively high proportion of zero entries are called sparse matrix

There are two types of n^2 sparse matrix:

1) Triangular
   - Lower
   - Upper
2) Tridiagonal

Lower Triangle

$$\begin{bmatrix} * & & & \\ * & * & & \\ * & * & * & \\ * & * & * & * \end{bmatrix}$$

Upper Triangle

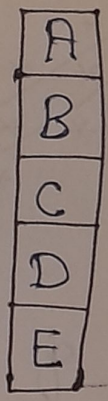
$$\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & & * \end{bmatrix}$$

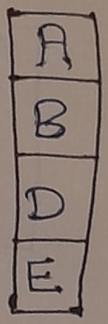
* → non-zero elements

Deletion from Array

Delete (A, N, k, ITEM)

1. ITEM ← A[k]
2. Repeat for i ← k to N-1
3. A[i] ← A[i+1]
4. N ← N-1
5. End





Traversal

Traverse (A, LB, UB, k)

1. Repeat for k = LB to UB
2. Apply process to A[k]
3. End

Array Representation of Polynomial

$1 + x + 2x^2 + 3x^3 + 4x^4 + 5x^5$

Coefficient	1	1	2	3	4	5
Exponent	0	1	2	3	4	5

Linked List:

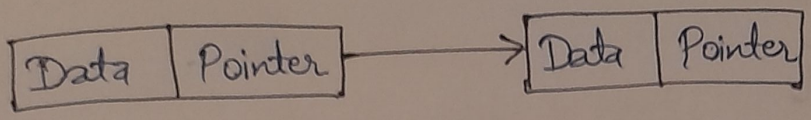
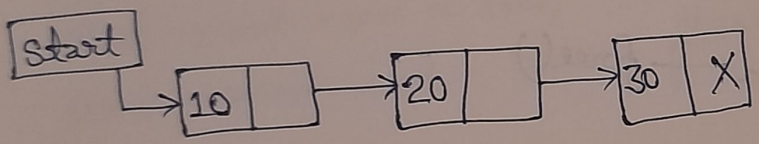


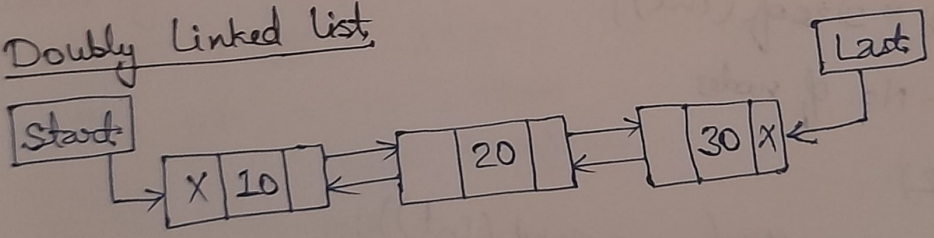
Fig: A Linked List

Types:
- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Circular Doubly Linked List

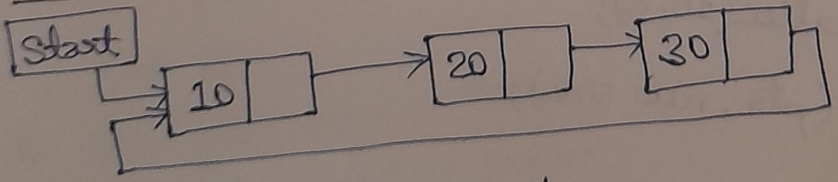
1) Singly Linked List:



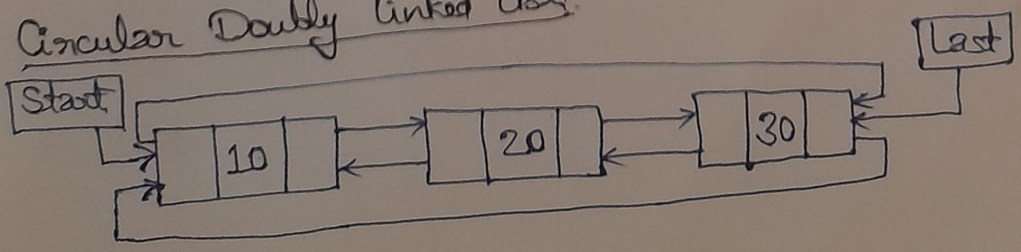
2) Doubly Linked List:



3) Circular Linked List:



4) Circular Doubly Linked List:



Declaration of linked list:

```
struct node {
        int a;
        struct node * next;
    };
typedef struct node NODE;
NODE *start;
```

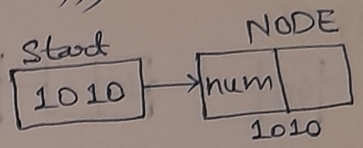
Memory Allocation ← malloc()
                  ← calloc()
                  ← realloc()

Memory deallocation — free()

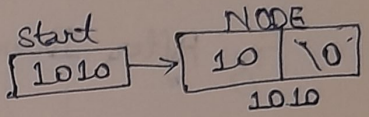
```
malloc (10 * sizeof(int))
        ↑ No. of nodes
calloc (10, 2)
ptr = (int *) malloc(10 * sizeof(int));
      (typecast)
ptr = (int *) calloc(10, 2);
ptr1 = realloc(ptr, new size);
free(ptr);
```

Creation of Linked List:-

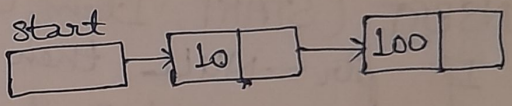
```
struct node {
        int num;
        struct node * next;
    };
typedef struct node NODE;
NODE *start;
start = (NODE *) malloc (sizeof(NODE));
```




```
start -> num = 10;
start -> next = '\0';
```




```
start -> next = (NODE*) malloc(sizeof(NODE));
start -> next.num = 100;
start -> next.next = '\0';
```



Inserting a node at the beginning:-

```
INSERT (START, ITEM)
1 If ptr == NULL then
        print Overflow
        Exit
  Else
        ptr = (NODE*)malloc(sizeof(NODE))
  End if
2. set ptr -> num = ITEM
3. set ptr -> next = START
4. set START = ptr
5. End
```

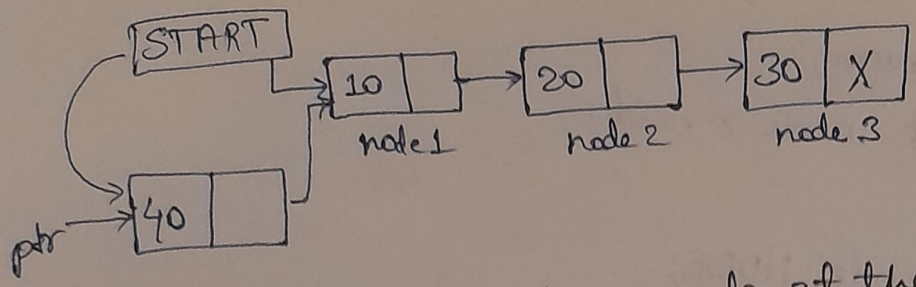


Fig. Insertion of new node at the beginning

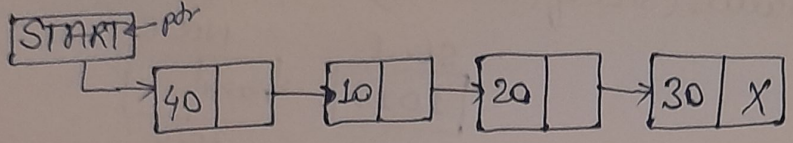


Fig After Insertion

2) Inserting a node at the end

```
INSERT (START, Item, LOC)
1. If ptr == NULL then
        print Overflow
        Exit
   Else
        ptr = (NODE*) malloc(sizeof(NODE))
   End if
2. set ptr->num = Item
3. set ptr->next = NULL
4. set LOC->next = ptr
5. End
```

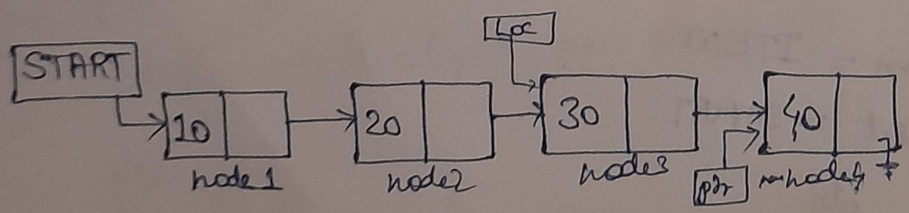


Fig. Insertion of new node at the end

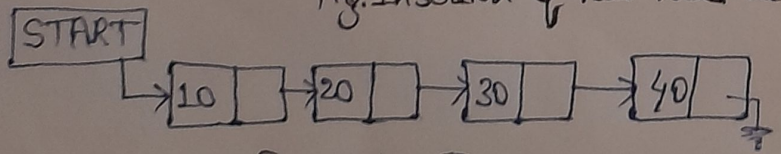


Fig. After Insertion

3) Inserting a node at the specified position.

INSERT (START, Item, temp)

1. If ptr == NULL then
   print Overflow
   Exit
   Else ptr = (NODE *) malloc (sizeof (NODE))
   End if
2. set ptr → num = Item
3. set ptr → next = temp → next
4. set temp → next = ptr
5. End

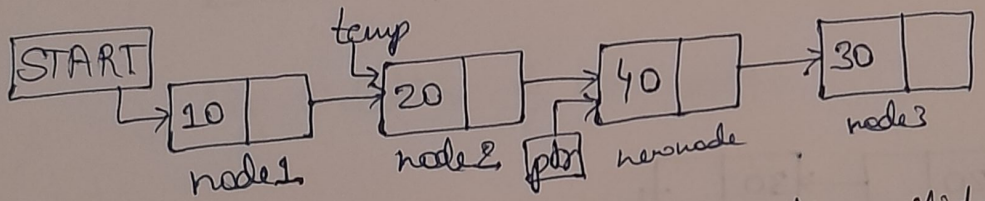


Fig. Insertion of new node at specified position.

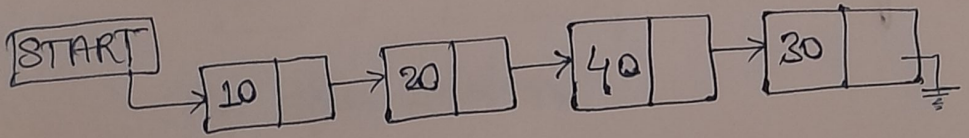


Fig. After Insertion.

Deletion

1) Deleting the Ist node.

DELETE (START)

1. If START == NULL then
   print Underflow
   Exit
   End if
2. START = ptr → next
3. free(ptr)
4. End

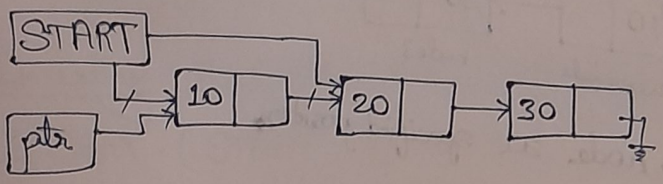


Fig. Deletion of Ist node

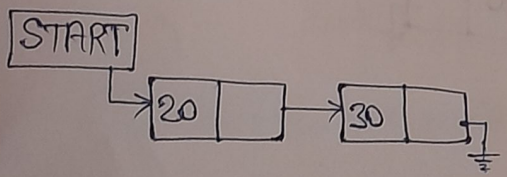


Fig. After Deletion

2) Deleting the Last node.

DELETE (START, LOC)

1. If START == NULL then
   print Underflow
   Exit
   End If
2. set LOC → next = NULL
3. free (ptr)
4. End

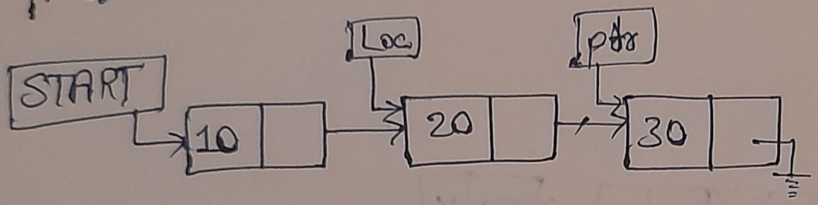


Fig Deletion of Last node.

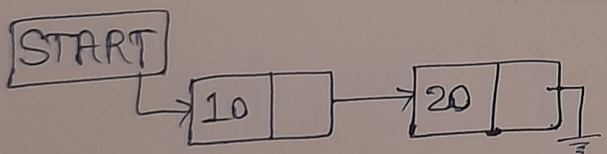


Fig. After Deletion

3) Deleting the node from the specified position:

DELETE (START, temp)

1. If START == NULL then
   print Underflow
   Exit
   End if
2. temp = ptr
3. ptr = ptr → next
4. free (temp)
5. End

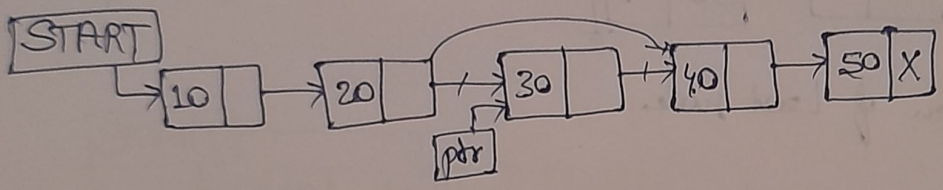


Fig. Deletion of node at specified position

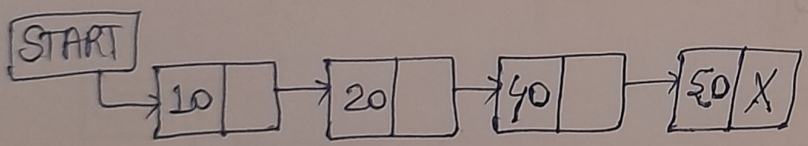


Fig. After Deletion

OR

DELETE (START, temp)

1. If START == NULL then
   print Underflow
   Exit

   End if
2. temp → next = ptr → next
3. free(ptr)
4. End

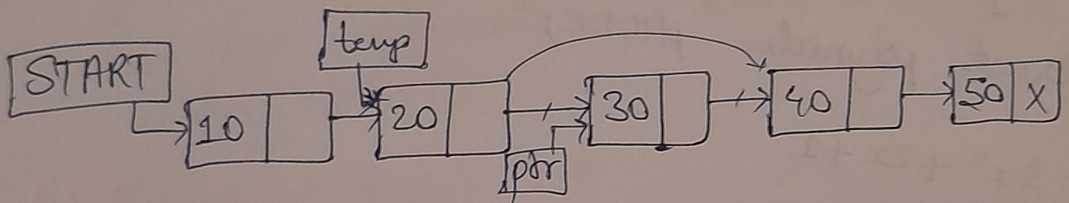


Fig. Deletion of node at specified position

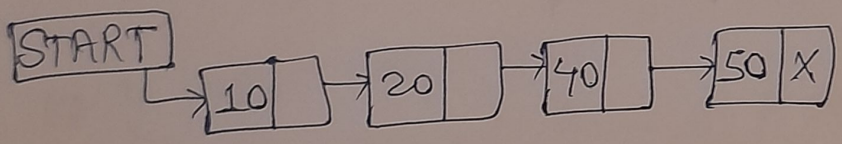
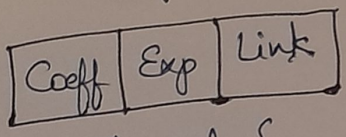


Fig After Deletion

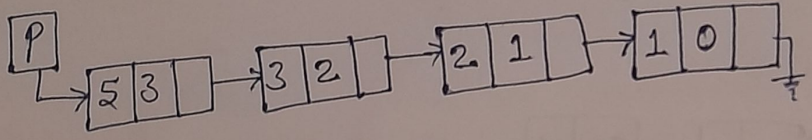
Application of Linked List

1) Polynomial Representation




```
struct polynode {
        int coeff;
        int exp;
        struct polynode *ptr;
    }
typedef struct polynode pNODE;
```

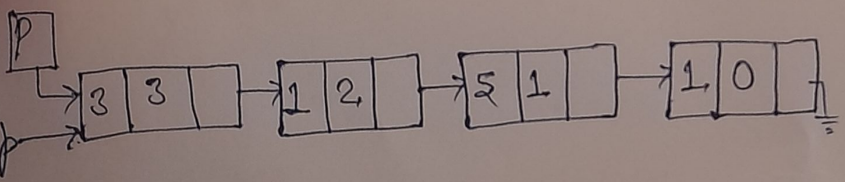
$p = 5x^3 + 3x^2 + 2x + 1$

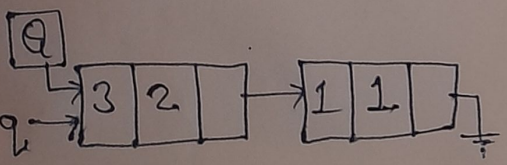


2) Polynomial Addition

Let $p = 3x^3 + x^2 + 5x + 1$

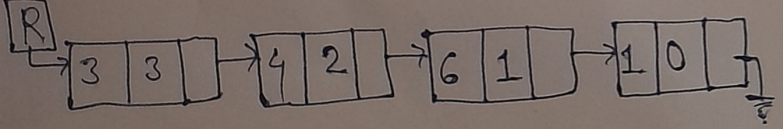
and $Q = 3x^2 + x$





Resultant will be:

$R = 3x^3 + 4x^2 + 6x + 1$

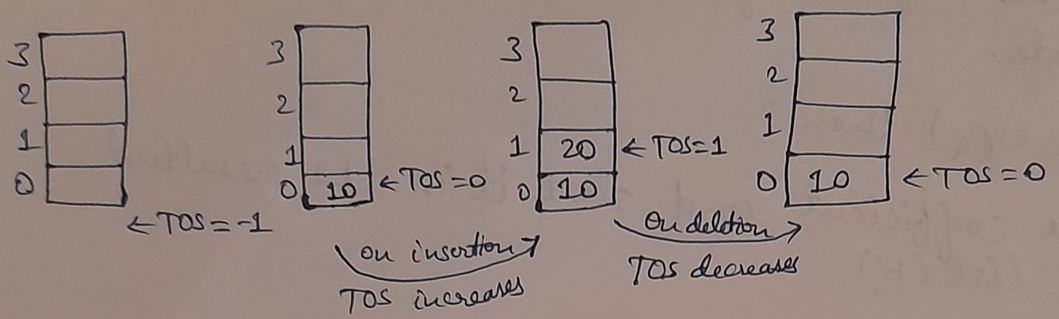


Algorithm:

1. READ the no. of terms, coefficient, exponents of Ist polynomial.
2. READ the no. of terms, coefficient, exponents of IInd poly-nomial.
3. Set the temp positions p & q to traverse the polynomial.
4. Compare the exponent of both polynomial starting from first node.
   (a) If ($exp_p = exp_Q$) then
   Add the coefficient and store it in the resultant linked list (R)
   (b) If ($exp_p < exp_Q$) then
   Add the current term of Q to the R and move q to point next node in Q.
   (c) If ($exp_p > exp_Q$) then
   Add the current term of P to the R and move p to point next node in P.
5. Append the remaining node of either of the polynomial to R.

STACK

- LIFO
- STACK Top increases during insertion and decreases during deletion.



Implementation
- Static
- Dynamic

PUSH (Insertion)

PUSH (Stack[n], Item) // n = size of stack

1. If Top = n-1
   then print 'Overflow'
   Exit
2. set Top ← Top+1
3. set Stack[Top] ← Item
4. End

POP (Deletion)

```
POP (Stack[n], Item)          //n= size of stack
1. If Top < 0
       then print 'Underflow'
2. Else set Item ← Stack[Top]
       set Top ← Top-1
   Return the deleted Item
3. End
```

Application of Stack

1) Conversion of infix to prefix expression.
2) Conversion of infix to postfix expression
3) Evaluation of postfix expression.
4) Recursion.

Infix to Prefix

1) $(A * B) + C$

   $*AB + C$

   $\Rightarrow + *ABC$

2) $A / (B \wedge C) + D$

   $A / \wedge BC + D$

   $/A \wedge BC + D$

   $\Rightarrow + / A \wedge BCD$

(3) $(A - B/C) * (D * E - F)$

$(A - /BC) * (*DE - F)$

$-A/BC * -*DEF$

$\Rightarrow * - A/BC - *DEF$

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

$(A * B + /CD) - F$

$(*AB + /CD) - F$

$+ *AB/CD - F$

$\Rightarrow - + *AB/CDF$

(5) A / B ^ C - D

A / ^BC - D

/ A ^ BC - D

⇒ - / A ^ BCD

(6) A + B * C

A + *BC

⇒ + A*BC

Infix to Postfix

(1) A + B * C

A + BC*

⇒ ABC*+

(2) A + [(B+C)+(D+E)*F] / G

A + [BC+ + DE+ * F] / G

A + [BC+ + DE+F*] / G

A + BC+DE+F*+ / G

A + BC+DE+F*+G/

⇒ ABC+DE+F*+G/+

(3) A + B - C

AB+ - C

⇒ AB+C-

(4) A * B + C / D

A * B + CD/

AB* + CD/

⇒ AB*CD/+

(5) A * B + C

AB* + C

⇒ AB* C+

(6) A + B / C - D

A + BC/ - D

ABC/ + - D

⇒ ABC/+D-

(7) (A+B) / (C - D)

AB+ / CD-

⇒ AB+CD-/

(8) (A+B) * C/D

AB+ * C/D

AB+ * CD/

⇒ AB+CD/*

(9) (A+B) * C/D + E^F/G

AB+ * C/D + EF^/G

AB+ * CD/ + EF^/G

AB+ * CD/ + EF^G/

AB+CD/* + EF^G/

⇒ AB+CD/*EF^G/+

(10) A + (B*C - (D/E^F) * G) * H

A + (B*C - (D/EF^) * G) * H

A + (B*C - DEF^/ * G) * H

A + (B*C - DEF^/G*) * H

A + (BC* - DEF^/G*) * H

A + BC*DEF^/G* - * H

A + BC*DEF^/G* - H*

⇒ ABC*DEF^/G* - H* +

(11) A - B / (C * D ^ E)

A - B / (C * DE^)

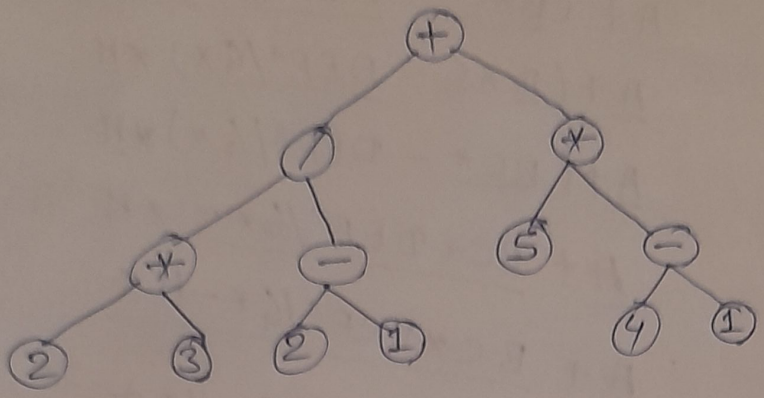
A - B / CDE^*

A - BCDE^*/

⇒ ABCDE^*/-

Binary Expression Tree

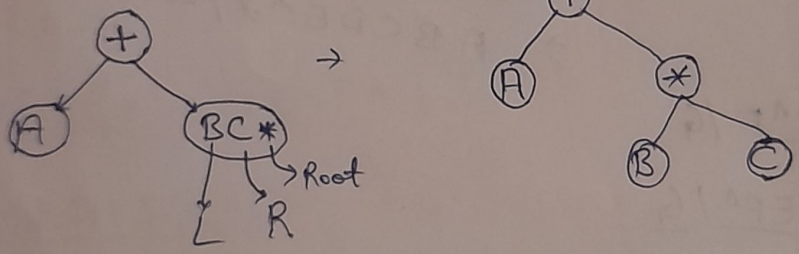
2 * 3 / (2 - 1) + 5 * (4 - 1)



A + B * C - Infix

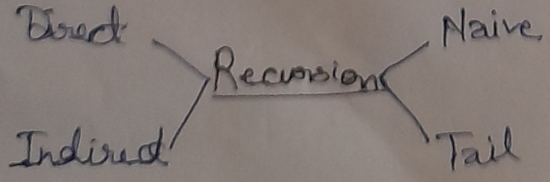
A BC * + Postfix

L R Root



Recursion (calling function within itself)

```
fun() {
    fun();
}
```



```
fun(){
    ------
    ------
    fun();
}
```

Tail

```
fun() {
    fun();
    --------
    --------
}
```

Naive

```
int abc(){
    ------
    ------
    abc();
}
```

Direct

```
int abc(){
    xyz();
}
int xyz(){
    abc();
}
```

Indirect

Recursive definition

1) Factorial

$$f(n) = \begin{cases} 1 & \text{If } n = 0|1 \\ n * f(n-1) & \text{Else} \end{cases}$$

$3 * f(2)$
$2 * f(1)$
1
Backtracking

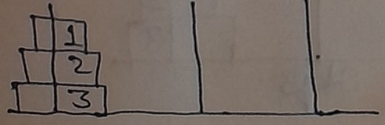
2) Fibonacci Series

$$fib(n) = \begin{cases} 0 & \text{If } n = 0|1 \\ 1 & \text{If } n = 2 \\ fib(n-1) + fib(n-2) & \text{If } n > 2 \end{cases}$$

3) Tower of Hanoi Problem

[Hanoi: capital city of Vietnam]

X Z Y



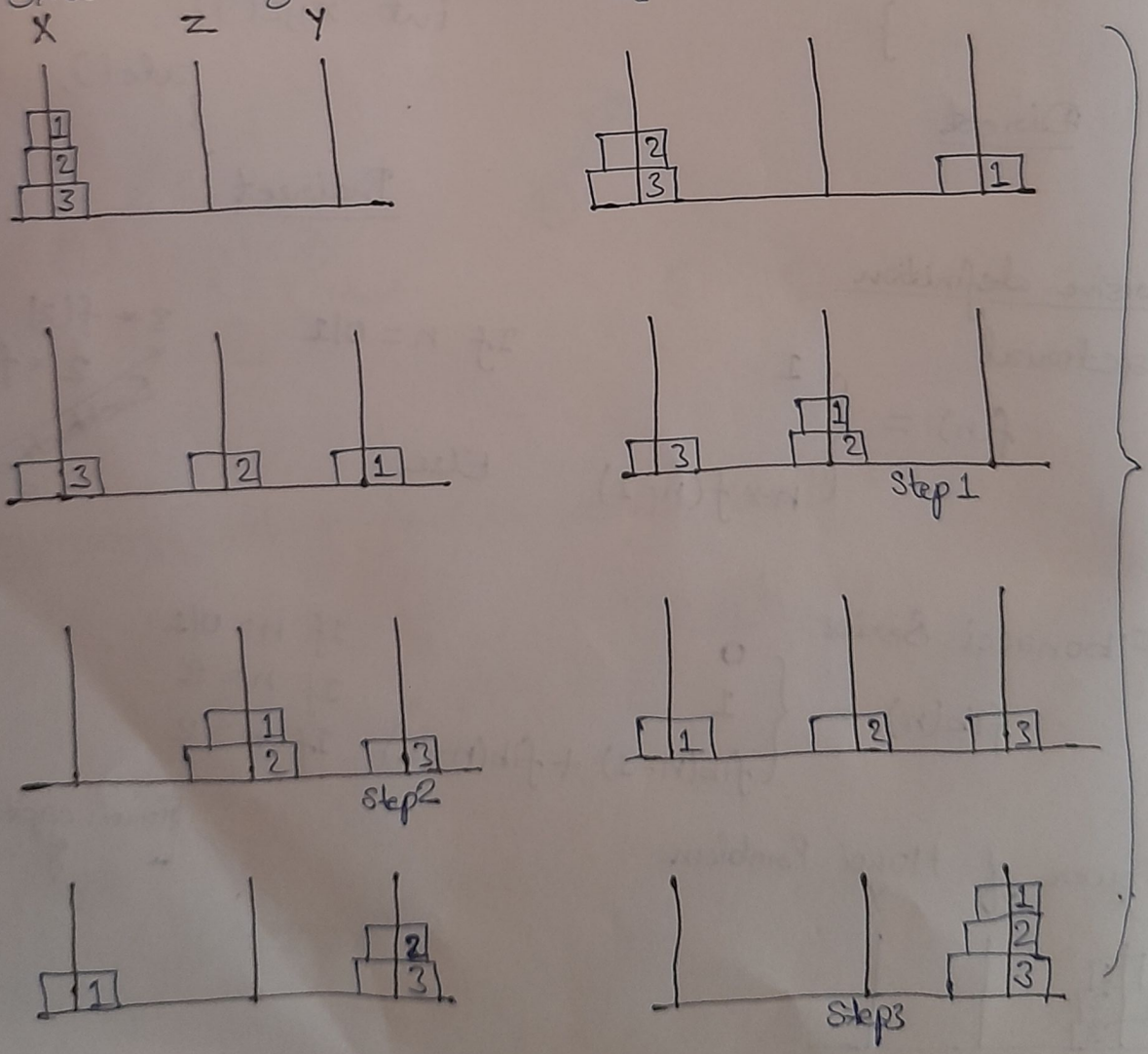
Rules of the game

1) Transferring the disk from the source peg X to the destination peg Y such that at any point of transfer no longer disk is placed on smaller ones.

2) Only one disk may be moved at a time

3) Each disk must be stacked on any one of the pegs.

4) In the beginning, the disks are stacked on peg X ie the largest sized disk on bottom and smallest size disk on top.



Recursive:-

It can be generalised for 'n' no. of disks as per following recursive definition:-

(1) Move the top (n-1) disks from the X peg to the Z peg

(2) Move the n^{th} disk to Y peg

(3) Move the (n-1) disks stacked on the Z peg to the Y peg

Advantages of Recursion

- Reduces lines of code

Disadvantages of Recursion:-

- Increases time complexity
- Goes in infinite if not terminated correctly
- It consumes more storage space
- Not suggested when problem can be solved through iteration.

C * B + A $\xrightarrow{\text{Postfix}}$ C B * A +

↓

(C * B + A)

Symbol Scanned	Stack	Postfix Expression
(	(	
C	(	C
*	(*	C
B	(*	C B
+	(+	C B *
A	(+	C B * A
)	–	C B * A +

(A + B * C) $\xrightarrow{\text{Prefix}}$ + A * B C

↓ Reverse

C * B + A $\xrightarrow{\text{Postfix}}$ C B * A +

↓ Reverse

+ A * B C

Procedure Postfix (Q, P)

Arithmetic Expression in Infix

Equivalent Postfix Expression

1) Scan Q from Left to Right and repeat step (2) to (5) for each element of Q until the stack is empty.

2) If an operand is encountered then add it to P.

3) If a Left parenthesis "(" is encountered then push it on the stack.

4) If an operator ⊗ is encountered then
   (a) Add operator ⊗ to stack.
   (b) Repeatedly pop from stack and add to P each operator which has a same or higher precedence than operator ⊗.

5) If a Right parenthesis ")" is encountered then
   (a) Repeatedly pop from stack and add to P each operator until a Left parenthesis "(" is encountered.
   (b) Remove the Left parenthesis "(", do not add to P.

6) End

Converting Infix to Prefix

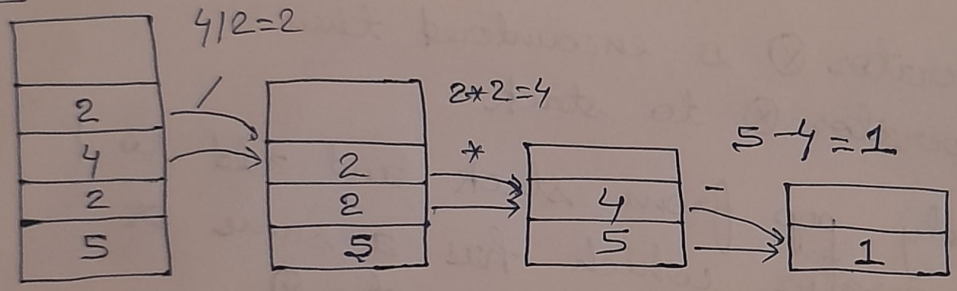
1) First, reverse the given expression
2) Convert it into postfix expression
3) Reverse the result to get prefix expression
4) End

Evaluation of postfix expression using a stack:-

Infix :- 5 - 2 * 4 / 2 = 1

Postfix :- 5 2 4 2 / * -

B * A



Procedure Evaluation (P, A, B)

P → Postfix

1) Scan P from left to Right and repeat step (2) & (3) for each element of P until the right parenthesis ")" is encountered (end)
2) If an operand is encountered put it on the stack
3) If an operator ⊗ is encountered then
   (a) Remove the top elements of stack where A is the top element and B is the next to top element.
   (b) Evaluate B ⊗ A.