INTRODUCTION TO DATA STRUCTURES

PAGE NO.: 1
07 | 7 | 22

1. INTRODUCTION.

Basic terminologies:

'Data' refers to values / set of values
'Data item' refers to single unit of values.
Data items that can be divided into subitems are called "group items" while those that are not are called "elementary items"

Ex: Student name can be divided into first / last name
Student roll no cannot be divided.

Entity is something that has certain attributes or properties that can be assigned values.

Ex: Attributes:	Name.	Age.	Sex	Aadhar No.
Values	Aditya	30	M	xxxx 1234 xxxx

Entities with similar attributes for an "entity set"
Ex: Employees in an organization, students in college.

A field is a single unit of information representing an attribute of an entity.

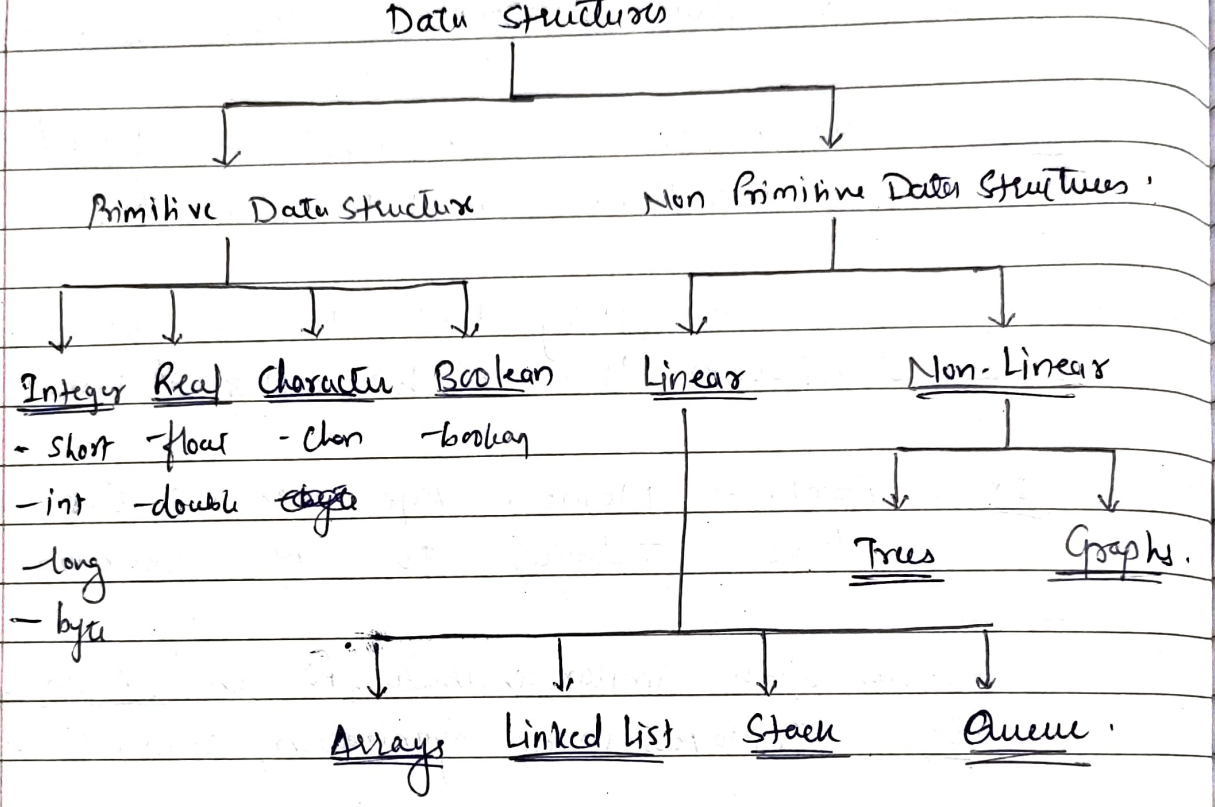
A record is a collection of field values of a given entity.

A file is the collection of records of the entities in entity set.

* Data Structures

The logical or mathematical model of a particular organization of data is called data structure.

Classification of Data Structures



A DS is said to be linear if its elements form a sequence or a linear list.
Here, data is arranged in a linear fashion although the way they are stored in memory need not be sequential. Ex: Array, Stack, Queue

A DS is non linear if data is not arranged in sequence.
Ex: Trees, Graphs

Arrays

Simplest type of DS is 1-D Array. List of finite no. 'n' of similar data elements.

If A is array, then its elements are denoted by -

$a_1, a_2, a_3, \ldots, a_n$

A(1), A(2), A(3), ..., A(n)

A[1], A[2], A[3], --, A[n].

Number 'k' in A[k] is called subscript and A[k] is called subscripted variable.

Linked List

Ex :- A brokerage firm maintains a file where each record contains a customer's name and his/her salesperson.

Customer	Salesperson
Adams	Smith
Brown	Ray
Clark	Jones.
Drew	Ray
Evans	Smith
Farmer	Jones
Geller	Ray
Hill	Smith
Infeld	Ray.

Fig 1

	Customer	Pointer
1	A	3
2	B	2
3	C	1
4	D	2
5	E	3
6	F	1
7	G	2
8	H	3
9	I	2

	Salesperson
1	Jones
2	Ray
3	Smith

Fig 2

Suppose, we want the list of customers for a given salesperson.
Fig 2 would search through entire customer file.

	Salesperson	Pointer
1	Jones	3, 6
2	Ray	2, 4, 7, 9
3	Smith	1, 5, 8.

Fig. 3.

Another very popular way of storing the data would be -

	Customer	Link
1	A	5
2	B	4
3	C	6
4	D	7
5	E	8
6	F	0
7	G	9
8	H	0
9	I	0

Salesperson	Pointer	
Jones	3	1
Ray	2	2
Smith	1	3

Fig 4.

Using this representation, one can easily ~~represent~~ obtain entire list of customer for a given salesperson.

Pointer :- When element in one list points to element in different list.

Link :- When an element in a list points to an element in that same list.

* **Stack :-**

Also called as LIFO system, is a linear list where insertions and deletions can take place only at one end, called the "top".

Ex: Stack of dishes. Dishes are inserted only at top and can be deleted only from the top of the stack.

* **Queue :-**

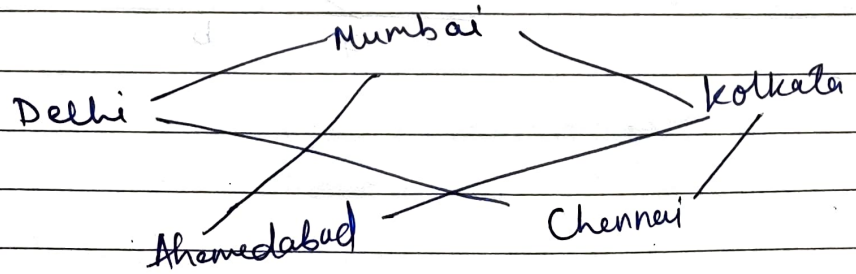
Also called as FIFO system, is a linear list where deletions can take place only at one end of the list, called the "front" of the list. Insertions can take place only at the other end of the list called the "rear" of the list.

Ex :- Line of people waiting at a bus stop.
Cars waiting at signal.

* **Graphs :-**

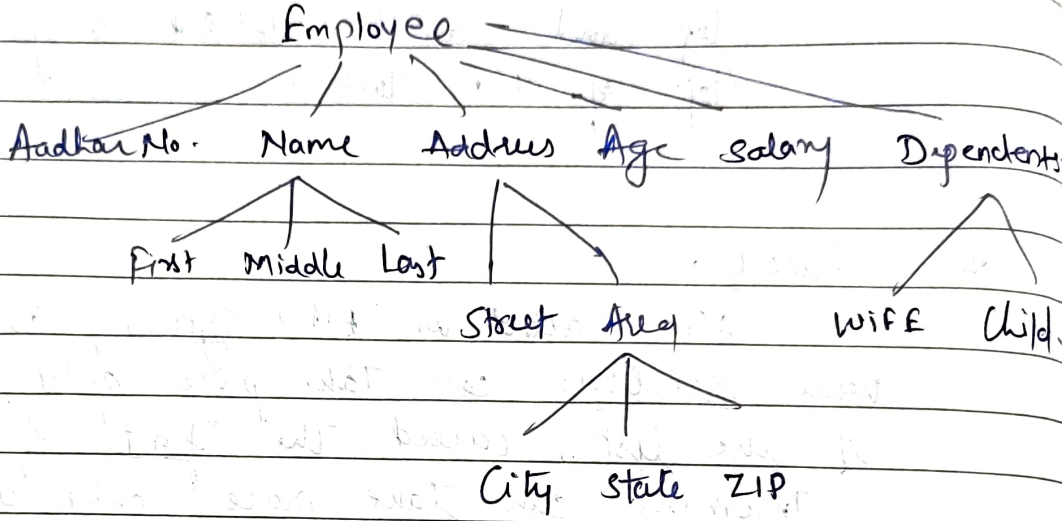
Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. This type of relationship is called a graph.

Ex :- Airlines flying between various cities.



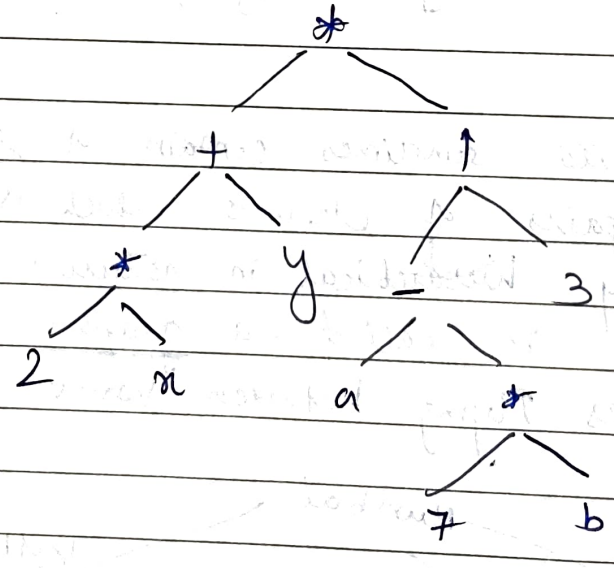
Trees

Data frequently contains a hierarchical relationship between various elements. D.S. that reflects this relationship is called a rooted tree graph, or simply a tree.



or

$(2x+y)(a-7b)^3$



* Data structures Operations

Data appearing in our D.S. are processed by means of certain operations.

① **Traversing** : Accessing each record exactly once. (Also called as "visiting" the record)

② **Searching** : Find location of the record with a given key value.

③ **Insertion** : Adding a new record to D.S.

④ **Deleting** : Removing a record from D.S.

Multiple operations may be performed.
Ex:. Deleting may require searching first.

Two operations for special situation.

① **Sorting** : Arranging records in some logical order.

② **Merging** : Combining the records in two different sorted files into a single sorted file.

Mathematical Notations and Functions

⇒ Floor and Ceiling Functions

x = real number.

x lies between two integers called the floor and the ceiling of x

$\lfloor x \rfloor$ = floor of x = greatest integer that does not exceed x

$\lceil x \rceil$ = ceiling of x = least integer that ~~does not~~ is not less than x.

If x = integer then $\lfloor x \rfloor = \lceil x \rceil$

otherwise $\lfloor x \rfloor + 1 = \lceil x \rceil$.

Examples :-

$\lfloor 3.14 \rfloor = 3$ $\quad \lfloor \sqrt{5} \rfloor = 2$

$\lfloor -8.5 \rfloor = -9$ $\quad \lfloor 7 \rfloor = 7 = \lceil 7 \rceil$.

$\lceil 3.14 \rceil = 4$ $\quad \lceil \sqrt{5} \rceil = 3$ $\quad \lceil -8.5 \rceil = -8$

⇒ Remainder function (Modular Arithmatic)

Let 'k' be any integer and M be the integer then -

$k \pmod M$

denotes integer remainder when 'k' is divided by M.

$k \pmod M$ is unique integer 'r' such that

$k = Mq + r$ where $0 \le r \le M$.

Examples :-

When k is positive. -

$25 \pmod 7 = 4$ $25 \pmod 5 = 0$ $35 \pmod{11} = 2$

$3 \pmod 8 = 3$.

When 'k' is negative - divid |k| by modulus to obtain r'. Then $k \equiv -r' \pmod M$.

Hence, $k \pmod M = M - r'$ when $r' \neq 0$

$= M - (k \bmod M)$.

$-26 = -5 \bmod 7$

$-26 \pmod 7 = 7 - 5 = 2$ / $-371 \pmod 8 = 8 - 3 = 5$

-26, closest no less than -26 and divisible by 7 is -28 so Ans $= -26 - (-28) = 2$

Remainder theorem

$-6 \bmod 5 = 2$

$-6 = 5 \times q + r$ Range of $r = 0$ to 4.

Let $q = -1$

$-6 = 5 \times (-1) + r$

$r = -1$ This is wrong as $r = 0$ to 4

Let $q = -2$

$-6 = 5 \times (-2) + r$.

$r = 4$ This is correct.

$-2345 \pmod 6 = 6 - 5 = 1$ $-39 \pmod 3 = 0$.

'mod' also used for mathematical congruence.

$a \equiv b \pmod M$ iff M divides $b - a$.

$0 \equiv M \pmod M$ and $a \pm M \equiv a \pmod M$.

Arithmatic modulo 'M' refers to arithmatic operations of Add, Subs, Multi. where arithmatic value is replaced by its equivalent value in set.

$\{0, 1, 2, \ldots, M-1\}$.

or

$\{1, 2, 3, \ldots, M\}$.

Example:

Arithmatic Modulo 12 –

$6+9 \equiv 3$, $7 \times 5 \equiv 11$, $1-5 \equiv 8$, $2+10 \equiv 0 \equiv 12$

⇒ Integer and Absolute value function

Let x = Real no, Integer value of x

INT(x) convert to integer by truncating fractional part.

INT(3.14) = 3 INT($\sqrt{5}$) = 2 INT(−5.5) = −5.

INT(7) = 7.

INT(x) = $\lfloor x \rfloor$ or $\lceil x \rceil$ according to whether x is +ve or −ve.

Absolute value of 'x' written as ABS(x) or $|x|$ is defined as greatest of x or $-x$.

$x = 0$ ABS(0) = 0

$x \neq 0$ ABS(x) = x or ABS(x) = −x. depending on whether x is +ve or −ve.

$|-15| = 15$ $|-3.33| = 3.33$ $|4.4| = 4.4$ $|-0.75| = 0.75$.

⇒ Summation Symbol ; Sums.

$a_1 + a_2 + \cdots + a_n$ and $a_m + a_{m+1} + \cdots + a_n$.

denoted by

$\sum_{k}$ $\sum_{j=1}^{n} a_j$ and $\sum_{j=m}^{n} a_j$

Σ = Greek letter Sigma.

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$1 + 2 + 3 + \cdots + 50 = \frac{50(51)}{2} = 1275$$

→ Factorial:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2)(n-1)n$$

$$0! = 1$$

Proof :- $n! = (n)(n-1)!$

for $n = 1$

$$1! = 1 \cdot (1-1)!$$

$$1 = 1 \cdot 0!$$

$$\boxed{0! = 1}$$

→ Permutations :-

for n elements $n!$ permutations are possible

$n = 3$ $\quad 3! = 6$.

a, b, c $\quad$ abc, acb.

bac, bca

cab, cba

⇒ Exponent and Logarithms

$a^m = a \cdot a \cdot a \cdots$ (m times) $\quad a^0 = 1 \quad a^{-m} = \frac{1}{a^m}$

$a^{m/n} = \sqrt[n]{a^m} = (\sqrt[n]{a})^m$

$\log_b x = y$ then $b^y = x$.

$\log_2 8 = 3$ since $2^3 = 8$

$\log_{10} 100 = 2$ since $10^2 = 100$.

for any base -

$\log_b 1 = 0$ since $b^0 = 1$

$\log_b b = 1$ since $b^1 = b$.

⇒ Algorithmic Notations

Algorithm Finite step-by-step list of well-defined instructions for solving a particular problem.

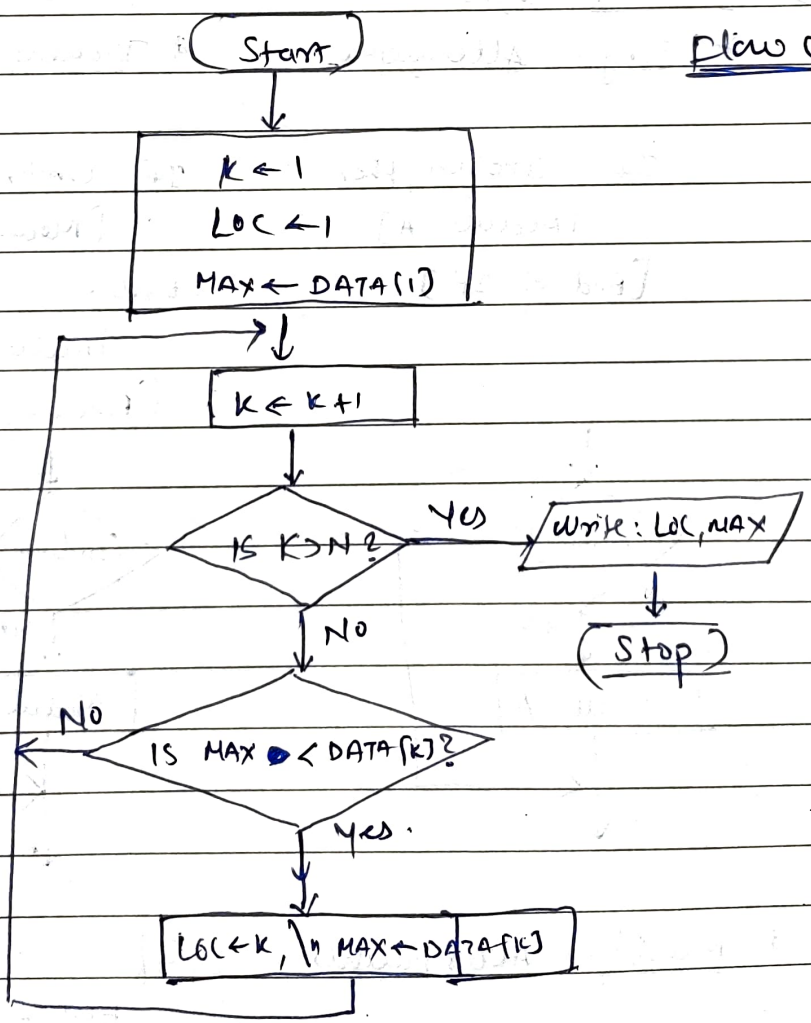
Algo to find max element in array and its location

Array - DATA[]
N ⇒ size of array
LOC ⇒ store location of max element
MAX ⇒ largest element
K ⇒ counter variable

Step 1. [Initialize] Set K := 1, LOC := 1 and MAX := DATA[1].
Step 2. [Increment counter] Set K := K+1.
Step 3. [Test counter] If K > N then:
Write: LOC, MAX and Exit.
Step 4. [Compare and update] If MAX < DATA[K] then:
set LOC := K and MAX = DATA[K].
Step 5. [Repeat Loop] Go to step 2.

* Comments :- In square Brackets.
* Variable Names :- in capital letters
* Assignment Statement :- := Notation used in Pascal.
* I/p and o/p :- READ and
Read : Variable Name
Write : Variable Name / message.

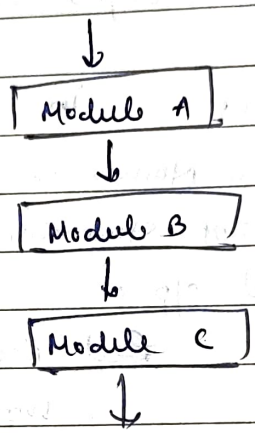
⇒ Flow chart



→ Control Structures

1. Sequence logic / sequential flow
2. Selection logic / conditional flow
3. Iteration logic / Repetitive flow

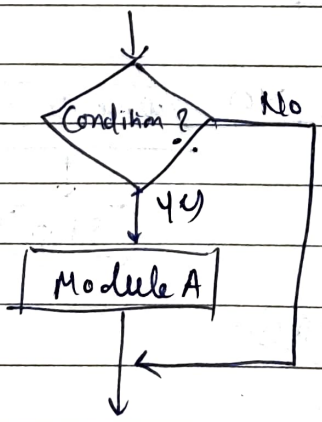
(1) Sequential logic



(2) Selection logic

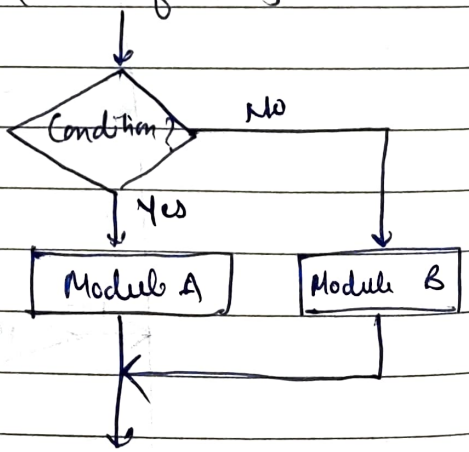
* Single Alternative

If condition, then
[Module A]
[End of If]



* Double Alternative

If condition, then
[Module A]
Else:
[Module B]
[End of If].



* Multiple Alternatives If

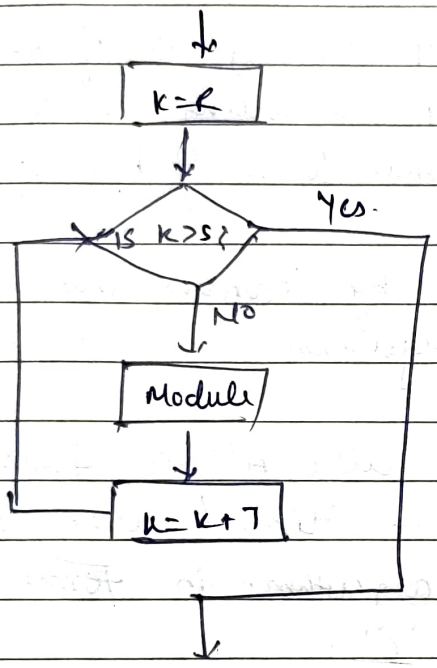
Elif.

③ Iteration logic

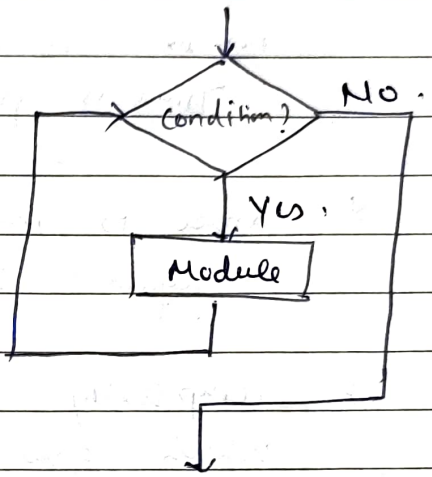
Begins with Repeat statement.

Repeat for K = R to S by T:
[Module].
[End of loop].

Repeat while cond^n
[Module]
[End of loop].



Repeat for structure



Repeat while structure

Algo :- to find max element using loops -

1. Set K = 1 LOC = 1 and MAX = DATA[1].
2. Repeat steps 3 and 4 while K ≤ N:
3. If MAX < DATA[K] then
Set LOC = K and MAX := DATA[K].
[End of If].
4. Set K = K+1.
[End of Step 2 loop]
5. Write: LOC, MAX.
6. Exit.

COMPLEXITY ANALYSIS

⇒ Complexity of Algorithms

* Analysis of algorithm is important task in CS.
* Some Criteria to measure efficiency of algo.
  - space
  - time

* Time is measured by counting no. of key operations for ex. In searching and sorting, no. of comparisons. Time is proportional to no. of key operations.

* Space is measured by counting max. of memory required by algorithm.

* Complexity of an algo 'A' is function $f(n)$ which gives running time / storage space requirement of algorithm in terms of input data size 'n'.

* Storage required is simply in multiple of 'n', so, we use term complexity to refer to running time of the algorithm.

(1) Worst Case :- maximum value of $f(n)$ for any input
(2) Average Case :- expected value of $f(n)$
(3) Best Case :- minimum possible value of $f(n)$.

Example :- Algo. to perform linear search.

A linear array DATA with N elements ~~are given~~ and specific ITEM are given. This algo finds location LOC of ITEM in the array DATA or sets LOC = 0 if not found.

1. Set K=1 and LOC=0
2. Repeat 3 & 4 while LOC=0 and K≤N
3. If ITEM = DATA[K] then
4. Set LOC=K;
   Set K=K+1
   [End of step 2 loop]
5. If LOC=0 then
   Write : ITEM is not in the DATA
   Else :
   Write : LOC is location of DATA.
6. Exit.

* Worst Case. occurs when ITEM is the last element of DATA or is not there at all in array.
  $C(n) = n$ — worst case complexity of linear search.

* Average case :- ITEM appear in DATA and is equally likely to appear at any position.
  No. of comparison can be any of 1, 2, 3, ..., n. and each occur with probability 1/n.

$$C(n) = 1\cdot\frac{1}{n} + 2\cdot\frac{1}{n} + 3\cdot\frac{1}{n} + \cdots + n\cdot\frac{1}{n}$$

$$= \frac{1}{n}(1+2+3+4+\cdots+n)$$

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

Note: Average case complexity of an algo is usually much more complicated to analyze than worst case. Also, probabilistic distribution we assume for average case may not actually apply in real situation.

* **Rate of Growth : Big O Notation**

Let M = algorithm n = size of input data
$f(n)$ increases as 'n' increases. Thus, it is rate of increase of $f(n)$ that we want to examine.

Ex:- $\log_2 n$ $n \log_2 n$ n^2 n^3 2^n

n \ $g(n)$	$\log n$	n	$n \log n$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	10^2	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	1000	10000	10^6	10^9	10^{300}

Rate of Growth of Standard functions.

Suppose $f(n)$ and $g(n)$ are functions defined on positive integers with property that $f(n)$ is bounded by some multiple of $g(n)$ for almost all 'n'.
ie.
There exist a positive integer n_0 and a positive no. 'k' such that for all $n > n_0$ –

$|f(n)| \leq k \cdot |g(n)|$ ie. $f(n) = O(g(n))$

For any polynomial of degree 'm', we get -

$$P(n) = O(n^m)$$

Ex :-

$$8n^3 - 576n^2 + 832n - 248 = O(n^3)$$

Linear search : $O(n)$
Binary search : $O(\log n)$
Bubble sort $O(n^2)$
Merge sort $O(n \log n)$.

* Omega Notation (Ω).

This used when the function $g(n)$ defines a lower bound for the fuction $f(n)$.

$f(n) = \Omega g(n)$ iff

There exist a positive integer n_0 and a positive no. 'k' such that -

$$f(n) \geq k \cdot g(n), \text{ for all } n > n_0.$$

Ex :- $f(n) = 18n + 9$

$f(n) > 18 \cdot n$ for all 'n'

$f(n) = \Omega(n)$.

Also : $f(n) = 5n + 1$

$f(n) = \Omega(n)$

$f(n) = \Omega(1)$ → we never consider latter.

as. $f(n) = \Omega(n)$ represents the largest possible fuction of 'n' satisfying the definition of Ω.

* Theta Notation

$f(n)$ is bounded from both above and below by function $g(n)$

$f(n) = \Theta(g(n))$ iff

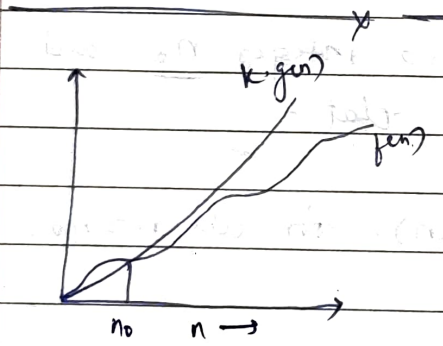
there exist two +ve constants k_1 & k_2 and a positive integer n_0 such that -

① $k_1 g(n) \leq f(n) \leq k_2 g(n)$ for all $n \geq n_0$

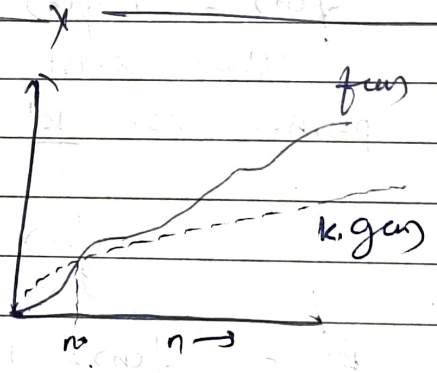
Ex: $f(n) = 18n + 9$

$f(n) \geq 18n$, $f(n) \leq 27n$ for $n \geq 1$

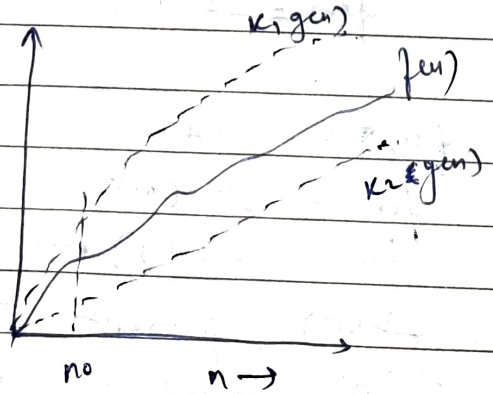
$f(n) = \Theta(n)$.



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$.



$f(n) = \Theta(g(n))$

ARRAYS

* Arrays, Records, Pointers.

Operations on Linear structure

1. Traversal
2. Searching
3. Insertion
4. Deletion
5. Sorting
6. Merging

⇒ Linear Arrays

↳ List of finite no. 'n' of homogeneous data elements (elements of same type).

↳ Elements are referenced by 'index set' [1 to n]

↳ Array elements are stored in successive memory locations.

Length or size of array = UB - LB + 1

Example :- DATA = 247, 56, 629, 135, 89, 158.

AUTO [k] = no. of automobiles sold in year 'k'.

Representation in memory -

Computer Memory

Let - LOC [A[K]] = address of element A[K] of array A.

As elements are stored in continuous memory locations, no need to keep track of addresses of all elements. We can only track address of first element

Base (A) → base address of A.

then -

$$LOC(A[K]) = Base(A) + w(K - LB)$$

where -

w - no. of words per memory cell.

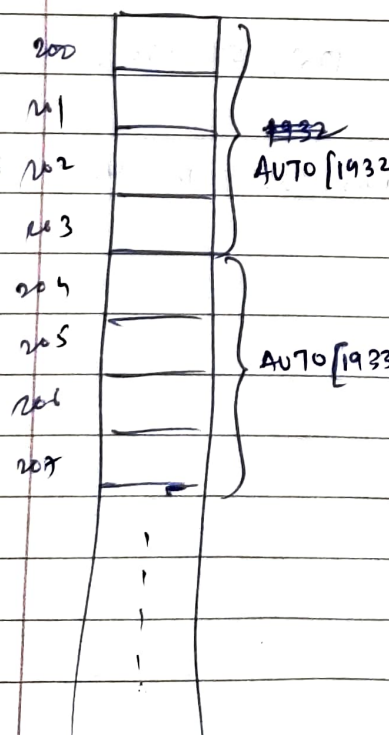
Ex: LOC (AUTO (1932)) = 200

LOC (AUTO [1933]) = 204

⋮

Address for k = 1965 -

$$LOC(AUTO[1965]) = Base(AUTO) + w(1965 - LB)$$
$$= 200 + 4(1965 - 1932)$$
$$= 200 + 4 \times 33$$
$$= 332$$



* Traversing the array

Algo :-

1. Repeat for K = LB to UB.
   Apply PROCESS to A[K].
   [End of loop]
2. Exit

Or

1. Set K := LB.
2. Repeat steps ③ & ④ while K <= UB.
3. Apply PROCESS to A[K].
4. K = K+1
   [End of while loop]
5. Exit.

Example :- Find no. of years when sales of automobiles is greater than 300 :-

If AUTO[K] > 300 then set COUNT = COUNT + 1

or

Print Year and no. of automobiles sold

Write K, AUTO[K].

* Insertion and Deletion

Insert :- Add element to array

Delete :- Remove element from array.

INSERT (A, N, K, ITEM). → Insert item at K^{th} position

1. Set J = N
2. Repeat 3 and 4 while J ≥ K.
3. Set A[J+1] = A[J]
4. Set J = J-1 ;

[End of loop].

5. Set A[K] = ITEM
6. Set N = N+1
7. Exit.

DELETE (A, N, K, ITEM)

1. Set ITEM = A[K]
2. Repeat for J = K to N-1
   Set A[J] = A[J+1].

[End of loop]

3. Set N = N-1
4. Exit.

Note :- Array is not an efficient way of storing data when we have to frequently insert and delete items.

Assignment 1.

1) Write and algorithm for Bubble sort. Do its complexity analysis, Best, Average, worst, write a working example for 5 elements

2) Write an algorithm for Linear search. Do its complexity analysis : Best, Average, Worst

Binary Search

BINARY (DATA , LB , UB , ITEM , LOC)

(1) Set BEG = LB END = UB MID = INT((BEG+END) /2)

(2) Repeat steps (3) & (4) while BEG ≤ END and DATA[MID] ≠ ITEM

(3) If ITEM < DATA[MID] then:
Set END = MID - 1
Else
Set BEG = MID + 1
[End of if]

(4) Set MID = INT((BEG+END) /2)
[END of step 2 loop]

(5) If DATA[MID] = ITEM then:
Set LOC = MID.
Else
Set LOC = NULL
[End of If structure].

(6) Exit.

Example :-

DATA = 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

ITEM = 40.

(1) BEG = 1 END = 13 MID = (1+13)/7 = 7 DATA(7) = 55

(2) 40 < 55 so END = MID - 1 = 6.
MID = INT((1+6)/2) = 3 DATA[3] = 30

(3) 40 > 30. so BEG = MID + 1 = 4.
MID = (4+6)/2 = 5 DATA[5] = 40.
We have found ITEM then LOC = MID = 5

Suppose ITEM = 85

	BEG	END	MID	DATA[MID]	Condition
(1)	1	13	7	55	55 < 85
(2)	8	13	10	77	77 < 85
(3)	11	13	12	88	88 > 85
(4)	11	11	11	80	80 < 85
5	(12)	(11)	→ BEG > END → out of loop.		

Complexity for Binary search

worst case when value is not present or present at extreme ends

Iterations	Size of Array
0	n
1	n/2
2	$n/4 = n/2^2$
⋮	
k	$n/2^k = 1$

$n = 2^k$.

$$k = \log_2 n$$

or form the recurrence equation.

$$T(n) = T(n/2) + 1$$

$$T(n) = O(\log n)$$

Example: DATA contains 1000000 elements

the $2^{10} \geq 1024 > 1000$

$2^{20} \geq (1000)^2 = 1000000$

} only 20 comparisons.

Multidimensional Array

Rows \ Columns				
	A(1,1)	A(1,2)	A(1,3)	A(1,4)
	A(2,1)	A(2,2)	A(2,3)	A(2,4)
	A(3,1)	A(3,2)	A(3,3)	A(3,4)

Two dimensional 3×4 Array.

Length = UB − LB + 1

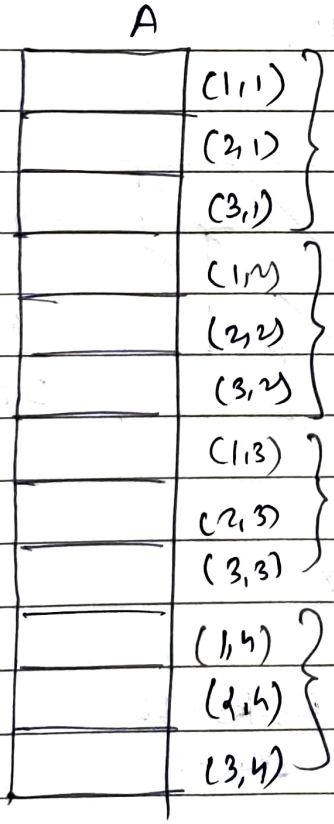
Ex: In FORTRAN INTEGER NUMB (2:5, −3:1)

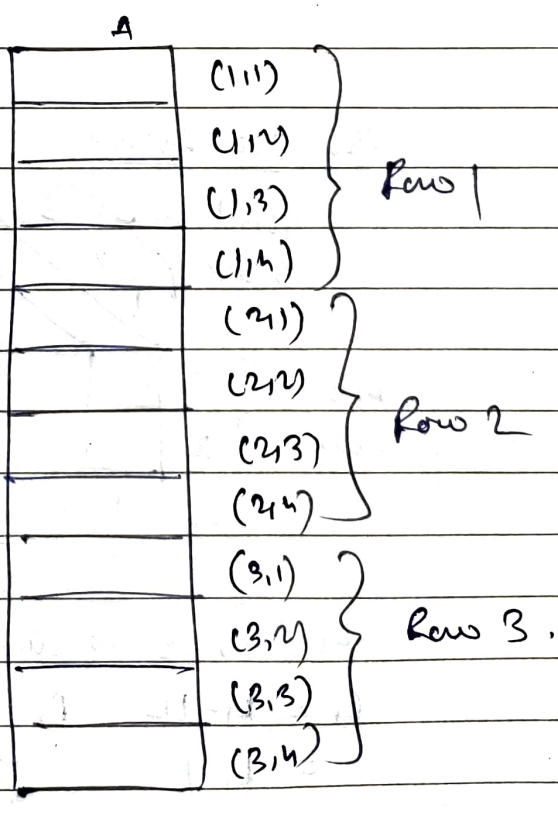
Length of 1st dimension: 5 − 2 + 1 = 4

" " 2nd dimension: 1 − (−3) + 1 = 5

Representation of 2-D Array in Memory.

Recall LOC(A[I,K]) = Base(A) + w(K − LB)





(a) Column-Major order (b) Row Major order

Column Major order.

$$Loc(A[J,K]) = Base(A) + w[M(K-1) + (J-1)]$$

Row Major order

$$LOC(A[J,K]) = Base(A) + w[N(J-1) + (K-1)]$$

Example :-

Consider 2 5×4 Matrix SCORE.

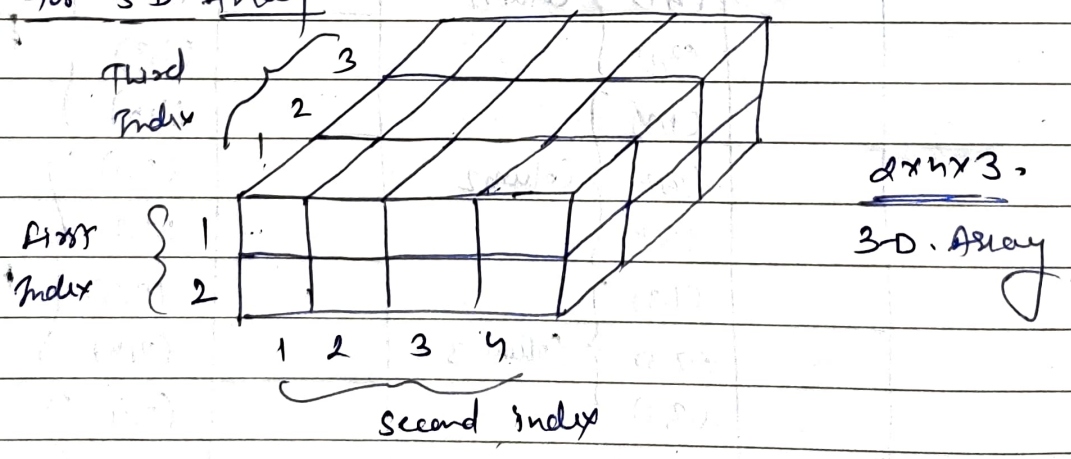
Base (SCORE) = 200 and w=4.

It is stored using row major order.

SCORE (12,3) = ?

$$LOC(SCORE[12,3]) = 200 + 4[4(12-1) + (3-1)]$$
$$= 200 + 4[44+2]$$
$$= 384.$$

For 3D Array



- L_1, L_2, L_3 → dimensions

$E_1 = k_1 - LB$ $E_2 = k_2 - LB$ $E_3 = k_3 - LB$

$= k_1 - 1$ $= k_2 - 1$ $= k_3 - 1$

STACK

* Stacks

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

Special terminologies are used for two basic operations associated with stack.

a) PUSH - insert an element into a stack
b) POP - delete an element from a stack.

These terms are used only with stacks and not with other data structures.

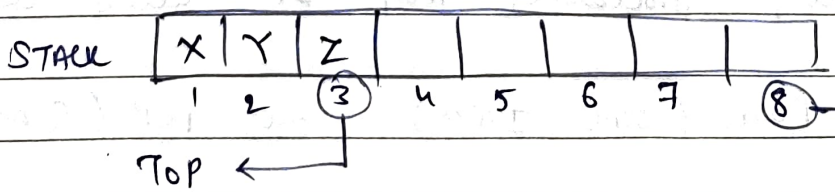
Example :- function call within another function.

* Array Representation of Stack

Stacks are represented in computer in various ways. Unless and otherwise stated, we consider that stacks are maintained by linear array called STACK, a pointer variable TOP, which contains address of the top element and a variable MAXSTK which gives the maximum no. of elements that can be held by a stack.

TOP=0 or TOP=NULL → stack is empty.

Ex :-



PUSH and POP operation on STACK

Before Push, we first check whether there is room for inserting one element, else we have the condition known as "Overflow".

Similarly, befor pop, we first check whether there are elements to be deleted, else we have the condition known as "Underflow".

PUSH (STACK, TOP, MAXSTK, ITEM)

(1) [Stack Already Filled]
    If TOP = MAXSTK then print: OVERFLOW and Return
(2) Set TOP = TOP+1 [Increase TOP by 1]
(3) Set STACK [TOP] = ITEM
(4) Return.

POP (STACK, TOP, ITEM).

(1) [Stack has an item to be removed ?]
    If TOP=0 then print: UNDERFLOW and Return.
(2) set ITEM= STACK [TOP]
(3) TOP := TOP - 1;
(4) Return.

Frequently, TOP and MAXSTK are global variables, hence, procedures may be called using only -

- PUSH (STACK, ITEM) and POP (STACK, ITEM)

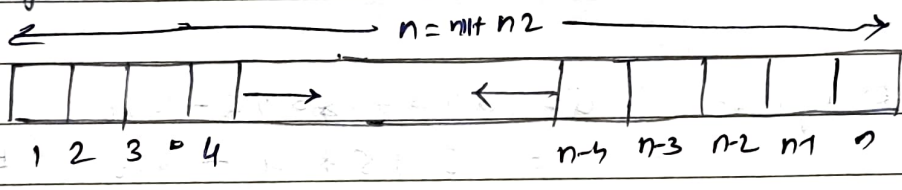
* Minimizing Overflows

There is trade-off involved while reserving amount of memory for stack.

As size ↑ overflow ↓

size ↓ overflow ↑.

So various techniques have been developed that modify standard array representation of stack. These techniques lie outside the scope of this subject.



Stack A (n1) Stack B (n2)

This could be a solution to decrease the overflow even though we have not increased the total amount of space reserved for the two stacks. Here, overflow will occur only when A and B together have more than $n = n1 + n2$ elements.

* Linked list Representation of Stack

This will be covered later when we finish studying linked list.

Please go to Page No ____ for this topic.

* Arithmatic Expression ; POLISH Notation

Let Q = arithmatic expression that contains operators and operands.

Here, we try to obtain the value of Q using reverse POLISH Notation. (Postfix). Here stack would be a useful data structure.

Recall, Highest — Exponentiation (↑)
Next Highest — Multiplication (*) and Division (/)
Lowest — Addition (+) and Subtraction (-)

Example :- Q = a↑3 + 5 * a↑2 - 12/6.
= 8 + 5 * 4 - 12/6 (Expo)
= 8 + 20 - 2 (Multi / Divi)
= 26

* POLISH Notation

A+B C-D E*F G/H → Infix Notation.

Important to distinguish between.

(A+B) * C and A+(B*C)

Thus, order of operators or operands may not uniquely determine the operation order that are to be performed.

POLISH notation (named after Jan Lukasiewicz) Refers to the notation, where operator symbol is placed before its two operands.

Ex :.

+AB - CD * EF / GH.

Example - Infix to Polish - (Also known as prefix)

$(A+B) * C = [+AB] * C = *+ABC$

$A+(B*C) = A+(*BC) = +A*BC$

$(A+B)/(C-D) = (+AB)/(-CD) = /+AB-CD$.

Imp property of Polish Notation is that order in which operations are to be performed is completely determined by the positions of operators and operands. Accordingly, we never need paranthesis when using Polish Notations.

Reverse Polish notation refers to the analogous notation in which the operator symbol is placed after its two operands.

AB+ CD- EF* GH/ (Also known as postfix).

Computer usually evaluate the arithmatic expression written in infix notation in two ways-
first, it converts the expression into postfix notation and then evaluate the postfix expression.
At each step, stack the main tool that is used to accomplish the task.

Evaluate Postfix Expression

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

Commas are used as separators.

↳ Infix notation is - 5*(6+2) - 12/4.

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

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

↑ Sentinel

Symbol Scanned		STACK
(1)	5	5
(2)	6	5, 6
(3)	2	5, 6, 2
(4)	+	5, 8
(5)	*	40
(6)	12	40, 12
(7)	4	40, 12, 4
(8)	/	40, 3
(9)	-	37 → Final Answer.
(10)	)	Stop.

Ex
34+5* = 35
231*+9- = -4.

Algo

1. Add a paranthesis ")" at the end of P. [As a sentinel]
2. Scan P from left to right and repeat (3) and (4) for each element of 'P' until ")" is encountered.
3. If an operand is encountered, push it on STACK
4. If an operator ⊗ is encountered then:
   a) Remove top two elements from STACK, where A is top element and B is next to top element.
   b) Evaluate B ⊗ A
   c) Push the result back in STACK.
5. Set VALUE equal to top element on STACK. (6) Exit.

Transform Infix to Postfix :-

Example :- Q = A + (B * C - (D / E ↑ F) * G) * H

Push "(" onto STACK and add ")" to end of Q.

Q: A + (B * C - (D / E ↑ F) * G) * H)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Symbol scanned	STACK	Expression P.
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) -	(+ (-	A B C *
(8) (	(+ (- (	A B C *
(9) D	(+ (- (	A B C * D
(10) /	(+ (- (/	A B C * D
(11) E	(+ (- (/	A B C * D E
(12) ↑	(+ (- (/ ↑	A B C * D E
(13) F	(+ (- (/ ↑	A B C * D E F
(14))	(+ (-	A B C * D E F ↑ /
(15) *	(+ (- *	A B C * D E F ↑ /
(16) G	(+ (- *	A B C * D E F ↑ / G
(17))	(+	A B C * D E F ↑ / G * -
(18) *	(+ *	A B C * D E F ↑ / G * -
(19) H	(+ *	A B C * D E F ↑ / G * - H
(20))		A B C * D E F ↑ / G * - H * +

↳ Required Postfix Notation.

POLISH (Q, P)

(1) Push "(" onto STACK and add ")" at the end of Q.

(2) Scan Q from left to right and repeat (3) to (6) for each element of Q until the STACK is empty.

(3) If an operand is encountered, add it to P.

(4) If a left Parantheris is encountered, push it onto STACK.

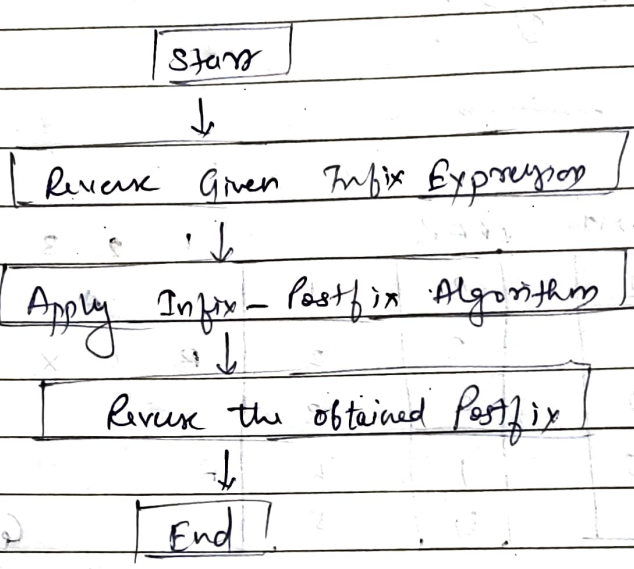
(5) If an '⊗' operator is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
   (b) Add ⊗ to STACK
   [End of If structure].

(6) If a right parenthesis is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add to P].
   [End of If structure].
[End of step 2 loop].

(7) Exit.

Infix to Prefix Conversion

Steps:-



Example :- $(A+B\wedge C)*D+E\wedge 5$

Step 1 Reverse and make '(' as ')' & ')' as '('.

$5\wedge E+D*((\wedge B+A)$

Step 2

Symbol Scanned	Stack	Expression
5	Empty	5
$\wedge$	$\wedge$	5
E	$\wedge$	5E
+	+	5E$\wedge$
D	+	5E$\wedge$D
*	+*	5E$\wedge$D
(	+*(	5E$\wedge$D
C	+*(	5E$\wedge$DC
$\wedge$	+*($\wedge$	5E$\wedge$DC
B	+*($\wedge$	5E$\wedge$DCB
+	+*(+	5E$\wedge$DCB$\wedge$

A	+ * (+	5E^DCB^A
)	+ *	5E^DCB^A+
End	Empty	5E^DCB^A+*+

Step 3 Reverse the obtained postfix.

+*+A^BCD^E5 → Required prefix Notation.

Cross check.

(A + B ^ C) * D + E ^ 5

→ (A + ^BC) * D + ^E5

→ +*+A^BCD^E5 → prefix.

Example (2)

Infix :- (A-B/C) * (A/K-L)

Postfix :- ABC/-AK/L-*

Prefix :- *-A/BC-/AKL

By Algorithm.

Reverse :- (L-K/A) * (C/B-A)

Postfix :- LKA/- CB/A- *.

Reverse :- *-A/BC-/AKL.

RECURSION

* Recursion

Suppose 'P' is a procedure that contains a call statement to itself or a call statement to another procedure that may eventually result in a call statement back to original procedure P. Then 'P' is called a recursive procedure.

Recursive fuction must have two properties-

(1) There must be certain criteria, called base criteria, for which the procedure does not call itself.

(2) Each time the procedure does call itself (directly / indirectly) it must be closer to base criteria.

Procedure with above properties is said to be well defined.

⇒ Factorial fuction

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

Also,

$$n! = n \cdot (n-1)!$$

Factorial fuction can be defined as:

a) If $n=0$ then $n! = 1$

b) If $n>0$ then $n! = n(n-1)!$

This definition is recursive in nature.

Example

Lets calculate 4! using this definition.

1) $4! = 4 \cdot 3!$
2) $3! = 3 \cdot 2!$
3) $2! = 2 \cdot 1!$
4) $1! = 1 \cdot 0!$
5) $0! = 1 \rightarrow$ Base co
6) $1! = 1 \cdot 1 = 1$
7) $2! = 2 \cdot 1 = 2$
8) $3! = 3 \cdot 2 = 6$
9) $4! = 4 \cdot 6 = 24$

FACTORIAL (FACT, N)

1. If N=0 the set FACT = 1 and Return.
2. Set FACT = 1
3. Repeat for K=1 to N
   Set FACT = FACT * K.
   [End of loop]
4. Return.

FACTORIAL (FACT, N)

1. If N=0 then set FACT=1 and Return.
2. Call FACTORIAL (FACT, N-1)
3. Set FACT = N * FACT.
4. Return.

Complexity = O(n). $t(n) = t(n-1) + 1$

$t(n) = t(n-2) + 1 + 1$

* FIBONACCI Sequence

The fibonacci sequence (usually denoted by $f_0, f_1, f_2, \ldots$) is as follows -

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$f_0 = 0$ & $f_1 = 1$ and each succeeding term is the sum of previous two terms.

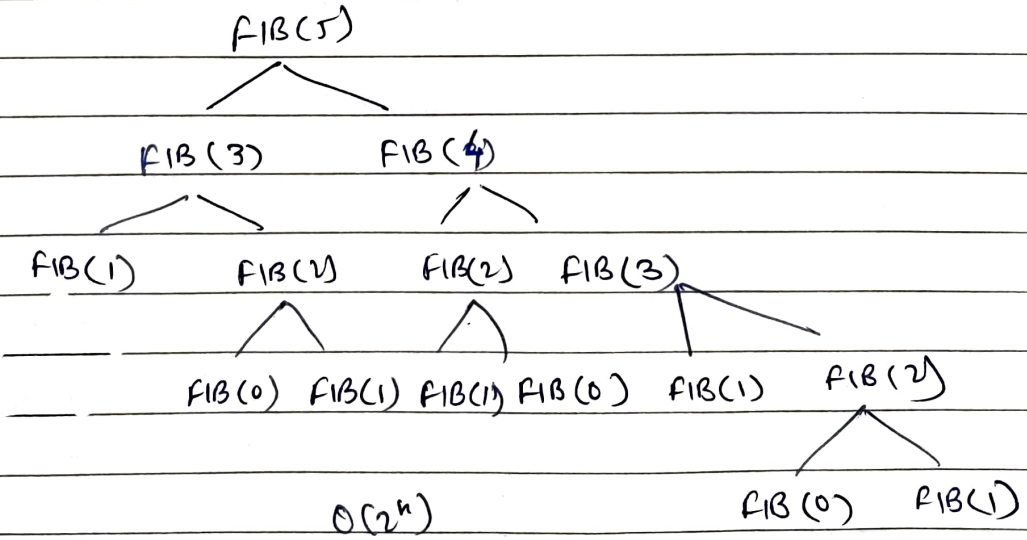
Definition

a) If $n=0$ or $n=1$ $f_n = n$

b) If $n>1$ then $f_n = f_{n-2} + f_{n-1}$

FIBONACCI (FIB, N)

1. If N=0 or N=1 then set FIB := N & Return
2. Call FIBONACCI (FIBA, N-2)
3. Call FIBONACCI (FIBB, N-1)
4. Set FIB = FIBA + FIBB
5. Return.

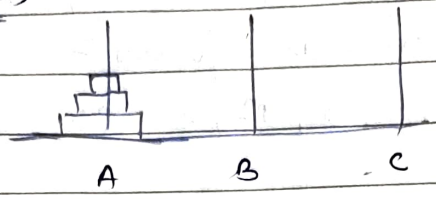


$O(n)$ ⇓ for non recursive call

$O(2^n)$ ⇓ for Recursive call.

TOWER of HANOI

for N=3



A→C , A→B , C→B , A→C , B→A , B→C , A→C

TOH can be reduced to the following subproblems

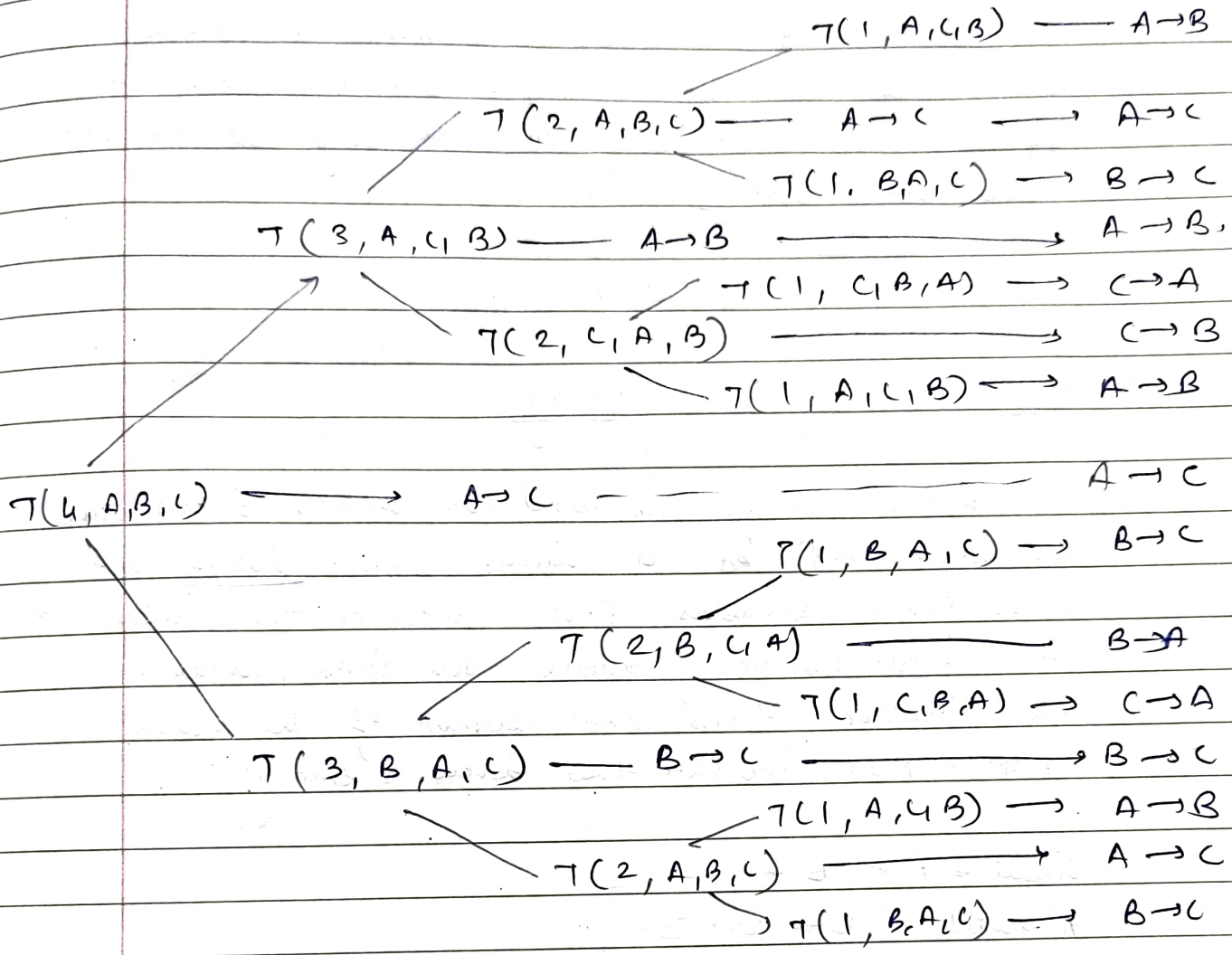
1. Move top (n-1) disk from A → B
2. Move top disk from A → C
3. Move the top (n-1) disk from B → C

when N=1 , TOWER (1, BEG, AUX, END)

BEG → END.

when N>1

1. TOWER (N-1, BEG, END, AUX)
2. TOWER (1, BEG, AUX, END) or BEG → END
3. TOWER (N-1, AUX, BEG, END)



Recursive solution to TOH for n=4.

QUEUE

* Queue :-

Queue is a linear list of elements in which deletion can take place only at one end called the "front" and insertion can take place only at the other end called the "rear".

Queue are also called FIFO lists.

* Representation of Queue (Array)

Queue may be represented as array. Queue can be maintained by a linear array "QUEUE" and two pointer variables FRONT containing the location of the first element of queue, and REAR, containing the location of the rear element of the queue.
FRONT = NULL indicate that Queue is empty.

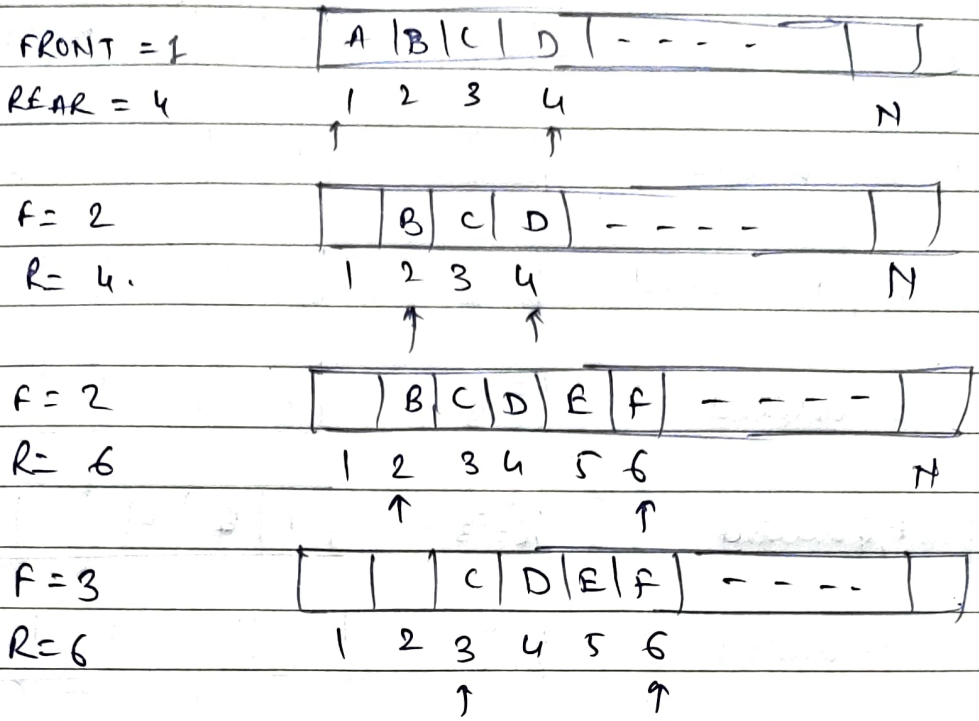
Queue → A → B → C → D.
'A' deleted B → C → D.
E, F inserted B → C → D → E → F
'B' deleted C → D → E → F.

whenever the element is deleted from the queue, the value of FRONT is increased by 1.

FRONT = FRONT + 1

Similarly, whenever an element is added to the queue the value of REAR is increased by 1.

REAR = REAR + 1



When Queue is full to right but space is available at left and we want to insert element, in this scenario, we will have to shift all elements to left. This is quiet expensive. Hence we consider that queue is circular, i.e. QUEUE[1] comes after QUEUE[N].

Ex:- N = 5 - Queue is initially empty.

a) Empty — F=0, R=0

b) A, B, C, inserted — F=1, R=3 — A B C

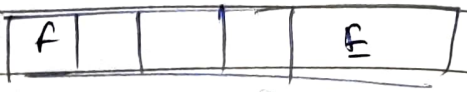
c) A deleted — F=2, R=3 — B C

d) D, E inserted — F=2, R=5 — B C D E

e) B, C, deleted — F=4, R=5 — D E

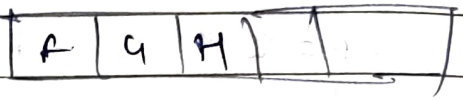
f) F inserted — F=4, R=1

g) D deleted — F=5, R=1



h) G, H inserted — F=5, R=3

i) E deleted — F=1, R=3



j) F, G deleted — F=3, R=3

k) H deleted — F=0, R=0

QINSERT (QUEUE, N, FRONT, REAR, ITEM)

① [Queue already filled?]
If FRONT = 1 and REAR = N or FRONT = REAR + 1
Write: OVERFLOW and Return.

② [Find new value of REAR]
If FRONT = NULL then
Set FRONT = REAR = 1
Else If REAR = N then
Set REAR = 1
Else
Set REAR = REAR + 1
[End of If].

③ Set QUEUE(REAR) = ITEM

④ Return.

QDELETE (QUEUE, N, FRONT, REAR, ITEM)

(1) [Queue already empty?].
If FRONT = NULL then
Write: UNDERFLOW and Return.

(2) Set ITEM = QUEUE[FRONT]

(3) [Find new value of FRONT]
If FRONT = REAR then [only one element in queue]
Set FRONT = REAR = NULL.
Else If FRONT = N then
Set FRONT = 1
Else
Set FRONT = FRONT + 1
[End of If structure].

(4) Return

* Linked List Representation of Queue.

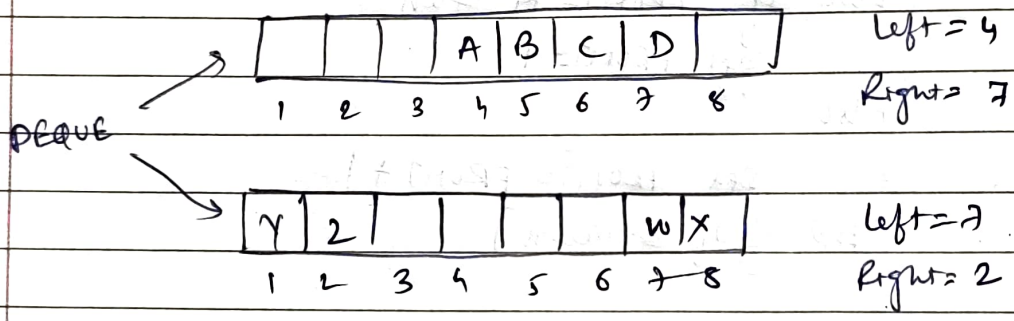
This will be covered later when we finish studying linked list.

Please go to Page No_____ for this topic.

DEQUES

A deque is a linear list in which elements can be added or removed at either end but not in the middle. The term is contraction of name "Double Ended Queue".

Unless and until implied we assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, pointing to the two ends of deque.



Two variations of a deque -

(1) Input Restricted dequeue:-

It is a deque which allow insertions at only one end of the list but allows deletions at both end of the list.

(2) Output Restricted deque.

It is a deque which allows deletions at only one end of the list and allows insertions at both end of the list.

Assignment 1:-

3) Write algos to insert element from front & rear in DEQUE and for delete from front & rear in Deque.

PRIORITY QUEUES

A priority queue is a collection of elements such that elements has been assigned a priority and such that the order in which elements are deleted and processed" comes from the following rules :-

① An element of higher priority is processed before any element of lower priority.

② Two elements with same priority are processed according to the order in which they were added to the queue.

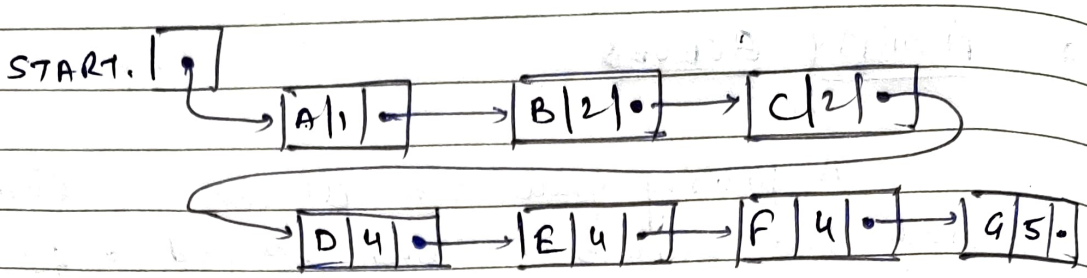
We will see two ways of maintaining a queue in memory. One that uses a one-way list and the other uses multiple queues.

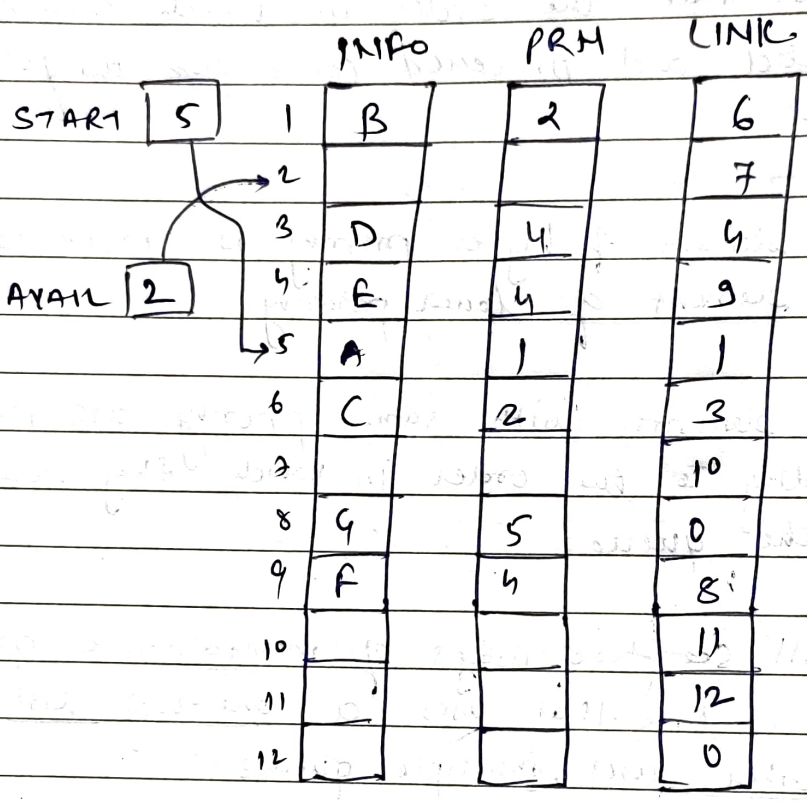
One-way list Representation of Priority Queue.

a) Each node in the list contains three items of information : an information field INFO, a priority number PRN and a link no. LINK.

b) A node 'x' precedes a node 'y' in the list
1) when 'x' has higher priority than 'y' or
2) both have same priority but 'x' was added to the list before 'y'.

Note : Lower the priority number, higher the priority.





Array Representation of PRIORITY QUEUE

Another way to maintain a priority Queue in a memory is to use separate queue for each level of priority (or each priority number).

Each such queue will appear in its own circular way and must have its own pair of pointers FRONT and REAR.

Thus, we can use a Two dimensional array to represent PRIORITY QUEUE instead of multiple linear arrays.

Example :-

	FRONT
1	2
2	1
3	0
4	5
5	4

	REAR
1	2
2	3
3	0
4	1
5	4

	1	2	3	4	5	6
1			A			
2	B	C	X			
3						
4	F				D	E
5				G.		

NOTE :- FRONT [K] and REAR [K] contains respectively the front and the rear elements of row 'k' of QUEUE, the row that contains the queue of elements with priority 'k'.

Algo to Delete

1. Find the smallest 'k' such that FRONT [K] ≠ NULL.
2. Delete and process the front element in row 'k' of QUEUE.
3. Exit.

Algo to INSERT

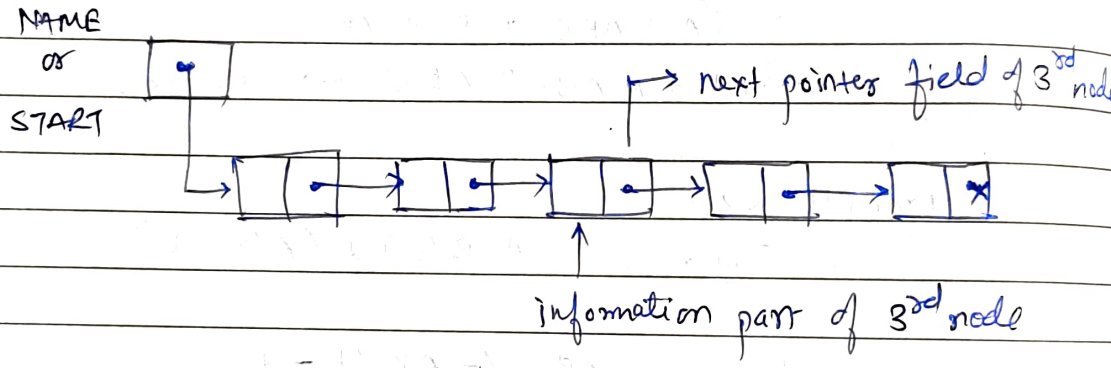
1. Insert ITEM as the rear element in row 'M' of QUEUE, where M = priority of ITEM.
2. Exit

LINKED LISTS

LINKED LIST.

A linked list or a one-way list is a linear collection of data elements called "nodes" where the linear order is given by means of "pointers". That is each node is divided into two parts, the first part contains the information of the element and the second part, called the link field, contains the address of the next node in the list.

Schematic diagram of a linked list with 5 nodes.



The pointer of the last node contains a special value, called the "null" pointer, which is any invalid address. In practice, 0 or a negative number is used for the null pointer.

X - to indicate null pointer in list.
START - contains address of the first node.
We need only this address to trace through the list.

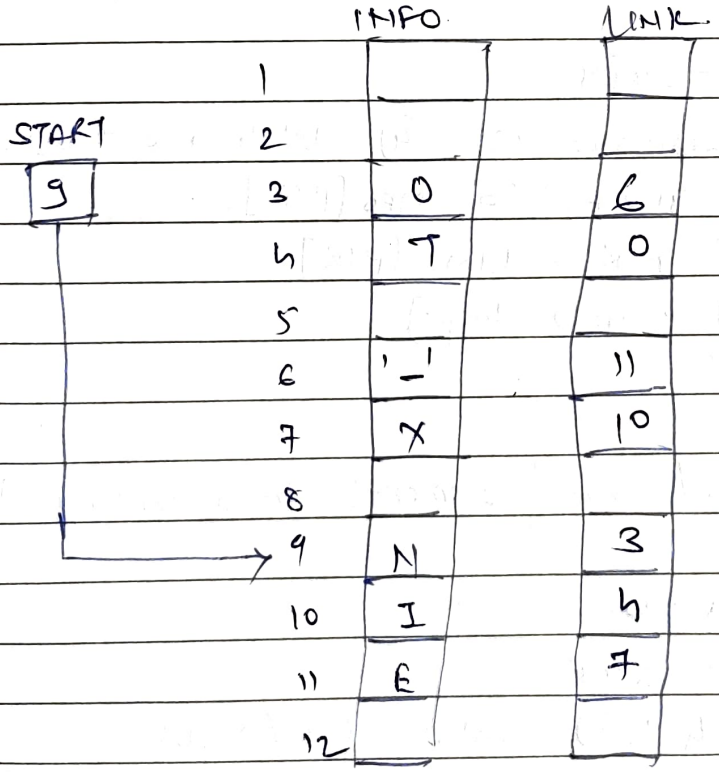
A special case is the list that has no nodes. Such a list is called the null list or empty list and is denoted by the null pointer in START.

* Representation of LINKED LIST IN MEMORY.

Let LIST, be a linked list. First, LIST requires two linear arrays — INFO and LINK — such that INFO[K] and LINK[K] contain, respectively the information part and next pointer field of a node of LIST.

LIST also requires a variable - such as START - which contains the location of the beginning of the list and a next nextpointer sentinel denoted by NULL - which indicates the end of the list. In general, we choose NULL = 0.

Example :-



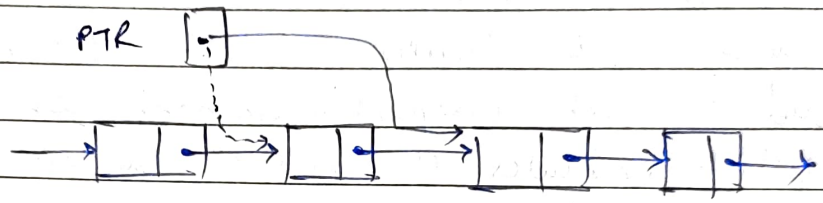
Similarly, we can maintain two linked list (or more) using these two arrays.

Example :- Given in Introduction of linked list.

Traversing a linked list

Here, for traversing algorithm, we used a pointer variable PTR which points to the node. Accordingly, LINK[PTR] points to the next node.

Thus, PTR = LINK[PTR]



PTR = LINK[PTR].

Algorithm :- TRVERSE(INFO, LINK, PTR, START).

1. Set PTR = START
2. Repeat ~~step~~ step 3 & 4 while PTR ≠ NULL.
3. Apply PROCESS to INFO[PTR].
4. Set PTR := LINK[PTR].
5. [End of step 2 loop]
6. Exit.

Algo to find no. of elements in a linked list

1. Set NUM := 0
2. Set PTR := START.
3. Repeat 4 & 5 while PTR ≠ NULL
4. Set NUM := NUM + 1
5. Set PTR := LINK[PTR]
   [End of step 3 loop]
6. Exit.

* Searching a Linked List

LIST is Unsorted

SEARCH (INFO, LINK, START, ITEM, LOC).

1. Set PTR = START.
2. Repeat 3 while PTR ≠ NULL
3. If ITEM = INFO[PTR] then:
   Set LOC = PTR and Exit.
   Else
   Set PTR = LINK[PTR]
   [End of if structure]
   [End of step 2 loop]
4. Set LOC := NULL
5. Exit.

List is Sorted

SEARCHSL (INFO, LINK, START, ITEM, LOC)

1. Set PTR := START
2. Repeat step 3 while PTR ≠ NULL
3. If ITEM > INFO[PTR] then:
   Set PTR := LINK[PTR]
   Else If ITEM = INFO[PTR] then
   set LOC := PTR and Exit.
   Else:
   Set LOC := NULL and Exit
   [End of if structure]
   [End of step 2 loop]
4. Set LOC = NULL
5. Exit.

* Memory Allocation ; Garbage Collection.

⇒ Maintenance of linked list assume possibility of inserting new nodes in lists and hence, some mechanism which provides unused memory space for these new nodes.

⇒ So, some mechanism is required whereby the memory space of deleted nodes becomes available for future use.

⇒ Thus, a special list is maintained having its own pointer, called as "list of available space" or "free storage list" or the "free pool"
We call this list as "AVAIL"

Thus linked list will be denoted by writing -
LIST (INFO, LINK, START, AVAIL).

Ex :-

START = 5

AVAIL = 10

	INFO	LINK
1	E	7
2		6
3	B	11
4	G	12
5	A	3
6		10
7	F	4
8	D	1
9	I	0
10		2
11	C	8
12	H	9

⇒ Suppose some memory space becomes reusable because a node is deleted from list or a list is deleted from program.

⇒ We want this space to be available for future use. So, we immediately reinsert this space into the "free storage list".

⇒ This is what we do when we implement linked list using arrays. However, this method can be too time-consuming for ~~os~~ Operating system, which may choose to use following alternative -

⇒ OS periodically collect all deleted space onto the free storage list. Any technique, which does this collection is called garbage collection.

⇒ It usually takes place in two steps -

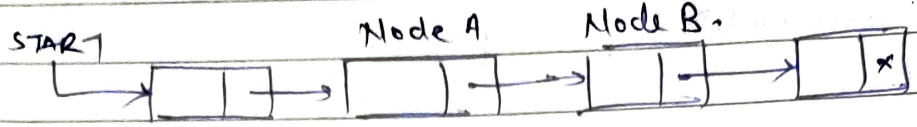
i) First, computer runs through all lists tagging those cells that are currently in use.

ii) Then computer runs through memory, collecting all untagged cells into free storage list.

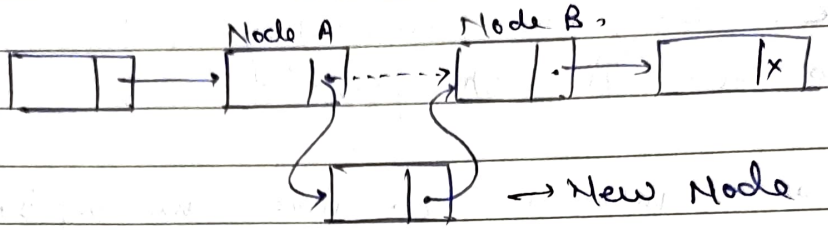
⇒ Garbage collection takes place when there there is little space or no space left in the storage list. Or when CPU is idle.

⇒ Generally speaking, garbage collection is invisible to the programmer.

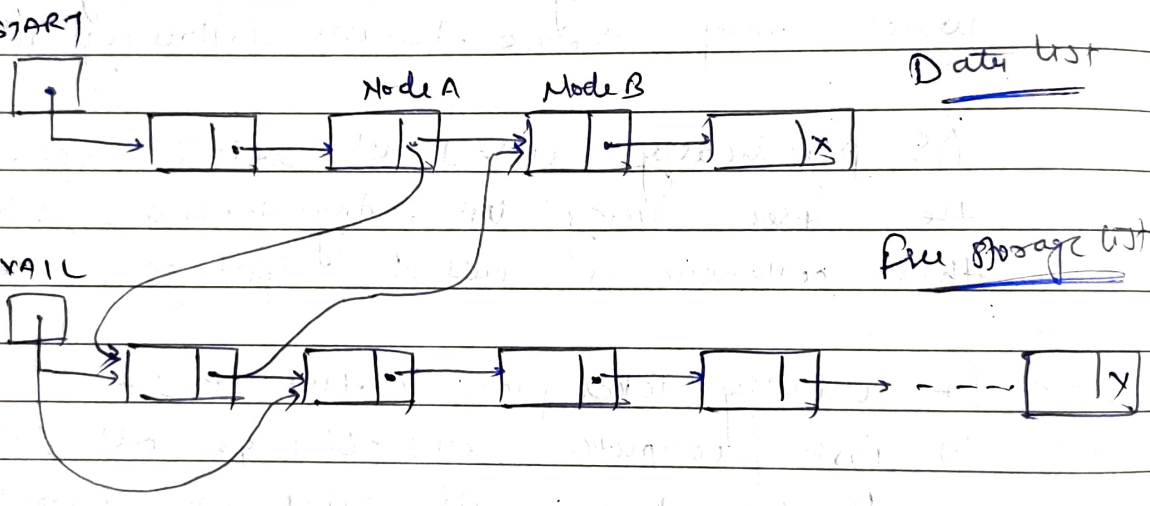
* Insertion into a linked list



Before insertion



After insertion



NEW = AVAIL and AVAIL := LINK [AVAIL]

INFO [NEW] = ITEM.

Removing first node from AVAIL.

Allocating item to new node

This new node is now ready to be inserted at its correct position.

Insert at the beginning.

* INSERT (INFO, LINK, START, AVAIL, ITEM)

① If AVAIL = NULL then:
write: overflow and Exit.

② [Remove first node from avail]
set NEW := AVAIL , AVAIL := LINK [AVAIL].

③ Set INFO [NEW] := ITEM

④ Set LINK [NEW] = START.

⑤ Set START := NEW.

* Insert after given node:
INSERT LOC (INFO, LINK, STAR, AVAIL, LOC, ITEM).

① If AVAIL = NULL : then overflow and Exit

② NEW = AVAIL and AVAIL := LINK[AVAIL].

③ INFO [NEW] = ITEM.

④ If LOC = NULL then : [Insert at first].
LINK [NEW] := START & START = NEW.
Else
⑤ LINK [NEW] = LINK [LOC] and LINK[LOC] = NEW.

⑥ Exit.

Assignment 1

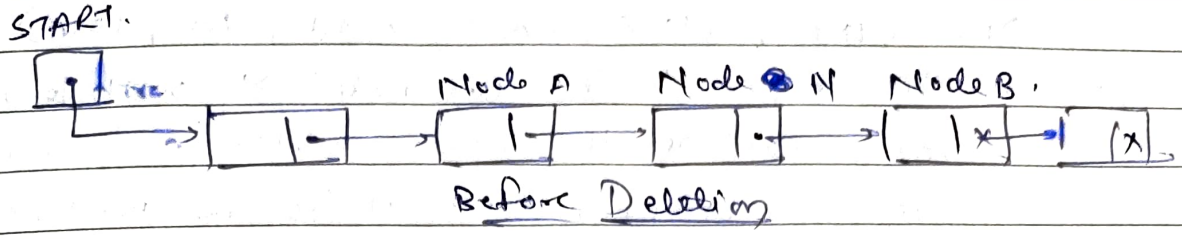
4) Write an algorithm to insert a node into sorted linked list. Assume that linked list is sorted in ascending order.

Hint:- Obtain "LOC" value to get the location where an item is to be inserted. Then call previous algo by passing this loc.

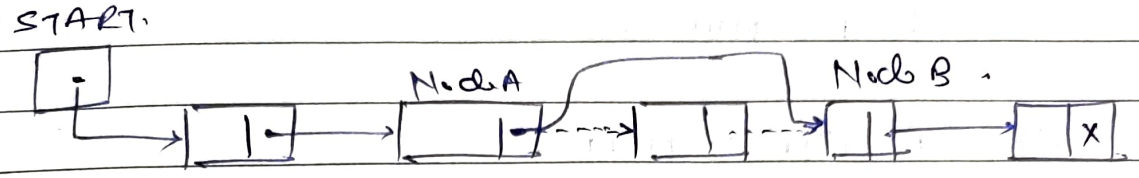
FINDA (INFO, LINK, START, ~~AVAIL~~, ITEM, LOC)

1. If START = NULL then set LOC := NULL & Exit
2. [special case]
   If ITEM < INFO[START] then set LOC := NULL and Exit.
3. Set SAVE := START and PTR := LINK[PTR].
4. Repeat 5 & 6 while PTR ≠ NULL.
5. If ITEM < INFO[PTR] then
   Set LOC := SAVE and Return.
   [End of if].
6. Set SAVE := PTR and PTR := LINK[PTR].
   [End of Step 4 loop].
7. Set LOC := SAVE
8. Exit.

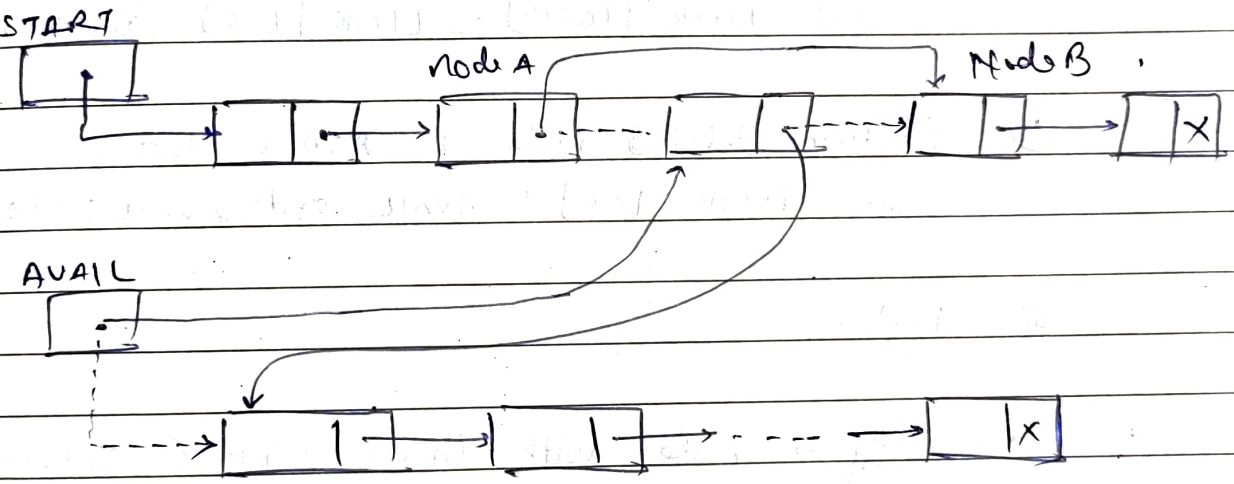
* Deletion from a linked list.



Before Deletion



After Deletion



fw storage list.

LINK [LOC] := AVAIL and AVAIL := LOC.

Add the deleted node into avail and then link node A with node B.

* Deleting a Node following a given node

DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

⇒ This algo deletes the node N with location LOC. LOCP is the location of the node which precedes N or, when N is the first node, LOCP = NULL.

① If LOCP := NULL,
Set START := LINK [START] [Delete 1st node]
Else
Set LINK [LOCP] := LINK [LOC] [Delete node N]

② [Return deleted node to AVAIL list]
Set LINK [LOC] := AVAIL and AVAIL := LOC

③ Exit.

* Deleting a node with Given ITEM of Information

FINDB (INFO, LINK, START, ITEM, LOC, LOCP)

① [List empty] If START := NULL then:
Set LOC := NULL and LOCP := NULL & Return

② [ITEM in first Node]. If INFO [START] = ITEM then
Set LOC := START and LOCP := NULL & Return

③ Set SAVE := START and PTR := LINK [START]

④ Repeat Steps ⑤ and ⑥ while PTR := ≠ NULL.

(5) If INFO[PTR] = ITEM then
Set LOC := PTR and LOCP := SAVE and Return

(6) Set SAVE := PTR and PTR = LINK[PTR]
[End of Step (5) Loop]

(7) Set LOC := NULL (Search Unsuccessful)

(8) Return.

- DELETE (INFO, LINK, START, AVAIL, ITEM)

(1) Call FINDB (INFO, LINK, START, ITEM, LOC, LOCP)

(2) If LOC = NULL then:
Write: ITEM not in LIST and Exit.

(3) [Delete node]
If LOCP = NULL then:
Set START := LINK[START]
Else
Set LINK[LOCP] := LINK[LOC]

(4) Set LINK[LOC] := AVAIL and AVAIL := LOC.

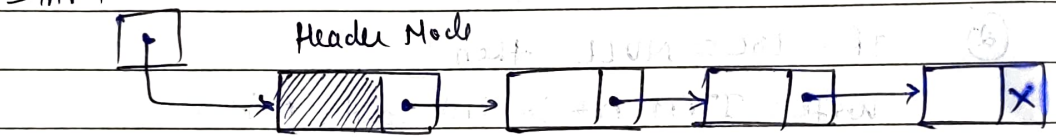
(5) Exit.

* Header Linked List

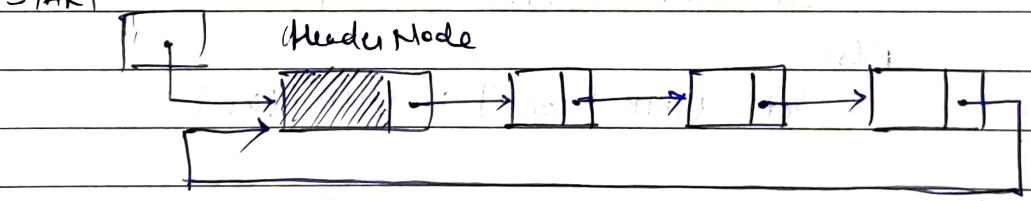
A header linked list is a linked list which always contains a special node, called the "header node" at the beginning of the list.

The following are the two widely used header lists :-

(1) A grounded header list :- It is a header list where the last node contains the null pointer.



(2) A circular header list :- It a header list where the last node points back to the header node



Unless otherwise stated or implied, our header list will always be circular. Accordingly, in such cases, the header node also acts as a sentinel indicating the end of the list.

LINK [START] = NULL ⟶ Grounded list is empty

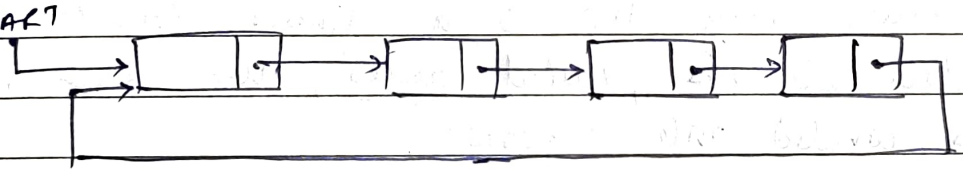
LINK [START] = START ⟶ Circular list is empty.

Example:- Traversing a Circular Header List.

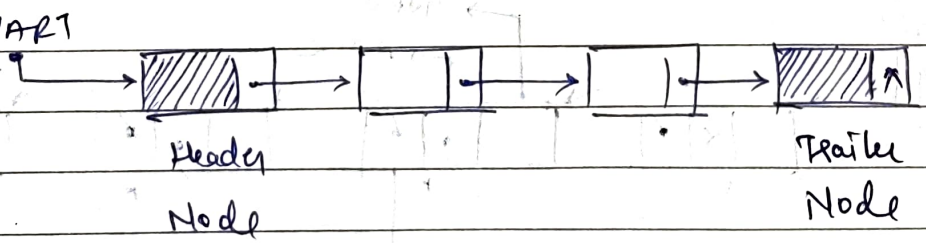
1. Set PTR := LINK[START]
2. Repeat steps 3 & 4 while PTR ≠ START
3. Apply PROCESS to INFO[PTR]
4. Set PTR := LINK[PTR]
5. Exit.

* Variations of Linked List -

1. A Linked List whose last node points back to the first node instead of containing null pointer, is called a circular list.



2. A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of the list.



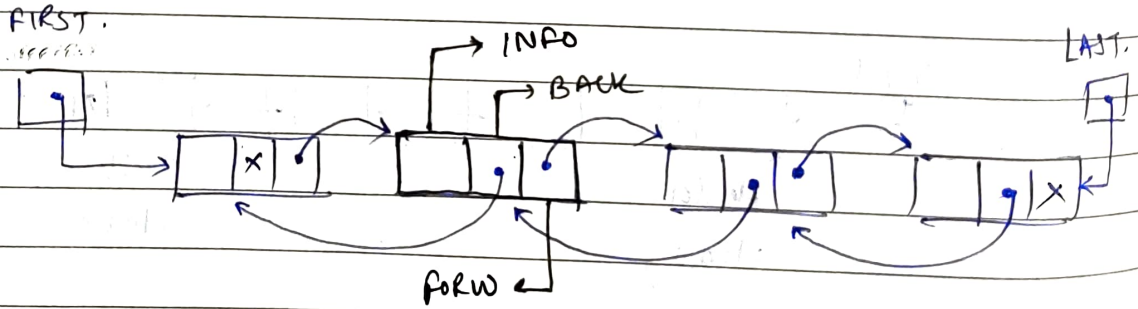
* Two - Way List

It is a list that can be traversed in both the directions. That is, in the forward direction from the beginning of the list to the end, or in the backward direction, from the end of the list to the beginning.

Furthermore, given the location Loc of a node 'N', we have access to both the next node and previous node in the list. That is, one can delete the node without traversing the list.

A two way list is a linear collection of data elements, called 'nodes', where each node is divided into 3 parts :-

1) INFO part -
2) Location of Next node - FORW.
3) Location of previous node - BACK.



The list also requires two list pointers :-

1) FIRST :- points to the first node in the list
2) LAST :- points to the last node in the list.

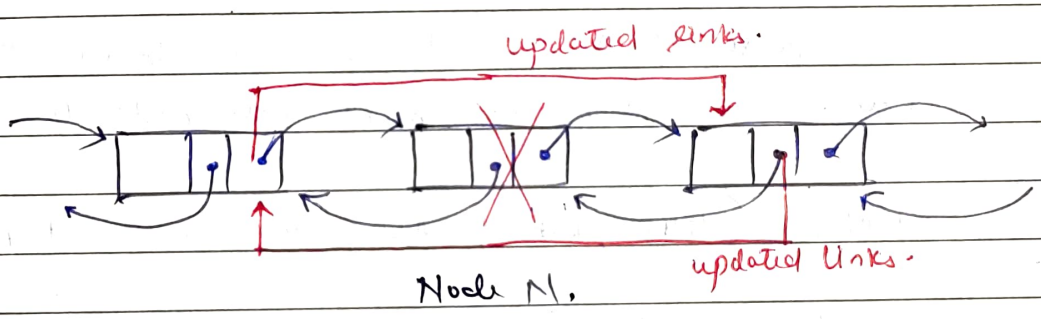
* Operations on Two-way lists.

⇒ **Traversing and Searching**

Both these operations can be performed exactly similar to that of one-way linear list. There is no advantage of having a two-way list for traversing and searching.

→ **Delete :-**

Suppose we want to delete node 'N' with location given as 'LOC' from list -



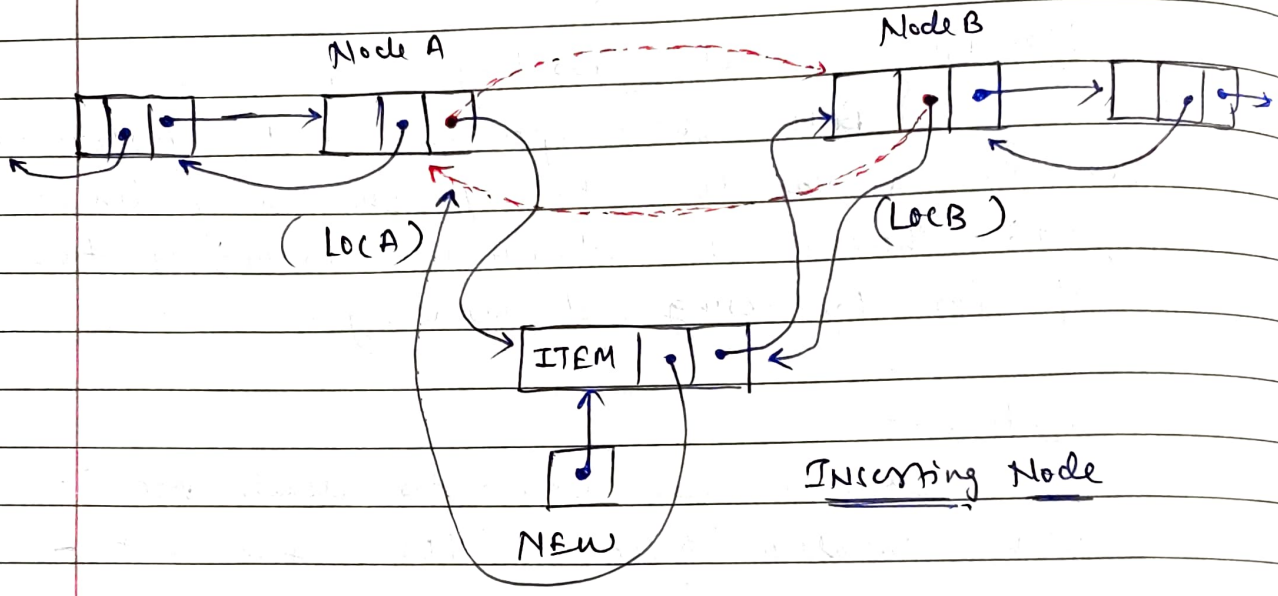
DELETE (INFO, FORW, BACK, START, AVAIL, LOC).

1) [Delete Node],
Set FORW[BACK[LOC]] := FORW[LOC] and
BACK[FORW[LOC]] := BACK[LOC]

2) [Add deleted node to AVAIL list]
Set FORW[LOC] := AVAIL and AVAIL := LOC.

3) Exit.

* Insert a node in two-way list -



Suppose LOCA and LOCB are location of adjacent node. we need to insert new node in between.

INSERT (INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

① [Overflow] If AVAIL := NULL then;
Write: Overflow and Return.

② [Remove node from AVAIL]
Set NEW := AVAIL, AVAIL := FORW[AVAIL]

③ INFO[NEW] := ITEM.

④ [Insert a Node]
Set FORW[LOCA] := NEW, FORW[NEW] := LOCB
BACK[LOCB] := NEW, BACK[NEW] := LOCA

⑤ Exit.

* STACK using Linked List -

PUSH_LINK_STACK (INFO, LINK, TOP, AVAIL, ITEM)

(1) If AVAIL = NULL then
Overflow and Exit

(2) NEW = AVAIL and
AVAIL = LINK[AVAIL]
} Remove first node from AVAIL.

(3) Set INFO[NEW] = ITEM

(4) Set LINK[NEW] = TOP
and TOP = NEW.

(5) Exit.

POP_LINK_STACK (INFO, LINK, TOP, AVAIL, ITEM)

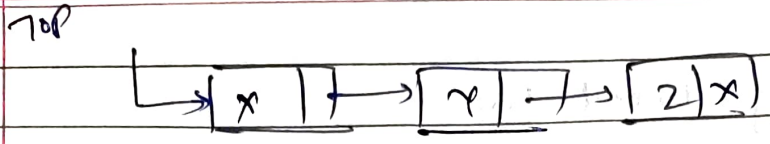
(1) If TOP = NULL then
Underflow and Exit

(2) Set ITEM = INFO[TOP]

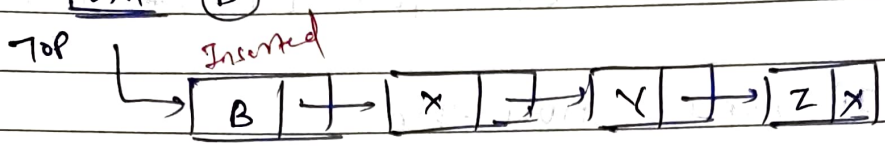
(3) Set TEMP = TOP and
TOP = LINK[TOP]
} Delete node & store it in TEMP.

(4) Set LINK[TEMP] = AVAIL and
AVAIL = TEMP
} Add deleted node to AVAIL.

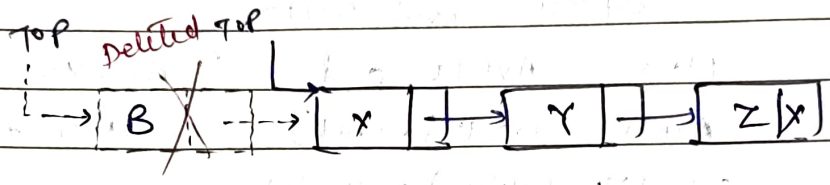
(5) Exit



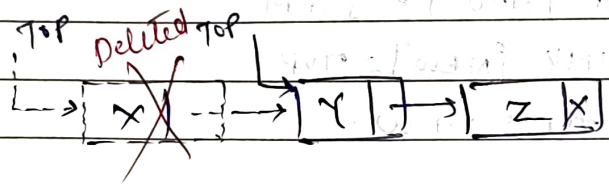
Push (B)



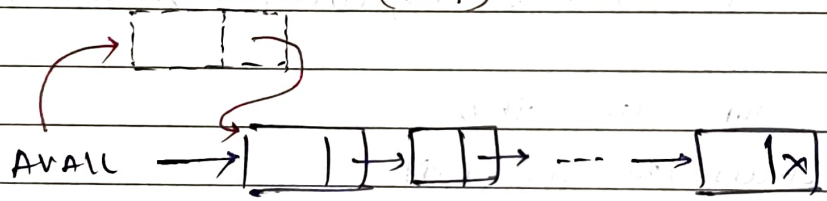
Pop



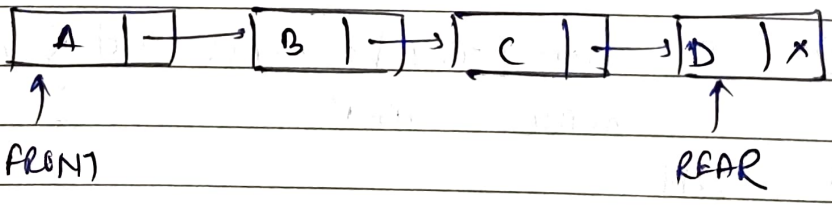
Pop



Add the deleted node to Avail List



Linked list Representation of Queue



Add elements from REAR

Delete elements from FRONT.

LINK_QINSERT (INFO, LINK, FRONT, REAR, AVAIL, ITEM)

1. If AVAIL = NULL then overflow and Exit
2. Set NEW = AVAIL
   AVAIL = LINK[AVAIL]
   } Remove first node from AVAIL.
3. Set INFO[NEW] = ITEM and
   LINK[NEW] = NULL
   } Copy item into new node
4. If FRONT = NULL then
   FRONT = REAR = NEW.
   Else.
   LINK[REAR] = NEW and REAR = NEW.
5. Exit.

LINK_QDELETE (INFO, LINK, FRONT, REAR, AVAIL, ITEM)

1. If FRONT = NULL then: Underflow and Exit.
2. Set TEMP = FRONT
3. ITEM = INFO[TEMP]
4. FRONT = LINK[TEMP] → Delete node
5. LINK[TEMP] = AVAIL and
   AVAIL = TEMP.
   } Add deleted node to AVAIL.
6. Exit.

TREES

TREES

Tree data structure is mainly used to represent data containing a hierarchical relationship between elements.

We study a special type of tree called a binary tree.

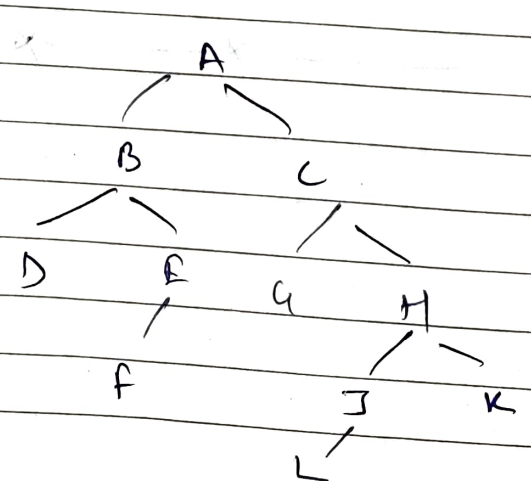
A Binary tree 'T' is defined as a finite set of elements called "nodes" such that -

a) T is empty (called as null tree or empty tree) or

b) T contains a distinguished node R, called the root of T, and the remaining of T forms an ordered pair of disjoint binary trees T_1 and T_2.

If tree 'T' contains root R, then T_1 and T_2 are called respectively the left and right subtree of R. If T_1 is non empty then its root is called left successor of R, similarly if T_2 is non-empty then its root is called right successor of R.

Ex:

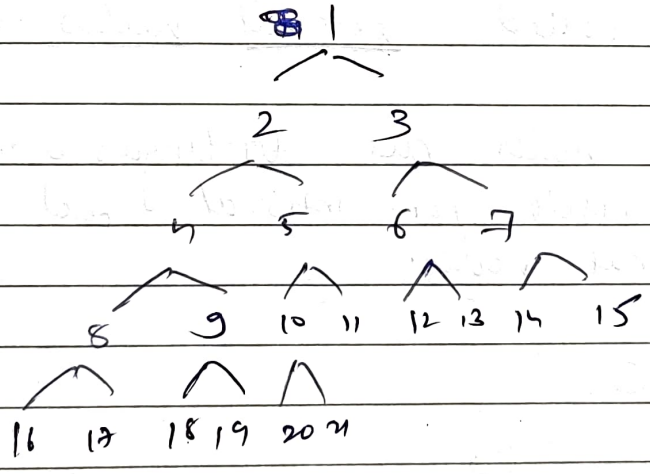


* Any node in binary tree T has either 0, 1 or 2 successor.
* The nodes with no successors are called terminal nodes. / leaf nodes

Complete Binary tree

A Binary tree node can have at most two children. Accordingly, level 'i' of T can have at most 2^i nodes. The tree is said to be complete if all the nodes except possibly the last have maximum no. of successors and if all nodes at last level appears as far left as possible.

Ex:-



Complete Binary Tree (T)

With this labelling, we can exactly determine the children and parent of any node 'k' in any complete tree 'T'. Specifically, the left and right child of the node 'k' are respectively $2k$ and $2k+1$ and parent of 'k' is $\lfloor k/2 \rfloor$

The depth of complete tree T_n with 'n' nodes is given by -

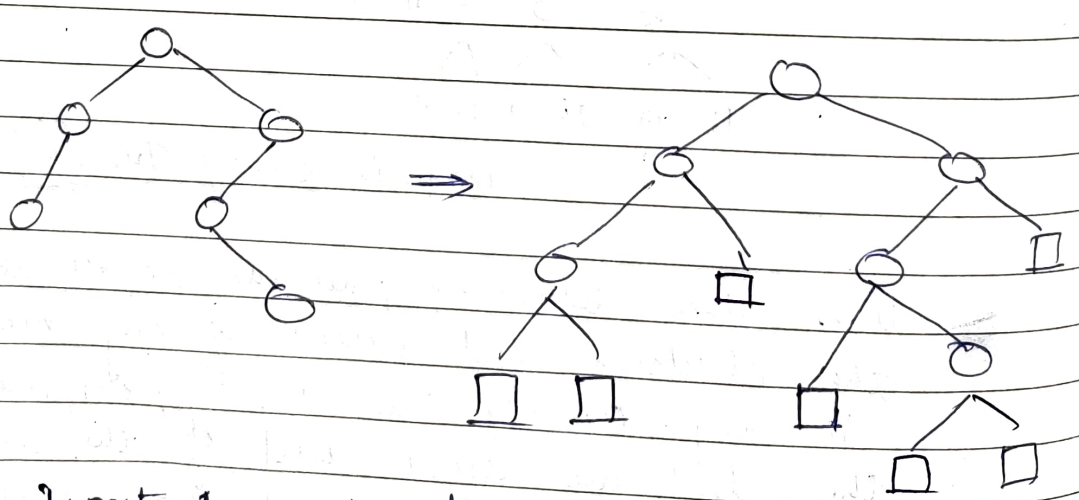
$$D = \lfloor \log_2 n + 1 \rfloor$$

Ex :- If T_n has 1000000 nodes then $D = 21$

Extended Binary trees or 2-Trees

A binary tree 'T' is said to be a 2-tree or an extended binary tree if each node 'N' has either '0' or '2' children. In such case, nodes with 2 children are called internal nodes and nodes with 0 children are called external nodes.

These nodes are distinguished in diagram by using circle for internal and square for external nodes.



Important example of 2-tree is tree 'T' corresponding to algebraic expression, where variables = external node, operators = internal nodes.

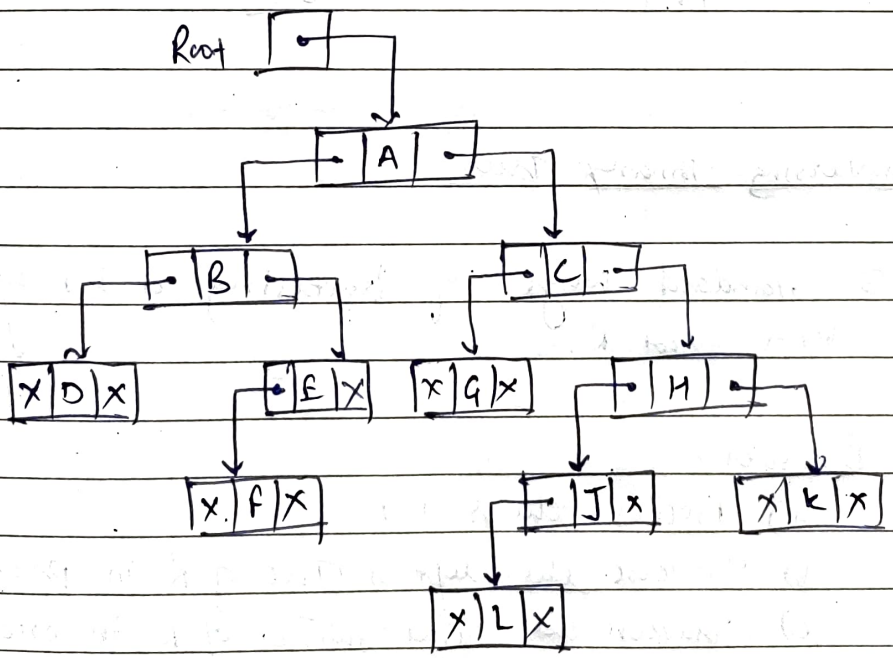
Representing trees in memory

Main requirement for any representation of 'T' in memory is that one should have direct access to the root 'R' of 'T' and given any node 'N' of 'T', one should have direct access to the children of N.

Linked Representation of Binary Tree

To represent Binary tree, we need '3' arrays

1. INFO [K] contains data of Node N.
2. LEFT [K] contains location of the left child of N.
3. RIGHT [K] contains location of right child of N.



		INFO	LEFT	RIGHT
ROOT 5	1	K	0	0
	2	C	3	6
	3	G	0	0
	4		(7)	
→	5	A	10	2
	6	H	8	1
	7		(9)	
	8	J	15	0
AVAIL 9	9		(0)	
	10	B	14	13
→	11		(4)	
	12	F	0	0
	13	E	12	0
	14	D	0	0
	15	L	0	0

* Traversing Binary Trees

3 standard ways of Traversing a binary tree 'T' with root R.

(1) Preorder.

a) Process the root R

b) Traverse the left subtree of R in preorder

c) Traverse the right subtree of R in preorder

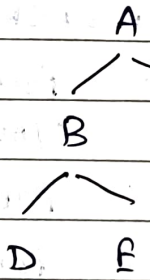
(2) Inorder

a) Traverse the left subtree of R in inorder
b) Process the root R
c) Traverse the right subtree of R in inorder

(3) Postorder

a) Traverse the left subtree of R in postorder
b) Traverse the right subtree of R in postorder
c) Process the root R

Ex :-



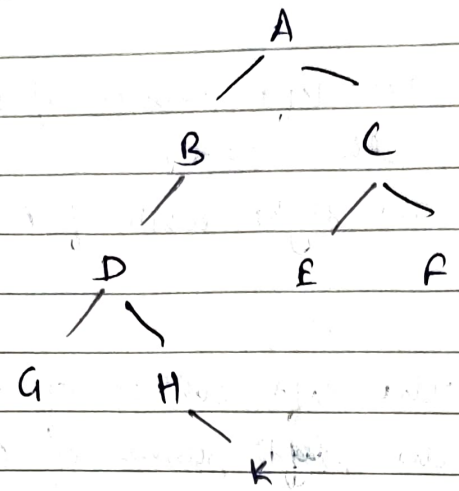
Preorder : A B D E C
Inorder : D B E A C
Postorder : D E B C A.

* Traversing using Stack

(1) Preorder :- (INFO, LEFT, RIGHT, ROOT)

1. [Initially push NULL onto STACK, and initialize PTR]
   Set TOP := 1 STACK[1] := NULL & PTR = ROOT
2. Repeat (3) to (5) while PTR ≠ NULL
3. Apply PROCESS to INFO[PTR]
4. [Right child] If RIGHT[PTR] ≠ NULL
   Set TOP = TOP+1 & STACK[TOP] := RIGHT[PTR]
   [End of If]
5. [Left child] If LEFT[PTR] ≠ NULL then,
   Set PTR = LEFT[PTR].
   Else PTR := STACK[TOP] & TOP = TOP - 1
   [End of If]
6. Exit. [End of step (2) loop]

Ex :-



1. STACK = 0
   PTR = A.
2. Print 'A'
   STACK = 0, C
   PTR = B.
3. Print 'B'.
   STACK = 0, C
   PTR = D
4. Print 'D'.
   STACK = 0, C, H
   PTR = G
5. Print 'G'.
   STACK = 0, C
   PTR = H.
   [G has no child, so POP]
6. Print 'H'
   STACK : 0, C, K.
7. PTR = K
   Print 'K'
   STACK : 0, C
8. Pop C and set and PTR = C
9. STACK = 0
10. Print 'C'
    STACK = 0, F
    PTR = E
11. Print 'E'
    STACK = 0, F
    No left child
12. Pop F & set PTR = F
13. STACK = 0
14. Print 'F'
    No left child or right child
    PTR = NULL [Pop]

Loop stops

Preorder
A B D G H K C E F.

(4) INORDER (INFO, LEFT, RIGHT, ROOT).

(1) Set TOP = 1 STACK[TOP] = NULL, PTR = ROOT.

(2) Repeat while PTR ≠ NULL.

a) Set TOP = TOP + 1 STACK[TOP] = PTR

b) PTR = LEFT[PTR].

[End of loop]

(3) Set PTR := STACK[TOP] and TOP = TOP − 1

(4) Repeat steps (5) to (7) while PTR ≠ NULL

(5) Apply PROCESS to INFO[PTR]

(6) [Right child] If RIGHT[PTR] ≠ NULL then

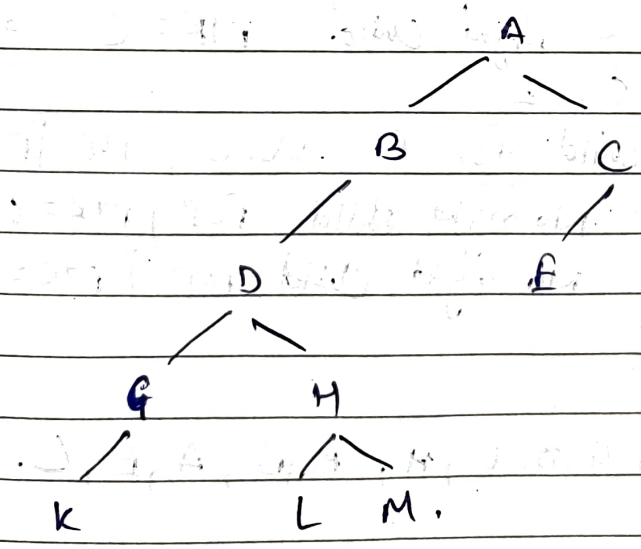
a) Set PTR = RIGHT[PTR]

b) Goto step (2)

[End of if]

(7) Set PTR = STACK[TOP] & TOP = TOP − 1

8. Exit.



1) STACK = 0
PTR = A

2) STACK : 0, A, B, D, G, K.
No left child for K. Hence, POP. [PTR = K]

3) Print (K), - No right child so pop [PTR=G]

4) Print (G) - No right child so POP [PTR=D]

5) Print (D) - Right child PTR = H go to (2)

6) STACK : 0, A, B, H, L
No left child for L Hence POP [PTR=L]

7) Print (L), -No Right child POP A [PTR = H]

8) STACK : 0, A, B, _ _ _ _

9) Print (H) - Right child PTR = M goto 2

10) STACK : 0, A, B, M
No left child for M Hence, POP [PTR=M]

11) Print (M) - No right child POP [PTR = B]

12) Print (B) - No right child POP [PTR = A]

13) Print (A) - Right child PTR = C goto (2)

14) STACK, 0, C, E
No left child for E Hence, POP [PTR=E]

15) Print (E) - No right child POP [PTR = C]

16) Print (C) - No right child POP [PTR = NULL]

Stop

INorder :- K, G, D, L, H, M, B, A, E, C.

(3) Post order (INFO, LEFT, RIGHT, ROOT)

(1) Set TOP = 1 STACK[1] = NULL PTR = ROOT.

(2) [Push left most path on STACK]
Repeat (3) to (5) while PTR ≠ NULL.

(3) TOP = TOP + 1 STACK[TOP] = PTR.

(4) If RIGHT[PTR] ≠ NULL then:
TOP = TOP + 1 STACK[TOP] = RIGHT[PTR] *(-1)
i.e. = - RIGHT[PTR].
[End if.]

(5) PTR = LEFT[PTR]
[End of step (2) loop]

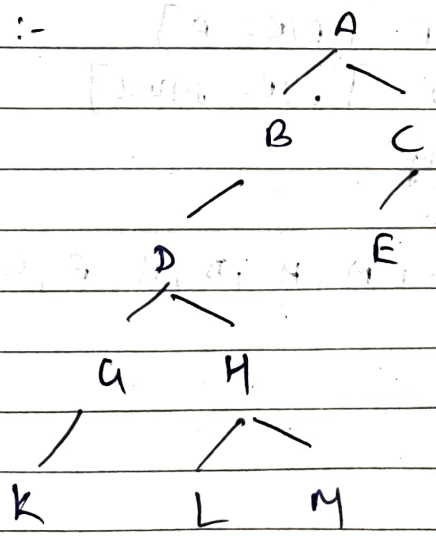
(6) PTR := STACK[TOP] and TOP = TOP - 1

(7) Repeat while PTR > 0
a) Apply PROCESS to INFO[PTR].
b) PTR := STACK[TOP] & TOP = TOP - 1
[End of loop]:

(8) If PTR < 0 then:
a) Set PTR := - PTR
b) Goto step (2)

(9) EXIT.

Ex :-



1) STACK := NULL ; PTR = A

2) STACK = NULL, A, -C, B, D, -H, G, K
K has not left child so POP [PTR = K]

3) Print 'K' POP [PTR = G]

4) Print 'G' POP [PTR = -H]
-ve so PTR = H Go to (2)

5) STACK = NULL, A, -C, B, D, H, -M, L
L has no left child so POP [PTR = L]

6) Print 'L' POP [PTR = -M]
-ve so PTR = M Go to (2)

7) STACK = NULL, A, -C, B, D, H, M
M has not left child so POP [PTR = M]

8) Print 'M' POP [PTR = H]

9) Print 'H' POP [PTR = D]

10) Print 'D' POP [PTR = B]

11) Print 'B' POP [PTR = -C]
-ve so PTR = C Go to (2)

12) STACK : NULL, A, C, E
E has no left child so POP [PTR = E]

13) Print 'E' POP [PTR = C]

14) Print 'C' POP [PTR = A]

15) Print 'A' POP [PTR = NULL]
Stop

Postorder :- K, G, L, M, H, D, B, E, C, A.

Binary Search Tree

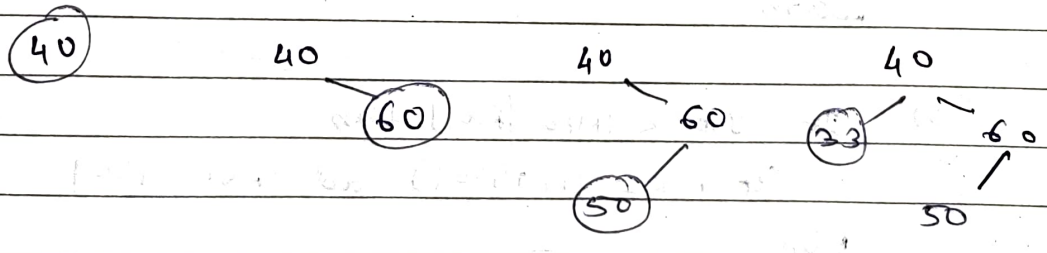
Searching and Inserting

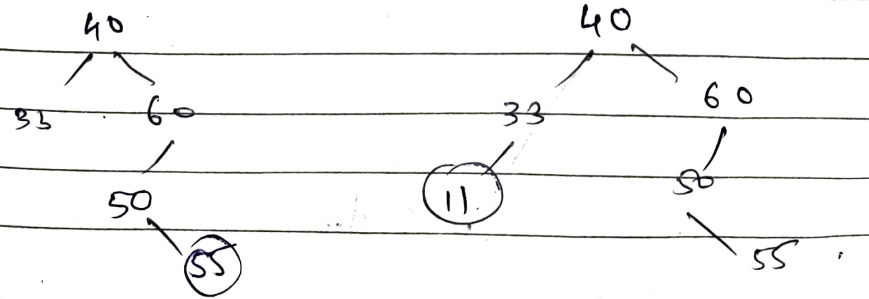
Suppose ITEM is given. following logic finds the location of ITEM in binary search tree 'T' or insert ITEM as a new node in its appropriate place in the tree.

a) Compare ITEM with root 'N' of tree.
   i) If ITEM < N, proceed to left child of N.
   ii) If ITEM > N, proceed to right child of N.

b) Repeat step (a) until one of the following occurs -
   i) We meet node 'N' such that item = N. In this case the search is successful.
   ii) We meet an empty subtree, which indicates that the search is unsuccessful and we insert ITEM in place of empty subtree.

Ex :- 40, 60, 50, 33, 55, 11





There are 3 special cases here -

i) LOC = NULL and PAR = NULL - indicates that the tree is empty.

ii) LOC ≠ NULL and PAR = NULL - indicates that ITEM is root of 'T'.

iii) LOC = NULL and PAR ≠ NULL - indicates ITEM is not in tree but can be added to 'T' as a child of 'N' with location PAR

Algorithm :-

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

This procedure finds LOC of ITEM in tree and also location PAR of parent of ITEM.

1) [Tree empty?]
If ROOT = NULL then set LOC = NULL & PAR = NULL.
Return

2) [ITEM at Root?]
If ITEM = INFO [ROOT] set LOC = ROOT, PAR = NULL
Return

3) If ITEM < INFO [ROOT] then
Set PTR = LEFT [ROOT] and SAVE = ROOT
Else
Set PTR = RIGHT [ROOT] and SAVE = ROOT

contd..

④ Repeat steps ⑤ and ⑥ while PTR ≠ NULL.

⑤ If ITEM = INFO[PTR] then
Set LOC = PTR & PAR = SAVE
Return

⑥ If ITEM < INFO[PTR] then
Set SAVE = PTR and PTR = LEFT[PTR]
Else
Set SAVE = PTR and PTR = RIGHT[PTR].

⑦ [Search unsuccessful] Set LOC = NULL, PAR = SAVE.

⑧ Exit.

INSERT (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC).

① Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
② If LOC ≠ NULL then Exit.
③ [copy ITEM into new node in AVAIL LIST].
a) If AVAIL = NULL then overflow & Exit.
b) Set NEW = AVAIL & AVAIL = LEFT[AVAIL]
INFO[NEW] = ITEM.
c) Set LOC = NEW LEFT[NEW] = NULL
RIGHT[NEW] = NULL.
④ [Add ITEM in the tree].
If PAR = NULL then
Set ROOT = NEW
Else if ITEM < INFO[PAR] then:
Set LEFT[PAR] = NEW
Else
RIGHT[PAR] = NEW
⑤ Exit.

Delete Node from BST.

Delete Node with 0 or 1 child

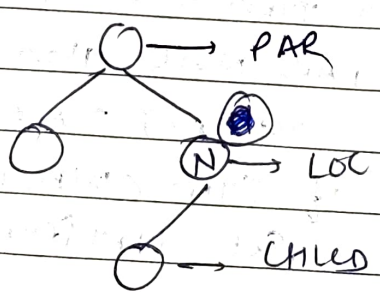
This algorithm deletes node N at location LOC where 'N' does not have two children. The PAR gives location of the parent of N. If PAR = NULL then N = Root. The CHILD gives location of the only child of N. If CHILD = NULL then N has no children

CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

(1) If LEFT[LOC] = NULL and RIGHT[LOC] = NULL then
    set CHILD = NULL.
  Else If LEFT[LOC] ≠ NULL then
    set CHILD = LEFT[LOC]
  Else
    Set CHILD = RIGHT[LOC]

(2) If PAR ≠ NULL
    If LOC = LEFT[PAR] then
      Set LEFT[PAR] = CHILD
    Else
      Set RIGHT[PAR] = CHILD
  Else
    Set ROOT = CHILD,

(3) Exit.



Delete with Node with 2 child

This algorithm deletes node 'N' at 'LOC' when 'N' has two children.
The SUC gives location of inorder successor of N, and PARSUC gives location of parent of inorder successor.

CASE B(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

(1) [Find SUC and PARSUC]
   a) Set PTR = RIGHT[LOC] and SAVE = LOC
   b) Repeat while LEFT[PTR] ≠ NULL
      Set SAVE = PTR & PTR = LEFT[PTR].
   c) Set SUC = PTR and PARSUC = SAVE.

(2) [Delete inorder successor]
   Call CASE A (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).

(3) [Replace 'N' by its inorder successor].
   a) If PAR ≠ NULL then
      If LOC = LEFT[PAR] then
         Set LEFT[PAR] = SUC
      Else
         Set RIGHT[PAR] = SUC.
   Else
      Set ROOT = SUC.
   b) Set LEFT[SUC] = LEFT[LOC] and
      RIGHT[SUC] = RIGHT[LOC]

(4) Exit

Complete Algorithm to delete a node 'N' from Binary Search Tree TS —

DEL (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

① Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

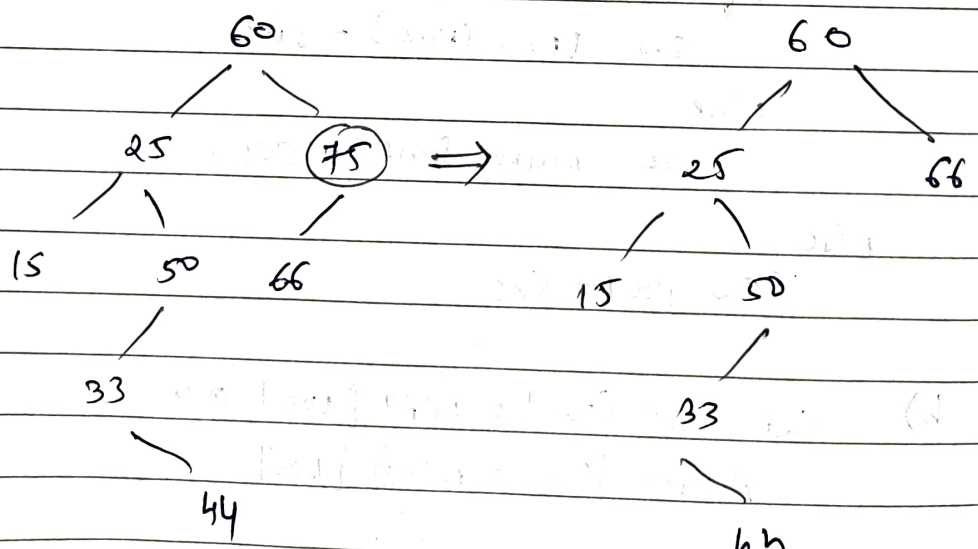
② If LOC = NULL then
write: ITEM not in tree.

③ If RIGHT [LOC] ≠ NULL and LEFT [LOC] ≠ NULL then
Call CASE3 (INFO, LEFT, ROOT, RIGHT, LOC, PAR)
Else
Call CASE2 (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

④ Set LEFT [LOC] = AVAIL and AVAIL = LOC

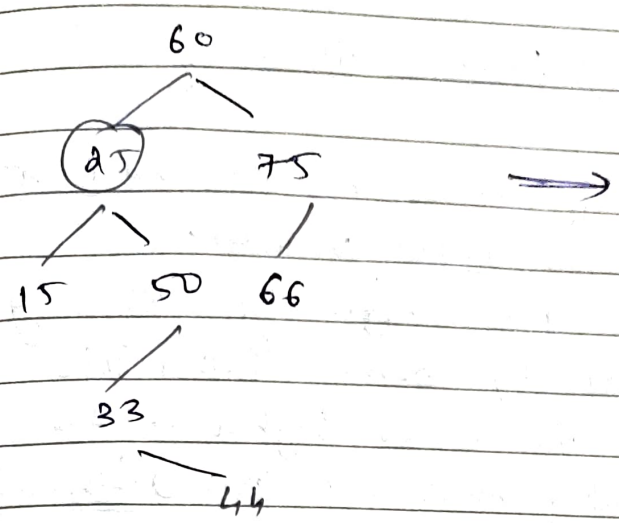
⑤ Exit.

Example

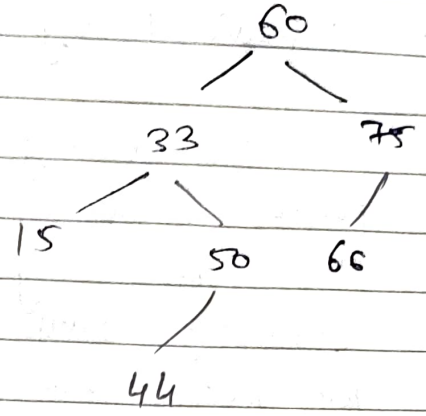


Before deletion

After deletion



Before deletion



After deletion

In Memory

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	50	2	7
4	66	0	0
5			
6			
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

	INFO	LEFT	RIGHT
1	33	8	10
2			
3	60	1	7
4	66	0	0
5			
6			
7	75	4	0
8	15	0	0
9	44	0	0
10	50	9	0

* Threaded Binary Trees / Inorder Threading.

Approximately, half entries in LEFT and RIGHT pointer fields contains NULL values. This space may be more efficiently used by replacing NULL entries by some other info. We make them point to nodes higher in the tree. This special pointers are called "threads" and binary trees with such pointers are called "threaded trees".

Threads in trees are indicated by dotted lines. We use 1 bit TAG field to distinguish threads from ordinary pointers. Alternatively, they may be denoted by negative integers when ordinary pointers are denoted by positive integers.

There are many ways of threading, each corresponds to a particular traversal of tree. We can also have one-way threading or two-way threading [Most popular is inorder threading]

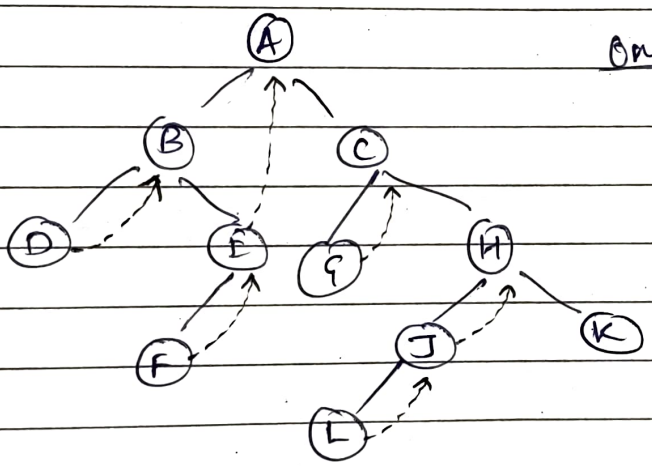
One-way threading - A thread will appear in RIGHT pointer of a node and will point to next node in inorder traversal of T.

Two-way threading - A thread will appear in LEFT and RIGHT pointer pointing to previous and next node in inorder traversal, respectively.

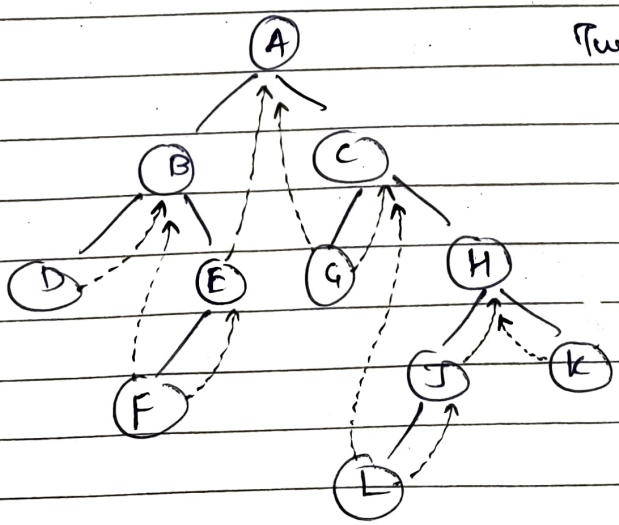
Advantages -

1) It enables linear traversal of trees.
2) Eliminates use of stack.
3) Enables finding parent node without explicit parent pointer.
4) Threaded trees gives backward and forward traversal of nodes. by inorder fashion.

```
Struct Node                    ⟹ One way threading.
{
  int data
  Struct Node *left, *right;
  bool rightThread;
};
```



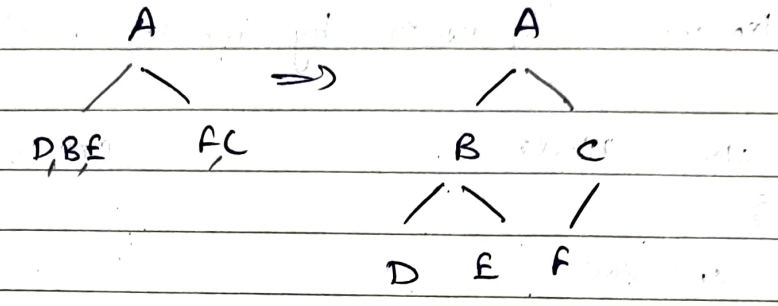
One way threading



Two way threading

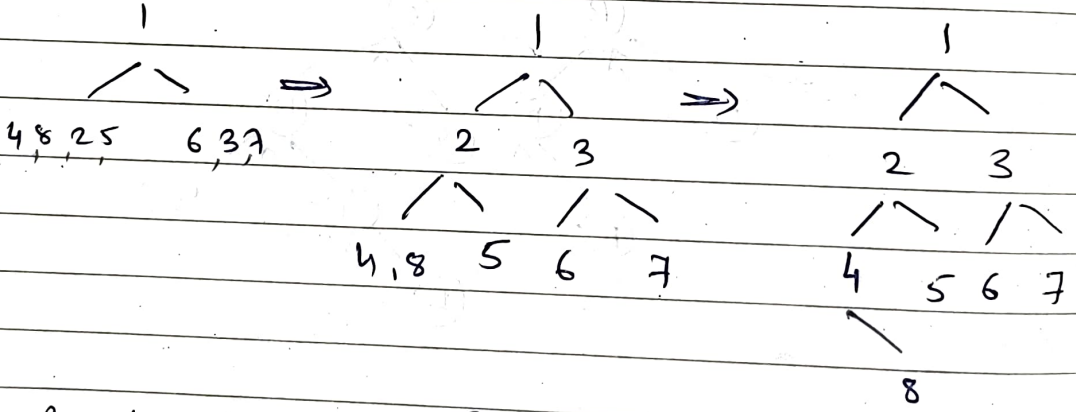
* Draw Binary Tree from preorder and inorder

Inorder :- D B E A F C
Preorder - A B D E C F



* Draw Binary Tree from postorder and inorder

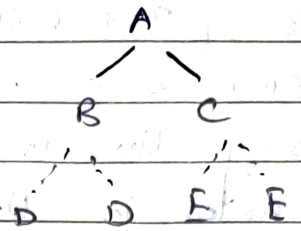
Inorder: 4 8 2 5 1 6 3 7
Postorder: 8 4 5 2 6 7 3 1



Preorder A B C
Postorder B C A
} It is not possible to construct a general Binary Tree from pre & post order

However if we know that Binary Tree is full, we can construct the tree without ambiguity.

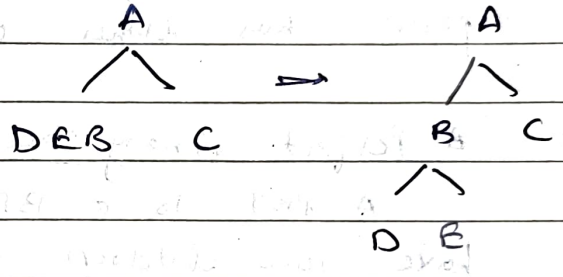
Ex:- → A B D C E
Pre → D B E C A
Post



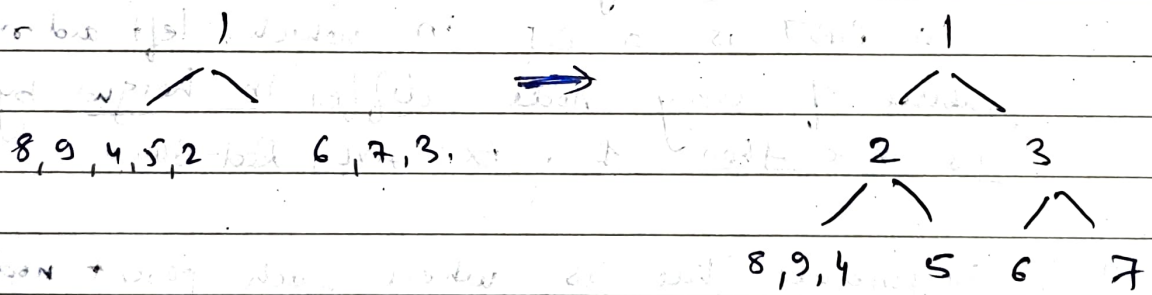
'D' can be left or right child of B [same order]
'E' can be left or right child of C [same order].

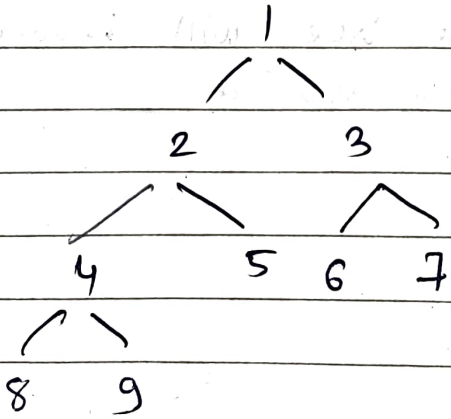
For full tree

Pre ABDEC
Post DEBCA



Pre 1 2 4 8 9 5 3 6 7
Post 8 9 4 5 2 6 7 3 1





⇒ Full Binary Tree.

* Complete Binary Tree (CBT)

A CBT is a BT in which every level except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

* Full Binary Tree (FBT)

A FBT is a BT in which all nodes except a leaf nodes have two children.

FBT is special type of BT where every parent has either 0 or 2 children.

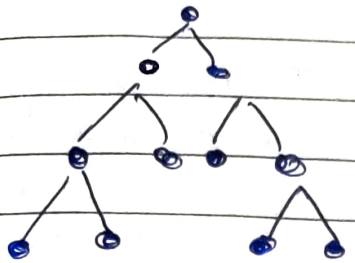
* Perfect Binary Tree (PBT)

A PBT is a BT in which all interior nodes have two children and all leaves have the same depth or same level.

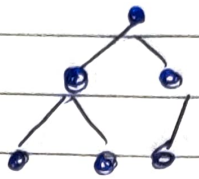
* Balanced Binary Tree (BBT) in terms of height.

A BBT is a BT in which left and right subtree of every node differ in height by no more than 1. Ex: AVL, Red-Black.

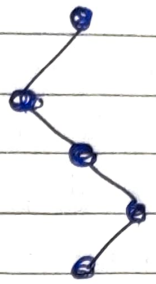
* Degenerate tree is where each parent node has only one associated child. This means that the tree will behave like a linked list data structure.



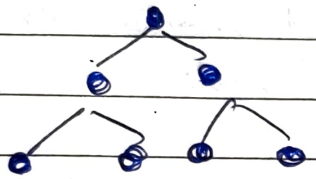
Full



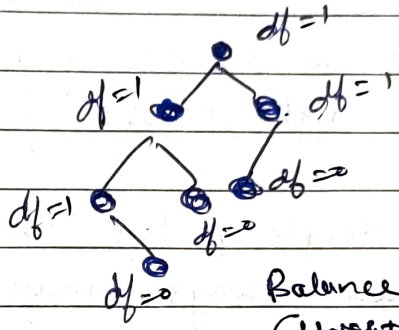
Complete



Degenerate

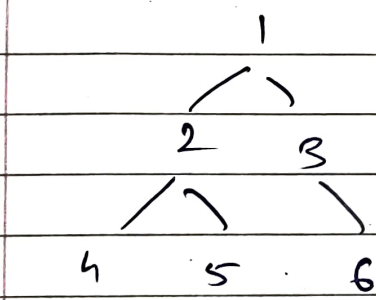


Perfect

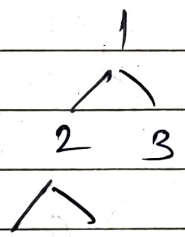


Balanced. (Height)

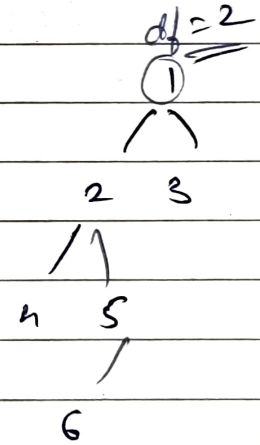
df = height of left child − height of right child.



Not full ✗
Not Complete ✗



full ✓
complete ✓



Not Balanced (Height)

Weight Balanced Tree

is a BT in which for each node the no. of nodes in left subtree is at least half and at most twice the no. of nodes in right subtree.

Ex: Huffman Tree

AVL

AVL tree is a balanced BST in which each node maintains extra information called a balanced factor whose value is either -1, 0, or 1 [Inventor - Georgy Adelson - Velsky and Landis].

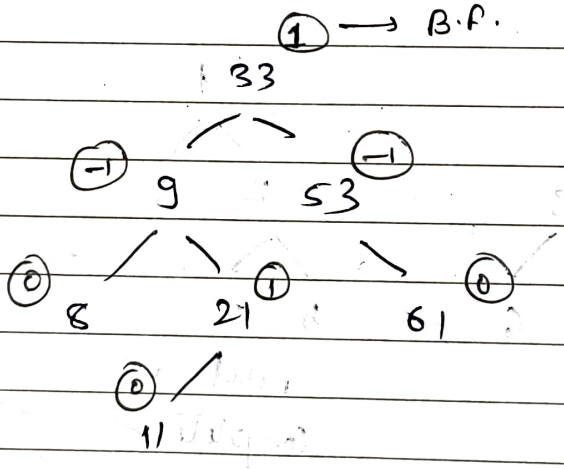
Balance factor

BF of a node in AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

BF = Height of Left subtree - Height of Right Subtree

BF value always should be -1, 0, 1.

Ex :-

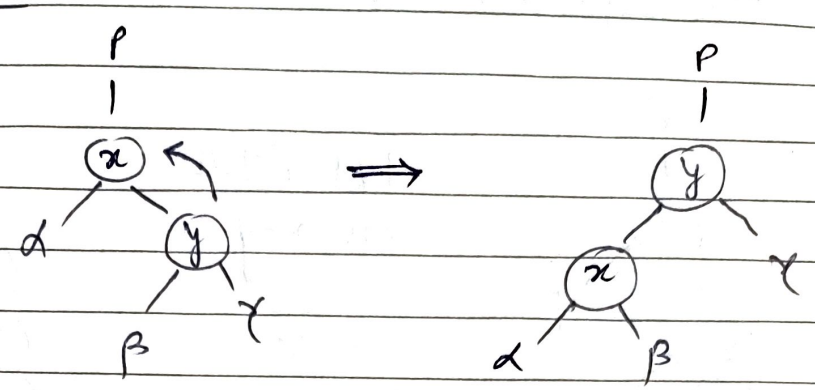


AVL tree

Operations on AVL trees

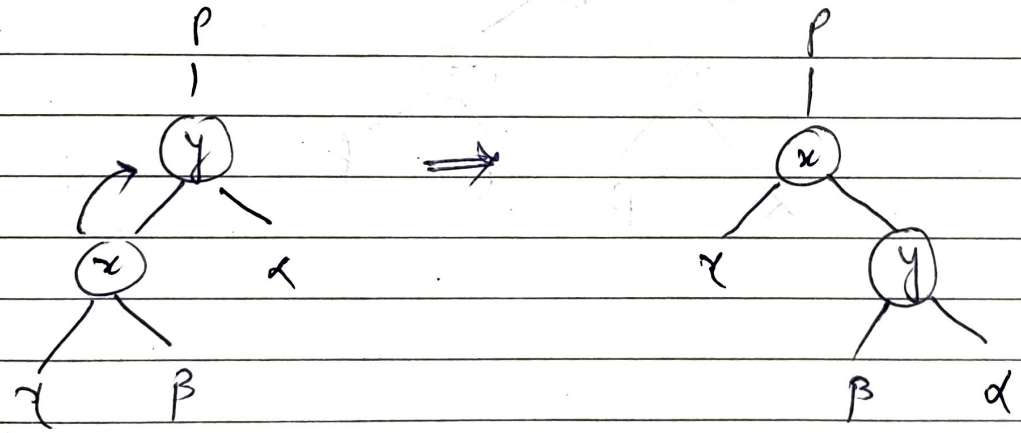
1. Rotation
   - Left Rotate
   - Right Rotate
   - Left Right Rotate
   - Right Left Rotate,
2. Insertion
3. Deletion

(1) Left Rotate



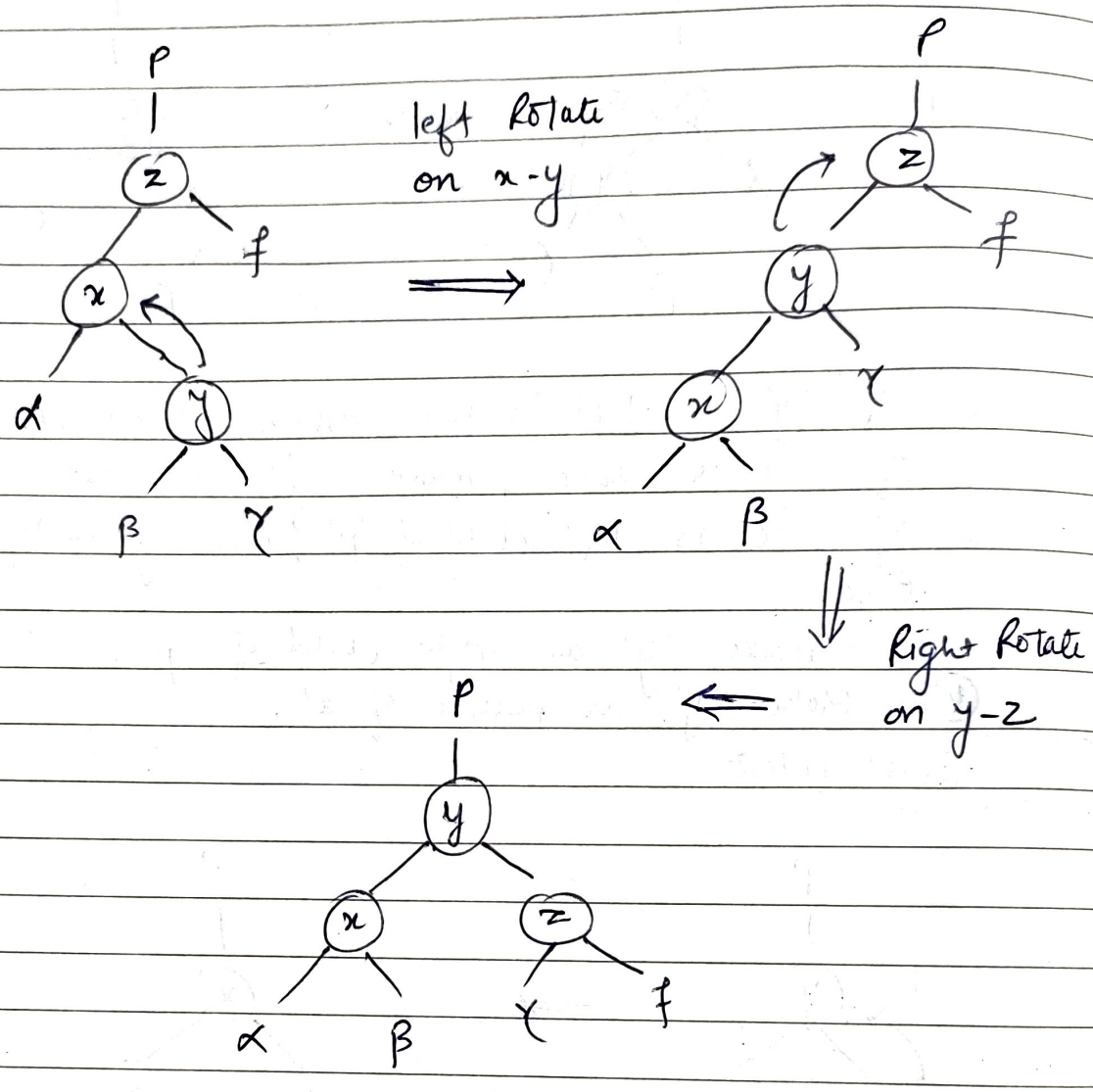
① If 'y' has left subtree, assign 'x' as its parent.
② If x is root, make 'y' as root
③ If x is left child of 'p', make 'y' as left child of 'p'.
Else
make 'y' as right child of 'p'.
④ Make 'y' as parent of 'x'.

(2) Right Rotate

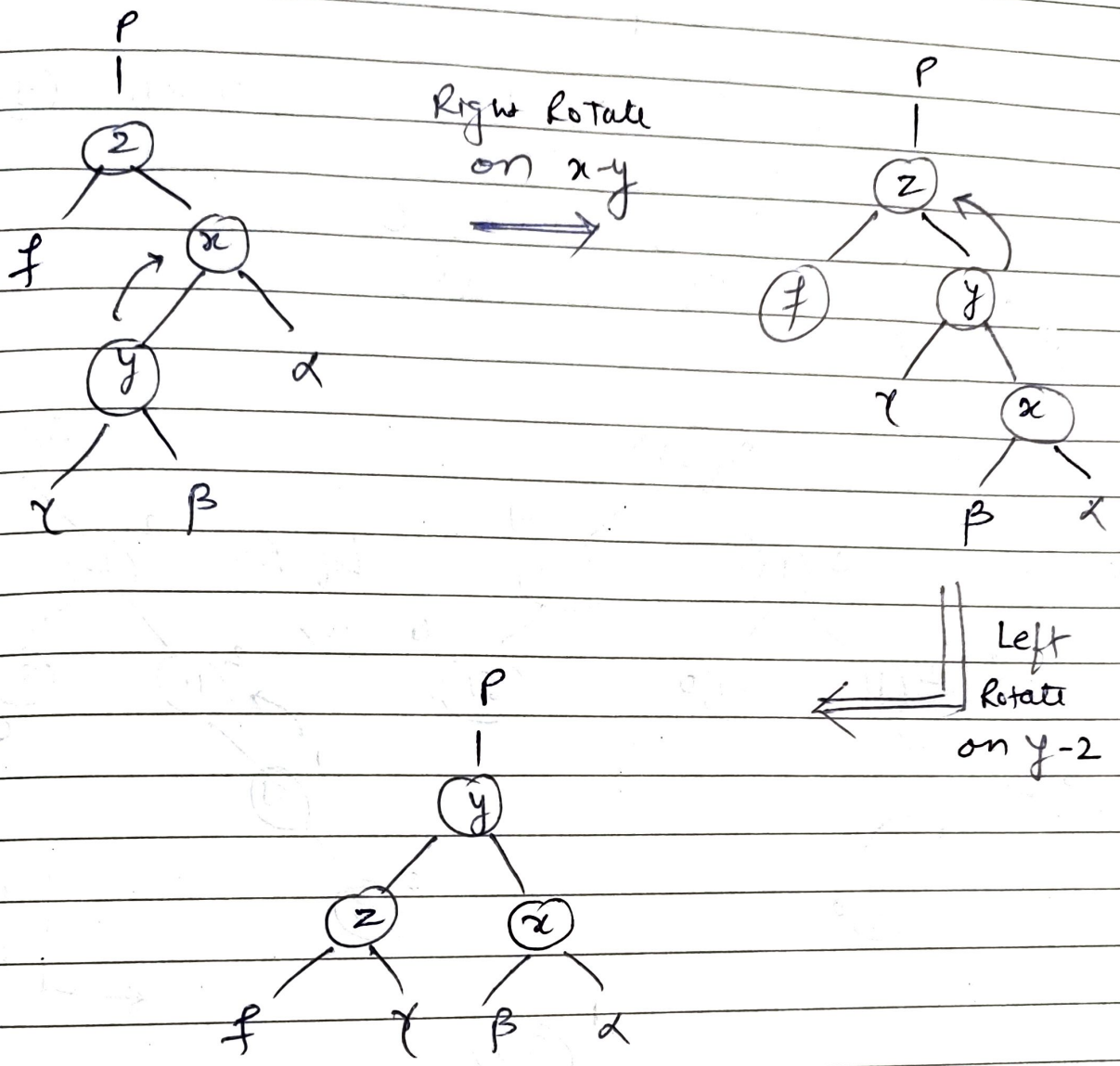


① If 'x' has right subtree, assign 'y' as its parent
② If 'y' is root, make x as root of tree.
③ If 'y' is right child of 'p', make 'x' as right child of 'p'.
Else
make 'x' as left child of 'p'.
④ Make 'x' as parent of y.

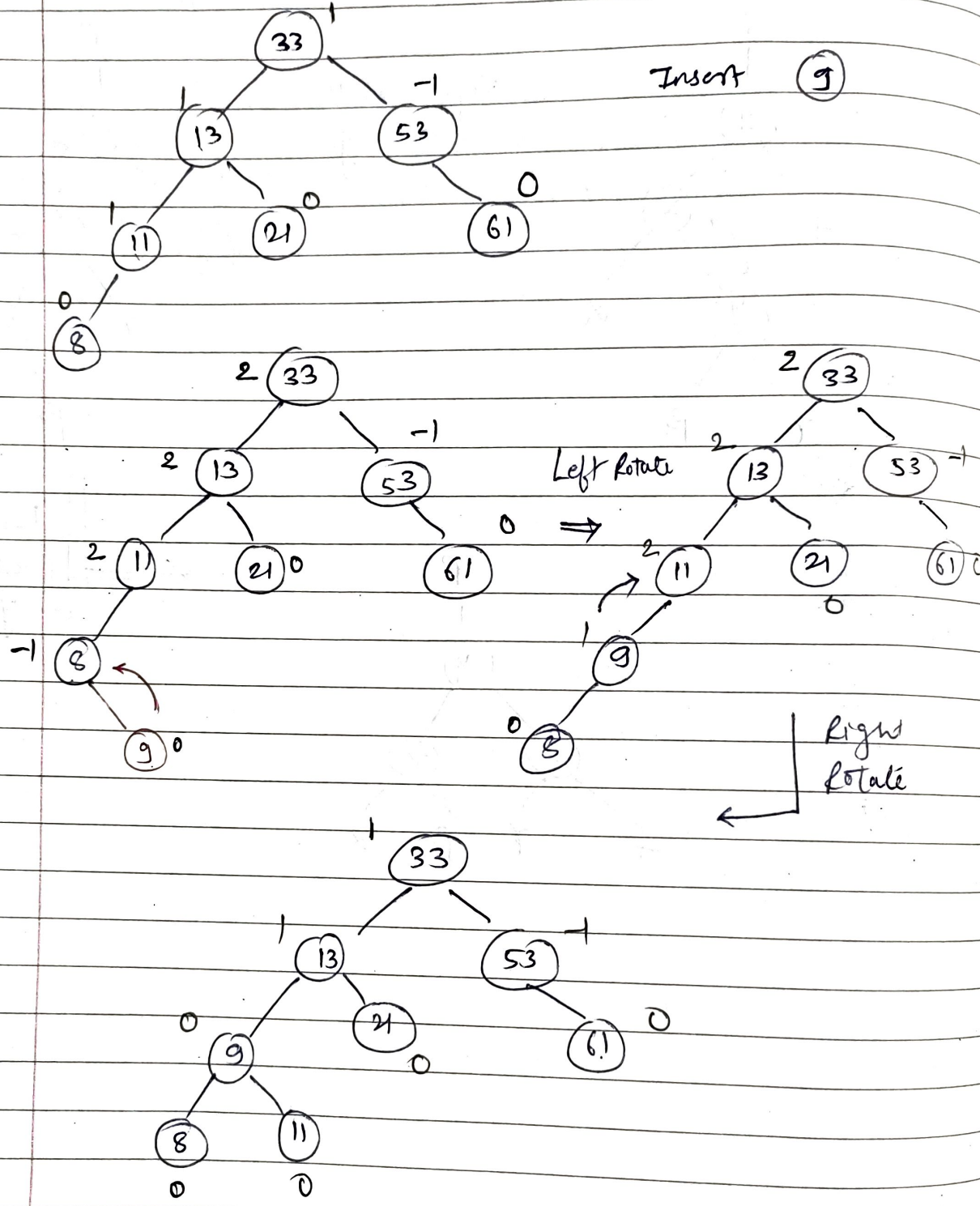
③ Left Right Rotation



(4) Right Left Rotation



Insertion in AVL



If BF > 1 then

Height of left subtree > Height of Right subtree.

So,

Do Right Rotation or Left Right Rotation.

New node is left child ⇒ Right rotation
New node is right child ⇒ Left Right rotation.

If BF < -1 then

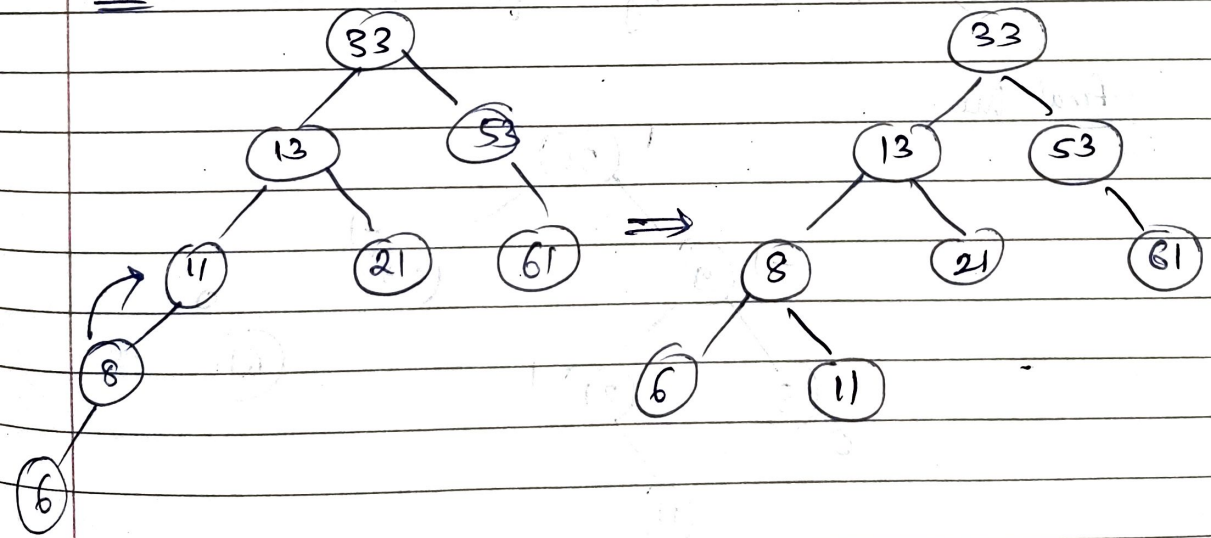
Height of left subtree < Height of Right subtree

So,

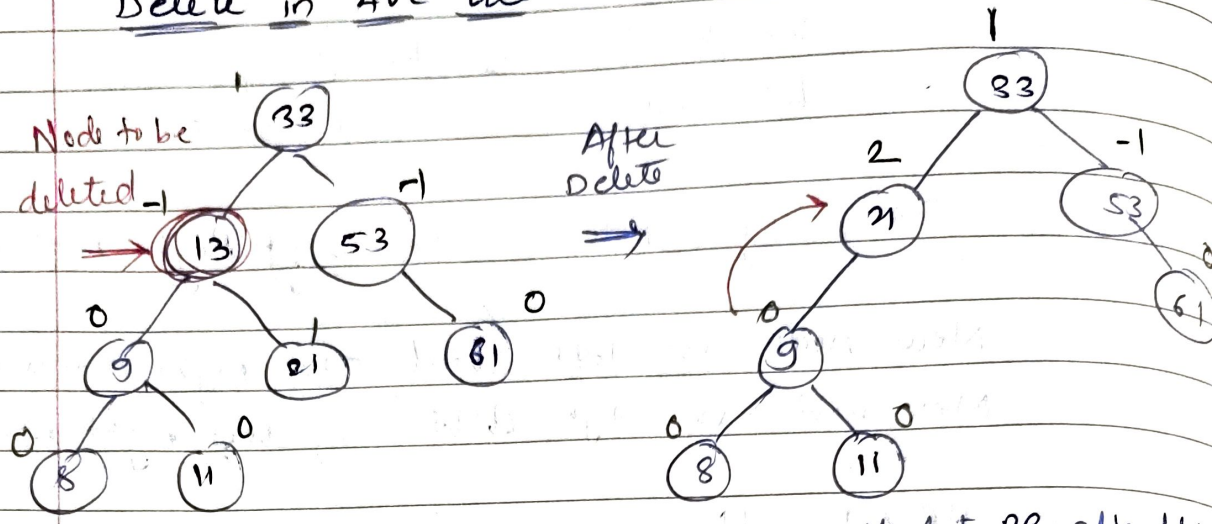
Do left Rotation or Right left rotation.

New Node is Right child ⇒ Left Rotation
New node is left child ⇒ Right Left Rotation.

Ex: Add 6 to the :



Delete in AVL tree



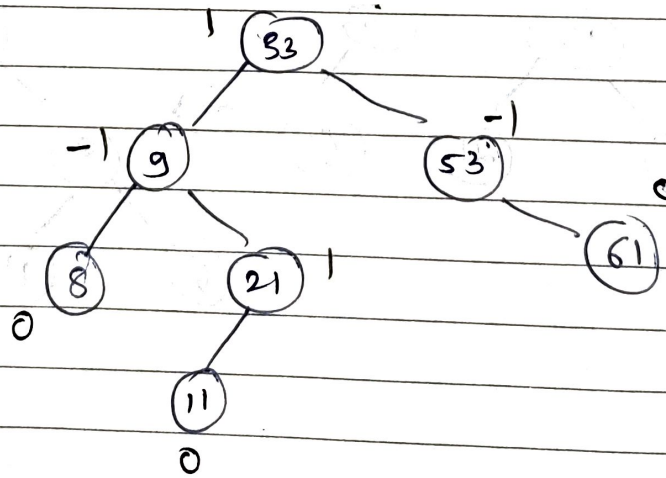
Update BF after delete

Rebalance the tree if BF of any node is not -1, 0, 1

1a) If BF of a node > 1
   a) If BF of left child >= 0 ⇒ Right rotation
   b) Else do Left-Right rotation

2). If BF of a node < -1
   a) If BF of Right child <= 0 do ⇒ Left Rotation
   b) Else do right left rotation;

Final tree



GRAPHS

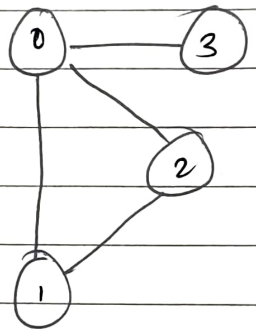
Graphs

* A Graph is a data structure that has collection of nodes containing data and these are connected to other nodes

Precisely, a graph 'G' is a D.S. (V, E) that consist of -

- set of Vertices 'V' / nodes
- set of Edges 'E', represented as ordered pairs of vertices (u, v).

Ex..



Vertices and Edges.

- Adjacency - A vertex is said to be adjacent to another vertex if there is an edge connecting them. In above graph, 2 & 3 are not adjacent.

- Path : A sequence of edges that allows to go from vertex 'A' to vertex 'B' is called a Path. Ex: 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2

- Directed Graph A graph in which edge (u, v) doesn't necessarily mean that there is ~~a~~ an edge (v, u) as well. The edges in such a graph are represented by arrows.

Graph Representation in Memory

① Adjacency Matrix

An adjacency matrix is a 2D-Array of V x V vertices. Each row and column represent a vertex.

a[i][j] = 1 represents that there is an edge connecting vertex 'i' and 'j'.

Ex:

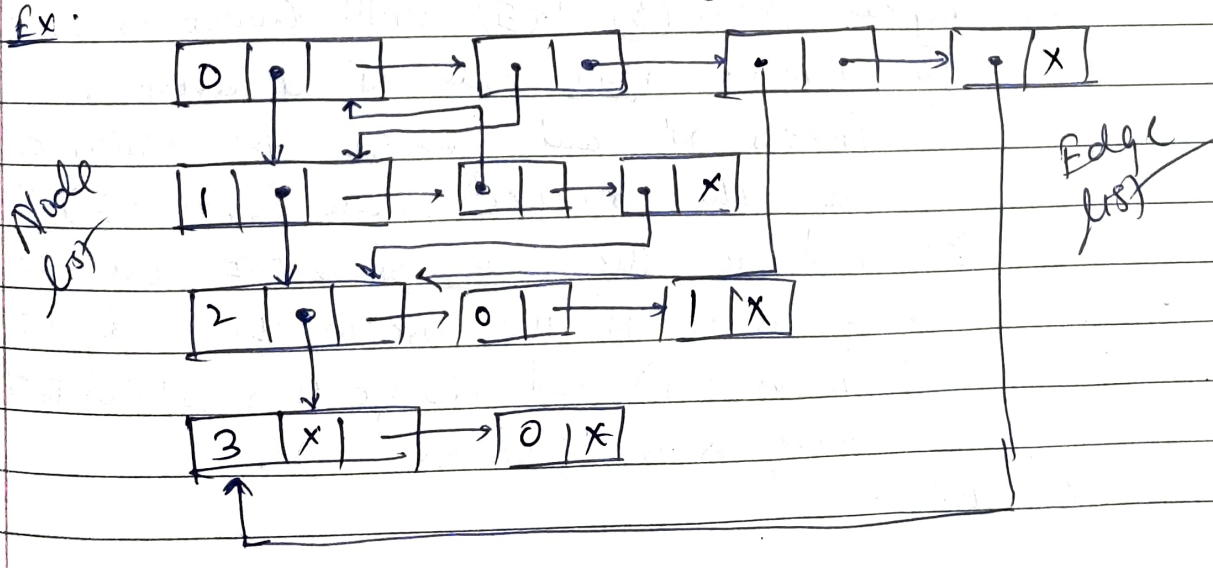
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

② Adjacency list

An adjacency list represents a graph as an array of linked list.

Index of array represents a vertex and each element in its linked list represent the other vertices that form an edge with the vertex.

Ex:



* Traversing a Graph

For traversal algorithms each node 'N' in 'G' is considered to be in one of the three states.

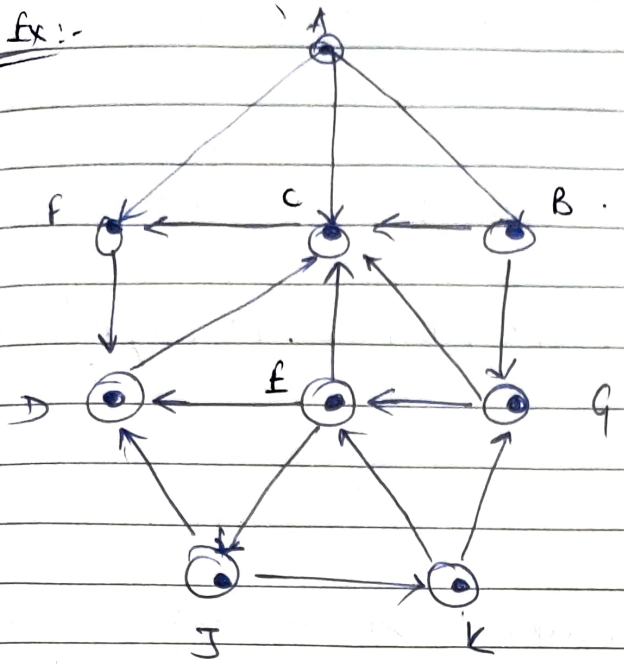
STATUS = 1 ⇒ (Ready State) Initial state of Node 'N'.
STATUS = 2 ⇒ (Waiting State) Node 'N' is on queue or stack, waiting to be processed.
STATUS = 3 ⇒ (Processed State) Node 'N' has been processed.

* Breadth First Search (BFS).

This algorithm executes BFS on 'G' beginning at a starting node 'A'.

1. Initialize all nodes to the ready state (STATUS = 1)
2. Put the starting node 'A' in QUEUE and change its state to the waiting state (STATUS = 2)
3. Repeat steps 4 & 5 until QUEUE is empty.
   4. Remove front node 'N' of QUEUE. Process 'N' and change the status of N to the processed state (STATUS = 3)
   5. Add to the rear of QUEUE all the neighbors of 'N' that are in the ready state. (STATUS = 1) and change their status to the waiting state (STATUS = 2).
6. Exit.

Ex :-



Adjacency List

A : F, C, B.
B : G, C
C : F
D : C
E : D, C, J.
F : D
G : C, E
J : D, K.
K : E, G.

(1) Add 'A' to QUEUE and SEQ = NULL

QUEUE = A
SEQ = NULL

States A B C D E F G J K
2 1 1 1 1 1 1 1

(2) QUEUE = ~~A~~, F, C, B
SEQ = A

States A B C D E F G J K
3 2 2 1 1 2 1 1 1

(3) QUEUE = ~~A~~, F, C, B, D
SEQ = A, F
F removed and D added to QUEUE.

States A B C D E F G J K
3 2 2 2 1 3 1 1 1

(4) QUEUE = ~~A~~, ~~F~~, C, B, D
SEQ = A, F, C

States A B C D E F G J K
3 2 3 2 1 3 1 1 1

(5) Q : ~~A~~, F, ~~C~~, B, D, G
SEQ : A, F, C, B

States A B C D E F G J K
3 3 3 2 1 3 2 1 1

(6) Q : ~~A~~, ~~F~~, ~~C~~, B, ~~D~~, G,
S : A, F, C, B, D

States A B C D E F G J K
3 3 3 3 1 3 2 1 1

(7) Q : ~~A~~, ~~F~~, ~~G~~, ~~B~~, ~~D~~, ~~G~~, E
S : A, F, C, B, D, G

States - A B C D E F G J K
3 3 3 3 2 3 3 1 1

(8) Q : ~~A~~, ~~F~~, ~~C~~, ~~B~~, ~~D~~, ~~G~~, ~~E~~, J
S : A, F, C, B, D, G, E

S :- A B C D E F G J K
3 3 3 3 3 3 3 2 1

(9) Q: A, F, C, B, D, G, E, J, K Status A B C D E F G J K
S: A, F, C, B, D, G, E, J 3 3 3 3 3 3 3 3 2

(10) Q: Empty
S: A, F, C, B, D, G, E, J, K Status 3 3 3 3 3 3 3 3 3

BFS - A, F, C, B, D, G, E, J, K

* **DFS:-**

(1) Initialize all nodes to ready state [STATUS = 1]
(2) Push starting node 'A' onto STACK and change its state to waiting state [STATUS = 2]
(3) Repeat (4) & (5) while STACK is not empty
(4) Pop top element 'N' from STACK, change its state to "Processed state" [STATUS = 3]
(5) Push on STACK all neighbors of 'N' that are in ready state [STATUS = 1] and change their state to waiting state [STATUS = 2].
(6) Exit

For above example

DFS:- A, B, G, E, J, K, D, C, F

* Spanning tree

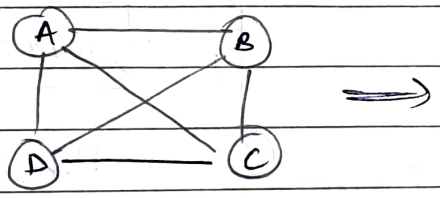
A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with minimum possible no. of edges.

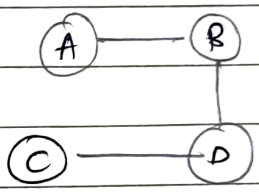
If a vertex is missed then it is not a spanning tree

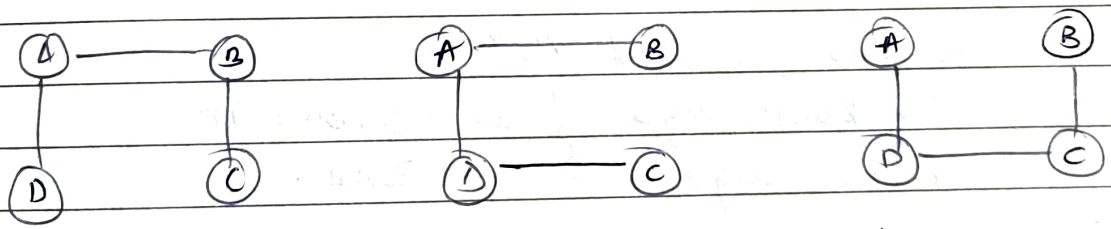
Complete Graph containing 'n' vertices may have n^{n-2} no. of spanning trees.

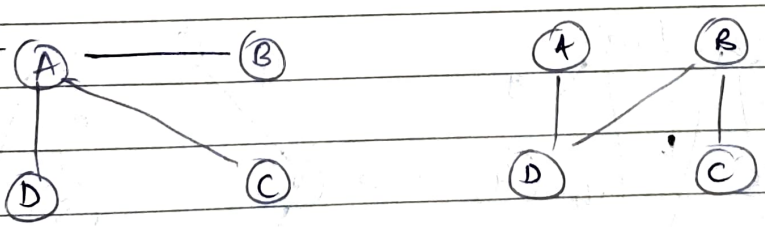
Ex :- $n=4$ max. no. of spanning trees possible is equal to $4^{4-2} = 16$. from a complete graph

Ex. of Spanning tree





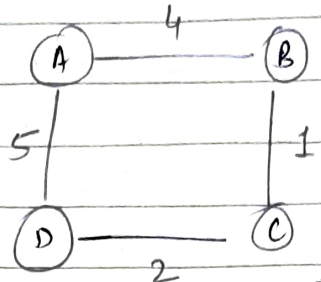




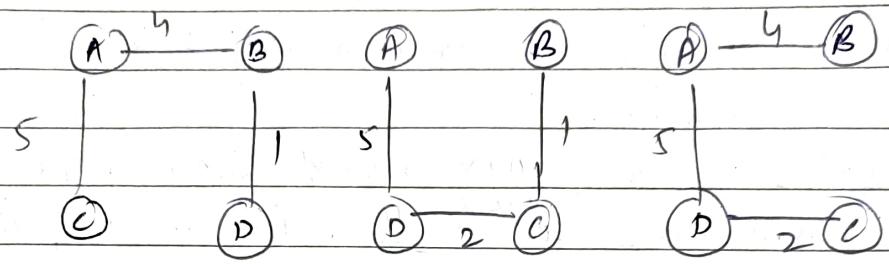
Minimum Spanning tree (MST)

A MST is a spanning tree in which the sum of the weight of edges is as minimum as possible

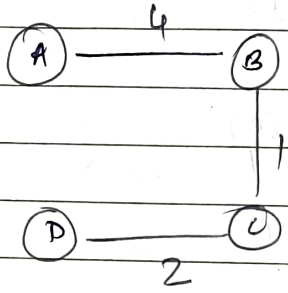
Example :-



Possible spanning trees from above graphs are -



sum = 10 sum = 8 sum = 11



sum = 7 - minimum

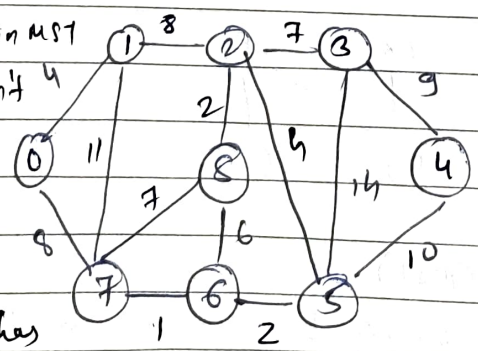
Hence, MST.

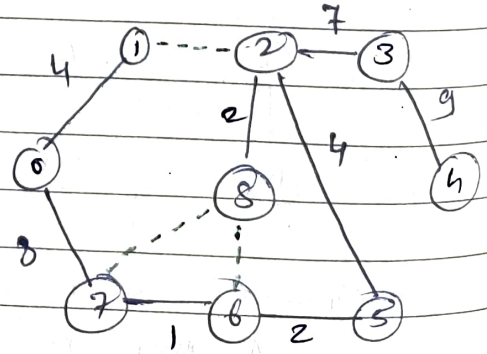
Algorithms to find MST

1. Kruskal's algo
2. Prim's algo

⇒ covered in DAA.

1) Sort edges.
2) Include in MST if it doesn't form a cycle.
3) Repeat till MST has "n-1" edges.





MST.

Given Edge ⇒ Not included as they form cycle

HASHING

Hashing

PAGE NO.: 11
10 | 11 | 2021

Hashing is a technique of mapping a large set of arbitrary data to tabular indexes using a hash function.

It is a method of representing dictionaries for large dataset.

It allows lookups, updating and retrieval operation to occur in a constant time ie. O(1)

Why it is required?

Linear and Binary search perform lookup/search with complexity of O(n) and O(log n) resp. As the size of the dataset increases, these complexities also become significantly high, which is not acceptable.

We need a technique that is independent of the size of data. Hashing allows to do this in constant time O(1).

Hash function -

A hash function is used for mapping each element of a dataset to indexes in the table.

Hash table

Hash table is the data structure that stores elements in key-value pairs -

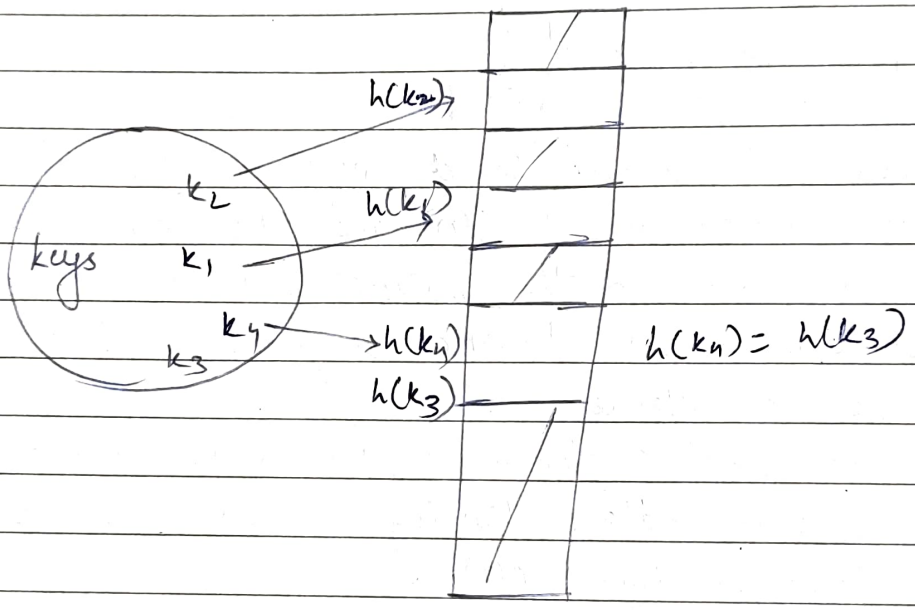
Key - unique integer that is used for indexing the values

Value - data that are associated with keys.

| key | data |

In a hash table, a new index is processed using a key. And, the element corresponding to that key is stored in the index. This is called hashing.

Let 'k' be a key and h(x) be a hash function. Here, h(k) will give us a new index to store the element linked with 'k'.



Search keys – 24, 52, 91, 67, 48, 83

Hash function -
- k mod n
- Mid square method
- Folding method

k mod 10

24 mod 10 = 4

0	
1	91
2	52
3	83
4	24
5	
6	
7	67
8	48

* Division Method.

$h(k) = k \bmod m$

m = larger than numbers

to minimize collision, m should be prime or no. with small divisor.

Ex: 10, 15, 12 if m = 5

$h(10) = 10 \bmod 5 = 0$

$h(15) = 15 \bmod 5 = 0 \Rightarrow$ collision

$h(12) = 12 \bmod 5 = 2$

* Mid-Square Method.

$h(k) = l$

key is squared (k^2).

'l' is obtained by deleting the digits from both ends.

k -	10	15	12
k^2 -	100	225	144
l -	0	2	4

* Folding Method

$h(k) = k_1 + k_2 + k_3 + \dots + k_n$

- keys are partitioned into parts.
- parts are added together.

k -	10	21	11
	1, 0	2, 1	1, 1
h(k) =	1	3	2

* Multiplication Method

1) Choose constant 'A' such that $0 < A < 1$
2) Multiply 'A' with 'key'.
3) Extract the fractional part of k·A
4) Multiply the result of above by size of hash table i.e. M.
5) Resulting hash value is obtained by taking the floor of the result in (4).

$$h(k) = \lfloor M (kA \bmod 1) \rfloor$$

Ex:- $k = 12345$

$A = 0.35784$

$M = 100$

$$h(12345) = \text{floor}\left[100\,(12345 \times 0.35784 \bmod 1)\right]$$

$$= \text{floor}\left[100\,(4417.53485 \bmod 1)\right]$$

$$= \text{floor}\left[100 \times 0.53485\right]$$

$$= \lfloor 53.48 \rfloor$$

$$\boxed{h(12345) = 53}$$

* **Collision :-**

Since a hash fuction generates small no. for a given key, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision.

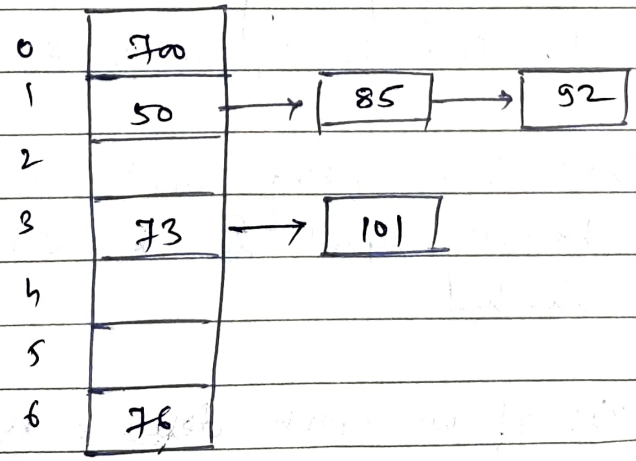
* **Collision Resolution Techniques.**

1. Separate Chaining
2. Open Addressing

(1) Separate Chaining :-

This is most popular and commonly used technique. Linked list data structure is used to implement this technique. When multiple keys are hashed into same index, then these elements are inserted into a singly-linked list (known as chain).

Ex :- $h(k) = k \bmod 7$ keys - 50, 700, 76, 85, 92, 73, 101



Advantages :-

* Easy to implement
* Hash table never fills up, We can always add more elements to chain.
* Less sensitive to hash function
* Mostly used when it is unknown how frequently keys are inserted or deleted.

Disadvantages:

* Wastage of space
* If chain becomes long, then searching can become O(n) in worst case
* Use extra spaces for links.

(d) Open Addressing.

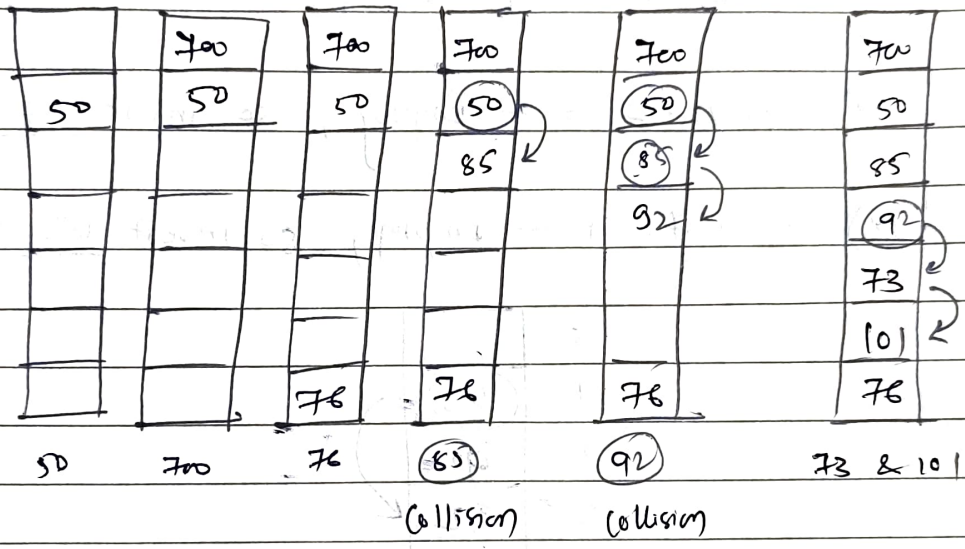
Here, all elements are stored in hash table itself. So, at any time, size of hash table must be greater than or equal to the no. of keys to be stored. This is based on probing.

- Insert (k) :- keep probing until an empty slot is found. Once found, insert k.
- Search (k) : keep probing until slot's key doesn't become equal to 'k' or an empty slot is reached.
- Delete (k): If we simply delete a key, then search may fail. So, deleted key's slots are marked as "deleted" Insert(k) can 'insert at deleted slot but search(k) doesn't stop at deleted slot.

A) Linear probing

Here, hash table is searched sequentially, that starts from the original location of hash. If location that we get is occupied, then we check for the next location.

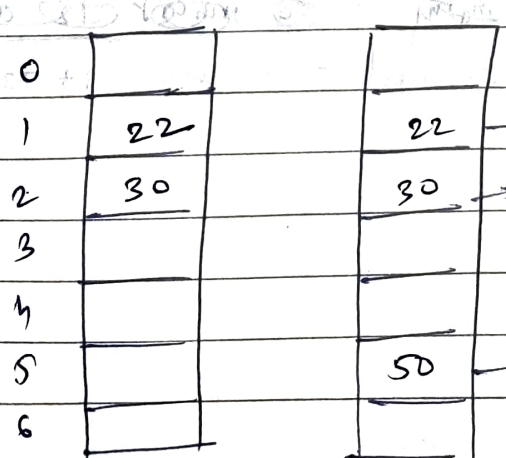
Ex: $h(k) = k \bmod 7$ keys are - 50, 700, 76, 85, 92, 73, 101.



B) Quadratic Probing

Here, we look for the $(i^{th})^2$ slot in i^{th} iteration. We always start from original hash location. If location is occupied, then we check for other slots.

Ex: $h(k) = k \bmod 7$ Insert - 22, 30, 50.



50 % 7 = 1, occupied
so search - 1+1² = 2
Again occupied, so
search 1+2² = 4, unoccupied
Hence, Insert at 4th index

c) **Double Hashing**

In this techniques, the increment for probing sequence are computed by using another hash function.

If $h(x)$ is full, we check $[h(x) + i * h_2(x)] \% m$. where 'm' is table size. $i = 1, 2, \ldots$

Ex:- Hash table size = 7

$h_1(k) = k \bmod 7$ $\quad h_2(k) = 1 + k \bmod 5$

Insert - 27, 43, 92, 74

(1) $27 \% 7 = 6$ - Empty so insert 27 in 6th slot

(2) $43 \% 7 = 1$: Empty. so insert 43 in 1st slot

(3) $92 \% 7 = 1$ - occupied so check -

$[h_1(k) + i * h_2(k)] \% 7$

$= [92 \% 7 + 1 * (1 + 92 \% 5)] \% 7$

$= [1 + 1 * (1+2)] \% 7 = 4 \% 7 = 4.$

Insert 92 at 4th slot

(4) $74 \% 7 = 4$ - occupied -

$[74 \% 7 + 1 * (74 \% 5)] \% 7$

$= [4 + 1 * (1+4)] \% 7 = 9 \% 7 = 2$

Insert 74 at 2nd slot

0	
1	43
2	74
3	
4	92
5	
6	27