Data Structure & Algorithms

What is data ?

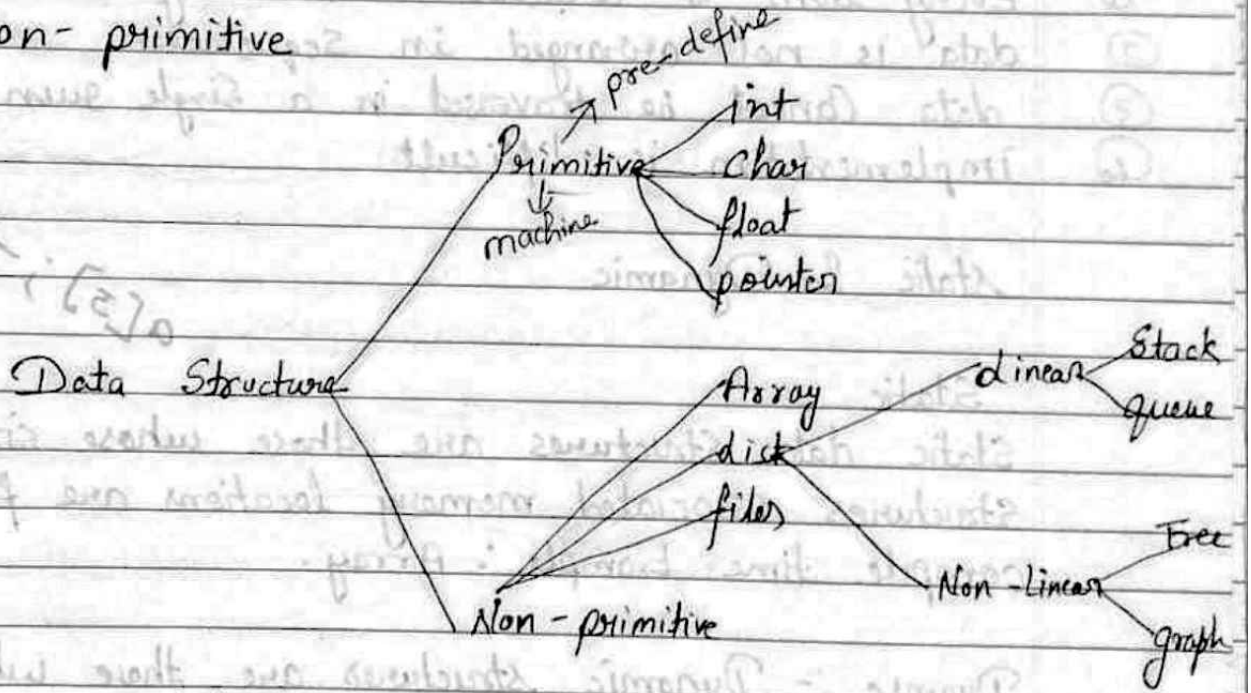
Ans Data are the raw facts, unorganized facts that need to be processed.

information → process facts, organized facts that are processed.

① Data Structure is a way of collecting & organizing data in such a way that we can perform operations on these data in an effective way.

Type :-

① primitive
② Non- primitive



Primitive data structure are the basic data structure that directly operate upon the machine instructions.

Non - primitive are more complicated data structure.

and are derived from primitive data structure. They emphasize on grouping same or different data items with relationship between each data item.

Linear & non - linear

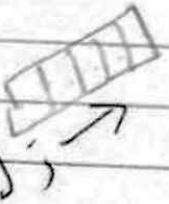
Linear D.S

① every time is related to its previous & next time
② data is arranged in linear sequence
③ data items can be traversed in a single run.
④ implementation is easy.

Non - linear D.S

① every item is attached with many others items.
② data is not arranged in sequence.
③ data cannot be traversed in a single run.
④ implementation is difficult.

Static & Dynamic



Static

Static data structures are those whose sizes & structures associated memory locations are fixed, at compile time. Example : Array.

Dyamic :- Dynamic structures are those which expands or shrinks depending upon the program need & its execution. Also, their associated memory locations changes. Example : Linked List created using pointers.

Homogeneous :- In homogeneous data structures, all the elements are of same type. Example :- Array

Non-homogeneous :- In Non-homogeneous d.s the elements may or may not be of the same type example :- Structures

Frequency count

The efficiency of program is measured by inserting a count in the algorithm in order to count the number of times the basic operation is executed. This is straightforward method of measuring the time complexity of the program.

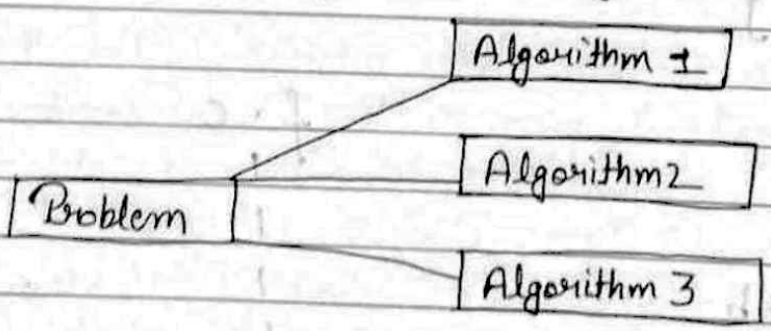
```
void display()
{
    int a, b, c;
    a = 10;  ——— (1)
    b = 20;  ——— (1)
    c = a+b; ——— (1)
    printf("%d", c); ——— (1)
}
```

Code	f : c	
a = 10	1	
b = 20	1	
c = a+b	1	O(1)
printf("%d", c)	1	
Total	4	

```
for (i=1; i<=n; i++)
{
for (j=1; j<=n; j++)
    x = x+1;
}
}
```

Code	f.c	
i = 1	1	
i <= n	n+1	
i++	n	
j = 1	n	$O(n^2)$
j <= i	n(n+1)	
j++	n * n	
x = x+1	n * n	
Total	$3n^2+4n+2$	

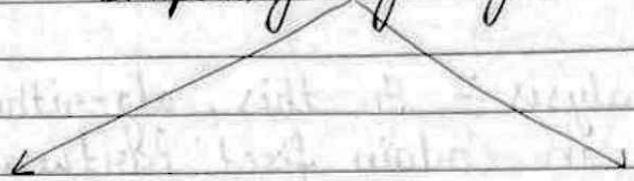
Why we analyze the algorithm



for a Given problem, there are many ways to design algorithm for it.

analysis of algorithm helps to determine which algorithm should be chosen to solve the problem.

Performance analysis of algorithm
or
Complexity of algorithm



Time Complexity

Time complexity of an algorithm is the total time required by the program to run till its completion

Space Complexity

Space complexity is the total space required by an algorithm to run till its Completion.

Time & Space complexity depends upon lots of things like hardware, OS, processor etc.

we don't consider any of these factors while we analyzing the algorithm.

we will only consider execution time of an algorithm.

Ex:- Swapping of two numbers

Using 3rd variable

```
A = 2;
B = 3;

T = A;
A = B;
B = T;
```

More Space

without using 3rd variable

```
A = 2
B = 3

A = A + B;
B = A - B;
A = A - B;
```

Less Space

Analysis

Posteriori Analysis

Priori Analysis

Posteriori Analysis :- In this, algorithm is implemented & executed on certain fixed hardware & software. Then we select which algorithm will take less time to execute.

Priori analysis :- In this, the time of algorithm is found prior to implementing it in any programming language. Here time used is not in secords or any such times units. Instead of it represents the number of operations that are carried out while executing the algorithms.

In priori analysis, we built an equation that relates number of operations (steps) that a particular algorithm does to the size of algorithm

we have to found rate of growth frm eqn

R/G

Problem

Algorithm 1 $f_1(n) = n^2 + 100n + 5$ n^2

Algorithm 2 $f_2(n) = n^3 + e^{5n}$ n^3

↑

Slow

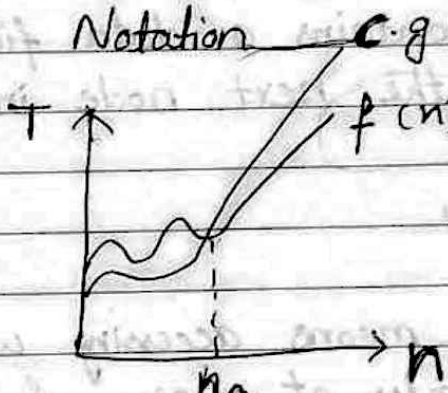
The growth rate of an algorithm is the value at which the running time (cost) of algorithm grows as the size of the input grows.

Ex. $N = 64$

2^n	n^3	n^2	$n \log_2 n$	n	$\log_2 n$
2^{64}	64^3	64^2	$64 \log_2 64$	64	$\log_2 64$

$\log_2 64$ → 6 → fastest Algo.

Big O Notation



O(n) — Worst Case

upper bound

$f(n) \leq c\,g(n)$
$n \geq n_0$
$c > 0$, $n_0 \geq 1$

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

Insertion Sort (A)

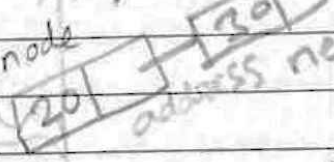
```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence
       A[1..j-1].
4      i = j-1
5      while i > 0 and A[i] > key
```

```
6    A [i+1]  = A [i]
7      i =  i+1
8    A [ i+1] = Key
```

Linked List

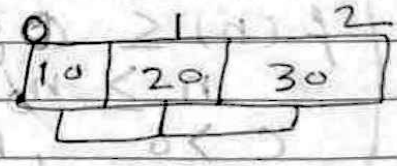
A linked list is a linear data structure, in which the elements are not stored at Contiguous memory location. The element in a linked list are linked using pointers.

In Simple words, a linked list consist of nodes where each node contains a data field & a reference (link) to the next node in the list



Traversing an array

Traversing an array means accessing with each element of array only at once, So that it can be processed



LA - Linear Array
LB - Lower Bound
UB - Upper Bound

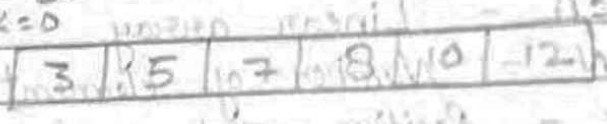
Algorithm of traversing

```
1    [Initialize Counter]
     Set K = LB
2    Repeat Step 3 & 4 while K <= UB
3      [Visit Element]
     Apply process LA [K]
```

4 [Increment Counter]
Set K := K+1 — [End of step 2 loop]
5 Exit

For Loop

1 Repeat FOR K = LB to UB
2 Apply Process to LA [K]
[End of Loop]
3 Exit



Inserting an element in array

The algorithm inserts a data element Item into the K^{th} position in an array LA with N elements

LA → Linear Array
N → Number of elements
K → Positive integer for position in Array
Such that K <= N
ITEM = Element to be inserted

1 [Initialize counter]
Set J := N
2 Repeat step 3 and 4 while J >= K
3 [Move J^{th} element forward]
Set LA [J+1] := LA [J]
4 [Decrement Counter]
J := J-1
[End of Step 2 loop]

```
5    Set LA[K] := ITEM
6    Set N := N+1        [Reset N]
7    Exit
```

Deleting an array Element

The algorithm deletes a data element Item from the K^{th} position in an array LA with N element.

LA - Linear array
N - Number of element
K - Positive Integer for position in array such the K <= N
Item - Element to be Deleted

```
1    Set Item = LA[K]
2    Repeat FOR J = K to N-1
       [Move (J+1)Th element backward]
3      Set LA[J] = LA[J+1]
       [End of Loop]
       [Reset N]
       N := N-1
4      Exit
```

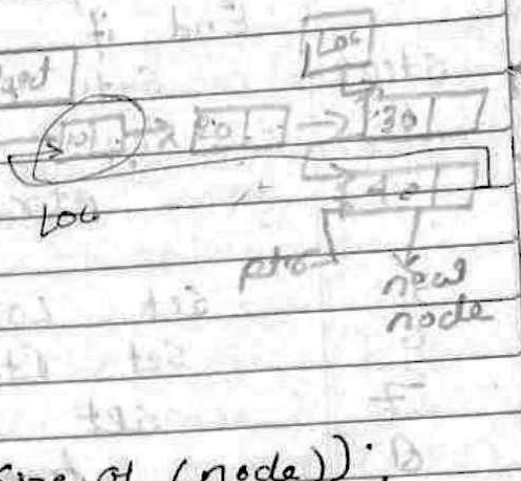
Inserting node at the begining in linked list

```
1    [Check for overflow]
     if ptr = NULL then
       print overflow
     Else
     ptr = (Node*) malloc (Size of (Node))
```

Step2: Set ptr → INFO = Item
Step3: Set ptr → Next = Start
Step4: Set ptr → Start = Ptr

At the end

Step1: Check for overflow
if ptr = NULL than
print overflow
Exit
Else
ptr = (Node*) malloc (Size of (node));



2 Set ptr → Info = Item
3 Set ptr → Next = Null
4 if Start = NULL then
Set Start = ptr
Else
5 Set LOC = Start
6 Repeat Step 7 until LOC → Next != NULL
7 Set LOC = LOC → Next
8 Set Loc → Next = ptr

Deleting the last node in Single linked

Check underflow

1 if Start = NULL than
print Linked List is Empty
Exit
2 if Start → Next = NULL then
Set ptr = start

Set Start = NULL
print element deleted is = Ptr → info
free (Ptr)
End if

Step 3 Set Ptr = Start
4 Repeat 5 & 6 untill
ptr → next != null

5 Set Loc = Ptr
6 Set Ptr = Ptr → next
7 Set Loc → Next = NULL
8. free (Ptr)

Delete the first node in Single Linked list

1 Check underflow
If start = NULL then
print linked list empty
Exit

2 Set ptr = start
3 Set Start = Start → next
4 Print Element deleted is ptr → info
5 free (ptr)

insert node at the beginning in a Circular linked list

Stack

Stack is non - primitive linear data structure

It is an ordered list in which addition of new data item & deletion of already Existing data item is done from only one end known as Top of Stack (TOS)

The last added Element will be the first to be removed from the Stack this is the reason stack is called Last in first out (LIFO)

Operation on stack

PUSH → The process of adding a new Element to the top of stack is called push operation.

Every new Element is adding to stack top is increment by 1

In case the array is full & no new Element can be added it's called stack overflow

POP → The process of deleting an element from the top of stack is called POP

After Every pop operation the Stack is decrement by 1

if there is no element on the stack & the POP is performed this will Called Stack underflow

PUSH Operation

Push (Stack [maxSize] , item)

1. intialize
   Set top = -1
2. Repeat steps 3 to 5 , Top < maxSize - 1
3. Read item
4. Set top = top + 1
5. Set Stack [Top] = item
6. Print "Stack overflow"

POP operation

Pop (Stack (maxsize) , item)

1. Repeat 2 to 4 until top ≥ 0
2. Set item = Stack [Top]
3. Set top = top - 1
4. print, no. deleted is, item
5. Print Stack under flow

Queue

Queue is a non-primitive linear data structure

It is an homogeneous collection of element in which new element are added at one end called the rear end, & the existing element are deleted from other End called the front end.

The first added Element will be the first to be removed from the queue that is the reason queue is called (FIFO) first in first out type list

In queue every insert operation rear is incremented by one

$R = R + 1$

and Every deleted operation front is increment by one

$F = F + 1$

insert operation in queue

Insert [queue [maxsize], Item]

1 Initialize
  Set front = -1
  Set rear = -1

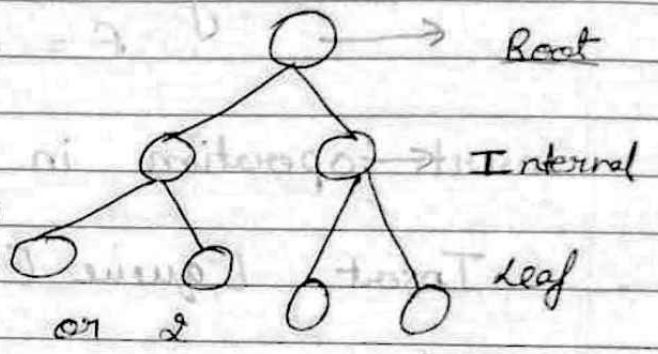
2 Repeat 3 to 5 until Rear < maxsize-1
3 Read item
4 if front = = -1 then
  front = 0
  rear = 0
5 Set queue [Rear] = item
6 Print queue is overflow

Delete an element from queue

Q Delete (queue [maxsize], item)

1 Repeat Step 2 to 4 until front >= 0
2 Set item = queue [front]
3 if front == Rear
Set front = -1
Set Rear = -1
Else
front = front + 1
4 print, No. deleted is, item
5 Print "queue is Empty or underflow

Tree



Binary Tree :- At most 2 Children

Full BT : Either 0 or 2 children

Complete BT :- No holes

Tree:

Tree are Connected the nodes through edges. Binary tree is a special data structure used for data stored ~~process~~ ~~pup~~ ~~ps~~ purpose. A binary tree has a special condition that each node can have a maximum of two children.

Path -> Path refers to the sequence of nodes along the edges of a tree.

Root -> The node at the top of the tree is

called root.

Parent → Any node except the root node has a one edge upward to a node is called parent.

child :- the node below a given node connected by its edge downward to a node called its child node.

leaf :- The node which does not have any child node is called the leaf node.

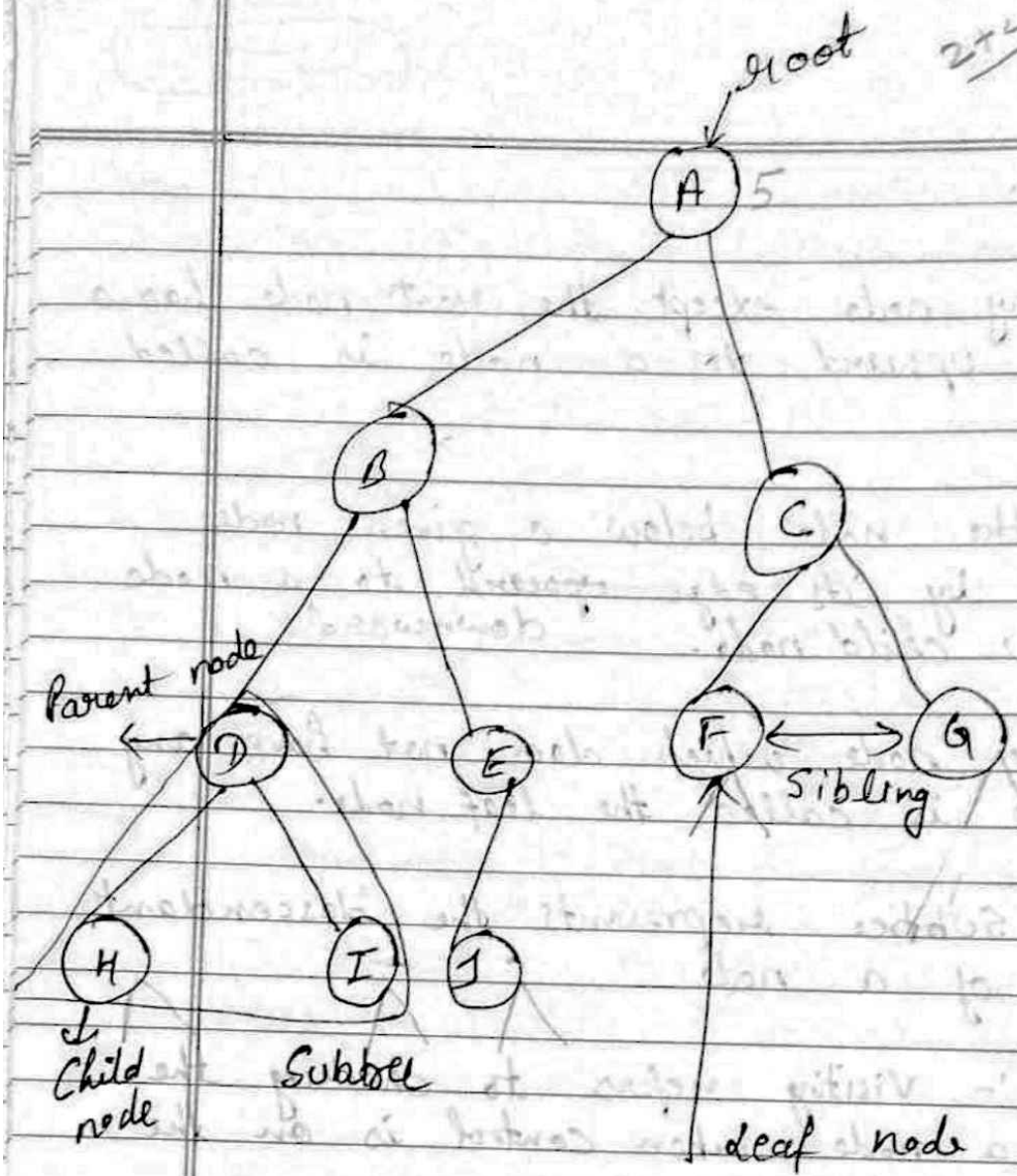
Subtree :- Subtree represents the descendants of a node

Visit Visiting :- Visiting refers to checking the value of a node when control is on the node.

Traversing :- Traversing means passing through nodes in a specific order.

Levels :→ Level represent the generation of the nodes

Key :- Key represent a value of node based on which search operation is carried out.



BST (Insertion)

The very first node insertion creates the tree. Afterwards whenever an element is to be inserted, first node locate its proper location. start searching from the root node, then if the data is less than the key value search for empty location in the left subtree & insert the data. otherwise, search for the empty location in the right subtree & insert the data

Algorithm for inserting node in BST

1 if root is NULL
then creat a root node node
return

2 if root exists then
Compare the data with node.data
while until insertion position is located

3 if data is greater than node.data
goto right subtree
else
goto left subtree
end while

4 insert data

5 end if
Ex. Exit

6

Search option

whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. otherwise search for the element in the right subtree.

Page

Algo for Searching

1 if root.data is equal to search.data
  return root
else
  while data not found

2 if data is greater than node.data
  goto right subtree
else
  goto left subtree

3 if data found
  return node
end while

4 return data not found

5 end if

6 Exit

In-order Traversal (Algo)

Until all nodes are traversed

1 Recursively traverse left subtree.
2 visit root node.
3 Recursively traverse right subtree.

Pre Order Traversal (Algo)

Until all nodes are traversed

1. Visit root node
2. Recursively traverse left subtree
3. Recursively traverse right subtree

Post order traversal

Until all nodes are traversed

1. Recursively traverse left subtree
2. Recursively traverse right subtree
3. Visit root node

Representation of binary tree in memory

① Linked representation
② Sequential representation

Linked Representation

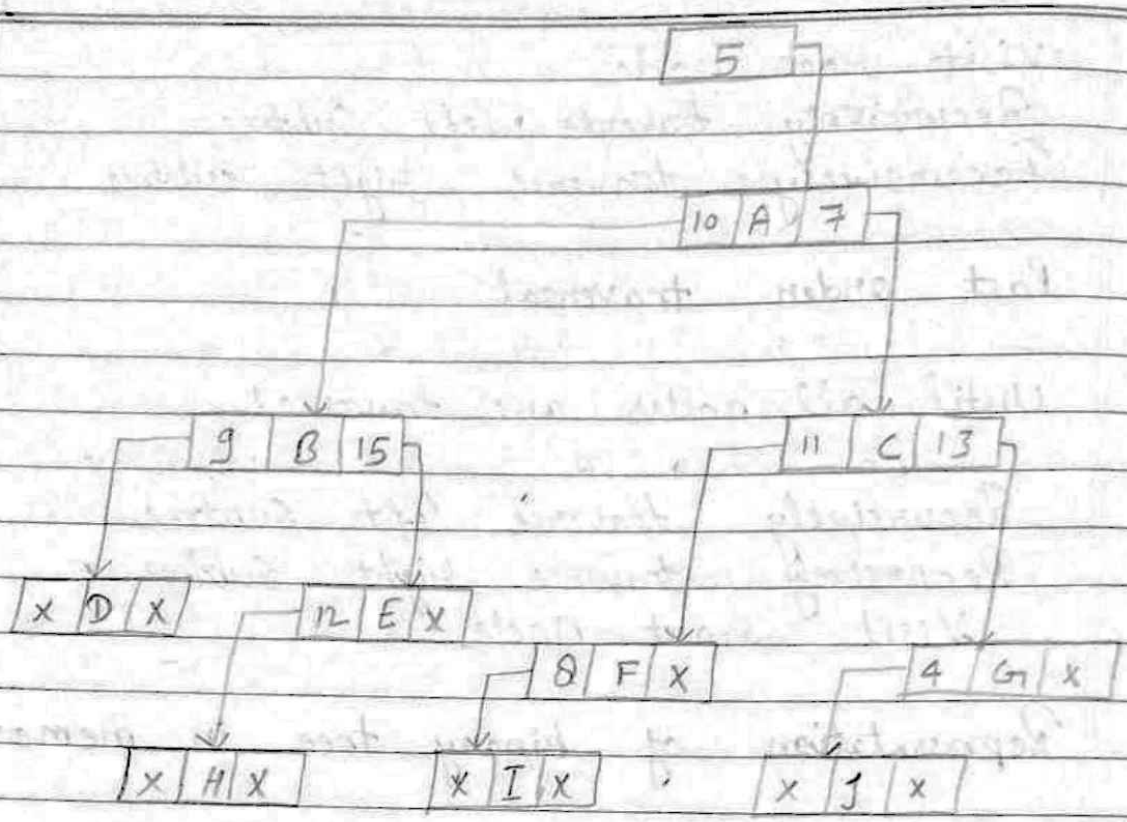
Three 11 array are used info, left, right

info [K] contains the data at Node N

Left [K] contains the locations of left child of node N

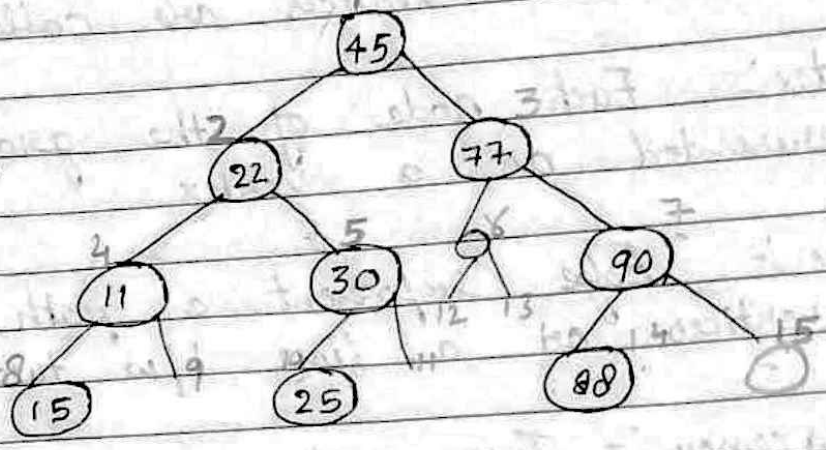
Root will contain the location of left & right Node

if any subtree is empty then corresponding pointer will contain a null pointer.

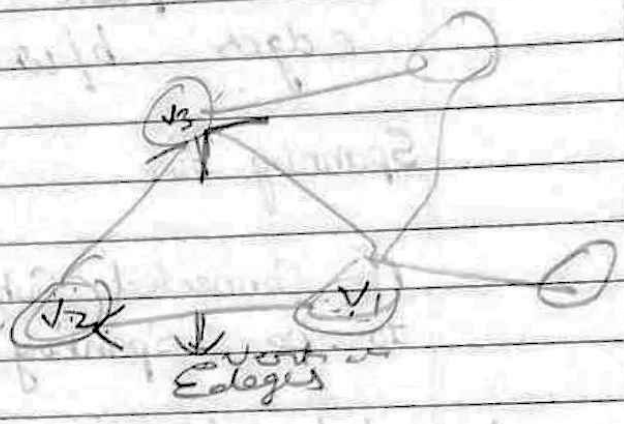


Loc	INFO	LEFT	RIGHT
1			
2			
3			
4	J	0	0
5	A	10	7
6			
7	C	11	13
8	I	0	0
9	D	0	0
10	B	9	15
11	F	8	0
12	H	0	0
13	G	4	0
14			
15	E	12	0

Sequential Representation



1	45
2	22
3	77
4	11
5	30
6	-
7	90
8	15
9	-
10	25
11	-
12	-
13	-
14	88
15	-



Graph

A Graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed

termed as vertices and the links that connected the vertices are called edges.

Vertex :- Each node of the graph is represented as a vertex

Edge :- Edge represent a path b/w two vertices or a line b/w two vertices.

Adjacency :- Two nodes or vertices are adjacent if they are connected to each other through an edge.

Path - path represent a sequence of edges b/w the two vertices.

Spanning tree

A Connected Subgraph 'S' of Graph $G(V,E)$ is said to be Spanning if

1) 'S' should Contain all vertices of G
2) 'S' should Contain $(|V|-1)$ edges

find Spining tree = n^{n-2} → Complete graph