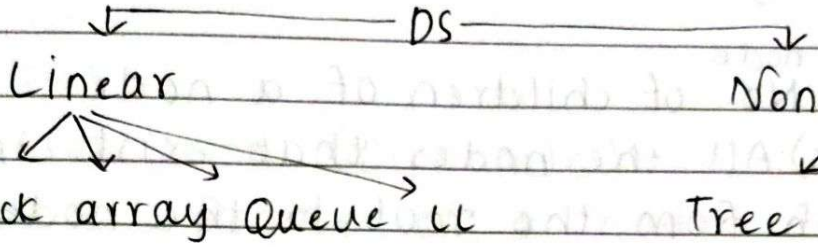
27/8/25

Module 3

Non Linear Data Structures



- Non-linear : Collection of data when relationship b/w elements are not linear

Tree : Non linear data structure in which the relationship is hierarchical.

1. Node : element / data
2. Edge : shows relationship b/w two nodes (link)
3. Child / children : B, C, D are children of A
   Immediate descendent is a child
4. Parent : Immediate accensor is a parent
   D is a parent of H and I
5. Root : A node with no parent is the root of the tree. A tree have exactly one root.
6. Leaf / leaves : A node with no child
7. Siblings(s) : Children of same parent
8. Level : Rank given to each and every element in the hierarchy.
9. Path : Sequence of nodes and edges which takes us from one node to another.

* In a tree, there is exactly a unique path from source to destination / sink.

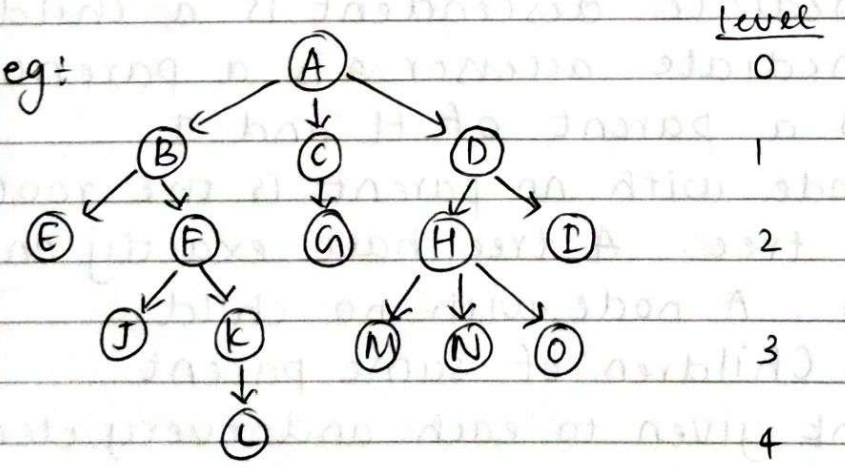
whereas in a graph, there are multiple paths to go from source to sink

* A node in tree have exactly one parent
A node in graph can have many parents.

10. Degree of a node : No. of children of a node
11. Ancestor(s): All the nodes that exists in the path from the root to the node
12. Descendant(s) : All the nodes that exists in the path from the leaf to the node
13. Height of a tree : Length of longest path from root to leaf. (h)
14. Length of a path : No. of nodes in path including source and sink. (l)
15. Depth of a tree : Length of longest path from leaf to root.

$h = l + 1$ | height = depth

eg:



8/9/25

Tree : It is a finite collection of one or more nodes such that

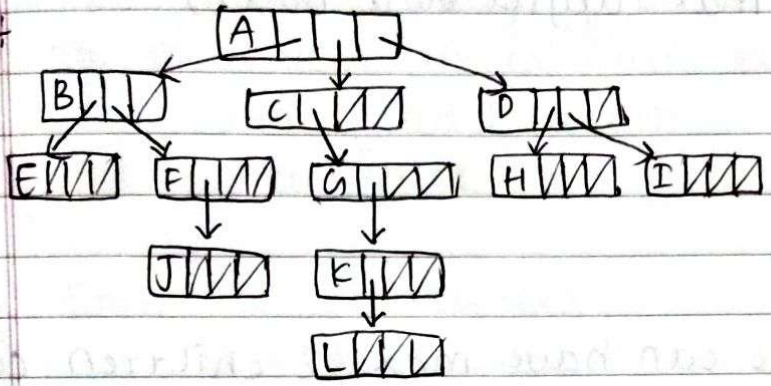
a) There is a specially designated node called root

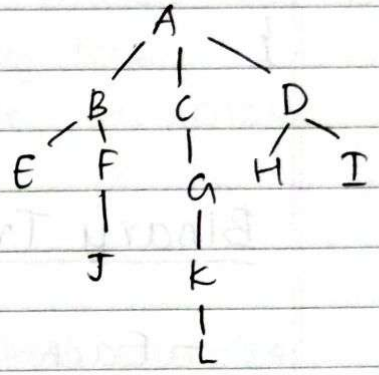
b) The remaining nodes are partitioned into n ($n>0$) disjoint sets $t_1, t_2 \ldots t_n$ where each t_i ($i = 1, 2, 3 \ldots n$) is a tree
$t_1, t_2 \ldots t_n$ are called subtrees of the root.

Representation (Dynamic)

1. Rep I : Degree of a node

eg:



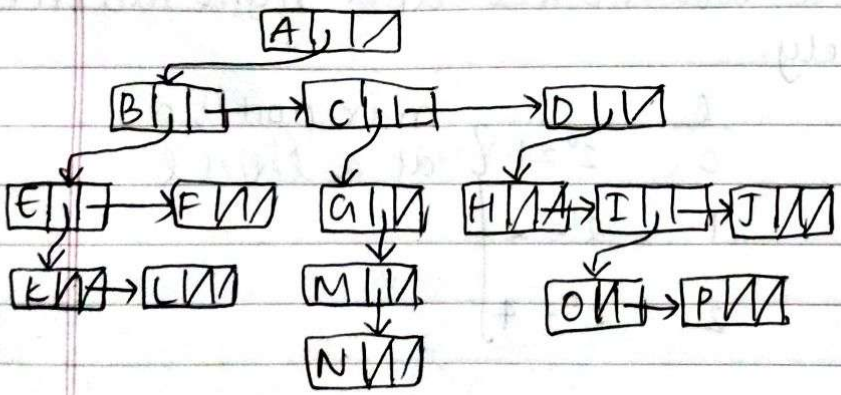


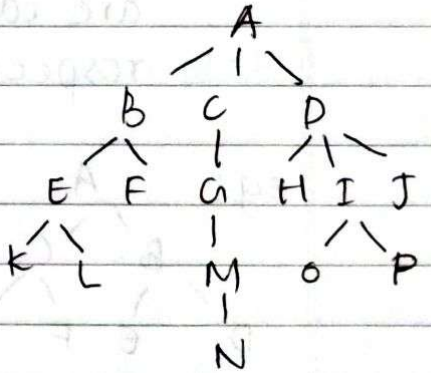
(max degree : 3)

- The disadvantage is when ~~a~~ ~~sing~~ one of the nodes has a higher degree as compared to other nodes.

10/9/25

2. Rep II : Lefmost child, right sibling





Rep II

```
struct node{
    char data;
    struct node *lc, *rs;
}
```

data	lc	rs

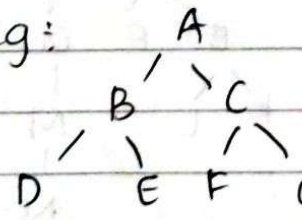
Rep I

```
struct node{
    char data;
    struct node *c1, *c2, ... *cn; (where n is
                max degree of a node)
}
```

Binary Tree

- Each node can have max 2 children can be ~~as~~ left child or right child
- Binary tree is a finite collection of nodes (T) such that
  a) T is empty (called the empty binary tree)
  b) or T contains a specially designated node called root and remaining nodes of ~~tree~~ T forms 2 disjoint sets T_1 and T_2 which are called left sub tree and right sub tree respectively.

eg:



l		
0	$2^0 = 1$	max nodes n at a level l
1	$2^1 = 2$	
2	$2^2 = 4$	

→

Types of Binary Tree

1. Skewed Binary tree

When ~~the~~ every nodes of the tree would have only one child, either entirely on the left (left-skewed) or on right (right-skewed)

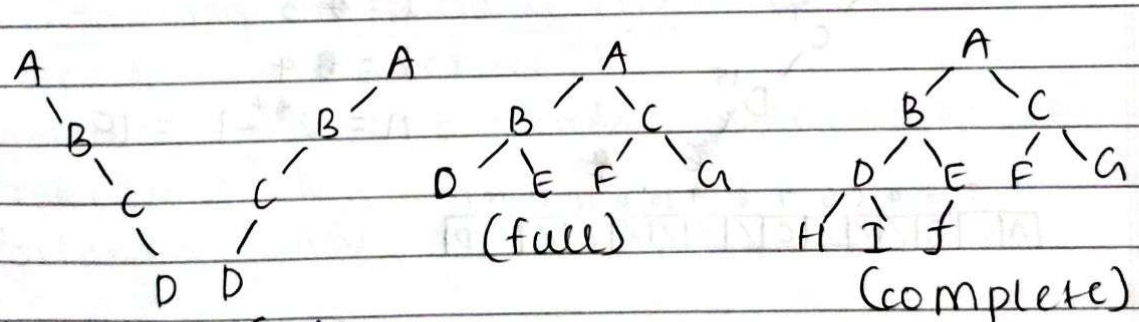
2. Full Binary Tree

- When every level have maximum (2^l) no. of nodes it is called full binary tree.
- Cannot add another node ~~it~~ unless add a new level.

3. Complete binary tree

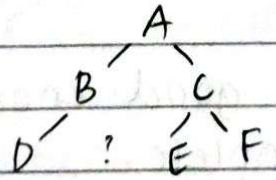
- It is full in all levels ~~but can be~~ except the last level. If last level is not full, any element present must be as far left as possible

Eg:



(right skewed) (left skewed) (full) (complete)

But:



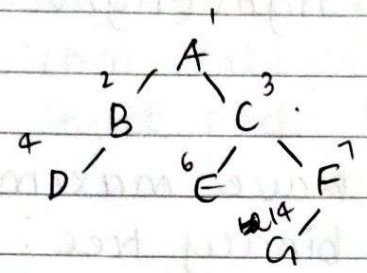
(not complete)

Representation of Binary Tree

Sequential (static/array) | List (dynamic / LL)

1. Sequential

eg:

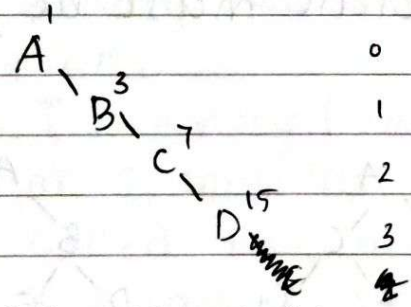


max level = 3 (l)
height = l+1 = 4
max no. of nodes, $n = 2^h - 1$
$= 2^4 - 1 = 15$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	/	E	F	/	/	/	/	/	/	G	/

→ If parent is at index i:
left child is at index 2i
right " " " " 2i+1

eg:



l = 3
h = 4
$n = 2^4 - 1 = 15$

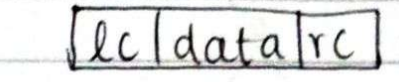
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	/	B	/	/	/	C	/	/	/	/	/	/	/	D

→ If child is at index i:
parent is at index $\lfloor i/2 \rfloor$
eg: If child at 7, parent at $\lfloor 7/2 \rfloor = 3$

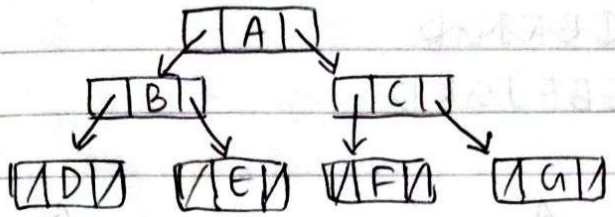
⇒ Sequential representation is good enough when binary tree is full or complete.

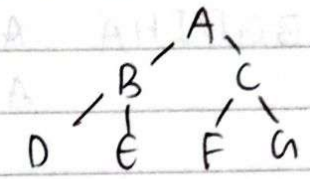
2. List

```
struct node{
    char data;
    struct node *lc, *rc;
}
```



eg:





→ No need to know the size in advance

→ Disadvantage : must store memory location to lc and rc along with data
eg: (in above eg) : 3 x 7 = 21 units of memory used.

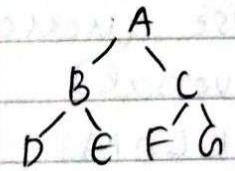
Traversal of Binary Tree (use stack)

(depth wise traversal)

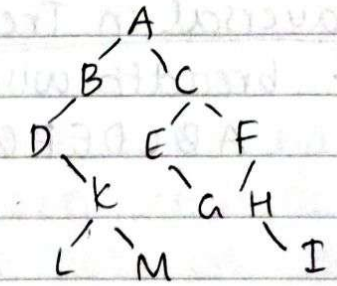
1. Inorder L(N)R (left subtree, node, right subtree)
2. Preorder (N)LR
3. Postorder LR(N)

1. Inorder LNR

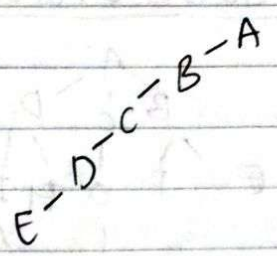
eg:



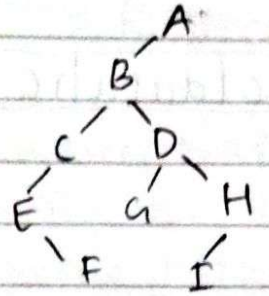
DBEAFCG

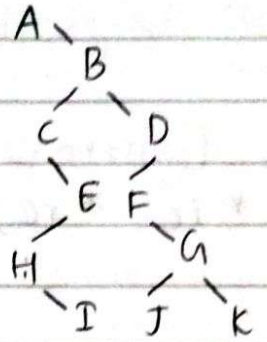


DLKMBAEGCHIF



EDCBA



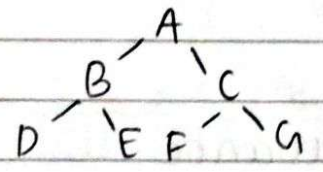


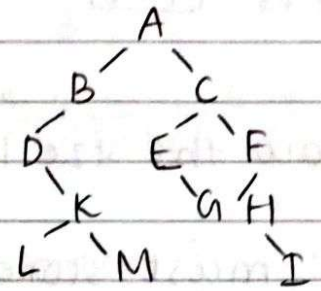
EFCBGDIHA

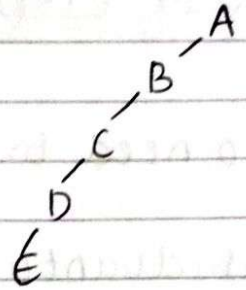
~~ACEHIBFJGK~~

ACHIEBFJGKD

eg:-

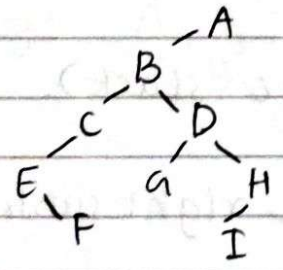


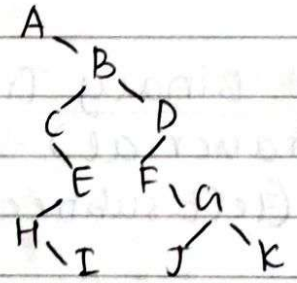




Preorder: ABDECFG ABDKLMCEGFHI ABCDE

Postorder: DEBFGCA LMKDBGEIHFCA EDCBA



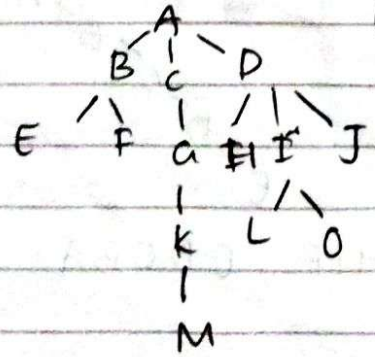


Preorder: ABCEFDGHI ABCEHIDFGJK

Postorder: FECGIHDBA IHECJKGFDBA

Level Order Traversal in Tree (use queue)

(· breadth wise traversal)



ABCDEFGHIJKLOM

11/9

Recursive Algorithm for Traversal

1.
```
Inorder (node *p){
  (p != NULL){
      inorder (p→lc);
      print (p→data);
      inorder (p→rc);
} return();
}
```

2.
```
Preorder (node *p){
   (p != NULL){
        preord print (p→data);
        preorder (p→lc);
        preorder (p→rc);
} return();
}
```

3.
```
postorder (node *p){
     (p != NULL){
          postorder (p→lc);
          postorder (p→rc);
          print (p→data);
} return();
}
```

⇒ In sequential representation, if pointer child is given, it is possible to find the parent but not in case of linked list

Binary Search Tree (BST)

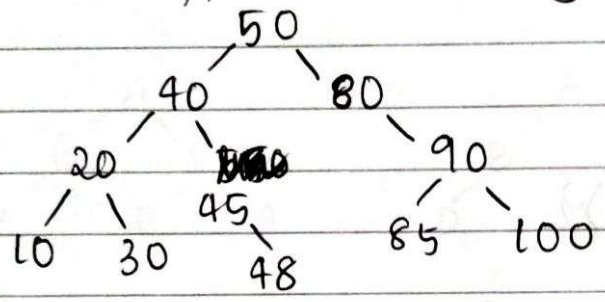
Binary search tree have the foll. properties:

a) All the values of of the left subtree is less than the node value

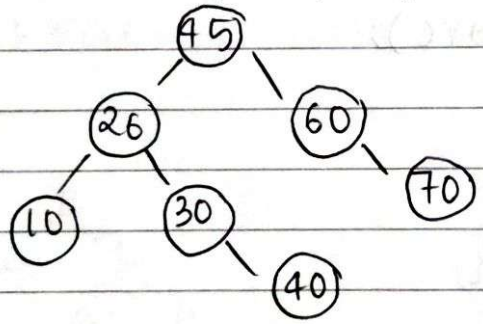
b) All the values of the right subtree is greater than the node value

here, it applies for every node of the tree

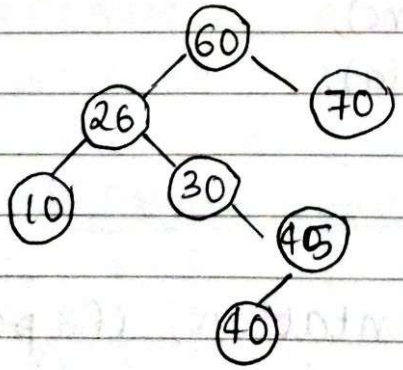
eg:



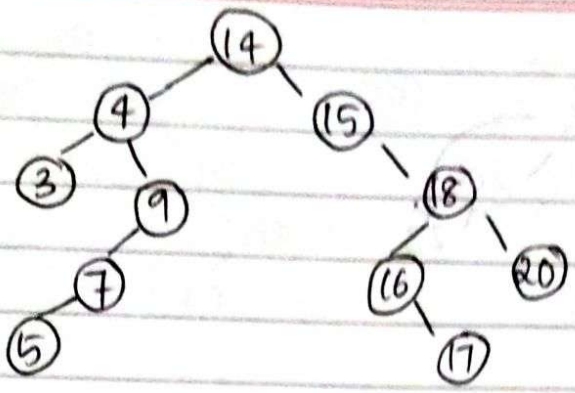
Q. 1) Insert : 45, 26, 10, 60, 70, 30, 40 into a BST.



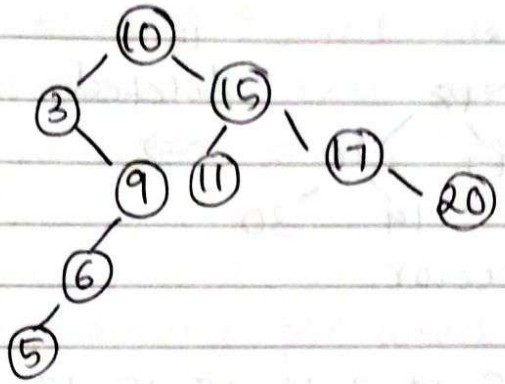
2) 60, 26, 70, 30, 10, 45, 40



3) 14, 15, 4, 9, 7, 18, 3, 5, 16, [illegible], 20, 17, [illegible]



4) 10, 15, 3, 17, 20, 9, 11, 6, 5

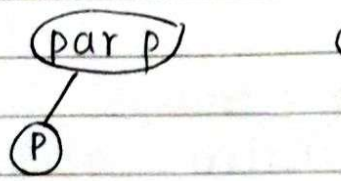


12/9

Deletion in BST

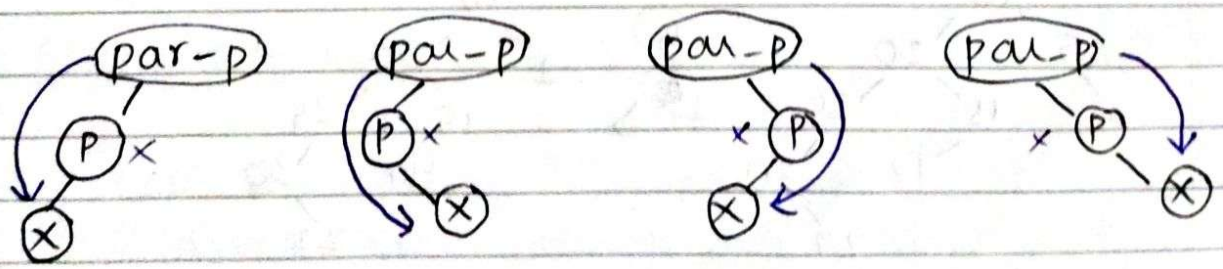
case I : node to be deleted is a leaf node
case II : " " " " has a single child
case III : " " " " has 2 children

I

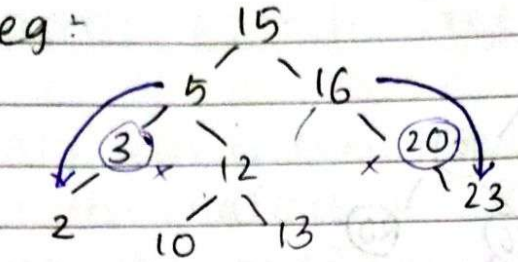


: can just delete

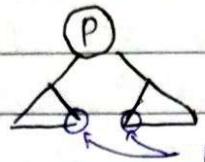
II



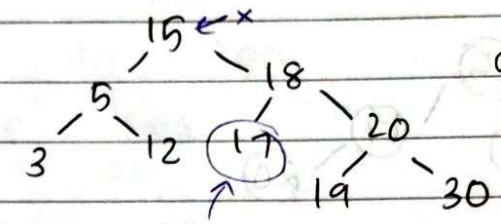
eg :-



III

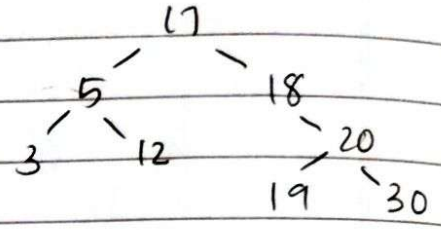


Inorder successor



inorder successor

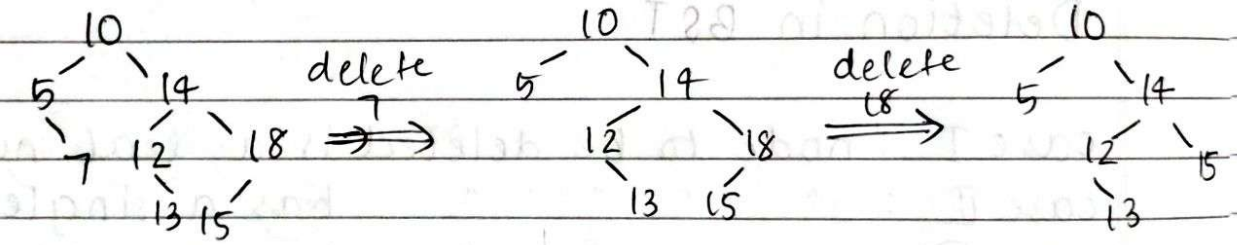
15 deleted ⇒

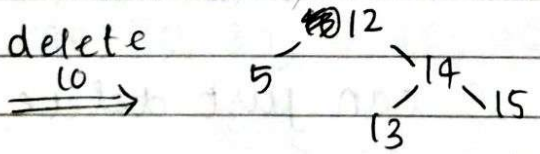


Q.5) Insert : 10, 5, 14, 7, 12, 18, 15, 13

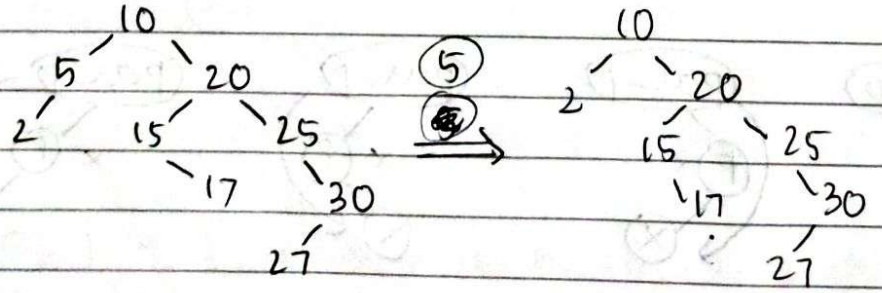
Delete : 7, 18, 10

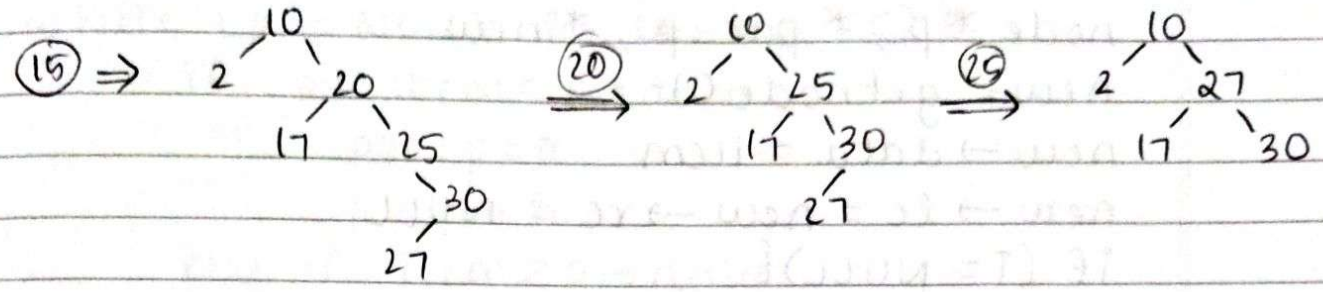
⇒





6) Delete : 5, 15, 20, 25





Algorithm to search_list BST

```
search_bst(node * T, int key)
    //search whether key exists in BST with
      root pointed to by T //

found = 0
p = T  // p is temperory ptr used to search

while (p != NULL) && (found = 0){
    if (key < p→data)
          p = p→lc
    else if (key > p→data)
          p = p→rc
    else
          found = 1
}
if (found = 0) printf "key not found"
else print "Element found".
```

Algorithm to Insert_BST

```
insert_bst(node *T, int item){
   // insert 'item' into a bst whose root node is
      ptd to' by T //
```

```
node *p, *par_p, *new
new = getnode()
new → data = item
new → lc = new → rc = NULL
if (T = NULL){
        T = new
        return()
}
p = T , par_p = NULL
while(p != NULL)
if item < p → data
        par_p = p
        p = p → lc
else if item > p → data
        par_p = p
        p = p → rc
else print "Element already exists", return
}
if (item < par_p → data)
        par_p → lc = new
else
        par_p → rc = new
return
}
```

Algorithm for deleting node

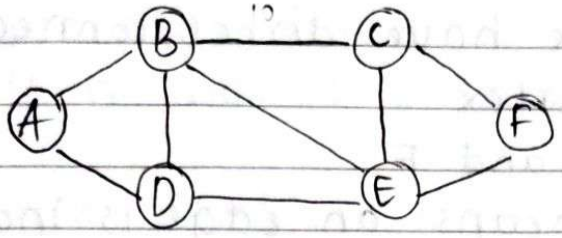
```
delete-BST (node *T, int item){
  // delete 'item' from bst pointed to by T

node *p = T , par_p = NULL
int found = 0, case
```

```
while (p != NULL) && (bound = 0) {
    if item < p→data
        par_p = p
        p = p→lc
    else if item > p→data
        par_p = p
        p = p→rc
    else bound = 1
}
if (p == NULL)
    printf("item not found, deletion not possible");
    return ();

// p points to the node to be deleted //
decide the case of deletion
if p→lc = NULL && p→rc = NULL
    case = 1
else if p→lc != NULL && p→rc != NULL
    case = 3
else
    case = 2

if (case = 1){   // delete a leaf node
    if (p == par_p→lc)
        par_p→lc = NULL
    else
        par_p→rc = NULL
    free(p);
}
if (case = 2){
    if (p = par_p→lc){
        if (p→lc = NULL)
```

```
                par_p->lc = p->rc
        else
                par_p->lc = p->lc
        }
    if (p=par_p->rc){
        if (p->lc = NULL)
                par_p->rc = p->rc
        else
                par_p->rc = p->lc
    }} free(p)
  }}
    if (case=3){
        ptr1 = inorder_succ(p)
        temp = ptr1->data
        delete_BST (T, temp)
        p->data = temp
    }
node* inorder_succ (p){
    node *ptr1
    if (p->rc != NULL)
        ptr1 = p->rc
    while (ptr1->lc != NULL)
        ptr1 = ptr1->lc
    return ptr1
}
```

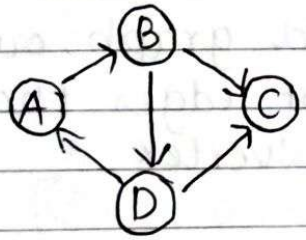
17/9/25

Graphs



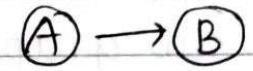
Undirected graph
$G = \{V, E\}$

- Graph is the collection of vertices and edges
ie, $G = (V, E)$
where, V: finite set of vertices
$V = \{A, B, C, D, E, F\}$
E: finite set of edges
$E = \{AB, AD, BD, BC, BE, DE, CE, CF, EF\}$



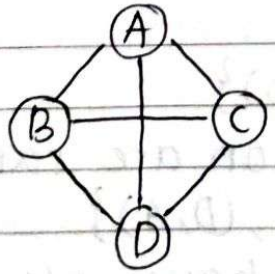
Directed graph, $G_1 = \{V_1, E_1\}$
$V_1 = \{A, B, C, D\}$
$E_1 = \{\langle A,B\rangle, \langle B,C\rangle, \langle D,C\rangle, \langle B,D\rangle, \langle D,A\rangle\}$

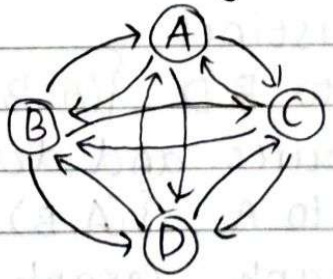
→ Head : end point
→ Tail : starting point



→ Weighted : graph in which the edge is labelled.
→ Complete : graph in which there is direct connection b/w every vertices

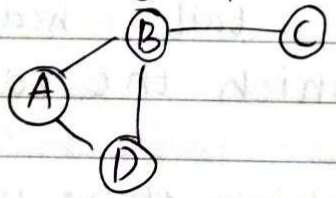
eg:





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

→ Path
eg: A to F : {(A,B), (B,E), (E,F)}
→ Adjacent : a vertex have direct connection to another vertex
eg: A and B, C and F
→ Incident : (A,B) means an edge is incident b/w A and B (A one end & B another)
→ Degree of a vertex : no. of edges which is incident on the vertex
eg: degree of A = 2
→ Indegree : in case of directed graph, indegree of a vertex is the no. of edges that comes to the vertex
eg: Indegree of A = 1
→ Outdegree : In case of directed graph, outdegree of a vertex is the no. of edges that comes ~~from~~ out of the vertex
eg: Outdegree of A = 1
→ Sub graph : Set of vertices and edges of sub graph is subset of original graph.
eg:



→ Simple Path : source and destination are distinct
eg: A to D {(A,B), (B,D)}
→ Cycle : source and destination are same
eg: A to A {(A,B), (B,D), (D,A)}
→ Cyclic graph : Graph that have atleast a cycle is cyclic graph
Non-cyclic graph : doesnot have a single cyclic path.

Representation of Graphs

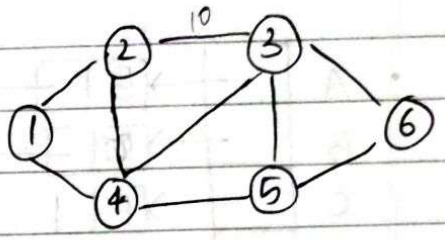
1. Adjacenty matrix
   - If n : no. of vertices, matrix will be "n×n"

Matrix $A[i,j] = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$

eg:
- Undirected graph

A	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	1	1	0	0
3	0	1	0	1	1	1
4	1	1	1	0	1	0
5	0	0	1	1	0	0
6	0	0	1	0	1	0

6×6

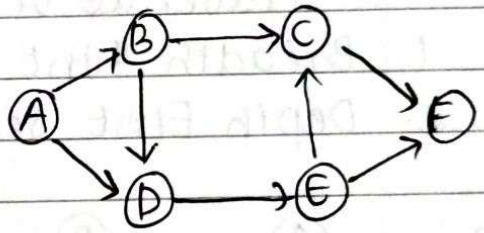


Symmetric

- Directed graph

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	1	0	0
C	0	0	0	0	0	1
D	0	0	0	0	1	0
E	0	0	1	0	0	1
F	0	0	0	0	0	0

6×6

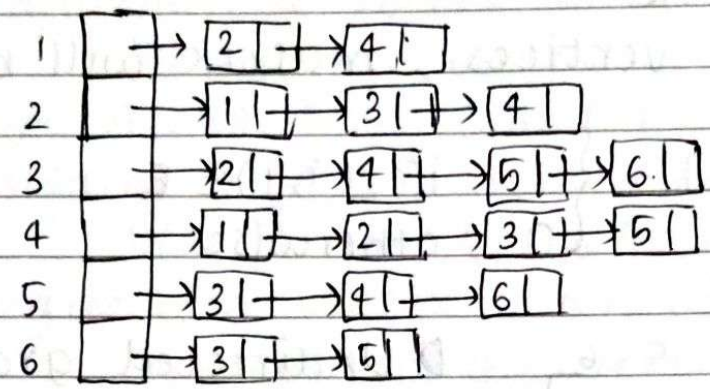


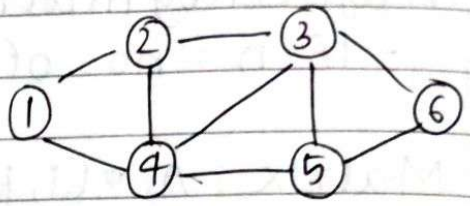
Unsymmetric

- If there is weight, the value will be stored in place of 1.

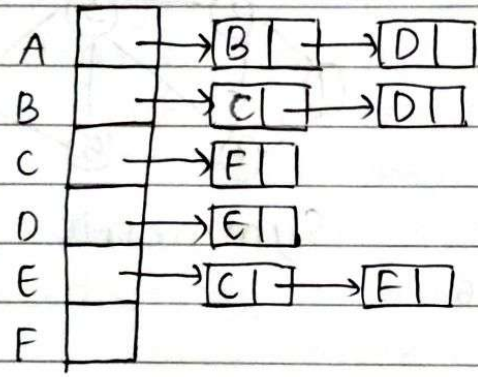
2. Adjacency List

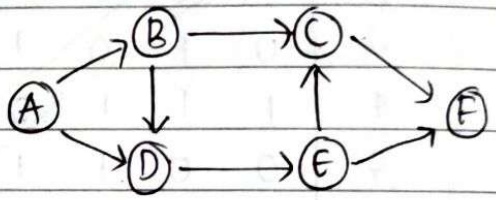
eg •





•



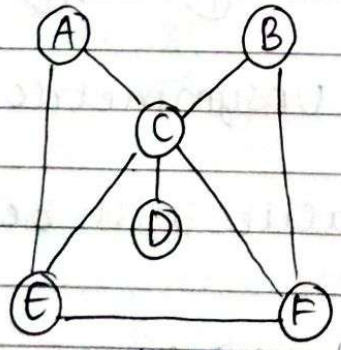


18/9

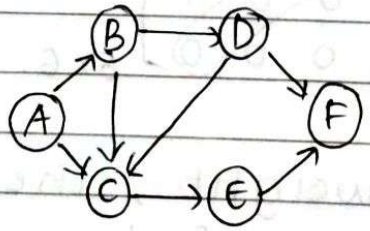
Traversal of Graphs

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

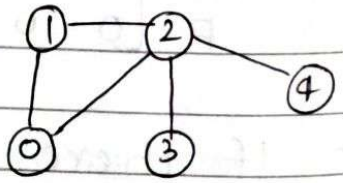
eg :



BFS : A C E B D F
DFS : A C B F D E



BFS : A B C D E F
DFS : A B C E F D



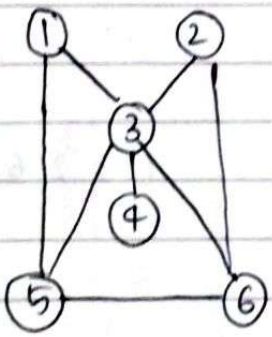
BFS : 0 1 2 3 4
DFS : 0 1 2 3 4

→ A graph wai is a tree but not vise versa.

19/9/25

```
Algorithm_DFS (int v){
// perform DFS traversal on a graph with
   start vertex value v //
push(v)
while (stack not empty){
     pop_v = pop();  <-- if (visited[pop_v] = 0){
     print(pop_v)
     visited [pop_v] = 1
     for (int j = n; j ≥ 1; j--){
        if adj[pop_v][j] = 1 && visited[j] = 0
              push(j);
     } // end of for
} end of while
 return()
} // end of DFS
```

Working



adj

	1	2	3	4	5	6
1	0	0	1	0	1	0
2	0	0	1	0	0	1
3	1	1	0	1	1	1
4	0	0	1	0	0	0
5	1	0	1	0	0	1
6	0	1	1	0	1	0

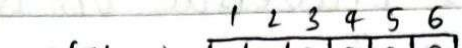
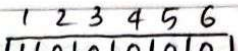
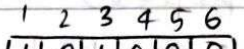
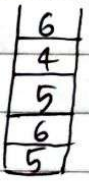
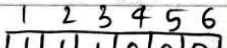
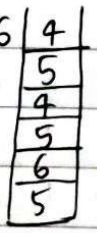
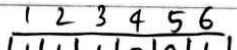
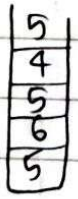
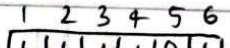
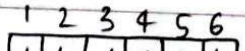
← j

initially stack will be

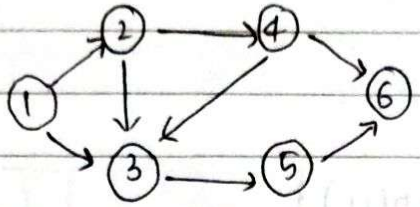
1	2	3	4	5	6
0	0	0	0	0	0

0 : unvisited

1 : visited

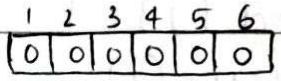
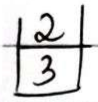
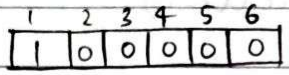
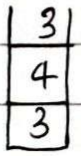
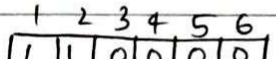
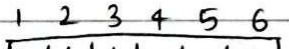
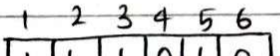
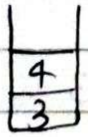
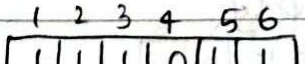
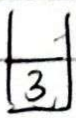
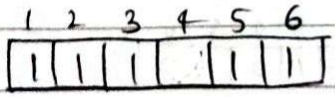
Iteration	stack	visited	o/p
• Initial	empty →  		-
• 1.	pop_v=1   =>  (for)		1
• 2.	pop_v=3  		1 3
• 3.	pop_v=2  		1 3 2
• 4.	pop_v=6  		1 3 2 6
• 5.	pop_v=4  no change 		1 3 2 6 4
• 6.	pop_v=5  		1 3 2 6 4 5
• 7	every node visited ∴ 		

Q. Trace working of



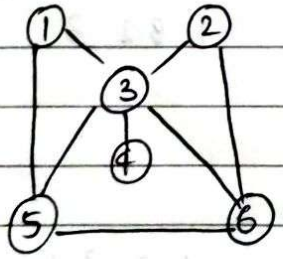
	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	1	1	0	0
3	0	0	0	0	1	0
4	0	0	1	0	0	1
5	0	0	0	0	0	1
6	0	0	0	0	0	0

6×6

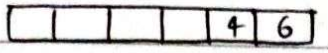
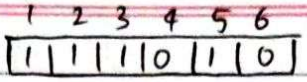
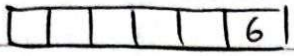
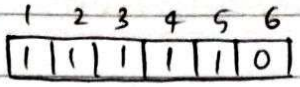
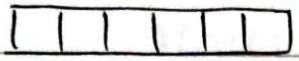
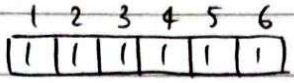
	Stack	visited	o/p
1.			-
2.	pop_v = 1 		1
3.	pop_v = 2 		1 2
4.	pop_v = 3 		1 2 3
5.	pop_v = 5 		1 2 3 5
6.	pop_v = 6 		1 2 3 5 6
7.	pop_v = 4 		1 2 3 5 6 4 (~~does not print 3~~)

```
Algorithm BFS (int v){
    print (v)
    visited [v] = 1
    enqueue (v)
    while (queue not empty){
        v = dequeue()
        for (j=1; j<=n; j++){
            if (adj[v][j]=1 && visited[j]=0){
                print(j)
                visited [j] = 1
                enqueue (j)
            } end if
        } end for
    } // end while
    return()
}
```

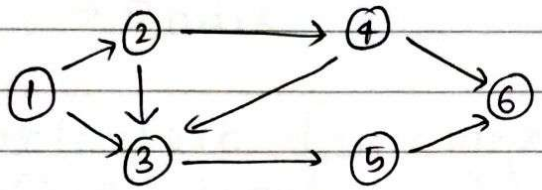
Trace working

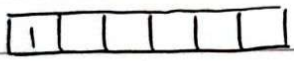
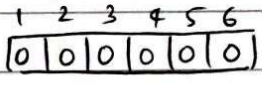
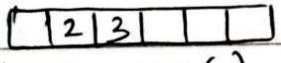
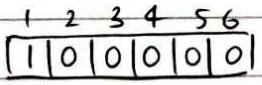
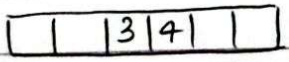
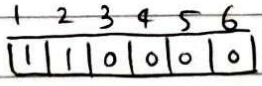
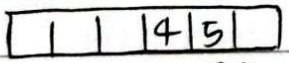
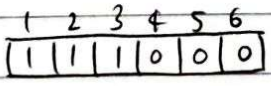
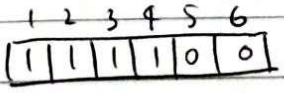
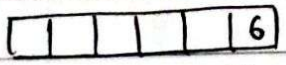
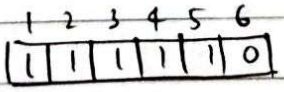
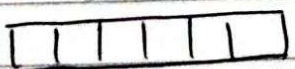
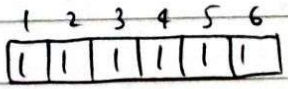


	Queue	Visited	o/p
		1 2 3 4 5 6 0 0 0 0 0 0	
1.	1	empty	empty
2.	3 5	1 0 0 0 0 0	1
3.	5 2 4 6	1 0 1 0 0 0	1 3
4.	2 4 6	1 0 1 0 1 0	1 3 5

5.			1 3 5 2
6.			1 3 5 2 4
7.			1 3 5 2 4 6

Trace working of :



	Queue	Visited	Op
1.			empty
2.	 dequeue()		1
3.	 dequeue()		1 2
4.	 dequeue()		1 2 3
5.	 dequeue()		1 2 3 4
6.	 dequeue()		1 2 3 4 5
7.	 dequeue()		1 2 3 4 5 6