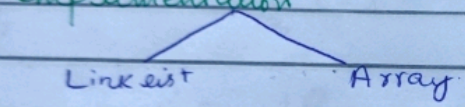
15 Nov, 2023 Wednesday

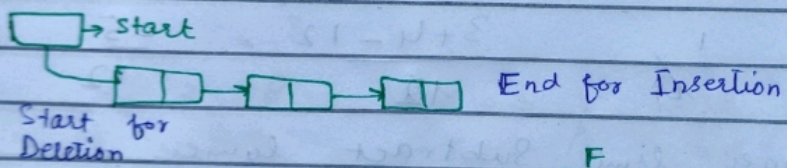
Data Structure & Algorithm

Queue

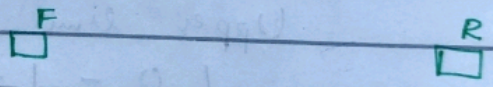
1- Linear data structure
2- Stores data tempararily
3- Principle FIFO (First in First Out) LIFO Last Out
4- Two-ended list / double ended
   ↓
   NO Insertion / Deletion
   First Front end for deletion
   Rear / Backend for Insertion
   Native FIFO
   Priority Least Integer → Highest Priority
   If priority same → FIFO.

Implementation

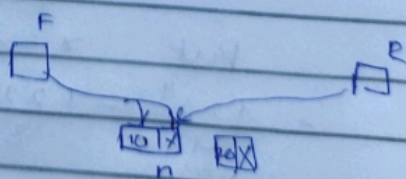




Queue → LinkList



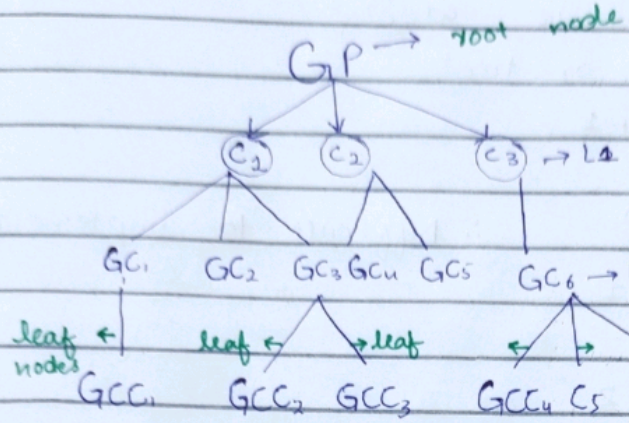

```
node *Front = null;
node *Rear = null;
enque(int value) → Insert At End.
{
    if (R == null) // If R → null → means whole LL is null.
{   node *n = new node();
    n→d = value;
    n→l = null;
    Front = n;
    Rear = n; }
else
    node *n = new node();
    n→d = value;
    n→l = null;
```



22 Nov, 2023 Wednesday

Data Structure & Algorithm

Tree (Non-linear)



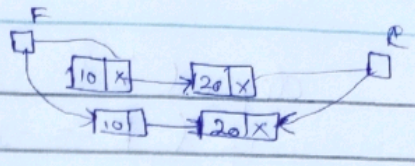
Hierarcy
One-to-many connectivity
Type of graph.

- Non-linear data structure
- Store hierarchial data
- Collection of vertices/nodes & edges.
- Vertices hold the data.
- Edges will define the relationship among data (Parent-child relationship)
- Growth starts from node → root node → Parent that has no parent → Level 0.
- Unidirectional growth.
- No loops or cyclic path.
- Nodes with zero child → external node / leaf node / terminal.
- Other than leaf nodes are internal nodes.

Generation or Levels

- All the nodes at same level → cousin nodes
- For sibling's → level no should be same and parent also.
- Ancestor → Complete chain join one line
- Predecessor → One step up
- Successor → One step down
- Descendants → From particular node to leaf.

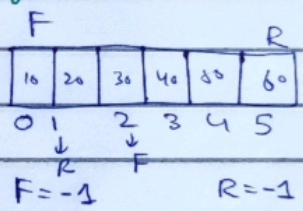
```
Rear -> l = n;
Rear = n;  }
```



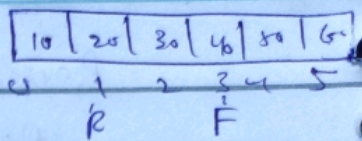

```
deque ( ) {          ∴ deletion
  if ( Front == null )
{   cout << "Underflow";
                              }
else if ( Front == Rear ) {
     cout << Front -> d;  Front, Rear = null; }
else
   cout << Front -> d;
   Front = Front -> l;
                              }
```

Queue Implementation using Arrays. (Circular Array)

```
int queue [ 6 ];
int F, R = -1;
enque ( int value ) {
if ( R == size - 1 )
{
   cout << "Overflow";
                              }
else if ( F == -1 && R == -1 ) {
     F++;  R++;
     queue [R] = value;
                              }
else if ( F > 0 && R == size-1) {
   R = 0;
     queue [R] = value; }
else
     R++;
   queue [R] = value; }
```




```
if ( F==0 && R==size-1) ||
                ( F=R+1 )
```




```
dequeue ( ) {
  if ( F == -1 && R == -1 ) {
      cout << " Underflow "; }
  else if ( F == R ) {
    cout << queue [F];
      F = R = = -1; }
  else if ( R > 0 && F == size - 1 )
  {       cout << queue [F];
      F = 0; }

    else
        cout << queue [F];
            F++;
```

28Nov, 2023 Tuesday

Data Structure

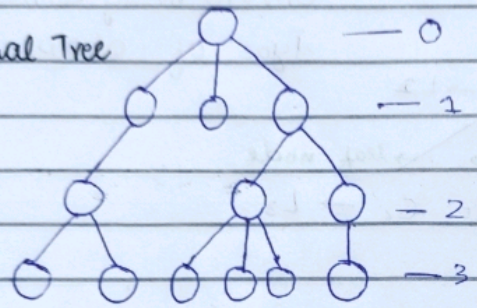
: Tree

Path:- Sequence of consective edges.

Nodes: Define path or height or tree

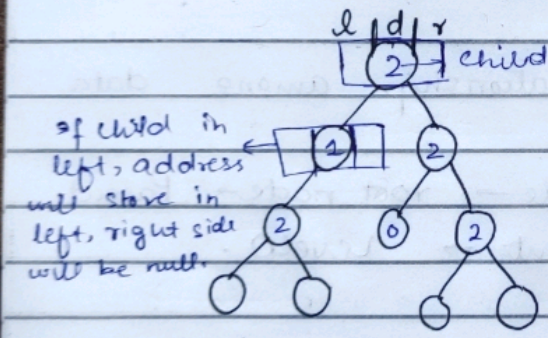
depth / height = l max + 1

General Tree



∴ Difficult to implement

Binary Tree: In which every node has max 2 and 0 child.



∴ No back tracking

- Level sequence
- Left to right

```
Struct node {
        int d;
    tnode "left";
    tnode "right";
    }
```

Find maximum number of nodes

$l_0 = 2^0 = 1$

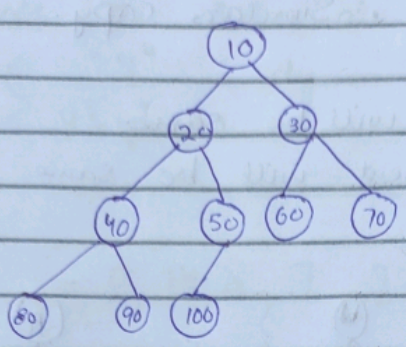
$l_1 = 2^1 = 2$

$l_2 = 2^2 = 4$

$l_3 = 2^3 = 8$

: Complete Binary

10, 20, 30, 40, 50, 60, 70, 80, 90, 200



Complete Binary
∴ 2nd last level will be full.
∵ Last level must have insertions from left to right
Full : Last level will be full.

In Array:

$Lc(k) = 2k+1$ $\quad$ $Rc(k) = 2k+2$

$Lc(3) = 2(3)+1 = 7$ $\quad$ $Rc(3) = 2(3)+2 = 8$

∴ Lc = Left child
∴ Rc = Right child
∴ P = Parent

$$P(k) = \left\lfloor \frac{k}{2} \right\rfloor$$

$$P(k) = \left\lfloor \frac{3}{2} \right\rfloor$$

$P(k) = 1$

$Lc(1) = 2(1)+1 = 3$ $\quad$ $Rc(1) = 2(1)+2 = 4$

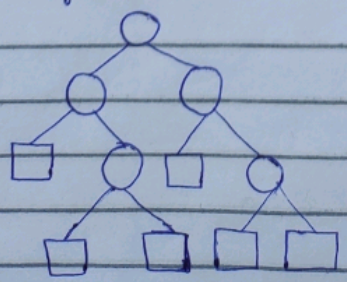
$$P(1) = \left\lfloor \frac{1}{2} \right\rfloor$$

$= 0$

Extended / 2-tree:- Type of Binary Tree

Every node has either zero child or 2 child

Internal child → Circle ○ $\quad$ External child → Square □

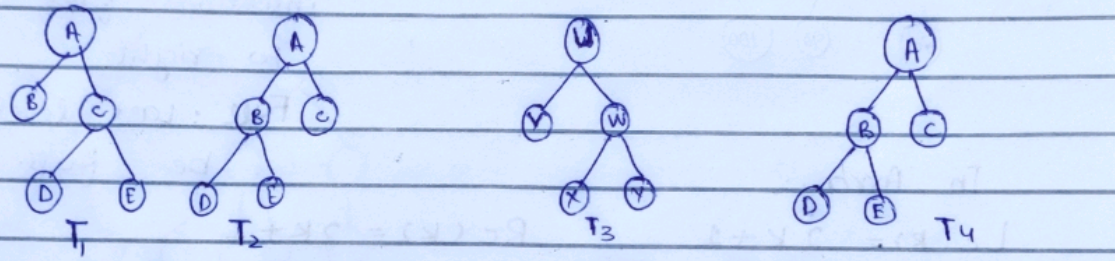


- Internal nodes = n = 5
- External nodes = n+1 = 6

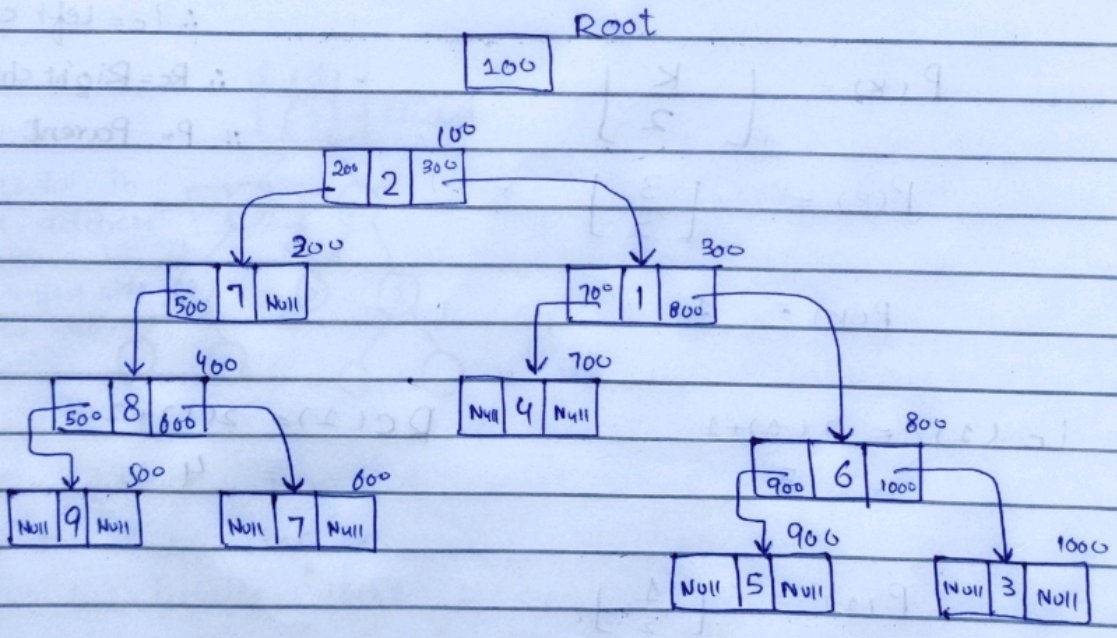
Total = 2n+1 = 11

How to tell difference b/w two o trees
Will be different, Similar, copy.

Similar:: Geometry/ shape will be same.
Copy:: Shape & data element will be same



T_1, T_2 = different $T_1 T_3$ = Similar $T_2 T_4$ = Copy



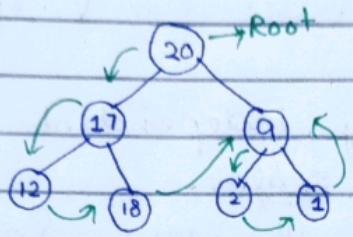
Pre - Order Traversal
In - Order Traversal
Post - Order Traversal

Data Structure

Traversal Trees

Pre-order

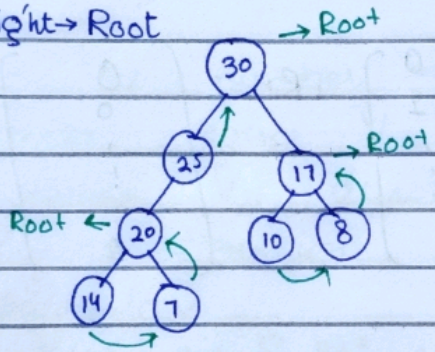
Root → left → Right

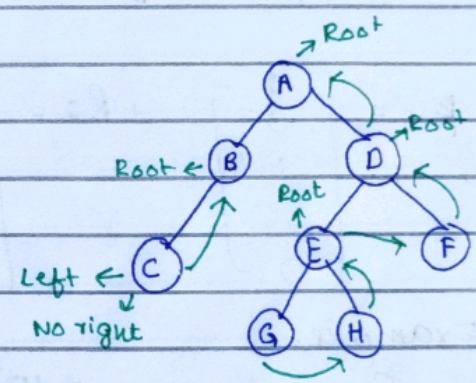


20 → 17 → 12 → 18 → 9 → 2 → 1 (Pre-order)

Post-order

Left → Right → Root



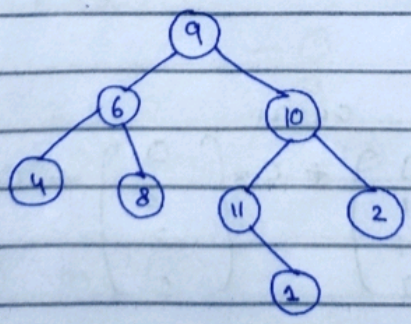


14 → 7 → 20 → 10 → 8 → 17 → 25 → 30 (Post-order)

C → B → G → H → E → F → D → A (Post-order)

In-order

Left → Root → Right

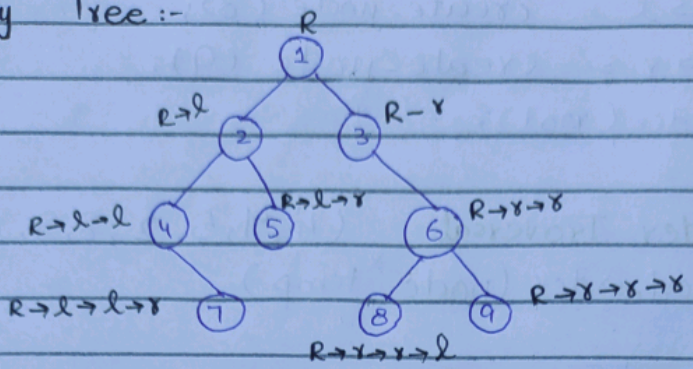


4 → 6 → 8 → 9 → 11 → 1 → 10 → 2 (In-order)

5 Dec, 2023 Tuesday

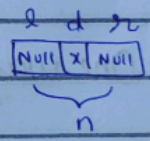
Data Structure

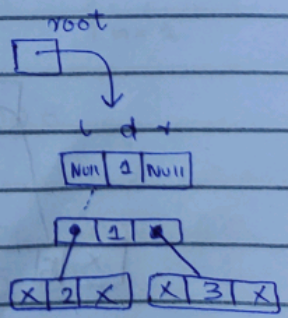
Binary Tree :-



Node structure are uniform

```
struct node
{
  int d;
  node * left;
  node * right;
  };
  node *root = null;
*node create_node(int x) {
  node n = new node();
    n → d = x;
    n → left = Null;
    n → right = Null;
    return n;
        }
int main {
  node *temp;
  temp = create_node(1);
    root = temp;
root → l = create_node(2);
root → r = create_node(3);
root → l → l = create_node(4);
root → l → r = create_node(5);
root → r → r = create_node(6);
```





```
root → l → l → r = create-node (7);
root → r → r → l = create-node (8);
root → r → r → r = create-node (9);
post-order (root);
```

Post Order Traversal (4, 4, 5, 2, 8, 9, 6, 3, 1)

```
void post.order (node *temp)
{
    if (temp != Null) {
    post.order (temp → l);
    post.order (temp → r);
     cout << temp → d;
```

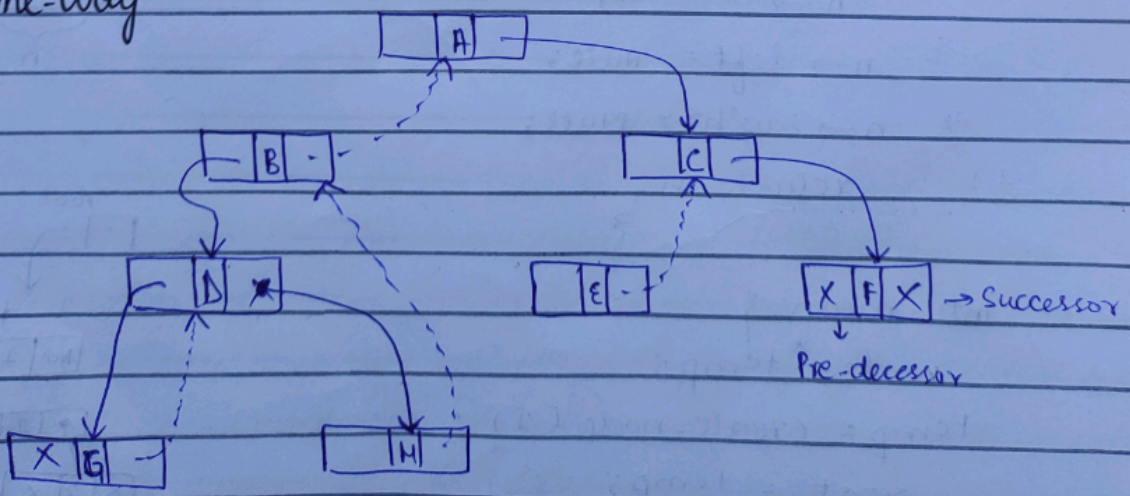
1. root
1.1: root → l
1.1.1: r → l → l
1.1.1.1: r → l → l → F
1.1.1.2: r → l → l → r
1.1.3: Print 7

Threading in Trees

One-way Inorder threading
Two-way Inorder threading
Two-way Inorder & threading with header nodes.

G, D, H, B, A, E, C, F

One-way



6 Dec, 2023 Wednesday.

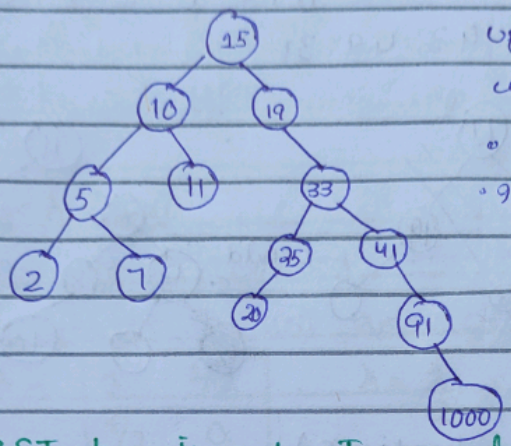
Data Structure and Algorithm

Binary Search Tree

15, 19, 33, 41, 10, 25, 5, 7, 2, 11, 91, 100, 20

- First number will become root

Trees
To make searching & sorting easier



Upcoming number will be compared with root
- If greater → right child
- If too number smaller then root → left child

Check BST by In-order Traversal

2, 5, 7, 10, 11, 15, 19, 20, 25, 33, 41, 91, 1000

Deletion at Specific Point (In-order) (One right most left)

Move one step downward right then move the most left child until it becomes Null

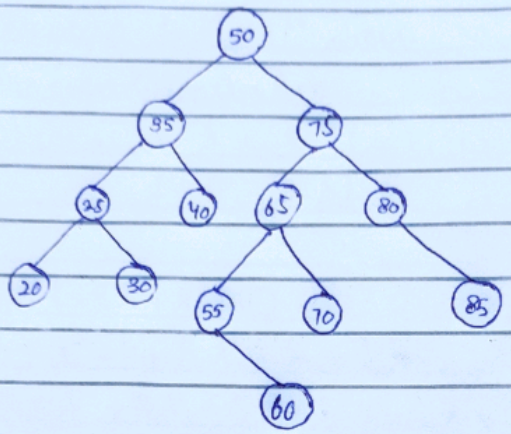
If found same data, move data to the right

BST → Concept
Searching
Sorting
Insertion → Find & Insert
Deletion
- Zero child → leaf
- One child, Two child → Non-leaf

12 Dec, 2023 Tuesday

Data Structure & Algorithm

BST



Deletion

1 : No child

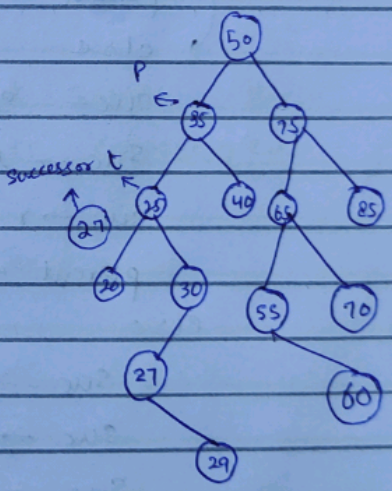
2 : One child

3 : Two child

```
Struct node* findchild (node *temp) {
    Struct *node child;
    if ( temp → l && temp → r == Null) {
            child = Null;
            return child; }
    else if ( temp → l != Null) {
            child = temp → l;
            return child;
                    }
    else
            child = temp → r;
            return child;
                    }
void delete (node * root, int key) {
    node * temp = root;
    node * parent = Null;
    while ( tmp != Null) {
    if ( key < temp → d ) {
            parent = tmp;
    temp = temp → l;
                    }
```

```
else if ( key > temp → d )
  {       parent = temp;
          temp = temp → r; }
else {
   if (parent != Null) {
     if (parent
   if  (temp = parent → l) {
   parent → l = findchild (temp);
                                  }
   else {
     parent → r = find child ( temp); }
                                  }
   else
      root = find child (temp);
```

Deletion of two childs




```
struct node* successor (node* temp) {
 struct *node  suc = temp → r;
 struct *node  parentSuccessor = temp;
 while ( suc → l != Null) {
   parentSuccessor = Successor;
        Suc = Suc → l;
                              }
struct *child = find child ( suc);
    if ( suc = parentSuccessor → l)
        parentSuccessor → l = child;
   else
        paredSuccessor → r = child;
   return  Successor;
                              }
void delete (node* root, int Key){
   node * temp = root;
   node * parent = Null;
```

```
while( temp != Null) {
 if   (key < temp → d) {
        parent = temp;
        temp = temp → l;   }
   else if ( key > temp → d ) {
      parent =  temp;
       temp= temp → r;
                          }
   else {
       if ( parent != Null) {
       if ( temp= parent → l) {
       Successor= Successor (temp);
        suc → l = temp → l;
        suc → r =  temp → r;
        parent → left = Successor;
       else
        suc=  Suc (temp);
        suc → l =  temp → l;
        suc → r  = temp → r;
        parent → r =  suc;
   else
        suc=  suc ( temp );
        suc → l = temp → l;
        suc → r = temp → r;
        root =  Successor;
                          }
                          }
```

19Dec, Tuesday

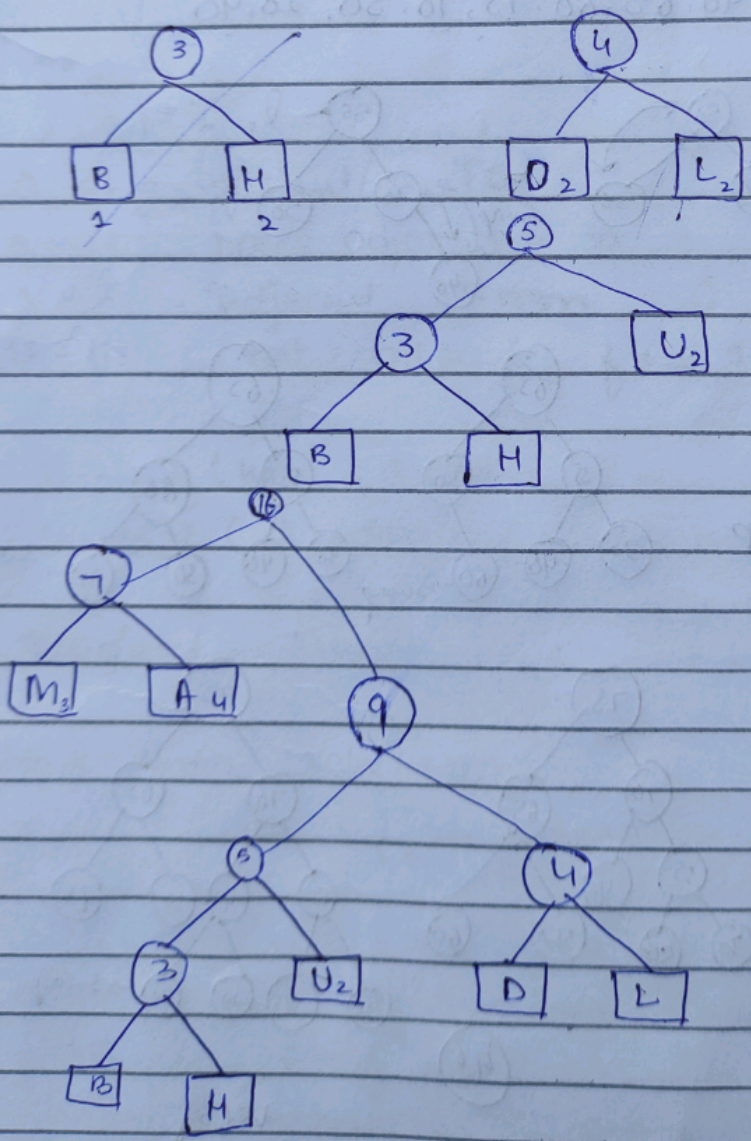
Data Structure

Huffman Algorithm

MUHAMMAD ABDULLAH

- M → 3
- U → 2
- H → 2
- A → 4
- D → 2
- B → 1
- L → 2

Minimum Weight
B & H



Heap Tree

Form of complete tree

- 2nd last full, left to right insertion

Unsorted data → ascending (use min heap)

Unsorted → descending (max heap)

Minheap (Parent < child)

↓

Parent either smaller or equal → child

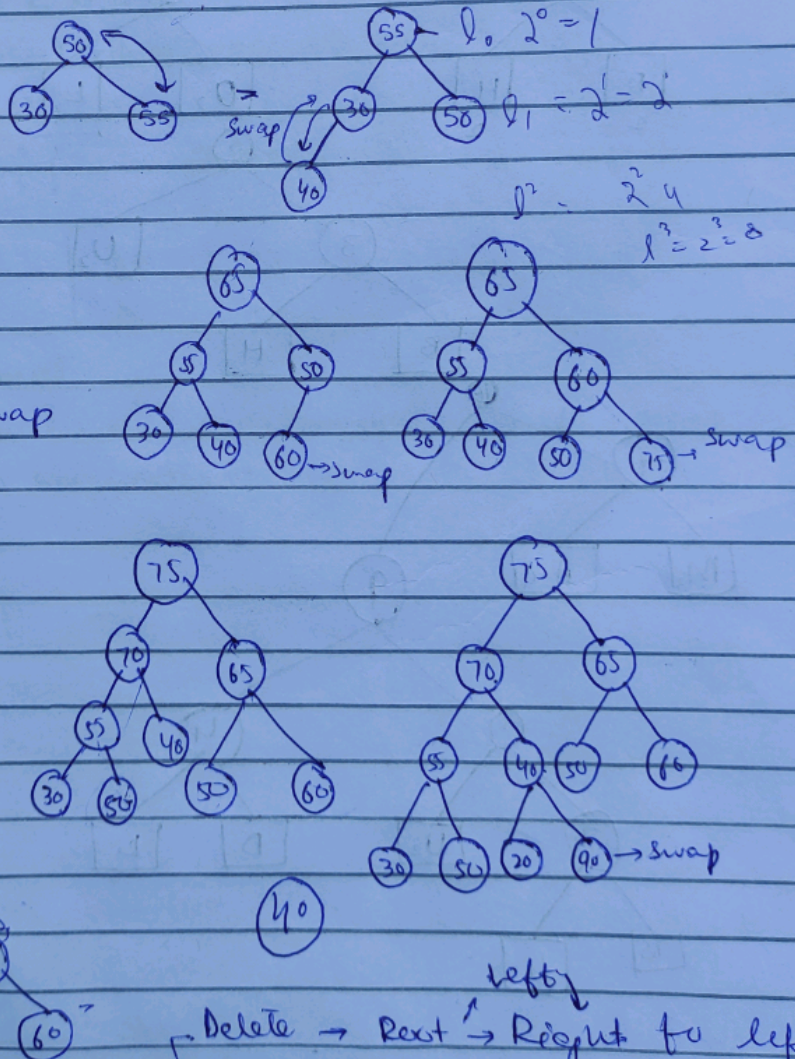
Maxheap (Parent > child)

↓

parent either larger ~~or~~ equal to its child

50, 30, 55, 40, 65, 60, 75, 70, 50, 20, 90

Maxheap:-

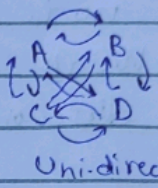


- 90, 75

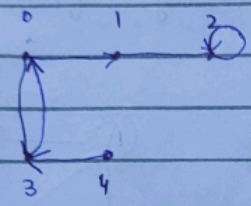
Graph

- Non-linear data Structure
- Combination of vertices & edges
  vertices (nodes), edges (connection)
  V (0, 1, 2, 3, 4)
  E (0,1)(1,2)(0,3)(3,0)(2,2)(4,3)
  (source, destination)

Directed Graph



Uni-direction

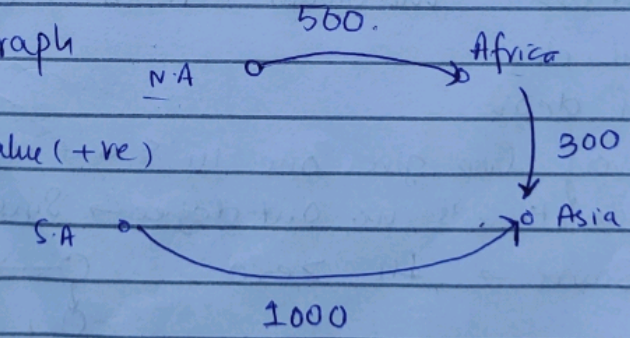


Directed Graph

- 0 is adjacent/incident TO 1.
- 1 is not adjacent to 1.
- 1 is adjacent From 0.
- 0 is not adjacent from 1.

u → v (has a direction)
u — v (Bi-direction)

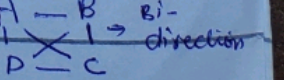
Weighted Graph



"Edges contain value (+ve)

Path:- Sequence of vertices that connect two nodes in Graph.
: Source (Lahore to Islamabad, Car is path,
Node → Path.

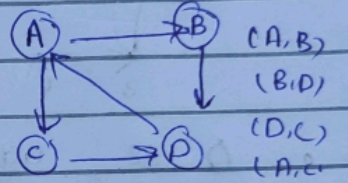
Complete Path:- Every vertex is connected to another vertex. (For undirected Graph).



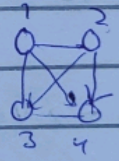
Cyclic Path / Cycle:
Soure & destination will be same.
Sequence always written in square brackets
If Soure & destination → differnt → simple path.
No of edges connected to one node → degree.

Strongly connected → can easily go from one node to other directly. → Bi-directional
Weak connectivity → can go only from one direction → NO back way.

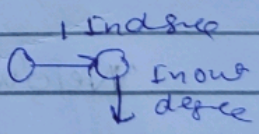
Directed graph Strongly connected



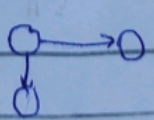
Complete graph → nodes reached diredtly.
Multi graph →
Simple → No loop → no edges
Undirected Graph



$n*(n-1)/2$
$4(4-1)/2 \Rightarrow 4(3)/2 = 12/2 = 6$
Directed maximum $n*(n-1)$ case



Out degree
In degree
Every loop give one In & out degree.
Node, there's no out degree → Sink degree
Soure → In zero



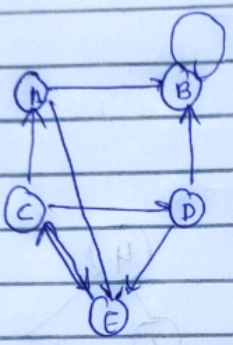
Two ways to Implement Graph
- Array → Adjacency Matrix
- LinkList → Adjacency List

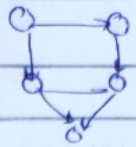
Adjacency Matrix

n = 5

Matrix M 5×5

	A 0	B 1	C 2	D 3	E 4
A 0	A,A	A,B			
B 1	B,A				
C 2					
D 3					
E 4					





n = 5

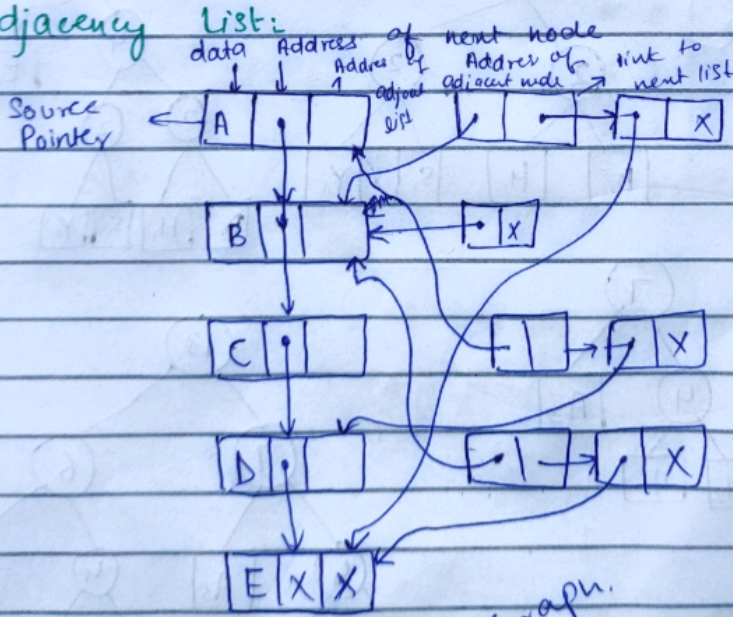
	A 0	B 1	C 2	D 3	E 4
A 0	AA	AB	AC	AD	AE
B 1	BA	BB	BC	BD	BE
C 2	CA	CB	CC	CD	CE
D 3	DA	DB	DC	DD	DE
E 4	EA	EB	EC	ED	EE

AA → loop = zero

AB → A to B = Exist = 1

	A 0	B 1	C 2	D 3	E 4
A	0	1	0	0	1
B	0	1	0	0	0
C	1	0	0	1	0
D	0	1	0	0	1
E	0	0	1	0	0

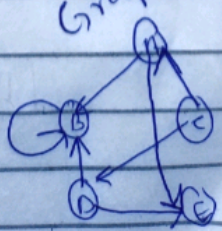
Adjacency List:

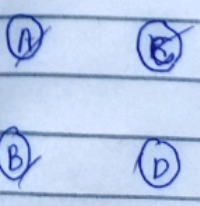


Connectivity will be shown by adjacency list

A	B, E
B	B
C	A, D
D	B, E
E	NO

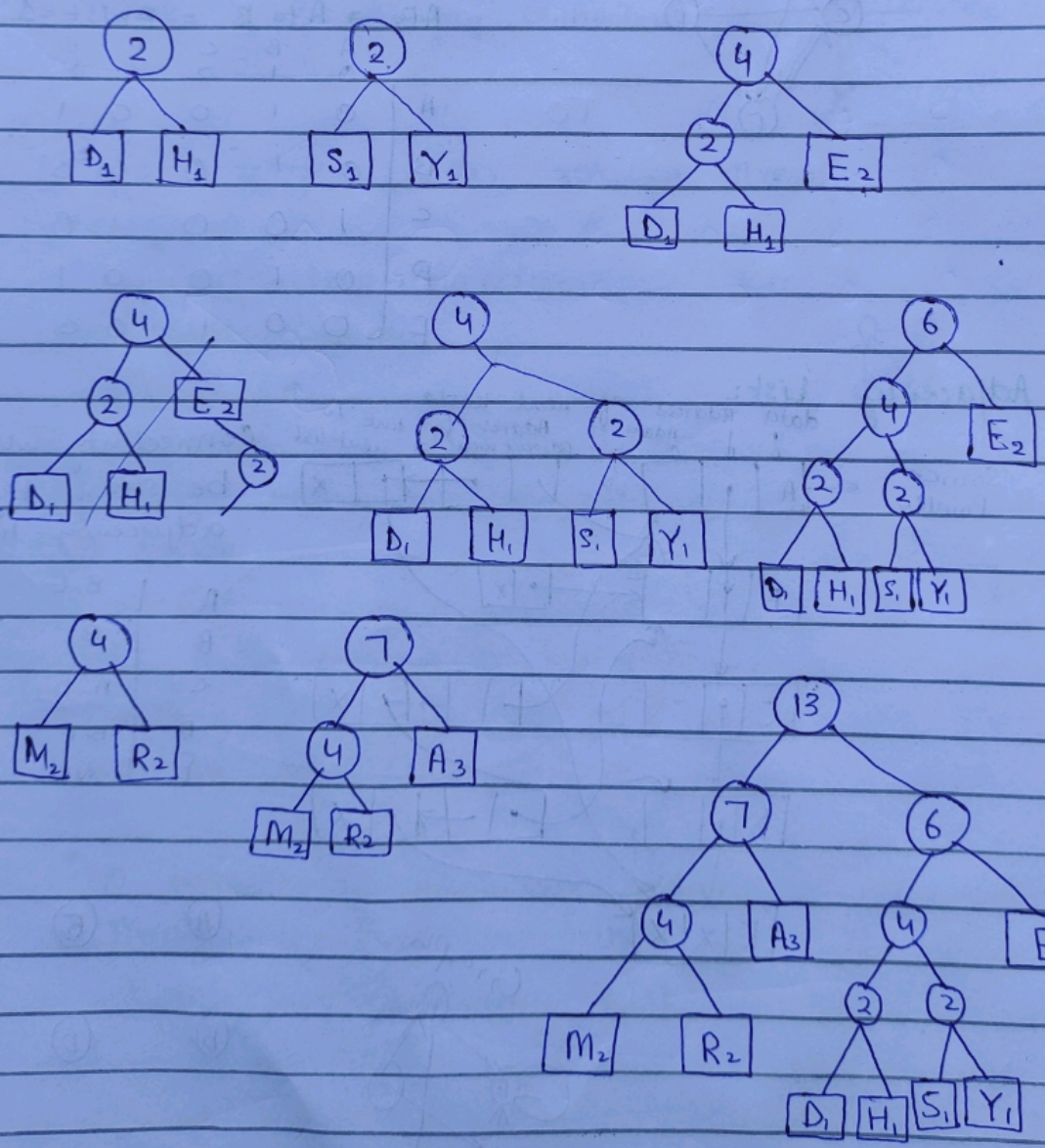
Graph.





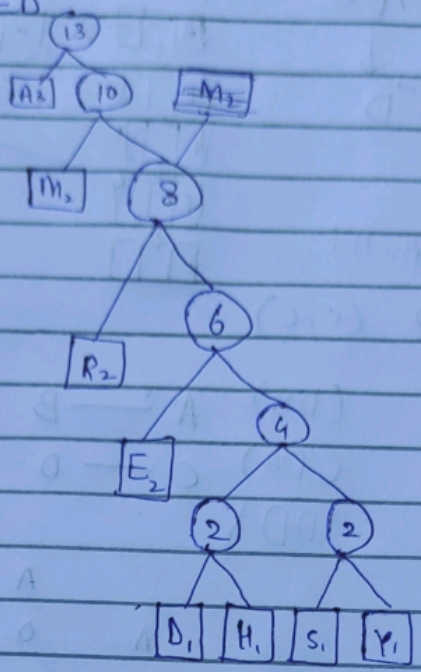
MARYAM RASHEED

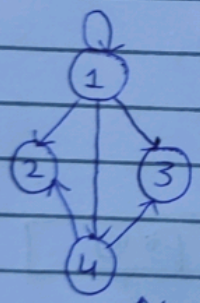
- M – 2
- A – 3
- R – 2
- Y – 1
- S – 1
- H – 1
- E – 2
- D – 1



MARYAM RASHEED

M-2
A-3
R-2
Y-1
S-1
H-1
E-1
D-1





n=4

Directed Graph

	A 1	B	C	D 4
A 1	1	1	1	1
B 2	0	0	0	0
C 3	0	0	0	0
D 4	0	1	1	0

4×4

Adjacency List:-

Two Methods of Representing