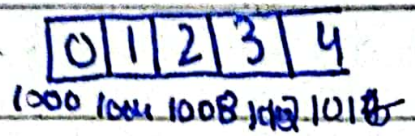
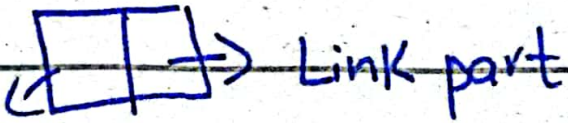
25/4/24 (After Mid)

* Linear D.S:
  - → Array
  - → stack
  - → Queve
  - → List



→ Linked List:



Reference part

→ class Node
{
int data;
Node * node;
next
}

→ Constructor:

```
Node(int d)
{
    data = d
    Node = nullptr;
}
```

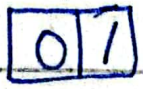
Next

```
Node()
{
    data = 0
    next = nullptr;
}
```

```
Node(int d, Node* n)
{ data = d;
  Next = n;
}
```

→ Node Head = new Node(0);

Head.next = First

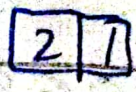


Node First = new Node(1);

First.next = second

Node Second = new Node(2);

Second.next = Third

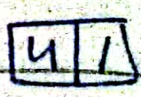


Node Third = new Node(3);

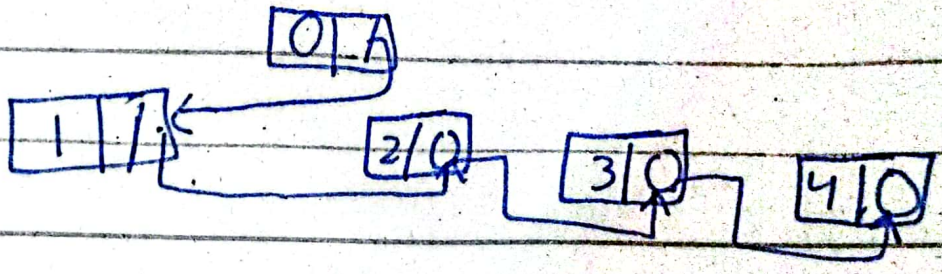
Third.next = Fourth

Node Fourth = new Node(4);

Fourth.next = null;



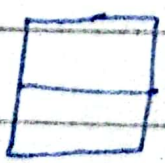
(Scattered Node)



→ Node * p = new Node(10)

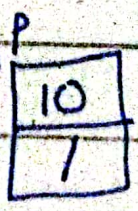
Steps

(a) P

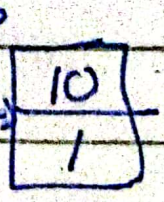


(b) P

(c) P

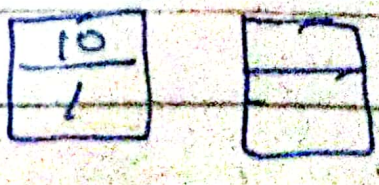


(d) P

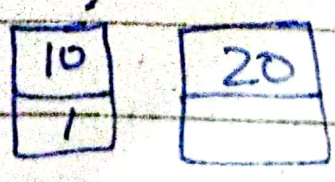


→ P.next = new Node(20)

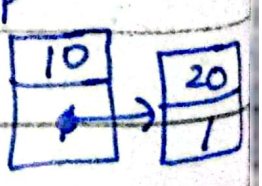
P



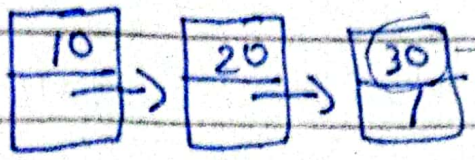
P



P



→ P.next.next = new Node(30)



→ P.next.next.next.data
(Not feasible for 1000 Nodes).

→ For proper solution, we make class List:

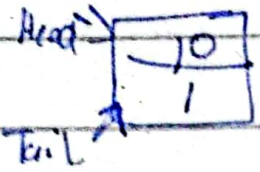
```
class List{
        Node * head;
        Node * tail;
      }
→ List(){
    { head=tail= nullptr;
    }
→ bool isEmpty()
    { if(Head ==0)
        { return 1;
        }
      else
        { return 0;
        }
    }
```

→ For First element head and tail points to same.

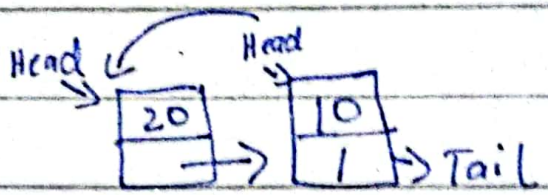
→ void addToHead(int e)

```
{
    Head = new Node(e, Head);
    if(Tail==0)
    { Tail = Head;
    }
```

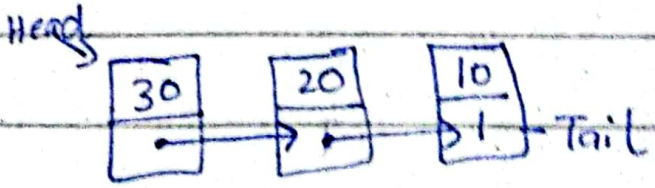
: addToHead(10)



: addToHead(20)



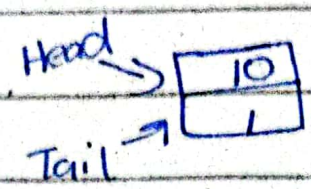
: addToHead(30)



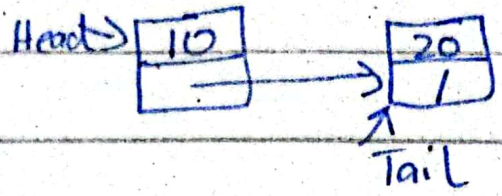
→ void addToTail(int e)

```
{   if(Tail==0)
    { Head = Tail = new Node(e)
    }
    else          // Tail != 0
    { Tail.Next = new Node(e)
      Tail = Tail.Next;
    }
```

: addTotail(10)

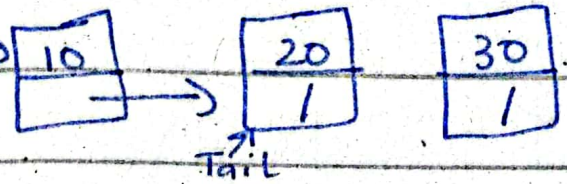


: addTotail(20)

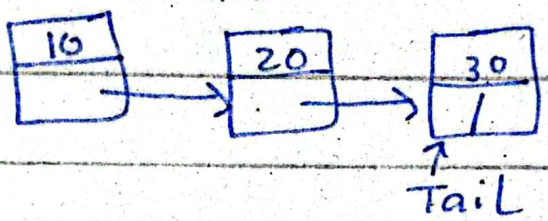


: addToTail(30)

(a)

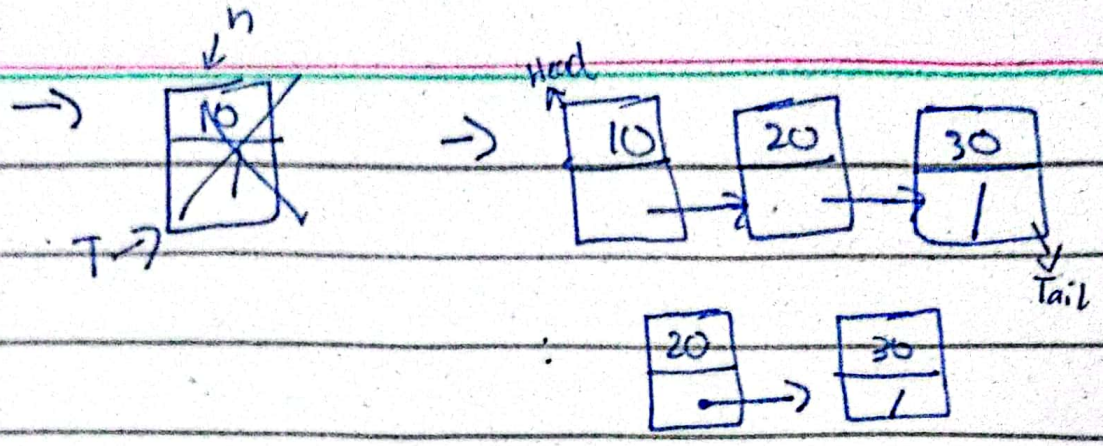


(b)

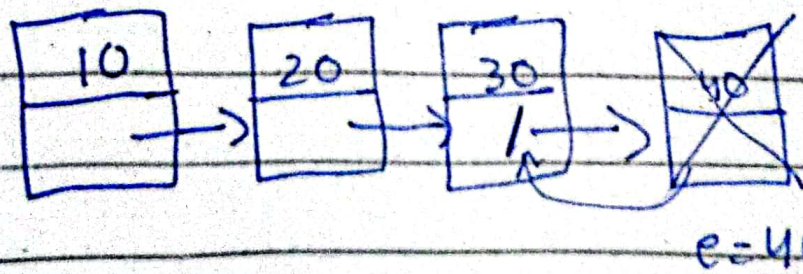


→ int deleteFromHead()

```
{   int e = Head->data;
    Node *temp = Head;
    if (Head == Tail)
        {   delete Head;
            Head = Tail = 0;
    Head = Head.next;
    Delete temp
    return e;
}       }
```




```
-> int deleteFromTail()
   {
      int el = Tail->data;
      If( Head == Tail )
      {
         Delete Head;
         Head = Tail = 0;
      }
      Node * temp;
      for ( temp = head; temp->next != tail;
                                  temp = temp->next );
      delete Tail;
      Tail = temp;
      Tail->next = 0;
      return e;
```



30/4/24

Single Linked List

→ deleteNode(int e1)

→ Node *Pred, * temp;

for(Pred = Head; temp = Head.Next;)

{

while(temp != '\0' || !(temp.data == e1))

{

Pred = Pred.Next;

temp = temp.next;

if(temp != 0)

{

Pred.next = temp.next

if(Temp == tail)

{ Tail = Pred

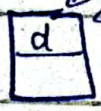
delete temp;

}

}

}

① Head, Tail



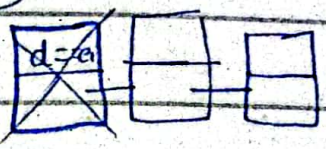
Head.data = e1

Head = Tail

delete Head

Head = Tail = 0

②

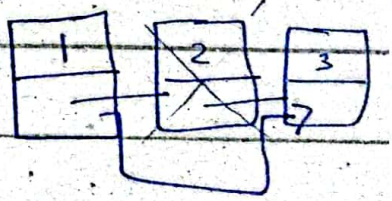


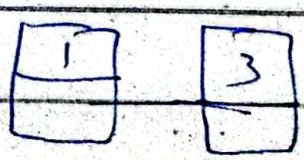
Node * temp = head;

Head = Head.next;

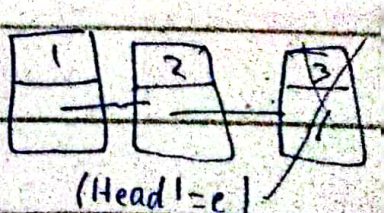
delete temp;

③





→



(Head != e)

→ bool isINList(int e1)

{ Node * temp = Head;

while(temp.Next != NULL)

```
{ if (temp.data == e1)
    { return true;
    }
  temp = temp.next;
}
return false;
}
```

→ for(temp = head ; temp != 0 && !(temp.data == e1))

```
{ temp = temp.next;
  if( temp == 0)
    { return 0;          : (other way)
    }
  else
    { return 1;
    }
}
```

→ Destructor:

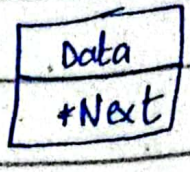
① ~List()

```
{ Node * temp = Head.next;
  while( temp != NULL)
    { delete Head;
      Head = temp;
    } temp = temp.next;
```

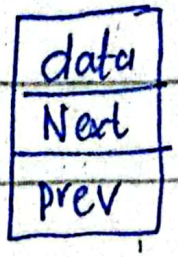
```
    delete Tail;
}
~LinkedList()
{
    while (!head)
    {
        Node * temp = head;
        head = head.next
        delete temp;
    }
    head = tail = 0;
}
for (temp = head ; head != 0 ; head = head.next)
{
    delete temp;
}
~List()
{
    for (Node *p = head ; !Head() )
    {
        p = p.Next;
        delete Head
        Head = p;
    }
```

2/5/24

(*) SLLIST
(Single Linked)



(*) DLLIST
(Double Linked)



→ DLLIST Node

```
{ int data;
    DLLIST Node * Next;
    DLLIST Node * Prev;
}
```

```
: DLLIST Node()
{
    * Next = 0;
    * Prev = 0;
}
```

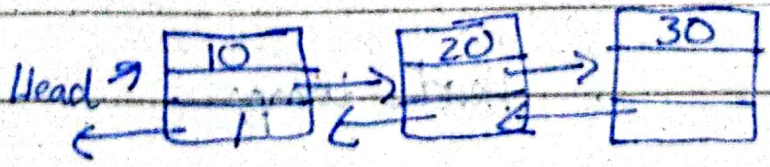
```
: DLLIST Node (int d, DLLIST Node * n,
                DLLIST Node * p)
{
    data = d;
    Next = n
    Prev = p
}
```

(*) DLLIST

```
{
    DLLISTNode * Head;
    DLLISTNode * Tail;
    DLLIST()
    {
        Head = Tail = 0;
    }
}
```

: add To DLLIST Tail(int e1)

```
{
    if (Head == 0)
    { Head = tail = new DLLISTNode(e1, 0, 0);
    }
    else
    { Tail = new DLLISTNode(e1, 0, tail)
      Tail.Prev.Next = Tail;
    }
}
```



: delete from DLLIST Tail()

```
{
    e1 = Tail.data();
    if (Head == Tail)
    { delete Head;
      Tail = Head = 0;
```

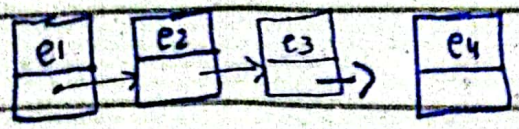
```
else
{   Tail = Tail.Prev;
    delete Tail.next
    Tail.Next = 0;
}

: add to DLLIST Head (int e1)
{
    if (Head == 0)
    {  Head.data = e1;
       Head.prev = 0;
       Hed.Next = 0;
    }  Head = Tail = new DLLIST(e1,0,0)
else
{  DLLISTNode *temp = new
                      DLLISTNode(e1)
   temp.next = Head;
   temp.prev = 0;
   Head.prev = temp;
   Head = temp;
}

:
{  Head.prev = new DLLIST Node(e1, hea
}  Head = Head.prev;                    o
```

```
: { if(Head == 0)
    { Head = new Node(n,0,0);
    }
  else
    { Head = new Node(n,Head,0);
      Head->next->prev = Head;
    }
```

```
: delete from DLLIST Head()
(1) { if (Head == 0)
      { return;
      }
    else
      { Head = Head.next;
        delete Head.prev;
        Head.prev = 0;
      }
  }
```

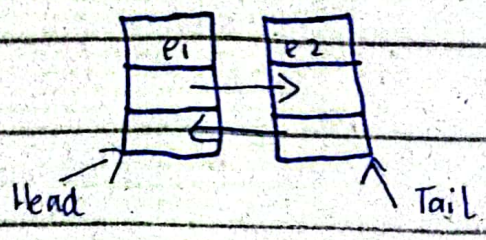
7/5/24

① Single Linked List:

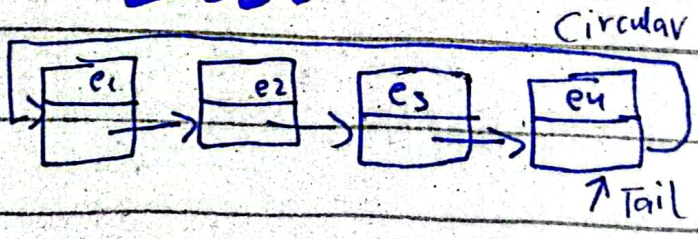


(Delting tail problems)

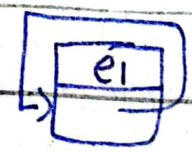
② Double Linked List:

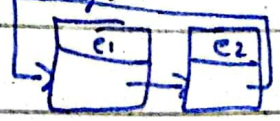


③ Circular Single Linked List:




```
->. addTo tail ( int e1 )
    { if(Tail == Tail.next)
      { Tail.next= new Node(e1, Tail);
      }
    }
```






```
:addTo tail (int e1 )
    { if( Tail == 0)
        [ Tail = new Node(e,0)
          Tail.next= Tail;
        }
```

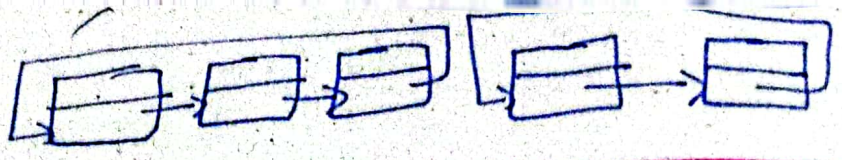
```
else
{   Tail->next= new Node(e1, Tail.next);
        Tail = Tail.Next;
    }
}
```

→ add to Head (int e1)

```
{   if(Tail ==0)
    {  Tail = newNode(e1,0)
       Tail.next= Tail;
    }
    else
    {  Tail.next= new Node(e1, Tail).next
    }
```

→ deleteFromHead()

```
{   int e1 = tail.data;
        if (tail == tail.next)
        {  delete tail;
           Tail=0;
        }  exit(0);
```



```
else
 { int e1 = tail.next.data;
    delete tail.next;
 }

: else
   { Node *temp= Tail.next;
     Tail.next = Tail.next.next;
      delete temp;
   }
```

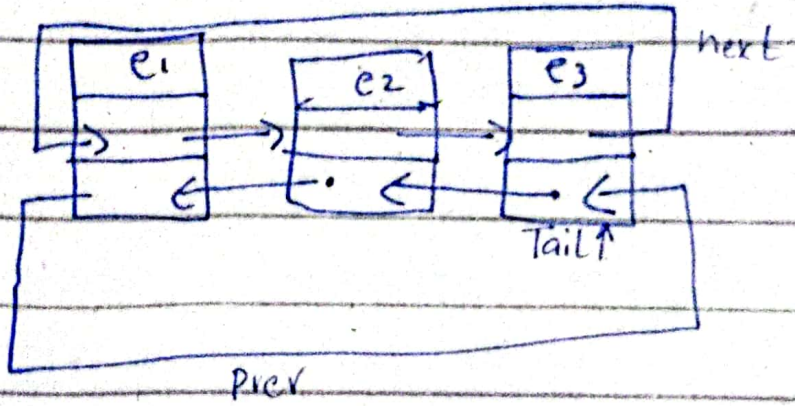
→ delete from Tail()

```
{ if(tail.next==tail)
   { delete tail;
     tail = 0;
   }
  else
   { Node * temp = Tail;
     While (temp.next != Tail)
      { temp = temp.next;
      } temp.next = Tail.next
     delete Tail;
     Tail= temp;
   }
}
```

Tail.next = temp

```
else
    for( temp=Tail.next ; temp.next != Tail;
                                    temp=temp->next)
    temp.next = Tail.next;
    delete Tail;
    Tail = temp;
}
```

⊛ Circular Double Linked List:

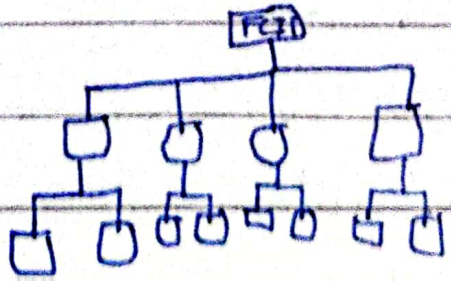


⊛ Assignment circular Double Linked List (4 Operations) (Hand Written).

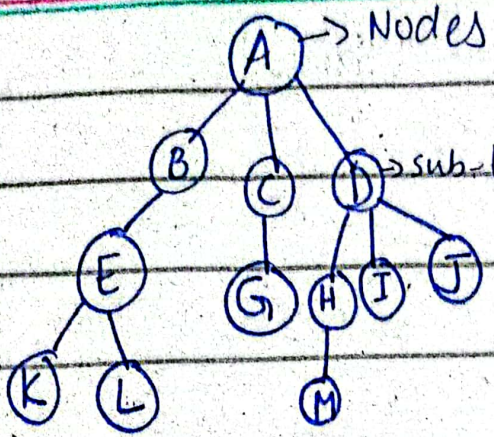
14/5/24

⊛ TREE :

→Non-Linear Data Structure



(*)



Root
: 1 Route Node (A)
: Recursive Structure (Tree)

→ Number of sub-trees of a node is called its degree. From node A, 3 degree

Imp for Interview

→ Nodes that have degree zero are called as leaf nodes or terminal nodes. (K, L, G, I, J)
Non-terminal nodes (A, B, C, D, ...)

→ ~~Nod~~ The roots of a sub-tree of a node x are called children of x and x is parent of it.

→ Children of the same parents are called siblings. (B, C, D.....)

→ The terminology can be extended so we can ask for grandparent M (E.g: GP of M is D).

→ The degree of tree is the

maximum of the degree of nodes in the tree. (Maximum 3).

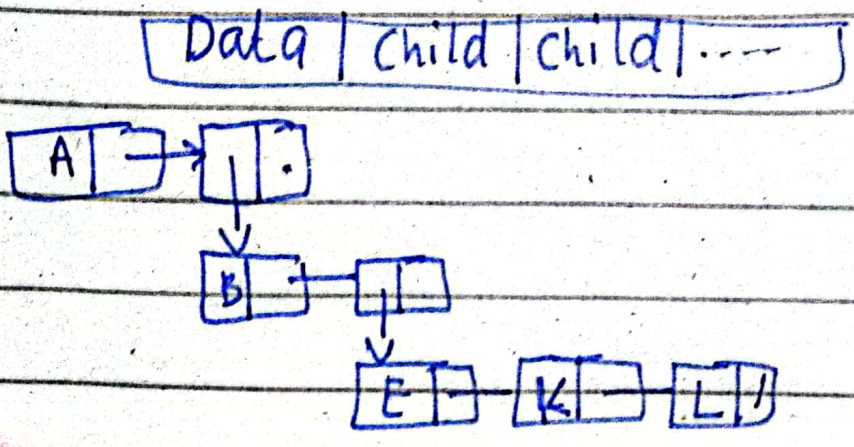
→ The Ancestors of the node along the path from the root to that node. (L → (A, B, E))

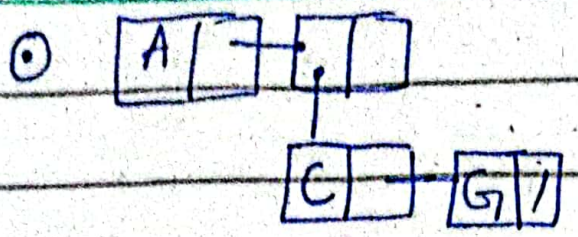
→ The level of the node is defined by ~~netting~~ mapping letting the root Be at level 1. If a node is at level l, then it children are on n+1.

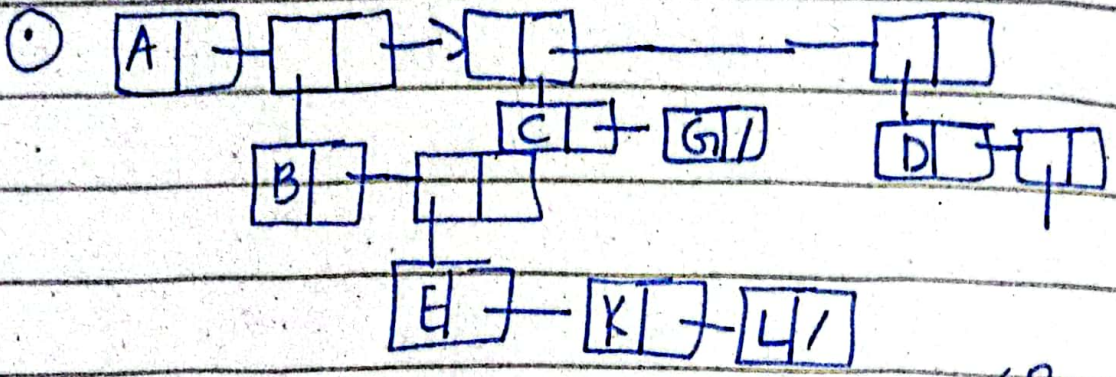
→ The height or depth of a tree is defined to be the maximum level of any node in the tree. (height: 4).

④ List representation:

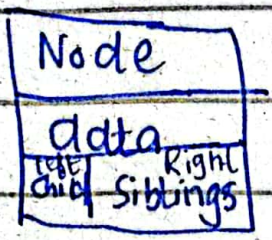
: (A(B(E(K,L)), C(G), D(H(M), I, J)))

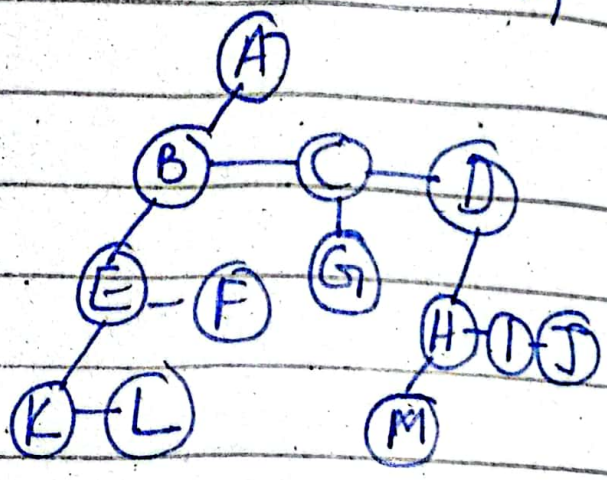


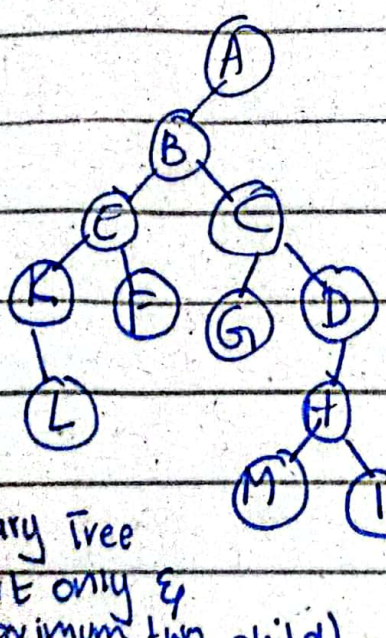




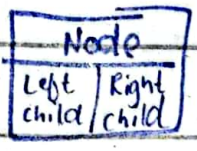
(Practice it)







(By rotating)



Binary Tree (start only & maximum two child).

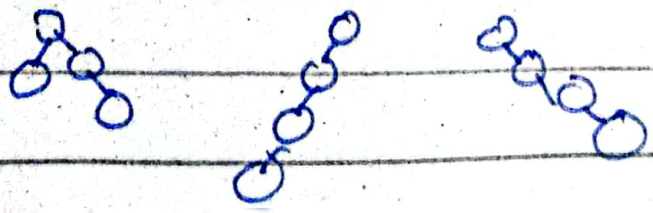
2^{i-1} → Number of nodes at particular Level (Verify it)

(*)

2^{i-1}

Height = $2^{k}-1$: K = maximum

(*)



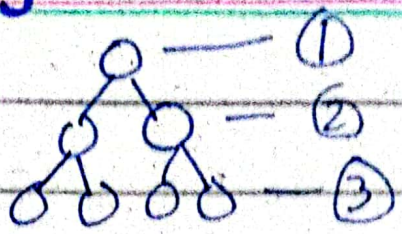
(maximum two)

(one sided)

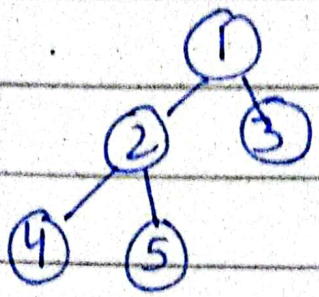
(Skewed Binary Tree)

: Binary Tree

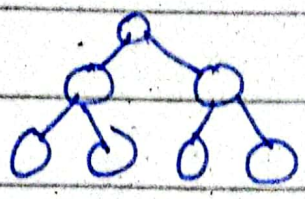
(*)



(*)

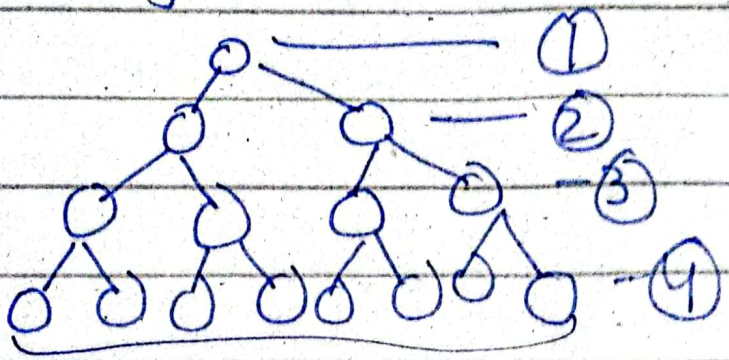


: filled from left to right complete Binary Tree



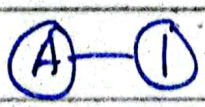
: Full Binary Tree

(*) Height = 4



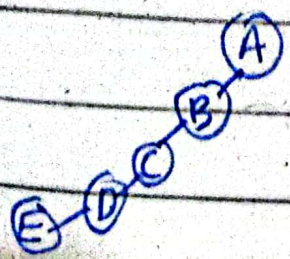
$2^{4-1} = 8$

(*)

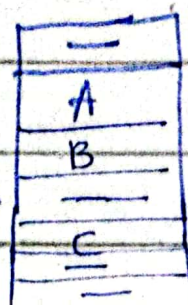


$2i$: Left $\leq n$ condition
$2i+1$: right $\leq n$ condition
Parent : $i/2$

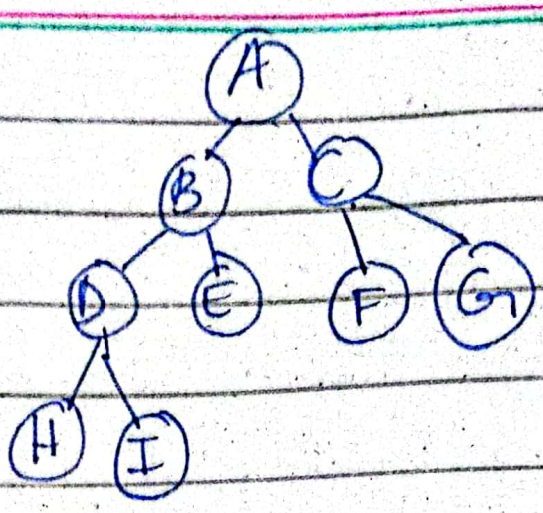
(*) Skewed Binary Tree By Array:



: blank spaces remains (too much)



(*)

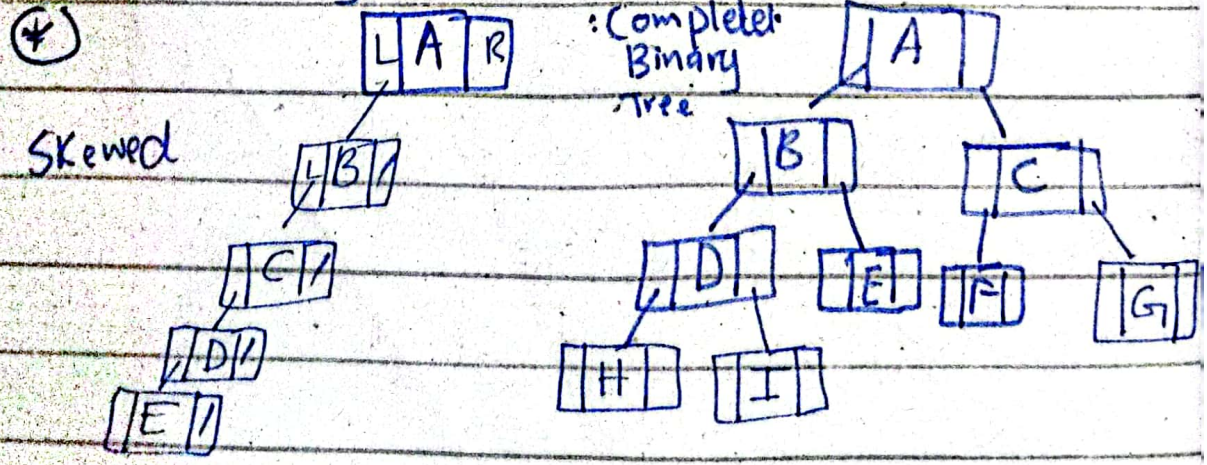


0	—
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H
9	I

(*) Tree ADT :

```
-> TreeNode()
    { int data;
      TreeNode * LeftChild;
      TreeNode * rightChild;
    }

-> Tree
    { TreeNode * root;
    }
```

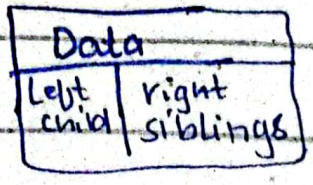
(*)



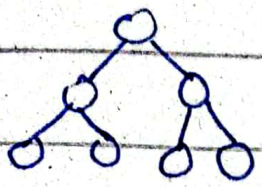
16/5/24

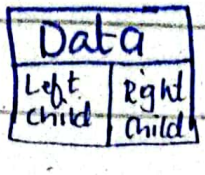
⊙ Tree and Graph have bit difference.

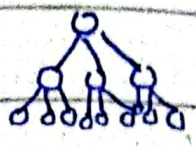
⊙ Tree:



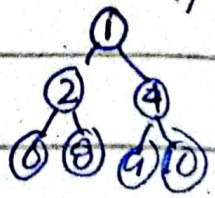
⊙ Binary Tree:






(Ternary Tree)

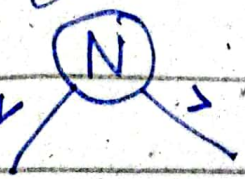
(BST) ⊙ Binary Search Tree:



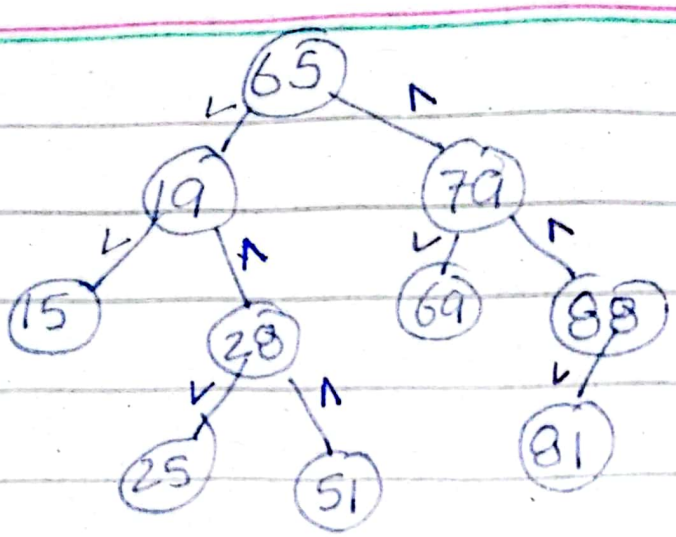
(1(2(6,3),4(9,10))

→ Binary Tree is said to be Binary Search Tree when it satisfy the following properties:

(i) The value of each Node 'N' is greater than every value in the left sub-tree of N and is less than every value in right sub-tree of N.

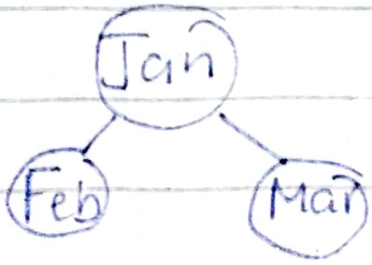


E.g: There is a binary search tree which satisfy it:



: BST should satisfy this property

- For months:



: Compare alphabetical order (ASCII value)

- Operations:
  - → Searching
  - → Insertion
  - → Deletion
  - → Traversing (Search or move through tree)

- TreeNode

```
{   int data;
    TreeNode * left;
    "  "  * right;
}
```

Tree

```
{
    TreeNode * Root;
}
```

→ Search (ITEM)

{

PTR = Root;

if (ITEM < PTR.data)

{ left subtree;

if (ITEM > PTR.data)

{ right subtree;

if (ITEM == PTR.data)

{ Return true;

}

}

BST: T
ITEM
ROOT

: Pseudo Code

: { PTR = Root, Flag = False;

while (PTR != Null && flag == false)

{

if (ITEM < PTR.data)

{ PTR = PTR.left;

}

if (ITEM > PTR.data)

{ PTR = PTR.right;

}

if (ITEM == PTR.data)

{ Flag = true;

}

}

: Code

```
If (flag == true)
    { return true;
    }
}
```

→ Insertion (ITEM)

```
{ PTR = Root, flag = false;
  PTR1 = Root;          (For Parent data)
```

: Search bring where new node to be inserted

```
while((PTR1 = NULL) && (flag == false))
    {
        if (ITEM < PTR.data)
            { PTR1 = PTR;
              PTR = PTR.left;
            }
        else if (ITEM > PTR.data)
            { PTR1 = PTR;
              PTR = PTR.right;
            }
        else if (ITEM == PTR.data)
            { Flag = True;
              return true;
            }
    }
    if (flag != True)
        { N = new NodeTree(ITEM);
```

if (flag == null)
{ return;
}

```
if(PTR.data > ITEM)
    { PTR-Left = N;
    }
else
    { PTR-Right = N;
    }
}
```

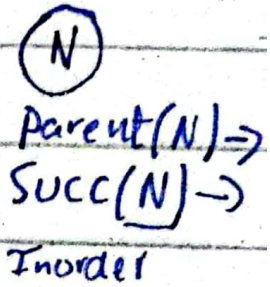
→ Deletion(ITEM)

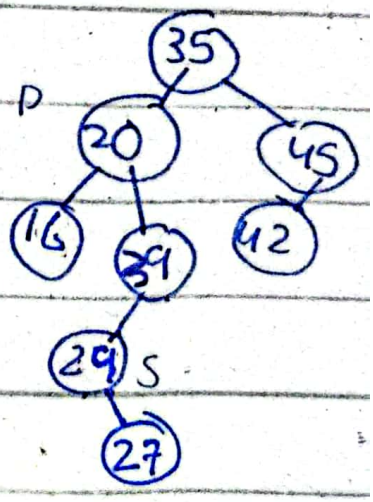
(null pointer) (i) N is leaf Node (27)

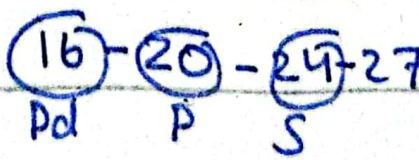
(Parent points to node child) (ii) N has 1 child (45)

(iii) N has 2 child (20)

(Parent node will be assigned successor) then check for successor node for leaf or 1 child and apply again case-I and case-II)







* In-order traversal can be done using inorder.

:successor:
subtree right → left most data

21/5/24

Searching
Insertion } → BST
Deletion

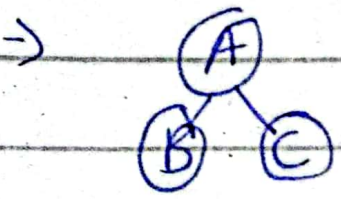
Traversing → Recursive Functions

* Traversing:

L → Move left
V → Visit
R → More Right

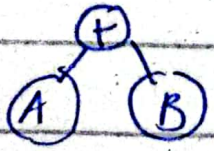
LVR
LRV
VLR
VRL
RVL
RLV

Left Node is visited first



Inorder : BAC
Preorder : ABC
Postorder : BCA

Inorder ← LVR
Postorder ← LRV
Pre-order ← VLR
(Three methods of traversing)



Inorder : A+B
Preoraer : +AB
Post order : AB+

:) Points of expression Tree:

(i) Convert to postfix expression
(ii) Create stack of TreeNode type

TreeNode
{ data;
Left
, right;

(iii) Read the expression from left to right until end of expression reached.

(iv) If the character is operand, create new node of that character and push to the stack.

: n_1 = new TreeNode(ch);
st.push(n_1)

: Else

ch == operator

t = new TreeNode(ch);
t_1 = s.pop(); : Takes two nodes and put it into tree
t_2 = s.pop();

t->Right = t_1, t->left = t_2;

st.push(t); : At last elements parent Node present

: Last element:
: st.pop();

(*) A + B - E * F * G

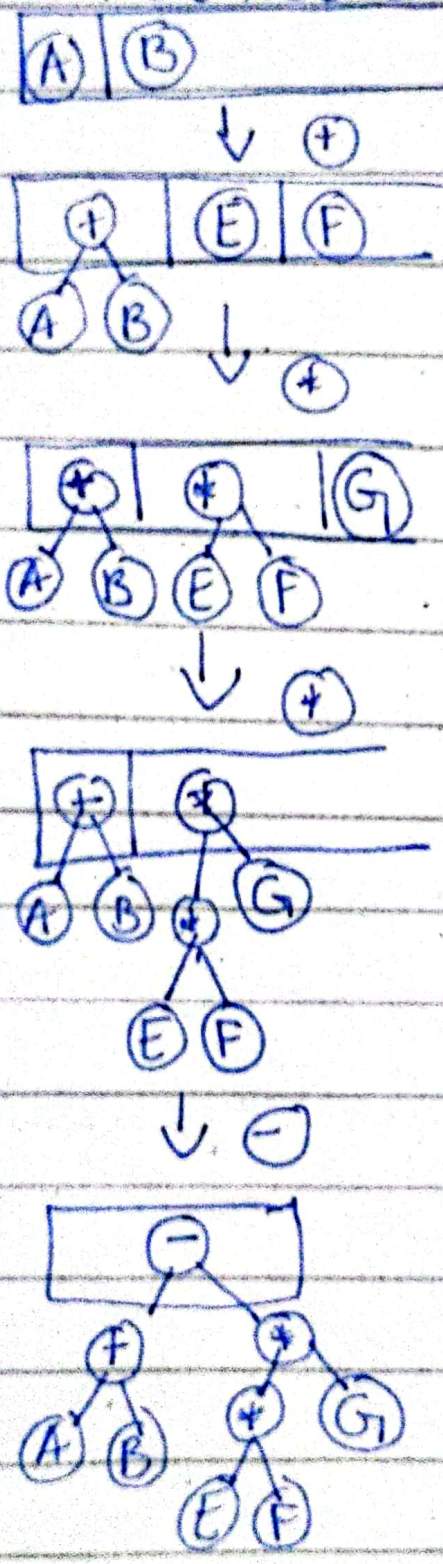
AB + - E * FG *

AB + E FG * * -

: AB + EF * G * -
Correct

AB+EF*G1*-

Tree Node



: Ch = A B +
E *

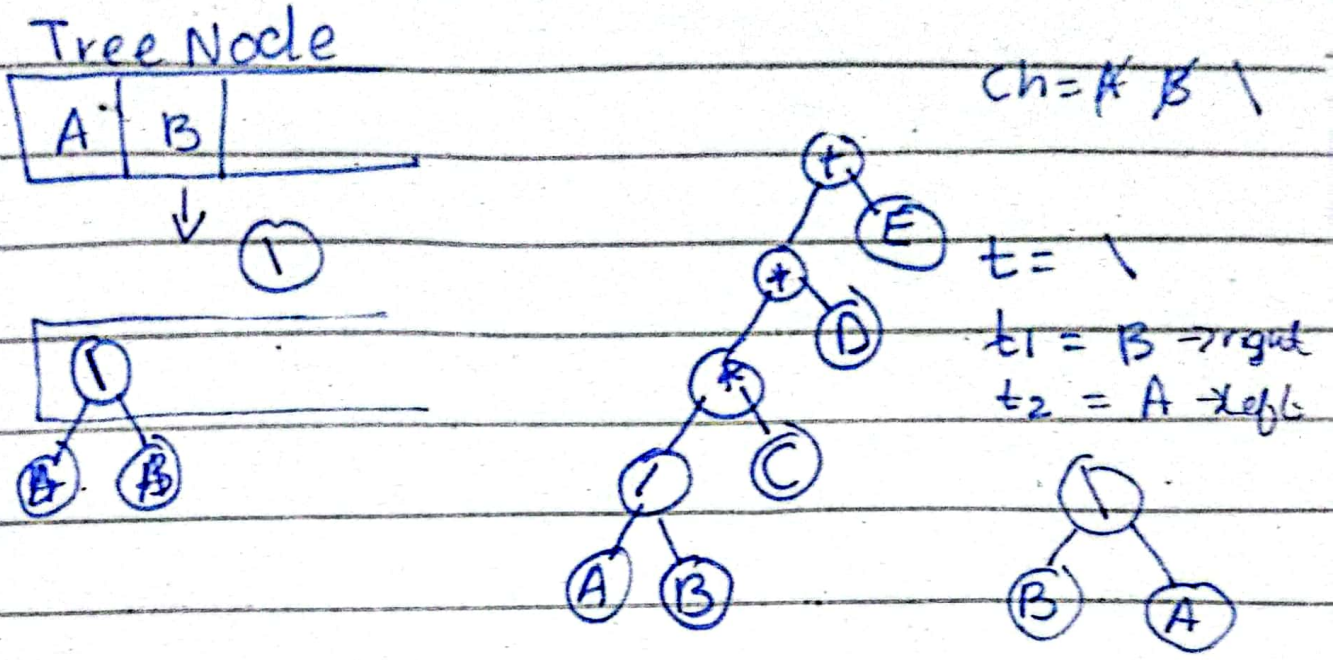
t1 = B
t2 = A

+

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

Postfix: AB*CD*E* : AB/C*D*E+
AB\CD (correct)

AB|C*D*E+



(*) Inorder(Tree Node * currNode)

{ If (currNode)

{ Inorder(currNode->left);

(print) Visit (currNode);

Inorder (currNode->Right);

} }

(*) Preorder(Tree Node * currNode)

{ If (currNode)

{ Visit (currNode);

Inorder (currNode->left);

Inorder (currNode->right);

} }

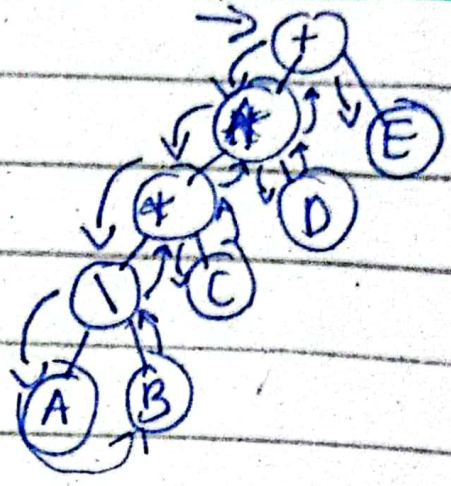
*E+

)

: VLR

⊙ Prefix:

+ * * \ A B C D E



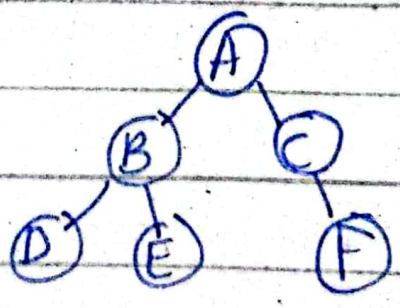
(LRV)

⊛ Postfix (currNode)

```
{ If (currNode)
  Inorder(currNode->left);
   Inorder(currNode->right);
    visit(currNode);
  }
}
```

Postfix: A B \ C * D * E +

⊛ without expression:



*: short cut (Preorder- right hanger Inorder- Center hanger Post order- left hanger)

LVR

⊙ Inorder: ~~B E~~ D B E A F C A

(VLR)

⊙ Preorder: ~~B D E A~~ A B D E C F

(LRV)

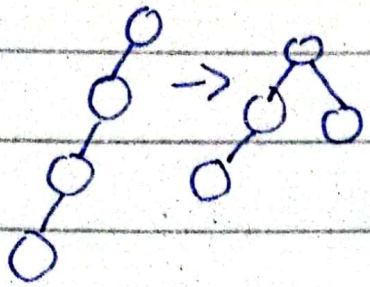
⊙ Post order: D B E F C A

⊛ Next Class Quiz on Traversal.

23/5/24

* (*) In-order of BST is sorted data.
(Faster algorithm for sorting).

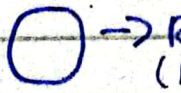
(*) Skewed form (Problem)
(Balancing of Tree)



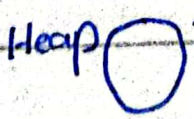
(*) Heap:

Tree data structure.
(Removing data in coustant time).

→ The element to be removed should be root element for heap.



→ Root data should be greater than all other data.

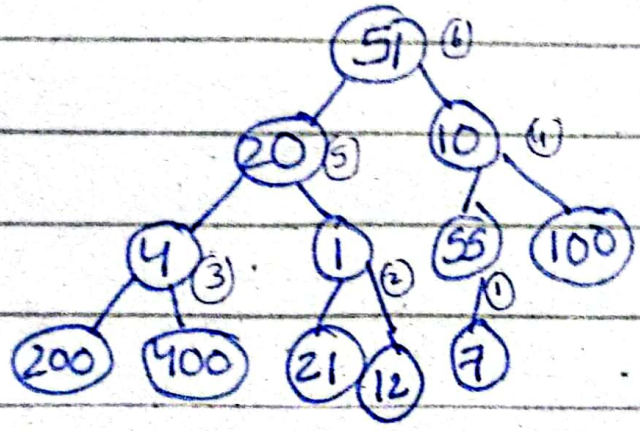


→ Root data smaller than other.

⊕ All the data to be made a heap data.

⊕ If left data is smaller than swap with greater (Max heap) and vice versa.

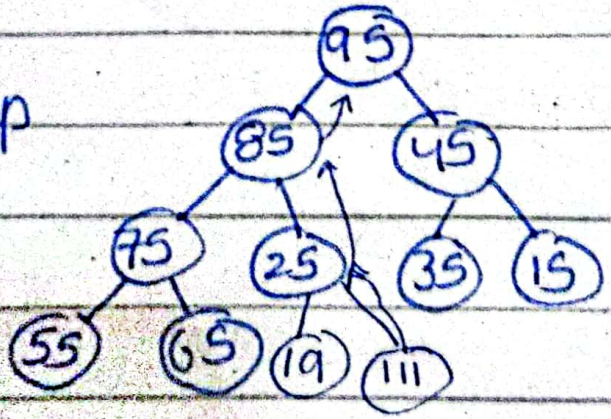
⊙ 51, 20, 10, 4, 1, 55, 100, 200, 400, 21, 7



: Heapify Process (Greatest element on top) (Then it will deleted)

: Internal (which has no leaf)

⊙ 95, 85, 45, 75, 25, 35, 15, 55, 65

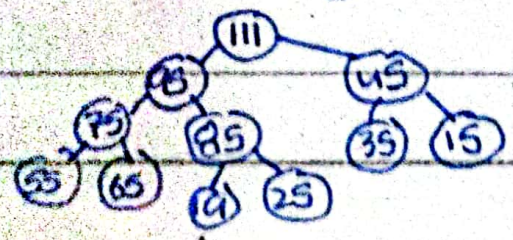


: Maxtleap

: Already heap satisfying

: Insert(19)

: Insert (111)



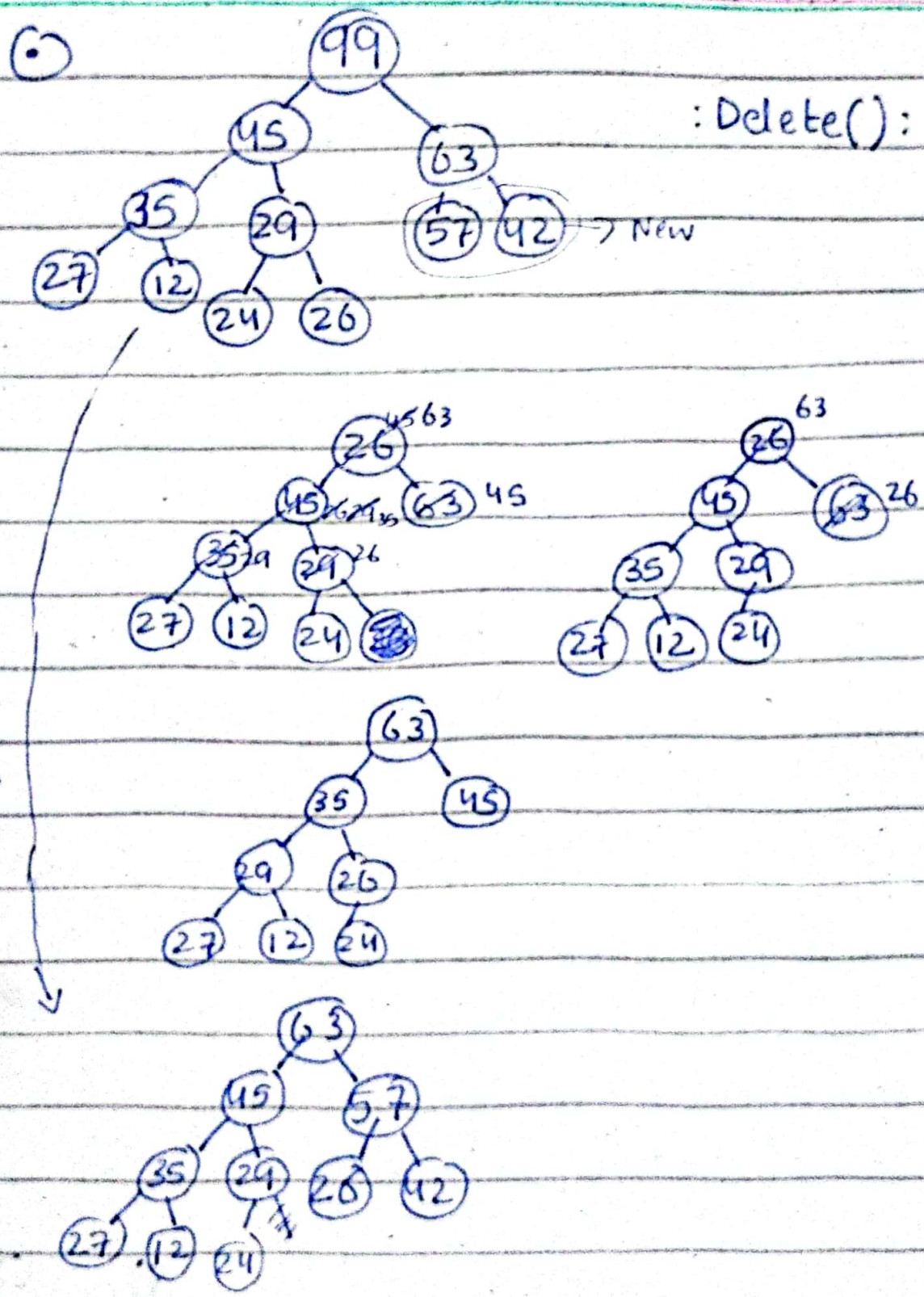
: Inserted (111)

: Delete() : top one root

replace root with last node and delete last node and then again follow heapify process

: Insert: Leaf → Root
Delete: Root → leaf

: Delete() :

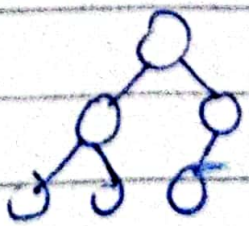


30/5/24

⊙ Height Balance Tree

$B.F = R_H - R_L$

$\{-1, 0, 1\}$



: Avial

⊙ Tree: Generalized tree

< 0 : Left heavy

> 0 : Right heavy

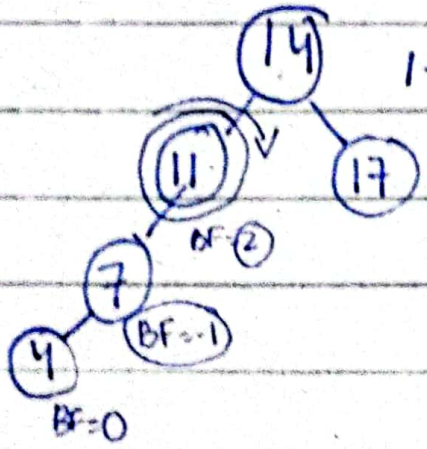
⊙ Binary

Heap

: Balance Factor

{ -1, 0, 1 }

→ 14, 17, 11, 7, 53, 4, 13

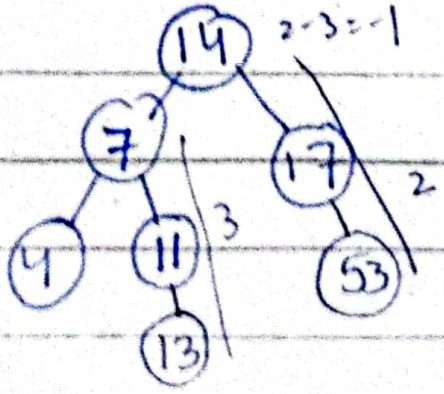


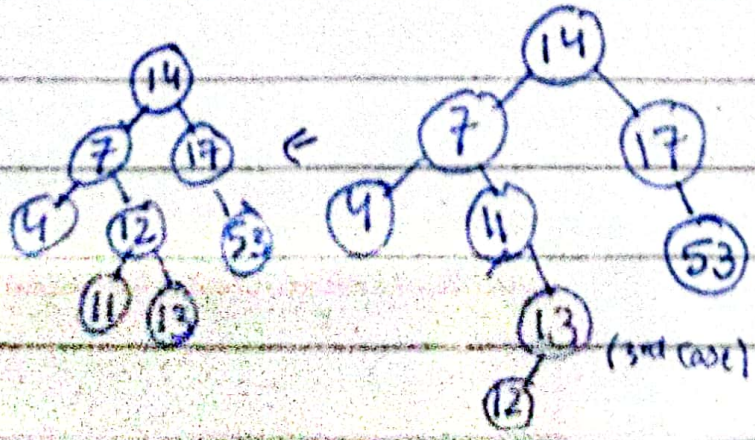
Rotate opposite side

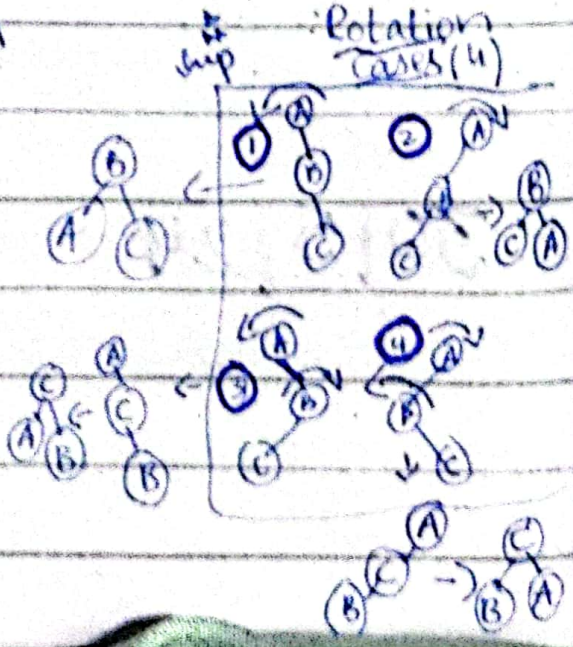
: checking Node Parent if balance factor is out or not.

: (Right Sub-tree) - (Left Sub-tree)

→ Rotating



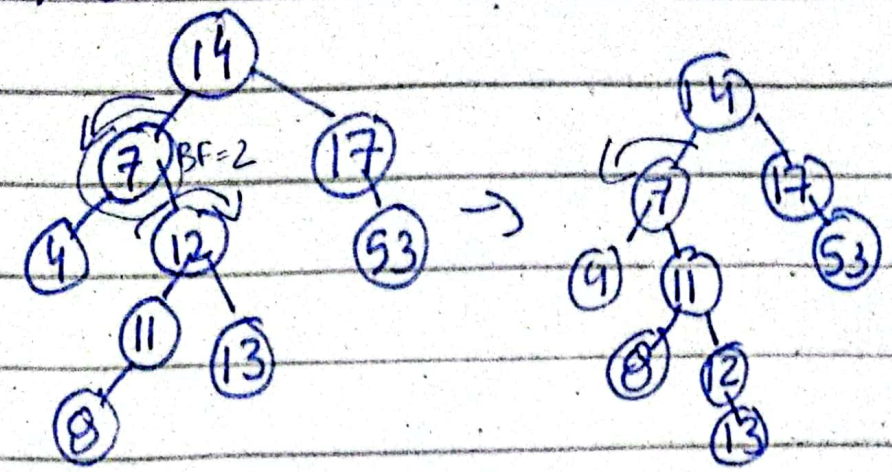


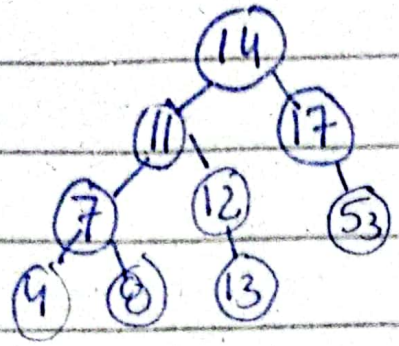


: BST (Important)

: 7:45 (Lecture on Monday)

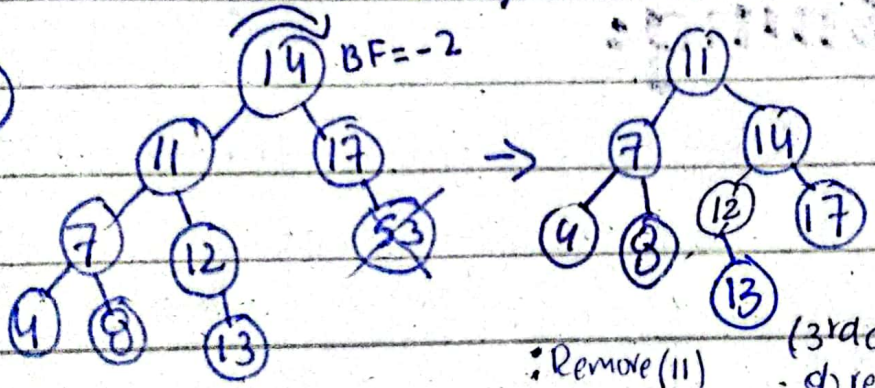
→ 12, 8 :





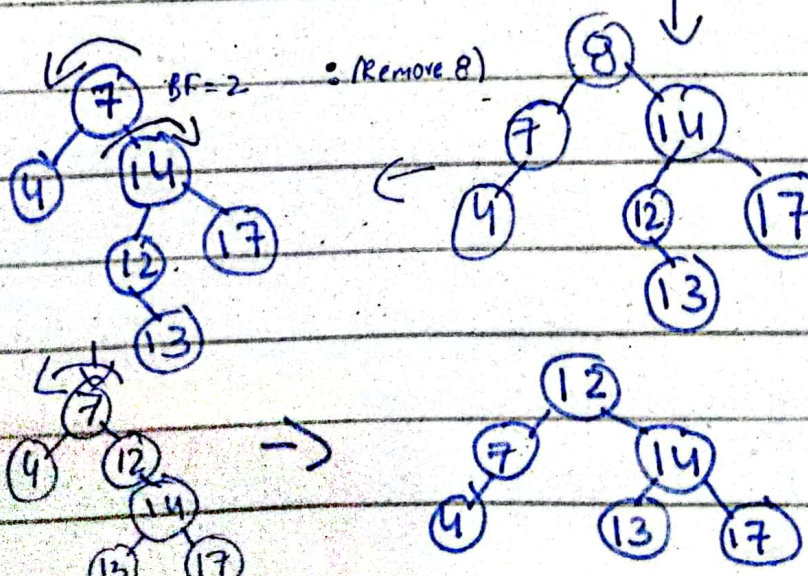
: Hardcode to add left child on previous right child and vice versa

: Remove (53).

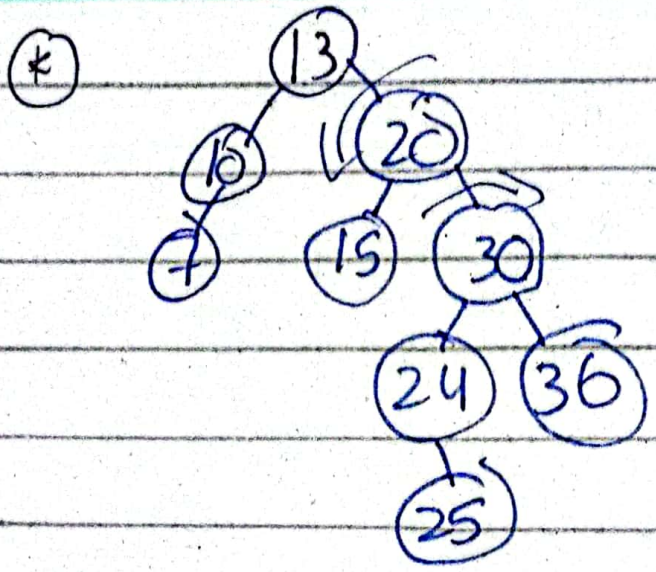


: Practice of Coding

: Remove (11) (3rd case of removing)



: In-order Predessor
: In-order success



3/5/24

* Trees → Graphs → Hashing.

* Hashing:

Concept from Hash table (An Array)

- Use to store key values.
  (K = i, i+1, ... n) → 
- → Save and Retrieve data in constant time
- → Generate mapping between key values and index.
  : H(K) → I
- → Implementation is important and easy of all Data structure

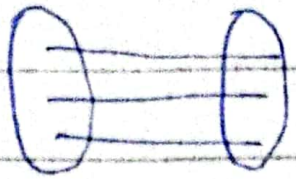
(*) 12, 13, 14, 15

$H(12) = (1+2) = 3$

$H(13) = 1+3 = 4$

Multiple key have index same

$\therefore 3+1 = 4$
$1+3 = 4$

(*)  Hashing.

→ Hash Table:

⊙ We have to place a key value into a location in the hash table. The location will be calculated from the key value itself.

→ Hashing:

⊙ To find the one-to-one correspondence between a key value and index in hash table where the key value will be placed.

⊙ **Properties:**

→ Easy to use → Unique ID

∴ : AVL, Graph, Hash, Linked List (Paper Questions)

E.g:

Hash Table = 10

0 9

10, 19, 35, 43, 62, 59, 31, 49, 77, 33

- Add the digit
- Take the unit place digit as index and ignore the decimal place digit.

0	19
1	10
2	-
3	49
4	59, 31, 77
5	-
6	33
7	43
8	35, 62
9	

collision ① ②

③

(3 collisions)
(1st data no collision)

K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

: 10

: Same Index (Collisions)

: 14

: 13

: 14

① Division Method

(Circular Function).

$K \bmod h \Rightarrow$ size of table

(h: Normally taken prime number)

(Solve collisions).

② Mid Square Method:

$(321^2 \rightarrow 7862)$

* 1234 2345 3456.

$K_1 = 1234$
$K_2 = 2345$
$K_3 = 3456$

1 3 2 4 3 5

$K_1 = 1234 \Rightarrow 525$
$K_2 = 2345 \Rightarrow 492$
$K_3 = 3456 \Rightarrow 933$

: Chances of collision is there

→ Collision Resolution Techniques:

① Closed Hashing (Linear Probing)
② Open Hashing (Chaining)

① Closed Hashing:

Element at index already exist, then place element at i+1. If again already exist then i+2 till size and then (i-1).

→ If we again come at the same place then no space for it.

⊙ Example:

0 ..... 9 (10 index)

$K \bmod(h) + 1$: function

For Quadratic Probing ($H(K)=i^2$) $h = 7$

15, 11, 25, 16, 9, 8, 12, 8

Solve by Linear Probing.

K	15	11	25	16	9	8	12
I	2	5	6	3	4	7	8

(Primary clustering) (collection of data at one place) (i+1)

→ Solution of Primary Clustering is quadratic probing:

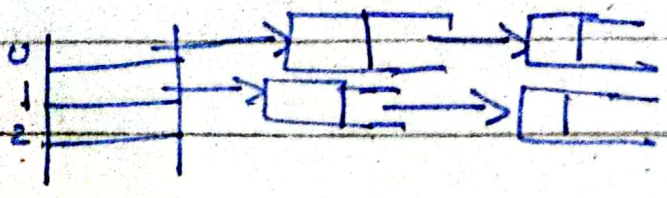
: $i+1^2$, $i+2^2$, $i+3^2$ .....

Ans ⊕ Do it with quadratic probing

⊛ Overflowing is still an error.

→ Open-Hashing:

Linked List on each index



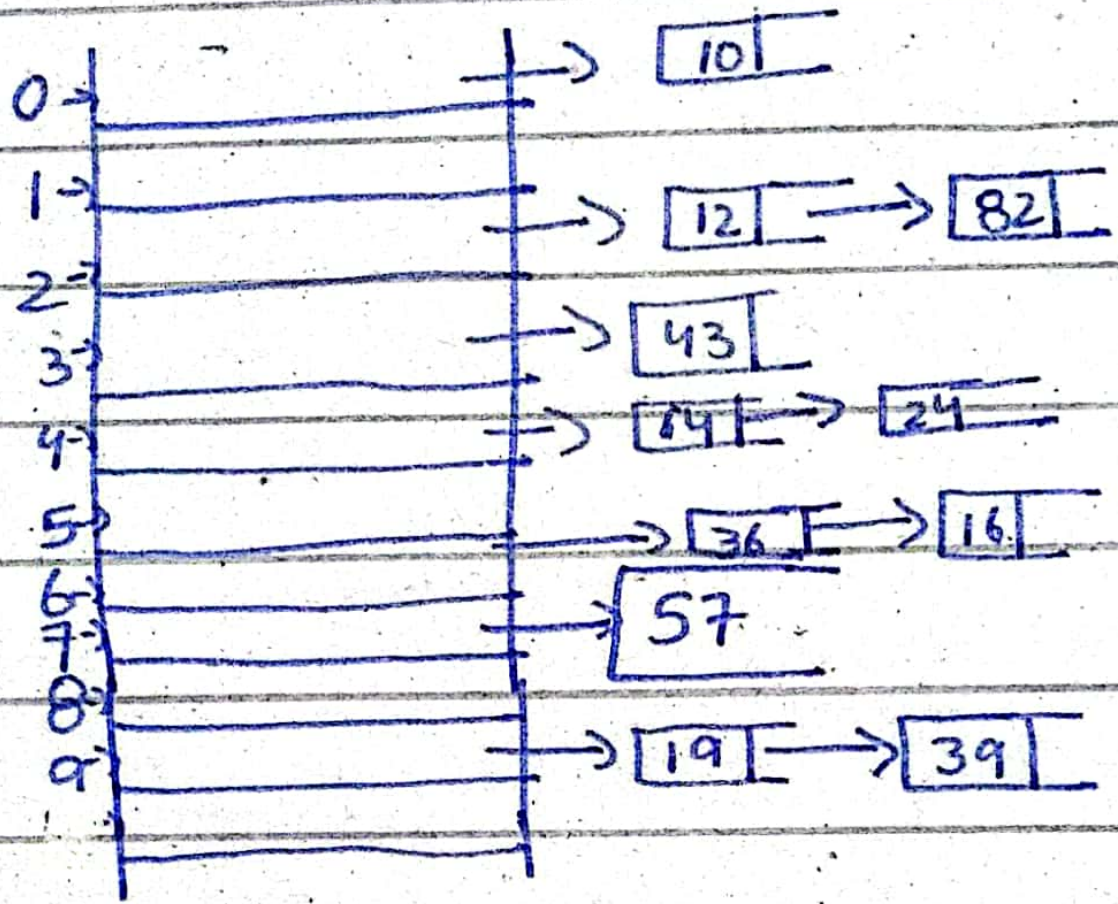
⊙Eg:

Data : 57, 36, 19, 39, 16, 10, 43, 64

12, 82, 24

Table Size = 10

: Right Side Index (Function)



: Collision and overflowing fixed.

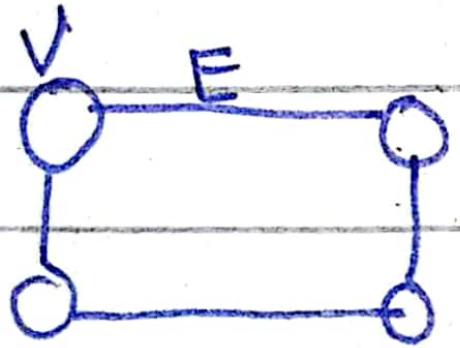
4/6/24

⊙ Graph:

Parent and child hare relation between them.

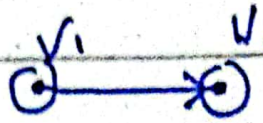
: Vertices or Nodes

· Edges or Arcs



: used for shortest path.

* Digraph : Directed graph



$(V_1, V_2) \neq (V_2, V_1)$

* Weighted graph :

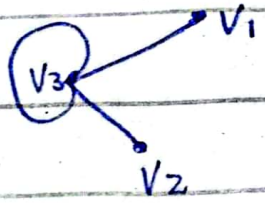
Labelled edges or paths

* Adjacent Vertices :

Source and Destination

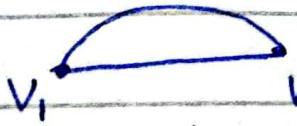
* Self loop :

Reaches some vertex



* Parallel edges :

Multiple ways or vertices of edges.



: Multi-Graph

-> If loop and parallel exist, it is multi graph.

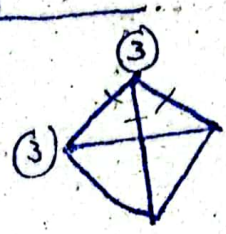
-> If no loop and not parallel it is simple graph.

16/20

⊙ Acyclic Graph:

If transversing, reaches same node or vertex. other is isolated (one edge not connected)

⊙ Degree of vertex:



⊙ Complete Graph:

Every Node is connected adjacent.

⊙ Connected Graph:

Every vertex not directly connected.

⊙ Path:

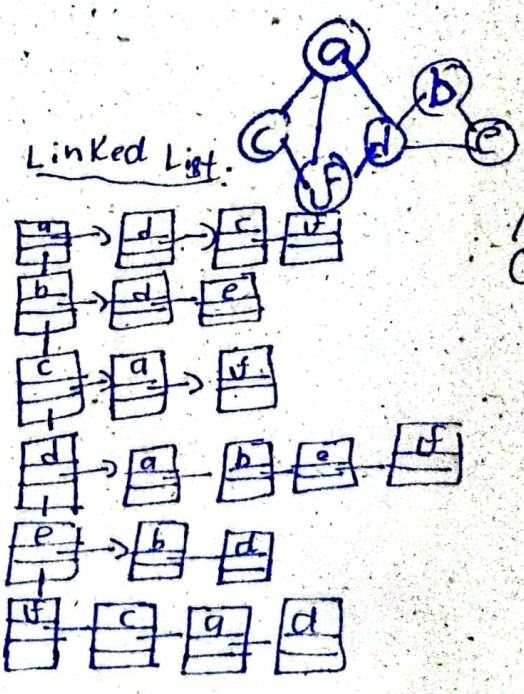
In a sequence of edges, to go from one edges to other.

⊙ Adjacency List.

Table

a	d c f
b	d e
c	a f
d	a b e f
e	b d
f	c a d

Linked List:



: Stack
Trees
Graphs
Hashing
Linked List
(Tutorials
Questions)

⊙ Adjacency Matrix

	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

⊙ Incidence Matrix

	ac	af	ad	df	cf	db	be	ed
a	1	1	1					
b	0					1	1	
c	1							
d								
e								
f								
g								

: Graph Most Portion of Paper

: Traversal Technique of Graph

: Visited-1
UnVisited-0

: Forward-Traversed

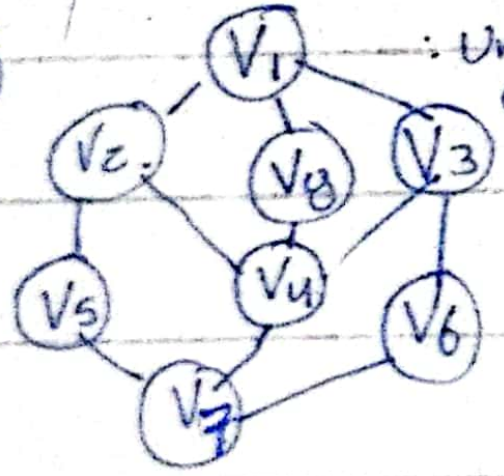
: Backward-Remaining

: Tree form after d is spanning

① ⊙ Depth First Search

② ⊙ Breadth First Search

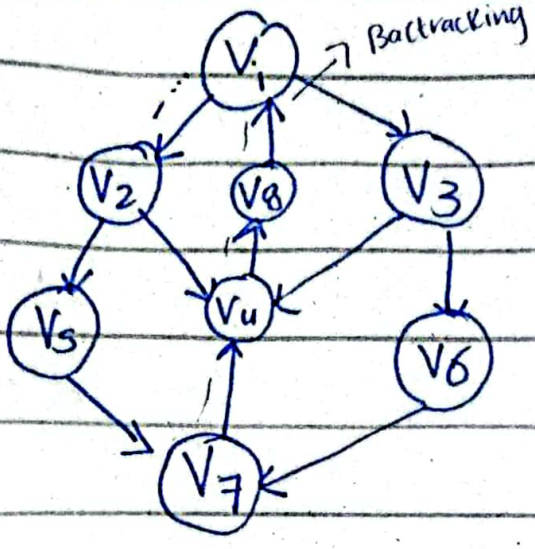
DFS ⊙



: Undirected Graph

⊙ $V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_7 \rightarrow V_4 \rightarrow V_8$
$V_q \leftarrow V_6 \leftarrow V_3 \leftarrow$

4/6/24



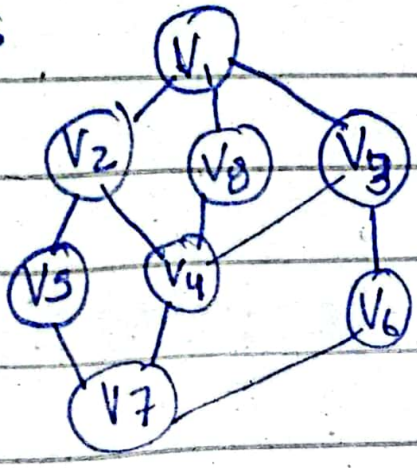
: $V_1 \to V_2 \to V_5 \to V_7 \to V_4 \to V_8 \to V_6 \to V_3$

: DFS : Helping Data Structure stack

⊙ BFS :

: All levels

: First Node first on each level

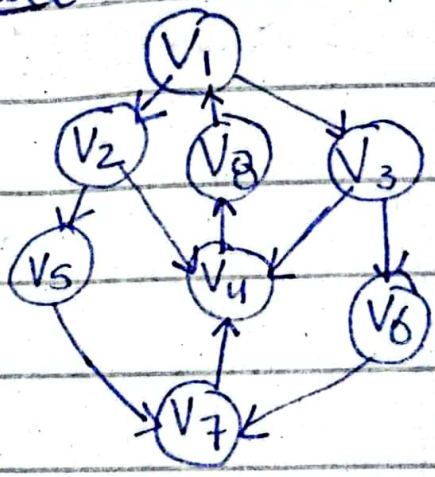


: Helping Data Structure Queue

: $V_1 - V_2 - V_8 - V_3 - V_5 - V_4 - V_6 - V_7$

(Undirected)

: Directed :



: $V_1 \to V_2 - V_3 - V_5 - V_4 \to V_8 - V_7 - V_6$