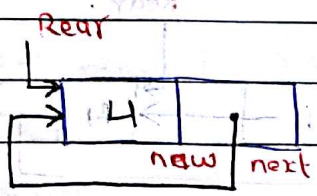
PAGE No.
DATE / /

* Circular Linked List

(1)



new → next = new

Rear = new

```
struct Node {
    int data;
struct Node* next;
};
    typedef struct Node* Nodeptr;
```

→ Insert in an empty list

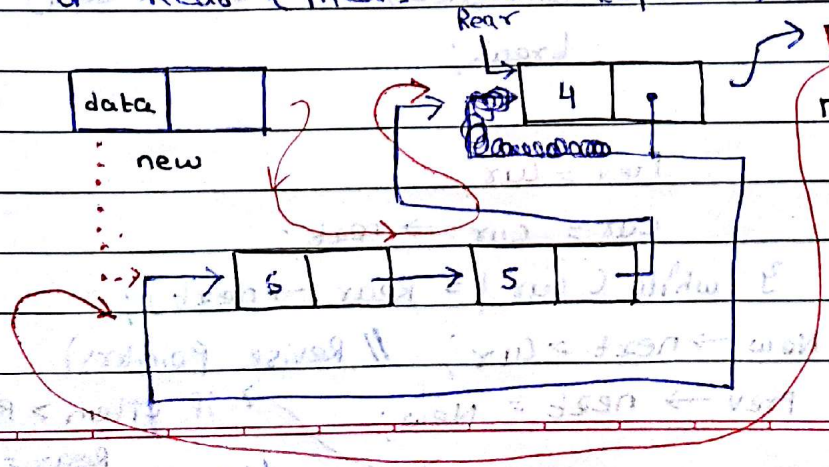
→ Import library / package

```
NodePtr New = new Node;
New → data = 10;
Rear = New;
Rear → next = Rear;
```

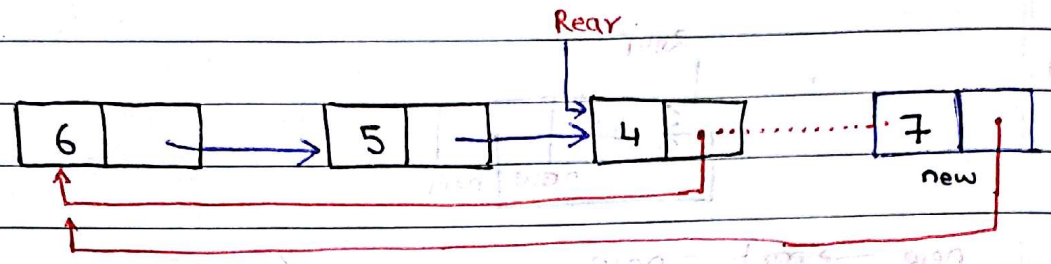
→ Insert in Front of Rear or Rear [tail] of Rear]

Front of Rear

⇒ Front of Rear (means at topmost)

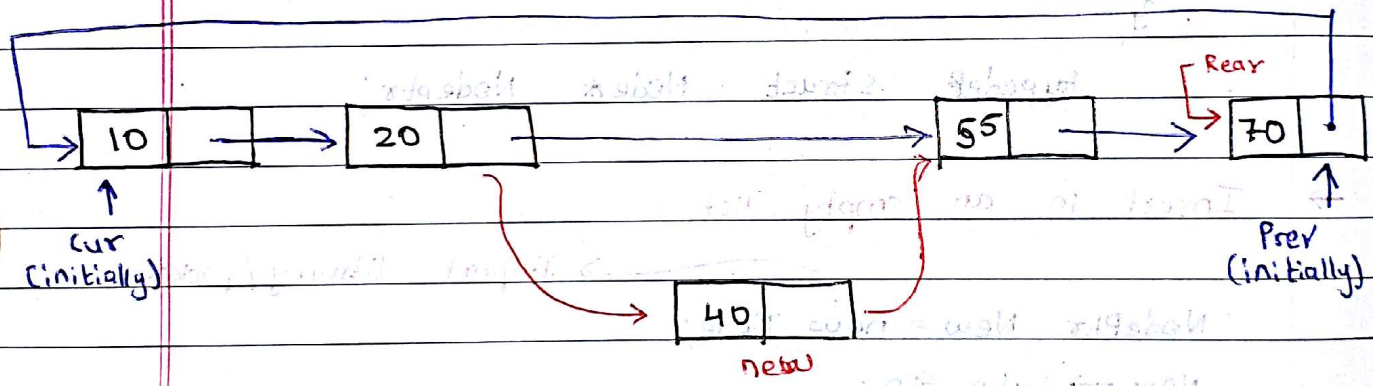


⇒ Tail of Rear




```
New -> next = Rear -> next
Rear -> next = new
```

→ Insert to middle of a Circular Linked List btwn Pre and Cur




```
Prev = Rear;
Cur = Rear -> next;

~~while ( cur -> data > new -> data)~~
do {
    if (item [new) <= cur -> data)
        break;

    Prev = cur;
    Cur = cur -> next;
} while ( cur != Rear -> next);
New -> next = cur;  // Revise pointers)
Prev -> next = New;
```

if (item > Rear -> data)
// revise rear ptr Rear = New; // if adding to end

free ≡ delete

PAGE No.
DATE / /

* Delete single node

```
if (cur == prev) {
        Rear = NULL;
        delete cur;
}
```

* Traverse the list

```
NodePtr cur;
if (Rear != NULL) {
   Cur = Rear -> next;
   do {
        cur = cur -> next;
   } while (cur != Rear -> next);
```

10 9 8 7 6 5 4 3 2 1

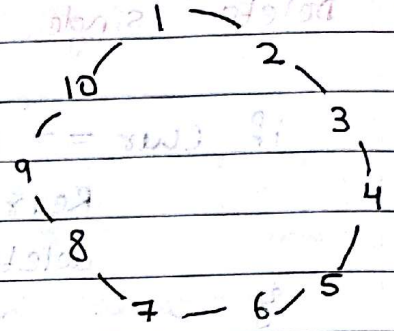
curr

PAGE No.
DATE

Rear

Prev

* Josephus problem



m = 2 ; n = 10

```
Kill ( NODE* rear)
{
    NODE* prev = rear;
    NODE* curr = (rear) → next);

    int Killctr = 0;

    while ( prev != curr)
    {
        if ( Killctr % 2 == 1 )
        {
            rear → next = curr → next;
            free (curr);
            curr = curr → next;
        }
        prev = curr;
        curr = curr → next;
        Killctr ++;
    }
```

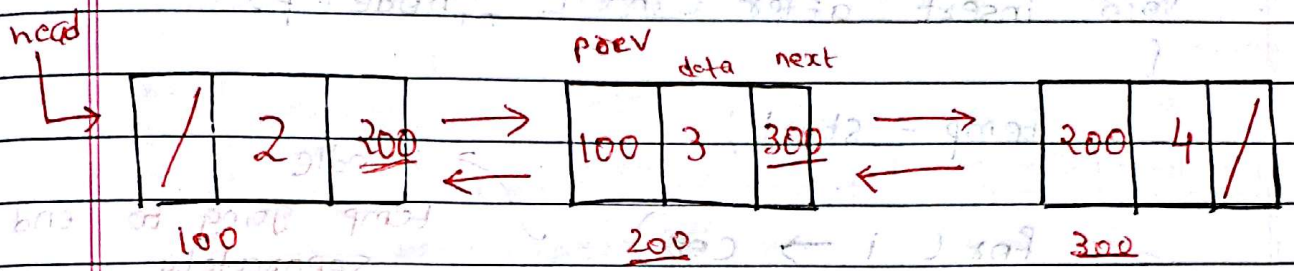
m (m-1)

Use temp variable

Also check for rear deletion

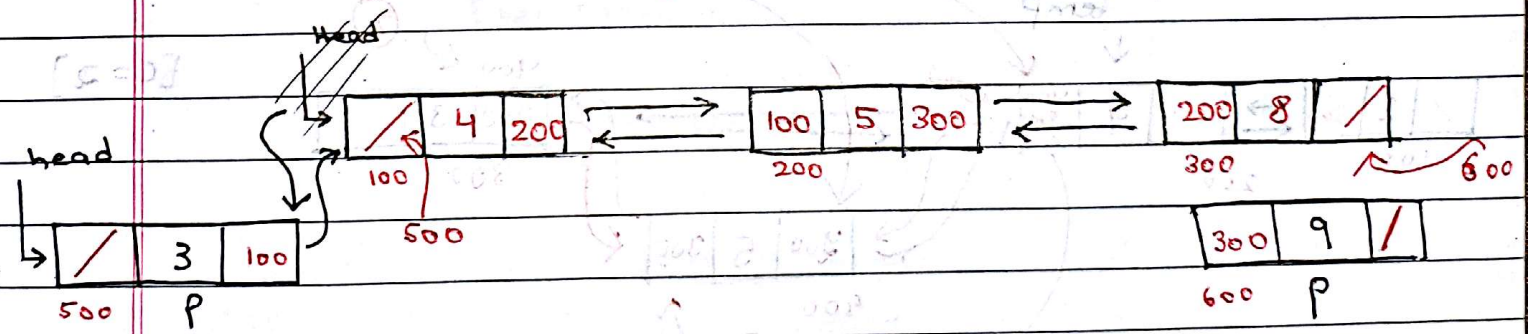
PAGE No. DATE / /

* Doubly Linked List




```
struct node
{
    int data;
struct node * next;
struct node * previous;
};
```

→ Insert at beginning




```
p -> next = head
head -> prev = p
head = p
```

→ Insert at end [temp variable]

```
temp = head
while ( temp -> next != NULL ) {
    temp = temp -> next;
}
temp -> next = p
p -> prev = temp
```

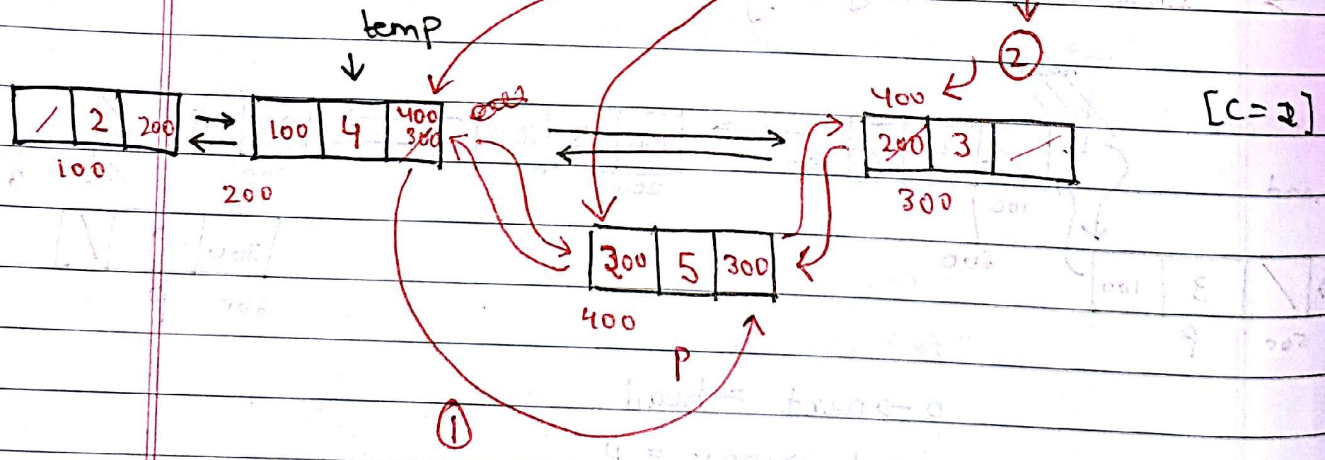
PAGE No.
DATE / /

→ Insert at count (post?)

```
void insert_after (int c , node *p)
{
    temp = start;
    for ( i → c)
    {
        temp = temp → next;
    }
```

→ Handle temp going to end separately

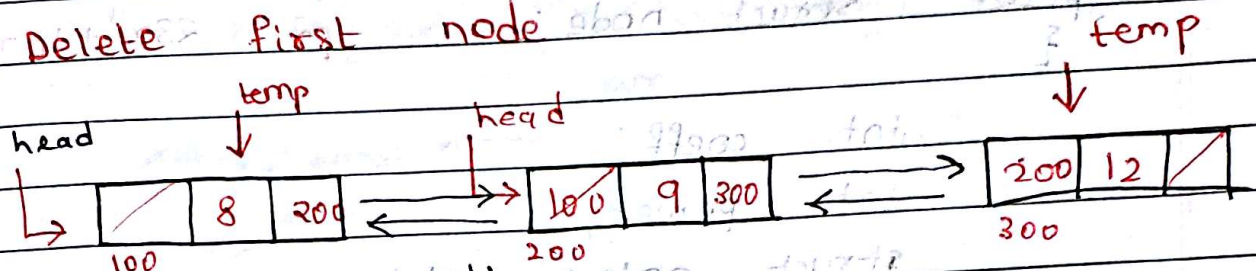
① p → next = temp → next;
② temp → next → previous = p;
temp → next = p;
p → prev = temp



PAGE NO.
DATE / /

→ Deleting a node :

• Delete first node




```
temp = head;
head = head → next;
head → prev = null;

free (temp);
```

• Traverse → Delete Last node

```
temp = last node after traverse
temp → prev → next = null;
free (temp);
```

PAGE No.
DATE / /

* Polynomials

$p1 = 8x^3 + 3x^2 + 2x + 6$
$p2 = 23x^4 + 18x - 3$

```
typedef   struct  node
 {
      int   coeff;
      int   power;
      struct  node *  next;
 } NODE;


int  main()
 {
      NODE   p1;
      NODE   p2;

      for ( )
      {
         Enter  coeff  &  power  for  p1 & p2
      }

      NODE *   head1 = null;
      NODE *   head2 = null;

      add ( p1, p2, head1, head2);
}


 add (                    )
{
     ForLoop ( ) {
          p1 → coeff  =  p1 → coeff  + p2 → coeff;
          p1 = p1 → next;
          p2 = p2 → next;
     }
```

returns (head1, head2) pointing to 1st NODE

1 2 3 4 5 6

6 5 4 7 2

PAGE No.

DATE / /

* Reverse a doubly LL

~~while (tp → next != NULL) {~~

or

~~Node* temp =~~

```
temp = NULL
current = head
while (current != NULL)
{
    temp = current → prev;
    current → prev = current → next;
    current → next = temp;
    current = current → prev; // Move to prev node
}

if (temp != NULL) {
    head = temp → prev;
}
```

swapping current node address

which is actually next because of swapping

PAGE No.
DATE / /

* Hash tables

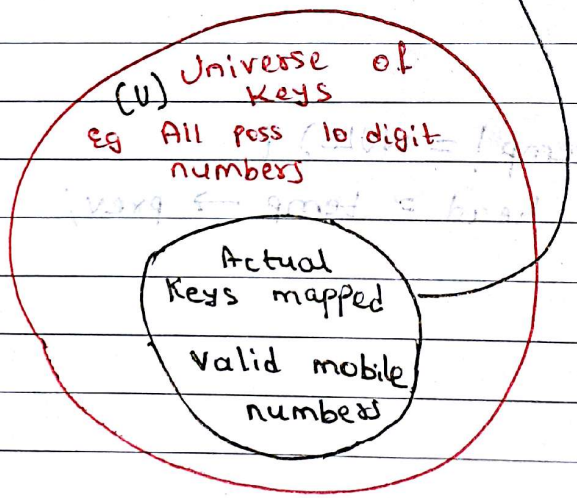
h(Key)

But h[Key] may shorten it out

Hash Function
[simple Arithmetic operations]

	Key	Address
UID →	2018 26 →	1000
	2018 27	2000
	2018 28	3000

3 keys each storing address of corresponding location where data is stored



But here all possible numbers were possible, but only valid UIDs are mapped

Open Addressing : Actual data is inside Hash table

Closed Addressing : Outside

→ Collision : Hash Functions may shorten (not always fails) two keys to same value, and collision occurs

Eg : 20181400 26 → N % 100 = 26

20191400 26

collision between h[k]

No of Keys

$|K| \leq m$ → No. of element in Hash table

$|K| > m$ → collision chances high

→ collision less but still may exist

Eg: Max Input : $2^2 = 4$ [0, 1] [U]

Hash table has 2 values only [Actual]

↳ But from which of these 4 values 2 has to be removed

→ If 1 0 0] No collision

But 0 1 1] Collison

↳ Here we consider unit's place as per program (may change)

* Handling Collisions

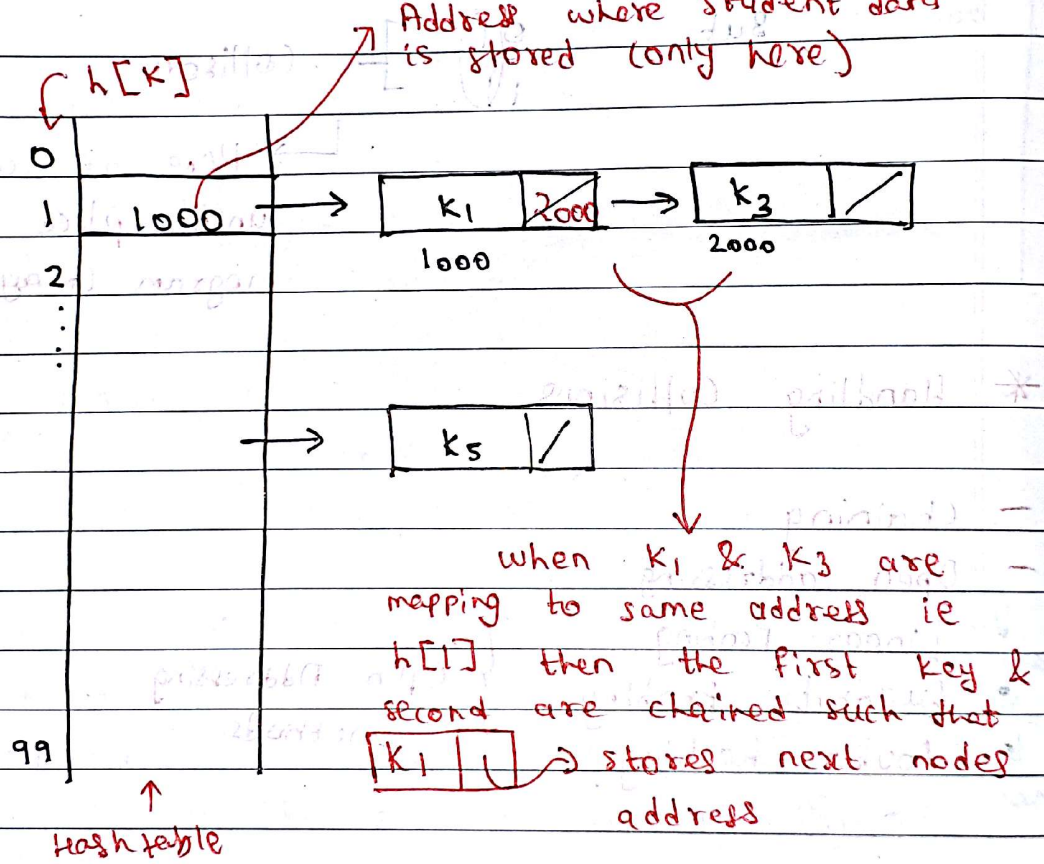
- Chaining
- Open addressing
  - Linear Probing
  - Quadratic Probing
  - Double hashing

} Open Addressing methods

PAGE No.
DATE / /

→ Chaining

Put all elements that hash to the same slot into a linked list.



Hash Function : The mapping of keys to indices of a hash table is called hash function.

Usually consists of :

- Hash code [map Keys]
- ~~Compression proop~~ [Non integer Keys → Integer]
  [Hash code map]
- Compression map

PAGE No.
DATE / /

→ Popular Hash-code maps

i] Integer Cast :

ii] Component sum : Key which has more than 4 bytes then you can take it in chunks of 32 bits and add them up.

* Compression Map

1] The Division Method.

$$h(K) = K \bmod m \qquad [K : 0 \to (m-1)]$$

m ← size

2] Multiplication Method

multiply

$$h(K) = \lfloor m(K \cdot A \bmod 1) \rfloor \quad ; \text{ constant } A,\ 0 < A < 1$$

i] x K by a const A

ii] extract the frac part of KA

iii] x frac part by m

iv] Take the floor of the result.

$$\lfloor m \cdot (KA - \lfloor KA \rfloor) \rfloor$$

→ Compression map called the Multiply, Add and Divide (MAD)

$h(K) = |ak + b| \bmod N$

N : size of table
a, b : some fixed nos.
a & N : has to be co-primes

a should not be a multiple of N [always b]

* Open Addressing

If we have enough contiguous memory to store all the keys ($m > N$) ⇒ store the keys in the table itself. No need to use linked lists anymore

Basic idea :
Insertion : If a slot is full, try another one, until you find an empty one
Search : follow the same sequence of probes
Deletion : more difficult.

Search time depends on the length of the probe seq

Better to have size of table as prime number

PAGE No. / DATE / /

→ Linear Poobe

Insert Keys :

K = 18 , 41 , 22 , 44 , 59 , 32 , 31 , 73

h(K) = K mode 13 m = 13

5 2 9 5 7 6 5 8

or h(K) = (K+i) mod m (13)

initially i=0

collision i=1 → 6

collision → 7 i=2 → 8

→ 6 → 7 → 8 → 9 → 10 i=5

→ 9 → 10 → 11 i=3

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	

i] Take the Key → mod by m = 13 place key in answer eg : 18 %. 13 = 5 Put 18 in 5

ii] If two mods give same answer eg : 44 %. 13 = 5 then start during collision counter (i = 1 → i++ (M-1)) %. 13 and start checking for emptly slot in the locations after it. Eg : 5 is occupied so go to 6

Probing

till empty slot found

PAGE No.
DATE / /

→ Searching

i] Let n = 44 be number to be searched Check for it at n %. 13 = 5 if number found return, else start probing in next locations.

ii] If number say n = 9 n %. 13 = 9 search for it at '9 th' location but actually the number is not present so on probing ~~we search~~ if we end on an empty slot, we say that the number doesn't exist.

iii] Also if all locations are filled the probe i exhausts from 1 → (m-1) & we say number doesn't exist.

→ Deletion

i] Suppose we want to delete 32
Now, 32 %. 13 = 6
↳ i = 1
i = 2
We search for 32 and delete it, But on making space empty ~~succeed~~ next search queries may return NULL
↳ so necessary measures have to be taken. Like replacing it with some flag

PAGE NO.
DATE / /

↗ Primary clustering

Clustering : Since probe increements by 1, tendency of program to store keys in clusters (ie close to each other) is high.
quadratic probing
↳ Hence double hashing, etc. is used to prevent this by generating random probes.
↳ so that keys are uniformly distributed.

* Quadratic Probing

$h(K,i) = [h'(K) + c_1 i + c_2 i^2] \bmod m$
where $h' : U \rightarrow (0, 1, \ldots, m-1)$

- Clustering problem is less serious but still an issue (secondary clustering)

(always remains same as non-collision linear probe value)
$[h'(K)]$

This was originally the $h(K)$, but collision occurred, hence probes are added to get $h(K,i)$

P.T.O

→ Keys = 10, 23, 31, 4, 15, 28, 17, 88, 59

$m = 11$; $c_1 = 1$

$c_2 = 3$

Linear : $h(K) = K \bmod m$

K :	10	23	31	4	15	28	17	88	59
K mod 11	10	1	9	4	4 → 5 (i=1)	6	6 → 7 (i=1)	0	4 → 8 (i=4)

0	1	2	3	4	5	6	7	8	9	10
88	23			4	15	28	17	59	31	10

Quadratic : $h(K,i) = (h'(K) + c_1 i + c_2 i^2) \bmod m$

$h(K,i) = [h'(K) + i + 3i^2] \bmod 11$

K :	10	23	31	4	15	28	17	88	59
	10	1	9	4	4 → 8 (i=1)	6	6 → 10 (i=1) → 9 (i=2) → 2 (i=3)	0	4 → 8 (i=1) → 7 (i=2)

$h(K,1) = 4 + 1 + 3 = 8$

$h(K,1) = 6 + 1 + 3 = 10$

$h(K,2) = (6 + 2 + 12) \bmod 11 = 9$

$h(K,3) = (6 + 3 + 27) \bmod 11 = 2$

$h(K,1) = 8$

$h(K,2) = 4 + 2 + 12 = 18 \bmod 11 = 7$

0	1	2	3	4	5	6	7	8	9	10
88	23		17	4		28	59	15	31	10

PAGE No.
DATE / /

* Double Hashing

1] Use one hash fun^{n} to determine the first slot
2] Use a second hash fun^{n} to determine the increement for the probe sequence.

$$h(K,i) = [h_1(K) + i\,h_2(K)] \bmod m \quad , \quad i = 0, 1, \ldots$$

Advtg : = avoids clustering
Disadvtg : harder to delete an element

Can generate m^2 probe sequences maximum

→ Let $h_1(K) = K \bmod 13$

$h_2(K) = 8 - (K \bmod 8)$

Keys : 18, 41, 22, 44, 59, 32, 31, 73

K:	18	41	22	44	59	32	31	73
$h_1(K)$ =	5 2	9		5	7	6	5	8
$h_2(K)$ =				$8-(4)$ $= 4$			$= 8-7$ $= 1$	$= 8-1 = 7$
				$i=1, 9$			$i=1;\ 6$	$i=1$ $15 \to 2$
				$i=2, 0$			$i=2;\ 7$	$i=2$ $22 \to 9$
							$i=3;\ 8$	$i=3$ $29 \to 3$

0	1	2	3	4	5	6	7	8	9	10	11
44		41	73		18	32	59	31	22		

PAGE No.
DATE / /

→ If m : powers of 2
h_2 : odd nos.

→ If m : is prime
h_2 : +ve int less than m

Eg : $h_1(K) = K \bmod 13$ → Prime
$h_2(K) = 1 + (K \bmod 11)$
→ max $10 + 1 = 11$

Bad idea

Choose this
Because range of nos (ie remainder) generated is high.

⇒ $h(K) = K \,\%\, m$

10	1	0	1 0
12	1	1	0 0
15	1	1	1 1

2^2

0
1
2
3

10 %. 4 : 2
12 %. 4 : 0
15 %. 4 : 3

Remaining digits have no use hence

PAGE No.
DATE / /

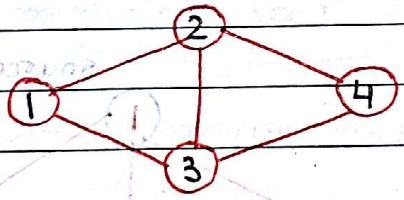
* Graphs

Graph G is defined as G = (V, E)

V ≡ set of vertices
E ≡ set of Edges

Edges may or may not have weights.
↓ (Like font-weight)
Factors on which the edges may depend

→ Representation of Graphs



n = 4 , m = 5

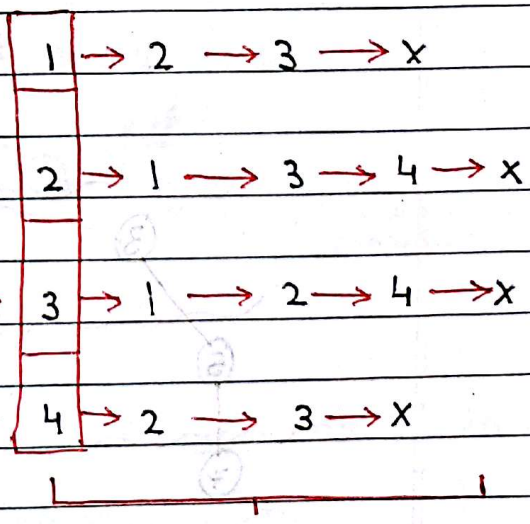
Adjacency Matrix :

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	0

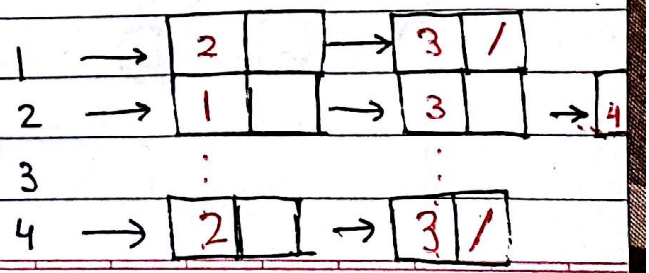
Edge NOT Present btw^n 1, 4

Edge Present between 2, 4

Adjacency List :



Adjacency relationship



PAGE No.
DATE / /

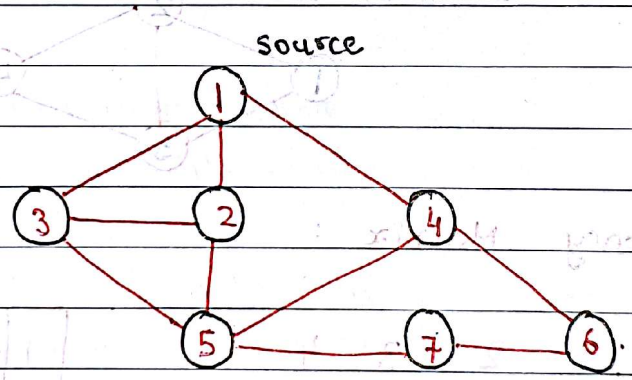
- Adjacency list representation is usually preferred since it is more efficient in representing sparse graphs.

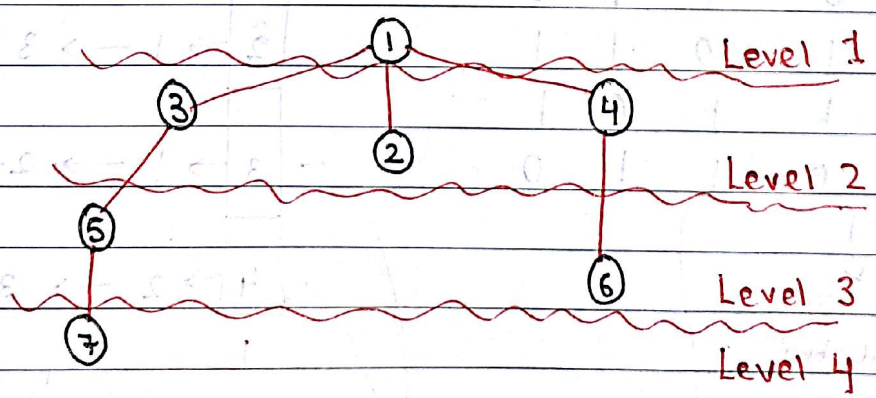
→ Graphs for which $|E|$ is much less than $|V|^2$

- Adjacency list req memory of the order of $\theta(V+E)$

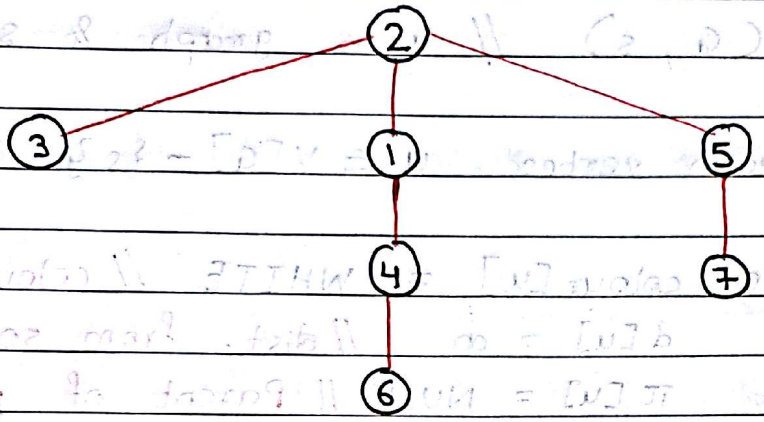
* Breadth First Search (BFS)

This search systematically explores the edges of G to discover every vertex that is reachable from s.





If (2) is source



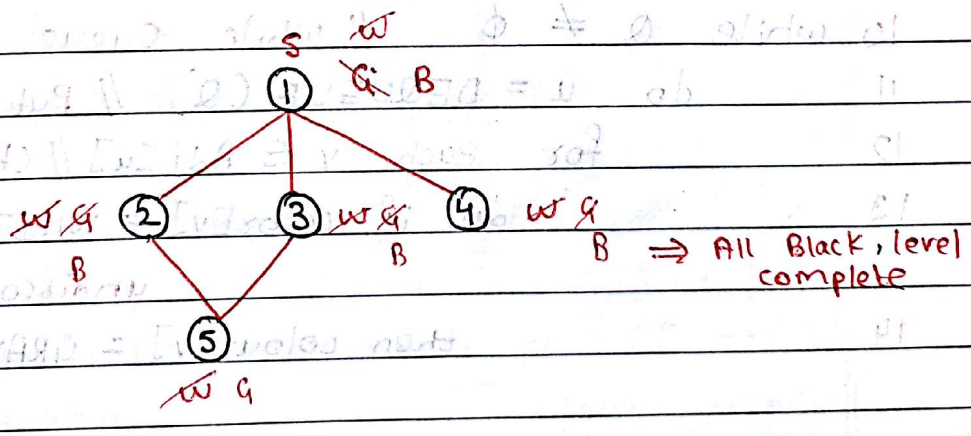
⇒ Reachability of all numbers from source (s)

white : Not discovered [Initially all white]

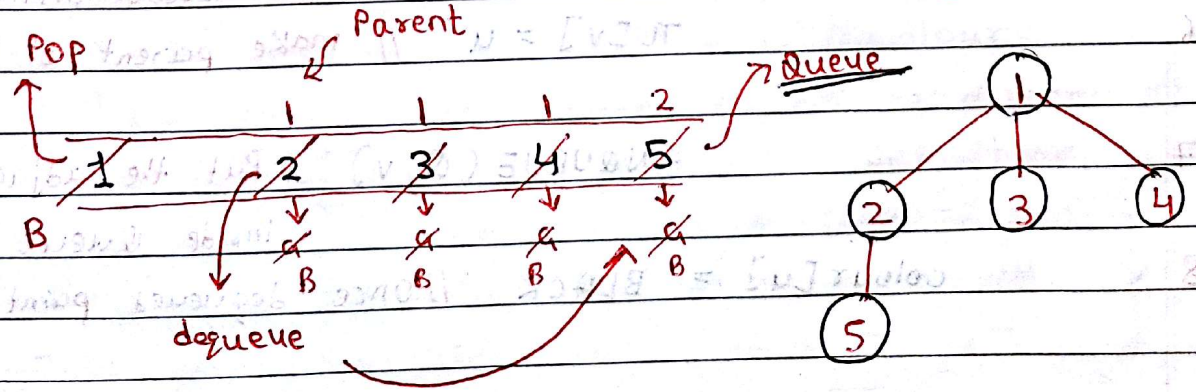
grey : Just now exploring/ node discovering [source : grey]

go to those nodes which are white

black : Visited



white : Not yet entered in Queue



PAGE No.
DATE / /

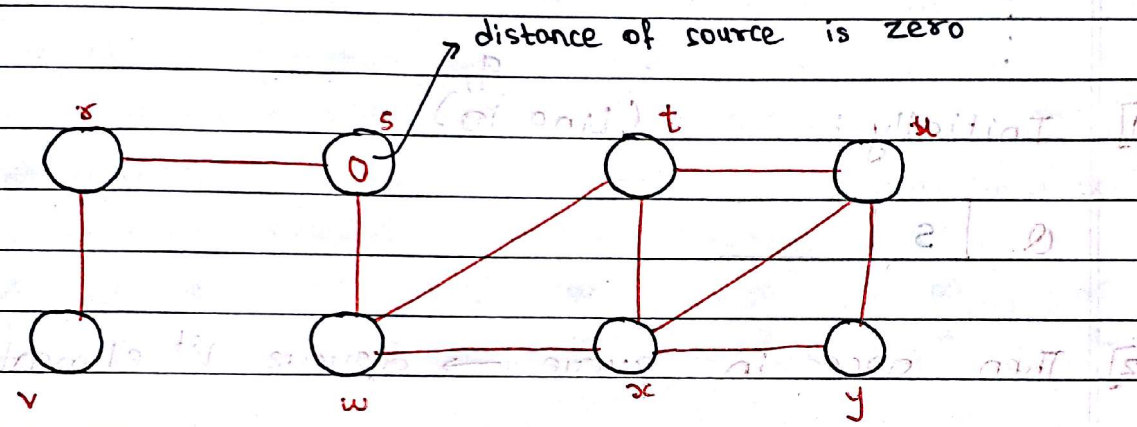
⇒ Algorithm :

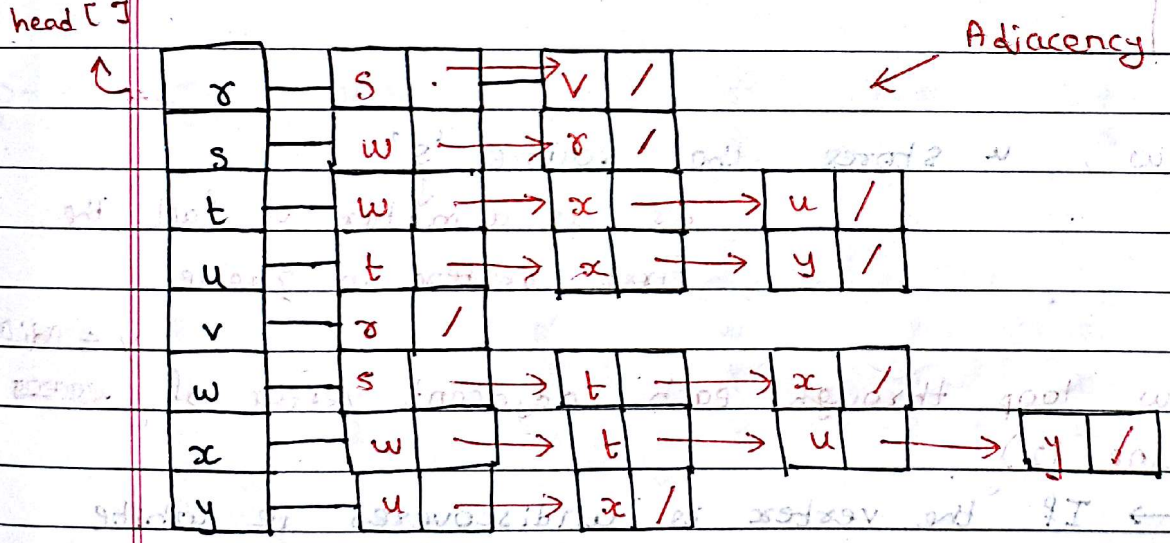
starting node
↑
BFS (G, s) // G is graph & s is the source

1 for each vertex $u \in V[G] - \{s\}$
2 do colour [u] = WHITE // colour of vertex u
3 d [u] = ∞ // dist. from source s to vertex u
4 π [u] = NULL // Parent of u
5 Color [s] = GRAY // Make source Gray
6 d [s] = 0 // Make distance 0
7 π [s] = NULL // Make parent NULL
8 Q = ϕ // Initialize Queue
9 ENQUEUE (Q, s) // Put source inside Queue
10 while Q ≠ ϕ // while Queue is not empty
11 do u = DEQUEUE (Q) // Put First element of Q in u
12 for each v ∈ Adj [u] // Check all adjacents of u
13 do if color[v] = WHITE // discover the undiscovered adjacent vertices
14 then colour[v] = GRAY // enqueued whenever Painted gray
15 d[v] = d [u] +1 // increase distance
16 π [v] = u // make parent of v as u
17 ENQUEUE (Q, v) // Put the adjacents of u inside Queue
18 colour [u] = BLACK // Once dequeued paint them black

It is so named because: It discovers all vertices at distance K from s before discovering vertices at distance K+1

→ All vertices adjacent to black one are visited





Each vertex has { int K; // Data
int c; // colour
int p; // it's parent
int d; // dist from source }

	source						
r	s	t	u	v	w	x	y
dist → ∞	0	∞	∞	∞	∞	∞	∞
p → –	–	–	–	–	–	–	–
	↑ NULL						

Initially

PAGE No.
DATE / /

- We intially need an empty Queue [FIFO]

→ Front = rear = null

then when source is enqueued

→ front = rear = source

then when r is enqueued

front = s ; rear = r

⋮

1] Initially : (Line 10)

Q | s |

2] Then once in queue → dequeue 1st element (Line 11)

Q |

3] Now, u stores the source 's'
or as a matter of fact the first vertex in queue.

4] Now loop through each adjacent vertex of (Line 12)

↳ If the vertex is undiscovered ie white then make it Gray (Line 14)

↳ Now increase it's distance = distance of parent + 1 (Line 15)

↳ Make v's parent u (Line 16)

↳ Enqueue v into the Queue

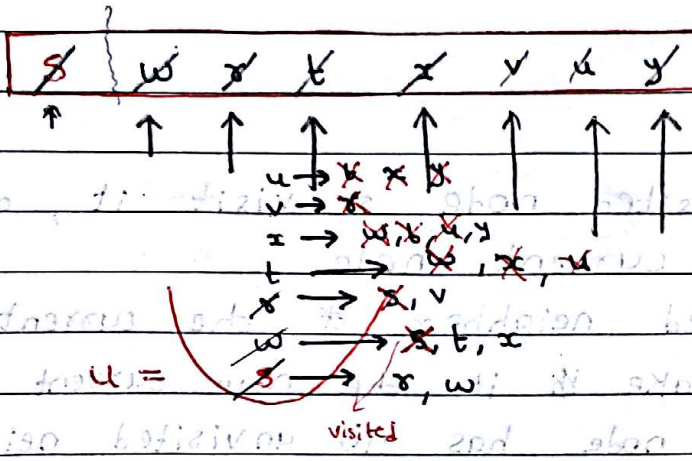
↳ Do this for all adjacents

5] One all adjacents are checked or enqueue ie discovered only then make the u ie parent black

PAGE No.
DATE / /

Queue:

Q



	s	w	r	t	x	v	u	y
d →	∞	∞	∞	∞	∞	∞	∞	∞
	1	0	2	3	2	1	2	3
p →	⊥	NIL	t	r	t	r	x	t
	s		w	t	s	w	x	x
c →	∅	∅	∅	∅	∅	∅	∅	∅
	⊥	⊥ (Initially)	⊥	⊥	⊥	⊥	⊥	⊥
	2	2	2	2	2	2	2	2

PAGE No.
DATE / /

* Depth First search (DFS)

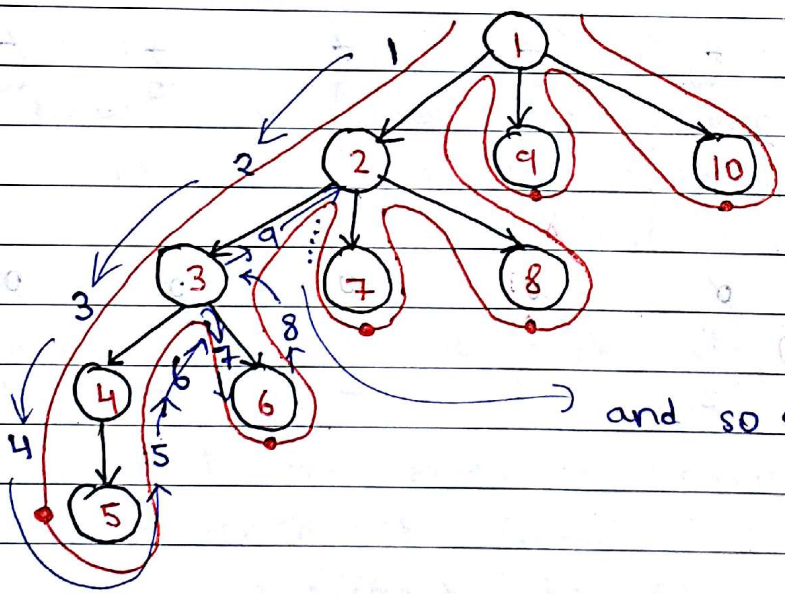
Rules :

i] Select an unvisited node s, visit it, and treat as the current node.

ii] Find an unvisited neighbor of the current node, visit it and make it the new current node.

iii] If the current node has no unvisited neighbours, backtrack to the it's parent, and make that the new current node; Repeat the above two steps until no more nodes can be visited.

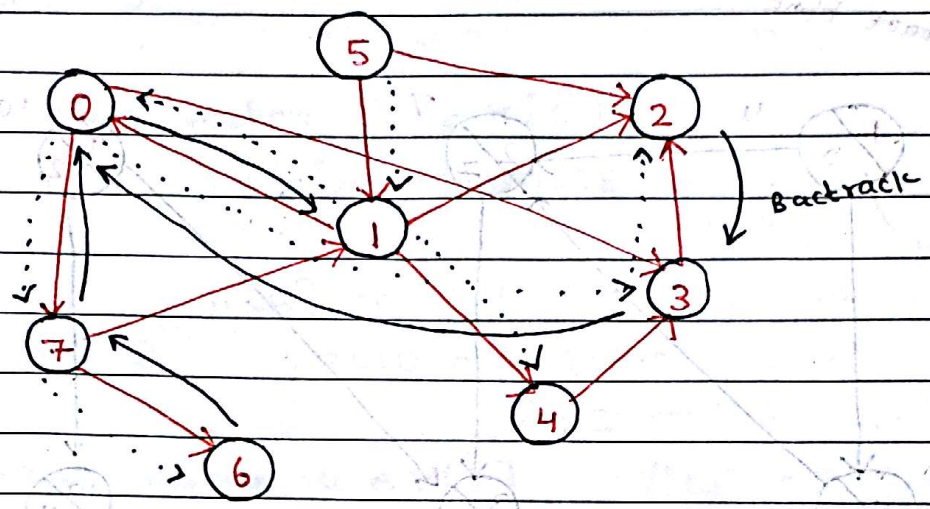
iv] If there are still unvisited nodes, repeat from step1



→ Go depth first until reach : leaf node
backtrack till parent has child
↳ If yes go to child and backtrack.
↳ If no backtrack.

PAGE No.
DATE / /

Example : start with Node 5



5 1 0 3 2 7 6 4

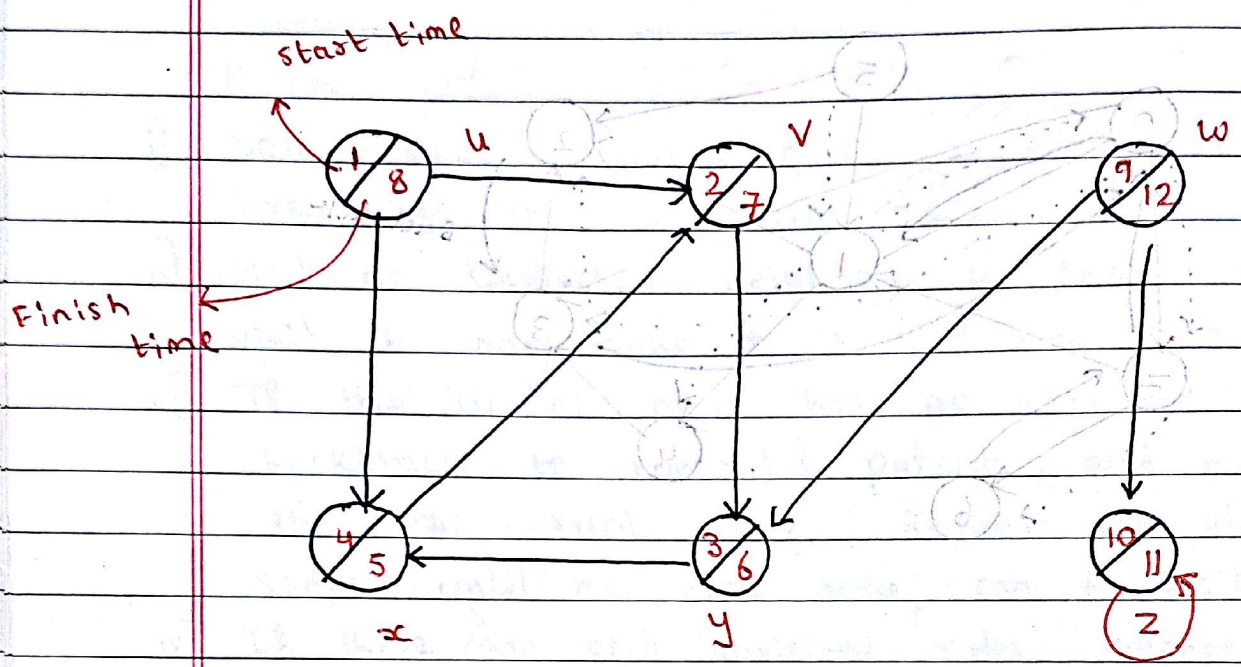
(: start time

) : Finish time

PAGE No.

DATE / /

→ Example



1	2	3	4	5	6	7	8	9	10	11	12
(	(	(	(	)	)	)	)	(	(	)	)

Strongly connected components

PAGE No.
DATE / /

⇒ Algorithm :

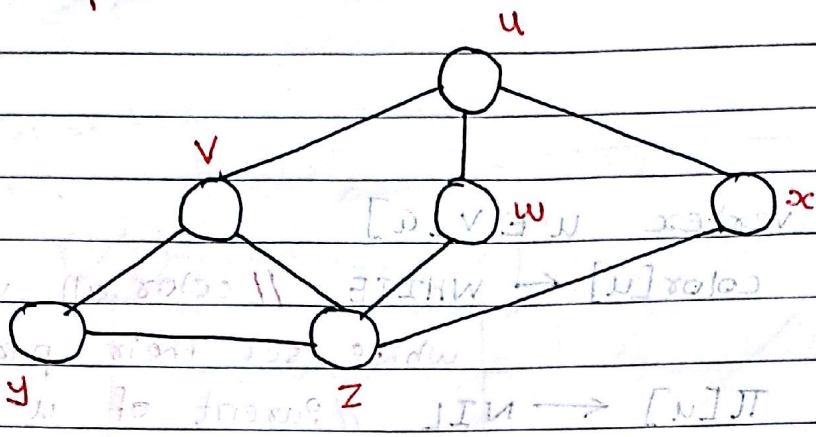
DFS (G)

```
1  for each vertex u ∈ V[G]
2        do color[u] ← WHITE   // color all vertices white, set their parents NIL.
3           π[u] ← NIL         // Parent of u
4  time ← 0                    // zero out time
5  for each vertex u ∈ V[G]    // call only for unexplored vertices
6        do if color[u] = WHITE // this may result in multiple sources.
7           then DFS-VISIT(u)
```

DFS-VISIT(u)

```
color[u] ← GRAY    // whiten the vertex u have just discovered
time ← time + 1
d[u] = time        // record the discovery time

for each v ∈ Adj[u]    // Explore edge (u,v)
    do if color[v] = WHITE
          then π[v] ← u        // set the parent value
               DFS-VISIT(v)    // recursive call

color[u] = BLACK       // Blacken u, it is finished
f[u] >>> time ← time + 1
```

→ Example



for (u — z)

DFS (u) — for (v, w, x)

DFS (v) — for (y, z)

DFS (y) — for (z)

DFS (z) — for (w, x)

DFS (x) — for (NULL)

DFS (w) — for (NULL)

→ Enqueue & Dequeue happen only once for each node : O(V)

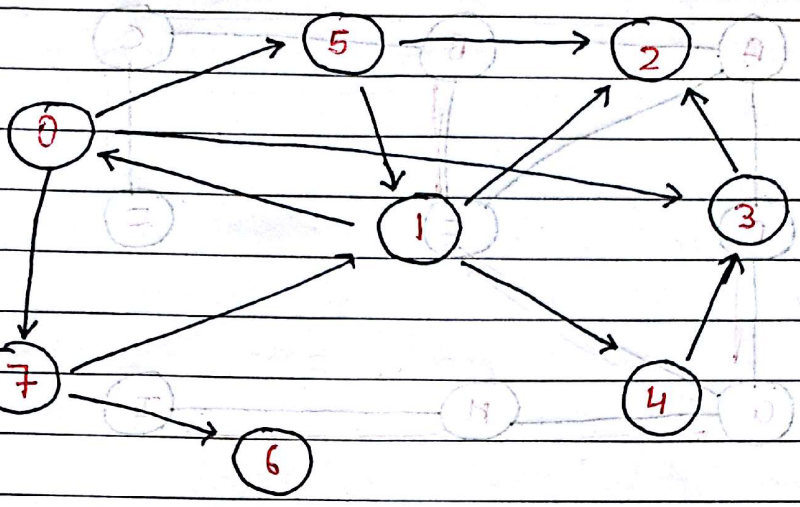
→ Sum of the lengths of adjacency lists : O(E)

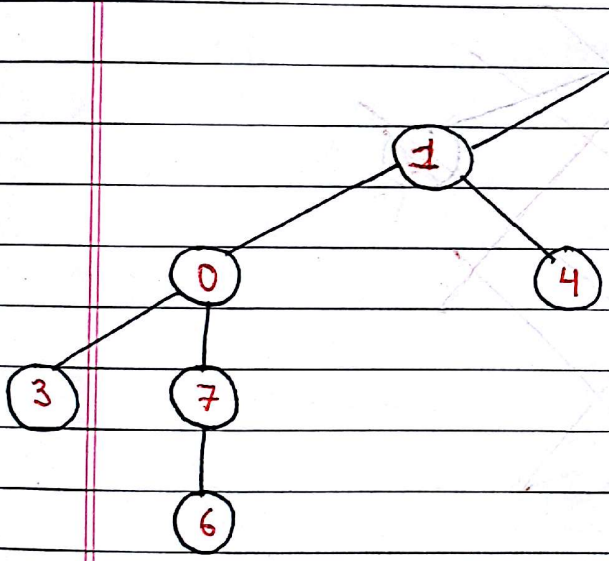
→ Initialization overhead : O(V)

Total runtime : O(V + E)

PAGE No.
DATE / /

→ Example : BFS start with Node 5 ←





0 → 3 → 5 → 7

1 → 0 → 2 → 4

2

3 → 2

4 → 3

5 → 1 → 2

6

7 → 1 → 6

0

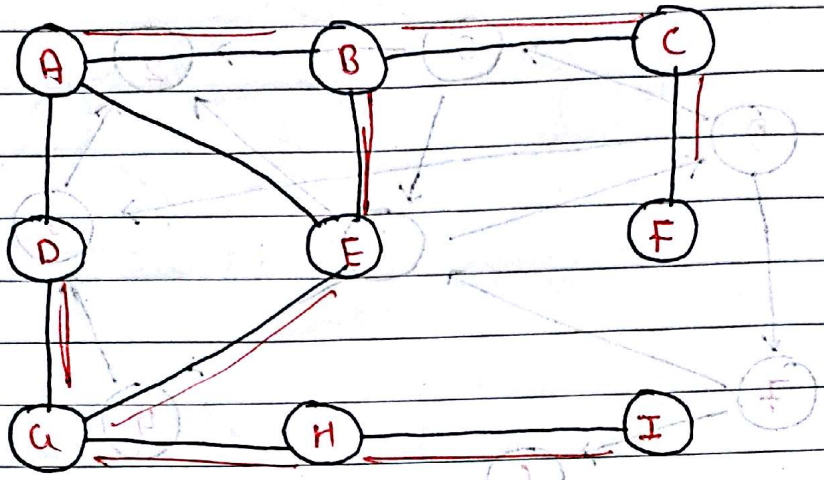
5

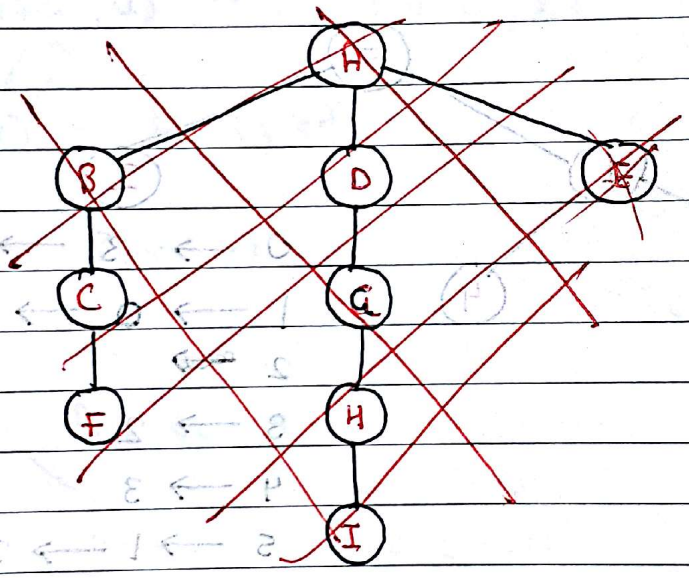
5 1 2

5 1 2 0 4 3 7 6

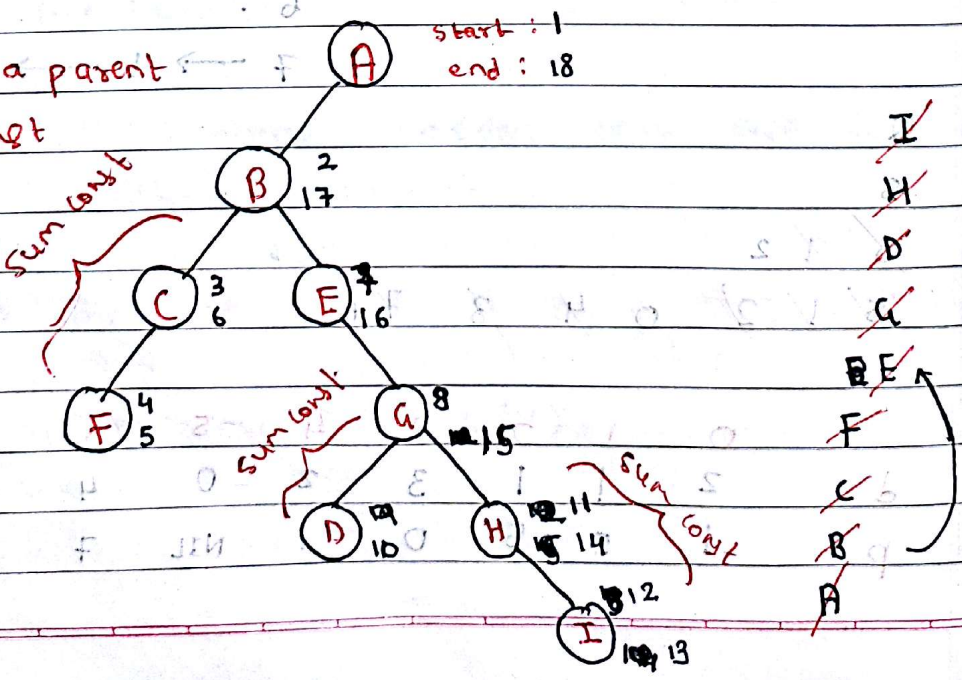
	0	1	2	3	4	5	6	7
d	2	1	1	3	2	0	4	3
p	1	5	5	0	1	NIL	7	0

→ Example : start from A (DFS)



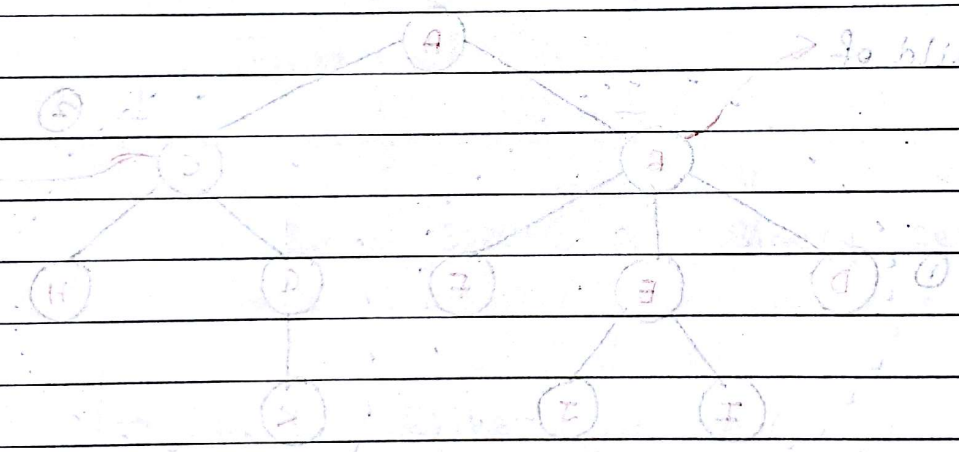


Each line of a parent has sum const



PAGE No.
DATE / /

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



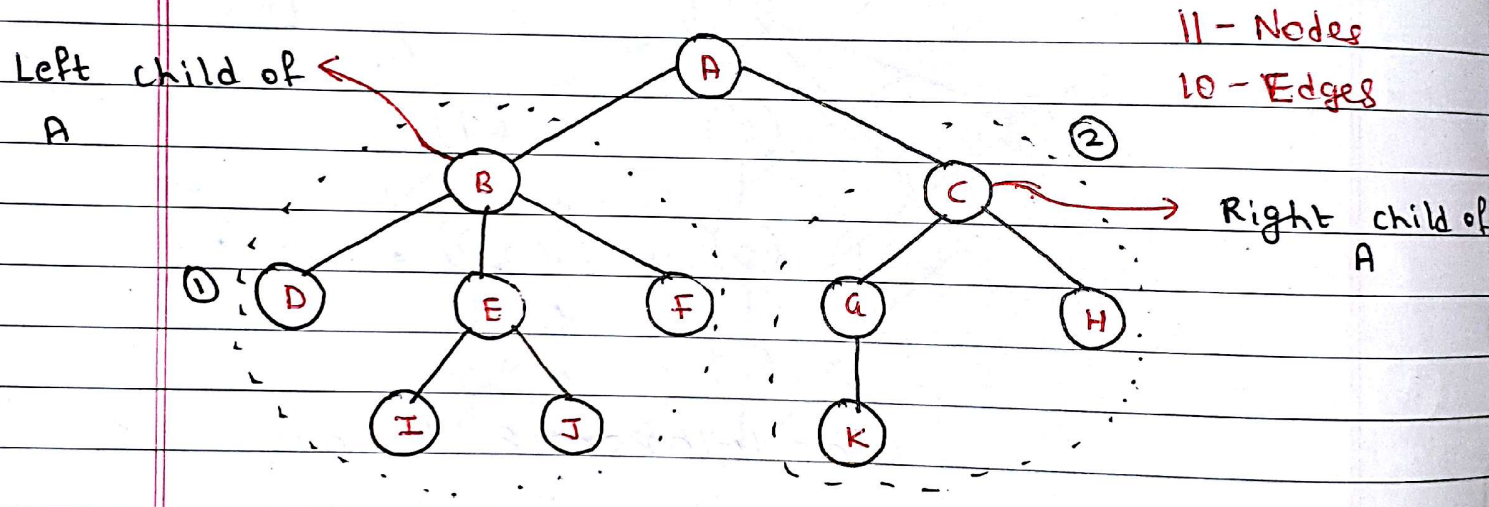
PAGE No.
DATE / /

* Tree

It is a non-linear DS which organizes data in hierarchial structure and this is a recursive definition.

It is a collection of nodes

- Always a single root node at the top.
- In a tree with 'N' nodes there will be maximum of 'N-1' edges.



- Nodes have children are internal node [B, C, E, G]
- A : Special Node - Root
- Nodes not having any children Leaf node [D, I, J, F, K, H]
- Successor nodes - eg: of B : [D, I, J, F]
- Left Sub-tree - (1)
- Right Sub-tree - (2)

→ Degree 0 : Leaf or terminal node

PAGE No. / DATE / /

- Degree : No. of children (sub-trees)
- Level : Root - Level 0

⋮ ⋮

- Height : Number of edges on the longest path from that node / or edges from deepest leaf to given node [Top down till longest path leaf reached]
  - ↳ Root (in case of entire tree)
  - ~~Level of the deepest node - level of given node~~
- Depth = = = Level

Types of Trees

General
any number of children

Binary Tree
max 2 children

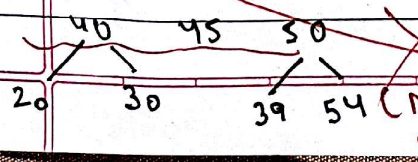
Binary search Tree
order maintained

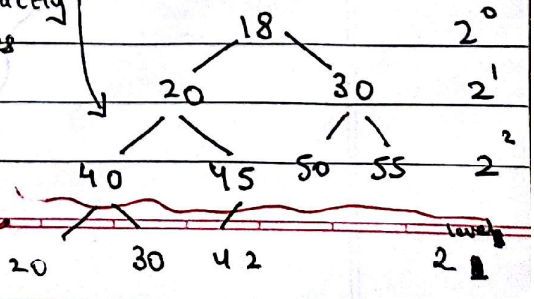
→ Full BT : Either zero or 2 children

→ Complete BT : All levels are completely filled except last level.

→ Perfect BT : All internal nodes should have exactly 2 nodes and leaf nodes should be at same level

Nodes at last levels should be as left as possible

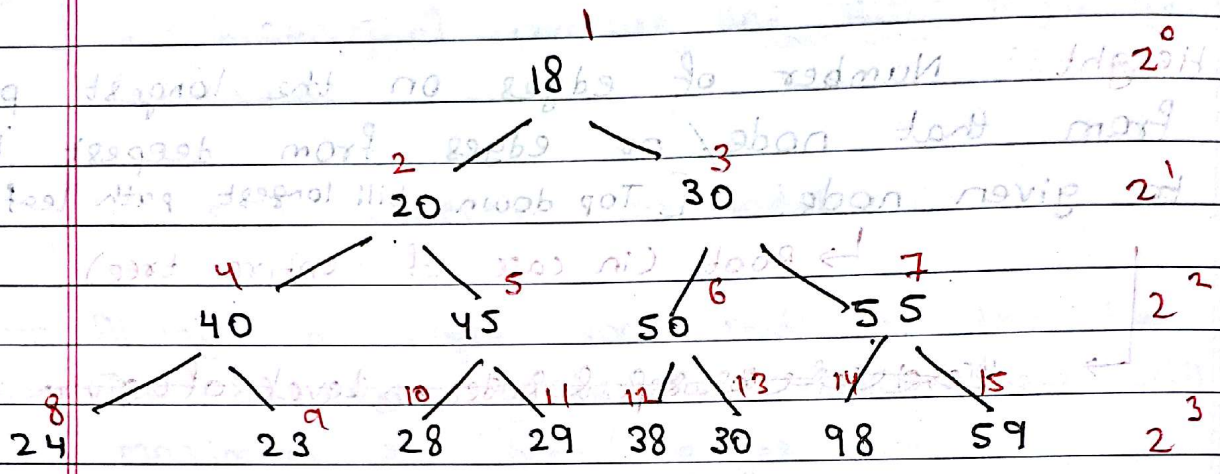




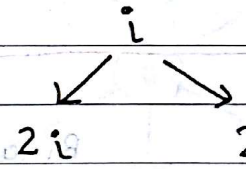
Array representation is good

- Binary Perfect Tree :

Max/ Total no. of nodes $= \sum_{i=0}^{h} 2^i$ → height of tree



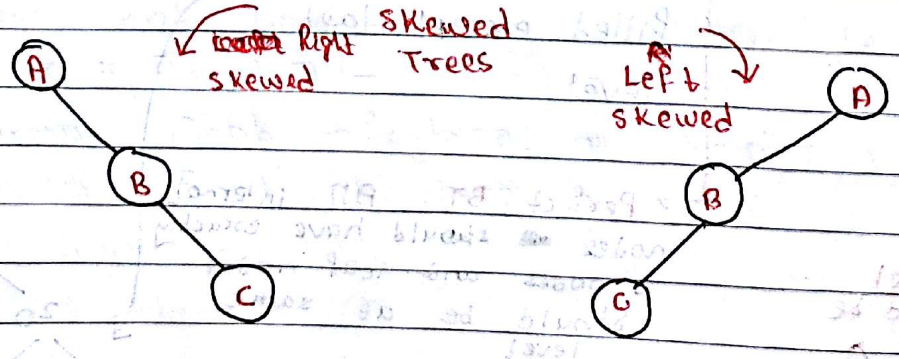
Only for perfect :



→ Sequential Representation :

i] Waste Space

ii] Insertion / Deletion Problem

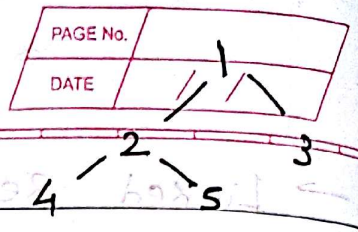


→ Linked Representation :

```
typedef struct node* tree_pointer;

typedef struct node {
    int data;
    tree_pointer left_child, right_child;
};
```

```
struct node {
    int data;
    struct node* left;
    struct node* right;
};

struct node* newNode(int data) {

    struct node* node = (struct node*) malloc(
                        sizeOf(struct node));

    node -> data = data;
    node -> left = NULL;
    node -> right = NULL;

    return node;
}
```

PAGE No.
DATE

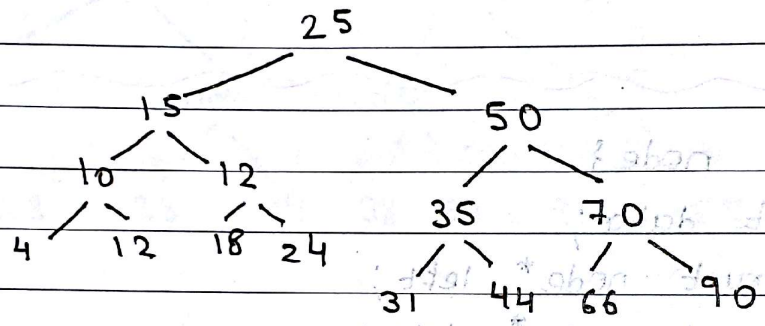


* Tree Traversals : Root

a] Inorder - [Left, Root, Right] : 4 2 5 1 3
b] Preorder - [Root, Left, Right] : 1 2 4 5 3
c] Post order - [Left, Right, Root] : 4 5 2 3 1

Root

Q]



~~Pre : 25 15 10 4 12~~

In: 4, 10, 12, 15, 18, 12, 24, 25, 31, 35, 44, 50, 66, 70, 90

Pre

~~Post~~ : ~~25, 10, 4, 12, 15, 12, 18, 24, 35, 31, 44~~, 25, 15, 10, 4, 12, 12, 18, 24, 50, 35, 31, 44, 70, 66, 90

Post: 4, 12, 10, 18, 24, 12, 15, 31, 44, 35, 66, 90, 70, 50, 25

Inorder : For BST - Ascending Order
Preorder : Root of tree at start
Postorder : Root of tree at end

PAGE No.
DATE / /

→ Inorder Traversal :

1.

```
void printInorder ( struct Node* node) {

    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node → left);

    /* then print the data of node */
    cout << node → data << " ";

    /* now recur on right child */
    printInorder (node → right);
```

PAGE No.
DATE / /

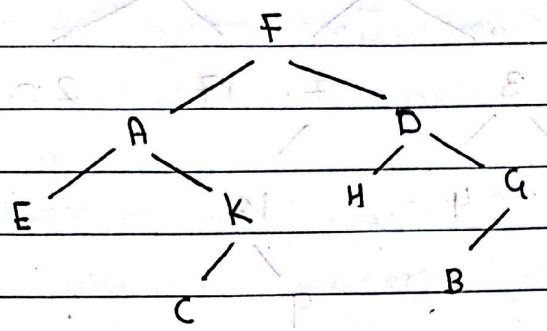
→ PreOrder

```
void printPreOrder (struct Node* node){

    if (node == NULL)
        return;

/* first print data of node */
  cout << node → data << " ";

/* then recur on left subtree */
  printPreOrder( node → left);

/* then recur on right subtree */
  printPreorder (node → right);

}
```

PAGE No.
DATE / /

Q] Given : In - E A C K F H D B G [L Ro R]

Pre - F A E K C D H G B [Ro L R]



→ Post order

```
void printPostOrder (struct Node* node) {

    if (node == NULL)
        return;

    printPostOrder (node → left);

    printPostOrder (node → right);

    cout << node → data << " ";

}
```

In : L, Ro, R

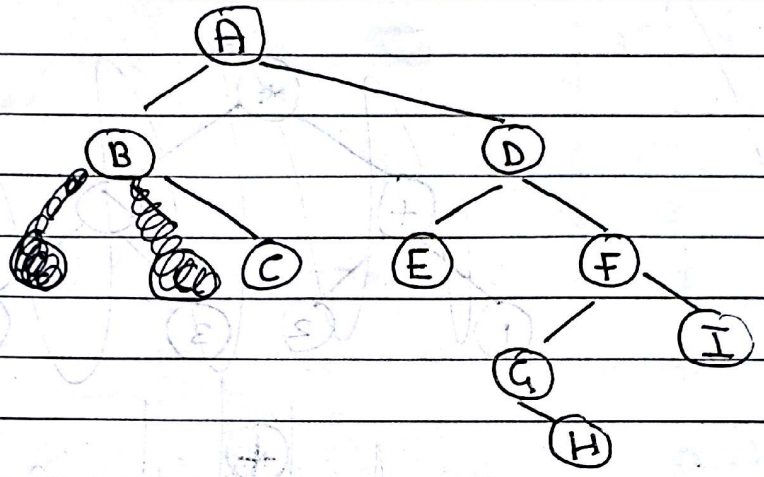
Pre : Ro, L, R

Post : L, R, Ro

Root

① Postorder :- C, B, E, H, G, I, F, D, A

Inorder :- B, C, A, E, D, G, H, F, I

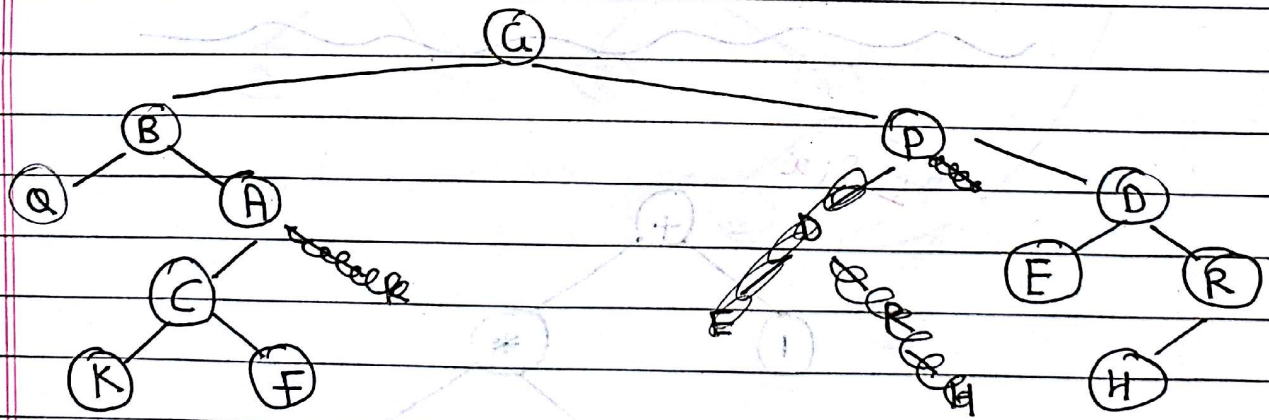


Ro, L, R ② Preorder :- G, B, Q, A, C, K, F, P, D, E, R, H

L, Ro, R Inorder :- Q, B, K, C, F, A, G, P, F, D, H, R

Root L R

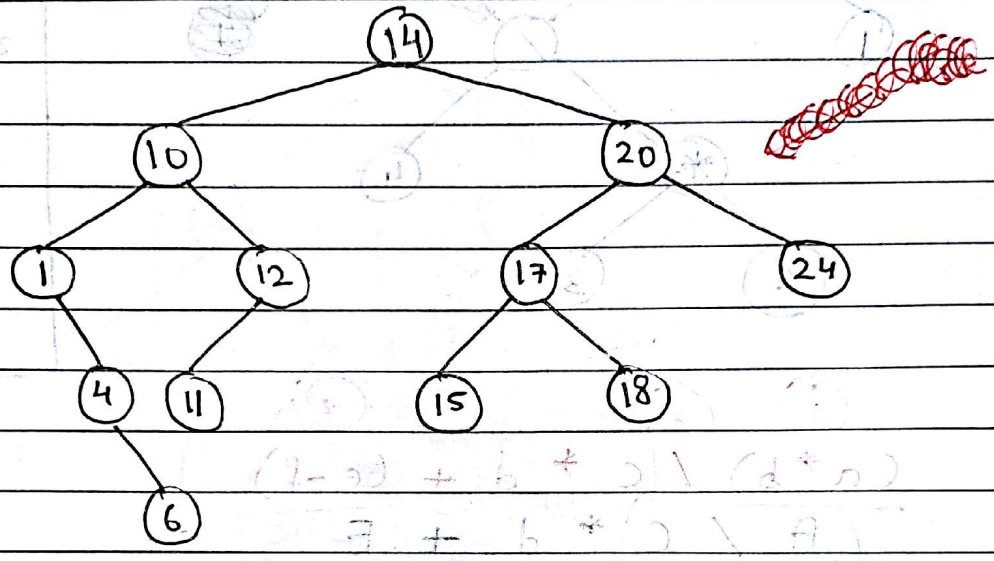
L R



PAGE NO.
DATE / /

* BST

14, 10, 1, 20, 17, 24, 18, 12, 15, 11, 4, 6



* Binary Tree Search Algorithm :

```
TREE SEARCH (x, K)
1.  If   x == NIL   or   K == x.key
2.          return x
3.  If   K < x.key
4.          return  TREESEARCH (x.left, K)
5.  else  return  TREESEARCH (x.right, K)
```

PAGE NO.
DATE / /

* Minimum Key or Element

```
TREE MIN (x)
1. WHILE  x.left ≠ NIL
2.            x = x.left
3. return  x
```

* Maximum Key or Element

```
TREEMAX (x)
1. WHILE    x.right ≠ NIL
2.              x = x.right
3. return x
```

* Insert a value into the BST

```
TREE INSERT (T, z)
1.  y = NIL
2.  x = T.root
3.  While   x ≠ NIL
4.              y = x
5.              If   z.Key < x.Key
6.                       x = x.left
7.              else   x = x.right
8.  z.p = y
9.  if   y == NIL
10.             T.root = z
11. elseif   z.Key < y.Key
12.             y.left = z
13. else   y.right = z
```

PAGE No.
DATE / /

Q] Suppose we have numbers between 1 - 1000. We want to search 363. Which of the foll. sequences could not be the seq of nodes: Examine:

a] 2, 252, 401, 398, 330, 344, 397, 363

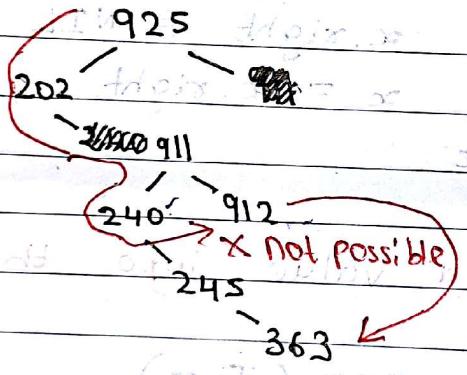
b] 924, 220, 911, 244, 898, 258, 362, 363

c] 925, 202, 911, 240, 912, 245, 363

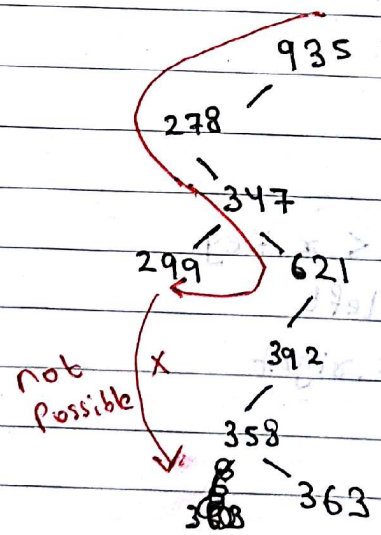
d] 2, 399, 387, 219, 266, 382, 381, 278, 363

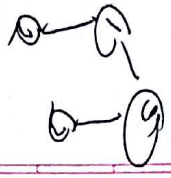
e] 935, 278, 347, 621, 299, 392, 358, 363

c]



e]

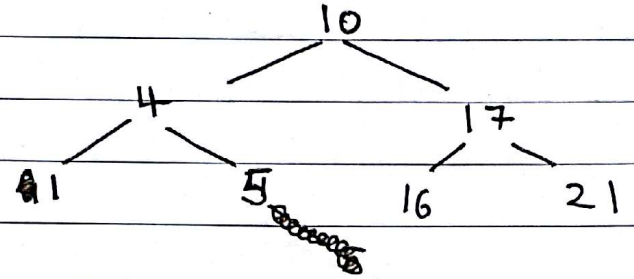




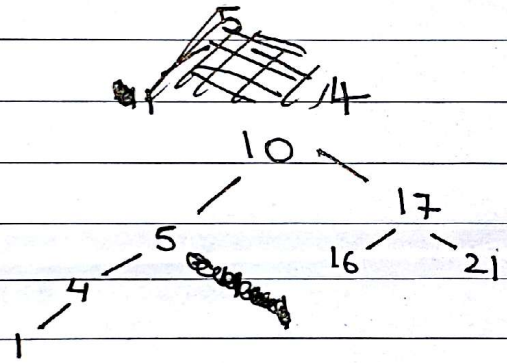
PAGE No.
DATE / /

Q] Keys = {1, 4, 5, 10, 16, 17, 21}
Create BST of h = 2, 3, 4, 5, 6

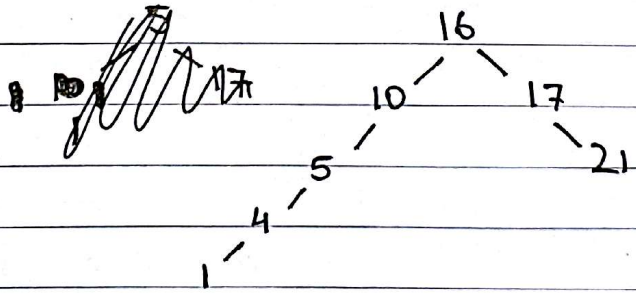
Soln: h = 2



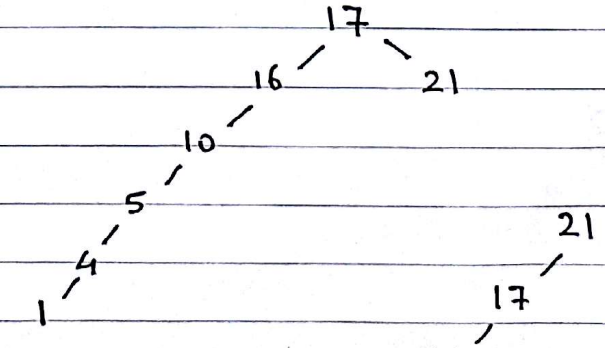
h = 3



h = 4

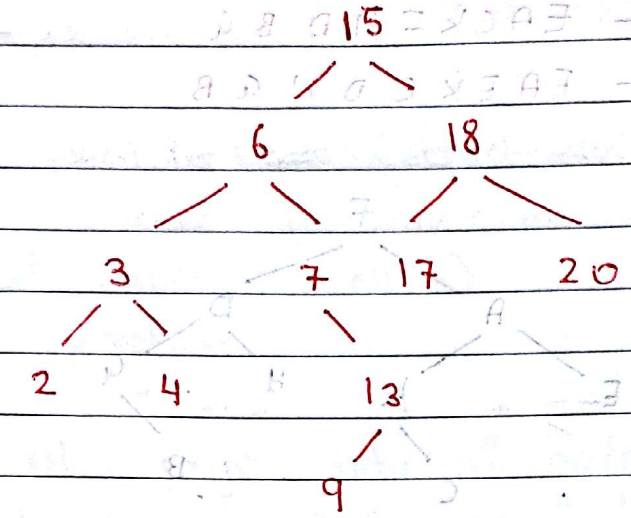


h = 5



h = 6

Q



Inorder : 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20 (Ascending)

Inorder - Successor : of 7 → 9

of 9 → 13

of 20 → X

Inorder - Predecessor : of 7 → 6

of 6 → 4

of 2 → X

- Successor : Right subtree's Leftmost element
- Predecessor : " Left subtree's Rightmost element

→ Tree successor (x)

- Smallest key greater than x.key
- Leftmost node in right subtree
- If right sub-tree is empty, then x has successor of y
- Then y is the lowest ancestor of x whose left-child is also an ancestor of x.

1] If x.right ≠ NULL then
return Tree-Minimum (x, right)

and if

y = x.p

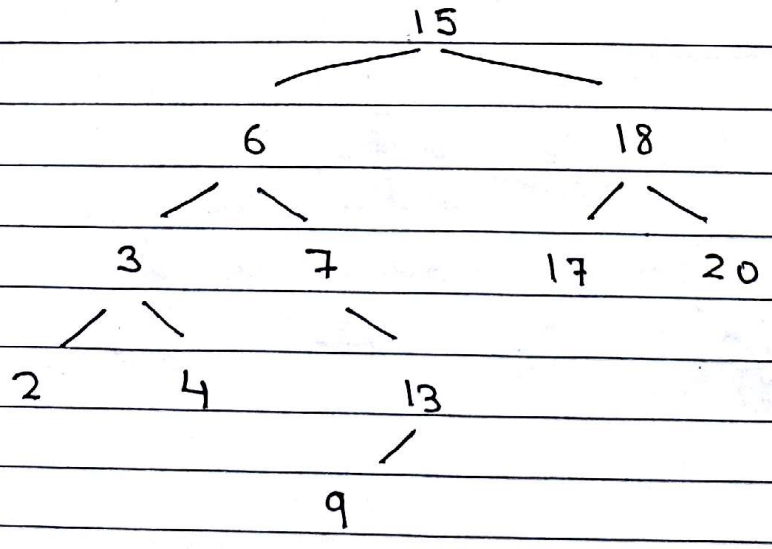
while y ≠ NIL and x == y.right do

x = y

y = y.p

end while

return y



PAGE No.
DATE / /

→ Tree-predecessor (x)

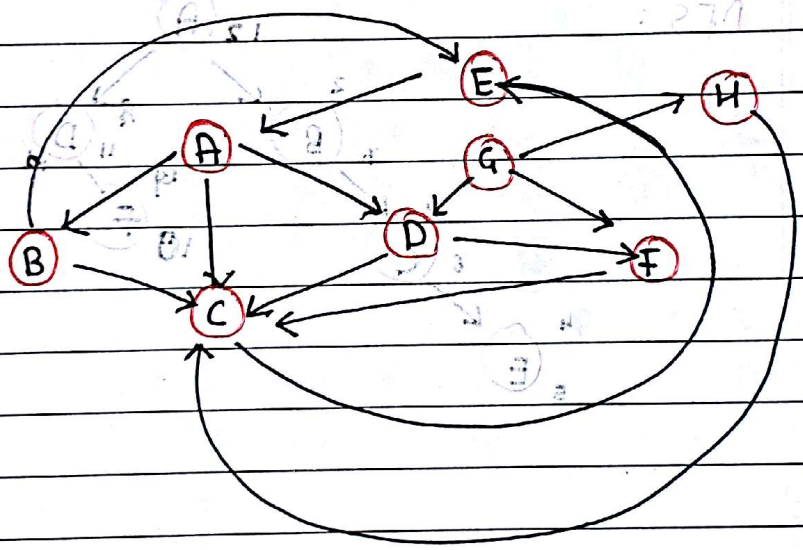
If x.left ≠ NULL then
return Tree-maximum (x.left)
and if
y = x.p
while y ≠ NIL and x == y.left do
x = y
y = y.p
end while
return y

PAGE No.
DATE / /

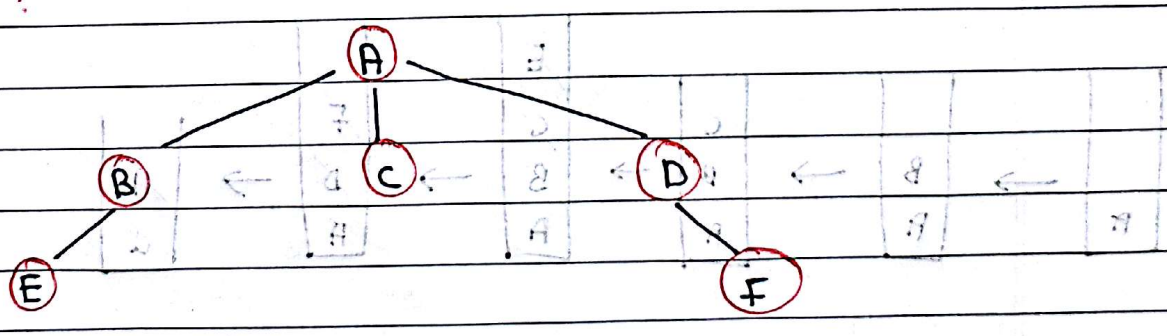
* Revision

→ Pre-process

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

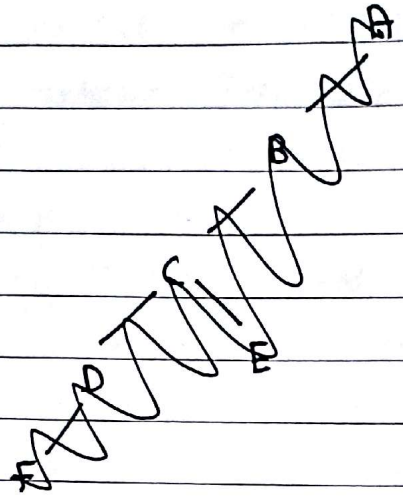


BFS :

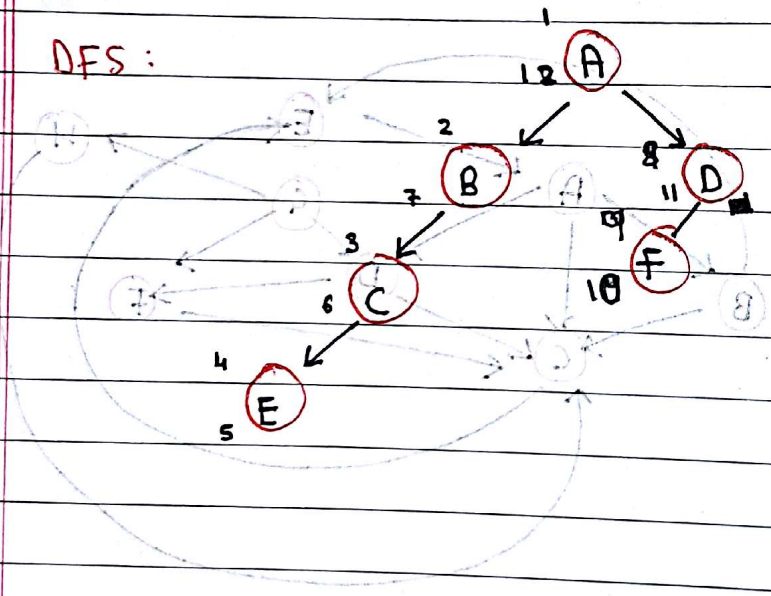


PAGE No.
DATE / /

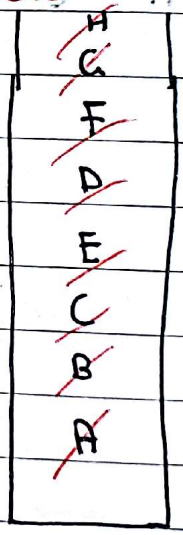
DFS:

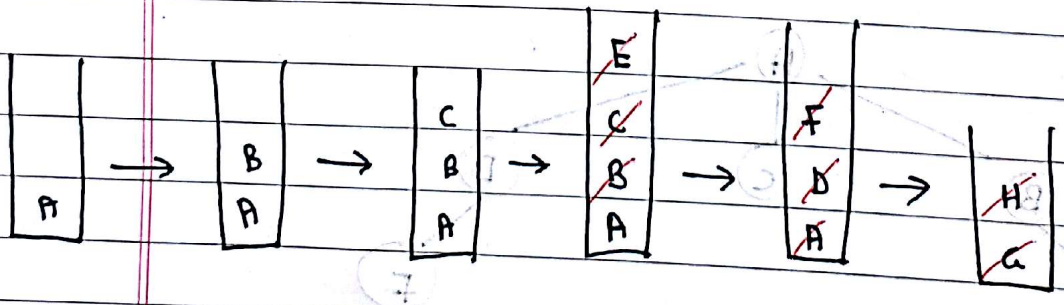


DFS:



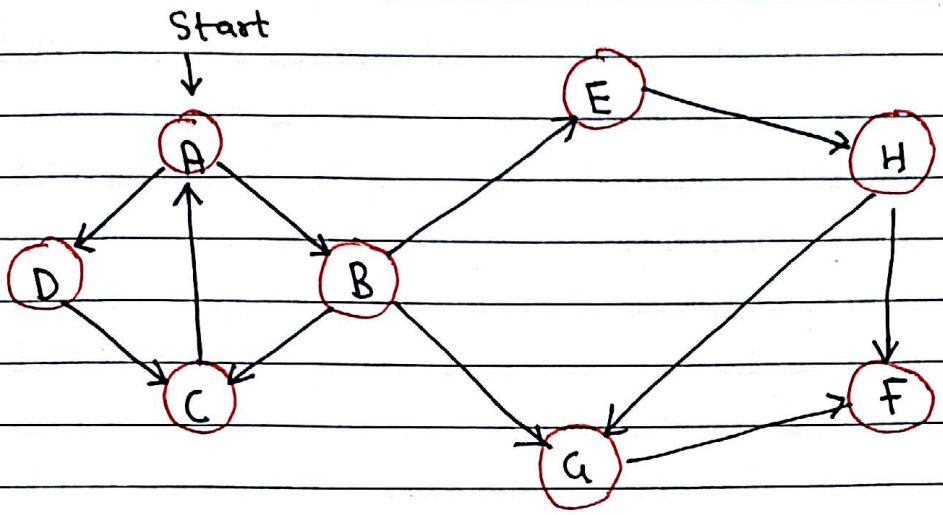
Stack :: H



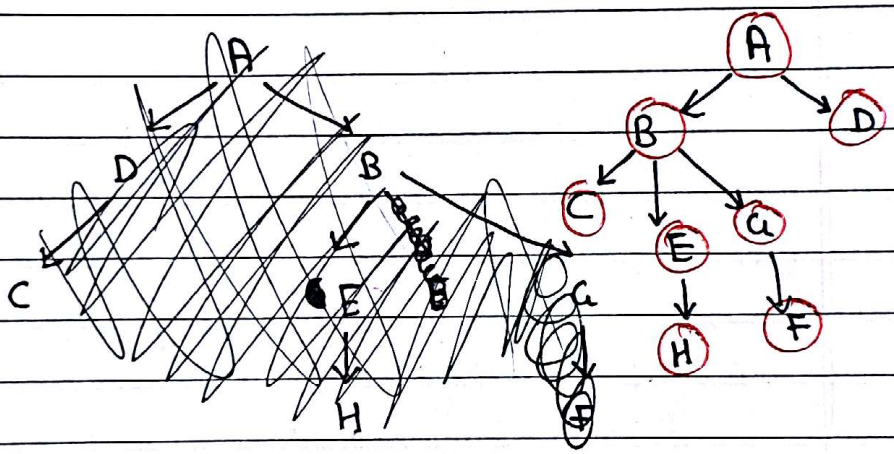


PAGE No.
DATE / /

Q]



BFS :



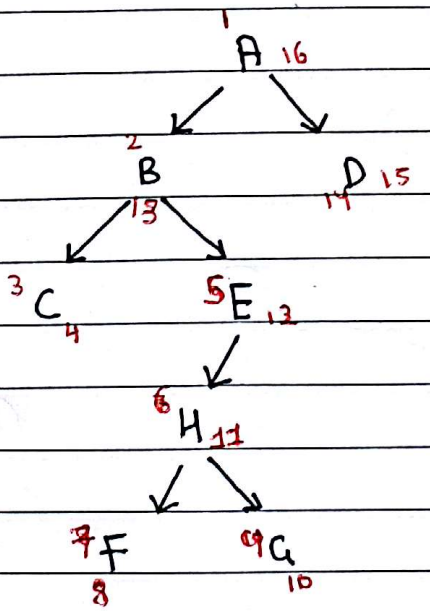
Queue :

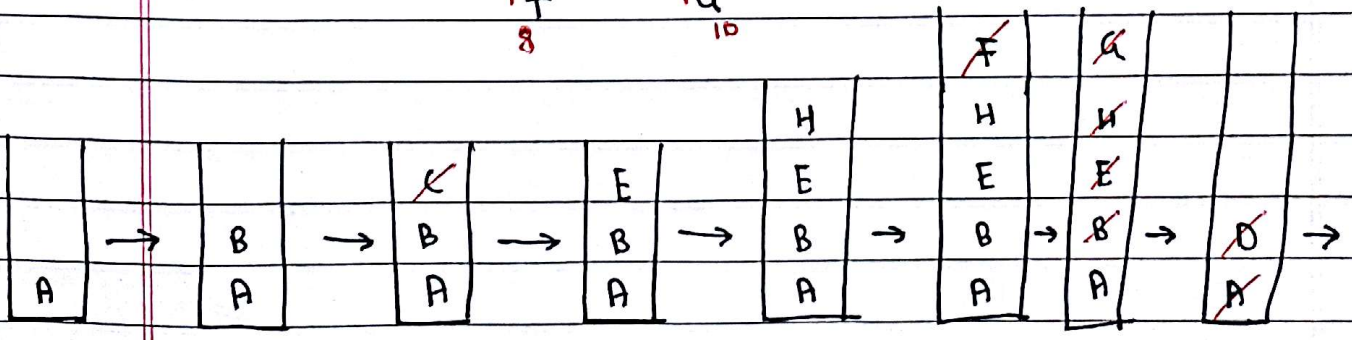
A
~~A~~ B D
~~B~~ D C ~~E~~ G

~~D~~ E G H
~~E~~ G ~~H~~ ~~F~~

	A	B	C	D	E	F	G	H
P	–	A	B	A	B			
D	0	1	2	1	2			2

DFS :

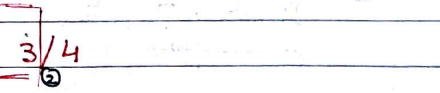


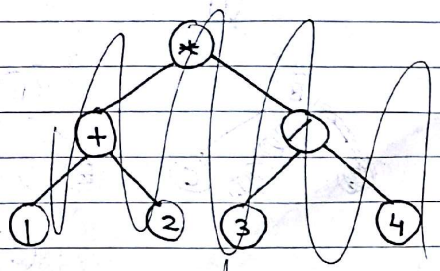


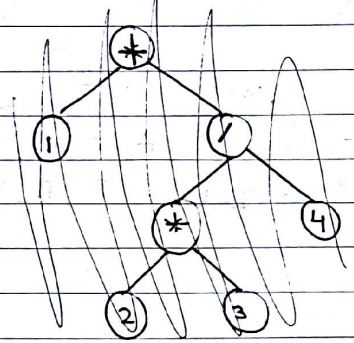
* Expression Tree

1 + 2 / 3 * 4

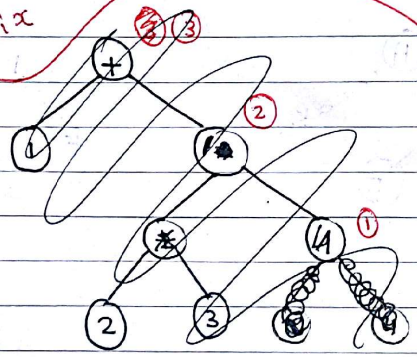
1] 1 + 2 * 3 / 4

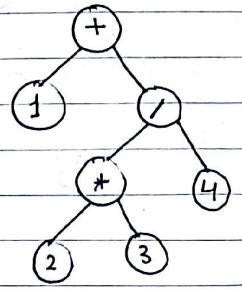




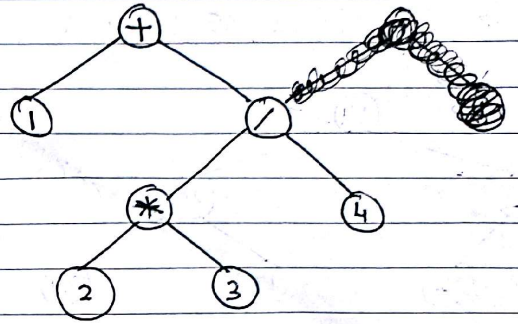


Infix





2] 1 2 3 * 4 / + (Post)



2*3 4
4
2*3
3
2
1

3] (a*b) / c * d + (e-f)

(A / C) * d + E

a
b

ab*c/d*ef-+