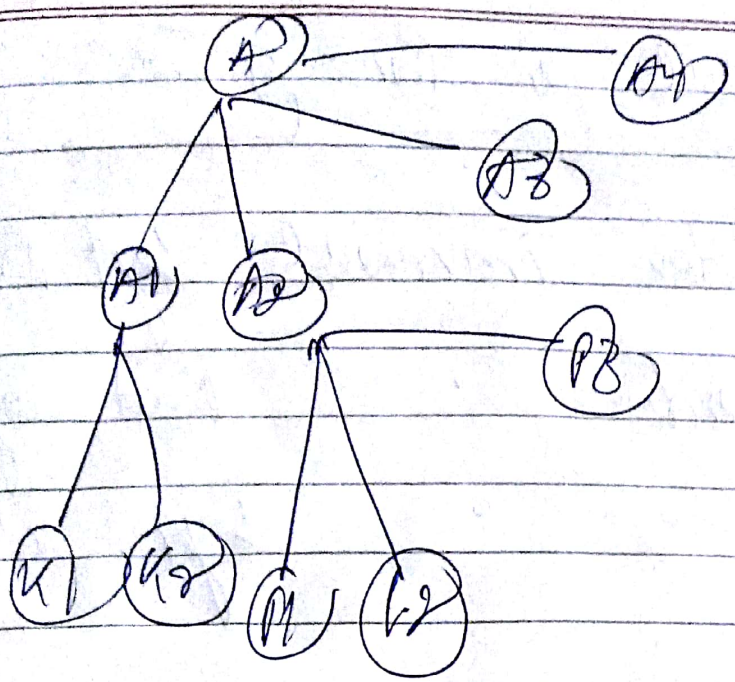
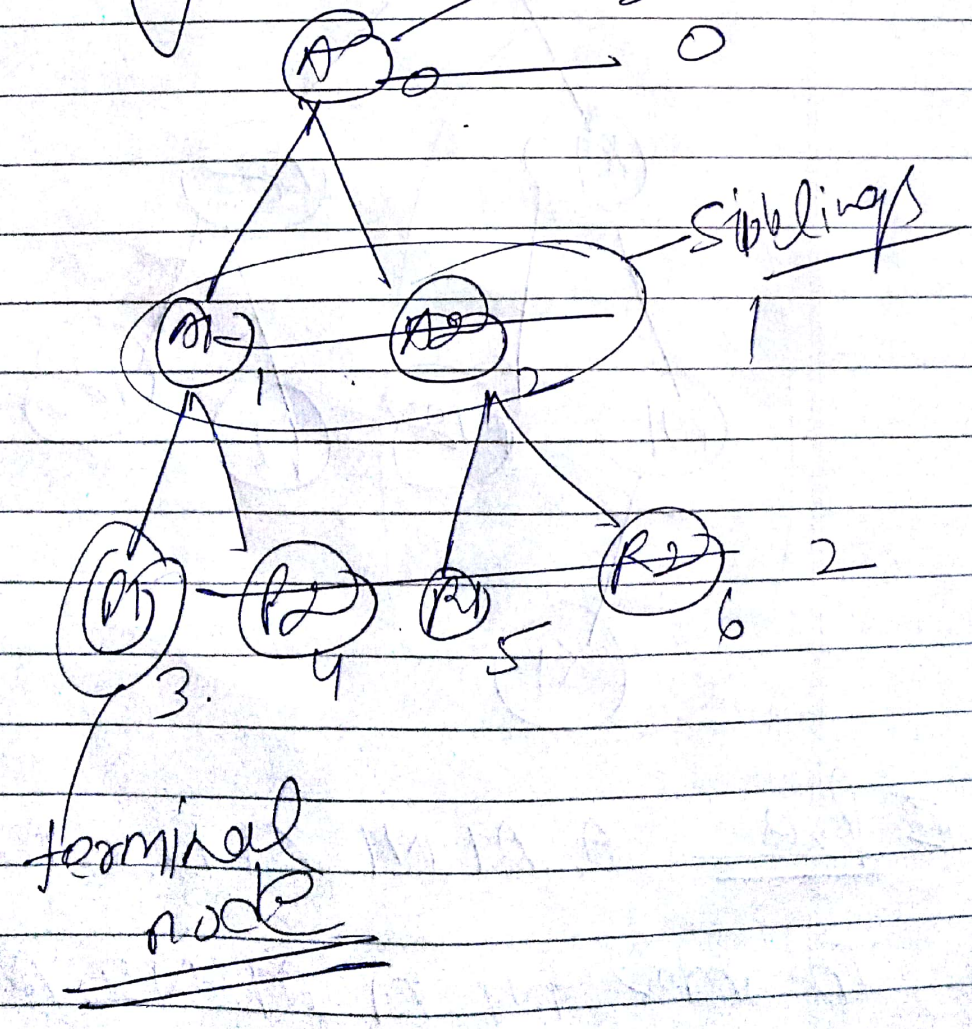


Binary Tree

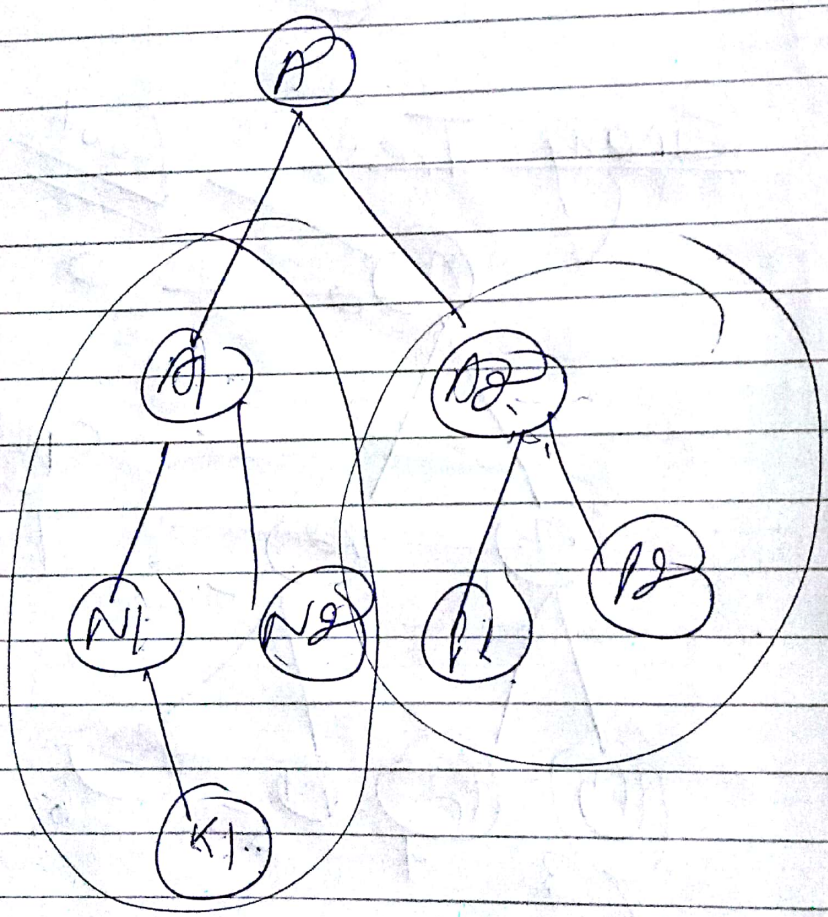


Traversal of Binary Tree

Inorder Traversal Left Root Right

Pre-order : Root Left Right

Post : : Left Right Root



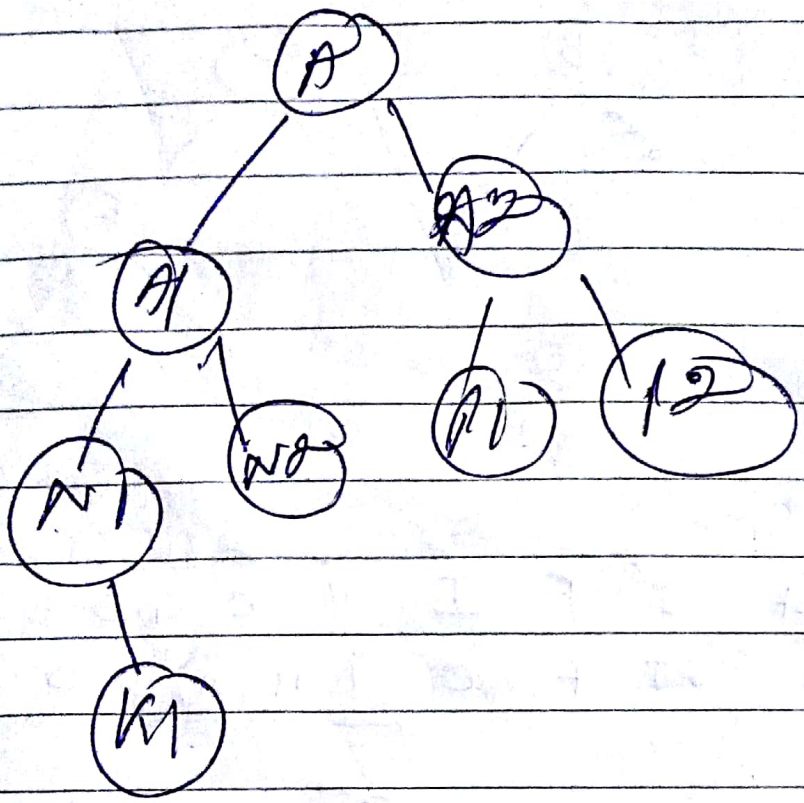
Pre A B1 N1 K1 N2 B2 P1 P2

In-order N1 K1 B1 N2 A P1 B2 P2

Post-order K1 N1 N2 B1 P1 P2 B2 A

Pre (A) A1 N1 K1 N2 A2 P1 P2

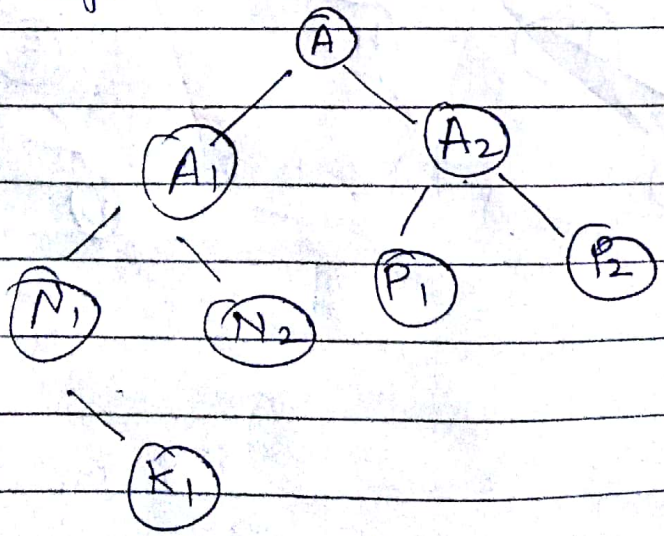
In M K1 A1 N2 (A) P1 A2 P2



Post K1 N1 N2 A1 P1 P2 A2 A

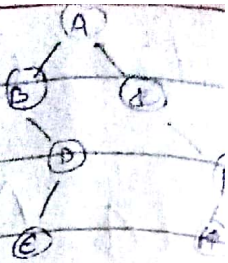
In N1 K1 A1 N2 (A) P1 A2 P2

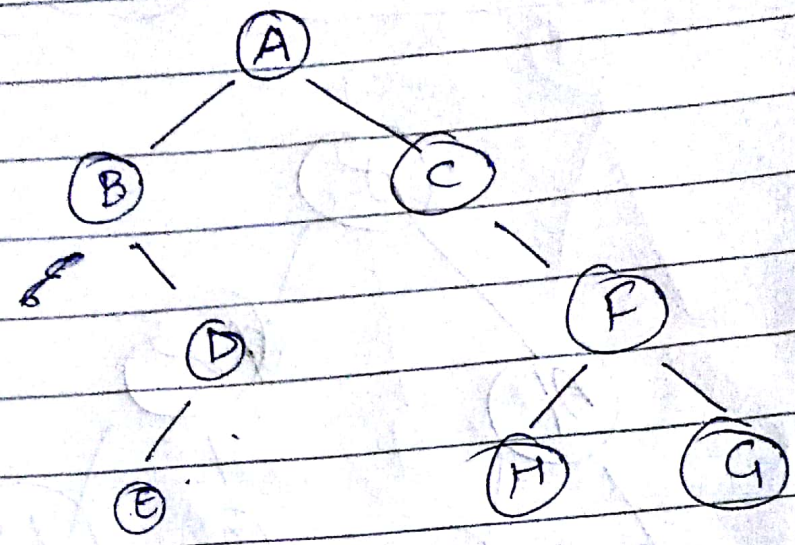
Left Right



Q In:- B E D A C H F G

Pre:- A B D E C F H G

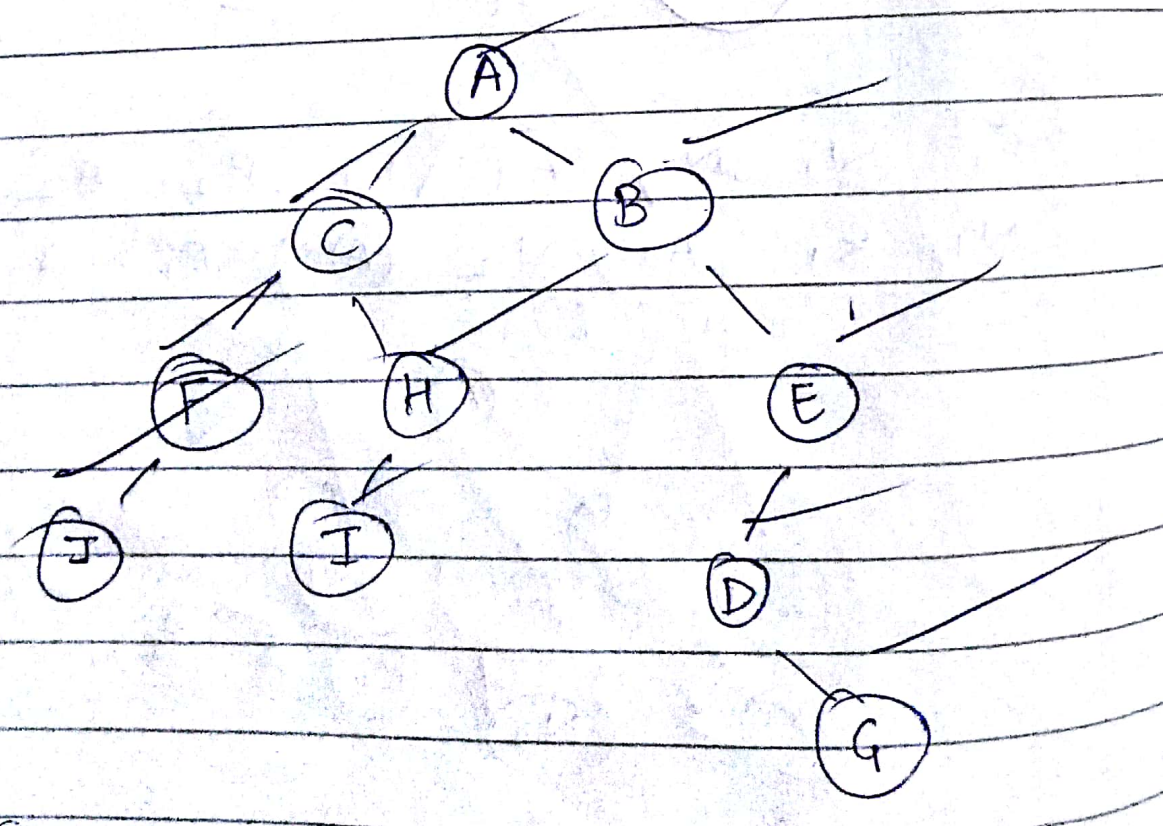




~~B E D A C H F G~~

Q- Post J F I H C G D E B A

In: J F C I H A B D G E



TREES

Depending on the requirement to represent data in the memory and to process the data. Trees are of many types some of the commonly used tree types in the data structure are follows:

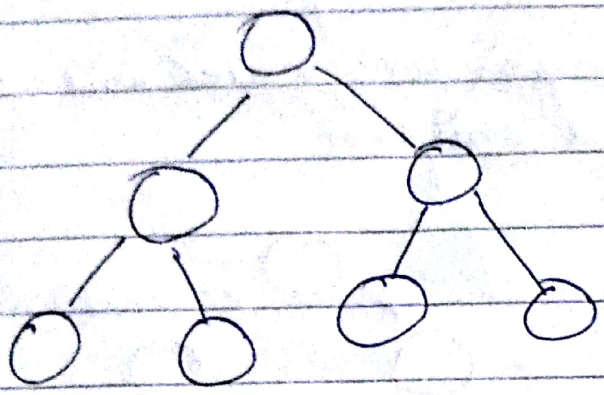
(i) Binary Tree
(ii) Binary Search Tree
(iii) Threaded tree
(iv) Heap tree
(v) B - tree
(vi) AVL - tree

Binary Tree

Binary tree is the finite set of elements represented in terms of nodes. The data element or nodes are linked together in hierarchical manner. i.e top to down. There is present a special node at the top of the tree called root of tree. Rest

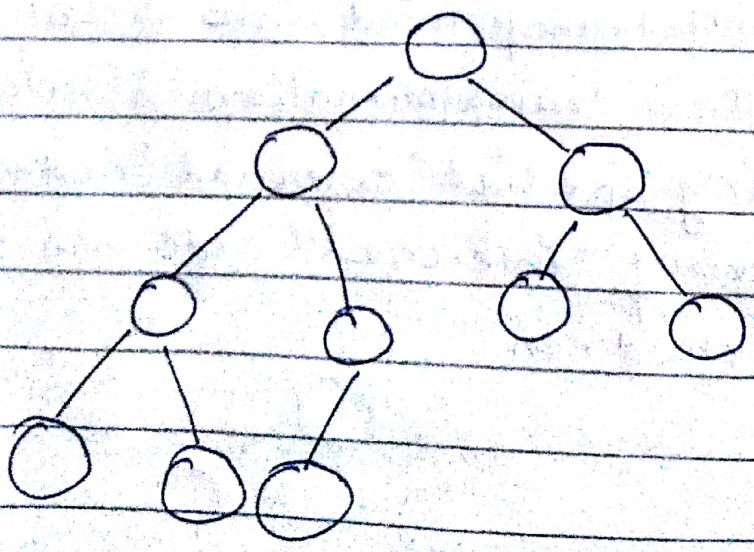
Complete Binary Tree

- a complete binary tree is defined as a binary tree whose non leaf nodes have non empty left and right sub-tree and all leaves are at the same level



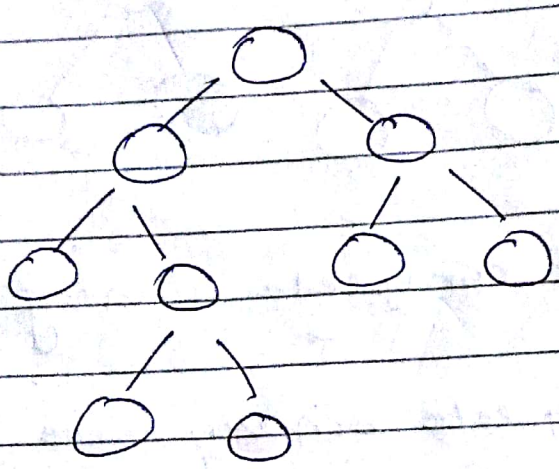
Almost complete Binary Tree

- a complete binary Tree is defined as a binary tree whose non leaf nodes have non empty left and right sub-tree and all leaves are either at the last level or second last level



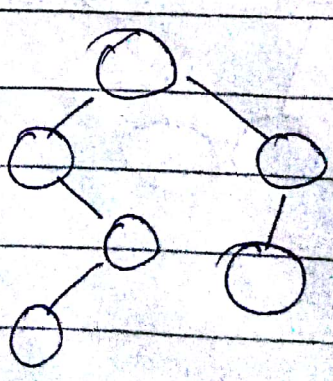
Strict Binary Tree

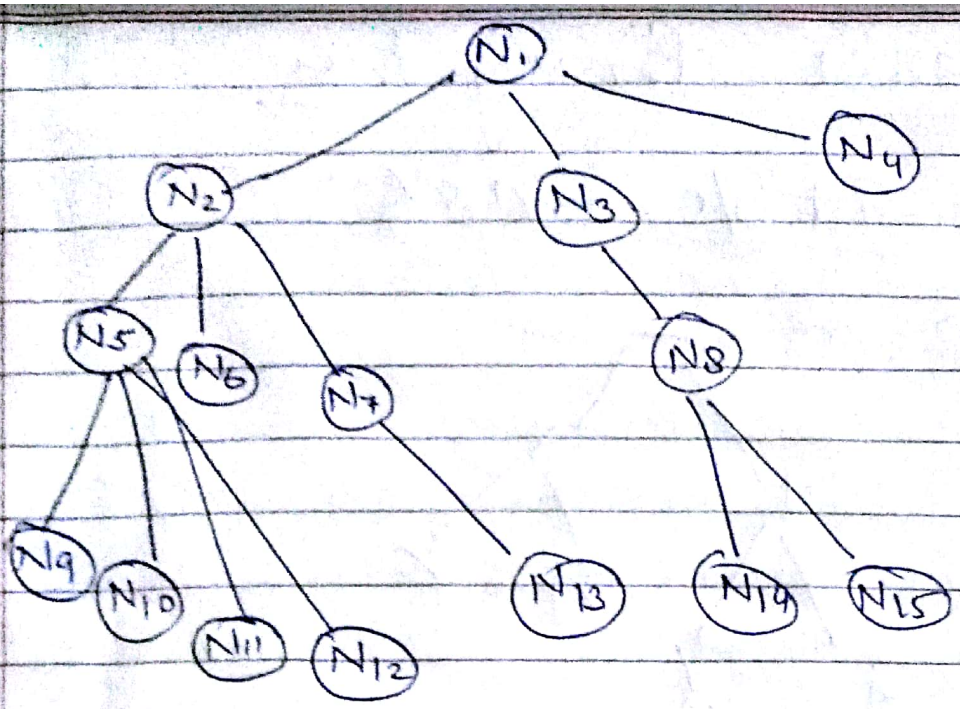
- A binary tree is called a strictly binary tree if every non leaf node in the binary tree has non empty left and right subtree.
- It means each node will have either 0 or 2 children.



Extended Binary Tree

- An extended binary tree is a tree that has been transformed into a full binary tree. This transformation is achieved by inserting special "external" nodes such that every "internal" node has exactly two children.



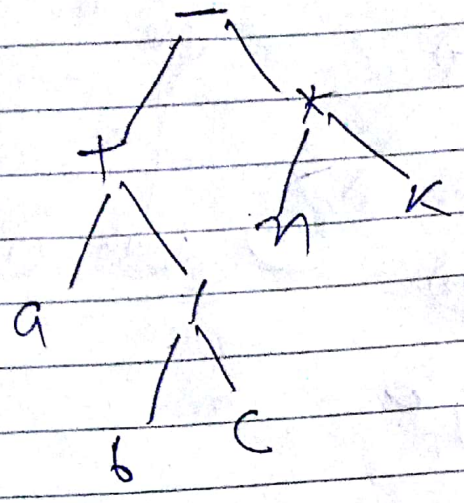


Terminologies

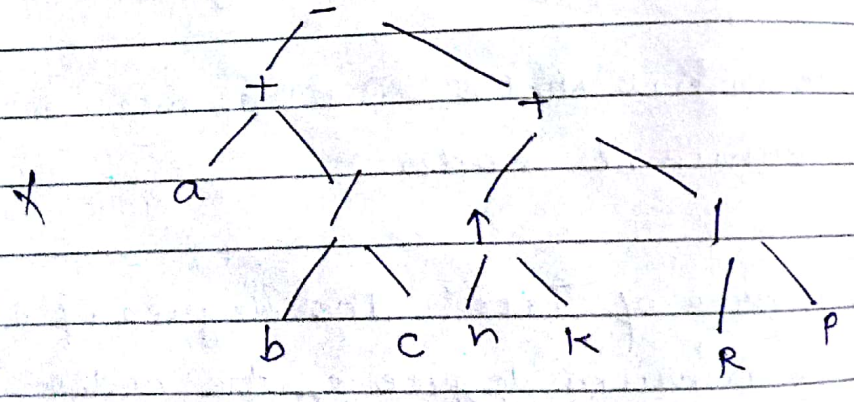
- Degree: The no. of sub-trees of a node is called its degree.
- Leaf: A node with degree zero is called leaf
- Terminal Nodes: The leaf nodes are also calle. terminal nodes.
- Degree of Tree: The degree of the tree is maximum degree of the nodes in the tree.

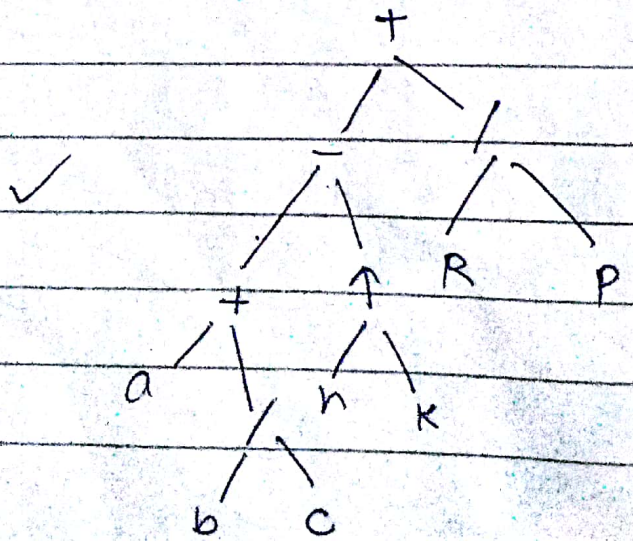
Expression Binary tree

$a+(b/c)-n*k$

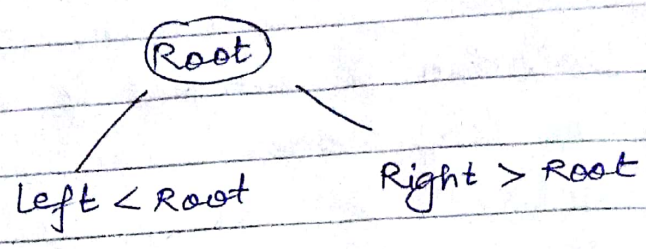


0. $a+(b/c)-n\uparrow k+R/P$

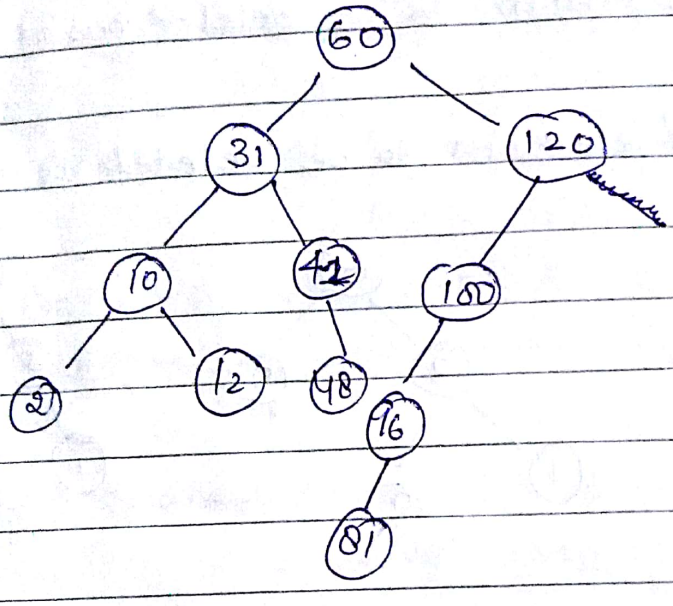




Binary Search Tree

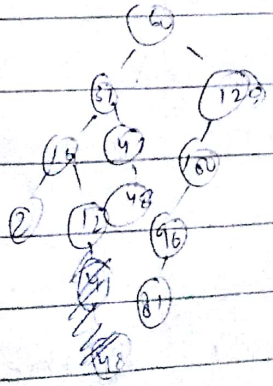


Q. 60 31 120 100 10 2 96 81 12 41 48



Algo.

if root == NULL
  create root node
  return
if root exists then
    if data > node
  goto right subtree
  else
  goto left subtree
endif
  Insert data
end if

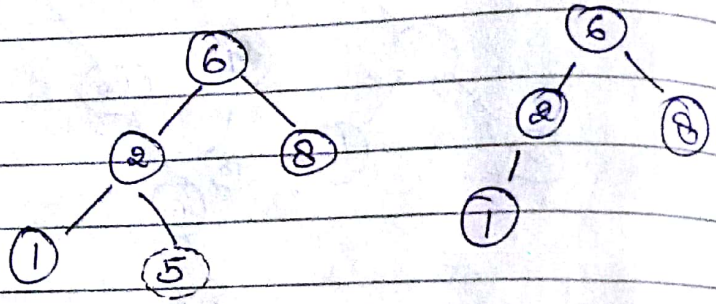


Deletion

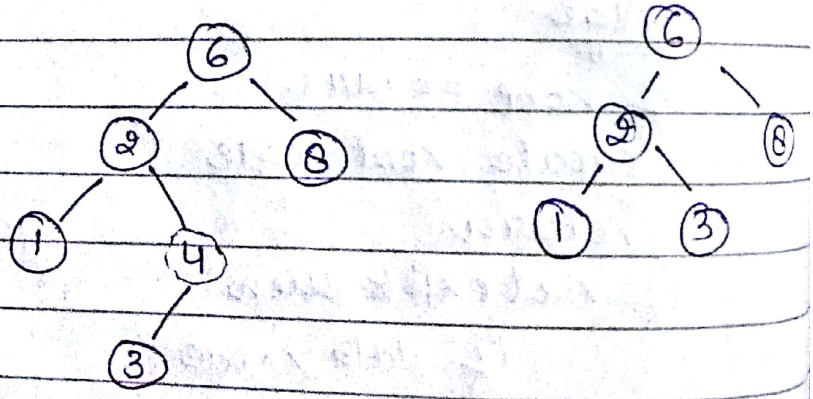
- Deletion is simple if we are deleting a leaf node.

- Three cases
  - if the element to be deleted is a leaf node
  - if the element to be deleted has one child
  - if the element to be deleted has both children

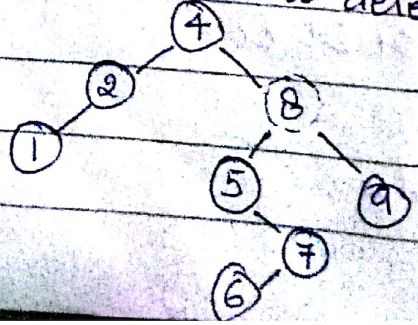
- if the element to be deleted is a leaf node



- if the element to be deleted has one child

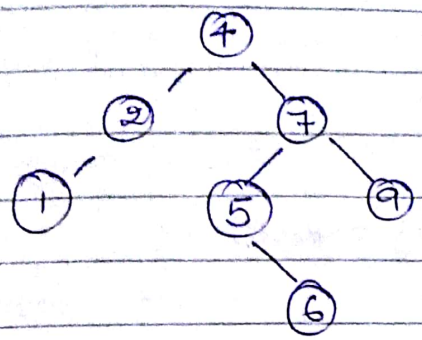


- if the element to be deleted has both children



Find inorder successor of node, then copy content of inorder successor to node and delete inorder successor.

In-order (1) (2) (4) (5) (6) (7) (8) (9)



Code (Binary Search Tree)

```
struct BST node * Find (struct BSTNode *root, int
                        data)
{
if(root == NULL)
   return(NULL);
if(data < root -> data)
   return (Find (root -> left, data));
else if (data > root -> data)
   return (Find(root -> right, data));
return(root);
}

void insert (struct BST Node *root, int data)
{
struct BST Node * par;
struct BST Node * n = malloc(size of (struct BST Node));
n -> left = NULL;
n -> data = data;
n -> right = NULL;
```

```
if (root == NULL)
root = n;
else
{
par = root;
while (par != NULL)
{
if (par -> data > data)
 {
 if (par -> left == NULL)
   par -> left = n;
   par = par -> left;
 }
if (par -> data < data)
 {
 if (par -> right == NULL)
       par -> right = n;
     par = par -> right;
 }
}// end of while
} //end of else.

Struct BST Node * Delete (struct BST Node * root,
  int data)
 {
 struct BST Node * temp;
 if (root == NULL)
 printf ("No Such element exists");
 elseif (data < root -> data)
```

```
root->left = Delete(root->left, data);
else if(data > root->data)
    root->right = Delete(root->right, data);
else
{ // element found
if( root->left && root->right )
  { // both children
    temp = findMax(root->left);
    root->data = temp->data;
    root->left = Delete(root->left, root->data);
  }
else
{ // one or none child
    temp = root;
  if(root->left == NULL)
      root = root->right;
  if(root->right == NULL)
      root = root->left;
      free(temp);
  }
} // end of else
return(root);
} // end of function
```

Problem with BST

- The disadvantage of a skewed binary search tree is that the worst case time complexity of a search is O(n).

AVL Tree

- An empty binary tree is an AVL tree.
- A non empty binary tree T is an AVL tree iff given T^L and T^R to be the left and right subtrees of T and $h(T^L)$ and $h(T^R)$ to be the heights of subtrees T^L and T^R respectively, T^L and T^R are AVL trees and $|h(T^L) - h(T^R)| \leq 1$
- There is a need to maintain the binary search tree to be of balanced height, so that it is possible to be obtained for the search option a time complexity of $O(\log n)$ in the worst case.
- One of the most popular balanced tree was introduced by Adelson-Velskii and Landis (AVL)

Searching in AVL Tree

- Searching an AVL search tree for an element is exactly similar to the method used in a binary search tree

Insertion in AVL Tree

- If after insertion of the element, the balance factor of any node in the tree is affected so as to render the binary search tree unbalanced, we resort to techniques called rotations to restore the balance of the search tree.

Rotations

- To perform the rotations it is necessary to identify a specific node A whose balance factor (BF) is neither 0, 1 or -1 and which is the nearest ancestor to the inserted node on the path from the inserted node on the path from the inserted node to the root.

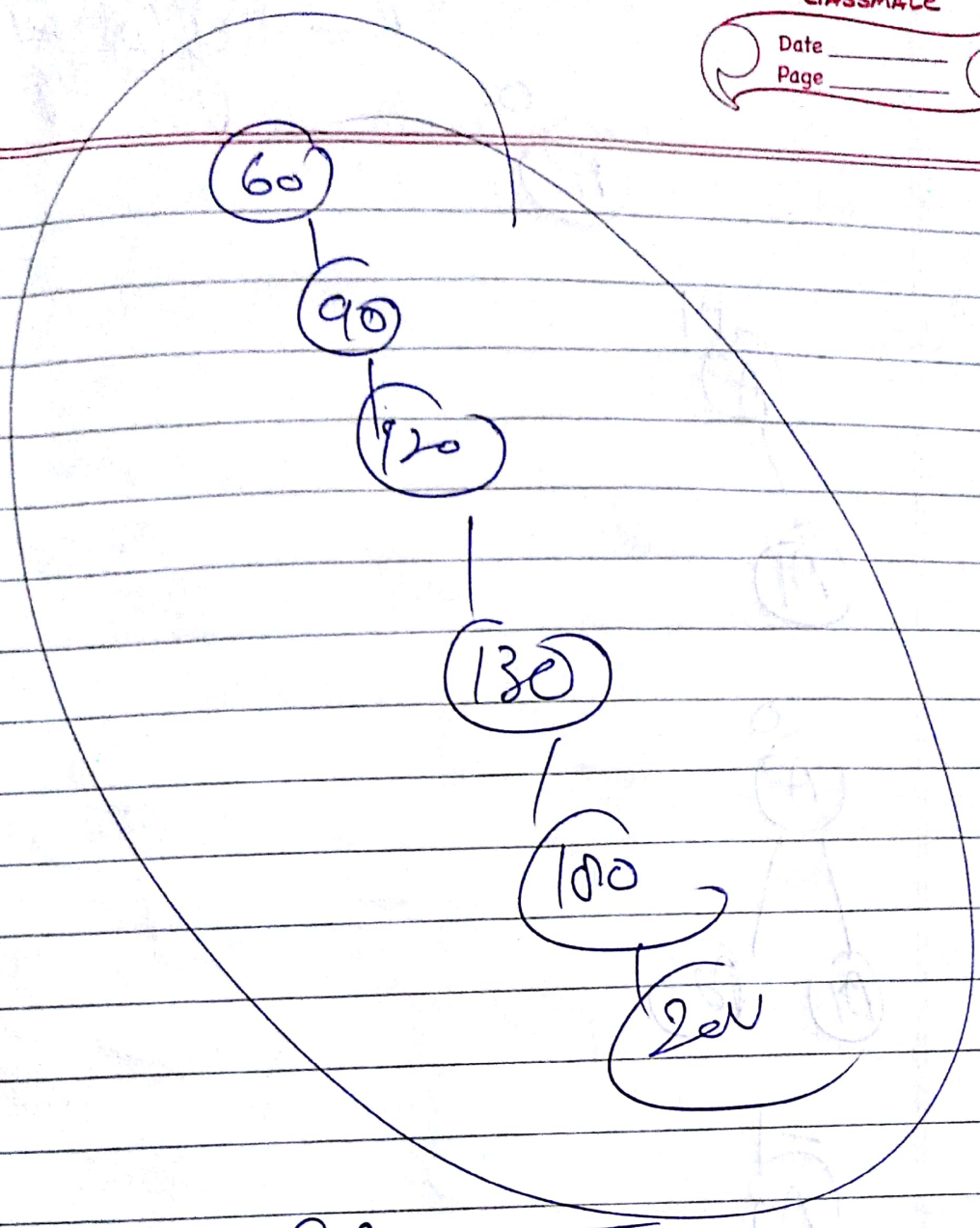
Trick to solve

- For LL and RR rotation identify A and B then make A as a child of B

- for LR and RL rotations identify A, B and C the make A and B child of C

Rotation types

- **LL rotation**: Inserted node is in the left subtree of the left subtree of node A.
- **RR rotation**: Inserted node is in the right subtree of right subtree of node A.
- **LR rotation**: inserted node is in the right subtree of left subtree of node A.
- **RL rotation**: Inserted node is in the left subtree of right subtree of node A.

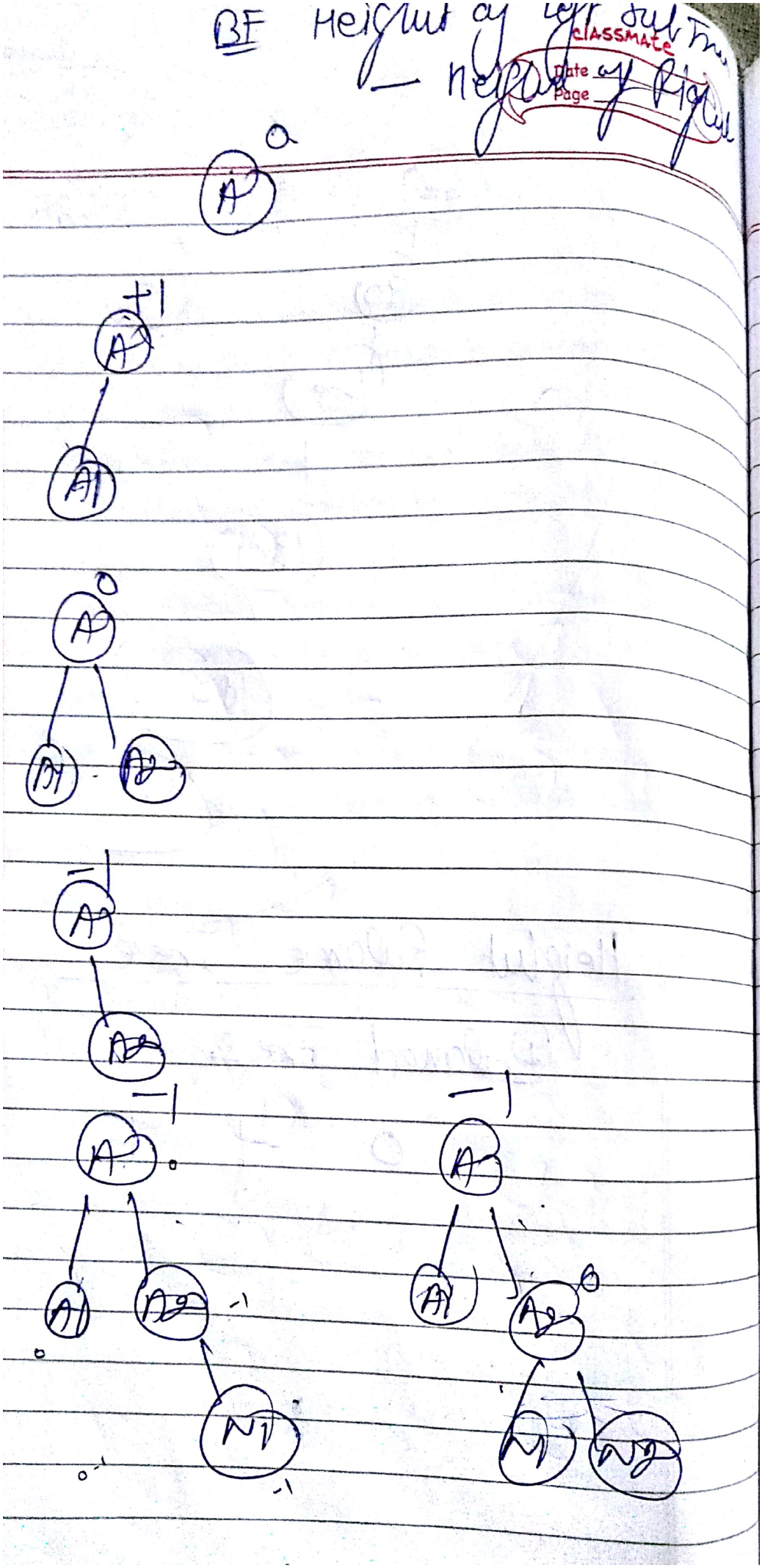


Height Balance Tree

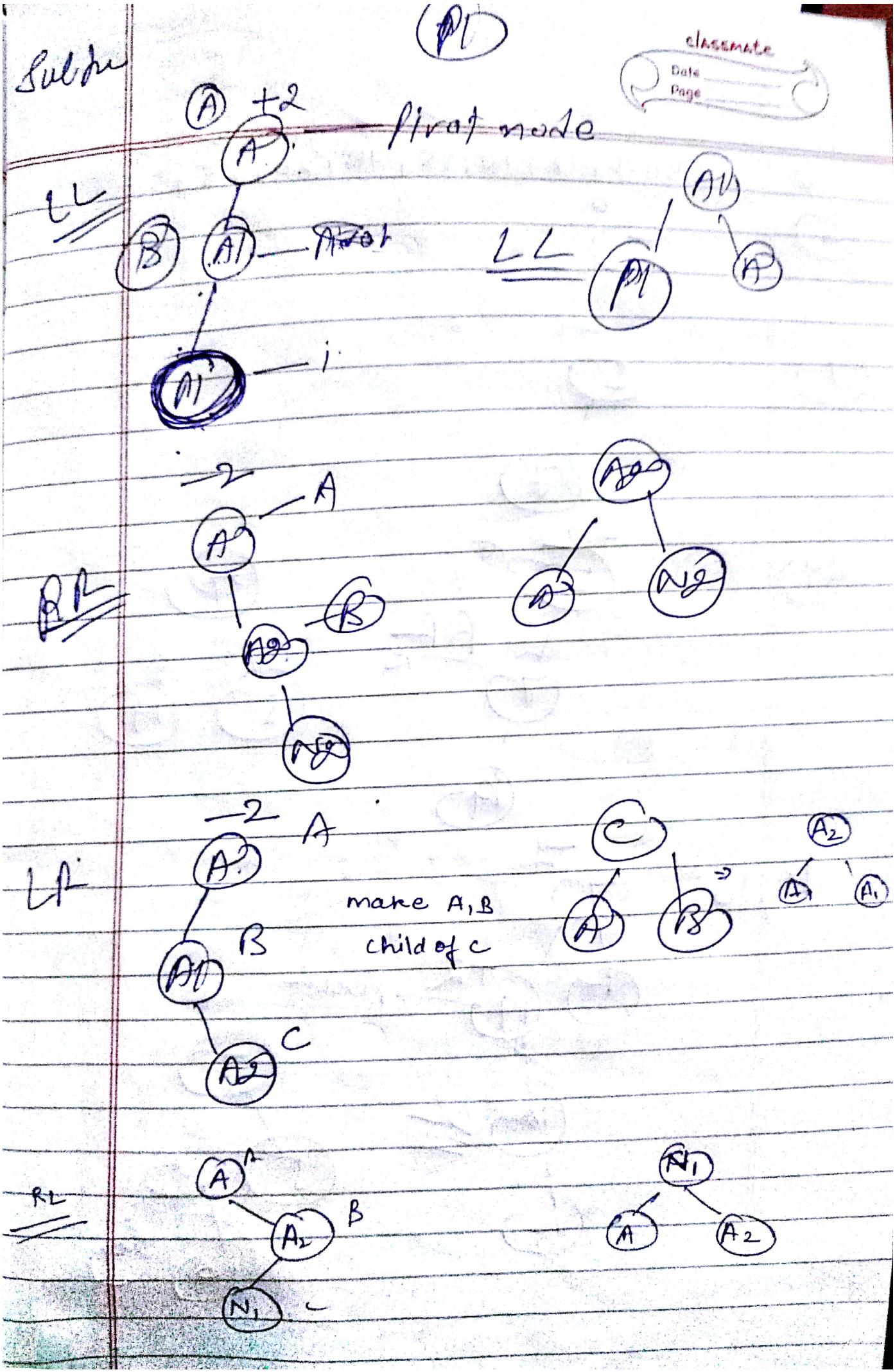
Balanced Tree

0 -1 1

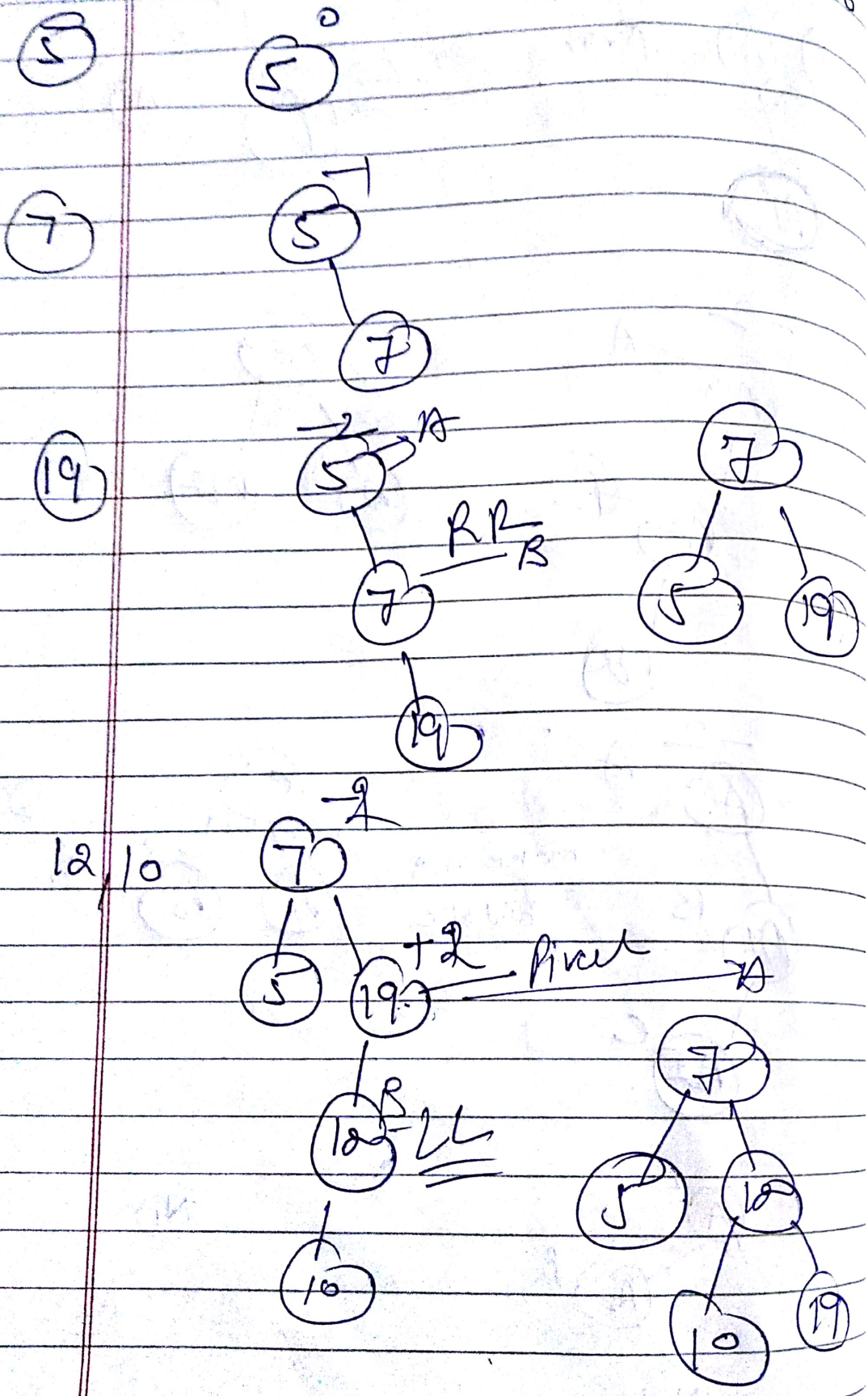
BF Height of left subtree − height of right subtree

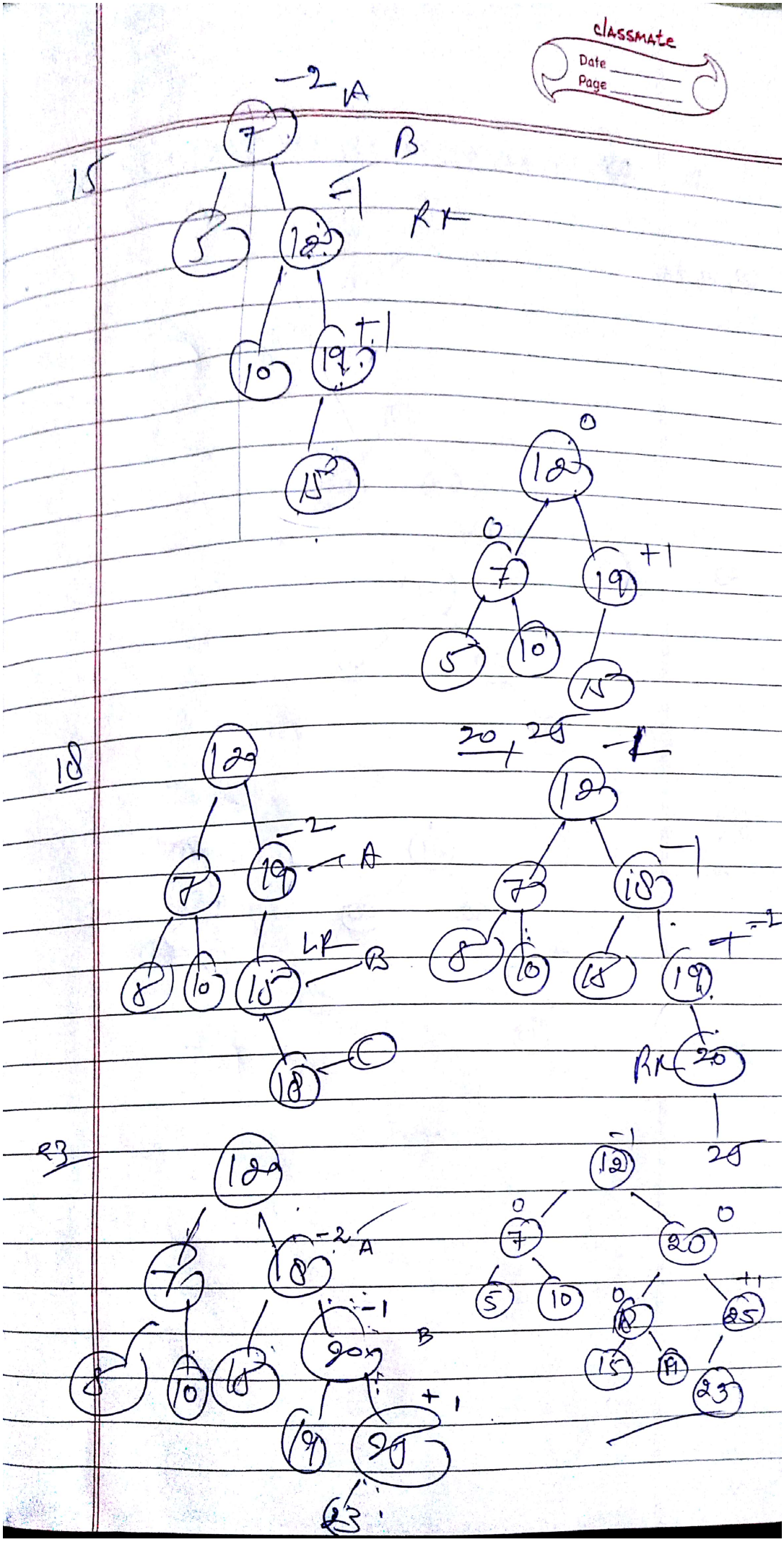


Subha



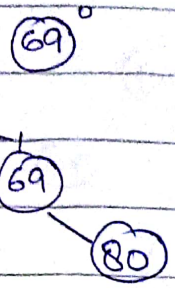
Q. 5, 7, 19, 12, 10, 15, 18, 20, 25, 23



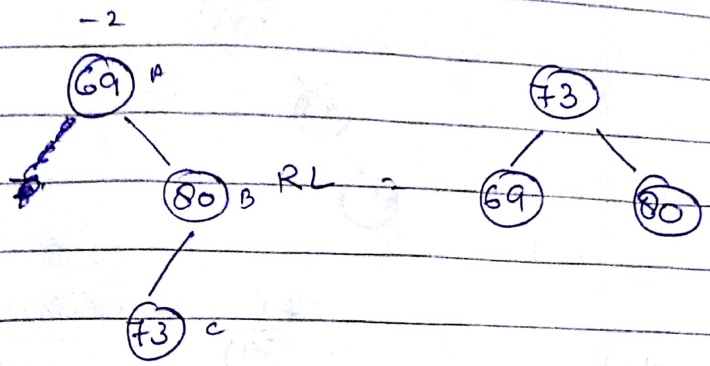


Q. 69, 80, 73, 40, 33, 70, 1, 86, 90

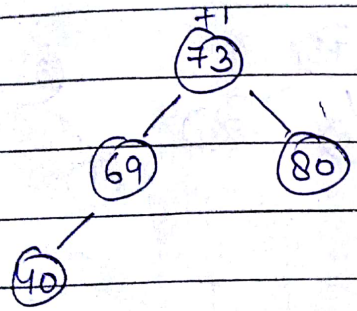
69



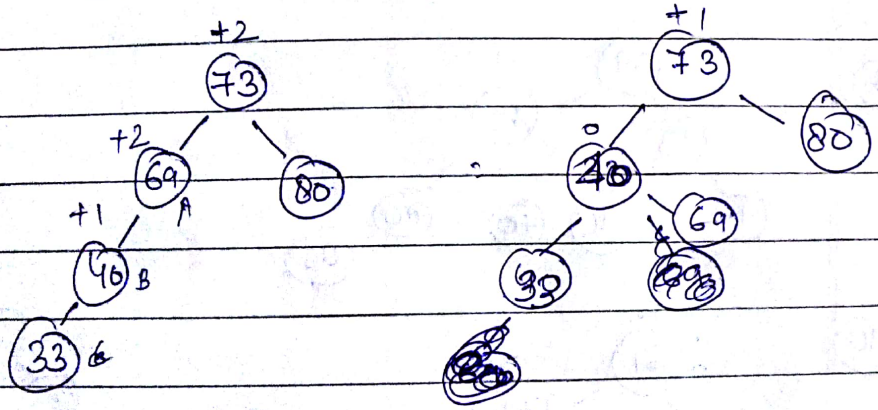
80



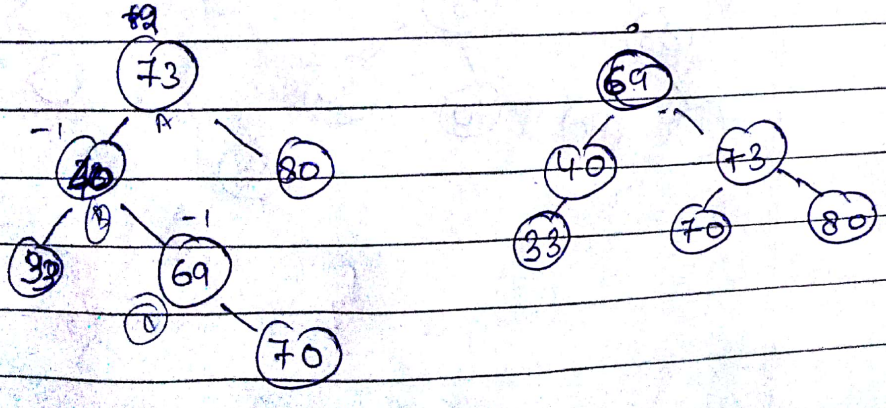
73



40

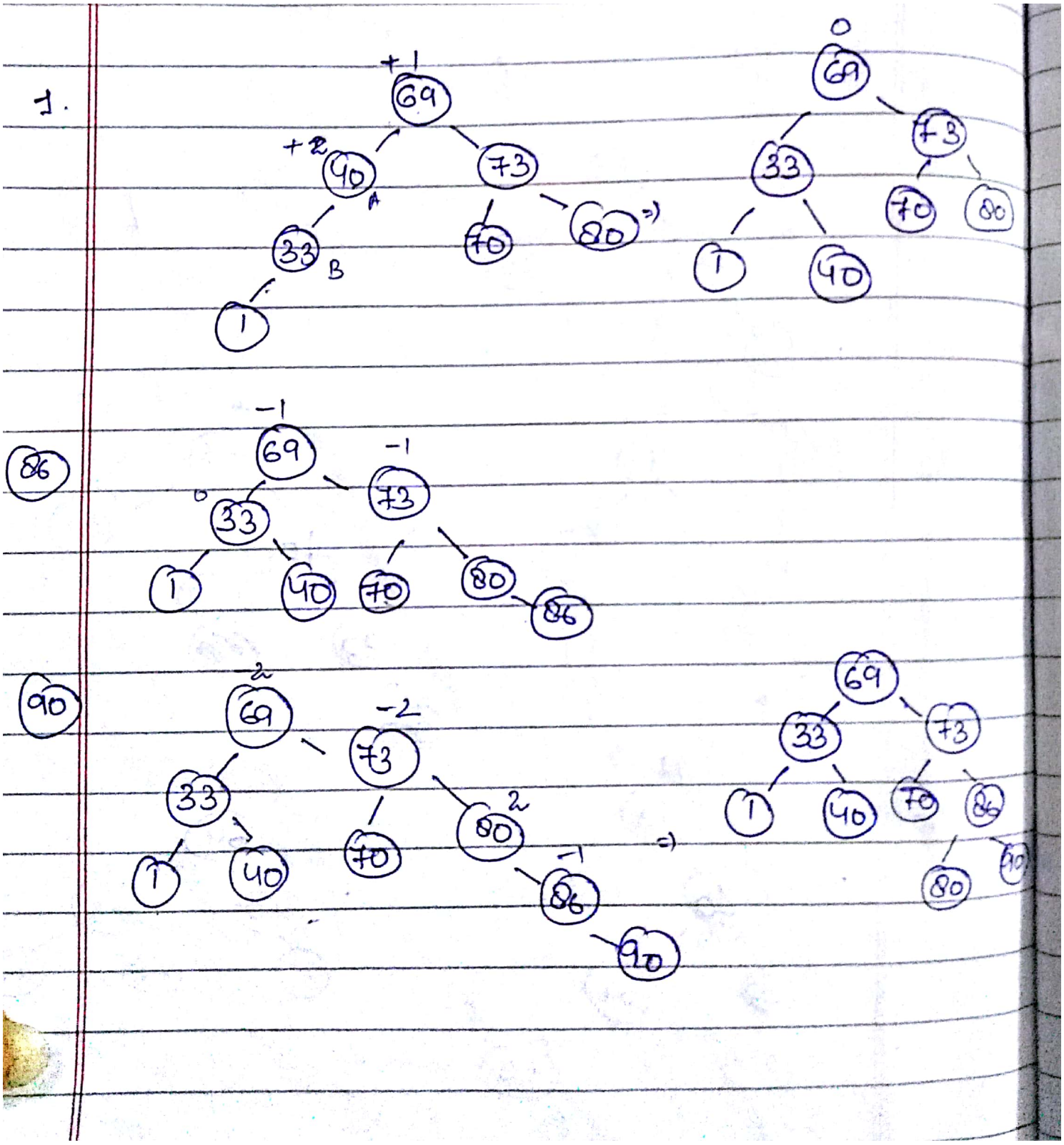


33



70

1.



Heap Tree

There can be 2 types of heap tree

(i) Max heap (ii) Min heap

Max heap => Root will always be maximum

Min heap => Root will always be minimum

Max heap

=> Root => max

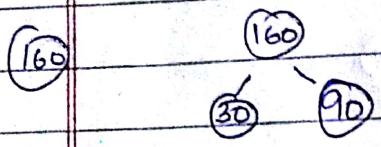
=> it will work on index array

90, 30, 160, 45, 100, 130, 200, 65, 87, 12, 4

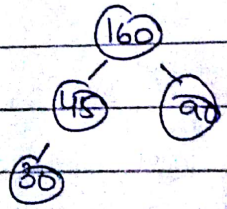
90

30

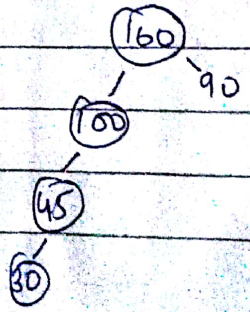
160



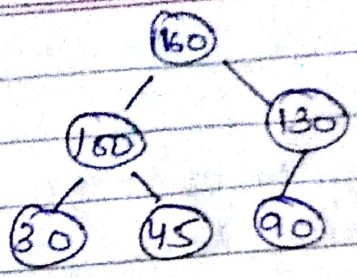
45



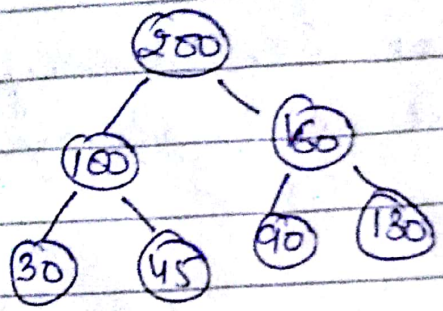
100



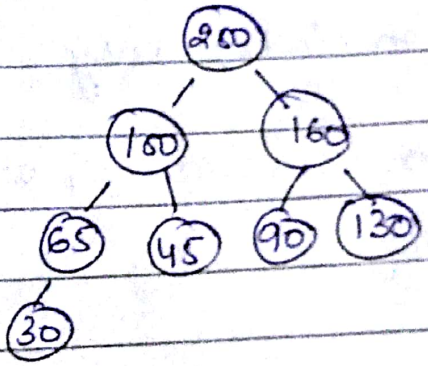
130



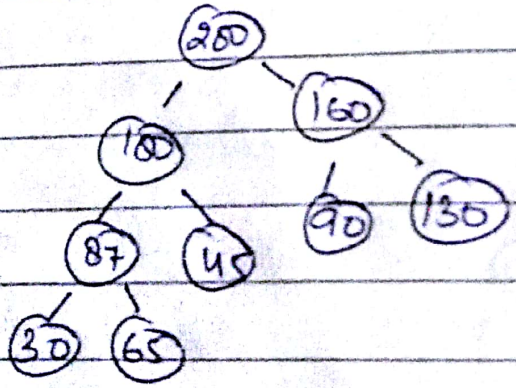
200



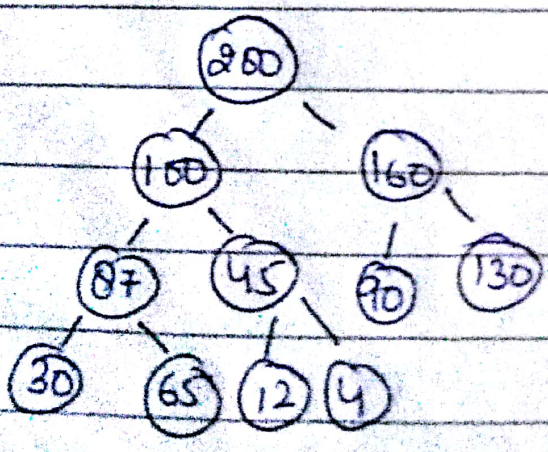
65



87

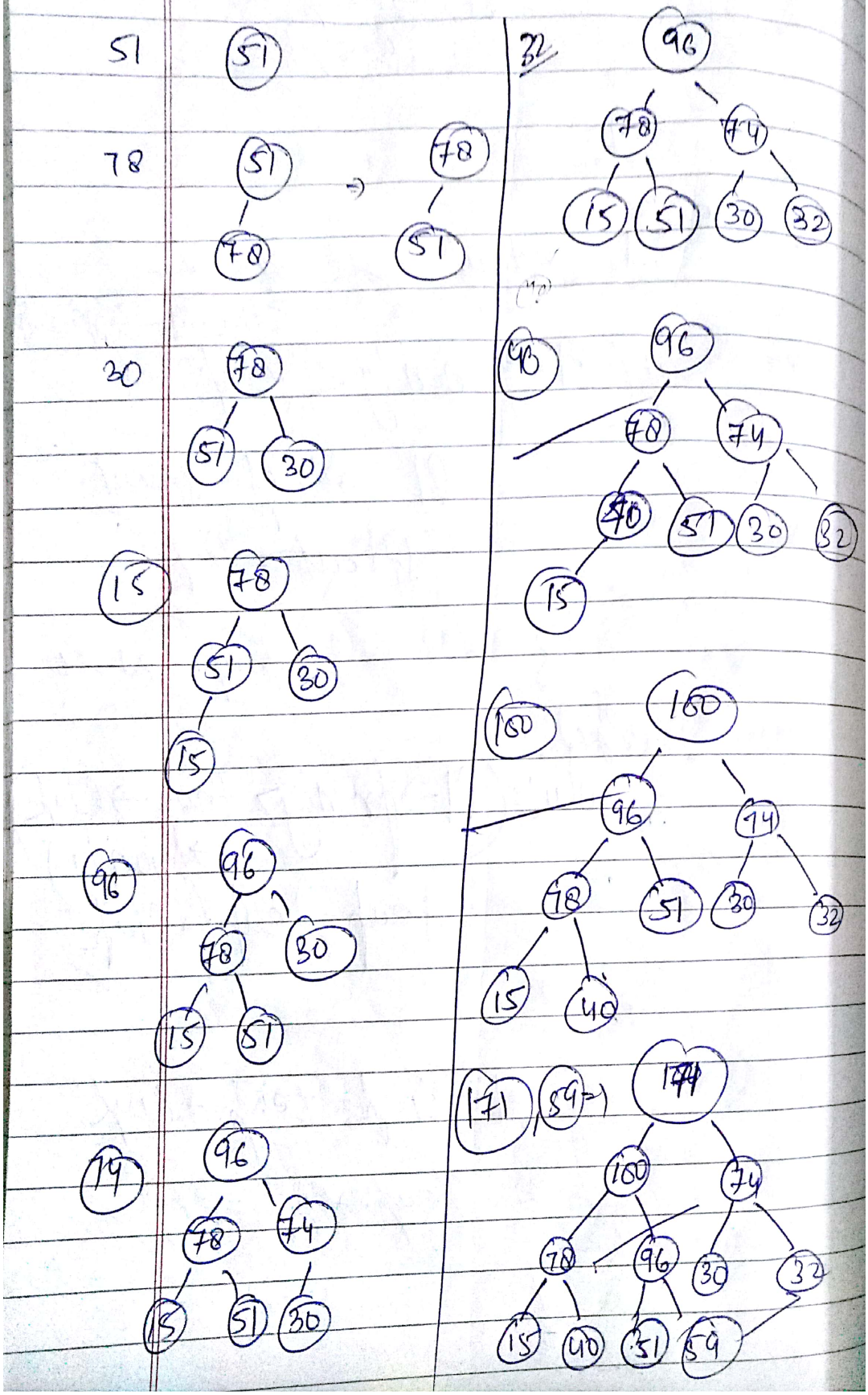


12,4

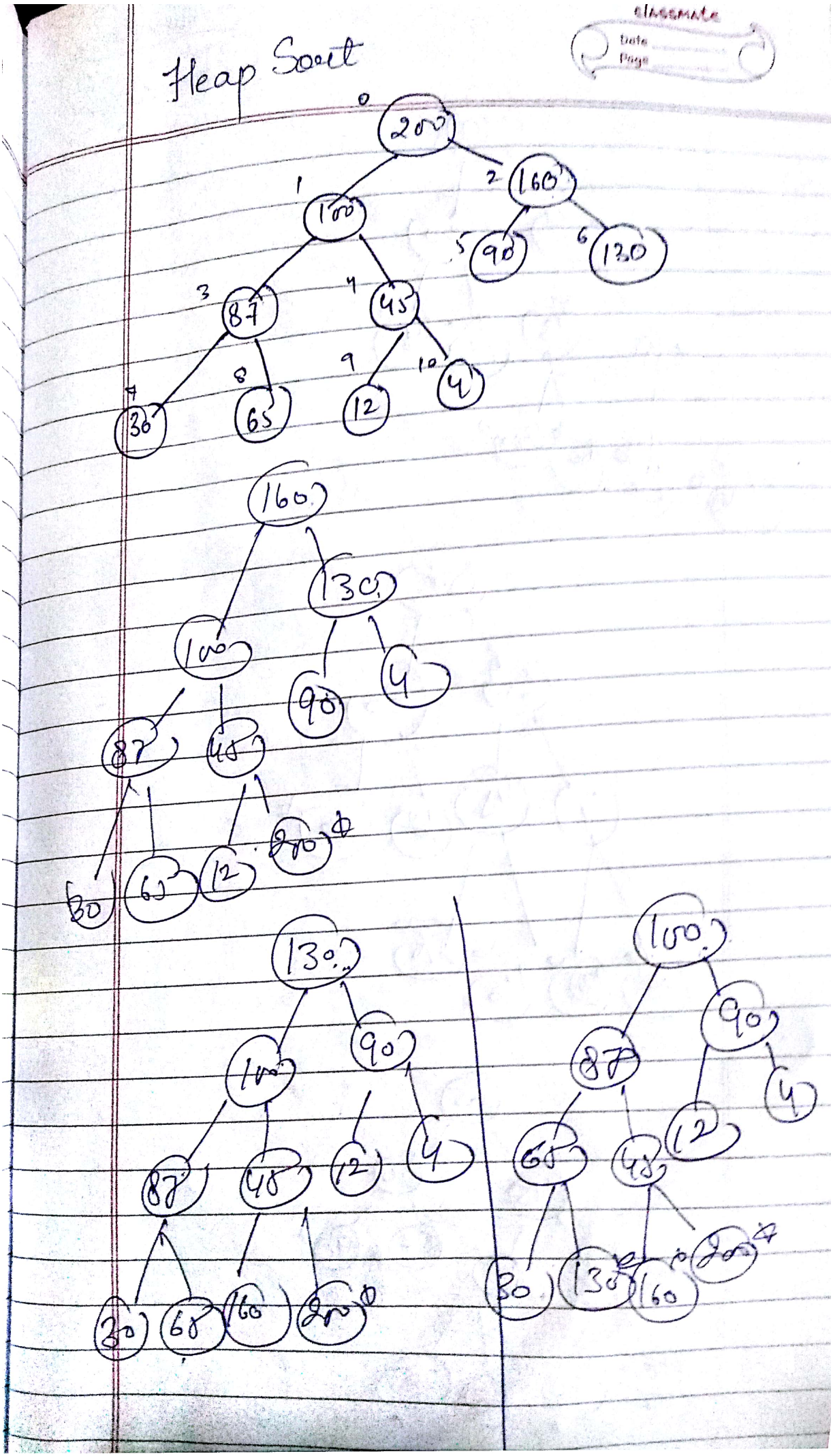


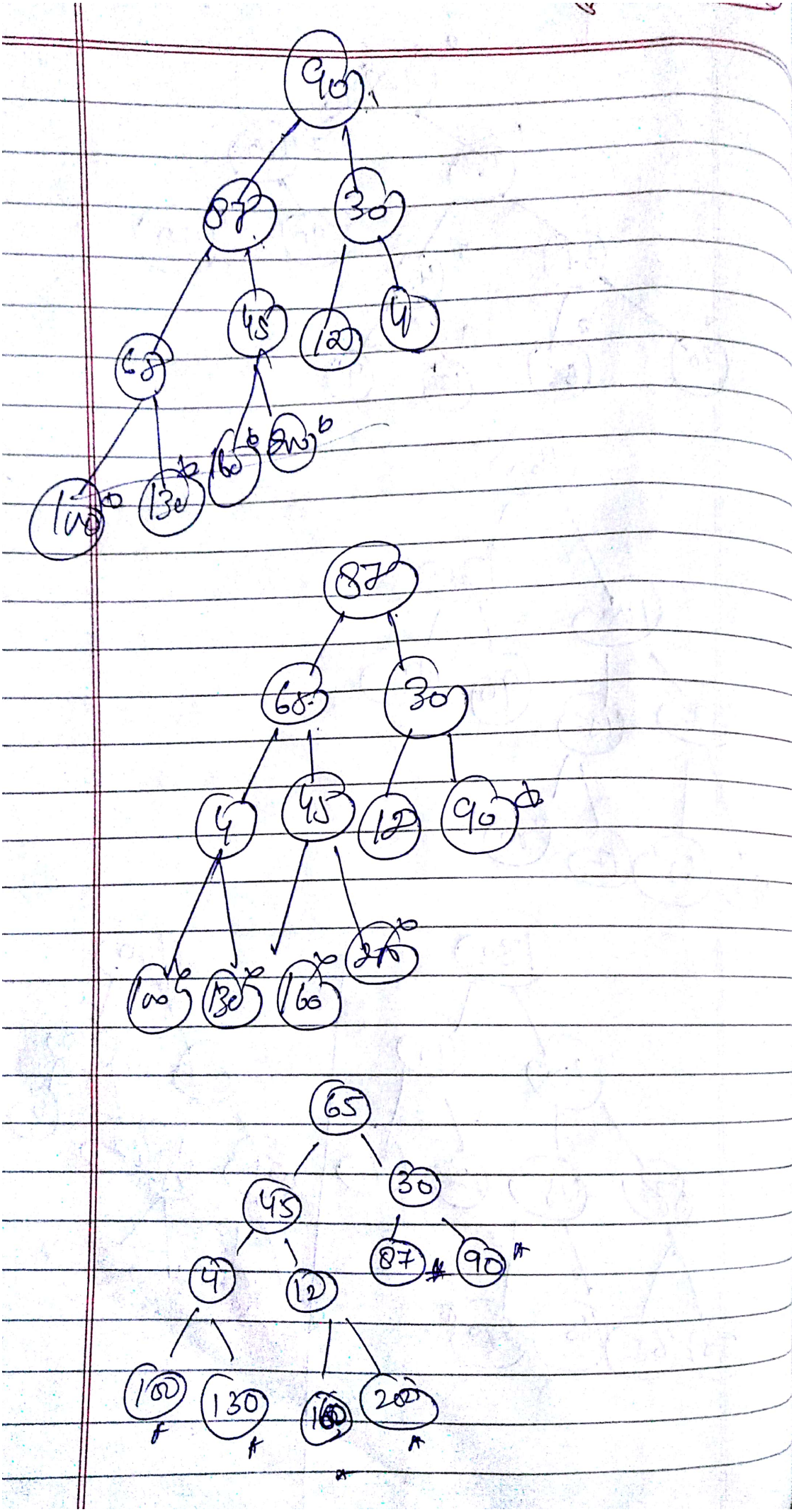
Implementation

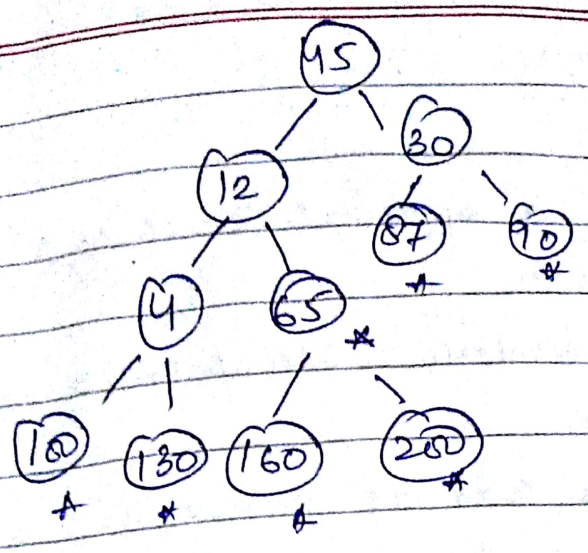
Q) 51 78 30 15 96 74 32
40 100 171 59

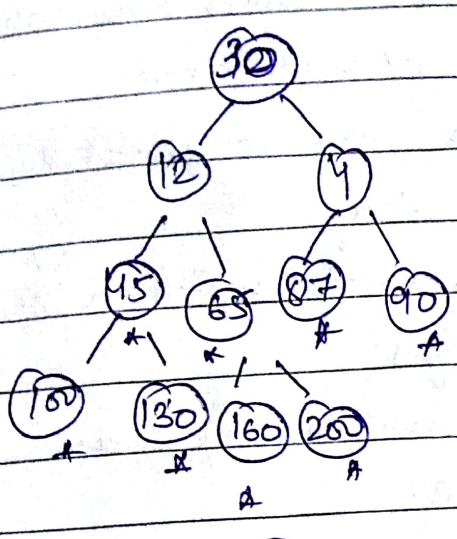


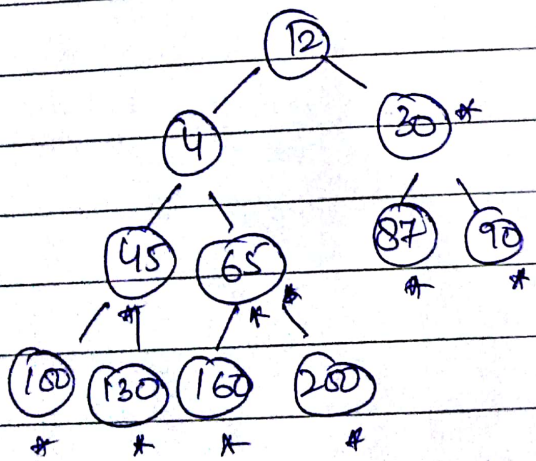
Heap Sort

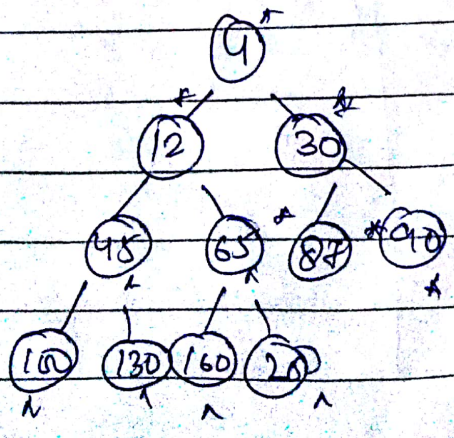


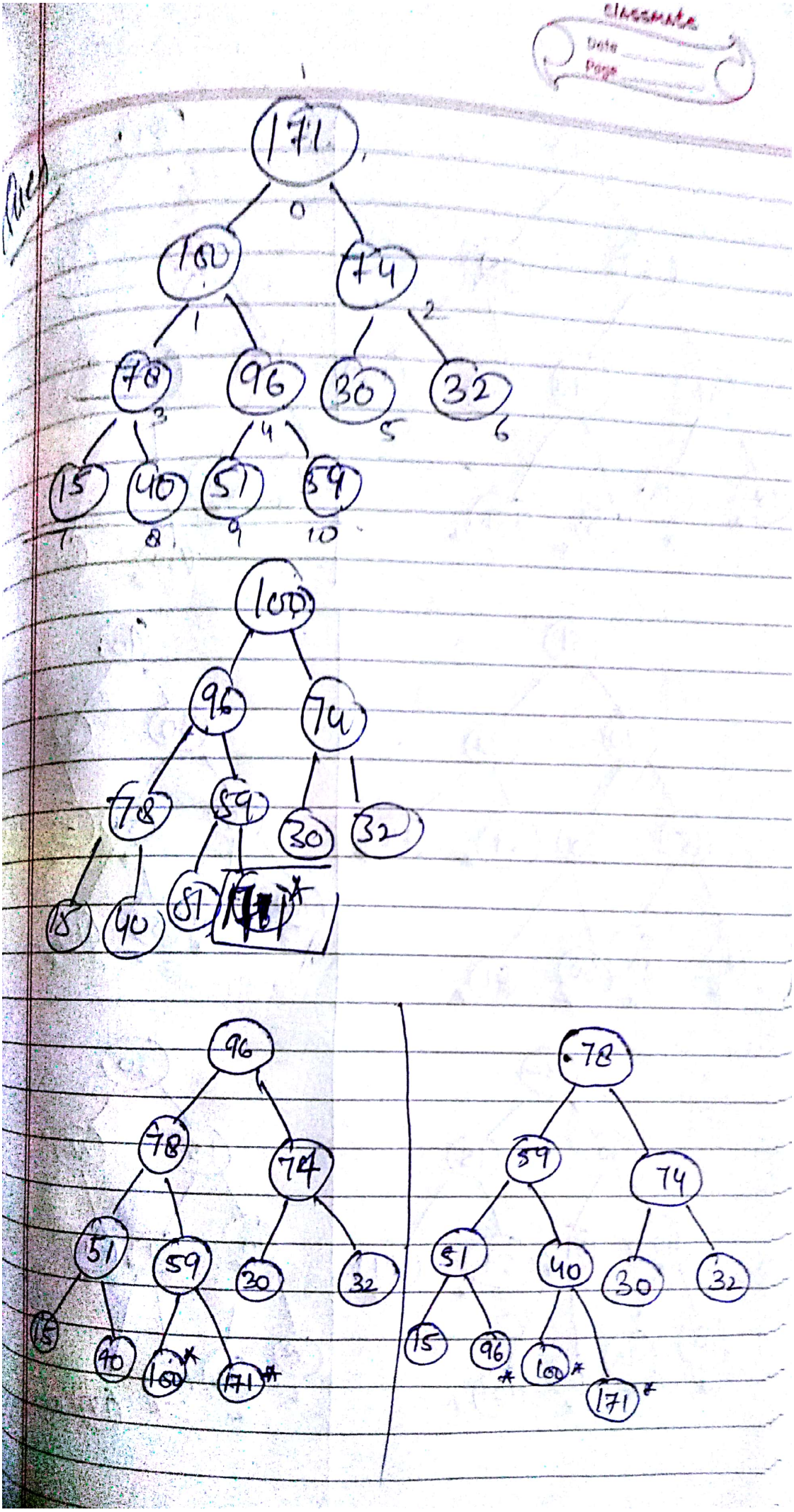


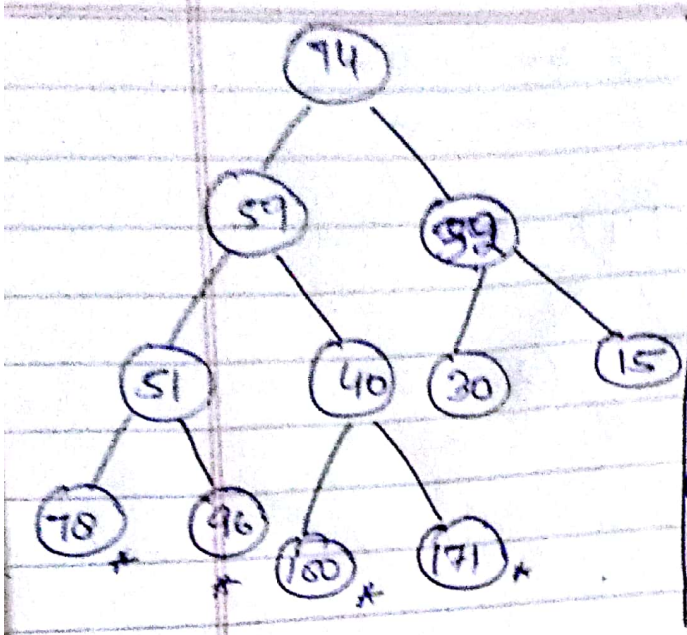


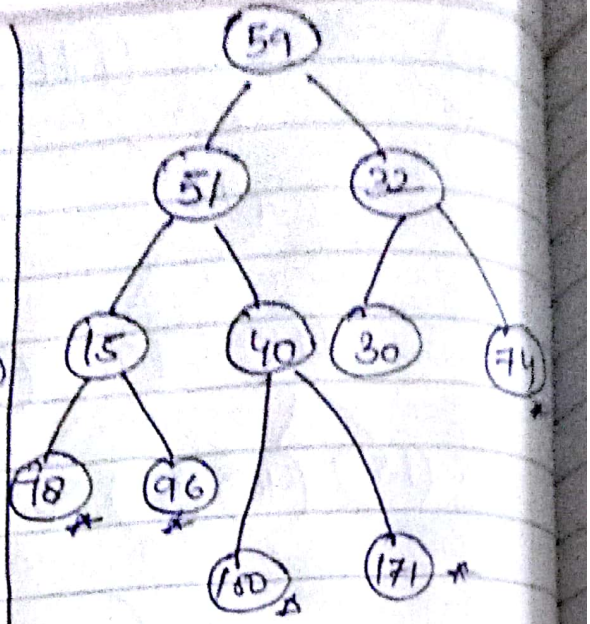


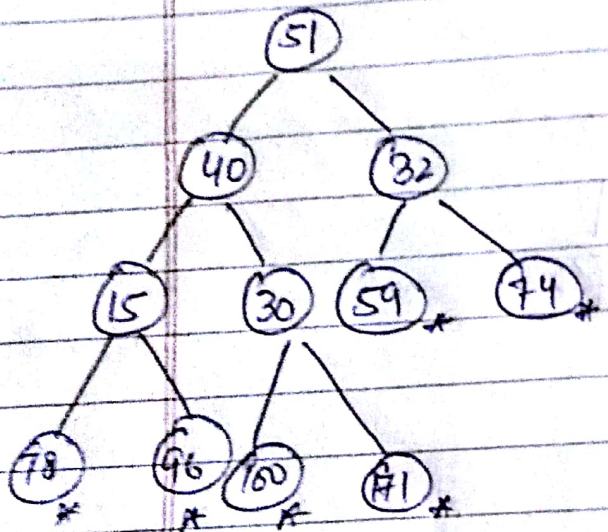


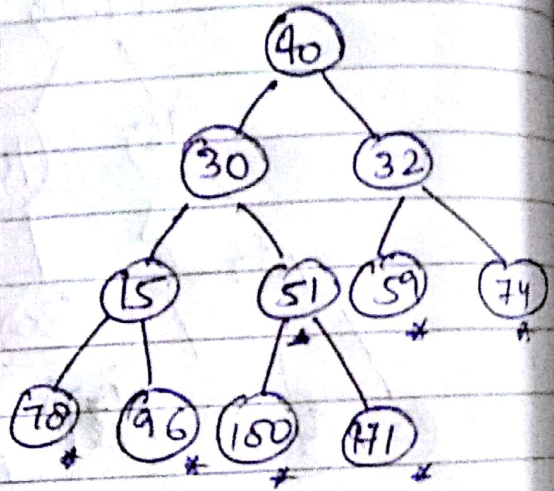


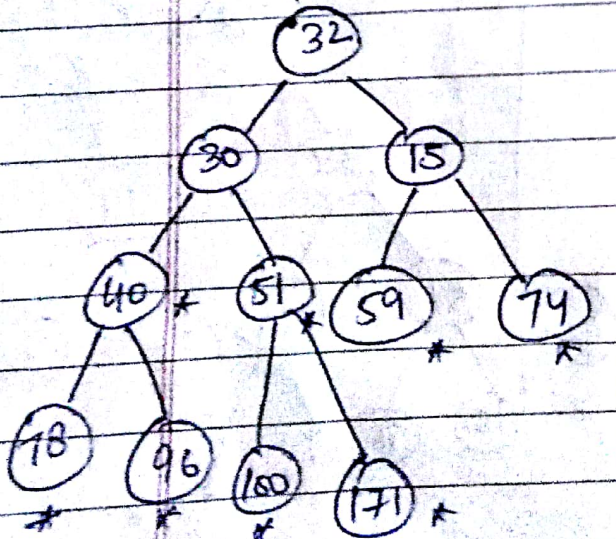


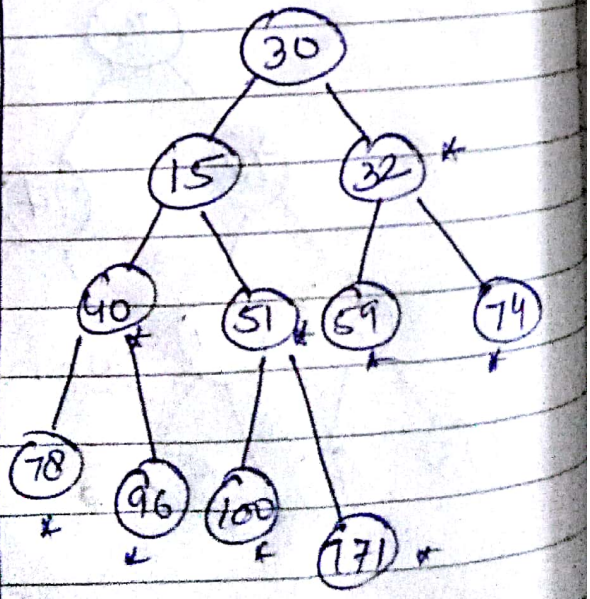


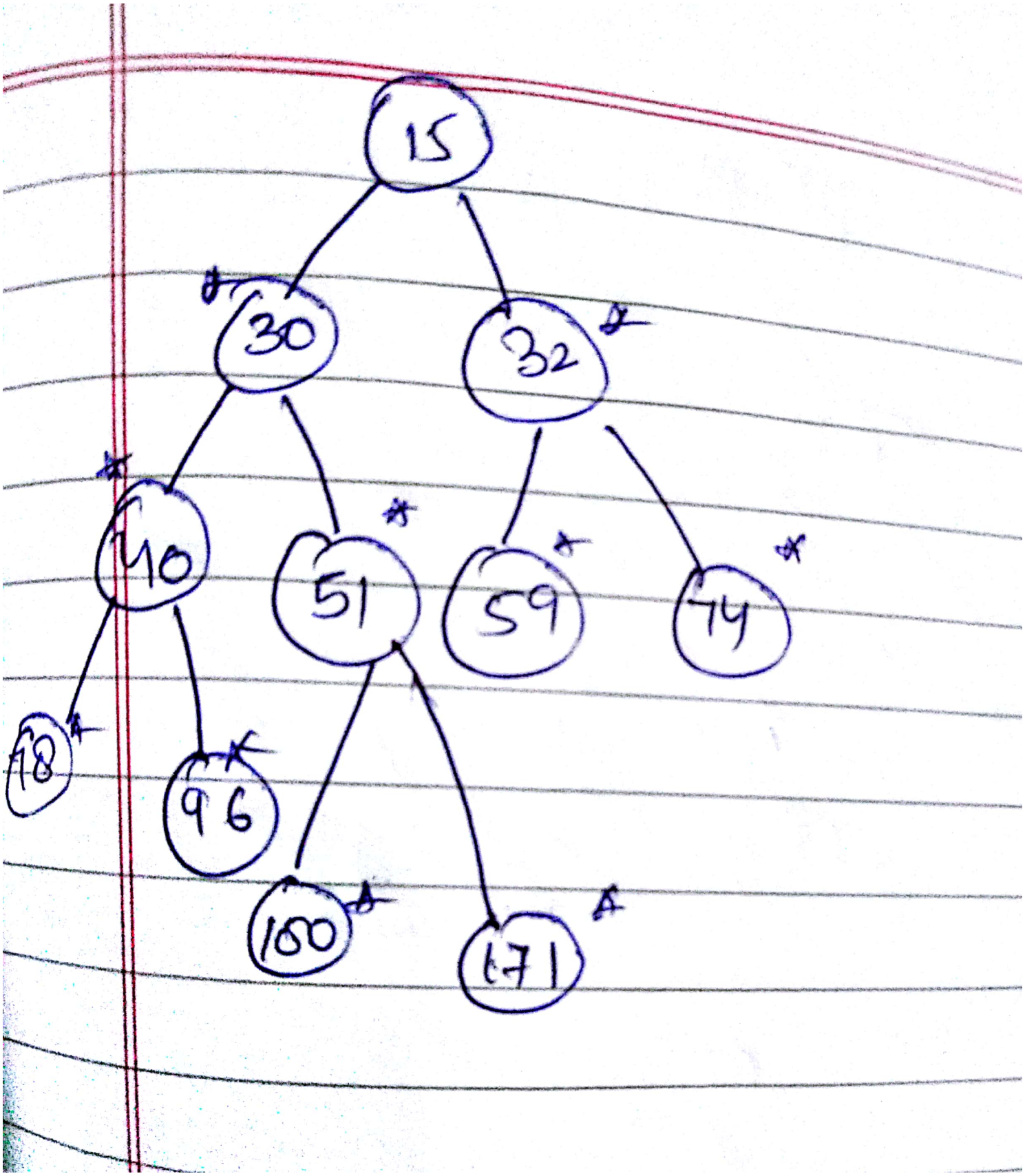












Threaded Binary Tree

- There are two types of threaded binary trees
  - single threaded
  - double threaded

- Single threaded :- where NULL right pointers is made to point to the inorder sucessor (if successor exists).

- Double threaded : where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively.

→ In simple threaded binary tree, the NUL right pointers are used to store inorder sucessor.

→ where- ever a right pointer is NULL, it is used to store inorder sucessor.

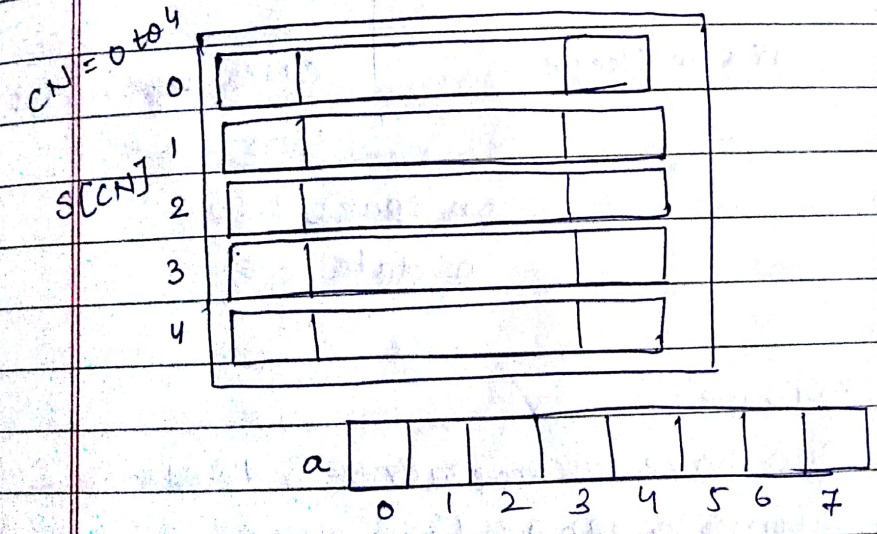
Structure Of threaded BT

```
Struct Node
{
  int Key;
  struct Node * left, * right;
  bool isThreaded;
};
```

Hashing

Searching : Time complexity

- Linked list
  - O(n)
- Array (unsorted)
  - O(n)
- Array (Sorted) [Binary search]
  - $O(\log_2 n)$
- Binary search Tree
  - $O(\log_2 h)$
- Hash Table
  - O(1)

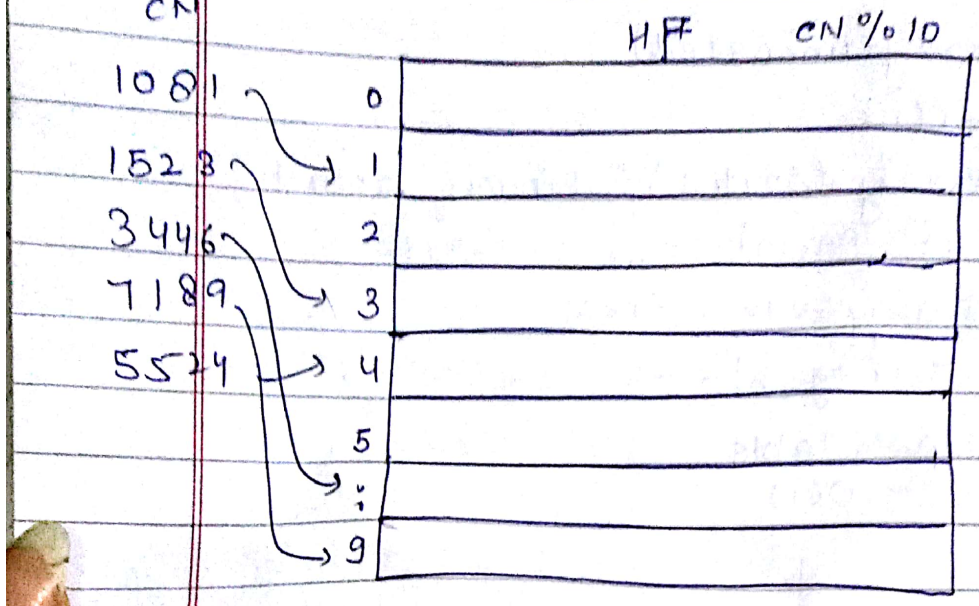



```
struct student
{
int roll no;
char name[20];
int age;
};
```

Example 1

- To store the key/value pair, we can use simple array like D.S where keys directly can be used as index to store values.

↓ = HF(CN)



n ≤ m ≤ keys

CN (KEY)

1000 - 8999

m spaces = 10

n data = 5

Example 2

- We have 10 complaints indexed with a number ranges from 0 to 9 (say complaint no)
- We have an array of 10 spaces to store 10 complaints
- Here we can use complaint no: as key and it is nothing but index no in an array

Terms

- Hash Table
  - Mostly it is an array to store dataset. It is the data structure

- Hash Function
  - A hash function is any function that can be used to map dataset of arbitary size to dataset of fixed size which falls into the hashtable
  - The values returned by a hash func are called hash values, hash codes, hash sums, or simply hashes.

- Hashing
  - In hashing, large keys are converted into small ones by using hash function and then the values are stored in data structure called hash tables.

Hash Functions

- A hash func. usually means a func. that compresses, meaning the o/p is shorter than the i/p.
- A hash function is any func. that can be used to map data of arbitrary size to data of fixed size.

Parameters of good hash func.

- Easy to compute
- Even distribution
- Minimize collisions.

Perfect Hashing

- Perfect hashing maps each valid i/p to a different hash value (no collision).

Division Remainder method

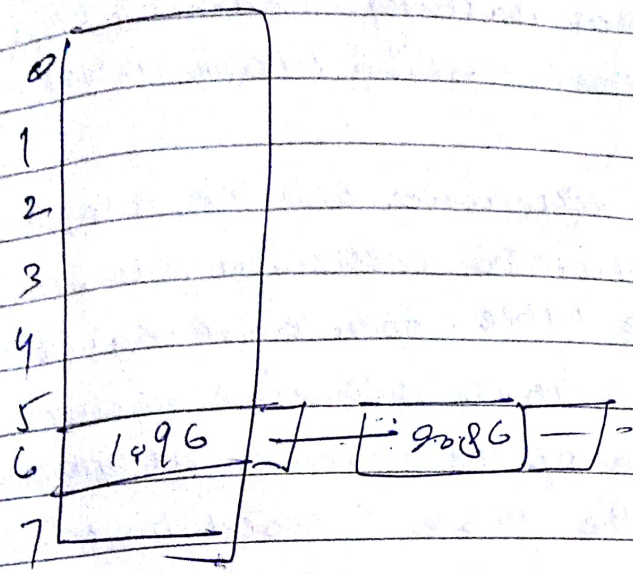
In DSM, we will calculate the address by using remainder i.e divide the key with the no. of available space & whatever the remainder will be the location of search record.

Mid-Square method

In MSM, squares the keys value & then take out the middle arbits of the result and giving a value in the range 0 to 2^{r-1} (r ⇒ digits of mid)

Recall Hashing

Collisions



1096

2086

- A situation when the resultant hashes for two or more data elements in the data set, maps to the same location in the hash table, is called a hash collision.

- In such a situation two or more data elements would qualify to be stored/mapped to the same location in the hash table.

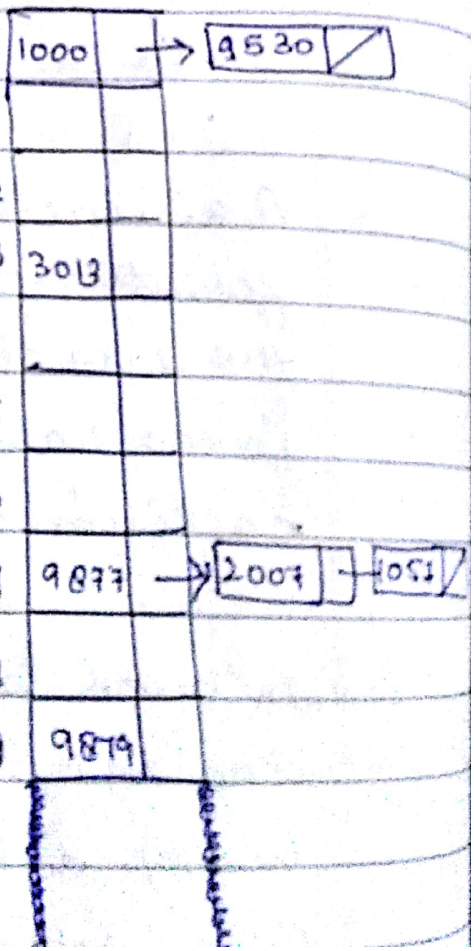
Hash collision resolution

- Two types of collision resolution
  - Open Hashing (chaining)
  - closed Hashing (open Addressing)

- The difference b/w the two has to do with
  - whether collisions are stored outside the table (open hashing), or
  - whether collisions result in storing one of the records at another slot in the table (closed hashing).

Open Hashing

- The simplest form of open hashing defines each slot in the hash table to the head of a linked list.
- All records that hash to a particular slot are placed on that slot's linked list.



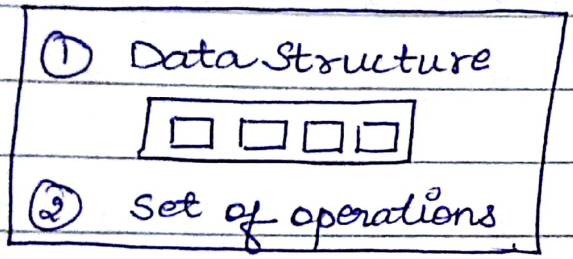
Closed Hashing

- Linear Probing
- Quadratic Probing
- Double Hashing

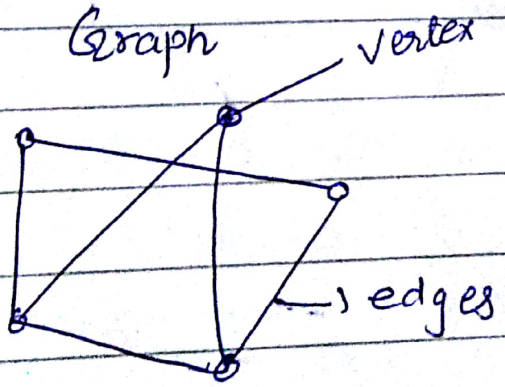
Graphs

- A graph is an abstract data type meant to implement the graph concepts from mathematics.
- A graph is a collection of nodes, connected by edges.

Abstract data type



Example :- through graph we can represent which cities are connected by railway tract from each other example two friends relationship



Page

Definition

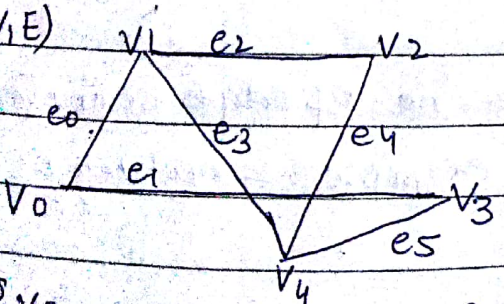
- In computing
  - A graph is an abstract data structure that implements the mathematical definition of a graph

- In mathematics
  - A graph G is composed by a set of N vertices or nodes, connected through a set E of edges, or links
  - $G = (E, V)$

Definition of Graph

- A graph G consists of two things
  - A set V of elements called nodes (or points or vertices)
  - A set E of edges such that each edge e in E is identified with a unique (unordered) pair [u, v] of nodes in V, denoted by $e = [u, v]$.
  - We indicate the parts of the graph by writing $G = (V, E)$

$G = (V, E)$



$e_0 = (V_0, V_1)$

$e_1 = (V_0, V_3)$

$V = \{V_0, V_1, V_2, V_3, V_4\}$

$E = \{e_0, e_1, e_2, e_3, e_4, e_5\}$

Date
Page

Data fit into graph structure

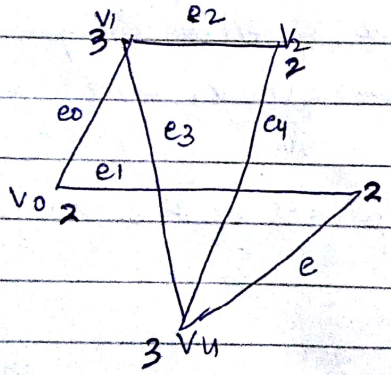
- Set of cities connected via rail route
- Set of people, where some are friends
- Connection among computers (routers)
- Tree is a graph.

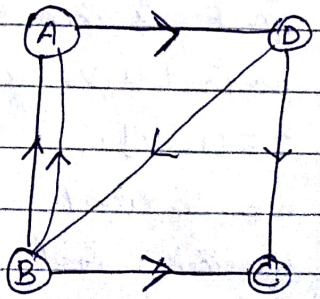
Adjacent Nodes

- If e=[u,v], the u and v are nodes called end points of e. u and v are said to be adjacent nodes or neighbours.

Degree of node

- The degree of node u, written deg(u), is the number of edges containing u.





Out Degree :- no. of edges beginning at in given example: 2 outdeg of C is 0.

In Degree :- no. of edges ending at V indegree

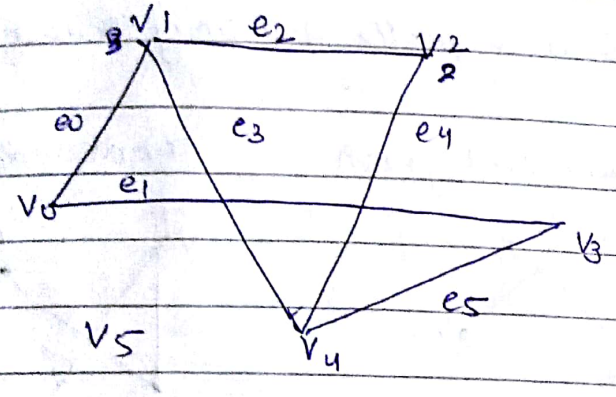
of c will be 2

Source node indegree(0):- no source node in given graph.

Sink node outdegree(0) :- c is sink node

isolated node

- If deg(u) = 0, then u is called an isolated node.



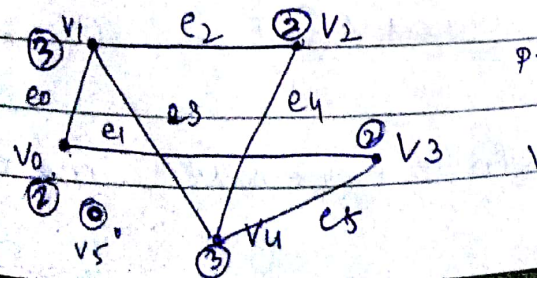
Path

- A path P of length n from a node u to a node v is defined as a sequence of n+1 nodes.

  P = (V0, V1, V2, ----, Vn)

Closed path

- The path is said to be closed if V0 = Vn.



P = (V0, V1, V2)

V0 V1 V2 V4 V3 V0

Date
Page

Simple path

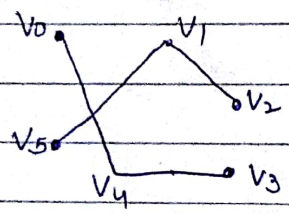
- The path is said to be simple if all the nodes are distinct, with the exception that V_0 may equal V_n.

$V_0\ V_1\ V_4\ V_2\ V_1\ V_4\ V_3\ V_0$

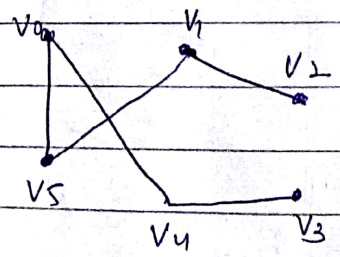
Connected graph

- A graph G is said to be connected if there is a path b/w any two of its nodes.

not connected graph



connected graph



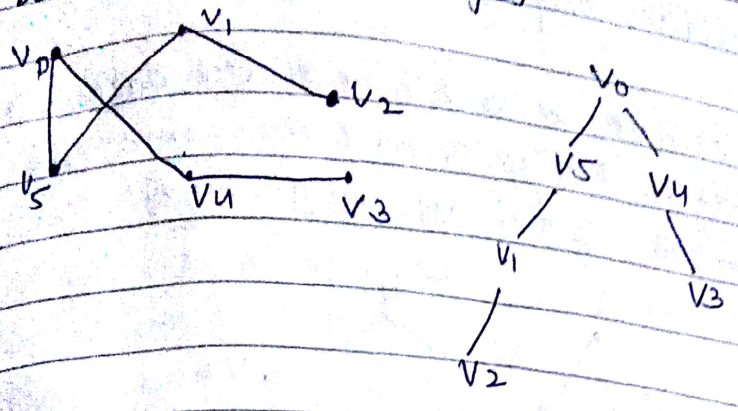
Tree

- A connected graph T without any cycles is called a tree graph or free tree or simply a tree.

- This means in particular, that there is a unique simple path P between any two nodes u and v in T.

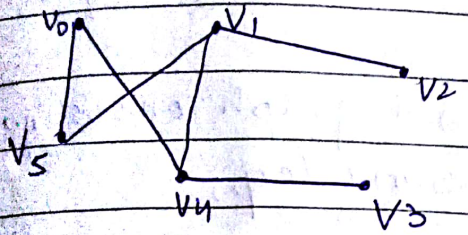
- If T is finite tree with no nodes, then T

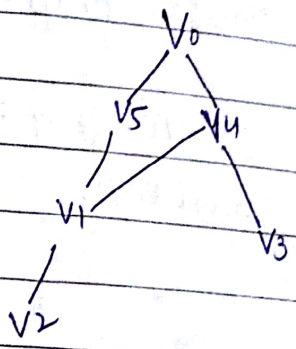
will have m-1 edges.



Graph without cycle

Graph with cycle

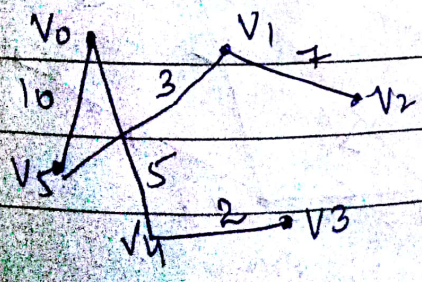




labelled graph

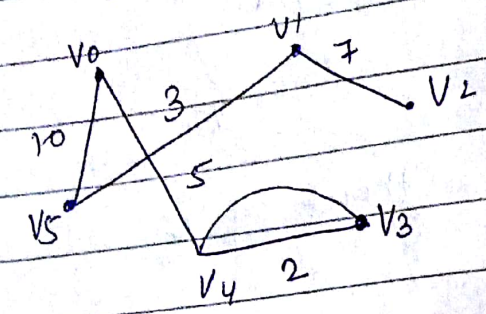
- A graph is to be labeled if its edges are assigned data.

- A graph G is said to be weighted if each edge e in G is a non negative numerical value w(e) called the weight or length of e



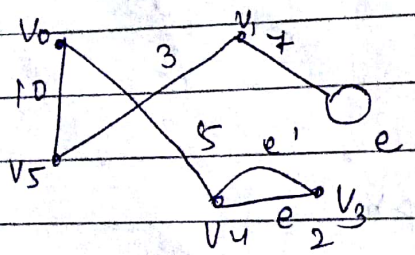
Multiple edges

- Distinct edges e and e' are called multiple edges if they connect the same end points, that is, if $e=[u,v]$ and $e'=[u,v]$



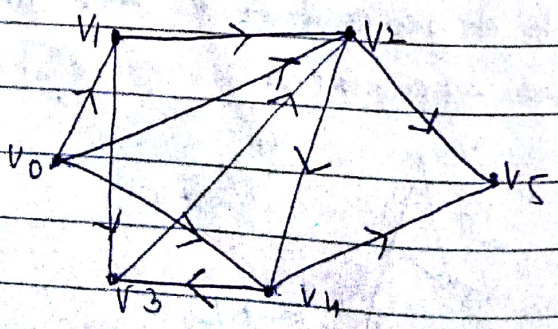
Multi-graph

- Multi-graph is a graph consisting of multiple edges and loops.



Graph Types

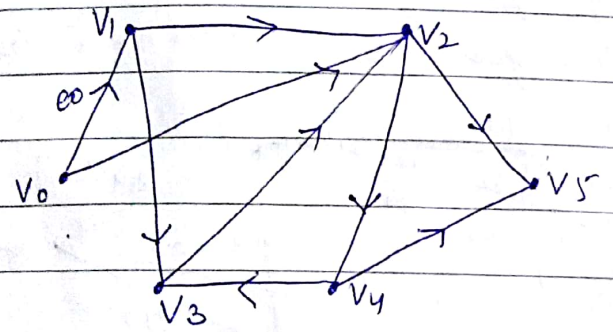
- Directed Graph
- Undirected graph



Directed Graph

- A directed graph G also called digraph or graph is same as multi-graph except that each edge e in G is assigned a direction, or in other words, each edge e is identified with an ordered pair (u,v) of nodes in G rather than an unordered pair [u,v].

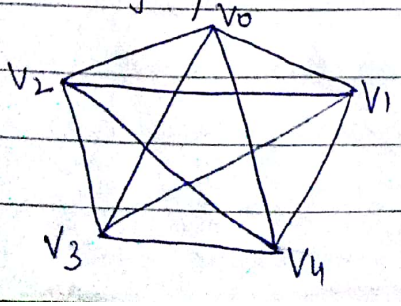
- Suppose G is directed graph with directed edges e=(u,v). Then e is also called an arc.



$e_0 = (V_0, V_1)$

Complete graph

- A simple graph in which there exists an edge b/w every pair of vertices is called a complete graph. It is also known as a universal graph or clique.

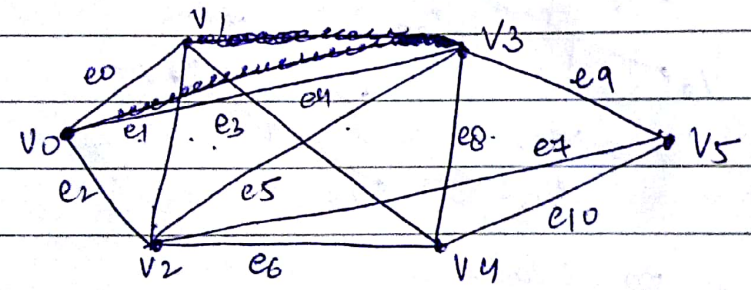


Null Graph

- A graph without any edge is called a null graph.
- In other words, every vertex in a null graph is an isolated vertex.

Sub-graph

- A graph $G' = (V', E')$ is a sub-graph of graph $G = (V, E)$ if V' is a subset of V and E' is a subset of E. Thus for G' to be a sub-graph of graph G, all the vertices and edges of G' should be in G



$G = (V, E)$

$V = \{v_0\ v_1\ v_2\ v_3\ v_4\ v_5\}$

$E = \{e_0\ e_1\ e_2\ e_3\ e_4\ e_5\ e_6\ e_7\ e_8\ e_9\ e_{10}\}$

$G' = (V', E')$ $V' \subset V$ $E' \subset E$

$V' = \{v_0\ v_3\ v_4\ v_2\}$

$E' = \{e_1\ e_2\ e_5\ e_6\ e_8\}$

Representation of graph DS

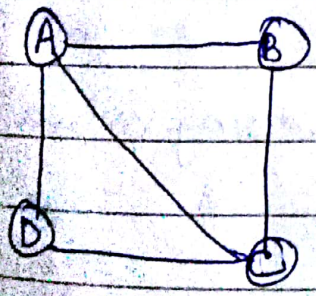
- There are 3 ways to represent graph data structure
  1. Adjacency Matrix
  2. Adjacency list
  3. Adjacency set

Adjacency Matrix

- To represent graph we need number of vertices, number of edges and also their interconnections.

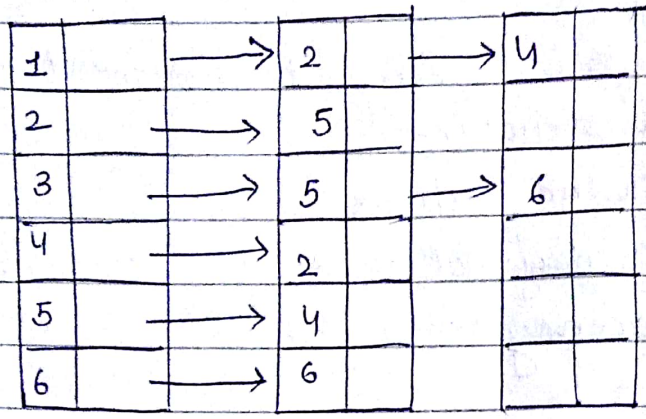
```
struct Graph {
    int V;
    int E;
    int **Adj;
};
```

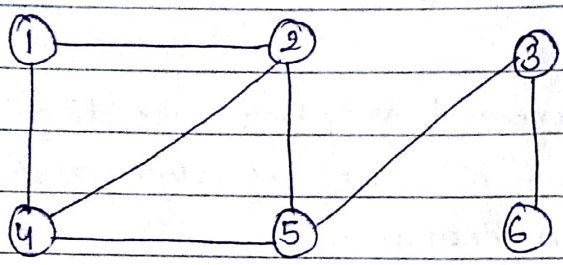
V = 4 E = 5



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

Adjecanay list representation





Graph traversal algorithms

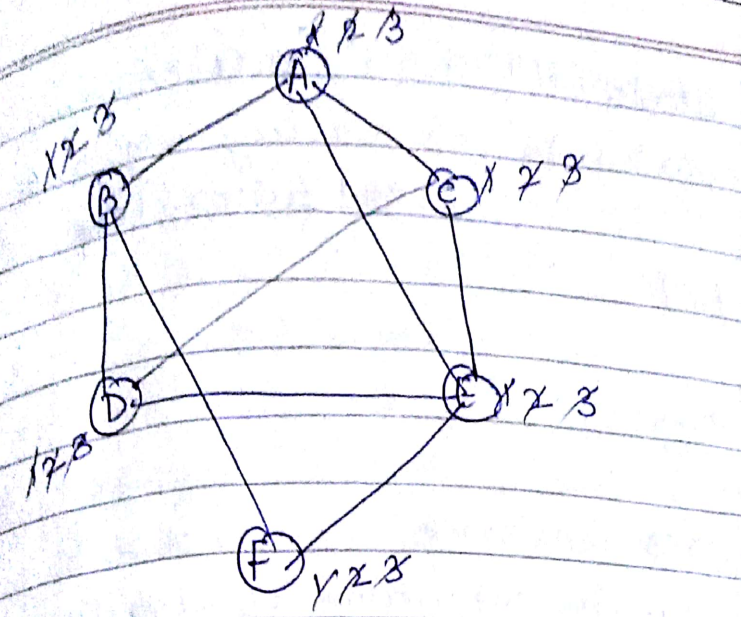
- Breadth First Search (BFS)
- Depth First Search (DFS)

BFS

- BFS uses queue
- Similar to level order traversing in tree

Logic

- During execution of algorithms, each node N of G will be in one of three state called status of N:
  - STATUS = 1 (Ready State)
  - STATUS = 2 (Waiting State)
  - STATUS = 3 (Processed State)



A B C E D F

A	B	C	E	D	F

Algorithm BFS : This algorithm executes a breadth first search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1)
2. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2)
3. [LOOP]
   Repeat steps 4 and 5 until Queue is empty:
4. Remove the front node N of Queue. Process N and change the status of N to the processed state (STATUS = 3)
5. Add to the rear of Queue all the neighbors of N that are in the ready

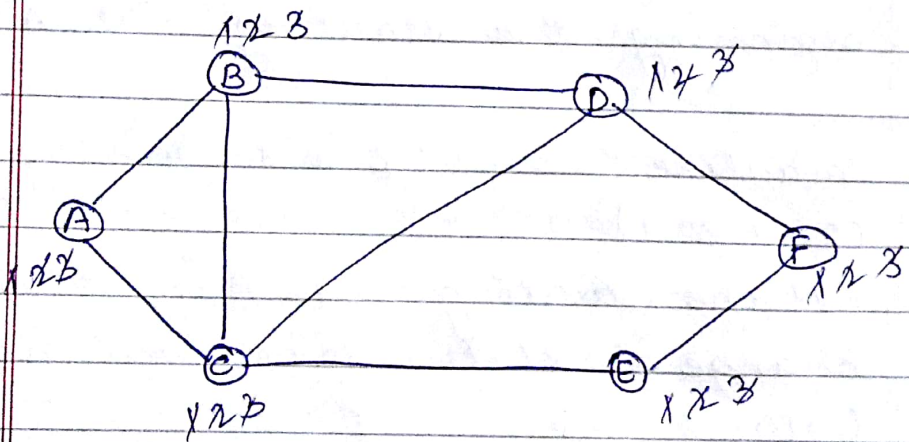
state (STATUS 2), and change their status to the waiting state (STATUS 2)

[end of Step 3 loop]

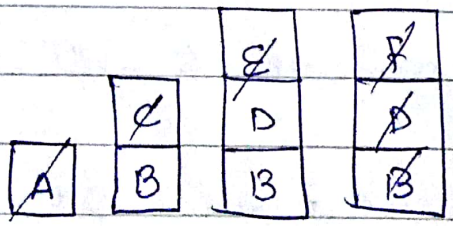
6. Exit

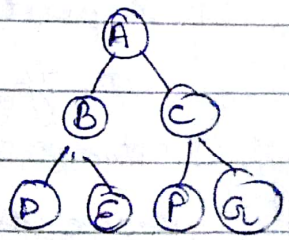
DFS

- DFS uses stack
- During execution of algorithms, each node N of G will be in one of three states called status of N:
  - STATUS=1 (Ready state)
  - STATUS=2 (waiting state)
  - STATUS=3 (Processed state)



ACEFDB

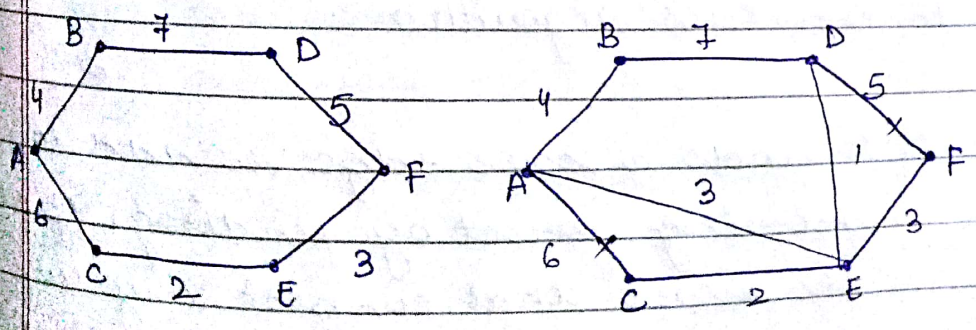




Algorithm DFS :- This algorithm executes a depth first search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1)
2. Push the starting node A onto the STACK and change its status to the waiting state (STATUS = 2)
3. [LOOP]
   Repeat steps 4 and 5 until STACK is empty:
4. Pop the top node N of Stack. Process N and change its status to the processed state (STATUS = 3)
5. Push onto STACK all the neighbors of N that are still in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
   [End of Step 3 Loop]
6. Exit.

Spanning Tree



① connect
② cost min

Minimal Spanning Tree

- We then wish to find an acyclic subset T that connects all of the vertices and whose total weight is minimized.

- Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it spans the graph G

- We call the problem of determining the tree T the minimum spanning tree problem.

Solution

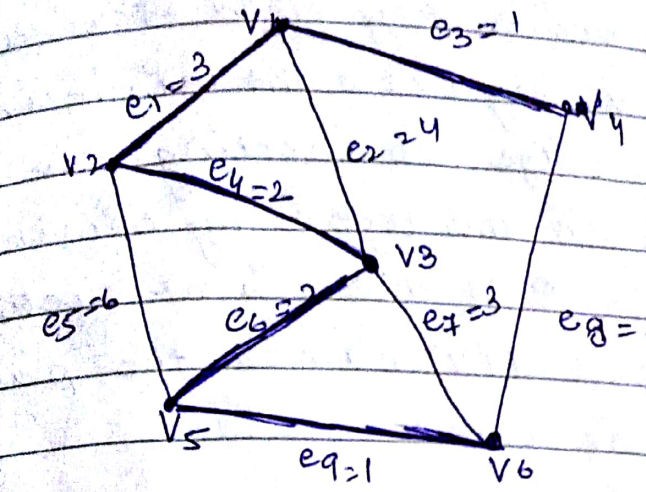
- Two algorithms for solving the minimum spanning tree problem
  - Kruskal's algorithm
  - Prim's algorithm

Kruskal's algorithm

- It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge $[u, v]$ of least weight.

Procedures

- MAKE-SET (V)
- FIND-SET (u)
- UNION (u, v)



S_1	$V_1, V_4, V_2, V_3, V_5, V_6$
S_2	~~V_2, V_3, V_5, V_6~~
S_3	~~V_3~~
S_4	~~V_4~~
S_5	~~V_5~~ V_6
S_6	~~V_6~~

$\{e_3, e_9, e_4, e_6, e_7, e_1, e_2, e_8, e_5\}$

Algorithm

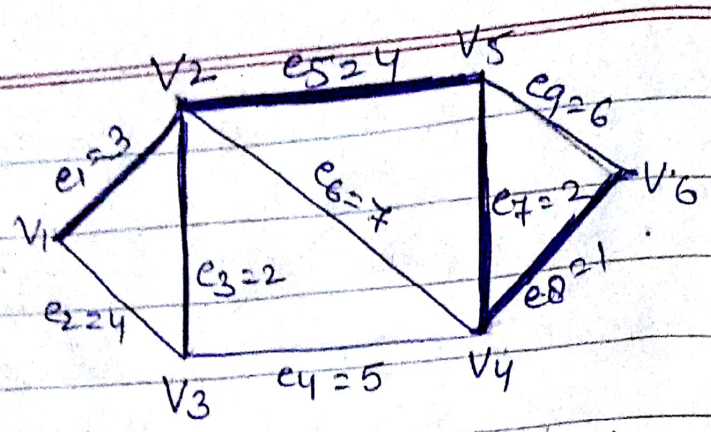
1. [Initialize] A := ϕ
2. [Loop] For each vertex $V \in V[G]$ do step 3
3. Make-set (v)
4. sort the edges of E into non decreasing order by weight w:
5. [Loop] For each edge $[u, v] \in E$ taken in non-decreasing order by weight do step 6.
6. If Find-set (u) ≠ Find-set (v) then;
   A := AU{[u, v]}, UNION (u, v)
7. Return A

Prim's Algorithm

Algorithm MSTPRIM (G, w, r): All vertices that are not in the tree reside in a min priority queue Q based on a key field. For each vertex v, key[v] is the minimum weight of any edge connecting v to a vertex in the tree; by convention key[v] = ∞ if there is no such edge. The field π(v) names the parent of v in the tree.

1. [loop] for each u ∈ V[G], steps 2
2. key[u] := ∞, π(u) := NIL
3. key[r] := 0
4. Q := V[G]
5. Repeat steps 6, to 8, while Q ≠ ϕ
6. u := EXTRACT-MIN(Q)
7. for each v ∈ Adj[u]
8. if v ∈ Q and w(u,v) < key[v]
   then: π[v] := u, key[v] := w(u,v)
9. Exit

V_1, V_2, V_3, V_5
V_4, V_6



	Key	π
~~V_1~~	0	-
~~V_2~~	~~∞~~ 3	~~V_1~~
~~V_3~~	~~∞~~ ~~4~~ 2	~~V_1~~ V_2
~~V_4~~	~~∞~~ ~~7~~ ~~6~~ 2	~~V_2~~ ~~V_3~~ V_5
~~V_5~~	~~∞~~ 4	V_2
V_6	~~∞~~ ~~6~~ 1	~~V_5~~ V_4

$U = V_1$
$V = \{V_2, V_3\}$
$3 < \infty$
$4 < \infty$

$U = V_2$
$V = \{V_3, V_4, V_5\}$
$2 < 4$
$7 < \infty$
$4 < \infty$

$U = V_3$
$V = \{V_4\}$
$5 < 7$

$U = V_5$
$V = \{V_4, V_6\}$
$2 < 5$
$6 < \infty$

$U = V_4$
$V = \{V_6\}$
$1 < 6$

Shortest path algorithm

- A famous solution for shortest path problem was given by Dijkstra.
- Dijkstra's algorithm is generalization of BFS algorithm.

Distance Table

Vertex	Distance[V]	Previous vertex which gave distance [V] (P[V])

- The algorithm works by keeping the shortest distance of vertex V from the source in distance table
- Distance [V] holds the distance from S to V
- The shortest distance of the source to itself is 0
- Distance [V] of all other vertices are set to INFINITY to indicate that those vertices are not yet processed.

Dijkstra's algorithm

- It uses greedy method
- It uses priority queue to store unvisited vertices by distance from s
- Doesnot work with -ve weights

Algorithm

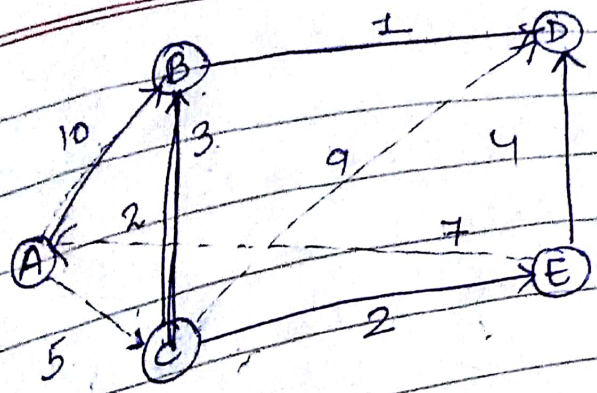
DIJKSTRA(G, w, s)

1. Initialize-single-source(G, s)
2. S = ϕ, Q = V[G]
3. Repeat thru steps 4 to 6 while Q ≠ ϕ
4. u = Extract-Min(Q), S = S ∪ {u}
5. Repeat for each vertex v ∈ Adj[u]
6. Relax(u, v, w)
7. Exit

Relax

Procedure RElax(u, v, w)

1. If d[v] > d[u] + w[u, v]
   then d[v] = d[u] + w[u, v], π[v] = u
2. Return



$U = A$

$S = \{ B, C \}$

V	d[v]	$\pi(v)$
A	0	—
B	~~10~~ 8	A
C	5	A
D	~~14~~ 9	C
E	7	C

Relax (A, B, 10)

$\infty > 0 + 10$

Relax (A, C, 5)

$\infty > 0 + 5$

$U = C$

$S = \{ D, B, E \}$

Relax (C, D, 9)

$\infty > 5 + 9$

Relax (C, B, 3)

$10 > 5 + 3$

Relax (C, E, 2)

$\infty > 5 + 2 = 7$

$U = E$

$S = \{ D \}$

Relax (E, D, 6)

$14 > 7 + 6 = 13$

$U = B$

$S = \{ D \}$

Relax (B, D, 1)

$13 > 8 + 1 = 9$

$U = D$