```
int keyToSearch = 60;
if (search(root, keyToSearch)) {
    System.out.println(keyToSearch + "found in the
    tree.");
} else {
    System.out.println(keyToSearch + "not found
    in the tree");
}
}
static class Node{
    int key;
    Node left, right;
    Node(int item){
        key = item;
        left = right = null;
    }
}
static Node insert(Node root, int key){
    if(root == null){
        return new Node(key);
    }
    if (key < root.key){
        root.left = insert(root.left, key);
    }else if (key > root.key){
        root.right = insert(root.right, key);
    }
    return root;
}
```

```
static boolean search(Node root, int key){
    if(root == null){
        return false;
    }
    if(root.key == key){
        return true;
    }
    if(key < root.key){
        return search(root.left, key);
    }else{
        return search(root.right, key);
    }
}
```

* Why we should use trees when compared with other d.s ?

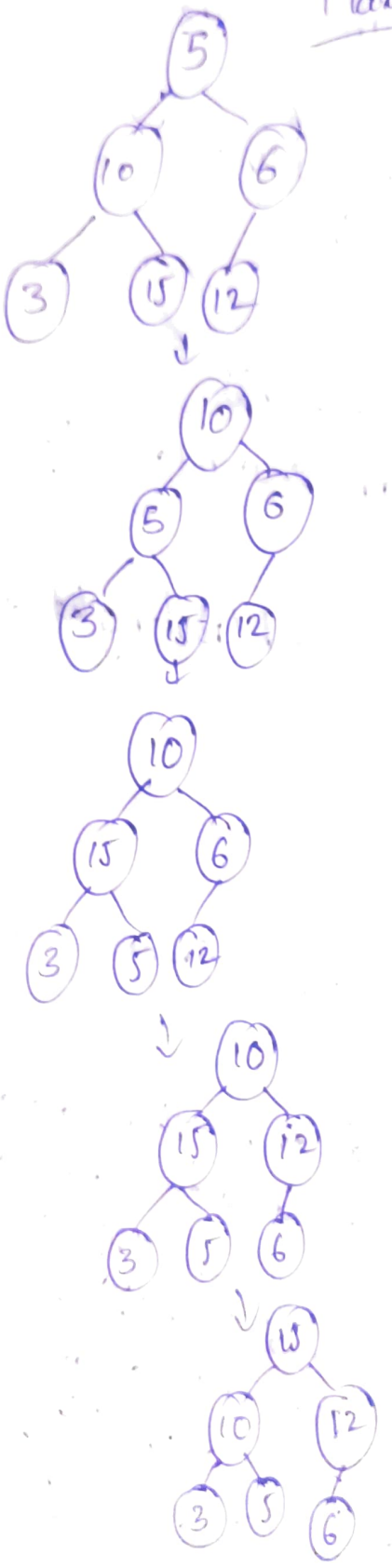
A: Trees are preferred over other data Structures like arrays or linked lists when dealing with hierarchial data or when Efficient Searching and Sorting are needed. Their hierarchial Structure allows for Easy organization and representation of relationships.

-) Faster Search, Insert, Delete (Espacially BST)
   Dynamic data handling

26/08/25

5, 10, 6, 3, 15, 12

Max heap



Priority Queue

* Priority queue implements a priority heap based queue that process the Elements based on the priority rather than FIFO.
* The Elements of the priority queue are ordered according to the natural ordering and Elements must implement the comparable function.
* The Size of the priority queue is dynamic that means it will decrease or increase as per the insertion or deletion of an Element.
* Priority Queues are unbounded queues
* The head of the priority queue is the least Element w.r.to the Specified ordering
* The priority queue retrieval operations are ~~poke~~ poll, remove, peek.
* Priority Queue provides O(logn) for adding & deletion of the Elements

Syntax: PriorityQueue<E>
pq = new PriorityQueue<Comparator>

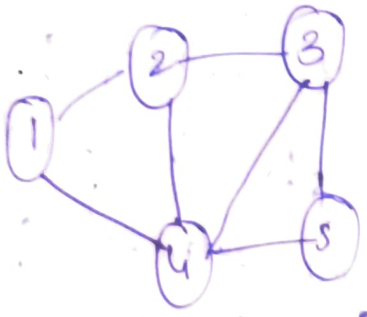
* This Constructor creates the priority queue with the default initial capacity and whose

Elements are ordered according to the Specified Comparator

2.) What is the pq default initial capacity if the d.c is reached & new Elements are added So ~~what~~ how much Capacity will it be increased.

* Sparse Dense

Dense: all nodes - edges



Adj mat

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

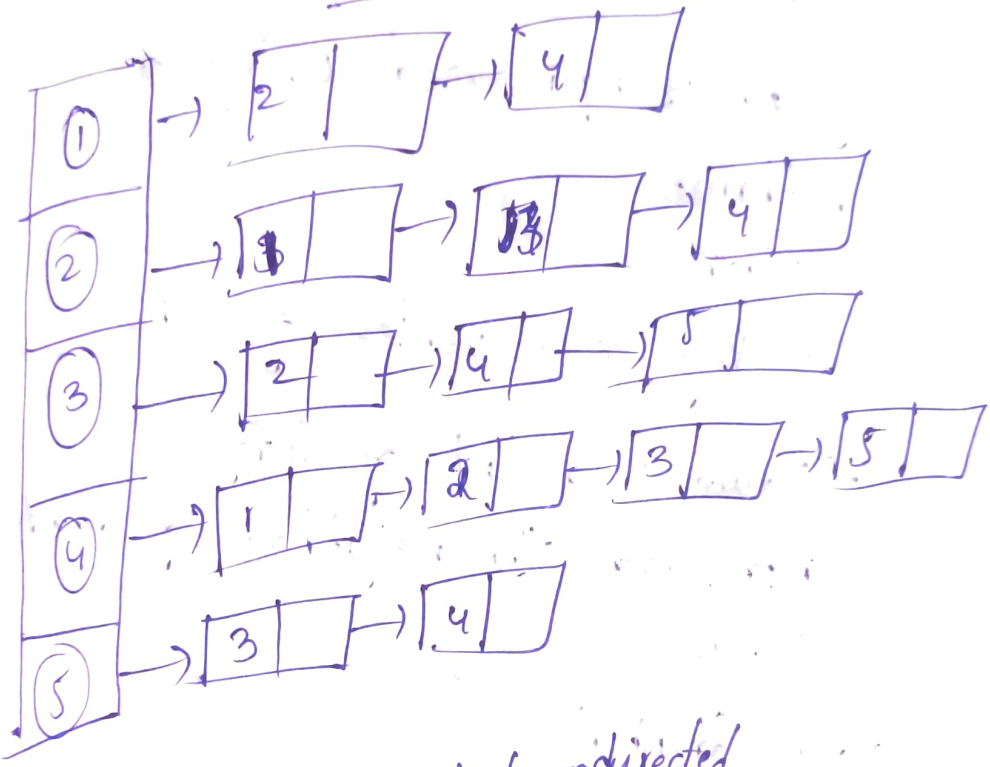
note:

More no of edges - graph (mat)

less no of edges - list

{1:[2,4], 2:[1, 3, 4], 3:[2,4,5],

Array List



8 diff graphs (Directed, undirected
↓ ↓
w: 1→2 el w: 1→2 el
w1: 1→2 el

↓
2 representation
↓
mat / list

step (1):

Graph class

graph is represented using a hashmap
where:
-> key is a vertex (eg. 0, 1, 2 ...),
-> Value is a list of integers representing the neighbors of that Vertex

```
class Graph {
    private Map<Integer, List<Integer>> adjacencyList;
```

Constructor

```
adjacencyList = new HashMap<>();
```

-> initialises the graph by creating an Empty hashmap

Adding Vertex

```
addVertex(int Vertex)
```

-> add a new vertex
-> Creates an Empty list for that Vertex in the map to hold its neighbors

Adding edge

-) addEdge (int source, int destination)

-) adds a directed edge from Source to dest

Remove Vertex

void removeVertex (int vertex) {

→ removes a vertex from the graph Entirely

-) Also removes it from all the neighbor lists where it may appear.

Remove edge

-> removes the edge from S to d

Get neighboy

-) Returns a list of neighboys (connected Veitices) of the given vertex

Print the Graph:

0 -> 1 2

1 -> 2

2 ->

main class & Execution:

Creates and use graph

→ Adds 3 vertices : 0, 1, 2

→ Adds directed edges

0 → 1

0 → 2

1 → 2

→ Prints the current Structure

→ Removes the edge 0 → 1 and prints the updated graph

→ Removes Vertex 2 from the graph (and any connections to it) and prints the final result.

O/p:

Graph:

0 → 1 2

1 → 2

2 →

After removing edge (0, 1):

0 → 2

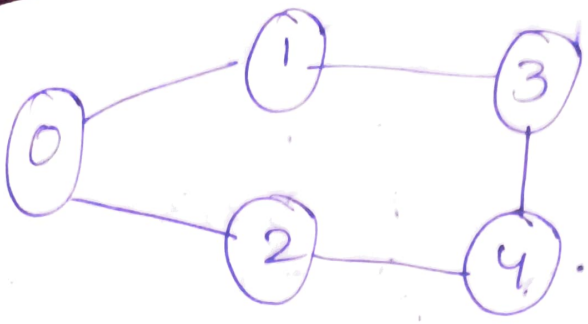
1 → 2

2 →

After removing vertex 2:

0 →

1 →



Bfs :-

0 → 1 → 2 → 3 → 4

Dfs :-

0 → 1 → 3 → 4 → 2

---

2) In Java, PQ

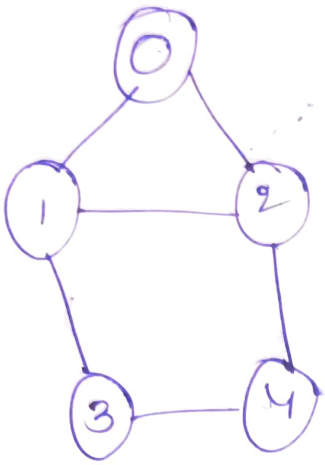
Default initial Capacity

⇒ 11

If we add more than initial capacity

for Ex :- 15

The capacity will be 15



0 → 1, 2
1 → 0, 2, 3
2 → 0, 4
3 → 1, 4
4 → 2, 3

0 → 1 → 3 → 4

→ ArrayList<ArrayList<Integer>>

↑ Nested list creation

→ adj.add(new ArrayList<>(Arrays.asList(1,2)));

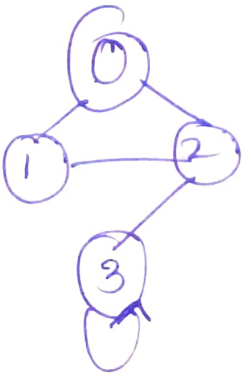
→ ArrayList<Integer> ans = bfs(adj);

→ int V = adj.size();

Dfs: 0 → 1 → 2 → 4 → 3

DFS:-

0 → 1, 2
1 → 2
2 → 0, 3
3 → 3



Stack, visited[] array

Step 1: S | V

[0] [T, F, F, F] → Push start node 0

[1] [T, F, F, F] → Pop 0, visit it

[2, 1] [T, F, F, F] → Push neighbors of 0 (2, 1)

[1] [T, T, F, F] → Pop 1, visit it

While loop –

int n: Total nodes = 4
adj sirf k

boolean – visited nodes ki track
chestharu

Array list → DFS result ni store
chestharu

stack (int type) → DFS traversal ki stack.
st.push(0); → Start DFS from node 0

Loop 1: Stack: [0]

pop [0]

visited: f → t

res: [0]

Neighbors of 0 (reverse order): Push 2 then 1

Stack: [2, 1]

Loop 2:

Stack: [2, 1]

pop: [1]

v = f → t

res: [0, 1]

neighbors of 1: push 2

Stack: [2, 2]

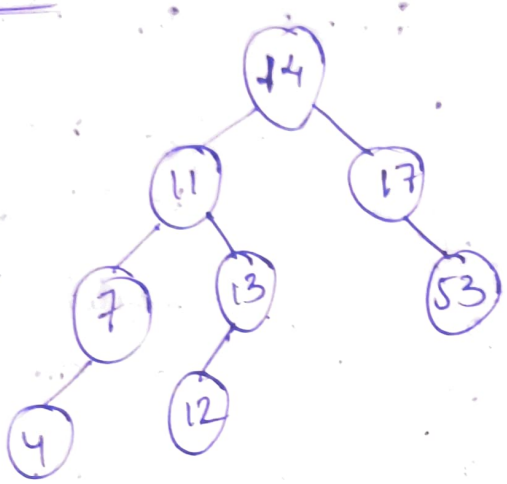
We use backtracking here, to get the unvisited nodes

AVL Tree

14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20.

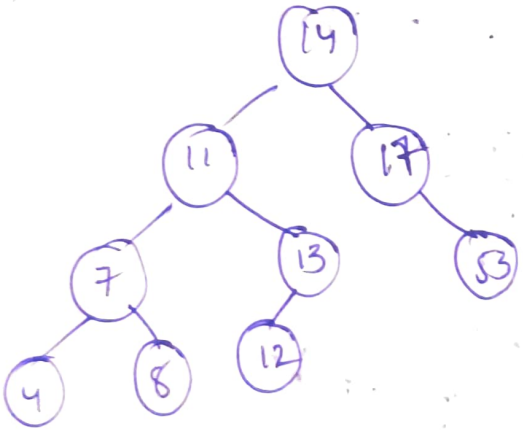
~~R Rotation~~

=)

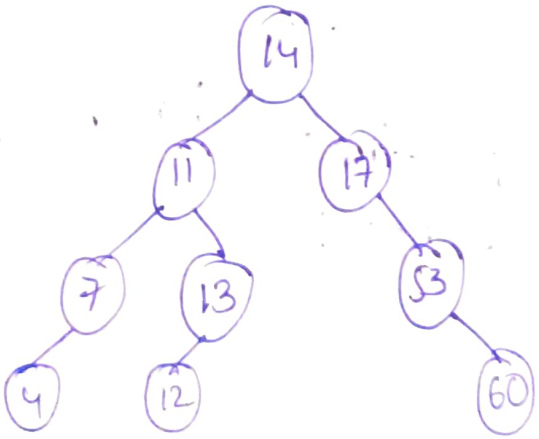


8 will be right of 7

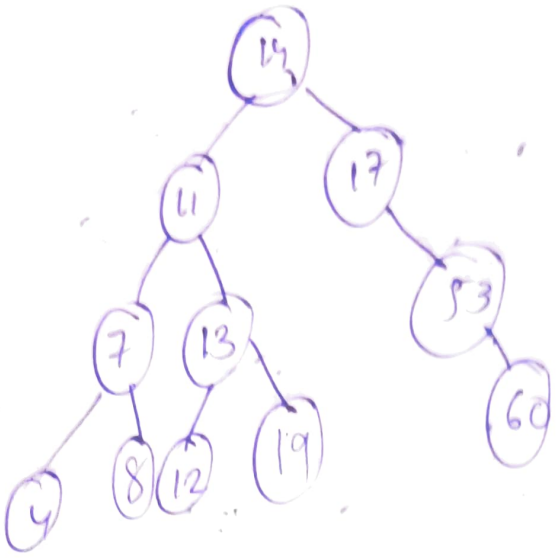
=)



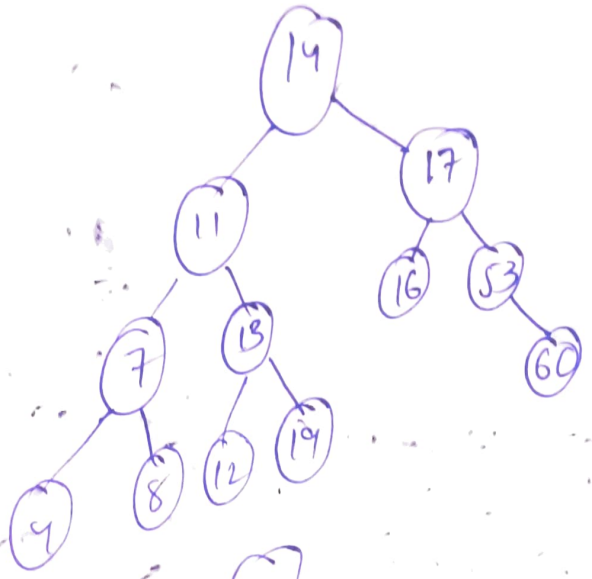
60 =)



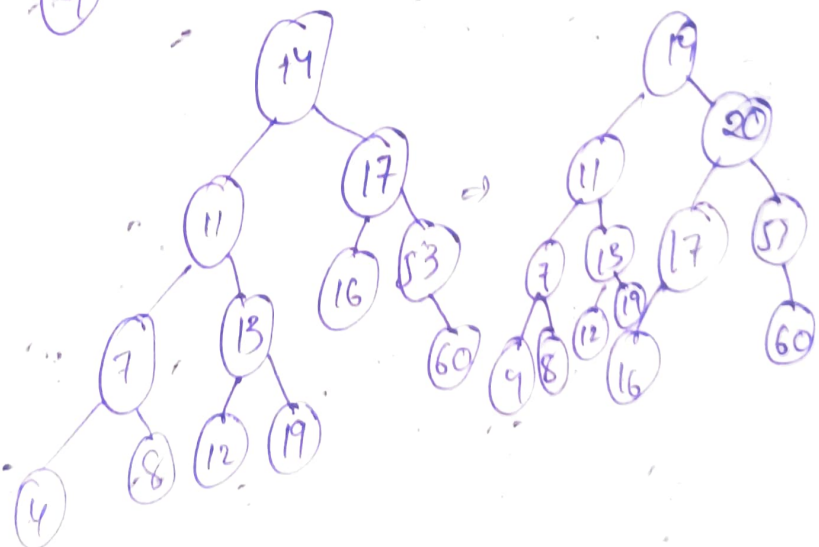
19 :)

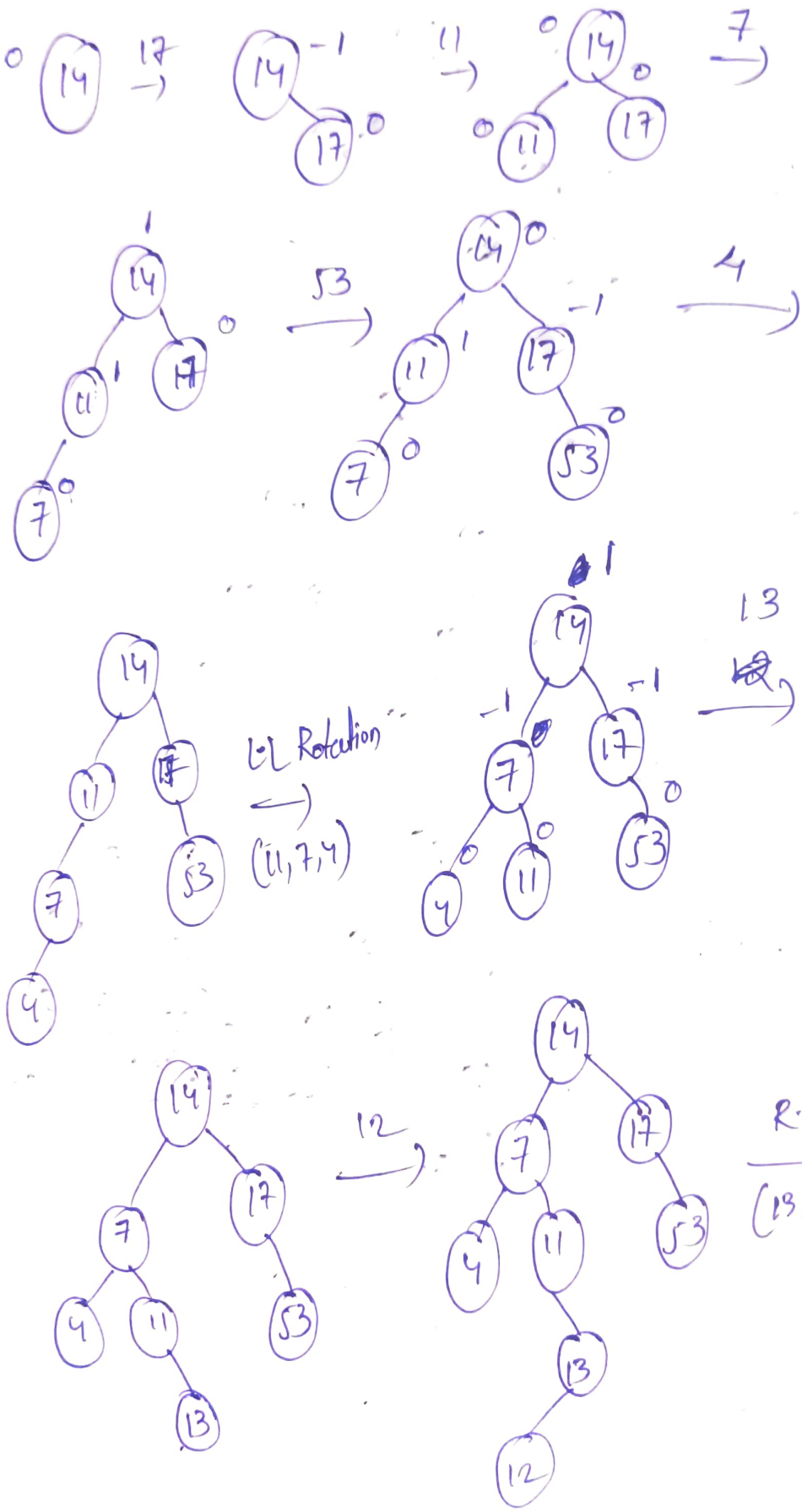

14
11
17
7
13
53
4
8
12
19
60

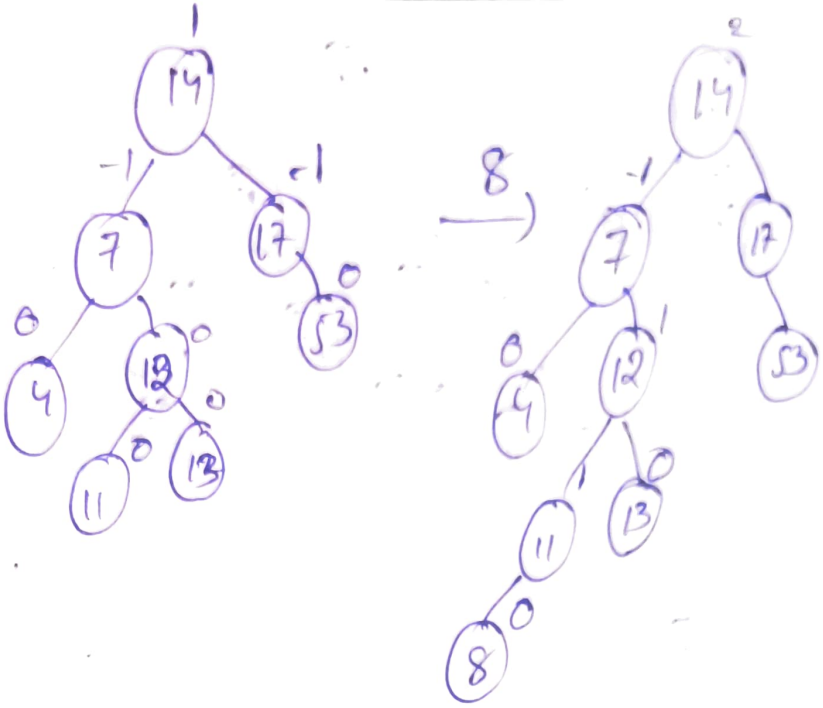
16 :)

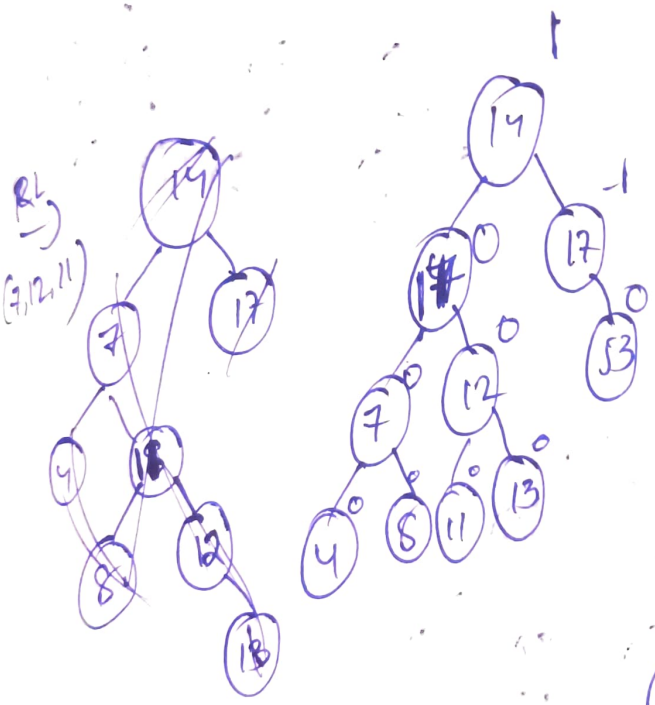

14
11
17
7
13
16
53
4
8
12
19
60

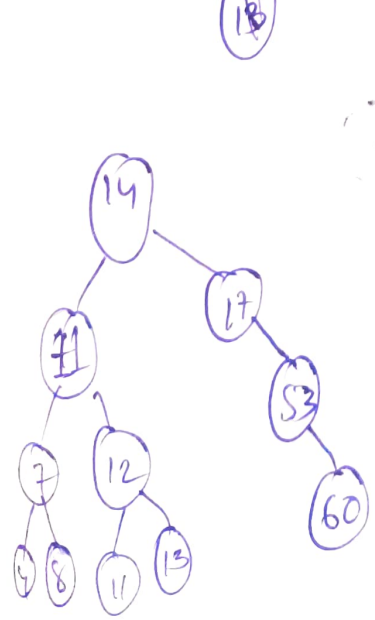
20 :)

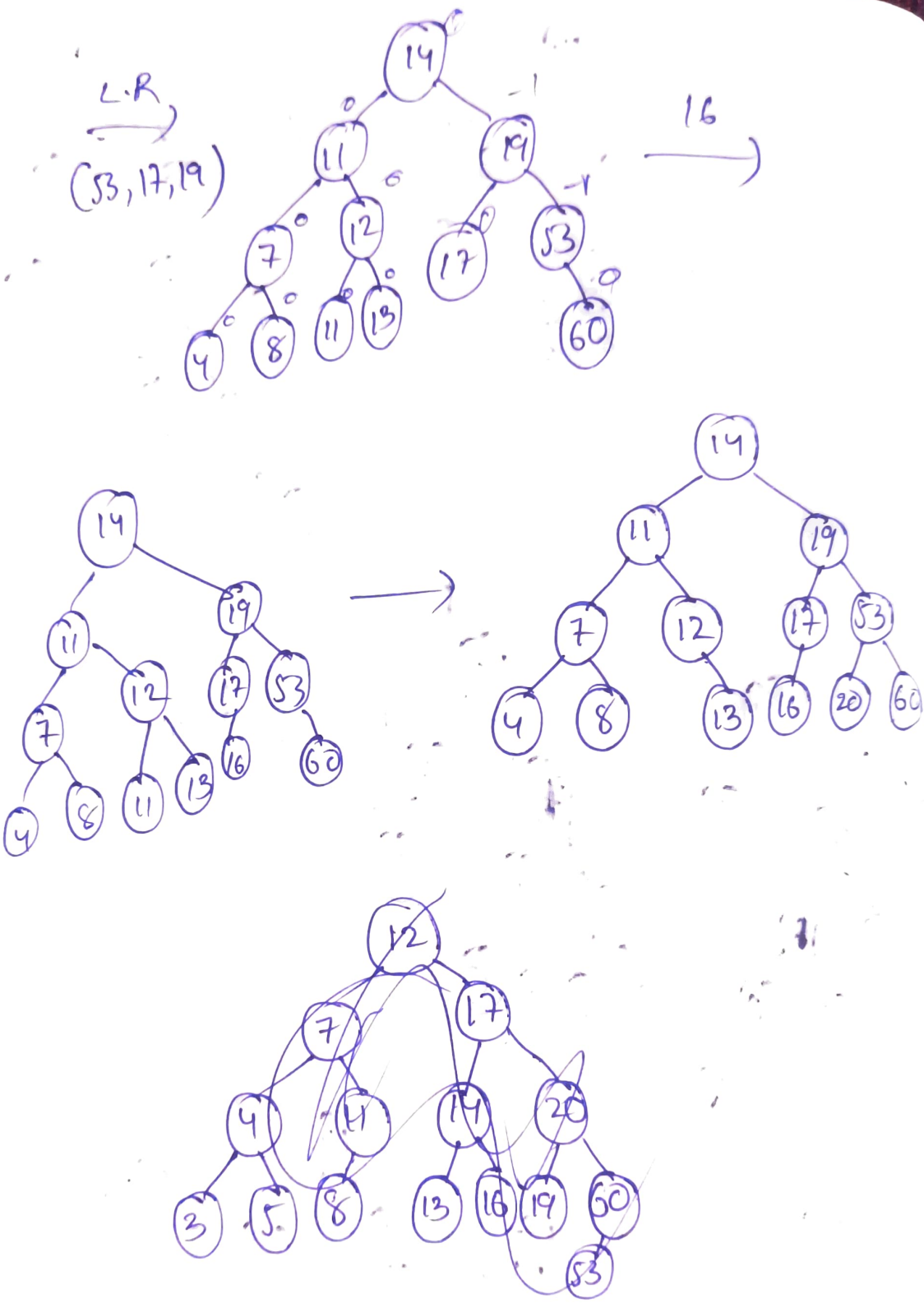

14
11
17
7
13
16
53
4
8
12
19
60
19
11
20
7
13
17
53
4
8
12
19
16
60


14
17
-1
11
0
7
1
53
4
LL Rotation
(11,7,4)
13
12
R-L
(13,14,11)


14
7
17
4
12
53
11
13
8


RL
(7,12,11)
60


RR
(14,53,60)
19.


L.R
(53, 17, 19)
14
11
19
7
12
17
53
4
8
11
13
60
16
14
11
19
7
12
17
53
4
8
11
13
16
60
14
11
19
7
12
17
53
4
8
13
16
20
60
12
7
17
4
11
14
20
3
5
8
13
16
19
60
53

1.) Directed unweighted graph

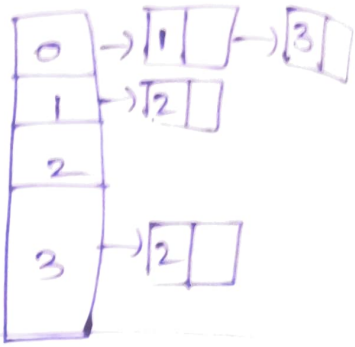
Vertices → 0, 1, 2, 3

Edges : 0→1, 0→3, 1→2, 3→2

Adj mat

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

~~Adj~~ Array List



2.) Undirected Unweighted graph

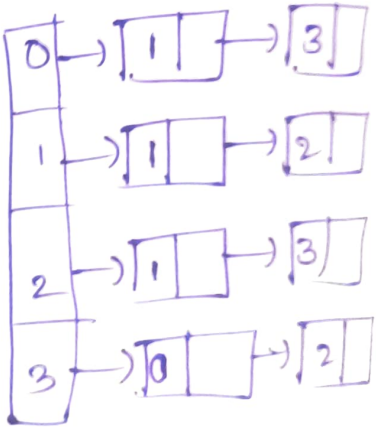
Vertices : 0, 1, 2, 3

Edges : 0-1, 0-3, 1-2, 2-3

Adj mat

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

Array List



3.) Directed Weighted Graph

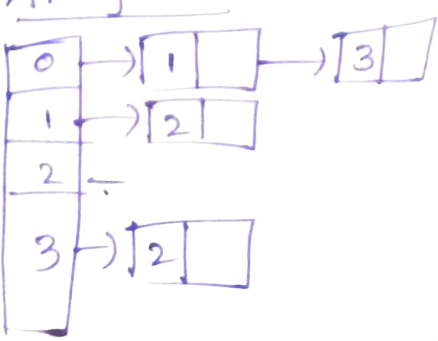
Vertices : 0, 1, 2, 3

Edges : 0→1 (7), 0→3 (2), 1→2 (3), 3→2 (1)

Adj (mat)

	0	1	2	3
0	0	7	0	2
1	0	0	3	0
2	0	0	0	0
3	0	0	1	0

Array List



4) Undirected weighted graph

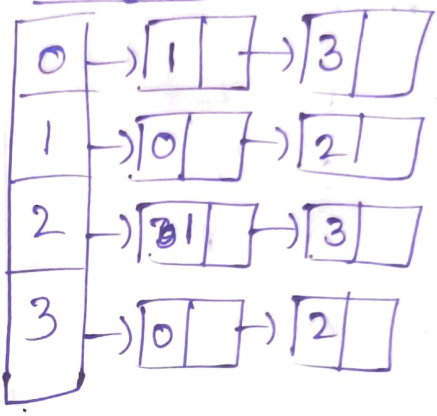
V: 0, 1, 2, 3

e: 0-1(7), 0-3(2), 1-2(3), 2-3(4)

Adj mat

	0	1	2	3
0	0	7	0	2
1	7	0	3	0
2	0	3	0	4
3	2	0	4	0

Array List



5) Directed unweighted graph (with isolated node)

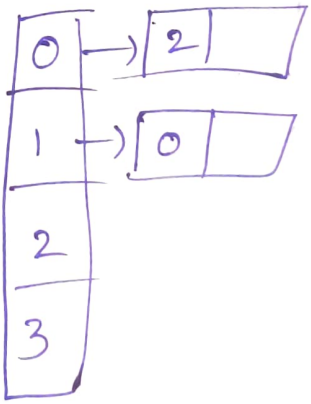
V: 0, 1, 2, 3

e = 0→2, 1→0

Adj mat

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

Array List



6) Undirected unweighted graph (with isolated node)

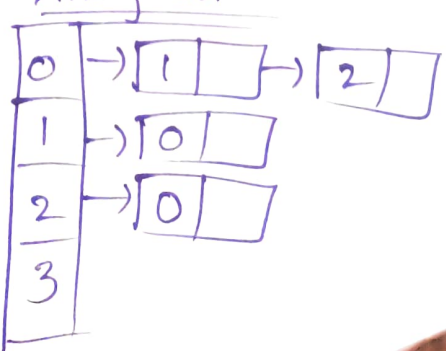
V: 0, 1, 2, 3

e = 0-2, 1-0

Adj mat

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

Array List



7.) Directed weighted graph (with Self loop)

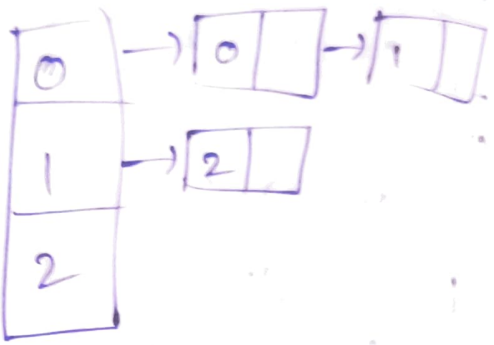
v: 0, 1, 2

e: 0→0(4), 0→1(5), 1→2(1)

Adj mat.

	0	1	2
0	4	5	0
1	0	0	1
2	0	0	0

Array List



8.) Undirected weighted Complete graph (3 nodes)

v: 0, 1, 2

e: 0-1(10), 1-2(20), 0-2(30)

A.M

	0	1	2
0	0	10	30
1	10	0	20
2	30	20	0

A.L: