Sunbeam Institute of Information Technology
Pune and Karad

Module – Data Structures and Algorithms

Trainer - Devendra Dhande

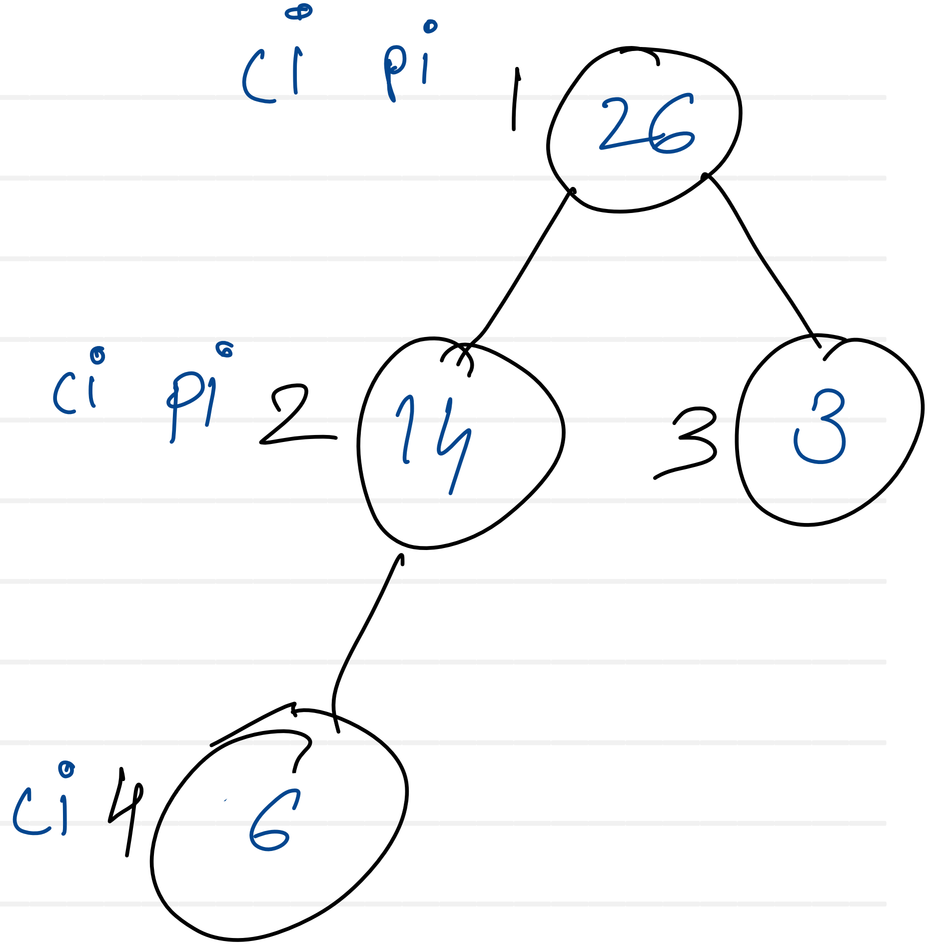
Email – devendra.dhande@sunbeaminfo.com

Heap - Add

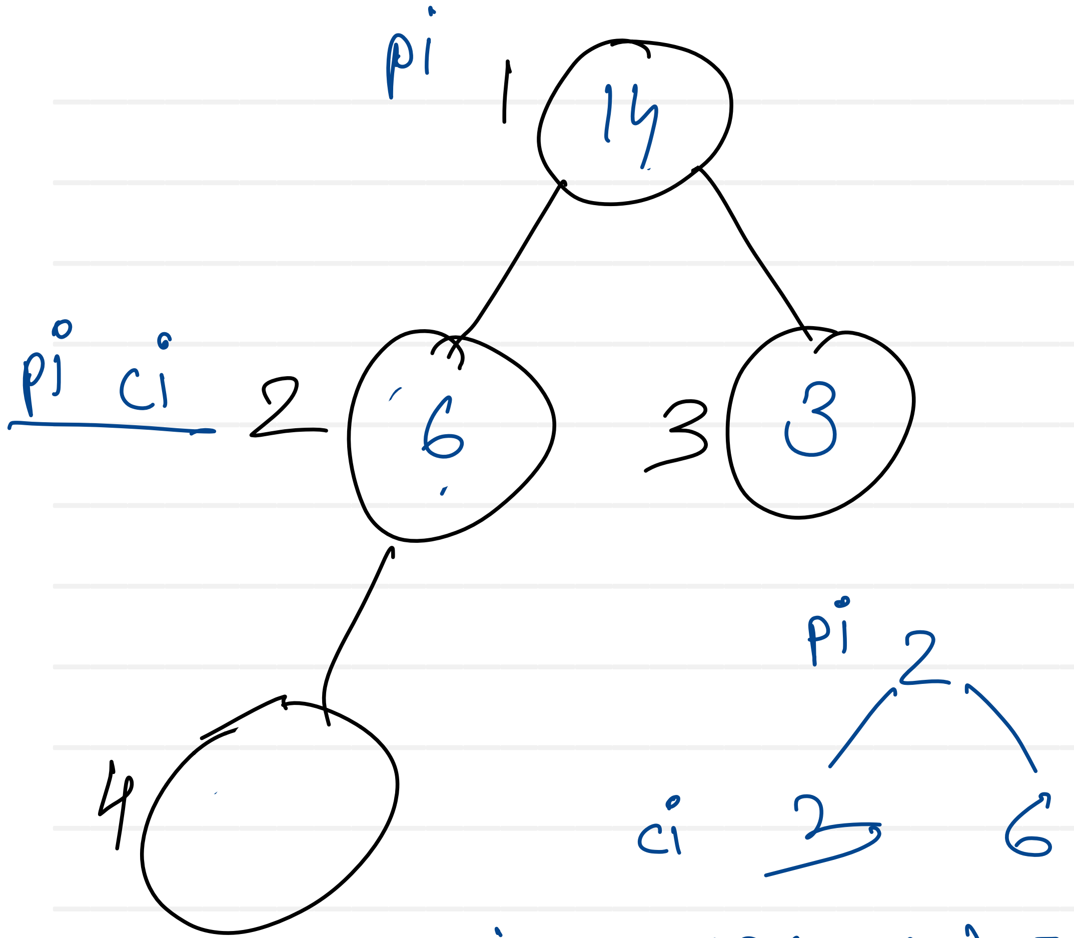
pi=0

```
size=0
add(value) {
    size++;
    arr[size]=value;
    ci=size;
    pi=ci/2;
    while(pi >= 1) {
        if(arr[pi] > arr[ci])
            break;
        int temp = arr[pi];
        arr[pi] = arr[ci];
        arr[ci] = temp;

        ci = pi;
        pi = ci/2;
    }
}
```



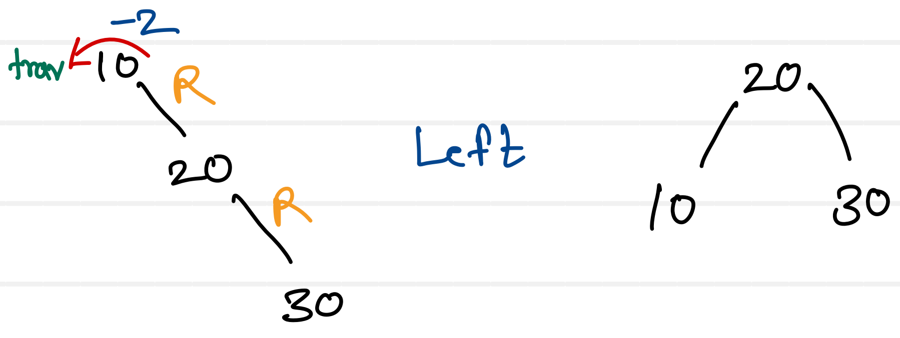
Heap - Delete



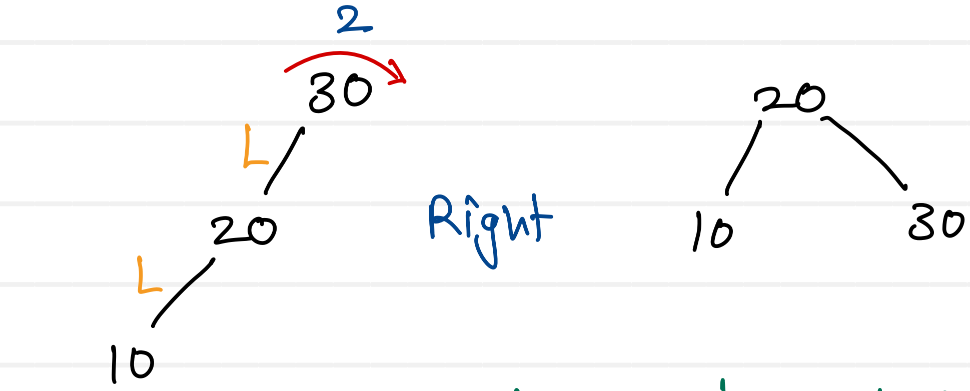
ci=2 if(arr[ci+1] > arr[ci])
ci = ci+1

```
delete( ) {
    arr[1] = arr[size];
    size--;
    pi = 1
    ci = pi * 2;
    while( ci <= size) {
        if((ci+1) <= size && arr[ci+1] > arr[ci])
            ci = ci+1;
        if(arr[pi] > arr[ci])
            break;
        int temp = arr[pi]
        arr[pi] = arr[ci]
        arr[ci] = temp
        pi = ci;
        ci = pi * 2;
    }
}
```

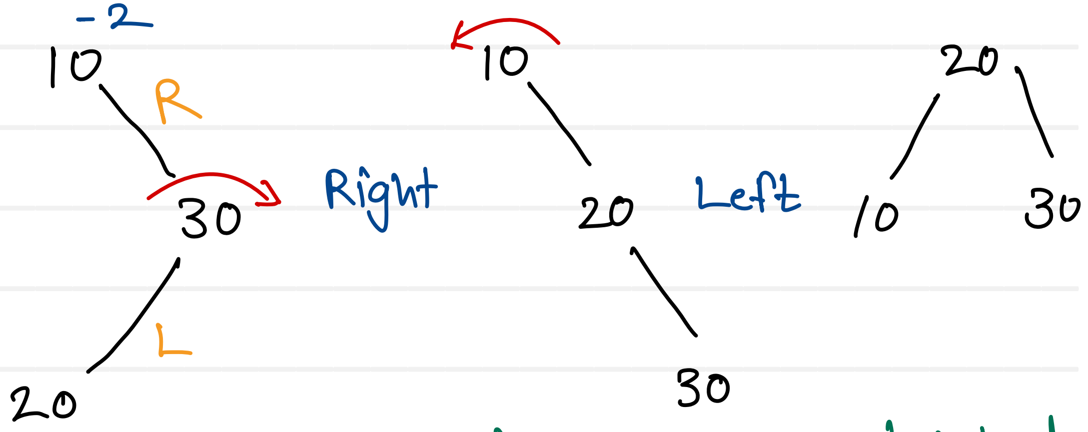
Rotation cases



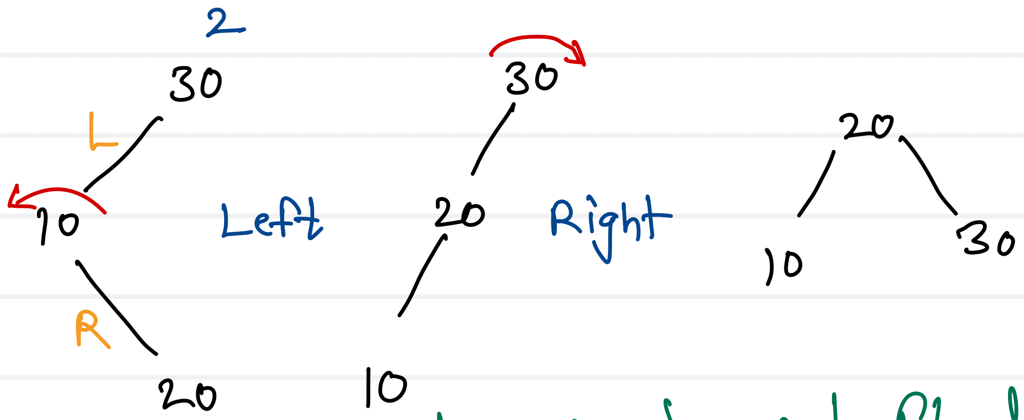
bf < -1 && value > trav.right.data



bf > 1 && value < trav.left.data

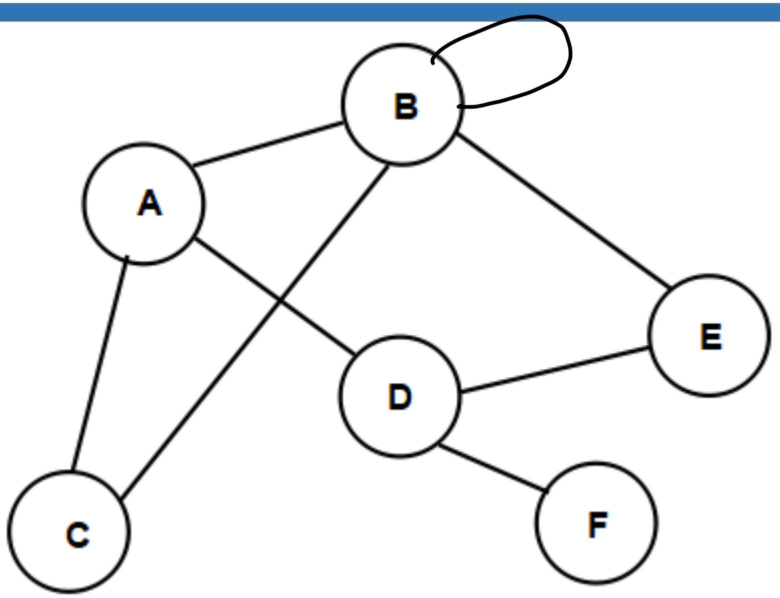


bf < -1 && value < trav.right.data



bf > 1 && value > trav.left.data

Graph : Terminologies



- **Graph** is a non linear data structure having set of vertices (nodes) and set of edges (arcs).
  - G = {V, E}

    Where V is a set of vertices and E is a set of edges
  - **Vertex (node)** is an element in the graph

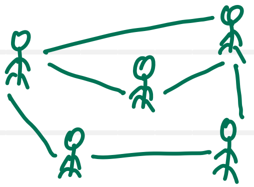
    V = {A, B, C, D, E, F}
  - **Edge (arc)** is a line connecting two vertices

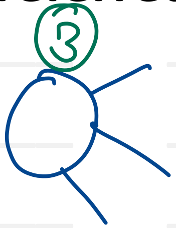
    E = {(A,B), (A,C), (B,C), (B,E), (D, E), (D,F),(A,D)}
- Vertex A is set be **adjacent** to B, if and only if there is an edge from A to B.
- **Degree of vertex :-** Number of vertices adjacent to given vertex
- **Path :-** Set of edges connecting any two vertices is called as path between those two vertices.
  - Path between A to D = {(A, B), (B, E), (E, D)}
- **Cycle :-** Set of edges connecting to a node itself is called as cycle.
  - {(A, B), (B, E), (E, D), (D, A)}
- **Loop :-** An edge connecting a node to itself is called as loop. Loop is smallest cycle.

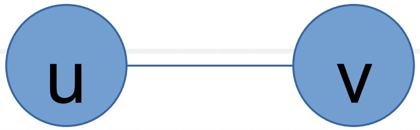
Graph : Types

- **Undirected graph.**
  - If we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**
  - **graph which contains undirected edges referred as undirected graph.**

degree : no. of adjacent vertices





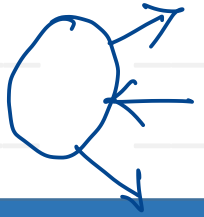


(u, v) == (v, u)

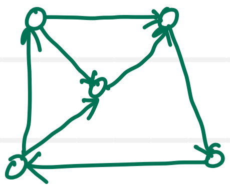
- **Directed Graph (Di-graph)**
  - If we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **ordered pair of vertices** i.e. directed edge.
  - **graph which contains set of directed edges referred as directed graph (di-graph).**
  - graph in which each edge has some direction

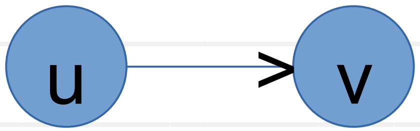
in degree : no. of in coming edges 1

out degree : no. of out going edges 2



maps

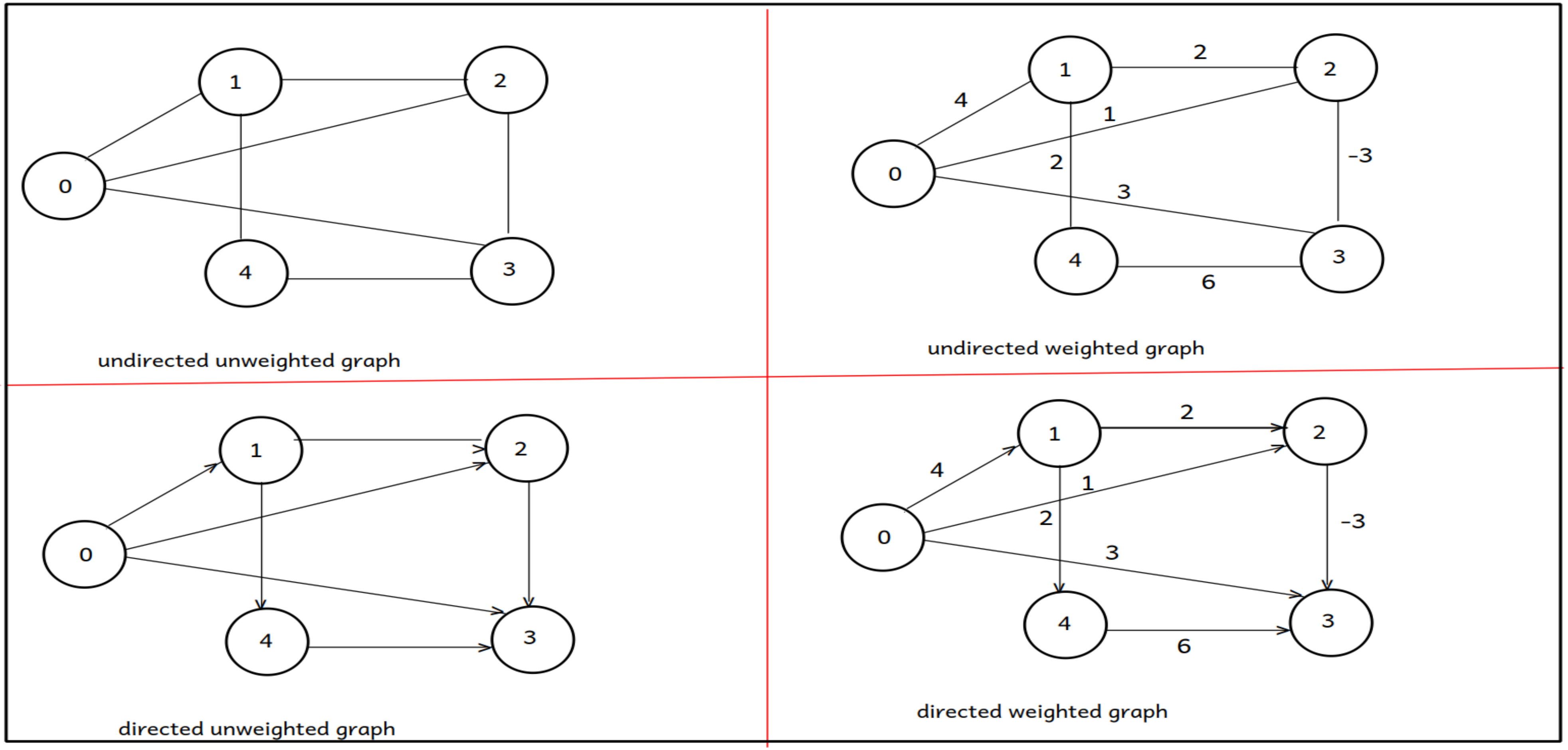




(u, v) != (v, u)

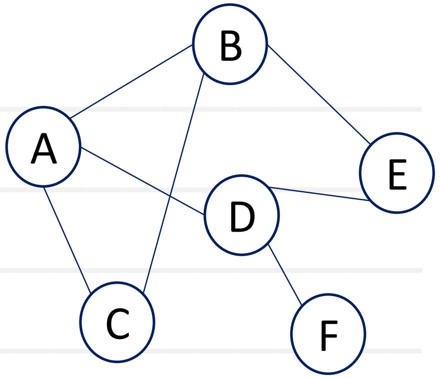
Graph : Types

- **Weighted Graph**
  - A graph in which edge is associated with a number (ie weight)



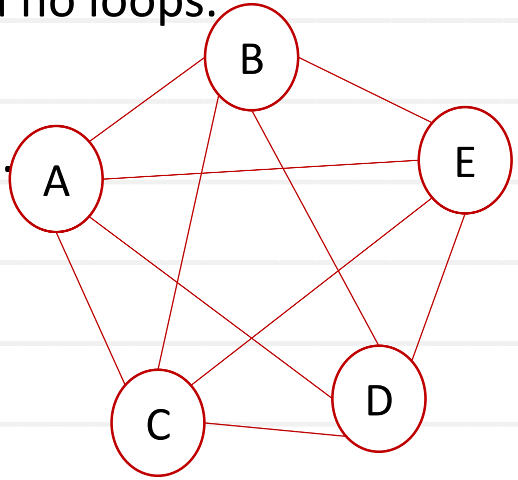
Graph : Types

- **Simple Graph**
  - Graph not having multiple edges between adjacent nodes and no loops.

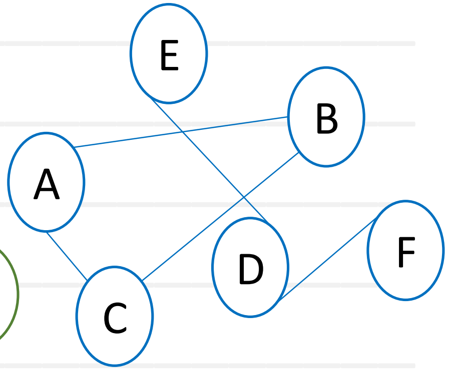


- **Complete Graph**
  - Simple graph in which node is adjacent with every other node.
  - Un-Directed graph: Number of Edges = n (n -1) / 2

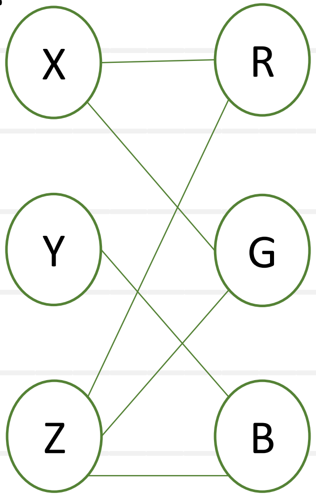
    where, n – number of vertices
  - Directed graph: Number of edges = n (n-1)



- **Connected Graph**
  - Simple graph in which there is some path exist between any two vertices.
  - Can traverse the entire graph starting from any vertex.

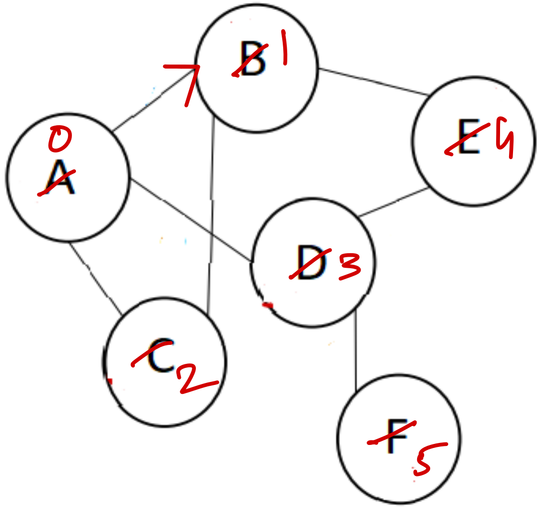


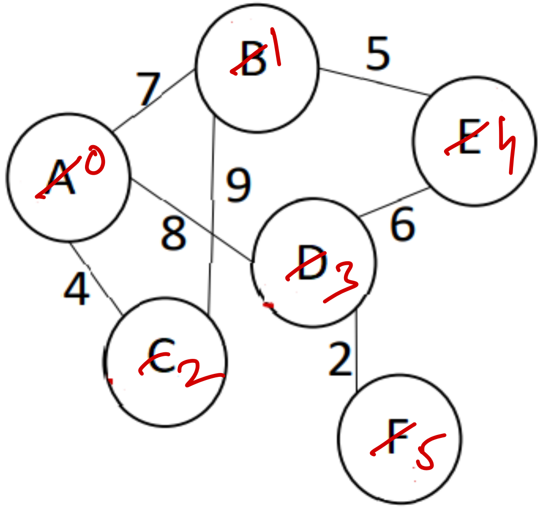
- **Bi-partite graph**
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.



Graph Implementation – Adjacency Matrix

- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For non-weighted graph, 1 indicate edge and 0 indicate no edge.
- For weighted graph, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is O(V^2).



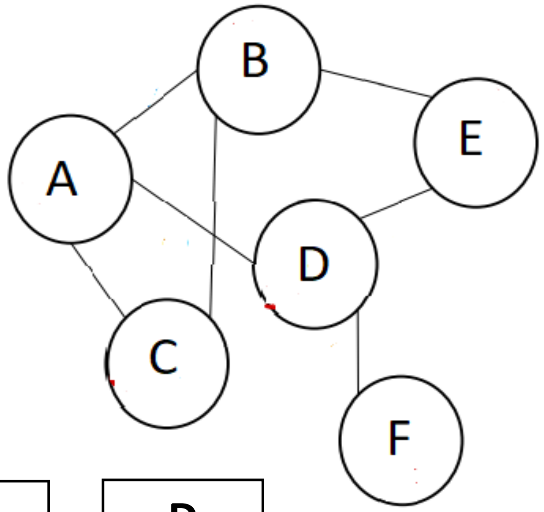


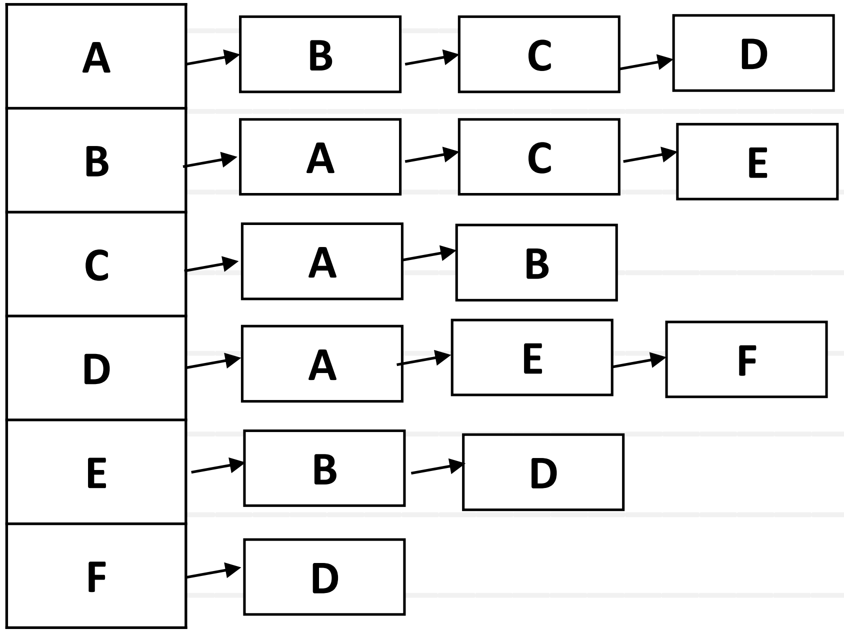
	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

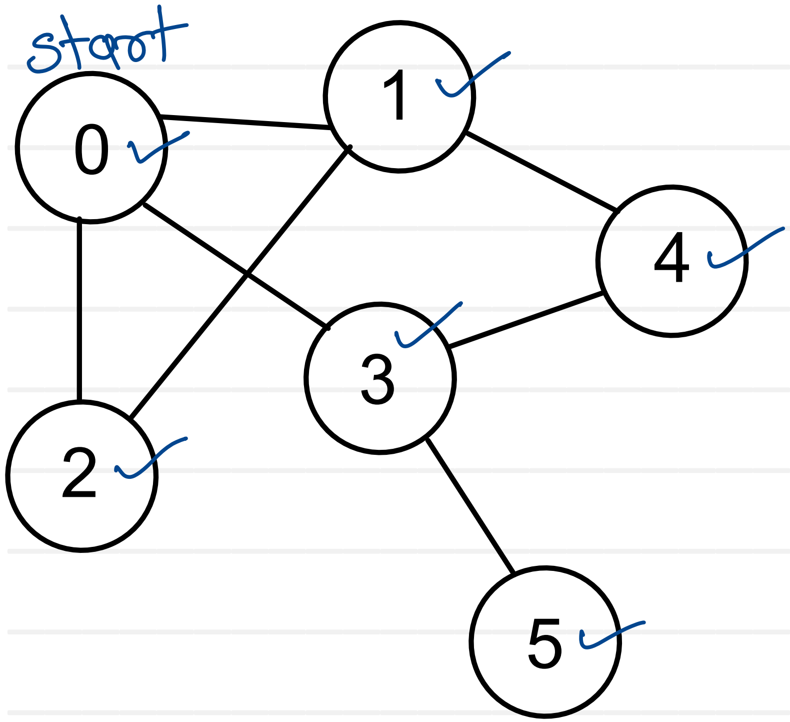
Graph Implementation – Adjacency List

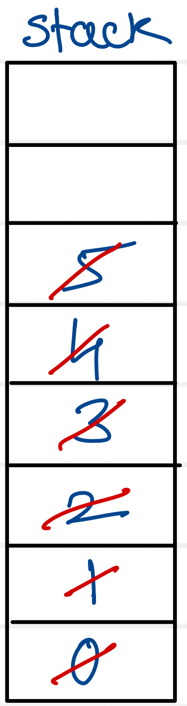
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbor vertices are stored.
- For weighted graph, neighbor vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).



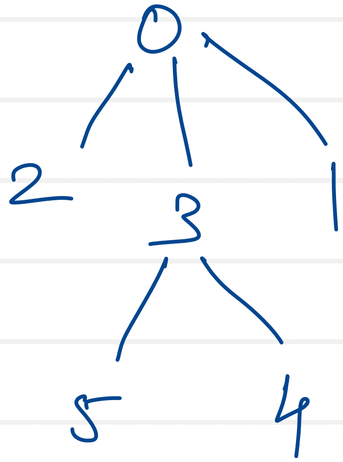


DFS Traversal



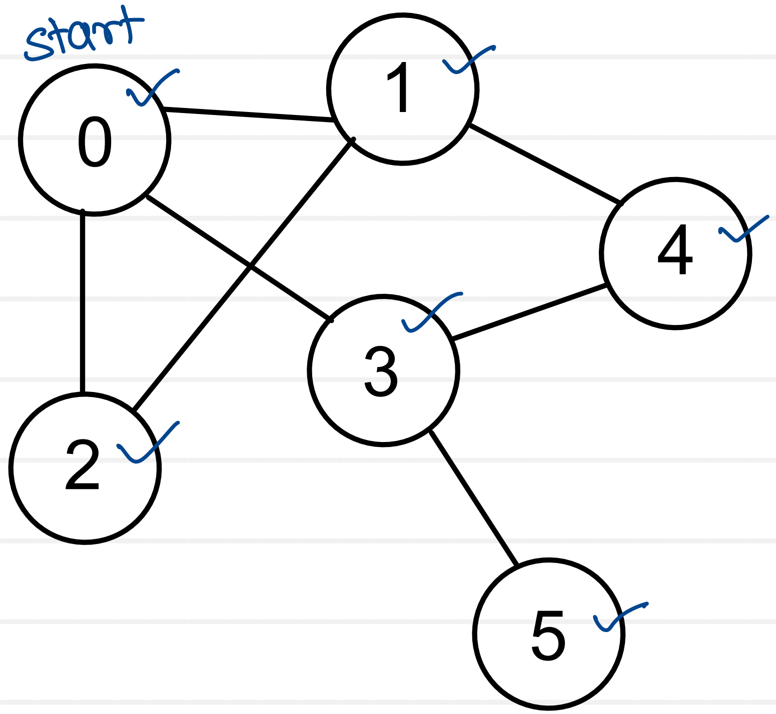


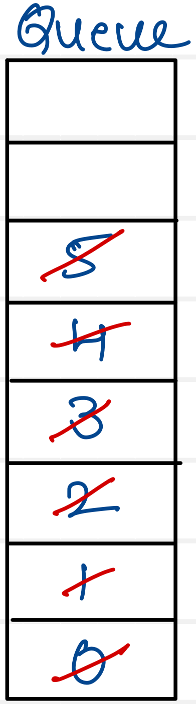
1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.



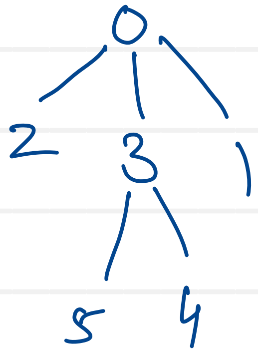
Traversal : 0, 3, 5, 4, 2, 1

BFS Traversal





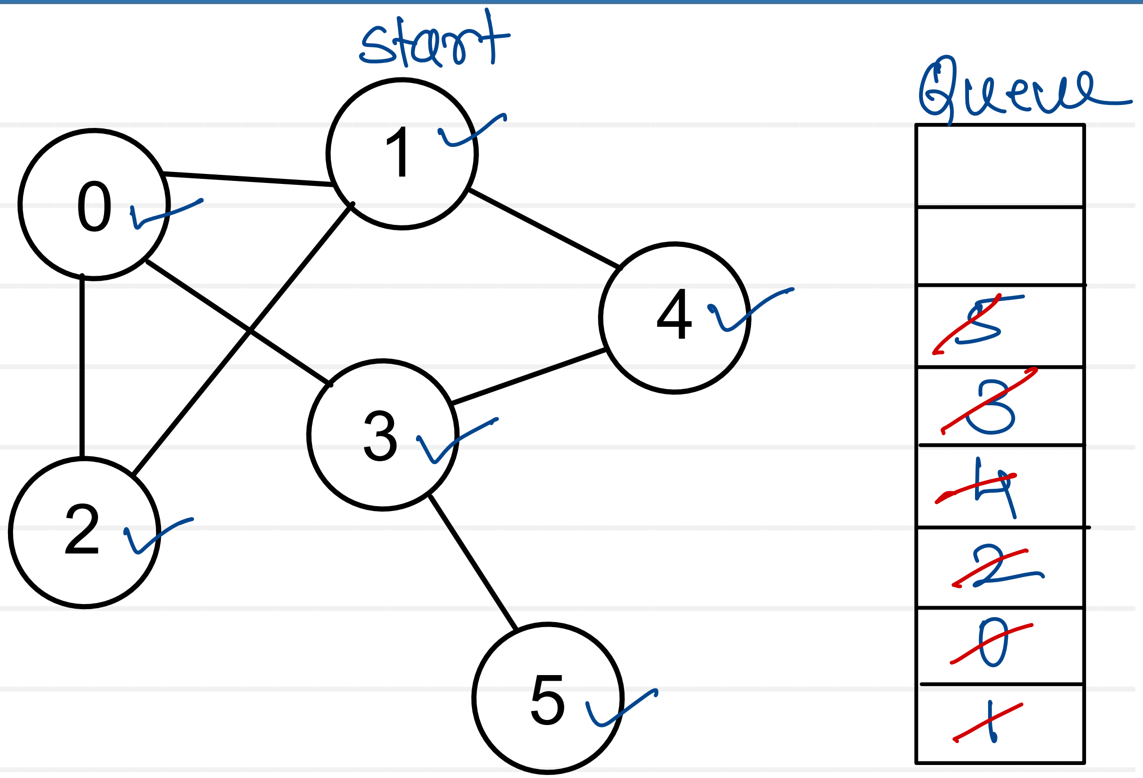
1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it
3. Pop vertex from queue.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.



Traversal : 0, 1, 2, 3, 4, 5

Single Source Path Length

1. Create path length array to keep length of vertex from start vertex.
2. push start on queue & mark it. length[start] = 0
3. pop the vertex.
4. push all its non-marked neighbors on the queue, mark them.
5. For each such vertex calculate length as length[neighbor] = length[current] + 1
6. print current vertex to that neighbor vertex edge.
7. repeat steps 3-6 until queue is empty.
8. Print path length array.

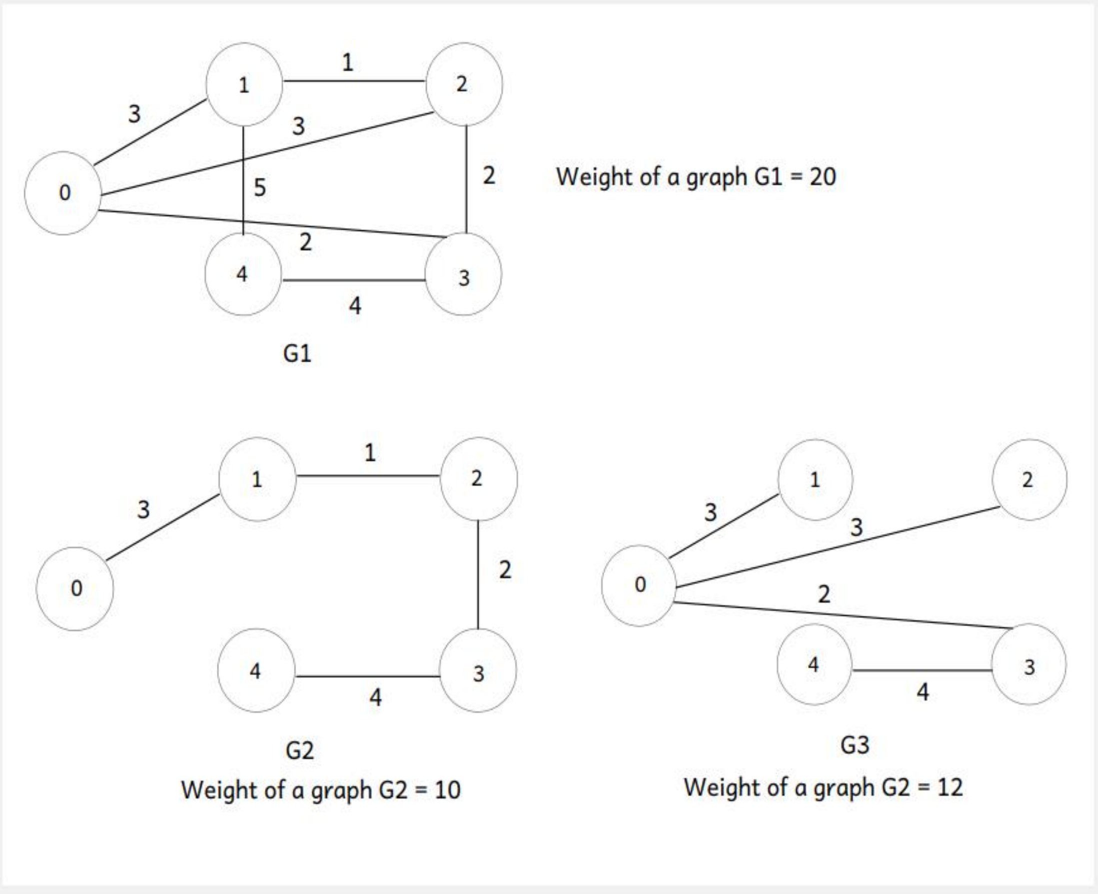


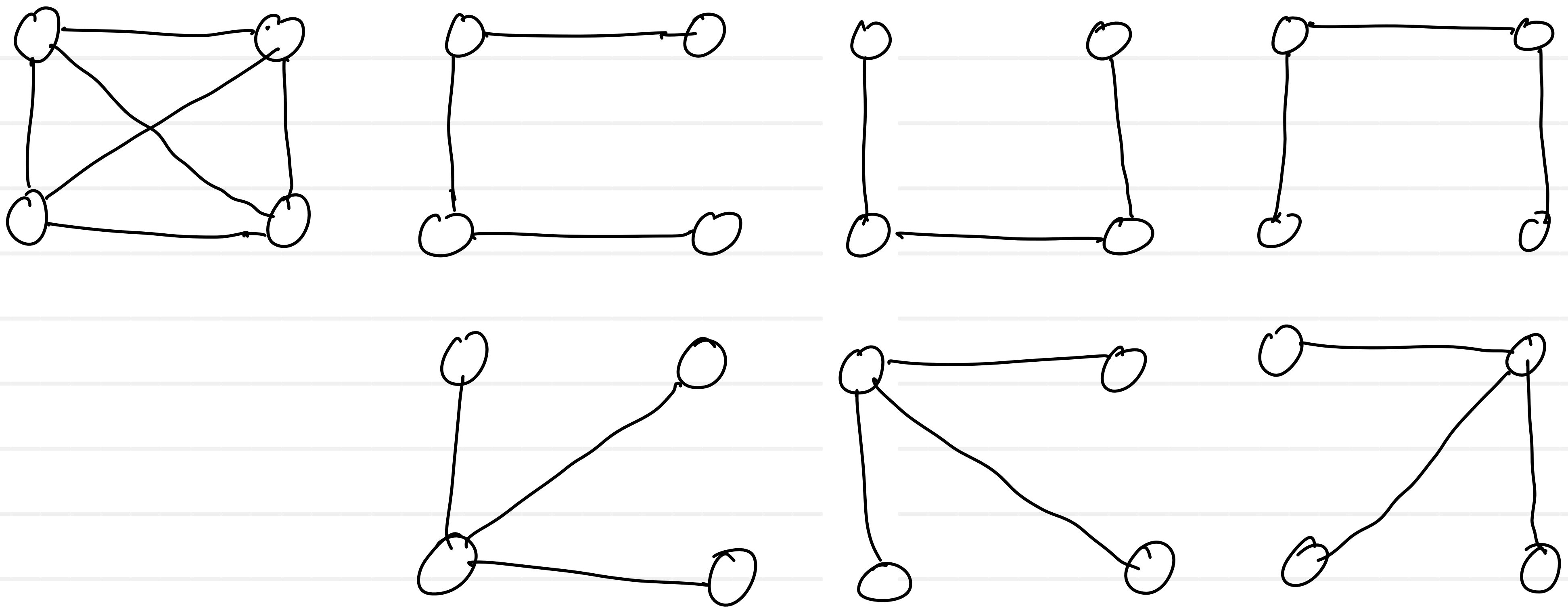
Length	0	1	2	3	4	5
	1	0	1	2	1	3

Path length tree : (1-0) (1-2) (1-4) (0-3) (3-5)

Spanning Tree

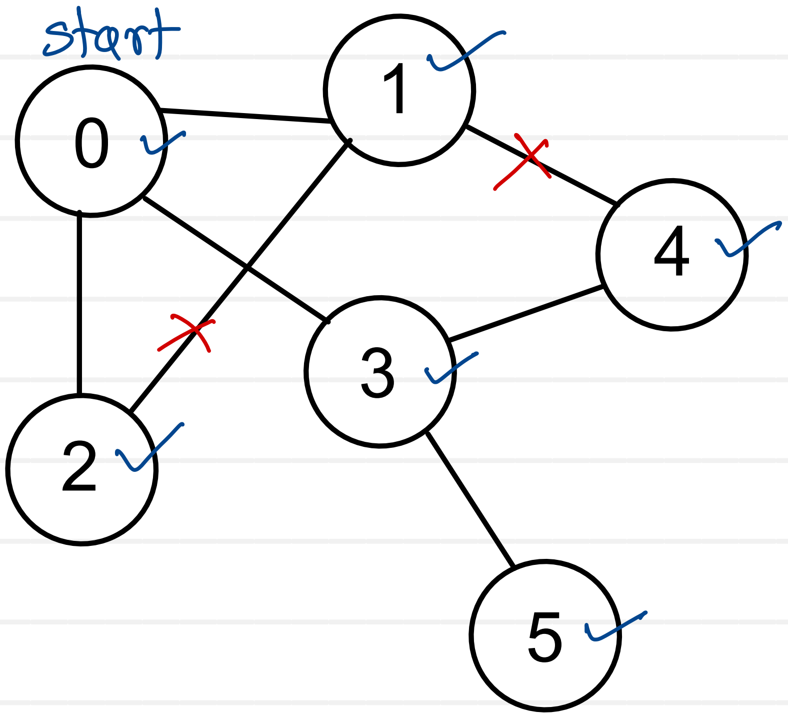
- Tree is a graph without cycles. Includes all V vertices and V-1 edges.
- Spanning tree is connected sub-graph of the given graph that contains all the vertices and sub-set of edges.
- Spanning tree can be created by removing few edges from the graph which are causing cycles to form.
- One graph can have multiple different spanning trees.
- In weighted graph, spanning tree can be made who has minimum weight (sum of weights of edges). Such spanning tree is called as Minimum Spanning Tree.
- Spanning tree can be made by various algorithms.
  - BFS Spanning tree
  - DFS Spanning tree
  - Prim's MST
  - Kruskal's MST

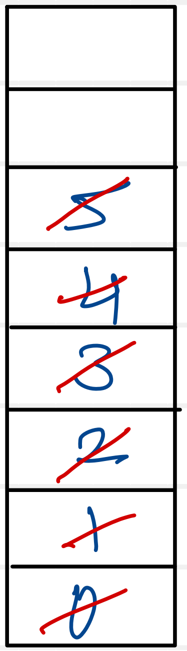




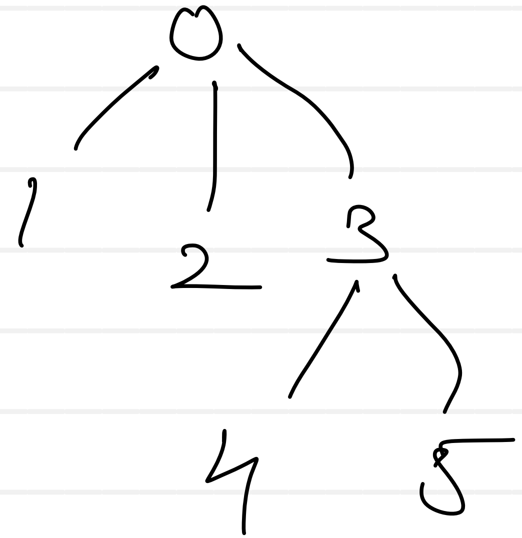
Spanning Tree

DFS Spanning Tree



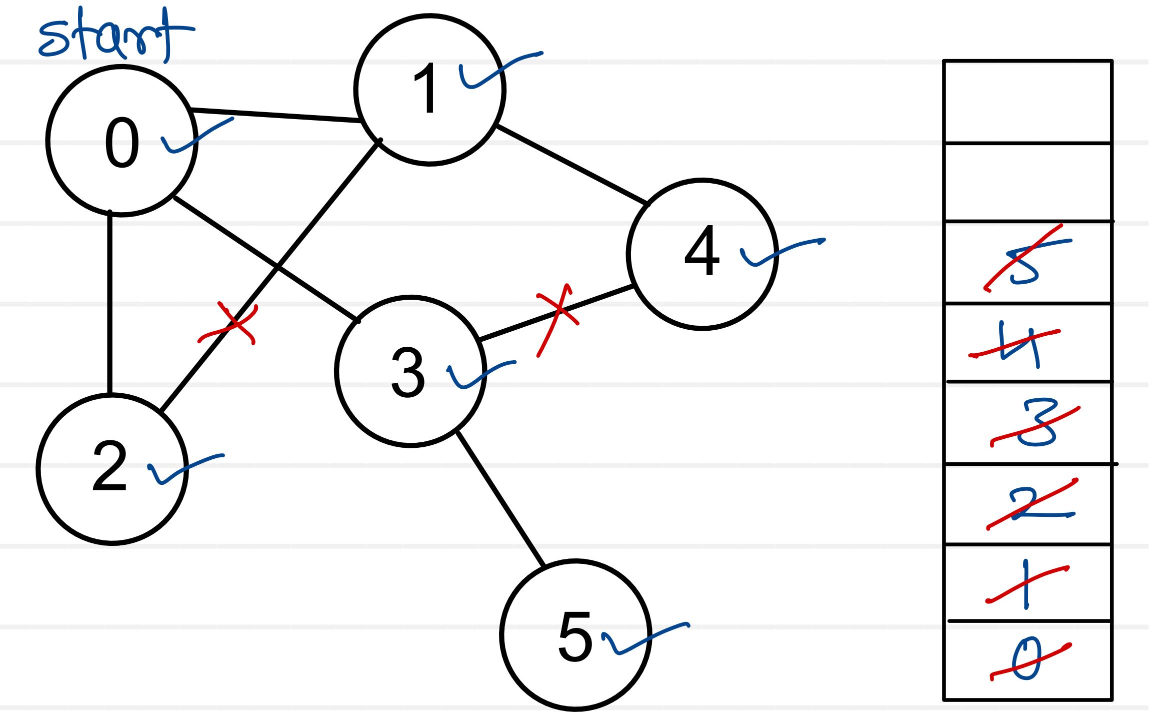


1. push starting vertex on stack & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the stack, mark them and also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until stack is empty.

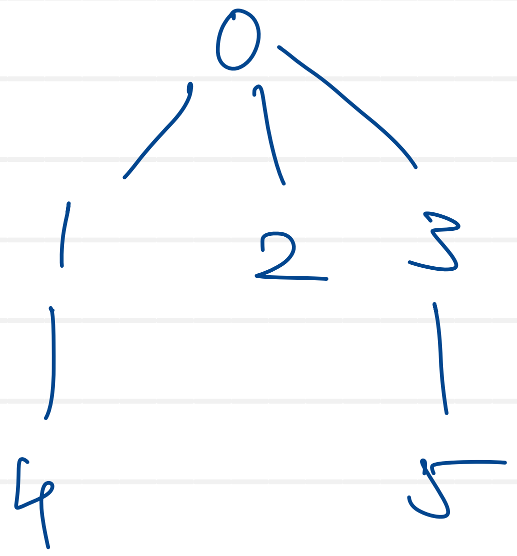


Spanning tree : (0-1), (0-2), (0-3), (3-4), (3-5)

BFS Spanning Tree



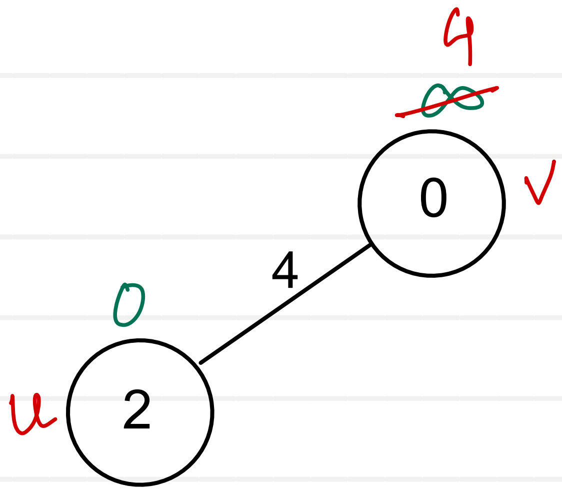
1. push starting vertex on queue & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the queue, mark them and also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until queue is empty.



Spanning tree : (0-1) (0-2) (0-3) (1-4) (3-5)

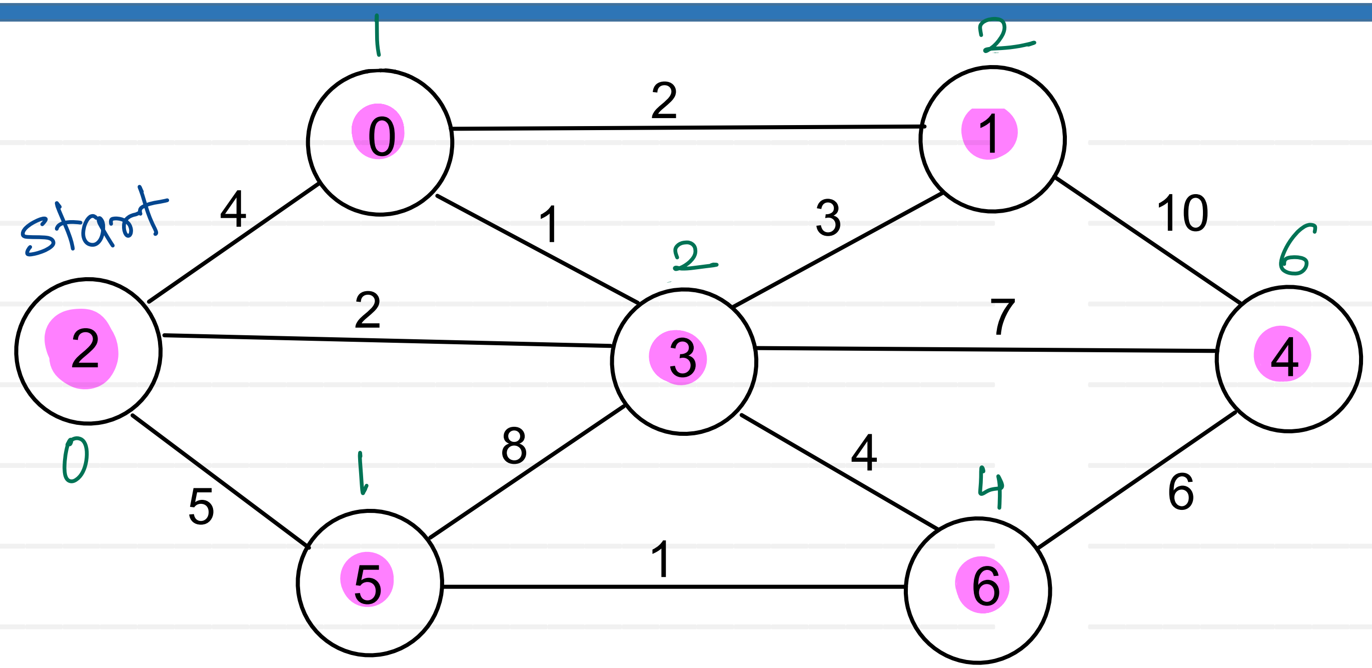
Prim’s Algorithm

1. Create a set mst to keep track of vertices included in MST.
2. Also keep track of parent of each vertex. Initialize parent of each vertex -1.
3. Assign a key to all vertices in the input graph. Key for all vertices should be initialized to INF. The start vertex key should be 0.
4. While mst doesn't include all the vertices
   - i. Pick a vertex u which is not there in mst and has minimum key.
   - ii. Include vertex u to mst.
   - iii. Update key and parent of all adjacent vertices of u.
     - a. For each adjacent vertex v,
       if weight of edge u-v is less than the current key of v,
       then update the key as weight of u-v.
     - b. Record u as parent of v.



if(adjmat[u][v] < key[v])
key[v] = adjmat[u][v]

Prim's Algorithm



	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	6	6
5	1	6
6	4	3

16

	K	P
0	4	2
1	∞	-1
2	0	-1
3	2	2
4	∞	-1
5	5	2
6	∞	-1

	K	P
0	1	3
1	3	3
2	0	-1
3	2	2
4	7	3
5	5	2
6	4	3

	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	7	3
5	5	2
6	4	3

	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	7	3
5	5	2
6	4	3

	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	6	6
5	1	6
6	4	3

	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	6	6
5	1	6
6	4	3