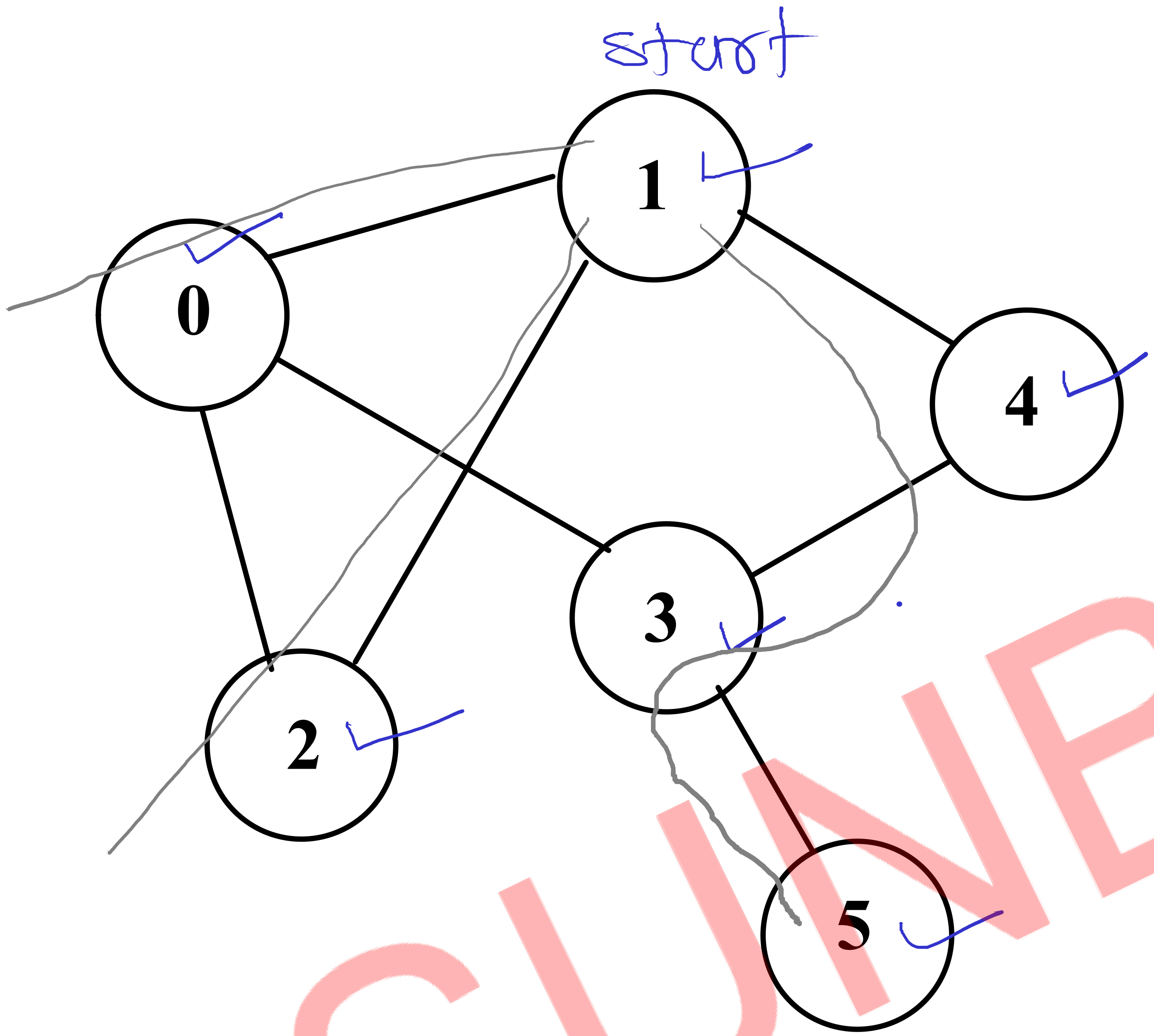
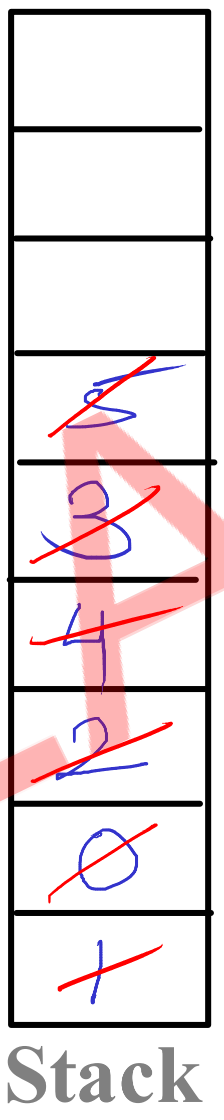
DFS Traversal





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

//1. Choose a vertex as start vertex.

//2. Push start vertex on stack & mark it.

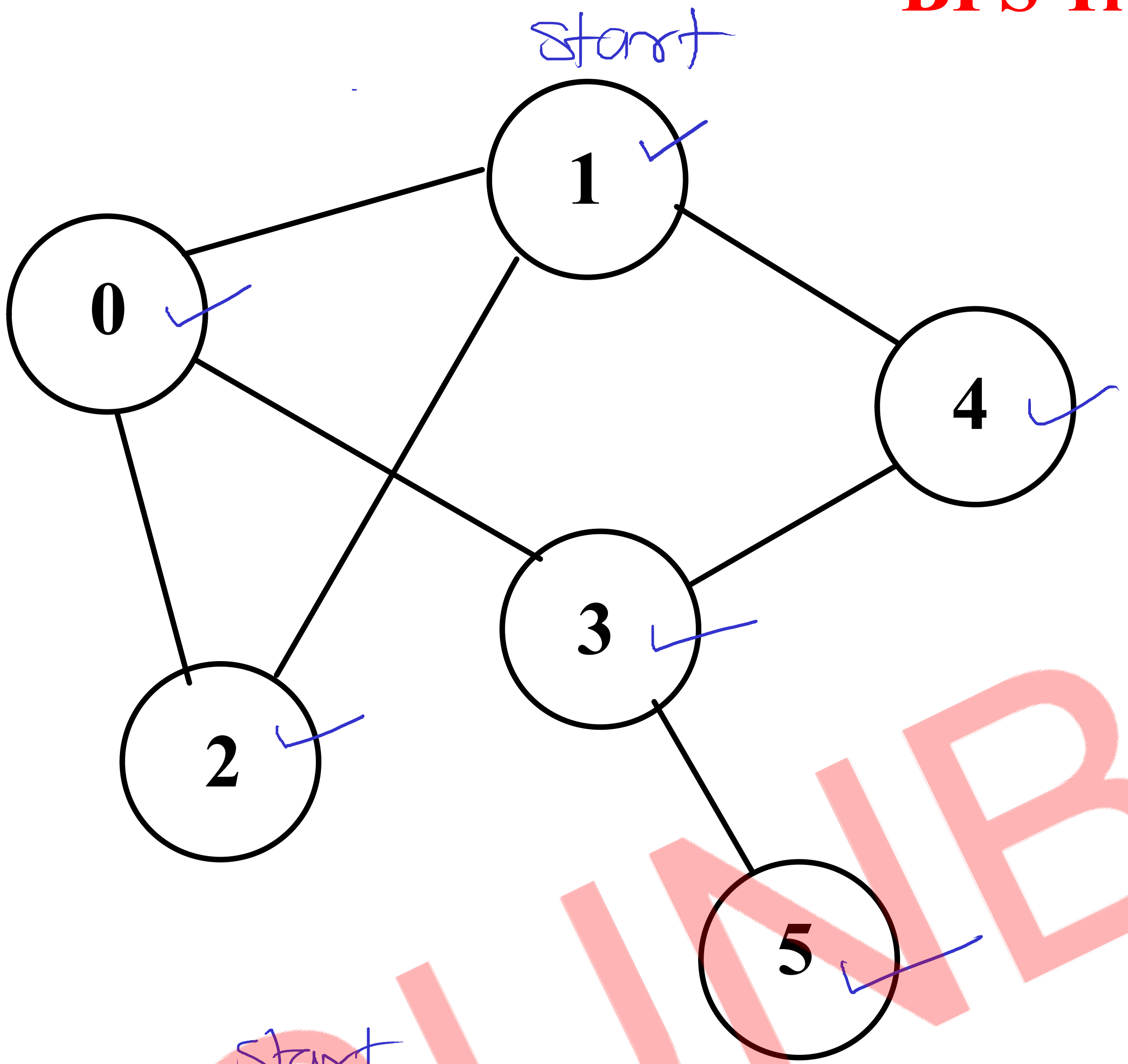
//3. Pop vertex from stack.

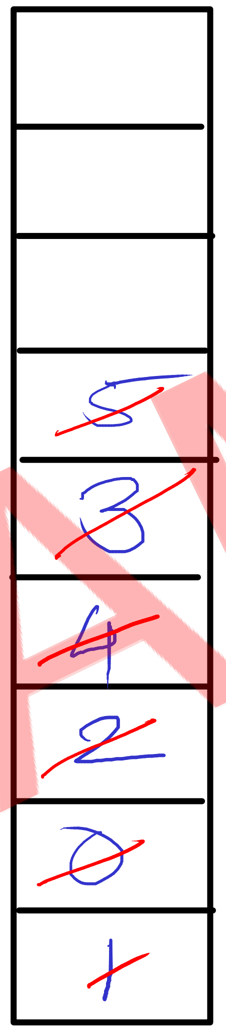
//4. Print the vertex.

//5. Put all non-visited neighbours of the vertex
//on the stack and mark them.

//6. Repeat 3-5 until stack is empty.

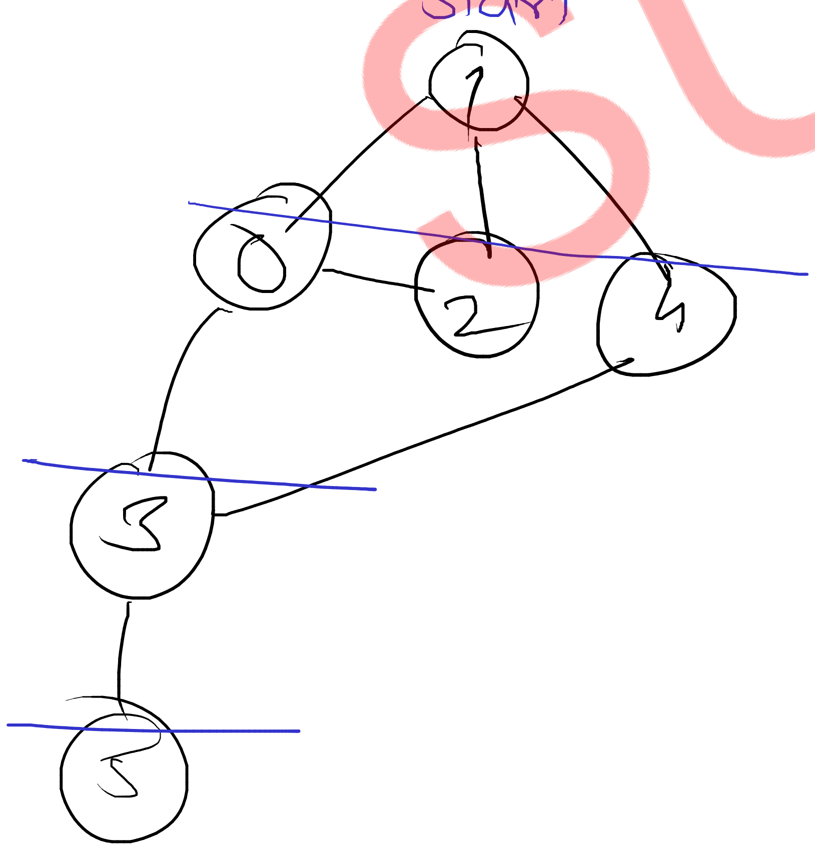
BFS Traversal





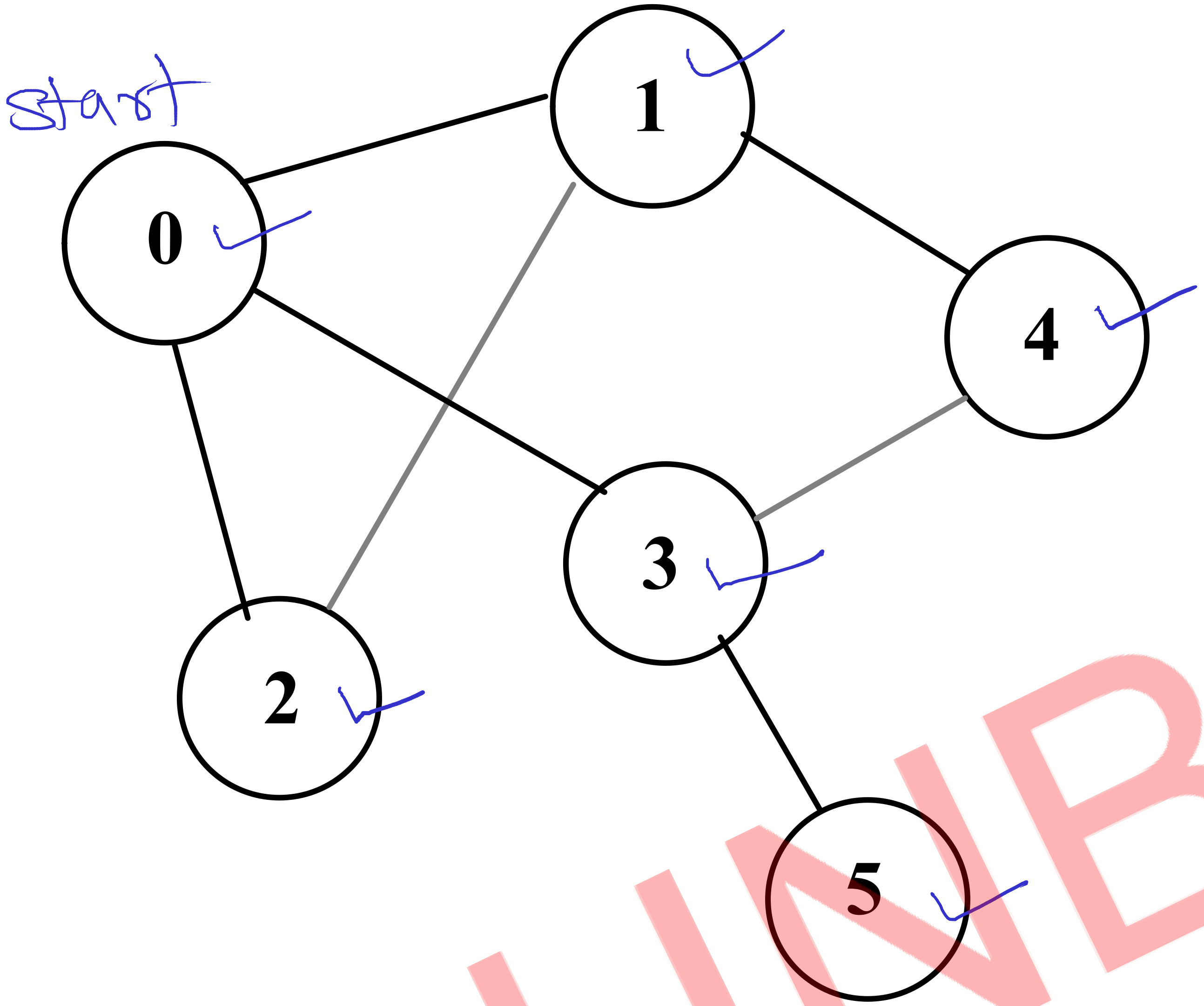
Queue

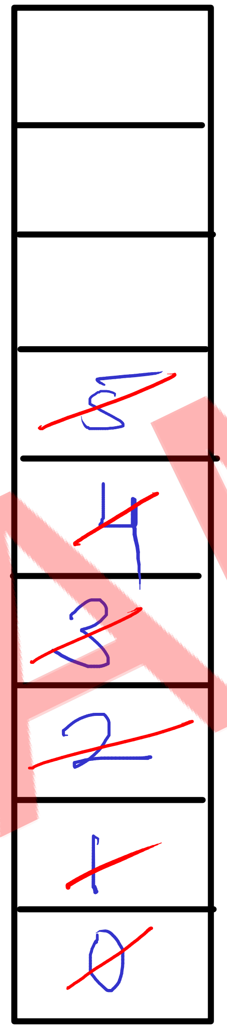
Traversal:
1, 0, 2, 4, 3, 5



//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.

Single Source Path length





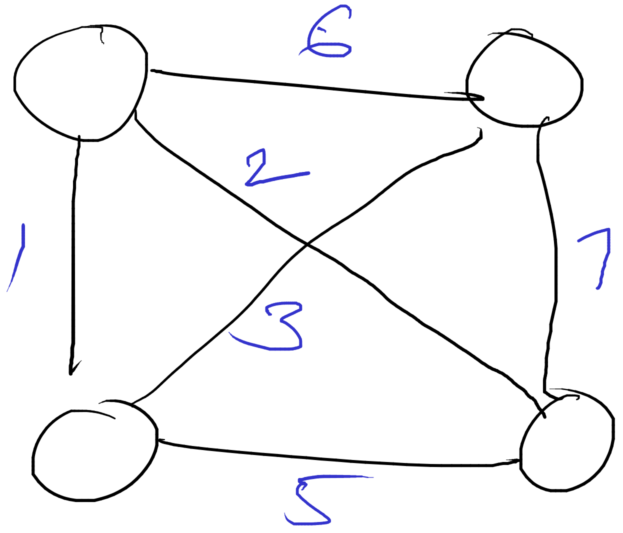
Queue

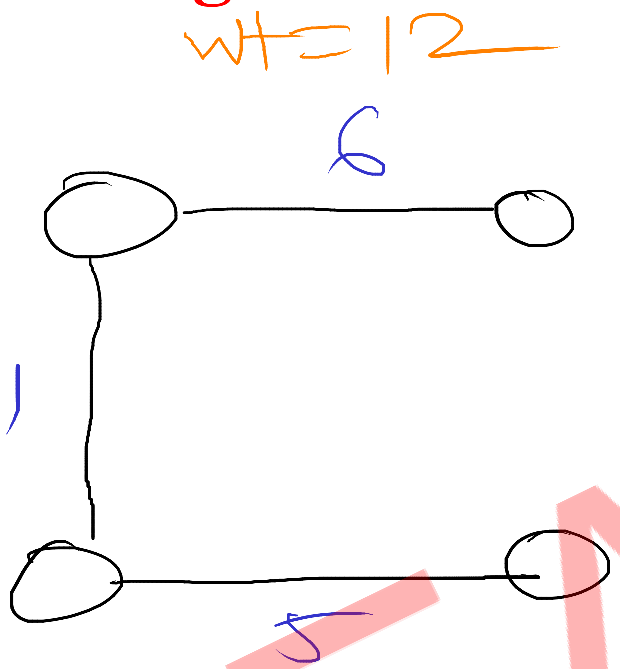
0	0
1	1
2	1
3	1
4	2
5	2

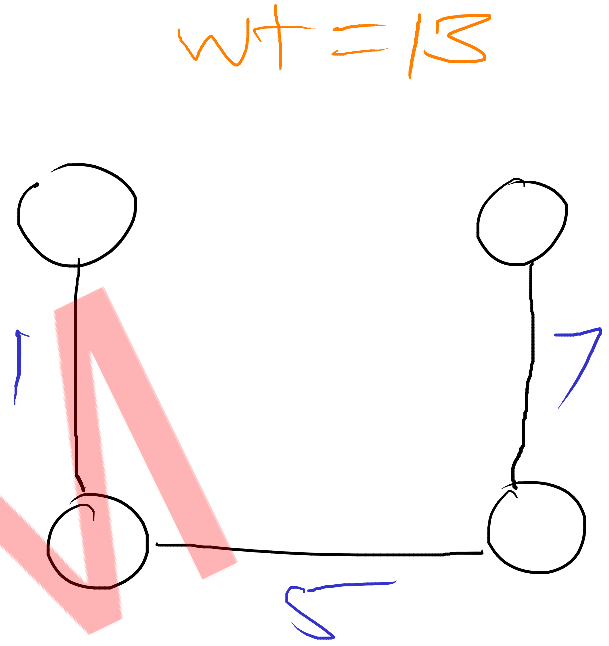
Length

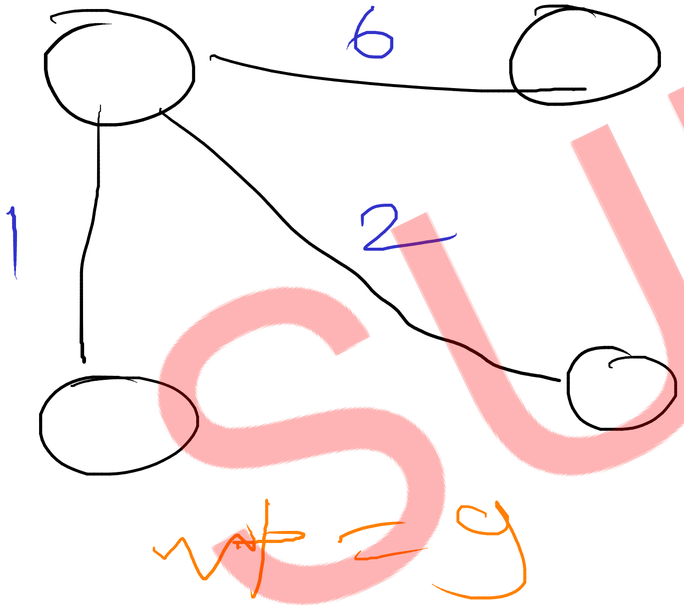
//1. Create path length array to keep length of vertex from start vertex.
//2. push start on queue & mark it.
//3. pop the vertex.
//4. push all its non-marked neighbors on the queue, mark them.
//5. For each such vertex calculate length as length[neighbor] = length[current] + 1
//6. print current vertex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

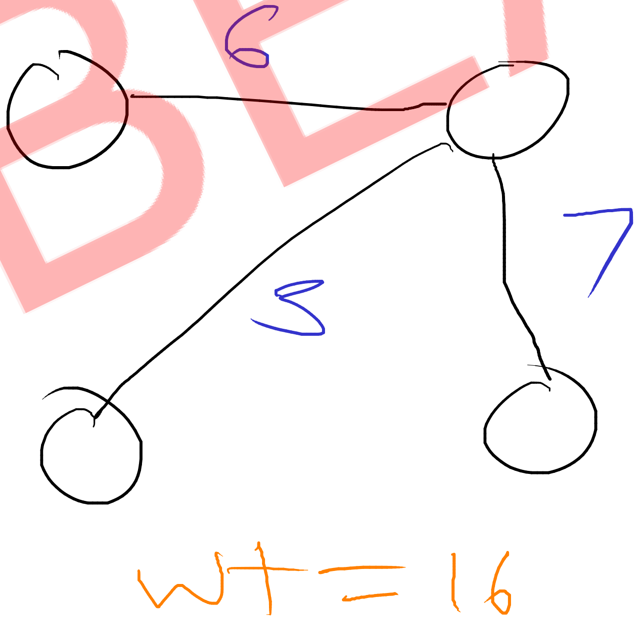
Spanning Tree





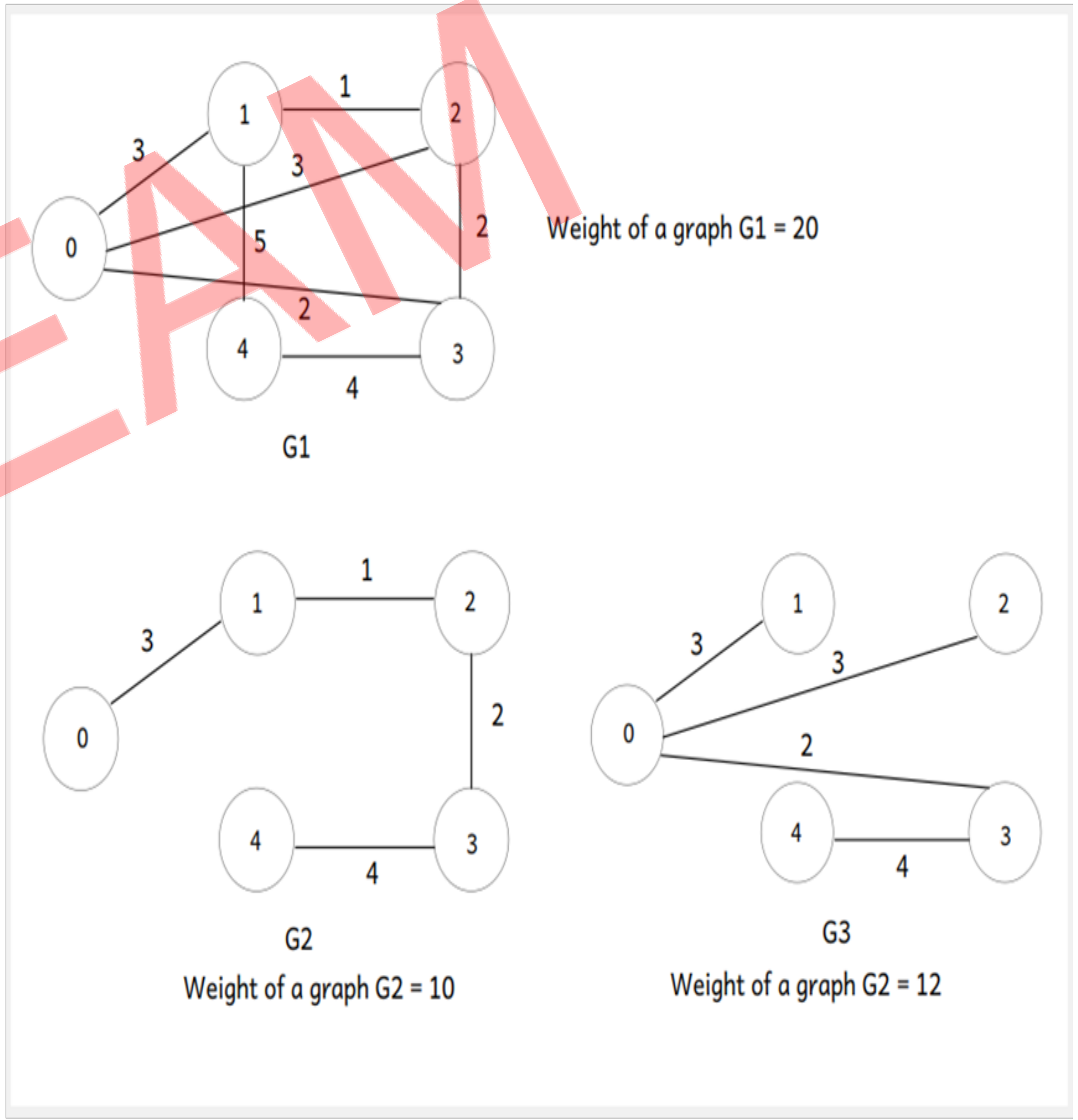






Spanning Tree

- 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



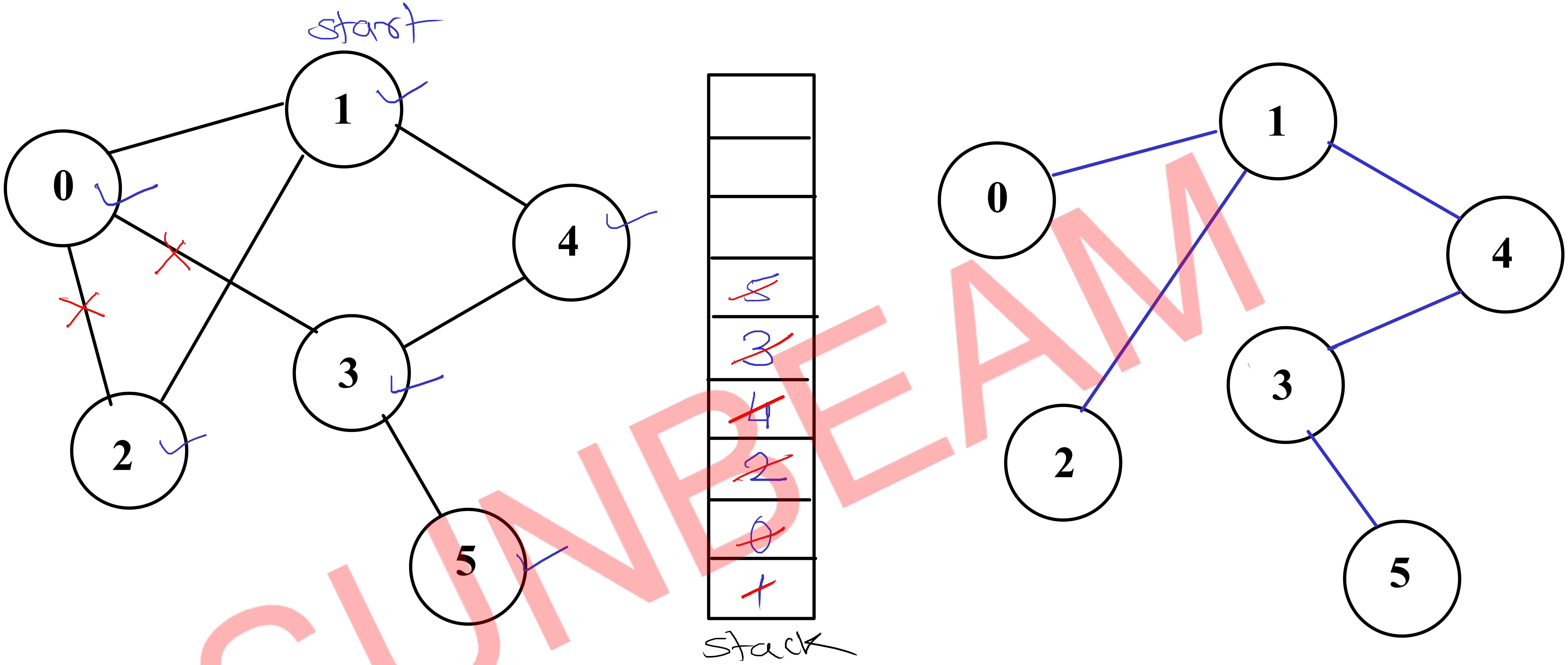
Weight of a graph G1 = 20

G1

G2
Weight of a graph G2 = 10

G3
Weight of a graph G2 = 12

DFS Spanning Tree



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

//1. push starting vertex on stack & mark it.

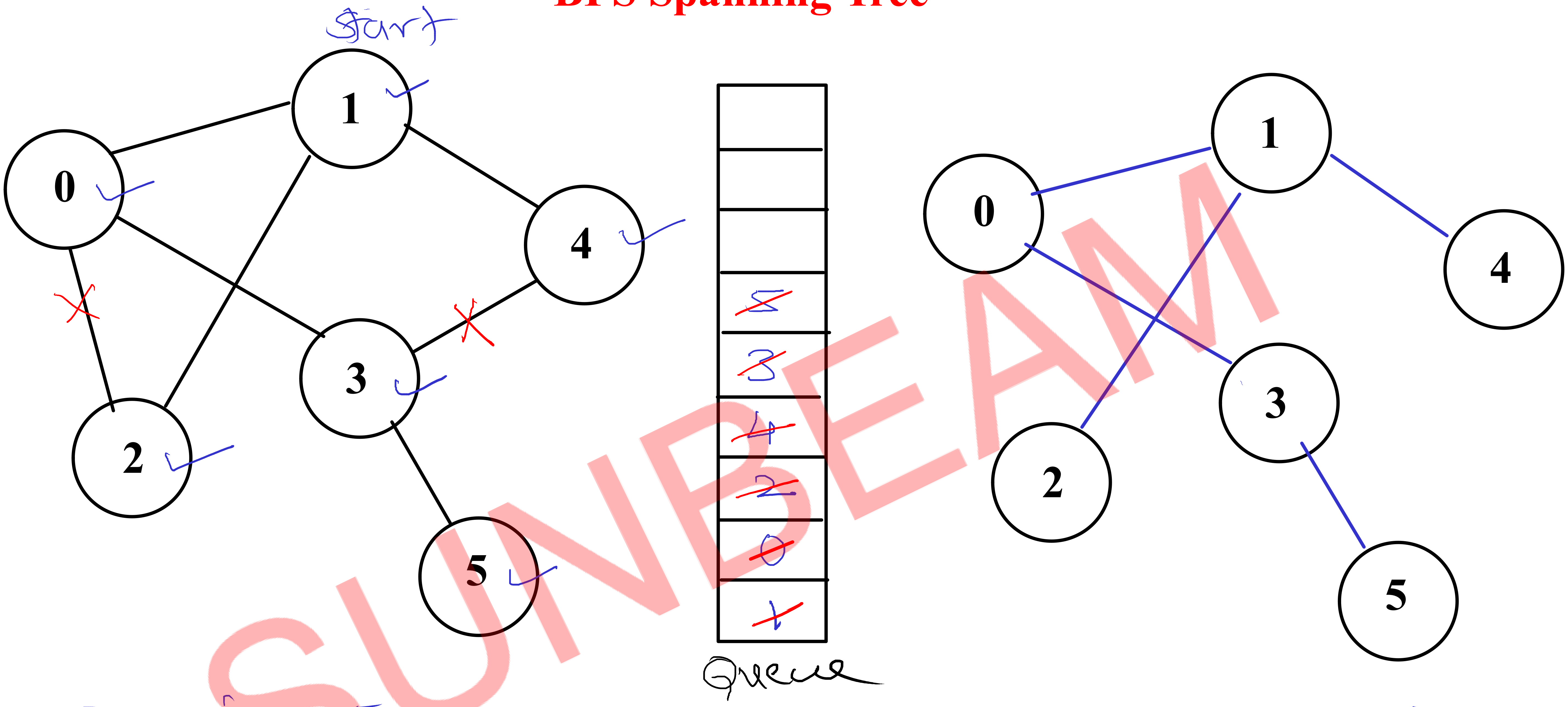
//2. pop the vertex.

//3. push all its non-marked neighbors on the stack, mark them.

//Also print the vertex to neighboring vertex edges.

4. repeat steps 2-3 until stack is empty.

BFS Spanning Tree



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

//1. push starting vertex on queue & mark it.

//2. pop the vertex.

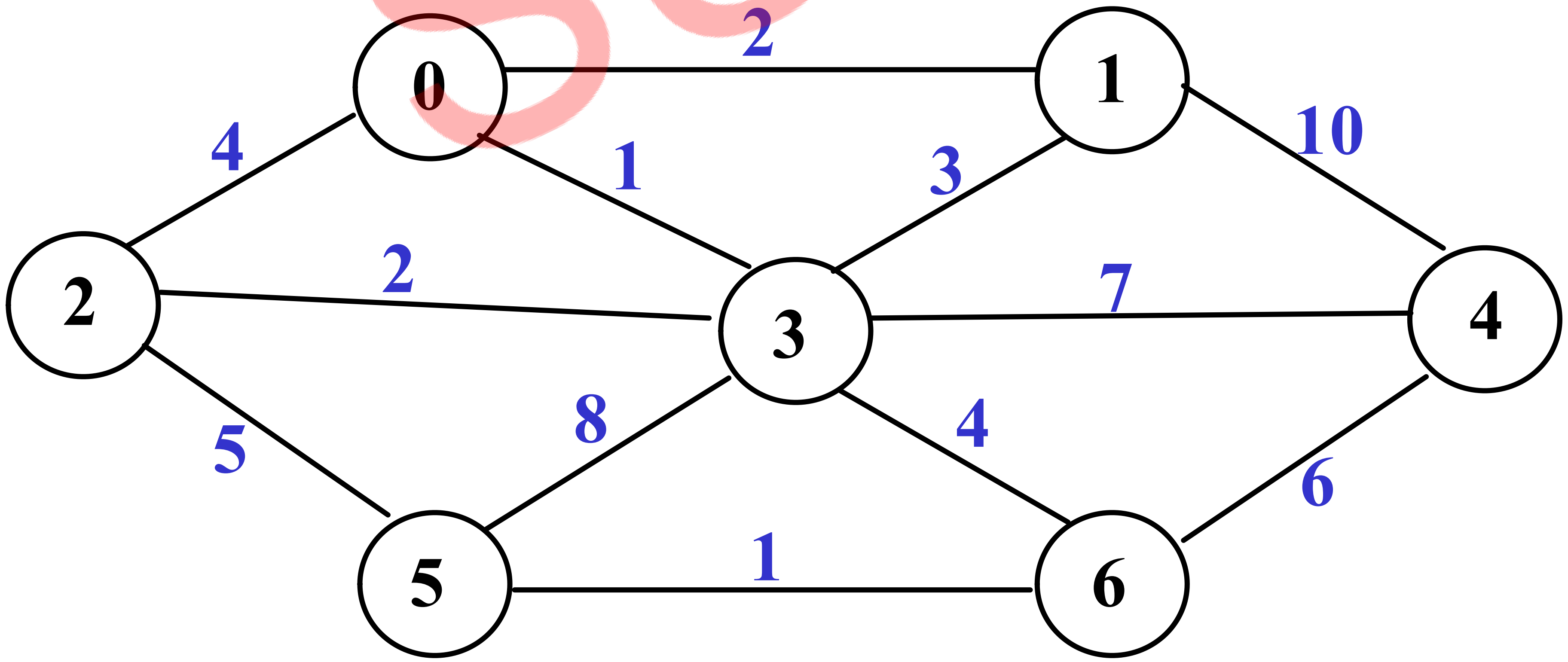
//3. push all its non-marked neighbors on the queue, mark them.

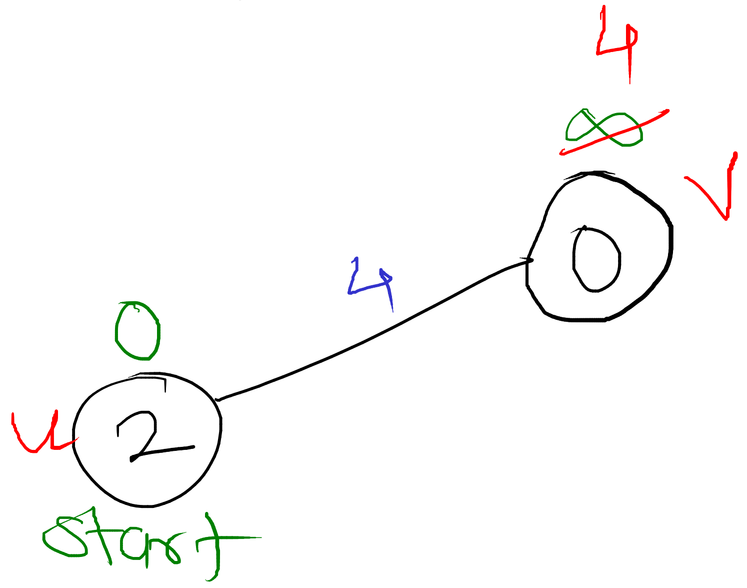
//Also print the vertex to neighboring vertex edges.

//4. repeat steps 2-3 until queue is empty.

Prim's MST

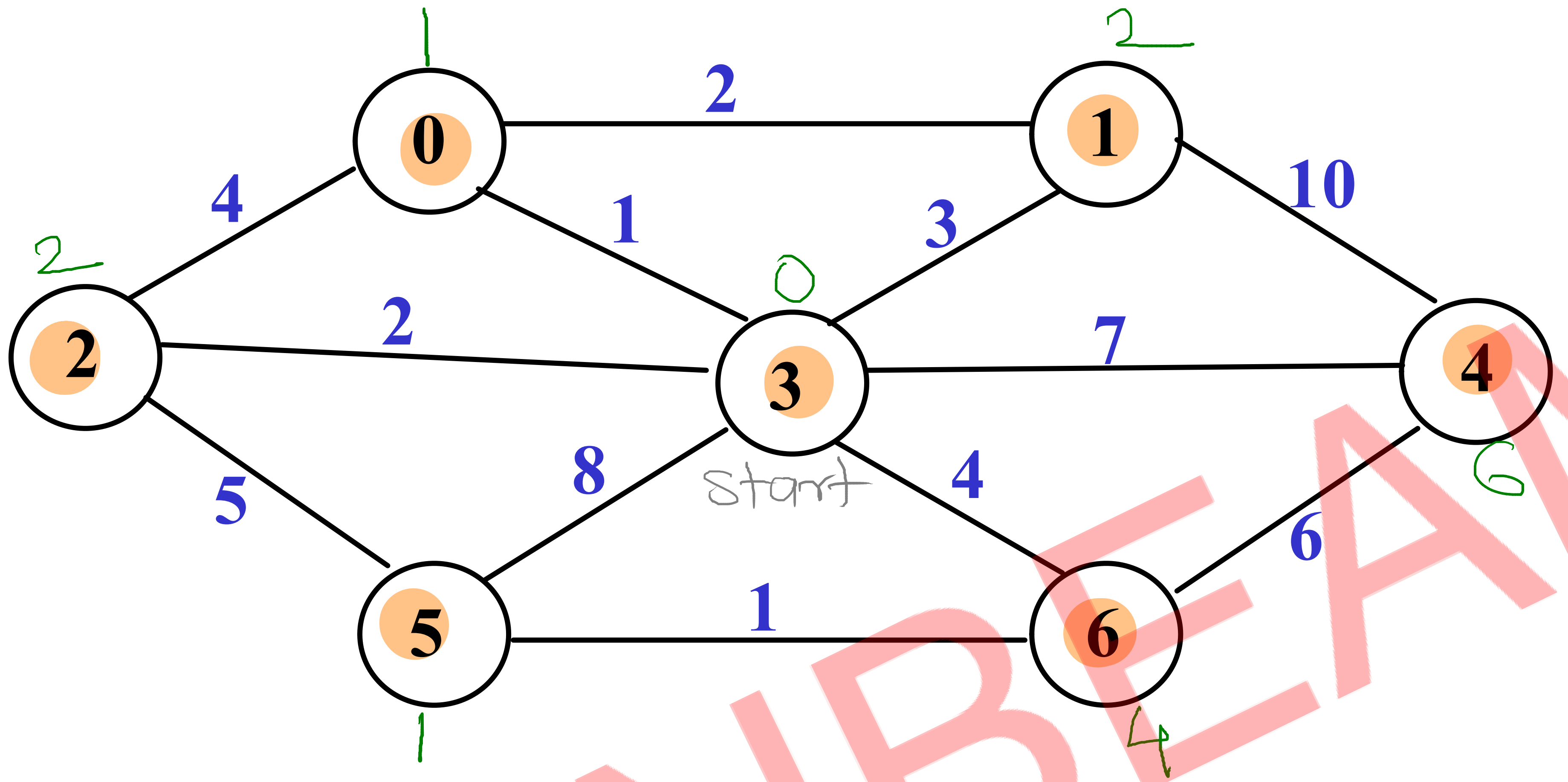
1. Create a set mst to keep track of vertices included in MST.
2. Also keep track of parent of each vertex. Initialize parent of each vertex -1.
3. Assign a key to all vertices in the input graph. Key for all vertices should be initialized to INF. The start vertex key should be 0.
4. While mst doesn't 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(wt(u,v) < key(v))
   key(v) = wt(u,v);
```

Prim's MST



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

	K	P
0	1	3
1	3	3
2	2	3
3	0	-1
4	7	3
5	8	3
6	4	3

	K	P
0	1	3
1	2	0
2	2	3
3	0	-1
4	7	3
5	8	3
6	4	3

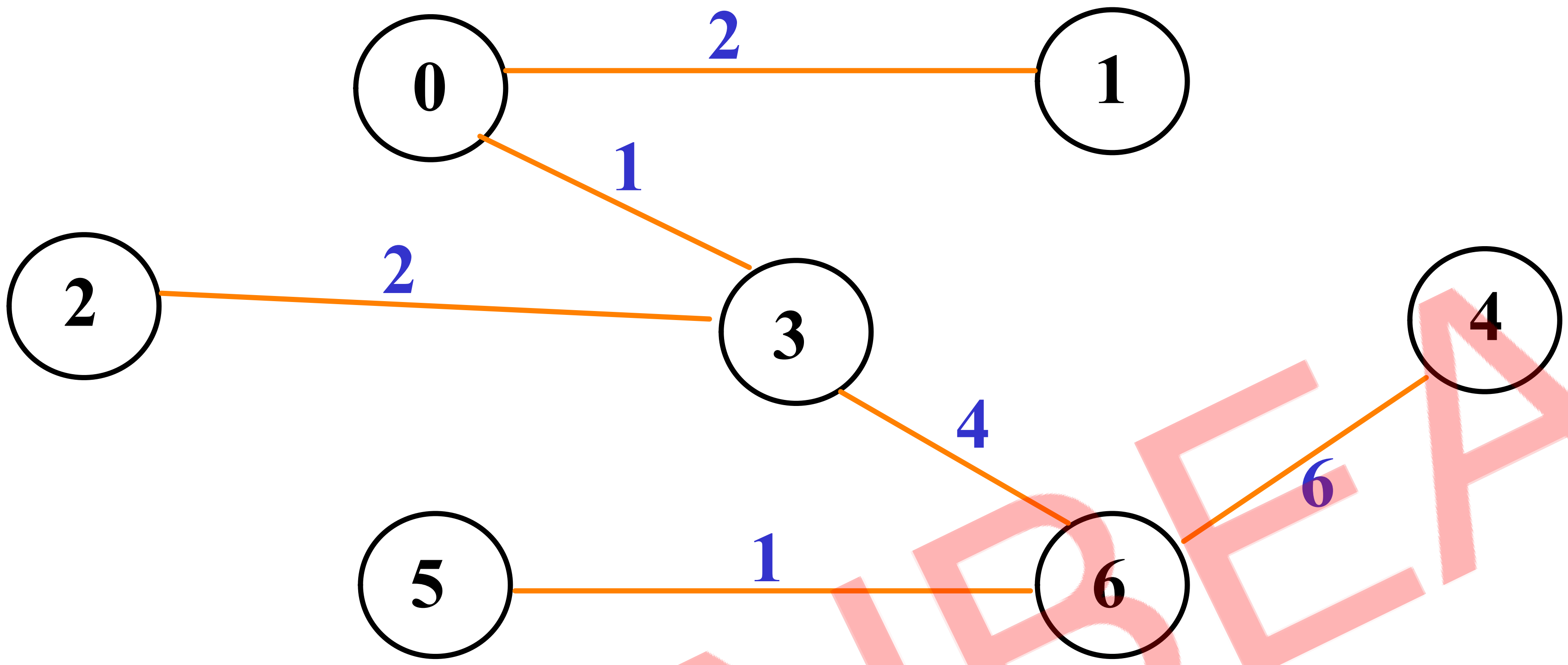
	K	P
0	1	3
1	2	0
2	2	3
3	0	-1
4	7	3
5	8	3
6	4	3

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

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

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

Prim's MST



weight = 16

	K	P
0	1	3
1	2	0
2	2	3
3	0	-1
4	6	6
5	1	6
6	4	3
	16	

```
public int findMinKeyVertex() {
                    int minKey = 999, minKeyVertex = -1;
                    for(int i = 0 ; i < keys.length ; i++) {
                           if(!mst[i] && keys[i] < minKey) {
                                  minKey = keys[i];
                                  minKeyVertex = i;
                           }
                    }
                    return minKeyVertex;
             }
```

minkey	minkeyVertex
∞	-1
0	3

Keys	∞	∞	∞	0	∞	∞	∞
	0	1	2	3	4	5	6

mst	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Dijkstra's Algorithm

1. Create a set spt to keep track of vertices included in shortest path tree.

2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.

3. While spt doesn't include all the vertices

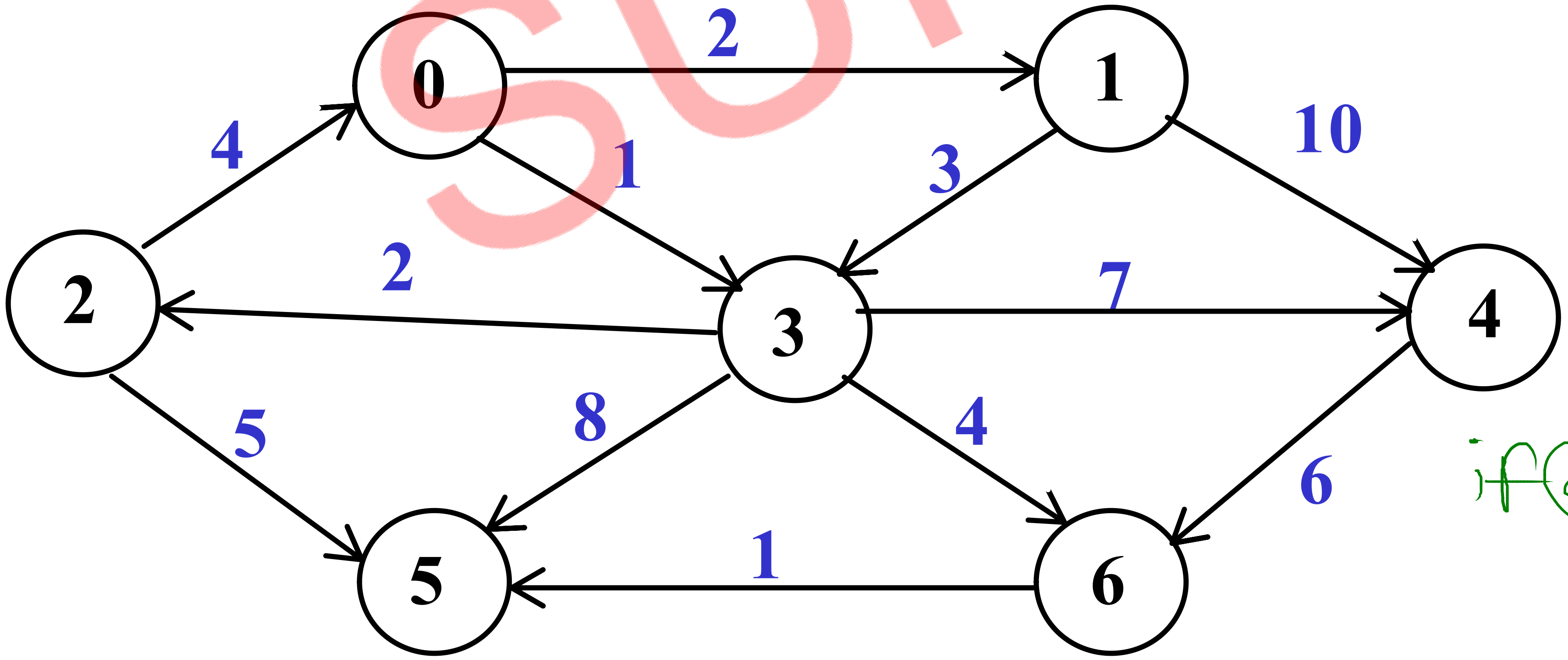
    i. Pick a vertex u which is not there in spt and has minimum distance.

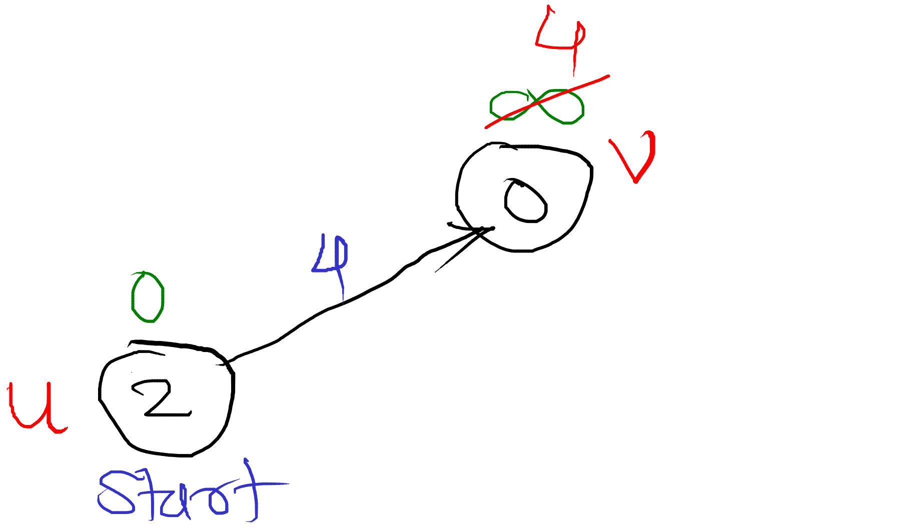
    ii. Include vertex u to spt.

    iii. Update distances of all adjacent vertices of u.

    For each adjacent vertex v,

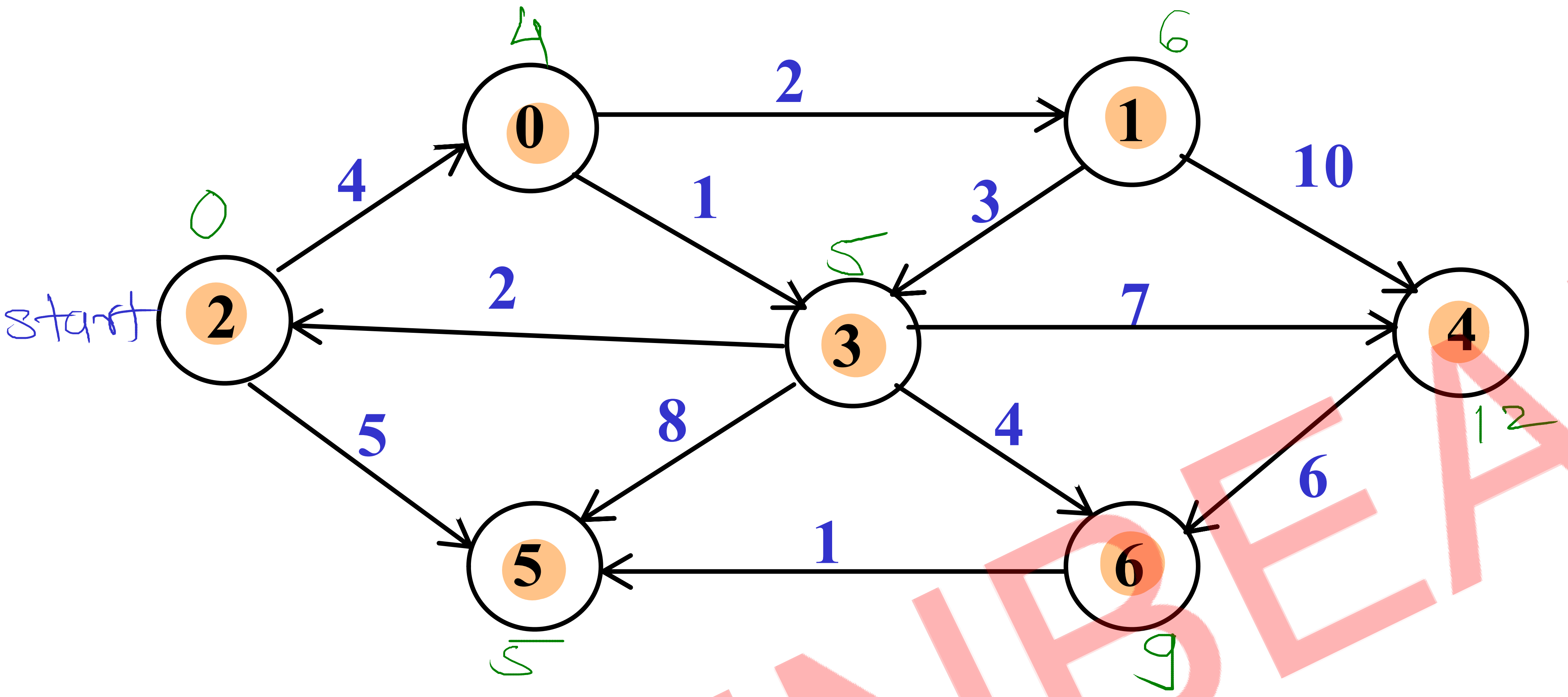
        if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.





```
if(dist[u]+wt(u-v)<dist[v])
    dist[v]=dist[u]+wt(u-v);
```

Dijkstra's Algorithm



	D
0	4
1	6
2	0
3	5
4	12
5	5
6	9

	D
0	4
1	∞
2	0
3	∞
4	∞
5	5
6	∞

	D
0	4
1	6
2	0
3	5
4	∞
5	5
6	∞

	D
0	4
1	6
2	0
3	5
4	12
5	5
6	9

	D
0	4
1	6
2	0
3	5
4	12
5	5
6	9

	D
0	4
1	6
2	0
3	5
4	12
5	5
6	9

	D
0	4
1	6
2	0
3	5
4	12
5	5
6	9