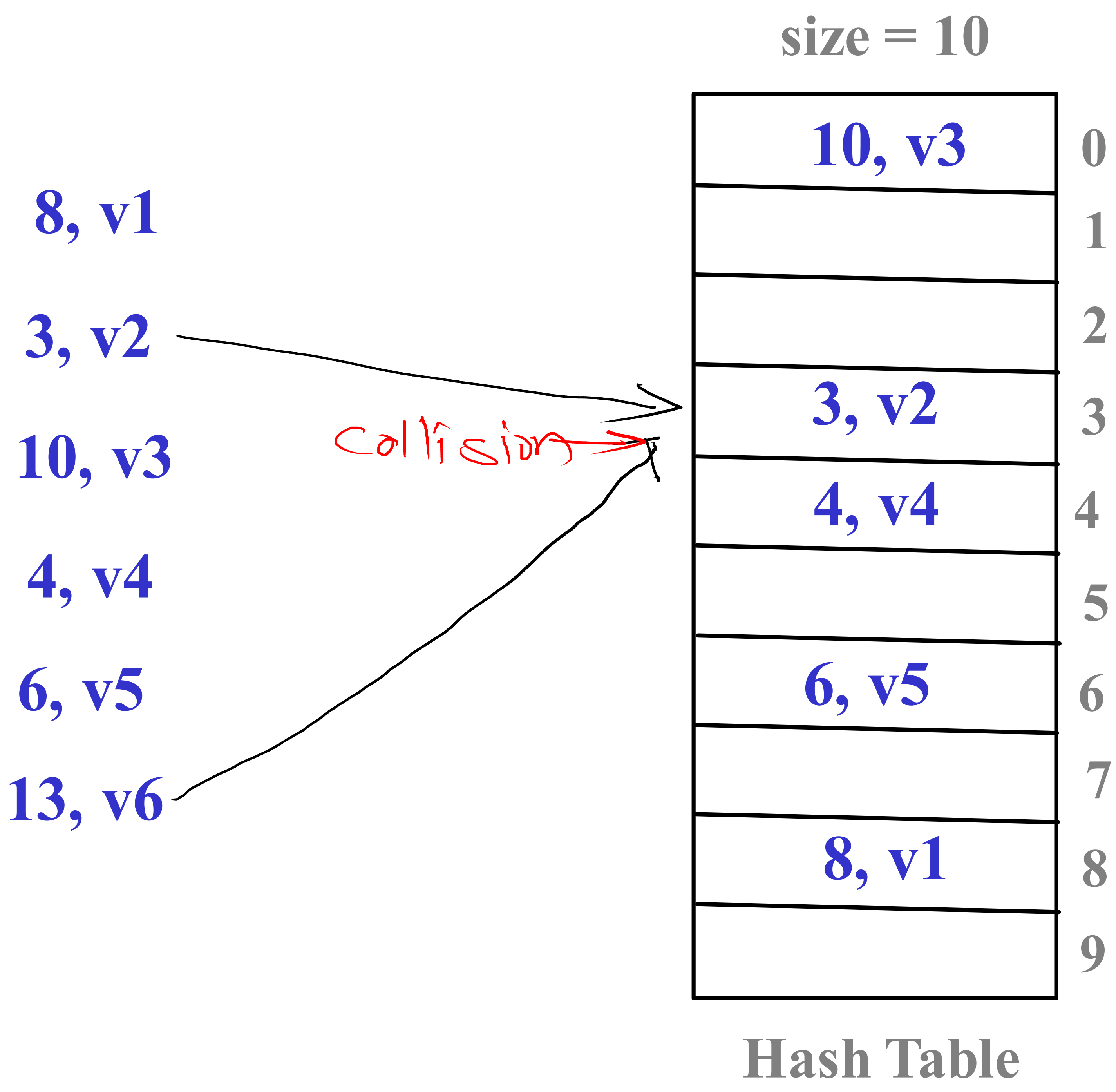
# Hashing

$h(k) = k \% size$




$h(8) = 8 \% 10 = 8$

$h(3) = 3 \% 10 = 3$

$h(10) = 10 \% 10 = 0$

$h(4) = 4 \% 10 = 4$

$h(6) = 6 \% 10 = 6$

$h(13) = 13 \% 10 = 3$ (collision)

# Closed Addressing/ Seperate Chaining / Chaining

8, v1

3, v2

10, v3

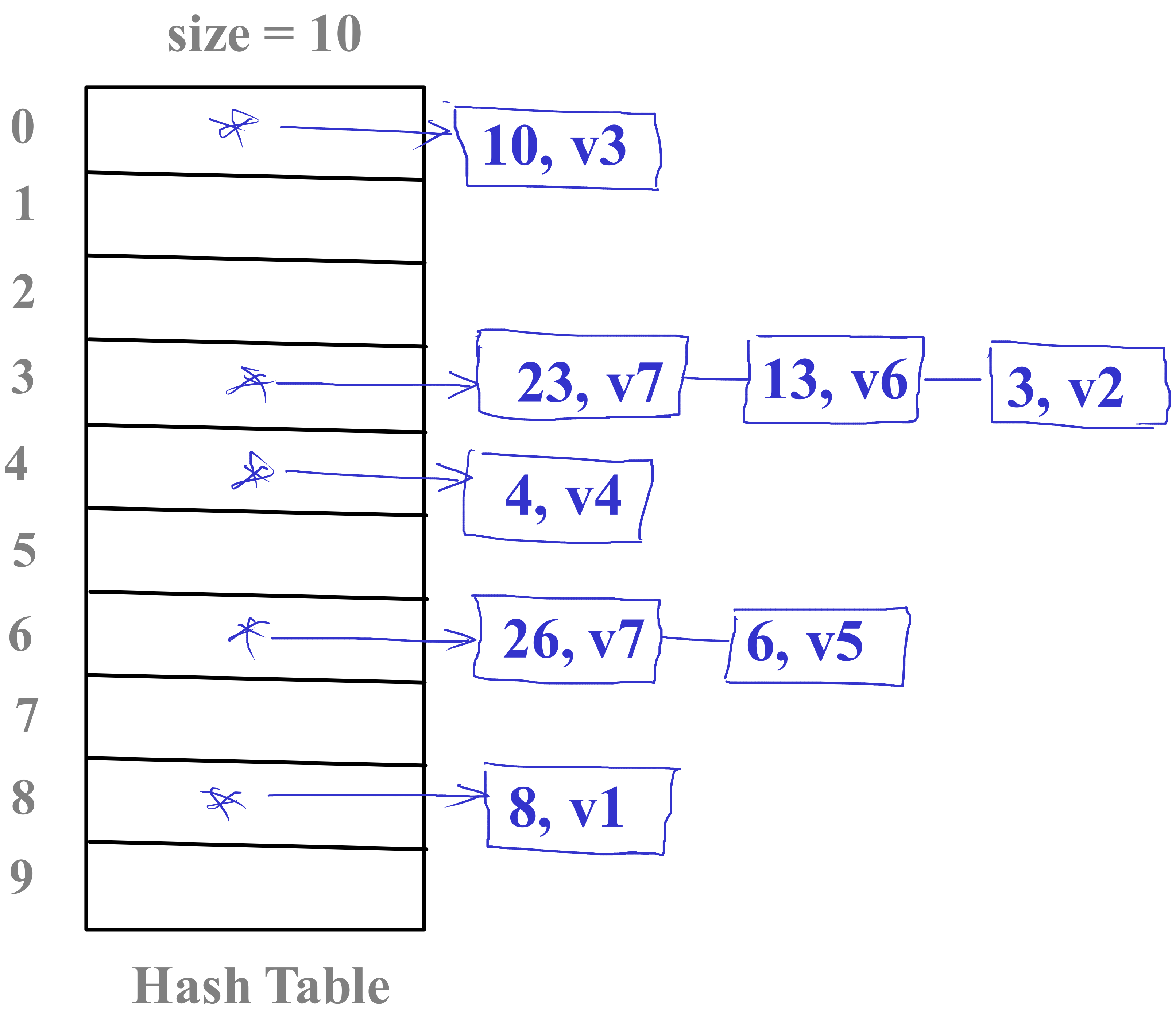
4, v4

6, v5

13, v6

23, v7

26, v7




$h(k) = k \% size$

$h(8) = 8 \% 10 = 8$

$h(3) = 3 \% 10 = 3$

$h(10) = 10 \% 10 = 0$

$h(4) = 4 \% 10 = 4$

$h(6) = 6 \% 10 = 6$

$h(13) = 13 \% 10 = 3$

$h(23) = 23 \% 10 = 3$

$h(26) = 26 \% 10 = 6$

# Open Addressing - Linear Probing

8, v1

3, v2

10, v3

4, v4

6, v5

13, v6

collision

size = 10

| | |
|---|---|
| 0 | 10, v3 |
| 1 | |
| 2 | |
| 3 | 3, v2 |
| 4 | 4, v4 |
| 5 | 13, v6 |
| 6 | 6, v5 |
| 7 | |
| 8 | 8, v1 |
| 9 | |

Hash Table

**h(k) = key % size**

**h(k, i) = [ h(k) + f(i) ] % size**

**f(i) = i**

**where i = 1, 2, 3, .....**

$h(8) = 8 \% 10 = 8$

$h(3) = 3 \% 10 = 3$

$h(10) = 10 \% 10 = 0$

$h(4) = 4 \% 10 = 4$

$h(6) = 6 \% 10 = 6$

$h(13) = 13 \% 10 = 3$ (collision)

$h(13, 1) = [3 + 1] \% 10$

$= 4$ (collision) (1st probe)

$h(13, 2) = [3 + 2] \% 10$

$= 5$ (2nd probe)

**Primary Clustering**

**- takes long run to find slot "near" key position**

# Open Addressing - Quadratic Probing

8, v1

3, v2

10, v3

4, v4

6, v5

13, v6

size = 10

collision

| | |
|---|---|
| 10, v3 | 0 |
| | 1 |
| | 2 |
| 3, v2 | 3 |
| 4, v4 | 4 |
| | 5 |
| 6, v5 | 6 |
| 13, v6 | 7 |
| 8, v1 | 8 |
| | 9 |

Hash Table

$h(k) = key \% size$

$h(k, i) = [\ h(k) + f(i)\ ] \% size$

$f(i) = i^2$

where i = 1, 2, 3, .....

$h(13) = 13 \% 10 = 3$ (collision)

$h(13,1) = [3+1] \% 10$

$= 4$ (1st probe)(collision)

$h(13,2) = [3+4] \% 10$

$= 7$ (2nd probe)

# Open Addressing - Quadratic Probing

size = 10

| | |
|---|---|
| 10, v3 | 0 |
| | 1 |
| 23, v7 | 2 |
| 3, v2 | 3 |
| 4, v4 | 4 |
| | 5 |
| 6, v5 | 6 |
| 13, v6 | 7 |
| 8, v1 | 8 |
| 33, v8 | 9 |

Hash Table

23, v7

33, v8

**h(k) = key % size**

**h(k, i) = [ h(k) + f(i) ] % size**

**f(i) = i^2**

**where i = 1, 2, 3, .....**

h(23) = 23 % 10 = 3 (collision)

h(23, 1) = [3 + 1] % 10 = 4 (1st) (collision)

h(23, 2) = [3 + 4] % 10 = 7 (2nd) (collision)

h(23, 3) = [3 + 9] % 10 = 2 (3rd)

h(33) = 33 % 10 = 3 (collision)

h(33, 1) = [3 + 1] % 10 = 4 (1st) (collision)

h(33, 2) = [3 + 4] % 10 = 7 2nd) (collision)

h(33, 3) = [3 + 9] % 10 = 2 (3rd) (collision)

h(33, 4) = [3 + 16] % 10 = 9 (4th)

**Secondary clustering**

**- takes long run to find slot "away" form key position**

# Hashing - Double Hashing

**8, v1**

**3, v2**

**10, v3**

**14, v6**

size = 11

| | |
|---|---|
| | 0 |
| | 1 |
| | 2 |
| **3, v2** | 3 |
| | 4 |
| | 5 |
| **14, v6** | 6 |
| | 7 |
| **8, v1** | 8 |
| | 9 |
| **10, v3** | 10 |

Hash Table

$h1(k) = key \% size$

$h2(k) = 7 - (key \% 7)$

$h(k, i) = [h1(k) + i * h2(k)] \% size$

$h_1(8) = 8 \% 11 = 8$

$h_1(3) = 3 \% 11 = 3$

$h_1(10) = 10 \% 11 = 10$

$h_1(14) = 14 \% 11 = 3$ (collision)

$h_2(14) = 7 - 0 = 7$

$h(14, 1) = [3 + 1 * 7] \% 11$

$= 10$ ($1^{st}$) (collision)

$h(14, 2) = [3 + 2 * 7] \% 11$

$= 6$ ($2^{nd}$)

# Rehashing

$$\text{Load Factor} = \frac{n}{N}$$
$(\lambda)$

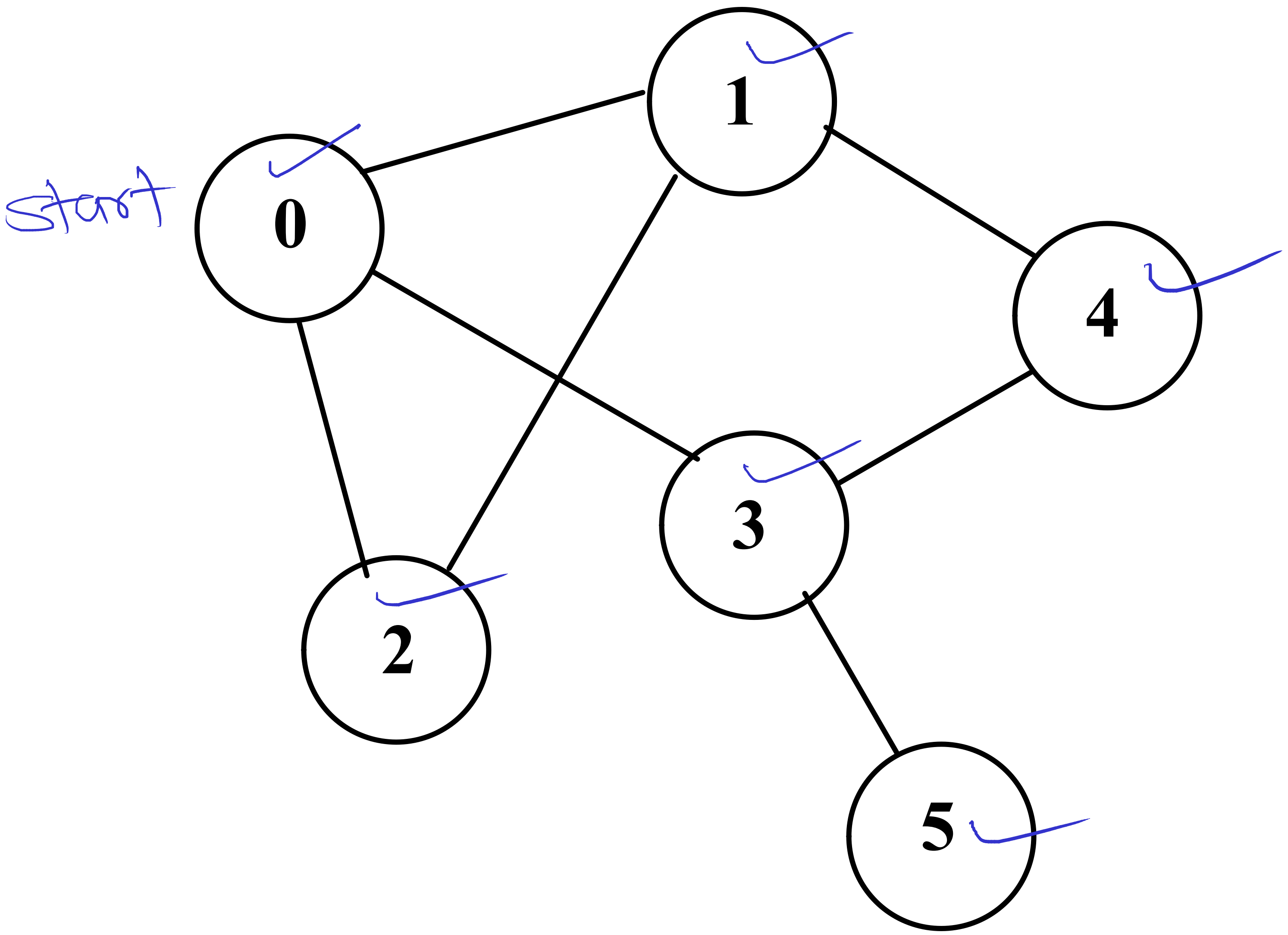
n - Number of elements (key value pairs) in hash table
N - Number of slots in hash table

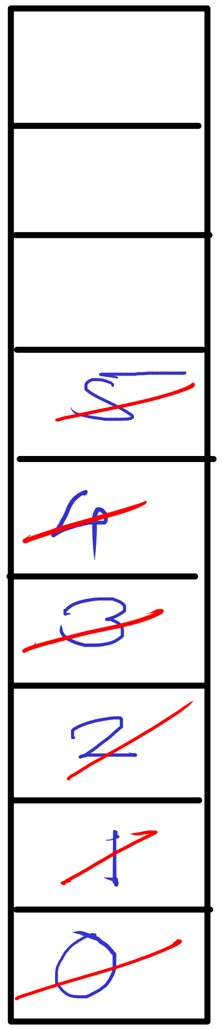
| | | |
|---|---|---|
| if $n < N$ | Load factor $< 1$ | - free slots are available |
| if $n = N$ | Load factor $= 1$ | - no free slots |
| if $n > N$ | Load factor $> 1$ | - can not insert at all |

- Rehashing is make the hash table size twice of existing size if hash table is 70 or 75 % full

- In rehashing existing key value pairs are again mapped according to new hash table size

# DFS Traversal



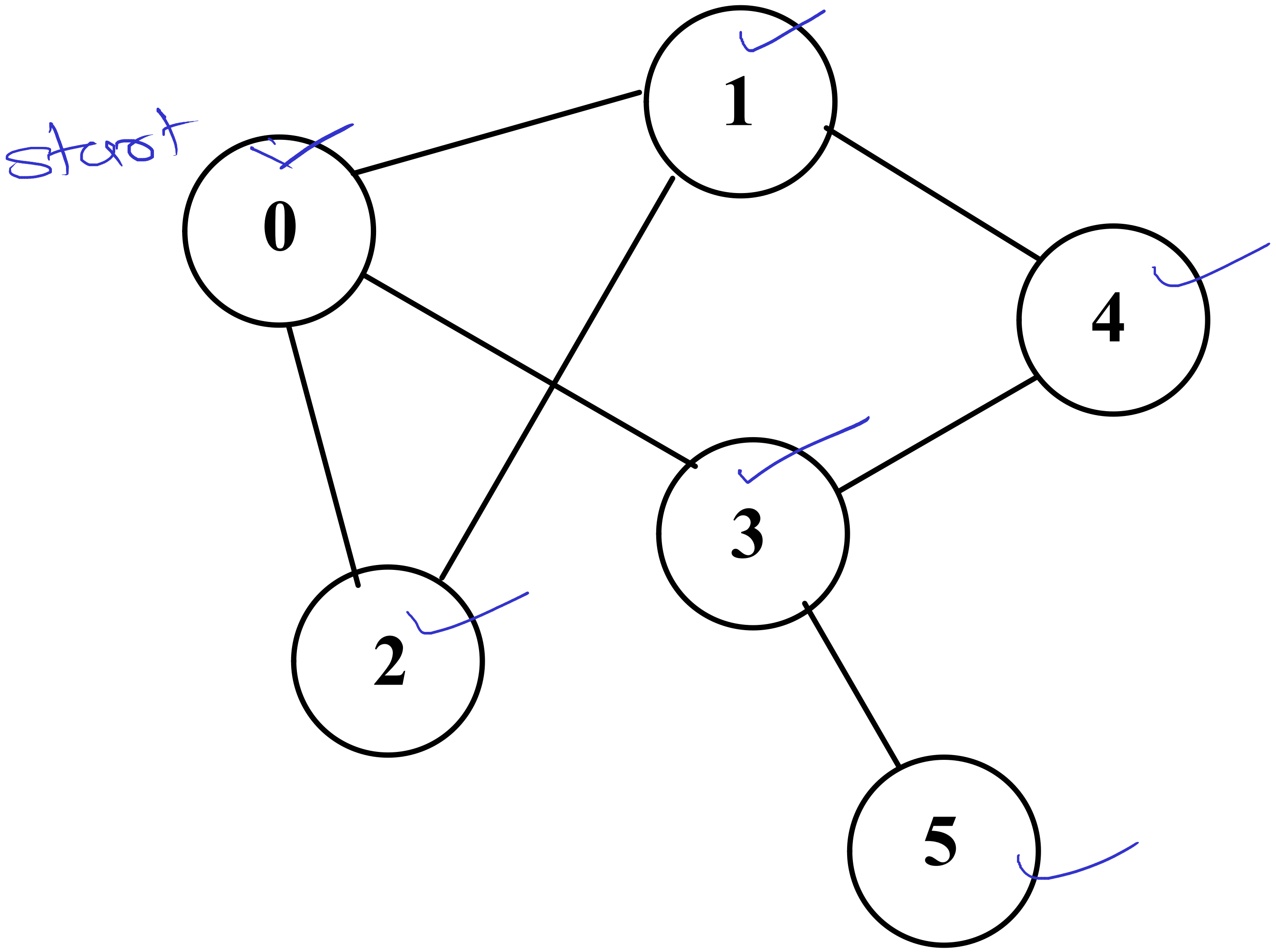



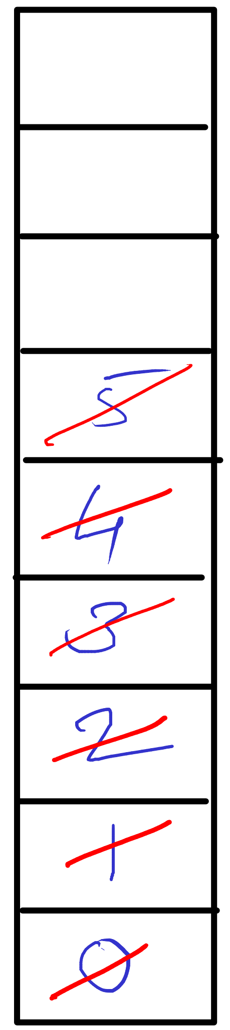

Stack

0, 3, 5, 4, 2, 1

//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



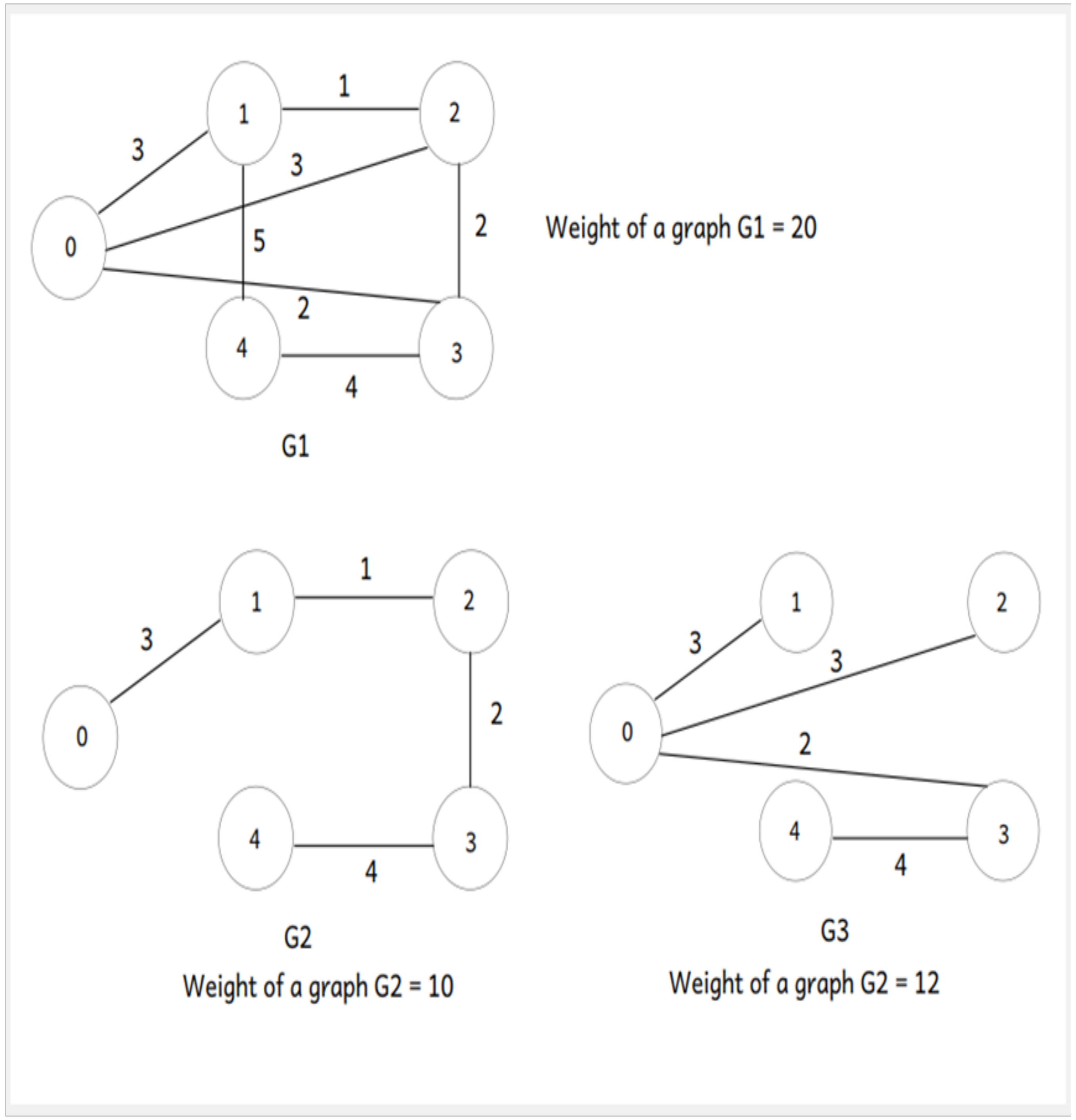

Queue

0, 1, 2, 3, 4, 5

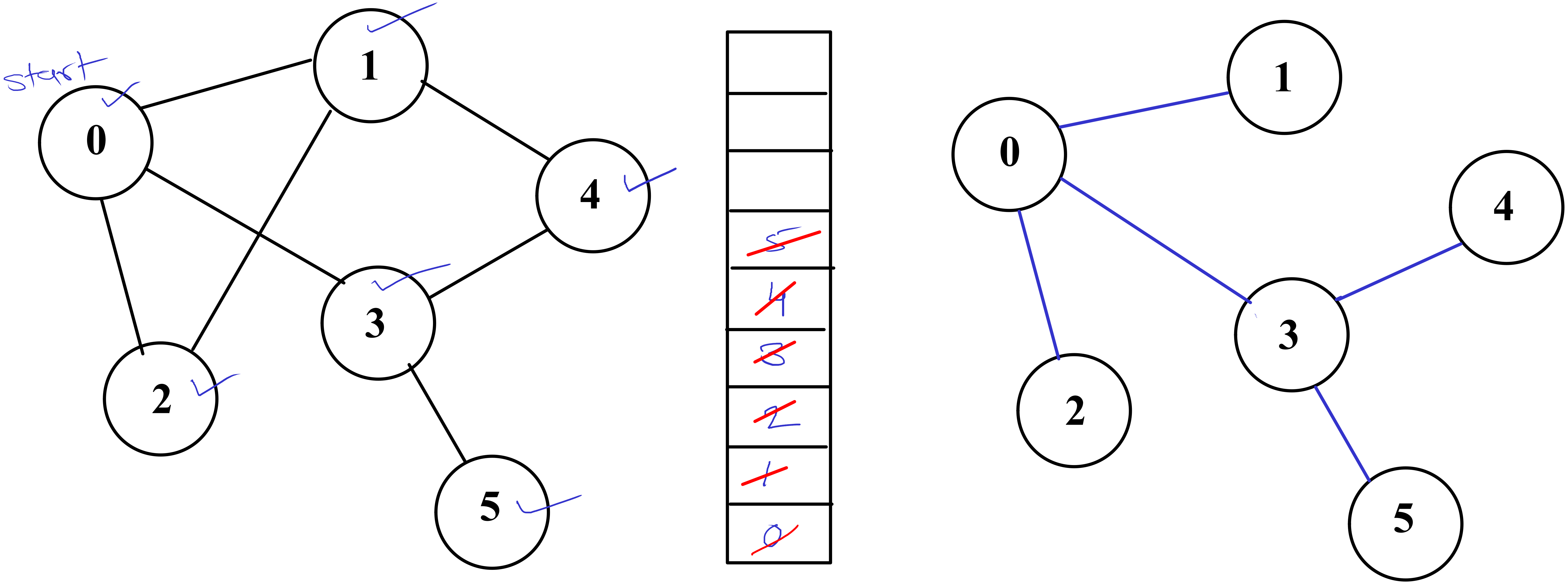
//1. Choose a vertex as start vertex.
//2. Push start vertex on queue & mark it
//3. Pop vertex from queue.
//4. Print the vertex.
//5. Put all non-visited neighbours of the vertex
//on the queue and mark them.
//6. Repeat 3-5 until queue is empty.

# 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

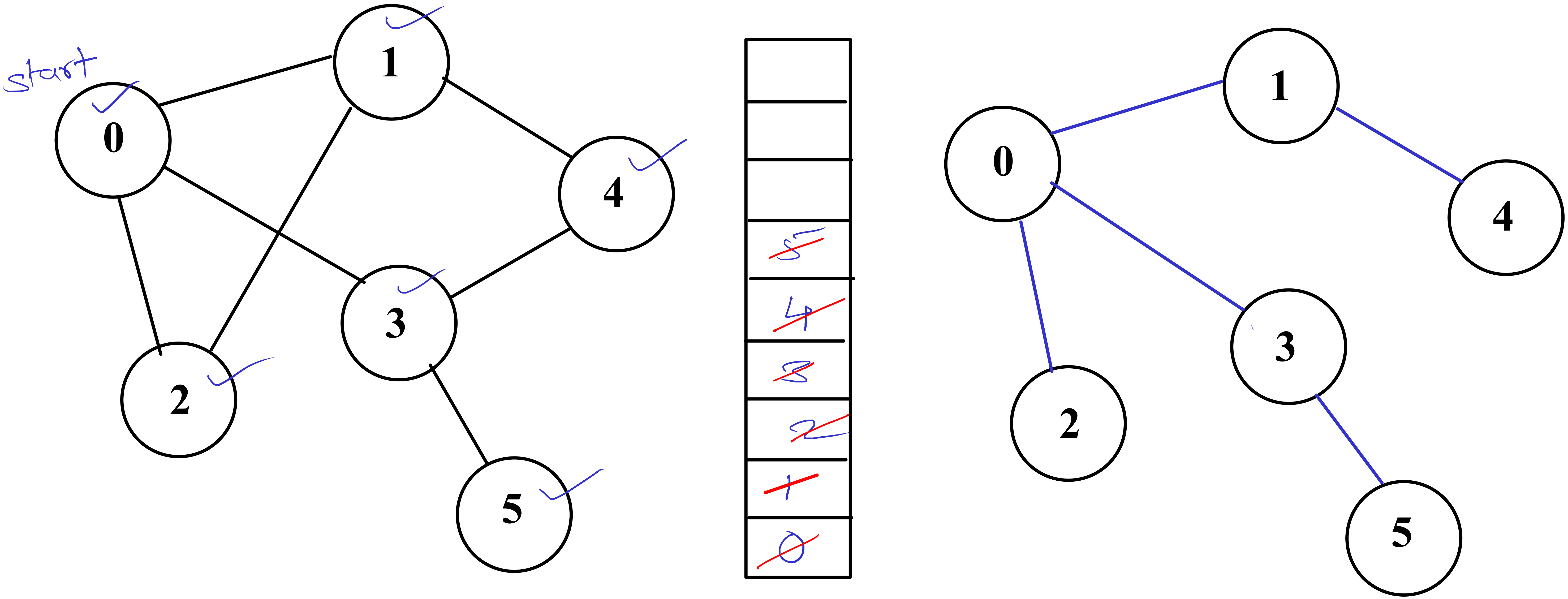


# DFS Spanning Tree



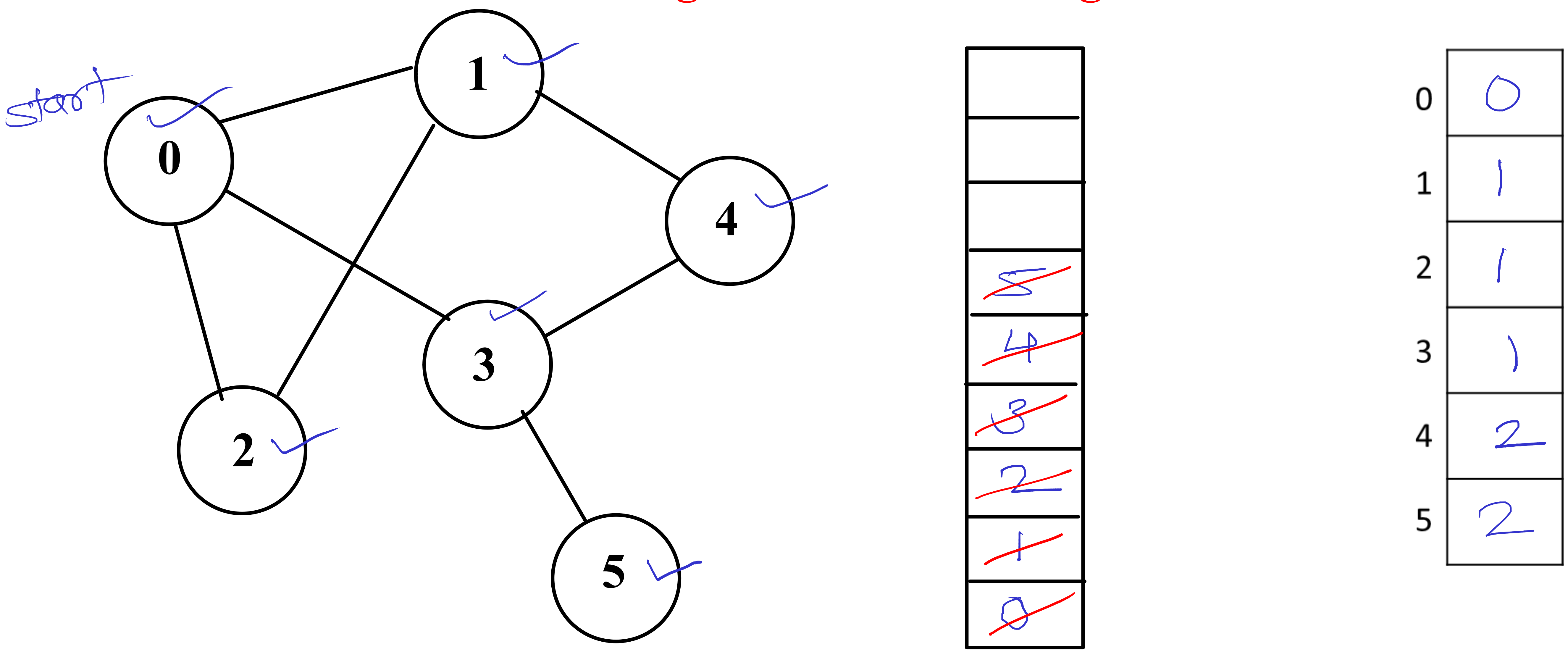

**//1. push starting vertex on stack & mark it.**
**//2. pop the vertex.**
**//3. push all its non-marked neighbors on the stack, mark them.**
**//Also print the vertex to neighboring vertex edges.**
**4. repeat steps 2-3 until stack is empty.**

# 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.**
**//Also print the vertex to neighboring vertex edges.**
**//4. repeat steps 2-3 until queue is empty.**

# Single Source Path length




//1. Create path length array to keep distance 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 distance as dist[neighbor] = dist[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.