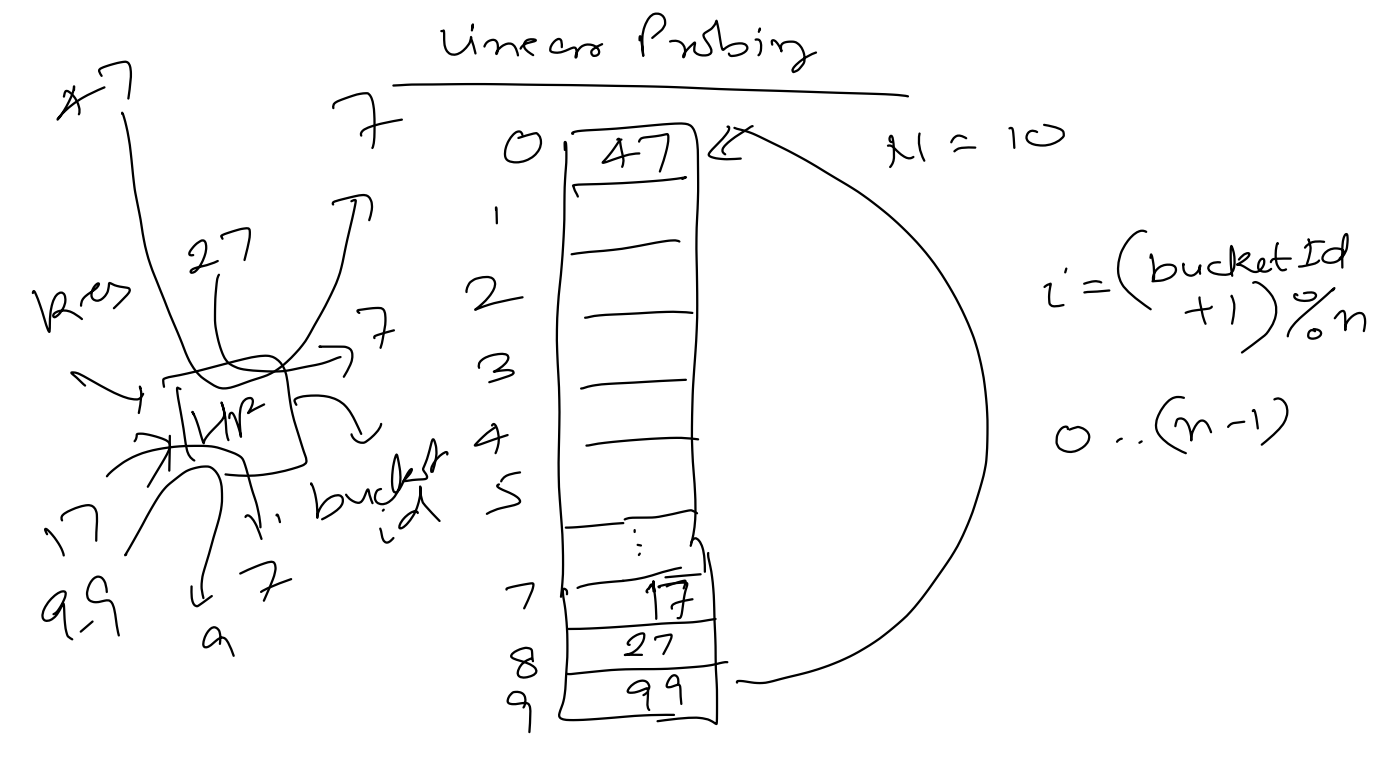

Linear Probing
47
7
27
99
17
HF
bucket id
0
1
2
3
4
5
7
8
9
n1 = 10
i = (bucketId +1) % n
0 .. (n-1)

5 → add ⇒ HF ⇒ 5

10 → add ⇒ HF ⇒ 0

4 → search ⇒ HF ⇒ 4

10 → search ⇒ HF ⇒ 0

15 → add ⇒ 5

15 → search ⇒ 5

25 → search ⇒ 5

N = 10

36

0	10
1	36
2	
3	
4	
5	5
6	15
7	25
8	35
9	45

Linear Probing

$id = (hf(key) + i) \text{ MOD } N$

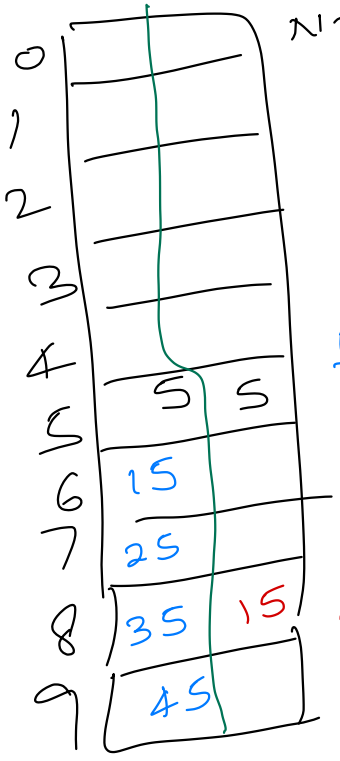
Clustering

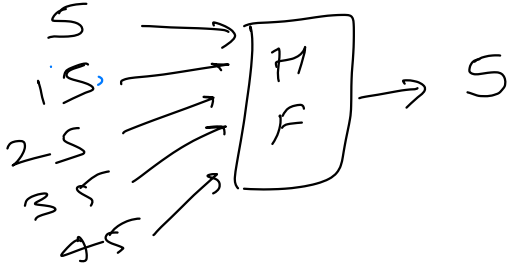
Quadratic Probing

$id = (hf(key) + c\,i + d\,i^2) \text{ MOD } N$

c = 1

d = 2





k → 25
i → ~~1~~ ~~2~~ 3
$5 + 1\times1 + 2\times1^2$
= 8
$5 + 1\times2 + 2\times2^2$
$= 5 + 2 + 8$
= 15 MOD 10
= 5

Linear Probing
$id = (hf(k) + i) \text{ MOD } N$

	15	25	35	45
k →				
i → 1		~~1~~ 2	~~1~~ ~~2~~ 3	~~1~~ ~~2~~ ~~3~~ 4

Quadratic Probing
$id = (hf(k) + ci + di^2) \text{ MOD } N$
c → 1
d → 2
$5 + 1\times1 + 2\times1^2 = 5 + 1 + 2 = 8$
p → 15
i → 1

Storing N elements for SEARCH

(1) Use Search Tree
Time Complexity of Search = $O(\log n)$
—"— of insert = $O(\log n)$

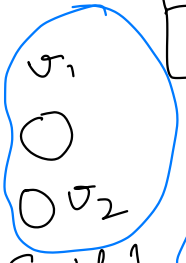
(2) Hash Table with collision handling using chaining & each bucket is a Search Tree
Bucket Size(k) is very small as compared to hash table size (N)
Time Complexity of insert = $O(\log k)$
—"— search = $O(\log k)$

GRAPH

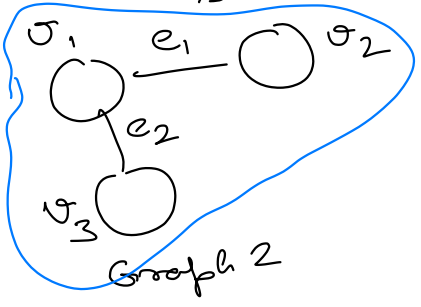
→ What is a GRAPH

- Data Structure
- Non-linear
- $G = \{V, E\}$ → a set of edges / links / lines
  - V → a non-empty set of Vertices / Nodes / objects

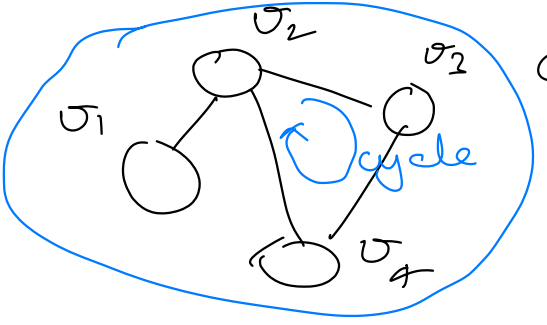
Every tree is a graph but every graph is not a tree



Graph 1

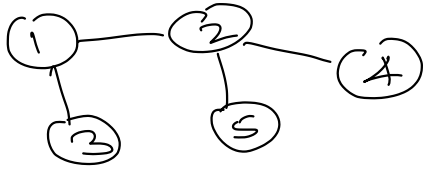


Graph 2

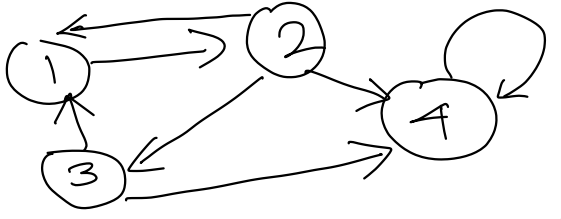


Graph 3

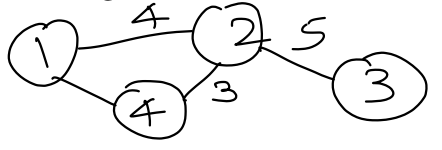
Undirected Graph : Edge do not have directions.



Directed Graph : each edge has a direction



weighted graph : each edge has a weight



Application of Graph

→ Graph is used to represent a "flow"

→ Store map information.

→ Social media : LinkedIn, Facebook, Instagram, etc...
(Network)

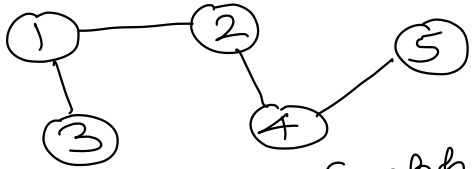
→ Operating System : Job Scheduling.
: Resource Alloc.
: Dead lock detection

How to store graph in memory.

1. Adjacency Matrix

It's a 2D array of size $|V| \times |V|$ (Size of set V)

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

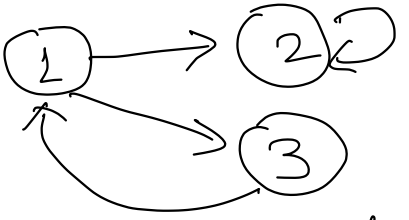


Undirected Graph

2D array of size 5 x 5

Each cell $(i, j) = 1$, if there is a edge between vertex i & j, $= 0$, otherwise

For undirected graph adj matrix is a mirror image around main diagonal.



Directed Graph

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

Weighted Graph.

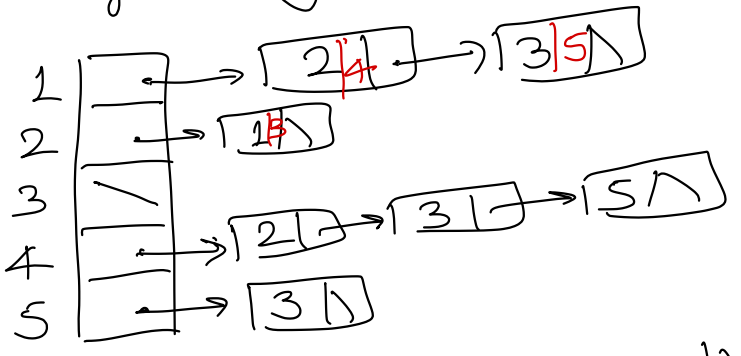
cell (i, j) = weight of the edge connecting vertex i & vertex j

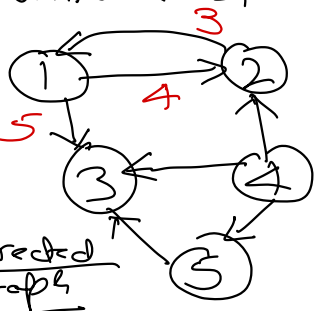
→ Issue $|E| < |V|^2$

$|E|$ → no of edges.

$|V|^2$ → no of cells in adj matrix

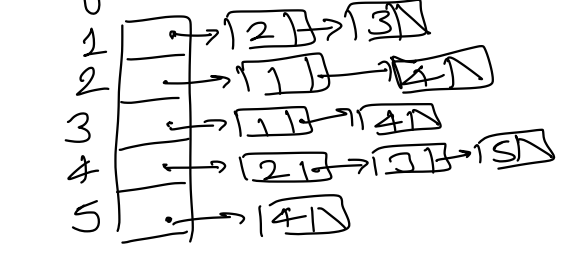
Adjacency list : A array of linked list

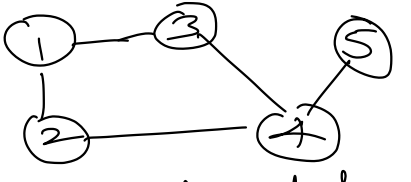




Directed Graph

Adj list is a array of |V| elements.





Undirected Graph

Weighted Graph

List node also store edge weight

Why is adj matrix inefficient.

$$|E| < |V|^2$$

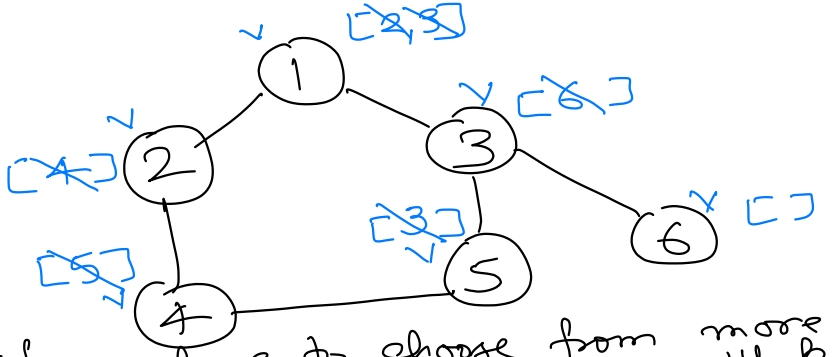
No of cells in adj matrix = $|V|^2$

$5 \times 5 = 25$ cells.

$|E| = 5$

Graph Traversal

- DFS — Depth First Search
- BFS — Breadth First Search



If we have to choose from more than one vertex, we will pick the smaller no vertex first.

Start vertex : 1

DFS : 1 2 4 5 3 6

DFS → for a vertex, visit one of its adjacent vertex and repeat the process until we reach a vertex with no more adj' vertex. then we backtrack.

dfs(v)

1. Mark every vertex as not visited.
2. Start traversal for vertex v.
   // dfs_helper(v)

dfs_helper(v)

1. If v is visited then STOP
   Mark v as visited
2. Find all adj' vertex to v that are not visited.
3. For each adj' vertex (w), dfs_helper(w).

BFS