Hash Table

$O(n^2) - O(\log n) - O(k \log_k n)$

$O(\log k \times \log_k n)$

Multiway Search Tree of Order k

we search in a node using binary search.

Hash Table lets us search in $O(1)$.

→ Bucket : Place in hash table where data is stored.

→ Hash Table : A collection of buckets.

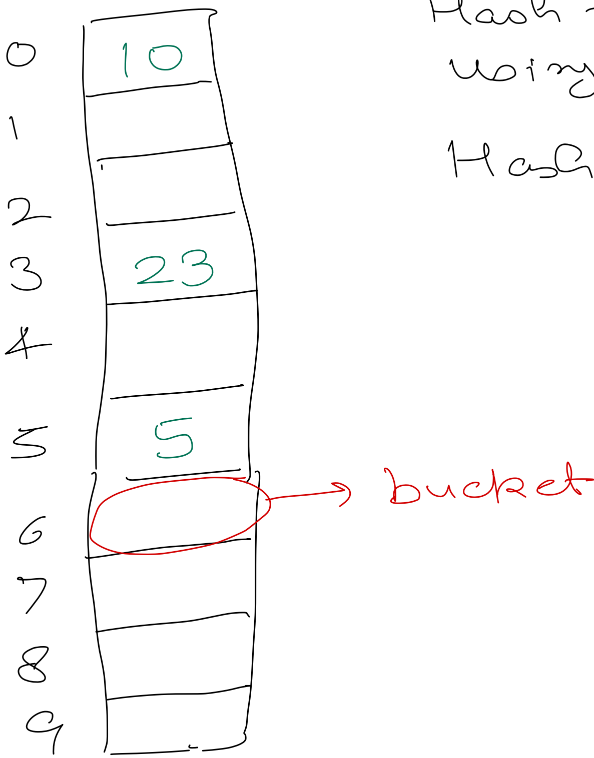
→ Hash function : It is a function ~~that~~ uniquely maps an key/element to a bucket.

Ideal case

Hash table implemented using array.

Hash Function : MOD N

Size of hashtable.



N = 10

Size of hash table

Add (5)

→ Find bucket id using hash function

→ Store element in bucket.

Add (10)

Add (23)

Search (5)

FOUND

Search (6)

NOT FOUND

→ Find bucket id using hash function

→ Is element present in bucket?

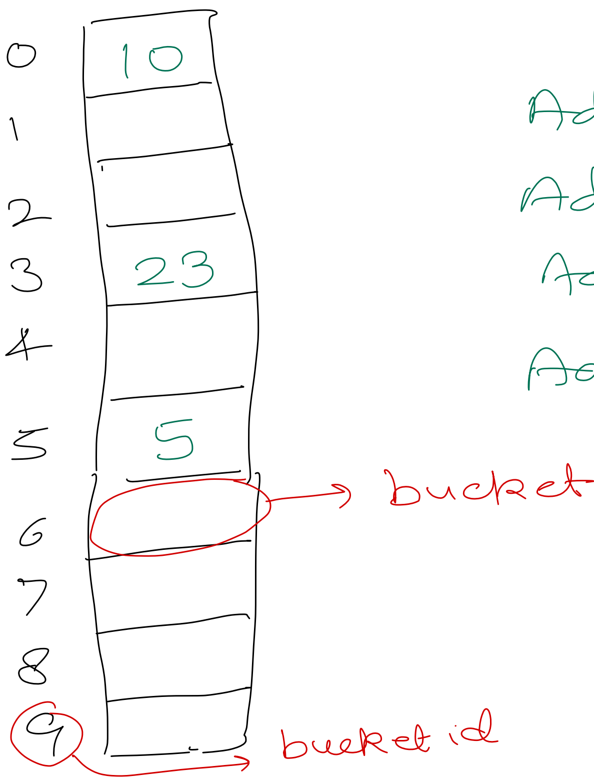
HashFunction(element) => O(1)
- BucketId = element MOD N → 1
- Return BucketId → 1

N = Size of Hash Table.

Add(element) => O(1)
- BucketId = HashFunction(element) → O(1)
- Store element in bucket with id as BucketId. → 1

Search) element) => O(1)
- BucketId = HashFunction(element) → O(1)
- if element present in bucket with id BucketId then → 1
  - Element found → 1

  Else
  - Element not found. → 1



N = 10

Size of hash table

Keys			Bucket id
Add(5):	5	→ HF →	5
Add(10):	10	→ HF →	0
Add(23):	23	→ HF →	3
Add(3):	3	→ HF →	3

Collision
↓
when different Keys are mapped to same bucket id by hash function.

Collision Handling using different Hash Functions

→ Reduce Chance of Collision

1. Use different Hash Function.
   - → MOD N.
   - → Folding : Divide key into multiple parts & combine them.

$$\begin{array}{r} 23 \\ \downarrow \\ 2 \\ +\ 3 \\ \hline 5 \end{array} \qquad \begin{array}{r} 53 \\ \downarrow \\ 5 \\ +\ 3 \\ \hline 8 \end{array}$$

Divide key into multiple parts

Combine multiple parts.

$$\begin{array}{r} 35 \\ \downarrow \\ 3 \\ +\ 5 \\ \hline 8 \end{array}$$

   - → Mid-Square

23	53	35
↓	↓	↓
5$\boxed{2}$9	2$\boxed{80}$9	1$\boxed{22}$5
↓	↓	↓
2	80	22
↓	↓	↓
2	0	2

Middle of square of key

⇓ Map these to bucket id range. 0..9

Other Collision Handling Techniques

→ Chaining : Each bucket can store multiple keys.

↳ Each bucket is an array of keys.

→ Each bucket is a list.

Search Tree
- Insert $O(\log k)$
- Search $O(\log k)$

Insert $O(k)$

Search $O(k)$

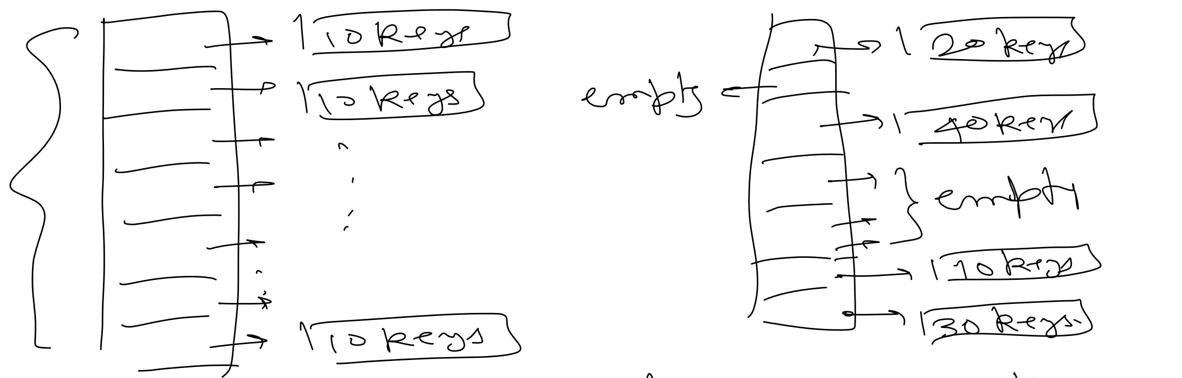
Insert to keep array sorted $O(k)$

Search in sorted array $O(\log \underline{k})$ → Bucket size.

$n = 100{,}000$

$k = 100$ $\Leftarrow$ $k \underline{<<} n$ Very small

What we want | What might happen



10 buckets & store 100 keys => ~10 keys per bucket.

But based on actual keys, more keys are mapped to a few buckets.

⇓

Buckets space not efficiently used, as bucket is of fixed size.

→ Probing.
- → Linear Probing.
- → Quadratic Probing.

Key → HF → bucket id

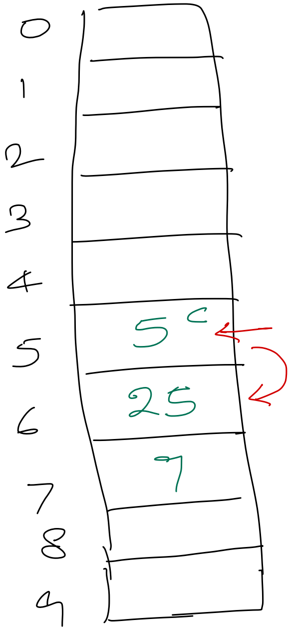
5 → HF → 5

7 → HF → 7

25 → HF → 5 ⇒ Collision

⇓

Probe for an empty bucket to store this key.



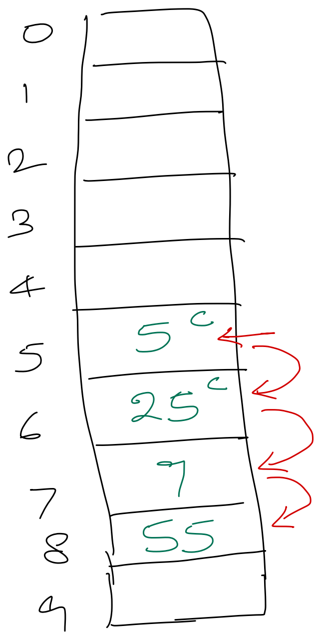
Linear Probing

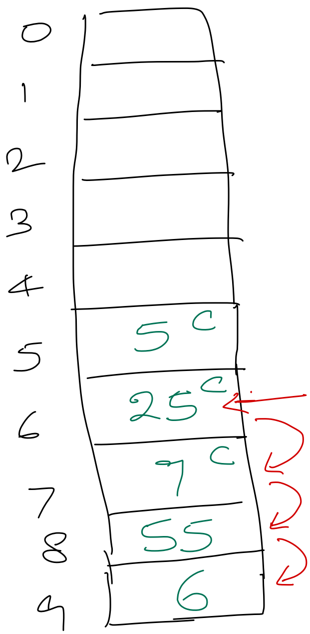
look at next bucket. Empty?

55 → HF → 5

⇓

Collision





$6 \rightarrow \boxed{HF} \rightarrow 6$

$\Downarrow$

Collision

Search (15)

$15 \rightarrow \boxed{HF} \rightarrow 5$

Static Hash Table

↓

Symbol Table

- → Keywords
- → identifiers

Problem with linear Probing

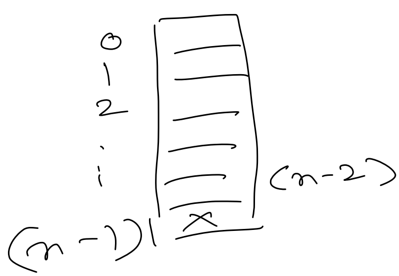
$\Downarrow$

clustering.

Linear Probing

$$\underbrace{(hf(key) + i) \bmod N}_{\text{bucketId}}$$

$i = 0 \ldots (N-2)$



Quadratic Probing

$$\left(hf(key) + \boxed{ci + di^2}\right) \bmod n$$

Some quadratic equation

$i \rightarrow 0 \ldots (n-2)$

$c \rightarrow 1$

$d \rightarrow 2$

LinearProbing(key)
- hf = HashFunction(key)
- for i = 0 to (n - 1) do
  - BucketId = (hf + i) Mod n
  - If bucket with BucketId is empty then
    - Stop linear probing with BucketId as result

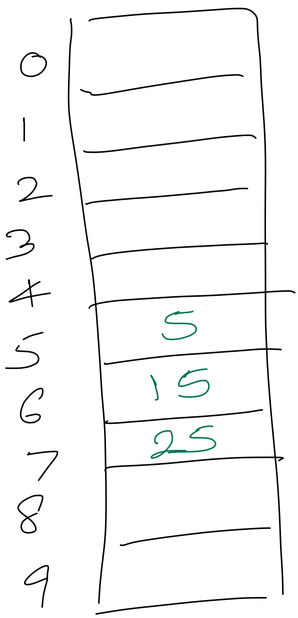
after last bucket comes first bucket.

$c = 1$

$d = 2$

$hf + ci + di^2$

$hf + i + 2i^2$



Linear Probing

$5 \rightarrow \boxed{HF} \rightarrow 5$

$15 \rightarrow \boxed{HF} \rightarrow 5$

Q.P

$i \rightarrow 0, 1, 2 ..$

$0: 5 + 0 + 0 = 5$

$1: 5 + 1 + 2 = 8$

$25 \rightarrow \boxed{HF} \rightarrow 5$

Q.P

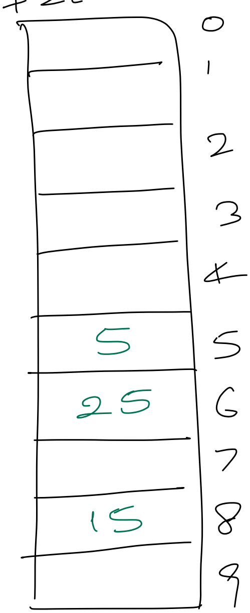
$i \rightarrow 0, 1, 2 \ldots$

$0: 5 + 0 + 0 = 5$

$1: 5 + 1 + 2 = 8$

$2: 5 + 2 + 2 \times 4 = 15 = 5$

$3: 5 + 3 + 2 \times 9 = 26 = 6$



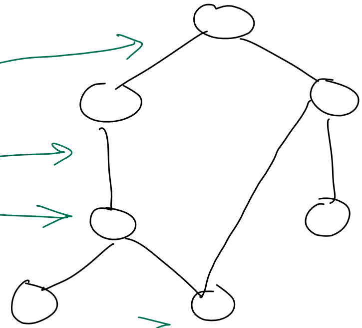
Quadratic Probing.

GRAPH

$G = \{V, E\}$

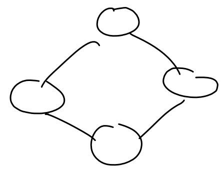
Edges

Vertices

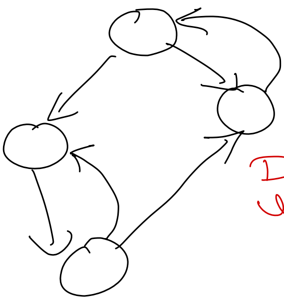


Undirected graph → each edge do not have direction

Directed graph → each edge has a direction



Undirected Un weighted Graph



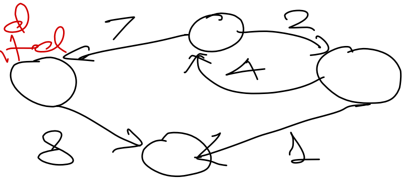
Directed Unweighted Graph

weighted graph → each edge has a weight associated with it

un-weighted graph → each edge has no weight assigned to it

Undirected weighted Graph

Directed weighted



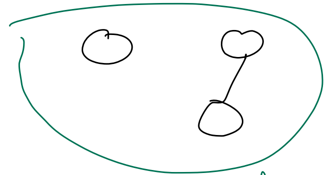
→ Social Media

→ Maps

→ Job Scheduling

Resource Allocation

Deadlock detection

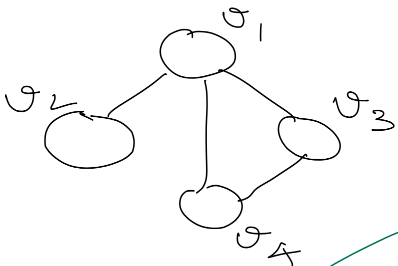


Graph disjoint

Storing Graph

→ Adj Matrix ⇒ 2D array of size $N \times N$

N ↳ Number of Vertex in Graph.



Adj List ⇓

- v_1 → v_2 → v_3 → v_4
- v_2 → v_1
- v_3 → v_1 → v_4
- v_4 → v_1 → v_3

For undirected graph, main diagonal is all 0's

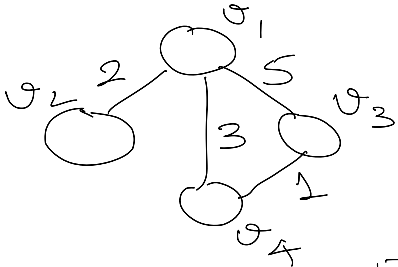
upper triangle is a mirror image of lower triangle for undirected graph.

	v_1	v_2	v_3	v_4
v_1	0	1	1	1
v_2	1	0	0	0
v_3	1	0	0	1
v_4	1	0	1	0

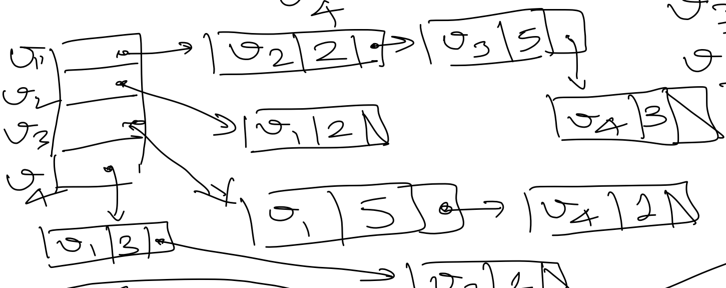
⇐ Adj Matrix

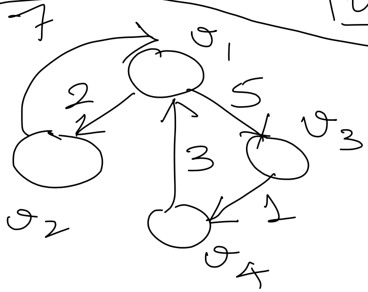
(i, j) cell will $\boxed{\text{be } 1}$, if there is a edge between v_i & v_j

$\boxed{\text{be weight}}$ in weighted graph.



	v1	v2	v3	v4
v1	0	2	5	3
v2	2	0	0	0
v3	5	0	0	1
v4	3	0	1	0





	v1	v2	v3	v4
v1	0	2	5	0
v2	7	0	0	0
v3	0	0	0	1
v4	3	0	0	0

Adj Matrix: we can have lots of 0's as no edge between two vertices.

⇒ waste of memory.

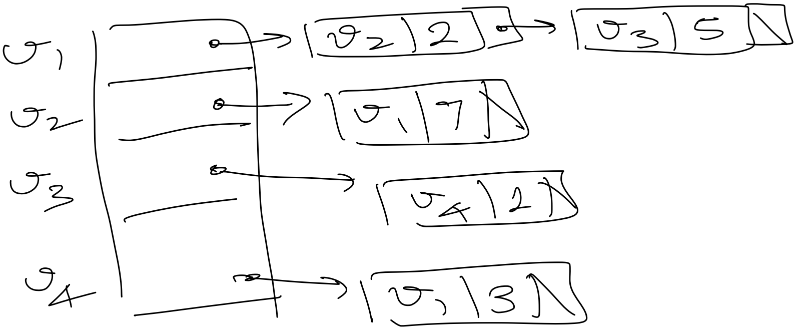
⇓

Use Sparse array.

→ Adj List: An array of linked list

⇓

(N)



→ Adj Multi List