



**#ASLI ENGINEERING**

# Linear Probing in Hash Tables

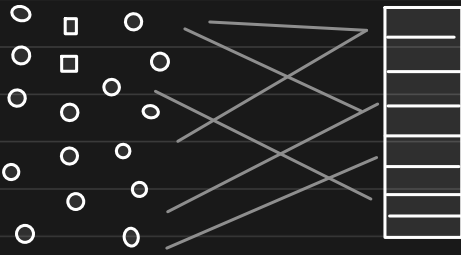


**BY**

**ARPIT BHAYANI**

# Conflict Resolution with Linear Probing

Conflicts are inevitable!



With Open Addressing, we use a probing function to find the slot where the key should be placed

One such method is linear Probing

## Probing Function

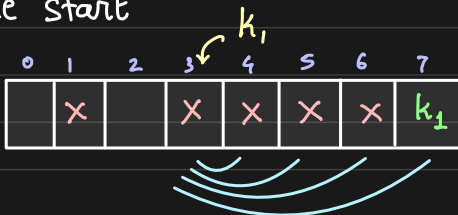
Probing Function is defined as  $p(k, i) = j \leftarrow \text{index}$

key  $\rightarrow$  attempt

we use the probing function to find the first available slot  
The same function is used during lookups

Linear Probing  $p(k, i) = (h(k) + i) \% m$

We search linearly from the hashed index until the end of the table and then wrap from the start

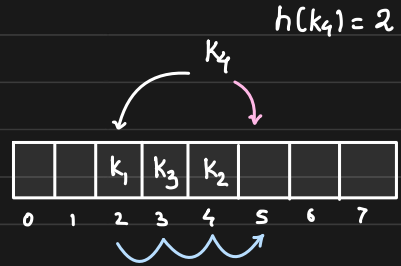


## Hash Table Operations: Adding a key

We invoke the probing function to find the slot in the hash table.

if that slot is occupied, we traverse the hash table and find first available slot.

It is like a linear search from the slot



## Hash Table Operations: Key Lookup

We invoke the probing function to get the slot.

if key present at that index, we **return**

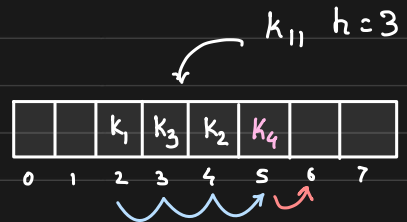
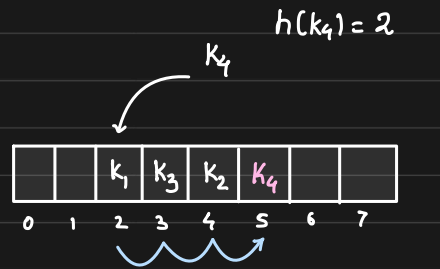
if not, we traverse to the right and

try to find the key.

if we encounter the key, we **return**

if we reach the end, we start from index 0

we stop iteration as soon as we encounter an empty slot.

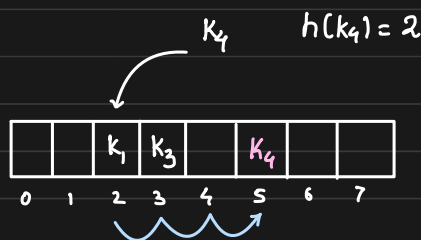


Worst case: Linear lookup across entire table

## Hash Table Operations : Deleting a key

Say, DEL  $K_2$  then GET  $K_4$

Delete is a soft delete so that we could continue reaching to the keys slotted further

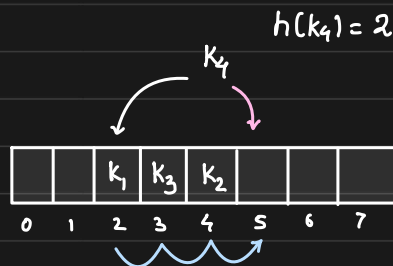


Linear probing is simple and fast

Simple: we literally linearly iterate to find the next slot

Fast: isn't linear traversing slow?

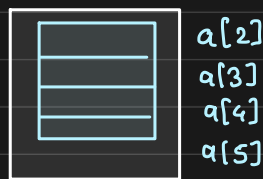
Not really! because we leverage *localized access*.



When we access a memory location,

the page is cached on CPU and the page contains neighbouring elements.

CPU cache



So, subsequent accesses are served from CPU cache

Linear probing gives a constant time performance

Worst case is still painful

↳ avg case = few collisions

## Challenges with Linear Probing

1. A bad hash function would make linear probing a full hash table search and hence inefficient

Hence, using a good hash function is very important

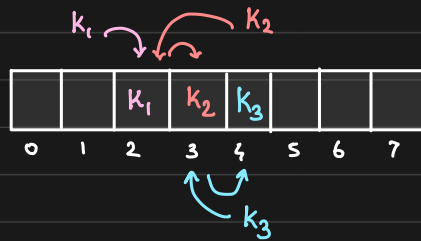
Murmurhash is preferred

2. Linear probing suffers from clustered collisions

if  $k_1$  hashes to 2

$k_2$  hashes to 2

$k_3$  hashes to 3



Because  $k_1$  and  $k_2$  collided,

$k_3$ 's primary slot got occupied!

and this would also impact keys hashed to subsequent locations

Hence a good **uniform** hash function is essential for

linear probing to be efficient.