Unit - 3

Storage strategies

Indices :

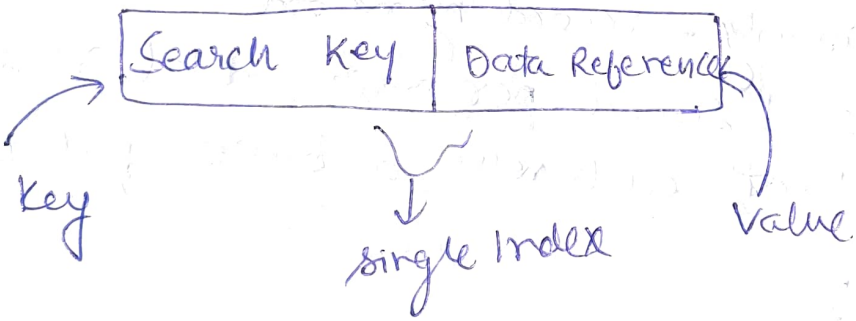
→ Indexing in DBMS is used to speed up data retrieval by minimizing disk scans.

→ Instead of searching through all rows, the DBMS uses index structures to quickly locate data using key values.

→ When an index is created, it stores sorted key values and pointers to actual data rows.

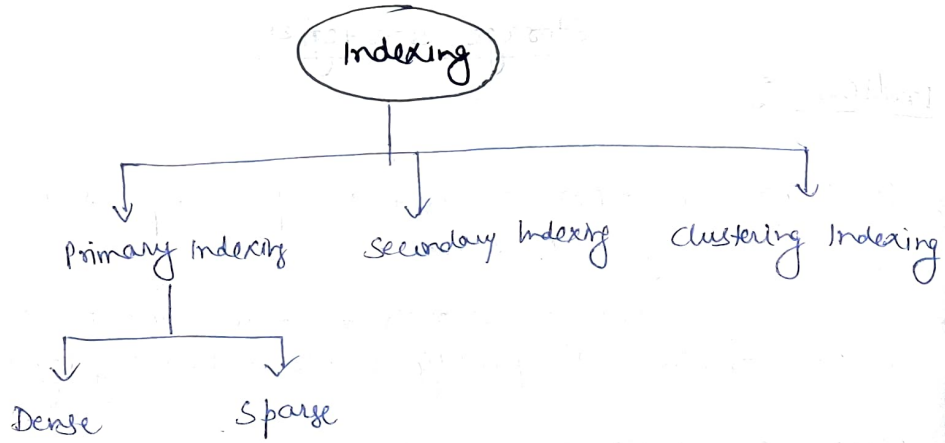
→ This reduces the no. of disk accesses, improving performance especially on large datasets.

Structure of Index in DB



Attributes of Indexing :

1. Access Types – types of access, such as value based search, range ~~based~~ access.
2. Access Time
3. Insertion Time
4. Deletion time
5. space overhead – additional space required by the Index.



How Indexing makes DB faster?

Let us takes an example we have 32 records of data:

Without Indexing:

32 rows of data
Searching done linearly – record by record
Suppose it takes 1 second per row, total 32 sec, and it will treat each row a block of data.

With Indexing:

→ In indexing, storing is done in sorted order based on index values.

→ Let's imagine, 32 rows grouped into blocks, 8 rows per block → total 4 blocks

Index will store just one entry per block.

Block 1 → key 1
Block 2 → key 9
Block 3 → key 17
Block 4 → key 25

so instead of 32 seconds, it might take $\log_2(32)$ or less
It reduces no. of disk read.

Ex:

age	id
21	2
22	3
22	5
23	1
23	4
24	6

Each entry in index is
4B + 4B = 8B

There will be 100 entries in the index, so total size of index will be 8 x 100 = 800B

1 disk block = 600B

for 800B = 2 disk block

Query: find all users with age == 23 with index

Flow:

↳ 1) iterate index
  ↳ block by block
2) check age == 23 in entries
3) if yes, add the 'id' in a buffer
4) if no, discard

} Read the index not the relevant ids matching criteria

5) for all the relevant id in the buffer
  ↳ read the records from the disk
  ↳ add an o/p buffer

} for the relevant ids fetch the actual records from the disk.

6) return the output buffer

Advantages of Indexing :

1. Faster Queries
2. Efficient Access
3. Improved sorting
4. Consistent performance
5. Data Integrity

Disadvantages :

1. Increased storage space
2. Increased maintenance overhead
3. Slower insert/update operations
4. Complexity in choosing the right index

B- Trees

→ Specialized m-way tree designed to optimize data access, especially on disk based storage.

→ In a B-tree of order m, each node can have upto m children and m-1 keys, allowing it to efficiently manage large datasets.

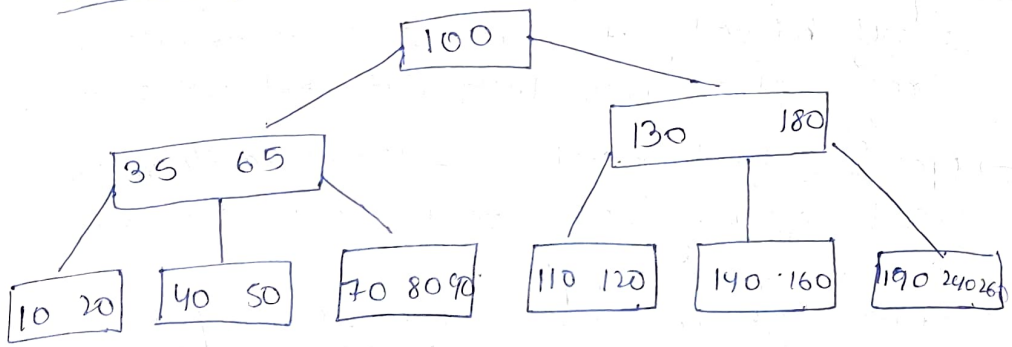
→ Value of m is decided based on disk block and key sizes.

→ Standout feature of b-tree :

1) Ability to store a significant number of keys within a single node, including large key values.

Example :

B- Tree of order 5



Properties of B-Tree :

1. All the leaf node of a Btree are at the same level i.e. they have the same depth (height of tree)
2. The keys of each node of a B tree, should be stored in ascending order.
3. All non-leaf nodes (except root node) should have at least $m/2 - 1$ keys.
5. if root node is a leaf node = no children
   Atleast one key
   if root node is non-leaf node = 2 children (atleast)
   Atleast one key
6. A non-leaf node with n-1 key values should have n non NULL children.

→ Height when the B-trees is completly full

$$h_{min} = \lceil \log (n+1) \rceil - 1$$

→ Height when the B-trees is least filled

$$h_{max} = \lfloor \log_t \left(\frac{n+1}{2} \right) \rfloor$$

Need of a B-tree :

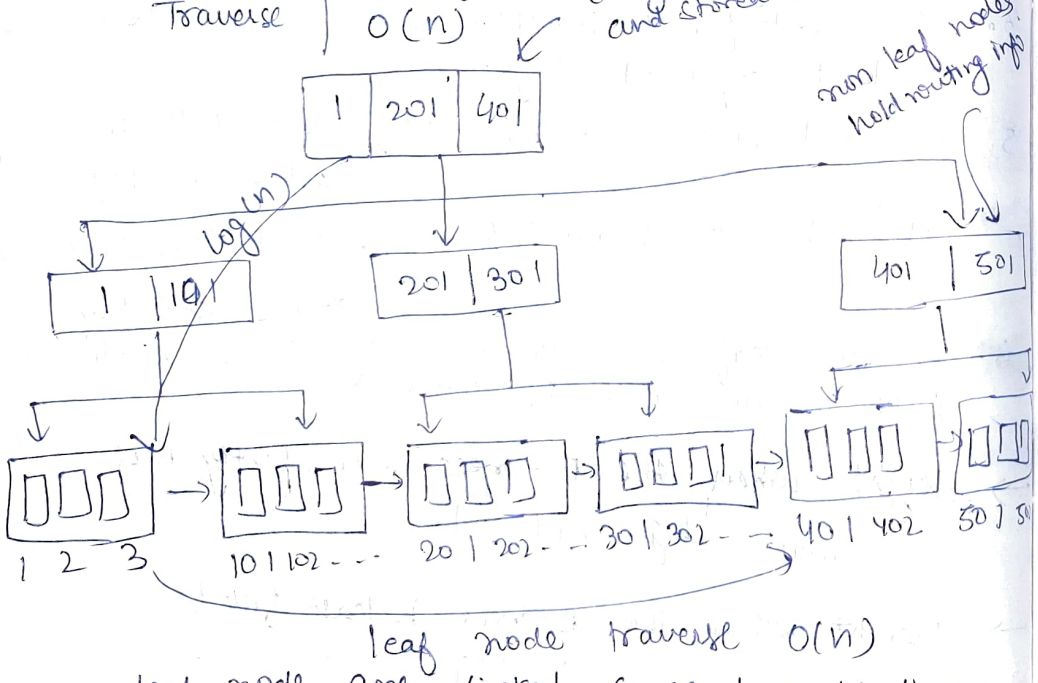
→ Improved performance over M-way trees, because it is self balanced, it makes suitable for external storage systems where fast data retrieval is crucial.

→ Optimized for large datasets - it can handle millions of record efficiently. Their reduced height and balanced structure make it happen.

→ It has high concurrency & high-throughput
↳ fast & scalable in heavy load

Operations on B-Tree

Operation	Time complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)
Traverse	O(n)



leaf node are linked so as to enable linear traversal.

Facts :

→ If 1 B+ tree node is 4000 B big and row/document size is 40B then each node = 100 rows (max)

→ Size of B+ tree node ≃ disk block size
↳ in one disk read we read 1 node ≃ 100 rows

Q: Why we we we required to use B-tree?

⇒ Say our table/records are stored in one file sequentially.

Insert - Cannot efficiently insert in the middle
O(n) - TC

Update - Update, can override within the same width. O(N) - TC

Find one - linear scan . O(n) - TC

Range Queries - Possible only when rows are ordered by it. O(N) - TC

Delete - Create a new file without that entry/row.
O(N) - TC

Hashing

Internal structure of Hash Table:

Hash Tables → one of the most widely used data structure

→ Hash tables are also used as building blocks for constructing: 1) classes and its members
2) variable lookup tables

→ Hash Tables are designed to provide constant time → insertions
deletion
lookups

Key → Value

apple	5
banana	15
cat	18
dog	3

Two ideas to construct Hash Table:

1. Application key to hash key [0, N]
   apple → 12762179
2. Hash key to a smaller range
   12762179 → 17

Application Keys to Hash Keys

→ we cannot put anything as a key in hash table.
→ it is limited to a specific set of types
e.g. string, int, tuple etc.
→ we can also use custom types as keys if they implement the 'hash' function that spits out an integer for the object.

* For some native types, the hash function is internally implemented.

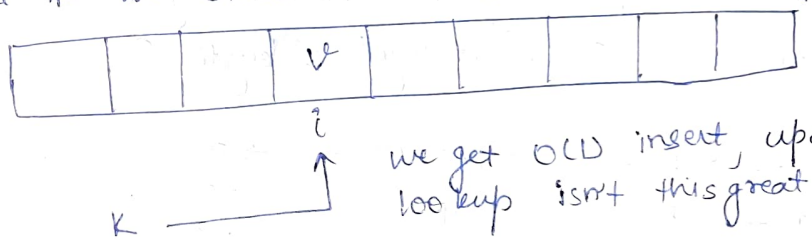
Hash keys have a larger range say $[0, 2^{32}]$ and a hash function converts the object to this int range.

apple → f → 12762179
banana → f → 51962
cat → f → 62
dog → f → 1962719

String → Integer

Naive Implementation

Because we are already getting int (i) from key (k), what if we store the value (v) at index i?



we get O(1) insert, update & lookup isn't this great?

This approach works well, only when N is small

space required for holding array when

N = 10 → 4 x 10 ≃ 40 B
N = 100 → 4 x 100 ≃ 400 B
N = 1000 → 4 x 1000 ≃ 4 KB
N = 1m → 4 x 1m ≃ 4 MB
if N = int32 range → 4 x 4 billion ≃ 16 GB

Challenges:

1. finding this big chunk of memory is tough
2. lots of slot would remain empty

Mapping hash keys to smaller range

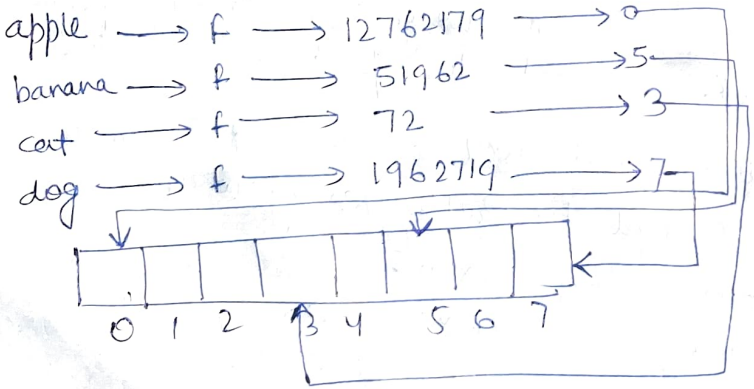
→ if we are planning to store K keys in the hash table, we have to have an array of size m, such that

$$m \in O(k)$$

This requires us to have a second step that reduces hash key from range to smaller bin range

$$[0, N] \longrightarrow [0, m)$$

Say, we are planning to store 4 keys in hash table we can have a bin [array] of length 8



Adding more keys

if we add more keys to our hash table, the holding array to be resized.

$$m \longrightarrow 2m$$

Why are even hashing stuff into int?

The first step is simplifying our problem statement for the second step, making it easier to optimize int → int distribution.

The first step also allows us to give great abstraction

Object → INT32

enabling us to support complex data types as key.

conflicts

Key1 → f → h4

Key 2 → f → h4

multiple keys can produce same hash key upon hashing.

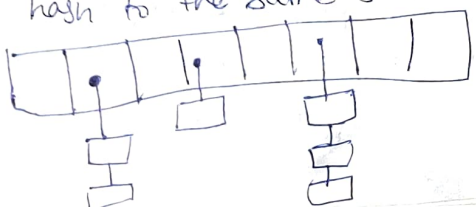
So, how can we store multiple keys in the same slot?

The two classical ways of achieving this

1. Chaining
2. Open Addressing

Core idea:

from a chain of keys that hash to the same slot

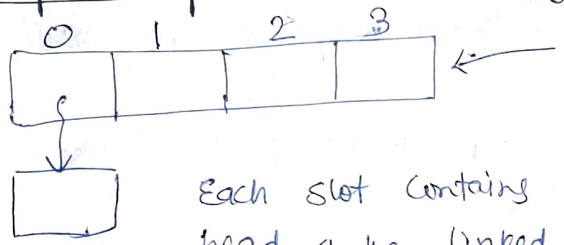


Chaining :

We put colliding keys in a data structure that hold them well.

most common implementation → LinkedList

Simplest Implementation : Singly linked list



Each slot containing the pointer to the head of the linked list.

```
Struct slot {
    Struct node * head;
}
```

→ Each node of the list contains :

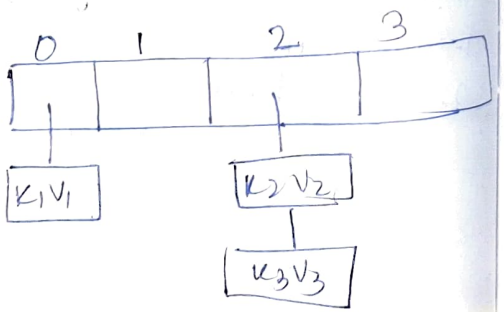
1. pointer to the actual key
2. pointer to the next node of the list

```
Struct {
    Void * key;
    Struct node * next;
}
```

Hash Table Operations

1. Adding a key

put (K_1, V_1)
put (K_2, V_2)
put (K_3, V_3)



Given a key and a value, we

1. Pass the key through the hash function and get index i.
2. Create a new linked list node with K, V
3. Add it to the chain present at index i

Possible implementations

1. Insertion can always happen at the head → fast
2. Insertion can always happen at the tail → fast
3. Insertion can happen as per the sort order → Linear iteration

2. **Delete a key**

Delete K_2, K_1, K_3

Delete operation is simple

1. Reach the slot in O(1)
2. Iterate through the list and find node → key == K_i
3. While iterating keep track of prev node
4. Adjust the pointers
5. Delete the intended node

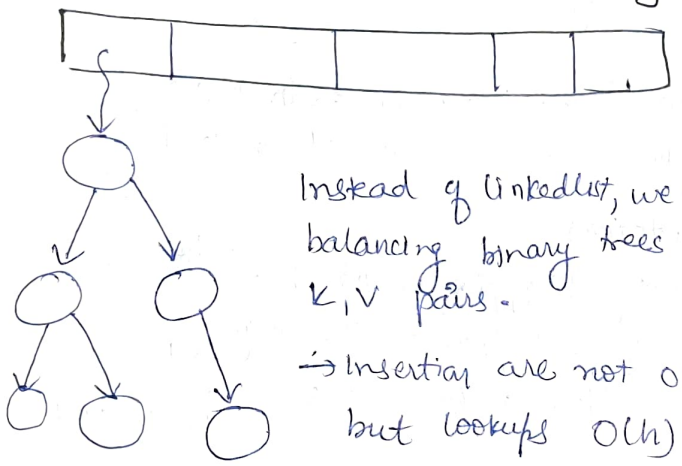
3. **Lookup a key**

GET K_1, K_2, K_3

Lookups similar to delete

1. reach the node in O(1)
2. Linearly iterate through the list until we find node → key == K_i

Other data structure for chaining

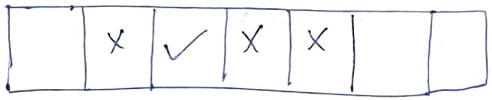


Instead of LinkedList, we can use a self balancing binary trees to store collided K,V pairs.

→ Insertion are not O(1) but lookups O(h)

Open Addressing :

when keys collide, find a way to hunt for an available slot in the array — deterministically.



Probing : finding the next available slot called probing

Probing strategy can be defined as

$j = p(K, i)$ that spits out the new index where key K can be placed in attempt i.

$j \in [0, m]$ $i \in [0, m]$

m = size of hash table

Hence, first insert $j = p(K, 0)$, if that is occupied we try

$j = p(K, 1)$, if that is occupied we try

$j = p(K, 2)$, if that is occupied we try

⋮

$j = p(K, m-1)$, if that is occupied we try

Good Probing function

→ The probing function should generate the permutation of numbers $[0, m-1]$ so as to cover the entire space eventually.

Implementing probing function

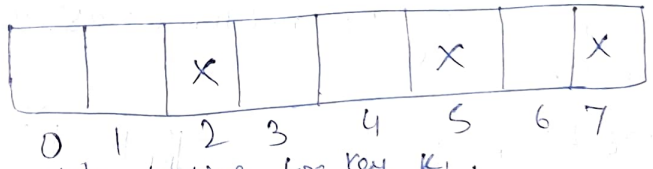
→ It is a simple mathematical or algorithmic function that deterministically tells us our next slot for a particular key.

Key attempt

$p(K, 0) \rightarrow 5$ 1st attempt probing func gave 5

$p(K, 1) \rightarrow 7$ 2nd attempt probing func gave 7

$p(K, 2) \rightarrow 2$ 3rd attempt probing func gave 2



So, while looking for key K_1,
we first look at $p(K_1, 0) = 5$
if we can't find, $p(K, 1) = 7$,
If we can't find, $p(K, 2) = 2$

Hash Table operations : Adding a key

→ Until we find a free slot, keep probing and check at the first free slot, put the key.

Hash Table operations: lookup

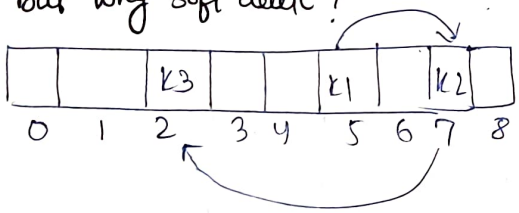
→ Similar to adding. Using probing function we try to find keys in slots.

→ Iterations stops when we find the key or we exhaust iterating over all the slots.

Hash Table operations : Deleting a key

The deletion is a soft delete. we lookup the key using probing function and upon discovering, we mark the slot is deleted.

But why soft delete?

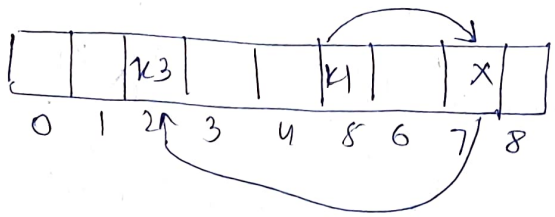


K_1, K_2, K_3 hashed at index 5, with open addressing, say we get them placed at 5, 7 and 2.

Say, we had delete key k_2 and now we are looking for K_3.

DELETE k_2

GET k_3



If you delete K_2, then how would we ever reach K_3?

empty slot == Stop iteration

So, we will never be able to reach K_3 during lookup

Hence, we need to differentiate b/w free and delete.

↓

Soft deletion is the way to go.

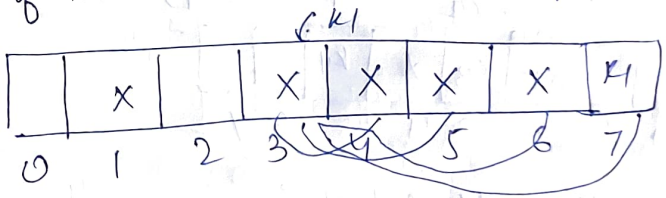
Limitations of open Addressing

Max number of keys = # slots in array

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 operation (Adding a key)

→ we involve 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 linear search from slot

Hash Table Operations: Key lookup

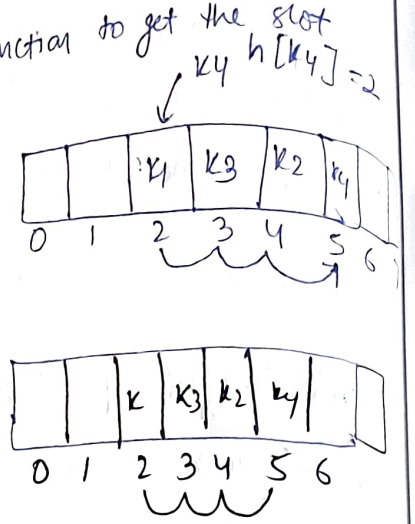
→ We invoke the probing function to get the slot

→ If key present at that index, we return

→ If not, we traverse to the right & try to find the key

→ If we encounter the key, we return

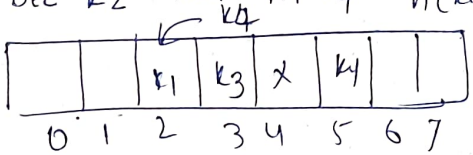
→ If we reach the end, we start from index 0 we stop iteration as soon as we encounter an empty slot



Hash Table Operations: Deleting a key

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

Say, DEL k_2 then GET k_4 $h(k_4) = 2$



Linear Probing is simple & fast:

Simple:
→ We literally iterate to find next slot

fast: isn't it slow?
not really! because we leveraged localized access

→ When we access a memory location, the page is cached on CPU and the page contains neighbouring elements.

So, subsequent accesses are served from CPU cache

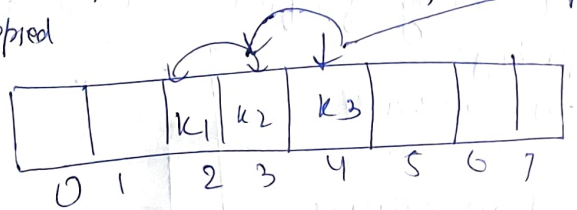
→ It gives a constant time performance.

Challenges:

→ A bad hash function would make linear probing a full hash table search and hence efficient!
Hence, good hash function is very necessary.
Murmurhash is preferred.

→ Linear probing suffer 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



Hence, good uniform hash function is essential for linear probing to be efficient.

Quadratic Probing

→ Instead of placing the collided key in the neighbouring slot, quadratic probing adds successive value of an arbitrary quadratic polynomial.

$p(k, i) = h(k) + i$ $P(k,i) = h(k) + c_1 i + c_2 i^2$

Linear Quadratic

Sample sequence:

$h(k), h(k)+1^2, h(k)+2^2, h(k)+3^2, \ldots$

How it is better than linear probing?

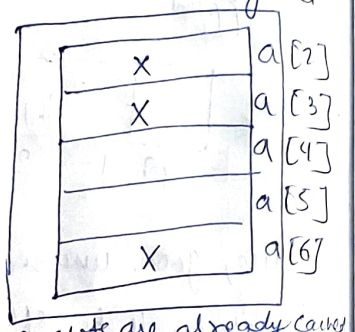
→ It reduces clustering and cascaded collisions as collided keys are placed further away from each other

→ **Properties of Quadratic Probing:**

1. It reduces clustered collision by distributing it quadratically.
   * It is not immune to it, but still it reduces it to a good extent

2. It has a good locality of refrence but not as great as linear probing.
   * leverage CPU cache well

Unless there are large collisions on same key, atleast a couple of subsequent slots will be in CPU cache.



→ Two slots are already cached on CPU.

$h(k) = 2$

$h(k)+1^2 = 3$

$h(k)+2^2 = 6$

Double Hashing

→ Double hashing technique uses 2 hash functions to find the slot.

→ first hash function gives primary slot and upon collision, it uses second has offset times attempt until an empty slot is reached.

$$p(k,i) = (h_1(k) + i * h_2(k)) \bmod m$$

→ Given that we are using another hash function to offset, we are minimizing repeated collisions and effects of clustering.

→ follows no specific pattern and gives near-uniform yet random offset from the index (primary slot)

sequence: $h_1(k), h_1(k)+1\,h_2(k), h_1(k)+2\,h_2(k), \ldots$

→ **Choosing the second hash function**

1. It should never return 0
2. It should cycle through entire table (order doesn't matter)
3. Fast to compute

→ **Advantages:**

1. Uniform spread over collision
2. follows no specific offset pattern, purely dependent on key
3. least prone to clustering problem.
   ⇒ offset from primary slot is uniformly distributed.