Hashing:

What is the problem? Why we tend to use hashing?

Suppose we are storing employee records, the primary key for which is employee's telephone number.

Telephone	Name	City	Dept	...
9864567654	Sam	NYC	HR	
9854354543	Tom	DC	IT	
.....				

We need to perform these operations:

1. **Insert** Employee Record
2. **Search** for an Employee
3. **Delete** Employee Record

Solutions:

Using an **array: search** will be **ok**, but insertion and deletion becomes **costly**.

Using a **Linked List:** insertion and deletion will be **ok**, but **search** becomes **costly**.

Using a **Balanced BST:** insertion takes **O(logn)**, deletion takes **O(logn)**, search takes **O(logn)**.

Creating a Direct Access Table: insertion takes **O(1)**, deletion takes **O(1)**, search takes **O(1)**.

Direct Access Table:

Telephone.	Add
.....	
9864567654	0x1..
9854354543	0x2..
.....	

Name	City	Dept	...
Tom	DC	IT	..

Name	City	Dept	...
Sam	NYC	HR	..

Limitations for Direct Access Table:

a) **Size** of the created table
b) Integer may not hold the **size of n digits**

Hashing:

So we are going to improve that Direct Access Table by **Hashing**.

We just need a black box (**hash function**) to take the phone number and convert it into a less digit number.

So **Hash Function** maps a big number (phone number) to a small digit to use in a **hash** table as an index.

Hash Function H(x):

H(x) = x mod 7

x = 9864567654

H(x) = 9864567654 mod 7 = 4

Properties of good Hash Function H(x):

- Efficiently computable
- Should uniformly distribute the keys

Collision:

H(x) = x mod 7

x = 9864567654

H(x) = 9864567654 mod 7 = 4

H(x) = x mod 7

x = 9854354542

H(x) = 9854354542 mod 7 = 4

Two telephone numbers with the same key that is the definition of the collision.

Collision Handling:

Chaining:

The idea is to make each cell of hash table point to a **linked list** of records that have same hash function value (key).

Open Addressing:

All elements are sorted in hash table itself.

- **Linear** probing
- **Quadratic** probing
- **Double** hashing

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value (key).

Let's get the point by an example:

Hash Function: H(x) = x mod 7

Keys: 50, 700, 76, 85, 92, 73, 101

Step 1: 50 mod 7 = 1

Step 2: 700 mod 7 = 0

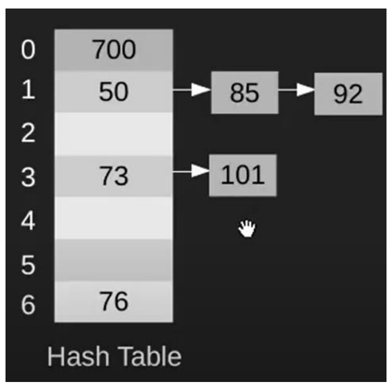
Step 3: 76 mod 7 = 6

Step 4: 85 mod 7 = 1

Step 5: 92 mod 7 = 1

Step 6: 73 mod 7 = 3

Step 7: 101 mod 7 = 3



Advantages of Separate Chaining:

- **Simple** to implement.
- Hash table **never fills up**, we can always **add** more elements to the chain.
- Less sensitive to the **hash function** or **load factors**.
- It is mostly used when it is **unknown** how many and how frequently keys may be inserted or deleted.

Disadvantages of Separate Chaining:

- **Cash** performance of chaining is not good as keys are sorted using linked list.
- **Wasting** of space.
- If the chain becomes long, then **search** time can become **O(n)** in worst case.
- Uses **extra** spaces for links.

Complexity of Separate Chaining:

O (1 + α)

As {α =n/m, n=number of keys sorted in table, m=number of slots in table}

Open Addressing:

- Method of resolving **collision** in hashing
- All items(keys) are sorted in **table itself**
- Size of table **>=** number of keys
- Hash function specifies order of slots to probe (try) for a key (for insert/search/delete), not just **one** slot

1. Linear probing:

Hash Function: hi(X) = (Hash(X) + i) % HashTableSize

If $h0$ = (Hash(X) + 0) % HashTableSize is full we try for $h1$
If $h1$ = (Hash(X) + 1) % HashTableSize is full we try for $h2$
And so on …

Let's get the point by an example:

Keys: 7, 36, 18, 62

Insert (7):

$h0$ (7) = (7 + 0) % 11 = 7 mod 11 = 7

Insert (36):

$h0$ (36) = (36 + 0) % 11 = 36 mod 11 = 3

Insert (18):

$h0$ (18) = (18+ 0) % 11 = 18 mod 11 = 7

$h1$ (18) = (18+ 1) % 11 = 19 mod 11 = 8

Insert (62):

$h0$ (62) = (62+ 0) % 11 = 62 mod 11 = 7

$h1$ (62) = (62+ 1) % 11 = 62 mod 11 = 8

$h2$ (62) = (62+ 2) % 11 = 62 mod 11 = 9

We do the same operation for searching and deleting the elements.

Notice: the deleted element can't stop the search and if we need to **occupy** we can on it.

2. **Quadratic probing:**

Hash Function: hi(X) = (Hash(X) + i^2) % HashTableSize

If $h0$ = (Hash(X) + 0) % HashTableSize is full we try for $h1$
If $h1$ = (Hash(X) + 1^2) % HashTableSize is full we try for $h2$
If $h2$ = (Hash(X) + 2^2) % HashTableSize is full we try for $h3$
And so on …

Let’s get the point by an example:

Keys: 7, 36, 18, 62

Insert (7):

$h0$ (7) = (7 + 0) % 11 = 7 mod 11 = 7

Insert (36):

$h0$ (36) = (36 + 0) % 11 = 36 mod 11 = 3

Insert (18):

$h0$ (18) = (18+ 0) % 11 = 18 mod 11 = 7

$h1$ (18) = (18+ 1^2) % 11 = 19 mod 11 = 8

Insert (62):

$h0$ (62) = (62+ 0) % 11 = 62 mod 11 = 7

$h1$ (62) = (62+ 1^2) % 11 = 62 mod 11 = 8

$h2$ (62) = (62+ 2^2) % 11 = 62 mod 11 = 0

3. **Double Hashing:**

Double hashing is a **collision** resolution technique used in hash tables. It works by using **two** hash functions to compute **two** different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence.

Hash Function: hi(X) = (Hash(X) + i*Hash2(x)) % HashTableSize

If $h0$ = (Hash(X) + 0) % HashTableSize is full we try for $h1$
If $h1$ = (Hash(X) + 1*Hash2(x)) % HashTableSize is full we try for $h2$
If $h2$ = (Hash(X) + 2*Hash2(x)) % HashTableSize is full we try for $h3$
And so on …

Comparison:

Linear probing	Quadratic probing	Double Hashing
• Easy to implement • Best cache performance • Suffers from clustering	• Average cache performance • Suffers a lesser clustering than linear probing	• Poor cache performance • No clustering • Requires more computation time

Complexity:

n=number of keys to be inserted in table

m=number of slots in table

load factor α =n/m (< 1)

Expected time for search/insert/delete **< 1/ (1- α)**

So search/insert/delete takes **O (1/ (1- α))** time