**Hashing**

**I. Introduction**:

Suppose that we want to design a system for storing student records keyed using Student ID and we want following queries to be performed efficiently:

1. Insert a Student ID and corresponding information.

2. Search a Student ID and fetch the information.

3. Delete a Student ID and related information.

For example, to insert a Student ID, we create a record with details of given Student ID, use Student ID as index and store the pointer to the created record in table. But this solution has many practical limitations. The problem with this solution is extra space required is huge. For example, in my university, the student ID has 8 digits, we need O(m \* 108) space for table where m is size of pointer to record. So, hashing appeared to solve this problem.

*Hashing* is the solution that can be used in almost all such situations and performs extremely well compared to data structures like Array, Linked List, … in practice. With hashing we get O(1) search time on average and O(n) in worst case.

Hash function:

The idea is to use hash function that converts a given big number or string to a smaller integer and uses the small number as index in a table called hash table.

A good hash function should have following properties:

1. Efficiently computable.

2. Should uniformly distribute the keys.

Hash Table:

An array that stores the pointers to records corresponding to a given element.

Collision Handling:

Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called *collision* and must be handled using some collision handling technique. We are going to discuss them in part II, III.

**II. 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.

Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 101, 693, 57, 84, 94, 73.



Advantages:

1. Simple to implement.

2. Hash table never fills up, we can always add more elements to the chain.

3. It is mostly used when it is unknown how many keys may be inserted or deleted.

Disadvantages:

1. Cache performance of chaining is not good as keys are stored using a linked list.

2. Wastage of Space (Some Parts of hash table are never used).

3. If the chain becomes long, then search time can become O(n) in the worst case.

4. Uses extra space for links.

**III. Open Addressing**

Open addressing is also a method for handling collisions. In this method, all elements are stored in the hash table itself. So, at any point, the size of the table must be greater than or equal to the total number of keys.

The idea of Open Addressing:

* Insert(k): Keep probing until an empty slot is found. After an empty slot is found, insert k.
* Search(k): Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.
* Delete(k): If we simply delete a key, then the search may fail. So, slots of deleted keys are marked specially as “deleted”. The insert can insert an item in a deleted slot, but the search doesn’t stop at a deleted slot.

**1. Linear probing**

If the current position has been taken, we will find another position. In linear probing, we linearly probe for next slot.

h (k, i) = (hash(k) + i) mod S

Where:

i is the order of the attempt (i = 0, 1, 2, …)

hash(k): hash function

S: size of the array

Let **hash(x)** be the slot index computed using a *hash function* and **S** be the table size:

If slot [hash(x) % S] is full, then we try [(hash(x) + 1) % S]

If [(hash(x) + 1) % S] is also full, then we try [(hash(x) + 2) % S]

If [(hash(x) + 2) % S] is also full, then we try [(hash(x) + 3) % S]

……………………………………………………………………

Let us consider a simple hash function as “**key mod 7**” and a sequence of keys as 50, 700, 76, 85, 92.



Some problems with **Linear Probing**:

1. Many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. Two records only have the same collision chain if their initial position is the same.

**2. Quadratic Probing**

Like linear probing, but we use quadratic function.

h (k, i) = (hash(k) + i2) mod S

Where:

i: the order of the attempt (i = 0, 1, 2, …)

hash(k): hash function

S: size of the array

Let **hash(x)** be the slot index computed using hash function.

If slot [hash(x) % S] is full, then we try [(hash(x) + 1 \* 1) % S]

If [(hash(x) + 1 \* 1) % S] is also full, then we try [(hash(x) + 2 \* 2) % S]

If [(hash(x) + 2 \* 2) % S] is also full, then we try [(hash(x) + 3 \* 3) % S]

……………………………………………………………………………

Let us consider a simple hash function as “**key mod 7**” and a sequence of keys as 76, 40, 48, 5, 55.



**3. Double Hashing**

Double hashing uses the idea of applying a second hash function to key when a collision occurs.

h (k, i) = (hash1(k) + i\* hash2(k)) mod S

Where:

i: the order of the attempt (i = 0, 1, 2, …)

hash1(k) and hash2(k): hash functions

S: size of the array

let hash(x) be the slot index computed using hash function.

If slot [hash(x) % S] is full, then we try [(hash(x) + 1\*hash2(x)) % S]

If [(hash(x) + 1\*hash2(x)) % S] is also full, then we try [(hash(x) + 2\*hash2(x)) % S]

If [(hash(x) + 2\*hash2(x)) % S] is also full, then we try [(hash(x) + 3\*hash2(x)) % S]

.......................................................................................................................................

First hash function is typically hash1(k) = k % S

A popular second hash function is: hash2(k) = PRIME – (k % PRIME) where PRIME is a prime smaller than the TABLE\_SIZE.

A good second Hash function is:

* It must never evaluate to zero.
* Must make sure that all cells can be probed.

Let us consider 2 simple hash functions as “**key mod 13**” and “**key mod 7**”, and a sequence of keys as 19, 27, 36, 10.

