Hashing in DBMS

In a large database, data is stored at various locations. It becomes hectic and time-consuming when locating a specific type of data in a database via linear search or binary search. This problem is solved by "Hashing".
Hashing is an advantageous technique which uses a hash function to find the exact location of a data record in minimum amount of time.
For example, we recorded data of multiple students in an alphabetical format in a database of college. But, it is still difficult to locate data every time using linear search.

Why do we need Hashing?

In DBMS, hashing is a technique to directly search the location of desired data on the disk without using index structure. Data is stored in the form of data blocks whose address is generated by applying a hash function in the memory location where these records are stored known as a data block or data bucket.

Here, are the situations in the DBMS where you need to apply the Hashing method:

For a huge database structure, it's tough to search all the index values through all its level and then you need to reach the destination data block to get the desired data.

Hashing method is used to index and retrieve items in a database as it is faster to search that specific item using the shorter hashed key instead of using its original value.

Hashing is an ideal method to calculate the direct location of a data record on the disk without using index structure.

It is also a helpful technique for implementing dictionaries.

Important Terminologies using in Hashing

Here, are important terminologies which are used in Hashing:

Data bucket – Data buckets are memory locations where the records are stored. It is also known as Unit of Storage.

Key: A DBMS key is an attribute or set of an attribute which helps you to identify a row(tuple) in a relation(table). This allows you to find the relationship between two tables.

Hash function: A hash function, is a mapping function which maps all the set of search keys to the address where actual records are placed.

Linear Probing – Linear probing is a fixed interval between probes. In this method, the next available data block is used to enter the new record, instead of overwriting on the older record.

Quadratic probing- It helps you to determine the new bucket address. It helps you to add Interval between probes by adding the consecutive output of quadratic polynomial to starting value given by the original computation.

Hash index – It is an address of the data block. A hash function could be a simple mathematical function to even a complex mathematical function.

Double Hashing –Double hashing is a computer programming method used in hash tables to resolve the issues of has a collision.

Bucket Overflow: The condition of bucket-overflow is called collision. This is a fatal stage for any static has to function.

There are mainly two types of SQL hashing methods:

1. Static Hashing
2. Dynamic Hashing

Static Hashing

In the static hashing, the resultant data bucket address will always remain the same.

Therefore, if you generate an address for say Student_ID = 10 using hashing function mod(3), the resultant bucket address will always be 1. So, you will not see any change in the bucket address.

Therefore, in this static hashing method, the number of data buckets in memory always remains constant.

HASH functions

A hash function used in hashing is also called 'hashing algorithm'. A hashing algorithm uses a hash key to locate a data record. A hash key is a string of characters which is transformed into shorter-length hash address by a hashing algorithm.

A hashing function can vary from a simplest mathematical function to any complex function. Though it is required that a hash-function should be easy and quick to generate results.

While inserting a data record, the hash function uses a hash key to produce a relative hash address and allocate this hash address to each data record. Any time, when the database needs to search, update or delete a record; it looks for hash addresses related to each record and performs its desired operation.

A hash-function is termed to be "good" if it does not generate same hash-address for different hash-keys.

Following are some known hashing-algorithms used in the database. All of these hashing algorithms are easy and quick to compute results:-

- **Division method: -** A hash function which uses division method is represented as;

$H(k) = k(\text{mod } m)$ or $H(k) = k(\text{mod } m) + 1$

where; k signifies a hash key and m is chosen to be a prime no. which is greater than total no. of keys. In the above formulas, k(mod m) indicates the remainder when k is divided by m. The first formulae range hash addresses from 0 to m-1 where second formulae range hash addresses from 1 to m.

Consider a class with 68 students and each student is given a unique 4-digit student number. 1024, 2448 and 3466 are few unique student numbers. Here, student nos. denote k i.e. hash key and we choose m to be 71 which is greater than total no. of keys and also a prime number.

Using first formulae, value of hash address H(k) results into:

H(1024)=1024(mod 71)=14, H(2448)=2448(mod 71)=34, H(3466)=3466(mod 71)=48

Where 14, 34 and 48 are the hash-addresses of students associated with student no. 1024, 2448 and 3466. Similarly, if we use second formulae, the value of hash-addresses results into 15, 35 and 49 respectively.

- **Mid-square method:-** In this hashing algorithm, we square hash key(k) and eliminate digits from both ends of k^2. It is recommended that same positions are always used from all the k^2 values to retrieve hash-addresses. The formulae of mid-square method is represented below;

$$H(k) = l$$

Where l is the value of hash address computed after eliminating digits from both ends of k^2.

Following are the results of above example using mid-square method;

k	1024	2448	3466
k^2	1048576	5992704	12013156
h(k)	48	92	13

Notice that fourth and fifth digits are used to retrieve hash address counting from right.

- **Folding method: -** This hashing algorithm chop a hash key into no. of parts and compute a hash address after adding these parts and ignoring the carry. We can also reverse even-numbered parts to retrieve a hash key. Folding method hashing-algorithm is represented by;

$$H(k)=k_1+k_2+.....+k_n$$

Considering above example, hash keys available are: 1024, 2448 and 3466. Now compute hash-address using this method;

H(1024) = 10+24 = 34, H(2448) = 24+88 = 12, H(3466) = 34+66 = 00

Carry is ignored when we generated results using keys 2448 and 3466; '00' is also a hash address in the database, it is not a nil number.

Now, observe that each hashing-algorithm results into a different hash address related to each student no. and these hash addresses are not equivalent to serial nos. of students.

Static Hash Functions

Inserting a record: When a new record requires to be inserted into the table, you can generate an address for the new record using its hash key. When the address is generated, the record is automatically stored in that location.

Searching: When you need to retrieve the record, the same hash function should be helpful to retrieve the address of the bucket where data should be stored.

Delete a record: Using the hash function, you can first fetch the record which is you wants to delete. Then you can remove the records for that address in memory.

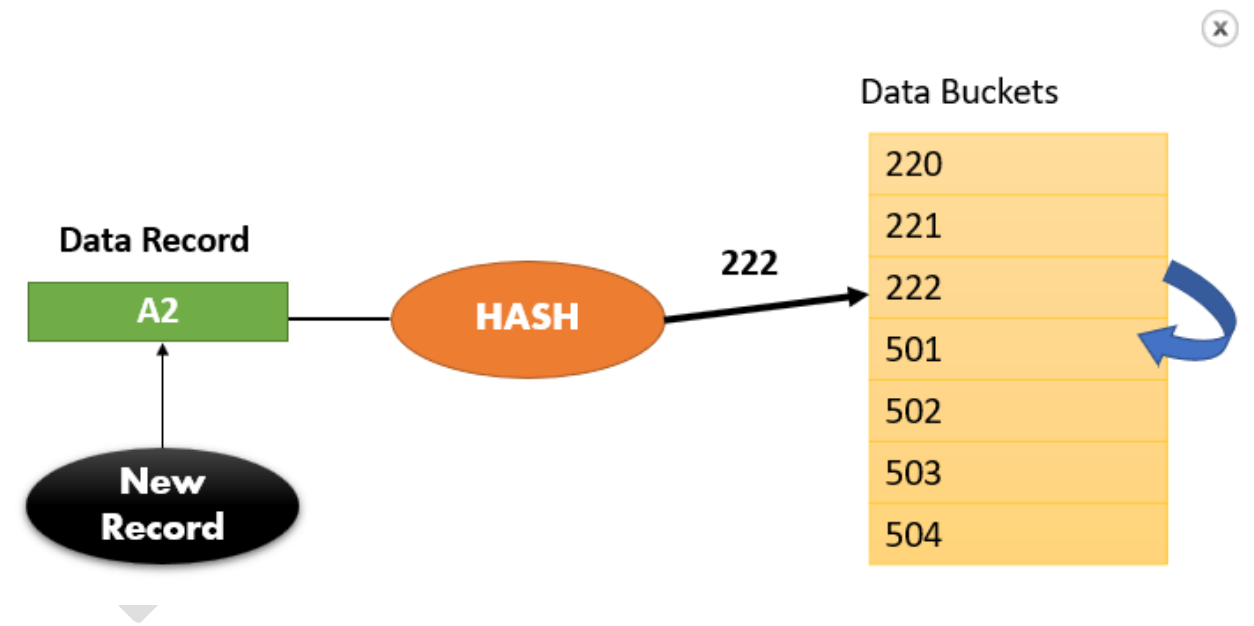
Static hashing is further divided into

1. Open hashing
2. Close hashing.

Open Hashing

In Open hashing method, instead of overwriting older one the next available data block is used to enter the new record, This method is also known as linear probing.

For example, A2 is a new record which you wants to insert. The hash function generates address as 222. But it is already occupied by some other value. That's why the system looks for the next data bucket 501 and assigns A2 to it.



Close Hashing

In the close hashing method, when buckets are full, a new bucket is allocated for the same hash and result are linked after the previous one.

Dynamic Hashing

Dynamic hashing offers a mechanism in which data buckets are added and removed dynamically and on demand. In this hashing, the hash function helps you to create a large number of values.

Hash Collision

Hash collision is a state when the resultant hashes from two or more data in the data set, wrongly map the same place in the hash table.

How to deal with Hashing Collision?

Collision Resolution Techniques

There are two broad ways of collision resolution:

1. Separate Chaining: An array of linked list implementation.

2. Open Addressing: Array-based implementation.

(i) Linear probing (linear search)

(ii) Quadratic probing (nonlinear search)

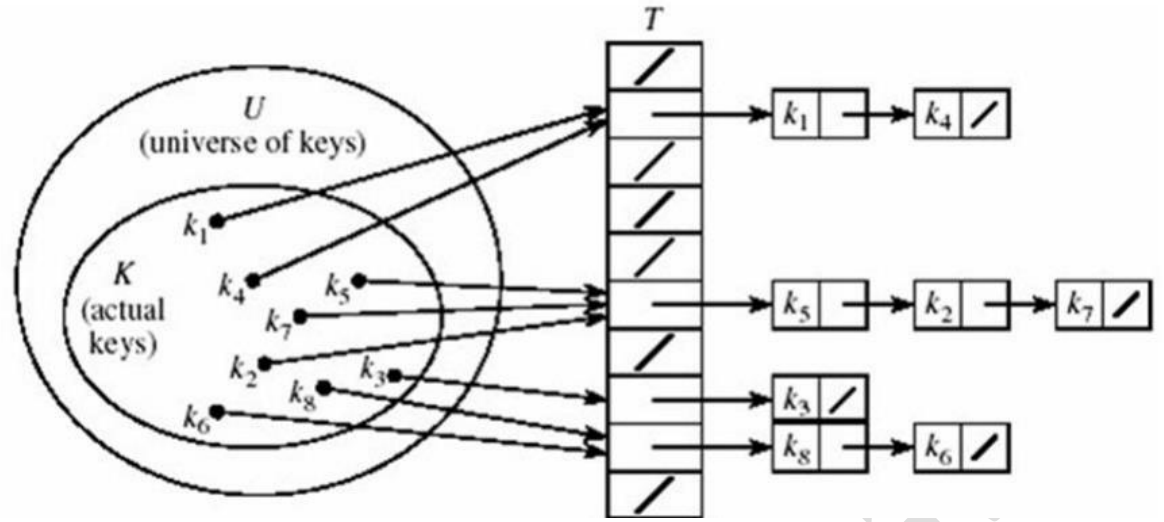
(iii) Double hashing (uses two hash functions)

Separate Chaining

• The hash table is implemented as an array of linked lists.

• Inserting an item, r, that hashes at index i is simply insertion into the linked list at position i.

• Synonyms are chained in the same linked list.



Retrieval of an item,

r, with hash address,

i, is simply retrieval from the linked list

at position

i.

•

Deletion of an item,

r, with hash address,

i, is simply deleting

r from the linked list

at position i.

• **Example:** Load the keys 23, 13, 21, 14, 7, 8, and 15 , in this order, in a hash table of size 7 using separate chaining with the hash function: h(key) = key % 7

h(23) = 23 % 7 = 2

h(13) = 13 % 7 = 6

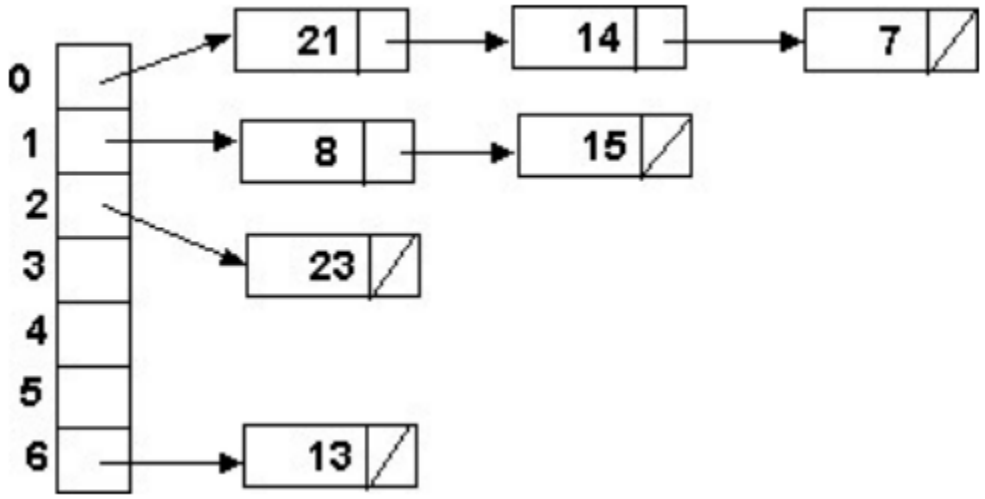
h(21) = 21 % 7 = 0

h(14) = 14 % 7 = 0 collision

h(7) = 7 % 7 = 0 collision

h(8) = 8 % 7 = 1

h(15) = 15 % 7 = 1 collision



Separate Chaining with String Keys

•

Recall that search keys can be numbers, strings or some other object.

•

A hash function for a string s = c0c1c2…cn-1 can be defined as:

hash = (c0 + c1 + c2 + …+ cn-1) % tableSize

Use the hash function hash to load the following commodity items into a hash table of size 13 using separate chaining:

onion	1	10.0
tomato	1	8.50
cabbage	3	3.50
carrot	1	5.50
okra	1	6.50
mellon	2	10.0
potato	2	7.50
Banana	3	4.00
olive	2	15.0
salt	2	2.50
cucumber	3	4.50
mushroom	3	5.50
orange	2	3.00

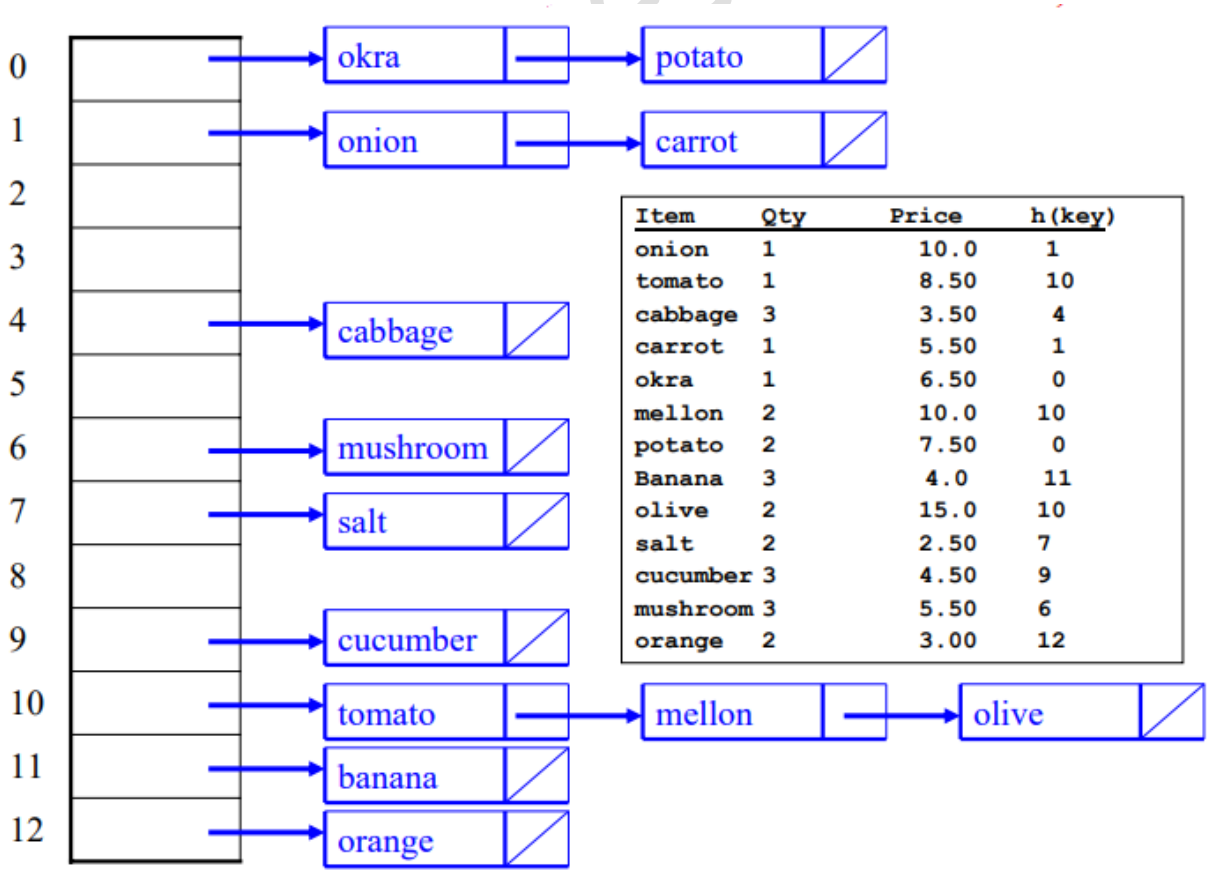
- Solution:

character	a	b	c	e	g	h	i	k	l	m	n	o	p	r	s	t	u	v
ASCII code	97	98	99	101	103	104	105	107	108	109	110	111	112	114	115	116	117	118

hash(onion) = (111 + 110 + 105 + 111 + 110) % 13 = 547 % 13 = 1

hash(salt) = (115 + 97 + 108 + 116) % 13 = 436 % 13 = 7

hash(orange) = (111 + 114 + 97 + 110 + 103 + 101)%13 = 636 %13 = 12



Item	Qty	Price	h(key)
onion	1	10.0	1
tomato	1	8.50	10
cabbage	3	3.50	4
carrot	1	5.50	1
okra	1	6.50	0
mellon	2	10.0	10
potato	2	7.50	0
Banana	3	4.0	11
olive	2	15.0	10
salt	2	2.50	7
cucumber	3	4.50	9
mushroom	3	5.50	6
orange	2	3.00	12

Alternative hash functions for a string

s = c0c1c2…cn-1

exist, some are:

•hash = (c0 + 27 * c1 + 729 * c2) % tableSize

• hash = (c0 + cn-1 + s.length()) % tableSize

• hash =[26*k s.charAt(k) "]%tableSize

Separate Chaining versus Open-addressing

Separate Chaining has several advantages over open addressing:

•Collision resolution is simple and efficient.

•The hash table can hold more elements without the largeperformance deterioration of open addressing (The load factor canbe 1 or greater)

•The performance of chaining declines much more slowly thanopen addressing.

•Deletion is easy - no special flag values are necessary.

•Table size need not be a prime number.

•The keys of the objects to be hashed need not be unique.

Disadvantages of Separate Chaining:

•It requires the implementation of a separate data structure for

chains, and code to manage it.

•The main cost of chaining is the extra space required for the linked lists.

•For some languages, creating new nodes (for linked lists) isexpensive and slows down the system.

Introduction to Open Addressing

• All items are stored in the hash table itself.

• In addition to the cell data (if any), each cell keeps one of the three states: EMPTY,

OCCUPIED, DELETED.

• While inserting, if a collision occurs, alternative cells are tried until an empty cell is found.

• Deletion: (lazy deletion): When a key is deleted the slot is marked as DELETED rather than EMPTY otherwise subsequent searches that hash at the deleted cell will fail.

• Probe sequence: A probe sequence is the sequence of array indexes that is followed in searching for an empty cell during an insertion, or in searching for a key during find or delete operations.

• The most common probe sequences are of the form:

hi(key) = [h(key) + c(i)] % n, for i = 0, 1, …, n-1.

where

h is a hash function and

n is the size of the hash table

• The function c(i) is required to have the following two properties:

Property 1: c(0) = 0

Property 2: The set of values {c(0) % n, c(1) % n, c(2) % n, . . . , c(n-1) % n} must be a

permutation of {0, 1, 2,. . ., n –1}, that is, it must contain every integer between

0 and n -1 inclusive.

The function c(i) is used to resolve collisions.

• To insert item r, we examine array location

h0(r) = h(r). If there is a collision, array locations

h1(r), h2(r), ..., hn-1(r) are examined until an empty slot is found.

• Similarly, to find item r, we examine the same sequence of locations in the same order.

• Note: For a given hash function h(key), the only difference in the open addressing collision resolution techniques (linear probing, quadratic probing and double hashing) is in the definition of the function c(i).

• Common definitions of c(i) are:

Collision resolution technique	c(i)
Linear probing	**i**
Quadratic probing	**$\pm i^2$**
Double hashing	**$i*h_p(key)$**

where $\mathbf{h_p(key)}$ is another hash function.

Advantages of Open addressing:

–All items are stored in the hash table itself. There is no need for another data structure.

–Open addressing is more efficient storage-wise.

Disadvantages of Open Addressing:

–The keys of the objects to be hashed must be distinct.

–Dependent on choosing a proper table size.

–Requires the use of a three-state (Occupied, Empty, or Deleted) flag in each cell.

In general, primes give the best table sizes.

•With any open addressing method of collision resolution, as the table fills, there can be a severe degradation in the table performance.

•Load factors between 0.6 and 0.7 are common.

•Load factors > 0.7 are undesirable.

•The search time depends only on the load factor, not on the table size.

•We can use the desired load factor to determine appropriate table size:

$$\text{table size} = \text{smallest prime} \geq \frac{\text{number of items in table}}{\text{desired load factor}}$$

Open Addressing: Linear Probing

It is a Scheme in Computer Programming for resolving collision in hash tables.

Suppose a new record R with key k is to be added to the memory table T but that the memory locations with the hash address H (k). H is already filled.

Our natural key to resolve the collision is to crossing R to the first available location following T (h). We assume that the table T with m location is circular, so that T [i] comes after T [m].

The above collision resolution is called "**Linear Probing".**

Linear probing is simple to implement, but it suffers from an issue known as primary clustering. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot proceeded by i full slots gets filled next with probability (i + 1)/m. Long runs of occupied slots tend to get longer, and the average search time increases.

Given an ordinary hash function h': U {0, 1...m-1}, the method of linear probing uses the hash function.

h (k, i) = (h' (k) + i) mod m

Where 'm' is the size of hash table and h' (k) = k mod m. for i=0, 1....m-1.

Given key k, the first slot is T [h' (k)]. We next slot T [h' (k) +1] and so on up to the slot T [m-1]. Then we wrap around to slots T [0], T [1]....until finally slot T [h' (k)-1]. Since the initial probe position dispose of the entire probe sequence, only m distinct probe sequences are used with linear probing.

Example1: Perform the operations given below, in the given order, on an initially empty hash table of size 13 using linear probing with c(i) = i and the hash function: **h(key) = key % 13:**

insert(18), insert(26), insert(35), insert(9), find(15), find(48), delete(35), delete(40), find(9), insert(64), insert(47), find(35)

•The required probe sequences are given by:

hi(key) = (h(key) + i) % 13 i = 0, 1, 2, . . ., 12

OPERATION	PROBE SEQUENCE	COMMENT
insert(18)	$h_0(18) = (18 \% 13) \% 13 = 5$	SUCCESS
insert(26)	$h_0(26) = (26 \% 13) \% 13 = 0$	SUCCESS
insert(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	SUCCESS
insert(9)	$h_0(9) = (9 \% 13) \% 13 = 9$	COLLISION
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
find(15)	$h_0(15) = (15 \% 13) \% 13 = 2$	FAIL because location 2 has **Empty** status
find(48)	$h_0(48) = (48 \% 13) \% 13 = 9$	COLLISION
	$h_1(48) = (9 + 1) \% 13 = 10$	COLLISION
	$h_2(48) = (9 + 2) \% 13 = 11$	FAIL because location 11 has **Empty** status
withdraw(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	SUCCESS because location 9 contains 35 and the status is **Occupied** The status is changed to **Deleted**; but the key 35 is not removed.
find(9)	$h_0(9) = (9 \% 13) \% 13 = 9$	The search continues, location 9 does not contain 9; but its status is **Deleted**
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
insert(64)	$h_0(64) = (64 \% 13) \% 13 = 12$	SUCCESS
insert(47)	$h_0(47) = (47 \% 13) \% 13 = 8$	SUCCESS
find(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	FAIL because location 9 contains 35 but its status is **Deleted**

Index	Status	Value
0	O	26
1	E	
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	D	35
10	O	9
11	E	
12	O	64

Another Example

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

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

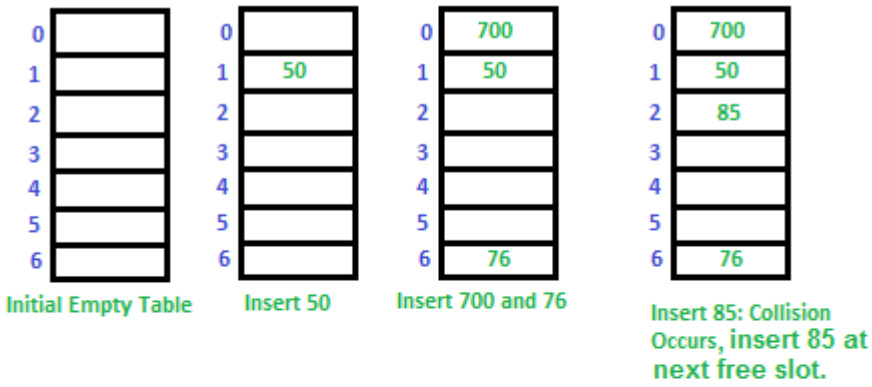
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

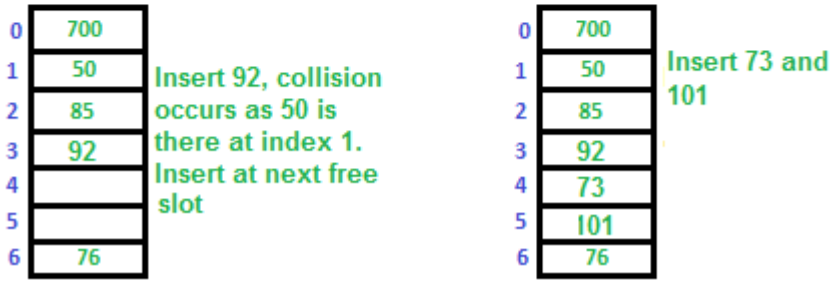
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

.....

.....

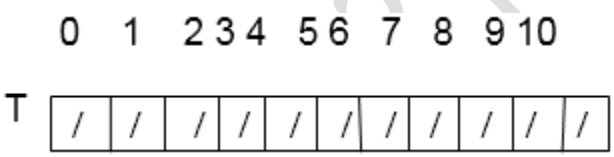
Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.





Example3: Consider inserting the keys 24, 36, 58,65,62,86 into a hash table of size m=11 using linear probing, consider the primary hash function is h' (k) = k mod m.

Solution: Initial state of hash table



Insert 24. We know h (k, i) = [h' (k) + i] mod m

Now h (24, 0) = [24 mod 11 + 0] mod 11

= (2+0) mod 11 = 2 mod 11 = 2

Since T [2] is free, insert key 24 at this place.

Insert 36. Now h (36, 0) = [36 mod 11 + 0] mod 11

= [3+0] mod 11 = 3

Since T [3] is free, insert key 36 at this place.

Insert 58. Now h (58, 0) = [58 mod 11 +0] mod 11

= [3+0] mod 11 =3

Since T [3] is not free, so the next sequence is

h (58, 1) = [58 mod 11 +1] mod 11
= [3+1] mod 11= 4 mod 11=4

T [4] is free; Insert key 58 at this place.

Insert 65. Now h (65, 0) = [65 mod 11 +0] mod 11
= (10 +0) mod 11= 10

T [10] is free. Insert key 65 at this place.

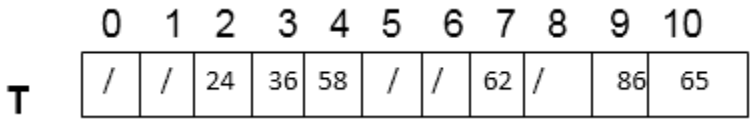
Insert 62. Now h (62, 0) = [62 mod 11 +0] mod 11
= [7 + 0] mod 11 = 7

T [7] is free. Insert key 62 at this place.

Insert 86. Now h (86, 0) = [86 mod 11 + 0] mod 11
= [9 + 0] mod 11 = 9

T [9] is free. Insert key 86 at this place.

Thus,



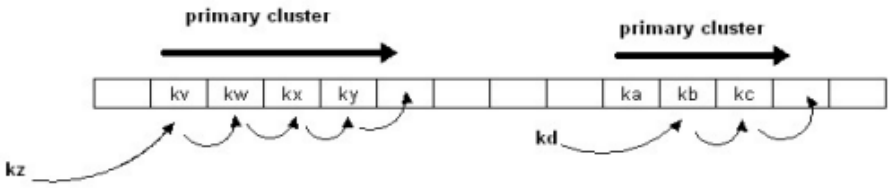
Disadvantage of Linear Probing: Primary Clustering

Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

•Linear probing is subject to a primary clustering phenomenon.

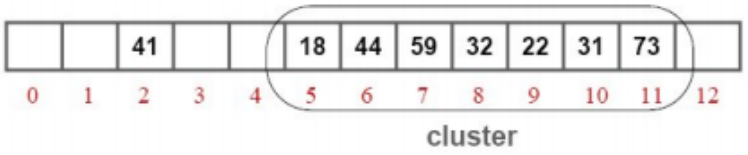
•Elements tend to cluster around table locations that they originally hash to.

•Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency.



Example of a primary cluster: Insert keys: **18, 41, 22, 44, 59, 32, 31, 73**, in this order, in an originally empty hash table of size **13**, using the hash function **h(key) = key % 13** and **c(i) = i**:

h(18) = 5
h(41) = 2
h(22) = 9
h(44) = 5+1
h(59) = 7
h(32) = 6+1+1
h(31) = 5+1+1+1+1+1
h(73) = 8+1+1+1



Quadratic Probing

We look for i^2'th slot in i'th iteration.

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

.....

.....

Quadratic Probing uses a hash function of the form

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Where (as in linear probing) h' is an auxiliary hash function c1 and c2 ≠0 are auxiliary constants and i=0, 1...m-1. The initial position is T [h' (k)]; later position probed is offset by the amount that depend in a quadratic manner on the probe number i.

The method of quadratic probing is found to be better than linear probing. However, to ensure that the full hash table is covered, the values of c1, and c2 are constrained. It may happen that two keys produce the same probe sequence such that:

$h(k_1, i) = h(k_2, i)$

Therefore, this leads to a kind of clustering called secondary clustering. Just as in linear probing, the initial probe position determines the entire probe sequence.

Example: Consider inserting the keys 74, 28, 36,58,21,64 into a hash table of size m =11 using quadratic probing with c_1=1 and c_2=3. Further consider that the primary hash function is h' (k) = k mod m.

Solution: For Quadratic Probing, we have

h (k, i) = [k mod m +c_1i +c_2 i^2) mod m

This is the initial state of hash table

Here c_1= 1 and c_2=3

h (k, i) = [k mod m + i + $3i^2$] mod m

Insert 74.

h (74,0)= (74 mod 11+0+3x0) mod 11

= (8 +0+0) mod 11 = 8

T [8] is free; insert the key 74 at this place.

Insert 28.

h (28, 0) = (28 mod 11 + 0 + 3 x 0) mod 11

= (6 +0 + 0) mod 11 = 6.

T [6] is free; insert key 28 at this place.

Insert 36.

h (36, 0) = (36 mod 11 + 0 + 3 x 0) mod 11

= (3 + 0+0) mod 11=3

T [3] is free; insert key 36 at this place.

Insert 58.

h (58, 0) = (58 mod 11 + 0 + 3 x 0) mod 11

= (3 + 0 + 0) mod 11 = 3

T [3] is not free, so next probe sequence is computed as

h (59, 1) = (58 mod 11 + 1 + 3 x1^2) mod 11

= (3 + 1 + 3) mod 11

=7 mod 11= 7

T [7] is free; insert key 58 at this place.

Insert 21.

h (21, 0) = (21 mod 11 + 0 + 3 x 0)

= (10 + 0) mod 11 = 10

T [10] is free; insert key 21 at this place.

Insert 64.

h (64, 0) = (64 mod 11 + 0 + 3 x 0)

= (9 + 0+ 0) mod 11 = 9.

T [9] is free; insert key 64 at this place.

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	/	36	/	/	28	58	74	64	21

3. Double Hashing

Double Hashing is one of the best techniques available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form

h (k, i) = (h1(k) + i h2 (k)) mod m

Where h1 and h2 are auxiliary hash functions and m is the size of the hash table.

h1 (k) = k mod m or h2 (k) = k mod m'. Here m' is slightly less than m (say m-1 or m-2)..

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

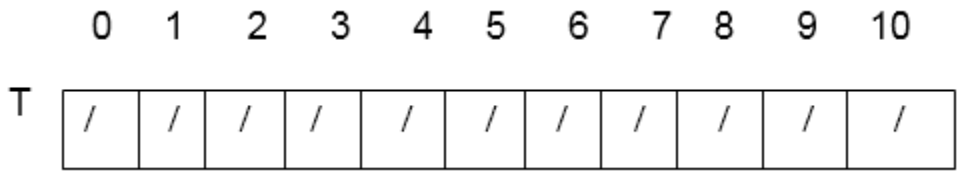
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

.....

.....

Example: Consider inserting the keys 76, 26, 37,59,21,65 into a hash table of size m = 11 using double hashing. Consider that the auxiliary hash functions are h_1 (k)=k mod 11 and h_2(k) = k mod 9.

Solution: Initial state of Hash table is



1. Insert 76.

h_1(76) = 76 mod 11 = 10

h_2(76) = 76 mod 9 = 4

h (76, 0) = (10 + 0 x 4) mod 11

= 10 mod 11 = 10

T [10] is free, so insert key 76 at this place.

2. Insert 26.

h_1(26) = 26 mod 11 = 4

h_2(26) = 26 mod 9 = 8

h (26, 0) = (4 + 0 x 8) mod 11

= 4 mod 11 = 4

T [4] is free, so insert key 26 at this place.

3. Insert 37.

h_1(37) = 37 mod 11 = 4

h_2(37) = 37 mod 9 = 1

h (37, 0) = (4 + 0 x 1) mod 11 = 4 mod 11 = 4

T [4] is not free, the next probe sequence is

h (37, 1) = (4 + 1 x 1) mod 11 = 5 mod 11 = 5

T [5] is free, so insert key 37 at this place.

4. Insert 59.

$h_1(59) = 59 \bmod 11 = 4$

$h_2(59) = 59 \bmod 9 = 5$

h (59, 0) = (4 + 0 x 5) mod 11 = 4 mod 11 = 4

Since, T [4] is not free, the next probe sequence is

h (59, 1) = (4 + 1 x 5) mod 11 = 9 mod 11 = 9

T [9] is free, so insert key 59 at this place.

5. Insert 21.

$h_1(21) = 21 \bmod 11 = 10$

$h_2(21) = 21 \bmod 9 = 3$

h (21, 0) = (10 + 0 x 3) mod 11 = 10 mod 11 = 10

T [10] is not free, the next probe sequence is

h (21, 1) = (10 + 1 x 3) mod 11 = 13 mod 11 = 2

T [2] is free, so insert key 21 at this place.

6. Insert 65.

$h_1(65) = 65 \bmod 11 = 10$

$h_2(65) = 65 \bmod 9 = 2$

h (65, 0) = (10 + 0 x 2) mod 11 = 10 mod 11 = 10

T [10] is not free, the next probe sequence is

h (65, 1) = (10 + 1 x 2) mod 11 = 12 mod 11 = 1

T [1] is free, so insert key 65 at this place.

Thus, after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76