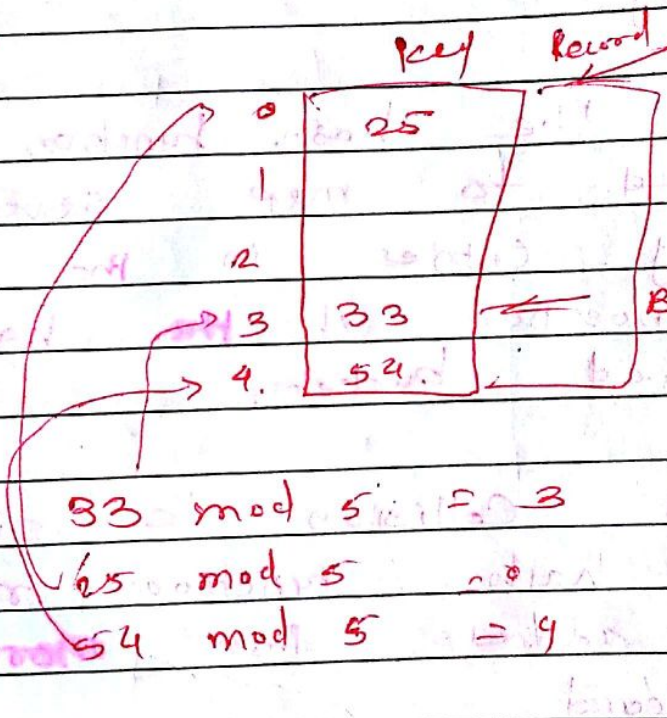
3. Hashing

Concept:

Hashing is an effective way to reduce the number of comparisons. Already hashing deals with the idea of proving the direct address of the record where the record is likely to store

e.g.



Basic Concepts in Hashing:

1) Hash Table: Hash table is a data structure used for storing & retrieving data quickly. Every entry in the hash table is made using hash table function

2) Hash function :—

Hash func^n is a fun^n used to place data in hash table & to retrieve data from hash table.

This hash fun^n is used to implement hash table.

e.g.

3) Bucket : The hash function $H(key)$ is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

4) Collision : Collision is situation in which hash function returns the same address for more than one record.

e.g. $25 \bmod 5 = 0$

$35 \bmod 5 = 0$.

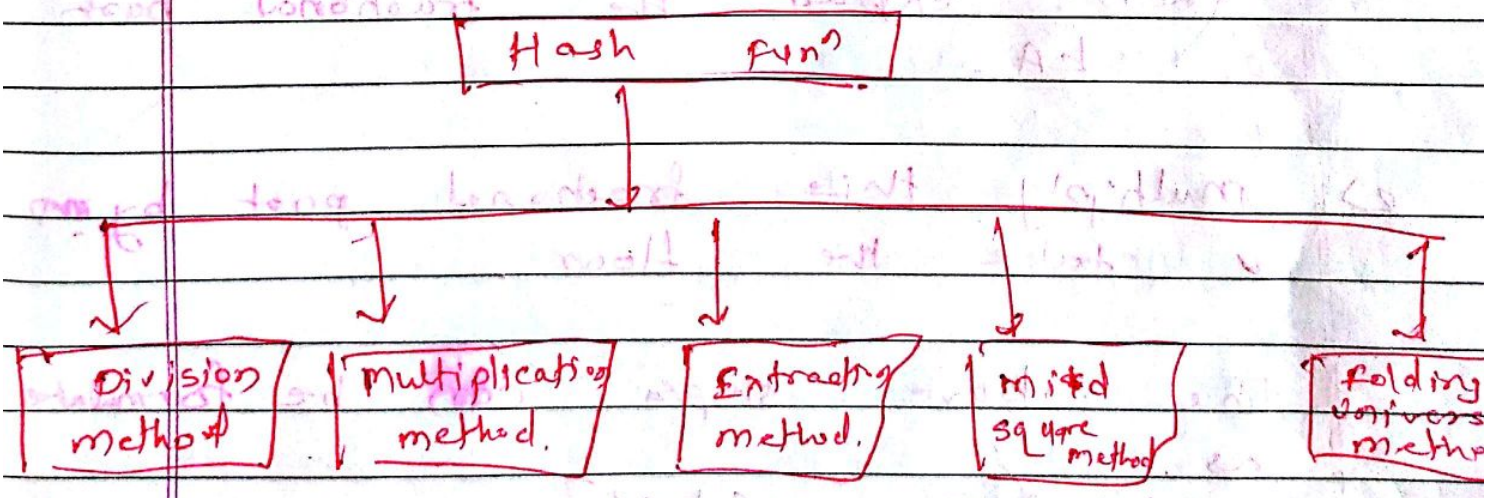
5) Probe : Each calculation address for success is known as probe.

6) Synonym : The pair of keys that has the same location are synonyms. eg. 25 & 55

7) Overflow : When Hash table becomes full & new record needs to be inserted then it is called overflow

Hash functions :

Types



Basic Concept Continue →

Overflow: The result of many keys hashing to a single address & lack of room in the bucket is known as an overflow. Collision & overflow are synonymous when the bucket is of size 1.

Open or external hashing: When we allow records to be stored in potentially unlimited space, it is called as open or external hashing.

Closed or internal hashing: When we use fixed space for storage eventually limiting the number of records to be stored, it is called as closed or internal hashing

Hash fun^n: Hash fun^n is an arithmetic fun^n that transforms a key into an address which is used for storing & retrieving a record.

Perfect hash function: The hash function that transforms different keys into different addresses is called a perfect hash function. The worth of a hash function depends on how well it avoids collision

Load density : The maximum storage capacity, that is the maximum no. of records that can be accommodated is called as loading density.

Load factor: Load factor is the number of records stored in a table divided by the maximum capacity of the table, expressed in terms of percentage.

1) Division method :

54 % 10 = 4
72 % 10 = 2
89 % 10 = 9
37 % 10 = 7

Hash Table

0	
1	
2	
3	37
4	54
5	
6	
7	37
8	
9	89

2) Multiplication Hash fun^n :

1) Multiply the key 'k' by a const A where A is in range $0 < A$ Then extract the fractional part of kA.

2) Multiply this fractional part by m & take the floor.

The above steps can be formula as

$$h(k) = \lfloor m\{kA\} \rfloor$$

↑
fractional part.

Donald knuth suggested to use

A = 0.6180339887

eg. k = 107, assume m = 50.

$h(k) = [m * (107 * 0.6 \ldots\ldots)]$

$= [50 * 0.12]$

$= 6$

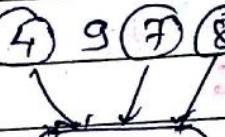
That means key 107 will be placed at index 6 in hash table.

Scanned by CamScanner

* Extraction :

In this method some digits are extracted from the key to form the address location in the table.

e.g. Suppose first, third & fourth digit from left is selected for hash key.



→ at 478 location in the hash table of size 1000 the key can be stored.

4) "mid square"

This method works in following steps.

1) Square the key.
2) Extract middle part of the result. This will indicate the location of the key element in the hash table.

Cond": Mid-square is used when the key size is less than or equal to 4 digits because if key is large then it is very difficult to store its square as it should exceed the storage limit.

Let key = 311

$\therefore (311)^2$

$\Rightarrow$ 96 | 783 | 21

For the hash table of size of 1000

H (311) = 783

5) Folding:

There are two folding techniques

i) Fold shift ii) Fold boundary

i) Fold shift: In this method the key is divided into separate parts whose size matches with the size of required address. Then left & right parts are shifted & added with middle part.

ii) Fold boundary: In this method the key is divided into separate parts. The leftmost & rightmost parts are folded on a fixed boundary & added with the middle part.

key.

345678123

i) Fold shift.

345 | 678 | 123

↓

345
+ 678
123
1146

↑
Discard

∴ Index = 146.

ii) Fold Boundry.

345 | 678 | 123

↓ digit reversed

543

678

2321

1542

↑ Discard.

∴ Index = 542

* Properties of Good "Hash fund"

1. The "hash fund" should be simp to compute.
2. No. of collisions should be le while placing the record in th hash table. Ideally no coll should occur. Such a function is called perfect hash function.

3. Hash fun^n should produce such keys which will get distributed uniformly over array.

4. The hash fun^n should depend on every bit of the key. Thus the hash fun^n that simply extracts the portion of a key is not suitable.

6. Rotation :

When the key are serial, they vary only in the last digit & this leads to the creation of synonyms. Rotation the key would minimize this problem. Here the key is rotated right by 1 digit & then folding technique is used to avoid synonyms eg. 12065, when it is rotated we get 51206.

* Universal Hashing :

The main idea behind universal hashing is to select the hash function at random at runtime from a carefully designed set of functions.

Because of randomization, the algorithm can behave differently on each execution, even for same input. This approach guarantees good average performance, no matter what keys are provided as input.

* Collision Resolution strategies:

three methods.

1. Open addressing: (closed Hashing)
   a) Linear probing → chaining.
   b) Quadratic probing
   c) Double hashing.
   d) Rehashing.
2. Separate chaining (Linked List.)
3. Bucket Hashing

instead of same table { Open Addressing & closed Hashing
external Hashing

closed Hashing
open Hashing
Internal Hashing

1) Open Addressing :

This is collision handling technique in which the entire hash table is searched in systemic way for empty cell to insert new item if collision occurs.

2) Linear probing :

When collision occurs i.e. when 2 records demand for the same location in the hash table, then the collision can be solved by placing second record linearly down wherever the empty location is found.

(key + i) mod Max (table size).

~~(2 + 2) % 10~~

~~4 % 10 = 4~~.

e.g. Table of size 100 & key1 = 1044 & key2 = 3544.

key1 is stored at location

Hash (key1) = key1 % 100

= 1044 % 100

= 44.

Index	key
0	
1	
2	
:	
44	1044
45	3544
:	
99	

Now have to store 2nd key = 35

then Hash(key2) = 3544 % 100
= 44.

But location 44 is already occupied so store it at next empty location other 44.
So 3544 is stored at location 45.

* Problem with linear probing:

One major problem with linear probing is major primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

eg.

0	39
1	29
2	9
3	
4	
5	
6	
7	
8	18
9	19

} cluster.

19 % 10 = 9
18 % 10 = 8
39 % 10 = 9
29 % 10 = 9
8 % 10 = 8

This

Linear probing using 2 ways.

i) With Replacement
ii) Without replacement.

i) Linear probing with Replacement :

0	20
1	
2	22
3	13
4	24
5	15
6	16
7	14
8	19
9	

eg. (24, 13, 16, 15, 19, 20, 22, 14, 17, 26) ~~84, 9~~.

Place 14 Here, cause 14 % 10 = 4
The index 4 contain the elemen 24. This is a proper record at its place.

Hence to place 14 we have to move down in search of an empty slot.

0	20
1	26
2	22
3	13
4	24
5	15
6	16
7	17
8	14
9	19

← 26 is placed over here as empty spa is available.

Next element 17 % 10 = 7. this positi was occupied by 14. But 14 is not the proper record at this pla Hence replace 14 by 17. Then place 14 at next empty slot.

Next element 26, but 26 % 10 = 6 & 16 is a proper record at this place.
Hence at next empty location 26 is placed

ii) Linear probing without replacement.

0	20
1	26
2	22
3	13
4	24
5	15
6	18
7	14
8	17
9	19

← Collision occurs at index 4. Hence probing 14 at the next empty slot.

↓ at index 7, the 14 is already placed. Hence at next empty slot 17 is placed.

↳ 26 % 10 = 6. The collision occurs. Hence 26 is placed at next empty slot.

2) Quadratic Probing :

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H_i(key) = (Hash(key) + i^2) \% m$$

where m can be a table size or any prime number.

eg. Hash table of size 10 & 37, 90, 55, 22, 11, 17, 49, 87.

	0	10
37 % 10 = 7	1	11
90 % 10 = 0	2	22
55 % 10 = 5	3	
22 % 10 = 2	4	
11 % 10 = 1	5	55
	6	
	7	37
	8	
	9	

Now if we want to place 17 a collision will occur as 17 % 10 = 7 and bucket-7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

$$H(key) = (Hash(key) + i^2) \% m$$

where $i = 0, 1, 2, \ldots$

$i = 0$, $(17 + 0)^2 \% 10 = 7 \rightarrow$ No

$i = 1$, $(17 + 1^2) \% 10 = 8$

So 17 will get placed at 8.

Then comes 49 which will be placed at index 9.

Now to place 37 ..

$(37 + 0^2) \% 10 = 7 \rightarrow$ No.

$(37 + 1^2) \% 10 = 8$ .. but already occupied.

$(37 + 2^2) \% 10 = 1$ .. already occupied

$(37 + 3^2) \% 10 = 6$ .. this slot is free.

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	17
9	49
10	

8) **Double Hashing**

- Double Hashing is technique in which a second hash fun^n is applied to the key when a collision occurs. By applying the second hash fun^n we will get the numbers of positions from the point of collision to insert.

2 rules for 2nd fun^n.

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

The formula →

H_1(key) = key % table size

H_2(key) = m − (key % m)

Double Hashing: Hash(key) = H_1(key) + H_2(key) ; not necessary.

where m is a prime number smaller than the size of the table.

e.g. Consider the elements,

37, 90, 45, 22, 17, 49, 55

H_1(37) = 37 % 10 = 7

H_1(90) = 90 % 10 = 0

H_1(45) = 45 % 10 = 5

H_1(22) = 22 % 10 = 2

0	90
1	
2	22
3	
4	
5	45
6	
7	37
8	
9	

Now if 17 is to be inserted then

H_1(17) = 17 % 10 = 7

H_2(17) = m − (key % m)

Here m is a prime number smaller than the table size. the prime number smaller than table size 10 is 7.

Hence m = 7.

$H_2(17) = 7 - (17 \% 7) = 7 - 3 = 4$.

That means we have to insert the element 17 at 4 places from 37. In short we have to take 4 jumps.

∴ the 17 will be placed at index 1.

How to insert 55.

$H_1(55) = 55 \% 10 = 5$ Collision

$H_2(55) = 7 - (55 \% 7) = 7 - 6 = 1$

That means we have to take one jump from index 5 to 55 finally the hash table will be

0	90
1	17
2	22
3	
4	
5	45
6	55
7	37
8	
9	49

4. **Rehashing** : Hash Table Overflow :

- It is a technique in which the table is resized, i.e. the size of the table is doubled by creating a new table. It is preferable if the total size of table is a prime number.
There are situations in which the rehashing is required.-

- When table is completely full.

- With quadratic probing when the table is filled half.

- When insertion fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by recomputing their positions using suitable hash fun

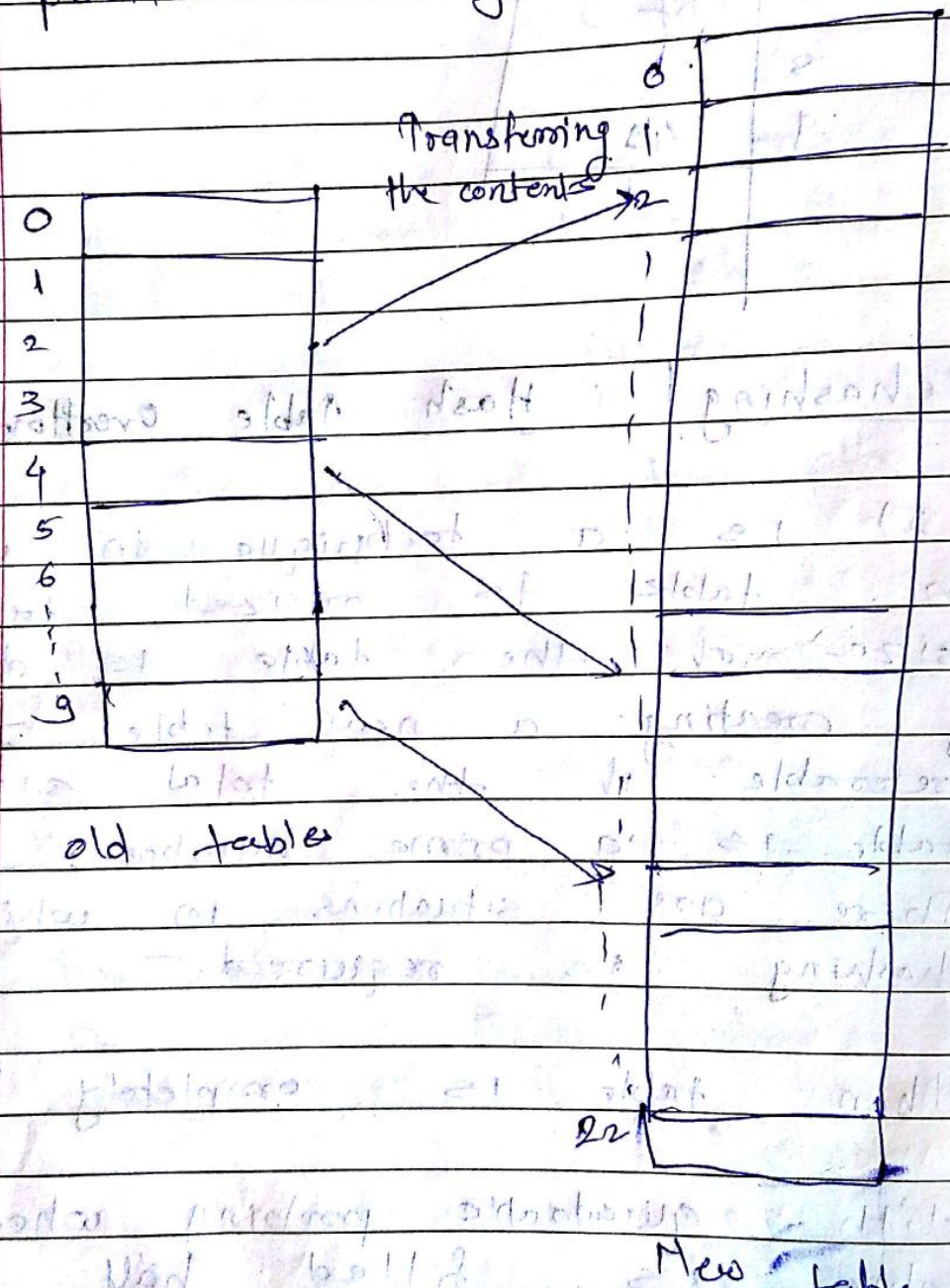


fig: Rehashing

e.g. 37, 90, 55, 22, 17, 49, 87

H(key) = key %. tablesize

37 %. 10 = 7

90 %. 10 = 0

55 %. 10 = 5

22 %. 10 = 2

17 %. 10 = 7 → collision solved by linear probing.

49 %. 10 = 9

87 %. 10 = 7

0	90
1	37
2	22
3	
4	
5	55
6	
7	87
8	17
9	49

Now this table is almost full & if we try to insert more & more element then overflow will occur. So will rehash by doubling the table size. So double is 20 but it is not prime no. so next is 23.

Then

87 % 23 = 14
90 % 23 = 21
55 % 23 = 9
22 % 23 = 22
17 % 23 = 17
49 % 23 = 3
87 % 23 = 18

0	
1	
2	
3	49
4	
5	
6	
7	
8	
9	55
10	
11	
12	
13	
14	87
15	
16	
17	17
18	87
19	
20	
21	90
22	22

W. Examples

1. Give the input (4371, 1323, 6173, 4199, 4344, 9679, 1989) & hash fun hex = x mod 10, show the

results for the following.

i) Open addressing hash table using linear probing.

ii) Open addressing hash table using quadratic probing.

iii) Open addressing hash table with 2nd hash fun^n $h^2(x) = 7 - (x \bmod 7)$

// program for Linear probing

```
// hash fun^n to get position.
int hash(int key)
{
  return (key % max);
}

// fun^n to insert new record.
int linear_probe(int hash table[], int key)
{
  int pos, i;
  pos = hash(key);
```

```
if ( Hashtable [pos] == 0)   // empty
{
  Hashtable [pos] = key;
  return  pos;
}
else
{
  for (i= pos+1 ; i (%max) != pos ; i++)   (no need)
  {
    if ( Hashtable [i] == 0)
    {
      Hashtable [i] = key;
      return  i;
    }
  }
}
return  -1 ;   // overflow
}
```

```
struct node
{
  int data;
  node *next;
} h [10];
```

2] Chaining [Linked List] :

In this technique, we can store the linked list inside the hash table, in the unused slot. This is used to handle synonyms is chaining; it contains together all the records that hash to the same address. Instead of relocating synonyms, a linked list of synonyms is created whose head is home address of synonyms. As involvement of pointers is there so extra memory is needed for storing pointers.

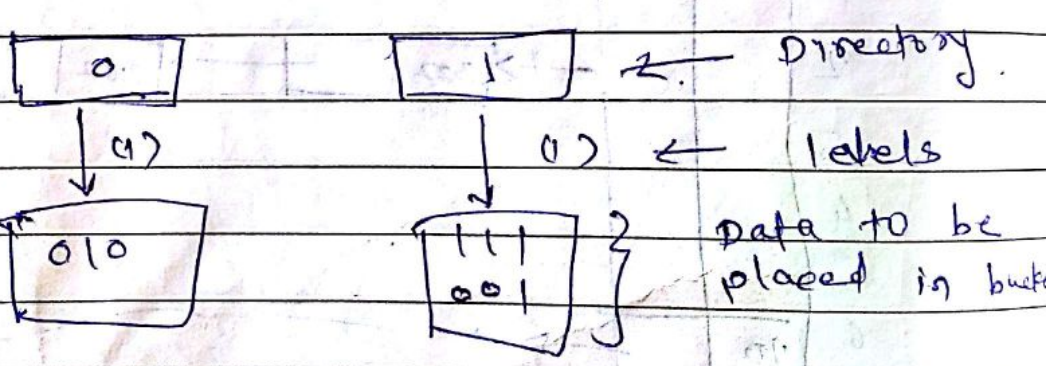
address.

Extendible Hashing :

- Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.

- In extensible hashing referring the size of directory the elements are to be placed in buckets The levels are indicated in parenth

eg.



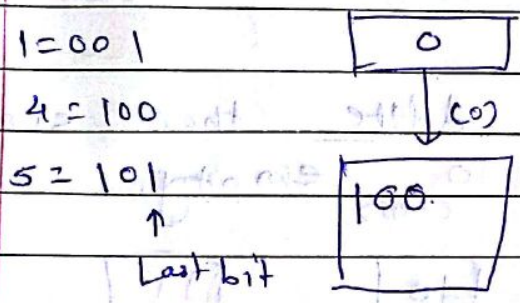
- The bucket can hold the dat of its global depth.

If data in bucket is more than global depth then, split the bucket and double the directory.

Insertion operation:

eg. to insert 1, 4, 5, 7, 8, 10. Assume each page can hold 2 data entries (2 is the depth).

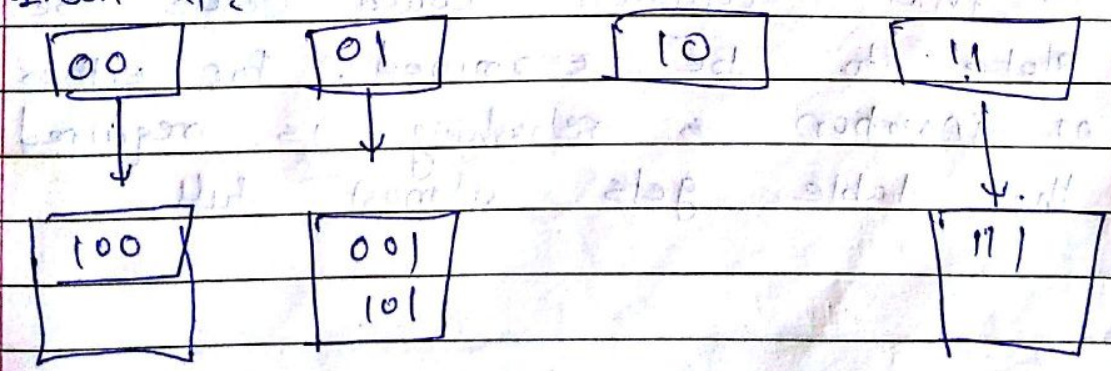
Step 1: Insert 1, 4.



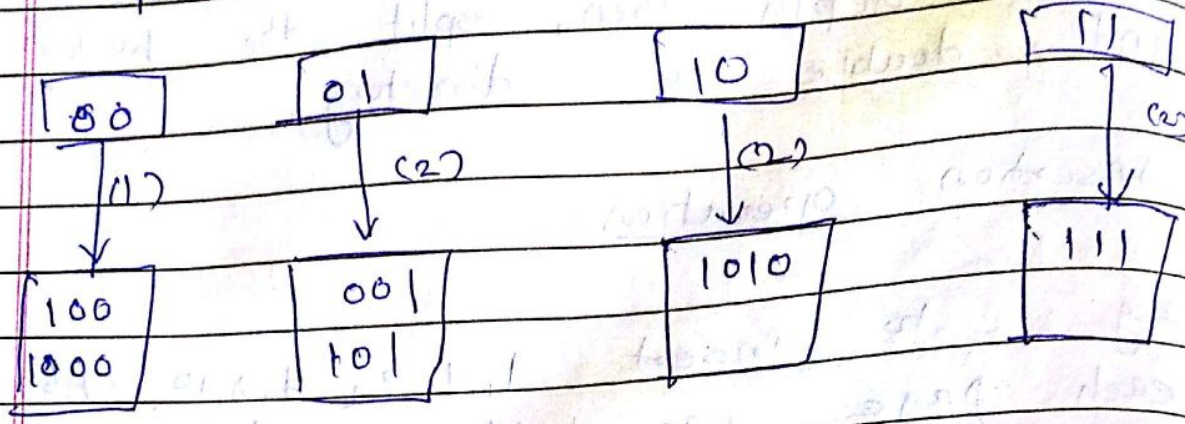
We will examine last bit of data & insert the data in bucket.

The bucket is full. Hence double the directory.

Insert 7:



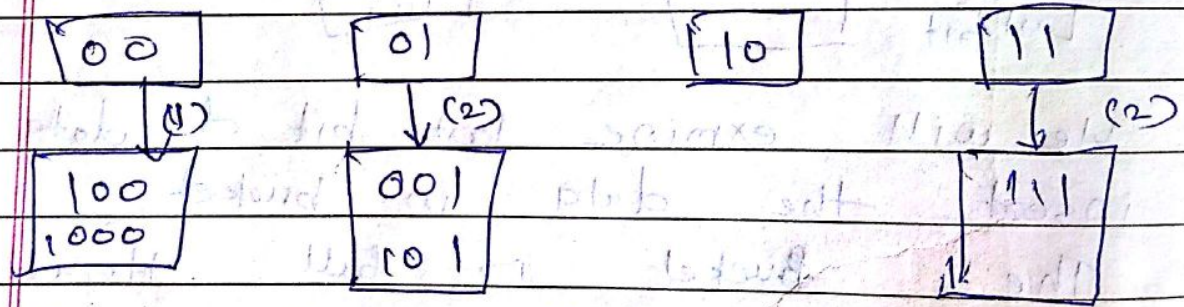
Step 3 & 4: Insert 8 & 10. i.e. 1000 & 1010



Thus data is inserted using extensible hashing.

Deletion Operation:

If 10 is to be delete, then simply make bucket of 10 (item) empty.



- The problem with open & closed Hashing is that collision could cause several slots to be examined. For either scheme insertion & rehashing is required when the table gets almost full.

Closed Addressing:

[Chaining without Replacement]

e.g. 131, 3, 4, 21, 61, 6, 71, 8, 9

Index	key	chain
0	-1	-1
1	131	2
2	21	5
3	3	-1
4	4	-1
5	61	7
6	6	-1
7	71	-1
8	8	-1
9	9	-1

Chaining with Replacement

e.g. 131, 21, 31, 4, 5, 2,

Index	key	chain
0	-1	-1
1	131	6
2	2+2	$-1
3	31	-1
4	4	-1
5	5	-1
6	21	3
7	-1	-1
8	-1	-1
9		

* Appl^n of Hashing :

1. In Compilers to keep track of declared variables.

2. For online spelling checking hashing functions are used.

3. Hashing helps in Game playing to store more moves.

4. For browser program while each the web pages, hashing is

* Dictionary : is a collection of key and value where each value is associated with the corresponding key.

* Load Density : It is denoted by

$$\alpha = \frac{n}{b}$$

n = no. of keys
b = no. of buckets.

It should be less than = 0.5

n = 5, b = 10

$$\frac{5}{10} = 0.5$$

* **Dictionary** :

is a collection of pairs of key and value where every value is associated with the corresponding key.

e.g. Library Database.

ADT for Dictionary

{

instance : Dictionary is pair of key & value.

Operations :

size ();

Insert ();

delete ()

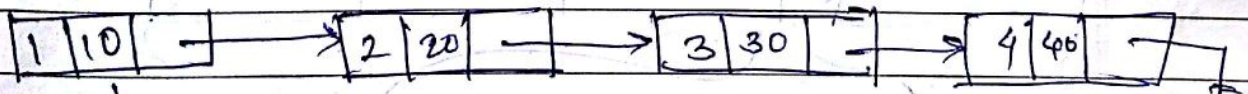
search ();

}

* **Representation of Dictionary** :

1) Sorted array

2) Sorted chain (Linked List).



Key ↓ value

```
struct node
{
  int key;
  int value;
  node *next;
};
```

Diff^n operat^n on Dictionary.

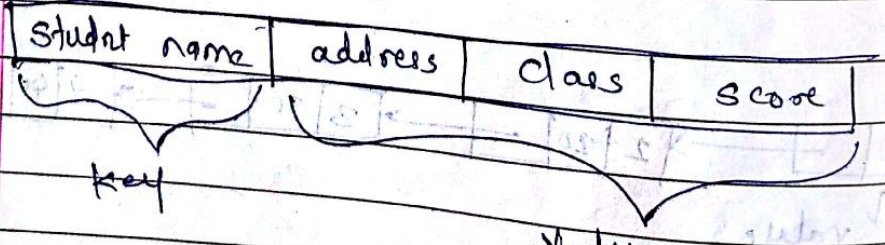
1) Insert
2) Delete
3) Search
4) size.

All operations same as linked l

* Appl^n of Dictionary:

1) student registration appl^n:

A record of student can be i
deleted or updated from the dictio
When a new student registers for
course his/her record is inserted
the dictionary.

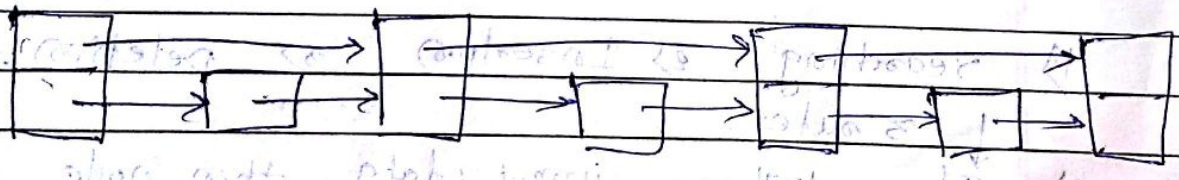


Identifiers - variable, fun, arrays, struct.

2) Telephone directory, Dictionary
3) Word dictionary
4) Symbol Table

* Skip List

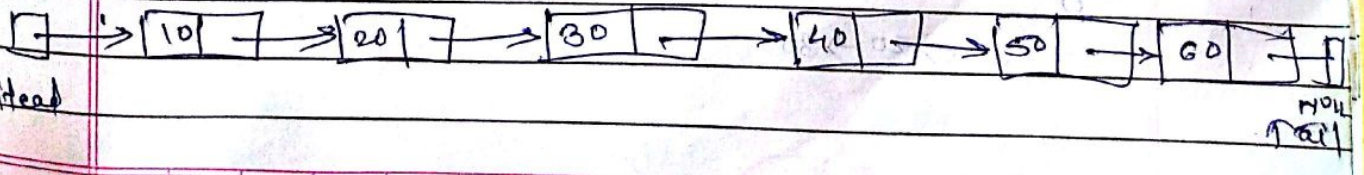
- Skip list is a variant list for the Linked list.
- Skip lists are made up of a series of nodes connected one after the other.
- The no. of references each node contains is determined randomly. The no. of references a node contains is called its node level.
- There are two special nodes in the skip list one is head node which is the starting node of the list & tail node is the last node of the list.



Head Node

Tail node.

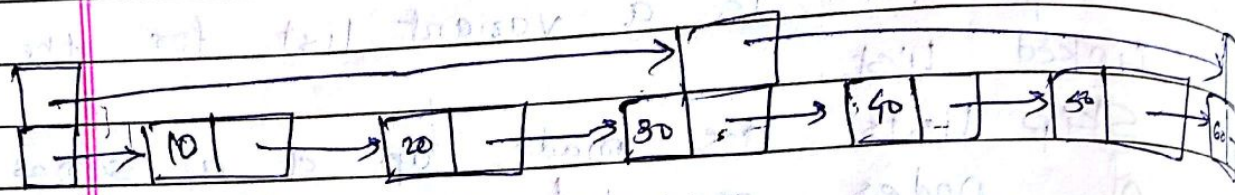
Consider a sorted chain as



Head

Tail

Now to search any node from above sorted chain we have to search the sorted chain from head node by visiting each node. But this searching time can be reduced if we add level in every alternate node. This extra level contains the forward pointer of same node.



Take one more

Add one more level

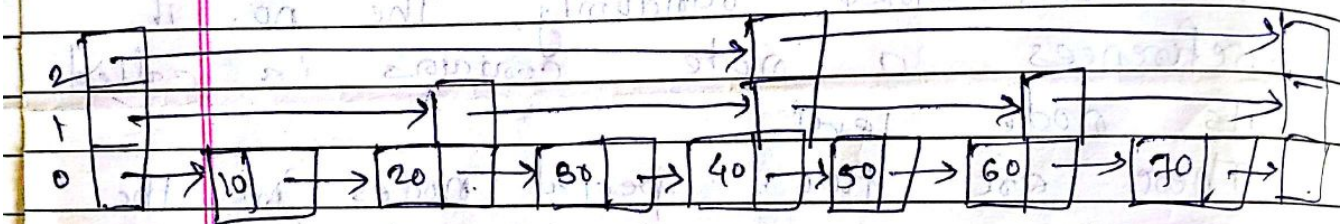


fig. Skip list.

* Operations on Skip List :

1) Searching. 2) Insertion 3) Deletion.

↓ 3 rules.

1) If key == current → data, then node is found.
2) If key < current → data, go down one level.
3) If key > current → data, go right along the same level.

e.g. above e.g. search. 20

→ 20 → 30.

Hashing - Extra

Explain linear probing with & w/o replacement using the following data

12, 01, 04, 03, 07, 08, 10, 02, 05, 14, 06, 28.

Assume bucket from 0 to 9 & each bucket has one slot. Calculate average cost/no. of comparision for both.

Soln: We assume hash fun^n = key mod 10. As buckets are from 0 to 9 & each bucket has one slot. the keys can be inserted in the hash table using our hash function as follows.

Index	key
0	10
1	01
2	12
3	03
4	04
5	02
6	05
7	07
8	08

Now hash table is full. We can not insert 06 & 28 in the hash table

key	12	01	04	03	07	08	10	02	05	14	06	28
No. of Comparisons	1	1	1	1	1	1	1	4	02	6	10	10

Thus total no. of comparisons =

= 1+1+1+1+1+1+1+4+02+6+10+10

= 39 buckets are examined.

Load density $\alpha = \frac{n}{b}$

$= \frac{12}{10} = 1.2.$

The avg. cost can be compu by calculating successful search (S_n) & unsuccessful sear (U_n).

$S_n = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$.

$\frac{1}{2}\left(1 + \frac{1}{(1-1.2)}\right)$

$S_n = -2$

$$U_n = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

$$= \frac{1}{2}\left(1 + \frac{1}{(1-1.2)^2}\right)$$

$$= \frac{1}{2}\left(1 + \frac{1}{0.04}\right)$$

$$= 13$$

Total cost $= S_n + U_n$

$= -2 + 13$

$= 11$.