M. Prasanth

UNIT-II

Hashing

Hashing:

Hashing is the process of mapping large amount of data item to a smaller table with the helping of hashing function.

Adv: This method is used to handle the vast amount of data

Hash Table:

* Hash Table is a data structure used for storing and retrieving data very quickly.
* Insertion of data in the hash table is based on the key value.
* Every Entry in the hash table is associated with some ke

Ex: storing an employee record in the hash table, employee ID will work as a key.

* Using the hash key the required piece of data can be searched in the hash table by few (or) more comparisons.

The searching time dependent upon the size of the hash table

Hash Function:

* Hash Function is a function which is used to put the data i the hash Table. Hence one can use the same function to retrieve the data from the hash Table.

Thus hash function is used to implement the hash Ta

* The integer returned by the hash function is called Hash key.

Hash key (H) = key % Table size

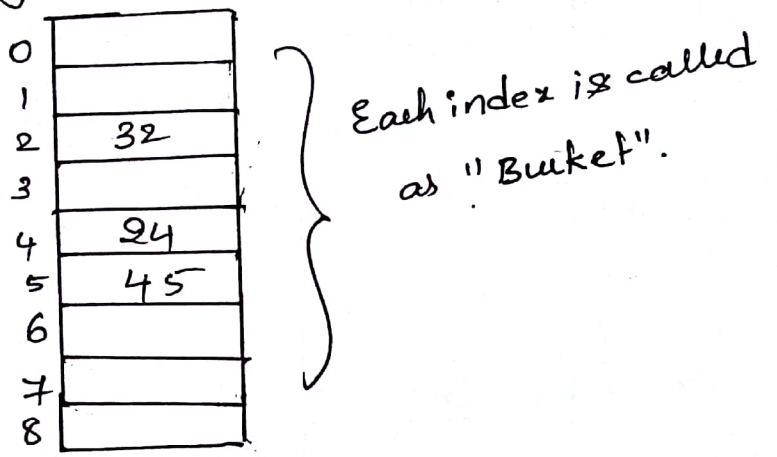
Ex:- 24, 32, 45, ...

Table size = 10

Hash key(H) = 24 % 10 = 4.

Hash key(H) = 32 % 10 = 2

Hash Key(H) = 45 % 10 = 5



Types of Hash Functions:

There are various types of hash functions that are used to place the record in the hash Table.

1. Division Method
2. Mid- Square Method
3. Multiplicative Hash Function.
4. Digit Folding.

Division Method:

The hash function depends upon the remainder of division.

Ex: 22, 24, 26, 89, 75, . . .

Table size = 10.

H(key) = key % Table size

H(22) = 22 % 10 = 2

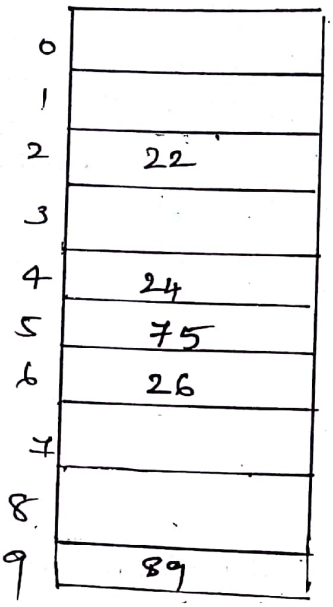
where 'p'
if key

H(24) = 24 % 10 = 4

H(26) = 26 % 10 = 6

H(89) = 89 % 10 = 9

H(75) = 75 % 10 = 5



Mid-Square Method:

In mid square method the key is squared and middle (or) mid part of the result is used as index.

Consider, key = 3111; then..

find Square of a given number

$(3111)^2 = 96\,(783)\,21$

Middle num.

The Hash index is 783, then place 3111 into 783 pos

Multiplicative Hash Function:

* In multiplicative hash function the given record is multiplied with some constant value.

$$H(Key) = floor(p * (key * A))$$

where 'p' is integer constant, 'A' is constant real number.

if key=107, and p=50 then.

H(key) = floor (P * (key * A)) → Author Donald knuth Suggested A = 0.618033..

= floor (50 * (107 * 0.618033))

= floor (3306.481..)

H(key) = 3306

At location 3306, the record 107 will be placed.

Digit folding:

The key is divided into seperate parts and using simple function, these parts are combined to produce the hash key.

for Ex: num = 1237654

↳ if you want to place the given record into hash table.

first divide the given number into parts.

123, 765, 4..

Now combine = 123 + 765 + 4 = 892

"892" will be the index to place the record "i.e 1237654".

Collision:

* The hash function is a function that returns the key value using which the record can be placed in the hash table

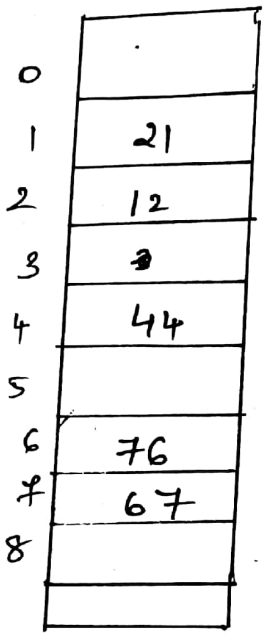
* The function needs to be designed very carefully and it should not return the same hash key address for two different records.

Definition:-

The hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different record is called collision.

Ex: 12, 21, 32, 44, 76, 67, 86...

Assume Table size = 10.



H(12) = 12 % 10 = 2

H(21) = 21 % 10 = 1

H(32) = 32 % 10 = 2 * collision occured.

H(44) = 44 % 10 = 4

H(76) = 76 % 10 = 6

H(67) = 67 % 10 = 7

H(86) = 86 % 10 = 6 *

collisior occured

To avoid the collision, have some resolution techniques.

Collision Resolution Techniques:

If collision occur then it should be handled by applying some teihniques.

1. Chaining
2. open addressing (Linear probing)
3. Quadratic probing
4. Double hashing.

Chaining:

In collision handling Method Chaining is a concept which introduces an additional field with data i.e Chain.

Seperate chain table is maintained for colliding data. (4
When collision occurs then a linked list(chain) is maintaind at the home bucket.

for Ex: 121, 32, 42, 67, 76, 41, 82, 89, 19...

Table size = "10"

H(Key) = 121 % 10 = 1

H(32) = 32 % 10 = 2

H(42) = 42 % 10 = 2 *

H(67) = 67 % 10 = 7

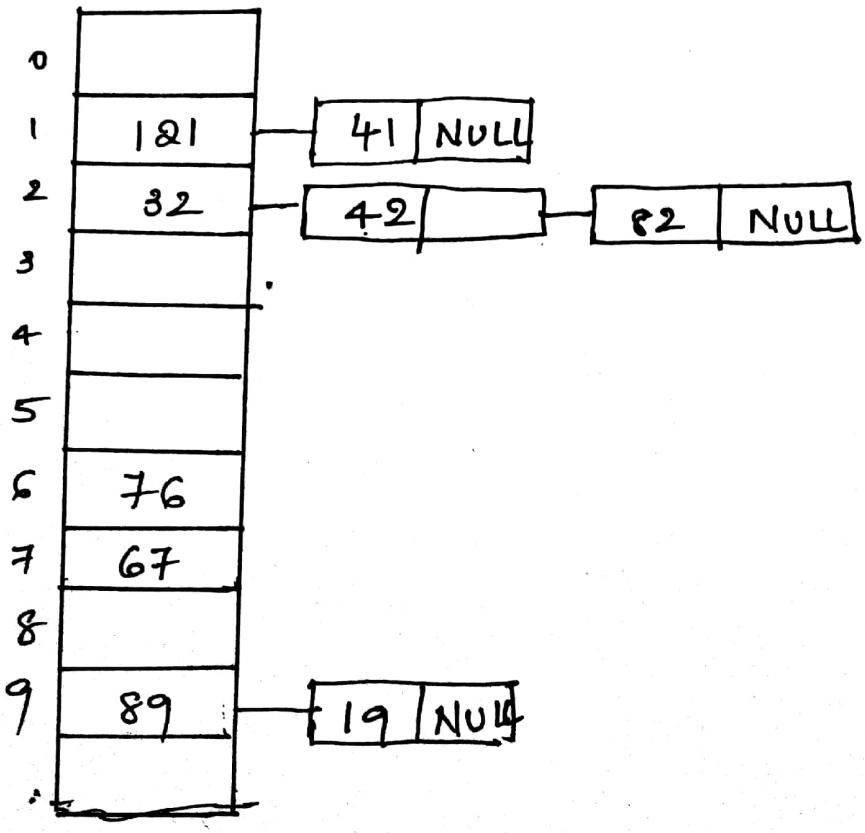
H(76) = 76 % 10 = 6

H(41) = 41 % 10 = 1 *

H(82) = 82 % 10 = 2 *

H(89) = 89 % 10 = 9

H(19) = 19 % 10 = 9 *



HashTable with chaining.

Open-addressing: (Linear probing).

* Easiest Method of handling collision.
* when collision occurs, i.e when two records demand for the same bucket in the hash table then collision can be solved by placing "the second record linearly down whenever the empty bucket is found."

Ex: Records = 31, 42, 86, 99, 45, 32, 76, ...

H(key) = key % Table size

H(31) = 31 % 10 = 1
H(42) = 42 % 10 = 2
H(86) = 86 % 10 = 6
H(99) = 99 % 10 = 9
H(45) = 45 % 10 = 5
H(32) = 32 % 10 = 2 *
H(76) = 76 % 10 = 6 *

0	NULL
1	31
2	42
3	32
4	NULL
5	45
6	86
7	76
8	NULL
9	99

"32" but already filled with some other num. so go linearly down. next bucket is free, so insert 32 into 3rd bucket

Problem with Linear probing:

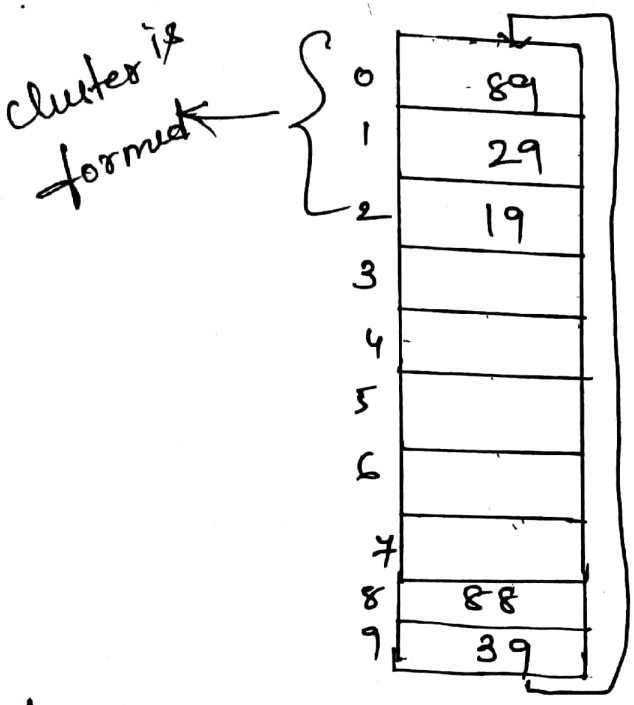
Problem with Linear probing is primary clustering.

Primary Clustering:

↳ it is a process in which a block of data is formed in the hash table when collision resolved.

Ex: 39, 89, 29, 19, 88.....

Table size = 10.



H(39) = 39 % 10 = 9
H(89) = 89 % 10 = 9
H(29) = 29 % 10 = 9
H(19) = 19 % 10 = 9
H(88) = 88 % 10 = 8

Quadratic Probing:

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

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

Ex: 15, 22, 27, 84, 66, 37, ...

Table size = 10.

H(15) = 15 % 10 = 5

H(22) = 22 % 10 = 2

$H(27) = 27 \% 10 = 7$

$H(84) = 84 \% 10 = 4$

$H(56) = 56 \% 10 = 6$

$H(37) = 37 \% 10 = 7$ * (collision)

7th bucket has already an element. Hence apply Quadratic probing.

$$H(Key) = Has(Key + i^2) \% m$$

M can be table size (or) prime number.

Consider $i = 0$, then

$$H(Key) = (37 + 0^2) \% 10 = 7.$$

When $i = 1$,

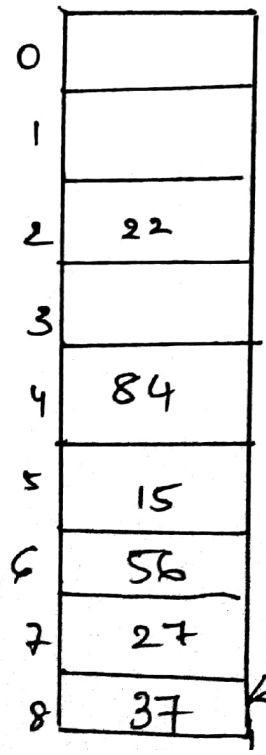
$$H(Key) = (37 + (1)^2) \% 10$$

$$= (37 + 1) \% 10$$

$$= 38 \% 10$$

$$= 8$$

8 th bucket is empty so, simply insert '37' into 8 th bucket, if 8th bucket is not empty then continue with $i = 2, 3, \ldots$



'37' inserted into '8' th bucket by using "Quadratic probing"

Double Hashing:

Double Hashing is technique in which a second hash function is applied to the key when a collision occurs.

$$H_1(key) = key \,\%\, Table\ size$$
$$H_2(key) = M - (key \,\%\, M)$$

M is a prime number smaller than the size of the table.

Eg: 37, 49, 25, 62, 33, 17. . . .

Table size = 10

$H(37) = 37 \,\%\, 10 = 7$

$H(49) = 49 \,\%\, 10 = 9$

$H(25) = 25 \,\%\, 10 = 5$

$H(62) = 62 \,\%\, 10 = 2$

$H(33) = 33 \,\%\, 10 = 33$

$H(17) = 17 \,\%\, 10 = 7$ ← collision occured.

$H_2(key) = M - (key \,\%\, M)$

Table size 10, Take 'M' as nearest prime number to the table size, M = 7.

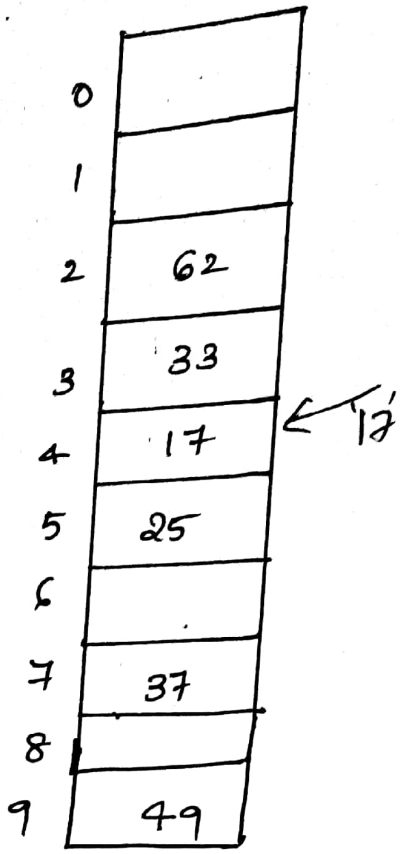
$H_2(17) = 7 - (17 \,\%\, 7)$

$= 7 - (3)$

$= 7 - 3 = 4$

$H_2(17) = 4$

Insert 17 into 4th bucket.



Rehashing:

* Rehashing is a technique in which table is resized. i,e the size of the table is doubled by creating a new table.

* There are three situations in which rehashing is required.

  * when table is completely full
  * with Quadratic probing when the table is filled half.
  * when insertions fail due to Overflow.

eg: consider some elements...

37, 90, 55, 22, 17, 49 & 87 . . . . .

H(Key) = Key % Table size.

H(37) = 37 % 10 = 7.
H(90) = 90 % 10 = 0

H(55) = 55 % 10 = 5

H(22) = 22 % 10 = 2

H(17) = 17 % 10 = 7 ~~*~~

H(49) = 49 % 10 = 9

H(87) = 87 % 10 = 7 Collision occured.

make

low Double the table size. Nothing but 20 !

Here '20' is not a prime number.

so, '23' as new table size.

Now calculate Hash keys with new table size.

H(37) = 37 % 23 = 14

H(90) = 90 % 23 = 21

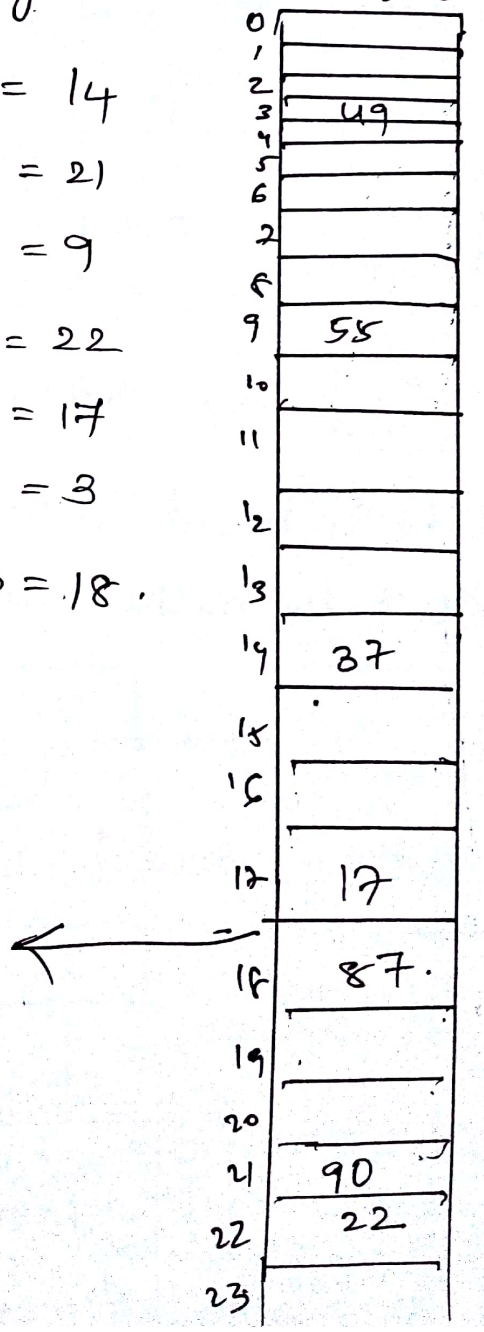
H(55) = 55 % 23 = 9

H(22) = 22 % 23 = 22

H(17) = 17 % 23 = 17

H(49) = 49 % 23 = 3

H(87) = 87 % 23 = 18.

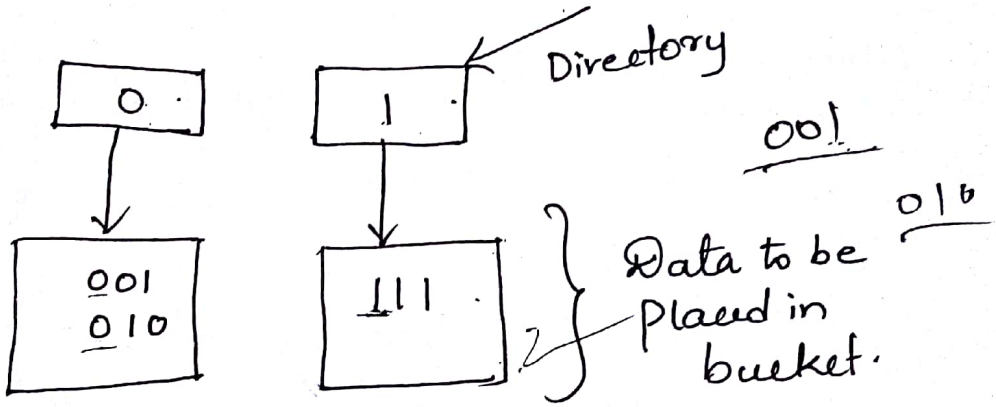


Now the hash table is sufficiently large to accomodate new insertions.

Extendible Hashing:

* Extendible 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 Extendible hashing referring the size of directory the elements are to be placed in buckets.
* ~~The terms~~

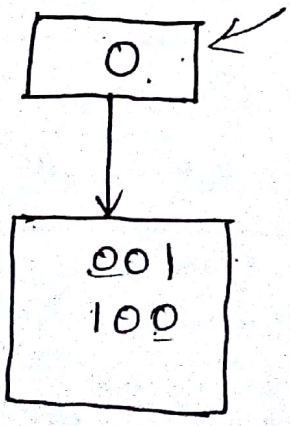
eg:



The bucket can hold the data of its global depth. If data in bucket is more than global depth then, Split the bucket and double the directory.

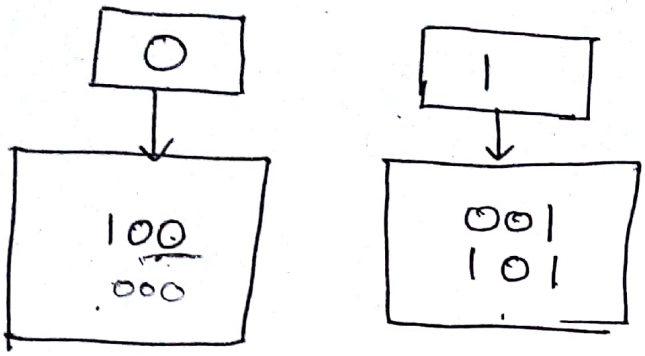
eg: consider, 1, 4, 5, 7, 8 & 10; (2 is depth).

step 1: insert 1, 4.



1 = 001
4 = 100

Step 2: insert '5', the bucket is full, Hence double the directory.

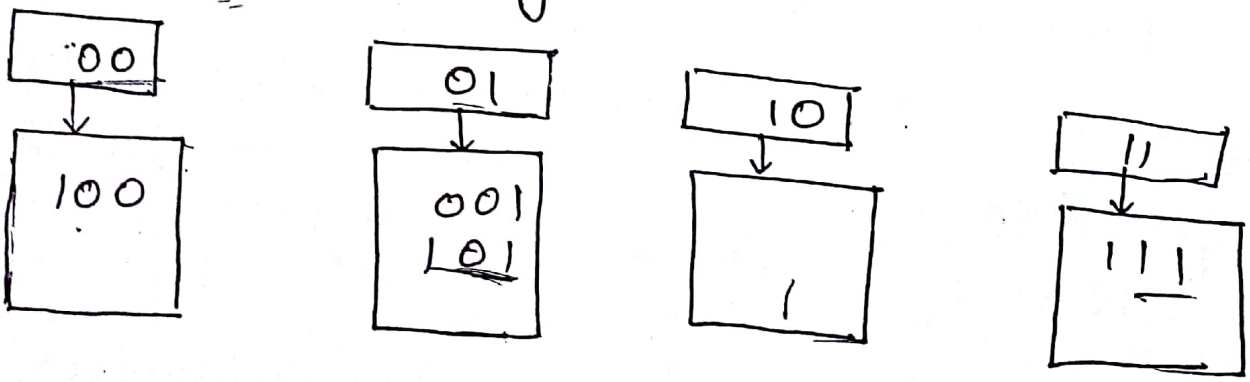


1 = 001
4 = 100
5 = 101

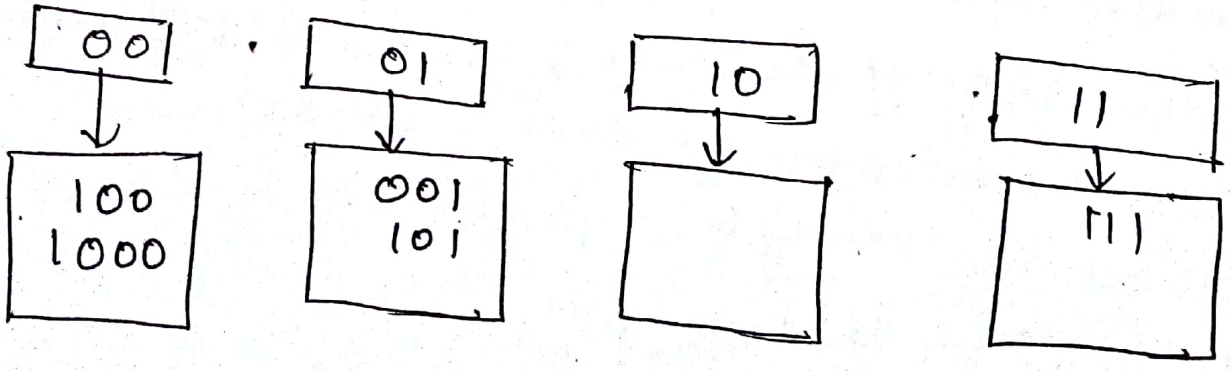
Based on $2^2 \Rightarrow 4$ last bit data is inserted.

Step 3: Insert '7', 7 = 111

But the depth is full, Hence double the directory, split the bucket.



Step 4: Insert 8, 8 = 1000.



Thus the data is inserted using Extendible hashing.

Applications of Hashing:

* In compilers to keep track of declared variables
* For online spelling checking the hash functions are used.
* Hashing Helps in game playing programs to store the moves made.
* For browser program while caching the web pag hashing is used.

Perfect Hash Functions:

* Difference between General hash function and Perfect Hash function is, Here we are defining the hash functions to generate the Hash keys.
* In General, hash function was first devised and then the data were processed.
* If the body of the data is fixed and a hash function can be devised after the data are known. such a function may really be a perfect hash function. if it has items on the first attempt.
* perfect hash functions may be used to implement a look-up table with constant worst case time.

hash function is one that maps
key values to the table without any collisions.

Chelli's Method:

This is used primarily when it necessary to hash elatively small collection of keys.

$$h(word) = length(word) + g(firstLetter(word)) + g(lastLetter(word)) \bmod Tablesize$$

where g() is constructed using Sichelli's method so that h() will return a different hash value for each word in the set.

Algorithm:

Choose a value for max.

compute the number of occurances of each first and last letter in the set of all words;

order all words in accordance to the frequency of the first and last letters;

```
Search (wordList)
    if wordList is empty
        halt;
    word = first word from wordList;
WordList = wordList with the first word deleted;
if the first and last letters of word are assigned
        g-values.
```

try (word, -1, -1); // -1 signifies value assigned g-value

if success
Search (wordList);
Put the word at the beginning of wordList and de
its hash value

else if neither the first nor the last letter has a g-value.

for each n,m in {0,.. max}.
try (word, n, m);
if success
Search (wordList);

put word at the beginning of wordList and detach its hash value;

else if either the first or the last letter has a g-value

for each n in {0, ..., max}

try (word, -1, n) or try (word, n, -1)

if success

Search (wordlist);

put word at the begining of wordList and detach its hash value;

try (word, firstLetter value, LastLetter value)
If (h(word) has not been claimed
release h(word).

assign firstLetterValue and/or lastLetterValue as g-values of firstLetter(word) and/or LastLetter(word) if they are not −1;

return success;
return failure:

[Determine frequency with which each first & last letter occurs]

Ex: Take some words.

Calliope 8
Clio 4
Erato 5
Euterpe 7
Melpomene 8
Polyhymnia 10
Terpsichore 9
Thalia 6
Urania 6

[find score of the words by summing the frequencies of the first letter and last letters, and then sort them in that order.]

Letter	e	a	c	o	t	m	p	u
frequency count:	6	3	2	2	2	1	1	1

⇓ final data

Length:

Euterpe → 7
calliope → 8
erato → 5
Terpsichore → 11
melpomene → 9
Thalia → 6
clio → 4
polyhymnia → 10
urania → 6

c=3
erato
c e

Sphere

6 + g(c) + g(e)
Sph: 6+0+0
⇒ 4+0+0 ⇒ 4
⇒ 5 + g(c) + g(e) 0 to 4
= 5+3+0 ⇒ —
5+3+1

1) Euterpe ⇒ h(word) = length(word) + g(firstLetter(word)) + g(LastLetter(word)) % Table size.

⇓

h(Euterpe) = (g(E) + g(e)) % 9

= (7 + 0 + 0) % 9

= 7 % 9

h(Euterpe) = 7

2) calliope ⇒

h(word) = (h(calliope) + g(c) + g(e)) % 9

= (8 + 0 + 0) % 9

= 8 % 9

h(calliope) = 8

3) Now Erato

h(Erato) = (l(Erato) + g(E) + g(o)) % 9

= (5 + 0 + 0) % 9

= 5 % 9.

h(Erato) = 5

4) Terpsichore ⇒

h(Terpsichore) = (l(Terpsichore) + g(t) + g(e)) % 9

= (11 + 0 + 0) % 9

= 11 % 9

h(Terpsichore) = 2

h(Melpomene) = (l(melphomene) + g(M) + g(e)) % 9

= (9 + g(o) + g(o)) % 9

= 9 % 9

h(Melpomene) = 0

6) h(Thalia) = (l(Thalia) + g(T) + g(a)) % 9

= (6 + 0 + 0) % 9

= 6 % 9

h(Thalia) = 6

7) h(clio) = (l(clio) + g(c) + g(o)) % 9

= (4 + 0 + 0) % 9

= 4 % 9

h(clio) = 4

8) h(polyhymnia) = (l(polyhymnia) + g(p) + g(a)) % 9

= (10 + 0 + 0) % 9

= 10 % 9

h(polyhymnia) = 1

9) h(urania) = l(urania) + g(u) + g(a) % 9

= (6 + 0 + 0) % 9

= 6 % 9

= 6 * collision occurred.

$h(\text{urania}) = (l(\text{urania}) + g(1) + g(0)) \% 9$

$= (6 + 1 + 0) \% 9$

$= 7 \% 9$

$= 7$ * collision.

$v = 2$:

$h(\text{urania}) = (l(\text{urania}) + g(2) + g(0)) \% 9$

$= (6 + 2 + 0) \% 9$

$= 8 \% 9$

$= 8$ * collision

$v = 3$

$h(\text{urania}) = (l(\text{urania}) + g(3) + g(0)) \% 9$

$= (6 + 3 + 0) \% 9$

$= 9 \% 9$

$= 0$ * collision

$v = 4$

$h(\text{urania}) = (l(\text{urania}) + g(4) + g(0)) \% 9$

$= (6 + 4 + 0) \% 9$

$= 10 \% 9$

$= 1$ * collision.

g possible values is from 0 to 4.

for
urania, all possibus cases collision is occurred, so. Now apply backtracking concept.

↓

Moves to previous word.

Remove previous word from hash table, and calculate hash(word) with different values.

i,e we have to re calculate polyhymnia with different values.

$h(\text{polyhymnia}) = (d(\text{polyhymnia}) + g(1) + g(0)) \% 9$

$P=1 \quad = (10+1+0) \% 9$

$= 11 \% 9$

$= 2$ * collision

$P=2 \Rightarrow \quad = (d(\text{polyhymnia}) + g(2) + g(0)) \% 9$

$= (10+2+0) \% 9$

$= 12 \% 9$

$= 3$ No collision, so

now insert polyhaymnia into 3rd bucket..

Now calculate urnia,

$h(\text{word})$

$h(\text{urania}) = (d(\text{urania}) + g(0) + g(0) \% 9)$

$= (6+0+0) \% 9$

$= 6 \% 9 = 6$ * collision.

$u=1 \Rightarrow \quad (d(\text{urania}) + g(1) + g(0)) \% 9$

$= (6+1+0) \% 9$

$= 7 \% 9$

= 7 * collision

Now

u = 2

h(urania) = (l(urania) + g(2) + g(0))%.9

= (6 + 2 + 0) %.9

= 8%.9

h(urania) = 8 → X collision

U = 3

h(urania) = (l(urania) + g(3) + g(0)) %.9

= (6 + 3 + 0) %.9

= 9%.9

h(urania) = 0 * collision

U = 4

h(urania) = (l(urania) + g(4) + g(0)) %.9

= (6 + 4 + 0) %.9

= 10%.9

h(urania) = 1

→ No collision, so insert urania in 1st index.

⇓

Then we:

Reserved hash values.

Euterpe	E=0, h=7	{7}
calliope	c=0, h=8	{7,8}
Erato	0=0, h=5	{5,7,8}
Terpsichore	T=0, h=2	{2,5,7,8}
Melpomene	M=0, h=0	{0,2,5,7,8}
Thalia	A=0, h=6	{0,2,5,6,7,8}
clio	h=4	{0,2,4,5,6,7,8}
polyhymnia	P=0, h=1	{0,1,2,4,5,6,7,8}
urania	u=0, h=6 *	{0,1,2,4,5,6,7,8}
urania	u=1, h=7 *	{0,1,2,4,5,6,7,8}
urania	u=2, h=8 *	{0,1,2,4,5,6,7,8}
urania	u=3, h=0 *	{0,1,2,4,5,6,7,8}
urania	u=4, h=1 *	{0,1,2,4,5,6,7,8}
polyhymnia	P=1, h=2 *	{0,2,4,5,6,7,8}
polyhymnia	P=2, h=3	{0,2,3,4,5,6,7,8}
urania	u=0, h=6 *	{0,2,3,4,5,6,7,8}
urania	u=1, h=7 *	{0,2,3,4,5,6,7,8}
urania	u=2, h=8 *	{0,2,3,4,5,6,7,8}
urania	u=3, h=0 *	{0,2,3,4,5,6,7,8}
urania	u=4, h=1	{0,1,2,3,4,5,6,7,8}

FHCD Algorithm:

An extension of Sichelli's approach was proposed by Thomas Sager.

$$h(word) = h_0(word) + g(h_1(word)) + g(h_2(word))$$

where 'g' is the function to be determined by the algorithm. To define the functions h_i, three tables T_0, T_1 & T_2 of random numbers are defined.

$$h_0 = (T_0(c_1) + \dots + T_0(c_m)) \bmod n$$

$$h_1 = (T_1(c_1) + \dots + T_1(c_m)) \bmod n$$

$$h_2 = (C\,T_2(c_1) + \dots + T_2(c_m)) \bmod n) + n$$

where 'n' is number of all words in the body of data, 'n' is a parameter usually equal to $n/2$ (or) less. and $T_i(c_j)$ is the number generated in Table T_i for c_j.

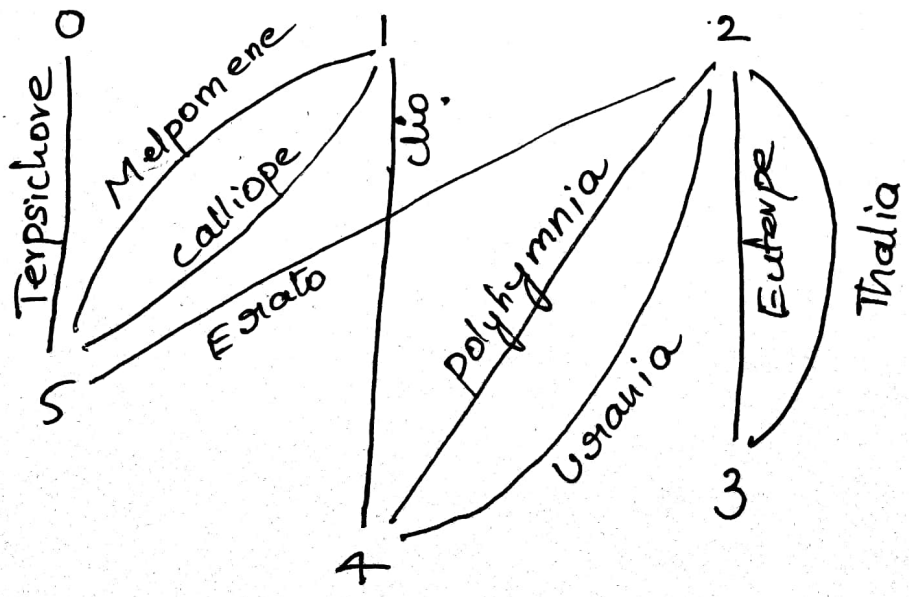
The function 'g' found in three steps.

mapping, ordering and searching.

In mapping step 'n' triples $(h_0(word), h_1(word), h_2(word))$ are created. The randomness of function h_i usually guarantees the uniqueness of these triples; should they not be unique, new Tables T_i are generated.

Value of	h_0	h_1	h_2
Calliope	(0	1	5)
Clio	(7	1	4)
Erato	(3	2	5)
Euterpe	(6	2	3)
Melpomene	(3	1	5)
polyhymnia	(8	2	4)
Terpsichore	(8	0	5)
Thalia	(8	2	3)
Urania	(0	2	4)

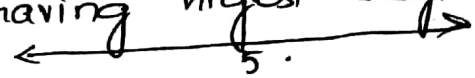
A dependency Graph is built, it is a bipartie graph with half of its corresponding to the h_1 values and labeled 0 through $r-1$; and other half of the h_2 values and labeled r' through $2r-1$



ordering step:

* In ordering step, first select highest vertex, based on degree.

1. Here vertex "2" is having higest degree.



" "

Now consto 2 associated vertexes are 3, 4, 5, and find degree of each vertex.

3 degree = 2

4 degree = 3

5 degree = 4

Now draw vertex in between 2 —— 5. and name according h_1 and h_2 values.

Same procedure for all nodes.

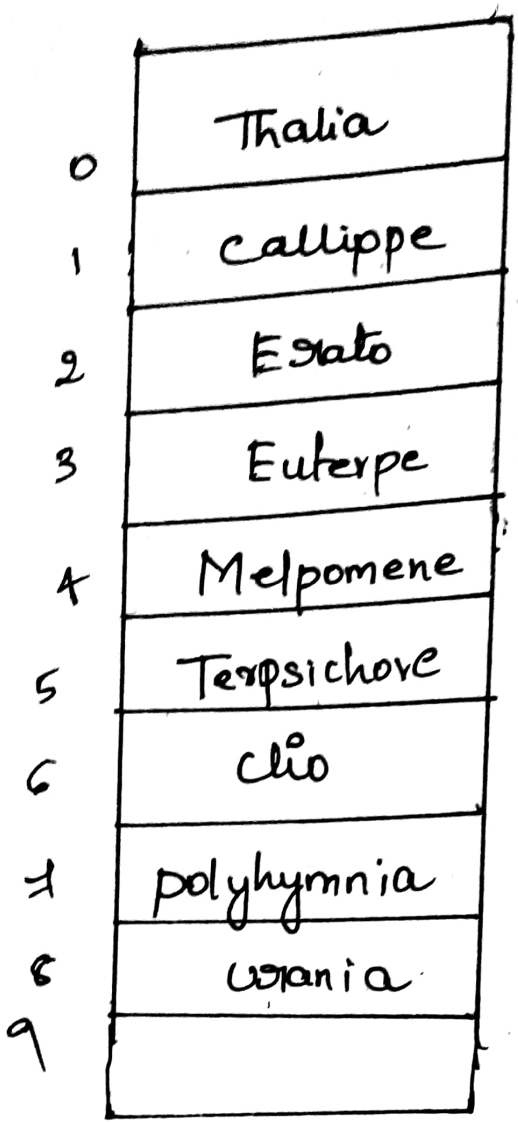
Level	Node	Arcs
0	2	2
1	5	Erato
2	1	Calliope, Melpomene
3	4	Clio, Polyhymnia, Urania
4	3	Euterpe, Thalia
5	0	Terpsichore.

the last step, searching hash values are assigned to keys level by level.

The g-value for the first vertex is chosen randomly among the numbers $0, \ldots, n-1, 0$

Level	Vertex	g-value	
0	2	2	
1	5	6	h(Erato) = (3+2+6)%9 = 2.
2	1	4	h(calliope) = (0+4+6)%9 = 1
2	1	4	h(Melpomene) = (3+4+6)%9 = 4
3	4	2	h(clio) = (7+6+2)%9 = 6
3	4	2	h(polyhymnia) = (8+6+2)%9 = 7
3	4	2	h(urania) = (0+6+2)%9 = 8.
4	3	4	h(Euterpe) = (6+2+4)%9 = 3
4	3	4	h(Thalia) = (8+2+4)%9 = 0
5	0	4	h(Terpsichore) = (5+4+5)%9 = 5.

So 11 different functi keys generated by using random 'g' values.



HashTable.